

**Д.В. Ефремов, Н.Ю. Комаров, А.В. Хорошилов**

# **КОНСТРУИРОВАНИЕ ЯДРА ОПЕРАЦИОННОЙ СИСТЕМЫ**

Москва  
2015

УДК 004.415.5

ББК 32.973-18

*Печатается по решению Редакционно-издательского Совета  
факультета вычислительной математики и кибернетики  
МГУ имени М.В. Ломоносова*

Рецензенты:

А.Н. Томилин, профессор, д.ф.-м.н.

И.В. Машечкин, профессор, д.ф.-м.н.

**Д.В. Ефремов, Н.Ю. Комаров, А.В. Хорошилов.**

**Конструирование ядра операционной системы.** — М.

Издательский отдел факультета ВМК МГУ имени М.В. Ломоносова  
(лицензия ИД № 05899 от 24.09.2001); МАКС Пресс, 2015. — 161 с.

ISBN 978-5-89407-549-5

ISBN

Данное пособие предназначено для изучения на практике основных принципов устройства ядра операционной системы и механизмов аппаратной поддержки его работы. Пособие рекомендуется студентам и аспирантам, желающим получить навыки программирования компонентов ядра операционной системы, работающих в привилегированном режиме работы процессора.

The textbook augments a course aimed for learning basic concepts of operating system kernels and corresponding hardware features. The textbook is recommended for students who are interested in getting practical experience in programming operating system kernels and debugging code in privileged processor mode.

Ключевые слова: операционная система, планировщик, виртуальная память, прерывания, системный вызов, файловая система.

Keywords: operating system, scheduler, virtual memory, interrupts, system call, file system.

ISBN 978-5-89407-531-0 © Факультет вычислительной математики  
и кибернетики МГУ имени М.В.Ломоносова, 2015

ISBN © Д.В. Ефремов, Н.Ю. Комаров, А.В. Хорошилов, 2015

# Оглавление

1 Введение.....	7
2 Конструирование ядра ОС.....	8
2.1 Однопоточная ОС без виртуальной памяти.....	8
2.1.1 Адресное пространство x86.....	9
2.1.2 BIOS.....	11
2.1.3 Загрузчик.....	14
2.1.4 Загрузка ядра.....	16
2.1.5 Монитор.....	19
2.1.6 Стек.....	20
2.1.7 Отладочная информация.....	22
2.2 Многопоточная ОС без виртуальной памяти.....	26
2.2.1 Управление процессами.....	27
2.2.2 Создание и запуск процесса.....	31
2.2.3 Планировщик.....	34
2.2.4 Связывание.....	35
2.2.5 Управление часами реального времени.....	37
2.2.6 Синхронизация между процессами.....	41
2.3 ОС с виртуальной памятью и процессами.....	44
2.3.1 Обход зависимости от местоположения.....	45
2.3.2 Управление страницами.....	48
2.3.3 Виртуальная память.....	52
2.3.4 Управление каталогами и таблицами страниц.....	58
2.3.5 Адресное пространство ядра.....	59
2.3.6 Процессы в пользовательском режиме.....	62
2.3.7 Выделение массива процессов.....	62
2.3.8 Создание и запуск процесса.....	63
2.4 Прерывания.....	66
2.4.1 Обработка прерываний и исключений.....	67
2.4.2 Защищенная передача управления.....	67
2.4.3 Типы исключений и прерываний.....	70
2.4.4 Пример.....	70

2.4.5 Вложенные исключения и прерывания.....	72
2.4.6 Создание таблицы дескрипторов прерываний.....	73
2.4.7 Вопросы.....	75
2.4.8 Обработка ошибок страниц.....	76
2.5 Системные вызовы.....	77
2.5.1 Точка останова.....	78
2.5.2 Вопросы.....	79
2.5.3 Запуск пользовательского режима.....	79
2.5.4 Ошибки страниц и защита памяти.....	80
2.5.5 Системные вызовы для создания процессов.....	83
2.5.6 Fork() с копированием при записи.....	86
2.5.7 Обработка ошибок страниц в пользовательском режиме.....	87
2.5.8 Установка обработчика ошибки страницы.....	88
2.5.9 Обычный стек и стек исключений в пользовательских процессах.....	89
2.5.10 Вызов пользовательского обработчика ошибок страниц.....	90
2.5.11 Точка входа ошибки страницы пользовательского режима.....	91
2.5.12 Реализация fork() с копированием при записи....	93
2.6 Межпроцессное взаимодействие.....	96
2.6.1 Базовый механизм межпроцессного взаимодействия в JOS.....	96
2.6.2 Отправка и прием сообщений.....	96
2.6.3 Передача страниц.....	97
2.7 Файловая система.....	99
2.7.1 Доступ к диску.....	100
2.7.2 Кэш блоков.....	102
2.7.3 Операции с файлами.....	104
2.7.4 Интерфейс файловой системы.....	105
2.8 Командная оболочка.....	108
2.8.1 Запуск процессов.....	109

2.8.2 Разделение файловых дескрипторов.....	109
2.8.3 Клавиатура.....	112
2.8.4 Командная оболочка.....	112
2.9 Системные вызовы без переключения контекста.....	114
2.9.1 Получение текущего времени.....	114
2.9.2 Системные вызовы без переключения контекста	116
3 Справочные материалы.....	117
3.1 Настройка окружения и запуск JOS.....	117
3.2 Работа с Git.....	120
3.2.1 Клонирование репозитория.....	120
3.2.2 Просмотр состояния репозитория.....	121
3.2.3 Отслеживание новых файлов.....	122
3.2.4 Индексация измененных файлов.....	122
3.2.5 Просмотр изменений.....	124
3.2.6 Фиксация изменений.....	125
3.2.7 Пропуск индексации.....	126
3.2.8 Удаление файлов.....	127
3.2.9 Просмотр истории коммитов.....	128
3.2.10 Ограничение вывода команды log.....	130
3.2.11 Изменение последнего коммита.....	130
3.2.12 Отмена индексации файла.....	131
3.2.13 Отмена внесенных изменений.....	132
3.2.14 Ветвление.....	132
3.2.15 Конфликты при слиянии.....	134
3.2.16 Stash.....	136
3.2.17 Работа с удаленными репозиториями.....	138
3.2.18 Добавление и удаление удаленных репозиториев .....	139
3.2.19 Получение данных из удаленного репозитория	140
3.2.20 Отправка изменений в удаленный репозиторий	141
3.2.21 Инспекция удаленного репозитория.....	141
3.2.22 Удаленные ветки.....	142
3.3 Организация исходного кода JOS.....	144

3.4 Способы отладки.....	151
3.4.1 GDB.....	151
3.4.2 Монитор QEMU.....	152
3.5 Некоторые возможные проблемы и подходы к их решению.....	156
3.5.1 Планировщик.....	156
3.5.2 Процессы и память.....	157
3.5.3 Другие проблемы.....	158
4 Список литературы.....	160

# 1 Введение

Преподавание основ архитектуры ЭВМ и системного программного обеспечения, включающего в себя операционные системы, ведется на факультете ВМК МГУ им. М.В. Ломоносова с момента его основания в 1970 году [1]. Как отдельный предмет курс «Операционные системы», разработанный по руководством И.В. Машечкина [2], читается с конца 90-х годов. Отдельные вопросы разработки операционных систем освещаются в курсах по архитектуре вычислительных систем [3] и по распределенным вычислительным системам [4].

Настоящее пособие предназначено для студентов курса «Конструирование ядра операционной системы», который предлагает дальнейшие погружение в особенности работы ядра операционной системы и позволяет познакомиться на практике с его основными концепциями, такими как планировщик, виртуальная память, прерывания, механизмы синхронизации, системные вызовы, файловые системы и т. д.

В рамках курса каждым студентом ведется постепенная разработка учебной операционной системы на языке Си со вставками на ассемблере x86. Система запускается на эмуляторе ЭВМ QEMU. По мере изучения тем курса слушателям выдаются варианты каркаса операционной системы, наполнение каркаса осуществляется слушателями в ходе выполнения заданий, представленных в данном пособии.

Настоящее методическое пособие построено на основе материалов курса 6.828 «Конструирование операционных систем» Массачусетского технологического института [5] и развивает их по ряду направлений. Например, были добавлены новые лабораторные работы, рассматривающие построение операционной системы с полноценным планировщиком, но при этом работающей в едином адресном пространстве с пользовательскими процессами.

## **2 Конструирование ядра ОС**

### **2.1 Однопоточная ОС без виртуальной памяти**

Перед началом работы необходимо настроить окружение, скопировать исходный код JOS и запустить ее, как описано в разд. 3.1.

В данном разделе рассматривается первая задача, которая возникает перед разработчиком операционной системы: запуск на компьютере произвольной программы. Эта задача не так проста, как может показаться на первый взгляд. При разработке обычных программ способ их запуска, поддержка ввода-вывода и другие необходимые для работы условия считаются по умолчанию выполненными: программа имеет в качестве окружения операционную систему и стандартную библиотеку, что позволяет сконцентрироваться на реализации нужного алгоритма. При разработке операционной системы ситуация кардинально меняется: необходимо напрямую работать с аппаратным обеспечением компьютера, вручную реализовывать программную поддержку его возможностей и особенностей. В результате разработка даже, казалось бы, простейшей программы, выводящей на экран произвольную строку, требует глубокого понимания процесса загрузки компьютера, структуры физической памяти, аппаратной поддержки ввода-вывода и т. д.

Первая часть данного раздела соответствует лабораторной работе №1. Цель этой работы – произвести настройку окружения, ознакомиться с процессом загрузки операционной системы и ознакомиться с базовыми способами отладки с помощью GDB.

### 2.1.1 Адресное пространство x86

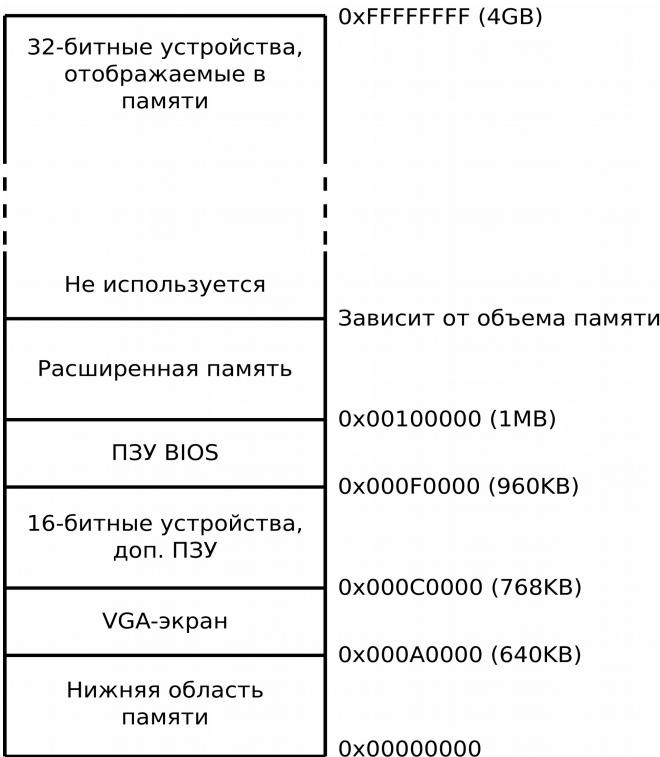


Рисунок 2.1.1: Организация адресного пространства компьютера

Рассмотрим более подробно процесс загрузки компьютера. Схема организации физического адресного пространства компьютера приведена на рис. 2.1.1. Первые компьютеры IBM PC, основанные на 16-битных процессорах Intel 8088, могли адресовать только 1 МБ физической памяти; их физическое адресное пространство начиналось с 0x00000000, но заканчивалось на 0x000FFFFF вместо 0xFFFFFFFF. Эти компьютеры могли использовать только область размером в 640 КБ, отмеченную на рисунке 1 как «нижняя область памяти» (Low Memory); причем некоторые компьютеры были сконфигури-

рованы для использования только 16, 32 или 64 КБ памяти. Область размером 384 КБ от 0x000A0000 до 0x000FFFFF была зарезервирована для специального применения, например, в качестве буфера дисплея или для отображения микропрограммы, находящиеся в энергонезависимой памяти. Наиболее важной частью этой зарезервированной области является BIOS (базовая система ввода-вывода), которая занимает область размером 64 КБ от 0x000F0000 до 0x000FFFFF. В ранних компьютерах BIOS находился в настоящих ПЗУ, доступных только для чтения, в современных компьютерах BIOS хранится в перезаписываемой флэш-памяти. BIOS отвечает за выполнение основных операций инициализации системы, таких как активация видеoadаптера и проверка объема установленной памяти. После выполнения инициализации BIOS загружает операционную систему с дискеты, жесткого диска, компакт диска или из сети, а затем передает ей контроль над компьютером.

С выпуском процессоров 80286 и 80386 объем поддерживаемой памяти был увеличен до 16 МБ и 4 ГБ соответственно, но организация нижнего 1 МБ физического адресного пространства была сохранена для обеспечения обратной совместимости с существующим программным обеспечением. Поэтому современные компьютеры имеют «дыры» в физической памяти от 0x000A0000 до 0x00100000, а память делится на «нижнюю» или «обычную» («low memory» — первые 640 КБ) и «расширенную» («extended memory» — все остальное). Кроме того, некоторые области в самом верху адресного пространства, выше всей физической памяти, обычно резервируются BIOS для использования 32-разрядными PCI-устройствами.

Современные x86-процессоры могут поддерживать более 4 ГБ физической памяти и использовать области выше 0xFFFFFFFF. В этом случае BIOS должен позаботиться о том,

чтобы оставить еще одну неиспользуемую область памяти для 32-разрядных PCI-устройств в верхней части 32-битной области памяти. Из-за конструктивных ограничений JOS использует только первые 256 МБ физической памяти компьютера, и эта проблема не является существенной. Однако нельзя забывать, что использование аппаратной части компьютера, которая развивалась на протяжении многих лет, и, в частности, физического адресного пространства с его сложной исторически сложившейся организацией, является одной из важных практических задач в разработке операционных систем.

### 2.1.2 BIOS

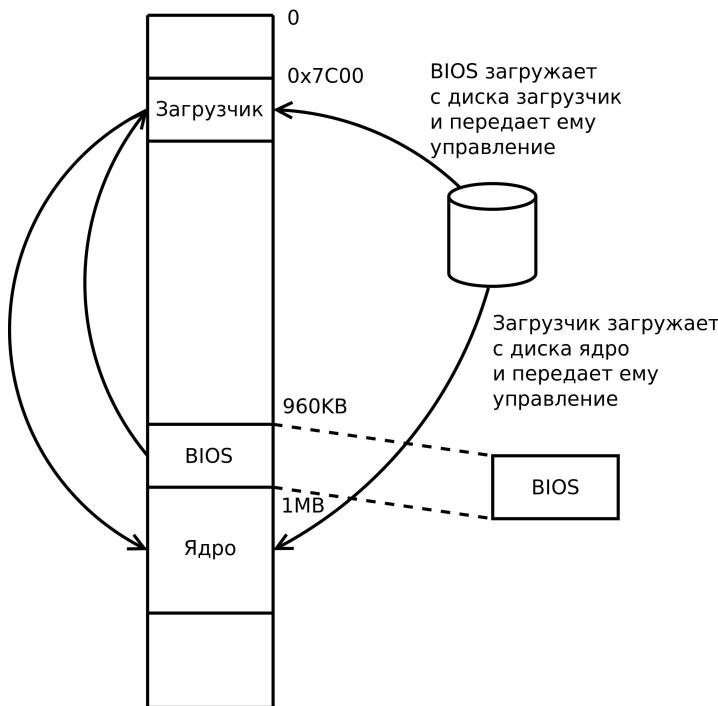


Рисунок 2.1.2: Загрузка компьютера

В этой части работы необходимо использовать отладочные функции QEMU и отладчик GDB для изучения процесса загрузки операционной системы, схема которого приведена на рис. 2.1.2. Руководство по работе с GDB находится в разд. 3.4.1.

Откройте два окна терминала, в обоих перейдите в каталог с исходным кодом лабораторных работ. В одном выполните команду make qemu-gdb. Эта команда запустит QEMU, после чего остановит ее сразу перед началом выполнения процессором первой инструкции и будет ожидать подключения отладчика. Во втором выполните команду gdb. Запустится отладчик GDB и появится текст следующего вида:

```
$ gdb
GNU gdb (GDB) 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show
copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
+ target remote localhost:1234
The target architecture is assumed to be i8086
[f000:fff0] 0xfffff0:    ljmp  $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb)
```

Строка

```
[f000:fff0] 0xfffff0:    ljmp  $0xf000,$0xe05b
```

— первая инструкция, которая будет исполнена, в дисассемблированном виде. Из этого вывода можно сделать следующие заключения:

- Процессор начинает выполнять инструкции с физического адреса 0x000FFFF0, который находится в самом начале зарезервированной для BIOS области размером 64 КБ.
- Процессор стартует со значениями регистров CS=0xF000 и IP=0xFFFF0.
- Первая инструкция, которая будет выполнена — jmp, который перейдет по адресу CS=0xF000 и IP=0xE05B.

► *Выполните несколько следующих инструкций и попытайтесь понять, что они делают.*

После старта BIOS создает таблицу прерываний и инициализирует различные устройства, такие как экран. После инициализации шины PCI и всех необходимых устройств BIOS ищет устройство, с которого можно загрузить операционную систему, например, жесткий или компакт-диск. Когда BIOS находит такое устройство, он считывает с него специальный фрагмент, называемый загрузчиком, и передает ему управление.

В последние годы BIOS постепенно заменяется новым стандартом UEFI, обладающим некоторыми преимуществами, такими как:

- Модульная архитектура.
- Встроенный менеджер загрузки операционных систем, отсутствие необходимости установки сторонних загрузчиков.
- Поддержка GPT (GUID Partition Table), нового способа разметки дисков, поддерживающего, в отличие от MBR (Master Boot Record), диски размером более 2ТБ и неограниченное количество разделов.

### **2.1.3 Загрузчик**

Дискеты и жесткие диски делятся на области длиной в 512 байт, называемые секторами. Сектор является минимальной единицей передачи данных с диска: каждая операция чтения или записи должна быть размером в один или несколько секторов и выровнена по границе сектора. Если диск является загрузочным, в его первом секторе находится код загрузчика. Если BIOS обнаруживает загрузочную дискету или жесткий диск, он загружает 512 байт загрузочного сектора в память по физическим адресам с 0x7C00 по 0x7DFF, а затем использует инструкцию JMP, чтобы записать в CS:IP 0000:7C00, передавая управление загрузчику. Как и адрес загрузки BIOS, эти адреса изначально выбраны произвольно, но они являются фиксированными и стандартизированы для компьютеров с архитектурой x86.

JOS использует обычный механизм загрузки с жесткого диска. Это означает, что загрузчик должен помещаться в 512 байт. Исходный код загрузчика состоит из двух файлов: одного на языке ассемблера (boot/boot.S) и одного на языке C (boot/main.c). Загрузчик выполняет две основные функции:

1. Сначала загрузчик переключает процессор из реального режима в 32-битный защищенный режим, потому что только в этом режиме программы могут получить доступ к памяти выше 1 МБ. Защищенный режим кратко описан в разд. 1.2.7 и 1.2.8 книги PC Assembly Language [6] и подробно — в третьем томе руководства Intel по архитектуре процессора для разработчиков [8].
2. Затем загрузчик считывает ядро с жесткого диска, обращаясь напрямую к регистрам IDE-устройства с помощью специальных инструкций ввода-вывода.

После изучения кода загрузчика просмотрите файл obj/boot/boot.asm. Этот файл представляет собой дизассемблированный загрузчик, который создается после компи-

ляции. Он позволяет увидеть, где именно в физической памяти находится код загрузчика, и упрощает отслеживание с помощью GDB работы загрузчика. Кроме того, obj/kern/kernel.asm содержит дизассемблированное ядро JOS, которое также может быть полезно при отладке. Работа с отладчиком GDB описана в разд. 3.4.1.

► Установите точку прерывания по адресу 0x7C00 – в место, где начнет загружаться загрузчик. Продолжите выполнение до срабатывания этой точки. Затем последовательно выполняйте инструкции, отслеживая выполнение в исходном коде и дизассемблированном загрузчике, используя просмотр инструкций в дизассемблиированном виде с помощью GDB. Сравнивайте исходный код загрузчика с его дизассемблированным кодом (*obj/boot/boot.asm*) и выводом GDB. Также может быть полезным просмотр содержимого регистров с помощью GDB.

Выполните код до функции *bootmain()*, находящейся в *boot/main.c*, а затем до функции *readsect()*. Найдите инструкции, соответствующие каждому из выражений в функции *readsect()*. Дойдите до конца функции *readsect()* и обратно до функции *bootmain()*. Найдите начало и конец цикла *for*, который читает с диска оставшиеся секторы ядра. Найдите, какой код будет выполняться после окончания цикла. Установите там точку прерывания и продолжите выполнение до нее, а затем выполняйте инструкции до конца работы загрузчика.

Ответьте на вопросы:

- В какой момент процессор начинает выполнять 32-битный код? Что заставляет его переключиться из 16-битного режима в 32-битный?
- Какая инструкция является последней выполняющейся инструкцией загрузчика, а какая – первой инструкцией ядра, которое он только что загрузил?

- Где в памяти находится первая инструкция ядра?
- Каким образом загрузчик определяет, сколько секторов он должен прочитать, чтобы полностью загрузить ядро с диска?

### 2.1.4 Загрузка ядра

Рассмотрим более подробно вторую часть загрузчика, находящуюся в файле boot/main.c.

Ядро находится на диске в двоичном формате ELF (Executable and Linkable Format, «Исполняемый и компонуемый формат»). При компиляции и компоновке программы на С (например, ядра JOS) компилятор создает на основе каждого исходного файла (.c) объектный (.o), содержащий инструкции процессора в двоичном формате. Затем компоновщик объединяет все скомпилированные объектные файлы в единый двоичный образ, который в случае с JOS является двоичным файлом в формате ELF и находится в obj/kern/kernel.

ELF-файл можно рассматривать как заголовок с метаданными, за которыми идет несколько секций программы, каждая из которых представляет собой непрерывный фрагмент кода или данных, предназначенный для загрузки в память по указанному адресу. Загрузчик никак не меняет код или данные, он загружает их в память и начинает выполнение.

Метаданные в ELF-файлах состоят из ELF-заголовка фиксированной длины и заголовка программы переменной длины, в котором перечислены все секции, которые должны быть загружены. Определения на языке С для этих заголовков, позволяющие программно обрабатывать их, находятся в inc/elf.h. Нас интересуют следующие секции программы:

- .text: исполняемые инструкции программы;
- .rodata: данные только для чтения (например, строковые константы);

- .data: инициализированные данные программы, например, глобальные переменные, объявленные с помощью конструкций вида `int x = 5;`

Когда компоновщик вычисляет расположение программы в памяти, он оставляет пространство для неинициализированных глобальных переменных в секции .bss, которая следует сразу после .data. Стандарт языка C требует, чтобы такие переменные инициализировались нулевым значением. Таким образом, нет необходимости хранить содержимое .bss в ELF-файле. Вместо этого компоновщик записывает только адрес и размер .bss. Загрузчик или сама программа должны инициализировать секцию .bss нулевыми значениями.

Изучите полный список названий, размеров и адресов всех секций в ядре с помощью objdump:

```
$ objdump -h obj/kern/kernel
```

Вы увидите значительно больше секций, но для данной работы имеют значение только вышеперечисленные.

Обратите особое внимание на значения VMA (адрес связывания) и LMA (адрес загрузки) секции .text. Адрес загрузки — это тот адрес, по которому секция должна быть загружена в память. В структуре, описывающей формат ELF, это значение хранится в поле `ph->p_ra`. Адрес связывания секции — это адрес, с которого секция ожидает начать выполнение. Компоновщик использует адрес связывания, например, при вычислении адреса глобальной переменной, в результате чего исполняемый файл, как правило, не будет работать, если он начинает выполнение с неправильного адреса. Как правило, адреса связывания и загрузки совпадают. Например, посмотрите на секцию .text загрузчика:

```
$ objdump -h obj/boot/boot.out
```

BIOS загружает загрузочный сектор в память, начиная с адреса 0x7C00, который, таким образом, является адресом

загрузки загрузочного сектора. С этого же места загрузочный сектор начинает выполнение, то есть этот адрес является одновременно и адресом связывания. Адрес связывания устанавливается при передаче компоновщику параметра `-Ttext 0x7C00` в файле `boot/Makefrag`.

► *Проследите выполнение первых нескольких инструкций загрузчика еще раз и определите первую команду, которая выполнит неправильные действия, если адрес связывания загрузчика будет неправильным. Затем измените адрес связывания в boot/Makefrag, чтобы он был неправильным, выполните make clean, затем make, и проследите выполнение загрузчика снова, чтобы увидеть, что произойдет. Не забудьте изменить адрес связывания обратно и снова выполнить make clean!*

Посмотрите на адреса загрузки и связывания ядра. В отличие от загрузчика, в ядре эти два адреса не совпадают: ядро загружается в память по низкому адресу (1 МБ), но ожидает начать выполнение с высокого адреса. Мы выясним, за счет чего ядро правильно работает, в следующих лабораторных работах.

Помимо информации о секциях, есть еще одно важное поле в заголовке ELF: `e_entry`. В этом поле находится адрес точки входа в программу — адрес в секции `.text`, с которого программа должна начать выполнение. Вы можете просмотреть адрес точки входа:

```
$ objdump -f obj/kern/kernel
```

Таким образом, загрузчик ELF в `boot/main.c` читает каждую секцию ядра с диска в память по адресу, соответствующему адресу связывания секции, а затем переходит к точке входа в ядро.

► *Перезагрузите машину (выйдите из QEMU и GDB и запустите их снова). Изучите 8 слов по адресу 0x00100000 в*

*момент, когда BIOS передает управление загрузчику, а затем еще раз в момент, когда загрузчик передает управление ядру. Почему они различаются? Что находится там на второй точке прерывания?*

### 2.1.5 Монитор

Вторая часть раздела соответствует лабораторной работе №2. В ней используется тот же код, что и в работе №1.

Взаимодействие ядра с пользователем в данный момент осуществляется с помощью монитора — интерактивной программы управления. Для тестирования ядро JOS настроено таким образом, что оно выводит данные не только на экран, но и в последовательный порт, данные из которого QEMU использует в качестве стандартного вывода; ввод данных также осуществляется как с клавиатуры, так и из последовательного порта. Таким образом, команды монитора можно вводить как в окне QEMU, так и в окне терминала. В данный момент заданы только две команды монитора: help и kerninfo.

► Просмотрите файл *kern/monitor.c*. Как задаются команды монитора и как происходит их вызов? Каким образом происходит вывод текста в консоль? Добавьте свою команду, которая выводит в консоль произвольный текст.

Для вывода данных монитор использует функции форматированного вывода (в частности, sprintf). Обычно для этого используются функции из стандартной библиотеки C, однако в данном случае она избыточна. Поэтому JOS использует собственный упрощенный вариант этих функций.

► Просмотрите файлы *kern/printf.c*, *lib/printfmt.c* и *kern/console.c*. В одном из них пропущен небольшой фрагмент кода: печать чисел в восьмеричном виде при использовании последовательности %. Найдите это место и восполните недостающий фрагмент.

## 2.1.6 Стек

Для работы с локальными переменными и вызова функций используется область памяти, называемая стеком. Указатель стека (регистр ESP) указывает на вершину стека – начало области стека, которая используется в настоящее время. Вся память ниже этого указателя является свободной. При добавлении значения в стек указатель стека уменьшается, а затем нужное значение записывается в новую вершину стека. При извлечении значения происходит чтение вершины стека, а затем его указатель увеличивается. В 32-битном режиме стек может содержать только 32-битные значения, поэтому ESP всегда кратно 4. Регистр ESP используется различными инструкциями x86, такими как call.

Регистр EBP (указатель базы), в отличие от ESP, связан со стеком программным образом. При входе в С-функцию код пролога функции обычно сохраняет указатель базы предыдущей функции, записывая его в стек, а затем копирует текущее значение ESP в EBP на время выполнения функции. Если все функции в программе следуют этому правилу, то в любой момент выполнения программы можно проследить порядок вызова функций через стек, следя цепочке сохраненных указателей EBP, и определить, какая последовательность вызовов функций привела к достижению данного места выполнения программы. Эта возможность может быть особенно полезна, например, когда та или иная функция вызывает assert или panic из-за неправильных аргументов, но вы не уверены, какая именно функция передала эти аргументы. Трассировка стека позволяет найти эту функцию.

► *Определите, где происходит инициализация стека, и где стек расположен в памяти. Как ядро резервирует область памяти для стека? Какое место в этой области памяти является началом стека?*

► Чтобы ознакомиться с соглашением о вызове C, найдите адрес функции `test_backtrace` в файле `obj/kern/kernel.asm`, установите там точку прерывания и посмотрите, что происходит при каждом вызове функции после загрузки ядра. Сколько 32-разрядных слов каждый уровень `test_backtrace` добавляет в стек, что это за слова? Чтобы выполнить это упражнение, желательно использовать модифицированную версию QEMU. В противном случае вам придется вручную переводить все адреса в линейные.

Приведенное выше упражнение должно дать вам информацию, необходимую для реализации функции трассировки стека `mon_backtrace()`. Прототип для этой функции уже находится в `kern/monitor.c`. Вы можете написать функцию на C без использования ассемблера — для этого может оказаться полезной функция `read_ebp()` в `inc/x86.h`. Вы также должны добавить новую функцию в список команд монитора, чтобы она могла быть вызвана пользователем.

Функция трассировки должна отображать данные в следующем формате:

Stack backtrace:

```
ebp f0109e58 eip f0100a62 args 00000001 f0109e80 f0109e98  
f0100ed2 00000031
```

```
ebp f0109ed8 eip f01000d6 args 00000000 00000000 f0100058  
f0109f28 00000061
```

...

Первая строка соответствует выполняемой в данный момент функции (`mon_backtrace`), вторая — функции, которая вызвала `mon_backtrace` и так далее. Изучив файл `kern/entry.S`, вы найдете простой способ определить момент, когда нужно остановиться.

В каждой строке значение EBP соответствует базовому указателю на область стека, используемую данной функцией, то есть положению указателя стека сразу после того, как функция начала работать и код пролога функции установил базовый указатель. Значение EIP является указателем на адрес возврата: адресом, по которому будет передано управление после выхода из функции. Указатель на адрес возврата обычно указывает на следующую инструкцию после `call` (почему?). Пять шестнадцатеричных значений, перечисленных после `args`, соответствуют первым пяти аргументам функции, которые были добавлены в стек перед ее вызовом. Если функция была вызвана с меньшим количеством аргументов, не все эти значения будут иметь смысл. (Почему код трассировки не может обнаружить, сколько аргументов было на самом деле? Как это можно исправить?)

► *Реализуйте функцию трассировки, как описано выше.*

*Используйте тот же формат, что и в примере, иначе скрипт проверки не сработает. Если вы считаете, что функция работает правильно, запустите `make grade`, чтобы увидеть, соответствует ли вывод функции тому, что ожидает скрипт проверки, и исправьте его, если не соответствует.*

### 2.1.7 Отладочная информация

В данный момент функция трассировки выводит адреса функций в стеке, которые привели к выполнению `mon_backtrace()`. Однако на практике удобно знать имена функций, соответствующих этим адресам — например, чтобы узнать, в каких функциях может находиться ошибка, приводящая к краху ядра. Для этого используется отладочная информация, которая создается компилятором и компоновщиком при сборке двоичного файла (в данном случае — образа ядра). Отладочная информация может храниться

в самом исполняемом файле или отдельно от него. Существуют различные форматы отладочных данных, такие как STABS, COFF, DWARF. В JOS используется формат STABS, а отладочные данные хранятся в самом образе ядра.

Для работы с отладочными данными можно использовать функцию `debuginfo_eip()`, которая ищет значение EIP в таблице символов и возвращает отладочную информацию для этого адреса. Эта функция определена в `kern/kdebug.c`.

► Измените функцию трассировки стека таким образом, чтобы она отображала имена функций, файлов и номера строк.

Откуда берутся значения `__STAB_*` в `debuginfo_eip?` Чтобы найти ответ, попробуйте сделать следующее:

- Просмотрите файл `kern/kernel.ld` на предмет `__STAB_*`.
- Выполните `objdump -h obj/kern/kernel`.
- Выполните `objdump -G obj/kern/kernel`.
- Выполните `gcc -pipe -nostdinc -O2 -fno-builtin -I. -MD -Wall -Wno-format -DJOS_KERNEL -gstabs -c -S kern/init.c`, после чего просмотрите файл `init.s`.
- Проверьте, загружает ли загрузчик в память таблицу символов при загрузке ядра.

Дополните реализацию функции `debuginfo_eip` вызовом `stab_binsearch` для нахождения номера строки в файле.

Добавьте команду монитора `backtrace`, вызывающую функцию `mon_backtrace`, и расширьте реализацию `mon_backtrace` вызовом `debuginfo_eip` для вывода номера строки для каждого кадра стека в следующем виде:

```
K> backtrace
```

```
Stack backtrace:
```

```
    ebp f010ff78  eip f01008ae  args 00000001 f010ff8c 00000000  
f0110580 00000000
```

```
kern/monitor.c:143: monitor+106
ebp f010ffd8 eip f0100193 args 00000000 00001aac 00000660
00000000 00000000
kern/init.c:49: i386_init+59
ebp f010fff8 eip f010003d args 00000000 00000000 0000ffff
10cf9a00 0000ffff
kern/entry.S:70: <unknown>+0
K>
```

Для каждого значения EIP выводится имя файла и номер строки, за ними следуют имя функции и смещение значения EIP от начала этой функции (например, *monitor*+106 означает, что EIP указывает на 106-й байт от начала функции *monitor*). Обратите внимание: чтобы скрипт проверки работал правильно, имя функции и файла должны выводиться на отдельной строке.

Строки, не завершенные нулем (в том числе и строки из таблиц STABS), можно легко вывести с помощью функции *printf* следующим образом: вызов *printf*(“%.*\*s*”, *length*, *string*); выведет максимум *length* байт из строки *string* (более подробную информацию о работе функции *printf* можно получить на ее *ман-странице*).

Вы можете обнаружить, что некоторые вызовы функций (например, *runcmd()*) отсутствуют в трассировке. Это происходит из-за встраивания их вызовов компилятором. Кроме того, из-за других оптимизаций, вносимых компилятором, могут выводиться неправильные номера строк. Если удалить опцию компилятора *-O2* в *GNUMakefile*, трассировка станет более правильной (но ядро будет работать несколько медленнее).

После выполнения этого задания рекомендуется проверить разработанный код с помощью *make grade*. Скрипт должен выводить OK для всех тестов.

*По окончании выполнения работы следует создать новую ветку `working-lab2` (разд. 3.2.14) и сохранить в ней внесенные изменения (разд. 3.2.6). После этого нужно добавить удаленный репозиторий (разд. 3.2.17) для сохранения результатов по предоставленному преподавателем адресу и отправить в него эту ветку (разд. 3.2.22).*

## **2.2 Многопоточная ОС без виртуальной памяти**

Одним из основных ресурсов компьютера является центральный процессор. Для эффективного использования процессора операционная система должна управлять задачами, которые выполняются на нем. Современные компьютеры, как правило, должны выполнять несколько задач одновременно. В случае с персональным компьютером помимо той программы, с которой в данный момент осуществляется взаимодействие, могут выполняться программы обмена сообщениями, музыкальные проигрыватели, программы синхронизации с облачным хранилищем и т. д. Веб-сервер может обрабатывать сразу несколько запросов, часть из которых может ожидать обмена данными с диском. Даже вычислительные программы для более полного использования современных многоядерных процессоров рекомендуется по возможности разделять на несколько одновременно выполняющихся вычислений.

С точки зрения операционной системы для управления одновременно выполняющимися задачами используется модель процессов. При этом процессор переключается между процессами по определенному алгоритму, заданному операционной системой в виде планировщика процессов, и предоставляет каждому процессу определенный интервал времени для выполнения. Процесс — это экземпляр выполняемой программы, включающий в себя помимо ее кода значения регистров процессора и прочие необходимые для выполнения данные. Концептуально каждый процесс использует свой собственный виртуальный процессор.

Более подробно об управлении процессами можно прочитать, например, в главе 2 книги Э. Таненбаума «Современные операционные системы» [10].

В данном разделе в JOS добавляется возможность запускать одновременно несколько процессов, а также все сопутствующие возможности, требуемые для их работы (управление процессами, планировщик процессов и т. д.), причем все процессы работают в едином адресном пространстве, то есть могут иметь прямой доступ к памяти друг друга и к памяти ядра ОС. В связи с этим ядро и процессы не могут быть защищены друг от друга, и любая ошибка в программе может привести к порче данных ядра или другого процесса, что некритично для учебной ОС, но недопустимо для настоящей. Данная проблема будет решена в следующем разделе.

### **2.2.1 Управление процессами**

Первая часть раздела соответствует лабораторной работе №3. Для получения файлов и изменений, необходимых для ее выполнения, следует обновить удаленный репозиторий, создать новую ветвь под названием `working-lab3`, после чего выполнить слияние с ней ветви `lab3`, которая появилась в репозитории. О работе с `git` можно прочитать в разд. 3.2.

Эта лабораторная работа посвящена реализации основных возможностей ядра, необходимых для запуска процессов в режиме ядра без виртуальной памяти. В частности, нужно реализовать создание в ядре JOS структур для отслеживания процессов, функции создания нового процесса, загрузки образа программы и запуска процесса. Процессы на данном этапе будут функционировать в режиме ядра. Некоторые участки кода защищены макросом `CONFIG_KSPACE`. Это необходимо для того, чтобы в дальнейшем, когда процессы будут работать в пользо-

вательском режиме, этот специфичный код не использовался или использовался его альтернативный вариант.

В этой работе может быть полезной возможность GCC встраивать в код программы фрагменты на языке ассемблера — для этого используется оператор `asm`. Работа может быть выполнена и без их использования, однако такие фрагменты уже присутствуют в исходном коде, и их необходимо понимать.

Новый заголовочный файл `inc/env.h` содержит основные определения для процессов в JOS. Ядро использует структуру `Env` (от `environment` — окружение; название подчеркивает отличие интерфейса управления процессами от традиционного, используемого в UNIX-подобных операционных системах) для отслеживания каждого процесса. В результате выполнения этой работы будет создан только один процесс, но возможность создания нескольких процессов будет добавлена.

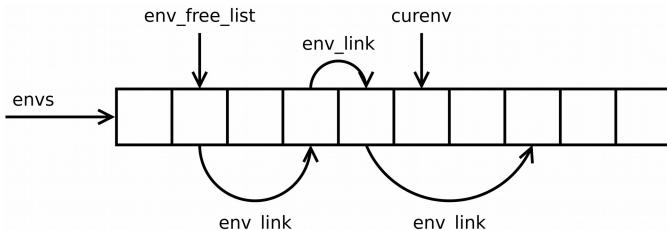


Рисунок 2.2.1: Переменные, используемые для управления процессами

Как можно видеть в `kern/env.c`, ядро поддерживает три основных глобальных переменных, относящихся к процессам:

```
struct Env *envs; // Массив всех процессов, используемых и свободных
struct Env *curenv = NULL; // Указатель на текущий процесс
static struct Env *env_free_list; // Список свободных процессов
```

После того, как JOS запущена, указатель envs указывает на массив структур Env, соответствующих всем возможным процессам в системе. Ядро JOS поддерживает максимум NENV (константа, определенная в inc/env.h) одновременно активных процессов, хотя обычно таких процессов гораздо меньше. После того, как массив envs выделен, он содержит один экземпляр структуры Env для каждого из NENV возможных процессов.

Ядро JOS содержит все неактивные структуры Env в связном списке env\_free\_list, что дает возможность легкого выделения и освобождения процессов путем их добавления в список и удаления из него.

Ядро также использует curenv для отслеживания выполняемого в данный момент процесса. Во время загрузки ядра, до первого запуска процесса, curenv равен NULL.

Структура Env определена в inc/env.h следующим образом:

```
struct Env {  
    struct Trapframe env_tf;  
    struct Env *env_link;  
    evid_t env_id;  
    evid_t env_parent_id;  
    enum EnvType env_type;  
    unsigned env_status;  
    uint32_t env_runs;  
};
```

Рассмотрим подробнее поля этой структуры:

- *env\_tf*:

Структура, определенная в inc/trap.h. Содержит сохраненные значения регистров для процесса в момент, когда процесс не выполняется, а выполняется код ядра или другой процесс. Ядро сохраняет эти зна-

чения при переключении процессов, чтобы работа процесса в дальнейшем могла быть возобновлена в том же месте, где она была прервана.

- *env\_link*:

Ссылка на следующий Env в `env_free_list`. Сама переменная `env_free_list` указывает на первый свободный процесс в списке.

- *env\_id*:

Значение, однозначно идентифицирующее процесс, который в настоящее время использует данную структуру Env (т.е. данный слот в массиве `envs`). После завершения процесса ядро может использовать ту же структуру Env для нового процесса, но новый процесс будет иметь другое значение `env_id`.

- *env\_parent\_id*:

Идентификатор (`env_id`) родительского процесса, т.е. процесса, который создал данный. Таким образом, процессы формируют «родословную», которая может быть полезна, например, для контроля доступа.

- *env\_type*:

Тип процесса. В настоящий момент это поле может принимать только одно значение `ENV_TYPE_KERNEL`. В дальнейшем будет введено еще несколько типов процессов.

- *env\_status*:

Статус процесса. Принимает одно из следующих значений:

- `ENV_FREE`: структура неактивна и, следовательно, находится в `env_free_list`.
- `ENV_RUNNABLE`: структура соответствует процессу, который ждет возможности работать на процес-сопре.

- ENV\_RUNNING: структура соответствует процессу, работающему в данный момент.
- ENV\_NOT\_RUNNABLE: структура соответствует процессу, который активен, но в данный момент не готов к работе: например, ожидает межпроцессного взаимодействия (IPC) с другим процессом.
- ENV\_DYING: структура соответствует зомби-процессу. Зомби-процесс будет освобожден в следующий раз, когда он вызовет исключение ядра. В данной работе это значение не используется.

Как и Unix-процесс, процесс JOS связывает понятия «поток» и «адресное пространство». Поток определяется прежде всего сохраненными регистрами (поле env\_tf), а адресное пространство — закрепленной за каждым процессом индивидуальной областью памяти. Области памяти разных процессов не должны пересекаться. Сами адреса, которые закреплены за процессом, задаются компоновщиком при сборке. Чтобы запустить процесс на выполнение, ядро должно установить сохраненные значения регистров процессора.

### **2.2.2 Создание и запуск процесса**

Теперь следует написать в kern/env.c код, необходимый для запуска процесса. Так как файловая система отсутствует, ядро загружает статический двоичный образ программы в формате ELF, встроенный в само ядро.

GNUmakefile создает ряд двоичных образов в obj/prog/, а в kern/Makefrag эти файлы встраиваются непосредственно в ядро, как если бы они были файлами .o. Параметр командной строки компоновщика -b binary заставляет его вставлять эти файлы в образ ядра в том виде, в котором они находятся на диске, никак их не интерпретируя и не изменяя; с точки зрения компоновщика эти файлы могут являться даже не ELF-образами, а, например, текстовыми файлами или фотографи-

ями. В файле obj/kern/kernel.sym после сборки ядра можно видеть ряд символов, соответствующих этим файлам, с названиями вида \_binary\_obj\_prog\_test1\_start, \_binary\_obj\_prog\_test1\_end и \_binary\_obj\_prog\_test1\_size. Эти символы обеспечивают возможность ссылаться на эти встроенные двоичные файлы в обычном коде ядра.

В функции i386\_init() в kern/init.c находится код для запуска этих двоичных образов в качестве процессов. Тем не менее, критически важные функции настройки процессов не завершены, вы должны написать их самостоятельно.

► В файле env.c допишите следующие функции:

- *env\_init()*

*Инициализирует все структуры Env в массиве envs и добавляет их в env\_free\_list. Также вызывает env\_init\_percr, которая настраивает оборудование для сегментации с отдельными сегментами для уровня привилегий 0 (ядро) и 3 (пользователь).*

- *load\_icode()*

*Декодирует двоичный ELF-образ так же, как это уже делает загрузчик, и загружает его содержимое в адресное пространство нового процесса. Пока не следует обращать внимания на функцию bind\_functions.*

- *env\_create()*

*Выделяет процесс с помощью env\_alloc и загружает в него двоичный ELF-образ путем вызова load\_icode.*

- *env\_run()*

*Запускает процесс.*

В этих функциях может быть полезной новая строка форматирования sprintf %e — она выводит описание, соответствующее коду ошибки. Например,

r = -E\_NO\_MEM;

```
panic("env_alloc: %e", r);
```

остановит работу ядра с сообщением «env\_alloc: out of memory».

Ниже приведен график вызовов кода до точки, в которой запускается код процесса. Убедитесь, что вы понимаете цели каждого шага.

- start (kern/entry.S)
- i386\_init (kern/init.c)
  - cons\_init
  - env\_init
  - env\_create
  - sched\_yield
    - env\_run
      - env\_pop\_tf

После реализации всех вышеупомянутых функций следует скомпилировать и запустить ядро. Система должна запустить двоичное приложение test1, которое проработает до первой инструкции push. Так как стек процесса не был задан (это будет сделано в следующем задании), будет генерировано исключение общей защиты. Ядро обнаружит, что его оно также не может обработать, что создает двойное исключение, которое на данном этапе также не обрабатывается. Наконец, процессор генерирует «тройную ошибку» (triple fault). Обычно в таких случаях процессор сбрасываеться и система перезагружается. Это усложняет разработку ядра, поэтому модифицированная версия QEMU в таких случаях выводит содержимое регистров и сообщение «Triple fault».

В ближайшее время эта проблема будет решена, но сейчас можно использовать отладчик, чтобы убедиться, что осуществляется передача управления процессу. Запустите make qemu-gdb и установите точку прерывания в env\_pop\_tf,

которая передает управление процессу. Пройдите эту функцию с помощью команды `si`; непосредственно передачу управления процессу производит инструкция `ret`. Система должна сгенерировать тройную ошибку раньше нее, на инструкции `push` — она записывает некоторые данные в стек, но адрес стека в этот момент еще не задан. Если ошибка происходит раньше, код управления процессами или загрузки программы ошибочен.

► В функции `env_alloc (kern/env.c)` содержатся строки:

```
// LAB 3: your code here  
// e->env_tf.tf_esp = 0xf0210000;
```

*Закомментированная строка задает адрес стека для процесса. Раскомментируйте ее и исправьте таким образом, чтобы для каждого процесса (т.е. при каждом вызове `env_alloc`) задавался уникальный адрес стека.*

*Примечание: Не имеет смысла выделять под стек большие объемы памяти. Как правило, должно хватать двух страницных кадров.*

После выполнения этого упражнения программы должны работать до первой попытки вызова любой функции ядра (`cprintf`, `sys_yield`, `sys_exit`). Эти функции в программах представляют собой глобальные указатели, которые еще не инициализированы и равны нулю. Это проблема решается в дальнейших упражнениях.

### 2.2.3 Планировщик

Для поддержки переключения процессов необходим модуль, который определяет очередность работы процессов на основании некоторого алгоритма. Такой модуль называется планировщиком. Существуют различные алгоритмы планирования процессов, каждый из которых выполняет свои задачи и имеет свои положительные и отрицательные стороны. Напри-

мер, в ОС Linux используется планировщик CFS (Completely Fair Scheduler), основной задачей которого является максимизация использования процессорного времени при одновременном сохранении максимальной интерактивности системы [9]. Одним из простейших алгоритмов планирования процессов является Round-robin: в нем процессы выбираются для выполнения циклически по очереди, без использования приоритетов и других эвристик. Именно этот алгоритм следует реализовать для планирования процессов в JOS.

- В функции `sched_yield` (`kern/sched.c`) необходимо реализовать простейший алгоритм планировщика *Round-robin*, в котором должны учитываться состояния процессов `ENV_RUNNABLE` и `ENV_RUNNING`. Комментарии в функции `sched_yield` описывают некоторые подробности задачи.
- В файле `kern/entry.S` присутствует функция `sys_yield`. Эту функцию процессы вызывают для добровольной передачи управления ядру. В ней происходит сохранение контекста процесса, переключение на контекст ядра и передача управления планировщику. В этом же файле присутствует функция `sys_exit`, вызываемая процессом для завершения работы.  
Допишите `sys_exit` по аналогии с `sys_yield` (примечание: в `sys_exit` сохранение контекста процесса, отдающего управление, не требуется).

## 2.2.4 Связывание

- В функции `load_icode` (`kern/env.c`) помимо загрузки образа процесса по необходимым ему адресам (эта задача была решена в предыдущих упражнениях) необходимо реализовать связывание (функция `bind_functions`) для загружаемых программ, чтобы они могли вызы-

вать функции ядра. Обратите внимание на закомментированные строки:

```
*((int *) 0xf0302008) = (int) &cprintf;  
*((int *) 0xf0302010) = (int) &sys_exit;  
*((int *) 0xf0302004) = (int) &sys_yield;  
*((int *) 0xf020200c) = (int) &sys_exit;  
*((int *) 0xf0202004) = (int) &sys_yield;
```

В них по адресам глобальных указателей на функции записываются адреса функций ядра. Необходимо сделать то же самое, но автоматическим образом. Для этого в функции *bind\_functions* нужно получить адреса глобальных переменных из таблицы символов ELF-файла (подобно тому, как это делают программы *nm -n*, *objdump -x*; обратите внимание на функции в файле *lib/string.c* и на раздел *Symbol Table* в спецификации ELF-формата) и каждой глобальной переменной, имя которой совпадает с именем одной из функций ядра, присваивать адрес последней. Адреса функций ядра можно получить из отладочной информации. Для этого необходимо реализовать функцию *find\_function* (*kern/kdebug.c*). Строковые имена функций хранятся в *stabstr*, их адреса в *stab* (поле *n\_value*).

После выполнения всех заданий при запуске make чети вывод IOS должен выглядеть примерно следующим образом:

```
[00000000] new env 00001000  
[00000000] new env 00001001  
envrun RUNNABLE: 1  
HERE  
envrun RUNNABLE: 0  
envrun RUNNABLE: 1  
envrun RUNNABLE: 0
```

```
envrun RUNNABLE: 1  
envrun RUNNABLE: 0  
envrun RUNNABLE: 1  
envrun RUNNABLE: 0  
[00001000] free env 00001000  
envrun RUNNABLE: 1  
envrun RUNNING: 1  
[00001001] free env 00001001
```

► По окончании выполнения работы следует сохранить внесенные изменения и отправить их в удаленный репозиторий.

### 2.2.5 Управление часами реального времени

Следующая часть раздела соответствует лабораторной работе №4. Для получения файлов, необходимых для выполнения дальнейших заданий, следует обновить удаленный репозиторий, создать новую ветвь под названием working-lab4, после чего выполнить слияние с ней ветви lab4, которая появилась в репозитории. О работе с git можно прочитать в разд. 3.2.

Представленные ниже задания посвящены работе с часами реального времени (RTC – Real-Time Clock). Часы реального времени являются одним из механизмов работы с реальным временем в процессорах Intel (существуют и другие механизмы: улучшенный программируемый контроллер прерываний APIC, программируемый интервальный таймер PIT и т. д.). Работа с реальным временем необходима для реализации планировщика процессов, который должен учитывать и контролировать время работы каждого процесса.

После запуска компьютера прерывания от часов реального времени отключены. После инициализации часы будут периодически генерировать прерывание 8 (IRQ\_CLOCK).

Для управления часами используются три байта в памяти CMOS, называемые регистрами А, В и С. Для работы с этими регистрами используются порты ввода-вывода 0x70 и 0x71 (в JOS для работы с ними используются константы IO\_RTC\_CMND и IO\_RTC\_DATA).

Более подробная информация о работе с часами содержится в их документации [7].

► Необходимо дописать в файле *kern/kclock.c* недостающий код функций *rtc\_init*, в которой происходит инициализация часов *rtc*, и *rtc\_check\_status*, в которой происходит проверка статуса часов. Алгоритм инициализации выглядит следующим образом:

1. Переключение на регистр часов В.
2. Чтение значения регистра В из порта ввода-вывода.
3. Установка бита RTC\_PIE.
4. Запись обновленного значения регистра в порт ввода-вывода

Для проверки статуса часов в функции *rtc\_check\_status* необходимо прочитать значение регистра часов С.

► Во время выполнения кода ядра ОС прерывания на процессоре должны быть замаскированы. Это является архитектурной особенностью ядра JOS. Для этого необходимо расставить в файле *kern/entry.S* в нужных местах три инструкции *cli*, которые осуществляют маскирование прерываний. Поведение ОС в данном случае недетерминировано; при установке инструкций *cli* в неверных местах ОС может работать правильно. Для проверки правильности выполнения задания в модифицированной версии *qemu* был добавлен параметр командной строки *-oscourse*; запуск JOS с данным параметром осуществляется командой *make qemu-oscheck*. В этом случае *qemu* генерирует прерывание *IRQ\_CLOCK* перед тем, как выполнить инструкцию *cli*.

*Кроме того, проверка бессмысленна до выполнения следующего задания.*

- После инициализации часов RTC и программируемого контроллера прерываний PIC необходимо размаскировать на контроллере линию IRQ\_CLOCK, по которой приходят прерывания от часов. Для этого можно использовать функцию *irq\_setmask\_8259A*.
- После того, как прерывание сгенерировано и обработано, перед вызовом планировщика необходимо прочесть регистр статуса RTC и отправить сигнал EOI на контроллер прерываний, сигнализируя об окончании обработке прерывания. Иначе дальнейшие прерывания от часов не будут генерироваться PIC. Для этого можно использовать функции *rtc\_check\_status()*, *pic\_send\_eoi()*.

Выполнив эти задания и запустив команду make qemu, вы увидите примерно следующий вывод:

```
6828 decimal is 15254 octal!
END: 0xf0123da8
enabled interrupts: 2 8
[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
envrun RUNNABLE: 0
envrun RUNNABLE: 1
HERE
envrun RUNNABLE: 2
TEST3
envrun RUNNABLE: 0
envrun RUNNABLE: 1
envrun RUNNABLE: 2
```

```
envrun RUNNABLE: 0
envrun RUNNABLE: 1
envrun RUNNABLE: 2
envrun RUNNABLE: 0
[00001000] free env 00001000
envrun RUNNABLE: 1
envrun RUNNABLE: 2
envrun RUNNABLE: 1
[00001001] free env 00001001
envrun RUNNABLE: 2
envrun RUNNING: 2
....
```

► В данный момент RTC работает на некой стандартной частоте. Измените процедуру *rtc\_init* так, чтобы прерывания от часов приходили раз в полсекунды. Для этого необходимо изменить делитель частоты, которому соответствуют младшие 4 бита регистра часов A.

Следующее упражнение заключается в реализации секундомера. В процессорах архитектуры x86 есть 64-битный регистр TSC (Time Stamp Counter), в котором находится число тактов процессора с его последнего сброса (reset). Инструкция rdtsc копирует значение TSC в регистры EDX:EAX. Зная частоту процессора (число тактов в секунду) и количество тактов между двумя моментами времени, можно легко вычислить прошедшее между ними время. Хотя данный метод дает приемлемую точность, в настоящее время он практически не ис-

пользуется (используется более сложный НРЕТ — High Precision Event Timer).

► В репозиторий были добавлены файлы `tsc.h` и `tsc.c`. В них уже реализован подсчет частоты процессора с помощью PIT (*Programmable Interval Timer*) — рекомендуется ознакомиться с этим кодом. Ваша задача заключается в реализации функций `timer_start` и `timer_stop` из `tsc.c`. Если функция `timer_stop` была вызвана после `timer_start`, она должна выводить на экран время в секундах между этими вызовами. В случае вызова `timer_stop` без предварительного вызова `timer_start` должна быть выведена ошибка.

Для проверки необходимо добавить команды монитора `start` и `stop`, которые будут вызывать `timer_start` и `timer_stop` соответственно. Чтобы система переходила в монитор, нужно отключить старт бесконечно выполняющихся программ в `kern/init.c`.

По окончании выполнения работы следует сохранить внесенные изменения и отправить в удаленный репозиторий.

## 2.2.6 Синхронизация между процессами

Последняя часть раздела соответствует лабораторной работе №5. Для получения файлов и изменений, необходимых для ее выполнения, следует обновить удаленный репозиторий, создать новую ветвь под названием `working-lab5`, после чего выполнить слияние с ней ветви `lab5`, которая появилась в репозитории. О работе с `git` можно прочитать в разд. 3.2.

При параллельной работе нескольких процессов с одной и той же областью памяти могут возникать так называемые состояния гонки. В случае с JOS, например, при выделении памяти процессы могут одновременно обращаться к списку свободных страниц, так как все процессы в настоящий момент

работают в одном адресном пространстве и вышеупомянутый список является для них общим.

В качестве примера состояния гонки рассмотрим простое выражение:

$$b = b + 1$$

Представим, что данное выражение одновременно выполняется двумя процессами, причем переменная  $b$  является общей для них, а ее начальное значение – 5.

Вот возможный пример порядка выполнения такой программы:

- Процесс 1 загружает  $b$  в регистр процессора.
- Процесс 2 загружает  $b$  в регистр процессора.
- Процесс 1 увеличивает значение регистра на 1 с результатом 6.
- Процесс 2 увеличивает значение регистра на 1 с результатом 6.
- Процесс 1 сохраняет значение регистра (6) в переменную  $b$ .
- Процесс 2 сохраняет значение регистра (6) в переменную  $b$ .

Начальным значением переменной  $b$  было 5, каждый из процессов увеличил ее на 1, однако окончательным результатом оказалось 6 вместо ожидаемого 7.

Для предотвращения состояний гонки используется синхронизация между процессами, реализуемая с помощью примитивов синхронизации, таких как мьютексы, семафоры, блокировки с активным ожиданием (спинлоки) и т. д. В следующем задании необходимо использовать реализованные в JOS спинлоки для синхронизации между процессами при выделении и освобождении памяти.

► В репозиторий были добавлены файлы `alloc.c` и `spinlock.c`. В первом находится исходный код аллокатора

*памяти, которым пользуются процессы для выделения и освобождения памяти. Во втором – механизм блокировок с активным ожиданием. Необходимо добавить в функции alloc и free (файл alloc.c) защиту блокировками (функции spin\_lock, spin\_unlock) таким образом, чтобы функции выделения и освобождения памяти не могли работать со списком страниц одновременно, чтобы процессы, которые одновременно их вызывают, не могли повредить список свободных страниц памяти.*

*Ошибка повреждения списочной структуры хранения блоков свободной памяти проявляет себя нерегулярно. Уменьшив интервал срабатываний таймера, можно увеличить вероятность ее возникновения. При возникновении ошибки на экран выводится строка вида: kernel panic at kern/alloc.c:22: Corrupted list.*

*По окончании выполнения работы следует сохранить внешние изменения и отправить в удаленный репозиторий.*

## **2.3 ОС с виртуальной памятью и процессами**

Помимо процессора, важным ресурсом компьютера является оперативная память. Часть операционной системы, которая ей управляет, называется диспетчером памяти. Простейший диспетчер памяти уже реализован в JOS и позволяет процессам и ядру выделять и освобождать память. Однако, как было сказано выше, на данном этапе развития JOS все процессы работают в едином адресном пространстве, то есть могут иметь прямой доступ к памяти друг друга и к памяти ядра ОС, что может привести к фатальным последствиям. Для решения этой проблемы используется модель виртуальной памяти.

В основе виртуальной памяти лежит идея, что у каждого процесса имеется свое собственное адресное пространство, разделенное на участки, называемые страницами. Каждая страница может быть отображена на участок физической памяти того же размера, но отображение всех возможных страниц сразу необязательно. При доступе к памяти программа использует не физические адреса, а виртуальные, которые затем с учетом таблицы страниц процесса транслируются процессором в физические. Кроме того, процессор может, например, ограничивать доступ процесса на запись в определенные страницы памяти — т. е. процесс может получить доступ к некоторым страницам памяти ядра без опасений, что он испортит его.

Еще одна проблема, которая решается при помощи механизма виртуальной памяти — это проблема фрагментации памяти. Когда операционная система работает в течении продолжительного времени, она выделяет фрагменты памяти для тех или иных целей, а затем освобождает их. В результате

свободные фрагменты памяти чередуются с все еще используемыми и может оказаться, что несмотря на наличие достаточного количества свободной памяти, нет ни одного свободного фрагмента нужного размера.

О поддержке управления памятью в процессорах x86 можно более подробно прочитать в главах 3, 4, 5 третьего тома руководства Intel [8]. На концептуальном уровне управление памятью описано в главе 3 уже упомянутой книги Э. Таненбаума [10].

В данном разделе в JOS добавляются возможности управления памятью: выделение и освобождение страниц, учет выделенных страниц, работа с виртуальной памятью, изоляция процессов друг от друга и от ядра с помощью механизма виртуальной памяти.

Первая часть раздела соответствует лабораторной работе №6. Для получения файлов и изменений, необходимых для выполнения работы, следует обновить удаленный репозиторий, создать новую ветвь под названием `working-lab6`, после чего выполнить слияние с ней ветви `lab6`, которая появилась в репозитории. О работе с `git` можно прочитать в разд. 3.2.

Далее будет реализован аллокатор памяти, который позволит ядру выделять и освобождать память. Этот аллокатор должен оперировать страницами памяти размером 4096 байт. Для его реализации нужно вести учет выделенных и свободных физических страниц, а также количества процессов, использующих каждую страницу.

### **2.3.1 Обход зависимости от местоположения**

Ядро операционной системы часто имеет очень высокий виртуальный адрес связывания, например, 0xF0100000, чтобы оставить нижнюю часть виртуального адресного пространства

для пользовательских программ. Причины этого станут яснее в последующих лабораторных работах.

При этом многие компьютеры не имеют физической памяти по адресу 0xF0100000, поэтому мы не можем рассчитывать на возможность хранить там ядро. Вместо этого мы будем использовать инструменты процессора для управления памятью, чтобы сопоставить виртуальный адрес 0xF0100000 (адрес связывания, по которому код ядра ожидает быть запущенным) физическому адресу 0x00100000 (по которому загрузчик загружает ядро в физическую память). Таким образом, хотя виртуальный адрес ядра достаточно высок, чтобы оставить достаточно адресного пространства для пользовательских процессов, оно загружается в физическую память в точку 1 МБ, сразу над ПЗУ BIOS. Этот подход требует, чтобы компьютер имел хотя бы несколько мегабайт физической памяти (чтобы физический адрес 0x00100000 был доступен), но это условие скорее всего выполняется любым компьютером, выпущенным после приблизительно 1990 года.

В этой и следующих лабораторных работах мы будем сопоставлять все нижние 256 МБ физического адресного пространства компьютера, от 0x00000000 до 0xFFFFFFFF, виртуальным адресам от 0xF0000000 до 0xFFFFFFF соответственно. Теперь вы должны понимать, почему JOS может использовать только первые 256 МБ физической памяти.

На данный момент сопоставлены только первые 4 МБ физической памяти, которых достаточно, чтобы дать возможность ядру запуститься и работать. Это делается с помощью написанных вручную и статически инициализируемых каталога и таблицы страниц в kern/entrypgdir.c. До установки флага CR0\_PG в kern/entry.S ссылки на память рассматриваются как физические адреса (строго говоря, это линейные адреса, но в boot/boot.S создается тождественное отображение из линейных адресов в физические). После установки CR0\_PG

ссылки на память являются виртуальными адресами, которые транслируются процессором в физические адреса. `entry_pgdir` транслирует виртуальные адреса в диапазоне от 0xF0000000 до 0xF0400000 в физические от 0x00000000 до 0x00400000, а также виртуальные адреса от 0x00000000 до 0x00400000 в физические от 0x00000000 до 0x00400000. Обращение по любому виртуальному адресу, который не находится в одном из этих двух диапазонов, вызовет аппаратное исключение, которое, так как мы еще не создали обработку прерываний, заставит QEMU сохранить состояние виртуальной машины и выйти (или бесконечно перезагружаться, если вы не используете исправленную версию QEMU).

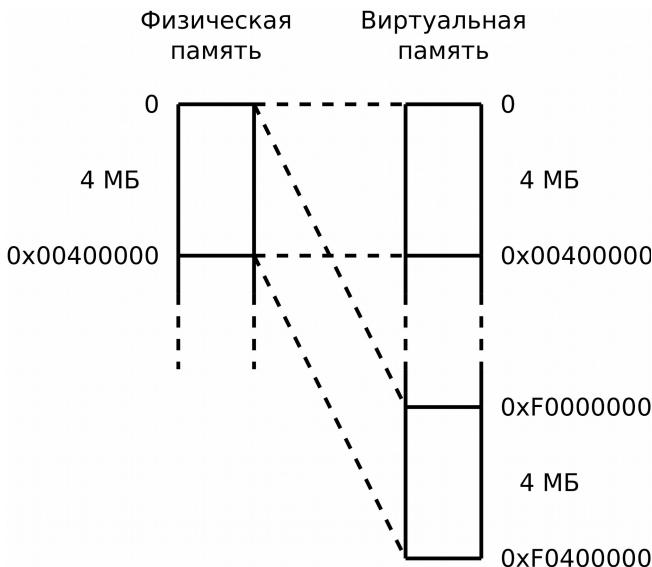


Рисунок 2.3.1: Отображение памяти

► Используйте QEMU и GDB, чтобы проследить работу ядра JOS и остановиться на `movl %eax, %cr0`. Изучите память по адресам 0x00100000 и 0xF0100000. Теперь выполните одну инструкцию, используя команду GDB `stepi`. Еще раз изучите память по адресам 0x00100000 и

`0xF0100000`. Убедитесь, что вы понимаете, что произошло.

Какая инструкция первой не сможет выполниться правильным образом, если отображение не будет установлено? Закомментируйте `movl %eax, %cr0` в `kern/entry.S`, проследите выполнение и проверьте, были ли вы правы.

Запишите ответы на эти вопросы в файл `lab6-answers.txt` и поместите этот файл в репозиторий.

### 2.3.2 Управление страницами

Операционная система должна отслеживать, какие части физической памяти в настоящий момент свободны, а какие используются. Данную задачу выполняет модуль, который называется менеджером памяти. Для этого он использует связный список структур  `PageInfo`, каждая из которых соответствует физической странице. Вы должны написать аллокатор страниц, прежде чем вы можете реализовать остальную часть кода управления виртуальной памятью, т. к. код управления таблицами страниц должен выделять физическую память под сами таблицы страниц.

► В файле `kern/rmap.c` необходимо исправить или написать функции, список которых приведен ниже. Для тестирования работы функций удалите или закомментируйте строку `panic(...)` в функции `mem_init()`.

- `boot_alloc()`

Аллокатор, используемый при запуске системы виртуальной памяти JOS. Если параметр  $n > 0$ , выделяет достаточное для хранения  $n$  байт количество страниц. Не инициализирует память. Если  $n = 0$ , возвращает адрес следующей свободной страницы, ничего не выделяя. Если свободной памяти нет — вызывает `kernel panic`. В файле `kern/entrypgdir.c` вручную задана таблица страниц

для первых 4 МБ памяти. Данная функция должна выделять память из этой области.

- *mem\_init()* (только до вызова *check\_page\_free\_list(1)*)  
Создает таблицу страниц, причем только для адресного пространства ядра (начиная с UTOP). Выделите массив *pages* из структур  *PageInfo* количеством *npages*. Обратите внимание, что необходимо использовать написанный выше аллокатор (*boot\_alloc()*). Этот массив используется для управления физическими страницами: каждая структура соответствует одной странице.
- *page\_init()*

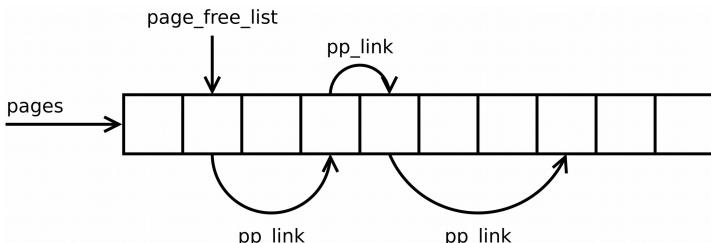


Рисунок 2.3.2: Список свободных страниц

Инициализирует список свободной памяти *page\_free\_list*. В этом списке должны находиться все еще не выделенные страницы из *pages*. После ее вызова использовать *boot\_alloc()* нельзя — памятью будут управлять заданные ниже функции *page\_alloc()* и *page\_free()*. В данный момент функция добавляет в список все страницы из *pages*, однако на самом деле не все они свободны. Первая страница соответствует структурам BIOS и IDT. Остальная часть базовой памяти свободна. Затем от *IOPHYSMEM* до *EXTPHYSMEM* находится т.н. *IO Hole* — струк-

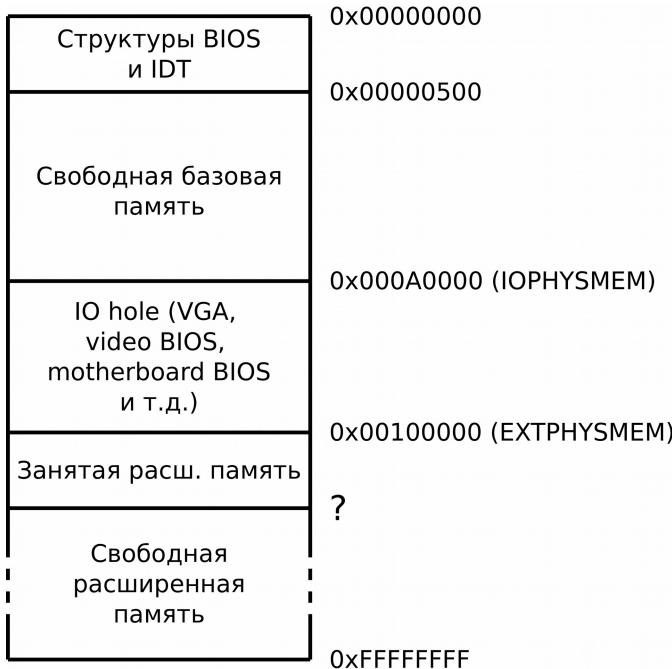


Рисунок 2.3.3: Занятая и свободная память

туры BIOS, предназначенные для ввода-вывода. Далее находится расширенная память, часть которой занята (уже выделена с помощью `boot_alloc()`), а часть свободна. Необходимо изменить функцию таким образом, чтобы в список добавлялись только свободные страницы.

- `page_alloc()`

Выделяет физическую страницу. Если в параметре `alloc_flags` стоит флаг `ALLOC_ZERO`, функция заполняет ее нулями. Не увеличивает счетчик ссылок на страницу (`pp_ref`) – это должна сделать та функция, которая вызывает `page_alloc()`. Для реализации данной функции могут быть по-

лезны функции *page2kva()* (возвращает виртуальный адрес страницы) и *memset()*.

- *page\_free()*

Возвращает страницу в список свободных (*page\_free\_list*). Функцию следует вызывать, только если счетчик ссылок на страницу равен 0.

Кроме того, необходимо добавить в монитор команду, выводящую список всех физических страниц с указанием на то, выделены они или свободны, в сокращенном виде, например:

```
1 ALLOCATED
2..160 FREE
161..323 ALLOCATED
324..1020 FREE
1021..1024 ALLOCATED
1025..16639 FREE
```

Функции *check\_page\_free\_list()* и *check\_page\_alloc()*, запускаемые в *mem\_init()*, проверяют реализацию аллокатора. Запустите JOS и убедитесь в том, что *check\_page\_alloc()* сообщает об успешном выполнении. При необходимости исправьте код так, чтобы эта функция срабатывала. Для отладки может быть полезным использование GDB и добавление различных вызовов *assert()*. При запуске JOS вы должны увидеть следующий вывод:

```
check_page_alloc() succeeded!
kernel    panic    at    kern/pmap.c:696:    assertion    failed:
page_insert(kern_pgdir, pp1, 0x0, PTE_W) < 0
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

При запуске make grade система должна проходить первую проверку (Physical Page Allocator).

- *По окончании выполнения работы следует сохранить внесенные изменения и отправить их в удаленный репозиторий.*

### 2.3.3 Виртуальная память

Вторая часть раздела соответствует лабораторной работе №7. Для получения файлов и изменений, необходимых для выполнения работы, следует обновить удаленный репозиторий, создать новую ветвь под названием working-lab7, после чего выполнить слияние с ней ветви lab7, которая появилась в репозитории. О работе с git можно прочитать в разд. 3.2.

Перед продолжением ознакомьтесь с архитектурой управления памятью x86, а именно с сегментацией и со страничной трансляцией. Прочтайте главы 3 и 4 руководства Intel [8]. Разделы, касающиеся сегментации, не следует пропускать: JOS использует страничную организацию памяти, но сегментная трансляция и сегментная защита памяти не могут быть отключены на процессорах x86, так что вам потребуются базовые знания о них.

В терминологии x86 виртуальный адрес состоит из селектора сегмента и смещения внутри сегмента. Линейный адрес получается после сегментной трансляции, но до страничной трансляции. Физический адрес является результатом сегментной и страничной трансляции; в конечном итоге, на аппаратном уровне используется именно он.

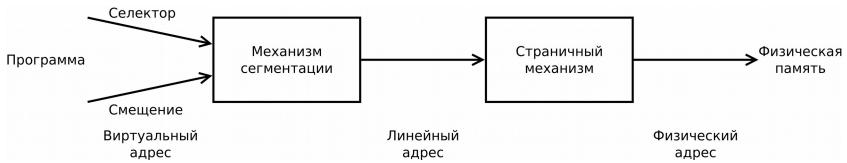


Рисунок 2.3.4: Трансляция адресов

Указатель в языке C содержит компонент «смещение» виртуального адреса. В boot/boot.S мы установили глобальную таблицу дескрипторов (GDT), которая отключила сегментную трансляцию путем установки всех базовых адресов сегментов в 0 и размеров в 0xFFFFFFFF. Таким образом, «селектор» не имеет никакого эффекта и линейный адрес всегда равен смещению виртуального адреса.

При страницной трансляции (см. рис. 2.3.5) информация об отображении физических страниц на виртуальные хранится в каталоге страниц. Адрес текущего каталога страниц должен находиться в контролльном регистре CR3; таким образом, процессор может переключаться между несколькими каталогами страниц для разных процессов и ядра.

В каталоге страниц хранятся структуры PDE (Page Directory Entry — запись каталога страниц), в которых содержится адрес физической страницы для таблицы страниц и флаги для нее. В свою очередь, в таблице страниц содержатся структуры PTE (Page Table Entry — запись таблицы страниц), в которых содержится адрес конкретной физической страницы и флаги для нее. Таким образом, каталог страниц является двухуровневым, а виртуальный адрес состоит из трех компонент: номера PDE, номера PTE и смещения на конкретной странице.

В PDE заданы следующие флаги:

- S — Size, размер страницы: если установлен, используются страницы размером 4 МБ, если нет — 4 КБ.

- A — Accessed, обращение. Процессор устанавливает этот флаг каждый раз, когда обращается к таблице для чтения или записи.
- D — Cache Disabled, запрещение кэша. Запрещает кэширование таблицы страниц.
- W — Write Through, сквозная запись. Управляет кэшированием страниц (в JOS не используется).
- U — User, пользовательская таблица. Если установлен — таблица доступна из пользовательского режима, если нет — только из режима ядра.
- R — Read/Write, чтение/запись. Определяет тип доступа к таблице: если установлен — таблица доступна на запись, если нет — только на чтение.
- P — Present, присутствие. Означает, что страница отображена и может использоваться при трансляции адреса.

В PTE заданы другие флаги:

- G — Global. Влияет на процесс кэширования страниц (в JOS не используется).
- D — Dirty, изменение. Процессор устанавливает этот флаг каждый раз, когда обращается к странице для записи.
- A — Accessed, обращение. Процессор устанавливает этот флаг каждый раз, когда обращается к странице для чтения или записи.
- C — Cache Disabled, запрещение кэша. Запрещает кэширование страницы.
- W — Write Through, сквозная запись. Управляет кэшированием страниц (в JOS не используется).
- U — User, пользовательская страница. Если установлен — страница доступна из пользовательского режима, если нет — только из режима ядра.

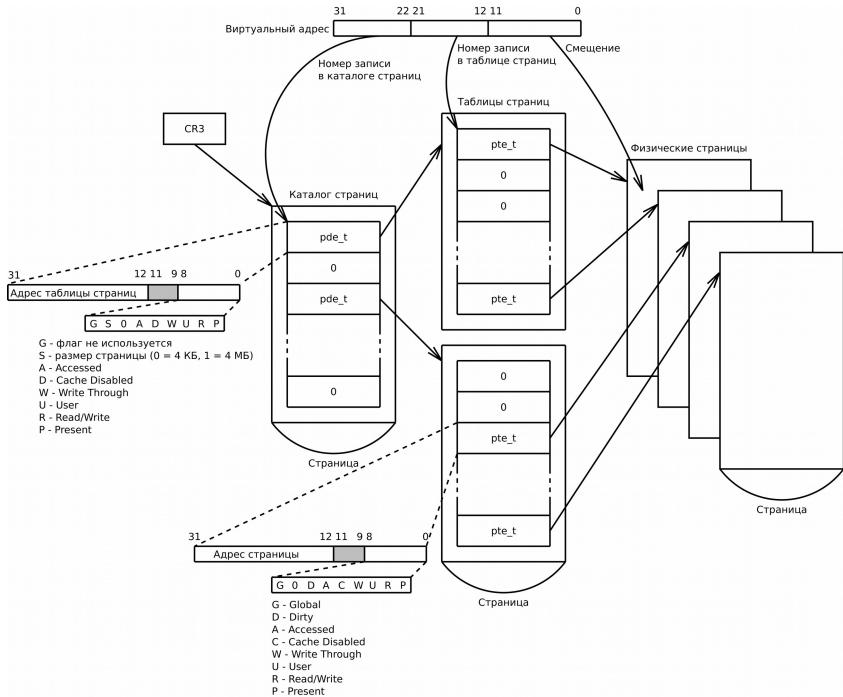


Рисунок 2.3.5: Страницчная трансляция

- R — Read/Write, чтение/запись. Определяет тип доступа к странице: если установлен — страница доступна на запись, если нет — только на чтение.
- P — Present, присутствие. Означает, что страница отображена и может использоваться при трансляции.

В kern/entrypgdir.c задан простой каталог страниц, позволяющий ядру работать по своему адресу связывания 0xF0100000, хотя на самом деле оно загружается в физическую память сразу после ROM BIOS по адресу 0x00100000. Этот каталог отображает только 4 МБ памяти. В данной работе это пространство будет расширено до первых 256 МБ памяти, начиная с виртуального адреса 0xF0000000.

► В то время как GDB может обращаться к памяти QEMU только по виртуальным адресам, часто бывает полезно иметь возможность контролировать физическую память при настройке виртуальной памяти. Изучите команды монитора QEMU (разд. 3.4.2), особенно команду `xr`, которая позволяет просматривать физическую память. Используйте команду `xr` монитора QEMU и команду `x` GDB для проверки памяти по соответствующим физическим и виртуальным адресам и убедитесь, что вы видите одни и те же данные.

Для дальнейшей работы также могут быть полезны команды `info mem` (показывает обзор отображенных областей виртуальной памяти и их разрешения) и `info pg` (показывает компактное, но детальное представление текущих таблиц страниц, включая все отображенные диапазоны памяти, разрешения и флаги).

При нахождении в защищенном режиме (переход в который происходит в файле `boot/boot.S`) нет никакого способа непосредственно использовать линейный или физический адрес. Все ссылки на память интерпретируются как виртуальные адреса и транслируются MMU (Memory Management Unit – блок управления памятью процессоров x86). Это означает, что все указатели в С содержат виртуальные адреса.

Периодически возникает необходимость манипулировать физическими и виртуальными адресами как значениями или как целыми числами, без разыменования, например, в аллокаторе физической памяти. Для этого в JOS используется два типа: `uintptr_t` представляет собой виртуальный адрес, а `physaddr_t` – физический. Оба типа являются синонимами для 32-разрядных целых чисел (`uint32_t`), поэтому компилятор не запретит присвоение значения одного типа переменной другого. Так как эти типы не являются указателями, их невозможно напрямую разыменовать. Для разыменования

`uintptr_t` можно привести его к указателю, однако разыменовать физический адрес разумным образом нельзя, так как MMU преобразовывает все обращения к памяти. Если привести `physaddr_t` к указателю, можно обращаться по этому виртуальному адресу, но он, скорее всего, не будет соответствовать необходимому физическому.

Иногда необходимо обращаться к памяти, для которой известен только физический адрес. Например, добавление отображения в таблицу страниц может потребовать выделения физической памяти для хранения каталога страниц, а затем инициализации этой памяти. Тем не менее, ядро, как и любое другое программное обеспечение, не может обойти трансляцию адресов и, следовательно, не может напрямую использовать физические адреса. Одна из причин, по которым JOS распределяет всю физическую память, начиная с физического адреса 0, по виртуальному адресу 0xF0000000 — необходимость помочь ядру использовать память, для которой оно знает только физический адрес. Для того, чтобы перевести физический адреса в виртуальный, которые ядро может использовать, нужно добавить к физическому адресу 0xF0000000. Для этого следует использовать макрос `KADDR(pa)`.

Иногда нужно, наоборот, найти физический адрес по виртуальному адресу, где хранятся данные ядра. Глобальные переменные ядра и память, выделенная `boot_alloc()`, находятся в области, где ядро было загружено, начиная с 0xF0000000 — в той самой области, куда мы отобразили всю физическую память. Таким образом, чтобы превратить виртуальный адрес в этой области в физический, нужно просто вычесть 0xF0000000. Для этого следует использовать макрос `PADDR(va)`.

### 2.3.4 Управление каталогами и таблицами страниц

Далее следует написать набор функций для управления таблицами страниц: выделения страниц, добавления и удаления отображений линейных адресов на физические.

► В файле *kern/pmap.c* необходимо реализовать код следующих функций:

- *pgdir\_walk()*

По указателю на каталог страниц (*pgdir*) возвращает запись в таблице страниц для линейного адреса *va*. Соответствующая запись может еще не существовать, в таком случае, если *create == false*, возвращает *NULL*, иначе выделяет новую таблицу страниц с помощью *page\_alloc*. Увеличивает счетчик ссылок на страницу. Флаги должны быть установлены в *perm|PTE\_P*.

- *boot\_map\_region()*

Отображает виртуальные адреса  $[va, va+size)$  по физическим  $[ra, ra+size)$  в каталоге страниц *pgdir*. *size* кратен *PGSIZE*, *va* и *ra* также кратны *PGSIZE*. Флаги должны быть установлены в *perm|PTE\_P*.

- *page\_lookup()*

Возвращает страницу, отображенную по адресу *va*.

- *page\_remove()*

Удаляет отображение физической страницы по адресу *va*. Если страница по этому адресу отсутствует, ничего не делает. Счетчик ссылок должен быть уменьшен, а если он достиг 0, страница должна быть освобождена.

- *page\_insert()*

Отображает физическую страницу *pp* по адресу *va*. Флаги должны быть установлены в *perm | PTE\_P*. Если страница по этому адресу уже отображена, она

*должна быть удалена с помощью page\_remove(). Если необходимо, таблица страниц должна быть выделена и добавлена в pgdir. Счетчик ссылок должен быть увеличен.*

Функция check\_page(), вызываемая из mem\_init(), проверяет ваши функции управления таблицами страниц. Вы должны убедиться, что она сообщает об успехе, прежде чем продолжить.

### 2.3.5 Адресное пространство ядра

JOS делит 32-битное линейное адресное пространство процессора на две части. Пользовательские процессы будут контролировать схему и содержимое нижней части, а ядро всегда сохраняет полный контроль над верхней частью. Разделительная линия определяется условно символом ULIM в файле inc/memlayout.h, резервируя около 256 МБ виртуального адресного пространства для ядра. Именно для этого нужен такой высокий адрес связывания ядра: в противном случае было бы недостаточно места в виртуальном адресном пространстве для отображения пользовательских процессов ниже ядра.

Поскольку и память ядра, и память пользователя присутствуют в адресном пространстве каждого процесса, нужно использовать биты разрешений в таблицах страниц, чтобы разрешить пользователю доступ только к пользовательской части адресного пространства. В противном случае ошибки в пользовательском коде могут изменить данные ядра, что может привести к сбоям или ошибкам; пользовательский код может также украсть данные других процессов.

Пользовательские процессы не будут иметь разрешений для любой памяти выше ULIM, в то время как ядро сможет читать и писать в эту память. Для диапазона адресов [UTOP, ULIM) ядро, и пользовательские процессы имеют одинако-

вые разрешения: они могут читать, но не писать в этот диапазон адресов. Этот диапазон адресов используется для представления определенных структур данных ядра пользовательским процессам только для чтения. Наконец, адресное пространство ниже UTOP доступно для использования пользовательским процессам: процесс может сам задавать разрешения для доступа к этой памяти.

Теперь следует правильным образом отобразить адресное пространство выше UTOP — часть адресного пространства, принадлежащую ядру. В файле inc/memlayout.h показана схема, которую следует использовать. Для создания соответствующих отображений нужно использовать написанные в предыдущем задании функции.

► Допишите недостающий код в *mem\_init()* после вызова *check\_page()*.

Теперь код должен проходить проверки *check\_kern\_pgdir()* и *check\_page\_installed\_pgdir()*.

После выполнения этого задания рекомендуется проверить разработанный код с помощью *make grade*. Скрипт должен выводить OK для всех тестов.

► Какие записи (строки) в каталоге страниц заполнены в данный момент? Какие адреса они отображают и куда указывают? Иными словами, максимально заполните табл. 2.3.1.

Запись	Базовый вирт. адрес	Содержимое
1023	?	Таблица страниц для верхних 4 МБ физической памяти
1022	?	
...		
2	0x00800000	?
1	0x00400000	?
0	0x00000000	?

Таблица 2.3.1: Данные в каталоге страниц

Мы поместили пользовательские процессы и ядро в одном адресном пространстве. Почему пользовательские программы не смогут читать или писать в память ядра? Какие конкретные механизмы защищают память ядра? Каков максимальный объем физической памяти, который данная операционная система может поддерживать? Почему?

Сколько дополнительных затрат памяти потребовалось бы для управления памятью, если мы на самом деле имели максимальный объем физической памяти? На какие части эти накладные расходы разбиты?

Изучите еще раз механизм настройки таблицы страниц в *kern/entry.S* и в *kern/entrypgdir.c*. Сразу после включения страничной организации памяти EIP по-прежнему имеет низкое значение (чуть более 1 МБ). В какой точке мы переходим к использованию EIP выше KERNBASE? Что делает возможным продолжение выполнения с низким EIP между моментом, когда мы включаем страничную организацию, и моментом, когда мы начинаем использовать EIP выше KERNBASE? Почему этот переход необходим?

*Запишите ответы на эти вопросы в файл lab7-answers.txt и поместите этот файл в репозиторий.*

*По окончании выполнения работы следует сохранить внесенные изменения и отправить в удаленный репозиторий.*

### **2.3.6 Процессы в пользовательском режиме**

Третья часть раздела соответствует лабораторной работе №8, которая продолжается в следующем разделе. Для получения файлов и изменений, необходимых для выполнения работы, следует обновить удаленный репозиторий, создать новую ветвь под названием working-lab8, после чего выполнить слияние с ней ветви lab8, которая появилась в репозитории. О работе с git можно прочитать в разд. 3.2.

Структура Env в inc/env.h изменилась, в ней появилось поле env\_pgdир — виртуальный адрес каталога страниц для этого процесса. Адресное пространство процесса определяется по каталогу страниц и таблицам страниц, на которые указывает env\_pgdир. Чтобы запустить процесс, ядро должно установить в процессоре и сохраненные регистры, и соответствующее адресное пространство.

### **2.3.7 Выделение массива процессов**

В лабораторной работе №6 в mem\_init() выделялась память для массива pages[], позволяющего отслеживать занятые и свободные страницы памяти. Аналогичный массив envs[] из структур Env был задан статически. Нужно изменить mem\_init(), чтобы он выделялся динамически аналогично pages[].

► Измените mem\_init() в kern/pmap.c, чтобы она выделяла массив envs. Этот массив состоит ровно из NENV экземпляров структуры Env и выделяется так же, как массив pages. Кроме того, как и для массива страниц, память, в которой содержится envs, должна быть отоб-

ражена пользователю только для чтения в UENVS (*inc/memlayout.h*), чтобы пользовательские процессы могли читать из этого массива.

Вы должны запустить код и убедиться, что *check\_kern\_pgdir()* завершается успешно.

### 2.3.8 Создание и запуск процесса

Теперь в *kern/env.c* нужно написать код, необходимый для запуска пользовательского процесса. GNUmakefile данной работы создает ряд двоичных файлов в *obj/user/*, аналогично работе 3; исходный код этих файлов берется из *user/*. В функции *i386\_init()* в *kern/init.c* находится код для запуска одного из этих файлов в качестве процесса. Основные функции настройки пользовательских процессов были написаны в работе 3, однако необходимо изменить их таким образом, чтобы процессы использовали виртуальную память.

► В файле *env.c* допишите следующие функции:

- *env\_setup\_vmt()*

Выделяет каталог страниц для нового процесса и инициализирует часть адресного пространства нового процесса, относящуюся к ядру.

- *region\_alloc()*

Выделяет и отображает физическую память для процесса.

- *load\_icode()*

Основная часть этой функции уже написана, необходимо изменить ее таким образом, чтобы она использовала виртуальную память, и добавить отображение страницы для стека процесса.

- *env\_create()*

Выделяет процесс с помощью *env\_alloc* и загружает в него двоичный ELF-образ путем вызова *load\_icode*.

- *env\_run()*  
Запускает данный процесс в пользовательском режиме.

Ниже приведен граф вызовов кода до точки, в которой запускается пользовательский код. Убедитесь, что вы понимаете цели каждого шага. В данный момент планировщик отключен, запускается сразу *env\_run()*.

- start (kern/entry.S)
- i386\_init (kern/init.c)
  - cons\_init
  - mem\_init
  - env\_init
  - trap\_init (в данный момент не завершена)
  - env\_create
  - env\_run
    - env\_pop\_tf

Теперь нужно скомпилировать ядро и запустить его в QEMU. Если задания выполнены правильно, система должна перейти в пользовательский режим и запустить приложение hello, которое должно проработать до системного вызова с инструкцией int. В этот момент возникнет «тройная ошибка», поскольку оборудование еще не настроено для перехода из пространства пользователя в ядро.

В дальнейших упражнениях эта проблема будет решена, но сейчас можно использовать отладчик, чтобы убедиться, что система переходит в пользовательский режим. Запустите make qemu-gdb и установите точку прерывания в *env\_pop\_tf*, которая должна быть последней функции, исполняемой до входа в пользовательский режим. Пройдите эту функцию с помощью команды si; процессор должен перейти в пользовательский режим после инструкции iret. После этого вы должны увидеть первую инструкцию в исполняемом пользовательском процессе — инструкцию cmpl на метке *start* в lib/entry.S. Теперь ис-

пользуйте команду `b * 0x...` для установки точки прерывания на `int $0x30` в `sys_cputs()` в `hello` (см. адрес в `obj/user/hello.asm`). Этот `int` – системный вызов для отображения символов в консоли. Если в процессе выполнения проблемы возникают раньше инструкции `int`, настройка адресного пространства или код загрузки программы ошибочны.

## 2.4 Прерывания

Для взаимодействия операционной системы с процессами и с различными устройствами они должны иметь возможность в нужный момент вызвать ее. Для этого используется механизм прерываний. При возникновении прерывания выполнение текущей последовательности команд приостанавливается, и управление передается обработчику прерывания, заданному ядром. Этот обработчик нужным образом реагирует на возникшее событие, после чего возвращает управление в прерванный код.

Прерывания делятся на:

1. Асинхронные, или внешние – события, которые исходят от внешних источников, например, сигнал от часов, клавиатуры или дискового накопителя.
2. Синхронные, или внутренние (исключения) – события в самом процессоре, например, деление на ноль, обращение по недопустимому адресу, недопустимый код операции. Частным случаем исключений являются программные прерывания, инициируемые выполнением специальной инструкции в коде программы – например, системный вызов или точка останова.

Более подробную информацию о механизме прерываний можно получить в главе 6 «Interrupt and exception handling» руководства Intel [8], а на концептуальном уровне – в книге Э. Таненбаума «Архитектура компьютера» [11].

Данный раздел содержит продолжение лабораторной работы №8 и лабораторную работу №9. В ходе дальнейшей работы вы реализуете основные возможности ядра, связанные с обработкой прерываний и исключений, а также механизм системных вызовов.

## **2.4.1 Обработка прерываний и исключений**

В данный момент первый вызов инструкции `int $0x30` в пользовательском пространстве является тупиком: как только процессор переходит в пользовательский режим, нет способа вернуться обратно. Необходимо реализовать простой механизм обработки исключений и системных вызовов, чтобы дать возможность ядру восстановить контроль над процессором из пользовательского режима.

Обычно мы следуем терминологии Intel касательно прерываний, исключений и т.п. Тем не менее, такие термины, как исключение, ловушка, прерывание, отказ не имеют стандартного значения для разных архитектур и операционных систем, и часто используются без учета тонких различий между ними для конкретной архитектуры, такой как x86. Когда вы видите эти термины вне данной работы, их значения могут несколько отличаться.

## **2.4.2 Защищенная передача управления**

Исключения и прерывания являются «защищенными передачами управления», которые заставляют процессор перейти из пользовательского режима в режим ядра ( $CPL = 0$ ), не давая пользовательскому коду возможности вмешаться в функционирование ядра или других процессов. В терминологии Intel прерывание является защищенной передачей управления, вызываемой асинхронными событиями, как правило, внешними по отношению к процессору, например, уведомлениями от внешних устройств ввода/вывода. Исключение, напротив, является защищенной передачей управления, вызываемой синхронно выполняемым в данный момент кодом, например, из-за деления на ноль или неправильного доступа к памяти.

Чтобы гарантировать, что эти защищенные передачи управления на самом деле являются защищенными, механизм

прерываний/исключений процессора разработан таким образом, что код, работающий в настоящее время, не имеет возможности выбирать, где и как передавать управление ядру. Вместо этого процессор гарантирует, что управление ядру может быть передано только в контролируемых условиях. В x86 эту защиту обеспечивает совместная работа двух механизмов:

- **Таблица дескрипторов прерываний.** Процессор гарантирует, что прерывания и исключения могут застать ядро начать выполняться только в нескольких конкретных местах, четко определенных самим ядром, а не кодом, выполняемым при возникновении прерывания или исключения.

Архитектура x86 позволяет использовать до 256 различных точек входа из прерывания или исключения в ядро, каждая со своим вектором прерывания. Вектор является числом от 0 до 255. Вектор прерывания определяется источником прерывания: различные устройства, ошибки и запросы приложений к ядру генерируют прерывания с разными векторами. Процессор использует вектор как индекс в таблице дескрипторов прерываний (IDT), которую ядро создает в своей открытой памяти, как и GDT. Из соответствующей записи в этой таблице процессор получает:

- значение для загрузки в регистр EIP, указывающее на код ядра, предназначенный для обработки этого типа исключений;
- значение для загрузки в регистр CS, которое включает в себя в битах 0-1 уровень привилегий, с которым будет запускаться обработчик исключения (в JOS все исключения обрабатываются в режиме ядра с уровнем привилегий 0).

- **Сегмент состояния задачи.** Процессору необходимо место для сохранения состояния перед прерыванием или исключением, в том числе значений EIP и CS до вызова обработчика исключения, чтобы в дальнейшем восстановить это состояние и возобновить прерванное выполнение с того места, где оно было прервано. Но эта область сохранения состояния процессора, в свою очередь, должна быть защищена от непривилегированного кода пользовательского режима, в противном случае ошибка или вредоносный пользовательский код может поставить под угрозу ядро.

Поэтому, принимая прерывание или ловушку, которые вызывают изменение уровня привилегий от пользовательского режима до режима ядра, процессор также переключается на стек в памяти ядра. Структура под названием Сегмент состояния задачи (Task State Segment, TSS) указывает селектор сегмента и адрес, по которому находится стек. Процессор добавляет в этот новый стек регистры SS, ESP, EFLAGS, CS, EIP и опциональный код ошибки. Затем он загружает CS и EIP из дескриптора прерывания и устанавливает значения ESP и SS для нового стека.

Хотя TSS имеет большой размер и потенциально может использоваться в различных целях, JOS использует его только для определения стека ядра, на который процессор должен переключаться, когда переходит из пользовательского режима в режим ядра. Поскольку «режим ядра» в JOS соответствует уровню привилегий 0 в x86, процессор использует поля TSS ESP0 и SS0 для определения стека ядра при входе в режим ядра. JOS не использует никаких других полей TSS.

### **2.4.3 Типы исключений и прерываний**

Все синхронные исключения, которые может генерировать процессор x86, используют векторы прерываний от 0 до 31, и, следовательно, соответствуют записям IDT 0-31. Например, ошибка страницы всегда вызывает исключение по вектору 14. Вектора прерываний выше 31 используются только программными прерываниями, которые могут генерироваться инструкцией `int`, или асинхронными аппаратными прерываниями, вызываемыми внешними устройствами, когда они требуют внимания.

Далее будет реализована обработка исключений в векторах 0-31. В следующем разделе будет реализована обработка программного вектора прерывания 48 (0x30), который JOS использует в качестве вектора прерывания для системных вызовов. Затем в следующей лабораторной работе будет реализована обработка внешних аппаратных прерываний, в т.ч. прерываний от часов реального времени.

### **2.4.4 Пример**

Объединим вышеописанные части вместе и проследим их работу на примере. Пусть процессор выполняет код пользовательского процесса и встречает инструкцию деления, которая пытается делить на ноль.

- Процессор переключается на стек, определенный полями SS0 и ES0 в TSS, которые в JOS будут содержать соответственно значения GD\_KD и KSTACKTOP.
- Процессор добавляет параметры исключения в стек ядра, начиная с адреса KSTACKTOP, в соответствии с рис. 2.4.1.
- Так как обрабатывается ошибка деления, которая соответствует вектору прерывания 0 в x86, процессор чита-

	KSTACKTOP
0x0000   Старый SS	KSTACKTOP - 4
Старый ESP	KSTACKTOP - 8
Старый EFLAGS	KSTACKTOP - 12
0x0000   Старый CS	KSTACKTOP - 16
Старый EIP	KSTACKTOP - 20 <----- ESP

Рисунок 2.4.1: Параметры исключения

ет запись IDT с номером 0 и устанавливает CS:EIP на функцию-обработчик, описываемую этой записью.

- Управление передается функции-обработчику, которая обрабатывает исключение, например, путем завершения пользовательского процесса.

Для некоторых типов исключений процессор записывает в стек еще одно слово в дополнение к обычным пяти машинным словам выше, содержащее код ошибки. Примером является исключение ошибки страницы (номер 14). Просмотрите руководство [8], чтобы определить, для каких номеров исключений процессор записывает код ошибки, и что код ошибки в этих случаях означает. Когда процессор записывает в стек код ошибки, стек будет выглядеть в соответствии с рис. 2.4.2.



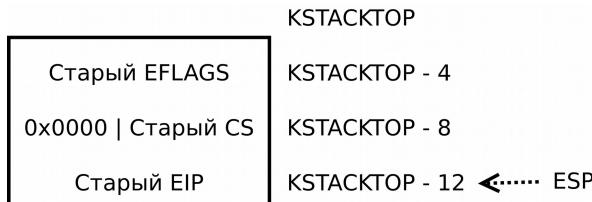
*Рисунок 2.4.2: Параметры исключения и код ошибки*

## 2.4.5 Вложенные исключения и прерывания

Процессор может принимать исключения и прерывания, находясь как в режиме ядра, так и в пользовательском режиме. Однако только при входе в режим ядра из пользовательского режима процессор x86 автоматически переключает стек перед сохранением состояния регистров и вызовом соответствующего обработчика исключения через IDT. Если процессор уже находится в режиме ядра, когда происходит прерывание или исключение (нижние 2 бита регистра CS уже равны нулю), процессор только записывает дополнительные значения в том же стеке ядра. В этом случае ядро может корректно обработать вложенные исключения, вызванные кодом внутри самого ядра. Эта возможность является важным инструментом в реализации защиты, как мы увидим далее в разделе о системных вызовах (разд. 2.5).

Если процессор уже находится в режиме ядра и принимает вложенное исключение, он не сохраняет значения SS и ESP, так как не требует переключения стеков. Для типов исключений, которые не используют код ошибки, стек ядра, сле-

довательно, при входе в обработчик исключения выглядит в соответствии с рис. 2.4.3.



*Рисунок 2.4.3: Параметры исключения в режиме ядра*

Для типов исключений, которые используют код ошибки, процессор помещает код ошибки сразу же после EIP, как и ранее.

Существует важная проблема в обработке вложенных исключений. Если процессор принимает исключение, уже находясь в режиме ядра, и не может сохранить свое состояние в стек ядра по той или иной причине, например, из-за отсутствия места в стеке, то процессор не может ничего сделать и просто перезагружается. Излишне говорить, что ядро должно быть сконструировано таким образом, чтобы этого не могло произойти.

## **2.4.6 Создание таблицы дескрипторов прерываний**

Теперь вы можете создать IDT и обрабатывать исключения в JOS. На данный момент вы создадите IDT для обработки векторов прерываний 0-31 (исключения процессора). Прерывания системных вызовов будут обрабатываться далее в этой работе, а прерывания 32-47 (прерывания устройств) — в следующей.

Заголовочные файлы inc/trap.h и kern/trap.h содержат важные определения, связанные с прерываниями и исключениями. С этими определениями вам нужно ознакомиться.

Файл kern/trap.h относится только к ядру, а inc/trap.h содержит определения, которые также могут быть полезны для пользовательских программ и библиотек.

Общий поток управления, которого нужно достичь, изображен на рис. 2.4.4.

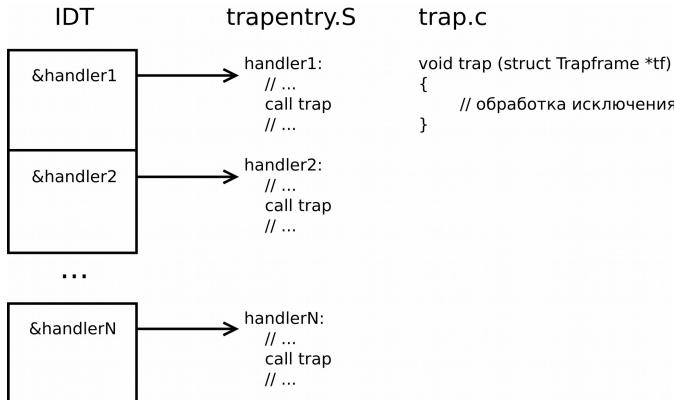


Рисунок 2.4.4: Обработка исключения

Каждое исключение или прерывание должно иметь свой собственный обработчик в trapentry.S, а trap\_init() должна инициализировать IDT адресами этих обработчиков. Каждый из обработчиков должен создавать структуру Trapframe (см. inc/trap.h) в стеке и вызывать trap() (в trap.c) с указателем на Trapframe. Функция trap() затем обрабатывает исключение/прерывание или направляет к конкретной функции-обработчику.

Некоторые исключения в диапазоне 0-31 определены Intel как зарезервированные. Так как они никогда не будут генерироваться процессором, неважно, как их обрабатывать.

► Отредактируйте trapentry.S и trap.c и реализуйте вышеописанные функции. Для этого полезны макросы TRAPHANDLER и TRAPHANDLER\_NOEC в trapentry.S и константы T\_\* в inc/trap.h. Необходимо добавить точку

входа в *trapentry.S* с помощью этих макросов для каждого прерывания или исключения, определенного в *inc/trap.h*, а также функцию *\_alltraps*, на которую ссылаются макросы *TRAPHANDLER*. Также нужно изменить *trap\_init()*, инициализируя *IDT* таким образом, чтобы он указывал на каждую из этих точек входа, определенных в *trapentry.S*; для этого полезен макрос *SETGATE*.

Функция *\_alltraps* должна:

- Записать значения в стек таким образом, чтобы стек был похож на структуру *Trapframe*;
- Записать *GD\_KD* в *%ds* и *%es*;
- С помощью *pushl %esp* передать указатель на *Trapframe* в качестве аргумента функции *trap()*;
- Вызвать *trap* (может ли *trap* вернуться?).

Вам может пригодиться инструкция *pushal*, которая сохраняет регистры процессора в стек в порядке, соответствующем *struct PushRegs*.

Код обработки прерываний можно проверить с помощью тех тестовых программ в каталоге *user*, которые вызывают исключения — например, *user/divzero*. Скрипт *make grade* должен срабатывать на вызовах *divzero*, *softint* и *badsegment*.

## 2.4.7 Вопросы

► Ответьте на следующие вопросы в файле *lab8-answers.txt*:

- Какова цель наличия отдельных функций-обработчиков для каждого исключения/прерывания? Если бы для всех исключений/прерываний вызывался один и тот же обработчик, какая функция, присутствующая в текущей реализации, не могла бы быть реализована?

- Нужно ли было что-либо делать, чтобы программа `user/softint` работала правильно? Скрипт проверки ожидает, что она создаст общую ошибку защиты (ловушка 13), но в коде `softint` находится инструкция `int $14`. Почему она фактически создает исключение с вектором 13? Что произойдет, если ядро позволит инструкции `int $14` в `softint` запустить обработчик ошибок ядра (с вектором прерывания 14)?

#### 2.4.8 Обработка ошибок страниц

Теперь, когда ядро имеет базовые возможности обработки исключений, можно использовать их для обработки таких исключений, как ошибки страниц (Page Fault), системные вызовы (System Call) и точки останова (Breakpoint).

Исключение ошибки страницы с вектором прерывания 14 (T\_PGFLT) используется в этой и в следующей работе. Когда процессор принимает ошибку страницы, он сохраняет линейный (т.е. виртуальный) адрес, который вызвал ошибку, в специальном регистре CR2. В `trap.c` есть заготовка специальной функции `page_fault_handler()` для обработки исключений ошибок страниц.

► Измените `trap_dispatch()` для отправки исключения ошибки страницы в `page_fault_handler()`. Теперь `make grade` должен срабатывать на `faultread`, `faultreadkernel`, `faultwrite`, и `faultwritekernel`. Если какой-либо из тестов не выполняется, выясните, почему, и исправьте проблему. Помните, что вы можете загрузить JOS в конкретную пользовательскую программу с помощью `make run-X` или `make run-X-nox`.

## 2.5 Системные вызовы

Пользовательские процессы могут отправлять ОС запросы на определенные действия путем системных вызовов. Когда пользовательский процесс выполняет системный вызов, процессор переходит в режим ядра, затем процессор и ядро сохраняют состояние пользовательского процесса, ядро выполняет соответствующий код системного вызова, а затем возобновляет выполнение процесса. Точный способ передачи пользовательским процессом информации о вызове в ядро и определения конкретного системного вызова в разных операционных системах различаются.

В ядре JOS в качестве прерывания системного вызова используется int \$0x30. Этому значению соответствует константа T\_SYSCALL. Необходимо создать дескриптор прерывания, чтобы позволить пользовательским процессам вызывать его. Обратите внимание на то, что прерывание 0x30 не может быть сгенерировано аппаратно, поэтому возможность для пользовательского кода генерировать его не создает никакой двусмысленности.

Программа передает номер системного вызова и его аргументы в регистрах, чтобы не обращаться к стеку пользовательского процесса или потоку инструкций. Номер системного вызова находится в %eax, а аргументы (количеством до пяти) находятся в %edx, %ecx, %ebx, %edi и %esi соответственно. Возвращаемое значение передается в %eax. Ассемблерный код для системного вызова находится в функции syscall() в lib/syscall.c. Вы должны прочесть его и убедиться, что понимаете, что происходит.

- Добавьте обработчик для вектора прерывания T\_SYSCALL. Для этого нужно внести изменения в kern/trapentry.S и в функцию trap\_init() (kern/trap.c).

*Кроме того, необходимо изменить `trap_dispatch()` для обработки прерывания системного вызова вызовом функции `syscall()` (определенной в `kern/syscall.c`) с соответствующими аргументами, а затем вернуть возвращаемое значение в пользовательский процесс через регистр `%eax`. Наконец, необходимо реализовать функцию `syscall()` в `kern/syscall.c`. Эта функция должна возвращать `-E_INVAL`, если системный вызов с переданным номером не существует. Вы должны прочитать и понять `lib/syscall.c` (особенно ассемблерную часть), чтобы подтвердить ваше понимание механизма системных вызовов. Также может быть полезным чтение `inc/syscall.h`.*

*Запустите программу `user/hello` в вашем ядре (`make run-hello`). Она должна вывести в консоль «`hello, world`», а затем вызвать ошибку страницы в пользовательском режиме. Если этого не происходит, вероятно, присутствует ошибка в обработчике системного вызова. Кроме того, теперь должен срабатывать тест `testbss` в `make grade`.*

### 2.5.1 Точка останова

Иключение точки останова с номером 3 (T\_BRKPT) обычно используется, чтобы дать возможность отладчикам устанавливать точки останова в коде программы, временно заменяя соответствующую инструкцию специальной однобайтной инструкцией `int3`, вызывающей программную точку останова. В JOS мы используем это исключение не совсем правильным образом, превратив его в примитивный псевдосистемный вызов, который позволит любому пользовательскому процессу вызвать монитор JOS. Это использование в определенной степени уместно, если думать о мониторе ядра JOS как о примитивном отладчике. Например, реализация `panic()` в `lib/panic.c` выполняет `int3` после показа сообщения.

► Измените `trap_dispatch()`, чтобы точки останова вызывали монитор. Теперь `make grade` должен проходить проверку `breakpoint`.

## 2.5.2 Вопросы

► Ответьте на следующие вопросы в файле `lab8-answers.txt`:

- Тест `breakpoint` генерирует либо исключение точки останова, либо общую ошибку защиты в зависимости от того, как вы инициализировали точку входа в точку останова в IDT (то есть в зависимости от вашего вызова `SETGATE` из `trap_init`). Почему? Как настроить его таким образом, чтобы заставить точку останова работать правильным образом, и какая неправильная настройка вызовет общую ошибку защиты?
- В чем, по-вашему, смысл этих механизмов, особенно в свете того, что делает программа `user/softint`?

## 2.5.3 Запуск пользовательского режима

Пользовательская программа начинает работать в начале `lib/entry.S`. После некоторых предварительных действий этот код вызывает `libmain()`, в `lib/libmain.c`. Вы должны изменить `libmain()` для инициализации глобального указателя `thisenv`, чтобы он указывал на соответствующую текущему процессу структуру `Env` в массиве `envs[]`. (Заметим, что в `lib/entry.S` уже определены `envs`, чтобы указывать на отображение `UENVS`, созданное ранее.) Для этого полезно просмотреть `inc/env.h` и использовать `sys_getenvid`.

Функция `libmain()` вызывает `umain`, которая, в случае программы `hello`, находится в `user/hello.c`. Обратите внимание,

что после вывода «Hello, World» она пытается получить доступ к `thisenv->env_id`. Именно поэтому ранее она вызывала ошибку. Теперь, когда вы инициализировали `thisenv` должным образом, ошибки быть не должно. Если ошибка все еще есть, вы, вероятно, не отобразили область `UENVS` с возможностью чтения пользователем (ранее, в `rmar.c`; это первый раз, когда мы использовали область `UENVS`).

► *Добавьте необходимый код в пользовательскую библиотеку, а затем загрузите ядро. Вы должны увидеть, что `user/hello` печатает «Hello, World», а затем «i am environment 00001000». Программа `user/hello` затем пытается «выйти» путем вызова `sys_env_destroy()` (см. `lib/libmain.c` и `lib/exit.c`). Так как ядро в настоящее время поддерживает только один пользовательский процесс, оно должно сообщить, что уничтожило единственный процесс, а затем запустить монитор. Теперь `make grade` должен срабатывать на тесте `hello`.*

## 2.5.4 Ошибки страниц и защита памяти

Задача памяти является важнейшей особенностью операционной системы. Она гарантирует, что ошибки в одной программе не могут повредить другие программы или саму операционную систему.

Операционные системы обычно используют аппаратную поддержку для реализации защиты памяти. ОС сообщает аппаратуре о том, какие виртуальные адреса являются верными, а какие нет. Когда программа пытается получить доступ к неверному адресу или к такому, для которого она не имеет разрешения, процессор останавливает программу на инструкции, вызвавшей данную проблему, а затем переходит в ядро с информацией о неудавшейся операции. Если ошибку можно исправить, ядро исправляет ее и программа продолжает работу. Если ошибку исправить нельзя, программа не

может продолжаться, так как она никогда не перейдет дальше инструкции, вызвавшей ошибку.

В качестве примера поправимой ошибки можно рассмотреть автоматически увеличивающийся стек. Во многих системах ядро первоначально выделяет одну страницу для стека, а затем, если программа вызывает неисправность при доступе к страницам далее вниз по стеку, ядро будет автоматически выделять эти страницы и позволять программе продолжаться. Таким образом, ядро выделяет столько памяти под стек, сколько на самом деле используется в каждый момент времени, но программа может работать в иллюзии, что стек имеет сколь угодно большой размер.

Системные вызовы представляют собой интересную задачу для защиты памяти. Большинство интерфейсов системных вызовов позволяет пользователю передавать указатели в ядро. Эти указатели указывают на пользовательские буферы, которые можно читать или записывать. Затем ядро разыменовывает эти указатели при выполнении системного вызова. Здесь есть две проблемы:

- Ошибка страницы в ядре — потенциально гораздо более серьезная вещь, чем ошибка страницы в пользовательской программе. Если ошибка страниц в ядре возникает при манипуляции собственными структурами данных ядра, это ошибка в ядре, и обработчик ошибок должен вызывать панику ядра (и, следовательно, всей системы). Но когда ядро разыменовывает указатели, предоставляемые ему пользовательской программой, оно должно найти способ запомнить, что любые ошибки страниц при разыменовании этих указателей на самом деле являются ошибками в пользовательской программе.
- Ядро обычно имеет больше прав доступа к памяти, чем пользовательские программы. Пользовательская

программа может передать системному вызову указатель на память, которую ядро может читать или писать, а программа не может. В ядре нужно соблюдать осторожность, чтобы не разыменовывать такой указатель, поскольку это может раскрыть закрытую информацию или нарушить целостность ядра.

По обеим причинам ядро должно быть чрезвычайно осторожным при обращении с указателями, предоставленными пользовательскими программами.

Обе проблемы решаются использованием одного и того же механизма, который изучает все указатели, передаваемые из пользовательского пространства в ядро. Когда программа передает ядру указатель, ядро должно убедиться, что адрес находится в пользовательской части адресного пространства, а таблица страниц позволяет такой доступ к памяти.

Таким образом, ядро никогда не вызовет ошибку страницы из-за разыменования пользовательского указателя. Если ядро все же вызывает ошибку страницы, оно должно паниковать и прекращать работу.

► Измените *kern/trap.c*, чтобы он вызывал панику, если ошибка страницы происходит в режиме ядра.

Подсказка: для определения, в режиме ядра или в пользовательском режиме произошла неисправность, можно проверить нижние биты *tf.cs*.

Прочитайте *user\_mem\_assert* в *kern/pmap.c* и реализуйте *user\_mem\_check* в том же файле.

Измените *kern/syscall.c* для проверки аргументов системных вызовов.

Загрузите ядро, выполните *user/buggyhello*. Процесс должен быть уничтожен, но ядро не должно вызывать *panic()*. Вы должны увидеть:

```
[00001000] user_mem_check assertion failure for va 00000001
```

```
[00001000] free env 00001000
```

```
Destroyed the only environment – nothing more to do!
```

Наконец, измените `debuginfo_eip` в `kern/kdebug.c` для вызова `user_mem_check` для `usd`, `stabs` и `stabstr`. Если вы сейчас запустите `user/breakpoint`, вы должны быть в состоянии запустить `backtrace` в мониторе и увидеть трассировку в `lib/libmain.c` перед паникой ядра с ошибкой страницы. Каковы причины этой ошибки страницы? Вам не нужно исправлять ее, но вы должны понимать, почему она происходит.

Обратите внимание, что тот же механизм, который вы только что реализовали, работает и для вредоносных пользовательских приложений (например, `user/evilhello`).

Загрузите ядро, выполнив `user/evilhello`. Процесс должен быть уничтожен, но ядро не должно вызывать `panic()`. Вы должны увидеть:

```
[00000000] new env 00001000
```

```
[00001000] user_mem_check assertion failure for va f0100020
```

```
[00001000] free env 00001000
```

По окончании выполнения работы следует сохранить внешние изменения и отправить в удаленный репозиторий.

## 2.5.5 Системные вызовы для создания процессов

Следующая часть раздела соответствует лабораторной работе №9. Для получения файлов и изменений, необходимых для выполнения работы, следует обновить удаленный репозиторий, создать новую ветвь под названием `working-lab9`, после чего выполнить слияние с ней ветви `lab9`, которая появилась в репозитории. О работе с `git` можно прочитать в разд. 3.2.

В настоящий момент ядро уже может запускать процессы и переключаться между ними, но по-прежнему ограничива-

ется лишь теми процессами, которые само запустило. В данной работе будут реализованы системные вызовы, необходимые для создания и запуска новых процессов уже запущенными.

В Unix для создания процессов используется системный вызов `fork()`. Вызов `fork()` копирует все адресное пространство вызывающего (родительского) процесса, создавая новый (дочерний) процесс. Единственное различие между двумя процессами, наблюдаемое из пользовательского пространства — идентификаторы самих процессов и родительских процессов (`getpid()` и `getppid()`). В родительском процессе `fork()` возвращает идентификатор созданного дочернего процесса, а в дочернем — 0. Обычно каждый процесс получает свое собственное адресное пространство, и модификации памяти, выполняемые другими процессами, ему не видны.

Далее необходимо разработать набор более примитивных, чем `fork()`, системных вызовов для создания новых пользовательских процессов в JOS. С помощью этих системных вызовов можно далее реализовать Unix-подобный `fork()` и другие способы создания процессов полностью в пользовательском пространстве. Вы должны написать следующие системные вызовы для JOS:

- `envid_t sys_exofork(void):`

Создает новый почти чистый процесс: ничего не отображено в пользовательской части его адресного пространства, и он не является `Runnable`. Новый процесс должен иметь то же состояние регистров, что и родительский в момент вызова `sys_exofork`. В родительском процессе `sys_exofork` должен возвращать `envid_t` вновь созданного процесса (или отрицательный код ошибки, если создать процесс не удалось). В дочернем процессе он должен вернуть 0. (Поскольку дочерний процесс изначально не-`Runnable`, он не выйдет из `sys_exofork`,

пока родитель не сделает его runnable с помощью следующего вызова).

- `int sys_env_set_status(envid_t envid, int status):`  
Устанавливает состояние указанного процесса в ENV\_RUNNABLE или ENV\_NOT\_RUNNABLE. Этот системный вызов, как правило, используется для обозначения того, что новый процесс готов к запуску, когда его адресное пространство и регистры полностью инициализированы.
- `int sys_page_alloc(envid_t envid, void *va, int perm):`  
Выделяет страницу физической памяти и отображает ее по данному виртуальному адресу в адресном пространстве данного процесса.
- `int sys_page_map(envid_t srcenvid, void *srcva, envid_t dstenvid, void *dstva, int perm):`  
Создает новое отображение для физической страницы, уже отображенной в адресном пространстве одного процесса, для другого процесса.
- `int sys_page_unmap(envid_t envid, void *va):`  
Удаляет отображение страницы по данному виртуальному адресу в данном процессе.

Для всех системных вызовов, которые принимают идентификаторы процессов, ядро поддерживает соглашение о том, что значение 0 означает «текущий процесс». Это соглашение осуществляется с помощью `envid2env()` в `kern/env.c`.

В тестовой программе `user/dumbfork.c` разработана примитивная реализация Unix-подобного `fork()`. Эта программа использует системные вызовы, указанные выше, для создания и запуска дочернего процесса с копией своего собственного адресного пространства. Затем два процесса переключаются между собой, используя `sys_yield`. Родительский процесс завершает работу после 10 итераций, а дочерний — после 20.

► Реализуйте системные вызовы, описанные выше, в *kern/syscall.c*. Для этого нужно использовать различные функции из *kern/pmap.c* и *kern/env.c*, в том числе *envid2env()*. При вызове *envid2env()* передавайте 1 в качестве параметра *checkperm*. Убедитесь, что вы проверяете аргументы системных вызовов на правильность, при необходимости возвращая *-E\_INVAL*. Кроме того, необходимо исправить функцию *env\_destroy*, чтобы она поддерживала работу нескольких пользовательских процессов одновременно, и *env\_alloc*, чтобы она включала для процесса прерывания в пользовательском режиме. Проверьте ядро с помощью *user/dumbfork* и убедитесь, что эта программа работает правильно, прежде чем продолжать.

## 2.5.6 Fork() с копированием при записи

Как было сказано ранее, в Unix системный вызов *fork()* используется в качестве основного способа создания процессов. Вызов *fork()* копирует адресное пространство вызывающего процесса (родительского), создавая новый процесс (дочерний).

Тем не менее, очень часто после вызова *fork()* почти сразу происходит вызов *exec()* в дочернем процессе, который заменяет его память новой программой. В этом случае время, потраченное на копирование адресного пространства родительского процесса, будет потрачено практически впустую, потому что дочерний процесс использует лишь незначительную часть своей памяти до вызова *exec()*.

Поэтому в современных версиях Unix с помощью аппаратной поддержки виртуальной памяти используется механизм, известный как копирование при записи (*copy-on-write*): родительский и дочерний процессы делят между собой память, отображаемую в их адресные пространства, до момента, когда один из процессов ее изменит. Для этого при вызове

`fork()` ядро копирует от родительского процесса к дочернему только отображение адресного пространства, но не содержимое отображенных страниц, и одновременно отмечает эти общие страницы как «только для чтения». Когда один из двух процессов попытается записать что-либо в одну из общих страниц, произойдет ошибка страницы. При возникновении такой ошибки ядро создаст новую, доступную для записи копию страницы для процесса, вызвавшего ошибку. Таким образом, содержимое отдельных страниц не будет фактически скопировано, пока в них не будет произведена запись. Эта оптимизация делает вызов `fork()` с последующим вызовом дочерним процессом `exec()` гораздо быстрее: скорее всего, дочерний процесс скопирует только одну страницу (текущую страницу своего стека) до вызова `exec()`.

В следующей части этой лабораторной работы вы реализуете «правильный» Unix-подобный `fork()` с копированием при записи в качестве библиотечной функции в пользовательском пространстве. Помимо того, что реализация `fork()` и копирования при записи в пользовательском пространстве делает ядро проще и, следовательно, надежнее, она также позволяет отдельным программам пользовательского режима определять свою собственную семантику для `fork()`; программа, которой необходимо реализовать другое поведение (например, всегда делать копию, как `dumbfork()`, или, наоборот, всегда делить память между родительским и дочерним процессом), может использовать собственную функцию.

### **2.5.7 Обработка ошибок страниц в пользовательском режиме**

Для вышеописанной реализации `fork()` вспомогательный механизм должен иметь возможность обрабатывать в пользовательском режиме ошибки при доступе к страницам, защищенным от записи.

Копирование при записи — только одно из многих возможных применений для такой возможности. Необходимость выполнять некоторое действие при возникновении ошибки страницы возникает во множестве различных случаев. Например, большинство ядер Unix-подобных операционных систем изначально отображает только одну страницу в области стека нового процесса; дополнительные страницы выделяются и отображаются по мере увеличения размера стека процесса, при возникновении ошибок страниц при доступе по адресам в стеке, которые еще не отображены. Обычно ядро должно предпринимать разные действия при возникновении ошибки страницы в разных областях адресного пространства процесса. Например, ошибка в области стека, как правило, требует выделения и отображения новой страницы физической памяти. При ошибке в области BSS, как правило, нужно также выделить и отобразить новую страницу, а также заполнить ее нулями. В некоторых системах исполняемые файлы выделяют память по необходимости: при ошибке в сегменте .text будут прочитаны с диска и отображены соответствующие страницы.

### 2.5.8 Установка обработчика ошибки страницы

Для того, чтобы обрабатывать ошибки страниц, пользовательскому процессу нужно зарегистрировать точку входа обработчика ошибок страниц в ядре. Он делает это с помощью нового системного вызова `sys_env_set_pgfault_upcall`. Для хранения этой информации к структуре Env было добавлено поле `env_pgfault_upcall`.

- Реализуйте `sys_env_set_pgfault_upcall`. Не забудьте проверку разрешений при поиске идентификаторов процессов, так как это «опасный» системный вызов.

## 2.5.9 Обычный стек и стек исключений в пользовательских процессах

Во время нормальной работы программы пользовательский процесс использует обычный стек: его регистр esp изначально указывает на USTACKTOP, а данные, добавляемые в стек, находятся между USTACKTOP-PGSIZE и USTACKTOP-1 включительно. Однако при возникновении ошибки страницы ядро запускает обработчик ошибок страниц с использованием стека пользовательского исключения. Таким образом, происходит автоматическое «переключение стека» аналогично тому, как процессор x86 переключает стек при переходе из пользовательского режима в режим ядра.

Стек пользовательского исключения JOS имеет размер в одну страницу, а его вершина определяется виртуальным адресом UXSTACKTOP, то есть данные стека находятся от UXSTACKTOP-PGSIZE до UXSTACKTOP-1 включительно. Во время работы обработчик ошибок страниц пользовательского режима может использовать обычные системные вызовы JOS для добавления новых отображений страниц или настройки отображений таким образом, чтобы исправить проблему, изначально вызвавшую ошибку страницы. Затем обработчик через обертку, написанную на языке ассемблера, переходит к коду, вызвавшему ошибку, который использует обычный стек.

Каждый пользовательский процесс, использующий обработку ошибок страниц в пользовательском режиме, должен выделить память для своего собственного стека исключений с помощью `sys_page_alloc()`.

## 2.5.10 Вызов пользовательского обработчика ошибок страниц

Теперь необходимо изменить код обработки ошибок страниц в kern/trap.c для обработки ошибок страниц из пользовательского режима, как описано ниже.

Если нет зарегистрированного обработчика ошибки страницы, ядро JOS уничтожает пользовательский процесс с соответствующим сообщением, как и ранее. В противном случае ядро добавляет кадр стека исключений, который выглядит как структура UTrapframe в inc/trap.h (рис. 2.5.1).

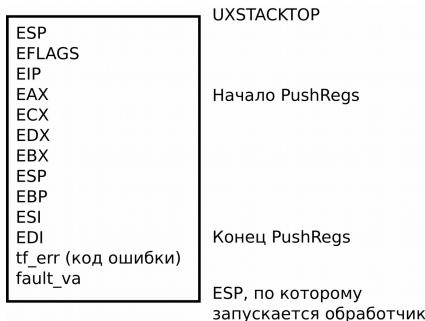


Рисунок 2.5.1: Кадр стека исключений

Затем ядро организует для пользовательского процесса продолжение работы с обработчиком ошибки страницы, работающим на стеке исключений с этого кадра стека. Вы должны выяснить, как правильно реализовать такое поведение. Значение `fault_va` – это виртуальный адрес, который вызвал ошибку страницы.

Если пользовательский процесс уже запущен на стеке исключения при возникновении исключения, значит, обработчик ошибки страницы сам вызвал ошибку. В этом случае вы должны начать новый кадр стека сразу после текущего `tf->`

`tf_esp`, а не с `UXSTACKTOP`. Вы должны сначала добавить в стек пустое 32-разрядное слово, а затем структуру `UTrapframe`.

Для проверки, находится ли `tf_esp` в пользовательском стеке исключений, проверьте, находится ли его значение в диапазоне между `UXSTACKTOP-PGSIZE` и `UXSTACKTOP-1` включительно.

► Реализуйте необходимый для отправки ошибок страниц обработчику пользовательского режима код в `page_fault_handler` в `kern/trap.c`. Примите соответствующие меры предосторожности при записи данных в стек исключений. (Что произойдет, если у пользовательского процесса закончится пространство в стеке исключений?)

### 2.5.11 Точка входа ошибки страницы пользовательского режима

Далее необходимо реализовать процедуру на языке ассемблера, которая вызывает обработчик ошибки страницы на С и возобновляет выполнение с инструкции, которая вызвала обработчик ошибки. Эта процедура — обработчик, который будет зарегистрирован ядром с помощью `sys_env_set_pgfault_upcall()`.

► Реализуйте `_pgfault_upcall` в `lib/pfentry.S`. Важным моментом является возвращение в исходную точку пользовательского кода, вызвавшую ошибку страницы: оно происходит без перехода в ядро. Сложность заключается в одновременном переключении стеков и значения `EIP`.

Наконец, реализуйте пользовательскую часть механизма обработки ошибок страниц в виде библиотеки на С — `set_pgfault_handler()` в `lib/pgfault.c`.

Запустите `user/faultread`. Вы должны увидеть:

...

[00000000] new env 00001000

```
[00001000] user fault va 00000000 ip 0080003a
```

TRAP frame ...

```
[00001000] free env 00001000
```

Запустите user/faultdie. Вы должны увидеть:

```
...
```

```
[00000000] new env 00001000
```

i faulted at va deadbeef, err 6

```
[00001000] exiting gracefully
```

```
[00001000] free env 00001000
```

Запустите user/faultalloc. Вы должны увидеть:

```
...
```

```
[00000000] new env 00001000
```

fault deadbeef

this string was faulted in at deadbeef

fault cafебffe

fault cafec000

this string was faulted in at cafебffe

```
[00001000] exiting gracefully
```

```
[00001000] free env 00001000
```

Если вы видите только первую строку «this string...», это означает, что вы не обрабатываете рекурсивные ошибки страниц должным образом.

Запустите user/faultallocbad. Вы должны увидеть:

```
...
```

```
[00000000] new env 00001000
```

```
[00001000] user_mem_check assertion failure for va deadbeef
```

```
[00001000] free env 00001000
```

Убедитесь, что вы понимаете, почему user/faultalloc и user/faultallocbad ведут себя по-разному.

## 2.5.12 Реализация fork() с копированием при записи

Теперь у вас есть поддержка со стороны ядра для реализации fork() с копированием при записи полностью в пространстве пользователя.

Заготовка для fork() уже находится в lib/fork.c. Как и dumbfork(), fork() должен создавать новый процесс, сканировать все адресное пространство родительского процесса и создавать соответствующие отображения страниц в дочернем. Ключевым отличием является то, что, в то время как dumbfork() копирует страницы, fork() изначально будет только создавать новые отображения для тех же физических страниц. Каждая страница копируется только тогда, когда один из процессов пытается писать в нее.

Поток управления fork() заключается в следующем:

1. Родитель устанавливает pgfault() в качестве обработчика ошибок страницы С-уровня, используя set\_pgfault\_handler().
2. Родитель вызывает sys\_exofork(), чтобы создать дочерний процесс.
3. Для каждой записываемой или copy-on-write-страницы в своем адресном пространстве ниже УТОР родитель вызывает duppage, которая должна отобразить страницу как copy-on-write в адресное пространство дочернего процесса, а затем переназначить страницу как copy-on-write в своем собственном адресном пространстве. Функция duppage устанавливает оба PTE так, чтобы страница не была доступна для записи, и так, что они содержат PTE\_COW в поле avail, чтобы отличить copy-on-write-страницы от страниц, на самом деле доступных только для чтения.

Стек исключений не переназначается таким образом. Вместо этого нужно выделить для него в дочернем процессе новую страницу. Поскольку обработчик ошибки страницы производит фактическое копирование страниц и при этом работает на стеке исключений, стек исключений не может быть сделан copy-on-write. `fork()` также должен обрабатывать страницы, которые присутствуют (present), но не являются доступными для записи или copy-on-write.

4. Родительский процесс устанавливает такую же, как у себя, точку входа в обработчик ошибки страницы в дочернем процессе.
5. Дочерний процесс в настоящее время готов работать, так что родитель помечает его как runnable.

Каждый раз, когда один из процессов пишет в copy-on-write-страницу, в которую он еще не писал, будет происходить ошибка страницы. Вот поток управления пользовательского обработчика ошибок страниц:

1. Ядро передает ошибку страницы в `_pgfault_upcall`, который вызывает обработчик `pgfault()`.
2. `pgfault()` проверяет, что ошибка является доступом на запись (проверка на `FEC_WR` в коде ошибки), а PTE для страницы помечен `PTE_COW`. Если нет, вызывает `panic()`.
3. `pgfault()` выделяет новую страницу, отображаемую во временном месте, и копирует содержимое вызвавшей ошибку страницы в нее. Затем обработчик ошибок отображает новую страницу на соответствующий адрес с разрешениями на чтение/запись вместо старого отображения только для чтения.

► Реализуйте `fork()`, `duppage()` и `pgfault()` в `lib/fork.c`.

Проверьте свой код программой `forktree`. Она должна выдавать следующие сообщения, с вкраплениями сообщений

*«new env», «free env» и «exiting gracefully». Сообщения не-обязательно появляются в таком порядке, и идентифи-каторы процессов могут отличаться.*

```
1000: I am "
1001: I am '0'
2000: I am '00'
2001: I am '000'
1002: I am '1'
3000: I am '11'
3001: I am '10'
4000: I am '100'
1003: I am '01'
5000: I am '010'
4001: I am '011'
2002: I am '110'
1004: I am '001'
1005: I am '111'
1006: I am '101'
```

## **2.6 Межпроцессное взаимодействие**

Ранее мы сосредоточились на изоляции процессов и способах создания иллюзии, что для каждой программы компьютер находится в полном ее распоряжении. Другой важной функцией операционной системы является предоставление механизмов для взаимодействия программ между собой. Один из примеров такого механизма — сигналы Unix.

Существуют различные модели межпроцессного взаимодействия, каждая из которых имеет свои достоинства и недостатки. Далее мы реализуем одну из таких моделей.

### **2.6.1 Базовый механизм межпроцессного взаимодействия в JOS**

С точки зрения процесса механизм межпроцессного взаимодействия будет реализован в виде двух новых системных вызовов, `sys_ipc_recv()` и `sys_ipc_try_send()`, и двух библиотечных оберток для них, `ipc_recv()` и `ipc_send()`.

«Сообщения», которые пользовательские процессы могут отправлять друг другу, используя механизм IPC, состоят из двух компонентов: 32-битное значение и optionalный адрес начала страницы. Возможность для процессов передавать страницы в сообщениях является простым и эффективным способом передачи большего объема данных, чем одно 32-битное целое число, а также позволяет процессам использовать общую память.

### **2.6.2 Отправка и прием сообщений**

Для получения сообщения процесс вызывает `sys_ipc_recv()`. Этот системный вызов усыпляет текущий процесс и не запускает его снова, пока сообщение не получено. Когда процесс ждет сообщения, любой другой процесс может послать ему сообщение — необязательно некий конкретный

процесс, и необязательно процесс, имеющий с принимающим связи родительский/дочерний. Другими словами, при IPC не происходит никакой проверки прав доступа, так как его механизм разработан так, чтобы быть «безопасным»: один процесс не может испортить другой путем посылки IPC-сообщения.

Чтобы попытаться отправить сообщение, процесс вызывает `sys_ipc_try_send()` с `id` целевого процесса и значением, которое будет отправлено. Если указанный процесс действительно в данный момент принимает сообщения (вызвал `sys_ipc_recv()` и еще не получил значение), то функция доставляет сообщение и возвращает 0. В противном случае функция возвращает `-E_IPC_NOT_RECV`, чтобы указать, что целевой процесс в данный момент не ожидает сообщения.

Функция `ipc_recv()` в пользовательском пространстве вызывает `sys_ipc_recv()` и ищет информацию о полученных значениях в `struct Env` текущего процесса. Аналогично, функция `ipc_send()` вызывает `sys_ipc_try_send()` до тех пор, пока передача не удастся.

### 2.6.3 Передача страниц

Когда процесс вызывает `sys_ipc_recv()` с правильным параметром `dstva` (ниже UTOP), он таким образом заявляет, что готов получить страницу. Если отправитель посыпает страницу, то эта страница будет отображена по адресу `dstva` в адресном пространстве получателя. Если получатель уже имеет страницу, отображенную по адресу `dstva`, то отображение этой страницы удаляется.

Когда процесс вызывает `sys_ipc_try_send()` с правильным `srcva` (ниже UTOP), это означает, что отправитель хочет отправить страницу, отображенную в данный момент по адресу `srcva`, с разрешениями `perm`. После успешного осуществления IPC отправитель сохраняет свое первоначальное отображение для страницы по адресу `srcva` в своем адресном

пространстве, а получатель принимает отображение для этой же физической страницы по адресу dstva, первоначально указанному получателем, в адресном пространстве получателя. В результате эта страница становится общей для отправителя и получателя.

Если отправитель или получатель не заявляют, что должна быть передана страница, то страница не передается. После любого IPC ядро устанавливает новое поле env\_ipc\_perm в структуре Env получателя на права доступа к полученной странице, или 0, если страница не была получена.

► Реализуйте `sys_ipc_recv()` и `sys_ipc_try_send()` в `kern/syscall.c`. Прочтайте комментарии к обеим функциям до того, как начинать работать над ними, так как они должны работать вместе. При вызове `envid2env` в этих функциях вы должны установить флаг `checkperm` в 0, это означает, что любой процесс имеет право отправлять IPC-сообщения любому другому процессу, и ядро не делает никаких специальных проверок разрешений, кроме подтверждения того, что целевой `envid` является правильным.

Затем реализуйте функции `ipc_recv()` и `ipc_send()` в `lib/ipc.c`.

Используйте `user/pingpong` и `user/primes` для тестирования механизма IPC.

После выполнения этого задания рекомендуется проверить разработанный код с помощью `make grade`. Скрипт должен выводить OK для всех тестов.

По окончании выполнения работы следует сохранить внешние изменения и отправить в удаленный репозиторий.

## 2.7 Файловая система

Еще одной проблемой, решаемой операционной системой, является долговременное хранение программ и данных: процесс может хранить в своем адресном пространстве лишь ограниченный объем данных, и по завершении процесса эти данные теряются. Кроме того, иногда есть необходимость предоставить доступ к данным сразу нескольким процессам.

Эта проблема решается путем использования внешних устройств: магнитных дисков, твердотельных накопителей, оптических дисков и т. д. BIOS предоставляет доступ к таким устройствам в виде линейной последовательности блоков, однако для пользовательских программ такой способ почти всегда неудобен по множеству причин. Поэтому операционная система предоставляет доступ к внешним устройствам хранения на более абстрактном уровне в виде логических блоков, называемых файлами. Способ их хранения, интерфейс для доступа к ним, контроль доступа к ним различных процессов и другие аспекты хранения описываются файловой системой. О различных разновидностях файловых систем и их реализации можно прочитать в главе 6 уже упоминавшейся книги Э. Танненбаума [10].

Данный раздел соответствует лабораторной работе №10. Для получения файлов и изменений, необходимых для выполнения работы, следует обновить удаленный репозиторий, создать новую ветвь под названием `working-lab10`, после чего выполнить слияние с ней ветви `lab10`, которая появилась в репозитории. О работе с `git` можно прочитать в разд. 3.2.

Файлы, которые появились в этой работе, в основном связаны с поддержкой файловой системы и расположены в новой директории `fs`. Просмотрите все файлы в этой директории. Кроме того, некоторые файлы, связанные с поддержкой файловой системы, появились в директориях `user` и `lib`.

После слияния кода данной работы запустите программы pingpong, primes и forktree из предыдущей работы. Для начала нужно закомментировать строку ENV\_CREATE(fs\_fs) в kern/init.c, потому что fs/fs.c пытается совершить некоторые еще не реализованные операции ввода/вывода. Кроме того, временно закомментируйте вызов close\_all() в lib/exit.c; эта функция также использует еще не реализованные функции. Если ваши предыдущие работы выполнены верно, тесты должны работать правильно; не продолжайте, пока не убедитесь в этом. После выполнения первого упражнения раскомментируйте эти строки.

Поддержка файловой системы, которая поддерживает все базовые возможности (создание, чтение, запись, удаление файлов), в целом уже реализована. Вам нужно будет лишь изменить существующий код таким образом, чтобы файловая система заработала в JOS. Для этого необходимо знать все детали реализации файловой системы или ее структуру на диске, но важно ознакомиться с принципами ее работы и программным интерфейсом.

Файловая система реализована в микроядерном стиле, вне ядра, в пределах своего собственного пользовательского процесса (т. н. сервера файловой системы). Другие процессы получают доступ к файловой системе с помощью IPC-запросов.

## 2.7.1 Доступ к диску

Сервер файловой системы в JOS должен получать доступ к диску, но эта возможность еще не реализована. Вместо того, чтобы использовать обычную стратегию монолитного ядра, — добавление IDE-драйвера в ядро вместе с необходимыми системными вызовами для доступа к файловой системе — IDE-драйвер будет реализован как часть сервера файловой системы. Однако для того, чтобы дать серверу файловой си-

стемы возможность непосредственно обращаться к диску, все же придется модифицировать ядро.

Реализовать доступ к диску в пользовательском пространстве несложно, если использовать PIO («programmed I/O») и не использовать дисковые прерывания. Можно реализовать дисковый драйвер, основанный на прерываниях, в пользовательском режиме (например, таким образом он реализован в ядрах L3 и L4), но это труднее, так как ядро должно отправлять дисковые прерывания нужному пользовательскому процессу.

x86-процессор использует биты IOPL в регистре EFLAGS, чтобы определить, разрешается ли коду защищенного режима выполнять специальные инструкции ввода-вывода, такие как IN и OUT. Так как все регистры IDE-дисков, к которым необходимо получить доступ, расположены в пространстве ввода/вывода x86, не отображаясь в память, единственное, что нужно сделать для доступа к ним — дать процессу привилегию I/O. В сущности, биты IOPL регистра EFLAGS обеспечивают простой метод контроля доступа к пространству ввода-вывода для кода пользовательского режима — «все или ничего». Для разграничения доступа нужно, чтобы сервер файловой системы имел возможность доступа к пространству ввода-вывода, а другие процессы не имели.

► Функция *i386\_init* идентифицирует сервер файловой системы, передавая тип ENV\_TYPE\_FS в функцию создания процесса (*env\_create*). Измените функцию *env\_create* в *env.c*, чтобы она давала такому процессу привилегию ввода/вывода, но не давала ее никакому другому процессу. Убедитесь, что вы можете запустить сервер файловой системы без ошибки. При запуске *make grade* ваш код должен пройти тест «*fs i/o*».

► Ответьте на следующий вопрос в файле *answers-lab10.txt*:

*Нужно ли сделать что-нибудь еще, чтобы убедиться, что привилегия ввода/вывода будет сохранена и восстановлена должным образом при переключении между процессами? Почему?*

В предыдущих лабораторных работах образ ядра вместе с пользовательскими программами после сборки находился в файле `obj/kernel/kernel.img`, который при запуске использовался QEMU как образ для диска 0 («Диск С:» в DOS/Windows). В данной работе происходит то же самое, но, помимо этого, в качестве образа для диска 1 («Диск D:») используется новый файл `obj/fs/fs.img`, в котором находятся в т.ч. новые пользовательские программы. Файловая система должна использовать только диск 1; диск 0 используется только для загрузки ядра.

## 2.7.2 Кэш блоков

Кроме того, для файловой системы будет реализован простой кэш блоков с помощью системы виртуальной памяти процессора. Его код находится в `fs/bc.c`.

Файловая система будет иметь ограничение на размер диска — не более 3 ГБ. Для отображения в память диска резервируется фиксированная область адресного пространства сервера файловой системы размером 3 ГБ, от `0x10000000` (`DISKMAP`) до `0xD0000000` (`DISKMAP + DISKMAX`). Например, блок 0 отображается по виртуальному адресу `0x10000000`, блок 1 отображается по виртуальному адресу `0x10001000` и т.д. Функция `diskaddr` в `fs/bc.c` реализует эту трансляцию из номеров блоков в виртуальные адреса (вместе с некоторыми проверками).

Так как сервер файловой системы имеет собственное виртуальное адресное пространство, независимое от адресных

пространств других процессов, и единственное, что он должен делать — реализовать доступ к файлам, такое резервирование большей части адресного пространства допустимо. Для реальной реализации файловой системы на 32-битном процессоре это было бы странно, так как современные диски обычно больше 3ГБ. Однако для 64-битного процессора такой подход также может быть допустим.

Конечно, было бы неразумно читать в память сразу все содержимое диска. Вместо этого выделение страницы памяти и чтение необходимого блока с диска будет происходить при возникновении ошибки страницы. Таким образом создается иллюзия, что весь диск находится в памяти.

► *Реализуйте функции `bc_pgfault()` и `flush_block()` в `fs/bc.c`.*

*Функция `bc_pgfault()` используется в качестве обработчика ошибки страницы, при ее возникновении загружая страницу с диска. При ее написании имейте в виду, что (1) `addr` может не быть выровнен по границе блока и (2) `ide_read` оперирует секторами, а не блоками. Функция `flush_block()` записывает блок на диск, если необходимо, но ничего не делает в случаях, когда блок не находится в кэше или не помечен как *dirty*. Блок помечается как *dirty* (т.е. как измененный с тех пор, как было произведено чтение или запись) процессором при обращении к странице на запись (см. раздел 4.8 руководства [8]), то есть для проверки необходимости записи блока на диск достаточно проверить бит `PTE_D` в `uvpt`. После записи блока на диск функция `flush_block()` должна очищать флаг `PTE_D` с помощью `sys_page_map`.*

*Используйте `make grade` для тестирования кода. Код должен проходить проверки «`check_bc`», «`check_super`», «`check_bitmap`».*

Функция `fs_init()` в `fs/fs.c` является ярким примером использования кэша. После инициализации кэша она записыва-

ет указатели на область отображения диска в глобальной переменной `super`. С этого момента можно просто читать данные из структуры `super`, как если бы она находилась в памяти, а обработчик ошибок страницы будет читать их с диска по мере необходимости.

После установки в `fs_init()` указателя `bitmap` этот указатель можно рассматривать как битовый массив, в котором каждый бит соответствует блоку на диске; см., например, функцию `block_is_free()`, которая проверяет, помечен ли данный блок как свободный.

► Реализуйте `alloc_block()` по аналогии с `free_block()`. Эта функция должна находить свободный блок в `bitmap`, помечать его как используемый и возвращать его номер. Сразу после выделения блока необходимо записать изменившийся блок на диск с помощью `flush_block()`, чтобы сохранить целостность файловой системы. После выполнения этого упражнения ваш код должен проходить проверку «`alloc_block`».

### 2.7.3 Операции с файлами

Основные функции для операций с файлами представлены в файле `fs/fs.c`. Просмотрите их и убедитесь, что вы понимаете, что делает каждая из функций.

► Реализуйте функции `file_block_walk()` и `file_get_block()`. После выполнения этого упражнения ваш код должен проходить проверки «`file_open`», «`file_get_block`», «`file_flush/file_truncated/file_rewrite`», «`testfile`».

## 2.7.4 Интерфейс файловой системы

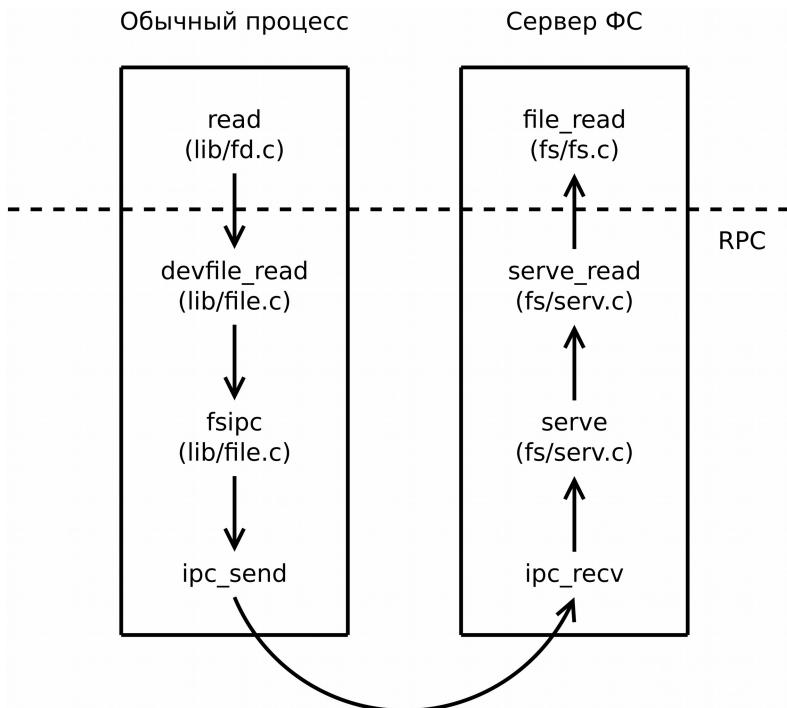


Рисунок 2.7.1: Вызов сервера файловой системы

Теперь, когда сервер файловой системы может получать доступ к диску, нужно сделать его доступным для других процессов, которые хотят использовать файловую систему. Так как другие процессы не могут напрямую вызывать функции в сервере файловой системы, доступ будет осуществляться с помощью удаленного вызова процедур, или RPC – абстракции, построенной на базе механизма IPC. Графически вызов сервера файловой системы (например, `read()`) изображен на рис. 2.7.1. Все, что ниже пунктирной линии, – просто механика передачи запроса на чтение от обычного процесса к серверу файловой системы. Функция `read()` работает на любых файловых дескрипторах и просто вызывает

ет соответствующую функцию чтения для устройства, в данном случае `devfile_read()` (теоретически могут существовать различные типы устройств, такие как каналы). Функция `devfile_read()` реализует `read()` для файлов на диске. Эта и другие функции `devfile_*` в `lib/file.c` реализуют клиентскую сторону операций с ФС и работают приблизительно одинаковым образом, собирая аргументы в структуру запроса, вызывая `fsipc` для отправки IPC-запроса, а затем распаковывая и возвращая результат. Функция `fsipc` обрабатывает отправку запроса к серверу и получение ответа.

Код сервера файловой системы можно найти в `fs/serv.c`. Он реализован как бесконечный цикл с функцией `serve()`, получающей запрос от IPC, диспетчеризующей запрос в соответствующую функцию-обработчик и отправляющей результат обратно через IPC. При выполнении операции чтения, например, `serve()` направит запрос в `serve_read()`, которая позаботится об особенностях IPC, соответствующих запросу на чтение, таких как распаковка структуры запроса и, наконец, вызов `file_read()`, фактически выполняющий чтение файла.

Напомним, что механизм межпроцессного взаимодействия JOS позволяет процессу отправить одно 32-битное число и, возможно, открыть страницу для совместного доступа. Для отправки запроса от клиента к серверу используется 32-битное число, обозначающее тип запроса (функции RPC сервера файловой системы пронумерованы, как и системные вызовы), а аргументы запроса хранятся в `union Fsipc` на странице, к которой открыт доступ с помощью IPC. На стороне клиента страница отображается с разделяемым доступом как `fsipcbuf`; на стороне сервера она отображается из входящего запроса в `fsreq` (0xFFFFF000).

Кроме того, сервер отправляет ответ обратно через IPC. 32-битное число здесь используется для кода возврата функции, и это единственное, что возвращает большая часть

функций RPC. Однако FSREQ\_READ и FSREQ\_STAT также возвращают данные, просто записывая их на страницу, на которой клиент отправил запрос. Нет необходимости отправлять эту страницу в ответе IPC, так как изначально именно клиент отправил ее серверу файловой системы. Кроме того, в своем ответе FSREQ\_OPEN открывает клиенту доступ к новой FD-странице (странице файлового дескриптора).

► Реализуйте функции `serve_read()`, `serve_write()` в `fs/serv.c` и `devfile_write()` в `lib/file.c`. Функция `serve_read()` должна, по сути, просто предоставлять интерфейс к функции `file_read()`. Просмотрите комментарии и код в функции `serve_set_size()`, чтобы понять, как должны быть устроены функции сервера файловой системы.

После этого код должен проходить проверки «`serve_open / file_stat / file_close`», «`file_read`», «`file_write`», «`file_read after file_write`», «`open`» и «`large file`».

После выполнения этого задания рекомендуется проверить разработанный код с помощью `make grade`. Скрипт должен выводить OK для всех тестов.

По окончании выполнения работы следует сохранить внесенные изменения и отправить в удаленный репозиторий.

## 2.8 Командная оболочка

В результате выполнения предыдущих работ было реализовано управление памятью, процессами, исключениями и прерываниями, системные вызовы, работа с внешними накопителями. Однако почти отсутствуют средства взаимодействия с пользователем: такое взаимодействие производится с помощью нескольких наперед заданных команд монитора, не позволяющих даже запускать произвольные программы, либо путем автоматического внесения изменений в код при запуске ОС (команды для запуска программ make run-X).

Простым и универсальным способом взаимодействия с операционной системой является текстовый интерфейс, реализацией которого является командная оболочка. Она присутствует в различном виде почти во всех операционных системах. Одной из наиболее распространенных командных оболочек для UNIX-подобных операционных систем, в частности, Linux, является bash (подробнее о ней можно прочитать в ее руководстве [15]).

В JOS командная оболочка в основном уже реализована, далее будут добавлены возможности операционной системы, необходимые ей для работы, и некоторые возможности самой оболочки. Помимо последовательного выполнения команд (REPL – Read-Eval-Print Loop) оболочки также поддерживает их пакетное выполнение из файлов на внешнем накопителе.

Данный раздел соответствует лабораторной работе №11. Для получения файлов и изменений, необходимых для выполнения работы, следует обновить удаленный репозиторий, создать новую ветвь под названием working-lab11, после чего выполнить слияние с ней ветви lab11, которая появилась в репозитории. О работе с git можно прочитать в разд. 3.2.

### 2.8.1 Запуск процессов

В репозиторий был добавлен код функции `spawn()`, которая создает новый процесс, загружает в него образ программы из файловой системы, а затем запускает. Родительский процесс затем продолжает выполнение независимо от дочернего. Функция `spawn()`, по сути, соответствует выполнению `fork()` с последующим выполнением в дочернем процессе `exec()`.

Мы реализовали `spawn()` вместо `exec()`, потому что `spawn()` легче реализовать в пространстве пользователя в стиле экзоядра, без использования средств ядра. Подумайте, что необходимо для реализации `exec()` в пространстве пользователя, и убедитесь, что вы понимаете, почему такой вариант труднее.

► *Функция `spawn()` использует новый системный вызов `sys_env_set_trapframe()` для инициализации состояния вновь созданного процесса. Реализуйте `sys_env_set_trapframe()`. Проверьте свой код, запустив `user/spawnhello` в `kern/init.c`. Эта программа будет пытаться выполнить `spawn()` для программы `/hello` в файловой системе.*

*Используйте `make grade` для тестирования написанного кода. Если при выполнении `user/spawnhello` возникает ошибка страницы (сообщение «`page fault in FS`»), проследите, как работает сервер файловой системы, и подумайте, что произойдет, если в системе не останется ни одного процесса кроме него.*

### 2.8.2 Разделение файловых дескрипторов

Файловые дескрипторы UNIX являются общим понятием, включающим в себя также каналы, консольный ввод-вывод и т.д. В JOS каждый из этих типов устройств имеет соот-

ветствующую структуру struct Dev с указателями на функции, реализующими чтение, запись и т. д. для данного типа устройств. В файле lib/fd.c находится реализация интерфейса UNIX-подобных файловых дескрипторов на ее основе. Каждая структура struct Fd соответствует своему типу устройства; большинство функций в lib/fd.c просто вызывает функции в соответствующих структурах Dev.

В lib/fd.c также находится управление начинаяющимися с FSTABLE таблицами дескрипторов в адресном пространстве каждого процесса. Эта область памяти резервирует страницу (4 КБ) памяти для каждого из файловых дескрипторов, которые программа может открыть одновременно (до MAXFD – в настоящее время 32). В любой момент времени каждая страница таблицы дескрипторов отображена только в том случае, если используется соответствующий дескриптор. Каждый дескриптор также имеет дополнительную «страницу данных» в области памяти, начинающейся с FILEDATA, которую устройства могут использовать, если им это необходимо.

Сейчас при вызове fork() эта память будет отмечена для копирования при записи, так что ее состояние будет копироваться. Это означает, что процессы не смогут перемещаться по файлам, открытым родительским процессом. После вызова spawn() память вообще не будет скопирована, то есть запущенные процессы начинают работать без открытых файловых дескрипторов. Необходимо, чтобы память родительского процесса, относящаяся к файловым дескрипторам, была доступна из дочерних процессов.

Далее нужно изменить fork() таким образом, чтобы эта функция учитывала, что определенные области памяти используются операционной системой и всегда должны быть общими. Вместо того, чтобы жестко задавать их список в коде, нужно установить специальный бит в записях таблицы страниц (аналогично PTE\_COW).

В lib/lib.h определена новая константа PTE\_SHARE. Она соответствует одному из трех бит записи таблицы страниц, которые описаны как «доступные для использования программами» в руководствах Intel и AMD. Мы установим соглашение, что, если у записи в таблице страниц установлен этот бит, данная запись должна быть скопирована непосредственно от родительского процесса к дочернему при вызове fork() и spawn. Обратите внимание, что это отличается от маркировки ее для копирования при записи: как описано в первом абзаце, мы хотим убедиться, что физическая страница является общей.

► Измените функцию durrepage в lib/fork.c так, чтобы она следовала новому соглашению. Если у записи таблицы страниц установлен бит PTE\_SHARE, просто скопируйте ее напрямую. Вы должны использовать константу PTE\_SYSCALL, а не 0xFFFF, чтобы маскировать соответствующие биты записи таблицы страниц. 0xFFFF маскирует также биты dirty и accessed.

Кроме того, реализуйте функцию copy\_shared\_pages() в lib/spawn.c. Она должна пройти по всем записям таблицы страниц текущего процесса (как сделал fork()), копируя любые отображения страниц, у которых установлен бит PTE\_SHARE, в дочерний процесс.

Используйте make run-testpteshare, чтобы проверить, что ваш код работает правильно. Вы должны увидеть строки «fork handles PTE\_SHARE right» и «spawn handles PTE\_SHARE right».

Используйте make run-testfdsharing, чтобы проверить, что файловые дескрипторы правильно разделяются. Вы должны увидеть строки «read in child succeeded» and «read in parent succeeded».

### 2.8.3 Клавиатура

Чтобы командная оболочка могла работать, нужно обрабатывать вводимые с клавиатуры данные. Ранее QEMU отображал вводимые с клавиатуры данные на дисплей и в последовательный порт, но ввод до сих пор использовался только при работе с монитором. В QEMU данные, вводимые в графическом окне, передаются в JOS в качестве входных данных с клавиатуры, в то время как данные, вводимые в консоли, передаются через последовательный порт. Файл kern/console.c уже содержит драйвера клавиатуры и последовательного порта, которые монитор ядра использовал начиная с работы №1, но теперь вам нужно объединить их с остальной частью системы.

► В файле *kern/trap.c* вызовите *kbd\_intr()* для обработки *IRQ\_OFFSET+IRQ\_KBD* и *serial\_intr()* для обработки *IRQ\_OFFSET+IRQ\_SERIAL*.

Проверьте свой код, вызвав *make run-testkbd* и введя что-либо с клавиатуры. Система должна показать введенные символы. Попробуйте вводить текст как в консоли, так и в графическом окне.

### 2.8.4 Командная оболочка

Выполните *make run-icode* или *make run-icode-nox*. Запустится JOS, а в ней *user/icode*, который, в свою очередь, запускает *user/init*. *Init* устанавливает консольный ввод-вывод как файлы с дескрипторами 0 (ввод) и 1 (вывод). Затем *init* запускает *sh* — командную оболочку. В оболочке попробуйте выполнить следующие команды:

```
echo hello world | cat
cat lorem |cat
cat lorem |num
cat lorem |num |num |num |num |num
```

Обратите внимание, что пользовательская библиотечная функция `sprintf` выводит данные напрямую в консоль, не используя файловый дескриптор. Это удобно для отладки, но не дает возможности передать выводимые данные в другие программы. Чтобы вывести данные в конкретный файловый дескриптор, используйте `fprintf(1, "...", ...)`. Функция `printf` выводит данные в дескриптор 1. В файле `user/lsfd.c` можно увидеть примеры использования этих функций.

► *Командная оболочка не поддерживает перенаправление ввода-вывода. Реализуйте его в файле user/sh.c. После этого вы должны иметь возможность выполнить команды:*

```
cat script  
sh <script
```

*Выполните make run-testshell, чтобы проверить командную оболочку. Программа testshell просто выполняет команды, указанные выше, и сравнивает вывод с файлом fs/testshell.key. На данный момент ваш код должен проходить все тесты, запускаемые make grade.*

*По окончании выполнения работы следует сохранить внесенные изменения и отправить в удаленный репозиторий.*

## **2.9 Системные вызовы без переключения контекста**

В связи с необходимостью переключения контекста, обычные системные вызовы являются относительно медленными. Системные вызовы, которые только получают данные от ядра, но не отправляют, в Linux реализуются в стиле `vsyscall` без переключения контекста путем простого чтения разделяемой между ядром и процессами памяти; при этом задача ядра — держать получаемые данные в актуальном состоянии.

В данной работе будет реализовано окружение, необходимое для запуска простого варианта программы `date` — без поддержки установки времени, часовых поясов, локализации и т.д. Для этого нужно реализовать системный вызов в обычном стиле, получающий из CMOS текущее время, и аналогичный системный вызов без переключения контекста.

Данный раздел соответствует лабораторной работе №12. Для получения файлов и изменений, необходимых для выполнения работы, следует обновить удаленный репозиторий, создать новую ветвь под названием `working-lab12`, после чего выполнить слияние с ней ветви `lab12`, которая появилась в репозитории. О работе с `git` можно прочитать в разд. 3.2.

### **2.9.1 Получение текущего времени**

Для получения текущего времени от часов RTC необходимо прочитать данные из памяти CMOS аналогично разд. 2.2.5. Данные расположены в указанных в табл. 2.9.1 регистрах CMOS, для которых также заданы соответствующие константы в `kern/kclock.h`, и представлены в двоично-десятичном формате (BCD). Для конвертации их в двоичный формат можно использовать макрос `BCD2BIN`.

Регистр	Значение
0x00	Секунды
0x02	Минуты
0x04	Часы
0x07	День
0x08	Месяц
0x09	Год

Таблица 2.9.1: Регистры CMOS

Микросхема часов RTC работает медленно, поэтому в момент, когда происходит обновление времени, она может выдавать неправильные значения. Например, при обновлении с 8:59:59 до 9:00:00 могут быть прочитаны не только эти два значения, но и 8:59:00 или 8:00:00. Для обработки такой ситуации существует бит *Update in progress* (бит 7 в регистре А – 0xA), однако просто проверить этот бит на равенство 0 недостаточно: обновление времени может начать происходить сразу после такой проверки. Простой способ обработать такую ситуацию — прочитать время два раза и сравнить полученные значения: если они изменились, значит, произошло обновление времени и полученные данные могут быть неверными; в таком случае необходимо прочитать их еще раз.

Часто используется представление времени в виде одного числа (UNIX time, POSIX time, timestamp): количества секунд с 00:00:00 1 января 1970 года. Этот формат удобен для передачи времени, в частности, между программой и ОС посредством системного вызова. Функции для конвертации времени в этот формат и из него присутствуют в `inc/time.h`.

► Получите текущее время из CMOS в формате UNIX time в функции `gettime` в `kern/kclock.c` и реализуйте системный вызов `sys_gettime`.

Теперь программа `date` должна выдавать текущее время (в часовом поясе UTC).

## 2.9.2 Системные вызовы без переключения контекста

В данном разделе будет реализован псевдо-системный вызов `vsys_gettime`, используемый программой `vdate`, который работает без переключения контекста путем простого чтения разделяемой между ядром и процессами памяти. Для его работы необходимо реализовать описанный выше механизм, а также обновлять по таймеру значение текущего времени, расположенного по известному процессам адресу.

► В `inc/memlayout.h` задана константа `UVSYS`, равная 0. Измените ее и другие значения таким образом, чтобы она соответствовала доступной процессам области памяти.

В `kern/pmap.c` выделите массив для разделяемых данных (длина массива определяется в `inc/vsyscall.h`) и отобразите его только для чтения для процессов по адресу `UVSYS`.

В `lib/entry.S` добавьте переменную, соответствующую этой области памяти, для пользовательских программ.

В `lib/vsyscall.c` реализуйте чтение ячейки памяти с номером, соответствующим системному вызову (номера заданы в `inc/vsyscall.h`).

Добавьте обновление соответствующей ячейки памяти при срабатывании прерывания от часов.

Теперь программа `vdate` должна выдавать текущее время. По окончании выполнения работы следует сохранитьнесенные изменения и отправить в удаленный репозиторий.

# 3 Справочные материалы

## 3.1 Настройка окружения и запуск JOS

Для выполнения лабораторных работ необходимо запустить операционную систему JOS в виртуальной машине QEMU. Для этого можно использовать специально подготовленную виртуальную машину в формате VirtualBox с уже установленной QEMU, которую можно скачать по адресу, предоставленному преподавателем. Затем следует установить систему виртуализации VirtualBox ([virtualbox.org](http://virtualbox.org)), запустить ее, выбрать в меню File -> Import Appliance... и выбрать скачанный файл. Логин и пароль – user/password.

Для работы VirtualBox требуется поддержка процессором аппаратной виртуализации. Ее поддерживает большая часть современных процессоров, но в некоторых случаях она оказывается отключена. Если виртуальная машина по непонятной причине не запускается, следует проверить настройку аппаратной виртуализации в BIOS. Для процессоров Intel настройка обычно содержит название технологии VT-x, для процессоров AMD – SVM или AMD-V.

Вместо виртуальной машины можно использовать любую Linux систему, в которую рекомендуется установить модифицированную версию QEMU. Эта версия содержит улучшения направленные на повышение удобства отладки ядра. Инструкция по установке:

1. Клонируйте git-репозиторий:

```
git clone http://sed.ispras.ru/git/qemu -b oscourse
```

2. В случае, если вы используете 64-битную операционную систему, установите 32-битные библиотеки. В

Debian/Ubuntu для этого нужно установить пакет gcc-multilib.

3. Установите библиотеку SDL, которая требуется QEMU для поддержки графического VGA-окна. На Debian/Ubuntu для этого нужно установить пакет libsdl1.2-dev.
4. При необходимости установите библиотеку pixman. На Debian/Ubuntu это пакет libpixman-1-dev.
5. Сконфигурируйте QEMU:

```
./configure --disable-kvm --disable-spice --target-list="i386-softmmu x86_64-softmmu" [--prefix=PFX]
```

Параметр target-list ограничивает набор поддерживаемых архитектур до необходимых. Опциональный параметр prefix указывает место для установки QEMU. По умолчанию QEMU будет установлена в /usr/local. Если у вас нет привилегий администратора, укажите в качестве префикса путь в домашней директории, например: --prefix=/home/student/qemu.

6. Выполните

```
make && sudo make install
```

После установки QEMU нужно клонировать репозиторий с исходным кодом лабораторных работ по адресу, предоставленному преподавателем, и переключиться на ветку lab1. Клонирование репозитория описано в разд. 3.2.1, управление ветками — в разд. 3.2.14 и 3.2.22.

Затем нужно перейти в каталог с репозиторием и собрать JOS, выполнив make. После этого можно запустить ее, выполнив make qemu:

```
$ cd oscourse  
$ make  
$ make qemu
```

Если все сделано правильно, в окне QEMU появится текст следующего вида:

```
Booting from Hard Disk...
6828 decimal is XXX octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

Все, что выведено после строки «Booting from Hard Disk...», выведено скелетом ядра JOS. «K>» — приглашение монитора (интерактивной программы управления).

## 3.2 Работа с Git

В данном разделе описаны некоторые приемы работы с системой контроля версий Git. В качестве более полного и подробного руководства можно использовать книгу Pro Git [13] или официальную документацию [14].

### 3.2.1 Клонирование репозитория

Для того, чтобы получить копию существующего git-репозитория, нужно использовать команду `git clone`. При этом Git копирует практически все данные, которые есть на сервере, в том числе каждую версию каждого файла.

Клонирование репозитория осуществляется командой `git clone <url>`. Например, если вы хотите клонировать исходный код лабораторных работ по данному курсу, вы можете сделать это следующим образом:

```
$ git clone http://sed.ispras.ru/git/oscourse
```

Эта команда создает каталог с именем `oscourse`, инициализирует в нем каталог `.git`, скачивает все данные для этого репозитория и извлекает в каталог репозитория рабочую копию последней версии кода. Если вы хотите клонировать репозиторий в каталог, отличный от `oscourse`, можно указать его название следующим параметром:

```
$ git clone http://sed.ispras.ru/git/oscourse oscourse-2015
```

Эта команда делает все то же самое, что и предыдущая, только результирующий каталог будет назван `oscourse-2015`.

В Git'е реализовано несколько транспортных протоколов, которые вы можете использовать. В предыдущем примере использовался протокол `http://`, вы также можете встретить `https://`, `git://` или `ssh://`.

### 3.2.2 Просмотр состояния репозитория

Для просмотра состояния репозитория используется команда `git status`. В случае, если в репозитории нет ни новых, ни измененных файлов, вывод этой команды будет выглядеть примерно так:

```
$ git status  
# On branch master  
nothing to commit, working directory clean
```

Таким образом, Git сообщает, что в рабочем каталоге нет ни измененных, ни добавленных файлов. Кроме того, команда выводит название текущей ветки.

Предположим, вы добавили в рабочий каталог новый файл под названием `README`. В таком случае вывод команды будет выглядеть так:

```
$ echo "readme" > README  
$ git status  
# On branch master  
# Untracked files:  
#   (use "git add <file>..." to include in what will be committed)  
#  
# README  
nothing added to commit but untracked files present (use "git add" to track)
```

Git сообщает, что в рабочем каталоге присутствует неотслеживаемый (`untracked`) файл. Статус «неотслеживаемый файл» означает, что файл отсутствует в предыдущем снимке состояния (коммите); чтобы этот файл отслеживался и был сохранен при следующем коммите, его нужно добавить в репозиторий явным образом.

### **3.2.3 Отслеживание новых файлов**

Чтобы начать отслеживать (добавить под версионный контроль) новый файл, необходимо использовать команду git add. Например, начать отслеживать файл README можно следующим образом:

```
$ git add README
```

Если теперь снова выполнить команду git status, файл README будет отмечен как отслеживаемый:

```
$ git status  
# On branch master  
# Changes to be committed:  
#   (use "git reset HEAD <file>..." to unstage)  
#  
#   new file: README
```

Файл находится в секции «Changes to be committed». Если выполнить коммит в этот момент, то версия файла, существовавшая на момент выполнения команды git add, будет сохранена. Команда git add принимает в качестве параметра путь к файлу или каталогу; при этом для каталога команда рекурсивно добавляет (индексирует) все файлы, содержащиеся в нем.

### **3.2.4 Индексация измененных файлов**

Если изменить отслеживаемый файл kern/pmap.c и после этого снова выполнить команду git status, результат будет примерно следующим:

```
$ git status  
# On branch master  
# Changes to be committed:
```

```
# (use "git reset HEAD <file>..." to unstage)
#
# new file: README
#
# Changes not staged for commit:
# (use "git add <file>..." to update what will be committed)
#
# modified: kern/pmap.c
```

Файл kern/pmap.c находится в секции «Changes not staged for commit» — это означает, что отслеживаемый файл был изменен в рабочем каталоге, но пока не проиндексирован. Чтобы проиндексировать его, необходимо выполнить команду git add:

```
$ git add kern/pmap.c
$ git status
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
# new file: README
# modified: kern/pmap.c
```

Теперь оба файла проиндексированы и войдут в следующий коммит. Git индексирует файл в том состоянии, в котором он находился на момент выполнения git add. Если после этого изменить kern/pmap.c еще раз, часть изменений будет отмечена как проиндексированная, а часть нет; чтобы проиндексировать новые изменения, нужно выполнить git add еще раз:

```
$ vim kern/pmap.c
$ git status
```

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file: README
# modified: kern/pmap.c
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
# modified: kern/pmap.c
$ git add kern/pmap.c
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file: README
# modified: kern/pmap.c
```

### 3.2.5 Просмотр изменений

Чтобы просмотреть изменения, внесенные в файл, следует использовать команду `git diff`:

```
$ git diff
diff --git a/kern/pmap.c b/kern/pmap.c
index 03efcbf..f191f9e 100644
--- a/kern/pmap.c
+++ b/kern/pmap.c
@@ -145,7 +145,7 @@ mem_init(void)
```

```
// Your code goes here:  
-  
+    pages = boot_alloc(num_pages);  
// Make 'envs' point to an array of size 'NENV' of 'struct Env'.
```

Как видите, в файле kern/pmap.c удалена пустая строка, а вместо нее добавлена строка кода.

Чтобы просмотреть уже проиндексированные изменения, которые войдут в следующий коммит, нужно выполнить git diff --cached (начиная с версии 1.6.1 также можно использовать более интуитивный вариант git diff --staged). Эта команда сравнивает индексированные изменения с последним коммитом:

```
$ git diff --cached  
diff --git a/README b/README  
new file mode 100644  
index 0000000..8178c76  
--- /dev/null  
+++ b/README  
@@ -0,0 +1 @@  
+readme
```

### 3.2.6 Фиксация изменений

Простейший способ зафиксировать изменения, сохранив в репозитории все измененные файлы, для которых была выполнена команда git add — выполнить команду git commit без параметров. Эта команда откроет текстовый редактор, в котором будет отображен следующий текст:

```
# Please enter the commit message for your changes. Lines starting  
# with '#' will be ignored, and an empty message aborts the commit.
```

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file: README
#       modified: kern/pmap.c
```

После этого нужно ввести сообщение с описанием изменений, причем все строки, начинающиеся с символа #, будут проигнорированы, и сохранить файл. После выхода из текстового редактора Git создаст коммит с введенным вами сообщением.

Есть и другой способ – можно набрать комментарий к коммиту в командной строке, указав его после параметра -m, как в следующем примере:

```
$ git commit -m "Fixed README"
[master]: created e2b6543: "Fixed README."
1 file changed, 1 insertion(+), 0 deletions(-)
create mode 100644 README
```

Команда commit вывела информацию о совершенном коммите: ветку (master), контрольную сумму (e2b6543), количество измененных файлов и статистику по добавленным/удаленным строкам.

### 3.2.7 Пропуск индексации

Чтобы пропустить индексацию, можно использовать параметр -a команды commit. В таком случае каждый уже отслеживаемый на момент коммита файл будет проиндексирован, и не будет необходимости выполнять команду git add:

```
$ git status
# On branch master
```

```
#  
# Changes not staged for commit:  
#  
#   modified: kern/pmap.c  
#  
$ git commit -a -m 'Added pages array allocation.'  
[master 8f73ac5] Added pages array allocation.  
 1 files changed, 5 insertions(+), 0 deletions(-)
```

### 3.2.8 Удаление файлов

Чтобы удалить файл из репозитория, необходимо удалить его из индекса, а затем выполнить коммит. Удалить файл из индекса позволяет команда `git rm`, которая также удаляет файл из рабочего каталога, чтобы он не оставался в качестве неотслеживаемого.

Если просто удалить файл из рабочего каталога, он будет показан в секции «Changes not staged for commit» вывода `git status`:

```
$ rm README  
$ git status  
# On branch master  
#  
# Changes not staged for commit:  
#   (use "git add/rm <file>..." to update what will be committed)  
#  
#       deleted: README
```

Затем, если вы выполните команду `git rm`, удаление файла попадет в индекс:

```
$ git rm README
```

```
rm 'README'  
$ git status  
# On branch master  
#  
# Changes to be committed:  
# (use "git reset HEAD <file>..." to unstage)  
#  
# deleted: README
```

После следующего коммита файл исчезнет и больше не будет отслеживаться. При этом, если файл был изменен и уже проиндексирован, нужно использовать принудительное удаление с параметром -f.

В качестве параметра команды git rm можно указывать файлы, каталоги, а также шаблоны, например:

```
$ git rm *.log
```

### 3.2.9 Просмотр истории коммитов

Чтобы просмотреть историю изменений репозитория, используется команда git log.

```
$ git log  
commit 0222bc94714ef60789110626f9957fc813a4af71  
Author: Ivan Ivanov <ivanov@example.com>  
Date: Wed Nov 26 19:18:52 2014 +0300
```

Lab 11 added.

```
commit 94e6aada12f62df37d186e4c6e620dd472086cbb  
Author: Ivan Ivanov <ivanov@example.com>  
Date: Wed Nov 19 18:38:08 2014 +0300
```

Lab 10 added.

```
commit f32239f2968489e565fdad32e717a853cb8aacbd
```

Author: Ivan Ivanov <ivanov@example.com>

Date: Fri Nov 14 15:22:45 2014 +0400

fix

По умолчанию, без аргументов, git log выводит список коммитов в обратном хронологическом порядке, то есть последние коммиты показываются первыми. Команда отображает каждый коммит вместе с его контрольной суммой SHA-1, именем и электронной почтой автора, датой создания и комментарием.

Один из наиболее полезных параметров команды git log – параметр -r, который показывает изменения для каждого коммита. Вы также можете использовать параметр -2, который ограничит вывод до 2-х последних записей. Этот параметр показывает ту же самую информацию плюс внесенные изменения, отображаемые непосредственно после каждого коммита.

Другой полезный параметр – параметр --pretty. Он позволяет изменить формат вывода лога и может принимать несколько значений. Значение oneline выводит каждый коммит в одну строку, что удобно для просмотра большого количества коммитов. Значения short, full и fuller позволяют выводить меньше или больше деталей соответственно.

```
$ git log --pretty=oneline
```

```
0222bc94714ef60789110626f9957fc813a4af71 Lab 11 added.
```

```
94e6aada12f62df37d186e4c6e620dd472086ccb Lab 10 added.
```

```
851f3221db1910d152697591accece3169faa13a Merge branch 'lab8'  
into lab9  
f32239f2968489e565fdad32e717a853cb8aacbd fix  
2e494a46b023d48ea5c42e72a880f08f6f55764d Lab 9 added.  
297ac279ecf72a5a97ebc45361d13052e2c108f2 Lab 8 added.
```

### 3.2.10 Ограничение вывода команды log

Кроме опций для форматирования вывода, git log имеет ряд ограничительных параметров, которые дают возможность отобразить лишь часть коммитов. Один из таких параметров — параметр `-<n>`, где `n` — количество отображаемых коммитов, например, `-3`. Параметры `--since` и `--until` позволяют ограничить вывод по времени. Например, следующая команда выдаст список коммитов, сделанных за последние две недели:

```
$ git log --since=2.weeks
```

Такая команда может работать с множеством форматов — вы можете указать точную дату («`2008-01-15`») или относительную дату, такую как «`2 years 1 day 3 minutes ago`».

Еще одна полезная опция для git log — путь. Указав имя файла (или каталога), вы ограничите вывод теми коммитами, которые вносят изменения в указанный файл.

### 3.2.11 Изменение последнего коммита

Иногда возникает необходимость изменить последний коммит, например, если коммит сделан слишком рано, в него не добавлены необходимые файлы и т. д. Для этого используется опция `--amend`, например:

```
$ git commit -m 'initial commit'  
$ git add forgotten_file
```

```
$ git commit --amend
```

Все три команды вместе дают один коммит — второй коммит заменяет результат первого.

### 3.2.12 Отмена индексации файла

Иногда возникает необходимость отменить индексацию файла; например, вы внесли изменения в два файла и хотите сохранить их как два отдельных коммита, но случайно набрали `git add *` и проиндексировали оба файла. Чтобы отменить индексацию одного из них, нужно использовать команду `git reset HEAD`:

```
$ git reset HEAD kern/pmap.c
kern/pmap.c: locally modified
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified: README
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified: kern/pmap.c
```

### 3.2.13 Отмена внесенных изменений

Чтобы отменить все внесенные изменения и вернуть файл в состояние, в котором он был после последнего коммита, используется команда `git checkout -- <имя файла>`:

```
$ git checkout -- kern/pmap.c
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified: README.txt
```

Как видите, изменения были отменены. Эта команда является довольно опасной: все изменения в файле пропадают, поверх него появляется другой файл.

### 3.2.14 Ветвление

Ветка в Git — это легковесный подвижный указатель на один из коммитов. В новом репозитории по умолчанию создается ветка `master`, которая изначально указывает на последний сделанный коммит. При каждом новом коммите она сдвигается вперед автоматически.

Если создать новую ветку, будет создан новый указатель, который можно будет перемещать. Для примера создадим новую ветку под названием `testing`. Это делается командой `git branch`:

```
$ git branch testing
```

Эта команда создаст новый указатель на текущий коммит. Git хранит специальный указатель, который называется `HEAD` (верхушка) и указывает на локальную ветку, на которой в данный момент находится репозиторий. Чтобы пе-

рейти на уже существующую ветку, используется команда `git checkout`. Например, чтобы перейти на ветку `testing`, нужно выполнить ее следующим образом:

```
$ git checkout testing
```

Можно также создать ветку и сразу перейти на нее, используя команду `checkout` с параметром `-b`:

```
$ git checkout -b testing
```

Теперь можно внести изменения и сделать коммит:

```
$ vim kern/pmap.c
```

```
$ git commit -a -m 'pmap.c changed'
```

Ветка `testing` будет указывать на него, но ветка `master` будет указывать на предыдущий коммит. Можно перейти обратно на ветку `master`:

```
$ git checkout master
```

При этом указатель `HEAD` передвинется назад на ветку `master`, а файлы в рабочем каталоге вернутся в состояние, на которое указывает `master`. Кроме того, дальнейшие изменения будут отвечаяться от старой версии проекта.

Теперь можно еще раз внести изменения в файл и сделать коммит:

```
$ vim kern/pmap.c
```

```
$ git commit -a -m 'pmap.c changed one more time'
```

При этом история проекта разветвится. Оба совершенных изменения изолированы в отдельных ветках; можно переключаться между ними, а также слить их. При слиянии изменения из одной ветки вносятся в другую. Слияние выполняется с помощью команды `git merge`:

```
$ git checkout master
```

```
$ git merge testing
```

```
Auto-merging kern/pmap.c
Merge made by the 'recursive' strategy.
kern/pmap.c | 2 ++
1 file changed, 1 insertion(+), 1 deletion(-)
```

Теперь в ветке master присутствуют изменения из ветки testing, которую можно удалить следующим образом

```
$ git branch -d testing
Deleted branch testing (454863e).
```

Команда git branch также позволяет просмотреть все ветки в репозитории:

```
$ git branch
lab9
* master
testing
```

Символ \*, стоящий перед веткой master, означает, что эта ветка является текущей.

Просмотреть последний коммит в каждой из веток можно с помощью параметра -v:

```
$ git branch -v
lab9  93b412c Lab 9 added.
* master 7a98805 Merge branch 'testing'
      testing 782fd34 pmap.c changed
```

### 3.2.15 Конфликты при слиянии

Если одна и та же часть файла изменена в двух разных ветках, их автоматическое слияние невозможно. При попытке слияния возникнет конфликт:

```
$ git merge testing
Auto-merging kern/pmap.c
```

CONFLICT (content): Merge conflict in kern/pmap.c

Automatic merge failed; fix conflicts and then commit the result.

Git приостановил слияние до тех пор, пока конфликт не будет разрешен. Чтобы посмотреть, какие файлы не прошли слияние, используется команда git status:

```
$ git status
kern/pmap.c: needs merge
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       # unmerged: kern/pmap.c
```

Git добавляет в измененные места файлов, которые имеют конфликт, оба варианта изменений, так что вы можете открыть их вручную и разрешить эти конфликты. Файл kern/pmap.c теперь содержит секцию, которая выглядит примерно так:

```
<<<<< HEAD
pages = boot_alloc(pages_number);
=====
pages = test_alloc(test_number);
>>>>> testing
```

В верхней части (выше разделителя =====) находится версия из HEAD (т. е. из ветки master), в нижней – версия из testing. Чтобы разрешить конфликт, нужно выбрать одну из этих частей или каким-либо способом объединить их. После этого нужно выполнить git add для каждого конфликтного

файла. Теперь можно выполнить git status еще раз, чтобы убедиться, что все конфликты были разрешены:

```
$ git status  
# On branch master  
# Changes to be committed:  
#   (use "git reset HEAD <file>..." to unstage)  
#  
#   modified: kern/pmap.c
```

После этого нужно выполнить git commit для завершения слияния. По умолчанию сообщение будет выглядеть следующим образом:

```
Merge branch 'testing' into master  
Conflicts:  
kern/pmap.c
```

При необходимости это сообщение можно дополнить информацией о том, как был разрешен конфликт.

### 3.2.16 Stash

Иногда возникает необходимость переключиться на другую ветку, при этом изменения в текущей ветке не являются законченными и фиксировать их не имеет смысла. В этом случае используется команда git stash. Она сохраняет измененные отслеживаемые файлы и изменения в индексе в стек незавершенных изменений, которые затем можно снова применить. Например, вы работаете над проектом и внесли некоторые изменения:

```
$ git status  
# On branch master  
# Changes to be committed:  
#   (use "git reset HEAD <file>..." to unstage)
```

```
#  
# modified: README  
#  
# Changes not staged for commit:  
# (use "git add <file>..." to update what will be committed)  
#  
# modified: kern/pmap.c
```

Теперь, чтобы поменять ветку или просто изучить файлы в их состоянии на момент последнего коммита, можно выполнить `git stash`:

```
$ git stash  
Saved working directory and index state WIP on lab11: 0222bc9 Lab 11  
added.  
HEAD is now at 0222bc9 Lab 11 added.
```

Рабочий каталог чист:

```
$ git status  
# On branch master  
nothing to commit, working directory clean
```

Чтобы посмотреть сохраненные изменения, необходимо использовать команду `git stash list`:

```
$ git stash list  
stash@{0}: WIP on lab11: 0222bc9 Lab 11 added.  
stash@{1}: WIP on lab9: 851f322 Merge branch 'lab8' into lab9
```

Можно снова применить сохраненные изменения с помощью команды `git stash apply`. Команде можно передать имя более старых изменений, например: `git stash apply stash@{1}`; иначе будут применены последние сохраненные изменения. Чтобы удалить сохраненные изменения, выполните `git stash drop` с их именем, например, `git stash drop stash@{0}`. Сочетанием этих двух команд является команда `git`

`stash pop`, которая применяет изменения и сразу же удаляет их из стека. Эта команда используется чаще всего.

### 3.2.17 Работа с удаленными репозиториями

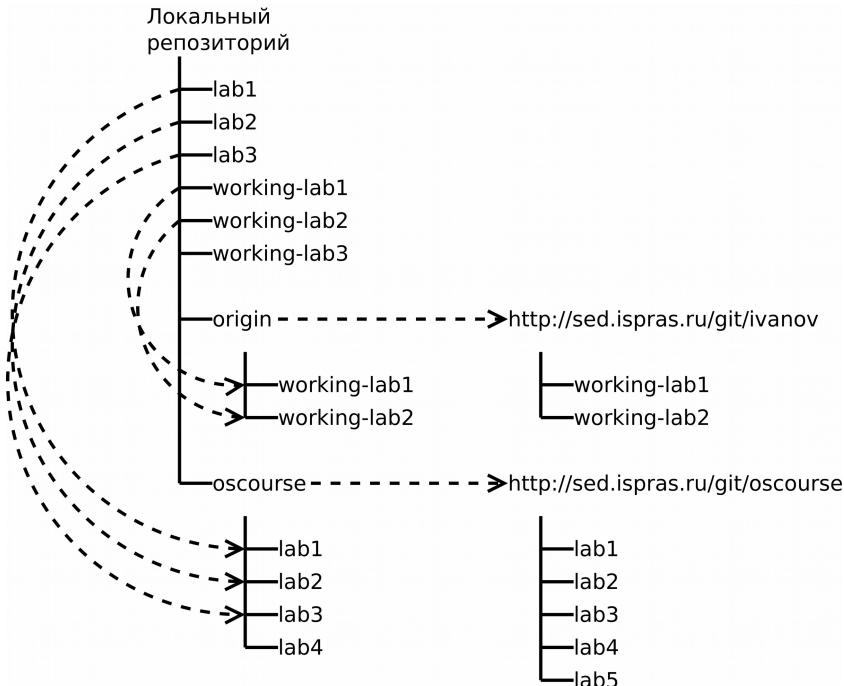


Рисунок 3.2.1: Работа с удаленными репозиториями

Удаленные репозитории — это модификации репозитория, которые хранятся на удаленном сервере. Их может быть несколько, каждый из них может быть доступен либо на чтение, либо на чтение и запись. Работа с ними включает в себя отправку (`push`) и получение (`pull`) данных.

Чтобы просмотреть, какие удаленные серверы уже настроены для репозитория, следует выполнить команду `git remote`. Если репозиторий был клонирован с удаленного сервера

ра, должен отобразиться по крайней мере origin – это имя по умолчанию, которое Git присваивает серверу, с которого репозиторий клонирован:

```
$ git clone http://sed.ispras.ru/git/oscourse  
(...)  
$ cd oscourse  
$ git remote  
origin
```

Чтобы посмотреть, какому URL соответствует имя, можно указать команде опцию -v:

```
$ git remote -v  
origin http://sed.ispras.ru/git/oscourse (fetch)  
origin http://sed.ispras.ru/git/oscourse (push)
```

Если удаленных репозиториев больше одного, команда покажет их все:

```
$ git remote -v  
origin http://sed.ispras.ru/git/ivanov (fetch)  
origin http://sed.ispras.ru/git/ivanov (push)  
oscourse      http://sed.ispras.ru/git/oscourse (fetch)  
oscourse      http://sed.ispras.ru/git/oscourse (push)
```

На рисунке 3.2.1 изображен локальный репозиторий, для которого настроено два удаленных репозитория.

### 3.2.18 Добавление и удаление удаленных репозиториев

Чтобы добавить новый удаленный репозиторий, нужно выполнить git remote add [имя] [url]:

```
$ git remote  
origin  
$ git remote add oscourse http://sed.ispras.ru/git/oscourse
```

```
$ git remote -v
origin http://sed.ispras.ru/git/ivanov (fetch)
origin http://sed.ispras.ru/git/ivanov (push)
oscourse      http://sed.ispras.ru/git/oscourse (fetch)
oscourse      http://sed.ispras.ru/git/oscourse (push)
```

Для удаления удаленного репозитория используется команда `git remote rm [имя]`.

### 3.2.19 Получение данных из удаленного репозитория

Чтобы извлечь (fetch) всю информацию, которая присутствует в удаленном репозитории `oscourse`, но отсутствует в локальном репозитории, можно выполнить `git fetch oscourse`:

```
$ git fetch oscourse
remote: Counting objects: 376, done.
remote: Compressing objects: 100% (372/372), done.
remote: Total 376 (delta 153), reused 0 (delta 0)
Receiving objects: 100% (376/376), 922.43 KiB | 0 bytes/s, done.
Resolving deltas: 100% (153/153), done.
From http://sed.ispras.ru/git/oscourse
 * [new branch] lab5    -> oscourse/lab5
```

Ветка `lab5` теперь доступна локально как `oscourse/lab5`. Команда `fetch` связывается с удаленным репозиторием и извлекает из него все данные, которых в локальном репозитории еще нет.

Для ветки, настроенной на отслеживание удаленной ветки, можно использовать команду `git pull`. Для текущей локальной ветки она извлекает данные из соответствующей удаленной ветки и выполняет их слияние с ней. Команда `git clone` при клонировании автоматически настраивает локальную ветку

master на отслеживание удаленной ветки master на сервере. Выполнение git pull, как правило, извлекает (fetch) данные с сервера, с которого репозиторий был изначально клонирован (origin), и автоматически пытается выполнить их слияние (merge) их с кодом в локальном репозитории.

### **3.2.20 Отправка изменений в удаленный репозиторий**

Чтобы отправить изменения, которые были зафиксированы, в удаленный репозиторий, используется команда git push. Например, чтобы отправить ветку working-lab3 из локального репозитория на сервер origin, команду нужно выполнить следующим образом:

```
$ git push origin working-lab3
```

Если текущая локальная ветка уже настроена на отслеживание соответствующей удаленной ветки, команду можно выполнить в более простом виде:

```
$ git push
```

### **3.2.21 Инспекция удаленного репозитория**

Чтобы получить больше информации об удаленном репозитории, можно использовать команду git remote show, например:

```
$ git remote show oscourse
* remote oscourse
  URL: http://sed.ispras.ru/git/oscourse
  Remote branch merged with 'git pull' while on branch master
    lab1
  Tracked remote branches
```

```
lab1  
lab2  
lab3  
lab4
```

Она выдает URL удаленного репозитория, а также информацию об отслеживаемых ветках; кроме того, команда сообщает, что, если выполнить при активной ветке `git pull`, ветка `master` с удаленного сервера будет автоматически слита с локальной.

### 3.2.22 Удаленные ветки

Локальным веткам могут соответствовать удаленные ветки — ссылки на состояние веток в удаленных репозиториях. Они выглядят как `<имя репозитория>/<ветка>`. Например, чтобы проверить, как выглядела ветка `master` на сервере `origin` во время последнего соединения с ним, можно просмотреть ветку `origin/master`. При выполнении `git fetch origin` эта ветка будет обновлена.

Получение локальной ветки из удаленной с помощью `git checkout` автоматически создает т. н. отслеживаемую ветку. Отслеживаемая ветка — это локальная ветка, связанная с удаленной. Для такой ветки `git pull` и `git push` будут работать автоматически, не требуя указания удаленного репозитория. При клонировании репозитория, как правило, автоматически создается ветка `master`, которая отслеживает `origin/master`, поэтому `git push` и `git pull` работают для этой ветки сразу. Однако, вы можете настроить отслеживание и других веток удаленного репозитория. Простейший пример — выполнить `git checkout`, передав в качестве параметра название ветки:

```
$ git checkout lab3
```

```
Branch lab3 set up to track remote branch lab3 from origin.
```

```
Switched to a new branch 'lab3'
```

Git автоматически найдет соответствующую удаленную ветку. Чтобы создать локальную ветку с именем, отличным от имени удаленной ветки, можно указать это имя следующим образом:

```
$ git checkout -b l3 origin/lab3
```

```
Branch l3 set up to track remote branch lab3 from origin.
```

```
Switched to a new branch 'lab3'
```

При необходимости можно удалить ветку на удаленном сервере следующим образом:

```
$ git push origin :working-lab3
```

### 3.3 Организация исходного кода JOS

<b>./boot</b>	<b>исходный код загрузчика</b>
boot.S	первая часть загрузчика, которая переводит процессор в защищенный режим, устанавливает указатель стека и передает управление второй части загрузчика
main.c	вторая часть загрузчика, которая загружает ядро с диска в память и запускает его
Makefrag	фрагмент Makefile для сборки загрузчика
<hr/>	
<b>./conf</b>	<b>конфигурационные файлы с переменными для сборки</b>
<hr/>	
<b>./doc</b>	<b>вспомогательная документация</b>
<hr/>	
<b>./fs</b>	<b>исходный код драйвера файловой системы и ее содержимое</b>
bc.c	реализация кэша блоков
fs.c	реализация файловой системы
fs.h	
fsformat.c	исходный код утилиты, создающей образ диска
ide.c	исходный код IDE-драйвера, основанного на PIO (программном вводе-выводе; без использования прерываний)
serv.c	исходный код сервера файловой системы, взаимодействующего с клиентами посредством

	IPC
test.c	программа для тестирования работы файловой системы
testshell.sh	скрипт, предназначенный для тестирования работы файловой системы
testshell.key	ожидаемый вывод при работе testshell.sh
lorem	текстовые файлы, используемые для тестирования работы файловой системы
motd	
newmotd	
script	
Makefrag	фрагмент Makefile для сборки образа файловой системы
./inc	<b>заголовочные файлы, общие для ядра и пользовательских программ</b>
args.h	функции и структуры, необходимые для обработки аргументов командной строки
assert.h	функции, вызываемые при неправильной работе ядра
elf.h	структуры и константы, необходимые для поддержки двоичных исполняемых файлов в формате ELF
env.h	структуры, необходимые для управления процессами
error.h	коды ошибок ядра
fd.h	функции и структуры, необходимые для поддержки файловых дескрипторов

fs.h	структуры и константы, необходимые для поддержки файловой системы
kbdreg.h	константы, необходимые для работы с клавиатурой (коды клавиш и т.д.)
lib.h	библиотека для пользовательских программ: системные вызовы, вспомогательные функции, поддержка IPC и т.д.
memlayout.h	структуры и константы, необходимые для управления памятью
mmu.h	определения, необходимые для работы с блоком управления памятью (MMU)
partition.h	структуры и константы, необходимые для поддержки дисковых разделов
random.h	функции для генерации случайных чисел
stab.h	структуры и константы, необходимые для обработки отладочной информации в формате STABS
stdarg.h	функции для поддержки переменного числа аргументов в функциях
stdio.h	функции для работы со стандартным вводом-выводом
string.h	функции для работы со строками
syscall.h	номера системных вызовов
trap.h	определения для поддержки прерываний
types.h	определения некоторых простых типов (например, «32-битное целое число») и вспомогательные макросы
x86.h	макросы для вызова напрямую некоторых инструкций процессора без использования

	ассемблерных вставок
<b>./kern</b>	<b>исходный код и заголовочные файлы ядра</b>
alloc.c	поддержка выделения памяти для программ в режиме ядра
alloc.h	
console.c	поддержка консольного ввода-вывода
console.h	
cpl.h	структуры и функции для работы с прерываниями
entry.S	точка входа в ядро
entrypgdir.c	таблица страниц, отображающая первые 4 мегабайта физической памяти на первые 4 мегабайта виртуальной
env.c	работа с пользовательскими процессами
env.h	
init.c	инициализация ядра
kclock.c	поддержка работы с часами
kclock.h	
kdebug.c	поддержка работы с отладочной информацией в формате STABS
kdebug.h	
kernel.ld	скрипт компоновки ядра
monitor.c	монитор ядра
monitor.h	
picirq.c	поддержка работы с часами PIC
picirq.h	
pmap.c	управление памятью

pmap.h	
printf.c	поддержка форматированного вывода
sched.c	планировщик
sched.h	
spinlock.c	поддержка примитивов циклической блокировки (спинлоков)
spinlock.h	
syscall.c	поддержка системных вызовов
syscall.h	
trap.c	поддержка обработки прерываний
trap.h	
trapentry.S	точка входа в прерывания
tsc.c	поддержка таймера
tsc.h	
Makefrag	фрагмент Makefile для сборки ядра
./lib	<b>реализация стандартной библиотеки для пользовательских программ</b>
args.c	обработка аргументов командной строки
console.c	консольный ввод-вывод
entry.S	точка входа для пользовательской программы
exit.c	завершение пользовательской программы
fd.c	работа с файловыми дескрипторами
file.c	работа с файлами

fork.c	системный вызов fork()
fprintf.c	форматированный вывод для файлов
ipc.c	межпроцессное взаимодействие
libmain.c	функция main()
pageref.c	получение количества ссылок на страницу памяти
panic.c	реализация паники для пользовательских программ
pfentry.S	точка входа для вызова ядром пользовательского обработчика ошибки страницы
pgfault.c	поддержка пользовательских обработчиков ошибки страницы
pipe.c	поддержка перенаправления ввода-вывода
printf.c	форматированный ввод-вывод
printfmt.c	форматирование строк в стиле printf
random.c	генерация случайных чисел
readline.c	поддержка консольного ввода
spawn.c	функция spawn()
string.c	работа со строками
syscall.c	поддержка системных вызовов
wait.c	ожидание окончания работы процесса
Makefrag	фрагмент Makefile для сборки стандартной библиотеки для пользовательских программ
./obj	<b>результат сборки загрузчика, ядра и пользовательских программ</b>
boot/boo	дизассемблированный загрузчик

t.asm	
kern/kern el.asm	дизассемблированное ядро
./prog	<b>тестовые программы, работающие в режиме ядра</b>
./user	<b>тестовые программы, работающие в пользовательском режиме</b>

## 3.4 Способы отладки

### 3.4.1 GDB

В данном разделе описаны некоторые команды отладчика GDB, полезные для отладки ядра. В качестве более полного руководства можно использовать его документацию [16].

Для запуска JOS с GDB нужно открыть два окна терминала, в обоих перейти в директорию с кодом лабораторных работ и в одном из них выполнить make qemu-gdb (или make qemu-run-X-gdb для запуска одной из программ), а в другом просто запустить gdb.

- **Ctrl+C**  
Данное сочетание клавиш останавливает работу JOS и переходит в GDB.
- **c** или **continue**  
Продолжает выполнение с текущего места до следующей точки останова или до нажатия Ctrl+C.
- **si** или **stepi**  
Выполняет одну инструкцию.
- **b <название функции>** или **b <файл:строка>** (или **breakpoint**)  
Устанавливает точку останова на заданную функцию или строку.
- **b \*<адрес>** (или **breakpoint**)  
Устанавливает точку останова на заданный адрес.
- **set print pretty**  
Включает красивую печать массивов и структур.
- **info registers**  
Выводит значения регистров общего назначения, EIP, EFLAGS и сегментных регистров. Также можно ис-

пользовать аналогичную команду QEMU (раздел 3.4.2).

- **x/Nx <адрес>**  
Выводит N слов, начиная с заданного виртуального адреса. Если N не задано, выводит одно слово. Вместо адреса можно использовать выражение.
- **x/Ni <адрес>**  
Выводит N ассемблерных инструкций, начиная с заданного адреса. Если вместо адреса задать \$eip, будут выведены инструкции, начиная с текущей. Вместо адреса можно также использовать выражение.
- **symbol-file <файл>**  
Переключается на заданный файл символов. GDB не различает процессы и ядро, поэтому при отладке процессов, а не ядра, рекомендуется выполнить эту команду, чтобы задать файл с отладочными символами. Например, если выполняется процесс hello.c, нужно выполнить команду symbol-file obj/user/hello.

### 3.4.2 Монитор QEMU

В QEMU встроен монитор, который позволяет просматривать состояние виртуальной машины и управлять им. Для входа в монитор используется комбинация клавиш Ctrl+a с в окне терминала; для выхода и возврата в консоль JOS используется та же комбинация клавиш.

Подробное руководство по монитору QEMU имеется в разд. 3.5 его руководства [12]. Ниже приведены отдельные полезные команды.

- **xp/Nx paddr**  
Выводит в шестнадцатеричном виде N слов, начиная с физического адреса paddr. Если N пропущено, оно считается равным 1. Данная команда аналогична команде GDB x.

- **info registers**

Выводит состояние регистров процессора машины, включая скрытую часть сегментных регистров, например:

```
CS =0008 10000000 ffffffff 10cf9a00 DPL=0 CS32 [-R-]
```

где:

CS =0008 – видимая часть сегментного регистра; данная информация также говорит о том, что используется глобальная таблица дескрипторов ( $0x0008 \& 4 = 0$ ) и уровень привилегий (CPL) равен  $0x0008 \& 3 = 0$ .

10000000 – база текущего сегмента. Таким образом, линейный адрес равен логическому адресу + 0x10000000.

fffffff – лимит текущего сегмента. Обращение к линейному адресу больше 0xFFFFFFFF вызовет ошибку.

10cf9a00 – флаги сегмента в «сыром» виде, приведенные в расшифрованном виде далее.

DPL=0 – уровень привилегий сегмента. Только код с уровнем привилегий 0 может загружать этот сегмент.

CS32 – 32-битный сегмент кода. Другие возможные значения: DS – сегмент данных, LDT – локальная таблица дескрипторов.

[-R-] – сегмент только для чтения.

- **info mem**

Выводит отображенную виртуальную память и ее разрешения, например:

```
ef7c0000-ef800000 00040000 urw
```

```
efbf8000-efc00000 00008000 -rw
```

0x00040000 байт с 0xEF7C0000 по 0xEF800000 отображены для чтения и записи и доступны в пользовательском режиме, а 0x00008000 байт с 0xEF800000

по 0xEFC00000 отображены для чтения и записи, но только для ядра.

- **info pg**

Данная команда присутствует только в исправленной версии QEMU. Отображает текущую таблицу страниц. Вывод похож на info mem, но различает таблицы страниц и каталоги страниц, выводя для них разрешения отдельно. Идущие подряд записи объединяются в одну строку. Например:

VPN range	Entry	Flags	Physical page
[00000-003ff]	PDE[000]	-----UWP	
[00200-00233]	PTE[200-233]	-----U-P	00380 0037e 0037d
		0037c ..	
[00800-00bff]	PDE[002]	----A--UWP	
[00800-00801]	PTE[000-001]	---A--U-P	0034b 00349
[00802-00802]	PTE[002]	-----U-P	00348

Можно видеть две записи каталога страниц для виртуальных адресов с 0x00000000 по 0x003FFFFF и с 0x00800000 по 0x00BFFFFFF. У обеих записей установлены флаги present, writable, user, а у второй также accessed. Вторая запись отображает три страницы от 0x00800000 до 0x00802FFF, у первых двух установлены флаги present, user и accessed, а у третьей только present и user. Первая из записей соответствует физической странице 0x34B.

Кроме этого, QEMU принимает некоторые полезные аргументы командной строки, которые можно передать через переменную QEMUEXTRA при запуске JOS:

make QEMUEXTRA='-d int' ...
-----------------------------

Параметр «-d int» заставляет QEMU сохранять в qemu.log все прерывания вместе с дампом регистров. Первые

две записи — «SMM: enter» и «SMM: after RMS» — генерируются перед входом в загрузчик. После них записи выглядят следующим образом:

```
4: v=30 e=0000 i=1 cpl=3 IP=001b:00800e2e pc=00800e2e
SP=0023:eebfdf28
EAX=00000005 EBX=00001002 ECX=00200000 EDX=00000000
ESI=00000805 EDI=00200000 EBP=eebfdf60 ESP=eebfdf28
...
...
```

Первая строка описывает прерывание. «4:» — счетчик записей в логе, v — вектор прерывания в шестнадцатеричном виде, e — код ошибки, i=1 говорит о том, что прерывание сгенерировано инструкцией int, а не аппаратурой, cpl — уровень привилегий и т.д.

## **3.5 Некоторые возможные проблемы и подходы к их решению**

Данный раздел создан на основе опыта поиска и исправления ошибок в работах студентов за прошлые годы. В разделе приведены лишь некоторые типичные ошибки; рассмотреть все возможные проблемы не представляется возможным: даже в рамках заданий исходный код обычно является индивидуальным, несмотря на наличие общей кодовой базы, что оставляет возможность возникновения бесконечного количества сложно выявляемых проблем. Тем не менее, во многих случаях приведенная ниже информация может быть полезной.

### **3.5.1 Планировщик**

**Проблема:** работает только один процесс, либо выполняются два процесса попеременно, игнорируя остальные.

**Работы:** 3, 4.

Характер ошибки говорит о том, что она связана с планировщиком. Соответственно, ошибка может быть либо в коде планировщика, либо в структурах данных, которыми он оперирует. Первым делом необходимо проверить функцию `sched_yield()`, в которой должен быть правильным образом реализован алгоритм Round-Robin. Просмотр списка процессов должен начинаться с последнего выполнявшегося. Кроме того, сам список процессов должен быть корректно инициализирован в функции `env_init()`.

**Проблема:** прерывания от часов не приходят.

**Работы:** 4.

В случае такой ошибки одна из программ будет выполняться в бесконечном цикле, а другие не будут выполняться вовсе. Для решения проблемы необходимо:

- Проверить правильность инициализации часов.
- Проверить, что на контроллере прерываний размаскирована линия, по которой должны приходить прерывания от часов. Лучше всего разместить инструкции для размаскирования линии сразу после инициализации часов.
  - Проверить, что контроллеру и часам поступает сигнал окончания прерывания (EOI - End Of Interrupt). Это должно происходить в обработчике прерывания после начальной обработки прерывания, но до вызова планировщика.

### 3.5.2 Процессы и память

**Проблема:** процессы не выполняются; ОС выдает ошибки доступа к памяти; Triple Fault с неправдоподобным значением EIP; данные по адресам, где расположены процессы, не совпадают с данными в файлах obj/prog/\*.asm.

**Работы:** 3.

В функции load\_icode для копирования байт старайтесь использовать стандартные функции memcpу, memmove, memset.

**Проблема:** происходит Triple Fault в функции env\_pop\_tf, когда она вызывается второй раз.

**Работы:** 3.

Неправильная реализация функции env\_alloc; проверьте ее еще раз.

**Проблема:** глобальные переменные в момент старта программы имеют значения, отличные от 0; код исполняется непредсказуемым образом; значения ячеек памяти равны 0x97.

### **Работы:** 6-11.

Где-то не очищается память, например, не вызывается memset(). Проверьте реализацию функций fork() и clone(), а также функций для работы с памятью процессов.

**Проблема:** При загрузке операционной системы возникает ошибка до вывода каких-либо данных.

### **Работы:** 6-11.

Попробуйте уменьшить количество процессов в папках prog/ и user/. Исключите из компиляции лишние.

### **3.5.3 Другие проблемы**

**Проблема:** Возникает Triple Fault.

### **Работы:** 2-11.

Для того, чтобы выяснить причины подобного поведения программы, можно:

1. Просмотреть описание Triple Fault. Наибольший интерес представляет регистр EIP; если он выглядит корректно, можно найти соответствующий адрес в файлах \*.asm в директории obj.
2. Выполнить команду ‘make QEMUEXTRA="-d int" qemu’, которая выводит в файл qemu.log все возникающие прерывания. Обратите внимание на два исключения перед Triple Fault. В поле v указан вектор прерывания. В следующем за ним поле для некоторых исключений содержится информация о коде ошибки.
3. В результате предыдущих действий вы получили информацию о типе ошибки и о месте ее возникновения. Теперь с помощью GDB и/или отладочной печати можно попытаться найти проблему и исправить ее. Иногда полезно отключать часть функциональности ядра (например, выключить часть процессов), чтобы выделить факторы, при которых проявляется ошибка, и в целом упростить процесс

отладки.

4. Чтобы просмотреть изменения, в результате которых могла возникнуть ошибка, можно использовать команду `git diff`. Если ошибка появилась до последнего коммита, можно просмотреть и предыдущие, выполнив `git log -p`. Кроме того, бывает полезно посмотреть, в каком коммите была внесена ошибка; для этого можно использовать команду `git blame <имя файла>`.

## **4 Список литературы**

1. Л.Н. Королев. Структуры ЭВМ и их математическое обеспечение // Наука, 1974.
2. Н.В. Вдовикова, И.В. Машечкин, А.Н. Терехин, А.Н. Томилин. Операционные системы: взаимодействие процессов // МАКС Пресс Москва, 2008.
3. В.Г. Баула, А.Н. Томилин, Д.Ю. Волканов. Архитектура ЭВМ и операционные среды // М. Academia, Серия: Высшее профессиональное образование. Бакалавриат ISBN 978-5-7695-9286-7; 2012 г.
4. В.А. Крюков, В.А. Бахтин. Распределенные системы // М. Academia, Серия: Высшее профессиональное образование. Бакалавриат ISBN 978-5-7695-9286-7; 2012 г.
5. Operating System Engineering. Frans Kaashoek, Robert Morris. Massachusetts Institute of Technology.  
<http://pdos.csail.mit.edu/6.828>
6. Paul A. Carter. PC Assembly Language.  
<http://sed.ispras.ru/fmprac/data/pcasm-book.pdf>
7. M48T86 5.0V PC Real-Time Clock Datasheet  
<http://www.farnell.com/datasheets/1499223.pdf>
8. Intel® 64 and IA-32 Architectures Software Developer Manuals, combined volumes 3A, 3B, 3C, and 3D: System programming guide  
<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>
9. Modular Scheduler Core and Completely Fair Scheduler  
<http://lwn.net/Articles/230501/>
10. Э. Таненбаум, Х. Бос. Современные операционные системы. 4-е издание. СПб., Питер, 2015.
11. Э. Таненбаум, Т. Остин. Архитектура компьютера. 6-е издание. СПб, Питер, 2015.

12. QEMU Emulator User Documentation  
<http://wiki.qemu.org/download/qemu-doc.html>
13. Scott Chacon, Ben Straub. Pro Git. Apress, 2nd Edition, 2014.  
<https://git-scm.com/book/en/v2>
14. Git Documentation  
<https://git-scm.com/documentation>
15. Bourne-Again Shell Manual  
<http://www.gnu.org/software/bash/manual/>
16. Debugging with GDB – Sourceware  
<https://sourceware.org/gdb/onlinedocs/gdb/>