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

Реверс-инжиниринг ПО (обратный инжиниринг, обратная разработка) это процесс восстановления 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 восстановление 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. Последний 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 получить 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 C 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 пользовательские 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 (директории /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.

1 Микроядро

1.1 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 обработки событий. Важные события, обрабатываемые посредством CLOCK, включают также решения по планированию и перепланированию процессов. CLOCK предлагает непосредственный 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 соображениям производительности.

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

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 попадаем 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 вынесенным 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 определены 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 прерываний 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), определённых 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 применяется 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 (расширяет 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 находятся 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 микроядра. 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 обрабатывается 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, используемого для выявления ошибок в ядре

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

1.2 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 областей 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 которого 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 низкоуровневый 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 область 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 сервером управления процессами (PM) после обработки сигнала процессу SYS_GETKSIG. Обычно это сигнал прерывания процесса.)

Параметры:

- SIG_ENDPT – Процесс, для которого выполнено задание сервера управления процессами (PM)

do_exec.c (4,0K) – Реализуется вызов микроядра SYS_EXEC (Замена контекста процесса.)

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 системную информацию, которая копируется в адресное пространство запрашивающего процесса. Этот вызов просто копирует соответствующие структуры данных запрашивающему процессу.)

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 частности, 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) – центральный 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 использовать 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 образец 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 привилегии 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 местоположении их таблицы профилирования и контрольной структуры. Вызов микроядра используется системой профилирования когда установлена опция профилирования вызовов.)

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 процесса 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 переключатель 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 сигнализирующий 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 сигналов 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 информацию 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 отладочную 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 отображения 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.

1.3 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, сегментного дескриптора защищённого режима процессора, страничных исключений

1.4 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 обработчик 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 данные 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 прерываний 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) хранится 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 предположении, 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.

2 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 возникает 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 низкоуровневый 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 произвольному 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.

3 Серверы

3.1 Виртуальная файловая система /servers/vfs

Виртуальная файловая система (virtual file system – VFS) – уровень абстракции поверх конкретной 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 файловой 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 функционирование. 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 пользователя,
- 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 обеспечивается 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 бесконечный 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 выполнения 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 ожидают 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 сообщений, а также синонимы в полях исходящих сообщений.

path.c – Основная процедура Lookup(), контролирующая нахождение имени пути (файла, 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 (связанные с операциями с файловым объектом pipe).

proto.h – Содержит все прототипы функций подсистемы (с комментариями, указывающими 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, используемых 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 (создания, 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 сокращениями 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        0        /* i_pipe is NO_PIPE if inode is not a pipe */
#define I_PIPE        1   /* i_pipe is I_PIPE if inode is a pipe */
```

3.2 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);
```

3.3 Управление процессами /servers/pm

Важнейшей частью операционной системы, непосредственно влияющей на функционирование вычислительной машины, является подсистема управления процессами (process management). Процесс (или по-другому, задача) - абстракция, описывающая выполняющуюся 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 процессами (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 пользователя;
- 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 использовать.

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 информируется 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 устанавливают 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 управления 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 системный 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 клавиатуры (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) процесс;
- 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), ответственную 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).

3.4 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 виртуальными 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 непрерывное 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 используемая 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 пометки 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 запускается.

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 полномочий 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 поместить 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 сопровождают 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.