# Руководство по архитектуре FreeBSD

Аннотация

Добро пожаловать в Руководство по архитектуре FreeBSD. Это руководство находится *в стадии разработки* и создаётся усилиями многих участников. Многие разделы пока не написаны, а существующие могут требовать обновления. Если вы хотите помочь в работе над этим проектом, напишите на электронную почту списка рассылки Список рассылки Проекта Документации FreeBSD.

Актуальная версия этого документа всегда доступна на официальном веб-сервере FreeBSD. Его также можно загрузить в различных форматах и с разными вариантами сжатия с сервера загрузок FreeBSD или одного из многочисленных зеркал mirror sites.

# Содержание

I: Ядро системы . . . 5
1. Начальная загрузка и инициализация ядра . . . 6
1.1. Обзор . . . 6
1.2. Обзор . . . 6
1.3. BIOS . . . 7
1.4. Главная загрузочная запись (`boot0`) . . . 8
1.5. Этап `boot1` . . . 15
1.6. Сервер BTX . . . 20
1.7. Этап загрузки boot2 . . . 28
1.8. Этап загрузчика (loader) . . . 29
1.9. Инициализация ядра . . . 30
2. Заметки о блокировках . . . 40
2.1. Mutexes . . . 40
2.2. Разделяемые эксклюзивные блокировки . . . 41
2.3. Атомарно защищённые переменные . . . 42
3. Объекты ядра . . . 43
3.1. Терминология . . . 43
3.2. Как работает Kobj . . . 43
3.3. Использование Kobj . . . 44
4. Подсистема клеток . . . 48
4.1. Архитектура . . . 48
4.2. Ограничения . . . 54
5. Фреймворк SYSINIT . . . 60
5.1. Терминология . . . 60
5.2. Работа механизма SYSINIT . . . 60
5.3. Использование SYSINIT . . . 61
6. Фреймворк TrustedBSD MAC . . . 63
6.1. Авторские права документации MAC . . . 63
6.2. Обзор . . . 63
6.3. Введение . . . 64
6.4. Общие сведения о политиках . . . 64
6.5. Архитектура MAC Framework в ядре . . . 65
6.6. Архитектура политик MAC . . . 70
6.7. Справочник по точкам входа политики MAC . . . 73
6.8. Пользовательская архитектура . . . 131
6.9. Заключение . . . 132
7. Система управления виртуальной памятью . . . 134
7.1. Управление физической памятью `vm_page_t` . . . 134

7.2. Унифицированный буферный кэш `vm_object_t` . . . 135
7.3. Ввод-вывод файловой системы `struct buf` . . . 135
7.4. Отображение таблиц страниц `vm_map_t, vm_entry_t` . . . 136
7.5. Отображение виртуальной памяти ядра . . . 136
7.6. Настройка системы виртуальной памяти во FreeBSD . . . 137
8. Документ по архитектуре SMPng . . . 139
8.1. Введение . . . 139
8.2. Основные инструменты и основы блокировки . . . 139
8.3. Общая архитектура и дизайн . . . 141
8.4. Конкретные стратегии блокировки . . . 145
8.5. Заметки о реализации . . . 149
8.6. Разные темы . . . 152
Глоссарий . . . 153
II: Драйверы устройств . . . 155
9. Написание драйверов устройств для FreeBSD . . . 156
9.1. Введение . . . 156
9.2. Динамический загрузчик модулей ядра - KLD . . . 156
9.3. Символьные устройства . . . 158
9.4. Блочные устройства (удалены) . . . 161
9.5. Драйверы сетевых устройств . . . 162
10. Драйверы устройств ISA . . . 163
10.1. Обзор . . . 163
10.2. Основная информация . . . 163
10.3. Указатель `device_t` . . . 165
10.4. Файл конфигурации и порядок определения и проверки при автоматической настройке . . . 166
10.5. Ресурсы . . . 168
10.6. Отображение памяти шины . . . 171
10.7. DMA . . . 179
10.8. xxx_isa_probe . . . 181
10.9. xxx_isa_attach . . . 188
10.10. xxx_isa_detach . . . 192
10.11. xxx_isa_shutdown . . . 193
10.12. xxx_intr . . . 193
11. Устройства PCI . . . 195
11.1. Обнаружение и подключение . . . 195
11.2. Ресурсы шины . . . 199
12. Контроллеры SCSI с общим методом доступа (CAM) . . . 203
12.1. Обзор . . . 203
12.2. Общая архитектура . . . 203
12.3. Глобальные переменные и Шаблонный код . . . 204

12.4. Конфигурация устройства: xxx_attach . . . 204
12.5. Обработка сообщений CAM: xxx_action . . . 207
12.6. Опрос xxx_poll . . . 225
12.7. Асинхронные события . . . 225
12.8. Прерывания . . . 226
12.9. Сводка ошибок . . . 233
12.10. Обработка таймаутов . . . 234
13. Устройства USB . . . 236
13.1. Введение . . . 236
13.2. Контроллеры хоста . . . 237
13.3. Информация об устройстве USB . . . 239
13.4. Обнаружение устройства и подключение . . . 242
13.5. Информация о протоколе драйверов USB . . . 243
14. Newbus . . . 246
14.1. Драйверы устройств . . . 246
14.2. Обзор Newbus . . . 246
14.3. Newbus API . . . 249
15. Звуковая подсистема . . . 251
15.1. Введение . . . 251
15.2. Файлы . . . 251
15.3. Обнаружение, подключение и т.д. . . . 252
15.4. Интерфейсы . . . 253
16. PC Card . . . 259
16.1. Добавление устройства . . . 259
III: Приложения . . . 265
Приложение A: Библиография . . . 266

# Часть I: Ядро системы

# Глава 1. Начальная загрузка и инициализация ядра

## 1.1. Обзор

Эта глава представляет собой обзор процессов загрузки и инициализации системы, начиная с POST в BIOS (микропрограмме) и заканчивая созданием первого пользовательского процесса. Поскольку начальные этапы загрузки системы сильно зависят от архитектуры, в качестве примера используется архитектура IA-32. Однако архитектуры AMD64 и ARM64 гораздо важнее и интереснее, и их следует рассмотреть в ближайшем будущем в соответствии с темой этого документа.

Процесс загрузки FreeBSD может быть удивительно сложным. После передачи управления от BIOS необходимо выполнить значительный объём низкоуровневой настройки перед загрузкой и выполнением ядра. Эта настройка должна быть выполнена простым и гибким способом, предоставляя пользователю широкие возможности для настройки и адаптации.

## 1.2. Обзор

Процесс загрузки — это операция, крайне зависимая от оборудования. Не только для каждой архитектуры компьютера должен быть написан код, но также могут существовать различные типы загрузки в рамках одной архитектуры. Например, список файлов в каталоге stand показывает большое количество кода, зависящего от архитектуры. Для каждой из поддерживаемых архитектур существует отдельный каталог. FreeBSD поддерживает стандарт загрузки CSM (Compatibility Support Module). Таким образом, CSM поддерживается (как с GPT, так и с MBR разметкой), а также загрузка через UEFI (GPT полностью поддерживается, MBR — в основном). Также поддерживается загрузка файлов с ext2fs, MSDOS, UFS и ZFS. FreeBSD поддерживает функцию загрузочного окружения ZFS, которая позволяет основной ОС передавать детали о том, что загружать, выходящие за рамки простого раздела, как это было возможно ранее. Однако в наши дни UEFI более актуален, чем CSM. В следующем примере показана загрузка компьютера x86 с жёсткого диска с MBR-разметкой, где используется мультизагрузчик FreeBSD boot0, сохранённый в самом первом секторе. Этот загрузочный код запускает трёхэтапный процесс загрузки FreeBSD.

Ключ к пониманию этого процесса заключается в том, что он состоит из последовательных стадий возрастающей сложности. Эти стадии — boot1, boot2 и loader (подробнее см. boot(8)). Система загрузки выполняет каждую стадию последовательно. Последняя стадия, loader, отвечает за загрузку ядра FreeBSD. Каждая стадия рассматривается в следующих разделах.

Вот пример вывода, сгенерированного на различных этапах загрузки. Фактический вывод может отличаться в зависимости от машины:

| Компонент FreeBSD | Вывод (может отличаться) |
|---|---|

| | |
|---|---|
| boot0 | F1 FreeBSD<br>F2 BSD<br>F5 Disk 2 |
| boot2 [1] | >>FreeBSD/x86 BOOT<br>Default: 0:ad(0p4)/boot/loader<br>boot: |
| loader | BTX loader 1.00 BTX version is 1.02<br>Consoles: internal video/keyboard<br>BIOS drive C: is disk0<br>BIOS 639kB/2096064kB available memory<br><br>FreeBSD/x86 bootstrap loader, Revision 1.1<br>Console internal video/keyboard<br>(root@releng1.nyi.freebsd.org, Fri Apr 9 04:04:45 UTC 2021)<br>Loading /boot/defaults/loader.conf<br>/boot/kernel/kernel text=0xed9008 data=0x117d28+0x176650 syms<br>=[0x8+0x137988+0x8+0x1515f8] |
| ядро системы | Copyright (c) 1992-2021 The FreeBSD Project.<br>Copyright (c) 1979, 1980, 1983, 1986, 1988, 1989, 1991, 1992, 1993,<br>1994<br>The Regents of the University of California. All rights<br>reserved.<br>FreeBSD is a registered trademark of The FreeBSD Foundation.<br>FreeBSD 13.0-RELEASE 0 releng/13.0-n244733-ea31abc261f: Fri Apr 9<br>04:04:45 UTC 2021<br><br>root@releng1.nyi.freebsd.org:/usr/obj/usr/src/i386.i386/sys/GENERIC<br>i386<br>FreeBSD clang version 11.0.1 (git@github.com:llvm/llvm-project.git<br>llvmorg-11.0.1-0-g43ff75f2c3fe) |

## 1.3. BIOS

При включении компьютера регистры процессора устанавливаются в некоторые предопределённые значения. Один из регистров — это регистр *указателя команд*, и его значение после включения питания чётко определено: это 32-битное значение `0xfffffff0`. Регистр указателя команд (также известный как Счётчик Команд) указывает на код, который должен быть выполнен процессором. Ещё один важный регистр — это 32-битный управляющий регистр `cr0`, и его значение сразу после перезагрузки равно `0`. Один из битов `cr0`, бит PE (Protection Enabled, Защита Включена), указывает, работает ли процессор в 32-битном защищённом режиме или 16-битном реальном режиме. Поскольку этот бит сброшен при загрузке, процессор запускается в 16-битном реальном режиме. Реальный

режим означает, среди прочего, что линейные и физические адреса идентичны. Причина, по которой процессор не запускается сразу в 32-битном защищённом режиме, — это обратная совместимость. В частности, процесс загрузки зависит от услуг, предоставляемых BIOS, а сам BIOS работает в устаревшем 16-битном коде.

Значение `0xffffff0` немного меньше 4 ГБ, поэтому, если в машине нет 4 ГБ физической памяти, оно не может указывать на действительный адрес памяти. Аппаратное обеспечение компьютера преобразует этот адрес так, чтобы он указывал на блок памяти BIOS.

BIOS (Basic Input Output System) — это микросхема на материнской плате, которая содержит относительно небольшой объём памяти только для чтения (ROM). Эта память включает различные низкоуровневые процедуры, специфичные для оборудования, поставляемого с материнской платой. Процессор сначала переходит по адресу 0xffffff0, который фактически находится в памяти BIOS. Обычно по этому адресу содержится инструкция перехода к процедурам POST BIOS.

POST (Power On Self Test) — это набор процедур, включающих проверку памяти, проверку системной шины и другую низкоуровневую инициализацию, чтобы процессор мог правильно настроить компьютер. Важным этапом на этой стадии является определение загрузочного устройства. Современные реализации BIOS позволяют выбирать загрузочное устройство, обеспечивая загрузку с дискеты, CD-ROM, жёсткого диска или других устройств.

Самым последним действием в POST является инструкция `INT 0x19`. Обработчик `INT 0x19` считывает 512 байт из первого сектора загрузочного устройства в память по адресу `0x7c00`. Термин *первый сектор* происходит из архитектуры жёстких дисков, где магнитная пластина разделена на множество цилиндрических дорожек. Дорожки нумеруются, и каждая дорожка разделена на несколько (обычно 64) секторов. Нумерация дорожек начинается с 0, но нумерация секторов начинается с 1. Дорожка 0 находится на внешней стороне магнитной пластины, а сектор 1, первый сектор, имеет особое назначение. Он также называется MBR (Master Boot Record) или Главная Загрузочная Запись. Остальные секторы на первой дорожке не используются.

Этот сектор является нашей точкой входа в последовательность загрузки. Как мы увидим, этот сектор содержит копию нашей программы boot0. BIOS выполняет переход по адресу `0x7c00`, и она начинает выполняться.

## 1.4. Главная загрузочная запись (`boot0`)

После получения управления от BIOS по адресу памяти `0x7c00` начинает выполняться boot0. Это первый код, который управляется FreeBSD. Задача boot0 довольно проста: просканировать таблицу разделов и позволить пользователю выбрать, с какого раздела загружаться. Таблица разделов — это специальная стандартная структура данных, встроенная в MBR (а значит, и в boot0), которая описывает четыре стандартных PC-раздела. boot0 находится в файловой системе как /boot/boot0. Это небольшой файл размером 512 байт, и именно его процедура установки FreeBSD записывает в MBR жёсткого диска, если во время установки была выбрана опция "bootmanager". Действительно, boot0 *и есть* MBR.

Как упоминалось ранее, мы вызываем прерывание BIOS `INT 0x19` для загрузки MBR (boot0) в

память по адресу `0x7c00`. Исходный файл для boot0 можно найти в stand/i386/boot0/boot0.S — это впечатляющий фрагмент кода, написанный Робертом Нордье.

Особая структура, начинающаяся со смещения `0x1be` в MBR, называется *таблицей разделов*. Она содержит четыре записи по 16 байт каждая, называемые *записями разделов*, которые определяют, как разделён жёсткий диск, или, в терминологии FreeBSD, нарезан. Один из этих 16 байт указывает, является ли раздел (срез) загрузочным или нет. Ровно одна запись должна быть с этом установленным флагом, иначе код boot0 откажется продолжать работу.

Запись о разделе содержит следующие поля:

- 1-байтовый тип файловой системы
- 1-байтовый флаг загрузки (`bootable`)
- 6-байтовый дескриптор в формате CHS
- 8-байтовый дескриптор в формате LBA

Дескриптор записи раздела содержит информацию о том, где именно раздел расположен на диске. Оба дескриптора, LBA и CHS, описывают одну и ту же информацию, но разными способами: LBA (Logical Block Addressing) содержит начальный сектор раздела и его длину, тогда как CHS (Cylinder Head Sector) содержит координаты первого и последнего секторов раздела. Таблица разделов завершается специальной сигнатурой `0xaa55`.

MBR должен помещаться в 512 байт, один сектор диска. Эта программа использует низкоуровневые «трюки», такие как использование побочных эффектов определённых инструкций и повторное использование значений регистров из предыдущих операций, чтобы максимально эффективно использовать минимально возможное количество инструкций. Также необходимо соблюдать осторожность при работе с таблицей разделов, которая встроена в сам MBR. По этим причинам будьте очень внимательны при изменении boot0.S.

Обратите внимание, что исходный файл boot0.S ассемблируется "как есть": инструкции переводятся одна за одной в бинарный код без дополнительной информации (например, без формата файла ELF). Такой низкоуровневый контроль достигается на этапе компоновки с помощью специальных флагов, передаваемых компоновщику. Например, текстовая секция программы располагается по адресу `0x600`. На практике это означает, что boot0 должен быть загружен в память по адресу `0x600` для корректной работы.

Стоит взглянуть на Makefile для boot0 (stand/i386/boot0/Makefile), так как он определяет некоторые аспекты поведения boot0 во время выполнения. Например, если для ввода-вывода используется терминал, подключённый к последовательному порту (COM1), необходимо определить макрос `SIO` (`-DSIO`). `-DPXE` включает загрузку через PXE при нажатии `F6`. Кроме того, программа определяет набор *флагов*, которые позволяют дополнительно настроить её поведение. Всё это проиллюстрировано в Makefile. Например, обратите внимание на директивы компоновщика, которые предписывают ему начинать секцию текста с адреса `0x600` и создавать выходной файл "как есть" (удаляя любое форматирование файла):

```
	BOOT_BOOT0_ORG?=0x600
```

```
	ORG=${BOOT_BOOT0_ORG}
```

*stand/i386/boot0/Makefile*

Приступим к изучению MBR, или boot0, начиная с точки входа.



В некоторые инструкции были внесены изменения для лучшего изложения. Например, некоторые макросы раскрыты, а некоторые проверки макросов опущены, когда результат проверки известен. Это относится ко всем приведённым примерам кода.

```
start:
	cld		# String ops inc
	xorw %ax,%ax	# Zero
	movw %ax,%es	# Address
	movw %ax,%ds	#  data
	movw %ax,%ss	# Set up
	movw $LOAD,%sp		#  stack
```

*stand/i386/boot0/boot0.S*

Этот первый блок кода является точкой входа программы. Именно сюда BIOS передаёт управление. Сначала он гарантирует, что строковые операции автоматически увеличивают указатели операндов (инструкция `cld`) [2]. Затем, не делая предположений о состоянии сегментных регистров, он их инициализирует. Наконец, он устанавливает регистр указателя стека (`%sp`) в ($LOAD = адрес `0x7c00`), чтобы обеспечить работоспособный стек.

Следующий блок отвечает за перемещение и последующий переход к перемещенному коду.

```
	movw %sp,%si	# Source
	movw $start,%di		# Destination
	movw $0x100,%cx		# Word count
	rep		# Relocate
	movsw		#  code
	movw %di,%bp	 # Address variables
	movb $0x8,%cl	 # Words to clear
	rep		# Zero
	stosw		#  them
	incb -0xe(%di)		# Set the S field to 1
	jmp main-LOAD+ORIGIN  # Jump to relocated code
```

*stand/i386/boot0/boot0.S*

Так как boot0 загружается BIOS по адресу `0x7C00`, он копирует себя по адресу `0x600` и передаёт управление туда (напомним, что он был слинкован для выполнения по адресу `0x600`). Исходный адрес, `0x7c00`, копируется в регистр `%si`. Конечный адрес, `0x600`, — в регистр `%di`. Количество слов для копирования, `256` (размер программы = 512 байт), копируется в регистр `%cx`. Далее инструкция `rep` повторяет следующую за ней инструкцию, то есть `movsw`, количество раз, указанное в регистре `%cx`. Инструкция `movsw` копирует слово, на которое

указывает %si, по адресу, на который указывает %di. Это повторяется ещё 255 раз. При каждом повторении оба регистра, исходный и конечный, %si и %di, увеличиваются на единицу. Таким образом, по завершении копирования 256 слов (512 байт), %di имеет значение 0x600`512`= `0x800`, а `%si` — значение `0x7c00`512= 0x7e00; таким образом, мы завершили *перемещение* кода. С момента последнего обновления этого документа инструкции копирования в коде изменились, поэтому вместо movsb и stosb были введены movsw и stosw, которые копируют 2 байта (1 слово) за одну итерацию.

Затем регистр назначения %di копируется в %bp. %bp получает значение 0x800. Значение 8 копируется в %cl для подготовки новой строковой операции (как в предыдущей movsw). Теперь stosw выполняется 8 раз. Эта инструкция копирует значение 0 по адресу, на который указывает регистр назначения (%di, то есть 0x800), и увеличивает его. Это повторяется ещё 7 раз, так что %di в итоге получает значение 0x810. Фактически это очищает диапазон адресов 0x800-0x80f. Этот диапазон используется как (фиктивная) таблица разделов для записи MBR обратно на диск. Наконец, полю сектора для CHS-адресации этого фиктивного раздела присваивается значение 1, и выполняется переход к основной функции из перемещённого кода. Обратите внимание, что до этого перехода к перемещённому коду любые ссылки на абсолютные адреса избегались.

Следующий блок кода проверяет, следует ли использовать номер диска, предоставленный BIOS, или тот, что хранится в boot0.

```
main:
      testb $SETDRV,_FLAGS(%bp) # Set drive number?
#ifndef CHECK_DRIVE /* disable drive checks */
      jz save_curdrive      # no, use the default
#else
      jnz disable_update    # Yes
      testb %dl,%dl     # Drive number valid?
      js save_curdrive      # Possibly (0x80 set)
#endif
```

*stand/i386/boot0/boot0.S*

Этот код проверяет бит SETDRV (0x20) в переменной *flags*. Напомним, что регистр %bp указывает на адрес 0x800, поэтому проверка выполняется для переменной *flags* по адресу 0x800-69= 0x7bb. Это пример типа изменений, которые можно внести в boot0. Флаг SETDRV не установлен по умолчанию, но его можно задать в Makefile. Если он установлен, используется номер диска, сохранённый в MBR, вместо предоставленного BIOS. Мы предполагаем значения по умолчанию и то, что BIOS предоставил корректный номер диска, поэтому переходим к save_curdrive.

Следующий блок сохраняет номер диска, предоставленный BIOS, и вызывает putn для вывода новой строки на экран.

```
save_curdrive:
      movb %dl, (%bp)       # Save drive number
      pushw %dx         # Also in the stack
#ifdef  TEST    /* test code, print internal bios drive */
```

```
	rolb $1, %dl
	movw $drive, %si
	call putkey
#endif
	callw putn		# Print a newline
```

*stand/i386/boot0/boot0.S*

Обратите внимание, что мы предполагаем, что `TEST` не определён, поэтому условный код в нём не собирается и не появится в нашем исполняемом файле boot0.

Следующий блок реализует фактическое сканирование таблицы разделов. Он выводит на экран тип раздела для каждой из четырёх записей в таблице разделов. Каждый тип сравнивается со списком известных файловых систем операционных систем. Примерами распознаваемых типов разделов являются NTFS (Windows®, ID 0x7), `ext2fs` (Linux®, ID 0x83) и, конечно же, `ffs`/`ufs2` (FreeBSD, ID 0xa5). Реализация довольно проста.

```
	movw $(partbl+0x4),%bx	# Partition table (+4)
	xorw %dx,%dx		# Item number

read_entry:
	movb %ch,-0x4(%bx)	# Zero active flag (ch == 0)
	btw %dx,_FLAGS(%bp)	# Entry enabled?
	jnc next_entry		# No
	movb (%bx),%al		# Load type
	test %al, %al		# skip empty partition
	jz next_entry
	movw $bootable_ids,%di	# Lookup tables
	movb $(TLEN+1),%cl	# Number of entries
	repne			# Locate
	scasb			#  type
	addw $(TLEN-1), %di	# Adjust
	movb (%di),%cl		# Partition
	addw %cx,%di		#  description
	callw putx		# Display it

next_entry:
	incw %dx		# Next item
	addb $0x10,%bl		# Next entry
	jnc read_entry		# Till done
```

*stand/i386/boot0/boot0.S*

Важно отметить, что флаг активности для каждой записи сбрасывается, поэтому после сканирования *ни одна* запись о разделе не активна в нашей копии boot0 в памяти. Позже флаг активности будет установлен для выбранного раздела. Это гарантирует, что только один активный раздел существует, если пользователь решит записать изменения обратно на диск.

Следующий блок проверяет наличие других дисков. При запуске BIOS записывает

количество дисков, присутствующих в компьютере, по адресу 0x475. Если есть другие диски, boot0 выводит текущий диск на экран. Пользователь может позже дать команду boot0 просканировать разделы на другом диске.

```
	popw %ax		# Drive number
	subb $0x80-0x1,%al		# Does next
	cmpb NHRDRV,%al		#  drive exist? (from BIOS?)
	jb print_drive		# Yes
	decw %ax		# Already drive 0?
	jz print_prompt		# Yes
```

*stand/i386/boot0/boot0.S*

Мы предполагаем, что присутствует только один диск, поэтому переход к print_drive не выполняется. Также мы предполагаем, что ничего необычного не произошло, поэтому переходим к print_prompt.

Следующий блок просто выводит приглашение с последующим вариантом по умолчанию:

```
print_prompt:
	movw $prompt,%si	# Display
	callw putstr	#  prompt
	movb _OPT(%bp),%dl	# Display
	decw %si		#  default
	callw putkey	#  key
	jmp start_input		# Skip beep
```

*stand/i386/boot0/boot0.S*

Наконец, выполняется переход к start_input, где используются сервисы BIOS для запуска таймера и чтения пользовательского ввода с клавиатуры; если таймер истекает, будет выбран вариант по умолчанию:

```
start_input:
	xorb %ah,%ah	# BIOS: Get
	int $0x1a		#  system time
	movw %dx,%di	# Ticks when
	addw _TICKS(%bp),%di	#  timeout
read_key:
	movb $0x1,%ah	# BIOS: Check
	int $0x16		#  for keypress
	jnz got_key		# Have input
	xorb %ah,%ah	# BIOS: int 0x1a, 00
	int $0x1a		#  get system time
	cmpw %di,%dx	# Timeout?
	jb read_key		# No
```

*stand/i386/boot0/boot0.S*

Прерывание запрашивается с номером `0x1a` и аргументом `0` в регистре `%ah`. BIOS имеет предопределённый набор сервисов, запрашиваемых приложениями как программно-генерируемые прерывания через инструкцию `int`, с получением аргументов в регистрах (в данном случае, `%ah`). Здесь, в частности, запрашивается количество тиков часов с момента последней полуночи; это значение вычисляется BIOS через RTC (Real Time Clock). Эти часы могут быть настроены на работу с частотой от 2 Гц до 8192 Гц. BIOS устанавливает их на 18,2 Гц при запуске. Когда запрос выполнен, 32-битный результат возвращается BIOS в регистрах `%cx` и `%dx` (младшие байты в `%dx`). Этот результат (часть `%dx`) копируется в регистр `%di`, и к `%di` добавляется значение переменной `TICKS`. Эта переменная находится в boot0 по смещению `_TICKS` (отрицательное значение) от регистра `%bp` (который, напомним, указывает на `0x800`). Значение этой переменной по умолчанию — `0xb6` (182 в десятичной системе). Идея заключается в том, что boot0 постоянно запрашивает время у BIOS, и когда значение, возвращённое в регистре `%dx`, становится больше значения, хранящегося в `%di`, время истекает и будет сделан выбор по умолчанию. Поскольку RTC тикает 18,2 раза в секунду, это условие выполнится через 10 секунд (это поведение по умолчанию можно изменить в Makefile). До истечения этого времени boot0 непрерывно опрашивает BIOS на предмет ввода пользователя; это делается через `int 0x16`, аргумент `1` в `%ah`.

Была нажата клавиша или истекло время, последующий код проверяет выбор. В зависимости от выбора, регистр `%si` устанавливается так, чтобы указывать на соответствующую запись раздела в таблице разделов. Этот новый выбор переопределяет предыдущий выбор по умолчанию. Действительно, он становится новым значением по умолчанию. Наконец, устанавливается флаг ACTIVE выбранного раздела. Если это было разрешено при компиляции, версия boot0 в памяти с этими изменёнными значениями записывается обратно в MBR на диске. Мы оставляем детали этой реализации читателю.

Мы завершаем наше изучение последним блоком кода из программы boot0:

```
        movw $LOAD,%bx          # Address for read
        movb $0x2,%ah       # Read sector
        callw intx13        #  from disk
        jc beep             # If error
        cmpw $MAGIC,0x1fe(%bx)    # Bootable?
        jne beep            # No
        pushw %si           # Save ptr to selected part.
        callw putn          # Leave some space
        popw %si            # Restore, next stage uses it
        jmp *%bx            # Invoke bootstrap
```

*stand/i386/boot0/boot0.S*

Вспомним, что `%si` указывает на выбранную запись раздела. Эта запись сообщает нам, где начинается раздел на диске. Мы предполагаем, конечно, что выбранный раздел действительно является срезом FreeBSD.



Отныне мы будем отдавать предпочтение использованию технически более точного термина "слайс" вместо "раздел".

Буфер передачи установлен в `0x7c00` (регистр `%bx`), и запрос на чтение первого сектора слайса

FreeBSD выполняется вызовом `intx13`. Мы предполагаем, что всё прошло успешно, поэтому переход к `beep` не выполняется. В частности, новый прочитанный сектор должен заканчиваться магической последовательностью `0xaa55`. Наконец, значение в `%si` (указатель на выбранную таблицу разделов) сохраняется для использования на следующем этапе, и выполняется переход по адресу `0x7c00`, где начинается выполнение нашего следующего этапа (только что прочитанного блока).

## 1.5. Этап `boot1`

До сих пор мы прошли следующую последовательность:

- BIOS выполнил первоначальную инициализацию оборудования, включая POST. MBR (boot0) был загружен по адресу `0x7c00` из абсолютного сектора один с диска. Управление выполнением было передано по этому адресу.
- boot0 переместил себя по адресу, по которому он был скомпонован для выполнения (`0x600`), после чего выполнил переход для продолжения выполнения в соответствующем месте. В завершение, boot0 загрузил первый сектор диска из раздела FreeBSD по адресу `0x7c00`. Управление выполнением было передано по этому адресу.

boot1 — это следующий шаг в последовательности загрузки. Это первая из трёх стадий загрузки. Обратите внимание, что до сих пор мы работали исключительно с секторами диска. Действительно, BIOS загружает самый первый сектор, а boot0 загружает первый сектор раздела FreeBSD. Обе загрузки происходят по адресу `0x7c00`. Мы можем концептуально представлять эти секторы диска как содержащие файлы boot0 и boot1, соответственно, но на самом деле это не совсем верно для boot1. Строго говоря, в отличие от boot0, boot1 не является частью загрузочных блоков [3]. Вместо этого, единый полноценный файл boot (/boot/boot) в итоге записывается на диск. Этот файл представляет собой комбинацию boot1, boot2 и `Boot Extender` (или BTX). Этот единый файл превышает размер одного сектора (больше 512 байт). К счастью, boot1 занимает *ровно* первые 512 байт этого файла, поэтому, когда boot0 загружает первый сектор раздела FreeBSD (512 байт), он фактически загружает boot1 и передаёт ему управление.

Основная задача boot1 — загрузить следующий этап загрузки. Этот следующий этап несколько сложнее. Он состоит из сервера под названием "Boot Extender" (BTX) и клиента под названием boot2. Как мы увидим, последний этап загрузки, loader, также является клиентом сервера BTX.

Давайте теперь подробно рассмотрим, что именно делает boot1, начиная, как мы это делали для boot0, с точки входа:

```
start:
    jmp main
```

*stand/i386/boot2/boot1.S*

Точка входа `start` просто переходит через специальную область данных к метке `main`, которая, в свою очередь, выглядит следующим образом:

```
main:
      cld           # String ops inc
      xor %cx,%cx       # Zero
      mov %cx,%es       # Address
      mov %cx,%ds       #  data
      mov %cx,%ss       # Set up
      mov $start,%sp        #  stack
      mov %sp,%si       # Source
      mov $MEM_REL,%di      # Destination
      incb %ch          # Word count
      rep           # Copy
      movsw         #  code
```

*stand/i386/boot2/boot1.S*

Как и boot0, этот код перемещает boot1, на этот раз по адресу `0x700`. Однако, в отличие от boot0, он не переходит туда. boot1 скомпонован для выполнения по адресу `0x7c00`, фактически там, куда он был изначально загружен. Причина этого перемещения будет рассмотрена далее.

Далее идёт цикл, который ищет слайс FreeBSD. Хотя boot0 загрузил boot1 из слайса FreeBSD, ему не была передана информация об этом [4], поэтому boot1 должен повторно просканировать таблицу разделов, чтобы найти начало слайса FreeBSD. Для этого он перечитывает MBR:

```
      mov $part4,%si        # Partition
      cmpb $0x80,%dl        # Hard drive?
      jb main.4         # No
      movb $0x1,%dh     # Block count
      callw nread       # Read MBR
```

*stand/i386/boot2/boot1.S*

В приведённом выше коде регистр `%dl` содержит информацию о загрузочном устройстве. Эти данные передаются BIOS и сохраняются MBR. Числа `0x80` и выше указывают на то, что мы имеем дело с жёстким диском, поэтому вызывается `nread`, где считывается MBR. Аргументы для `nread` передаются через `%si` и `%dh`. Адрес памяти по метке `part4` копируется в `%si`. Этот адрес памяти содержит "фальшивый раздел", который будет использован `nread`. Ниже приведены данные фальшивого раздела:

```
      part4:
    .byte 0x80, 0x00, 0x01, 0x00
    .byte 0xa5, 0xfe, 0xff, 0xff
    .byte 0x00, 0x00, 0x00, 0x00
    .byte 0x50, 0xc3, 0x00, 0x00
```

*stand/i386/boot2/boot1.S*

В частности, LBA для этой фиктивной раздела жёстко закодирован как ноль. Это

используется как аргумент для BIOS при чтении абсолютного сектора один с жёсткого диска. Или же может использоваться адресация CHS. В этом случае фиктивный раздел содержит цилиндр 0, головку 0 и сектор 1, что эквивалентно абсолютному сектору один.

Продолжим, рассмотрев `nread`:

```
nread:
      mov $MEM_BUF,%bx      # Transfer buffer
      mov 0x8(%si),%ax      # Get
      mov 0xa(%si),%cx      #  LBA
      push %cs           # Read from
      callw xread.1      #  disk
      jnc return         # If success, return
```

*stand/i386/boot2/boot1.S*

Напомним, что `%si` указывает на поддельный раздел. Слово [5] по смещению `0x8` копируется в регистр `%ax`, а слово по смещению `0xa` — в `%cx`. BIOS интерпретирует их как младшее 4-байтовое значение, обозначающее LBA для чтения (старшие четыре байта предполагаются нулевыми). Регистр `%bx` содержит адрес памяти, куда будет загружен MBR. Инструкция, помещающая `%cs` в стек, очень интересна. В данном контексте она ничего не делает. Однако, как мы скоро увидим, boot2 в сочетании с сервером BTX также использует `xread.1`. Этот механизм будет рассмотрен в следующем разделе.

Код в `xread.1` далее вызывает функцию `read`, которая фактически обращается к BIOS с запросом на чтение сектора диска:

```
xread.1:
    pushl $0x0      #  absolute
    push %cx        #  block
    push %ax        #  number
    push %es        # Address of
    push %bx        #  transfer buffer
    xor %ax,%ax     # Number of
    movb %dh,%al        #  blocks to
    push %ax        #  transfer
    push $0x10      # Size of packet
    mov %sp,%bp     # Packet pointer
    callw read      # Read from disk
    lea 0x10(%bp),%sp   # Clear stack
    lret            # To far caller
```

*stand/i386/boot2/boot1.S*

Обратите внимание на длинную инструкцию возврата в конце этого блока. Эта инструкция извлекает регистр `%cs`, помещённый в стек `nread`, и возвращает управление. В конце `nread` также возвращает управление.

С загрузкой MBR в память начинается фактический цикл поиска слайса FreeBSD:

```
    mov $0x1,%cx         # Two passes
main.1:
    mov $MEM_BUF+PRT_OFF,%si # Partition table
    movb $0x1,%dh        # Partition
main.2:
    cmpb $PRT_BSD,0x4(%si)   # Our partition type?
    jne main.3       # No
    jcxz main.5      # If second pass
    testb $0x80,(%si)    # Active?
    jnz main.5       # Yes
main.3:
    add $0x10,%si        # Next entry
    incb %dh         # Partition
    cmpb $0x1+PRT_NUM,%dh        # In table?
    jb main.2        # Yes
    dec %cx          # Do two
    jcxz main.1      #  passes
```

*stand/i386/boot2/boot1.S*

Если обнаружен слайс FreeBSD, выполнение продолжается на метке `main.5`. Обратите внимание, что при обнаружении слайса FreeBSD `%si` указывает на соответствующую запись в таблице разделов, а `%dh` содержит номер раздела. Мы предполагаем, что слайс FreeBSD найден, поэтому продолжаем выполнение на метке `main.5`:

```
main.5:
    mov %dx,MEM_ARG            # Save args
    movb $NSECT,%dh            # Sector count
    callw nread            # Read disk
    mov $MEM_BTX,%bx               # BTX
    mov 0xa(%bx),%si           # Get BTX length and set
    add %bx,%si            #  %si to start of boot2.bin
    mov $MEM_USR+SIZ_PAG*2,%di             # Client page 2
    mov $MEM_BTX+(NSECT-1)*SIZ_SEC,%cx             # Byte
    sub %si,%cx            #  count
    rep                # Relocate
    movsb                  #  client
```

*stand/i386/boot2/boot1.S*

Напомним, что в данный момент регистр `%si` указывает на запись среза FreeBSD в таблице разделов MBR, поэтому вызов `nread` фактически прочитает секторы в начале этого раздела. Аргумент, переданный в регистре `%dh`, указывает `nread` прочитать 16 секторов диска. Напомним, что первые 512 байт, или первый сектор слайса FreeBSD, совпадает с программой boot1. Также напомним, что файл, записанный в начало слайса FreeBSD, это не /boot/boot1, а /boot/boot. Давайте посмотрим на размер этих файлов в файловой системе:

```
-r--r--r--  1 root  wheel   512B Jan  8 00:15 /boot/boot0
-r--r--r--  1 root  wheel   512B Jan  8 00:15 /boot/boot1
```

```
-r--r--r--  1 root  wheel   7.5K Jan  8 00:15 /boot/boot2
-r--r--r--  1 root  wheel   8.0K Jan  8 00:15 /boot/boot
```

Оба файла boot0 и boot1 имеют размер 512 байт каждый, поэтому они занимают *ровно* один сектор диска. boot2 значительно больше, так как содержит как сервер BTX, так и клиент boot2. Наконец, файл под названием просто boot на 512 байт больше, чем boot2. Этот файл представляет собой объединение boot1 и boot2. Как уже отмечалось, boot0 записывается в самый первый сектор диска (MBR), а boot записывается в первый сектор раздела FreeBSD; boot1 и boot2 *не* записываются на диск. Команда, используемая для объединения boot1 и boot2 в единый файл boot, выглядит просто как `cat boot1 boot2 > boot`.

Итак, boot1 занимает ровно первые 512 байт boot, и, поскольку boot записывается в первый сектор слайса FreeBSD, boot1 полностью помещается в этот первый сектор. Когда `nread` читает первые 16 секторов слайса FreeBSD, он фактически читает весь файл boot [6]. Более подробно о том, как boot формируется из boot1 и boot2, мы увидим в следующем разделе.

Напомним, что `nread` использует адрес памяти `0x8c00` в качестве буфера передачи для хранения прочитанных секторов. Этот адрес выбран не случайно. Действительно, поскольку boot1 принадлежит первым 512 байтам, он оказывается в диапазоне адресов `0x8c00-0x8dff`. Следующие 512 байт (диапазон `0x8e00-0x8fff`) используются для хранения *bsdlabel* [7].

Начиная с адреса `0x9000` находится начало сервера BTX, и сразу за ним следует клиент boot2. Сервер BTX действует как ядро и выполняется в защищённом режиме с наивысшим уровнем привилегий. В отличие от этого, клиенты BTX (например, boot2) выполняются в пользовательском режиме. Мы увидим, как это реализовано, в следующем разделе. Код после вызова `nread` находит начало boot2 в буфере памяти и копирует его по адресу `0xc000`. Это связано с тем, что сервер BTX размещает boot2 для выполнения в сегменте, начинающемся с `0xa000`. Мы подробно рассмотрим это в следующем разделе.

Последний блок кода в boot1 разрешает доступ к памяти выше 1MB [8] и завершается переходом к начальной точке сервера BTX:

```
seta20:
    cli         # Disable interrupts
seta20.1:
    dec %cx         # Timeout?
    jz seta20.3     # Yes

    inb $0x64,%al       # Get status
    testb $0x2,%al      # Busy?
    jnz seta20.1        # Yes
    movb $0xd1,%al      # Command: Write
    outb %al,$0x64      #  output port
seta20.2:
    inb $0x64,%al       # Get status
    testb $0x2,%al      # Busy?
    jnz seta20.2        # Yes
    movb $0xdf,%al      # Enable
```

```
    outb %al,$0x60      #  A20
seta20.3:
    sti         # Enable interrupts
    jmp 0x9010      # Start BTX
```

*stand/i386/boot2/boot1.S*

Обратите внимание, что непосредственно перед переходом прерывания включаются.

# 1.6. Сервер BTX

Далее в нашей последовательности загрузки идёт сервер BTX. Давайте быстро вспомним, как мы сюда попали:

- BIOS загружает абсолютный сектор один (MBR или boot0) по адресу `0x7c00` и переходит туда.
- boot0 перемещает себя по адресу `0x600`, по которому он был слинкован для выполнения, и переходит туда. Затем он читает первый сектор среза FreeBSD (который содержит boot1) в адрес `0x7c00` и переходит туда.
- boot1 загружает первые 16 секторов среза FreeBSD по адресу `0x8c00`. Эти 16 секторов, или 8192 байта, представляют собой весь файл boot. Файл является объединением boot1 и boot2. boot2, в свою очередь, содержит сервер BTX и клиент boot2. Наконец, выполняется переход по адресу `0x9010`, точке входа сервера BTX.

Прежде чем изучать сервер BTX подробно, давайте рассмотрим, как создаётся единый, всеобъемлющий файл boot. Способ сборки boot определен в его Makefile (stand/i386/boot2/Makefile). Рассмотрим правило, которое создаёт файл boot:

```
  boot: boot1 boot2
    cat boot1 boot2 > boot
```

*stand/i386/boot2/Makefile*

Это говорит нам, что boot1 и boot2 необходимы, и правило просто объединяет их для создания одного файла с именем boot. Правила для создания boot1 также довольно просты:

```
  boot1: boot1.out
    ${OBJCOPY} -S -O binary boot1.out ${.TARGET}

  boot1.out: boot1.o
    ${LD} ${LD_FLAGS} -e start --defsym ORG=${ORG1} -T ${LDSCRIPT} -o ${.TARGET}
boot1.o
```

*stand/i386/boot2/Makefile*

Для применения правила создания boot1 необходимо собрать boot1.out. Это, в свою очередь, зависит от наличия boot1.o. Последний файл является результатом ассемблирования нашего знакомого boot1.S без компоновки. Теперь применяется правило создания boot1.out.

Оно указывает, что boot1.o должен быть скомпонован с точкой входа `start` и начальным адресом `0x7c00`. Наконец, boot1 создаётся из boot1.out применением соответствующего правила. Это команда objcopy, применяемая к boot1.out. Обратите внимание на флаги, передаваемые objcopy: `-S` указывает на удаление всей информации о перемещении и символов; `-O binary` указывает формат вывода, то есть простой, неформатированный двоичный файл.

Имея boot1, давайте посмотрим, как устроен boot2:

```
  boot2: boot2.ld
@set -- `ls -l ${.ALLSRC}`; x=$$((${BOOT2SIZE}-$$5)); \
    echo "$$x bytes available"; test $$x -ge 0
${DD} if=${.ALLSRC} of=${.TARGET} bs=${BOOT2SIZE} conv=sync

  boot2.ld: boot2.ldr boot2.bin ${BTXKERN}
btxld -v -E ${ORG2} -f bin -b ${BTXKERN} -l boot2.ldr \
    -o ${.TARGET} -P 1 boot2.bin

  boot2.ldr:
${DD} if=/dev/zero of=${.TARGET} bs=512 count=1

  boot2.bin: boot2.out
${OBJCOPY} -S -O binary boot2.out ${.TARGET}

  boot2.out: ${BTXCRT} boot2.o sio.o ashldi3.o
${LD} ${LD_FLAGS} --defsym ORG=${ORG2} -T ${LDSCRIPT} -o ${.TARGET} ${.ALLSRC}

  boot2.h: boot1.out
${NM} -t d ${.ALLSRC} | awk '/([0-9])+ T xread/ \
    { x = $$1 - ORG1; \
    printf("#define XREADORG %#x\n", REL1 + x) }' \
    ORG1=`printf "%d" ${ORG1}` \
    REL1=`printf "%d" ${REL1}` > ${.TARGET}
```

*stand/i386/boot2/Makefile*

Механизм сборки boot2 гораздо сложнее. Отметим наиболее важные моменты. Список зависимостей выглядит следующим образом:

```
  boot2: boot2.ld
  boot2.ld: boot2.ldr boot2.bin ${BTXDIR}
  boot2.bin: boot2.out
  boot2.out: ${BTXDIR} boot2.o sio.o ashldi3.o
  boot2.h: boot1.out
```

*stand/i386/boot2/Makefile*

Отметим, что изначально файл заголовка boot2.h отсутствует, но его создание зависит от boot1.out, который у нас уже есть. Правило его создания немного лаконично, но важно то,

что результат, boot2.h, выглядит примерно так:

```
#define XREADORG 0x725
```

*stand/i386/boot2/boot2.h*

Напомним, что boot1 был перемещён (т.е. скопирован из `0x7c00` в `0x700`). Это перемещение теперь обретает смысл, потому что, как мы увидим, сервер BTX освобождает часть памяти, включая область, куда boot1 был изначально загружен. Однако серверу BTX необходим доступ к функции `xread` из boot1; согласно выводу boot2.h, эта функция находится по адресу `0x725`. Действительно, сервер BTX использует функцию `xread` из перемещённого кода boot1. Теперь эта функция доступна из клиента boot2.

Следующее правило указывает компоновщику на необходимость связать различные файлы (ashldi3.o, boot2.o и sio.o). Обратите внимание, что выходной файл boot2.out компонуется для выполнения по адресу `0x2000` (${ORG2}). Напомним, что boot2 будет выполняться в пользовательском режиме внутри специального пользовательского сегмента, созданного сервером BTX. Этот сегмент начинается с адреса `0xa000`. Также помните, что часть boot2 в boot была скопирована по адресу `0xc000`, то есть со смещением `0x2000` от начала пользовательского сегмента, поэтому boot2 будет работать корректно при передаче управления на него. Далее, boot2.bin создаётся из boot2.out путем удаления символов и информации о формате; boot2.bin представляет собой *сырой* бинарный файл. Теперь обратите внимание, что файл boot2.ldr создаётся как 512-байтный файл, заполненный нулями. Это пространство зарезервировано для bsdlabel.

Теперь, когда у нас есть файлы boot1, boot2.bin и boot2.ldr, осталось только добавить сервер BTX перед созданием универсального файла boot. Сервер BTX находится в stand/i386/btx/btx; у него есть собственный Makefile со своим набором правил для сборки. Важно отметить, что он также компилируется как *сырой* бинарный файл и линкуется для выполнения по адресу `0x9000`. Подробности можно найти в stand/i386/btx/btx/Makefile.

Имея файлы, составляющие программу boot, последним шагом является их *объединение*. Это выполняется специальной программой под названием btxld (исходный код расположен в /usr/src/usr.sbin/btxld). Некоторые аргументы этой программы включают имя выходного файла (boot), его точку входа (`0x2000`) и формат файла (бинарный). Различные файлы окончательно объединяются этой утилитой в файл boot, который состоит из boot1, boot2, `bsdlabel` и сервера BTX. Этот файл, занимающий ровно 16 секторов или 8192 байта, записывается в начало раздела FreeBSD во время установки. Теперь перейдем к изучению программы сервера BTX.

Сервер BTX подготавливает простое окружение и переключается из 16-битного реального режима в 32-битный защищённый режим, непосредственно перед передачей управления клиенту. Это включает инициализацию и обновление следующих структур данных:

- Изменяет `Таблицу Векторов Прерываний (IVT)`. IVT предоставляет обработчики исключений и прерываний для кода в Реальном Режиме.
- Создаётся `Таблица дескрипторов прерываний (IDT)`. В ней предусмотрены записи для исключений процессора, аппаратных прерываний, двух системных вызовов и

интерфейса V86. IDT предоставляет обработчики исключений и прерываний для кода в защищённом режиме.

- Создаётся `Сегмент состояния задачи (TSS)`. Это необходимо, потому что процессор работает на *наименее* привилегированном уровне при выполнении клиента (boot2), но на *наиболее* привилегированном уровне при выполнении сервера BTX.
- Устанавливается GDT (Глобальная Таблица Дескрипторов). Создаются записи (дескрипторы) для кода и данных супервизора, кода и данных пользователя, а также кода и данных реального режима. [9]

Приступим к изучению фактической реализации. Напомним, что boot1 выполнил переход на адрес `0x9010` — точку входа сервера BTX. Прежде чем изучать выполнение программы там, обратите внимание, что сервер BTX имеет специальный заголовок в диапазоне адресов `0x9000-0x900f`, непосредственно перед точкой входа. Этот заголовок определён следующим образом:

```
start:                  # Start of code
/*
 * BTX header.
 */
btx_hdr:    .byte 0xeb          # Machine ID
        .byte 0xe           # Header size
        .ascii "BTX"            # Magic
        .byte 0x1           # Major version
        .byte 0x2           # Minor version
        .byte BTX_FLAGS         # Flags
        .word PAG_CNT-MEM_ORG>>0xc    # Paging control
        .word break-start       # Text size
        .long 0x0           # Entry address
```

*stand/i386/btx/btx/btx.S*

Обратите внимание, что первые два байта — это `0xeb` и `0xe`. В архитектуре IA-32 эти два байта интерпретируются как относительный переход за заголовок к точке входа, поэтому теоретически boot1 мог бы перейти сюда (адрес `0x9000`) вместо адреса `0x9010`. Обратите внимание, что последнее поле в заголовке BTX — это указатель на точку входа клиента (boot2)b2. Это поле исправляется во время компоновки.

Сразу после заголовка следует точка входа сервера BTX:

```
/*
 * Initialization routine.
 */
init:       cli             # Disable interrupts
        xor %ax,%ax         # Zero/segment
        mov %ax,%ss         # Set up
        mov $MEM_ESP0,%sp       #  stack
        mov %ax,%es         # Address
        mov %ax,%ds         #  data
```

```
        pushl $0x2          # Clear
        popfl               #  flags
```

*stand/i386/btx/btx/btx.S*

Этот код отключает прерывания, устанавливает рабочий стек (начиная с адреса `0x1800`) и очищает флаги в регистре EFLAGS. Обратите внимание, что инструкция `popfl` извлекает двойное слово (4 байта) из стека и помещает его в регистр EFLAGS. Поскольку извлекаемое значение фактически равно `2`, регистр EFLAGS эффективно очищается (IA-32 требует, чтобы бит 2 регистра EFLAGS всегда был равен 1).

Следующий блок кода очищает (устанавливает в `0`) диапазон памяти `0x5e00-0x8fff`. В этом диапазоне будут созданы различные структуры данных:

```
/*
 * Initialize memory.
 */
        mov $MEM_IDT,%di        # Memory to initialize
        mov $(MEM_ORG-MEM_IDT)/2,%cx    # Words to zero
        rep             # Zero-fill
        stosw               #  memory
```

*stand/i386/btx/btx/btx.S*

Напомним, что boot1 изначально загружался по адресу `0x7c00`, поэтому при такой инициализации памяти эта копия фактически исчезла. Однако также напомним, что boot1 был перемещён на адрес `0x700`, поэтому *эта* копия всё ещё находится в памяти, и сервер BTX будет её использовать.

Далее обновляется таблица векторов прерываний (IVT) в реальном режиме. IVT представляет собой массив пар сегмент/смещение для обработчиков исключений и прерываний. BIOS обычно сопоставляет аппаратные прерывания с векторами прерываний `0x8`–`0xf` и `0x70`–`0x77`, но, как будет показано, программируемый контроллер прерываний 8259A, микросхема, управляющая фактическим сопоставлением аппаратных прерываний с векторами прерываний, программируется для переназначения этих векторов прерываний с `0x8`–`0xf` на `0x20`–`0x27` и с `0x70`–`0x77` на `0x28`–`0x2f`. Таким образом, обработчики прерываний предоставляются для векторов прерываний `0x20`–`0x2f`. Причина, по которой обработчики, предоставляемые BIOS, не используются напрямую, заключается в том, что они работают в 16-битном реальном режиме, но не в 32-битном защищённом режиме. Вскоре будет выполнен переход в 32-битный защищённый режим. Однако сервер BTX настраивает механизм для эффективного использования обработчиков, предоставляемых BIOS:

```
/*
 * Update real mode IDT for reflecting hardware interrupts.
 */
        mov $intr20,%bx         # Address first handler
        mov $0x10,%cx           # Number of handlers
        mov $0x20*4,%di         # First real mode IDT entry
init.0:     mov %bx,(%di)           # Store IP
```

```
        inc %di             # Address next
        inc %di             #  entry
        stosw               # Store CS
        add $4,%bx          # Next handler
        loop init.0         # Next IRQ
```

*stand/i386/btx/btx/btx.S*

Следующий блок создаёт IDT (таблицу дескрипторов прерываний). IDT в защищённом режиме аналогична IVT в реальном режиме. То есть, IDT описывает различные обработчики исключений и прерываний, используемые, когда процессор работает в защищённом режиме. По сути, она также состоит из массива пар сегмент/смещение, хотя структура несколько сложнее, поскольку сегменты в защищённом режиме отличаются от реального режима, и применяются различные механизмы защиты:

```
/*
 * Create IDT.
 */
        mov $MEM_IDT,%di        # IDT's address
        mov $idtctl,%si         # Control string
init.1:     lodsb               # Get entry
        cbw             #  count
        xchg %ax,%cx            #  as word
        jcxz init.4         # If done
        lodsb               # Get segment
        xchg %ax,%dx            #  P:DPL:type
        lodsw               # Get control
        xchg %ax,%bx            #  set
        lodsw               # Get handler offset
        mov $SEL_SCODE,%dh      # Segment selector
init.2:     shr %bx             # Handle this int?
        jnc init.3          # No
        mov %ax,(%di)           # Set handler offset
        mov %dh,0x2(%di)        #  and selector
        mov %dl,0x5(%di)        # Set P:DPL:type
        add $0x4,%ax            # Next handler
init.3:     lea 0x8(%di),%di        # Next entry
        loop init.2         # Till set done
        jmp init.1          # Continue
```

*stand/i386/btx/btx/btx.S*

Каждая запись в `IDT` имеет длину 8 байт. Помимо информации о сегменте/смещении, они также описывают тип сегмента, уровень привилегий и присутствует ли сегмент в памяти. Структура организована так, что векторы прерываний от `0` до `0xf` (исключения) обрабатываются функцией `intx00`; вектор `0x10` (также исключение) обрабатывается `intx10`; аппаратные прерывания, которые позже настраиваются начиная с вектора `0x20` и до вектора `0x2f`, обрабатываются функцией `intx20`. Наконец, вектор прерывания `0x30`, используемый для системных вызовов, обрабатывается `intx30`, а векторы `0x31` и `0x32` обрабатываются `intx31`. Необходимо отметить, что только дескрипторы для векторов

прерываний `0x30`, `0x31` и `0x32` имеют уровень привилегий 3, такой же, как у клиента boot2, что означает, что клиент может выполнить программно-генерируемое прерывание к этим векторам через инструкцию `int` без ошибки (это способ, которым boot2 использует сервисы, предоставляемые сервером BTX). Также обратите внимание, что *только* программно-генерируемые прерывания защищены от кода, выполняющегося на более низких уровнях привилегий. Аппаратно-генерируемые прерывания и исключения, генерируемые процессором, *всегда* обрабатываются корректно, независимо от фактических привилегий.

Следующий шаг — инициализация TSS (сегмента состояния задачи). TSS — это аппаратная функция, которая помогает операционной системе или исполнительному ПО реализовать многозадачность через абстракцию процессов. Архитектура IA-32 требует создания и использования *как минимум* одного TSS, если используются механизмы многозадачности или определены различные уровни привилегий. Поскольку клиент boot2 выполняется на уровне привилегий 3, а сервер BTX работает на уровне привилегий 0, необходимо определить TSS:

```
/*
 * Initialize TSS.
 */
init.4:     movb $_ESP0H,TSS_ESP0+1(%di)    # Set ESP0
        movb $SEL_SDATA,TSS_SS0(%di)    # Set SS0
        movb $_TSSIO,TSS_MAP(%di)   # Set I/O bit map base
```

*stand/i386/btx/btx/btx.S*

Обратите внимание, что в TSS указано значение для указателя стека и сегмента стека уровня привилегий 0. Это необходимо, потому что если прерывание или исключение получено во время выполнения boot2 на уровне привилегий 3, процессор автоматически переключается на уровень привилегий 0, поэтому требуется новый рабочий стек. Наконец, полю базового адреса карты ввода-вывода TSS присваивается значение, которое представляет собой 16-битное смещение от начала TSS до битовой карты разрешений ввода-вывода и битовой карты перенаправления прерываний.

После создания IDT и TSS процессор готов к переходу в защищённый режим. Это выполняется в следующем блоке:

```
/*
 * Bring up the system.
 */
        mov $0x2820,%bx         # Set protected mode
        callw setpic            #  IRQ offsets
        lidt idtdesc            # Set IDT
        lgdt gdtdesc            # Set GDT
        mov %cr0,%eax           # Switch to protected
        inc %ax             #  mode
        mov %eax,%cr0           #
        ljmp $SEL_SCODE,$init.8     # To 32-bit code
        .code32
init.8:     xorl %ecx,%ecx          # Zero
```

```
        movb $SEL_SDATA,%cl     # To 32-bit
        movw %cx,%ss            #  stack
```

*stand/i386/btx/btx/btx.S*

Сначала вызывается `setpic` для программирования 8259A PIC (программируемого контроллера прерываний). Этот чип подключен к нескольким источникам аппаратных прерываний. При получении прерывания от устройства он сигнализирует процессору соответствующим вектором прерывания. Это можно настроить так, чтобы определённые прерывания были связаны с конкретными векторами прерываний, как объяснялось ранее. Затем регистры IDTR (Interrupt Descriptor Table Register) и GDTR (Global Descriptor Table Register) загружаются инструкциями `lidt` и `lgdt` соответственно. Эти регистры загружаются базовым адресом и предельным адресом для IDT и GDT. Следующие три инструкции устанавливают бит Protection Enable (PE) в регистре `%cr0`. Это фактически переключает процессор в 32-битный защищённый режим. Затем выполняется дальний переход на `init.8` с использованием селектора сегмента SEL_SCODE, который выбирает сегмент кода супервизора (Supervisor Code Segment). После этого перехода процессор фактически работает на уровне CPL 0 — наиболее привилегированном уровне. Наконец, для стека выбирается сегмент данных супервизора (Supervisor Data Segment) путем присвоения селектора сегмента SEL_SDATA регистру `%ss`. Этот сегмент данных также имеет уровень привилегий `0`.

Наш последний блок кода отвечает за загрузку TR (Регистра Задач) с селектором сегмента для TSS, который мы создали ранее, и настройку окружения пользовательского режима перед передачей управления исполнения клиенту boot2.

```
/*
 * Launch user task.
 */
        movb $SEL_TSS,%cl       # Set task
        ltr %cx             #  register
        movl $MEM_USR,%edx      # User base address
        movzwl %ss:BDA_MEM,%eax     # Get free memory
        shll $0xa,%eax          # To bytes
        subl $ARGSPACE,%eax     # Less arg space
        subl %edx,%eax          # Less base
        movb $SEL_UDATA,%cl     # User data selector
        pushl %ecx          # Set SS
        pushl %eax          # Set ESP
        push $0x202         # Set flags (IF set)
        push $SEL_UCODE         # Set CS
        pushl btx_hdr+0xc       # Set EIP
        pushl %ecx          # Set GS
        pushl %ecx          # Set FS
        pushl %ecx          # Set DS
        pushl %ecx          # Set ES
        pushl %edx          # Set EAX
        movb $0x7,%cl           # Set remaining
init.9:     push $0x0           #  general
        loop init.9         #  registers
#ifdef BTX_SERIAL
```

```
        call sio_init           # setup the serial console
#endif
        popa                #  and initialize
        popl %es            # Initialize
        popl %ds            #  user
        popl %fs            #  segment
        popl %gs            #  registers
        iret                # To user mode
```

*stand/i386/btx/btx/btx.S*

Обратите внимание, что среда клиента включает селектор сегмента стека и указатель стека (регистры `%ss` и `%esp`). Действительно, как только TR загружается соответствующим селектором сегмента стека (инструкция `ltr`), указатель стека вычисляется и помещается в стек вместе с селектором сегмента стека. Затем значение `0x202` помещается в стек; это значение, которое EFLAGS получит при передаче управления клиенту. Также в стек помещаются селектор сегмента кода пользовательского режима и точка входа клиента. Напомним, что эта точка входа прописывается в заголовке BTX во время компоновки. Наконец, селекторы сегментов (хранящиеся в регистре `%ecx`) для регистров сегментов `%gs, %fs, %ds и %es` помещаются в стек вместе со значением из `%edx` (`0xa000`). Примите во внимание эти значения, помещенные в стек (они скоро будут извлечены). Затем значения для оставшихся регистров общего назначения также помещаются в стек (обратите внимание на цикл `loop`, который помещает значение `0` семь раз). Теперь начнётся извлечение значений из стека. Сначала инструкция `popa` извлекает из стека последние семь помещённых значений. Они сохраняются в регистрах общего назначения в порядке `%edi, %esi, %ebp, %ebx, %edx, %ecx, %eax`. Затем различные селекторы сегментов, помещённые в стек, извлекаются в соответствующие регистры сегментов. В стеке остаются ещё пять значений. Они извлекаются при выполнении инструкции `iret`. Эта инструкция сначала извлекает значение, которое было помещено из заголовка BTX. Это значение является указателем на точку входа boot2. Оно помещается в регистр `%eip` — регистр указателя инструкций. Затем селектор сегмента кода пользователя извлекается и копируется в регистр `%cs`. Помните, что уровень привилегий этого сегмента — 3, наименее привилегированный уровень. Это означает, что мы должны предоставить значения для стека этого уровня привилегий. Именно поэтому процессор, помимо дальнейшего извлечения значения для регистра EFLAGS, выполняет ещё два извлечения из стека. Эти значения попадают в указатель стека (`%esp`) и сегмент стека (`%ss`). Теперь выполнение продолжается с точки входа `boot0`.

Важно отметить, как определяется сегмент пользовательского кода. *Базовый адрес* этого сегмента установлен на `0xa000`. Это означает, что адреса памяти кода являются *относительными* к адресу 0xa000; если код, который выполняется, извлекается из адреса `0x2000`, *фактический* адрес в памяти будет `0xa000+0x2000=0xc000`.

# 1.7. Этап загрузки boot2

`boot2` определяет важную структуру, `struct bootinfo`. Эта структура инициализируется `boot2` и передаётся загрузчику, а затем ядру. Некоторые узлы этой структуры устанавливаются `boot2`, остальные — загрузчиком. Эта структура, среди прочей информации, содержит имя файла

ядра, геометрию жёсткого диска в BIOS, номер диска в BIOS для загрузочного устройства, доступную физическую память, указатель `envp` и т.д. Ее определение выглядит так:

```
/usr/include/machine/bootinfo.h:
struct bootinfo {
    u_int32_t   bi_version;
    u_int32_t   bi_kernelname;      /* represents a char * */
    u_int32_t   bi_nfs_diskless;    /* struct nfs_diskless * */
                /* End of fields that are always present. */
#define bi_endcommon    bi_n_bios_used
    u_int32_t   bi_n_bios_used;
    u_int32_t   bi_bios_geom[N_BIOS_GEOM];
    u_int32_t   bi_size;
    u_int8_t    bi_memsizes_valid;
    u_int8_t    bi_bios_dev;        /* bootdev BIOS unit number */
    u_int8_t    bi_pad[2];
    u_int32_t   bi_basemem;
    u_int32_t   bi_extmem;
    u_int32_t   bi_symtab;      /* struct symtab * */
    u_int32_t   bi_esymtab;     /* struct symtab * */
                /* Items below only from advanced bootloader */
    u_int32_t   bi_kernend;     /* end of kernel space */
    u_int32_t   bi_envp;        /* environment */
    u_int32_t   bi_modulep;     /* preloaded modules */
};
```

`boot2` входит в бесконечный цикл, ожидая ввода пользователя, затем вызывает `load()`. Если пользователь ничего не нажимает, цикл прерывается по таймауту, и `load()` загружает файл по умолчанию (/boot/loader). Функции `ino_t lookup(char *filename)` и `int xfsread(ino_t inode, void *buf, size_t nbyte)` используются для чтения содержимого файла в память. /boot/loader — это ELF-бинарный файл, но с заголовком ELF, перед которым добавлена структура `struct exec` из a.out. `load()` анализирует ELF-заголовок загрузчика, загружает содержимое /boot/loader в память и передаёт управление на точку входа загрузчика:

```
stand/i386/boot2/boot2.c:
    __exec((caddr_t)addr, RB_BOOTINFO | (opts & RBX_MASK),
       MAKEBOOTDEV(dev_maj[dsk.type], dsk.slice, dsk.unit, dsk.part),
       0, 0, 0, VTOP(&bootinfo));
```

# 1.8. Этап загрузчика (loader)

Загрузчик также является клиентом BTX. Я не буду подробно описывать его здесь, существует исчерпывающая man-страница, написанная Майком Смитом: loader(8). Основные механизмы и BTX были рассмотрены выше.

Основная задача загрузчика — загрузить ядро. Когда ядро загружено в память, загрузчик вызывает его:

```
stand/common/boot.c:
    /* Call the exec handler from the loader matching the kernel */
    file_formats[fp->f_loader]->l_exec(fp);
```

# 1.9. Инициализация ядра

Давайте рассмотрим команду, которая компонует ядро. Это поможет определить точное местоположение, где загрузчик передаёт выполнение ядру. Это местоположение является фактической точкой входа ядра. Данная команда теперь исключена из sys/conf/Makefile.i386. Интересующее нас содержимое можно найти в /usr/obj/usr/src/i386.i386/sys/GENERIC/.

```
/usr/obj/usr/src/i386.i386/sys/GENERIC/kernel.meta:
ld -m elf_i386_fbsd -Bdynamic -T /usr/src/sys/conf/ldscript.i386 --build-id=sha1 --no
-warn-mismatch \
--warn-common --export-dynamic  --dynamic-linker /red/herring -X -o kernel locore.o
<lots of kernel .o files>
```

Вот несколько интересных наблюдений. Во-первых, ядро представляет собой динамически связанный бинарный файл ELF, но динамический компоновщик для ядра — это /red/herring, что явно является фиктивным файлом. Во-вторых, взглянув на файл sys/conf/ldscript.i386, можно понять, какие параметры ld используются при компиляции ядра. Читая первые несколько строк, видим, что строка

```
sys/conf/ldscript.i386:
ENTRY(btext)
```

говорит, что точка входа ядра — это символ `btext`. Этот символ определён в locore.s:

```
sys/i386/i386/locore.s:
    .text
/**********************************************************************
 *
 * This is where the bootblocks start us, set the ball rolling...
 *
 */
NON_GPROF_ENTRY(btext)
```

Сначала регистр EFLAGS устанавливается в предопределённое значение 0x00000002. Затем инициализируются все сегментные регистры:

```
sys/i386/i386/locore.s:
/* Don't trust what the BIOS gives for eflags. */
    pushl   $PSL_KERNEL
    popfl
```

```
/*
 * Don't trust what the BIOS gives for %fs and %gs.  Trust the bootstrap
 * to set %cs, %ds, %es and %ss.
 */
    mov %ds, %ax
    mov %ax, %fs
    mov %ax, %gs
```

btext вызывает подпрограммы `recover_bootinfo()` и `identify_cpu()`, которые также определены в locore.s. Вот описание их функций:

| | |
|---|---|
| `recover_bootinfo` | Эта процедура разбирает параметры, переданные ядру при загрузке. Ядро могло быть загружено тремя способами: загрузчиком (как описано выше), старыми загрузочными блоками диска или по старой процедуре загрузки без диска. Эта функция определяет метод загрузки и сохраняет структуру `struct bootinfo` в памяти ядра. |
| `identify_cpu` | Эта функция пытается определить, на каком процессоре она выполняется, сохраняя найденное значение в переменной `_cpu`. |

Следующие шаги включают активацию VME, если процессор поддерживает эту функцию:

```
sys/i386/i386/mpboot.s:
    testl   $CPUID_VME,%edx
    jz  3f
    orl $CR4_VME,%eax
3:  movl    %eax,%cr4
```

Затем, включение подкачки:

```
sys/i386/i386/mpboot.s:
/* Now enable paging */
    movl    IdlePTD_nopae, %eax
    movl    %eax,%cr3           /* load ptd addr into mmu */
    movl    %cr0,%eax           /* get control word */
    orl $CR0_PE|CR0_PG,%eax     /* enable paging */
    movl    %eax,%cr0           /* and let's page NOW! */
```

Следующие три строки кода необходимы, потому что была установлена подкачка, поэтому требуется переход для продолжения выполнения в виртуализированном адресном пространстве:

```
sys/i386/i386/mpboot.s:
    pushl   $mp_begin               /* jump to high mem */
    ret

/* now running relocated at KERNBASE where the system is linked to run */
mp_begin:   /* now running relocated at KERNBASE */
```

Функция `init386()` вызывается с указателем на первую свободную физическую страницу, после чего следует вызов `mi_startup()`. `init386` — это архитектурно-зависимая функция инициализации, а `mi_startup()` — архитектурно-независимая (префикс 'mi_' означает Machine Independent, то есть «независимая от машины»). Ядро никогда не возвращается из `mi_startup()`, и, вызывая её, завершает загрузку:

```
sys/i386/i386/locore.s:
    pushl   physfree            /* value of first for init386(first) */
    call    init386             /* wire 386 chip for unix operation */
    addl    $4,%esp
    movl    %eax,%esp           /* Switch to true top of stack. */
    call    mi_startup          /* autoconfiguration, mountroot etc */
    /* NOTREACHED */
```

### 1.9.1. `init386()`

`init386()` определена в sys/i386/i386/machdep.c и выполняет низкоуровневую инициализацию, специфичную для чипа i386. Переход в защищённый режим был выполнен загрузчиком. Загрузчик создал самую первую задачу, в которой ядро продолжает работать. Прежде чем рассматривать код, рассмотрим задачи, которые процессор должен выполнить для инициализации выполнения в защищённом режиме:

- Инициализировать настраиваемые параметры ядра, переданные из загрузочной программы.
- Подготовить GDT.
- Подготовить IDT.
- Инициализировать системную консоль.
- Инициализировать DDB, если он скомпилирован в ядро.
- Инициализировать TSS.
- Подготовить LDT.
- Настройка pcb для thread0.

`init386()` инициализирует настраиваемые параметры, переданные из bootstrap, устанавливая указатель окружения (envp) и вызывая `init_param1()`. Указатель envp был передан из loader в структуре `bootinfo`:

```
sys/i386/i386/machdep.c:
```

```
    /* Init basic tunables, hz etc */
    init_param1();
```

`init_param1()` определена в sys/kern/subr_param.c. Этот файл содержит ряд sysctl, а также две функции, `init_param1()` и `init_param2()`, которые вызываются из `init386()`:

```
sys/kern/subr_param.c:
    hz = -1;
    TUNABLE_INT_FETCH("kern.hz", &hz);
    if (hz == -1)
        hz = vm_guest > VM_GUEST_NO ? HZ_VM : HZ;
```

`TUNABLE_<typename>_FETCH` используется для получения значения из окружения:

```
/usr/src/sys/sys/kernel.h:
#define TUNABLE_INT_FETCH(path, var)    getenv_int((path), (var))
```

Sysctl `kern.hz` представляет собой такт системных часов. Кроме того, эти параметры sysctl устанавливаются функцией `init_param1()`: `kern.maxswzone`, `kern.maxbcache`, `kern.maxtsiz`, `kern.dfldsiz`, `kern.maxdsiz`, `kern.dflssiz`, `kern.maxssiz`, `kern.sgrowsiz`.

Затем `init386()` подготавливает Глобальную Таблицу Дескрипторов (GDT). Каждая задача на x86 выполняется в своём собственном виртуальном адресном пространстве, и это пространство адресуется парой сегмент:смещение. Например, если текущая инструкция, которую должен выполнить процессор, находится по адресу CS:EIP, то линейный виртуальный адрес этой инструкции будет "виртуальный адрес кодового сегмента CS"
EIP. Для удобства сегменты начинаются с виртуального адреса 0 и заканчиваются на границе 4 ГБ. Таким образом, линейный виртуальный адрес инструкции в данном примере будет просто значением EIP. Сегментные регистры, такие как CS, DS и другие, являются селекторами, то есть индексами в GDT (если быть более точным, индекс — это не сам селектор, а поле INDEX в селекторе). GDT в FreeBSD содержит дескрипторы для 15 селекторов на каждый CPU:

```
sys/i386/i386/machdep.c:
union descriptor gdt0[NGDT];    /* initial global descriptor table */
union descriptor *gdt = gdt0;   /* global descriptor table */

sys/x86/include/segments.h:
/*
 * Entries in the Global Descriptor Table (GDT)
 */
#define GNULL_SEL   0   /* Null Descriptor */
#define GPRIV_SEL   1   /* SMP Per-Processor Private Data */
#define GUFS_SEL    2   /* User %fs Descriptor (order critical: 1) */
#define GUGS_SEL    3   /* User %gs Descriptor (order critical: 2) */
#define GCODE_SEL   4   /* Kernel Code Descriptor (order critical: 1) */
#define GDATA_SEL   5   /* Kernel Data Descriptor (order critical: 2) */
```

```
#define GUCODE_SEL  6   /* User Code Descriptor (order critical: 3) */
#define GUDATA_SEL  7   /* User Data Descriptor (order critical: 4) */
#define GBIOSLOWMEM_SEL 8   /* BIOS low memory access (must be entry 8) */
#define GPROC0_SEL  9   /* Task state process slot zero and up */
#define GLDT_SEL    10  /* Default User LDT */
#define GUSERLDT_SEL    11  /* User LDT */
#define GPANIC_SEL  12  /* Task state to consider panic from */
#define GBIOSCODE32_SEL 13  /* BIOS interface (32bit Code) */
#define GBIOSCODE16_SEL 14  /* BIOS interface (16bit Code) */
#define GBIOSDATA_SEL   15  /* BIOS interface (Data) */
#define GBIOSUTIL_SEL   16  /* BIOS interface (Utility) */
#define GBIOSARGS_SEL   17  /* BIOS interface (Arguments) */
#define GNDIS_SEL   18  /* For the NDIS layer */
#define NGDT        19
```

Обратите внимание, что эти `#defines` не являются самими селекторами, а лишь полем `INDEX` селектора, поэтому они точно соответствуют индексам GDT. Например, реальный селектор для кода ядра (`GCODE_SEL`) имеет значение `0x20`.

Следующий шаг — инициализация таблицы дескрипторов прерываний (IDT). Эта таблица используется процессором при возникновении программного или аппаратного прерывания. Например, чтобы выполнить системный вызов, пользовательское приложение использует инструкцию `INT 0x80`. Это программное прерывание, поэтому аппаратное обеспечение процессора ищет запись с индексом 0x80 в IDT. Эта запись указывает на процедуру обработки данного прерывания, в данном конкретном случае это будет шлюз системных вызовов ядра. IDT может содержать максимум 256 (0x100) записей. Ядро выделяет NIDT записей для IDT, где NIDT — это максимум (256):

```
sys/i386/i386/machdep.c:
static struct gate_descriptor idt0[NIDT];
struct gate_descriptor *idt = &idt0[0]; /* interrupt descriptor table */
```

Для каждого прерывания устанавливается соответствующий обработчик. Также настраивается шлюз системного вызова для `INT 0x80`:

```
sys/i386/i386/machdep.c:
    setidt(IDT_SYSCALL, &IDTVEC(int0x80_syscall),
            SDT_SYS386IGT, SEL_UPL, GSEL(GCODE_SEL, SEL_KPL));
```

Итак, когда пользовательское приложение выполняет инструкцию `INT 0x80`, управление передаётся функции `_Xint0x80_syscall`, которая находится в сегменте кода ядра и будет выполнена с привилегиями супервизора.

Консоль и DDB инициализируются:

```
sys/i386/i386/machdep.c:
    cninit();
```

```
/* skipped */
  kdb_init();
#ifdef KDB
    if (boothowto & RB_KDB)
        kdb_enter(KDB_WHY_BOOTFLAGS, "Boot flags requested debugger");
#endif
```

Сегмент состояния задачи (TSS) — это ещё одна структура защищённого режима x86, используемая оборудованием для хранения информации о задаче при переключении задач.

Локальная таблица дескрипторов (LDT) используется для ссылки на код и данные пользовательского пространства. Определено несколько селекторов, указывающих на LDT, включая шлюзы системных вызовов, а также селекторы кода и данных пользователя:

```
sys/x86/include/segments.h:
#define LSYS5CALLS_SEL  0   /* forced by intel BCS */
#define LSYS5SIGR_SEL   1
#define LUCODE_SEL  3
#define LUDATA_SEL  5
#define NLDT        (LUDATA_SEL + 1)
```

Далее инициализируется структура Блока Управления Процессом (`struct pcb`) для proc0. proc0 — это структура `struct proc`, описывающая процесс ядра. Она всегда присутствует во время работы ядра, поэтому связана с thread0:

```
sys/i386/i386/machdep.c:
register_t
init386(int first)
{
    /* ... skipped ... */

    proc_linkup0(&proc0, &thread0);
    /* ... skipped ... */
}
```

Структура `struct pcb` является частью структуры proc. Она определена в /usr/include/machine/pcb.h и содержит информацию процесса, специфичную для архитектуры i386, такую как значения регистров.

### 1.9.2. `mi_startup()`

Эта функция выполняет сортировку пузырьком всех объектов инициализации системы, а затем вызывает вход каждого объекта по очереди:

```
sys/kern/init_main.c:
    for (sipp = sysinit; sipp < sysinit_end; sipp++) {
```

```
        /* ... skipped ... */

        /* Call function */
        (*((*sipp)->func))((*sipp)->udata);
        /* ... skipped ... */
    }
```

Хотя фреймворк sysinit описан в Руководстве разработчика, я рассмотрю его внутреннее устройство.

Каждый объект инициализации системы (объект sysinit) создаётся путем вызова макроса SYSINIT(). Возьмем, к примеру, объект sysinit `announce`. Этот объект выводит сообщение об авторских правах:

```
sys/kern/init_main.c:
static void
print_caddr_t(void *data __unused)
{
    printf("%s", (char *)data);
}
/* ... skipped ... */
SYSINIT(announce, SI_SUB_COPYRIGHT, SI_ORDER_FIRST, print_caddr_t, copyright);
```

Идентификатор подсистемы для этого объекта — SI_SUB_COPYRIGHT (0x0800001). Таким образом, сообщение об авторских правах будет выведено первым, сразу после инициализации консоли.

Давайте рассмотрим, что именно делает макрос `SYSINIT()`. Он раскрывается в макрос `C_SYSINIT()`. Макрос `C_SYSINIT()` затем раскрывается в статическое объявление структуры `struct sysinit` с вызовом другого макроса `DATA_SET`:

```
/usr/include/sys/kernel.h:
      #define C_SYSINIT(uniquifier, subsystem, order, func, ident) \
      static struct sysinit uniquifier ## _sys_init = { \ subsystem, \
      order, \ func, \ (ident) \ }; \ DATA_WSET(sysinit_set,uniquifier ##
      _sys_init);

#define SYSINIT(uniquifier, subsystem, order, func, ident)  \
    C_SYSINIT(uniquifier, subsystem, order,         \
    (sysinit_cfunc_t)(sysinit_nfunc_t)func, (void *)(ident))
```

Макрос `DATA_SET()` раскрывается в `_MAKE_SET()`, и именно в этом макросе скрыта вся магия инициализации системы:

```
/usr/include/linker_set.h:
#define TEXT_SET(set, sym) _MAKE_SET(set, sym)
```

```
#define DATA_SET(set, sym) _MAKE_SET(set, sym)
```

После выполнения этих макросов в ядре были созданы различные разделы, включая `set.sysinit_set`. Запустив objdump для бинарного файла ядра, можно заметить наличие таких небольших разделов:

```
% llvm-objdump -h /kernel
Sections:
Idx Name                          Size     VMA      Type
 10 set_sysctl_set                000021d4 01827078 DATA
 16 set_kbddriver_set             00000010 0182a4d0 DATA
 20 set_scterm_set                0000000c 0182c75c DATA
 21 set_cons_set                  00000014 0182c768 DATA
 33 set_scrndr_set                00000024 0182c828 DATA
 41 set_sysinit_set               000014d8 018fabb0 DATA
```

Это содержимое экрана показывает, что размер раздела set.sysinit_set составляет 0x14d8 байт, поэтому `0x14d8/sizeof(void *)` объектов sysinit скомпилировано в ядро. Другие разделы, такие как `set.sysctl_set`, представляют другие наборы компоновщика.

Определяя переменную типа `struct sysinit`, содержимое раздела `set.sysinit_set` будет "собрано" в эту переменную:

```
sys/kern/init_main.c:
  SET_DECLARE(sysinit_set, struct sysinit);
```

`struct sysinit` определена следующим образом:

```
sys/sys/kernel.h:
  struct sysinit {
    enum sysinit_sub_id subsystem;  /* subsystem identifier*/
    enum sysinit_elem_order order;      /* init order within subsystem*/
    sysinit_cfunc_t func;           /* function     */
    const void  *udata;         /* multiplexer/argument */
};
```

Возвращаясь к обсуждению `mi_startup()`, теперь должно быть понятно, как организованы объекты sysinit. Функция `mi_startup()` сортирует их и вызывает каждый. Самый последний объект — это системный планировщик:

```
/usr/include/sys/kernel.h:
enum sysinit_sub_id {
    SI_SUB_DUMMY        = 0x0000000,    /* not executed; for linker*/
    SI_SUB_DONE     = 0x0000001,    /* processed*/
    SI_SUB_TUNABLES     = 0x0700000,    /* establish tunable values */
    SI_SUB_COPYRIGHT    = 0x0800001,    /* first use of console*/
```

```
...
    SI_SUB_LAST     = 0xfffffff /* final initialization */
};
```

Системный планировщик sysinit определен в файле sys/vm/vm_glue.c, а точка входа для этого объекта — `scheduler()`. Эта функция фактически представляет собой бесконечный цикл и описывает процесс с PID 0, известный как процесс swapper. Структура thread0, упомянутая ранее, используется для его описания.

Первый пользовательский процесс, называемый *init*, создаётся объектом sysinit `init`:

```
sys/kern/init_main.c:
static void
create_init(const void *udata __unused)
{
    struct fork_req fr;
    struct ucred *newcred, *oldcred;
    struct thread *td;
    int error;

    bzero(&fr, sizeof(fr));
    fr.fr_flags = RFFDG | RFPROC | RFSTOPPED;
    fr.fr_procp = &initproc;
    error = fork1(&thread0, &fr);
    if (error)
        panic("cannot fork init: %d\n", error);
    KASSERT(initproc->p_pid == 1, ("create_init: initproc->p_pid != 1"));
    /* divorce init's credentials from the kernel's */
    newcred = crget();
    sx_xlock(&proctree_lock);
    PROC_LOCK(initproc);
    initproc->p_flag |= P_SYSTEM | P_INMEM;
    initproc->p_treeflag |= P_TREE_REAPER;
    oldcred = initproc->p_ucred;
    crcopy(newcred, oldcred);
#ifdef MAC
    mac_cred_create_init(newcred);
#endif
#ifdef AUDIT
    audit_cred_proc1(newcred);
#endif
    proc_set_cred(initproc, newcred);
    td = FIRST_THREAD_IN_PROC(initproc);
    crcowfree(td);
    td->td_realucred = crcowget(initproc->p_ucred);
    td->td_ucred = td->td_realucred;
    PROC_UNLOCK(initproc);
    sx_xunlock(&proctree_lock);
    crfree(oldcred);
    cpu_fork_kthread_handler(FIRST_THREAD_IN_PROC(initproc), start_init, NULL);
```

```
}
SYSINIT(init, SI_SUB_CREATE_INIT, SI_ORDER_FIRST, create_init, NULL);
```

Функция `create_init()` выделяет новый процесс, вызывая `fork1()`, но не помечает его как готовый к выполнению. Когда этот новый процесс будет запланирован для выполнения планировщиком, будет вызвана функция `start_init()`. Эта функция определена в init_main.c. Она пытается загрузить и выполнить бинарный файл init, сначала проверяя /sbin/init, затем /sbin/oinit, /sbin/init.bak и, наконец, /rescue/init:

```
sys/kern/init_main.c:
static char init_path[MAXPATHLEN] =
#ifdef  INIT_PATH
    __XSTRING(INIT_PATH);
#else
    "/sbin/init:/sbin/oinit:/sbin/init.bak:/rescue/init";
#endif
```

[1] Это приглашение появится, если пользователь нажмет клавишу сразу после выбора ОС для загрузки на этапе boot0.

[2] В случае сомнений мы отсылаем читателя к официальным руководствам Intel, где описана точная семантика каждой инструкции.

[3] Файл /boot/boot1 существует, но он не записывается в начало раздела FreeBSD. Вместо этого он объединяется с boot2, формируя файл boot, который записывается в начало раздела FreeBSD и считывается во время загрузки.

[4] На самом деле мы передали указатель на адрес слайса в регистре %si. Однако boot1 не предполагает, что он был загружен boot0 (возможно, его загрузил другой MBR и не передал эту информацию), поэтому он ничего не предполагает.

[5] В контексте 16-битного реального режима слово — это 2 байта.

[6] 512*16=8192 байта, ровно размер boot

[7] Исторически известной как disklabel. Если вам когда-либо было интересно, где FreeBSD хранит эту информацию, она находится в этой области — см. bsdlabel(8)

[8] Это необходимо по историческим причинам.

[9] Код и данные реального режима необходимы при переключении обратно в реальный режим из защищённого режима, как указано в руководствах Intel.

# Глава 2. Заметки о блокировках

*Эта глава сопровождается и поддерживается проектом FreeBSD SMP Next Generation.*

Этот документ описывает механизмы блокировки, используемые в ядре FreeBSD для эффективной многопроцессорной обработки. Блокировка может быть достигнута несколькими способами. Структуры данных могут защищаться мьютексами или блокировками (lock) из lockmgr(9). Некоторые переменные защищаются просто за счёт использования атомарных операций для доступа к ним.

## 2.1. Mutexes

Мьютекс — это просто блокировка, используемая для обеспечения взаимного исключения. Конкретно, мьютекс может принадлежать только одному объекту в один момент времени. Если другой объект хочет получить мьютекс, который уже принадлежит кому-то, он должен ждать, пока мьютекс не будет освобожден. В ядре FreeBSD мьютексы принадлежат процессам.

Мьютексы могут быть получены рекурсивно, но предполагается, что они удерживаются в течение короткого периода времени. В частности, нельзя переходить в режим сна, удерживая мьютекс. Если необходимо удерживать блокировку во время сна, используйте блокировку lockmgr(9).

Каждый мьютекс обладает несколькими важными свойствами:

**Имя переменной**

Имя переменной struct mtx в исходном коде ядра.

**Логическое имя**

Имя мьютекса, назначенное ему с помощью `mtx_init`. Это имя отображается в сообщениях трассировки KTR, ошибках и предупреждениях witness, а также используется для различения мьютексов в коде witness.

**Тип**

Тип мьютекса в терминах флагов `MTX_*`. Значение каждого флага связано с его значением, как описано в mutex(9).

`MTX_DEF`

Мьютекс блокировки с ожиданием

`MTX_SPIN`

Мьютекс с вращающейся блокировкой (spin mutex)

`MTX_RECURSE`

Этот мьютекс допускает рекурсию.

**Защищаемые системы**

Список структур данных или членов структур данных, которые защищает эта запись. Для

членов структур данных имя будет указано в формате `имя структуры.имя члена`.

**Зависимые функции**

Функции, которые могут быть вызваны только при удержании этого мьютекса.

*Таблица 1. Список мьютексов*

| Имя переменной | Логическое имя | Тип | Защищаемые системы | Зависимые функции |
|---|---|---|---|---|
| sched_lock | "sched lock" | MTX_SPIN \| MTX_RECURSE | _gmonparam, cnt.v_swtch, cp_time, curpriority, mtx.mtx_blocked, mtx.mtx_contested, proc.p_procq, proc.p_slpq, proc.p_sflag, proc.p_stat, proc.p_estcpu, proc.p_cpticks proc.p_pctcpu, proc.p_wchan, proc.p_wmesg, proc.p_swtime, proc.p_slptime, proc.p_runtime, proc.p_uu, proc.p_su, proc.p_iu, proc.p_uticks, proc.p_sticks, proc.p_iticks, proc.p_oncpu, proc.p_lastcpu, proc.p_rqindex, proc.p_heldmtx, proc.p_blocked, proc.p_mtxname, proc.p_contested, proc.p_priority, proc.p_usrpri, proc.p_nativepri, proc.p_nice, proc.p_rtprio, pscnt, slpque, itqueuebits, itqueues, rtqueuebits, rtqueues, queuebits, queues, idqueuebits, idqueues, switchtime, switchticks | setrunqueue, remrunqueue, mi_switch, chooseproc, schedclock, resetpriority, updatepri, maybe_resched, cpu_switch, cpu_throw, need_resched, resched_wanted, clear_resched, aston, astoff, astpending, calcru, proc_compare |
| vm86pcb_lock | "vm86pcb lock" | MTX_DEF | vm86pcb | vm86_bioscall |
| Giant | "Giant" | MTX_DEF \| MTX_RECURSE | почти всё | много |
| callout_lock | "callout lock" | MTX_SPIN \| MTX_RECURSE | callfree, callwheel, nextsoftcheck, proc.p_itcallout, proc.p_slpcallout, softticks, ticks | |

## 2.2. Разделяемые эксклюзивные блокировки

Эти блокировки обеспечивают базовую функциональность типа читатель-писатель и могут удерживаться спящим процессом. В настоящее время они реализованы через lockmgr(9).

*Таблица 2. Список разделяемых эксклюзивных блокировок*

| Имя переменной | Защищаемые системы |
|---|---|
| allproc_lock | allproc zombproc pidhashtbl proc.p_list proc.p_hash nextpid |

| Имя переменной | Защищаемые системы |
| --- | --- |
| `proctree_lock` | `proc.p_children` `proc.p_sibling` |

## 2.3. Атомарно защищённые переменные

Переменная с атомарной защитой — это специальная переменная, которая не защищена явной блокировкой. Вместо этого все операции доступа к данным этой переменной используют специальные атомарные операции, как описано в atomic(9). Очень немногие переменные обрабатываются таким образом, хотя другие примитивы синхронизации, такие как мьютексы, реализованы с использованием переменных с атомарной защитой.

- `mtx.mtx_lock`

# Глава 3. Объекты ядра

Объекты ядра, или *Kobj*, предоставляют объектно-ориентированную систему программирования на языке C для ядра. Таким образом, данные, с которыми производится работа, содержат описание того, как над ними следует выполнять операции. Это позволяет добавлять и удалять операции из интерфейса во время выполнения без нарушения бинарной совместимости.

## 3.1. Терминология

**Объект**

Набор данных - структура данных - аллокация данных.

**Метод**

Операция — функция.

**Класс**

Один или несколько методов.

**Интерфейс**

Стандартный набор из одного или нескольких методов.

## 3.2. Как работает Kobj

Kobj работает путем генерации описаний методов. Каждое описание содержит уникальный идентификатор, а также функцию по умолчанию. Адрес описания используется для однозначной идентификации метода в таблице методов класса.

Класс создаётся путем построения таблицы методов, связывающей одну или несколько функций с описаниями методов. Перед использованием класс компилируется. В процессе компиляции выделяется кэш и связывается с классом. Уникальный идентификатор назначается каждому описанию метода в таблице методов класса, если это ещё не было сделано другой компиляцией, ссылающейся на этот класс. Для каждого используемого метода скриптом генерируется функция для проверки аргументов и автоматического обращения к описанию метода для поиска. Сгенерированная функция ищет метод, используя уникальный идентификатор, связанный с описанием метода, в качестве хэша для доступа к кэшу, связанному с классом объекта. Если метод не найден в кэше, сгенерированная функция использует таблицу класса для поиска метода. Если метод найден, используется связанная с ним функция внутри класса; в противном случае используется функция по умолчанию, связанная с описанием метода.

Эти перенаправления можно визуализировать следующим образом:

```
object->cache<->class
```

## 3.3. Использование Kobj

### 3.3.1. Структуры

```
struct kobj_method
```

### 3.3.2. Функции

```
void kobj_class_compile(kobj_class_t cls);
void kobj_class_compile_static(kobj_class_t cls, kobj_ops_t ops);
void kobj_class_free(kobj_class_t cls);
kobj_t kobj_create(kobj_class_t cls, struct malloc_type *mtype, int mflags);
void kobj_init(kobj_t obj, kobj_class_t cls);
void kobj_delete(kobj_t obj, struct malloc_type *mtype);
```

### 3.3.3. Макросы

```
KOBJ_CLASS_FIELDS
KOBJ_FIELDS
DEFINE_CLASS(name, methods, size)
KOBJMETHOD(NAME, FUNC)
```

### 3.3.4. Заголовки

```
<sys/param.h>
<sys/kobj.h>
```

### 3.3.5. Создание шаблона интерфейса

Первым шагом в использовании Kobj является создание интерфейса. Создание интерфейса включает в себя создание шаблона, который скрипт src/sys/kern/makeobjops.pl может использовать для генерации заголовочного файла и кода объявлений методов и функций поиска методов.

В этом шаблоне используются следующие ключевые слова: `#include`, `INTERFACE`, `CODE`, `EPILOG`, `HEADER`, `METHOD`, `PROLOG`, `STATICMETHOD` и `DEFAULT`.

Включение директивы `#include` и всего, что следует за ней, копируется дословно в начало сгенерированного файла с кодом.

Для примера:

```
#include <sys/foo.h>
```

Ключевое слово `INTERFACE` используется для определения имени интерфейса. Это имя объединяется с каждым именем метода в формате [имя интерфейса]_[имя метода]. Его синтаксис: `INTERFACE [имя интерфейса];`.

Для примера:

```
INTERFACE foo;
```

Ключевое слово `CODE` копирует свои аргументы дословно в файл кода. Его синтаксис: `CODE { [что угодно] };`

Для примера:

```
CODE {
    struct foo * foo_alloc_null(struct bar *)
    {
        return NULL;
    }
};
```

Ключевое слово `HEADER` копирует свои аргументы в заголовочный файл без изменений. Его синтаксис: `HEADER { [что угодно] };`

Для примера:

```
HEADER {
        struct mumble;
        struct grumble;
};
```

Ключевое слово `METHOD` описывает метод. Его синтаксис: `METHOD [возвращаемый тип] [имя метода] { [объект [, аргументы]] };`

Для примера:

```
METHOD int bar {
    struct object *;
    struct foo *;
    struct bar;
};
```

Ключевое слово `DEFAULT` может следовать за ключевым словом `METHOD`. Оно расширяет ключевое слово `METHOD`, включая функцию по умолчанию для метода. Расширенный синтаксис выглядит так: `METHOD [тип возвращаемого значения] [имя метода] { [объект; [другие аргументы]] } DEFAULT [функция по умолчанию];`

Для примера:

```
METHOD int bar {
    struct object *;
    struct foo *;
    int bar;
} DEFAULT foo_hack;
```

Ключевое слово `STATICMETHOD` используется аналогично ключевому слову `METHOD`, за исключением того, что данные kobj не находятся в начале структуры объекта, поэтому приведение к типу kobj_t было бы некорректным. Вместо этого `STATICMETHOD` полагается на то, что данные Kobj указаны как 'ops'. Это также полезно для вызова методов напрямую из таблицы методов класса.

Ключевые слова `PROLOG` и `EPILOG` вставляют код непосредственно перед или сразу после `METHOD`, к которому они прикреплены. Эта функция в основном используется для профилирования в ситуациях, когда сложно получить информацию другим способом.

Другие полные примеры:

```
src/sys/kern/bus_if.m
src/sys/kern/device_if.m
```

### 3.3.6. Создание класса

Второй шаг в использовании Kobj — это создание класса. Класс состоит из имени, таблицы методов и размера объектов, если используются средства обработки объектов Kobj. Для создания класса используйте макрос `DEFINE_CLASS()`. Чтобы создать таблицу методов, создайте массив элементов kobj_method_t, завершающийся записью NULL. Каждую не-NULL запись можно создать с помощью макроса `KOBJMETHOD()`.

Для примера:

```
DEFINE_CLASS(fooclass, foomethods, sizeof(struct foodata));

kobj_method_t foomethods[] = {
    KOBJMETHOD(bar_doo, foo_doo),
    KOBJMETHOD(bar_foo, foo_foo),
    { NULL, NULL}
};
```

Класс должен быть "скомпилирован". В зависимости от состояния системы на момент инициализации класса, необходимо использовать статически выделенный кэш, "таблицу операций". Это может быть достигнуто путем объявления `struct kobj_ops` и использования `kobj_class_compile_static();` в противном случае следует использовать `kobj_class_compile()`.

### 3.3.7. Создание объекта

Третий шаг в использовании Kobj связан с определением объекта. Процедуры создания объекта Kobj предполагают, что данные Kobj находятся в начале объекта. Если это не подходит, вам придется самостоятельно выделить память для объекта, а затем использовать `kobj_init()` для части объекта, относящейся к Kobj; в противном случае вы можете использовать `kobj_create()` для автоматического выделения и инициализации части объекта, относящейся к Kobj. `kobj_init()` также может использоваться для изменения класса, который использует объект.

Для интеграции Kobj в объект следует использовать макрос `KOBJ_FIELDS`.

Например

```
struct foo_data {
    KOBJ_FIELDS;
    foo_foo;
    foo_bar;
};
```

### 3.3.8. Вызов методов

Последним шагом в использовании Kobj является простое использование сгенерированных функций для вызова нужного метода в классе объекта. Это так же просто, как использование имени интерфейса и имени метода с небольшими изменениями. Имя интерфейса должно быть соединено с именем метода с использованием символа '_' между ними, все в верхнем регистре.

Например, если имя интерфейса было foo, а метод — bar, то вызов будет выглядеть следующим образом:

```
[return value = ] FOO_BAR(object [, other parameters]);
```

### 3.3.9. Очистка

Когда объект, выделенный через `kobj_create()`, больше не нужен, можно вызвать для него `kobj_delete()`, а когда класс больше не используется, можно вызвать для него `kobj_class_free()`.

# Глава 4. Подсистема клеток

На большинстве систем UNIX® пользователь `root` обладает неограниченной властью. Это не способствует безопасности. Если злоумышленник получит права `root` в системе, у него окажутся все функции под рукой. В FreeBSD существуют sysctl-параметры, которые ограничивают власть `root`, чтобы минимизировать ущерб от действий злоумышленника. В частности, одна из таких функций называется `уровни безопасности`. Аналогично, другая функция, доступная начиная с FreeBSD 4.0, — это утилита jail(8) — клетка. Клетка создаёт chroot-окружение и накладывает определённые ограничения на процессы, запущенные внутри `клетки`. Например, процесс в `клетке` не может влиять на процессы вне её, использовать определённые системные вызовы или наносить какой-либо ущерб основной системе.

Клетка становится новой моделью безопасности. Пользователи запускают потенциально уязвимые серверы, такие как Apache, BIND и sendmail, внутри клеток, так что если злоумышленник получит права `root` внутри клетки, это будет лишь неудобством, а не катастрофой. Данная статья в основном сосредоточена на внутреннем устройстве (исходном коде) клетки. Для получения информации о настройке клетки см. раздел о клетках Руководства FreeBSD.

## 4.1. Архитектура

`Клетка` состоит из двух областей: пользовательской программы jail(8) и кода, реализованного в ядре: системного вызова jail(2) и связанных с ним ограничений. Я расскажу о пользовательской программе, а затем о том, как `клетка` реализована в ядре.

### 4.1.1. Код в пользовательском пространстве

Исходный код пользовательской части `клетки` находится в /usr/src/usr.sbin/jail и состоит из одного файла jail.c. Программа принимает следующие аргументы: путь к `клетке`, имя хоста, IP-адрес и команду для выполнения.

#### Структуры данных

В файле jail.c первое, на что я бы обратил внимание, это объявление важной структуры `struct jail j;`, которая была включена из /usr/include/sys/jail.h.

Определение структуры `jail` выглядит следующим образом:

```
/usr/include/sys/jail.h:

struct jail {
        u_int32_t       version;
        char            *path;
        char            *hostname;
        u_int32_t       ip_number;
};
```

Как видно, существует запись для каждого из аргументов, переданных программе jail(8), и действительно, они устанавливаются во время её выполнения.

```
/usr/src/usr.sbin/jail/jail.c
char path[PATH_MAX];
...
if (realpath(argv[0], path) == NULL)
    err(1, "realpath: %s", argv[0]);
if (chdir(path) != 0)
    err(1, "chdir: %s", path);
memset(&j, 0, sizeof(j));
j.version = 0;
j.path = path;
j.hostname = argv[1];
```

**Сетевое взаимодействие**

Один из аргументов, передаваемых программе jail(8), — это IP-адрес, по которому можно получить доступ к клетке через сеть. jail(8) преобразует указанный IP-адрес в порядок байтов хоста и сохраняет его в `j` (структура `jail`).

```
/usr/src/usr.sbin/jail/jail.c:
struct in_addr in;
...
if (inet_aton(argv[2], &in) == 0)
    errx(1, "Could not make sense of ip-number: %s", argv[2]);
j.ip_number = ntohl(in.s_addr);
```

Функция inet_aton(3) "интерпретирует указанную строку символов как интернет-адрес, помещая адрес в предоставленную структуру." Член структуры `ip_number` в структуре `jail` устанавливается только тогда, когда IP-адрес, помещённый в структуру `in` функцией inet_aton(3), преобразуется в порядок байтов хоста с помощью ntohl(3).

**Процесс в клетке**

Наконец, пользовательская программа помещает процесс в `клетку`. Теперь `клетка` становится самим заключённым процессом и выполняет команду, используя execv(3).

```
/usr/src/usr.sbin/jail/jail.c
i = jail(&j);
...
if (execv(argv[3], argv + 3) != 0)
    err(1, "execv: %s", argv[3]);
```

Как видно, вызывается функция `jail()`, и её аргументом является структура `jail`, заполненная аргументами, переданными программе. В конце выполняется указанная вами программа. Теперь я расскажу о том, как `клетка` реализована в ядре.

### 4.1.2. Пространство ядра системы

Мы сейчас рассмотрим файл /usr/src/sys/kern/kern_jail.c. В этом файле определены системный вызов jail(2), соответствующие sysctls и сетевые функции.

#### Управляемые переменные ядра sysctl

В файле kern_jail.c определены следующие параметры sysctl:

```
/usr/src/sys/kern/kern_jail.c:
int     jail_set_hostname_allowed = 1;
SYSCTL_INT(_security_jail, OID_AUTO, set_hostname_allowed, CTLFLAG_RW,
    &jail_set_hostname_allowed, 0,
    "Processes in jail can set their hostnames");

int     jail_socket_unixiproute_only = 1;
SYSCTL_INT(_security_jail, OID_AUTO, socket_unixiproute_only, CTLFLAG_RW,
    &jail_socket_unixiproute_only, 0,
    "Processes in jail are limited to creating UNIX/IPv4/route sockets only");

int     jail_sysvipc_allowed = 0;
SYSCTL_INT(_security_jail, OID_AUTO, sysvipc_allowed, CTLFLAG_RW,
    &jail_sysvipc_allowed, 0,
    "Processes in jail can use System V IPC primitives");

static int jail_enforce_statfs = 2;
SYSCTL_INT(_security_jail, OID_AUTO, enforce_statfs, CTLFLAG_RW,
    &jail_enforce_statfs, 0,
    "Processes in jail cannot see all mounted file systems");

int    jail_allow_raw_sockets = 0;
SYSCTL_INT(_security_jail, OID_AUTO, allow_raw_sockets, CTLFLAG_RW,
    &jail_allow_raw_sockets, 0,
    "Prison root can create raw sockets");

int    jail_chflags_allowed = 0;
SYSCTL_INT(_security_jail, OID_AUTO, chflags_allowed, CTLFLAG_RW,
    &jail_chflags_allowed, 0,
    "Processes in jail can alter system file flags");

int     jail_mount_allowed = 0;
SYSCTL_INT(_security_jail, OID_AUTO, mount_allowed, CTLFLAG_RW,
    &jail_mount_allowed, 0,
    "Processes in jail can mount/unmount jail-friendly file systems");
```

Каждый из этих параметров sysctl может быть доступен пользователю через программу sysctl(8). В ядре эти конкретные параметры sysctl распознаются по их именам. Например, имя первого параметра sysctl — `security.jail.set_hostname_allowed`.

### Системный вызов jail(2)

Как и все системные вызовы, системный вызов jail(2) принимает два аргумента: `struct thread *td` и `struct jail_args *uap`. `td` — это указатель на структуру `thread`, которая описывает вызывающий поток. В данном контексте `uap` — это указатель на структуру, в которой содержится указатель на структуру `jail`, переданную из пользовательского пространства jail.c. Ранее, когда я описывал пользовательскую программу, вы видели, что системному вызову jail(2) была передана структура `jail` в качестве собственного аргумента.

```
/usr/src/sys/kern/kern_jail.c:
/*
 * struct jail_args {
 *  struct jail *jail;
 * };
 */
int
jail(struct thread *td, struct jail_args *uap)
```

Следовательно, `uap->jail` можно использовать для доступа к структуре `jail`, которая была передана системному вызову. Далее системный вызов копирует структуру `клетка` в пространство ядра с помощью функции copyin(9). copyin(9) принимает три аргумента: адрес данных, которые нужно скопировать в пространство ядра (`uap->jail`), место для записи данных (`j`) и размер хранилища. Структура `jail`, на которую указывает `uap->jail`, копируется в пространство ядра и сохраняется в другой структуре `клетка` — `j`.

```
/usr/src/sys/kern/kern_jail.c:
error = copyin(uap->jail, &j, sizeof(j));
```

В jail.h определена ещё одна важная структура — `prison`. Структура `prison` используется исключительно в пространстве ядра. Вот определение структуры `prison`.

```
/usr/include/sys/jail.h:
struct prison {
        LIST_ENTRY(prison) pr_list;                     /* (a) all prisons */
        int                pr_id;                       /* (c) prison id */
        int                pr_ref;                      /* (p) refcount */
        char               pr_path[MAXPATHLEN];         /* (c) chroot path */
        struct vnode      *pr_root;                     /* (c) vnode to rdir */
        char               pr_host[MAXHOSTNAMELEN];     /* (p) jail hostname */
        u_int32_t          pr_ip;                       /* (c) ip addr host */
        void              *pr_linux;                    /* (p) linux abi */
        int                pr_securelevel;              /* (p) securelevel */
        struct task        pr_task;                     /* (d) destroy task */
        struct mtx         pr_mtx;
      void             **pr_slots;                   /* (p) additional data */
};
```

Системный вызов jail(2) затем выделяет память для структуры prison и копирует данные между структурой клетка и структурой prison.

```
/usr/src/sys/kern/kern_jail.c:
MALLOC(pr, struct prison *, sizeof(*pr), M_PRISON, M_WAITOK | M_ZERO);
...
error = copyinstr(j.path, &pr->pr_path, sizeof(pr->pr_path), 0);
if (error)
    goto e_killmtx;
...
error = copyinstr(j.hostname, &pr->pr_host, sizeof(pr->pr_host), 0);
if (error)
     goto e_dropvnref;
pr->pr_ip = j.ip_number;
```

Далее мы рассмотрим ещё один важный системный вызов jail_attach(2), который реализует функцию помещения процесса в клетку.

```
/usr/src/sys/kern/kern_jail.c:
/*
 * struct jail_attach_args {
 *      int jid;
 * };
 */
int
jail_attach(struct thread *td, struct jail_attach_args *uap)
```

Этот системный вызов вносит изменения, которые позволяют отличить процесс в клетке от процессов вне клетки. Чтобы понять, что делает jail_attach(2), необходима некоторая справочная информация.

В FreeBSD каждый видимый ядром поток идентифицируется своей структурой thread, а процессы описываются их структурами proc. Определения структур thread и proc можно найти в /usr/include/sys/proc.h. Например, аргумент td в любом системном вызове на самом деле является указателем на структуру thread вызывающего потока, как было указано ранее. Член td_proc в структуре thread, на которую указывает td, является указателем на структуру proc, представляющую процесс, содержащий поток, представленный структурой td. Структура proc содержит члены, которые могут описывать идентификацию владельца (p_ucred), ограничения ресурсов процесса (p_limit) и так далее. В структуре ucred, на которую указывает член p_ucred в структуре proc, есть указатель на структуру prison (cr_prison).

```
/usr/include/sys/proc.h:
struct thread {
    ...
    struct proc *td_proc;
    ...
};
```

```
struct proc {
    ...
    struct ucred *p_ucred;
    ...
};
/usr/include/sys/ucred.h
struct ucred {
    ...
    struct prison *cr_prison;
    ...
};
```

В файле kern_jail.c функция `jail()` вызывает функцию `jail_attach()` с заданным `jid`. Затем `jail_attach()` вызывает функцию `change_root()` для изменения корневого каталога вызывающего процесса. Функция `jail_attach()` создаёт новую структуру `ucred` и присоединяет её к вызывающему процессу после успешного присоединения структуры `prison` к структуре `ucred`. С этого момента вызывающий процесс считается находящимся в клетке. Когда в ядре вызывается функция `jailed()` с вновь созданной структурой `ucred` в качестве аргумента, она возвращает 1, указывая, что учётные данные связаны с клеткой. Общим родительским процессом для всех процессов, созданных внутри клетки, является процесс, запускающий jail(8), так как он вызывает системный вызов jail(2). При выполнении программы через execve(2) она наследует свойство клетки из структуры `ucred` родительского процесса, следовательно, у неё структура `ucred` тоже со свойством клетки.

```
/usr/src/sys/kern/kern_jail.c
int
jail(struct thread *td, struct jail_args *uap)
{
...
    struct jail_attach_args jaa;
...
    error = jail_attach(td, &jaa);
    if (error)
        goto e_dropprref;
...
}

int
jail_attach(struct thread *td, struct jail_attach_args *uap)
{
    struct proc *p;
    struct ucred *newcred, *oldcred;
    struct prison *pr;
...
    p = td->td_proc;
...
    pr = prison_find(uap->jid);
...
    change_root(pr->pr_root, td);
```

```
...
    newcred->cr_prison = pr;
    p->p_ucred = newcred;
...
}
```

Когда процесс создаётся из родительского процесса, системный вызов fork(2) использует `crhold()` для поддержания учётных данных нового процесса. Это автоматически сохраняет учётные данные нового дочернего процесса согласованными с родительским, поэтому дочерний процесс также остаётся в клетке.

```
/usr/src/sys/kern/kern_fork.c:
p2->p_ucred = crhold(td->td_ucred);
...
td2->td_ucred = crhold(p2->p_ucred);
```

# 4.2. Ограничения

В ядре существуют ограничения доступа, связанные с процессами в клетках. Обычно эти ограничения просто проверяют, находится ли процесс в клетке, и если да, возвращают ошибку. Например:

```
if (jailed(td->td_ucred))
    return (EPERM);
```

### 4.2.1. SysV IPC

System V IPC основан на сообщениях. Процессы могут отправлять друг другу эти сообщения, которые указывают им, как действовать. Функции, работающие с сообщениями: msgctl(3), msgget(3), msgsnd(3) и msgrcv(3). Ранее я упоминал, что существуют определённые sysctl, которые можно включать или выключать для изменения поведения клетки. Один из таких sysctl — `security.jail.sysvipc_allowed`. По умолчанию этот sysctl установлен в 0. Если бы он был установлен в 1, это бы свело на нет весь смысл клетки: привилегированные пользователи внутри клетки смогли бы влиять на процессы за её пределами. Разница между сообщением и сигналом заключается в том, что сообщение состоит только из номера сигнала.

/usr/src/sys/kern/sysv_msg.c:

- `msgget(key, msgflg)`: `msgget` возвращает (и, возможно, создаёт) дескриптор сообщения, который обозначает очередь сообщений для использования в других функциях.
- `msgctl(msgid, cmd, buf)`: С помощью этой функции процесс может запросить статус дескриптора сообщения.
- `msgsnd(msgid, msgp, msgsz, msgflg)`: `msgsnd` отправляет сообщение процессу.
- `msgrcv(msgid, msgp, msgsz, msgtyp, msgflg)`: процесс получает сообщения с помощью этой

функции

В каждом из системных вызовов, соответствующих этим функциям, присутствует следующее условие:

```
/usr/src/sys/kern/sysv_msg.c:
if (!jail_sysvipc_allowed && jailed(td->td_ucred))
    return (ENOSYS);
```

Системные вызовы семафоров позволяют процессам синхронизировать выполнение, атомарно выполняя набор операций над набором семафоров. По сути, семафоры предоставляют ещё один способ для процессов блокировать ресурсы. Однако процесс, ожидающий семафор, который уже используется, будет находиться в состоянии сна до тех пор, пока ресурсы не будут освобождены. Следующие системные вызовы семафоров блокируются внутри клетки: semget(2), semctl(2) и semop(2).

/usr/src/sys/kern/sysv_sem.c:

- `semctl(semid, semnum, cmd, ⋯)`: `semctl` выполняет указанную команду `cmd` для очереди семафоров, указанной в `semid`.
- `semget(key, nsems, flag)`: `semget` создаёт массив семафоров, соответствующих `key`.

  `key и flag имеют то же значение, как и в msgget.`

- `semop(semid, array, nops)`: `semop` выполняет набор операций, указанных в `array`, для набора семафоров, идентифицируемых `semid`.

Система IPC System V позволяет процессам использовать общую память. Процессы могут взаимодействовать напрямую друг с другом, разделяя части своего виртуального адресного пространства и затем читая и записывая данные в общей памяти. Эти системные вызовы заблокированы в среде *клетки*: shmdt(2), shmat(2), shmctl(2) и shmget(2).

/usr/src/sys/kern/sysv_shm.c:

- `shmctl(shmid, cmd, buf)`: `shmctl` выполняет различные управляющие операции над областью разделяемой памяти, идентифицируемой `shmid`.
- `shmget(key, size, flag)`: `shmget` обращается к существующей или создаёт новую область разделяемой памяти размером `size` байт.
- `shmat(shmid, addr, flag)`: `shmat` присоединяет область разделяемой памяти, идентифицируемую `shmid`, к адресному пространству процесса.
- `shmdt(addr)`: `shmdt` отсоединяет ранее присоединенную область разделяемой памяти по адресу `addr`.

### 4.2.2. Сокеты

Клетка обрабатывает системный вызов socket(2) и связанные низкоуровневые функции сокетов особым образом. Для определения, разрешено ли создание определённого сокета, сначала проверяется значение sysctl `security.jail.socket_unixiproute_only`. Если оно

установлено, сокеты разрешено создавать только в случае, если указанное семейство равно PF_LOCAL, PF_INET или PF_ROUTE. В противном случае возвращается ошибка.

```
/usr/src/sys/kern/uipc_socket.c:
int
socreate(int dom, struct socket **aso, int type, int proto,
    struct ucred *cred, struct thread *td)
{
    struct protosw *prp;
...
    if (jailed(cred) && jail_socket_unixiproute_only &&
        prp->pr_domain->dom_family != PF_LOCAL &&
        prp->pr_domain->dom_family != PF_INET &&
        prp->pr_domain->dom_family != PF_ROUTE) {
        return (EPROTONOSUPPORT);
    }
...
}
```

### 4.2.3. Berkeley Packet Filter

Берклиевский фильтр пакетов (BPF) предоставляет низкоуровневый интерфейс к канальному уровню, независимый от протокола. В настоящее время BPF управляется через devfs(8), который определяет возможность его использования в клетке.

### 4.2.4. Протоколы

Существуют определённые протоколы, которые очень распространены, такие как TCP, UDP, IP и ICMP. IP и ICMP находятся на одном уровне: сетевом уровне 2. Принимаются определённые меры предосторожности, чтобы предотвратить привязку протокола к определённому адресу процессом в клетке, только если установлен параметр nam. nam является указателем на структуру sockaddr, которая описывает адрес, к которому привязывается служба. Более точное определение заключается в том, что sockaddr "может использоваться как шаблон для ссылки на идентификационный тег и длину каждого адреса". В функции in_pcbbind_setup(), sin — это указатель на структуру sockaddr_in, которая содержит порт, адрес, длину и семейство доменов сокета, который должен быть привязан. В основном, это запрещает любым процессам из клетки указывать адрес, который не принадлежит клетке, в которой существует вызывающий процесс.

```
/usr/src/sys/netinet/in_pcb.c:
int
in_pcbbind_setup(struct inpcb *inp, struct sockaddr *nam, in_addr_t *laddrp,
    u_short *lportp, struct ucred *cred)
{
    ...
    struct sockaddr_in *sin;
    ...
    if (nam) {
```

```
        sin = (struct sockaddr_in *)nam;
        ...
        if (sin->sin_addr.s_addr != INADDR_ANY)
            if (prison_ip(cred, 0, &sin->sin_addr.s_addr))
                return(EINVAL);
        ...
        if (lport) {
            ...
            if (prison && prison_ip(cred, 0, &sin->sin_addr.s_addr))
                return (EADDRNOTAVAIL);
            ...
        }
    }
    if (lport == 0) {
        ...
        if (laddr.s_addr != INADDR_ANY)
            if (prison_ip(cred, 0, &laddr.s_addr))
                return (EINVAL);
        ...
    }
...
    if (prison_ip(cred, 0, &laddr.s_addr))
        return (EINVAL);
...
}
```

Вы можете задаться вопросом, какую функцию выполняет `prison_ip()`. `prison_ip()` принимает три аргумента: указатель на учётные данные (представленные как `cred`), любые флаги и IP-адрес. Она возвращает 1, если IP-адрес НЕ принадлежит клетке, и 0 в противном случае. Как видно из кода, если это действительно IP-адрес, не принадлежащий клетке, протоколу не разрешается привязываться к этому адресу.

```
/usr/src/sys/kern/kern_jail.c:
int
prison_ip(struct ucred *cred, int flag, u_int32_t *ip)
{
    u_int32_t tmp;

    if (!jailed(cred))
        return (0);
    if (flag)
        tmp = *ip;
    else
        tmp = ntohl(*ip);
    if (tmp == INADDR_ANY) {
        if (flag)
            *ip = cred->cr_prison->pr_ip;
        else
            *ip = htonl(cred->cr_prison->pr_ip);
        return (0);
```

```
    }
    if (tmp == INADDR_LOOPBACK) {
        if (flag)
            *ip = cred->cr_prison->pr_ip;
        else
            *ip = htonl(cred->cr_prison->pr_ip);
        return (0);
    }
    if (cred->cr_prison->pr_ip != tmp)
        return (1);
    return (0);
}
```

### 4.2.5. Файловая система

Даже пользователи с правами `root` внутри `клетки` не могут снять или изменить любые флаги файлов, такие как неизменяемый, только для добавления и неудаляемый, если уровень безопасности (`securelevel`) больше 0.

```
/usr/src/sys/ufs/ufs/ufs_vnops.c:
static int
ufs_setattr(ap)
    ...
{
    ...
        if (!priv_check_cred(cred, PRIV_VFS_SYSFLAGS, 0)) {
            if (ip->i_flags
                & (SF_NOUNLINK | SF_IMMUTABLE | SF_APPEND)) {
                    error = securelevel_gt(cred, 0);
                    if (error)
                        return (error);
            }
            ...
        }
}
/usr/src/sys/kern/kern_priv.c
int
priv_check_cred(struct ucred *cred, int priv, int flags)
{
    ...
    error = prison_priv_check(cred, priv);
    if (error)
        return (error);
    ...
}
/usr/src/sys/kern/kern_jail.c
int
prison_priv_check(struct ucred *cred, int priv)
{
```

```
    ...
    switch (priv) {
    ...
    case PRIV_VFS_SYSFLAGS:
        if (jail_chflags_allowed)
            return (0);
        else
            return (EPERM);
    ...
    }
    ...
}
```

# Глава 5. Фреймворк SYSINIT

SYSINIT — это фреймворк для общего механизма сортировки и диспетчеризации вызовов. В настоящее время FreeBSD использует его для динамической инициализации ядра. SYSINIT позволяет изменять порядок, добавлять, удалять и заменять подсистемы ядра FreeBSD во время линковки ядра при загрузке ядра или его модулей, без необходимости редактировать статически упорядоченные маршруты инициализации и перекомпилировать ядро. Эта система также позволяет модулям ядра (в настоящее время называемым *KLD*) компилироваться, линковаться и инициализироваться отдельно во время загрузки, а также загружаться позже, когда система уже работает. Это достигается с помощью «компоновщика ядра» (kernel linker) и «наборов компоновщика» (linker sets).

## 5.1. Терминология

**Набор компоновщика (Linker Set)**

> Техника компоновщика, при которой компоновщик собирает статически объявленные данные из всех исходных файлов программы в единый непрерывно адресуемый блок данных.

## 5.2. Работа механизма SYSINIT

SYSINIT полагается на способность компоновщика объединять статические данные, объявленные в нескольких местах исходного кода программы, в единый непрерывный блок данных. Этот метод компоновщика называется "набором компоновщика" (linker set). SYSINIT использует два набора компоновщика для поддержки двух наборов данных, содержащих порядок вызова, функцию и указатель на данные, передаваемые этой функции для каждого члена этих наборов данных.

SYSINIT использует два приоритета для упорядочивания функций при выполнении. Первый приоритет — это идентификатор подсистемы, задающий общий порядок вызова функций SYSINIT. Предварительно объявленные идентификаторы находятся в <sys/kernel.h> в перечислении `sysinit_sub_id`. Второй используемый приоритет — это порядок элементов внутри подсистемы. Предварительно объявленные порядки элементов подсистемы находятся в <sys/kernel.h> в перечислении `sysinit_elem_order`.

В настоящее время существует два варианта использования `SYSINIT`: вызов функций при загрузке системы и загрузке модулей ядра, а также вызов функций при завершении работы системы и выгрузке модулей ядра. Подсистемы ядра часто используют `SYSINIT` при старте системы для инициализации структур данных. Например, подсистема планирования процессов использует `SYSINIT` для инициализации структуры данных очереди выполнения. Драйверы устройств должны избегать прямого использования `SYSINIT()`. Вместо этого драйверы реальных устройств, входящих в структуру шины, должны использовать `DRIVER_MODULE()`, который предоставляет функцию для обнаружения устройства и, если оно присутствует, его инициализации. Этот макрос выполняет несколько действий, специфичных для устройств, а затем вызывает `SYSINIT()` самостоятельно. Для псевдоустройств, которые не входят в структуру шины, следует использовать `DEV_MODULE()`.

# 5.3. Использование SYSINIT

## 5.3.1. Интерфейс

### Заголовки

```
<sys/kernel.h>
```

### Макросы

```
SYSINIT(uniquifier, subsystem, order, func, ident)
SYSUNINIT(uniquifier, subsystem, order, func, ident)
```

## 5.3.2. Запуск

Макрос `SYSINIT()` создаёт необходимые данные SYSINIT в наборе данных инициализации системы, чтобы SYSINIT мог отсортировать и выполнить функцию при запуске системы и загрузке модуля. `SYSINIT()` принимает уникальный идентификатор, который SYSINIT использует для идентификации конкретных данных вызова функции, порядок подсистемы, порядок элемента подсистемы, функцию для вызова и данные для передачи в функцию. Все функции должны принимать аргумент в виде константного указателя.

*Пример 1. Пример* `SYSINIT()`

```
#include <sys/kernel.h>

void foo_null(void *unused)
{
        foo_doo();
}
SYSINIT(foo, SI_SUB_FOO, SI_ORDER_FOO, foo_null, NULL);

struct foo foo_voodoo = {
        FOO_VOODOO;
}

void foo_arg(void *vdata)
{
        struct foo *foo = (struct foo *)vdata;
        foo_data(foo);
}
SYSINIT(bar, SI_SUB_FOO, SI_ORDER_FOO, foo_arg, &foo_voodoo);
```

Обратите внимание, что `SI_SUB_FOO` и `SI_ORDER_FOO` должны быть в перечислениях `sysinit_sub_id` и `sysinit_elem_order`, как упоминалось выше. Можно использовать

существующие значения или добавить свои в эти перечисления. Также можно использовать математические операции для точной настройки порядка выполнения SYSINIT. В этом примере показан SYSINIT, который должен выполняться непосредственно перед SYSINIT, обрабатывающими настройку параметров ядра.

*Пример 2. Пример настройки порядка* `SYSINIT()`

```
static void
mptable_register(void *dummy __unused)
{

    apic_register_enumerator(&mptable_enumerator);
}

SYSINIT(mptable_register, SI_SUB_TUNABLES - 1, SI_ORDER_FIRST,
    mptable_register, NULL);
```

### 5.3.3. Выключение системы

Макрос `SYSUNINIT()` ведёт себя аналогично макросу `SYSINIT()`, за исключением того, что добавляет данные SYSINIT в набор данных завершения работы SYSINIT.

*Пример 3. Пример* `SYSUNINIT()`

```
#include <sys/kernel.h>

void foo_cleanup(void *unused)
{
        foo_kill();
}
SYSUNINIT(foobar, SI_SUB_FOO, SI_ORDER_FOO, foo_cleanup, NULL);

struct foo_stack foo_stack = {
        FOO_STACK_VOODOO;
}

void foo_flush(void *vdata)
{
}
SYSUNINIT(barfoo, SI_SUB_FOO, SI_ORDER_FOO, foo_flush, &foo_stack);
```

# Глава 6. Фреймворк TrustedBSD MAC

## 6.1. Авторские права документации MAC

Этот документ был разработан для проекта FreeBSD Крисом Костелло из Safeport Network Services и Network Associates Laboratories, подразделения исследований безопасности Network Associates, Inc., по контракту DARPA/SPAWAR N66001-01-C-8035 ("CBOSS") в рамках исследовательской программы DARPA CHATS.

Redistribution and use in source (SGML DocBook) and 'compiled' forms (SGML, HTML, PDF, PostScript, RTF and so forth) with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code (SGML DocBook) must retain the above copyright notice, this list of conditions and the following disclaimer as the first lines of this file unmodified.
2. Распространение в скомпилированной форме (преобразованное в другие DTD, конвертированное в PDF, PostScript, RTF и другие форматы) должно включать указанное выше уведомление об авторских правах, данный список условий и следующий отказ от ответственности в документации и/или других материалах, предоставляемых вместе с распространением.



ЭТА ДОКУМЕНТАЦИЯ ПРЕДОСТАВЛЯЕТСЯ NETWORKS ASSOCIATES TECHNOLOGY, INC «КАК ЕСТЬ», И ЛЮБЫЕ ЯВНЫЕ ИЛИ ПОДРАЗУМЕВАЕМЫЕ ГАРАНТИИ, ВКЛЮЧАЯ, НО НЕ ОГРАНИЧИВАЯСЬ ИМИ, ПОДРАЗУМЕВАЕМЫЕ ГАРАНТИИ ТОВАРНОЙ ПРИГОДНОСТИ И ПРИГОДНОСТИ ДЛЯ ОПРЕДЕЛЕННОЙ ЦЕЛИ ОТРИЦАЮТСЯ. НИ ПРИ КАКИХ ОБСТОЯТЕЛЬСТВАХ NETWORKS ASSOCIATES TECHNOLOGY, INC НЕ НЕСЕТ ОТВЕТСТВЕННОСТИ ЗА ЛЮБЫЕ ПРЯМЫЕ, КОСВЕННЫЕ, СЛУЧАЙНЫЕ, СПЕЦИАЛЬНЫЕ, ШТРАФНЫЕ ИЛИ КОСВЕННЫЕ УБЫТКИ (ВКЛЮЧАЯ, НО НЕ ОГРАНИЧИВАЯСЬ ИМИ, ЗАТРАТЫ НА ЗАМЕНУ ТОВАРОВ ИЛИ УСЛУГ; ПОТЕРЮ ИСПОЛЬЗОВАНИЯ, ДАННЫХ ИЛИ ПРИБЫЛИ; ЛИБО ПРЕРЫВАНИЕ БИЗНЕСА), ВЫЗВАННЫЕ ЛЮБЫМ ОБРАЗОМ И НА ОСНОВАНИИ ЛЮБОЙ ТЕОРИИ ОТВЕТСТВЕННОСТИ, БУДЬ ТО В РАМКАХ ДОГОВОРА, СТРОГОЙ ОТВЕТСТВЕННОСТИ ИЛИ ДЕЛИКТА (ВКЛЮЧАЯ НЕБРЕЖНОСТЬ ИЛИ ИНОЕ), ВОЗНИКШИЕ ВСЛЕДСТВИЕ ИСПОЛЬЗОВАНИЯ ЭТОЙ ДОКУМЕНТАЦИИ, ДАЖЕ ЕСЛИ БЫЛО ПРЕДУПРЕЖДЕНИЕ О ВОЗМОЖНОСТИ ТАКИХ УБЫТКОВ.

## 6.2. Обзор

FreeBSD включает экспериментальную поддержку нескольких политик обязательного контроля доступа, а также инфраструктуру для расширяемости безопасности ядра — TrustedBSD MAC Framework. MAC Framework представляет собой модульную инфраструктуру контроля доступа, позволяющую легко встраивать новые политики безопасности в ядро, загружать их при старте системы или динамически во время работы. Инфраструктура предоставляет множество возможностей для упрощения реализации новых политик безопасности, включая возможность легко присваивать метки безопасности (например,

информацию о конфиденциальности) объектам системы.

Эта глава представляет фреймворк политик MAC и содержит документацию для образца модуля политики MAC.

## 6.3. Введение

Фреймворк TrustedBSD MAC предоставляет механизм для расширения модели контроля доступа ядра во время компиляции или выполнения. Новые политики системы могут быть реализованы в виде модулей ядра и связаны с ним; если присутствуют несколько модулей политик, их результаты будут объединены. Фреймворк MAC предоставляет различные инфраструктурные сервисы контроля доступа для помощи разработчикам политик, включая поддержку временных и постоянных меток безопасности объектов, не зависящих от политик. В настоящее время эта поддержка считается экспериментальной.

Эта глава предоставляет информацию, предназначенную для разработчиков модулей политик, а также потенциальных пользователей сред с поддержкой MAC, чтобы узнать о том, как MAC Framework поддерживает расширение контроля доступа в ядре.

## 6.4. Общие сведения о политиках

Мандатное управление доступом (MAC — Mandatory Access Control) относится к набору политик контроля доступа, которые в обязательном порядке применяются операционной системой к пользователям. Политики MAC можно противопоставить защите на основе дискреционного управления доступом (DAC — Discretionary Access Control), при которой непривилегированные пользователи могут (по своему усмотрению) защищать объекты. В традиционных UNIX-системах защита DAC включает права доступа к файлам и списки контроля доступа; защита MAC включает управление процессами, предотвращающее отладку между пользователями, и межсетевые экраны. Различные политики MAC были разработаны создателями операционных систем и исследователями безопасности, включая политику конфиденциальности многоуровневой безопасности (MLS — Multi-Level Security), политику целостности Biba, управление доступом на основе ролей (RBAC — Role-Based Access Control), принудительное применение доменов и типов (DTE — Domain and Type Enforcement) и принудительное применение типов (TE — Type Enforcement). Каждая модель основывает решения на различных факторах, включая идентификатор пользователя, роль и уровень доступа, а также метки безопасности на объектах, представляющих такие концепции, как конфиденциальность и целостность данных.

Фреймворк TrustedBSD MAC способен поддерживать модули политик, реализующие все эти политики, а также широкий класс политик усиления защиты системы, которые могут использовать существующие атрибуты безопасности, такие как идентификаторы пользователей и групп, а также расширенные атрибуты файлов и другие свойства системы. Кроме того, несмотря на название, фреймворк MAC также может использоваться для реализации чисто дискреционных политик, поскольку модулям политик предоставляется значительная гибкость в том, как они авторизуют защиту.

# 6.5. Архитектура MAC Framework в ядре

Фреймворк TrustedBSD MAC позволяет модулям ядра расширять политику безопасности операционной системы, а также предоставляет функциональность инфраструктуры, необходимую многим модулям контроля доступа. Если одновременно загружено несколько политик, фреймворк MAC полезным образом (в некотором смысле полезным) объединит результаты этих политик.

## 6.5.1. Элементы ядра

В рамках MAC Framework реализован ряд элементов ядра:

- Интерфейсы управления фреймворком
- Параллелизм и примитивы синхронизации.
- Регистрация политики
- Расширяемая метка безопасности для объектов ядра
- Операторы композиции точки входа политики
- Примитивы управления метками
- Точка входа API, вызываемая службами ядра
- Точка входа API для модулей политик
- Реализации точек входа (жизненный цикл политики, жизненный цикл объекта/управление метками, проверки контроля доступа).
- Системные вызовы, независимые от политик, для управления метками
- `mac_syscall()` мультиплексный системный вызов
- Различные политики безопасности, реализованные в виде модулей политики MAC

## 6.5.2. Интерфейсы управления фреймворком

Фреймворком TrustedBSD MAC можно напрямую управлять с помощью sysctl, параметров загрузчика и системных вызовов.

В большинстве случаев одноимённые параметры sysctl и настройки загрузчика изменяют одни и те же параметры и управляют поведением, таким как применение защитных механизмов, связанных с различными подсистемами ядра. Кроме того, если в ядро включена поддержка отладки MAC, будет вестись несколько счётчиков для отслеживания выделения меток. Обычно рекомендуется не использовать общие настройки подсистем для управления поведением политик в рабочих средах, так как они широко влияют на работу всех активных политик. Вместо этого следует предпочитать настройки отдельных политик, поскольку они обеспечивают более высокую детализацию и большую операционную согласованность для модулей политик.

Загрузка и выгрузка модулей политики выполняется с использованием системных вызовов управления модулями и других системных интерфейсов, включая переменные загрузчика; модули политики получат возможность влиять на события загрузки и выгрузки, включая

предотвращение нежелательной выгрузки политики.

### 6.5.3. Список политик параллелизма и синхронизации

Поскольку набор активных политик может изменяться во время выполнения, а вызов точек входа не является атомарным, требуется синхронизация для предотвращения загрузки или выгрузки политик во время выполнения вызова точки входа, фиксируя набор активных политик на время выполнения. Это достигается с помощью счетчика занятости фреймворка: при входе в точку входа счетчик увеличивается; при выходе из неё — уменьшается. Пока счетчик занятости повышен, изменения списка политик запрещены, и потоки, пытающиеся изменить список политик, будут ждать, пока список не освободится. Счетчик занятости защищается мьютексом, а условная переменная используется для пробуждения потоков, ожидающих изменений списка политик. Побочным эффектом этой модели синхронизации является то, что рекурсивный вход в MAC Framework из модуля политики разрешен, хотя обычно не используется.

Для снижения накладных расходов счётчика занятости используются различные оптимизации, включая избегание полной стоимости увеличения и уменьшения, если список пуст или содержит только статические записи (политики, загруженные до старта системы, которые нельзя выгрузить). Также предоставляется опция на этапе компиляции, которая предотвращает любые изменения в наборе загруженных политик во время выполнения, что устраняет затраты на блокировку мьютексов, связанные с поддержкой динамически загружаемых и выгружаемых политик, поскольку синхронизация больше не требуется.

Поскольку MAC Framework не может блокировать некоторые точки входа, обычная блокировка сна не может быть использована; в результате попытка загрузки или выгрузки может блокироваться на значительное время, ожидая, пока фреймворк станет свободным.

### 6.5.4. Синхронизация меток

Поскольку к объектам ядра обычно может обращаться более одного потока одновременно, и допускается одновременный вход нескольких потоков в MAC Framework, хранение атрибутов безопасности, поддерживаемое MAC Framework, тщательно синхронизировано. Как правило, существующая синхронизация ядра для данных объектов ядра используется для защиты меток безопасности MAC Framework на объекте: например, метки MAC на сокетах защищаются с помощью существующего мьютекса сокета. Аналогично, семантика параллельного доступа обычно идентична семантике контейнерных объектов: для учётных данных поддерживается семантика копирования при записи для содержимого меток, как и для остальной структуры учётных данных. MAC Framework устанавливает необходимые блокировки на объекты при вызове с ссылкой на объект. Авторам политик необходимо учитывать эти семантики синхронизации, так как они иногда ограничивают типы доступа к меткам: например, когда ссылка только для чтения на учётные данные передаётся политике через точку входа, разрешены только операции чтения для состояния метки, прикрепленного к учётным данным.

### 6.5.5. Синхронизация политики и параллелизм

Модули политик должны быть написаны с учётом того, что множество потоков ядра могут одновременно войти в одну или несколько точек входа политики из-за параллельной и вытесняющей природы ядра FreeBSD. Если модуль политики использует изменяемое состояние, это может потребовать применения примитивов синхронизации внутри политики, чтобы предотвратить несогласованные представления этого состояния, ведущие к некорректной работе политики. Политики, как правило, могут использовать существующие примитивы синхронизации FreeBSD для этой цели, включая мьютексы, блокировки с ожиданием, условные переменные и счётные семафоры. Однако политики должны быть написаны так, чтобы применять эти примитивы осторожно, соблюдая существующие порядки блокировок в ядре и учитывая, что некоторые точки входа не допускают ожидания, ограничивая использование примитивов в этих точках входа мьютексами и операциями пробуждения.

Когда модули политики обращаются к другим подсистемам ядра, они обычно должны освобождать любые блокировки внутри политики, чтобы избежать нарушения порядка блокировок ядра или риска рекурсивных блокировок. Это позволит сохранить блокировки политики как конечные блокировки в глобальном порядке блокировок, помогая избежать взаимоблокировки.

### 6.5.6. Регистрация политики

Фреймворк MAC поддерживает два списка активных политик: статический список и динамический список. Списки отличаются только в отношении их семантики блокировки: для использования статического списка не требуется повышенный счетчик ссылок. Когда загружаются модули ядра, содержащие политики фреймворка MAC, модуль политики использует `SYSINIT` для вызова функции регистрации; когда модуль политики выгружается, `SYSINIT` аналогично вызывает функцию отмены регистрации. Регистрация может завершиться неудачей, если модуль политики загружается более одного раза, если для регистрации недостаточно ресурсов (например, политика может требовать маркировки, а доступного состояния маркировки может быть недостаточно), или другие предварительные условия политики могут не выполняться (некоторые политики могут быть загружены только до загрузки системы). Аналогично, отмена регистрации может завершиться неудачей, если политика помечена как невыгружаемая.

### 6.5.7. Точки входа

Ядро взаимодействует с MAC Framework двумя способами: вызывает набор API для уведомления фреймворка о соответствующих событиях и предоставляет указатель на структуру меток, не зависящую от политики, в объектах, связанных с безопасностью. Указатель метки управляется MAC Framework через точки входа управления метками, что позволяет фреймворку предоставлять службу маркировки модулям политик с относительно минимальными изменениями в подсистеме ядра, управляющей объектом. Например, указатели меток были добавлены к процессам, учётным данным процессов, сокетам, каналам, vnode, Mbuf, сетевым интерфейсам, очередям сборки IP-пакетов и множеству других структур, связанных с безопасностью. Ядро также вызывает MAC Framework при принятии важных решений по безопасности, позволяя модулям политик

дополнять эти решения на основе собственных критериев (включая, возможно, данные, хранящиеся в метках безопасности). Большинство этих критически важных решений по безопасности будут явными проверками контроля доступа; однако некоторые влияют на более общие функции принятия решений, такие как сопоставление пакетов для сокетов и переход меток при выполнении программы.

### 6.5.8. Композиция политик

Когда в ядро загружено более одного модуля политики одновременно, результаты работы модулей политики будут объединены фреймворком с использованием оператора композиции. Этот оператор в настоящее время жёстко закодирован и требует, чтобы все активные политики одобрили запрос для возврата успешного результата. Поскольку политики могут возвращать различные условия ошибки (успех, доступ запрещён, объект не существует, …), оператор старшинства выбирает результирующую ошибку из набора ошибок, возвращаемых политиками. В общем случае, ошибки, указывающие на то, что объект не существует, будут предпочтительнее ошибок, указывающих на запрет доступа к объекту. Хотя не гарантируется, что результирующая композиция будет полезной или безопасной, мы обнаружили, что это так для многих полезных наборов политик. Например, традиционные доверенные системы часто поставляются с двумя или более политиками, использующими аналогичную композицию.

### 6.5.9. Поддержка меток

Поскольку многие интересные расширения контроля доступа зависят от меток безопасности объектов, MAC Framework предоставляет набор системных вызовов для управления метками, не зависящих от политик, охватывающих различные объекты, доступные пользователю. Общие типы меток включают идентификаторы разделов, метки конфиденциальности, метки целостности, компартменты (compartment), домены, роли и типы. Под "не зависящими от политик" подразумевается, что модули политик могут полностью определять семантику метаданных, связанных с объектом. Модули политик участвуют в интернализации и экстернализации строковых меток, предоставляемых пользовательскими приложениями, и могут при необходимости предоставлять приложениям несколько элементов меток.

Метки в памяти хранятся в `struct label`, выделяемой через slab-аллокатор. Эта структура состоит из массива фиксированной длины, содержащего объединения, каждое из которых хранит указатель `void *` и значение типа `long`. Политикам, регистрирующим хранилище меток, назначается идентификатор "слота", который может использоваться для разыменования хранилища меток. Семантика хранилища полностью определяется модулем политики: модулям предоставляется набор точек входа, связанных с жизненным циклом объектов ядра, включая инициализацию, связывание/создание и уничтожение. Используя эти интерфейсы, можно реализовать подсчёт ссылок и другие модели хранения. Прямой доступ к структуре объекта, как правило, не требуется модулям политики для получения метки, поскольку MAC Framework обычно передаёт в точки входа как указатель на объект, так и прямой указатель на метку объекта. Основным исключением из этого правила являются учётные данные процесса, для доступа к метке которых требуется ручное разыменование. Это может измениться в будущих версиях MAC Framework.

Входные точки инициализации часто включают флаг режима сна, указывающий, разрешено ли инициализации переходить в режим сна; если сон не разрешен, может быть возвращена ошибка для отмены выделения метки (и, следовательно, объекта). Это может произойти, например, в сетевом стеке во время обработки прерывания, где сон не разрешен, или пока вызывающий удерживает мьютекс. Из-за затрат производительности на поддержание меток на передаваемых сетевых пакетах (Mbuf), политики должны явно объявлять требование о выделении меток для Mbuf. Динамически загружаемые политики, использующие метки, должны быть способны обрабатывать случай, когда их функция инициализации не была вызвана для объекта, так как объекты могут уже существовать при загрузке политики. MAC Framework гарантирует, что неинициализированные слоты меток будут содержать значение 0 или NULL, что политики могут использовать для обнаружения неинициализированных значений. Однако, поскольку выделение меток для Mbuf условно, политики также должны быть способны обрабатывать указатель на метку NULL для Mbuf, если они были загружены динамически.

В случае меток файловых систем предусмотрена специальная поддержка для постоянного хранения меток безопасности в расширенных атрибутах. Там, где это возможно, используются транзакции расширенных атрибутов, чтобы обеспечить согласованные составные обновления меток безопасности на vnode — в настоящее время такая поддержка присутствует только в файловой системе UFS2. Авторы политик могут выбрать реализацию многометочных меток объектов файловой системы с использованием одного (или нескольких) расширенных атрибутов. По соображениям эффективности метка vnode (`v_label`) является кэшем любой метки на диске; политики могут загружать значения в кэш при создании vnode и обновлять кэш по мере необходимости. В результате нет необходимости напрямую обращаться к расширенному атрибуту при каждой проверке контроля доступа.



В настоящее время, если помеченная политика разрешает динамическую выгрузку, её слот состояния не может быть освобождён, что накладывает строгое (и относительно низкое) ограничение на количество операций выгрузки-перезагрузки для помеченных политик.

### 6.5.10. Системные вызовы

В рамках MAC Framework реализован ряд системных вызовов: большинство из них поддерживают API для получения и управления метками, не зависящий от политики и доступный пользовательским приложениям.

Вызовы управления метками принимают структуру описания метки `struct mac`, которая содержит серию элементов метки MAC. Каждый элемент содержит строку с именем и строку со значением. Каждой политике будет предоставлена возможность запросить определённое имя элемента, позволяя политикам предоставлять несколько независимых элементов, если это необходимо. Модули политик выполняют интернализацию и экстернализацию между метками ядра и метками, предоставленными пользователем, через точки входа, что позволяет использовать различные семантики. Системные вызовы управления метками обычно обёрнуты в функции пользовательской библиотеки для выполнения выделения памяти и обработки ошибок, упрощая пользовательские приложения, которые должны управлять метками.

В ядре FreeBSD есть следующие системные вызовы, связанные с MAC:

- `mac_get_proc()` может использоваться для получения метки текущего процесса.
- `mac_set_proc()` может использоваться, чтобы запросить изменение метки текущего процесса.
- `mac_get_fd()` может использоваться для получения метки объекта (файл, сокет, канал, ...), на который ссылается файловый дескриптор.
- `mac_get_file()` может использоваться для получения метки объекта, на который ссылается путь в файловой системе.
- `mac_set_fd()` может использоваться для запроса изменения метки объекта (файл, сокет, канал, ...), на который ссылается файловый дескриптор.
- `mac_set_file()` может использоваться для запроса изменения метки объекта, указанного по пути в файловой системе.
- `mac_syscall()` позволяет модулям политик создавать новые системные вызовы без изменения таблицы системных вызовов; она принимает имя целевой политики, номер операции и непрозрачный аргумент для использования политикой.
- `mac_get_pid()` может использоваться для запроса метки другого процесса по его идентификатору.
- `mac_get_link()` идентична `mac_get_file()`, но не переходит по символической ссылке, если она является конечным элементом пути, поэтому может использоваться для получения метки на символьной ссылке.
- `mac_set_link()` идентична `mac_set_file()`, за исключением того, что она не следует по символической ссылке, если это конечный элемент пути, поэтому может использоваться для изменения метки на символьной ссылке.
- `mac_execve()` идентична системному вызову `execve()`, но также принимает запрошенную метку, которая будет установлена для процесса при начале выполнения новой программы. Это изменение метки при выполнении называется "переходом".
- `mac_get_peer()`, фактически реализованный через параметр сокета, извлекает метку удалённого узла на сокете, если она доступна.

В дополнение к этим системным вызовам, сетевые ioctl-команды `SIOCSIGMAC` и `SIOCSIFMAC` позволяют получать и устанавливать метки на сетевых интерфейсах.

## 6.6. Архитектура политик MAC

Политики безопасности либо непосредственно встроены в ядро, либо скомпилированы в загружаемые модули ядра, которые могут быть загружены при загрузке системы или динамически с использованием системных вызовов загрузки модулей во время выполнения. Модули политик взаимодействуют с системой через набор объявленных точек входа, предоставляя доступ к потоку системных событий и позволяя политике влиять на решения контроля доступа. Каждая политика содержит ряд элементов:

- Необязательные параметры конфигурации для политики.

- Централизованная реализация логики политики и параметров.
- Необязательная реализация событий жизненного цикла политики, таких как инициализация и уничтожение.
- Необязательная поддержка инициализации, обслуживания и удаления меток на выбранных объектах ядра.
- Дополнительная поддержка проверки процессов пользователя и изменения меток на выбранных объектах.
- Реализация выбранных точек входа контроля доступа, представляющих интерес для политики.
- Объявление идентификатора политики, точек входа модуля и свойств политики.

### 6.6.1. Объявление политики

Модули могут быть объявлены с использованием макроса `MAC_POLICY_SET()`, который задаёт имя политики, предоставляет ссылку на вектор точек входа MAC, указывает флаги загрузки, определяющие, как фреймворк политик должен обрабатывать политику, и при необходимости запрашивает выделение состояния метки фреймворком.

```
static struct mac_policy_ops mac_policy_ops =
{
        .mpo_destroy = mac_policy_destroy,
        .mpo_init = mac_policy_init,
        .mpo_init_bpfdesc_label = mac_policy_init_bpfdesc_label,
        .mpo_init_cred_label = mac_policy_init_label,
/* ... */
        .mpo_check_vnode_setutimes = mac_policy_check_vnode_setutimes,
        .mpo_check_vnode_stat = mac_policy_check_vnode_stat,
        .mpo_check_vnode_write = mac_policy_check_vnode_write,
};
```

Вектор точек входа политики MAC, `macpolicyops` в данном примере, связывает функции, определённые в модуле, с конкретными точками входа. Полный список доступных точек входа и их прототипов можно найти в разделе справочника по точкам входа MAC. Особый интерес при регистрации модуля представляют точки входа .mpo_destroy и .mpo_init. .mpo_init будет вызван после успешной регистрации политики в модульной системе, но до активации других точек входа. Это позволяет политике выполнять любые выделения и инициализации, специфичные для данной политики, такие как инициализация данных или блокировок. .mpo_destroy будет вызван при выгрузке модуля политики для освобождения выделенной памяти и уничтожения блокировок. В настоящее время эти две точки входа вызываются с удержанием мьютекса списка политик MAC, чтобы предотвратить вызов других точек входа: это будет изменено, но до тех пор политики должны быть осторожны с используемыми примитивами ядра, чтобы избежать проблем с порядком блокировок или сном.

Поле имени модуля в объявлении политики существует для того, чтобы модуль мог быть однозначно идентифицирован с целью управления зависимостями модулей. Следует

выбрать подходящую строку. Полное имя политики отображается пользователю в журнале ядра при загрузке и выгрузке, а также экспортируется при предоставлении информации о статусе процессам в пользовательском пространстве.

### 6.6.2. Флаги политик

Поле флагов объявления политики позволяет модулю предоставлять фреймворку информацию о своих возможностях во время загрузки модуля. В настоящее время определены три флага:

**MPC_LOADTIME_FLAG_UNLOADOK**

Этот флаг указывает, что модуль политики может быть выгружен. Если этот флаг не указан, то фреймворк политики отклонит запросы на выгрузку модуля. Этот флаг может использоваться модулями, которые выделяют состояние метки и не могут освободить это состояние во время выполнения.

**MPC_LOADTIME_FLAG_NOTLATE**

Этот флаг указывает, что модуль политики должен быть загружен и инициализирован на раннем этапе процесса загрузки. Если флаг указан, попытки зарегистрировать модуль после загрузки будут отклонены. Флаг может использоваться политиками, которые требуют повсеместной маркировки всех системных объектов и не могут обрабатывать объекты, не прошедшие надлежащую инициализацию политикой.

**MPC_LOADTIME_FLAG_LABELMBUFS**

Этот флаг указывает, что модуль политики требует маркировки Mbuf, и память всегда должна выделяться для хранения меток Mbuf. По умолчанию MAC Framework не выделяет память для хранения меток Mbuf, если хотя бы одна загруженная политика не установила этот флаг. Это заметно улучшает производительность сети, когда политики не требуют маркировки Mbuf. Существует опция ядра `MAC_ALWAYS_LABEL_MBUF`, которая заставляет MAC Framework выделять память для хранения меток Mbuf независимо от установки этого флага, и может быть полезной в некоторых средах.



Политики, использующие `MPC_LOADTIME_FLAG_LABELMBUFS` без установленного флага `MPC_LOADTIME_FLAG_NOTLATE`, должны корректно обрабатывать переданные `NULL` указатели меток Mbuf в точках входа. Это необходимо, так как Mbuf в процессе передачи без хранилища меток могут сохраняться после загрузки политики, включающей маркировку Mbuf. Если политика загружена до активации сетевой подсистемы (т.е. политика не загружается поздно), то все Mbuf гарантированно имеют хранилище меток.

### 6.6.3. Точки входа политики

Четыре класса точек входа предоставляются политикам, зарегистрированным в рамках системы: точки входа, связанные с регистрацией и управлением политиками, точки входа, обозначающие инициализацию, создание, уничтожение и другие события жизненного цикла объектов ядра, события, связанные с решениями контроля доступа, на которые политика может влиять, и вызовы, связанные с управлением метками на объектах. Кроме того, предоставляется точка входа `mac_syscall()`, позволяющая политикам расширять

интерфейс ядра без регистрации новых системных вызовов.

Авторы модулей политик должны быть осведомлены о стратегии блокировок в ядре, а также о том, какие блокировки объектов доступны на различных точках входа. Им следует избегать сценариев взаимоблокировок, не захватывая нелистовые блокировки внутри точек входа, а также соблюдать протокол блокировок для доступа и изменения объектов. В частности, авторы должны учитывать, что хотя необходимые блокировки для доступа к объектам и их меткам обычно удерживаются, достаточные блокировки для изменения объекта или его метки могут отсутствовать для всех точек входа. Информация о блокировках аргументов документирована в описании точек входа фреймворка MAC.

Точки входа политики будут передавать ссылку на метку объекта вместе с самим объектом. Это позволяет помеченным политикам не знать внутренней структуры объекта, но при этом принимать решения на основе метки. Исключением из этого являются учётные данные процесса, для которые предполагается, что политики понимают их, как объект безопасности первого класса в ядре.

# 6.7. Справочник по точкам входа политики MAC

## 6.7.1. Общие точки входа модуля

`mpo_init`

```
void mpo_init(struct mac_policy_conf *conf);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `conf` | Определение политики MAC | |

Событие загрузки политики. Мьютекс списка политик удерживается, поэтому операции ожидания выполнить нельзя, а вызовы других подсистем ядра должны осуществляться с осторожностью. Если во время инициализации политики требуются потенциально блокирующие выделения памяти, их следует выполнять с использованием отдельного модуля SYSINIT().

`mpo_destroy`

```
void mpo_destroy(struct mac_policy_conf *conf);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `conf` | Определение политики MAC | |

Событие загрузки политики. Мьютекс списка политик удерживается, поэтому следует соблюдать осторожность.

`mpo_syscall`

```
int mpo_syscall(struct thread *td, int call, void *arg);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `td` | Вызывающий поток | |
| `call` | Номер системного вызова, зависящий от политики | |
| `arg` | Указатель на аргументы системного вызова | |

Этот точку входа предоставляет мультиплексированный системный вызов на основе политик, что позволяет политикам предоставлять дополнительные сервисы пользовательским процессам без регистрации конкретных системных вызовов. Имя политики, указанное при регистрации, используется для демультиплексирования вызовов из пользовательского пространства, а аргументы будут переданы в эту точку входа. При реализации новых сервисов модули безопасности должны убедиться, что вызывают соответствующие проверки контроля доступа из MAC-фреймворка по мере необходимости. Например, если политика реализует расширенную функциональность сигналов, она должна вызывать необходимые проверки контроля доступа сигналов для задействования MAC-фреймворка и других зарегистрированных политик.



Модули в настоящее время должны самостоятельно выполнять `copyin()` для данных системного вызова.

`mpo_thread_userret`

```
void mpo_thread_userret(struct thread *td);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `td` | Возвращающий поток | |

Этот точка входа позволяет модулям политики выполнять события, связанные с MAC, когда поток возвращается в пользовательское пространство, через возврат системного вызова, возврат из ловушки или иным образом. Это необходимо для политик, имеющих плавающие метки процессов, так как не всегда возможно получить блокировку процесса в произвольных точках стека во время обработки системного вызова; метки процессов могут представлять традиционные данные аутентификации, информацию об истории процесса или другие данные. Для использования этого механизма предполагаемые изменения метки учётных данных процесса могут быть сохранены в `p_label`, защищённом спин-блокировкой для каждой политики, а затем установить флаг `TDF_ASTPENDING` для потока и флаг `PS_MACPENDM` для процесса, чтобы запланировать вызов точки входа `userret`. С этой точки входа политика может создать замену учётных данных с меньшими опасениями относительно контекста блокировки. Авторам политик следует учитывать, что порядок событий, связанных с планированием AST и выполнением AST, может быть сложным и переплетённым в

многопоточных приложениях.

### 6.7.2. Операции с метками

`mpo_init_bpfdesc_label`

```
void mpo_init_bpfdesc_label(struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `label` | Новая метка для инициализации | |

Инициализировать метку на только что созданном bpfdesc (дескрипторе BPF). Разрешено использование режима сна.

`mpo_init_cred_label`

```
void mpo_init_cred_label(struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `label` | Новая метка для инициализации | |

Инициализировать метку для вновь созданных учётных данных пользователя. Разрешено приостанавливать выполнение.

`mpo_init_devfsdirent_label`

```
void mpo_init_devfsdirent_label(struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `label` | Новая метка для инициализации | |

Инициализировать метку на только что созданной записи devfs. Разрешено использование режима сна.

`mpo_init_ifnet_label`

```
void mpo_init_ifnet_label(struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| label | Новая метка для инициализации | |

Инициализировать метку на только что созданном сетевом интерфейсе. Разрешено приостанавливать выполнение.

### mpo_init_ipq_label

```
void mpo_init_ipq_label(struct label *label, int flag);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| label | Новая метка для инициализации | |
| flag | Спящий/неспящий malloc(9); см. ниже | |

Инициализировать метку в только что созданной очереди сборки IP-фрагментов. Поле flag может принимать одно из значений M_WAITOK или M_NOWAIT и должно использоваться, чтобы избежать выполнения "спящего" malloc(9) во время этого вызова инициализации. Выделение очереди сборки IP-фрагментов часто происходит в средах, чувствительных к производительности, и реализация должна избегать "спящих" или длительных операций. Этой точке входа разрешено завершаться неудачей, что приведёт к невозможности выделения очереди сборки IP-фрагментов.

### mpo_init_mbuf_label

```
void mpo_init_mbuf_label(int flag, struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| flag | Спящий/неспящий malloc(9); см. ниже | |
| label | Метка политики для инициализации | |

Инициализировать на только что созданном заголовке пакета mbuf метку (mbuf). Поле flag может принимать одно из значений M_WAITOK или M_NOWAIT и должно использоваться, чтобы избежать выполнения "спящего" malloc(9) во время этого вызова инициализации. Выделение mbuf часто происходит в чувствительных к производительности средах, и реализация должна избегать "спящего" режима или длительных операций. Этой точке входа разрешено завершаться неудачей, что приведёт к невозможности выделения заголовка mbuf.

`mpo_init_mount_label`

```
void mpo_init_mount_label(struct label *mntlabel, struct label *fslabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `mntlabel` | Метка политики для инициализации самой точки монтирования | |
| `fslabel` | Метка политики для инициализации файловой системы | |

Инициализировать метки на новой точке монтирования. Разрешено приостанавливать выполнение.

`mpo_init_mount_fs_label`

```
void mpo_init_mount_fs_label(struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `label` | Метка для инициализации | |

Инициализировать метку на только что смонтированной файловой системе. Разрешено приостановление работы

`mpo_init_pipe_label`

```
void mpo_init_pipe_label(struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `label` | Метка для заполнения | |

Инициализировать метку для только что созданного канала. Разрешено приостановление выполнения.

`mpo_init_socket_label`

```
void mpo_init_socket_label(struct label *label, int flag);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `label` | Новая метка для инициализации | |

| Параметр | Описание | Блокировка |
|---|---|---|
| `flag` | флаги malloc(9) | |

Инициализировать метку для нового сокета. Поле `flag` может принимать одно из значений M_WAITOK или M_NOWAIT и должно использоваться, чтобы избежать выполнения спящего malloc(9) во время этого вызова инициализации.

### `mpo_init_socket_peer_label`

```
void mpo_init_socket_peer_label(struct label *label, int flag);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `label` | Новая метка для инициализации | |
| `flag` | флаги malloc(9) | |

Инициализировать метку однорангового узла (peer) для вновь созданного сокета. Поле `flag` может принимать одно из значений M_WAITOK или M_NOWAIT и должно использоваться для избежания выполнения спящего malloc(9) во время этого вызова инициализации.

### `mpo_init_proc_label`

```
void mpo_init_proc_label(struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `label` | Новая метка для инициализации | |

Инициализировать метку для вновь созданного процесса. Разрешено приостановление выполнения.

### `mpo_init_vnode_label`

```
void mpo_init_vnode_label(struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `label` | Новая метка для инициализации | |

Инициализировать метку на только что созданном vnode. Разрешено приостанавливать выполнение.

### mpo_destroy_bpfdesc_label

```
void mpo_destroy_bpfdesc_label(struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| label | bpfdesc label | |

Уничтожить метку на дескрипторе BPF. В этой точке входа политика должна освободить любое внутреннее хранилище, связанное с label, чтобы её можно было уничтожить.

### mpo_destroy_cred_label

```
void mpo_destroy_cred_label(struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| label | Метка уничтожается | |

Уничтожить метку на учётных данных. В этой точке входа модуль политики должен освободить любое внутреннее хранилище, связанное с label, чтобы её можно было уничтожить.

### mpo_destroy_devfsdirent_label

```
void mpo_destroy_devfsdirent_label(struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| label | Метка уничтожается | |

Уничтожить метку на записи devfs. В этой точке входа модуль политики должен освободить любое внутреннее хранилище, связанное с label, чтобы её можно было уничтожить.

### mpo_destroy_ifnet_label

```
void mpo_destroy_ifnet_label(struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| label | Метка уничтожается | |

Уничтожить метку на удалённом интерфейсе. В этой точке входа модуль политики должен освободить любое внутреннее хранилище, связанное с label, чтобы её можно было уничтожить.

### mpo_destroy_ipq_label

```
void mpo_destroy_ipq_label(struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| label | Метка уничтожается | |

Уничтожить метку в очереди IP-фрагментов. В этой точке входа модуль политики должен освободить любое внутреннее хранилище, связанное с label, чтобы её можно было уничтожить.

### mpo_destroy_mbuf_label

```
void mpo_destroy_mbuf_label(struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| label | Метка уничтожается | |

Уничтожить метку в заголовке mbuf. В этой точке входа модуль политики должен освободить любое внутреннее хранилище, связанное с label, чтобы её можно было уничтожить.

### mpo_destroy_mount_label

```
void mpo_destroy_mount_label(struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| label | Точка монтирования метки уничтожается | |

Уничтожить метки на точке монтирования. В этой точке входа модуль политики должен освободить внутреннее хранилище, связанное с mntlabel, чтобы её можно было уничтожить.

### mpo_destroy_mount_label

```
void mpo_destroy_mount_label(struct label *mntlabel, struct label *fslabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| mntlabel | Точка монтирования метки уничтожается | |
| fslabel | Метка файловой системы уничтожается | |

Уничтожить метки на точке монтирования. В этой точке входа модуль политики должен освободить внутреннее хранилище, связанное с `mntlabel` и `fslabel`, чтобы их можно было уничтожить.

#### `mpo_destroy_socket_label`

```
void mpo_destroy_socket_label(struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `label` | Уничтожение метки сокета | |

Уничтожить метку на сокете. В этой точке входа модуль политики должен освободить любое внутреннее хранилище, связанное с `label`, чтобы её можно было уничтожить.

#### `mpo_destroy_socket_peer_label`

```
void mpo_destroy_socket_peer_label(struct label *peerlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `peerlabel` | Сокет: метка однорангового узла уничтожается | |

Уничтожить метку однорангового узла на сокете. В этой точке входа модуль политики должен освободить любое внутреннее хранилище, связанное с `label`, чтобы её можно было уничтожить.

#### `mpo_destroy_pipe_label`

```
void mpo_destroy_pipe_label(struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `label` | Метка канала (pipe) | |

Уничтожить метку на канале. В этой точке входа модуль политики должен освободить всю внутреннюю память, связанную с `label`, чтобы её можно было уничтожить.

#### `mpo_destroy_proc_label`

```
void mpo_destroy_proc_label(struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `label` | Метка процесса | |

Уничтожить метку на процессе. В этой точке входа модуль политики должен освободить любое внутреннее хранилище, связанное с `label`, чтобы её можно было уничтожить.

#### `mpo_destroy_vnode_label`

```
void mpo_destroy_vnode_label(struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `label` | Метка процесса | |

Уничтожить метку на vnode. В этой точке входа модуль политики должен освободить любое внутреннее хранилище, связанное с `label`, чтобы её можно было уничтожить.

#### `mpo_copy_mbuf_label`

```
void mpo_copy_mbuf_label(struct label *src, struct label *dest);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `src` | Метка источника | |
| `dest` | Метка назначения | |

Скопировать информацию метки из `src` в `dest`.

#### `mpo_copy_pipe_label`

```
void mpo_copy_pipe_label(struct label *src, struct label *dest);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `src` | Метка источника | |
| `dest` | Метка назначения | |

Скопировать информацию метки из `src` в `dest`.

#### `mpo_copy_vnode_label`

```
void mpo_copy_vnode_label(struct label *src, struct label *dest);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `src` | Метка источника | |
| `dest` | Метка назначения | |

Скопировать информацию метки из `src` в `dest`.

### mpo_externalize_cred_label

```
int mpo_externalize_cred_label(struct label *label, char *element_name,
    struct sbuf *sb, int *claimed);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| label | Метка для вынесения во внешний ресурс | |
| element_name | Имя политики, метка которой должна быть вынесена во внешний ресурс | |
| sb | Буфер строки для заполнения текстовым представлением метки | |
| claimed | Должно быть увеличено, когда element_data может быть заполнено. | |

Создать внешнее представление метки на основе переданной структуры метки. Внешнее представление метки состоит из текстового представления содержимого метки, которое может использоваться пользовательскими приложениями и прочитано пользователем. В настоящее время будут вызываться точки входа externalize всех политик, поэтому реализация должна проверить содержимое element_name перед попыткой заполнить sb. Если element_name не соответствует имени вашей политики, просто верните 0. Возвращайте ненулевое значение только в случае ошибки при внешнем представлении данных метки. После того как политика заполнит element_data, *claimed должен быть увеличен.

### mpo_externalize_ifnet_label

```
int mpo_externalize_ifnet_label(struct label *label, char *element_name,
    struct sbuf *sb, int *claimed);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| label | Метка для вынесения во внешний ресурс | |
| element_name | Имя политики, метка которой должна быть вынесена во внешний ресурс | |
| sb | Буфер строки для заполнения текстовым представлением метки | |

| Параметр | Описание | Блокировка |
|---|---|---|
| claimed | Должно быть увеличено, когда element_data может быть заполнено. | |

Создать внешнее представление метки на основе переданной структуры метки. Внешнее представление метки состоит из текстового представления содержимого метки, которое может использоваться пользовательскими приложениями и прочитано пользователем. В настоящее время будут вызываться точки входа externalize всех политик, поэтому реализация должна проверить содержимое element_name перед попыткой заполнить sb. Если element_name не соответствует имени вашей политики, просто верните 0. Возвращайте ненулевое значение только в случае ошибки при внешнем представлении данных метки. После того как политика заполнит element_data, *claimed должен быть увеличен.

### mpo_externalize_pipe_label

```
int mpo_externalize_pipe_label(struct label *label, char *element_name,
    struct sbuf *sb, int *claimed);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| label | Метка для вынесения во внешний ресурс | |
| element_name | Имя политики, метка которой должна быть вынесена во внешний ресурс | |
| sb | Буфер строки для заполнения текстовым представлением метки | |
| claimed | Должно быть увеличено, когда element_data может быть заполнено. | |

Создать внешнее представление метки на основе переданной структуры метки. Внешнее представление метки состоит из текстового представления содержимого метки, которое может использоваться пользовательскими приложениями и прочитано пользователем. В настоящее время будут вызываться точки входа externalize всех политик, поэтому реализация должна проверить содержимое element_name перед попыткой заполнить sb. Если element_name не соответствует имени вашей политики, просто верните 0. Возвращайте ненулевое значение только в случае ошибки при внешнем представлении данных метки. После того как политика заполнит element_data, *claimed должен быть увеличен.

### mpo_externalize_socket_label

```
int mpo_externalize_socket_label(struct label *label, char *element_name,
```

```
    struct sbuf *sb, int *claimed);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| label | Метка для вынесения во внешний ресурс | |
| element_name | Имя политики, метка которой должна быть вынесена во внешний ресурс | |
| sb | Буфер строки для заполнения текстовым представлением метки | |
| claimed | Должно быть увеличено, когда element_data может быть заполнено. | |

Создать внешнее представление метки на основе переданной структуры метки. Внешнее представление метки состоит из текстового представления содержимого метки, которое может использоваться пользовательскими приложениями и прочитано пользователем. В настоящее время будут вызываться точки входа externalize всех политик, поэтому реализация должна проверить содержимое element_name перед попыткой заполнить sb. Если element_name не соответствует имени вашей политики, просто верните 0. Возвращайте ненулевое значение только в случае ошибки при внешнем представлении данных метки. После того как политика заполнит element_data, *claimed должен быть увеличен.

### mpo_externalize_socket_peer_label

```
int mpo_externalize_socket_peer_label(struct label *label, char *element_name,
    struct sbuf *sb, int *claimed);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| label | Метка для вынесения во внешний ресурс | |
| element_name | Имя политики, метка которой должна быть вынесена во внешний ресурс | |
| sb | Буфер строки для заполнения текстовым представлением метки | |
| claimed | Должно быть увеличено, когда element_data может быть заполнено. | |

Создать внешнее представление метки на основе переданной структуры метки. Внешнее

представление метки состоит из текстового представления содержимого метки, которое может использоваться пользовательскими приложениями и прочитано пользователем. В настоящее время будут вызываться точки входа `externalize` всех политик, поэтому реализация должна проверить содержимое `element_name` перед попыткой заполнить `sb`. Если `element_name` не соответствует имени вашей политики, просто верните 0. Возвращайте ненулевое значение только в случае ошибки при внешнем представлении данных метки. После того как политика заполнит `element_data`, `*claimed` должен быть увеличен.

### `mpo_externalize_vnode_label`

```
int mpo_externalize_vnode_label(struct label *label, char *element_name,
    struct sbuf *sb, int *claimed);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `label` | Метка для вынесения во внешний ресурс | |
| `element_name` | Имя политики, метка которой должна быть вынесена во внешний ресурс | |
| `sb` | Буфер строки для заполнения текстовым представлением метки | |
| `claimed` | Должно быть увеличено, когда `element_data` может быть заполнено. | |

Создать внешнее представление метки на основе переданной структуры метки. Внешнее представление метки состоит из текстового представления содержимого метки, которое может использоваться пользовательскими приложениями и прочитано пользователем. В настоящее время будут вызываться точки входа `externalize` всех политик, поэтому реализация должна проверить содержимое `element_name` перед попыткой заполнить `sb`. Если `element_name` не соответствует имени вашей политики, просто верните 0. Возвращайте ненулевое значение только в случае ошибки при внешнем представлении данных метки. После того как политика заполнит `element_data`, `*claimed` должен быть увеличен.

### `mpo_internalize_cred_label`

```
int mpo_internalize_cred_label(struct label *label, char *element_name,
    char *element_data, int *claimed);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `label` | Метка для заполнения | |

| Параметр | Описание | Блокировка |
|---|---|---|
| element_name | Имя политики, метка которой должна быть приведена к внутреннему представлению | |
| element_data | Текстовые данные для преобразования к внутреннему представлению | |
| claimed | Должно увеличиваться, когда данные могут быть успешно преобразовываться во внутреннее представление. | |

Создать внутреннюю структуру меток на основе данных метки во внешнем представлении в текстовом формате. В настоящее время, при запросе преобразования во внутреннее представление вызываются точки входа internalize всех политик, поэтому реализация должна сравнивать содержимое element_name со своим именем, чтобы убедиться, что она должна преобразовывать данные в element_data. Как и в точках входа externalize, точка входа должна возвращать 0, если element_name не совпадает с её собственным именем, или когда данные могут быть успешно преобразованы, в этом случае *claimed должен быть увеличен.

### mpo_internalize_ifnet_label

```
int mpo_internalize_ifnet_label(struct label *label, char *element_name,
    char *element_data, int *claimed);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| label | Метка для заполнения | |
| element_name | Имя политики, метка которой должна быть приведена к внутреннему представлению | |
| element_data | Текстовые данные для преобразования к внутреннему представлению | |
| claimed | Должно увеличиваться, когда данные могут быть успешно преобразовываться во внутреннее представление. | |

Создать внутреннюю структуру меток на основе данных метки во внешнем представлении в текстовом формате. В настоящее время, при запросе преобразования во внутреннее представление вызываются точки входа internalize всех политик, поэтому реализация

должна сравнивать содержимое element_name со своим именем, чтобы убедиться, что она должна преобразовывать данные в element_data. Как и в точках входа externalize, точка входа должна возвращать 0, если element_name не совпадает с её собственным именем, или когда данные могут быть успешно преобразованы, в этом случае *claimed должен быть увеличен.

### mpo_internalize_pipe_label

```
int mpo_internalize_pipe_label(struct label *label, char *element_name,
    char *element_data, int *claimed);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| label | Метка для заполнения | |
| element_name | Имя политики, метка которой должна быть приведена к внутреннему представлению | |
| element_data | Текстовые данные для преобразования к внутреннему представлению | |
| claimed | Должно увеличиваться, когда данные могут быть успешно преобразовываться во внутреннее представление. | |

Создать внутреннюю структуру меток на основе данных метки во внешнем представлении в текстовом формате. В настоящее время, при запросе преобразования во внутреннее представление вызываются точки входа internalize всех политик, поэтому реализация должна сравнивать содержимое element_name со своим именем, чтобы убедиться, что она должна преобразовывать данные в element_data. Как и в точках входа externalize, точка входа должна возвращать 0, если element_name не совпадает с её собственным именем, или когда данные могут быть успешно преобразованы, в этом случае *claimed должен быть увеличен.

### mpo_internalize_socket_label

```
int mpo_internalize_socket_label(struct label *label, char *element_name,
    char *element_data, int *claimed);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| label | Метка для заполнения | |

| Параметр | Описание | Блокировка |
|---|---|---|
| element_name | Имя политики, метка которой должна быть приведена к внутреннему представлению | |
| element_data | Текстовые данные для преобразования к внутреннему представлению | |
| claimed | Должно увеличиваться, когда данные могут быть успешно преобразовываться во внутреннее представление. | |

Создать внутреннюю структуру меток на основе данных метки во внешнем представлении в текстовом формате. В настоящее время, при запросе преобразования во внутреннее представление вызываются точки входа internalize всех политик, поэтому реализация должна сравнивать содержимое element_name со своим именем, чтобы убедиться, что она должна преобразовывать данные в element_data. Как и в точках входа externalize, точка входа должна возвращать 0, если element_name не совпадает с её собственным именем, или когда данные могут быть успешно преобразованы, в этом случае *claimed должен быть увеличен.

### mpo_internalize_vnode_label

```
int mpo_internalize_vnode_label(struct label *label, char *element_name,
    char *element_data, int *claimed);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| label | Метка для заполнения | |
| element_name | Имя политики, метка которой должна быть приведена к внутреннему представлению | |
| element_data | Текстовые данные для преобразования к внутреннему представлению | |
| claimed | Должно увеличиваться, когда данные могут быть успешно преобразовываться во внутреннее представление. | |

Создать внутреннюю структуру меток на основе данных метки во внешнем представлении в текстовом формате. В настоящее время, при запросе преобразования во внутреннее представление вызываются точки входа internalize всех политик, поэтому реализация

должна сравнивать содержимое `element_name` со своим именем, чтобы убедиться, что она должна преобразовывать данные в `element_data`. Как и в точках входа `externalize`, точка входа должна возвращать 0, если `element_name` не совпадает с её собственным именем, или когда данные могут быть успешно преобразованы, в этом случае `*claimed` должен быть увеличен.

### 6.7.3. События метки

Этот класс точек входа используется фреймворком MAC для разрешения политикам поддерживать информацию о метках на объектах ядра. Для каждого помеченного объекта ядра, представляющего интерес для политики MAC, могут быть зарегистрированы точки входа для соответствующих событий жизненного цикла. Все объекты реализуют хуки инициализации, создания и уничтожения. Некоторые объекты также реализуют перемаркировку, позволяя пользовательским процессам изменять метки на объектах. Некоторые объекты также реализуют специфичные для объекта события, такие как события меток, связанные с повторной сборкой IP. Типичный помеченный объект будет иметь следующий жизненный цикл точек входа:

```
Label initialization          o
(object-specific wait)         \
Label creation                  o
                                 \
Relabel events,                   o--<--.
Various object-specific,          |     |
Access control events             ~-->--o
                                        \
Label destruction                        o
```

Инициализация меток позволяет политикам выделять память и устанавливать начальные значения для меток без контекста использования объекта. Слот метки, выделенный для политики, по умолчанию будет обнулен, поэтому некоторым политикам может не потребоваться выполнять инициализацию.

Создание метки происходит, когда структура ядра связывается с реальным объектом ядра. Например, Mbuf могут быть выделены и оставаться неиспользованными в пуле до тех пор, пока они не понадобятся. Выделение mbuf приводит к инициализации метки на mbuf, но создание mbuf происходит, когда mbuf связывается с датаграммой. Обычно для события создания предоставляется контекст, включая обстоятельства создания и метки других значимых объектов в процессе создания. Например, когда mbuf создаётся из сокета, сокет и его метка будут переданы зарегистрированным политикам в дополнение к новому mbuf и его метке. Выделение памяти в событиях создания не рекомендуется, так как это может происходить в чувствительных к производительности участках ядра; кроме того, вызовы создания не могут завершиться неудачей, поэтому невозможность выделить память не может быть сообщена.

События, привязанные к объектам, обычно не попадают в другие классы событий меток, но, как правило, предоставляют возможность изменить или обновить метку объекта на основе дополнительного контекста. Например, метка в очереди сборки IP-фрагментов может быть

обновлена во время точки входа `MAC_UPDATE_IPQ` в результате принятия дополнительного mbuf в эту очередь.

События контроля доступа подробно рассматриваются в следующем разделе.

Уничтожение метки позволяет политикам освобождать хранилище или состояние, связанное с меткой во время её ассоциации с объектом, чтобы структуры данных ядра, поддерживающие объект, могли быть повторно использованы или освобождены.

В дополнение к меткам, связанным с определёнными объектами ядра, существует дополнительный класс меток: временные метки. Эти метки используются для хранения информации об обновлениях, отправляемых пользовательскими процессами. Они инициализируются и уничтожаются так же, как и другие типы меток, но событие создания — это `MAC_INTERNALIZE`, которое принимает пользовательскую метку для преобразования во внутреннее представление в ядре.

### Действия с событиями меток объектов файловой системы

`mpo_associate_vnode_devfs`

```
void mpo_associate_vnode_devfs(struct mount *mp, struct label *fslabel,
    struct devfs_dirent *de, struct label *delabel, struct vnode *vp,
    struct label *vlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `mp` | Точка монтирования devfs | |
| `fslabel` | Метка файловой системы devfs (`mp→mnt_fslabel`) | |
| `de` | Запись каталога devfs | |
| `delabel` | Метка политики, связанная с `de` | |
| `vp` | узел vnode, связанный с `de` | |
| `vlabel` | Метка политики, связанная с `vp` | |

Заполнить метку (`vlabel`) для только что созданного devfs vnode на основе записи каталога devfs, переданной в `de`, и её метки.

`mpo_associate_vnode_extattr`

```
int mpo_associate_vnode_extattr(struct mount *mp, struct label *fslabel,
    struct vnode *vp, struct label *vlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| mp | Точка монтирования файловой системы | |
| fslabel | Метка файловой системы | |
| vp | Узел vnode для метки | |
| vlabel | Метка политики, связанная с vp | |

Попытка получить метку для vp из расширенных атрибутов файловой системы. В случае успеха возвращается значение 0. Если получение расширенных атрибутов не поддерживается, допустимым резервным вариантом является копирование fslabel в vlabel. В случае ошибки должно быть возвращено соответствующее значение errno.

#### mpo_associate_vnode_singlelabel

```
void mpo_associate_vnode_singlelabel(struct mount *mp, struct label *fslabel,
    struct vnode *vp, struct label *vlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| mp | Точка монтирования файловой системы | |
| fslabel | Метка файловой системы | |
| vp | Узел vnode для метки | |
| vlabel | Метка политики, связанная с vp | |

На файловых системах без поддержки multilabel эта точка входа вызывается для установки метки политики для vp на основе метки файловой системы fslabel.

#### mpo_create_devfs_device

```
void mpo_create_devfs_device(dev_t dev, struct devfs_dirent *devfs_dirent,
    struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| dev | Устройство, соответствующее devfs_dirent | |
| devfs_dirent | Запись в каталоге devfs, для которой создаётся метка. | |
| label | Метка для devfs_dirent, которую нужно заполнить. | |

Заполнить метку на devfs_dirent, создаваемом для переданного устройства. Этот вызов будет

выполнен при монтировании файловой системы устройств, её восстановлении или при появлении нового устройства.

#### mpo_create_devfs_directory

```
void mpo_create_devfs_directory(char *dirname, int dirnamelen,
    struct devfs_dirent *devfs_dirent, struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| dirname | Имя создаваемого каталога | |
| namelen | Длина строки dirname | |
| devfs_dirent | Запись в devfs для создаваемого каталога. | |

Заполнить метку на devfs_dirent, создаваемом для переданного каталога. Этот вызов будет выполнен при монтировании файловой системы устройств, её восстановлении или при появлении нового устройства, требующего определённой иерархии каталогов.

#### mpo_create_devfs_symlink

```
void mpo_create_devfs_symlink(struct ucred *cred, struct mount *mp,
    struct devfs_dirent *dd, struct label *ddlabel, struct devfs_dirent *de,
    struct label *delabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| mp | Точка монтирования devfs | |
| dd | Место назначения ссылки | |
| ddlabel | Метка, связанная с dd | |
| de | Точка входа символьной ссылки | |
| delabel | Метка, связанная с de | |

Заполнить метку (delabel) для новой структуры devfs(5) символьной ссылки.

#### mpo_create_vnode_extattr

```
int mpo_create_vnode_extattr(struct ucred *cred, struct mount *mp,
    struct label *fslabel, struct vnode *dvp, struct label *dlabel,
    struct vnode *vp, struct label *vlabel, struct componentname *cnp);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `mount` | Точка монтирования файловой системы | |
| `label` | Метка файловой системы | |
| `dvp` | Родительский каталог vnode | |
| `dlabel` | Метка, связанная с `dvp` | |
| `vp` | Вновь созданная vnode | |
| `vlabel` | Метка политики, связанная с `vp` | |
| `cnp` | Название компонента для `vp` | |

Записать метку для `vp` в соответствующий расширенный атрибут. Если запись прошла успешно, заполняет `vlabel` меткой и возвращает 0. В противном случае вернет соответствующую ошибку.

`mpo_create_mount`

```
void mpo_create_mount(struct ucred *cred, struct mount *mp, struct label *mnt,
    struct label *fslabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `mp` | Объект; файловая система, которая монтируется | |
| `mntlabel` | Метка политики для заполнения в `mp` | |
| `fslabel` | Метка политики для файловой системы, монтируемой в `mp`. | |

Заполнить метки на точке монтирования, создаваемой переданными учётными данными субъекта. Этот вызов будет выполнен при монтировании новой файловой системы.

`mpo_create_root_mount`

```
void mpo_create_root_mount(struct ucred *cred, struct mount *mp,
    struct label *mntlabel, struct label *fslabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|

См. `mpo_create_mount`.

Заполнить метки на точке монтирования, создаваемой переданными учётными данными субъекта. Этот вызов будет выполнен при монтировании корневой файловой системы после `mpo_create_mount;`.

#### `mpo_relabel_vnode`

```
void mpo_relabel_vnode(struct ucred *cred, struct vnode *vp,
    struct label *vnodelabel, struct label *newlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `vp` | vnode для перемаркировки | |
| `vnodelabel` | Существующая метка политики для `vp` | |
| `newlabel` | Новая, возможно частичная метка для замены `vnodelabel` | |

Обновить метку на переданном vnode с учётом переданной обновленной метки vnode и переданных учётных данных субъекта.

#### `mpo_setlabel_vnode_extattr`

```
int mpo_setlabel_vnode_extattr(struct ucred *cred, struct vnode *vp,
    struct label *vlabel, struct label *intlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `vp` | vnode, для которой записывается метка | |
| `vlabel` | Метка политики, связанная с `vp` | |
| `intlabel` | Метка для записи | |

Записать политику из `intlabel` в расширенный атрибут. Этот метод вызывается из `vop_stdcreatevnode_ea`.

#### `mpo_update_devfsdirent`

```
void mpo_update_devfsdirent(struct devfs_dirent *devfs_dirent,
    struct label *direntlabel, struct vnode *vp, struct label *vnodelabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| devfs_dirent | Объект; каталожная запись devfs | |
| direntlabel | Метка политики для devfs_dirent, которая будет обновлена. | |
| vp | Родительский vnode | Заблокирован |
| vnodelabel | Метка политики для vp | |

Обновить метку devfs_dirent из переданной метки devfs vnode. Этот вызов будет выполнен, когда devfs vnode успешно перемаркирован, чтобы зафиксировать изменение метки, чтобы оно сохранилось, даже если vnode будет переиспользован. Он также будет выполнен при создании символьной ссылки в devfs после вызова mac_vnode_create_from_vnode для инициализации метки vnode.

### Действия с событиями меток объектов IPC

#### mpo_create_mbuf_from_socket

```
void mpo_create_mbuf_from_socket(struct socket *so, struct label *socketlabel,
    struct mbuf *m, struct label *mbuflabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| socket | Сокет | Блокировка сокетов — работа в процессе |
| socketlabel | Метка политики для socket | |
| m | Объект; mbuf | |
| mbuflabel | Метка политики для заполнения для m | |

Установить метку на только что созданном заголовке mbuf из переданной метки сокета. Этот вызов выполняется, когда новый датаграмма или сообщение генерируется сокетом и сохраняется в переданном mbuf.

#### mpo_create_pipe

```
void mpo_create_pipe(struct ucred *cred, struct pipe *pipe,
    struct label *pipelabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| pipe | Канал | |

| Параметр | Описание | Блокировка |
|---|---|---|
| `pipelabel` | Метка политики, связанная с `pipe` | |

Установить метку на только что созданном канале из переданных учётных данных субъекта. Этот вызов выполняется при создании нового канала.

`mpo_create_socket`

```
void mpo_create_socket(struct ucred *cred, struct socket *so,
    struct label *socketlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | Неизменяемый |
| `so` | Объект; сокет для добавления метки | |
| `socketlabel` | Метка для заполнения для `so` | |

Установить метку на новом сокете из переданных учётных данных субъекта. Этот вызов выполняется при создании сокета.

`mpo_create_socket_from_socket`

```
void mpo_create_socket_from_socket(struct socket *oldsocket,
    struct label *oldsocketlabel, struct socket *newsocket,
    struct label *newsocketlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `oldsocket` | Сокет, вызвавший listen | |
| `oldsocketlabel` | Метка политики, связанная с `oldsocket` | |
| `newsocket` | Новый сокет | |
| `newsocketlabel` | Метка политики, связанная с `newsocketlabel` | |

Создать метку сокета `newsocket`, только что принявшему соединение через accept(2), на основе сокета `oldsocket`, вызвавшего listen(2) .

`mpo_relabel_pipe`

```
void mpo_relabel_pipe(struct ucred *cred, struct pipe *pipe,
    struct label *oldlabel, struct label *newlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `pipe` | Канал | |
| `oldlabel` | Текущая метка политики, связанная с `pipe` | |
| `newlabel` | Обновление метки политики для применения к `pipe` | |

Применить новую метку `newlabel` к `pipe`.

#### `mpo_relabel_socket`

```
void mpo_relabel_socket(struct ucred *cred, struct socket *so,
    struct label *oldlabel, struct label *newlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | Неизменяемый |
| `so` | Объект; сокет | |
| `oldlabel` | Текущая метка для `so` | |
| `newlabel` | Метка обновления для `so` | |

Обновить метку на сокете из переданного обновления метки сокета.

#### `mpo_set_socket_peer_from_mbuf`

```
void mpo_set_socket_peer_from_mbuf(struct mbuf *mbuf, struct label *mbuflabel,
    struct label *oldlabel, struct label *newlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `mbuf` | Первый датаграмм, полученный через сокет | |
| `mbuflabel` | Метка для `mbuf` | |
| `oldlabel` | Текущая метка для сокета | |
| `newlabel` | Метка политики для заполнения сокета | |

Установить метку однорангового узла на потоковом сокете из переданной метки mbuf. Этот вызов будет выполнен при получении первого датаграммы потоковым сокетом, за исключением сокетов домена Unix.

`mpo_set_socket_peer_from_socket`

```
void mpo_set_socket_peer_from_socket(struct socket *oldsocket,
    struct label *oldsocketlabel, struct socket *newsocket,
    struct label *newsocketpeerlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `oldsocket` | Локальный сокет | |
| `oldsocketlabel` | Метка политики для `oldsocket` | |
| `newsocket` | Сокет однорангового узла (peer socket) | |
| `newsocketpeerlabel` | Метка политики для заполнения для `newsocket` | |

Установите метку однорангового узла на потоковом UNIX-сокете из переданной конечной точки удалённого сокета. Этот вызов будет выполнен при соединении пары сокетов и будет произведен для обеих конечных точек.

### Действия с событиями меток сетевых объектов

`mpo_create_bpfdesc`

```
void mpo_create_bpfdesc(struct ucred *cred, struct bpf_d *bpf_d,
    struct label *bpflabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | Неизменяемый |
| `bpf_d` | Объект; дескриптор bpf | |
| `bpf` | Метка политики для заполнения для `bpf_d` | |

Установить метку на новом дескрипторе BPF из переданных учётных данных субъекта. Этот вызов будет выполнен при открытии узла устройства BPF процессом с переданными учётными данными субъекта.

`mpo_create_ifnet`

```
void mpo_create_ifnet(struct ifnet *ifnet, struct label *ifnetlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `ifnet` | Сетевой интерфейс | |
| `ifnetlabel` | Метка политики для заполнения для `ifnet` | |

Установить метку на вновь созданном интерфейсе. Этот вызов может быть выполнен, когда новое физическое устройство становится доступным системе, или когда псевдо-интерфейс создаётся во время загрузки или в результате действия пользователя.

`mpo_create_ipq`

```
void mpo_create_ipq(struct mbuf *fragment, struct label *fragmentlabel,
    struct ipq *ipq, struct label *ipqlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `fragment` | Первый полученный IP-фрагмент | |
| `fragmentlabel` | Метка политики для `fragment` | |
| `ipq` | Очередь повторной сборки IP, которой добавляется метка | |
| `ipqlabel` | Метка политики для заполнения в `ipq` | |

Установить метку на вновь созданной очереди сборки IP-фрагментов из заголовка mbuf первого полученного фрагмента.

`mpo_create_datagram_from_ipq`

```
void mpo_create_create_datagram_from_ipq(struct ipq *ipq,
    struct label *ipqlabel, struct mbuf *datagram, struct label *datagramlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `ipq` | Очередь повторной сборки IP | |
| `ipqlabel` | Метка политики для `ipq` | |
| `datagram` | Датаграмма для добавления метки | |
| `datagramlabel` | Метка политики для заполнения в `datagramlabel` | |

Установите метку на вновь собранный IP-датаграмму из очереди сборки IP-фрагментов, из которой он был сгенерирован.

`mpo_create_fragment`

```
void mpo_create_fragment(struct mbuf *datagram, struct label *datagramlabel,
    struct mbuf *fragment, struct label *fragmentlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `datagram` | Датаграмма | |
| `datagramlabel` | Метка политики для `datagram` | |
| `fragment` | Фрагмент, которому будет установлена метка | |
| `fragmentlabel` | Метка политики для заполнения для `datagram` | |

Установить метку на заголовке mbuf вновь созданного IP-фрагмента из метки на заголовке mbuf датаграммы, из которой он был сгенерирован.

`mpo_create_mbuf_from_mbuf`

```
void mpo_create_mbuf_from_mbuf(struct mbuf *oldmbuf, struct label *oldmbuflabel,
    struct mbuf *newmbuf, struct label *newmbuflabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `oldmbuf` | Существующий (исходный) mbuf | |
| `oldmbuflabel` | Метка политики для `oldmbuf` | |
| `newmbuf` | Новый mbuf для добавления метки | |
| `newmbuflabel` | Метка политики для заполнения в `newmbuf` | |

Установить метку в заголовке mbuf для вновь созданной датаграммы на основе заголовка mbuf существующей датаграммы. Этот вызов может быть выполнен в ряде ситуаций, включая случаи, когда для mbuf заново выделяется память для целей выравнивания.

`mpo_create_mbuf_linklayer`

```
void mpo_create_mbuf_linklayer(struct ifnet *ifnet, struct label *ifnetlabel,
    struct mbuf *mbuf, struct label *mbuflabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `ifnet` | Сетевой интерфейс | |
| `ifnetlabel` | Метка политики для `ifnet` | |
| `mbuf` | заголовок mbuf для новой датаграммы | |
| `mbuflabel` | Метка политики для заполнения для `mbuf` | |

Установить метку в заголовке mbuf для вновь созданной датаграммы, сгенерированного для целей ответа на канальном уровне для переданного интерфейса. Этот вызов может быть выполнен в ряде ситуаций, включая ответы ARP или ND6 в стеках IPv4 и IPv6.

`mpo_create_mbuf_from_bpfdesc`

```
void mpo_create_mbuf_from_bpfdesc(struct bpf_d *bpf_d, struct label *bpflabel,
    struct mbuf *mbuf, struct label *mbuflabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `bpf_d` | Дескриптор BPF | |
| `bpflabel` | Метка политики для `bpflabel` | |
| `mbuf` | Новый mbuf для добавления метки | |
| `mbuflabel` | Метка политики для заполнения `mbuf` | |

Установить метку на заголовок mbuf вновь созданной датаграммы, сгенерированной с использованием переданного дескриптора BPF. Этот вызов выполняется при записи в устройство BPF, связанное с переданным дескриптором BPF.

`mpo_create_mbuf_from_ifnet`

```
void mpo_create_mbuf_from_ifnet(struct ifnet *ifnet, struct label *ifnetlabel,
    struct mbuf *mbuf, struct label *mbuflabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `ifnet` | Сетевой интерфейс | |
| `ifnetlabel` | Метка политики для `ifnetlabel` | |
| `mbuf` | заголовок mbuf для новой датаграммы | |
| `mbuflabel` | Метка политики для заполнения для `mbuf` | |

Установить метку на заголовке mbuf вновь созданной датаграммы, сгенерированной из переданного сетевого интерфейса.

`mpo_create_mbuf_multicast_encap`

```
void mpo_create_mbuf_multicast_encap(struct mbuf *oldmbuf,
    struct label *oldmbuflabel, struct ifnet *ifnet, struct label *ifnetlabel,
    struct mbuf *newmbuf, struct label *newmbuflabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `oldmbuf` | Заголовок mbuf для существующего датаграммы | |
| `oldmbuflabel` | Метка политики для `oldmbuf` | |
| `ifnet` | Сетевой интерфейс | |
| `ifnetlabel` | Метка политики для `ifnet` | |
| `newmbuf` | Заголовок mbuf для пометки новой датаграммы | |
| `newmbuflabel` | Метка политики для заполнения в `newmbuf` | |

Установить метку в заголовке mbuf для вновь созданной датаграммы, сгенерированной из существующей переданной датаграммы, при её обработке переданным интерфейсом мультикастовой инкапсуляции. Этот вызов происходит при доставке mbuf с использованием виртуального интерфейса.

#### `mpo_create_mbuf_netlayer`

```
void mpo_create_mbuf_netlayer(struct mbuf *oldmbuf, struct label *oldmbuflabel,
    struct mbuf *newmbuf, struct label *newmbuflabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `oldmbuf` | Полученная датаграмма | |
| `oldmbuflabel` | Метка политики для `oldmbuf` | |
| `newmbuf` | Вновь созданная датаграмма | |
| `newmbuflabel` | Метка политики для `newmbuf` | |

Установить метку на заголовок mbuf вновь созданной датаграммы, сгенерированной стеком IP в ответ на полученную датаграмму (`oldmbuf`). Этот вызов может быть выполнен в различных ситуациях, включая ответ на датаграммы ICMP-запросов.

#### `mpo_fragment_match`

```
int mpo_fragment_match(struct mbuf *fragment, struct label *fragmentlabel,
    struct ipq *ipq, struct label *ipqlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `fragment` | Фрагмент IP-датаграммы | |
| `fragmentlabel` | Метка политики для `fragment` | |
| `ipq` | Очередь сборки IP-фрагментов | |

| Параметр | Описание | Блокировка |
|---|---|---|
| ipqlabel | Метка политики для ipq | |

Определить, соответствует ли заголовок mbuf, содержащий фрагмент IP-датаграммы (fragment), метке переданной очереди сборки IP-фрагментов (ipq). Возвращает (1) при успешном совпадении или (0) при отсутствии совпадения. Этот вызов выполняется, когда IP-стек пытается найти существующую очередь сборки фрагментов для вновь полученного фрагмента; если поиск не удаётся, для фрагмента может быть создана новая очередь сборки. Политики могут использовать эту точку входа, чтобы предотвратить сборку в остальном подходящих IP-фрагментов, если политика не разрешает их сборку на основе метки или другой информации.

#### mpo_relabel_ifnet

```
void mpo_relabel_ifnet(struct ucred *cred, struct ifnet *ifnet,
    struct label *ifnetlabel, struct label *newlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| ifnet | Объект; Сетевой интерфейс | |
| ifnetlabel | Метка политики для ifnet | |
| newlabel | Метка обновления для применения к ifnet | |

Обновить метку сетевого интерфейса, ifnet, на основе переданной новой метки, newlabel, и переданных учётных данных субъекта, cred.

#### mpo_update_ipq

```
void mpo_update_ipq(struct mbuf *fragment, struct label *fragmentlabel,
    struct ipq *ipq, struct label *ipqlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| mbuf | IP фрагмент | |
| mbuflabel | Метка политики для mbuf | |
| ipq | Очередь сборки IP-фрагментов | |
| ipqlabel | Метка политики для обновления для ipq | |

Обновить метку в очереди сборки IP-фрагментов (ipq) на основе принятия переданного заголовка IP-фрагмента mbuf (mbuf).

### Действия с событиями меток процессов

`mpo_create_cred`

```
void mpo_create_cred(struct ucred *parent_cred, struct ucred *child_cred);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `parent_cred` | Учётные данные субъекта -родителя | |
| `child_cred` | Учётные данные дочернего субъекта | |

Установить метку вновь созданного субъекта из переданного субъекта. Этот вызов будет выполнен при вызове crcopy(9) для только что созданной структуры `struct ucred`. Этот вызов не следует путать с событием создания или ветвления процесса.

`mpo_execve_transition`

```
void mpo_execve_transition(struct ucred *old, struct ucred *new,
    struct vnode *vp, struct label *vnodelabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `old` | Учётные данные существующего субъекта | Неизменяемый |
| `new` | Учётные данные нового субъекта для добавления метки | |
| `vp` | Файл для выполнения | Заблокирован |
| `vnodelabel` | Метка политики для `vp` | |

Обновить метку учётных данных вновь созданного субъекта (`new`) на основе переданных учётных данных существующего субъекта (`old`) в соответствии с переходом метки, вызванным выполнением переданного vnode (`vp`). Этот вызов происходит, когда процесс выполняет переданный vnode, и одна из политик возвращает успех из точки входа `mpo_execve_will_transition`. Политики могут выбрать реализацию этого вызова просто путем вызова `mpo_create_cred` и передачи двух субъектов учётных данных, чтобы не реализовывать событие перехода. Политики не должны оставлять эту точку входа нереализованной, если они реализуют `mpo_create_cred`, даже если они не реализуют `mpo_execve_will_transition`.

`mpo_execve_will_transition`

```
int mpo_execve_will_transition(struct ucred *old, struct vnode *vp,
    struct label *vnodelabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `old` | Учётные данные субъекта перед execve(2) | Неизменяемый |
| `vp` | Файл для выполнения | |
| `vnodelabel` | Метка политики для `vp` | |

Определить, будет ли политика выполнять событие перехода в результате выполнения переданного vnode с использованием переданных учётных данных субъекта. Вернуть 1, если переход требуется, и 0, если нет. Даже если политика возвращает 0, она должна корректно обрабатывать неожиданный вызов `mpo_execve_transition`, так как этот вызов может произойти из-за запроса перехода другой политикой.

#### `mpo_create_proc0`

```
void mpo_create_proc0(struct ucred *cred);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта для заполнения | |

Создать учётные данные субъекта процесса 0, родителя всех процессов ядра.

#### `mpo_create_proc1`

```
void mpo_create_proc1(struct ucred *cred);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта для заполнения | |

Создать учётные данные субъекта процесса 1, родителя всех пользовательских процессов.

#### `mpo_relabel_cred`

```
void mpo_relabel_cred(struct ucred *cred, struct label *newlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `newlabel` | Обновление метки для применения к `cred` | |

Обновить метку на учётных данных субъекта из переданной обновляемой метки.

### 6.7.4. Проверки контроля доступа

Точки входа контроля доступа позволяют модулям политики влиять на решения по контролю доступа, принимаемые ядром. Обычно, хотя и не всегда, аргументы точки входа контроля доступа включают одно или несколько удостоверяющих полномочий, информацию (возможно, включая метку) для любых других объектов, участвующих в операции. Точка входа контроля доступа может вернуть 0 для разрешения операции или значение ошибки errno(2). Результаты вызова точки входа через различные зарегистрированные модули политики будут объединены следующим образом: если все модули разрешают успешное выполнение операции, будет возвращен успех. Если один или несколько модулей возвращают ошибку, будет возвращена ошибка. Если более одного модуля возвращают ошибку, значение errno, которое будет возвращено пользователю, выбирается с использованием следующего приоритета, реализованного функцией `error_select()` в kern_mac.c:

| | |
|---|---|
| Наивысший приоритет | EDEADLK |
| | EINVAL |
| | ESRCH |
| | EACCES |
| Наименьший приоритет | EPERM |

Если ни одно из значений ошибок, возвращаемых всеми модулями, не указано в таблице приоритетов, будет возвращено произвольно выбранное значение из набора. В общем случае правила устанавливают следующий порядок приоритетов ошибок: сбои ядра, неверные аргументы, отсутствие объекта, отсутствие доступа, прочие.

`mpo_check_bpfdesc_receive`

```
int mpo_check_bpfdesc_receive(struct bpf_d *bpf_d, struct label *bpflabel,
    struct ifnet *ifnet, struct label *ifnetlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `bpf_d` | Субъект; Дескриптор BPF | |
| `bpflabel` | Метка политики для `bpf_d` | |
| `ifnet` | Объект; сетевой интерфейс | |
| `ifnetlabel` | Метка политики для `ifnet` | |

Определить, должен ли framework MAC разрешать доставку датаграмм с переданного интерфейса в буферы переданного BPF-дескриптора. Возвращает (0) при успехе или значение `errno` при ошибке. Рекомендуемые ошибки: EACCES при несоответствии меток, EPERM при отсутствии привилегий.

`mpo_check_kenv_dump`

```
int mpo_check_kenv_dump(struct ucred *cred);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |

Определить, следует ли разрешить субъекту получать доступ к окружению ядра (см. kenv(2)).

### mpo_check_kenv_get

```
int mpo_check_kenv_get(struct ucred *cred, char *name);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| name | Имя переменной окружения ядра | |

Определить, следует ли разрешить субъекту получать значение указанной переменной окружения ядра.

### mpo_check_kenv_set

```
int mpo_check_kenv_set(struct ucred *cred, char *name);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| name | Имя переменной окружения ядра | |

Определить, следует ли разрешить субъекту устанавливать указанную переменную окружения ядра.

### mpo_check_kenv_unset

```
int mpo_check_kenv_unset(struct ucred *cred, char *name);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| name | Имя переменной окружения ядра | |

Определить, следует ли разрешить субъекту сбросить указанную переменную окружения

ядра.

`mpo_check_kld_load`

```
int mpo_check_kld_load(struct ucred *cred, struct vnode *vp,
    struct label *vlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `vp` | vnode модуля ядра | |
| `vlabel` | Метка, связанная с `vp` | |

Определить, следует ли разрешить субъекту загружать указанный файл модуля.

`mpo_check_kld_stat`

```
int mpo_check_kld_stat(struct ucred *cred);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |

Определить, следует ли разрешить субъекту получать список загруженных файлов модулей ядра и связанную с ними статистику.

`mpo_check_kld_unload`

```
int mpo_check_kld_unload(struct ucred *cred);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |

Определить, следует ли разрешить субъекту выгружать модуль ядра.

`mpo_check_pipe_ioctl`

```
int mpo_check_pipe_ioctl(struct ucred *cred, struct pipe *pipe,
    struct label *pipelabel, unsigned long cmd, void *data);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `pipe` | Канал | |

| Параметр | Описание | Блокировка |
|---|---|---|
| `pipelabel` | Метка политики, связанная с `pipe` | |
| `cmd` | команда nioctl(2) | |
| `data` | данные ioctl(2) | |

Определить, следует ли разрешить субъекту выполнять указанный вызов ioctl(2).

### `mpo_check_pipe_poll`

```
int mpo_check_pipe_poll(struct ucred *cred, struct pipe *pipe,
    struct label *pipelabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `pipe` | Канал | |
| `pipelabel` | Метка политики, связанная с `pipe` | |

Определить, следует ли разрешить субъекту опрашивать `pipe`.

### `mpo_check_pipe_read`

```
int mpo_check_pipe_read(struct ucred *cred, struct pipe *pipe,
    struct label *pipelabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `pipe` | Канал | |
| `pipelabel` | Метка политики, связанная с `pipe` | |

Определить, следует ли разрешить субъекту доступ на чтение к `pipe`.

### `mpo_check_pipe_relabel`

```
int mpo_check_pipe_relabel(struct ucred *cred, struct pipe *pipe,
    struct label *pipelabel, struct label *newlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `pipe` | Канал | |

| Параметр | Описание | Блокировка |
|---|---|---|
| pipelabel | Текущая метка политики, связанная с pipe | |
| newlabel | Обновление метки до pipelabel | |

Определить, следует ли разрешить субъекту изменять метку pipe.

### mpo_check_pipe_stat

```
int mpo_check_pipe_stat(struct ucred *cred, struct pipe *pipe,
    struct label *pipelabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| pipe | Канал | |
| pipelabel | Метка политики, связанная с pipe | |

Определить, следует ли разрешить субъекту получать статистику, связанную с pipe.

### mpo_check_pipe_write

```
int mpo_check_pipe_write(struct ucred *cred, struct pipe *pipe,
    struct label *pipelabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| pipe | Канал | |
| pipelabel | Метка политики, связанная с pipe | |

Определить, следует ли разрешить субъекту запись в pipe.

### mpo_check_socket_bind

```
int mpo_check_socket_bind(struct ucred *cred, struct socket *socket,
    struct label *socketlabel, struct sockaddr *sockaddr);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| socket | Сокет для привязки | |

| Параметр | Описание | Блокировка |
|---|---|---|
| `socketlabel` | Метка политики для `socket` | |
| `sockaddr` | Адрес `socket` | |

### `mpo_check_socket_connect`

```
int mpo_check_socket_connect(struct ucred *cred, struct socket *socket,
    struct label *socketlabel, struct sockaddr *sockaddr);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `socket` | Сокет для подключения | |
| `socketlabel` | Метка политики для `socket` | |
| `sockaddr` | Адрес `socket` | |

Определить, может ли субъект с учётными данными (`cred`) подключить переданный сокет (`socket`) к переданному адресу сокета (`sockaddr`). Возвращает 0 при успехе или значение `errno` при ошибке. Рекомендуемые ошибки: EACCES при несоответствии меток, EPERM при отсутствии прав.

### `mpo_check_socket_receive`

```
int mpo_check_socket_receive(struct ucred *cred, struct socket *so,
    struct label *socketlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `so` | Сокет | |
| `socketlabel` | Метка политики, связанная с `so` | |

Определить, следует ли разрешить субъекту получать информацию из сокета `so`.

### `mpo_check_socket_send`

```
int mpo_check_socket_send(struct ucred *cred, struct socket *so,
    struct label *socketlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `so` | Сокет | |

| Параметр | Описание | Блокировка |
|---|---|---|
| socketlabel | Метка политики, связанная с so | |

Определить, следует ли разрешить субъекту передавать информацию через сокет so.

### mpo_check_cred_visible

```
int mpo_check_cred_visible(struct ucred *u1, struct ucred *u2);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| u1 | Учётные данные субъекта | |
| u2 | Учётные данные объекта | |

Определить, может ли субъект с учётными данными u1 "видеть" другие субъекты с переданными учётными данными u2. Возвращает 0 при успехе или значение errno при ошибке. Рекомендуемые ошибки: EACCES при несоответствии меток, EPERM при отсутствии привилегий или ESRCH для скрытия видимости. Этот вызов может выполняться в различных ситуациях, включая системные вызовы состояния межпроцессного взаимодействия, используемые ps, и при поиске в procfs.

### mpo_check_socket_visible

```
int mpo_check_socket_visible(struct ucred *cred, struct socket *socket,
    struct label *socketlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| socket | Объект; сокет | |
| socketlabel | Метка политики для socket | |

### mpo_check_ifnet_relabel

```
int mpo_check_ifnet_relabel(struct ucred *cred, struct ifnet *ifnet,
    struct label *ifnetlabel, struct label *newlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| ifnet | Объект; сетевой интерфейс | |
| ifnetlabel | Существующая метка политики для ifnet | |

| Параметр | Описание | Блокировка |
|---|---|---|
| `newlabel` | Обновление метки политики для последующего применения к `ifnet` | |

Определить, может ли учётные данные субъекта перемаркировать переданный сетевой интерфейс в соответствии с переданным обновлением метки.

`mpo_check_socket_relabel`

```
int mpo_check_socket_relabel(struct ucred *cred, struct socket *socket,
    struct label *socketlabel, struct label *newlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `socket` | Объект; сокет | |
| `socketlabel` | Метка существующей политики для `socket` | |
| `newlabel` | Обновление метки для последующего применения к `socketlabel` | |

Определить, могут ли учётные данные субъекта перемаркировать переданный сокет в соответствии с переданным обновлением метки.

`mpo_check_cred_relabel`

```
int mpo_check_cred_relabel(struct ucred *cred, struct label *newlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `newlabel` | Обновление метки для последующего применения к `cred` | |

Определить, могут ли учётные данные субъекта перемаркировать себя в соответствии с переданным обновлением метки.

`mpo_check_vnode_relabel`

```
int mpo_check_vnode_relabel(struct ucred *cred, struct vnode *vp,
    struct label *vnodelabel, struct label *newlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | Неизменяемый |
| `vp` | Объект; vnode | Заблокирован |
| `vnodelabel` | Существующая метка политики для `vp` | |
| `newlabel` | Обновление метки политики для последующего применения к `vp` | |

Определить, могут ли учётные данные субъекта изменить метку переданного vnode на переданную обновлённую метку.

#### `mpo_check_mount_stat`

```
int mpo_check_mount_stat(struct ucred *cred, struct mount *mp,
    struct label *mountlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `mp` | Объект; точка монтирования файловой системы | |
| `mountlabel` | Метка политики для `mp` | |

Определить, могут ли учётные данные субъекта видеть результаты выполнения statfs для файловой системы. Возвращает 0 при успехе или значение `errno` при ошибке. Рекомендуемые ошибки: EACCES при несоответствии меток или EPERM при отсутствии привилегий. Этот вызов может выполняться в различных ситуациях, включая вызовы statfs(2) и связанных функций, а также для определения, какие файловые системы исключать из списка, например, при вызове getfsstat(2).

#### `mpo_check_proc_debug`

```
int mpo_check_proc_debug(struct ucred *cred, struct proc *proc);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | Неизменяемый |
| `proc` | Объект; процесс | |

Определить, могут ли учётные данные субъекта отлаживать переданный процесс. Возвращает 0 при успехе или значение `errno` при ошибке. Рекомендуемые ошибки: EACCES при несоответствии метки, EPERM при недостатке прав или ESRCH для скрытия видимости цели. Этот вызов может использоваться в различных ситуациях, включая использование API ptrace(2) и ktrace(2), а также для некоторых операций с procfs.

### mpo_check_vnode_access

```
int mpo_check_vnode_access(struct ucred *cred, struct vnode *vp,
    struct label *label, int flags);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| vp | Объект; vnode | |
| label | Метка политики для vp | |
| flags | флаги access(2) | |

Определить, как должны возвращаться вызовы access(2) и связанные вызовы для субъекта с указанными учётными данными при выполнении на переданном vnode с использованием переданных флагов доступа. Обычно это должно быть реализовано с использованием той же семантики, что и в mpo_check_vnode_open. Возвращает 0 при успехе или значение errno при ошибке. Рекомендуемые ошибки: EACCES при несоответствии меток или EPERM при отсутствии привилегий.

### mpo_check_vnode_chdir

```
int mpo_check_vnode_chdir(struct ucred *cred, struct vnode *dvp,
    struct label *dlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| dvp | Объект; vnode, в который делается chdir(2) | |
| dlabel | Метка политики для dvp | |

Определить, могут ли учётные данные субъекта изменить рабочий каталог процесса на переданный vnode. Возвращает 0 при успехе или значение errno при ошибке. Рекомендуемые ошибки: EACCES при несоответствии метки или EPERM при отсутствии привилегий.

### mpo_check_vnode_chroot

```
int mpo_check_vnode_chroot(struct ucred *cred, struct vnode *dvp,
    struct label *dlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| dvp | vnode каталога | |

| Параметр | Описание | Блокировка |
|---|---|---|
| dlabel | Метка политики, связанная с dvp | |

Определить, следует ли разрешить субъекту выполнять chroot(2) в указанный каталог (dvp).

### mpo_check_vnode_create

```
int mpo_check_vnode_create(struct ucred *cred, struct vnode *dvp,
    struct label *dlabel, struct componentname *cnp, struct vattr *vap);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| dvp | Объект; vnode | |
| dlabel | Метка политики для dvp | |
| cnp | Название компонента для dvp | |
| vap | атрибуты vnode для vap | |

Определить, могут ли учётные данные субъекта создать vnode с указанным родительским каталогом, информацией о имени и атрибутами. Возвращает 0 при успехе или значение errno при ошибке. Рекомендуемые ошибки: EACCES при несоответствии метки или EPERM при отсутствии привилегий. Этот вызов может выполняться в различных ситуациях, включая вызовы open(2) с O_CREAT, mkfifo(2) и другие.

### mpo_check_vnode_delete

```
int mpo_check_vnode_delete(struct ucred *cred, struct vnode *dvp,
    struct label *dlabel, struct vnode *vp, void *label,
    struct componentname *cnp);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| dvp | Родительский каталог vnode | |
| dlabel | Метка политики для dvp | |
| vp | Объект; vnode для удаления | |
| label | Метка политики для vp | |
| cnp | Название компонента для vp | |

Определить, может ли субъект с данными учётными данными удалить vnode из переданного родительского каталога и переданной информации о имени. Возвращает 0 при успехе или значение errno при ошибке. Рекомендуемые ошибки: EACCES при несоответствии метки или EPERM при отсутствии привилегий. Этот вызов может быть

выполнен в различных ситуациях, включая вызовы unlink(2) и rmdir(2). Политики, реализующие эту точку входа, также должны реализовывать mpo_check_rename_to для авторизации удаления объектов в результате их переименования.

### mpo_check_vnode_deleteacl

```
int mpo_check_vnode_deleteacl(struct ucred *cred, struct vnode *vp,
    struct label *label, acl_type_t type);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | Неизменяемый |
| vp | Объект; vnode | Заблокирован |
| label | Метка политики для vp | |
| type | Тип ACL | |

Определить, могут ли учётные данные субъекта удалить ACL указанного типа из переданного vnode. Возвращает 0 при успехе или значение errno при ошибке. Рекомендуемые ошибки: EACCES при несоответствии метки или EPERM при отсутствии привилегий.

### mpo_check_vnode_exec

```
int mpo_check_vnode_exec(struct ucred *cred, struct vnode *vp,
    struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| vp | Объект; vnode для выполнения | |
| label | Метка политики для vp | |

Определить, могут ли учётные данные субъекта выполнить переданный vnode. Проверка права на выполнение осуществляется отдельно от решений о любом переходном событии. Возвращает 0 при успехе или значение errno при ошибке. Рекомендуемые ошибки: EACCES при несоответствии метки или EPERM при отсутствии прав.

### mpo_check_vnode_getacl

```
int mpo_check_vnode_getacl(struct ucred *cred, struct vnode *vp,
    struct label *label, acl_type_t type);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `vp` | Объект; vnode | |
| `label` | Метка политики для `vp` | |
| `type` | Тип ACL | |

Определить, может ли учётное данное субъекта получить ACL указанного типа из переданного vnode. Возвращает 0 при успехе или значение `errno` при ошибке. Рекомендуемые ошибки: EACCES при несоответствии метки или EPERM при отсутствии привилегий.

### `mpo_check_vnode_getextattr`

```
int mpo_check_vnode_getextattr(struct ucred *cred, struct vnode *vp,
    struct label *label, int attrnamespace, const char *name, struct uio *uio);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `vp` | Объект; vnode | |
| `label` | Метка политики для `vp` | |
| `attrnamespace` | Пространство имён расширенных атрибутов | |
| `name` | Имя расширенного атрибута | |
| `uio` | Указатель структуры ввода-вывода; см. uio(9) | |

Определить, может ли субъект с указанными учётными данными получить расширенный атрибут с заданным пространством имён и именем из указанного vnode. Политики, реализующие маркировку с использованием расширенных атрибутов, могут требовать особой обработки операций с этими атрибутами. Возвращает 0 при успехе или значение `errno` при ошибке. Рекомендуемые ошибки: EACCES при несоответствии метки или EPERM при отсутствии привилегий.

### `mpo_check_vnode_link`

```
int mpo_check_vnode_link(struct ucred *cred, struct vnode *dvp,
    struct label *dlabel, struct vnode *vp, struct label *label,
    struct componentname *cnp);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |

| Параметр | Описание | Блокировка |
|---|---|---|
| `dvp` | vnode каталога | |
| `dlabel` | Метка политики, связанная с `dvp` | |
| `vp` | vnode целевого линка | |
| `label` | Метка политики, связанная с `vp` | |
| `cnp` | Имя компонента для создаваемой ссылки | |

Определить, следует ли разрешить субъекту создавать ссылку на vnode `vp` с именем, указанным в `cnp`.

#### `mpo_check_vnode_mmap`

```
int mpo_check_vnode_mmap(struct ucred *cred, struct vnode *vp,
    struct label *label, int prot);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `vp` | vnode для mmap | |
| `label` | Метка политики, связанная с `vp` | |
| `prot` | Защита mmap (см. mmap(2)) | |

Определить, следует ли разрешить субъекту отображать vnode `vp` с указанными в `prot` правами доступа.

#### `mpo_check_vnode_mmap_downgrade`

```
void mpo_check_vnode_mmap_downgrade(struct ucred *cred, struct vnode *vp,
    struct label *label, int *prot);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | См. `mpo_check_vnode_mmap`. | |
| `vp` | | |
| `label` | | |
| `prot` | Защита mmap для понижения уровня | |

Понизить уровень защиты mmap на основе меток субъекта и объекта.

### mpo_check_vnode_mprotect

```
int mpo_check_vnode_mprotect(struct ucred *cred, struct vnode *vp,
    struct label *label, int prot);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| vp | Отображенный vnode | |
| prot | Защита памяти | |

Определить, следует ли разрешить субъекту устанавливать указанные защиты памяти для памяти, отображенной из vnode vp.

### mpo_check_vnode_poll

```
int mpo_check_vnode_poll(struct ucred *active_cred, struct ucred *file_cred,
    struct vnode *vp, struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| active_cred | Учётные данные субъекта | |
| file_cred | Учётные данные, связанные со структурой file | |
| vp | vnode, на котором вызывается poll | |
| label | Метка политики, связанная с vp | |

Определить, следует ли разрешить субъекту вызывать poll на vnode vp.

### mpo_check_vnode_rename_from

```
int mpo_vnode_rename_from(struct ucred *cred, struct vnode *dvp,
    struct label *dlabel, struct vnode *vp, struct label *label,
    struct componentname *cnp);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| dvp | vnode каталога | |
| dlabel | Метка политики, связанная с dvp | |
| vp | vnode для переименования | |

| Параметр | Описание | Блокировка |
|---|---|---|
| label | Метка политики, связанная с vp | |
| cnp | Название компонента для vp | |

Определить, следует ли разрешить субъекту переименовать vnode vp во что-то другое.

### mpo_check_vnode_rename_to

```
int mpo_check_vnode_rename_to(struct ucred *cred, struct vnode *dvp,
    struct label *dlabel, struct vnode *vp, struct label *label, int samedir,
    struct componentname *cnp);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| dvp | vnode каталога | |
| dlabel | Метка политики, связанная с dvp | |
| vp | vnode, который будет перезаписан | |
| label | Метка политики, связанная с vp | |
| samedir | Логическое значение; 1, если исходный и целевой каталоги совпадают | |
| cnp | Имя компонента назначения | |

Определить, следует ли разрешить субъекту переименование vnode vp в каталог dvp или в имя, представленное cnp. Если не существует файла для перезаписи, vp и label будут NULL.

### mpo_check_socket_listen

```
int mpo_check_socket_listen(struct ucred *cred, struct socket *socket,
    struct label *socketlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| socket | Объект; сокет | |
| socketlabel | Метка политики для socket | |

Определить, могут ли учётные данные субъекта прослушивать переданный сокет (вызывать listen). Возвращает 0 при успехе или значение errno при ошибке. Рекомендуемые

ошибки: EACCES при несоответствии метки или EPERM при отсутствии привилегий.

### mpo_check_vnode_lookup

```
int mpo_check_vnode_lookup(struct ucred *cred, struct vnode *dvp,
    struct label *dlabel, struct componentname *cnp);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| dvp | Объект; vnode | |
| dlabel | Метка политики для dvp | |
| cnp | Имя компонента, который ищется | |

Определить, могут ли учётные данные субъекта выполнить поиск указанного имени в каталог с переданном vnode. Возвращает 0 при успехе или значение errno при ошибке. Рекомендуемые ошибки: EACCES при несоответствии метки или EPERM при отсутствии привилегий.

### mpo_check_vnode_open

```
int mpo_check_vnode_open(struct ucred *cred, struct vnode *vp,
    struct label *label, int acc_mode);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| vp | Объект; vnode | |
| label | Метка политики для vp | |
| acc_mode | режим доступа от open(2) | |

Определить, могут ли учётные данные субъекта выполнить операцию открытия переданного vnode с указанным режимом доступа. Возвращает 0 в случае успеха или значение errno при ошибке. Рекомендуемые ошибки: EACCES при несоответствии метки или EPERM при отсутствии привилегий.

### mpo_check_vnode_readdir

```
int mpo_check_vnode_readdir(struct ucred *cred, struct vnode *dvp,
    struct label *dlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |

| Параметр | Описание | Блокировка |
|---|---|---|
| dvp | Объект; vnode каталога | |
| dlabel | Метка политики для dvp | |

Определить, могут ли учётные данные субъекта выполнить операцию readdir для переданного vnode каталога. Возвращает 0 при успехе или значение errno при ошибке. Рекомендуемые ошибки: EACCES при несоответствии метки или EPERM при отсутствии привилегий.

### mpo_check_vnode_readlink

```
int mpo_check_vnode_readlink(struct ucred *cred, struct vnode *vp,
    struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| vp | Объект; vnode | |
| label | Метка политики для vp | |

Определить, могут ли учётные данные субъекта выполнить операцию readlink для переданного символьного vnode. Возвращает 0 в случае успеха или значение errno в случае ошибки. Рекомендуемые ошибки: EACCES при несоответствии метки или EPERM при отсутствии привилегий. Этот вызов может быть выполнен в различных ситуациях, включая явный вызов readlink пользовательским процессом или неявный readlink во время поиска имени процессом.

### mpo_check_vnode_revoke

```
int mpo_check_vnode_revoke(struct ucred *cred, struct vnode *vp,
    struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| vp | Объект; vnode | |
| label | Метка политики для vp | |

Определить, могут ли учётные данные субъекта отозвать доступ к переданному vnode. Возвращает 0 при успехе или значение errno при ошибке. Рекомендуемые ошибки: EACCES при несоответствии метки или EPERM при отсутствии привилегий.

### mpo_check_vnode_setacl

```
int mpo_check_vnode_setacl(struct ucred *cred, struct vnode *vp,
```

```
    struct label *label, acl_type_t type, struct acl *acl);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `vp` | Объект; vnode | |
| `label` | Метка политики для `vp` | |
| `type` | Тип ACL | |
| `acl` | ACL | |

Определить, могут ли учётные данные субъекта установить переданный ACL указанного типа для переданного vnode. Возвращает 0 при успехе или значение `errno` при ошибке. Рекомендуемые ошибки: EACCES при несоответствии метки или EPERM при отсутствии привилегий.

### `mpo_check_vnode_setextattr`

```
int mpo_check_vnode_setextattr(struct ucred *cred, struct vnode *vp,
    struct label *label, int attrnamespace, const char *name, struct uio *uio);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `vp` | Объект; vnode | |
| `label` | Метка политики для `vp` | |
| `attrnamespace` | Пространство имён расширенных атрибутов | |
| `name` | Имя расширенного атрибута | |
| `uio` | Указатель структуры ввода-вывода; см. uio(9) | |

Определить, могут ли учётные данные субъекта установить расширенный атрибут с переданным именем и пространством имён на переданном vnode. Политики, реализующие метки безопасности, основанные на расширенных атрибутах, могут предусматривать дополнительные защиты для этих атрибутов. Кроме того, политикам следует избегать принятия решений на основе данных, на которые ссылается `uio`, так как существует потенциальное состояние гонки между этой проверкой и фактической операцией. `uio` также может быть `NULL`, если выполняется операция удаления. Возвращает 0 при успехе или значение `errno` при ошибке. Рекомендуемые ошибки: EACCES при несоответствии метки или EPERM при отсутствии привилегий.

### `mpo_check_vnode_setflags`

```
int mpo_check_vnode_setflags(struct ucred *cred, struct vnode *vp,
```

```
    struct label *label, u_long flags);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `vp` | Объект; vnode | |
| `label` | Метка политики для `vp` | |
| `flags` | Флаги файла; см. chflags(2) | |

Определить, могут ли учётные данные субъекта установить переданные флаги на переданном vnode. Возвращает 0 при успехе или значение `errno` при ошибке. Рекомендуемые ошибки: EACCES при несоответствии метки или EPERM при отсутствии привилегий.

### `mpo_check_vnode_setmode`

```
int mpo_check_vnode_setmode(struct ucred *cred, struct vnode *vp,
    struct label *label, mode_t mode);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `vp` | Объект; vnode | |
| `label` | Метка политики для `vp` | |
| `mode` | Режим файла; см. chmod(2) | |

Определить, могут ли учётные данные субъекта установить переданный режим для переданного vnode. Возвращает 0 при успехе или значение `errno` при ошибке. Рекомендуемые ошибки: EACCES при несоответствии метки или EPERM при недостатке привилегий.

### `mpo_check_vnode_setowner`

```
int mpo_check_vnode_setowner(struct ucred *cred, struct vnode *vp,
    struct label *label, uid_t uid, gid_t gid);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `vp` | Объект; vnode | |
| `label` | Метка политики для `vp` | |
| `uid` | User ID | |
| `gid` | Идентификатор группы | |

Определить, могут ли учётные данные субъекта установить переданный uid и переданный gid в качестве uid файла и gid файла для переданного vnode. Идентификаторы могут быть установлены в (-1) для запроса отсутствия обновления. Возвращает 0 при успехе или значение errno при ошибке. Рекомендуемые ошибки: EACCES при несоответствии метки или EPERM при отсутствии привилегий.

#### mpo_check_vnode_setutimes

```
int mpo_check_vnode_setutimes(struct ucred *cred, struct vnode *vp,
    struct label *label, struct timespec atime, struct timespec mtime);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| vp | Объект; vp | |
| label | Метка политики для vp | |
| atime | Время доступа; см. utimes(2) | |
| mtime | Время изменения; см. utimes(2) | |

Определить, могут ли учётные данные субъекта установить переданные времена доступа на переданном vnode. Возвращает 0 при успехе или значение errno при ошибке. Рекомендуемые ошибки: EACCES при несоответствии метки или EPERM при отсутствии привилегий.

#### mpo_check_proc_sched

```
int mpo_check_proc_sched(struct ucred *ucred, struct proc *proc);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| proc | Объект; процесс | |

Определить, могут ли учётные данные субъекта изменить параметры планирования переданного процесса. Возвращает 0 при успехе или значение errno при ошибке. Рекомендуемые ошибки: EACCES при несоответствии метки, EPERM при отсутствии привилегий или ESRCH для ограничения видимости.

См. setpriority(2) для получения дополнительной информации.

#### mpo_check_proc_signal

```
int mpo_check_proc_signal(struct ucred *cred, struct proc *proc, int signal);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `proc` | Объект; процесс | |
| `signal` | Сигнал; см. kill(2) | |

Определить, могут ли учётные данные субъекта доставить указанный сигнал указанному процессу. Возвращает 0 при успехе или значение `errno` при ошибке. Рекомендуемые коды ошибок: EACCES при несоответствии метки, EPERM при недостатке прав или ESRCH для ограничения видимости.

### `mpo_check_vnode_stat`

```
int mpo_check_vnode_stat(struct ucred *cred, struct vnode *vp,
    struct label *label);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `vp` | Объект; vnode | |
| `label` | Метка политики для `vp` | |

Определить, могут ли учётные данные субъекта выполнять `stat` для переданного vnode. Возвращает 0 при успехе или значение `errno` при ошибке. Рекомендуемые ошибки: EACCES при несоответствии метки или EPERM при отсутствии привилегий.

См. stat(2) для получения дополнительной информации.

### `mpo_check_ifnet_transmit`

```
int mpo_check_ifnet_transmit(struct ucred *cred, struct ifnet *ifnet,
    struct label *ifnetlabel, struct mbuf *mbuf, struct label *mbuflabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `ifnet` | Сетевой интерфейс | |
| `ifnetlabel` | Метка политики для `ifnet` | |
| `mbuf` | Объект; mbuf для отправки | |
| `mbuflabel` | Метка политики для `mbuf` | |

Определить, может ли сетевой интерфейс передать mbuf, переданный в качестве параметра. Возвращает 0 при успехе или значение `errno` при ошибке. Рекомендуемые ошибки: EACCES при несоответствии метки или EPERM при отсутствии привилегий.

### mpo_check_socket_deliver

```
int mpo_check_socket_deliver(struct ucred *cred, struct ifnet *ifnet,
    struct label *ifnetlabel, struct mbuf *mbuf, struct label *mbuflabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| ifnet | Сетевой интерфейс | |
| ifnetlabel | Метка политики для ifnet | |
| mbuf | Объект; mbuf для доставки | |
| mbuflabel | Метка политики для mbuf | |

Определить, может ли сокет принять датаграмму, хранящуюся в переданном заголовке mbuf. Возвращает 0 при успехе или значение errno при ошибке. Рекомендуемые ошибки: EACCES при несоответствии метки или EPERM при отсутствии привилегий.

### mpo_check_socket_visible

```
int mpo_check_socket_visible(struct ucred *cred, struct socket *so,
    struct label *socketlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | Неизменяемый |
| so | Объект; сокет | |
| socketlabel | Метка политики для so | |

Определить, могут ли учётные данные субъекта cred "видеть" переданный сокет (socket), используя функции системного мониторинга, такие как те, что применяются в netstat(8) и sockstat(1). Возвращает 0 при успехе или значение errno при ошибке. Рекомендуемые ошибки: EACCES при несоответствии меток, EPERM при отсутствии привилегий или ESRCH для скрытия видимости.

### mpo_check_system_acct

```
int mpo_check_system_acct(struct ucred *ucred, struct vnode *vp,
    struct label *vlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| ucred | Учётные данные субъекта | |
| vp | Файл учёта; acct(5) | |
| vlabel | Метка, связанная с vp | |

Определить, следует ли разрешить субъекту включение учёта, основываясь на его метке и метке файла журнала учёта.

### mpo_check_system_nfsd

```
int mpo_check_system_nfsd(struct ucred *cred);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |

Определить, следует ли разрешить субъекту вызывать nfssvc(2).

### mpo_check_system_reboot

```
int mpo_check_system_reboot(struct ucred *cred, int howto);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| howto | Параметр howto из reboot(2) | |

Определить, следует ли разрешить субъекту перезагружать систему указанным способом.

### mpo_check_system_settime

```
int mpo_check_system_settime(struct ucred *cred);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |

Определить, разрешено ли пользователю устанавливать системные часы.

### mpo_check_system_swapon

```
int mpo_check_system_swapon(struct ucred *cred, struct vnode *vp,
    struct label *vlabel);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| cred | Учётные данные субъекта | |
| vp | Устройство подкачки | |
| vlabel | Метка, связанная с vp | |

Определить, следует ли разрешить субъекту добавлять vp как устройство подкачки.

`mpo_check_system_sysctl`

```
int mpo_check_system_sysctl(struct ucred *cred, int *name, u_int *namelen,
    void *old, size_t *oldlenp, int inkernel, void *new, size_t newlen);
```

| Параметр | Описание | Блокировка |
|---|---|---|
| `cred` | Учётные данные субъекта | |
| `name` | См. sysctl(3) | |
| `namelen` | | |
| `old` | | |
| `oldlenp` | | |
| `inkernel` | Логический; 1, если вызвано из ядра | |
| `new` | См. sysctl(3) | |
| `newlen` | | |

Определить, следует ли разрешить субъекту выполнять указанную транзакцию sysctl(3).

### 6.7.5. Вызовы при управления метками

События изменения метки происходят, когда пользовательский процесс запрашивает изменение метки на объекте. Происходит двухэтапное обновление: сначала выполняется проверка контроля доступа, чтобы определить, является ли обновление допустимым и разрешённым, а затем само обновление выполняется через отдельную точку входа. Точки входа для изменения метки обычно принимают объект, ссылку на метку объекта и новую метку, предоставленную процессом. Выделение памяти во время изменения метки не рекомендуется, так как вызовы изменения метки не могут завершиться неудачей (ошибка должна быть обнаружена ранее на этапе проверки изменения метки).

## 6.8. Пользовательская архитектура

В фреймворк TrustedBSD MAC входит ряд элементов, не зависящих от политик, включая интерфейсы MAC-библиотеки для абстрактного управления метками, изменения в управлении системными учётными данными и библиотеках входа в систему для поддержки назначения MAC-меток пользователям, а также набор инструментов для мониторинга и изменения меток процессов, файлов и сетевых интерфейсов. Более подробная информация о пользовательской архитектуре будет добавлена в этот раздел в ближайшее время.

### 6.8.1. API для управления метками, не зависящими от политики

Фреймворк TrustedBSD MAC предоставляет ряд библиотечных и системных вызовов, позволяющих приложениям управлять метками MAC на объектах с использованием политико-независимого интерфейса. Это позволяет приложениям манипулировать метками для различных политик без необходимости поддержки конкретных политик. Эти

интерфейсы используются универсальными инструментами, такими как ifconfig(8), ls(1) и ps(1), для просмотра меток на сетевых интерфейсах, файлах и процессах. API также поддерживают инструменты управления MAC, включая getfmac(8), getpmac(8), setfmac(8), setfsmac(8) и setpmac(8). API MAC документированы в mac(3).

Приложения обрабатывают метки MAC в двух формах: внутренней форме, используемой для возврата и установки меток для процессов и объектов (`mac_t`), и внешней форме, основанной на строках C, подходящих для хранения в конфигурационных файлах, отображения пользователю или ввода от пользователя. Каждая метка MAC содержит ряд элементов, каждый из которых состоит из пары имя-значение. Модули политик в ядре привязываются к определённым именам и интерпретируют значения специфичным для политики образом. Во внешней строковой форме метки представляются списком пар имя-значение, разделённых запятыми и символом `/`. Метки могут быть напрямую преобразованы в текст и обратно с использованием предоставленных API; при извлечении меток из ядра внутреннее хранилище меток должно быть сначала подготовлено для желаемого набора элементов метки. Обычно это делается одним из двух способов: с использованием mac_prepare(3) и произвольного списка желаемых элементов метки, или одной из вариаций вызова, который загружает набор элементов по умолчанию из конфигурационного файла mac.conf(5). Значения по умолчанию для каждого объекта позволяют разработчикам приложений удобно отображать метки, связанные с объектами, без необходимости знать о присутствующих в системе политиках.



В настоящее время прямое манипулирование элементами меток, кроме как путем преобразования в текстовую строку, редактирования строки и обратного преобразования во внутреннюю метку, не поддерживается библиотекой MAC. Такие интерфейсы могут быть добавлены в будущем, если окажется, что они необходимы разработчикам приложений.

### 6.8.2. Привязка меток к пользователям

Стандартный интерфейс управления контекстом пользователя, setusercontext(3), был изменён для получения меток MAC, связанных с классом пользователя, из login.conf(5). Эти метки устанавливаются вместе с остальным контекстом пользователя, когда указан `LOGIN_SETALL` или явно указан `LOGIN_SETMAC`.



Ожидается, что в будущей версии FreeBSD база данных меток MAC будет отделена от абстракции классов пользователей login.conf и будет поддерживаться в отдельной базе данных. Однако API setusercontext(3) должно остаться неизменным после такого изменения.

## 6.9. Заключение

Фреймворк TrustedBSD MAC позволяет модулям ядра расширять политику безопасности системы высокоинтегрированным способом. Они могут делать это на основе существующих свойств объектов или данных меток, которые поддерживаются с помощью фреймворка MAC. Фреймворк достаточно гибкий для реализации различных типов политик, включая политики безопасности информационных потоков, такие как MLS и Biba, а также

политики, основанные на существующих учётных данных BSD или защите файлов. Авторам политик может быть полезно ознакомиться с этой документацией, а также с существующими модулями безопасности при реализации новой службы безопасности.

# Глава 7. Система управления виртуальной памятью

## 7.1. Управление физической памятью `vm_page_t`

Физическая память управляется постранично с использованием структуры `vm_page_t`. Страницы физической памяти классифицируются посредством размещения соответствующих структур `vm_page_t` в одной из нескольких очередей подкачки.

Страница может находиться в одном из следующих состояний: зафиксированном (Wired), активном (Active), неактивном (Inactive), кэшированном (Cache) или свободном (Free). За исключением зафиксированного состояния, страница обычно помещается в двусвязный список, представляющий её текущее состояние. Зафиксированные страницы не помещаются ни в одну из очередей.

FreeBSD реализует более сложную очередь подкачки для кэшированных и свободных страниц с целью реализации раскраски страниц. Каждое из этих состояний включает несколько очередей, организованных в соответствии с размерами кэшей L1 и L2 процессора. Когда требуется выделить новую страницу, FreeBSD пытается получить такую, которая будет достаточно хорошо выровнена с точки зрения кэшей L1 и L2 относительно объекта виртуальной памяти, для которого выделяется страница.

Кроме того, страница может удерживаться с помощью счётчика ссылок или быть заблокированной с помощью счётчика занятости. Система виртуальной памяти также реализует состояние «абсолютной блокировки» страницы с использованием бита PG_BUSY в флагах страницы.

В общем случае каждая из очередей подкачки работает по принципу LRU (наименее недавно использованная). Обычно страница изначально помещается в зафиксированное или активное состояние. В зафиксированном состоянии страница, как правило, ассоциирована с какой-либо таблицей страниц. Система виртуальной памяти «состаривает» страницу, просматривая страницы в более активной очереди страниц (LRU), чтобы переместить их в менее активную очередь. Страницы, перемещённые в кэш, всё ещё связаны с объектом виртуальной памяти, но могут быть немедленно повторно использованы. Страницы в очереди свободных действительно являются полностью свободными. FreeBSD стремится минимизировать количество страниц в очереди свободных, но определённый минимальный объём таких страниц должен поддерживаться для обеспечения возможности выделения памяти во время обработки прерываний.

Если процесс пытается обратиться к странице, которой нет в его таблице страниц, но которая присутствует в одной из очередей подкачки (например, в неактивной или кэшированной), происходит относительно недорогой сбой повторной активации страницы, в результате которого страница повторно активируется. Если же страницы вообще нет в памяти системы, процесс вынужден блокироваться до тех пор, пока страница не будет загружена с диска.

FreeBSD динамически настраивает свои очереди страниц и старается поддерживать

разумные пропорции количества страниц в различных очередях, а также сбалансированное соотношение чистых и грязных страниц. Объём перераспределения зависит от текущей нагрузки на память системы. Это перераспределение выполняется демоном выгрузки страниц (pageout daemon) и включает в себя очистку (laundering) грязных страниц (синхронизацию их с резервным хранилищем), отслеживание активности страниц по ссылкам на них (сброс их положения в очередях LRU или перемещение между очередями), миграцию страниц между очередями при разбалансировке очередей и другие действия. Система виртуальной памяти FreeBSD допускает определённое количество сбоев повторной активации страниц, чтобы точнее определить, насколько страница фактически активна или неактивна. Это позволяет принимать более обоснованные решения о том, когда очищать или выгружать страницу в файл подкачки.

## 7.2. Унифицированный буферный кэш `vm_object_t`

FreeBSD реализует концепцию универсального «объекта виртуальной памяти» (VM-объекта). VM-объекты могут быть связаны с резервным хранилищем различных типов: без резервного хранилища, с подкачкой, с физическим устройством или с файловым хранилищем. Поскольку файловая система использует те же VM-объекты для управления данными в оперативной памяти, связанными с файлами, в результате получается унифицированный буферный кэш.

Объекты виртуальной памяти могут быть *затемнены* (shadowed), то есть размещены друг над другом в виде стека. Например, объект виртуальной памяти с подкачкой может быть размещён поверх объекта, связанного с файловым хранилищем, для реализации отображения mmap() с флагом MAP_PRIVATE. Такое стековое размещение также используется для реализации различных общих свойств, включая копирование при записи (copy-on-write), для адресных пространств созданных при форке процесса.

Следует отметить, что структура `vm_page_t` может быть связана только с одним объектом виртуальной памяти (VM-объектом) в данный момент времени. Механизм затемнения VM-объектов реализует кажущееся совместное использование одной и той же страницы между несколькими экземплярами.

## 7.3. Ввод-вывод файловой системы `struct buf`

VM-объекты, поддерживаемые vnode, такие как объекты, связанные с файловым хранилищем, обычно должны поддерживать собственную информацию о чистоте/грязности, независимую от представления о чистоте/грязности в системе виртуальной памяти. Например, когда система виртуальной памяти решает синхронизировать физическую страницу с её резервным хранилищем, система виртуальной памяти должна отметить страницу как чистую до того, как она будет фактически записана в резервное хранилище. Кроме того, файловые системы должны иметь возможность отображать части файла или метаданных файла в KVM для выполнения операций с ними.

Сущности, используемые для управления этим процессом, известны как файловые буферы, структуры `struct buf` или `bp`. Когда файловая система должна работать с частью объекта виртуальной памяти, она обычно отображает часть объекта в структуру struct buf, а затем

отображает страницы из этой структуры в KVM. Точно так же операции с дисковым вводом-выводом обычно выполняются путём отображения частей объектов в структуры буферов и последующего выполнения ввода-вывода на этих структурах. Соответствующие им страницы vm_page_t обычно блокируются на время выполнения операции ввода-вывода. Файловые буферы также имеют собственное представление о состоянии занятости, что полезно для кода драйвера файловой системы, который предпочитает работать с файловыми буферами, а не с обычными страницами виртуальной памяти.

FreeBSD резервирует ограниченный объём виртуальной памяти ядра для хранения отображений из структур struct buf на физические страницы памяти, но следует подчеркнуть, что эта память используется исключительно для хранения отображений и не ограничивает возможность кэширования данных. Кэширование физических данных строго связано с `vm_page_t`, а не с файловыми буферами. Однако, поскольку файловые буферы используются для операций ввода-вывода, они ограничивают количество возможных параллельных операций ввода-вывода. Тем не менее, поскольку обычно доступно несколько тысяч файловых буферов, это редко становится проблемой.

## 7.4. Отображение таблиц страниц `vm_map_t`, `vm_entry_t`

FreeBSD разделяет топологию физических таблиц страниц от системы виртуальной памяти. Все основные таблицы страниц, связанные с каждым процессом, могут быть восстановлены динамически и обычно считаются временными (throwaway). Специальные таблицы страниц, такие как те, которые управляют KVM (виртуальной памятью ядра), обычно предварительно выделяются навсегда. Эти таблицы страниц не являются временными.

FreeBSD связывает части vm_object с диапазонами адресов в виртуальной памяти через структуры `vm_map_t` и `vm_entry_t`. Таблицы страниц напрямую создаются из иерархии `vm_map_t` / `vm_entry_t`/ `vm_object_t`. Напоминаем, что физические страницы непосредственно связаны только с `vm_object`, однако это не совсем так. Структуры `vm_page_t` также привязаны к таблицами страниц, с которыми они активно ассоциированы. Одна структура `vm_page_t` может быть привязана к нескольким *pmaps* (так называют таблицы страниц). Тем не менее, иерархическая ассоциация сохраняется, и все ссылки на одну и ту же страницу в одном объекте ссылаются на один и тот же `vm_page_t`, что, в свою очередь, обеспечивает унификацию буферного кэша.

## 7.5. Отображение виртуальной памяти ядра

FreeBSD использует KVM для хранения различных структур ядра. Самой крупной сущностью, хранящейся в KVM, является кэш файловых буферов. То есть, отображения, связанные с сущностями `struct buf`.

В отличие от Linux, FreeBSD *не* отображает всю физическую память в KVM. Это означает, что FreeBSD может обрабатывать конфигурации памяти до 4 ГБ на 32-битных платформах. На самом деле, если бы MMU это позволял, FreeBSD теоретически могла бы обрабатывать конфигурации памяти до 8 ТБ на 32-битной платформе. Однако, поскольку большинство 32-битных платформ способны отображать только 4 ГБ оперативной памяти, этот вопрос не имеет практического значения.

KVM управляется через несколько механизмов. Основным механизмом для управления KVM является *аллокатор зон*. Аллокатор зон берет блок KVM и делит его на блоки памяти постоянного размера для выделения структур определённого типа. Вы можете использовать команду `vmstat -m`, чтобы получить обзор текущего использования KVM, разбитого по зонам.

## 7.6. Настройка системы виртуальной памяти во FreeBSD

Были преложены значительные усилия сделать ядро FreeBSD динамически настраиваемым. Обычно вам не нужно вмешиваться в настройки, за исключением параметров конфигурации ядра `maxusers` и `NMBCLUSTERS`. Параметры компиляции ядра указанны (обычно) в файле /usr/src/sys/i386/conf/CONFIG_FILE. Описание всех доступных параметров конфигурации ядра можно найти в файле /usr/src/sys/i386/conf/LINT.

В большой системе конфигурации, возможно, вам потребуется увеличить значение `maxusers`. Обычно значения варьируются от 10 до 128. Обратите внимание, что слишком большое значение для `maxusers` может привести к переполнению доступной KVM, что вызовет непредсказуемую работу системы. Лучше оставить `maxusers` на разумном уровне и добавлять другие параметры, такие как `NMBCLUSTERS`, для увеличения конкретных ресурсов.

Если ваша система будет активно использовать сеть, возможно, вам захочется увеличить значение `NMBCLUSTERS`. Типичные значения варьируются от 1024 до 4096.

Параметр `NBUF` также традиционно используется для масштабирования системы. Этот параметр определяет объём виртуальной памяти ядра, который система может использовать для отображения файловых буферов для ввода-вывода. Обратите внимание, что этот параметр никак не связан с унифицированным буферным кэшем! В ядрах 3.0-CURRENT и позднее этот параметр динамически настраивается, и в целом не следует настраивать его вручную. Мы рекомендуем вам *не* пытаться задавать параметр NBUF. Позвольте системе выбрать его автоматически. Слишком малое значение может привести к крайне неэффективной работе файловой системы, в то время как слишком большое значение может привести к истощению очередей страниц из-за того, что слишком много страниц будет закреплено в памяти.

По умолчанию ядра FreeBSD не оптимизированы. Вы можете установить флаги отладки и оптимизации с помощью директивы `makeoptions` в конфигурации ядра. Обратите внимание, что не следует использовать флаг `-g`, если вы не можете разместить в системе большие ядра (обычно более 7 МБ), которые получаются в результате.

```
makeoptions     DEBUG="-g"
makeoptions     COPTFLAGS="-O -pipe"
```

Sysctl предоставляет способ настройки параметров ядра во время работы системы. Обычно вам не нужно изменять какие-либо переменные sysctl, особенно те, что связаны с виртуальной памятью.

Настройка виртуальной памяти (VM) и системы во время работы относительно проста. Во-первых, используйте Soft Updates на ваших файловых системах UFS/FFS, когда это возможно. Файл /usr/src/sys/ufs/ffs/README.softupdates содержит инструкции (и ограничения) по конфигурации этой функции.

Во-вторых, настройте достаточный объём подкачки (swap). Вы должны создать раздел подкачки на каждом физическом диске, до четырёх, даже на "рабочих" дисках. Объём подкачки должен быть как минимум в 2 раза больше объёма основной памяти, а возможно — и больше, если у вас немного ОЗУ. Размер раздела подкачки следует выбирать с учётом максимальной конфигурации памяти, которую вы когда-либо планируете установить на эту машину, чтобы в будущем не пришлось переразбивать диски. Если вы хотите иметь возможность сохранять дампы при сбое, ваш первый swap-раздел должен быть не меньше объёма основной памяти, а каталог /var/crash должен иметь достаточно свободного места для хранения дампа.

Подкачка на базе NFS вполне допустима в системах версии 4.X и новее, однако необходимо учитывать, что основная нагрузка на подкачку ляжет на NFS-сервер.

# Глава 8. Документ по архитектуре SMPng

## 8.1. Введение

В этом документе представлены текущая архитектура и реализация SMPng. Сначала вводятся основные примитивы и инструменты. Затем излагается общая архитектура модели синхронизации и выполнения ядра FreeBSD. Далее обсуждаются стратегии блокировок для конкретных подсистем, описывающие подходы к внедрению детализированной синхронизации и параллелизма для каждой подсистемы. В заключение приводятся подробные заметки по реализации, объясняющие выбор проектных решений и информирующие читателя о важных последствиях использования конкретных примитивов.

Этот документ находится в стадии разработки и будет обновляться в соответствии с текущими проектированием и реализацией, связанными с проектом SMPng. Многие разделы в настоящее время существуют только в виде набросков, но будут дополняться по мере продвижения работы. Обновления или предложения по документу могут быть направлены редакторам документа.

Цель SMPng — обеспечить параллелизм в ядре. Ядро представляет собой одну довольно большую и сложную программу. Чтобы сделать ядро многопоточным, мы используем те же инструменты, что и для многопоточности других программ. К ним относятся мьютексы, разделяемые/монопольные блокировки, семафоры и условные переменные. Для определений этих и других терминов, связанных с SMP, см. раздел Глоссарий в этой статье.

## 8.2. Основные инструменты и основы блокировки

### 8.2.1. Атомарные инструкции и барьеры памяти

Можно найти много описаний барьеров памяти и атомарных инструкций, поэтому в этом разделе не будет много деталей. Проще говоря, нельзя читать переменные без блокировки, если блокировка используется для защиты записи в эту переменную. Это становится очевидным, если учесть, что барьеры памяти лишь определяют относительный порядок операций с памятью; они не дают никаких гарантий относительно времени выполнения этих операций. То есть, барьер памяти не принуждает к сбросу содержимого локального кэша или буфера записи процессора. Вместо этого, барьер памяти при освобождении блокировки просто гарантирует, что все записи в защищённые данные будут видны другим процессорам или устройствам, если видна запись, освобождающая блокировку. Процессор может хранить эти данные в своём кэше или буфере записи сколько угодно долго. Однако, если другой процессор выполняет атомарную инструкцию над тем же данным, первый процессор должен гарантировать, что обновлённое значение будет видно второму процессору, наряду с любыми другими операциями, которые могут потребоваться согласно барьерам памяти.

Например, предполагая простую модель, в которой данные считаются видимыми, когда они находятся в основной памяти (или в глобальном кэше), когда начинается выполнение атомарной инструкции на одном процессоре, буферы записи и кэши других процессоров

должны выполнить все записи в ту же строку кэша вместе с любыми ожидающими операциями за барьером памяти.

Это требует особой осторожности при использовании элемента, защищённого атомарными инструкциями. Например, в реализации мьютекса сна мы должны использовать `atomic_cmpset` вместо `atomic_set` для установки бита `MTX_CONTESTED`. Причина в том, что мы считываем значение `mtx_lock` в переменную и затем принимаем решение на основе этого чтения. Однако значение, которое мы ранее прочитали, может быть устаревшим или измениться, пока мы принимаем решение. Таким образом, когда выполняется `atomic_set`, это может привести к установке бита на другом значении, отличном от того, на котором мы основывали своё решение. Поэтому мы должны использовать `atomic_cmpset`, чтобы установить значение только в том случае, если значение, на котором мы приняли решение, актуально и действительно.

Наконец, атомарные инструкции позволяют обновить или прочитать только один элемент. Если необходимо атомарно обновить несколько элементов, вместо этого следует использовать блокировку. Например, если требуется прочитать два счётчика и получить их значения, согласованные друг с другом, то эти счётчики должны быть защищены блокировкой, а не отдельными атомарными инструкциями.

### 8.2.2. Блокировки на чтение и блокировки на запись

Блокировки на чтение не требуют такой же строгости, как блокировки на запись. Оба типа блокировок должны гарантировать, что данные, к которым они обращаются, не устарели. Однако запись требует монопольного доступа. Несколько потоков могут безопасно читать значение. Использование разных типов блокировок для чтения и записи может быть реализовано несколькими способами.

Во-первых, блокировки sx могут использоваться таким образом: монопольная блокировка при записи и разделяемая блокировка при чтении. Этот метод достаточно прост.

Второй метод несколько менее очевиден. Вы можете защитить данные несколькими блокировками. Для чтения данных достаточно получить блокировку на чтение одной из блокировок. Однако для записи данных необходимо получить блокировку на запись всех блокировок. Это может сделать запись довольно затратной, но может быть полезно, когда данные доступны различными способами. Например, указатель на родительский процесс защищён как `proctree_lock` sx-блокировкой, так и мьютексом процесса. Иногда блокировка процесса удобнее, так как мы просто проверяем, кто является родителем уже заблокированного процесса. Однако в других случаях, таких как `inferior`, необходимо обходить дерево процессов через указатели на родителя, и блокировка каждого процесса была бы слишком затратной, а также сложной для гарантии того, что проверяемое условие остаётся верным как во время проверки, так и при выполнении действий, основанных на этой проверке.

### 8.2.3. Условия и результаты блокировки

Если вам нужна блокировка для проверки состояния переменной, чтобы можно было выполнить действие на основе прочитанного состояния, вы не можете просто удерживать блокировку во время чтения переменной, а затем снять блокировку перед выполнением

действия на основе прочитанного значения. Как только вы снимаете блокировку, переменная может измениться, что сделает ваше решение недействительным. Таким образом, вы должны удерживать блокировку как во время чтения переменной, так и во время выполнения действия в результате проверки.

# 8.3. Общая архитектура и дизайн

## 8.3.1. Обработка прерываний

Следуя примеру нескольких других многопоточных ядер UNIX®, FreeBSD реализовала обработчики прерываний, предоставив им собственный контекст потока. Предоставление контекста для обработчиков прерываний позволяет им блокироваться на блокировках. Однако, чтобы избежать задержек, потоки обработки прерываний выполняются с приоритетом реального времени в ядре. Таким образом, обработчики прерываний не должны выполняться слишком долго, чтобы не лишать ресурсов другие потоки ядра. Кроме того, поскольку несколько обработчиков могут использовать один поток прерываний, обработчики прерываний не должны переходить в режим сна или использовать блокировки, допускающие сон, чтобы не лишать ресурсов другие обработчики прерываний.

Текущие потоки обработки прерываний в FreeBSD называются тяжеловесными потоками обработки прерываний. Они получили такое название, потому что переключение на поток обработки прерывания включает в себя полное переключение контекста. В первоначальной реализации ядро не было вытесняющим, поэтому прерывания, которые прерывали поток ядра, должны были ждать, пока поток ядра не заблокируется или не вернётся в пользовательское пространство, прежде чем у них появится возможность выполниться.

Для решения проблем с задержками ядро FreeBSD стало вытесняющим. В настоящее время вытеснение потока ядра происходит только при освобождении мьютекса сна или при поступлении прерывания. Однако планируется сделать ядро FreeBSD полностью вытесняющим, как описано ниже.

Не все обработчики прерываний выполняются в контексте потока. Вместо этого, некоторые обработчики выполняются непосредственно в основном контексте прерывания. Эти обработчики прерываний в настоящее время ошибочно называются "быстрыми" обработчиками прерываний, поскольку для их обозначения применяется флаг `INTR_FAST`, использовавшийся в более ранних версиях ядра. Единственные прерывания, которые в настоящее время используют такие обработчики прерываний, — это прерывания от часов и последовательных устройств ввода-вывода. Поскольку эти обработчики не имеют собственного контекста, они не могут захватывать блокирующие блокировки и, следовательно, могут использовать только спин-мьютексы.

Наконец, существует одна дополнительная оптимизация, которую можно добавить в код MD, называемая легковесными переключениями контекста. Поскольку поток обработки прерывания выполняется в контексте ядра, он может заимствовать vmspace любого процесса. Таким образом, при легковесном переключении контекста переход к потоку обработки прерывания не меняет vmspace, а заимствует vmspace прерванного потока. Чтобы гарантировать, что vmspace прерванного потока не исчезнет во время работы,

прерванному потоку запрещается выполнение до тех пор, пока поток обработки прерывания больше не использует его vmspace. Это может произойти, когда поток обработки прерывания либо блокируется, либо завершается. Если поток обработки прерывания блокируется, то при повторном запуске он будет использовать свой собственный контекст. Таким образом, он может освободить прерванный поток.

Недостатки этой оптимизации заключаются в том, что они очень специфичны для конкретной машины и сложны, поэтому стоят усилий только в случае значительного улучшения производительности. На данный момент, вероятно, ещё рано делать выводы, и, фактически, это может даже ухудшить производительность, так как почти все обработчики прерываний будут немедленно блокироваться на Giant и потребуют исправления потока при блокировке. Кроме того, Майк Смит предложил альтернативный метод обработки прерываний, который работает следующим образом:

1. Каждый обработчик прерывания состоит из двух частей: предиката, который выполняется в основном контексте прерывания, и обработчика, который выполняется в контексте собственного потока.
2. Если у обработчика прерывания есть предикат, то при срабатывании прерывания этот предикат выполняется. Если предикат возвращает значение `true`, прерывание считается полностью обработанным, и ядро возвращается из прерывания. Если предикат возвращает `false` или предиката нет, то система планирует запуск обработчика в контексте собственного потока.

Встраивание легковесных переключений контекста в эту схему может оказаться довольно сложным. Поскольку мы, возможно, захотим перейти на эту схему в будущем, вероятно, лучше отложить работу над легковесными переключениями контекста до тех пор, пока мы не определимся с окончательной архитектурой обработки прерываний и не выясним, как легковесные переключения контекста могут (или не могут) в неё вписаться.

## 8.3.2. Ядро с вытеснением и критические секции

#### Ядро и вытеснение вкратце

Вытеснение ядра довольно просто. Основная идея заключается в том, что процессор всегда должен выполнять наиболее приоритетную доступную работу. Ну, это в идеале, по крайней мере. Есть несколько случаев, когда затраты на достижение идеала не стоят совершенства.

Реализация полной вытесняющей многозадачности в ядре очень проста: когда вы планируете выполнение потока, помещая его в очередь выполнения, вы проверяете, является ли его приоритет выше, чем у текущего выполняемого потока. Если да, вы инициируете переключение контекста на этот поток.

Хотя блокировки могут защитить большинство данных в случае вытеснения, не все части ядра безопасны для вытеснения. Например, если поток, удерживающий спин-блокировку, будет вытеснен, а новый поток попытается захватить ту же спин-блокировку, новый поток может вращаться вечно, так как прерванный поток может никогда не получить шанс на выполнение. Кроме того, некоторый код, такой как код для назначения номера адресного пространства процессу во время `exec` на Alpha, не должен быть вытеснен, так как он поддерживает фактический код переключения контекста. Для таких участков кода

вытеснение отключается с использованием критической секции.

**Критические Секции**

Ответственность API критической секции заключается в предотвращении переключения контекста внутри критической секции. В полностью вытесняющем ядре каждый вызов `setrunqueue` для потока, отличного от текущего, является точкой вытеснения. Одна из реализаций заключается в том, что `critical_enter` устанавливает флаг для каждого потока, который сбрасывается его парной функцией. Если `setrunqueue` вызывается, когда этот флаг установлен, вытеснение не происходит, независимо от приоритета нового потока относительно текущего. Однако, поскольку критические секции используются в спин-блокировках для предотвращения переключения контекста и может быть захвачено несколько спин-блокировок, API критической секции должен поддерживать вложенность. По этой причине текущая реализация использует счетчик вложенности вместо одиночного флага для каждого потока.

Для минимизации задержек прерывания внутри критической секции откладываются, а не отбрасываются. Если поток, который в обычных условиях должен быть вытеснен, становится готовым к выполнению, пока текущий поток находится в критической секции, то устанавливается флаг для данного потока, указывающий на ожидающее прерывание. При выходе из самой внешней критической секции флаг проверяется. Если флаг установлен, текущий поток вытесняется, чтобы позволить выполниться потоку с более высоким приоритетом.

Прерывания создают проблему для спин-мьютексов. Если обработчик низкоуровневого прерывания требует блокировки, он не должен прерывать любой код, которому нужна эта блокировка, чтобы избежать возможного повреждения структур данных. В настоящее время этот механизм реализован через API критических секций с помощью функций `cpu_critical_enter` и `cpu_critical_exit`. Сейчас этот API отключает и снова включает прерывания на всех текущих платформах FreeBSD. Такой подход может быть не идеально оптимальным, но он прост для понимания и надежен в реализации. Теоретически, этот второй API нужен только для спин-мьютексов, используемых в основном контексте прерываний. Однако, для упрощения кода, он используется для всех спин-мьютексов и даже для всех критических секций. Возможно, стоит отделить MD API от MI API и использовать его только совместно с MI API в реализации спин-мьютексов. Если будет принят такой подход, то MD API, вероятно, потребуется переименовать, чтобы показать, что это отдельный API.

**Компромиссы проектирования**

Как упоминалось ранее, были сделаны некоторые компромиссы, чтобы пожертвовать случаями, когда идеальная вытесняющая многозадачность не всегда обеспечивает наилучшую производительность.

Первый компромисс заключается в том, что код вытеснения не учитывает другие процессоры. Предположим, у нас есть два процессора A и B, где приоритет потока A равен 4, а приоритет потока B равен 2. Если процессор B делает поток с приоритетом 1 готовым к выполнению, то теоретически мы хотим, чтобы процессор A переключился на новый поток, чтобы выполнялись два потока с наивысшим приоритетом. Однако стоимость определения,

на какой процессор нужно применить вытеснение, а также фактическая сигнализация этому процессору через IPI вместе с необходимой синхронизацией были бы огромными. Таким образом, текущий код вместо этого заставит процессор B переключиться на поток с более высоким приоритетом. Заметим, что это всё равно улучшает состояние системы, так как процессор B выполняет поток с приоритетом 1, а не поток с приоритетом 2.

Второй компромисс ограничивает немедленное вытеснение ядра только потоками ядра с реальным временем. В простом случае вытеснения, описанном выше, поток всегда вытесняется немедленно (или как только будет покинута критическая секция), если становится доступным поток с более высоким приоритетом. Однако многие потоки, выполняющиеся в ядре, работают в контексте ядра лишь короткое время перед тем, как либо заблокироваться, либо вернуться в пользовательское пространство. Таким образом, если ядро вытеснит эти потоки для выполнения другого потока ядра без реального времени, оно может переключиться с выполняемого потока как раз перед тем, как тот собирается завершиться или перейти в режим ожидания. Кэш процессора должен затем адаптироваться к новому потоку. Когда ядро возвращается к вытесненному потоку, оно должно восстановить все потерянные кэшированные данные. Кроме того, выполняются два дополнительных переключения контекста, которых можно было бы избежать, если бы ядро отложило вытеснение до момента, пока первый поток не заблокируется или не вернётся в пользовательское пространство. Таким образом, по умолчанию код вытеснения будет немедленно вытеснять поток только в том случае, если поток с более высоким приоритетом имеет приоритет реального времени.

Включение полной вытесняющей многозадачности для всех потоков ядра полезно в качестве средства отладки, так как позволяет выявить больше состояний гонки. Это особенно полезно на однопроцессорных системах (UP), где многие гонки сложно воспроизвести другими способами. Таким образом, существует опция ядра `FULL_PREEMPTION` для включения вытеснения для всех потоков ядра, которая может использоваться для целей отладки.

### 8.3.3. Миграция потоков

Простыми словами, поток мигрирует, когда переходит с одного CPU на другой. В неперемещаемом ядре это может происходить только в определённых точках, например, при вызове `msleep` или возврате в пользовательское пространство. Однако в перемещаемом ядре прерывание может вызвать вытеснение и возможную миграцию в любой момент. Это может негативно сказаться на данных, специфичных для CPU, поскольку, за исключением `curthread` и `curpcb`, данные могут изменяться при любой миграции. Поскольку потенциально миграция может произойти в любой момент, это делает незащищённый доступ к данным, специфичным для CPU, практически бесполезным. Поэтому желательно иметь возможность отключать миграцию для участков кода, где требуется стабильность данных, специфичных для CPU.

Критические секции в настоящее время предотвращают миграцию, поскольку они не допускают переключения контекстов. Однако это может быть слишком строгим требованием в некоторых случаях, так как критическая секция также эффективно блокирует потоки прерываний на текущем процессоре. В результате был предоставлен другой API, позволяющий текущему потоку указать, что если он будет вытеснен, он не

должен мигрировать на другой CPU.

Этот API известен как закрепление потока и предоставляется планировщиком. API состоит из двух функций: `sched_pin` и `sched_unpin`. Эти функции управляют счетчиком вложенности `td_pinned` для каждого потока. Поток считается закрепленным, когда его счетчик вложенности больше нуля, и прекращает быть закрепленным с нулевым счетчиком вложенности. Каждая реализация планировщика должна гарантировать, что закрепленные потоки выполняются только на том CPU, на котором они выполнялись при первом вызове `sched_pin`. Поскольку счетчик вложенности изменяется только самим потоком и читается другими потоками только тогда, когда закрепленный поток не выполняется, но удерживается `sched_lock`, то `td_pinned` не требует блокировки. Функция `sched_pin` увеличивает счетчик вложенности, а `sched_unpin` уменьшает его. Обратите внимание, что эти функции работают только с текущим потоком и привязывают текущий поток к CPU, на котором он выполняется в данный момент. Для привязки произвольного потока к определённому CPU следует использовать функции `sched_bind` и `sched_unbind`.

### 8.3.4. Обратные вызовы

Функция ядра `timeout` позволяет службам ядра регистрировать функции для выполнения в рамках программного прерывания `softclock`. События планируются на основе заданного количества тактов часов, и вызовы предоставленной потребителем функции будут происходить приблизительно в нужное время.

Глобальный список ожидающих событий с таймаутом защищен глобальной спин-блокировкой `callout_lock`; любой доступ к списку таймаутов должен выполняться с удержанием этой блокировки. Когда `softclock` пробуждается, он сканирует список ожидающих таймаутов на предмет тех, которые должны сработать. Чтобы избежать инверсии блокировок, поток `softclock` освобождает блокировку `callout_lock` при вызове предоставленной функции обратного вызова `timeout`. Если флаг `CALLOUT_MPSAFE` не был установлен во время регистрации, то `Giant` будет захвачен перед вызовом обратного вызова, а затем освобожден после него. Блокировка `callout_lock` будет повторно захвачена перед продолжением работы. Код `softclock` аккуратно поддерживает список в согласованном состоянии во время освобождения блокировки. Если включен `DIAGNOSTIC`, то измеряется время выполнения каждой функции, и если оно превышает пороговое значение, генерируется предупреждение.

## 8.4. Конкретные стратегии блокировки

### 8.4.1. Учётные данные

`struct ucred` — это внутренняя структура учётных данных ядра, которая обычно используется в качестве основы для управления доступом на уровне процессов внутри ядра. Системы, производные от BSD, используют модель «копирования при записи» для учётных данных: могут существовать множественные ссылки на структуру учётных данных, и когда требуется внести изменение, структура дублируется, изменяется, а затем ссылка заменяется. Благодаря широко распространённому кэшированию учётных данных для реализации контроля доступа при открытии, это приводит к значительной экономии памяти. С переходом на детализированную SMP (симметричную многопроцессорность), эта

модель также существенно экономит на операциях блокировки, требуя, чтобы модификации выполнялись только для неразделяемых учётных данных, избегая необходимости явной синхронизации при использовании известных разделяемых учётных данных.

Структуры учётных данных с единственной ссылкой считаются изменяемыми; разделяемые структуры учётных данных не должны изменяться, иначе возникает риск состояния гонки. Мьютекс `cr_mtxp` защищает счетчик ссылок структуры `struct ucred` для поддержания согласованности. Любое использование структуры требует действительной ссылки на протяжении всего времени использования, иначе структура может быть освобождена из-под нелегитимного потребителя.

Мьютекс `struct ucred` является листовым мьютексом и реализован через пул мьютексов по соображениям производительности.

Обычно учётные данные используются в режиме только для чтения для принятия решений по контролю доступа, и в этом случае `td_ucred`, как правило, предпочтительнее, поскольку не требует блокировки. Когда учётные данные процесса обновляются, блокировка `proc` должна удерживаться на протяжении операций проверки и обновления, чтобы избежать состояний гонки. Учётные данные процесса `p_ucred` должны использоваться для операций проверки и обновления, чтобы предотвратить гонки между временем проверки и временем использования.

Если при системных вызовах будет выполняться контроль доступа после обновления учётных данных процесса, значение `td_ucred` также должно быть обновлено до текущего значения процесса. Это предотвратит использование устаревших учётных данных после изменения. Ядро автоматически обновляет указатель `td_ucred` в структуре потока из `p_ucred` процесса всякий раз, когда процесс входит в ядро, что позволяет использовать свежие учётные данные для контроля доступа в ядре.

### 8.4.2. Дескрипторы файлов и таблицы дескрипторов файлов

Подробности будут позже.

### 8.4.3. Структуры клеток

`struct prison` хранит административные данные, связанные с обслуживанием клеток, созданных с использованием API jail(2). Это включает имя хоста для каждой клетки, IP-адрес и связанные настройки. Эта структура имеет счетчик ссылок, так как указатели на её экземпляры разделяются многими структурами учётных данных. Один мьютекс, `pr_mtx`, защищает чтение и запись счётчика ссылок и всех изменяемых переменных внутри `struct jail`. Некоторые переменные устанавливаются только при создании клетки, и действительной ссылки на `struct prison` достаточно для чтения этих значений. Точная блокировка каждой записи документирована в комментариях файла sys/jail.h.

### 8.4.4. MAC Framework

Фреймворк TrustedBSD MAC поддерживает данные в различных объектах ядра в виде `struct label`. Как правило, метки в объектах ядра защищаются тем же механизмом блокировки,

что и остальная часть объекта ядра. Например, метка `v_label` в `struct vnode` защищается блокировкой vnode.

В дополнение к меткам, поддерживаемым в стандартных объектах ядра, MAC Framework также поддерживает список зарегистрированных и активных политик. Список политик защищен глобальной мьютекс-блокировкой (`mac_policy_list_lock`) и счетчиком использования (также защищённым мьютексом). Поскольку множество проверок контроля доступа может выполняться параллельно, вход в framework для доступа только на чтение к списку политик требует удержания мьютекса во время увеличения (и последующего уменьшения) счетчика использования. Мьютекс не обязательно удерживать на протяжении всей операции входа в MAC — некоторые операции, такие как операции с метками на объектах файловой системы, выполняются длительное время. Для изменения списка политик, например во время регистрации и отмены регистрации политик, мьютекс должен быть удержан, а счетчик ссылок должен быть равен нулю, чтобы предотвратить изменение списка во время его использования.

Условная переменная `mac_policy_list_not_busy` доступна для потоков, которым необходимо дождаться освобождения списка, но ожидание на этой условной переменной допустимо только если вызывающий поток не удерживает других блокировок, иначе может возникнуть нарушение порядка блокировок. Фактически, счетчик занятости действует как форма разделяемой/исключающей блокировки доступа к фреймворку: отличие в том, что, в отличие от sx-блокировки, потребители, ожидающие освобождения списка, могут подвергаться голоданию, вместо того чтобы допускать проблемы порядка блокировок в отношении счетчика занятости и других блокировок, которые могут удерживаться при входе в (или внутри) MAC Framework.

### 8.4.5. Модули

Для подсистемы модулей существует единая блокировка, которая используется для защиты общих данных. Эта блокировка является shared/exclusive (SX) и с высокой вероятностью потребует захвата (разделяемого или исключительного), поэтому были добавлены несколько макросов для упрощения работы с ней. Эти макросы можно найти в sys/module.h, и их использование довольно простое. Основные структуры, защищаемые этой блокировкой, — это структуры `module_t` (при разделяемом доступе) и глобальная структура `modulelist_t` modules. Для более глубокого понимания стратегии блокировок рекомендуется изучить соответствующий исходный код в kern/kern_module.c.

### 8.4.6. Дерево устройств Newbus

Система newbus будет использовать одну блокировку sx. Читатели будут удерживать разделяемую (read) блокировку (sx_slock(9)), а писатели — эксклюзивную (write) блокировку (sx_xlock(9)). Внутренние функции не будут выполнять блокировку вообще. Внешне видимые функции будут блокироваться по мере необходимости. Элементы, для которых не важно, выиграна гонка или проиграна, не будут блокироваться, так как они обычно читаются во многих местах (например, device_get_softc(9)). Изменения в структурах данных newbus будут относительно редкими, поэтому одной блокировки должно быть достаточно, и это не приведёт к снижению производительности.

### 8.4.7. Каналы (pipe)

...

### 8.4.8. Процессы и потоки

- иерархия процессов
- блокировки и ссылки proc
- потокоспецифичные копии записей proc для заморозки во время системных вызовов, включая td_ucred
- межпроцессные операции
- группы процессов и сеансы

### 8.4.9. Планировщик

Множество ссылок на `sched_lock` и примечания, указывающие на конкретные примитивы и связанные с ними особенности в других частях документа.

### 8.4.10. Select и Poll

Функции `select` и `poll` позволяют потокам блокироваться в ожидании событий на файловых дескрипторах — чаще всего, доступности файловых дескрипторов для чтения или записи.

...

### 8.4.11. SIGIO

Служба SIGIO позволяет процессам запрашивать доставку сигнала SIGIO своей группе процессов при изменении статуса чтения/записи указанных файловых дескрипторов. Не более одного процесса или группы процессов может зарегистрироваться для получения SIGIO от любого заданного объекта ядра, и такой процесс или группа называется владельцем. Каждый объект, поддерживающий регистрацию SIGIO, содержит поле-указатель, которое имеет значение `NULL`, если объект не зарегистрирован, или указывает на структуру `struct sigio`, описывающую регистрацию. Это поле защищено глобальным мьютексом `sigio_lock`. Вызывающие функции обслуживания SIGIO должны передавать это поле «по ссылке», чтобы локальные копии регистра не создавались без защиты блокировкой.

Один `struct sigio` выделяется для каждого зарегистрированного объекта, связанного с любым процессом или группой процессов, и содержит обратные ссылки на объект, владельца, информацию о сигнале, учётные данные и общее состояние регистрации. Каждый процесс или группа процессов содержит список зарегистрированных структур `struct sigio`: `p_sigiolst` для процессов и `pg_sigiolst` для групп процессов. Эти списки защищены блокировками процесса или группы процессов соответственно. Большинство полей в каждой `struct sigio` остаются постоянными на протяжении регистрации, за исключением поля `sio_pgsigio`, которое связывает `struct sigio` со списком процесса или группы процессов. Разработчикам, реализующим новые объекты ядра с поддержкой SIGIO,

как правило, следует избегать удержания блокировок структур при вызове функций поддержки SIGIO, таких как `fsetown` или `funsetown`, чтобы не определять порядок блокировок между блокировками структур и глобальной блокировкой SIGIO. Обычно это возможно за счет использования повышенного счетчика ссылок на структуру, например, путем опоры на ссылку файлового дескриптора на канал во время операции с каналом.

### 8.4.12. Sysctl

Сервис `sysctl` MIB вызывается как из ядра, так и из пользовательских приложений с использованием системного вызова. По крайней мере, два вопроса возникают в отношении блокировок: во-первых, защита структур, поддерживающих пространство имен, и во-вторых, взаимодействие с переменными и функциями ядра, к которым обращается интерфейс `sysctl`. Поскольку `sysctl` позволяет прямое экспортирование (и изменение) статистики ядра и параметров конфигурации, механизм `sysctl` должен учитывать соответствующие семантики блокировок для этих переменных. В настоящее время `sysctl` использует единую глобальную sx-блокировку для сериализации использования `sysctl`; однако предполагается, что он работает под защитой Giant, и другие защиты не предоставляются. Оставшаяся часть этого раздела рассматривает возможные изменения в блокировках и семантике `sysctl`.

- Необходимо изменить порядок операций для sysctl, которые обновляют значения из чтения старого, copyin и copyout, записи нового на copyin, блокировку, чтение старого и запись нового, разблокировку, copyout. Обычные sysctl, которые просто копируют старое значение и устанавливают новое, которое они копируют, могут по-прежнему следовать старой модели. Однако, возможно, будет чище использовать вторую модель для всех обработчиков sysctl, чтобы избежать операций блокировки.
- Для упрощения распространённого случая, sysctl может включать указатель на мьютекс в макросах SYSCTL_FOO и в структуре. Это будет работать для большинства sysctl. Для значений, защищённых sx-блокировками, спин-мьютексами или другими стратегиями синхронизации, отличными от одиночного мьютекса сна, можно использовать узлы SYSCTL_PROC для обеспечения корректной блокировки.

### 8.4.13. Очередь задач

Интерфейс `taskqueue` имеет две основные блокировки, связанные с ним, для защиты соответствующих общих данных. Мьютекс `taskqueue_queues_mutex` предназначен для защиты TAILQ `taskqueue_queues`. Другая блокировка мьютекса, связанная с этой системой, находится в структуре данных `struct taskqueue`. Использование примитива синхронизации здесь необходимо для защиты целостности данных в `struct taskqueue`. Следует отметить, что нет отдельных макросов, помогающих пользователю заблокировать свою собственную работу, поскольку эти блокировки, скорее всего, не будут использоваться за пределами kern/subr_taskqueue.c.

## 8.5. Заметки о реализации

### 8.5.1. Очереди сна

Очередь сна — это структура, которая содержит список потоков, ожидающих на канале ожидания. Каждый поток, который не находится в состоянии ожидания на канале ожидания, хранит структуру очереди сна при себе. Когда поток блокируется на канале ожидания, он передаёт свою структуру очереди сна этому каналу. Очереди сна, связанные с каналом ожидания, хранятся в хеш-таблице.

Хеш-таблица очередей сна содержит очереди сна для каналов ожидания, у которых есть хотя бы один заблокированный поток. Каждая запись в хеш-таблице называется цепочкой очереди сна. Цепочка содержит связанный список очередей сна и спин-мьютекс. Спин-мьютекс защищает список очередей сна, а также содержимое структур очередей сна в списке. С каждым каналом ожидания связана только одна очередь сна. Если несколько потоков блокируются на одном канале ожидания, то очереди сна, связанные со всеми потоками, кроме первого, хранятся в списке свободных очередей сна в главной очереди сна. Когда поток удаляется из очереди сна, он получает одну из структур очереди сна из свободного списка главной очереди, если он не является единственным потоком в очереди. Последний поток получает главную очередь сна при возобновлении. Поскольку потоки могут удаляться из очереди сна в порядке, отличном от порядка добавления, поток может покинуть очередь сна с другой структурой очереди сна, чем та, с которой он в неё попал.

Функция `sleepq_lock` блокирует спин-мьютекс цепи очереди сна, соответствующей определённому каналу ожидания. Функция `sleepq_lookup` выполняет поиск в хеш-таблице главной очереди сна, связанной с заданным каналом ожидания. Если главная очередь сна не найдена, функция возвращает `NULL`. Функция `sleepq_release` разблокирует спин-мьютекс, связанный с заданным каналом ожидания.

Поток добавляется в очередь ожидания с помощью `sleepq_add`. Эта функция принимает канал ожидания, указатель на мьютекс, защищающий канал ожидания, строку описания сообщения ожидания и маску флагов. Цепь очереди ожидания должна быть заблокирована с помощью `sleepq_lock` перед вызовом этой функции. Если канал ожидания не защищен мьютексом (или защищен мьютексом Giant), то аргумент указателя на мьютекс должен быть `NULL`. Аргумент флагов содержит поле типа, указывающее на вид очереди ожидания, в которую добавляется поток, и флаг, указывающий, является ли ожидание прерываемым (`SLEEPQ_INTERRUPTIBLE`). В настоящее время существует только два типа очередей ожидания: традиционные очереди, управляемые через функции `msleep` и `wakeup` (`SLEEPQ_MSLEEP`), и очереди ожидания условных переменных (`SLEEPQ_CONDVAR`). Тип очереди ожидания и аргумент указателя на блокировку используются исключительно для внутренних проверок утверждений. Код, вызывающий `sleepq_add`, должен явно разблокировать любой блокировочный механизм, защищающий канал ожидания, после того как связанная цепь очереди ожидания будет заблокирована через `sleepq_lock` и перед блокировкой в очереди ожидания с помощью одной из функций ожидания.

Таймаут для сна устанавливается вызовом `sleepq_set_timeout`. Функция принимает канал ожидания и время таймаута в виде относительного количества тиков в качестве аргументов. Если сон должен быть прерван поступающими сигналами, следует также вызвать функцию `sleepq_catch_signals`. Эта функция принимает канал ожидания в качестве единственного параметра. Если для данного потока уже есть ожидающий сигнал, то `sleepq_catch_signals` вернёт номер сигнала; в противном случае она вернёт 0.

После добавления потока в очередь ожидания он блокируется с использованием одной из функций sleepq_wait. Существует четыре функции ожидания в зависимости от того, хочет ли вызывающий код использовать таймаут, прерывание сна перехваченными сигналами или прерывание от планировщика потоков пользовательского пространства. Функция sleepq_wait просто ожидает, пока текущий поток не будет явно возобновлён одной из функций пробуждения. Функция sleepq_timedwait ожидает, пока поток не будет явно возобновлён или пока не истечёт таймаут, установленный предыдущим вызовом sleepq_set_timeout. Функция sleepq_wait_sig ожидает, пока поток не будет явно возобновлён или его сон не будет прерван. Функция sleepq_timedwait_sig ожидает, пока поток не будет явно возобновлён, не истечёт таймаут, установленный предыдущим вызовом sleepq_set_timeout, или сон потока не будет прерван. Все функции ожидания принимают канал ожидания в качестве первого параметра. Кроме того, функция sleepq_timedwait_sig принимает второй логический параметр, указывающий, обнаружил ли предыдущий вызов sleepq_catch_signals ожидающий сигнал.

Если поток явно возобновлен или прерван сигналом, функция ожидания возвращает ноль, указывая на успешное завершение сна. Если поток возобновлен по таймауту или прерыванию от планировщика потоков в пользовательском пространстве, вместо этого возвращается соответствующее значение errno. Обратите внимание, что поскольку sleepq_wait может возвращать только 0, она ничего не возвращает, и вызывающая сторона должна считать сон успешным. Также, если сон потока прерывается одновременно по таймауту и другим причинам, sleepq_timedwait_sig вернет ошибку, указывающую на срабатывание таймаута. Если возвращено значение ошибки 0 и для блокировки использовались sleepq_wait_sig или sleepq_timedwait_sig, следует вызвать функцию sleepq_calc_signal_retval для проверки ожидающих сигналов и вычисления соответствующего возвращаемого значения, если таковые обнаружены. Номер сигнала, полученный при предыдущем вызове sleepq_catch_signals, должен быть передан в качестве единственного аргумента в sleepq_calc_signal_retval.

Потоки, находящиеся в состоянии ожидания на канале ожидания, явно возобновляются функциями sleepq_broadcast и sleepq_signal. Обе функции принимают канал ожидания, с которого нужно возобновить потоки, приоритет, на который нужно поднять возобновлённые потоки, и аргумент флагов, указывающий тип очереди ожидания, которую нужно возобновить. Аргумент приоритета трактуется как минимальный приоритет. Если у возобновляемого потока уже есть более высокий приоритет (численно меньший), чем указанный в аргументе, его приоритет не изменяется. Аргумент флагов используется для внутренних проверок, чтобы гарантировать, что очереди ожидания не обрабатываются как неправильный тип. Например, функции условных переменных не должны возобновлять потоки на традиционной очереди ожидания. Функция sleepq_broadcast возобновляет все потоки, заблокированные на указанном канале ожидания, тогда как sleepq_signal возобновляет только поток с наивысшим приоритетом, заблокированный на канале ожидания. Перед вызовом этих функций цепочка очереди ожидания должна быть заблокирована с помощью функции sleepq_lock.

Спящий поток может быть прерван с помощью вызова функции sleepq_abort. Эта функция должна вызываться с удержанием sched_lock, а поток должен находиться в очереди сна. Поток также может быть удалён из определённой очереди сна с помощью функции sleepq_remove. Эта функция принимает как поток, так и канал ожидания в качестве

аргументов и пробуждает поток только в том случае, если он находится в очереди сна для указанного канала ожидания. Если поток не находится в очереди сна или находится в очереди сна для другого канала ожидания, эта функция ничего не делает.

### 8.5.2. Турникеты

- Сравнение/сопоставление с очередями сна.
- Поиск/ожидание/освобождение. - Описать состояние гонки TDF_TSNOBLOCK.
- Приоритетное распространение.

### 8.5.3. Подробности реализации мьютекса

- Должны ли мы требовать владения мьютексами для `mtx_destroy()`, так как иначе мы не можем безопасно утверждать, что они не принадлежат кому-либо ещё?

#### Вращающиеся мьютексы

- Использование критической секции…

#### Мьютексы сна (Sleep Mutexes)

- Опишите гонки с оспариваемыми мьютексами
- Почему безопасно читать mtx_lock оспариваемой мьютекса при удержании блокировки цепочки турникета.

### 8.5.4. Witness

- Что это делает
- Как это работает

## 8.6. Разные темы

### 8.6.1. Источники прерываний и абстракции ICU

- struct isrc
- pic драйверы

### 8.6.2. Другие случайные вопросы/темы

- Передавать ли блокировку в `sema_wait`?
- Должны ли мы иметь sx блокировки без возможности сна?
- Добавить некоторую информацию о правильном использовании счетчиков ссылок.

# Глоссарий

**atomic (атомарный)**

Операция является атомарной, если все её эффекты видны другим процессорам одновременно при соблюдении соответствующего протокола доступа. В простейшем случае атомарные инструкции предоставляются непосредственно архитектурой процессора. На более высоком уровне, если несколько элементов структуры защищены блокировкой, то набор операций является атомарным, если все они выполняются при удержании блокировки без её освобождения между любыми из операций.

См. также operation (операция).

**block (блокировать)**

Поток блокируется, когда он ожидает блокировку, ресурс или условие. К сожалению, этот термин в результате немного перегружен.

См. также sleep (спать).

**critical section (критическая секция)**

Фрагмент кода, который не может быть вытеснен. Критическая секция начинается и завершается с использованием API critical_enter(9).

**MD**

Машинозависимый (machine dependent).

Смотри также MI.

**memory operation (операция с памятью)**

Операция с памятью выполняет чтение и/или запись в ячейку памяти.

**MI**

Машинонезависимый (machine independent).

См. также MD.

**operation (операция)**

См. memory operation (операция с памятью).

**primary interrupt context (основной контекст прерывания)**

Основной контекст прерывания относится к коду, который выполняется при возникновении прерывания. Этот код может либо напрямую запускать обработчик прерывания, либо планировать выполнение асинхронного потока прерывания для обработчиков прерываний данного источника.

**realtime kernel thread (поток ядра реального времени)**

Высокоприоритетный поток ядра. В настоящее время единственными потоками ядра с реальным приоритетом являются потоки обработки прерываний.

См. также thread (поток).

**sleep (спать)**

Поток находится в состоянии сна, когда он заблокирован на условной переменной или в очереди сна через `msleep` или `tsleep`.

См. также block (блокировать).

**sleepable lock (блокировка с возможностью сна)**

блокировка с возможностью сна — это блокировка, которая может удерживаться потоком, находящимся в состоянии сна. В настоящее время в FreeBSD единственными блокировками с возможностью сна являются lockmgr и sx. В будущем некоторые sx-блокировки, такие как allproc и proctree, могут стать блокировками с возможностью сна.

См. также sleep (спать).

**thread (поток)**

Поток ядра, представленный структурой `struct thread`. Потоки владеют блокировками и содержат единственный на поток контекст выполнения.

**wait channel (канал ожидания)**

Виртуальный адрес ядра, на котором потоки могут переходить в режим ожидания.

# Часть II: Драйверы устройств

# Глава 9. Написание драйверов устройств для FreeBSD

## 9.1. Введение

В этой главе представлено краткое введение в написание драйверов устройств для FreeBSD. Устройство в данном контексте — это термин, используемый в основном для аппаратных компонентов системы, таких как диски, принтеры или графический дисплей с клавиатурой. Драйвер устройства — это программный компонент операционной системы, который управляет конкретным устройством. Также существуют так называемые псевдоустройства, где драйвер эмулирует поведение устройства программно, без использования какого-либо конкретного аппаратного обеспечения. Драйверы устройств могут быть статически скомпилированы в систему или загружены по требованию через механизм динамической загрузки модулей ядра `kld`.

Большинство устройств в операционной системе, подобной UNIX®, доступны через специальные файлы устройств, также называемые узлами устройств. Эти файлы обычно расположены в каталоге /dev в иерархии файловой системы.

Драйверы устройств можно условно разделить на две категории: символьные драйверы и драйверы сетевых устройств.

## 9.2. Динамический загрузчик модулей ядра - KLD

Интерфейс kld позволяет системным администраторам динамически добавлять и удалять функциональность в работающей системе. Это позволяет разработчикам драйверов устройств загружать свои новые изменения в работающее ядро без постоянной перезагрузки для проверки изменений.

Интерфейс kld используется с помощью следующих команд:

- `kldload` - загружает новый модуль ядра
- `kldunload` — выгружает модуль ядра
- `kldstat` — выводит список загруженных модулей

Каркасная структура модуля ядра

```
/*
 * KLD Skeleton
 * Inspired by Andrew Reiter's Daemonnews article
 */

#include <sys/types.h>
#include <sys/systm.h>  /* uprintf */
#include <sys/errno.h>
#include <sys/param.h>  /* defines used in kernel.h */
```

```
#include <sys/module.h>
#include <sys/kernel.h> /* types used in module initialization */

/*
 * Load handler that deals with the loading and unloading of a KLD.
 */

static int
skel_loader(struct module *m, int what, void *arg)
{
    int err = 0;

    switch (what) {
    case MOD_LOAD:                /* kldload */
        uprintf("Skeleton KLD loaded.\n");
        break;
    case MOD_UNLOAD:
        uprintf("Skeleton KLD unloaded.\n");
        break;
    default:
        err = EOPNOTSUPP;
        break;
    }
    return(err);
}

/* Declare this module to the rest of the kernel */

static moduledata_t skel_mod = {
    "skel",
    skel_loader,
    NULL
};

DECLARE_MODULE(skeleton, skel_mod, SI_SUB_KLD, SI_ORDER_ANY);
```

### 9.2.1. Makefile

FreeBSD предоставляет системный makefile для упрощения компиляции модуля ядра.

```
SRCS=skeleton.c
KMOD=skeleton

.include <bsd.kmod.mk>
```

Запуск `make` с этим makefile создаст файл skeleton.ko, который можно загрузить в ядро, набрав:

```
# kldload -v ./skeleton.ko
```

## 9.3. Символьные устройства

Драйвер символьного устройства — это драйвер, который передаёт данные напрямую между устройством и пользовательским процессом. Это наиболее распространённый тип драйвера устройств, и в дереве исходного кода есть множество простых примеров.

Этот простой пример псевдоустройства запоминает все значения, записанные в него, и может затем воспроизводить их при чтении.

*Пример 4. Пример образца драйвера псевдоустройства Echo для FreeBSD 10.X - 12.X*

```
/*
 * Simple Echo pseudo-device KLD
 *
 * Murray Stokely
 * Søren (Xride) Straarup
 * Eitan Adler
 */

#include <sys/types.h>
#include <sys/systm.h>  /* uprintf */
#include <sys/param.h>  /* defines used in kernel.h */
#include <sys/module.h>
#include <sys/kernel.h> /* types used in module initialization */
#include <sys/conf.h>   /* cdevsw struct */
#include <sys/uio.h>    /* uio struct */
#include <sys/malloc.h>

#define BUFFERSIZE 255

/* Function prototypes */
static d_open_t      echo_open;
static d_close_t     echo_close;
static d_read_t      echo_read;
static d_write_t     echo_write;

/* Character device entry points */
static struct cdevsw echo_cdevsw = {
    .d_version = D_VERSION,
    .d_open = echo_open,
    .d_close = echo_close,
    .d_read = echo_read,
    .d_write = echo_write,
    .d_name = "echo",
};
```

```
struct s_echo {
    char msg[BUFFERSIZE + 1];
    int len;
};

/* vars */
static struct cdev *echo_dev;
static struct s_echo *echomsg;

MALLOC_DECLARE(M_ECHOBUF);
MALLOC_DEFINE(M_ECHOBUF, "echobuffer", "buffer for echo module");

/*
 * This function is called by the kld[un]load(2) system calls to
 * determine what actions to take when a module is loaded or unloaded.
 */
static int
echo_loader(struct module *m __unused, int what, void *arg __unused)
{
    int error = 0;

    switch (what) {
    case MOD_LOAD:                /* kldload */
        error = make_dev_p(MAKEDEV_CHECKNAME | MAKEDEV_WAITOK,
            &echo_dev,
            &echo_cdevsw,
            0,
            UID_ROOT,
            GID_WHEEL,
            0600,
            "echo");
        if (error != 0)
            break;

        echomsg = malloc(sizeof(*echomsg), M_ECHOBUF, M_WAITOK |
            M_ZERO);
        printf("Echo device loaded.\n");
        break;
    case MOD_UNLOAD:
        destroy_dev(echo_dev);
        free(echomsg, M_ECHOBUF);
        printf("Echo device unloaded.\n");
        break;
    default:
        error = EOPNOTSUPP;
        break;
    }
    return (error);
}

static int
```

```
echo_open(struct cdev *dev __unused, int oflags __unused, int devtype __unused,
    struct thread *td __unused)
{
    int error = 0;

    uprintf("Opened device \"echo\" successfully.\n");
    return (error);
}

static int
echo_close(struct cdev *dev __unused, int fflag __unused, int devtype __unused,
    struct thread *td __unused)
{

    uprintf("Closing device \"echo\".\n");
    return (0);
}

/*
 * The read function just takes the buf that was saved via
 * echo_write() and returns it to userland for accessing.
 * uio(9)
 */
static int
echo_read(struct cdev *dev __unused, struct uio *uio, int ioflag __unused)
{
    size_t amt;
    int error;

    /*
     * How big is this read operation?  Either as big as the user wants,
     * or as big as the remaining data.  Note that the 'len' does not
     * include the trailing null character.
     */
    amt = MIN(uio->uio_resid, uio->uio_offset >= echomsg->len + 1 ? 0 :
        echomsg->len + 1 - uio->uio_offset);

    if ((error = uiomove(echomsg->msg, amt, uio)) != 0)
        uprintf("uiomove failed!\n");

    return (error);
}

/*
 * echo_write takes in a character string and saves it
 * to buf for later accessing.
 */
static int
echo_write(struct cdev *dev __unused, struct uio *uio, int ioflag __unused)
{
    size_t amt;
```

```
    int error;

    /*
     * We either write from the beginning or are appending -- do
     * not allow random access.
     */
    if (uio->uio_offset != 0 && (uio->uio_offset != echomsg->len))
        return (EINVAL);

    /* This is a new message, reset length */
    if (uio->uio_offset == 0)
        echomsg->len = 0;

    /* Copy the string in from user memory to kernel memory */
    amt = MIN(uio->uio_resid, (BUFFERSIZE - echomsg->len));

    error = uiomove(echomsg->msg + uio->uio_offset, amt, uio);

    /* Now we need to null terminate and record the length */
    echomsg->len = uio->uio_offset;
    echomsg->msg[echomsg->len] = 0;

    if (error != 0)
        uprintf("Write failed: bad address!\n");
    return (error);
}

DEV_MODULE(echo, echo_loader, NULL);
```

Загрузив этот драйвер, попробуйте:

```
# echo -n "Test Data" > /dev/echo
# cat /dev/echo
Opened device "echo" successfully.
Test Data
Closing device "echo".
```

Реальные аппаратные устройства описаны в следующей главе.

## 9.4. Блочные устройства (удалены)

Другие системы UNIX® могут поддерживать второй тип дисковых устройств, известный как блочные устройства. Блочные устройства — это дисковые устройства, для которых ядро предоставляет кэширование. Это кэширование делает блочные устройства практически непригодными или, по крайней мере, опасно ненадёжными. Кэширование изменяет порядок операций записи, лишая приложение возможности точно знать содержимое диска в любой момент времени.

Это делает невозможным предсказуемое и надёжное восстановление после сбоев для структур данных на диске (файловых систем, баз данных и т. д.). Поскольку операции записи могут быть отложены, ядро не может сообщить приложению, какая именно операция записи столкнулась с ошибкой, что усугубляет проблему согласованности.

По этой причине ни одно серьезное приложение не полагается на блочные устройства, и фактически почти все приложения, которые обращаются к дискам напрямую, прилагают значительные усилия, чтобы указать, что следует всегда использовать символьные (или "сырые") устройства. Поскольку реализация псевдонимов для каждого диска (раздела) в виде двух устройств с разной семантикой значительно усложняла соответствующий код ядра, FreeBSD отказалась от поддержки кэшируемых дисковых устройств в рамках модернизации инфраструктуры ввода-вывода для дисков.

## 9.5. Драйверы сетевых устройств

Драйверы сетевых устройств не используют узлы устройств для доступа. Их выбор основан на других решениях, принимаемых внутри ядра, и вместо вызова open() использование сетевого устройства обычно осуществляется через системный вызов socket(2).

Для получения дополнительной информации см. ifnet(9), исходный текст loopback-устройства.

# Глава 10. Драйверы устройств ISA

## 10.1. Обзор

Эта глава знакомит с вопросами, относящимся к написанию драйвера устройства ISA. Представленный здесь псевдокод довольно детализирован и напоминает реальный код, но всё же остаётся псевдокодом. Он избегает деталей, не относящихся к теме обсуждения. Реальные примеры можно найти в исходном коде настоящих драйверов. В частности, драйверы `ep` и `aha` являются хорошими источниками информации.

## 10.2. Основная информация

Типичному драйверу ISA могут потребоваться следующие включаемые файлы:

```
#include <sys/module.h>
#include <sys/bus.h>
#include <machine/bus.h>
#include <machine/resource.h>
#include <sys/rman.h>

#include <isa/isavar.h>
#include <isa/pnpvar.h>
```

Они описывают особенности, специфичные для подсистемы ISA и обобщенной шины.

Подсистема шины реализована в объектно-ориентированном стиле, её основные структуры доступны через методы, связанные с объектами.

Список методов шины, реализуемых драйвером ISA, аналогичен списку для любой другой шины. Для гипотетического драйвера с именем "xxx" они будут:

- `static void xxx_isa_identify (driver_t *, device_t);` Обычно используется для драйверов шины, а не для драйверов устройств. Однако для устройств ISA этот метод может иметь особое применение: если устройство предоставляет специфический (не PnP) способ автоматического обнаружения устройств, эта процедура может его реализовывать.
- `static int xxx_isa_probe (device_t dev);` Проверка наличия устройства в известном (или PnP) расположении. Эта процедура также может учитывать автоопределение параметров, специфичных для устройства, в случае частично настроенных устройств.
- `static int xxx_isa_attach (device_t dev);` Подключение и инициализация устройства.
- `static int xxx_isa_detach (device_t dev);` Отсоединение устройства перед выгрузкой модуля драйвера.
- `static int xxx_isa_shutdown (device_t dev);` Выполняет завершение работы устройства перед выключением системы.
- `static int xxx_isa_suspend (device_t dev);` Приостанавливает устройство перед переходом системы в энергосберегающий режим. Также может прервать переход в

энергосберегающий режим.

- `static int xxx_isa_resume (device_t dev);` Возобновляет активность устройства после возврата из энергосберегающего состояния.

`xxx_isa_probe()` и `xxx_isa_attach()` являются обязательными, остальные процедуры опциональны и зависят от потребностей устройства.

Драйвер связан с системой следующим набором описаний.

```
    /* table of supported bus methods */
    static device_method_t xxx_isa_methods[] = {
        /* list all the bus method functions supported by the driver */
        /* omit the unsupported methods */
        DEVMETHOD(device_identify,  xxx_isa_identify),
        DEVMETHOD(device_probe,     xxx_isa_probe),
        DEVMETHOD(device_attach,    xxx_isa_attach),
        DEVMETHOD(device_detach,    xxx_isa_detach),
        DEVMETHOD(device_shutdown,  xxx_isa_shutdown),
        DEVMETHOD(device_suspend,   xxx_isa_suspend),
        DEVMETHOD(device_resume,    xxx_isa_resume),

    DEVMETHOD_END
    };

    static driver_t xxx_isa_driver = {
        "xxx",
        xxx_isa_methods,
        sizeof(struct xxx_softc),
    };

    static devclass_t xxx_devclass;

    DRIVER_MODULE(xxx, isa, xxx_isa_driver, xxx_devclass,
        load_function, load_argument);
```

Здесь структура `xxx_softc` — это специфичная для устройства структура, которая содержит приватные данные драйвера и дескрипторы ресурсов драйвера. Код шины автоматически выделяет один дескриптор softc для каждого устройства по мере необходимости.

Если драйвер реализован в виде загружаемого модуля, то `load_function()` вызывается для выполнения специфичной для драйвера инициализации или очистки при загрузке или выгрузке драйвера, а `load_argument` передаётся в качестве одного из её аргументов. Если драйвер не поддерживает динамическую загрузку (другими словами, он всегда должен быть связан с ядром), то эти значения должны быть установлены в 0, и последнее определение будет выглядеть следующим образом:

```
DRIVER_MODULE(xxx, isa, xxx_isa_driver,
      xxx_devclass, 0, 0);
```

Если драйвер предназначен для устройства с поддержкой PnP, то должна быть определена таблица поддерживаемых PnP ID. Таблица состоит из списка PnP ID, поддерживаемых этим драйвером, и удобочитаемых описаний типов аппаратного обеспечения и моделей, имеющих эти ID. Это выглядит следующим образом:

```
    static struct isa_pnp_id xxx_pnp_ids[] = {
        /* a line for each supported PnP ID */
        { 0x12345678,   "Our device model 1234A" },
        { 0x12345679,   "Our device model 1234B" },
        { 0,        NULL }, /* end of table */
    };
```

Если драйвер не поддерживает устройства PnP, ему все равно нужна пустая таблица идентификаторов PnP, например:

```
    static struct isa_pnp_id xxx_pnp_ids[] = {
        { 0,        NULL }, /* end of table */
    };
```

## 10.3. Указатель `device_t`

`device_t` — это тип указателя на структуру устройства. Здесь мы рассматриваем только методы, представляющие интерес с точки зрения разработчика драйверов устройств. Методы для работы со значениями в структуре устройства следующие:

- `device_t device_get_parent(dev)` Получить родительскую шину устройства.
- `driver_t device_get_driver(dev)` Получить указатель на структуру его драйвера.
- `char *device_get_name(dev)` Получить имя драйвера, например `"xxx"` в нашем примере.
- `int device_get_unit(dev)` Получить номер устройства (устройства нумеруются с 0 для устройств, связанных с каждым драйвером).
- `char *device_get_nameunit(dev)` Получить имя устройства, включая номер юнита, например, "xxx0", "xxx1" и так далее.
- `char *device_get_desc(dev)` Получить описание устройства. Обычно оно описывает точную модель устройства в удобочитаемом виде.
- `device_set_desc(dev, desc)` Установить описание. Это заставляет описание устройства указывать на строку desc, которая не может быть освобождена или изменена после этого.
- `device_set_desc_copy(dev, desc)` Установить описание. Описание копируется во внутренний динамически выделяемый буфер, поэтому строка desc может быть изменена впоследствии без негативных последствий.
- `void *device_get_softc(dev)` Получить указатель на дескриптор устройства (структура `xxx_softc`), связанный с данным устройством.
- `u_int32_t device_get_flags(dev)` Получить флаги, указанные для устройства в файле

конфигурации.

Функция для удобства `device_printf(dev, fmt, ⋯)` может использоваться для вывода сообщений из драйвера устройства. Она автоматически добавляет имя устройства и двоеточие перед сообщением.

Методы device_t реализованы в файле kern/bus_subr.c.

## 10.4. Файл конфигурации и порядок определения и проверки при автоматической настройке

Устройства ISA описываются в файле конфигурации ядра следующим образом:

```
device xxx0 at isa? port 0x300 irq 10 drq 5
       iomem 0xd0000 flags 0x1 sensitive
```

Значения порта, IRQ и т. д. преобразуются в ресурсы, связанные с устройством. Они являются необязательными, в зависимости от потребностей устройства и его способностей к автонастройке. Например, некоторым устройствам вообще не нужен DRQ, а некоторые позволяют драйверу читать настройку IRQ из портов конфигурации устройства. Если в машине несколько шин ISA, точная шина может быть указана в строке конфигурации, например `isa0` или `isa1`, иначе устройство будет искаться на всех шинах ISA.

`sensitive` — это ресурс, указывающий, что данное устройство должно быть проверено перед всеми нечувствительными устройствами. Он поддерживается, но, похоже, не используется ни в одном текущем драйвере.

Для устаревших устройств ISA во многих случаях драйверы всё ещё могут определять параметры конфигурации. Однако каждое устройство, которое необходимо настроить в системе, должно иметь строку конфигурации. Если в системе установлено два устройства одного типа, но для соответствующего драйвера есть только одна строка конфигурации, например:

```
device xxx0 at isa?
```

```
тогда будет настроено только одно устройство.
```

Однако для устройств, поддерживающих автоматическую идентификацию с помощью Plug-n-Play или какого-либо проприетарного протокола, достаточно одной строки конфигурации для настройки всех устройств в системе, как в примере выше или просто:

```
device xxx at isa?
```

Если драйвер поддерживает как автоматически определяемые, так и устаревшие

устройства, и оба типа установлены одновременно в одной машине, то достаточно описать в конфигурационном файле только устаревшие устройства. Автоматически определяемые устройства будут добавлены автоматически.

При автоматической настройке шины ISA события происходят в следующем порядке:

Все процедуры идентификации драйверов (включая процедуру идентификации PnP, которая определяет все устройства PnP) вызываются в случайном порядке. Когда они идентифицируют устройства, они добавляют их в список на шине ISA. Обычно процедуры идентификации драйверов связывают свои драйверы с новыми устройствами. Процедура идентификации PnP пока не знает о других драйверах, поэтому не связывает ни один из них с новыми устройствами, которые она добавляет.

Устройства PnP переводятся в режим сна с использованием протокола PnP, чтобы предотвратить их обнаружение как устаревших устройств.

Вызываются процедуры обнаружения для устройств, не поддерживающих PnP, помеченных как `sensitive`. Если процедура обнаружения для устройства завершилась успешно, вызывается процедура присоединения для него.

Вызов процедур обнаружения и присоединения всех устройств, не поддерживающих PNP, выполняется аналогичным образом.

Устройства PnP выводятся из состояния сна и получают запрошенные ресурсы: диапазоны адресов ввода-вывода и памяти, IRQ и DRQ, причём все они не конфликтуют с подключёнными устаревшими устройствами.

Затем для каждого устройства PnP вызываются процедуры обнаружения всех присутствующих драйверов ISA. Первый драйвер, который заявит о поддержке устройства, будет присоединен. Возможна ситуация, когда несколько драйверов заявят о поддержке устройства с разным приоритетом; в этом случае побеждает драйвер с наивысшим приоритетом. Процедуры обнаружения должны вызывать `ISA_PNP_PROBE()` для сравнения фактического PnP ID со списком ID, поддерживаемых драйвером, и если ID отсутствует в таблице, возвращать ошибку. Это означает, что абсолютно каждый драйвер, даже те, которые не поддерживают никакие PnP устройства, должны вызывать `ISA_PNP_PROBE()`, хотя бы с пустой таблицей PnP ID, чтобы возвращать ошибку для неизвестных PnP устройств.

Процедура обнаружения возвращает положительное значение (код ошибки) в случае ошибки, ноль или отрицательное значение в случае успеха.

Отрицательные возвращаемые значения используются, когда устройство PnP поддерживает несколько интерфейсов. Например, старый совместимый интерфейс и новый расширенный интерфейс, которые поддерживаются разными драйверами. В этом случае оба драйвера обнаружат устройство. Драйвер, возвращающий большее значение в процедуре обнаружения, получает приоритет (другими словами, драйвер, возвращающий 0, имеет наивысший приоритет, возвращающий -1 — следующий, возвращающий -2 — за ним и так далее). В результате устройства, поддерживающие только старый интерфейс, будут обрабатываться старым драйвером (который должен возвращать -1 из процедуры обнаружения), тогда как устройства, поддерживающие также новый интерфейс, будут обрабатываться новым драйвером (который должен возвращать 0 из процедуры

обнаружения). Если несколько драйверов возвращают одинаковое значение, побеждает тот, который был вызван первым. Таким образом, если драйвер возвращает значение 0, он может быть уверен, что выиграл арбитраж приоритетов.

Процедуры идентификации, специфичные для устройства, также могут назначать устройству не драйвер, а класс драйверов. Затем все драйверы в этом классе проверяются на совместимость с устройством, как в случае с PnP. Эта возможность не реализована ни в одном существующем драйвере и далее в этом документе не рассматривается.

Поскольку устройства PnP отключены при проверке устаревших устройств, они не будут присоединены дважды (один раз как устаревшие и один раз как PnP). Однако в случае процедур идентификации, зависящих от устройства, ответственность за то, чтобы одно и то же устройство не было присоединено драйвером дважды (один раз как настроенное пользователем устаревшее и один раз как автоматически идентифицированное), лежит на драйвере.

Еще одно практическое следствие для автоматически определяемых устройств (как PnP, так и специфичных для устройства) заключается в том, что флаги не могут быть переданы им из файла конфигурации ядра. Поэтому они либо не должны использовать флаги вообще, либо использовать флаги из устройства unit 0 для всех автоматически определяемых устройств, либо использовать интерфейс sysctl вместо флагов.

Другие нестандартные конфигурации могут быть реализованы путем прямого доступа к ресурсам конфигурации с использованием функций семейств `resource_query_*()` и `resource_*_value()`. Их реализации находятся в kern/subr_bus.c. Примеры такого использования есть в старом драйвере диска IDE i386/isa/wd.c. Однако стандартные методы конфигурации всегда должны быть предпочтительны. Оставьте разбор ресурсов конфигурации коду настройки шины.

## 10.5. Ресурсы

Информация, которую пользователь вводит в файл конфигурации ядра, обрабатывается и передаётся ядру в виде ресурсов конфигурации. Эта информация анализируется кодом конфигурации шины и преобразуется в значение структуры `device_t` и связанные с ней ресурсы шины. Драйверы могут напрямую обращаться к ресурсам конфигурации, используя функции `resource_*` для более сложных случаев конфигурации. Однако, как правило, в этом нет необходимости, и это не рекомендуется, поэтому данный вопрос далее не рассматривается.

Ресурсы шины связаны с каждым устройством. Они идентифицируются по типу и номеру внутри типа. Для шины ISA определены следующие типы:

- *SYS_RES_IRQ* - номер прерывания
- *SYS_RES_DRQ* - номер канала ISA DMA
- *SYS_RES_MEMORY* - диапазон памяти устройства, отображенный в системное адресное пространство
- *SYS_RES_IOPORT* - диапазон регистров ввода-вывода устройства

Перечисление внутри типов начинается с 0, поэтому если устройство имеет две области памяти, оно будет иметь ресурсы типа `SYS_RES_MEMORY` с номерами 0 и 1. Тип ресурса не связан с типом языка C, все значения ресурсов имеют тип `unsigned long` в языке C и должны быть приведены по мере необходимости. Номера ресурсов не обязательно должны быть последовательными, хотя для ISA они обычно таковыми являются. Допустимые номера ресурсов для устройств ISA:

```
IRQ: 0-1
DRQ: 0-1
MEMORY: 0-3
IOPORT: 0-7
```

Все ресурсы представлены в виде диапазонов с начальным значением и количеством. Для ресурсов IRQ и DRQ количество обычно равно 1. Значения для памяти относятся к физическим адресам.

Три типа действий могут выполняться над ресурсами:

- Установка — set/get
- Выделение — allocate/release
- Активация — activate/deactivate

Установка задает диапазон, используемый ресурсом. Выделение резервирует запрошенный диапазон, чтобы никакой другой драйвер не смог его зарезервировать (и проверяет, что никакой другой драйвер уже не зарезервировал этот диапазон). Активация делает ресурс доступным для драйвера, выполняя все необходимые для этого действия (например, для памяти это может быть отображение в виртуальное адресное пространство ядра).

Функции для управления ресурсами:

- `int bus_set_resource(device_t dev, int type, int rid, u_long start, u_long count)`

  Устанавливает диапазон для ресурса. Возвращает 0 при успешном выполнении, в противном случае — код ошибки. Обычно эта функция возвращает ошибку только в том случае, если одно из значений `type`, `rid`, `start` или `count` выходит за пределы допустимого диапазона.

  - dev - устройство драйвера
  - тип - тип ресурса, SYS_RES_*
  - rid - номер ресурса (ID) в пределах типа
  - начало, количество - диапазон ресурсов
- `int bus_get_resource(device_t dev, int type, int rid, u_long *startp, u_long *countp)`

  Получает диапазон ресурса. Возвращает 0 при успехе, код ошибки, если ресурс ещё не определён.

- `u_long bus_get_resource_start(device_t dev, int type, int rid)` и `u_long`

`bus_get_resource_count (device_t dev, int type, int rid)`

Удобные функции для получения только начала или количества. Возвращают 0 в случае ошибки, поэтому если начало ресурса может законно содержать 0, невозможно определить, является ли значение 0 или произошла ошибка. К счастью, ни один ресурс ISA для дополнительных драйверов не может иметь начальное значение, равное 0.

- `void bus_delete_resource(device_t dev, int type, int rid)`

  Удаляет ресурс, делает его неопределённым.

- `struct resource * bus_alloc_resource(device_t dev, int type, int *rid, u_long start, u_long end, u_long count, u_int flags)`

  Выделяет ресурс как диапазон значений count, не выделённых никем другим, где-то между start и end. Увы, выравнивание не поддерживается. Если ресурс ещё не был установлен, он автоматически создаётся. Специальные значения start равное 0 и end равное ~0 (все единицы) означают, что должны использоваться фиксированные значения, ранее установленные `bus_set_resource()`: start и count как есть, а end=(start+count). В этом случае, если ресурс не был определён ранее, возвращается ошибка. Хотя rid передаётся по ссылке, он нигде не устанавливается кодом выделения ресурсов шины ISA. Другие шины могут использовать иной подход и изменять его.

Флаги представляют собой битовую карту. Интересные для вызывающей стороны флаги:

- *RF_ACTIVE* - приводит к автоматической активации ресурса после его выделения.
- *RF_SHAREABLE* - ресурс может использоваться одновременно несколькими драйверами.
- *RF_TIMESHARE* - ресурс может разделяться по времени несколькими драйверами, т.е. выделяться одновременно многими, но активироваться только одним в любой момент времени.
- Возвращает 0 при ошибке. Выделенные значения могут быть получены из возвращённого дескриптора с использованием методов `rhand_*()`.
- `int bus_release_resource(device_t dev, int type, int rid, struct resource *r)`
- Освобождает ресурс, r — это дескриптор, возвращённый `bus_alloc_resource()`. Возвращает 0 при успехе, код ошибки в противном случае.
- `int bus_activate_resource(device_t dev, int type, int rid, struct resource *r) int bus_deactivate_resource(device_t dev, int type, int rid, struct resource *r)`
- Активирует или деактивирует ресурс. Возвращает 0 при успехе, в противном случае — код ошибки. Если ресурс разделяемый и в данный момент активирован другим драйвером, возвращается `EBUSY`.
- `int bus_setup_intr(device_t dev, struct resource *r, int flags, driver_intr_t *handler, void *arg, void **cookiep) int bus_teardown_intr(device_t dev, struct resource *r, void *cookie)`
- Связывает или разрывает связь обработчика прерывания с устройством. Возвращает 0 при успехе, код ошибки в противном случае.
- r - активированный обработчик ресурсов, описывающий IRQ

flags - уровень приоритета прерывания, один из:

- `INTR_TYPE_TTY` - терминалы и другие аналогичные символьные устройства. Для их маскировки используйте `spltty()`.
- `(INTR_TYPE_TTY | INTR_TYPE_FAST)` - терминальные устройства с малым буфером ввода, критичные к потере данных на входе (например, устаревшие последовательные порты). Для их маскирования используйте `spltty()`.
- `INTR_TYPE_BIO` - блочные устройства, за исключением тех, что подключены к контроллерам CAM. Для их маскирования используйте `splbio()`.
- `INTR_TYPE_CAM` - контроллеры шины CAM (Common Access Method). Для их маскирования используйте `splcam()`.
- `INTR_TYPE_NET` - контроллеры сетевых интерфейсов. Для их маскирования используйте `splimp()`.
- `INTR_TYPE_MISC` — прочие устройства. Нет другого способа их маскировки, кроме `splhigh()`, который маскирует все прерывания.

Когда обработчик прерывания выполняется, все другие прерывания, соответствующие его уровню приоритета, будут заблокированы. Единственное исключение — уровень MISC, для которого никакие другие прерывания не блокируются и который сам не блокируется другими прерываниями.

- *handler* - указатель на функцию-обработчик, тип driver_intr_t определён как `void driver_intr_t(void *)`
- *arg* - аргумент, передаваемый обработчику для идентификации конкретного устройства. Приводится обработчиком от void* к фактическому типу. Старая конвенция для обработчиков прерываний ISA предполагала использование номера устройства в качестве аргумента, новая (рекомендуемая) конвенция предполагает использование указателя на структуру softc устройства.
- *cookie[p]* - значение, полученное из `setup()`, используется для идентификации обработчика при передаче в `teardown()`

Определены несколько методов для работы с обработчиками ресурсов (struct resource *). Вот те из них, которые представляют интерес для разработчиков драйверов устройств:

- `u_long rman_get_start(r)` `u_long rman_get_end(r)` Получают начало и конец выделенного диапазона ресурсов.
- `void *rman_get_virtual(r)` Получает виртуальный адрес активированного ресурса памяти.

# 10.6. Отображение памяти шины

Во многих случаях данные передаются между драйвером и устройством через память. Возможны два варианта:

(a) память расположена на карте устройства

(b) память — это основная память компьютера

В случае (a) драйвер всегда копирует данные между памятью на карте и основной памятью по мере необходимости. Для отображения памяти на карте в виртуальное адресное пространство ядра физический адрес и длина памяти на карте должны быть определены как ресурс `SYS_RES_MEMORY`. Этот ресурс может быть затем выделен и активирован, а его виртуальный адрес получен с помощью `rman_get_virtual()`. Более старые драйверы использовали для этой цели функцию `pmap_mapdev()`, которую больше не следует использовать напрямую. Теперь это один из внутренних шагов активации ресурса.

Большинство ISA-карт имеют память, настроенную на физическое расположение в диапазоне 640 КБ–1 МБ. Некоторые ISA-карты требуют большего диапазона памяти, который должен быть размещён ниже 16 МБ (из-за 24-битного ограничения адресации на шине ISA). В таком случае, если в машине больше памяти, чем начальный адрес памяти устройства (другими словами, они пересекаются), необходимо настроить "дыру" в памяти по диапазону адресов, используемому устройствами. Многие BIOS позволяют настроить "дыру" в памяти размером 1 МБ, начиная с 14 МБ или 15 МБ. FreeBSD корректно обрабатывает "дыры" в памяти, если BIOS правильно их сообщает (эта функция может не работать в старых BIOS).

В случае (b) только адрес данных отправляется на устройство, и устройство использует DMA для фактического доступа к данным в основной памяти. Существуют два ограничения: во-первых, карты ISA могут обращаться только к памяти ниже 16 МБ. Во-вторых, непрерывные страницы в виртуальном адресном пространстве могут не быть непрерывными в физическом адресном пространстве, поэтому устройству может потребоваться выполнять операции scatter/gather. Подсистема шины предоставляет готовые решения для некоторых из этих проблем, остальное должно быть реализовано самими драйверами.

Для выделения памяти DMA используются две структуры: `bus_dma_tag_t` и `bus_dmamap_t`. Тег (`tag`) описывает свойства, необходимые для памяти DMA. Карта (`map`) представляет собой блок памяти, выделенный в соответствии с этими свойствами. С одним тегом может быть связано несколько карт.

Теги организованы в иерархию в виде дерева с наследованием свойств. Дочерний тег наследует все требования родительского тега и может делать их более строгими, но никогда более мягкими.

Обычно создаётся один корневой тег (без родителя) для каждого устройства. Если для каждого устройства требуется несколько областей памяти с разными требованиями, то для каждой из них может быть создан тег как дочерний по отношению к родительскому тегу.

Теги могут быть использованы для создания карты двумя способами.

Сначала может быть выделен (а затем освобожден) блок непрерывной памяти, соответствующий требованиям тега. Обычно это используется для выделения относительно долгоживущих областей памяти для взаимодействия с устройством. Загрузка такой памяти в карту тривиальна: она всегда рассматривается как один блок в соответствующем диапазоне физической памяти.

Второй момент: произвольная область виртуальной памяти может быть загружена в карту. Каждая страница этой памяти будет проверяться на соответствие требованиям карты. Если она соответствует, то остаётся на своём исходном месте. Если нет, то выделяется новая

соответствующая промежуточная страница (bounce page), которая используется как промежуточное хранилище. При записи данных с несоответствующих исходных страниц они сначала копируются на свои промежуточные страницы, а затем передаются с промежуточных страниц на устройство. При чтении данные поступают с устройства на промежуточные страницы, а затем копируются на свои несоответствующие исходные страницы. Процесс копирования между исходными и промежуточными страницами называется синхронизацией. Обычно это используется для каждой передачи: буфер для каждой передачи загружается, передача выполняется, и буфер выгружается.

Функции, работающие с памятью DMA:

- `int bus_dma_tag_create(bus_dma_tag_t parent, bus_size_t alignment, bus_size_t boundary, bus_addr_t lowaddr, bus_addr_t highaddr, bus_dma_filter_t *filter, void *filterarg, bus_size_t maxsize, int nsegments, bus_size_t maxsegsz, int flags, bus_dma_tag_t *dmat)`

  Создать новый тег. Возвращает 0 при успехе, код ошибки в противном случае.

  - *parent* - родительский тег, или NULL для создания тега верхнего уровня.
  - *alignment* - требуемое физическое выравнивание области памяти, которая будет выделена для этого тега. Используйте значение 1 для "без специфического выравнивания". Применяется только к будущим вызовам `bus_dmamem_alloc()`, но не `bus_dmamap_create()`.
  - *boundary* - физическая граница адреса, которую нельзя пересекать при выделении памяти. Используйте значение 0 для обозначения "нет границы". Применяется только к будущим вызовам `bus_dmamem_alloc()`, но не `bus_dmamap_create()`. Должна быть степенью 2. Если память планируется использовать в некаскадном режиме DMA (т.е. адреса DMA будут предоставляться не самим устройством, а контроллером DMA ISA), то граница не должна превышать 64 КБ (64*1024) из-за ограничений аппаратного обеспечения DMA.
  - *lowaddr*, *highaddr* - названия немного вводят в заблуждение; эти значения используются для ограничения допустимого диапазона физических адресов, используемых для выделения памяти. Точное значение зависит от предполагаемого будущего использования:
    - Для `bus_dmamem_alloc()` все адреса от 0 до lowaddr-1 считаются разрешёнными, а более высокие — запрещёнными.
    - Для `bus_dmamap_create()` все адреса вне включительного диапазона [lowaddr; highaddr] считаются доступными. Адреса страниц внутри диапазона передаются в функцию-фильтр, которая определяет, доступны ли они. Если функция-фильтр не предоставлена, то весь диапазон считается недоступным.
    - Для устройств ISA обычные значения (без функции фильтрации) следующие:

      lowaddr = BUS_SPACE_MAXADDR_24BIT

      highaddr = BUS_SPACE_MAXADDR
  - *filter*, *filterarg* - функция фильтра и её аргумент. Если передаётся NULL для filter, то весь диапазон [lowaddr, highaddr] считается недоступным при выполнении

`bus_dmamap_create()`. В противном случае физический адрес каждой страницы в диапазоне [lowaddr; highaddr] передаётся в функцию фильтра, которая определяет, доступна ли она. Прототип функции фильтра: `int filterfunc(void *arg, bus_addr_t paddr)`. Функция должна вернуть 0, если страница доступна, и ненулевое значение в противном случае.

- *maxsize* - максимальный размер памяти (в байтах), который может быть выделен через этот тег. Если сложно оценить или он может быть произвольно большим, для устройств ISA следует использовать значение `BUS_SPACE_MAXSIZE_24BIT`.
- *nsegments* - максимальное количество сегментов scatter-gather, поддерживаемых устройством. Если ограничений нет, следует использовать значение `BUS_SPACE_UNRESTRICTED`. Это значение рекомендуется для родительских тегов, фактические ограничения затем будут указаны для дочерних тегов. Теги с nsegments равным `BUS_SPACE_UNRESTRICTED` не могут использоваться для фактической загрузки отображений, они могут применяться только как родительские теги. Практический предел для nsegments составляет около 250-300, более высокие значения вызовут переполнение стека ядра (аппаратное обеспечение обычно не поддерживает такое большое количество scatter-gather буферов в любом случае).
- *maxsegsz* — максимальный размер сегмента scatter-gather, поддерживаемый устройством. Максимальное значение для устройства ISA будет `BUS_SPACE_MAXSIZE_24BIT`.
- *flags* - битовая маска флагов. Единственный интересный флаг:
  - *BUS_DMA_ALLOCNOW* - запрашивает выделение всех потенциально необходимых промежуточных страниц при создании тега.
- *dmat* - указатель на хранилище для нового возвращаемого тега.

- `int bus_dma_tag_destroy(bus_dma_tag_t dmat)`

  Уничтожить тег. Возвращает 0 при успехе, код ошибки в противном случае.

  dmat - тег, который должен быть уничтожен.

- `int bus_dmamem_alloc(bus_dma_tag_t dmat, void** vaddr, int flags, bus_dmamap_t *mapp)`

  Выделить область непрерывной памяти, описанную тегом. Размер выделяемой памяти соответствует maxsize тега. Возвращает 0 при успехе, иначе код ошибки. Результат всё ещё должен быть загружен с помощью `bus_dmamap_load()` перед использованием для получения физического адреса памяти.

  - *dmat* - тег
  - *vaddr* - указатель на хранилище для возвращаемого виртуального адреса ядра выделенной области.
  - flags - битовая карта флагов. Единственный интересный флаг:
    - *BUS_DMA_NOWAIT* - если память недоступна немедленно, вернуть ошибку. Если этот флаг не установлен, то процедуре разрешено ожидать до тех пор, пока память не станет доступной.
  - *mapp* - указатель на хранилище для возвращаемой новой карты.

- `void bus_dmamem_free(bus_dma_tag_t dmat, void *vaddr, bus_dmamap_t map)`

  Освободить память, выделенную `bus_dmamem_alloc()`. В настоящее время освобождение памяти, выделенной с ограничениями ISA, не реализовано. В связи с этим рекомендуется сохранять и повторно использовать выделенные области как можно дольше. Не следует без необходимости освобождать область и вскоре снова её выделять. Это не означает, что `bus_dmamem_free()` не следует использовать вовсе: есть надежда, что вскоре она будет реализована должным образом.

  - *dmat* - тег
  - *vaddr* - виртуальный адрес памяти ядра
  - *map* - карта памяти (как возвращается из `bus_dmamem_alloc()`)
- `int bus_dmamap_create(bus_dma_tag_t dmat, int flags, bus_dmamap_t *mapp)`

  Создать карту для тега, которая будет использоваться в `bus_dmamap_load()` позже. Возвращает 0 при успехе, в противном случае — код ошибки.

  - *dmat* - тег
  - *flags* - теоретически, битовая карта флагов. Однако пока никакие флаги не определены, поэтому в настоящее время значение всегда будет 0.
  - *mapp* - указатель на хранилище для новой карты, которая будет возвращена
- `int bus_dmamap_destroy(bus_dma_tag_t dmat, bus_dmamap_t map)`

  Уничтожить карту. Возвращает 0 при успехе, в противном случае — код ошибки.

  - dmat - тег, с которым ассоциирована карта
  - map - карта, подлежащая уничтожению
- `int bus_dmamap_load(bus_dma_tag_t dmat, bus_dmamap_t map, void *buf, bus_size_t buflen, bus_dmamap_callback_t *callback, void *callback_arg, int flags)`

  Загрузить буфер в карту (карта должна быть предварительно создана с помощью `bus_dmamap_create()` или `bus_dmamem_alloc()`). Все страницы буфера проверяются на соответствие требованиям тега, и для несоответствующих выделяются промежуточные страницы. Создаётся массив дескрипторов физических сегментов и передаётся в подпрограмму обратного вызова. Ожидается, что эта подпрограмма обработает его каким-либо образом. Количество промежуточных буферов в системе ограничено, поэтому, если эти буферы требуются, но недоступны немедленно, запрос будет поставлен в очередь, и обратный вызов будет выполнен, когда промежуточные буферы станут доступны. Возвращает 0, если обратный вызов был выполнен немедленно, или `EINPROGRESS`, если запрос был поставлен в очередь для выполнения в будущем. В последнем случае синхронизация с подпрограммой обратного вызова, поставленной в очередь, является обязанностью драйвера.

  - *dmat* - тег
  - *map* - карта
  - *buf* - виртуальный адрес буфера в пространстве ядра

    - *buflen* - длина буфера
    - *callback*, `callback_arg` - функция обратного вызова и её аргумент

      Прототип функции обратного вызова: `void callback(void *arg, bus_dma_segment_t *seg, int nseg, int error)`

    - *arg* - то же самое, что и callback_arg, переданный в `bus_dmamap_load()`
    - *seg* - массив дескрипторов сегментов
    - *nseg* - количество дескрипторов в массиве
    - *error* - указание на переполнение номера сегмента: если установлено значение `EFBIG`, значит буфер не поместился в максимальное количество сегментов, разрешённых тегом. В этом случае в массиве будет только разрешённое количество дескрипторов. Обработка этой ситуации зависит от драйвера: в зависимости от желаемой семантики он может либо считать это ошибкой, либо разделить буфер на две части и обработать вторую часть отдельно

      Каждая запись в массиве segments содержит поля:

    - *ds_addr* - физический адрес шины сегмента
    - *ds_len* - длина сегмента
- `void bus_dmamap_unload(bus_dma_tag_t dmat, bus_dmamap_t map)`

  выгрузить карту.

    - *dmat* - тег
    - *map* - загруженная карта
- `void bus_dmamap_sync (bus_dma_tag_t dmat, bus_dmamap_t map, bus_dmasync_op_t op)`

  Синхронизировать загруженный буфер с его промежуточными страницами до и после физической передачи на устройство или с устройства. Это функция, которая выполняет все необходимое копирование данных между исходным буфером и его отображенной версией. Буферы должны быть синхронизированы как до, так и после выполнения передачи.

    - *dmat* - тег
    - *map* - загруженная карта
    - *op* - тип операции синхронизации для выполнения:
    - `BUS_DMASYNC_PREREAD` - перед чтением с устройства в буфер
    - `BUS_DMASYNC_POSTREAD` - после чтения из устройства в буфер
    - `BUS_DMASYNC_PREWRITE` - перед записью буфера в устройство
    - `BUS_DMASYNC_POSTWRITE` - после записи буфера в устройство

На данный момент PREREAD и POSTWRITE являются пустыми операциями, но это может измениться в будущем, поэтому их нельзя игнорировать в драйвере. Синхронизация не требуется для памяти, полученной из `bus_dmamem_alloc()`.

Перед вызовом функции обратного вызова из `bus_dmamap_load()` массив сегментов сохраняется в стеке. Он предварительно выделяется для максимального количества сегментов, разрешенного тегом. В результате этого практический предел количества сегментов на архитектуре i386 составляет около 250-300 (размер стека ядра — 4 КБ минус размер структуры пользователя, размер элемента массива сегментов — 8 байт, и необходимо оставить некоторое пространство). Поскольку массив выделяется исходя из максимального числа, это значение не должно быть установлено выше, чем действительно необходимо. К счастью, для большинства оборудования максимально поддерживаемое количество сегментов значительно ниже. Но если драйвер должен обрабатывать буферы с очень большим количеством сегментов scatter-gather, он должен делать это по частям: загрузить часть буфера, передать его устройству, загрузить следующую часть буфера и так далее.

Еще одно практическое следствие заключается в том, что количество сегментов может ограничивать размер буфера. Если все страницы в буфере окажутся физически несмежными, то максимальный поддерживаемый размер буфера для такого фрагментированного случая будет равен (nsegments * page_size). Например, если поддерживается максимальное количество сегментов, равное 10, то на i386 максимальный гарантированно поддерживаемый размер буфера составит 40K. Если требуется больший размер, то в драйвере следует использовать специальные приемы.

Если оборудование не поддерживает scatter-gather вообще или драйвер хочет поддерживать некоторый размер буфера, даже если он сильно фрагментирован, то решение состоит в выделении непрерывного буфера в драйвере и использовании его в качестве промежуточного хранилища, если исходный буфер не подходит.

Ниже представлены типичные последовательности вызовов при использовании карты в зависимости от её назначения. Символы → используются для обозначения последовательности во времени.

Для буфера, который остаётся практически неизменным в течение всего времени между присоединением и отсоединением устройства:

bus_dmamem_alloc → bus_dmamap_load → …use buffer… → → bus_dmamap_unload → bus_dmamem_free

Для буфера, который часто изменяется и передаётся извне драйвера:

```
bus_dmamap_create ->
-> bus_dmamap_load -> bus_dmamap_sync(PRE...) -> do transfer ->
-> bus_dmamap_sync(POST...) -> bus_dmamap_unload ->
...
-> bus_dmamap_load -> bus_dmamap_sync(PRE...) -> do transfer ->
-> bus_dmamap_sync(POST...) -> bus_dmamap_unload ->
-> bus_dmamap_destroy
```

При загрузке карты, созданной `bus_dmamem_alloc()`, переданные адрес и размер буфера должны быть такими же, как использованные в `bus_dmamem_alloc()`. В этом случае гарантируется, что весь буфер будет отображён как один сегмент (так что обратный вызов

может основываться на этом предположении) и запрос будет выполнен немедленно (EINPROGRESS никогда не будет возвращен). Все, что нужно сделать обратному вызову в этом случае, — это сохранить физический адрес.

Типичный пример:

```
          static void
        alloc_callback(void *arg, bus_dma_segment_t *seg, int nseg, int error)
        {
          *(bus_addr_t *)arg = seg[0].ds_addr;
        }

          ...
          int error;
          struct somedata {
            ....
          };
          struct somedata *vsomedata; /* virtual address */
          bus_addr_t psomedata; /* physical bus-relative address */
          bus_dma_tag_t tag_somedata;
          bus_dmamap_t map_somedata;
          ...

          error=bus_dma_tag_create(parent_tag, alignment,
           boundary, lowaddr, highaddr, /*filter*/ NULL, /*filterarg*/ NULL,
           /*maxsize*/ sizeof(struct somedata), /*nsegments*/ 1,
           /*maxsegsz*/ sizeof(struct somedata), /*flags*/ 0,
           &tag_somedata);
          if(error)
          return error;

          error = bus_dmamem_alloc(tag_somedata, &vsomedata, /* flags*/ 0,
             &map_somedata);
          if(error)
             return error;

          bus_dmamap_load(tag_somedata, map_somedata, (void *)vsomedata,
             sizeof (struct somedata), alloc_callback,
             (void *) &psomedata, /*flags*/0);
```

Выглядит немного длинно и сложно, но это правильный способ. Практическое следствие таково: если несколько областей памяти выделяются всегда вместе, было бы отличной идеей объединить их все в одну структуру и выделять как единое целое (если ограничения выравнивания и границ позволяют).

При загрузке произвольного буфера в карту, созданную `bus_dmamap_create()`, необходимо принять специальные меры для синхронизации с обратным вызовом, если он будет задержан. Код будет выглядеть следующим образом:

```
        {
         int s;
         int error;

         s = splsoftvm();
         error = bus_dmamap_load(
             dmat,
             dmamap,
             buffer_ptr,
             buffer_len,
             callback,
             /*callback_arg*/ buffer_descriptor,
             /*flags*/0);
         if (error == EINPROGRESS) {
             /*
              * Do whatever is needed to ensure synchronization
              * with callback. Callback is guaranteed not to be started
              * until we do splx() or tsleep().
              */
            }
         splx(s);
        }
```

Два возможных подхода для обработки запросов:

1. Если запросы завершаются путём явной пометки их как выполненных (например, запросы CAM), то было бы проще поместить всю дальнейшую обработку в драйвер обратного вызова, который отмечал бы запрос по его завершении. В этом случае не потребуется много дополнительной синхронизации. По соображениям управления потоком может быть полезно заморозить очередь запросов до завершения этого запроса.
2. Если запросы завершаются при возврате функции (например, классические запросы на чтение или запись для символьных устройств), то в дескрипторе буфера должен быть установлен флаг синхронизации и вызвана функция `tsleep()`. Позже, когда будет вызван обратный вызов, он выполнит свою обработку и проверит этот флаг синхронизации. Если флаг установлен, обратный вызов должен инициировать пробуждение. При таком подходе функция обратного вызова может либо выполнить всю необходимую обработку (как в предыдущем случае), либо просто сохранить массив сегментов в дескрипторе буфера. Затем после завершения обратного вызова вызывающая функция может использовать этот сохранённый массив сегментов и выполнить всю обработку.

# 10.7. DMA

Прямой доступ к памяти (DMA) реализован в шине ISA через контроллер DMA (на самом деле их два, но это несущественная деталь). Чтобы сделать ранние устройства ISA простыми и дешёвыми, логика управления шиной и генерации адресов была сосредоточена в контроллере DMA. К счастью, FreeBSD предоставляет набор функций, которые в основном скрывают раздражающие детали работы контроллера DMA от драйверов устройств.

Самый простой случай — для достаточно интеллектуальных устройств. Например, устройства с bus mastering на PCI могут сами генерировать шинные циклы и адреса памяти. Единственное, что им действительно нужно от контроллера DMA, — это арбитраж шины. Для этой цели они притворяются каскадированными подчинёнными контроллерами DMA. И единственное, что требуется от системного контроллера DMA, — это включить каскадный режим на канале DMA, вызвав следующую функцию при присоединении драйвера:

```
void isa_dmacascade(int channel_number)
```

Все последующие действия выполняются путем программирования устройства. При отсоединении драйвера нет необходимости вызывать функции, связанные с DMA.

Для более простых устройств всё становится сложнее. Используются следующие функции:

- `int isa_dma_acquire(int chanel_number)`

  Зарезервировать канал DMA. Возвращает 0 при успехе или EBUSY, если канал уже зарезервирован этим или другим драйвером. Большинство устройств ISA не способны совместно использовать каналы DMA, поэтому обычно эта функция вызывается при присоединении устройства. Это резервирование стало избыточным с появлением современного интерфейса ресурсов шины, но всё ещё должно использоваться в дополнение к последнему. Если резервирование не использовать, то в дальнейшем другие процедуры DMA вызовут панику ядра.

- `int isa_dma_release(int chanel_number)`

  Освободить ранее зарезервированный канал DMA. На момент освобождения канала не должно быть активных передач (дополнительно устройство не должно пытаться инициировать передачу после освобождения канала).

- `void isa_dmainit(int chan, u_int bouncebufsize)`

  Выделить промежуточный буфер для использования с указанным каналом. Запрашиваемый размер буфера не может превышать 64 КБ. Этот промежуточный буфер будет автоматически использован в дальнейшем, если передаваемый буфер окажется не физически непрерывным, находится вне памяти, доступной шине ISA, или пересекает границу 64 КБ. Если передача всегда будет выполняться из буферов, соответствующих этим условиям (например, выделенных с помощью `bus_dmamem_alloc()` с соответствующими ограничениями), то вызов `isa_dmainit()` не требуется. Однако довольно удобно передавать произвольные данные с использованием контроллера DMA. Промежуточный буфер автоматически решит проблемы в ситуациях, когда данные разбросаны в памяти, и их надо собирать.

  - *chan* - номер канала
  - *bouncebufsize* - размер промежуточного буфера в байтах

- `void isa_dmastart(int flags, caddr_t addr, u_int nbytes, int chan)`

  Подготовка к началу передачи DMA. Эта функция должна быть вызвана для настройки контроллера DMA перед фактическим началом передачи на устройстве. Она проверяет, что буфер является непрерывным и попадает в диапазон памяти ISA, если нет, то

автоматически используется промежуточный буфер. Если требуется промежуточный буфер, но он не настроен с помощью `isa_dmainit()` или слишком мал для запрошенного размера передачи, система перейдет в состояние паники. В случае запроса на запись с промежуточным буфером данные будут автоматически скопированы в этот буфер.

- flags - битовая маска, определяющая тип выполняемой операции. Бит направления B_READ и B_WRITE являются взаимоисключающими.
  - B_READ - чтение с шины ISA в память
  - B_WRITE - запись из памяти на шину ISA
  - B_RAW - если установлен, то контроллер DMA запомнит буфер и после завершения передачи автоматически переинициализирует себя для повторной передачи того же буфера (конечно, драйвер может изменить данные в буфере перед инициированием следующей передачи на устройстве). Если не установлен, то параметры будут работать только для одной передачи, и перед инициированием следующей передачи снова потребуется вызвать `isa_dmastart()`. Использование B_RAW имеет смысл только если промежуточный буфер не используется.
- addr - виртуальный адрес буфера
- nbytes - длина буфера. Должна быть меньше или равна 64 КБ. Длина 0 не допускается: контроллер DMA интерпретирует это как 64 КБ, в то время как код ядра поймёт это как 0, что приведёт к непредсказуемым последствиям. Для каналов номер 4 и выше длина должна быть чётной, так как эти каналы передают по 2 байта за раз. В случае нечётной длины последний байт не будет передан.
- chan - номер канала
- `void isa_dmadone(int flags, caddr_t addr, int nbytes, int chan)`

  Синхронизировать память после того, как устройство сообщает о завершении передачи. Если это была операция чтения с промежуточным буфером, то данные будут скопированы из этого буфера в исходный буфер. Аргументы такие же, как у `isa_dmastart()`. Флаг B_RAW разрешён, но он никак не влияет на `isa_dmadone()`.

- `int isa_dmastatus(int channel_number)`

  Возвращает количество оставшихся для передачи байт в текущей передаче. Если флаг B_READ был установлен в `isa_dmastart()`, возвращаемое значение никогда не будет равно нулю. В конце передачи оно автоматически сбрасывается обратно к длине буфера. Обычное использование — проверка количества оставшихся байт после того, как устройство сигнализирует о завершении передачи. Если количество байт не равно 0, то, вероятно, в передаче произошла ошибка.

- `int isa_dmastop(int channel_number)`

  Прерывает текущую передачу и возвращает количество непереданных байтов.

## 10.8. xxx_isa_probe

Эта функция проверяет наличие устройства. Если драйвер поддерживает автоматическое

определение некоторых параметров конфигурации устройства (таких как вектор прерывания или адрес памяти), это автоматическое определение должно выполняться в данной процедуре.

Как и для любой другой шины, если устройство не может быть обнаружено, или обнаружено, но не прошло самопроверку, или возникла другая проблема, то возвращается положительное значение ошибки. Значение `ENXIO` должно возвращаться, если устройство отсутствует. Другие значения ошибок могут означать иные условия. Нулевые или отрицательные значения означают успех. Большинство драйверов возвращают ноль в случае успеха.

Отрицательные возвращаемые значения используются, когда устройство PnP поддерживает несколько интерфейсов. Например, старый совместимый интерфейс и новый расширенный интерфейс, которые поддерживаются разными драйверами. В этом случае оба драйвера обнаружат устройство. Драйвер, который возвращает большее значение в процедуре обнаружения, получает приоритет (другими словами, драйвер, возвращающий 0, имеет наивысший приоритет, возвращающий -1 — следующий, возвращающий -2 — за ним и так далее). В результате устройства, поддерживающие только старый интерфейс, будут обрабатываться старым драйвером (который должен возвращать -1 из процедуры probe), а устройства, поддерживающие также новый интерфейс, будут обрабатываться новым драйвером (который должен возвращать 0 из процедуры обнаружения).

Структура дескриптора устройства `xxx_softc` выделяется системой до вызова процедуры обнаружения. Если процедура обнаружения возвращает ошибку, дескриптор автоматически освобождается системой. Поэтому при возникновении ошибки обнаружения драйвер должен убедиться, что все ресурсы, использованные во время обнаружения, освобождены и ничто не мешает безопасному освобождению дескриптора.Если обнаружение завершается успешно, дескриптор сохраняется системой и позже передаётся в процедуру `xxx_isa_attach()`. Если драйвер возвращает отрицательное значение, он не может быть уверен, что получит наивысший приоритет и его процедура присоединения будет вызвана. Поэтому в этом случае он также должен освободить все ресурсы перед возвратом и, если необходимо, выделить их снова в процедуре присоединения. Когда `xxx_isa_probe()` возвращает 0, освобождение ресурсов перед возвратом также является хорошей практикой, и корректно работающий драйвер должен так поступать. Однако в случаях, когда возникают проблемы с освобождением ресурсов, драйверу разрешается сохранять ресурсы между возвратом 0 из процедуры обнаружения и выполнением процедуры присоединения.

Типичная процедура обнаружения начинается с получения дескриптора устройства и номера устройства:

```
        struct xxx_softc *sc = device_get_softc(dev);
         int unit = device_get_unit(dev);
         int pnperror;
         int error = 0;

         sc->dev = dev; /* link it back */
         sc->unit = unit;
```

Затем проверьте устройства PnP. Проверка осуществляется с помощью таблицы, содержащей список PnP ID, поддерживаемых этим драйвером, и удобочитаемые описания моделей устройств, соответствующих этим ID.

```
        pnperror=ISA_PNP_PROBE(device_get_parent(dev), dev,
        xxx_pnp_ids); if(pnperror == ENXIO) return ENXIO;
```

Логика работы `ISA_PNP_PROBE` следующая: если данная карта (устройство) не была обнаружена как PnP, то будет возвращено `ENOENT`. Если она была обнаружена как PnP, но её обнаруженный ID не совпадает ни с одним из ID в таблице, то возвращается `ENXIO`. Наконец, если устройство поддерживает PnP и его ID совпадает с одним из ID в таблице, возвращается `0`, а соответствующее описание из таблицы устанавливается с помощью `device_set_desc()`.

Если драйвер поддерживает только устройства PnP, то условие будет выглядеть следующим образом:

```
          if(pnperror != 0)
              return pnperror;
```

Для драйверов, которые не поддерживают PnP, не требуется специальной обработки, так как они передают пустую таблицу идентификаторов PnP и всегда будут получать ENXIO при вызове на PnP-карте.

Функция обнаружения обычно требует как минимум некоторый минимальный набор ресурсов, например, номер порта ввода-вывода, чтобы найти карту и проверить её. В зависимости от оборудования драйвер может автоматически обнаружить другие необходимые ресурсы. Устройства PnP имеют все ресурсы, предварительно установленные подсистемой PnP, поэтому драйверу не нужно обнаруживать их самостоятельно.

Обычно минимальная информация, необходимая для доступа к устройству, — это номер порта ввода-вывода. Затем некоторые устройства позволяют получить остальную информацию из регистров конфигурации устройства (хотя не все устройства это поддерживают). Поэтому сначала мы пытаемся получить начальное значение порта:

```
sc->port0 = bus_get_resource_start(dev,
       SYS_RES_IOPORT, 0 /*rid*/); if(sc->port0 == 0) return ENXIO;
```

Базовый адрес порта сохраняется в структуре softc для последующего использования. Если он будет использоваться очень часто, то вызов функции ресурса каждый раз будет неприемлемо медленным. Если мы не получаем порт, мы просто возвращаем ошибку. Некоторые драйверы устройств могут вместо этого быть умнее и попытаться обнаружить все возможные порты, например:

```
          /* table of all possible base I/O port addresses for this device */
          static struct xxx_allports {
```

```
        u_short port; /* port address */
        short used; /* flag: if this port is already used by some unit */
    } xxx_allports = {
        { 0x300, 0 },
        { 0x320, 0 },
        { 0x340, 0 },
        { 0, 0 } /* end of table */
    };

    ...
    int port, i;
    ...

    port =  bus_get_resource_start(dev, SYS_RES_IOPORT, 0 /*rid*/);
    if(port !=0 ) {
        for(i=0; xxx_allports[i].port!=0; i++) {
            if(xxx_allports[i].used || xxx_allports[i].port != port)
                continue;

            /* found it */
            xxx_allports[i].used = 1;
            /* do probe on a known port */
            return xxx_really_probe(dev, port);
        }
        return ENXIO; /* port is unknown or already used */
    }

    /* we get here only if we need to guess the port */
    for(i=0; xxx_allports[i].port!=0; i++) {
        if(xxx_allports[i].used)
            continue;

        /* mark as used - even if we find nothing at this port
         * at least we won't probe it in future
         */
         xxx_allports[i].used = 1;

        error = xxx_really_probe(dev, xxx_allports[i].port);
        if(error == 0) /* found a device at that port */
            return 0;
    }
    /* probed all possible addresses, none worked */
    return ENXIO;
```

Конечно, обычно для таких вещей следует использовать процедуру `identify()` драйвера. Однако может быть одна веская причина, почему лучше сделать это в `probe()`: если этот обнаружение может привести к сбою другого чувствительного устройства. Процедуры обнаружения упорядочены с учётом флага `sensitive`: чувствительные устройства проверяются первыми, а остальные устройства — позже. Но процедуры `identify()` вызываются до любого обнаружения, поэтому они не учитывают чувствительные

устройства и могут вызвать их сбой.

Вот, после того как мы получили начальный порт, необходимо установить количество портов (за исключением устройств PnP), так как в файле конфигурации ядра эта информация отсутствует.

```
        if(pnperror /* only for non-PnP devices */
        && bus_set_resource(dev, SYS_RES_IOPORT, 0, sc->port0,
        XXX_PORT_COUNT)<0)
            return ENXIO;
```

Наконец, выделите и активируйте часть адресного пространства порта (специальные значения start и end означают "используйте те, что мы установили через `bus_set_resource()`"):

```
         sc->port0_rid = 0;
         sc->port0_r = bus_alloc_resource(dev, SYS_RES_IOPORT,
         &sc->port0_rid,
             /*start*/ 0, /*end*/ ~0, /*count*/ 0, RF_ACTIVE);

         if(sc->port0_r == NULL)
             return ENXIO;
```

Теперь, имея доступ к регистрам с отображением на порты, мы можем каким-либо образом взаимодействовать с устройством и проверить, реагирует ли оно так, как ожидается. Если этого не происходит, вероятно, по этому адресу находится другое устройство или его там нет вовсе.

Обычно драйверы не настраивают обработчики прерываний до вызова процедуры присоединения. Вместо этого они выполняют проверки в режиме опроса, используя функцию `DELAY()` для таймаута. Процедура проверки никогда не должна зависать навсегда, все ожидания ответа от устройства должны выполняться с таймаутами. Если устройство не отвечает в течение заданного времени, вероятно, оно неисправно или неправильно настроено, и драйвер должен вернуть ошибку. При определении интервала таймаута следует давать устройству дополнительное время для надёжности: хотя предполагается, что `DELAY()` задерживает выполнение на одинаковое время на любой машине, существует некоторая погрешность, зависящая от конкретного процессора.

Если процедура проверки действительно хочет убедиться, что прерывания работают, она может также настроить и провести обнаружение прерываний. Однако это не рекомендуется.

```
        /* implemented in some very device-specific way */
        if(error = xxx_probe_ports(sc))
            goto bad; /* will deallocate the resources before returning */
```

Функция `xxx_probe_ports()` также может устанавливать описание устройства в зависимости от конкретной модели обнаруженного устройства. Но если поддерживается только одна модель устройства, это можно сделать и жёстко заданным способом. Конечно, для PnP-устройств поддержка PnP автоматически устанавливает описание из таблицы.

```
        if(pnperror)
            device_set_desc(dev, "Our device model 1234");
```

Затем процедура обнаружения должна либо определить диапазоны всех ресурсов, читая регистры конфигурации устройства, либо убедиться, что они были явно заданы пользователем. Мы рассмотрим это на примере встроенной памяти. Процедура обнаружения должна быть как можно менее навязчивой, поэтому выделение и проверку функциональности остальных ресурсов (кроме портов) лучше оставить для процедуры присоединения.

Адрес памяти может быть указан в конфигурационном файле ядра, а на некоторых устройствах он может быть предварительно настроен в энергонезависимых конфигурационных регистрах. Если доступны оба источника, и они различаются, какой из них следует использовать? Вероятно, если пользователь явно указал адрес в конфигурационном файле ядра, он знает, что делает, и этот адрес должен иметь приоритет. Пример реализации может выглядеть так:

```
        /* try to find out the config address first */
        sc->mem0_p = bus_get_resource_start(dev, SYS_RES_MEMORY, 0 /*rid*/);
        if(sc->mem0_p == 0) { /* nope, not specified by user */
            sc->mem0_p = xxx_read_mem0_from_device_config(sc);

        if(sc->mem0_p == 0)
                /* can't get it from device config registers either */
                goto bad;
        } else {
            if(xxx_set_mem0_address_on_device(sc) < 0)
                goto bad; /* device does not support that address */
        }

        /* just like the port, set the memory size,
         * for some devices the memory size would not be constant
         * but should be read from the device configuration registers instead
         * to accommodate different models of devices. Another option would
         * be to let the user set the memory size as "msize" configuration
         * resource which will be automatically handled by the ISA bus.
         */
         if(pnperror) { /* only for non-PnP devices */
            sc->mem0_size = bus_get_resource_count(dev, SYS_RES_MEMORY, 0 /*rid*/);
            if(sc->mem0_size == 0) /* not specified by user */
                sc->mem0_size = xxx_read_mem0_size_from_device_config(sc);

            if(sc->mem0_size == 0) {
```

```
                /* suppose this is a very old model of device without
                 * auto-configuration features and the user gave no preference,
                 * so assume the minimalistic case
                 * (of course, the real value will vary with the driver)
                 */
                sc->mem0_size = 8*1024;
            }

            if(xxx_set_mem0_size_on_device(sc) < 0)
                goto bad; /* device does not support that size */

            if(bus_set_resource(dev, SYS_RES_MEMORY, /*rid*/0,
                    sc->mem0_p, sc->mem0_size)<0)
                goto bad;
        } else {
            sc->mem0_size = bus_get_resource_count(dev, SYS_RES_MEMORY, 0 /*rid*/);
        }
```

Ресурсы для IRQ и DRQ легко проверить по аналогии.

Если всё прошло успешно, то освободите все ресурсы и верните успешный статус.

```
        xxx_free_resources(sc);
        return 0;
```

Наконец, обработайте проблемные ситуации. Все ресурсы должны быть освобождены перед возвратом. Мы используем тот факт, что перед передачей нам структуры `softc` она обнуляется, поэтому мы можем определить, был ли выделен какой-либо ресурс: если его дескриптор не равен нулю.

```
        bad:

        xxx_free_resources(sc);
        if(error)
              return error;
        else /* exact error is unknown */
            return ENXIO;
```

Вот и всё для процедуры обнаружения. Освобождение ресурсов выполняется из нескольких мест, поэтому оно вынесено в функцию, которая может выглядеть так:

```
static void
          xxx_free_resources(sc)
            struct xxx_softc *sc;
        {
            /* check every resource and free if not zero */
```

```
        /* interrupt handler */
        if(sc->intr_r) {
            bus_teardown_intr(sc->dev, sc->intr_r, sc->intr_cookie);
            bus_release_resource(sc->dev, SYS_RES_IRQ, sc->intr_rid,
                sc->intr_r);
            sc->intr_r = 0;
        }

        /* all kinds of memory maps we could have allocated */
        if(sc->data_p) {
            bus_dmamap_unload(sc->data_tag, sc->data_map);
            sc->data_p = 0;
        }
         if(sc->data) { /* sc->data_map may be legitimately equal to 0 */
            /* the map will also be freed */
            bus_dmamem_free(sc->data_tag, sc->data, sc->data_map);
            sc->data = 0;
        }
        if(sc->data_tag) {
            bus_dma_tag_destroy(sc->data_tag);
            sc->data_tag = 0;
        }

        ... free other maps and tags if we have them ...

        if(sc->parent_tag) {
            bus_dma_tag_destroy(sc->parent_tag);
            sc->parent_tag = 0;
        }

        /* release all the bus resources */
        if(sc->mem0_r) {
            bus_release_resource(sc->dev, SYS_RES_MEMORY, sc->mem0_rid,
                sc->mem0_r);
            sc->mem0_r = 0;
        }
        ...
        if(sc->port0_r) {
            bus_release_resource(sc->dev, SYS_RES_IOPORT, sc->port0_rid,
                sc->port0_r);
            sc->port0_r = 0;
        }
    }
```

## 10.9. xxx_isa_attach

Процедура присоединения фактически подключает драйвер к системе, если процедура обнаружения вернула успех, и система решила подключить этот драйвер. Если процедура обнаружения вернула 0, то процедура присоединения может ожидать, что получит

структуру устройства `softc` в неизменном виде, как она была установлена процедурой обнаружения. Также, если обнаружение возвращает 0, она можно ожидать, что процедура присоединения для этого устройства будет вызвана в какой-то момент в будущем. Если процедура обнаружения возвращает отрицательное значение, то драйвер не может делать никаких из этих предположений.

Процедура присоединение возвращает 0 при успешном завершении или код ошибки в противном случае.

Процедура присоединения начинается так же, как и процедура обнаружения, с помещения часто используемых данных в более доступные переменные.

```
        struct xxx_softc *sc = device_get_softc(dev);
        int unit = device_get_unit(dev);
        int error = 0;
```

Затем выделите и активируйте все необходимые ресурсы. Обычно диапазон портов освобождается перед возвратом из обнаружения, поэтому его необходимо выделить снова. Мы предполагаем, что процедура обнаружения корректно установила все диапазоны ресурсов, а также сохранила их в структуре softc. Если процедура обнаружения оставила некоторые ресурсы выделенными, то их не нужно выделять снова (это будет считаться ошибкой).

```
        sc->port0_rid = 0;
        sc->port0_r = bus_alloc_resource(dev, SYS_RES_IOPORT,  &sc->port0_rid,
            /*start*/ 0, /*end*/ ~0, /*count*/ 0, RF_ACTIVE);

        if(sc->port0_r == NULL)
             return ENXIO;

        /* on-board memory */
        sc->mem0_rid = 0;
        sc->mem0_r = bus_alloc_resource(dev, SYS_RES_MEMORY,  &sc->mem0_rid,
            /*start*/ 0, /*end*/ ~0, /*count*/ 0, RF_ACTIVE);

        if(sc->mem0_r == NULL)
              goto bad;

        /* get its virtual address */
        sc->mem0_v = rman_get_virtual(sc->mem0_r);
```

Канал запроса DMA (DRQ) выделяется аналогично. Для его инициализации используйте функции семейства `isa_dma*()`. Например:

```
isa_dmacascade(sc→drq0);
```

Строка запроса прерывания (IRQ) является особенной. Помимо выделения, обработчик прерывания драйвера должен быть связан с ней. Исторически в старых драйверах ISA

аргумент, передаваемый системой обработчику прерывания, был номером устройства. Однако в современных драйверах принято передавать указатель на структуру `softc`. Важная причина этого заключается в том, что когда структуры `softc` выделяются динамически, получение номера устройства из `softc` является простым, в то время как получение `softc` из номера устройства затруднительно. Кроме того, такое соглашение делает драйверы для различных шин более однородными и позволяет им совместно использовать код: каждая шина получает свои собственные процедуры обнаружения, присоединения, отсоединения и другие специфичные для шины функции, в то время как основная часть кода драйвера может быть общей для них.

```
        sc->intr_rid = 0;
        sc->intr_r = bus_alloc_resource(dev, SYS_RES_MEMORY,  &sc->intr_rid,
              /*start*/ 0, /*end*/ ~0, /*count*/ 0, RF_ACTIVE);

        if(sc->intr_r == NULL)
            goto bad;

        /*
         * XXX_INTR_TYPE is supposed to be defined depending on the type of
         * the driver, for example as INTR_TYPE_CAM for a CAM driver
         */
        error = bus_setup_intr(dev, sc->intr_r, XXX_INTR_TYPE,
            (driver_intr_t *) xxx_intr, (void *) sc, &sc->intr_cookie);
        if(error)
            goto bad;
```

Если устройству необходимо выполнять DMA в основную память, то эта память должна быть выделена, как описано ранее:

```
        error=bus_dma_tag_create(NULL, /*alignment*/ 4,
            /*boundary*/ 0, /*lowaddr*/ BUS_SPACE_MAXADDR_24BIT,
            /*highaddr*/ BUS_SPACE_MAXADDR, /*filter*/ NULL, /*filterarg*/ NULL,
            /*maxsize*/ BUS_SPACE_MAXSIZE_24BIT,
            /*nsegments*/ BUS_SPACE_UNRESTRICTED,
            /*maxsegsz*/ BUS_SPACE_MAXSIZE_24BIT, /*flags*/ 0,
            &sc->parent_tag);
        if(error)
            goto bad;

        /* many things get inherited from the parent tag
         * sc->data is supposed to point to the structure with the shared data,
         * for example for a ring buffer it could be:
         * struct {
         *   u_short rd_pos;
         *   u_short wr_pos;
         *   char    bf[XXX_RING_BUFFER_SIZE]
         * } *data;
         */
        error=bus_dma_tag_create(sc->parent_tag, 1,
```

```
            0, BUS_SPACE_MAXADDR, 0, /*filter*/ NULL, /*filterarg*/ NULL,
            /*maxsize*/ sizeof(* sc->data), /*nsegments*/ 1,
            /*maxsegsz*/ sizeof(* sc->data), /*flags*/ 0,
            &sc->data_tag);
        if(error)
            goto bad;

        error = bus_dmamem_alloc(sc->data_tag, &sc->data, /* flags*/ 0,
            &sc->data_map);
        if(error)
             goto bad;

        /* xxx_alloc_callback() just saves the physical address at
         * the pointer passed as its argument, in this case &sc->data_p.
         * See details in the section on bus memory mapping.
         * It can be implemented like:
         *
         * static void
         * xxx_alloc_callback(void *arg, bus_dma_segment_t *seg,
         *     int nseg, int error)
         * {
         *    *(bus_addr_t *)arg = seg[0].ds_addr;
         * }
         */
        bus_dmamap_load(sc->data_tag, sc->data_map, (void *)sc->data,
            sizeof (* sc->data), xxx_alloc_callback, (void *) &sc->data_p,
            /*flags*/0);
```

После выделения всех необходимых ресурсов устройство должно быть инициализировано. Инициализация может включать проверку работоспособности всех ожидаемых функций.

```
        if(xxx_initialize(sc) < 0)
             goto bad;
```

Подсистема шины автоматически выводит на консоль описание устройства, установленное при проверке. Однако, если драйвер хочет вывести дополнительную информацию об устройстве, он может это сделать, например:

```
    device_printf(dev, "has on-card FIFO buffer of %d bytes\n", sc->fifosize);
```

Если в процессе выполнения инициализации возникают какие-либо проблемы, рекомендуется выводить сообщения об этих проблемах перед возвратом ошибки.

Последним шагом процедуры присоединения является подключение устройства к его функциональной подсистеме в ядре. Конкретный способ зависит от типа драйвера: символьное устройство, блочное устройство, сетевое устройство, устройство шины CAM SCSI и так далее.

Если всё прошло успешно, вернуть успех.

```
        error = xxx_attach_subsystem(sc);
        if(error)
            goto bad;

        return 0;
```

Наконец, обработаем проблемные ситуации. Все ресурсы должны быть освобождены перед возвратом ошибки. Мы используем тот факт, что перед передачей структуры `softc` она обнуляется, поэтому мы можем определить, был ли выделен какой-либо ресурс: если его дескриптор ненулевой.

```
        bad:

        xxx_free_resources(sc);
        if(error)
            return error;
        else /* exact error is unknown */
            return ENXIO;
```

Вот и всё для процедуры присоединения.

## 10.10. xxx_isa_detach

Если эта функция присутствует в драйвере и драйвер скомпилирован как загружаемый модуль, то драйвер получает возможность быть выгруженным. Это важная функция, если оборудование поддерживает горячее подключение. Однако шина ISA не поддерживает горячее подключение, поэтому эта функция не особенно важна для устройств ISA. Возможность выгрузки драйвера может быть полезна при его отладке, но во многих случаях установка новой версии драйвера потребуется только после того, как старая версия каким-то образом заблокирует систему и перезагрузка все равно будет необходима, поэтому усилия, затраченные на написание процедуры отсоединения, могут не окупиться. Другой аргумент, что выгрузка позволит обновлять драйверы на рабочей машине, кажется в основном теоретическим. Установка новой версии драйвера — это опасная операция, которую никогда не следует выполнять на рабочей машине (и которая не разрешена, когда система работает в безопасном режиме). Тем не менее, процедура отсоединения может быть предоставлена для полноты.

Процедура отсоединения возвращает 0, если драйвер был успешно отсоединён, или код ошибки в противном случае.

Логика отсоединения является зеркальной по отношению к присоединению. Первое, что нужно сделать, — это отсоединить драйвер от его подсистемы ядра. Если устройство в настоящее время открыто, у драйвера есть два варианта: отказаться от отсоединения или принудительно закрыть устройство и продолжить отсоединение. Выбор зависит от возможности конкретной подсистемы ядра выполнить принудительное закрытие и от

предпочтений автора драйвера. Как правило, принудительное закрытие кажется предпочтительным вариантом.

```
        struct xxx_softc *sc = device_get_softc(dev);
        int error;

        error = xxx_detach_subsystem(sc);
        if(error)
            return error;
```

Далее драйвер может сбросить аппаратное обеспечение в согласованное состояние. Это включает остановку любых текущих передач, отключение каналов DMA и прерываний, чтобы избежать повреждения памяти устройством. Для большинства драйверов это именно то, что делает процедура выключения, поэтому, если она присутствует в драйвере, мы можем просто вызвать её.

```
xxx_isa_shutdown(dev);
```

И наконец освободить все ресурсы и вернуть успех.

```
        xxx_free_resources(sc);
        return 0;
```

# 10.11. xxx_isa_shutdown

Эта процедура вызывается, когда система собирается быть выключена. Ожидается, что она приведет оборудование в согласованное состояние. Для большинства устройств ISA не требуется никаких специальных действий, поэтому функция не является действительно необходимой, так как устройство будет переинициализировано при перезагрузке в любом случае. Однако некоторые устройства должны быть выключены с помощью специальной процедуры, чтобы убедиться, что они будут правильно обнаружены после мягкой перезагрузки (это особенно актуально для многих устройств с проприетарными протоколами идентификации). В любом случае отключение DMA и прерываний в регистрах устройства и остановка любых текущих передач — это хорошая идея. Точные действия зависят от оборудования, поэтому мы не рассматриваем их здесь подробно.

# 10.12. xxx_intr

Обработчик прерывания вызывается при получении прерывания, которое может быть от данного конкретного устройства. Шина ISA не поддерживает разделение прерываний (за исключением некоторых специальных случаев), поэтому на практике, если вызывается обработчик прерывания, то прерывание почти наверняка поступило от его устройства. Тем не менее, обработчик прерывания должен опросить регистры устройства и убедиться, что прерывание было сгенерировано его устройством. Если нет, он должен просто вернуть управление.

Старая практика для драйверов ISA заключалась в получении номера устройства в качестве аргумента. Это устарело, и новые драйверы получают любой аргумент, указанный для них в процедуре присоединения при вызове `bus_setup_intr()`. Согласно новой практике, это должен быть указатель на структуру softc. Таким образом, обработчик прерываний обычно начинается так:

```
static void
xxx_intr(struct xxx_softc *sc)
{
```

Он выполняется на уровне приоритета прерывания, указанном параметром типа прерывания в `bus_setup_intr()`. Это означает, что все остальные прерывания того же типа, а также все программные прерывания, отключены.

Во избежание гонок это обычно записывается в виде цикла:

```
while(xxx_interrupt_pending(sc)) {
    xxx_process_interrupt(sc);
    xxx_acknowledge_interrupt(sc);
}
```

Обработчик прерывания должен подтвердить прерывание только устройству, но не контроллеру прерываний, система позаботится о последнем.

# Глава 11. Устройства PCI

Эта глава расскажет о механизмах FreeBSD для написания драйвера устройства на шине PCI.

## 11.1. Обнаружение и подключение

Информация о том, как код шины PCI перебирает неприсоединённые устройства и проверяет, сможет ли только что загруженный kld присоединиться к любому из них.

### 11.1.1. Пример исходного кода драйвера (mypci.c)

```
/*
 * Simple KLD to play with the PCI functions.
 *
 * Murray Stokely
 */

#include <sys/param.h>        /* defines used in kernel.h */
#include <sys/module.h>
#include <sys/systm.h>
#include <sys/errno.h>
#include <sys/kernel.h>       /* types used in module initialization */
#include <sys/conf.h>     /* cdevsw struct */
#include <sys/uio.h>      /* uio struct */
#include <sys/malloc.h>
#include <sys/bus.h>      /* structs, prototypes for pci bus stuff and DEVMETHOD
macros! */

#include <machine/bus.h>
#include <sys/rman.h>
#include <machine/resource.h>

#include <dev/pci/pcivar.h>   /* For pci_get macros! */
#include <dev/pci/pcireg.h>

/* The softc holds our per-instance data. */
struct mypci_softc {
    device_t    my_dev;
    struct cdev *my_cdev;
};

/* Function prototypes */
static d_open_t     mypci_open;
static d_close_t    mypci_close;
static d_read_t     mypci_read;
static d_write_t    mypci_write;

/* Character device entry points */
```

```
static struct cdevsw mypci_cdevsw = {
    .d_version =    D_VERSION,
    .d_open =   mypci_open,
    .d_close =  mypci_close,
    .d_read =   mypci_read,
    .d_write =  mypci_write,
    .d_name =   "mypci",
};

/*
 * In the cdevsw routines, we find our softc by using the si_drv1 member
 * of struct cdev.  We set this variable to point to our softc in our
 * attach routine when we create the /dev entry.
 */

int
mypci_open(struct cdev *dev, int oflags, int devtype, struct thread *td)
{
    struct mypci_softc *sc;

    /* Look up our softc. */
    sc = dev->si_drv1;
    device_printf(sc->my_dev, "Opened successfully.\n");
    return (0);
}

int
mypci_close(struct cdev *dev, int fflag, int devtype, struct thread *td)
{
    struct mypci_softc *sc;

    /* Look up our softc. */
    sc = dev->si_drv1;
    device_printf(sc->my_dev, "Closed.\n");
    return (0);
}

int
mypci_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct mypci_softc *sc;

    /* Look up our softc. */
    sc = dev->si_drv1;
    device_printf(sc->my_dev, "Asked to read %zd bytes.\n", uio->uio_resid);
    return (0);
}

int
mypci_write(struct cdev *dev, struct uio *uio, int ioflag)
{
```

```
    struct mypci_softc *sc;

    /* Look up our softc. */
    sc = dev->si_drv1;
    device_printf(sc->my_dev, "Asked to write %zd bytes.\n", uio->uio_resid);
    return (0);
}

/* PCI Support Functions */

/*
 * Compare the device ID of this device against the IDs that this driver
 * supports.  If there is a match, set the description and return success.
 */
static int
mypci_probe(device_t dev)
{

    device_printf(dev, "MyPCI Probe\nVendor ID : 0x%x\nDevice ID : 0x%x\n",
        pci_get_vendor(dev), pci_get_device(dev));

    if (pci_get_vendor(dev) == 0x11c1) {
        printf("We've got the Winmodem, probe successful!\n");
        device_set_desc(dev, "WinModem");
        return (BUS_PROBE_DEFAULT);
    }
    return (ENXIO);
}

/* Attach function is only called if the probe is successful. */

static int
mypci_attach(device_t dev)
{
    struct mypci_softc *sc;

    printf("MyPCI Attach for : deviceID : 0x%x\n", pci_get_devid(dev));

    /* Look up our softc and initialize its fields. */
    sc = device_get_softc(dev);
    sc->my_dev = dev;

    /*
     * Create a /dev entry for this device.  The kernel will assign us
     * a major number automatically.  We use the unit number of this
     * device as the minor number and name the character device
     * "mypci<unit>".
     */
    sc->my_cdev = make_dev(&mypci_cdevsw, device_get_unit(dev),
        UID_ROOT, GID_WHEEL, 0600, "mypci%u", device_get_unit(dev));
    sc->my_cdev->si_drv1 = sc;
```

```
    printf("Mypci device loaded.\n");
    return (0);
}

/* Detach device. */

static int
mypci_detach(device_t dev)
{
    struct mypci_softc *sc;

    /* Teardown the state in our softc created in our attach routine. */
    sc = device_get_softc(dev);
    destroy_dev(sc->my_cdev);
    printf("Mypci detach!\n");
    return (0);
}

/* Called during system shutdown after sync. */

static int
mypci_shutdown(device_t dev)
{

    printf("Mypci shutdown!\n");
    return (0);
}

/*
 * Device suspend routine.
 */
static int
mypci_suspend(device_t dev)
{

    printf("Mypci suspend!\n");
    return (0);
}

/*
 * Device resume routine.
 */
static int
mypci_resume(device_t dev)
{

    printf("Mypci resume!\n");
    return (0);
}

static device_method_t mypci_methods[] = {
```

```
    /* Device interface */
    DEVMETHOD(device_probe,     mypci_probe),
    DEVMETHOD(device_attach,    mypci_attach),
    DEVMETHOD(device_detach,    mypci_detach),
    DEVMETHOD(device_shutdown,  mypci_shutdown),
    DEVMETHOD(device_suspend,   mypci_suspend),
    DEVMETHOD(device_resume,    mypci_resume),

    DEVMETHOD_END
};

static devclass_t mypci_devclass;

DEFINE_CLASS_0(mypci, mypci_driver, mypci_methods, sizeof(struct mypci_softc));
DRIVER_MODULE(mypci, pci, mypci_driver, mypci_devclass, 0, 0);
```

### 11.1.2. Makefile для примера драйвера

```
# Makefile for mypci driver

KMOD=   mypci
SRCS=   mypci.c
SRCS+=  device_if.h bus_if.h pci_if.h

.include <bsd.kmod.mk>
```

Если вы поместите исходный файл выше и Makefile в каталог, вы можете запустить `make` для компиляции примера драйвера. Дополнительно можно выполнить `make load` для загрузки драйвера в текущее ядро и `make unload` для выгрузки драйвера после его загрузки.

### 11.1.3. Дополнительные ресурсы

- Группа по стандартам PCI
- PCI System Architecture, Fourth Edition by Tom Shanley, et al.

## 11.2. Ресурсы шины

FreeBSD предоставляет объектно-ориентированный механизм для запроса ресурсов от родительской шины. Почти все устройства будут дочерними элементами какого-либо типа шины (PCI, ISA, USB, SCSI и т.д.), и этим устройствам необходимо получать ресурсы от своей родительской шины (такие как сегменты памяти, линии прерываний или каналы DMA).

### 11.2.1. Регистры базовых адресов

Для выполнения каких-либо полезных действий с устройством PCI необходимо получить *регистры базовых адресов* (BAR) из конфигурационного пространства PCI. Специфичные для PCI детали получения BAR абстрагированы в функции `bus_alloc_resource()`.

Например, типичный драйвер может содержать что-то подобное в функции `attach()`:

```
    sc->bar0id = PCIR_BAR(0);
    sc->bar0res = bus_alloc_resource(dev, SYS_RES_MEMORY, &sc->bar0id,
                  0, ~0, 1, RF_ACTIVE);
    if (sc->bar0res == NULL) {
        printf("Memory allocation of PCI base register 0 failed!\n");
        error = ENXIO;
        goto fail1;
    }

    sc->bar1id = PCIR_BAR(1);
    sc->bar1res = bus_alloc_resource(dev, SYS_RES_MEMORY, &sc->bar1id,
                  0, ~0, 1, RF_ACTIVE);
    if (sc->bar1res == NULL) {
        printf("Memory allocation of PCI base register 1 failed!\n");
        error =  ENXIO;
        goto fail2;
    }
    sc->bar0_bt = rman_get_bustag(sc->bar0res);
    sc->bar0_bh = rman_get_bushandle(sc->bar0res);
    sc->bar1_bt = rman_get_bustag(sc->bar1res);
    sc->bar1_bh = rman_get_bushandle(sc->bar1res);
```

Дескрипторы для каждого регистра базовых адресов хранятся в структуре `softc`, чтобы их можно было использовать для записи на устройство в дальнейшем.

Эти дескрипторы затем могут быть использованы для чтения или записи из регистров устройства с помощью функций `bus_space_*`. Например, драйвер может содержать сокращённую функцию для чтения из специфичного для платы регистра, как показано ниже:

```
uint16_t
board_read(struct ni_softc *sc, uint16_t address)
{
    return bus_space_read_2(sc->bar1_bt, sc->bar1_bh, address);
}
```

Аналогично, можно записать в регистры с помощью:

```
void
board_write(struct ni_softc *sc, uint16_t address, uint16_t value)
{
    bus_space_write_2(sc->bar1_bt, sc->bar1_bh, address, value);
}
```

Эти функции существуют в 8-битных, 16-битных и 32-битных версиях, и вам следует

использовать bus_space_{read|write}_{1|2|4} соответственно.



В FreeBSD 7.0 и более поздних версиях вы можете использовать функции bus_* вместо bus_space_*. Функции bus_* принимают указатель на структуру resource * вместо тега шины и дескриптора. Таким образом, вы можете удалить тег шины и дескриптор шины из softc и переписать функцию board_read() как:

```
uint16_t
board_read(struct ni_softc *sc, uint16_t address)
{
    return (bus_read(sc->bar1res, address));
}
```

### 11.2.2. Прерывания

Прерывания выделяются объектно-ориентированным кодом шины аналогично ресурсам памяти. Сначала ресурс IRQ должен быть выделен из родительской шины, а затем должен быть настроен обработчик прерывания для работы с этим IRQ.

Вот пример из функции attach() устройства, который скажет больше, чем слова.

```
/* Get the IRQ resource */

    sc->irqid = 0x0;
    sc->irqres = bus_alloc_resource(dev, SYS_RES_IRQ, &(sc->irqid),
                 0, ~0, 1, RF_SHAREABLE | RF_ACTIVE);
    if (sc->irqres == NULL) {
    printf("IRQ allocation failed!\n");
    error = ENXIO;
    goto fail3;
    }

    /* Now we should set up the interrupt handler */

    error = bus_setup_intr(dev, sc->irqres, INTR_TYPE_MISC,
               my_handler, sc, &(sc->handler));
    if (error) {
    printf("Couldn't set up irq\n");
    goto fail4;
    }
```

Некоторые меры предосторожности должны быть приняты в процедуре отключения драйвера. Необходимо остановить поток прерываний устройства и удалить обработчик прерываний. Как только bus_teardown_intr() завершится, можно быть уверенным, что обработчик прерываний больше не будет вызываться и все потоки, которые могли выполнять этот обработчик, завершили работу. Поскольку эта функция может засыпать,

нельзя удерживать какие-либо мьютексы при её вызове.

### 11.2.3. DMA

Этот раздел устарел и приведён только в исторических целях. Правильный способ решения этих проблем — использование функций `bus_space_dma*()`. Этот абзац можно удалить, когда раздел будет обновлён с учётом данного подхода. Однако на данный момент API находится в состоянии изменения, поэтому, когда он стабилизируется, будет полезно обновить этот раздел соответствующим образом.

На ПК периферийные устройства, которые хотят использовать DMA с управлением шиной, должны работать с физическими адресами. Это проблема, поскольку FreeBSD использует виртуальную память и работает почти исключительно с виртуальными адресами. К счастью, существует функция `vtophys()`, которая поможет.

```
#include <vm/vm.h>
#include <vm/pmap.h>

#define vtophys(virtual_address) (...)
```

Однако решение немного отличается на alpha, и на самом деле нам нужна функция под названием `vtobus()`.

```
#if defined(__alpha__)
#define vtobus(va)      alpha_XXX_dmamap((vm_offset_t)va)
#else
#define vtobus(va)      vtophys(va)
#endif
```

### 11.2.4. Освобождение ресурсов

Очень важно освободить все ресурсы, которые были выделены во время `attach()`. Необходимо внимательно следить за освобождением правильных ресурсов даже в случае ошибки, чтобы система оставалась работоспособной при завершении работы вашего драйвера.

# Глава 12. Контроллеры SCSI с общим методом доступа (CAM)

## 12.1. Обзор

Этот документ предполагает, что читатель имеет общее представление о драйверах устройств в FreeBSD и о протоколе SCSI. Большая часть информации в этом документе была извлечена из драйверов:

- ncr (/sys/pci/ncr.c) от Wolfgang Stanglmeier и Stefan Esser
- sym (/sys/dev/sym/sym_hipd.c) от Gerard Roudier
- aic7xxx (/sys/dev/aic7xxx/aic7xxx.c) от Justin T. Gibbs

и из самого кода CAM (автор Justin T. Gibbs, см. /sys/cam/*). Когда какое-то решение выглядело наиболее логичным и было практически дословно взято из кода Justin T. Gibbs, я отмечал его как "рекомендуемое".

Документ иллюстрирован примерами на псевдокоде. Хотя иногда примеры содержат много деталей и выглядят как настоящий код, это всё ещё псевдокод. Он был написан, чтобы продемонстрировать концепции в понятной форме. Для реального драйвера могут быть более модульные и эффективные подходы. Также он абстрагируется от деталей оборудования, а также от вопросов, которые могли бы затмить демонстрируемые концепции или которые предполагается описать в других главах руководства разработчика. Такие детали обычно показаны в виде вызовов функций с описательными именами, комментариев или псевдооператоров. К счастью, полные примеры из реальной жизни со всеми деталями можно найти в реальных драйверах.

## 12.2. Общая архитектура

CAM означает Common Access Method (Общий Метод Доступа). Это универсальный способ адресации шин ввода-вывода в стиле SCSI. Это позволяет отделить общие драйверы устройств от драйверов, управляющих шиной ввода-вывода: например, драйвер диска получает возможность управлять дисками как на SCSI, IDE, так и на любой другой шине, так что часть драйвера диска не нужно переписывать (или копировать и изменять) для каждой новой шины ввода-вывода. Таким образом, двумя наиболее важными активными сущностями являются:

- *Модули периферийных устройств* - драйвер для периферийных устройств (диски, ленты, CD-ROM и т.д.)
- *Модули интерфейса SCSI* (SIM) - драйверы адаптеров шины для подключения к шине ввода-вывода, такой как SCSI или IDE.

Периферийный драйвер получает запросы от ОС, преобразует их в последовательность команд SCSI и передаёт эти команды SCSI модулю интерфейса SCSI. Модуль интерфейса SCSI отвечает за передачу этих команд реальному оборудованию (или, если оборудование не

поддерживает SCSI, а использует, например, IDE, также преобразует команды SCSI в собственные команды оборудования).

Так как мы заинтересованы в написании драйвера адаптера SCSI, с этого момента мы будем рассматривать всё с точки зрения модуля SCSI-интерфейса (SIM).

## 12.3. Глобальные переменные и Шаблонный код

Типичный драйвер SIM должен включать следующие заголовочные файлы, связанные с CAM:

```
#include <cam/cam.h>
#include <cam/cam_ccb.h>
#include <cam/cam_sim.h>
#include <cam/cam_xpt_sim.h>
#include <cam/cam_debug.h>
#include <cam/scsi/scsi_all.h>
```

## 12.4. Конфигурация устройства: xxx_attach

Первое, что должен сделать каждый драйвер SIM, — это зарегистрироваться в подсистеме CAM. Это выполняется в функции `xxx_attach()` драйвера (здесь и далее xxx_ используется для обозначения уникального префикса имени драйвера). Сама функция `xxx_attach()` вызывается кодом автонастройки системной шины, который мы здесь не описываем.

Это достигается в несколько этапов: сначала необходимо выделить очередь запросов, связанных с этой SIM:

```
    struct cam_devq *devq;

    if ((devq = cam_simq_alloc(SIZE)) == NULL) {
        error; /* some code to handle the error */
    }
```

Вот `SIZE` — это размер выделяемой очереди, максимальное количество запросов, которые она может содержать. Это количество запросов, которые драйвер SIM может обрабатывать параллельно на одной SCSI-карте. Обычно его можно вычислить как:

```
SIZE = NUMBER_OF_SUPPORTED_TARGETS * MAX_SIMULTANEOUS_COMMANDS_PER_TARGET
```

Далее мы создаем описание нашего SIM:

```
    struct cam_sim *sim;

    if ((sim = cam_sim_alloc(action_func, poll_func, driver_name,
```

```
            softc, unit, mtx, max_dev_transactions,
            max_tagged_dev_transactions, devq)) == NULL) {
        cam_simq_free(devq);
        error; /* some code to handle the error */
    }
```

Обратите внимание, что если мы не сможем создать дескриптор SIM, мы также освобождаем `devq`, потому что больше ничего не можем с ним сделать и хотим сэкономить память.

Если SCSI-карта имеет несколько шин SCSI, то каждой шине требуется собственная структура `cam_sim`.

Интересный вопрос: что делать, если SCSI-карта имеет более одной SCSI-шины, нужна ли одна структура `devq` на карту или на SCSI-шину? Ответ, приведённый в комментариях к коду CAM, таков: как угодно, на усмотрение автора драйвера.

Аргументы:

- `action_func` - указатель на функцию `xxx_action` драйвера.

```
static void xxx_action(struct cam_sim *, union ccb *);
```

- `poll_func` - указатель на функцию `xxx_poll()` драйвера

  ```
  static void xxx_poll(struct cam_sim *);
  ```

- `driver_name` — имя фактического драйвера, например `ncr` или `wds`.
- `softc` — указатель на внутренний дескриптор драйвера для данной SCSI-карты. Этот указатель будет использоваться драйвером в дальнейшем для получения приватных данных.
- unit - номер управляющего устройства, например, для контроллера "mps0" это число будет 0
- mtx - Блокировка, связанная с данной SIM. Для SIM, которые не поддерживают блокировку, передаётся Giant. Для SIM, которые поддерживают, передаётся блокировка, используемая для защиты структур данных этой SIM. Эта блокировка будет удерживаться при вызовах xxx_action и xxx_poll.
- max_dev_transactions - максимальное количество одновременных транзакций на целевом SCSI-устройстве в режиме без тегов. Это значение почти всегда равно 1, за исключением возможных исключений только для не-SCSI карт. Также драйверы, которые надеются получить преимущество, подготавливая одну транзакцию во время выполнения другой, могут установить его в 2, но это не кажется оправданным из-за сложности.
- max_tagged_dev_transactions - то же самое, но в режиме с тегами. Теги — это способ в SCSI инициировать несколько транзакций на устройстве: каждая транзакция получает

уникальный тег и отправляется на устройство. Когда устройство завершает транзакцию, оно возвращает результат вместе с тегом, чтобы SCSI-адаптер (и драйвер) могли определить, какая транзакция была завершена. Этот аргумент также известен как максимальная глубина тега. Он зависит от возможностей SCSI-адаптера.

Наконец, мы регистрируем шины SCSI, связанные с нашим SCSI-адаптером:

```
    if (xpt_bus_register(sim, softc, bus_number) != CAM_SUCCESS) {
        cam_sim_free(sim, /*free_devq*/ TRUE);
        error; /* some code to handle the error */
    }
```

Если существует одна структура `devq` на каждую шину SCSI (т.е. мы рассматриваем карту с несколькими шинами как несколько карт с одной шиной каждая), то номер шины всегда будет 0, в противном случае каждая шина на SCSI-карте должна получить уникальный номер. Каждой шине требуется своя отдельная структура `cam_sim`.

После этого наш контроллер полностью подключён к системе CAM. Значение `devq` теперь можно отбросить: sim будет передаваться в качестве аргумента во всех последующих вызовах из CAM, а devq можно получить из него.

CAM предоставляет инфраструктуру для подобных асинхронных событий. Некоторые события возникают на нижних уровнях (драйверы SIM), некоторые — в драйверах периферийных устройств, а некоторые — в самой подсистеме CAM. Любой драйвер может зарегистрировать обработчики для определённых типов асинхронных событий, чтобы получать уведомления при их возникновении.

Типичным примером такого события является сброс устройства. Каждая транзакция и событие идентифицируют устройства, к которым они применяются, с помощью "пути". Специфичные для целевого устройства события обычно происходят во время транзакции с этим устройством. Таким образом, путь из этой транзакции может быть повторно использован для сообщения о данном событии (это безопасно, потому что путь события копируется в процедуре сообщения о событии, но не освобождается и не передаётся дальше). Также безопасно динамически выделять пути в любое время, включая процедуры обработки прерываний, хотя это влечёт определённые накладные расходы, и возможная проблема такого подхода заключается в том, что в этот момент может не быть свободной памяти. Для события сброса шины нам необходимо определить путь-шаблон, включающий все устройства на шине. Поэтому мы можем заранее создать путь для будущих событий сброса шины и избежать проблем с возможной нехваткой памяти в будущем:

```
    struct cam_path *path;

    if (xpt_create_path(&path, /*periph*/NULL,
                cam_sim_path(sim), CAM_TARGET_WILDCARD,
                CAM_LUN_WILDCARD) != CAM_REQ_CMP) {
        xpt_bus_deregister(cam_sim_path(sim));
        cam_sim_free(sim, /*free_devq*/TRUE);
        error; /* some code to handle the error */
```

```
    }

    softc->wpath = path;
    softc->sim = sim;
```

Как вы можете видеть, путь включает:

- Идентификатор драйвера периферийного устройства (NULL здесь, так как у нас его нет)
- Идентификатор драйвера SIM (`cam_sim_path(sim)`)
- Номер целевого устройства SCSI (CAM_TARGET_WILDCARD означает "все устройства")
- Номер SCSI LUN подустройства (CAM_LUN_WILDCARD означает "все LUN")

Если драйвер не может выделить этот путь, он не сможет нормально работать, поэтому в таком случае мы демонтируем эту шину SCSI.

И мы сохраняем указатель пути в структуре `softc` для дальнейшего использования. После этого сохраняем значение sim (или можем также отбросить его при выходе из `xxx_probe()`, если захотим).

Вот и всё для минималистичной инициализации. Чтобы сделать всё правильно, остался ещё один вопрос.

Для драйвера SIM есть одно особенно важное событие: когда целевое устройство считается потерянным. В этом случае может быть хорошей идеей сбросить SCSI-переговоры с этим устройством. Поэтому мы регистрируем обратный вызов для этого события в CAM. Запрос передаётся в CAM путём запроса действия CAM в блоке управления CAM для этого типа запроса:

```
    struct ccb_setasync csa;

    xpt_setup_ccb(&csa.ccb_h, path, /*priority*/5);
    csa.ccb_h.func_code = XPT_SASYNC_CB;
    csa.event_enable = AC_LOST_DEVICE;
    csa.callback = xxx_async;
    csa.callback_arg = sim;
    xpt_action((union ccb *)&csa);
```

## 12.5. Обработка сообщений CAM: xxx_action

```
static void xxx_action(struct cam_sim *sim, union ccb *ccb);
```

Выполнить некоторое действие по запросу подсистемы CAM. Sim описывает SIM для запроса, CCB — это сам запрос. CCB расшифровывается как "CAM Control Block" (блок управления CAM). Это объединение множества конкретных экземпляров, каждый из которых описывает аргументы для определённого типа транзакций. Все эти экземпляры

имеют общий заголовок CCB, в котором хранится общая часть аргументов.

CAM поддерживает SCSI-контроллеры, работающие как в режиме инициатора («обычном»), так и в режиме цели (эмулирующем SCSI-устройство). Здесь мы рассматриваем только часть, относящуюся к режиму инициатора.

Существует несколько функций и макросов (другими словами, методов), определённых для доступа к публичным данным в структуре sim:

- `cam_sim_path(sim)` - идентификатор пути (см. выше)
- `cam_sim_name(sim)` — имя sim
- `cam_sim_softc(sim)` - указатель на структуру softc (приватные данные драйвера)
- `cam_sim_unit(sim)` - номер устройства
- `cam_sim_bus(sim)` - идентификатор шины

Для идентификации устройства `xxx_action()` может получить номер устройства и указатель на его структуру softc, используя следующие функции.

Тип запроса хранится в `ccb->ccb_h.func_code`. Поэтому, как правило, `xxx_action()` состоит из большого оператора switch:

```
    struct xxx_softc *softc = (struct xxx_softc *) cam_sim_softc(sim);
    struct ccb_hdr *ccb_h = &ccb->ccb_h;
    int unit = cam_sim_unit(sim);
    int bus = cam_sim_bus(sim);

    switch (ccb_h->func_code) {
    case ...:
        ...
    default:
        ccb_h->status = CAM_REQ_INVALID;
        xpt_done(ccb);
        break;
    }
```

Как видно из случая по умолчанию (если получена неизвестная команда) код возврата команды устанавливается в `ccb->ccb_h.status`, а завершённый CCB возвращается обратно в CAM вызовом `xpt_done(ccb)`.

`xpt_done()` не обязательно вызывать из `xxx_action()`: Например, запрос ввода-вывода может быть поставлен в очередь внутри драйвера SIM и/или его SCSI-контроллера. Затем, когда устройство пошлет прерывание, сигнализирующее о завершении обработки этого запроса, `xpt_done()` может быть вызван из процедуры обработки прерывания.

На самом деле, статус CCB не только присваивается в качестве кода возврата, но и CCB всегда имеет какой-то статус. Перед тем как CCB передаётся в процедуру `xxx_action()`, он получает статус CCB_REQ_INPROG, означающий, что запрос находится в процессе выполнения. В /sys/cam/cam.h определено удивительно большое количество значений

статуса, которые должны детально отражать состояние запроса. Что ещё интереснее, статус фактически представляет собой "побитовое ИЛИ" перечисленного значения статуса (младшие 6 бит) и возможных дополнительных флагов (старшие биты). Перечисленные значения будут подробно рассмотрены далее. Их краткое описание можно найти в разделе "Сводка ошибок". Возможные флаги статуса:

- *CAM_DEV_QFRZN* - если драйвер SIM получает серьёзную ошибку (например, устройство не отвечает на выборку или нарушает протокол SCSI) при обработке CCB, он должен заморозить очередь запросов, вызвав `xpt_freeze_simq()`, вернуть другие поставленные в очередь, но ещё не обработанные CCB для этого устройства обратно в очередь CAM, затем установить этот флаг для проблемного CCB и вызвать `xpt_done()`. Этот флаг заставляет подсистему CAM разморозить очередь после обработки ошибки.
- *CAM_AUTOSNS_VALID* - если устройство вернуло состояние ошибки и флаг CAM_DIS_AUTOSENSE не установлен в CCB, драйвер SIM должен автоматически выполнить команду REQUEST SENSE, чтобы извлечь данные sense (расширенную информацию об ошибке) из устройства. Если попытка была успешной, данные sense должны быть сохранены в CCB, а этот флаг установлен.
- *CAM_RELEASE_SIMQ* - аналогично CAM_DEV_QFRZN, но используется в случае возникновения проблем (или нехватки ресурсов) с самим SCSI-контроллером. В этом случае все последующие запросы к контроллеру должны быть остановлены с помощью `xpt_freeze_simq()`. Очередь контроллера будет возобновлена после того, как драйвер SIM устранит нехватку и уведомит CAM, вернув некоторый CCB с установленным этим флагом.
- *CAM_SIM_QUEUED* - этот флаг должен быть установлен, когда SIM помещает CCB в свою очередь запросов (и снят, когда этот CCB извлекается из очереди перед возвратом в CAM). В настоящее время этот флаг нигде не используется в коде CAM, поэтому его назначение чисто диагностическое.
- *CAM_QOS_VALID* - Данные QOS теперь действительны.

Функция `xxx_action()` не может находиться в состоянии ожидания, поэтому вся синхронизация доступа к ресурсам должна выполняться с использованием SIM или заморозки очереди устройств. Помимо упомянутых флагов, подсистема CAM предоставляет функции `xpt_release_simq()` и `xpt_release_devq()` для разморозки очередей напрямую, без передачи CCB в CAM.

Заголовок CCB содержит следующие поля:

- *path* - идентификатор пути для запроса
- *target_id* - идентификатор целевого устройства для запроса
- *target_lun* - идентификатор LUN целевого устройства
- *timeout* - интервал таймаута для этой команды, в миллисекундах
- *timeout_ch* - удобное место для драйвера SIM, чтобы хранить обработчик таймаута (сама подсистема CAM не делает никаких предположений о нём)
- *flags* - различные биты информации о запросе spriv_ptr0, spriv_ptr1 — поля, зарезервированные для приватного использования драйвером SIM (например, для связи

с очередями SIM или приватными блоками управления SIM); фактически они существуют как объединения: spriv_ptr0 и spriv_ptr1 имеют тип (void *), spriv_field0 и spriv_field1 имеют тип unsigned long, sim_priv.entries[0].bytes и sim_priv.entries[1].bytes - это байтовые массивы размера, согласованного с другими вариантами объединения, а sim_priv.bytes - это один массив, вдвое большего размера.

Рекомендуемый способ использования приватных полей SIM в CCB — это определить для них осмысленные имена и использовать эти осмысленные имена в драйвере, например:

```
#define ccb_some_meaningful_name    sim_priv.entries[0].bytes
#define ccb_hcb spriv_ptr1 /* for hardware control block */
```

Наиболее распространённые запросы в режиме инициатора:

### 12.5.1. *XPT_SCSI_IO* — выполнить транзакцию ввода-вывода

Экземпляр "struct ccb_scsiio csio" объединения ccb используется для передачи аргументов. Они включают:

- *cdb_io* - указатель на буфер команды SCSI или сам буфер
- *cdb_len* - длина команды SCSI
- *data_ptr* - указатель на буфер данных (усложняется, если используется scatter/gather)
- *dxfer_len* - длина передаваемых данных
- *sglist_cnt* - счетчик сегментов scatter/gather
- *scsi_status* - место для возврата статуса SCSI
- *sense_data* - буфер для информации SCSI sense, если команда возвращает ошибку (драйвер SIM должен автоматически выполнить команду REQUEST SENSE в этом случае, если флаг CCB CAM_DIS_AUTOSENSE не установлен)
- *sense_len* - длина этого буфера (если она окажется больше размера sense_data, драйвер SIM должен без уведомления принять меньшее значение)
- *resid, sense_resid* — если передача данных или SCSI sense вернула ошибку, это счётчики остаточных (не переданных) данных. Они не кажутся особенно значимыми, поэтому в случаях, когда их сложно вычислить (например, подсчёт байтов в FIFO-буфере SCSI-контроллера), подойдёт и приблизительное значение. Для успешно завершённой передачи они должны быть установлены в ноль.
- *tag_action* - тип используемого тега:
  - `CAM_TAG_ACTION_NONE` - не использовать теги для данной транзакции
  - MSG_SIMPLE_Q_TAG, MSG_HEAD_OF_Q_TAG, MSG_ORDERED_Q_TAG — значение, соответствующее указанному теговому сообщению (см. /sys/cam/scsi/scsi_message.h); указывает только тип тега, значение тега должно быть назначено самим драйвером SIM

Общая логика обработки этого запроса следующая:

Первое, что нужно сделать, это проверить возможные состояния гонки, чтобы убедиться, что команда не была прервана, пока находилась в очереди:

```
    struct ccb_scsiio *csio = &ccb->csio;

    if ((ccb_h->status & CAM_STATUS_MASK) != CAM_REQ_INPROG) {
        xpt_done(ccb);
        return;
    }
```

Также мы проверяем, что устройство вообще поддерживается нашим контроллером:

```
    if (ccb_h->target_id > OUR_MAX_SUPPORTED_TARGET_ID
    || cch_h->target_id == OUR_SCSI_CONTROLLERS_OWN_ID) {
        ccb_h->status = CAM_TID_INVALID;
        xpt_done(ccb);
        return;
    }
    if (ccb_h->target_lun > OUR_MAX_SUPPORTED_LUN) {
        ccb_h->status = CAM_LUN_INVALID;
        xpt_done(ccb);
        return;
    }
```

Затем выделяем все необходимые структуры данных (такие как зависящий от карты блок управления оборудованием), которые нам нужны для обработки этого запроса. Если мы не можем этого сделать, то замораживаем очередь SIM и запоминаем, что у нас есть отложенная операция, возвращаем CCB обратно и просим CAM поставить его в очередь снова. Позже, когда ресурсы станут доступны, очередь SIM должна быть разморожена путём возврата CCB с установленным битом `CAM_SIMQ_RELEASE` в его статусе. В противном случае, если всё прошло успешно, связываем CCB с блоком управления оборудованием (HCB) и помечаем его как поставленный в очередь.

```
    struct xxx_hcb *hcb = allocate_hcb(softc, unit, bus);

    if (hcb == NULL) {
        softc->flags |= RESOURCE_SHORTAGE;
        xpt_freeze_simq(sim, /*count*/1);
        ccb_h->status = CAM_REQUEUE_REQ;
        xpt_done(ccb);
        return;
    }

    hcb->ccb = ccb; ccb_h->ccb_hcb = (void *)hcb;
    ccb_h->status |= CAM_SIM_QUEUED;
```

Извлечь целевые данные из CCB в аппаратный блок управления. Проверить, запрошено ли

назначение тега, и если да, то сгенерировать уникальный тег и построить сообщения тега SCSI. Драйвер SIM также отвечает за согласование с устройствами для установки максимальной взаимно поддерживаемой ширины шины, синхронной скорости и смещения.

```
    hcb->target = ccb_h->target_id; hcb->lun = ccb_h->target_lun;
    generate_identify_message(hcb);
    if (ccb_h->tag_action != CAM_TAG_ACTION_NONE)
        generate_unique_tag_message(hcb, ccb_h->tag_action);
    if (!target_negotiated(hcb))
        generate_negotiation_messages(hcb);
```

Затем настройте команду SCSI. Хранилище команды может быть указано в CCB различными способами, определяемыми флагами CCB. Буфер команды может содержаться в CCB или указываться на него; в последнем случае указатель может быть физическим или виртуальным. Поскольку оборудованию обычно требуется физический адрес, мы всегда преобразуем адрес в физический, как правило, используя API busdma.

В случае, если запрашивается физический адрес, допустимо вернуть CCB со статусом `CAM_REQ_INVALID`, текущие драйверы так и делают. При необходимости физический адрес также может быть преобразован или отображён обратно в виртуальный, но с большими трудностями, поэтому мы этого не делаем.

```
    if (ccb_h->flags & CAM_CDB_POINTER) {
        /* CDB is a pointer */
        if (!(ccb_h->flags & CAM_CDB_PHYS)) {
            /* CDB pointer is virtual */
            hcb->cmd = vtobus(csio->cdb_io.cdb_ptr);
        } else {
            /* CDB pointer is physical */
            hcb->cmd = csio->cdb_io.cdb_ptr ;
        }
    } else {
        /* CDB is in the ccb (buffer) */
        hcb->cmd = vtobus(csio->cdb_io.cdb_bytes);
    }
    hcb->cmdlen = csio->cdb_len;
```

Теперь настало время настроить данные. Опять же, хранилище данных может быть указано в CCB различными интересными способами, определяемыми флагами CCB. Сначала мы получаем направление передачи данных. Самый простой случай — если нет данных для передачи:

```
    int dir = (ccb_h->flags & CAM_DIR_MASK);

    if (dir == CAM_DIR_NONE)
```

```
        goto end_data;
```

Затем мы проверяем, находятся ли данные в одном фрагменте или в списке scatter-gather, а также являются ли адреса физическими или виртуальными. SCSI-контроллер может обрабатывать только ограниченное количество фрагментов ограниченной длины. Если запрос превышает это ограничение, мы возвращаем ошибку. Мы используем специальную функцию для возврата CCB, чтобы в одном месте обрабатывать нехватку ресурсов HCB. Функции для добавления фрагментов зависят от драйвера, и здесь мы оставляем их без детальной реализации. Подробности о проблемах трансляции адресов см. в описании обработки SCSI-команд (CDB). Если какая-то вариация слишком сложна или невозможна для реализации с конкретной картой, допустимо вернуть статус `CAM_REQ_INVALID`. На самом деле, похоже, что возможность scatter-gather нигде в коде CAM сейчас не используется. Но как минимум случай с единичным неразделённым виртуальным буфером должен быть реализован, так как он активно используется CAM.

```
    int rv;

    initialize_hcb_for_data(hcb);

    if ((!(ccb_h->flags & CAM_SCATTER_VALID)) {
        /* single buffer */
        if (!(ccb_h->flags & CAM_DATA_PHYS)) {
            rv = add_virtual_chunk(hcb, csio->data_ptr, csio->dxfer_len, dir);
            }
        } else {
            rv = add_physical_chunk(hcb, csio->data_ptr, csio->dxfer_len, dir);
        }
    } else {
        int i;
        struct bus_dma_segment *segs;
        segs = (struct bus_dma_segment *)csio->data_ptr;

        if ((ccb_h->flags & CAM_SG_LIST_PHYS) != 0) {
            /* The SG list pointer is physical */
            rv = setup_hcb_for_physical_sg_list(hcb, segs, csio->sglist_cnt);
        } else if (!(ccb_h->flags & CAM_DATA_PHYS)) {
            /* SG buffer pointers are virtual */
            for (i = 0; i < csio->sglist_cnt; i++) {
                rv = add_virtual_chunk(hcb, segs[i].ds_addr,
                    segs[i].ds_len, dir);
                if (rv != CAM_REQ_CMP)
                    break;
            }
        } else {
            /* SG buffer pointers are physical */
            for (i = 0; i < csio->sglist_cnt; i++) {
                rv = add_physical_chunk(hcb, segs[i].ds_addr,
                    segs[i].ds_len, dir);
                if (rv != CAM_REQ_CMP)
```

```
                break;
            }
        }
    }
    if (rv != CAM_REQ_CMP) {
        /* we expect that add_*_chunk() functions return CAM_REQ_CMP
         * if they added a chunk successfully, CAM_REQ_TOO_BIG if
         * the request is too big (too many bytes or too many chunks),
         * CAM_REQ_INVALID in case of other troubles
         */
        free_hcb_and_ccb_done(hcb, ccb, rv);
        return;
    }
    end_data:
```

Если отключение запрещено для этого CCB, мы передаем эту информацию в hcb:

```
    if (ccb_h->flags & CAM_DIS_DISCONNECT)
        hcb_disable_disconnect(hcb);
```

Если контроллер способен самостоятельно выполнять команду REQUEST SENSE, то ему также следует передать значение флага CAM_DIS_AUTOSENSE, чтобы предотвратить автоматическое выполнение REQUEST SENSE, если подсистема CAM этого не требует.

Осталось только установить таймаут, передать наш hcb оборудованию и вернуться, остальное будет сделано обработчиком прерывания (или обработчиком таймаута).

```
    ccb_h->timeout_ch = timeout(xxx_timeout, (caddr_t) hcb,
        (ccb_h->timeout * hz) / 1000); /* convert milliseconds to ticks */
    put_hcb_into_hardware_queue(hcb);
    return;
```

И вот возможная реализация функции, возвращающей CCB:

```
    static void
    free_hcb_and_ccb_done(struct xxx_hcb *hcb, union ccb *ccb, u_int32_t status)
    {
        struct xxx_softc *softc = hcb->softc;

        ccb->ccb_h.ccb_hcb = 0;
        if (hcb != NULL) {
            untimeout(xxx_timeout, (caddr_t) hcb, ccb->ccb_h.timeout_ch);
            /* we're about to free a hcb, so the shortage has ended */
            if (softc->flags & RESOURCE_SHORTAGE)  {
                softc->flags &= ~RESOURCE_SHORTAGE;
                status |= CAM_RELEASE_SIMQ;
            }
```

```
            free_hcb(hcb); /* also removes hcb from any internal lists */
        }
        ccb->ccb_h.status = status |
            (ccb->ccb_h.status & ~(CAM_STATUS_MASK|CAM_SIM_QUEUED));
        xpt_done(ccb);
    }
```

### 12.5.2. *XPT_RESET_DEV* — отправить устройству сообщение SCSI "BUS DEVICE RESET"

В CCB не передаются данные, кроме заголовка, и наиболее интересным аргументом в нём является target_id. В зависимости от аппаратного обеспечения контроллера может быть создан аппаратный блок управления (как для запроса XPT_SCSI_IO, см. описание запроса XPT_SCSI_IO) и отправлен контроллеру, или SCSI-контроллер может быть немедленно запрограммирован на отправку этого сообщения RESET устройству, или этот запрос может просто не поддерживаться (и возвращать статус `CAM_REQ_INVALID`). Также при завершении запроса все отключённые транзакции для этого целевого устройства должны быть прерваны (вероятно, в процедуре прерывания).

Кроме того, все текущие переговоры для цели теряются при сбросе, поэтому они также могут быть очищены. Или их очистка может быть отложена, так как в любом случае цель запросит повторные переговоры при следующей транзакции.

### 12.5.3. *XPT_RESET_BUS* — отправить сигнал RESET на шину SCSI

В CCB не передаются аргументы, единственный интересный аргумент — это шина SCSI, указанная структурой sim.

Минималистичная реализация могла бы пропустить SCSI-переговоры для всех устройств на шине и вернуть статус CAM_REQ_CMP.

Правильная реализация дополнительно должна фактически сбросить шину SCSI (возможно, также сбросить контроллер SCSI) и пометить все обрабатываемые CCB, как находящиеся в аппаратной очереди, так и отключённые, как завершённые со статусом CAM_SCSI_BUS_RESET. Например:

```
    int targ, lun;
    struct xxx_hcb *h, *hh;
    struct ccb_trans_settings neg;
    struct cam_path *path;

    /* The SCSI bus reset may take a long time, in this case its completion
     * should be checked by interrupt or timeout. But for simplicity
     * we assume here that it is really fast.
     */
    reset_scsi_bus(softc);

    /* drop all enqueued CCBs */
    for (h = softc->first_queued_hcb; h != NULL; h = hh) {
```

```
        hh = h->next;
        free_hcb_and_ccb_done(h, h->ccb, CAM_SCSI_BUS_RESET);
    }

    /* the clean values of negotiations to report */
    neg.bus_width = 8;
    neg.sync_period = neg.sync_offset = 0;
    neg.valid = (CCB_TRANS_BUS_WIDTH_VALID
        | CCB_TRANS_SYNC_RATE_VALID | CCB_TRANS_SYNC_OFFSET_VALID);

    /* drop all disconnected CCBs and clean negotiations  */
    for (targ=0; targ <= OUR_MAX_SUPPORTED_TARGET; targ++) {
        clean_negotiations(softc, targ);

        /* report the event if possible */
        if (xpt_create_path(&path, /*periph*/NULL,
                cam_sim_path(sim), targ,
                CAM_LUN_WILDCARD) == CAM_REQ_CMP) {
            xpt_async(AC_TRANSFER_NEG, path, &neg);
            xpt_free_path(path);
        }

        for (lun=0; lun <= OUR_MAX_SUPPORTED_LUN; lun++)
            for (h = softc->first_discon_hcb[targ][lun]; h != NULL; h = hh) {
                hh=h->next;
                free_hcb_and_ccb_done(h, h->ccb, CAM_SCSI_BUS_RESET);
            }
    }

    ccb->ccb_h.status = CAM_REQ_CMP;
    xpt_done(ccb);

    /* report the event */
    xpt_async(AC_BUS_RESET, softc->wpath, NULL);
    return;
```

Реализация сброса шины SCSI в виде функции может быть хорошей идеей, так как она может быть повторно использована функцией таймаута в качестве последнего средства, если что-то пойдёт не так.

### 12.5.4. *XPT_ABORT* — прервать указанный CCB

Аргументы передаются в экземпляре "struct ccb_abort cab" объединения ccb. Единственное поле аргумента в нём:

- *abort_ccb* — указатель на CCB, который необходимо прервать

Если прерывание не поддерживается, просто верните статус CAM_UA_ABORT. Это также простой способ минимальной реализации этого вызова — в любом случае возвращать CAM_UA_ABORT.

Трудный путь — честно реализовать этот запрос. Сначала проверьте, что прерывание применяется к SCSI-транзакции:

```
    struct ccb *abort_ccb;
    abort_ccb = ccb->cab.abort_ccb;

    if (abort_ccb->ccb_h.func_code != XPT_SCSI_IO) {
        ccb->ccb_h.status = CAM_UA_ABORT;
        xpt_done(ccb);
        return;
    }
```

Затем необходимо найти этот CCB в нашей очереди. Это можно сделать, пройдясь по списку всех наших блоков управления оборудованием в поисках связанного с этим CCB:

```
    struct xxx_hcb *hcb, *h;

    hcb = NULL;

    /* We assume that softc->first_hcb is the head of the list of all
     * HCBs associated with this bus, including those enqueued for
     * processing, being processed by hardware and disconnected ones.
     */
    for (h = softc->first_hcb; h != NULL; h = h->next) {
        if (h->ccb == abort_ccb) {
            hcb = h;
            break;
        }
    }

    if (hcb == NULL) {
        /* no such CCB in our queue */
        ccb->ccb_h.status = CAM_PATH_INVALID;
        xpt_done(ccb);
        return;
    }

    hcb=found_hcb;
```

Теперь мы рассмотрим текущее состояние обработки HCB. Он может находиться в очереди, ожидая отправки на шину SCSI, передаваться в данный момент, быть отключённым и ожидать результата команды, или фактически завершённым с точки зрения аппаратуры, но ещё не отмеченным программным обеспечением, как выполненный. Чтобы избежать состояний гонки с аппаратурой, мы помечаем HCB как прерванный, так что если этот HCB вот-вот будет отправлен на шину SCSI, контроллер SCSI увидит этот флаг и пропустит его.

```
    int hstatus;
```

```
    /* shown as a function, in case special action is needed to make
     * this flag visible to hardware
     */
    set_hcb_flags(hcb, HCB_BEING_ABORTED);

    abort_again:

    hstatus = get_hcb_status(hcb);
    switch (hstatus) {
    case HCB_SITTING_IN_QUEUE:
        remove_hcb_from_hardware_queue(hcb);
        /* FALLTHROUGH */
    case HCB_COMPLETED:
        /* this is an easy case */
        free_hcb_and_ccb_done(hcb, abort_ccb, CAM_REQ_ABORTED);
        break;
```

Если CCB передаётся в данный момент, мы хотели бы сигнализировать контроллеру SCSI аппаратно-зависимым способом, что хотим прервать текущую передачу. Контроллер SCSI установит сигнал SCSI ATTENTION, и когда целевое устройство ответит на него, отправит сообщение ABORT. Мы также сбрасываем таймаут, чтобы убедиться, что целевое устройство не засыпает навсегда. Если команда не будет прервана в разумное время, например, за 10 секунд, процедура таймаута продолжит работу и сбросит всю шину SCSI. Поскольку команда будет прервана в разумные сроки, мы можем просто вернуть запрос на прерывание как успешно выполненный и пометить прерванный CCB как прерванный (но пока не помечать его как завершённый).

```
    case HCB_BEING_TRANSFERRED:
        untimeout(xxx_timeout, (caddr_t) hcb, abort_ccb->ccb_h.timeout_ch);
        abort_ccb->ccb_h.timeout_ch =
            timeout(xxx_timeout, (caddr_t) hcb, 10 * hz);
        abort_ccb->ccb_h.status = CAM_REQ_ABORTED;
        /* ask the controller to abort that HCB, then generate
         * an interrupt and stop
         */
        if (signal_hardware_to_abort_hcb_and_stop(hcb) < 0) {
            /* oops, we missed the race with hardware, this transaction
             * got off the bus before we aborted it, try again */
            goto abort_again;
        }

        break;
```

Если CCB находится в списке отключённых, то настроить его как запрос прерывания и повторно поставить в начало аппаратной очереди. Сбросить таймаут и сообщить о завершении запроса прерывания.

```
    case HCB_DISCONNECTED:
        untimeout(xxx_timeout, (caddr_t) hcb, abort_ccb->ccb_h.timeout_ch);
        abort_ccb->ccb_h.timeout_ch =
            timeout(xxx_timeout, (caddr_t) hcb, 10 * hz);
        put_abort_message_into_hcb(hcb);
        put_hcb_at_the_front_of_hardware_queue(hcb);
        break;
    }
    ccb->ccb_h.status = CAM_REQ_CMP;
    xpt_done(ccb);
    return;
```

Вот и все, что касается запроса ABORT, хотя есть ещё один момент. Поскольку сообщение ABORT очищает все текущие транзакции на LUN, нам необходимо пометить все остальные активные транзакции на этом LUN как прерванные. Это должно быть выполнено в процедуре аппаратного прерывания после того, как транзакция будет прервана.

Реализация прерывания CCB в виде функции может быть довольно хорошей идеей, эта функция может быть повторно использована, если транзакция ввода-вывода превысит время ожидания. Единственное различие будет в том, что для транзакции с истекшим временем ожидания будет возвращён статус CAM_CMD_TIMEOUT. Тогда код в case XPT_ABORT будет небольшим, например:

```
    case XPT_ABORT:
        struct ccb *abort_ccb;
        abort_ccb = ccb->cab.abort_ccb;

        if (abort_ccb->ccb_h.func_code != XPT_SCSI_IO) {
            ccb->ccb_h.status = CAM_UA_ABORT;
            xpt_done(ccb);
            return;
        }
        if (xxx_abort_ccb(abort_ccb, CAM_REQ_ABORTED) < 0)
            /* no such CCB in our queue */
            ccb->ccb_h.status = CAM_PATH_INVALID;
        else
            ccb->ccb_h.status = CAM_REQ_CMP;
        xpt_done(ccb);
        return;
```

### 12.5.5. *XPT_SET_TRAN_SETTINGS* — явно установить значения настроек передачи SCSI

Аргументы передаются в экземпляре "struct ccb_trans_setting cts" объединения ccb:

- *valid* - битовая маска, показывающая, какие настройки должны быть обновлены:
  - *CCB_TRANS_SYNC_RATE_VALID* - скорость синхронной передачи

    - *CCB_TRANS_SYNC_OFFSET_VALID* - синхронное смещение
    - *CCB_TRANS_BUS_WIDTH_VALID* - ширина шины
    - *CCB_TRANS_DISC_VALID* - установить разрешение/запрет отключения
    - *CCB_TRANS_TQ_VALID* - установить разрешение/запрет очередей с тегами
- *flags* - состоит из двух частей: бинарных аргументов и идентификации подопераций. Бинарные аргументы:
    - *CCB_TRANS_DISC_ENB* - разрешить отключение
    - *CCB_TRANS_TAG_ENB* - разрешить тегированную очередь
- подоперации:
    - *CCB_TRANS_CURRENT_SETTINGS* - изменить текущие параметры согласования
    - *CCB_TRANS_USER_SETTINGS* - сохранять желаемые пользовательские значения sync_period, sync_offset - самоочевидные параметры; если sync_offset==0, то запрашивается асинхронный режим bus_width - ширина шины в битах (не в байтах)

Поддерживаются два набора согласованных параметров: пользовательские настройки и текущие настройки. Пользовательские настройки не так часто используются в драйверах SIM, это в основном просто область памяти, где верхние уровни могут сохранять (и позже извлекать) свои представления о параметрах. Установка пользовательских параметров не вызывает повторного согласования скоростей передачи. Однако, когда SCSI-контроллер выполняет согласование, он никогда не должен устанавливать значения выше пользовательских параметров, так что они по сути являются верхней границей.

Текущие настройки, как следует из названия, являются текущими. Их изменение означает, что параметры должны быть повторно согласованы при следующей передаче. Опять же, эти «новые текущие настройки» не предназначены для принудительного применения к устройству, они лишь используются в качестве начального шага переговоров. Кроме того, они должны быть ограничены реальными возможностями SCSI-контроллера: например, если SCSI-контроллер имеет 8-битную шину, а запрос требует установки 16-битных передач, этот параметр должен быть тихо усечён до 8-битных передач перед отправкой на устройство.

Один нюанс заключается в том, что ширина шины и синхронные параметры относятся к цели, тогда как параметры отключения и включения тегов относятся к логическому устройству LUN.

Рекомендуемая реализация заключается в хранении 3 наборов согласованных параметров (ширина шины и синхронная передача):

- *user* - пользовательский набор, как указано выше
- *current* - тот, который фактически действуют
- *goal* - тот набор, который запрошен для установки параметров в качестве "текущих"

Код выглядит следующим образом:

```
    struct ccb_trans_settings *cts;
    int targ, lun;
    int flags;

    cts = &ccb->cts;
    targ = ccb_h->target_id;
    lun = ccb_h->target_lun;
    flags = cts->flags;
    if (flags & CCB_TRANS_USER_SETTINGS) {
        if (flags & CCB_TRANS_SYNC_RATE_VALID)
            softc->user_sync_period[targ] = cts->sync_period;
        if (flags & CCB_TRANS_SYNC_OFFSET_VALID)
            softc->user_sync_offset[targ] = cts->sync_offset;
        if (flags & CCB_TRANS_BUS_WIDTH_VALID)
            softc->user_bus_width[targ] = cts->bus_width;

        if (flags & CCB_TRANS_DISC_VALID) {
            softc->user_tflags[targ][lun] &= ~CCB_TRANS_DISC_ENB;
            softc->user_tflags[targ][lun] |= flags & CCB_TRANS_DISC_ENB;
        }
        if (flags & CCB_TRANS_TQ_VALID) {
            softc->user_tflags[targ][lun] &= ~CCB_TRANS_TQ_ENB;
            softc->user_tflags[targ][lun] |= flags & CCB_TRANS_TQ_ENB;
        }
    }
    if (flags & CCB_TRANS_CURRENT_SETTINGS) {
        if (flags & CCB_TRANS_SYNC_RATE_VALID)
            softc->goal_sync_period[targ] =
                max(cts->sync_period, OUR_MIN_SUPPORTED_PERIOD);
        if (flags & CCB_TRANS_SYNC_OFFSET_VALID)
            softc->goal_sync_offset[targ] =
                min(cts->sync_offset, OUR_MAX_SUPPORTED_OFFSET);
        if (flags & CCB_TRANS_BUS_WIDTH_VALID)
            softc->goal_bus_width[targ] = min(cts->bus_width, OUR_BUS_WIDTH);

        if (flags & CCB_TRANS_DISC_VALID) {
            softc->current_tflags[targ][lun] &= ~CCB_TRANS_DISC_ENB;
            softc->current_tflags[targ][lun] |= flags & CCB_TRANS_DISC_ENB;
        }
        if (flags & CCB_TRANS_TQ_VALID) {
            softc->current_tflags[targ][lun] &= ~CCB_TRANS_TQ_ENB;
            softc->current_tflags[targ][lun] |= flags & CCB_TRANS_TQ_ENB;
        }
    }
    ccb->ccb_h.status = CAM_REQ_CMP;
    xpt_done(ccb);
    return;
```

Затем, когда следующий запрос ввода-вывода будет обработан, он проверит, нужно ли повторное согласование, например, вызовом функции target_negotiated(hcb). Это может

быть реализовано следующим образом:

```
int
target_negotiated(struct xxx_hcb *hcb)
{
    struct softc *softc = hcb->softc;
    int targ = hcb->targ;

    if (softc->current_sync_period[targ] != softc->goal_sync_period[targ]
    || softc->current_sync_offset[targ] != softc->goal_sync_offset[targ]
    || softc->current_bus_width[targ] != softc->goal_bus_width[targ])
        return 0; /* FALSE */
    else
        return 1; /* TRUE */
}
```

После пересогласования значений полученные значения должны быть присвоены как текущим, так и целевым параметрам, чтобы для будущих операций ввода-вывода текущие и целевые параметры совпадали, и функция `target_negotiated()` возвращала TRUE. При инициализации карты (в `xxx_attach()`) текущие параметры согласования должны быть инициализированы узким асинхронным режимом, а целевые и текущие значения должны быть инициализированы максимальными значениями, поддерживаемыми контроллером.

### 12.5.6. *XPT_GET_TRAN_SETTINGS* — получить значения настроек передачи SCSI

Эта операция является обратной XPT_SET_TRAN_SETTINGS. Заполните экземпляр CCB "struct ccb_trans_setting cts" данными, запрошенными флагами CCB_TRANS_CURRENT_SETTINGS или CCB_TRANS_USER_SETTINGS (если установлены оба, существующие драйверы возвращают текущие настройки). Установите все биты в поле valid.

### 12.5.7. *XPT_CALC_GEOMETRY* — вычислить логическую (BIOS) геометрию диска

Аргументы передаются в экземпляре "struct ccb_calc_geometry ccg" объединения ccb:

- *block_size* - вход, размер блока (также известный как сектор) в байтах
- *volume_size* - вход, размер тома в байтах
- *cylinders* - выход, логические цилиндры
- *heads* - выход, логические головки
- *secs_per_track* - выход, логических секторов на дорожку

Если возвращённая геометрия значительно отличается от той, которую предполагает BIOS SCSI-контроллера, и диск на этом SCSI-контроллере используется как загрузочный, система может не загрузиться. Типичный пример расчёта, взятый из драйвера `aic7xxx`, выглядит следующим образом:

```
    struct    ccb_calc_geometry *ccg;
    u_int32_t size_mb;
    u_int32_t secs_per_cylinder;
    int   extended;

    ccg = &ccb->ccg;
    size_mb = ccg->volume_size
        / ((1024L * 1024L) / ccg->block_size);
    extended = check_cards_EEPROM_for_extended_geometry(softc);

    if (size_mb > 1024 && extended) {
        ccg->heads = 255;
        ccg->secs_per_track = 63;
    } else {
        ccg->heads = 64;
        ccg->secs_per_track = 32;
    }
    secs_per_cylinder = ccg->heads * ccg->secs_per_track;
    ccg->cylinders = ccg->volume_size / secs_per_cylinder;
    ccb->ccb_h.status = CAM_REQ_CMP;
    xpt_done(ccb);
    return;
```

Это даёт общее представление, точный расчет зависит от особенностей конкретной BIOS. Если BIOS не предоставляет возможности установить флаг "расширенной трансляции" в EEPROM, этот флаг обычно следует считать равным 1. Другие популярные геометрии:

```
    128 heads, 63 sectors - Symbios controllers
    16 heads, 63 sectors - old controllers
```

Некоторые системные BIOS и SCSI BIOS конфликтуют друг с другом с переменным успехом. Например, комбинация Symbios 875/895 SCSI и Phoenix BIOS может выдавать геометрию 128/63 после включения питания и 255/63 после жёсткого сброса или мягкой перезагрузки.

### 12.5.8. *XPT_PATH_INQ* — запрос пути, другими словами, получение свойств драйвера SIM и контроллера SCSI (также известного как HBA - Host Bus Adapter)

Свойства возвращаются в экземпляре "struct ccb_pathinq cpi" объединения ccb:

- `version_num` - номер версии драйвера SIM, в настоящее время все драйверы используют 1
- hba_inquiry - битовая маска функций, поддерживаемых контроллером:
  - PI_MDP_ABLE - поддерживает сообщение MDP (что-то из SCSI3?)
  - PI_WIDE_32 — поддерживает 32-битную широкую SCSI
  - PI_WIDE_16 — поддерживает 16-битную широкую SCSI

    - PI_SDTR_ABLE - может согласовать синхронную скорость передачи
    - PI_LINKED_CDB - поддерживает связанные команды
    - PI_TAG_ABLE - поддерживает помеченные команды
    - PI_SOFT_RST — поддерживает альтернативу мягкого сброса (жёсткий сброс и мягкий сброс являются взаимоисключающими в пределах шины SCSI)
- target_sprt - флаги поддержки целевого режима, 0 если не поддерживается
- hba_misc - различные функции контроллера:
    - PIM_SCANHILO - сканирование шины от высокого ID к низкому ID
    - PIM_NOREMOVE - съемные устройства не включены в сканирование
    - PIM_NOINITIATOR - роль инициатора не поддерживается
    - PIM_NOBUSRESET - пользователь отключил начальный BUS RESET
- hba_eng_cnt - загадочное количество движков HBA, что-то связанное со сжатием, в настоящее время всегда устанавливается в 0
- vuhba_flags - уникальные флаги производителя, в настоящее время не используются
- max_target - максимальный поддерживаемый идентификатор целевого устройства (7 для 8-битной шины, 15 для 16-битной шины, 127 для Fibre Channel)
- max_lun - максимально поддерживаемый идентификатор LUN (7 для старых SCSI-контроллеров, 63 для новых)
- async_flags - битовая маска установленных обработчиков Async, в настоящее время не используется
- hpath_id - наивысший Path ID в подсистеме, в настоящее время не используется
- unit_number - номер контроллера, cam_sim_unit(sim)
- bus_id - номер шины, cam_sim_bus(sim)
- initiator_id - SCSI ID самого контроллера
- base_transfer_speed - номинальная скорость передачи в КБ/с для асинхронных узкополосных передач, равна 3300 для SCSI
- sim_vid - идентификатор производителя драйвера SIM, строка с нулевым окончанием максимальной длины SIM_IDLEN, включая завершающий ноль
- hba_vid - идентификатор производителя SCSI-контроллера, строка с нулевым окончанием максимальной длины HBA_IDLEN, включая завершающий ноль
- dev_name - имя драйвера устройства, строка с нулевым окончанием максимальной длины DEV_IDLEN, включая завершающий ноль, эквивалентно cam_sim_name(sim)

Рекомендуемый способ установки строковых полей — использование strncpy, например:

```
    strncpy(cpi->dev_name, cam_sim_name(sim), DEV_IDLEN);
```

После установки значений установите статус в CAM_REQ_CMP и пометьте CCB как завершённый.

## 12.6. Опрос xxx_poll

```
static void xxx_poll(struct cam_sim *);
```

Функция poll используется для имитации прерываний, когда подсистема прерываний не функционирует (например, когда система аварийно завершила работу и создаёт дамп памяти). Подсистема CAM устанавливает соответствующий уровень прерывания перед вызовом процедуры poll. Таким образом, все, что ей нужно сделать, — это вызвать процедуру прерывания (или наоборот, процедура poll может выполнять реальные действия, а процедура прерывания просто вызывает процедуру poll). Зачем тогда нужна отдельная функция? Это связано с различными соглашениями о вызовах. Процедура `xxx_poll` получает указатель на структуру cam_sim в качестве аргумента, в то время как процедура прерывания PCI по общему соглашению получает указатель на структуру `xxx_softc`, а процедура прерывания ISA получает только номер устройства. Таким образом, процедура poll обычно выглядит следующим образом:

```
static void
xxx_poll(struct cam_sim *sim)
{
    xxx_intr((struct xxx_softc *)cam_sim_softc(sim)); /* for PCI device */
}
```

или

```
static void
xxx_poll(struct cam_sim *sim)
{
    xxx_intr(cam_sim_unit(sim)); /* for ISA device */
}
```

## 12.7. Асинхронные события

Если была настроена асинхронная callback-функция для события, то callback-функция должна быть определена.

```
static void
ahc_async(void *callback_arg, u_int32_t code, struct cam_path *path, void *arg)
```

- callback_arg - значение, переданное при регистрации callback
- code - определяет тип события
- path - определяет устройства, к которым применяется событие
- arg - аргумент, специфичный для события

Реализация для одного типа события, AC_LOST_DEVICE, выглядит следующим образом:

```
    struct xxx_softc *softc;
    struct cam_sim *sim;
    int targ;
    struct ccb_trans_settings neg;

    sim = (struct cam_sim *)callback_arg;
    softc = (struct xxx_softc *)cam_sim_softc(sim);
    switch (code) {
    case AC_LOST_DEVICE:
        targ = xpt_path_target_id(path);
        if (targ <= OUR_MAX_SUPPORTED_TARGET) {
            clean_negotiations(softc, targ);
            /* send indication to CAM */
            neg.bus_width = 8;
            neg.sync_period = neg.sync_offset = 0;
            neg.valid = (CCB_TRANS_BUS_WIDTH_VALID
                | CCB_TRANS_SYNC_RATE_VALID | CCB_TRANS_SYNC_OFFSET_VALID);
            xpt_async(AC_TRANSFER_NEG, path, &neg);
        }
        break;
    default:
        break;
    }
```

## 12.8. Прерывания

Точный тип процедуры прерывания зависит от типа периферийной шины (PCI, ISA и так далее), к которой подключен SCSI-контроллер.

Прерывания в драйверах SIM выполняются на уровне прерывания splcam. Поэтому в драйвере следует использовать `splcam()` для синхронизации между обработчиком прерывания и остальной частью драйвера (для драйверов, учитывающих многопроцессорность, ситуация становится ещё сложнее, но здесь мы этот случай не рассматриваем). Псевдокод в этом документе беззаботно игнорирует проблемы синхронизации. Реальный код так делать не должен. Простейший подход — установить `splcam()` при входе в другие функции и сбросить при выходе, защищая их одной большой критической секцией. Чтобы гарантировать восстановление уровня прерывания, можно определить обёрточную функцию, например:

```
    static void
    xxx_action(struct cam_sim *sim, union ccb *ccb)
    {
        int s;
        s = splcam();
        xxx_action1(sim, ccb);
        splx(s);
```

```
}

static void
xxx_action1(struct cam_sim *sim, union ccb *ccb)
{
    ... process the request ...
}
```

Этот подход прост и надежен, но проблема в том, что прерывания могут блокироваться на относительно долгое время, что негативно скажется на производительности системы. С другой стороны, функции семейства `spl()` имеют довольно высокие накладные расходы, поэтому большое количество мелких критических секций также может быть нежелательным.

Условия, обрабатываемые процедурой прерывания, и детали сильно зависят от оборудования. Мы рассматриваем набор "типичных" условий.

Сначала проверяем, было ли на шине событие SCSI сброса (вероятно, вызванное другим SCSI-контроллером на той же SCSI-шине). Если это так, мы отменяем все поставленные в очередь и отключённые запросы, сообщаем о событиях и повторно инициализируем наш SCSI-контроллер. Важно, чтобы во время этой инициализации контроллер не инициировал ещё один сброс, иначе два контроллера на одной SCSI-шине могут бесконечно обмениваться сбросами. Случай фатальной ошибки/зависания контроллера может быть обработан в том же месте, но, вероятно, также потребуется отправка сигнала RESET на SCSI-шину для сброса состояния соединений с SCSI-устройствами.

```
int fatal=0;
struct ccb_trans_settings neg;
struct cam_path *path;

if (detected_scsi_reset(softc)
|| (fatal = detected_fatal_controller_error(softc))) {
    int targ, lun;
    struct xxx_hcb *h, *hh;

    /* drop all enqueued CCBs */
    for(h = softc->first_queued_hcb; h != NULL; h = hh) {
        hh = h->next;
        free_hcb_and_ccb_done(h, h->ccb, CAM_SCSI_BUS_RESET);
    }

    /* the clean values of negotiations to report */
    neg.bus_width = 8;
    neg.sync_period = neg.sync_offset = 0;
    neg.valid = (CCB_TRANS_BUS_WIDTH_VALID
        | CCB_TRANS_SYNC_RATE_VALID | CCB_TRANS_SYNC_OFFSET_VALID);

    /* drop all disconnected CCBs and clean negotiations  */
    for (targ=0; targ <= OUR_MAX_SUPPORTED_TARGET; targ++) {
```

```
            clean_negotiations(softc, targ);

            /* report the event if possible */
            if (xpt_create_path(&path, /*periph*/NULL,
                    cam_sim_path(sim), targ,
                    CAM_LUN_WILDCARD) == CAM_REQ_CMP) {
                xpt_async(AC_TRANSFER_NEG, path, &neg);
                xpt_free_path(path);
            }

            for (lun=0; lun <= OUR_MAX_SUPPORTED_LUN; lun++)
                for (h = softc->first_discon_hcb[targ][lun]; h != NULL; h = hh) {
                    hh=h->next;
                    if (fatal)
                        free_hcb_and_ccb_done(h, h->ccb, CAM_UNREC_HBA_ERROR);
                    else
                        free_hcb_and_ccb_done(h, h->ccb, CAM_SCSI_BUS_RESET);
                }
        }

        /* report the event */
        xpt_async(AC_BUS_RESET, softc->wpath, NULL);

        /* re-initialization may take a lot of time, in such case
         * its completion should be signaled by another interrupt or
         * checked on timeout - but for simplicity we assume here that
         * it is really fast
         */
        if (!fatal) {
            reinitialize_controller_without_scsi_reset(softc);
        } else {
            reinitialize_controller_with_scsi_reset(softc);
        }
        schedule_next_hcb(softc);
        return;
    }
```

Если прерывание не вызвано условием, общим для всего контроллера, то, вероятно, что-то произошло с текущим блоком управления аппаратным обеспечением. В зависимости от оборудования могут быть и другие события, не связанные с HCB, но мы их здесь не рассматриваем. Затем мы анализируем, что произошло с этим HCB:

```
    struct xxx_hcb *hcb, *h, *hh;
    int hcb_status, scsi_status;
    int ccb_status;
    int targ;
    int lun_to_freeze;

    hcb = get_current_hcb(softc);
    if (hcb == NULL) {
```

```
        /* either stray interrupt or something went very wrong
         * or this is something hardware-dependent
         */
        handle as necessary;
        return;
    }

    targ = hcb->target;
    hcb_status = get_status_of_current_hcb(softc);
```

Сначала мы проверяем, завершился ли HCB, и если да, то проверяем возвращённый статус SCSI.

```
    if (hcb_status == COMPLETED) {
        scsi_status = get_completion_status(hcb);
```

Затем проверьте, связан ли этот статус с командой REQUEST SENSE, и если да, обработайте его простым способом.

```
        if (hcb->flags & DOING_AUTOSENSE) {
            if (scsi_status == GOOD) { /* autosense was successful */
                hcb->ccb->ccb_h.status |= CAM_AUTOSNS_VALID;
                free_hcb_and_ccb_done(hcb, hcb->ccb, CAM_SCSI_STATUS_ERROR);
            } else {
        autosense_failed:
                free_hcb_and_ccb_done(hcb, hcb->ccb, CAM_AUTOSENSE_FAIL);
            }
            schedule_next_hcb(softc);
            return;
        }
```

Иначе сама команда завершена, уделяйте больше внимания деталям. Если автоопределение не отключено для этого CCB и команда завершилась неудачно с данными состояния, выполните команду REQUEST SENSE для получения этих данных.

```
        hcb->ccb->csio.scsi_status = scsi_status;
        calculate_residue(hcb);

        if ((hcb->ccb->ccb_h.flags & CAM_DIS_AUTOSENSE)==0
        && (scsi_status == CHECK_CONDITION
                || scsi_status == COMMAND_TERMINATED)) {
            /* start auto-SENSE */
            hcb->flags |= DOING_AUTOSENSE;
            setup_autosense_command_in_hcb(hcb);
            restart_current_hcb(softc);
            return;
        }
```

```
        if (scsi_status == GOOD)
            free_hcb_and_ccb_done(hcb, hcb->ccb, CAM_REQ_CMP);
        else
            free_hcb_and_ccb_done(hcb, hcb->ccb, CAM_SCSI_STATUS_ERROR);
        schedule_next_hcb(softc);
        return;
    }
```

Типичным примером могут быть события согласования: сообщения согласования, полученные от цели SCSI (в ответ на нашу попытку согласования или по инициативе цели), или если цель не может согласовать (отклоняет наши сообщения согласования или не отвечает на них).

```
    switch (hcb_status) {
    case TARGET_REJECTED_WIDE_NEG:
        /* revert to 8-bit bus */
        softc->current_bus_width[targ] = softc->goal_bus_width[targ] = 8;
        /* report the event */
        neg.bus_width = 8;
        neg.valid = CCB_TRANS_BUS_WIDTH_VALID;
        xpt_async(AC_TRANSFER_NEG, hcb->ccb.ccb_h.path_id, &neg);
        continue_current_hcb(softc);
        return;
    case TARGET_ANSWERED_WIDE_NEG:
        {
            int wd;

            wd = get_target_bus_width_request(softc);
            if (wd <= softc->goal_bus_width[targ]) {
                /* answer is acceptable */
                softc->current_bus_width[targ] =
                softc->goal_bus_width[targ] = neg.bus_width = wd;

                /* report the event */
                neg.valid = CCB_TRANS_BUS_WIDTH_VALID;
                xpt_async(AC_TRANSFER_NEG, hcb->ccb.ccb_h.path_id, &neg);
            } else {
                prepare_reject_message(hcb);
            }
        }
        continue_current_hcb(softc);
        return;
    case TARGET_REQUESTED_WIDE_NEG:
        {
            int wd;

            wd = get_target_bus_width_request(softc);
            wd = min (wd, OUR_BUS_WIDTH);
            wd = min (wd, softc->user_bus_width[targ]);
```

```
            if (wd != softc->current_bus_width[targ]) {
                /* the bus width has changed */
                softc->current_bus_width[targ] =
                softc->goal_bus_width[targ] = neg.bus_width = wd;

                /* report the event */
                neg.valid = CCB_TRANS_BUS_WIDTH_VALID;
                xpt_async(AC_TRANSFER_NEG, hcb->ccb.ccb_h.path_id, &neg);
            }
            prepare_width_nego_rsponse(hcb, wd);
        }
        continue_current_hcb(softc);
        return;
    }
```

Затем мы обрабатываем любые ошибки, которые могли произойти во время автоопределения, тем же простым способом, что и раньше. В противном случае мы снова внимательно изучаем детали.

```
    if (hcb->flags & DOING_AUTOSENSE)
        goto autosense_failed;

    switch (hcb_status) {
```

Следующее событие, которое мы рассматриваем, — это неожиданное отключение. Оно считается нормальным после сообщения ABORT или BUS DEVICE RESET и аномальным в остальных случаях.

```
    case UNEXPECTED_DISCONNECT:
        if (requested_abort(hcb)) {
            /* abort affects all commands on that target+LUN, so
             * mark all disconnected HCBs on that target+LUN as aborted too
             */
            for (h = softc->first_discon_hcb[hcb->target][hcb->lun];
                    h != NULL; h = hh) {
                hh=h->next;
                free_hcb_and_ccb_done(h, h->ccb, CAM_REQ_ABORTED);
            }
            ccb_status = CAM_REQ_ABORTED;
        } else if (requested_bus_device_reset(hcb)) {
            int lun;

            /* reset affects all commands on that target, so
             * mark all disconnected HCBs on that target+LUN as reset
             */

            for (lun=0; lun <= OUR_MAX_SUPPORTED_LUN; lun++)
                for (h = softc->first_discon_hcb[hcb->target][lun];
```

```
                    h != NULL; h = hh) {
                hh=h->next;
                free_hcb_and_ccb_done(h, h->ccb, CAM_SCSI_BUS_RESET);
            }

        /* send event */
        xpt_async(AC_SENT_BDR, hcb->ccb->ccb_h.path_id, NULL);

        /* this was the CAM_RESET_DEV request itself, it is completed */
        ccb_status = CAM_REQ_CMP;
    } else {
        calculate_residue(hcb);
        ccb_status = CAM_UNEXP_BUSFREE;
        /* request the further code to freeze the queue */
        hcb->ccb->ccb_h.status |= CAM_DEV_QFRZN;
        lun_to_freeze = hcb->lun;
    }
    break;
```

Если цель отказывается принимать теги, мы уведомляем CAM об этом и возвращаем все команды для этого LUN:

```
case TAGS_REJECTED:
    /* report the event */
    neg.flags = 0 & ~CCB_TRANS_TAG_ENB;
    neg.valid = CCB_TRANS_TQ_VALID;
    xpt_async(AC_TRANSFER_NEG, hcb->ccb.ccb_h.path_id, &neg);

    ccb_status = CAM_MSG_REJECT_REC;
    /* request the further code to freeze the queue */
    hcb->ccb->ccb_h.status |= CAM_DEV_QFRZN;
    lun_to_freeze = hcb->lun;
    break;
```

Затем мы проверяем ряд других условий, при этом обработка в основном ограничивается установкой статуса CCB:

```
case SELECTION_TIMEOUT:
    ccb_status = CAM_SEL_TIMEOUT;
    /* request the further code to freeze the queue */
    hcb->ccb->ccb_h.status |= CAM_DEV_QFRZN;
    lun_to_freeze = CAM_LUN_WILDCARD;
    break;
case PARITY_ERROR:
    ccb_status = CAM_UNCOR_PARITY;
    break;
case DATA_OVERRUN:
case ODD_WIDE_TRANSFER:
```

```
        ccb_status = CAM_DATA_RUN_ERR;
        break;
    default:
        /* all other errors are handled in a generic way */
        ccb_status = CAM_REQ_CMP_ERR;
        /* request the further code to freeze the queue */
        hcb->ccb->ccb_h.status |= CAM_DEV_QFRZN;
        lun_to_freeze = CAM_LUN_WILDCARD;
        break;
    }
```

Затем мы проверяем, была ли ошибка достаточно серьёзной, чтобы заморозить очередь ввода до её обработки, и если да, то делаем это:

```
    if (hcb->ccb->ccb_h.status & CAM_DEV_QFRZN) {
        /* freeze the queue */
        xpt_freeze_devq(ccb->ccb_h.path, /*count*/1);

        /* re-queue all commands for this target/LUN back to CAM */

        for (h = softc->first_queued_hcb; h != NULL; h = hh) {
            hh = h->next;

            if (targ == h->targ
            && (lun_to_freeze == CAM_LUN_WILDCARD || lun_to_freeze == h->lun))
                free_hcb_and_ccb_done(h, h->ccb, CAM_REQUEUE_REQ);
        }
    }
    free_hcb_and_ccb_done(hcb, hcb->ccb, ccb_status);
    schedule_next_hcb(softc);
    return;
```

На этом общее описание обработки прерываний завершается, хотя для некоторых контроллеров могут потребоваться дополнительные действия.

## 12.9. Сводка ошибок

При выполнении запроса ввода-вывода может произойти множество ошибок. Причина ошибки может быть указана в статусе CCB с большим количеством деталей. Примеры использования разбросаны по всему документу. Для полноты изложения приведём сводку рекомендуемых действий при типичных ошибках:

- *CAM_RESRC_UNAVAIL* — некоторый ресурс временно недоступен, и драйвер SIM не может сгенерировать событие, когда он станет доступен. Примером такого ресурса может быть некоторый внутренний аппаратный ресурс контроллера, для которого контроллер не генерирует прерывание при его доступности.
- *CAM_UNCOR_PARITY* - произошла неисправимая ошибка чётности

- *CAM_DATA_RUN_ERR* - переполнение данных или неожиданная фаза данных (направление передачи не соответствует указанному в CAM_DIR_MASK) или нечётная длина передачи для широкой передачи
- *CAM_SEL_TIMEOUT* - произошел таймаут выбора (цель не отвечает)
- *CAM_CMD_TIMEOUT* - произошло превышение времени ожидания команды (сработала функция таймаута)
- *CAM_SCSI_STATUS_ERROR* - устройство вернуло ошибку
- *CAM_AUTOSENSE_FAIL* - устройство вернуло ошибку и команда REQUEST SENSE завершилась неудачно
- *CAM_MSG_REJECT_REC* - получено сообщение MESSAGE REJECT
- *CAM_SCSI_BUS_RESET* - получен сброс шины SCSI
- *CAM_REQ_CMP_ERR* - произошла «невозможная» фаза SCSI или что-то столь же странное, либо это просто общая ошибка, если дополнительная информация недоступна
- *CAM_UNEXP_BUSFREE* - произошло неожиданное отключение
- *CAM_BDR_SENT* - Сообщение BUS DEVICE RESET было отправлено целевому устройству
- *CAM_UNREC_HBA_ERROR* - невосстановимая ошибка адаптера шины хоста
- *CAM_REQ_TOO_BIG* - запрос слишком велик для данного контроллера
- *CAM_REQUEUE_REQ* - этот запрос должен быть повторно поставлен в очередь для сохранения порядка транзакций. Обычно это происходит, когда SIM обнаруживает ошибку, которая должна заморозить очередь, и необходимо поместить другие запросы в очереди для цели на уровне SIM обратно в очередь XPT. Типичными случаями таких ошибок являются тайм-ауты выбора, тайм-ауты команд и другие подобные условия. В таких случаях проблемная команда возвращает статус, указывающий на ошибку, а другие команды, которые ещё не были отправлены на шину, повторно ставятся в очередь.
- *CAM_LUN_INVALID* - идентификатор LUN в запросе не поддерживается контроллером SCSI
- *CAM_TID_INVALID* - идентификатор целевого устройства в запросе не поддерживается контроллером SCSI

# 12.10. Обработка таймаутов

Когда время ожидания для HCB истекает, этот запрос должен быть прерван, как и в случае с запросом XPT_ABORT. Единственное отличие заключается в том, что возвращаемый статус прерванного запроса должен быть CAM_CMD_TIMEOUT вместо CAM_REQ_ABORTED (вот почему реализацию прерывания лучше сделать в виде функции). Но есть ещё одна возможная проблема: что если сам запрос на прерывание зависнет? В этом случае шина SCSI должна быть сброшена, как и при запросе XPT_RESET_BUS (и идея о реализации этого в виде функции, вызываемой из обоих мест, применима и здесь). Также мы должны сбросить всю шину SCSI, если запрос на сброс устройства завис. В итоге функция обработки таймаута будет выглядеть следующим образом:

```
static void
xxx_timeout(void *arg)
{
    struct xxx_hcb *hcb = (struct xxx_hcb *)arg;
    struct xxx_softc *softc;
    struct ccb_hdr *ccb_h;

    softc = hcb->softc;
    ccb_h = &hcb->ccb->ccb_h;

    if (hcb->flags & HCB_BEING_ABORTED || ccb_h->func_code == XPT_RESET_DEV) {
        xxx_reset_bus(softc);
    } else {
        xxx_abort_ccb(hcb->ccb, CAM_CMD_TIMEOUT);
    }
}
```

Когда мы прерываем запрос, все остальные отключённые запросы к тому же целевому устройству/LUN также прерываются. Возникает вопрос: следует ли возвращать их со статусом CAM_REQ_ABORTED или CAM_CMD_TIMEOUT? Текущие драйверы используют CAM_CMD_TIMEOUT. Это кажется логичным, потому что если один запрос превысил время ожидания, то, вероятно, с устройством происходит что-то действительно плохое, и если их не трогать, они бы сами превысили время ожидания.

# Глава 13. Устройства USB

## 13.1. Введение

Универсальная последовательная шина (USB) — это новый способ подключения устройств к персональным компьютерам. Архитектура шины поддерживает двустороннюю связь и была разработана в ответ на усложнение устройств, требующих большего взаимодействия с хостом. Поддержка USB включена во все современные чипсеты ПК и, следовательно, доступна во всех недавно собранных компьютерах. Выпуск Apple iMac только с USB стал серьёзным стимулом для производителей оборудования выпускать USB-версии своих устройств. Согласно будущим спецификациям ПК, все устаревшие разъёмы должны быть заменены одним или несколькими USB-разъёмами, обеспечивающими универсальные возможности plug and play. Поддержка USB-оборудования появилась в NetBSD на очень раннем этапе и была разработана Леннартом Аугустссоном для проекта NetBSD. Код был портирован в FreeBSD, и в настоящее время мы поддерживаем общую кодовая базу. Для реализации подсистемы USB важны некоторые особенности USB.

*Леннарт Аугустссон выполнил большую часть работы по реализации поддержки USB для проекта NetBSD. Огромная благодарность за этот невероятный объём работы. Также большое спасибо Арди и Дирку за их комментарии и вычитку этой статьи.*

- Устройства подключаются к портам компьютера напрямую или через устройства, называемые концентраторами, образуя древовидную структуру устройств.
- Устройства можно подключать и отключать во время работы.
- Устройства могут приостанавливать свою работу и инициировать возобновление работы основной системы
- Поскольку устройства могут получать питание от шины, программное обеспечение хоста должно отслеживать энергопотребление для каждого концентратора.
- Различные требования к качеству обслуживания для разных типов устройств, а также максимум в 126 устройств, которые могут быть подключены к одной шине, требуют правильного планирования передач по общей шине, чтобы полностью использовать доступную пропускную способность в 12 Мбит/с (более 400 Мбит/с для USB 2.0)
- Устройства являются интеллектуальными и содержат легко доступную информацию о себе

Разработка драйверов для подсистемы USB и подключённых к ней устройств поддерживается спецификациями, которые были разработаны и будут разрабатываться. Эти спецификации общедоступны на домашних страницах USB. Apple активно продвигает стандартизированные драйверы, предоставляя драйверы для универсальных классов в своей операционной системе MacOS и не поощряя использование отдельных драйверов для каждого нового устройства. Эта глава пытается собрать основную информацию для базового понимания реализации стека USB 2.0 в FreeBSD/NetBSD. Однако рекомендуется читать её вместе с соответствующими спецификациями 2.0 и другими ресурсами для разработчиков:

- Спецификация USB 2.0 (http://www.usb.org/developers/docs/usb20_docs/)

- Универсальный интерфейс хост-контроллера (UHCI) Спецификация (ftp://ftp.netbsd.org/pub/NetBSD/misc/blymn/uhci11d.pdf)
- Спецификация интерфейса Open Host Controller (OHCI)(ftp://ftp.compaq.com/pub/supportinformation/papers/hcir1_0a.pdf)
- Раздел для разработчиков на домашней странице USB (http://www.usb.org/developers/)

### 13.1.1. Структура стека USB

Поддержка USB в FreeBSD может быть разделена на три уровня. Самый нижний уровень содержит драйвер контроллера хоста, предоставляющий универсальный интерфейс к оборудованию и его механизмам планирования. Он поддерживает инициализацию оборудования, планирование передач и обработку завершённых и/или неудачных передач. Каждый драйвер контроллера хоста реализует виртуальный концентратор, обеспечивающий независимый от оборудования доступ к регистрам, управляющим корневыми портами на задней панели машины.

Средний уровень обрабатывает подключение и отключение устройства, базовую инициализацию устройства, выбор драйвера, каналы связи (pipe) и управляет ресурсами. Этот сервисный уровень также контролирует стандартные каналы и запросы устройств, передаваемые через них.

Верхний уровень содержит отдельные драйверы, поддерживающие конкретные (классы) устройств. Эти драйверы реализуют протокол, используемый в каналах, отличных от стандартного. Они также реализуют дополнительную функциональность для обеспечения доступа к устройству другим частям ядра или пользовательского пространства. Они используют интерфейс драйвера USB (USBDI), предоставляемый уровнем сервисов.

## 13.2. Контроллеры хоста

Хост-контроллер (HC) управляет передачей пакетов на шине. Используются кадры длительностью 1 миллисекунда. В начале каждого кадра хост-контроллер генерирует пакет Start of Frame (SOF).

Пакет SOF используется для синхронизации начала кадра и отслеживания номера кадра. Внутри каждого кадра передаются пакеты, либо от хоста к устройству (исходящие), либо от устройства к хосту (входящие). Передачи всегда инициируются хостом (опросные передачи). Поэтому на каждой шине USB может быть только один хост. Каждая передача пакета имеет стадию статуса, в которой получатель данных может вернуть либо ACK (подтверждение приема), NAK (повторить), STALL (ошибка) либо ничего (искаженная стадия данных, устройство недоступно или отключено). В разделе 8.5 спецификации USB 2.0 подробно объясняются детали пакетов. На шине USB могут происходить четыре различных типа передач: управляющие, массовые, прерывания и изохронные. Типы передач и их характеристики описаны ниже.

Крупные передачи между устройством на шине USB и драйвером устройства разделяются на несколько пакетов хост-контроллером или драйвером HC.

Запросы устройств (управляющие передачи) к конечным точкам по умолчанию являются

особыми. Они состоят из двух или трёх фаз: SETUP, DATA (опционально) и STATUS. Пакет настройки отправляется на устройство. Если присутствует фаза данных, направление пакета(ов) данных указывается в пакете настройки. Направление в фазе статуса противоположно направлению во время фазы данных или IN, если фазы данных не было. Аппаратное обеспечение хост-контроллера также предоставляет регистры с текущим состоянием корневых портов и изменениями, произошедшими с момента последнего сброса регистра изменений статуса. Доступ к этим регистрам предоставляется через виртуализированный концентратор, как предложено в спецификации USB. Виртуальный концентратор должен соответствовать классу устройств концентратора, указанному в главе 11 этой спецификации. Он должен предоставлять канал по умолчанию, через который можно отправлять запросы устройств. Он возвращает стандартные и специфичные для класса концентратора наборы дескрипторов. Также он должен предоставлять прерывающий канал, сообщающий об изменениях, происходящих на его портах. В настоящее время доступны две спецификации для хост-контроллеров: - Universal Host Controller Interface (UHCI) от Intel и Open Host Controller Interface (OHCI) от Compaq, Microsoft и National Semiconductor. Спецификация UHCI разработана для уменьшения аппаратной сложности, требуя от драйвера хост-контроллера предоставления полного расписания передач для каждого кадра. Контроллеры типа OHCI гораздо более независимы, предоставляя более абстрактный интерфейс и выполняя большую часть работы самостоятельно.

### 13.2.1. UHCI

Контроллер UHCI поддерживает список кадров (framelist) с 1024 указателями на структуры данных для каждого кадра. Он распознаёт два типа данных: дескрипторы передачи (TD) и головы очередей (QH). Каждый TD представляет пакет для передачи в конечную точку устройства или из неё. QH служат для группировки TD (и других QH) вместе.

Каждая передача состоит из одного или нескольких пакетов. Драйвер UHCI разделяет большие передачи на несколько пакетов. Для каждой передачи, за исключением изохронных, выделяется QH. Для каждого типа передачи эти QH собираются в QH для соответствующего типа. Изохронные передачи должны выполняться первыми из-за требования фиксированной задержки и непосредственно указываются указателем в списке кадров. Последний изохронный TD ссылается на QH для прерывающих передач для этого кадра. Все QH для прерывающих передач указывают на QH для управляющих передач, который, в свою очередь, указывает на QH для массовых передач. Следующая диаграмма даёт графическое представление этого:

В результате выполняется следующее расписание в каждом кадре. После получения указателя на текущий кадр из списка кадров контроллер сначала выполняет TDs для всех изохронных пакетов в этом кадре. Последний из этих TDs ссылается на QH для прерывающих передач этого кадра. Хост-контроллер затем переходит от этого QH к QHs для отдельных прерывающих передач. После завершения этой очереди, QH для прерывающих передач ссылается на QH для всех управляющих передач. Он выполняет все подочереди, запланированные там, а затем все передачи, поставленные в очередь в QH для массовых передач. Для упрощения обработки завершённых или неудачных передач аппаратное обеспечение генерирует различные типы прерываний в конце каждого кадра. В последнем TD для передачи бит **Interrupt-On Completion** устанавливается драйвером HC, чтобы

вызвать прерывание по завершении передачи. Прерывание ошибки возникает, если TD достигает максимального количества ошибок. Если в TD установлен бит **short packet detect** и передано меньше установленной длины пакета, это прерывание уведомляет драйвер контроллера о завершении передачи. Задача драйвера хост-контроллера — определить, какая передача завершилась или вызвала ошибку. При вызове процедура обслуживания прерывания находит все завершённые передачи и вызывает их callback-функции.

Обратитесь к спецификации UHCI для более подробного описания.

### 13.2.2. OHCI

Программирование OHCI-контроллера значительно проще. Контроллер предполагает, что доступен набор конечных точек, и учитывает приоритеты планирования и порядок типов передач в кадре. Основная структура данных, используемая хост-контроллером, — это дескриптор конечной точки (ED), к которому присоединена очередь дескрипторов передачи (TD). ED содержит максимальный размер пакета, разрешенный для конечной точки, а аппаратное обеспечение контроллера выполняет разделение на пакеты. Указатели на буферы данных обновляются после каждой передачи, и когда начальный и конечный указатели становятся равны, TD перемещается в очередь завершённых (done-queue). Четыре типа конечных точек (прерывание, изохронная, управление и массовая) имеют свои собственные очереди. Управляющие и массовые конечные точки помещаются каждая в свою очередь. ED прерываний организуются в дерево, где уровень в дереве определяет частоту их выполнения.

Порядок действий, выполняемых хост-контроллером в каждом кадре, выглядит следующим образом. Контроллер сначала запускает непериодические очереди управления и массовых передач, вплоть до ограничения времени, установленного драйвером HC. Затем выполняются прерывающие передачи для данного номера кадра, используя младшие пять битов номера кадра в качестве индекса уровня 0 дерева ED прерываний. В конце этого дерева подключены изохронные ED, которые затем обходятся. Изохронные TD содержат номер кадра, в котором должна быть запущена первая передача. После выполнения всех периодических передач очереди управления и массовых передач обходятся снова. Периодически вызывается процедура обслуживания прерывания для обработки очереди завершённых операций и вызова обратных вызовов для каждой передачи, а также для перепланирования прерывающих и изохронных конечных точек.

См. UHCI Specification для более подробного описания. Средний уровень обеспечивает контролируемый доступ к устройству и управляет ресурсами, используемыми различными драйверами и уровнем сервисов. Этот уровень отвечает за следующие аспекты:

- Информация о конфигурации устройства
- Каналы для взаимодействия с устройством
- Обнаружение, подключение и отключение от устройства.

## 13.3. Информация об устройстве USB

### 13.3.1. Информация о конфигурации устройства

Каждое устройство предоставляет различные уровни информации о конфигурации. У каждого устройства есть одна или несколько конфигураций, из которых одна выбирается во время обнаружения/подключения. Конфигурация определяет требования к питанию и пропускной способности. В каждой конфигурации может быть несколько интерфейсов. Интерфейс устройства — это набор конечных точек. Например, USB-колонки могут иметь интерфейс для аудиоданных (Audio Class) и интерфейс для регуляторов, ручек и кнопок (HID Class). Все интерфейсы в конфигурации активны одновременно и могут быть подключены разными драйверами. Каждый интерфейс может иметь альтернативные варианты, предоставляющие различные параметры качества обслуживания. Например, в камерах это используется для поддержки разных размеров кадра и количества кадров в секунду.

В каждом интерфейсе может быть указано 0 или более конечных точек. Конечные точки — это однонаправленные точки доступа для связи с устройством. Они предоставляют буферы для временного хранения входящих или исходящих данных устройства. Каждая конечная точка имеет уникальный адрес в конфигурации — номер конечной точки плюс её направление. Конечная точка по умолчанию, конечная точка 0, не является частью какого-либо интерфейса и доступна во всех конфигурациях. Она управляется уровнем сервисов и не доступна напрямую драйверам устройств.

Эта иерархическая конфигурационная информация описывается в устройстве стандартным набором дескрипторов (см. раздел 9.6 спецификации USB). Они могут быть запрошены через Get Descriptor Request. Сервисный уровень кэширует эти дескрипторы, чтобы избежать ненужных передач по шине USB. Доступ к дескрипторам предоставляется через вызовы функций.

- Дескрипторы устройств: Общая информация об устройстве, такая как Vendor (производитель), Product (продукт) и Revision Id (идентификатор ревизии), поддерживаемый класс устройства, подкласс и протокол, если применимо, максимальный размер пакета для конечной точки по умолчанию и т. д.
- Дескрипторы конфигурации: количество интерфейсов в данной конфигурации, поддержка функций приостановки и возобновления работы, а также требования к питанию.
- Дескрипторы интерфейса: класс интерфейса, подкласс и протокол (если применимо), количество альтернативных настроек интерфейса и количество конечных точек.
- Дескрипторы конечных точек: Адрес конечной точки, направление и тип, максимальный поддерживаемый размер пакета и частота опроса, если тип является конечной точкой прерывания. Для конечной точки по умолчанию (конечная точка 0) дескриптора не существует, и она никогда не учитывается в дескрипторе интерфейса.
- Дескрипторы строк: В остальных дескрипторах для некоторых полей указываются индексы строк. Эти индексы могут использоваться для получения описательных строк, возможно, на нескольких языках.

Спецификации классов могут добавлять свои собственные типы дескрипторов, которые доступны через запрос GetDescriptor.

Канальный обмен данными с конечными точками устройства осуществляется через так называемые **каналы**. Драйверы передают данные конечным точкам через канал и предоставляют функцию обратного вызова, которая вызывается при завершении или сбое передачи (асинхронные передачи) или ожидают завершения (синхронная передача). Передачи данных в конечную точку сериализуются в канале. Передача может завершиться успешно, завершиться с ошибкой или превысить время ожидания (если оно было задано). Существует два типа таймаутов для передач. Таймауты могут происходить из-за истечения времени на шине USB (миллисекунды). Эти таймауты рассматриваются как сбои и могут быть вызваны отключением устройства. Вторая форма таймаута реализована на уровне программного обеспечения и срабатывает, если передача не завершается в течение заданного времени (секунды). Это происходит, когда устройство отрицательно подтверждает (NAK) переданные пакеты. Причинами могут быть неготовность устройства к приему данных, переполнение буфера или пустой буфер, либо ошибки протокола.

Если передача через канал превышает максимальный размер пакета, указанный в соответствующем дескрипторе конечной точки, хост-контроллер (OHCI) или драйвер HC (UHCI) разделит передачу на пакеты максимального размера, при этом последний пакет может быть меньше максимального размера.

Иногда для устройства не является проблемой вернуть меньше данных, чем запрошено. Например, при передаче данных в режиме bulk-in на модем может быть запрошено 200 байт данных, но у модема в данный момент доступно только 5 байт. Драйвер может установить флаг короткого пакета (SPD). Это позволяет хост-контроллеру принять пакет, даже если объём переданных данных меньше запрошенного. Этот флаг действителен только для in-передач, так как объём данных, отправляемых на устройство, всегда известен заранее. Если во время передачи в устройстве происходит неустранимая ошибка, канал останавливается (stalled). Прежде чем принимать или отправлять дополнительные данные, драйвер должен устранить причину остановки и сбросить условие остановки конечной точки, отправив запрос `clear endpoint halt` через канал по умолчанию. Конечная точка по умолчанию никогда не должна останавливаться.

Существует четыре различных типа конечных точек и соответствующих каналов: - Управляющий канал / канал по умолчанию: На каждое устройство приходится один управляющий канал, подключённый к конечной точке по умолчанию (конечная точка 0). Этот канал передаёт запросы устройства и связанные с ними данные. Разница между передачами через канал по умолчанию и другими каналами заключается в том, что протокол для этих передач описан в спецификации USB. Эти запросы используются для сброса и настройки устройства. Базовый набор команд, который должен поддерживаться каждым устройством, приведён в главе 9 спецификации USB. Команды, поддерживаемые на этом канале, могут быть расширены спецификацией класса устройства для обеспечения дополнительной функциональности.

- Массовый канал: Это USB-эквивалент среды передачи данных в сыром виде.
- Прерывающий канал: Хост отправляет запрос данных на устройство, и если устройству нечего отправлять, оно отвечает NAK на пакет данных. Прерывающие передачи планируются с частотой, указанной при создании канала.
- Изохронный канал: Эти каналы предназначены для изохронных данных, например, видео- или аудиопотоков, с фиксированной задержкой, но без гарантированной

доставки. В текущей реализации доступна некоторая поддержка каналов этого типа. Пакеты в управляющих, массовых и прерывающих передачах повторяются, если во время передачи возникает ошибка или устройство отрицательно подтверждает пакет (NAK) из-за, например, нехватки буферного пространства для хранения входящих данных. Однако изохронные пакеты не повторяются в случае неудачной доставки или NAK пакета, так как это может нарушить временные ограничения.

Доступность необходимой полосы пропускания рассчитывается во время создания канала. Передачи планируются в рамках интервалов в 1 миллисекунду. Распределение полосы пропускания внутри интервала регламентируется спецификацией USB, раздел 5.6 [2]. Изохронные и прерывающие передачи могут использовать до 90% полосы пропускания в рамках интервала. Пакеты для управляющих и массовых передач планируются после всех изохронных и прерывающих пакетов и используют всю оставшуюся полосу пропускания.

Дополнительная информация о планировании передач и освобождении пропускной способности доступна в главе 5 спецификации USB, разделе 1.3 спецификации UHCI и разделе 3.4.2 спецификации OHCI.

## 13.4. Обнаружение устройства и подключение

После уведомления от концентратора о подключении нового устройства сервисный уровень включает порт, предоставляя устройству ток 100 мА. На этом этапе устройство находится в своём исходном состоянии и прослушивает адрес устройства 0. Сервисный уровень продолжит получение различных дескрипторов через стандартный канал. После этого он отправит запрос Set Address, чтобы переместить устройство с исходного адреса (адрес 0). Несколько драйверов устройств могут поддерживать данное устройство. Например, драйвер модема может поддерживать ISDN TA через интерфейс совместимости AT. Однако драйвер, предназначенный для конкретной модели ISDN-адаптера, может обеспечить гораздо лучшую поддержку этого устройства. Для обеспечения такой гибкости пробы возвращают приоритеты, указывающие уровень их поддержки. Поддержка конкретной версии продукта имеет наивысший приоритет, а универсальный драйвер — самый низкий. Также возможно, что несколько драйверов могут быть подключены к одному устройству, если в одной конфигурации присутствует несколько интерфейсов. Каждому драйверу требуется поддерживать только подмножество интерфейсов.

Поиск драйвера для нового подключённого устройства сначала проверяет наличие специфичных для устройства драйверов. Если они не найдены, код проверки перебирает все поддерживаемые конфигурации, пока драйвер не будет подключен в одной из них. Для поддержки устройств с несколькими драйверами на разных интерфейсах проверка перебирает все интерфейсы в конфигурации, которые ещё не были заняты драйвером. Конфигурации, превышающие выделенный бюджет мощности для концентратора, игнорируются. Во время подключения драйвер должен инициализировать устройство в его рабочее состояние, но не сбрасывать его, так как это приведёт к отключению устройства от шины и перезапуску процесса проверки. Чтобы избежать излишнего потребления пропускной способности, не следует запрашивать канал прерывания во время подключения, а отложить его выделение до момента открытия файла и фактического использования данных. При закрытии файла канал следует снова закрыть, даже если устройство остаётся подключённым.

### 13.4.1. Отключение и извлечение устройства

Драйвер устройства должен ожидать получения ошибок во время любой транзакции с устройством. Дизайн USB поддерживает и поощряет отключение устройств в любой момент времени. Драйверы должны гарантировать корректную обработку ситуаций, когда устройство исчезает.

Кроме того, устройство, которое было отключено и снова подключено, не будет повторно присоединено к тому же экземпляру устройства. Это может измениться в будущем, когда больше устройств будут поддерживать серийные номера (см. дескриптор устройства) или будут разработаны другие способы определения идентификатора устройства.

Отключение устройства сигнализируется концентратором в пакете прерывания, передаваемом драйверу концентратора. Информация об изменении состояния указывает, на каком порту произошло изменение подключения. Метод отключения устройства для всех драйверов устройств, подключённых к этому порту, вызывается, и структуры очищаются. Если состояние порта указывает, что за это время к порту было подключено устройство, начнется процедура обнаружения и подключения устройства. Сброс устройства вызовет последовательность отключения-подключения на концентраторе и будет обработан, как описано выше.

## 13.5. Информация о протоколе драйверов USB

Используемый протокол для каналов, отличных от стандартного, не определен спецификацией USB. Информацию об этом можно найти из различных источников. Наиболее точный источник — раздел для разработчиков на домашних страницах USB. На этих страницах доступно растущее количество спецификаций классов устройств. Эти спецификации определяют, каким должно быть совместимое устройство с точки зрения драйвера, базовую функциональность, которую оно должно предоставлять, и протокол, используемый на каналах связи. Спецификация USB включает описание класса концентраторов (Hub Class). Спецификация класса устройств человеко-машинного интерфейса (HID) была создана для поддержки клавиатур, планшетов, сканеров штрих-кодов, кнопок, регуляторов, переключателей и т. д. Третий пример — спецификация класса устройств хранения данных (Mass Storage). Полный список классов устройств можно найти в разделе для разработчиков на домашних страницах USB.

Для многих устройств информация о протоколе ещё не опубликована. Сведения об используемом протоколе могут быть доступны у компании-производителя устройства. Некоторые компании потребуют подписания соглашения о неразглашении (NDA), прежде чем предоставить спецификации. В большинстве случаев это исключает возможность сделать драйвер открытым.

Еще один хороший источник информации — исходные коды драйверов Linux, так как ряд компаний начал предоставлять драйверы для Linux для своих устройств. Всегда полезно связаться с авторами этих драйверов для получения информации.

Пример: Устройства интерфейса пользователя (HID) — Спецификация для устройств интерфейса пользователя, таких как клавиатуры, мыши, планшеты, кнопки, регуляторы и т.д., упоминается в других спецификациях классов устройств и используется во многих

устройствах.

Например, аудиоколонки предоставляют конечные точки для цифро-аналоговых преобразователей и, возможно, дополнительный канал для микрофона. Они также предоставляют конечную точку HID в отдельном интерфейсе для кнопок и регуляторов на передней панели устройства. То же самое справедливо для класса управления монитором. Реализовать поддержку этих интерфейсов достаточно просто с помощью доступных библиотек ядра и пользовательского пространства, а также драйвера класса HID или универсального драйвера. Другим примером устройства с несколькими интерфейсами в одной конфигурации, управляемыми разными драйверами, является недорогая клавиатура со встроенным устаревшим портом для мыши. Чтобы избежать затрат на включение аппаратного обеспечения USB-концентратора в устройство, производители объединили данные мыши, полученные с порта PS/2 на задней панели клавиатуры, и нажатия клавиш в два отдельных интерфейса в одной конфигурации. Драйверы мыши и клавиатуры подключаются к соответствующему интерфейсу и выделяют каналы для двух независимых конечных точек.

Пример: Загрузка микропрограммы — многие разработанные устройства основаны на процессоре общего назначения с добавленным USB-ядром. Поскольку разработка драйверов и микропрограмм для USB-устройств всё ещё очень нова, многие устройства требуют загрузки микропрограммы после подключения.

Процедура выполняется следующим образом. Устройство идентифицирует себя через идентификаторы производителя и продукта. Первый драйвер проверяет его и подключается к нему, затем загружает в него микропрограмму. После этого устройство выполняет мягкую перезагрузку, и драйвер отключается. После небольшой паузы устройство снова появляется на шине. При этом идентификаторы производителя, продукта и версии устройства изменятся, что отражает факт загрузки микропрограммы, и в результате второй драйвер проверит его и подключится к нему.

Примером таких устройств является плата ввода-вывода ActiveWire, основанная на чипе EZ-USB. Для этого чипа доступен универсальный загрузчик микропрограмм. Микропрограмма, загруженная в плату ActiveWire, изменяет идентификатор ревизии. Затем выполняется мягкий сброс USB-части чипа EZ-USB для отключения от USB-шины и повторного подключения.

Пример: Поддержка устройств хранения данных в основном построена на существующих протоколах. Устройство Iomega USB Zipdrive основано на SCSI-версии их накопителя. SCSI-команды и статусные сообщения упаковываются в блоки и передаются через массовые каналы к устройству и от него, эмулируя SCSI-контроллер через USB-соединение. ATAPI и UFI команды поддерживаются аналогичным образом.

Спецификация Mass Storage поддерживает 2 различных типа обёртки командного блока. Первоначальная попытка была основана на отправке команды и состояния через канал по умолчанию с использованием массовых передач для данных, перемещаемых между хостом и устройством. На основе опыта был разработан второй подход, основанный на обёртке командных и статусных блоков и их отправке через конечные точки массовой передачи (bulk out и bulk in). Спецификация точно определяет, что должно происходить и когда, а также что необходимо делать в случае возникновения ошибки. Наибольшую сложность

при написании драйверов для таких устройств представляет встраивание USB-ориентированного протокола в существующую поддержку устройств хранения данных. CAM предоставляет механизмы для этого достаточно прямолинейным способом. С ATAPI всё менее просто, так как исторически интерфейс IDE никогда не имел множества различных вариантов реализации.

Поддержка USB-дисковода от Y-E Data также не прямолинейна, так как был разработан новый набор команд.

# Глава 14. Newbus

*Особая благодарность Мэтью Н. Додду, Уорнеру Лошу, Биллу Полу, Дагу Рэбсону, Майку Смиту, Питеру Вемму и Скотту Лонгу*.

Эта глава подробно объясняет фреймворк устройств Newbus.

## 14.1. Драйверы устройств

### 14.1.1. Назначение драйвера устройства

Драйвер устройства — это программный компонент, который предоставляет интерфейс между обобщённым представлением периферийного устройства (например, диска, сетевого адаптера) в ядре и его фактической реализацией. *Интерфейс драйвера устройства (DDI)* — это определённый интерфейс между ядром и компонентом драйвера устройства.

### 14.1.2. Типы драйверов устройств

В UNIX®, а следовательно, и в FreeBSD, были времена, когда определялось четыре типа устройств:

- драйверы блочных устройств
- драйверы символьных устройств
- драйверы сетевых устройств
- драйверы псевдоустройств

*Блочные устройства* работали таким образом, что использовали блоки [данных] фиксированного размера. Этот тип драйвера зависел от так называемого *буферного кэша*, который кэшировал доступные блоки данных в выделенной части памяти. Часто этот буферный кэш был основан на отложенной записи (write-behind), что означало, что при изменении данных в памяти они синхронизировались с диском во время периодической очистки диска системой, тем самым оптимизируя запись.

### 14.1.3. Символьные устройства

Однако в версиях FreeBSD 4.0 и выше различие между блочными и символьными устройствами перестало существовать.

## 14.2. Обзор Newbus

*Newbus* — это реализация новой архитектуры шины, основанной на уровнях абстракции, которая была впервые представлена в FreeBSD 3.0, когда порт для Alpha был добавлен в дерево исходного кода. Однако только в версии 4.0 она стала системой по умолчанию для использования с драйверами устройств. Её цель — предоставить более объектно-ориентированный способ взаимодействия между различными шинами и устройствами, которые хост-система предоставляет *операционной системе*.

Основные функции включают, среди прочего:

- динамическое присоединение
- простая модуляризация драйверов
- псевдо-шины

Одним из наиболее заметных изменений является переход от плоской и нерегламентированной системы к структуре дерева устройств.

На верхнем уровне находится устройство *"root"*, которое является родительским для всех остальных устройств. Для каждой архитектуры обычно существует единственный дочерний элемент "root", к которому подключены такие компоненты, как *мосты host-to-PCI* и т.д. Для x86 этим устройством "root" является устройство *"nexus"*. Для Alpha различные модели Alpha имеют разные устройства верхнего уровня, соответствующие различным аппаратным наборам микросхем, включая *lca*, *apecs*, *cia* и *tsunami*.

Устройство в контексте Newbus представляет собой отдельную аппаратную сущность в системе. Например, каждое PCI-устройство представлено устройством Newbus. Любое устройство в системе может иметь дочерние устройства; устройство, у которого есть дочерние устройства, часто называют *"шиной"*. Примерами распространённых шин в системе являются ISA и PCI, которые управляют списками устройств, подключённых к шинам ISA и PCI соответственно.

Часто соединение между различными типами шин представлено устройством *"мост"*, которое обычно имеет один дочерний элемент для подключённой шины. Примером этого является *PCI-to-PCI мост*, который представлен устройством *pcibN* на родительской PCI-шине и имеет дочерний элемент *pciN* для подключённой шины. Такая структура упрощает реализацию дерева PCI-шин, позволяя использовать общий код как для верхнеуровневых, так и для соединенных через мост шин.

Каждое устройство в архитектуре Newbus запрашивает у своего родителя отображение своих ресурсов. Затем родитель запрашивает у своего собственного родителя, пока запрос не достигнет nexus. Таким образом, по сути, nexus - это единственная часть системы Newbus, которая знает обо всех ресурсах.



Устройство ISA может захотеть отобразить свой порт ввода-вывода по адресу `0x230`, поэтому оно запрашивает у своего родителя, в данном случае — шины ISA. Шина ISA передаёт запрос мосту PCI-to-ISA, который, в свою очередь, запрашивает шину PCI. Запрос доходит до моста host-to-PCI и, наконец, до nexus. Прелесть этого восходящего перехода в том, что есть возможность преобразовывать запросы. Например, запрос порта ввода-вывода `0x230` может быть преобразован в отображение памяти по адресу `0xb0000230` на системе MIPS с помощью моста PCI.

Распределение ресурсов может контролироваться в любом месте дерева устройств. Например, на многих платформах Alpha прерывания ISA управляются отдельно от прерываний PCI, а распределение ресурсов для прерываний ISA осуществляется устройством шины ISA Alpha. На IA-32 прерывания ISA и PCI управляются устройством

верхнего уровня nexus. Для обеих архитектур управление пространством памяти и портов осуществляется единым объектом — nexus для IA-32 и соответствующим драйвером чипсета на Alpha (например, CIA или tsunami).

Для стандартизации доступа к памяти и ресурсам, отображённым на порты, Newbus интегрирует API `bus_space` из NetBSD. Они предоставляют единый API для замены inb/outb и прямых операций чтения/записи в память. Преимущество этого подхода в том, что один драйвер может легко использовать либо регистры, отображённые в память, либо регистры, отображённые на порты (некоторое оборудование поддерживает оба варианта).

Эта поддержка интегрирована в механизм распределения ресурсов. При выделении ресурса драйвер может получить связанные `bus_space_tag_t` и `bus_space_handle_t` из этого ресурса.

Newbus также позволяет определять методы интерфейса в файлах, предназначенных для этой цели. Это файлы с расширением .m, которые находятся в иерархии src/sys.

Ядро системы Newbus представляет собой расширяемую модель «объектно-ориентированного программирования». Каждое устройство в системе имеет таблицу поддерживаемых методов. Система и другие устройства используют эти методы для управления устройством и запроса услуг. Различные методы, поддерживаемые устройством, определяются рядом «интерфейсов». «Интерфейс» — это просто группа связанных методов, которые могут быть реализованы устройством.

В системе Newbus методы для устройства предоставляются различными драйверами устройств в системе. Когда устройство подключается к драйверу во время *автоконфигурации*, оно использует таблицу методов, объявленную драйвером. Устройство может позже *отключиться* от своего драйвера и *подключиться* к новому драйверу с новой таблицей методов. Это позволяет динамически заменять драйверы, что может быть полезно для разработки драйверов.

Интерфейсы описываются языком определения интерфейсов, похожим на язык, используемый для определения операций vnode для файловых систем. Интерфейс хранится в файле методов (который обычно называется foo_if.m).

*Пример 5. Методы Newbus*

```
  # Foo subsystem/driver (a comment...)

  INTERFACE foo

METHOD int doit {
    device_t dev;
};

# DEFAULT is the method that will be used, if a method was not
# provided via: DEVMETHOD()

METHOD void doit_to_child {
    device_t dev;
    driver_t child;
```

```
    } DEFAULT doit_generic_to_child;
```

Когда этот интерфейс компилируется, он генерирует заголовочный файл "foo_if.h", который содержит объявления функций:

```
int FOO_DOIT(device_t dev);
int FOO_DOIT_TO_CHILD(device_t dev, device_t child);
```

Исходный файл `foo_if.c` также создаётся для сопровождения автоматически сгенерированного заголовочного файла; он содержит реализации функций, которые ищут расположение соответствующих функций в таблице методов объекта и вызывают эту функцию.

Система определяет два основных интерфейса. Первый фундаментальный интерфейс называется *"device"* (устройство) и включает методы, которые относятся ко всем устройствам. Методы в интерфейсе *"device"* включают *"probe"* (обнаружение), *"attach"* (присоединение) и *"detach"* (отсоединение) для управления обнаружением оборудования, а также *"shutdown"* (выключение), *"suspend"* (приостановка) и *"resume"* (возобновление) для уведомления о критических событиях.

Второй, более сложный интерфейс — *"bus"*. Этот интерфейс содержит методы, подходящие для устройств, имеющих дочерние элементы, включая методы доступа к специфичной для шины информации об устройстве [1], уведомления о событиях (`child_detached`, `driver_added`) и управление ресурсами (`alloc_resource`, `activate_resource`, `deactivate_resource`, `release_resource`).

Многие методы в интерфейсе "bus" выполняют сервисы для некоторого дочернего устройства шины. Эти методы обычно используют первые два аргумента для указания шины, предоставляющей сервис, и дочернего устройства, запрашивающего сервис. Для упрощения кода драйвера многие из этих методов имеют вспомогательные функции, которые находят родительское устройство и вызывают метод у родителя. Например, метод `BUS_TEARDOWN_INTR(device_t dev, device_t child, ⋯)` может быть вызван с помощью функции `bus_teardown_intr(device_t child, ⋯)`.

Некоторые типы шин в системе определяют дополнительные интерфейсы для предоставления доступа к специфичной для шины функциональности. Например, драйвер шины PCI определяет интерфейс "pci", который имеет два метода `read_config` и `write_config` для доступа к конфигурационным регистрам устройства PCI.

## 14.3. Newbus API

Поскольку API Newbus очень обширен, в этом разделе предпринята попытка его документирования. Дополнительная информация будет добавлена в следующей версии этого документа.

### 14.3.1. Важные места в иерархии исходного кода

src/sys/[arch]/[arch] - Код ядра для конкретной аппаратной архитектуры находится в этом каталоге. Например, архитектура `i386` или архитектура `SPARC64`.

src/sys/dev/[bus] - поддержка устройств для конкретной `[bus]` находится в этом каталоге.

src/sys/dev/pci - Код поддержки шины PCI находится в этом каталоге.

src/sys/[isa|pci] - В этом каталоге находятся драйверы устройств PCI/ISA. Код поддержки шины PCI/ISA располагался в этом каталоге в FreeBSD версии `4.0`.

### 14.3.2. Важные структуры и определения типов

`devclass_t` - Это определение типа указателя на `struct devclass`.

`device_method_t` - Это то же самое, что и `kobj_method_t` (см. src/sys/kobj.h).

`device_t` - Это определение типа указателя на структуру `struct device`. `device_t` представляет устройство в системе. Это объект ядра. Подробности реализации см. в src/sys/sys/bus_private.h.

`driver_t` - Это определение типа, которое ссылается на `struct driver`. Структура `driver` является классом объекта ядра `device`; она также содержит данные, приватные для драйвера.

- Реализация *driver_t**

```
struct driver {
  KOBJ_CLASS_FIELDS;
  void    *priv;          /* driver private data */
};
```

Тип `device_state_t`, который является перечислением, `device_state`. Он содержит возможные состояния устройства Newbus до и после процесса автонастройки.

**Состояния устройств _device_state_t**

```
/*
 * src/sys/sys/bus.h
 */
typedef enum device_state {
  DS_NOTPRESENT,  /* not probed or probe failed */
  DS_ALIVE,       /* probe succeeded */
  DS_ATTACHED,    /* attach method called */
  DS_BUSY         /* device is open */
} device_state_t;
```

[1] bus_generic_read_ivar(9) и bus_generic_write_ivar(9)

# Глава 15. Звуковая подсистема

## 15.1. Введение

Подсистема звука FreeBSD чётко разделяет общие вопросы обработки звука и детали, специфичные для устройств. Это упрощает добавление поддержки нового оборудования.

pcm(4) — это центральный компонент подсистемы звука. В основном он реализует следующие элементы:

- Интерфейс системных вызовов (read, write, ioctls) для работы с оцифрованным звуком и функциями микшера. Набор команд ioctl совместим с устаревшим интерфейсом *OSS* или *Voxware*, что позволяет портировать распространённые мультимедийные приложения без изменений.
- Общий код для обработки звуковых данных (преобразование форматов, виртуальные каналы).
- Единый программный интерфейс к аппаратно-зависимым модулям аудиоинтерфейсов.
- Дополнительная поддержка некоторых распространённых аппаратных интерфейсов (ac97) или общий код для специфичного оборудования (например: подпрограммы ISA DMA).

Поддержка конкретных звуковых карт реализована аппаратно-специфичными драйверами, которые предоставляют интерфейсы каналов и микшера для подключения к общему коду pcm.

В этой главе термин pcm будет относиться к центральной, общей части звукового драйвера, в отличие от аппаратно-зависимых модулей.

Разработчик драйверов, только начинающий свою разработку, конечно, захочет начать с существующего модуля и использовать его код в качестве основного источника информации. Однако, хотя код подсистемы звука чист и аккуратен, он в основном лишён комментариев. Этот документ пытается дать обзор интерфейса фреймворка и ответить на некоторые вопросы, которые могут возникнуть при адаптации существующего кода.

В качестве альтернативы или в дополнение к началу разработки с примера драйвера из кода системы, вы можете найти шаблон драйвера с комментариями по адресу https://people.FreeBSD.org/~cg/template.c

## 15.2. Файлы

Весь соответствующий код находится в /usr/src/sys/dev/sound/, за исключением определений публичного интерфейса ioctl, которые можно найти в /usr/src/sys/sys/soundcard.h

В каталоге /usr/src/sys/dev/sound/, папка pcm/ содержит основной код, тогда как каталоги pci/, isa/ и usb/ содержат драйверы для плат PCI и ISA, а также для USB-аудиоустройств.

# 15.3. Обнаружение, подключение и т.д.

Драйверы звуковых устройств выполняют обнаружение и подключение почти так же, как и любой модуль драйвера оборудования. Возможно, вам будет полезно ознакомиться с разделами руководства, посвящёнными ISA или PCI, для получения дополнительной информации.

Однако драйверы звука отличаются в некоторых аспектах:

- Они объявляют себя как устройства класса pcm, с приватной структурой устройства `struct snddev_info`:

  ```
  static driver_t xxx_driver = {
      "pcm",
      xxx_methods,
      sizeof(struct snddev_info)
  };

  DRIVER_MODULE(snd_xxxpci, pci, xxx_driver, pcm_devclass, 0, 0);
  MODULE_DEPEND(snd_xxxpci, snd_pcm, PCM_MINVER, PCM_PREFVER,PCM_MAXVER);
  ```

  Большинству звуковых драйверов необходимо хранить дополнительную приватную информацию о своём устройстве. Приватная структура данных обычно выделяется в процедуре подключения (attach). Её адрес передаётся в pcm через вызовы `pcm_register()` и `mixer_init()`. pcm позже передаёт обратно этот адрес в качестве параметра при вызовах интерфейсов звукового драйвера.

- Процедура подключения (attach) звукового драйвера должна объявить свой интерфейс MIXER или AC97 для pcm, вызвав `mixer_init()`. Для интерфейса MIXER это, в свою очередь, приводит к вызову `xxxmixer_init()`.
- Процедура подключения драйвера звука объявляет свою общую конфигурацию CHANNEL для pcm, вызывая `pcm_register(dev, sc, nplay, nrec)`, где `sc` — это адрес структуры данных устройства, используемый при последующих вызовах из pcm, а `nplay` и `nrec` — количество каналов воспроизведения и записи.
- Подпрограмма подключения звукового драйвера объявляет каждый из своих каналов вызовами `pcm_addchan()`. Это настраивает связующий слой канала в pcm и, в свою очередь, вызывает вызов `xxxchannel_init()`.
- Драйвер звука должен вызвать `pcm_unregister()` в процедуре отключения (detach) перед освобождением своих ресурсов.

Существует два возможных способа работы с устройствами, не поддерживающими PnP:

- Используйте метод `device_identify()` (пример: sound/isa/es1888.c). Метод `device_identify()` проверяет наличие оборудования по известным адресам и, если находит поддерживаемое устройство, создаёт новое pcm-устройство, которое затем передаётся для probe/attach.
- Используйте пользовательскую конфигурацию ядра с соответствующими подсказками

для устройств pcm (пример: sound/isa/mss.c).

pcm драйверы должны реализовывать подпрограммы `device_suspend`, `device_resume` и `device_shutdown`, чтобы управление питанием и выгрузка модулей работали корректно.

# 15.4. Интерфейсы

Интерфейс между ядром pcm и звуковыми драйверами определяется в терминах объектов ядра Kobj.

Существует два основных интерфейса, которые обычно предоставляет драйвер звука: *CHANNEL* и либо *MIXER*, либо *AC97*.

Интерфейс *AC97* — это очень небольшой интерфейс доступа к оборудованию (чтение/запись регистров), реализованный драйверами для устройств с кодеком AC97. В этом случае фактический интерфейс MIXER предоставляется общим кодом AC97 в pcm.

## 15.4.1. Интерфейс CHANNEL

#### Общие примечания для параметров функций

Драйверы звука обычно имеют приватную структуру данных для описания своего устройства и по одной структуре для каждого канала воспроизведения и записи, который они поддерживают.

Для всех функций интерфейса CHANNEL первый параметр — это непрозрачный указатель.

Второй параметр представляет собой указатель на приватную структуру данных канала, за исключением `channel_init()`, где передаётся указатель на приватную структуру устройства (и возвращается указатель на канал для дальнейшего использования pcm).

#### Обзор операций передачи данных

Для надёжной передачи звуковых данных ядро pcm и драйверы звука взаимодействуют через общую область памяти, описываемую структурой `struct snd_dbuf`.

`struct snd_dbuf` является приватной для pcm, и драйверы звука получают нужные значения через вызовы функций доступа (`sndbuf_getxxx()`).

Область разделяемой памяти имеет размер `sndbuf_getsize()` и разделена на блоки фиксированного размера по `sndbuf_getblksz()` байт.

При воспроизведении общий механизм передачи выглядит следующим образом (для записи идея обратная):

- pcm сначала заполняет буфер, затем вызывает функцию `xxxchannel_trigger()` драйвера звука с параметром PCMTRIG_START.
- Звуковой драйвер затем организует повторяющуюся передачу всей области памяти (`sndbuf_getbuf()`, `sndbuf_getsize()`) на устройство блоками по `sndbuf_getblksz()` байт. Для каждого переданного блока он вызывает функцию `chn_intr()`pcm (обычно это

происходит во время прерывания).

- `chn_intr()` организует копирование новых данных в область, которая была передана устройству (теперь свободна), и вносит соответствующие обновления в структуру `snd_dbuf`.

### channel_init

`xxxchannel_init()` вызывается для инициализации каждого из каналов воспроизведения или записи. Вызовы инициируются из процедуры подключения драйвера звука. (См. раздел Обнаружение и подключение).

```
static void *
xxxchannel_init(kobj_t obj, void *data,
   struct snd_dbuf *b, struct pcm_channel *c, int dir) ①
{
    struct xxx_info *sc = data;
    struct xxx_chinfo *ch;
     ...
    return ch; ②
 }
```

① `b` — это адрес для канала `struct snd_dbuf`. Он должен быть инициализирован в функции вызовом `sndbuf_alloc()`. Размер буфера, который следует использовать, обычно представляет собой небольшое кратное от 'типичного' размера единицы передачи данных для вашего устройства. `c` — это указатель на структуру управления каналом pcm. Это непрозрачный объект. Функция должна сохранить его в локальной структуре канала для использования в последующих вызовах pcm (например: `chn_intr(c)`). `dir` указывает направление канала (`PCMDIR_PLAY` или `PCMDIR_REC`).

② Функция должна возвращать указатель на приватную область, используемую для управления этим каналом. Этот указатель будет передаваться в качестве параметра при других вызовах интерфейса канала.

### channel_setformat

`xxxchannel_setformat()` должен настроить оборудование для указанного канала под указанный звуковой формат.

```
static int
xxxchannel_setformat(kobj_t obj, void *data, u_int32_t format) ①
{
    struct xxx_chinfo *ch = data;
     ...
    return 0;
 }
```

① `format` указывается как значение `AFMT_XXX` (soundcard.h).

### channel_setspeed

`xxxchannel_setspeed()` настраивает оборудование канала для указанной скорости дискретизации и возвращает возможно скорректированную скорость.

```
        static int
        xxxchannel_setspeed(kobj_t obj, void *data, u_int32_t speed)
        {
            struct xxx_chinfo *ch = data;
             ...
            return speed;
         }
```

### channel_setblocksize

`xxxchannel_setblocksize()` устанавливает размер блока, который является размером единичных транзакций между pcm и звуковым драйвером, а также между звуковым драйвером и устройством. Обычно это количество байт, передаваемых до возникновения прерывания. Во время передачи звуковой драйвер должен вызывать `chn_intr()` из pcm каждый раз, когда передаётся данный размер.

Большинство драйверов звука здесь учитывают только размер блока, который будет использоваться при начале фактической передачи.

```
        static int
        xxxchannel_setblocksize(kobj_t obj, void *data, u_int32_t blocksize)
        {
            struct xxx_chinfo *ch = data;
              ...
            return blocksize; ①
         }
```

① Функция возвращает, возможно, скорректированный размер блока. Если размер блока действительно изменён, следует вызвать `sndbuf_resize()` для корректировки буфера.

### channel_trigger

`xxxchannel_trigger()` вызывается pcm для управления операциями передачи данных в драйвере.

```
        static int
        xxxchannel_trigger(kobj_t obj, void *data, int go) ①
        {
            struct xxx_chinfo *ch = data;
             ...
            return 0;
         }
```

① `go` определяет действие для текущего вызова. Возможные значения:



Если драйвер использует ISA DMA, перед выполнением действий с устройством следует вызвать `sndbuf_isadma()`, которая позаботится о том, что делает DMA-чип.

#### channel_getptr

`xxxchannel_getptr()` возвращает текущее смещение в буфере передачи. Обычно этот вызов выполняется функцией `chn_intr()`, и именно так pcm узнаёт, куда можно передавать новые данные.

#### channel_free

`xxxchannel_free()` вызывается для освобождения ресурсов канала, например, при выгрузке драйвера, и должна быть реализована, если структуры данных канала динамически выделены или если `sndbuf_alloc()` не использовалась для выделения буфера.

#### channel_getcaps

```
struct pcmchan_caps *
xxxchannel_getcaps(kobj_t obj, void *data)
{
    return &xxx_caps; ①
 }
```

① Подпрограмма возвращает указатель на (обычно статически определённую) структуру `pcmchan_caps` (определена в sound/pcm/channel.h). Эта структура содержит минимальную и максимальную частоты дискретизации, а также поддерживаемые звуковые форматы. Пример можно найти в любом драйвере звукового устройства.

#### Дополнительные функции

`channel_reset()`, `channel_resetdone()` и `channel_notify()` предназначены для специальных целей и не должны реализовываться в драйвере без обсуждения на Список рассылки, посвящённый поддержке средств мультимедиа под FreeBSD.

`channel_setdir()` устарела.

### 15.4.2. Интерфейс MIXER

#### mixer_init

`xxxmixer_init()` инициализирует оборудование и сообщает pcm, какие устройства микшера доступны для воспроизведения и записи

```
static int
xxxmixer_init(struct snd_mixer *m)
```

```
{
    struct xxx_info   *sc = mix_getdevinfo(m);
    u_int32_t v;

    [Initialize hardware]

    [Set appropriate bits in v for play mixers] ①
    mix_setdevs(m, v);
    [Set appropriate bits in v for record mixers]
    mix_setrecdevs(m, v)

    return 0;
}
```

① Установите биты в целочисленном значении и вызовите `mix_setdevs()` и `mix_setrecdevs()`, чтобы сообщить pcm, какие устройства существуют.

Определения битов микшера можно найти в soundcard.h (значения `SOUND_MASK_XXX` и сдвиги битов `SOUND_MIXER_XXX`).

#### mixer_set

`xxxmixer_set()` устанавливает уровень громкости для одного устройства микшера.

```
static int
xxxmixer_set(struct snd_mixer *m, unsigned dev,
                 unsigned left, unsigned right) ①
{
    struct sc_info *sc = mix_getdevinfo(m);
    [set volume level]
    return left | (right << 8); ②
}
```

① Устройство указывается как значение `SOUND_MIXER_XXX`. Значения громкости задаются в диапазоне [0-100]. Значение ноль должно отключать звук устройства.

② Поскольку уровни оборудования, вероятно, не совпадут с входной шкалой и будет происходить округление, процедура возвращает фактические значения уровней (в диапазоне 0-100), как показано.

#### mixer_setrecsrc

`xxxmixer_setrecsrc()` устанавливает устройство источника записи.

```
static int
xxxmixer_setrecsrc(struct snd_mixer *m, u_int32_t src) ①
{
    struct xxx_info *sc = mix_getdevinfo(m);

    [look for non zero bit(s) in src, set up hardware]
```

```
            [update src to reflect actual action]
            return src; ②
        }
```

① Желаемые устройства записи указываются в виде битового поля

② Возвращаются фактические устройства, настроенные для записи. Некоторые драйверы могут настраивать только одно устройство для записи. Функция должна возвращать -1 в случае ошибки.

**mixer_uninit, mixer_reinit**

`xxxmixer_uninit()` должен гарантировать, что весь звук отключен, и, если возможно, аппаратный микшер должен быть переведен в режим пониженного энергопотребления.

`xxxmixer_reinit()` должна гарантировать, что аппаратура микшера включена и все настройки, не управляемые `mixer_set()` или `mixer_setrecsrc()`, восстановлены.

### 15.4.3. Интерфейс AC97

Интерфейс *AC97* реализован драйверами с кодеком AC97. У него есть только три метода:

- `xxxac97_init()` возвращает количество найденных кодеков ac97.
- `ac97_read()` и `ac97_write()` читают или записывают указанный регистр.

Интерфейс *AC97* используется кодом AC97 в pcm для выполнения операций более высокого уровня. В качестве примера можно посмотреть sound/pci/maestro3.c или другие файлы в sound/pci/.

# Глава 16. PC Card

Эта глава расскажет о механизмах FreeBSD для написания драйвера устройства для PC Card или CardBus устройства. Однако в настоящее время она лишь документирует, как добавить новое устройство к существующему драйверу pccard.

## 16.1. Добавление устройства

Драйверы устройств знают, какие устройства они поддерживают. В ядре существует таблица поддерживаемых устройств, которую драйверы используют для подключения к устройству.

### 16.1.1. Обзор

PC Cards идентифицируются одним из двух способов, оба основаны на *Card Information Structure* (CIS), хранящейся на карте. Первый метод — использование числовых идентификаторов производителя и продукта. Второй метод — использование удобочитаемых строк, также содержащихся в CIS. Шина PC Card использует централизованную базу данных и некоторые макросы для упрощения шаблона проектирования, помогающего автору драйвера сопоставлять устройства с его драйвером.

Производители оригинального оборудования (OEM) часто разрабатывают эталонный дизайн для продуктов PC Card, а затем продают этот дизайн другим компаниям для продвижения на рынке. Эти компании дорабатывают дизайн, продвигают продукт для своей целевой аудитории или географического региона и размещают на карте свою собственную торговую марку. Доработки физической карты обычно очень незначительны, если они вообще вносятся. Чтобы усилить свой бренд, такие поставщики указывают название своей компании в читаемых человеком строках в пространстве CIS, но оставляют идентификаторы производителя и продукта без изменений.

Из-за такой практики драйверы FreeBSD обычно полагаются на числовые идентификаторы для распознавания устройств. Использование числовых идентификаторов и централизованной базы данных усложняет добавление ID и поддержку карт в систему. Необходимо тщательно проверять, кто на самом деле произвел карту, особенно когда кажется, что у производителя карты уже может быть другой идентификатор производителя в центральной базе данных. Linksys, D-Link и NetGear — это несколько американских производителей сетевого оборудования, которые часто продают один и тот же дизайн. Эти же дизайны могут продаваться в Японии под такими названиями, как Buffalo и Corega. Часто все эти устройства будут иметь одинаковые идентификаторы производителя и продукта.

Код шины PC Card хранит централизованную базу данных информации о картах, но не о том, какой драйвер с ними связан, в /sys/dev/pccard/pccarddevs. Он также предоставляет набор макросов, которые позволяют легко создавать простые записи в таблице, используемой драйвером для заявки устройств.

Наконец, некоторые устройства очень низкого уровня вообще не содержат идентификации производителя. Эти устройства должны быть обнаружены путем сопоставления читаемых

человеком строк CIS. Хотя было бы хорошо, если бы нам не нужен был этот метод в качестве запасного варианта, он необходим для некоторых очень дешевых CD-плееров и Ethernet-карт. Этот метод, как правило, следует избегать, но ряд устройств перечислен в этом разделе, потому что они были добавлены до осознания OEM-характера бизнеса PC Card. При добавлении новых устройств предпочтительнее использовать числовой метод.

### 16.1.2. Формат файла pccarddevs

В файле pccarddevs есть четыре раздела. Первый раздел содержит номера производителей для вендоров, которые их используют. Этот раздел отсортирован в числовом порядке. Следующий раздел включает все продукты, используемые этими вендорами, вместе с их идентификаторами продуктов и строкой описания. Строка описания обычно не используется (вместо этого мы устанавливаем описание устройства на основе читаемого CIS, даже если совпадение найдено по числовому идентификатору). Затем эти два раздела повторяются для устройств, использующих метод строкового сопоставления. Наконец, C-подобные комментарии, заключённые между символами `/ и/`, допускаются в любом месте файла.

Первая часть файла содержит идентификаторы производителей. Пожалуйста, сохраняйте этот список в числовом порядке. Также, пожалуйста, согласовывайте изменения в этом файле, так как мы делимся им с NetBSD для создания общего центра обработки этой информации. Например, вот первые несколько идентификаторов производителей:

```
vendor FUJITSU          0x0004  Fujitsu Corporation
vendor NETGEAR_2        0x000b  Netgear
vendor PANASONIC        0x0032  Matsushita Electric Industrial Co.
vendor SANDISK          0x0045  Sandisk Corporation
```

Вероятно, запись `NETGEAR_2` на самом деле относится к OEM-производителю, у которого NETGEAR приобретал карты, и автор поддержки этих карт не знал на тот момент, что Netgear использовал чужой идентификатор. Эти записи довольно просты. Ключевое слово `vendor` обозначает тип строки, за которым следует название производителя. Это название будет повторяться позже в pccarddevs, а также использоваться в таблицах соответствия драйверов, поэтому оно должно быть коротким и допустимым идентификатором в C. Числовой идентификатор в шестнадцатеричном формате указывает производителя. Не добавляйте идентификаторы вида `0xffffffff` или `0xffff`, так как они зарезервированы (первый означает "идентификатор не установлен", а второй иногда встречается в крайне некачественных картах для указания "отсутствует"). Наконец, следует строковое описание компании, производящей карту. Эта строка в FreeBSD ни для чего не используется, кроме как в комментариях.

Вторая секция файла содержит продукты. Как показано в этом примере, формат аналогичен строкам поставщиков:

```
/* Allied Telesis K.K. */
product ALLIEDTELESIS LA_PCM    0x0002 Allied Telesis LA-PCM
```

```
/* Archos */
product ARCHOS ARC_ATAPI    0x0043 MiniCD
```

Ключевое слово `product` следует за именем производителя, повторяющимся сверху. После него идёт название продукта, которое используется драйвером и должно быть допустимым идентификатором в C, но также может начинаться с цифры. Как и в случае с производителями, шестнадцатеричный идентификатор продукта для этой карты следует тем же соглашениям для `0xffffffff` и `0xffff`. Наконец, идёт строковое описание самого устройства. Эта строка обычно не используется в FreeBSD, поскольку драйвер шины pccard в FreeBSD формирует строку из читаемых человеком записей CIS, но она может быть использована в редких случаях, когда этого недостаточно. Продукты перечислены в алфавитном порядке по производителю, затем в числовом порядке по идентификатору продукта. Перед записями каждого производителя есть комментарий в C, а между записями — пустая строка.

Третий раздел аналогичен предыдущему разделу производителей, но все числовые идентификаторы производителей установлены в `-1`, что означает "совпадение с любым найденным" в коде шины pccard FreeBSD. Поскольку это идентификаторы C, их имена должны быть уникальными. В остальном формат идентичен первому разделу файла.

Последний раздел содержит записи для тех карт, которые должны быть идентифицированы по строковым значениям. Формат этого раздела немного отличается от общего раздела:

```
product ADDTRON AWP100      { "Addtron", "AWP-100&spWireless&spPCMCIA",
"Version&sp01.02", NULL }
product ALLIEDTELESIS WR211PCM  { "Allied&spTelesis&spK.K.", "WR211PCM", NULL, NULL }
Allied Telesis WR211PCM
```

Знакомое ключевое слово `product` сопровождается названием производителя и именем карты, как и во втором разделе файла. Здесь формат отличается от использованного ранее. Идёт группировка {}, за которой следует несколько строк. Эти строки соответствуют производителю, продукту и дополнительной информации, определённой в кортеже CIS_INFO. Эти строки фильтруются программой, которая генерирует pccarddevs.h, чтобы заменить &sp на реальный пробел. Строки NULL означают, что соответствующую часть записи следует игнорировать. В приведённом здесь примере есть некорректная запись. Она не должна содержать номер версии, если только он не критичен для работы карты. Иногда у производителей может быть множество различных версий карты в обращении, которые все работают, и в таком случае эта информация только затрудняет использование аналогичной карты с FreeBSD. Иногда это необходимо, когда производитель хочет продавать множество различных компонентов под одним брендом из-за рыночных соображений (доступность, цена и т. д.). Тогда это может быть критично для различения карты в тех редких случаях, когда производитель сохранил ту же пару производитель/продукт. На данный момент использование регулярных выражений для сопоставления недоступно.

### 16.1.3. Пример процедуры обнаружения

Чтобы понять, как добавить устройство в список поддерживаемых, необходимо разобраться в процедурах `probe` (обнаружение) и/или `match` (сопоставление), которые есть во многих драйверах. В FreeBSD 5.x это немного сложнее из-за наличия слоя совместимости с OLDCARD. Поскольку различия лишь косметические, здесь будет представлена идеализированная версия.

```
static const struct pccard_product wi_pccard_products[] = {
    PCMCIA_CARD(3COM, 3CRWE737A, 0),
    PCMCIA_CARD(BUFFALO, WLI_PCM_S11, 0),
    PCMCIA_CARD(BUFFALO, WLI_CF_S11G, 0),
    PCMCIA_CARD(TDK, LAK_CD011WL, 0),
    { NULL }
};

static int
wi_pccard_probe(dev)
    device_t    dev;
{
    const struct pccard_product *pp;

    if ((pp = pccard_product_lookup(dev, wi_pccard_products,
        sizeof(wi_pccard_products[0]), NULL)) != NULL) {
        if (pp->pp_name != NULL)
            device_set_desc(dev, pp->pp_name);
        return (0);
    }
    return (ENXIO);
}
```

Вот простая процедура проверки pccard, которая соответствует нескольким устройствам. Как упоминалось выше, название может отличаться (если это не `foo_pccard_probe()`, то это будет `foo_pccard_match()`). Функция `pccard_product_lookup()` является обобщенной функцией, которая проходит по таблице и возвращает указатель на первую запись, которой соответствует. Некоторые драйверы могут использовать этот механизм для передачи дополнительной информации о некоторых картах остальной части драйвера, поэтому в таблице могут быть вариации. Единственное требование — каждая строка таблицы должна содержать `struct pccard_product` в качестве первого элемента.

Рассматривая таблицу `wi_pccard_products`, можно заметить, что все записи имеют вид `PCMCIA_CARD(foo, bar, baz)`. Часть *foo* — это идентификатор производителя из pccarddevs. Часть *bar* — это идентификатор продукта. *baz* — ожидаемый номер функции для этой карты. Многие pccard-устройства могут иметь несколько функций, поэтому требуется способ различать функцию 1 и функцию 0. Вы можете встретить `PCMCIA_CARD_D`, который включает описание устройства из pccarddevs. Также могут встречаться `PCMCIA_CARD2` и `PCMCIA_CARD2_D`, которые используются, когда необходимо сопоставить как строки CIS, так и номера производителей, в вариантах «использовать описание по умолчанию» и «взять описание из pccarddevs».

### 16.1.4. Собираем все вместе

Для добавления нового устройства необходимо сначала получить идентификационную информацию от устройства. Проще всего это сделать, вставив устройство в слот PC Card или CF и выполнив команду `devinfo -v`. Пример вывода:

```
        cbb1 pnpinfo vendor=0x104c device=0xac51 subvendor=0x1265 subdevice=0x0300
class=0x060700 at slot=10 function=1
          cardbus1
          pccard1
            unknown pnpinfo manufacturer=0x026f product=0x030c cisvendor="BUFFALO"
cisproduct="WLI2-CF-S11" function_type=6 at function=0
```

`manufacturer` и `product` являются числовыми идентификаторами данного продукта, в то время как `cisvendor` и `cisproduct` представляют собой строки описания продукта из CIS.

Поскольку мы сначала хотим предпочесть числовой вариант, попробуем сначала создать запись на его основе. Приведённая выше карта была слегка изменена для целей данного примера. Производитель — BUFFALO, у которого, как мы видим, уже есть запись:

```
vendor BUFFALO          0x026f  BUFFALO (Melco Corporation)
```

Но нет записи для этой конкретной карты. Вместо этого мы видим:

```
/* BUFFALO */
product BUFFALO WLI_PCM_S11 0x0305  BUFFALO AirStation 11Mbps WLAN
product BUFFALO LPC_CF_CLT  0x0307  BUFFALO LPC-CF-CLT
product BUFFALO LPC3_CLT    0x030a  BUFFALO LPC3-CLT Ethernet Adapter
product BUFFALO WLI_CF_S11G 0x030b  BUFFALO AirStation 11Mbps CF WLAN
```

Чтобы добавить устройство, мы можем просто добавить эту запись в pccarddevs:

```
product BUFFALO WLI2_CF_S11G    0x030c  BUFFALO AirStation ultra 802.11b CF
```

После выполнения этих шагов карту можно добавить в драйвер. Это простая операция добавления одной строки:

```
static const struct pccard_product wi_pccard_products[] = {
    PCMCIA_CARD(3COM, 3CRWE737A, 0),
    PCMCIA_CARD(BUFFALO, WLI_PCM_S11, 0),
    PCMCIA_CARD(BUFFALO, WLI_CF_S11G, 0),
+   PCMCIA_CARD(BUFFALO, WLI_CF2_S11G, 0),
    PCMCIA_CARD(TDK, LAK_CD011WL, 0),
    { NULL }
};
```

Обратите внимание, что я добавил символ ‘+’ перед строкой, которую добавил, но это только для выделения строки. Не добавляйте его в реальный драйвер. После добавления строки вы можете пересобрать ядро или модуль и протестировать его. Если устройство распознано и работает, отправьте патч. Если оно не работает, определите, что необходимо для его работы, и отправьте патч. Если устройство не распознаётся вообще, вы где-то ошиблись и следует перепроверить каждый шаг.

Если вы коммиттер исходного кода FreeBSD, и всё работает корректно, то можете закоммитить изменения в дерево. Однако есть несколько небольших нюансов, которые следует учесть. pccarddevs должен быть закоммичен в дерево первым. Затем pccarddevs.h необходимо перегенерировать и закоммитить вторым шагом, убедившись, что правильный тег $FreeBSD$ присутствует в последнем файле. В конце закоммитьте добавления в драйвер.

### 16.1.5. Отправка кода для нового устройства

Пожалуйста, не отправляйте записи о новых устройствах автору напрямую. Вместо этого оформите их как PR и сообщите автору номер PR для учёта. Это гарантирует, что записи не будут потеряны. При отправке PR нет необходимости включать в патч diff-файлы pccardevs.h, так как они будут перегенерированы. Однако необходимо включить описание устройства, а также патчи для клиентского драйвера. Если название устройства неизвестно, используйте имя OEM99, и автор скорректирует OEM99 после изучения. Коммиттеры не должны коммитить OEM99, а вместо этого найти наибольший OEM-номер и закоммитить на единицу больше.

# Часть III: Приложения

# Приложение A: Библиография

[1] *Marshall Kirk McKusick, Keith Bostic, Michael J Karels, and John S Quarterman.* Copyright © 1996 Addison-Wesley Publishing Company, Inc.. 0-201-54979-4. Издано Addison-Wesley Publishing Company, Inc.. The Design and Implementation of the 4.4 BSD Operating System. 1-2.