**Тезисы семинаров и практических работ по модулю МДК 01.01 Системное программирование для специальности 230115 Программирование компьютерных систем**

**СЕМИНАР 1. ОСНОВНЫЕ ПОНЯТИЯ И ОПРЕДЕЛЕНИЯ СИСТЕМНОГО ПРОГРАММИРОВАНИЯ**

**Основные понятия:** программа, программное обеспечение, свойства программного обеспечения, программный модуль, исходный модуль, макропроцессор, трансляция, машинный язык, трансляторы, автокод, язык Ассемблера, язык высокого уровня, объектный модуль, редактором связей, загрузочный модуль,связывающий загрузчик, интерпретация, кросс-системама.

**План:**

1. Программа и программное обеспечение.

2. Этапы подготовки программы: исходный, объектный и загрузочный модули.

3. Назначение и компоненты классического СПО. Трансляторы: интерпретаторы и компиляторы.

**Содержание по плану:**

**1. Программа и программное обеспечение.**

**Программа** — это данные, предназначенные для управления конкретными компонентами системы обработки информации (СОИ) в целях реализации определенного алгоритма.

*Обратить внимание: программа — это данные.*

Один из основных принципов машины фон Неймана — то, что и программы, и данные хранятся в одной и той же памяти. Сохраняемая в памяти программа представляет собой некоторые коды, которые могут рассматриваться как данные. Возможно, с точки зрения программиста программа — активный компонент, она выполняет некоторые действия. Но с точки зрения процессора команды программы — это данные, которые процессор читает и интерпретирует. С другой стороны программа — это данные с точки зрения обслуживающих программ, например, с точки зрения компилятора, который на входе получает одни данные — программу на языке высокого уровня (ЯВУ), а на выходе выдает другие данные — программу в машинных кодах.

**Программное обеспечение (ПО)** — совокупность программ СОИ и программных документов, необходимых для их эксплуатации

Существенно, что ПО — это программы, предназначенные для многократного использования и применения разными пользователями. В связи с этим следует обратить внимание на ряд необходимых свойств ПО.

*Необходимость документирования*

По определению программы становятся ПО только при наличии документации. Конечный пользователь не может работать, не имея документации. Документация делает возможным тиражирование ПО и продажу его без его разработчика.

*Эффективность*

ПО, рассчитанное на многократное использование (например, ОС, текстовый редактор) пишется и отлаживается один раз, а выполняется многократно.

*Надежность*

Сюда входит:

* Тестирование программы при всех допустимых спецификациях входных данных
* Защита от неправильных действий пользователя
* Защита от взлома — пользователи должны иметь возможность взаимодействия с ПО только через легальные интерфейсы

Появление ошибок любого уровня не должно приводить к краху системы. Ошибки должны вылавливаться, диагностироваться и (если их невозможно исправить) превращаться в корректные отказы.

Системные структуры данных должны сохраняться безусловно.

Сохранение целостности пользовательских данных желательно.

*Возможность сопровождения*

Возможные цели сопровождения — адаптация ПО к конкретным условиям применения, устранение ошибок, модификация.

**Вывод:** Во всех случаях требуется тщательное структурирование ПО и носителем информации о структуре ПО должна быть программная документация.

**2. Этапы подготовки программы: исходный, объектный и загрузочный модули.**

При разработке программ используется **принцип модульности**, разбиения сложной программы на составные части, каждая из которых может подготавливаться отдельно. Модульность облегчает его разработку, отладку и сопровождение.

**Программный модуль** — программа или функционально завершенный фрагмент программы, предназначенный для хранения, трансляции, объединения с другими программными модулями и загрузки в оперативную память.

При выборе модульной структуры должны учитываться следующие основные соображения:

* Функциональность — модуль должен выполнять законченную функцию
* Несвязность — модуль должен иметь минимум связей с другими модулями, связь через глобальные переменные и области памяти нежелательна
* Специфицируемость — входные и выходные параметры модуля должны четко формулироваться

Программа пишется в виде исходного модуля.

**Исходный модуль** — программный модуль на исходном языке, обрабатываемый транслятором и представляемый для него как целое, достаточное для проведения трансляции.

Первым (не для всех языков программирования обязательным) этапом подготовки программы является **обработка** ее **Макропроцессором** (или Препроцессором, разновидность компилятора). Макропроцессор обрабатывает текст программы и на выходе его получается новая редакция текста. В большинстве систем программирования Макропроцессор совмещен с транслятором, и для программиста его работа и промежуточный ИМ «не видны».

Следует иметь в виду, что Макропроцессор выполняет обработку текста, это означает, с одной стороны, что он «не понимает» операторов языка программирования и «не знает» переменных программы, с другой, что все операторы и переменные Макроязыка (тех выражений в программе, которые адресованы Макропроцессору) в промежуточном ИМ уже отсутствуют и для дальнейших этапов обработки «не видны».

Так, если Макропроцессор заменил в программе некоторый текст A на текст B, то транслятор уже видит только текст B, и не знает, был этот текст написан программистом «своей рукой» или подставлен Макропроцессором.

Следующим этапом является трансляция.

**Трансляция** — преобразование программы, представленной на одном языке программирования, в программу на другом языке программирования, в определенном смысле равносильную первой.

Как правило, выходным языком транслятора является машинный язык целевой вычислительной системы (целевая ВС — та ВС, на которой программа будет выполняться.)

**Машинный язык** — язык программирования, предназначенный для представления программы в форме, позволяющей выполнять ее непосредственно техническими средствами обработки информации.

**Трансляторы** — общее название для программ, осуществляющих трансляцию. Они подразделяются на Ассемблеры и Компиляторы — в зависимости от исходного языка программы, которую они обрабатывают. Ассемблеры работают с Автокодами или языками Ассемблера, Компиляторы — с языками высокого уровня.

**Автокод** — символьный язык программирования, предложения которого по своей структуре в основном подобны командам и обрабатываемым данным конкретного машинного языка.

**Язык Ассемблера** — язык программирования, который представляет собой символьную форму машинного языка с рядом возможностей, характерных для языка высокого уровня (обычно включает в себя макросредства).

**Язык высокого уровня** — язык программирования, понятия и структура которого удобны для восприятия человеком.

**Объектный модуль** — программный модуль, получаемый в результате трансляции исходного модуля.

Поскольку результатом трансляции является модуль на языке, близком к машинному, в нем уже не остается признаков того, на каком исходном языке был написан программный модуль. Это создает принципиальную возможность создавать программы из модулей, написанных на разных языках. Специфика исходного языка, однако, может сказываться на физическом представлении базовых типов данных, способах обращения к процедурам/функциям и т.п. Для совместимости разноязыковых модулей должны выдерживаться общие соглашения. Большая часть объектного модуля — команды и данные машинного языка именно в той форме, в какой они будут существовать во время выполнения программы. Однако, программа в общем случае состоит из многих модулей. Поскольку транслятор обрабатывает только один конкретный модуль, он не может должным образом обработать те части этого модуля, в которых запрограммированы обращения к данным или процедурам, определенным в другом модуле. Такие обращения называются внешними ссылками. Те места в объектном модуле, где содержатся внешние ссылки, транслируются в некоторую промежуточную форму, подлежащую дальнейшей обработке. Говорят, что объектный модуль представляет собой программу на машинном языке с неразрешенными внешними ссылками. Разрешение внешних ссылок выполняется на следующем этапе подготовки, который обеспечивается **Редактором Связей** (**Компоновщиком**). Редактор Связей соединяет вместе все объектные модули, входящие в программу. Поскольку Редактор Связей «видит» уже все компоненты программы, он имеет возможность обработать те места в объектных модулях, которые содержат внешние ссылки. Результатом работы Редактора Связей является загрузочный модуль.

**Загрузочный модуль** — программный модуль, представленный в форме, пригодной для загрузки в оперативную память для выполнения.

Загрузочный модуль сохраняется в виде файла на внешней памяти. Для выполнения программа должна быть перенесена (загружена) в оперативную память. Иногда при этом требуется некоторая дополнительная обработка (например, настройка адресов в программе на ту область оперативной памяти, в которую программа загрузилась). Эта функция выполняется Загрузчиком, который обычно входит в состав операционной системы. Возможен также вариант, в котором редактирование связей выполняется при каждом запуске программы на выполнение и совмещается с загрузкой. Это делает **Связывающий Загрузчик**. Вариант связывания при запуске более расходный, т.к. затраты на связывание тиражируются при каждом запуске. Но он обеспечивает:

* большую гибкость в сопровождении, так как позволяет менять отдельные объектные модули программы, не меняя остальных модулей;
* экономию внешней памяти, т.к. объектные модули, используемые во многих программах не копируются в каждый загрузочный модуль, а хранятся в одном экземпляре.

Вариант интерпретации подразумевает прямое исполнение исходного модуля.

**Интерпретация** — реализация смысла некоторого синтаксически законченного текста, представленного на конкретном языке.

Интерпретатор читает из исходного модуля очередное предложение программы, переводит его в машинный язык и выполняет.

Не обязательно подготовка программы должна вестись на той же вычислительной системе и в той же операционной среде, в которых программа будет выполняться. Системы, обеспечивающие подготовку программ в среде, отличной от целевой, называются **кросс-системами**. В кросс-системе может выполняться вся подготовка или ее отдельные этапы:

* Макрообработка и трансляция
* Редактирование связей
* Отладка

Типовое применение кросс-систем — для тех случаев, когда целевая вычислительная среда просто не имеет ресурсов, необходимых для подготовки программ, например, встроенные системы.

**3. Назначение и компоненты классического СПО. Трансляторы: интерпретаторы и компиляторы.**

**Системная программа** — программа, предназначенная для поддержания работоспособности СОИ или повышения эффективности ее использования.

**Прикладная программа** — программа, предназначенная для решения задачи или класса задач в определенной области применения СОИ.

**Системное программирование** — это процесс разработки системных программ (в том числе, управляющих и обслуживающих).

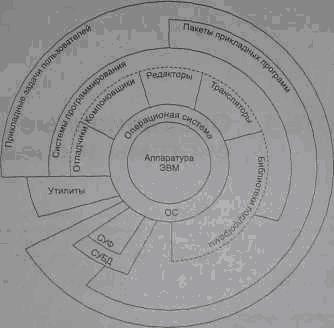
С другой стороны, система — единое целое, состоящее из множества компонентов и множества связей между ними. Тогда системное программирование — это разработка программ сложной структуры.

Подразделение ПО на системное и прикладное является до некоторой степени устаревшим. Сегодняшнее деление предусматривает по меньшей мере три градации ПО:

* Системное
* Промежуточное
* Прикладное

Промежуточное ПО (middleware) – совокупность программ, осуществляющих управление вторичными (конструируемыми самим ПО) ресурсами, ориентированными на решение определенного (широкого) класса задач. К такому ПО относятся менеджеры транзакций, серверы БД, серверы коммуникаций и другие программные серверы.

Значительная часть системного и практически все прикладное ПО пишется на языках высокого уровня, что обеспечивает сокращение расходов на их разработку/модификацию и переносимость.

СУФ – система управления файлами

Системное ПО подразделяется на системные управляющие программы и системные обслуживающие программы.

**Управляющая программа** — системная программа, реализующая набор функций управления, который включает в себя управление ресурсами и взаимодействие с внешней средой СОИ, восстановление работы системы после проявления неисправностей в технических средствах.

**Программа обслуживания (утилита)** — программа, предназначенная для оказания услуг общего характера пользователям и обслуживающему персоналу СОИ (утилиты для диагностики аппаратного или программного обеспечения, архиваторы, утилиты восстановления после сбоев).

Управляющая программа совместно с набором необходимых для эксплуатации системы утилит составляют операционную систему (ОС).

**Система программирования** — система, образуемая языком программирования, компилятором или интерпретатором программ, представленных на этом языке, соответствующей документацией, а также вспомогательными средствами для подготовки программ к форме, пригодной для выполнения.

**Контрольные вопросы:**

1. Раскройте понятие систменая программа.

2. Укажите разновидности системных программ.

3. Перечислите этапы подготовки программы.

4. Укажите отличия и сходства между компиляторами и трансляторами.

**Литература:**

1. А.П. Побегайло, Системное программирование в Windows, БХВ-Петербург, 2006, 1056 с.
2. Ахо, Лам, Сети, Ульман - Компиляторы. Принципы, технологии, инструменты, Изд.: Вильямс, 2008, 1185 с.
3. Л.Бек, Введение в системное программирование, М., Мир, 2008г.
4. Р. И. Компаниец, Системное программирование : Основы построения трансляторов, Изд.: СПб., 2000.
5. Ф.Льюис, Д.Розенкранц, Р.Стирнз. Теоретические основы проектирования компиляторов. - М.: Мир, 20099, 655 с.
6. Э. Таненбау, Современные операционные системы, Изд.: Питер, 2011, 1120 с. А. В. Гордеев, А. Ю. Молчанов, Системное программное обеспечение, учебник для ВУЗов, СПб.: Питер, 2010г.

**СЕМИНАР №2. СТРУКТУРА ДРАЙВЕРА УСТРОЙСТВА**

**Основные понятия:** драйвер, структура драйвепа, функции драйвера, низкоуровневые и аппаратные драйверы.

**План скминара:**

1. Понятие драйвера устройства.
2. Функции драйвера.
3. Структура драйвера.
4. Типы драйверов.

**Содержание по плану:**

1. **Понятие драйвера устройства.**

Значительную часть ядра любой операционной системы занимают драйвера устройств.

**Драйвер устройства** – специальная программа, которая управляет обменом информации с периферийным устройством (принтер, клавиатура).

Для разных ОС нужны разные драйверы.

Желательно, чтобы драйверы были частью ядра, что бы получить доступ к регистрам контроллера.

Это одна из основных причин приводящих к краху ОС. Потому что драйверы, как правило, пишутся производителями устройств, и вставляются в ОС.



Обмен данными между контроллерами и драйверами идет по шине.

1. **Функции драйвера.**

**Функции драйверов**:

* обработка запросов чтения или записи
* инициализация устройства
* управление энергопотреблением устройства
* прогрев устройства (сканера)
* включение устройства

1. **Структура драйвера.**

**Структура драйвера устройства**

Драйвер устройства реализует множество стандартных процедур:

* Обязательная процедура инициализации
* Обязательный набор процедур диспетчеризации
* Необязательная процедура запуска
* Необязательная процедура обслуживания прерываний
* Необязательный отложенный вызов
* Необязательная процедура завершения
* Обязательная процедура выгрузка
* Необязательная процедура отмены
* Обязательная процедура уведомление об отключении системы
* Необязательная процедура протоколирования ошибок

Все драйверы устройств Windows имеют одинаковую структуру. Каждый драйвер имеет объект драйвера, который создается диспетчером ввода-вывода при загрузке драйвера.

Объект драйвера выделяется из невыгружаемой памяти. Он содержит важную информацию, например таблицу вызовов драйвера, которая, в свою очередь, содержит адреса для различных процедур драйвера.

1. **Типы драйверов.**

Со временем появились 2 типа драйверов:

**Драйверы устройств** (низкоуровневые или аппаратные драйверы) – традиционные драйверы.

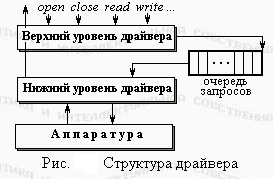
**Высокоуровневые драйверы**, которые распо­лагаются в общей модели подсистемы ввода-вывода над традиционными драйве­рами.

При таком подходе повышается гибкость и расширяемость функций по управле­нию устройством — вместо жесткого набора функций, сосредоточенных в един­ственном драйвере, администратор ОС может выбрать требуемый набор функций, установив нужный высокоуровневый драйвер.

На практике чаще всего используют от двух до пяти уровней драйверов — слишком большое количество уровней может снизить скорость операций ввода-вывода. Несколько драйверов, управляющих одним устройством, но на разных уровнях, можно рассматривать как набор отдельных драйверов или как один многоуровневый драйвер.

В подсистеме управления графическими устройствами, такими как графические мониторы и принтеры на ниж­нем уровне работают аппаратные драйверы, которые позволяют управлять кон­кретным графическим адаптером или принтером определенного типа, Са­мый верхний уровень подсистемы составляет менеджер окон, который создает для каждого приложения виртуальный образ экрана в виде набора окон, в кото­рые приложение может выводить свои графические данные. Менеджер управля­ет окнами, отображая их в определенную область физического экрана или делая их невидимыми, а также предоставляет к ним совместный доступ с контролем прав доступа

Аппаратные драйверы взаимодействуют с системой прерываний. Драйверы более вы­соких уровней вызываются уже не по прерываниям, а по инициативе аппарат­ных драйверов или драйверов вышележащего уровня.



Первый (верхний) уровень принимает системные вызовы от процессов и формирует на основании каждого вызова запрос. Этот же уровень выстраивает запросы в очередь и поддерживает упорядоченность этой очереди в соответствии с принятой дисциплиной обслуживания. Второй (нижний) уровень драйвера выбирает из очереди первый запрос и обслуживает его: формирует управляющие воздействия и передает их на устройство, обрабатывает прерывания от устройства и сообщает ядру ОС о наступлении событий, связанных с вводом-выводом.

**Некоторые типы драйверов**

**Драйверы GDI** (Graphic Device Interface) представляют собой высокоуровневые драйверы графических устройств (мониторов, принтеров, плоттеров). Эти драйверы выполняют трансляцию графических вызовов Windows (таких, как «провести линию», «залить область», «выдать текст», «выбрать текущий шрифт, текущее перо, текущую заливку») в команды, выполняющие соответствующие действия на конкретном устройстве. Выдача этих команд на устройство выполняется уже другим, низкоуровневым драйвером. Благодаря наличию драйверов GDI одна и та же программа может выдавать графическое изображение на разные устройства. Яркий пример этого — имеющийся в различных редакторах режим предварительного просмотра, который отображает страницы на экране точно в том виде, как они будут напечатаны.

**Драйверы клавиатуры и мыши**, помимо стандартных для драйвера операций, выполняют дополнительную нагрузку. Они генерируют сообщения о событиях на соответствующем устройстве (нажатие и отпускание клавиши, перемещение мыши, нажатие и отпускание кнопок мыши) и помещают их в системную очередь сообщений. Затем система переправляет каждое сообщение процессу, которому оно было предназначено, для дальнейшей обработки.

**Драйверы виртуализации устройств** служат для того, чтобы разделять устройства между процессами, создавая иллюзию, что процесс монопольно владеет устройством. На самом деле драйвер организует очередь заявок от процессов, переключает устройство в нужный для очередного процесса режим и т.п. Примером может служить драйвер виртуализации монитора. Консольное приложение (например, программа MS-DOS) работает со всем экраном в текстовом режиме. Но если такое приложение запущено в окне Windows, то драйвер имитирует текстовый режим в графике. Для этого драйвер должен перехватывать попытки программы обратиться напрямую к адресам видеопамяти и преобразовывать координаты знакомест текстового режима в координаты соответствующих позиций в окне.

**Контрольные вопросы:**

1. Дайте определение понятию драйвер устройства.

2. Укажите основные функции драйверов.

3. Перечислите свойства драйверов.

4. Перечислите типы драйверов.

5. Опишите структуру драйвера устройства.

**Литература:**

1. А.П. Побегайло, Системное программирование в Windows, БХВ-Петербург, 2006, 1056 с.
2. Ахо, Лам, Сети, Ульман - Компиляторы. Принципы, технологии, инструменты, Изд.: Вильямс, 2008, 1185 с.
3. Л.Бек, Введение в системное программирование, М., Мир, 2008г.
4. Р. И. Компаниец, Системное программирование : Основы построения трансляторов, Изд.: СПб., 2000.
5. Ф.Льюис, Д.Розенкранц, Р.Стирнз. Теоретические основы проектирования компиляторов. - М.: Мир, 20099, 655 с.
6. Э. Таненбау, Современные операционные системы, Изд.: Питер, 2011, 1120 с. А. В. Гордеев, А. Ю. Молчанов, Системное программное обеспечение, учебник для ВУЗов, СПб.: Питер, 2010г.

**СЕМИНАР №3: «ОСНОВЫ КОМПИЛЯТОРОВ».**

**Основные понятия:**космиляция, трансляция, интепретация, прямой копилятор, раскрутка, кросс-транслятор, виртуальная машина, компиляция "на лету".

**План семинара:**

1. Основные понятия. Компиляторы и интерпретаторы.
2. Входной язык, целевой язык, язык реализации. T-диаграммы.
3. Прямой компилятор. Раскрутка. Кросс-трансляторы.
4. Виртуальные машины.Компиляция "на лету".

**Содержание по плану:**

1. **Основные понятия. Компиляторы и интерпретаторы.**

Задача транслятора (translator) сделать программу, написанную на некотором языкепрограммирования, понятной компьютеру. Этого можно добиться одним из двухспособов: компиляцией (compilation) или интерпретацией (interpretation). Программу,являющуюся входными данными транслятора, будем называть исходной программой(source program). Обычно язык, на котором написана исходная программа, - это языквысокого уровня (high-levellanguage).

Компилятор (compiler) переводит исходную программу в эквивалентную программу наязыке, понятном компьютеру, то есть на машинном языке. Мы будем называтьпрограмму, получающуюся в результате работы компилятора, целевой программой(target program). Процесс компиляции и последующего выполнения программы можноизобразить следующим образом:



Сам компилятор также является программой, причем программой на машинном языке.

Хотя, как мы увидим в дальнейшем, первоначально компилятор может быть написан накаком-нибудь языке более высокого уровня, чем машинный язык, а затемоткомпилирован в машинный язык. Язык, на котором написан компилятор, будемназывать языком реализации.

Если мы обозначим язык, на котором написана исходная программа, то есть *исходныйязык* (*source language*), L1, язык целевой программы (*целевой язык – target language*) L2,а язык реализации (*implementation language*) L3, то мы можем представить компилятор,как отображение множества L1 в множество L2 , т.е. 1 2 : 3 *K L L L* → .

Процесс компиляции состоит из двух частей: анализа (*analysis*) и синтеза (*synthesis*).

Анализирующая часть компилятора разбивает исходную программу на составляющие ееэлементы (конструкции языка – *language constructions*) и создает промежуточноепредставление исходной программы. Синтезирующая часть из промежуточногопредставления создает новую, целевую, программу.

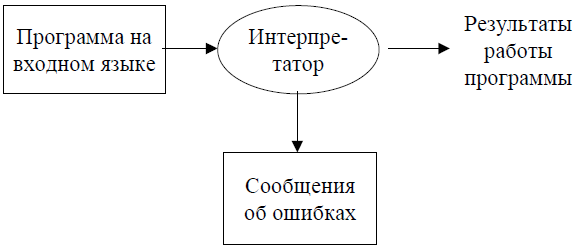
Отметим, что однажды полученная в результате компиляции целевая программа может вдальнейшем выполняться много раз с различными входными данными.

Далеко не всегда исходные программы корректны с точки зрения исходного языка.

Более того, некорректные программы подаются на вход компилятору значительно чаще,чем корректные – таков уж современный процесс разработки программ. Поэтому крайневажной частью процесса компиляции является точная диагностика ошибок, допущенныхво входной программе. Впрочем, это замечание верно не тоько для компиляторов, но идля интерпретаторов.

**Интерпретатор**

Интерпретатор анализирует и выполняет программу навходном языке.

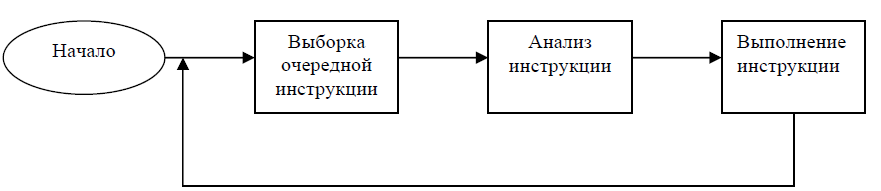


В отличие от компилятора интерпретатор не создает никакой новой программы.Входными данными интерпретатора является не только исходная программа, но ивходные данные самой исходной программы.



Интерпретатор, так же, как и компилятор, анализирует программу на входном языке,создает промежуточное представление, а затем выполняет операции, содержащиеся втексте этой программы. Например, интерпретатор может построить дерево разбора, азатем выполнить операции, которыми помечены узлы этого дерева.

В том случае, если исходный язык достаточно прост (например, если это языкассемблера или Basic), то никакое промежуточное представление не нужно, и тогдаинтерпретатор – это простой цикл. Он выбирает очередную инструкцию языка извходного потока, анализирует и выполняет ее. Затем выбирается следующаяинструкция. Этот процесс продолжается до тех пор, пока не будут выполнены всеинструкции, либо пока не встретится инструкция, означающая окончание процессаинтерпретации.

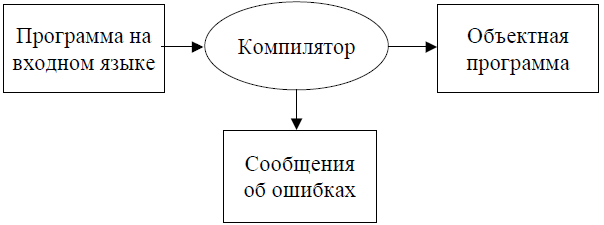


Понятно, что при повторном запуске программы она должна интерпретироваться ссамого начала.

Интерпретация приводит к более гибкой и лучшей диагностике ошибок, чемкомпиляция. Поскольку исходная программа исполняется непосредственно,интерпретатор может включать хороший *отладчик* (*debugger*). Кроме того,интерпретатор может легко справиться с языками, позволяюшими создавать программы,некоторые характеристики которых (например, размеры и типы переменных) могутзависеть от входных данных. Некоторые черты языков программирования таковы, чтоони не могут быть реализованы иначе, чем с использованием интерпретации.

Современная ситуация такова, что в трансляторах часто используются как элементыкомпиляции, так и интерпретации.

**Компилятор** анализирует программу на входномязыке и создает эквивалентнуюобъектную программу.



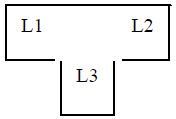
При обсуждении процесса компиляции необходимо рассмотретьСуществует огромноеколичество различных языков программирования, начиная с таких традиционныхязыков программирования как Fortran и Pascal и кончая современными объектно-ориентированными языками такими, как C# и Java. Практически каждый языкпрограммирования имеет какие-то особенности с точки зрения создателя транслятора.

1. **Входной язык, целевой язык, язык реализации. T-диаграммы.**

Если мы обозначим язык, на котором написана исходная программа, то есть *исходный язык* (*source language*), L1, язык целевой программы (*целевой язык – target language*) L2, а язык реализации (*implementation language*) L3, то мы можем представить компилятор, как отображение множества L1 в множество L2 , т.е. 1 2 : 3 *K L L L* → .

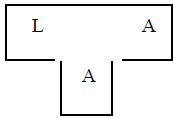
Т-диаграммы – это удобный способ графического представления компиляторов.

Мы обозначили компилятор, переводящий цепочки языка L1 в цепочки языка L2 инаписанный на языке L3, следующим образом: KL3 : L1→L3. Для представлениякомпилятора мы можем использовать также так называемые Т-диаграммы:



Компилятор, переводящий язык L в язык ассемблера A и написанный на языке A, можно

представить следующими способами: KA: L→ A или



Т-диаграммы достаточно удобны при обсуждении различных способов получениякомпиляторов.

Методики создания компилятора:

• Прямой

• Раскрутка

• Кросс-транслятор

• Виртуальная машина

• Компиляция "на лету"

1. **Прямой компилятор. Раскрутка. Кросс-трансляторы.**

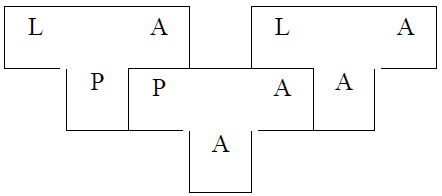
Компилятор – это весьма большая и сложная программа; на написание и отладкукомпилятора на языке ассемблера можно истратить слишком много времени. Для того,чтобы как-то справиться с этой проблемой, был придуман метод раскрутки, сутькоторого заключается в следующем.

Пусть есть компилятор *KA: P→A*, где *P* – некоторый язык более высокого уровня, чемязык ассемблера. Тогда напишем *KP: L→A*, а затем применим компилятор *KA* ккомпилятору *KP*, т.е. получим *KA=KA(KP): L→A*. Такая схема проиллюстрирована спомощью Т-диаграмм на слайде и называется *раскруткой* (*bootstrapping1*).

Описанная схема может быть использована при написании компилятора некоторогоязыка на нем самом. Пусть у нас есть компилятор некоторого подмножества *S* языка *L* вязык *A*, написанный на языке *A*, *KA: S→A*. Тогда мы можем написать *KL: L→A* и получимновый компилятор *KA = KA(KL)*. Мы используем это подмножество *S* для того, чтобынаписать компилятор языка *L* в язык *A*, *KS: L→A*. Если теперь мы применим компилятор*KA* к программе *KS*, то получим *KA=KA(KS): L→A*.

Впервые такая схема была применена в 1960 году при реализации языка Neliac. В 1971году Вирт написал с использованием раскрутки транслятор языка Pascal, причем самыйпервый компилятор был оттранслирован вручную. Количество шагов раскрутки былобольше 1, т.е. была построена последовательность языков *S*1 ⊂*S*2 ⊂...*Sn*= *L* ипостроена последовательность компиляторов: *KA: S1→A, KA1=KA(KS1: S2→A), ...*

Раскрутку можно использовать и в следующей ситуации. Пусть у нас есть недостаточноэффективный компилятор *KA: L→A*. Можно написать более эффективный компилятор*KL: L→A*, а затем применить раскрутку.



**Кросс-транслятор**

Пусть у нас есть два компьютера: компьютер M с языком ассемблера A и компьютер M1с языком ассемблера A1. Кроме того, предположим, что имеется компилятор KA1: P→A1,а сам компьютер M по каким-то причинам не доступен либо пока еще не существуеткомпилятор KA: P→A. Нас интересует компилятор KA: L→A. В такой ситуации мыможем использовать M1 в качестве инструментальной машины и написать компиляторKP: L→A, который принято называть кросс-транслятором (cross-compiler). Как толькомашина M станет доступной, мы сможем перенести KP на M и "раскрутить" его спомощью KA. Понятно, что это решение достаточно трудоемко, поскольку могутвозникнуть проблемы при переносе, например, из-за различий операционных систем.

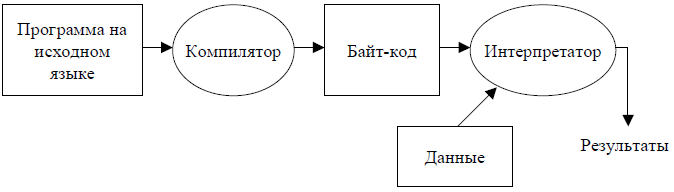
Под переносимой (portable) программой понимается программа, которая может безперетрансляции выполняться на нескольких (по меньшей мере, на двух) платформах. Всвязи с этим возникает вопрос о переносимости объектных программ, создаваемыхкомпилятором. Компиляторы, созданные по методикам, рассмотренным выше,порождают непереносимые (non-portable) объектные программы. Посколькукомпилятор, в конечном итоге, является программой, то мы можем говорить и опереносимых компиляторах. Одним из способов получения переносимых объектныхпрограмм является генерация объектной программы на языке более высокого уровня,чем язык ассемблера. Такие компиляторы иногда называют конвертерами (converter).

1. Виртуальные машины.Компиляция "на лету".

Другой способ получения *переносимой* (*portable*) объектной программы связан сиспользованием *виртуальных машин* (*virtualmachine*). При таком подходе исходныйязык транслируется в коды некоторой специально разработанной машины, которуюникто не собирается реализовывать "в железе". Затем для каждой целевой платформыпишется интерпретатор виртуальной машины.

Понятно, что архитектура виртуальной машины должна быть разработана такимобразом, чтобы конструкции исходного языка удобно отображались в систему команд исама система команд не была слишком сложной. При выполнении этих условий можнодостаточно быстро написать интерпретатор виртуальной машины.

Одна из первых широко известных виртуальных машин была разработана в 70-х годахН. Виртом при написании компилятора Pascal-P. Этот компилятор генерировалспециальный код, названный P-кодом и представляющий собой последовательностьинструкций гипотетической стековой машины. Сегодня идея виртуальных машинприобрела широкую известность благодаря языку Java, компиляторы которого обычногенерируют так называемый *байт-код*, т.е. объектный код, который представляет собойпоследовательность команд виртуальной Java-машины.



**Компиляция "на лету"**

Основная неприятность, связанная с использованием виртуальных машин, заключаетсяв том, что обычно время выполнения интерпретируемой программы значительнобольше, чем время работы программы, оттранслированной в "родной" машинный код.

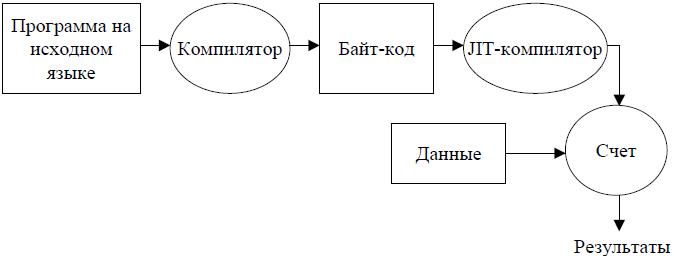
Для того, чтобы увеличить скорость работы приложений, была разработана технологиякомпиляции "на лету" (Just-In-Time compiling; иногда такой подход называют такжединамической компиляцией). Идея заключается в том, что JIT-компилятор генерируетмашинный код прямо в оперативной памяти, не сохраняя его. Это приводит кзначительному увеличению скорости выполнения приложения. Именно так и устроена платформа .NET.

Часто JIT-компилятор используется вместе с интерпретатором виртуальной машины.Организовывается это следующим образом. Вначале сгенерированный байт-код

поступает на вход интерпретатору виртуальной машины, которая его интерпретирует.

Одновременно с интерпретатором работает программа, которая вычисляет времяинтерпретации какого-то куска байт-кода, например, процедуры. Если оказывается, чтовремя интерпретации некоторого куска кода достаточно большое, то вызывается JIT-компилятор, который транслирует его в "родные" машинные коды. Когда привыполнении приложения произойдет повторное обращение к этому куску кода, то онуже не будет интерпретироваться, а будет выполняться сгенерированный фрагментмашинного кода.

Использование связки "компилятор+интерпретатор+JIT-компилятор" позволяет заметноповысить скорость выполнения исходной программы, причем переносимость кода,создаваемого компилятором, естественно, сохраняется.



**Контрольные вопросы:**

1. Раскройте понятие компиляции.

2. Опишите процесс компиляции программы.

3. Раскройте понятие интерпретации.

4. Опишите процесс интерпретации программы.

5. Укажите различия в понятиях входной, целевой и язык реализации.

6. Опишите процесс построения T-диаграммы.

7. перечислите и охарактеризуйте методы создания компиляторов.

**Литература:**

1. А.П. Побегайло, Системное программирование в Windows, БХВ-Петербург, 2006, 1056 с.
2. Ахо, Лам, Сети, Ульман - Компиляторы. Принципы, технологии, инструменты, Изд.: Вильямс, 2008, 1185 с.
3. Л.Бек, Введение в системное программирование, М., Мир, 2008г.
4. Р. И. Компаниец, Системное программирование : Основы построения трансляторов, Изд.: СПб., 2000.
5. Ф.Льюис, Д.Розенкранц, Р.Стирнз. Теоретические основы проектирования компиляторов. - М.: Мир, 20099, 655 с.
6. Э. Таненбау, Современные операционные системы, Изд.: Питер, 2011, 1120 с. А. В. Гордеев, А. Ю. Молчанов, Системное программное обеспечение, учебник для ВУЗов, СПб.: Питер, 2010г.

**СЕМИНАР №4: «ФАЗЫ КОМПИЛЯЦИИ И МЕТОДЫ СОЗДАНИЯ КОМПИЛЯТОРОВ»**

**Основные понятия:** компиляция, лексический анализ, синтаксический анализ, видозависимый анализ, оптимизация, генерация кода, конечный автомат, алфавит, язык, цепочка языка.

**План семинара:**

1. Компоненты компилирующей программы.
2. Лексический и синтаксический анализаторы, генерация и оптимизация объектного кода.
3. Конечный автомат как средство построения лексического анализатора.
4. Формальные грамматики как средство описания синтаксиса языка.
5. Описание бесконечного языка конечными средствами.

**Содержание по плану:**

1. **Компоненты компилирующей программы.**

Процесс создания компилятора можно свести к решению нескольких задач, которыепринято называть фазами компиляции (compilation phases). Обычно компилятор состоит

из следующих фаз:

• лексический анализ

• синтаксический анализ

• видозависимый анализ

• оптимизация

• генерация кода.

Сформулируем основные цели каждой из фаз компиляции. Мы продемонстрируемпреобразования, которым подвергается исходная программа на перечисленных фазахкомпиляции, на небольшом примере — мы рассмотрим оператор присваиванияposition = initial + rate \* 60, причем предположим, что все переменные вещественные.

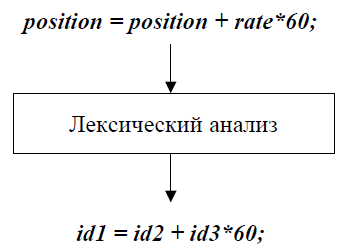
1. **Лексический и синтаксический анализаторы, генерация и оптимизация объектного кода.**

**Лексический анализ**

Входом компилятора служит программа на исходном языке программирования. С точкизрения компилятора это просто последовательность символов. Задача первой фазыкомпиляции, лексического анализатора (lexical analysis), заключается в выделениинекоторых более "крупных" единиц, лексем. Примерами лексем являются ключевыеслова, идентификаторы, константные значения (числа, строки, логические) и т.п.

На этапе лексического анализа обычно также выполняются такие действия, как удалениекомментариев и обработка директив условной компиляции.

Для отображения некоторых лексем достаточно всего одного числа, то естьлексического класса, в то время как для записи других лексем может потребоватьсяпара, состоящая из номера лексического класса и ссылки в таблицу внешнихпредставлений. Хорошая модель лексического анализатора – конечный преобразователь.



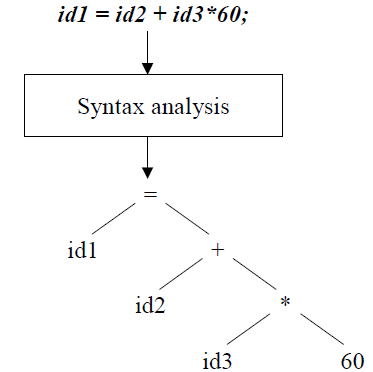
**Синтаксический анализ**

Синтаксический анализатор (syntax analyzer, parser) получает на вход результат работылексического анализатора и разбирает его в соответствии с некоторой грамматикой. Этаграмматика аналогична грамматике, используемой при описании входного языка.

Однако грамматика входного языка обычно не уточняет, какие конструкции следуетсчитать лексемами.

Синтаксический анализ является одной из наиболее формализованных и хорошоизученных фаз компиляции.

После синтаксического анализа можно считать, что исходная программа преобразованав некоторое промежуточное представление. Пока мыостановимся на одной формой промежуточного представления, которая будетиспользована в нашем курсе, – на дереве разбора программы (иногда его такженазывают синтаксическим деревом). В дереве разбора программы внутренние узлысоответствуют операциям, а листья представляют операнды.

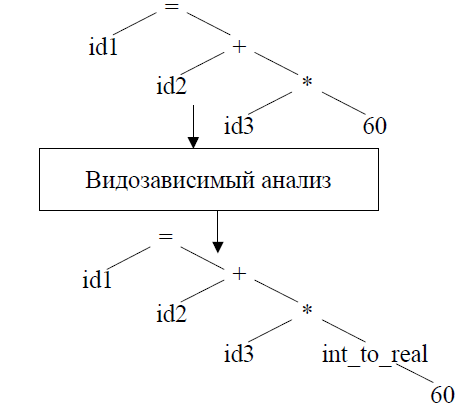


**Видозависимый анализ**

Видозависимый анализ (type checking), иногда также называемый семантическиманализом (semantic analysis), обычно заключается в проверке правильности типовданных, используемых в программе. Кроме того, на этом этапе компилятор должентакже проверить, соблюдаются ли определенные контекстные условия входного языка.

В современных языках программирования одним из примеров контекстных условийможет служить обязательность описания переменных: для каждого использующеговхождения идентификатора должно существовать единственное определяющеевхождение. Другой пример контекстного условия: число и атрибуты фактическихпараметров вызова процедуры должны быть согласованы с определением этойпроцедуры.

Такие контекстные условия не всегда могут быть проверены во время синтаксическогоанализа и потому обычно выделяются в отдельную фазу.



**Оптимизация кода**

Основная цель фазы оптимизации (code optimization) заключается в преобразованиипромежуточного представления программы в целях повышения эффективностирезультирующей объектной программы. Отметим, что существует различные критерииэффкетивности, например, скорость исполнения или объем памяти, требуемыйпрограмме. Очевидно, что все преобразования, осуществляемые на фазе оптимизации,должны приводить к программе, эквивалентной исходной.

Некоторые оптимизации тривиальны, другие требуют достаточно сложного анализапрограммы.

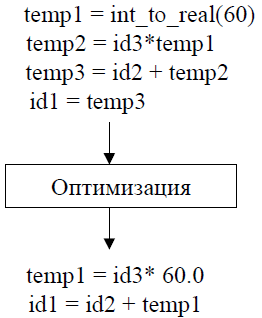
Рассмотрим распространенные оптимизации:

• константные вычисления

• уменьшение силы операций

• выделение общих подвыражений

• чистка циклов и т.д.



**Генерация кода**

Наконец, по оптимизированной версии промежуточного представления генерируетсяобъектная программа. Эту задачу решает фаза генерации кода (code generator). Во многих случаях такой подход может значительно улучшить качествопорождаемого кода.

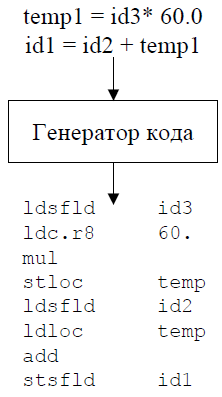
Помимо собственно генерации кода, на этом этапе необходимо решить множествосопутствующих проблем, например:

• распределение памяти, т.е. отображение имен исходной программы в адреса памяти

• распределение регистров, т.е. определение для каждой точки программы множествапеременных, которые должны быть размещены в регистрах

• выбор такой последовательности записи значений в регистры, которая избавила быот необходимости частой выгрузки значений из регистров, а затем повторнойзагрузки

Практически все эти задачи решаются окружением времени исполнения .NET и потомуостались за пределами данного курса.



1. **Конечный автомат как средство построения лексического анализатора.**

Два наиболее распространенных способа конечного задания формального языка - это грамматики и автоматы. Автоматами в данном контексте называют математические модели некоторых вычислительных устройств.

Конечный автомат (finiteautomaton, finite-statemachine) - это пятерка http://www.intuit.ru/img/tex/862654275aa6483a87e6b803564fc6d3.png, где http://www.intuit.ru/img/tex/c32a818e4a2cb9a83da2d3390aa4a488.png- конечный входной алфавит (или просто алфавит ) данного конечного автомата, Q и http://www.intuit.ru/img/tex/8efd8b2448e76d78d147a875d618289d.png- конечные множества,

http://www.intuit.ru/img/tex/8c5d65500ec46ffae630b6150c45681e.png

http://www.intuit.ru/img/tex/957063f9f2b14d94b525e435a096e194.png, http://www.intuit.ru/img/tex/9383f31181bfac777af7b4dbdc7fc36b.png. Элементы Q называются состояниями (state), элементы I - начальными (initial) состояниями, элементы F - заключительными или допускающими (final, accepting) состояниями. Если http://www.intuit.ru/img/tex/2852c4b7c99d23b982a287674841fbc7.png, то http://www.intuit.ru/img/tex/6a90f0be0a4706ddc81f1d000d329cc9.pngназывается переходом (transition) из p в q, а слово x - меткой (label) этого перехода.

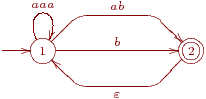
Пример: Пусть Q = {1,2}, http://www.intuit.ru/img/tex/5fc7fb0ee814e66c421345be808c9802.png, I = {1}, F = {2},

http://www.intuit.ru/img/tex/48724eb63ec077ec0378588713db3a79.png

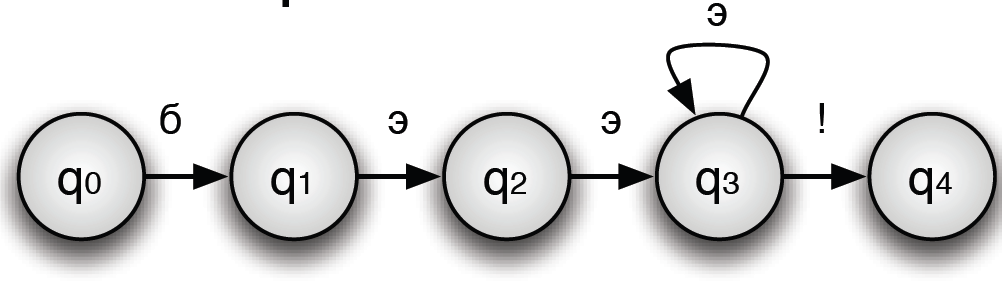
Тогда http://www.intuit.ru/img/tex/289218e86c303ddb80bfdcab6cfae2b9.png- конечный автомат.

Конечные автоматы можно изображать в виде диаграмм состояний (statediagram) (иногда их называют диаграммами переходов (transitiondiagram)). На диаграмме каждое состояние обозначается кружком, а переход - стрелкой. Стрелка из p в q, помеченная словом x, показывает, что http://www.intuit.ru/img/tex/6a90f0be0a4706ddc81f1d000d329cc9.pngявляется переходом данного конечного автомата. Каждое начальное состояние распознается по ведущей в него короткой стрелке. Каждое допускающее состояние отмечается на диаграмме двойным кружком.

На диаграмме изображен конечный автомат из примера.



1. **Формальные грамматики как средство описания синтаксиса языка.**

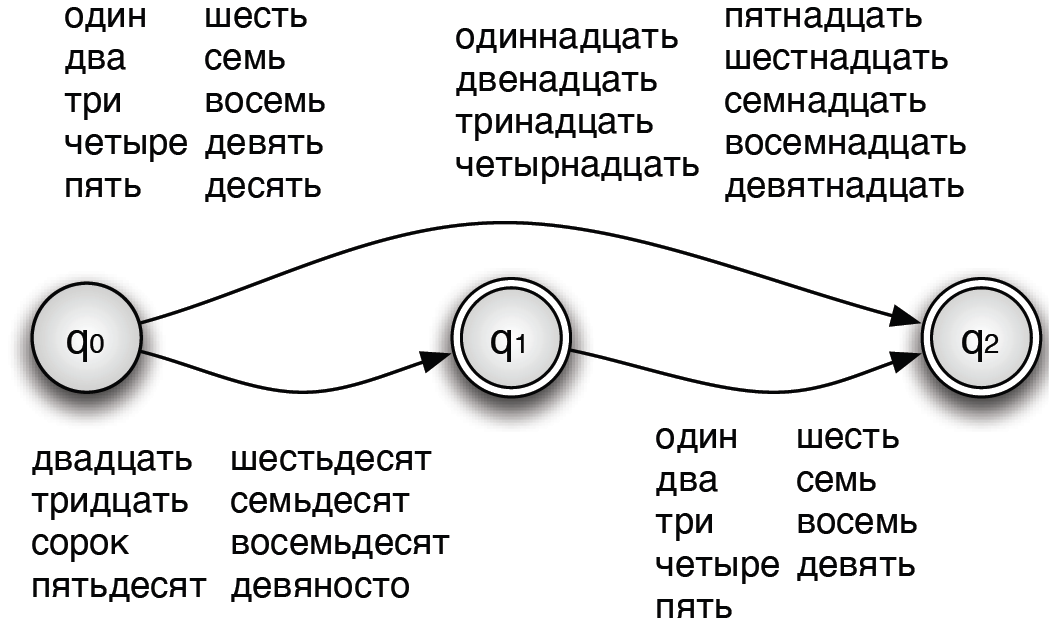


Формальный язык — это множествоконечных слов (строк, цепочек) надконечным алфавитом.

Σ = {a, b, !}

L(m) = {baa!, baaa!, baaaa!, . . .}

Пример формального языка



1. **Описание бесконечного языка конечными средствами.**

Одна из первых задач, возникающих в процессе компиляции - это определение рассматриваемого языка программирования. При рассмотрении языков, состоящих из конечного множества цепочек, проще всего явным образом перечислить все допустимые входные цепочки. Но что делать с языками, не вводящими никаких ограничений на длину входной цепочки? Для потенциально бесконечных языков нам потребуется ввести какой-то конструктивный способ описания, который позволит нам задать правила, описывающие порождаемый ими язык. Такое описание должно удовлетворять некоторым свойствам:

* Само описание должно иметь конечную длину
* Для данного описания языка должен существовать алгоритм, который мог бы проверить принадлежность некоторой входной цепочки языку

Существует целый ряд математических формализмов, в той или иной степени удобных для задания языков - вообще, этап анализа входной программы наиболее разработан и лучше всего поддержан математическими теориями. Наиболее распространенным механизмом являются грамматики, которые задают все подходящие цепочки языка с помощью некоторых порождающих правил. Очевидное достоинство грамматик заключается в том, что существует множество систем, которые по заданной грамматике генерируют программу, проверяющую соответствие входной цепочки определяемому языку. Более того, полезную работу синтаксического анализатора (например, построение дерева разбора) можно проводить параллельно с самим распознаванием языка.

Другая часто используемая идея заключается в том, что создается некоторый обобщенный алгоритм, проверяющий за конечное число шагов принадлежность данной цепочки языку. Такой алгоритм либо останавливается после конечного числа шагов и говорит "да", либо останавливается и говорит "нет". Теоретически, нас могло бы устроить и зацикливание алгоритма на неподходящих входных цепочках, но на практике такое поведение не совсем удобно.

**Контрольные вопросы:**

1. Перечислите компоненты компилирующей программы.

2. Перечислите фазы компиляции.

3. Раскройте понятие и охарактеризуйте лексический анализатор.

4. Раскройте понятие и охарактеризуйте синтаксический анализатор.

5. Раскройте понятие и охарактеризуйте генерацию объектного кода.

6. Раскройте понятие и охарактеризуйте оптимизация объектного кода.

7. Раскройте понятие конечного автомата как средства построения лексического анализатора.

8. Раскройте понятие формальных грамматик как средства описания синтаксиса языка.

**Литература:**

1. А.П. Побегайло, Системное программирование в Windows, БХВ-Петербург, 2006, 1056 с.

2. Ахо, Лам, Сети, Ульман - Компиляторы. Принципы, технологии, инструменты, Изд.: Вильямс, 2008, 1185 с.

3. Л.Бек, Введение в системное программирование, М., Мир, 2008г.

4. Р. И. Компаниец, Системное программирование : Основы построения трансляторов, Изд.: СПб., 2000.

5. Ф.Льюис, Д.Розенкранц, Р.Стирнз. Теоретические основы проектирования компиляторов. - М.: Мир, 20099, 655 с.

6. Э. Таненбау, Современные операционные системы, Изд.: Питер, 2011, 1120 с.

7. А. В. Гордеев, А. Ю. Молчанов, Системное программное обеспечение, учебник для ВУЗов, СПб.: Питер, 2010г.

**СЕМИНАР 5: «ТРАНСЛЯЦИЯ АРИФМЕТИЧЕСКИХ ВЫРАЖЕНИЙ»**

**План семинара:**

1. Методы трансляция арифметических выражений.
2. Польская суффиксная запись как промежуточный код.
3. Автомат с магазинной памятью и его применение для анализа исходного языка.

**Содержание по плану:**

1. **Методы трансляция арифметических выражений.**

Первыми транслирующими программами были программы, обеспечивающие перевод формульной записи выражений в машинный язык. В основе такого перевода представление выражения в виде последовательности троек, где каждая тройка включает два адреса операндов, адрес результата и операцию:

***EC*= *EL*©*ER***, где *EC* – результат, *EL*, *ER* – левый и правый операнды, © – операция.

Основной проблемой при этом является необходимость учета приоритетов операций. Например, для выражения:

***d* = *a*+*b*\**c***должны быть построены тройки:***T*1 = *b*\**c*, *d*= *a*+*T*1**

Исторически первым для решения этой задачи был метод Рутисхаузера.

**Метод Рутисхаузера.** Метод требует, чтобы выражение было записано в полной скобочной записи, когда порядок выполнения операций указывается скобками. Так, выражение ***d*= *a*+*b*\**c***должно быть записано в виде ***d* = *a*+(*b*\**c*)**, в противном случае сначала будет выполняться операция сложения. Метод заключается в следующем.

1. Каждому символу строки ***Si*** ставится в соответствие индекс ***Ni*** по алгоритму:

***N*[0]:=0**

***J*:=1**

**Цикл-пока*S*[*J*]≠’\_’**

**Если*S*[*J*] = ‘(‘** или ***S*[*J*] = <**операнд>

**то *N*[*J*]:=*N*[*J*-1] +1**

**иначе *N*[*J*]:=*N*[*J*-1] -1**

**Все-если**

***J*:=*J*+1**

**Все-цикл**

***N*[*J*]:=0**

1. Определяется наибольшее значение индексав структуре вида ***k*(*k*-1)*k***или при наличии скобок **(*k*-1)*k*(*k*-1)*k*(*k*-1)** и строится соответствующая тройка.

3. Обработанные символы вместе со скобками удаляются, и на их место ставится значение ***N*=*k*-1**.

4. Операции 2, 3 повторяются до завершения выражения.

***Пример*.** (((a+b)\*c)+d)/k

1. **S: ( ( ( a + b ) \* c ) + d ) / k**

**N: 0 1 2 3 4 3 4 3 2 3 21 2 1 0 1 0 => T1 = a+b**

b) **S: ( ( T1 \* c ) + d ) / k**

**N: 0 1 2 3 2 3 21 2 1 0 1 0 => T2 = T1\*c**

c) **S: ( T2 + d ) / k**

**N: 0 1 21 2 1 0 1 0 => T3 = T2+d**

d) **S: T3 / k**

**N: 0 1 0 1 0 =>T4 = T3/k**

Недостаток метода – требование полной скобочной структуры. Как правило, для этого используется специальная программа, которая вставляет скобки.

Решение прочих проблем компиляции было не таким простым. Первые компиляторы являлись крайне сложными программами, написание которых требовало неординарных способностей, и содержали большое количество ошибок. Так первый компилятор Fortran потребовал 18 человеко-лет работы. С тех пор разработаны систематические технологии решения многих задач компиляции, предложен соответствующий инструментарий, в результате чего небольшой компилятор может стать темой курсового проекта.

1. **Польская суффиксная запись как промежуточный код.**

Для описания форм синтаксических деревьев используем мнемонические имена для операндов и операций, подразумевая при этом соответствующую кодировку. Примеры операндов: *A*, *B*, *c*, *d*, и т.д. Примеры операций: *+*, *-*, /, *\**, *PLUS*, *MINUS*, *BRL* (переход на метку), *BM* (переход по минусу), *BZ* (переход по нулю) и т.д.

Для оператора присваивания *A:=B+C\*D\*–E*c инфиксной скобочной записью*A:=B+((C\*D)\*–E))* арифметическое дерево формально определяется так:



1. любая внутренняя вершина помечена операцией;
2. все листья помечены именами пере­менных.

Правое скобочное представление дерева есть



Обратный порядок вершин дерева соответствует пост­фиксному польскому представлению (***польской инверсной записи*** – ПОЛИЗ) 

Поскольку для каждой операции, помечающей вершину, известна арность, то дерево без труда можно восстановить в префиксном польском представлении. Причем, здесь «–» – унарная операция, ее часто обозначают @.

1. **Автомат с магазинной памятью и его применение для анализа исходного языка.**

Распознавание КС-грамматик выполняется автоматом с магазинной памятью. ***Автомат с магазинной памятью*** определяется семеркой:

*PM* = (*Q*, S, Г, d, *q*0, *z*0, *F*),

где *Q* – конечное множество состояний автомата;

S – конечный входной алфавит;

Г – конечное множество магазинных символов;

d (*q*, *ck*, *zj*) – функция переходов;

*q*0 Î *Q* – начальное состояние автомата;

*z*0 Î Г – символ, находящийся в магазине в начальный момент,

*F* Í *Q* – множество заключительных (допускающих) состояний.

**Контрольные вопросы:**

1. Охарактеризуйте методы трансляция арифметических выражений.

2. Раскройте понятие польской суффиксной записи.

3. Опишите автомат с магазинной памятью.

**Литература:**

1. А.П. Побегайло, Системное программирование в Windows, БХВ-Петербург, 2006, 1056 с.

2. Ахо, Лам, Сети, Ульман - Компиляторы. Принципы, технологии, инструменты, Изд.: Вильямс, 2008, 1185 с.

3. Л.Бек, Введение в системное программирование, М., Мир, 2008г.

4. Р. И. Компаниец, Системное программирование : Основы построения трансляторов, Изд.: СПб., 2000.

5. Ф.Льюис, Д.Розенкранц, Р.Стирнз. Теоретические основы проектирования компиляторов. - М.: Мир, 20099, 655 с.

6. Э. Таненбау, Современные операционные системы, Изд.: Питер, 2011, 1120 с.

7. А. В. Гордеев, А. Ю. Молчанов, Системное программное обеспечение, учебник для ВУЗов, СПб.: Питер, 2010г.

**СЕМИНАР №6: «РАЗБОР ЦЕПОЧЕК ЯЗЫКА»**

**План семинара:**

1. Нисходящий и восходящий грамматический разбор, правый и левый выводы.
2. Дерево грамматического разбора цепочек языка.

Содержание по плану:

1. **Нисходящий и восходящий грамматический разбор, правый и левый выводы.**

*Нисходящий разбор (деревом)* цепочки *w* – это процесс построения дерева разбора для *w*, который начинается с корня, и на каждом шаге (развертки) в дерево добавляются все сыновья одного нетерминального узла.

*Восходящий разбор (деревом)* цепочки *w* – это процесс построения дерева разбора для *w*, который начинается с листьев (тривиальных деревьев), и на каждом шаге (свертки) в дерево добавляется узел, сыновьями которого являются вершины уже построенных поддеревьев.

Эти два способа лежат в основе практических методов синтаксического анализа.

Примеры

КС-грамматика, описывающая язык арифметических выраженийнад переменными a:

G = <**T** = {a, +, -, \*, /, (, ) },

**N** ={ E, T, F },

**S**= E,

**P** = { E→E+E | E–E | E\*E | E/E | (E) | a } >

Выводы разверткой для цепочки a+a\*a:

–произвольный: E⇒E\*E⇒E\*a⇒E+E\*a⇒a+E\*a⇒a+a\*a

–левосторонний: E⇒*l*E+E⇒*l*a+E⇒*l*a+E\*E⇒*l*a+a\*E⇒*l*a+a\*a

–правосторонний: E⇒*r*E+E⇒*r*E+E\*E⇒*r*E+E\*a⇒*r*E+a\*a⇒*r*a+a\*a

Два различных дерева вывода, иллюстрирующие неоднозначность G:

1: E 2: E семантически соответствуют:

/|\ /|\ 1: a+(a\*a)

E **+** E E**\*E** 2: (a+a)\*a

| /|\ /|\ |

**aE\*E**E **+ E** a выделеносечение: a+E\*E

| | | |

aa**a**a

Нисходящийвыводдеревомдляцепочкиa+a\*a:

E**E**EEEE

/|\ /|\ /|\ /|\ /|\

E + E **E** + E E + **E**E + E E + E

| | /|\ | /|\ | /|\

aa E \* E a **E** \* E a E \* **E**

| | |

aaa

Восходящий вывод деревом для цепочки a+a\*a:

**E**

/|\

**E**//E

/|\ | | /|\

**E**E**E**EE**E**EE|EE | E|E

| | | | | | | ||| | | |||

a + a\*a a + a\*a a + a\*a a + a\*a a + a\*a a + a\*a

1. **Дерево грамматического разбора цепочек языка.**

Графическим способом отображения процесса разбора цепочек является дерево разбора (или дерево вывода).

Деревом разбора грамматики называется дерево, которое соответствует некоторой цепочке вывода и удовлетворяет следующим условиям:

* каждая вершина дерева обозначается символом грамматики ;
* корнем дерева является вершина, обозначенная начальным символом грамматики S;
* листьями дерева (концевыми вершинами) являются вершины, обозначенные терминальными символами грамматики или символом пустой строки e;
* если некоторый узел дерева обозначен символом , а связанные с ним узлы – символами , то в грамматике существует правило .

Дерево разбора можно построить двумя способами: сверху вниз и снизу вверх.

Пример:

S->Aa|Ab|aa|cc

A->BA|a

B->BC|x

C->d|f

E->cH|r

H->Ef|dB|d

Дереворазбора:

S

/ \

A \

/ \ \

B A \

/ \ \ \

B C \ \

| | | |

x d a a

**Контрольные вопросы:**

1.Охараеткризуйте нисходящий грамматический разбор.

2. Охарактеризуйте восходящий грамматический разбор.

3. Охараеткризуйте правый вывод.

4. Охараеткризуйте левый вывод.

5. Опишите процесс построения дерева грамматического разбора цепочек языка.

**Литература:**

1. А.П. Побегайло, Системное программирование в Windows, БХВ-Петербург, 2006, 1056 с.

2. Ахо, Лам, Сети, Ульман - Компиляторы. Принципы, технологии, инструменты, Изд.: Вильямс, 2008, 1185 с.

3. Л.Бек, Введение в системное программирование, М., Мир, 2008г.

4. Р. И. Компаниец, Системное программирование : Основы построения трансляторов, Изд.: СПб., 2000.

5. Ф.Льюис, Д.Розенкранц, Р.Стирнз. Теоретические основы проектирования компиляторов. - М.: Мир, 20099, 655 с.

6. Э. Таненбау, Современные операционные системы, Изд.: Питер, 2011, 1120 с.

7. А. В. Гордеев, А. Ю. Молчанов, Системное программное обеспечение, учебник для ВУЗов, СПб.: Питер, 2010г.

**СЕМИНАР №6: «ОБЗОР МЕТОДОВ ГЕНЕРАЦИИ И ОПТИМИЗАЦИИ КОДА».**

Генерация объектного кода - это перевод компилятором внутреннего представления исходной программы в результирующую объектную программу на языке ассемблера или непосредственно на машинном языке (машинных кодах).

Генерация объектного кода выполняется после того, как выполнен синтаксический анализ программы и все необходимые действия по подготовке к генерации кода: распределено адресное пространство под функции и переменные, проверено соответствие имен и типов переменных, констант и функций в синтаксических конструкциях исходной программы и т.д.

Оптимизация программы - это обработка, связанная с переупорядочиванием и изменением операций в компилируемой программе с целью получения более эффективной результирующей объектной программы. Оптимизация выполняется на этапах подготовки к генерации и непосредственно при генерации объектного кода.

Лучшие оптимизирующие компиляторы могут получать объектные программы из сложных исходных программ, написанных на языках высокого уровня, почти не уступающие по качеству программам на языке ассемблера. Временные и трудовые затраты на создание такой программы существенно меньше, чем при ее реализации на ассемблере. У современных компиляторов существуют возможности выбора тех или иных критериев оптимизации, исходя из которых оценивается эффективность объектной программы. Так, с одной стороны, возможна оптимизация с минимизацией размера программы, с другой стороны - оптимизация с увеличением скорости ее выполнения. При этом не требуется изменять текст программы на исходном языке.

Все эти преимущества говорят в пользу применения оптимизации. Единственным, но существенным недостатком оптимизации является необходимость тщательной ее проработки при создании компилятора. Используемые методы оптимизации ни при каких условиях не должны приводить к изменению “смысла” исходной программы (т.е. к таким ситуациям, когда результат выполнения программы изменяется после ее оптимизации). К сожалению, не все методы оптимизации, используемые создателями компиляторов, могут быть теоретически обоснованы и доказаны для всех возможных видов исходных программ. Поэтому большинство компиляторов предусматривает возможность отключать те или иные из возможных методов оптимизации. (Часто при оптимизации компиляторы выдают предупреждения разработчику программы, если тот или иной ее участок вызывает подозрения в отношении правильности его “смысла”). Применение оптимизации также нецелесообразно в процессе отладки исходной программы.

Различаются две основные категории оптимизирующих преобразований:

· преобразования исходной программы (в форме ее внутреннего представления в компиляторе), не зависящие от результирующего объектного языка;

· преобразования результирующей объектной программы.

Последний тип преобразований может зависеть не только от свойств объектного языка (что очевидно), но и от архитектуры вычислительной системы, на которой будет выполняться результирующая программа. (Так, например, при оптимизации может учитываться объем кэш-памяти и методы организации конвейерных операций центрального процессора). Этот тип преобразований здесь рассматриваться не будет. Именно эти преобразования могут повлиять на “смысл” исходной программы. В большинстве случаев они являются “ноу-хау” производителей компиляторов и строго ориентированы на определенные архитектуры вычислительных машин.

Методы преобразования программы зависят от типов синтаксических конструкций исходного языка. Теоретически разработаны методы оптимизации для многих типовых конструкций языков программирования. Далее будут рассмотрены только методы оптимизации линейных участков - они встречаются в любой программе и составляют существенную часть программного кода.

Линейный участок программы - это выполняемая по порядку последовательность операций имеющая один вход и один выход. Чаще всего линейный участок содержит последовательность арифметических операций и операторов присвоения значений переменным.

**Контрольные вопросы:**

1.Перечислите и охарактеризуйте методы генерации кода.

2. Перечислите и охарактеризуйте методы оптимизации кода.

**Литература:**

1. А.П. Побегайло, Системное программирование в Windows, БХВ-Петербург, 2006, 1056 с.

2. Ахо, Лам, Сети, Ульман - Компиляторы. Принципы, технологии, инструменты, Изд.: Вильямс, 2008, 1185 с.

3. Л.Бек, Введение в системное программирование, М., Мир, 2008г.

4. Р. И. Компаниец, Системное программирование : Основы построения трансляторов, Изд.: СПб., 2000.

5. Ф.Льюис, Д.Розенкранц, Р.Стирнз. Теоретические основы проектирования компиляторов. - М.: Мир, 20099, 655 с.

6. Э. Таненбау, Современные операционные системы, Изд.: Питер, 2011, 1120 с.

7. А. В. Гордеев, А. Ю. Молчанов, Системное программное обеспечение, учебник для ВУЗов, СПб.: Питер, 2010г.

**СЕМИНАР №7: «ВВЕДЕНИЕ В ОПЕРАЦИОННЫЕ СИСТЕМЫ».**

**План семинара:**

1. Общие характеристика, назначение и возможности операционных систем.
2. Критерии классификации ОС.
3. Назначение, возможности и функции операционных систем.
4. Особенности организации семейства Windows-подобных систем.

**Содержание по плану:**

1. **Общие характеристика, назначение и возможности операционных систем.**

В настоящее время существует большое количество различных типов операционных систем, отличающихся областями применения, аппаратными платформами, способами реализации и др. Назначение операционных систем можно разделить на четыре основные составляющие.

1. Организация (обеспечение) удобного интерфейса между приложениями и пользователями, с одной стороны, и аппаратурой компьютера – с другой. Вместо реальной аппаратуры компьютера ОС представляет пользователю расширенную виртуальную машину, с которой удобнее работать и которую легче программировать. Вот список основных сервисов, предоставляемых типичными операционными системами.

Разработка программ: ОС представляет программисту разнообразные инструменты разработки приложений: редакторы, отладчики и т.п. Ему не обязательно знать, как функционируют различные электронные и электромеханические узлы и устройства компьютера. Часто пользователь не знает даже системы команд процессора, поскольку он может обойтись мощными высокоуровневыми функциями, которые представляет ОС.

Исполнение программ. Для запуска программы нужно выполнить ряд действий: загрузить в основную память программу и данные, инициализировать устройства ввода-вывода и файлы, подготовить другие ресурсы. ОС выполняет всю эту рутинную работу вместо пользователя.

Доступ к устройствам ввода-вывода. Для управления каждым устройством используется свой набор команд. ОС предоставляет пользователю единообразный интерфейс, который скрывает все эти детали и обеспечивает программисту доступ к устройствам ввода-вывода с помощью простых команд чтения и записи. Если бы программист работал непосредственно с аппаратурой компьютера, то для организации, например, чтения блока данных с диска ему пришлось бы использовать более десятка команд с указанием множества параметров. После завершения обмена программист должен был бы предусмотреть еще более сложный анализ результата выполненной операции.

Контролируемый доступ к файлам. При работе с файлами управление со стороны ОС предполагает не только глубокий учет природы устройства ввода-вывода, но и знание структур данных, записанных в файлах. Многопользовательские ОС, кроме того, обеспечивают механизм защиты при обращении к файлам.

Системный доступ. ОС управляет доступом к совместно используемой или общедоступной вычислительной системе в целом, а также к отдельным системным ресурсам. Она обеспечивает защиту ресурсов и данных от несанкционированного использования и разрешает конфликтные ситуации.

Обнаружение ошибок и их обработка. При работе компьютерной системы могут происходить разнообразные сбои за счет внутренних и внешних ошибок в аппаратном обеспечении, различного рода программных ошибок (переполнение, попытка обращения к ячейке памяти, доступ к которой запрещен и др.). В каждом случае ОС выполняет действия, минимизирующие влияние ошибки на работу приложения (от простого сообщения об ошибке до аварийной остановки программы).

Учет использования ресурсов. Хорошая ОС имеет средства учета использования различных ресурсов и отображения параметров производительности вычислительной системы. Эта информация важна для настройки (оптимизации) вычислительной системы с целью повышения ее производительности.

В результате реальная машина, способная выполнить только небольшой набор элементарных действий (машинных команд), с помощью операционной системы превращается в виртуальную машину, выполняющую широкий набор гораздо более мощных функций. Виртуальная машина тоже управляется командами, но уже командами более высокого уровня, например: удалить файл с определенным именем, запустить на выполнение прикладную программу, повысить приоритет задачи, вывести текст файла на печать и т.д. Таким образом, назначение ОС состоит в предоставлении пользователю (программисту) некоторой расширенной виртуальной машины, которую легче программировать и с которой легче работать, чем непосредственно с аппаратурой, составляющей реальный компьютер, систему или сеть.

2. Организация эффективного использования ресурсов компьютера. ОС не только представляет пользователям и программистам удобный интерфейс к аппаратным средствам компьютера, но и является своеобразным диспетчером ресурсов компьютера. К числу основных ресурсов современных вычислительных систем относятся процессоры, основная память, таймеры, наборы данных, диски, накопители на МЛ, принтеры, сетевые устройства и др. Эти ресурсы распределяются операционной системой между выполняемыми программами. В отличие от программы, которая является статическим объектом, выполняемая программа – это динамический объект, он называется процессом и является базовым понятием современных ОС.

Управление ресурсами вычислительной системы с целью наиболее эффективного их использования является вторым назначением операционной системы. Критерии эффективности, в соответствии с которыми ОС организует управление ресурсами компьютера, могут быть различными. Например, в одних системах важен такой критерий, как пропускная способность вычислительной систем, в других – время ее реакции. Зачастую ОС должны удовлетворять нескольким, противоречащим друг другу критериям, что доставляет разработчикам серьезные трудности.

Управление ресурсами включает решение ряда общих, не зависящих от типа ресурса задач:

* планирование ресурса – определение, какому процессу, когда и в каком качестве (если ресурс может выделяться частями) следует выделить данный ресурс;
* удовлетворение запросов на ресурсы – выделение ресурса процессам;
* отслеживание состояния и учет использования ресурса – поддержание оперативной информации о занятости ресурса и распределенной его доли;
* разрешение конфликтов между процессами, претендующими на один и тот же ресурс.

Для решения этих общих задач управления ресурсами разные ОС используют различные алгоритмы, особенности которых, в конечном счете, определяют облик ОС в целом, включая характеристики производительности, область применения и даже пользовательский интерфейс. Таким образом, управление ресурсами составляют важное назначение ОС. В отличие от функций расширенной виртуальной машины большинство функций управления ресурсами выполняются операционной системой автоматически и прикладному программисту недоступны.

3. Облегчение процессов эксплуатации аппаратных и программных средств вычислительной системы. Ряд операционных систем имеет в своем составе наборы служебных программ, обеспечивающие резервное копирование, архивацию данных, проверку, очистку и дефрагментацию дисковых устройств и др.

Кроме того, современные ОС имеют достаточно большой набор средств и способов диагностики и восстановления работоспособности системы. Сюда относятся:

* диагностические программы для выявления ошибок в конфигурации ОС;
* средства восстановления последней работоспособной конфигурации;
* средства восстановления поврежденных и пропавших системных файлов и др.

4. Возможность развития. Современные ОС организуются таким образом, что допускают эффективную разработку, тестирование и внедрение новых системных функций, не прерывая процесса нормального функционирования вычислительной системы. Большинство операционных систем постоянно развиваются (нагляден пример Windows). Происходит это в силу следующих причин.

Обновление и возникновение новых видов аппаратного обеспечения. Например, ранние версии ОС UNIX и OS/2 не использовали механизмы страничной организации памяти (что это такое, мы рассмотрим позже), потому, что они работали на машинах, не обеспеченных соответствующими аппаратными средствами.

Новые сервисы. Для удовлетворения пользователей или нужд системных администраторов ОС должны постоянно предоставлять новые возможности. Например, может потребоваться добавить новые инструменты для контроля или оценки производительности, новые средства ввода-вывода данных (речевой ввод). Другой пример – поддержка новых приложений, использующих окна на экране дисплея.

Исправления. В каждой ОС есть ошибки. Время от времени они обнаруживаются и исправляются. Отсюда постоянные появления новых версий и редакций ОС. Необходимость регулярных изменений накладывает определенные требования на организацию операционных систем. Очевидно, что эти системы (как, впрочем, и другие сложные программы системы) должны иметь модульную структуру с четко определенными межмодульными связями (интерфейсами). Важную роль играет хорошая и полная документированность системы.

Перейдем к рассмотрению состава компонентов и функций ОС. Современные операционные системы содержат сотни и тысячи модулей (например, W2000 содержит 29 млн строк исходного кода на языке С). Функции ОС обычно группируются либо в соответствии с типами локальных ресурсов, которыми управляет ОС, либо в соответствии со специфическими задачами, применимыми ко всем ресурсам. Совокупности модулей, выполняющих такие группы функций, образуют подсистемы операционной системы.

Наиболее важными подсистемами управления ресурсами являются подсистемы управления процессами, памятью, файлами и внешними устройствами, а подсистемами, общими для всех ресурсов, являются подсистемы пользовательского интерфейса, защиты данных и администрирования.

Управление процессами. Подсистема управления процессами непосредственно влияет на функционирование вычислительной системы. Для каждой выполняемой программы ОС организует один или более процессов. Каждый такой процесс представляется в ОС информационной структурой (таблицей, дескриптором, контекстом процессора), содержащей данные о потребностях процесса в ресурсах, а также о фактически выделенных ему ресурсах (область оперативной памяти, количество процессорного времени, файлы, устройства ввода-вывода и др.). Кроме того, в этой информационной структуре хранятся данные, характеризующие историю пребывания процесса в системе: текущее состояние (активное или заблокированное), приоритет, состояние регистров, программного счетчика и др.

В современных мультипрограммных ОС может существовать одновременно несколько процессов, порожденных по инициативе пользователей и их приложений, а также инициированных ОС для выполнения своих функций (системные процессы). Поскольку процессы могут одновременно претендовать на одни и те же ресурсы, подсистема управления процессами планирует очередность выполнения процессов, обеспечивает их необходимыми ресурсами, обеспечивает взаимодействие и синхронизацию процессов.

Управление памятью. Подсистема управления памятью производит распределение физической памяти между всеми существующими в системе процессами, загрузку и удаление программных кодов и данных процессов в отведенные им области памяти, настройку адресно-зависимых частей кодов процесса на физические адреса выделенной области, а также защиту областей памяти каждого процесса. Стратегия управления памятью складывается из стратегий выборки, размещения и замещения блока программы или данных в основной памяти. Соответственно используются различные алгоритмы, определяющие, когда загрузить очередной блок в память (по запросу или с упреждением), в какое место памяти его поместить и какой блок программы или данных удалить из основной памяти, чтобы освободить место для размещения новых блоков.

Одним из наиболее популярных способов управления памятью в современных ОС является виртуальная память. Реализация механизма виртуальной памяти позволяет программисту считать, что в его распоряжении имеется однородная оперативная память, объем которой ограничивается только возможностями адресации, предоставляемыми системой программирования.

Важная функция управления памятью – защита памяти. Нарушения защиты памяти связаны с обращениями процессов к участкам памяти, выделенной другим процессам прикладных программ или программ самой ОС. Средства защиты памяти должны пресекать такие попытки доступа путем аварийного завершения программы-нарушителя.

Управление файлами. Функции управления файлами сосредоточены в файловой системе ОС. Операционная система виртуализирует отдельный набор данных, хранящихся на внешнем накопителе, в виде файла – простой неструктурированной последовательности байтов, имеющих символьное имя. Для удобства работы с данными файлы группируются в каталоги, которые, в свою очередь, образуют группы – каталоги более высокого уровня. Файловая система преобразует символьные имена файлов, с которыми работает пользователь или программист, в физические адреса данных на дисках, организует совместный доступ к файлам, защищает их от несанкционированного доступа.

Управление внешними устройствами. Функции управления внешними устройствами возлагаются на подсистему управления внешними устройствами, называемую также подсистемой ввода-вывода. Она является интерфейсом между ядром компьютера и всеми подключенными к нему устройствами. Спектр этих устройств очень обширен (принтеры, сканеры, мониторы, модемы, манипуляторы, сетевые адаптеры, АЦП разного рода и др.), сотни моделей этих устройств отличаются набором и последовательностью команд, используемых для обмена информацией с процессором и другими деталями.

Программа, управляющая конкретной моделью внешнего устройства и учитывающая все его особенности, называется драйвером. Наличие большого количества подходящих драйверов во многом определяет успех ОС на рынке. Созданием драйверов занимаются как разработчики ОС, так и компании, выпускающие внешние устройства. ОС должна поддерживать четко определенный интерфейс между драйверами и остальными частями ОС. Тогда разработчики компаний-производителей устройств ввода-вывода могут поставлять вместе со своими устройствами драйверы для конкретной операционной системы.

Защита данных и администрирование. Безопасность данных вычислительной системы обеспечивается средствами отказоустойчивости ОС, направленными на защиту от сбоев и отказов аппаратуры и ошибок программного обеспечения, а также средствами защиты от несанкционированного доступа. Для каждого пользователя системы обязательна процедура логического входа, в процессе которой ОС убеждается, что в систему входит пользователь, разрешенный административной службой. Администратор вычислительной системы определяет и ограничивает возможности пользователей в выполнении тех или иных действий, т.е. определяет их права по обращению и использованию ресурсов системы.

Важным средством защиты являются функции аудита ОС, заключающегося в фиксации всех событий, от которых зависит безопасность системы. Поддержка отказоустойчивости вычислительной системы реализуется на основе резервирования (дисковые RAID-массивы, резервные принтеры и другие устройства, иногда резервирование центральных процессоров, в ранних ОС – дуальные и дуплексные системы, системы с мажоритарным органом и др.). Вообще обеспечение отказоустойчивости системы – одна из важнейших обязанностей системного администратора, который для этого использует ряд специальных средств и инструментов.

Интерфейс прикладного программирования. Прикладные программисты используют в своих приложениях обращения к операционной системе, когда для выполнения тех или иных действий им требуется особый статус, которым обладает только ОС. Возможности операционной системы доступны программисту в виде набора функций, который называется интерфейсом прикладного программирования (Application Programming Interface, API). Приложения обращаются к функциям API с помощью системных вызовов. Способ, которым приложение получает услуги операционной системы, очень похож на вызов подпрограмм.

Способ реализации системных вызовов зависит от структурной организации ОС, особенностей аппаратной платформы и языка программирования.

В ОС UNIX системные вызовы почти идентичны библиотечным процедурам. Ситуация в Windows иная (более подробно это рассмотрим далее).

Пользовательский интерфейс. ОС обеспечивает удобный интерфейс не только для прикладных программ, но и для пользователя (программиста, администратора). В ранних ОС интерфейс сводился к языку управления заданиями и не требовал терминала. Команды языка управления заданиями набивались на перфокарты, а результаты выполнения задания выводились на печатающее устройство.

Современные ОС поддерживают развитые функции пользовательского интерфейса для интерактивной работы за терминалами двух типов: алфавитно-цифрового и графического. При работе за алфавитно-цифровым терминалом пользователь имеет в своем распоряжении систему команд, развитость которой отражает функциональные возможности данной ОС. Обычно командный язык ОС позволяет запускать и останавливать приложения, выполнять различные операции с каталогами и файлами, получать информацию о состоянии ОС, администрировать систему. Команды могут вводиться не только в интерактивном режиме с терминала, но и считываться из так называемого командного файла, содержащего некоторую последовательность команд.

Программный модуль ОС, ответственный за чтение отдельных команд или же последовательности команд из командного файла, иногда называют командным интерпретатором (в MS-DOS – командным процессором).

Вычислительные системы, управляемые из командной строки, например UNIX-системы, имеют командный интерпретатор, называемый оболочкой (Shell). Она, собственно, не входит в состав ОС, но пользуется многими функциями операционной системы. Когда какой-либо пользователь входит в систему, запускается оболочка. Стандартным терминалом для нее является монитор с клавиатурой. Оболочка начинает работу с печати приглашения (prompt) – знака доллара (или иного знака), говорящего пользователю, что оболочка ожидает ввода команды (аналогично управляется MS-DOS). Если теперь пользователь напечатает какую-либо команду, оболочка создает системный вызов и ОС выполнит эту команду. После завершения оболочка опять печатает приглашение и пытается прочесть следующую входную строку.

Ввод команд может быть упрощен, если операционная система поддерживает графический пользовательский интерфейс. В этом случае пользователь выбирает на экране нужный пункт меню или графический символ (так это происходит, например, в ОС Windows).

1. **Критерии классификации ОС.**

Операционные системы классифицируются по:

* количеству одновременно работающих пользователей: однопользовательские, многопользовательские;
* числу процессов, одновременно выполняемых под управлением системы;
* количеству решаемых задач: однозадачные, многозадачные;
* количеству поддерживаемых процессоров: однопроцессорные, многопроцессорные;
* разрядности кода ОС: 8-разрядные, 16-разрядные, 32-разрядные, 64-разрядные;
* типу интерфейса: командные (текстовые) и объектно-ориентированные (графические);
* типу доступа пользователя к ЭВМ: с пакетной обработкой, с разделением времени, реального времени;
* типу использования ресурсов: сетевые, локальные.

**3. Назначение, возможности и функции операционных систем.**

В 1965-1975 гг. произошел переход от отдельных полупроводниковых элементов к интегральным микросхемам, что открыло путь к появлению следующего типа компьютеров. Большие функциональные возможности интегральных схем сделали возможным осуществление на практике сложных компьютерных архитектур, таких, например, как IBM/360. В это период были реализованы практически все основные механизмы, присущие современным ОС: мультипрограммирование, мультипроцессирование, поддержка многотерминального многопользовательского режима, виртуальная память, файловые системы, разграничение доступа и сетевая работа.. **Революционным событием данного этапа явилась промышленная реализация мультипрограммирования** – способа организации вычислительного процесса, при котором в памяти компьютера находилось одновременно несколько попеременно выполняющихся программ. Такое усовершенствование значительно улучшило эффективность вычислительной системы – компьютер использовался теперь постоянно. В начале 70-х годов появились первые сетевые ОС, которые в отличие от многотерминальных ОС позволяли не только рассредоточить пользователей, но и организовать распределенное хранение и обработку данных между несколькими компьютерами, связанными электрическими сетями. К середине 70-х годов наряду с мэйнфрэймами широкое распространение получили такие мини-компьютеры, как PDP-11, NOVA, HP. Мини-компьютеры первыми использовали преимущества больших интегральных схем, позволившие реализовать достаточно мощные функциональные возможности при сравнительно невысокой стоимости.Важным событием в истории мини-компьютеров и вообще в истории ОС явилось создание ОС UNIX. Первоначальна она предназначалась для поддержания режима разделения времени в мини-компьютере PDP-11. С середины 70-х годов началось массовое использование ОС UNIX. В 80-е годы был разработан протокол2 TCP/IP, признанный в 1983 Министерством обороны США военным стандартом и реализованный для ОС BSD UNIX. Все это десятилетие отмечалось постоянным появлением новых, все более совершенных версий ОС UNIX. Среди них были и фирменные версии UNIX: SunOS, HP-UX, Irix, AIX и многие другие, к которым производители компьютеров адаптировали код ядра и системных утилит для своей аппаратуры. Разнообразие версий породило проблему их совместимости, периодически решаемую различными организациями. В результате этой работы появились стандарты POSIX и XPG, определяющие интерфейсы ОС для приложений, а специальное подразделение компании AT&T выпустило несколько версий UNIX System III и UNIX System V, предназначенных для консолидации разработчиков на уровне кода ядра. Начало 80-х годов также ознаменовалось важным событием в истории ОС – появились персональные компьютеры. Они стали широко использоваться неспециалистами, что потребовало разработки так называемого дружественного программного обеспечения. Предоставление этих дружественных функций стало, естественно, прямой обязанностью ОС. Первая версия наиболее популярной ОС раннего этапа развития персональных компьютеров – MS-DOS компании Microsoft – была лишена этих функциональных возможностей. В 1987 году в результате совместных усилий Microsoft и IBM появилась на свет первая многозадачная ОС для персональных компьютеров OS/2 с процессором Intel 80286, в полной мере использующая возможности защищенного режима. В 90-е годы практически все ОС, занимающие заметное место на рынке, стали сетевыми. Сетевые функции сегодня являются неотъемлемой частью ОС и встраиваются в ее ядро. ОС получили средства для работы со всеми основными технологиями локальных (Ethernet, FastEthernet, GigabitEthernet, TokenRing, FDDI, ATM) и глобальных (X.25, framerelay, ISDN, ATM) сетей, а также средства для создания составных сетей (IP, IPX, AppleTalk, RIP, OSPF, NLSP

Во второй половине 90-х годов все компании-производители ОС резко усилили поддержку своими системами средств работы с Интернетом. Кроме самого стека протоколов TCP/IP, в комплект поставки начали включать утилиты, реализующие такие популярные сервисы, как telnet, ftp, DNS и Web.

Рассмотрев краткую историю развития операционных систем, мы можем более подробно остановиться на обсуждении **выполняемых ими функций.**

**Операционная система** (ОС) компьютера представляет собой комплекс программ, организующих вычислительный процесс в вычислительной системе.

ОС выполняет две группы функций:

1) предоставление пользователю вместо реальной аппаратуры компьютера некой расширенной машины, с которой удобнее работать и которую легче программировать;

2) повышение эффективности использования компьютера путем рационального управления его ресурсами в соответствии с некоторым критерием.

ОС не только предоставляет пользователям и программистам удобный интерфейс к аппаратным средствам компьютера, но и является механизмом, распределяющим ресурсы компьютера. К **ресурсам** вычислительной системы относят такие ее средства, которые могут быть выделены процессу обработки данных.

Ресурсы вычислительной системы можно разбить на *первичные* – аппаратные ресурсы и *вторичные* – логические, программные и информационные ресурсы.

К числу первичных ресурсов современных вычислительных систем относятся процессоры, основная память, диски и др., за которыми стоят реальные аппаратные средства. Они являются наиболее значимыми для вычислительного процесса.

Вторичные ресурсы связаны с техническими устройствами косвенно, так как являются логическими, виртуальными. Однако их введение – это необходимая абстракция, удобная не только для создателей ОС, но и для пользователей.

**Управление ресурсами включает решение следующих общих, не зависящих от типа ресурса задач**:

1) **планирование ресурса** – определение, какому процессу, когда и в каком количестве (если ресурс может быть разбит на части) следует выделить данный ресурс;

2) удовлетворение запросов на ресурсы;

3) отслеживание состояния и **учет использования ресурса** – поддержание оперативной информации о занятости ресурса и распределенной его доли;

4) разрешение конфликтов между процессами, использующими один ресурс.

Разработка и реализация алгоритмов управления ресурсами является очень важным этапом при проектировании системы.

**Функции ОС обычно** группируются либо в соответствии с типами локальных ресурсов, которыми управляет ОС, либо в соответствии со специфическими задачами, применимыми ко всем ресурсам вычислительной системы. Такие группы именуются **подсистемами**. Наиболее важные – это подсистемы управления процессами, памятью, файлами и внешними устройствами, а подсистемами, общими для всех ресурсов, являются подсистемы пользовательского интерфейса, защиты данных и администрирования. Рассмотрим возложенные на них основные задачи.

*Подсистема управления процессами* генерирует системные информационные структуры, содержащие данные о потребностях в ресурсах вычислительной системы, а также о фактически выделенных ресурсах для каждой задачи. **Процесс** (задача) – представляет собой базовое понятие современных ОС и часто кратко определяется как программа в стадии выполнения. Программа – это статический объект, представляющий собой файл с кодами и данными, процесс является динамическим объектом, который возникает в операционной системе после того, как пользователь или ОС решает запустить программу на выполнение. Во многих современных ОС для обозначения минимальной единицы работы ОС используют термин “нить”, или “поток”, при этом изменяется суть термина “процесс”.

Вмультипрограммной ОС одновременно может существовать несколько процессов, часть из которых, называемая **пользовательскими процессами**, порождается по инициативе пользователей. Другие процессы инициализируются самой ОС для выполнения своих функций и называются **системными**.

Поскольку процессы довольно часто одновременно претендуют на одни и те же ресурсы вычислительной системы, на ОС ложится задача поддержания очередей заявок на такие ресурсы.

ОС берет на себя также функции синхронизации процессов, позволяющие процессу приостанавливать свое выполнение до наступления какого-либо события в системе, которым может быть, например, завершение операции ввода-вывода, осуществляемой ОС по его запросу.

*На подсистему управления памятью*возложены очень важные функции, поскольку процесс может выполняться только в том случае, если его коды и данные находятся в оперативной памяти компьютера. Управление памятью включает в себя распределение имеющейся в вычислительной системе физической памяти между всеми существующими в данный момент в системе процессами, загрузку кодов и данных процессов в отведенные им области памяти, настройку адресно-зависимых частей кодов процесса на физические адреса выделенной области, а также защиту областей памяти каждого процесса. Одним из наиболее популярных способов управления памятью в современных ОС является так называемая *виртуальная память*. Наличие в ОС механизма виртуальной памяти позволяет программисту писать программу так, как будто в его распоряжении имеется однородная оперативная память большого объема, часто существеннопревышающего объем имеющейся физической памяти.

**Защита памяти** – это избирательная способность предохранять выполняемую задачу от операций записи или чтения памяти, назначенной другой задаче. Правильно написанные программы не делают некорректных попыток обратиться к памяти, используемой другими программами. Однако вследствие того, что в реальных программах часто содержатся ошибки, такие попытки могут предприниматься. Средства защиты памяти, реализованные в ОС, должны пресекать несанкционированный доступ процессов к чужим областям памяти.

Способность ОС к “экранированию” сложностей реальной аппаратуры ярко проявляется в одной из основных подсистем ОС – *файловой системе*. ОС виртуализирует отдельный набор данных, хранящихся на внешнем накопителе, в виде *файла* – простой неструктурированной последовательности байтов, имеющей символьное имя. Для удобства работы с данными файлы группируются в *каталоги*, которые, в свою очередь, образуют *группы* – каталоги более высокого уровня. Пользователь, применяя средства ОС, может выполнять над файлами и каталогами такие действия, как поиск по имени, удаление, вывод содержимого на внешнее устройство, изменение и сохранение содержимого

*Подсистема управления внешними устройствами*, называемая также *подсистемой ввода-вывода*, является интерфейсом ко всем устройствам, подключенным к компьютеру. Множество этих устройств очень велико. Номенклатура выпускаемых накопителей на жестких, гибких и оптических дисках принтеров, сканеров, мониторов, плоттеров, модемов, сетевых адаптеров и специализированных устройств ввода-вывода (например, аналого-цифровых преобразователей, устройств считывания шрих-кода и т.д.) насчитывает тысячи моделей. Эти модели отличаются набором и последовательностью команд, используемых для обмена информацией с процессором и памятью компьютера, скоростью работы, кодировкой передаваемых данных, возможностью совместного использования, поддерживаемыми функциональными возможностями и множеством других деталей. Программа, управляющая конкретной моделью внешнего устройства и учитывающая все его особенности, называется **драйвером** этого устройства (от англ. drive – управлять). Драйвер может управлять единственной моделью устройства или же группой устройств определенного типа. Для конечного пользователя важное значение имеет наличие в ОС как можно большего количества разнообразных драйверов (для широкого спектра устройств), поскольку это гарантирует возможность подключения к компьютеру большого числа внешних устройств различных производителей. От наличия подходящих драйверов во многом зависит успех ОС на рынке. Так, отсутствие многих необходимых драйверов внешних устройств было одной из причин низкой популярности ОС OS/2 и ранних версий Windows NT. Поэтому, прежде чем выпускать на рынок новую ОС, разработчики стараются включить в ее состав драйверы для поддержки самого разного имеющегося в продаже оборудования. Созданием драйверов занимаются как разработчики конкретной ОС, так и специалисты компаний, выпускающих внешние устройства. ОС должна поддерживать четко определенный интерфейс между драйверами и остальными частями ОС, чтобы разработчики из компаний-производителей устройств ввода-вывода могли поставлять вместе со своими устройствами драйверы для данной ОС.

**4. Особенности организации семейства Windows-подобных систем.**

ОС Windows *представляют* собой *замкнутую* рабочую среду. Практически все операции (запуск прикладной программы, форматирование дискет, печать текстов и т. д.) могут быть выполнены без выхода из Windows. Система имеет следующие особенности:

Графический *интерфейс.* Основными элементами пользовательского интерфейса являются *Рабочий стол, окно, значок* (пиктограмма, иконка), *меню* и *Панель задач.*Windows в переводе с английского означает «окна». Дадим пояснения предназначению основных элементов. Каждая задача запускается в отдельном окне, которое называется окном приложения, может быть приостановлена при его сворачивании или завершена при закрытии. В строке заголовка окна приложения обычно отображается название запущенной программы и кнопки управления окном.

Каждый объект (файл определённого типа, окно, папка, кнопка и т.д.) изображается конкретным значком или пиктограммой*.* В ОС Windows имеются различные меню: ниспадающие, кнопочные, контекстные.

Например, при выборе одного из пунктов главного меню окна любого приложения, расположенных горизонтально, появляются ниспадающие вертикальные меню следующего уровня.

Кнопочное меню срабатывает при выборе кнопки или списка, оно может быть ниспадающим (список «Шрифт» Ms Word) или иметь вид палитры(список «Цвет шрифта» Ms Word). Чаще всего кнопочные меню входят в состав панелей инструментов. Панель инструментов(ПИ) в ОС Windows представляет собой блок экранных кнопок или значков, позволяющих быстро вызывать часто используемую команду, входящую в одну из групп( Например ПИ «Форматирование» Ms Word).

Контекстное (*контекстно-зависимое меню* (КЗМ)) вызывается правой кнопкой мыши. Его содержание *-* зависит от того, где находится указатель мыши (на рабочем столе, в конкретном окне, на главном меню и т. д.). В дальнейшем фраза: «Выполнить команду КЗМ «Вырезать»» будет означать, что нужно вызвать КЗМ, нажав на правую кнопку мыши, а затем из появившегося списка команд выбрать команду «Вырезать», щёлкнув по ней. А если выполнение команды приводит к вызову меню следующего уровня, то будем говорить: «Выполнить команды КЗМ «…» и «…»».

Реализация графического интерфейса невозможна *без широкого использования манипулятора мышь.* С помощью мыши по экрану перемещается указатель, форма которого может меняться в зависимости от используемой программы или выполняемого действия. В дальнейшем, говоря о мыши, будем использовать следующие термины: щелчок (позволяет: нажать кнопку; выбрать команду меню; выделить файл или папку; установить (снять) флажок; раскрыть вкладку и т.д.); двойной щелчок (позволяет открыть папку или файл, запустить приложение и т.д.), перетаскивание ( позволяет переместить объект или изменить его размеры, перетаскивая границы объекта).

«Щелчок по «\*\*\*»» - означает, что нужно поставить указатель на объект «\*\*\*», а затем нажать и сразу отпустить левую кнопку мыши. «Двойной щелчок по «\*\*\*»» - означает, что нужно поставить указатель на объект «\*\*\*», а затем быстро, с очень коротким интервалом, дважды нажать и отпустить левую клавишу мыши. «Перетаскивание объекта». Применяется при перемещении объекта на новое место или при изменении его размеров. Для осуществления перетаскивания поставьте указатель на объект и нажмите на левую кнопку мыши, а затем, удерживая ее в нажатом состоянии, переместите мышь, в результате чего будет двигаться соответствующий объект. Отпустите кнопку мыши, и объект будет зафиксирован на новом месте. Для изменения размера объекта указатель нужно поставить на его границу, чтобы он принял форму × или, а затем перетащить границу на новое место.

*Windows — интегрированная среда*, под ее управлением могут работать не только специальные программы, разработанные под Windows (Windows-приложения), но и другие программы, в том числе многочисленные DOS-приложения. Кроме того, Windows обеспечивает эффективный и комфортабельный *обмен информацией* между отдельными программами и приложениями, работающими под ее управлением. Для обмена данными между различными приложениями и документами используется достаточно простое, но очень эффективное средство — *буфер обмена,* а также технология *связывания и внедрения* объектов — OLE (Object Linking and Embedding — связь и внедрение объектов). *OLE* — это технология работы с объектами, которые созданы в различных приложениях, универсальный механизм для создания и обработки составных документов, содержащих одновременно объекты различного происхождения, разной природы, например текст, таблицы, фотографии, звук и т. п.

С понятием интегрированности связывают также возможность совместного использования различными программами ресурсов компьютера. Windows обеспечивает оптимальное управление оперативной памятью и эффективный оперативный контроль за работающими приложениями. С точки зрения управления выполнением приложений различают однозадачные и многозадачные операционные системы. *Однозадачные ОС*, например MS DOS, передают все ресурсы вычислительной системы одному исполняемому приложению и не допускают ни параллельного выполнения другого приложения, ни его приостановки и запуска другого приложения. Одной из важнейших особенностей Windows является ее *многозадачность,* т. е. способность одновременно работать с несколькими программами и легко переключаться с одной программы на другую, а также *многопоточность* — свойство ОС разделять работающие программы на несколько частей (потоков), самостоятельно претендующих на процессорное время, и обеспечивать одновременное выполнение программой нескольких не связанных друг с другом операций (потоков).

Последние версии Windows реализуют возможность без проблем подключать новые внешние устройства и обеспечивать самонастройку системы под аппаратные средства (технология Plug and Play — «Включи и работай»).

В Windows предусмотрена возможность широкого представления мультимедийных средств — от воспроизведения звуковых файлов до работы с видеофильмами.

Начиная с Windows 3.11 все последующие версии Windows обеспечивают возможность эффективной работы в локальной сети. Причем ОС Windows имеет вполне законченные средства для работы ПК, как в качестве сервера, так и рабочей станции.

В состав последних версий Windows входят пакеты программ, которые обеспечивают широкие возможности работы в телекоммуникационных средах и в Интернете.

**Контрольные вопросы:**

1.Охараеткризуйте нисходящий грамматический разбор.

2. Охарактеризуйте восходящий грамматический разбор.

3. Охараеткризуйте правый вывод.

4. Охараеткризуйте левый вывод.

5. Опишите процесс построения дерева грамматического разбора цепочек языка.

**Литература:**

1. А.П. Побегайло, Системное программирование в Windows, БХВ-Петербург, 2006, 1056 с.

2. Ахо, Лам, Сети, Ульман - Компиляторы. Принципы, технологии, инструменты, Изд.: Вильямс, 2008, 1185 с.

3. Л.Бек, Введение в системное программирование, М., Мир, 2008г.

4. Р. И. Компаниец, Системное программирование : Основы построения трансляторов, Изд.: СПб., 2000.

5. Ф.Льюис, Д.Розенкранц, Р.Стирнз. Теоретические основы проектирования компиляторов. - М.: Мир, 20099, 655 с.

6. Э. Таненбау, Современные операционные системы, Изд.: Питер, 2011, 1120 с.

7. А. В. Гордеев, А. Ю. Молчанов, Системное программное обеспечение, учебник для ВУЗов, СПб.: Питер, 2010г.

**СЕМИНАР №8: «ВВЕДЕНИЕ В ОПЕРАЦИОННЫЕ СИСТЕМЫ».**

**План семинара:**

1. Принципы построения ОС.

2. Составные части ОС, их назначение, основные функции.

3. Выполнение заданий под управлением операционной системы.

4. Взаимодействие пользователей и прикладных программ с ОС.

**Содержание по плану:**

**1. Принципы построения ОС.**

Среди множества принципов, которые используются при построении ОС, перечислим несколько наиболее важных

**Принцип модульности**

Под ***модулем***в общем случае понимают функционально законченный элемент системы, выполненный в соответствии с принятыми межмодульными интерфейсами.

По своему определению модуль предполагает возможность без труда заменить его на другой при наличии заданных интерфейсов.

Способы обособления составных частей ОС в отдельные модули могут существенно различаться, но чаще всего разделение происходит именно по функциональному признаку. В значительной степени разделение системы на модули определяется используемым методом проектирования ОС (снизу вверх или наоборот).

Мы говорим о системных программных модулях (они являются программными ресурсами и могут быть распределены между выполняющимися процессами).

Программные модули могут быть однократно и многократно используемыми.

**Однократно используемыми** назвают такие модули, которые могут быть правильно выполнены только один раз. Т.е. в процессе выполнения они могут испортить себя (повреждается часть кода или исходные данные). Очевидно, что они являются неделимыми ресурсами. Эти модули обычно используются на этапе загрузки ОС (файлы на системном диске, в которых записаны эти модули при этом не портятся, поэтому могут быть использованы при следующих запусках).

**Многократно используемые** программные модули делятся на

-         *привилегированные,*

-         *повторно входимые* и

-         *реентерабельные* модули.

**Привилегированные** работают в привилегированном режиме (т.е. при отключенной системе прерываний). Таким образом, никакие внешние события не могут нарушить порядок их выполнения). Эти модули являются попеременно разделяемым ресурсом.

**Непривилегированные модули** – это обычные программные модули, которые могут быть прерваны во время своей работы.

(В общем случае их нельзя считать разделяемыми, так как после прерывания выполнения такого модуля, исполняемого в рамках одного процесса, запусить его еще раз по требованию другого процесса, то промежуточные результаты прерванных вычислений могут быть потерянными.

Противоположны им **реентерабельные модули** (reenterable – допускающий повторное обращение).

Они допускают повторное многократное прерывание своего исполнения и повторный запуск.

Для этого они создаются таким образом, чтобы было обеспечено сохранение промежуточных вычислений и возврат с прерванной точки. Это может быть реализовано двумя способами: с помощью статических и динамических методов выделения памяти под сохраняемые значения.

Чаще используется динамический метод.

С помощью обращения из системной привилегированной секции осуществляется запрос на получение в системной области памяти блока ячеек, необходимого для размещения текущих данных. При этом на вершину стека помещается указатель на начало выделенной области и ее объем. Теперь включается система прерываний (завершилась привилегированная част модуля). Поэтому возможно прерывание во время выполнения основной части модуля. Если прерывание не возникает, то после завершения основной части модуля выполняется запрос на освобождение блока системной области памяти. Если возникает прерывание и другой процесс обращается к этому же самому реентерабельному модулю, для нового процесса вновь заказывается новый блок памяти и на вершину стека записывается новый указатель.

При статическом способе выделения памяти заранее для фиксированного числа процессов резервируются области памяти, в которых будут располагаться переменные реентерабельных модулей. Для каждого процесса – своя область.

Чаще всего такими процессами являются процессы ввода\вывода и речь идет о реентерабельных драйверах, которые могут управлять параллельно несколькими однотипными устройствами).

Существуют еще и **повторно входимые** (re-entrance) модули. Они также допускают многократное паралльное использование, но в отличие от реентерабельных, их нельзя прерывать. Они состоят из привилегированных секций и повторное обращение к ним возможно только после завершения какой-нибудь из секций. Другими словами, в повторно входимых модулях четко предопределены все допустимые точки входа.

Заметим, что повторно входимые модули встречаются чаще, чем реентерабельные.

*Принцип модульности отражает технологические и эксплуатационные свойства системы. Наибольший эффект от его использования достижим в случае, когда принцип распространен одновременно на операционную систему, прикладные программы и аппаратуру.*

**Принцип функциональной избирательности**

В ОС выделяется некоторая часть важных модулей, которые должны постоянно находиться в оперативной памяти для более эффективной организации вычислительного процесса. Эту часть в ОС называют ядром, так как это действительно основа системы.

При формировании состава ядра требуется учитывать два противоречивых требования.

1)     В состав ядра должны войти наиболее часто используемые системные модули.

2)     Количество модулей должно быть таковым, чтобы объем памяти, занимаемый ядром, был бы не слишком большим.

В состав ядра, как правило, входят модули

-         по управлению системой прерываний,

-         средства по переводу программ из состояния счета в состояние ожидания, готовности и обратно,

-         средства по распределению таких основных ресурсов, как оперативная память и процессор.

Помимо программных модулей, входящих в состав ядра и постоянно располагающихся в оперативной памяти, может быть много других системных программных модулей, которые получают название *транзитных.* Транзитные программные модули загружаются в оперативную память только при необходимости и в случае отсутствия свободного пространства могут быть замещены другими транзитными модулями.

**Принцип генерируемости ОС**

Основное положение этого принципа определяет такой способ исходного представления центральной системной управляющей программы ОС (ее ядра и основных компонентов, которые должны постоянно находиться в оперативной памяти), который позволял бы настраивать эту системную супервизорную часть, исходя из конкретной конфигурации конкретного вычислительного комплекса и круга решаемых задач.

 Эта процедура проводится редко, перед достаточно протяженным периодом эксплуатации ОС.

Процесс генерации осуществляется с помощью специальной программы-генератора и соответствующего входного языка для этой программы, позволяющего описывать программные возможности системы и конфигурацию машины. В результате генерации получается полная версия ОС. Сгенерированная версия ОС представляет собой совокупность системных наборов модулей и данных.

Упомянутый раньше принцип модульности положительно проявляется при генерации ОС. Он существенно упрощает настройку ОС на требуемую конфигурацию вычислительной системы.

В наши дни при использовании персональных компьютеров с принципом генерируемости ОС можно столкнуться разве что только при работе с Linux. В этой UNIX-системе имеется возможность не только использовать какое-либо готовое ядро ОС, но и самому сгенерировать (скомпилировать) такое ядро, которое будет оптимальным для данного конкретного персонального компьютера и решаемых на нем задач. Кроме генерации ядра в Linux имеется возможность указать и набор подгружаемых драйверов и служб, то есть часть функций может реализовываться модулями, непосредственно входящими в ядро системы, а часть — модулями, имеющими статус подгружаемых, транзитных.

В остальных современных распространенных ОС для персональных компьютеров конфигурирование ОС под соответствующий состав оборудования осуществляется на этапе инсталляции, а потом состав драйверов и изменение некоторых параметров ОС может быть осуществлено посредством редактирования конфигурационного файла.

**Принцип функциональной избыточности**

Этот принцип учитывает возможность проведения одной и той же работы различными средствами.

В состав ОС может входить несколько типов мониторов (модулей супервизора, управляющих тем или другим видом ресурса), различные средства организации коммуникаций между вычислительными процессами.

Наличие нескольких типов мониторов, нескольких систем управления файлами позволяет пользователям быстро и наиболее адекватно адаптировать ОС к определенной конфигурации вычислительной системы, обеспечить максимально эффективную загрузку технических средств при решении конкретного класса задач, получить максимальную производительность при решении заданного класса задач.

**Принцип виртуализации**

Построение виртуальных ресурсов, их распределение и использование теперь используется практически в любой ОС. Этот принцип позволяет представить структуру системы в виде определенного набора планировщиков процессов и распределителей ресурсов (мониторов) и использовать единую централизованную схему распределения ресурсов.

Наиболее естественным и законченным проявлением концепции виртуальности является понятие *виртуальной* машины. По сути, любая операционная система, являясь средством распределения ресурсов и организуя по определенным правилам управление процессами, скрывает от пользователя и его приложений реальные аппаратные и иные ресурсы, заменяя их некоторой абстракцией. В результате пользователи видят и используют виртуальную машину как некое устройство, способное воспринимать их программы, написанные на определенном языке программирования, выполнять их и выдавать результаты. При таком языковом представлении пользователя совершенно не интересует реальная конфигурация вычислительной системы, способы эффективного использования ее компонентов и подсистем. Он мыслит и работает с машиной в терминах используемого им языка и тех ресурсов, которые ему предоставляются в рамках виртуальной машины.

Чаще виртуальная машина, предоставляемая пользователю, воспроизводит архитектуру реальной машины, но архитектурные элементы в таком представлении выступают с новыми или улучшенными характеристиками, часто упрощающими работу с системой. Характеристики могут быть произвольными, но чаще всего пользователи желают иметь собственную «идеальную» по архитектурным характеристикам машину в следующем составе:

-   единообразная по логике работы память (виртуальная) практически неограниченного объема. Среднее время доступа соизмеримо со значением этого параметра оперативной памяти. Организация работы с информацией в такой памяти производится в терминах обработки данных — в терминах работы с сегментами данных на уровне выбранного пользователем языка программирования;

-  произвольное количество процессоров (виртуальных), способных работать параллельно и взаимодействовать во время работы. Способы управления процессорами, в том числе синхронизация и информационные взаимодействия, реализованы и доступны пользователям на уровне используемого языка в терминах управления процессами;

-  произвольное количество внешних устройств (виртуальных), способных работать с памятью виртуальной машины параллельно или последовательно, асинхронно или синхронно по отношению к работе того или иного виртуального процессора, которые инициируют работу этих устройств. Информация, передаваемая или хранимая на виртуальных устройствах, не ограничена допустимыми размерами. Доступ к такой информации осуществляется на основе либо последовательного, либо прямого способа доступа в терминах соответствующей системы управления файлами. Предусмотрено расширение информационных структур данных, хранимых на виртуальных устройствах.

Степень приближения к «идеальной» виртуальной машине может быть большей или меньшей в каждом конкретном случае. Чем больше виртуальная машина, реализуемая средствами ОС на базе конкретной аппаратуры, приближена к «идеальной» по характеристикам машине и, следовательно, чем больше ее архитектурно-логические характеристики отличны от реально существующих, тем больше степень виртуальности у полученной пользователем машины.

Одним из аспектов виртуализации является организация возможности выполнения в данной ОС приложений, которые разрабатывались для других ОС. Другими словами, речь идет об организации нескольких операционных сред.

Реализация этого принципа позволяет такой ОС иметь очень сильное преимущество перед аналогичными ОС, не имеющими такой возможности.

Примером реализации принципа виртуализации может служить *VDM-машина* (virtualDOSmachine) — защищенная подсистема, предоставляющая полную среду MS-DOS и консоль для выполнения MS-DOS приложений. Одновременно может выполняться практически произвольное число VDM-сессий. Такие VDM-машины имеются и в системах MicrosoftWindows, и в OS/2.

**Принцип независимости программ от внешних устройств**

Этот принцип реализуется сейчас в подавляющем большинстве ОС общего применения. Мы уже говорили о нем, рассматривая принципы организации ввода/ вывода. Пожалуй, впервые наиболее последовательно данный принцип был реализован в ОС UNIX. Реализован он и в большинстве современных ОС для ПК. Напомним, этот принцип заключается в том, что связь программ с конкретными устройствами производится не на уровне трансляции программы, а в период планирования ее исполнения. В результате перекомпиляция при работе программы с новым устройством, на котором располагаются данные, не требуется.

Принцип позволяет одинаково осуществлять операции управления внешними устройствами независимо от их конкретных физических характеристик. Например, программе, содержащей операции обработки последовательного набора данных, безразлично, на каком носителе эти данные будут располагаться. Смена носителя и данных, размещаемых на них (при неизменности структурных характеристик данных), не принесет каких-либо изменений в программу, если в системе реализован принцип независимости.

**Принцип совместимости**

Одним из аспектов совместимости является способность ОС выполнять программы, написанные для других ОС или для более ранних версий данной операционной системы, а также для другой аппаратной платформы.

Необходимо разделять вопросы двоичной совместимости и совместимости на уровне исходных текстов приложений. Двоичная совместимость достигается в том случае, когда можно взять исполняемую программу и запустить ее на выполнение на другой ОС. Для этого необходимы: совместимость на уровне команд процессора, совместимость на уровне системных вызовов и даже на уровне библиотечных вызовов, если они являются динамически связываемыми.

Совместимость на уровне исходных текстов требует наличия соответствующего транслятора в составе системного программного обеспечения, а также совместимости на уровне библиотек и системных вызовов. При этом необходима перекомпиляция имеющихся исходных текстов в новый выполняемый модуль.

Гораздо сложнее достичь двоичной совместимости между процессорами, основанными на разных архитектурах. Для того чтобы один компьютер выполнял программы другого (например, программу для ПК типа IBMPC желательно выполнить на ПК типа Macintosh фирмы Apple), этот компьютер должен работать с машинными командами, которые ему изначально непонятны. В таком случае процессор типа 680x0 (или PowerPC) на Мае должен исполнять двоичный код, предназначенный для процессора 180x86. Процессор 80x86 имеет свои собственные дешифратор команд, регистры и внутреннюю архитектуру. Процессор 680x0 не понимает двоичный код 80x86, поэтому он должен выбрать каждую команду, декодировать ее, чтобы определить, для чего она предназначена, а затем выполнить эквивалентную подпрограмму, написанную для 680x0. Так как к тому же у 680x0 нет в точности таких же регистров, флагов и внутреннего арифметико-логического устройства, как в 80x86, он должен имитировать все эти элементы с использованием своих регистров или памяти. И он должен тщательно воспроизводить результаты каждой команды, что требует специально написанных подпрограмм для 680x0, гарантирующих, что состояние эмулируемых регистров и флагов после выполнения каждой команды будет в точности таким же, как и на реальном 80x86. Выходом в таких случаях является использование так называемых прикладных сред или эмуляторов. Учитывая, что основную часть программы, как правило, составляют вызовы библиотечных функций, прикладная среда имитирует библиотечные функции целиком, используя заранее написанную библиотеку функций аналогичного назначения, а остальные команды эмулирует каждую по отдельности.

Одним из средств обеспечения совместимости программных и пользовательских интерфейсов является соответствие стандартам POSIX. Использование стандарта POSIX позволяет создавать программы в стиле UNIX, которые впоследствии могут легко переноситься из одной системы в другую.

**Принцип открытой и наращиваемой ОС**

Открытая ОС доступна для анализа как пользователям, так и системным специалистам, обслуживающим вычислительную систему. Наращиваемая (модифицируемая, развиваемая) ОС позволяет не только использовать возможности генерации, но и вводить в ее состав новые модули, совершенствовать существующие и т. д. Другими словами, необходимо, чтобы можно было легко внести дополнения и изменения, если это потребуется, и не нарушить целостность системы. Прекрасные возможности для расширения предоставляет подход к структурированию ОС по типу клиент—сервер с использованием микроядерной технологии. В соответствии с этим подходом ОС строится как совокупность привилегированной управляющей программы и набора непривилегированных услуг — «серверов». Основная часть ОС остается неизменной и в то же время могут быть добавлены новые серверы или улучшены старые.

Этот принцип иногда трактуют как расширяемость системы.

К открытым ОС, прежде всего, следует отнести UNIX-системы и, естественно, ОС Linux.

**Принцип мобильности (переносимости)**

Операционная система относительно легко должна переноситься с процессора одного типа на процессор другого типа и с аппаратной платформы (которая включает наряду с типом процессора и способ организации всей аппаратуры компьютера, иначе говоря, архитектуру вычислительной системы) одного типа на аппаратную платформу другого типа. Заметим, что принцип переносимости очень близок принципу совместимости, хотя это и не одно и то же.

Написание переносимой ОС аналогично написанию любого переносимого кода — нужно следовать некоторым правилам. Во-первых, большая часть ОС должна быть написана на языке, который имеется на всех системах, на которые планируется в дальнейшем ее переносить. Это, прежде всего, означает, что ОС должна быть написана на языке высокого уровня, предпочтительно стандартизованном, например на языке С. Программа, написанная на ассемблере, не является в общем случае переносимой. Во-вторых, важно минимизировать или, если возможно, исключить те части кода, которые непосредственно взаимодействуют с аппаратными средствами. Зависимость от аппаратуры может иметь много форм. Некоторые очевидные формы зависимости включают прямое манипулирование регистрами и другими аппаратными средствами. Наконец, если аппаратно-зависимый код не может быть полностью исключен, то он должен быть изолирован в нескольких хорошо локализуемых модулях. Аппаратно-зависимый код не должен быть распределен по всей системе. Например, можно спрятать аппаратно-зави-симую структуру в программно задаваемые данные абстрактного типа. Другие модули системы будут работать с этими данными, а не с аппаратурой, используя набор некоторых функций. Когда ОС переносится, то изменяются только эти данные и функции, которые ими манипулируют.

Введение стандартов POSIX преследовало цель обеспечить переносимость создаваемого программного обеспечения.

**Принцип обеспечения безопасности вычислений**

Обеспечение безопасности при выполнении вычислений является желательным свойством для любой многопользовательской системы. Правила безопасности определяют такие свойства, как защита ресурсов одного пользователя от других и установление квот по ресурсам для предотвращения захвата одним пользователем всех системных ресурсов (таких, как память).

Обеспечение защиты информации от несанкционированного доступа является обязательной функцией сетевых операционных систем.

**2. Составные части ОС, их назначение, основные функции.**

В состав операционной системы входят: Ядро операционной системы, обеспечивающей распределение и управление ресурсами вычислительной системы; базовый набор прикладного программного обеспечения, системные библиотеки и программы обслуживания. Ядро системы - это набор функций, структур данных и отдельных программных модулей, загружаемых в память компьютера при загрузке операционной системы и обеспечивают три типа системных сервисов: управления вводом-выводом информации (подсистема ввода-вывода ядра ОС); управления оперативной памятью (подсистема управления оперативной памятью ядра ОС); управление процессами (подсистема управления процессами ядра ОС). Каждая из этих подсистем представлена соответствующими функциями ядра системы. Многозадачные операционные системы также включают еще одну обязательную составляющую - механизм поддержки многозадачности. Эта составляющая не предоставляется в качестве системного сервиса и поэтому не может быть отнесена к одной из подсистем.

Существует три основных механизма обеспечения многозадачности (планирование задач): путем предоставления процессора отдельной задачи на квант времени, который определяется самой задачей (кооперативная Многозадачность; последнее время практически не используется или область использования значительно ограничена внутри процессов); путем предоставления процессора отдельной задачи на квант времени, который определяется оборудованием вычислительной системы - интервальным таймером; выделение под отдельную задачу отдельного процессора в многопроцессорных системах. В первых двух случаях на каждом из процессоров в отдельно взятый момент времени обсчитывается только одна задача, но за счет достаточно малого кванта времени (в пределах миллисекунд), что поочередно предоставляется каждой из задач, возникает иллюзия одновременного выполнения в системе многих задач. В современных системах, как правило комбинируется методы 2 и 3.

**3. Выполнение заданий под управлением операционной системы.**

**Подсистема управления процессами** является одной из наиболее важных в операционной системе. В ее функции входит создание, уничтожение и обеспечение взаимодействия между процессами, а также распределение между ними процессорного времени. Определим понятия процесса и потока. В юните “Основы операционных систем” уже было дано простейшее определение процесса. Однако начиная с 1960-х годов в вычислительных системах используется **мультипрограммирование**, или **мультизадачность** (multitasking) – способ организации вычислительного процесса, при котором на одном процессоре выполняются сразу несколько программ. В совместном использовании этих программ находятся и другие ресурсы системы: оперативная память, дисковое пространство, данные.

Основными критериями для оценки эффективности вычислительной системы являются:

- **пропускная способность** – количество задач, выполняемых вычислительной системой в единицу времени;

- *удобство работы пользователей*, заключающееся, например, в том, что они могут одновременно работать в интерактивном режиме с несколькими приложениями на одном компьютере;

- *реактивность системы*.

Для поддержки мультипрограммирования в операционной системе (ОС) определяются внутренние единицы работы, между которыми и разделяются ресурсы. В настоящее время в большинстве операционных систем определены два типа единиц работы. Под *процессом* (*задачей*) при этом понимается более крупная единица работы, требующая для своего выполнения несколько единиц более мелких работ, называемых “поток”, или “нить”. Объясним принципиальные различия, существующие между этими понятиями.

В самом простейшем случае процесс может состоять только из одного потока и тогда понятие “процесс” полностью включает в себя понятие “поток” и остается только один вид единицы работы и потребления всех ресурсов вычислительной системы – процесс. Мультипрограммирование осуществляется в таких ОС на уровне процессов.

Каждый процесс должен иметь возможность влиять только на принадлежащие ему коды и данные, поэтому в задачи ОС при управлении процессами входит изоляция одного процесса от другого. Реализация этих задач подразумевает выделение операционной системой каждому процессу **виртуального адресного пространства** – совокупность адресов, которыми может манипулировать программный модуль процесса. Отображение этого виртуального пространства на реальную отведенную данному процессу физическую память также осуществляется операционной системой. В тех случаях, когда нескольким процессам необходимо взаимодействовать, например с целью обмена данными, они обращаются к операционной системе, которая, выполняя функции посредника, предоставляет им средства межпроцессной связи.

Потоки возникли в операционных системах как средство распараллеливания вычислений. Конечно, задача распараллеливания вычислений в рамках одного приложения может быть решена и традиционными способами.

Во-первых, прикладной программист может взять на себя сложную задачу органи­зации параллелизма, выделив в приложении некоторую подпрограмму-диспет­чер, которая периодически передает управление той или иной ветви вычислений. При этом программа получается логически весьма запутанной, с многочислен­ными передачами управления, что существенно затрудняет ее отладку и моди­фикацию.

Во-вторых, решением является создание для одного приложения нескольких про­цессов для каждой из параллельных работ. Однако использование для создания процессов стандартных средств ОС не позволяет учесть тот факт, что эти про­цессы решают единую задачу, а значит, имеют много общего между собой – они могут работать с одними и теми же данными, использовать один и тот же кодо­вый сегмент, наделяться одними и теми же правами доступа к ресурсам вычис­лительной системы.

Таким образом, в операционной системе наряду с про­цессами нужен другой механизм распараллеливания вычислений, который учи­тывал бы тесные связи между отдельными ветвями вычислений одного и того же приложения. Для этих целей современные ОС предлагают **механизм мпогопоточной обработки** (multithreading). При этом вводится новая единица работы – поток выполнения, а понятие “процесс” в значительной степени меняет смысл. Понятию “поток” соответствует последовательный переход процессора от одной команды программы к другой. ОС распределяет процессорное время между пото­ками. Процессу ОС назначает адресное пространство и набор ресурсов, которые совместно используются всеми его потоками.

Создание потоков, а не процессов, требует от ОС меньших накладных расходов. В отличие от процессов, которые принадлежат разным, вообще говоря, конкури­рующим приложениям, все потоки одного процесса всегда принадлежат одному приложению, поэтому ОС изолирует потоки в гораздо меньшей степени, нежели процессы в традиционной мультипрограммной системе. Все потоки одного про­цесса используют общие файлы, таймеры, устройства, одну и ту же область опе­ративной памяти, одно и то же адресное пространство. Это означает, что они раз­деляют одни и те же глобальные переменные. Поскольку каждый поток может иметь доступ к любому виртуальному адресу процесса, один поток может ис­пользовать стек другого потока. Между потоками одного процесса нет полной защиты, потому что, во-первых, это невозможно, а во-вторых, не нужно. Чтобы организовать взаимодействие и обмен данными, потокам вовсе не требуется об­ращаться к ОС, им достаточно использовать общую память – один поток запи­сывает данные, а другой читает их. С другой стороны, потоки разных процессов по-прежнему хорошо защищены друг от друга.

Итак, мультипрограммирование более эффективно на уровне потоков, а не процес­сов. Каждый поток имеет собственный счетчик команд и стек. Задача, оформлен­ная в виде нескольких потоков в рамках одного процесса, может быть выполнена быстрее за счет псевдопараллельного (или параллельного в мультипроцессорной системе) выполнения ее отдельных частей. Например, если электронная таблица была разработана с учетом возможностей многопоточной обработки, то пользо­ватель может запросить пересчет своего рабочего листа и одновременно продол­жать заполнять таблицу. Особенно эффективно можно использовать многопоточность для выполнения распределенных приложений, например многопоточный сервер может параллельно выполнять запросы сразу нескольких клиентов.

Использование потоков связано не только со стремлением повысить производи­тельность системы за счет параллельных вычислений, но и с целью создания более читабельных, логичных программ. Введение нескольких потоков выполне­ния упрощает программирование. Например, в задачах типа “писатель-читатель” один поток выполняет запись в буфер, а другой считывает записи из него. По­скольку они разделяют общий буфер, не стоит их делать отдельными процесса­ми. Другой пример использования потоков – управление сигналами, такими, как прерывание с клавиатуры (del или break). Вместо обработки сигнала прерывания один поток назначается для постоянного ожидания поступления сигналов. Та­ким образом, использование потоков может сократить необходимость в прерыва­ниях пользовательского уровня. В этих примерах не столь важно параллельное выполнение, сколь важна ясность программы.

**Управление процессором в однопрограммном режиме**

Рассмотрим задачу управления процессором для однопрограммных операционных систем. В такой ОС существует два процесса: *системный процесс* – процесс выполнения программ ОС и *пользовательский процесс* – процесс выполнения программ пользователя.

Системный процесс существует постоянно: с момента загрузки операционной системы и до конца ее работы. Он может быть заблокирован (например, ждать команды пользователя на выполнение программы).

Переключение “пользовательский процесс – системный процесс” в однопрограммной ОС связано со следующими событиями в вычислительной системе:

- завершение пользовательского процесса;

- обращение пользовательского процесса к системному процессу для выполнения каких-либо функций ОС.

Переключение “системный процесс – пользовательский процесс” связано с:

- созданием пользовательского процесса;

- завершением выполнения функций ОС, используемых пользовательским процессом.

С точки зрения процессора разделение процессов на системный и пользовательский является абстрактным, поскольку процессору, как техническому устройству, безразлично, команды какой программы он выполняет: пользовательской или системной.

**Управление процессором в режиме мультипрограммирования**

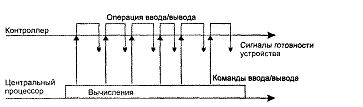
**Понятие мультипрограммирования**

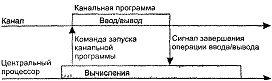
Операционные системы, поддерживающие мульти-програм-мирование, в зависимости от выбранного критерия эффективности вычислительной системы можно разделить на несколько основных типов: системы пакетной обработки, системы разделения времени и системы реального времени. Некоторые операционные системы могут поддерживать одновремен­но несколько режимов, например часть задач может выполняться в режиме па­кетной обработки, а часть – в режиме реального времени или в режиме разделе­ния времени.

**Мультипрограммирование в системах пакетной обработки**

Рациональный режим функционирования в системах пакетной обработки достигается благодаря использованию следующей схемы функционирования: в начале работы формируется пакет заданий, каждое задание содержит требование к системным ресурсам; из этого пакета за­даний формируется **мультипрограммная смесь**, т.е. множество одновремен­но выполняемых задач. Причем смесь строится таким образом, что одновременно должны выполняться задачи, предъявляющие разные требования к ресурсам с целью обеспечения сба­лансированной загрузки всех устройств вычислительной машины. Например, в мультипрограммной смеси желательно одновременное присутствие вычислитель­ных задач и задач с интенсивным вводом-выводом. Таким образом, выбор нового задания из пакета заданий зависит от внутренней ситуации, складывающейся в системе, т.е. выбирается “выгодное” с точки зрения критерия эффективности системы задание. Следствием такой схемы организации мультипрограммирования в вычисли­тельных системах, работающих под управлением пакетных ОС, является отсутствие гарантии выполнения какого-либо задания в течение определенного пе­риода времени.

Рассмотрим более детально совмещение во времени операций ввода-вывода и вычислений.

Достигнуть описанного выше совмещения можно несколькими способами. Один из них характе­рен для компьютеров, имеющих специализированный процессор ввода-вывода. В компьютерах класса мэйнфреймов такие процессоры называют **каналами**. Как правило, канал имеет систему команд, отличающуюся от системы команд цен­трального процессора. Эти команды специально предназначены для управления внешними устройствами. Ка­нальные программы могут храниться в той же оперативной памяти, что и про­граммы центрального процессора. Однако в системе команд центрального процессора предусматривается специальная инструкция, с помощью которой каналу переда­ются параметры и указания на то, какую программу ввода-вывода он должен вы­полнить. Начиная с этого момента центральный процессор и канал могут работать параллельно (см. рисунки ниже).



 Существует и другой способ совмещения вычислений с операциями ввода-вывода. Он реализован в компьютерах, управление внешними устройствами в которых осуществляется так называемыми контроллерами. Каждое внешнее устройство (или группа внеш­них устройств одного типа) имеет свой собственный контроллер, который авто­номно отрабатывает команды, поступающие от центрального процессора. При этом контроллер и центральный процессор работают асинхронно. Многие внешние устройства включают электромеханические узлы, вследствие чего контроллер выпол­няет свои команды управления устройствами существенно медленнее, чем цен­тральный процессор свои. Именно это обстоятельство используется для организации параллельного выполнения вычислений и операций ввода-вывода: в промежутке между передачей команд контроллеру центральный процессор может выполнять вычисления (рис.1,*б*). Контроллер может сообщить центральному процессору о том, что он готов принять следующую команду, с помощью специального сигнала, либо центральный процессор узнает об этом, периодически опрашивая состояние кон­троллера.

Таким образом, для достижения максимальной эффективности решения задач операционной системой необходимо обеспечить наиболее возможное совмещение вычислений и ввода-вывода. Рассмотрим случай, когда процессор выполня­ет только одну задачу. В этой ситуации степень ускорения зависит от природы данной задачи и от того, насколько тщательно был выявлен возможный паралле­лизм при ее программировании. В задачах, в которых преобладают либо вычисле­ния, либо ввод-вывод, ускорение почти отсутствует. Параллелизм в рамках од­ной задачи невозможен также, когда для продолжения вычислений необходимо полное завершение операции ввода-вывода, например, когда дальнейшие вычис­ления зависят от вводимых данных. В таких случаях неизбежны простои цен­трального процессора или канала.

**4. Взаимодействие пользователей и прикладных программ с ОС.**

**Мультипрограммирование в системах разделения времени**

В том случае, когда критерием эффективности работы системы является удобство работы пользователя, то применяется другой способ мультипрограммирования – разделения времени. В системах разделения времени пользователям (или одному пользователю) предоставляется возможность интерактивной работы сразу с несколькими приложениями. Для этого каждое приложение должно регулярно получать возможность “общения” с пользователем. Понятно, что в пакетных системах возможности диалога пользо­вателя с приложением весьма ограничены.

В системах разделения времени эта проблема решается за счет того, что ОС принудительно периодически приостанавливает приложения, не дожидаясь, ког­да они добровольно освободят процессор. Всем приложениям попеременно вы­деляется квант процессорного времени, вследствие чего пользователи, запустив­шие программы на выполнение, получают возможность поддерживать с ними диалог.

В системах разделения времени исправлен основной недостаток пакетной обработки – изолированность пользователя от процесса выполнения его задач. Каждому пользователю в этом случае предоставляется терми­нал, с которого он может вести диалог со своей программой. Время ответа в таких системах оказывается вполне приемлемым для пользователей, поскольку каждой задаче выделяется только квант процессорного вре­мени, а следовательно, ни одна задача не занимает процессор надолго. Конечно, время реакции напрямую зависит от выбранного в системе кванта времени, выделяемого каждому процессу. Если квант выбран достаточно небольшим, то у всех пользователей, одновременно работающих на одной и той же машине, складывается впечат­ление, что каждый из них единолично использует машину.

Одной из основных причин, по которой системы разделения времени обладают меньшей пропускной способ­ностью, чем системы пакетной обработки, является то, что на выполнение принимается каждая запущенная пользователем задача, а не та, которая “выгодна” ОС. Кроме того, производительность системы снижается из-за возросших накладных расходов вычислительной мощности на более частое переключение процессора с задачи на задачу. Это вполне соответствует тому, что критерием эффективно­сти систем разделения времени является не максимальная пропускная способ­ность, а удобство и эффективность работы пользователя. Вместе с тем мульти­программное выполнение интерактивных приложений повышает и пропускную способность компьютера (пусть и не в такой степени, как пакетные системы). Аппаратура загружается лучше, поскольку в то время, пока одно приложение ждет сообщения пользователя, другие приложения могут обрабатываться процес­сором.

**Мультипрограммирование в системах реального времени**

Еще одна разновидность мультипрограммирования используется в системах ре­ального времени, предназначенных для управления с помощью компьютера различными техническими объектами (например, станком, спутником, научной эксперименталь­ной установкой и т. д.) или технологическими процессами. Во всех этих случаях существует пре­дельно допустимое время, в течение которого должна бытьвыполнена та или иная управляющая объектом программа. В противном случае может произойти авария. Таким образом, критерием эффективности здесь является время реакции системы. Требо­вания ко времени реакции зависят от специфики управляемого процесса. Кон­троллер робота может требовать от встроенного компьютера ответ в течение менее 1 мс, в то время как при моделировании полета может быть приемлем от­вет в 40 мс.

В системах реального времени мультипрограммная смесь отличается от рассмотренных ранее и представляет собой фик­сированный набор заранее разработанных программ, а выбор программы на выпол­нение осуществляется исходя из текущего состояния объекта или в соответствии с расписанием плановых работ.

Способность аппаратуры компьютера и ОС к быстрому ответу зависит в основном от скорости переключения с одной задачи на другую. Время переключения между задачами в системах реального времени часто определяет требования к классу процессора даже при небольшой его загрузке.

В системах реального времени не стремятся максимально загружать все устройства, наоборот, при проектировании программного управляющего комплекса обычно за­кладывается некоторый “запас” вычислительной мощности на случай пиковой на­грузки. Во мно­гих ситуациях для систем управления неприменимы статистические гипотезы о малой вероятности одновременного возникновения большого количества независимых событий. Например, в системе управления атомной электростанцией в случае возникновения крупной аварии атомного реактора многие аварийные датчики сработают одновременно и создадут коррелирован­ную нагрузку. Если система реального времени не спроектирована для поддерж­ки пиковой нагрузки, то может случиться так, что система не справится с рабо­той именно тогда, когда она нужна в наибольшей степени. большую память между всеми процес­сорами. **Масштабируемость**, или возможность наращивания числа процессоров, в симмет­ричных системах ограничена вследствие того, что все они пользуются одной и той же оперативной памятью и, следовательно, должны располагаться в одном корпусе. Такая конструкция, называемая масштабируемой по вертикали, на прак­тике ограничивает число процессоров до четырех или восьми. В симметричных архитектурах все процессы пользуются одной и той же схемой отображения памяти. Они могут очень быстро обмениваться данными, так что обеспечивается достаточно высокая производительность для тех приложений (например, при работе с базами данных), в которых несколько задач должны ак­тивно взаимодействовать между собой.

В асимметричной архитектуре разные процессоры могут отличаться как своими характеристиками (производительностью, надежностью, системой команд и т.д., вплоть до модели микропроцессора), так и функциональной ролью, которая по­ручается им в системе. Например, одни процессоры могут предназначаться для работы в качестве основных вычислителей, другие – для управления подсисте­мой ввода-вывода, третьи – еще для каких-то особых целей. Такая функциональная неоднородность влечет за со­бой структурные отличия во фрагментах системы, содержащих разные процес­соры системы. Например, они могут отличаться схемами подключения про­цессоров к системной шине, набором периферийных устройств и способами взаимодействия процессоров с устройствами. Масштабирование в асимметричной архитектуре реализуется иначе, чем в сим­метричной. Так как требование единого корпуса отсутствует, система может со­стоять из нескольких устройств, каждое из которых содержит один или несколь­ко процессоров. Это масштабирование по горизонтали. Каждое такое устройство называется **кластером**, а вся мультипроцессорная система – кластерной.

Другим аспектом мультипроцессорных систем, который может характеризовать­ся симметрией или ее отсутствием, является способ организации вычислительного процесса. Последний, как известно, определяется и реализуется операцион­ной системой.

Асимметричное мультипроцессирование – наиболее простой способ организации вычислительного процесса в системах с несколькими процессора­ми. Этот способ часто называют также “ведущий-ведомый”.

Функционирование системы по принципу “ведущий-ведомый” предполагает вы­деление одного из процессоров в качестве “ведущего”, на котором работает опе­рационная система и который управляет всеми остальными “ведомыми” процес­сорами, т.е. ведущий процессор берет на себя функции распределения задач и ресурсов, а ведомые процессоры работают только как обрабатывающие устрой­ства и никаких действий по организации работы вычислительной системы не выполняют.

Так как операционная система работает только на одном процессоре и функции управления полностью централизованы, то такая операционная система оказы­вается не намного сложнее ОС однопроцессорной системы.

Асимметричная организация вычислительного процесса может быть реализова­на как для симметричной мультипроцессорной архитектуры, в которой все про­цессоры аппаратно неразличимы, так и для несимметричной, для которой харак­терна неоднородность процессоров, их специализация на аппаратном уровне.

В системах с асимметричной архитектурой на роль ведущего процессора может быть назначен наиболее надежный и производительный процессор. Если в набо­ре процессоров имеется специализированный процессор, ориентированный, на­пример, на матричные вычисления, то при планировании процессов операци­онная система, реализующая асимметричное мультипроцессирование, должна учитывать специфику этого процессора. Такая специализация снижает надеж­ность системы в целом, так как процессоры не являются взаимозаменяемыми.

Симметричное мультипроцессирование как способ организации вычислительно­го процесса может быть реализовано в системах только с симметричной мульти­процессорной архитектурой. Напомним, что в таких системах процессоры рабо­тают с общими устройствами и разделяемой основной памятью.

Симметричное мультипроцессирование реализуется общей для всех процес­соров операционной системой. При симметричной организации все процессоры равноправно участвуют и в управлении вычислительным процессом, и в выпол­нении прикладных задач. Например, сигнал от принтера, который распечатывает данные прикладного процесса, выполняемого на некотором про­цессоре, может быть обработан совсем другим процессором. Разные процессоры могут в какой-то момент одновременно обслуживать как разные, так и одинако­вые модули общей операционной системы. Для этого программы операционной системы должны обладать свойством повторной входимости (реентерабельнос­тью). Операционная система при этом полностью децентрализована. Модули ОС выполняются на любом доступном процессоре. Как только процессор завершает выполнение очередной задачи, он передает управление планировщику задач, который выби­рает из общей для всех процессоров системной очереди задачу, которая будет выполняться на данном процессоре следующей. Все ресурсы выделяются для каждой выполняемой задачи по мере возникновения в них потребностей и никак не закрепляются за процессором. При таком подходе все процессоры работают с одной и той же динамически выравниваемой нагрузкой. В решении одной за­дачи могут участвовать сразу несколько процессоров, если она допускает такоераспараллеливание, например путем представления в виде нескольких потоков.

В случае отказа одного из процессоров симметричные системы, как правило, сравнительно просто реконфигурируются, что является их большим преимуще­ством перед плохо реконфигурируемыми асимметричными системами.

Симметричная и асимметричная организация вычислительного процесса в муль­типроцессорной системе не связана напрямую с симметричной или асимметрич­ной архитектурой, она определяется типом операционной системы. Так, в сим­метричных архитектурах вычислительный процесс может быть организован как симметричным образом, так и асимметричным. Однако асимметричная архитек­тура непременно влечет за собой и асимметричный способ организации вычис­лений.

**Контрольные вопросы:**

1. Перечислите принципы построения ОС.

2. Укажите составные части ОС и их назначение.

3. Перечислите основные функции ОС.

4. Укажите принципы выполнения заданий под управлением операционной системы.

5. Опишите процесс взаимодействия пользователей и прикладных программ с ОС.

**Литература:**

1. А.П. Побегайло, Системное программирование в Windows, БХВ-Петербург, 2006, 1056 с.

2. Ахо, Лам, Сети, Ульман - Компиляторы. Принципы, технологии, инструменты, Изд.: Вильямс, 2008, 1185 с.

3. Л.Бек, Введение в системное программирование, М., Мир, 2008г.

4. Р. И. Компаниец, Системное программирование : Основы построения трансляторов, Изд.: СПб., 2000.

5. Ф.Льюис, Д.Розенкранц, Р.Стирнз. Теоретические основы проектирования компиляторов. - М.: Мир, 20099, 655 с.

6. Э. Таненбау, Современные операционные системы, Изд.: Питер, 2011, 1120 с.

7. А. В. Гордеев, А. Ю. Молчанов, Системное программное обеспечение, учебник для ВУЗов, СПб.: Питер, 2010г.

**СЕМИНАР №9: «ПОНЯТИЕ И РЕАЛИЗАЦИЯ ПРОЦЕССОВ».**

**План семинара:**

1. Процессы. Понятие процесса.

2. Модель представления процесса в ОС.

3. Состояния процесса, контекст процесса, процедуры сохранения и переключения контекста – на примере Windows -подобных систем.

4. Операции над процессами – на примере Windows -подобных систем.

**Содержание по плану:**

**1. Процессы. Понятие процесса.**

В первой лекции, поясняя понятие "операционная система" и описывая способы построения операционных систем, мы часто применяли слова "программа" и "задание". Мы говорили: вычислительная система исполняет одну или несколько программ, операционная система планирует задания, программы могут обмениваться данными и т. д. Мы использовали эти термины в некотором общеупотребительном, житейском смысле, предполагая, что все читатели одинаково представляют себе, что подразумевается под ними в каждом конкретном случае. При этом одни и те же слова обозначали и объекты в статическом состоянии, необрабатывающиеся вычислительной системой (например, совокупность файлов на диске), и объекты в динамическом состоянии, находящиеся в процессе исполнения. Это было возможно, пока мы говорили об общих свойствах операционных систем, не вдаваясь в подробности их внутреннего устройства и поведения, или о работе вычислительных систем первого-второго поколений, которые не могли обрабатывать более одной программы или одного задания одновременно, по сути дела не имея операционных систем. Но теперь мы начинаем знакомиться с деталями функционирования современных компьютерных систем, и нам придется уточнить терминологию.

Рассмотрим следующий пример. Два студента запускают программу извлечения квадратного корня. Один хочет вычислить квадратный корень из 4, а второй – из 1. С точки зрения студентов, запущена одна и та же программа; с точки зрения компьютерной системы, ей приходится заниматься двумя различными вычислительными процессами, так как разные исходные данные приводят к разному набору вычислений. Следовательно, на уровне происходящего внутри вычислительной системы мы не можем использовать термин "программа" в пользовательском смысле слова.

Рассматривая системы пакетной обработки, мы ввели понятие "задание" как совокупность программы, набора команд языка управления заданиями, необходимых для ее выполнения, и входных данных. С точки зрения студентов, они, подставив разные исходные данные, сформировали два различных задания. Может быть, термин "задание" подойдет нам для описания внутреннего функционирования компьютерных систем? Чтобы выяснить это, давайте рассмотрим другой пример. Пусть оба студента пытаются извлечь корень квадратный из 1, то есть пусть они сформировали идентичные задания, но загрузили их в вычислительную систему со сдвигом по времени. В то время как одно из выполняемых заданий приступило к печати полученного значения и ждет окончания операции ввода-вывода, второе только начинает исполняться. Можно ли говорить об идентичности заданий внутри вычислительной системы в данный момент? Нет, так как состояние процесса их выполнения различно. Следовательно, и слово "задание" в пользовательском смысле не может применяться для описания происходящего в вычислительной системе.

Это происходит потому, что термины "программа" и "задание" предназначены для описания статических, неактивных объектов. Программа же в процессе исполнения является динамическим, активным объектом. По ходу ее работы компьютер обрабатывает различные команды и преобразует значения переменных. Для выполнения программы операционная система должна выделить определенное количество оперативной памяти, закрепить за ней определенные устройства ввода-вывода или файлы (откуда должны поступать входные данные и куда нужно доставить полученные результаты), то есть зарезервировать определенные ресурсы из общего числа ресурсов всей вычислительной системы. Их количество и конфигурация с течением времени могут изменяться. Для описания таких активных объектов внутри компьютерной системы вместо терминов "программа" и "задание" мы будем использовать новый термин – " процесс ".

В ряде учебных пособий и монографий для простоты предлагается рассматривать процесс как абстракцию, характеризующую программу во время выполнения. На наш взгляд, эта рекомендация не совсем корректна. Понятие процесса характеризует некоторую совокупность набора исполняющихся команд, ассоциированных с ним ресурсов (выделенная для исполнения память или адресное пространство, стеки, используемые файлы и устройства ввода-вывода и т. д.) и текущего момента его выполнения (значения регистров, программного счетчика, состояние стека и значения переменных), находящуюся под управлением операционной системы. Не существует взаимно-однозначного соответствия между процессами и программами, обрабатываемыми вычислительными системами. Как будет показано далее, в некоторых операционных системах для работы определенных программ может организовываться более одного процесса или один и тот же процесс может исполнять последовательно несколько различных программ. Более того, даже в случае обработки только одной программы в рамках одного процесса нельзя считать, что процесс представляет собой просто динамическое описание кода исполняемого файла, данных и выделенных для них ресурсов. Процесс находится под управлением операционной системы, поэтому в нем может выполняться часть кода ее ядра (не находящегося в исполняемом файле!), как в случаях, специально запланированных авторами программы (например, при использовании системных вызовов), так и в непредусмотренных ситуациях (например, при обработке внешних прерываний).

**2. Модель представления процесса в ОС.**

Модель процессов упрощает представление о внутреннем поведении системы. Некоторые процессы запускают программы, выполняющие команды, введенные с клавиатуры пользователем. Другие процессы являются частью системы и обслуживают такие задачи, как выполнение запросов файловой подсистемы, управление запуском диска или магнитного накопителя.

В случае дискового прерывания система останавливает текущий процесс и запускает дисковый процесс, который был заблокирован в ожидании этого прерывания. Вместо прерываний мы можем представлять себе дисковые процессы, процессы пользователя, терминала и т. п., блокирующиеся на время ожидания событий. Когда событие произошло (информация прочитана с диска или клавиатуры), блокировка снимается и процесс вправе быть запущен.

Рассмотренный подход описывается моделью, представленной на рис. 2.3. Нижний уровень операционной системы — это планировщик, выше него — все множество процессов. За обработку прерываний и детали, связанные с остановкой и запуском процессов, отвечает то, что мы назвали планировщиком, который является, по сути, совсем небольшой программой.

Вся остальная часть операционной системы упорядочена в виде набора процессов. Очень немногие существующие системы структурированы столь удобно.

Для реализации модели процессов операционная система поддерживает таблицу (массив структур), называемую таблицей процессов, с одним элементом для каждого процесса. (Некоторые авторы называют эти элементы блоками управления процессом).

Элемент таблицы содержит информацию о состоянии процесса, счетчике команд, указателе стека, распределении памяти, состоянии открытых файлов, об использовании и распределении ресурсов, а также всю остальную информацию, которую необходимо сохранять при переключении в состояние готовности или блокировки для последующего запуска, — как если бы процесс не останавливался.

**3. Состояния процесса, контекст процесса, процедуры сохранения и переключения контекста – на примере Windows -подобных систем.**

При использовании такой абстракции все, что выполняется в вычислительных системах (не только программы пользователей, но и, возможно, определенные части операционных систем), организовано как набор процессов. Понятно, что реально на однопроцессорной компьютерной системе в каждый момент времени может исполняться только один процесс. Для мультипрограммных вычислительных систем псевдопараллельная обработка нескольких процессов достигается с помощью переключения процессора с одного процесса на другой. Пока один процесс выполняется, остальные ждут своей очереди.

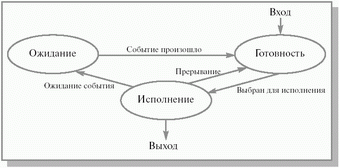
Как видим, каждый процесс может находиться как минимум в двух состояниях: процесс исполняется и процесс не исполняется. Диаграмма состояний процесса в такой модели изображена на [рис. 1](http://www.intuit.ru/department/os/osintro/2/osintro_2.html#image.2.1).



**Рис. 1.**  Простейшая диаграмма состояний процесса

Процесс, находящийся в состоянии процесс исполняется, через некоторое время может быть завершен операционной системой или приостановлен и снова переведен в состояние процесс не исполняется. Приостановка процесса происходит по двум причинам: для его дальнейшей работы потребовалось какое-либо событие (например, завершение операции ввода-вывода) или истек временной интервал, отведенный операционной системой для работы данного процесса. После этого операционная система по определенному алгоритму выбирает для исполнения один из процессов, находящихся в состоянии процесс не исполняется, и переводит его в состояние процесс исполняется. Новый процесс, появляющийся в системе, первоначально помещается в состояние процесс не исполняется.

Это очень грубая модель, она не учитывает, в частности, то, что процесс, выбранный для исполнения, может все еще ждать события, из-за которого он был приостановлен, и реально к выполнению не готов. Для того чтобы избежать такой ситуации, разобьем состояние процесс не исполняется на два новых состояния: готовность и ожидание (см. [рис. 2](http://www.intuit.ru/department/os/osintro/2/osintro_2.html#image.2.2)).



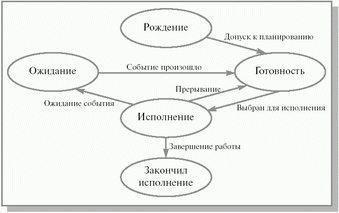
**Рис. 2.**  Более подробная диаграмма состояний процесса

Всякий новый процесс, появляющийся в системе, попадает в состояние готовность. Операционная система, пользуясь каким-либо алгоритмом планирования, выбирает один из готовых процессов и переводит его в состояние исполнение. В состоянии исполнение происходит непосредственное выполнение программного кода процесса. Выйти из этого состояния процесс может по трем причинам:

* операционная система прекращает его деятельность;
* он не может продолжать свою работу, пока не произойдет некоторое событие, и операционная система переводит его в состояние ожидание ;
* в результате возникновения прерывания в вычислительной системе (например, прерывания от таймера по истечении предусмотренного времени выполнения) его возвращают в состояние готовность.

Из состояния ожидание процесс попадает в состояние готовность после того, как ожидаемое событие произошло, и он снова может быть выбран для исполнения.

Наша новая модель хорошо описывает поведение процессов во время их существования, но она не акцентирует внимания на появлении процесса в системе и его исчезновении. Для полноты картины нам необходимо ввести еще два состояния процессов: рождение и закончил исполнение (см. [рис. 3](http://www.intuit.ru/department/os/osintro/2/osintro_2.html#image.2.3)).



**Рис. 3.**  Диаграмма состояний процесса, принятая в курсе

Теперь для появления в вычислительной системе процесс должен пройти через состояние рождение. При рождении процесс получает в свое распоряжение адресное пространство, в которое загружается программный код процесса ; ему выделяются стек и системные ресурсы; устанавливается начальное значение программного счетчика этого процесса и т. д. Родившийся процесс переводится в состояние готовность. При завершении своей деятельности процесс из состояния исполнение попадает в состояние закончил исполнение.

В конкретных операционных системах состояния процесса могут быть еще более детализированы, могут появиться некоторые новые варианты переходов из одного состояния в другое. Так, например, модель состояний процессов для операционной системы Windows NT содержит 7 различных состояний, а для операционной системы Unix – 9. Тем не менее, так или иначе все операционные системы подчиняются изложенной выше модели.

**4. Операции над процессами – на примере Windows -подобных систем.**

**Набор операций**

Процесс не может перейти из одного состояния в другое самостоятельно. Изменением состояния процессов занимается операционная система, совершая операции над ними. Количество таких операций в нашей модели пока совпадает с количеством стрелок на диаграмме состояний. Удобно объединить их в три пары:

* создание процесса – завершение процесса ;
* приостановка процесса (перевод из состояния исполнение в состояние готовность ) – запуск процесса (перевод из состояния готовность в состояние исполнение );
* блокирование процесса (перевод из состояния исполнение в состояние ожидание ) – разблокирование процесса (перевод из состояния ожидание в состояние готовность ).

В дальнейшем, когда мы будем говорить об алгоритмах планирования, в нашей модели появится еще одна операция, не имеющая парной: изменение приоритета процесса.

Операции создания и завершения процесса являются одноразовыми, так как применяются к процессу не более одного раза (некоторые системные процессы при работе вычислительной системы не завершаются никогда). Все остальные операции, связанные с изменением состояния процессов, будь то запуск или блокировка, как правило, являются многоразовыми. Рассмотрим подробнее, как операционная система выполняет операции над процессами.

**Process Control Block и контекст процесса**

Для того чтобы операционная система могла выполнять операции над процессами, каждый процесс представляется в ней некоторой структурой данных. Эта структура содержит информацию, специфическую для данного процесса:

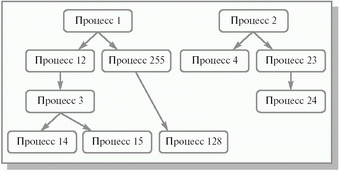
* состояние, в котором находится процесс ;
* программный счетчик процесса или, другими словами, адрес команды, которая должна быть выполнена для него следующей;
* содержимое регистров процессора;
* данные, необходимые для планирования использования процессора и управления памятью (приоритет процесса, размер и расположение адресного пространства и т. д.);
* учетные данные (идентификационный номер процесса, какой пользователь инициировал его работу, общее время использования процессора данным процессом и т. д.);
* сведения об устройствах ввода-вывода, связанных с процессом (например, какие устройства закреплены за процессом, таблицу открытых файлов).

Ее состав и строение зависят, конечно, от конкретной операционной системы. Во многих операционных системах информация, характеризующая процесс, хранится не в одной, а в нескольких связанных структурах данных. Эти структуры могут иметь различные наименования, содержать дополнительную информацию или, наоборот, лишь часть описанной информации. Для нас это не имеет значения. Для нас важно лишь то, что для любого процесса, находящегося в вычислительной системе, вся информация, необходимая для совершения операций над ним, доступна операционной системе. Для простоты изложения будем считать, что она хранится в одной структуре данных. Мы будем называть ее PCB (Process Control Block) или блоком управления процессом. Блок управления процессом является моделью процесса для операционной системы. Любая операция, производимая операционной системой над процессом, вызывает определенные изменения в PCB . В рамках принятой модели состояний процессов содержимое PCB между операциями остается постоянным.

Информацию, для хранения которой предназначен блок управления процессом, удобно для дальнейшего изложения разделить на две части. Содержимое всех регистров процессора (включая значение программного счетчика) будем называть регистровым контекстом процесса, а все остальное – системным контекстом процесса. Знания регистрового и системного контекстов процесса достаточно для того, чтобы управлять его работой в операционной системе, совершая над ним операции. Однако этого недостаточно для того, чтобы полностью охарактеризовать процесс. Операционную систему не интересует, какими именно вычислениями занимается процесс, т. е. какой код и какие данные находятся в его адресном пространстве. С точки зрения пользователя, наоборот, наибольший интерес представляет содержимое адресного пространства процесса, возможно, наряду с регистровым контекстом определяющее последовательность преобразования данных и полученные результаты. Код и данные, находящиеся в адресном пространстве процесса, будем называть его пользовательским контекстом. Совокупность регистрового, системного и пользовательского контекстов процесса для краткости принято называть просто контекстом процесса. В любой момент времени процесс полностью характеризуется своим контекстом.

**Одноразовые операции**

Сложный жизненный путь процесса в компьютере начинается с его рождения. Любая операционная система, поддерживающая концепцию процессов, должна обладать средствами для их создания. В очень простых системах (например, в системах, спроектированных для работы только одного конкретного приложения) все процессы могут быть порождены на этапе старта системы. Более сложные операционные системы создают процессы динамически, по мере необходимости. Инициатором рождения нового процесса после старта операционной системы может выступить либо процесс пользователя, совершивший специальный системный вызов, либо сама операционная система, то есть, в конечном итоге, тоже некоторый процесс. Процесс, инициировавший создание нового процесса, принято называть процессом-родителем (parent process), а вновь созданный процесс – процессом-ребенком (child process). Процессы-дети могут в свою очередь порождать новых детей и т. д., образуя, в общем случае, внутри системы набор генеалогических деревьев процессов – генеалогический лес. Пример генеалогического леса изображен на рисунке 1. Следует отметить, что все пользовательские процессы вместе с некоторыми процессами операционной системы принадлежат одному и тому же дереву леса. В ряде вычислительных систем лес вообще вырождается в одно такое дерево.



**Рисунок 1 –**  Упрощенный генеалогический лес процессов. Стрелочка означает отношение родитель–ребенок

При рождении процесса система заводит новый PCB с состоянием процесса рождение и начинает его заполнять. Новый процесс получает собственный уникальный идентификационный номер. Поскольку для хранения идентификационного номера процесса в операционной системе отводится ограниченное количество битов, для соблюдения уникальности номеров количество одновременно присутствующих в ней процессов должно быть ограничено. После завершения какого-либо процесса его освободившийся идентификационный номер может быть повторно использован для другого процесса.

Обычно для выполнения своих функций процесс-ребенок требует определенных ресурсов: памяти, файлов, устройств ввода-вывода и т. д. Существует два подхода к их выделению. Новый процесс может получить в свое распоряжение некоторую часть родительских ресурсов, возможно разделяя с процессом-родителем и другими процессами-детьми права на них, или может получить свои ресурсы непосредственно от операционной системы. Информация о выделенных ресурсах заносится в PCB.

После наделения процесса-ребенка ресурсами необходимо занести в его адресное пространство программный код, значения данных, установить программный счетчик. Здесь также возможны два решения. В первом случае процесс-ребенок становится дубликатом процесса-родителя по регистровому и пользовательскому контекстам, при этом должен существовать способ определения, кто для кого из процессов-двойников является родителем. Во втором случае процесс-ребенок загружается новой программой из какого-либо файла. Операционная система Unix разрешает порождение процесса только первым способом; для запуска новой программы необходимо сначала создать копию процесса-родителя, а затем процесс-ребенок должен заменить свой пользовательский контекст с помощью специального системного вызова. Операционная система VAX/VMS допускает только второе решение. В Windows NT возможны оба варианта (в различных API).

Порождение нового процесса как дубликата процесса-родителя приводит к возможности существования программ (т. е. исполняемых файлов), для работы которых организуется более одного процесса. Возможность замены пользовательского контекста процесса по ходу его работы (т. е. загрузки для исполнения новой программы) приводит к тому, что в рамках одного и того же процесса может последовательно выполняться несколько различных программ.

После того как процесс наделен содержанием, в PCB дописывается оставшаяся информация, и состояние нового процесса изменяется на готовность. Осталось сказать несколько слов о том, как ведут себя процессы-родители после рождения процессов-детей. Процесс-родитель может продолжать свое выполнение одновременно с выполнением процесса-ребенка, а может ожидать завершения работы некоторых или всех своих "детей".

Мы не будем подробно останавливаться на причинах, которые могут привести к завершению жизненного цикла процесса. После того как процесс завершил свою работу, операционная система переводит его в состояние закончил исполнение и освобождает все ассоциированные с ним ресурсы, делая соответствующие записи в блоке управления процессом. При этом сам PCB не уничтожается, а остается в системе еще некоторое время. Это связано с тем, что процесс-родитель после завершения процесса-ребенка может запросить операционную систему о причине "смерти" порожденного им процесса и/или статистическую информацию о его работе. Подобная информация сохраняется в PCB отработавшего процесса до запроса процесса-родителя или до конца его деятельности, после чего все следы завершившегося процесса окончательно исчезают из системы. В операционной системе Unix процессы, находящиеся в состоянии закончил исполнение, принято называть процессами-зомби.

Следует заметить, что в ряде операционных систем (например, в VAX/VMS) гибель процесса-родителя приводит к завершению работы всех его "детей". В других операционных системах (например, в Unix) процессы-дети продолжают свое существование и после окончания работы процесса-родителя. При этом возникает необходимость изменения информации в PCB процессов-детей о породившем их процессе для того, чтобы генеалогический лес процессов оставался целостным. Рассмотрим следующий пример. Пусть процесс с номером 2515 был порожден процессом с номером 2001 и после завершения его работы остается в вычислительной системе неограниченно долго. Тогда не исключено, что номер 2001 будет использован операционной системой повторно для совсем другого процесса. Если не изменить информацию о процессе-родителе для процесса 2515, то генеалогический лес процессов окажется некорректным – процесс 2515 будет считать своим родителем новый процесс 2001, а процесс 2001 будет открещиваться от нежданного потомка. Как правило, "осиротевшие" процессы "усыновляются" одним из системных процессов, который порождается при старте операционной системы и функционирует все время, пока она работает.

**Многоразовые операции**

Одноразовые операции приводят к изменению количества процессов, находящихся под управлением операционной системы, и всегда связаны с выделением или освобождением определенных ресурсов. Многоразовые операции, напротив, не приводят к изменению количества процессов в операционной системе и не обязаны быть связанными с выделением или освобождением ресурсов.

В этом разделе мы кратко опишем действия, которые производит операционная система при выполнении многоразовых операций над процессами. Более подробно эти действия будут рассмотрены далее в соответствующих лекциях.

Запуск процесса. Из числа процессов, находящихся в состоянии готовность, операционная система выбирает один процесс для последующего исполнения. Критерии и алгоритмы такого выбора будут подробно рассмотрены в лекции 3 – "Планирование процессов ". Для избранного процесса операционная система обеспечивает наличие в оперативной памяти информации, необходимой для его дальнейшего выполнения. То, как она это делает, будет в деталях описано в лекциях 8-10. Далее состояние процесса изменяется на исполнение , восстанавливаются значения регистров для данного процесса и управление передается команде, на которую указывает счетчик команд процесса. Все данные, необходимые для восстановления контекста, извлекаются из PCB процесса, над которым совершается операция.

Приостановка процесса. Работа процесса, находящегося в состоянии исполнение, приостанавливается в результате какого-либо прерывания. Процессор автоматически сохраняет счетчик команд и, возможно, один или несколько регистров в стеке исполняемого процесса, а затем передает управление по специальному адресу обработки данного прерывания. На этом деятельность hardware по обработке прерывания завершается. По указанному адресу обычно располагается одна из частей операционной системы. Она сохраняет динамическую часть системного и регистрового контекстов процесса в его PCB, переводит процесс в состояние готовность и приступает к обработке прерывания, то есть к выполнению определенных действий, связанных с возникшим прерыванием.

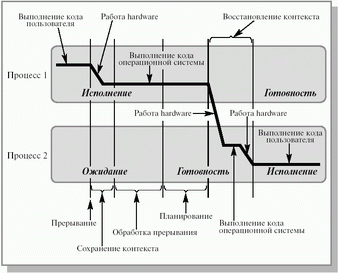
Блокирование процесса. Процесс блокируется, когда он не может продолжать работу, не дождавшись возникновения какого-либо события в вычислительной системе. Для этого он обращается к операционной системе с помощью определенного системного вызова. Операционная система обрабатывает системный вызов (инициализирует операцию ввода-вывода, добавляет процесс в очередь процессов, дожидающихся освобождения устройства или возникновения события, и т. д.) и, при необходимости сохранив нужную часть контекста процесса в его PCB, переводит процесс из состояния исполнение в состояние ожидание . Подробнее эта операция будет рассматриваться в лекции 13.

Разблокирование процесса. После возникновения в системе какого-либо события операционной системе нужно точно определить, какое именно событие произошло. Затем операционная система проверяет, находился ли некоторый процесс в состоянии ожидание для данного события, и если находился, переводит его в состояние готовность, выполняя необходимые действия, связанные с наступлением события (инициализация операции ввода-вывода для очередного ожидающего процесса и т. п.).

**Переключение контекста**

До сих пор мы рассматривали операции над процессами изолированно, независимо друг от друга. В действительности же деятельность мультипрограммной операционной системы состоит из цепочек операций, выполняемых над различными процессами, и сопровождается переключением процессора с одного процесса на другой.

Давайте для примера упрощенно рассмотрим, как в реальности может протекать операция разблокирования процесса, ожидающего ввода-вывода (см. [рисунок](http://www.intuit.ru/department/os/osintro/2/osintro_2.html#image.2.5) 2). При исполнении процессором некоторого процесса (на рисунке – процесс 1) возникает прерывание от устройства ввода-вывода, сигнализирующее об окончании операций на устройстве. Над выполняющимся процессом производится операция приостановки. Далее операционная система разблокирует процесс, инициировавший запрос на ввод-вывод (на рисунке – процесс 2) и осуществляет запуск приостановленного или нового процесса, выбранного при выполнении планирования (на рисунке был выбран разблокированный процесс ). Как мы видим, в результате обработки информации об окончании операции ввода-вывода возможна смена процесса, находящегося в состоянии исполнение.



**Рисунок 2 –**  Выполнение операции разблокирования процесса. Использование термина "код пользователя" не ограничивает общности рисунка только пользовательскими процессами

Для корректного переключения процессора с одного процесса на другой необходимо сохранить контекст исполнявшегося процесса и восстановить контекст процесса, на который будет переключен процессор. Такая процедура сохранения/восстановления работоспособности процессов называется переключением контекста. Время, затраченное на переключение контекста, не используется вычислительной системой для совершения полезной работы и представляет собой накладные расходы, снижающие производительность системы. Оно меняется от машины к машине и обычно колеблется в диапазоне от 1 до 1000 микросекунд. Существенно сократить накладные расходы в современных операционных системах позволяет расширенная модель процессов, включающая в себя понятие threads of execution (нити исполнения или просто нити).

**Контрольные вопросы:**

1. Раскройте понятие процесса.

2. Охарактеризуйте модель представления процесса в ОС.

3. Укажите схему стандартных состояний процесса.

4. Раскройте понятие контекста процесса.

5. Охарактеризуйте процедуры сохранения и переключения контекста – на примере Windows -подобных систем.

6. Перечислите операции над процессами – на примере Windows -подобных систем.

**Литература:**

1. А.П. Побегайло, Системное программирование в Windows, БХВ-Петербург, 2006, 1056 с.

2. Ахо, Лам, Сети, Ульман - Компиляторы. Принципы, технологии, инструменты, Изд.: Вильямс, 2008, 1185 с.

3. Л.Бек, Введение в системное программирование, М., Мир, 2008г.

4. Р. И. Компаниец, Системное программирование : Основы построения трансляторов, Изд.: СПб., 2000.

5. Ф.Льюис, Д.Розенкранц, Р.Стирнз. Теоретические основы проектирования компиляторов. - М.: Мир, 20099, 655 с.

6. Э. Таненбау, Современные операционные системы, Изд.: Питер, 2011, 1120 с.

7. А. В. Гордеев, А. Ю. Молчанов, Системное программное обеспечение, учебник для ВУЗов, СПб.: Питер, 2010г.

**СЕМИНАР №10: «ПЛАНИРОВАНИЕ ПРОЦЕССОВ».**

**План семинара:**

1. Уровни планирования процессов.

2. Критерии планирования процессов.

3. Параметры планирования.

4. Алгоритмы планирования.

**Содержание по плану:**

**1. Уровни планирования процессов.**

Планирование заданий появилось в пакетных системах после того, как для хранения сформированных пакетов заданий начали использоваться магнитные диски. Магнитные диски, являясь устройствами прямого доступа, позволяют загружать задания в компьютер в произвольном порядке, а не только в том, в котором они были записаны на диск. Изменяя порядок загрузки заданий в вычислительную систему, можно повысить эффективность ее использования. Процедуру выбора очередного задания для загрузки в машину, т. е. для порождения соответствующего процесса, мы и назвали планированием заданий. Планирование использования процессора впервые возникает в мультипрограммных вычислительных системах, где в состоянии готовность могут одновременно находиться несколько процессов. Именно для процедуры выбора из них одного процесса, который получит процессор в свое распоряжение, т. е. будет переведен в состояние исполнение, мы использовали это словосочетание. Теперь, познакомившись с концепцией процессов в вычислительных системах, оба вида планирования мы будем рассматривать как различные уровни планирования процессов.

Планирование заданий используется в качестве долгосрочного планирования процессов. Оно отвечает за порождение новых процессов в системе, определяя ее степень мультипрограммирования, т. е. количество процессов, одновременно находящихся в ней. Если степень мультипрограммирования системы поддерживается постоянной, т. е. среднее количество процессов в компьютере не меняется, то новые процессы могут появляться только после завершения ранее загруженных. Поэтому долгосрочное планирование осуществляется достаточно редко, между появлением новых процессов могут проходить минуты и даже десятки минут. Решение о выборе для запуска того или иного процесса оказывает влияние на функционирование вычислительной системы на протяжении достаточно длительного времени. Отсюда и название этого уровня планирования – долгосрочное. В некоторых операционных системах долгосрочное планирование сведено к минимуму или отсутствует вовсе. Так, например, во многих интерактивных системах разделения времени порождение процесса происходит сразу после появления соответствующего запроса. Поддержание разумной степени мультипрограммирования осуществляется за счет ограничения количества пользователей, которые могут работать в системе, и особенностей человеческой психологии. Если между нажатием на клавишу и появлением символа на экране проходит 20–30 секунд, то многие пользователи предпочтут прекратить работу и продолжить ее, когда система будет менее загружена.

Планирование использования процессора применяется в качестве краткосрочного планирования процессов. Оно проводится, к примеру, при обращении исполняющегося процесса к устройствам ввода-вывода или просто по завершении определенного интервала времени. Поэтому краткосрочное планирование осуществляется, как правило, не реже одного раза в 100 миллисекунд. Выбор нового процесса для исполнения оказывает влияние на функционирование системы до наступления очередного аналогичного события, т. е. в течение короткого промежутка времени, чем и обусловлено название этого уровня планирования – краткосрочное.

В некоторых вычислительных системах бывает выгодно для повышения производительности временно удалить какой-либо частично выполнившийся процесс из оперативной памяти на диск, а позже вернуть его обратно для дальнейшего выполнения. Такая процедура в англоязычной литературе получила название swapping, что можно перевести на русский язык как "перекачка", хотя в специальной литературе оно употребляется без перевода – свопинг. Когда и какой из процессов нужно перекачать на диск и вернуть обратно, решается дополнительным промежуточным уровнем планирования процессов – среднесрочным .

**2. Критерии планирования процессов.**

Для каждого уровня планирования процессов можно предложить много различных алгоритмов. Выбор конкретного алгоритма определяется классом задач, решаемых вычислительной системой, и целями, которых мы хотим достичь, используя планирование. К числу таких целей можно отнести следующие:

* Справедливость – гарантировать каждому заданию или процессу определенную часть времени использования процессора в компьютерной системе, стараясь не допустить возникновения ситуации, когда процесс одного пользователя постоянно занимает процессор, в то время как процесс другого пользователя фактически не начинал выполняться.
* Эффективность – постараться занять процессор на все 100% рабочего времени, не позволяя ему простаивать в ожидании процессов, готовых к исполнению. В реальных вычислительных системах загрузка процессора колеблется от 40 до 90%.
* Сокращение полного времени выполнения ( turnaround time ) – обеспечить минимальное время между стартом процесса или постановкой задания в очередь для загрузки и его завершением.
* Сокращение времени ожидания ( waiting time ) – сократить время, которое проводят процессы в состоянии готовность и задания в очереди для загрузки.
* Сокращение времени отклика ( response time ) – минимизировать время, которое требуется процессу в интерактивных системах для ответа на запрос пользователя.

**3. Параметры планирования.**

Для осуществления поставленных целей разумные алгоритмы планирования должны опираться на какие-либо характеристики процессов в системе, заданий в очереди на загрузку, состояния самой вычислительной системы, иными словами, на параметры планирования . В этом разделе мы опишем ряд таких параметров, не претендуя на полноту изложения.

Все параметры планирования можно разбить на две большие группы: статические параметры и динамические параметры. Статические параметры не изменяются в ходе функционирования вычислительной системы, динамические же, напротив, подвержены постоянным изменениям.

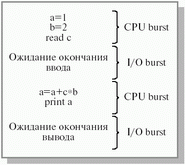
К статическим параметрам вычислительной системы можно отнести предельные значения ее ресурсов (размер оперативной памяти, максимальное количество памяти на диске для осуществления свопинга, количество подключенных устройств ввода-вывода и т. п.). Динамические параметры системы описывают количество свободных ресурсов на данный момент.

К статическим параметрам процессов относятся характеристики, как правило присущие заданиям уже на этапе загрузки.

* Каким пользователем запущен процесс или сформировано задание.
* Насколько важной является поставленная задача, т. е. каков приоритет ее выполнения.
* Сколько процессорного времени запрошено пользователем для решения задачи.
* Каково соотношение процессорного времени и времени, необходимого для осуществления операций ввода-вывода.
* Какие ресурсы вычислительной системы (оперативная память, устройства ввода-вывода, специальные библиотеки и системные программы и т. д.) и в каком количестве необходимы заданию.

Алгоритмы долгосрочного планирования используют в своей работе статические и динамические параметры вычислительной системы и статические параметры процессов (динамические параметры процессов на этапе загрузки заданий еще не известны). Алгоритмы краткосрочного и среднесрочного планирования дополнительно учитывают и динамические характеристики процессов. Для среднесрочного планирования в качестве таких характеристик может использоваться следующая информация:

* сколько времени прошло с момента выгрузки процесса на диск или его загрузки в оперативную память;
* сколько оперативной памяти занимает процесс;
* сколько процессорного времени уже предоставлено процессу.



**Рисунок 1 –** Фрагмент деятельности процесса с выделением промежутков непрерывного использования процессора и ожидания ввода-вывода

Для краткосрочного планирования нам понадобится ввести еще два динамических параметра. Деятельность любого процесса можно представить как последовательность циклов использования процессора и ожидания завершения операций ввода-вывода. Промежуток времени непрерывного использования процессора носит название CPU burst, а промежуток времени непрерывного ожидания ввода-вывода – I/O burst . На рисунке 1показан фрагмент деятельности некоторого процесса на псевдоязыке программирования с выделением указанных промежутков. Для краткости мы будем использовать термины CPU burst и I/O burst без перевода. Значения продолжительности последних и очередных CPU burst и I/O burst являются важными динамическими параметрами процесса.

**4. Алгоритмы планирования.**

Независимо от поставленных целей планирования желательно также, чтобы алгоритмы обладали следующими свойствами.

* Были предсказуемыми. Одно и то же задание должно выполняться приблизительно за одно и то же время. Применение алгоритма планирования не должно приводить, к примеру, к извлечению квадратного корня из 4 за сотые доли секунды при одном запуске и за несколько суток – при втором запуске.
* Были связаны с минимальными накладными расходами. Если на каждые 100 миллисекунд, выделенные процессу для использования процессора, будет приходиться 200 миллисекунд на определение того, какой именно процесс получит процессор в свое распоряжение, и на переключение контекста, то такой алгоритм, очевидно, применять не стоит.
* Равномерно загружали ресурсы вычислительной системы, отдавая предпочтение тем процессам, которые будут занимать малоиспользуемые ресурсы.
* Обладали масштабируемостью, т. е. не сразу теряли работоспособность при увеличении нагрузки. Например, рост количества процессов в системе в два раза не должен приводить к увеличению полного времени выполнения процессов на порядок.

Существует достаточно большой набор разнообразных алгоритмов планирования, которые предназначены для достижения различных целей и эффективны для разных классов задач. Многие из них могут использоваться на нескольких уровнях планирования. В этом разделе мы рассмотрим некоторые наиболее употребительные алгоритмы применительно к процессу кратковременного планирования.

**First-Come, First-Served (FCFS)**

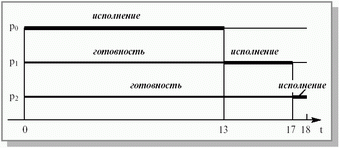
Простейшим алгоритмом планирования является алгоритм, который принято обозначать аббревиатурой FCFS по первым буквам его английского названия – First-Come, First-Served (первым пришел, первым обслужен). Представим себе, что процессы, находящиеся в состоянии готовность, выстроены в очередь. Когда процесс переходит в состояние готовность, он, а точнее, ссылка на его PCB помещается в конец этой очереди. Выбор нового процесса для исполнения осуществляется из начала очереди с удалением оттуда ссылки на его PCB. Очередь подобного типа имеет в программировании специальное наименование – FIFO, сокращение от First In, First Out (первым вошел, первым вышел).

Такой алгоритм выбора процесса осуществляет невытесняющее планирование. Процесс, получивший в свое распоряжение процессор, занимает его до истечения текущего CPU burst . После этого для выполнения выбирается новый процесс из начала очереди.

|  |  |  |  |
| --- | --- | --- | --- |
| Таблица 1. | | | |
| **Процесс** | p0 | p1 | p2 |
| **Продолжительность очередного CPU burst** | 13 | 4 | 1 |

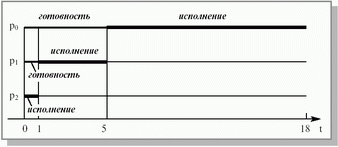
Преимуществом алгоритма FCFS является легкость его реализации, но в то же время он имеет и много недостатков. Рассмотрим следующий пример. Пусть в состоянии готовность находятся три процесса p0, p1 и p2, для которых известны времена их очередных CPU burst . Эти времена приведены в [таблице 1.](http://www.intuit.ru/department/os/osintro/3/3.html#table.3.1) в некоторых условных единицах. Для простоты будем полагать, что вся деятельность процессов ограничивается использованием только одного промежутка CPU burst , что процессы не совершают операций ввода-вывода и что время переключения контекста так мало, что им можно пренебречь.

Если процессы расположены в очереди процессов, готовых к исполнению, в порядке p0, p1, p2, то картина их выполнения выглядит так, как показано на [рисунке](http://www.intuit.ru/department/os/osintro/3/3.html#image.3.2) 1. Первым для выполнения выбирается процесс p0, который получает процессор на все время своего CPU burst , т. е. на 13 единиц времени. После его окончания в состояние исполнение переводится процесс p1, он занимает процессор на 4 единицы времени. И, наконец, возможность работать получает процесс p2. Время ожидания для процесса p0 составляет 0 единиц времени, для процесса p1 – 13 единиц, для процесса p2 – 13 + 4 = 17 единиц. Таким образом, среднее время ожидания в этом случае – (0 + 13 + 17)/3 = 10 единиц времени. Полное время выполнения для процесса p0 составляет 13 единиц времени, для процесса p1 – 13 + 4 = 17 единиц, для процесса p2 – 13 + 4 + 1 = 18 единиц. Среднее полное время выполнения оказывается равным (13 + 17 + 18)/3 = 16 единицам времени.



**Рисунок 1.**  Выполнение процессов при порядке p0,p1,p2

Если те же самые процессы расположены в порядке p2, p1, p0, то картина их выполнения будет соответствовать [рисунку 2](http://www.intuit.ru/department/os/osintro/3/3.html#image.3.3).Время ожидания для процесса p0 равняется 5 единицам времени, для процесса p1 – 1 единице, для процесса p2 – 0 единиц. Среднее время ожидания составит (5 + 1 + 0)/3 = 2 единицы времени. Это в 5 (!) раз меньше, чем в предыдущем случае. Полное время выполнения для процесса p0 получается равным 18 единицам времени, для процесса p1 – 5 единицам, для процесса p2 – 1 единице. Среднее полное время выполнения составляет (18 + 5 + 1)/3 = 8 единиц времени, что почти в 2 раза меньше, чем при первой расстановке процессов.

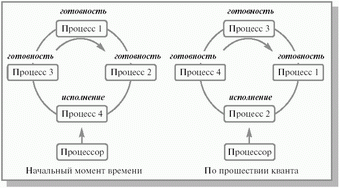


**Рисунок 2** – Выполнение процессов при порядке p2, p1, p0

Как мы видим, среднее время ожидания и среднее полное время выполнения для этого алгоритма существенно зависят от порядка расположения процессов в очереди. Если у нас есть процесс с длительным CPU burst , то короткие процессы, перешедшие в состояние готовность после длительного процесса, будут очень долго ждать начала выполнения. Поэтому алгоритм FCFS практически неприменим для систем разделения времени – слишком большим получается среднее время отклика в интерактивных процессах.

**Round Robin (RR)**

Модификацией алгоритма FCFS является алгоритм, получивший название Round Robin (Round Robin – это вид детской карусели в США) или сокращенно RR. По сути дела, это тот же самый алгоритм, только реализованный в режиме вытесняющего планирования. Можно представить себе все множество готовых процессов организованным циклически – процессы сидят на карусели. Карусель вращается так, что каждый процесс находится около процессора небольшой фиксированный квант времени, обычно 10 – 100 миллисекунд (см. [рис. 3](http://www.intuit.ru/department/os/osintro/3/3.html#image.3.4)). Пока процесс находится рядом с процессором, он получает процессор в свое распоряжение и может исполняться.



**Рисунок 3 –** Процессы на карусели

Реализуется такой алгоритм так же, как и предыдущий, с помощью организации процессов, находящихся в состоянии готовность, в очередь FIFO. Планировщик выбирает для очередного исполнения процесс, расположенный в начале очереди, и устанавливает таймер для генерации прерывания по истечении определенного кванта времени. При выполнении процесса возможны два варианта.

* Время непрерывного использования процессора, необходимое процессу (остаток текущего CPU burst ), меньше или равно продолжительности кванта времени. Тогда процесс по своей воле освобождает процессор до истечения кванта времени, на исполнение поступает новый процесс из начала очереди, и таймер начинает отсчет кванта заново.
* Продолжительность остатка текущего CPU burst процесса больше, чем квант времени. Тогда по истечении этого кванта процесс прерывается таймером и помещается в конец очереди процессов, готовых к исполнению, а процессор выделяется для использования процессу, находящемуся в ее начале.

Рассмотрим предыдущий пример с порядком процессов p0, p1, p2 и величиной кванта времени равной 4. Выполнение этих процессов иллюстрируется [таблицей 2](http://www.intuit.ru/department/os/osintro/3/3.html#table.3.2). Обозначение "И" используется в ней для процесса, находящегося в состоянии исполнение, обозначение "Г" – для процессов в состоянии готовность, пустые ячейки соответствуют завершившимся процессам. Состояния процессов показаны на протяжении соответствующей единицы времени, т. е. колонка с номером 1 соответствует промежутку времени от 0 до 1.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Таблица 2. | | | | | | | | | | | | | | | | | | |
| **Время** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| **p0** | И | И | И | И | Г | Г | Г | Г | Г | И | И | И | И | И | И | И | И | И |
| **p1** | Г | Г | Г | Г | И | И | И | И |  |  |  |  |  |  |  |  |  |  |
| **p2** | Г | Г | Г | Г | Г | Г | Г | Г | И |  |  |  |  |  |  |  |  |  |

Первым для исполнения выбирается процесс p0. Продолжительность его CPU burst больше, чем величина кванта времени, и поэтому процесс исполняется до истечения кванта, т. е. в течение 4 единиц времени. После этого он помещается в конец очереди готовых к исполнению процессов, которая принимает вид p1, p2, p0. Следующим начинает выполняться процесс p1. Время его исполнения совпадает с величиной выделенного кванта, поэтому процесс работает до своего завершения. Теперь очередь процессов в состоянии готовность состоит из двух процессов, p2 и p0. Процессор выделяется процессу p2. Он завершается до истечения отпущенного ему процессорного времени, и очередные кванты отмеряются процессу p0 – единственному не закончившему к этому моменту свою работу. Время ожидания для процесса p0 (количество символов "Г" в соответствующей строке) составляет 5 единиц времени, для процесса p1 – 4 единицы времени, для процесса p2 – 8 единиц времени. Таким образом, среднее время ожидания для этого алгоритма получается равным (5 + 4 + 8)/3 = 5,6(6) единицы времени. Полное время выполнения для процесса p0 (количество непустых столбцов в соответствующей строке) составляет 18 единиц времени, для процесса p1 – 8 единиц, для процесса p2 – 9 единиц. Среднее полное время выполнения оказывается равным (18 + 8 + 9)/3 = 11,6(6) единицы времени.

Легко увидеть, что среднее время ожидания и среднее полное время выполнения для обратного порядка процессов не отличаются от соответствующих времен для алгоритма FCFS и составляют 2 и 8 единиц времени соответственно.

На производительность алгоритма RR сильно влияет величина кванта времени. Рассмотрим тот же самый пример с порядком процессов p0, p1, p2 для величины кванта времени, равной 1 (см. [табл. 3.](http://www.intuit.ru/department/os/osintro/3/3.html#table.3.3)). Время ожидания для процесса p0 составит 5 единиц времени, для процесса p1 – тоже 5 единиц, для процесса p2 – 2 единицы. В этом случае среднее время ожидания получается равным (5 + 5 + 2)/3 = 4 единицам времени. Среднее полное время исполнения составит (18 + 9 + 3)/3 = 10 единиц времени.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Таблица 3. | | | | | | | | | | | | | | | | | | |
| **Время** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| **p0** | И | Г | Г | И | Г | И | Г | И | Г | И | И | И | И | И | И | И | И | И |
| **p1** | Г | И | Г | Г | И | Г | И | Г | И |  |  |  |  |  |  |  |  |  |
| **p2** | Г | Г | И |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

При очень больших величинах кванта времени, когда каждый процесс успевает завершить свой CPU burst до возникновения прерывания по времени, алгоритм RR вырождается в алгоритм FCFS. При очень малых величинах создается иллюзия того, что каждый из n процессов работает на собственном виртуальном процессоре с производительностью ~ 1/n от производительности реального процессора. Правда, это справедливо лишь при теоретическом анализе при условии пренебрежения временами переключения контекста процессов. В реальных условиях при слишком малой величине кванта времени и, соответственно, слишком частом переключении контекста накладные расходы на переключение резко снижают производительность системы.

#### Shortest-Job-First (SJF)

При рассмотрении алгоритмов FCFS и RR мы видели, насколько существенным для них является порядок расположения процессов в очереди процессов, готовых к исполнению. Если короткие задачи расположены в очереди ближе к ее началу, то общая производительность этих алгоритмов значительно возрастает. Если бы мы знали время следующих CPU burst для процессов, находящихся в состоянии готовность, то могли бы выбрать для исполнения не процесс из начала очереди, а процесс с минимальной длительностью CPU burst. Если же таких процессов два или больше, то для выбора одного из них можно использовать уже известный нам алгоритм FCFS. Квантование времени при этом не применяется. Описанный алгоритм получил название "кратчайшая работа первой" или ShortestJobFirst( SJF ).

SJF-алгоритмкраткосрочного планирования может быть как вытесняющим, так и невытесняющим. При невытесняющем SJF - планировании процессор предоставляется избранному процессу на все необходимое ему время, независимо от событий, происходящих в вычислительной системе. При вытесняющем SJF - планировании учитывается появление новых процессов в очереди готовых к исполнению (из числа вновь родившихся или разблокированных) во время работы выбранного процесса. Если CPU burst нового процесса меньше, чем остаток CPU burstу исполняющегося, то исполняющийся процесс вытесняется новым.

Рассмотрим пример работы невытесняющего алгоритма SJF. Пусть в состоянии готовность находятся четыре процесса, p0, p1, p2 и p3, для которых известны времена их очередных CPU burst. Эти времена приведены в [таблице 4](http://www.intuit.ru/department/os/osintro/3/4.html#table.3.4). Как и прежде, будем полагать, что вся деятельность процессов ограничивается использованием только одного промежутка CPU burst, что процессы не совершают операций ввода-вывода и что временем переключения контекста можно пренебречь.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Таблица 4. | | | | |
| **Процесс** | p0 | p1 | p2 | p3 |
| **Продолжительность очередного CPU burst** | 5 | 3 | 7 | 1 |

При использовании невытесняющегоалгоритма SJF первым для исполнения будет выбран процесс p3, имеющий наименьшее значение продолжительности очередного CPU burst. После его завершения для исполнения выбирается процесс p1, затем p0 и, наконец, p2. Эта картина отражена в [таблице 5](http://www.intuit.ru/department/os/osintro/3/4.html#table.3.5).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Таблица 5. | | | | | | | | | | | | | | | | |
| **Время** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| **p0** | Г | Г | Г | Г | И | И | И | И | И |  |  |  |  |  |  |  |
| **p1** | Г | И | И | И |  |  |  |  |  |  |  |  |  |  |  |  |
| **p2** | Г | Г | Г | Г | Г | Г | Г | Г | Г | И | И | И | И | И | И | И |
| **p3** | И |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Как мы видим, среднее время ожидания для алгоритма SJF составляет (4 + 1 + 9 + 0)/4 = 3,5 единицы времени. Легко посчитать, что для алгоритма FCFS при порядке процессов p0, p1, p2, p3 эта величина будет равняться (0 + 5 + 8 + 15)/4 = 7 единицам времени, т. е. будет в два раза больше, чем для алгоритма SJF. Можно показать, что для заданного набора процессов (если в очереди не появляются новые процессы) алгоритм SJF является оптимальным с точки зрения минимизации среднего времени ожидания среди класса невытесняющих алгоритмов.

Для рассмотрения примера вытесняющего SJFпланирования мы возьмем ряд процессов p0, p1, p2 и p3 с различными временами CPU burst и различными моментами их появления в очереди процессов, готовых к исполнению (см. [табл. 6.](http://www.intuit.ru/department/os/osintro/3/4.html#table.3.6)).

|  |  |  |
| --- | --- | --- |
| Таблица 6. | | |
| **Процесс** | **Время появления в очереди очередного CPU burst** | **Продолжительность** |
| **p0** | 0 | 6 |
| **p1** | 2 | 2 |
| **p2** | 6 | 7 |
| **p3** | 0 | 5 |

В начальный момент времени в состоянии готовность находятся только два процесса, p0 и p3. Меньшее время очередного CPU burst оказывается у процесса p3, поэтому он и выбирается для исполнения (см. [таблицу 7.](http://www.intuit.ru/department/os/osintro/3/4.html#table.3.7)). По прошествии2 единиц времени в систему поступает процесс p1. Время его CPU burst меньше, чем остаток CPU burst у процесса p3, который вытесняется из состояния исполнение и переводится в состояние готовность. По прошествии еще 2 единиц времени процесс p1 завершается, и для исполнения вновь выбирается процесс p3. В момент времени t = 6 в очереди процессов, готовых к исполнению, появляется процесс p2, но поскольку ему для работы нужно 7 единиц времени, а процессу p3 осталось трудиться всего 1 единицу времени, то процесс p3 остается в состоянии исполнение. После его завершения в момент времени t = 7 в очереди находятся процессы p0 и p2, из которых выбирается процесс p0. Наконец, последним получит возможность выполняться процесс p2.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Таблица 7. | | | | | | | | | | | | | | | | | | | | |
| **Время** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| **p0** | Г | Г | Г | Г | Г | Г | Г | И | И | И | И | И | И |  |  |  |  |  |  |  |
| **p1** |  |  | И | И |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **p2** |  |  |  |  |  |  | Г | Г | Г | Г | Г | Г | Г | И | И | И | И | И | И | И |
| **p3** | И | И | Г | Г | И | И | И |  |  |  |  |  |  |  |  |  |  |  |  |  |

Основную сложность при реализации алгоритма SJF представляет невозможность точного знания продолжительности очередного CPU burst для исполняющихся процессов. В пакетных системах количество процессорного времени, необходимое заданию для выполнения, указывает пользователь при формировании задания. Мы можем брать эту величину для осуществления долгосрочного SJF - планирования. Если пользователь укажет больше времени, чем ему нужно, он будет ждать результата дольше, чем мог бы, так как задание будет загружено в систему позже. Если же он укажет меньшее количество времени, задача может не досчитаться до конца. Таким образом, в пакетных системах решение задачи оценки времени использования процессора перекладывается на плечи пользователя. При краткосрочном планировании мы можем делать только прогноз длительности следующего CPU burst, исходя из предыстории работы процесса. Пусть http://www.intuit.ru/img/tex/b9137cf9c02aec73c6c61693e97a4db0.png– величина n -го CPU burst, T(n + 1) – предсказываемое значение для n + 1 -го CPU burst, http://www.intuit.ru/img/tex/b6b89f160d0059d8c918387cc5cf8947.png– некоторая величина в диапазоне от 0 до 1.

Определим рекуррентное соотношение

http://www.intuit.ru/img/tex/278315c98e84f9b11fa72c6b7fbc6690.png

T(0) положим произвольной константой. Первое слагаемое учитывает последнее поведение процесса, тогда как второе слагаемое учитывает его предысторию. Приhttp://www.intuit.ru/img/tex/6d28efc2baabf304d4d866867bec0e93.pngмы перестаем следить за последним поведением процесса, фактически полагая

T(n)= T(n+1)=...=T(0)

т. е. оценивая все CPU burst одинаково, исходя из некоторого начального предположения.

Положив http://www.intuit.ru/img/tex/bd283091ec405ddc46861837bf316532.png, мы забываем о предыстории процесса. В этом случае мы полагаем, что время очередного CPU burst будет совпадать со временем последнего CPU burst:

http://www.intuit.ru/img/tex/4d8647cc87234e4381d0fc7048b2903b.png

Обычно выбирают http://www.intuit.ru/img/tex/2223fa99480149101bcb797af808e27c.pngдля равноценного учета последнего поведения и предыстории. Надо отметить, что такой выбор http://www.intuit.ru/img/tex/b6b89f160d0059d8c918387cc5cf8947.pngудобен и для быстрой организации вычисления оценки T(n + 1). Для подсчета новой оценки нужно взять старую оценку, сложить с измеренным временем CPU burst и полученную сумму разделить на 2, например, сдвинув ее на 1 бит вправо. Полученные оценки T(n + 1) применяются как продолжительности очередных промежутков времени непрерывного использования процессора для краткосрочного SJF - планирования.

**Контрольные вопросы:**

1. Перечислите уровни планирования процессов.

2. Перечислите критерии планирования процессов.

3. Перечислите параметры планирования.

4. Перечислите и озхарактеризуйте алгоритмы планирования.

**Литература:**

1. А.П. Побегайло, Системное программирование в Windows, БХВ-Петербург, 2006, 1056 с.

2. Ахо, Лам, Сети, Ульман - Компиляторы. Принципы, технологии, инструменты, Изд.: Вильямс, 2008, 1185 с.

3. Л.Бек, Введение в системное программирование, М., Мир, 2008г.

4. Р. И. Компаниец, Системное программирование : Основы построения трансляторов, Изд.: СПб., 2000.

5. Ф.Льюис, Д.Розенкранц, Р.Стирнз. Теоретические основы проектирования компиляторов. - М.: Мир, 20099, 655 с.

6. Э. Таненбау, Современные операционные системы, Изд.: Питер, 2011, 1120 с.

7. А. В. Гордеев, А. Ю. Молчанов, Системное программное обеспечение, учебник для ВУЗов, СПб.: Питер, 2010г.

**СЕМИНАР №11: «ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ».**

**План семинара:**

1. Взаимодействие процессов.

2. Логическая организация взаимодействия.

**Содержание по плану:**

**1. Взаимодействие процессов.**

Для достижения поставленной цели различные процессы (возможно, даже принадлежащие разным пользователям) могут исполняться псевдопараллельно на одной вычислительной системе или параллельно на разных вычислительных системах, взаимодействуя между собой.

Для чего процессам нужно заниматься совместной деятельностью? Какие существуют причины для их кооперации?

* Повышение скорости работы. Пока один процесс ожидает наступления некоторого события (например, окончания операции ввода-вывода), другие могут заниматься полезной работой, направленной на решение общей задачи. В многопроцессорных вычислительных системах программа разбивается на отдельные кусочки, каждый из которых будет исполняться на своем процессоре.
* Совместное использование данных. Различные процессы могут, к примеру, работать с одной и той же динамической базой данных или с разделяемым файлом, совместно изменяя их содержимое.
* Модульная конструкция какой-либо системы. Типичным примером может служить микроядерный способ построения операционной системы, когда различные ее части представляют собой отдельные процессы, взаимодействующие путем передачи сообщений через микроядро.
* Наконец, это может быть необходимо просто для удобства работы пользователя, желающего, например, редактировать и отлаживать программу одновременно. В этой ситуации процессы редактора и отладчика должны уметь взаимодействовать друг с другом.

Процессы не могут взаимодействовать, не общаясь, то есть не обмениваясь информацией. "Общение" процессов обычно приводит к изменению их поведения в зависимости от полученной информации. Если деятельность процессов остается неизменной при любой принятой ими информации, то это означает, что они на самом деле в "общении" не нуждаются. Процессы, которые влияют на поведение друг друга путем обмена информацией, принято называть **кооперативными** или взаимодействующими процессами , в отличие от независимых процессов, не оказывающих друг на друга никакого воздействия.

Различные процессы в вычислительной системе изначально представляют собой обособленные сущности. Работа одного процесса не должна приводить к нарушению работы другого процесса. Для этого, в частности, разделены их адресные пространства и системные ресурсы, и для обеспечения корректного взаимодействия процессов требуются специальные средства и действия операционной системы. Нельзя просто поместить значение, вычисленное в одном процессе, в область памяти, соответствующую переменной в другом процессе, не предприняв каких-либо дополнительных усилий. Давайте рассмотрим основные аспекты организации совместной работы процессов.

**Категории средств обмена информацией**

Процессы могут взаимодействовать друг с другом, только обмениваясь информацией. По объему передаваемой информации и степени возможного воздействия на поведение другого процесса все средства такого обмена можно разделить на три категории.

* Сигнальные. Передается минимальное количество информации – один бит, "да" или "нет". Используются, как правило, для извещения процесса о наступлении какого-либо события. Степень воздействия на поведение процесса, получившего информацию, минимальна. Все зависит от того, знает ли он, что означает полученный сигнал, надо ли на него реагировать и каким образом. Неправильная реакция на сигнал или его игнорирование могут привести к трагическим последствиям. Вспомним профессора Плейшнера из кинофильма "Семнадцать мгновений весны". Сигнал тревоги – цветочный горшок на подоконнике – был ему передан, но профессор проигнорировал его. И к чему это привело?
* Канальные. "Общение" процессов происходит через линии связи, предоставленные операционной системой, и напоминает общение людей по телефону, с помощью записок, писем или объявлений. Объем передаваемой информации в единицу времени ограничен пропускной способностью линий связи. С увеличением количества информации возрастает и возможность влияния на поведение другого процесса.
* Разделяемая память. Два или более процессов могут совместно использовать некоторую область адресного пространства. Созданием разделяемой памяти занимается операционная система (если, конечно, ее об этом попросят). "Общение" процессов напоминает совместное проживание студентов в одной комнате общежития. Возможность обмена информацией максимальна, как, впрочем, и влияние на поведение другого процесса, но требует повышенной осторожности (если вы переложили на другое место вещи вашего соседа по комнате, а часть из них еще и выбросили). Использование разделяемой памяти для передачи/получения информации осуществляется с помощью средств обычных языков программирования, в то время как сигнальным и канальным средствам коммуникации для этого необходимы специальные системные вызовы. Разделяемая память представляет собой наиболее быстрый способ взаимодействия процессов в одной вычислительной системе.

**2. Логическая организация взаимодействия.**

При рассмотрении любого из средств коммуникации нас будет интересовать не их физическая реализация (общая шина данных, прерывания, аппаратно разделяемая память и т. д.), а логическая, определяющая в конечном счете механизм их использования. Некоторые важные аспекты логической реализации являются общими для всех категорий средств связи, некоторые относятся к отдельным категориям. Давайте кратко охарактеризуем основные вопросы, требующие разъяснения при изучении того или иного способа обмена информацией.

**Как устанавливается связь?**

Могу ли я использовать средство связи непосредственно для обмена информацией сразу после создания процесса или первоначально необходимо предпринять определенные действия для инициализации обмена? Например, для использования общей памяти различными процессами потребуется специальное обращение к операционной системе, которая выделит необходимую область адресного пространства. Но для передачи сигнала от одного процесса к другому никакая инициализация не нужна. В то же время передача информации по линиям связи может потребовать первоначального резервирования такой линии для процессов, желающих обменяться информацией.

К этому же вопросу тесно примыкает вопрос о способе адресации при использовании средства связи. Если я передаю некоторую информацию, я должен указать, куда я ее передаю. Если я желаю получить некоторую информацию, то мне нужно знать, откуда я могу ее получить.

Различают два способа адресации: прямую и непрямую. В случае прямой адресации взаимодействующие процессы непосредственно общаются друг с другом, при каждой операции обмена данными явно указывая имя или номер процесса, которому информация предназначена или от которого она должна быть получена. Если и процесс, от которого данные исходят, и процесс, принимающий данные, указывают имена своих партнеров по взаимодействию, то такая схема адресации называется симметричной прямой адресацией . **Ни один другой процесс не может вмешаться в процедуру симметричного прямого общения двух процессов, перехватить посланные или подменить ожидаемые данные**. Если только один из взаимодействующих процессов, например передающий, указывает имя своего партнера по кооперации, а второй процесс в качестве возможного партнера рассматривает любой процесс в системе, например ожидает получения информации от произвольного источника, то такая схема адресации называется асимметричной прямой адресацией .

При непрямой адресации данные помещаются передающим процессом в некоторый промежуточный объект для хранения данных, имеющий свой адрес, откуда они могут быть затем изъяты каким-либо другим процессом. Примером такого объекта может служить обычная доска объявлений или рекламная газета. При этом передающий процесс не знает, как именно идентифицируется процесс, который получит информацию, а принимающий процесс не имеет представления об идентификаторе процесса, от которого он должен ее получить.

При использовании прямой адресации связь между процессами в классической операционной системе устанавливается автоматически, без дополнительных инициализирующих действий. Единственное, что нужно для использования средства связи, – это знать, как идентифицируются процессы, участвующие в обмене данными.

При использовании непрямой адресации инициализация средства связи может и не требоваться. Информация, которой должен обладать процесс для взаимодействия с другими процессами, – это некий идентификатор промежуточного объекта для хранения данных, если он, конечно, не является единственным и неповторимым в вычислительной системе для всех процессов.

**Информационная валентность процессов и средств связи**

Следующий важный вопрос – это вопрос об информационной валентности связи. Слово "валентность" здесь использовано по аналогии с химией. Сколько процессов может быть одновременно ассоциировано с конкретным средством связи? Сколько таких средств связи может быть задействовано между двумя процессами?

Понятно, что при прямой адресации только одно фиксированное средство связи может быть задействовано для обмена данными между двумя процессами, и только эти два процесса могут быть ассоциированы с ним. При непрямой адресации может существовать более двух процессов, использующих один и тот же объект для данных, и более одного объекта может быть использовано двумя процессами.

К этой же группе вопросов следует отнести и вопрос о направленности связи. Является ли связь однонаправленной или двунаправленной? Под однонаправленной связью мы будем понимать связь, при которой каждый процесс, ассоциированный с ней, может использовать средство связи либо только для приема информации, либо только для ее передачи. При двунаправленной связи каждый процесс, участвующий в общении, может использовать связь и для приема, и для передачи данных. В коммуникационных системах принято называть однонаправленную связь симплексной , двунаправленную связь с поочередной передачей информации в разных направлениях – полудуплексной , а двунаправленную связь с возможностью одновременной передачи информации в разных направлениях – дуплексной . Прямая и непрямая адресация не имеет непосредственного отношения к направленности связи.

**Контрольные вопросы:**

1. Раскройте понятие взаимодействия процессов.

2. Опишите логическую организацию взаимодействия процессов.

**Литература:**

1. А.П. Побегайло, Системное программирование в Windows, БХВ-Петербург, 2006, 1056 с.

2. Ахо, Лам, Сети, Ульман - Компиляторы. Принципы, технологии, инструменты, Изд.: Вильямс, 2008, 1185 с.

3. Л.Бек, Введение в системное программирование, М., Мир, 2008г.

4. Р. И. Компаниец, Системное программирование : Основы построения трансляторов, Изд.: СПб., 2000.

5. Ф.Льюис, Д.Розенкранц, Р.Стирнз. Теоретические основы проектирования компиляторов. - М.: Мир, 20099, 655 с.

6. Э. Таненбау, Современные операционные системы, Изд.: Питер, 2011, 1120 с.

7. А. В. Гордеев, А. Ю. Молчанов, Системное программное обеспечение, учебник для ВУЗов, СПб.: Питер, 2010 г.

**СЕМИНАР №12: «СИНХРОНИЗАЦИЯ ПРОЦЕССОВ».**

**План семинара:**

1. Алгоритмы синхронизации процессов.

2. Понятие состязания, взаимоисключения, чередования, критической секции.

3. Программные и аппаратные алгоритмы взаимодействия процессов.

4. Механизмы синхронизации.

5. Концепция семафора.

6. Мониторы.

7. Очереди сообщений.

**Содержание по плану:**

**1. Алгоритмы синхронизации процессов.**

Для корректного взаимодействия процессов недостаточно одних организационных усилий операционнойсистемы. Необходимы определенные внутренние изменения в поведении процессов.

Interleaving, race condition и взаимоисключения

Под активностями мы будем понимать последовательное выполнение рядадействий, направленных на достижение определенной цели. Активности могут иметь место в программном и техническом обеспечении, в обычной деятельности людей и животных. Мы будем разбивать активности на некоторые неделимые, или атомарные, операции. Например, активность "приготовление бутерброда" можно разбить на следующие атомарные операции:

1. Отрезать ломтик хлеба.

2. Отрезать ломтик колбасы.

3. Намазать ломтик хлеба маслом.

4. Положить ломтик колбасы на подготовленный ломтик хлеба.

Неделимые операции могут иметь внутренние невидимые действия (взять батон хлеба в левую руку,взять нож в правую руку, произвести отрезание). Мы же называем их неделимыми потому, что считаемвыполняемыми за раз, без прерывания деятельности.

Пусть имеется две активности

P: a b c

Q: d e f

где a, b, c, d, e, f – атомарные операции. При последовательном выполнении активностей мы получаемтакую последовательность атомарных действий:

PQ: a b c d e f

Что произойдет при исполнении этих активностей псевдопараллельно, в режиме разделения времени?

Активности могут расслоиться на неделимые операции с различным чередованием, то есть может произойти то, что на английском языке принято называть словом interleaving. Возможные варианты чередования:

а b c d e f

a b d c e f

a b d e c f

a b d e f c

a d b c e f

......

d e f a b c

Атомарные операции активностей могут чередоваться всевозможными различными способами с сохранением порядка расположения внутри активностей. Так как псевдопараллельное выполнение двух актив-ностей приводит к чередованию их неделимых операций, результат псевдопараллельного выполненияможет отличаться от результата последовательного выполнения. Рассмотрим пример. Пусть у нас имеется две активности P и Q, состоящие из двух атомарных операций каждая:

P: x=2 Q: x=3

y=x-1 y=x+1

Что мы получим в результате их псевдопараллельного выполнения, если переменные x и y являются дляактивностей общими? Очевидно, что возможны четыре разных набора значений для пары (x, y): (3, 4), (2,1), (2, 3) и (3, 2). . Мы будем говорить, что набор активностей (например, программ) детерминирован, если всякий раз при псевдопараллельном исполнении для одного и того же набора входных данных он дает одинаковые выходные данные. В противном случае он недетерминирован. Выше приведен пример недетерминированного набора программ. Понятно, что детерминированный набор активностей можно безбоязненно выполнять в режиме разделения времени. Для недетерминированного набора такое исполнениенежелательно.

Можно ли до получения результатов определить, является ли набор активностей детерминированнымили нет? Для этого существуют достаточные условия Бернстайна. Изложим их применительно к программам с разделяемыми переменными.

Введем наборы входных и выходных переменных программы. Для каждой атомарной операции наборывходных и выходных переменных – это наборы переменных, которые атомарная операция считывает изаписывает. Набор входных переменных программы R(P) (R от слова read) суть объединение набороввходных переменных для всех ее неделимых действий. Аналогично, набор выходных переменных программы W(P) (W от слова write) суть объединение наборов выходных переменных для всех ее неделимыхдействий. Например, для программыP: x=u+v,y=x\*wполучаем R(P) = {u, v, x, w}, W(P) = {x, y}. Заметим, что переменная x присутствует как в R(P), так и вW(P).

Теперь сформулируем условия Бернстайна.

Если для двух данных активностей P и Q:

• пересечение W(P) и W(Q) пусто,

• пересечение W(P) с R(Q) пусто,

• пересечение R(P) и W(Q) пусто,

тогда выполнение P и Q детерминировано.

Если эти условия не соблюдены, возможно, параллельное выполнение P и Q детерминировано, а можетбыть, и нет.

Случай двух активностей естественным образом обобщается на их большее количество.

Условия Бернстайна информативны, но слишком жестки. По сути дела, они требуют практически невзаимодействующих процессов. А нам хотелось бы, чтобы детерминированный набор образовывали активности, совместно использующие информацию и обменивающиеся ею. Для этого нам необходимо ограничить число возможных чередований атомарных операций, исключив некоторые чередования с помощью механизмов синхронизации выполнения программ, обеспечив тем самым упорядоченный доступпрограмм к некоторым данным.

Про недетерминированный набор программ (и активностей вообще) говорят, что он имеет race condition(состояние гонки , состояние состязания). В приведенном выше примере процессы состязаются за вычисление значений переменных x и y.

Задачу упорядоченного доступа к разделяемым данным (устранение race condition) в том случае, когданам не важна его очередность, можно решить, если обеспечить каждому процессу эксклюзивное праводоступа к этим данным. Каждый процесс, обращающийся к разделяемым ресурсам, исключает для всехдругих процессов возможность одновременного общения с этими ресурсами, если это может привести кнедетерминированному поведению набора процессов. Такой прием называется взаимоисключением(mutual exclusion). Если очередность доступа к разделяемым ресурсам важна для получения правильныхрезультатов, то одними взаимоисключениями уже не обойтись, нужна взаимосинхронизация поведенияпрограмм.

**2. Понятие состязания, взаимоисключения, чередования, критической секции.**

1. **Программные и аппаратные алгоритмы взаимодействия процессов.**

Требования, предъявляемые к алгоритмам

Организация взаимоисключения для критических участков, конечно, позволит избежать возникновения race condition, но не является достаточной для правильной и эффективной параллельной работы кооперативных процессов. Сформулируем пять условий, которые должны выполняться для хорошего программного алгоритма организации взаимодействия процессов, имеющих критические участки, если они могут проходить их в произвольном порядке.

Задача должна быть решена чисто программным способом на обычной машине, не имеющей специальных команд взаимоисключения. При этом предполагается, что основные инструкции языка программирования (такие примитивные инструкции, как load, store, test ) являются атомарными операциями.

Не должно существовать никаких предположений об относительных скоростях выполняющихся процессов или числе процессоров, на которых они исполняются.

Если процесс Pi исполняется в своем критическом участке, то не существует никаких других процессов, которые исполняются в соответствующих критических секциях. Это условие получило название условия взаимоисключения (mutual exclusion).

Процессы, которые находятся вне своих критических участков и не собираются входить в них, не могут препятствовать другим процессам входить в их собственные критические участки. Если нет процессов в критических секциях и имеются процессы, желающие войти в них, то только те процессы, которые не исполняются в remainder section, должны принимать решение о том, какой процесс войдет в свою критическую секцию. Такое решение не должно приниматься бесконечно долго. Это условие получило название условия прогресса (progress) .

Не должно возникать неограниченно долгого ожидания для входа одного из процессов в свой критический участок. От того момента, когда процесс запросил разрешение на вход в критическую секцию, и до того момента, когда он это разрешение получил, другие процессы могут пройти через свои критические участки лишь ограниченное число раз. Это условие получило название условия ограниченного ожидания (bound waiting) .

Надо заметить, что описание соответствующего алгоритма в нашем случае означает описание способа организации пролога и эпилога для критической секции.

**Запрет прерываний**

Наиболее простым решением поставленной задачи является следующая организация пролога и эпилога:

while (some condition) {

запретить все прерывания

critical section

разрешитьвсепрерывания

remainder section

}

Поскольку выход процесса из состояния исполнение без его завершения осуществляется по прерыванию, внутри критической секции никто не может вмешаться в его работу. Однако такое решение может иметь далеко идущие последствия, поскольку позволяет процессу пользователя разрешать и запрещать прерывания во всей вычислительной системе. Допустим, что пользователь случайно или по злому умыслу запретил прерывания в системе и зациклил или завершил свой процесс. Без перезагрузки системы в такой ситуации не обойтись.

Тем не менее запрет и разрешение прерываний часто применяются как пролог и эпилог к критическим секциям внутри самой операционной системы, например при обновлении содержимого PCB.

**Переменная-замок**

В качестве следующей попытки решения задачи для пользовательских процессов рассмотрим другое предложение. Возьмем некоторую переменную, доступную всем процессам, с начальным значением равным 0. Процесс может войти в критическую секцию только тогда, когда значение этой переменной-замка равно 0, одновременно изменяя ее значение на 1 – закрывая замок. При выходе из критической секции процесс сбрасывает ее значение в 0 – замок открывается (как в случае с покупкой хлеба студентами в разделе " Критическая секция ").

shared int lock = 0;

/\* shared означает, что \*/

/\* переменнаяявляетсяразделяемой \*/

while (somecondition) {

while(lock); lock = 1;

critical section

lock = 0;

remaindersection

}

К сожалению, при внимательном рассмотрении мы видим, что такое решение не удовлетворяет условию взаимоисключения, так как действие while(lock); lock = 1; не является атомарным. Допустим, процесс P0 протестировал значение переменной lock и принял решение двигаться дальше. В этот момент, еще до присвоения переменной lock значения 1, планировщик передал управление процессу P1. Он тоже изучает содержимое переменной lock и тоже принимает решение войти в критический участок. Мы получаем два процесса, одновременно выполняющих свои критические секции.

**Строгое чередование**

Попробуем решить задачу сначала для двух процессов. Очередной подход будет также использовать общую для них обоих переменную с начальным значением 0. Только теперь она будет играть не роль замка для критического участка, а явно указывать, кто может следующим войти в него. Для i-го процесса это выглядит так:

shared int turn = 0;

while (some condition) {

while(turn != i);

critical section

turn = 1-i;

remainder section

}

Очевидно, что взаимоисключение гарантируется, процессы входят в критическую секцию строго по очереди: P0, P1, P0, P1, P0, ... Но наш алгоритм не удовлетворяет условию прогресса. Например, если значение turn равно 1, и процесс P0 готов войти в критический участок, он не может сделать этого, даже если процесс P1 находится в remainder section.

**Флаги готовности**

Недостаток предыдущего алгоритма заключается в том, что процессы ничего не знают о состоянии друг друга в текущий момент времени. Давайте попробуем исправить эту ситуацию. Пусть два наших процесса имеют разделяемый массив флагов готовности входа процессов в критический участок

shared int ready[2] = {0, 0};

Когда i-й процесс готов войти в критическую секцию, он присваивает элементу массива ready[i] значение равное 1. После выхода из критической секции он, естественно, сбрасывает это значение в 0. Процесс не входит в критическую секцию, если другой процесс уже готов к входу в критическую секцию или находится в ней.

while (some condition) {

ready[i] = 1;

while(ready[1-i]);

critical section

ready[i] = 0;

remainder section

}

Полученный алгоритм обеспечивает взаимоисключение, позволяет процессу, готовому к входу в критический участок, войти в него сразу после завершения эпилога в другом процессе, но все равно нарушает условие прогресса. Пусть процессы практически одновременно подошли к выполнению пролога. После выполнения присваивания ready[0]=1 планировщик передал процессор от процесса 0 процессу 1, который также выполнил присваивание ready[1]=1. После этого оба процесса бесконечно долго ждут друг друга на входе в критическую секцию. Возникает ситуация, которую принято называть тупиковой (deadlock).

**Алгоритм Петерсона**

Первое решение проблемы, удовлетворяющее всем требованиям и использующее идеи ранее рассмотренных алгоритмов, было предложено датским математиком Деккером (Dekker). В 1981 году Петерсон (Peterson) предложил более изящное решение. Пусть оба процесса имеют доступ к массиву флагов готовности и к переменной очередности.

shared int ready[2] = {0, 0};

shared int turn;

while (some condition) {

ready[i] = 1;

turn =1-i;

while(ready[1-i] && turn == 1-i);

critical section

ready[i] = 0;

remainder section

}

При исполнении пролога критической секции процесс Pi заявляет о своей готовности выполнить критический участок и одновременно предлагает другому процессу приступить к его выполнению. Если оба процесса подошли к прологу практически одновременно, то они оба объявят о своей готовности и предложат выполняться друг другу. При этом одно из предложений всегда следует после другого. Тем самым работу в критическом участке продолжит процесс, которому было сделано последнее предложение.

Давайте докажем, что все пять наших требований к алгоритму действительно удовлетворяются.

Удовлетворение требований 1 и 2 очевидно.

Докажем выполнение условия взаимоисключения методом от противного. Пусть оба процесса одновременно оказались внутри своих критических секций. Заметим, что процесс Pi может войти в критическую секцию, только если ready[1-i] == 0 или turn == i. Заметим также, что если оба процесса выполняют свои критические секции одновременно, то значения флагов готовности для обоих процессов совпадают и равны 1. Могли ли оба процесса войти в критические секции из состояния, когда они оба одновременно находились в процессе выполнения цикла while? Нет, так как в этом случае переменная turn должна была бы одновременно иметь значения 0 и 1 (когда оба процесса выполняют цикл, значения переменных измениться не могут). Пусть процесс P0 первым вошел в критический участок, тогда процесс P1 должен был выполнить перед вхождением в цикл while по крайней мере один предваряющий оператор (turn = 0;). Однако после этого он не может выйти из цикла до окончания критического участка процесса P0, так как при входе в цикл ready[0] == 1 и turn == 0, и эти значения не могут измениться до тех пор, пока процесс P0 не покинет свой критический участок. Мы пришли к противоречию. Следовательно, имеет место взаимоисключение.

Докажем выполнение условия прогресса. Возьмем, без ограничения общности, процесс P0. Заметим, что он не может войти в свою критическую секцию только при совместном выполнении условий ready[1] == 1 и turn == 1. Если процесс P1 не готов к выполнению критического участка, то ready[1] == 0, и процесс P0 может осуществить вход. Если процесс P1 готов к выполнению критического участка, то ready[1] == 1 и переменная turn имеет значение 0 либо 1, позволяя процессу P0 либо процессу P1 начать выполнение критической секции. Если процесс P1 завершил выполнение критического участка, то он сбросит свой флаг готовности ready[1] == 0, разрешая процессу P0 приступить к выполнению критической работы. Таким образом, условие прогресса выполняется.

Отсюда же вытекает выполнение условия ограниченного ожидания. Так как в процессе ожидания разрешения на вход процесс P0 не изменяет значения переменных, он сможет начать исполнение своего критического участка после не более чем одного прохода по критической секции процесса P1.

**Алгоритм булочной (Bakery algorithm)**

Алгоритм Петерсона дает нам решение задачи корректной организации взаимодействия двух процессов. Давайте рассмотрим теперь соответствующий алгоритм для n взаимодействующих процессов, который получил название алгоритм булочной, хотя применительно к нашим условиям его следовало бы скорее назвать алгоритм регистратуры в поликлинике. Основная его идея выглядит так. Каждый вновь прибывающий клиент (он же процесс) получает талончик на обслуживание с номером. Клиент с наименьшим номером на талончике обслуживается следующим. К сожалению, из-за неатомарности операции вычисления следующего номера алгоритм булочной не гарантирует, что у всех процессов будут талончики с разными номерами. В случае равенства номеров на талончиках у двух или более клиентов первым обслуживается клиент с меньшим значением имени (имена можно сравнивать в лексикографическом порядке). Разделяемые структуры данных для алгоритма – это два массива

shared enum {false, true} choosing[n];

shared int number[n];

Изначально элементы этих массивов инициируются значениями false и 0 соответственно. Введем следующие обозначения

(a,b) < (c,d), если a < c

или если a == c и b < d

max(a0, a1, ...., an) – это число k такое, что

k >= ai для всех i = 0, ...,n

Структура процесса Pi для алгоритма булочной приведена ниже

while (some condition) {

choosing[i] = true;

number[i] = max(number[0], ...,

number[n-1]) + 1;

choosing[i] = false;

for(j = 0; j < n; j++){

while(choosing[j]);

while(number[j] != 0 && (number[j],j) <

(number[i],i));

}

critical section

number[i] = 0;

remainder section

}

Доказательство того, что этот алгоритм удовлетворяет условиям 1 – 5, выполните самостоятельно в качестве упражнения.

**Аппаратная поддержка взаимоисключений**

Наличие аппаратной поддержки взаимоисключений позволяет упростить алгоритмы и повысить их эффективность точно так же, как это происходит и в других областях программирования. Мы уже обращались к общепринятому hardware для решения задачи реализации взаимоисключений, когда говорили об использовании механизма запрета/разрешения прерываний.

Многие вычислительные системы помимо этого имеют специальные команды процессора, которые позволяют проверить и изменить значение машинного слова или поменять местами значения двух машинных слов в памяти, выполняя эти действия как атомарные операции. Давайте обсудим, как концепции таких команд могут использоваться для реализации взаимоисключений.

1. **Механизмы синхронизации.**

Описаны в вопросе №3, а именно это:запрет прерываний, переменная-замок, Строгое чередование, флаги готовности, алгоритм Петерсона, алгоритм булочно).

**5. Концепция семафора.**

Одним из первых механизмов, предложенных для синхронизации поведения процессов, стали семафоры, концепцию которых описал Дейкстра (Dijkstra) в 1965 году.

Семафор представляет собой целую переменную, принимающую неотрицательные значения, доступ любого процесса к которой, за исключением момента ее инициализации, может осуществляться только через две атомарные операции: P (от датского слова proberen – проверять) и V (от verhogen – увеличивать). Классическое определение этих операций выглядит следующим образом:

P(S): пока S == 0 процесс блокируется;

S = S – 1;

V(S): S = S + 1;

Эта запись означает следующее: при выполнении операции P над семафором S сначала проверяется его значение. Если оно больше 0, то из S вычитается 1. Если оно меньше или равно 0, то процесс блокируется до тех пор, пока S не станет больше 0, после чего из S вычитается 1. При выполнении операции V над семафором S к его значению просто прибавляется 1. В момент создания семафор может быть инициализирован любым неотрицательным значением.

Подобные переменные-семафоры могут с успехом применяться для решения различных задач организации взаимодействия процессов. В ряде языков программирования они были непосредственно введены в синтаксис языка (например, в ALGOL-68), в других случаях реализуются с помощью специальных системных вызовов. Соответствующая целая переменная располагается внутри адресного пространства ядра операционной системы. Операционная система обеспечивает атомарность операций P и V, используя, например, метод запрета прерываний на время выполнения соответствующих системных вызовов. Если при выполнении операции P заблокированными оказались несколько процессов, то порядок их разблокирования может быть произвольным, например, FIFO.

**6. Мониторы.**

Хотя решение задачи producer-consumer с помощью семафоров выглядит достаточно изящно, программирование с их использованием требует повышенной осторожности и внимания, чем отчасти напоминает программирование на языке Ассемблера. Допустим, что в рассмотренном примере мы случайно поменяли местами операции P, сначала выполнив операцию для семафора mutex, а уже затем для семафоров full и empty. Допустим теперь, что потребитель, войдя в свой критический участок ( mutex сброшен), обнаруживает, что буфер пуст. Он блокируется и начинает ждать появления сообщений. Но производитель не может войти в критический участок для передачи информации, так как тот заблокирован потребителем. Получаем тупиковую ситуацию.

В сложных программах произвести анализ правильности использования семафоров с карандашом в руках становится очень непросто. В то же время обычные способы отладки программ зачастую не дают результата, поскольку возникновение ошибок зависит от interleaving атомарных операций, и ошибки могут быть трудновоспроизводимы. Для того чтобы облегчить работу программистов, в 1974 году Хором (Hoare) был предложен механизм еще более высокого уровня, чем семафоры, получивший название мониторов. Мы с вами рассмотрим конструкцию, несколько отличающуюся от оригинальной.

Мониторы представляют собой тип данных, который может быть с успехом внедрен в объектно-ориентированные языки программирования. Монитор обладает собственными переменными, определяющими его состояние. Значения этих переменных извне могут быть изменены только с помощью вызова функций-методов, принадлежащих монитору. В свою очередь, эти функции-методы могут использовать в работе только данные, находящиеся внутри монитора, и свои параметры. На абстрактном уровне можно описать структуру монитора следующим образом:

monitor monitor\_name {

описание внутренних переменных ;

void m1(...){...

}

void m2(...){...

}

...

void mn(...){...

}

{

блок инициализации

внутренних переменных;

}

}

Здесь функции m1,..., mn представляют собой функции-методы монитора, а 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. Поскольку мониторы представляют собой особые конструкции языка программирования, компилятор может отличить вызов функции, принадлежащей монитору, от вызовов других функций и обработать его специальным образом, добавив к нему пролог и эпилог, реализующий взаимоисключение. Так как обязанность конструирования механизма взаимоисключений возложена на компилятор, а не на программиста, работа программиста при использовании мониторов существенно упрощается, а вероятность возникновения ошибок становится меньше.

Однако одних только взаимоисключений недостаточно для того, чтобы в полном объеме реализовать решение задач, возникающих при взаимодействии процессов. Нам нужны еще и средства организации очередности процессов, подобно семафорам full и empty в предыдущем примере. Для этого в мониторах было введено понятие условных переменных (condition variables)1), над которыми можно совершать две операции wait и signal, отчасти похожие на операции P и V над семафорами.

Если функция монитора не может выполняться дальше, пока не наступит некоторое событие, она выполняет операцию wait над какой-либо условной переменной. При этом процесс, выполнивший операцию wait, блокируется, становится неактивным, и другой процесс получает возможность войти в монитор.

Когда ожидаемое событие происходит, другой процесс внутри функции-метода совершает операцию signal над той же самой условной переменной. Это приводит к пробуждению ранее заблокированного процесса, и он становится активным. Если несколько процессов дожидались операции signal для этой переменной, то активным становится только один из них. Что можно предпринять для того, чтобы у нас не оказалось двух процессов, разбудившего и пробужденного, одновременно активных внутри монитора? Хор предложил, чтобы пробужденный процесс подавлял исполнение разбудившего процесса, пока он сам не покинет монитор. Несколько позже Хансен (Hansen) предложил другой механизм: разбудивший процесс покидает монитор немедленно после исполнения операции signal. Мы будем придерживаться подхода Хансена.

Необходимо отметить, что условные переменные, в отличие от семафоров Дейкстры, не умеют запоминать предысторию. Это означает, что операция signal всегда должна выполняться после операции wait. Если операция signal выполняется над условной переменной, с которой не связано ни одного заблокированного процесса, то информация о произошедшем событии будет утеряна. Следовательно, выполнение операции wait всегда будет приводить к блокированию процесса.

Реализация мониторов требует разработки специальных языков программирования и компиляторов для них. Мониторы встречаются в таких языках, как параллельный Евклид, параллельный Паскаль, Java и т. д. Эмуляция мониторов с помощью системных вызовов для обычных широко используемых языков программирования не так проста, как эмуляция семафоров. Поэтому можно пользоваться еще одним механизмом со скрытыми взаимоисключениями, механизмом, о котором мы уже упоминали, – передачей сообщений.

**7. Очереди сообщений.**

Для прямой и непрямой адресации достаточно двух примитивов, чтобы описать передачу сообщений по линии связи – send и receive. В случае прямой адресации мы будем обозначать их так:

send(P, message) – послать сообщение message процессу P ;

receive(Q, message) – получить сообщение message от процесса Q.

В случае непрямой адресации мы будем обозначать их так:

send(A, message) – послать сообщение message в почтовый ящик A ;

receive(A, message) – получить сообщение message из почтового ящика A.

Примитивы send и receive уже имеют скрытый от наших глаз механизм взаимоисключения. Более того, в большинстве систем они уже имеют и скрытый механизм блокировки при чтении из пустого буфера и при записи в полностью заполненный буфер. Реализация решения задачи producer-consumer для таких примитивов становится неприлично тривиальной. Надо отметить, что, несмотря на простоту использования, передача сообщений в пределах одного компьютера происходит существенно медленнее, чем работа с семафорами и мониторами.

**Контрольные вопросы:**

1. Охарактеризуйте алгоритмы синхронизации процессов.

2. Раскройте понятия состязания, взаимоисключения, чередования, критической секции.

3. Укажите программные и аппаратные алгоритмы взаимодействия процессов.

4. Охарактеризуйте механизмы синхронизации.

5. Охарактеризуйте концепция семафора.

6. Раскройте понятие мониторов.

7. Раскройте понятие очереди сообщений.

**Литература:**

1. А.П. Побегайло, Системное программирование в Windows, БХВ-Петербург, 2006, 1056 с.

2. Ахо, Лам, Сети, Ульман - Компиляторы. Принципы, технологии, инструменты, Изд.: Вильямс, 2008, 1185 с.

3. Л.Бек, Введение в системное программирование, М., Мир, 2008г.

4. Р. И. Компаниец, Системное программирование : Основы построения трансляторов, Изд.: СПб., 2000.

5. Ф.Льюис, Д.Розенкранц, Р.Стирнз. Теоретические основы проектирования компиляторов. - М.: Мир, 20099, 655 с.

6. Э. Таненбау, Современные операционные системы, Изд.: Питер, 2011, 1120 с.

7. А. В. Гордеев, А. Ю. Молчанов, Системное программное обеспечение, учебник для ВУЗов, СПб.: Питер, 2010 г.

**СЕМИНАР №13: «ОРГАНИЗАЦИЯ ПАМЯТИ».**

**План семинара:**

* 1. Организация памяти.
  2. Простейшие способы управления памятью.
  3. Иерархия памяти.
  4. Адресное пространство.
  5. Сегменты памяти.
  6. Связывание адресов.
  7. Страничная память.

**Содержание по плану:**

1. **Организация памяти.**

**Физическая организация памяти компьютера**

Запоминающие устройства компьютера разделяют, как минимум, на два уровня: основную (главную, оперативную, физическую ) и вторичную (внешнюю) память.

Основная память представляет собой упорядоченный массив однобайтовых ячеек, каждая из которых имеет свой уникальный адрес (номер). Процессор извлекает команду из основной памяти, декодирует и выполняет ее. Для выполнения команды могут потребоваться обращения еще к нескольким ячейкам основной памяти. Обычно основная память изготавливается с применением полупроводниковых технологий и теряет свое содержимое при отключении питания.

Вторичную память (это главным образом диски) также можно рассматривать как одномерное линейное адресное пространство, состоящее из последовательности байтов. В отличие от оперативной памяти, она является энергонезависимой, имеет существенно большую емкость и используется в качестве расширения основной памяти.

**Локальность**

Оказывается, при таком способе организации по мере снижения скорости доступа к уровню памяти снижается также и частота обращений к нему.

Ключевую роль здесь играет свойство реальных программ, в течение ограниченного отрезка времени способных работать с небольшим набором адресов памяти. Это эмпирически наблюдаемое свойство известно как принцип локальности или локализации обращений.

Свойство локальности (соседние в пространстве и времени объекты характеризуются похожими свойствами) присуще не только функционированию ОС, но и природе вообще. В случае ОС свойство локальности объяснимо, если учесть, как пишутся программы и как хранятся данные, то есть обычно в течение какого-то отрезка времени ограниченный фрагмент кода работает с ограниченным набором данных. Эту часть кода и данных удается разместить в памяти с быстрым доступом. В результате реальное время доступа к памяти определяется временем доступа к верхним уровням, что и обусловливает эффективность использования иерархической схемы. Надо сказать, что описываемая организация вычислительной системы во многом имитирует деятельность человеческого мозга при переработке информации. Действительно, решая конкретную проблему, человек работает с небольшим объемом информации, храня не относящиеся к делу сведения в своей памяти или во внешней памяти (например, в книгах).

Кэш процессора обычно является частью аппаратуры, поэтому менеджер памяти ОС занимается распределением информации главным образом в основной и внешней памяти компьютера. В некоторых схемах потоки между оперативной и внешней памятью регулируются программистом (см. например, далее оверлейные структуры ), однако это связано с затратами времени программиста, так что подобную деятельность стараются возложить на ОС.

Адреса в основной памяти, характеризующие реальное расположение данных в физической памяти, называются физическими адресами. Набор физических адресов, с которым работает программа, называют физическим адресным пространством.

**Логическая память**

Аппаратная организация памяти в виде линейного набора ячеек не соответствует представлениям программиста о том, как организовано хранение программ и данных. Большинство программ представляет собой набор модулей, созданных независимо друг от друга. Иногда все модули, входящие в состав процесса, располагаются в памяти один за другим, образуя линейное пространство адресов. Однако чаще модули помещаются в разные области памяти и используются по-разному.

Схема управления памятью, поддерживающая этот взгляд пользователя на то, как хранятся программы и данные, называется сегментацией. Сегмент – область памяти определенного назначения, внутри которой поддерживается линейная адресация. Сегменты содержат процедуры, массивы, стек или скалярные величины, но обычно не содержат информацию смешанного типа.

По-видимому, вначале сегменты памяти появились в связи с необходимостью обобществления процессами фрагментов программного кода (текстовый редактор, тригонометрические библиотеки и т. д.), без чего каждый процесс должен был хранить в своем адресном пространстве дублирующую информацию. Эти отдельные участки памяти, хранящие информацию, которую система отображает в память нескольких процессов, получили название сегментов. Память, таким образом, перестала быть линейной и превратилась в двумерную. Адрес состоит из двух компонентов: номер сегмента, смещение внутри сегмента. Далее оказалось удобным размещать в разных сегментах различные компоненты процесса (код программы, данные, стек и т. д.). Попутно выяснилось, что можно контролировать характер работы с конкретным сегментом, приписав ему атрибуты, например права доступа или типы операций, которые разрешается производить с данными, хранящимися в сегменте.

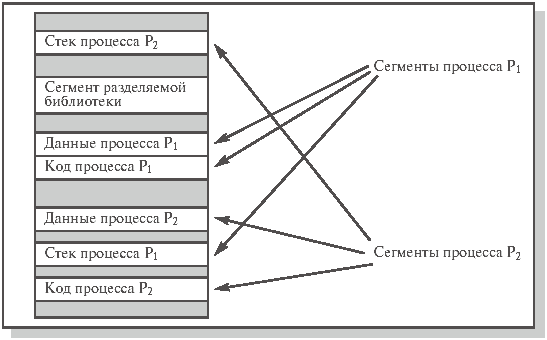


Рис. 8.2. Расположение сегментов процессов в памяти компьютера

Некоторые сегменты, описывающие адресное пространство процесса, показаны на рис. 8.2. Более подробная информация о типах сегментов имеется в лекции 10.

Большинство современных ОС поддерживают сегментную организацию памяти. В некоторых архитектурах (Intel, например) сегментация поддерживается оборудованием.

Адреса, к которым обращается процесс, таким образом, отличаются от адресов, реально существующих в оперативной памяти. В каждом конкретном случае используемые программой адреса могут быть представлены различными способами. Например, адреса в исходных текстах обычно символические. Компилятор связывает эти символические адреса с перемещаемыми адресами (такими, как n байт от начала модуля). Подобный адрес, сгенерированный программой, обычно называют логическим (в системах с виртуальной памятью он часто называется виртуальным) адресом. Совокупность всех логических адресов называется логическим (виртуальным) адресным пространством.

1. **Простейшие способы управления памятью.**

Первые ОС применяли очень простые методы управления памятью. Вначале каждый процесс пользователя должен был полностью поместиться в основной памяти, занимать непрерывную область памяти, а система принимала к обслуживанию дополнительные пользовательские процессы до тех пор, пока все они одновременно помещались в основной памяти. Затем появился "простой свопинг" (система по-прежнему размещает каждый процесс в основной памяти целиком, но иногда на основании некоторого критерия целиком сбрасывает образ некоторого процесса из основной памяти во внешнюю и заменяет его в основной памяти образом другого процесса). Такого рода схемы имеют не только историческую ценность. В настоящее время они применяются в учебных и научно-исследовательских модельных ОС, а также в ОС для встроенных (embedded) компьютеров.

**Схема с фиксированными разделами**

Самым простым способом управления оперативной памятью является ее предварительное (обычно на этапе генерации или в момент загрузки системы) разбиение на несколько разделов фиксированной величины. Поступающие процессы помещаются в тот или иной раздел. При этом происходит условное разбиение физического адресного пространства. Связывание логических и физических адресов процесса происходит на этапе его загрузки в конкретный раздел, иногда – на этапе компиляции.

Каждый раздел может иметь свою очередь процессов, а может существовать и глобальная очередь для всех разделов(см. рис. 1).

Эта схема была реализована в IBM OS/360 (MFT), DEC RSX-11 и ряде других систем.

Подсистема управления памятью оценивает размер поступившего процесса, выбирает подходящий для него раздел, осуществляет загрузку процесса в этот раздел и настройку адресов.

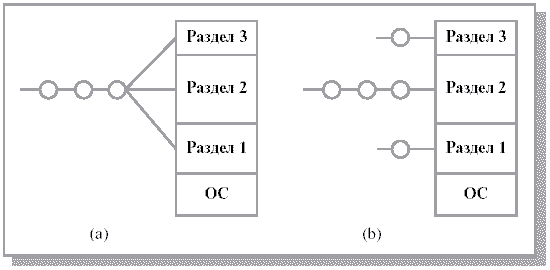


Рис. 1. Схема с фиксированными разделами: (a) – с общей очередью процессов, (b) – с отдельными очередями процессов

Очевидный недостаток этой схемы – число одновременно выполняемых процессов ограничено числом разделов.

Другим существенным недостатком является то, что предлагаемая схема сильно страдает от внутренней фрагментации – потери части памяти, выделенной процессу, но не используемой им. Фрагментация возникает потому, что процесс не полностью занимает выделенный ему раздел или потому, что некоторые разделы слишком малы для выполняемых пользовательских программ.

**Один процесс в памяти**

Частный случай схемы с фиксированными разделами – работа менеджера памяти однозадачной ОС. В памяти размещается один пользовательский процесс. Остается определить, где располагается пользовательская программа по отношению к ОС – в верхней части памяти, в нижней или в средней. Причем часть ОС может быть в ROM (например, BIOS, драйверы устройств). Главный фактор, влияющий на это решение, – расположение вектора прерываний, который обычно локализован в нижней части памяти, поэтому ОС также размещают в нижней. Примером такой организации может служить ОС MS-DOS.

Защита адресного пространства ОС от пользовательской программы может быть организована при помощи одного граничного регистра, содержащего адрес границы ОС.

**Оверлейная структура**

Так как размер логического адресного пространства процесса может быть больше, чем размер выделенного ему раздела (или больше, чем размер самого большого раздела), иногда используется техника, называемая оверлей (overlay) или организация структуры с перекрытием. Основная идея – держать в памяти только те инструкции программы, которые нужны в данный момент.

Потребность в таком способе загрузки появляется, если логическое адресное пространство системы мало, например 1 Мбайт (MS-DOS) или даже всего 64 Кбайта (PDP-11), а программа относительно велика. На современных 32-разрядных системах, где виртуальное адресное пространство измеряется гигабайтами, проблемы с нехваткой памяти решаются другими способами.

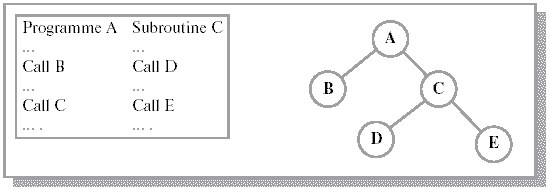


Рис. 2. Организация структуры с перекрытием. Можно поочередно загружать в память ветви A-B, A-C-D и A-C-E программы

Коды ветвей оверлейной структуры программы находятся на диске как абсолютные образы памяти и считываются драйвером оверлеев при необходимости. Для описания оверлейной структуры обычно используется специальный несложный язык (overlay description language). Совокупность файлов исполняемой программы дополняется файлом (обычно с расширением .odl), описывающим дерево вызовов внутри программы. Для примера, приведенного на рис. 2, текст этого файла может выглядеть так:

A-(B,C)

C-(D,E)

Синтаксис подобного файла может распознаваться загрузчиком. Привязка к физической памяти происходит в момент очередной загрузки одной из ветвей программы.

Оверлеи могут быть полностью реализованы на пользовательском уровне в системах с простой файловой структурой. ОС при этом лишь делает несколько больше операций ввода-вывода. Типовое решение – порождение линкером специальных команд, которые включают загрузчик каждый раз, когда требуется обращение к одной из перекрывающихся ветвей программы.

Тщательное проектирование оверлейной структуры отнимает много времени и требует знания устройства программы, ее кода, данных и языка описания оверлейной структуры. По этой причине применение оверлеев ограничено компьютерами с небольшим логическим адресным пространством. Как мы увидим в дальнейшем, проблема оверлейных сегментов, контролируемых программистом, отпадает благодаря появлению систем виртуальной памяти.

Заметим, что возможность организации структур с перекрытиями во многом обусловлена свойством локальности, которое позволяет хранить в памяти только ту информацию, которая необходима в конкретный момент вычислений.

**Динамическое распределение. Свопинг**

Имея дело с пакетными системами, можно обходиться фиксированными разделами и не использовать ничего более сложного. В системах с разделением времени возможна ситуация, когда память не в состоянии содержать все пользовательские процессы. Приходится прибегать к свопингу (swapping) – перемещению процессов из главной памяти на диск и обратно целиком. Частичная выгрузка процессов на диск осуществляется в системах со страничной организацией (paging) и будет рассмотрена ниже.

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.

Свопинг не имеет непосредственного отношения к управлению памятью, скорее он связан с подсистемой планирования процессов. Очевидно, что свопинг увеличивает время переключения контекста. Время выгрузки может быть сокращено за счет организации специально отведенного пространства на диске (раздел для свопинга). Обмен с диском при этом осуществляется блоками большего размера, то есть быстрее, чем через стандартную файловую систему. Во многих версиях Unix свопинг начинает работать только тогда, когда возникает необходимость в снижении загрузки системы.

**Схема с переменными разделами**

В принципе, система свопинга может базироваться на фиксированных разделах. Более эффективной, однако, представляется схема динамического распределения или схема с переменными разделами, которая может использоваться и в тех случаях, когда все процессы целиком помещаются в памяти, то есть в отсутствие свопинга. В этом случае вначале вся память свободна и не разделена заранее на разделы. Вновь поступающей задаче выделяется строго необходимое количество памяти, не более. После выгрузки процесса память временно освобождается. По истечении некоторого времени память представляет собой переменное число разделов разного размера (рис. 3). Смежные свободные участки могут быть объединены.

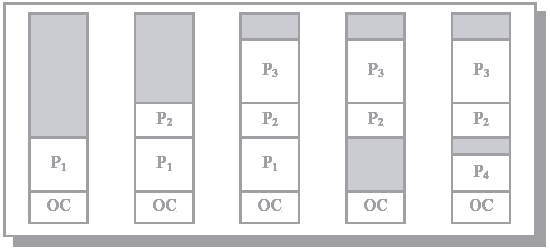


Рис. 3. Динамика распределения памяти между процессами (серым цветом показана неиспользуемая память)

В какой раздел помещать процесс? Наиболее распространены три стратегии.

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.

Моделирование показало, что доля полезно используемой памяти в первых двух случаях больше, при этом первый способ несколько быстрее. Попутно заметим, что перечисленные стратегии широко применяются и другими компонентами ОС, например для размещения файлов на диске.

Типовой цикл работы менеджера памяти состоит в анализе запроса на выделение свободного участка (раздела), выборе его среди имеющихся в соответствии с одной из стратегий (первого подходящего, наиболее подходящего и наименее подходящего), загрузке процесса в выбранный раздел и последующих изменениях таблиц свободных и занятых областей. Аналогичная корректировка необходима и после завершения процесса. Связывание адресов может осуществляться на этапах загрузки и выполнения.

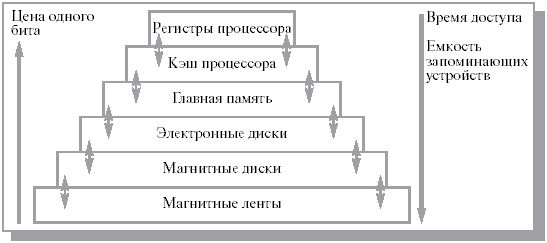
Этот метод более гибок по сравнению с методом фиксированных разделов, однако ему присуща внешняя фрагментация – наличие большого числа участков неиспользуемой памяти, не выделенной ни одному процессу. Выбор стратегии размещения процесса между первым подходящим и наиболее подходящим слабо влияет на величину фрагментации. Любопытно, что метод наиболее подходящего может оказаться наихудшим, так как он оставляет множество мелких незанятых блоков.

Статистический анализ показывает, что пропадает в среднем 1/3 памяти! Это известное правило 50% (два соседних свободных участка в отличие от двух соседних процессов могут быть объединены).

Одно из решений проблемы внешней фрагментации – организовать сжатие, то есть перемещение всех занятых (свободных) участков в сторону возрастания (убывания) адресов, так, чтобы вся свободная память образовала непрерывную область. Этот метод иногда называют схемой с перемещаемыми разделами. В идеале фрагментация после сжатия должна отсутствовать. Сжатие, однако, является дорогостоящей процедурой, алгоритм выбора оптимальной стратегии сжатия очень труден и, как правило, сжатие осуществляется в комбинации с выгрузкой и загрузкой по другим адресам.

1. **Иерархия памяти.**

Разновидности памяти могут быть объединены в иерархию по убыванию времени доступа, возрастанию цены и увеличению емкости.



Многоуровневую схему используют следующим образом. Информация, которая находится в памяти верхнего уровня, обычно хранится также на уровнях с большими номерами. Если процессор не обнаруживает нужную информацию на i-м уровне, он начинает искать ее на следующих уровнях. Когда нужная информация найдена, она переносится в более быстрые уровни.

1. **Адресное пространство.**

Чтобы допустить одновременное размещение в памяти нескольких приложений без создания взаимных помех, нужно решить две проблемы, относящиеся к защите и перемещению. Примитивное решение первой из этих проблем мы уже рассматривали на примере IBM 360: участки памяти помечались защитным ключом, и ключ выполняемого процесса сличался с ключом каждого выбранного слова памяти. Но этот подход не решал второй проблемы, хотя она могла быть решена путем перемещения программ в процессе их загрузки, но это было слишком медленным и сложным решением.

Более подходящее решение — придумать для памяти новую абстракцию: адресное пространство. Так же как понятие процесса создает своеобразный абстрактный центральный процессор для запуска программ, понятие адресного пространства создает своеобразную абстрактную память, в которой существуют программы. Адресное пространство — это набор адресов, который может быть использован процессом для обращения к памяти. У каждого процесса имеется свое собственное адресное пространство, независимое от того адресного пространства, которое принадлежит другим процессам (за исключением тех особых обстоятельств, при которых процессам требуется совместное использование их адресных пространств).

Понятие адресного пространства имеет весьма универсальный характер и появляется во множестве контекстов. Возьмем телефонные номера. В США и многих других странах местный телефонный номер состоит обычно из семизначного номера. Поэтому адресное пространство телефонных номеров простирается от 0000000 до 9999999, хотя некоторые номера, к примеру, те, что начинаются с 000, не используются. С ростом количества сотовых телефонов, модемов и факсов это пространство стало слишком тесным, а в этом случае необходимо использовать больше цифр. Адресное пространство портов ввода-вывода процессора Pentium простирается от 0 до 16 383. Протокол IPv4 обращается к 32-разрядным номерам, поэтому его адресное пространство простирается от 0 до 232 - 1 (опять-таки с некоторым количеством зарезервированных номеров).

Адресное пространство не обязательно должно быть числовым. Набор интернет-доменов .com также является адресным пространством. Это адресное пространство состоит из всех строк длиной от 2 до 63 символов, которые могут быть составлены из букв, цифр и дефисов, за которыми следует название домена — .com. Теперь вам должна стать понятной сама идея, в которой нет ничего сложного.

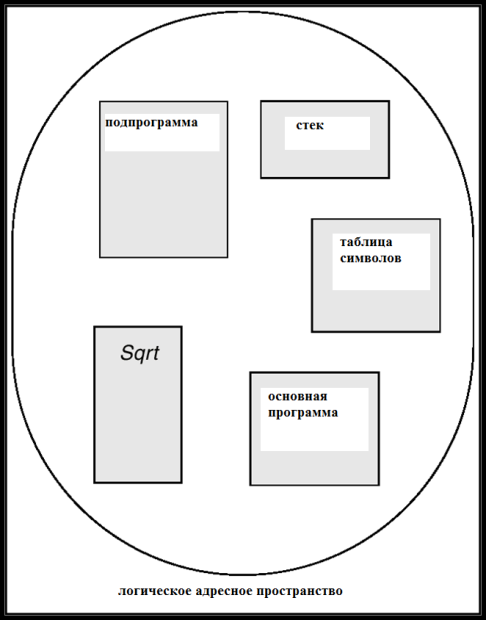
Немного сложнее понять, как каждой программе можно выделить свое собственное адресное пространство, поскольку адрес 28 в одной программе означает иное физическое место, чем адрес 28 в другой программе. Далее мы рассмотрим простой способ, который ранее был распространен, но вышел из употребления с появлением возможностей размещения на современных центральных процессорах более сложных (и более совершенных) схем.

1. **Сегменты памяти.**

Сегментная организация памяти (segmentation) - схема распределения памяти в виде сегментов переменной длины, соответствующая пользовательской трактовке распределения памяти, т.е. логической структуре программ и данных. С точки зрения пользователя (разработчика программы), программа – это набор модулей кода и данных, каждому из которых должен соответствовать свой сегмент в памяти. Сегмент – логическая единица распределения памяти, предназначенная для размещения в памяти одного модуля программного кода или данных. Например, в виде сегментов памяти могут быть представлены:

* основная программа;
* процедура;
* функция;
* метод;
* объект;
* набор локальных переменных;
* набор глобальных переменных;
* общий блок данных (например, COMMON-блок в языке FORTRAN);
* стек;
* таблица символов;
* массив.

Программа как набор сегментов представлена на рисунке.



1. **Связывание адресов.**

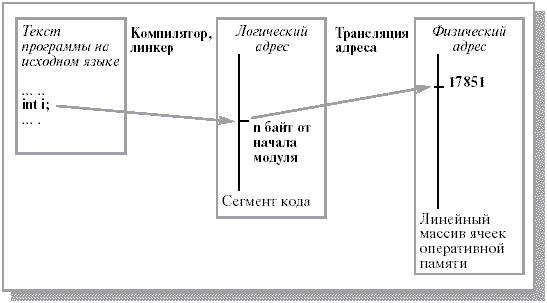
Логические и физические адресные пространства ни по организации, ни по размеру не соответствуют друг другу. Максимальный размер логического адресного пространства обычно определяется разрядностью процессора (например, 232) и в современных системах значительно превышает размер физического адресного пространства. Следовательно, процессор и ОС должны быть способны отобразить ссылки в коде программы в реальные физические адреса, соответствующие текущему расположению программы в основной памяти. Такое отображение адресов называют трансляцией (привязкой) адреса или связыванием адресов (см. рис.).

Связывание логического адреса, порожденного оператором программы, сфизическим должно быть осуществлено до начала выполнения оператора или в момент его выполнения. Таким образом, привязка инструкций и данных к памяти в принципе может быть сделана на следующих шагах.

Этап компиляции (Compiletime). Когда на стадии компиляции известно точное место размещения процесса в памяти, тогда непосредственно генерируются физические адреса. При изменении стартового адреса программы необходимо перекомпилировать ее код. В качестве примера можно привести .com программы MS-DOS, которые связывают ее с физическими адресами на стадии компиляции.

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

Этап выполнения (Executiontime). Если процесс может быть перемещен во время выполнения из одной области памяти в другую, связывание откладывается до стадии выполнения. Здесь желательно наличие специализированного оборудования, например регистров перемещения. Их значение прибавляется к каждому адресу, сгенерированному процессом. Большинство современных ОС осуществляет трансляцию адресов на этапе выполнения, используя для этого специальный аппаратный механизм.



**7. Страничная память.**

Описанные выше схемы недостаточно эффективно используют память, поэтому в современных схемах управления памятью не принято размещать процесс в оперативной памяти одним непрерывным блоком.

В самом простом и наиболее распространенном случае страничной организации памяти (или paging) как логическое адресное пространство, так и физическое представляются состоящими из наборов блоков или страниц одинакового размера. При этом образуются логические страницы (page), а соответствующие единицы в физической памяти называют физическими страницами или страничными кадрами (page frames). Страницы (и страничные кадры) имеют фиксированную длину, обычно являющуюся степенью числа 2, и не могут перекрываться. Каждый кадр содержит одну страницу данных. При такой организации внешняя фрагментация отсутствует, а потери из-за внутренней фрагментации, поскольку процесс занимает целое число страниц, ограничены частью последней страницы процесса.

Логический адрес в страничной системе – упорядоченная пара (p,d), где p – номер страницы в виртуальной памяти, а d – смещение в рамках страницы p, на которой размещается адресуемый элемент. Заметим, что разбиение адресного пространства на страницы осуществляется вычислительной системой незаметно для программиста. Поэтому адрес является двумерным лишь с точки зрения операционной системы, а с точки зрения программиста адресное пространство процесса остается линейным.

Описываемая схема позволяет загрузить процесс, даже если нет непрерывной области кадров, достаточной для размещения процесса целиком. Но одного базового регистра для осуществления трансляции адреса в данной схеме недостаточно. Система отображения логических адресов в физические сводится к системе отображения логических страниц в физические и представляет собой таблицу страниц, которая хранится в оперативной памяти. Иногда говорят, что таблица страниц – это кусочно-линейная функция отображения, заданная в табличном виде.

Интерпретация логического адреса показана на рис. Если выполняемый процесс обращается к логическому адресу v = (p,d), механизм отображения ищет номер страницы p в таблице страниц и определяет, что эта страница находится в страничном кадре p', формируя реальный адрес из p' и d.

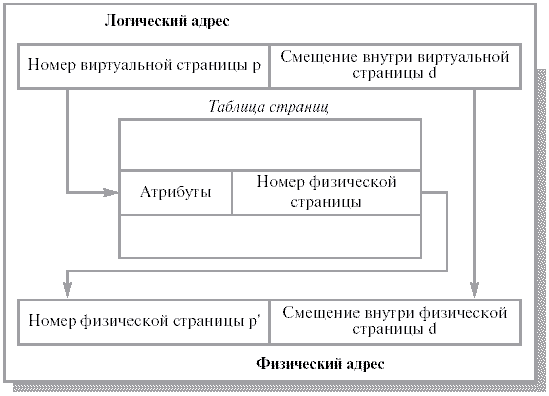


Таблица страниц (page table) адресуется при помощи специального регистра процессора и позволяет определить номер кадра по логическому адресу. Помимо этой основной задачи, при помощи атрибутов, записанных в строке таблицы страниц, можно организовать контроль доступа к конкретной странице и ее защиту.

Отметим еще раз различие точек зрения пользователя и системы на используемую память. С точки зрения пользователя, его память – единое непрерывное пространство, содержащее только одну программу. Реальное отображение скрыто от пользователя и контролируется ОС. Заметим, что процессу пользователя чужая память недоступна. Он не имеет возможности адресовать память за пределами своей таблицы страниц, которая включает только его собственные страницы.

Для управления физической памятью ОС поддерживает структуру таблицы кадров. Она имеет одну запись на каждый физический кадр, показывающий его состояние.

Отображение адресов должно быть осуществлено корректно даже в сложных случаях и обычно реализуется аппаратно. Для ссылки на таблицу процессов используется специальный регистр. При переключении процессов необходимо найти таблицу страниц нового процесса, указатель на которую входит в контекст процесса.

**Контрльные вопросы:**

* 1. Опишите организацию памяти.
  2. Перечислите способы управления памятью.
  3. Раскройте понятие иерархия памяти.
  4. Раскройте понятие адресное пространство.
  5. Раскройте понятие сегменты памяти.
  6. Раскройте понятие связывание адресов.
  7. Охарактеризуйте страничную память.

**Литература:**

1. А.П. Побегайло, Системное программирование в Windows, БХВ-Петербург, 2006, 1056 с.

2. Ахо, Лам, Сети, Ульман - Компиляторы. Принципы, технологии, инструменты, Изд.: Вильямс, 2008, 1185 с.

3. Л.Бек, Введение в системное программирование, М., Мир, 2008г.

4. Р. И. Компаниец, Системное программирование : Основы построения трансляторов, Изд.: СПб., 2000.

5. Ф.Льюис, Д.Розенкранц, Р.Стирнз. Теоретические основы проектирования компиляторов. - М.: Мир, 20099, 655 с.

6. Э. Таненбау, Современные операционные системы, Изд.: Питер, 2011, 1120 с.

7. А. В. Гордеев, А. Ю. Молчанов, Системное программное обеспечение, учебник для ВУЗов, СПб.: Питер, 2010 г.

**СЕМИНАР №14: «ВИРТУАЛЬНАЯ ПАМЯТЬ».**

**План семинара:**

1. Понятие виртуальной памяти.
2. Архитектурные средства поддержки ВП.
3. Страничная и сегментно-страничная ВП.
4. Ассоциативная память.
5. Исключительные ситуации при работе с памятью.
6. Стратегии управления страничной памятью.

**Содержание по плану:**

1. **Понятие виртуальной памяти.**

Виртуальная память: что это такое и для чего это нужно? Дело в том, что с развитием и появлением новых компьютерных технологий, машины, несомненно, преобразились в лучшую сторону: в мире профессиональных программистов уже не существует понятия "ОЗУ на ферритовых сердечниках" или "накопителей на магнитных лентах". Что и говорить, с изобретением персонального компьютера, даже простой непрофессиональный пользователь получил возможность использовать ПК для собственных целей и нужд.

Вместе с появлением компьютеров нового поколения у пользователя появилась тенденция "оседлать" быструю машину заставить её делать как можно больше. Как экономить машинное время и в то же время производить больше? Ответ на этот вопрос был найден посредством организации мультипрограммной работы ЭВМ. Этот метод был признан очень удобным, так как при организации мультипрограммного цикла:

1. Машина не простаивала зря: при одновременном выполнении нескольких программ и команд в работе процессора появлялась новая функция анализ и распределение машинного времени, отведённого на выполнение каждой программы;

2. За каждый машинный такт выполнялась одна или несколько команд.

Действительно, метод мультипрограммной работы оказался потрясающе эффективным, но здесь мы сталкиваемся с проблемой нехватки оперативной памяти. Хорошо видно, что внедрение более рационального решения сталкивается с единственной проблемой памяти. Но можно ли каким-то образом решить эту проблему? Именно на этом этапе на помощь пользователю приходит виртуальная память, которая позволяет модифицировать ресурсы памяти, сделать объём оперативной памяти намного больше, для того чтобы пользователь, поместив туда как можно больше программ, реально сэкономил время и повысил эффективность своего труда. "Открытие" виртуальной памяти (далее ВП) внесло огромную контрибуцию в развитие современных технологий, облегчило работу, как профессионального программиста, так и обычного пользователя, обеспечивая процесс более эффективного решения задач на ЭВМ. Но как же построена ВП? Дело в том, что при работе машины, использующей виртуальную память, обязательно используется Внешнее Запоминающее Устройство (ВЗУ), которое обычно представляет собой накопитель на гибком магнитном диске или жестком диске типа "винчестер". (Последнее устройство используется чаще).

Действительно, использование виртуальной памяти обязательно подразумевает обращение к диску, так как при разработке и внедрению систем с таким методом организации памяти, было представлено, что ячейки оперативной памяти, и памяти на диске будут представлять собой единое целое. По своей сути ВП не такая уж сложная структура напротив, недостаток оперативной памяти компенсируется наличием свободного дискового пространства, которое задействовано в роли ОП. Необходимо понимать, что часть программ, которые мы не смогли разместить в ОП из-за её нехватки, теперь будут размещены на диске и это будет эквивалентно размещению в оперативной памяти. Виртуальная память представляет собой совокупность всех ячеек памяти оперативной и внешней, имеющих сквозную нумерацию от нуля до предельного значения адреса. Использование ВЗУ очень удобно, так как в это время пользователь оперирует с общим адресным пространством и ему безразлично, какая физическая память при этом используется внешняя или внутренняя. При работе ВП всего лишь подразумевается различие между виртуальными адресами и физическими.

1. **Архитектурные средства поддержки ВП.**

Очевидно, что невозможно создать полностью машинно-независимый компонент управления виртуальной памятью. С другой стороны, имеются существенные части программного обеспечения, связанного с управлением виртуальной памятью, для которых детали аппаратной реализации совершенно не важны. Одним из достижений современных ОС является грамотное и эффективное разделение средств управления виртуальной памятью на аппаратно-независимую и аппаратно-зависимую части. Коротко рассмотрим, что и каким образом входит в аппаратно-зависимую часть подсистемы управления виртуальной памятью.

В самом распространенном случае необходимо отобразить большое виртуальное адресное пространство в физическое адресное пространство существенно меньшего размера. Пользовательский процесс или ОС должны иметь возможность осуществить запись по виртуальному адресу, а задача ОС – сделать так, чтобы записанная информация оказалась в физической памяти (впоследствии при нехватке оперативной памяти она может быть вытеснена во внешнюю память). В случае виртуальной памяти система отображения адресных пространств помимо трансляции адресов должна предусматривать ведение таблиц, показывающих, какие области виртуальной памяти в данный момент находятся в физической памяти и где именно размещаются.

1. **Страничная и сегментно-страничная ВП.**

**Страничная виртуальная память**

Как и в случае простой страничной организации, страничная виртуальная память и физическая память представляются состоящими из наборов блоков или страниц одинакового размера. Виртуальные адреса делятся на страницы (page), соответствующие единицы в физической памяти образуют страничные кадры (page frames), а в целом система поддержки страничной виртуальной памяти называется пейджингом (paging). Передача информации между памятью и диском всегда осуществляется целыми страницами.

После разбиения менеджером памяти виртуального адресного пространства на страницы виртуальный адрес преобразуется в упорядоченную пару (p,d), где p – номер страницы в виртуальной памяти, а d – смещение в рамках страницы p, внутри которой размещается адресуемый элемент. Процесс может выполняться, если его текущая страница находится в оперативной памяти. Если текущей страницы в главной памяти нет, она должна быть переписана (подкачана) из внешней памяти. Поступившую страницу можно поместить в любой свободный страничный кадр.

Поскольку число виртуальных страниц велико, таблица страниц принимает специфический вид (см. раздел "Структура таблицы страниц "), структура записей становится более сложной, среди атрибутов страницы появляются биты присутствия, модификации и другие управляющие биты.

При отсутствии страницы в памяти в процессе выполнения команды возникает исключительная ситуация, называемая страничное нарушение (page fault) или страничный отказ. Обработка страничного нарушения заключается в том, что выполнение команды прерывается, затребованная страница подкачивается из конкретного места вторичной памяти в свободный страничный кадр физической памяти и попытка выполнения команды повторяется. При отсутствии свободных страничных кадров на диск выгружается редко используемая страница. Проблемы замещения страниц и обработки страничных нарушений рассматриваются в следующей лекции.

Для управления физической памятью ОС поддерживает структуру таблицы кадров. Она имеет одну запись на каждый физический кадр, показывающую его состояние.

В большинстве современных компьютеров со страничной организацией в основной памяти хранится лишь часть таблицы страниц, а быстрота доступа к элементам таблицы текущей виртуальной памяти достигается, как будет показано ниже, за счет использования сверхбыстродействующей памяти, размещенной в кэше процессора.

**Сегментно-страничная организации виртуальной памяти**

Как и в случае простой сегментации, в схемах виртуальной памяти сегмент – это линейная последовательность адресов, начинающаяся с 0. При организации виртуальной памяти размер сегмента может быть велик, например, может превышать размер оперативной памяти. Повторяя все ранее приведенные рассуждения о размещении в памяти больших программ, приходим к разбиению сегментов на страницы и необходимости поддержки своей таблицы страниц для каждого сегмента.

На практике, однако, появления в системе большого количества таблиц страниц стараются избежать, организуя неперекрывающиеся сегменты в одном виртуальном пространстве, для описания которого хватает одной таблицы страниц. Таким образом, одна таблица страниц отводится для всего процесса. Например, в популярных ОС Linux и Windows 2000 все сегменты процесса, а также область памяти ядра ограничены виртуальным адресным пространством объемом 4 Гбайт. При этом ядро ОС располагается по фиксированным виртуальным адресам вне зависимости от выполняемого процесса.

1. **Ассоциативная память.**

Поиск номера кадра, соответствующего нужной странице, в многоуровневой таблице страниц требует нескольких обращений к основной памяти, поэтому занимает много времени. В некоторых случаях такая задержка недопустима. Проблема ускорения поиска решается на уровне архитектуры компьютера.

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.

Естественное решение проблемы ускорения – снабдить компьютер аппаратным устройством для отображения виртуальных страниц в физические без обращения к таблице страниц, то есть иметь небольшую, 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).

Одна запись таблицы в ассоциативной памяти (один вход) содержит информацию об одной виртуальной странице: ее атрибуты и кадр, в котором она находится. Эти поля в точности соответствуют полям в таблице страниц.

Так как ассоциативная память содержит только некоторые из записей таблицы страниц, каждая запись в TLB должна включать поле с номером виртуальной страницы. Память называется ассоциативной, потому что в ней происходит одновременное сравнение номера отображаемой виртуальной страницы с соответствующим полем во всех строках этой небольшой таблицы. Поэтому данный вид памяти достаточно дорого стоит. В строке, поле виртуальной страницы которой совпало с искомым значением, находится номер страничного кадра. Обычное число записей в TLB от 8 до 4096. Рост количества записей в ассоциативной памяти должен осуществляться с учетом таких факторов, как размер кэша основной памяти и количества обращений к памяти при выполнении одной команды.

Рассмотрим функционирование менеджера памяти при наличии ассоциативной памяти.

В начале информация об отображении виртуальной страницы в физическую отыскивается в ассоциативной памяти. Если нужная запись найдена – все нормально, за исключением случаев нарушения привилегий, когда запрос на обращение к памяти отклоняется.

Если нужная запись в ассоциативной памяти отсутствует, отображение осуществляется через таблицу страниц. Происходит замена одной из записей в ассоциативной памяти найденной записью из таблицы страниц. Здесь мы сталкиваемся с традиционной для любого кэша проблемой замещения (а именно какую из записей в кэше необходимо изменить). Конструкция ассоциативной памяти должна организовывать записи таким образом, чтобы можно было принять решение о том, какая из старых записей должна быть удалена при внесении новых.

Число удачных поисков номера страницы в ассоциативной памяти по отношению к общему числу поисков называется hit (совпадение) ratio (пропорция, отношение). Иногда также используется термин "процент попаданий в кэш". Таким образом, hit ratio – часть ссылок, которая может быть сделана с использованием ассоциативной памяти. Обращение к одним и тем же страницам повышает hit ratio. Чем больше hit ratio, тем меньше среднее время доступа к данным, находящимся в оперативной памяти.

Предположим, например, что для определения адреса в случае кэш-промаха через таблицу страниц необходимо 100 нс, а для определения адреса в случае кэш-попадания через ассоциативную память – 20 нс. С 90% hit ratio среднее время определения адреса – 0,9x20+0,1x100 = 28 нс.

Вполне приемлемая производительность современных ОС доказывает эффективность использования ассоциативной памяти. Высокое значение вероятности нахождения данных в ассоциативной памяти связано с наличием у данных объективных свойств: пространственной и временной локальности.

Необходимо обратить внимание на следующий факт. При переключении контекста процессов нужно добиться того, чтобы новый процесс "не видел" в ассоциативной памяти информацию, относящуюся к предыдущему процессу, например очищать ее. Таким образом, использование ассоциативной памяти увеличивает время переключения контекста.

Рассмотренная двухуровневая ( ассоциативная память + таблица страниц ) схема преобразования адреса является ярким примером иерархии памяти, основанной на использовании принципа локальности.

1. **Исключительные ситуации при работе с памятью.**

Отображение виртуального адреса в физический осуществляется при помощи таблицы страниц. Для каждой виртуальной страницы запись в таблице страниц содержит номер соответствующего страничного кадра в оперативной памяти, а также атрибуты страницы для контроля обращений к памяти.

Что же происходит, когда нужной страницы в памяти нет или операция обращения к памяти недопустима? Естественно, что операционная система должна быть как-то оповещена о происшедшем. Обычно для этого используется механизм исключительных ситуаций (exceptions). При попытке выполнить подобное обращение к виртуальной странице возникает исключительная ситуация "страничное нарушение" (pagefault), приводящая к вызову специальной последовательности команд для обработки конкретного вида страничного нарушения.

Страничное нарушение может происходить в самых разных случаях: при отсутствии страницы в оперативной памяти, при попытке записи в страницу с атрибутом "только чтение" или при попытке чтения или записи страницы с атрибутом "только выполнение". В любом из этих случаев вызывается обработчик страничного нарушения, являющийся частью операционной системы. Ему обычно передается причина возникновения исключительной ситуации и виртуальный адрес, обращение к которому вызвало нарушение.

Нас будет интересовать конкретный вариант страничного нарушения - обращение к отсутствующей странице, поскольку именно его обработка во многом определяет производительность страничной системы. Когда программа обращается к виртуальной странице, отсутствующей в основной памяти, операционная система должна выделить страницу основной памяти, переместить в нее копию виртуальной страницы из внешней памяти и модифицировать соответствующий элемент таблицы страниц.

Повышение производительности вычислительной системы может быть достигнуто за счет уменьшения частоты страничных нарушений, а также за счет увеличения скорости их обработки. Время эффективного доступа к отсутствующей в оперативной памяти странице складывается из:

* обслуживания исключительной ситуации (pagefault);
* чтения (подкачки) страницы из вторичной памяти (иногда, при недостатке места в основной памяти, необходимо вытолкнуть одну из страниц из основной памяти во вторичную, то есть осуществить замещение страницы);
* возобновления выполнения процесса, вызвавшего данныйpagefault.

Для решения первой и третьей задач ОС выполняет до нескольких сот машинных инструкций в течение нескольких десятков микросекунд. Время подкачки страницы близко к нескольким десяткам миллисекунд. Проведенные исследования показывают, что вероятности pagefault 5x10-7 оказывается достаточно, чтобы снизить производительность страничной схемы управления памятью на 10%. Таким образом, уменьшение частоты pagefaults является одной из ключевых задач системы управления памятью. Ее решение обычно связано с правильным выбором алгоритма замещения страниц.

1. **Стратегии управления страничной памятью.**

Программное обеспечение подсистемы управления памятью связано с реализацией следующих стратегий:

Стратегия выборки (fetch policy) - в какой момент следует переписать страницу из вторичной памяти в первичную. Существует два основных варианта выборки - по запросу и с упреждением. 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). Такой алгоритм призван уменьшить накладные расходы, связанные с большим количеством исключительных ситуаций, возникающих при работе со значительными объемами данных или кода; кроме того, оптимизируется работа с диском.

Стратегия размещения (placement policy) - в какой участок первичной памяти поместить поступающую страницу. В системах со страничной организацией все просто - в любой свободный страничный кадр. В случае систем с сегментной организацией необходима стратегия, аналогичная стратегии с динамическим распределением.

Стратегия замещения (replacement policy) - какую страницу нужно вытолкнуть во внешнюю память, чтобы освободить место в оперативной памяти. Разумная стратегия замещения, реализованная в соответствующем алгоритме замещения страниц, позволяет хранить в памяти самую необходимую информацию и тем самым снизить частоту страничных нарушений. Замещение должно происходить с учетом выделенного каждому процессу количества кадров. Кроме того, нужно решить, должна ли замещаемая страница принадлежать процессу, который инициировал замещение, или она должна быть выбрана среди всех кадров основной памяти.

**Контрольные вопросы:**

1. Раскройте понятие виртуальной памяти.
2. Перечислите архитектурные средства поддержки ВП.
3. Охарактеризуйте страничную и сегментно-страничную ВП.
4. Раскройте понятие ассоциативная память.
5. Перечислите исключительные ситуации при работе с памятью.
6. Укажите стратегии управления страничной памятью.

**Литература:**

1. А.П. Побегайло, Системное программирование в Windows, БХВ-Петербург, 2006, 1056 с.

2. Ахо, Лам, Сети, Ульман - Компиляторы. Принципы, технологии, инструменты, Изд.: Вильямс, 2008, 1185 с.

3. Л.Бек, Введение в системное программирование, М., Мир, 2008г.

4. Р. И. Компаниец, Системное программирование : Основы построения трансляторов, Изд.: СПб., 2000.

5. Ф.Льюис, Д.Розенкранц, Р.Стирнз. Теоретические основы проектирования компиляторов. - М.: Мир, 20099, 655 с.

6. Э. Таненбау, Современные операционные системы, Изд.: Питер, 2011, 1120 с.

7. А. В. Гордеев, А. Ю. Молчанов, Системное программное обеспечение, учебник для ВУЗов, СПб.: Питер, 2010 г.

**СЕМИНАР №15: «ФАЙЛОВАЯ СИСТЕМА».**

**План семинара:**

1. Функции и интерфейс файловой системы (ФС).
2. Понятие ФС.
3. Организация файлов и доступ к ним.
4. Операции над файлами.
5. Каталогифайлов. Логическая структура каталогов.
6. Операции над каталогами.
7. Защитафайлов и каталогов.
8. Общая структура ФС.
9. Управление внешней памятью.
10. Производительность ФС.

**Содержание по плану:**

1. **Функции и интерфейс файловой системы (ФС).**

Основные функции файловой системы:

* Идентификация файлов. Связывание имени файла с выделенным ему пространством внешней памяти.
* Распределение внешней памяти между файлами.
* Обеспечение надежности и отказоустойчивости.
* Обеспечение защиты от несанкционированного доступа.
* Обеспечение совместного доступа к файлам, так чтобы пользователю не приходилось прилагать специальных усилий по обеспечению синхронизации доступа.
* Обеспечение высокой производительности.

1. **Понятие ФС.**

История систем управления данными во внешней памяти начинается еще с магнитных лент, но современный облик они приобрели с появлением магнитных дисков. До этого каждая прикладная программа сама решала проблемы именования данных и их структуризации во внешней памяти. Это затрудняло поддержание на внешнем носителе нескольких архивов долговременно хранящейся информации.

Историческим шагом стал переход к использованию централизованных систем управления файлами. Система управления файлами берет на себя распределение внешней памяти, отображение имен файлов в адреса внешней памяти и обеспечение доступа к данным.

Файловая система (ФС) - это часть операционной системы, назначение которой состоит в том, чтобы организовать эффективную работу с данными, хранящимися во внешней памяти, и обеспечить пользователю удобный интерфейс при работе с такими данными.

Организовать хранение информации на магнитном диске непросто. Это требует, например, хорошего знания устройства контроллера диска, особенностей работы с его регистрами. Непосредственное взаимодействие с диском - прерогатива компонента системы ввода-вывода ОС, называемого драйвером диска. Для того чтобы избавить пользователя компьютера от сложностей взаимодействия с аппаратурой, была придумана ясная абстрактная модель файловой системы. Операции записи или чтения файла концептуально проще, чем низкоуровневые операции работы с устройствами.

Основная идея использования внешней памяти состоит в следующем.

· ОС делит память на блоки фиксированного размера, например, 4096 байт.

· файл, обычно представляющий собой неструктурированную последовательность однобайтовых записей, хранится в виде последовательности блоков (не обязательно смежных);

· каждый блок хранит целое число записей.

В некоторых ОС (MS-DOS) адреса блоков, содержащих данные файла, могут быть организованы в связный список и вынесены в отдельную таблицу в памяти.

В других ОС (Unix) адреса блоков данных файла хранятся в отдельном блоке внешней памяти (так называемом индексе или индексном узле). Этот прием, называемый индексацией, является наиболее распространенным для приложений, требующих произвольного доступа к записям файлов.

Индекс файла состоит из списка элементов, каждый из которых содержит номер блока в файле и сведения о местоположении данного блока. Считывание очередного байта осуществляется с так называемой текущей позиции, которая характеризуется смещением от начала файла. Зная размер блока, легко вычислить номер блока, содержащего текущую позицию. Адрес же нужного блока диска можно затем извлечь из индекса файла. Базовой операцией, выполняемой по отношению к файлу, является чтение блока с диска и перенос его в буфер, находящийся в основной памяти.

Файловая система позволяет при помощи системы справочников (каталогов, директорий) связать уникальное имя файла с блоками вторичной памяти, содержащими данные файла. Иерархическая структура каталогов, используемая для управления файлами, может служить другим примером индексной структуры. В этом случае каталоги или папки играют роль индексов, каждый из которых содержит ссылки на свои подкаталоги. С этой точки зрения вся файловая система компьютера представляет собой большой индексированный файл. Помимо собственно файлов и структур данных, используемых для управления файлами (каталоги, дескрипторы файлов, различные таблицы распределения внешней памяти), понятие "файловая система" включает программные средства, реализующие различные операции над файлами.

1. **Организация файлов и доступ к ним.**

Программист воспринимает файл в виде набора однородных записей.

Запись - это наименьший элемент данных, который может быть обработан как единое целое прикладной программой при обмене с внешним устройством.

Причем в большинстве ОС размер записи равен одному байту. В то время как приложения оперируют записями, физический обмен с устройством осуществляется большими единицами (обычно блоками). Поэтому записи объединяются в блоки для вывода и разблокируются - для ввода. Вопросы распределения блоков внешней памяти между файлами рассматриваются в следующей лекции.

ОС поддерживают несколько вариантов структуризации файлов.

**Последовательный файл**

Простейший вариант - так называемый последовательный файл. То есть файл является последовательностью записей. Поскольку записи, как правило, однобайтовые, файл представляет собой неструктурированную последовательность байтов.

Обработка подобных файлов предполагает последовательное чтение записей от начала файла, причем конкретная запись определяется ее положением в файле. Такой способ доступа называется последовательным (модель ленты). Если в качестве носителя файла используется магнитная лента, то так и делается. Текущая позиция считывания может быть возвращена к началу файла (rewind).

**Файл прямого доступа**

В реальной практике файлы хранятся на устройствах прямого (random) доступа, например на дисках, поэтому содержимое файла может быть разбросано по разным блокам диска, которые можно считывать в произвольном порядке. Причем номер блока однозначно определяется позицией внутри файла.

Здесь имеется в виду относительный номер, специфицирующий данный блок среди блоков диска, принадлежащих файлу.

Естественно, что в этом случае для доступа к середине файла просмотр всего файла с самого начала не обязателен. Для специфицирования места, с которого надо начинать чтение, используются два способа: с начала или с текущей позиции, которую дает операция seek.

Файл, байты которого могут быть считаны в произвольном порядке, называется файлом прямого доступа.

Таким образом, файл, состоящий из однобайтовых записей на устройстве прямого доступа, - наиболее распространенный способ организации файла. Базовыми операциями для такого рода файлов являются считывание или запись символа в текущую позицию. В большинстве языков высокого уровня предусмотрены операторы посимвольной пересылки данных в файл или из него.

Подобную логическую структуру имеют файлы во многих файловых системах, например в файловых системах ОС Unix и MS-DOS. ОС не осуществляет никакой интерпретации содержимого файла. Эта схема обеспечивает максимальную гибкость и универсальность. С помощью базовых системных вызовов (или функций библиотеки ввода/вывода) пользователи могут как угодно структурировать файлы. В частности, многие СУБД хранят свои базы данных в обычных файлах.

**Другие формы организации файлов**

Известны как другие формы организации файла, так и другие способы доступа к ним, которые использовались в ранних ОС, а также применяются сегодня в больших мэйнфреймах (mainframe), ориентированных на коммерческую обработку данных.

Первый шаг в структурировании - хранение файла в виде последовательности записей фиксированной длины, каждая из которых имеет внутреннюю структуру. Операция чтения производится над записью, а операция записи переписывает или добавляет запись целиком.

Ранее использовались записи по 80 байт (это соответствовало числу позиций в перфокарте) или по 132 символа (ширина принтера). В ОС CP/M файлы были последовательностями 128-символьных записей. С введением CRT-терминалов данная идея утратила популярность.

Другой способ представления файлов - последовательность записей переменной длины, каждая из которых содержит ключевое поле в фиксированной позиции внутри записи (см. рис. 2.1).

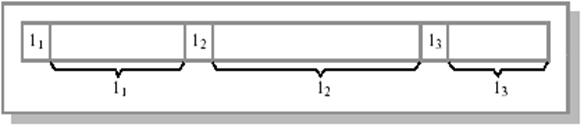


Рис. 2.1. Файл как последовательность записей переменной длины

В некоторых системах ускорение доступа к файлу обеспечивается конструированием индекса файла. Индекс обычно хранится на том же устройстве, что и сам файл, и состоит из списка элементов, каждый из которых содержит идентификатор записи, за которым следует указание о местоположении данной записи. Для поиска записи вначале происходит обращение к индексу, где находится указатель на нужную запись.

Такие файлы называются индексированными, а метод доступа к ним - доступ с использованием индекса.

Предположим, у нас имеется большой несортированный файл, содержащий разнообразные сведения о студентах, состоящие из записей с несколькими полями, и возникает задача организации быстрого поиска по одному из полей, например по фамилии студента.

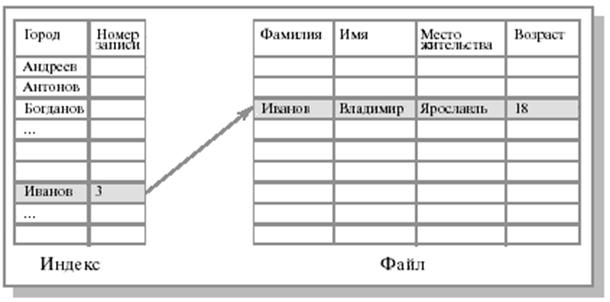


Рис. 2.2 иллюстрирует решение данной проблемы - организацию метода доступа к файлу с использованием индекса.

Базисная операция в данном случае - считать запись с каким-либо значением ключа. Записи могут располагаться в файле последовательно (например, отсортированные по значению ключевого поля) или в более сложном порядке.

Метод доступа по значению ключевого поля к записям последовательного файла называется индексно-последовательным.

Способ выделения дискового пространства при помощи индексных узлов, применяемый в ряде ОС (Unix и некоторых других), может служить примером организации индекса.

В этом случае ОС использует древовидную организацию блоков, при которой блоки, составляющие файл, являются листьями дерева, а каждый внутренний узел содержит указатели на множество блоков файла. Для больших файлов индекс может быть слишком велик. В этом случае создают индекс для индексного файла (блоки промежуточного уровня или блоки косвенной адресации).

1. **Операции над файлами.**

Операционная система должна предоставить в распоряжение пользователя набор операций для работы с файлами, реализованных через системные вызовы.

Чаще всего при работе с файлом пользователь выполняет не одну, а несколько операций. Во-первых, нужно найти данные файла и его атрибуты по символьному имени, во-вторых, считать необходимые атрибуты файла в отведенную область оперативной памяти и проанализировать права пользователя на выполнение требуемой операции. Затем следует выполнить операцию, после чего освободить занимаемую данными файла область памяти.

**Основные файловые операции**

· Создание файла, не содержащего данных. Смысл данного вызова - объявить, что файл существует, и присвоить ему ряд атрибутов. При этом выделяется место для файла на диске и вносится запись в каталог.

· Удаление файла и освобождение занимаемого им дискового пространства.

· Открытие файла. Перед использованием файла процесс должен его открыть. Цель данного системного вызова - разрешить системе проанализировать атрибуты файла и проверить права доступа к нему, а также считать в оперативную память список адресов блоков файла для быстрого доступа к его данным.

· Закрытие файла. Если работа с файлом завершена, его атрибуты и адреса блоков на диске больше не нужны. В этом случае файл нужно закрыть, чтобы освободить место во внутренних таблицах файловой системы.

· Позиционирование. Дает возможность специфицировать место внутри файла, откуда будет производиться считывание (или запись) данных, то есть задать текущую позицию.

· Чтение данных из файла. Обычно это делается с текущей позиции. Пользователь должен задать объем считываемых данных и предоставить для них буфер в оперативной памяти.

· Запись данных в файл с текущей позиции. Если текущая позиция находится в конце файла, его размер увеличивается, в противном случае запись осуществляется на место имеющихся данных, которые, таким образом, теряются.

Есть и другие операции, например, переименование файла, получение атрибутов файла и т. д.

1. Каталоги файлов. Логическая структура каталогов.

Количество файлов на компьютере может быть большим. Отдельные системы хранят тысячи файлов, занимающие сотни гигабайтов дискового пространства. Эффективное управление этими данными подразумевает наличие в них четкой логической структуры.

Все современные файловые системы поддерживают многоуровневое именование файлов за счет наличия во внешней памяти дополнительных файлов со специальной структурой - каталогов (или директорий).

Каждый каталог содержит список каталогов и/или файлов, содержащихся в данном каталоге. Каталоги имеют один и тот же внутренний формат, где каждому файлу соответствует одна запись в файле директории (см., например, рис. 2.3).

Число директорий зависит от системы. В ранних ОС имелась только одна корневая директория, затем появились директории для пользователей (по одной директории на пользователя). В современных ОС используется произвольная структура дерева директорий.

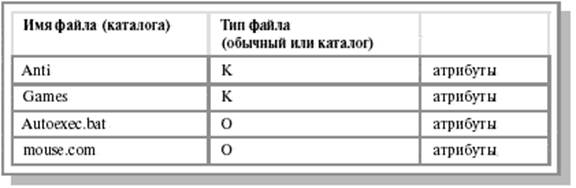


Рис. 2.3. Директории

Таким образом, файлы на диске образуют иерархическую древовидную структуру (см. рис. 2.4).

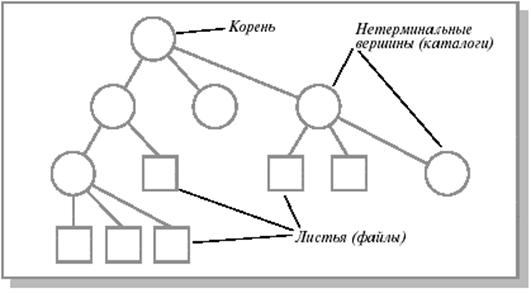


Рис. 2.4. Древовидная структура файловой системы

Существует несколько эквивалентных способов изображения дерева. Структура перевернутого дерева, приведенного на рис. 2.4, наиболее распространена. Верхнюю вершину называют корнем. Если элемент дерева не может иметь потомков, он называется терминальной вершиной или листом (в данном случае является файлом). Нелистовые вершины - справочники или каталоги содержат списки листовых и нелистовых вершин. Путь от корня к файлу однозначно определяет файл.

Подобные древовидные структуры являются графами, не имеющими циклов. Можно считать, что ребра графа направлены вниз, а корень - вершина, не имеющая входящих ребер. Как мы увидим в следующей лекции, связывание файлов, которое практикуется в ряде операционных систем, приводит к образованию циклов в графе.

Внутри одного каталога имена листовых файлов уникальны. Имена файлов, находящихся в разных каталогах, могут совпадать.

Для того чтобы однозначно определить файл по его имени (избежать коллизии имен), принято именовать файл так называемым абсолютным или полным именем (pathname), состоящим из списка имен вложенных каталогов, по которому можно найти путь от корня к файлу плюс имя файла в каталоге, непосредственно содержащем данный файл.

То есть полное имя включает цепочку имен - путь к файлу, например /usr/games/doom. Такие имена уникальны. Компоненты пути разделяют различными символами: "/" (слэш) в Unix или обратными слэшем в MS-DOS (в Multics - ">"). Таким образом, использование древовидных каталогов минимизирует сложность назначения уникальных имен.

Указывать полное имя не всегда удобно, поэтому применяют другой способ задания имени - относительный путь к файлу. Он использует концепцию рабочей или текущей директории, которая обычно входит в состав атрибутов процесса, работающего с данным файлом.

Тогда на файлы в такой директории можно ссылаться только по имени, при этом поиск файла будет осуществляться в рабочем каталоге. Это удобнее, но, по существу, то же самое, что и абсолютная форма.

Для получения доступа к файлу и локализации его блоков система должна выполнить навигацию по каталогам.

Рассмотрим для примера путь /usr/linux/progr.c. Алгоритм одинаков для всех иерархических систем. Сначала в фиксированном месте на диске находится корневая директория. Затем находится компонент пути usr, т. е. в корневой директории ищется файл /usr. Исследуя этот файл, система понимает, что данный файл является каталогом, и блоки его данных рассматривает как список файлов и ищет следующий компонент linux в нем. Из строки для linux находится файл, соответствующий компоненту usr/linux/. Затем находится компонент progr.c, который открывается, заносится в таблицу открытых файлов и сохраняется в ней до закрытия файла.

Многие прикладные программы работают с файлами, находящимися в текущей директории, не указывая явным образом ее имени. Это дает пользователю возможность произвольным образом именовать каталоги, содержащие различные программные пакеты. Для реализации этой возможности в большинстве ОС, поддерживающих иерархическую структуру директорий, используется обозначение "." - для текущей директории и ".." - для родительской.

1. **Операции над каталогами.**

Как и в случае с файлами, система обязана обеспечить пользователя набором операций, необходимых для работы с директориями, реализованных через системные вызовы. Несмотря на то что директории - это файлы, логика работы с ними отличается от логики работы с обычными файлами и определяется природой этих объектов, предназначенных для поддержки структуры файлового архива. Совокупность системных вызовов для управления директориями зависит от особенностей конкретной ОС. Напомним, что операции над каталогами являются прерогативой ОС, то есть пользователь не может, например, выполнить запись в каталог начиная с текущей позиции.

**Некоторые операции с каталогами.**

·Создание директории. Вновь созданная директория включает записи с именами '.' и '..', однако считается пустой.

· Удаление директории. Удалена может быть только пустая директория.

·Открытие директории для последующего чтения. Например, чтобы перечислить файлы, входящие в директорию, процесс должен открыть директорию и считать имена всех файлов, которые она включает.

· Закрытие директории после ее чтения для освобождения места во внутренних системных таблицах.

· Поиск. Данный системный вызов возвращает содержимое текущей записи в открытой директории. Вообще говоря, для этих целей может использоваться системный вызов Read, но в этом случае от программиста потребуется знание внутренней структуры директории.

· Получение списка файлов в каталоге.

· Переименование. Имена директорий можно менять, как и имена файлов.

· Создание файла. При создании нового файла необходимо добавить в каталог соответствующий элемент.

· Удаление файла. Удаление из каталога соответствующего элемента. Если удаляемый файл присутствует только в одной директории, то он вообще удаляется из файловой системы, в противном случае система ограничивается только удалением специфицируемой записи.

Очевидно, что создание и удаление файлов предполагает также выполнение соответствующих файловых операций. Имеется еще ряд других системных вызовов, например связанных с защитой информации.

1. **Защита файлов и каталогов.**

Информация в компьютерной системе должна быть защищена как от физического разрушения (reliability), так и от несанкционированного доступа (protection).

Здесь мы коснемся отдельных аспектов защиты, связанных с контролем доступа к файлам.

**Контроль доступа к файлам**

Наличие в системе многих пользователей предполагает организацию контролируемого доступа к файлам. Выполнение любой операции над файлом должно быть разрешено только в случае наличия у пользователя соответствующих привилегий.

Обычно контролируются следующие операции: чтение, запись и выполнение. Другие операции, например копирование файлов или их переименование, также могут контролироваться. Однако они чаще реализуются через перечисленные. Так, операцию копирования файлов можно представить как операцию чтения и последующую операцию записи.

**Списки прав доступа**

Наиболее общий подход к защите файлов от несанкционированного использования - сделать доступ зависящим от идентификатора пользователя, то есть связать с каждым файлом или директорией список прав доступа (accesscontrollist), где перечислены имена пользователей и типы разрешенных для них способов доступа к файлу.

Любой запрос на выполнение операции сверяется с таким списком. Основная проблема реализации данного способа - список может быть длинным. Чтобы разрешить всем пользователям читать файл, необходимо всех их внести в список.

Для решения этих проблем создают классификации пользователей, например, в ОС Unix все пользователи разделены на три группы.

· Владелец (Owner).

· Группа (Group). Набор пользователей, разделяющих файл и нуждающихся в типовом способе доступа к нему.

·Остальные (Univers).

Итак, файловая система представляет собой набор файлов, директорий и операций над ними. Имена, структуры файлов, способы доступа к ним и их атрибуты - важные аспекты организации файловой системы. Обычно файл представляет собой неструктурированную последовательность байтов. Главная задача файловой системы - связать символьное имя файла с данными на диске. Большинство современных ОС поддерживает иерархическую систему каталогов или директорий с возможным вложением директорий. Безопасность файловой системы, базирующаяся на ведении списков прав доступа, - одна из важнейших концепций ОС.

**8. Общая структура ФС.**

Система хранения данных на дисках может быть структурирована следующим образом (см. рис. 3.1).

Нижний уровень - оборудование.

Это в первую очередь магнитные диски с подвижными головками - основные устройства внешней памяти, представляющие собой пакеты магнитных пластин (поверхностей), между которыми на одном рычаге двигается пакет магнитных головок. Шаг движения пакета головок является дискретным, и каждому положению пакета головок логически соответствует цилиндр магнитного диска.

Цилиндры делятся на дорожки (треки), а каждая дорожка размечается на одно и то же количество блоков (секторов) таким образом, что в каждый блок можно записать по максимуму одно и то же число байтов.

Следовательно, для обмена с магнитным диском на уровне аппаратуры нужно указать номер цилиндра, номер поверхности, номер блока на соответствующей дорожке и число байтов, которое нужно записать или прочитать от начала этого блока. Таким образом, диски могут быть разбиты на блоки фиксированного размера и можно непосредственно получить доступ к любому блоку (организовать прямой доступ к файлам).

Непосредственно с устройствами (дисками) взаимодействует часть ОС, называемая системой ввода-вывода. Система ввода-вывода предоставляет в распоряжение более высокоуровневого компонента ОС - файловой системы - используемое дисковое пространство в виде непрерывной последовательности блоков фиксированного размера.

Система ввода-вывода имеет дело с физическими блоками диска, которые характеризуются адресом, например диск 2, цилиндр 75, сектор 11.

Файловая система имеет дело с логическими блоками, каждый из которых имеет номер (от 0 или 1 до N). Размер логических блоков файла совпадает или является кратным размеру физического блока диска и может быть задан равным размеру страницы виртуальной памяти, поддерживаемой аппаратурой компьютера совместно с операционной системой.

В структуре системы управления файлами можно выделить базисную подсистему, которая отвечает за выделение дискового пространства конкретным файлам, и более высокоуровневую логическую подсистему, которая использует структуру дерева директорий для предоставления модулю базисной подсистемы необходимой ей информации, исходя из символического имени файла. Она также ответственна за авторизацию доступа к файлам.

Стандартный запрос на открытие (open) или создание (create) файла поступает от прикладной программы к логической подсистеме. Логическая подсистема, используя структуру директорий, проверяет права доступа и вызывает базовую подсистему для получения доступа к блокам файла. После этого файл считается открытым, он содержится в таблице открытых файлов, и прикладная программа получает в свое распоряжение дескриптор (или handle в системах Microsoft) этого файла.

Дескриптор файла является ссылкой на файл в таблице открытых файлов и используется в запросах прикладной программы на чтение-запись из этого файла.

Запись в таблице открытых файлов указывает через систему выделения блоков диска на блоки данного файла. Если к моменту открытия файл уже используется другим процессом, то есть содержится в таблице открытых файлов, то после проверки прав доступа к файлу может быть организован совместный доступ. При этом новому процессу также возвращается дескриптор - ссылка на файл в таблице открытых файлов.

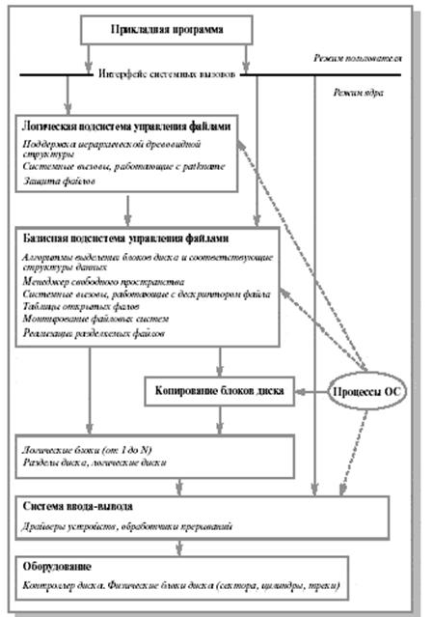


Рис. 3.1. Блок-схема файловой системы

**9. Управление внешней памятью.**

Прежде чем описывать структуру данных файловой системы на диске, необходимо рассмотреть алгоритмы выделения дискового пространства и способы учета свободной и занятой дисковой памяти. Эти задачи связаны между собой.

Методы выделения дискового пространства

Ключевым, безусловно, является вопрос, какой тип структур используется для учета отдельных блоков файла, то есть способ связывания файлов с блоками диска.

В ОС используется несколько методов выделения файлу дискового пространства. Для каждого из методов запись в директории, соответствующая символьному имени файла, содержит указатель, следуя которому можно найти все блоки данного файла.

Выделение непрерывной последовательностью блоков

Простейший способ - хранить каждый файл как непрерывную последовательность блоков диска. При непрерывном расположении файл характеризуется адресом и длиной (в блоках). Файл, стартующий с блока b, занимает затем блоки b+1, b+2, ... b+n-1.

Эта схема имеет два преимущества. Во-первых, ее легко реализовать, так как выяснение местонахождения файла сводится к вопросу, где находится первый блок. Во-вторых, она обеспечивает хорошую производительность, так как целый файл может быть считан за одну дисковую операцию.

Непрерывное выделение используется в ОС IBM/CMS, в ОС RSX-11 (для выполняемых файлов) и в ряде других.

Этот способ распространен мало. В процессе эксплуатации диск представляет собой некоторую совокупность свободных и занятых фрагментов. Не всегда имеется подходящий по размеру свободный фрагмент для нового файла. Проблема непрерывного расположения может рассматриваться как частный случай более общей проблемы выделения блока нужного размера из списка свободных блоков. Метод страдает от внешней фрагментации в зависимости от размера диска и среднего размера файла.

Единственным приемлемым решением перечисленных проблем является периодическое уплотнение содержимого внешней памяти, или "сборка мусора", цель которой состоит в объединении свободных участков в один большой блок. Но это дорогостоящая операция, которую невозможно осуществлять слишком часто.

Таким образом, когда содержимое диска постоянно изменяется, данный метод нерационален. Однако для стационарных файловых систем, например для файловых систем компакт-дисков, он вполне пригоден.

**Связный список**

Внешняя фрагментация - основная проблема рассмотренного выше метода - может быть устранена за счет представления файла в виде связного списка блоков диска.

Запись в директории содержит указатель на первый и последний блоки файла (иногда в качестве варианта используется специальный знак конца файла - EOF).

Каждый блок содержит указатель на следующий блок (см. рис. 3.2).



Рис. 3.2. Хранение файла в виде связного списка дисковых блоков

Внешняя фрагментация для данного метода отсутствует. Любой свободный блок может быть использован для удовлетворения запроса. Заметим, что нет необходимости декларировать размер файла в момент создания. Файл может расти неограниченно.

Связное выделение имеет, однако, несколько существенных недостатков.

Во-первых, при прямом доступе к файлу для поиска i-го блока нужно осуществить несколько обращений к диску, последовательно считывая блоки от 1 до i-1, то есть выборка логически смежных записей, которые занимают физически несмежные секторы, может требовать много времени. Здесь мы теряем все преимущества прямого доступа к файлу.

Во-вторых, данный способ не очень надежен. Наличие дефектного блока в списке приводит к потере информации в оставшейся части файла и потенциально к потере дискового пространства, отведенного под этот файл.

Наконец, для указателя на следующий блок внутри блока нужно выделить место, что не всегда удобно. Емкость блока, традиционно являющаяся степенью двойки (многие программы читают и пишут блоками по степеням двойки), таким образом, перестает быть степенью двойки, так как указатель отбирает несколько байтов.

Поэтому метод связного списка обычно в чистом виде не используется.

**Таблица отображения файлов**

Одним из вариантов предыдущего способа является хранение указателей не в дисковых блоках, а в индексной таблице в памяти, которая называется таблицей отображения файлов (FAT - file allocation table) (см. рис. 3.3).

Этой схемы придерживаются многие ОС (MS-DOS, OS/2, MS Windows и др.)

По-прежнему существенно, что запись в директории содержит только ссылку на первый блок. Далее при помощи таблицы FAT можно локализовать блоки файла независимо от его размера. В тех строках таблицы, которые соответствуют последним блокам файлов, обычно записывается некоторое граничное значение, например EOF.

Главное достоинство данного подхода состоит в том, что по таблице отображения можно судить о физическом соседстве блоков, располагающихся на диске, и при выделении нового блока можно легко найти свободный блок диска, находящийся поблизости от других блоков данного файла. Минусом данной схемы может быть необходимость хранения в памяти этой довольно большой таблицы.



Рис. 13.3. Метод связного списка с использованием таблицы в оперативной памяти

**Индексные узлы**

Наиболее распространенный метод выделения файлу блоков диска - связать с каждым файлом небольшую таблицу, называемую индексным узлом (i-node), которая перечисляет атрибуты и дисковые адреса блоков файла (см. рис 3.4).

Запись в директории, относящаяся к файлу, содержит адрес индексного блока. По мере заполнения файла указатели на блоки диска в индексном узле принимают осмысленные значения.

Индексирование поддерживает прямой доступ к файлу, без ущерба от внешней фрагментации. Индексированное размещение широко распространено и поддерживает как последовательный, так и прямой доступ к файлу.

Обычно применяется комбинация одноуровневого и многоуровневых индексов.

Первые несколько адресов блоков файла хранятся непосредственно в индексном узле, таким образом, для маленьких файлов индексный узел хранит всю необходимую информацию об адресах блоков диска. Для больших файлов один из адресов индексного узла указывает на блок косвенной адресации. Данный блок содержит адреса дополнительных блоков диска.

Если этого недостаточно, используется блок двойной косвенной адресации, который содержит адреса блоков косвенной адресации. Если и этого не хватает, используется блок тройной косвенной адресации.

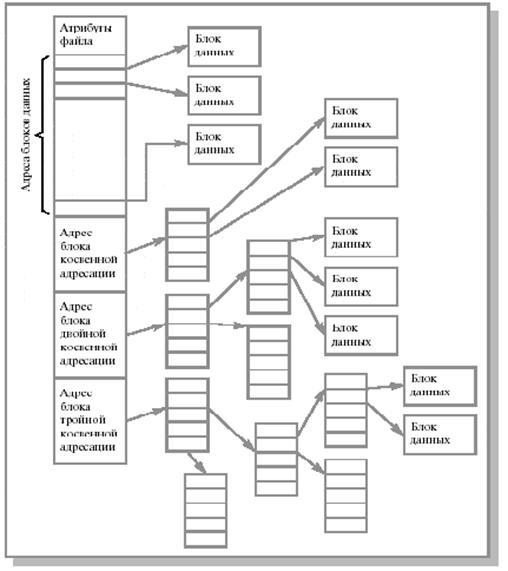


Рис. 3.4. Структура индексного узла

Данную схему используют файловые системы Unix (а также файловые системы HPFS, NTFS и др.). Такой подход позволяет при фиксированном, относительно небольшом размере индексного узла поддерживать работу с файлами, размер которых может меняться от нескольких байтов до нескольких гигабайтов. Существенно, что для маленьких файлов используется только прямая адресация, обеспечивающая максимальную производительность.

**10. Реализация каталогов.**

Как уже говорилось, директория или каталог - это файл, имеющий вид таблицы и хранящий список входящих в него файлов или каталогов.

Основная задача файлов-директорий - поддержка иерархической древовидной структуры файловой системы. Запись в директории имеет определенный для данной ОС формат, поэтому блоки данных файла-директории заполняются не через операции записи, а при помощи специальных системных вызовов (например, создание файла).

Для доступа к файлу ОС использует путь (pathname), сообщенный пользователем. Запись в директории связывает имя файла или имя поддиректории с блоками данных на диске (см. рис. 3.6). В зависимости от способа выделения файлу блоков диска эта ссылка может быть номером первого блока или номером индексного узла. В любом случае обеспечивается связь символьного имени файла с данными на диске.

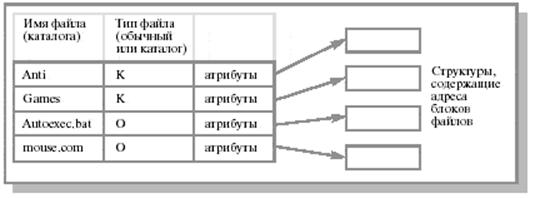


Рис. 3.6. Реализация директорий

Когда система открывает файл, она ищет его имя в директории. Затем из записи в директории или из структуры, на которую запись в директории указывает, извлекаются атрибуты и адреса блоков файла на диске. Эта информация помещается в системную таблицу в главной памяти.

Все последующие ссылки на данный файл используют эту информацию. Атрибуты файла можно хранить непосредственно в записи в директории, как показано на рис. 3.6. Однако для организации совместного доступа к файлам удобнее хранить атрибуты в индексном узле, как это делается в Unix.

Рассмотрим несколько конкретных примеров.

Примеры реализации директорий в некоторых ОС

Директории в ОС MS-DOS

В ОС MS-DOS типовая запись в директории имеет вид, показанный на рис. 3.7.



Рис. 3.7. Вариант записи в директории MS-DOS

В ОС MS-DOS, как и в большинстве современных ОС, директории могут содержать поддиректории (специфицируемые битом атрибута), что позволяет конструировать произвольное дерево директорий файловой системы.

Номер первого блока используется в качестве индекса в таблице FAT. Далее по цепочке в этой таблице могут быть найдены остальные блоки.

**Директории в ОС Unix**

Структура директории проста. Каждая запись содержит имя файла и номер его индексного узла (см. рис. 3.8). Вся остальная информация о файле (тип, размер, время модификации, владелец и т. д. и номера дисковых блоков) находится в индексном узле.



Рис. 3.8. Вариант записи в директории Unix

В более поздних версиях Unix форма записи претерпела ряд изменений, например, имя файла описывается структурой.

**Поиск в директории**

Список файлов в директории обычно не является упорядоченным по именам файлов. Поэтому правильный выбор алгоритма поиска имени файла в директории имеет большое влияние на эффективность и надежность файловых систем.

**Линейный поиск**

Существует несколько стратегий просмотра списка символьных имен.

Простейшей из них является линейный поиск. Директория просматривается с самого начала, пока не встретится нужное имя файла.

Хотя это наименее эффективный способ поиска, оказывается, что в большинстве случаев он работает с приемлемой производительностью. Например, авторы Unix утверждали, что линейного поиска вполне достаточно. По-видимому, это связано с тем, что на фоне относительно медленного доступа к диску некоторые задержки, возникающие в процессе сканирования списка, несущественны.

Метод прост, но требует временных затрат. Для создания нового файла вначале нужно проверить директорию на наличие такого же имени. Затем имя нового файла вставляется в конец директории (если, разумеется, файл с таким же именем в директории не существует, в противном случае нужно информировать пользователя). Для удаления файла нужно также выполнить поиск его имени в списке и пометить запись как неиспользуемую.

Реальный недостаток данного метода - последовательный поиск файла. Информация о структуре директории используется часто, и неэффективный способ поиска будет заметен пользователями. Можно свести поиск к бинарному, если отсортировать список файлов. Однако это усложнит создание и удаление файлов, так как требуется перемещение большого объема информации.

**Хеш-таблица**

Хеширование - другой способ, который может использоваться для размещения и последующего поиска имени файла в директории.

В данном методе имена файлов также хранятся в каталоге в виде линейного списка, но дополнительно используется хеш-таблица. Хеш-таблица, точнее построенная на ее основе хеш-функция, позволяет по имени файла получить указатель на имя файла в списке.

Таким образом, можно существенно уменьшить время поиска.

В результате хеширования могут возникать ситуации, когда функция хеширования, примененная к разным именам файлов, дает один и тот же результат. Обычно имена таких файлов объединяют в связные списки, предполагая в дальнейшем осуществление в них последовательного поиска нужного имени файла. Выбор подходящего алгоритма хеширования позволяет свести к минимуму число ситуаций. Однако всегда есть вероятность неблагоприятного исхода. В таком случае преимущество использования этой схемы по сравнению с последовательным поиском практически утрачивается.

**Другие методы поиска**

Помимо описанных методов поиска имени файла, в директории существуют и другие. В качестве примера можно привести организацию поиска в каталогах файловой системы NTFS при помощи так называемого B-дерева, которое стало стандартным способом организации индексов в системах баз данных.

**10. Производительность ФС.**

Поскольку обращение к диску - операция относительно медленная, минимизация количества таких обращений - ключевая задача всех алгоритмов, работающих с внешней памятью.

Наиболее типичная техника повышения скорости работы с диском - кэширование.

**Кэширование**

Кэш диска представляет собой буфер в оперативной памяти, содержащий ряд блоков диска (см. рис. 3.9).

Если имеется запрос на чтение/запись блока диска, то сначала производится проверка на предмет наличия этого блока в кэше. Если блок в кэше имеется, то запрос удовлетворяется из кэша, в противном случае запрошенный блок считывается в кэш с диска.

Аккуратная реализация кэширования требует решения нескольких проблем.

Во-первых, емкость буфера кэша ограничена. Когда блок должен быть загружен в заполненный буфер кэша, возникает проблема замещения блоков, то есть отдельные блоки должны быть удалены из него. Здесь работают те же стратегии и те же FIFO, Second Chance и LRU-алгоритмы замещения, что и при выталкивании страниц памяти.

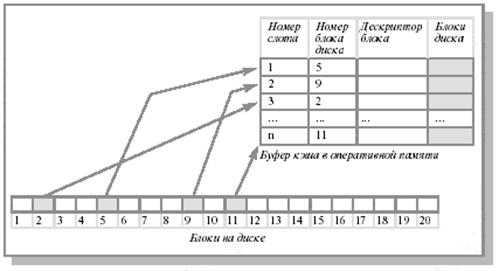


Рис. 3.9. Структура блочного кэша

Замещение блоков должно осуществляться с учетом их важности для файловой системы. Блоки должны быть разделены на категории, например: блоки индексных узлов, блоки косвенной адресации, блоки директорий, заполненные блоки данных и т. д., и в зависимости от принадлежности блока к той или иной категории можно применять к ним разную стратегию замещения.

Во-вторых, поскольку кэширование использует механизм отложенной записи, при котором модификация буфера не вызывает немедленной записи на диск, серьезной проблемой является "старение" информации в дисковых блоках, образы которых находятся в буферном кэше.

Несвоевременная синхронизация буфера кэша и диска может привести к очень нежелательным последствиям в случае отказов оборудования или программного обеспечения. Поэтому стратегия и порядок отображения информации из кэша на диск должна быть тщательно продумана.

Наконец, проблема конкуренции процессов на доступ к блокам кэша решается ведением списков блоков, пребывающих в различных состояниях, и отметкой о состоянии блока в его дескрипторе. Например, блок может быть заблокирован, участвовать в операции ввода-вывода, а также иметь список процессов, ожидающих освобождения данного блока.

**Оптимальное размещение информации на диске**

Кэширование - не единственный способ увеличения производительности системы. Другая важная техника - сокращение количества движений считывающей головки диска за счет разумной стратегии размещения информации. Например, массив индексных узлов в Unix стараются разместить на средних дорожках. Также имеет смысл размещать индексные узлы поблизости от блоков данных, на которые они ссылаются и т. д.

Кроме того, рекомендуется периодически осуществлять дефрагментацию диска (сборку мусора), поскольку в популярных методиках выделения дисковых блоков (за исключением, может быть, FAT) принцип локальности не работает, и последовательная обработка файла требует обращения к различным участкам диска.

Контрольные вопросы:

1. Перечислите функции ФС.
2. Раскройте понятие интерфейс ФС.
3. Опишите организацию файлов и доступ к ним.
4. Перечислите операции над файлами.
5. Раскройте понятие каталоги файлов.
6. Опишите логическую структуру каталогов.
7. Перечислите операции над каталогами.
8. Опишите схему защиты файлов и каталогов.
9. Охарактеризуйте общую структуру ФС.

10. Опишите управление внешней памятью.

11. Раскройте понятие производительность ФС и охаракетризуйте его.

**Литература:**

1. А.П. Побегайло, Системное программирование в Windows, БХВ-Петербург, 2006, 1056 с.

2. Ахо, Лам, Сети, Ульман - Компиляторы. Принципы, технологии, инструменты, Изд.: Вильямс, 2008, 1185 с.

3. Л.Бек, Введение в системное программирование, М., Мир, 2008г.

4. Р. И. Компаниец, Системное программирование : Основы построения трансляторов, Изд.: СПб., 2000.

5. Ф.Льюис, Д.Розенкранц, Р.Стирнз. Теоретические основы проектирования компиляторов. - М.: Мир, 20099, 655 с.

6. Э. Таненбау, Современные операционные системы, Изд.: Питер, 2011, 1120 с.

7. А. В. Гордеев, А. Ю. Молчанов, Системное программное обеспечение, учебник для ВУЗов, СПб.: Питер, 2010 г.