Белоус В.С., Переверзев В.А.

ОСНОВЫ РЕАЛИЗАЦИИ ОПЕРАЦИОННЫХ СИСТЕМ

Учебное пособие

Версия от 24 сентября 2020 г.

На примере учебной операционной системы рассмотрены основные вопросы создания ядра операционной системы, начиная от его загрузки и заканчивая реализацией вытесняющей многозадачности. В ходе практических работ 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».

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ

В настоящем отчете о НИР применяют следующие термины с соответствующими определениями.

16-битный режим — унаследованный режим или режим совместимости, в котором размер адреса по умолчанию составляет 16 бит.

32-битный режим — унаследованный режим или режим совместимости, в котором размер адреса по умолчанию составляет 32 бита.

64-битный режим — подрежим длинного режима. В данном режиме размер адреса по умолчанию составляет 64 бита и доступны новые возможности, такие как расширенный набор регистров.

Режим совместимости — подрежим длинного режима. В данном режиме размер адреса по умолчанию составляет 32 бита, он позволяет запускать существующее 16-битное и 32-битное прикладное ПО без перекомпиляции.

Унаследованный режим — режим работы процессора в котором существующее 16-битное и 32-битное ПО можно запустить без модификаций. Включает 3 подрежима: реальный режим, защищенный режим, режим виртуального 8086.

Длинный режим — режим работы процессора уникальный для архитектуры AMD64. Имеет 2 подрежима: 64-битный режим и режим совместимости.

Системное ПО — привилегированное ПО, которое управляет аппаратными ресурсами системы и контролирует доступ к этим ресурсам.

Логический адрес — адрес до сегментного преобразования, включающий в себя сегментный регистр (селектор) и смещение.

Эффективный адрес — смещение в сегменте.

Линейный (виртуальный) адрес — адрес полученный в результате сегментного преобразования.

Физический адрес — адрес в физическом адресном пространстве.

Каноническая форма адреса — форма адреса, в которой все биты, начиная с наиболее значимого (в данной реализации) и до 63-го совпадают.

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

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

ОС — операционная система.

ЦП — центральный процессор.

ПО — программное обеспечение.

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.

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

PML4 — таблица страниц верхнего уровня в длинном режиме (англ. Page Map Level 4).

PML4E — элемент таблицы страниц 4го уровня (англ. Page Map Level 4 Entry).

PDP — таблица указателей на директории страниц (англ. Page Directory Pointer).

PDPE — элемент таблицы указателей на директории страниц (англ. Page Directory Pointer Entry).

PDE — элемент директории страниц (англ. Page Directory Entry).

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

CPL — текущий уровень привилегий процессора (англ. Current Privilege Level).

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

RPL — уровень привилегий процесса, создавшего селектор (англ. Requestor Privilege Level).

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

LDT — локальная таблица дескрипторов (англ. Local Descriptor Table).

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

IST — таблица указателей стека в длинном режиме (англ. Interrupt Stack Table).

TSS — сегмент состояния задачи (англ. Task State Segment).

PAE — механизм расширения физических адресов, позволяет использовать физические адреса длинной до 52 бит (англ. Physical Address Extension).

EFER — моделезависимый регистр включения расширенных возможностей (англ. Extended Feature Enable Register).

FLAGS — 16-битный регистр флагов.

EFLAGS — 32-битный регистр флагов.

RFLAGS — 64-битный регистр флагов.

IP — 16-битный счетчик команд.

EIP — 32-битный счетчик команд.

RIP — 64-битный счетчик команд.

CS — регистр сегмента кода.

CR0, CR2 – CR4, CR8 — управляющие регистры процессора.

MSR — моделезависимый регистр (англ. Model Specific Register).

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

PIC — программируемый контроллер прерываний (англ. Programmable Interrupt Controller).

APIC — улучшенный программируемый контроллер прерываний (англ. Advanced Programmable Interrupt Controller).

IOAPIC — контроллер, расположенный на системной плате, используемый для управления внешними прерываниями (англ. I/O Advanced Programmable Interrupt Controller).

ВВЕДЕНИЕ

Для изучения основ функционирования операционных систем недостаточно изучения только теоретического материала. Для понимания работы ядра ОС необходимо изучать и модифицировать его исходный код.

В настоящее время существует множество операционных систем с открытым исходным кодом: GNU Linux, FreeBSD, ReactOS и др. Однако, ядра этих операционных систем плохо подходят для учебного процесса, т.к. они имеют большой объем исходного кода и обладают высокой сложностью.

По этой причине были разработаны несколько учебных операционных систем: JOS, xv6 и PhantomEx. Ядра этих ОС имеют небольшой объем исходного кода и небольшую сложность, по сравнению с эксплуатируемыми ОС, что делает их пригодными для обучения.

Однако, данные ОС имеют один существенный недостаток: они разработаны под устаревшую архитектуру x86.

Поэтому было принято решение разработать учебную ОС под более современную архитектуру x86_64, автор решил назвать ее **Ann**.

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 семейства x86_64, организации в них виртуальной памяти и обработки прерываний. Вторая глава посвящена обзору исходных текстов Ann и используемых диалектов языков программирования. Остальные главы содержат описание практических занятий по разработке операционной системы на базе имеющихся исходных кодов Ann.

В конце глав приводятся контрольные вопросы для проверки усвоенных студентом знаний. Некоторые главы содержат задания по написаню исходного кода, которого не хватает для полноценной работы разрабатываемой операционной системы.

Для выполнения практических заданий пособия читателю потребуется подключенный к интернету компьютер с POSIX-системой, имеющей следующие программы (указаны версии, которые были у автора, более новые тоже должны подойти): gcc-4.9.3, binutils-2.25.1, automake-1.15, autoconf-2.69, libtool-2.4.6, perl-5.10, make-4.1, gdb-7.10.1, git-2.10.2, qemu-2.7.0, произвольный текстовый редактор для работы с исходными текстами Ann (если не знаете что выбрать - попробуйте vim).

Авторы хотели бы поблагодарить за помощь Келарева Ивана Андреевича, Горина Сергея Викторовича, Арышеву Анну Григорьевну, Оленева Антона Александровича и всех причастных.

1 Основы архитектуры x86_64

1.1 История процессоров x86

Первым процессором из линейки x86 был процессор Intel 8086 – 16-разрядный процессор с адресным пространством равным 1 Мб. За ним последовали 80186 и 80286 (i286), в последнем появился защищенный режим, в котором можно было адресовать до 1 Гб памяти. Следующим был процессор 80386 (i386), который позволял адресовать 4 Гб физической памяти и имел страничное преобразование (для реализации виртуального адресного пространства). Последующие поколения добавляли новые возможности: встроенный сопроцессор, кэш-память, PAE, MMX, SSE, SSE2 и т.д.

В начале 2000-х годов стало очевидно, что 32-битного адресного пространства недостаточно для приложений, работающих с большими объемами данных (например с видео и базами данных), т.к. в этом случае процессор может использовать только 4 гигабайта виртуального адресного пространства.

Для решения этой проблемы компания Intel разработала спецификацию IA-64 (используется в процессорах семейства Itanium). Для сохранения обратной совместимости с унаследованным 32-битным ПО, в данной архитектуре использовался режим эмуляции, который уступал по производительности оригинальным процессорам x86.

Компания AMD предложила другое решение: она добавила 64-битное расширение к существующей 32-разрядной архитектуре x86. Это позволило использовать физические адреса длинной до 52 бит (архитектурный предел). Данная архитектура была названа x86-64, а затем переименована в AMD64.

Успех процессоров на базе AMD64 привел к тому, что Intel лицензировала набор инструкций AMD64. Новая версия архитектуры получила название EM64T (IA-32e), позже была переименована в Intel 64.

1.2 Режимы работы процессора

Унаследованная (legacy) архитектура x86 предусматривает четыре режима работы процессора:

1. Реальный режим (англ. Real Mode)
2. Защищенный режим (англ. Protected Mode)
3. Режим виртуального 8086 (англ. Virtual-8086 Mode)
4. Режим системного управления (англ. System Management Mode)

Архитектура AMD64 поддерживает все эти режимы и добавляет новый режим, названный «длинный» режим (англ. Long Mode).

1.2.1 Унаследованный режим

Унаследованный режим состоит из трех подрежимов: реальный режим, защищенный режим и режим виртуального 8086. Страничное преобразование в защищенном режиме не является обязательным. Унаследованный режим сохраняет бинарную совместимость не только с существующим 16-битным и 32-битным прикладным ПО, но и с существующим 16-битным и 32-битным системным ПО.

Реальный режим. В данном режиме, также называемом режимом реальных адресов, процессору доступен 1 мегабайт физической памяти. Обработка прерываний и формирование адреса выполняется так же, как и реальном режиме процессора 80286. Страничное преобразование адреса не поддерживается. Все ПО выполняется на нулевом уровне привилегий.

Процессор начинает работу в реальном режиме.

Защищенный режим. В данном режиме процессору доступно 4 гигабайта физической и виртуальной памяти. Доступны все возможности сегментного преобразования и аппаратного переключения задач. Если страничное преобразование не используется – виртуальные адреса совпадают с физическими.

В защищенном режиме ПО выполняется на уровнях привилегий 0-3. Как правило, прикладное ПО выполняется на третьем уровне привилегий, а системное - на 0, 1 и 2.

1.2.2 Длинный режим

Длинный режим включает в себя 2 подрежима: 64-битный режим и режим совместимости. 64-битный режим поддерживает несколько новых возможностей, включая возможность использовать 64-битное адресное пространство. Режим совместимости обеспечивает бинарную совместимость с существующим 16-битным и 32-битным прикладным ПО при работе в 64-битном окружении.

Перед активацией и переходом в длинный режим, операционная система должна перейти в защищенный режим. Процесс перехода в длинный режим описан в главе 1.13.

1.2.3 64-битный режим

64-битный режим – подрежим длинного режима, предусматривает поддержку 64-разрядного ПО. Режим имеет следующие особенности:

1. 64-битные виртуальные адреса.
2. Доступ к битам 63:32 регистров общего назначения.
3. Дополнительные 8 регистров общего назначения (R8-R15).
4. 64-битный счетчик команд (RIP).
5. Плоская модель памяти с одним сегментом кода, данных и стека.

Данный режим может быть активирован системным ПО для различных сегментов кода. В данном режиме механизм сегментного преобразования адреса отключен. Для управления памятью используется механизм страничного преобразования.

Следует обратить внимание на то, что в 64-битном режиме процессор может использовать 52-битные физические адреса (архитектурный предел) и 64-битные виртуальные адреса (из них только первые 48 бит используются механизмом страничного преобразования).

1.2.4 Режим совместимости

Режим совместимости – подрежим длинного режима, позволяет системному ПО обеспечивать бинарную совместимость с существующим 16-битным и 32-битным прикладным ПО, т.е. запускать данное ПО без перекомпиляции в 64-битной ОС в длинном режим.

В режиме совместимости, приложениям доступны только первые 4 гигабайта виртуального адресного пространства.

В данном режиме сегментное преобразование адреса работает так же, как и в унаследованной архитектуре x86. С точки зрения прикладного ПО, режим совместимости не отличается от унаследованного защищенного режима. С точки зрения системного ПО – необходимо использовать механизмы длинного режима для преобразования адресов и обработки исключений и прерываний.

В данной работе режим совместимости будет использоваться только для перехода в 64-битный режим.

1.3 Системные ресурсы

Операционная система выполняет системные операции (управление памятью, изменение режима работы процессора и др.) используя системные ресурсы. Эти ресурсы состоят из системных регистров (управляющих и моделезависимых) и системных структур данных (различные таблицы).

Для доступа к моделезависимым регистрам (англ. MSR) используются команды RDMSR и WRMSR. Данные команды предполагают что в регистре ECX находится номер регистра. Команда RDMSR сохраняет значение MSR в регистры EDX:EAX (все MSR имеют размер 64 бита). Команда WRMSR записывает в регистр MSR значение, содержащееся в EDX:EAX.

Управляющие регистры

Регистры, управляющие работой процессора в архитектуре AMD64, включают:

CR0 — Позволяет изменять ражим работы процессора и управляет некоторыми возможностями процессора.

CR2 — Используется механизмом страничного преобразования. При возникновении страничного исключения, содержит виртуальный адрес по которому произошло исключение.

CR3 — Используется механизмом страничного преобразования. Содержит базовый адрес таблицы страниц верхнего уровня, управляет кешированием данной таблицы.

CR4 — Содержит дополнительные флаги для различных возможностей процессора.

CR8 — Используется для управления приоритетами внешних прерываний.

RFLAGS — Хранит состояние процессора и некоторые управляющие флаги. В основном, используется для управления аппаратным переключением задач, прерываниями и режимом виртуального 8086.

EFER — моделезависимый регистр, содержащий состояние процессора и управляющие флаги, для возможностей, которые не управляются регистрами CR0 и CR4. Для доступа к данному MSR в регистр ECX нужно занести значение **0xC000 0080**.

В унаследованном режиме все управляющие регистры, в том числе RFLAGS – 32-битные. EFER – 64-битный во всех режимах. Архитектура AMD64 расширяет все 32-битные управляющие регистры до 64 бит.

Далее в работе под обозначением **rFLAGS** будет иметься в виду 16-битный, 32-битный или 64-битный регистр флагов. Аналогично, **rIP** – 16-битный, 32-битный или 64-битный счетчик команд.

1.4 Сегментное преобразование адреса

Унаследованная архитектура x86 поддерживает механизм сегментного преобразования адреса, который позволяет системному ПО создавать отдельное виртуальное адресное пространство для каждого процесса. Размер и расположение сегмента в виртуальном адресном пространстве произвольны. Инструкции и данные могут располагаться как в одном, так и в нескольких сегментах, каждому из которых будут назначены отдельные атрибуты доступа.

Механизм сегментного преобразования использует 10 сегментных регистров, каждый из которых определяет один сегмент. 6 из этих регистров (CS, DS, ES, FS, GS и SS) определяют пользовательские сегменты. Пользовательские сегменты содержат команды, данные и стек. Они доступны как для системного, так и для прикладного ПО. Оставшиеся 4 регистра (GDTR, LDTR, IDTR и TR) определяют системные сегменты. Системные сегменты содержат структуры данных инициализируемые и используемые только системным ПО. Сегментные регистры содержат (в теневой части) базовый адрес, указывающий на начало сегмента, лимит сегмента и атрибуты доступа.

Несмотря на то, что сегментное преобразование обеспечивает высокую гибкость в изоляции и защите данных, обычно, эти задачи решаются эффективнее с использованием программной и аппаратной поддержки страничного преобразования. По этой причине большинство современных систем не используют сегментное преобразование.

В длинном режиме, работа сегментного преобразования зависит от того, в каком из подрежимов находится процессор:

— В режиме совместимости сегментное преобразование работает так же, как и в унаследованном режиме.

— В 64-битном режиме сегментное преобразование отключено, задавая плоское 64-битное адресное пространство. Однако некоторые функции сегментных регистров (в частности системных сегментных регистров) продолжают использоваться.

Системное ПО может использовать механизм сегментного преобразования для реализации одной из двух основных моделей: плоская модель памяти (используется один сегмент) и мульти-сегментная модель памяти (используется несколько сегментов).

Плоская модель памяти

Плоская модель памяти – это простейшая форма сегментного преобразования. Плоская модель памяти позволяет системному ПО обойти часть механизмов сегментного преобразования. В плоской модели памяти все базовые адреса сегментов равны 0, а лимиты сегментов равны 4 гигабайтам. Установка базового адреса сегмента в 0 фактически отключает сегментное преобразование (сегмент:смещение = смещение).

Сегментное преобразование в 64-битном режиме

В 64-битном режиме сегментное преоабразование отключено. Аппаратное обеспечение игнорирует значение базового адреса сегмента и обрабатывает его как 0. Игнорируются размер и большинство атрибутов. Системные сегментные регистры всегда используются в 64-битном режиме.

1.4.1 Структуры данных сегментного преобразования

Механизм сегментного преобразования использует следующие структуры данных:

— Дескрипторы сегментов. Описывают сегменты (базовый адрес сегмента в виртуальном адресном пространстве, его лимит, атрибуты доступа и некоторые другие характеристики).

— Таблицы дескрипторов. Сегментные дескрипторы хранятся в памяти в одной из трех таблиц: GDT, LDT, IDT.

— Сегмент состояния задачи (TSS). Специальный тип системного сегмента, который содержит информацию о состоянии процесса (задачи), в том числе ссылки на необходимые процессу структуры данных.

— Сегментные селекторы. Используются для выбора дескрипторов из таблиц дескрипторов.

Механизм сегментного преобразования использует следующие регистры: CS, DS, ES, SS, GS, FS, GDTR, IDTR, LDTR, TR. Структуры данных связаны с регистрами следующим образом:

— Сегментные регистры (CS, DS, ES, FS, GS, SS). Используются чтобы ссылаться на пользовательские сегменты. При загрузке селектора сегмента в сегментный регистр процессор автоматически загружает выбранный дескриптор в теневую часть сегментного регистра.

— Регистры таблиц дескрипторов (GDTR, LDTR, IDTR). Задают виртуальный базовый адрес и лимит таблиц дескрипторов.

— Регистр задачи (TR). Задает положение и размер текущего сегмента состояния задачи (TSS).

Сегментные селекторы

Селекторы сегментов указывают на дескрипторы в GDT и LDT. Формат селектора сегмента показан на рис. 1.1.

Рисунок 1.1 — Формат селектора сегмента

Селектор состоит из следующих полей:

1. Индекс. Биты 15:3. Указывает на элемент в таблице дескрипторов. Дескрипторы имеют размер 8 байт, поэтому индекс умножается на 8 чтобы получить смещение в таблице дескрипторов. Смещение прибавляется к базовому адресу GDT или LDT (в зависимости от значения TI), чтобы получить виртуальный адрес дескриптора.

Некоторые элементы таблицы дескрипторов имеют размер 16 байт, вместо 8. Они занимают 2 элемента в таблице. Однако в длинном режиме смещение по прежнему вычисляется путем умножения индекса на 8. Системное ПО должно назначать селекторы так, чтобы они указывали на начало расширенных элементов.

2. Индикатор таблицы (TI). Бит 2. Указывает в какой таблице хранится дескриптор. Если бит сброшен в 0, это означает что селектор ссылается на запись в GDT. Иначе - селектор ссылается на запись в LDT.

3. Уровень привилегий (RPL). Биты 1:0. Равен уровню привилегий на котором находился процессор (CPL) при создании селектора. Используется для проверки прав доступа.

Нулевые селекторы (индекс 0 и TI=0) используются чтобы сделать сегментные регистры недействительными (аналог нулевых указателей). При использовании

сегментного регистра (не в 64-битном режиме), содержащего нулевой селектор произойдет исключение #GP.

Сегментные регистры

Для улучшения производительности, при загрузке нового значения в сегментный регистр, процессор загружает дескриптор, соответствующий селектору, в теневую часть сегментного регистра. Благодаря этому снижается число обращений к памяти.

На рис. 1.2 показаны видимая и теневая части сегментного регистра. ПО не имеет прямого доступа к теневой части сегментных регистров (кроме GS и FS).

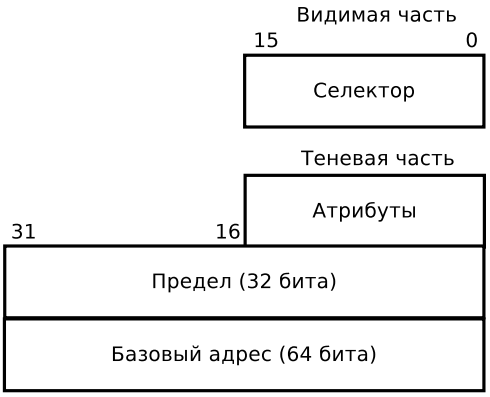

Рисунок 1.2 — Формат сегментного регистра

При вычислении адреса, базовый адрес считается равным 0, выход за пределы сегмента не проверяется. Вместо этого выполняется проверка, что адрес находится в канонической форме. Содержимое теневой части регистров DS, ES и SS игнорируется полностью.

1.4.2 Таблицы дескрипторов

Механизм сегментного преобразования использует таблицы дескрипторов. Эти таблицы содержат дескрипторы, которые описывают расположение сегмента в виртуальной памяти, его лимит (размер - 1) и атрибуты доступа. Обращение к полям дескриптора происходит при каждом обращении к памяти (выборка инструкций, чтение/запись данных).

Как было сказано ранее, архитектура x86 поддерживает 3 типа таблиц дескрипторов: GDT, LDT (как правило, не используется), IDT.

GDT

Для перехода в защищенный режим необходимо создать GDT. GDT содержит дескрипторы сегментов кода и данных для сегментов, которые являются общими для всех задач. Кроме пользовательских сегментов, GDT может содержать дескрипторы шлюзов и другие системные дескрипторы. Системное ПО может расположить GDT в произвольной области памяти, недоступной непривилегированному ПО.

На рис. 1.3 показано как происходит доступ к GDT/LDT.

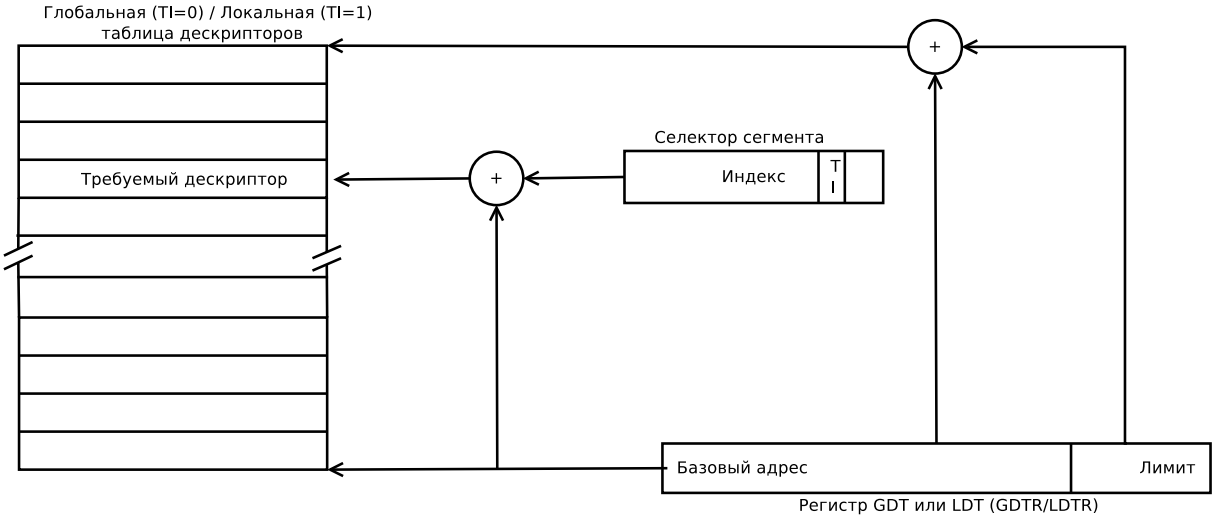

Рисунок 1.3 — Доступ к GDT/LDT

Регистр глобальной таблицы дескрипторов

Регистр GDT (GDTR) содержит базовый адрес и размер GDT. Регистр загружается командой LGDT. На рис. 1.4 показан формат GDTR в унаследованном режиме и режиме совместимости. На рис. 1.5 показан формат GDTR в длинном режиме.

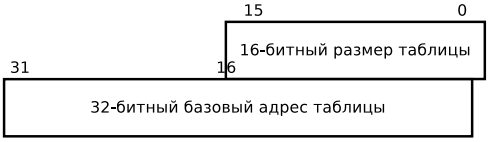

Рисунок 1.4 — Формат GDTR и IDTR в унаследованном режиме

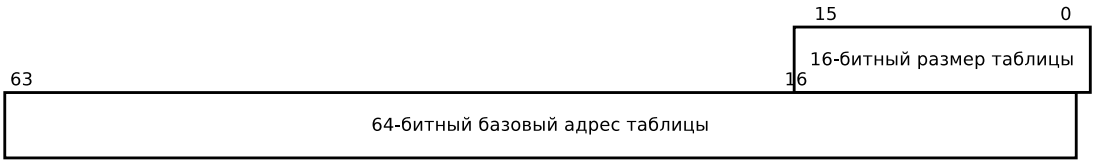

Рисунок 1.5 — Формат GDTR и IDTR в длинном режиме

GDTR состоит из двух полей:

Лимит — 2 байта. Задает лимит GDT в байтах. При обращении ПО за пределы GDT произойдет исключение #GP.

Базовый адрес — 8 байт. Содержит виртуальный адрес начала таблицы. GDT может быть расположена по любому адресу, однако системному ПО следует использовать адрес, выровненный по 4-байтной границе, чтобы избежать снижения производительности из-за доступа к невыровненным данным.

В архитектуре AMD64 размер базового адреса в GDTR увеличен до 64 бит, что позволяет системному ПО, работающему в длинном режиме, расположить GDT в произвольном месте 64-битного адресного пространства. В унаследованном режиме процессор игнорирует старшие 4 байта.

IDT

IDT, так же как GDT и LDT, может быть расположена в произвольной области памяти, недоступной непривилегированному ПО.

IDT может содержать дескрипторы следующих типов:

— Шлюз прерывания (англ. interrupt gate)

— Шлюз ловушки (англ. trap gate)

— Шлюз задачи (англ. task gate)

В главе 1.11 описано, как механизм обработки прерываний использует дескрипторы шлюзов.

Обращение к элементам IDT происходит по номеру вектора прерывания. Смещение в таблице вычисляется путем умножения номера вектора прерывания на размер элемента таблицы. Размер элемента таблицы зависит от режима работы процессора следующим образом:

— В длинном режиме размер элемента IDT составляет 16 байт.

— В унаследованном режиме размер элемента IDT составляет 8 байт.

На рис. 1.6 показано, как происходит индексация в IDT по номеру вектора прерывания.

Регистр IDT

Регистр IDT (IDTR) содержит базовый адрес и размер IDT. Содержимое регистра загружается командой LIDT. Формат IDTR совпадает с форматом GDTR во

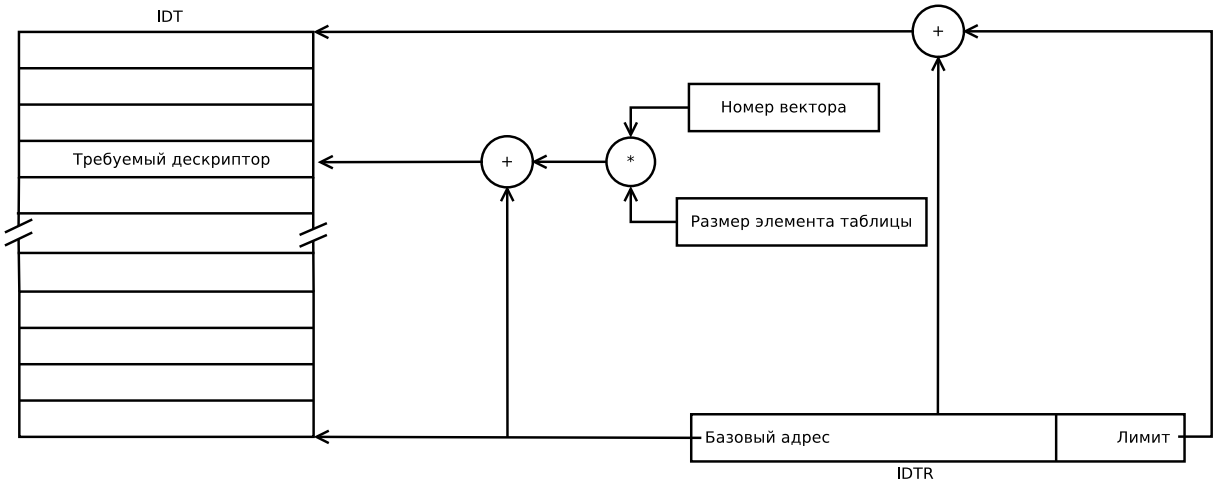

Рисунок 1.6 — Индексация в IDT

всех режимах работы процессора. На рис. 1.4 показан формат IDT в унаследованном режиме, а на рис. 1.5 – в длинном режиме.

1.4.3 Унаследованные дескрипторы сегментов

Формат дескриптора

Дескрипторы сегментов определяют и изолируют сегменты друг от друга. Существует 2 основных типа дескрипторов, каждый из которых используется для описания сегментов (шлюзов) разных типов:

— Дескрипторы пользовательских сегментов – дескрипторы сегментов кода и данных (в том числе стека).

— Дескрипторы системных сегментов – дескрипторы LDT, TSS и шлюзов (описывают программные точки входа).

На рис. 1.7 показан общий формат дескриптора сегмента в унаследованном режиме. В унаследованном режиме размер сегмента составляет 8 байт (2 двойных слова). На рисунке старшее двойное слово (смещение +4) изображено вверху, младшее – внизу.

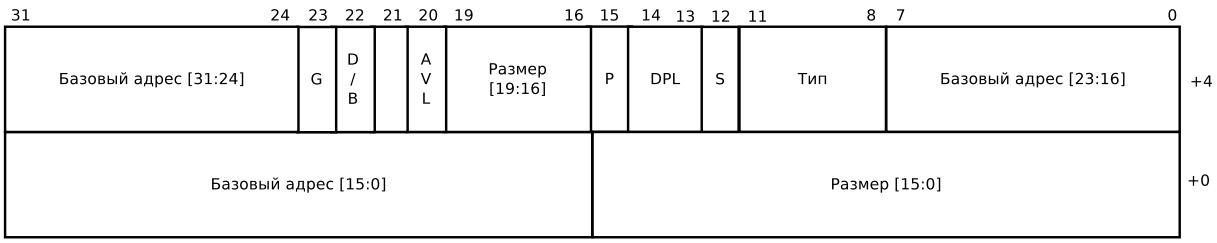

Рисунок 1.7 — Общий формат дескриптора в унаследованном режиме

Дескриптор имеет следующие поля:

— Лимит сегмента. 20-битный лимит сегмента формируется путем объединения бит 19:16 старшего двойного слова и 0:15 младшего двойного слова.

— Базовый адрес. 32-битный базовый адрес формируется путем объединения бит 31:24 и 7:0 старшего двойного слова с битами 15:0 младшего двойного слова. Содержит адрес начала сегмента в виртуальной памяти.

— Бит «S». Если равен 0 – системный сегмент (LDT, TSS, шлюз), иначе – пользовательский (код, данные).

— «Тип». Определяет тип сегмента.

— «DPL». Определят уровень привилегий дескриптора. Может иметь значения от 0 до 3, где 0 – наибольший уровень привилегий, 3 – наименьший.

— Бит «P». Определяет присутствует (загружен) ли сегмент в памяти.

— «AVL». Доступно для использования системным ПО.

— Бит «D/B». Размер операнда по умолчанию.

— Бит «G». Бит гранулярности – определяет как обрабатывать размер сегмента. Если равен 0 – размер сегмента задается в байтах. Если равен 1 – размер сегмента задан в 4-килобайтных блоках.

Дескрипторы сегментов кода

На рис. 1.8 показан формат дескриптора сегмента кода. Сегменты кода определяют режим работы процессора и уровень привилегий. Сегменты кода доступны только для исполнения, либо только для чтения и исполнения. Запись в сегмент кода, на который ссылается регистр CS, запрещена.

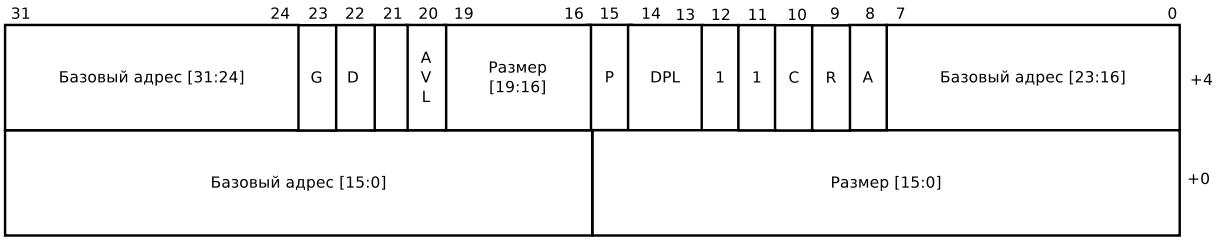

Рисунок 1.8 — Формат дескриптора сегмента кода в унаследованном режиме

Для дескрипторов сегментов кода бит «S» установлен в 1, означая что это пользовательский сегмент. Бит 11 используется чтобы отличать сегменты кода и сегменты данных (если бит установлен в 1 – это сегмент кода). Биты 10:8 определяют характеристики доступа к сегменту кода:

— Бит «C». Управляет CPL при передаче управления.

— Бит «R». Разрешает чтение данных из сегмента.

— Бит «A». Устанавливается при копировании дескриптора в регистр CS.

Дескрипторы сегментов данных

На рис. 1.9 показан формат дескриптора сегмента данных. Сегменты данных могут быть доступны либо только для чтения, либо для чтения/записи. Доступ к сегментам данных осуществляется с использованием регистров DS, ES, FS, GS, SS. Регистр DS содержит селектор сегмента данных, используемый по умолчанию.

Сегмент стека это одна из форм сегмента данных. Доступ к нему осуществляется через регистр SS. Он должен быть доступен для чтения и записи.

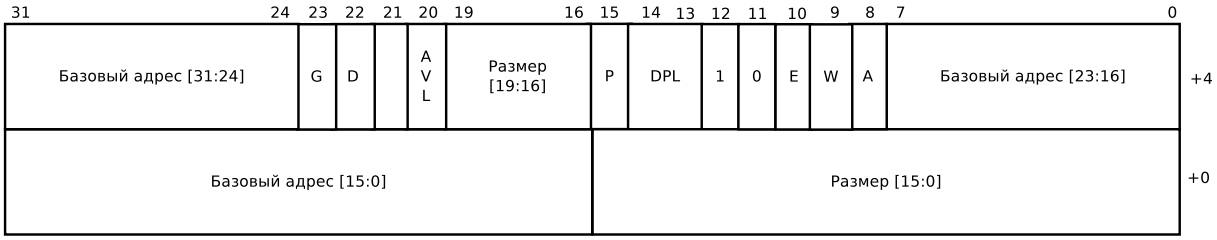

Рисунок 1.9 — Формат дескриптора сегмента данных в унаследованном режиме

Для дескрипторов сегментов данных бит «S» установлен в 1, означая что это пользовательский сегмент. Бит 11 сброшен в 0, обозначая сегмент данных. Биты 10:8 определяют характеристики доступа к сегменту:

— Бит «E». Определяет расширяющийся вниз сегмент.

— Бит «W». Разрешает операции записи в сегменте.

— Бит «A». Устанавливается при копировании дескриптора в DS, ES, FS, GS или SS.

1.4.4 Сегментные дескрипторы длинного режима

Дескрипторы сегмента кода

В длинном режиме сегменты кода продолжают использоваться. Сегменты кода, их дескрипторы и селекторы необходимы для установки режима работы процессора и уровня привилегий. Атрибут «L» определяет в каком режиме работает процессор – в 64-битном или режиме совместимости. На рис. 1.10 показан формат дескриптора сегмента кода в длинном режиме. В режиме совместимости все поля дескриптора интерпретируются так же, как и в унаследованном режиме.

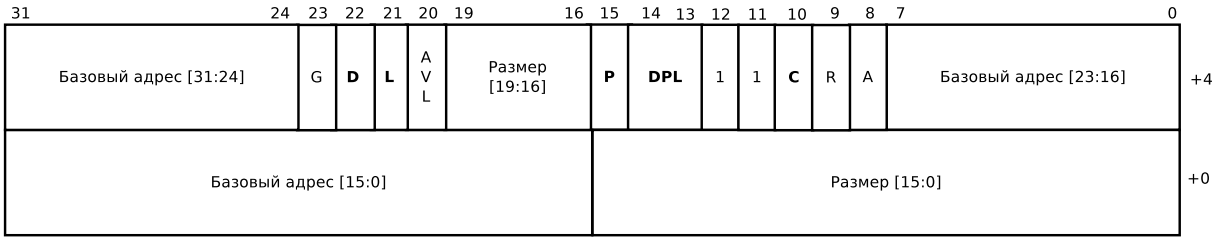

Рисунок 1.10 — Формат дескриптора сегмента кода в длинном режиме

Игнорируемые в 64-битном режиме поля. В 64-битном режиме сегментное преобразование отключено, сегменты кода занимают все адресное пространство. В этом режиме значение поля «базовый адрес» игнорируется. При вычислении виртуального адреса оно считается равным 0.

Проверка выхода за границу сегмента не выполняется, игнорируются размер и бит «G». Вместо этого выполняется проверка, что адрес находится в канонической форме.

Также игнорируются биты «R» и «A» в поле «тип».

Бит «L». В длинном режиме у дескриптора кода появился дополнительный атрибут – бит «L». Если он равен 1 – процессор работает в 64-битном режиме, иначе – в режиме совместимости. В унаследованном режиме этот бит игнорируется.

Режим совместимости имеет бинарную совместимость с существующим 16-битным и 32-битным прикладным ПО. Переход в режим совместимости осуществляется на основании атрибутов сегмента кода, это позволяет 64-битному системному ПО исполнять унаследованное ПО вместе с 64-битным ПО. Для запуска унаследованных 16-битных и 32-битных приложений системному ПО достаточно сбросить бит «L» в дескрипторе сегмента кода в 0.

Если процессор работает в 64-битном режиме (L=1) – бит «D» должен быть равен 0. Это приводит к тому, что размер операнда по умолчанию составляет 32 бита, а размер адреса по умолчанию составляет 64 бита. Комбинация L=1 и D=1 зарезервирована для использования в будущем.

Дескрипторы сегмента данных

В длинном режиме сегменты данных продолжают использоваться. На рис. 1.11 показан формат дескриптора сегмента данных в длинном режиме. В режиме совместимости все поля дескриптора интерпретируются также, как и в унаследованном режиме.

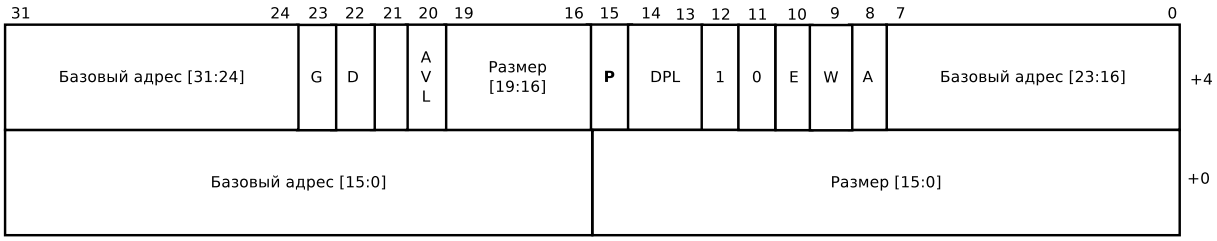

Рисунок 1.11 — Формат дескриптора сегмента данных в длинном режиме

Игнорируемые в 64-битном режиме поля. В 64-битном режиме сегментное преобразование отключено. Интерпретация базового адреса зависит от используемого сегментного регистра:

— При использовании регистров DS, ES, SS поле «базовый адрес» игнорируется и считается равным 0.

— Регистры GS и FS обрабатываются специальным образом. При использовании этих регистров можно использовать ненулевое значение базового адреса для вычисления виртуального адреса.

Игнорируются все атрибуты дескриптора, кроме бита «P».

Системные дескрипторы

Как показано на рис. 1.12 в 64-битном режиме системные дескрипторы LDT и TSS увеличены на 64 бита. Увеличение дескрипторов, позволяет хранить в них 64-битный базовые адреса, поэтому сегменты, которые они описывают могут быть расположены в произвольном месте в памяти. Расширенные дескрипторы могут быть загружены в соответствующие регистры (LDTR или TR) только из 64-битного режима.

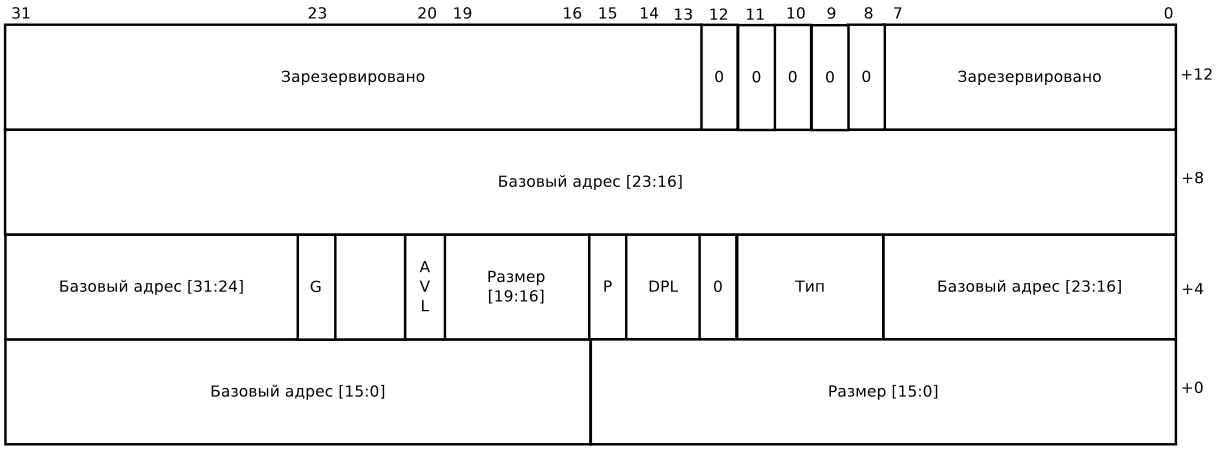

Рисунок 1.12 — Формат системного дескриптора в длинном режиме

Базовый адрес 64-битного системного сегмента должен быть в канонической форме, иначе при загрузке сегмента произойдет исключение #GP. Выход за границу системного сегмента проверяется и в 64-битном режиме и в режиме совместимости с учетом бита гранулярности «G».

На рис. 1.12 показано, что биты 12:8 двойного слова +12 должны быть сброшены в 0. Эти биты соответствуют биту «S» и полю «тип» в унаследованном дескрипторе. Сброс этих бит в 0 соответствует неправильному типу дескриптора в унаследованном режиме и приведет к возникновению исключения #GP при попытке обратиться отдельно к старшей половине 64-битного системного сегмента.

Дескрипторы шлюзов

В длинном режиме дескрипторы шлюзов увеличены на 64 бита, что позволяет хранить в них 64-битные смещения. Формат 64-битного дескриптора шлюза прерывания и шлюза ловушки показан на рис. 1.13.

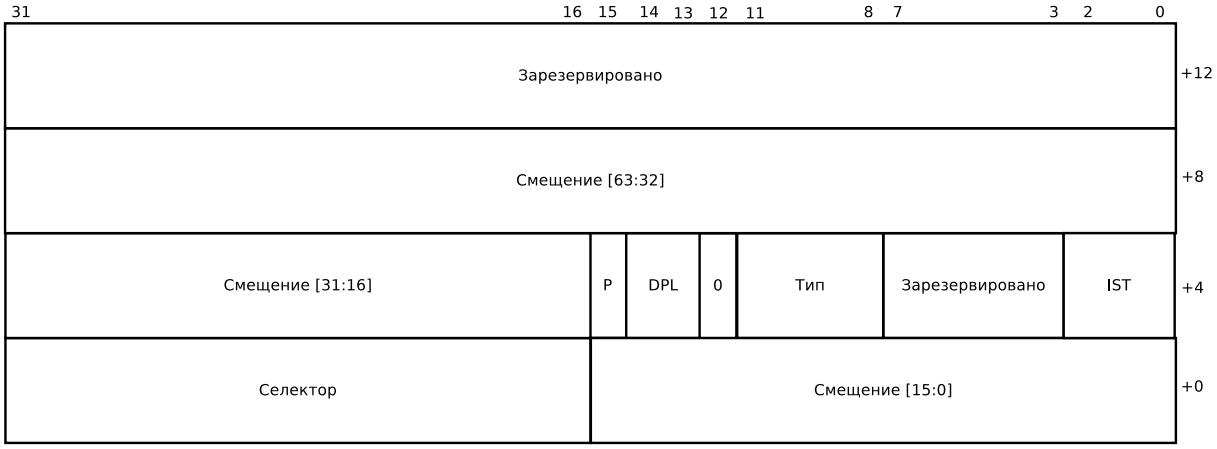

Рисунок 1.13 — Формат дескриптора шлюза прерывания и шлюза ловушки в длинном режиме

Селектор сегмента (в дескрипторе шлюза) должен указывать на 64-битный сегмент кода (CS.L=1, CS.D=0). В противном случае, при обращении к дескриптору произойдет исключение #GP.

Исключение также возникает, если адрес, содержащийся в поле «смещение», находится не в канонической форме.

В 64-битном режиме элементы таблицы векторов прерываний имеют размер 128 бит. Процессор автоматически умножает номер вектора прерывания на 16 для определения смещения.

Поле «IST». Биты 2:0 байта +4. В длинном режиме в дескрипторах шлюзов прерываний и ловушек появилось новое 3-битное поле – IST. Данное поле используется

в качестве индекса в IST фрагменте TSS длинного режима. Если IST не равно 0 – индекс ссылается на элемент IST в TSS, значение которого процессор загружает в регистр RSP при возникновении прерывания. Если IST равно 0 – процессор использует унаследованный механизм переключения стека.

1.5 Ограничение доступа к памяти

Архитектура AMD64 разработана, чтобы полностью поддерживать унаследованный механизм защиты сегментов. Данный механизм позволяет системному ПО ограничивать процессам доступ к данным и коду других процессов.

Защита на уровне сегментов включена в режиме совместимости. 64-битный режим устраняет часть проверок, оставляя только проверки доступа к таблицам системных дескрипторов.

Предпочтительный метод организации защиты памяти в операционной системе длинного режима – использование механизма страничного преобразования.

Концепция уровней доступа

Механизм защиты сегментов используется чтобы изолировать и защищать код и данные различных процессов. В защищенном режиме (CR0.PE=1) данный механизм поддерживает 4 уровня привилегий. Уровни привилегий обозначаются номерами от 0 до 3, где 0 обозначает наибольший уровень привилегий, 3 – наименьший. Как правило, уровень 0 используется системным ПО, уровень 3 – прикладным ПО, а уровни 1 и 2 не используются.

Типы уровней привилегий

Существует три типа уровней привилегий, используемых процессором для контроля доступа к сегментам: CPL, DPL и RPL.

CPL. Текущий уровень привилегий – уровень привилегий, на котором процессор находится в данный момент. Он хранится во внутреннем регистре процессора, недоступном для ПО. Изменить текущий уровень привилегий можно путем передачи управления на сегмент кода с другим уровнем привилегий.

DPL. Уровень привилегий дескриптора – уровень привилегий, который системное ПО назначает сегменту (шлюзу). Используется при проверке прав доступа, чтобы определить можно ли приложению обращаться к сегменту (шлюзу), на который ссылается дескриптор. Данное поле хранится в дескрипторе сегмента (шлюза).

RPL. Отражает уровень привилегий процесса, который создал селектор сегмента (шлюза). RPL может быть использован в вызываемой программе, чтобы определить уровень привилегий вызывающей программы. Данное поле хранится в селекторе сегмента (шлюза).

Дополнительная информация о проверках прав доступа с использованием CPL, DPL и RPL приведена в [1].

1.6 Страничное преобразование

Механизм страничного преобразования x86 позволяет системному ПО создавать отдельные адресные пространства для различных процессов. Эти адресные пространства известны как виртуальные адресные пространства. Системное ПО использует механизм страничного преобразования для отображения виртуальных страниц на физические, используя иерархию таблиц страничного преобразования, известных как таблицы страниц.

Механизм страничного преобразования и таблицы страниц используются чтобы обеспечить каждый процесс областью физической памяти для хранения кода и данных, недоступной другим процессам. Процесс не может получить доступ к физической памяти, которая не отображена в его адресное пространство системным ПО.

Системное ПО может использовать механизм страничного преобразования чтобы отобразить одну физическую страницу в различные адресные пространства. Такие страницы можно делать доступными только для чтения, чтобы предотвратить их модификацию процессами.

Общие страницы, как правило, используются для организации доступа к разделяемым библиотекам из разных процессов. Доступная только для чтения копия библиотеки отображается в виртуальное адресное пространство каждого процесса, однако в физической памяти находится только одна копия библиотеки. Данная возможность также позволяет хранить копию операционной системы и различных драйверов устройств в адресном пространстве процессов. Это позволяет избежать накладных расходов связанных с переключением на другое адресное пространство при обращении к функциям ОС.

Область адресного пространства, отведенная для системного ПО, содержит системные данные, которые должны быть недоступны для прикладного ПО. Системное ПО использует таблицы страниц для защиты этих данных, отмечая такие страницы как доступные только супервизору, тем самым запрещая доступ к ним со стороны прикладного (непривилегированного) ПО.

Системное ПО может использовать механизм страничного преобразования, чтобы отображать большие виртуальные адресные пространства в меньший объем физической памяти. Каждому процессу доступно 32-битное или 64-битное виртуальное адресное пространство. Системное ПО использует свободные физические страницы для отображения наиболее часто используемых виртуальных страниц. Наименее часто используемые виртуальные страницы выгружаются на диск.

1.6.1 Механизм страничного преобразования

Унаследованная архитектура x86 поддерживает преобразование 32-битных виртуальных адресов в 32-битные физические. Архитектура AMD64 расширяет эту возможность, позволяя преобразовывать 64-битные виртуальные адреса в 52-битные физические (различные реализации процессоров могут поддерживать более короткие виртуальные и физические адреса).

Виртуальные адреса преобразуются в физические, используя иерархию таблиц страниц, созданную и управляемую системным ПО. Элементы таблиц указывают на таблицы страниц следующего уровня. Одна таблица может содержать до 1024 элементов, каждый из которых указывает на таблицу страниц следующего уровня. Элементы таблицы страниц заключительного уровня указывают на физические страницы.

На рис. 1.14 показана иерархия таблиц страниц, использующаяся в длинном режиме. Как показано на рисунке, виртуальный адрес делится на несколько частей, каждая из которых используется как смещение в соответствующей таблице страниц. Младшая часть виртуального адреса используется как смещение в физической странице.

Опции страничного преобразования

Режим страничного преобразования зависит от того, какие опции были активированы. Существует 4 опции, влияющие на страничное преобразование:

— Активация страничного преобразования (бит CR0.PG)

— Расширение физических адресов (бит CR4.PAE)

— Расширение размеров страниц (бит CR4.PSE)

— Активация длинного режима (бит EFER.LME)

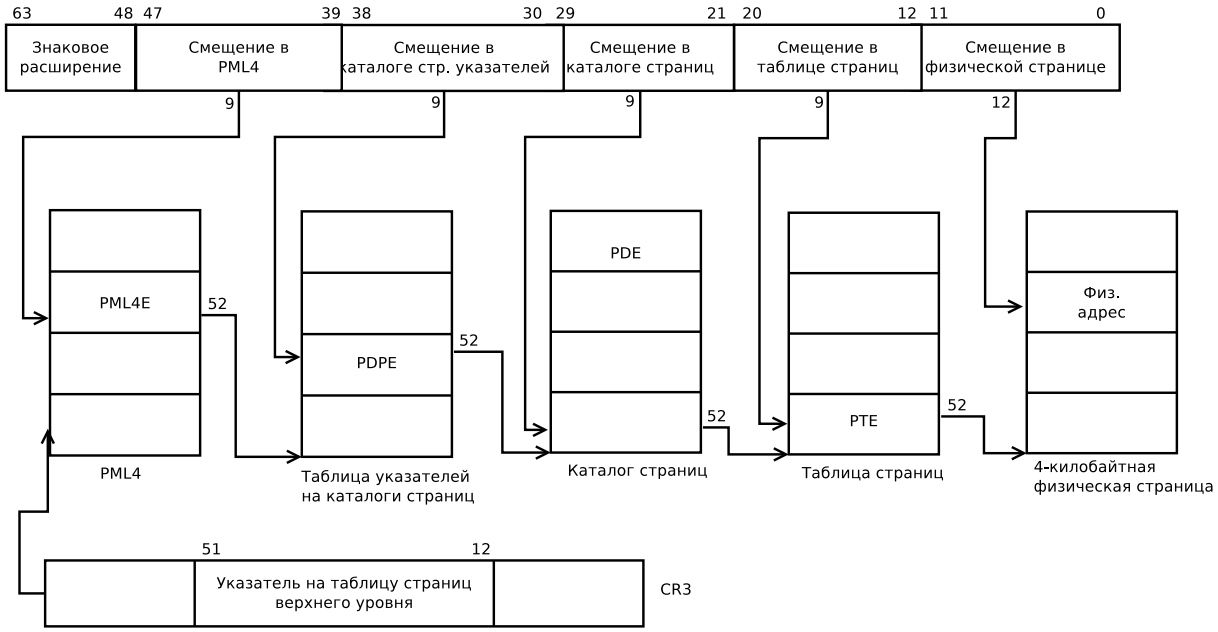

Рисунок 1.14 — Использование 4-килобайтных страниц в длинном режиме

Активация страничного преобразования

Страничное преобразование управляется битом «PG» регистра CR0 (бит 31). Если бит CR0.PG установлен в 1 – страничное преобразование включено, иначе – выключено.

Расширение физических адресов (бит «PAE»)

Возможность расширения физических адресов управляется битом «PAE» в регистре CR4 (бит 5). Если бит CR4.PAE установлен в 1 – используется расширение физических адресов, в противном случае – не используется.

Установка CR4.PAE в 1 включает поддержку преобразования виртуальных адресов в 52-битные физические адреса. Это приводит к увеличению длинны полей структур данных страничного преобразования с 32 бит до 64 (для возможности хранения расширенных физических адресов).

1.6.2 Страничное преобразование в длинном режиме

Для работы страничного преобразования в длинном режиме необходимо использовать PAE. CR4.PAE должен быть установлен в 1 до перехода в длинный режим, в противном случае произойдет исключение #GP.

Структуры данных страничного преобразования при включенном PAE позволяют отображать 64-битные виртуальные адреса в 52-битные физические. PAE

расширяет элементы каталога страниц (PDE) и элементы таблицы страниц (PTE) до 64 бит, позволяя использовать физические адреса длинной более 32 бит.

Архитектура AMD64 расширяет формат PDPE, определяя ранее зарезервированные биты. Также была добавлена таблица страниц 4го уровня (PML4).

Так как в длинном режиме PAE всегда включен – бит «PS» в элементах каталога страниц (PDE.PS) позволяет выбирать между 4-килобайтными и 2-мегабайтными страницами.

Регистр CR3

В длинном режиме регистр CR3 используется для указания базового адреса PML4. Размер регистра CR3 был увеличен до 64 бит, чтобы иметь возможность хранить PML4 в произвольном месте физического адресного пространства. На рис. 1.15 показан формат регистра CR3 в длинном режиме.

Рисунок 1.15 — Формат регистра CR3 в длинном режиме

Регистр CR3 имеет следующие поля:

— Базовый адрес. Содержит физический адрес таблицы страниц верхнего уровня. Это 40-битное поле, занимает биты 51:12 и указывает на начало PML4. Адрес PML4 выровнен по 4-килобайтной границе (младшие 12 бит равны 0).

— Бит «PWT» (3). Бит сквозной записи. Определяет политику кеширования.

— Бит «PCD» (4). Бит отключения кеширования.

— Зарезервированные поля. Должны быть сброшены в 0 при записи нового значения в регистр CR3.

Использование 4-килобайтных страниц

При использовании 4-килобайтных физических страниц в длинном режиме виртуальный адрес делится на 6 частей. 4 из них используются в качестве индексов в таблицах страниц. Как показано на рис. 1.14, поля виртуального адреса используются следующим образом:

— Биты 63:48 – знаковое расширение бита 47, как этого требует каноническая форма адреса.

— Биты 47:39 – индекс в PML4.

— Биты 38:30 – индекс в PDP.

— Биты 29:21 – индекс в каталоге страниц.

— Биты 20:12 – индекс в таблице страниц.

— Биты 11:0 – смещение в физической странице.

На рис. 1.16 показан формат элемента таблицы страниц.

Каждая таблица имеет размер 4 килобайта и содержит 512 64-битных элементов. Поля этих элементов описаны в 1.6.3.

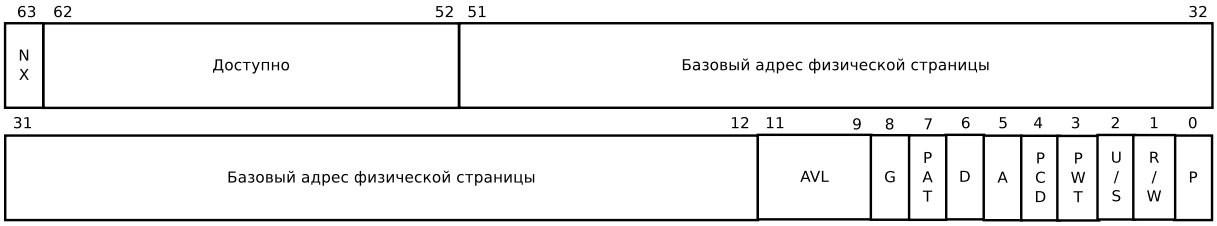

Рисунок 1.16 — Элемент таблицы страниц при использовании 4-килобайтных страниц в длинном режиме

1.6.3 Поля элементов таблиц страниц

Элементы таблиц страниц содержат контрольные и информационные поля, используемые для управления страницами в виртуальной памяти.

Базовый адрес таблицы страниц. Содержит физический адрес таблицы страниц следующего уровня. Адреса таблиц страниц всегда выровнены по границе 4 килобайта, поэтому необходимо хранить только биты, старше 11 (т.к. биты 11:0 всегда равны 0).

Базовый адрес страницы. Содержит базовый адрес физической страницы. Страницы могут иметь размер 4 Кб, 2 Мб, 4 Мб или 1 Гб. Их адреса всегда выровнены по границе равной размеру страницы.

Бит «P». Бит 0. Показывает находится ли таблица/физическая страница в памяти. Если сброшен в 0 – таблицы/физической страницы в памяти нет.

ОС сбрасывает этот бит, чтобы показать, что таблица/страница выгружена. При доступе к такой таблице/странице произойдет страничное исключение (#PF). ОС должна загрузить отсутствующую страницу/таблицу в память и установить данный бит в 1.

Когда этот бит сброшен в 0 – все остальные биты доступны для ОС. Такие страницы никогда не модифицируются процессором и не попадают в TLB.

Бит «R/W». Бит 1. Этот бит контролирует возможность чтения/записи для всех физических страниц, отображенных данным элементом таблицы. Если бит установлен в 1 – страницы доступны для чтения и записи, в противном случае – только для чтения.

Бит «U/S». Бит 2. Этот бит контролирует возможность доступа непривилегированного ПО (CPL=3) ко всем страницам, отображенным данным элементом таблицы. Если бит установлен в 1 – доступ разрешен для всех уровней привилегий, в противном случает – только для уровней привилегий 0, 1 и 2.

Бит «PWT». Бит 3. Определяет политику кеширования таблицы/страницы.

Бит «PCD». Бит 4. Отключает кеширование таблицы/страницы.

Бит «A». Бит 5. Показывает был ли доступ к таблице/странице.

Бит «D». Бит 6. Показывает была ли страница модифицирована.

Бит «PS». Бит 7. Позволяет изменять размер используемых страниц.

Бит «G». Бит 8. Определяет является ли страница глобальной.

Биты «AVL». Данные биты доступны для использования системным ПО.

Бит «PAT». Используется механизмом страничных атрибутов.

Бит «NX». Бит 63. Запрещает исполнение кода на странице.

Зарезервированные биты. Должны быть сброшены в 0.

1.6.4 Кеш TLB

При включенном страничном преобразовании виртуальные адреса автоматически преобразуются в физические, используя иерархию таблиц страниц. Кеши TLB, служат для уменьшения накладных расходов, связанных с преобразованием. TLB – аппаратный кеш, который содержит результаты последних отображений виртуальных адресов в физические. При каждом обращении к памяти, адрес проверяется в TLB. Если отображение найдено – результат немедленно возвращается процессору, это позволяет избежать лишних обращений к памяти (для доступа к таблицам страниц).

Системное ПО должно инвалидировать записи в TLB при внесении изменений в структуры данных страничного преобразования (при удалении отображения, изменении виртуального адреса, либо ограничении прав доступа).

1.7 Организация физической памяти

В 64-битном режиме может использоваться до 48 бит виртуального адреса (9 + 9 + 9 + 9 + 12), таким образом доступное адресное пространство составляет 256 терабайт.

Как правило, ядро отображается в виртуальное адресное пространство каждого процесса (но при этом код ядра остается доступным только на нулевом уровне привилегий), что позволяет снизить накладные расходы при обращении к сервисам ядра.

Существует 2 основных способа расположения ядра в виртуальной памяти: в верхней части (англ. Higher Half Kernel) и в нижней. Основным преимуществом расположения ядра в верхней части виртуального адресного пространства является то, что:

— 32-битные процессы могут использовать все 4 гигабайта адресного пространства;

— Приложения не зависят от размеров адресного пространства ядра. (Если ядро расположено в нижней части адресного пространства, то при изменении размеров ядра может потребоваться перекомпиляция существующих прикладных программ.)

Поэтому было решено разместить ядро в верхней части виртуального адресного пространства. Область, занимаемая ядром в виртуальной памяти, будет расположена начиная с адреса 0xFFFF FFF8 0000 0000, т.к. это позволит иметь прямой доступ к первым 32 гигабайтам физической памяти. Этого будет достаточно для учебной операционной системы. Далее в тексте для обозначения базового адреса ядра будет использоваться обозначение **KERNEL_BASE**.

Ниже **KERNEL_BASE** расположены стек, отображения и структуры данных необходимые для работы ядра.

Процессам уровня пользователя выделен 1 терабайт (0x0000 0100 0000 0000) виртуального пространства. Далее в тексте для обозначения верхнего лимита пространства пользователя будет использоваться обозначение **USER_TOP**.

На рис. 1.17 показана схема организации виртуальной памяти ОС.

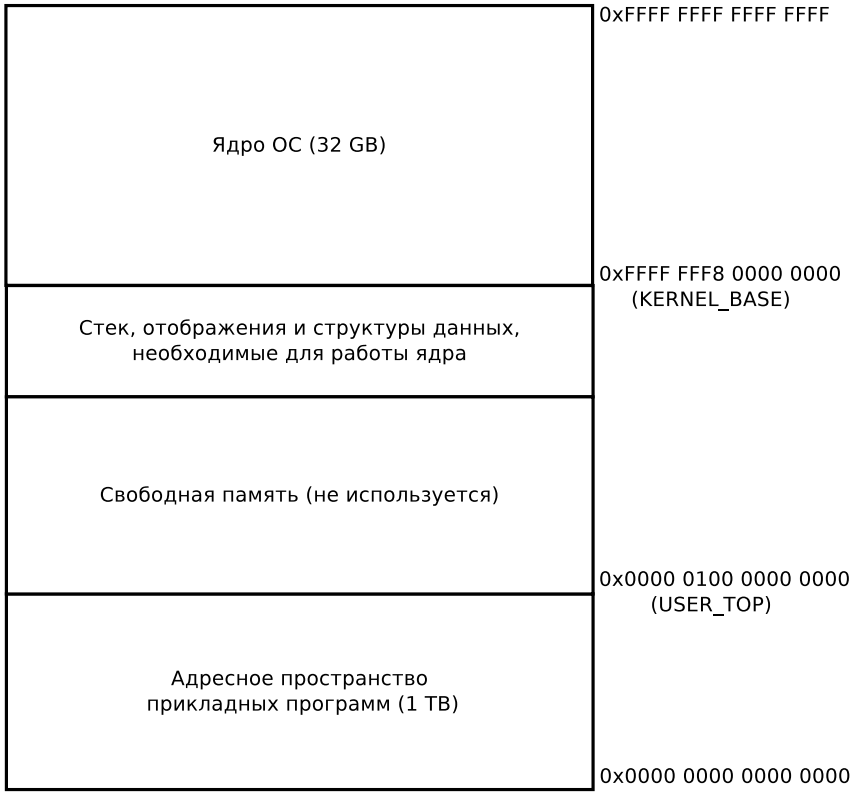

Рисунок 1.17 — Организация памяти ОС

1.8 Расположение ядра на диске

BIOS считывает первый сектор диска в память по адресу 0x7C00 и передает на него управление. Для упрощения кода загрузчика было решено использовать следующие соглашения: в первом секторе диска располагается первый загрузчик, начиная со второго сектора располагается второй загрузчик, ядро начинается с 2048-го сектора (1 мегабайт). На второй загрузчик отведен почти 1 мегабайт дискового пространства, чтобы его можно было расширять, без необходимости перемещать ядро. На рис. 1.18 приведена схема использования дискового пространства.

Рисунок 1.18 — Использование дискового пространства

1.9 Использование стека при вызове функций

В программах на языке C для архитектуры **x86** регистры **esp** (**rsp**) и **ebp** (**rbp**) обычно имеют специальное назначение. Указатель стека **esp** указывает на текущую точку разделения между свободной и используемой областями стека. В архитектуре AMD64, 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.

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 зна чение регистра **esp** в регистр ebp. Типичный код пролога в нотации ассемблера UNIX **ebp** ниже, вторая инструкция здесь копирует содержимое регистра **esp** в регистр **ebp**.

```
1 push %ebp
2 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). 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.

1.10 Сегмент состояния задачи (TSS)

Задача (также называемая процессом) это программа, которую процессор может исполнять, приостанавливать и позже продолжать исполнение с места последней остановки. Пока задача приостановлена могут исполняться другие задачи. Каждая задача имеет свой контекст, в который входят: rIP, rFLAGS, CR3, сегмент кода и данный и некоторые другие ресурсы.

1.10.1 Ресурсы управления задачами

В длинном режиме операционная система должна инициализировать некоторые ресурсы управления задачами:

1. Сегмент состояния задачи (TSS) – сегмент, в котором хранится состояние процессора, связанное задачей.

2. Дескриптор TSS – дескриптор сегмента, описывающий TSS. Формат описан ранее в главе .

3. Селектор TSS – селектор сегмента, который ссылается на дескриптор TSS, расположенный в GDT, т.е. бит TI всегда равен 0.

4. Регистр задачи (TR) – регистр, в котором хранится селектор и дескриптор TSS текущей задачи.

Регистр задачи (TR)

Регистр задачи хранит адрес TSS, определяет его лимит и атрибуты. TR состоит из двух частей – видимой и теневой. В видимой (доступной для ПО) части хранится селектор TSS, теневая часть содержит дескриптор TSS. При загрузке селектора TSS в TR процессор автоматически загружает дескриптор TSS из GDT в теневую часть. Загрузка нового значения в TR осуществляется инструкцией LTR. На рис. 1.19 показан формат TR в длинном режиме.

Рисунок 1.19 — Формат регистра задачи (TR)

Формат TSS

Системное ПО должно создать как минимум один TSS для использования в длинном режиме и выполнить инструкцию LTR в 64-битном режиме, чтобы загрузить в TR 64-битный TSS.

TSS включает следующую информацию:

1. RSPn – Байты 0x1B-0x04. 64-битные адреса указателей стека (RSP) для уровней привилегий 0,1 и 2.

2. ISTn – Байты 0x5B-0x24. 64-битные адреса указателей стека, используемые механизмом IST.

3. Базовый адрес битовой карты разрешений ввода/вывода. Байты 0x67-0x66.

1.11 Исключения и прерывания

Исключения и прерывания приводят к передаче управления функциям операционной системы. Эти функции называются обработчиками исключений и прерываний. Как правило, прерывания обрабатываются незаметно для прерванного процесса. Перед передачей управления обработчику исключения/прерывания процессор сохраняет в его стеке rIP прерванной инструкции (адрес возврата). Обработчик прерывания/исключения должен сохранить контекст прерванного процесса, для возможности его последующего продолжения после завершения обработчика.

При возникновении прерывания или исключения процессор использует номер прерывания как индекс в IDT. IDT используется во всех режимах работы процессора.

Прерывания можно разделить на 3 категории [1]: исключения (англ. Exceptions), программные прерывания (англ. Software Interrupts) и внешние прерывания (англ. External Interrupts).

Исключения возникают в результате ошибок во время исполнения ПО или внутренних ошибок процессора. Исключения также могут возникать без наличия ошибочных ситуаций, например при пошаговом выполнении программы. Исключения считаются синхронными событиями, т.к. они возникают в результате исполнения прерванной инструкции.

Программные прерывания возникают в результате вызова прерывания (INT). В отличие от исключений и внешних прерываний, программные прерывания позволяют намеренно вызывать обработчики прерываний. Как и исключения, программные прерывания являются синхронными событиями.

Внешние прерывания генерируются системой в результате возникновения ошибки или какого-либо события вне процессора. Они доставляются на шину процессора используя внешние сигналы. Внешние прерывания являются асинхронными событиями, т.к. они возникают независимо от прерванной инструкции.

При обработке некоторых исключений процессор использует коды ошибок. Код ошибки помещается в стек перед вызовом обработчика исключения. Код ошибки может быть в двух форматах: в формате селектора и в формате страничного исключения [1].

1.11.1 Основные характеристики

Исключения и прерывания имеют несколько различных характеристик, которые зависят от того как они были доставлены и определяют как следует продолжить выполнение прерванного процесса.

Типы исключений

Исключения можно разделить на 3 типа:

1. Ошибки (англ. Faults) – возникают на границе перед инструкцией, вызвавшей исключение. Все изменения, вызванные данной инструкцией отменятся, поэтому инструкция может быть выполнена повторно. Например, страничное исключение (#PF).

2. Ловушки (англ. Traps) – возникают на границе после инструкции вызвавшей исключение. Процессор полностью выполнил инструкцию, вызвавшую исключение, т.е. все изменения, вызванные инструкцией сохранены. rIP указывает на инструкцию, следующую за той, которая вызвала исключение. Например, точка останова (#BP).

3. Аварии (англ. Aborts) – возникают на неопределенной границе, поэтому обычно они не позволяют продолжить корректное выполнение процесса. Например, двойная ошибка (#DF).

Маскирование внешних прерываний

Программное обеспечение может маскировать некоторые прерывания и исключения. Маскирование может задержать или не допустить запуск механизма обработки прерываний при возникновении прерывания. Внешние прерывания можно разделить на маскируемые и немаскируемые.

Маскируемые прерывания приводят к вызову обработчика прерывания только если rFLAGS.IF=1. В противном случае они не доставляются до тех пор, пока rFLAGS.IF не станет равным 0.

Немаскируемые прерывания (NMI) обрабатываются независимо от значения rFLAGS.IF. Однако при возникновении NMI прерывания, последующие NMI прерывания будут замаскированы до выполнения инструкции IRET.

1.11.2 Векторы прерываний

Определенным исключениям и прерываниям назначены фиксированные номера (также называемые векторами прерываний). Вектор прерывания используется механизмом обработки прерываний для определения точки входа в обработчик прерывания. Можно использовать до 256 векторов прерываний. Первые 32 вектора зарезервированы для предопределенных прерываний и исключительных ситуаций. Для программных прерываний можно использовать любые из свободных векторов.

Далее приведено кратное описание некоторых векторов прерываний. Подробное описание всех существующих прерываний можно найти в [1].

#DE Деление на 0 (Вектор 0). Возникает когда делитель в инструкциях DIV и IDIV равен 0. Так же возникает, если результат не помещается в регистры назначения.

Не возвращает код ошибки. Сохраненный rIP указывает на инструкцию, вызвавшую исключение.

#BP Точка останова (Вектор 3). Возникает при выполнении инструкции INT3. Не возвращает код ошибки.

#DF Двойная ошибка (Вектор 8). Может возникнуть, когда второе исключение возникло, во время обработки первого. Возвращает 0 в качестве кода ошибки.

#GP Нарушение защиты (Вектор 13). Возникает при нарушении защиты или неверном использовании функций AMD64. Возвращает код ошибки в формате селектора.

#PF Страничная ошибка (Вектор 14). Может возникнуть при доступе к памяти, в одной из следующих ситуаций:

1. Элемент таблицы страничного преобразования, задействованный в преобразовании адреса, отсутствует в физической памяти.

2. Попытка процессора выполнить инструкцию из неисполняемой страницы.

3. При доступе к памяти была нарушена одна из проверок (пользователь/супервизор, чтение/запись, или обе).

4. Зарезервированный бит в одном из элементов таблицы страничного преобразования установлен в 1.

Процессор сохраняет виртуальный адрес, вызвавший страничное исключение в регистре CR2. Формат кода ошибки показан на рис. 1.20.

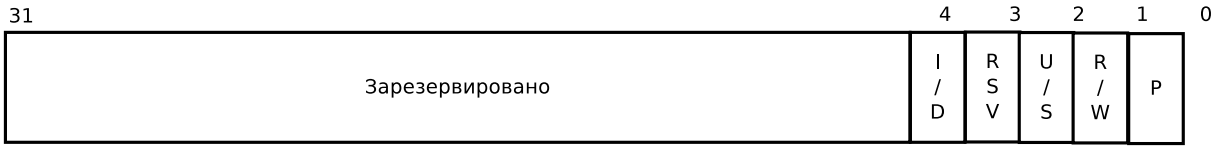

Рисунок 1.20 — Код ошибки при страничном исключении

Код ошибки в содержит следующую информацию:

1. Бит «P» – бит 0. Если этот бит сброшен в 0, значит страничное исключение было вызвано отсутствием страницы. Иначе – оно было вызвано нарушением одной из страничных проверок.

2. Бит «R/W» – бит 1. Если этот бит сброшен в 0 – исключение было сгенерировано при операции чтения. Иначе – исключение было сгенерировано при операции записи.

3. Бит «U/S» – бит 2. Если этот бит сброшен в 0 – ошибка была вызвана при доступе в режиме супервизора (CPL=0, 1, 2). В противном случае – ошибка была вызвана в режиме пользователя (CPL=3).

4. Бит «RSV» – бит 3. Если этот бит установлен в 1 – ошибка была вызвана в результате чтения процессором единицы (1) из зарезервированного поля в элементе таблицы страничного преобразования.

5. Бит «I/D» – бит 3. Если этот бит установлен в 1 – ошибка была сгенерирована при попытке извлечения инструкции. В противном случае этот бит сброшен в 0. Этот бит определен только если включена функция запрета исполнения (EFER.NXE=1 и CR4.PAE=1).

Прерывания определенные пользователем (Вектора 32-255). Возникают в следующих случаях:

1. Процессору поступает сигнал о наличии внешнего прерывания.

2. Программное обеспечение выполняет инструкцию INTn, в которой n определяет номер прерывания.

Не возвращают код ошибки. Значение rIP зависит от источника прерывания:

1. Обработчики внешних прерываний вызываются на границах инструкций. Сохраненный rIP указывает на прерванную инструкцию (не выполненную).

2. Если прерывание возникло в результате выполнения инструкции INTn, сохраненный rIP указывает на инструкцию, следующую за INTn.

Внешние прерывания можно замаскировать, установив rFLAGS.IF=0. Программные прерывания нельзя отключить.

1.11.3 Обработка прерываний в длинном режиме

Шлюзы прерываний и шлюзы ловушек

В длинном режиме передача управления обработчику прерывания/исключения осуществляется через дескрипторы шлюзов. В данном режиме IDT состоит из 256 16-байтных дескрипторов. Дескрипторы шлюзов делятся на 2 типа: дескрипторы шлюзов прерываний (англ. Interrupt Gates) и дескрипторы шлюзы ловушек (англ. Trap Gates).

Отличие между ними в том, что при переходе через шлюз прерывания процессор автоматически запрещает прерывания (сбрасывает RFLAGS.IF в 0).

Обработчики прерываний

При возникновении прерывания процессор умножает номер прерывания на 16 и использует результат в качестве смещения в IDT. Найденный дескриптор шлюза

содержит селектор сегмента и 64-битное смещение (виртуальный адрес обработчика прерывания). Селектор сегмента указывает на дескриптор сегмента кода расположенный в GDT или LDT. Дескриптор сегмента кода используется только для проверки доступа и перевода процессора в длинный режим.

На рис. 1.21 показано как происходит поиск обработчика прерывания в длинном режиме.

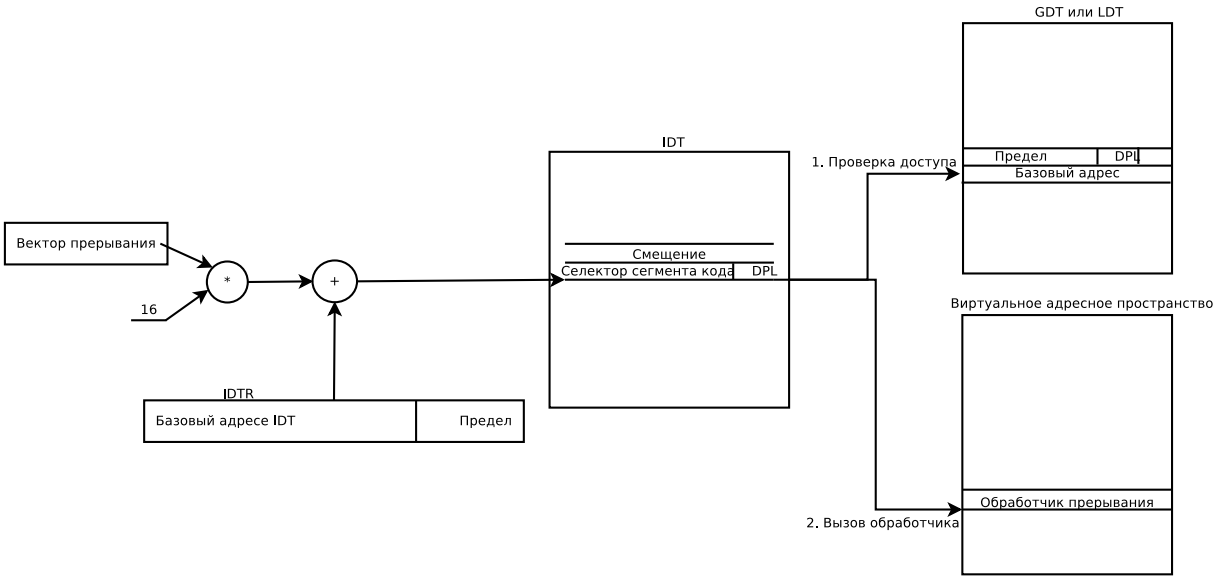

Рисунок 1.21 — Определение положения обработчика прерывания в длинном режиме

Стек обработчика прерывания

В длинном режиме указатель стека прерванной программы (SS:RSP) всегда заносится в стек обработчика прерывания, независимо от того был ли изменен уровень привилегий. Регистр SS не используется в 64-битном режиме, SS заносится для обеспечения возможности возврата из обработчика прерывания в режиме совместимости.

В длинном режиме при вызове обработчика прерывания процессор выполняет следующие действия [1]:

1. Выравнивает стек обработчика прерывания (выполняет побитовое «И» RSP с маской 0xFFFF FFFF FFFF FFF0)

2. Если поле IST в дескрипторе шлюза прерывания не равно 0, оно используется в качестве индекса в массиве IST TSS. Полученное из TSS значения заносится в RSP.

3. Если необходимо изменить уровень привилегий, уровень привилегий требуемого дескриптора (DPL) используется в качестве индекса в TSS для выбора нового указателя стека (если не используется IST), в регистр SS заносится 0.

4. Заносит в новый стек предыдущие значения SS:RSP. Значение SS дополняется 6 байтами, чтобы быть выровненным по 8-байтной границе.

5. Заносит 64-битное значение RFLAGS в новый стек. Старшие 32 бита имеют значение 0.

6. Сбрасывает флаги TF, NT, RF в регистре флагов RFLAGS в 0.

7. Если дескриптор шлюза описывает шлюз прерывания – процессор сбрасывает RFLAGS.IF в 0.

8. Заносит в стек CS:RIP прерванной программы. Значение CS дополняется 6 байтами, чтобы быть выровненным по 8-байтной границе.

9. Если данное прерывание возвращает код ошибки – процессор заносит этот код в стек. Код ошибки дополняется 4 байтами, чтобы быть выровненным по 8-байтной границе.

10. Загружает селектор сегмента из дескриптора шлюза в регистр CS. Процессор проверяет что целевой сегмент кода является 64-битным сегментом кода.

11. Загружает смещение из дескриптора шлюза в RIP.

На рис. 1.22 показано состояние стека после передачи управления обработчику прерывания.

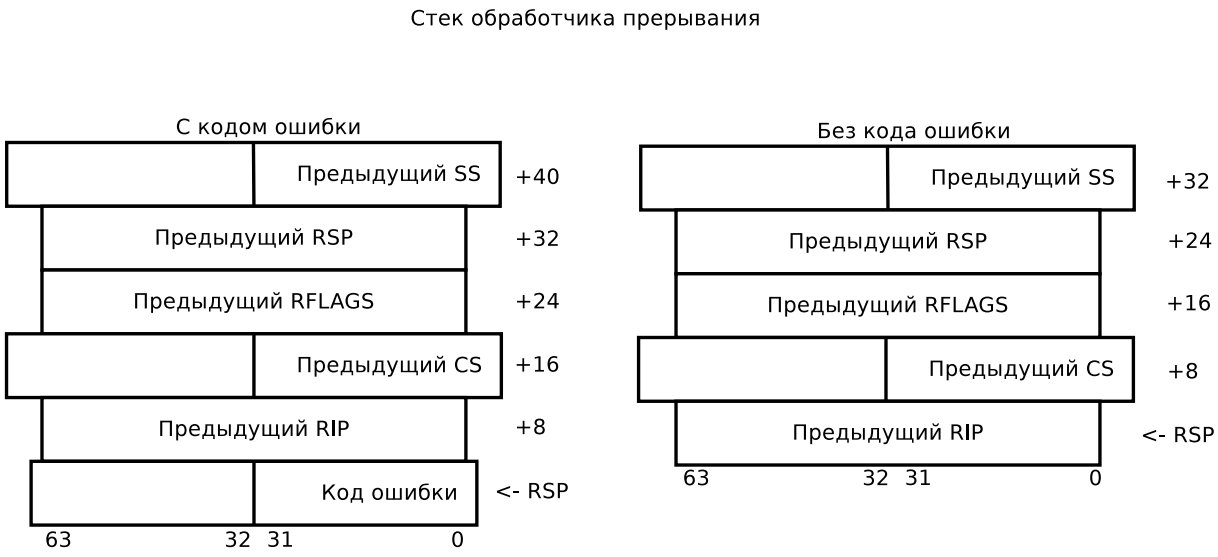

Рисунок 1.22 — Стек обработчика прерывания

В длинном режиме при переключении стека из-за смены уровня привилегий, новое значение для SS выбирается не из TSS (как в защищенном режиме). В длинном режиме из TSS берется только значение RSP, а в SS заносится ноль, позволяя тем самым обрабатывать вложенные прерывания. В SS.RPL заносится значение текущего уровня привилегий (CPL).

Стек обработчика прерывания при смене уровня привилегий, выглядит так же, как стек обработчика прерывания без смены уровня привилегий, меняется только регистр SS (в него заносится 0).

Таблица стеков обработчиков прерываний

В длинном режиме введен новый механизм переключения стеков – IST, который можно использовать в качестве альтернативы описанному выше механизму.

При использование данного механизма указатель стека переключается всегда. Данный механизм можно использовать отдельно для разных векторов прерываний, используя поле IST в дескрипторах шлюзов (в IDT).

На рис. 1.23 показано как используется данный механизм. Если при возникновении прерывания поле IST не равно 0, процессор использует значение IST в качестве индекса в TSS, загружая таким образом указатель стека (RSP) для обработчика прерывания. Если изменяется текущий уровень привилегий – в регистр SS загружается 0 и в SS.RPL заносится значение нового уровня привилегий. После того как стек загружен, процессор сохраняет в него предыдущий указатель стека, флаги процессора (RFLAGS), адрес возврата и код ошибки (если необходим). После этого управление передается обработчику прерывания. Если два обработчика прерывания используют один и тот же стек и второе прерывание возникнет во время обработки первого – оно затрет стек обработчика первого прерывания.

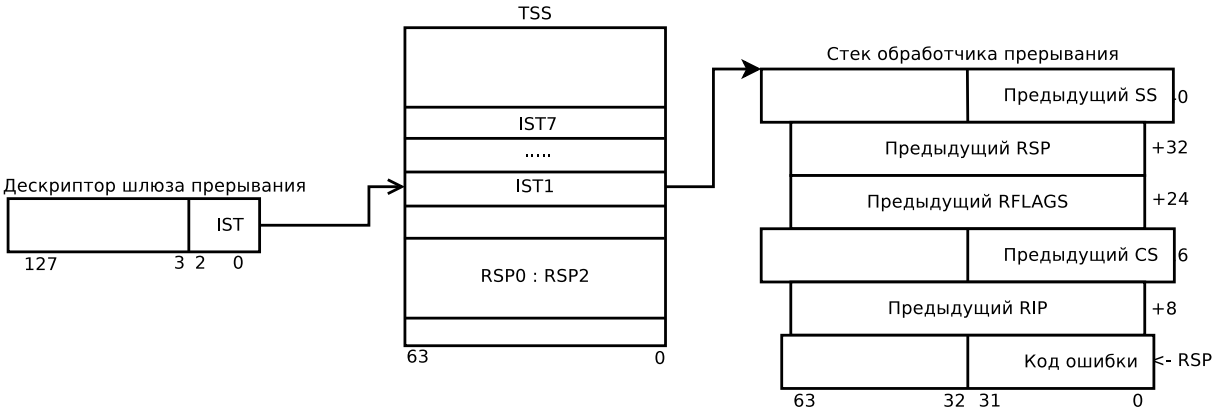

Рисунок 1.23 — Переключение стека с использованием IST

Возврат из обработчика прерывания

Для возврата в прерванную программу используется команда IRET. Механизм обработки прерываний всегда использует 64-битный стек при сохранении значений для обработчика прерываний, обработчик прерываний всегда выполняется в 64-битном режиме.

Если при вызове обработчика прерывания в длинном режиме происходит изменение уровня привилегий – в SS будет загружен нулевой селектор. Если обработчик прерывания будет прерван – нулевой селектор будет сохранен в стеке, а в SS будет повторно загружен другой нулевой селектор. Использование нулевого селектора таким способом позволяет процессору правильно обрабатывать вложенные вызовы обработчиков прерываний.

В длинном режиме, нулевой селектор в SS говорит о наличии вложенных обработчиков прерываний, поэтому в длинном режиме инструкция IRET при определенных условиях (обработчик находится на 0, 1 или 2 уровне привилегий в 64-битном режиме) может выталкивать из стека нулевой селектор в SS и не вызывать исключение #GP.

1.12 PIC, APIC и IOAPIC

Программируемый контроллер прерываний 8259 (PIC) использовался для доставки процессору прерываний от внешних устройств, что позволяло избежать трат процессорного времени на опрос устройств. В настоящее время считается устаревшим, современные архитектуры вместо него используют APIC и IOAPIC.

APIC осуществляет поддержку прерываний в архитектуре AMD64. Локальный APIC принимает прерывания и доставляет их процессору. На рис. 1.24 показана схема реализации APIC.

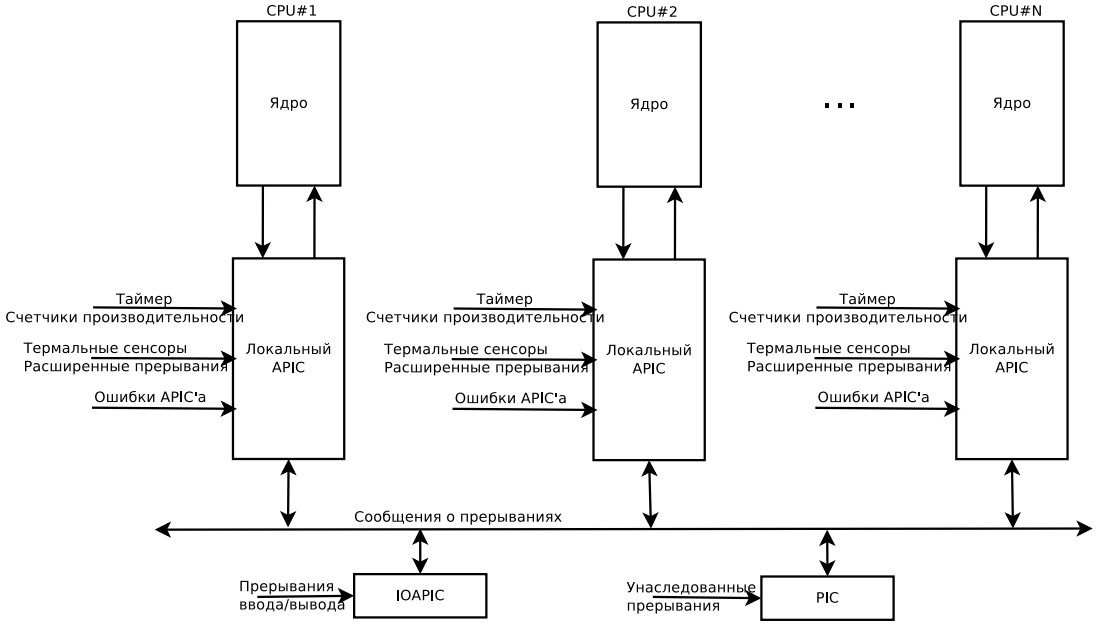

Рисунок 1.24 — Схема реализации APIC

1.12.1 Источники прерываний локального APIC

С каждым ядром ЦП связан локальный APIC, который принимает прерывания от следующих источников:

— Внешние прерывания от IOAPIC (включая LINT0 и LINT1)

— Унаследованные прерывания (INTR и NMI) от PIC

— Межпроцессорные прерывания (IPI) от других локальных APIC. Используются для отправки прерываний различным ядрам ЦП.

— Локальные прерывания. Локальный APIC получает прерывания от таймера, счетчиков производительности, термальных сенсоров, а также при возникновении ошибок APIC.

1.12.2 Локальный APIC

Локальный APIC контролируется битом «AE» регистра базового адреса APIC (APIC Base Address Register). Это моделезависимый регистр, который имеет номер 0x0000 001B (для доступа к регистру необходимо использовать команды RDMSR и WRMSR, с номером нужного регистра в регистре ECX). На рис. 1.25 показан формат регистра базового адреса APIC.

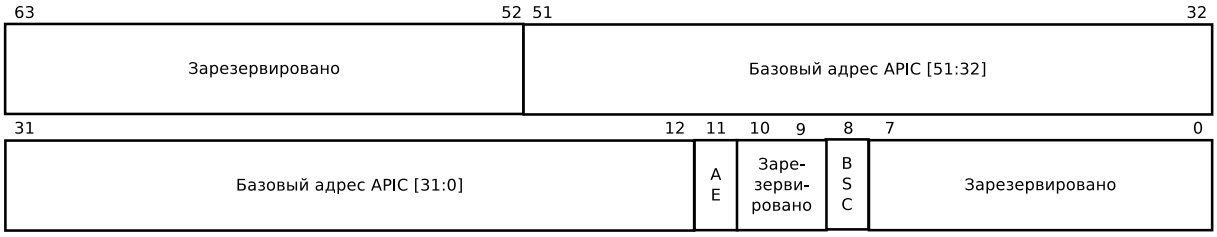

Рисунок 1.25 — Формат регистра базового адреса APIC

Регистр базового адреса APIC включает следующие поля:

— Бит «BSC». Бит 8. Показывает является ли текущее ядро загрузочным ядром загрузочного процессора.

— Бит «AE». Бит 11. Данный бит активирует APIC.

— Базовый адрес APIC. Биты 51:12. Задает базовый физический адрес регистров APIC (по умолчанию равен 0xFEE0 0000).

1.12.3 Регистры APIC

Настройка APIC производится с использованием отображенных в память регистров APIC. Адреса регистров вычисляются путем сложения базового адреса APIC

и смещения регистра. Базовый адрес APIC можно задавать через моделезависимый регистр базового адреса APIC.

Регистры APIC выровнены по 16-байтной границе. Доступ к ним должен осуществляться по адресам, выровненным по 4-байтной границе.

Подробное описание и назначение регистров APIC можно найти в [1].

Каждое ядро процессора в системе имеет уникальный идентификатор локального APIC (APIC ID). Его значение определяется аппаратным обеспечением в зависимости от идентификатора процессора и количества ядер. Как показано на рис. 1.26, идентификатор локального APIC находится в регистре APIC ID, в битах 31:24.

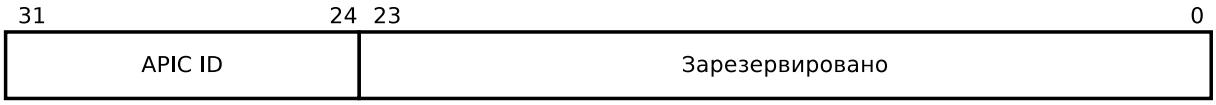

Рисунок 1.26 — Формат регистра APIC ID (смещение 0x20)

1.12.4 Таймер APIC

Таймер APIC это программируемый 32-битный счетчик, используемый ПО, для генерации событий. Таймер может работать в двух режимах – разовый и периодичный, в зависимости от значения бита «TM» (17) в регистре LVT (показан на рис. 1.27). Когда значение счетчика таймера достигает нуля – генерируется прерывание (если бит «M» в регистре LVT сброшен). В случае использования таймера в периодичном режиме, после генерации прерывания, счетчик таймера повторно инициализируется значением из регистра, содержащего начальное значение счетчика и отсчет начинается сначала.

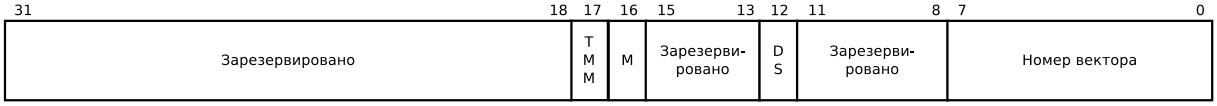

Рисунок 1.27 — Формат регистра APIC Timer LVT (смещение 0x320)

Для управления таймером определены 3 регистра: CCR (англ. Current Count Register), ICR (англ. Initial Count Register) и DCR (англ. Divide Configuration Register).

Регистр ICR имеет размер 32 бита и задает начальное значение счетчика таймера.

Регистр CCR имеет размер 32 бита, инициализируется значением регистра ICR и уменьшается на каждый тик таймера на значение, зависящее от регистра DCR. Когда значение данного регистра достигает нуля, генерируется прерывание.

Регистр DCR (рис. 1.28) имеет размер 32 бита и задает значение, которое вычитается из ICR на каждом тике таймера. Значение определяется в соответствии с таблицей 1.1.

Рисунок 1.28 — Формат регистра APIC Timer DCR (смещение 0x3E0)

Таблица 1.1 — Значения делителя, в зависимости от значения DCR

Биты 3, 1:0 регистра DCR	Значение делителя
000	2
001	4
010	8
011	16
100	32
101	64
110	128
111	1

1.12.5 IOAPIC

IOAPIC (I/O Advanced Programmable Interrupt Controller) используется для управления внешними прерываниями. Каждый IOAPIC может обрабатывать до 24-х типов прерываний.

Настройка IOAPIC производится с использованием трех отображенных на память регистров: IOAPICBASE, IOREGSEL и IOWIN. Для доступа к внутренним регистрам IOAPIC используется косвенная адресация через регистры IOREGSEL и IOWIN, расположение которых определяется значением регистра IOAPICBASE. Доступ ко всем регистрам осуществляется с использованием 32-битных операций чтения и записи, т.е. для модификация одного поля (бит, байт и т.д.) необходимо прочитать 32 бита, изменить нужное поле и записать обратно.

Подробное описание всех регистров IOAPIC приведено в [2].

Отображенные на память регистры IOAPIC

IOREGSEL. Регистр IOREGSEL используется для выбора внутреннего регистра IOAPIC для чтения/записи. После выбора требуемого регистра, данные можно

прочитать/записать используя регистр IOWIN. Формат регистра IOREGSEL показан на рис. 1.29. По умолчанию данный регистр расположен по адресу 0xFEC0 0000.

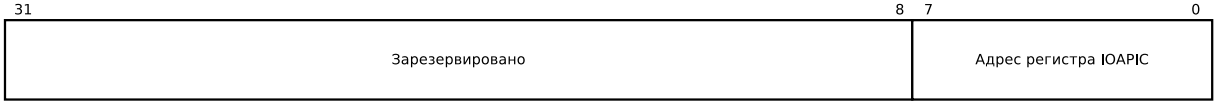

Рисунок 1.29 — Формат регистра IOREGSEL

IOWIN. Данный регистр имеет размер 32 бита и используется для чтения/записи во внутренний регистр IOAPIC, который был выбран с использованием регистра IOREGSEL. По умолчанию данный регистр расположен по адресу 0xFEC0 0010.

Внутренние регистры IOAPIC

IOAPICID. Как показано на рис. 1.30, данный регистр содержит идентификатор IOAPIC (биты 27:24), который используется в качестве физического имени IOAPIC. В данный регистр необходимо записать корректный идентификатор IOAPIC перед использование других регистров IOAPIC. Данный регистр имеет номер 0 (для доступа через IOREGSEL).

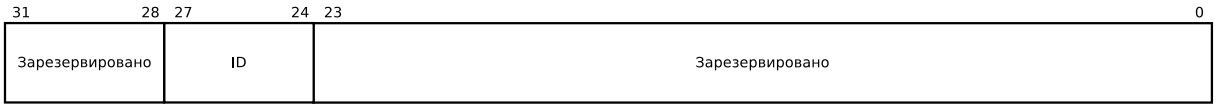

Рисунок 1.30 — Формат регистра IOAPICID

IOREDTBL[23:0]. Включает 24 64-битных элемента, для 24-х различных сигналов. В отличие от PIC – приоритеты прерываний не зависят от физического номера сигнала (они определяются номером обработчика прерывания). Для каждого сигнала ОС может задать полярность (высокая или низкая), режим возникновения прерывания (по фронту или по спаду) и некоторые другие свойства. Информация из регистра IOREDTBL используется для преобразования номера физического сигнала в сообщение для APIC. На рис. 1.31 показан формат элемента IOREDTBL. Так как элементы имеют размер 64-бита, каждый из них описывается двумя регистрами APIC, например для доступа к первому элементу используются регистры с номерами 0x10 и 0x11, ко второму – 0x12 и 0x13 и т.д.

Каждый элемент IOREDTBL включает следующие поля:

— ID получателя. Биты 63:56. Определяет получателя прерывания.

— Бит «IM» (16). Если установлен в 1 – прерывания замаскированы.

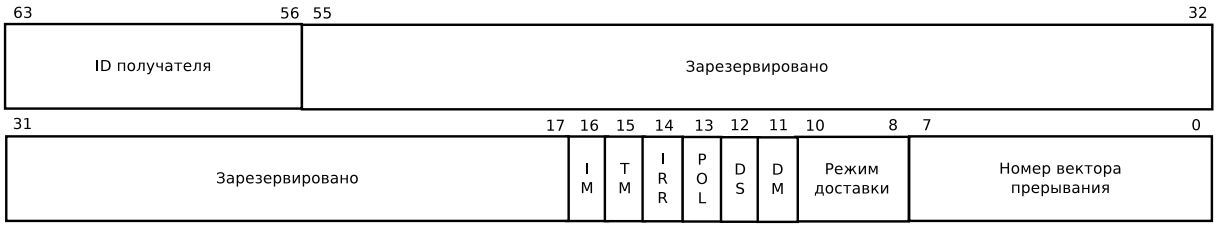

Рисунок 1.31 — Формат регистров IOREDTBL[23:0]

— Бит «TM» (15). Определяет режим срабатывания прерываний.

— Бит «IRR» (14). Устанавливается при получении прерывания.

— Бит «POL» (13). Определяет полярность сигнала прерывания.

— Бит «DS» (12). Содержит текущий статус доставки прерывания.

— Бит «DM» (11). Определяет как обрабатывать поле «ID получателя».

— Режим доставки. Биты 10:8. Существует несколько режимов доставки, в данной работе используется фиксированный режим (значение 0) – доставляет указанное прерывание всем ядрам процессоров, перечисленным в поле «ID получателя». Подробное описание всех режимов приведено в [2].

— Номер вектора. Биты 7:0. Содержит номер вектора прерывания, который необходимо вызвать. Допустимые значения лежат в диапазоне от 0x10 до 0xFE.

1.13 Инициализация процессора и переход в длинный режим

2 Организация исходных текстов Ann

При разработке операционных систем традиционно используется язык Си по ряду причин [3]:

— возможность прямого доступа к памяти, манипуляции битовыми структурами, свободное приведение типов;

— возможность собирать программы без подключения стандартной библиотеки;

— совпадение имён в программе и в исходном файле (отсутствие декорирования имен, 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_64: 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, Ann 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 Си.

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

1) **b** – byte, один байт;
2) **w** – word, два байта;
3) **l** – long, четыре байта;
4) **q** – quad, восемь 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, quad 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 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 — **somvar** 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 Ann используется диалект языка C, известный как GNU C. Подчеркнём, что в дальнейшем используется именно язык и компилятор Си, 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), то знания о ней нам почти не понадобятся.

Опишем две основные возможности диалекта GNU C.

Операция typeof. Исходные тексты Ann 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 Ann встречаются многочисленные inline-функции, в основном. Такие функции были введены в стандарте C99.

в диалекте GNU Си был добавлен составной оператор внутри выражения, причём возвращающий значение.

```
1 x = {int c = 2; c} + 1; // x == 3
```

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 ("....")**:

```
1 asm ("movl $0xfe, %eax \n"
2       "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(code
    : output variables
    : input variables
    : modified registers);
```

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.

2.4 Ограничение оптимизации обращений к переменным

Допустим, имеется следующий код.

```
1 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 значение её в теле цикла.

В языке C (стандарт C99) 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.

```
1 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 Си. Модификатор volatile укажет компилятору, что надо вставить ассемблерный код, несмотря на отсутствие видимых побочных эффектов:

```
1 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.

```
1 if (0) {
2     asm volatile("hlt"); // not reachable
3 }
```

2.5 Обзор исходных текстов Ann

Исходные тексты Ann размещены в каталоге **src**. Исходники Ann разделены по следующим каталогам:

— **src/kernel** – основные файлы ядра ОС;

— **src/kernel/boot** – файлы первого загрузчика ОС;

— **src/kernel/loader** – файлы второго загрузчика ОС;

— **src/kernel/lib** – служебная библиотека ядра (используется ядром и вторым загрузчиком);

— **src/kernel/misc** – вспомогательные заголовочные файлы;

— **src/kernel/interrupt** – файлы обработки прерываний;

— **src/stdlib** – стандартная библиотека (используется ядром и программами пользователя);

— **src/user** – файлы режима пользователя.

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.

Отладка с помощью GDB. Для отладки ОС можно воспользоваться стандартным отладчиком Unix-подобных систем – GDB. Чтобы немного упростить

жизнь, автор написал несколько команд для этого отладчика, они находятся в файле **src/.gdbinit**:

— **qemu** – подключиться к эмулятору QEMU;

— **debug-loader** – подготовить окружение к отладке загрузчиков (первого и второго);

— **debug-kernel** – подготовить окружение к отладке ядра.

Как этим пользоваться (все действия нужно выполнять в директории **src**):

1. Собираем ядро (например: **make lab1**);
2. Запускаем QEMU в режиме отладки (**make qemu-gdb**);
3. В новой вкладке терминала запускаем GDB (**gdb**);
4. Подключаемся к QEMU (**qemu**);
5. Подготавливаем окружение для отладки загрузчика (**debug-loader**);
6. Ставим точку останова (**break *0x7c00** или **break loader_main**);
7. Просим GDB выполнить инструкции до точки останова (**continue**);
8. Можете приступать к отладке.

Отладка ядра. Для отладки ядра последовательность действий нужно немного изменить, т.к. оно работает в длинном режиме, а загрузчики – в реальном и защищенном. GDB становится плохо при переходе из защищенного режима в длинный, поэтому необходимо прибегнуть к небольшой уловке (это актуально только для лабораторной №3 и далее), нужно сделать программную точку останова в ядре (например в начале функции **kernel_main()**):

```
1 bool debug = true;
2 while (debug == true) {
3     // wait, until someone change condition above
4 }
```

После этого выполнить следующие шаги:

1. Собираем ядро (например: **make lab3**);
2. Запускаем QEMU в режиме отладки (**make qemu-gdb**);
3. В новой вкладке терминала запускаем GDB (**gdb**);
4. Подключаемся к QEMU (**qemu**);
5. Просим GDB начать выполнять инструкции (**continue**);

6. Ждем несколько секунд и прерываем выполнение (<**CTRL**> + **C**);

7. Подготавливаем окружения для отладки ядра (**debug-kernel**);

8. Выходим из нашей программной точки останова: (**set variable debug = false**);

9. Можете приступать к отладке.

3 Начальная загрузка системы

В этой главе начинается работа над лабораторной работой №1

В этой работе мы соберем первую версию Ann, затем изучим процесс начальной инициализации от включения компьютера до начала работы ядра системы, используя эмулятор QEMU и отладчик GDB.

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 несколько раз 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 — все дальнейшие команды мы будем выполнять в ней. Перейдем в каталог, выбранный как рабочий, и затем выполним следующие команды для клонирования репозитория исходных текстов Ann, использующего систему контроля версий Git.

```
1 git clone https://gitlab.com/OperatingSystemMasters/AnnOSTemp
```

Теперь у нас есть каталог **AnnOSTemp**, содержащий локальный репозиторий исходных текстов Ann. 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 **AnnOSTemp** или любом его подкаталоге.

3.1 Загрузчик ядра операционной системы

Так уж сложилось что нас ОС – 64-битная, а процесс перехода в длинный режим не совсем тривиальный, это было одной из причин почему автор решил сделать двухэтапную загрузку.

Исходные тексты первого загрузчика Ann находятся в файлах **src/kernel/boot/boot.S** и **src/kernel/boot/main.c**. По соглашениям BIOS

загрузчик будет загружен из первого сектора устройства по адресу 0x7C00, а его размер не может превышать 512 байт.

Используемый в нашей системе загрузчик предполагает, что начиная со второго сектора на диске находится второй загрузчик ОС – в первом секторе диска находится сам загрузчик. Для передачи управления второму загрузчику наш первый загрузчик должен выполнить следующие шаги:

а) Сохранить карту областей физической памяти (memory map), чтобы ей мог воспользоваться второй загрузчик.

б) Переключить процессор в 32-битный защищённый режим и включить доступ к памяти свыше 1 Мб.

в) Считать заголовок второго загрузчика с диска в некоторую область памяти путём прямого программирования контроллера IDE (наша система умеет работать только с IDE-дисками).

г) Извлечь из заголовка ELF-файла второго загрузчика информацию о его секциях и адрес точки входа.

д) Считать необходимые секции второго загрузчика с диска в ОЗУ.

е) Передать управление на адрес, рассчитанный на основе адреса точки входа из заголовка ELF-файла.

3.2 Начальная загрузка компьютера

Загрузка системы начинается с того, что после включения компьютера управление передается на реальный адрес 0xFFFF FFF0 [1], где находится ПЗУ с кодом BIOS [3].

После того, как BIOS проинициализирует устройства, вектора прерываний реального режима и выберет загрузочный дисковод или жесткий диск, он загружает его первый сектор (512 байт) по адресу 0x7c00, после чего передает управление на него (реальный адрес 0000:7c00). Два последних байта сектора должны иметь значения 0x55 и 0xAA, иначе BIOS не сочтет сектор загрузочным. Таким образом, размер начальной части загрузчика ОС фактически не может превышать 510 байт.

3.3 Инициализация защищенного режима

Перед активацией длинного режима необходимо перейти в защищенный режим. Для этого необходимо загрузить GDT, содержащую дескриптор сегмента кода и дескриптор сегмента данных, доступный для чтения/записи.

После того, как новая GDT загружена, переход в защищенный режим можно выполнить установив CR0.PE в 1.

3.4 Инициализация длинного режима

В защищенном режиме системное ПО может подготовить структуры данных, необходимые для перехода в длинный режим и сохранить их в произвольном месте в пределах первых 4-х гигабайт физической памяти. Эти структуры данных можно будет переместить за пределы 4-х гигабайт после перехода в длинный режим. Для перехода в длинный режим необходимы следующие структуры данных:

— 4 уровня таблиц страниц. Для перехода в длинный режим, также необходимо активировать PAE.

— GDT, содержащая дескрипторы сегментов для ПО работающего в 64-битном режиме и в режиме совместимости, включающая:

1. Дескриптор сегмента кода для длинного режима (должен быть установлен бит «L»).
2. Дескриптор сегмента данных для ПО, работающего в режиме совместимости.

Существующая GDT защищенного режима может быть использована для хранения перечисленных выше дескрипторов.

3.5 Активация и переход в длинный режим

Для активации длинного режима необходимо установить бит EFER.LME в 1. Однако переход в длинный режим не будет выполнен до активации страничного преобразования. После того, как системное ПО активирует страничное преобразование, при активированном длинном режиме, процессор перейдет в длинный режим, установив при этом бит EFER.LMA в 1.

Для выбора подрежима работы процессора в длинном режиме используются 2 бита дескриптора сегмента кода (CS.L и CS.D). Комбинация CS.L=1, CS.D=0 – переводит процессор в 64-битный режим. Комбинация CS.L=0, CS.D=0 – в режим совместимости.

3.6 Переход в длинный режим

Для перехода в защищенный режим, системное ПО должно выполнить следующие действия:

1. В любой последовательности:

— Активировать механизм расширения физических адресов, установив бит CR4.PAE в 1. Это необходимо сделать до активации страничного преобразования.

— Загрузить в регистр CR3 физический адрес PML4.

— Активировать длинный режим, установив бит EFER.LME в 1.

2. Активировать страничное преобразование, установив бит CR0.PG в 1. В результате процессор установит бит EFER.LMA в 1.

3.7 Обновление ссылок на таблицы системных дескрипторов

После перехода в длинный режим регистр GDTR ссылается на GDT унаследованного режима, которая расположена в пределах первых 4-х гигабайт виртуального адресного пространства. Системному ПО необходимо обновить GDTR, чтобы он указывал на 64-битную GDT, используя команду LGDT.

3.8 Формат файла ядра

Собранное ядро находится будет находиться в файле **src/kernel/kernel** и представлять 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 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 заголовке. Например, следующая команда покажет основную информацию о содержимом файла ядра Ann.

```
1 readelf -hS src/kernel/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 **kernel/misc/elf.h**. Для компоновки второго загрузчика и ядра используется сценарий компоновщи-

ка: **src/kernel/loader/linker.ld** и **src/kernel/linker.ld** соответственно. В них, в частности, указаны базовые адреса компоновки.

3.9 Сборка и запуск первой лабораторной работы

Для сборки исходников Ann в бинарный файл с образом диска нужно перейти в каталог **ann/src** и выполнить команду **make**. После успешной сборки образа системы командой **make** можно приступать к её запуску в эмуляторе QEMU. Этот процесс уже был описан в главе 2.6. Но я продублирую его тут ещё раз, попробуем пошагово пройтись по инициализации и рассмотреть некоторые моменты.

Если вы не знакомы с командами gdb, вот небольшая подборка (за более подробной информацией обращайтесь к документации gdb):

— **n** выполнить текущую строку кода, не заходя в функцию (если на данной строке есть вызов функции);

— **s** выполнить текущую строку кода, с заходом в функцию (если на данной строке есть вызов функции);

— **si** выполнить текущую машинную инструкцию, с заходом в функцию (если текущая инструкция – вызов функции);

— **ni** выполнить текущую машинную инструкцию, без захода в функцию (если текущая инструкция – вызов функции);

— **b <expr>** установить точку останова, **<expr>** может быть номером строки, адресом или функцией;

— **c** продолжить выполнение до первой встреченной точки останова;

— **info registers** показать содержимое регистров;

— **p <expr>** вывести результат **<expr>**, например **p $eax** покажет содержимое регистра **eax**;

Собираем ядро. Идем в директорию **ann/src**. Выполняем команду **make lab1**;

Запускаем QEMU в режиме отладки. Выполняем команду **make qemu-gdb**;

Запускаем GDB. Открываем новую вкладку терминала. Идем в директорию **ann/src**. Выполняем команду **gdb**;

Подключаемся к QEMU. Выполняем команду **qemu**;

Подготавливаем окружение для отладки загрузчика. Выполняем команду **debug-loader**;

Ставим точку останова. Выполняем команду **break *0x7c00**;

Просим GDB выполнить инструкции до точки останова. Выполняем команду **continue**;

Пошаговое исполнение. Отлично, теперь можно немного подебажить. С помощью команды **si** и **ni** можно пошагово пройтись по коду, посмотреть как меняется содержимое регистров. Однако есть одна проблема: из-за того, что загрузчик собран с опцией **-Os** (если верить интернетам), команда **ni** работает так же как и **si**. Поэтому gdb будет заходить во всех функции, в т.ч. и в прерывания.

Чтобы вас сильно не травмировать можно сразу поставить точку останова на инструкцию, когда все прерывания уже были вызваны и продолжить пошаговое выполнение оттуда, для этого выполняем следующие команды: **b memory_detected** и **c**.

После инструкции **call bootmain** вы попадете в сишную часть первого загрузчика, можете попробовать выполнить его пошагово, но поскольку он собран с **-Os** поведение отладчика может вас немного удивить (вобще заниматься отладкой оптимизированного кода через gdb – дело неблагодарное).

В конечном счете вы сможете добраться до перехода во второй отладчик **((void (*)(void))(elf_header->e_entry))();**. Зайдите внутрь это команды (команда **s**) и вы окажетесь в коде второго отладчика, начиная с этого момента вы уже можете наслаждаться работой gdb, т.к. тут он себя будет вести предсказуемо и все будет работать.

3.10 Задание №1

Пока что вам ничего дописывать не нужно, но надо хорошо освоиться в исходниках, скриптах компоновщика и отладчике, а чтобы вам помочь, вот несколько вопросов:

1. Посмотрите документацию к **BIOS INT 15h, AX=E820h**, расскажите, что за магия происходит внутри **detect_high_memory**?

2. Зачем надо включать линию A20?

3. Какой командой загрузчик переключает процессор в защищённый режим?

4. Как первый загрузчик инициализирует регистр указателя стека? Где находится стек при работе загрузчика?

5. Как устроена GDT загрузчика?

6. Начиная с какого физического адреса загружается первый загрузчик?

7. Начиная с какого физического адреса загружается второй загрузчик? А с какого виртуального?

8. Как загрузчик выделяет память для чтения второго загрузчика с диска?

9. Какая последняя исполняемая команда первого загрузчика (особо любознательным может помочь **layout asm**)?

10. Какая первая исполняемая команда второго загрузчика?

4 Страничное управление памятью

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

Для перехода на использования 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.

Включением страничной адресации будет заниматься второй загрузчик, который необходимо будет дописать. Ядро ОС и второй загрузчик используют общий код: управление памятью, вывод на экран, работа с диском. Ядро является 64-битным, а загрузчик – 32-битным.

4.1 Лабораторная работа №2

Собираем ядро. Идем в директорию **ann/src**. Выполняем команду **make lab2**;

Запускаем QEMU в режиме отладки. Выполняем команду **make qemu-gdb**;

Запускаем GDB. Открываем новую вкладку терминала. Идем в директорию **ann/src**. Выполняем команду **gdb**;

Подключаемся к QEMU. Выполняем команду **qemu**;

Подготавливаем окружение для отладки загрузчика. Выполняем команду **debug-loader**;

Ставим точку останова. Выполняем команду **debug-loader-b**. В этом макросе 4 точки остановки. Когда освоитесь с первыми двумя, их можно будет удалить из файл .gdbinit.

4.2 Организация виртуального адресного пространства

В операционных системах виртуальное линейное адресное пространство обычно разделено на две части см. рис. 1.17. 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.

Основные моменты выбранного отображения виртуальных адресов:

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

2. Младшие адреса до адреса **USER_TOP** включительно содержат код, данные и стек программы пользователя;

3. Сначала в области ядра идет стек ядра, растущий вниз, а затем – таблица страниц текущего пространства;

4. Начиная с адреса **KERNEL_BASE** расположена область, отображения в первые 32ГБ физической памяти. В ней находится код и данные ядра, за исключением расположенных на стеке локальных переменных

Также необходимо обязательно обратить внимание на макросы определенные в файле **kernel/lib/memory/layout.h**

— **VADDR** – вычисление виртуального адреса;

— **PADDR** – вычисление физического адреса памяти;

Данные макросы помогают находить виртуальный или физический адрес. Однако есть хитрость в их работе – все зависит от того, где используются данные макросы: в загрузчике или уже в самом ядре. Внимательно изучите их,

особенно значение переменной **VADDR_BASE**. Посмотрите на ключи в файле /**src**/**kernel**/**loader**/**Makefile.am** и в файле /**src**/**kernel**/**Makefile.am**.

В файле **kernel**/**lib**/**memory**/**mmu.h**

— **PML4E_ADDR** –

— **PDPE_ADDR** –

— **PDE_ADDR** –

— **PTE_ADDR** –

— **PAGE_ADDR** – вычисление линейного адреса из индексов и смещения, не забываем про каноничную форму.

4.3 Загрузка 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 перестает выделять память под собственные нужды: дальнейшее выделение физической памяти происходит только для запуска и работы программ пользователя, включая создание новых таблиц страниц. Этими действиями занимается функция **loader_main** расположенная в файле /**kernel**/**loader**/**loader.c**. Последовательность действий:

1. Сохраняем указатель на массив дескрипторов областей памяти;
2. Сохраняем количество элементов в этом массиве;
3. Инициализация терминала;
4. Считываем заголовки ядра из elf файла;
5. Определить количество доступной памяти;
6. Инициализация доступной памяти;
7. Перейти в **длинный режим** (**long mode**) работы процессора.

4.4 Выделение памяти. Задание №2

Дописать функцию **loader_alloc**. Функция позволяет выделить память необходимым размером. Она находится в файле /**kernel**/**loader**/**loader.c**. Для хранения адресов можно использовать тип uint32_t, т.к. наш загрузчик все еще 32 битный и на 64-битную адресацию мы еще не перешли. Для округления адресов и выравнивания используйте макрос ROUND_UP.

4.5 Загрузка ядра с диска. Задание №3

Дописать функцию **loader_read_kernel**. Чтобы загрузить ядро, необходимо определить точку входа и прочитать заголовки elf файла. Память для elf_header выделяется с мощью функции **loader_alloc**. При чтение заголовков из elf файла, важно помнить, что необходимо правильно отображать виртуальные адреса в физические. Для этого старшие байты виртуального адреса надо отбросить, чтобы поддерживался следующий мэппинг памяти **[KERNBASE; KERNBASE+FREEMEM) -> [0; FREEMEM)**. Незабываем обновить значение **kernel_entry_point**.

— **struct bios_mmap_entry** - дескриптор области памяти;

— **mm** - массив дескрипторов;

— **cnt** - количество элементов в этом массиве.

4.6 Подготовка памяти. Задание №4

Дописать функцию **loader_detect_memory**. Дескриптор области памяти содержит тип памяти, базовый адрес, длину. Необходимо проверять тип памяти на доступность, поверить, чтобы сумма базового адреса и длины области была не больше **max_physical_address** (Надо разобраться, почему!). **max_physical_address** хранится максимальное количество доступной памяти.

Количество страниц можно узнать поделив **max_physical_address** на размер одной страницы. Незабываем выравнивать память макросами **ROUND_UP** или **ROUND_DOWN**. (экспериментируйте)

Страница памяти представляет собой простую структуру данных, содержащую указать на счетчик обращения к ней и указать на список свободных страниц. 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.

```
1 struct page{
2 uint32_t ref;
3 LIST_ENTRY(page) link;
4 };
```

— **page_init** - инициализирует список дескрипторов страниц памяти, которые могут быть использованы. Такие страницы отмечаются в качестве свободных, для чего счётчик ссылок (поле ref) в их дескрипторе устанавливается равным нулю;

— **page_alloc** - выделяет страницу памяти, удаляя её дескриптор из списка свободных страниц (эта функция не должна изменять счётчик ссылок на эту страницу,т.к. сама она не добавляет какие-либо ссылки на неё в таблицы страниц);

— **page_free** - возвращает указанный дескриптор страницы в список свободных страниц;

— **pa2page** - преобразования физического адреса в адрес дескриптора его страницы;

— **page2pa** - преобразования виртуального адреса в физический адрес;

— **page_insert** - отображает виртуальный адрес на заданный физический;

— **page_lookup** - поиск дескриптора физической страницы, отображаенной по данному виртуальному адресу;

— **page_remove** - для уничтожения отображения виртуальной страницы в физическую.

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. Переход будет осуществляться функцией **loader_init_memory**и **loader_enter_long_mode**.

У функции есть несколько этапов работы. Первая часть выделяет и заполняет память для управляющих структур, таких как:**struct mmap_state state**, **struct kernel_config *config**, **pml4 (page map level 4)** и т.п. Загружает новую таблицу GDT, теперь нужно 5 дескрипторов: нулевой, два дескриптора для ядра и два дескриптора для пользователя. Смотрим функцию **loader_init_gdt**. Обновляем поля в структуре **config**, обновляем поля в структуре **state**.

Одно из самых важных действий функции - создание список свободных страниц. Если страница памяти уже используется ядром или видеопамятью, то обновляем счетчик ссылок на нее **ref**.

Отображением памяти занимается функция **loader_map_section**. Она 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. Отметим, что сама эта функция не выделяет физическую память каким-либо образом — она понадобится для

отображения кода ядра, его стека и статических данных. Разберитесь, что происходит в коде этой функции. Есть несколько секций для которых это надо сделать:

— Стек ядра

— Информация для ядра

— APIC

— IOAPIC

— Отображаем сам загрузчик

— Отображаем ядро

Функция вернет управление **loader_main**. Останется только вызвать функцию **loader_enter_long_mode** передав в нее точку входу ядра. Вам необходимо будет разобраться самим, что делает данная функция. В помощь код из файла **boot.S**.

4.8 Окончательная загрузка ядра. Лабораторная работа №3

Важное замечание. Для дебага лабораторной работы №3 и дальше надо использовать иниструкцию по отладке ядра с помощью gdb.

Для работы ядра необходимы структуры данных, подготовленные вторым загрзучиком: адрес GDT, PML4 и массив дескрипторов страниц. Загрузчик создал отображение в виртуальном адресном пространстве ядра для доступа в этим структурам (`KERNEL_INFO`).

Содержащиеся в структуре `KERNEL_INFO` адреса находятся в виртуальном адресном пространстве загрузчика и совпадают с физическими (за счет использования загрузчиком плоской модели памяти).

Для их преобразования в корректные адреса в виртуальном адресном пространстве ядра достаточно прибавить к ним значение `KERNEL_BASE` (благодаря наличию отображения, выполненного загрузчиком).

После перехода на плоскую модель памяти и включения страничного преобразования, ядро может начать работать в нужном нам режиме. Оставшиеся действия:

1. Сконвертировать физические адреса в виртуальные в структуре **kernel_config**;

2. Заново инициализировать структуру **state**;

3. Сконверитровать 32-битные дескрипторы в 64-битные.

4.9 Контрольные вопросы

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

2. Зачем нужно использовать макрос **ROUND_UP**?

3. Для чего используется список свободных страниц и счетчик ссылок на страницу?

4. Какой размер, в нашем случае, имеет страница памяти?

5. Что описывает экземпляр структуры **page**?

6. Сколько памяти может отобразить одна страница?

7. Как работает страничное преобразование?

8. Строение дескриптора таблицы страницы в длином режиме?

9. Что такое и как работает TLB?

10. Почему адреса таблиц страниц выравниваются на размер страницыи и нужно ли это делать?

11. Какие сегменты памяти необходимо проверять, чтобы узнать свободна ли страница памяти или нет?

12. Какие дескрипторы имеются в GDT, загруженные после подготовки страничного преобразования?

13. Режим legacy mode и long mode, зачем они нужны?

14. Как работают макросы **VADDR** и **PADDR**, почему у значения **VADDR_BASE** разное значение?

15. Зачем надо заново инициализировать структуру **config**?

16. Зачем надо заново инициализировать структуру **state**?

17. Почему ОС может работать дальше, несмотря на то, что мы поменяли систему управления памяти?

5 Ядро ОС

5.1 Управление процессами

Одной из основных задач ядра операционной системой является управление процессами.

Для реализации многозадачности с использованием одного ядра ЦП необходимо иметь возможность приостанавливать и продолжать процессы. Для этого надо хранить контекст процесса (значения регистров и адрес PML4). Структура контекста процесса была выбрана таким образом, чтобы она формировалась естественным образом при возникновении прерываний. Формат структуры контекста процесса показан на рис. 5.1.

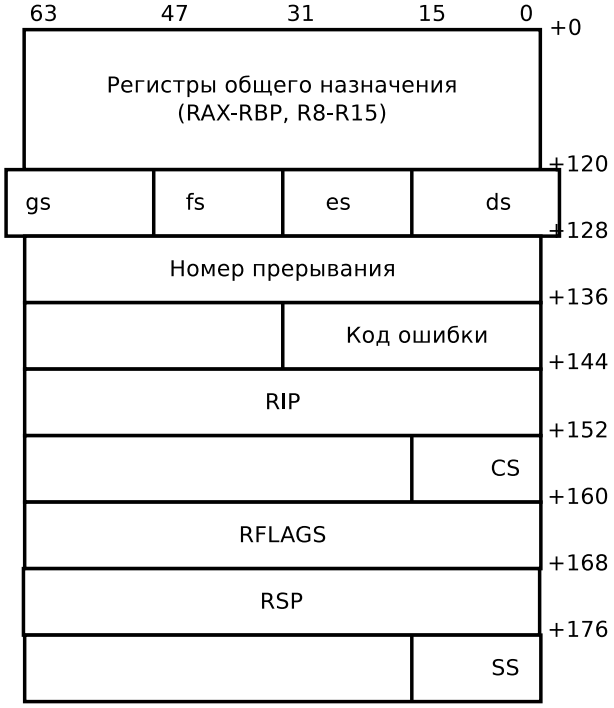

Рисунок 5.1 — Контекст процесса

Однако, для управления процессами одного контекста процесса недостаточно, поэтому каждый процесс имеет дескриптор процесса, который включает: контекст процесса, имя и идентификатор процесса, текущее состояние, виртуальный и физический адрес PML4. Формат дескриптора процесса показан на рис. 5.2.

Согласно [4], процесс может находиться в трех состояниях:

— `выполняемый` – в данный момент используется центральный процессор;

— `готовый` – работоспособный, но временно приостановленный, чтобы дать возможность выполняться другому процессу;

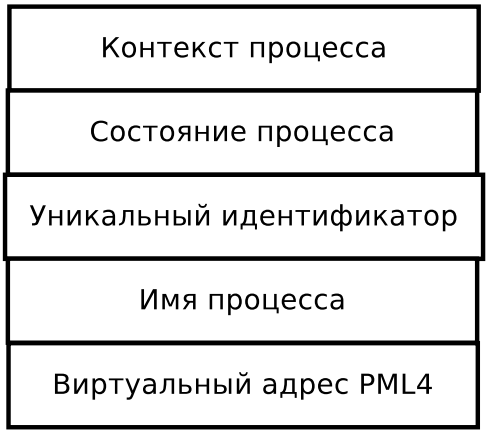

Рисунок 5.2 — Дескриптор процесса

— `заблокированный` – неспособный выполняться, пока не возникнет какое-либо внешнее событие.

В разрабатываемой ОС состоянию `выполняемый` соответствует состояние `RUN`, состоянию `готовый` – `READY`, состоянию `заблокированный` – `DONT_RUN`.

Также было введено дополнительное состояние `FREE`, которое означает что дескриптор свободен и может быть использован для создания нового процесса.

На рис. 5.3 представлена диаграмма состояний процесса.

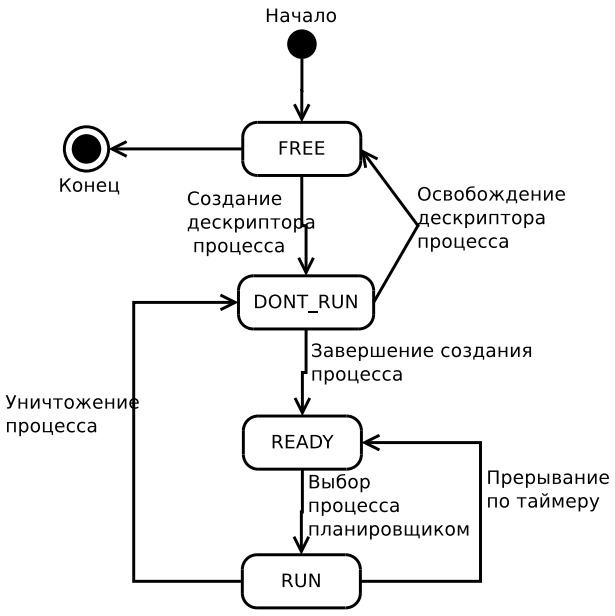

Рисунок 5.3 — Диаграмма состояний процесса

6 Прерывания и исключения

6.1 Лабораторная работа №4

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 вызовоми в данной главе мы добавим его в нашу систему, для чего сначала нам нужно реализовать обработку прерываний ядром.

Обработка программных исключений и аппаратных прерываний нужна 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;

— реализация системного вызова.

6.1.1 Прерывания и исключения в Ann

В Ann исключения и прерывания обрабатываются в режиме ядра, на нулевом уровне привилегий. Для упрощения системы, прерывания будут запрещены в режиме ядра, а исключения в режиме ядра будут рассматриваться как фатальная ошибка операционной системы.

Для обработки прерываний и исключений используется IDT. Ядру необходимо создать обработчики прерываний, исключений, и загрузить IDT, содержащую ссылки на эти обработчики.

Надо реализовать базовую обработку исключений и системных вызовов. Для этого необходимо написать обработчики исключений с номерами от 0 до 31, некоторые из исключений в этом диапазоне 0-31 Intel определяет как зарезервированные и не используемые в настоящий момент.

Прерывания Ядро должно инициализировать TSS (установить указатель стека для нулевого уровня привилегий, добавить указатели стеков в массив IST для пре-

рываний и исключений), выделить память под стеки обработчиков прерываний и исключений. После чего загрузить в регистр TR дескриптор TSS.

Для описания обработчиков прерываний и исключений используются шлюзы дескрипторов прерываний. Это позволяет упростить обработчики прерываний и исключений, т.к. при этом они могут быть нереентерабельными и использовать глобальные переменные.

Обработчики аппаратных прерываний используют механизм IST для переключения стека, для этого в дескрипторе шлюза прерывания в поле IST заносится индекс IST в TSS. Это позволяет разрешить прерывания в потоках ядра, используя небольшой стек в 1 страницу памяти.

Поскольку в момент прерывания ни в регистрах, ни на стеке процессором не сохранен 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 **/kernel/interrupt/interrupt_entry.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 Ann используется структура **task_context**, находится в файле **kernel/task.h**. Заполнив эту структуру прямо «на стеке», мы без всяких дополнительных усилий включим в неё сохраненные 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.

Изучите структуру **task_context** рис. 5.1. Надо заметить, что единственным полем, зависящим от обработчика, является номер прерывания. Таким образом, можно выделить общую для всех обработчиков часть, которая находится после метки **interrupt_handler_common**. Эта часть будет завершаться вызовом функции **interrupt_handler**. Часть, специфичная для обработчика, будет генерироваться специальными макросами – **interrupt_handler_no_error_code** и **interrupt_handler_with_error_code**. По названию макросов можно сделать вывод, что макросы различаются еще возможностью добавлять код ошибки в структуру.

6.1.2 Задание №5

В файле **kernel/interrupt/interrupt.c** необходимо дописать точки входа для прерываний и системных вызовов. Имена названий функций можно посмотреть в массиве **interrupt_name**. Также посмотрите файл **kernel/interrupt/interrupt.h**

Каждому прерыванию соответствует свой номер. Можете написать сразу все функции или только для диапазона прерываний 0-30. Обработка системных прерываний будет рассматриваться в лабораторной работе №5

6.1.3 Задание №6

Нужно дописать код после метки **interrupt_handler_common** в файле **kernel/interrupt/interrupt_entry.S**. Смотрим на рис. 5.1. Идея в том, что мы помещаем структуру **task_context** на верх стека, т.е. надо запушить поля из данной структуры в стек. Обратите внимание что регистры ds, es, fs, gs 16-битные и все 4 могут уместиться в одном дескрипторе.

После того, как все поля структуры добавлены в стек, необходимо обновить регистры ds, es т.к. мы переключаемся на новый стек. Перед этим не забудьте сохранить регистр `rax`. В конце вызываем функцию **interrupt_handler**.

6.1.4 Задание №7

После того, как дописан общий код для обработки прерываний, надо дописать точки входа для прерываний и ошибок в файле **kernel/interrupt/interrupt_entry.S**. Прерывания могут обрабатываться поразному: с передачей кода ошибки или без передачи кода ошибки. Чтобы узнать, какой макрос надо использовать надо прочитать мануал по процессорам Intel [5] глава 9.8

6.1.5 Задание №8

В файле **kernel/interrupt/interrupt.c** необходимо дописать функцию **interrupt_handler**. На эту функцию передается управление после выполнения общего обработчика из предыдущего задания. Структура (**task_context**) находится на стеке и поэтому мы можем легко к ней обращаться из функции. Необходимо дописать **switch** условие.

Для прерывания **INTERRUPT_VECTOR_BREAKPOINT** надо будет возвращать функцию **schedule()**(сейчас есть только прототип этой функции, но это не важно). А прерывание **INTERRUPT_VECTOR_PAGE_FAULT** уже написано. Сейчас нужно дописать условия для 17 оставшихся зарезервированных прерываний из диапазона 0-30. И мы считаем, что все эти прерывания являются необработанными и система должна выдать предупреждение об этом.

6.1.6 Задание №9

Надо инициализировать массив **idt** в функции **interrupt_init** значениями. Элементом в массиве является **descriptor64**. Номера элементов, которые надо инициализировать надо взять из файла **kernel/interrupt/interrupt.h**. В 4 лабораторной работе, создайте массив только с индексами от 0 до 30. Используйте макрос **INTERRUPT_GATE**.

6.1.7 Задание №10

И последнее задание, необходимо выделить память под стек прерываний и под стек ошибок. Для выделения памяти воспользуйтесь функцией **page_alloc**. Для выделенную странице памяти следуют обновить счетчик ссылок на нее, в этом поможет функция **page_insert**. Обязательно проверяйте выделилась ли память.

Чтобы протестировать написанный код, после инициализации прерываний функция **interrupt_init**, вызовите прерывание командой `int`. Можно использовать любой другой номер прерывания. Посмотрите правильно ли передается код, обработчику прерываний. Надо выделить количество памяти под размер стека. Его можно посмотреть в файле **kernel/leb/memory/layout.h**

```
1  asm volatile ("int 0x14\n");
```

6.2 Контрольные вопросы

1. В каких состояниях может находится процесс?

2. С какой целью реализуются прерывания и исключения?

3. Где в Ann могут происходить прерывания, почему?

4. Сколько есть прерываний, доступны ли они все для пользователя или нет?

5. Для чего используется структура TSS?

6. Чем различаются обработчики ошибок и прерываний?

7. Какая часть структуры заполняется общим обработчиком **interrupt_handler_common**?

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

9. Какой размер стека у обработчика ошибок и прерываний?

7 Системные вызовы

Для работы прикладных программ, как правило, необходим доступ к различным сервисам ядра: вывод на экран, выделение памяти, создание процессов. Однако, давать прикладным процессам прямой доступ к данным возможностям не безопасно, т.к. ошибка в прикладном процессе может привести к полной неработоспособности всего ядра. Поэтому для обращения к системным сервисам ядро предоставляет строго определенные точки входа: обработчики системных вызовов. Процесс передачи управление такому обработчику называется системным вызовом. В длинном режиме существует два способа выполнения системного вызова:

1. Используя инструкцию INT. В качестве параметра данной инструкции передается номер прерывания;

2. Используя инструкции SYSCALL и SYSRET.

Для использования инструкций SYSCALL и SYSRET необходимо инициализировать управляющие регистры процессора и создавать GDT в строго определенном формате, поэтому было принято решение для выполнения системных вызовов использовать инструкцию INT. В Ann системный вызов имеет номер 34.

Выполнение системных вызовов. Для выполнения системного вызова прикладные процессы используют библиотеку, которая скрывает детали реализации системных вызовов. С точки зрения прикладных программ системный вызов выглядит как вызов подпрограммы. Библиотечные функции сохраняют переданные аргументы и номер требуемого системного вызова в регистры, после чего выполняют команду `INT 34`. Для возврата значения из системного вызова используется регистр RAX, его содержимое после выполнения системного вызова возвращается библиотечной функцией прикладному процессу.

PUTS. Данный системный вызов принимает один аргумент в качестве параметра: указатель на строку, завершающуюся нулевым байтом и выводит ее на экран. Вывод на экран выполняется по одному символу, если адрес текущего символа находится вне адресного пространства вызывающего процесса: произойдет страничное прерывание и процесс будет уничтожен. При достижении конца строки экрана (80 символов), вывод продолжается со следующей строки. При достижении конца последней строки (25 строка), все строки сдвигаются на одну вверх (самая первая строка при этом теряется), последняя строка очищается и вывод продолжается с последней строки.

EXIT. Данный системный вызов не принимает аргументов. При выполнении системного вызова текущий процесс уничтожается и освобождаются занятые им

системные ресурсы: физические страницы, дескриптор процесса. Для того, чтобы определить какой процесс выполняется в данный момент (совершил системный вызов), ядро использует глобальную переменную, которая содержит указатель на дескриптор текущего процесса. Следует обратить внимание, что системный вызов освобождает только страницы и таблицы страниц, которые отвечают за отображение виртуальных адресов процесса, т.е. находящихся в полуинтервале `[0, USER_TOP)`. После освобождения всех страниц занятых процессом, ядро устанавливает статус дескриптора процесса равным `FREE` и добавляет дескриптор в список свободных дескрипторов процессов.

YIELD. Данный системный вызов позволяет процессу добровольно освободить процессор. При выполнении системного вызова ОС изменяет статус текущего процесса на `READY`. Вызвает планировщик, который выбирает следующий поцесс и запускает его. Если в очереди больше нет готовых процессов – будет снова запущен текущий процесс.

FORK. Данный системный вызов используется для порождения процессов. При выполнении системного вызова создается новый процесс (процесс-потомок). Для этого ОС выделяет новый дескриптор из таблицы дескрпторов процессов и копирует в структуру, описывающую контекст процесса-потомка содержимое структуры, описывающей контекст процесса-родителя, после чего заносит в регистр RAX, в контекте процесса-потока значение 0. Таким образом процесс потомок полностью идентичен процессу-родителю, за исключением одного момента: системный вызов возвращает 0 в процессе-потомке и идентификатор процесса (отличное от 0 значения) в процессе-родителе. Для клонирования процессов используется механизм копирования при записи (англ. Copy On Write). Все доступные для записи страницы помечаются как доступные только для чтения, т.е. для них сбрасывается бит «W» и устанавливается бит «COW»: это один из неиспользуемых страничным преобразованием бит (11й бит, если считать с 0), доступный для использования системным ПО. Следует отметить, что после выполнения системного вызова содержимое таблиц PML4 дочернего и родительского процессов совпадает.

При попытке записи одним из процессов в страницу доступную только для чтения произойдет страничное исключение. Обработчик страничного исключения, для страниц у которых установлен бит COW, выделяет новую страницу из списка свободных страниц и отображает ее в неиспользуемую область виртуального адресного пространства: `KERNEL_TEMP`, добавляя права для записи (бит «W»). После этого копирует содержимое оригинальной страницы в эту область, удаляет отображение для оригинальной страницы и отображает вместо него новую страницу. После чего,

удаляет отображение для KERNEL_TEMP. На рис. 7.1 показаны основные этапы создания копии процесса.

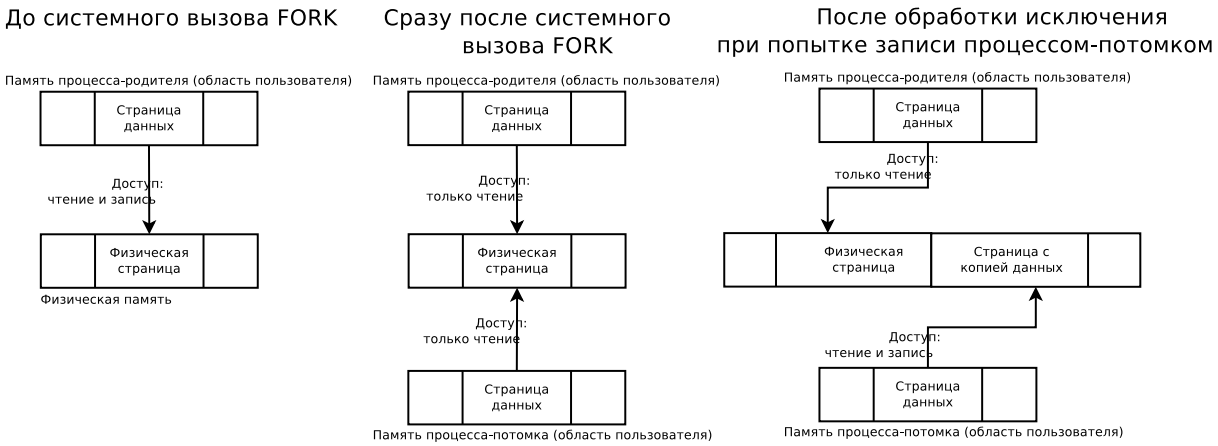

Рисунок 7.1 — Создание копии процесса и механизм копирования при записи

7.1 Лабораторная работа №5

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 34 (константа **INTERRUPT_VECTOR_SYSCALL**). Для выполнения системного вызова используется код приведенный в листинге 7.1. Код находится в файле `src/user/syscalls.c`.

Для передачи номера системного вызова используется регистр RAX, этот же регистр используется для последующего возврата результата системного вызова.

Параметры в системный вызов передаются через регистры RBX, RCX, RDX, RDI, RSI. Системные вызовы в разрабатываемой ОС либо не принимают аргументов, либо принимают один аргумент, поэтому 5 регистров достаточно. При реализации системных вызовов, требующих больше 5 аргументов, прикладному ПО необходимо создать на стеке структуру, содержащую требуемые аргументы и передавать ее адрес в одном из регистров. Ядро сможет обращаться к этой структуре, поскольку обработка системного вызова выполняется в контексте процесса, совершившего системный вызов.

Листинг 7.1 — Выполнение системных вызовов

```
1 asm volatile("int %1\n"
2     : "=a" (ret)
3     : "i" (INTERRUPT_VECTOR_SYSCALL),
4       "a" (syscall),
5       "b" (arg1),
6       "c" (arg2),
```

```
7         "d" (arg3),
8         "D" (arg4),
9         "S" (arg5)
10      : "cc", "memory");
```

Для выполнение системных вызовов прикладные процессы используют функции: `sys_puts`, `sys_yield`, `sys_exit`, `sys_fork`.

7.2 Клонирование процессов

Одной из особенностей разработанной ОС является возможность эффективного клонирования процессов, используя механизм копирования при записи. Клонирование процессов осуществляется с использование системного вызова `FORK`. Данный системный вызов создает новый процесс, выделяет ему PML4 и отображает в нее все страницы родительского процесса, причем для страниц, которые были доступны для записи сбрасывается бит «W» и устанавливается бит «COW» (11), означающий что страница должна быть скопирована при попытке записи в нее.

Когда один из процессов попытается записать данные в одну из страниц с битом «COW» произойдет страничное исключение. Обработчик страничного исключения скопирует страницу и установит для ее копии бит «W». Для того чтобы скопировать страницу, обработчик прерывания должен выполнить следующие действия:

1. выделить новую физическую страницу из списка свободных страниц, используя функцию `page_alloc`;

2. отобразить новую физическую страницу по виртуальному адресу `KERNEL_TEMP`, используя функцию `page_insert`;

3. копировать в `KERNEL_TEMP` содержимое оригинальной страницы, используя функцию `memcpy`;

4. отобразить новую физическую страницу по адресу, по которому произошло исключение, используя функцию `page_insert`, при этом удаляется отображение оригинальной физической страницы по этому адресу;

5. удалить отображение новой страницы в `KERNEL_TEMP`, т.к. оно больше не нужно.

7.2.1 Задание №11

Прежде всего надо дописать точку вхождения в прерывание системного вывоза **INTERRUPT_VECTOR_SYSCALL**. Лабораторная рабо-

та №4 в помощь. В **interrupt_handler** в swith инструкции для значения **INTERRUPT_VECTOR_SYSCALL** надо будет вызвать **syscall** и передать task.

7.2.2 Задание №12

Необходимо дописать функцию **syscall** в файле `src/kernel/syscall.c`. Номер системногов вызова храниться в регистре RAX. Перечисление системных вызовов находится `src/stdlib/syscalls.h`. Каждый вызов обрабатывается по своему. Добавьте проверку на неопознанный системный вызов. Если вызов неопознан, то можно использовать **panic()**).

1. **SYSCALL_PUTS** – вызов выводит информацию на экран;

2. **SYSCALL_EXIT** – прежде чем, завершить задачу, выведите какая задача, завершается. В этом случае функция должна передать управление функции **schedule**;

3. **SYSCALL_FORK** – вызвать функцию sys_fork, не забудьте сохранить возращаемое значение;

4. **SYSCALL_YIELD** – в этом случае функция должна передать управление функции **schedule**.

7.2.3 Задание №13

Переходим к функции **task_share_page**. Удалите аттрибут `unused`. Комментарии помогут вам понять, что надо сделать. Проверить разрешение на запись или что страница должна быть скопирована при попытке записи в нее. Если обе проверки не завершились успешно, то достаточно только вставить страницу в dest.

7.2.4 Задание №14

Осталось написать функцию **sys_fork**. Удалите аттрибут `unused`. Комментарии к функции должны помочь вам понять, что надо сделать. Основная идея в том, что надо пройтись по всем таблицам вниз, проверять на наличие записи в каждой из таблице и когда дойдете до таблицы PTE надо будет вызвать **task_share_page**. Чтобы получить 12 нижних битов, которые определяют разрешения на чтение/запись и т.д., можно использовать значение PTE_FLAGS_MASK

7.3 Контрольные вопросы

1. Какие есть способы реализации прерываний? Какой способ выбран и почему именно он?

2. Как выполняются системные вызовы?

3. Какие есть системные вызовы?

4. Как реализован вызов **sys_fork**?

8 Прикладные процессы

8.1 Лабораторная работа №6

Современные ОС для упрощения работы планировщика создают холостой (англ. idle) процесс, который используется для сбора различной статистики о системе. В разрабатываемой ОС такой процесс будет реализован с использованием потоков ядра. Его единственной задачей будет вызов планировщика для передачи управления другим процессам, таким образом в системе всегда будет как минимум один рабочий процесс, который можно запустить.

Разрабатываемая ОС не поддерживает приоритеты процессов, планирование осуществляется по алгоритму Round Robin [4].

Вытеснение процессов происходит при возникновении прерывания по таймеру или посредством системного вызова.

8.1.1 Задание №15

Дописать функцию **schedule**. Индекс процесса будет вычисляться, как остаток от деления от суммы next_task_idx и индекса массива tasks и TASK_MAX_CNT.

8.1.2 Задание №16

Дописать функцию **task_create**. Выделить память для стека (хватит одной страницы). Замапить стек, с правильными битами разрешения. После этого обновить регистры.

8.1.3 Задание №17

Дописать функцию **task_load**. Прежде всего надо загрузить новое значение pml4 в регистр `cr3` из task_context. Загрузить заголовки программы, по аналогии с первым и вторым загрузчиком. После этого загрузить в регистр `rip` точку входа `e_entry`. Если при проверке будут происходить ошибки надо будет перезагрузить регистр `cr3`.

8.1.4 Задание №18

Дописать функцию **task_load_segment**.

8.1.5 Задание №19

Дописать функцию **task_new**.

8.1.6 Задание №20

Дописать функцию **task_kill**. Пройтись по массиву процессов, проверить состояние процесса на TASK_STATE_RUN и на TASK_STATE_READY, что идентификатор id != task_id. Проверить регистр cs на доступ. Если все проверки прошли успешно, то вызвать функцию **task_destroy**.

9 Потоки ядра

9.1 Поток

Особенностью ANN ОС являются потоки ядра. Они реализованы на основе процессов, но выполняются в адресном пространстве ядра. Для создания потока ядра используется функция `thread_create`, которая позволяет задавать имя потока, главную функцию потока, аргумент, который необходимо передать в функцию и его размер.

Начальной функцией для всех потоков является функция `thread_foo`, которая принимает в качестве параметров указатель на дескриптор потока, указатель на функцию потока и указатель на аргумент, который нужно передать этой функции. Код функции `thread_foo` приведен в листинге 9.1.

Листинг 9.1 — Точка входа потоков ядра

```
1 static void thread_foo(struct task *thread, thread_func_t foo, void *arg) {
2     assert(thread != NULL && foo != NULL);
3     foo(arg);
4     task_destroy(thread);
5
6     // call schedule
7     asm volatile ("int3");
8 }
```

Согласно [6], для передачи аргументов в функции используются регистры RDI, RSI, RDX, R10, R8, R9. В листинге 9.2 приведен фрагмент кода, используемый для передачи параметров в функцию `thread_foo`.

Листинг 9.2 — Передача параметров в функцию `thread_foo`

```
1 task->context.gprs.rdi = (uintptr_t)task;
2 task->context.gprs.rsi = (uintptr_t)foo;
3 task->context.gprs.rdx = (uintptr_t)data;
```

Стек потоков ядра можно было реализовать несколькими способами:

1. Заранее выделить пул страниц, которые могут быть использованы в качестве стека потоков ядра;

2. Выделять страницы из общего пула и отображать их в определенный виртуальный адрес.

Преимущество первого подхода заключается в том, что он минимизирует накладные расходы, связанные с переключением между потоками, т.к. в этом случае все потоки могут использовать одну общую таблицу PML4 и нет необходимости изменять значение регистра CR3 при переключении контекста. Основным недостатком являет-

ся ограниченное число потоков, т.к. область памяти, в которой будут расположены страницы, которые можно использовать для стека, должна быть выделена заранее.

Преимуществом второго подхода является отсутствие необходимости резервировать определенные области ядра только для создания стеков. Кроме того, данный способ позволяет работать с потоками так же, как и с процессами, т.к. использовать одни и те же функции для передачи управления и уничтожения, что упрощает код. Основным недостатком данного способа являются накладные расходы, связанные с обновлением значения регистра CR3. Однако, в учебной ОС данный недостаток не является существенным, поэтому был выбран второй способ.

Следует заметить, что стек потока ядра находится вне виртуального адресного пространства ядра, поэтому из потока ядра нельзя напрямую вызвать планировщик для переключения задач. Для решения данной проблемы ядро использует прерывание int3, обработчик которого вызывает переключение задач.

9.1.1 Задание №21

Дописать функцию **thread_run**. Достаточно поменять состояние потока на `TASK_STATE_READY`.

9.1.2 Задание №22

Дописать функцию **thread_create**. По действиям функция почти аналогична функции **task_create**. Сначала лучше всего написать метку cleanup - после которой надо проверить task на не равенство NULL, иначе надо уничтожить выделенный task, можно использовать функцию `task_destroy` и возвращать NULL значение.

Теперь можно приступить к написанию актуальной части функции. Необходимо создать новый поток, в этом может помочь `task_new`, если функция возвратила NULL значение, то надо перейти на метку cleanup.

При успешном предыдущем шаге, теперь надо выделить память под stack (page_alloc()). При неудачном выполнении функции перейти на метку cleanup.

После успешного выделения памяти для стека, необходимо замапить стек, используйте page_insert. При неудачном выполнении функции перейти на метку cleanup.

Теперь надо подготовить стек и аргументы. Перезагрузить регистр cr3, новым значением из потока, не забыть сохранить предыдущее значение cr3 (поможет rcr3).

Теперь можно заняться загрузкой данных (data). Прежде всего проверить значение на NULL. После этого указатель на данные должен быть выровнен. После

этого можно скопировать данные функцией memcpy. Сохранить указатель в data, stack_top из data_ptr.

Адрес возврата будет высчитывать как разница между stack_top и размером uintptr_t. В этот адрес необходимо записать значение 0. Необходимо помнить только, что stack_top надо привести к типу uintptr_t. После этого можно воспользоваться кодом из листинге 9.2. В конце концов надо восстановить значение cr3, сохраненное в самом начале.

После этого надо загрузить сегментные регистры cs, ds, es, ss и регистры rip, rsp;

После всех действий этого можно возвратить созданный поток.

ЗАКЛЮЧЕНИЕ

В результате работы по созданию ядра операционной системы была реализована его основная функция как средства разделения аппаратных ресурсов: физической памяти, процессорного времени и устройства вывода информации. В итоге была создана многозадачная однопользовательская однопроцессорная система с монолитным ядром.

Недостатками полученной ОС являются отсутствие файловой системы и поддержки многопроцессорной архитектуры. Тем не менее, созданная система поддерживает современную архитектуры процессоров AMD64, реализует полноценную изоляцию процессов, виртуальную память на основе страниц, вытесняющую многозадачность, потоки уровня ядра и эффективное клонирование процессов на основе копирования памяти при ее изменении.

Разработанную операционную систему можно улучшить следующим образом: добавить возможность работы с несколькими процессорами [7] и средства синхронизации, добавить файловую систему и межпроцессное взаимодействие (англ. IPC), добавить поддержку дополнительного аппаратного обеспечения, например – сетевой карты.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Advanced Micro Devices. — AMD64 Architecture Programmer's Manual. Volume 2: System Programming, 2015.

2. Intel Corporation. — 82093AA I/O Advanced Programmable Interrupt Controller (IOAPIC), 1996.

3. *В.А., Крищенко*. Основы реализации операционных систем / Крищенко В.А.

4. *Э.С., Таненбаум*. Современные операционные системы / Таненбаум Э.С. — СПб.: Питер, 2015.

5. INTEL 80386 PROGRAMMER'S REFERENCE MANUAL 1986. — `http://css.csail.mit.edu/6.858/2011/readings/i386.pdf`.

6. System V Application Binary Interface AMD64 Architecture Processor Supplement. — `http://www.x86-64.org/documentation/abi.pdf`. — [Электронный ресурс; доступ 22 мая 2016].

7. Intel Corporation. — MultiProcessor Specification, 1997.