**1.) Процесс загрузки и инициализации персональных ЭВМ архитектуры x86. Функции загрузчика. Загрузка ядра JOS.**

После включения персонального компьютера (ПК) на базе процессора семейства Intel 80x86, его процессор начинает работу в реальном режиме адресации с сегментной организацией и выполнение инструкций процессора с адреса FFFF:0, инициализированного в паре регистров CS:IP (Code Segment : Instruction Pointer) после снятия сигнала RESET. В конце доступного процессору адресного пространства оперативной памяти из ПЗУ материнской платы отображен загрузчик базовой системы ввода-вывода (BIOS), на который передается выполнение по инструкции безусловного перехода, отображенной по адресу FFFF:0.

Программа инициализации BIOS с помощью программы POST проверяет, что устройства компьютера работают корректно и инициализирует их.

Затем BIOS опрашивает по порядку устройства из заранее сформированного (в консольной утилите CMOS SETUP) списка загрузочных устройств, пока не найдет загрузочное устройство. Если такое устройство не будет найдено, BIOS выведет на консоль ПК сообщение об ошибке и процесс загрузки будет остановлен. При обнаружении BIOS’ом загрузочного устройства, с последнего будет произведена попытка чтения данных его нулевого сектора (512 байт, в которых, как предполагается, должна находиться загрузочная запись). После успешного завершения чтения данных нулевого сектора загрузочного устройства в оперативную память по начальному адресу 0000:7С00, BIOS передает на этот адрес дальнейшее исполнение инструкций процессора.

Нулевой сектор загрузочного жесткого диска содержит так называемую «Главную загрузочную запись» (MBR — Master Boot Recorв), содержащую как данные первых четырёх записей таблицы разделов, так и инструкции процессора Intel 80x86, обеспечивающие поиск по этим записям активного раздела, с которого, посредством одной из функций базовой системы ввода-вывода — INT 13h, реализованной в виде обработчика прерывания процессора, будет считана загрузочная запись операционной системы. Этот загрузочный сектор, как правило, зависит от операционной системы и должен произвести загрузку в оперативную память ядра операционной системы с последующей передачей на него исполнения инструкций процессора. Если активного раздела не существует, или загрузочный сектор активного раздела некорректен, MBR может загрузить резервный начальный загрузчик и передать управление ему. Резервный начальный загрузчик должен выбрать раздел (зачастую с помощью пользователя), загрузить его загрузочный сектор и передать ему управление.

В процессе загрузки компьютера x86 вначале всегда отрабатывается BIOS. На этой стадии, кроме тестирования и инициализации оборудования компьютера, происходит также и выбор устройства, с которого будет происходить дальнейшая загрузка. Это может быть дискета, жёсткий диск, сетевой ресурс, встроенное ПЗУ или любое иное устройство (алгоритм выбора загрузочного устройства может быть различным и зависит от реализации BIOS). После выбора загрузочного устройства управление всей дальнейшей загрузкой BIOS полностью передаёт этому устройству.

**Загрузчик операционной системы** — [системное программное обеспечение](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5), обеспечивающее загрузку [операционной системы](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) непосредственно после включения [компьютера](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80).

Загрузчик операционной системы:

* обеспечивает необходимые средства для [диалога с пользователем](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D0%B0%D0%BB%D0%BE%D0%B3%D0%BE%D0%B2%D0%BE%D0%B5_%D0%BE%D0%BA%D0%BD%D0%BE) компьютера (например, загрузчик позволяет выбрать [операционную систему](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) для загрузки);
* приводит аппаратуру компьютера в состояние, необходимое для старта [ядра операционной системы](https://ru.wikipedia.org/wiki/%D0%AF%D0%B4%D1%80%D0%BE_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%BE%D0%B9_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B) (например, на не-x86 архитектурах перед запуском ядра загрузчик должен правильно настроить виртуальную память);
* загружает ядро операционной системы в [ОЗУ](https://ru.wikipedia.org/wiki/%D0%9E%D0%97%D0%A3). Загрузка ядра операционной системы не обязательно происходит с [жесткого диска](https://ru.wikipedia.org/wiki/%D0%96%D1%91%D1%81%D1%82%D0%BA%D0%B8%D0%B9_%D0%B4%D0%B8%D1%81%D0%BA). Загрузчик может получать ядро по сети. Ядро может храниться в [ПЗУ](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%BD%D0%BE%D0%B5_%D0%B7%D0%B0%D0%BF%D0%BE%D0%BC%D0%B8%D0%BD%D0%B0%D1%8E%D1%89%D0%B5%D0%B5_%D1%83%D1%81%D1%82%D1%80%D0%BE%D0%B9%D1%81%D1%82%D0%B2%D0%BE) или загружаться через последовательные интерфейсы;
* формирует параметры, передаваемые ядру операционной системы (например, ядру [Linux](https://ru.wikipedia.org/wiki/Linux%22%20%5Co%20%22Linux) передаются параметры, указывающие способ подключения корневой файловой системы);
* передаёт управление ядру операционной системы.

При компиляции и компоновке программы на C, такой как ядро JOS, компилятор преобразовывает каждый исходный файл (.с) в объектный (.о), содержащий инструкции на языке ассемблера, кодированные в двоичном формате, ожидаемом процессором. Затем компоновщик объединяет все скомпилированные объектные файлы в единый двоичный образ, такой как obj/kern/kernel, который в данном случае является двоичным файлом в формате ELF. ELF-файл начинается с ELF-заголовка фиксированной длины, затем идет заголовок программы переменной длины, в котором перечислены все секции, которые должны быть загружены. Определения на языке C для этих ELF-заголовков находятся в inc/elf.h. Нас интересуют следующие секции программы:

.text: исполняемые инструкции программы;

.rodata: данные только для чтения, такие как строковые константы (мы не будем запрещать чтение на аппаратном уровне);

.data: инициализированные данные программы

VMA (адреса связывания) и LMA (адреса загрузки) секции .text. Адрес загрузки - это тот адрес, по которому секция должна быть загружена в память. В структуре ELF это значение хранится в поле ph->p\_pa (в данном случае это действительно физический адрес, однако в спецификации ELF фактическое значение этого поля является неопределенным).

Адрес связывания секции - это адрес, с которого секция ожидает начать выполнение. Компоновщик кодирует адрес связывания различными способами, например, при использовании адреса глобальной переменной, в результате чего исполняемый файл, как правило, не будет работать, если он начинает выполнение с неправильного адреса.

BIOS загружает загрузочный сектор в память, начиная с адреса 0x7c00, который, таким образом, является адресом загрузки загрузочного сектора. С этого же места загрузочный сектор начинает выполнение, так что этот адрес также является адресом связывания.

Он читает каждую секцию ядра с диска в память по адресу, соответствующему адресу связывания секции, а затем переходит к точке входа в ядро.

**2.) Переключение контекстов. Кооперативное разделение времени.**

**Переключение контекста** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Context Switch*) — в [многозадачных](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B3%D0%BE%D0%B7%D0%B0%D0%B4%D0%B0%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C) [ОС](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) и средах, процесс прекращения выполнения процессором одной задачи (процесса, потока, нити) с сохранением всей необходимой информации и состояния, необходимых для последующего продолжения с прерванного места, и восстановления и загрузки состояния задачи, к выполнению которой переходит процессор.

В процедуру переключения контекста входит т. н. [планирование задачи](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D1%81%D0%BF%D0%B5%D1%82%D1%87%D0%B5%D1%80_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%BE%D0%B9_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B) — процесс принятия решения к какой задаче передать управление.

При переключении контекста происходит сохранение и восстановление следующей информации:

* Регистровый контекст регистров общего назначения (в том числе флаговый регистр)
* Контекст состояния сопроцессора с плавающей точкой
* Состояние регистров MMX/SSE (x86)
* Состояние сегментных регистров (x86)
* Состояние некоторых управляющих регистров (например, регистр CR3 отвечающий за страничное отображение памяти процесса) (x86)

В ядре ОС с каждым потоком связаны следующие структуры:

* Общая информация pid, tid, uid, gid, euid, egid,…
* Состояние процесса/потока
* Права доступа
* Используемые потоком ресурсы и блокировки
* Счетчики использования ресурсов (например таймеры использованного процессорного времени)
* Регионы памяти, выделенные процессу

С точки зрения прикладного уровня переключение контекста можно разделить на добровольное (voluntary) и принудительное (non-voluntary): выполняющийся процесс/поток может сам передать управление другому потоку, либо ядро само может отобрать управление.

* Ядро ОС может отобрать управление у выполняющегося процесса/потока при истечении кванта времени, выделенного на выполнение.
* Выполнение блокирующего системного вызова. Когда приложение производит ввод-вывод, ядро может решить, что можно отдать управление другому потоку/процессу в ожидании, пока запрошенный данным потоком дисковый либо сетевой ввод-вывод будет выполнен.
* Синхронизирующие примитивы ядра. Мьютексы, Семафоры и т. д.

**Кооперативная многозадачность**

Тип многозадачности, при котором следующая задача выполняется только после того, как текущая задача явно объявит себя готовой отдать процессорное время другим задачам. Как частный случай такое объявление подразумевается при попытке захвата уже занятого объекта, а также при ожидании поступления следующего сообщения от подсистемы пользовательского интерфейса.

При простом переключении активная программа получает все процессорное время, а фоновые приложения полностью замораживаются. При кооперативной многозадачности приложение может захватить фактически столько процессорного времени, сколько оно считает нужным. Все приложения делят процессорное время, периодически передавая управление следующей задаче.

* Преимущества кооперативной многозадачности: отсутствие необходимости защищать все разделяемые структуры данных объектами типа критических секций и mutex’ов, что упрощает программирование, особенно перенос кода из однозадачных сред в многозадачные.
* Недостатки: неспособность всех приложений работать в случае ошибки в одном из них, приводящей к отсутствию вызова операции «отдать процессорное время». Крайне затрудненная возможность реализации многозадачной архитектуры ввода-вывода в ядре ОС, позволяющей процессору исполнять одну задачу в то время, как другая задача инициировала операцию ввода-вывода и ждет её завершения.

**3.) Прерывания в x86. Инициализация таблицы дескриптеров прерываний IDT.**

Прерывание (англ. interrupt) — сигнал, сообщающий процессору о наступлении какого-либо события. При этом выполнение текущей последовательности команд приостанавливается, и управление передаётся обработчику прерывания, который реагирует на событие и обслуживает его, после чего возвращает управление в прерванный код.

Прерывания подразделяются на аппаратные (маскируемые и немаскируемые), вызываемые электрическими сигналами на входах процессора, и программные, выполняемые по команде INT xx. Программные прерывания, строго говоря, прерываниями не являются — это лишь своеобразный способ вызова процедур, но процессором они обрабатываются как разновидность прерываний.

Прерывания и исключения нарушают нормальный ход выполнения программы для обработки внешних событий или сообщения о возникновении особых условий или ошибок.

Аппаратные прерывания подразделяются на маскируемые и немаскируемые.

Маскируемые прерывания вызываются переходом в высокий уровень сигнала на входе INTR (Interrupt Request) при установленном флаге разрешения (IF=1). В этом случае процессор сохраняет в стеке регистр флагов, сбрасывает флаг. Прерывание с полученным номером вектора выполняется процессором так же, как и программное. Обработка текущего прерывания может быть в свою очередь прервана немаскируемым прерыванием, а если обработчик установит флаг IF, то и другим маскируемым аппаратным прерыванием.

Немаскируемые прерывания выполняются независимо от состояния флага IF по сигналу NMI (Non Mascable Interrupt). Его обработка не может прерываться под действием сигнала на входе NMI до выполнения команды IRET.

Обработка прерываний в защищенном режиме базируется на таблице дескрипторов прерываний (шлюзов прерываний) IDT – адрес начала и размер которой хранятся в регистре IDTR

Таблица прерываний может содержать до 256 дескрипторов. При попытке обслуживания прерывания с номером, выходящим за размер таблицы, генерируется исключение #DF. Под исключения отданы первые 32 номера (0 ¸31).

Векторы маскируемых прерываний определяются аппаратно. Контроллеры внешних прерываний (например, Intel 8259, программируемый контроллер прерываний) передают вектор на шину процессора. Использоваться могут любые векторы в диапазоне значений от 32 до 255.

Таблица векторов прерываний (англ. Interrupt Descriptor Table, IDT) используется в архитектуре x86 и служит для определения корректного ответа на прерывания и исключения.

В IDT используются следующие типы прерываний: аппаратные прерывания, программные прерывания и прерывания, зарезервированные процессором, называемые исключениями (первые 32) на случай возникновения некоторых событий (деление на ноль, ошибка трассировки, переполнение).

Первый этап инициализации выполняется BIOS, перед загрузкой ОС. Второй непосредственно самой операционной системой. Операционной системе доступно изменение некоторых адресов прерываний.

**4.) Обработка прерываний таймера. Вытесняющее разделение времени. Алгоритмы планирования процессорного времени.**

Программируемый интервальный таймер - микросхема, подключенная к IRQ0. Он может прервать работу процессора с частотой, определяемой пользователем (между 18,2 Hz и 11931 МГц). PIT является основным способом, используемым для реализации системных часов и единственным доступным методом реализации многозадачности (переключение процессов при прерывании).

В PIT есть внутренний генератор частоты, работающий приблизительно на частоте 11931 MHz. Этот тактовый сигнал подается через делитель частоты для получения в результате модуляции окончательной выходной частоты. В таймере есть 3 канала, каждый со своим собственным делителем частоты.

Канал 0 - наиболее часто используемый. Его выход подключен к IRQ0.

Канал 1 - чаще всего не используется и в современной аппаратуре просто не реализован. Используется для управления частотой регенерации памяти DRAM.

Канал 2 управляет работой динамика компьютера.

Приходит по IRQ\_CLOCK и обрабатывается и перед вызовом планировщика необходимо прочесть регистр статуса RTC и послать сигнал EOI на контроллер.

**Вытесняющая многозадачность**

Вид многозадачности, в котором операционная система сама передает управление от одной выполняемой программы другой в случае завершения операций ввода-вывода, возникновения событий в аппаратуре компьютера, истечения таймеров и квантов времени, или же поступлений тех или иных сигналов от одной программы к другой. В этом виде многозадачности процессор может быть переключен с исполнения одной программы на исполнение другой без всякого пожелания первой программы и буквально между любыми двумя инструкциями в её коде. Распределение процессорного времени осуществляется планировщиком процессов. К тому же каждой задаче может быть назначен пользователем или самой операционной системой определенный приоритет, что обеспечивает гибкое управление распределением процессорного времени между задачами (например, можно снизить приоритет ресурсоёмкой программе, снизив тем самым скорость её работы, но повысив производительность фоновых процессов). Этот вид многозадачности обеспечивает более быстрый отклик на действия пользователя.

* Преимущества: возможность полной реализации многозадачного ввода-вывода в ядре ОС, когда ожидание завершения ввода-вывода одной программой позволяет процессору тем временем исполнять другую программу. Сильное повышение надежности системы в целом, в сочетании с использованием защиты памяти — идеал в виде «ни одна программа пользовательского режима не может нарушить работу ОС в целом» становится достижимым хотя бы теоретически, вне вытесняющей многозадачности он не достижим даже в теории. Возможность полного использования многопроцессорных и многоядерных систем.
* Недостатки: необходимость особой дисциплины при написании кода, особые требования к его реентрантности, к защите всех разделяемых и глобальных данных объектами типа критических секций и mutex’ов.

**Алгоритмы планирования.**

1. First-Come, First-Served (FCFS - первым пришел, первым обслужен). Представим себе, что процессы, находящиеся в состоянии готовность, выстроены в очередь. Когда процесс переходит в состояние готовность, он помещается в конец этой очереди. Выбор нового процесса для исполнения осуществляется из начала очереди с удалением оттуда ссылки на него. Очередь подобного типа имеет в программировании специальное наименование – FIFO, сокращение от First In, First Out (первым вошел, первым вышел).

Такой алгоритм выбора процесса осуществляет невытесняющее планирование. Процесс занимает столько процессорного времени, сколько ему необходимо. После этого для выполнения выбирается новый процесс из начала очереди.

2. Round Robin (RR). Модификацией алгоритма FCFS является алгоритм, получивший название Round Robin (Round Robin – это вид детской карусели). По сути дела, это тот же самый алгоритм, только реализованный в режиме вытесняющего планирования. Можно представить себе все множество готовых процессов организованным циклически – процессы сидят на карусели. Карусель вращается так, что каждый процесс находится около процессора небольшой фиксированный квант времени, обычно 10 – 100 миллисекунд. Пока процесс находится рядом с процессором, он получает процессор в свое распоряжение и может исполняться.

3. Приоритетное планирование. При приоритетном планировании каждому процессу присваивается определенное числовое значение – приоритет, в соответствии с которым ему выделяется процессор. Процессы с одинаковыми приоритетами планируются в порядке FCFS.

Планирование с использованием приоритетов может быть как вытесняющим, так и невытесняющим. При вытесняющем планировании процесс с более высоким приоритетом, появившийся в очереди готовых процессов, вытесняет исполняющийся процесс с более низким приоритетом. В случае невытесняющего планирования он просто становится в начало очереди готовых процессов.

4. Алгоритм SJF (Shortest-Job-First) представляет собой частный случай приоритетного планирования. Для алгоритма SJF в качестве приоритета выступает оценка продолжительности выполнения процесса. Чем меньше значение этой оценки, тем более высокий приоритет имеет процесс.

5. Многоуровневые очереди (Multilevel Queue). Для систем, в которых процессы могут быть легко рассортированы по разным группам, был разработан другой класс алгоритмов планирования. Для каждой группы процессов создается своя очередь процессов, находящихся в состоянии готовность. Этим очередям приписываются фиксированные приоритеты. Например, приоритет очереди системных процессов устанавливается выше, чем приоритет очередей пользовательских процессов. Внутри этих очередей для планирования могут применяться самые разные алгоритмы. Так, например, для больших счетных процессов, не требующих взаимодействия с пользователем (фоновых процессов), может использоваться алгоритм FCFS, а для интерактивных процессов – алгоритм RR.

**5.) Обработка вложенных прерываний в x86. Средства синхронизации, состояние гонок, взаимные блокировки. Запрет прерываний, спинлоки, мьютексы, семафоры. Read-Copy-Update.**

**Вложенные исключения и прерывания**

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

Если процессор уже находится в режиме ядра и принимает вложенное исключение, он не сохраняет значения SS и ESP, так как не требует переключения стеков. Для типов исключений, которые не используют код ошибки, стек ядра, следовательно, выглядит следующим образом при входе в обработчик исключения:

+---------------------+ <---- старый ESP

| старый EFLAGS | " - 4

| 0x00000 | старый CS | " - 8

| старый EIP | " - 12

+---------------------+

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

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

**Синхронизация процессов** — приведение двух или нескольких процессов к такому их протеканию, когда определённые стадии разных процессов совершаются в определённом порядке, либо одновременно.

Синхронизация необходима в любых случаях, когда параллельно протекающим процессам необходимо взаимодействовать. Для её организации используются средства межпроцессного взаимодействия. Среди наиболее часто используемых средств — сигналы и сообщения, семафоры и мьютексы, каналы (англ. pipe), совместно используемая память.

Файл, Сигнал, Сокет, Канал, Именованный канал, Семафор, Разделяемая память,

Обмен сообщениями, Проецируемый в память файл, Очередь сообщений, Почтовый ящик

**Состоя́ние го́нки** (англ. race condition) — ошибка проектирования многопоточной системы или приложения, при которой работа системы или приложения зависит от того, в каком порядке выполняются части кода.

**Взаи́мная блокиро́вка** (англ. deadlock) — ситуация в многозадачной среде, при которой несколько процессов находятся в состоянии бесконечного ожидания ресурсов, занятых самими этими процессами.

|  |  |  |
| --- | --- | --- |
| **Шаг** | **Процесс 1** | **Процесс 2** |
| 0 | Хочет захватить A и B, начинает с A | Хочет захватить A и B, начинает с B |
| 1 | Захватывает ресурс A | Захватывает ресурс B |
| 2 | Ожидает освобождения ресурса B | Ожидает освобождения ресурса A |
| 3 | Взаимная блокировка |

**Запрет прерываний**

Это можно сделать командой CLI. Ее нужно поместить в начало критической последовательности команд, а в конце расположить команду STI, разрешающую процессору воспринимать прерывания. Команда CLI запрещает только маскируемые прерывания, немаскируемые всегда обрабатываются процессором. Для маскирования определенных аппаратных прерываний нужно просто послать требуемую цепочку битов в порт с адресом 21H, который соответствует регистру маски прерываний (IMR).

**Спинлок** (англ. Spinlock — циклическая блокировка) — низкоуровневый примитив синхронизации, применяемый в многопроцессорных системах для реализации взаимного исключения.

Физически спинлок представляет собой переменную в памяти и реализуется на атомарных операциях, которые должны присутствовать в системе команд процессора. Каждый процессор, желающий получить доступ к разделяемому ресурсу, атомарно записывает условное значение «занято» в эту переменную, используя аналог операции swap (в архитектуре x86 — xchg). Если предыдущее значение переменной (возвращаемое командой) было «свободно» то считается, что данный процессор получил доступ к ресурсу, в противном случае, процессор возвращается к операции swap и крутится в цикле ожидая, пока спинлок будет освобождён. После работы с разделяемым ресурсом процессор-владелец спинлока должен записать в него условное значение «свободно».

**Мью́текс** (англ. mutex, от mutual exclusion — «взаимное исключение») — одноместный семафор, служащий в программировании для синхронизации одновременно выполняющихся потоков.

Мьютексы — это простейшие двоичные семафоры, которые могут находиться в одном из двух состояний — отмеченном или неотмеченном (открыт и закрыт соответственно). Когда какой-либо поток, принадлежащий любому процессу, становится владельцем объекта mutex, последний переводится в неотмеченное состояние. Если задача освобождает мьютекс, его состояние становится отмеченным.

**Семафо́р** — объект, ограничивающий количество потоков, которые могут войти в заданный участок кода. Определение введено Эдсгером Дейкстрой. Семафоры используются при передаче данных через разделяемую память.

P-операция над семафором представляет собой попытку декремента значения семафора на 1. Если перед выполнением P-операции значение семафора было больше 0, то P-операция выполняется без задержек. Если перед выполнением P-операции значение семафора было равным 0, то процесс, выполняющий P-операцию, переводится в состояние ожидания до тех пор, пока значение семафора не станет большим 0.

V-операция над семафором представляет инкремент значения семафора на 1. Если при этом имеются процессы, задержанные на выполнении P-операции на данном семафоре, один из этих процессов выходит из состояния ожидания и может выполнить свою P-операцию.

**Read – Copy Update** (RCU) – техника синхронизации, предназначенная для «почти read-only», то есть редко изменяемых, структур данных.

Первоначально идея была очень проста: есть некоторая редко изменяемая структура данных. Если нам требуется изменить её, то мы делаем её копию и производим изменение — добавление или удаление данных — именно в копии. При этом параллельные читатели работают с первоначальной, не измененной структурой. В некоторый безопасный момент времени, когда нет читателей, мы можем подменить структуру данных на измененную копию. В результате все последующие читатели будут видеть изменения, произведенные писателем.

**6.) Виртуальная память. Модели организации оперативной памяти. Сегментная и страничная трансляция x86. Таблицы трансляции.**

**Виртуа́льная па́мять** (англ. Virtual memory) — технология управления памятью ЭВМ, разработанная для многозадачных операционных систем. При использовании данной технологии для каждой программы используются независимые схемы адресации памяти, отображающиеся тем или иным способом на физические адреса в памяти ЭВМ. Позволяет увеличить эффективность использования памяти несколькими одновременно работающими программами, организовав множество независимых адресных пространств, и обеспечить защиту памяти между различными приложениями. Также позволяет программисту использовать больше памяти, чем установлено в компьютере, за счет откачки неиспользуемых страниц на вторичное.

При использовании виртуальной памяти упрощается программирование, так как программисту больше не нужно учитывать ограниченность памяти, или согласовывать использование памяти с другими приложениями. Для программы выглядит доступным и непрерывным все допустимое адресное пространство, вне зависимости от наличия в ЭВМ соответствующего объёма ОЗУ.

**Одиночное непрерывное распределение**

Данная модель распределения оперативной памяти является одной из самых простых и основывается на том, что все адресное пространство подразделяется на два компонента. В одной части памяти располагается и функционирует операционная система, а другая часть выделяется для выполнения прикладных процессов.

**Распределение неперемещаемыми разделами**

Данная модель строится по следующим принципам. Опять же, все адресное пространство оперативной памяти делится на две части. Одна часть отводится под операционную систему, все оставшееся пространство отводится под работу прикладных процессов, причем это пространство заблаговременно делится на N частей (назовем их разделами), каждая из которых в общем случае имеет произвольный фиксированный размер. Эта настройка происходит на уровне операционной системы. Соответственно, очередь прикладных процессов разделяется по этим разделам.

**Распределение перемещаемыми разделами**

Данная модель распределения разрешает загрузку произвольного (нефиксированного) числа процессов в оперативную память, и под каждый процесс отводится раздел необходимого размера. Соответственно, система допускает перемещение раздела, а, следовательно, и процесса. Такой подход позволяет избавиться от фрагментации.

**Сегментная адресация памяти**

Рассмотрим, как связаны между виртуальный и физический адреса. Пусть у нас уже имеется виртуальный адрес (см. Рис.1). При формировании физического адреса можно выделить три стадии:

1. Вычисление эффективного адреса(смещение);

2. Затем находится адрес в линейном виртуальном 4-х гигабайтном адресном пространстве; (базовый адрес складывается с эффективным)

3. Линейное виртуальное адресное пространство отражается на физические блоки в основной памяти

На первом этапе просто берутся младшие 16 бит, которые и являются эффективным адресом, их еще называют смещением внутри сегмента.



На втором этапе нужно найти сам сегмент. В этом нам поможет вторая часть нашего виртуального адреса, которая служит индексом в таблице дескрипторов. Дескриптор это структура данных, которая описывает сегмент (см. Рис. 2). Она имеет размер 8 байт и содержит в себе такую информацию, как базовый адрес, права доступа и предел сегмента. Таким образом, зная индекс в таблице дескрипторов, мы знаем всю информацию о нужном нам сегменте. Затем уже происходит вычисление физического адреса путем сложения смещения и базы сегмента.



Но процессору необходимо знать где находится таблица дескрипторов. Для этого есть специальные регистры. GDTR (Global Descriptor Table Register) - регистр из 6 байт: первые 32 бита определяют адрес глобальной дескрипторной таблицы (GDT - Global Descriptor Table), оставшиеся 16 бит содержат в себе предел таблицы или, другими словами, еѐ размер. Помимо неѐ есть еще множество локальных дескрипторных таблиц (LDT Local Descriptor Table), из которых в один момент времени доступна только одна, какая - определяется регистром LDTR. LDTR (Local Desriptor Table Register) - регистр из 16 бит, который содержит в себе селектор необходимой таблицы. Селектор это структура (см. Рис.3), размером 2 байта, которая содержит в себе индекс дескриптора в GDT, который описывает сегмент, где расположена таблица, и прочую административную информацию. Обращение к глобальной таблице происходит только один раз, когда изменяется LDTR, затем найденый дескриптор помещается в недоступную часть регистра. Стоит отметить, что GDT присутствует в системе в единственном экземпляре, а количество LDT может доходить до 8192.



**Страничная адресация памяти**

Виртуальный адрес разбивается на три части: индекс в каталоге таблиц, индекс в таблице страниц и смещение внутри страницы. Запись, соответствующая индексу в каталоге таблиц, содержит в себе физический адрес нужной таблицы страниц. Затем индекс в таблице страниц указывает нам на запись, в которой находится ссылка на конкретную страницу, а именное еѐ адрес. И наконец, к адресу страницы прибавляется смещение и мы получаем нужную ячейку памяти. На рисунке ниже показана схема страничной адресации.



В механизме страничной адресации присутствуют такие типы данных как каталоги таблиц (Page Directory) и таблицы страниц (Page Table), обе эти структуры состоят из 1024 элементов(записей) размером 4 байта. Записи из каталога таблиц ссылаются на таблицы страниц, которые в свою очередь ссылаются на конкретные страницы. Все эти структуры выравнены по границе страницы.

 Записи в таблице директорий и каталогах страниц называются дескрипторами таблиц страниц и каталогов страниц соответственно. Помимо базы адреса в них содержатся управляющие флаги. Таким образом с помощью структуры этих дескрипторов решается задача изолирования адресного пространства процессов.

 

**Буфер ассоциативной трансляции** (англ. Translation lookaside buffer, TLB) — это специализированный кэш центрального процессора, используемый для ускорения трансляции адреса виртуальной памяти в адрес физической памяти. TLB используется всеми современными процессорами с поддержкой страничной организации памяти. TLB содержит фиксированный набор записей (от 8 до 4096) и является ассоциативной памятью. Каждая запись содержит соответствие адреса страницы виртуальной памяти адресу физической памяти.

**7.) Переключение между режимами работы процессора. Прерывания и системные вызовы. Выполнение системных вызовов без переключения в привилегированный режим.**

**Переключение в защищенный режим:**

Переключение процессора в защищенный режим из реального осуществляется загрузкой в системный регистр CR0 слова с единичным значением бита РЕ (Protect Enable). До переключения в памяти должны быть проинициализированы необходимые таблицы дескрипторов IDT и GDT. Переключение процессора из защищенного режима в реальный возможно не только через аппаратный сброс, как это было у 80286, но и сбросом бита РЕ в CR0. До этого переключения также необходимо загрузить в сегментные регистры селекторы дескрипторов, описывающие свойства сегментов стандартного реального режима.

**Возврат в реальный режим:**

* Выполняется сброс процессора.
* Восстанавливаются регистры реального режима.
* Затем разрешаются маскируемые и немаскируемые прерывания.
* Для полного демаскирования прерываний следует сбросить маску в контроллере прерываний.

**Прерывание** (англ. interrupt) — сигнал, сообщающий процессору о наступлении какого-либо события. При этом выполнение текущей последовательности команд приостанавливается, и управление передаётся обработчику прерывания, который реагирует на событие и обслуживает его, после чего возвращает управление в прерванный код.

В зависимости от источника возникновения сигнала прерывания делятся на:

* асинхронные, или внешние (аппаратные) — события, которые исходят от внешних источников (например, периферийных устройств) и могут произойти в любой произвольный момент: сигнал от таймера, сетевой карты или дискового накопителя, нажатие клавиш клавиатуры, движение мыши. Факт возникновения в системе такого прерывания трактуется как запрос на прерывание (англ. Interrupt request, IRQ);
* синхронные, или внутренние — события в самом процессоре как результат нарушения каких-то условий при исполнении машинного кода: деление на ноль или переполнение стека, обращение к недопустимым адресам памяти или недопустимый код операции;
* программные (частный случай внутреннего прерывания) — инициируются исполнением специальной инструкции в коде программы. Программные прерывания как правило используются для обращения к функциям встроенного программного обеспечения (firmware), драйверов и операционной системы.

Внешние прерывания в зависимости от возможности запрета делятся на:

* маскируемые — прерывания, которые можно запрещать установкой соответствующих битов в регистре маскирования прерываний (в x86-процессорах — сбросом флага IF в регистре флагов);
* немаскируемые (англ. Non-maskable interrupt, NMI) — обрабатываются всегда, независимо от запретов на другие прерывания. К примеру, такое прерывание может быть вызвано сбоем в микросхеме памяти.

**Систе́мный вы́зов** (англ. system call) в программировании и вычислительной технике — обращение прикладной программы к ядру операционной системы для выполнения какой-либо операции.

Современные операционные системы (ОС) предусматривают разделение времени между выполняющимися вычислительными процессами (многозадачность) и разделение полномочий, препятствующее исполняемым программам обращаться к данным других программ и оборудованию. Ядро ОС исполняется в привилегированном режиме работы процессора. Для выполнения межпроцессной операции или операции, требующей доступа к оборудованию, программа обращается к ядру, которое, в зависимости от полномочий вызывающего процесса, исполняет либо отказывает в исполнении такого вызова.

**8.) Аппаратно-программные средства поддержки мультипрограммного режима – система прерываний, защита памяти, привилегированный режим.**

**Система прерываний**. Система прерываний необходима как для режима разделения времени, так и для обеспечения параллельной работы центрального процессора и периферийных устройств, так как она обеспечивает саму возможность реакции на события и автоматического переключения с одной программы на другую.

**Механизм защиты памяти**. Этот механизм обеспечивает безопасность одновременного нахождения в оперативной памяти нескольких независимых программ. Защита памяти гарантирует, что одна программа не сможет случайно или же предумышленно обратиться в память другой программы (по записи или даже по чтению данных). Очевидно, что без такого механизма мультипрограммный режим просто невозможен. Незаконное обращение к чужим ресурсам (в частности, к чужой оперативной памяти) "по научному" называется несанкционированным доступом.

Механизм защиты оперативной памяти на современных ЭВМ устроен весьма сложно, и часто связан с механизмом так называемой виртуальной памяти.

**Аппарат привилегированных команд**. Это свойство иногда называется аппаратом привилегированных команд, а иногда – защищённым режимом работы центрального процессора, и заключается оно в следующем: все команды, которые может выполнять центральный процессор, разбиваются на два класса. Команды из одного класса называются обычными командами или командами пользователя, а команды из другого класса – привилегированными или запрещёнными командами.

Далее, в центральном процессоре располагается специальный одноразрядный регистр режима работы, который может, естественно, принимать только два значения: 0 и 1. Значение этого регистра и определяют тот режим, в котором в данный момент работает центральный процессор: обычный режим (или режим пользователя) или привилегированный режим.

В привилегированном режиме центральному процессору разрешается выполнять все команды языка машины, а в режиме пользователя – только обычные (не привилегированные) команды. При попытке выполнить привилегированную команду в пользовательском режиме центральным процессором вырабатывается сигнал прерывания, а сама команда, естественно, не выполняется.

Привилегированными должны быть и все команды, которые обращаются к внешним (периферийным) устройствам. Выход один – программа пользователя должна обратиться к определённым служебным процедурам, с просьбой выполнить для неё ту работу, которую сама программа пользователя сделать не в состоянии. Эти служебные процедуры, естественно, должны работать в привилегированном режиме. Переключение из привилегированного режима в режим пользователя обычно производится по некоторой (не привилегированной) машинной команде. Значительно сложнее обстоит дело с такой опасной операцией, как переключение центрального процессора из обычного режима работы в привилегированный режим. Это переключение невозможно выполнить по какой-либо машинной команде. Обычно переключение в привилегированный режим производится автоматически при обработке центральным процессором сигнала прерывания, в этом случае процедура-обработчик прерывания уже начинает свою работу в привилегированном режиме. Иногда переключение в привилегированный режим производится центральным процессором при вызове специальных системных процедур, которые имеют полномочия для работы в привилегированном режиме.

**Таймер**. Встроенные в компьютер электронные часы (таймер) появились ещё до возникновения мультипрограммного режима работы. Действительно, это единственное внешнее устройство, которое гарантированно и периодически посылает центральному процессору сигналы прерываний. Без таких сигналов некоторые программы могли бы войти в выполнение бесконечного цикла, и ничто не могло бы вывести компьютер из этого состояния.

**9.) Файловые системы. Основные задачи файловых систем.**

**Фа́йловая систе́ма** (англ. file system) — порядок, определяющий способ организации, хранения и именования данных на носителях информации в компьютерах. Файловая система определяет формат содержимого и способ физического хранения информации, которую принято группировать в виде файлов. Конкретная файловая система определяет размер имен файлов и (каталогов), максимальный возможный размер файла и раздела, набор атрибутов файла. Некоторые файловые системы предоставляют сервисные возможности, например, разграничение доступа или шифрование файлов.

Файловая система связывает носитель информации с одной стороны и API для доступа к файлам — с другой. Когда прикладная программа обращается к файлу, она не имеет никакого представления о том, каким образом расположена информация в конкретном файле, так же, как и на каком физическом типе носителя (CD, жёстком диске, магнитной ленте, блоке флеш-памяти или другом) он записан. Всё, что знает программа — это имя файла, его размер и атрибуты. Эти данные она получает от драйвера файловой системы. Именно файловая система устанавливает, где и как будет записан файл на физическом носителе (например, жёстком диске).

С точки зрения операционной системы (ОС), весь диск представляет собой набор кластеров (как правило, размером 512 байт и больше). Драйверы файловой системы организуют кластеры в файлы и каталоги (реально являющиеся файлами, содержащими список файлов в этом каталоге). Эти же драйверы отслеживают, какие из кластеров в настоящее время используются, какие свободны, какие помечены как неисправные.

Однако файловая система не обязательно напрямую связана с физическим носителем информации. Существуют виртуальные файловые системы, а также сетевые файловые системы, которые являются лишь способом доступа к файлам, находящимся на удалённом компьютере.

**Задачи файловой системы**

Основные функции любой файловой системы нацелены на решение следующих задач:

* именование файлов;
* программный интерфейс работы с файлами для приложений;
* отображения логической модели файловой системы на физическую организацию хранилища данных;
* организация устойчивости файловой системы к сбоям питания, ошибкам аппаратных и программных средств;
* содержание параметров файла, необходимых для правильного его взаимодействия с другими объектами системы (ядро, приложения и пр.).
* В многопользовательских системах появляется ещё одна задача: защита файлов одного пользователя от несанкционированного доступа другого пользователя, а также обеспечение совместной работы с файлами, к примеру, при открытии файла одним из пользователей, для других этот же файл временно будет доступен в режиме «только чтение».