МОСКОВСКИЙ АВТОМОБИЛЬНО-ДОРОЖНЫЙ
ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ (МАДИ)

К.Н. МЕЗЕНЦЕВ

ОПЕРАЦИОННЫЕ СИСТЕМЫ

КУРС ЛЕКЦИЙ

МОСКОВСКИЙ АВТОМОБИЛЬНО-ДОРОЖНЫЙ
ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
(МАДИ)

К.Н. МЕЗЕНЦЕВ

ОПЕРАЦИОННЫЕ СИСТЕМЫ

КУРС ЛЕКЦИЙ

Под редакцией д-ра техн. наук, проф. А.Б. Николаева

Утверждено
в качестве учебного издания
редсоветом МАДИ

МОСКВА
МАДИ
2016

УДК 004.451
ББК 32.972.51
М442

Рецензенты:
Юрчик П.Ф. – д-р техн. наук, проф. каф. «Автоматизированные системы управления» (МАДИ),
Никитченко И.И. – канд. техн. наук, доц., зав. каф. «Информатика и информационные таможенные технологии»
(Российская таможенная академия).

Мезенцев, К.Н.
М442 Операционные системы: курс лекций / К.Н. Мезенцев; под ред. д-ра техн. наук, проф. А.Б. Николаева. – М.: МАДИ, 2016. – 140 с.

Курс лекций по дисциплине «Операционные системы» предназначен для студентов, обучающихся по направлениям подготовки бакалавриата 09.03.01 – «Информатика и вычислительная техника» и 09.03.02 – «Информационные системы и технологии».

Данное учебное издание представляет собой конспект лекций для семестрового курса обучения. В нем рассмотрены концепции организации современных операционных систем, методы диспетчеризации процессов, методы управления распределением оперативной памяти и устройствами ЭВМ, правила организации файловых систем, а также основы системного программирования в операционной системе Linux и концепции разработки операционных систем для многопроцессорных ЭВМ.

УДК 004.451
ББК 32.972.51

ВВЕДЕНИЕ

Материал данного курса лекций позволит студентам получить навыки работы в качестве пользователя любой из современных операционных систем (ОС). Под такими ОС понимаются ОС семейства MS Windows, Mac OS или Linux.

Лекции разделены на девять учебных модулей.

В первом модуле «Основы построения современных ОС», (лекция №1) рассматриваются концепции построения ядер современных ОС, методы организации прикладных сред, обсуждаются вопросы совместимости программного обеспечения, созданного для различных ОС, приводится классификация ОС с точки зрения организации эффективной работы пользователя и взаимодействия с периферийными устройствами ЭВМ.

Во втором модуле «Управление процессами», (лекции № 2, 3) вводится понятие «процесс», обсуждаются стратегии выполнения процессов в среде ОС, приводятся сведения о высокоуровневых механизмах синхронизации процессов, таких как семафор Дейкстры, монитор Хора и сообщения.

В третьем модуле «Управление распределением оперативной памяти» (лекции № 4, 5) рассмотрена классификация современных устройств, обеспечивающих хранение данных; приводятся сведения о правилах связывания адресов процесса с физическими адресами оперативной памяти; обсуждаются технологии выделения оперативной памяти процессам на программном и аппаратном уровне, изучаются правила организации и функционирования виртуальной памяти ЭВМ.

В четвертом модуле «Организация ввода – вывода данных» (лекции № 6, 7) рассматриваются принципы организации ЭВМ, отвечающие архитектуре фон Неймана; обсуждаются правила взаимодействия между процессором ЭВМ, устройствами и процессами на основе механизма прерываний; приводятся сведения о логической организации базовой системы ввода – вывода ЭВМ.

В пятом модуле «Организация файловой системы» (лекции №8, 9) приводятся сведения о правилах физической организации хранения

данных на магнитном носителе; обсуждаются стратегии организации эффективного доступа к магнитному носителю в процессе хранения и записи данных; даются сведения о логической организации файловой системы на основе технологии FAT, правила организации файловой системы NTFS и файловой системы ОС UNIX.

В шестом модуле «Основы системного программирования» (лекции №10, 11, 12) приводятся сведения о командах терминала ОС Linux; обсуждаются правила разработки Bash-сценариев путем использования команд организации ветвления, циклического выполнения команд; даются сведения о правилах организации ввода – вывода данных в Bash-сценариях и использовании функций в сценариях; обсуждаются правила управления процессами, создания процессов и организации взаимодействия между процессами.

В седьмом модуле «Основы низкоуровневого программирования» (лекция №13) приводятся сведения о правилах создания системных программ на языке Ассемблер для ОС Linux, использующих стандартные потоки ввода – вывода байтов; изучается технология использования системных вызовов в ассемблерной программе для выполнения действий над файлами и директориями.

В восьмом модуле «Поиск данных и файлов» (лекция №14) изучаются правила организации поиска объектов файловой системы ОС Linux путем использования утилиты find; приводятся сведения о правилах организации поиска текстовой информации в файлах путем использования утилиты grep.

В девятом модуле «Основы организации распределенных ОС» (лекция №15) рассматриваются концепции построения многопроцессорных ЭВМ, обсуждаются правила организации распределенной обработки данных и особенности ОС, необходимые для организации многопроцессной и распределенной обработки данных.

Каждая лекция модуля заканчивается списком контрольных вопросов. Основными источниками для составления конспекта лекций послужили классические труды Э. Танненбаума [4, 10] и материалы учебного пособия [1], созданного при поддержке компании Intel.

Лекция №1. ОРГАНИЗАЦИЯ ОПЕРАЦИОННЫХ СИСТЕМ

Классификация программного обеспечения, архитектура ОС, прикладные среды, эффективность работы ОС.

Общие понятия

Программное обеспечение современных ЭВМ может быть разделено на два больших класса – «Системное программное обеспечение» и «Прикладное программное обеспечение».

Рассмотрим особенности системного программного обеспечения. Системное программное обеспечение подразделяется на ***утилиты*** и ***операционные системы***.

Операционную систему можно рассматривать как программу, под управлением которой выполняются задачи пользователя ЭВМ и осуществляется диалог между пользователем и ЭВМ.

Операционная система состоит из ядра и оболочки. Составные части ядра:

- программа управления файлами;
- драйвера – программы для взаимодействия с устройствами ЭВМ;
- программа управления памятью;
- планировщик (диспетчер) для управления выполнением заданий процессором ЭВМ;
- оболочка – служит для организации взаимодействия с пользователем.

Организация взаимодействия в среде ОС между пользователем и ЭВМ строится на базе двух концепций: графический интерфейс и работа в режиме консоли.

Графический интерфейс пользователя – GUI (Graphical User Interface). Главная особенность такого интерфейса – наличие графических окон и «инструментов» – пиктограмм для быстрого доступа к функциям программ.

Консоль – ввод команд для управления работой ОС в режиме терминала. Пользователь ОС получает в свое распоряжение специ-

альное окно – консоль, где он должен вводить команду для выполнения действия в среде ОС. Такой режим работы также называют текстовым, так как результат выполнения команды – набор строк сообщений.

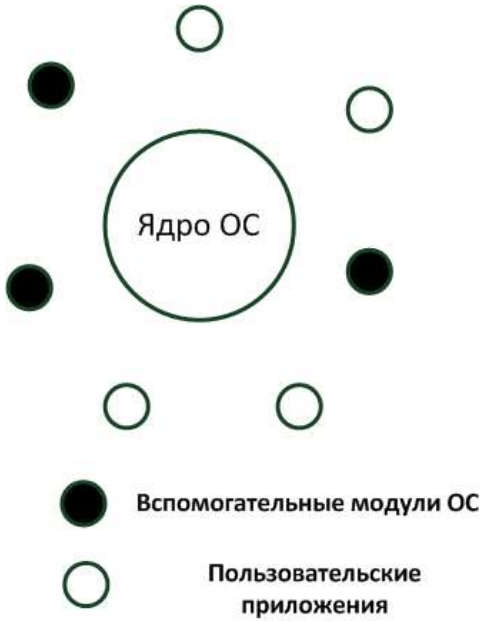

Рис. 1.1. Концептуальная модель ОС

На рис. 1.1 показана принципиальная схема организации ОС. В составе ОС выделяют ядро и утилиты. Это вспомогательные модули ОС. Утилиты позволяют решать сервисные задачи пользователю ОС. Например, создавать файлы и редактировать содержание файлов. Осуществлять управление файловой системой, выполнять тестирование ЭВМ и целостности файловой системы. Функции ядра ОС будут рассмотрены далее.

Пользовательский режим и режим ядра

Процессоры современных ЭВМ поддерживают два режима работы:

- пользовательский (user mode);
- привилегированный, который также называют режимом ядра (kernel mode) или режимом супервизора (supervisor mode).

Операционная система или некоторые ее части работают в привилегированном режиме, а приложения – в пользовательском режиме. На рис. 1.2 показана схема организации ОС с учетом двух режимов работы процессора.

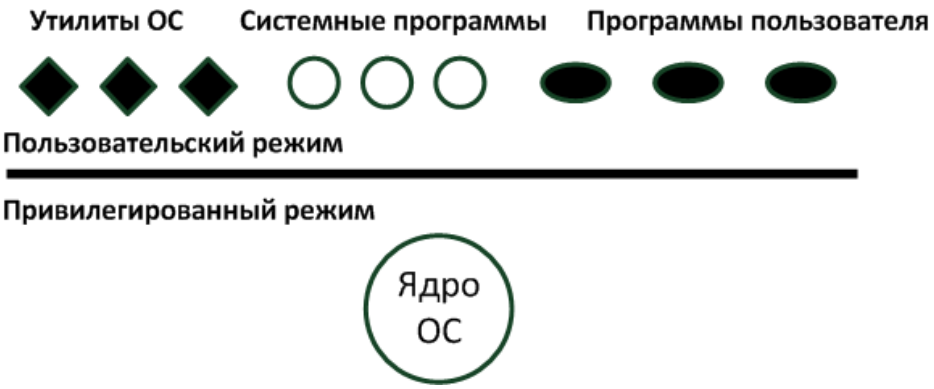

Рис. 1.2. Пользовательский и привилегированный режимы

Для выполнения определенного действия в среде ОС нужно запросить сервис операционной системы. Например, создание файла, запуск программы и т.д. Процесс запроса сервиса заключается в обращении к ядру ОС путем выполнения системного вызова. На рис. 1.3 показана временная диаграмма организации системного вызова ОС.

Слои ядра ОС

Ядро ОС выступает посредником между прикладным программным обеспечением и аппаратурой ЭВМ так, как это показано на рис. 1.4. Рассмотрим структуру ядра ОС. В составе ОС можно выделить ряд слоев по функциональному признаку. Состав этих слоев может варьироваться в конкретных реализациях ОС.

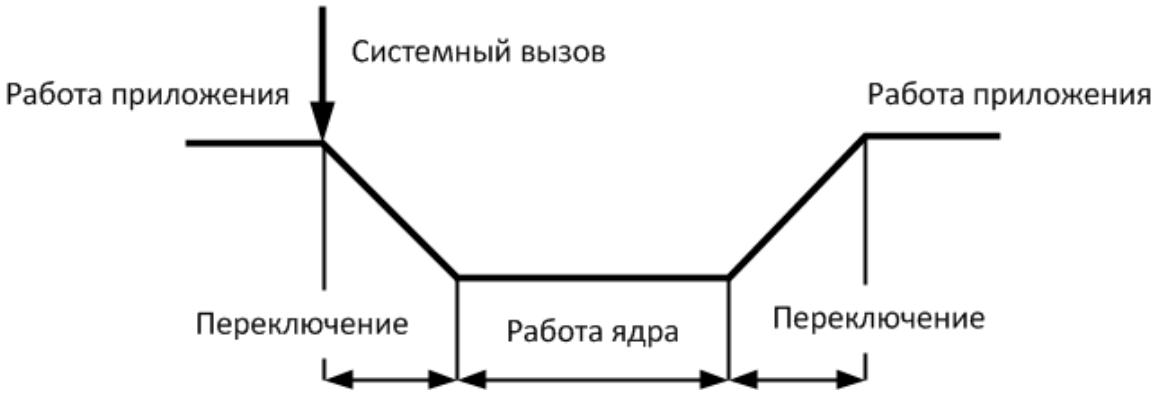

Рис. 1.3. Системный вызов ОС

1. Средства аппаратной поддержки ОС:

- средства поддержки привилегированного режима;
- система прерываний;

- средства переключения контекстов процессов;
- средства защиты областей памяти.

2. Машинно-зависимые компоненты ОС.

Этот слой образуют программные модули, в которых отражается специфика аппаратной платформы компьютера.

Этот слой отделяет вышележащие слои ядра от особенностей аппаратуры.

3. Базовые механизмы ядра. Этот слой выполняет наиболее примитивные операции ядра:

- программное переключение контекстов процессов;
- диспетчеризацию прерываний;
- перемещение страниц из памяти на диск и обратно.

4. Менеджеры ресурсов. Этот слой состоит из функциональных модулей, реализующих стратегические задачи по управлению основными ресурсами вычислительной системы. Обычно на данном слое работают менеджеры (называемые также диспетчерами):

- процессов;
- ввода-вывода;
- файловой системы;
- оперативной памяти.

Каждый из менеджеров ведёт учёт свободных и используемых ресурсов определённого типа и планирует их распределение в соответствии с запросами приложений.

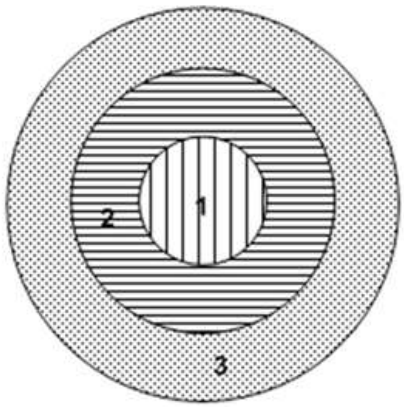

Рис. 1.4. Ядро ОС и аппаратура ЭВМ:
1 – аппаратура; 2 – ядро; 3 – утилиты и прикладные программы

Внутри слоя менеджеров существуют тесные взаимные связи, отражающие тот факт, что для выполнения процессу нужен доступ одновременно к нескольким ресурсам – процессору, области памяти, возможно, к определённому файлу или устройству ввода-вывода.

5. Интерфейс системных вызовов. Этот слой является самым верхним слоем ядра и взаимодействует непосредственно с приложениями и системными утилитами, образуя прикладной программный интерфейс операционной системы API. Функции API (Application Programming Interface) – интерфейса программирования приложений, обслуживающего системные вызовы, предоставляют доступ к ресурсам системы в удобной и компактной форме, без указания деталей их физического расположения.

Архитектуры ОС

Абстрактные машины

Процессор вместе с ОС можно рассматривать как машину, которая исполняет программы пользователя ЭВМ путем вызова функций ОС. Такую машину можно рассматривать как машину, реализованную с помощью программного обеспечения. Эту машину можно назвать как абстрактную. Абстрактная машина реализует сложные операции посредством серии простых операций. При этом архитектура ОС представляет собой набор «слоев» (рис. 1.5). Каждый такой слой – абстрактная машина. Чем выше слой в иерархии слоев, тем больше унификация. Чем ниже слой, тем более примитивные, атомарные функции он реализует. Например, на верхнем слое определена функция «Создать файл», выполнение этой функции требует использования «примитивных функций» нижнего слоя: «Отыскать свободный блок на диске», «Запись байтов в блоки», «Обновление таблицы файлов ОС».

Таким образом, для выполнения определенного действия выполняется обращение к верхнему слою иерархию, который затем обращается к нижним слоям иерархии для выполнения нужного действия.

Несколько абстрактных машин образуют единую архитектуру ОС. Абстрактные машины коммутируются друг с другом посредством стандартного интерфейса.

В результате возникает иерархическая организация ОС. Верхний слой образует стандартная библиотека системных функций. Обращение к такой функции – системный вызов, который выполняется в режиме ядра.

Монолитная архитектура

Верхний слой в таких системах состоит всего из одной – главной функции, которая перехватывает вызов системных функций ОС и переадресует их сервисным функциям нижележащего слоя. Эти сервисные функции в свою очередь выполняют вызов соответствующих функций нижележащего слоя. Такие функции называют вспомогательными. Концептуальная схема монолитной ОС показана на рис. 1.6.

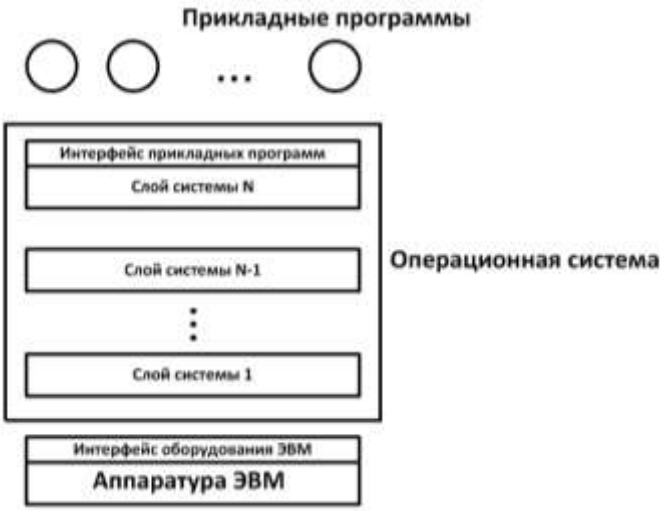

Рис. 1.5. Слои, абстрактные машины

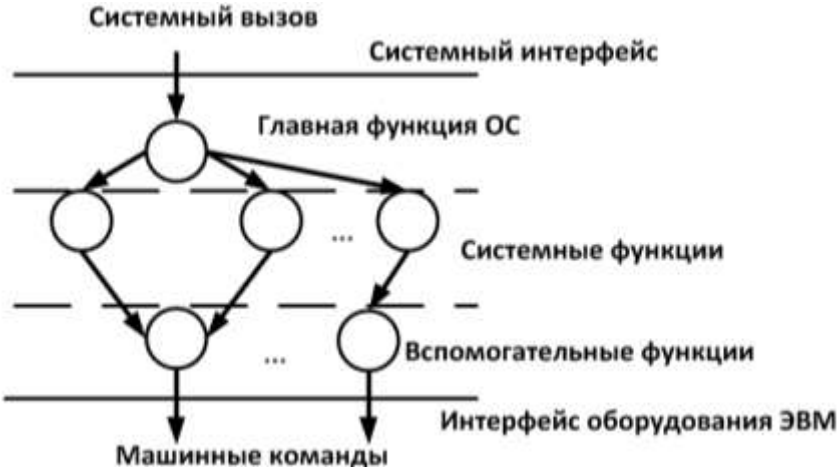

Рис. 1.6. Концепция монолитной ОС

Микроядерная архитектура

При таком подходе в привилегированном режиме работает только очень небольшая часть ОС, называемая ***микроядром***. Концепция микроядерной архитектуры показана на рис. 1.7.

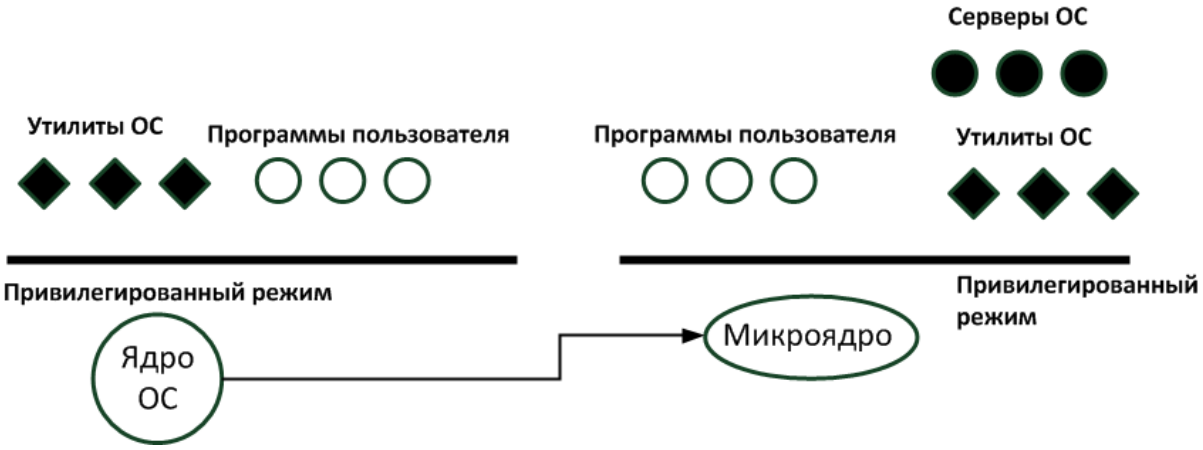

Рис. 1.7. Микроядерная архитектура

В идеальном случае микроядро может состоять только из средств передачи сообщений, средств взаимодействия с аппаратурой, средств доступа к механизмам привилегированной защиты.

Временная диаграмма системного вызова для микроядерной архитектуры показана рис. 1.8.

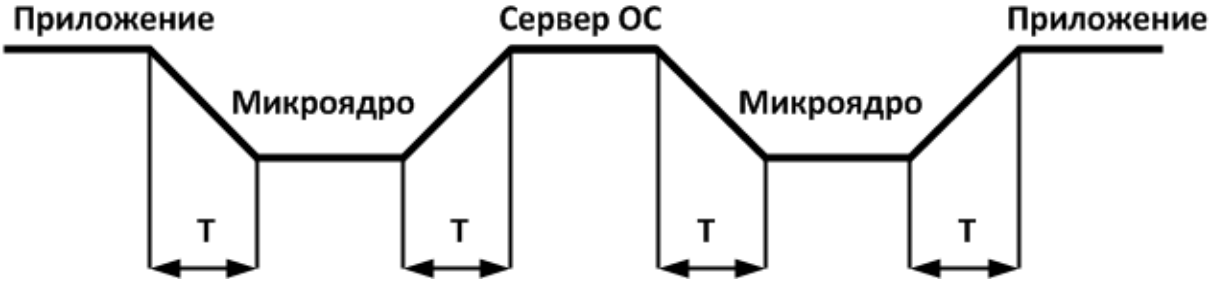

Рис. 1.8. Временная диаграмма системного вызова для микроядерной архитектуры

Системный вызов выполняется в два этапа, так как в микро ядерной архитектуре для взаимодействия с ядром ОС используется набор серверов, работающих в пользовательском режиме. На рис. 1.9 представлена схема организации микроядерной ОС.

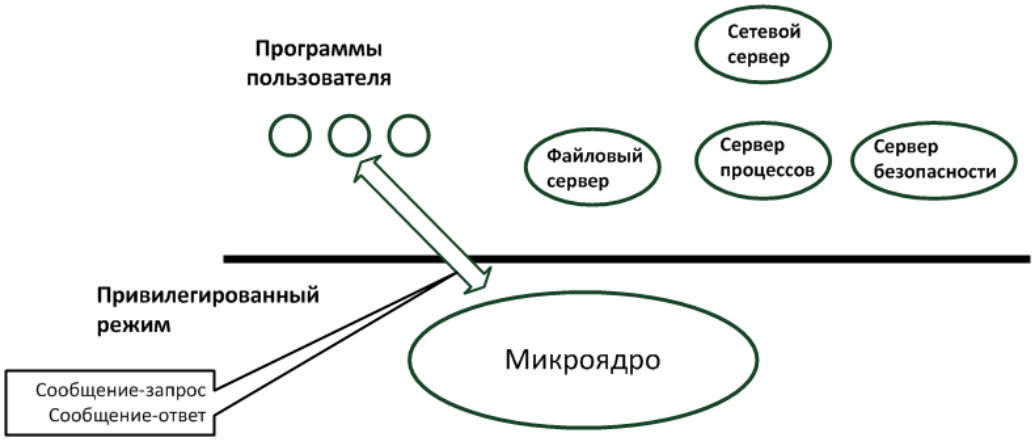

Рис. 1.9. Ядро ОС и сервера

Прикладная среда

Прикладная среда – модель окружения операционной системы, обеспечивающего предоставление разнообразных интерфейсов. Интерфейсы реализуется путем использования прикладного программного обеспечения для удовлетворения потребностей пользователя вычислительной системы. Перенос прикладного программного обеспечения из одной ОС в другую требует решения проблемы совместимости программ. При этом различают:

- совместимость на двоичном уровне;
- совместимость на уровне исходных текстов.

Двоичная совместимость достигается в том случае, когда можно взять исполняемую программу и запустить ее на выполнение в среде другой ОС.

Совместимость на уровне исходных текстов требует наличия соответствующего компилятора в составе программного обеспечения компьютера, на котором предполагается выполнять данное приложение.

Требуется перекомпиляция имеющихся исходных текстов в новый исполняемый модуль. Для решения проблемы совместимости используют различные подходы.

Транслятор системных вызовов

Это совместимость на уровне библиотек и системных вызовов. Транслятор заменяет системные вызовы приложения на эквивалент-

ные системные вызовы ОС данной ЭВМ. Концепция транслятора представлена на рис. 1.10.

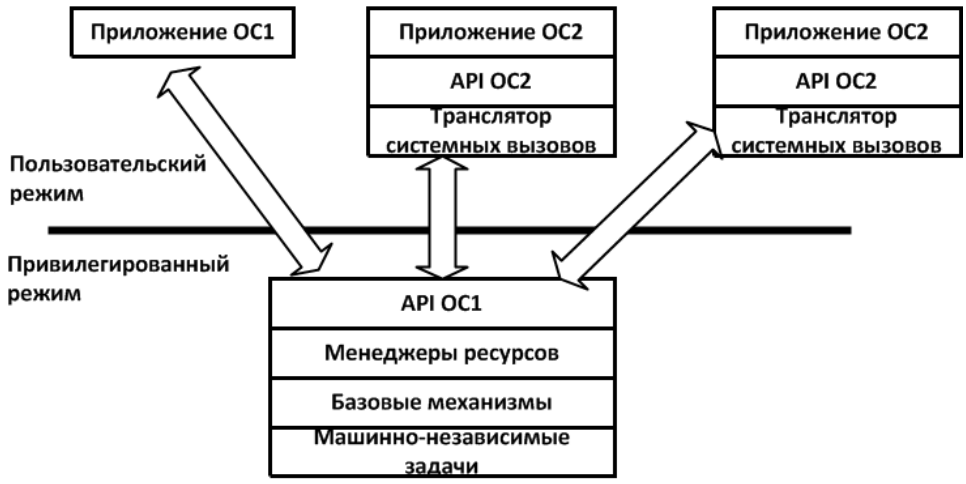

Рис. 1.10. Трансляция системных вызовов

Мульти вариантный API

При таком подходе в состав ядра ОС включает несколько версий API для различных ОС. При выполнении приложения определяется тип API и ему предоставляется соответствующий API ядра.

Концепция ядра с несколькими вариантами API представлена на рис. 1.11.

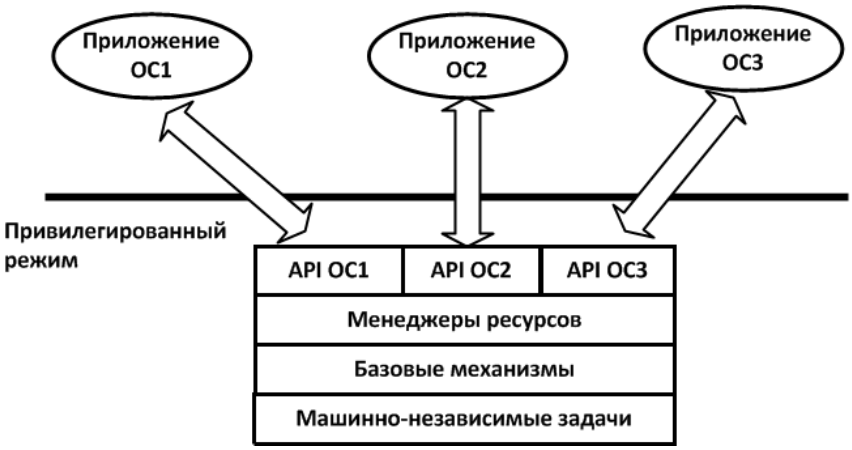

Рис. 1.11. Ядро с несколькими API

Прикладные среды – серверы

Такой подход основан на использовании концепции микроядерной архитектуры. В состав ОС вводят набор серверов, каждый такой

сервер – прикладная среда для программ из других ОС. Концепция серверов представлена на рис. 1.12.

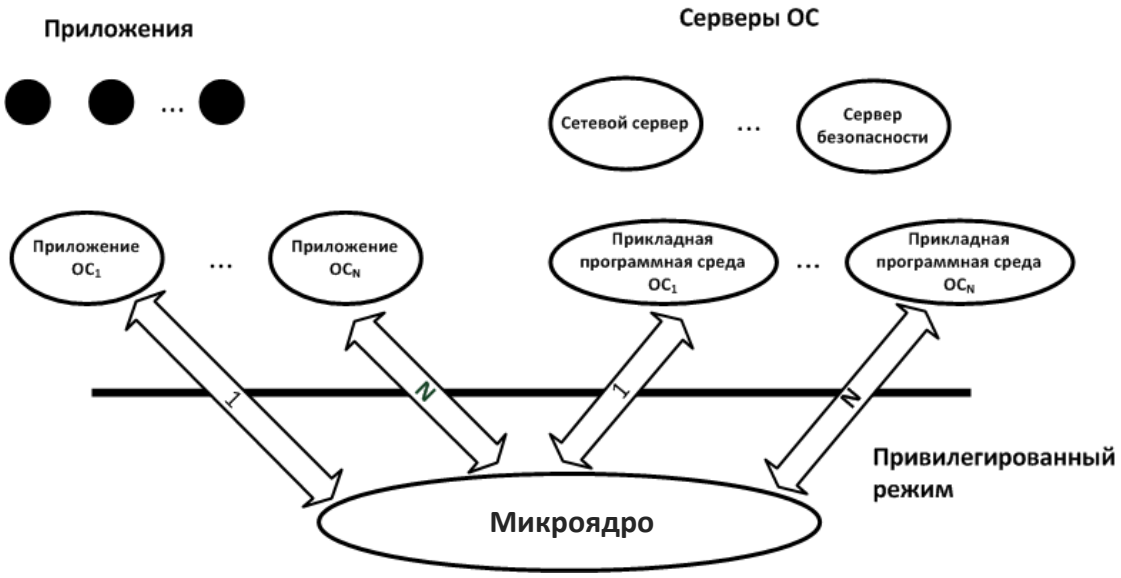

Рис. 1.12. Микроядерная архитектура и прикладные среды

Оценка эффективности работы ОС

Оценить эффективность работы ОС можно на основе следующих показателей: пропускная способность, удобство работы, реактивность.

Пропускная способность: количество задач, выполняемых системой в единицу времени.

Удобство работы: пользователь может интерактивно работать с несколькими программами на одной ЭВМ.

Реактивность системы: возможность системы выдерживать заранее заданные интервалы времени между запуском программы и получением результата.

С точки зрения эффективности использования ОС можно выделить системы с пакетной обработкой, системы с разделением времени, системы реального времени.

Системы пакетной обработки обеспечивают минимизацию времени простоя устройств компьютера и процессора. При этом формируется очередь заданий. Если текущее задание требует данных, то оно блокируется до тех пор, пока не будут получены данные. Процессор переключается на задание, которое закончило ввод и готово к исполнению.

Системы с разделением времени обеспечивают интерактивную работу пользователя сразу с несколькими приложениями. Всем приложениям выделяется квант времени по истечении кванта времени происходит переход к следующему приложению, а текущее приложение, если не закончило свою работу, блокируется и ставится в очередь. Затем при очередном кванте времени оно извлекается из очереди для дальнейшей работы.

Системы реального времени используются для управления техническими объектами или процессами. Управляющая объектом программа должна быть выполнена за предельно допустимое время, которое должно быть меньше времени протекания процессов в управляемом объекте.

Контрольные вопросы и задания

1. Дайте характеристику ОС как системной программе.
2. Какие базовые компоненты можно выделить в составе ОС?
3. Как связана ОС с аппаратным обеспечением ЭВМ?
4. В чем разница между пользовательским режимом и режимом ядра?
5. Какие уровни можно выделить в составе ядра ОС?
6. Какие архитектурные концепции используются при построении современных ОС?
7. Сравните архитектуру ОС с монолитным ядром и микроядром.
8. Что такое прикладная среда ОС?
9. Для чего используется транслятор системных вызовов ОС?
10. Для чего служит API – интерфейс ОС?
11. Как можно классифицировать ОС с точки зрения эффективности их работы?

Лекция №2. ПРОЦЕССЫ В СРЕДЕ ОПЕРАЦИОННЫХ СИСТЕМ

Понятие процесса, планирование процессов, стратегии планирования.

Процесс это выполнение программы процессором с учетом ее окружения. Окружение программы – ***контекст процесса***. Основные компоненты контекста процесса:

- значение счетчика команд процессора ЭВМ;
- значения в регистрах, хранящих данные, адреса и состояние процессора ЭВМ;
- данные в запоминающем устройстве;
- исполняемый программный код.

Один процесс может порождать другой процесс: родительский процесс и дочерний процессы. В текущий момент времени в системе с одним процессором может быть активным только один процесс.

Идентификация процессов в среде ОС выполняется на основе дескрипторов.

Дескриптор процесса состоит из [1]:

- идентификатора процесса – кода;
- состояния процесса;
- данных о степени привилегированности процесса;
- адреса кодового сегмента программы.

Процесс, в своем жизненном цикле, проходит ряд состояний, показанных на рис. 2.1.

Рис. 2.1. Состояния процесса

Перечислим эти состояния:

- готовность (ready): Процесс может выполняться, но процессор занят;

- исполнение (running): Процесс исполняется процессором;
- блокировка (blocked), ожидание (idle): Процесс ожидает внешнего события, либо процесс обработан или прерван.

Заметим, что процесс может перейти в состояние «Исполнение» только из состояния «Готовность».

Управление процессами

Управление процессами в среде ОС реализует специальная подсистема «Планировщик». Планировщик – подсистема ОС: управляет порядком выделения центрального процессорного устройства (ЦПУ) ЭВМ-процессам. Для состояний процесса ready и blocked формируются очереди.

Планировщик должен использовать определенную стратегию, что бы принять решение о том, какой процесс будет выполняться ЦПУ следующим. Другая задача – определить, с каким периодом времени будет использоваться ЦПУ.

Цель любых стратегий планирования – обеспечить:

- максимальную загрузку ЦПУ;
- высокую пропускную способность (число процессов, обработанных в единицу времени);
- минимизацию времени обработки для каждого процесса;
- минимизацию времени отклика – обслуживание всех процессов в системе.

При этом можно выделить три вида планирования обработки процессов: долгосрочное, краткосрочное и среднесрочное.

Долгосрочное планирование процессов. Оно отвечает за порождение новых процессов в системе, определяя ее степень мультипрограммирования, т.е. количество процессов, одновременно находящихся в системе.

Краткосрочное планирование – планирование использования процессора.

Среднесрочное планирование – когда и какой из процессов нужно перекачать на диск или вернуть обратно – свопинг.

Параметры процесса

Для управления процессами необходимо использовать ряд параметров процесса. Параметры процесса бывают статические и динамические.

Статическими параметрами процесса являются:

- размер оперативной памяти, необходимый для успешного выполнения процесса;
- максимальное количество памяти на диске для осуществления свопинга;
- количество подключенных устройств ввода-вывода.

Динамические параметры системы описывают количество свободных ресурсов на данный момент.

Кроме перечисленных параметров, важную роль играют временные характеристики процесса. Работу процесса, порожденного запуском программы, можно подразделить на ряд временных интервалов CPU burst и I/O burst. Временной интервал CPU burst – это время, которое процесс затрачивает на обработку данных, I/O burst – это время, затрачиваемое процессом на выполнение операций ввода – вывода. На рис. 2.2 показан вариант распределения временных интервалов для процесса [1].

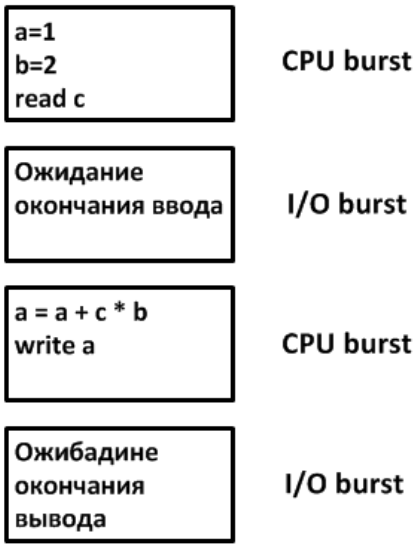

Рис. 2.2. Распределение временных интервалов

Стратегии планирования

Современные стратегии планирования подразделяют на вытесняющие и невытесняющие стратегии.

Preemptive (***вытесняющие***) стратегии: активный процесс лишают доступа к ЦПУ на некоторое время либо полностью удаляют из системы.

Nonpreemtive (***невытесняющие***) стратегии: активный процесс работает до тех пор, пока он не закончиться или не перейдет в состояние блокировки (например, когда он ожидает доступа к устройству ввода-вывода).

В современных ОС используются преимущественно вытесняющие стратегии. Невытесняющие стратегии используются в ОС специального назначения.

Предпосылкой для использования вытесняющих стратегий является наличие в ЭВМ аппаратного таймера: после истечения периода времени t (например, 50 мс) планировщик получает вследствие прерывания доступ к ЦПУ.

Далее рассмотрим базовые стратегии планирования процессов.

Очередь процессов

Выбор нового процесса для исполнения осуществляется из начала очереди процессов в состоянии готовности. Очередь подобного типа имеет в системном программировании специальное наименование – FIFO, сокращение от First In, First Out (первым вошел, первым вышел).

Такой алгоритм выбора процесса осуществляет невытесняющее планирование.

Управление с разделением времени

Каждый процесс на период времени T_0 получает доступ к ЦПУ. По истечении времени T_0, если он не закончен, то он снова поступает в очередь ожидания.

Такой алгоритм выбора процесса осуществляет вытесняющее планирование. На рис. 2.3 показан пример вытесняющего планирования на примере четырех процессов. Четвертый процесс не успел за-

вершится на CPU (Central Process Unit – ЦПУ) и он переводится в состояние ожидания. Из состояния готовности выбирается третий процесс, на очереди перехода в состояние готовности второй процесс.

Такая стратегия планирования получила условное обозначение RR (Round Robin – карусель) [4, 10].

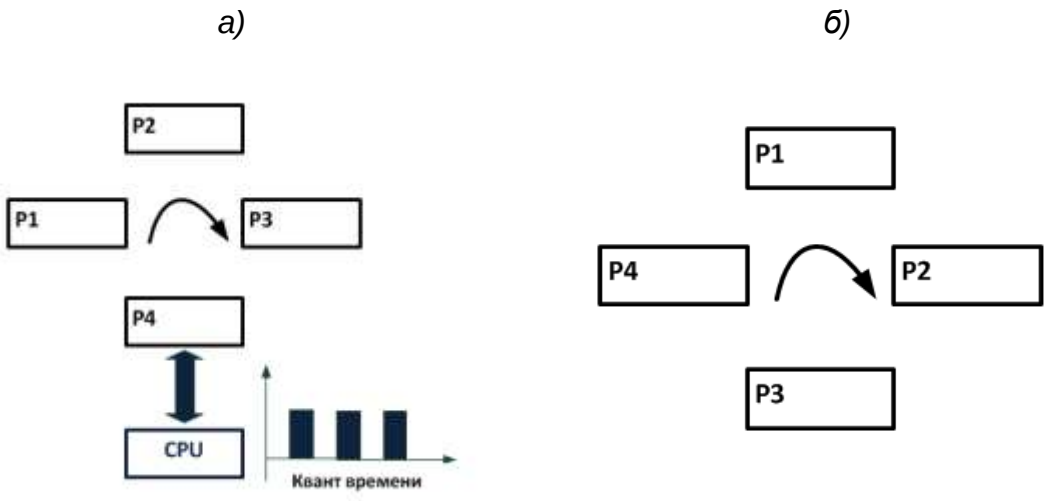

Рис. 2.3. Разделение времени: а – фаза 1; б – фаза 2

Алгоритм Shortest-Job-First (SJF)

Из очереди процессов в состоянии готовности выбирается тот процесс, для выполнения которого нужно использовать минимальное время работы ЦПУ. Для реализации алгоритма нужно знать CPU burst для процессов, находящихся в состоянии «Готовность». Выбирается для исполнения процесс с минимальной длительностью CPU burst.

Если же таких процессов два или больше, то для выбора одного из них используется алгоритм FIFO.

Shortest Remaining Time First (SRTF)

После каждого прерывания некоторый процесс получает доступ к ЦПУ, если оставшееся время для его завершения минимально.

Алгоритм Highest Priority First (HPF)

Каждый процесс i получает приоритет p_i. Процесс занимает ЦПУ до тех пор, пока он не закончится, либо пока не будет блокирован, ли-

бо при прерывании новый процесс с приоритетом $p_j > p_i$ окажется в состоянии готовности.

Гарантированное планирование

Рассмотрим вычислительную систему, в которой N пользователей. Каждый пользователь должен в своем распоряжении иметь ~1/N часть процессорного времени.

Для каждого пользователя с номером i введем две величины:

- T_i – время нахождения пользователя в системе;
- t_i – суммарное процессорное время уже выделенное всем его процессам в течение сеанса.

Для процессов вычисляется значение коэффициента справедливости [1]:

$$K = t_i N / T_i.$$

Если T_i начинает расти, то коэффициент справедливости уменьшается. Процессы, у которых K стремится к нулю, долго находятся в системе и должны быть либо обслужены в первую очередь, либо выгружены из системы.

Многоуровневые очереди (Multilevel Queue)

Используются в вычислительных системах, в которых процессы могут быть рассортированы по разным группам в соответствии с приоритетом. В вычислительных системах обычно высшим приоритетом является приоритет равный нолю. Это системные процессы. На рис. 2.4 показана схема обработки процессов в пяти группах. Для обработки процессов в группах используется стратегия планирования RR [1].

Многоуровневые очереди с обратной связью

Процесс не приписан к определенной очереди, а может мигрировать из одной очереди в другую в зависимости от своего поведения. Перевод процессов из одной группы в другую может выполнять администратор вычислительной системы супервизор [1].

Рис. 2.4. Группирование процессов

Контрольные вопросы и задания

1. Что такое контекст процесса?
2. Какие составные элементы можно выделить в составе дескриптора процесса?
3. Перечислите состояния процесса, дайте им характеристику.
4. Что понимается под стратегией планирования процессов?
5. Дайте характеристику правилам планирования процессов ОС.
6. Перечислите параметры процесса.
7. Какие стратегии планирования использует планировщик ОС?
8. Как выполняется планирование процессов в режиме разделения времени?
9. Как выполняется приоритетное управление процессами?
10. Как реализуется алгоритм гарантированного планирования?
11. Как выполняется управление процессами на основе многоуровневых очередей?
12. Дайте краткую характеристику алгоритмам планирования FIFO, SJF, SRTF.

Лекция №3. СИНХРОНИЗАЦИЯ ПРОЦЕССОВ

Понятие синхронизации, семафоры, мониторы, сообщения, тупики.

При обработке процессов возникает проблема их согласованной работы. Это ситуация может возникнуть, когда процесс требует для

своей работы данные, подготовленные другим процессом. Другая проблема: один процесс ждет завершения фазы работы другого процесса для дальнейшего параллельного исполнения.

Часто проблема синхронизации возникает, когда два процесса и более одновременно используют один и тот же ресурс. Например, изменяют общий файл данных на диске.

Для организации синхронизации процессов используют базовые механизмы синхронизации:

- семафор;
- монитор;
- сообщения.

Семафоры

Семафор был предложен нидерландским ученым Э.Б. Дейкстрой в 1965 году. ***Семафор*** – переменная S, для которой определено две атомарных операции P (открытие) и V (закрытие) [4, 10].

Базовая реализация:

```
P(S): [
while (S<=0) {/* нет активности
процесс можно прервать */}; S=S-1
]
V(S): [S=S+1]
```

Сначала инициализируется S=1

```
P(S);
Критическая секция;
V(S);
```

Здесь критическая секция – участок программного кода, подлежащий исполнению только одним процессом. Если процесс вошел в критическую секцию, то другой процесс не может получить доступ к обрабатываемому ресурсу до тех пор, пока процесс, его использующий, не покинет критическую секцию.

Семафоры нашли широкое применение при разработке многопроцессных программ в среде таких операционных систем, как UNIX и Linux. В качестве недостатка использования семафоров отмечается, что процесс, который выполняет операцию P(S), требует много процессорного времени.

Для эффективного использования семафора S объявляют как объект со свойствами:

- S.ctr (значение семафора);
- S.list (список процессов в состоянии ожидания).

Тогда псевдокод использования семафора примет вид:

```
P(S): [S.ctr=S.ctr-1;
if (S.ctr<0){
 put pid in S.list;sleep()}]
V(S): [S.ctr=S.ctr+1;
if (S.ctr<=0){
get pid from S.list wakeup(pid)}
]
```

Здесь pid – код процесса. Системный вызов sleep() приводит к тому, что pid процесса блокируется и таким образом нет надобности тратить процессорное время. Процесс будет разблокирован с помощью системного вызова wakeup(pid).

Мониторы

В 1974 году Хоаром (Чарльз Хоар) был предложен механизм синхронизации еще более высокого уровня, чем семафоры, получивший название ***мониторов***.

На абстрактном уровне можно описать структуру монитора следующим образом [1]:

```
monitor monitor_name {
    описание внутренних переменных;
void m1(...){...   }
void m2(...){...   }
...
void mn(...){...   }
{блок инициализации внутренних переменных; }
}
```

Здесь функции m_1,..., m_n представляют собой функции-методы монитора.

Блок инициализации внутренних переменных содержит операции, которые выполняются один и только один раз: при создании монитора или при самом первом вызове какой-либо функции-метода до ее исполнения.

Важной особенностью мониторов является то, что в любой момент времени только один процесс может быть активен, т.е. находиться в состоянии «Готовность» или «Исполнение» внутри данного монитора.

Концепция монитора используется в программах, написанных для виртуальной машины Java.

Сообщения

Синхронизовать работу процессов можно на основе механизма уведомления процессов о состоянии друг друга на основе механизма сообщений.

При этом для прямой и непрямой адресации сообщений достаточно двух примитивов, чтобы описать передачу сообщений по линии связи – send и receive.

Реализация примитивов:

- send(P, message) – послать сообщение message процессу P;
- receive(Q, message) – получить сообщение message от процесса Q.

В случае непрямой адресации используют буфер – почтовый ящик:

- send(A, message) – послать сообщение message в почтовый ящик A;
- receive(A, message) – получить сообщение message из почтового ящика A.

Эквивалентность семафоров, мониторов и сообщений

В рамках одной вычислительной системы, когда процессы имеют возможность использовать разделяемую память, все они эквивалентны.

Это означает, что любые два из предложенных механизмов могут быть реализованы на базе третьего, оставшегося, механизма [1]. Например, семафоры и мониторы можно реализовать на базе механизма сообщений.

Тупики

В случае, когда требуемый ресурс удерживается другим ожидающим процессом, первый процесс не сможет сменить свое состояние. Такая ситуация называется ***тупиком*** (deadlock).

В мультипрограммной системе процесс находится в состоянии тупика, если он ожидает события, которое никогда не произойдет.

Состояние тупика – блокировки для двух процессов – показано на рис. 3.1.

Направление стрелки от процесса к ресурсу означает его запрос, а от ресурса к процессу – его выделение процессу.

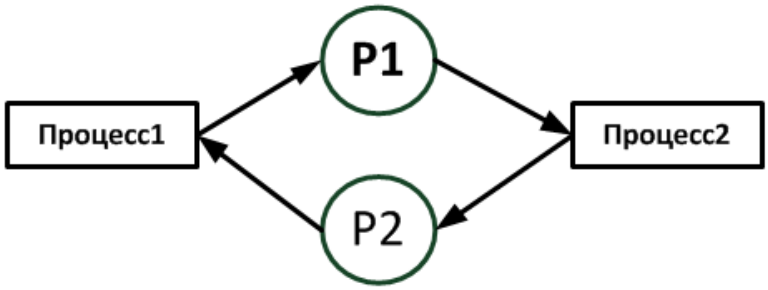

Рис. 3.1. Состояние тупика – блокировки: P1 и P2 – ресурсы

Рассмотрим следующую ситуацию. Пусть данная ЭВМ обладает одним печатающим устройством (ПУ). На диске ЭВМ имеется файл D. Процессор ЭВМ выполняет две программы – процесса A и B. Процесс A делает попытку прочитать файл D и распечатать его на ПУ. Процесс B получил доступ к ПУ и делает попытку распечатать файл, в результате получим схему запроса процессами ресурсов, показанную в табл. 3.1.

Таблица 3.1

Взаимная блокировка

Процесс A	Процесс B
Запрос D	Запрос ПУ
Запрос ПУ	Запрос D
Печать файла	Печать файла

Когда процесс A запрашивает ПУ, то ОС сообщает, что оно выделено процессу B. Когда процесс B запрашивает файл D, ОС сообщает, что он занят процессом A.

Условия возникновения тупиков были сформулированы в 1970 г. Коффманом, Элфиком и Шошани.

- условие взаимоисключения (Mutual exclusion). Одновременно использовать ресурс может только один процесс;
- условие ожидания ресурсов (Hold and wait). Процессы удерживают ресурсы, уже выделенные им, и могут запрашивать другие ресурсы;
- условия закрепления ресуросов (No preemtion). Ресурс, выделенный ранее, не может быть принудительно забран у процесса. Освобождены ресурсы могут быть только процессом, который их удерживает;
- условие кругового ожидания (Circular wait). Существует кольцевая цепь процессов, в которой каждый процесс ждет доступа к ресурсу, удерживаемому другим процессом цепи.

Борьба с тупиками

Основные направления борьбы с тупиками:

- игнорирование проблемы в целом;
- предотвращение тупиков;
- обнаружение тупиков;
- восстановление после тупиков;
- алгоритм банкира.

Среди перечисленных наибольший интерес представляет собой алгоритм, предложенный Э.Б. Дейкстрой, который базируется на так называемых безопасных или надежных состояниях (safe state).

Безопасное состояние – это такое состояние, для которого имеется по крайней мере одна последовательность событий, которая не приведет к взаимоблокировке.

Рассмотрим пример. Пусть у ОС в наличии n устройств, например, дисков. ОС принимает запрос от пользовательского процесса, если его максимальная потребность не превышает n.

Пользователь гарантирует, что если ОС в состоянии удовлетворить его запрос, то все устройства будут возвращены системе в течение конечного времени.

Текущее состояние системы называется надежным, если ОС может обеспечить всем процессам их выполнение в течение конечного времени.

В соответствии с алгоритмом банкира выделение устройств возможно, только если состояние системы остается надежным.

Пусть в системе 3 пользователями и 11 устройствами [1]:

- 9 устройств задействовано;
- 2 имеется в резерве.

Пусть текущая ситуация такова, как в табл. 3.2.

Таблица 3.2

Пользователи – ресурсы

Пользователи	Максимальная потребность в ресурсах	Выделенное пользователям количество ресурсов
Первый	9	6
Второй	10	2
Третий	3	1

Вначале ОС следует удовлетворить запросы третьего пользователя. Затем дождаться, когда он закончит работу и освободит свои три устройства. Затем можно обслужить первого и второго пользователей.

В современных ОС используется следующий, достаточно эффективный простой алгоритм. В его основе лежит следующее правило:

Если процесс не может получить сразу все ресурсы, которые необходимы для его работы, то он должен вернуть все захваченные на данный момент ресурсы и попытаться через некоторое время повторить попытку захвата требуемых ресурсов.

Контрольные вопросы и задания

1. Что такое синхронизация процессов?
2. Перечислите базовые механизмы синхронизации процессов.
3. Что такое критическая секция процесса?
4. Опишите алгоритм использования семафора Дейкстры.
5. Опишите объектную модель использования семафора.
6. Дайте характеристику монитору Хора.
7. Как реализуется взаимодействие между процессами на основе механизма сообщений?

8. Опишите состояние блокировки процессов.

9. Какие особенности ОС могут приводить к состоянию блокировки процессов?

10. Опишите алгоритм Дейкстры борьбы с блокировками процессов.

Лекция №4. УПРАВЛЕНИЕ РАСПРЕДЕЛЕНИЕМ ОПЕРАТИВНОЙ ПАМЯТИ

Иерархия оперативной памяти, адреса памяти, распределение памяти на программном уровне, распределение памяти на аппаратном уровне.

Современные ЭВМ оснащаются двумя видами оперативной памяти. Различают основную (главную, оперативную, физическую) и вторичную (внешнюю) память.

Память современных ЭВМ можно классифицировать по иерархическому принципу. Иерархия оперативной памяти показана на рис. 4.1. Самая быстрая память – регистры процессора, но это так же самая дорогая память ЭВМ. Внизу иерархии располагается память на магнитных носителях.

В течение ограниченного отрезка программа способна работать с небольшим набором адресов памяти. Это свойство программного обеспечения принято называть принципом локальности или локализации обращений к ячейкам оперативной памяти [10].

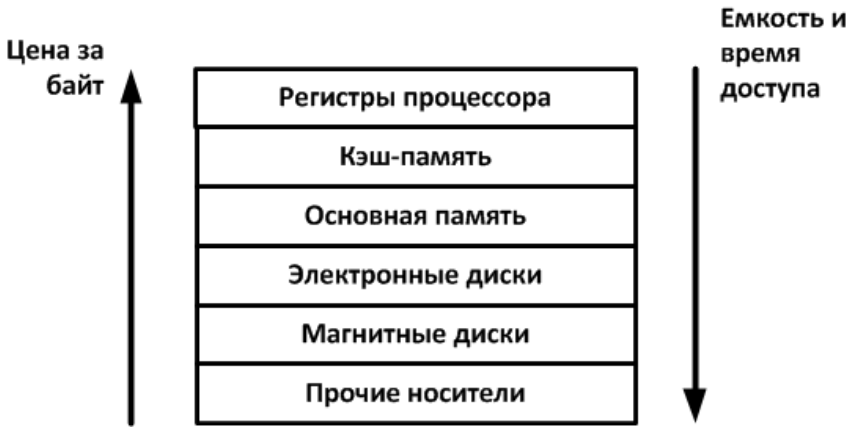

Рис. 4.1. Иерархия оперативной памяти

Адресация оперативной памяти

Память современных ЭВМ это последовательность нумерованных ячеек определенной разрядности. Номер ячейки – целое число называется адресом. Различают логическое адресное пространство и физическое адресное пространство.

Максимальный размер логического адресного пространства обычно определяется разрядностью процессора (например, для 32 разрядного процессора он равен 2^{32}). Физическое адресное пространство – фактически установленный объем оперативной памяти на данной ЭВМ.

Связывание адресов

Для обращения к ячейкам памяти их адреса должны быть связаны с программным кодом. При этом говорят о логическом адресе – адрес, полученный в процессе перевода текста программы в машинный код.

Логический адрес должен быть отображен в физический (реальный адрес памяти). Связывание может быть выполнено разными способами.

Этап компиляции (Compile time). Если на стадии компиляции известно точное место размещения процесса в памяти, тогда непосредственно генерируются физические адреса. При изменении стартового адреса программы необходимо перекомпилировать ее код.

Этап выполнения (Execution time). Если процесс может быть перемещен во время выполнения из одной области памяти в другую, то связывание откладывается до стадии выполнения.

Этап загрузки (Load time). Если информация о размещении программы на стадии компиляции отсутствует, то компилятор генерирует перемещаемый код. В этом случае окончательное связывание откладывается до момента загрузки. Если стартовый адрес меняется, то нужно перезагрузить код с учетом измененной величины адреса.

Функции системы управления памятью

Подсистема управления памятью ОС должна решать следующие задачи:

- **отображение** адресного пространства процесса на конкретные области физической памяти;
- **распределение** памяти между конкурирующими процессами;
- **контроль доступа** к адресным пространствам процессов;
- **выгрузка процессов** (целиком или частично) во внешнюю память, когда в оперативной памяти недостаточно места;
- **учет** свободной и занятой памяти.

Схемы распределения памяти

Рассмотрим различные схемы распределения памяти. Схема с фиксированными разделами. Такая схема представлена в виде двух вариантов, показанных на рис. 4.2 и 4.3. Это схемы с общей очередью процессов и с отдельными очередями процессов.

Недостатки такого распределения памяти:

- число одновременно выполняемых процессов ограничено числом разделов;
- наличие внутренней фрагментации – потеря части памяти, выделенной процессу, но не используемой им.

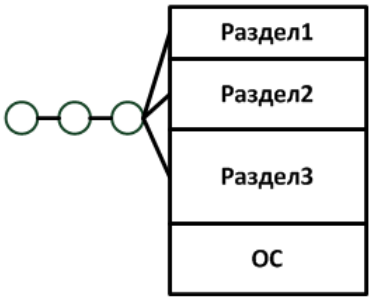

Рис. 4.2. Фиксированные разделы (общая очередь процессов)

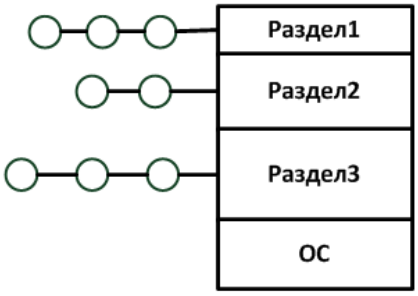

Рис. 4.3. Фиксированные разделы (очереди процессов закреплены за разделами)

Оверлейная организация памяти (overlay)

Принцип управления – держать в памяти только те инструкции программы, которые нужны в данный момент. Код программы делится на блоки. Блок кода – узел дерева (ветвь). Коды ветвей оверлейной структуры программы находятся на диске как абсолютные образы памяти. Они считываются драйвером оверлеев при необходимости.

Для описания оверлейной структуры используется специальный язык ODL (overlay description language). Совокупность файлов исполняемой программы дополняется файлом с расширением odl, описывающим дерево вызовов внутри программы.

Рассмотри некую программу A, при выполнении которой выполняется два вызова процедур B и C (Call B и Call C). Структура программы показана на рис. 4.4.

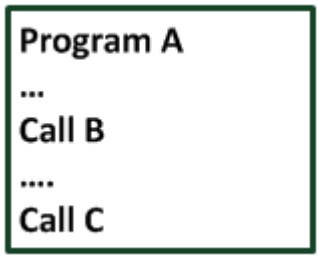

Рис. 4.4. Вызов процедур

В свою очередь процедура C (Sub C) содержит обращение к двум процедурам D и E (Call D и Call E), так как это показано на рис. 4.5.

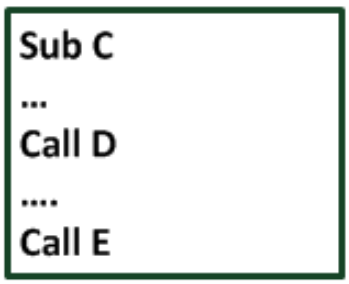

Рис. 4.5. Вызов процедур внутри подпрограммы

Тогда дерево оверлейных вызовов будет иметь вид, показанный на рис. 4.6.

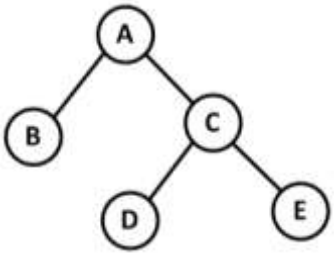

Рис. 4.6. Дерево оверлейных вызовов

Динамическое распределение памяти

Такая технология используется в системах с разделением времени. Память не в состоянии содержать все пользовательские процессы. В этом случае используют свопинг (swapping) – перемещение процессов из главной памяти на диск – долговременную память и обратно целиком.

Выгруженный процесс может быть возвращен в то же самое адресное пространство или в другое. Это ограничение диктуется методом связывания. Для схемы связывания на этапе выполнения можно загрузить процесс в другое место памяти.

Переменные разделы

Вновь поступающей задаче выделяется строго необходимое количество памяти. После выгрузки процесса память временно освобождается. По истечении некоторого времени память представляет собой переменное число разделов разного размера. Смежные свободные участки могут быть объединены.

Рассмотрим пример. Пусть в распоряжении ОС имеется три раздела оперативной памяти, так как это показано на рис. 4.7.

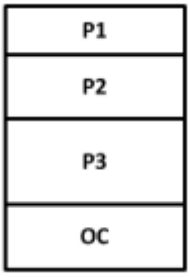

Рис. 4.7. Начальное состояние разделов памяти

Затем в первый и третий разделы предоставляется исполняемым процессам (рис. 4.8).

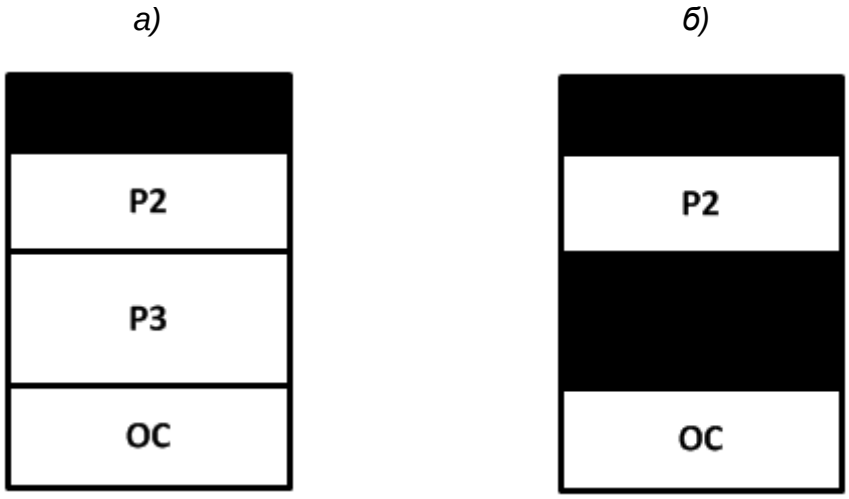

Рис. 4.8. Состояние памяти после загрузки двух процессов: а – занят первый раздел; б – занят третий раздел

Затем процесс из третьего раздела был выгружен и в него был загружен новый процесс. Из рис. 4.9 видно, что выбор раздела был сделан с «запасом» и в нем осталось невостребованное свободное место.

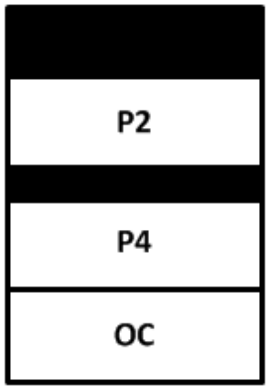

Рис. 4.9. Третий раздел занят не полностью

Таким образом, возникает проблема выбора стратегии загрузки процессов в разделы [1].

Стратегии выбора раздела

Стратегия первого подходящего (First fit). Процесс помещается в первый подходящий по размеру раздел.

Стратегия наиболее подходящего (Best fit). Процесс помещается в тот раздел, где после его загрузки останется меньше всего свободного места.

Стратегия наименее подходящего (Worst fit). При помещении в самый большой раздел в нем остается достаточно места для возможного размещения еще одного процесса.

Рассмотренные выше технологии выделения оперативной памяти используются на уровне программного обеспечения. Так, технология выделения разделов различного объема используется при написании программ на языке Си и Си++ с динамическим размещением их объектов в памяти.

Далее рассмотрим «реальные» технологии выделения оперативной памяти процессам на аппаратном уровне.

Сегментная адресация

Оперативная память ЭВМ подразделяется на блоки, такие блоки называются ***сегментами***. Сегменты могут иметь постоянный или переменный размер. Тогда логическое адресное пространство ЭВМ – набор сегментов [4, 10].

Каждый сегмент имеет имя, размер и другие параметры (уровень привилегий, разрешенные виды обращений, флаги присутствия в оперативной памяти и т.д.).

Схема сегментной адресации представлена на рис. 4.10.

Организация доступа к памяти ЭВМ требует определения физического адреса – номера ячейки оперативной памяти. Программа использует логический адрес, который состоит из двух частей: номера сегмента и смещения к ячейке памяти внутри сегмента. Номер сегмента – его идентификатор, дескриптор. Учет сегментов ведется с помощью таблицы дескрипторов. Каждая строка таблицы – запись, соответствующая определенному сегменту. В этой записи хранятся характеристики сегмента (атрибуты) и адрес начала сегмента в оперативной памяти.

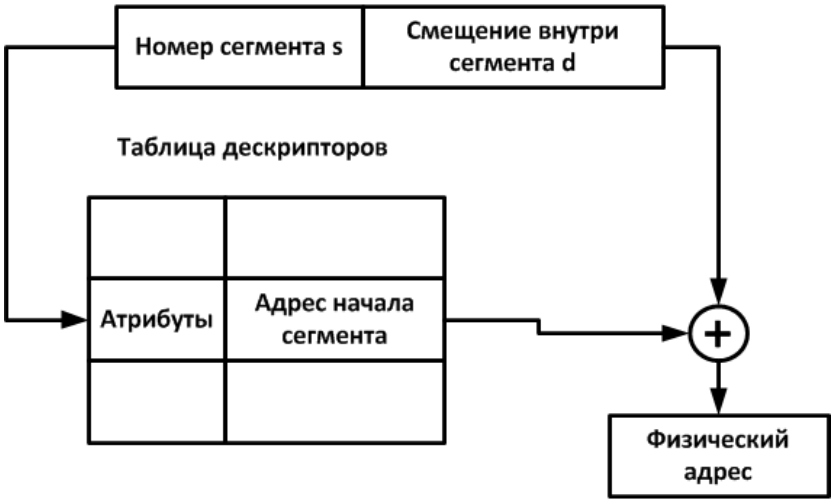

Рис. 4.10. Сегментная адресация

Страничная память

Описанная выше схема недостаточно эффективно использует память ЭВМ, более выгодно не размещать процесс одним непрерывным блоком. Вместо сегментов используют набор страниц. Страница – блок памяти фиксированной длины небольшого размера. Так, у процессоров семейства Intel – размер блока 4 Кб.

Логический адрес в страничной системе – упорядоченная пара (p, d), где p – номер страницы; d – смещение в рамках страницы p к ячейке памяти.

Схема страничной адресации представлена на рис. 4.11 [4, 10].

Рис. 4.11. Страничная адресация

Учет страниц ведется с помощью таблицы. Каждая строка таблица – это запись с атрибутами страницы и номером – адресом страницы.

Сегментно-страничная адресация

Сегментно-страничная и страничная организация памяти позволяет организовать совместное использование одних и тех же данных и программного кода разными задачами.

Для этого различные логические блоки памяти разных процессов отображают в один и тот же блок физической памяти, где размещается разделяемый фрагмент кода или данных.

Схема сегментно-страничной адресации показана на рис. 4.12 [4, 10].

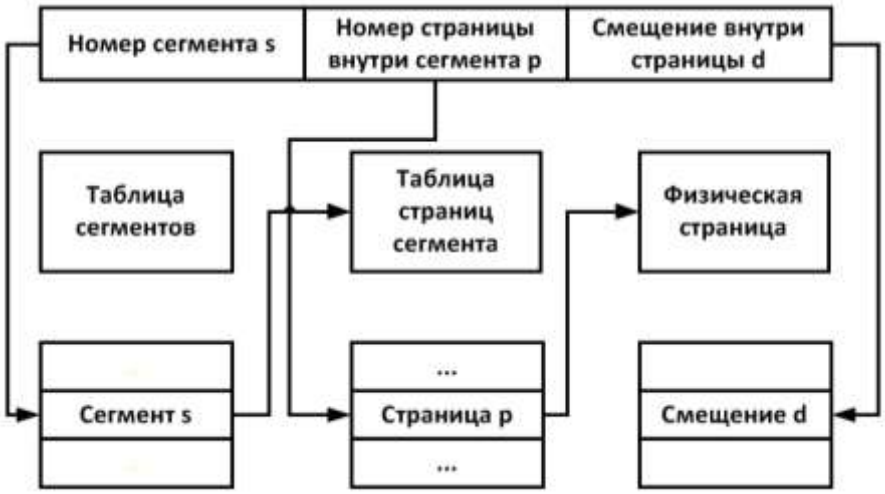

Рис. 4.12. Сегментно-страничная адресация

Логический адрес состоит из трех полей: номера сегмента, номера страницы внутри сегмента и смещения к адресуемой ячейке страницы. Для учета памяти требуется наличие таблицы сегментов и таблиц страниц сегментов.

При организации доступа к ячейке по таблице сегментов определяется его адрес – номер. Из записи для сегмента считывается номер – адрес таблицы страниц сегмента. В таблице страниц определяется номер страницы по второй составляющей адреса – p. К номеру страницы, адресу прибавляется смещение d – третья составляющая адреса.

Контрольные вопросы и задания

1. Дайте классификацию видам оперативной памяти ЭВМ.
2. Что такое физический и логический адрес ЭВМ?

3. Перечислите особенности процедуры связывания адресов для исполняемой программы.

4. Перечислите функции подсистемы управления оперативной памятью ОС.

5. Опишите программные методы выделения памяти процессам ОС.

6. Как выполняется выделение памяти процессам ОС при использовании сегментной адресации.

7. Какую структуру имеет дескриптор сегмента оперативной памяти?

8. В чем заключается разница между страничной и сегментной адресацией оперативной памяти?

9. Как выполняется выделение памяти процессам ОС при использовании страничной адресации.

10. Как выполняется выделение памяти процессам ОС при использовании сегментно-страничной адресации.

Лекция №5. ОРГАНИЗАЦИЯ ВИРТУАЛЬНОЙ ПАМЯТИ

Понятие виртуальной памяти, таблица страниц, управление виртуальной памятью, стратегии вытеснения, рабочее множество процесса, страничные демоны.

Основные понятия

Виртуальная память (Virtual Memory) в первые была реализована в 1959г. на компьютере «Атлас», разработанном в Манчестерском университете. Перечислим концепции, лежащие в основе технологии использования виртуальной памяти [1]:

- занимаемая процессом память разбивается на несколько частей – страниц;
- логический адрес (логическая страница), к которому обращается процесс, динамически транслируется в физический адрес (физическую страницу);
- когда страница, к которой обращается процесс, не находится в физической памяти, нужно организовать ее подкачку с диска;

- для контроля наличия страницы в памяти вводится специальный бит присутствия, входящий в состав атрибутов страницы в таблице страниц.

Преимущества виртуальных страниц

Программа не ограничена объемом физической памяти. Упрощается разработка программ, поскольку можно задействовать большие виртуальные пространства, не заботясь о размере используемой памяти.

Увеличение загрузки процессора и пропускной способности системы. Это обусловлено тем, что появляется возможность частичного помещения программы (процесса) в память и гибкого перераспределения памяти между программами. Как следствие в памяти ЭВМ находится больше программ пользователя.

Увеличение скорости работы программы. Так как объем ввода-вывода для выгрузки части программы на диск будет меньше, чем в варианте классического свопинга.

Архитектура виртуальной памяти

Виртуальная память и физическая память представляются состоящими из наборов блоков или страниц одинакового размера (4Кб). Виртуальные адреса делятся на страницы. Соответствующие единицы в физической памяти образуют страничные кадры.

Система поддержки страничной виртуальной памяти называется ***пейджингом***. Передача информации между памятью и диском всегда осуществляется целыми страницами.

Реализуется доступ к виртуальной памяти на основе таблицы страниц. Виртуальный адрес состоит из виртуального номера страницы и смещения. Номер записи в таблице страниц соответствует номеру виртуальной страницы.

Для контроля за состоянием страницы вводятся специальные атрибуты – флаги. Такой флаг представляет собой бит в активном или сброшенном состоянии.

Рассмотрим назначение битов:

- бит присутствия и защиты (например, 0 – read/write, 1 – read only...);
- бит модификации, который устанавливается, если содержимое страницы модифицировано, и позволяет контролировать необходимость перезаписи страницы на диск;
- бит ссылки, который помогает выделить малоиспользуемые страницы.

Для того чтобы избежать размещения в памяти огромной таблицы страниц, ее разбивают на ряд фрагментов. В оперативной памяти хранят лишь некоторые, необходимые для конкретного момента исполнения фрагменты таблицы страниц.

В силу свойства локальности адресного пространства число таких фрагментов относительно невелико.

Принципиальная схема организации многоуровневой таблицы страниц показана на рис. 5.1. Такая организация – двухуровневая с размером страниц 4 Кбайт, реализована в 32-разрядной архитектуре Intel.

Таблица, состоящая из 2^{20} строк, разбивается на 2^{10} таблиц второго уровня по 2^{10} строк. Эти таблицы второго уровня объединены в общую структуру при помощи одной таблицы первого уровня, состоящей из 2^{10} строк.

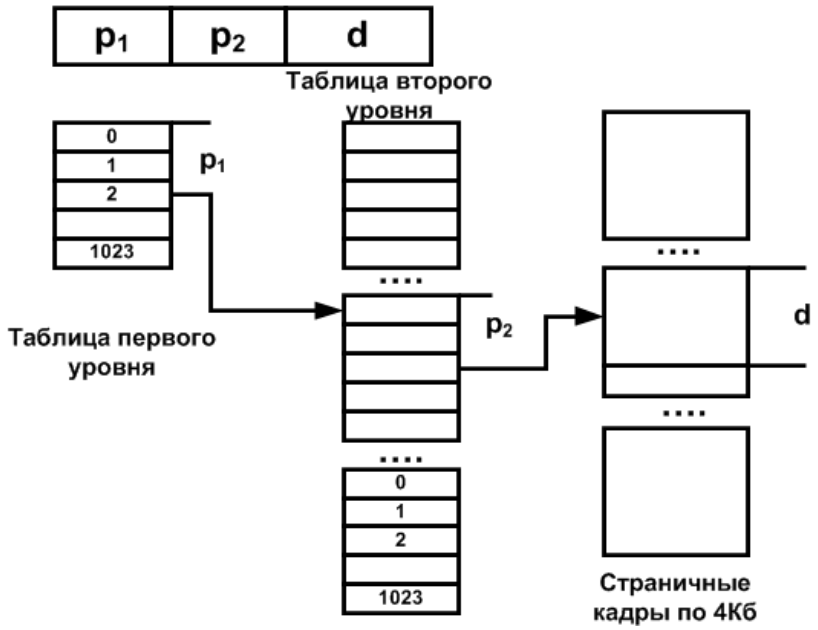

Рис. 5.1. Двухуровневая таблица страниц

Виртуальный адрес состоит из трех составляющих: p_1 – адреса записи в таблице страниц первого уровня; p_2 – адреса записи в таблице второго уровня и смещения d к ячейке памяти на физической странице.

Ассоциативная память

Поиск номера кадра, соответствующего нужной странице, в многоуровневой таблице страниц требует нескольких обращений к основной памяти, поэтому занимает много времени, и, как следствие, происходит задержка при выполнении программы.

Для ускорения доступа к памяти, при ее страничной организации, производители ЭВМ предлагают аппаратное устройство для отображения виртуальных страниц в физические без обращения к таблице страниц. Это устройство представляет собой быструю кэш-память, хранящую необходимую на данный момент часть таблицы страниц.

Это устройство называется ***ассоциативной памятью***, иногда также употребляют термин «***буфер поиска трансляции***» (Translation Lookaside Buffer – TLB) [4, 10].

В соответствии со свойством локальности большинство программ в течение некоторого промежутка времени обращаются к небольшому количеству страниц. Поэтому активно используется только небольшая часть таблицы страниц, которая может быть загружена с помощью TLB.

Страничные нарушения

Ошибки при обращении к страничной памяти называются ***страничным нарушением***. Страничное нарушение может происходить в самых разных случаях. Перечислим примеры нарушений:

- отсутствие страницы в оперативной памяти;
- попытка записи в страницу с атрибутом «только чтение»;
- при попытке чтения или записи страницы с атрибутом «только выполнение».

Время эффективного доступа к отсутствующей в оперативной памяти странице складывается из:

- обслуживания исключительной ситуации отсутствия страницы в оперативной памяти;
- чтения (подкачки) страницы из дисковой памяти;
- при недостатке места в основной памяти необходимо «вытолкнуть» одну из страниц из основной памяти во вторичную, то есть осуществить замещение страниц;
- возобновление выполнения процесса, вызвавшего данное страничное нарушение.

Стратегии управления

Подсистема ОС, выполняющая управление страничными кадрами, использует определенные стратегии. При обсуждении стратегий управления будем использовать следующие понятия:

- первичная память – оперативная память ЭВМ;
- вторичная память – долговременная, дисковая память ЭВМ.

Стратегия выборки (fetch policy) – в какой момент следует переписать страницу из вторичной памяти в первичную. Алгоритм выборки по запросу вступает в действие в тот момент, когда процесс обращается к отсутствующей странице, содержимое которой находится на диске. Его реализация заключается в загрузке страницы с диска в свободную физическую страницу и коррекции соответствующей записи таблицы страниц.

Алгоритм выборки с упреждением осуществляет опережающее чтение, то есть кроме страницы, вызвавшей исключительную ситуацию, в память также загружается несколько страниц, окружающих ее (обычно соседние страницы располагаются во внешней памяти последовательно и могут быть считаны за одно обращение к диску).

Стратегия размещения (placement policy) – в какой участок первичной памяти поместить поступающую страницу.

В системах со страничной организацией – любой свободный страничный кадр. В случае систем с сегментной организацией необходима стратегия, аналогичная стратегии с динамическим распределением памяти.

Стратегия замещения (replacement policy) – какую страницу нужно вытолкнуть во внешнюю память, чтобы освободить место в оперативной памяти. Оптимальная стратегия замещения, реализованная в соответствующем алгоритме замещения страниц, позволяет хранить в памяти самую необходимую информацию и тем самым снизить частоту страничных нарушений.

В процессе управления страницами может возникнуть такой эффект как аномалия Билэди или «аномалия FIFO». Она заключается в том, что определенные последовательности обращений приводят к увеличению числа страничных нарушений при увеличении кадров, выделенных процессу.

Рассмотрим далее алгоритмы оптимального замещения страниц.

Оптимальный алгоритм

Замещается страница, которая не будет использоваться в течение самого длительного периода времени. Каждая страница должна быть помечена числом инструкций, которые будут выполнены, прежде чем на эту страницу будет сделана первая ссылка. Выталкиваться должна страница, для которой это число наибольшее.

Алгоритм FIFO

Каждой странице присваивается временная метка. Создается очередь страниц, в конец которой страницы попадают, когда загружаются в физическую память. Из начала берутся страницы, когда требуется освободить память. Для замещения выбирается старейшая страница.

Эта стратегия с достаточной вероятностью будет приводить к замещению активно используемых страниц. ***Пример:*** страница кода текстового процессора при редактировании файла.

Алгоритм LRU

Технология (LRU – Least Recently Used) позволяет для вытеснения выбрать страницу, которая не использовалась в течение самого долгого времени. Создается связанный список всех страниц в памяти,

в начале которого будут хранится недавно использованные страницы. Список обновляется при каждом обращении к памяти.

На каждую страницу отводится специальный 64-битный счетчик. Счетчик автоматически увеличивается на единицу после выполнения каждой инструкции. В таблице страниц имеется соответствующее поле, в которое заносится значение счётчика при каждой ссылке на страницу.

При возникновении страничного нарушения выгружается страница с наименьшим значением этого поля. Проблема заключается в том, что не существует аппаратной поддержки реализации алгоритма LRU.

Алгоритм NFU

Технология NFU (Not Frequently Used) предусматривает выгрузку редко используемой страницы. На каждую страницу отводится счетчик числа обращений и флаг обращения R(ead). Достоинство этого алгоритма заключается в том, что он может быть реализован в виде программы и позволяет организовать выгрузку страниц в соответствии с алгоритмом LRU [4].

Рассмотрим особенности данного алгоритма.

- счетчик и флаг R получают начальные значения равные нулю и связаны с каждой страницей;
- при каждом прерывании от таймера ОС сканирует всё находящиеся в памяти страницы. Для каждой страницы к счетчику добавляется значение бита R, равное 0 или 1. Значение 1 означает, что к станице было выполнено обращение процессом;
- счетчики позволяют приблизительно отследить частоту обращений к каждой странице;
- при возникновении ошибки отсутствия страницы для замещения выбирается та страница, чей счетчик имеет наименьшее значение.

Рассмотренный алгоритм обладает «памятью». В источнике [4] приводится пример отслеживания страниц процесса – компилятора программ. При многопроходной компиляции те страницы, которые интенсивно использовались при первом проходе, могут по-прежнему сохранять большие значения счетчиков и при последующих проходах.

Таким образом, если на первый проход компиляции затрачивается больше времени, чем на все остальные проходы, то страницы, содержащие код для последующих проходов, могут всегда иметь более низкие показатели счетчиков, чем страницы, использовавшиеся при первом проходе. Поэтому ОС будет замещать нужные страницы вместо тех, надобность в которых уже отпала.

Для устранения этого недостатка применяется модифицированный алгоритм NFU. Значение счетчика сдвигается в право, а в крайней левой части счетчика устанавливается бит, значение которого представляет собой бит обращения к странице R. Как правило, разрядность счетчика равна 8 битам [4]. Приведем пример состояния счетчика страницы (табл. 5.1). К странице было произведено обращение, а затем обращения прекратились.

На каждом такте значение счетчика сдвигается вправо и соответственно уменьшается. В результате будет корректно обнаружена страница, подлежащая вытеснению.

Таблица 5.1

Состояние счетчика страницы

Такт 0	1	0	1	0	1	1	53
Такт 1	0	1	0	1	0	1	21
Такт 2	0	0	1	0	1	0	10
Такты	R/5	4	3	2	1	0	Номера битов

Трешинг

Высокая частота страничных нарушений называется ***трешинг*** (thrashing – «пробуксовка»). Для устранения трешинга или его минимизации процессу необходимо предоставить в оперативной памяти набор страниц, который называется рабочим множеством W(t,T) процесса.

Число страниц в рабочем множестве определяется параметром T, является неубывающей функцией T и относительно невелико. Параметр T называют размером окна рабочего множества, через которое ведется наблюдение за процессом. Параметр t – множество кад-

ров процесса, не вошедших в рабочее множество. На рис. 5.2 показано рабочее множество некоторого процесса [1]:

$$W(t,T)=\{5,4,6,7\}.$$

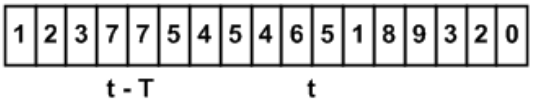

Рис. 5.2. Рабочее множество процесса

Страничные демоны

Алгоритмы, обеспечивающие поддержку системы в состоянии отсутствия трешинга, в современных ОС реализованы в составе фоновых процессов. Эти процессы часто называют ***демонами*** или ***сервисами***, которые периодически «просыпаются» и инспектируют состояние памяти.

Если свободных кадров оказывается мало, то фоновый процесс может сменить стратегию замещения. Назначение такого фонового процесса поддерживать систему в состоянии наилучшей производительности.

Контрольные вопросы и задания

1. Перечислите основные особенности организации виртуальной памяти ЭВМ.
2. Как организуется виртуальная память на основе двухуровневой таблицы страниц?
3. Как используется ассоциативная память в процессе обслуживания виртуальной памяти?
4. Дайте описание страничных нарушений.
5. Что такое стратегия управления виртуальной памятью?
6. Как используется стратегия FIFO для управления виртуальной памятью?
7. Сравните алгоритмы управления виртуальной памятью LRU и NFU.
8. Что такое рабочее множество памяти?
9. Что такое трешинг?
10. Опишите функции страничного демона ОС.

Лекция №6. ФИЗИЧЕСКАЯ ОРГАНИЗАЦИЯ ПОДСИСТЕМЫ ВВОДА-ВЫВОДА

Системная шина, контроллеры устройств, прерывания в среде ОС, контроллер прерываний.

Современные ЭВМ создаются на основе архитектуры фон Неймана (Джон фон), принципиальная схема такой ЭВМ показана на рис. 6.1.

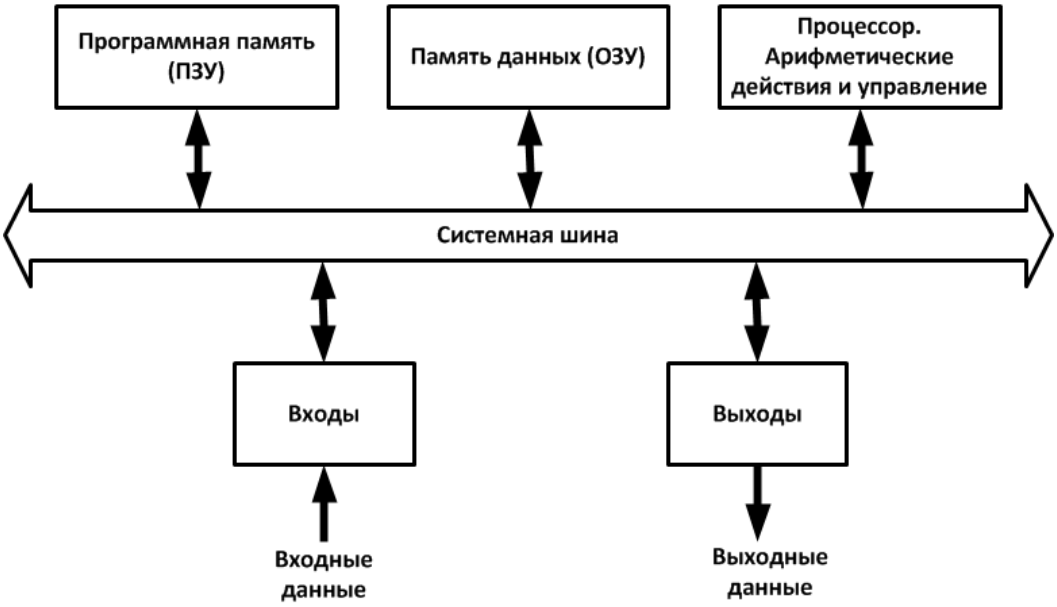

Рис. 6.1. Принципиальная схема ЭВМ

В состав ЭВМ входит память, которая подразделяется на оперативную память ОЗУ (оперативное запоминающее устройство), которая содержит код исполняемой программы и данные, необходимые для ее работы, и ПЗУ (постоянное запоминающее устройство), которое содержит программы, управляющие работой ЭВМ и выполняющие ее тестирование при подаче питания до момента загрузки ОС.

Микропроцессор, состоящий из устройства управления и арифметико-логического устройства, выполняет инструкции – команды программы, находящиеся в ОЗУ.

Взаимодействие процессора с памятью и устройствами реализуется посредством среды передачи данных, которая называется ***системной шиной***.

Системная шина

Процессор, память и многочисленные внешние устройства связаны большим количеством электрических соединений – линий. Линии принято называть локальной магистралью компьютера – системной шиной.

Внутри системной шины находятся линии, служащие для передачи сходных сигналов и предназначенные для выполнения сходных функций. Их принято группировать в магистрали. Магистрали также принято называть шинами. Таких шин три.

Шина данных – служит для передачи информации между процессором и памятью, процессором и устройствами ввода-вывода, памятью и внешними устройствами.

Адресная шина – служит для задания адреса ячейки памяти или указания устройства ввода-вывода, участвующих в обмене информацией.

Шина управления – состоит из линий управления системной шины и линий ее состояния, определяющих поведение системной шины. В некоторых ЭВМ линии состояния выносятся из этой шины в отдельную шину состояния.

Характеристики шин

Разрядность – количество линий, входящих в состав шины.

Ширина адресной шины определяет максимальный размер оперативной памяти, которая может быть установлена в вычислительной системе.

Ширина шины данных определяет максимальный объем информации, которая за один раз может быть получена или передана по этой шине.

Подключение устройств к шине

Внешнее устройство подключается к шине через порт ввода – вывода. ***Порт ввода-вывода*** – точка подключения устройства к магистрали. ***Адресное пространство ввода-вывода*** – служит для отображения портов. При этом каждый порт ввода-вывода получает

свой номер или адрес. Занесение информации в порт представляет собой инициализацию операции ввода-вывода.

Что именно должны делать устройства, приняв информацию через свой порт, и каким именно образом они должны поставлять информацию для чтения из порта, определяется контроллером устройства.

Механизм ввода-вывода

Устройства ввода-вывода подключаются к системе через порты. Могут существовать два адресных пространства: пространство памяти и пространство ввода-вывода.

Порты, как правило, отображаются в адресное пространство ввода-вывода и иногда – непосредственно в адресное пространство памяти. Использование того или иного адресного пространства определяется типом команды, выполняемой процессором, или типом ее операндов.

Физическим управлением устройством ввода-вывода, передачей информации через порт и выставлением некоторых сигналов на магистрали занимается контроллер устройства.

Архитектура контроллера

Основу архитектуры контроллера образуют его регистры. Регистр – внутренняя память контроллера. Для доступа к содержимому этих регистров вычислительная система может использовать один или несколько портов. Рассмотрим базовый состав регистров контроллера.

Регистр состояния содержит биты, значение которых определяется состоянием устройства ввода-вывода и которые доступны только для чтения вычислительной системой. В его составе выделяют флаги, биты:

Бит занятости – завершение выполнения текущей команды на устройстве.

Бит готовности данных – наличие очередного данного в регистре выходных данных.

Бит ошибки – возникновение ошибки при выполнении команды.

Регистр управления получает данные, которые записываются вычислительной системой для инициализации устройства ввода-вывода или выполнения очередной команды, а также изменения режима работы устройства.

Регистр выходных данных служит для помещения в него данных для чтения вычислительной системой.

Регистр входных данных предназначен для помещения в него информации, которая должна быть выведена на устройство.

Емкость перечисленных регистров не превышает ширину линии данных. Некоторые контроллеры могут использовать в качестве регистров очередь FIFO для буферизации поступающей информации.

Взаимодействие с контроллером

Процессор в цикле читает информацию из порта регистра состояний и проверяет значение бита занятости. Если бит занятости установлен, то это означает, что устройство еще не завершило предыдущую операцию и процессор уходит на новую итерацию цикла. Если бит занятости сброшен, то устройство готово к выполнению новой операции и процессор переходит на следующий шаг.

Процессор записывает код команды вывода в порт регистра управления.

Процессор записывает данные в порт регистра входных данных.

Стратегии взаимодействия с контроллером

Ожидание освобождения устройства путём непрерывного опроса значение бита занятости. Если скорости работы процессора и устройства ввода-вывода примерно равны, то это не приводит к существенному уменьшению полезной работы, совершаемой процессором [3].

Более эффективный способ взаимодействия между устройством и процессором является оповещение внешним устройством процессора о завершении команды вывода или команды ввода путем использования механизма прерываний.

Виды прерываний

Маскированные прерывания – прерывания, которые могут быть запрещены программным путем. Соответственно немаскированные прерывания – часть прерываний, которые не возможно запретить.

Кроме того, различают ***аппаратные*** и ***программные*** прерывания. Аппаратные прерывания – прерывания внешние.

Внешнее прерывание обнаруживается процессором между выполнением команд. Прерывание происходит асинхронно с работой процессора и непредсказуемо, программист не может предугадать, в каком именно месте работы программы произойдёт прерывание.

Процессор при переходе на обработку прерывания сохраняет часть своего состояния перед выполнением следующей команды.

К прерываниям также относят ***исключительные ситуации***. Они возникают во время выполнения процессором команды. К их числу относятся ситуации переполнения, деления на ноль, и т.д. Исключительные ситуации обнаруживаются процессором во время выполнения команд.

Процессор при переходе на выполнение обработки исключительной ситуации сохраняет часть своего состояния перед выполнением текущей команды. Исключительные ситуации возникают синхронно с работой процессора, но непредсказуемо для программиста.

Программные прерывания возникают после выполнения специальных команд, как правило, для выполнения привилегированных действий внутри системных вызовов. Процессор при выполнении программного прерывания сохраняет своё состояние перед выполнением следующей команды. Программные прерывания, возникают синхронно с работой процессора и абсолютно предсказуемы программистом.

Вектор прерываний и контроллер

Обработку прерываний современные ОС выполняют на основе механизма векторов прерываний. ***Вектор прерываний*** – номер ячейки памяти, где хранится адрес обработчика прерывания. Получив сигнал прерывания, процессор приостанавливает обработку команд текущего процесса – программы и переходит на вектор прерывания. По адресу, который хранится в векторе прерывания, процессор пере-

ходит на программу обработки прерывания, и по завершении этой программы процессор возвращается к прерванному процессу и продолжает его обработку.

Вектора прерываний образуют таблицу прерываний ОС. Для эффективной обработки аппаратных прерываний в состав современных ЭВМ включают специальный контроллер. На рис. 6.2 показана принципиальная схема контроллера прерываний, предназначенного для совместной работы с процессором Intel 8086.

Устройства ЭВМ связаны линиями IRQ с контроллером прерывания. Получив сигнал прерывания от устройства, контроллер передаёт процессору сигнал прерывания и вектор прерывания.

В табл. 6.1 приводится описание линий получения сигнала прерывания.

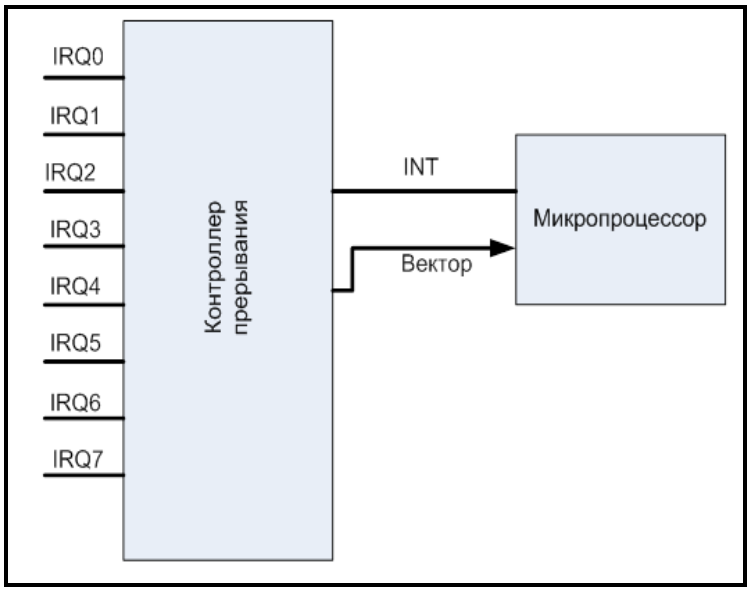

Рис. 6.2. Контроллер аппаратных прерываний

Таблица 6.1

Линии сигналов аппаратных прерываний

Линия прерывания	Вектор прерывания	Устройство
IRQ0	08h	Таймер
IRQ1	09h	Клавиатура
IRQ2	0Ah	Зарезервирован
IRQ3	0Bh	Последовательный порт 2

Продолжение табл. 6.1

Линия прерывания	Вектор прерывания	Устройство
IRQ4	0Ch	Последовательный порт 1
IRQ5	0Dh	Жёсткий диск
IRQ6	0Eh	Гибкий диск
IRQ7	0Fh	Принтер, параллельный порт

Контрольные вопросы и задания

1. Дайте описание компонентам ЭВМ, которая отвечает архитектуре фон Неймана.
2. Какими особенностями обладает системная шина ЭВМ?
3. Что такое порт ввода-вывода устройства?
4. Для чего используется контроллер устройства ЭВМ?
5. Какие регистры входят в состав контроллера устройства?
6. Дайте классификацию флагов-битов регистра состояния контроллера.
7. Как процессор взаимодействует с контроллером?
8. Приведите классификацию стратегий взаимодействия процессора и устройств ЭВМ.
9. Дайте классификацию прерываний в среде ОС.
10. Что такое вектор прерывания ОС?
11. Как используется контроллер аппаратных прерываний?

Лекция №7. ЛОГИЧЕСКАЯ ОРГАНИЗАЦИЯ ПОДСИСТЕМЫ ВВОДА-ВЫВОДА

Устройства ЭВМ, драйвера ОС, структура БСВВ (базовой системы ввода-вывода), прямой доступ к памяти, спулинг, поддержка подсистемы ввода-вывода, особенности устройств, порты, линии прерывания, канал DMA.

Особенности устройств ЭВМ

Скорость обмена информацией меду процессором и устройством ЭВМ может варьироваться в диапазоне от нескольких бай-

тов в секунду (клавиатура) до нескольких гигабайтов в секунду (сетевые карты).

Одни устройства могут использоваться несколькими процессами параллельно (являются разделяемыми), в то время как другие требуют монопольного захвата процессом.

Устройства могут запоминать выведенную информацию для ее последующего ввода и обеспечивать последовательный доступ к данным в жёстко заданном порядке.

Устройство может находить и передавать только необходимую порцию данных. Часть устройств может передавать данные только по одному байту последовательно (символьные устройства). Часть устройств может передавать блок байтов как единое целое (блочные устройства).

Кроме того, существуют устройства, предназначенные только для ввода информации или только для вывода информации, и устройства, которые могут выполнять и ввод, и вывод одновременно.

На рис. 7.1 показана схема организации БСВВ. Данная подсистема – посредник между ядром ОС и аппаратным обеспечением ЭВМ (Hardware).

Драйвер

Подсистема БСВВ реализует взаимодействие с устройством ЭВМ посредством драйверов. Драйвер – это программа для управления устройством в составе ядра ОС. Она принимает системный вызов от программы и передает его устройству. Программа драйвер использует для своей работы механизм прерываний ОС.

Программа драйвер состоит из нескольких секций [3]:

Секция запуска – инициирует операцию ввода – вывода. Эта секция запускается при включении устройства ввода – вывода, либо при выполнении очередной команды.

Секция продолжения – осуществляет основную работу по передаче данных. Обработчик прерывания. Может быть несколько таких секций при нескольких прерываниях.

Секция завершения – отключает устройство ввода – вывода либо завершает операцию.

Функции базовой подсистемы ввода-вывода

Базовая подсистема ввода-вывода служит посредником между процессами вычислительной системы и набором драйверов.

Системные вызовы для выполнения операций ввода-вывода трансформируются ею в вызовы функций необходимого драйвера устройств (см. рис. 7.1).

Перечень функций:

- поддержка блокирующихся, не блокирующихся и асинхронных системных вызовов;
- буферизация и кэширование входных и выходных данных;
- осуществление спулинга и монопольного захвата внешних устройств;
- обработка ошибок и прерываний, возникающих при операциях ввода-вывода;
- планирование последовательности запросов на выполнение этих операций.

Рис. 7.1. Организация БСВВ

Системные вызовы

Взаимодействие программы с устройством требует выполнения специальных системных вызовов. Такие вызовы делятся на следующие категории [1]:

- блокирующиеся;
- не блокирующиеся;
- асинхронные системные вызовы.

Блокирующийся вызов приводит к блокировке инициировавшего его процесса, т.е. процесс переводится операционной системой из состояния исполнения в состояние ожидания. Завершив выполнение всех операций ввода-вывода, предписанных системным вызовом, операционная система переводит процесс из состояния ожидание в состояние готовности. После того как процесс будет снова выбран для исполнения, в нем произойдет окончательный возврат из системного вызова.

Типичным для применения такого системного вызова является случай, когда процессу необходимо получить от устройства строго определенное количество данных, без которых он не может выполнять работу.

При неблокирующемся системном вызове процесс не переводится в состояние ожидания. Системный вызов возвращается немедленно, выполнив предписанные ему операции ввода-вывода полностью, частично или не выполнив совсем, в зависимости от текущей ситуации.

В более сложных ситуациях процесс может блокироваться, условием его разблокирования является завершение всех необходимых операций или окончание некоторого промежутка времени. ***Пример*** – клавиатурный ввод.

При асинхронном системном вызове процесс никогда в нем не блокируется. Системный вызов инициирует выполнение необходимых операций ввода-вывода и немедленно возвращается, после чего процесс продолжает свою обычную деятельность.

Независимо от типа блокировки по окончании завершения операции ввода-вывода ОС впоследствии информирует процесс:

- изменением значений некоторых переменных;
- передачей ему сигнала или сообщения.

Буферизация

Под буфером обычно понимается некоторая область памяти для запоминания информации при обмене данных [1]:

- между двумя устройствами;
- двумя процессами;
- процессом и устройством.

Первая причина буферизации – это разные скорости приема и передачи информации, которыми обладают участники обмена. ***Пример:*** набор символов сообщения с помощью клавиатуры и передача их в сеть через модем.

Вторая причина буферизации – это разные объемы данных, которые могут быть приняты или получены участниками обмена одновременно. ***Пример:*** получение информации из сети через модем и запись их на жесткий диск.

Третья причина буферизации связана с необходимостью копирования информации из приложений, осуществляющих ввод-вывод в буфер ядра операционной системы, и обратно.

Канал прямого доступа к памяти

Прямой доступ к памяти (Direct Memory Access – DMA) применяется когда запись или чтение большого количества информации из адресного пространства ввода-вывода приводят к большому количеству операций ввода-вывода, которые должен выполнять процессор.

Для освобождения процессора от операций последовательного вывода данных из оперативной памяти или последовательного ввода в неё был предложен механизм прямого доступа внешних устройств к памяти – DMA.

При прямом доступе к памяти процессор и контроллер DMA по очереди управляют локальной магистралью – шиной ЭВМ.

При подключении к системе нового устройства, которое умеет использовать прямой доступ к памяти, обычно необходимо программно или аппаратно задать номер канала DMA, к которому будет приписано устройство.

В отличие от прерываний, где один номер прерывания мог соответствовать нескольким устройствам, каналы DMA всегда находятся в монопольном владении устройств.

Спулинг

Технология спулинга получило свое название от английского слова spool – буфер, содержащий входные или выходные данные для устройства, на котором следует избегать чередования его использования (гонки за ресурс)

В современных вычислительных системах спулинг используется в основном для накопления выходной информации при выводе ее на печать.

Поддержка подсистемы ввода – вывода

Поддержка БСВВ в современных ОС реализуется на основе специальных системных таблиц (рис. 7.2) [1]. Рассмотрим эти таблицы.

Таблица, содержащая информацию обо всех устройствах ввода – вывода – таблица оборудования. Элемент таблицы – строка UCB (Unit Control Block) – блок управления устройством ввода – вывода. В блоке записывается:

- тип устройства и модель;
- как устройство подключено;
- номер и адрес канала прерывания;
- тип драйвера;
- тип буферизации;
- состояние устройства;
- указатель на очередь задач.

Для связи виртуальных устройств (логических устройств) с реальными устройствами используется таблица DRT (Device Reference Table). Виртуальное устройство эмулируется программным кодом в отличие от реального устройства, подключенного к системной шине. Таблица прерываний служит для связи с ядром ОС.

Для управления работой устройства БСВВ использует также специальную структуру DCB (Data Control Block) – структура, сопровож-

дающая системный запрос ввода – вывода. С её помощью ведется учёт характеристик устройства и операций по преобразованию данных.

Рис. 7.2. Системные таблицы

Контрольные вопросы и задания

1. Приведите общую классификацию устройств ЭВМ.
2. Какую роль играет программа драйвера в среде ОС?
3. Из каких подсистем состоит программа драйвера?
4. Перечислите функции подсистемы ввода-вывода ОС.
5. Дайте классификацию системным вызовам ОС.
6. Для чего используется буферизация при работе с устройствами ОС?
7. Для чего служит канал прямого доступа к памяти ЭВМ?
8. Как используется технология спулинга в современных ОС?
9. Как учитывается информация об устройствах в БСВВ?
10. Для чего используются UCB и DCB структуры в БСВВ?

Лекция №8. ДИСКОВАЯ ПОДСИСТЕМА ВВОДА – ВЫВОДА

Подсистема CHS, стратегии чтения данных с диска, ввод-вывод данных на низком уровне.

Дисковая подсистема ввода-вывода позволяет организовать хранение и чтение данных в долговременной памяти ЭВМ. Долговременная память современных ЭВМ представляет собой жесткий диск большого объема. Емкость современных жестких дисков ЭВМ достигает нескольких терабайт.

Дисковые накопители ЭВМ представляют собой пакет пластин. Для доступа к данным и их записи используется система адресации, условно названная как CHS:

- C (Cylinder) – цилиндр диска.
- H (Head) – головка привода дисков.
- S (Sector) – сектор пластины диска.

Принципиальная схема дискового накопителя показана на рис. 8.1. Основу носителя образует, как уже отмечалось, пакет пластин, покрытых магнитным слоем. Пакет пластин приводится во вращение шпинделем электродвигателя накопителя. Кроме того, в состав накопителя входит шаговый электродвигатель, который выполняет возвратно – поступательное движение пакета головок, выполняющих либо запись данных на диске, либо их чтение. Пластина накопителя имеет определенную физическую организацию, показанную на рис. 8.2.

Элементами такой структуры являются магнитные дорожки, сектора, кластеры.

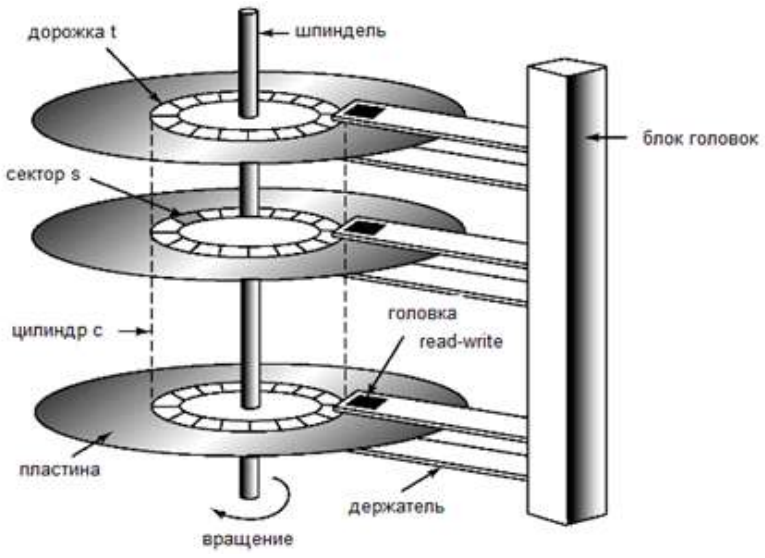

Рис. 8.1. Принципиальная схема дискового накопителя ЭВМ

На рис. 8.2 приняты условные обозначения:

A – дорожка (цилиндр);

B – сектор;

C – блок данных;

D – кластер.

Емкость сектора и размер кластера – постоянные величины. Кластер это объединение нескольких секторов. Кластер – минимальная порция данных, которая может быть либо записана на магнитный носитель, либо прочитана с поверхности носителя БСВВ.

Рассмотренная физическая структура пластины носителя формируется на заводе – изготовителе. Она может быть либо удалена, либо создана заново также с помощью специальной процедуры, которая называется ***низкоуровневым форматированием***. Для этого используется специальное программное обеспечение.

Общая емкость пакета диска может быть определена по формуле

$$Vd=C \times H \times S \times Vs,$$

где Vs – емкость сектора.

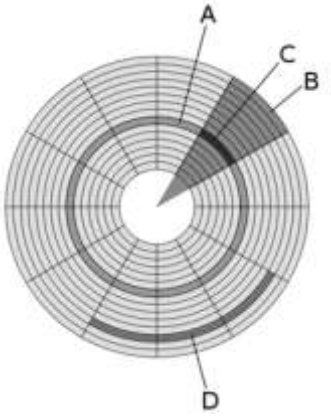

Рис. 8.2. Физическая структура носителя

Доступ к магнитному диску

Для эффективного использования долговременной памяти БСВВ должна учитывать физические особенности ЭВМ. Так, время доступа у современных ЭВМ к оперативной памяти составляет примерно 0.5 нс, а чтение данных с диска около 20 мс. Поэтому для эффективного использования внешней памяти используют кэширование и алгоритмы оптимизации чтения данных с жесткого диска.

Кэширование

Кэширование – использование буфера в оперативной памяти для работы с диском. Рассмотрим варианты кэширования.

Операция отложенной записи. При записи данные помещаются в кэш оперативной памяти и программа продолжает свою работу. Содержание буфера – кэширование записывается на диск позже фоновыми процессами.

Упреждающее чтение. С диска считываются требуемые блоки данных и ряд блоков, которые находятся рядом с затребованным.

Методы выполнения запросов к диску

Использование оптимальных стратегий – методов запросов к жесткому диску – позволяет увеличить скорость обмена данными [1].

SSTF (shortest seek time first) – с наименьшим временем поиска. Следующим выбирается запрос, для которого нужно минимальное перемещение с цилиндра на цилиндр. Очередь запросов не имеет значения.

Недостатки:

- дискриминация запросов;
- концентрация запросов – увеличение времени обслуживания запросов.

Достоинство: максимальная пропускная способность.

Scan – сканирование. Движение головок то в одном, то в другом привилегированном направлении. По пути обслуживаются подходящие запросы. Если запросов нет – движение происходит в обратном направлении.

Next-Step Scan. На каждом проходе обслуживаются только те запросы, которые уже существовали на момент начала прохода. Новые запросы – формируют новую очередь запросов для обслуживания их на обратном ходу.

C-Scan. Циклическое сканирование. Циклическое движение головок от самой наружной дорожки к внутренним. По пути обслуживаются имеющиеся запросы. Затем выполняется переход к наружным цилиндрам.

Запросы, поступившие во время текущего хода, обслуживаются не попутно, а при следующем ходе. Метод позволяет исключить дис-

криминацию запросов к самым крайним цилиндрам. Это «элеваторная дисциплина» обслуживания.

Контрольные вопросы и задания

1. Почему магнитный диск современных ЭВМ называют пакетом магниных дисков?
2. Какую логическую структуру имеет магнитный диск ЭВМ?
3. Как определить емкость магнитного диска ЭВМ?
4. Что такое сектор магнитного диска?
5. Что такое цилиндр магнитного диска?
6. Как называется минимальная порция данных, записанная на магнитный диск?
7. Для чего используется кэширование при работе с магнитным диском?
8. Как реализуется кэширование при работе с магнитным диском?
9. Дайте характеристику методам работы с диском SSTF и Scan.
10. Дайте характеристику методам работы с диском Next-Step Scan и C-Scan.

Лекция №9. ОРГАНИЗАЦИЯ ФАЙЛОВОЙ СИСТЕМЫ

Понятие файловой системы, адресация дискового пространства, файловая система NTFS, файловая система UNIX.

Структура магнитного диска

Файловая система современных ОС, таких как MS Windows, UNIX, Linux, Mac OS, предполагает определенную организацию носителя данных.

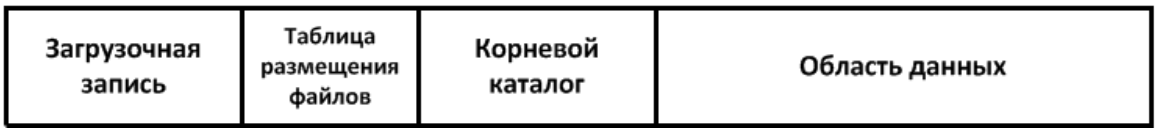

Рис. 9.1. Логическая организация диска в ОС MS Windows

В операционной системе MS Windows на носителе данных можно выделить следующие области (рис. 9.1):

- загрузочная запись;
- таблица размещения файлов;
- корневой каталог;
- область данных.

Область данных жесткого диска может быть поделена на разделы. Таким разделам присваиваются буквы латинского алфавита. Буквы A и B зарезервированы для дисководов гибких дисков – дискет. В настоящее время такие носители данных практически полностью вышли из употребления. Таким образом, первый раздел получает букву C. При подключении к ЭВМ внешних носителей данных они также рассматриваются как разделы и им присваивается буква [3].

Элементы файловой системы

В области данных носителей формируется логическая структура данных. Введем базовые определения.

- ***файл*** – именованная область на диске для хранения информации;
- ***каталог*** – именованная область на диске для хранения каталогов и других файлов;
- ***корневой каталог*** – главный каталог верхнего уровня, от которого строятся все остальные каталоги.

В результате возникает разветвленная древовидная структура хранения информации.

Таблица размещения файлов

Таблица размещения файлов используется для учета дискового пространства в области данных. Она определяет, какая цепочка кластеров принадлежит файлу или каталогу.

Рассмотрим особенности FAT (File Allocation Table) – таблицы размещения файлов.

FAT16 – объем диска до 4Гб, максимальный размер файла 2Гб.

Отводится от 512 байт до 32Кб на кластер. Использовалась в семействе ОС MS DOS.

FAT32 – максимальный размер диска 8Тб (терабайт). Системными средствами ОС MS Windows можно создать раздел на диске емкостью не более 32Гб. Кластер от 512байт до 32кб используется в ОС Windows 32x, NT, Windows 2000, Windows XP. Максимальный размер файла не может превышать значение 4Гб.

Цифра после обозначения FAT – число разрядов, отводимых для адресации кластеров на диске.

Так, для FAT16 общее число кластеров будет равно 2^{16}=65536. В FAT16 12 кластеров – резервные, полное адресное пространство будет составлять 65524 кластеров.

В FAT32 резервируется 4 бита, полный объем адресного пространства будет составлять величину 2^{28} = 268435456 кластеров.

Запись корневого каталога

Запись корневого каталога играет важную роль в технологии FAT. В FAT16 запись корневого каталога состоит из следующих полей:

- имя файла (8 байт);
- расширение (3байта);
- код атрибута файла (1 байт);
- резервное поле (10 байт);
- поле времени создания файла (2 байта);
- поле даты создания файла (2 байта);
- номер первого кластера, занимаемого файлом. Точка входа в FAT (2 байта);
- размер файла (4 байта).

Атрибуты файла позволяют определить правила его использования. В технологии FAT поддерживается четыре атрибута:

- A – атрибут архивации;
- Sy – системный файл;
- H – скрытый файл;
- R – атрибут только чтения.

Правила организации доступа к данным при использовании технологии FAT показаны на рис. 9.2.

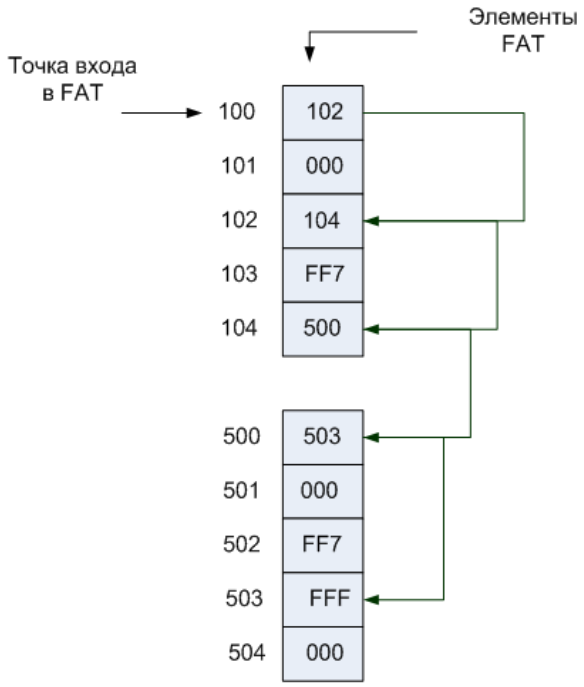

Рис. 9.2. Адресация кластеров

Файл занимает следующие цепочки кластеров на диске:
100 – 102 – 104 – 500 – 503. Адрес точки входа в FAT равен 100.
Служебные коды:

- FF7 – поврежденный кластер;
- FFF – признак конца цепочки кластеров.

Ошибки файловой системы

Повреждение записи корневого каталога FAT. Это потерянные цепочки кластеров, объявленных как занятые, они не принадлежат никакому каталогу или файлу.

Пересекающиеся кластеры. Такие кластеры разделяются одним и тем же файлом или каталогом.

Поврежденный кластер. Появляется из-за физического дефекта или износа носителя данных.

ОС Windows – логические имена файлов

При записи имен и расширений используются буквы латинского алфавита (A–Z, a–z), цифры (0–9) и специальные символы: -, _, $, &, @, %, (,), ^, ', ', {,}, ~, !, #.

При использовании виртуальной FAT (VFAT) допускается использовать кириллицу. Виртуальная VFAT впервые появилась в ОС MS Windows 95. Имя файла и расширение могут состоять из 255 символов, допускается использование пробелов.

Как было показано выше, технология FAT позволяет хранить файлы, имена которых могут иметь максимум три символа. Такая система получила уловное обозначение 8 x 3.

При использовании программ, которые не поддерживают расширение имен в соответствии с технологией виртуальной FAT, выполняется преобразование имен к формату 8 x 3.

Пример. Усечение имен при переходе к системе 8 и 3:

- Письмо~1.doc
- Письмо~2.doc

Берутся первые шесть символов имени файла, если имена получаются одинаковыми, то они дополняются двумя символами ~ и порядковым номером файла.

Полная спецификация имени файла в ОС Windows имеет вид:
буква_раздела:\путь\имя_файла

Где буква раздела – одна из латинских букв A–Z, путь – перечень каталогов, которые нужно «пройти», чтобы получить доступ к файлу. В качестве разделителя используется символ \.

Примеры:

d:\myprg.exe (файл расположен в корне раздела d)
c:\temp\doc\myfile.txt (файл находиться в подкаталоге doc каталога temp раздела c)

Регистр символов в именах каталогов и файлов не учитывается.

NTFS (New Technology File System)

Файловая система поддерживает «длинные» имена файлов. Обладает расширенным набором файловых атрибутов. На носителе может находиться до 17 миллиардов гигабайт данных.

Имена каталогов и файлов записываются в кодировке UNICODE, что позволяет использовать файловую систему для различных вариантов локализации ОС.

Отсутствуют ограничения на размер файла и каталога. Размер файла может превышать 4 Гб. Размер кластера может быть изменен при создании файловой системы.

Файловая система позволят разделять права доступа к файлам со стороны пользователей и групп.

Работы с файловой системой строится на основе механизма поддержки трансакций. Изменение файловой системы фиксируется. Если возникла ошибка, то изменения отвергаются, а файловая система возвращается к прежнему состоянию.

Файловая система поддерживает Hot Fix технологию. В случае, если сектор на магнитном носителя физически поврежден, то информация о нем заносится в специальную таблицу повреждённых секторов. Если в секторе находились данные, то делается попытка их и переноса в неповрежденный сектор [3].

Файловая система включает специальную структуру MFT (Master File Table). Каждому каталогу или файлу соответствует определенная запись в MFT (рис. 9.3).

Сама MFT является системным файлом со следующей структурой:

- первые 16 записей таблицы зарезервированы;
- первая запись описывает MFT;
- вторая запись – резервирует запись MFT;
- третья запись ссылается на специальный файл, данные которого используются для восстановления каталогов и файлов.

Данные о небольших каталогах и файлах записываются непосредственно в MFT.

Вложенные каталоги и данные организуются по принципу бинарного дерева.

Рис. 9.3. Файловая система NTFS и MFT

Правила задания имен файлов в файловой системе NTFS такие же, как и для файловой системы FAT32 (виртуальной FAT).

Файловая система ОС UNIX

Принцип организации файловой системы показан на рис. 9.4 [1, 10].

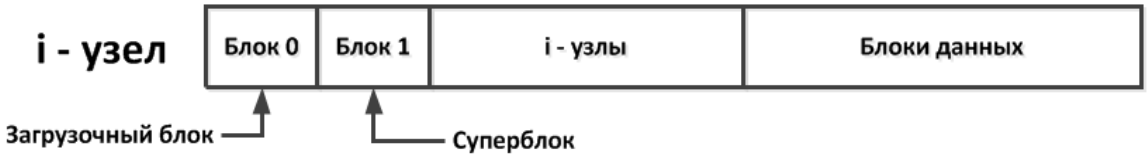

Рис. 9.4. Файловая система ОС UNIX

В составе файловой системы можно выделить четыре блока. Первые два блока – служебные. В одном из них хранится программа загрузки операционной системы в оперативную память ЭВМ, а в другом – данные о конфигурации ЭВМ, необходимые для загрузки ОС. Третий блок используется для адресации блоков данных диска с помощью структур, которые называются i узлами. Наконец последний блок – адресное пространство магнитного носителя данных.

Рис. 9.5. Структура i-го узла

Структура i-го узла показана на рис. 9.5. Блок позволяет адресовать дополнительные узлы, ссылающиеся на блоки данных каталога или файла [1, 4].

Правила задания имен файлов для ОС UNIX приводятся в лекции №10.

Контрольные вопросы и задания

1. Какую логическую структуру имеет магнитный диск в ОС MS Windows?
2. Дайте классификацию технологий FAT организации данных на диске.
3. Как выполняется адресация дискового пространства при использовании технологии FAT?
4. Что имеют в виду, когда говорят о системе 8 x 3 для FAT16?
5. Какие ошибки характерны для файловой системы FAT?
6. Какими особенностями обладает виртуальная FAT?
7. Перечислите особенности файловой системы NTFS.
8. Для чего используется MFT структура в файловой системе NTFS?
9. Какую логическую структуру имеет диск в ОС UNIX?
10. Какую структуру имеет i-узел в ОС UNIX?

Лекция №10. КОМАНДЫ ТЕРМИНАЛА Bash Linux

Терминал Linux, операции с директориями, создание файлов, вывод информации из файлов, операции с файлами, права доступа, сбор статистики о файлах, сценарии.

Окно терминала ОС Linux показано на рис. 10.1. Открыть окно можно используя команду интерфейса ОС – «Терминал».

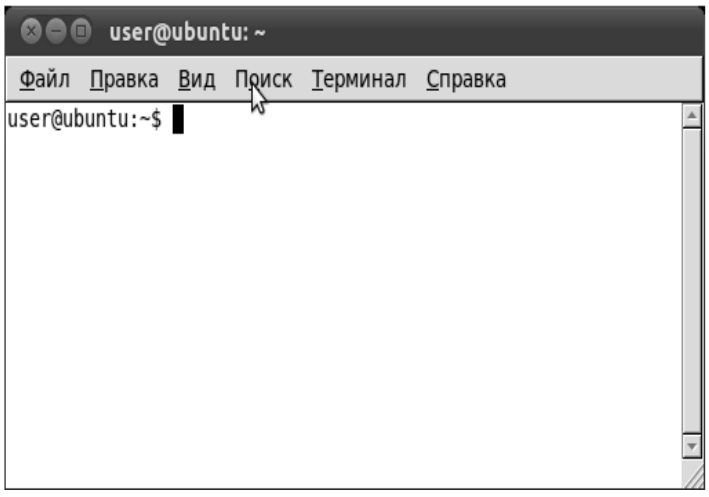

Рис. 10.1. Окно терминала

Окно терминала предназначено для ввода команд, предназначенных для выполнения системных действий в среде ОС. Команды вводятся в строке приглашения. В ОС Linux строка приглашения терминала имеет следующий формат:

имя_пользователя@имя_компьютера:~$

Команды вводятся после символа $, для принятия команды нужно нажать клавишу Enter.

Команды терминала представляют собой инструкции для интерпретатора Bash (Bourne again shell, «Born again» shell – «возрождённый» shell).

Можно получить информацию по использованию любой команды, используя команду:

man имя_команды

Здесь имя_команды – название команды для которой необходимо вывести справочную информацию. Информация о команде выводится в специальном окне. Для выхода из окна следует нажать клавишу Q – команда (Quit).

Команды Bash набираются строчными буквами, аргументы команд вводятся строчными либо прописными буквами. Выбор символов зависит от назначения аргументов команды.

Далее рассмотрим правила использования команд Bash для выполнения системных действий.

Информация о системе

Перечень команд:

pwd – вывод рабочей директории пользователя;

clear – очистка терминала;

who – получение сведений о пользователе;

tty – получение сведений о терминале;

arch – вывод кодового имени архитектуры ЭВМ;

uname -r – вывод версии ядра Linux;

Пример. Получение информации о файле терминала

```
user@ubuntu:~$ tty
/dev/pts/0
```

В системной директории **pts**, которая находится в каталоге устройств **dev**, создан символьный файл 0 для работы с терминалом.

Пример. Сведения о пользователе:

```
user@ubuntu:~$ who
user     tty7         2014-03-29 04:00 (:0)
user     pts/0        2014-03-29 04:02 (:0.0)
```

Директории

При работе с директориями – каталогами следует учитывать ряд особенностей файловой системы ОС Linux. В ОС используется единая файловая система, начинающаяся от корневого каталога.

При подсоединении носителя к ЭВМ, например, флэш накопителя, выполняется процедура монтирования. Монтирование – подсоединение к файловой системе ОС системы носителя. Для носителя отводится отдельный каталог.

В ОС Linux имеется определенный набор системных директорий [2]. Их название и количество зависят от типа сборки ОС. Однако существует определенный стандартный набор директорий:

/bin – хранит исполняемые файлы общего назначения;

/boot – содержит образ загружаемого ядра;

/dev – файлы устройств;

/etc – конфигурационные файлы общего пользования;

/home – домашние каталоги пользователей, включая программы и файлы личных предпочтений;

/lib – общесистемные библиотеки;

/mnt – каталог монтирования внешних файловых систем;

/proc – виртуальная файловая система для чтения информации о процессах;

/root – домашний каталог супер пользователя;

/sbin – программы системного администрирования;

/tmp – каталог для хранения временной информации;

/usr – каталог пользовательских прикладных программ со всеми их исполнимыми и конфигурационными файлами;

/var – каталог для хранения часто изменяющихся файлов, например, спулера печати, различных лог-файлов, почтовых сообщений и т.п.;

/lost+found – каталог для нарушенных фрагментов файлов, обнаруженных в результате проверки файловой системы после сбоя.

В директории home расположены «домашние» директории пользователей системы. Имя директории соответствует имени пользователя. Символом / обозначается корневой каталог файловой системы ОС.

Работа с директориями

ls – просмотр содержания рабочей директории, текущей директории пользователя:

```
user@ubuntu:~$ ls
Desktop  Downloads      Music  Public  Videos
Documents examples.desktop Pictures Templates
```

Команда ls-l вывод полной информации об объектах директории:

```
user@ubuntu:~$ ls -l
итого 36
drwxr-xr-x 2 user user 4096 2014-03-28 02:31 Desktop
drwxr-xr-x 2 user user 4096 2014-03-15 05:02 Documents
drwxr-xr-x 2 user user 4096 2015-09-10 07:27 Downloads
```

Смена директории выполняется командой cd. В качестве аргумента нужно задать путь к нужной директории. При задании путей используется разделитель /.

Пример команды:

```
cd /home/user/Desktop
```

При использовании команды могут быть использованы специальные символы [2]:

.. – переход в предшествующую директорию;

~ – переход в домашний каталог;

/ – переход в корневой каталог;

- – переход в директорию, которая была текущей до перехода в данную.

Создание каталогов выполняется командой:

```
mkdir имя_каталога
```

Допускается создавать сразу несколько директорий с помощью команды:

```
mkdir каталог1 каталог 2
```

Для построения одной командой дерева каталогов используется параметр –p.

Примеры использования команды:

```
user@ubuntu:~/Documents$ mkdir D1
user@ubuntu:~/Documents$ mkdir D2 D3 D4
user@ubuntu:~/Documents$ mkdir -p D4/Work/Zip
```

Удаление каталога выполняется командой:

```
rmdir имя_каталога
```

Каталог должен быть пустой !

Создание файлов

При задании имен файлов в командах следует учитывать ряд особенностей команд Bash.

Перед пробелами в имени файла следует использовать символ \. Другой вариант – имя файла или каталога должно быть взято в двойные кавычки.

Пример:

```
“Мой файл”
Мой\ файл
```

В ОС Linux, в отличие от ОС Windows, нет понятия раздела жесткого диска. ОС использует единую файловую систему. Файловая система представляет собой дерево каталогов – директорий. В основании иерархии каталогов находится корневой каталог, который имеет условное обозначение /. При присоединении сменных носителей к ЭВМ их файловая система (если она совместима с ОС) присоединяется (монтируется) к файловой системе ОС. Доступ к файловой системе сменного носителя, открывается через директорию, которая создается либо в директории mnt, либо в директории media в зависимости от типа сборки Linux. Эта директория получает системное имя.

Полная спецификация имени файла имеет вид:

```
/путь/имя_файла
```

Имена директорий и файлов регистр зависимы.

Пример:

```
/home/user/myDir/my.lst
```

Задание полного имени файла, расположенного в директории myDir, которая находиться в домашней директории пользователя user.

Создание файла может быть выполнено с помощью команды cat и символа переадресации ввода >. Команда cat выполняет вывод байтов – символов из системного устройства или файла.

Формат команды:

cat > имя_файла

Такая команда означает передачу байтов при их вводе в консоли в файл. Если файла нет, то он создается. Если он существует, то перезаписывается. Добавить байты к файлу можно с помощью операции переадресации >>.

Затем вводятся байты строк. Каждая строка завершается нажатием клавиши ENTER. Заканчивается процесс создания файла вводом признака конца файла путем нажатия комбинации клавиш CTRL+D.

Просмотреть содержание файла можно, подав команду:

cat имя_файла

Создать файл можно также, используя команду echo. Команда выводит строку аргумента на терминал, используя переадресацию строку отправляют в файл:

echo текст > имя_файла

Пример создания файла:

```
user@ubuntu:~$ echo Это мой файл > myfile
user@ubuntu:~$ cat myfile
Это мой файл
```

Создание файла в директории home/user/Documents:

```
user@ubuntu:~$ echo Test ! > /home/user/Documents/testfile
user@ubuntu:~$ cat /home/user/Documents/testfile
Test !
```

Создать файл можно, используй команду touch. Формат команды:

touch имя_файла – команда создания пустого файла.

Если файл существует, то изменяются две временные метки.

- первая метка – дата и время изменения файла;
- вторая метка – дата и время доступа.

Изменение временных меток:

```
touch –d “November 23 2012” myfile
touch –t 211506181500.55 myfile
```

При использовании ключа –t временная метка задается по определенному шаблону:

[[CC]YY]MMDDhhmm[.ss]
CC – век
YY – год
MM – месяц
DD – дата
hh – часы
mm – минуты
ss – секунды

Так, во второй команде была задана метка:

21 – век
2015 – год
Месяц: июнь
18 – число
15:00:55 – время.

Управление выводом байтов из файла

Оболочка Bash предоставляет ряд команд, которые могут быть использованы для управления выводом информации из текстовых файлов. В отличие от двоичного файла текстовый файл содержит коды символов и управляющие коды формирования строк.

Команда tac. Формат команды:

tac имя_файла – вывод содержимого файла на консоль в обратном порядке следования строк.

Команда more. Формат команды:

more имя_файла – постраничный вывод содержания.

Команда less. Формат команды:

less имя_файла – постраничный вывод с пролистыванием в обе стороны.

Команда head. Формат команды:

head -n имя_файла – вывод от начала файла n строк.

Команда tail. Формат команды:

tail -n имя_файла – вывести с конца файла n строк.

Работа с файлами

Команда mv позволяет переименовать файл или директорию. Кроме того, команда позволяет перемещать объекты.

Примеры использования команд:

mv myfile newfile – переименование файла.
mv newfile /home/user – перемещение файла.

Удаление объектов файловой системы выполняется командой rm:

rm -f имя_файла

Копирование файлов выполняется командой cp:

cp имя_файла.

Ссылка

Ссылка на файл – это указатель на его месторасположение. Символьные ссылки могут указывать на файл в другой директории. Символьная ссылка может указывать на директорию.

Чтобы создать ссылку, нужно выполнить команду:

ln -s имя_файла имя_ссылки – создание символьной ссылки.

В ОС Linux используются также жесткие ссылки. Жесткая ссылка – втрое название физического файла на диске. Команда создания жесткой ссылки:

ln имя_файла имя_ссылки – жесткая ссылка.

При удалении названия адресуемого файла на диске жесткая ссылка продолжает открывать доступ к данным файла.

При просмотре содержания директории командой ls –l ссылка помечается символом -> и ее спецификация начинается с буквы l.

Пример ссылки:

-rwxr--r-- 2 use use 19 2014-04-12 06:20 myfile.sh
lrwxrwxrwx 1 use use 9 2014-04-12 06:23 ref_1 -> myfile.sh

Ссылка ref_1 ссылается на файл myfile.sh

Атрибуты файла

Каждый файл в ОС Linux характеризуется набором атрибутов, определяющих режим доступа:

r (4) – чтение;

w (2) – запись;

x (1) – исполнение.

В скобках указан код режима доступа. Режимы доступа группируются в три группы, определяя права доступа к файлу. В ОС Linux существует три субъекта, для которых назначают права доступа.

Владелец файла (user) – пользователь, создавший этот файл.

Группа (group) – группа, в состав которой входит владелец.

Остальные (other) пользователи.

Совокупность значений атрибутов для владельца, группы и прочих пользователей образуют структуру из девяти символьных значений. По три атрибута на каждого субъекта владения. Код доступа к файлу восьмеричный.

В начале набора символьных атрибутов указывается символ идентификации объекта файловой системы:

d – директория.

- файл.

Другая буква – системный объект. Числовой код прав доступа по умолчанию имеет значение 0644. Этому коду соответствует набор символьных атрибутов -rw-r--r--.

Просмотреть атрибуты любого файла можно с помощью команды:

ls -l имя_файла

Смена атрибутов выполняется командой:

chmod числовой_код newfile

Пример использования команды:

chmod 744 myfile

Файл получит маску прав доступа:

rwxr--r–

Другой вариант использования команды – применение опций и операторов для работы с опциями.

Опции:

u – владелец;

g – группа;

o – остальные;

a – все пользователи.

Права:

r – чтение;

w – запись;

x – выполненные.

Команды:

+ добавить право;

- удалить право;

= установить абсолютные права, игнорируя существующие.

Для директории право x означает возможность использования команды cd !

Примеры использования команды:

```
chmod g+x myfile
chmod u+x,g-r,o+r myfile
chmod a-r myfile
chmod u=x myfile
```

Получение полной информации о файле

Получить полную системную информацию о файле можно с помощью команды:

```
stat имя_файла
```

Пример использования команды:

```
use@ubuntu:~$ stat myfile
File: «myfile»
Size: 12 Blocks: 8   IO Block: 4096   обычный файл
Device: 801h/2049d      Inode: 1191564     Links: 1
Access:(0644/-rw-r--r--) Uid:(1001/use)Gid:(1001/use)
Access: 2014-04-12 05:14:20.424192162 -0700
Modify: 2014-04-12 05:53:43.052980819 -0700
Change: 2014-04-12 05:53:43.052980819 -0700
```

Здесь:

File: имя файла;

Size: размер файла в байтах, число блоков, размер блока;

Device: тип устройства, Inode – номер информационного дескриптора, Links – количество жестких ссылок;

Access: права доступа к файлу в восьмеричной системе, маска прав доступа, идентификатор владельца Uid, имя владельца, идентификатор группы Gid, имя группы.

Далее следуют временные метки: чтения, модификации, изменения статуса.

Команда может быть подана с ключом f для получения информации о файловой системе:

stat –f имя_файла

```
use@ubuntu:~$ stat -f myfile
File: "myfile"
ID: d1e47a741939f6d Namelen: 255 Type: ext2/ext3
Block size: 4096 Fundamental block size: 4096
Blocks: Total: 5031335 Free: 4322533 Available: 4066956
Inodes: Total: 1277952 Free: 1122144
```

Где:

File: имя файла;

ID: идентификатор файловой системы, Namelen – максимальная длина имени для файловой системы, Type – тип файловой системы;

Block size: размер блока для файловой системы;

Blocks: полное количество блоков, всего, свободно, доступно;

Inodes – число информационных дескрипторов (I узлов): всего, свободно.

Создание файла сценария

Группу команд управления файловой системой можно выполнить автоматически. Для этого их следует поместить в текстовый файл сценария. В ОС Linux такие файлы обычно снабжают уточнением – расширением sh.

Такой файл будет исполняться системой только в том случае, если его владелец обладает правом на его исполнение – x.

Пример запуска сценария:

```
chmod u+x mytest.sh
./mytest.sh
```

Для запуска сценария нужно указать./

Точка – указатель на рабочий каталог пользователя

Контрольные вопросы и задания

1. Какую структуру имеет строка приглашения терминала ОС Linux?

2. Перечислите команды терминала, позволяющие получить информацию о системе.

3. Чем отличается рабочая директория от домашней директории?

4. Как вывести на терминал полную информацию о содержании директории?

5. Перечислите команды для работы с директориями.

6. Как создать файл, используя команду cat?

7. Как дополнить текстовый файл новыми строками?

8. Какие команды используются для работы с файлами?

9. Какие виды ссылок поддерживаются в ОС Linux?

10. Как в ОС Linux назначаются права доступа к файлам и каталогам?

11. Перечислите правила создания и запуска файла Bash сценария.

Лекция №11. СЦЕНАРИИ Bash

Структура сценария, формальные и фактические параметры, проверка условий, циклы в сценариях, использование функций, контроль за выполнением сценария.

В общем виде файл сценария состоит из набора команд и комментариев. Комментарий начинается с символа #.

Сценарии в системном программировании принято также называть ***скриптами***. ***Пример*** файла сценария script.sh:

```
#Параметры
echo " Название файла " $0 " Получено параметров " $#
echo $1 $2
```

Команда echo выводит строку на консоль. При создании скрипта можно использовать формальные параметры. Формальный параметр обозначается символом $.

Файл скрипта создается в текстовом редакторе. После сохранения ему нужно присвоить атрибуты, позволяющие его выполнить в среде Linux.

Это можно сделать с помощью команды:

```
chmod 755 script.sh
```

В результате выполнения команды файл получит атрибуты:

```
$ ls -l script.sh
-rwxr-xr-x 1 sysop sysop 132 2015-04-04 17:17 script.sh
```

Выполнение в консоли с помощью команды:

```
$./script.sh
```

Название файла./script1.sh Получено параметров 0.

Формальные параметры

Формальный параметр с номером 0 хранит имя исполняемого файла. Формальный параметр # позволяет извлечь количество фактических параметров, переданных файлу [11]. Формальные параметры получают фактические значения из командной строки при вызове файла сценария. Нумерация этих параметров начинается с единицы.

Пример. Вызов файла сценария с двумя параметрами.

```
$./script.sh p1 p2
```

Название файла./script.sh Получено параметров 2

```
p1 p2
```

Создание параметров

Формальные параметры для файла сценария могут быть созданы с помощью команды set.

Пример. Получение системной даты с помощью команды date и вывод названия месяца:

```
#Bash
echo $(date)
set $(date)
echo $2
```

Результат работы сценария:

```
Пт. нояб. 2 03:20:21 PST 2013
нояб.
```

Переменные и их использование

В файле сценария, кроме формальных параметров, можно использовать переменные для временного хранения значений. Переменные объявляются в виде:

```
имя_переменной = значение
```

Для извлечения значения перед именем нужно поставить символ $.

Примеры использования переменных:

```
mes1=“One”
mes2=“Two”
echo “mes1=$mes1 mes2=$mes2”
echo “mes1=$mes1 mes2=${mes2}ok!”
```

Использование пары скобок {} позволяет включать значение переменной в строку.

При проведении вычислений с переменными используется оператор expr. Значения переменных должны быть целыми числами. Операции:

+ сложение

- вычитание

/ деление

% остаток от деления

* умножение

При использовании умножения знак умножения необходимо «экранировать», так как это специальный символ в командах сценариях [11]:

```
a=$(expr 5 \* 2)
```

Примеры вычислений:

```
#Переменные
a=10
mes=" Вычисления "
b=0
c=0
b=$(expr $a + $a)
c=$(expr $b / 10)
echo $mes
echo " b= " $b " c= " $c
```

Следует иметь в виду, что при написании команд сценария нужно правильно использовать символ пробела, разделяя лексемы команд.

Создание переменной-команды

При работе с командами в сценариях можно получить вывод команды и сохранить его как символьную переменную.

Для этого используется объявление вида:

```
v=$(Команда)
```

Пример. Получение результат команды ls в переменную
mes=$(ls -l testset)

Переменные окружения

После запуска оболочки – терминала в среде ОС Linux формируется процесс, которому передается определенный набор системных переменных окружения.

Перечислим некоторые переменные [11]:

PATH – перечень каталогов, в которых отыскиваются исполняемые программы при подаче команды в терминале.

PWD – полный путь к рабочей директории пользователя.

USERNAME – имя пользователя.

HOME – путь к домашней директории пользователя.

Вывести на терминал полный список окружения можно с помощью команды env.

Если в теле файла сценария выполняется вызов другого файла сценария, то ему можно передать переменную как, переменную его окружения. Для этого используется команда:
export имя_перменной.

Пример. Передача сценарию имени файла.

```
myfile=“lines.txt”
export myfile
./new.sh
```

Вывод сообщений

Команда echo позволяет выводить текст на терминал, при этом автоматически выполняется переход на следующую строку.

Для подавления перехода на следующую строку следует использовать команду с параметром –n:

```
echo -n “Hello World !”
```

Проверка условий

Для изменения порядка выполнения команд сценария используется команда if. Общий формат команды имеет вид:

```
if условие
then
  команды
else
 команды
fi
```

Для проверки условий используется команда:

test команда.

Рассмотрим команды проверки условий.

Проверка объектов файловой системы

-f Name – проверка наличия файла на диске;

-d Name – проверка наличия каталога на диске;

-L Name – файл – символьная ссылка;

-r Name – имеется разрешение на чтение файла;

-w Name – имеется разрешение на запись в файл;

-x Name – имеет разрешение на выполнение файла;

-s Name – файл не пустой.

Здесь Name – наименование файла или каталога.

Проверка числовых условий

test Number1 -eq Number2 – проверка на равенство двух чисел.

test Number1 -ne Number2 – проверка неравенства двух чисел.

test Number1 -ge Number2 – первое число больше или равно.

test Number1 -gt Number2 – первое число больше второго.

test Number1 -le Number2 – первое число меньше или равно.

test Number1 -lt Number2 – первое число меньше второго.

Проверка строк

test –n String – не пустая строка (существует).

test –z String – строка нулевой длины (не существует).

test String1 = String2 – сравнение строк.

test String1 != String2 – проверка неравенства строк.

Пример использования условия. Контроль числа формальных параметров.

```
#Использование условий
echo "Название файла " $0 " введено " $# " параметров"
if test $# -eq 2
then
   echo "Параметры заданы верно"
else
   echo "Неправильное задание параметров"
fi
if test "$1" = "new"
then
  echo "Выполнение команды " $1
else
  echo "Отмена операции" $1
fi
if [ "$2" = "file" ]
then
 echo "Обработка " $2
else
 echo "Неверное имя !"
fi
```

Ключевое слово test может быть заменено парой скобок [ ].

Использование команды elif

Данная команда позволяет проверять дополнительные условия при использовании команды else.

Сценарий работы с файловой системой. Обратите внимание на использование символа «;». Он позволяет записывать ключевое слово then на одной строке с условием.

```
#Bash
cat << !
```

Введите режим работы:

```
1 – краткий вывод списка файлов
2 – полный вывод списка файлов
3 – просмотр прав доступа для файла
!
read work
if test "$work" = "1"; then
  ls
elif test "$work" = "2"; then
  ls -l
```

```
elif test "$work" = "3"; then
   echo "Введите имя файла:"
   read fname
   ls -l $fname
else
   echo "Неверный ввод:"
fi
```

Для вывода нескольких строк текста использована команда cat с символами << и !

Список типа «И»

Список позволяет выполнить ряд команд по следующему правилу: команда1 && команда2 && команда3 && ...

Оператор && служит для проверки результата выполнения предыдущей команды. Если результат выполнения команда «ИСТИНА» – true, то выполняется следующая команда списка.

Пример.

```
#Bash
a="one"
b="two"
c="three"
if test "$a" = "one" && test "$b" = "two" &&↵
test "$c" = "three"; then
  echo "Правильный ряд !"
else
  echo "Неверный ряд !"
fi
```

В результате выполнения списка будет выведено сообщение «Правильный ряд».

Использование операторного блока

Операторный блок записывается с помощью пары фигурных скобок {}.

Он позволяет выполнить несколько действий там, где по умолчанию используется только один оператор.

Пример. Создание файла с текущей календарной датой:

```
#Bash
```

```
a="no"
if test "$a" = "go" && {
date > cur_date
cat cur_date
}; then
  echo "Файл создан !"
else
  echo "Отмена операции !"
fi
```

Список типа «ИЛИ»

Если команда возвращает значение «ЛОЖЬ» – false, то выполняется следующая команда. Последовательность команд или списка выполняется до тех пор, пока команда списка не вернёт значение «ИСТИНА».

Такой список имеет вид:

```
команда 1  ||   команда 2 || команда 3     ||  …
#Bash
a="one"
b="ten"
c="seven"
if test "$a" = "one" || test "$b" = "two" || test "$c" = "three"; then
  echo "Правильный ряд !"
else
  echo "Неверный ряд !"
fi
```

Множественный выбор

Множественный выбор – селекторный выбор. Организация селекторного выбора выполняется с помощью оператора case.

```
case Имя_переменной  in
Маска) Команды;;
Маска) Команды;;
*) Команды, выполняемые по умолчанию;;
esac
```

Пример тестирования названия цветов:

```
case "$1" in
    red) echo "Красный";;
    green) echo "Зеленый";;
```

```
    blue)  echo "Синий";;
    *) echo "Не определено";;
esac
```

В языке написания скриптов используют маски (регулярные выражения):

* – замена нескольких символов.

[буквaБуква] – проверка на наличие определенного символа в слове.

| – альтернативная проверка масок: Либо такой вариант, либо другой.

? – проверка наличия одного символа.

Пример. Проверка имени файла.

Требуется разработать сценарий получения имени файла от пользователя. Допустимые варианты имени файла:
Data_file, datta_file, первая буква файла f, первая буква файла F.

Сценарий:

```
#Bash
case "$1" in
    [Dd]ata_file) echo "Задан файл данных";;
    f* | F*)  echo "Имя файла задано верно";;
    *) echo "Имя файла неверное";;
esac
```

Цикл while

Общая форма циклической конструкции:

```
while Условие
do
Команды
done
```

Данная конструкция выполняется до тех пор, пока «Условие» имеет значение true (ИСТИНА).

Для работы с формальными параметрами удобно использовать команду shift. Данная команда выбирает последовательно фактические значения из командной строки при вызове файла скрипта. Тогда проверку наличия фактических значений можно выполнить командой while.

Например файл сценария был вызван в указанием ряда параметров:

```
$ test_script  p1 p2 p3 p4 p5
```

Тогда перебор формальных параметров можно выполнить с помощью сценария:

```
#Пример последовательного
#выбора параметров
while test -n "$1"
do
  echo "Parameter – $1"
  shift
done
```

Команда for

Данная команда позволяет выполнить определенное число итераций – шагов. Число шагов определяется числом фактических значений в списке команды:

```
for переменная in список
do
  команды
done
```

Переменная получает фактическое значение из списка.

Пример использования команды:

```
#Создание файла с дополненным #содержанием:
for line in "Fist line" "Second Line" "Third Line"
do
 echo $line
 echo $line >> test_file
done
```

Цикл until

Цикл выполняется до тех пор, пока условие выхода из цикла не станет истинным:

```
until условие
do
 операторы
done
```

Пример использования команды:

```
#Сценарий – прототип контроля
#пароля пользователя.
Ключевое #слово «ok».
clear
```

```
input="***"
until test "$input" = "ok"
do
  clear
  echo "Введите ключевое слово"
  read input
done
```

Управление работой цикла

Команда break – прерывает работу цикла и передает управление следующему оператору. Команда continue – передает управление на заголовок цикла для проверки условия.

Использование команды break:

```
#Bash 1
for fname in "file 1" "file 2" "file 3"
do
  echo $fname
  if test "$fname" = "file 2"; then
    break
  fi
done
```

Использование команды continue:

```
#Bash 2
for fname in "file 1" "file 2" "file 3"
do
  if test "$fname" = "file 2"; then
   continue
  fi
  echo $fname
done
```

Перебор файлов

Цикл for удобно использовать для организации перебора объектов файловой системы с помощью регулярных выражений.

В данных выражениях использую специальные символы:

* – Замена нескольких символов.

? – Замена одного символа.

Примеры формирования списков:

in * – Перебор всех объектов файловой системы в текущем каталоге.

in *f – Перебор в текущем каталоге объектов, у которых последний символ f, а остальные произвольные.

in t??? – Перебор в текущем каталоге объектов, начинающихся на букву t и имеющих три произвольных символа далее.

Формирование списка значений

При организации цикла можно использовать команду $(command).

Команда формирует список значений для переменной цикла.

Пример. Вывод списка файлов в текущем каталоге, которые начинаются на букву u:

```
#Bash
clear
for file in $(ls u*)
do
  echo $file
done
```

Создание функции

Функция это именованная часть скрипта, которая может вызываться в любой его точке. Объявляется функция в следующем виде:

```
Имя_Функции()
{
Команды функции
}
```

Команда вызова функции имеет вид:

```
Имя_Функции p1 p2 …
```

Фактические параметры передаются в функцию так же, как и при вызове сценария. Формальные параметры функции нумеруются, начиная с 1, и обозначаются как $n, где n = {1,2,…}.

Параметры функции:

```
#Функция без параметров
getInfo(){
clear
echo "User:"
```

```
who
echo "Terminal:"
tty
}
echo "User inforamtion:"
getInfo
```

Вызов функции с двумя параметрами:

```
msg_function()
{
echo $1
echo $2
}
msg_function One Two
```

Функция получит два фактических значения – строки “One” и “Two”. Для получения строки, которая содержит все переданные функции фактические значения, нужно извлечь значение параметра *.

Функция. Вывод параметров

```
#Работа с параметрами.
get_param()
{
  echo $1
  echo $2
  echo $*
}
one="Hello"
two=" World !"
get_param $one $two
```

Управление вводом-выводом

Вывод большого объёма текста с помощью команды может быть произведён с помощью режима переадресации << и символа !.

```
#Создание файла
functionMakeFile()
{
cat <<!
Введите строки
после каждой строки нажмите enter
завершение ввода ctrl+d
!
cat > $1
}
```

```
echo "Input file name:"
read fname
functionMakeFile $fname
```

Для ввода данных используют команду read. Она позволяет получить от пользователя сценария строку текста.

Контроль результата

При работе с функциями можно использовать технологию возврата результата работы функции. Для этого служит оператор return n. Здесь n – возвращаемое значение. Значение – числовой код.

После выполнения команды управление передаётся в вызывающую часть сценария.

Пример возврата кода.

```
#Bash
get_login()
{
  echo "Введите Ваше имя "
  read name
  if test "$name" = "user"
  then
    return 0
  else
    return 1
  fi
}
if get_login
    then
     echo "Login – OK !!!"
    else
     echo "Login – False !!!"
fi
```

Для прерывания работы сценария используется команда exit n.

Здесь n – числовой код. Этот оператор прерывает работу сценария в любой его точке и передаёт управление операционной системе.

При использовании команд return и exit следует иметь в виду: если возвращаемое значение равно нолю, то оно принимается как «ИСТИНА» – true.

Контрольные вопросы и задания

1. Какую структуру имеет файл сценария?
2. Для чего используются формальные и фактические параметры сценария?
3. Как используются переменные в файле сценария?
4. Что такое переменная окружения?
5. Как выполнить ввод в файле сценария?
6. Как выполняется проверка условий в файле сценария?
7. Как реализуется селекторный (множественный) выбор в файле сценария?
8. Дайте классификацию для команд организации циклов в файле сценария.
9. Как используются функции в файле сценария?
10. Как организовать возврат результата работы функции?
11. Как выполнить ввод данных в файле сценария?

Лекция №12. ПРОЦЕССЫ ОПЕРАЦИОННОЙ СИСТЕМЫ Linux

Понятие процесса, мониторинг процессов, управление процессами, создание процессов, взаимодействие процессов.

Полными возможностями по управления процессами в ОС Linux обладает суперпользователь. Суперпользователь Linux – root пользователь. Напрямую нельзя выполнить регистрацию от имени root. Можно выполнить действия в режиме root (супер пользователя). Для перехода в режим суперпользователя нужно подать команду su. Затем следует ввести пароль. Выход из режима происходит после подачи команды exit.

При переходе в режим супер пользователя в строке приглашения терминала появляется символ #:

```
use@ubuntu:~$ su
Пароль:<-Ввод пароля
root@ubuntu:/home/use#
```

Установить новый пароль root может пользователь с правами администратора с помощью команды:

sudo passwd root

Затем следует ввести новый пароль и его подтвердить.

Понятие процесса в среде ОС

Процесс – некоторая деятельность в среде операционной системы. Например – выполняющаяся программа. Следует различать поток от процесса. Поток – деятельность внутри процесса.

Для просмотра информации о процессах служит команда ps.

Полная информация о процессах выводится после подачи команды

ps –ef.

Процессы в ОС Linux имеют специальный атрибут – статус. Для просмотра статуса процессов подают команду:

ps ax.

Просмотр процессов пользователя выполняется командой:

ps –U имя_пользователя

В таблице 12.1 приводится информация о статусе процессов.

Таблица 12.1

Атрибуты статуса процесса

Статус процесса	Описание процесса
S	Спящий. Ждет появления события, такого как сигнал, или активизации ввода
R	Выполняющийся. Это процесс, стоящий в очереди на выполнение, либо выполняющийся, либо готовый к выполнению
D	Непрерывно спящий. Процесс ждет завершения ввода или вывода
T	Остановленный. Остановленный системой управления заданиями командной оболочки или находящийся под контролем отладчика
Z	Умерший. Процесс «Зомби»
N	Задача с низким приоритетом
s	Ведущий процесс сеанса
+	Процесс в группе фоновых процессов
l	Многопотоковый процесс
<	Задача с высоким приоритетом

Каждый процесс имеет свой уникальный идентификатор PID (Process Identificator). Операционная система Linux связывает системное имя с идентификатором пользователя в системе – UID (User ID). UID – это положительное целое число, по которому система и отслеживает пользователей. Значение UID позволяет определить пользователя, который вызвал к жизни процесс. Кроме того характеристикой процесса является значение PPID (Process Parent ID) – идентификатор процесса – родителя (предка) данного процесса.

Узнать UID можно командами:

id – код текущего пользователя;

id имя_пользователя – код конкретного пользователя.

Приведем примеры использования команды ps.

Пример №1. Краткий вывод:

```
use@ubuntu:~$ ps
  PID   TTY      TIME       CMD
 2083   pts/0    00:00:00   bash
 2581   pts/0    00:00:00   ps
```

Здесь CMD – имя команды, программы породившей процесс, TIME.

Пример №2. Полная форма: ps -ef

```
UID PID  PPID  C STIME TTY    TIME        CMD
root 1   0     0 01:38  ?    00:00:01    /sbin/init
root 2   0     0 01:38  ?    00:00:00    [kthreadd]
root 3   2     0 01:38  ?    00:00:00    [ksoftirqd/0]
root 4   2     0 01:38  ?    00:00:00    [kworker/0:0]
root 5   2     0 01:38  ?    00:00:00    [kworker/u:0]
root 6   2     0 01:38  ?    00:00:00    [migration/0]
root 7   2     0 01:38  ?    00:00:00    [cpuset]
root 8   2     0 01:38  ?    00:00:00    [khelper]
root 9   2     0 01:38  ?    00:00:00    [netns]
```

Здесь STIME – начало функционирования процесса.

Пример №3. Статус процессов: ps ax

```
PID  TTY      STAT   TIME     COMMAND
 1   ?        Ss     0:01     /sbin/init
 2   ?        S      0:00     [kthreadd]
 3   ?        S      0:00     [ksoftirqd/0]
 4   ?        S      0:00     [kworker/0:0]
 5   ?        S      0:00     [kworker/u:0]
 6   ?        S      0:00     [migration/0]
```

```
7   ?      S<     0:00   [cpuset]
8   ?      S<     0:00   [khelper]
9   ?      S<     0:00   [netns]
```

Приоритеты процессов

Приоритет процесса – nice («Привлекательность»). По умолчанию он равен нолю, изменяется системой. При параллельной работе нескольких процессов тому процессу, у которого значение nice больше отдается меньше процессорного времени. Такой процесс будет обрабатываться ЭВМ дольше, чем остальные. Это значение может изменяться в пределах от -20 до +19 [7]. По умолчанию значение nice равно нулю.

Назначить приоритет, порождаемым по команде, процессам можно командами:

nice –число_nice команда
nice –n число_nice команда

Пользователи системы могут только понижать приоритет своих процессов, задавая значения от 0 до 19.

Суперпользователь системы может повышать и понижать приоритет своих задач.

Пример. Первая команда создает процесс с приоритетом 9, а вторая – с приоритетом -10.

nice -9 команда
nice --10 команда

Допускается изменять приоритет уже активных процессов системы командами:

renice число_nice PID
renice –n число_nice опции

Опции:

-p PID – изменение приоритета заданного процесса;

-u имя_пользователя – изменение приоритета процессов пользователя.

Пример. Изменение приоритета для процесса по его PID:

renice +7 5889

Чтобы вывести на экран терминала список процессов с указанием их nice значения, нужно использовать команду ps с ключом –l.

Уничтожение процесса

Для взаимодействия с процессами используется команда kill. Команда посылает процессу сигнал [8]. Прервать процесс можно командами:

```
kill PID
kill –KILL PID
```

Во втором случае процессу подается сигнал KILL – прекратить работу.

Фоновые процессы

Общий формат запуска фонового процесса имеет вид:

```
команда_запуска &
```

После запуска процесса можно продолжать запускать в терминале новые процессы – программы.

Пример. Фоновой процесс выводящий сообщение Process active ! с интервалом в 20 секунд. Процесс реализуем в виде скрипта tpr.sh:

```
#Bash
while true; do
  echo "Process activ !"
  sleep 20
done
```

Запуск скрипта в фоновом режиме:

```
$./tpr.sh &
```

На фоне этого процесса можно продолжать работу в терминале и через каждые 20 секунд работа будет прерываться выводом сообщения.

Взаимодействие процессов. Сигналы

Сигнал – событие, асинхронно пересылаемое программе. В операционной системе Linux существует стандартный набор сигналов, которые можно передать процессу. Каждый сигнал имеет определенное условное обозначение.

При написании сценариев часто используют сигнал INT, который возникает при прерывании работы сценария.

Этот сигнал посылается командой – сочетанием клавиш CTRL + C.

Для перехвата сигнала в коде скрипта используют команду trap:

trap команда signal

С помощью данной команды назначают действие, которое нужно выполнить при возникновении сигнала (signal).

Пример. Сценарий, выполняющий перехват сигнала INT. При появлении сигнала значение переменной job становится пустым и цикл while будет прерван, на терминал будет выведено содержание файла sys_info. Сценарий должен быть запущен в фоновом режиме:

```
trap 'job=' INT
job="ok"
uname -r > sys_info
while test -n "$job"
do
 echo $job
 echo "Press CTRL + C"
done
echo "Done !"
cat sys_info
exit 0
```

Запуск нового процесса программным кодом

При разработке программ для ОС Linux, в которых требуется порождать новые процессы, следует использовать язык программирования Си. Заметим, что данный язык является системным средством программирования в ОС Linux [8].

Так, для запуска копии терминала для выполнения команды служит функция system. Формат функции:

```
stdlib.h
int system (const char *string);
```

Здесь string – строка представляющая собой строку запуска программы (команды Linux).

Коды возврата:

-1 Ошибка.
127 – оболочку – терминал нельзя запустить.

Пример:

```
char command[] = "ls -l";
  if (system(command) == -1){
```

```
   perror("Ошибка запуска оболочки");
   exit(1);
  }
exit(0);
```

Системный вызов запускает командную оболочку для выполнения команды просмотра содержимого директории. Процесс – оболочка ждет завершения выполнения команды.

Семейство вызовов exec

Организовать запуск процесса без запуска копии терминала – оболочки можно, использовав функции execl(), execlp() и execle() [8].

Последняя буква в названии функции определяет правила ее использования:

p – функция выполняет поиск пути для запуска программы.

l – можно задать список аргументов для запускаемой программы.

e – при запуске программы используются переменные окружения операционной системы.

Функции принимают переменное число аргументов. Последний аргумент – нулевой указатель (NULL). Можно использовать также функции execv() и execvp(). Они требуют задать аргумент – массив для указания параметров запуска программы.

Пример. Функция execlp().

```
unistd.h
int execlp(const char *file,const char *arg0,
...,(char *)0);
```

Первый аргумент – имя файла программы, которая содержит исполняемый код. Второй аргумент – команда запуска, далее нужно указать параметры для запуска.

По умолчанию файл с кодом ищется в каталогах, которые находятся в системной переменной окружения PATH. Системный вызов exec заменяет вызывающий процесс новым.

Воспользуемся этой функцией для запуска процесса, который представляет собой программу ОС Linux ps. Код запуска примет вид:

```
puts("Просмотр nice процессов !");
execlp("ps","ps","-l",NULL);
```

Разветвление процесса (клонирование)

Клонирование – разделение исходного процесса на два с помощь системного вызова fork. Реализуется вызов одноименной функцией:

```
sys/types.h
unistd.h
pid_t fork(void);
```

Возвращает системный вызов идентификатор процесса. Схема системного вызова показана на рис. 12.1.

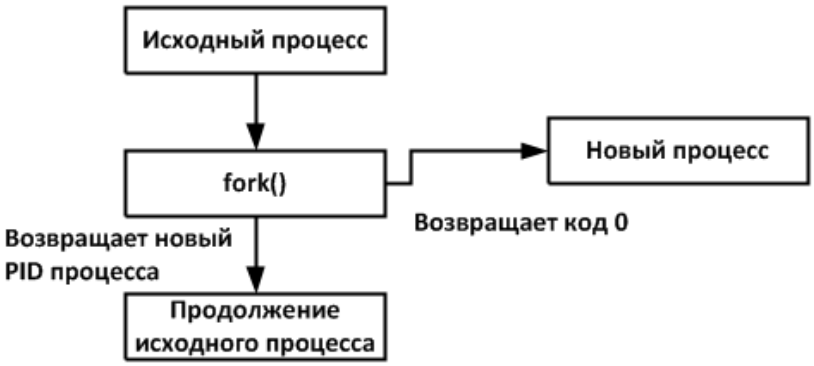

Рис. 12.1. Клонирование процесса

Исходный процесс получает новый PID, а его клон получает PID равный нулю.

С системным вызовом fork связано такое явление, как процесс «Зомби». Если клонированный процесс закончится раньше исходного, то исходный процесс получит статус Z «Зомби процесса». Так как о нем нет сведений в таблице процессов ОС, то такие процессы уничтожаются системным процессом init.

Структура программы с системным вызовом fork() примет вид:

```
//Деятельность исходного процесса

//Клонирование – разделение процесса
pid_t pid=fork();

if (pid == -1){
  perror("Ошибка вызова fork !");
  exit(1);//Аварийное завершение программы
}
if (pid != 0) {
  //Деятельность родительского процесса
```

```
}
else{
  //Деятельность нового процесса
}
```

Каналы

Канал – позволяет обмениваться байтами между процессами. Каналы могут быть неименованными и именованными.

Для использования неименованного канала использую конвейер. Условное обозначение | для конвейера.

Пример №1. Постраничный просмотр содержания файла большого объема с помощью программы ОС more:

```
cat fasm.txt |  more
```

Пример №2. Упорядочивание вывода в файл и постраничный просмотр на экране терминала. Использована программа ОС sort.

```
ps | sort > listing.ps
ps | sort | more
```

Именованный канал создаётся командой:

```
mkfifo имя_файла
```

Канал работает по принципу: FIFO (First Input First Output – Первый пришел, первый вышел).

Организация взаимодействия работы в двух терминалах происходит следующим образом. В первом терминале создается именованный канал и выполняется обращение к каналу командой cat.

```
mkfifo myFIFO
cat myFIFO
```

Во втором терминале нужно записать байты в канал:

```
echo "Test FIFO" > myFIFO
```

Те же действия можно выполнить в одном терминале, использовав фоновый режим работы. Канал в фоновом режиме:

```
$ cat myFIFO &
[1] 2721
$ echo "Test FIFO" > myFIFO
$ Test FIFO
```

Нажатие комбинации клавиш CTRL+C – завершение фонового процесса:

```
[1]+  Готово cat myFIFO
$
```

Контрольные вопросы и задания

1. Как выполнить переход в режим супервизора в ОС Linux?
2. Что такое статус процесса?
3. Дайте описание таким характеристикам процесса, как PID, PPID и UID.
4. Какая команда используется для вывода информации о процессах?
5. Что показывает значение nice процесса?
6. Как изменить приоритет процесса?
7. Как выполнить запуск программы в терминале в фоновом режиме?
8. Как выполняется перехват сигналов в файле сценария?
9. Как выполнить посылку сигнала процессу в терминале?
10. Как выполняется системный вызов system в программе?
11. Как выполняется системный вызов fork?
12. Как выполняют системный вызов exec в программе?
13. Как используются неименованные каналы и процессы?
14. Как использовать именованный канал для взаимодействия с процессом?

Лекция №13. СИСТЕМНЫЕ ВЫЗОВЫ Linux в АССЕМБЛЕРНОЙ ПРОГРАММЕ

Структура Ассемблерной программы, выполнение системного вызова, создание файлов, чтения байтов из файла, запись байтов в файл, работа с файлами, обслуживание каталогов.

Программа формат ELF

Выполнить системный вызов означает обратиться к ядру ОС. При этом возникает прерывание, для обработки которого требуется вызов системной функции.

Создать программу, использующую системный вызов, можно путем использования языка Ассемблера. При написании программы на языке Ассемблера для операционной системы Linux ее текст должен быть написан в соответствии с форматом ELF (Executable and Linkable Format). Поэтому при написании кода программы для компилирования ее с помощью Ассемблерного компилятора нужно придерживаться правильной структуры программы с точки зрения ОС.

Рассмотрим правила создания и компилирования Ассемблерных программ с помощью свободно распространяемого компилятора FASM (Flat Assembler). Создавая программу ELF с помощью компилятора FASM, программист должен учитывать следующие правила:

Задать директиву format компилятору для указания типа создаваемого программного кода.

Указать «точку» входа entry. Точка входа – это метка – адрес первой исполняемой команды процессором. С этой команды начинается выполнение команд программы:

Задать сегмент (segment) команд программы.

Задать сегмент (segment) кодов – данных программы.

С учётом этих правил структура ассемблерной программы примет вид:

```
format ELF executable 3
entry start

segment readable executable
start:
 ;Исполняемые команды программы

segment readable writeable
 ;Код данных
```

Для указания формата ELF программы используется директива format с ключевыми словами ELF executable. Цифра 3 – код операционной системы Linux.

При задании сегмента нужно указать, как будут использованы его коды процессором. Для сегмента кодов команд указывают ключевые слова – атрибуты: readable (содержание сегмента можно считывать), executable (коды сегмента – коды команд процессора).

Для сегмента кодов данных программы используют атрибуты readable и writeable. Последний атрибут означает, что команды программы могут как считывать коды, так и изменять их значения.

В приведённом шаблоне программы точка входа называется start. Это название выбирает программист.

Системный сервис

Обращение к сервису – обращение к ядру операционной системы Linux для выполнения определённого действия. К таким действиям относятся организация ввода-вывода в среде ОС и завершение работы программы.

Для реализации программой любого системного действия она должна вызвать соответствующую функцию ядра ОС. В операционной системе Linux функции ядра обладают определённым номером и именем. При этом говорят, что для выполнения действия в ОС программа выполняет именованный системный вызов.

В Ассемблерной программе системный вызов выполняется по следующим правилам:

- в регистр EAX заносится номер функции;
- если функция для своей работы требует задания определённых кодов, то они последовательно заносятся в регистры EBX, ECX, EDX, ESI, EDI;
- затем следует вызвать прерывание с номером 80h для выполнения выбранной функции с помощью команды INT.

Результат работы функции возвращается в регистре EAX. Если регистр хранит отрицательное значение, то функция выполнилась с ошибкой.

Для вывода сообщения на экран – консоль следует использовать системный вызов write. Номер этого вызова 4.

Обращение:

EAX = 4
EBX = 1
ECX = адрес буфера вывода
EDX = число выводимых байтов

В регистр EBX нужно записать номер (дескриптор) потока вывода. В операционной системе Linux консоль экран называется устройством стандартного ввода-вывода. За этим устройством закреплён дескриптор с номером 1. Это поток стандартного вывода.

Заметим, что для вывода на экран можно использовать дескриптор с номером 2. Это номер устройства «стандартной ошибки». Вывод по этому дескриптору выполняют при выводе диагностических сообщений об ошибке в программе.

Адрес буфера вывода должен содержать адрес в сегменте данных участка памяти, где находятся байты, подлежащие выводу.

Для выполнения ввода с консоли используется системный вызов read. Это системный вызов с номером 3.

Формат вызова:

EAX = 3
EBX = 0
ECX = адрес буфера
EDX = число символов

В регистр EBX нужно записать код 0. Это номер дескриптора стандартного потока ввода с консоли. В регистр ECX записывают адрес буфера в сегменте данных, куда будут записаны вводимые байты. В регистр EDX записывают ожидаемое число вводимых байтов.

После вызова функции пользователь должен ввести символы с клавиатуры, завершается ввод нажатием комбинации клавиш CTRL + D. Такая комбинация в ОС Linux вводит в стандартный поток ввода признак конца файла и ввод прекращается. В регистре EAX будет храниться число введённых байтов.

Для завершения работы программы и передачи управления операционной системе нужно выполнить системный вызов exit с номером 1.

Вызов этой функции требует записать в регистры коды:

EAX = 1
EBX = код

Здесь код – код завершения работы программы. Если программа не передаёт код ОС для последующего его анализа, то это значение принимают равным нолю.

Открытие файла

Для выполнения операций чтения байтов из файла либо записи байтов в файл программа должна открыть файл, получив его дескриптор. Дескриптор – целое число, которое определяет поток вывода (ввода) байтов для физического файла, открытого программой. Получают дескриптор системным вызовом open с номером 05.

Системный вызов open требует задания следующих значений:

- filename – имя файла, записанное в ASCIIZ формате. Последний байт в имени файла должен хранить код ноль;
- flags – целое число, активные биты которого определяют режим использования файла;
- mode – маска прав доступа к файлу.

После выполнения системного вызова в регистре EAX будет записано целое число, дескриптор открытого – созданного файла. При неудачном выполнении в регистре хранится код -1.

При задании имени файла его следует завершить кодом ноль. ***Пример*** правильного задания имени файла в сегменте данных программы:

```
filename db 'test.txt',0
```

Режим использования файла кодируется значениями, которые приведены в табл. 13.1 [8].

Таблица 13.1

Режимы открытия файла

Номер	Условное обозначение	Код	Значение
1	O_RDONLY	000h	Чтение
2	O_WRONLY	001h	Запись
3	O_RDWR	002h	Чтение и запись
4	O_CREAT	040h	Разрешить создание файла
5	O_EXCL	080h	Потребовать создание файла
6	O_TRUNC	200h	Если файл существует, уничтожить его содержание
7	O_APPEND	400h	Дополнить файл новым содержанием

Режимы доступа

Режим №5 может использоваться совместно с режимом №4, это гарантирует, что программа обязана создать файл.

Задавая код доступа, используют комбинацию режимов. Для этого служит операция битового сложения OR. Например:

O_RDWR or O_CREAT

Фактически операция OR приводит к сложению кодов режимов. Рассмотренному набору кодов будет соответствовать число 042h. Его можно задать как константу в сегменте данных программы:

flags = 0x042

Маска прав

Задание прав доступа к файлу должно выполняться с учётом маски пользовательских прав операционной системы Linux! [7,8].

Маска пользовательских прав – это восьмеричное число, которое по умолчанию назначено пользователю при его входе в систему. Число трёхразрядное.

Первая цифра задаёт ограничение прав пользователя.

Вторая цифра определяет ограничения для группы, в которую входит пользователь.

Третья цифра – ограничение прав остальных пользователей.

Узнать права доступа можно с помощью команды терминала umask.

Права доступа приводятся в табл. 13.2.

Таблица 13.2

Права доступа – ограничения в системе

Номер	Значение	Описание
1	0	Наличие всех прав
2	4	Нет права чтения данных (r)
3	2	Нет права на запись данных (w)
4	1	Нет права на исполнение программ (x)

Обычно пользователю предоставляется маска прав со следующим кодом 022. Это означает, что у пользователя есть все права, группа и остальные пользователи могут читать данные из файлов и запускать программы.

Реальные права доступа к файлу будут получены в результате сопоставления с имеющейся маской прав доступа в ОС, по умолчанию:

mode and not umask

mode – желаемые права доступа к файлу

По умолчанию mode задают в виде 666:

mode = 0666o

То есть пользователю, группе пользователя и остальным делается попытка назначить маску прав доступа к файлу в виде:

rw-rw-rw-.

Тогда, если значение umask равно 022, то получим в двоичной системе:

not
000010010
=
111101101

Создаваемый файл получит реальные права доступа:

110110110
and
111101101
=
110100100
=
644_8

Этому коду соответствует маска прав доступа к файлу:

rw-r–r–

Операцию получения маски прав доступа можно получить в более компактном виде, используя операцию XOR. Тогда если значение umask равно 022, то получим в двоичной системе:

mode XOR umask

110110110
XOR
000010010
=

110100100
=
644_8

Таким образом, полное задание в сегменте данных программы спецификации файла для вызова open требует задания имени файла, кода режима доступа и маски прав доступа.

Например:

```
fname db 'test.txt',0;Имя файла
flags = 241h;O_TRUNC+O_CREAT+O_WRONLY
mode  = 0666o;Маска доступа
```

В сегмент кода программы следуют записать команды открытия файла:

```
mov eax,5;Создать файл
mov ebx,fname
mov ecx,flags
mov edx,mode
int 80h
push eax;Дескриптор в стек
```

Закрытие файла

Чтобы зарыть файл, нужно выполнить системный вызов close. Код вызова равен 06. Для выполнения вызова используется один параметр – дескриптор открытого файла. При ошибке закрытия файла EAX возвращает значение равное -1.

Запись байтов в файл, чтение байтов из файла

Запись байтов выполняется системным вызовом write. Для выполнения записи системному вызову нужно передать требуемый файловый дескриптор.

Для выполнения операции чтения файл должен открываться системным вызовом open. При этом маска доступа не задаётся. Вместо **неё нужно задать код ноль**!

После открытие файла с режимом чтения можно выполнить чтение байтов файла с помощью системного вызова read по его дескриптору.

Пример. Команды чтения байтов из файла.

```
go:
;==========================
  mov eax,3  ;Читать из файла – вызов read
  pop ebx    ;Получить дескриптор файла из стека
  mov ecx,buf;Адрес буфера
  mov edx,1  ;Читать один байт
  push ebx   ;Дескриптор в стек
  int 80h    ;Читать байт
  cmp eax,0  ;Все байты?
  je fin_read;Да, на выход
;==========================
  mov ecx,buf;Прочитанный байт
  mov edx,1  ;Один байт
  call get_mes;Вывести байт
loop go      ;Продолжить чтение
;==========================
fin_read:
;Закрыть файл
```

Для компактной записи кода команды вывода прочитанного байта на консоль собраны в процедуру get_mes, которая вызывается командой call.

Произвольный доступ к файлу

Технология произвольного доступа предполагает прямой доступ к байтам файле при их чтении или записи. Выполняется произвольный доступ с помощью системного вызова lseek с номером 19.

Параметры системного вызова:

- fd – файловый дескриптор для указания файла прямого доступа;
- offset – смещение к байту или блоку байтов;
- origin – константа, которая задает правила отсчёта смещения (табл. 13.3).

Операции с файлами

Рассмотрим некоторые базовые операции для работы с файлами. Удаление файла выполняется с помощью системного вызова unlink с номером 10.

Для выполнения системного вызова нужно задать имя удаляемого файла. Файл должен быть закрыт. При ошибке удаления регистр EAX возвращает значение -1.

Таблица 13.3

Правила отсчета смещения файла

Код	Значение	Описание
0	SEEK_SET	Смещение вычисляется от начала файла
1	SEEK_CUR	Смещение вычисляется от текущей позиции указателя в файле
2	SEEK_END	Смещение вычисляется от конца файла

Переименование файла выполняется системным вызовом rename с номером 38.

Выполнение вызова требует задания двух параметров:

- существующее имя файла;
- новое имя файла.

Файл, имя которого меняется, должен существовать и быть закрытым. При ошибке EAX возвращает значение -1.

Работа с каталогами

При использовании системных вызовов для каталогов следует помнить, что имена каталогов задаются так же, как и имена файлов в ASCIIZ формате. В случае ошибки системные вызовы возвращают в регистре EAX код -1.

Создание каталога выполняется системным вызовом mkdir с номером 39. Для выполнения системного вызова нужно задать:

- dirname – имя каталога;
- mode – права доступа к каталогу.

Права доступа к каталогу – это код, который задаётся по тем же правилам, что и при создании файлов.

Права доступа к каталогу обычно задают в виде восьмеричного значения 777. С учётом стандартный маски пользователя 022 получим права доступа:

000010010
XOR
111111111
=
111101101

Таким образом, каталог получит маску прав доступа:
755 (rwxr-xr-x)

Смена каталога выполняется с помощью вызова chdir с номером 12. При выполнении вызова нужно иметь в виду, что программа, выполнившая этот вызов, представляет собой процесс. Смена каталога касается только этого процесса. После успешного завершения вызова для программы текущим будет каталог, указанный в вызове.

Когда программа завершится, текущим будет тот каталог, который был открыт в момент запуска программы.

Вызов требует задания только одного параметра – имени каталога.

Удаление каталога выполняется системным вызовом rmdir с кодом 40. Выполнение вызова требует задания имени каталога. Каталог должен быть пустым.

Контрольные вопросы и задания

1. Какую структуру имеет Ассемблерная программа для формата ELF?
2. Как выполняется системный вызов в Ассемблерной программе?
3. Как выполняется задание режимов работы с файлом для системного вызова open?
4. Как назначить права доступа к файлу при использовании системного вызова open?
5. Какие реальные права доступа будут назначены файлу после выполнения системного вызова open?
6. Для чего используется файловый дескриптор в Ассемблерной программе?
7. Как используют стандартные потоки ввода – вывода в программах?
8. Как в Ассемблерный программе реализуется прямой доступ к файлу?

9. Приведите примеры системных вызовов для обслуживания фалов?

10. Приведите примеры системных вызовов для обслуживания директорий.

Лекция №14. ОПЕРАЦИОННАЯ СИСТЕМА Linux, РАБОТА С ФАЙЛАМИ И ДАННЫМИ

Поиск файлов, анализ содержания файлов, регулярные выражения.

Чтобы отыскать файл, ОС Linux предоставляет пользователю системы специальную программу – команду find [5].

Формат команды:

find [путь] [опции] критерий [действия]

Обязательным является критерий поиска, который задается в виде:

–name “имя_шаблон”

Здесь шаблон – маска поиска файлов.

Приведем ряд примеров.

Пример №1. Поиск в текущем каталоге файлов, которые не имеют расширения и их имя состоит из двух символов:

$ find./test -name “??”

Результат поиска:

./test/f1
./test/f2
./test/f3

Пример №2. Поиск в текущем каталоге файлов, которые не имеют расширения и их имя начинается на букву s.

$ find “s*”

Результат поиска:

sig2.c
spiski.o
stringp.c
stringp.o

Выполнить расширенный поиск файлов можно, задавая опции, приведенные в табл. 14.1.

Кроме того, поиск можно расширить введя дополнительные критерии, приведенные в табл. 14.2.

Таблица 14.1

Опции поиска

Опция	Описание
-depth	Поиск в подкаталогах перед поиском в самом каталоге
-follow	Следовать по символическим ссылкам
-maxdepths N	При поиске проверять не N вложенных уровней каталога
-mount	Не искать в каталогах других файловых систем

Отыскивая файлы определенного типа, нужно задать определенный тип в виде буквы. Допустимые значения приведены в табл. 14.3.

Таблица 14.2

Формирование критерия

Критерий	Описание
-atime N	К файлу обращались последние N дней назад
-mtime N	Файл последний раз изменялся N дней назад
-name шаблон	Имя или шаблон с использованием символов * или ?
-newer другой_файл	Текущий файл, измененный позже, чем другой файл
-type c	Поиск файла определенного типа
-user имя_пользователя	Поиск файлов по владельцу.

Таблица 14.3

Типы объектов файловой системы

Тип	Объект
d	Директория
f	Файл
l	Символическая ссылка
p	Именованный канал

Рассмотрим пример поиска в директории test-директорий, имена которых состоят из двух символов. Вызов программы find примет вид:
$ find./test -name “??” -type d

Результат поиска:

./test/d2
./test/d1

Допускается объединение критериев с помощь операторов, приведенных в таблице 14.4.

Таблица 14.4

Объединение критериев

Оператор	Описание
!	Инвертирование критерия
-a	Оба критерия должны быть истинны
-o	Один из критериев должен быть истинным

Рассмотрим пример объединения критериев. Требуется отыскать все файлы в текущей директории, в которые внесли изменения позже, чем в файл f1.

Команда поиска примет вид:

```
/test$ find \(-name "*" -a -newer f1 \) -type f
```

Результат поиска:

./f2
./f3
./file1

Группировка критериев требует наличия экранированных круглых скобок символом \.

Существует возможность указать в команде поиска определенное действие, которое нужно выполнить, если обнаружен объект поиска. Для этого предусмотрены специальные ключи, показанные в табл. 14.5.

Таблица 14.5

Выполнение команд

Действия	Описание
-exec команда	Выполнение команды
-ok команда	Выполнение команды
-print	Вывод на экран имени файла
ls	Команда ls –dils к текущему файлу

Укажем ряд особенностей задания команд для исполнения:

- команда должна заканчиваться парой символов \;
- OK режим требует подтверждения для выполнения команды ввод y(es);
- символы {} – формальный параметр, заменяется полным путем к текущему файлу.

Примеры команд

Пример №1. В текущей директории нужно отыскать все файлы, имена которых состоят их двух символов, и выполнить команду ls -l для каждого найденного файла:

```
~/test$ find -name "??" -type f -exec ls -l  {} \\;
-rw-r--r-- 1 use use 0 2015-05-10 00:43./f1
-rw-r--r-- 1 use use 0 2015-05-10 00:43./f2
-rw-r--r-- 1 use use 3 2015-05-10 02:12./f3
```

Пример №2. Те же действия, что и в примере №1, запрос на исполнение команды ls:

```
~/test$ find -name "??" -type f -ok ls -l  {} \\;
< ls..../f1 > ? y
-rw-r--r-- 1 use use 0 2015-05-10 00:43./f1
< ls..../f2 > ? y
-rw-r--r-- 1 use use 0 2015-05-10 00:43./f2
< ls..../f3 > ? y
-rw-r--r-- 1 use use 3 2015-05-10 02:12./f3
```

Команда grep

Выполнить поиск по содержанию текстовых файлов можно с помощью команды – программы grep. Команда получила свое название от сокращения GREP (General Regular Expression Parser) [5].

Общий формат команды имеет вид:

grep [опции] шаблон файл

Опции поиска приведены в таб. 14.6.

Для тестирования работы команды воспользуемся файлом listing_dir, который содержит информацию о директории, в которой было создано 12 файловых объектов.

Содержание файла:

```
drwxr-xr-x 2 use use 4096 2014-05-10 00:52 d1
drwxr-xr-x 2 use use 4096 2014-05-10 00:52 d2
-rw-r--r-- 1 use use    0 2014-05-10 00:43 f1
-rw-r--r-- 1 use use    0 2014-05-10 00:43 f2
-rw-r--r-- 1 use use    3 2014-05-10 02:12 f3
-rw-r--r-- 1 use use    0 2014-05-10 00:43 file1
-rw-r--r-- 1 use use    0 2014-05-10 02:36 listing_dir
```

Пример простого анализа содержания файла:

```
grep drwx listing_dir
```

Обязательным параметром является шаблон поиска и имя файла. В данном случае следует отыскать строки, в которых хранится сочетание символов drwx.

Результат поиска:

```
drwxr-xr-x 2 use use 4096 2014-05-10 00:52 d1
drwxr-xr-x 2 use use 4096 2014-05-10 00:52 d2
```

Покажем пример использования опций. Пусть требуется отыскать строки с сочетанием символов drwx и нужно вывести количество найденных строк.

Таблица 14.6

Опции поиска содержания

Опция	Описание
-c	Вывод числа найденных строк по шаблону
-E	Использование расширенных регулярных выражений
-i	Не учитывать регистр символов
-l	Перечисление имен файлов с данными по шаблону
-v	Вывод не совпадающих строк по шаблону

Вызов команды и результат поиска примут вид:

```
~/test$ grep -c drwx listing_dir
2
```

Другой пример. При поиске не нужно учитывать регистр символов в шаблоне. Вызов команды и результат поиска примут вид:

```
~/test$ grep -i Drwx listing_dir
drwxr-xr-x 2 use use 4096 2014-05-10 00:52 d1
drwxr-xr-x 2 use use 4096 2014-05-10 00:52 d2
```

Приведем пример реверсивного поиска. Требуется вывести строки, которые не совпадают с заданным поисковым шаблоном:

```
~/test$ grep  -v drwx listing_dir
-rw-r--r-- 1 use use    0 2014-05-10 00:43 f1
-rw-r--r-- 1 use use    0 2014-05-10 00:43 f2
-rw-r--r-- 1 use use    3 2014-05-10 02:12 f3
-rw-r--r-- 1 use use    0 2014-05-10 00:43 file1
-rw-r--r-- 1 use use    0 2014-05-10 02:36 listing_dir
```

Команда позволяет получить список файлов, в которых содержатся строки с заданным шаблоном. Для этого нужно вызвать команду с ключом –l, а в качестве имени файла указать шаблон * – искать во всех файлах текущей директории.

Команда и результат поиска:

```
~/test$ grep -l drwx *
listing_dir
test.lst
```

Регулярные выражения

Более сложные варианты поиска символьной информации возможны при использовании ***регулярных выражений***. В табл. 14.7 приводятся специальные символы, для формирования таких выражений [11].

Таблица 14.7

Специальные символы

Символ	Описание
^	Привязка к началу строки
$	Привязка к концу строки
.	Любой символ
[]	Задание диапазона символов

Приведем особенности использования специальных символов:

- символы при использовании должны экранироваться. Например, \$;
- диапазон фиксированный: abc;
- диапазон большого размера: a-z;
- инвертированный диапазон ^abcdefg;

- проверка наличия пробела [:space:].

Регулярные выражения могут формироваться на основе ряда дополнительных символов – опций, приведенных в табл. 14.8.

Таблица 14.8

Дополнительные символы

Опция	Описание
?	Совпадение не обязательно, возможно не более одного раза
*	Совпадение может не быть, может быть однократным или многократным
+	Совпадение должно быть однократным или многократным
{n}	Совпадение должно быть n раз
{n, }	Совпадение n раз или более
{n, m}	Совпадение должно быть от n до m раз включительно

Пример использования регулярных выражений для файла listing_dir:

```
drwxr-xr-x 2 use use 4096 2014-05-10 00:52 d1
drwxr-xr-x 2 use use 4096 2014-05-10 00:52 d2
-rw-r--r-- 1 use use    0 2014-05-10 00:43 f1
-rw-r--r-- 1 use use    0 2014-05-10 00:43 f2
-rw-r--r-- 1 use use    3 2014-05-10 02:12 f3
-rw-r--r-- 1 use use    0 2014-05-10 00:43 file1
-rw-r--r-- 1 use use    0 2014-05-10 02:36 listing_dir
```

При использовании регулярных выражений требуется отыскать строки, которые заканчиваются символом 1. Команды с результатом поиска примут вид:

```
~/test$ grep 1$ listing_dir
drwxr-xr-x 2 use use 4096 2014-05-10 00:52 d1
-rw-r--r-- 1 use use    0 2014-05-10 00:43 f1
-rw-r--r-- 1 use use    0 2014-05-10 00:43 file1
```

Поиск строк, которые содержат лексему, отвечающую следующим правилам: первый символ произвольный, второй символ 2, далее три произвольных символа, затем следуют пробелы. Команда и результат поиска примут вид:

```
~/test$ grep.2...[[:space:]]  listing_dir
-rw-r--r-- 1 use use    3 2014-05-10 02:12 f3
-rw-r--r-- 1 use use    0 2014-05-10 02:36 listing_dir
use@ubuntu:~/test$
```

При поиске по диапазону символов нужно найти строки, где имеются символы f и l (латинская буква эл). Команда и результат поиска:

```
~/test$ grep [fl] listing_dir
-rw-r--r-- 1 use use    0 2014-05-10 00:43 f1
-rw-r--r-- 1 use use    0 2014-05-10 00:43 f2
-rw-r--r-- 1 use use    3 2014-05-10 02:12 f3
-rw-r--r-- 1 use use    0 2014-05-10 00:43 file1
-rw-r--r-- 1 use use    0 2014-05-10 02:36 listing_dir
```

Расширенный синтаксис

Дополнительные символы, опции из табл. 14.8 называют расширенным синтаксисом регулярного выражения. Рассмотрим в качестве примера файл с информацией о директории:

```
drwxr-xr-x 2 use use 4096 2014-05-10 00:52 d1
drwxr-xr-x 2 use use 4096 2014-05-10 00:52 d2
-rw-r--r-- 1 use use    0 2014-05-10 00:43 f1
-rw-r--r-- 1 use use    0 2014-05-10 00:43 f2
-rw-r--r-- 1 use use    3 2014-05-10 02:12 f3
-rw-r--r-- 1 use use    0 2014-05-10 00:43 file1
-rw-r--r-- 1 use use    0 2014-05-10 02:36 listing_dir
```

Сформируем следующий критерий поиска: нужно отыскать строки, в которых имеется сочетание двух символов 4 и 3.

Тогда вызов команды поиска примет вид:

```
use@ubuntu:~/test$ grep -E [43]\{2\} listing_dir
```

Результат поиска:

```
-rw-r--r-- 1 use use    0 2014-05-10 00:43 f1
-rw-r--r-- 1 use use    0 2014-05-10 00:43 f2
-rw-r--r-- 1 use use    0 2014-05-10 00:43 file1
```

При использовании расширенного синтаксиса нужно использовать ключ –E. Кроме того, специальные символы расширенного синтаксиса должны быть экранированы! Экранирование выполняется путем использование символа \.

Контрольные вопросы и задания

1. Для чего используется программа find?
2. Как назначаются шаблоны имен файлов для команды find?
3. Как выполнить поиск в каталоге объекта файловой системы определенного типа?

4. Как выполнить поиск объектов файловой системы по нескольким критериям?

5. Как выполнить требуемую команду при обнаружении объекта файловой системы?

6. Для каких целей используется программа grep?

7. Приведите примеры использования команды grep.

8. Как создают регулярные выражения, приведите примеры.

9. Какие возможности предоставляет расширенный синтаксис для регулярных выражений.

10. Как используют расширенный синтаксис для регулярных выражений при выполнении команды grep?

Лекция №15. РАСПРЕДЕЛЕННЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ

Распределенные вычислительные системы, характеристики распределенных вычислительных систем, сетевые и распределенные операционные системы.

Распределенные вычислительные системы характеризуются рядом особенностей.

От пользователей скрыты различия между компьютерами и способы связи между ними. То же самое относится и к внешней организации распределенных систем.

Должен быть механизм, при помощи которого пользователи и приложения единообразно работают в распределенных системах, независимо от того, где и когда происходит их взаимодействие.

Распределенные системы должны относительно легко поддаваться расширению, или масштабированию.

Пользователи и приложения не должны уведомляться о том, что части системы заменены, восстановлены или добавлены новые части для поддержки дополнительных пользователей или приложений.

Распределенная система – это набор независимых компьютеров, представляющийся их пользователям единой объединенной системой.

Основная задача распределенных систем – облегчить пользователям доступ к удаленным ресурсам и обеспечить их совместное использование, регулируя этот процесс.

Использование Интернет в настоящее время привело к появлению многочисленных виртуальных организаций, в которых географически удаленные друг от друга группы сотрудников работают вместе при помощи систем групповой работы.

По мере роста числа подключений и степени совместного использования ресурсов все более и более важными становятся вопросы безопасности:

- защита сетевых коммуникаций;
- фильтрация входящего потока данных;
- прозрачность.

Распределенные системы, которые представляются пользователям и приложениям в виде единой компьютерной системы, называются прозрачными. В табл. 15.1 рассмотрены преимущества прозрачных систем [4].

Таблица 15.1

Прозрачность

Свойство	Описание
1	2
Доступ	Скрывается разница в представлении данных и доступе к ресурсам
Местоположение	Скрывается местоположение ресурса
Перенос	Сокрытие факта переноса ресурса в другое место
Смена местоположения	Сокрытие факта переноса ресурса в другое место в процессе обработки
Репликация	Сокрытие дублирования, резервирования ресурса
Параллельный доступ	Сокрытие факта параллельного использования ресурса несколькими конкурирующими пользователями
Отказ	Сокрытие факта отказа и восстановления ресурса
Сохранность	Сокрытие факта нахождения ресурса в оперативной памяти или долговременной памяти

Открытая распределенная система – это система, предлагающая службы, вызов которых требует стандартный синтаксис и семантику. Правила работы служб определяются протоколами.

Службы определяются через интерфейсы. Интерфейсы описываются при помощи языка определения интерфейсов (Interface Definition Language, IDL).

Способность к взаимодействию характеризует, насколько две реализации систем или компонентов от разных производителей в состоянии совместно работать, полагаясь только на то, что службы каждой из них соответствуют общему стандарту.

Переносимость характеризует то, насколько приложение, разработанное для распределенной системы A, может без изменений выполняться в распределенной системе B, реализуя те же, что и в A интерфейсы.

Отделение правил от механизмов. При этом выделяют интерфейсы верхнего уровня, с которыми работают пользователи и приложения. Кроме того, выделяют интерфейсы внутренних модулей системы и описания взаимодействия этих модулей.

Система может быть масштабируемой по отношению к ее размеру, что означает легкость подключения к ней дополнительных пользователей и ресурсов.

Система может масштабироваться географически, то есть пользователи и ресурсы могут быть разнесены в пространстве.

Система может быть масштабируемой в административном смысле, то есть быть проста в управлении при работе во множестве административно независимых организаций.

Однако масштабируемость может иметь определенные границы [4]. Ограничения масштабируемости приведены в табл. 15.2.

Таблица 15.2

Ограничения масштабируемости

Концепция	Пример
Централизованные службы	Один сервер на всех пользователей
Централизованные данные	Общая база данных, доступная в режиме подключения
Централизованные алгоритмы	Организация маршрутизации данных на основе полной информации

Асинхронная связь. Приложение посылает запрос системе и, когда получает ответ, оно прерывает свою работу и вызывает специального обработчика для завершения отправленного ранее запроса.

Распределение по частям. Распределение предполагает разбиение компонентов на мелкие части и последующее разнесение этих частей по системе.

Кэширование. Результатом кэширования является создание копии ресурса, обычно в непосредственной близости от клиента, использующего этот ресурс. Это действие, предпринимаемое потребителем ресурса, а не его владельцем.

Мультипроцессорные системы

Мультипроцессорные системы: все процессоры имеют прямой доступ к общей памяти. В кэше сохраняются данные, обращение к которым происходит наиболее часто. Схема мультипроцессорной системы показана на рис. 15.1.

Если запрошенные данные находятся в кэш-памяти, то на запрос процессора реагирует она и обращения к шине не выполняются. Эффективность кэша – коэффициентом кэш-попаданий.

Рис. 15.1. Мультипроцессорная система с общей оперативной памятью

Если процессор А записывает слово в память, а процессор В микросекундой позже считывает слово из памяти, то память, обладающая таким поведением, называется согласованной.

Проблема мультипроцессорных систем шинной архитектуры состоит в их ограниченной масштабируемости, даже в случае использования кэша.

Проблема согласования памяти

В системах с несколькими процессорами и блоками памяти может быть использована коммутирующая решетка, показанная на рис. 15.2. Когда процессор желает получить доступ к конкретному модулю памяти, то соединяющие их узловые коммутаторы мгновенно открываются, организуя запрошенный доступ.

Недостатком коммутирующей решетки является то, что при наличии n процессоров и n модулей памяти потребуется n^2 узловых коммутаторов.

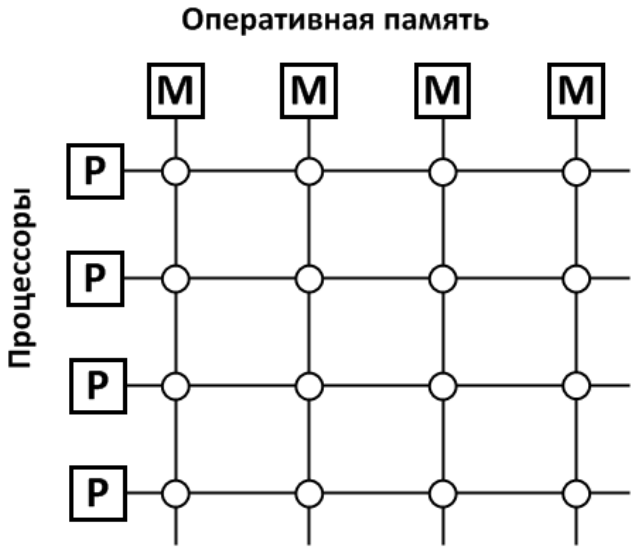

Рис. 15.2. Коммутационная решетка

Для организации более эффективного взаимодействия между процессорами и памятью ЭВМ используют омега-сеть, представленная на рис. 15.3.

Такая сеть содержит четыре коммутатора типа 2 x 2. Каждый из них имеет по два входа и два выхода. Каждый коммутатор может соединять любой вход с любым выходом.

Любой процессор может получить доступ к любому блоку памяти.

Однако следует иметь в виду, что такой способ коммутации имеет и отрицательную сторону. Недостаток коммутирующих сетей состоит в том, что сигнал, идущий от процессора к памяти или обратно, вынужден проходить через несколько коммутаторов.

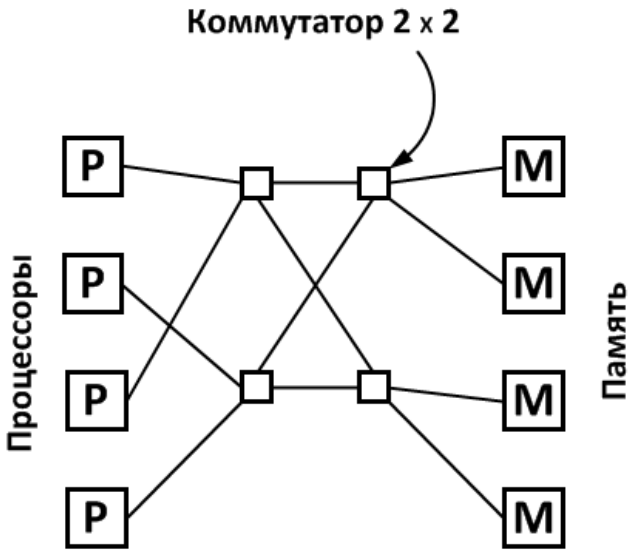

Рис. 15.3. Омега-сеть

Иерархические системы

В этом случае с каждым процессором ассоциируется некоторая область памяти. Каждый процессор может быстро получить доступ к своей области памяти. Доступ к другой области памяти происходит значительно медленнее.

Пример: машина с неунифицированным доступом к памяти (NonUniform Memory Access, NUMA).

Машины NUMA имеют лучшее среднее время доступа к памяти, чем машины на базе омега-сетей. Размещение программ и данных необходимо производить так, чтобы большая часть обращений шла к локальной памяти, при этом каждый процессор напрямую связан со своей локальной памятью.

Гомогенные системы

В таких вычислительных системах узлы монтируются в большой стойке и соединяются единой высокоскоростной сетью. В мультикомпьютерных системах с шинной архитектурой процессоры соединяются при помощи разделяемой сети множественного доступа. Скорость передачи данных в сети обычно равна 100 Мбит/с.

В коммутируемых мультикомпьютерных системах сообщения, передаваемые от процессора к процессору, маршрутизируются в со-

единительной сети в отличие от принятых в шинной архитектуре широковещательных рассылок.

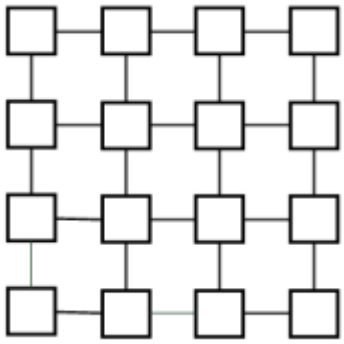

Рис. 15.4. Коммутационная решетка

Реализация таких систем выполняется на основе «коммутационной решетки» (рис. 15.4). Решётки просты для понимания и удобны для разработки на их основе печатных плат.

Гомогенные системы могут быть представлены также в виде гиперкуба. Гиперкуб представляет собой куб размерностью n.

Его можно представить в виде двух обычных кубов, с 8 вершинами и 12 ребрами каждый (рис. 5.15). Каждая вершина – это процессор. Каждое ребро – это связь между двумя процессорами. Для расширения гиперкуба к этой фигуре еще один комплект из двух связанных кубов, соединив соответствующие вершины двух половинок фигуры.

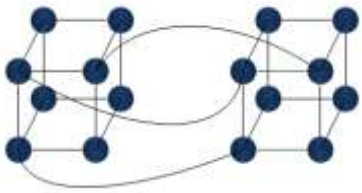

Рис. 15.5. Гиперкуб

Суперкомпьютеры и распределенные системы

Отнести вычислительную систему к классу суперкомпьютеров можно на основе классификация М. Флинна. Она была предложена в конце 60-х годов прошлого века. Основанием классификации является понятие двух потоков: команд и данных. На основе числа этих потоков выделяется четыре класса архитектур ЭВМ.

1. SISD (Single Instruction Single Data) – единственный поток команд и единственный поток данных. Классическая машина фон Неймана. К этому классу относятся все однопроцессорные системы.

2. SIMD (Single Instruction Multiple Data) – единственный поток команд и множественный поток данных. Типичными представителями являются матричные компьютеры, в которых все процессорные элементы выполняют одну и ту же программу, применяемую к своим локальным данным.

3. MISD (Multiple Instruction Single Date) – множественный поток команд и единственный поток данных. М. Флинн не смог привести ни одного примера реально существующей системы, работающей по этому принципу.

4. MIMD (Multiple Instruction Multiple Date) – множественный поток команд и множественный поток данных. К этому классу относятся практически все современные многопроцессорные системы.

К классу суперкомпьютеров можно отнести все вычислительные системы, кроме типа 1. Отличительной особенностью суперкомпьютеров является то, что это, как правило, большие и соответственно чрезвычайно дорогие многопроцессорные системы. В большинстве случаев в суперкомпьютерах используются те же серийно выпускаемые процессоры, что и в обычных ЭВМ. Поэтому зачастую различие между ними не столько качественное, сколько количественное.

Современные супер компьютеры реализуются на основе процессоров с массовым параллелизмом (Massively Parallel Processors, MPP). Такие системы содержат тысячи процессоров, которые используются в рабочих станциях или персональных компьютерах.

В качестве суперкомпьютера могут рассматриваться вычислительные сети. Это сравнительно дешевая альтернатива суперкомпьютерам. Система требуемой производительности собирается из готовых серийно выпускаемых компьютеров, объединенных с помощью некоторого серийно выпускаемого коммуникационного оборудования. В этом случае имеет место распределенная обработка данных.

Следует отметить, что наибольшее число существующих в настоящее время распределенных систем построено по схеме гетерогенных мультикомпьютерных.

Перечислим характерные особенности таких систем:

- наличие высокоскоростных соединительных сетей;
- подсистема защиты системы от сбоев;
- кластеры рабочих станций.

Компьютеры, являющиеся частями этой системы, могут быть крайне разнообразны, например, по типу процессора, размеру памяти и производительности каналов ввода-вывода. В их состав могут входить высокопроизводительные параллельные системы.

Распределенные ОС

Существует два типа распределенных операционных систем. Мультипроцессорная операционная система управляет ресурсами мультипроцессора.

Мультикомпьютерная операционная система разрабатывается для гомогенных мультикомпьютеров.

Распределенная ОС позволяет использовать одно и то же аппаратное обеспечение различными приложениями изолированно друг от друга.

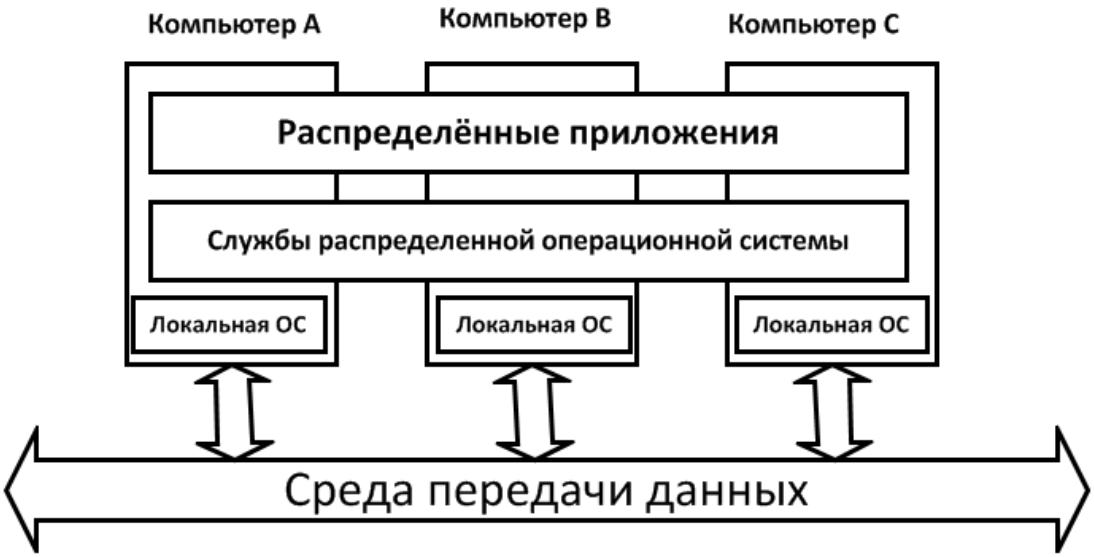

Рис. 15.6. Концепция распределенной операционной системы

Для приложения это выглядит так, словно эти ресурсы находятся в его полном распоряжении, при этом в одной системе может выполняться одновременно несколько приложений, каждое со своим собственным набором ресурсов.

В этом смысле говорят, что операционная система реализует виртуальную машину, предоставляя приложениям средства мультизадачности. На рис. 15.6 показана схема взаимодействия с распределенным приложением посредством служб распределенной вычислительной системы [4].

Многопроцессорные ОС

Обеспечение прозрачности числа процессоров для приложения. Взаимодействие между различными приложениями или их частями требует тех же примитивов, что и в многозадачных однопроцессорных операционных системах.

Все взаимодействие происходит путем работы с данными в специальной совместно используемой области данных.

Требуется защита данных от одновременного доступа к ним. Защита осуществляется посредством примитивов синхронизации. Вид взаимодействия – передача сообщений.

Мультикомпьютерные ОС

Каждый узел имеет свое ядро, которое содержит модули для управления локальными ресурсами – памятью, локальным процессором, локальными дисками и т. д.

Каждый узел имеет отдельный модуль для межпроцессорного взаимодействия, то есть посылки сообщений на другие узлы и приема сообщений от них.

Каждый узел имеет свое ядро, которое содержит модули для управления локальными ресурсами – памятью, локальным процессором, локальными дисками и т.д. Концепция мультикомпьютерной ОС представлена на рис. 15.7 [4].

Каждый узел имеет отдельный модуль для межпроцессорного взаимодействия, то есть посылки сообщений на другие узлы и приема сообщений от них.

Поверх каждого локального ядра лежит уровень программного обеспечения общего назначения [4, 10].

Этот уровень реализует операционную систему в виде виртуальной машины, поддерживающей параллельную работу над различными задачами.

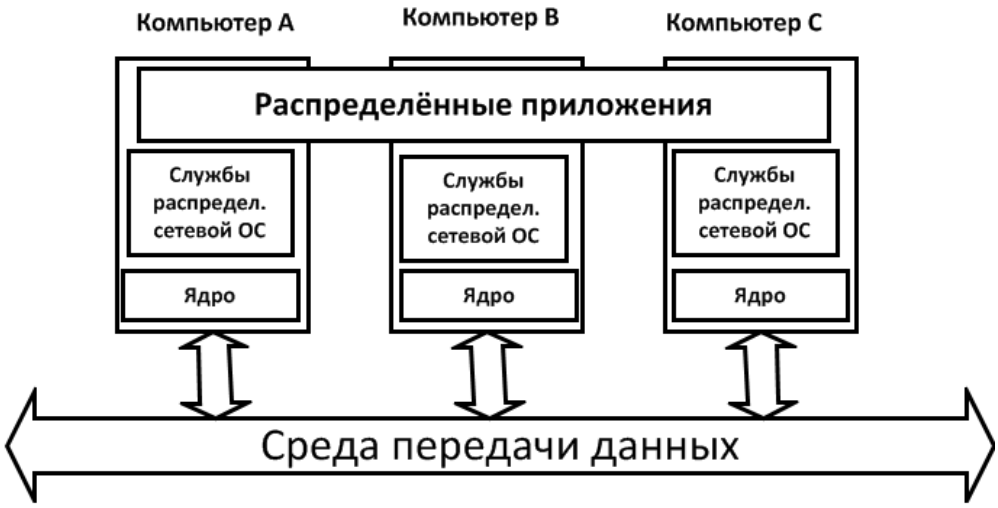

Рис. 15.7. Концепция мультикомпьютерной операционной системы

Сетевые операционные системы

Сетевые операционные системы не нуждаются в том, чтобы аппаратное обеспечение, на котором они функционируют, было гомогенно и управлялось как единая система.

Сетевая операционная система позволяет пользователям использовать службы, расположенные на конкретной машине.

Приведем перечень служб сетевой ОС:

- удаленное соединение пользователя с другой машиной;
- удаленное копирование файлов с одной машины на другую;
- создание глобальной общей файловой системы, доступной со всех рабочих станций;

Файловая система поддерживается одной или несколькими машинами, которые называются файловыми серверами (file servers).

Файловые серверы принимают запросы от программ пользователей, запускаемых на других машинах (не на серверах), которые называются клиентами (clients), на чтение и запись файлов.

Каждый пришедший запрос проверяется и выполняется, а результат пересылается назад.

Программное обеспечение промежуточного уровня. Как было показано выше, в распределенных ОС главную роль по организации взаимодействия играет так называемое промежуточное программное обеспечение (рис. 15.8).

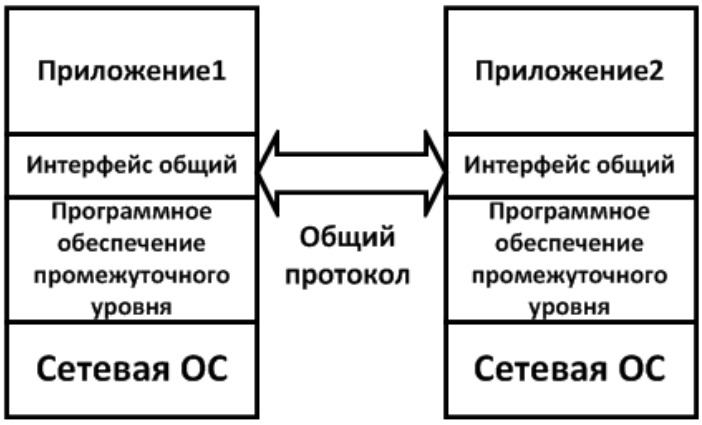

Рис. 15.8. Программное обеспечение промежуточного уровня

Необходимо, чтобы и протоколы промежуточного уровня, и его интерфейсы были одинаковы.

Рассмотрим некоторые базовые концепции его организации [4].

Представление всех объектов вычислительной системы в виде файлов. Такой подход используется в ОС семейства UNIX. Все ресурсы, включая устройства ввода-вывода, такие как клавиатура, мышь, диск, сетевые интерфейсы и т.д., рассматриваются как файлы. Файлы совместно используются несколькими процессами.

Распределенная файловая система. Прозрачность распределения поддерживается только для стандартных файлов (то есть файлов, предназначенных только для хранения данных). Программное обеспечение промежуточного уровня, основанное на модели распределенной файловой системы, достаточно легко масштабируется.

Удаленные вызовы процедур (Remote Procedure Calls, RPC). Сокрытие сетевого обмена за счет того, что процессу разрешается вызывать процедуры, реализация которых находится на удаленной машине. При вызове такой процедуры параметры прозрачно переда-

ются на удаленную машину, где выполняется процедура, после чего результат выполнения возвращается в точку вызова процедуры.

В табл. 15.3 сравниваются различные типы операционных систем [4].

Таблица 15.3

Сравнение ОС

Характеристика	Распределённая система		Сетевая операционная система	Распределённая система промежуточного уровня
	Мультипроцессорные	Мультикомпьютерные		
Степень прозрачности	Очень высокая	Высокая	Низкая	Высокая
Идентичность ОС на всех узлах	Поддерживается	Поддерживается	Не поддерживается	Поддерживается
Число копий ОС	1	N	N	N
Коммуникации на основе	Совместно используемой памяти	Сообщений	Файлов	В зависимости от модели
Управление ресурсами	Глобальное централизованное	Глобальное распределенное	Отдельно на узле	Отдельно на узле
Масштабируемость	Отсутствует	Умеренная	Да	Различная

Распределенные объекты. Объект находится на удаленной машине. Каждый объект реализует интерфейс, который скрывает все внутренние детали объекта от его пользователя. Интерфейс содержит методы, реализуемые объектом. Все, что видит процесс, – это интерфейс.

Распределенные документы. Модель принята в Web, информация организована в виде документов, каждый из которых размещен на машине, расположение которой абсолютно прозрачно. Документы содержат ссылки, связывающие текущий документ с другими. Если сле-

довать по ссылке, то документ, с которым связана эта ссылка, будет извлечен из места его хранения и выведен на экран пользователя. Концепция документа не ограничивается текстовой информацией. Для доступа к документам используются адреса единого формата. В качестве такого формата используется универсальный адрес ресурса URL. Такой адрес содержит имя сервера, на котором находится документ с данным URL адресом.

Контрольные вопросы и задания

1. Какими функциями должна обладать распределенная ОС в отличие от обычной ОС?
2. Какие проблемы по обеспечению безопасности можно выделить в распределенных ОС?
3. Дайте характеристику такому свойству ОС, как прозрачность.
4. Как реализация масштабируемость в распределенных ОС?
5. Дайте описание мультипроцессорной вычислительной системы.
6. Какие способы организации доступа к оперативной памяти используются в мультипроцессорных системах?
7. Какую организацию имеют иерархические вычислительные системы?
8. Чем отличается гомогенная вычислительная система от гетерогенной?
9. Что понимается под таким понятием как "Супер ЭВМ" при организации распределенной обработки?
10. Перечислите свойства вычислительных сетей, необходимые для организации распределенной обработки.

ЛИТЕРАТУРА

Основная

1. Карпов, В.Е. Основы операционных систем: учеб. пособие / В.Е. Карпов, К.А. Коньков. – М.: ИНТУИТ.РУ «Интернет-университет Информационных технологий», 2005. – 536 с.
2. Орлофф, Дж. Ubuntu. Бесплатная альтернатива Windows / Дж. Орлофф. – М.: Эксмо,2009. – 352 с.
3. Партыка, Т.Л. Операционные системы, среды и оболочки: учеб. пособие / Т.Л. Партыка, И.И. Попов. – 5-е изд., испр. и доп. – М.: ФОРУМ, 2013. – 560 с.
4. Таненбаум, Э. Современные операционные системы / Э. Таненбаум. – 3-е изд. – СПб.: Питер, 2010. – 1120 с.

Дополнительная

5. Баррет, Даниэл Дж. Linux – основные команды: карманный справочник / Дж. Баррет Даниэл. – М.: КУДИЦ-ОБРАЗ, 2007. – 288 с.
6. Волков, В.Б. Линукс Юниор: книга для учителя / В.Б. Волков. – М.: ALT Linux; Издательский дом ДМК-пресс, 2009. – 363 с.
7. Гласс, Г. UNIX для программистов и пользователей / Г. Гласс, К. Эйблс. – 3-е изд., перераб. и доп. – СПб.: БХВ-Петербург, 2004. – 848 с.
8. Джонсон, М. Разработка приложений в среде Linux / М. Джонсон, В. Троан Эрик. – 2-е изд. – М.: ООО «И.Д. Вильямс», 2007. – 544 с.
9. Олифер, В.Г. Сетевые операционные системы / В.Г. Олифер, Н.А. Олифер. – СПб.: Питер, 2002. – 544 с.
10. Таненбаум, Э. Архитектура компьютера / Э. Таненбаум. – 5-е изд. – СПб.: Питер, 2007. – 844 с.
11. Тейнсли, Д. Linux и UNIX: программирование в shell. Руководство разработчика / Д. Тейнсли. – Киев: Издательская группа BHV, 2001. – 464 с.

СОДЕРЖАНИЕ

Учебное издание

МЕЗЕНЦЕВ Константин Николаевич

ОПЕРАЦИОННЫЕ СИСТЕМЫ

КУРС ЛЕКЦИЙ

Под редакцией д-ра техн. наук, проф. А.Б. Николаева

Редактор И.А. Короткова

Подписано в печать 01.11.2016 г. Формат 60×84/16.
Усл. печ. л. 8,75. Тираж 100 экз. Заказ . Цена 285 руб.
МАДИ, 125319, Москва, Ленинградский пр-т, 64.