Курс читал: доцент И.В.Машечкин, сентябрь-декабрь 1997г. ВМиК МГУ, 2 курс

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ. ЛЕКЦИИ

ЛЕКЦИЯ 1.

СТРУКТУРА И ОСНОВНЫЕ КОМПОНЕНТЫ ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ

Работая на машине, мы реально не работаем с микросхемами и “железом”, наша работа происходит с программным обеспечением (ПО), которое размещено на аппаратуре. Поэтому вводится понятие Вычислительной системы.

Вычислительная система – это программно-аппаратный комплекс, который предоставляет услуги пользователю.

Структуру вычислительной системы можно представить в виде пирамиды:

Аппаратные средства.

Ресурсы ВС разделяются на два типа:

–не участвующие в управлении программой (объем винчестера и т.д.).

–участвующие в управлении программой (размер ячейки памяти, объем оперативной памяти, скорость выполнения команд).

Ресурсы второго типа называются физическими ресурсами аппаратуры.

Управление физическими устройствами.

Управление физическими устройствами осуществляют программы, ориентированные на аппаратуру, взаимодействующие с аппаратными структурами, знающие "язык" аппаратуры.

Управление логическими устройствами.

Этот уровень ориентирован на пользователя. Команды данного уровня не зависят от физических устройств, они обращены к предыдущему уровню. На базе этого уровня могут создаваться новые логические ресурсы.

Системы программирования.

Система программирования — это комплекс программ для поддержки всего технологического цикла разработки программного обеспечения.

Прикладное программное обеспечение.

Прикладное программное обеспечение необходимо для решения задач из конкретных областей.

Операционная система (ОС) – программа, обеспечивающая взаимодействие пользователя с ВС, а также управляющая ресурсами ВС (логическими и физическими). К ОС мы будем относить второй и третий уровень нашей пирамиды.

Структура ЭВМ:

Основной функцией центрального процессора (ЦП) является обработка информации и взаимодействие с устройствами. Обмениваться данными ЦП может только с ОЗУ (Оперативно Запоминающее Устройство).

В ОЗУ размещается выполняемая в данный момент программа. ОЗУ состоит из ячеек памяти. Каждая ячейка имеет свой уникальный адрес, и каждая разбита на два поля: поле внутрисистемной информации (которое, например, может содержать бит четности) и машинное слово, содержащее команду или данные. Машинное слово состоит из некоторого количества двоичных разрядов, которое определяет разрядность системы.

ЦП выбирает из ОЗУ последовательность команд для выполнения. ЦП состоит из двух компонентов:

–Устройство Управления (УУ) принимает очередное слово из ОЗУ и разбирается – команда это или данные. Если это команда – то УУ выполняет ее, иначе передает АУ.

–Арифметическое Устройство (АУ) занимается исключительно вычислениями.

УУ работает с регистровой памятью, время доступа к которой значительно быстрее, чем к ОЗУ, и которая используется специально для сглаживания дисбаланса в скорости обработки информации процессором и скорости доступа к ОЗУ.

Лекция 2.

Мы определили, что вычислительная система (ВС) – это некоторое объединение аппаратных средств, средств управления аппаратурой (физическими ресурсами), средств управления логическими ресурсами, систем программирования и прикладного программного обеспечения.

Мы определили, что нижний уровень — это чисто аппаратура, это то, что делается из металла, пластика и прочих материалов, используемых для производства “железа”, или hardware, компьютера.

Следующий уровень — это программы, но программы, ориентированные на качество и свойства аппаратуры. Эти программы и разработчики этих программ досконально знают особенности управления каждого типа из аппаратных компонентов. Нижний уровень между физическим уровнем и аппаратурой — это интерфейс этого управления, т.е. некоторые наборы команд управления физическими ресурсами (каждое устройство имеет свой язык или свой набор команд управления).

Следующий уровень — уровень, ориентированный на сглаживание аппаратных особенностей. Он целиком и полностью предназначен для создания более комфортных условий в работе пользователя. Если, предположим, мы работаем с устройством внешней памяти “жесткий магнитный диск”, то параметры, которые характерны для конкретного диска, могут быть, например, такими: сколько считывающих и записывающих головок имеет это устройство, сколько поверхностей, на которых находится хранящий информацию слой. И, соответственно, набор команд управления этого устройства ориентирован на эти параметры. И конечно, вам, как программистам, не интересно работать в терминах “считать бит со второй поверхности десятого цилиндра такой-то дорожки”. Это тяжело и неэффективно. Уровень логических ресурсов создает некоторое обобщенное устройство, одно на всю систему, и пользователь работает в терминах этого обобщенного устройства. А уже программы логического уровня разбираются, к какой из программ управления физическими устройствами надо обратиться, чтобы запрос пользователя к логическому устройству правильно оттранслировать к конкретному физическому устройству.

Мы говорили о том, что в разных текстах либо два уровня управления — логический и физический, либо три — логический, физический и система программирования, относят к операционной системе. Мы будем считать операционной системой два уровня — логический и физический. Мы начали рассматривать основные свойства этой иерархии, которую объявили, и нарисовали достаточно простую и традиционную схему, или структуру, вычислительной машины.

Центральный процессор (ЦП) — это процессорный элемент, т.е. устройство, которое перерабатывает информацию, оперативная память (Оперативное Запоминающее Устройство, ОЗУ) и устройства управления внешними устройствами (УУВУ). Мы определили основное качество оперативной памяти: именно, в оперативной памяти лежит исполняемая в данный момент программа, и процессор все последующие команды исполняемой программы берет из оперативной памяти. Если чего-то не хватает, идет запрос к внешним устройству, информация подкачивается в оперативную память, и опять-таки из оперативной памяти команды поступают в процессор на обработку.

В принципе, внешнее устройство можно реализовать на оперативной памяти. Если вы знаете, есть такая замечательная программа, которая называется MS-DOS. Эта операционная система (хотя классически она не является операционной системой) имеет ограничения на размер используемой памяти 640Кб, а аппаратура реальных машин на сегодняшний день может иметь физическую оперативную память существенно больших размеров. В этой системе можно создавать логический диск, который размещается на оперативной памяти, т.е. по всем интерфейсам работа с ним будет осуществляться как с жестким диском, но размещаться он будет в оперативной памяти. Здесь разница в том, что из тех 640Кб процессор берет команды на исполнение, а из оставшихся, которые мы объявили логическим диском, не берет, потому что он будет работать с ним, как с обычным жестким диском или любым другим носителем.

Мы с вами начали более подробно говорить о процессоре и зафиксировали одну из основных проблем, которая имеет место быть в области вычислительной техники. Это несоответствие в скоростях доступа и обработки информации различных компонентов вычислительной системы, ведь у каждого компонента есть своя предыстория. Где-то это само по себе медленное устройство, где-то на его скорость влияет длина проводников, которые находятся между процессорным элементом и конкретным устройством. Для каждого случая причина своя. Но проблему в том, что реальная оперативная память на порядки более медленна, чем скорость обработки информации в процессоре. И тем более, зачем нам повышать производительность процессора, если доступ к памяти (а мы все время что-то берем из памяти, так как работа процессора это обработка информации, которую он берет из памяти) настолько замедлен. Очевидно, если ничего не будет сделано конструктивно, то скорость всей системы будет равняться скорости работы компонента, имеющего наименьшую скорость.

Регистры.

Мы начали смотреть, какие конструктивные решения есть в аппаратуре вычислительной системы, которые предназначены для сглаживания этого дисбаланса. И первое, о чем мы начали говорить, — это регистры. В процессоре имеются устройства, способные хранить некоторую информацию. К этим устройствам возможен доступ прямым или косвенным способом из программы, выполняемой на машине. При этом есть группа регистров, которые называются регистрами общего назначения, которые доступны из всех команд. Эти регистры могут обладать свойством хранения и обработки определенных типов данных — это могут быть вещественные данные, короткие целые данные, которые используются, предположим, для индексирования, это могут быть длинные целые данные. При этом скорость доступа к регистрам общего назначения соизмерима со скоростью обработки информации в процессоре. При умелом программировании можно использовать регистры общего назначения в целях сокращения числа обращений к оперативной памяти. Это означает, что торможение на участке процессор-оперативная память сокращается. Рассмотрим другие группы регистров.

Специальные регистры**.** К этой группе относятся две подгруппы регистров.

1)Первая подгруппа — это регистры, отвечающие за состояние исполняемой программы. К этим регистрам относится счетчик команд. Этот регистр содержит адрес исполняемой в данный момент команды. Это тот самый регистр, который можно изменять

только косвенно, передавая управление куда-то. Второй регистр из этой же подгруппы — регистр результата (flags), содержащий флаги результата выполнения последней команды. По значению этого регистра можно организовывать те или иные действия. К этой подгруппе относится также регистр указателя стека. Есть команды, которые работают со стеком. Эти команды обычно используются для программирования переходов из функции и в функцию. Стек в системе используется для передачи параметров и организации автоматической памяти. Автоматическая память занимается относительно вершины стека при входе в функцию, и, при выходе, она освобождается. Поэтому в автоматических переменных нельзя хранить данные после выхода из функции.

2)Вторая подгруппа регистров — это регистры управления компонентами вычислительной системы, или управляющие регистры. Практически в любой вычислительной системе имеются регистры, предназначенные для организации взаимосвязи процессора с внешним миром. Эти регистры связываются с УУВУ, и через эти регистры процессор может организовывать управление внешними устройствами. Например, если возьмем регистр управления жестким диском, то у него могут быть следующие поля:

a)Поле, указывающее, кому предназначена информация на этом регистре в данный момент времени (процессору или диску).

b)Если эта команда имеет формат “от процессора к устройству”, в нем может находиться код операции управления устройством, могут находиться некоторые операнды и т.д. Устройство пытается выполнить эту команду, и по результату ее выполнения возвращается результат так же в управляющий регистр (это может быть информация о том, что обмен закончен успешно, или что обмен не закончен и причина этого).

Система прерываний.

К средствам, управляющим взаимосвязью с внешними устройствами, можно отнести систему прерываний. В каждой вычислительной машине имеется предопределенный, заданный при разработке и производстве, набор некоторых событий и аппаратных реакций на возникновение каждого из этих событий. Эти события называются прерываниями. Аппарат прерываний используется для управления внешними устройствами и для получения возможности асинхронной работы с внешними устройствами. Синхронная работа осуществляется так — система говорит “Дай мне блок информации”, а затем стоит и ждет этого блока. Работа асинхронна, если система говорит “Принеси мне, пожалуйста, блок информации” и продолжает свою работу, а когда приходит блок, она прерывается (по прерыванию завершения обмена) и принимает информацию. Такова схема прерываний. Одним из прерываний, которые есть в системе, является прерывание по завершению обмена.

В момент возникновения прерывания действия в аппаратуре ВС следующие:

1)В некоторые специальные регистры аппаратно заносится (сохраняется) информация о выполняемой в данный момент программе. Это минимальные действия, необходимые для начала обработки прерывания. Обычно, в этот набор данных входит счетчик команд, регистр результата, указатель стека и несколько регистров общего назначения. (Эти действия называются малым упрятыванием).

2)В некоторый специальный управляющий регистр, условно будем называть его регистром прерываний, помещается код возникшего прерывания.

3)Запускается программа обработки прерываний операционной системы.

Запущенная программа в начале потребляет столько ресурсов (не более), сколько освобождено при аппаратном упрятывании информации. Эта программа производит анализ причины прерывания. Если это прерывание было фатальным (деление на ноль, например), то продолжать выполнение программы бессмысленно, и управление передается части операционной системы, которая эту программу выкинет. Если это

прерывание не фатальное, происходит дополнительный анализ, который приводит к ответу на вопрос: можно ли оперативно обработать прерывание? Пример прерывания, которое всегда можно обработать оперативно — прерывание по таймеру. А например, прерывание, связанное с приходом информации по линии связи нельзя обработать оперативно, т.к. происходит расчищение в системе места для программы операционной системы, которая займется обработкой этого прерывания. Происходит, так называемое, полное упрятывание. Теперь прячется не только информация о некоторых регистрах исполнявшейся программы — теперь все регистры сохраняются в таблицах системы (а не в аппаратных регистрах, как при малом упрятывании) и фиксируется то, что пространство оперативной памяти, занимаемое программой, может быть перенесено (при необходимости) на внешнее устройство.Далее следует обработка прерывания и возврат из него.

Здесь надо отметить одно важное свойство: прерывания могут быть инициированы схемами контроля процессора (например, при делении на ноль), могут быть инициированы внешним устройством (при нажатии клавиши на клавиатуре возникает прерывание, по которому процессор считывает из некоторого регистра нажатый символ).

Возвращаемся к нашей основной проблеме. ВЗУ на многие порядки более медленно, чем оперативная память, т.е. возникает проблема торможения ВС. Если бы все обмены с внешними устройствами происходили в синхронном режиме, то производительность ВС была бы очень низкой. Одной из причин появления аппарата прерываний была необходимость сглаживания скоростей доступа к внешним устройствам и к оперативной памяти (оперативная память здесь выступает как более высокоскоростное устройство). То, что, используя аппарат прерываний, можно было работать с внешними устройствами в асинхронном режиме, т.е. задать заказ на обмен и забыть о нем до прерывания завершения обмена, позволило в целом увеличить производительность ВС.

Регистры буферной памяти (Cache, КЭШ).

Следующая группа регистров — регистры, относящиеся к т.н. буферной памяти. Мы возвращаемся к проблеме взаимодействия процессора и оперативной памяти и сглаживанию скоростей доступа в оперативную память.

Предположим, у нас есть некоторая программа, которая производит вычисление некоторого выражения, при этом, процесс вычисления этого выражения будет представим следующим образом. В какие-то моменты идут обращения за операндами в оперативную память, в какие-то моменты обработанные данные записываются в оперативную память. Есть один из нескольких путей, которые сглаживают несоответствие скоростей процессора и оперативной памяти, который заключается в сокращении реальных обращений к оперативной памяти. Процессоры содержат быстродействующую регистровую память, призванную буферизовать обращения к оперативной памяти.

Алгоритм чтения из оперативной памяти следующий:

–Проверяется наличие в специальном регистровом буфере строчки, в которой находится исполнительный адрес, совпадающий с исполнительным адресом требуемого операнда. Если такая строчка имеется, то соответствующее этому адресу значение, считается значением операнда и передается в процессор для обработки (т.е. обращение в оперативную память не происходит).

–Если такой строчки нет, то происходит обмен с оперативной памятью, и копия полученного значения помещается в регистровый буфер и помечается исполнительным адресом этого значения в оперативной памяти. Содержимое операнда поступает в процессор для обработки. При этом решается проблема размещения новой строчки. Аппаратно ищется свободная строка (но она может быть только в начале работы машины), и если таковая не найдена, запускается аппаратный процесс вытеснения из этого буфера наиболее “старой” строчки. “Старость” определяется по некоторому предопределенному критерию. Например, признаком старения может бытьколичество

обращений к этому буферу, при котором нет обращений к этой строчке. В каждом таком случае число в третьем столбце таблицы увеличивается на единицу. Короче говоря, аппаратура решает, какую из строк надо вытолкнуть из таблицы, чтобы на ее место записать новое содержимое. При этом учитывается информация о том, были ли обращения к данной строке с использованием команд записи в память. Если такие обращения были, то перед выталкиванием происходит запись в ОЗУ по исполнительному адресу содержимого нашей строчки.

Алгоритм записи в оперативную память симметричен. Когда в программе встречается команда записи операнда в память, аппаратура выполняет следующие действия. Проверяется наличие в буфере строки с заданным исполнительным адресом. Если такая строка есть, то в поле “Содержимое” записывается новое значение и аппаратно корректируется признак старения строк. Если такой строчки нет, то запускается описанный выше процесс выталкивания, и затем информация размещается в освободившейся строке.

Этот буфер чтения/записи служит достаточно мощным средством для минимизации обращений к ОЗУ. Наибольший эффект достигается при небольших циклах, когда все операнды размещаются в буфере, и после этого циклический процесс работает без обращений к ОЗУ. Иногда эти буфера называют КЭШ-буферами, а также ассоциативной памятью, потому что доступ к этой памяти осуществляется не по адресу (как в ОЗУ), а по значению поля. Реально, все механизмы могут быть устроены иначе, чем мы здесь изучаем, т.к. мы изучаем некоторую обобщенную систему.

Оперативная память

Следующим компонентом, который мы с вами рассмотрим, с точки зрения системного подхода (а системный подход подразумевает то, что мы рассматриваем вещь не саму по себе, а в контексте взаимосвязи с другими компонентами) будут некоторые свойства ОЗУ.

1-й блок		2-й блок	. . .	k-й блок
0		1	. . .	k-1
k		k+1	. . .	2k-1
. . .		. . .	. . .	. . .

Использование расслоения памяти. Физически ОЗУ представимо в виде объединения k устройств, способных хранить одинаковое количество информации и способных взаимодействовать с процессором независимо друг от друга. При этом адресное пространство ВС организовано таким образом, что подряд идущие адреса, или ячейки памяти, находятся в соседних устройствах (блоках) оперативной памяти.

Программа состоит (в большей степени) из линейных участков. Если использовать этот параллелизм, то можно организовать в процессоре еще один буфер, который организован так же, но в котором размещаются машинные команды. За счет того, что есть параллельно работающие устройства, то этот буфер автоматически заполняется вперед. Т.е. за одно обращение можно прочесть k машинных слов и разместить их в этом буфере. Далее, действия с буфером команд похожи на действия с буфером чтения/записи. Когда нужна очередная команда (ее адрес находится в счетчике команд), происходит ее поиск (по адресу) в буфере, и если такая команда есть, то она считывается. Если такой команды нет, то опять-таки работает внутренний алгоритм выталкивания строки, новая строка считывается изпамяти и копируется в буфер команд. Расслоение памяти в идеале

увеличивает скорость доступа в k раз, плюс буфер команд позволяет сократить обращения к ОЗУ.

Лекция 3.

Виртуальная память, мультипрограммный режим

Мы продолжаем рассмотрение нашей упрощенной схемы. В современных машинах имеется еще одно аппаратное средство, которое призвано поддерживать работу вычислительной системы. Это, так называемая, виртуальная память. На сегодняшний день все вычислительные машины (за исключением особо раритетных) работают в мультипрограммном (мультипроцессорном) режиме. Суть его заключается в том, что у имеется несколько процессов, которые одновременно выполняются в вычислительной системе. Посмотрим, как этот мультипрограммный режим влияет на использование оперативной памяти.

Оперативная память:

Операционная система
Задача №1
Задача №2
Задача №3
…

Предположим: в начальный момент времени какую-то часть оперативной памяти заняла операционная система (так оно и происходит). После этого была загружена программа №1 , затем программы №2, №3 (и т.д.). Операционная система начала выполнять эти программы в мультипрограммном режиме. Возникает вопрос: всегда ли любую программу можно поместить в произвольный диапазон адресного пространства оперативной памяти? Обладает ли программа свойством перемещения по памяти? Насколько задача связана с адресным пространством, на которое ее запрограммировали? Это первая проблема. Вторая проблема: этот процесс идет, и понятно, что в какие-то моменты времени какие-то из этих задач заканчиваются (например, Задача №2). При этом в памяти образуются свободные фрагменты.

Оперативная память		Оперативная память
Операционная Система		Операционная Система
Задача №1		Задача №1
		Задача №6
Задача №3		Задача №3

Программа операционной системы, которая загружает задачи в память, может посмотреть, какие из задач ожидают обработки, и если есть задача, которая помещается в один из свободных фрагментов, она может ее загрузить. Но возникает второй вопрос: а как быть, если нет такой задачи, которая поместится в освободившийся фрагмент. При этом проблема заключается в том, что даже если найдется задача (№6), которая поместится в указанном фрагменте, то останется еще меньший фрагмент памяти, в который уже нельзя практически ничего записать. Количество таких, никому не нужных, фрагментов постепенно увеличивается. Это процесс фрагментации памяти.

Фрагментация может происходить и в оперативной памяти, и на внешнем запоминающем устройстве. В результате через некоторое время возникает достаточно интересная и грустная ситуация: свободного адресного пространства много, но при этом мы не можем загрузить ни одной новой задачи. Это означает, что система в целом начинает деградировать. Например, если Вы пошлете заказ на выполнение какого-то действия, то Вам придется неоправданно долго ждать.

Для борьбы с фрагментацией памяти, а также для решения проблемы перемещения программы по адресному пространству, используется, так называемая, виртуальная память. Суть ее работы заключается в следующем. Пусть имеется некоторое адресное пространство программы, то есть то адресное пространство, в терминах которого оперирует программа. И имеется адресное пространство физическое, которое зависит от времени. Оно характеризует реальное состояние физической оперативной памяти.

В машинах, поддерживающих виртуальную память, существует механизм преобразования адресов из адресного пространства программы в физическое адресное пространство, то есть при загрузке задачи в память машины операционная система размещает реальную задачу в той оперативной памяти, которая является свободной, вне зависимости от того, является ли этот фрагмент непрерывным, либо он фрагментирован. Это первое действие выполняет операционная система. Она знает о состоянии своих физических ресурсов: какие свободны, какие заняты.

Второе: операционная система заполняет некоторые аппаратные таблицы, которые обеспечивают соответствие размещения программы в реальной оперативной памяти с адресным пространством, используемым программой. То есть можно определить, где в физической памяти размещена какая часть программы, и какая часть адресного пространства программы поставлена ей в соответствие.

После этого запускается программа, и начинает действовать аппарат (или механизм) виртуальной памяти. Устройство управления выбирает очередную команду. Из этой команды оно выбирает операнды, то есть адреса и те индексные регистры, которые участвуют в формировании адреса. Устройство управления (автоматически) вычисляет исполнительный адрес того значения, с которым надо работать в памяти. После этого автоматически (аппаратно) происходит преобразование адреса исполнительного программного (или виртуального) в адрес исполнительный физический с помощью тех самых таблиц, которые были сформированы операционной системой при загрузке данной программы в память. И продолжается выполнение команды. Аналогично выполняется и, например, команда безусловного перехода на какой-то адрес. Точно так же устройство управления вычисляет сначала адрес исполнительный, после чего он преобразуется в адрес физический, а потом значение этого физического адреса помещается в счетчик команд. Это и есть механизм виртуальной памяти.

Рассмотрим простейший пример организации виртуальной памяти: вычислительную систему с, так называемой, страничной организацией памяти. Суть страничности памяти заключается в том, что аппаратно все адресное пространство оперативной памяти разделено на блоки фиксированного объема. Обычно размер таких блоков равен степени двойки. При этом сохраняется сквозная нумерация ячеек памяти.

Структура адреса в данной вычислительной машине такова: адрес любой ячейки памяти представлен в виде двух полей; старшие его разряды являются номером страницы, а младшие разряды являются смещением относительно страницы (здесь используется тот факт, что размер страницы равен степени двойки).

Исполнительный адрес

№ Страницы	Смещение относительно страницы

Итак, мы имеем машину со страничной организацией памяти. Для управления этой страничной памятью, процессор содержит, так называемую, таблицу приписки (ТП). Это аппаратное средство машины, реализованное с помощью регистровой памяти. Структура этой таблицы:

Таблица приписки

№ Виртуальной страницы	№ Физической страницы
0	25
1	1
2	30
.	.
.	.
.	.
L-1	-1

ТП имеет L строк, где L — максимальное число страниц, адресуемых в данной машине (то есть адресное пространство программы может состоять не более чем из L страниц). Каждая строка этой таблицы соответствует виртуальной странице программы с номером, совпадающим с номером строки. Строка содержит номер физической страницы, соответствующей данной виртуальной странице.

При загрузке и запуске программы операционная система размещает виртуальные страницы в некоторых физических страницах оперативной памяти, а их соответствие устанавливается в ТП. Например, операционная система взяла 0-ую виртуальную страницу памяти и поместила ее в 25-ую страницу физической памяти, 1-ую страницу разместила в 1-ой странице, а 2-ую — в 30-ой (и так далее распределила все страницы). После этого передается управления на начало программы, а за тем работает тот механизм, который был описан выше: по номеру виртуальной страницы из таблицы аппаратно выбирается соответствующая строка, и содержащийся в ней номер физической страницы аппаратно подставляется висполнительный адрес вместо номера виртуальной страницы, то есть происходит просто замена старших битов исполнительного адреса.

На самом деле, когда операционная система обращается к какой-то строчке таблицы, то сначала она производит контроль. В действительности, строка ТП содержит некоторый код, который является либо адресом физической страницы, либо некоторым специальным кодом, на который схемы управления процессора реагируют определенным образом. Например, если этот код больше либо равен нулю, это значит, что можно продолжать работу. Если же в этой строке содержится код меньше нуля, например -1, то обращение к ней устройств преобразования виртуального адреса в физический, вызовет

прерывание. Это прерывание обычно называется прерыванием по защите памяти. Дело в том, что операционная система, заполняя ТП, указывает, какие из виртуальных страниц адресного пространства не принадлежат данной программе, и если происходит выход за пределы дозволенной памяти, генерируется прерывание по защите памяти.

Для каждой программы нужна своя таблица приписки. При переходе от одной программы к другой содержимое ТП сохраняется операционной системой в некоторой своей программной таблице (массиве) и затем изменяются значения в ТП.

(Продолжение в теме “Операционные системы”: Подкачка, или SWAPPING)

Внешние устройства

Внешние устройства можно определить как все те устройства, которые отличаются от процессора и памяти.

Управление внешними устройствами осуществляется через систему прерываний. Внешние устройства можно подразделить на Внешние Запоминающие Устройства (ВЗУ) и Устройства Ввода/Вывода (УВВ) информации. ВЗУ — это устройства, способные хранить информацию некоторое время, связанное с физическими свойствами конкретного устройства, и обеспечивать чтение и/или запись этой информации в оперативную память. Если рассматривать ВЗУ с точки зрения использования различными компонентами программного обеспечения, то можно выделить следующие типы устройств:

1. Магнитный барабан. Магнитный барабан — это устройство, которое характерно для больших вычислительных комплексов. Обычно оно используется операционной системой для хранения системной информации. Суть работы этого устройства состоит в следующем.

Имеется металлический цилиндр большого веса (вес здесь имеет значение для поддержания стабильной скорости вращения), который вращается вокруг своей оси. Поверхность этого цилиндра покрыта слоем материала, способного хранить информацию (с него можно читать и на него можно записывать информацию). Над поверхностью барабана размещается *p* считывающих головок. Их положение зафиксировано над поверхностями, которые называются треками (track). Каждый трек разделен на равные части, которые называются секторами. В каждый момент времени в устройстве может работать только одна головка. Запись информации происходит по трекам магнитного барабана, начиная с определенных секторов. Координатами информации служат следующие параметры (№Трека, №Сектора и Объем информации).

Для чтения информации с магнитного барабана производятся следующие действия: включается головка, соответствующая номеру трека, и прокручивается барабан до появления под головкой начала сектора с заданным номером. После этого начинается обмен. Практически во всех ВЗУ, основанных на вращении носителя, существует понятие сектора, и в каждый момент времени устройство знает, над каким сектором оно находится. Магнитные барабаны — это устройства, имеющие одну из самых больших скоростей доступа, так как электронные и механические действия в его работе минимальны (вращение барабана).

2. Магнитные диски.

Имеется несколько дисков, размещенных на одной оси, которые вращаются с некоторой постоянной скоростью. Каждый такой диск может иметь две информационно-несущие поверхности (верхнюю и нижнюю), покрытые слоем, способным фиксировать информацию. Диски имеют номера; поверхности каждого диска также пронумерованы (0,1). Концентрическим окружностям одного радиуса на каждом диске соответствует условный цилиндр. Диск также разбит на сектора. Координаты информации на диске (№Диска, №Поверхности, №Цилиндра, №Сектора).

Механически управляемая штанга имеет щупы, на концах которых находятся считывающие и записывающие головки. Количество этих щупов может быть равно количеству дисков (считывается либо верхняя, либо нижняя поверхность).

Обмен информацией осуществляется следующим образом: на блок управления диском подается набор координат с требуемым объемом информации. Блок головок вводится внутрь диска между поверхностями до заданного номера цилиндра. Затем, включается головка, читающая заданную поверхность заданного диска. После этого ожидается подход заданного сектора и начинается обмен. Здесь, в отличие от магнитного барабана, уже два механических действия, что ухудшает скоростные свойства магнитных дисков. Примерами магнитных дисков являются винчестер и гибкие диски (floppy).

Лекция 4.

Мы с вами продолжаем обзор свойств архитектуры вычислительной системы. И вновь следует обратить внимание на тот факт, что мы с вами рассматриваем не просто ЭВМ, как набор плат, размещенных на ней микросхем, каких-то проводников, механических устройств и всего прочего, а мы рассматриваем вычислительную систему, то есть то образование, к которому мы уже привыкли — кто-то из вас привык к одному типу и считает, что это почти везде так — есть персональные компьютеры ибольше ничего. Нет, на самом деле вычислительной техники очень много, она разнообразна: она есть большая и маленькая, встроенная и т.д. И мы с вами рассматриваем это, как вычислительную систему — систему, объединяющую аппаратуру и программное обеспечение. Мы рассматриваем это объединение, как систему — как программное обеспечение, оказывает влияние на аппаратуру, и наоборот. Мы с вами уже посмотрели и поговорили на предыдущих лекциях о некоторых таких взаимосвязях, м.б. в некоторых местах я не обращал ваше внимание — сейчас обращаю:

–виртуальная память — это чистое средство аппаратуры, обеспечивающее поддержку программного обеспечения, для повышения эффективности ПО

–буферизация — аппаратное свойство, которое поддерживает программное обеспечение в целях повышения эффективности

–система прерываний — это также аппаратное средство, которое ориентировано на поддержку ПО. Она обеспечивает взаимодействие программ с внешним миром

И так далее.

Я еще раз замечаю, что цель наших лекций — это не изучение ОС Unix, которую мы будем рассматривать, и не изучение страничной организации памяти. Наша цель — понимание термина “система” в выражении “вычислительная система”, то есть понимание взаимосвязей внутри некоторого комплекса. Эти связи очень сильные и напутанные, как, например, уже рассмотренные механизмы буферизации или система прерываний. А если мы возьмем какую-нибудь сложную машину, у который не один процессорный элемент, а несколько процессорных элементов, которые работают на одну память, то у нас возникает еще более сложная проблема, например, с той же буферизацией памяти.

И то, что мы с вами рассматриваем на лекциях — это очень простой срез, необходимый для того, чтобы у вас сложилось концептуальное понимание работы системы.

Прошлую лекцию мы закончили рассмотрением некоторых типовых внешних запоминающих устройств. Мы рассмотрели такие устройства, как магнитный барабан (МБ) и магнитный диск (МД), выяснили их отличия. Самоцель наших лекций — не изучить работу магнитного барабана, а получить некий инструмент, который позволял бы сравнивать некоторые качества компонент вычислительной системы. Мы выяснили, что внешние ЗУ, такого класса, как МБ и МД, могут характеризоваться степенью проявления механических действий при обработке заказа на обмен. При обмене с МБ — вращение барабана, с МД — добавляется, помимо вращения блина МД, механическое действие по подводу считывающих головок. Чем больше механический движений, тем она медленнее.

Есть еще один экзотический вид памяти — **память на магнитных доменах.**

Суть этой памяти в следующем:

В магнетизме существует элементарная единица — магнитный домен, одна сторона этой частицы положительна, другая — отрицательна. Существует магнитный барабан, у которого есть треки и головки для чтения и записи. Сам барабан не вращается, а за счет некоторых магнитно-электрических элементов осуществляется перемещение по треку цепочки доменов. При этом каждый домен может быть однозначно ориентирован — это есть кодирование нуля и единицы.

Основанные на таком принципе память очень быстродейственна, за счет отсутствия механических действий. Но эти устройства достаточно специфичны, они используются во встроенных вычислительных системах, в частности, в американских шаттлах используется такой тип памяти.

Итак, мы рассмотрели ВЗУ с точки зрения их эффективности. Давайте посмотрим на них с точки зрения методов доступа.

Суть ЗУ заключается в том, что информация в нем записывается некоторыми блоками или записями. В некоторых устройствах размер блока или записи фиксирован. В других устройствах размер записи произволен и определяется начальным и конечным маркером, который можно программно записать на носитель этого устройства.

В контексте работы с записями и блоками ВЗУ можно разделить на два типа: устройства *прямого* и *последовательного* доступа. Разница заключается в том, что при прямом доступе можно считать или записать произвольный блок (например, компакт-диск, МБ, МД и др.), а при последовательном — чтобы добраться до определенной записи требуется просмотр всех предыдущих записей (например, аудио кассета, стриммеры и др.)

Это два взгляда на ВЗУ. Давайте посмотрим на ВЗУ с точки зрения управления.

На ранних стадиях устройство управления ВУ являлось некоторым интерфейсом в получении всех управляющих команд от процессора и передачи их конкретному внешнему устройству. Это означает, что ЦП должен был обрабатывать практически все действия, предусмотренные системой команд управления конкретного устройства. Несмотря на то, что при этом был уже реализован и использовался аппарат прерываний, позволяющий проводить асинхронный обмен, были достаточно большие потери за счет того, что ЦП должен был постоянно прерываться на выполнение небольших по размеру последовательностей команд для управления внешними устройствами. И появились, на самом деле это есть также вычислительные машины, специализированные устройства, которые называются каналами.

Канал — это специализированная вычислительная машина, имеющая внутри себя, процессорный элемент и необходимую память. Канал обычно имеет следующую структуру: у него имеется высокоскоростной канал с памятью основной машины, управляющий канал для взаимодействия с ЦП и имеется некоторое количество каналов для подключения внешних устройств.

Функция канала — выполнение макрокоманд, обеспечивающих ввод-вывод, то есть ЦП подает не последовательность команд (к примеру, такую как — включить шаговый двигатель МД, переместить головку на заданный цилиндр, дождаться нужного сектора, считать блок данных, проверить считанное и т.д.), а команду более общую (произвести обмен данными указанного объема с заданным устройством). Маленькие же команды (перемещение головок, ожидание сектора) выполняет уже канал. В некоторых случаях каналом может являться отдельная специализированная вычислительная машина.

Именно с этого место следует особо подчеркивать терминологию — *Центральный Процессор — это основной процессор, ибо помимо него может существовать множество вспомогательных процессоров, обеспечивающих работу с различными компонентами системы.* Каналы же обеспечивают обмен с внешними устройствами.

Таким образом каналы разгружают ЦП, при этом сами каналы могут быть достаточно интеллектуальны. Например, за счет собственной памяти в нем может быть организована буферизация, по аналогии с буферизацией при работе с ОЗУ. То есть информация может не сразу записываться на диск, а сосредотачиваться в некотором

буфере. Это позволяет минимизировать количество физических обращений к запоминающему устройству, а следовательно, повысить скорость работы с ним.

Соответственно, современные ЭВМ могут иметь множество таких каналов, которые помогают организовывать управление и забирают лишние функции у ЦП.

Это, наверное, все, что можно или нужно было рассказать о работе с внешними устройствами.

Все то, что было сказано — есть лишь небольшие ключики, которые позволяют:

–а) как-то классифицировать ВУ и понять, что хорошо, что плохо;

–б) постараться сформировать взгляд на все это, как на систему.

Мультипрограммный режим

Давайте посмотрим еще один внутрисистемный аспект, который призван продемонстрировать взаимовлияние программного обеспечения и аппаратуры друг на друга. Этот аспект связан с мультипрограммированием.

Начертим следующую диаграмму:

Она показывает работу ЦП во времени.

При счете задачи “1” в какой-то момент ЦП потребовались данные, которые находятся на ВЗУ. Формируется заказ на обмен, но данные сразу не поступают (из-за низкой скорости обмена). И какое-то время ЦП простаивает (разрыв в черте). Далее обмен завершился и программа продолжает выполняться. Но потомвозможен еще такой же запрос к ВЗУ, а потом еще и еще. В зависимости от типа или класса решаемой на вычислительной машине задачи таких простоев может быть до 99% времени выполнения программы, что приводит к неэффективности работы вычислительной системы. Поэтому было бы неплохо при наличии асинхронно работающих устройств в промежутке времени, когда одна программа не может выполняться (например, ожидает данных), запускать другую программу, которая будет “жить” по тем же правилам — она будет выполняться на ЦП, а если ей чего-то не будет хватать, то последует обращение на внешнее устройство с вытекающей отсюда паузой в ее работе. А в эту паузу можно запустить третью программу и т.д.

Режим работы ПО и аппаратуры, обеспечивающий одновременную обработку или одновременное выполнение нескольких программ называется мультипрограммным режимом. Изначально мультипрограммирование появилось в целях максимальной загрузки ЦП, поскольку когда-то это устройство было самым дорогостоящим. На сегодняшний день ЦП, пожалуй, одно из самых дешевых устройств.

Давайте посмотрим, что нужно от аппаратуры вычислительной системы для поддержания мультипрограммирования. Для начала перечислим те проблемы, которые могут возникнуть, начиная с того момента, когда помимо ОС и одной программы пользователя появилась еще одна программа пользователя.

Сначала программа была одна и “творила” все, что хотела — ошибка в программе приводила к ошибке в системе, но кроме самой программы по сути дела больше никто не страдал. А когда на машине появляется еще одна программа, то начинают возникать проблемы:

Кто-то взял и записал в пространство работающей программы свою информацию или

считал из пространства программы какую-то информацию, возможно конфиденциальную.

Итак, **первая проблема** — проблема защиты памяти. То есть в вычислительной системе должен быть реализован на аппаратном уровне механизм, обеспечивающий защиту адресного пространства программ от несанкционированного доступа других программ. Это означает, что будет в системе механизм, который будет при обеспечении доступа по указанному адресу проверять корректность этого доступа.

На прошлой лекции мы рассматривали механизм виртуальной памяти на примере страничной организации памяти. Таблица приписки, в которой номер строки — номер виртуальной страницы памяти, а содержимое — некоторый код, если он больше, либо равен нулю, то все хорошо — такая страница есть и она приписана некоторой физической странице, и происходит нужное преобразование адреса. Если же код меньше нуля, то это означает, что такой виртуальной страницы нету, и это означает, что срабатывает аппарат защиты памяти. А срабатывает он следующим образом — если код меньше нуля и происходит обращение к данной странице, то в системе автоматически возникает прерывание по защите памяти. При обработке этого прерывания ОС смотрит — действительно ли этого листа нету (действительно ли он чужой) — в этом случае ОС прерывает выполнение процесса, вызвавшего ошибку с соответствующей диагностикой защиты памяти. Может быть другая ситуация — какие-то страницы ещене загружены, и ОС у себя в таблицах отметила, что такая страница есть на самом деле и находится, например, на внешнем устройстве. В этом случае прерывание игнорируется, так как нет достаточной информации для продолжения программы.

Вторая проблема. Пусть, например, имеет место в системе устройство печати. И есть два процесса, сначала один процесс обращается к устройству вывода — напечатать строку из какой-то таблице, которая должна быть выведена на печать, а другой процесс начинает печатать другой документ и тоже обращается к устройству печати. В итоге получается документ, представляющий из себя набор бесполезных строк. Другой пример — таблица приписки в ОС заполняется программно, предположим, что два процесса приписали себе одинаковые страницы памяти — в результате получается неразбериха.

Проблема же заключается в том, что в обоих примерах пользователю были доступны команды управления устройствами (в первом случае — ВУ, во втором — ОЗУ). И вроде как, исходя из наших примеров, это нехорошо. И действительно — организовать процесс мультипрограммирования в таких условиях будет тяжело, ибо какие-либо программы могут оказаться несогласованными, а это в свою очередь приведет к краху системы.

Итак, вторым условием существования мультипрограммирования — наличие привилегированного режим в системе. Это такой режим, в котором программе доступны все возможные команды ЦП. Соответственно, непривилегированный (пользовательский или математический) режим — это такой режим, в котором программе доступна лишь часть команд ЦП, и программы пользователя не смогут обратиться к некоторым функциям напрямую.

Обычно в привилегированном режиме работает ОС. Она же является посредником между программами и устройствами. То есть программа формирует запрос, где параметрами служат конкретные описания того, что следует сделать. В частности, обращение к внешним устройствам проходит по данной схеме — через обращение к ОС.

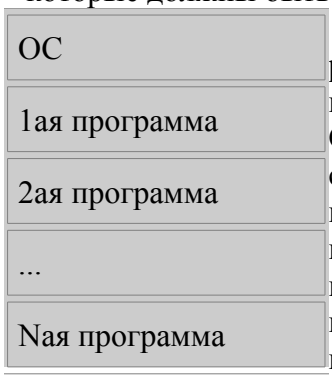
Таким образом, вывод на печать является не прямым обращением к устройству печати, а является обращением к ОС с заказом напечатать некоторую строчку. ОС принимает заказ и строку и буферизует информацию в некоторых своих программных буферах. То есть все, что мы пожелаем напечатать, будет аккумулировано в буферах ОС. Распечатка этого буфера будет происходить только когда будет ясно, что заказов на печать от конкретной задачи больше не поступит. То есть печать производится последовательно. Это еще один элемент связи между аппаратным и программным обеспечением.

Третья проблема. Имеется мультипрограммный режим. И в одной из программ появилась ошибка — программа зациклилась. Система повисла, ни одна программа не может работать. Что в этом случае может помочь? Помощь может оказать некоторое средство, которое будет переодически прерывать выполнение программ. Как минимум, это должно быть прерывание по таймеру — чтобы раз в некоторый диапазон времени управление передавалось в ОС. А далее уже должна реагировать сама ОС. А вот наличие

такого прерывание и есть третье необходимое условие организации мультипрограммного режима.

Вообще при наличии прерывания по таймеру можно вообще отказаться от наличия асинхронной системы прерываний.

Итак, мы перечислили три условия на аппаратуру вычислительной системы, которые должны быть выполнены для организации мультипрограммного режима работы.



Следующая проблема — в системе, которая работает в МП режиме, находится много программ, обрабатываемых ЦП. Может возникнуть ситуация (сходная с проблемой виртуальной памяти). Существует некоторая физическая память, пусть она будет страничная. Если работает режим мультипрограммирования, то какую-то часть памяти занимает ОС, какую-то часть занимает первая программа, вторая и т.д. В простейшем случае в системе может находиться и обрабатываться столько программ,сколько позволит разместить в себе оперативная память. Но это достаточно неэффективно, потому что все равно выполнение программы локализовано в нескольких виртуальных страницах, и выполнение перемещается не так быстро, то есть могут быть, организованы какие-то циклы в пределах одной страницы, обращение к небольшим функциям и т.д. Это означает, что если мы будем размещать в ОЗУ весь код и данные, то большая часть ОЗУ будет простаивать. Поэтому имеется естественное желание в ОЗУ держать только те фрагменты кода и данных, которые в настоящий момент используются. А для этого используется *аппарат подкачки (swapping)*. Это аппаратно-программное средство, суть которого заключается в следующем — в таблице приписки виртуальной памяти, в каждой строке может быть еще одно поле — поле, которое характеризует частоту обращения к странице. Это поле аппаратно формируется. То есть также, примерно, как собирается информация о “старении” информации в буфере ОЗУ, собирается информация об обращениях к данной странице в виртуальной памяти. По этим данным ОС может откачивать страницы, принадлежащие какому-либо процессу во внешнюю память. Например, после загрузки и некоторого времени выполнения программы, ОС начинает те страницы, к которым количество обращений было минимальным, сбрасывать на внешний носитель (на больших машинах обычно для этого используются магнитные барабаны). При этом, если страница откачена, то в таблицу приписки записывается отрицательное значение, и при возникновении обращения к этой строке (и, соответственно, возникновении прерывания) ОС сначала смотрит — а не есть ли это обращение к своей памяти, но которая откачена. Если это есть такое обращение, то ОС устанавливает заказ на подкачку этой страницы обратно, а пока это происходит, программа ожидает подхода информации, и в это время выполняются какие-то другие программы.

Такой механизм подкачки достаточно эффективен, он позволяет в ОЗУ от каждой из выполняемых программ держать незначительную часть. А когда в обработке находится большое число программ, то всегда при подкачке одной программы может выполняться другая.

С одной стороны подкачка поддерживается аппаратурой (подсчет “активности” страниц памяти сложно организовать программно), с другой стороны — все реальное управление программно.

Вот то, о чем можно сказать, рассматривая взаимодействие аппаратных и программных средств.

Теперь, если мы вернемся к нашей пирамиде. То мы видим, что все, о чем говорилось в предыдущих и этой лекции — взаимосвязано. То есть нельзя сказать, что в системе есть мультипрограммный режим, если нет аппаратной поддержки этого режима. И наоборот — при наличии аппаратных средств и отсутствии программ, использующих эти средства, вряд ли можно будет чего-либо добиться.

Это относится почти ко всему, за исключением, наверное, буферизации памяти между ОЗУ и ЦП — это практически полностью аппаратное средство.

Одна из наших целей — это пытаться видеть подобное взаимодействие между различного рода компонентами системы.

Вторая же проблема, которую мы всегда видели — это буферизация и сглаживание различных скоростей работы различных компонент. Мы говорили о сглаживании ОЗУ-ЦП и ВЗУ-ОЗУ.

Это, пожалуй, все, что относится к первой теме.

Лекция 5

Операционная Система (ОС)

Мы начинаем блок тем, называемых “Операционная система.”

Назначение и основные функции ОС

В принципе каждый из вас на сегодняшний день имеет некоторое представление об операционных системах. Но реально под ОС мы будем понимать комплекс программ, функциями которого является контроль за использованием и распределением ресурсов вычислительной системы. Мы с вами говорили о том, что в вычислительной системе есть физические ресурсы, то есть ресурсы, так или иначе связанные с реальным оборудованием, которое существует на самом деле — магнитные диски, оперативная память и прочее. Мы говорили о том, что в системе существуют логические ресурсы, то есть ресурсы, которые не имеют физического воплощения, но они существуют в виде некоторых средств, представляемых пользователю.

Все это мы будем называть *ресурсами вычислительной системы.*

Любая ОС оперирует некоторыми сущностями, которые во многом характеризуют свойства ОС. К таким сущностям могут относиться понятия файла, процесса, объекта и т.д. Конкретно, каждая ОС имеет свой набор этих сущностей, к примеру, если мы возьмем ОС WinNT, то к таким сущностям можно отнести понятие объекта ОС, и уже через это понятие, через управление этими ресурсами, предоставляются все возможные сервисные функции ОС. Если мы посмотрим на ОС Unix, то там такой сущностью является в первую очередь понятие файла, во вторую очередь — понятие процесса. И, возвращаясь, к последнему, процесс — это некоторая сущность (мы не говорим “объект” дабы не было путаницы с ООП), которая присутствует практически во всех ОС. Давайте попробуем определить понятие процесса, как одного из фундаментальных.

Итак, *процесс — это программа, имеющая права собственности на ресурсы*.

Рассмотрим две программы (код-данные) и рассмотрим все те ресурсы, которые принадлежат программе (пространство ОЗУ, данные на ВЗУ, права владения линией связи и т.п.). И если у двух программ множества ресурсов, принадлежащих им, совпадают, то в этом случае мы не можем говорить, что эти две программы образуют два процесса. Это один процесс.

Если же у каждой программы есть свое множество ресурсов, причем эти множества могут пересекаться (но не совпадать!), то мы говорим о двух процессах. Соответственно, в том случае, если множества ресурсов двух или нескольких процессов имеют непустое пересечение, то возникает проблема использования разделяемых ресурсов. Частично об этом было сказано на прошлой, когда приводился пример об одновременном использовании устройства печати двумя пользователями. У нас может быть масса процессов, каждый из которых имеет устройство печати одним из своих ресурсов. Раньше мы смотрели на этот пример с одной стороны, сейчас же мы смотрим на него с точки зрения синхронизации взаимодействия различных процессов. И эта синхронизация на примере устройства печати иллюстрировала нам одну из функций ОС, заключающуюся в управлении функционирования процессов.

Давайте посмотрим, что понимается под “управлением процессами”. Это достаточно просто:

1. Управление использованием времени ЦП. Планирование ЦП.

В какой момент времени, какая из задач будет использовать время ЦП.

2. Управление подкачкой и буфером ввода процесса.

Предположим, что запущено множество процессов, в результате в системе образовалась масса задач, превышающая возможности обработки вычислительной системы. В этом случае образуется буфер, в котором процессы ожидают начало своей обработки. Возникает проблема очередности выбора из этого буфера процессов или задач.

При выполнении нескольких процессов может возникнуть необходимость освободить пространство в памяти для помещения туда кода или данных какой-либо задачи. В этом случае возникает проблема откачки на внешний носитель части других процессов. Возникает проблема — покакому алгоритму их откачивать? Чтобы процесс откачки был наиболее эффективным.

3. Управление разделяемыми ресурсами.

Когда у разных процессов имеется непустое пересечение ресурсов. То есть имеется набор ресурсов, доступ к которым в произвольный момент времени, организуется со сторон различных процессов (это та самая коллизия с устройством печати). И одной из реально сложных функций, которые определяют свойства ОС, это есть функция, обеспечивающая организацию взаимодействия процессов и использования общих ресурсов. Но помимо тривиальных вещей (устройство печати, МД и др.), доступ к которым разделяет почти любая ОС, существуют более сложные проблемы — например, когда два процесса имеют общую оперативную память. И управление таким разделяемым ресурсом — проблема.

Теперь давайте посмотрим на структуру ОС. Практически любая ОС имеет понятие ядра, ядра операционной системы. *Ядром обычно является резидентная часть ОС (то есть та часть, которая не участвует в свопинге), работающая специализированном режиме (том самом, о котором говорилось в прошлой лекции).* В функции ядра входит базовые средства управления основными сущностями ОС, обработка прерываний, а также может входить набор программ, обеспечивающих управление некоторыми физическими устройствами. (Мы иногда будем называть программы, управляющие физическими устройствами — драйверами устройств). Например, в ядро ОС может и должен входить драйвер ОЗУ. Далее вокруг ядра наращиваются программы управления ресурсами вычислительной системы. Первый уровень в основном состоит из программ управления физическими устройствами или драйверов физических устройств. Следующий уровень — это управление логическими устройствами. И так далее — таких уровней может быть достаточно много. Просто, чем дальше от ядра, тем большая абстрактность имеет место. У нас где-то могут появиться драйверы управления файлами, а на самом деле они связаны с драйверами управления логическими дисками, а те в свою очередь — с драйверами физических дисков.

Следует отметить следующий факт — что вовсе необязательно все компоненты ОС работают в режиме супервизора (в режиме операционной системы). Это первый тезис. Второе — вовсе необязательно все части ОС расположены резидентно в ОЗУ. Наоборот, многие из функций ОС реализуются в режимепользовательской программы. Для них режима супервизора не требуется.

Давайте теперь перейдем к более подробному рассмотрению основных функций ОС.

1. Управление использованием времени ЦП.

На самом деле от того, какой алгоритм выбора очередной задачи, для передачи ей активности ЦП, определен, зависят многие реальные эксплуатационные свойства ОС. А

выбор алгоритма почти целиком и полностью определяется теми критериями эффективности, которые используются для оценки эффективности работы ОС. И поэтому управление ЦП мы рассмотрим на фоне типов ОС.

Первая ситуация: у нас есть большое число задач, требующих большого объема вычислительных мощностей системы. И эти задачи должны выполняться в одной вычислительной системе. Что будет являться критерием эффективности при работе системы для выполнения данных задач. Какой набор параметров можно взять и сказать, что если они маленькие — то хорошо, или если они большие, то хорошо или наоборот. Для такой ситуации критерием эффективности является степень загрузки ЦП. Если ЦП простаивает мало (либо ОС сама загружает процессор), то есть если потерянное время маленькое, то такая система работает хорошо и эффективно. Как добиться такой работы? С помощью хорошего алгоритма планирования. Мы запускаем для обработки тот набор задач, который у нас есть. То есть в системе работает режим мультипрограммирования. Алгоритм передачи ЦП от одного процесса к другому будет следующий — если ЦП выделен одному из процессов, то он будет владеть ЦП до одного из следующих моментов:

1)обращение к внешнему устройству

2)завершение процесса

3)зафиксированный факт зацикливания процесса

Как только наступило одно из перечисленных событий — ЦП передается следующему процессу. Количество таких передач минимизировано и, соответственно, так как при передаче с одного процесса на другой ОС должна совершить ряд действий (загрузка таблиц приписки и прочее). Здесь эти потери минимизированы. Такой режим работы ОС называется пакетным. А сама ОС — пакетной.

Теперь рассмотрим следующий пример. Несколько человек сидят за терминалами и работают в текстовых редакторах. ОС — пакетная. Тогда, если один человек не будет нажимать клавиш, то время ЦП будет “висеть” у него (обмен не выполняется, не зациклен и не завершен), а другие пользователи будут обязаны ждать. То есть общая производительность системы несоизмеримо падает. Это означает, что алгоритм, который хорош для пакетной системы, здесь неприменим. Поэтому для задач, связанных с подобными режимами работы нужен другой алгоритм разделения времени. Какой? Что будет являтьсяпризнаком того — хорошо система работает или плохо? В качестве критерия эффективности в данной ситуации может использоваться время ожидания реакции на какое-либо событие (нажатия на клавишу и соответственной реакцией на это текстового процессора, например). Чем эффективнее работает система, тем среднестатистическое время ожидания — меньше.

Рассмотрим прежнюю ситуацию, когда в обработке находится сразу несколько процессов. И наша задача распределить время ЦП таким образом, чтобы время реакции системы на запрос пользователя было минимальным, либо хотя бы гарантированным. И вот следующая схема: в системе используется некоторый параметр (dt), который называется квантом времени. В общем случае квант времени это есть некоторое значение, которое может меняться при настройке системы. Далее, все множество процессов, которое находится в мультипрограммной обработке, делится на два подмножества.

Первое подмножество — это те процессы, которые еще не готовы к продолжению выполнения (ожидающие обмена, подкачки и т.п.).

Второе — которые готовы к выполнению. В этом случае работа будет осуществляться по правилу: тот процесс, который владеет ЦП, будет выполняться до тех пор, пока не наступит одно из трех событий:

1)запрос на обмен

2)завершение процесса

3)исчерпание выделенного процессу кванта времени

При наступлении одного из этих событий планировщик ОС выбирает из процессов один, готовый к выполнению, и передает ему ресурсы ЦП. Как выбирает? Это уже

некоторые нюансы реализации алгоритма планирования. Процесс может выбираться произвольно, последовательно, по времени, которое данный процесс не владел ЦП (ОС может выбирать тот, у которого это время наибольшее).

Все эти алгоритмы реализуются в ОС и все они должны быть максимально простыми, дабы система не загружалась на их выполнение.

ОС с перечисленными выше свойствами называются ОС разделения времени.

У пользователя такой ОС должна создаться иллюзия, что все ресурсы системы предоставлены только ему.

Давайте рассмотрим следующее. Предположим, у нас есть самолет, который идет на снижение. Вы, наверное, знаете, что у самолета имеется прибор, который меряет высоту от поверхности земли до самолета. Самолет находится в режиме автопилота. Во время снижения автопилот с помощью указанного прибора отслеживает высоту. Но помимо этой задачи, он выполняет еще контроль за двигателями, остатком топлива, герметизацией и т.д. Для каждого из этих действий выделен отдельный процесс. Предположим, у нас имеется пакетная система. Мы внимательно контролируем в данный момент содержимое топливных баков... Как-то не складывается... А предположим, у нас система разделения времени. Одним из качеств такой системы является ее неэффективность, так как время ЦП расходуется на большое количество переключений между процессами — а это функция достаточно трудоемкая. Получаем в результате, что у нас высота уже подходит к нулю, а ОС занимается восстановлением таблиц приписки... Нехорошо получается... То есть для анализа каких-то ситуаций в один случаях наше dt годится, а в других — нет.

То есть для подобных задач нужны свои средства планирования. И в этих случаях используются так называемые ОС реального времени, основным критерием которых является время гарантированной реакции системы на возникновение того или иного события из набора заранее предопределенных события. То есть в системе есть события, которые будут обработаны в любой ситуации за некоторое наперед заданное время. Для нашего примера такими событиями могут быть показания прибора высоты, работы двигателей и т.д.

Реально для ОС этого класса используются достаточно простые алгоритмы планирования.

Подводя некоторую черту под этой первой функцией управления временем ЦП, следует обратить внимание на два факта:

–те алгоритмы, которые организованы в системе планирования времени ЦП, во многом определяют эксплуатационные свойства вычислительной системы.

–мы с вами рассмотрели три типовых системы (пакетные, разделения времени, реального времени). На сегодняшний день можно говорить о том, что система реального времени — это отдельный класс ОС. Например, Win95, семейство Unix и многие другие не являются системами реального времени. Система реального времени — отдельная ОС. Первые две ОС можно с эмулировать. Часто же общеупотребимые ОС являются смежными, то есть в них есть как пакетные средства, так и средства разделения времени. Примером такой схемы организации планирования может быть следующее — планировщик является двухуровневым (в системе могут быть счетные и интерактивные задачи), он определяет сначала — какому классу задач отдать приоритет, а на втором уровне он в пределах одного класса выбирает задачу на выполнение. Если в данный момент нет ни одной готовой к выполнению интерактивной задачи, то ЦП передается счетным задачам, но с условием — как только появляется хотя бы одна интерактивная задача — счетная задача прерывается и управление передается интерактивной задаче.

Это все, что касается первой функции.

2. Управление подкачкой и буфером ввода.

Здесь алгоритмы планирования также нужны, но они не столь критичны. В общем-то в реальных системах совмещается буфер подкачки и буфер ввода процесса. Это первое замечание. Второе замечание. Современные ОС достаточно “ленивые” и откачку они зачастую осуществляют не блоками памяти, а целыми процессами. Здесь возникает два вопроса — критерий замещения процесса в ЦП и критерий выбора процесса из буфера для обработки. Самый простейший вариант заключается в использовании времени нахождения в том или ином состоянии. То есть в первом, если мы решаем вопрос об откачке процесса из активного состояния в область подкачки, то мы можем взять тот процесс, который дольше всего находится в состоянии обработки. Обратный процесс может быть также симметричен. То есть из области подкачки мы можем брать тот процесс, который дольше всего находится на буфере ввода.

На самом деле это очень простые алгоритмы, и они могут видоизменяться из соображений различных критериев в той или иной ОС. Один из критериев — все задачи могут быть подразделены на несколько категорий (например, задачи ОС могут всегда выполняться в первую очередь).

3. Управление разделяемыми ресурсами.

И в двух словах о третьей функции — об управлении разделяемыми ресурсами. Здесь будет обозначена только проблема, ибо решения мы с вами будем рассматривать уже конкретно на примере ОС Unix. Предположим имеется два процесса, которые работают на общее пространство ОЗУ. У каждого процесса есть область ОЗУ, которая одновременно принадлежит и другому процессу. Здесь имеется целый букет проблем. При этом разделяемые ресурсы могут работать в разных режимах. Например, два процесса могут работать на разных машинах, но при этом могут быть связаны общим полем ОЗУ. В этом случае возникает проблема с буферизацией с работой памяти, так как на каждой из машин есть свои механизмы буферизации. И может возникнуть ситуация, когда состояние физической памяти не соответствует реальному ее содержимому. И возникают проблемы работы такой системы. Следующая проблема — два процесса работают на одной машине, проблем с буферизацией нет, но должны быть какие-то средства синхронизации разделяемой памяти, иначе такая организация бессмысленна, то есть позволят создать условия, при которых обмен каждого из работающих процессов с разделяемой памятью будет проходить корректно, а это означает то, что при каждом чтении информации из разделяемой памяти должно быть гарантировано то, что все те, кто хотел что-то прочитать или записать и начал это делать, уже этот процесс завершили, то есть должна быть синхронизация по обмену с разделяемой памятью.

Реально зачастую не требуется при решении задач таких разделяемых ресурсов, как общая память. Но хотелось бы, чтобы процессы, которые функционируют одновременно могли оказывать влияние друг на друга, аналогичное аппарату прерываний. Для реализации этого во многих ОС имеется средство передачи между процессами сигналов, то есть возникаетнекоторая программная эмуляция прерывания. Один процесс передает сигнал другому процессу, и в другом процессе происходит прерывание и передача управления на некоторую предопределенную функцию, которая должна обработать этот сигнал.

Было обращено внимание на те функции, которые влияют на эксплуатационные функции ОС. Реально же функций у ОС гораздо больше.

Лекция 6

На прошлой лекции мы с вами начали рассматривать определения, состав, основные функции ОС. Мы определили, что ОС это комплекс программ, обеспечивающих контроль за распределением и использованием ресурсов вычислительной системы, что практически каждая ОС оперирует некоторым набором базовых сущностей. И одной из главных таких сущностей является процесс.

Было определено, что процесс, это есть нечто, обладающее ресурсами вычислительной системы. И мы приводили примеры процессов, которые могут иметь определенный набор ресурсов системы, возможно, с некоторыми пересечениями. Соответственно, мы определили, что процесс и программа это не одно и то же. Программа может состоять из нескольких процессов.

В общем случае, в рамках одного процесса может быть реализовано несколько программ. Мы еще будем говорить об этом, а сейчас главное понять, что процесс и программа — это две большие разницы.

Мы говорили о том, что одной из проблем ОС является проблема распределения времени ЦП между конкурирующими за это время процессами. При этом на концептуальном уровне были рассмотрены алгоритмы распределения времени между процессами и влияние этих алгоритмов на реальные эксплуатационные характеристики ОС.

Мы рассмотрели, что есть пакетная ОС и какие алгоритмы отдачи времени ЦП приемлемы для пакетной ОС, было сказано про ОС разделения времени.

В подавляющем большинстве случаем мы работаем в режиме разделения времени.

Предварительно мы говорили, что любая ОС реализует в себе буферизацию ввода/вывода. Буферизация в ОС — есть одна из основных функций. И по аналогии борьбы с разными скоростями доступа к различным компонентам вычислительной системы, ОС вводит в своих пределах программную буферизацию запросов на ввод/вывод, которая также решает проблемы сглаживания времени доступа, а также проблемы синхронизации в целом. Синхронизацию в целом можно пояснить примером с разделением устройства печати. А сглаживание доступа заключается в том, что практически каждая современная ОС имеет кэш-буфера, которые аккумулируют обращения к ВЗУ. То есть суть такова, что также как и при работе ОЗУ, реальные обмены могут не происходить, этот механизм организован похоже, за тем исключением, что реализован он программно, в отличие от аппаратной реализации буферизации ОЗУ. Это позволяет существенно оптимизировать ОС. Признаком наличия такой буферизации является требование прекратить выполнение ОС перед завершением работы вычислительной системы (например, MS DOS такого требования не предъявляет — компьютер под его управлением можно включать и выключать в любое время без потерь, а в таких системах, как Unix, Win95 и других, подобное выключение машины может привести к потере информации, так как в момент выключения часть данных может находиться еще в буфере.) Степень этой буферизации определяет реальную эффективность системы.

Приведем пример. Если поменять процессор 486 на Pentium в системе, где работает Win95, то быстродействие системы увеличится ненамного. Это означает, что быстродействие системы упирается в работу с внешними устройствами, механизм буферизации неэффективен — реальных обменов с внешними устройствами остается также много. В то время как ОС Unix, при такой же замене ЦП, дает качественный скачок вперед по быстродействию, так как в этой ОС, буферизация организована значительно лучше.

Мы с вами говорили сегодня и в прошлый раз о том, что каждая ОС оперирует с некоторыми сущностями, и одна из них — это процесс. Есть и вторая сущность, которая также важна, и во многих ОС она занимает главенствующую роль. Эта сущность — понятие файла.

Файловая система — это компонент ОС, обеспечивающий организацию создания, хранения и доступа к именованным наборам данных. Эти именованные наборы данных называются файлами.

Свойства файлов

1)Файл — это некий объект, имеющий имя и позволяющий оперировать с содержим файла, через ссылку на это имя. Обычно имя — это последовательность букв, цифр и разделителей длины, которая зависит от конкретной ОС.

2)Файл независим от расположения. Для работы с файлом не требуется информация о месторасположении данных на ВЗУ.

3)Защита файлов. На самом деле многие стратегические решения повторяются как на аппаратном уровне, так и на уровне ОС. Если мы вспомним мультипрограммный режим, то одной из проблем была возможность защиты памяти. Файловая система может также, как и ОС быть однопользовательской, либо многопользовательской. Во втором случае мы имеем ситуацию, когда возможно обеспечить корректную работу нескольких пользователей (аналогично мультипрограммному режиму). Примером некорректной работы могут служить MS DOS, MS Windows — при ошибке в одном процессе могут быть испорчены и сама ОС и соседние процессы. Корректная работа означает, что при организации многопользовательской системы никто не может испортить чужой процесс или файл. Многопользовательская файловая система означает защиту от несанкционированного доступа к информации файлов. Одним из атрибутов файловых систем является атрибут защиты. Вообще, реальная ОС должна обеспечивать защиту не только файлов, но и всех ресурсов, принадлежащих процессам, запущенным от имени одного пользователя.

Практически каждая ОС обеспечивает набор функций ввода/вывода для работы с файлами.

Обычно этот набор состоит из следующих функций:

–открыть файл для работы в данном процессе. Открыть файл можно как и существующий, так и новый. Зачем открывать файл, если можно сразу по имени читать и писать? На самом деле — открытие файла — средство, чтобы сообщить системе, что процесс будет работать с данным файлом. А ОС уже будет решать — давать доступ к файлу другим процессам или что-то еще.

–чтение/запись. Обычно обмен с файлами организуется некоторыми блоками данных. Причем этот блок данных, которым происходит обмен, несет двоякую функцию. С одной стороны — для любой вычислительной системы известен размер блока, наиболее эффективного для обмена (некоторые программно-аппаратные размеры, зависящие от конкретных устройств). С другой стороны эти блоки данных при реальном обмене могут варьироваться достаточно произвольно программистом. И в функции чтения/записи обычно фигурирует размер блока данных для обмена и количество блоков, которые следует прочесть или записать. От выбранного размера блока может зависеть эффективность программы. Например, эффективный размер — 256kb, а программы выполняет обмены по 128kb, имеет место неэффективность (количество обращений будет больше), этот процесс может быть сглажен ОС, если у нее хорошо организована буферизация. Плюс к этому с каждым открытым файлом связывается понятие файлового указателя. Этот указатель по аналогии с регистром счетчика команд в процессе в каждый момент времени показывает на точку внутри файла:

Указатель: Файл:

	Блок 0
->	Блок 1
	...
	Блок N-1

После каждого обмена указатель показывает на следующий блок (относительный адрес по файлу), с которым можно произвести обмен. Это означает, что если в какой-то

момент мы из какой-то позиции читаем блок данных, то указатель переносится на следующий блок. То же самое происходит и при записи.

–управление файловым указателем. Понятно, что для организации работы с файлом требуется уметь управлять файловым указателем. Чтобы можно было добавить данные в файл (переместив указатель на конец файла, начать писать). Такая функция позволяет перемещать указатель произвольным образом в пределах доступного.

Указатель, это некоторая переменная, доступная для программы. Она создается в момент открытия файла.

–закрытие файла. Данная функция может быть в двух вариантах — закрыть и оставить содержимое в файле, и вторая — уничтожить файл. По закрытию все связи с файлом обрываются и файл приходит в некоторое канонической состояние.

Структура файловой системы

Давайте рассмотрим некоторое пространство ВЗУ и рассмотрим, как мы можем организовать файлов в пределах этого пространств.

Одноуровневая организация файлов непрерывными сегментами.

Ее суть такова — в пределах пространства ВЗУ выделяется некоторая область для хранения данных, которая называется “Каталог”. Каталог имеет следующую структуру

Каталог		
Имя	начальный блок	конечный блок
...	...	...
...	...	...

Имя — имя некоторого файла

Начальный блок — относительный адрес блока, с которого начинается файл.

Конечный блок — относительный адрес последнего блока файла.

Соответственно, функция открытия файла сводится к тому, чтобы в этой таблице “Каталог” найти по имени нужный файл и определить начало его размещения и конец размещения. Действие очень простое, так как дополнительных обменов не требуется — весь каталог можно хранить в памяти ОС. Если создается новый файл, то он записывается на свободное место. Альтернативно каталогу имен существует таблица свободных фрагментов пространства ВЗУ.

С открытием файла все просто — по имени сразу получается диапазон размещения файла, причем реально данные могут занимать меньше места, чем диапазон начала-конца.

Чтение-запись происходят практически без дополнительных обменов, так как для получения координат файла на носители не требуется совершать лишних действий.

Но что будет, когда требуется записать в такой файл дополнительную информацию, а свободного пространства за файлом нет? В этом случае файловая система может поступить двояко: в первом случае — она скажет, что места нет, и пользователь должен что-то сделать сам (например, перенести файл на большее место, где хватит места для того, чтобы дописать информацию в файл — но такой перенос дело очень дорогостоящее), во втором случае — будет просто отказано в обмене, то есть при открытии файла мы должны были сразу зарезервировать необходимое место.

Итак, организация простая, при обменах — эффективная, но в случае нехватки пространства для файлов начинается неэффективность и при долговременной работе случается фрагментация (когда имеются свободные фрагменты, которые объемом позволяют разместить файл, а расположением — нет). Это та же самая проблема, которая

имела место в ОЗУ. Решить данную проблему можно путем компрессии — долгого и опасного для данных процесса, который передвигает блоки по ВЗУ с целью разместить их оптимальным образом. Можно сделать эту компрессию так, чтобы в конец каждого файла был добавлен кусок свободного места (например, по 10% от текущей длины).

Такая файловая система может быть пригодна только для однопользовательской системы. Ибо при большом потоке информации, в случае многопользовательской системы, происходит быстрая деградация файлового пространства и требуется долговременный процесс компрессии. Она одноуровневая — то есть не может быть двух файлов с одинаковыми именами. Но она очень проста и требует минимум дополнительных расходов.

Файловая система с блочной организацией файлов.

0	
1	
2	
N-1	

Пространство ВЗУ разделено на блоки (блоки удобных для обмена размеров или кратных им). В файловой системе такого типа происходит аналогично распределению процесса в памяти со страничной организацией. То есть в общем случае с каждым именем файла связан набор номеров блоков устройства, в которых размещены данные этого файла,

Name {Блок1, Блок2, ..., БлокМ}

причем эти блоки имеют произвольный порядок на ВЗУ:

ВЗУ:

...
БлокМ
Блок4
пустой блок или блок другого файла
Блок1
...

При такой организации мы имеем потери кратные блоку (если хотя бы один байт блока занят, то он считается непустым и не может быть отдан другому файлу), но при такой схеме нет проблемы фрагментации. Как следствие, нет надобности в компрессии. Следовательно, такая файловая система может использоваться при многопользовательской работе.

В последнем случае с каждым файлом у нас связано несколько атрибутов:

Имя файла	Имя пользователя

И, соответственно, доступ к файлу осуществляется заданием двух параметров. Таким образом, уже необязательна уникальность имен файлов во всей файловой системе,

нужна лишь уникальность имен файлов у одного пользователя. У разных пользователей могут быть файлы с одинаковыми именами.

Организация таких файлов может быть через каталог. Структура каталога может быть следующая:

i->	блок занят или свободен, если занят, то кем.

Каталог содержит строки, каждая iая строка соответствует iому блоку файловой системы, в этой строке содержится информация о занятости блока. Если он занят, то в этой строке указывается имя файла (или ссылка на него) и имя пользователя, может быть и какая-то дополнительная информация.

При открытии файла система может, например, пробегать по ВЗУ и строить таблицу соответствия между логическими блоками файла и физическими (блоками носителя).

При данной организации файловой системы мы имеем одноуровневую структуру пользователей и, как следствие, все файлы связаны в группы только по принадлежности какому-либо пользователю.

Иерархическая файловая система.

Все файлы файловой структуры строятся в дерево. Корнем дерева является так называемый корень файловой системы. Если узел дерева является листом, то это файл, который может содержать либо данные, либо являться каталогом. Узлы, отличные от листьев являются каталогами. Соответственно, именование в такой системе может происходить разными способами. Первый — именование фала относительно ближайшего каталога. Если мы посмотрим файлы, которые являются ближайшими для каталога F0 — это файл F1 (он тоже является каталогом) и файл F2. То есть если мы каким-то образом подразумеваем (системным образом), что работаем в каталоге F0, то можем обращаться к файлам в данном каталоге только по их именам (F1 и F2). Соответственно, на одном уровне имена должны быть уникальны (в пределах одного каталога). Так как мы имеем структуру дерева, то можно говорить о полном имени файла, которое составляет путь от корня дерева, до файла. Например, путь к файлу F3 будет выглядеть, как “/F0/F1/F3”. В одно и то же время мы можем работать как с полным, так и с коротким именем файла. А так как по свойству дерева путь к каждому листу однозначен, то мы сразу решаем проблему унификации имен.

Первой такая организация появилась в ОС Multics, которая разрабатывалась в университете Беркли в конце 60х годов. Это было давно, но такое хорошее и красивое решение с тех пор стало появляться во многих ОС.

Соответственно с иерархией каждому файлу можно привязывать некоторые атрибуты, связанные с правами доступа, этими атрибутами могут обладать как файлы, так и каталоги. То есть структурная организация такой файловой системы хороша для многопользовательской системы. Ибо с одной стороны нет проблемы именования, а с другой стороны такая система может сильно и хорошо наращиваться.

Защита данных в ОС

Идентификация — возможность ОС распознать определенного пользователя и выполнять в зависимости от определения нужные действия по защите данных и т.п.

Например, MS DOS — однопользовательская ОС. Существуют системы, которые позволяют регистрировать пользователей, но эти пользователи никак между собой не связаны (примером могут являться некоторые ОС фирмы IBM для мейнфреймов), а значит их нельзя организовать в группы. Но было бы удобно выделить в отдельную группу — лабораторию, кафедру, учебную группу студентов и т.п.

В иерархической организации пользователей есть понятие группы. А в группе есть реальные пользователи.

При регистрации конкретного пользователя его следует отнести к какой-либо группе.

Раз пользователи разделены на группе, то по аналогии с разделением между конкретными пользователями, можно разделять ресурсы с группой (то есть пользователь может сделать свои файлы доступными для всех членов какой-то группы.

И такое деление на группы может быть также многоуровневым с соответствующим распределением прав и возможностей.

Маленькое замечание — сейчас появляются ОС, в которых права доступа могут быть не только иерархическими, но и более сложные — например, нарушая иерархию (какой-то файл может быть доступным конкретному пользователю из группы другой ветви дерева).

Вот, наверное, и все, что следовало бы сказать о свойствах и функциях ОС. Естественно, мы рассмотрели далеко не все функции ОС. Что-то было специально упущено, так как мы рассматриваем ОС в упрощенной модели. Ибо наша цель — не изучение конкретной ОС, а научиться классифицировать ОС, с каких точек зрения следует на нее смотреть и сравнивать различные типы ОС.

Лекция 7

ОС Unix

Сегодня мы с вами переходим к началу рассмотрения ОС Unix, поскольку многие решения, которые принимаются в ОС мы будем рассматривать на примере этой ОС.

В середине 60х годов в Bell лаборатории фирмы AT&T проводились исследования и разработка одной из первых ОС в современном ее понимании — ОС Multics. Это ОС разделения времени, многопользовательская, а также в этой системе были предложены фактически основные решения по организации файловых систем. В частности, была предложена иерархическая древообразная файловая система. Это, ориентировочно, 1965 год. От этой разработки через некоторое время получила начало ОС Unix. Одна из предысторий говорит, что на фирме был ненужный компьютер PDP8 с очень малоразвитым программным обеспечением. А требовалась машина, которая бы позволяла организовывать удобную работу пользователя, в частности, удобный ввод информации. И известная группа людей — Томпсон и Ритчи занялись разработкой на этой машине новой ОС. Другой вариант был таков, что они занимались реализацией новой игры, а те средства, которые имелись были недоступны или неудобны, и они решили поиграться с этой машиной. Результатом стало появление ОС Unix. Особенностью этой системой являлось то, что она являлась первой системной программой написанной на языке, отличном от языка ассемблера. Дляцели написания этого системного программного обеспечения, в частности, ОС Unix, также параллельно проводились работы, которые начинались от языка BCPL, из него был образован язык B, который оперировал с машинными словами, далее абстракция машинных слов — BN и, наконец, язык “C”. И после 1973 года ОС Unix была переписана окончательно на язык “C”. В результате появилась ОС, 90% кода которой было написано на языке высокого уровня, языке, не зависящем от архитектуры машины и системы команд, а 10% было написанона ассемблере, в эти 10% входят наиболее критичные к реализации по времени части ядра ОС.

Многих программистов в то время это немного шокировало, мало кто верил, что такая ОС способна жить, поскольку всегда язык высокого уровня ассоциировался с большой неэффективностью. Но язык “C” тем не менее был сконструирован таким образом, что позволял писать эффективные программы и транслировать их в также достаточно эффективный машинный код.

Из таких конструктивных свойств следует отметить то, что “C” сильно построен на работе с указателями. Когда мы пишем программу на ассемблере, то очень часто для достижения требуемого результата нам нужно манипулировать с адресами. Возможность оперировать указателями — первое свойство “C”, которое позволяет эффективно транслировать программу на этом языке в машинный код.

Если мы посмотрим на нормальную программу на ассемблере, то заметим следующее — при программировании каких-то блоков мы часто используем побочный эффект (например, во время вычисления выражения мы можем получать и куда-то откладывать промежуточные результаты), также можно поступать и в языке “C”. Таким образом, понятие выражения в “C” было гораздо шире, чем в других языках того времени. И в выражениях, кроме новых операций, таких как работа с указателем, смещения сдвиги и т.п., появилась принципиально новая операция — операция присваивания. Почему она новая? Потому что во многих языках до “C”, а также и после него не было операции присваивания — был оператор присваивания. Разница в одном — если мы имеем оператор присваивания, во-первых, требуется, чтобы в правой части такой операции уже не было (мы не можем использовать побочный эффект), и второе — левая часть оператора присваивания — это некоторая ссылка на единичную область памяти. Внесение оператора присваивания внутрь выражения позволило решать проблему побочных эффектов (значения подвыражений, которые могут быть использованы во вне — а они в свою очередь сокращают число обменов с ОЗУ), а это средство эффективности.

Эти и, наверное, только эти свойства языка определили его живучесть, пригодность для программирования системных компонентов и возможность оптимальной трансляции кода различных машин. С профессиональной точки зрения, язык “C” — ужасный язык. Основным требованием, которое предъявляется сегодня к языкам программирования является безопасность программирования. То есть средства языка должны минимизировать количество возможных ошибок.

И свойствам таких языков относится следующее:

1)Жесткий контроль типов. То есть если мы попробуем умножить целочисленную переменную на плавающую, то язык выдаст ошибку. Все преобразования типов по умолчанию недопустимы.

2)Обеспечение контроля за доступом в память программы. Это означает, что если у нас в памяти число было записано, как целое, то и считать его оттуда мы можем только как целое, а не как плавающее или символ. В “C” же и других языках бесконтрольный доступ к памяти предоставляет указатель, более того, через указатель мы с одной стороны теряем любую информацию о типе, а с другой стороны мы можем обманывать функциипо части фактических и формальных параметров.

3)Контроль за взаимодействием модулей. Суть этого свойства в том, что много ошибок появляется в том случае, что если функция продекларировала один набор параметров, а обращение к ней идет с другим набором, причем различие может быть как в количестве, так и в типах. Язык “C”, несмотря даже на версию ANSI C, которая пыталась отчасти решить эту проблему — всегда остается возможность обмануть функцию и передать ей параметр другого типа, вместо шести параметров можно передать один параметр.

Вот по этим трем позициям язык “C” является нехорошим языком. Но тем не менее это “менталитет” программистов, который заключается в том, что почему-то наиболее живучими языками являются концептуально плохие языки, к таким языкам помимо “C” можно добавить еще Фортран.

Итак, 1973 год. Появление ОС Unix, причем она уже была написана на языке “C”. Какими основными свойствами уже тогда обладала эта ОС. Первое свойство — концепция файлов, основным объектом, которым оперирует ОС — это файл. Файл — это набор данных, файл с точки зрения Unix — это внешнее устройство, файл — это каталог, который содержит информацию о принадлежащих ему файлах и т.д. На сегодняшний день стратегия файлов распространена в Unix’е практически на все. Второе свойство, которое является продолжением или следствием первого, это то, что ОС построена очень интересно. В отличии от предыдущих ОС, где каждая команда была зашита внутрь, и эту команду нельзя было модифицировать, убрать из системы, создать новую команду — вUnix’е проблемы команд пользователя решены очень элегантно за счет двух моментов. Первый — Unix декларирует стандартный интерфейс передачи параметров извне внутрь процесса. Второй — все команды реализованы в виде файлов, это означает, что можно свободно добавлять новые команды в систему, которые будут доступны либо мне, либо группе пользователей, либо всем, а можно удалять команды.

Давайте начнем рассмотрение конкретных свойств ОС Unix. Первое, что мы будем рассматривать, это файловая система, организация работы с файлами.

Файловая система Unix

Файловая система Unix, это иерархическая, многопользовательская файловая система. Ее можно представить в виде дерева:

В корне дерева находится “корневой каталог”, узлами, отличными от листьев дерева являются каталоги. Листьями могут являться: файлы (в традиционном понимании — именованные наборы данных), пустые каталоги (каталоги, с которыми не ассоциировано ни одного файла). В системе определено понятие имени файла — это имя, которое ассоциировано с набором данных в рамках каталога, которому принадлежит этот файл. Например, каталогу D1 принадлежат файлы: N1, N2, N3; каталогу D0 принадлежат: N4, N5 и D1, последний тоже является файлом, но специальный. Итак, имя — это имя, которое ассоциировано снабором данных в контексте принадлежности каталогу. Кроме того, есть понятие полного имени. Полное имя — это уникальный путь от корня файловой системы до конкретного файла. Первый символ имени — это корневой каталог “/”, а далее через наклонную черту перечислены все каталоги, пока не дойдет до нужного файла. Например, файл N3 имеет полное имя “/D0/D1/N3”. За счет того, что такой путь для каждого файла в любом каталоге уникален, то мы можем именовать одинаковыми именами файлы в различных каталогах. Например, имя N4 присутствует в каталогах D0 и D4, но это разные файлы, так как полные пути к ним различны (/D4/N4, /D0/N4).

Замечание. На самом деле файловая система Unix не является древообразной. Все то, что говорилось выше — правильно, но в системе имеется возможность нарушения красивой и удобной иерархии в виде дерева, так как имеется возможность ассоциировать несколько имен с одним и тем же содержимым файла. И могут возникать такие ситуации, когда, например, “/D4/N3” и “/D0/D1/N1” являются, по сути дела, одним файлом с двумя именами.

Еще одно замечание. В ОС Unix используется трехуровневая иерархия пользователей:

Первый уровень — все пользователи. Они подразделены на группы и, соответственно, группы состоят из реальных пользователей. В связи с этой трехуровневой организацией пользователей каждый файл обладает тремя атрибутами:

1)Владелец файла. Этот атрибут связан с одним конкретным пользователем, который автоматически назначается системой владельцем файла. Владельцем можно стать по умолчанию, создав файл, а также есть команда, которая позволяет менять владельца файла.

2)Защита доступа к файлу. Доступ к каждому файлу (от файла ядра системы до обыкновенного текстового файла) лимитируется по трем категориям:

права владельца (что может делать владелец с этим файлом, в общем случае — не обязательно все, что угодно);

права группы, которой принадлежит владелец файла. Владелец сюда не включается (например, файл может быть закрыт на чтение для владельца, а все остальные члены группы могут свободно читать из этого файла;

все остальные пользователи системы;

По этим трем категориям регламентируются три действия: чтение из файла, запись в файл и исполнение файла (в мнемонике системы R,W,X, соответственно). В каждом файле по этим трем категориям определено — какой пользователь может читать, какой писать, а кто может запускать его в качестве процесса.

Это некоторые предварительные данные по файловой системе. Теперь давайте рассмотрим структуру файловой системы на диске.

Сначала определим некоторые понятия:

Для любой вычислительной системы определено понятие системного внешнего запоминающего устройства (ВЗУ). Это устройство, к которому осуществляет доступ аппаратный загрузчик системы с целью запуска ОС. Суть заключается в следующем — практически любая вычислительная система имеет диапазон адресного пространства оперативной памяти, размещенной в ПЗУ. В ПЗУ размещается небольшая программа (хотя понятие размера относительно, но она действительно небольшая), которая при включении вычислительной машины обращается к фиксированному блоку ВЗУ, считывает его в память и передает управление на фиксированный адрес, относящийся к считанному блоку данных.

Считается, что считанный блок данных является программным загрузчиком и программный загрузчик раскручивает запуск ОС. Следует отметить, что если аппаратный загрузчик в подавляющем большинстве машин системно независим (то есть он не знает, какая ОС будет загружена), то программный загрузчик — это уже компонент ОС, ему известно, что будет загружаться конкретная ОС, он знает, где размещаются нужные для загрузки данные.

В любой системе принято разбиение пространства ВЗУ на некоторые области данных, которые называются блоками. Размер блока (логического блока в ОС) является фиксированным атрибутом. В ОС Unix в различных ее вариациях размер блока был параметром меняющимся в зависимости от варианта ОС. Для простоты и единообразия мы будем считать, что логический блок ВЗУ равен 512 байт.

Итак, рассмотрим структуру файловой системы. Представим адресное пространство системного ВЗУ в виде последовательности блоков.

Блок начальной загрузки	Суперблок	Индексные дескрипторы	Блоки файлов	Область сохранения
0				N-M+1

Будем считать, что этих блоков N+M-1.

Первый блок — это блок начальной загрузки. Размещение этого блока в нулевом блоке системного устройства определяется аппаратурой, так как аппаратной загрузчик всегда обращается к конкретному блоку системного устройства (к нулевому блоку). Это последний компонент файловой системы, который зависит от аппаратуры.

Следующий блок — суперблок файловой системы. Он содержит оперативную информацию о состоянии файловой системы, а также данные о параметрах настройки файловой системы. В частности суперблок имеет информацию о

–количестве индексных дескрипторов (ИД) в файловой системе;

–размере файловой системы;

–свободных блоках файлов;

–свободных ИД;

–еще ряд данных, которые мы не будем перечислять в силу уникальности их назначения.

Третий блок — область индексных дескрипторов. ИД — это специальная структура данных файловой системы, которая взаимооднозначно соответствует файлу. С каждым содержимым файла связан один и только один ИД. ИД организуют не один блок, а пространство блоков, размеры которого определяются параметром генерации файловой системы (определяется по количеству ИД,указанном в суперблоке). Соответственно, каждый индексный регистр содержит следующую информацию:

–поле, определяющее тип файла (каталоги и все остальные файлы);

–код привилегии/защиты;

–количество ссылок к данному ИД из всевозможных каталогов файловой системы;

–длина файла;

–даты и времена (время последней записи, дата создания и т.д.);

–поле адресации блоков файла.

Далее идут блоки файлов. Это пространство ВЗУ, в котором размещается вся информация, находящаяся в файлах и о файлах, которая не поместилась в уже перечисленных блоках.

Последняя область данных (она в разных системах размещается по-разному), но для простоты изложения мы будем считать, что эта область находится сразу за блоками файлов — это область сохранения.

Это концептуальная схема структуры файловой системы. Теперь давайте вернемся и рассмотрим некоторые ее части более детально.

Прежде всего интерес вызывают области свободных блоков файлов и свободных ИД. В Unix видно влияние двух факторов: первый — это то, что файловая система разрабатывалась тогда, когда ВЗУ объемом 5-10Мб считалось очень большим и в реализации алгоритмов по работе с системой видны старания автором по оптимизации этого процесса; и второй — это свойства файловой системы по оптимизации доступа, критерием которого является количество обменов, которые файловая система производит для своих нужд, не связанных с чтением или записью информации файлов.

Суперблок содержит список свободных блоков файлов, он состоит из 50 элементов. Суть работы с этим списком заключается в следующем — в буфере, состоящем из 50 элементов (при условии того, что блок — 512 байт, 1 блок — 16 битное слово), в них записаны номера свободных блоков пространства блоков файлов с 2 до 49. В 0 элементе содержится указатель на продолжение массива, а в последнем элементе содержится указатель на свободный элемент в массиве.

Если какому-то процессу для расширения файла требуется свободный блок, то система по указателю N/B (номер блока) выбирает элемент массива, и этот блок предоставляется файлу. Если происходит сокращение файла, то высвободившиеся номера добавляются в массив свободных блоков и корректируется указатель N/B.

Так как размер массива — 50 элементов, то возможны две критические ситуации:

1)Когда мы освобождаем блоки файлов, а они не могут поместиться в этом массиве. В этом случае из файловой системы выбирается один свободный блок и заполненный полностью массив свободных блоков копируется в этот блок, после этого значение указателя N/B обнуляется, а в нулевой элемент массива, который находится в суперблоке, записывается номер блока, который мы выбрали для копирования содержимого массива. Таким образом, если мы постоянно освобождаем блоки, то образуется список, в котором будут размещены все свободные блоки файловой системы.

2)Когда мы выбрали все свободные блоки и содержимое элементов массива свободных блоков исчерпалось. Если нулевой элемент массива равен нулю, то это означает, что исчерпано все пространство файловой системы. Если этот элемент нулю не

равен, то это означает, что существует продолжение массива. Это продолжение считывается в копию суперблока в оперативной памяти.

Для получения свободного блока и его освобождения в большинстве случаев не требуется дополнительного обмена. Дополнительный обмен требуется тогда, когда исчерпается содержимое 49 блоков. У нас получается хорошая буферизация, которая сокращает накладные расходы ОС.

Список свободных ИД. Это буфер, состоящий из 100 элементов. В нем находится информация о 100 номерах ИД, которые свободны в данный момент. Соответственно, когда нужен новый ИД, то его номер берется из списка свободных ИД, если номер освобождается, то заносится в этот массив. Если же массив переполнен, а освобождается 101 элемент, то это никуда не записывается. Если список ИД переполняется, то система “пробегает” по списку и формирует содержимое этого буфера заново.

В ситуации, когда нужно создать файл и нужен новый ИД, а в массиве нет ни одного элемента — запускается процесс поиска нового ИД, и он ничего не находит. Тогда возможны две ситуации:

1)Больше нет свободных блоков для файлов;

2)Нет больше новых ИД.

Вот информация о суперблоке. Какие можно сделать выводы и замечания?

–суперблок всегда находится в ОЗУ;

–все операции по освобождению блоков, занятию блоков файлов, по занятию и освобождению ИД происходят в ОЗУ (минимизация обменов с диском). Если же содержимое суперблока не записать на диск и выключить питание, то возникнут проблемы (несоответствие реального состояния файловой системы и содержимого суперблока). Но это уже требование к надежности аппаратуры системы.

Лекция 8

Индексные Дескрипторы

Рассмотрим подробнее Индексные Дескрипторы. *ИД — это объект Unix, который ставится во взаимнооднозначное соответствие с содержимым файла.* То есть для каждого ИД существует только одно содержимое и наоборот, за исключением лишь той ситуации, когда файл ассоциирован с каким-либо внешним устройством. Напомним содержимое ИД:

–поле, определяющее тип файла (каталоги и все остальные файлы);

–код привилегии/защиты;

–количество ссылок к данному ИД из всевозможных каталогов файловой системы;

–(нулевое значение означает свободу ИД)

–длина файла в байтах;

–даты и времена (время последней записи, дата создания и т.д.);

–поле адресации блоков файла.

Как видно — в ИД нет имени файла. Давайте посмотрим, как организована адресация блоков, в которых размещается файл.

В поле адресации находятся номера первых десяти блоков файла, то есть если файл небольшой, то вся информация о размещении данных файла находится непосредственно в ИД. Если файл превышает десять блоков, то начинает работать некая списочная структура, а именно, 11й элемент поля адресации содержит номер блока из пространства блоков файлов, в которых размещены 128 ссылок на блоки данного файла. В том случае, если файл еще больше — то используется 12й элемент поля адресации. Сутьего в следующем — он содержит номер блока, в котором содержится 128 записей о номерах блоках, где каждый блок содержит 128 номеров блоков файловой системы. А если файл еще больше, то используется 13 элемент — где глубина вложенности списка увеличена еще на единицу.

Таким образом мы можем получить файл размером $(10+128+128^2+128^3)*512$.

Если мы зададим вопрос — зачем все это надо (таблицы свободных блоков, ИД и т.д.), то вспомним, что мы рассматриваем взаимосвязь между аппаратными и программными средствами вычислительной системы, а в данном случае подобное устройство файловой системы позволяет сильно сократить количество реальных обменов с ВЗУ, причем эшелонированная буферизация в ОС Unix делает число этих обменов еще меньше.

Рассмотрим следующую область — область сохранения. На схеме она изображена сразу за блоками файлов. На самом же деле она может размещаться по-разному: перед блоками файлов, в каком-нибудь файле или еще где-нибудь, например, на другом ЗУ. Все это зависит от конкретной реализации системы.

В область сохранения происходит откачка процессов, она же используется для оптимизации запуска наиболее часто запускающихся процессов (использование так называемого Т-бита файла).

Мы с вами рассмотрели структуру файловой системы и ее организацию на системном устройстве. Эта структура и алгоритмы работы с ней достаточно простые, это сделано для того, чтобы накладные расходы, связанные с функционированием системы, не выходили за пределы разумного.

Элементы файловой системы:

Каталоги

Мы говорили, что вся информация в Unix размещается в файлах. Нету каких-то специальных таблиц, которые размещены вне файловой системы и используются системой, за исключением тех таблиц, которые создает ОС во время работы в пространстве оперативной памяти.

Каталог с точки зрения ОС — это файл, обычный файл, в котором размещены данные о всех файлах, которые принадлежат каталогу.

Мы говорим, что в каталоге “A” содержатся файлы: “B”, “C” и “D” — из которых “B” и “C” могут быть как файлами, так и каталогами, а “D” — заведомо каталог.

Каталог имеет следующую структуру. Он состоит из элементов, объединяющих в себе два поля — номер ИД и имя файла:

Каталог = { {ИД, Имя}, {ИД,Имя}, ..., {ИД, Имя}}

Что есть номер ИД? — это порядковый номер элемента в списке индексных дескрипторов. Так, первый элемент этого списка — ИД#1 принадлежит корневому каталогу “.”.

В общем случае, в каталоге могут неоднократно встречаться записи, ссылающиеся на один и тот же ИД, но в каталоге не могут встречаться записи с одинаковыми именами. То есть с содержимым файла может быть связано произвольное количество имен. При создании каталога в нем всегда создаются две записи:

{ИД_самого_каталога, “.”} и {ИД_родительского_каталога, “..”}

Так на картинке файл “A” имеет ИД#7, “D” — ИД#5, “F” — ИД#10, “G” — ИД#101. В этом случае файл-каталог D будет иметь следующее содержимое:

{{ 5, “.” },
{ 7, “..”},
{10, “F”},
{101,”G”}}

(Для корневого каталога родитель ссылается на него же самого.)

Чем отличается файл-каталог от обычного файла? Он отличается полем типа в индексном дескрипторе.

Давайте посмотрим, как схематично могут использоваться полные имена и ссылки на каталоги. В системе в каждый момент времени определен для пользователя текущий каталог. То есть каталог, полное имя которого подставляется ко всем файлам, имя которых не начинается с символа “/”. Если текущий каталог “D”, то можно говорить

просто о файле “F” или файле “G”, если же текущий каталог “D”, а требуется добраться до файла “B”, то оперировать просто с именем“B” нельзя, так как он не принадлежит каталогу “D”, файл “B” можно достать, указав его полное имя от корня, либо использовать специальный файл “..”, в этом случае файл “B” будет иметь имя: “../B”. Если при открытии мы ссылаемся на “..”

Для того, чтобы в этом случае открыть файл “B”, придется выполнить ряд косвенных операций — взять ИД родитель, и по нему выбирается содержимое файла-каталога “A”, в “A” мы выбираем строку с именем “B” и определяем его ИД. Эта процедура достаточно трудоемка, но так как открытие и закрытие файлов происходит достаточно редко, то “криминала” в этом никакого нету.

За счет такой организации каталогов у нас содержимое файла разорвано с его именем. Имя может быть определено неоднозначно.

Так как с одним файлом может ассоциировано несколько имен, то можно говорить о том, что этот файл может быть одновременно открыт несколькими процессами (вообще говоря, имея одно имя мы тоже можем открыть этот файл из нескольких процессов, суть проблемы от этого уточнения не изменяется). Как организуется синхронизация в этом случае? Как мы увидим позже, здесь все решается корректно.

Файлы устройств

Эта разновидность файлов характеризуется типом и их интерпретация происходит следующим образом. В принципе, содержимого у файлов устройств нету, то есть это лишь ИД и имя, которое с ним ассоциировано. В ИД указывается информация о том, какой тип устройства ассоциирован с этим файлом, соответственно, система Unix все устройства подразделяет на два типа: байт- и блок-ориентированные. Байт-ориентированные устройства — это те устройства, обмен с которыми происходит по байтам (например, клавиатура), блок-ориентированные — это такие устройства, обмен с которыми происходит блоками. В ИД имеется поле, указывающее эту характеристику, там же имеется поле, определяющее номер драйвера, связанного с этим устройством. В системе каждый драйвер связан с конкретным одним устройством, но у устройства может быть несколько драйверов. Это поле, определяющее номер драйвера, на самом деле есть номер в таблице драйверов соответствующего класса устройств (имеются две таблицы — для блок- и байт- устройств). Также в ИД существует некоторый цифровой параметр, который может быть передан драйверу в качестве параметра, уточняющего информацию о работе.

Это то, что можно сказать о специальных файлах, связанных с внешними устройствами.

Обмен данными с файлами

Следующее из системной организации файловой системы — это организация обменом данными с файлом. Определим понятия, связанные с низкоуровневым вводом/выводом. В Unix определены специальные функции, которые называются системными вызовами. Эти вызовы осуществляют непосредственное обращение к ОС, они выполняют некоторые системные функции. По употреблению они практически не отличаются от использования библиотечных функций, тогда как по реализации и действии их отличие достаточно существенное. Библиотечная функция будет загружена в тело процесса, а системный вызов сразу передает управление ОС, и последняя выполняет заказанное действие. В Unix для обеспечения низкоуровневого (путем системных вызовов) ввода-вывода имеется набор этих функций:

open(...) — для работы с содержимым файла процесс должен зарегистрировать в системе этот факт, параметрами этой функции являются строка, содержащая имя файла и атрибуты на режим работы с файлом (только чтение, чтение-запись и т.п.), а возвращает эта функция некоторое число, которое называется файловым дескриптором (ФД). В теле процесса пользователя, а также данных, ассоциированных с этим процессом, размещается

некая служебная информация. В частности, размещается таблица файловых дескрипторов. Она, как и все таблицы в Unix — позиционна, то есть номер строки в таблице соответствует ФД с этим номером. С ФД ассоциировано имя файла и прочие атрибуты. Нумерация ФД — прерогатива процесса, то есть ФД уникальны в пределах одного процесса.

Количество одновременно открытых файлов (точнее, максимальное количество ФД, ассоциированных с файлами) для процесса регламентируется системой.

Итак, функция open(...) — открытие существующего файла.

creat(...) — это функция открытия нового файла, ее параметрами служат: имя файла и некоторые параметры открытия, также как и у open.

read(...)/write(...) — их параметрами являются номер ФД и некоторые параметры доступа. Эти функции служат для чтение/записи из или в файл.

close(...) — завершение работы с файлом. После выполнения этой функции ФД этого файла освобождается.

Все это системные вызовы. Также в Unix можно осуществлять ввод-вывод через библиотечные функции (например, fopen, fread, fwrite, fclose, ...).

Рассмотрим организацию обмена с системной точки зрения в Unix.

При организации обмена система подразделяет все данные на две категории — первая, это данные, ассоциированные с процессом пользователя, и данные, ассоциированные с ОС.

Первая таблица данных, связанных с ОС — это таблица индексных дескрипторов открытых файлов (ТИДОФ), эта таблица содержит записи, каждая из которых содержит копию ИД для каждого открытого в системе файла. Через копию ИД мы осуществляем доступ к блокам файла. Каждая из этих записей содержитполе, характеризующее количество открытых файлов в системе, использующих данные ИД. То есть, если мы открываем один и тот же файлов от имени двух процессов, то запись в ТИДОФ создается одна, но каждое открытие этого ИД увеличивает счетчик на единицу.

Следующее. Таблица файлов — эта таблица содержит информацию об имени открытого файла и имеет ссылку на ИД данного файла в ТИДОФ.

Подробнее эта схема будет рассмотрена на следующей лекции.

Лекция 9

Мы начали рассмотрение принципов организации работы ОС Unix с файловой системы. Точнее организации обработки ввода-вывода.

Теперь давайте посмотрим, как организуется обработка низкоуровневого обмена с точки зрения ОС. Понятно, что мы будем рассматривать чуть более общую модель и не уделять времени не самым значащим деталям.

Для поддержания ввода-вывода в системе все данные в системе подразделяются на два типа: общесистемные данные (ТИДОФ, например). Размер ТИДОФ фиксирован — это еще один параметр настройки системы. Каждая запись этой таблицы содержит некоторую информацию, из которой нас будет интересовать следующее:

1)Копия ИД открытого файла. То есть для любого открытого файла ИД, который характеризует содержимое этого файла, копируется и размещается в ТИДОФ. После этого все манипуляции с файлами происходят через копию ИД. Не с ИД, который на диске, а с его копией. Таким образом доступ к информации осуществляется оперативно.

2)Счетчик открытых в данный момент файлов, связанных в данный момент с данным ИД. Это означает, что на любое количество открытий файла, связанного с данным ИД, система работает с одной копией этого ИД.

Рассмотрим следующую таблицу — Таблицу Файлов (ТФ) — она также состоит из определенного числа записей. Каждая запись в ТФ соответствует открытому в системе файлу. При этом в подавляющем большинстве случаев это есть взаимооднозначное

соответствие (исключения мы рассмотрим позже). Каждая запись ТФ содержит указатели чтения-записи по файлу. Это означает, что если мы открыли один и тот же файл в двух или в одном процессе, то с каждым таким открытием связано по одному указателю, и они друг с другом независимы, то есть если мы изменяем один указатель, то другой остается в прежнем состоянии. И так почти всегда, за исключением некоторых случаев. Кроме того, в строке ТФ содержится некоторая запись — число, которое называется индексом наследственности. Это данные уровня ОС, то есть данные, которые описывают состояние системы в целом. С каждым процессом связана так называемая таблица открытых файлов (ТОФ).

Номер записи в данной таблице — номер файлового дескриптора, каждая строка имеет ссылку на соответствующую строку ТФ. Это означает, что информация об указателях как бы разорвана. То есть файловый дескриптор, являющийся атрибутом процесса, с другой стороны является атрибутом ОС.

Для того, чтобы мы имели возможность рассмотреть все грани данного вопроса — забежим немного вперед и рассмотрим некоторые вещи, связанные с формированием процесса.

ОС Unix имеет функцию, которая называется fork(...) — это системный вызов, при обращении к которому происходит некоторое “бесполезное” действие — создается процесс двойник — полная (с некоторыми замечаниями) копия процесса, в котором встретилась эта функция. Для чего это нужно? Нужно, для чего — увидим позже.

При формировании процесса двойника есть две особенности:

–процесс-сын имеет все те же файлы открытыми, что были открыты в процессе отца;

–система позволяет идентифицировать — где сын, где отец;

Предположим, у нас есть процесс #1 и с ним ассоциирована ТОФ1, в этом процессе открыт файл с именем “name”, ему соответствует ФД i. Это означает, что строка i ТОФ будет иметь ссылку на строку из ТФ, где содержится системная информация о файле, в том числе указатели чтения/записи. Записи в ТФ имеют ссылку на строку ТИДОФ, где находится копия ИД файла с именем “name”. Предположим, что в этом же процессе мы открыли еще раз файл с именем “name”. Система поставила ему в соответствие ФД j, это означает, что в записи j ТОФ будет ссылка на запись в ТФ, соответствующую второму открытию файла “name”.Индексы наследственности в обоих случаях будут равны единице.

Соответственно, когда мы изменяем указатель файла i (читаем или пишем), то файловый указатель j изменяться не будет. Обе записи в ТФ ссылаются на один и тот же ИД файла.

Теперь предположим, что процесс #1 выполнил обращение к функции fork(...). Образовалась копия этого процесса, причем обе копии начинают работать на выходе из процесса. И, соответственно, со вторым процессом будет ассоциирована ТОФ2.

Файлы “name” с дескрипторами i,j будут также открыты во втором процессе. Но, когда процесс получает открытые файлы в наследство от родителя, то ссылки из соответствующих строк таблицы ТОФ будут происходить не на новые строки ТФ, а на те же самые, на которые ссылались ФД родителя. Это означает, что у обоих процессов будут одинаковые указатели файлов — при перемещении указателя для ФД i в процессе-отце будет также изменен файловый указатель для ФД i в сыне и наоборот. Вообще говоря, два процесса по ФД i (или любому переданному в наследство открытому файлу) будут иметь общий указатель файла.

Вот это случай, когда нет взаимооднозначного соответствия между строками ТФ и ТОФ. И во время создания процесса сына счетчик наследственности увеличивается на единицу.

Что означает такая организация доступа к данным файла? Это означает, что этот доступ осуществляется централизованно, то есть в конечном итоге все заказы на обмен идут через одну единственную запись, сколько бы раз файл ни был открыт в системе. Отсюда мы получаем отсутствие путаницы при доступе к файлу.

При любом формировании нового процесса, система автоматически связывает 0, 1 и 2 ФД с предопределенными файлами:

0 — системный файл ввода (обычно — файл устройства клавиатура);

1 — системный файл вывода (обычно — файл устройства монитор);

2 — файл вывода диагностических сообщений (обычно — также файл устройства монитор).

Рассмотрим типовые действия при обращении к тем или иными системным вызовам.

При обращении к функции fork(...) система создает копию процесса и дублирует ТОФ родителя в ТОФ сыновнего процесса, а также увеличивает на единицу индексы наследственности в соответствующих строках ТФ, и увеличивает счетчик связей в ТИДОФ.

При выполнении системного вызова open(...):

1)По полному имени определяется каталог, в котором размещается файл;

2)Определяется номер ИД файла;

3)По номеру ИД осуществляется поиск в ТИДОФ, если запись с данным номером обнаружена, то номер соответствующей строки ТИДОФ фиксируется и переходим к шагу 5;

4)Если такой записи не обнаружено, происходит формирование новой строки в ТИДОФ, соответствующей новому ИД и фиксируется ее номер;

5)Корректируется счетчик ссылок (количество открытых файлов, использующих данный ИД) в ТИДОФ. Номер строки ТИДОФ записывается в строку ТФ, а ее номер возвращается в ТОФ;

При операциях ввода-вывода мы идем по ссылкам и добираемся до нужного блока данных.

Взаимодействие с устройствами.

Мы говорили, что все устройства, которые обслуживает ОС Unix, могут быть подразделены на два типа: байт- и блок-ориентированные. С первыми устройствами все обмены осуществляются порциями по одному байту, с остальными — некоторыми порциями байт. С точки зрения ОС одно и то же устройство может рассматриваться как байт- и как блок-ориентированное. Примером такого устройства может быть оперативная память.

Различие составляет наличие или отсутствие соответствующих драйверов, ибо существует две таблицы — байт- и блок-ориентированных драйверов. На эти таблицы имеются ссылки в ИД специальных файлов.

Основной особенностью организации работы с блок-ориентированными устройствами является возможность буферизации обмена. В оперативной памяти организован пул буферов.

Каждый из буферов пула состоит из буфера размером один блок. Каждый из этих блоков может быть ассоциирован с драйвером одного из блок-ориентированных устройств. Посмотрим, какие происходят действия при выполнения заказа на чтение блока.

Пусть поступил заказ на чтение Nного блока из устройства с номером M. Тогда:

1)Среди буферов пула ищется содержащий Nный блок Ммого устройства. Если он найден, то фиксируется номер этого буфера (следует отметить, что в этом случае реального обращения к устройству нет, а чтением информации является предоставление информации из найденного буфера) и переходим на четвертый шаг;

2)Если поиск оказался неудачным, то в пуле осуществляется поиск буфера для чтения и размещения содержимого данного блока. Если есть свободный буфер (реально такая ситуация может быть только при старте системы), то фиксируем его номер и переходим к пункту 3. Если же свободного буфера нет, то выбирается буфер, обращений к которому не было самое длительное время. Если для него имеется установленный признак записи информации в буфер (при последнем обращении была произведена запись) — происходит запись информации из буфера на физическое устройство (если признака записи не было, то просто игнорируем содержимое), и, фиксируя номер, переходим к шагу 3.

3)Осуществляется чтение Nного блока устройства M в найденный буфер.

4)Происходит обнуление счетчика времени для данного буфера, а счетчики времени всех остальных буферов пула увеличиваем на единицу.

5)Передаем в качестве результата содержимое буфера.

Это последовательность действий, связанных с операцией чтения блока. Мы видим, что здесь есть элемент оптимизации связанный с количеством реальных обращений к физическому устройству. Запись блоков осуществляется по аналогичной схеме.

Очевидны преимущества, недостатком же является то, что система в случае буферизации является критичной к несанкционированным выключениям системы. То есть в ситуации, когда буфера системы не выгружены, а происходит нештатное завершение программ ОС, может произойти потеря данных.

Другой недостаток — то, что при буферизации разорваны во времени обращения к системе за обменом и реальные обмены. Этот недостаток проявляется в том, что если при записи в буфер система возвращает процессу результат, что запись прошла успешно, а при реальном обмене с физическом устройстве происходит сбой — эта ситуация плоха.

Для борьбы с вероятностью потери информации во время появления внештатных ситуаций система действует следующим образом — в ОС есть некоторый параметр, который определяет периоды времени, через которые происходит сброс буферов. Второе — имеется команда, которая может быть доступна пользователю — команда sync() — по этой команде осуществляется сброс данных на диск. И третье — система обладает некоторое избыточностью, позволяющей в случае потери информации произвести набор действий, которые эту информацию восстановят полностью. Если же для некоторых блоков принадлежность к файлу установить не получается, то эти блоки будут записаны в отдельные файлы.

Но на самом деле с развитием ОС и аппаратуры фатальные потери информации встречаются редко.

Мы начинали разговор о том, что существуют системные вызовы, а существуют библиотеки ввода-вывода. Библиотеки ввода-вывода также позволяют оптимизировать работу системы.

Рассмотрим стандартную библиотеку stdio.h. Концептуальная суть обменов через нее такая же, как и через системные вызовы. Но если open возвращает номер ФД, то fopen возвращает указатель на некоторую структуру — это первое. Второе и основное — многие функции сервиса, которые предоставляет библиотека реализуются внутри адресного пространства процесса, в частности такой функцией сервиса является еще один уровень буферизации ввода-вывода. Суть его в том, что на ресурсах процесса можно выделить буфер, который будет работать аналогично буферному пулу ОС и будет минимизировать обращение процесса к системным вызовам ввода-вывода.

Двойная буферизация, очевидно, вещь полезная — если мы обращаемся за обменом, например, на полблока, то буфер в процессе соберет эти половинки и системный вызов будет запрашивать уже обмен с целым блоком. Что невыгодно? То, что буферизация существует в пределах адресного пространства процесса, и мы теряем всякую синхронизацию между процессами по обмену с общим открытым файлом, так как

в каждом процессе может быть свой внутренний буфер. Но тем не менее стандартная библиотека ввода-вывода удобна. А уж пользователь берет на себя проблемы синхронизации, да и сама библиотека позволяет блокировать внутреннюю буферизацию.

Лекция 10

Говоря о двойной буферизации, следует сказать и о многоуровневой буферизации — когда ВЗУ само обладает некоторой оперативной памятью с помощью которой обеспечивает еще один уровень буферов.

При многоуровневой буферизации реальный обмен и мнимый происходят с некоторой задержкой, это может плохо сказаться на функционировании процессов.

Следует отметить, что Unix максимально экономит число обращений к внешним устройствам:

–суперблок, индексные дескрипторы — считываются в память и дальнейшая работа уже идет с ними;

–буферизация обменов, многоуровневая буферизация;

Продолжим рассматривать файловую систему Unix

Атрибуты файлов

Мы говорили об организации пользователей системы, она имеет иерархическую трехуровневую структуру

Любой пользователь, присутствующий в системе, принадлежит некоторой группе. В соответствии с иерархией пользователей определена иерархия защиты файлов. Определено понятие владельца файла. Изначально владельцем файла является пользователь (процесс пользователя), который создал этот файл. Атрибут “владелец файла” может быть изменен командой “chown”. Каждый файл имеет атрибуты защиты, причем они также иерархичны:

–права на чтение/запись/исполнение (RWX) владельца файла;

–права группы (для всех пользователей, кроме владельца);

–права всех пользователей (за исключением группы владельца и самого владельца);

Изменение прав доступа можно осуществить с помощью команды “chmod”.

Кроме атрибутов доступа каждый файл может иметь некоторые признаки, так называемые:

–t-bit: так как открытие файла — действие долгое, то для оптимизации этой процедуры введен этот атрибут, которым помечаются исполняемые файлы. Тогда при первом вызове файла за сеанс работы системы происходит копирование тела файла в область сохранения. При каждом повторном вызове файла сначала происходит просмотр области сохранения, если файл там есть, то он забирается оттуда, а в области сохранения накладных расходов на открытие файлов нет. T-bit устанавливается системным администратором, обычно это делается для наиболее часто запускаемых процессов.

–s-bit: все средства системы кому-то принадлежат, так как представляются в виде файлов, поэтому у каждой самой простой команды имеется владелец файла. Соответственно, каждая из этих команд может обращаться за чтением или записью в системные файлы. И возникает проблема — с одной стороны должна быть защита от несанкционированного доступа к файлу, а с другой стороны какие-то файлы должны быть доступны всем командам. Такие файлы помечаются s-bit’ом, суть его в следующем: владелец файла с s-bit’ом остается неизменным, но при запуске владельцу запустившего процесса предоставляются права по доступу к данным, как у владельца файла с s-bit’ом. Поясним на примере:

Предположим, имеется исполняемый файл “file”, он каким-то образом работает с файлом “file2”, в котором находится конфиденциальная (недоступная всем) информация, предположим, “file” корректирует “file2”, где хранится,

например, информация обо всех пользователях. И, в частности, он может менять пароль пользователя в системе. Если мы запустим “file” от имени пользователя “mash”, то доступ к “file2” будет заблокирован, так как “mash” не имеет права доступа к этому файлу. В противном случае, “file2” не имел бы никакой защиты. В этом случае как раз и работает s-bit, его суть в том, что владелец “file” — пользователь “root”, предположим, что его захотел запустить пользователь “mash”, если у него файла нет s-bit’a, то владельцем файла является “root”, а владельцем процесса — “mash” и в этом случае файла, которые недоступны пользователю “mash” будут недоступны процессу. Например, при таком раскладе изменить пользователю собственный пароль будет весьма затруднительно. А s-bit позволяет продлить права владельца файла на права владельца процесса. И на время работы с этим файлом процесс будет иметь все нужные права доступа. Но надо отметить, что, рассматривая наш пример, доступ к “file2” может быть осуществлен только через “file”. Напрямую “mash” по-прежнему ничего не сможет сделать.

Для установки s- и t-bit’a также существует определенная команда системы.

Итак, мы разобрались с доступом к конкретным файлам, а как же быть с каталогами? Как интерпретируются права доступа к каталогам?

–Разрешение на чтение из каталога. Это означает, что разрешен вход в каталог и открытие файлов из него;

–Разрешение на запись. Предоставляется возможность создавать и уничтожать файлы.

–Разрешение на исполнение. Возможность поиска в данном каталоге.

Предположим, мы исполняем команду “ls */mash*”. Это означает, что мы ищем в текущем каталоге подкаталоги, внутри которых есть файлы, начинающиеся с “mash”. При этом выполняется поиск файлов. Это и есть тот поиск, который ассоциируется с выполнением файла-каталога.

Структура файловой системы с точки зрения пользователя

Сразу следует отметить, что мы будем рассматривать некоторую более общую файловую систему, потому что на сегодняшний день она может изменяться от версии к версии.

Файлы “\”:

“unix” — этот файл запускается программным загрузчиком и в итоге формирует ядро системы;

“passwd” — все пользователи зарегистрированы в этом файле, то есть каждому пользователю соответствует строка в этом файле, которая содержит набор некоторых данных, разделенных между собой некоторым разделителем. В частности, эта строка содержит номер группы пользователя, она может содержать закодированный пароль на вход пользователя. Однако, современные unix’ы хранят пароли обычно в специальной защищенной базе данных, так как находятся умельцы, которые с помощью переборных задач (алгоритм кодирования паролей известен) подбирают пароли. Далее, строка содержит ФИО пользователя, его статус, домашний каталог (каталог, который устанавливается при входе пользователя), тип интерпретатора команд, с которым будет работать пользователь.

“rc” — в этом файле в текстовом виде находится набор команд, которые будут выполнены при загрузке ОС.

И так далее...

Следует заметить, что Unix подавляющее большинство своей системной информации содержит в обыкновенных текстовых файлах. Это позволяет легко изменять многие параметры ОС без использования специальных средств.

В этом же каталоге находятся еще полезные команды, которые позволяют: изменять пароли, ремонтировать файловую систему (локальную), запускать процесс тестирования и коррекции файловой системы “fsch” и т.д.

Каталоги:

“etc” — содержит стандартные файлы данных системы и команды, обеспечивающие управление некоторым уровнем функционирования системы;

“bin” — здесь находится подавляющее число команд, доступных пользователю, таких как: “rm”,”ls”,...

“mnt” — каталог, к которому можно “примонтировать” локальные файловые системы. Мы говорили об организации файловой системы на одном внешнем устройстве, но реально для систем этого мало. Обычно требуется иметь основную файловую систему и любое число локальных, расположенных на других устройствах. Они монтируются с помощью команды “mount”. Это означает, что корень локальной файловой системы после монтирования ассоциируется с этим каталогом, и доступ к любому файлу из локальной файловой системы есть лишь указание пути от корня файловой системы до соответствующего узла через каталог “mnt”.

“dev” — в этом каталоге размещаются файлы, ассоциированные с конкретными драйверами внешних устройств. К примеру, “tty01”, “tty02”, “consolp”, ... Как было сказано ранее — эти файлы не имеют содержимого, но имеют специальные поля в ИД, определяющие ссылки на таблицу драйверов и некоторую другую информацию.

“usr”: “lib” — содержит библиотеки группового пользования, как то — компилятор Си, например.

“bin” — каталог для размещения команд. Есть некоторые негласные правила, которые определяют — стоит положить команду в “\bin” или в “\usr\bin”. Обычно в “\bin” кладут стандартные команды.

“include” — здесь лежат файлы заголовков, которые, например, используются компилятором Си, когда мы употребляем команду “include <stdio.h>” файл “stdio.h” ищется как раз в “\usr\include”. Также интерес представляет подкаталог “\usr\include\sys” — здесь лежат файлы-заголовки, которые несут в себе информацию о системных функциях.

На этом мы заканчиваем рассмотрение файловой системы Unix. Подведем итог —

Файловая система Unix:

–иерархическая;

–многопользовательская;

–имеет глубокую многоярусную буферизацию при обменах с реальными устройствами;

–является информационной основой функционирования самой ОС;

–расширяемая;

–с точки зрения логической организации имеет понятную и прозрачную структуру, но это накладывает определенные условия на администрацию системы, однако, это влечет за собой проблемы распределения уровней доступа к информации и размещении информации;

Лекция 11

Управление процессами. Ядро системы.

Мы говорили о том, что вторым по значимости понятием в ОС является понятие процесса. Процесс — вещь, которая определяется по-разному. Это может быть — “упорядоченный набор команд и принадлежащих ему ресурсов”. С точки зрения ОС Unix процесс — это объект, зарегистрированный в специальной таблице процессов. Структура этой таблицы следующая: она позиционна (как практически и все таблице в Unix), то есть номер записи в таблице — есть идентификатор процесса “PID”. Формируются процесс с 0,

до N-1, где N — предельное число процессов, которые система может одновременно обрабатывать. Это параметр настройки ОС.

Рассмотрим информативную нагрузку таблицы.

В строке (записи) таблицы находится ссылка на контекст процесса, там же находится ссылка на тело процесса.

Телом процесса мы будем называть набор команд и данных, которыми оперирует процесс.

Контекст процесса — атрибут, который присутствует практически во всех ОС, в разных ОС он может называться по-разному. Контексты всех процессов размещаются в адресном пространстве ОС и содержат оперативную информацию о состоянии процесса и текущую информацию, связанную с процессом и его запуском.

Контекст содержит:

номера пользователя и группы;

указатель на ИД текущего каталога;

специфические условия работы процесса:

- обработка сигналов;

Рассмотрим это подробнее. В ОС Unix каждый процесс может послать другому процессу некоторое воздействие, которое называют “сигнал”, соответственно, если процесс-отправитель имеет право передать сигнал процессу-получателю, то при выполнении передачи в последнем возникает событие, связанное с сигналом.

Это событие очень похоже на прерывание, возникающее в аппаратуре вычислительной системы. В ОС имеется набор сигналов, которые могут передавать процессы друг другу. Перечень сигналов описан в файле “signal.h”, отправитель может подать некоторым образом команду ОС, что он передает сигнал с заданным номером процессу-получателю, процесс-получатель может прореагировать на сигнал тремя способами: прекращение выполнения и причиной завершения является пришедший сигнал; игнорирование сигнала (здесь следует отметить, что игнорировать можно далеко не все сигналы); вызывается предопределенная процессом функция, которая может выполнить какие-то действия и возврат из этой функции есть возврат в точку прихода сигнала.

- информация об открытых в процессе файлах;

- информация о текущем состоянии процесса на случай его приостановки;

Тогда, когда процесс остановлен, ОС “упрятывает” в соответствующий контекст информацию, нужную для его продолжения: режимы программы в момент приостановки, состояние регистров, адрес точки прерывания.

Тело процесса — как уже было сказано, его можно представить в виде объединения сегмента текста (кода) и сегмента данных. Развитые ОС позволяют размещать сегменты текста и данных в различных, не зависящих друг от друга местах оперативной памяти. Это хорошо, так как вместоодного большого куска памяти нам требуется два маленьких. Но еще лучше следующее — такая организация позволяет использовать повторно входимый код. Суть его в том, что допускается существование в системе еще одного процесса с собственным контекстом, сегментом данных, но у которого общий с другими процессами сегмент текста.

Если K пользователей вызывают один текстовой редактор, то в системе находится одна копия этого редактора и K копий сегмента данных и контекстов (копии, надо заметить, не идентичные). Это вещь полезная, так как отсюда сразу же можно увеличить “разум” планировщика откачки (он может, например, откачивать сегмент данных, а не сегмент текста).

Мы перечислили не все содержимое контекста, и в дальнейшем эта информация будет дополняться и уточняться.

Мы говорили, каким образом в Unix можно создать копию текущего процесса — это функция fork(), она работает следующим образом:

fork(): >0 PID сыновнего процесса (мы находимся в процессе-отце)

=0 (мы находимся в процессе-сыне)

=-1 ошибка — невозможность создать новый процесс (остаемся в процессе-отце), эта ошибка может возникнуть при недостатке места в таблице процессов, при нехватке места в системных областях данных и т.п.

Система поддерживает родственные взаимоотношения между процессами, это означает, что существуют некоторые функции, характерные для работы с процессами, которые доступны только процессам, являющимся родственниками.

При порождении сыновнего процесса с использованием fork() порожденный процесс наследует:

–окружение. При формировании процесса ему передается некоторый набор параметров переменных, используя которые процесс может взаимодействовать с операционным окружением (интерпретатором команд и т.д.);

–файлы, открытые в процессе-отце, за исключением тех, которым было запрещено передаваться специальным параметром при открытии;

–способы обработки сигналов;

–разрешение переустановки действующего идентификатора пользователя (это то, что связано с s-bit'ом)*

–установленный режим отладки. В ОС имеется возможность системными вызовами осуществлять отладку (профилирование) программы, в процессе может быть указано — разрешено или нет профилирование;

–все присоединенные разделяемые сегменты памяти. У нас есть механизм управления разделяемыми ресурсами и в качестве одного из разделяемых ресурсов может выступать оперативная память, в ней может быть выделен сегмент, к которому одновременно имеют доступ несколько процессов. При формировании сыновнего процесса эта часть памяти также будет унаследована;

–текущий рабочий каталог и корневой каталог;

Не наследуется при создании нового процесса идентификатор процесса (почему — очевидно).

Возвращаясь к функции fork(), следует заметить, что она сама по себе бессмысленна, ибо применение такому созданию точной копии процесса найти весьма сложно. Поэтому функция fork() используется совместно с группой функций exec(...). Эта группа объединяет в себе функции, которые частью своего имени имеют слово "exec" и имеют по сути дела похожее назначение, отличающееся лишь деталями (набором или интерпретацией параметров).

Суть exec в следующем: при обращении к ней происходит замена тела текущего процесса, оно заменяется в соответствии с именем исполняемого файла, указанного одним из параметров функции. Функция возвращает: -1, если действие не выполнено и код отличный от -1, если операция прошла успешно. Здесь следует отметить следующий факт — в Unix при работе с системными вызовами иногда возникают диагностические сообщения в виде кода ответа, которые невозможно разделить на конкретные причины, вызвавшие возврат этого кода. Примером этого являются коды "-1" для fork() и exec(...). Для того, чтобы обойти это неудобство, следует включить в программу файл "errno.h", и после этого при возникновении отказов в выполнении системных вызовов в переменной "errno" будет код конкретной причины отказа выполнения заказа. Всевозможные коды отказа описаны в самом "errno.h".

Давайте приведем небольшой пример:

Мы напишем программу, которая будет запускать файлы, имена которых перечислены при вызове.

```
main(argc, argv)
int argc;
char **argv;
```

```
{ int i, pid;
for (i=1; i<argc; i++) {
if (pid=fork()) continue; /* отец*/
execlp(argv[i], argv[i], (char *) 0);
}
}
```

Здесь, в случае, если pid=0, то мы замещаем тело процесса-сына процессом, имя файла которого нам передается в качестве параметра. Если же pid>0, то есть мы находимся в процессе-отце, то продолжаем создавать сыновьи процессы пока есть аргументы.

В качестве иллюстрации работы fork() можно привести следующую картинку:

Здесь процесс с PID=105 создается процессом с PID=101.

Также следует отметить, что если убивается процесс-отец, то новым отцом становится 1ый процесс ОС.

Связка fork-exec по своей мощности сильнее, чем если бы была единая функция, которая сразу бы создавала новый процесс и замещала бы его содержимое. А fork-exec позволяют вставить между ними вставить еще некоторую программу, которая будет содержать какие-то полезные действия.

Мы начали рассматривать организацию процессов. Мы на пальцах показали, как размещается информация в ОС.

В принципе, вся информация, которая отражает оперативное состояние ОС, а также программы ОС, которые управляют этой информацией, которые управляют наиболее важными устройствами составляют ядро ОС.

Ядро ОС — программа, функцией которой является управления базовыми объектами системы (для Unix это два объекта — файл и процесс). Ядро в своем теле размещает необходимые таблицы данных. И вообще ядро считается некоторой неразделяемой частью ОС. Ядро обычно работает в режиме супервизора, все остальные функции ОС могут работать и в других режимах.

Лекция 12

На прошлой лекции мы начали говорить о процессах в операционной системе UNIX. Можно однозначно говорить о том, что процессы и механизмы управления процессами в операционной системе это есть одна из принципиальных особенностей операционной системы UNIX, т.е. тех особенностей, которые отличали систему при создании и отличают ее до сих пор. Более того, несмотря на старания господина Гейтса ситуация такова, что Гейтс повторяет те программные интерфейсы, которые используются для взаимодействия управления процессами, а не фирмы разработчики UNIX-ов повторяют те интерфейсы которые появились в Windows. Очевидно, первенство операционной системы UNIX.

Мы говорили о том, что процесс в UNIX-е это есть нечто, что зарегистрировано в таблице процессов. Соответственно каждая запись в таблице процессов имеет номер. Номера идут от нуля до некоторого предельного значения, которое предопределено при установке системы. Номер в таблице процессов это есть так называемый идентификатор процесса, который в системе обозначается PID. Соответственно, подавляющее большинство действий, которые можно выполнить с процессом, выполняются при помощи указания на идентификатор процесса. Каждый процесс характеризуется контекстом процесса. Это блок данных, характеризующий состояние процесса, в том числе в этом блоке данных указывается информация об открытых файлах, о правилах обработки событий, возникающих в процессе. В этом наборе данных хранится информация, которая образуется при полном упрятывании процесса при переключении системы с процесса на процесс. Т.е., когда происходит по той или иной причине переключение выполнения с одного процесса на другой, для того, чтобы можно было

восстановить работу процесса, некий набор данных размещается в контексте процесса. Этот набор данных представляет из себя содержимое регистровой памяти, некоторые режимы, которые установила программа и в которые пришел процессор (например, содержимое регистра результата), точку возврата из прерывания. Плюс контекст содержит много полезной информации, о которой мы будем говорить.

Мы говорили о том. что в некотором смысле определено понятие тела процесса. Тело процесса состоит из двух сегментов — сегмента текста и сегмента данных. Сегмент текста это часть данных процесса, которые включают в себя код исполняемой программы. Сегмент данных это те пространства оперативной памяти, которые могут статически содержать данные. Мы говорили, что имеется возможность в системе иметь разделенные сегменты текста и сегменты данных. В свою очередь,система позволяет с одним сегментом текста связывать достаточно произвольную группу сегментов данных. Это, в частности, бывает полезно, когда в системе одновременно работают несколько одинаковых процессов.

Важная и принципиальная вещь, связанная с организацией управлением процессами — механизм fork/exec. При обращении к функции fork происходит создание нового процесса, который является копией текущего процесса. Незначительные отличия этих процессов есть в идентификаторе процессов. Возможны некоторые отличия в содержимом контекста процесса.

Функция exec позволяет заменять тело процесса, т.е. при обращении к этой функции, в случае успешного выполнения ее, тело процесса меняется на новое содержимое, которое указано в виден аргументов функции exec, точнее в виде имени некоторого файла. Мы говорили о том, что сама по себе функция fork почти бессмысленна. Смысл функции exec можно уловить, т.е. можно выполнять в рамках одного процесса несколько задач. Возникает вопрос: почему формирование этого процесса раздроблено на две функции fork и exec, чем это обосновано? Во многих системах есть одна функция, формирующая процесс с заданным содержимым. Дело в том, что при обращении к функции fork, как уже неоднократно было сказано, создается копия процесса, в том числе процесс сын наследует все те файлы, которые были открыты в процессе отце и многие другие права и привилегии. Бывает ситуация, когда не хотелось бы, чтобы наследник наследовал все особенности отца. И есть возможность между выполнением функций fork и exec выполнить какие-то действия по закрытию файлов, открытию новых файлов, по переопределению чего-то, и т.д. В частности, вы при практических занятиях должны освоить отладчик системы deb. Какова суть его работы?

Пусть есть процесс-отладчик deb, запускается процесс, который отлаживается, и передавая некоторою информацию от отладчика к отлаживаемому процессу происходит процесс отладки. Но отлаживать процесс можно только тот, который разрешил себя отлаживать. Как раз здесь используется раздвоение fork/exec. Сначала я делаю копию своего процесса deb′ , после этого я разрешаю проводить трассировку текущего процесса, а после этого я запускаю функцию exec с отлаживаемой программой. Получается ситуация, что в процессе образуется именно та программа, которую надо отладить, и она, не зная ничего, уже работает в режиме отладки.

Загрузка операционной системы и образование начальных процессов.

Сегодня мы с вами поговорим о загрузке операционной системы и образовании начальных процессов. При включении вычислительной системы, из ПЗУ запускается аппаратный загрузчик. Осуществляется чтение нулевого блока системного устройства. Из этого нулевого блока запускается программный загрузчик ОС UNIX. этот программный загрузчик осуществляет поиск и запуск файла с именем unix, который является ядром операционной системы. В начальный момент происходят действия ядра по инициализации системы. Это означает, что в соответствии с параметрами настройки системы, формируются необходимые таблицы, инициализируются некоторые аппаратные

интерфейсы (инициализация диспетчера памяти, часов, и т.д.). После этого ядро системы создает процесс №0. При этом нулевой процесс является вырожденным процессом с точки зрения организации остальных процессов. Нулевой процесс не имеет кода, он содержит только контекст процесса. Считается, что нулевой процесс активен, когда работает ядро, и пассивен во всех остальных случаях.

К моменту образования нулевого процесса в системе уже образованы таблицы, произведена необходимая инициализация, и система близка к готовности работать. Затем ядро копирует нулевой процесс в первый процесс. При этом под первый процесс уже резервируются те ресурсы, которые необходимы для полноценного процесса. Т.е. для него резервируются сегмент контекста процесса, и для него резервируется память для размещения тела процесса. После этого в первый процесс загружается программа init. При этом запускается диспетчер процессов. И ситуация такова: существует единственный процесс реально готовый к выполнению. Процесса init реально завершает запуск системы.

Запуск системы может происходить в двух режимах. Первый режим — это однопользовательский режим. В этом случае процесс init загружает интерпретатор команд shell и ассоциирует его с консольным терминалом, а также запускает стартовый командный файл /etc/rc. Этот файлсодержит произвольные команды, которые может устанавливать администратор системы, которые он считает необходимым выполнить при старте системы. Это могут быть команды, предположим, запуска программы тестирования целостности файловой системы. Это могут бытькоманды проверки расписания и в зависимости от расписания запуска процесса, который будет архивировать файловую систему и т.д. Т.е. в этом командном плане в общем случае могут быть произвольные команды, установленные администратором системы. При этом, если запускается система в однопользовательском режиме, на консольный терминал запускается интерпретатор команд shell и считается, что консольный терминал находится в режиме супервизора (суперпользователя) со всеми правами, которые можно предоставить администратору системы.

Второй режим — многопользовательский режим. Если однопользовательский режим обычно используется в ситуациях, когда в системе произошла аварийная ситуация и необходимы действия администратора системы или системного программиста, то многопользовательский режим — это штатный режим, который работает в нормальной ситуации. При многопользовательском режиме процесс init запускает для каждого активного терминала процесс getty. Список терминалов берется из некоторого текстового файла, а активность или его пассивность — это прерогатива аппаратных свойств конкретного терминала и драйвера, который обслуживает данный терминал (когда вы включаете терминал, идет сигнал по соответствующему интерфейсу о включении нового устройства, система осуществляет идентификацию этого устройства в соответствии с портом, к которому подключен этот терминал).

Процесс getty при запуске запрашивает сразу же login. Копия процесса getty работает на один сеанс работы с пользователем, т.е. пользователь подтвердил свое имя и пароль, выполняет какие-то действия, и когда он выполняет команду завершения работы, то копия процесса getty завершает свою работу. И после завершения работы процесса getty, связанного с конкретным терминалом, запускается новая копия процесса getty.

Вот такая схема. Это те нетрадиционные формы формирования процессов в UNIX-е. Нетрадиционно формируется нулевой процесс (и он сам по себе нетрадиционен), нетрадиционно формируется первый процесс (который также нетрадиционен). Все остальные процессы работают по схеме fork/exec.

Эффективные и реальные идентификаторы процесса.

С каждым процессом связано три идентификатора процесса. Первый — идентификатор самого процесса, который был получен при формировании. Второй — это т.н. эффективный идентификатор (ЭИ). ЭИ — это идентификатор, связанный с

пользователем, запустившем этот процесс. Реальный идентификатор (РИ) — это идентификатор, связанный с запущенным в виде процесса файлом (если я запускаю свой файл, то ЭИ и РИ будут одинаковы, если я запускаю чужой файл, и у этого файла есть s-бит, то в этом случае РИ будет идентификатором владельца файла и это означает, что этому процессу будут делегированы права этого владельца).

Планирование процессов в ОС UNIX.

Планирование основывается на понятии приоритета. Чем выше числовое значение приоритета, тем меньше приоритет. Приоритет процесса — это параметр, который размещен в контексте процесса, и по значению этого параметра осуществляется выбор очередного процесса для продолжения работы или выбор процесса для его приостановки. В вычислении приоритета используются две составляющие — P_NICE и P_CPU. P_NICE — это пользовательская составляющая приоритета. Она наследуется от родителя и может изменяться по воле процесса. Изменяться она может только в сторону увеличения значения (до некоторого предельного значения). Т.е. пользователь может снижать приоритет своих процессов. P_CPU — это системная составляющая. Она формируется системой следующим образом: по таймеру через предопределенные периоды времени P_CPU увеличивается на единицу для процесса, работающего с процессором (когда процесс откачивается на ВЗУ, то P_CPU обнуляется).

Процессор выделяется тому процессу, у которого приоритет является наименьшим. Упрощенная формула вычисления приоритета такова:

ПРИОРИТЕТ = P_USER + P_NICE + P_CPU

Константа P_USER различается для процессов операционной системы и остальных пользовательских процессов. Для процессов операционной системы она равна нулю, для процессов пользователей она равна некоторому значению (т.е. “навешиваются гирьки на ноги” процессам пользователя, что бы они не “задавливали” процессы системы). Это позволяет априори повысить приоритет системных процессов.

Схема планирования свопинга.

Мы говорили о том что в системе определенным образом выделяется пространство для области свопинга. Имеется проблема. Есть пространство оперативной памяти в котором находятся процессы, обрабатываемые системой в режиме мультипрограммирования. Есть область на ВЗУ предназначенная для откачки этих процессов. В ОС UNIX (в модельном варианте) свопирование осуществляется всем процессом, т.е. откачивается не часть процесса а весь. Это правило действует в подавляющем числе UNIX-ов, т.е. свопинг в UNIX-е в общем не эффективен. Упрощенная схема планирования подкачки основывается на использовании некоторого приоритета, который называется P_TIME, и который также находится в контексте процесса. В этом параметре аккумулируется время пребывания процесса в состоянии мультипрограммной обработки или в области свопинга.

При перемещении процесса из оперативной памяти в область свопинга или обратно, система обнуляет значение параметра P_TIME. Для загрузки процесса в память из области свопинга выбирается процесс с максимальным значением P_TIME. Если для загрузки этого процесса нет свободного пространства оперативной памяти, тосистема ищет среди процессов в оперативной памяти процесс, ожидающий ввода/вывода, и имеющий максимальное значение P_TIME (т.е. тот, который находился в оперативной памяти дольше всех). Если такого процесса нет то выбирается просто процесс с максимальным значением P_TIME.

Эта схема не совсем эффективна. Первая неэффективность — это то, что обмены из оперативной памяти в область свопинга происходят всем процессом. Вторая неэффективность (связанная с первой) заключается в том, что если процесс закрыт по причине заказа на обмен, то этот обмен реально происходит не со свопированным

процессом. Т.е. для того чтобы обмен нормально завершился весь процесс должен быть возвращен в оперативную память. Это тоже плохо, потому что если бы свопинг происходил блоками памяти, то можно было бы откачать процесс без той страницы, с которой надо меняться, а после обмена докачать из области свопинга весь процесс обратно в оперативную память. Современные UNIX-ы имеют возможность свопирования не всего процесса, а какой-то его части.

Лекция 13

На прошлой лекции мы с вами посмотрели каким образом может осуществляться планирование в операционной системе UNIX. Мы с вами определили, что в принципе планированию в системе поддаются два типа процессов. Первый тип — это те процессы, которые находятся в оперативной памяти и между которыми происходит разделение времени ЦП. Мы выяснили, что этот механизм достаточно прост и строится на вычислении некоторого значения приоритета. А что будет, если системная составляющая достигнет максимального значения? В этом случае у процесса просто будет низший приоритет. Второй тип процессов, процессы которые находятся на диске, поддается планированию свопинга. Любой процесс в системе может находиться в двух состояниях — либо он весь откачан на ВЗУ, либо он весь находится в оперативной памяти. И в том и в другом случае с процессом ассоциирована некоторое значение P_TIME, которое растет по мере нахождения процесса в том конкретном состоянии. Это значение обнуляется, когда процесс меняет свое состояние (то есть перекачивается в оперативную память или обратно). В свою очередь система использует P_TIME как значение некоторого приоритета (чем больше это значение, тем более вероятно, что процесс сменит свой статус).

Возникал вопрос, что является причиной для инициации действия по докачке процесса из области свопинга в оперативную память. Этот вопрос не имеет однозначного ответа, потому что в каждом UNIX-е это сделано по-своему. Есть два решения. Первое решение заключается в том, что при достижении P_TIME некоторого граничного значения, то операционная система начинает стараться его перекачать в оперативную память для дальнейшей обработки. Второе возможное решение может состоять в том, что имеется некоторое условия на системную составляющую нулевого процесса (нулевой процесс — это ядро). Как только в системе возникает ситуация, что ядро в системе начинает работать очень много, то это становится признаком того что система недогружена, т.е. у системы может быть много процессов в оперативной памяти, но они все занимаются обменом, и ЦП простаивает. Система может в этой ситуации какие-то процессы откачать, а какие-то ввести в мультипрограммную обработку.

Мы с вами говорили о том, что разные UNIX-ы могут по-разному представлять процесс в ходе его обработки. Некоторые UNIX-ы представляют тело процесса как единое целое (и код и данные) и все перемещения осуществляются согласно тому, что это единое целое. Некоторые (современные) UNIX-ы рассматривают процесс как объединение двух сегментов — сегмента кода и сегмента данных. С этим связаны проблемы запуска процессов, планирования времени процессора и планирования свопинга.

При запуске какого-то процесса система должна понять, нет ли этого процесса уже в числе запущенных, чтобы не запускать лишний сегмент кода, а привязать новые данные к уже функционирующему сегменту кода. Это определяется достаточно просто — в контексте процесса есть параметр, который содержит значение ИД файла, из которого был запущен данный процесс. И когда система пытается загрузить новый процесс (из файла), то передэтим осуществляется просмотр контекстов существующих процессов и система смотрит, нет ли уже в оперативной памяти процесса с заданным ИД, т.е. процесса, запущенного из того же файла. Аналогично происходит учет при свопировании, т.е. сначала свопированию отдаются сегменты данных, а затем могут рассматриваться

кодовые сегменты. Обращаю внимание, что при выполнении функции exec в контексте процесса сменится соответствующая информация (информация об ИД).

Напоминаю, что цель нашего курса не есть изучение того, как реализована та или иная функция в той или иной версии системы UNIX. Мы хотим посмотреть, как это можно сделать, чтобы у вас не было представления чуда, когда вы видите работающую операционную систему и вас пробирает дрожь, что это что-то от всевышнего. Все предельно просто. Есть правило, что чем более системной является программа, тем более прозрачными должны быть алгоритмы и использованные идеи. Мудреные программы живут с трудом, и это подтверждено практикой. Прозрачные программы живут долго. Пример — UNIX — прозрачная программа, и пример Windows — программа, построенная на очень высоком уровне, но там нет прозрачности на всех уровнях, и к сожалению система имеет достаточное количество особенностей, которые приводят к непредсказуемым результатам ее работы. Так везде. Если мы посмотрим языки программирования — был совершенно фантастический проект языка АДА, когда на конкурсной основе были образованы несколько профессиональных команд, которые разрабатывали язык конца XX века. Он должен был уметь делать все. Получилась очень красивая вещь. С профессиональной точки зрения этот язык во всем хорош, но он не нашел практического применения, потому что сложен. Совершенно “бездарный” язык Си существует и еще долго будет существовать. То же самое можно сказать о языках Вирта (это дядя, который придумал Паскаль, Модулу и Оберон), они тоже не прижились.

Процессы и взаимодействие процессов

С этого момента времени мы начинаем долго и упорно рассматривать различные способы взаимодействия процессов в операционной системе UNIX. Маленькое техническое добавление. Я сейчас вам продекларирую две системные функции, которыми мы будем пользоваться впоследствии. Это функции дублирования файловых дескрипторов (ФД).

int dup(fd); int dup2(fd, to_fd);
int fd; int fd, to_fd;

Аргументом функции *dup* является файловый дескриптор открытого в данном процессе файла. Эта функция возвращает -1 в том случае если обращение не проработало, и значение больше либо равное нулю если работа функции успешно завершилась. Работа функции заключается в том, что осуществляется дублирование ФД в некоторый свободный ФД. Т.е. можно как бы продублировать открытый файл.

Функция *dup2* дублирует файловый дескриптор *fd* в некоторый файловый дескриптор с номером *to_fd*. При этом, если при обращении к этой функции ФД в который мы хотим дублировать был занят, то происходит закрытие файла, работающего с этим ФД, и переопределение ФД.

Пример:

```
int fd;
char s[80];
fd = open(“a.txt”,O_RDONLY);
dup2(fd,0);
close(fd);
gets(s,80);
```

Программа открывает файл с именем *a.txt* только на чтение. ФД который будет связан с этим файлом, находится в *fd*. Далее программа обращается к функции *dup2*, в результате чего будет заменен стандартный ввод процесса на работу с файлом *a.txt*. Далее можно закрыть дескриптор *fd*. Функция *gets* прочтет очередную строку из файла *a.txt*. Вы видите, что переопределение осуществляется очень просто.

Программные каналы.

Сначала несколько слов о концепции. Есть два процесса, и мы хотим организовать взаимодействие между этими процессами путем передачи данных от одного процесса к другому. В системе UNIX для этой цели используются т.н. каналы. С точки зрения программы, канал есть некая сущность, обладающая двумя файловыми дескрипторами. Через один ФД процесс может писать информацию в канал, через другой ФД процесс может читать информацию из канала. Так как канал это нечто, связанное с файловыми дескрипторами, то канал может передаваться по наследству сыновним процессам. Это означает, что два родственных процесса могут обладать одним и тем же каналом. Это означает, чтоесли один процесс запишет какую-то информацию в канал, то другой процесс может прочесть эту информацию из этого же канала.

Особенности работы с каналом. Под хранение информации передаваемой через канал выделяется некоторый фиксированный объем оперативной памяти. В некоторых системах этот буфер может быть продолжен на внешнюю память. Что происходит, если процесс хочет записать информацию в канал, но буфер переполнен, или прочесть информацию из канала, но в буфере нет еще данных? В обоих случаях процесс приостанавливает свое выполнение и дожидается, пока не освободится место либо, соответственно, пока в канале не появится информация. Надо заметить, что в этих случаях работа процесса может изменяться в зависимости от установленных параметров, которые можно менять программно (и реакцией на эти ситуации может быть не ожидание, а возврат некоторого кода ответа).

Давайте посмотрим, как эти концепции реализуются в системе. Есть функция *pipe*. Аргументом этой функции должен быть указатель на массив двух целых переменных.

```
int pipe(pipes);
int pipes[2];
```

Нулевой элемент массива после обращения к функции *pipe* получает ФД для чтения, первый элемент этого массива получает ФД для записи. Если нет свободных ФД, то эта функция возвращает -1. Признак конца файла для считывающего дескриптора не будет получен до тех пор, пока не закрыты все дескрипторы, связанные с записью в этот канал. Рассмотрим небольшой пример:

```
char *s = “Это пример”;
char b[80];
int pipes[2];
pipe(pipes);
write(pipes[1],s, strlen(s)+1);
read(pipes[0],s, strlen(s)+1);
```

Это пример копирования строки (понятно, что так копировать строки не надо, и вообще никто функцией pipe в пределах одного процесса не пользуется). В этом примере и в последующих не обрабатываются случаи отказа. Теперь давайте рассмотрим более содержательный пример. Напишем пример программы, которая запустит и свяжет каналом два процесса.

```
main()
{
int fd[2];
pipe(fd); /* в отцовском процессе образуем два дескриптора канала */
if (fork()) /* образуем процесс-сын у которого будут те же дескрипторы */
{ /* эта часть программы происходит в процессе-отце */
dup2(fd[1],1); /* заменяем стандартный вывод выводом в канал */
close(fd[1]); /* закрываем дескрипторы канала */
close(fd[0]); /* теперь весь вывод итак будет происходить в канал */
execl(“/bin/ls”,“ls”,(char*)0); /* заменяем тело отца на ls */
} /* отсюда начинает работать процесс-сын */
dup2(fd[0],0); /* в процессе сыне все делаем аналогично */
```

close(fd[0]);
close(fd[1]);
execl("/bin/wc","wc",(char)0);*
}

Этот пример связывает конвейером две команды — *ls* и *wc*. Команда *ls* выводит содержимое каталога, а команда *wc* подсчитывает количество строк. Результатом выполнения нашей программы будет подсчет строк, выведенных командой *ls*.

В отцовском процессе запущен процесс *ls*. Всю выходную информацию *ls* загружает в канал, потому что мы ассоциировали стандартное устройство вывода с каналом. Далее мы в сыне запустили процесс *wc* у которого стандартное устройство ввода (т.е. то, откуда *wc* читает информацию) связано с дескриптором чтения из канала. Это означает, что все то, что будет писать *ls* в свое стандартное устройство вывода, будет поступать на стандартное устройство ввода команды *wc*.

Мы говорили о том, что для того, чтобы канал работал корректно, и читающий дескриптор получил признак конца файла, должны быть закрыты все пишущие дескрипторы. Если бы в нашей программе не была бы указана выделенная строка, то процесс, связанный с *wc* завис бы, потому что в этом случае функция, читающая из канала, не дождется признака конца файла. Она будет ожидать его бесконечно долго. В процессе отце подчеркнутую строку можно было бы не указывать, т.к. дескриптор закрылся бы при завершении процесса, а в процессе сыне такая строка нужна. Т.е. вывод таков, что перед завершением работы должны закрываться все дескрипторы каналов, связанные с записью.

Каналом можно связывать только родственные процессы. Технически можно связывать несколько процессов каналом, но могут возникнуть проблемы.

Лекция 14

Сигналы

Рассмотрим взаимодействие между процессами с помощью приема-передачи сигналов. Мы уже говорили о том, что в системе Unix можно построить аналогию механизму прерываний из некоторых событий, которые могут возникать при работе процессов.

Эти события, также как прерывания, однозначно определены для конкретной версии ОС. То есть определен набор сигналов. Возникновение сигналов, почти также как и возникновение прерываний может происходить по следующим причинам:

–некоторое событие внутри программы, например, деление на ноль или переполнение;

–событие, связанное с приходом некоторой информации от устройства, например, событие, связанное с передачей от клавиатуры комбинации "Ctrl+C";

–событие, связанное с воздействием одного процесса на другой, например, "sig_kill".

Система имеет фиксированный набор событий, которые могут возникать. Каждое событие имеет свое уникальное имя, эти имена обычно едины для всех версий Unix. Такие имена называются сигналами.

Перечень сигналов находится в include-файле "signal.h".

Есть сигналы, которые присутствуют практически во всех Unix, но также есть сигналы, специфичные лишь для конкретной версии Unix (FreeBSD, SCO Unix, Linux, ...) Например, в версии BSD есть сигнал приостановки работы процесса, реакцией на него является замораживание процесса, а есть сигнал, который размораживает процесс. Это сигнал FreeBSD версии.

Прототип функции обработки сигнала:

void (* signal (sig, fun)) ()

int sig;

void (* fun) ();

При обращении к signal мы передаем:

sig — имя сигнала;

fun — указатель на функцию, которая будет обрабатывать событие, связанное с возникновением этого сигнала. Она возвращает указатель на предыдущую функцию обработки данного сигнала.

Мы говорили о том, что событие, связанное с возникновением сигнала может быть обработано в системе тремя способами:

1)SIG_DEF — стандартная реакция на сигнал, которая предусмотрена системой;

2)SIG_IGN — игнорирование сигнала (следует отметить, что далеко не все сигналы можно игнорировать, например, SIG_KILL);

3)Некоторая пользовательская функция обработки сигнала.

Соответственно, указывая либо имена предопределенных констант, либо указатель на функцию, которую мы хотим определить, как функцию-обработчик сигнала, можно предопределить реакцию на тот или иной сигнал. Установка обработки сигнала происходит одноразово, это означает то, что если мы установили некоторую обработку, то по этому правилу будет обработано только одно событие, связанное с появлением данного сигнала. И при приходе в функцию обработчика устанавливается стандартная реакция на сигнал. Возврат из функции-обработчика происходит в точку прерывания процесса.

Приведем пример программы “Будильник”. Средствами ОС мы будем “заводить” будильник. Функция alrm инициализирует появление сигнала SIG_ALRM.

```
main ()
{ char s[80];
signal(SIG_ALRM, alrm); /* установка режима связи с событием SIG_ALRM на функцию alrm */
alarm(5); /* заводим будильник */
printf(“Введите имя \n”);
for (;;) {
printf(“имя:”);
if (gets(s,80) != NULL) break;
};
printf(“OK! \n”);
}
alrm () {
printf(“\n жду имя \n”);
alarm(5);
signal (SIG_ALRM,alrm);
}
```

В начале программы мы устанавливаем реакцию на сигнал SIG_ALRM на функцию alrm, далее мы заводим будильник, запрашиваем “Введите имя” и ожидаем ввода строки символов. Если ввод строки задерживается, то будет вызвана функция alrm, которая напомнит, что программа “ждет имя”, опять заведет будильник и поставит себя на обработку сигнала SIG_ALRM еще раз. И так будет до тех пор, пока не будет введена строка.

Здесь имеется один нюанс: если в момент выполнения системного вызова возникает событие, связанное с сигналом, то система прерывает выполнение системного вызова и возвращает код ответа, равный “-1”. Это мы можем также проанализировать по функции errno.

Надо отметить, что одноразово устанавливается только “свой” обработчик. Дефолтный обработчик или игнорирование устанавливается многоразово, то есть его не надо каждый раз подтверждать после обработки сигнала.

Еще две функции, которые необходимы нам для организации взаимодействия между процессами:

int kill(int pid, sig) — это функция передачи сигнала процессу. Она работает следующим образом: процессу с номером pid осуществляется попытка передачи сигнала, значение которого равно sig. Соответственно, сигнал может быть передан в рамках процессов, принадлежащих одной группе. Код ответа: -1, если сигнал передать не удалось, пояснение опять же можно найти в errno. Функция kill может использоваться для проверки существования процесса с заданным идентификатором. Если функция выполняется с sig=0, то это тестовый сигнал, который определяет — можно или нетпередать процессу сигнал, если можно, то код ответа kill отличен от "-1". Если pid=0, то заданный сигнал передается всем процессам, входящим в группу.

int wait(int *wait_ret). Ожидание события в сыновнем процессе. Если сыновнего процесса нету, то управление возвращается сразу же с кодом ответа "-1" и расшифровкой в errno. Если в процессе-сыне возникло событие, то анализируются младшие 16 бит в значении wait_ret:

Если сын приостановлен (трассировка или получение сигнала), тогда старшие 8 бит wait_ret — код сигнала, который получил процесс-сын, а младшие содержат код 0177.

Если сыновий процесс успешно завершился через обращение к функции exit. Тогда младшие 8 бит равны нулю, а старшие 8 бит равны коду, установленному функцией exit.

Если сын завершился из-за возникновения у него необрабатываемого сигнала, то старшие 8 бит равны нулю, а младшие — номер сигнала, который завершил процесс.

Функция wait возвращает идентификатор процесса в случае успешного выполнения и "-1" в противном случае. Если одно из перечисленных событий произошло до обращения к функции, то результат возвращается сразу же, то есть никакого ожидания не происходит, это говорит о том, что информация о событиях в процессе безвозвратно не теряется.

Давайте рассмотрим еще один пример. Наш будильник будет уже многопроцессный.

```
alr() { printf("\n Быстрее!!! \n");
signal (SIG_ALRM, alr);
}
main () { char s[80; int pid;
signal(SIG_ALRM, alr);
if (pid=fork()) for (;;)
{sleep(5); kill(pid, SIG_ALRM);}; /* приостанавливаем процесс на 5 секунд и отправляем сигнал SIG_ALRM процессу сыну */
print("имя?");
for (;;) {printf("имя?");
if gets(s,80)!=NULL) break;
}
printf("OK!\n");
kill(getpid(), SIG_KILL); /* убиваем зациклившегося отца */
}
```

Следует заметить, что в разных версиях Unix имена сигналов могут различаться.

Наша программа реализуется в двух процессах.

Как и в предыдущем примере имеется функция реакции на сигнал alr(), которая выводит на экран надпись и переустанавливает функцию реакции на сигнал опять же на себя. В основной программе мы также указываем alr(), как реакцию на SIG_ALRM. После этого мы запускаем сыновий процесс, и в отцовский процесс (бесконечный цикл) "засыпает" на 5 единиц времени, после чего сыновнему процессу будет отправленсигнал SIG_ALRM. Все, что ниже цикла будет выполняться в процессе-сыне: мы ожидаем ввода строки, если ввод осуществлен, то происходит убиение отца (SIG_KILL).

Таким образом, мы описали базовые средства взаимодействия процессов в Unix: порождение процесса, замена тела процесса, взаимодействие при помощи передач/приемов сигналов.

Замечание: мы говорим о некотором обобщенном Unix, реальные Unix'ы могут иметь некоторые отличия друг от друга. На сегодняшний день имеются достаточно формализованные стандарты на интерфейсы ОС, в частности Unix. Это POSIX-standard, то есть были проведены работы по стандартизации интерфейсов всех уровней для открытых систем. Основной задачей является унификация работы с системами, как на уровне запросов от пользователя, так и на уровне системных вызовов. В принципе, на сегодняшний день практически все разработчики ОС стараются привести свои системы к стандарту POSIX. В частности, Microsoft объявила, что системные вызовы и работа с файлами в Windows NT происходит в стандарте POSIX. Но так или иначе реальные коммерческие системы от этого стандарта отходят.

Второе замечание: мы начали рассматривать примеры, но крайне важно, чтобы все эти примеры были реализованы на практике, дабы убедиться, что они работают, посмотреть как они работают и добиться этой работы, так как версии Unix могут не совпадать. Для этого следует посмотреть мануалы и, если надо, подправить программы.

Лекция 15

Трассировка

Трассировка — это возможность одного процесса управлять кодом и выполнением другого процесса.

Давайте для начала посмотрим на действия, которые выполняются при отладке:

1)Установка контрольной точки;

2)Обработка ситуации, связанной с приходом в контрольную точку;

3)Чтение/запись информации в отлаживаемой программе;

4)Остановка/продолжение выполнения отлаживаемого процесса;

5)Шаговый режим отладки (режим, при котором отлаживаемая программа будет останавливаться после выполнения каждой инструкции);

6)Передача управления на произвольную точку отлаживаемой программы;

7)Обработка аварийных остановок;

Это семь позиций, которые реализуются почти в любом средстве отладки с точностью до добавленных или удаленных возможностей будь-то Windows 95/NT, Unix, OS/2, DOS и т.д. Есть некоторый джентельментский набор, который обычно предоставляет отладчик. Теперь посмотрим,какими средствами можно организовать выполнение этих функций в ОС Unix.

Итак, есть функция, которая называется

int ptrace(int op, pid, addr, data)

Суть этой функции заключается в следующем: ptrace в подавляющем большинстве случаев работает в отцовском процессе, и через возможности ptrace организуется управление сыном. В общем случае нельзя трассировать любой процесс. Для того, чтобы процесс можно было отлаживать, процесс-сын должен подтвердить согласие на собственную трассировку, в последнем случае следует в самом начале выполнения вызвать ptrace с кодом операции равном нулю (op=0), этот вызов разрешает в дальнейшем трассироваться процессом-отцом. После этого в сыновнем процессе обращений к ptrace может и не быть.

Посмотрим, какие возможности есть у отцовского процесса по управлению процессом-сыном. Для начала разберем параметры функции ptrace:

op — код операции;

pid — PID сыновнего процесса;

addr — некоторый адрес внутри сыновнего процесса (будет объяснено при описании параметра op);

data — слово информации;

op	addr	data	
0			Разрешение на трассировку себя
1,2			ptrace возвращает слово, расположенное по адресу, заданному в параметре addr. Это есть чтение по адресу. (Два значения задается из-за того, что в рамках процесса может быть двойная адресация — по коду и по данным)
3			Чтение из контекста процесса. Обычно речь идет о доступе к информации из контекста, сгруппированной в некоторую структуру. И в этом случае параметр addr указывает на смещение внутри данной структуры. В структуре мы можем найти регистры, текущее состояние процесса, счетчик адреса и т.д. Этот набор немного варьируется от системы к системе.
4,5			Запись данных, расположенных в data по адресу addr. Этот параметр также двойной по причине возможного разделения кода и данных процесса. Если происходит ошибка, то ptrace возвращает “-1” (уточнение ошибки в errno)
6			Запись данных из data в контекст процесса. Роль addr такая же, как и при op=3. То есть мы можем изменить регистры трассируемого процесса, в том числе регистр счетчика команд (сделать переход внутри процесса).
7			Продолжение выполнения трассируемого процесса. Тут есть некоторые тонкости. Если трассируемый процесс был по какой-то причине остановлен. Пока он стоит к нему могут прийти какие-то сигналы от других процессов. Что делать, когда к остановленному процессу пришли сигналы? Здесь играют роль параметры data и addr.
		0	Процесс, который был приостановлен, продолжит свое выполнение и при этом все пришедшие и необработанные сигналы будут проигнорированы.
		N_SIG	Будет смоделирована ситуация прихода сигнала с заданным номером N_SIG. Все остальные сигналы будут проигнорированы.
	=1		Сыновний процесс продолжает выполняться с места, в котором он был приостановлен.
	>1		Происходит переход по адресу (абсолютному адресу) на addr внутри процесса. (goto addr)
8			Завершение трассируемого процесса.

9			Установка бита трассировки. Этот бит позволяет делать пошаговое выполнение команд. После выполнения каждого машинного кода происходит реакция на сигнал SIG_TRAP.

Мы рассмотрели функцию ptrace в некоторой модельной нотации. Модельная нотация заключается в том, что в разных системах эта функция может иметь несколько другую интерфейсную часть (одни константы op могут быть присутствовать, другие отсутствовать или иметь другой номер, отличается доступ к контексту процесса и т.п.)

Все действия, о которых мы говорили, выполняются при остановленном сыновнем процессе (он может быть остановлен из-за ошибки (деление на ноль, например), прихода сигнала от отца или другого процесса и т.п.) Для того, чтобы процесс-отец мог остановить сына, то он должен выполнить следующие действия:

1)Послать сыну сигнал SIG_TRAP;

2)Выполнить wait;

3)После того, как сын остановился, отец получает соответствующий код ответа от функции wait.

После этого считается, что можно выполнять все действия, которые были описаны выше.

Установка контрольной точки. Считается, что в отладчике имеется некоторая таблица, которая содержит информацию о контрольных точках внутри отлаживаемого процесса:

Номер контрольной точки	Адрес контрольной точки	Сохраненное слово	Счетчик
...	...	_...	...

Что происходит при установке контрольной точки? При запросе на установку контрольной точки по заданному адресу отладчик читает содержимое отлаживаемого процесса по этому адресу и записывает это содержимое в таблицу (Сохраненное слово). Затем, отладчик по указанному адресу записывает машинную команду, которая вызовет возникновение события, связанного с некоторым сигналом (например, команда деления на ноль, вызов программного прерывания или любая команда, вызывающая событие, которое должно быть известно), после этого мы можем запустить отлаживаемый процесс (ptrace, op=7). В тот момент, когда управление придет на адрес, где мы установили контрольную точку, произойдет прерывание процесса и событие, связанное с известной установкой. Для отладчика это будет видно следующим образом — после запуска на продолжение отлаживаемого процесса выполняется функция wait, которая ожидает события в отлаживаемом процессе. Событие произошло, его причиной был некоторый сигнал. Действия отладчика:

1)Не совпадает ли сигнал, который прервал процесс с сигналом, который является контрольным сигналом. Если не совпадает, то произошла какая-то ситуация, которая обрабатывается по-своему. Если же совпадает, то мы, скорее всего, попали на контрольную точку (предположим, что у нас это сигнал деления на ноль).

2)После этого мы читаем из контекста процесса точку останова — если она совпала с одной из позиций таблицы отладчика, то это означает, что мы совершенно точно попали на контрольную точку. Если такого совпадения не обнаружено, то это означает, что тревога ошибочна и обработка сигнала уходит по другому направлению.

3)Если все-таки мы оказались на контрольной точке, то можем выполнить какие-то действия над отлаживаемым процессом (чтение/запись регистров, ячеек памяти и прочее).

4)Наконец, нам хочется продолжить выполнение программы, но при этом мы бы хотели сохранить контрольную точку. Что может сделать отладчик? Он восстанавливает оригинальное содержимое машинного слова. Включает режим трассировки и запускает

программу с прерванного адреса. После чего выполняется команда, которую мы восстановили и процесс приостанавливается (так как включена трассировка) на следующей команде. Теперь мы опять по нужному адресу пишем команду деления на ноль и продолжаем процесс уже вне режима трассировки. Таким образом мы восстановили контрольную точку.

Снятие контрольной точки происходит тривиально — восстанавливается содержимое по соответствующему адресу и строка данной точки выбрасывается из таблицы.

Контрольные точки могут устанавливаться на какое-то количество прохождений — в таблицу включается позиция счетчика, куда вносится какое-то число, и при каждом прохождении через данную контрольную точку из счетчика будет вычитаться единица, а по достижении нуля точка будет снята.

Итак, мы обсудили с точностью до некоторых деталей структуру адресного отладчика.

Если возникает необходимость символьной отладки (отладки в терминах языка высокого уровня), то добавляются некоторые средства, позволяющие определять адреса и свойства переменных и адреса операторов. В этом случае, например, команда чтения языковой переменной программы будет осуществляться следующим образом: если надо найти переменную с именем name, то отладчик ищет ее в таблице, проверяет области видимости и смотрит ее атрибуты. Если переменная статическая, то выбирается ее адрес, и мы обращаемся через ptrace к соответствующей адресной ячейке. Если переменная автоматическая, то через соответствующее смещение относительно стека, мы, читая из контекста вершину стека и прибавляя смещение, читаем нужную переменную. Если переменная регистровая, то в таблице содержится номер регистра, на котором она размещена, соответственно, для ее чтения нам достаточно прочитать из контекста соответствующий регистр.Для изменения содержимого переменных используется этот же аппарат.

Приведем небольшой пример трассировки:

Процесс-сын:

```
int main()
{ int i;
return i/0;
}
```

Процесс отец:

(Мы будем писать программу в нотации ОС FreeBSD)

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/ptrace.h>
#include <sys/wait.h>
#include <machine/reg.h>
int main(int argc; char *argv[])
{ pid_t pid;
int status;
struct reg REG;
switch (pid=fork()) {
case -1: perror(“ошибка fork”); exit(); /* ошибка создания сына */
case 0: ptrace(PT_TRACE_ME,0,0,0); /* если в сыне, то разрешаем трассировку */
execl(“son”,”son”,0); /* и замещаем тело процесса */
```

```
/* после входа в сына, процесс-сын будет остановлен с сигналом SIG_TRAP, это правила игры — как только отлаживаемый процесс меняет тело, то происходит приостановка в самом начале */
default: break; /* если в отце, то выходим из switch */
}
/* в отце */
for (;;) {
wait(&status); /* ждем, когда отлаживаемый процесс приостановится (первый раз это произойдет сразу после замены тела сыновнего процесса), то есть, когда в нем возникнет событие */
ptrace(PT_GET_REGS,pid, (caddr_t) &REGS, 0); /* теперь мы можем прочесть и вывести содержимое некоторых регистров */
printf("EIP=%O.8X\+ESP=%O.8X\n", REG.r_eip, REG.r_esp);
if (WIFSTOPPED(status) || WIFSIGNALED(status)) {
/* Выше написанные макросы описаны в файлах include, они обрабатывают структуру, возвращаемую функцией wait, если у нас условия приостановки нормальные, то начинаем разбирать причину */
printf("сигнал:");
switch(WSTOPSIG(status)) {
case SIGINT: printf("INT\n"); break;
case SIGTRAP: ...
...
defalt: printf("%d", WSTOPSIG(status));
};
if (WSTOPSIG(status)!=SIGTRAP) exit(1);
};
if (WIFEXITED(status)) { /* если отлаживаемый процесс завершился */
printf("процесс закончен с кодом = %d\n", WEXITSTATUS(status));
exit};
ptrace (PT_CONTINUE, pid, (caddr_t) 1, 0); /* продолжаем выполнение трассируемого процесса */
}
exit(0);
}
```

При первой итерации бесконечного цикла мы остановимся при получении сыном сигнала SIG_TRAP, посмотрим, не закончился ли наш процесс нормально (а он нормально закончиться не может, так как делит на ноль), то мы обратимся к ptrace, которая продолжит выполнение трассируемого процесса. На второй итерации мы попадем на событие деления на ноль. Таким образом, мы получим две порции информации — первая связана с самой первой приостановкой сыновнего процесса, когда заменяется его тело, а вторая — связана с делением на ноль.

Лекция 16

Теперь мы с вами обсудим некоторые дополнительные возможности по организации управления ходом вычисления в процессах Unix.

Нелокальные переходы

Может получиться так, что возникает необходимость предоставления в процессе возможностей перезапуска каких-то из веточек процесса при возникновении некоторых ситуаций. Предположим, имеется некоторый процесс, который занимается обработкой достаточно больших наборов данных, и будет работать следующим образом:

В начальный момент времени процесс получает набор данных и начинает выполнять вычисления. Известно, что при некоторых наборах данных возможно возникновение внештатных ситуаций, например, переполнения или деления на ноль. Мы бы хотели написать программу, которая бы при возникновении внештатных ситуаций обрабатывала бы их, загружала новые данные, переходила в начальную точку процесса и выполняла бы вычисления с другим набором данных. То есть возникает необходимость в повторном использовании некоторых цепей программы. Для решения такой задачи мы должны уметь делать:

–обрабатывать ситуации, возникающие в процессе. Но для этого у нас есть функция signal;

–возвращаться в некоторые помеченные в программе точки не через последовательность входов и выходов функций, а через безусловную передачу управления (аналог GOTO). Почему? Потому что просто механизм обработки сигналов не позволит нам корректно работать с поставленной задачей — можно было бы написать функцию обработки сигналов, которая по возникновениюнужного сигнала делала бы повторный вызов всей программы. Но это некорректно, так как при вызове функции обработчика сигнала фиксируется состояние стека, и в общем случае система ожидает корректного выхода из функции обработчика (через return или последнюю скобку). Таким образом мы будем накапливать невозвращенные части стека, что приведет к деградации системы.

Для решения второй проблемы в Unix имеется две функции, которые обеспечивают нелокальные переходы:

#include <setjmp.h>

int setjmp(jmp_buf env);

Эта функция фиксирует точку обращения к этой функции, то есть в структуре данных, связанных с переменной env сохраняется текущее состояние процесса в точке обращения к setjmp, в том числе состояние стека. При обращении к этой функции она возвращает нулевое значение.

void longjmp(jmp_buf env, int val);

Нелокальный переход. При обращении к longjmp переходит передача управления на точку, атрибуты которой зафиксированы в env.

Если мы сделали setjmp, а затем откуда-то longjmp с той же переменной env, то мы вернемся на обращение к функции setjmp и в качестве кода ответа setjmp получим значение val.

То есть setjmp — это декларация некоторой точки, на которую мы можем затем вернуться с помощью, а longjmp — переход на эту точку, где параметр val задает код ответа setjmp.

Пример:

```
#include <setjmp.h>
jmp_buf save; /* объявляем глобальный буфер save */
main()
{ int ret;
switch(ret=setjmp(save)){
case 0: printf(“до нелокального перехода\n”);
a();
printf(“после нелокального перехода\n”); /* этот текст никогда не будет напечатан */
default: break;
}
}
a() {longjmp(save,1)};
```

Рассмотрим функцию main() — в переключателе мы обращаемся к setjmp, она запомнит состояние процесса на тот момент и вернет ноль. После этого мы перейдем по варианту связанному с нулем — печатаем текст и вызываем a(). В a() мы вызываем longjmp(save,1), после этого мы попадем опять на переключатель, но на этот раз переменная ret будет равна единице. Произойдет завершение процесса.

Вообще говоря, это некорректная возможность ОС, так как некорректно входить в блочные структуры не сначала и выходить не через конец. Но такие возможности есть и они полезны.

Как работает длинный переход со стеком? Он не запоминает стек, он запоминает указатель стека и восстанавливает его. Конечно, мы можем смоделировать ситуацию, в которой переход будет работать некорректно, например, вызовем функцию, в ней сделаем setjmp, выйдем из функции, как-то поработаем дальше и попробуем сделать longjmp на функцию, из которой уже вышли. Информация в стеке будет уже потеряна и наш переход приведет к ошибке. Такие ситуации отдаются на откуп программистам.

Нелокальный переход работает в пределах одного процесса.

До этого мы говорили о взаимодействии между родственными процессами (отцом и сыном, детьми одного отца и т.п.). Реально же Unix имеет набор средств, поддерживающих взаимосвязь между произвольными процессами. Один из таких механизмов — система межпроцессного взаимодействия (IPC- interprocess communication). Суть этой системы заключается в следующем — имеется некоторое количество ресурсов, которые называют в системе разделяемыми. К одному и тому же разделяемому ресурсу может быть организован доступ произвольного количества произвольных процессов. При этом возникает некоторая проблема именования ресурсов. Если мы вспомним каналы, то в них за счет наследования нужные файловые дескрипторы были известны и с именованием проблем не возникало.

Но это свойство родственных связей. В системе IPC ситуация совершенно иная — есть некоторый ресурс, в общем случае произвольный, и к этому ресурсу могут добираться все кому не лень — все, кто может именовать этот ресурс. Для именования такого ресурсов в системе предусмотрен механизм генерации так называемых ключей. Суть его в следующем — по некоторым общеизвестным данным (текстовые строки или цифровые наборы) генерируется уникальный ключ, который ассоциируется с разделяемым ресурсом,

соответственно, если мы подтверждаем этот ключ и созданный разделяемый ресурс доступен для моего процесса, то мы можем работать с этим ресурсом.

Следующее концептуальное утверждение — разделяемый ресурс создается некоторым процессом-автором. Это к проблеме первичного возникновения ресурса. Автор определяет основные свойства (размер, например) и права доступа. Права доступа разделяются на три категории — доступ автора, доступ всех процессов, имеющих тот же идентификатор, что и автор, и все остальные.

Система позволяет некоторому процессу создать ресурс, защитить его некоторым ключом и забывать про него. Затем, все те, кто знает ключ, могут работать с этим процессом.

Сразу возникает вопрос — а если сразу трое подошли к ресурсу? То есть очевидна проблема синхронизации доступа к разделяемому ресурсу.

Мы с вами будем рассматривать конкретные средства IPC, которые будем рассматривать далее. А пока отмечу, что IPC поддерживает три разновидности разделяемых ресурсов:

1)Разделяемая память. Возможность иметь в нескольких произвольных процессах общее поле оперативной памяти и работать с ним, как с неким массивом, на который имеется указатель. Мы видим, что проблема синхронизации здесь огромна, хотя базово никакой синхронизации не предусмотрено;

2)Механизм передачи сообщений. В качестве разделяемого ресурса — очередь сообщений. Суть в том, что есть объект, который называется очередью сообщений, он может содержать произвольное в пределах разумного количество сообщений разной длины и типа. Тип сообщения — атрибут, который приписывается сообщению. Соответственно, очередь сообщений может рассматриваться как единое хронологическое объединение, так и множество объединенных по типам подочередей. Есть возможность читать и писать сообщения. Проблема синхронизации также имеется;

3)Семафоры. Это есть нечто, что позволяет синхронизировать доступ к разделяемым ресурсам;

Вот об этом мы будем говорить далее.

Лекция 17

Interprocess Communication

Мы с вами говорили, что далее речь пойдет о разделяемых ресурсах, доступ к которым может осуществляться со стороны произвольных процессов, в общем случае, в произвольном порядке. Эти ресурсы доступны любому процессу, а процессы не обязательно должны быть родственными. При наличии такой схемы возникают две принципиальные проблемы:

1)Именование;

2)Синхронизация;

Проблемы именования связаны с тем, что родственных связей нет и по наследству передать ничего нельзя.

Если проблема именования решена, то возникает проблема синхронизации доступа — как организовать обмен с ресурсами, чтобы этот обмен был корректным. Если у нас есть, например, ресурс “оперативная память”, то когда один процесс еще не дописал информацию, а другой процесс уже прочитал весь блок, то возникает некорректная ситуация.

Решения этих проблем мы и будем рассматривать.

Проблема именования решается за счет ассоциирования с каждым ресурсом некоторого ключа. В общем случае это целочисленное значение. То есть при создании разделяемого ресурса его автор приписывает ему номер и определяет права доступа к этому ресурсу. После этого любой процесс, который укажет системе, что он хочет общаться с разделяемым ресурсом с ключом N, и обладает необходимыми правами доступа, будет допущен для работы с этим ресурсом.

Однако такое решение не является идеальным, так как вполне возможна коллизия номеров — когда совпадают номера разделяемых ресурсов. В этом случае процессы будут путаться, что неизбежно приведет к ошибкам. Поэтому в системе предусмотрено стандартное средство генерации уникальных ключей. Для генерации уникального ключа используется функция ftok

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(char *s, char c);
```

Суть ее действия — по текстовой строке и символу генерируется уникальное для каждой такой пары значение ключа. После этого сгенеренным ключом можно пользоваться как для создания ресурса, так и для подтверждения использования ресурса. Более того, для исключения коллизий, рекомендуется указывать в качестве параметра указателя на строку путь к некоторому своему файлу. Второй аргумент — символьный, который позволяет создавать некоторые варианты ключа, связанного с этим именем, этот аргумент называется проектом (project). При таком подходе можно добиться отсутствия коллизий.

Давайте посмотрим конкретные средства работы с разделяемыми ресурсами.

Разделяемая память.

Общая схема работы с разделяемыми ресурсами такова — есть некоторый процесс-автор, создающий ресурс с какими-либо параметрами. При создании ресурса разделяемой памяти задаются три параметра — ключ, права доступа и размер области памяти. После создания ресурса к нему могут быть подключены процессы, желающие работать с этой памятью. Соответственно, имеется действие подключения к ресурсу с помощью ключа, который генерируется по тем же правилам, что и ключ для создания ресурса. Понятно, что здесь имеется момент некоторой рассинхронизации, который связан с тем, что потребитель разделяемого ресурса (процесс, который будет работать с ресурсом, но не является его автором) может быть запущен и начать подключаться до запуска автора ресурса. В этой ситуации особого криминала нету, так как имеются функции управления доступом к разделяемому ресурсу, с использованием которых можно установить некоторые опции, определяющие правила работы функций, взаимодействующих с разделяемыми ресурсами. В частности, существует опция, заставляющая процесс дождаться появления ресурса. Это также, может быть, не очень хорошо, например, автор может так и не появиться, но другого выхода нету, это есть некоторые накладные расходы. Вот в общих словах — что есть что.

Давайте рассмотрим те функции, которые предоставляются нам для работы с разделяемыми ресурсами.

Первая функция — создание общей памяти.

int shmget (key_t key, int size, int shmemflg);

key — ключ разделяемой памяти

size — размер раздела памяти, который должен быть создан

shmemflg — флаги

Данная функция возвращает идентификатор ресурса, который ассоциируется с созданным по данному запросу разделяемым ресурсом. То есть в рамках процесса по аналогии с файловыми дескрипторами каждому разделяемому ресурсу определяется его идентификатор. Надо разделять ключ — это общесистемный атрибут, и идентификатор, используя который мы работаем с конкретным разделяемым ресурсом в рамках процесса.

С помощью этой функции можно как создать новый разделяемый ресурс “память” (в этом случае во флагах должен быть указан IPC_CREAT)?, а также можно подключиться к существующему разделяемому ресурсу. Кроме того, в возможных флагах может быть указан флаг IPC_EXECL, он позволяет проверить и подключиться к существующему ресурсу — если ресурс существует, то функция подключает к нему процесс и возвращает код идентификатора, если же ресурс не существует, то функция возвращает -1 и соответствующий код в errno.

Следующая функция — доступ к разделяемой памяти:

char *shmat(int shmid, char *shmaddr, int shmflg);

shmid — идентификатор разделяемого ресурса

shmaddr — адрес, с которого мы хотели бы разместить разделяемую память, при этом, если его значение — адрес, то память будет подключена, начиная с этого адреса, если его значение — нуль, то система сама подберет адрес начала. Также в качестве значений этого аргумента могут быть некоторые предопределенные константы, которые позволяют организовать, в частности выравнивание адреса по странице или началу сегмента памяти.

shmflg — флаги. Они определяют разные режимы доступа, в частности, SHM_RDONLY.

Эта функция возвращает указатель на адрес, начиная с которого будет начинаться запрашиваемая разделяемая память. Если происходит ошибка, то возвращается -1.

Хотелось бы немного поговорить о правах доступа. Они реально могут использоваться и корректно работать не всегда. Так как, если аппаратно не поддерживается закрытие области данных на чтение или на запись, то в этом случае могут

возникнуть проблемы с реализацией такого рода флагов. Во-первых, они не будут работать, так как мы получаем указатель и начинаем работать с указателем, как с указателем, и общая схема здесь не предусматривает защиты. Второе, можно программно сделать так, чтобы работали флаги, но тогда мы не сможем указывать произвольный адрес, в этом случае система будет подставлять и возвращать в качестве адрес разделенной памяти некоторые свои адреса, обращение к которым будет создавать заведомо ошибочную ситуацию, возникнет прерывание процесса, во время которого система посмотрит — кто и почему был инициатором некорректного обращения к памяти, и если тот процесс имеет нужные права доступа — система подставит нужные адреса, иначе доступ для процесса будет заблокирован. Это похоже на установку контрольной точки в программе при отладке, когда создавалась заведомо ошибочная ситуация для того, чтобы можно было прервать процесс и оценить его состояние.

Третья функция — открепление разделяемой памяти:

int shmdt(char *shmaddr);

shmaddr — адрес прикрепленной к процессу памяти, который был получен при подключении памяти в начале работы.

Четвертая функция — управление разделяемой памятью:

int shmctl(int shmid, int cmd, struct shmid_ds *buf);

shmid — идентификатор разделяемой памяти

cmd — команда управления. В частности, могут быть: IPC_SET (сменить права доступа и владельца ресурса — для этого надо иметь идентификатор автора данного ресурса или суперпользователя), IPC_STAT (запросить состояние ресурса — в этом случае заполняется информация в структуру, указатель на которую передается третьим параметром, IPC_RMID (уничтожение ресурса — после того, как автор создал процесс — с ним работают процессы, которые подключаются и отключаются, но не уничтожают ресурс, а с помощью данной команды мы уничтожаем ресурс в системе)

Это все, что касается функций управления разделяемой памятью.

Передача сообщений.

Следующим средством взаимодействия процессов в системе IPC — это передача сообщений. Ее суть в следующем: в системе имеется так называемая очередь сообщений, в которой каждое сообщение представляет из себя структуру данных, с которой ассоциирован буфер, содержащий тело сообщения и признак, который называется типом сообщения. Очередь сообщений может быть рассмотрена двояко:

–очередь рассматривается, как одна единственная сквозная очередь, порядок сообщений в которой определяется хронологией их попадания в эту очередь.

–кроме того, так как каждое сообщение имеет тип (на схеме — буква рядом с номером сообщения), то эту очередь можно рассматривать, как суперпозицию очередей, связанную с сообщениями одного типа.

Система IPC позволяет создавать разделяемый ресурс, называемый “очередь сообщений” — таких очередей может быть произвольное количество. По аналогии с разделяемой памятью — мы можем создать очередь, подключиться к ней, послать сообщение, принять сообщение, уничтожить очередь и т.д. Рассмотрим функции работы с очередями сообщений:

Создание очереди сообщений:

int msgget(key_t key, int flags);

В зависимости от флагов при обращении к данной функции либо создается разделяемый ресурс, либо осуществляется подключение к уже существующему.

Отправка сообщения:

int msgsnd(int id, struct msgbuf *buf, int size, int flags);

id — идентификатор очереди сообщения;

struct msgbuf {long type; char mtext[s]} *buf — первое поле — тип сообщения, а второе — указатель на тело сообщения;

size — размер сообщения, здесь указывается размер сообщения, размещенного по указателю buf;

flags — флаги, в частности, флагом может быть константа IPC_NOWAIT. При наличии такого флага будут следующие действия — возможна ситуация, когда буфера, предусмотренные системой под очередь сообщений, переполнены. В этом случае возможны два варианта — процесс будет ожидать освобождения пространства, если не указано IPC_NOWAIT, либо функция вернет -1 (с соответствующим кодом в errno), если было указано IPC_NOWAIT.

Прием сообщения:

int msgrcv(int id, struct msgbuf *buf, int size, long type, int flags);

id — идентификатор очереди;

buf — указатель на буфер, куда будет принято сообщение;

size — размер буфера, в котором будет размещено тело сообщения;

type — если тип равен нулю, то будет принято первое сообщение из сквозной очереди, если тип больше нуля, то в этом случае будет принято первое сообщение из очереди сообщений, связанной с типом, равным значению этого параметра.

flags — флаги, в частности, IPC_NOWAIT, он обеспечит работу запроса без ожидания прихода сообщения, если такого сообщения в момент обращения функции к ресурсу не было, иначе процесс будет ждать.

Управление очередью:

int msgctl(int id, int cmd, struct msgid_dl *buf);

id — идентификатор очереди;

cmd — команда управления, для нас интерес представляет IPC_RMID, которая уничтожит ресурс.

buf — этот параметр будет оставлен без комментария.

Мы описали два средства взаимодействия между процессами.

Что же мы увидели? Понятно, что названия и описания интерфейсов мало понятны. Прежде всего следует заметить то, что как только мы переходим к вопросу взаимодействия процессов, у нас возникает проблема синхронизации. И здесь мы уже видим проблемы, связанные с тем, что после того, как мы поработали с разделяемой памятью или очередью сообщений, в системе может оставаться “хлам”, например, процессы, которые ожидают сообщений, которые в свою очередь не были посланы. Так, если мы обратились к функции получения сообщений с типом, которое вообще не пришло, и если не стоит ключ IPC_NOWAIT, то процесс будет ждать его появления, пока не исчезнет ресурс. Или мы можем забыть уничтожить ресурс (и система никого не поправит) — этот ресурс останется в виде загрязняющего элемента системы.

Когда человек начинает работать с подобными средствами, то он берет на себя ответственность за все последствия, которые могут возникнуть. Это первый набор проблем — системная синхронизация и аккуратность. Вторая проблема — синхронизация данных, когда приемник и передатчик работают синхронно. Заметим, что самый плохой по синхронизации ресурс из рассмотренных нами — разделяемая память. Это означает, что корректная работа с разделяемой памятью не может осуществляться без использования средств синхронизации, и, в частности, некоторым элементом синхронизации может быть очередь сообщений. Например, мы можем записать в память данные и послать сообщение приемнику, что информация поступила в ресурс, после чего приемник, получив сообщение, начинает считывать данные. Также в качестве синхронизирующего средства могут применяться сигналы.

И это главное — не язык интерфейсов, а проблемы, которые могут возникнуть при взаимодействии параллельных процессов.

Лекция 18

Итак, мы к текущему моменту разобрали два механизма взаимодействия процессов в системе IPC: разделяемую память и механизм сообщений.

Давайте попробуем написать программу, в которой первый процесс будет принимать некую текстовую строку и в случае, если некоторая строка начинается с буквы “a”, то эта текстовая строка будет передана процессу A, если “b” — процессу B, если “q”, то сообщение будет передано процессам A и B, и будет осуществлен выход

Основной процесс:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
struct { long mtype;
char Data[256];
} Message;
int main(void)
{ key_t key;
int msgid;
char str[256];
```

key = ftok(“/usr/mash”,’S’); /* создаем ключ для работы с ресурсом — ключ уникальный и однозначно определяет доступ к разделяемому ресурсу одного типа, то есть с одним ключом могут быть связаны разделяемые ресурсы памяти, очередь сообщений и семафоров, но две области памяти связаны одним ключом быть не могут */

```
msgid = msgget(key, 0666 | IPC_CREAT); /* создаем очередь сообщений, 0666 — права доступа к очереди, разрешают всем читать и писать */
for (;;) {
gets(str); /* получаем строку */
strcpy(Message.Data, str); /* копируем ее в буфер сообщения */
switch(str[0]) {
case ‘a’:
```

case ‘A’: Message.mtype = 1; /* если строка начинается с “a”, то ставим тип сообщения равным единице, это означает, что приемником будет первый процесс */

```
msgsnd(msgid, (struct msgbuf *) &Message), 1+strlen(str),0); /* отправляем сообщение */
break;
case ‘b’:
case ‘B’: Message.mtype = 2; /* посылаем сообщение второму процессу */
msgsnd(msgid, (struct msgbuf *) &Message), 1+strlen(str),0);
break;
case ‘q’:
case ‘Q’: Message.mtype = 1;
msgsnd(msgid, (struct msgbuf *) &Message), 1+strlen(str),0);
Message.mtype = 2;
msgsnd(msgid, (struct msgbuf *) &Message), 1+strlen(str),0);
sleep(10); /* берем таймаут для гарантии, что все предыдущие сообщения дошли */
msgctl(msgid, IPC_RMID, NULL); /* убиваем разделяемый ресурс */
exit(0); /* завершаем процесс */ }
}
```

Давайте рассмотрим процесс-приемник. Рассмотрим только процесс A, так как B будет аналогичен, за исключением указания типа сообщений для приема.

```
proc_A:
```

```
...
int main(void)
{ key_t key;
int msgid;
key = ftok("/usr/mash", 'S'); /* получаем ключ к очереди по параметрам, аналогичным процессу-отправителю */
msgid = msgget(key, 0666 | IPC_CREAT); /* создаем или подключаемся к очереди сообщений */
for (;;) {msgrcv(msgid, (struct msgbuf *) (&Message), 256, 1, 0); /* принимаем сообщения с типом 1 */
if (Message.Data[0]=='q') || (Message.Data[0]=='Q') break; /* если сообщение начинается с "q" — заканчиваем выполнение процесса-получателя */
printf("...");
}
exit(0);
}
```

Семафоры.

С точки зрения тех проблем, с которыми мы знакомимся — семафоры — законное и существующее понятие. Впервые их ввел достаточно известный ученый Дейкстра. Суть этого объекта заключается в следующем. Семафор — это есть некоторый объект, который имеет целочисленное значениеs, и две операции — P(s) и V(s). P — уменьшает значение семафора на единичку и, если s>=0 после уменьшения процесс продолжает работать, если s<0, то процесс будет приостановлен и встанет в очередь ожидания, связанную с семафором s. Операция V увеличивает семафор на единицу. Если s>0 после увеличения, то процесс продолжает свое выполнение, если s<=0, то разблокируется один из процессов в очереди ожидания. Считается, что операции P и V неделимы, то есть их выполнение не может прерываться.

Также бывают двоичные семафоры, максимальное значение которого — 1. При значении 1 считается, что ни один из процессов не находится в критическом участке. При равенстве 0 — один процесс находится в критическом участке, другой работает нормально. Значение "-1" означает, чтоодин семафор находится в очереди ожидания, а другой — в критическом участке. Двоичные семафоры наиболее часто находили практическое применение в аппаратных реализациях.

Кроме тех вычислительных машин, которые являются однопроцессорными, бывают и многомашинные, многопроцессорные комплексы, для этих комплексов необходимо внесение в систему команд поддержки семафоров.

Это мы рассмотрели семафоры в общем случае. Сейчас же рассмотрим семафоры в системе IPC.

Существует разделяемый ресурс массив семафоров. Система позволяет процессам, работающим с этим ресурсом изменять элементы массива на произвольное число. Система позволяет ожидание процессом обнуления одного или нескольких семафоров. И, наконец, система позволяет уменьшать значение семафоров.

int semget(key_t key, int n, int flags)

Данная функция создает массив размерности n семафоров с заданным ключом и флагами. Функция возвращает идентификатор ресурса или -1, если произошла ошибка.

int semop(int semid, struct sembuf *SOPS, int n)

semid — идентификатор ресурса семафоров;

SOPS — указатель на структуру sembuf;

n — количество указателей на эту структуру, которые передаются функции semop, соответственно в структуре sembuf передается вся информация о необходимом действии;

struct sembuf

{short sem_num; /* номер семафора в массиве семафоров */
short sem_op; /* код операции над семафором */
short sem_flg; /* флаги */ }

Все это интерпретируется следующим образом. Пусть значение семафора с номером sem_num есть число sem_val. Если значение операции semop не равно нулю, то оценивается значение сумма sem_val+semop, если эта сумма больше или равна нулю, то значение данного семафора устанавливается новым, равным сумме предыдущего значения плюс код операции (semop), если эта сумма меньше нуля, то действие процесс будет приостановлено до наступления одного из следующих событий: до тех пор, пока значение этой суммы не станет >=0; придет сигнал (при приходе сигнала процесс снимется с ожидание) при этом semop будет равно "-1". Если же semop=0, то процесс будет ожидать обнуления значения семафора, при этом, если мы обратились к функции semop с нулевым кодом операции, а значение семафора уже было нуль, то ничего не произойдет. Если значение флага равно нулю, то флаги не используются. Флагов на самом деле нет, но, например, есть флаг IPC_NOWAIT, когда процесс ничего ждать не будет.

Заметим, что мы можем передать n структур и выполнить действия с n семафорами.

Управление массивом семафоров:
int semctl(int semid, int n, int cmd, union semun arg);
semid — идентификатор ресурса
n — номер семафора
cmd — команда (команды над семафорами, в том числе IPC_RMID)
arg — объединение, содержащее информацию о семафорах.

Лекция 19

Мы остановились на средствах синхронизации доступа к разделяемым ресурсам — на семафорах. Мы говорили о том, что семафоры — это тот формализм, который изначально был предложен ученым в области компьютерный наук Дейкстрой, поэтому часто в литературе их называют семафорами Дейкстры. Семафор — это есть некоторая переменная и над ней определены операции P и V. Одна позволяет увеличивать значение семафора, другая — уменьшать. Причем с этими изменениями связаны возможности блокировки процесса и разблокировки процесса. Обратим внимание, что речь идет о неразделяемых операциях, то есть тех операциях, которые не могут быть прерваны, если начались. То есть не может быть так, чтобы во время выполнения P или V пришло прерывание, и система передала управление другому процессу. Это принципиально. Поэтому семафоры можно реализовывать программно, но при этом мы должны понимать, что эта реализация не совсем корректна, так как

1)программа пишется человеком, а прерывается аппаратурой, отсюда возможно нарушение неразделяемости;

2)в развитых вычислительных системах, которые поддерживают многопроцессорную обработку или обработку разделяемых ресурсов в рамках одного процесса, предусмотрены семафорные команды, которые фактически реализовывают операции P и V. Это важно.

Мы говорили о реализации семафоров в Unix в системе IPC и о том, что эта система позволяет создать разделяемый ресурс "массив семафоров", соответственно, как и к любому разделяемому ресурсу, к этому массиву может быть обеспечен доступ со стороны различных процессов, обладающих нужными правами и ключом к данному ресурсу.

Каждый элемент массива — семафор. Для управления работой семафора есть функции:

semop, которая позволяет реализовывать операции P и V над одним или несколькими семафорами;

segctl — управление ресурсом. Под управлением здесь понимается три вещи:

- получение информации о состоянии семафора;
- возможность создания некоторого режима работы семафора, уничтожение семафора;
- изменение значения семафора (под изменением значения здесь понимается установление начальных значений, чтобы использовать в дальнейшем семафоры, как семафоры, а не ящички для передачи значений, другие изменения — только с помощью semop);

Давайте приведем пример, которым попытаемся проиллюстрировать использование семафоров на практике.

Наша программа будет оперировать с разделяемой памятью.

1 процесс — создает ресурсы “разделяемая память” и “семафоры”, далее он начинает принимать строки со стандартного ввода и записывает их в разделяемую память.

2 процесс — читает строки из разделяемой памяти.

Таким образом мы имеем критический участок в момент, когда один процесс еще не дописал строку, а другой ее уже читает. Поэтому следует установить некоторые синхронизации и задержки.

Следует отметить, что, как и все программы, которые мы приводим, эта программа не совершенна. Но не потому, что мы не можем ее написать (в крайнем случае можно попросить своих аспирантов или студентов), а потому, что совершенная программа будет занимать слишком много места, и мы сознательно делаем некоторые упрощения. Об этих упрощениях мы постараемся упоминать.

1й процесс:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int main(void)
{ key_t key;
int semid, shmid;
struct sembuf sops;
char *shmaddr;
char str[256];
key = ftok(“/usr/mash/exmpl”,’S’); /* создаем уникальный ключ */
semid = semget(key,1,0666 | IPC_CREAT); /* создаем один семафор с определенными правами доступа */
shmid = shmget(key,256, 0666 | IPC_CREAT); /*создаем разделяемую память на 256 элементов */
shmaddr = shmat(shmid, NULL, 0); /* подключаемся к разделу памяти, в shaddr — указатель на буфер с разделяемой памятью*/
semctl(semid,0,IPC_SET, (union semun) 0); /*инициализируем семафор со значением 0 */
sops.sem_num = 0; sops.sem_flg = 0;
/* запуск бесконечного цикла */
while(1) { printf(“Введите строку:”);
if ((str = gets(str)) == NULL) break;
sops.sem_op=0; /* ожидание обнуления */
semop(semid, &sops, 1); /* семафора */
strcpy(shmaddr, str); /* копируем строку в разд. память */
sops.sem_op=3; /* увеличение семафора на 3 */
semop(semid, &sops, 1);
```

```
}
shmaddr[0]='Q'; /* укажем 2ому процессу на то, */
sops.sem_op=3; /* что пора завершаться */
semop(semid, &sops, 1);
sops.sem_op = 0; /* ждем, пока обнулится семафор */
semop(semid, &sops, 1);
shmdt(shmaddr); /* отключаемся от разд. памяти */
semctl(semid, 0, IPC_RMID, (union semun) 0); /* убиваем семафор */
shmctl(shmid, IPC_RMID, NULL); /* уничтожаем разделяемую память */
exit(0);
}
```

2й процесс:

```
/* здесь нам надо корректно определить существование ресурса, если он есть — подключиться, если нет — сделать что-то еще, но как раз этого мы делать не будем*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int main(void)
{ key_t key; int semid;
struct sembuf sops;
char *shmaddr;
char st=0;
/* далее аналогично предыдущему процессу — инициализации ресурсов */
semid = semget(key,1,0666 | IPC_CREAT);
shmid = shmget(key,256, 0666 | IPC_CREAT);
shmaddr = shmat(shmid, NULL, 0);
sops.sem_num = 0; sops.sem_flg = 0;
/* запускаем цикл */
while(st!='Q') {
printf("Ждем открытия семафора \n");
/* ожидание положительного значения семафора */
sops.sem_op=-2;
semop(semid, &sops, 1);
/* будем ожидать, пока "значение семафора"+"значение sem_op" не перевалит за 0, то есть если придет "3", то "3-2=1" */
/* теперь значение семафора равно 1 */
st = shmaddr[0];
{/*критическая секция — работа с разделяемой памятью — в этот момент первый процесс к разделяемой памяти доступа не имеет*/}
/*после работы — закроем семафор*/
sem.sem_op=-1;
semop(semid, &sops, 1);
/* вернувшись в начало цикла мы опять будем ждать, пока значение семафора не станет больше нуля */
}
shmdt(shmaddr); /* освобождаем разделяемую память и выходим */
exit(0);
}
```

Это программа, состоящая из двух процессов, синхронно работающих с разделяемой памятью. Понятно, что при наличии интереса можно работать и с сообщениями.

На этом мы заканчиваем большую и достаточно важную тему организации взаимодействия процессов в системе.

Наша самоцель — не изучение тех средств, которые предоставляет Unix, а изучение принципов, которые предоставляет система для решения тех или иных задач, так как другие ОС предоставляют аналогичные или похожие средства управления процессами.

Системы программирования.

Мы с вами в начале курса говорили о системах программирования.

Определение: Комплекс программных средств, обеспечивающих поддержку технологий проектирования, кодирования, тестирования и отладки, называется системой программирования

Проектирование.

Было сказано, что на сегодняшний день достаточно сложно, а практически невозможно создавать программное обеспечение без этапа проектирования, такого же долгого, нудного и детального периода, который проходит во время проектирования любого технического объекта. Следует понять, что те программы, которые пишутся студентами в качестве практических и дипломных задач не являются по сути дела программами — это игрушки, так как их сложность невелика, объемы незначительны и такого рода программы можно писать слегка. Реальные же программы так не создаются, так же, как и не создаются сложные технические объекты. Никто никогда не может себе представить, чтобы какая-нибудь авиационная компания продекларировала создание нового самолета и дала команду своим заводам слепить лайнер с такими-то параметрами. Так не бывает. Каждый из элементов такого объекта, как самолет, проходит сложный этап проектирования.

Например, фирма Боинг подняла в воздух самолет “Боинг-777”, замечательность этого факта заключается в том, что самолет взлетел без предварительной продувки в аэродинамической трубе. Это означает, что весь самолет был спроектирован и промоделирован на программных моделях, и это проектирование и моделирование было настолько четким и правильным, что позволило сразу же поднять самолет в воздух. Для справки — продувка самолета в аэродинамической трубе стоит сумасшедшие деньги.

Примерно та же ситуация происходит при создании сложных современных программных систем. В начале 80х гг была начата СОИ (стратегическая оборонная инициатива), ее идея была в том, чтобы создать сложной технической системы, которая бы в автоматическом режиме установила контроль за пусковыми установками СССР и стран Варшавского блока, и в случае фиксации старта с наших позиций какого-то непродекларированного объекта автоматически начиналась война. То есть запускалисьбы средства уничтожения данного объекта и средства для ответных действий. Реально тот департамент вооруженных сил, который занимался этим проектом, испытал ряд кризисов в связи с тем, что ведущие специалисты в области программного обеспечения отказывалисьучаствовать в реализации этого проекта из-за невозможности корректно его спроектировать, потому что система обладала гигантским потоком входных данных, на основе которых должны были быть приняты однозначные решения, ответственность за которые оценить весьма сложно. На самом деле эта проблема подтолкнула к развитию с одной стороны — языков программирования, которые обладали надежностью, в частности, язык Ада, одной из целью которого было создание безошибочного ПО. В таких языках накладывались ограничения наместа, где наиболее вероятно возникновение ошибки (межмодульные интерфейсы; выражения, где присутствуют разные типы данных

и т.п.) Заметим, что язык С не удовлетворяет требованиям безопасности. С другой стороны — к детальному проектированию, которое бы позволяло некоторым формальным образом описывать создаваемый проект и работать с проектом в части его детализации. Причем, переход от детализации к кодированию не имел бы четкой границы. Понятно, что это есть некоторая задача не сегодняшнего, а завтрашнего дня, но реально разработчики программ находятся на пути создания таких средств, которые позволили бы совместить проектирование и кодирование. Сегодняшние системы программирования, которые строятся на объектно ориентированном подходе, частично решают эту проблему.

Следующая проблема проектирования. Мы продкларировали модули, объявили их взаимосвязи, как-то описали семантику модулей (это тоже проблема). Но никто не даст гарантии, что этот проект правилен. Для решения этой проблемы используется моделирование программных систем. То есть, когда вместе с построением проекта, который декларирует все интерфейсы, функциональность и прочее, мы можем каким-то образом промоделировать работу всей или частей создаваемой системы. Реально при создании больших программных систем на сегодняшний день нет единых инструментариев для таких действий. Каждые из существующих систем имеют разные подходы. Иногда эти подходы (как и у нас, так и за рубежом) достаточно архаичны.

Но тем не менее следует понимать, что период проектирования есть очень важный момент.

Кодирование.

Если составлен нормальный проект, то с кодированием проблем нет. Но следует обратить внимание на то, что специалист в программировании это не тот, кто быстро пишет на С, а тот, кто хорошо и подробно сможет спроектировать задачу. При современном развитии инструментальных средств закодировать сможет любой школьник, а спроектировать систему — это и есть профессиональная задача людей, занимающихся программированием — выбрать инструментальные средства, составить проект, промоделировать решение.

Основной компонент системы кодирования — язык программирования. В голове каждого программиста лежит иерархия языков программирования — от машинного кода и ассемблера до универсальных языков программирования (FORTRAN, Algol, Pascal, C и т.д.), специализированных языков (SQL, HTML, Java и т.д.)

Мы имеем ЯП и программу, которая написана в нотации этого языка. Система программирования обеспечивает перевод исходной программы в объектный язык. Этот процесс перевода называется трансляцией. Объектный язык может быть как некоторым языком программирования высокого уровня (трансляция), так и машинный язык (компиляция). Мы можем говорить о трансляторах-компиляторах и трансляторах-интерпретаторах.

Компилятор — это транслятор, переводящий текст программы в машинный код.

Интерпретатор — это транслятор, который обычно совмещает процесс перевода и выполнения программы (компилятор сначала переводит программу, а только затем ее можно выполнить). Он, грубо говоря, выполняет каждую строчку, при этом машинный код не генерируется, а происходит обращение к некоторой стандартной библиотеке программ интерпретатора. Если результат работы компилятора — код программы на машинном языке, то результат работы транслятора — последовательность обращений к функциям интерпретации. При этом, также как и при компиляции, когда создается оттранслированная программа, у нас тоже может быть создана программа, но в этом интепретируемом коде (последовательности обращений к функциям интерпретации).

Понятна разница — компиляторы более эффективны, так как в интерпретаторах невозможна оптимизация и постоянные вызовы функций также не эффективны. Но интерпретаторы более удобны за счет того, что при интерпретации возможно включать в функции интерпретации множество сервисных средств: отладки, возможность интеграции интерпретатора и языкового редактора (компиляция это делать не позволяет).

На сегодняшний день каждый из методов — и компиляция и интерпретация занимают свои определенные ниши.

Лекция 20

На прошлой лекции мы начали рассматривать системы программирования. На самом деле эта тема может быть основанием целого курса, потому что эта тема включает в себя все то, что может быть в современной науке о компьютерах — это и хорошие практические решения, и разработанные, реально применяемые, теоретические решения, и многое другое. Мы говорили, что система программирования — это комплекс программ, обеспечивающий жизненный цикл программы в системе. Жизненный цикл создаваемого программного обеспечения содержит следующие этапы:

–проектирование

–кодирование

–тестирование

–отладка

Мы с вами говорили о важности этапа проектирования, о том что программный продукт представляет из себя сложнейший объект, имеющий огромное число связей между своими компонентами. Пропустить этап проектирования нельзя. Для проектирования программных систем, необходимы специализированные средства, которые позволили бы (в идеале) описывать проект некоторым формальным образом, и последовательно уточняя его, приводить формальное описание проекта в реальный код программы.

Мы говорили, что важным этапом проектирования, является этап моделирования системы, т.е. тот этап, когда мы берем внешние нагрузки, которые могут поступать на систему, приписываем свойству потока событий, связанных с этими внешними нагрузками, определенный статистический закон, и рассматриваем предположительное поведение системы при такой эмуляции внешней среды. Понятно, что без этапа моделирования тоже трудно создать какой-либо программный продукт.

Этап кодирования

Мы также говорили, что важной частью системы программирования являются средства кодирования. Этап кодирования в жизненном цикле программы традиционно (и обычно не правильно) однозначно связывается с понятием системы программирования. Очень многие, когда начинают говорить о системе программирования, подразумевают под этим транслятор языка программирования. Хотелось бы этими лекциями вам показать, что система программирования — это нечто существенно более широкое, чем транслятор. Все компоненты одинаково необходимы.

Транслятор — это программа, которая переводит программу в нотации одного языка, в нотацию другого языка. Компилятор — это транслятор, который переводит программу из нотации одного языка в нотацию машинного языка. Машинным языком может быть либо код конкретной машины, либо объектный код. Трансляторы могут быть интерпретаторами, т.е. совмещать анализ исходной программы с ее выполнением. Результатом работы интерпретатора является не машинный код, а последовательность обращений к библиотеке функций интерпретатора. Интерпретатор, в отличие от транслятора, может выбирать одну за одной инструкции и сразу их выполнять. При интерпретации (в отличие от трансляции или компиляции), может быть начато выполнение программы которая имеет синтаксические ошибки.

Кросс-трансляторы.

Если рассматривать системы трансляции, то есть еще один вид трансляторов — кросс-трансляторы (и кросс-компиляторы). Кросс транслятор работает на некотором типе вычислительной системы, которая называется инструментальная ЭВМ. Инструментальная

ЭВМ может характеризоваться своей архитектурой и/или операционным окружением, которое функционирует на ней. Кросс-транслятор обеспечивает перевод программы, записанной в нотации некоторого языка, в код вычислительной системы, отличной от инструментальной ЭВМ. Та вычислительная система,для которой генерируется код, называется объектной ЭВМ, и соответственно, тот код, который мы получаем, называется объектным кодом (это не тоже, что объектный модуль). Например, компьютеру, который управляет двигательной установкой самолета, совершенно ненужно иметь операционную среду, которая обеспечит работу пользователя по разработке программ для него. Ему совершенно не нужно иметь средства редактирования текста, трансляции и т.д., потому что у него одна функция — управлять двигательной установкой. На этом компьютере будет работать операционная система реального времени. Для создания программ для такого рода компьютеров и используются системы кросс-программирования и кросс-трансляторы. На обычной машине типа PC может быть размещен транслятор, который будет генерировать код для заданного компьютера.

На самом деле, бывают ситуации, когда тип объектной машины совпадает с типом инструментальной машины, но отличаются операционные среды, которые функционируют на данных машинах. В этом случае также нужна система кросс-программирования.

Кросс-трансляторы также нужны разработчикам новых машин, которые хотят параллельно с ее появлением создать программное обеспечение, которое будет работать на этой машине.

Обработка модульной программы.

С точки зрения этапа кодирования можно рассмотреть последовательность обработки программы более крупноблочно, чем с точки зрения трансляции. Мы с вами говорили на начальных лекциях о том что современные языки программирования поддерживают модульность (программа представляется в виде группы модулей и взаимосвязь между этими модулями осуществляется за счет соответствующих объявлений в них). Давайте посмотрим, что происходит на этапе обработки программы, написанной на одном из модульных языков.

Пусть есть некоторая группа модулей и есть соответствующие этим модулям тексты программ, на языках, используемых для программирования. Языковыми средствами определены связи между модулями.

Первый этап, который происходит — это этап трансляции (либо компиляции) каждого из модулей. После трансляции модуля в виде исходного текста мы получаем объектный модуль — это есть машинно-ориентированное представление программы, в котором присутствуют фрагменты программы в машинном коде, а также информация о необходимых внешних связях (ссылки на объекты в других модулях). Информация о необходимых внешних связях (помимо информации о местонахождении внешних объектов) также включает в себя ссылки на те места машинного кода, которые пытаются использовать адреса внешних объектов, т.е. на те недообработанные команды, которые нельзя обработать из-за того, что при трансляции модуля еще не известно где какие объекты находятся. Т.е. объектный модуль — это машинное представление программного кода, в котором еще не разрешены внешние связи. Объектный модуль может содержать дополнительную информацию (например, информацию, необходимую для отладки — таблицы имен и т.д.).

Для каждого из исходных модулей мы получим объектный модуль. После этого все объектные модули, которые составляют нашу программу, а также модули требуемых библиотек функций, поступают на вход редактору внешних связей. Редактор внешних связей моделирует размещение объектных модулей в оперативной памяти и разрешает все связи между ними. В итоге мы получаем исполняемыймодуль, который может быть

запущен как процесс. Иногда трансляторы в качестве результата трансляции выдают модуль на ассемблере соответствующей машины.

В эту же схему также часто добавляется этап оптимизации программы, причем оптимизация может происходить до этапа трансляции (т.е. в терминах исходного языка) или/и после трансляции (в терминах машинного кода). Например до трансляции можно вычислить все константные подвыражения и т.д. Для машин типа PC этап оптимизации может быть не столь важен, потому что этот вопрос обычно разрешается покупкой какого-нибудь более быстрого компонента, но есть класс машин (mainframe), для которых этот этап необходим.

Давайте посмотрим на проблему кодирования с другой стороны. Мы посмотрим как устроен этап трансляции.

Каждый транслятор при обработке программы выполняет следующие действия.

1)Лексический анализ.

2)Синтаксический анализ.

3)Семантический анализ и генерация кода.

Лексический анализ.

Лексический анализатор производит анализ исходного текста на предмет правильности записи лексических единиц входного языка. Затем он переводит программу из нотации исходного текста в нотацию лексем.

Лексические единицы — это минимальные конструкции, которые могут быть продекларированы языком. К лексическим единицам относятся:

–идентификаторы

–ключевые слова

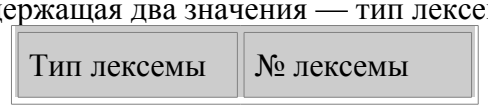
–код операции

–разделители

–константы

Вещественные константы в некоторых трансляторах могут представляться в виде группы лексических единиц, каждая из которых является целочисленной константой.

После этого исходная программа переводится в вид лексем. Лексема — это некоторая конструкция, содержащая два значения — тип лексемы и номер лексемы.



Тип лексемы — это код, который говорит о том, что данная лексема принадлежит одной из обозначенных нами групп. к примеру лексема может быть ключевым словом, тогда в поле типа будет стоять соответствующий номер. Номер лексемы уточняет конкретное значение этой лексемы. Если, к примеру, было ключевое слово begin, то номер лексемы будет содержать число, соответствующее ключевому слову begin. Если тип лексемы — идентификатор, то номер лексемы будет номером идентификатора в таблице имен которую создаст лексический анализатор. Если тип лексемы — константа, то номер лексемы тоже будет ссылкой на таблицу с константами.

После лексического анализатора мы получаем компактную программу, в которой нет уже ничего лишнего (пробелов, комментариев, и т.д.). Вся программа составлена в виде таких лексем, и поэтому она более компактна и проста.

Синтаксический анализ.

Программа в виде лексем поступает на вход синтаксическому анализатору, который осуществляет проверку программы на предмет правильности с точки зрения синтаксических правил. Результатом работы синтаксического анализатора является либо информация о том, что в программе имеются синтаксические ошибки и указание координат этих ошибок и их диагностика, либо представление программы в некотором промежуточном виде. Этим промежуточным видом может быть, предположим,

бесскобочная запись, либо запись в виде деревьев (хотя одно однозначно сводится кдругому). Это промежуточное представление, которое является синтаксически и лексически правильной программой, поступает на вход семантическому анализатору.

Семантический анализ.

Семантика — это все то, что не описывается синтаксисом и лексикой языка. К примеру, лексикой и синтаксисом языка сложно описать то, что нехорошо передавать управление в тело цикла не через начало цикла. Выявление таких ошибок — одна из функций семантического анализа. при этом семантический анализатор ставит в соответствие синтаксически и семантически правильным конструкциям объектный код, т.е. происходит генерация кода.

Система программирования и трансляции — очень наукоемкая область программного обеспечения. Организация трансляторов — это было первое применение теоретических достижений науки, которые заключались в следующем. За счет возможности использования тех или иных грамматик (наборов формальных правил построения лексических конструкций и синтаксических правил), можно разделить программную реализацию лексических и синтаксических анализаторов на два компонента. Первый компонент — это программа, которая в общем случае ничего не знает о том языке, который она будет анализировать. Второй компонент — это набор данных, представляющий из себя формальное описание свойств языка, который мы анализируем. Совмещение этих двух компонентов, позволяет автоматизировать процесс построения лексических и синтаксических анализаторов, а также генераторов кода, для различных языков программирования. Современные системы программирования в своем составе имеют средства автоматизации построения компиляторов. Для ОС UNIX есть пакет LEX — пакет генерации лексических анализаторов, и есть пакет YACC — для генерации синтаксических анализаторов. Это все достигается за счет возможности формализации свойств языка, и использования этого формального описания, как параметров для тех или иных инструментальных средств.

Проходы трансляторов.

Мы с вами посмотрели на транслятор с точки зрения функциональных этапов. Но очень часто мы слышим об однопроходных трансляторах, двухпроходных, трехпроходных, и т.д. С проблемой трансляции связано понятие "проход". Проход — это полный просмотр некоторого представления исходного текста программы.

Есть трансляторы однопроходные. Это означает, что транслятор просматривает исходный текст от начала и до конца, и к концу просмотра (в случае правильности программы) он получает объектный модуль.

Если мы посмотрим Си-компилятор, с которым вы работаете, то скорее всего он двухпроходный. Первый проход — это работа препроцессора. После первого прохода появляется чистая Си-программа без всяких препроцессорных команд. На Втором проходе происходит лексический, синтаксический и семантический анализ, и в итоге вы получаете объектную программу в виде ассемблера.

Количество проходов в некоторых трансляторах связано с количеством этапов, т.е. бывают реализации, для которых удобно сделать отдельный проход для лексического анализа, отдельный проход для синтаксического анализа и отдельный проход для семантического анализа. Если транслятор многопроходный,то возникает проблема сохранения промежуточной нотации программы между проходами.

Make-файл. К этой же проблеме кодирования относится средство поддержки разработки программных проектов. Одним из популярных средств, ориентированных на работу одного или нескольких программистов, является т.н. make-средство. Название происходит от соответствующей команды ОС UNIX. С make-командой связан т.н. make-файл, в котором построчно указываются взаимосвязи всевозможных файлов, получаемых

при трансляции, редактировании связей, и т.д., и те действия, которые надо выполнить, если эти взаимосвязи нарушаются. В частности можно сказать, что некоторый исполняемый файл зависит от группы объектных файлов, и если эта связь нарушена, то надо выполнить команду редактирования связей (link ...). Что значит нарушение зависимости и что значит связь? Make-команда проверяет существование этих объектных файлов. Если они существуют, то времена их создания должны быть более ранние, чем время создания исполняемого файла. В том случае, если это правило будет нарушено (а это проверяет make-команда), то будет запущен редактор связей (link), который заново создаст исполняемый файл. Тем самым такое средство позволяет нам работать с программой, состоящей из большого количества модулей, и не заботитьсяо том, соответствует ли в данный момент времени исполняемый файл набору объектных файлов или не соответствует (можно просто запустить make-файл).

Make-файлы могут содержать большое количество такого рода строчек, которые таким образом свяжут не только объектные и исполняемые файлы, но и каждый из исходных файлов с соответствующим объектным файлом, и т.д. Т.е. суть такова, что после работы не надо каждый раз для каждого файла запускать компилятор, редактор связей, а можно просто запустить make-файл, а он уже сам определит и выберет те файлы, которые нужно корректировать, и выполнит необходимые действия. На самом деле такими средствами сейчас обладают почти все системы программирования.

Система контроля версий. Если make-файл — это система, предназначенная для одного программиста, в лучшем случае, для нескольких программистов, то если у нас существует большой коллектив, который делает большой программный проект, то используется т.н. система контроля версий, которая позволяет организовывать корректную работу больших коллективов людей над одним и тем же проектом, которая основана на возможности декларации версий и осуществлении контроля за этими версиями.

Этапы тестирования и отладки

Тестирование — это поиск ситуации, в которой программный продукт не работает. При этом используются наборы тестов, определяющих внешнюю нагрузку на программный продукт. Можно сказать, что программа оттестирована на определенном наборе тестов. Утверждение, что программа оттестирована вообще в общем случае некорректно.

Отладка — это процесс поиска, локализации и исправления ошибки. Отладка осуществляется, когда мы имеем программную систему, и знаем, что она не работает на каком-то из тестов.

Проблемы тестирования и отладки — это есть проблемы крайней важности. По оценкам, на тестирование и отладку затрачивается порядка 30% времени разработки проекта. Сложность тестирования и отладки зависит от качества проектирования и кодирования. Тестирования зачастую выполнить сложно и часто для тестирования используются модельные нагрузки, напримермы тестируем бортовую сеть самолета, что-то мы сможем сделать на земле, а что-то так или иначе делается уже на реальном полете, когда собираются данные и фиксируется работает система или не работает.

Лекция 21

Командный язык ОС Unix cshell (CSH)

Для многих пользователей основным свойством, на которое они обращают внимание являются не тонкости создания ОС, а тот интерфейс, который предоставляет ОС при взаимодействии с ней. В этом плане практически каждая система имеет систему интерактивного взаимодействия с пользователем. Это означает, что имеются средства, позволяющие пользователю вводить запросы на выполнение действий, которые впоследствии система сможет интерпретировать и исполнять. ОС Unix поддерживает

возможность работы с произвольным количеством интерпретаторов команд или командных языков. В частности, при регистрации пользователя (информация о пользователе размещается в файле /etc/passwd) среди прочих параметров пользователя, есть строка, отвечающая за то, какой интерпретатор команд будет запущен при входе пользователя в систему (вообще говоря, может быть запущена произвольная программа).

Традиционными интерпретаторами команд Unix являются: sh, csh, bash и некоторые другие. В принципе, все эти интерпретаторы похожи друг на друга и являются развитием sh.

Рассмотрим csh.

Интерпретатор распознает структуру вводимой команды. Предполагается, что команда, это последовательность слов, заканчивающихся некоторым символом. Слово — последовательность символов, не содержащая разделителей. Разделители — это набор фиксированных символов, позволяющих разделять слова в командной строке (например, пробел, запятая, точка с запятой и т.д.), при этом разделители имеют в csh свою интерпретацию. В частности разделитель "||" — создание конвейера между командами (стандартный ввод одной команды будет стандартным выводом другой, например, "ls|| more").

Unix поддерживает в своих интерпретаторах так называемые метасимволы. Они обычно встречаются в словах команд и интерпретируются по специальным правилам. Рассмотрим некоторые из них:

"*": означает любую последовательность символов. Например, команда "rm *" удалит все файлы, которые не начинаются с символа "точка" (чтобы их удалить следует набрать "rm *.*");

"?": означает, что на месте этого символа может быть любой символ. Например "rm ?" удалит все файлы, имена которых состоят из одного символа;

"квадратные скобки": внутри скобок указываются альтернативные группы. Например, "[abc]" означает, что вместо квадратных скобок может быть подставлен один символ из набора "abc". Можно указать диапазон, например "[0-9]";

Командный интерпретатор csh позволяет объединять команды. Для этого также используются метасимволы:

"(...)" — Если внутри скобок перечислены команды, например "(cd /etc; ls -la || grep pas)", то фактически запустится новый интерпретатор команд, который выполнит последовательность команд, находящуюся в круглых скобках. Заметим отличие такого запуска от обычного — если мы выполним команду "cd /etc" обычным способом, то наш интерпретатор сменит каталог, тогда как при запуске еще одного интерпретатора каталог у основного интерпретатора не изменится.

"{...}" — все команды, перечисленные внутри фигурных скобок будут запущены последовательно слева направо, но при этом на стандартный вывод будет положена объединенная последовательность стандартных выводов всех команд. Например, "{more t.b; more t.c} > tt.b". В результате в файле tt.b окажутся стандартные выводы обеих команд "more", если бы фигурных скобок не было, то указатель перенаправления ввода вывода относился бы лишь к последней, в нашем случае второй, команде.

Интерпретатор команд имеет набор встроенных команд. Все команды, которые мы используем делятся на два типа: первый — это те, которые находятся в отдельных файлах, их можно добавлять, удалять, модифицировать; второй — которые "зашиты" внутри интерпретатора. В частности, команда "kill" — команда интерпретатора, то есть передача сигнала осуществляется от имени интерпретатора. Есть, например, встроенная команда "alias" — она используется для переименования существующих команд.

Интерпретатор команд csh позволяет работать с предысторией, то есть он организовывает буферизацию последних n команд и допускает доступ к списку этих команд — их можно запускать еще раз, редактировать и снова выполнять. Соответственно, интерпретатор команд csh имеет возможность именовать строки из

списка предыстории, в частности, для ссылки на командную строку из списка предыстории можно пользоваться следующими конструкциями: “!<...>”, например, “!!” означает повторить последнюю команду.Это также может быть номер: “!10” — повторить командную строку с номером 10. Если номер положительный — то это абсолютный номер команды, если отрицательный — относительный от текущего номера (например, “!-1” указывает выполнить предыдущую команду). Также в качестве параметра могут быть некоторые контекстные ссылки.

Интерпретатор команд предоставляет возможность программирования на уровне csh. Для этого в csh предусмотрена декларация переменных, присвоения им значения, а также набор высокоуровневых операторов, которые по своей семантике похожи на операторы языка C (поэтому он и называется cshell). Оперируя с переменными, csh можно создавать программы и выполнять многие другие действия. Кроме всего, имеются предопределенные имена переменных cshell, которые отвечают за настройку системы (например, количество строк в предыстории, ее сохранение — переменные HISTORY, SAVEHISTORY). Кроме таких переменных, через которые осуществляется настройка и которые мы можем изменять, есть еще один класс переменных cshell — это внутренние переменные, такие переменные, которые имеют предопределенные имена и определяют свое значение через внутренние функции интерпретатора команд.

В частности, есть переменная path. Ее суть в том, что в ней хранится (она является текстовым массивом) текстовые строки, содержащие полные имена некоторых каталогов; в соответствии с содержимым path cshell осуществляет поиск файлов, которые являются командами, введенными пользователями. Мы говорили о том, что кроме команд, встроенных в интерпретатор, вUnix больше специальных команд нет, и любой исполняемый файл может являться командой. Но тогда возникает вопрос — мы ввели некоторое имя name — где будет осуществляться поиск файла name? В текущем каталоге? Но это неправильно, так как, например, команда “ls” лежит не в текущем каталоге. Но с другой стороны хотелось бы иметь возможность запускать файлы и из него тоже. Содержимое переменной path определяет порядок каталогов, в которых будет осуществлен поиск команды, если ее нет в текущем каталоге (например,если path имеет значение “.;/bin;/usr/bin”, то поиск будет начат в текущем каталоге, затем в “/bin” и затем в “/usr/bin”, это означает, что если мы сделаем команду “ls” и поместим ее в текущий каталог, то она перекроет “ls”, лежащий в “/bin”).

Другие переменные:

home — имя домашнего каталога.

ignoreeof — установка этой переменной блокирует завершение сеанса по вводу Ctrl-D.

prompt — в системе можно варьировать приглашение (по умолчанию — “$”), оно может быть достаточно интеллектуальным (включить дату, путь и прочее).

Мы с вами рассмотрели path. Представьте себе, что мы вводим некоторую строку name, но такого файла нет. Тогда у нас будет осуществлен поиск по всем каталогам, перечисленным в path со всеми вытекающими последствиями (открытием каталогов, чтением, поиском и т.п.), это долго, а если учесть, что среда у нас многопользовательская, то если для каждого пользователя система будет так загружаться, то расходы получаются сумасшедшая. Но Unix — достаточно разумная ОС. При входе пользователя в систему на основании значения переменной path формируется некая hash-таблица имен исполняемых файлов, находящихся во всех перечисленных каталогах, эта таблица учитывает, естественно, порядок директорий и прочие атрибуты. И в этом случае поиск команды будет осуществляться следующим образом: сначала будет просмотрен текущий каталог, если там ничего найдено не будет, то последует обращение к быстрой hash-таблице. Эта скорость оплачена тем, что при входе в систему часть времени загрузки тратится на составление таблицы. Однако, здесь есть проблема — если в каталог из path была добавлена новая команда, то в таблицу она не попадет, так как появилась после

формирования таблицы. В этом случае можно поступить двояко — указать полное имя команды (не использовать hash-таблицу) или переформировать hash-таблицу (для этого существует команда rehash), и, соответственно, произойдет та же процедура, что происходит при входе в систему.

Кроме того, cshell имеет еще один тип переменных, которые называются переменными окружения. Вспомним, что при запуске процесса функции main передается несколько параметров: один из них — массив аргументов, второй — массив, содержащий переменные и значения окружения. В процесс можно передать те параметры, которые характеризуют сеанс работы. В частности, можно передать имя домашнего каталога, имя текущего каталога, имя терминала, которым вызван данный процесс и т.д.

Все переменные окружения можно модифицировать средствами cshell'а.

Работа с файлами.

С помощью средств Cshell можно составлять программы, в них могут фигурировать значения переменных, которые можно интерпретировать, как имена файлов. Средствами cshell можно определять ряд свойств файла, в частности, можно проверить существование, является ли файл каталогом, прав доступа, размера, можно запустить файл. Это есть полезное средство, с помощью которого реализовано много команд, которые являются командами, написанными на cshell.

Команды cshell можно вводить построчно, а можно cshell программу записать как файл и исполнять его.

Специальные файлы cshell.

Cshell имеет две разновидности этих файлов, это файлы, которые могут выполняться при старте cshell, и которые могут выполняться при завершении сеанса.

При старте cshell работает с файлами:

.cshrc — Это командный файл, в котором пользователь по своему усмотрению может размещать произвольное число команд на cshell, то есть там могут быть перечислены команды, которые пользователь хочет выполнить при старте системы (проверка файловой системы, например)

.login — Этот файл запускается после ".cshrc", он запускается при входе в систему (".cshrc" запускается при запуске cshell), то есть при запуске еще одного csh для нового сеанса будет запущен ".cshrc", но ".login" запущен уже не будет. С учетом этого следует его использовать (указывать, например, переобозначения команд).

При завершении работы cshell выполняется

.logout — Здесь можно выполнить действия, необходимые для завершения работы с сеансом (например, установить команду уничтожения промежуточных файлов).

Имеется стандартный файл, который образуется в ходе работы системы — это файл ".history", если определена возможность savehistory, то история пишется в этот файл.

Вот, наверное, и все, что стоит сказать о cshell в общих словах, остальное можно посмотреть в литературе, специально описывающих данный командный язык.

Лекция 22

Многомашинные ассоциации.

Первые многомашинные ассоциации появились в начале 60х годов, это было связано с двумя проблемами:

–проблема массового доступа к некоторым вычислительным ресурсам;

–появление задач, требовавших для их решения привлечения более, чем одной машины;

К первым многомашинным комплексам можно отнести терминальные комплексы.

Терминальные комплексы

Терминальный комплекс — это набор программных и аппаратных средств, предназначенных для обеспечения взаимодействия пользователей с вычислительной установкой через телефонную или телеграфную сеть. Надо отметить, что в качестве среды взаимодействия может быть не только телефон или телеграф, но и любая среда, обеспечивающая передачу данных. Структуру терминального комплекса можно изобразить следующим образом:

К некоторому каналу передачи данных вычислительной системы подключен мультиплексор. Это устройство, назначением которого является взаимодействие внешних устройств (У) с вычислительной системой через один канал ввода-вывода. То есть любое сообщение, поданное на один из входов мультиплексора, как-то преобразовывается и попадает в вычислительную систему через один единственный канал.

К мультиплексору могут быть подключены локальные терминалы (ЛТ) (это первый уровень средств доступа к системе). Также может быть подключен модем (М) (Модем — устройство, преобразующее цифровой сигнал в аналоговый), который позволяет выйти в телефонную или телеграфную сеть и передавать через нее данные. Единственное условие — принимающим устройством с другого конца может быть только модем. К модему на другом конце может быть подключен как терминал, так и еще один мультиплексор, к которому опять же могут быть подключены какие-то устройства.

Линия связи, которая связывает один удаленный терминал с вычислительной системой называется “точка-точка”. Такая связь может быть либо арендуемой (когда линию предоставляет телефонная станция и коммутация фиксирована), либо коммутируемой, когда телефонная станция устанавливает путь связи в момент соединения.

Линия связи может быть многоточечной, когда на входе находится не удаленный терминал, а удаленный мультиплексор. Многоточечные каналы тоже могут быть как коммутируемые, так и арендуемые.

Виды каналов связи.

Симплексный канал — канал, по которому передача данных ведется в одном направлении. Например, от терминала к системе.

Дуплексный канал — передача данный осуществляется в обоих направлениях. В пример можно привести телефонную связь, когда оба собеседника могут говорить и слышать друг друга одновременно.

Полудуплексный канал — осуществляет передачу данных в двух направлениях, но в каждый момент времени передача может происходить только в одну сторону. Здесь примером может послужить разговор по рации.

В качестве локальных терминалов в системе могут присутствовать и компьютеры, эмулирующие работу терминала.

Многомашинные вычислительные комплексы (МВК)

МВК — аппаратное-программное объединение группы вычислительных машин, на каждой из которых работает своя ОС (это требование является основным, отличающим МВК от многопроцессорного вычислительного комплекса), и имеются общие физические ресурсы. Последнее означает, что имеется группа вычислительных машин, которой доступны некоторые общие ресурсы, например, ОЗУ, жесткий диск, принтер и т.д.

Соответственно, в МВК возникают все те проблемы, которые возникают при использовании различными процессами общих ресурсов.

МВК использовались в качестве систем сбора и обработки больших потоков данных или организации на их базе больших терминальных комплексов, которые обеспечивали гипермассовый доступ к данной вычислительной системе.

МВК появились в начале 60х годов и на сегодняшний день также успешно используются. Одним из применений МВК является дублирование вычислительной мощности.

Вычислительные сети

Вычислительные сети — это следующее поколение многомашинных ассоциаций после терминальных комплексов и МВК.

Пусть у нас имеется некоторое образование, которое называется коммутационной средой. Коммутационная среда включает в себя каналы передачи данных и коммутационные машины.

Абонентские машины (АМ) могут взаимодействовать друг с другом с использованием коммутационной среды через каналы связи и коммутационные машины.

Существует ряд классических разновидностей сетей:

1)Сеть коммутации каналов.

Ее суть состоит в том, что если надо связать АМ2 и АМ3, то происходит физическое соединение каналов и комм. машин. Это соединение будет существовать до конца взаимодействия АМ2 и АМ3.

Достоинства: если мы канал коммутировали, то после этого взаимодействие идет без проблем со скоростью, равной минимальной скорости каналов, составляющих магистраль передачи данных.

Недостатки: такая организация может блокировать другие магистрали (если мы связали машины 2 и 3, то 5 и 1 машины связаться не смогут, так как каналы и комм. машины уже заняты). В качестве решения этой проблемы можно требовать от коммутационной среды большой избыточности (увеличить число коммутационных машин и каналов).

2)Сеть коммутации сообщений

Если коммутация каналов — коммутация на время всего сеанса связи, то коммутация сообщений — это связь, при которой весь сеанс уходит на передачу сообщений (логически завершенных порций данных). При этом сеть коммутаций сообщений работает также, как и сеть коммутации каналов, но коммутация происходит не на период всего сеанса связи, а на период передачи сообщения.

Недостатки: здесь возникает проблема нагрузки на коммуникационные машины, так как они должны обладать возможностью буферизации сообщений в связи с возможно неравномерным количеством и объемом сообщений от разных машин; а также во время передачи сообщения блокируется канал, по которому происходит общение, если сообщение слишком большое, то это приводит к ощутимым задержкам в передаче остальных машин.

Достоинства: простота, как на уровне программной, так и аппаратной реализации.

3)Сеть коммутации пакетов

Сеанс разбивается на сообщения, сообщение разбивается на порции данных одинакового объема, и по сети передаются как раз эти порции данных, которые называются пакетами. При передаче пакетов действует принцип “горячей картошки” — когда каждая из коммутационных машин стремиться, как можно скорее, избавиться от пришедшего к ней пакета.

Достоинства: так как все пакеты одинакового размера, то не возникает проблем с буферизацией; логически происходит достаточно быстрое соединение, ибо практически нет ситуаций, когда какие-то каналы заблокированы; из-за разбиения на пакеты мы можем качественно лучше исправлять ошибки — для исправления ошибки в потоке данных следует лишь передатьповторно тот пакет, в котором она произошла.

В реальных системах используются многоуровневые сети, одни их части работают в режиме коммутации каналов, другие в режиме коммутации сообщений, третьи — в

пакетном режиме. “Чистых” сетей, в принципе, не бывает, так как на каждом логическом уровне сети может проявиться тот или иной режим.

Вот некоторая предыстория развития многомашинных ассоциаций (МА).

Стандартизация взаимодействия в сетях. ISO/OSI

Развитие МА вообще и сетей ЭВМ в частности определило тот факт, что возникла необходимость в стандартизации взаимодействия, происходящего в сети. Поэтому в конце 70хх, начале 80хх годов, международная организация стандартизации предложила стандарт взаимодействия открытых систем ISO/OSI.

Суть ISO/OSI в следующем:

Была предложена семиуровневая модель организации взаимодействия компьютеров со средой передачей данных.

Уровни:

1)Физический (уровень сопряжения с физическим каналом). На этом уровне решаются самые “земные” задачи: уровни сигналов, их типы и т.д. Этот уровень, который выходит на конкретную физическую среду. Средой может быть “витая пара”, оптоволокно, коаксиальный кабель и т.д., каждая такая среда определяет свои правила общения с ней.

2)Канальный. Это уровень, на котором формализуются правила передачи данных через каналы. То есть, если физический уровень — уровень управления средой (кабелем, радиоканалом), то канальный уровень связан уже с передачей данных по этому каналу.

3)Сетевой. Он управляет сетью, связью в сети между машинами, здесь решается проблема адресации и маршрутизации данных.

4)Транспортный. Этот уровень называют иногда уровнем логического канала. Соответственно, здесь решаются проблемы управлением передачи данных и связанные с этим задачи: локализации и обработки ошибок, и непосредственно передачи данных.

5)Сеансовый. Обеспечивает взаимодействие программ, понятно, что машины сами по себе не взаимодействуют друг с другом — это делают программы. При этом решаются проблемы синхронизации передачи данных, подтверждение/установка паролей и т.д.

6)Представительский. На этом уровне решается проблема с представлением данных, ибо разные системы имеют разные способы представления данных.

7)Прикладной. Здесь на прикладном уровне решаются проблемы стандартизации взаимодействия с прикладными системами.

Было предложено использовать такую семиуровневую модель для стандартизации взаимодействия систем (сейчас практически на все среды передачи данных такие стандарты разработаны — но это только физический уровень) и желании использовать такую схему в реальной жизни. С последним возникла проблема, так как реальные сети этой схеме не удовлетворяют.

Считается, что на каждой из вычислительных систем, функционирующих в сети, существует набор уровней сетевого взаимодействия, соответствующего такой семиуровневой модели. При этом гарантированно считается, что существуют подряд идущие уровни снизу вверх. То есть если есть сеансовый уровень, то гарантировано есть все нижестоящие, если есть сетевой уровень, то есть и канальный и физический.

Далее, у нас есть две машины, на каждой из которых реализована такая семиуровневая модель. Система взаимодействия предусматривает то, что взаимодействие осуществляется между параллельными уровнями. То есть каждый уровень может общаться только с таким же на другой машине. Правило взаимодействия систем на одноименных уровнях называется протоколом передачи данных. При этом следует обратить внимание на то, что одноименные уровни напрямую друг с другом оперировать не могут. Они функционируют через нижние уровни. В пример, на схеме приводится путь взаимодействия сеансового уровня Машины 1 и сеансового уровня Машины 2

Каждый уровень модели может непосредственно взаимодействовать только с соседними уровнями. Правила взаимодействий между соседними уровнями называются интерфейсом.

Эта модель ISO/OSI требует жесткой стандартизации всех уровней, интерфейсов, протоколов. Если эти стандарты есть, то мы можем просто менять содержимое уровней (например, если у нас есть стандарты на вилки и розетки, то мы можем включать в электрическую сеть абсолютно разные приборы).

Интернет.

Теперь мы поговорим об Интернете, но поговорим о нем с внутренней — технической стороны.

Несколько слов предыстории:

В конце 60хх годов Американское Агенство Перспективных Исследований в Обороне (DARPA) приняло решение о создании экспериментальной сети ARPANET. Основным свойством этой сети являлось то, что предполагалось отсутствие какой-либо централизации в сети (во всех, рассмотренных нами многомашинных ассоциациях подразумевается некая централизация). В 70м годом ARPANET стала действующей сетью в США. В частности, через эту сеть можно было добираться до ведущих университетов и научных центров США. Например, во второй половине 70х годов появилось понятие суперкомпьютеров (их создавала Crey Researches),так вот через ARPANET можно было получить доступ к этим суперкомпьютерам и работать с ними.

В начале 80хх годов началась стандартизация протоколов взаимодействия в сети. Вся стандартизация проходила естественным путем (что нестандартно — то нежизнеспособно). Сначала стандартизации подверглись языки программирования, потом протоколы и взаимодействие в ОС. Также появилась модель ISO/OSI.

С этого момента (появления стандартов) можно говорить о начале Интернета.

Лекция 23

Мы начали обсуждать проблемы организации Internet и обозначили основное качество этой системы, заложенное изначально — что эта сеть абсолютна симметрична с той точки зрения, что она не подразумевала какой-либо централизации и иерархии. Это свойство, которое легло в основу сети, и создало тот бум, который наблюдается сейчас, то есть Internet может свободно расширяться.

Мы с вами рассмотрели вкратце предысторию сети. Изначально сеть подразумевала чисто экспериментальную работу и уже в дальнейшем получила университетскую распространенность, коммерция же пришла в Интернет где-то в 1994-95 годах.

Internet основан на протоколах TCP/IP. Иногда говорят: “протокол TCP/IP” — но это неправильно, так как под этой аббревиатурой скрывается целый набор протоколов, объединенных под одним названием. Кстати, здесь есть отдельно протокол TCP и протокол IP.

Семейство TCP/IP строится по 4х уровневой схеме. Рассмотрим таблицу соответствия TCP/IP модели ISO/OSI:

Уровни TCP/IP	Уровни ISO/OSI
I. Прикладных программ	Прикладных программ Представление данных
II. Транспортный	□ Сеансовый □ Транспортный

III. Межсетевой	5. Сетевой
IV. Доступа к сети	Канальный Физический

Уровень доступа к сети TCP/IP обеспечивают аппаратные интерфейсы и драйверы аппаратных средств. К примеру, протоколом уровня доступа к сети являются протоколы Ethernet. Их суть в следующем:

Ethernet — это система, обеспечивающая мнгновенный доступ с контролем несущей и обнаружением столкновений. Ethernet — широковещательная сеть, это означает, что любое сообщение, выходящее из источника становится видимым всем остальным ethernet-устройствам. Ethernet симметрична (нет никакого физического главенства), она предполагает наличие некоторой физической среды (разновидности коаксиального кабеля, кабель “витая пара”, СВЧ диапазон и др.), ethernet-устройства, которое осуществляет взаимодействие в рамках данной среды. Так как сеть симметрична, то возникает проблема столкновения пакетов передающихся данных, то есть, когда одновременно посылаются два пакета данных из разных устройств — в этом случае происходит отказ передачи данных у обоих устройств, после этого они замирают на некоторое время, а затем делают еще одну попытку. Это напоминает разговор вежливых людей в темной комнате, когда, два человека, начав говорить одновременно замолкают и делают паузу..

Следующее свойство Ethernet заключается в том, что каждое из ethernet-устройств имеет уникальный адрес, этот адрес присваивается ему при изготовлении. Существует ряд международных правил, которые создают невозможным появление в мире двух ethernet-устройств с одинаковым номером, будь-то уже сгоревшие устройства или еще находящиеся в строю. Этот адрес можно сравнить со штрих-кодом, который встречается на различных продуктах.

Широковещательность. Реально любое сообщение, посланное в сеть, проходит через все ethernet-устройства сети. Соответственно все устройства имеют адресацию, и сообщения могут адресоваться всем устройствам, либо какому-то отдельному, но в любом случае — сообщение пройдет через все устройства, а уж каждое из них само решит — оставить его или нет.

Вот в нескольких словах о примере четвертого уровня доступа протоколов TCP/IP, это наиболее распространенный вариант. Можно сказать, что такая сеть имеет ряд недостатков, заключающихся в том, что когда в сети возникает много активных пользователей, то учащаются столкновения сообщений и пропускная способность существенно снижается.

Сеть сетей

Следует обратить внимание, что когда мы говорим Интернет — сеть, то это также верно, как и то, что TCP/IP — протокол. То есть Интернет — это объединение сетей.

С этой точки зрения можно выделить два вида компьютеров, которые можно выделить в сети:

Это хост-компьютеры (host) и шлюзы (gate). В двух словах покажем, что есть что. Реально каждый из компьютеров, которые работают в сети, может классифицироваться по двум признакам: в компьютере расположена только одна сетевая карта или интерфейс (это хост-компьютер) и обычно он принадлежит какой-нибудь одной сети; в компьютере находится две и более сетевых карты, при этом каждая из карт подключается к своей сети (это компьютеры-шлюзы). Соответственно, через шлюзы можно объединять сети.

То есть, если смотреть с точки зрения принадлежности сетям — хост принадлежит одной сети, а шлюз принадлежит сразу двум или более сетям. Через шлюзы осуществляется взаимодействие между компьютерами в различных сетях. И этот

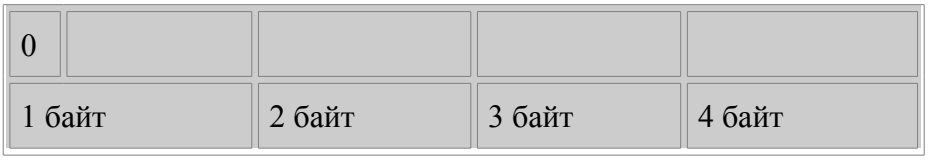
механизм объединения и доступа является одной из отличительных черт Интернета, которая базируется на межсетевом уровне TCP/IP, который в свою очередь базируется на протоколе IP.

Основная функция протокола IP — уникальная межсетевая адресация. Одним из основных свойств или качеств IP протокола является IP адрес. Это адрес, который приписывается как сети, так и конкретному компьютеру в сети. Исходя из этого мы можем сказать, что шлюз — это компьютер, имеющий два или более IP адреса (адрес одной сети и в другой сети), хост — компьютер, имеющий один IP адрес. Также в функции IP входит маршрутизация, то есть выбор пути, по которому будут передаваться сообщения, определение базовых блоков данных, которые передаются (они называются дейтаграмами) и взаимодействие с транспортным уровнем и уровнем доступа к сети, и, соответственно, в зависимости от этого взаимодействия возможна фрагментация и дефрагментация дейтаграм.

IP адресация

Два слова об IP адресации. IP адрес — это 4х байтовый код, в котором может размещаться информация об адресе сети и об адресе компьютера в сети. Существует несколько типов и категорий IP адресов:

Класс A.



Первый байт кодирует номер сети, при этом его старший бит является нулевым (это признак класса A), остальные биты определяют номер сети. Сетей класса A может быть 126 штук. Соответственно, последние три байта — номер компьютера в сети. Сети класса A — гигантские сети, которые могут принадлежать крупнейшим корпорациям.

Класс B.

Признак класса B — старшие два бита равны “10”. Для нумерации сети используется остаток первого и целиком второй байт. 3 и 4 байты — номер компьютера в сети. Это также большие сети, их может быть большое количество, но также ограниченное.

Класс C.

Признак класс C — старшие три бита равны “110”. Для нумерации сети используются: остаток первого байта, второй и третий байты целиком. Номер компьютера определяется четвертым байтом. Соответственно, класс C представляет большой набор небольших сетей.

Есть еще два класса — класс D и E, но они достаточно специфичны, и мы не будем о них говорить.

Существует международная организация, которая распределяет номера сетей, соответственно, здесь действует определенная иерархия, и организация, получившая номер сети может распределять номера компьютеров в пределах этой сети по собственному усмотрению.

Следует отметить, что, несмотря на огромное число адресов, которое можно представить через 4 байта, существует проблема их узкости, и идут разговоры о расширении IP адресации. Это колоссальная проблема, сравнимая разве что с проблемой 2000 года.

На межсетевом уровне кроме протокола IP существует еще группа вспомогательных протоколов. Часть из них зависит от того, чем мы будем пользоваться и что мы будем делать. В любом случае — основа для них — протокол IP.

Транспортные протоколы

Следующие протоколы — транспортные. Здесь присутствует два типа протоколов — UDP и TCP.

Протокол TCP обеспечивает передачу данных с контролем и исправлением ошибок. Кроме того, TCP гарантирует логическое соединение данных. То есть TCP позволяет создавать логические каналы данных, гарантируя отправку и прием порций данных в определенном порядке. Протокол жесткий, так как контролирует ошибки. Но за все надо платить, и TCP является ресурсоемким протоколом.

Протокол UDP — это быстрая доставка сообщений без осуществления контроля доставки.

Соответственно, протокол TCP больше рассчитан на использовании в Internet'е (для передачи на дальние расстояния, где не может гарантироваться безошибочность передачи). UDP ориентирован на работу в локальной сети, где гарантирован определенный уровень качества передачи данных. Протоколы транспортного уровня общаются с прикладными протоколами и межсетевыми.

Далее идет уровень прикладных систем. TCP/IP обладает тем свойством, что в семействе этих протоколов стандартизованы протоколы, на которых базируются прикладные системы. В частности, FTP (file transfer protocol). Реально система FTP присутствует в каждой системе. Но за счет того, что имеется стандарт FTP, все эти приложения работают единообразно. Есть сетевой продукт Telnet — сетевая эмуляция алфавитно-цифрового терминала.

То есть в системе стандартизованы протоколы с помощью которых организованы прикладные системы. И мы можем строить свои приложения FTP или Telnet из предоставленных кирпичиков.

Далее, разные прикладные системы общаются с разными протоколами — кто-то с UDP, кто-то с TCP. FTP и Telnet, например, работают через TCP, а сетевая файловая система NFS, которая позволяет объединять файловые системы разных машин в одну (и видеть их, как свою локальную), основывается на UDP.

Вот и все то, что можно было сказать о многомашинных ассоциациях, протоколах и по курсу в целом.