Крищенко В. А.

ОСНОВЫ РЕАЛИЗАЦИИ ОПЕРАЦИОННЫХ СИСТЕМ

Учебное пособие

Черновик от 16 декабря 2013 г.*

На примере учебной операционной системы рассмотрены основные вопросы создания ядра операционной системы, начиная от его загрузки и заканчивая реализацией вытесняющей многозадачности и межпроцессного взаимодействия. В ходе практических занятий студентам необходимо понять функционирование ядра ОС и написать недостающий исходный код.

Для студентов магистратуры кафедры «Программное обеспечение ЭВМ и информационные технологии» (ИУ7) МГТУ им. Н. Э. Баумана, обучающихся по направлению 231000 «Программная инженерия».

*С небольшими исправлениями и дополнениями Келарева И. А. от 11 января 2016 г.

Содержание

Обозначения и сокращения

ОС — операционная система.

ЦП — центральный процессор.

BIOS — встроенное в ПЗУ программное обеспечение для инициализации и доступа к аппаратуре компьютера архитектуры x86 (англ. *Basic Input-Output System*).

CPL — уровень привилегий исполняемого в данный момент времени кода (англ. *Current Privilege Level*).

DPL — уровень привилегий дескриптора (англ. *Descriptor Privilege Level*).

EOI — сигнал о завершении обработки прерывания, посылаемый контроллеру прерываний (англ. *End Of Interupt*).

GDT — глобальная таблица дескрипторов сегментов (англ. *Global Descriptor Table*).

IDT — таблица дескрипторов обработчиков прерываний (англ. *Interrupt Descriptor Table*).

IPC — механизм обмена данными между двумя исполняемыми программами (англ. *Inter-process Communications*).

IRQ — аппаратное прерывание (англ. *Interrupt Request*).

JOS — учебная операционная система Josh Operating System.

MMU — блок управления памятью в составе ЦП (англ. *Memory Managment Unit*).

PAE — режим работы MMU процессора x86, использующий 64-битные PDE и PTE при 32-битном линейном адресе (англ. *Physical Address Extension*).

PDE — элемент каталога страниц (англ. *Page Directory Entry*).

PTE — элемент таблицы страниц (англ. *Page Table Entry*).

SMP — многопроцессорная архитектура с симметричным доступом к памяти (англ. *Symmetric Multiprocessing*).

TSS — сегмент состояния задачи — специальная структура данных архитектуры x86 (англ. *Task State Segment*).

x86 — совокупное название архитектур компьютеров с 32-битными процессорами Intel i386, i486 и более старшими и совместимыми с ними, а также с работающими в 32-битном режиме совместимыми 64-битными процессорами.

Введение

Для изучения основ функционирования операционных систем недостаточно изучения одного теоретического материала: для понимания работы ядра ОС нужно изучить и его исходные коды. Хотя в настоящее время ядра многих эксплуатируемых ОС доступны для такого изучения, все они как возможный методический материал для учебного процесса обладают двумя недостатками: высокой сложностью и большим объемом исходных текстов. По этой причине коллективом сотрудников университета MIT была разработана учебная операционная система, позже названная JOS[1], предназначенная для практической части курса разработки операционных систем.

Помимо своей компактности, JOS как учебная система обладает ещё одним существенным достоинством: студент работает со все более усложняющимися вариантами системы, постепенно знакомясь с теми или иными функциями ядра ОС. Таким образом, термином «JOS» обозначен скорее сам процесс разработки системы, а не некая завершённая ОС. Исходные тексты JOS имеют лицензию MIT или BSD, т. е. она является свободным программным обеспечением и может использоваться с минимальными ограничениями.

Данное пособие предназначено для проведения лабораторных работ по курсу разработки операционных систем и построено следующим образом. Первая глава пособия содержит краткие сведения о работе процессоров x86 и организации в них виртуальной памяти и обработки прерываний. Вторая глава посвящена обзору исходных текстов JOS и используемых диалектов языков программирования. Главы с третьей по десятую содержат описание практических заданий по разработке операционной системы на базе имеющихся исходных текстов JOS.

Глава 3 охватывает инициализацию системы от работы BIOS до передачи управления ядру ОС. Глава 4 рассматривает начальную инициализацию ядра ОС, управление страничным преобразованием адресов и выделение страниц физической памяти. В главе 5 в системе появляется возможность выполнять прикладную программу в режиме минимальных привилегий. Глава 6 посвящена обработке прерываний и созданию механизма системных вызовов для взаимодействия пользовательской программы с ядром.

Последующие главы добавляют поддержку многозадачности в ядро разрабатываемой системы. Глава 7 добавляет в систему простейшее переключение процессов и кооперативную многозадачность. В главе 8 реализуется эффективное клонирование процессов на основе копирования при записи. В главе 9 в системе появляется принудительная вытесняющая многоза-

[1] JOS была названа в честь одного из своих первоначальных авторов, Joshua Cates, после его гибели в ДТП.

дачность. Наконец, в главе 10 создается механизм обмена данными между двумя процессами.

В конце всех глав приводятся контрольные вопросы для проверки усвоенных студентом знаний. Главы с третьей по десятую содержат задания по написанию исходного кода, которого не хватает для полноценной работы разрабатываемой операционной системы.

Пособие предполагает наличие у читателя предварительного знакомства с:

– основами теории операционных систем (в пределах [1]);
– системой команд процессора x86 (например, по [2]);
– языком программирования C (в пределах [3] или [4]);
– UNIX-подобными системами (например, по [5]);
– основами дискретной математики, типами и структурами данных.

Для выполнения практических заданий пособия читателю понадобится подключенный к интернету компьютер с POSIX-системой, имеющей:

– компилятор GNU C Compiler, ассемблер GNU и программы компоновки GNU Binutils, способные создать исполняемый файл формата ELF для процессора i486;
– утилиту для сборки программных проектов GNU Make;
– клиент системы контроля версий Git;
– эмулятор Bochs с включённым встроенным отладчиком;
– произвольный текстовый редактор (или среду разработки программ на языке C) для работы с исходными текстами JOS.

Указания по использованию в качестве такой системы Debian или Ubuntu можно найти на страничке пособия[1].

Автор хотел бы поблагодарить Келарева Ивана Андреевича, Короткова Ивана Адреевича, Крючкова Алексея Олеговича, Федотовскую Екатерину Викторовну и Горина Сергея Викторовича за замечания, найденные ошибки, советы и дополнения.

[1] Страничка пособия: **http://dev.iu7.bmstu.ru/trac/workbook_jos**.

1 Основы архитектуры x86

Операционная система JOS, создание которой рассматривается далее в этом пособии, отвечает прежде всего за разделение между выполняющимися программами аппаратных ресурсов ЭВМ: физической памяти, процессорного времени, устройства вывода. Для её работы необходим компьютер с процессором i486 или совместимым. Этот компьютер должен быть организован по спецификациям, восходящим к компьютерам IBM PC/AT и далее называемым «архитектурой x86». В этой главе приведены краткие сведения об устройстве такого компьютера с точки зрения системного программиста. За более подробными сведениями можно обратиться к документации Intel Corporation [6, 7].

Хотя разрабатываемая операционная система будет в основном написана на языке программирования C и содержит не так много ассемблерного кода, она слишком просто организована, чтобы быть хорошо абстрагированной от аппаратной платформы. В силу этого она довольно тяжело переносима на аппаратные платформы, отличные от x86. При выполнении практических заданий в нашем распоряжении будет набор готовых макросов, функций и структур, упрощающих взаимодействие с железом, тем не менее хорошее понимание происходящего с точки зрения аппаратуры всё равно является критичным для разработки ядра JOS. Излишнее абстрагирование от аппаратных возможностей в любом случае нельзя считать удачным для задачи разработки учебной ОС, в основном занимающейся разделением аппаратных ресурсов между программами.

1.1 Регистры центрального процессора

Все регистры процессора i486SX (SX — серия без математического сопроцессора) можно разделить на несколько групп, такие как: общего назначения, адреса точки выполнения и вершины стека, регистр флагов, управляющие регистры.

Регистры общего назначения — 32-битные регистры **eax**, **ebx**, **ecx**, **edx**, **esi**, **edi** — используются для хранения данных и смещений адресов. Регистры **esi** (англ. *Source Index*) и **edi** (англ. *Destination Index*) используются также особым образом как смещения при операциях копирования и сравнения массивов чисел.

Пара из регистра сегмента кода **cs** (англ. *Code Segment*) и регистра указателя команды **eip** (англ. *Instruction Pointer*) указывает на текущую выполняемую команду и обозначается как **cs:eip**. Значения этих регистров меняются самим процессором при выполнении любых инструкций. Обычно регистр указателя команды смещается к следующей команде, но инструк-

ции перехода, вызова функции, возврата из функции и ряд других меняет эту пару регистров более сложным образом.

Регистры сегмента стека **ss** (англ. *Stack Segment*) и смещения вершины стека **esp** (англ. *Stack Pointer*) используются при работе со стеком: пара **ss:esp** указывает на вершину аппаратного стека — участка памяти, используемого для передачи аргументов функциям, хранения локальных переменных и адресов возврата из функций. Регистр базового адреса стека **ebp** (англ. *Base Pointer*) также связан с использованием стека: в начале каждой функции часто помещается код, благодаря которому пара **ss:ebp** указывает на вершину стека в момент входа в текущую функцию и относительно этой пары адресуются аргументы и локальные переменные функции. Отметим, что с аппаратным стеком можно при необходимости работать и любым другим образом, скажем, используя смещение в регистре общего назначения — пара **ss:esp** лишь используется при выполнении ряда машинных инструкций (**push**, **pop**, **call**, **ret** и др.).

Регистры сегмента данных **ds** (англ. *Data Segment*) и дополнительного сегмента данных **es** (англ. *Extra Segment*) используются для доступа к данным вне стека. Регистр **es** при этом используется при всех так называемых строковых машинных операциях, когда процессор заполняет, сравнивает или копирует массивы одной повторяющейся инструкцией. Существуют ещё два сегментных регистра для доступа к данным (**fs** и **gs**), но они используются только при их явном указании в машинной инструкции, и по этой причине наша система не будет их использовать.

Полный адрес в x86 всегда образуется парой из сегмента и смещения, но последнее может быть получено разными способами, зависящими от режима работы ЦП. Все сегментные регистры (**cs**, **ss**, **ds**, **es**) — 16-битные. Регистры смещений **ebp** и **esp** — 32-битные, как и регистры общего назначения. Далее рассмотрены регистры, которые не используются для хранения адресов данных, инструкций или самих данных, а имеют некоторое специальное назначение.

Управляющий регистр флагов и статусов **eflags** хранит битовую маску как статусов (например, результат последнего сравнения), так и флагов, управляющих рядом операций. В частности, девятый бит этого регистра разрешает или запрещает аппаратные прерывания. Это странное решение вызвано тем, что на оригинальном процессоре 8086 не было никакой поддержки для нескольких уровней привилегий и ограничения доступа к памяти и аппаратуре: любая программа могла и сравнивать числа, и запрещать прерывания. Уже в процессоре 80286 ситуация изменилась, и не все биты данного регистра можно изменять в любой момент времени.

Все рассмотренные выше регистры могут меняться в ходе выполнения прикладной программы, хотя конкретно в нашей системе программы не бу-

дут менять значения сегментных регистров. По этой причине операционная система должна позаботиться о сохранении их значений в момент временного прекращения работы программы и восстановления их перед продолжением её выполнения. Наша система не будет сохранять и восстанавливать какие-либо иные регистры, в частности, регистры математического сопроцессора и регистры, появившиеся в процессорах Pentium и более поздних (регистры MMX, SSE и т. д.). В силу этого код прикладных программ в нашей системе должен быть скомпилирован для процессора не старше, чем Pentium I, и не может содержать операций с плавающей точкой.

Описанные далее в этом разделе регистры, наоборот, не меняются прикладными программами — более того, они даже не должны иметь такой возможности, поскольку это позволило бы им нарушить работу системы. Часть этих регистров содержат адреса управляющих таблиц и структур данных, которые используются процессором в своей работе и будут рассмотрены далее в этой главе.

Управляющие регистры **cr0**–**cr4** имеют специальное назначение. Нам будет важен регистр **cr0**, управляющий режимом работы процессора и включающий страничное преобразование, регистр **cr3**, указывающий на таблицу страничного преобразования адресов, и регистр **cr2**, хранящий адрес, обращение по которому вызвало страничное исключение. Заметьте, что эти регистры имеют вполне определенное постоянное назначение, хотя и не имеют осмысленных названий.

Процессор имеет ещё три особых управляющих регистра — регистр **gdtr** для хранения информации о размере и адресе таблицы GDT, регистр **idtr** для хранения адреса таблицы IDT и регистр **tr** для хранения адреса структуры TSS (эта структура данных, как мы увидим позже, не является таблицей).

1.2 Логический, линейный и физический адреса

До начала рассказа о сегментно-страничном преобразовании адресов, которое используется на платформе x86, напомним ряд определений.

Виртуальной памятью называется некоторая абстрактная адресуемая память, не совпадающая с физической, но отображаемая в нее некоторым образом на уровне аппаратной части ЭВМ. Адрес ячейки виртуальной памяти называется *виртуальным адресом*.

Ядро ОС должно указывать аппаратной части, как виртуальные адреса следует преобразовывать в физические. Для решения задачи разделения оперативной памяти между исполняющимися программами разумно предоставить каждой из них свое отображение виртуальных адресов в физические.

Виртуальным адресным пространством называется множество виртуальных адресов одной исполняемой программы, частично отображенное на множество физических адресов.

Процесс — исполняющаяся программа, имеющая своё собственное виртуальное адресное пространство и свой контекст выполнения, включающий значения регистров ЦП.

Размер виртуального адресного пространства определяется числом бит в виртуальном адресе. Виртуальный адрес в программах разрабатываемой операционной системы содержит 32 бита, что даёт размер виртуального адресного пространства, равный 2^{32} = 4 Гб. Формально в архитектуре x86 полный виртуальный адрес состоит из 32-битного смещения и 16-битного сегмента, причём два бита сегмента выделены для хранения информации о правах. Однако, разработчики современных ОС по описанным далее причинам используют сегментную часть адреса таким образом, что она не оказывает никакого влияния на размер виртуального пространства, поэтому такие системы считают 32-битными.

Отметим, что в домашнем обиходе «виртуальной памятью» часто ошибочно называют область подкачки на диске, используемую для временного хранения данных, выгруженных из физической памяти. При разработке операционных систем нам, конечно же, понадобится правильная терминология.

Поскольку архитектура x86 прошла сложный путь развития от 16-битного процессора Intel 8086 до полноценной платформы для многозадачных операционных систем, то в ней существует и реальный режим, оперирующий не более чем 1 Мб памяти, и сегментное преобразование, появившееся в процессоре Intel 80286, и, наконец, страничное преобразование памяти, появившееся в процессоре Intel 80386 (называемым также i386). Для двух последних режимов в состав процессора добавлен блок MMU, который управляет преобразованием виртуальных адресов в физические.

В *реальном режиме* x86 нет каких-либо управляющих трансляцией адресов таблиц и поэтому нет и самих виртуальных адресов, а преобразование задающей адрес пары «сегмент:смещение» (оба — 16 бит) в 20-битный физический адрес происходит так: адрес = 16 * сегмент + смещение (умножение на 16 эквивалентно сдвигу сегмента на 4 разряда влево). В реальном режиме процессору доступен для адресации только первый мегабайт памяти: хотя по приведённой формуле получается чуть большее значение, аппаратно физический адрес на процессоре 8086 был ограничен 20-ю битами. Компьютер x86 начинает свою работу именно в этом режиме.

Полноценная виртуальная память в архитектуре x86 появляется в так называемом *защищённом режиме*, который впервые появился в процессоре 80286, где работал только с сегментным преобразованием адресов.

В процессоре i386 появилось страничное преобразование и он принял вид, не претерпевший существенных изменений до появления процессоров архитектуры AMD64 с поддержкой 64-битного адреса (которые, тем не менее, могут работать и в 32-битном режиме, совместимом с i386). Защищённый режим позволяет не только организовать виртуальную память, но и вводит уровни привилегий исполняемого кода для защиты операционной системы и аппаратной части от некорректных действий прикладных программ. В частности, все изменения управляющих регистров ЦП и операции разрешения и запрещения прерываний допустимы только для кода, исполняемого на наивысшем уровне привилегий. При использовании защищённого режима код прикладных программ будут исполнятся в типичных операционных системах с наименьшими привилегиями.

Ядром операционной системы мы далее будем называть совокупность кода, исполняющегося на наивысшем (с точки зрения процессора) уровне привилегий и тех используемых этим кодом данных, которые недоступны прикладным программам. В основном в пособии далее рассматриваются вопросы создания именно ядра ОС. Отметим, что некоторые авторы и производители ПО дают и иные определения ядра ОС, но разрабатываемое ядро соответствует большинству из них.

Для перехода в защищённый режим, в котором процессор использует 32-битное смещение, необходимо сначала задать *сегментное преобразование адресов*, используя таблицу глобальных дескрипторов сегментов GDT. Эта таблица размещена в физической памяти, а адрес структуры, хранящей информацию о размере и адресе GDT, записывается в специальный регистр **gdtr** ЦП. При сегментом преобразовании значение сегментного регистра трактуется как смещение в таблице дескрипторов и называется *селектором*. Из элемента таблицы дескрипторов извлекается базовый адрес, который складывается со смещением исходного адреса, давая результат сегментного преобразования.

Активное использование сегментного преобразования операционной системой привело бы к ряду проблем. Все указатели стали бы 48-битными, а перед каждым обращением к памяти, вообще говоря, нужно было бы записать новое значение в сегментный регистр, что является весьма длительной операцией: именно в этот момент будет прочитана строка из таблицы дескрипторов (она кешируются в так называемой «теневой» части сегментного регистра, недоступной программно). В итоге интенсивное использование сегментов привело бы к многократному падению производительности. Заметной проблемой стала бы и сложность переноса такой системы на аппаратные архитектуры без сегментного преобразования (то есть фактически на все, отличные от x86).

Кроме глобальной таблицы дескрипторов, в архитектуре x86 могут существовать и локальные таблицы, связанные с отдельным процессом. Один из битов регистра селектора (второй, если считать с нулевого как самого младшего) используется для указания того, следует ли использовать глобальную или локальную таблицу дескрипторов сегментов. Локальные таблицы нам далее совершенно не понадобятся, поскольку являются пережитками процессора 80286

Сегментное преобразование на платформе x86 существует всегда с момента переключения в защищённый режим, а без его настройки нельзя обратиться к памяти за пределами одного мегабайта. Сегментное преобразование может быть вырожденным (селектор:смещение $\rightarrow$ смещение), если в дескрипторе сегмента базовым адресом указан ноль, а его концом — адрес 0xFFFFFFFF.

В архитектуре x86 *логическим адресом* называется адрес до сегментного преобразования, включающий в себя сегментный регистр (селектор) и смещение. Этот адрес является, очевидно, виртуальным, поскольку не совпадает с физическим. Результат сегментного преобразования является физическим адресом, если только не включено страничное преобразование адресов. При включённом страничном преобразовании он называется *линейным адресом*, который также является виртуальным и рассмотрен в следующем разделе. Размер и физического, и линейного адреса составляет 32 бита.

Отметим, что фирма Intel хотя и являлась пионером разработки ЦП, но сравнительно поздно представила свой процессор i386: к этому моменту на рынке уже были 32-битные процессоры с поддержкой уровней привилегий и страничного преобразования адресов, причём без наследия сегментного преобразования. Часть использованных в этом разделе терминов (реальный режим, защищённый режим, логический адрес и др.), относятся сугубо к терминологии фирмы Intel.

1.3 Страничное преобразование адресов

Страничное преобразование, с точки зрения процессора x86, не обязательно, но является при этом ключевым для современных операционных систем, включая и JOS. Наша операционная система будет активно использовать страничное преобразование, а сегментное преобразование будет использовать в минимально возможном объёме и в основном в вырожденном виде.

Если сегментное преобразование вырождено и для организации виртуальной памяти используется только страничное преобразование, то такая организация памяти называется *плоской*. В большинстве операционных

систем на платформе x86 используется именно она. Отметим, что для 64-битной архитектуры AMD64 плоская организация является единственно возможной, а на большинстве отличных от x86 современных архитектур сегментного преобразования нет вообще.

При включенном страничном преобразовании результат сегментного преобразования в x86 называют *линейным адресом*, поскольку в нём уже нет понятия селектора. Этот адрес, формально, также является виртуальным, поскольку является входным значением для страничного преобразования адресов. Страничное преобразование полностью прозрачно для прикладной программы, которая в случае плоской модели памяти просто оперирует 32-битными смещениями.

Страничное преобразование задаётся таблично. Для экономного использования памяти таблицы страничного преобразования организованы в два уровня. На рисунке 1.1 показано страничное преобразование адресов при размере страницы 4 Кб. Как видно по нему, физический адрес таблицы первого уровня — каталога страниц (англ. *page directory*) — загружается в регистр **cr3** ЦП. Адрес каталога страниц должен быть выровнен на границу страницы, и по этой причине старшие 20 бит регистра **cr3** называют регистром базы директории страниц (**pdbr**).

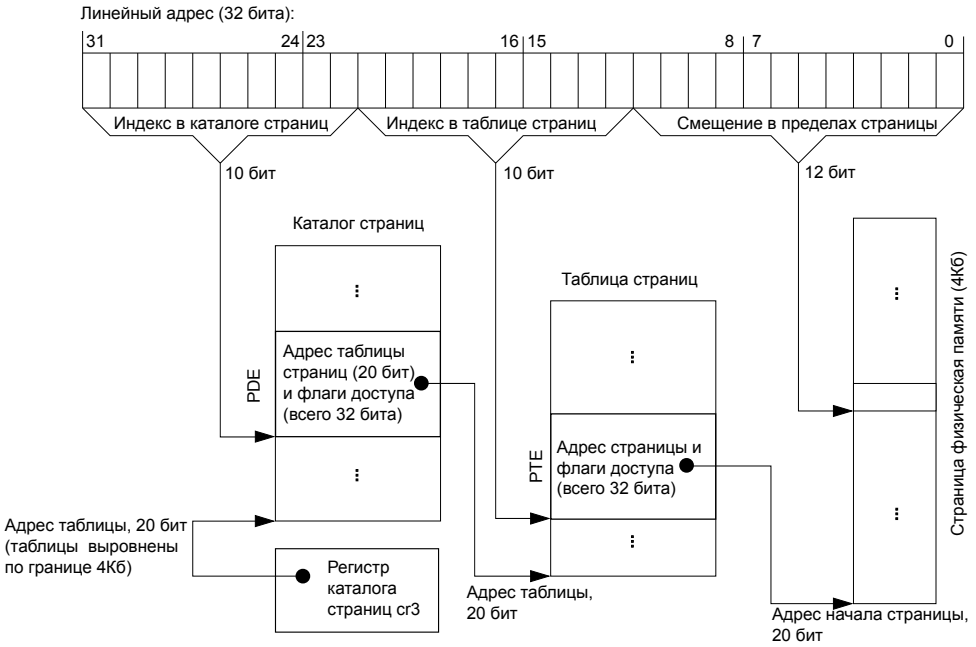

Рисунок 1.1 — Преобразование линейного адреса в физический

В элементах каталога страниц и таблиц страниц находятся 20-битные физические адреса (старшие биты), флаг присутствия страницы P, флаги доступа, которые будут рассмотрены ниже, и некоторые другие. Размер элемента — 32 бита. Адрес в элементе каталога страниц указывает на таблицу

страниц, а адрес в таблице страниц указывает на искомую физическую страницу. Таблицы страниц также должны быть выровнены в памяти на границу 20 бит. Как можно убедится, размер таблиц обоих уровней равен ровно одной странице в случае 32-битного адреса и размера страницы, равного 4 Кб. Это несколько упрощает дальнейшую реализацию ядра ОС.

Отметим, что в большинстве, отличных от x86 32-битных процессоров с поддержкой страничной памяти используется аналогичная схема страничного преобразования с двумя уровнями таблиц. Конкретные названия этих таблиц (в случае x86 — директория страниц и таблица страниц) не являются стандартизованными и зависят от производителя процессоров.

В нашей ОС мы пойдём традиционным для систем на базе x86 путём: к моменту запуска прикладных программ таблица GDT будет задавать вырожденное сегментное преобразование, а за виртуальное адресное пространство процессов будет отвечать страничное преобразование. Поэтому мы будем называть далее виртуальным адресом смещение логического адреса и оно будет равно линейному адресу к концу главы 4 и в последующих главах. При смене исполняемого процесса наша система должна будет менять значение регистра **cr3** и переходить таким образом на другое виртуальное адресное пространство.

Как видно по рисунку 1.1, страничное преобразование адресов использует два обращения к страницам трансляции на каждый транслируемый адрес. Чтобы избежать трёхкратного падения производительности при включённом сегментом преобразовании, необходимо кешировать результат преобразования адреса виртуальной страницы в физическую страницу.

В составе MMU имеется специальный кеш TLB (англ. *Translation Lookaside Buffer*), служащий как раз для этой цели: он кеширует информацию, полученную из таблиц страниц. Ключём в этом кеше является адрес виртуальной страницы (старшие 20 бит линейного адреса), а выходом — адреса физической страницы (старшие 20 бит физичесого адреса) и флаги доступа к ней. Содержимое этого кеша стирается при записи в регистр **cr3** (т. е. смене адресного пространства). Новые записи добавляются в него автоматически процессором (точнее, блоком MMU в составе ЦП). Тем не менее, операционной системе необходимо явным образом очищать этот кеш от устаревших вхождений, т. е. обеспечивать его когерентность при изменении содержимого таблиц страниц.

Для удаления из кеша TLB элемента, связанного с указанным линейным адресом, в процессоре i486 и более старших существует инструкция **invlpg**. Операционной системе не нужно проверять наличие устаревшего адреса в кеше, ей следует просто выполнить указанную инструкцию после любой смены содержимого таблицы страниц или каталога страниц, кроме

случая появления в них действующей ссылки (с установленным флагом присутствия) вместо отсутствующей ссылки.

Более подробно об организации сегментного и страничного преобразования можно прочитать, например, в [6], главы 5 и 6.

1.4 Ограничение доступа к памяти

Сегментное преобразование адресов в x86 поддерживает четыре уровня привилегий со значениями 0–3, причём нулевое значение соответствует самому высокому уровню привилегий. В дескрипторе сегмента хранится значение уровня привилегий DPL, необходимого для доступа к дескриптору, и права доступа в виде битовой маски флагов, разрешающих или запрещающих чтение, запись и исполнение. Текущий уровень привилегий (CPL) хранится в двух младших битах регистра **cs**. При доступе к дескриптору проверяется условие DPL $\geqslant$ CPL, если оно не выполняется — происходит исключение ошибки защиты памяти.

На практике из четырёх уровней привилегий обычно используется нулевой уровень, на котором работает ядро ОС, и третий, на котором работают все прикладные программы. Поэтому в нашем случае режим ядра является синонимом нулевого уровня привилегий, режим пользователя — синонимом третьего. Отметим, что с точки зрения привилегий кода все системные службы также являются прикладными программами: речь здесь идёт о противопоставлении ядра и всех прочих программ, называемых также пользовательскими.

В таблице GDT при плоской модели памяти в типичной ОС будет всего шесть элементов:

– нулевой дескриптор, первый по счёту, для обнаружения ошибочной попытки доступа по нулевому селектору;
– дескрипторы кода и данных режима ядра (DPL = 0) с соответствующими разрешениями;
– дескрипторы кода и данных режима пользователя (DPL = 3) с соответствующими разрешениями;
– дескриптор структуры TSS (см. раздел 1.9).

В страничном преобразовании в x86 за защиту памяти отвечают два бита в элементах каталога страниц и таблицы страниц: R/W (Read/Write) и U/S (User/Supervisor). Если бит R/W равен единице, то в данную страницу может производиться запись. Если бит U/S равен нулю, то к странице может производиться обращение только при CPL со значениями 0–2, а при единичном значении бита он возможен и при CPL = 3. Это даёт следующие комбинации прав доступа «ядро/пользователь»: R/R, RW/RW, R/-, RW/-.

Как видно, в архитектуре x86 на уровне страничного преобразования уровни привилегий 0, 1 и 2 рассматриваются как один уровень — «режим ядра». Это даёт основание считать, что к моменту разработки процессора i386 разработчики компании Intel поняли практическую ненужность четырёх уровней привилегий, введённых в 80286 на уровне сегментного преобразования.

Процессор i486 не имеет возможности запретить исполнение данных на уровне страничного преобразования, поскольку для этого не было выделено бита в элементе таблиц страниц. Поэтому при использовании плоской модели памяти любая страница памяти, которую процессор может читать, может и исполняться как код. Это открывает для злоумышленников широкие возможности при наличии на атакуемом компьютере некорректных программ, имеющих проблему переполнение буфера[1]. Для решения этой проблемы в итоге был добавлен бит запрещения записи, но только для 64-битной архитектуры AMD64 для процессоров Opteron (AMD) и Pentium 4 (Intel) и старше, работающих в режиме PAE[2], когда элемент таблиц страницы имеет размер 64 бита.

Для показанного на рисунке 1.1 режима с 32-битными элементами таблицы страниц защита содержимого стека от исполнения для любого процессора архитектуры x86 возможна только путём установки границы сегмента кода пользователя так, чтобы он заканчивался до начала его стека. Этот подход на практике используется, насколько известно автору, только в ОС OpenBSD.

1.5 Организация физической памяти

В физической памяти компьютера архитектуры x86 есть две области, зарезервированные для взаимодействия с устройствами и размещения BIOS: участок от адреса 0x000A0000 до адреса 0x000FFFFF и участок ниже адреса 0xFFFFFFFF, размер которого зависит от присутствующих устройств (рисунок 1.2).

Вся остальная наличествующая в системе оперативная память может быть использована в нашей ОС, но при этом только участок выше адреса 0x00100000 является непрерывной областью. Напомним, что в реальном режиме процессора, доставшемся в наследство от процессора 8086, программы не могут адресовать память выше адреса 0x00100000.

[1] Выход за границу локального массива, расположенного на стеке, затирает адрес возврата из функции и может быть использован для передачи управления на содержимое этого же массива.

[2] В режиме PAE (англ. *Physical Address Extension*) используется 32-битный виртуальный адрес и 52-битный физический адрес, см. [7].

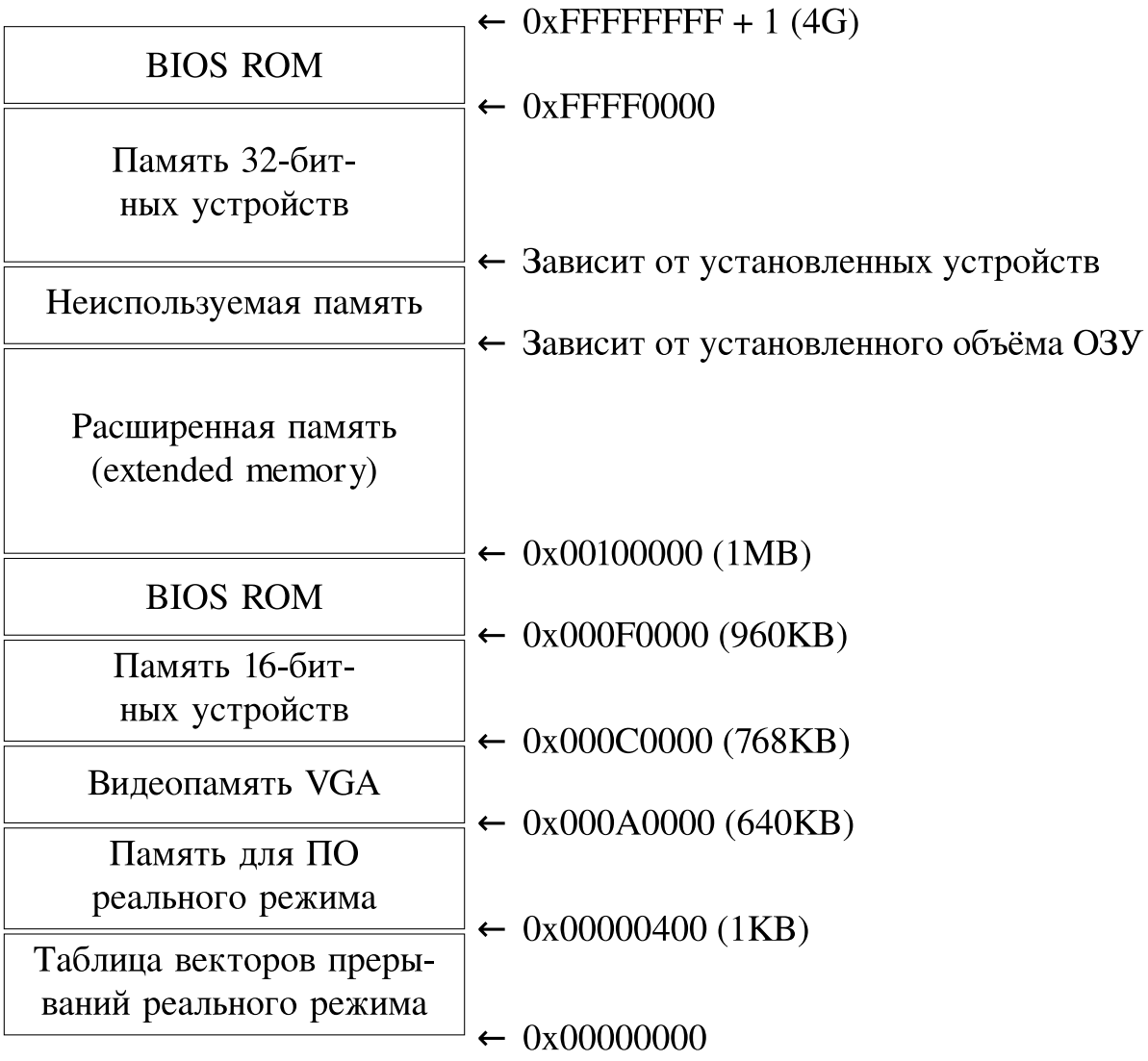

Рисунок 1.2 — Назначение областей физической памяти в ЭВМ архитектуры x86

В начале физической памяти находится особая, с точки зрения процессора, область — таблица прерываний реального режима, позволяющая вызывать прерывания BIOS и ОС реального режима (DOS). При работе процессора в защищённом режиме эта таблица совершенно бесполезна.

1.6 История адресной линии №20

Наша ОС будет работать в защищённом режиме с включённым страничным преобразованием, а её ядро будет располагаться в физической памяти начиная с адреса 1 Мб. С доступом к физической памяти свыше одного мегабайта в архитектуре x86 связана анекдотическая история, которую придётся учесть загрузчику нашей системы.

Дело в том, что схема преобразования реального режима, как можно убедиться, теоретически позволяет адресовать память за пределами 1 Мб, и объём этой области — 64 Кб. На процессоре 8086 физическая шина адреса имела 20 линий (с номерами A0–A19), что делало эту возможность исключительно гипотетической — двадцатый (считая с нулевого) бит физического адреса отбрасывался, и процессор попадал в младшие физические адреса. На процессоре 80286 линия адреса A20 появилась, и описанная возможность стала реальностью, чем и воспользовались разработчики системного ПО реального режима.

Однако, фирма Intel сочла эту ситуацию своей ошибкой: одна и та же пара сегмента и смещения приводила к обращению к разным физическим адресам на разных поколениях процессоров, что было сочтено грубым нарушением собственного обещания полной обратной совместимости между процессорами 8086 и 80286. Во имя обеспечения последней в компьютеры на процессоре 80286, известные как IBM PC/AT, было добавлено «исправление»: появился регулятор, известный как «вентиль линии A20», который в своём закрытом положении всегда сбрасывал злополучный бит адреса, обеспечивая таким образом полную обратную совместимость, но не допуская обращений к памяти выше 1 Мб. Вентиль мог быть открыт (и обычно открывался) и в реальном режиме, если желание получить прибавку в 10 процентов к адресуемой памяти было важнее полной обратной совместимости.

Вентиль линии A20 является внешним (по отношению к ЦП) устройством и мог в те времена управляться пользователем непосредственно через настройки BIOS. По умолчанию вентиль считается закрытым и может быть открыт программно, причём как минимум тремя способами. Самый первый из этих способов уже давно устарел, поскольку требует наличия старого контроллера AT-клавиатуры (Intel 8042). Второй способ использовал программный интерфейс BIOS (но не работал со всеми существующими BIOS), а третий появился на замену первому, но требует контроллера интерфейса PS/2 и стал стандартом позже появления машин с процессором i486. Наша система будет использовать первый способ, поскольку он точно соответствует возможностям используемого эмулятора x86. Современные операционные системы должны либо использовать третий, либо считать, что вентиль уже открыт, что верно для современных 64-битных процессоров.

Случай с линией A20 примечателен как полная победа требования обратной совместимости над здравым смыслом: в итоге не существует гарантированного способа открыть этот вентиль, который бы работал на всех машинах архитектуры x86. Особую анекдотичность ситуации придаёт тот факт, что программы, на которых бы сказалась эта потеря совместимости, так и не были обнаружены.

1.7 Начальная загрузка компьютера

Процесс начальной загрузки ОС в ЭВМ архитектуры x86 не претерпел принципиальных изменений за тридцать лет, прошедших с момента появления первых IBM PC до начала внедрения загрузки на основе интерфейса EFI вместо использования BIOS. Загрузка системы начинается с того, что после включения компьютера управление передаётся на реальный адрес FFFF:0000, где находится ПЗУ с кодом BIOS.

После того, как BIOS инициализирует устройства, вектора прерываний реального режима и выберет загрузочный дисковод или жёсткий диск, он загружает его первый сектор (512 байт) по адресу 0x7C00 и затем передаёт управление на него (реальный адрес 0000:7C00). Два последних байта сектора должны иметь значения 0x55 и 0xAA, иначе BIOS не сочтёт сектор загрузочным. Таким образом, размер начальной части загрузчика ОС фактически не может превышать 510 байт.

Поскольку многие современные загрузчики считывают файл ядра ОС как обычный файл файловой системы, то они уже не могут разместиться в одном секторе и используют сложную многоэтапную загрузку. В нашей же системе загрузчик предполагает, что ядро находится в фиксированных секторах диска, и поэтому загрузчик умещается в единственном секторе.

В момент передачи управления загрузчику от BIOS процессор находится в реальном режиме работы. Это означает невозможность простой работы с памятью свыше одного мегабайта, но даёт возможность использовать прерывания BIOS. С точки зрения нашего загрузчика указанный недостаток перевешивает возможные достоинства, поэтому он почти сразу перейдёт в защищённый режим.

1.8 Использование стека при вызове функций

В программах на языке C для архитектуры x86 регистры **esp** и **ebp** обычно имеют специальное назначение. Указатель стека **esp** указывает на текущую точку разделения между свободной и используемой областями стека. В архитектуре i386, как и на большинстве других процессоров, стек «растёт вниз» — указатель стека уменьшается при увеличении размера используемой области стека. В каждый момент времени указатель стека указывает на первый используемый байт стека, а всё, что расположено ниже, считается свободным. Добавление данных в стек инструкцией **push** уменьшает значение указателя стека, а извлечение инструкцией **pop** — увеличивает. Некоторые инструкции процессора неявно используют регистр указателя стека. Инструкция вызова функции **call** помещает в стек адрес возврата (адрес команды, следующей после самой инструкцией **call**), а инструкция

возврата из функции **ret** извлекает адрес возврата из стека и передаёт управление на него.

Отметим, что хотя аргументы функции передаются через стек, это не значит, что компилятор помещает их туда, используя инструкцию **push**. Вместо этого он просто смещает указатель стека «вниз» на нужное число байтов при помощи инструкции **sub**, а потом копирует в стек аргументы обычной инструкцией **mov**. «Очистка» стека от аргументов функции сводится к сдвигу указателя стека «вверх» инструкцией **add**. Аналогично выделяется место в стеке и для локальных переменных функции. Отметим, что в 32-битных системах положение аргументов функции в стеке выравнивается на границу четырёх байт, а инструкции **push** и **pop** не могут работать со значениями иных размеров.

Регистр **ebp** связан со стеком прежде всего типичными соглашениями о вызовах в языке C. На входе в функцию специальный код пролога обычно сохраняет этот регистр, помещая его в стек, а затем копирует текущее значение регистра **esp** в регистр **ebp**. Типичный код пролога в нотации ассемблера UNIX показан ниже, вторая инструкция здесь копирует содержимое регистра **esp** в регистр **ebp**.

```
push %ebp
movl %esp, %ebp
```

После этих команд регистр **ebp** будет указывать на предыдущее значение регистра **ebp**, сохранённое в стеке. Выше этого значения в стеке будет лежать адрес возврата, а перед ним — аргументы функции (рисунок 1.3). Эта структура, соответствующая одному вызову функции, называется *кадром стека* (англ. *stack frame*).

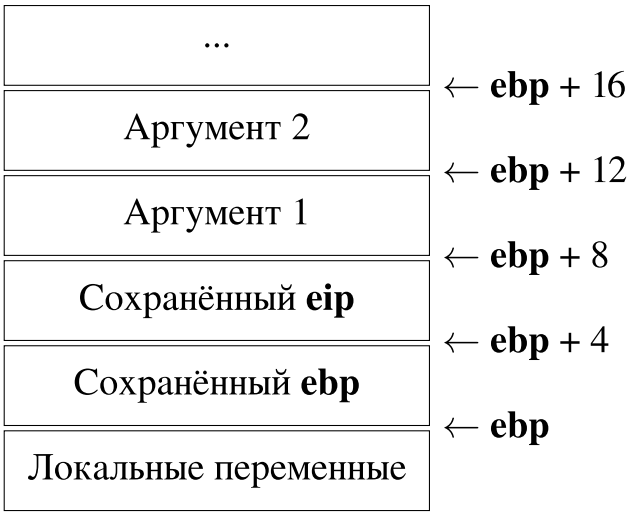

Рисунок 1.3 — Кадр стека

Таким образом, если все функции программы следуют этому соглашению, то в любой момент во время выполнения программы возможно отследить назад через стек цепочку сохранённых указателей **ebp** и, соответственно, это позволяет точно определить, какая последовательность вложенных вызовов функций привела к достижению данного места программы.

1.9 Защищенная передача управления

В пособии мы будем в целом следовать терминологии фирмы Intel, касающейся прерываний и исключений. Следует иметь в ввиду, что такие термины, как исключения (англ. *exceptions*), ловушки (англ. *traps*), прерывания (англ. *interrupts*), отказы (англ. *faults*) и аварийные завершения (англ. *aborts*) не имеют стандартизованного значения для различных архитектур и операционных систем и часто используются весьма приблизительно без особого внимания к тонким различиям между ними на конкретной архитектуре.

Исключения и прерывания являются способами защищенной передачи управления. Защищённая передача управления позволяет процессору переключаться из пользовательского режима в режим ядра, при этом не давая возможности пользовательскому коду перейти в произвольное место в ядре. В терминологии компании Intel *прерывание* — это защищённая передача управления, вызываемая асинхронным событием, обычно внешним для процессора. Примером такого события может быть уведомление о завершении операции ввода-вывода некоторым устройством. *Исключение* — это, напротив, способ защищенной передачи управления, вызываемый выполняющимся в настоящий момент кодом, например, делением на ноль или доступом к памяти с нарушением прав. В отличие от прерываний исключения являются синхронными относительно процесса выполнения кода.

Для обработки прерываний и исключений процессор должен перейти в привилегированный режим и перейти к исполнению кода ядра ОС. Механизм обработки прерываний и исключений процессора разработан так, чтобы выполняющийся в момент этого события код не мог сам решать, как следует передать управление в код ядра. Процессор гарантирует, что передача управления при повышении привилегий может быть осуществлена только так, как это было указано ядром ОС. В архитектуре x86 это обеспечивается при помощи двух структур: таблицы дескрипторов прерываний и сегмента состояния задачи.

Таблица дескрипторов прерываний. Процессор гарантирует, что прерывания и исключения могут вызывать вход в ядро только через несколько известных точек входа, которые задаются ядром и которые не

зависят от кода, который выполняется в момент получения прерывания или исключения.

В частности, прерывания и исключения x86 делятся на 256 возможных типов, каждый из которых ассоциируется с некоторым номером прерывания (часто называемым номером исключения или ловушки). Когда процессор идентифицирует конкретное прерывание или исключение, полученное им, он использует номер прерывания как индекс в таблице дескрипторов прерываний (IDT) — специальной таблице, похожей на GDT, которое ядро создает в своей области памяти. Инструкция **lidt** загружает линейный адрес этой таблицы в регистр **idtr**.

Из найденной записи этой таблицы процессор получает значение, загружаемое в сегментный регистр кода **cs**, которое содержит также в битах 0–1 уровень привилегий, с которыми должен выполняться обработчик прерывания или исключения, а также значение, загружаемое в регистр указателя инструкции **eip**. Пара **cs:eip** указывает на начало обработчика.

В таблице IDT могут присутствовать записи трёх разных типов, но далее мы опишем только один, наиболее типичный из них, который и будет использовать наша система. В терминологии Intel она называется *шлюзом прерывания* (англ. *Interrupt Gate*) и будет использоваться нами как для обработчиков прерываний, так и для обработчиков исключений. Существует важная для нашей ОС разница между шлюзом прерывания и шлюзами остальных двух видов: в момент начала работы обработчика, установленного через шлюз прерывания, аппаратные прерывания будут запрещены, в то время как записи иных видов разрешают такие прерывания в момент начала работы обработчика.

Сегмент состояния задачи (TSS). В дополнение к заранее определенной точке входа в ядро для обработки прерывания или исключения процессор также требует место для сохранения своего старого состояния перед тем, как случится прерывание или исключение. Это позволит сохранить, в частности, значения регистров **cs** и **eip** перед выполнением обработчика исключения. Таким образом, обработчик позднее может восстановить старое состояние и продолжить выполнение кода с того места, где он был прерван.

Область для хранения старого состояния процессора должна, в свою очередь, тоже быть защищена от непривилегированного пользовательского кода, иначе некорректный или вредоносный пользовательский код может легко поставить работу ядра под угрозу. Поэтому, когда процессор попадает на прерывание или в ловушку, что вызывает изменение уровня привилегий от пользовательского режима до режима ядра, он не только загружает новые значения в регистры **eip** и **cs**, но также загружает и новые значения в регистр указателя стека **esp** и сегмента стека **ss**, переключаясь на новый

стек, находящийся в области ядра. Затем процессор помещает в этот новый стек оригинальные значения всех указанных регистров, наряду с содержимым регистра **eflags**, прежде чем начать выполнение кода обработчика исключения ядра.

Новые значения регистров **esp** и **ss** находятся не в таблице IDT, а берутся из отдельной структуры, называемой *сегментом состояния задачи* (TSS). Ниже показана часть её полей, цифра в конце имени поля означает, что при переходе в момент прерывания на указанный уровень привилегий с более низкого уровня будут взяты именно эти значения для нового стека.

```
struct Taskstate {
    ...
    uint32_t esp0; uint16_t ss0;
    ...
    uint32_t esp1; uint16_t ss1;
    ...
    uint32_t esp2; uint16_t ss2;
    ...
};
```

Адрес структуры TSS должен быть помещён в свой собственный элемент таблицы GDT, а значение селектора этого элемента должно быть загружено инструкцией **ltr**.

1.10 Использование стека при прерываниях

Допустим, что процессор выполняет код в режиме пользователя и сталкивается с командой деления, которая пытается разделить на ноль, наш обработчик имеет селектор с нулевым уровнем привилегий, а вершина стека обработчика исключения обозначена как **kstacktop**. Сразу после обнаружения ошибки деления происходят следующие действия.

1) Процессор генерирует исключение с номером 0 (этот номер соответствует обнаруженной ошибке).

2) Процессор обращается к нулевому элементу таблицы IDT и обнаруживает, что хранящееся там значение селектора **cs** имеет нулевой уровень привилегий.

3) Процессор переключается на стек, определенный полями **ss0** и **esp0** в структуре TSS, причём поле **esp0** имеет значение **kstacktop**.

4) Процессор помещает значения регистров на момент исключения в стек ядра, начинающийся с адреса **kstacktop**. Как мы видим ниже, в стек попадает значение старой (пользовательской) вершины стека, точки выполнения, а также регистр флагов — итого пять слов (16-битные селекторы дополняются до 32-битного слова).

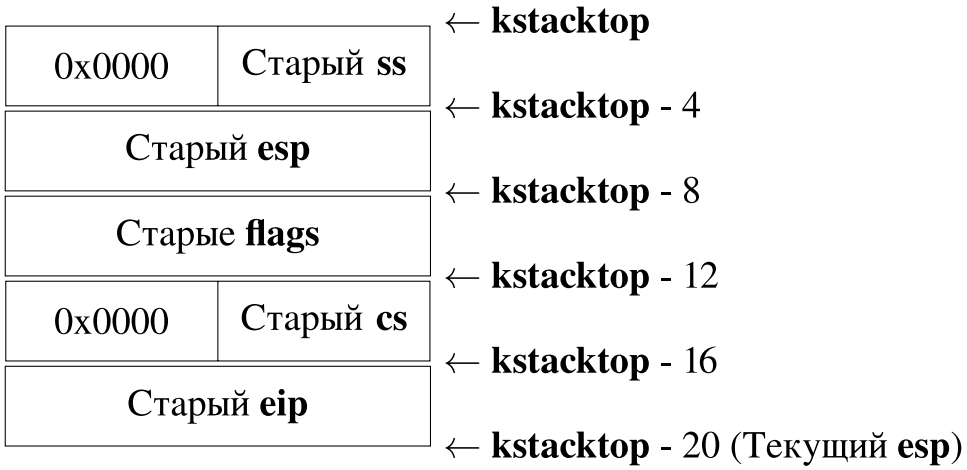

Рисунок 1.4 — Стек после переключения на режим ядра

5) Поскольку мы обрабатываем ошибку деления, которой соответствует номер прерывания 0 в архитектуре x86, процессор читает запись IDT с индексом 0 и устанавливает пару **cs:eip** в соответствующие значения.

6) Функция обработчика обрабатывает исключение.

7) В конце работы обработчик возвращается в режим пользователя, если есть хотя бы один выполняемый процесс. Следует отметить, что обработчик может вернуться не в тот же процесс, в котором случилось прерывание.

Для определенных типов исключений в x86, в дополнение к «стандартным» пяти словам выше, процессор помещает в стек ещё одно слово, содержащее код ошибки. Страничное исключение с десятичным номером 14 является важным примером. Используйте руководство по i386 [6, 7], чтобы определить, для каких номеров исключений процессор помещает в стек код ошибки и что в этом случае означает код ошибки.

Когда процессор помещает в стек код ошибки, в начале обработчика исключения после перехода из пользовательского режима стек выглядит следующим образом (рисунок 1.5).

Все синхронные исключения, которые могут быть сгенерированы самим процессором, используют номера прерываний от 0 до 31, и поэтому соответствуют записям IDT с индексами 0–31. Например, для обработчика страничных отказов (англ. *page faults*) должен использоваться номер прерывания 14. Номера прерываний, большие чем 31, используются только для прерываний, которые могут быть сгенерированы инструкцией **int** и для асинхронных аппаратных прерываний, вызываемых внешними устройствами.

Отметим, что исключения и прерывания позволяют перейти с более низкого уровня привилегий на более высокий или на тот же, но не на более

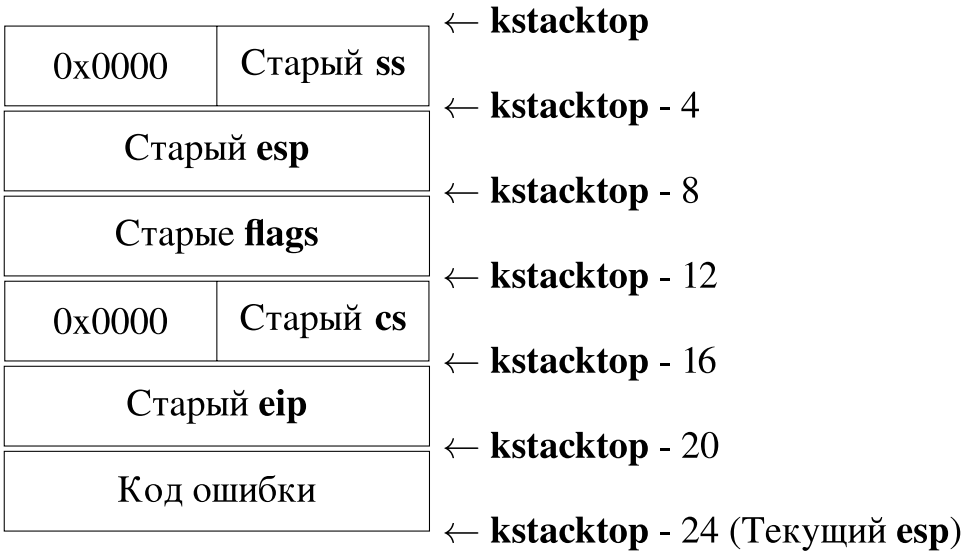

Рисунок 1.5 — Стек после переключения на режим ядра при наличии кода ошибки

низкий. При переходе на более высокий уровень, когда младшие два бита текущего значения регистра **cs** имеют значение больше, чем они же у поля **cs** в таблице IDT, процессор автоматически переключается на новый стек, прежде чем затолкнуть туда старое состояние регистров и вызвать соответствующий обработчик исключения. Если процессор уже находится на том же уровне, на котором должен выполняться обработчик прерывания, то ядро использует текущий стек. В последнем случае в стеке не сохраняется старое значение вершины стека. Инструкция возврата из прерывания **iret** поступает аналогично — при возврате на более низкий уровень привилегий стек будет сменен, при возврате на тот же уровень — нет. Таким образом, ядро может обрабатывать вложенные исключения, вызванные кодом непосредственно в ядре.

Если процессор в момент прерывания уже находится на том же уровне привилегий, на котором будет выполняться обработчик, то он не сохраняет старые значения регистров **ss** и **esp**. Это связано с тем, что в таком случае ему не нужно переключаться на использование стека другого уровня привилегий. Стек в этом случае будет иметь следующий вид для исключений, которые не помещают в стек код ошибки (рисунок 1.6).

Для типов исключений, которые помещают в стек код ошибки, процессор располагает его сразу же после старого значения **eip**, как и раньше.

1.11 Страничное исключение

Операционные системы обычно полагаются на аппаратное обеспечение (ЦП и MMU) при реализации защиты памяти процессов друг от друга и

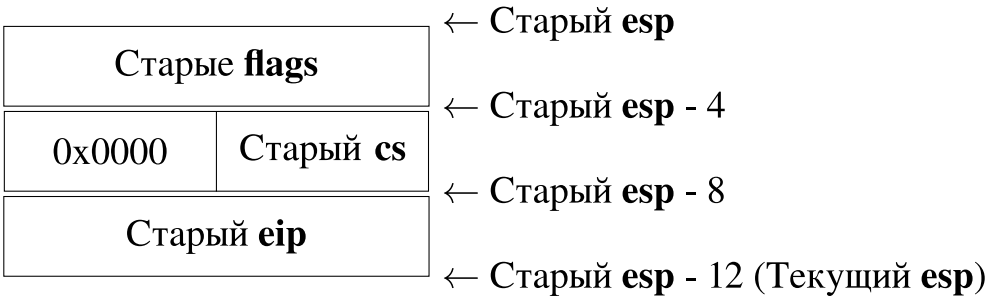

Рисунок 1.6 — Стек прерывания в случае перехода на тот же уровень привилегий

защиты памяти операционной системы от кода, работающего в режиме пользователя. Благодаря таблицам страниц ядро нашей ОС «сообщает» MMU, какие виртуальные адреса корректны и какие права имеет процесс при доступе к данным по данному адресу. При доступе по некорректному адресу или при нарушении прав доступа происходит страничное исключение (англ. *page fault*, номер — 14) и управление передаётся обработчику в ядре ОС.

При обработке этого исключения ОС может:

– исправить ситуацию, если это возможно;
– уничтожить проблемный процесс;
– аварийно завершить свою работу («запаниковать»), если проблема вызвана кодом самого ядра.

Причиной страничного исключения может быть как нарушение прав доступа к странице (попытка записи в страницу со сброшенным флагом R/W), так и обращение к странице, отсутствующей в физической памяти (со сброшенным флагом P).

В современных системах страничное исключение используется для автоматического увеличения размера стека программы и для реализации выгрузки содержимого малоиспользуемых страниц физической памяти на диск и последующей их «подкачки» с диска в момент исключения. В разрабатываемой нами системе этих возможностей, к сожалению, не будет.

Ещё одним возможным применением страничного исключения является быстрое клонирование процессов, используемое в POSIX-системах. Мы реализуем аналогичный механизм в главе 8.

1.12 Контрольные вопросы

Подготовьте ответы на следующие вопросы.

1) Сколько регистров общего назначения имеет процессор i486?

2) Как адресуется память в реальном режиме x86?

3) Какой адрес принимает инструкция **lgdt**?

4) Как адресуется память в защищённом режиме x86 без использования страниц?

5) В каталоге страниц хранятся виртуальные или физические адреса таблиц страниц?

6) Что хранится в регистре **cr3**?

7) Как по линейному адресу рассчитывается физический в случае размера страниц 4 Кб?

8) Может ли адрес таблицы страниц не быть кратен 0x1000?

9) Запрещает ли вентиль линии A20 доступ ко всей памяти свыше 1 Мб?

10) Сколько уровней защиты поддерживает страничное преобразование в архитектуре x86?

11) Можно ли при страничном преобразовании запретить доступ ядру к памяти, разрешив его пользователю?

12) Можно ли страничным преобразованием запретить исполнение содержимого памяти как кода?

13) Какой размер одной таблицы страниц и одного элемента таблицы страниц при размере страницы 4 Кб?

14) Какие части линейного и физического адреса хранятся в TLB?

15) В каких случаях ОС должна явно управлять содержимым TLB?

16) По какому адресу BIOS загружает начальный загрузчик ОС?

17) Как в программах обычно используется регистр **ebp**?

18) Можно ли запретить исключения?

19) Что извлекается из таблицы IDT в момент исключения или прерывания?

20) В чём принципиальная разница между шлюзами ловушек и шлюзами исключений?

21) Сохраняется ли на стеке информация о номере произошедшего прерывания?

22) Есть ли в сегменте состояния задачи поля **ss3** и **esp3**? Почему?

23) Для чего используется регистр **cr2**?

2 Организация исходных текстов JOS

При разработке операционных систем традиционно используется язык C по ряду причин:

– возможность прямого доступа к памяти, манипуляций битовыми структурами, свободное приведение типов;
– возможность собирать программы без подключения стандартной библиотеки;
– совпадение имён в программе и в объектном файле (отсутствие декорирования имён, которое используется, например, компилятором языка C++);
– отсутствие ненужных или трудно реализуемых в ядре сущностей, таких как обработка исключений или сборка мусора;
– возможность создания вставок на машинном языке;
– относительная простота реализации компилятора языка.

Часть операционной системы не может быть реализована на языке C, поскольку требует явного использования машинных команд, предназначенных для поддержки реализации операционной системы. Такие части реализуются на языке ассемблера — в виде отдельных модулей или в виде ассемблерных вставок в функции на языке C.

2.1 Ассемблер GNU Assembler

Исторически сложилось так, что существуют два альтернативных синтаксиса ассемблера процессоров x86: синтаксис Intel и синтаксис AT&T, известный как синтаксис UNIX. Первый из них был введен фирмой Intel в своих руководствах для пояснительных примеров и был затем использован авторами многих трансляторов, за заметным исключением GNU Assembler (до версии 2.10 он не поддерживал синтаксис Intel). Второй основан на том синтаксисе, который использовался в UNIX ещё до появления самой платформы x86.

Хотя синтаксис Intel претендует на стандарт де-факто, JOS использует синтаксис AT&T как стандартный для средств разработки GNU и большинства UNIX-подобных систем. Основные отличия между двумя синтаксисами перечислены ниже.

Порядок операндов. В синтаксисе AT&T присваивание идет **слева направо** (**movl $1, %eax**). В синтаксисе Intel — **справа налево** (**mov eax, 1**).

С одной стороны, пересылка в правом направлении совпадает с чтением текста слева направо, с другой, **sub eax, 1** выглядит очевиднее, чем

subl $1, %eax, поскольку в последнем меняется порядок вычитания, а в первом — напрашивается аналогия с операцией «-=» в языке C.

Суффиксы размерностей операндов. В AT&T команда заканчивается однобуквенным суффиксом, показывающим разрядность операндов команды:

1) **l** — long, четыре байта;
2) **w** — word, два байта;
3) **b** — byte, один байт.

Например: **movl somevar, %eax**.

В синтаксисе Intel отсутствует четкий механизм задания размерности, она определяется из разрядностей операндов: **mov eax, somevar**. В случае, если операнд — адрес памяти, в синтаксисе Intel используются префиксы **byte ptr**, **word ptr**, **long ptr**, например: **mov eax, [long ptr 0x1000]**.

Префиксы операндов. В AT&T операнд дополняется особым символом-префиксом (сиджилом), указывающим его вид:

– регистр — процентом: **%eax**;
– непосредственный операнд — символом USD: **$1**;
– косвенный (хранимый в памяти) адрес перехода — звездочкой: ***addr**.

Прочие операнды в памяти не имеют сиджила, например: **0x1000** (это не число 0x1000, а содержимое памяти по данному адресу!), **somevar**.

Благодаря префиксам в синтаксисе AT&T не возникает проблемы различения обращения к переменной (т. е. по адресу памяти) и к адресу переменной (т. е. использование его как непосредственного операнда). В синтаксисе AT&T **somevar** всегда означает ячейку памяти, а **$somevar** — адрес этой ячейки, тогда как синтаксис Intel не устанавливает четких правил на этот случай, что приводит к разногласиям в разных его реализациях. Например, у транслятора NASM это будут, соответственно, **[somevar]** и **somevar**, а у транслятора MASM — **somevar** и **OFFSET somevar**.

Адресация база-масштаб-смещение. В синтаксисе Intel сложение базы со смещением имеет интуитивный вид: **[var + eax + 2*ebx]**. В синтаксисе AT&T используется форма **var(%eax, 2, %ebx)**. В случае, если какая-то из частей составного адреса отсутствует, она пропускается: **-4(, %eax, 2)** есть **[2*eax - 4]** в нотации Intel.

2.2 Диалект GNU C

В JOS используется диалект языка C, известный как GNU C, близкий к изложенному в классическом издании [3]. Подчеркнём, что в дальнейшем используется именно язык и компилятор C, а не язык C++.

Поскольку при разработке ОС у нас не будет в распоряжении стандартной библиотеки языка C (пока мы сами не создадим некоторую её часть, начиная с главы 6), то знания о ней нам почти не понадобятся.

Опишем две основные возможности диалекта GNU C, которые отсутствуют в [3] и действующем стандарте [4].

Операция typeof. Исходные тексты JOS предполагают наличие конструкции **typeof**, крайне полезной для использования в макросах. Она не введена в стандарт языка, поскольку, формально, её сложно назвать операцией: её результат определяется во время компиляции (подобно операции **sizeof**) и является типом аргумента, если он известен компилятору.

Составной оператор внутри выражения и inline-функции. В JOS встречаются многочисленные inline-функции, в основном, в файле **inc/x86.h**. Такие функции были введены в стандарте C99 [4].

На момент начала разработки JOS inline-функции в компиляторе и стандарте ещё отсутствовали. Для того, чтобы макросы могли реализовывать аналогичный функционал, в диалекте GNU C был добавлен составной оператор внутри выражения, причём возвращающий значение.

```
x = {int c = 2; c} + 1;
// Значение переменной x будет равно трём.
```

Отметим, что макросы с параметрами и составным оператором не могут быть полностью заменены inline-функциями, поскольку код макроса имеет полный доступ к локальному контексту, а его «аргументами» могут быть и конструкции языка.

В файле **CODING** в исходных текстах JOS описан рекомендованный для её исходных текстов стандарт оформления кода. Отметим здесь основные из этих рекомендаций.

Имена функций всегда состоят из строчных букв, слова обычно разделяются символами подчёркивания (**boot_alloc**, **page_insert**). Имена макросов всегда состоят из прописных букв, кроме нескольких исключений (**assert**, **panic**, **static_assert**, **offsetof**). Функция без аргументов объявляется как **f(void)**.

При вызове функции пробел перед открывающей скобкой не ставится, в остальных случаях — ставится, в типе указателя пробел ставится перед

звёздочкой. Открывающая фигурная скобка составных операторов начинается на той же строке после пробела. Ниже показан корректно оформленный фрагмент кода.

```
if (a) {
    int *p = get_ptr();
    ...
```

При определении функции имя функции начинается с новой строки с первой позиции, чтобы это определение можно было тривиально найти в коде JOS примерно следующей командой.

```
grep -n "^foo(" -RI *
```

2.3 Ассемблерные вставки

Если нам не нужно использовать много кода на ассемблере, а достаточно лишь пары строк (например, записать значение переменной в порт ввода-вывода), имеет смысл использовать ассемблерные вставки (англ. *inline assembly*) — механизм, позволяющий вставлять ассемблерный код в тело функции на языке C. Для этого в компиляторе GCC существует ключевое слово **asm**. Простейшая форма ассемблерной вставки имеет вид **asm ("....")**:

```
asm ("movl $0xfe, %eax \n"
     "outb %al, $0x64");
```

Содержимое строки внутри **asm** посылается на вход транслятору gas (GNU Assembler), поэтому отдельные инструкции разделяются переносом строки (\n).

В ассемблерной вставке нельзя напрямую обратиться к переменной по её имени. Для этого существует полная форма:

```
asm ( код
      : список выходных переменных
      : список входных переменных
      : список модифицируемых внути вставки регистров);
```

Полная форма вставки позволяет указать GCC, какие переменные будут служить входными операндами для кода вставки, а какие — выходными. В коде вставки для обращения к операндам используется синтаксис %n, где n — порядковый номер операнда в общем списке.

Каждый операнд объявляется в списке при помощи синтаксиса `"constraint" (C expression)`, где **constraint** определяет режим адресации операнда. Существует довольно много форматов, из которых наиболее часто используются `"m"` (входной операнд в памяти), `"=m"` (выходной операнд в памяти), `"r"` и `"=r"` (входной-выходной операнд в любом регистре). Например, следующий код скопирует значение переменной **x** в переменную **y**.

```
int x = 1, y;
asm ("movl %1, %eax \n" // movl x, eax
     "movl %eax, %0"    // movl eax, y
     : "=m" (y)
     : "m" (x));
```

В приведённом примере вместо `%1` подставляется адресное выражение для переменной **x**, а вместо `%0` — **y**.

Используя `"r"`, можно указать GCC, что необходимо выделить какой-нибудь регистр общего назначения и сохранить туда значение указанного выражения перед выполнением кода вставки. В следующем примере сумма переменных **x** и **y** будет помещена в автоматически выбранный регистр общего назначения, а потом скопирована в переменную **z** через регистр **eax**.

```
int x = 1, y = 2, z;
asm ("movl %1, %eax \n"
     "movl %eax, %0"
     : "=m" (z)
     : "r" (x + y));
```

Список модифицируемых регистров содержит все регистры, которые модифицируются кодом вставки. Обычно его нет смысла указывать, так как GCC может сам определить, какие регистры затрагиваются вставкой.

Вместо **asm** можно также писать **__asm__**.

2.4 Ограничение оптимизации обращений к переменным

Допустим, имеется следующий код.

```
while (!ready);
```

Если переменная **ready** объявлена просто как **int ready**, реальное обращение к хранящей значение переменной ячейке памяти, возможно, произойдет лишь один раз, после чего последует бесконечный цикл. С точки зрения компилятора значение **ready** не может измениться — мы ведь не изменяем её значение в теле цикла.

В языке C (стандарт C99 [4]) модификатор **volatile** означает, что помеченная им переменная может изменить свое значение в любой момент. Если переменную выше объявить как **volatile int ready**, компилятор будет исходить из предположения, что некие другие сущности (например, другой поток выполнения) может изменить значение переменной, и будет считывать её новое значение из памяти после каждой итерации цикла.

Применительно к ассемблерным вставкам **volatile** имеет аналогичное значение: он является сигналом компилятору, что приведённый блок ассемблерного кода должен быть вставлен как есть, даже если, с точки зрения компилятора, его выполнение не приведёт к наблюдаемым побочным эффектам. Дело в том, что GCC пытается определить побочные эффекты ассемблерной вставки и, если они отсутствуют, убрать её (в отличие от большинства компиляторов, которые просто отключают большую часть оптимизаций для функции, где встречается хоть одна ассемблерная вставка). Допустим, надо сделать задержку с помощью операции **nop** (отметим, что на практике так делать нельзя, поскольку мы не знаем, сколько времени выполняется операция). Для этого можно написать следующий код.

```
asm ("nop \n nop \n nop");
```

Компилятор GCC посчитает, что эти команды просто не нужны, так же как он отбрасывает любые ненужные вычисления в коде на языке C. Модификатор **volatile** укажет компилятору, что надо вставить ассемблерный код, несмотря на отсутствие видимых побочных эффектов:

```
asm volatile ("nop \n nop \n nop");
```

Практически любые ассемблерные вставки, связанные не с оптимизацией вычислений (использованием SSE и т. п.), а с взаимодействием с внешними устройствами или другими потоками (**lock xchg**), требуют этого модификатора для правильной работы.

Следует отметить, что даже вставка с **volatile** может быть выкинута из кода, если она недостижима.

```
if (0) {
    asm volatile("hlt"); // не будет в исполняемом файле
}
```

Ключевое слово **__volatile__** является синонимом слова **volatile**.

2.5 Обзор исходных текстов JOS

Исходные тексты JOS организованы так, чтобы на каждом этапе выполнения заданий студент работал с как можно меньшим объемом исходного кода. Поэтому сначала, в главе 3, мы будем работать с версией ядра JOS, в которой нет почти ничего. Затем, в главе 4, в ядро будет добавлен код для работы со страницами памяти. В главах 5 и 6 в системе появится поддержка кода пользовательского режима. Наконец, в главах с 7 по 10 мы будем работать с ядром JOS, имеющим возможности по планированию процессов, клонированию процессов и межпроцессному взаимодействию.

Все изменения, которые мы будем вносить в исходные тексты ядра JOS в ходе выполнения задания для самостоятельной работы, необходимо будет переносить на последующие версии исходников JOS. Поэтому для организации исходников JOS мы будем использовать систему контроля версий Git[1], а каждая версия JOS будет представлена отдельной веткой в репозитории. Всего в репозитории существует четыре ветки:

1) ветка **lab1** используется в главе 3;
2) ветка **lab2** используется в главе 4;
3) ветка **lab3** используется в главе 5 и главе 6;
4) ветка **lab4** используется в главах 7–10.

Перенос выполненных читателем изменений из одной ветки в другую реализуется через слияние веток, как будет показано позже.

Для выполнения заданий будет достаточно использовать локальный репозиторий Git. Необходимые ветки мы будем по мере необходимости брать из готового репозитория JOS[2], который нам доступен только для чтения.

После успешного выполнения каждого задания, требующего изменения кода, следует зафиксировать изменения в системе контроля версий. Это можно сделать командой следующего вида с указанием номера задания.

```
git commit -am "Выполнено задание 1."
```

При желании, конечно, можно фиксировать и промежуточные изменения кода, не соответствующие полностью выполненному заданию.

Исходные тексты JOS размещены в каталоге **workbook_jos/labs**. Исходники JOS разделены по следующим каталогам:

[1]Подробнее о системе контроля Git при желании можно узнать например здесь: **http://git-scm.com/book/ru**.

[2]Адрес репозитория: **http://dev.iu7.bmstu.ru/git/workbook_jos**

– **boot** — файлы загрузчика ядра ОС;
– **inc** — заголовочные файлы;
– **kern** — файлы ядра ОС;
– **user** — файлы режима пользователя (до главы 5 этот каталог будет пуст);
– **lib** — служебная библиотека ядра (может быть собрана и как часть ядра и как часть программ пользователя);
– **obj** — место размещения файлов, полученных в ходе компиляции.

В каталоге **obj** в процессе сборки ядра помимо бинарных файлов создаются ассемблерные листинги с адресами функций скомпилированных файлов (файл **obj/kern/kernel.asm**).

2.6 Отладка кода ядра

В ходе выполнения практических заданий вы наверняка столкнётесь с ситуацией, когда написанный вами код не работает или даже приводит к аварийному завершению работы операционной системы. Поскольку в основном наш код будет работать в режиме ядра, аварийное завершение его работы будет довольно типичным результатом большинства ошибок.

К счастью, мы будем выполнять наш код на интерпретирующем эмуляторе, что даст нам возможность легко просматривать содержимое ячеек памяти и регистров ЦП, выводить содержимое стека, устанавливать точки останова как по конкретным адресам, так и связывать их с событиями изменения некоторых ячеек памяти и так далее. Информацию об адресах при этом можно получить из файла **obj/kern/kernel.asm**.

Несмотря на широкие возможности отладчика используемого эмулятора, практика чтения курса показала, что самым удобным приёмом отладки является обычная отладочная печать. Простое использование функции **cprintf** позволяет локализовать ошибки в большинстве практических заданий. Программисту при этом доступны основные спецификаторы вывода для вывода заначений в десятичном и шестнадцатеричном форматах. Практически единственным случаем, когда отладочная печать не может помочь в поиске ошибки, является случайная порча некоторой ячейки памяти в совершенно неизвестный момент времени.

Программисту также рекомендуется применять защитное программирование в виде использования макроса **assert**. Следующий пример показывает его возможное применение. Отметим, что выражениям-утверждениям крайне нежелательно иметь побочные эффекты.

```
#include <inc/assert.h>
...
// Указатель p сейчас не должен быть нулевым.
assert(p);
// Счетчик p->cnt не должен быть нулевым.
assert(p->cnt > 0);
p->cnt--;
```

Если требуемое в аргументе макроса **assert** условие не выполнено, то будет вызвана функция **panic** и ядро «запаникует»: оно перейдёт в режим исполнения введенных с клавиатуры команд, предназначенных для выявления причин ошибки. Этот режим называется режимом монитора. В разделе 3.5 мы добавим в этот режим команду для вывода на экран стека вызова функций. Это даст возможность просмотреть информацию об аргументах и адресах функций, приведших в итоге к панике.

Следует отметить, что при попытке разыменования нулевого указателя ядро в части заданий будет переходить в режим монитора самостоятельно в ходе обработки соответствующего исключения. Несмотря на это, при выполнении практических заданий перед разыменованием указателя часто стоит всё равно использовать макрос **assert**: обработка исключений сама может содержать ошибки или отсутствовать.

2.7 Контрольные вопросы

Подготовьте ответы на следующие вопросы.

1) Почему inline-функция не всегда может заменить макрос с параметрами?

2) Изменяет ли inline-функция значение регистра **ebp** в момент начала своей работы?

3) Когда и как следует фиксировать изменения в репозитории системы контроля версий?

4) Что такое проверка утверждений?

5) Какие способы отладки ядра ОС нам доступны?

6) Что такое «паника» ядра и когда она случается?

3 Начальная загрузка системы

В этой работе мы соберем первую версию JOS, затем изучим процесс начальной инициализации от включения компьютера до начала работы ядра системы, используя эмулятор Bochs и его встроенный отладчик, и выполним практическое задание, связанное с выводом информации о кадрах стека.

Платформа x86 была рассчитана на 16-битную ОС реального режима и поэтому очень мало чем поможет загрузчику ядра операционной системы: он не сможет даже использовать прерывания BIOS для чтения данных с диска. В итоге весь процесс загрузки от начала работы загрузчика до полной инициализации ядра чем-то напоминает «вытаскивание себя самого за волосы», причем итерационное: например, мы трижды сменим используемую таблицу GDT.

Эта особенность архитектуры x86 делает её, возможно, даже более привлекательной с учебной точки зрения: вместо необходимости изучения объёмистой документации по взаимодействию со сложной системой загрузки (например, современного стандарта UEFI) разработчик операционной системы должен просто сделать сам всю эту работу, и нам придётся принять в этом процессе самое непосредственное участие.

Перед началом работы нам следует создать на локальном диске репозиторий с исходниками системы. Для этого запустим командную оболочку[1] — все дальнейшие команды мы будем выполнять в ней. Перейдем в каталог, выбранный как рабочий, и затем выполним следующие команды для клонирования репозитория исходных текстов JOS, использующего систему контроля версий Git.

```
git clone http://dev.iu7.bmstu.ru/git/workbook_jos
cd workbook_jos
git checkout -t origin/lab1
```

Теперь у нас есть каталог **workbook_jos**, содержащий локальный репозиторий исходных текстов JOS. Выполнив последнюю команду(**git checkout**), мы переключились на ветку **lab1** в этом репозитории. Команды **git branch** и **git log** покажут нам текущую ветку и историю изменений репозитория. Все дальнейшие команды системы Git по работе с нашим репозиторием надо выполнять, находясь в каталоге **workbook_jos** или любом его подкаталоге.

[1] Обычно это интерпретатор Bash, запущенный внутри эмулятора терминала, такого как Gnome Terminal, Konsole, Xterm или аналогичного.

3.1 Сборка и запуск системы

Для сборки исходников JOS в бинарный файл с образом диска нужно перейти в каталог **workbook_jos/labs** и выполнить команду **make**. После успешной сборки образа системы командой **make** можно приступать к её запуску в эмуляторе Bochs. Для этого следует выполнить в этом же каталоге команду **bochs**, запустив виртуальную машину-эмулятор. После запуска эмулятор остановит выполнение системы в режиме встроенного отладчика[1], и мы увидим приглашение отладчика Bochs.

Продолжить выполнение команд в эмуляторе можно командой **c** (или без сокращения — **continue**). Подробную справку по отладчику Bochs можно увидеть в его документации[2]. Для вызова отладчика Bochs в ходе эмуляции можно в любой момент нажать комбинацию Ctrl+C. Для завершения работы эмулятора следует дать команду **q** в отладчике. Команда **help** выдаёт краткую подсказку.

После команды **c**, отданной отладчику, система начнёт выполнение кода BIOS, и в итоге дойдёт до запуска монитора ядра JOS: появится приглашение монитора (**K>**). В мониторе в настоящий момент доступны только команды **help** и **kerninfo**. Последняя позволяет просмотреть адреса специальных символов в бинарном файле ядра.

Перезапустим эмулятор Bochs и начнём пошаговое исполнение кода BIOS: используя команду **n** (или **next**), пройдёмся отладчиком по началу кода BIOS виртуальной машины. Для повтора последней команды эмулятора достаточно нажимать клавишу «Enter». Изучив список стандартных портов ввода-вывода[3], можно понять, с чего начинает свою работу код BIOS.

3.2 Загрузчик ядра операционной системы

Исходные тексты загрузчика JOS находятся в файлах **boot/boot.S** и **boot/main.c**. В файле **boot/Makefrag** находится сценарий его сборки, вызываемый из основного сценария сборки ядра. По соглашениям BIOS загрузчик будет загружен из первого сектора устройства в память по адресу 0x7C00, а его размер не может превышать 510 байт.

Используемый в нашей системе загрузчик предполагает, что начиная со второго сектора на диске находится ядро ОС — в первом секторе диска находится сам загрузчик. Для передачи управления ядру JOS наш загрузчик должен выполнить следующие шаги.

---

[1] Иногда нужно также выбрать пункт меню «Begin simulation».

[2] **http://bochs.sourceforge.net/doc/docbook/user/internal-debugger.html**

[3] **http://www.philipstorr.id.au/pcbook/book2/ioassign.htm**

1) Переключение процессора в 32-битный защищённый режим и включение доступа к памяти свыше 1 Мб.

2) Считывание заголовка ядра с диска в некоторую область памяти путём прямого программирования контроллера IDE (наша система умеет работать только с IDE-дисками).

3) Извлечение из заголовка ELF-файла ядра информации о его секциях и адресе точки входа в ядро.

4) Считывание необходимых секций файла ядра с диска в ОЗУ.

5) Передача управления на адрес, рассчитанный на основе адреса точки входа в ядро из заголовка ELF-файла.

Для создания точки останова в начале загрузчика нужно отдать в отладчике команду **b 0x7C00**. Можно использовать адреса функций в ассемблерных листингах в каталоге **obj/boot/** для установки таких точек на начало других функций загрузчика.

3.3 Формат файла ядра

Собранное ядро находится в файле **obj/kern/kernel** и представляет собой файл в формате ELF — стандартном формате исполняемых файлов во многих UNIX-подобных системах. Поскольку в таких системах уже есть компилятор и компоновщик для создания ELF-файлов, а сам формат хорошо документирован, то авторы JOS решили воспользоваться им.

Для просмотра информации о файле можно воспользоваться программой **readelf**. Помимо секций и их адресов, интерес представляет также адрес точки входа в ядро, показанный в заголовке. Например, следующая команда покажет основную информацию о содержимом файла ядра JOS.

```
readelf -hS obj/kern/kernel
```

Найдите размеры и начальные адреса следующих основных секций файла:

– машинный код: секция **.text**;

– неинициализированные (т. е. нулевые) глобальные данные: секция **.bss**;

– инициализированные глобальные данные: секция **.data**;

– неизменяемые данные: секция **.rodata**.

Наш загрузчик ОС представляет собой упрощённый загрузчик ELF-файлов. Бинарная структура таких файлов описана в заголовочном файле **inc/elf.h**. Для компоновки ядра используется сценарий компоновщика **kern/kernel.ld**. В нём, в частности, указан базовый адрес компоновки ядра.

3.4 Область размещения ядра в памяти

В большинстве 32-битных операционных систем код и данные ядра находятся в области старших виртуальных адресов, в то время как младшие адреса отданы для кода и данных программы, работающей в режиме пользователя (что часто сокращается до «отданы пользователю»). Граница между пользователем и ядром в существующих ОС часто проходит по виртуальному адресу 2 Гб или 3 Гб, но в JOS она проходит по адресу 0xEEC00000.

При переходе в защищённый режим загрузчик устанавливает тривиальное сегментное преобразование памяти, при котором смещение равно физическому адресу. Для этого он использует таблицу GDT, описанную в файле **boot/boot.S**. Ядро, как считает загрузчик, в дальнейшем предполагает размещение в старших виртуальных адресах, в силу чего считанные загрузчиком из ELF-файла ядра адреса не могут быть использованы напрямую и должны быть пересчитаны в физические.

При проектировании загрузчика ОС не хотелось бы, чтобы он оперировал неким тайным знанием о будущем расположении ядра в виртуальных адресах. По этой причине загрузчик JOS использует следующее соглашение: старший байт всех прочитаных из ELF-файла адресов ядра он просто обнуляет, что и приводит виртуальные адреса ядра (точнее, адреса компоновки ядра) к физическим. Оставшуюся часть — младшие три байта — загрузчик считает верным физическим адресом, или, что то же самое, смещением логического адреса при активной GDT загрузчика.

Ядро, таким образом, начнёт свою работу при указанном тривиальном сегментном преобразовании, установленном GDT загрузчика. Поскольку при таком преобразовании адреса компоновки ядра не равны его виртуальным адресам, то оно должно как можно быстрее перейти на собственную таблицу GDT. Файл **kern/entry.S** содержит таблицу GDT и команду **lgdt** для её загрузки. Переход на новую GDT включает операцию перезагрузки дескрипторов сегментных регистров, причём для перезагрузки регистра **cs** нужна инструкция дальнего перехода.

После перехода на новую GDT адреса компоновки ядра будут совпадать с виртуальными адресами, что позволит ядру нормально работать. Эта GDT преобразует старшие виртуальные адреса в младшие физические: её элементы имеют отрицательный базовый адрес **-KERNBASE** (в GDT этот адрес хранится в дополнительном коде). Сам код и статические данные ядра начинаются с адреса 0xF0100000, который преобразуется GDT ядра в адрес 0x00100000. Отметим, что преобразование адресов загрузчиком (он, как указано ранее, просто обнуляет старший байт) будет давать такой же результат, если первый байт адреса равен 0xF0.

В итоге код ядра JOS физически будет располагаться начиная со второго (если начинать отсчет с единицы) мегабайта, а в виртуальном адресном пространстве — не доходя 255 Мб до его верхней границы в 4 Гб.

3.5 Стек ядра и обратная трассировка вызовов функций

Ядро JOS использует собственный стек, который должен быть настроен до вызова первой функции инструкцией **call**, поскольку она использует стек неявным образом. Выяснить, как ядро настраивает свой стек, и где он располагается в памяти, можно в файле **kern/entry.S**. Выясните, сколько 32-битных слов помещает в стек каждый вложенный вызов функции **test_backtrace**. Для этого найдите адрес функции **test_backtrace** в файле **obj/kern/kernel.asm** и установите по нему точку останова в эмуляторе командой **lb**, линейный адрес функции для точки останова следует взять из файла **obj/kern/kernel.asm**. Изучите, что происходит каждый раз, когда эта функция вызывается после запуска ядра. Просматривать содержимое регистров можно командой отладчика **r**.

Описанная в разделе 1.8 возможность обратной трассировки кадров стека может быть особенно полезной, если некоторая функция ядра вызывает макросы **assert** или **panic**: возможно, этой функции были переданы неправильные аргументы, и хотелось бы узнать, каковы их значения и какая функция их передала.

При помощи обратной трассировки стека мы сможем проследить последовательность вызовов функций. Для этого нам необходимо реализовать функцию обратной трассировки стека, которую следует назвать **mon_backtrace**. Заготовка этой функции уже находится в файле **kern/monitor.c**. Её можно написать полностью на языке C, если использовать функцию **read_ebp**, которая находится в файле **inc/x86.h**. Эта функция, как видно по названию, возвращает значение регистра **ebp** в вызывающей функции, поскольку, благодаря нестандартному описанию[1], является гарантированно встраиваемой.

Функция обратной трассировки стека должна выводить распечатку вызовов в следующем виде.

```
ebp f0109e58 eip f0100a62 args 00000001 f0109e80 f0109e98 f0100ed2 00000031
ebp f0109ed8 eip f01000d6 args 00000000 00000000 f0100058 f0109f28 00000061
...
```

Первая строка показывает аргументы функции, выполняющейся в настоящий момент времени — здесь это функция **mon_backtrace**. Вторая

[1]Здесь используется нестандартный механизм атрибутов компилятора GCC, чтобы гарантировать, что функция всегда будет inline-функцией.

строка показывает аргументы функции, которая вызвала первую функцию, и так далее. В каждой строке, выводимой функцией **mon_backtrace**, значение **ebp** показывает значение указателя базы стека, используемое очередной вызванной функцией. Далее выводится сохранённое в стеке значение **eip** при возврате из данной функции — адрес инструкции сразу после инструкции **call**, вызвавшей функцию. Пять шестнадцатеричных значений в конце строки являются первыми пятью аргументами рассматриваемой функции, которые были положены в стек непосредственно перед тем, как функция была вызвана. Конечно, если функция была вызвана менее чем с пятью аргументами, то не все из этих значений будут полезны.

Нужно напечатать все кадры стека от момента первого вызова функции в ядре. Для определения конца кадров стека можно, например, завершить их связанный список нулевым указателем. Для этого ядро записывает в регистр **ebp** нулевое значение до первой инструкции **call**[1] — это приведет к его помещению в стек в начале первой вызванной функции.

Задание 1. Допишите функцию **mon_backtrace** и добавьте её в список команд монитора ядра так, чтобы эта функция могла быть вызвана пользователем в интерактивном режиме монитора командой **backtrace**. Обратите внимание, что если ваша функция выводит в качестве аргументов сплошные значения **0xFFFFFFFF**, то это с большой долей вероятности означает, что вы пытаетесь читать память по адресу за границами ОЗУ. При отладке написанного кода может помочь команда отладчика Bochs **print-stack**. Напомним, что отладочная печать является важным средством отладки кода, работающим в режиме ядра, и простое использование функции вывода на экран часто помогает быстро локализовать ошибку в коде.

После успешного выполнения каждого задания, требующего изменения кода, следует зафиксировать внесённые изменения в нашем репозитории исходников. Это можно сделать командой следующего вида.

```
git commit -am "выполнено задание 1 (backtrace)"
```

3.6 Контрольные вопросы

Подготовьте ответы на следующие вопросы.

1) Почему самой первой выполняемой после запуска эмулятора командой будет переход?

2) Почему загрузчик должен переключиться в защищённый режим до загрузки ядра?

[1] Убедитесь, что ядро действительно делает это. Внесите необходимые изменения в код в случае, если это не так.

3) Почему загрузчик не может использовать прерывание BIOS с номером 0x13 для чтения ядра с диска?

4) Как загрузчик ядра инициализирует регистр указателя стека? Где находится стек при работе загрузчика?

5) Какой командой загрузчик переключает процессор в защищённый режим?

6) Как устроена GDT загрузчика?

7) Начиная с какого физического адреса загружается ядро?

8) Сколько секторов с заголовком файла ядра считывается с диска загрузчиком?

9) Какие адреса компоновки загрузчик считывает из описания секций файла ядра?

10) Как выравниваются в файле ядра начала секций?

11) Есть ли в ядре секция стека? Почему?

12) Где находится ядро в физической памяти? Как происходит пересчет адресов компоновки ядра при его загрузке?

13) Сколько секторов с ядром считывается с диска загрузчиком? Куда они считываются?

14) Как загрузчик выделяет память для чтения ядра с диска?

15) Какие секции ядра считывает загрузчик?

16) Как загрузчик загружает секцию **.bss**? Почему он поступает именно так?

17) Какова последняя исполняемая команда загрузчика?

18) Какова первая исполняемая машинная команда ядра?

19) Когда и как ядро переходит на новую GDT? Как она устроена?

20) Может ли загрузчик сразу использовать GDT, аналогичную GDT ядра с точки зрения базовых адресов?

21) Где находится ядро в виртуальной памяти при использовании GDT ядра и где — при использовании GDT загрузчика?

22) Какой адрес работающего ядра совпадает с адресом его компоновки — физический или виртуальный?

23) Как ядро резервирует физическое место для своего стека? Куда будет установлен указатель стека?

24) Сколько 32-битных слов помещает в стек каждый вложенный вызов функции **test_backtrace**?

25) Как функция обратной трассировки обнаруживает, что она дошла до последнего кадра стека? Какая конкретная машинная инструкция в ядре помогает этой функции обнаружить конец кадров стека?

26) Может ли функция обратной трассировки определить, сколько на самом деле было передано аргументов в функцию?

27) Какой аргумент находится в стеке рядом с адресом возврата — самый левый (с точки зрения вызова в языке C) или самый правый?

28) Какое значение выводит функция обратной трассировки в поле **eip**? Можно ли из этого адреса тривиально получить адрес точки вызова функции?

4 Страничное управление памятью

Хотя на начальном этапе работы ядро JOS использует сегментное преобразование для отображения старших виртуальных адресов в младшие физические, это является временной мерой. После инициализации ядро JOS переходит на плоскую, с точки зрения сегментов, модель памяти, поскольку начинает использование страничного преобразования адресов.

Для перехода на использование страничной адресации в ядре необходимо создать функцию, отображающую линейные адреса в физические — с её помощью мы отобразим код и данные ядра, а в дальнейшем будем использовать и для отображения кода и данных программ пользователя. Кроме того, наша ОС должна иметь простейший менеджер памяти для выделения в будущем физических страниц памяти под программы пользователя а также освобождения этих страниц (само ядро). Поскольку несколько разных виртуальных адресов (в том числе, в разных процессах) могут ссылаться на одну и ту же страницу физической памяти, то для отслеживания состояния страницы наш менеджер будет использовать счётчик ссылок на неё.

Перед продолжением работы нужно переключиться на новую ветку репозитория. Убедитесь командой **git status**, что вы зафиксировали все изменения, при необходимости — зафиксируйте их. Затем выполните следующие команды для переключения в новую ветку.

```
git checkout -b lab2
git pull http://dev.iu7.bmstu.ru/git/workbook_jos lab2
```

После этих команд все зафиксированные вами изменения попадут в новую ветку.

4.1 Организация виртуального адресного пространства

В операционных системах виртуальное линейное адресное пространство обычно разделено на две части. В первой находятся данные, код и стек процесса пользовательской прикладной программы (иногда это сокращают до «данные пользователя»). Для разных процессов эти страницы отображаются в различные, вообще говоря, области физической памяти.

Во второй части находятся данные, код и стек ядра. Несколько упрощённо можно считать, что эта часть отображается в одну и ту же область физической памяти для разных процессов (или не отображается никуда). Процесс не должен иметь доступ в эту память, пока он находится в режиме пользователя.

Отметим, что первая страница обычно никогда не отображается в физическую память. Это делается с целью отслеживания попыток обращения по нулевому указателю.

Обычно нижняя часть (младшие адреса) виртуального адресного пространства отведена под код и данные пользователя, а верхняя — под код и данные ядра. Между ними может находится небольшая буферная зона, не отображаемая в физическую память.

Наша система в целом соответствует этим представлениям об организации адресного пространства. Самым заметным исключением является отображение части данных ядра в область памяти, доступную пользовательским программам для чтения. Это сделано с целью упростить взаимодействие ядра и прикладных программ и вынести в дальнейшем часть функций по клонированию процессов в код режима пользователя.

В файле **inc/memlayout.h** находится раскладка виртуального пространства, выбранная разработчиками JOS. Отметим её основные моменты.

1) Нулевая страница памяти не отображается никуда после завершения инициализации ядра. Это сделано для обнаружения обращений по нулевым указателям.

2) Младшие адреса до адреса **UTOP** включительно содержат код, данные и стек программы пользователя.

3) Адреса от адреса **UTOP** до адреса **UVPT** доступы процессу только для чтения и содержат информацию о доступных страницах и таблицу страничного преобразования[1]. Здесь же будет находиться доступный пользователю только для чтения массив дескрипторов процессов (он появится в следующей главе).

4) Выше адреса **VPT** начинается область, недоступная в режиме пользователя даже для чтения.

5) Сначала в области ядра идёт стек ядра (растущий вниз), затем — таблица страниц текущего пространства.

6) Начиная с адреса **KERNBASE** расположена область, отображенная в первые 256Мб физической памяти. В ней же находится код ядра и все данные ядра, за исключением расположенных на стеке локальных переменных.

Часть вспомогательных макросов для работы с адресами содержится в файле **kern/pmap.h**. В частности, макросы **PADDR** и **KADDR** преобразовывают физический адрес ядра в виртуальный и обратно.

[1]Это не совсем типичное решение принято для упрощения взаимодействия прикладной программы с ядром, особенно в главах 7 и 8.

4.2 Выделение памяти при инициализации ядра

До перехода на выделение памяти страницами системе нужно выделить память для ряда управляющих структур и каталога страниц. Отметим сразу, что после этого ядро практически перестаёт выделять память под собственные нужды: дальнейшее выделение физической памяти происходит только для запуска и работы программ пользователя, включая создание новых таблиц страниц.

Выделение памяти для ядра на этапе его инициализации выполняет функция **boot_alloc** в файле **kern/pmap.c**. Вам нужно дописать её самостоятельно, прочитав комментарии в коде. Для хранения адресов можно использовать тип **uint32_t**, поскольку наша ОС 32-битная и непортируемая. Для округления адресов используйте макрос **ROUNDUP**.

Задание 2. Допишите функцию **boot_alloc**.

После каждого выполненного задания здесь и далее следует фиксировать изменения в репозитории.

Теперь нам нужно выделить память для массива дескрипторов физических страниц **pages**. Число элементов этого массива соответствует числу адресуемых физических страниц, последняя величина хранится в переменной **npage**.

Задание 3. Используя функцию **boot_alloc**, выделите память для массива **pages** в функции **i386_vm_init**, как этого требуют комментарии (до вызова фукции **page_init**). Уберите макрос **panic** в функции **i386_vm_init**.

Все последующие задания в этой главе будут касаться изменения файла **kern/pmap.c**. Проверить их правильность имеющимися в JOS тестами вы сможете только после выполнения всех заданий.

4.3 Список свободных страниц памяти

После выделения памяти для массива дескрипторов физических страниц **pages** нам нужно дописать базовые функции для работы со списком свободных физических страниц **page_free_list**. Список организуется набором макросов (файл **inc/queue.h**), традиционным для BSD-систем. Эти макросы позволят создать список из самих элементов массива **pages**.

Не считая полей для организации списка, в дескрипторе страницы (структура **Page**) есть только одно поле — счётчик ссылок на страницу из таблиц страничного преобразования. Таким образом, на каждую физическую страницу могут ссылаться несколько виртуальных, а наша система будет помечать страницу как свободную, когда счётчик этих ссылок доходит до нуля.

Функция **page_init** инициализирует список дескрипторов страниц памяти **page_free_list**, которые могут быть использованы. Такие страницы отмечаются в качестве свободных, для чего счётчик ссылок (поле **pp_ref**) в их дескрипторе устанавливается равным нулю. У дескрипторов страниц, которые невозможно использовать для выделения памяти (BIOS, видеопамять и другие страницы с адресами от **IOPHYSMEM** до начала **EXTPHYSMEM**) или которые заняты ядром и таблицей прерываний реального режима, счётчик ссылок нужно установить равным единице. Их дескрипторы не должны присутствовать в созданном функцией списке свободных страниц.

Функция **page_alloc** должна выделять страницу памяти, удаляя её дескриптор из списка свободных страниц (эта функция не должна изменять счётчик ссылок на эту страницу — ведь сама она не добавляет какие-либо ссылки на неё в таблицы страниц). Функция **page_free** возвращает указанный дескриптор страницы в список свободных страниц. Она вызывается из функции **page_decref**, если счётчик ссылок на страницу достиг нулевого значения.

При выполнении задания используйте схему памяти и константы из файлов **inc/memlayout.h** и **kern/pmap.h**. Обратите внимание на величины **IOPHYSMEM** и **EXTPHYSMEM**.

Для добавления дескриптора в начало списка используйте макрос **LIST_INSERT_HEAD**, а для удаления — макрос **LIST_REMOVE** из файла **inc/queue.h**. Для преобразования физического адреса в адрес дескриптора его страницы и обратно используйте функции **pa2page** и **page2pa**.

Задание 4. В файле **kern/pmap.c** необходимо дописать функции **page_init**, **page_alloc** и **page_free**.

После выполнения этих заданий функция **check_page_alloc** должна сообщить об успехе, а функция **page_check** — ещё нет. Теперь у нас появилась возможность выделять страницы памяти, и мы воспользуемся этим для создания таблиц страничного преобразования. Как можно увидеть, память под текущий каталог страниц сейчас выделена функцией **boot_alloc**.

4.4 Работа с двухуровневой системой таблиц страниц

Как было показано на рисунке 1.1 на стр. 13, в архитектуре x86 при страничной адресации используется двухуровневая система таблиц, таблица первого уровня называется каталогом страниц, второго — таблицей страниц. В файле **inc/mmu.h** находятся макросы для работы с адресами и таблицами, там же находятся комментарии к организации таблицы страниц.

Созданием записей в страничном каталоге будет заниматься функция **pgdir_walk**, которую нам предстоит написать. Если в каталоге таблиц нет ссылки на соответствующую адресу **la** таблицу страниц, то функция должна

создать её при параметре **create**, не равном нулю. При отсутствии ссылки на таблицу страниц и нулевом значении этого параметра функция должна просто вернуть нулевой указатель. При наличии (или после создания) ссылки на таблицу страниц данная функция должна вернуть адрес элемента этой таблицы, соответствующий запрошенному адресу **la**.

Для работы с элементами таблицы страниц (таблицы второго уровня) используйте тип **pte_t** и указатели на него, с элементами каталога страниц — тип **pde_t**. Для получения индекса в каталоге страниц по линейному адресу используйте макрос **PDX** (англ. *Page Directory indeX*). Для получения индекса в таблице страниц по данному линейному адресу используйте макрос **PTX** (англ. *Page Table indeX*).

У существующих вхождений в эти таблицы, согласно документации по i386, установлен бит **PTE_P**. Если в каталоге таблиц нет ссылки на соответствующую адресу **la** таблицу страниц (бит **PTE_P** не установлен), то её нужно создать в случае параметра **create**, не равного нулю. Для выделения памяти под создаваемую таблицу страниц второго уровня используйте вызов **page_alloc**, не забыв увеличить счётчик ссылок в дескрипторе выделенной страницы. Полезны могут быть также функции **page2pa** и **memset**.

У заполняемого элемента каталога страниц установите все биты доступа и существования: для защиты памяти достаточно ограничить доступ на уровне элементов таблицы страниц. Поскольку физический адрес таблицы страниц уже выровнен по границе страницы и содержит нули в трёх младших шестнадцатеричных разрядах, то его можно использовать без изменений при создании элемента каталога страниц. Значение элемента каталога страниц тогда будет создаваться, например, следующим выражением.

```
table_phys_addr | PTE_P | PTE_W | PTE_U
```

Для получения адреса по элементу таблицы страниц или каталога страниц следует использовать макрос **PTE_ADDR**. Он просто выравнивает адрес до 0x1000 — эта же операция обнуляет специальные биты в элементе таблицы или каталога страниц.

Задание 5. Допишите функцию **pgdir_walk** по приведённым выше указаниям. В результате своей работы функция должна вернуть либо **NULL**, либо указатель на элемент таблицы страниц второго уровня, отвечающий адресу **la**. Поскольку в каталоге страниц хранятся физические адреса таблиц страниц, для перехода от них к виртуальным при написании функции пригодится макрос **KADDR**.

4.5 Отображение области виртуальных адресов на физические

К настоящему моменту код ядра и его данные никак не отражены в страничном преобразовании. Для этой цели нам нужно создать функцию **boot_map_segment**[1], которая будет заполнять таблицы страниц для отображения области виртуальных адресов, выровненной на границу страницы, в идущие подряд физические страницы.

Поскольку гарантируется, что размер и начальный адрес отображаемой области кратны размеру страницы, то эта функция реализуется весьма просто. Сначала ей нужно вызывать функцию **pgdir_walk** для каждой отображаемой виртуальной страницы и получить таким образом адрес элемента страницы второго уровня. Затем следует установить значение этого элемента с помощью побитовой операции «ИЛИ» адреса очередной физической страницы с разрешениями в параметре **perm** и с флагом **PTE_P**.

Задание 6. Допишите функцию **boot_map_segment**.

Отметим, что сама эта функция не выделяет физическую память каким-либо образом — она понадобится для отображения кода ядра, его стека и статических данных, а затем и данных, выделенных функцией **boot_alloc**. Физические страницы для этих областей уже выделены и не входят в созданный список свободных страниц.

4.6 Управление списком страниц

К настоящему моменту у нас работает инициализация списка свободных страниц и основные операции с ним: выделение страницы (функция **page_alloc**) и её освобождение (функция **page_free**). В ходе своей работы ОС будет отображать страницу физической памяти на некоторые виртуальные адреса, поэтому нам необходимо реализовать ещё три функции работы со списком страниц.

Задание 7. В файле **kern/pmap.c** необходимо дописать следующие функции:

1) функцию **page_insert** для отображения виртуального адреса на заданный физический;

2) функцию **page_lookup** для поиска дескриптора физической страницы, отображенной по данному виртуальному адресу;

3) функцию **page_remove** для уничтожения отображения виртуальной страницы в физическую.

[1] Несмотря на название, к сегментам в понимании GDT данная функция никакого отношения не имеет.

Функция **page_insert** будет использовать функцию **page_remove**, если данный виртуальный адрес уже отображен на некоторый физический. Функция **page_remove** будет вызывать функцию **page_lookup** для поиска страницы, чтобы уменьшить счётчик ссылок на неё через функцию **page_decref**. При удалении существующего отображения виртуального адреса в физический ОС должна убедится, что в кеше TLB (см. раздел 1.3) не осталось соответствующего элемента. Для этого функция **page_remove** должна вызвать функцию **tlb_invalidate**, передав ей адрес виртуальной страницы.

В ходе работы над указанными функциями вам пригодятся функции **pgdir_walk**, **pa2page**, **page2pa** и известные макросы. После выполнения этого задания функция **page_check** должна сообщить об успехе.

4.7 Переход на плоскую модель памяти

После выполнения всех предыдущих заданий наша система может наконец перейти на плоскую модель сегментов и начать использовать страничное преобразование адресов. Массив **gdt** в файле **kern/pmap.c** содержит новую таблицу GDT.

Для перехода нужно дописать недостающие части функции **i386_vm_init**, внимательно прочитав комментарии в ней. Нам нужно выполнить следующие действия с помощью функции **boot_map_segment**.

1) Отобразите память массива **pages** на адрес **UPAGES** с разрешениями **PTE_U** (доступ на чтение из режима пользователя).

2) Отобразите виртуальные адреса стека ядра (вершина стека — **KSTACKTOP**, размер — **KSTKSIZE**) на физический адрес, соответствующий виртуальному адресу ядра **bootstack**. Не забудьте, что вершина стека — это его старший адрес.

3) Отобразите всю виртуальную память от адреса **KERNBASE** и до конца виртуального пространства на физическую память, начиная с самого её начала. После перехода на плоские сегменты это корректно отобразит виртуальные адреса ядра на его фактическое местоположение.

Задание 8. Внесите указанные изменения и допишите функцию **i386_vm_init**. Проверьте корректность её работы: при запуске системы, в случае успеха, вы получите следующее сообщение.

```
check_boot_pgdir() succeeded!
```

Изучите, как ядро переходит на новую таблицу GDT и как оно включает страничное преобразование (регистры **cr0** и **cr3**). Для корректного перехода на плоскую модель памяти функция **i386_vm_init** временно устанавливает нулевой элемент каталога страниц так, чтобы он позволял корректно

работать ядру при старой, не плоской, таблице GDT. Это нужно для того, чтобы линейный адрес был равен физическому при использовании «старой» версии GDT. Поскольку у нас уже существует таблица страниц, ответственная за отображение в младшие физические адреса, где находится само ядро, то достаточно записать её адрес в нулевой элемент каталога страниц. После перехода на «новую» GDT этот элемент уже не используется для отображения области адресов ядра и поэтому он устанавливается в нулевое значение.

4.8 Контрольные вопросы

Подготовьте ответы на следующие вопросы.

1) Зачем в виртуальном адресном пространстве существует неотображаемая область ниже стека ядра?

2) Почему макросы **PADDR** и **KADDR** работают только с адресами ядра?

3) Зачем нужно использовать макрос **ROUNDUP** в функции **boot_alloc**?

4) Функция **boot_alloc** возвращает физический или виртуальный адрес выделенной памяти?

5) Где находится выделенная функцией **boot_alloc** память в виртуальном пространстве (относительно кода ядра)? В физическом пространстве?

6) Для чего используется список свободных страниц и счетчик ссылок на страницу?

7) Какие страницы памяти не использует страничный менеджер памяти JOS?

8) Какой размер, в нашем случае, имеет страница памяти?

9) Что описывает экземпляр структуры **Page**?

10) Сколько памяти может отобразить одна страница второго уровня?

11) Почему адреса таблиц страниц выравниваются на значение **PGSIZE** и нужно ли это делать?

12) Может ли корректно работать функция **boot_map_segment**, если физический или виртуальный адрес не выровнены по границе страницы? Можно ли это исправить в общем случае?

13) Какие поля присутствуют в дескрипторе физической страницы?

14) Где находится массив **pages** в физической памяти?

15) Зачем ядро ведет список свободных страниц, ведь свободную страницу можно найти по нулевому значению счетчика ссылок на неё?

16) Как выделяется память под элементы списка свободных страниц?

17) Где находится образ массива **pages** в виртуальном адресном пространстве, доступном для записи ядру?

18) Зачем функция **i386_vm_init** дважды меняет элемент **pgdir[0]** при переходе на плоскую модель памяти?

19) Какое максимальное количество физической памяти может быть выделено созданным преобразованием страниц?

20) Обнуляет ли функция **page_alloc** выделяемую память?

21) Где находятся ссылки, подсчёт которых ведется в поле **pp_ref** дескриптора страницы?

22) В каких случаях вызывается функция **tlb_invalidate**? Почему?

23) Какие дескрипторы имеются в GDT, загруженной после подготовки страничного преобразования?

5 Запуск первой прикладной программы

В терминологии JOS процесс, содержащий исполняемый с правами пользователя код, называется «окружением» (англ. *environment*). Виртуальное адресное пространство каждого окружения содержит как код и данные ядра ОС, так и код и данные какой-либо прикладной программы. Будучи учебной ОС, JOS допускает только один поток выполнения на одно адресное пространство и не поддерживает создание дополнительных нитей выполнения. Хотя пока что мы создадим только один процесс пользователя, наша ОС спроектирована как многозадачная система, и в дальнейшем у нас будут выполняться несколько процессов.

Перед продолжением работы нужно переключиться на новую ветку репозитория. Убедитесь командой **git status**, что вы зафиксировали все изменения, при необходимости — зафиксируйте их. Затем выполните следующие команды для переключения в новую ветку.

```
git checkout -b lab3
git pull http://dev.iu7.bmstu.ru/git/workbook_jos lab3
```

После этих команд все зафиксированные вами изменения попадут в новую ветку.

5.1 Дескрипторы процессов

Файл **inc/env.h** содержит основные определения для процессов. В частности, структура **Env** описывает процесс, и в дальнейшем мы будем называть экземпляры этой структуры дескрипторами процессов. Как можно увидеть в файле **kern/env.c**, ядро JOS содержит три следующие глобальные переменные, связанные с процессами.

```
struct Env *envs = NULL;
struct Env *curenv = NULL;
static struct Env_list env_free_list;
```

Указатель **envs** при инициализации системы должен указывать на массив из **NENV** дескрипторов процессов. Список свободных дескрипторов содержится в списке **env_free_list** для простого поиска свободного дескриптора. В этот список включаются неиспользуемые элементы массива **envs**. Указатель **curenv** указывает на дескриптор текущего процесса, в ходе начальной загрузки ядра там находится значение **NULL**.

Задание 9. Измените функцию **i386_vm_init** в файле **kern/pmap.c** так, чтобы она выделяла память под **NENV** элементов массива **envs** для хранения дескрипторов процессов с помощью функции **boot_alloc**. Это можно сделать аналогично тому, как была выделена память для массива **pages**. Массив **envs** должен быть отображен и на адрес **UENVS** (см. **inc/env.h**) с правом доступа **PTE_U**. Убедитесь, что массив дескрипторов процессов корректно отображается страничным преобразованием, при необходимости добавьте это отображение функцией **boot_map_segment**.

Рассмотрим подробнее содержимое дескриптора процесса — структуру **Env**.

```
struct Env {
    struct Trapframe env_tf;
    LIST_ENTRY(Env) env_link;
    envid_t env_id;
    envid_t env_parent_id;
    unsigned env_status;
    pde_t *env_pgdir;
    physaddr_t env_cr3;
};
```

Прокомментируем назначение основных полей дескриптора процесса.

1) Поле **env_tf** предназначено для сохранения значения регистров после перехода из режима пользователя в режим ядра (см. файл **inc/trap.h** и раздел 1.10).

2) Поле **env_link** служит для организации списка неиспользуемых дескрипторов **env_free_list**.

3) Поле **env_id** содержит уникальный идентификатор процесса.

4) Поле **env_parent_id** хранит идентификатор родителя (пока что не используется).

5) Поле **env_status** хранит состояние процесса.

6) Поле **env_pgdir** содержит виртуальный адрес каталога страниц процесса.

7) Поле **env_cr3** хранит физический адрес каталога страниц процесса для последующей загрузки в регистр **cr3**.

Как мы видим, в JOS дескрипторы выделяются также единым массивом, а затем из его элементов создаётся список неиспользуемых дескрипторов, аналогично дескрипторам физических страниц. В случае дескрипторов страниц такое решение, однако, более оправданно — их число в нашей системе фиксировано. В случае дескрипторов процессов такое решение выглядит достаточно натянутым по сравнению с выделением памяти под каждый дескриптор и созданием из них списка.

Выделение памяти сразу под все дескрипторы имеет одно не вполне очевидное следствие: в ядре JOS память для всех критичных структур ядра выделяется в итоге ещё на этапе его инициализации, за исключением таблиц страниц для пользовательской области процессов и каталогов страниц процессов. Это означает, что страничное преобразование общей для всех процессов области ядра не меняется в ходе работы JOS, что позволит тривиально решить проблему её синхронизации между разными процессами.

5.2 Состояния процесса и режимы выполнения

Многозадачная ОС может «отнять» процессор у потока выполнения по двум основным причинам: поток ожидает некоторого события, либо решено выделить процессор другому потоку [1]. Таким образом, у нас есть три основных состояния потока выполнения: выполняется, готов к выполнению, ожидает некоторого события.

Поскольку в нашей системе существует только один поток выполнения на каждый процесс, то поле **env_status** отражает эти состояния и факт занятости дескриптора:

– **ENV_FREE**: данный дескриптор не используется;
– **ENV_RUNNABLE**: процесс готов к выполнению (или даже уже выполняется);
– **ENV_NOT_RUNNABLE**: процесс заблокирован и не готов к выполнению.

Поле не имеет выделенного значения статуса для исполняющегося процесса — для этого можно использовать переменную **curenv**, указывающую на дескриптор выполняющегося процесса.

5.3 Создание процесса пользователя

Теперь нам нужно дописать код в файле **kern/env.c**, необходимый для создания и запуска пользовательского процесса. Поскольку у нас нет файловой системы, то исполняемый файл с программой пользователя будет включаться в ходе сборки системы непосредственно в образ ядра. Не считая этого момента, каждая программа является вполне самостоятельным ELF-файлом.

Сценарий сборки в данном задании создаёт ряд бинарных программ в каталоге **obj/user/**. В сценарии **kern/Makefrag** указано, как встраивать эти программы в образ ядра, используя опцию **-b** у компоновщика, благодаря которой исполняемый файл включается «как есть».

В файле **obj/kern/kernel.sym** видно, что после компоновки в ядре появились символы с названиями следующего вида:

– **_binary_obj_user_hello_start** — адрес начала программы **hello**;
– **_binary_obj_user_hello_end** — адрес конца программы;
– **_binary_obj_user_hello_size** — размер программы.

Благодаря этим символам код ядра может определить, где находятся встроенные в ядро образы программ пользователя.

В файле **kern/env.h** приведены макросы, которые используются в файле **kern/init.c** для загрузки программы в память. Окружение с программой создаётся функцией **env_create**, а начинает выполнятся функцией **env_run**. Функция **env_run** является основным способом перехода из режима ядра в режим пользователя, в главе 8 появится единственное исключение.

Задание 10. Завершите следующие функции в файле **env.c**.

1) Функция **env_init** заполняет начальными значениями все элементы массива **envs** и добавляет их в список **env_free_list**.

2) Функция **env_setup_vm** создаёт каталог страниц для нового процесса и инициализирует в нём «ядерную» часть пространства.

3) Функция **segment_alloc** выделяет процессу память и отображает данную область в неё.

4) Функция **load_icode** анализирует ELF-файл, загружает его содержимое в пользовательскую часть пространства созданного процесса.

5) Функция **env_create** выделяет память с помощью функции **env_alloc** и затем вызывает функцию **load_icode**.

6) Функция **env_run** начинает выполнять процесс, переключившись в режим пользователя.

Заметьте, что аргументы адреса и размера у функции **segment_alloc** не обязаны быть выровнены на размер страницы, поэтому их следует округлять в нужную сторону в самой функции. Эта функция должна отображать память с разрешением записи пользователя в неё.

Функция **load_icode** анализирует ELF-файл, подобно начальному загрузчику. При обработке секций файла нас интересуют только секции с типом **ELF_PROG_LOAD**.

Отметим, что функция **env_run** никогда не возвращает управления, поэтому любой код, размещённый после её вызова, — бесполезен. Перед возвратом в режим пользователя функция **env_run** должна загрузить в регистр **cr3** соответствующее поле из структуры дескриптора процесса, если только текущее значение регистра уже ему не равно: запись в регистр **cr3** сбрасывает содержимое TLB и тем самым снижает производительность.

В ходе написания этих функций вы можете использовать новый формат **%e** функции **cprintf**. Например, следующий код вызовет панику ядра с сообщением «env_alloc: out of memory».

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

5.4 Кадр ловушки

Для перехода в режим пользователя нами используется команда возврата из прерывания, поскольку в момент перехода нужно сменить:

- указатель кода (пару **cs:eip**);
- указатель стека (пару **ss:esp**);
- уровень привилегий (определяется младшими битами регистра **cs**);
- флаги.

Перед инструкцией возврата эта функция должна добиться того, чтобы указатель стека указывал на соответствующие регистры. Для этого можно использовать структуру кадра ловушки **Trapframe**. Эта структура будет рассмотрена в следующей главе, пока обратим внимание на часть нужных нам полей, показанных ниже.

```
struct Trapframe {
    struct PushRegs tf_regs;
    uint16_t tf_es;
    uint16_t tf_padding1;
    uint16_t tf_ds;
    uint16_t tf_padding2;
    uint32_t tf_trapno;
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
};
```

Как вы видите, эта структура, начиная с поля **tf_eip**, физически организована как раз в соответствии с ожиданиями инструкции **iret** о содержимом стека (см. раздел 1.10) перед возвратом из обработчика прерывания.

Структура кадра ловушки не взята из документации Intel, но вполне естественно возникает из состояния стека в момент прерывания и желания разработчиков как можно более простым образом сохранить все регистры для их дальнейшего восстановления. Как уже известно из раздела 1.9, часть этой структуры фактически уже существует в стеке в момент начала работы обработчика прерываний, а в следующем разделе нами как раз будет реализована обработка прерываний. Хотя для прерывания, в терминологии Intel, мы используем термин *interrupt gate*, а не *trap gate*, эта структура названа разработчиками JOS для лаконичности «кадром ловушки».

Структура кадра ловушки используется функцией **env_pop_tf**, которая ставит на неё указатель стека и затем выполняет инструкции извлечения регистров из стека, как показано ниже, вплоть до команды **iret**. Эта функция вызывается функцией **env_run** для возврата управления в режим пользователя.

```
void env_pop_tf(struct Trapframe *tf)
{
    __asm __volatile("movl %0,%%esp\n"
        "popal\n"
        "popl %%es\n"
        "popl %%ds\n"
        /* пропустить tf_trapno и tf_errcode */
        "addl $0x8,%%esp\n"
        "iret"
        : : "g" (tf) : "memory");
}
```

5.5 Использование отладчика

Следующие команды отладчика Bochs заслуживают отдельного упоминания.

– **info idt** — печатает таблицу IDT;
– **vb 0x08:0xf0101234** — установка точки останова по адресу ядра 0xF0101234;
– **vb 0x1b:0x80020** — установка точки останова по адресу пользователя 0x80020.

Далее приведена последовательность вызова функций от начала работы ядра до момента выполнения кода пользователя:

– **start** (файл **kern/entry.S**);
– **i386_init**;

– **cons_init**;
– **i386_detect_memory**;
– **i386_vm_init**;
– **page_init**;
– **env_init**;
– **idt_init** (будет полностью готова в следующей главе);
– **env_create**;
– **env_run**;
– **env_pop_tf**.

Установите точку останова (в отладчике Bochs) на функцию **env_pop_tf**, которая стоит в этой цепочке последней. Пройдя её по шагам, вы должны увидеть переключение в режим пользователя после команды **iret**. Вы увидите затем исполнение кода из файла **lib/entry.S**. Вы должны успешно дойти до инструкции **int $0x30**, которая реализует системный вызов и которая пока что не работает (см. **lib/syscall.c**).

5.6 Контрольные вопросы

Подготовьте ответы на следующие вопросы.

1) Что соответствует в случае JOS основным состояниям процесса (выполнение, готовность, ожидание)?

2) Почему в дескрипторе процесса JOS нет нужды в наличии состояния «выполняется»?

3) Для чего нужен список свободных дескрипторов процессов?

4) Как вы думаете, почему в JOS не используется динамический список дескрипторов процессов, а вместо него под дескрипторы выделяется фиксированный массив?

5) Что делает каждая из функций в последовательности от инициализации ядра до выполнения пользовательского кода процесса?

6) Как происходит переключение в режим пользователя из режима ядра? Что меняется при этом переключении?

7) Какие уровни привилегий (в терминах x86) соответствуют режиму пользователя и режиму ядра в JOS? Чем определяется текущий уровень привилегий?

8) Как JOS загружает из пользовательской программы секцию **.bss** (неинициализированные глобальные данные)?

9) Почему запись в регистр **cr3** его же значения нежелательна при выполнении функции **env_run**?

10) Как копируются при создании процесса таблицы страниц, отображающие область ядра?

11) Как инициализируется вершина стека пользовательской программы?

12) Какая инструкция выполняется первой в режиме пользователя?

13) Для чего в структуре кадра ловушки нужны поля **tf_padding***?

14) В момент перехода в режим пользователя стек ядра содержит несколько кадров стека в результате вызова функций. Должны ли мы сохранить перед переходом в режим пользователя вершину этого стека? Следует ли нам очистить этот стек перед переходом в режим пользователя?

15) Почему в функции **env_setup_vm** необходимо вручную корректировать число ссылок на страницу, используемую как директорию страниц процесса.

16) Как освобождается память, занятая кодом и данными программы?

17) Как освобождается память, отведенная под таблицы страниц пользовательской части процесса?

6 Прерывания и системные вызовы

Процесс должен перейти из режима пользователя в режим ядра, чтобы выполнить некоторые действия, например, выполнить ввод-вывод или просто завершить свою работу. После выполнения некоторых из этих действий процесс в итоге возвращается обратно в режим пользователя. Этот механизм называется *системным вызовом* [1] и в данной главе мы добавим его в нашу систему, для чего сначала нам нужно реализовать обработку прерываний ядром.

Обработка программных исключений и аппаратных прерываний нам понадобится для решения следующих задач:

– перехват страничного исключения, в том числе для реализации экономного клонирования процессов;
– перехват прочих программных исключений с целью удаления ошибочных процессов;
– обработка прерывания таймера при реализации вытесняющей многозадачности (глава 9);
– реализация системного вызова.

Поскольку в нашей системе нет поддержки файловых систем и полноценной поддержки обмена данными с устройствами, то для организации ввода-вывода прерывания нами, к сожалению, не будут использоваться.

6.1 Прерывания и исключения в JOS

Исключения и прерывания архитектуры x86 изложены в разделе 1.9 на стр. 21, далее мы остановимся на основных моментах реализации обработки прерываний в JOS.

В JOS все исключения и прерывания обрабатываются в режиме ядра, на нулевом уровне привилегий. С целью упрощения в нашей системе будут запрещены прерывания в режиме ядра, а исключения в режиме ядра будут рассматриваться как фатальная ошибка операционной системы.

Таким образом, прерывания и обрабатываемые исключения в нашей системе происходят только в режиме пользователя. Для обработки прерывания ядро ОС должно сообщить процессору, где находятся обработчики прерываний, и как именно следует сменить указатель на вершину стека при возникновении прерывания.

Для переключения стека при защищённой передаче управления в архитектуре x86 используется сегмент состояния задачи (TSS, стр. 22). Хотя эта структура данных может потенциально использоваться для нескольких задач, в том числе сохранения значений регистров процесса, в JOS, как и

во многих современных ОС, она будет применяться только для задания указателя стека, на который процессор должен переключиться при переходе процесса из режима пользователя в режим ядра. Для этого в структуре **Taskstate** (файл **inc/mmu.h**), соответствующей аппаратной организации TSS, будут использоваться поля **ts_ss0** и **ts_esp0**. Ни одно другое поле структуры **Taskstate** нам не понадобится. Поле **ts_ss0** будет иметь значения селектора данных ядра **GD_KD**, а поле **ts_esp0** — значение начальной вершины стека ядра (**KSTACKTOP**).

Используемая нами структура TSS содержится в глобальной переменной **ts**. Ссылка на неё должна быть помещена в таблицу GDT, а соответствующий селектор должен быть затем использован для инструкции **ltr** загрузки регистра **tr**. Поскольку в GDT уже имеется зарезервированный для TSS дескриптор, то JOS выполняет эти операции следующим образом (см. функцию **idt_init**).

```
gdt[GD_TSS >> 3] = SEG16(STS_T32A, (uint32_t) (&ts), sizeof(ts), 0);
gdt[GD_TSS >> 3].sd_s = 0;
...
ltr(GD_TSS);
```

6.2 Настройка обработки прерываний и исключений

К настоящему моменту наша система может перейти в режим пользователя и выполнять код прикладной программы, но мы не можем вернуться назад в режим ядра. Для исправления ситуации нам необходимо реализовать базовую обработку исключений и системных вызовов, чтобы ядро могло опять получить управление. В этом разделе мы расширим JOS так, чтобы обрабатывать исключения с номерами от 0 до 31. Отметим, что некоторые из исключений в диапазоне 0–31 Intel определяет как зарезервированные и не используемые в настоящий момент.

В следующем разделе мы также научим JOS обрабатывать программное прерывание 0x30, которое в JOS используется как номер прерывания системного вызова. Этот номер был выбран без каких-либо особых оснований, но он должен быть больше 31. В дальнейшем мы дополним JOS обработкой аппаратного прерывания таймера.

Заголовочные файлы **inc/trap.h** и **kern/trap.h** содержат важные определения, связанные с защищённой передачей управления, с которыми вам необходимо будет познакомиться. Файл **kern/trap.h**[1] содержит определения, которые используются только ядром, в то время как сопутствующий

[1]Как видно по названию файлов, авторы JOS предпочитают использовать термин «ловушка» как синоним процесса обработки исключений и прерываний или кода, реализующего такой процесс.

файл **inc/trap.h** содержит общие определения, которые могут также быть полезными для программ пользовательского режима и системных библиотек.

Поскольку в момент прерывания ни в регистрах, ни на стеке процессором не сохранён его номер, то для его получения системой у каждого исключения или прерывания должен быть свой собственный обработчик. Эти обработчики должны быть написаны на ассемблере и будут находиться в файле **trapentry.S**. Используемая таблица IDT будет содержать их адреса.

Для передачи управления коду, написанный на языке C, начальный обработчик прерывания должен сохранить значения всех регистров и номер прерывания в некоторой структуре. Для этой цели в JOS используется структура кадра ловушки **Trapframe**. Заполнив эту структуру прямо «на стеке», мы без всяких дополнительных усилий включим в неё сохранённые адреса возврата из стека. Остальные поля должны быть заполнены до вызова первой функции по правилам языка C нашим ассемблерным кодом, для чего нам пригодятся инструкции типа **push**. Каждый обработчик должен в итоге создать структуру **Trapframe** прямо на стеке, а затем вызвать функцию **trap**, передав ей адрес этой структуры. Таким образом, в данном случае «кадр ловушки» означает сохранённое до обработки прерывания состояние всех регистров прерванного процесса, восстанавливаемое затем при возврате в режим пользователя.

Функция **trap** вызывается по соглашениям языка C и находится в файле **trap.c**. Её параметр — указатель на **Trapframe** — должен быть положен в стек перед инструкцией **call trap**. Отметим, что эта структура уже лежит на стеке, но для большей гибкости мы передадим функции указатель на него по правилам языка C.

С вызова функции **trap** при обработке прерывания начинается последовательность вызовов функций ядра по соглашениям языка C. По этой причине перед этим вызовом следует обнулить регистр **ebp**: это позволит во время отладки при необходимости вывести на экран корректный перечень кадров стека в мониторе ядра (функция **mon_backtrace**).

Как видно по структуре **Trapframe**, приведённой выше, единственным её полем, зависящим от обработчика, является номер прерывания (поле **tf_trapno**). Таким образом, разумно выделить общую для всех обработчиков часть, которая находится после метки **_alltraps** в файле **trapentry.S**. Эта часть завершается вызовом функции **trap**. Часть, специфичная для обработчика, будет генерироваться специальным макросом, поскольку единственное её назначение — помещение в стек номера прерывания.

Задание 11. Отредактируйте файл **trapentry.S** и **trap.c** и реализуйте функциональность, описанную выше. Макросы **TRAPHANDLER** и **TRAPHANDLER_NOEC** в файле **trapentry.S** должны помочь вам при ге-

нерации кода обработчиков с кодом ошибки и без него соответственно. Определения с префиксом **T_*** в файле **inc/trap.h** и макросы в файле **inc/mmu.h** также могут быть полезны.

Вам нужно будет добавить точки входа в файле **trapentry.S**, используя макросы **TRAPHANDLER** и **TRAPHANDLER_NOEC** для каждой ловушки, определённой в файле **inc/trap.h**. Кроме того, вам нужно написать код, расположенный по метке **_alltraps**. Также вам будет нужно модифицировать функцию **idt_init** для инициализации таблицы **idt**, чтобы она указывала на каждую точку входа, определённую в **trapentry.S**. Для этого используйте макрос **SETGATE**.

В макросе **SETGATE** второй параметр определяет, будет ли создан шлюз прерывания или шлюз исключения. Как было отмечено в разделе 1.9, при выполнении обработчика шлюза прерывания будут запрещены аппаратные прерывания. Мы добавим чуть позже обработку прерываний в нашу систему, но она спроектирована, исходя из очень серьёзного ограничения: наше ядро не допускает повторный вход в себя (т. е. оно не реентерабельно), и поэтому прерывания в нём должны быть запрещены. Это означает, что все шлюзы, которые вы создадите в этом задании, сразу должны иметь тип шлюза прерывания, чтобы это не пришлось менять в дальнейшем (даже если мы, формально, обрабатываем программные исключения!). Отметим, что мы не можем решить проблему запрета прерываний с помощью инструкции **cli** в начале обработчика — это будет типичным примером гонок.

В файле **trapentry.S** по метке **_alltraps** ядро должно выполнить следующие шаги.

1) Поместить в стек значения регистров (общего назначения, **es** и **ds**) в порядке, определённом в структуре **Trapframe**.
2) Загрузить значение селектора **GD_KD** в регистры **ds** и **es**.
3) Выполнить инструкцию **pushl %esp** для передачи указателя на структуру **Trapframe** (заполненную прямо на стеке!) как аргумента в функцию **trap**.
4) Выполнить инструкцию **call trap**.

Отметим, что регистр **es**, казалось бы, вряд ли будет использоваться кодом ядра, и его можно не устанавливать при входе в ядро. Однако, это не так: машинные инструкции обработки массивов в x86 (например, **stos**) используют именно его для адресации массива, поэтому в него всегда следует заносить значение корректного селектора данных!

Для сохранения регистров разумно использовать инструкцию **pushal** — поля структуры **Trapframe** специально выбраны так, чтобы соответствовать сохранённым в стеке регистрам общего назначения.

Таким образом, создание структуры кадра ловушки на стеке становится практически тривиальным. Как можно отметить, ваш код будет во многом обратным коду функции **env_pop_tf**.

При изменении функции **idt_init** (файл **trap.c**) и использовании макросов **SETGATE** в качестве значения уровня привилегий дескриптора (DPL) следует использовать нулевое значение для всех обработчиков, кроме обработчика точки останова и системного вызова. Это запретит вызов всех прочих обработчиков инструкцией **int** — при попытке это сделать случится исключение ошибки защиты. Если этого не сделать, то злонамеренная программа сможет явно вызвать, например, обработчик страничного исключения инструкцией **int**, что может вывести ядро ОС из строя. Примером такой программы является **softint**.

6.3 Возврат управления в режим пользователя

В начале функции **trap** видно, что она сохраняет регистры из структуры **Trapframe** (которую ей создал код из файла **trapentry.S**) в поле **env_tf** сохранённого контекста дескриптора текущего окружения, как показано ниже.

```
void trap(struct Trapframe *tf)
{
    if ((tf->tf_cs & 3) == 3) {
        curenv->env_tf = *tf;
        tf = &curenv->env_tf;
    }
...
```

Копирование контекста в поле дескриптора происходит только в том случае, если прерывание произошло в режиме пользователя — впрочем, в иных случаях, как мы помним, обработка прерывания может закончиться только аварийным завершением работы («паникой») ядра. После того, как управление передано в функцию **trap**, эта функция обрабатывает исключение/прерывание или передает управление специальному обработчику через функцию **trap_dispatch**.

Функция **trap** не возвращает управление в файл **trapentry.S** — вместо этого она вызывает уже изученную нами функцию **env_run**. Поскольку к этому моменту в поле **curenv->env_tf** сохранён контекст на момент прерывания, то достаточно передать в функцию **env_run** указатель на текущий дескриптор процесса (**curenv**).

Проверьте работу своего кода тестовыми программами из директории **user**, которые вызывают исключения перед тем, как сделать какие-либо

системные вызовы, например **user/divzero**. К этому моменту у вас должны предсказуемо выполняться тестовые программы **divzero**, **softint** и **badsegment** — ошибки в их работе должны обнаруживаться ядром системы.

6.4 Обработка страничного исключения

Обращение к отсутствующей странице или попытка записи в страницу только для чтения вызывает прерывание с номером **T_PGFLT**. Это событие называется страничным отказом или страничным исключением.

В регистре **cr2** при этом будет сохранён адрес памяти, обращение по которому вызвало исключение. В файле **trap.c** уже создана функция **page_fault_handler** для обработки страничных отказов. Сейчас она всегда уничтожает вызвавший исключение процесс, что не совсем верно — если страничное исключение случилось в режиме ядра, то, возможно, сама пользовательская программа в этом и не виновата. Тем не менее, на данном этапе нас устроит и такой обработчик страничного исключения — далее мы исправим эту неточность.

Задание 12. Измените функцию **trap_dispatch** так, чтобы для обработки страничных отказов использовалась функция **page_fault_handler**.

После выполнения этого задания должны успешно проходить тесты **faultread**, **faultreadkernel**, **faultwrite** и **faultwritekernel**.

6.5 Системные вызовы

В ранних процессорах x86, до Pentium II, для системного вызова необходимо было использовать прерывание, затем стало возможно использовать и особую машинную инструкцию **sysenter** как альтернативу прерыванию. Для лучшей переносимости наша система будет использовать старый подход: для системного вызова выделено прерывание с номером 0x30 (константа **T_SYSCALL**).

Пользовательскому коду необходимо передать ядру номер системного вызова и его параметры. Для передачи номера будем использовать регистр **eax**, для передачи параметров — регистры **edx**, **ecx**, **ebx**, **edi** и **esi**, в указанном порядке. В нашей системе не будет системных вызовов более чем с пятью параметрами, поэтому нет нужды вводить более сложные соглашения о передаче данных между пользователем и ядром.

Результат системного вызова мы вернём в пользовательскую программу в регистре **eax**. Соглашения об используемых при системных вызовах регистрах зависят от ОС, но наше решение достаточно типично для ОС на платформе x86.

Изучите функцию **syscall** в файле пользовательской библиотеки **lib/syscall.c** и файл **inc/syscall.h** для понимания процесса системного вызова с точки зрения пользовательской программы.

Задание 13. Добавьте в ядро обработку прерывания с номером **T_SYSCALL**, которая будет приводить к вызову функции **syscall** в файле **kern/syscall.c**. Вам нужно будет изменить файлы **kern/trapentry.S** и **kern/trap.c** (функция **idt_init**). Кроме того, вам нужно изменить функцию **trap_dispatch**, которая теперь должна вызывать при обработке прерывания системного вызова функцию **syscall** с соответствующими аргументами. Наконец, нужно создать саму функцию **syscall** в файле **kern/syscall.c**.

Обратите внимание, что функция **syscall** должна вызываться из функции **trap_dispatch** с правильными аргументами, взятыми из указанных выше регистров, а её результат в итоге должен передаваться пользователю через регистр **eax**.

В файле **kern/syscall.c** уже реализован ряд системных вызовов, которым следует передавать управление из функции **syscall** в соответствии с полученным номером вызова:

– функция **sys_cputs** выводит строку на экран;
– функция **sys_cgetc** считывает символ с клавиатуры;
– функция **sys_getenvid** возвращает идентификатор текущего процесса;
– функция **sys_env_destroy** завершает указанный процесс.

Константы для номеров системных вызовов определены в **inc/syscall.h** Если номер системного вызова неверен, функции **syscall** следует вернуть значение **-E_INVAL**.

После выполнения задания вы должны иметь возможность выполнить программу **user/hello**, которая должна напечатать сообщение и затем вызвать исключение отсутствия страницы.

6.6 Старт и завершение программы пользователя

Программы пользователя начинают работу с кода в файле **lib/entry.S**. Здесь, в частности, определяется массив **envs**, указывающий на адрес **UENVS**: как мы помним, ядро JOS специально отображает по этому адресу массив дескрипторов процессов с возможностью чтения его пользователем. Из этого файла затем будет вызвана функция **libmain** в файле **lib/libmain.c**. Эта функция должна присвоить указателю **env** значение адреса дескриптора текущего процесса, взятого из массива **envs**. Затем вызывается функция **umain** — с точки зрения прикладного программиста, это и будет «первая» функция программы.

В программе **hello** функция **umain** находится в файле **user/hello.c**. Обратите внимание, что при запуске она пытается использовать значение **env->env_id** и пока ей это не удаётся, поскольку указатель **env** содержит нулевой адрес.

Задание 14. Добавьте в функцию **libmain** код, который будет устанавливать указатель **env** на соответствующий текущему окружению элемент в массиве **envs**.

Изучите содержимое файла **inc/env.h** и используйте вызов **sys_getenvid**. Обратите внимание, что функция **sys_getenvid** возвращает идентификатор процесса, в котором младшие биты равны номеру дескриптора процесса в массиве **envs**. Для выделения этого номера из идентификатора процесса используйте макрос **ENVX**.

Возможной причиной проблем при выполнении задания может быть то, что вы не отобразили массив **envs** на адрес **UENVS**, как это требовалось ранее, или сделали это некорректно. В случае успеха вы должны увидеть следующие строчки при запуске ядра (программа **user/hello** запустится автоматически после его инициализации).

```
hello, world
i am environment 00000800
```

После этого программа завершится, используя функцию **exit** в файле **lib/exit.c**, которая приведёт к выполнению функции ядра **sys_env_destroy** и завершению работы единственного окружения. После завершения работы всех программ пользователя ядро JOS перейдёт в режим монитора.

Для лучшего понимания того, во что компилируется ассемблерная вставка вызова прерывания в пользовательской программе, следует изучить файл **obj/user/hello.asm** с ассемблерным листингом.

6.7 Защита ядра от некорректных указателей

Часть параметров системного вызова могут быть адресами буферов программы, куда ядро должно записать некоторые данные, или откуда оно должно их считать. Поскольку ядро и программа находятся в одном адресном пространстве, то любой корректный указатель пользовательской программы является таковым же и с точки зрения ядра.

При ряде системных вызовов передаётся пара значений, которые ядро трактует как указатель на начало буфера и его длину. Это может привести к ошибке в работе ОС, если только оно не проверит корректность этих значений, используя информацию из таблиц страниц. Пользовательские программы могут передавать ядру при системном вызове некорректные указатели двух видов.

1) Указатели на память, обращение к которой даже в режиме ядра вызовет неисправимое страничное исключение (например, к таким относится нулевой указатель).

2) Указатель на память, обращение к которой возможно только в режиме ядра — таким образом приложение-злоумышленник может пытаться испортить данные ядра или считать конфиденциальные данные из ядра.

Для JOS сейчас корректна передача системному вызову указателя на буфер, который целиком расположен в памяти, доступной в режиме пользователя для чтения (как мы увидим позже, наличия прав на запись может не быть, например, в случае экономного клонирования процессов).

Нам нужно улучшить защиту ядра от возможной передачи некорректных адресов буферов при системном вызове. Для этого нужен механизм проверки полученных системным вызовом указателей на корректность, завершающий текущий процесс в случае обнаружения проблем. Если после включения этого механизма в режиме ядра всё-таки произойдёт страничное исключение, то нам остаётся только аварийно завершиться («запаниковать») — поскольку в JOS нет механизма выгрузки страниц из памяти на диск, то причиной такого исключения на данном этапе следует признать ошибку в самом ядре.

Задание 15. Измените обработчик страничного исключения в файле **kern/trap.c** следующим образом: если исключение произошло в режиме ядра, то ядру следует аварийно завершить свою работу вызовом макроса **panic**.

Для определения того, случилось ли исключение в режиме пользователя или в режиме ядра, используйте, например, два младших бита в поле **tf_cs**: в них содержится уровень привилегий кода, вызвавшего исключение. В дальнейшем мы добавим более сложную обработку этого исключения при реализации клонирования процессов.

Задание 16. Допишите функцию **user_mem_check** в файле **kern/pmap.c**. Измените код в файле **kern/syscall.c** так, чтобы он проверял получаемые системным вызовом аргументы с использованием этой функции.

Проверка указателей сейчас актуальна для системного вызова вывода строки на экран. Созданный механизм должен защитить ядро от некорректных указателей обоих видов. Для проверки защиты от ошибочных указателей на область пользователя у нас есть прогорамма **buggyhello**, а программа **evilhello** передаёт указатель на область ядра.

Измените файл **kern/init.c** так, чтобы ядро запускало программу **user/buggyhello** вместо программы **user/hello**. При верно выполненном за-

дании процесс должен быть уничтожен примерно со следующими сообщениями.

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

Теперь измените файл **kern/init.c** так, чтобы он запускал программу **user/evilhello**. При запуске системы этот процесс должен быть уничтожен, а ядро не должно аварийно завершиться. Примерный вид ожидаемых сообщений приведён ниже.

```
[00000000] new env 00001000
[00001000] user_mem_check assertion failure for va f0100020
[00001000] free env 00001000
```

В заключение отметим, что реализованная нами проверка указателей на область пользователя подходит только для операционных систем без реализации подкачки памяти, известной также как «свопинг» [1]. В реальных системах вполне валидный пользовательский буфер вполне может отсутствовать в физической памяти по причине отправки данных пользователя в область подкачки. Попытка обращения к такому буферу вызовет вполне обычное страничное исключение в ходе системного вызова.

6.8 Контрольные вопросы

Подготовьте ответы на следующие вопросы.

1) Где находится адрес точки входа в обработчик прерывания? Адрес вершины стека, им используемого?

2) Какой уровень привилегий следует использовать для дескриптора прерывания системного вызова? Для исключения отсутствия страницы?

3) Почему все прерывания у нас имеют отдельные функции-обработчики? Что не удастся реализовать, если у них будет единый обработчик, указанный во всех элементах IDT?

4) Какой адрес сохраняется в регистре **cr2**, линейный или физический?

5) Для чего наша система использует TSS?

6) Какая часть структуры **Trapframe** заполняется процессором в момент прерывания?

7) Будет ли перед вызовом обработчика сменён стек, если исключение возникнет при работе ядра JOS?

8) Для чего в JOS используется структура кадра ловушки?

9) Структура кадра ловушки предполагает наличие в стеке кода ошибки. В какой момент в стек будет положено фиктивное значение кода ошибки в случае, если когда это не делает процессор?

10) Как именно структура кадра ловушки копируется с аппаратного стека в поле в дескрипторе процесса?

11) Что передается в качестве фактического аргумента в функцию **trap**? Можно ли передать в неё не указатель, а саму структуру?

12) Что будет, если программа пользователя сможет напрямую вызывать страничное исключение явным образом (это пытается сделать программа **softint**)?

13) Наша система меняет при входе в ядро не только регистр **ds**, но и регистр **es**. По какой причине она это делает?

14) Почему в кадре ловушки нет полей для регистров **fs** и **gs**?

15) Почему функция **trap** проверяет значение указателя **curenv** в утверждении? Что может случиться без этой проверки (рассмотрите случаи до и после добавления проверки места исключения)?

16) Почему для создания корректного списка кадров стека в данной работе необходимо обнулить регистр **ebp**, ведь это уже было сделано при инициализации системы (глава 3)?

17) Как процесс переходит в режим ядра?

18) Какое максимальное количество 32-битных параметров можно передать при системном вызове через регистры процессора i386? Какие регистры будут задействованы в нашем случае и в каком порядке?

19) Что следовало бы сделать, чтобы задействовать для передачи параметров системному вызову ещё и регистр **ebp**?

20) Является ли регистр для передачи номера системного вызова закреплённым на аппаратном уровне? Является ли закреплённым номер прерывания системного вызова?

21) Как именно номер системного вызова передаётся от пользовательской программы до момента его проверки в ядре?

22) В чём разница между переменными **envs** в памяти ядра и в памяти пользователя?

23) В чём разница и что общего между функциями **cprintf** ядра и пользователя?

24) При системном вызове мы переходим в режим ядра из режима пользователя. Указывают ли полученные от прикладной программы указатели после этого на те же самые данные?

25) Как реализована зашита от «вредительских» указателей?

26) Можно ли решить проблему передачи некорректных указателей, просто анализируя значение адресов начала и конца переданного при системном вызове буфера?

27) От передачи каких указателей можно было бы защититься путём изменения обработки страничного исключения таким образом, чтобы исключение во время системного вызова уничтожало выполняющий системный вызов процесс?

28) Приведите конкретные примеры значений некорректных указателей обоих видов для нашей системы.

29) Есть ли необходимость в сравнении полученного системным вызовом адреса и значения **ULIM**, если мы всё равно затем проверяем права доступа к памяти?

30) Могут ли в данный момент пользовательские программы в нашей системе использовать регистры процессора с плавающей точкой?

31) Почему в функции **env_alloc** генерируются возрастающие, до переполнения, идентификаторы процессов, хотя, казалось бы, можно считать идентификатором порядковый номер дескриптора?

7 Создание процессов и кооперативная многозадачность

К настоящему моменту в нашей ОС может существовать лишь единственное пользовательское окружение, создаваемое самим ядром. В этой части работы мы реализуем несколько новых системных вызовов ядра, чтобы предоставить возможность создавать новые процессы коду, исполняемому в режиме пользователя. Также мы реализуем кооперативную многозадачность, позволяющую ядру переключаться с выполнения одного процесса на выполнение другого, если текущий процесс добровольно освобождает процессор или завершает своё выполнение [1].

Перед продолжением работы нужно переключиться на новую ветку репозитория. Убедитесь командой **git status**, что вы зафиксировали все изменения, при необходимости — зафиксируйте их. Затем выполните следующие команды для переключения в новую ветку.

```
git checkout -b lab4
git pull http://dev.iu7.bmstu.ru/git/workbook_jos lab4
```

После этих команд все зафиксированные вами изменения попадут в новую ветку. Не исключено, что в ходе слияния веток в файле **kern/init.c** обнаружится конфликт — вам следует отредактировать этот файл, устранив конфликт, и зафиксировать изменения.

7.1 Бездействующий процесс

С данного момента первый процесс с дескриптором в структуре **envs[0]** будет считаться бездействующим (англ. *idle*) процессом, всегда исполняющим код из файла **user/idle.c**. Задача этого процесса — всегда пытаться передать управление другому процессу, используя системный вызов **sys_yield**. Бездействующий процесс получает управление только тогда, когда нет иных кандидатов на выделение времени процессора. Введение такого процесса позволяет решить задачу, чем же системе занять процессор, если нет никаких процессов-кандидатов на выполнение — теперь такой кандидат есть всегда. После вызова **sys_yield** наш бездействующий процесс вызывает монитор ядра — исключительно с целью отладки: поскольку передача управления бездействующему процессу в нашей системе означает, что других выполнимых задач нет, то эта ситуация наверняка свидетельствует об ошибке.

7.2 Кооперативная многозадачность и переключение процессов

Первое, что нужно сделать для создания многозадачной системы — изменить ядро создаваемой ОС так, чтобы оно могло не только исполнять процесс с дескриптором в структуре **envs[0]**, но и могло переключаться между несколькими процессами с дескрипторами в массиве **envs**.

Кооперативная многозадачность в нашей системе должна работать следующим образом. Функция **sched_yield** в новом файле **kern/sched.c** будет отвечать за выбор следующего исполняемого процесса. Эта функция последовательно, по кругу, ищет в массиве **envs** кандидата на выполнение, начиная сразу после предыдущего выполнявшегося процесса или с начала массива, если запущенного процесса пользователя ещё нет. Кандидатом на выполнение будет первый найденный процесс с состоянием **ENV_RUNNABLE** (см. файл **inc/env.h**). Отметим, что этот способ, конечно, весьма неэффективен с точки зрения времени на поиск кандидата на выполнение: было бы лучше иметь список готовых к исполнению процессов, и на исполнение выбирать его следующий элемент.

Функция **sched_yield** должна «знать», что в структуре **envs[0]** находится дескриптор специального бездействующего процесса, и выбирать его в случае отсутствия иных кандидатов на исполнение — и только в этом случае.

Функция **sched_yield** затем вызывает функцию **env_run**, чтобы переключиться на выбранный процесс. Напомним, что функция **env_run** не возвращает управление в вызывающую функцию.

В этом задании нужно реализовать функционал нового системного вызова **sys_yield**, который может вызываться пользовательским процессом, чтобы выполнить функцию ядра **sched_yield** и, таким образом, добровольно передать процессор другому процессу.

Всякий раз, когда ядро переключается с одного процесса на другой, оно должно быть уверено, что регистры старого процесса сохранены и смогут быть правильно восстановлены позднее. Для сохранения регистров в дескрипторе процесса есть поле **env_tf**, и на текущий момент оно должно корректно заполняться при начале обработки прерывания и использоваться при возврате из прерывания. Поскольку это один из ключевых моментов работы системы, то вам следует хорошо понять, где и как это происходит.

Задание 17. Реализуйте описанный выше алгоритм планирования в функции **sched_yield**. Не забудьте изменить функцию **syscall** для обработки нового системного вызова **sys_yield**.

Измените файл **kern/init.c** так, чтобы создавалось два или более процессов, выполняющих программу, созданную из файла **user/yield.c**. Вы смо-

жете пронаблюдать за переключением этих процессов друг на друга и увидеть примерно следующие сообщения.

```
Hello, I am environment 00001001.
Hello, I am environment 00001002.
Back in environment 00001001, iteration 0.
Back in environment 00001002, iteration 0.
Back in environment 00001001, iteration 1.
Back in environment 00001002, iteration 1.
...
```

7.3 Системные вызовы для создания процесса

Хотя наше ядро теперь способно исполнять несколько пользовательских процессов и переключаться между ними, оно всё ещё ограничено исполнением только тех процессов, которые создаются при инициализации системы. Сейчас мы реализуем дополнительные системные вызовы, которые позволят самим пользовательским процессам создавать новые пользовательские процессы. Таким образом, у нас появится отношение «родительский-дочерний» на множестве процессов, а в структуре дескриптора процесса теперь будет задействовано поле идентификатора родителя.

Эти системные вызовы будут предоставлять относительно примитивные, по сравнению с известной функцией **fork** [5], возможности для создания нового пользовательского процесса. Позже с помощью них мы реализуем и вызов, полностью подобный вызову **fork** в UNIX. Далее перечислены новые системные вызовы, которые нужно создать в этой главе.

Системный вызов **sys_exofork** создает новый «пустой» процесс: его адресное пространство пусто в пользовательской части и он не является исполняемым. Новый процесс имеет то же состояние всех регистров, что и его родительский процесс на момент вызова **sys_exofork**. Для процесса-родителя этот системный вызов возвращает в случае успеха значение **envid_t** нового созданного процесса. Если создать новый процесс не удалось (например, закончились свободные дескрипторы в массиве **envs**), то системный вызов **sys_exofork** вернёт отрицательный код ошибки.

В дочернем процессе этот вызов возвращает 0, так же как и обычный вызов **fork**. Поскольку дочерний процесс помечается как неисполняемый и имеет состояние **ENV_NOT_RUNNABLE**, то этот вызов не возвращает управление в дочернем процессе до тех пор, пока родитель не пометит дочерний процесс как исполняемый с помощью системного вызова **sys_env_set_status**.

Системный вызов **sys_env_set_status** устанавливает состояние указанного процесса в **ENV_RUNNABLE** или **ENV_NOT_RUNNABLE**. Обычно этот системный вызов используется для того, чтобы пометить созданный

дочерний процесс как готовый к запуску, когда его адресное пространство было полностью сформировано.

Системный вызов **sys_page_alloc** выделяет процессу страницу физической памяти и отображает в неё указанный виртуальный адрес указанного процесса. Адрес должен соответствовать пользовательской части адресного пространства. Этот вызов может менять только виртуальное адресное пространство либо самого вызвавшего процесса, либо его дочернего процесса, иные варианты должны быть запрещены.

Системный вызов **sys_page_map** отображает страницу памяти по указанному виртуальному адресу (**dstva**) одного процесса (**dstenvid**) в ту же физическую страницу, куда отображена указанная страница памяти (**srcva**) другого процесса (**srcenvid**). Также она устанавливает указанные разрешения доступа к памяти.

Содержимое физической страницы при этом не копируется — оба виртуальных адреса, (**dstva** и **srcva**) будут просто указывать на одну и ту же страницу физической памяти.

Системный вызов **sys_page_unmap** удаляет связь виртуальной страницы с физической по заданному виртуальному адресу для заданного процесса.

Для всех описанных выше системных вызовов, принимающих идентификатор процесса, ядро JOS должно поддерживать соглашение, что значение идентификатора, равное нулю, соответствует текущему процессу. Это соглашение реализуется с помощью функции **envid2env** в файле **kern/env.c** и макроса **ENVX**.

Для лучшего понимания назначения создаваемых системных вызовов следует изучить исходники программы **user/dumbfork**. Он содержит, как следует из названия, очень примитивную реализацию функции, подобной функции **fork**: функция **dumbfork** использует приведённые выше системные вызовы, чтобы создать и запустить дочерний процесс с полной физической копией своего собственного адресного пространства, включая код, стек и статические данные. Для копирования кода и статических данных она копирует в цикле все страницы от адреса **UTEXT** (начало кода) до адреса переменной **end** — она создается при сборке программы и находится в конце статических данных. Для копирования стека достаточно скопировать одну страницу, полученную из адреса любой локальной переменнной, поскольку для стека программы JOS выделяет одну страницу. Конечно, это довольно спорный способ копирования адресного пространства, но как временное (до следующей главы) решение он годится.

Два процесса затем переключаются между собой при помощи вызова **sys_yield**, как и в предыдущем задании. Родительский процесс завершается после 10-ти итераций, тогда как дочерний процесс — после 20-ти.

Задание 18. Реализуйте описанные выше системные вызовы в файле **kern/syscall.c**. Вам потребуется использовать различные функции из файлов **kern/pmap.c** и **kern/env.c**.

Когда вы вызываете функцию **envid2env**, передавайте единицу в параметре **checkperm**. Убедитесь, что вы сделали проверку правильности аргументов системного вызова и возвращаете **-E_INVAL** в случае ошибки. Протестируйте своё ядро при помощи программы **user/dumbfork**: она должна работать корректно.

7.4 Контрольные вопросы

Подготовьте ответы на следующие вопросы.

1) В чём смысл существования бездействующего процесса? Можно ли обойтись без него в многозадачных системах?

2) Как созданный планировщик задач ищет кандидата на выполнение?

3) Есть ли в JOS явная очередь готовых к выполнению процессов?

4) Что будет, если вызову **sys_page_alloc** будет передан адрес памяти ядра?

5) Копирует ли системный вызов **sys_page_map** соответствие адресов в области памяти ядра? Почему?

6) Какие ограничения должен накладывать системный вызов **sys_page_alloc** на идентификаторы процессов для того, чтобы два произвольных несвязанных процесса не могли вмешаться в работу друг друга?

7) Должен ли системный вызов **sys_page_map** проверять как-либо идентификатор процесса, адресное пространство которого он меняет? Где это происходит?

8) Как работает программа **user/dumbfork** и каковы основные недостатки такого подхода к клонированию процессов?

9) Функция **env_run** при переключении процессов загружает новое значение в регистр **cr3**, что приводит к смене текущего адресного пространства. Почему же возможно обращаться к структурам ядра (например, по указателю на аргумент **e**) как перед изменением регистра **cr3**, так и после?

10) В момент создания дочернего процесса структура кадра ловушки в его дескрипторе заполняется значениями кадра ловушки родителя. Копирование каких регистров критично для корректного выполнения дочернего процесса, а каких — несущественно?

11) Могут ли в данный момент пользовательские программы в нашей системе использовать регистры процессора с плавающей точкой?

12) Как копируется стек для дочернего процесса?

13) Когда в дочернем процессе устанавливается значение переменной **env**?

14) Корректно ли работает функция **dumbfork**, если процесс до этого вызывал **sys_page_map**?

8 Эффективное клонирование процессов

В этом задании нам предстоит реализовать более экономичное и эффективное клонирование процессов системным вызовом **fork**, используя копирование при записи [1].

Для реализации копирования при записи мы должны временно пометить страницы обоих процессов как доступные только для чтения и копировать их в случае страничного исключения, вызванного попыткой записи.

8.1 Копирование при записи

Для эффективного создания копии процесса ядро ОС могла бы просто создать копии таблиц страниц, указывающие на те же физические страницы, что и таблицы родителя. Однако этот метод не может работать со страницами данных «в лоб»: если все процессы (оригинал и все копии) только читают содержимое некоторой страницы, то достаточно иметь одну физическую страницу для всех процессов, но уже в момент записи в неё одним из процессов он должен работать со своей собственной страницей-копией.

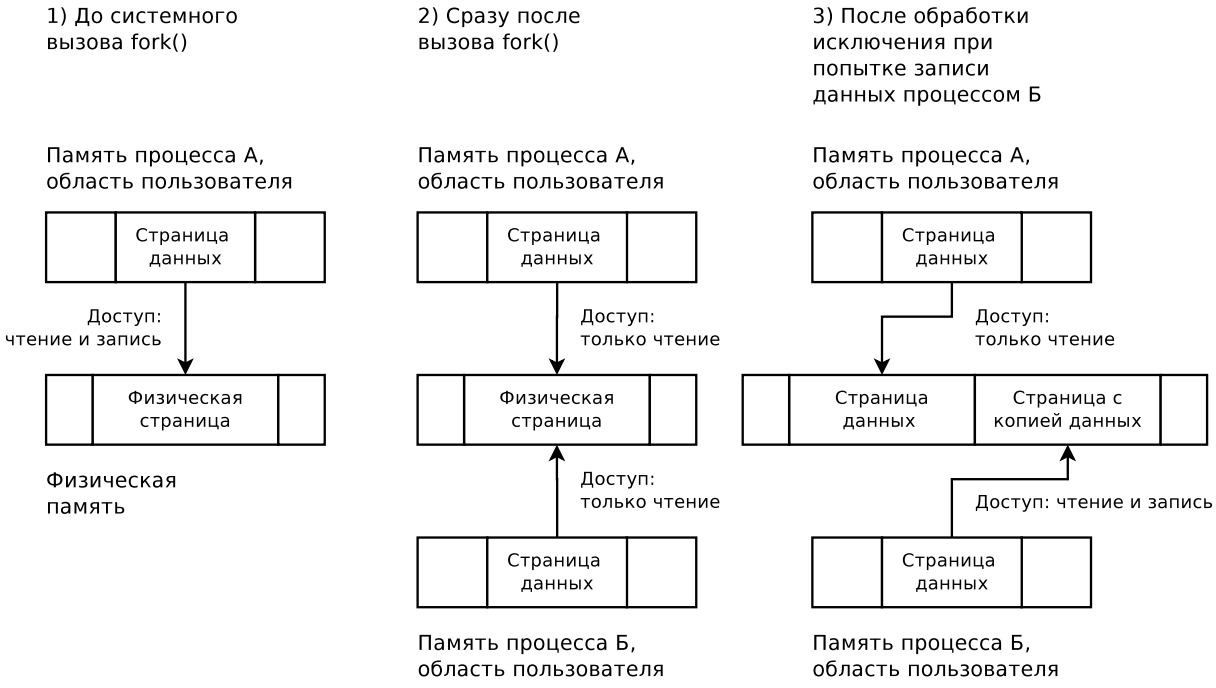

Рисунок 8.1 — Создание копии процесса и механизм копирования при записи

Для решения этой проблемы ядро может запретить запись во все страницы данных в пользовательской части как процесса родителя, так и процесса-потомка. При попытке записи в них произойдёт страничное исключение, при обработке которого ядро ОС должно создать страницу-копию

с разрешённой в неё записью и сделать соответствующие изменения в таблице страниц процесса, выполняющего запись, как показано на рисунке 8.1.

Идея о создании копии некоторого разделяемого ресурса (в нашем случае — физической страницы) при попытке его изменения называется *копированием при записи*. Мы используем именно этот принцип для реализации клонирования процесса. Для того, чтобы как-то выделить такие страницы, мы установим прямо в таблице страниц для них бит **PTE_COW**: в PTE имеется несколько бит, которые могут использоваться операционной системой на её усмотрение.

8.2 Схема клонирования процесса

Мы реализуем часть функционала известного системного вызова **fork** на уровне пользователя, используя уже имеющиеся и некоторые новые системные вызовы. Создаваемая реализация клонирования процессов должна включать следующие шаги.

1) Установка вторичного обработчика страничной ошибки, работающего в режиме пользователя и позволяющего отслеживать попытки записи в страницы, доступные только для чтения. Для установки такого обработчика мы добавим системный вызов **sys_env_set_pgfault_upcall**.

2) Создание пустого дочернего процесса вызовом **sys_exofork**.

3) Для всех доступных для записи (или клонированных методом копирования при записи) страниц родительского процесса с адресами ниже **UTOP** необходимо создать такие же страницы в пространстве потомка. Страницы потомка будут указывать на те же физические страницы, что и страницы родителя. После этого обе страницы, родительская и дочерняя, должны быть отмечены как доступные только для чтения, и у них должен быть установлен флаг **PTE_COW** в записи таблицы страниц.

4) Выделить дочернему процессу стек обработки прерываний. Для стека нужно выделить новую страницу физической памяти.

5) Установить вторичный обработчик страничной ошибки (страничного исключения) в дочернем процессе аналогично тому, как это сделано в родителе.

6) Пометить дочерний процесс как готовый к выполнению.

Обработка страничного прерывания теперь будет вестись следующим образом.

1) Ядро вызывает функцию **_pgfault_upcall**, которая вызывает установленный пользователем вторичный пользовательский обработчик (**pgfault**).

2) Пользовательский обработчик проверяет, была ли это ошибка записи (**FEC_WR**) и имела ли страница атрибут **PTE_COW**. Если хотя бы одно из условий не выполнено, то обработчик может аварийно завершить процесс.

3) В случае же успешной проверки обработчик выделяет новую страницу физической памяти, связывает её с некоторым виртуальным адресом и копирует туда содержимое страницы. Затем создаётся отображение страницы с «ошибочным» виртуальным адресом на эту созданную страницу с возможностью записи.

Как можно заметить, вся основная логика реализации **fork** выносится в часть пользователя, включая основную логику обработки страничного исключение. Это выглядит как весьма своеобразная идея авторов JOS, но это позволяет показать процесс асинхронной передачи управления из режима ядра в режим пользователя. Реальным примером использования такой передачи является, например, доставка сигналов в UNIX-системах, а в нашем случае её использование выглядит несколько натянутым.

8.3 Вызов пользовательского обработчика

Первое, что нам нужно сделать — добавить системный вызов для установки адреса пользовательского обработчика страничного исключения. В структуру **Env** для этого уже добавлено поле **env_pgfault_upcall**.

Задание 19. Реализуйте системный вызов **sys_env_set_pgfault_upcall**.

Процессу, использующему этот системный вызов, понадобятся два стека:

– обычный стек с вершиной по адресу **USTACKTOP**;
– стек с вершиной по адресу **UXSTACKTOP**, используемый в момент вызова пользовательского обработчика.

Процесс должен сам позаботиться о том, чтобы выделить одну страницу памяти под стек пользовательского обработчика, используя вызов **sys_page_alloc**, причём это нужно сделать и при создании дочернего процесса.

После того, как процесс выделил память под стек пользовательского обработчика страничного исключения и установил адрес этого обработчика, ядро должно вызвать его при обработке страничного исключения. Это является не вполне тривиальной задачей, поскольку в момент вызова управление передаётся от страничного обработчика исключения коду пользователя. Помните, что код пользователя должен по соображениям безопасности выполняться на уровне привилегий пользователя, а не ядра.

Сначала нужно изменить код обработчика страничного исключения в файле **kern/trap.c**. Если процесс, вызвавший исключение, не зарегистрировал обработчик исключения, то он должен быть уничтожен, как и происходило ранее. Если же процесс зарегистрировал такой обработчик, то нам необходимо передать ему управление из режима ядра.

Для передачи управления пользовательскому обработчику ядро должно создать на стеке обработки исключений (с вершиной в **UXSTACKTOP**) так называемый «кадр ловушки», который выглядит как структура **UTrapframe** из файла **inc/trap.h**. Состояние этого стека в момент передачи управления в режим пользователя показано на рисунке 8.2.

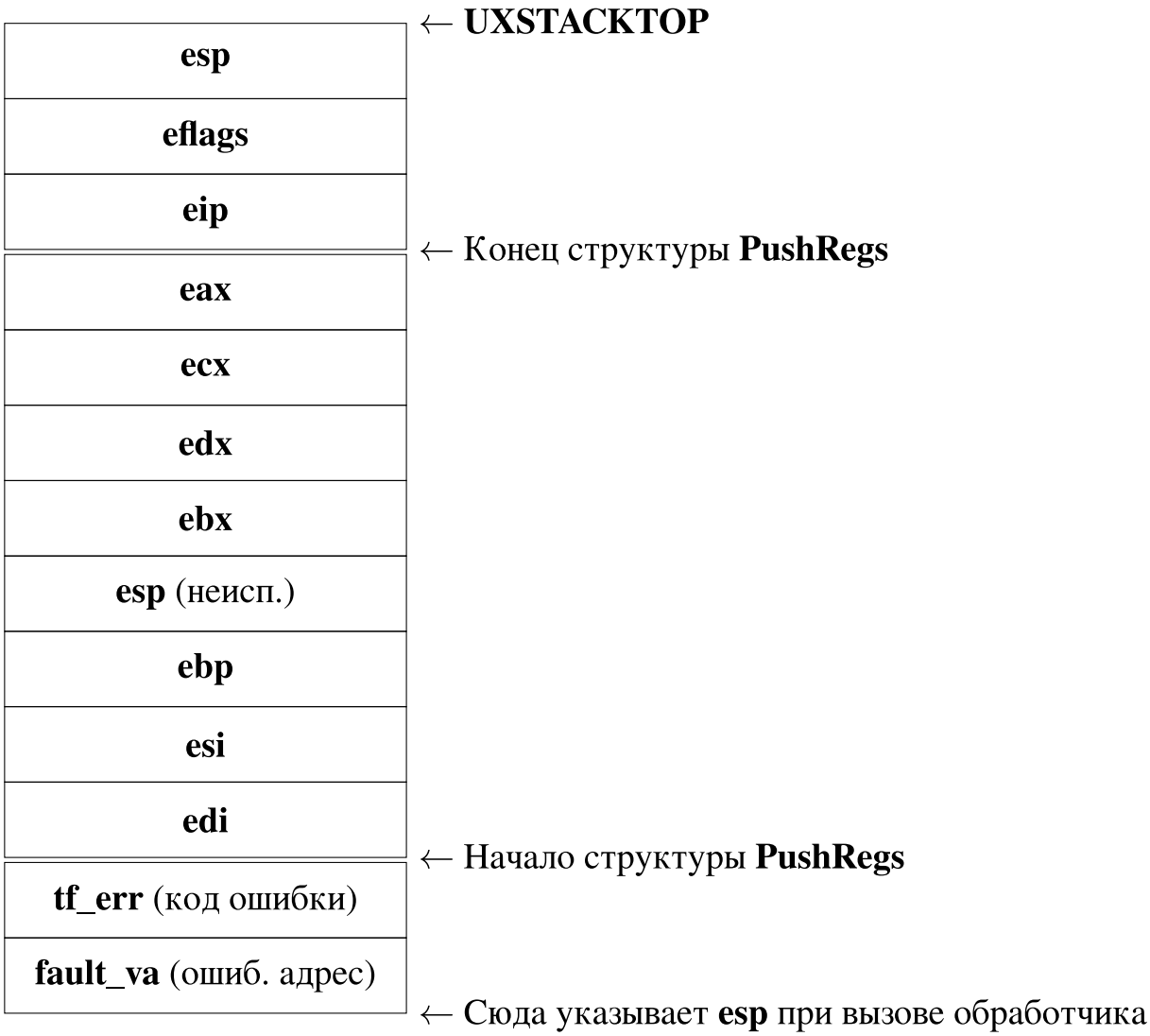

Рисунок 8.2 — Состояние стека обработки исключений перед вызовом вторичного обработчика страничного исключения

Отметим, что в стек здесь помещаются не текущие значения регистров, а их значения на момент возникновения страничного исключения.

Если процесс пользователя уже использовал этот стек в момент возникновения страничной ошибки, то это означает, что ошибка случилась в самом обработчике. Определить это можно, проверив, находится ли в значение поле **tf->tf_esp** на отрезке между **UXSTACKTOP - PGSIZE** и **UXSTACKTOP - 1**. В этом случае кадр стека нужно создавать не под адресом **UXSTACKTOP**, а ниже текущего значения **tf->tf_esp**. Положите в этот кадр стека обнулённое 32-битное слово — конкретное значение этого слова фактически не важно, важно лишь изменение указателя стека при этой операции. Затем в стек следует положить структуру **UTrapframe**.

Зарезервированное в случае вложенного исключения место в стеке позднее пригодится при манипуляциях со стеком перед возвратом управления из обработчика: мы сможем положить на его место адрес возврата для инструкции **ret** для возврата на место, где случилось «вложенное» страничное исключение.

В итоге на стеке обработки исключений создан кадр стека, который позволит вернуться в точку возникновения страничной ошибки по завершении работы пользовательского обработчика. Ядру осталось сделать так, чтобы процесс пользователя продолжил выполнение с начала обработчика исключения и с регистром **esp**, указывающим на самый низ созданного кадра. В поле **fault_va** должен находиться адрес, обращение к которому вызвало исключение. Решите сами, как именно следует передавать управление процессу пользователя с помощью инструкции **iret**.

Задание 20. Добавьте в функцию **page_fault_handler** (файл **kern/trap.c**) код, который передаст управление пользовательскому обработчику страничной ошибки.

Не забывайте, что под стек вторичной обработки исключения выделена только одна страница, и ядру нельзя выходить за её границы. Примите соответствующие меры, чтобы при исчерпании этого стека просто уничтожить проблемный процесс.

8.4 Пользовательский обработчик страничного исключения

Теперь у нас есть возможность вызова пользовательского обработчика страничной ошибки, правда, ещё не проверенная в работе. Как видно из предыдущего задания, этот обработчик вызывается не по правилам языка C. Поэтому нам придётся написать на ассемблере процедуру, которая будет вызывать часть обработчика, написанную на языке C, и возвращать управление в точку страничной ошибки. Именно адрес этого обработчика и следует передавать при вызове **sys_env_set_pgfault_upcall**.

Задание 21. Напишите обработчик **_pgfault_upcall** в файле **lib/pfentry.S**.

Сложность заключается в переключении на основной пользовательский стек и возврат в точку возникновения страничной ошибки после того, как пользовательский обработчик завершит работу. Отметим, что воспользоваться командой **iret** для переключения стека не удастся, поскольку пользовательский обработчик уже выполняется в режиме пользователя, и смены режимов, сопровождающейся переключением стеков, не произойдёт. Трудности возникают в силу необходимости восстановить сохраненные значения регистров, что означает, что свободно оперировать ими в коде **_pgfault_uplocal** не получится. Тщательно продумайте порядок восстановления регистров, который позволит добиться желаемого эффекта.

Вы можете использовать инструкцию **ret**, но перед её выполнением следует переключить стек на стек в момент страничного исключения: именно в этот стек и надо будет поместить адрес возврата! Вот почему в случае вложенного исключения мы пропустили слово в стеке обработчика страничного исключения, иначе адрес возврата затёр бы поле структуры **UTrapframe**, находящейся в этом же стеке. Таким образом, ядро «догадывается», как именно пользовательский обработчик вернётся в точку возникновения исключения, и чуть облегчает эту операцию.

Наконец, необходимо дописать связанный с обработчиком недостающий код в библиотеке пользовательских программ.

Задание 22. Закончите функцию **set_pgfault_handler** в файле **lib/pgfault.c**.

Для проверки текущего состояния работы нужно использовать программы **user/faultread**, **user/faultdie**, **user/faultalloc** и **user/faultallocbad**. Изучите их исходный код.

При запуске **user/faultread** вы должны получить следующее.

```
[00000000] new env 00001000
[00000000] new env 00001001
[00001001] user fault va 00000000 ip 0080003a
TRAP frame ...
[00001001] free env 00001001
```

При запуске **user/faultdie** вы должны получить следующее.

```
[00000000] new env 00001000
[00000000] new env 00001001
i faulted at va deadbeef, err 6
[00001001] exiting gracefully
[00001001] free env 00001001
```

При запуске **user/faultalloc** вы должны получить следующее. (Если вы видите только одну строку с текстом «this string», то, вероятно, вы неверно обрабатываете рекурсивные страничные ошибки.)

```
[00000000] new env 00001000
[00000000] new env 00001001
fault deadbeef
this string was faulted in at deadbeef
fault cafebffe
fault cafec000
this string was faulted in at cafebffe
[00001001] exiting gracefully
[00001001] free env 00001001
```

При запуске **user/faultallocbad** вы должны получить следующее.

```
[00000000] new env 00001000
[00000000] new env 00001001
[00001001] user_mem_check assertion failure for va deadbeef
[00001001] free env 00001001
```

8.5 Клонирование процесса копированием при записи

В файле **lib/fork.c** имеется заготовка функции **fork**. Подобно изученной ранее функции **dumbfork**, наш **fork** будет создавать новый пустой процесс и устанавливать отображение виртуального пространства нового процесса на те же страницы, что используются в дочернем процессе. Однако, в отличие от **dumbfork**, будут копироваться только адреса страниц, а не их содержимое.

Для начала нам нужно дописать функцию **duppage**, которая будет создавать в указанном процессе по указанному адресу страницу, указывающую на ту же физическую страницу, что и страница по этому адресу в текущем процессе. При этом она должна проверять бит записи в страницу (**PTE_W**) и бит **PTE_COW**: если хотя бы один из них установлен, то ей необходимо установить права на обе страницы в соответствии с рисунком 8.1.

Для своей работы функция **duppage** будет использовать системный вызов **sys_page_map** и массив **vpt**, который установлен на адрес **UVPT** в файле **entry.S**. В качестве отображения этого адреса, выровненного на 22 бита, по соответствующему ему элементу в текущую директорию страниц записана она же сама в качестве таблицы страниц, как показано ниже (файл **pmap.c**).

```
pgdir[PDX(UVPT)] = PADDR(pgdir) | PTE_U | PTE_P;
```

Всё это позволяет пользователю читать элементы таблиц страниц как элементы массива **vpt**.

Задание 23. Реализуйте функцию **duppage** в файле **lib/fork.c**.

Функция **fork** должна реализовывать, в отличие от функции **dumbfork**, ещё и корректный обход всего пользовательского адресного пространства. Для этого ей следует использовать уже упомянутый массив **vpt**. Однако, если некоторой таблицы страниц не существует (соответствующий элемент директории страниц равен нулю), то попытка чтения из попадающего на неё элемента массива **vpt** приведёт к страничному исключению. К счастью, благодаря тому, что директория страниц находится в себе самой как таблица страниц, рекурсивно можно дойти до неё и просто как до страницы! Массив **vpd** устанавливается в файле **entry.S** как раз на страничную директорию.

В итоге обход адресного пространства должен происходить в двойном цикле. Сначала происходит обход элементов массива **vpd** (соответствующего директории страниц), индекс **pdi** меняется в диапазоне от **VPD(0)** до **VPD(end)**. Если в элементе PDE установлен флаг **PTE_P**, то следует обойти и элементы массива **vpt** начиная с номера **pdi * NPDENTRIES** (число элементов — **NPTENTRIES**).

Отметим, что «раздвоение» не должно затрагивать страницы, доступные только для чтения. Чтобы в дальнейшем отличить такие страницы от страниц, клонированных методом копирования при записи, для всех подлежащих «раздвоению» страниц в соответствующие PTE добавляется флаг **PTE_COW**.

Поскольку адрес пользовательского обработчика страничной ошибки хранится в структуре ядра, то для дочернего процесса явным образом необходимо установить тот же обработчик страничной ошибки, что и в родительском процессе.

Задание 24. Реализуйте функцию **fork** в файле **lib/fork.c**.

Наконец, необходимо реализовать саму функцию пользовательского обработчика страничного исключения **pgfault**. Прежде всего она должна проверить, что имеет место быть ошибка при попытке записи в страницу — в поле **utf_err** должен быть установлен первый (если считать с нулевого) бит. Во-вторых, ей следует проверить в массиве **vpt**, стоит ли у страницы признак копирования при записи. В случае, если какое-либо из этих условий не выполняется, можно вызвать функцию **panic**.

В случае успеха указанных проверок следует выделить новую страницу, отобразив её на адрес **PFTEMP**. Затем следует скопировать в неё содержимое старой страницы и отобразить её по старому адресу с правами **PTE_W | PTE_U | PTE_P**.

Задание 25. Реализуйте функцию **pgfault** в файле **lib/fork.c**.

Если при проверке пользовательский обработчик страничного исключения вызывается в бесконечной рекурсии, то возможной причиной этого

является то, что скопированная страница имеет «старые» разрешения с битом **PTE_COW** вместо бита **PTE_W**.

Проверьте свой код программой **user/forktree**. Она должна вывести примерно следующие сообщения (перемешанные со строками «new env», «free env» и «exiting gracefully»). Порядок строк и идентификаторы процессов могут отличаться.

```
1001: I am ''
1802: I am '0'
2801: I am '00'
3802: I am '000'
2003: I am '1'
5001: I am '11'
4802: I am '10'
6801: I am '100'
5803: I am '110'
3004: I am '01'
8001: I am '011'
7803: I am '010'
4005: I am '001'
6006: I am '111'
7007: I am '101'
```

8.6 Контрольные вопросы

Подготовьте ответы на следующие вопросы.

1) Какие страницы не клонируются функцией **fork** в силу того, что они доступны только для чтения? Почему не клонируется стек вторичной обработки страничного исключения?

2) Насколько и по какой причине выровнен адрес **UVPT**?

3) Каким образом массив **vpd** отображён именно на элемент директории страниц?

4) Как функция **fork** обходит всё адресное пространство?

5) Является ли флаг **PTE_COW** стандартным с точки зрения архитектуры x86?

6) Если при клонировании по принципу копирования при записи каждый процесс создаёт свою копию защищённой от записи страницы, то как и кем в JOS освобождается исходная физическая страница?

7) Возвращается ли управление из пользовательского обработчика страничного исключения обратно в обработчик исключения в ядре?

8) Как передаётся управление из режима ядра пользовательскому обработчику?

9) Меняются ли при возврате из пользовательского обработчика уровни привилегий?

10) Почему для вторичной обработки исключения используется специальный стек? Как он будет использоваться при вложенном страничном исключении?

11) Почему в стек обработки исключений помещаются не значения регистров в момент перехода, а их значения в момент страничного исключения?

12) В чём отличие двух структур кадров ловушек (**Trapframe** и **UTrapframe**)? Как это связано с их использованием?

13) В какой именно момент в реализованном пользовательском обработчике страничного исключения может произойти устранимое страничное исключение? (Отметим, что это может произойти только в момент изменения памяти, попадающей под копирование при записи.)

14) В нашей системе нет системных вызовов, которые бы заполняли буфер в области пользователя, таких как функция **read** [5]. Будет ли такой системный вызов совместим с реализованным копированием при записи? Что следовало бы изменить в системе для реализации такого вызова?

15) Можно ли при выходе из пользовательского обработчика использовать команду **iret** для смены текущего стека?

16) Что случится, если ядро «проглядит» момент переполнения пользовательского стека исключения?

17) В чём разница обработки страничной ошибки для программ **user/faultalloc** и **user/faultallocbad**? Почему программе **user/faultallocbad** не удается вывести на экран строку, которую обработчик страничного исключения должен разместить по адресу 0xDEADBEEF?

18) Как именно программа **user/faultalloc** добивается второго страничного исключения?

9 Вытесняющая многозадачность

В этой главе в JOS будет добавлена вытесняющая многозадачность, которая позволяет ядру при помощи обработки прерывания таймера получать контроль над процессором после истечения некоторого интервала времени, даже если выполняемый процесс не желает этого. В результате обработки прерывания таймера процессор может быть отдан в распоряжение другого готового к выполнению процесса, то есть ближайший возврат из обработчика прерывания происходит, вообще говоря, не в том же самом процессе, в котором произошло прерывание.

Сначала убедимся, что мы корректно реализовали создание дочернего процесса клонированием, а вечный цикл в любом из процессов пока что лишает остальные процессы доступа к ЦП (наша система, напомним, является однопроцессорной).

Изменим файл **kern/init.c** так, чтобы он запускал программу **user/spin**. Данная программа создает потомка, который крутится в бесконечном цикле. Если мы всё реализовали правильно, то мы увидим, что управление не будет возвращено родительскому процессу, после того как дочерний процесс получит ЦП в своё распоряжение.

Для исправления этого недостатка системы нам необходимо добавить в ядро обработку прерывания, вызванного таймером, и возможность «отбирать» машинное время у исполняемого процесса. Таким образом, в этой главе мы перейдём от кооперативной многозадачности к *вытесняющей многозадачности* [1].

9.1 Аппаратные прерывания и их связь с IDT

Прерывание, исходящие от внешних по отношению к ЦП устройств, на платформе x86 принято называть IRQ. При возникновении прерывания устройство посылает сигнал на *контроллер прерываний*, а уже последний уведомляет о прерывании процессор. Таким образом, контроллер прерываний с точки зрения ЦП является единственным источником прерываний. Процессор получает от контроллера прерываний номер прерывания, с которым он и обращается к IDT для получения информации об обработчике прерывания. Контроллер прерываний поддерживает приоритеты прерываний, а также может не присылать сигнал об очередном прерывании с некоторым номером до получения сигнала завершения обработки прерывания (EOI) от процессора.

Архитектура x86 предполагает два идентичных восьмиканальных контроллера прерываний, причем второй подключён к первому каскадом. Таким образом, в x86 есть шестнадцать возможных IRQ с номерами от 0

до 15, причём IRQ2 всегда используется для каскадирования второго контроллера прерываний и не может быть использовано устройствами.

В нашей системе номера прерывания устройств связаны с элементами IDT с номерами от **IRQ_OFFSET** до **IRQ_OFFSET+15**. Значение **IRQ_OFFSET** выбрано так, чтобы аппаратные прерывания не пересекались с исключениями процессора (напомним, что их номера являются фиксированными) и прерыванием системного вызова. Поскольку с таймером в i386 связан нулевой номер IRQ, то в JOS адрес обработчика его прерываний находится в элементе **IDT[IRQ_OFFSET]**.

Связь между IRQ и IDT на уровне контроллера прерываний устанавливается функцией **pic_init** в файле **kern/picirq.c**. В этом же файле программируется контроллер прерываний: в частности, выбирается режим автоматического завершения прерывания (Auto-EOI). В этом режиме прерывание с некоторым номером может сразу возникать повторно, как только процессор подтвердил получение сигнала прерывания от контроллера прерывания [8].

Функция **pic_init** только сообщает контроллеру прерываний связь IDT и IRQ, а заполнить саму таблицу IDT необходимо нам. Отметим, что при вызове обработчика аппаратного прерывания процессор не проверяет DPL и никогда не помещает в стек код ошибки.

Задание 26. Измените файлы **kern/trapentry.S** и **kern/trap.c** так, чтобы они корректно инициализировали элементы IDT, связанные с IRQ.

Следует отметить, что прерывание таймера работает таким образом только на однопроцессорных системах архитектуры x86 (включая многопроцессорные системы, где ОС использует только один процессор из имеющихся), поскольку в данном случае у нас имеется всего один таймер. По мере развития архитектуры x86 появились и полноценные многопроцессорные и многоядерные вычислительные системы на её основе (SMP, [1]). В таких системах используется иной контроллер прерываний (APIC), что позволяет, в частности, каждому логическому процессору иметь свой собственный таймер. Наша упрощённая ОС в принципе не может поддерживать такую архитектуру: она не имеет синхронизации доступа к глобальным данным ядра и исходит из наличия единственного ЦП при планировании выполнения процессов.

9.2 Разрешение аппаратных прерываний

После включения системы прерывания запрещены, пока ОС явно не разрешит их. Разработчиками JOS было принято чрезвычайно упрощённое и неэффективное, с точки зрения производительности, решение о полном запрещении аппаратных прерываний в режиме ядра. Таким образом, пре-

рывания в JOS возможны только в режиме пользователя. Это существенно упрощает организацию системы, поскольку невозможен повторный вход в ядро, если управление уже передано ему.

Отметим, что для разрешения прерываний в режиме пользователя мы не можем просто использовать инструкцию разрешения прерываний **sti** перед **iret**. Во-первых, это разрешит прерывания в ядре, во-вторых, это приведёт к гонкам: прерывание может возникнуть сразу после **sti**, но до **iret**. В-третьих, в момент выполнения инструкции **iret** в регистр флагов загрузится новое его значение, которое и будет определять статус разрешения прерываний в режиме пользователя.

Поскольку в x86 прерывания разрешаются установкой флага **FL_IF** в регистре **eflags** (см. **inc/mmu.h**), то JOS достаточно сохранять и восстанавливать этот регистр при переключении режимов, а также установить его при запуске первого процесса в пользовательском режиме. За автоматический сброс флага **FL_IF** при входе в режим ядра отвечает тип дескриптора в IDT (раздел 1.9).

Задание 27. Измените функцию **env_alloc** в файле **kern/env.c** таким образом, чтобы процесс начинал работать в режиме пользователя с разрешёнными прерываниями. Нужный нам регистр будет взят при переходе в режим пользователя из поля **tf_eflags** кадра ловушки.

Для выполнения этого задания рекомендуется перечитать раздел 9.2 из [6] или раздел 6.8 из [7].

Теперь при запуске программы **user/spin** вы должны увидеть на экране кадры ловушек для аппаратных прерываний. Пока что при обработке прерывания JOS всегда просто уничтожает процесс. Убедитесь, что прерывания возникают только в режиме пользователя — непрерывный поток кадров ловушек на экране свидетельствует об обратном.

9.3 Генерация и обработка прерываний от таймера

Очевидно, что для реализации вытесняющей многозадачности прерывания от таймера должны возникать периодически. Функция **kclock_init** в файле **kclock.c** настраивает аппаратный таймер соответствующим образом.

Поскольку работа по настройке таймера и контроллера прерываний в JOS уже проделана, то осталось выполнить планирование процессов. Наш планировщик задач будет работать во время обработки прерывания от таймера. В ядре уже есть функция **sched_yield**, передающая управление следующему готовому к выполнению процессу, поэтому обработка прерывания таймера будет сводиться к её вызову.

Задание 28. Измените функцию **trap_dispatch** так, чтобы она вызывала функцию **sched_yield** при обработке прерывания таймера.

Теперь программа **user/spin** должна работать так, как это и задумывалось — дочерний процесс будет уничтожен родительским спустя некоторое время. Убедитесь в этом.

После выполнения заданий крайне желательно убедиться, что предыдущие тесты (например, **user/forktree**) всё ещё корректно работают и в ядре ничего не было «сломано» при добавлении вытесняющей многозадачности. В худшем случае может иметь смысл откатиться к предыдущим зафиксированным изменениям (используя команду **git reset**) и выполнить последние задания ещё раз.

9.4 Контрольные вопросы

Подготовьте ответы на следующие вопросы.

1) Чему равно значение **IRQ_OFFSET**?

2) Как ядро разрешает прерывания в режиме пользователя? Как они запрещаются в режиме ядра?

3) Какие функции выполняет контроллер прерываний?

4) Какой номер аппаратного прерывания не может быть использован устройством?

5) Как номера IRQ связываются с записями в IDT? Кем реализуется эта связь — процессором или контроллером прерываний?

6) Для чего служит сигнал EOI? Есть ли связь между ним и командой **iret**?

7) Когда в нашем случае контроллер прерываний считает прерывание обработанным?

8) Может ли прерывания таймера в нашей системе прервать выполнение системного вызова?

9) Могут ли пользовательские программы в нашей системе использовать регистры процессора с плавающей точкой после добавления вытесняющей многозадачности?

10) Можно ли утверждать, что JOS реализует дисциплину *Round-robin* [1] при распределении процессорного времени (исключая бездействующий процесс)? Какие отличия можно отметить?

11) Сколько уровней приоритета задач существует в JOS?

12) Где и как сохраняются значения регистров процесса, у которого «отобрали» ЦП? Как они восстанавливаются?

13) Выделяется ли в JOS процессу именно квант времени — иными словами, сбрасывается ли таймер при получении процессом управления?

10 Межпроцессное взаимодействие

К настоящему моменту мы достаточно много времени уделили изоляции памяти процессов друг от друга, но пока что ничего не сделали для обеспечения какого-либо взаимодействия процессов. В ходе данного занятия мы исправим сложившуюся ситуацию: один процесс сможет послать другому либо целое число, либо предоставить ему доступ к своей странице с данными.

10.1 Обмен сообщениями между процессами

Мы реализуем в нашей системе простейший обмен сообщениями: один процесс может отправить другому процессу целое число и виртуальный адрес некоторой своей страницы. При посылке виртуального адреса процесс-получатель получит доступ к той же физической странице, что и процесс-отравитель. Посланный адрес для этого должен быть, конечно же, ненулевым и указывающим на существующую страницу в пользовательской части процесса-отправителя.

Возможность «посылки» адреса позволит процессам обмениваться данными довольно большого объёма без дополнительных усилий со стороны ядра ОС: по сути мы реализуем таким образом механизм разделяемой между процессами физической памяти [5]. К сожалению, использование этого механизма в нашей системе существенно ограниченно тем, что у нас не будет предусмотрено механизмов синхронизации доступа к этой памяти, таких как, например, семафоры [1]. Единственным средством разделения доступа к странице в JOS является возможность установить для неё ограниченные разрешения для процесса-получателя: его доступ может быть ограничен только чтением.

Для реализации IPC в дескриптор процесса были добавлены поля, показанные ниже.

```
struct Env {
    bool env_ipc_recving;
    void *env_ipc_dstva;
    uint32_t env_ipc_value;
    envid_t env_ipc_from;
    int env_ipc_perm;
};
```

Флаг **env_ipc_recving** ненулевой, если процесс ожидает приёма сообщений. Поле **env_ipc_dstva** определяет адрес в процессе-получателе, который будет отображён в ту же физическую страницу, адрес которой (в своём пространстве) отправил процесс-отправитель. Поле **env_ipc_value** хранит

отправленное по IPC простое целое значение. Идентификатор процесса-отправителя сохраняется в поле **env_ipc_from**. Наконец, поле **env_ipc_perm** имеет или нулевое значение, если отправитель не послал адрес страницы, или хранит права доступа к ней, в ином случае.

10.2 Системные вызовы и библиотечные функции для обмена сообщениями

Для поддержки создаваемого механизма IPC нам понадобятся два новых системных вызова: **sys_ipc_recv** и **sys_ipc_try_send**, которые возвращают ноль в случае успеха и коды ошибок в ином случае.

```
int sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm);
int sys_ipc_recv(void *dstva);
```

Видно, что системный вызов получения не возвращает информацию о полученном числе — в этом, формально, нет необходимости, поскольку процесс имеет полный доступ на чтение к своему дескриптору. Благодаря аргументу **perm** процесс-отправитель может ограничить доступ получателя к странице только возможностью чтения данных. Кроме того, нужно добавить две вызывающие их функции (**ipc_recv** и **ipc_send**) в библиотеку пользователя.

Как можно увидеть, в поле дескриптора есть место для хранения только одного посланного сообщения. Поэтому после получения сообщения ядро должно сбросить в дескрипторе процесса-получателя флаг **env_ipc_recving**, после чего процессу нельзя будет послать сообщения, пока он не обработает его в режиме пользователя и затем опять не вызовёт **sys_ipc_recv**. Попытки послать сообщение процессу, не ожидающему их, в нашей системе будут заканчиваться ошибкой — в ядре просто нет места для их хранения. Чтобы подчеркнуть этот момент, системный вызов отправки был назван **sys_ipc_try_send**.

В межпроцессном взаимодействии в нашей системе должно участвовать два разных процесса. Из них один должен сделать системный вызов **sys_ipc_recv** для получения сообщения. В ходе этого системного вызова ядро должно выполнить следующие шаги.

1) Сделать пометку в дескрипторе этого процесса, что он ожидает сообщений (поле **env_ipc_recving**).

2) Проверить значение **dstva** на выравнивание по адресу страницы и по принадлежности области пользователя, и записать его в поле дескриптора **env_ipc_dstva** — сюда надо будет отобразить «посланную» страницу процесса-отправителя.

3) Заблокировать процесс, установив его состояние в значение **ENV_NOT_RUNNABLE**.

4) Сохранить результат системного вызова (ноль) в сохранённом контексте текущего процесса в поле **tf_regs.reg_eax**.

5) Активировать планировщик задач (вызвать функцию **sched_yield**) для начала выполнения другого процесса.

Ожидающий сообщений процесс будет готов к выполнению и сменит состояние на **ENV_RUNNABLE**, когда какой-либо другой процесс пошлёт ему сообщение.

Второй процесс-участник IPC должен выполнять системный вызов **sys_ipc_try_send**, указав в его параметрах идентификатор процесса-получателя, передаваемое значение и адрес, а также реквизиты доступа к общей странице памяти. В этом системном вызове проверяется, действительно ли указанный процесс существует и ждёт в данный момент сообщений. Если это так, то системный вызов **sys_ipc_try_send** должен сделать следующее.

1) Записать переданное целое значение и идентификатор процесса-отправителя в поля **env_ipc_value** и **env_ipc_from** дескриптора процесса-получателя.

2) Если метод получил ненулевой адрес, то отобразить «передаваемую» физическую страницу процесса-отправителя по адресу в поле **env_ipc_dstva** дескриптора процесса-получателя.

3) Если метод получил нулевой адрес, то записать ноль в **env_ipc_dstva**.

4) Отметить процесс-получатель как готовый к выполнению.

5) Завершиться и возвратить ноль.

Если указанный процесс не существует или не ожидает сообщений, то системный вызов возвращает значение **-E_IPC_NOT_RECV**. Как можно заметить, системный вызов **sys_ipc_try_send** не блокирует выполнение вызывающего процесса.

Теперь процесс-получатель получит управление и вернётся из системного вызова **sys_ipc_recv**.

Библиотечная функция **ipc_recv** просто вызывает **sys_ipc_recv**, а затем извлекает полученные значения **env_ipc_value** и **env_ipc_dstva** из дескриптора текущего процесса: как мы помним, он доступен для чтения из режима пользователя.

Библиотечная функция **ipc_send** повторяет вызовы **sys_ipc_try_send**, пока не добьётся успеха. Если системный вызов вернёт ошибку, то функция

ipc_send отдаёт управление другому процессу, а затем, когда планировщик задач опять предоставит ей ЦП, повторяет попытку вызова **sys_ipc_try_send**.

10.3 Передача адресов страниц

Передачу одного числа явно нельзя отнести к производительным и удобным способам обмена данным, и поэтому в реализуемом IPC есть возможность отправлять адрес страницы процесса. Для доступа получателя к этой странице мы должны отобразить соответствующую физическую страницу в его адресное пространство.

Если при вызове **sys_ipc_recv** параметр **dstva** не равен нулю, то это значит, что процесс готов принять адрес страницы от процесса-отправителя и эту страницу надо связать с виртуальным адресом **dstva**.

Если с этим адресом уже была связана физическая страница, то эта связь уничтожается. В параметре **perm** отправитель указывает разрешения на доступ к странице для получателя. Таким образом, если получатель принимает, а отправитель посылает адрес страницы (ненулевой аргумент **srcva**), то она должна быть отображена на адрес **dstva** в процессе-получателе, и адреса **dstva** в одном процессе и **srcva** в другом будут связаны с одной и той же физической страницей.

Таким образом, будет реализована память, разделяемая между двумя процессами. Заметьте, что ядро должно проверять передаваемый адрес на валидность, включая принадлежность пользовательской части пространства.

10.4 Реализация механизма обмена сообщениями и его проверка

Теперь необходимо реализовать два описанных системных вызова и две новые библиотечные функции, необходимые для реализации IPC.

Задание 29. Реализуйте системные вызовы **sys_ipc_recv** и **sys_ipc_try_send** в файле **kern/syscall.c**.

При использовании в этих вызовах функции **envid2env** её надо вызывать с нулевым параметром **checkperm**: реализованный IPC выглядит достаточно безопасным и каждый процесс может послать сообщение другому процессу, если он его ожидает.

Задание 30. Реализуйте функции библиотеки пользователя **ipc_recv** и **ipc_send** в файле **lib/ipc.c**.

Проверьте созданный IPC с помощью программ **user/pingpong** и **user/primes**. Первая из них создаст два процесса, посылающих друг друг

сообщения, а вторая реализует параллельный (точнее, конкурентный) вариант решета Эратосфена для расчёта простых чисел, число которых ограничено числом свободных дескрипторов процессов.

10.5 Состояния процесса в JOS

До этой главы мы использовали состояние процесса **ENV_NOT_-RUNNABLE** для обозначения того факта, что процесс ещё не до конца создан своим родителем (глава 7). Теперь это состояние используется и для обозначения факта блокировки выполнения процесса, который ожидает прихода сообщения от другого процесса по созданному механизму межпроцессного взаимодействия.

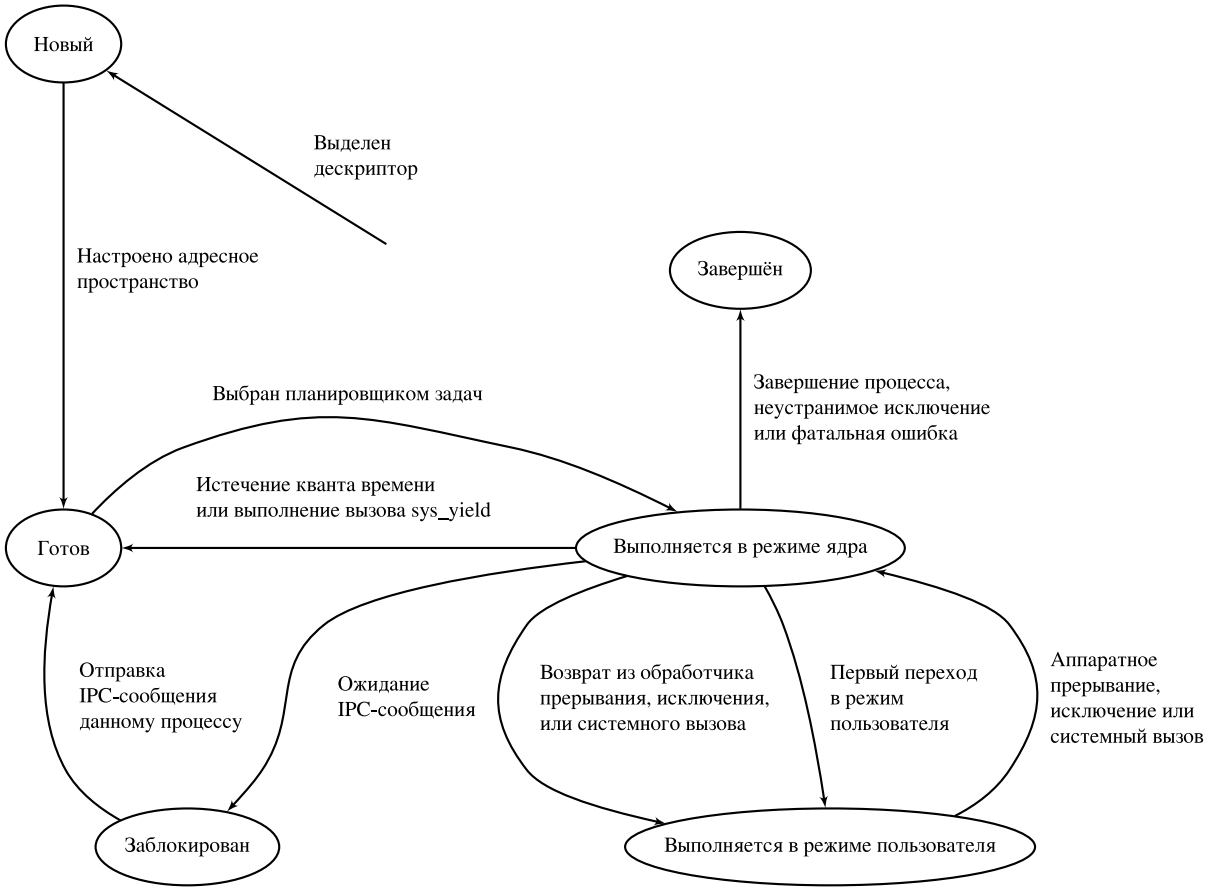

Рисунок 10.1 — Состояния процесса в JOS

Таким образом, у нас наконец-то есть примеры на все состояния процесса, описанные в [1, 5]. На рисунке 10.1 показаны все состояния процесса в нашей системе и переходы между ними. Режим выполнения процесса (пользователь/ядро) был учтён при выделении состояний, поскольку процесс может сменить своё состояние на «готов» или «заблокирован» только после перехода в режим ядра. На рисунке переход в режим пользователя в случае

возврата из исключения отделён от первого перехода в режим пользователя: хотя они оба выполняются функцией **env_run**, но с логической точки зрения они отличаются.

Приводящие к изменению состояния процесса события могут как происходить в нём самом, так и в других процессах. В первом случае имеет место переход из состояния выполнения в режиме ядра или в режиме пользователя в некоторое другое, во втором — переход из всех прочих состояний. Отметим, что истечение выделенного кванта процессорного времени или выполнение системного вызова **sys_yield** приводят к смене состояния процесса только в том случае, когда планировщик задач в ядре ОС выбрал некоторый другой процесс для исполнения — при выборе этого же процесса он продолжает своё выполнение в режиме ядра.

Отметим, что поле **env_status** в дескрипторе процесса не показывает всей полноты его состояний и само по себе не позволяет отличить, например, заблокированный в ожидании сообщения процесс от не полностью созданного, или исполняемый процесс от готового к выполнению (для этого надо сравнить адрес дескриптора процесса и значение переменной **curenv**). Причиной такого решения является минимизация кода, изменяющего и проверяющего это поле, а также стремление «не множить сущее без необходимости».

10.6 Контрольные вопросы

Подготовьте ответы на следующие вопросы.

1) Почему в нашей системе можно послать сообщение только процессу, заблокированному в ходе системного вызова **sys_ipc_recv**?

2) Получает ли системный вызов **sys_ipc_recv** управление назад после того, как вызовет планировщик задач, или процесс-получатель сразу получает управление в режиме пользователя?

3) Системный вызов **sys_ipc_recv** в случае успеха должен вернуть ноль, передаваемый по соглашениям через регистр **eax**. Как именно он возвращает это нулевое значение?

4) Является ли вызов **sys_ipc_recv** блокирующим? Вызов **sys_ipc_try_send**?

5) Приводит ли функция **ipc_send** к блокировке процесса, с точки зрения ядра ОС? С точки зрения прикладного программиста?

6) Можно ли организовать двунаправленный обмен данными между двумя процессами с помощью созданного IPC, если его использовать только для отправки чисел?

7) Возможно ли в JOS получить тупик (англ. *deadlock*, [1]) с помощью созданного IPC?

8) Возможно ли сейчас в JOS синхронизировать доступ процессов к общей странице?

9) Для чего служит аргумент разрешений доступа к странице?

10) Мы реализовали синхронный или асинхронный обмен сообщениями?

11) С какой скоростью, относительно работы планировщика процессов, идёт процесс обмена сообщениями?

12) Может ли реализованный механизм IPC быть использован процессом для связи с самим собою?

13) Если два процесса обмениваются числами, то может ли в нашей системе третий процесс увидеть, что они передают друг другу?

14) Почему программа **user/primes** находит только 1022 простых числа?

15) Перечислите все события, приводящие в нашей системе к смене состояния процесса.

16) Находится ли в нашей системе отправляющий процесс в заблокированном состоянии, если указанный процесс-получатель ещё не ожидает прихода сообщения?

17) Мог бы работать предложенный механизм IPC, если бы JOS поддерживала несколько процессоров или прерывания в режиме ядра? Какие изменения следовало бы внести?

18) Организует ли ядро JOS очередь передаваемых процессу сообщений?

19) Допустим, что два процесса постоянно посылают третьему сообщения. Пусть последний так же в вечном цикле вызывает функцию получения сообщений. Удастся ли втором процессу послать сообщение? При каком стечении обстоятельств это возможно?

Заключение

Мы познакомились с архитектурой x86 и её возможным использованием операционной системой. В результате работы по созданию ядра ОС были реализованы его основные функция как средства разделения аппаратных ресурсов: физической памяти, процессора, устройства вывода. При выполнении практических заданий был создан недостающий код в подсистемах управления памятью, планировщика задач, создания и клонирования процессов, межпроцессного взаимодействия. В итоге была создана многозадачная однопользовательская однопроцессорная система с вытесняющей многозадачностью, по классификации из [1].

Недостатками полученной ОС являются отсутствие ввода-вывода, файловой системы и поддержки многопроцессорной архитектуры. Тем не менее, созданная система реализует полноценную изоляцию процессов, виртуальную память на основе страниц, вытесняющую многозадачность и эффективное клонирование процессов на основе копирования памяти при её изменении.

Автор надеется, что основы разработки операционных систем были успешно изучены на практике теми читателями пособия, кто самостоятельно выполнил предложенные практические задания.

Список использованных источников

1. Э. Таненбаум. Современные операционные системы. — СПб: Питер, 2010. — 1120 с.

2. Митницкий В. Я. Архитектура IBM PC и язык Ассемблера. — М: Издательство МФТИ, 2000. — 148 с.

3. Керниган Б., Ритчи Д. Язык программирования С, 2-ое изд. — М: Вильямс, 2007. — 304 с.

4. ISO/IEC 9899:TC3 (стандарт C99). **http://open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf**

5. Робачевский А., Немнюгин С., Стесик О. Операционная система UNIX. — СПб: БХВ-Петербург, 2010 — 656 с.

6. Intel 80386 Programmer's Reference Manual. **http://css.csail.mit.edu/6.858/2011/readings/i386.pdf**

7. Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3A: System Programming Guide, Part 1. **http://download.intel.com/design/processor/manuals/253668.pdf**

8. Intel 8259A Programmable Interrupt Controller. **http://pdos.csail.mit.edu/6.828/2010/readings/hardware/8259A.pdf**