Санкт-Петербургский политехнический университет Петра Великого Институт Информационных Технологий и Управления

Кафедра компьютерных систем и программных технологий

Реферат

по предмету «Проектирование ОС и компонентов»

### Реверс-инжиниринг ОС Minix

Работу выполнил студент гр. 63501/3 Мартынов С. А. Работу принял преподаватель Душутина Е. В.

Санкт-Петербург 2016

**Содержание**

[Введение 3](#_TOC_250012)

1. [Микроядро 5](#_TOC_250011)
   1. [Директория /kernel 5](#_TOC_250010)
   2. [Директория /kernel/system 10](#_TOC_250009)
   3. [Директория /kernel/arch/i386/include 19](#_TOC_250008)
   4. [Директория /kernel/arch/i386 19](#_TOC_250007)
2. [Системный загрузчик 23](#_TOC_250006)
3. [Серверы 25](#_TOC_250005)
   1. [Виртуальная файловая система /servers/vfs 25](#_TOC_250004)
   2. [Межпроцессное взаимодействие /servers/ipc 30](#_TOC_250003)
   3. [Управление процессами /servers/pm 31](#_TOC_250002)
   4. [Виртуальная память /servers/vm 36](#_TOC_250001)

[Заключение 40](#_TOC_250000)

# Введение

Реверс-инжиниринг ПО (обратный инжиниринг, обратная разработка) это процесс восста- новления структуры, внутреннего устройства программы с целью понимания его принципа её работы.

Мотивацией проведения реверс-инжиниринга может являться извлечение спецификации, моделей (выявление архитектуры программы, получение алгоритмов работы, извлечение поведенческих моделей), понимание работы программы, деобфускация или даже восста- новление исходного кода. Необходимость в реверс-инжиниринг может возникнуть когда возникают задачи поддержки legacy кода, при подготовке к реинжинирингу либо когда нужно скопировать какую-то функциональность ПО без нарушения авторских прав. По- следний пункт требует отдельного изучения, т.к. иногда производитель ПО сопровождает свой продукт лицензией не допускающей его реверс-инжиниринга (к примеру Skype).

Предметом анализа реверс-инжиниринга обычно является:

* Исходный код ПО
* Бинарный код ПО
* Байт-код
* Программная документация

Результатом проведения реверс-инжиниринга может быть:

* Диаграммы классов
* Диаграммы компонентов
* Диаграммы модулей
* Диаграммы состояний
* Диаграммы последовательностей
* Схемы алгоритмов
* Модели данных
* Протокол исследования
* Любые другие текстовые и графические данные, соответствующие поставленной задаче

В зависимости от цели и от того, что есть в наличии у исследователя можно выбрать соответствующий инструмент. К примеру, если есть исходный код какого-то протокола

обмена, то средствами статического анализа (метод обратной трассировки) можно полу- чить диаграмму состояний. Если есть бинарный (исполняемый) код, то можно провести динамический анализ – запротоколировать и визуализировать трассы исполнения.

В данной работе мы проведём реверс-инжиниринг свободной (лицензия BSD) Unix-подобной микроядерной операционной системы Minix. Изначально (1987 год) ядро Minix состояло из 1 600 строк на С и 800 ассемблерных строк. Ко второй версии (1997 год) ее размер вырос до 62 200 строк. На данный момент последней стабильной версией является 3.3.0 (16 сентября 2014) и её объём составляет 1 415 811 строк в .c и .h файлах.

Нашей целью является реверс-инжиниринг текущей версии Minix для быстрой ориентации по исходному коду. Нами будет рассмотрено микроядро (директория /kernel), серверы (директория /servers), некоторые включаемые файлы (директория /include/), системный загрузчик (директория /boot). Из рассмотрения были исключены системные и пользова- тельские библиотеки (директория /lib), тесты (директория /test) и системные утилиты (директория /commands) т.к. они не являются частями собственно операционной системы. Драйверы содержат много специфичной информации, а вспомогательные файлы (дирек- тории /etc/ и /tools/) не требуют особых пояснений. Так же была исключён раздел с документацией (директории /man/ и /docs/) т.к. они не содержат файлов с исходным кодом.

В своей работе мы будем использовать стандартные инструменты разработчика – утилиты grep, find, cat и редактор vim с набором плагинов для подсветки кода.

# Микроядро

## Директория /kernel

Директория /kernel содержит следующие файлы:

**clock.c (12K)** – Содержит функции для инициализации таймера и обработчиков таймера. Содержит как обработчик аппаратного прерывания, так и бесконечный цикл для обработ- ки событий. Важные события, обрабатываемые посредством CLOCK, включают также решения по планированию и перепланированию процессов. CLOCK предлагает непосред- ственный интерфейс с процессами микроядра. Системные службы могут обращаться (к данному файлу) посредством системных вызовов, таких как sys\_setalarm().

При видоизменении данного файла нельзя использовать send() если получатель не готов принять сообщение. Вместо этого желательно использовать notify().

CLOCK – это один из процессов микроядра, не вынесенный в отдельный сервер по сообра- жениям производительности.

**clock.h (4,0K)** – Определяет функции для инициализации таймера и обработчиков тайме- ра.

**config.h (4,0K)** – Определяет конфигурацию микроядра. Позволяет установить размеры буферов ядра, включить или исключить отладочный код, функции контроля времени и отдельные вызовы микроядра. (поэтому здесь содержаться краткие их описания, тем не менее настоятельно рекомендуется сохранять все вызовы микроядра включёнными).

Этот файл по логике должен быть одним из основных координирующих файлов, однако похоже, что эта функция уже давно стала атавизмом...

**const.h (4,0K)** – Содержит макросы и константы, используемые в коде микроядра. В частности, содержит макросы для запрещения и разрешения аппаратных прерываний.

Общие определения и макросы помещаются в этом файле.

**debug.c (12K)** – Этот файл содержит отладочные функции, не включённые в стандартное микроядро. Доступные функции включают отсчёт времени для блокировок и проверочные функции для очередей управления.

**debug.h (4,0K)** – Определяет все отладочные константы и макросы,а также некоторые (глобальные) переменные. Некоторые отладочные функции требуют переопределения стандартных констант и макросов, поэтому данный заголовочный файл должен находится ПОСЛЕ остальных заголовочных файлов микроядра.

**glo.h (4,0K)** – Определяет используемые в микроядре глобальные переменные (сами переменные размещаются в table.o). На данный момент определяет:

* + - переменные режима ядра (исключение, выход из системы);
    - переменные-структуры информации о ядре;
    - диагностические сообщения;
    - генератора случайных чисел, средней загрузке;
    - указатель на текущий выполняющийся процесс;
    - указатель на следующий процесс, после restart();
    - указатель на процесс для подсчёта тиков;
    - указатель на первые процессы в очередях restart, request, pagefault;
    - переменную учёта тиков, не учтённых в задании CLOCK.

**interrupt.c (8,0K)** – Система аппаратных прерываний Minix3. Содержит процедуры для управления контроллером прерываний:

* + - регистрации/удаления обработчика прерываний (PUBLIC void irq\_handle(int irq) вызывается системно зависимой частью при возникновении внешнего прерывания);
    - разрешения/запрещения линии прерываний.

Определяет переменную:

PUBLIC irq\_hook\_t\* irq\_handlers[NR\_IRQ\_VECTORS] = {0};

**interrupt.h (4,0K)** – Прототипы для системы аппаратных прерываний.

**ipc.h (4,0K)** – Этот файл определяет константы для межпроцессного взаимодействия. Определения используются в /kernel/proc.c. Пока это константы NON\_BLOCKING, SEND, RECEIVE, SENDREC, NOTIFY, SENDNB, SENDA а также макрос WILLRECEIVE(target,

source\_ep).

**kernel.h (4,0K)** – Основной заголовочный файл микроядра Minix3. Однако сам по себе он состоит только из заголовков. В частности, он включает почти все заголовочные файлы из /kernel/ .

**main.c (16K)** – Описывает начальный старт микроядра (оно находится ещё в загрузочном образе и уже имеет набор готовых к исполнению системных процессов – серверов).

**priv.h (4,0K)** – Определяет структуру системы привилегий struct priv. Каждый системный

процесс имеет собственную структуру привилегий, для всех пользовательских процессов используется одна структура привилегий:

#define USER\_PRIV\_ID 0

**proc.c (60K)** – Вместе с mpx.s этот файл содержит наиболее низкоуровневую часть микроядра.

Содержит точку входа для внешних вызовов: sys\_call – системный вызов, когда мы попада- ем в микроядро посредством INT. Имеется также несколько точек входа для прерываний и уровня заданий: lock\_send – послать сообщение процессу.

Этот файл очень большой и требует дополнительного рассмотрения низкоуровневых функций.

Следует только заметить, что PROC является одним из процессов микроядра не вынесен- ным в отдельный сервер по соображениям производительности.

**proc.h (12K)** – Определяет таблицу процессов. Таблица процессов включает

* + - состояние регистров и флагов,
    - приоритет планировщика,
    - таблица памяти,
    - различные учётные данные,
    - информацию, используемую для передачи сообщений (IPC).

Многие ассемблерные процедуры обращаются к полям этой структуры. Смещения опреде- лены в ассемблерном включаемом файле kernel/arch/i386/sconst.h. Эти два файла должны соответствовать друг другу!

Кроме того определены флаги исполнения (runtime) и макросы, для их проверки, установки, очистки.

EXTERN struct proc proc[NR\_TASKS + NR\_PROCS]; /\* process table \*/

EXTERN struct proc \*rdy\_head[NR\_SCHED\_QUEUES]; /\* ptrs to ready list headers \*/ EXTERN struct proc \*rdy\_tail[NR\_SCHED\_QUEUES]; /\* ptrs to ready list tails \*/

**profile.c (8,0K)** – Этот файл содержит несколько функций и переменных, используемых для системного профилирования: статистическое профилирование (обработчик преры- ваний для часов профилирования) и профилирование вызовов (таблица, используемая для данных профилирования, а также функция для определения её размеров; функция

используется процессами микроядра для регистрации их управляющих структур и таблицы профилирования).

**profile.h (4,0K)** – Определяет переменные для профилирования. Зависит от /include/minix/profile.h Содержит общую опцию

#if SPROFILE /\* statistical profiling \*/

**proto.h (8,0K)** – Файл содержит все прототипы функций (публичного интерфейса), опре- делённых в файлах clock.c, main.c, utility.c, proc.c, start.c, system.c, system/do\_newmap.c, system/do\_vtimer.c, interrupt.c, debug.c, system/do\_safecopy.c, system/do\_sysctl.c, profile.c директории /kernel, включая части, зависящие от конкретной платформы.

**smp.c (8,0K)** – архитектурно-зависимая реализация режима мультипроцессорности.

**smp.h (4,0K)** – заголовок для симметричной мультипроцессорности.

### spinlock.h (4,0K)

**system.c (24,0K)** – Обеспечивает интерфейс между микроядром и серверами системных вызовов. Для отображения системных вызовов на функции, их обеспечивающие приме- няется внутренний вектор вызовов. Сами эти функции находятся в отдельных файлах (/kernel/system/). Вектор вызовов используется в основном цикле системного задания для обработки входящих запросов.

Кроме основной точки входа (sys\_task()), в котором и запускается основной цикл, имеется ещё несколько точек входа:

* + - get\_priv – заполняет структуру привилегий для пользовательского или системного процесса,
    - set\_sendto\_bit – позволяет процессу послать сообщение в новом направлении (рас- ширяет привилегии),
    - unset\_sendto\_bit – запрещает процессу послать сообщение в новом направлении (сужает привилегии),
    - send\_sig – посылает сигнал прямо системному процессу,
    - cause\_sig – выполняет действие вызванное сигналом вызывая событие через сервер управления процессами,
    - sig\_delay\_done – сообщает серверу управления процессами что процесс не посылает,
    - umap\_bios – отображает виртуальный адрес в BIOS\_SEG на физический,
    - get\_randomness – накапливает случайности в буфер,
    - clear\_endpoint – лишает процесс возможности посылать и принимать сообщения.

SYS является одним из заданий микроядра.

**system.h (8,0K)** – Прототипы функций системной библиотеки. Сами функции нахо- дятся в /kernel/system/. Если вызов микроядра не включён посредством конфигурации

/kernel/config.h, то функция становится синонимом do\_unused().

Системная библиотека делает доступным системный сервис посредством вызовов микрояд- ра. Системные вызовы трансформируются в сообщения-запросы к заданию SYS, способному выполнить соответствующий вызов.

По соглашению sys\_call() преобразуется в сообщение с типом SYS\_CALL, которое обраба- тывается функцией do\_call().

**table.c (4,0K)** – Содержит большинство данных микроядра. Непосредственно в данном файле определены

* + - константы препроцессора:
      * для размеров стеков заданий микроядра,
      * флаги для различных типов процессов (микроядра),
      * списки FS\_C и DRV\_C (разрешённых высовов микроядра),
    - макросы препроцессора:
      * для определения масок системных вызовов для различных типов процессов,
    - глобальные переменные:

**–** fs\_c[], pm\_c[], rs\_c[], ds\_c[], vm\_c[], drv\_c[], usr\_c[], tty\_c[], mem\_c[],

* + - * PUBLIC struct boot\_image image[]

EXTERN внутри данного файла принимает значение пустой строки, поэтому глобальные переменные определённые в заголовочных файлах реально привязываются к данному файлу (место им выделяется в table.o).

**type.h (4,0K)** – Определяет типы, связанные с таблицей процессов и другими свойствами (переменными) системы (микроядра): task\_t, proc\_nr\_t, sys\_id\_t, sys\_map\_t, struct boot\_image, irq\_policy\_t, irq\_id\_t, struct irq\_hook, irq\_hook\_t, irq\_handler\_t.

### usermapped\_data.c (4,0K)

**utility.c (4,0K)** – Этот файл содержит коллекцию различных процедур:

* + - minix\_panic – прерывает MINIX в связи с фатальной ошибкой,
    - kputc – буферизированный putc, используемый функцией kprintf,
    - kprintf – посредством включения файла lib/sysutil/kprintf.c и замены printf на kprintf посредством препроцессора.

**vm.h (4,0K)** – Содержит константы препроцессора: VMSUSPEND, EFAULT\_SRC, EFAULT\_DST и макросы: FIXLINMSG(prp), PHYS\_COPY\_CATCH(src, dst, size, a).

Файл связан с механизмами виртуальной памяти.

**watchdog.c (4,0K)** – реализация архитектурно-зависимого механизма watchdog, исполь- зуемого для выявления ошибок в ядре

**watchdog.h (4,0K)** – прототип механизма watchdog

## Директория /kernel/system

**do\_abort.c (4,0K)** – Реализуется вызов микроядра SYS\_ABORT (прерывание работы OS Minix3)

Параметры:

* + - ABRT\_HOW – как выполнить прерывание работы OS Minix3)
    - ABRT\_MON\_ENDPT – номер процесса параметры монитора которого берутся
    - ABRT\_MON\_LEN – длина параметров монитора
    - ABRT\_MON\_ADDR – виртуальный адрес параметров

**do\_copy.c (4,0K)** – Реализуется вызовы микроядра SYS\_VIRCOPY (копирование обла- стей виртуальной памяти), SYS\_PHYSCOPY (копирование областей физической памяти).

Параметры:

* + - CP\_SRC\_SPACE – виртуальный сегмент копируемых данных
    - CP\_SRC\_ADDR – смещение копируемых данных относительно сегмента
    - CP\_SRC\_PROC\_NR – номер процесса из виртуальной памяти которого происходит копирование
    - CP\_DST\_SPACE – виртуальный сегмент, куда копируются данные
    - CP\_DST\_ADDR – смещение относительно виртуального сегмента, куда копируются данные
    - CP\_DST\_PROC\_NR – номер процесса, в виртуальное адресное пространство кото- рого копируются данные
    - CP\_NR\_BYTES – размер копируемых данных в байтах

**do\_cprofile.c (8,0K)** – Реализуется вызов микроядра SYS\_CPROFILE (профилирование вызовов).

Параметры:

* + - PROF\_ACTION – получить/сбросить данные профилирования
    - PROF\_MEM\_SIZE – доступная память для данных
    - PROF\_ENDPT – узловая точка вызывающего
    - PROF\_CTL\_PTR – расположение информационной структуры
    - PROF\_MEM\_PTR – местоположение памяти для данных

**do\_devio.c (4,0K)** – Реализуется вызов микроядра SYS\_DEVIO (Осуществляет низко- уровневый ввод/вывод в порты ввода/вывода)

Параметры:

* + - DIO\_REQUEST – запрос на ввод или вывод
    - DIO\_PORT – порт для ввода/вывода
    - DIO\_VALUE – возвращает прочтённое значение

**do\_mapdma.c (4,0K)** – Реализуется вызов микроядра SYS\_MAPDMA (Выделяет об- ласть для выполнения операции устройства с непосредственным доступом к памяти.)

Параметры:

* + - CP\_SRC\_ADDR – Виртуальный адрес.
    - CP\_NR\_BYTES – Размер структуры данных.

**do\_endksig.c (4,0K)** – Реализуется вызов микроядра SYS\_ENDKSIG (Вызывается сер- вером управления процессами (PM) после обработки сигнала процессу SYS\_GETKSIG. Обычно это сигнал прерывания процесса.)

Параметры:

* + - SIG\_ENDPT – Процесс, для которого выполнено задание сервера управления про- цессами (PM)

**do\_exec.c (4,0K)** – Реализуется вызов микроядра SYS\_EXEC (Замена контекста процес- са.)

Параметры:

* + - PR\_ENDPT – Процесс, вызвавший exec
    - PR\_STACK\_PTR – Новый указатель на стек
    - PR\_NAME\_PTR – Указатель на имя программы
    - PR\_IP\_PTR – Новый указатель инструкций

**do\_exit.c (4,0K)** – Реализуется вызов микроядра SYS\_EXIT (Завершает процесс.) Параметры:

* + - PR\_ENDPT – Номер слота завершающегося процесса.

**do\_fork.c (8,0K)** – Реализуется вызов микроядра SYS\_FORK (Создание нового процесса

* копии родительского.) Параметры:
  + PR\_ENDPT – Родитель, процесс, который разветвляется
  + PR\_SLOT – Слот порождаемого процесса-ребёнка в таблице процессов.
  + PR\_MEM\_PTR – Новая карта памяти для процесса-ребёнка.
  + PR\_FORK\_FLAGS – Флаги-параметры вызова fork.

**do\_getinfo.c (8,0K)** – Реализуется вызов микроядра SYS\_GETINFO (Запрос на систем- ную информацию, которая копируется в адресное пространство запрашивающего процесса. Этот вызов просто копирует соответствующие структуры данных запрашивающему про- цессу.)

Параметры:

* + I\_REQUEST – Какую информацию?
  + I\_VAL\_PTR – Куда её поместить?
  + I\_VAL\_LEN – Максимальная возможная длина
  + I\_VAL\_PTR2 – Второй параметр (может не быть)
  + I\_VAL\_LEN2\_E – Вторая длина или номер процесса.

**do\_getksig.c (4,0K)** – Реализуется вызов микроядра: SYS\_GETKSIG (Сервер управления процессами (PM) готов обрабатывать сигналы и периодически делает вызов микроядра для получения очередного сигнала.)

Параметры:

* + SIG\_ENDPT – Процесс, посылающий сигнал
  + SIG\_MAP – Набор битов сигнала

**do\_irqctl.c (8,0K)** – Реализуется вызов микроядра SYS\_IRQCTL (Позволяет , в частно- сти, вставить новый обработчик прерываний. Возвращает индекс ловушки прерывания, назначенный в микроядре.)

Параметры:

* + IRQ\_REQUEST – Контрольная операция, которую надо выполнить.
  + IRQ\_VECTOR – Линия прерываний, которая должна быть проверена.
  + IRQ\_POLICY – Позволяет вновь разрешить прерывания.
  + IRQ\_HOOK\_ID – Предоставляет индекс, который будет возвращён при прерывании.

**do\_kill.c (4,0K)** – Реализуется вызов микроядра SYS\_KILL (Обеспечивает sys\_kill(). Вызывает посылку сигнала процессу. Сервер управления процессами (PM) – централь- ный сервер, где обрабатываются все сигналы и обеспечиваются регистрация порядка их обработки. Любой запрос, за исключением запросов сервера управления процессами (PM), добавляется в "карту"необработанных сигналов, а сервер управления процессами (PM) информируется о поступлении нового сигнала. Так как системные серверы не могут ис- пользовать нормальные POSIX сигналы (ввиду того, что они обычно блокируют процесс на их получении), они могут запросить сервер управления процессами (PM) преобразовать сигналы в сообщения. Это выполняется сервером управления процессами (PM) посредством вызова sys\_kill().)

Параметры:

* + SIG\_ENDPT – процесс, которому посылается сигнал/ необработанный
  + SIG\_NUMBER – Номер сигнала, который посылается процессу.

### do\_mcontext.c (4,0K)

**do\_memset.c (4,0K)** – Реализуется вызов микроядра SYS\_MEMSET (Записывает обра- зец в определённый участок памяти.)

Параметры:

* + MEM\_PTR – виртуальный адрес
  + MEM\_COUNT – возвращает физический адрес
  + MEM\_PATTERN – байт-образец, которым заполняется область

**do\_privctl.c (12K)** – Реализуется вызов микроядра SYS\_PRIVCTL (Обновляет приви- легии процесса. Если процесс пока не является системным процессом, выделяет ему его собственную структуру привилегий.)

Параметры:

* + CTL\_ENDPT – Точка окончания целевого процесса.
  + CTL\_REQUEST – Запрос контроля привилегий.
  + CTL\_ARG\_PTR – Указатель на запрашиваемые данные.

**do\_profbuf.c (4,0K)** – Реализуется вызов микроядра SYS\_PROFBUF (При помощи данного вызова микроядра профилируемые процессы и информируют микроядро о ме- стоположении их таблицы профилирования и контрольной структуры. Вызов микроядра используется системой профилирования когда установлена опция профилирования вызо- вов.)

Параметры:

* + PROF\_CTL\_PTR – Местоположение контрольной структуры.
  + PROF\_MEM\_PTR – Местоположение таблицы профилирования.

**do\_runctl.c (4,0K)** – Реализуется вызов микроядра SYS\_RUNCTL Контролирует флаги PROC\_STOP процесса. Используется для управления процессами. В некоторых случаях устанавливает MF\_SIG\_DELAY вместо PROC\_STOP. Используется сервером управления процессами (PM) для надёжности управления сигналами.)

Параметры

* + RC\_ENDPT – Номер контролируемого процесса.
  + RC\_ACTION – Останавливает или восстанавливает исполнение процесса.
  + RC\_FLAGS – Флаги запроса.

**do\_safecopy.c (12K)** – Реализуется вызовы микроядра SYS\_SAFECOPYFROM, SYS\_SAFECOPY SYS\_VSAFECOPY (Безопасное копирование. Копирование областей памяти с контролем разрешений.)

Параметры:

* + SCP\_FROM\_TO – другая точка окончания
  + SCP\_INFO – находящийся в собственности вызывающего процесса сегмент из/в который происходит копирование.
  + SCP\_GID – Идентификатор разрешения
  + SCP\_OFFSET – Смещение внутри разрешённой области.
  + SCP\_ADDRESS – Адрес в собственном адресном пространстве.
  + SCP\_BYTES – Размер копируемой области в байтах.

**do\_nice.c (4,0K)** – Реализуется вызов микроядра SYS\_NICE (Изменяет приоритет про- цесса или прекращает выполнение процесса.)

Параметры:

* + PR\_ENDPT – Номер процесса, приоритет которого изменяется.
  + PR\_PRIORITY – Новый приоритет.

**do\_segctl.c (4,0K)** – Реализуется вызов микроядра SYS\_SEGCTL (Возвращает пере- ключатель сегмента и смещение, которые могут быть использованы для достижения физических адресов, для использования в драйверах выполняющих отображённый на память ввод/вывод в области A0000 – DFFFF.)

Параметры:

* + SEG\_PHYS – Базовый физический адрес.
  + SEG\_SIZE – Размер сегмента.
  + SEG\_SELECT – Возвращает переключатель сегмента.
  + SEG\_OFFSET – Возвращает смещение внутри сегмента.
  + SEG\_INDEX – Возвращает индекс опосредованной памяти. **do\_sysctl.c (4,0K)** – Реализуется вызов микроядра SYS\_SYSCTL. Параметры:
  + SYSCTL\_CODE – Запрос.
  + SYSCTL\_ARG1 – Специфические для запроса аргументы.
  + SYSCTL\_ARG2 – Специфические для запроса аргументы.

**do\_setalarm.c (4,0K)** – Реализуется вызов микроядра SYS\_SETALARM (Выполняет запрос на синхронный сигнал, или на отмену синхронного сигнала.)

Параметры:

* + ALRM\_EXP\_TIME – Время до подачи сигнала.
  + ALRM\_ABS\_TIME – Абсолютное время до подачи сигнала.
  + ALRM\_TIME\_LEFT – Возвращает секунды, прошедшие от предыдущего сигнала.

**do\_setgrant.c (4,0K)** – Реализуется вызов микроядра SYS\_SETGRANT (Устанавливает разрешения.)

Параметры:

* + SG\_ADDR – Адрес таблицы разрешений в собственном адресном пространстве.
  + SG\_SIZE – Число записей таблицы

**do\_sigreturn.c (4,0K)** – Реализуется вызов микроядра SYS\_SIGRETURN (Запрос в стиле сигналов POSIX. Требует, чтобы sys\_sigreturn упорядочил всё прежде, чем сигнали- зирующий процесс мог снова выполняться.)

Параметры:

* + SIG\_ENDPT – Процесс, возвращающийся из обработки
  + SIG\_CTXT\_PTR – Указатель на структуру контекстов сигналов

**do\_sigsend.c (8,0K)** – Реализуется вызов микроядра SYS\_SIGSEND (Обеспечение сиг- налов в стиле POSIX.)

Параметры:

* + SIG\_ENDPT – Процесс для вызова обеспечения сигнала
  + SIG\_CTXT\_PTR – Указатель на структуру контекста сигналов
  + SIG\_FLAGS – Флаги для вызова S\_SIGRETURN.

**do\_sprofile.c (4,0K)** – Реализуется вызов микроядра SYS\_SPROFILE (Обеспечивает статистическое профилирование.)

Параметры:

* + PROF\_ACTION – Начинает/прекращает профилирование.
  + PROF\_MEM\_SIZE – Доступная память для данных.
  + PROF\_FREQ – Частота запрашиваемого образца.
  + PROF\_ENDPT – Конечная точка запрашивающего.
  + PROF\_CTL\_PTR – Местоположение информационной структуры.
  + PROF\_MEM\_PTR – Местоположение памяти для данных.

**do\_stime.c (4,0K)** – Реализуется вызов микроядра SYS\_STIME (Системное время) Параметры:

* + T\_BOOTITME – Время с момента загрузки системы

**do\_times.c (4,0K)** – Реализуется вызов микроядра SYS\_TIMES (Устанавливает инфор- мацию о времени в сообщение. Прерывание часов (таймера) может обновить системное время, что не мешает данному коду.)

Параметры:

* + T\_ENDPT – Получает информацию для данного процесса.

**do\_trace.c (8,0K)** – Реализуется вызов микроядра SYS\_TRACE (Обеспечивает отладоч- ную трассировку.)

Параметры:

* + CTL\_ENDPT – Трассируемый процесс.
  + CTL\_REQUEST – Запрос трассирования.
  + CTL\_ADDRESS – Адрес в пространстве трассируемого процесса.
  + CTL\_DATA – Данные, которые должны быть записаны, или место для возвращаемых данных.

**do\_umap.c (4,0K)** – Реализуется вызов микроядра SYS\_UMAP (Создаёт карту отобра- жения виртуальных адресов на физические)

Параметры:

* + CP\_SRC\_PROC\_NR – Номер процесса.
  + CP\_SRC\_SPACE – Сегмент, где находится адрес: T (код), D (данные), или S (стек).
  + CP\_SRC\_ADDR – Виртуальный адрес.
  + CP\_DST\_ADDR – Возвращает физический адрес.
  + CP\_NR\_BYTES – Размер структуры данных.

**do\_newmap.c (8,0K)** – Реализуется вызов микроядра SYS\_NEWMAP (Создаёт новую карту памяти)

Параметры:

* + PR\_ENDPT – устанавливает новую карту памяти для этого процесса
  + PR\_MEM\_PTR – указатель на новую карту памяти

**do\_vcopy.c (8,0K)** – Реализуется вызовы микроядра SYS\_VIRVCOPY, SYS\_PHYSVCOPY (Копирования физической/виртуальной памяти.)

Параметры:

* + VCP\_VEC\_SIZE – Размер вектора, запрашиваемого на копирование.
  + VCP\_VEC\_ADDR – Адрес вектора (в адресном пространстве запрашивающего процесса.
  + VCP\_NR\_OK – Число успешно скопированных байт, или элементов вектора.

**do\_vdevio.c (8,0K)** – Реализуется вызов микроядра SYS\_VDEVIO (Выполняет серию операций с устройствами ввода/вывода от имени процесса (а не задания микроядра). Адреса ввода/вывода и значения ввода/вывода получаются или возвращаются в некоторый буфер в адресном пространстве процесса. Реальный ввод/вывод обрамлен lock() и unlock(), во избежание прерываний. Является вызовом, родственным do\_devio, выполняющим одиночную операцию с устройством ввода/вывода.)

Параметры:

* + DIO\_REQUEST – Запрос на ввод или вывод.
  + DIO\_VEC\_ADDR – Указатель на пару порт/значение.
  + DIO\_VEC\_SIZE – Число портов для ввода/вывода.

**do\_vmctl.c (8,0K)** – Реализуется вызов микроядра SYS\_VMCTL (Обеспечивает нужды виртуальной памяти)

Параметры:

* + SVMCTL\_WHO – Какой процесс?
  + SVMCTL\_PARAM – Устанавливает имя параметра
  + SVMCTL\_VALUE – Устанавливает значение параметра **do\_vtimer.c (4,0K)** – Реализуется вызов микроядра SYS\_VTIMER Параметры:
  + VT\_WHICH – Таймер: VT\_VIRTUAL или VT\_PROF
  + VT\_SET – Установить или просто получить?
  + VT\_VALUE – Новое/старое время достижения в тиках.
  + VT\_ENDPT – Процесс, которому принадлежит таймер.

## Директория /kernel/arch/i386/include

**archconst.h (8,0K)** – Содержит константы для защищённого режима процессора i386.

**archtypes.h (4,0K)** – Определяет типы и структуры для регистров процессора, сегмент- ного дескриптора защищённого режима процессора, страничных исключений

## Директория /kernel/arch/i386

**arch\_do\_vmctl.c (4,0K)** – Реализуется вызов микроядра SYS\_VMCTL (Обеспечивает нужды виртуальной памяти. Архитектурно зависимая часть.)

Параметры:

* + - SVMCTL\_WHO – Какой процесс?
    - SVMCTL\_PARAM – Устанавливает имя параметра
    - SVMCTL\_VALUE – Устанавливает значение параметра

**clock.h (4.0K)** – Прототипы функций инициализации, сброса и чтения значения счётчика системного таймера 8253A. Архитектурно зависимая часть.

**clock.c (4.0K)** – Реализует функции:

PUBLIC int init\_8253A\_timer(unsigned freq)

Инициализирует канал 0 таймера 8253A устанавливая частоту 60 гц, регистрирует обра- ботчик прерываний задания CLOCK для выполнения каждый тик.

PUBLIC void stop\_8253A\_timer(void)

Сбрасывает частоту таймера на значение BIOS (Для перезагрузки.)

PUBLIC clock\_t read\_8253A\_timer(void)

Считывает счётчик по каналу 0 таймера 8253A. Счётчик отсчитывает в обратном порядке с частотой TIMER\_FREQ и возвращается в значение TIMER\_COUNT-1 , когда достигает нулевого значения. Аппаратное прерывание (тик) возникает в момент, когда счётчик достигает нулевого значения и возобновляет свой цикл.

**do\_int86.c (4.0K)** – Реализуется вызов микроядра SYS\_INT86. Параметры:

* + - INT86\_REG86

**do\_iopenable.c (4.0K)** – Реализуется вызов микроядра SYS\_IOPENABLE. Параметры:

* + - IO\_ENDPT – Процесс, которому устанавливаются биты уровня защиты ввода/вывода.

**do\_readbios.c (4.0K)** – Реализуется вызов микроядра SYS\_READBIOS (Получает дан- ные BIOS.)

Параметры:

* + - RDB\_SIZE – Число байт, которые надо скопировать
    - RDB\_ADDR – Абсолютный адрес в зоне BIOS.
    - RDB\_BUF – Адрес буфера в запрашивающем процессе.

**do\_sdevio.c (4.0K)** – Реализуется вызов микроядра SYS\_SDEVIO (Доступ к портам ввода/вывода.)

Параметры:

* + - DIO\_REQUEST – Запрос на вывод или ввод.
    - DIO\_PORT – Порт для чтения/записи.
    - DIO\_VEC\_ADDR – Виртуальный адрес буфера или ID разрешения.
    - DIO\_VEC\_SIZE – Число элементов.
    - DIO\_VEC\_PROC – Процесс, в котором находится буфер.
    - DIO\_OFFSET – Смещение в разрешении.

**exception.c (4.0K)** – Этот файл содержит простой обработчик исключений. Исключения в пользовательских процессах преобразуются в сигналы. Исключения в заданиях микроядра вызывают панику (и прерывание работы Minix3).

**i8259.c (8.0K)** – Этот файл содержит процедуры для инициализации контроллера преры- ваний 8259:

* + - put\_irq\_handler – регистрирует обработчик прерываний
    - rm\_irq\_handler – удаляет обработчик прерываний из регистра
    - intr\_handle – обрабатывает аппаратное прерывание,
    - intr\_init – осуществляет инициализацию контроллера (контроллеров) прерываний.

**klib386.S (8.0K)** – Этот файл содержит ассемблерный код процедур, необходимых для микроядра:

void monitor() /\* выйти из Minix и возвратиться в монитор\*/ void int86() /\* позволить монитору выполнить прерывание 8086\*/ exit /\* заглушки для библиотечных процедур \*/

main /\* заглушка для GCC \*/

void phys\_insw(Port\_t port, phys\_bytes buf, size\_t count) /\* перемещает данные с (кон void phys\_insb(Port\_t port, phys\_bytes buf, size\_t count) /\* тоже – по байтам void phys\_outsw(Port\_t port, phys\_bytes buf, size\_t count) /\* перемещает данны void phys\_outsb(Port\_t port, phys\_bytes buf, size\_t count) /\* тоже – по байтам phys\_bytes phys\_copy(phys\_bytes source, phys\_bytes destination, phys\_bytes bytecount) phys\_copy\_fault /\*точка входа: страничное исключение phys\_copy\*/ phys\_copy\_fault\_in\_kernel /\* точка входа: страничное исключение phys\_copy в ядре\*/ phys\_memset(phys\_bytes source, unsigned long pattern, phys\_bytes bytecount) /\* u16\_t mem\_rdw(U16\_t segment, u16\_t \*offset) /\* копирует одно слово с [segment: void reset() /\* reset the system – системный сброс\*/

idle\_task /\* точка входа: задание выполняемое когда нет работы \*/ void level0(void (\*func)(void)) /\* вызвать функцию на уровне 0 \*/ unsigned long read\_cpu\_flags(void) /\* прочесть флаги процессора \*/

Эти процедуры гарантирую сохранение только тех регистров, сохранение которых требует компилятор C: (ebx, esi, edi, ebp, esp, сегментные регистры, регистр флагов).

**memory.c (16K)** – Внутренние функции виртуальной памяти (VM)

**mpx386.S (32K)** – Файл mpx386.s включается mpx.s когда Minix компилируется для 32-х битного процессора Intel. Альтернативный mpx88.s включается для 16-битного процессора.

Этот файл является наиболее низкоуровневой частью Minix (наряду с /kernel/proc.c). Здесь осуществляется переключение процессов и поддержка обработки сообщений. Кроме того здесь содержаться обработчики 32-х битных прерываний и ассемблерный стартовый код (обеспечивает кооперацию с /kernel/start.c для обеспечения хорошего начального окружения для main()).

Каждый переход в микроядро происходит через данный файл. Входы в микроядро могут быть вложенными. Начальный вход может быть вызван системным вызовом (вызовом микроядра – при попытке послать или получить сообщение), исключением или аппаратным

прерыванием; вложенный (повторный) вход в микроядро может осуществляться только аппаратными прерываниями. Число вхождений в микроядра (степень вложенности) хра- нится в переменной k\_reenter. Важными моментами являются переключение на стек ядра и защита кода передачи сообщений /kernel/proc.c.

**protect.c (8.0K)** – Этот файл содержит код для инициализации защищённого режима процессора, инициализации дескрипторов сегментов кода и данных и для инициализации глобальных дескрипторов для локальных дескрипторов в таблице процессов.

**proto.h (4.0K)** – Содержит прототипы для:

* + - обработчиков аппаратных прерываний,
    - обработчиков исключений,
    - обработчиков программных прерываний,
    - прототипов функций из файлов
      * /kernel/arch/i386/memory.c
      * /kernel/arch/i386/exception.c
      * /kernel/arch/i386/klib386.S
      * /kernel/arch/i386/protect.c
    - прототипов для работы с таблицей векторов прерываний.

**sconst.h (4.0K)** – В данном файле содержаться различные константы, используемые в ассемблерном коде:

* + - размер машинного слова,
    - смещения в struct proc (должен быть согласован с /kernel/proc.h),
    - смещение на указатель текущего процесса сразу после прерывания, в предположе- нии, что мы всегда имеем код ошибки в стеке, макросы (ассемблерного характера) связанные с сохранением контекста (в случае аппаратных прерываний).

**system.c (16K)** – Системно зависимые функции для использования в микроядре в целом.

# Системный загрузчик

В Minix используется очень простой загрузчик. Все файлы представлены ниже.

**bootblock.s (8.0K)** – Проверка диска, а также пример загрузочного сектора.

**a.out2com (4.0K)** – Скрипт, превращающий Minix a.out файл в com-файл. У меня возни- кает такое подозрение, что a.out в Minix3 также реализован не полностью и представляет собой дополненный заголовком 32-х битный raw-файл.

**boothead.s (4.0K)** – Поддержка BIOS для boot.c . Файл содержит начальный и низко- уровневый код для вторичного загрузчика. Содержит функции для диска, tty (консоли) - ввода с клавиатуры, копирования памяти в произвольную точку.

**rawfs.h (4.0K)** – В этих файлах осуществляется поддержка файловой системы Minix v1 v2

* One function needs to be provided by the outside world:

\*

* void readblock(off\_t blockno, char \*buf, int block\_size);
* Read a block into the buffer. Outside world handles
* errors.

**rawfs.c (4.0K)** – В этих файлах осуществляется поддержка файловой системы Minix v1 v2

**addaout.c (4.0K)** – Маленькая утилита для добавления заголовка minix a.out к произ- вольному файлу. Это позволяет использовать произвольные данные в загрузочном образе так, что эти данные становятся образом участка оперативной памяти.

**boot.c (16K)** – Загружает и запускает Minix.

**boot.h (4.0K)** – Информация между различными частями процесса загрузки.

**bootimage.c (4.0K)** – Загружает образ и запускает его.

**doshead.s (8.0K)** – Файл содержит стартовую и низкоуровневою поддержку вторичной программы загрузчика. Данный вариант загрузчика запускается как com-файл из-под DOS-а.

**image.h (4.0K)** – Информация между инсталляцией загрузчика и загрузчиком.

**installboot.c (4.0K)** – Делает устройство загрузочным.

**jumpboot.s (4.0K)** – Этот код может быть помещён в любой свободный загрузочный

сектор, подобный первому сектору расширенной партиции, партицию файловой системы, отличной от базовой (root), или в основной загрузочный сектор. Этот код загружает новый загрузчик, диск партиция и слайс (субпартиция) помещается в данный код утилитой installboot. Если нажата клавиша ALT, то диск, партиция и субпартиция вводятся вручную. Ручной интерфейс используется (по умолчанию) также в том случае, если installboot (по какой-то причине) не занёс свои данные для загрузки.

**masterboot.s (4.0K)** – Код первичного загрузочного сектора.

**mkfhead.s (4.0K)** – Содержит начальный код и код низкоуровневой поддержки MKFILE.COM.

**mkfile.c (4.0K)** – Код MKFILE.COM. Работает в DOS, создаёт файл, который может быть использован ОС Minix как диск.

**updateboot.sh (4.0K)** – Скрипт устанавливает загрузочный сектор.

# Серверы

## Виртуальная файловая система /servers/vfs

Виртуальная файловая система (virtual file system – VFS) – уровень абстракции поверх кон- кретной реализации файловой системы. Целью VFS является обеспечение единообразного доступа клиентских приложений к различным типам файловых систем. VFS может быть использована, например, для прозрачного доступа к локальным и сетевым устройствам хранения данных без использования специального клиентского приложения (независимо от типа файловой системы). VFS определяет интерфейс между ядром и конкретной файло- вой системой, таким образом, можно легко добавлять поддержку новых типов файловых систем, внося изменения только в ядро операционной системы.

**const.h** – Содержит различные константы:

* + - размеры таблиц (filp , file locking , mount , vnode);
    - uid\_t суперпользователя MM и INIT, uid\_t для которого разрешена FSSIGNON, gid\_t MM и INIT;
    - константы для определения блокировок;
    - для аргументов: LOOK\_UP, ENTER, DELETE, IS\_EMPTY, DUP\_MASK, SYMLOOP, ROOT\_INODE;
    - константы – аргументы для dev\_io : VFS\_DEV\_(READ, WRITE, SCATTER, GATHER, IOCTL, SELECT);

Макросы: fp\_is\_blocked(fp)

**device.c** – Когда необходимый блок не находится в кэше, он должен быть получен из диска. Специальные символьные файлы также нуждаются в вводе/выводе. В данном файле находятся необходимые для этого процедуры.

Точки входа данного файла:

* + - для операций с устройствами (dev\_open, dev\_close, dev\_io, dev\_statu);
    - общие операции (gen\_opcl, gen\_io);
    - операции для несуществующих устройств: (no\_dev, no\_dev\_io);
    - для tty-устройств (tty\_opcl, ctty\_opcl, ctty\_io);
    - для системных вызовов (do\_ioctl, do\_setsid).

**dmap.h** – Таблица устройств: устройство – драйвер.

Индексами таблицы являются главный номер устройства. Таблица обеспечивает связь между главным номером устройства и процедурами, которые обеспечивают его функцио- нирование. Таблица может быть дополнена динамически.

Содержит константы-флаги: DMAP\_MUTABLE, DMAP\_BUSY, DMAP\_BABY.

**dmap.c** – Этот файл содержит таблицу: устройство - таблица драйверов.

В нём также содержаться некоторые процедуры для динамического добавления, удаления драйверов, а также изменения связей. Интерес представляет начальная инициализация init\_dmap[] , содержащая устройства ((/dev/)mem, fd0, c0, tty00, tty, lp, ip, c1, c2, c3, audio, klog, random).

**exec.c** – Этот файл обеспечивает системный вызов EXEC (замены контекста процесса). Порядок работы:

* + - проверяет разрешение файла на исполнение,
    - читает заголовок и определяет размеры,
    - получает начальные аргументы и переменные окружения из пространства пользова- теля,
    - выделяет память для нового процесса,
    - копирует начальный стек из сервера управления процессами (PM) в данный процесс,
    - читает сегменты кода и данных и копирует их в данный процесс,
    - устанавливает биты setuid, setgid,
    - изменяет таблицу ’mproc’,
    - сообщает микроядру о EXEC,
    - сохраняет смещение на начальные аргументы.

**file.h** – Таблица-прослойка между дескрипторами файлов и айнодами ФС.

**filedes.c** – Содержит процедуры для манипуляций с файловыми дескрипторами.

**fproc.h** – Это информация для каждого процесса. Слот резервируется для каждого потенциального процесса. Именно поэтому константа NR\_PROCS должна соответствовать конфигурации микроядра.

**fs.h** – Базовый заголовочный файл для файловой системы. Он включает несколько иных файлов и определяет принципиальные константы – константы режима компиляции (для библиотечных включаемых файлов).

**fscall.c** – Этот файл обеспечивает вложенные контр-запросы серверу виртуальной файловой системы (VFS), посылаемые файловыми серверами в ответ на запрос сервера VFS.

**glo.h** – определения глобальных переменных виртуальной файловой системы (реально выделяются в файле table.o данной подсистемы).

Также содержит переменные в которых сохраняются параметры вызова, возвращаемого результата (код ошибки), а также объявлены некоторые данные, которые инициализируются в другом месте

(\_PROTOTYPE (int (\*call\_vec[]), (void) ); /\* sys call table \*/,

char dot1[2]; /\* dot1 (&dot1[0]) and dot2 (&dot2[0]) have a special \*/ char dot2[3]; /\* meaning to search\_dir: no access permission check. \*/)

**link.c** – Этот файл обеспечивает системные вызовы LINK и UNLINK. Здесь также обес- печивается освобождение памяти, когда когда выполняется последний UNLINK и блоки должны быть возвращены в пул свободных блоков.

**lock.h** – Таблица блокировок файлов. Она указывает на таблицу айнод, однако в данном случае для получения информации о блокировках. Конкретно определяет только один массив: file\_lock[NR\_LOCKS];

**lock.c** – Обеспечивает предусмотренную стандартом POSIX систему блокировки файлов.

**main.c** – Стандартный файл для всех серверов и драйверов, в котором находится бесконеч- ный цикл, получающий сообщения о запросах на выполнения операций, обрабатывающий эти запросы соответствующим образом, а также возвращающим необходимые ответы.

**misc.c** – Этот файл содержит набор различных процедур. Некоторые из них выполняют простейшие системные вызовы. Другие выполняют небольшую часть работы. Связанной с системными вызовами, основную часть которых выполняет сервер управления памятью (MM и VM).

Точки входа:

* + - do\_dup – выполняет системный вызов DUP
    - do\_fcntl – выполняет системный вызов FCNTL
    - do\_sync – выполняет системный вызов SYNC
    - do\_fsync – выполняет системный вызов FSYNC
    - do\_reboot – обновляет диск с буферов в рамках подготовки к завершению работы системы
    - do\_fork – выполняет записи в таблицах VFS после выполнения системного вызова FORK
    - do\_exec – обеспечивает файлы с установленным флагом FD\_CLOEXEC после вы- полнения системного вызова EXEC
    - do\_exit – выполняет записи в таблицах, связанные с завершением процесса
    - do\_set – устанавливает uid или gid для некоторого процесса
    - do\_revive – пересматривает процессы (с целью некоторых действий), которые ожида- ют события в файловой системе (вроде TTY)
    - do\_svrctl – контроль файловой системы
    - do\_getsysinfo – выдаёт копию структур данных файловой системы
    - pm\_dumpcore – выполняет дамп памяти

**mmap.c** – Обеспечение функционала mmap в VFS. Точка входа do\_vm\_mmap – сервер виртуальной памяти вызывает VM\_VFS\_MMAP.

**mount.c** – Обеспечивает системные вызовы MOUNT и UMOUNT.

**open.c** – Этот файл содержит процедуры для создания, открытия, закрытия и изменения позиции "курсора текущей позиции"файла.

**param.h** – Устанавливает внутренние синонимы с полями в структурах входящих сообще- ний, а также синонимы в полях исходящих сообщений.

**path.c** – Основная процедура Lookup(), контролирующая нахождение имени пути (фай- ла, папки или другого файлового объекта). Она поддерживает точки монтирования и символические ссылки. Настоящий запрос на поиск посылается через функцию-обёртку req\_lookup.

**pipe.c** – Функция обеспечивает приостановку и возобновление исполнения процесса (свя- занные с операциями с файловым объектом pipe).

**proto.h** – Содержит все прототипы функций подсистемы (с комментариями, указывающи- ми на местонахождение этих функций – файл подсистемы, в котором реализован данный набор функций).

**read.c** – В этом файле находится механизм чтения и записи файла. Запросы на чтение и запись разделены между собой по чанкам которые не пересекают границы блока (диска). Каждый чанк обрабатывается в один ход. Чтение специальных файлов также определяется и поддерживается.

**request.h** – Низкоуровневые сообщения-запросы строятся и посылаются посредством функций-обёрток. Этот файл содержит описания структур запросов и ответов, используе- мых для доступа к этим функциям-обёркам.

**request.c** – Этот файл содержит функции-обёртки для подачи запросов и получения ответов от процессов файловой системы (FS). Каждая функция строит сообщение запроса в соответствии с запрашиваемыми параметрами, вызывает большинство низкоуровневых fs\_sendrec и копирует обратно конечный ответ – результат.

Низкоуровневый fs\_sendrec также обеспечивает механизм восстановления при отказе диска и повторяет запрос (не удавшийся в предыдущий раз).

**select.h** – Содержит только макроопределения SEL\_OK, EL\_ERROR, SEL\_DEFERRED.

**select.c** – Содержит точки входа, связанные с системным вызовом SELECT:

* + - do\_select – выполняет системный вызов SELECT;
    - select\_callback – сообщает системе SELECT о возможной операции с файловым дескриптором;
    - select\_notified – низкоуровневый вход для устройств сообщающих в контексте SELECT;
    - select\_unsuspend\_by\_endpt – отменяет блокировки при завершении процесса-драйвера.

**stadir.c** – Этот файл содержит код для выполнения 4-х системных вызовов, связанных с состоянием (do\_chdir do\_chroot do\_stat do\_fstat) и каталогами (файловой системы).

**table.c** – Этот файл содержит таблицы, позволяющие отображать номера системных вызовов на процедуры, их обеспечивающие.

**time.c** – Этот файл заботится о системных вызовах, которые связаны со временем (созда- ния, доступа, изменения файловых объектов).

**timers.c** – Библиотека таймеров файловой системы (FS).

**utility.c** – тот файл содержит некоторые процедуры-утилиты общего назначения. Точки входа:

* + - clock\_time – запрашивает процесс CLOCK о реальном времени, copy: копирует блок данных,
    - fetch\_name – получает имя пути из пространства пользователя (связано с сокраще- ниями для домашнего каталога,
    - no\_sys – отвергает системный вызов, который файловая система не обеспечивает;
    - panic – выполняется тогда, когда происходит событие, исключающее возможность дальнейшего исполнения Minix;

**vnode.h** – Описывает структуру ноды

EXTERN struct vnode

/\* indicates absence of vnode slot \*/

define NIL\_VNODE (struct vnode \*) 0

/\* Field values.

#define NO\_PIPE

#define I\_PIPE

\*/

0

1

/\* i\_pipe is NO\_PIPE if inode is not a pipe \*/

/\* i\_pipe is I\_PIPE if inode is a pipe \*/

## Межпроцессное взаимодействие /servers/ipc

Межпроцессное взаимодействие (англ. inter-process communication, IPC) — обмен данными между потоками одного или разных процессов. Реализуется посредством механизмов, предоставляемых ядром ОС или процессом, использующим механизмы ОС и реализующим новые возможности IPC. Может осуществляться как на одном компьютере, так и между несколькими компьютерами сети.

**inc.h** – Содержит стандартные константы и включения для подобного типа подсистем (привилегированных процессов), а также определения внутренних констант и прототипов функций.

Данный сервер является относительно новым (включён только в Minix3.1.5) и мал по объёму. Поэтому данный файл выполняет функции сразу нескольких файлов более объёмных подсистем.

**sem.c** – Содержит функции (связанные с семафорами) в.т.ч.:

int do\_semget(message \*m); int do\_semctl(message \*m); int do\_semop(message \*m); int is\_sem\_nil(void);

**shm.c** – Максимальное число областей разделяемой памяти:

#define MAX\_SHM\_NR 1024

Содержит функции (связанные с разделяемой памятью) в.т.ч.:

int do\_shmget(message \*m); int do\_shmat(message \*m);

void update\_refcount\_and\_destroy(void); int do\_shmdt(message \*m);

int do\_shmctl(message \*m); void list\_shm\_ds(void); int is\_shm\_nil(void);

**utility.c** Содержит только функцию

int check\_perm(struct ipc\_perm \*req, endpoint\_t who, int mode);

## Управление процессами /servers/pm

Важнейшей частью операционной системы, непосредственно влияющей на функциони- рование вычислительной машины, является подсистема управления процессами (process management). Процесс (или по-другому, задача) - абстракция, описывающая выполняющу- юся программу. Для операционной системы процесс представляет собой единицу работы, заявку на потребление системных ресурсов. Подсистема управления процессами планирует выполнение процессов, то есть распределяет процессорное время между несколькими одновременно существующими в системе процессами, а также занимается созданием и уничтожением процессов, обеспечивает процессы необходимыми системными ресурсами, поддерживает взаимодействие между процессами.

**alarm.c** – Этот файл обеспечивает системные вызовы, связанные с отложенными процес- сами (системным будильником), периодически посылая работу функциям в файле timer.c и check\_sig() в файле signal.c, для посылки сигнала пробуждения процессу.

Точки входа:

* + - do\_itimer – выполняет системный вызов ITIMER;
    - do\_alarm – выполняет системный вызов ALARM;
    - set\_alarm – сообщает интерфейсу таймера, чтобы запустить или остановить таймер процесса;
    - check\_vtimer – проверяет необходимость перезапуска виртуальных таймеров.

**break.c** – Содержит функцию int do\_brk(). Это точка входа для системного вызова brk(addr).

**const.h** – Константы, используемые сервером управления процессами (PM): NR\_PIDS, PM\_PID, INIT\_PID, NO\_TRACER, DUMPED, MAX\_SECS, NR\_ITIMERS.

**exec.c** – Этот файл обеспечивает системный вызов EXEC. Порядок выполняемой работы:

* + - проверяет, что файл может быть исполнен (по разрешениям UNIX);
    - читает заголовок и получает размеры;
    - получает начальные аргументы и переменные окружения из пространства пользова- теля;
    - выделяет память для нового процесса;
    - копирует начальный стек из PM в процесс;
    - читает сегменты кода и данных и копирует в процесс;
    - контролирует биты setuid, setgid;
    - исправляет (изменяет) таблицу «mproc»;
    - сообщает микроядру о EXEC;
    - сохраняет смещение для начального argc (для PS).

Точки входа этого файла:

* + - do\_exec – выполняет системный вызов EXEC;
    - exec\_newmem – выделяет новую карту памяти для процесса, который пытается выполнить EXEC;
    - do\_execrestart – заканчивает специальный вызов exec для сервера реинкарнации (RS);
    - exec\_restart – заканчивает обычный вызов exec;
    - find\_share – находит процесс, сегмент кода которого может быть совместно использо- вать.

**forkexit.c** – Этот файл обеспечивает создание процесса (через системный вызов FORK) и уничтожение (стирание) процесса (через EXIT/WAIT).

Когда процесс раздваивается (forks), для него (точнее - процесса-потомка) выделяется новый слот в таблице mproc, а также копия образа родительского процесса. Затем инфор- мируется ядро и файловая система.

Процесс удаляется из таблицы «mproc» когда происходят два события:

1. процесс завершил исполнение или был убит посредством сигнала
2. процесс-родитель выполнил WAIT.

Если процесс завершил исполнение прежде чем его родитель вызвал WAIT, то слот продолжает существовать до тех пор, пока процесс родитель не выполнит соответствующий WAIT.

Точки входа:

* + - do\_fork – выполняет системный вызов FORK;
    - do\_fork\_nb – неблокирующая версия FORK для сервера реинкарнации (RS);
    - do\_exit – выполняет системный вызов EXIT (посредством вызова exit\_proc());
    - exit\_proc – собственно и выполняет завершение процесса, а также сообщает файловой системе об этом;
    - exit\_restart – продолжает завершение процесса после того, как получен ответ от файловой системы (FS);
    - do\_waitpid – выполняет системные вызовы WAITPID или WAIT;
    - wait\_test – проверяет, ждёт ли процесс-родитель завершения данного процесса.

**getset.c** – Этот файл обеспечивает 6 системных вызовов, которые получают и устанавли- вают значения uid, gid. Он также обеспечивает getpid(), setsid(), и getpgrp().

Содержит функции:

* + - int do\_get(); – обеспечивает системные вызовы GETUID, GETGID, GETPID, GETPGRP.
    - int do\_set(); – обеспечивает системные вызовы ETUID, SETEUID, SETGID, SETEGID, SETSID (эти вызовы связаны также с VFS).

**glo.h** – Глобальные переменные сервера (управления процессами). Они реально выделяются в table.o.

**main.c** – Этот файл содержит функцию main для сервера управления процессами и некоторые связанные с main процедуры.

Когда MINIX запускается, в самом начале микроядро инициализирует себя свои задания, а затем оно передаёт управление серверам управления процессами (PM) и файловой системы (FS). Оба: PM и FS инициализуют себя настолько, насколько они могут. Сервер управ- ления процессами (PM) запрашивает микроядро для всей свободной памяти и начинает обслуживать запросы.

Точки входа:

* + - main – запускает сервер управления процессами;
    - setreply – устанавливает ответ, который отсылается процессу, выполняющему систем- ный вызов к серверу управления процессами (PM).

**misc.c** – Различные системные вызовы:

* + - do\_reboot – уничтожает все процессы, затем перезагружает систему;
    - do\_procstat – запрашивает статус процесса
    - do\_getsysinfo – запрашивает копию структуры данных PM;
    - do\_getprocnr – ищет номер слота процесса
    - do\_getpuid – по данной конечной точке находит uid/euid процесса;
    - do\_allocmem – выделяет (процессу) чанк памяти;
    - do\_freemem – удаляет (из адресного пространства процесса) чанк памяти;
    - do\_getsetpriority – получает/устанавливает приоритет процесса;
    - do\_svrctl – контроль управляющего процессами (планировщика).

**mproc.h** – Эта таблица содержит по одному слоту для каждого процесса. Она содержит всю информацию по управлению процессами для каждого процесса. Кроме всего прочего, она определяет сегменты кода, данных и стека, а также различные флаги.

Микроядро и файловая система также имеют таблицы, индексированные по процессам, которые содержат соответствующие слоты, относящиеся к одному процессу во всех трёх таблицах.

**pm.h** – Это основной заголовочный файл сервера управления процессами. Он включает некоторые другие файлы и определяет некоторые важнейшие константы: \_POSIX\_SOURCE,

\_MINIX, \_SYSTEM.

**signal.c** – Этот файл поддерживает сигналы, которые являются асинхронными событиями. Сигналы могут быть сгенерированы посредством системного вызова KILL, или от клавиату- ры (SIGINT) или от часов (SIGALRM). Во всех случаях контроль в дальнейшем передаётся

к функции check\_sig() для определения того, какому процессу посылается сигнал. Реально сигнал выполняется (т.е. изменяется состояние соответствующего процесса) посредством функции sig\_proc().

Точки входа:

* + - do\_sigaction – выполняет системный вызов SIGACTION;
    - do\_sigpending – выполняет системный вызов SIGPENDING;
    - do\_sigprocmask – выполняет системный вызов SIGPROCMASK;
    - do\_sigreturn – выполняет системный вызов SIGRETURN;
    - do\_sigsuspend – выполняет системный вызов SIGSUSPEND;
    - do\_kill – выполняет системный вызов KILL;
    - do\_pause – выполняет системный вызов PAUSE;
    - ksig\_pending – микроядро уведомляется о необработанном сигнале;
    - sig\_proc – прерывает или завершает сигнализирующий (или сигнализируемый) про- цесс;
    - check\_sig – проверяет, какие процессы должны сигнализироваться посредством sig\_proc();
    - check\_pending – проверяет, может ли необработанный (ранее) сигнал сейчас быть обработан;
    - restart\_sigs – возобновляет работу с сигналами после вызова сервера (серверов) файловой системы.

**table.c** – Этот файл содержит таблицу, которая используется для отображения номеров системных вызовов на процедуры, их обрабатывающие.

**trace.c** – Этот файл обеспечивает часть сервера управления процессами (PM), ответствен- ную за отладочные функции, использующие системный вызов ptrace. Большинство команд в дальнейшем посылаются в системное задание микроядра (SYSTEM) для завершения.

Доступные отладочные команды:

* + - T\_STOP – останавливают процесс,
    - T\_OK – включает трассировку родителем этого процесса,
    - T\_GETINS – возвращает значение из пространства инструкций,
    - T\_GETDATA – возвращает значение из пространства данных,
    - T\_GETUSER – возвращает значение из таблицы пользовательского процесса,
    - T\_SETINS – устанавливает значение в пространстве инструкций,
    - T\_SETDATA – устанавливает значение в пространстве данных,
    - T\_SETUSER – устанавливает значение в таблице пользовательского процесса,
    - T\_RESUME – восстановить исполнение,
    - T\_EXIT – выход,
    - T\_STEP – устанавливает бит трассирования,
    - T\_SYSCALL – системный вызов трассирования,
    - T\_ATTACH – подключить к существующему процессу,
    - T\_DETACH – отключить от существующего процесса,
    - T\_SETOPT – устанавливает опции трассирования.

**utility.c** – Этот файл содержит некоторые обслуживающие функции сервера управления процессами (PM).

## Виртуальная память /servers/vm

Виртуальная память (англ. virtual memory) – метод управления памятью компьютера, позволяющий выполнять программы, требующие больше оперативной памяти, чем имеется в компьютере, путём автоматического перемещения частей программы между основной памятью и вторичным хранилищем (например, жёстким диском). Для выполняющейся программы данный метод полностью прозрачен и не требует дополнительных усилий со стороны программиста, однако реализация этого метода требует как аппаратной поддержки, так и поддержки со стороны операционной системы.

В системе с виртуальной памятью используемые программами адреса, называемые вирту- альными адресами, транслируются в физические адреса в памяти компьютера. Трансляцию виртуальных адресов в физические выполняет аппаратное обеспечение, называемое блоком управления памятью. Для программы основная память выглядит как доступное и непре- рывное адресное пространство, либо как набор непрерывных сегментов, вне зависимости от наличия у компьютера соответствующего объёма оперативной памяти. Управление виртуальными адресными пространствами, соотнесение физической и виртуальной памяти,

а также перемещение фрагментов памяти между основным и вторичным хранилищами выполняет операционная система.

**addravl.h** – Содержит определения констант и макросов препроцессора.

**addravl.c** – Содержит только включения заголовков.

**alloc.c** – Этот файл связан с выделением и освобождением блоков физической памяти произвольной величины в контексте системных вызовов FORK и EXEC. Ключевая исполь- зуемая структура данных – таблица свободных участков, которая поддерживает список свободных участков памяти. Они поддерживаются в порядке возрастания адреса памяти. Эти адреса содержат ссылки на физическую память, начиная с абсолютного адреса 0 (т.е. они не являются относительными от начала (адресного пространства) сервера управления процессами (PM)). Во время инициализации системы, часть памяти, содержащая вектора прерываний, микроядро и сервер управления процессами (PM) размещены в смысле по- метки их (областей памяти) как недоступными для выделения и удаления их из списка свободных мест.

Точки входа:

* + - alloc\_mem – выделяет чанк памяти данного размера;
    - free\_mem – освобождает прежде выделенный чанк памяти;
    - mem\_init – инициализирует таблицы когда сервер управления процессами запускает- ся.

**break.c** – Модель выделения памяти MINIX резервирует участки памяти фиксированного размера для комбинированных сегментов кода, данных и стека. Эти размеры используются для процесса-потомка созданного посредством FORK те же, что и для процесса-родителя. Если процесс-потомок позже выполняет EXEC,будут использованы новые размеры из заголовка запускаемого бинарного исполняемого файла.

Расположение памяти состоит из сегмента текста, после которого следует сегмент данных, за которым следует куча (неиспользуемая память), после которой следует сегмент стека. Сегмент данных растёт вверх, а сегмент стека – вниз, таким образом каждый из них может заимствовать память из кучи. Если они встречаются процесс должен быть уничтожен. Процедуры данного файла заботятся о росте сегментов данных и стека.

Точки входа:

* + - do\_brk – системные вызовы BRK/SBRK для роста или сокращения сегмента данных;
    - adjust – смотрит разрешено ли запрашиваемое изменение сегментов (в смысле полно- мочий и привилегий).

**exec.c** – Содержит функции:

struct vmproc \*find\_share(vmp\_ign, ino, dev, ctime);

Ищет процесс, который исполняет файл <ino, dev, ctime>. Не обязательно найдёт vmp\_ign, так как это процесс, от лица которого этот вызов выполняется.

int do\_exec\_newmem(message \*msg);

int new\_mem(rmp, sh\_mp, text\_bytes, data\_bytes, bss\_bytes, stk\_bytes, tot\_bytes, stac

Выделяет новую память и освобождает старую память. Изменяет карту памяти и сообщает новую микроядру. Обнуляет bss, «кучу» и стек нового образа (памяти).

phys\_bytes find\_kernel\_top(void);

Определяет где находится микроядро, чтобы таким образом знать начало отображения пользовательских процессов.

int proc\_new(struct vmproc \*vmp,

phys\_bytes vstart, /\* где начать процесс в таблице страниц\*/ phys\_bytes text\_b phys\_bytes data\_bytes, /\* как много данных + bss, в байтах но ...\*/ phys\_bytes stack\_ phys\_bytes gap\_bytes, /\* «куча» , в байтах но ...\*/

phys\_bytes text\_start, /\* код начинается здесь, если предварительно выделен, иначе 0 phys\_bytes data\_start, /\*участок данных начинается здесь, если предварительно выделен phys\_bytes stacktop

);

**exit.c** – Содержит функции:

void free\_proc(struct vmproc \*vmp); void clear\_proc(struct vmproc \*vmp); int do\_exit(message \*msg);

int do\_willexit(message \*msg); void \_exit(int code);

void exit(int code);

**fork.c** – Содержит функцию:

int do\_fork(message \*msg);

**physravl.h** – Определяет константы и макросы.

**physravl.c** – Состоит только из включений заголовочных файлов. **proto.h** – Содержит прототипы функций сгруппированные по файлам. **slaballoc.c** – Содержит огромное множество определений макросов **util.h** – Содержит определения макросов

**utility.c** – Этот файл содержит некоторые служебные функции для сервера виртуальной памяти (VM):

int get\_mem\_map(proc\_nr, mem\_map); void get\_mem\_chunks(mem\_chunks);

Инициализирует свободную память. Переводит байтовые сдвиги и размеры в этом списке в клики, нарезанные правильным образом.

void reserve\_proc\_mem(mem\_chunks, map\_ptr);

Выбирает память сервера из списка свободной памяти. Монитор загрузки обещает поме- стить процессы в начале чанков памяти. Все задания (микроядра) используют одинаковый базовый адрес, таким образом только первое задание изменяет списки памяти. Серверы и init имеют свои собственные пространства памяти и их память в дальнейшем будет удалена из этого списка (свободной памяти).

int vm\_isokendpt(endpoint\_t endpoint, int \*proc); int get\_stack\_ptr(proc\_nr\_e, sp);

int brk(brk\_addr);

int do\_ctl(message \*m);

**vm.h** – Определяет некоторые общие константы и макросы.

# Заключение

Код Minix развивается и поддерживается в академической среде, с расчётом на то, что его будут изучать. В итоге, код отлично структурирован и документирован. Он читается как книга и не требует привлечения каких-либо дополнительных инструментов.

Основным источником информации при изучении кода стали комментарии, которые сопро- вождают каждый файл. Другим важным источником о принятых решениях в процессе проектирования являются комментарии к комитам в системе контроля версий.

Оба эти фактора повышают поддерживаемость исходного кода и обеспечивают быстрое включение в архитектуру системы.