Лекция №5

**Операционная система**

**1. Назначение и основные функции операционной системы.**

Под термином «операционная система» мы будем понимать комплекс программ, функциями которого является контроль за использованием и распределением ресурсов вычислительной системы. Мы говорили, что в вычислительной системе есть физические ресурсы, то есть те ресурсы, которые связаны с реальным оборудованием (магнитные диски, оперативная память, время работы процессора). Мы говорили также, что в системе для ее успешного функционирования имеются логические (иногда их называют виртуальными) ресурсы, то есть ресурсы, которые в виде реального оборудования не существуют, но реализуются в виде некоторых средств, предоставляемых пользователю. Физические и логические ресурсы мы будем называть просто ресурсами вычислительной системы.

Любая операционная система (ОС) оперирует некоторыми сущностями, которые вместе со способами управления ими во многом характеризуют ее свойства**.** К таким сущностям могут относиться понятия файла, процесса, объекта, и т.д. Каждая ОС имеет свой набор таких сущностей. К примеру, в ОС Windows NT к таким сущностям можно отнести понятие объекта, и уже через управление этой сущностью предоставляются все возможные функции. Если мы посмотрим UNIX, то в ней такой сущностью, в первую очередь, является понятие файла, а во вторую очередь, понятие процесса.

Процесс - это некоторая сущность, которая присутствует практически во всех ОС. Процесс - это программа, имеющая права собственности на ресурсы. Рассмотрим две программы (то есть код и данные, которые используются) и рассмотрим все те ресурсы, которые принадлежат программе (это могут быть: пространство оперативной памяти, данные на внешнем запоминающем устройстве, права владения прочими ресурсами, к примеру, линиями связи). Если множества ресурсов, принадлежащих двум программам, совпадают, то в этом случае мы не можем говорить об этих программах, как о двух процессах, - это один процесс. Если у каждой программы есть свое множество ресурсов, причем, эти множества могут пересекаться, но не совпадать, то мы говорим о двух процессах.

В том случае, когда множества ресурсов нескольких процессов имеют непустое пересечение, то у нас возникает вопрос об использовании, так называемых, разделяемых ресурсов. Частично мы говорили об этом на прошлой лекции: вспомните пример с устройством печати. У нас может быть несколько процессов, каждый из которых имеет в качестве своего ресурса устройство печати и в каждый момент времени может обратиться к этому ресурсу с заказом на печать какой-то информации. Синхронизация работы процессов на примере устройства печати иллюстрировала нам одну из функций ОС, заключающуюся в управлении функционированием процессов. Давайте посмотрим, что понимается под управлением процессом.

**Управление процессами:**

1. Управление использованием времени центрального процессора.
2. Управление «подкачкой» и буфером ввода.
3. Управление разделяемыми ресурсами.

**Основные проблемы управления процессами**.

Первое - управление использованием времени центрального процессора (ЦП), или эту проблему иногда называют планированием ЦП, то есть управление тем, в какой момент времени какая из задач или какой из процессов будет владеть активностью ЦП: на какой из процессов будет работать ЦП.

Второе - управление «подкачкой» и буфером ввода. Предположим ситуацию, когда большое количество людей, например весь курс, сидит за компьютерами, и все одновременно запустили какие-то задания в виде процессов. В системе образовалась масса задач (гарантированно больше ста). А вся вычислительная система не может принять для работы в мультипрограммном режиме сто задач - это слишком много. В этом случае образуется, так называемый, буфер ввода задач, или буфер ввода процессов, то есть буфер, в котором аккумулируются те процессы, которые ожидают начала своей обработки процессором. Возникает проблема очередности выбора процессов из этого буфера для начала обработки. Это задача планирования буфера.

Теперь рассмотрим задачу планирования «подкачки». Процессором обрабатывается несколько процессов, и перед нами стоит задача освободить реальную оперативную память для других задач. В этом случае возникает необходимость какие-то из обрабатываемых задач откачать на внешнее запоминающее устройство. А по какому алгоритму мы будем откачивать эти задачи? Какова будет стратегия откачки? Можно откачивать, например, каждую четную задачу. Как более или менее выгодно организовать процесс откачки - это проблема.

Третье - управление разделяемыми ресурсами. Имеется набор ресурсов, доступ к которым в определенные моменты времени организуется от имени различных процессов. Это та самая коллизия с устройством печати. Одна из функций, которая во многом определяет свойства ОС, это функция, обеспечивающая организацию взаимодействия процессов и использования общих ресурсов. Проблема из примера с устройством печати легко решается, а вот если две программы имеют общий фрагмент оперативной памяти, то управление таким разделяемым ресурсом - сложная задача.

Теперь давайте посмотрим на конструкцию ОС. Практически любая ОС имеет понятие ядра. Ядром ОС обычно является ее резидентная часть, то есть та часть ОС, которая не участвует в процессах подкачки (она всегда присутствует в оперативной памяти) и работает в режиме ОС, или в режиме супервизора (в том самом специализированном режиме, о котором мы говорили на прошлой лекции). В ядро входят базовые средства управления основными сущностями, характерными для данной ОС, а также может входить набор программ, обеспечивающих управление некоторыми физическими устройствами. В функции ядра, в частности, входит обработка прерываний.

Мы иногда будем называть программы, управляющие ресурсами, драйверами устройств (физических или логических). К примеру, в ядро ОС должен входить драйвер оперативного запоминающего устройства.

Далее, вокруг ядра наращиваются программы управления ресурсами вычислительной системы. Первый уровень в основном состоит из драйверов физических устройств. Следующий уровень - управление логическими устройствами. И так далее. Таких уровней может быть достаточно много. Чем дальше от ядра, тем большая абстрактность имеет место быть. К примеру, где-то в нашей схеме могут появиться драйверы управления файлами, которые, на самом деле, связаны с драйверами управления логическими дисками, а те, в свою очередь, связаны с драйверами реальных физических устройств и так далее.

Вовсе не обязательно, что все компоненты ОС работают в режиме супервизора, или в режиме ОС. Многие из компонентов, которые логически достаточно удалены от ядра, могут работать в обыкновенном пользовательском режиме. Так же не обязательно, все эти компоненты ОС работают в резидентном режиме. Обычно, для многих функций это не требуется.

Теперь перейдем к более подробному рассмотрению основных функций ОС.

***Управление использованием времени центрального процессора.***

На самом деле, от того, какой алгоритм выбора задачи для передачи ей активности ЦП реализован в ОС, зависят многие реальные эксплуатационные свойства этой ОС. Выбор алгоритма почти целиком определяется теми критериями эффективности, которые используются для оценки эффективности работы ОС. Поэтому управление использованием времени ЦП мы с вами рассмотрим на фоне рассмотрения типов ОС.

Первая ситуация. У меня есть большое количество задач или программ, требующих большого объема вычислительных мощностей системы. Это те задачи, которые называют счетными задачами; они требуют большого объема вычислений и мало обращаются к внешним устройствам. Эти задачи должны выполняться на одной вычислительной системе. Что будет являться критерием эффективности для работы системы при выполнении этого пакета задач? Какой набор параметров можно взять и сказать: если они большие - то хорошо, если наоборот - то плохо? Для такой ситуации критерием эффективности работы вычислительной системы является степень загрузки ЦП. Если он мало простаивает (т.е. работает в режиме ожидания, а все процессы занимаются обменом, либо ОС берет на себя время), то мы можем сказать, что такая система работает эффективно. Этого можно добиться с использованием соответствующего алгоритма планирования, который заключается в следующем. Мы запускаем для обработки тот набор задач, который у нас есть по возможностям ОС (либо максимум, либо все задачи), что обеспечивается режимом мультипрограммирования. Алгоритм планирования времени ЦП в этом случае будет следующий: если ЦП выделен одному из процессов, то этот процесс будет занимать ЦП до наступления одной из следующих ситуаций:

1. Обращение к внешнему устройству.
2. Завершение процесса.
3. Зафиксированный факт зацикливания процесса.

Как только наступила одна из этих ситуаций, управление передается другому процессу. Количество передач управления от одного процесса к другому минимизировано. Так как при передаче управления с одного процесса на другой ОС должна выполнить набор некоторых действий, а это потеря времени, то здесь эти потери минимизированы. Такой режим работы ОС называется пакетным режимом. ОС, которая работает в таком режиме, называется **пакетной ОС**.

Теперь представим ситуацию, когда значительное количество человек находится в компьютерном классе и каждый из них редактирует некоторый текст. С каждым из терминалов, связана своя копия текстового редактора. Посмотрим, что будет с системой, если мы применим алгоритм планирования, изложенный для первого случая. Предположим, кто-то из пользователей слегка задремал за терминалом и не проявляет никакой активности. Время ЦП будет связано с этим процессом, потому что этот процесс не выполняет обмена и не завершился, так как редактор готов к работе. В это время все оставшиеся пользователи будут вынуждены ждать пробуждения заснувшего. Сложится ситуация зависания. Это означает, что алгоритм, который хорош для первого случая, для этой системы не годится даже при наличии самой мощной машины. Поэтому для задач, которые решают проблемы обеспечения большого количества пользователей вычислительными услугами (интерактивных задач), применяются другие алгоритмы, исходящие из других критериев эффективности.

Для такой системы подойдет критерий времени ожидания пользователя: с момента, как он послал заказ на выполнение какого-то действия, до момента ответа системы на этот заказ. Чем эффективнее работает система, тем это среднестатистическое время ожидания в системе меньше.

Рассмотрим ситуацию для второго случая. В системе находится некоторое количество процессов, и задача планировщика распределить время ЦП таким образом, чтобы время реакции системы на запрос пользователя было минимальным, либо по крайней мере, гарантированным. Предлагается следующая схема. В системе используется некоторый параметр Δt, который называют квантом времени (в общем случае, квант времени - это некоторое значение, которое может изменяться при настройке системы). Все множество процессов, которое находится в мультипрограммной обработке, подразделяется на два подмножества. Первое подмножество составляют те процессы, которые еще не готовы к продолжению выполнения: например, те процессы, которые заказали себе обмен и ждут его результатов. А есть процессы, которые готовы к выполнению. Работа будет осуществляться следующим образом. Тот процесс, который в данный момент времени занимает ЦП, будет владеть им до наступления одного из следующих событий:

1. Обращение с заказом на обмен.
2. Завершение процесса.
3. Исчерпание выделенного данному процессу кванта времени Δt.

При наступлении одного из этих событий планировщик ОС выбирает из процессов, готовых к выполнению, некоторый процесс и передает ему ресурсы ЦП. А выбирает он этот процесс в зависимости от того алгоритма планирования, который был использован в данной конкретной ОС. Например, процесс может выбираться случайно. Второй способ заключается в том, что происходит как бы последовательный обход процессов, то есть мы взяли в работу сначала один из процессов, затем он освободился, и время ЦП будет предоставлено следующему по порядку процессу из готовых к выполнению. Третьим критерием, по которому отбирается очередная задача, может быть время, которое данный процесс не обслуживался ЦП. В этом случае система может выбирать процесс, у которого такое время самое большое. Эти алгоритмы должны быть реализованы в ОС, а значит, они должны быть простыми, иначе система будет работать неэффективно, сама на себя (хотя такие системы есть: в частности, этим страдает семейство Windows).

Такой тип ОС называется **ОС разделения времени**. Она работает в режиме, при котором минимизируется время реакции системы на запрос пользователя. В идеале, за счет того, что время ответа на запрос минимально, у пользователя должна создаваться иллюзия, что все ресурсы системы предоставлены только ему.

Теперь посмотрим следующую задачу. Предположим, у нас есть самолет, управляемый автопилотом, который на автопилоте производит операцию снижения. У каждого самолета есть прибор, который измеряет высоту от самолета до поверхности земли. Режим работы самолета таков, что управление его функциями по некоторой заданной программе осуществляет компьютер. Итак, если у нас есть система автопилота, и самолет снижается, это система должна контролировать высоту полета. Центральный компьютер этого самолета может решать несколько задач: он может контролировать высоту полета, уровень топлива в баках, какие-то показатели работы двигателей и т.д. Управлением каждой из этих функций занимается свой процесс. Предположим, у нас пакетная ОС, и мы внимательно контролируем уровень топлива в баках. При этом, очевидно, возникает аварийная ситуация, ведь самолет продолжает снижаться, а ОС этого не замечает.

Предположим, у нас система разделения времени. Одним из качеств системы разделения времени является неэффективность за счет того, что в системе предусмотрено большое количество переключений с процесса на процесс, а эта функция достаточно трудоемка. Та же ситуация: высота подходит к нулю, а ОС занимается переустановкой таблиц приписки. Такой вариант тоже не подходит.

Для решения такого рода задач нужны свои средства планирования. В этом случае используются, так называемые, **ОС реального времени**, основным критерием которых является время гарантированной реакции системы на возникновение того или иного события из набора заранее предопределенных событий. То есть в системе есть набор событий, на которые система в любой ситуации прореагирует и обработает их за некоторое наперед заданное время. Для нашего примера таким событием может быть поступление информации от датчика высоты. Реально для ОС этого класса используются достаточно простые алгоритмы. Все планирование заключается в этом критерии, то есть гарантируется обработка события за время, не превышающее некоторого порогового значения. Но ОС реального времени обычно имеет свое специфическое устройство, которое определяется не только этим простым алгоритмом планирования, но и внутренним переустройством системы.

Подводя некоторую черту под функцией управления использованием времени ЦП и планирования ЦП, обращаю внимание на два факта. Первый факт это то, что те алгоритмы, которые реализованы в системе планирования распределением времени ЦП во многом определяют эксплуатационные свойства вычислительной системы. Я специально приводил примеры, предлагая использовать разные ОС для разных целей. Второй факт. Мы рассмотрели три типовых разновидности ОС: системы пакетной обработки, системы разделения времени и системы реального времени. На сегодняшний день можно говорить о том, что система реального времени это отдельный класс ОС. Гарантированно, ОС Windows не будет управлять какими-то объектами, у которых это реальное время очень критично. Также не будет управлять такими объектами и ОС СОЛЯРИС или LINUX и т.д., потому что эти системы не являются системами реального времени.

Первые два режима, пакетный и разделения времени, можно сэмулировать на таких общепринятых ОС. Реально, большие и серьезные ОС являются смешанными системами, т.е. у них присутствует в элементах планирования ЦП как алгоритмы, позволяющие управлять счетными задачами, так и алгоритмы, позволяющие управлять интерактивными задачами либо задачами отладочными, для которых надо немного ЦП.

Примером такой организации планирования ЦП может быть следующая схема. Планировщик построен на двухуровневой схеме. Мы считаем, что множество задач может содержать, предположим, счетные задачи и интерактивные задачи. Первый уровень определяет приоритет между двумя классами задач и либо отдает ЦП сначала счетной задаче, либо интерактивной задаче. А второй уровень определяет то, о чем мы говорили перед этим, т.е. как выбрать задачу в пределах одного класса и как ее прервать. Такая смешанная система может работать следующим образом. Первый уровень планирования будет работать по такому принципу: если в данный момент нет ни одной интерактивной задачи, готовой к выполнению (а это вполне реальная ситуация, если пользователи занимаются редактированием текста), то ЦП передается счетным задачам, но добавляется одно условие: как только появляется хотя бы одна интерактивная задача, счетная задача прерывается и управление передается блоку интерактивных задач. Это то, что касается первой функции управления процессами.

***Управление подкачкой и буфером ввода.***

Здесь алгоритмы планирования нужные, но не столь критичные. В реальных системах зачастую совмещается буфер подкачки, т.е. то пространство на внешних носителях, куда осуществляется откачка информации из оперативной памяти, и буфер ввода процессов. Это первое замечание.

Второе замечание. Современные ОС достаточно «ленивы» и откачку зачастую осуществляют не единицами блоков памяти процессов, а откачивается весь процесс. Здесь возникает два вопроса: каков критерий замещения процесса и каков критерий выбора из буфера того процесса, который нам требуется ввести для мультипрограммной обработки. Самый простейший вариант заключается в использовании времени нахождения в том или ином состоянии. В первом случае, если мы решаем вопрос об откачке процесса из активного состояния из числа обрабатываемых в область подкачки, то можем взять тот процесс, который дольше всего находится в состоянии обработки по астрономическому времени. Обратный процесс может быть симметричен, т.е. мы можем брать из буфера ввода процессов тот процесс, который дольше всего там находится. На самом деле, это простые и реальные алгоритмы планирования, и они могут видоизменяться из соображения разных критериев. Один из критериев, если все задачи подразделены на различные категории, т. е. могут быть задачи ОС - в этом случае они рассматриваются в первую очередь (и среди них действует алгоритм оценки времени нахождения их там-то) и все остальные задачи. Такая модель похожа на модель жизненной несправедливости в обществе, где есть всемогущие люди, которым все дозволительно, а есть люди, которые могут и подождать.

***Управление разделяемыми ресурсами.***

Здесь мы обозначим только проблему, потому что конкретные ее решения мы рассмотрим на примере ОС UNIX.

Предположим, есть два процесса, которые работают на общем пространстве оперативной памяти. При этом разделяемые ресурсы могут работать в разных режимах, т.е. не исключена ситуация, когда два процесса реально находятся на разных машинах, но они связаны общем полем оперативной памяти. В этом случае возникает проблема с буферизацией работы с памятью, потому что на каждой из машин есть свои механизмы буферизации чтения-записи. Здесь возникает нехорошая ситуация, когда состояние физической памяти не соответствует реальному ее содержимому. А также возникают некоторые проблемы для ОС, работающей на двух машинах.

Следующая проблема. Пусть есть два процесса, которые работают на одной машине. Должны быть определенные средства, которые позволят синхронизовать доступ к разделяемой памяти, то есть позволят создать условия, при которых обмен каждого из работающих процессов с оперативной памятью будет происходить корректно. Это значит, что при каждом чтении информации из разделяемой памяти должно быть гарантированно, что все пользователи, которые начали писать что-то в эту память, уже этот процесс завершили - должна быть синхронизация по обмену с разделяемой памятью.

В действительности при решении задач зачастую не требуется таких разделяемых ресурсов, как общая память, но хотелось бы, чтобы процессы, которые функционируют одновременно, могли оказывать некоторое влияние друг на друга. Влияние, аналогичное аппарату прерываний. Для реализации этого во многих ОС имеются средства передачи сигналов между процессами, тогда возникает некоторая программная эмуляция прерываний. Один процесс говорит - передай сигнал другому процессу. В другом процессе возникает прерывание выполнения этого процесса и передача управления на некоторую предопределенную функцию, которая должна обработать полученный сигнал. Это третья функция ОС.

Я обратил ваше внимание на такие функции ОС, которые влияют на ее эксплуатационные свойства. Реально любая ОС также содержит огромный набор других функций, которые обеспечивают работу этой системы.

Лекция №6

На прошлой лекции мы говорили о том, что практически любая операционная система обеспечивает буферизацию ввода/вывода. На самом деле, это одна из основных функций операционной системы. По аналогии борьбы с разными скоростями доступа к различным компонентам вычислительной системы операционная система вводит в своих пределах программную буферизацию, которая также решает проблемы сглаживания времени доступа и проблемы синхронизации в целом (пример с устройством печати). Сглаживание проблем доступа заключается в том, что практически каждая операционная система имеет КЭШ-буфера, которые аккумулируют обращения к внешнему запоминающему устройству (ВЗУ) аналогично аппаратной буферизации при работе с оперативной памятью. Это позволяет существенно оптимизировать операционную систему. Признаком наличия такой буферизации является требование прекратить выполнение операционной системы перед выключением машины. Например, работая с операционной системой MS-DOS, можно выключить компьютер в любой момент времени, потому что такой буферизации в ней нет. В операционных системах типа Windows и UNIX считается некорректным просто выключить машину при работающей системе, в этом случае есть вероятность, что произойдет некоторая потеря информации (так как, например, моменты заказа на обмен и непосредственно обмена далеко не совпадают). Степень этой буферизации определяет реальную эффективность системы. Когда на нашем факультете стали появляться Pentium-ы, то обнаружилось, что при работе с Windows 95 практически нет качественного различия между тем, работает ли система на 486 процессоре или на Pentium-е. Это говорит о том, что эффективность системы не упирается в эффективность работы с внешним устройством. Если взять операционную систему UNIX, то эта разница будет заметна, так как здесь быстродействие процессора сильнее влияет на качество работы системы, нежели чем для Windows 95, потому что в системе Windows 95 обменов с внешним носителем значительно больше за счет некоторой «тупости» алгоритмов буферизации работы с внешними устройствами.

**2. Файловая система.**

Мы с вами говорили, что каждая из операционных систем оперирует некоторыми сущностями, одной из которых является процесс. Есть вторая сущность, которая также важна - это понятие файла. *Файловая система - это компонент операционной системы, обеспечивающий организацию создания, хранения и доступа к именованным наборам данных. Эти именованные наборы данных называются файлами*.

***Основные свойства файлов***

**1. Файл** - это некий объект, имеющий имя, и позволяющий оперировать с содержимым файла через ссылку на это имя. Обычно имя - это последовательность некоторых символов, длина которой зависит от конкретной операционной системы.

**2. Независимость файла от расположения**. Для работы с конкретным файлом не требуется иметь информацию о местоположении этого файла на внешнем устройстве.

**3. Набор функций ввода/вывода.** Практически каждая операционная система однозначно определяет набор функций, обеспечивающий обмен с файлом. Обычно, этот набор функций состоит из следующих запросов:

1. Открыть файл для работы. Открыть можно либо уже существующий, либо новый файл. Может возникнуть вопрос - зачем открывать файл? Почему нельзя сразу читать и писать в этот файл? На самом деле, это есть средство, для того чтобы централизованно объявить операционной системе, что файл будет работать с конкретным процессом. А она уже из этих сведений может принять какие-то решения (например, блокирование доступа в этот файл для других процессов).
2. Чтение/запись. Обычно обмен с файлами может организовываться некоторыми блоками данных. Блок данных, с которым происходит обмен, несет двоякую сущность. С одной стороны, для любой вычислительной системы известны размеры блоков данных, которые наиболее эффективны для обмена, то есть это программно-аппаратные размеры. С другой стороны, эти блоки данных при реальном обмене могут варьироваться достаточно произвольно программистом. В функциях чтения/записи обычно фигурирует размер блока данных для обмена и количество блоков данных, которые необходимо прочесть или записать. От выбранного размера блока данных может зависеть эффективность реальных обменов, потому что, предположим для некоторой машины размером эффективного блока данных является 256Кб, а вы хотите обмены проводить по 128Кб, и вы выполняете два обращения для прочтения ваших логических блоков по 128Кб. Очень вероятно, что вместо того, чтобы за один обмен прочесть блок в 256Кб, вы обращаетесь два раза к одному блоку и читаете сначала одну половину, а затем другую. Здесь есть элементы неэффективности, хотя они могут сглаживаться «умной» операционной системой, а если она не сглаживает, то это уже ваша вина.
3. Управление файловым указателем. Практически с каждым открытым файлом связывается понятие файлового указателя. Этот указатель, по аналогии с регистром счетчика команд, в каждый момент времени показывает на следующий относительный адрес по файлу, с которым можно произвести обмен. После обмена с данным блоком указатель переносится на позицию через блок. Для организации работы с файлом требуется уметь управлять этим указателем. Имеется функция управления файловым указателем, которая позволяет произвольно (в пределах доступного) перемещать указатель по файлу. Указатель есть некоторая переменная, доступная программе, которая связана с функцией открытия файла (создающей эту переменную).
4. Закрытие файла. Эта операция может осуществляться двумя функциями: 1) Закрыть и сохранить текущее содержимое файла. 2) Уничтожить файл.

После закрытия файла все связи с ним прекращаются, и он приходит в некоторое каноническое состояние.

**4. Защита данных**. Многие стратегические решения повторяются как на аппаратном уровне, так и на уровне операционной системы. Если мы вспомним мультипрограммный режим, то одним из необходимых условий его существования является обеспечение защиты (памяти и данных). Если мы рассмотрим файловую систему, то она так же, как и операционная система, может быть однопользовательской. В этом случае проблемы защиты данных не существует, потому что человек, который работает с этой операционной системой, является хозяином всех файлов. Примеры однопользовательских систем - MS-DOS или Windows 95. Можно загрузить машину и уничтожить все файлы других пользователей, которые размещены на диске, потому что в этих системах защиты нет никакой. Многопользовательская система обеспечивает корректную работу многих пользователей. MS-DOS также может работать в режиме мультипрограммирования, но он не достаточно корректен, потому что ошибка в одном процессе может привести к затиранию операционной системы и соседнего процесса. Также и в операционной системе Windows 95 может работать много пользователей, но эта работа некорректна, потому что эта операционная система не обеспечивает все права защиты. Итак, многопользовательская система должна обеспечивать защиту информации от несанкционированного доступа. На самом деле, проблема защиты связана не только с файловой системой. Реально операционная система обеспечивает защиту данных во всех областях: это и файлы, и процессы, и ресурсы, принадлежащие процессам, запущенным от имени одного пользователя. Здесь я обращаю ваше внимание на этот факт, потому что для файлов это наиболее критичная точка.

***Основные свойства файловых систем.***

Файловая система естественно включает в себя все те свойства, которые были перечислены для файлов, но добавляет еще некоторые. Эти свойства связаны со структурной организацией файловой системы.

Давайте рассмотрим некоторое пространства ВЗУ, и рассмотрим, как мы можем организовать размещение файлов в пределах этого пространства.

**1. Одноуровневая организация файлов непрерывными сегментами.** Термин «одноуровневая» означает, что система обеспечивает работу с файлами уникально именованными.В пределах пространства ВЗУ выделяется некоторая область для хранения данных, которая называется каталог. Каталог имеет следующую структуру:

|  |  |  |
| --- | --- | --- |
| имя | начальный блок | конечный блок |
|  |  |  |

«Начальный блок» ссылается на некоторый относительный адрес пространства ВЗУ, с которого начинается файл с заданным именем. «Конечный блок» определяет последний блок данного файла. Функция открытия файла сводится к нахождению в каталоге имени файла и определении его начала и конца (реально данные могут занимать несколько меньше места, об этом будет сказано позже). Это действие очень простое, к тому же каталог можно хранить в памяти операционной системы, и тем самым уменьшить количество обменов. Если создается новый файл, то он записывается на свободное место. Аналогично каталогу имен может иметься таблица свободных пространств (фрагментов).

Чтение/запись происходит почти без дополнительных обменов, так как при открытии мы получаем диапазон размещения данных. Чтение происходит в соответствии с этой блочной структурой и никакая дополнительная информация не требуется, соответственно обмен происходит очень быстро.

Что будет, когда нужно записать в такой файл дополнительную информацию, а свободного пространства за этим файлом нет? В этом случае система может поступить двояко. Первое, она скажет, что нет места и вы должны сделать что-то сами, например, запустить некий процесс, который перенесет этот файл в другое место и добавит нужную информацию. Этот перенос - функция достаточно дорогостоящая. Вторая возможность - в обмене будет отказано. Это означает, что при открытии файла нужно было заранее зарезервировать дополнительное место; при этом файловая система проверяет размер свободного буфера, и если его мало, то ищет свободное место там, где этот файл разместится.

Итак, мы видим, что эта организация проста, при обменах эффективна, но в случае нехватки пространства для файла начинается неэффективность. К тому же, при долговременной работе такой файловой системы на диске случается то же, что у нас случалось с оперативной памятью - фрагментация. То есть ситуация, когда есть свободные фрагменты, но среди них нет такого, куда можно было бы разместить файл. Борьба с фрагментацией для такой организации файловой системы - это периодическая компрессия, когда запускается долгий, тяжелый и опасный для содержимого файловой системы процесс, который прижимает все файлы плотно друг к другу.

Такая организация может быть пригодна для однопользовательской файловой системы, потому что при большом количестве пользователей очень быстро произойдет фрагментация, а постоянный запуск компрессии - смерть для системы. С другой стороны, система проста и не требует почти никаких накладных расходов.

**2. Файловая система с блочной организацией файлов.** Пространство ВЗУ разделено на блоки (те самые блоки, которые эффективны для обмена). В файловой системе такого типа распределение информации происходит аналогично распределению информации процесса в оперативной памяти со страничной организацией. В общем случае, с каждым именем файла связан набор номеров блоков устройства, в которых размещены данные этого файла. Причем, номера этих блоков имеют произвольный порядок, то есть блоки могут быть разбросаны по всему устройству в произвольном порядке. При такой организации нет фрагментации, хотя могут быть потери кратные блоку (если файл занял хотя бы один байт в блоке, то весь блок считается занятым). Следовательно, нет проблем компрессии, и эта система может использоваться при многопользовательской организации.

В этом случае с каждым файлом связан набор атрибутов: имя файла, имя пользователя, по которым происходит доступ к файлу. Такая организация позволяет уйти от уникальности имен, которая требовалась в предыдущем случае. В такой системе требуется уникальность имен лишь среди файлов одного пользователя.

Организация таких файлов может быть через каталог. Структура каталога может быть следующая. Каталог содержит строки; каждая i-тая строка соответствует i-тому блоку файловой системы. В этой строке содержится информация о том, является ли этот блок свободным или занятым. Если он занят, то в этой строке указывается имя файла (либо ссылка на него), имя пользователя, и может находиться какая-то дополнительная информация.

При обмене система может действовать по-разному. Либо при открытии файла система пробегает по всему каталогу и строит таблицу соответствия логических блоков файла, их размещению на диске. Либо при каждом обмене осуществляется поиск этого соответствия.

Такая организация файловой системы является одноуровневой в рамках одного пользователя, то есть все файлы связаны в группы по принадлежности к какому-то пользователю.

**3. Иерархическая файловая система.** Все файлы файловой системы построены в структуру, которая называется деревом. В корне дерева находится, так называемый, корень файловой системы. Если узел дерева является листом, то это файл, который может содержать данные пользователя, либо являться файлом-каталогом. Узлы дерева отличные от листа являются файлами-каталогами. Именование в такой иерархической файловой системе может происходить разными способами. Первый тип - именование файла относительно ближайшего каталога, т. е. если мы посмотрим файлы, которые являются ближайшими для каталога F0, - это файл F1, который является также каталогом, и файл F2. Для успешного именования в такой системе на одном уровне не могут повторяться имена. С другой стороны, так как все файлы связаны с помощью дерева, мы можем говорить о, так называемом, полном имени файла, которое составляется из всех имен файлов, которые составляют путь от корня файловой системы к конкретному файлу. Полное имя файла F3 будет обозначаться так: /F0/F1/F3. Такая организация хороша тем, что она позволяет работать как с коротким именем файла (если системно подразумевается, что мы работаем в данном каталоге), так и с полным именем файла. Полные имена файлов есть пути, а в любом дереве от его корня до любого узла существует единственный путь, следовательно, этим решается проблема унификации имен. Первый раз такой подход был использован в операционной системе Multix, которая разрабатывалась в университете Беркли в конце 60-х годов. Это красивое решение стало появляться впоследствии во многих операционных системах. Согласно этой иерархии, каждому из файлов можно привязывать какие-то атрибуты, связанные с правами доступа. Правами доступа могут обладать как пользовательские файлы, так и каталоги. Структура этой системы хороша для организации многопользовательской работы, за счет отсутствия проблемы именования, и такая система может очень хорошо наращиваться.

**4. Персонификация и защита данных в операционной системе.** Этот нюанс, который мы сейчас рассмотрим, и простой, и сложный. Простой - потому что мы скажем о нем буквально несколько фраз, а сложный, потому что существуют проблемы, о которых можно говорить долго.

Персонификация - это возможность операционной системы идентифицировать конкретного пользователя и в соответствии с этим принимать те или иные действия, в частности, по защите данных.

Если мы с вами посмотрим на любимую нами операционную систему MS-DOS, то там не было понятия пользователя со всеми вытекающими последствиями - она однопользовательская.

Второй уровень операционных систем - это операционные системы, которые позволяют регистрировать пользователей, но все пользователи представляются в виде единого набора некоторых субъектов и не связаны друг с другом никак. Примером таких операционных систем могут служить некоторые операционные системы фирмы IBM для mainframe-компьютеров. Например, лектор не знает, кто из его слушателей к какой группе относится, но все, сидящие перед ним, пользователи его курса. Это и хорошо, и плохо. С точки зрения прослушивания курса лекций - это хорошо, но для проведения этим лектором какого-то опроса это плохо, потому что за один день он не успеет опросить всех. Ему надо будет всех слушателей как-то поделить, а как - не известно.

Соответственно, при такой одномерной персонификации обеспечиваются все те функции, о которых мы с вами говорили (в частности защита), но такая организация пользователей не предполагает образования групп пользователей. А мне удобно, чтобы, предположим, на нашем факультетском сервере моя лаборатория была выделена, и в рамках этой лаборатории можно было бы предоставлять друг другу права доступа к файлам и т.д.

Соответственно, аналогично файловой системе, появляется иерархическая организация пользователей. То есть у нас имеется понятие «все пользователи» и понятие «группа пользователей». В группе есть реальные пользователи. Такая иерархическая организация персонификации влечет за собой следующие моменты. При регистрации какого-то пользователя необходимо сначала привязать его к какой-то группе - это может быть лаборатория, кафедра или учебная группа. Так как пользователи объединены в группы, то появляется возможность разделения прав доступа к ресурсам пользователей. То есть пользователь может, например, заявить, что все его ресурсы доступны для всех пользователей группы. Такая схема может быть многоуровневой (группы делятся на подгруппы и т.д.) с соответственным распределением прав и возможностей. Сейчас появляются операционные системы, в которых права доступа могут определяться не только такой иерархической структурой, но и могут быть более сложными, т. е. права доступа можно добавлять, нарушая эту иерархию.

Лекция №7

**3. Операционная система UNIX.**

Мы переходим к изучению операционной системы UNIX, поскольку многие решения, которые принимаются в операционных системах, мы будем рассматривать на примере этой операционной системы.

В середине 60-х годов в Bell Laboratories фирмы AT&T проводились исследования и разработка одной из первых операционных систем в современном ее понимании - операционной системы Multix. Эта операционная система обладала свойствами операционной системы разделения времени, многопользовательской системы, а также в этой системе были предложены основные решения по организации файловых систем, в частности, была предложена иерархическая древообразная файловая система. От этой разработки через некоторое время получила начало операционная система UNIX. Одна из историй разработки этой системы говорит о том, что на фирме был ненужный компьютер PDP-7 с очень малоразвитым программным обеспечением и требовалась машина, которая позволяла бы организовывать комфортную работу пользователя, в частности, обработку текстовой информации. Известная группа людей - это Кен Томпсон и Деннис Ритчи, занялись разработкой новой операционной системы. Другой вариант этой истории гласит о том, что якобы они занимались реализацией некоторой игры и те средства, которые были им доступны, оказались неудобны - тогда они решили поиграть с этой машиной. В результате появилась операционная система UNIX.

Особенностью этой системы являлось то, что она была первой системной программой, которая была написана с использованием языка, отличного от машинного языка (ассемблера). Для целей написания этого системного программного обеспечения, в частности, операционной системы UNIX, также 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 Си. С 1983 года операционная система UNIX (ее первоначальная версия) была переписана на язык Си, и получилось, что около 90% операционной системы было написано на языке высокого уровня, не зависящем от архитектуры машины, а 10% этой системы были написаны на ассемблере. В эти десять процентов вошли наиболее критичные по времени части операционной системы.

Итак, первым важным и революционным результатом было использование языка высокого уровня. Этот факт вызывал обсуждения, потому что никто не верил, что это может быть долговременно, поскольку всегда язык высокого уровня ассоциировался с большой неэффективностью. Язык Си был сконструирован таким образом, что позволял, с одной стороны, писать достаточно эффективные программы, с другой стороны, транслировать его в эффективный код.

Первое свойство языка Си, которое повышало его эффективность, - это работа с указателями. Второе свойство заключается в том, что при программировании на ассемблере мы часто используем побочные эффекты. Например, эффект, когда результатом вычисления какого-то выражения является не только записанной значение, но и какие-то промежуточные значения, которые можно по ходу записать для последующего их использования. Эти возможности стали доступны и в языке Си, потому что понятие выражения в языке Си гораздо шире, чем в тех языках, которые имелись в наличии в то время. В частности, появилась операция присваивания, вместо оператора присваивания, которая позволила программировать побочные эффекты. Эти свойства и определили «живучесть» языка, пригодность для программирования системных компонентов и возможность оптимальной трансляции кода различных машин.

С профессиональной (канонической) точки зрения, язык Си - ужасный язык. Основным требованием, которое предъявляется к языкам программирования, является обеспечение безопасности программирования. Средства языка должны минимизировать вероятность внесения ошибок в программу, и заведомо к таким средствам современных языков относится следующее: жесткий контроль типов (т.е. нельзя, например, сложить целочисленную переменную с вещественной, не преобразовав перед этим тип одной из них к типу другой). Язык Си обладает возможностью преобразования типов по умолчанию. Другая криминальная вещь - это обеспечение контроля за доступом к памяти программы (т.е. если в ячейке памяти хранится вещественное число, то мы не можем его проинтерпретировать никак иначе). Возможность бесконтрольного использования этих значений предоставляют указатели. Более того, через указатель можно «обманывать» функции в соответствии и несоответствии фактических параметров формальным параметрам и т.д. Третье свойство - это контроль за взаимодействием модулей. Много ошибок появляется в том случае, если в функции продекларирован один набор формальных параметров, а обращение к ней происходит по другому набору (причем отличия могут быть как по количеству параметров, так и по типам). В языке Си всегда можно обмануть программу - вместо формального параметра одного типа дать параметр другого типа, и вместо десяти параметров передать один. Это приводит к ошибкам.

Вот три позиции, по которым язык Си не удовлетворяет требованиям безопасности. Однако опыт показывает, что наиболее живучими оказываются плохие (с этой точки зрения) языки.

Итак, 1973 год - год появления написанной на языке Си операционной системы UNIX. Какими основными свойствами обладала эта система? Первое свойство - это концепция файлов. Основным объектом, которым оперирует операционная система, является файл. Файл, с точки зрения операционной системы UNIX, - это внешнее устройство. Файл - это каталог, который содержит информацию о содержащихся в нем файлах. И так далее, на сегодняшний день, файлом может считаться, в некотором смысле и процесс, который может работать.

Второе свойство - это особая структура операционной системы. В отличие от предыдущих операционных систем, в которых каждая команда была «зашита» внутрь операционной системы, т.е. ее нельзя было как -либо модифицировать, в UNIX-е проблемы команд решены очень элегантно. Во-первых, UNIX декларирует стандартный интерфейс передачи параметров извне внутрь процесса. Во-вторых, все команды реализованы в виде файлов. Это означает, что можно свободно добавлять новые команды в систему, а также убирать и модифицировать их. То есть система UNIX открыта и ее можно легко развивать.

Начнем рассмотрение конкретных свойств операционной системы.

***Файловая система. Организация файлов. Работа с файлами.***

Файловая система UNIX-а - это многопользовательская иерархическая файловая система. Ее структуру мы рисовали на прошлой лекции. Она представима деревом, корнем которого является, так называемый, корневой каталог. Узлами, отличными от листьев дерева, являются каталоги. Листьями могут являться либо файлы (в традиционном понимании), либо пустые каталоги. В системе определено понятие имени файла - это имя, которое ассоциировано с набором данных в рамках каталога, которому этот файл принадлежит. Кроме того, имеется понятие полного имени - это уникальный путь от корня файловой системы до конкретного файла. Разрешено совпадение имен файлов, находящихся в разных каталогах.

На самом деле, файловая система не совсем древообразная. Имеется возможность нарушения иерархии за счет средств, позволяющих ассоциировать несколько имен (полных) с одним и тем же содержимым файла, - это, так называемые, ссылки. Тем не менее, мы будем говорить, что файловая система иерархическая и древообразная.

В операционной системе UNIX используется трехуровневая иерархия пользователей (все пользователи разделены на группы), структура которой в общем случае была рассмотрена в предыдущей лекции. В связи с этим каждый файл файловой системы обладает двумя атрибутами. Первый атрибут - это, так называемый, владелец файла. Этот атрибут связан с одним конкретным пользователем, который является владельцем файла. Владельцем файла можно стать по умолчанию, если создать этот файл, а также есть команда, которая позволяет менять владельца файла. Второй атрибут - это атрибут, связанный с защитой доступа к файлу (подробнее об этом будет сказано позже).

Доступ к каждому файлу регламентируется по трем категориям. Первая категория - это права владельца файла. В общем случае владелец не обязательно может делать с этим файлом все что угодно. Вторая категория - права группы, к которой принадлежит владелец файла, (отличные от прав владельца). Третья категория - права остальных пользователей системы, за исключением соответствующей группы. По этим трем категориям регламентируются три действия - чтение из файла, запись в файл и исполнение файла. В каждом файле определено, может ли пользователь данной категории читать файл, писать в него и запускать его как исполняемый в качестве процесса.

***Работа с файлами***

Рассмотрим структуру файловой системы на диске. Для любой вычислительной системы определено понятие системного внешнего запоминающего устройства (СВЗУ). СВЗУ - это устройство, к которому осуществляет доступ аппаратный загрузчик машины с целью запуска операционной системы. Почти любая вычислительная машина имеет диапазон адресного пространства памяти, размещенный в, так называемом, постоянном запоминающем устройстве (ПЗУ). В ПЗУ размещается небольшая программа (аппаратный загрузчик), которая при включении или аппаратной перезагрузке машины, обращается к фиксированному блоку СВЗУ, размещает его в оперативной памяти и передает управление на фиксированный адрес, принадлежащий считанным данным. Считается, что этот считанный блок данных является, так называемым, программным загрузчиком. Программный загрузчик далее «раскручивает» запуск операционной системы. Следует отметить, что аппаратный загрузчик не зависит от операционной системы, а программный загрузчик является компонентом операционной системы. Он уже знает, где размещаются данные, необходимые для запуска операционной системы.

В любой системе принято разбиение пространства ВЗУ на некоторые области данных, которые называются блоками. Размер логического блока является фиксированным атрибутом операционной системы. В операционной системе UNIX размеры блока определяет некоторый параметр, который может меняться в зависимости от версии системы. Для определенности будем говорить, что логический блок ВЗУ равен 512 байт.

Представим адресное пространство СВЗУ в виде последовательности блоков. Нулевой блок СВЗУ - это блок начальной загрузки, или блок, в котором находится программный загрузчик. Размещение этого блока в нулевом блоке СВЗУ определяется аппаратно, потому что аппаратный загрузчик обращается всегда именно к нулевому блоку. Это последний компонент файловой системы, который зависит от аппаратуры.

Следующий блок - это суперблок файловой системы. Он содержит оперативную информацию о текущем состоянии операционной системы, а также данные о параметрах настройки файловой системы. В частности, суперблок содержит информацию о количестве, так называемых, индексных дескрипторов в файловой системе. Также суперблок содержит информацию о количестве блоков, составляющих файловую систему, а также информацию о свободных блоках файлов, о свободных индексных дескрипторах, и прочие данные, характеризующие время, дату модификации и другие специальные параметры.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Блок  начальной  загрузки | Суперблок  файловой системы | Область  индексных дескрипторов | Блоки  файлов | Область  сохранения |
| 0 |  |  |  |  |

За суперблоком следует область (пространство) индексных дескрипторов. Индексный дескриптор - это специальная структура данных файловой системы, которая ставится во взаимно однозначное соответствие с каждым файлом. Размер пространства индексных дескрипторов определяется параметром генерации файловой системы по количеству индексных дескрипторов, которые указаны в суперблоке. Каждый индексный дескриптор содержит следующую информацию:

1. Поле, определяющее тип файла (каталог или нет).
2. Поле кода защиты.
3. Количество ссылок к данному индексному дескриптору из всевозможных каталогов файловой системы (в ситуации нарушения дерева файловой системы). *Если значение этого поля равно нулю, то считается что этот индексный дескриптор свободен.*
4. Длина файла в байтах.
5. Статистика: поля, характеризующие дату и время создания и т.п.
6. Поле адресации блоков файлов.

Следующее пространство файловой системы - это блоки файлов. Это пространство на системном устройстве, в котором размещается вся информация, хранящаяся в файлах и о файлах, которая не поместилась в предыдущие блоки файловой системы.

Последняя область данных в разных системах реализуется по-разному, но для простоты изложения, мы будем считать, что эта область находится сразу за блоками файловой системы - это область сохранения.

Такова концептуальная структура файловой системы. Теперь рассмотрим детали.

**Суперблок**. Наибольший интерес в суперблоке вызывают последние два поля: поля информации о свободных блоках файлов и свободных индексных дескрипторах. В файловой системе UNIX-а заметно влияние двух факторов. Первый фактор - это то, что файловая система разрабатывалась в те времена, когда объем винчестера в 5-10Мб считался очень большим. Внутри структуры файловой системы заметны старания авторов оптимизировать использование пространства внешнего устройства. Второй фактор - свойство файловой системы по оптимизации доступа. Критерий оптимальности доступа - количество обменов файловой системы с внешним устройством, которые она производит для обеспечения нужд.

Список (массив) свободных блоков файлов состоит из пятидесяти элементов и занимает 100 байтов. В буфере, состоящем из пятидесяти элементов, записаны номера свободных блоков пространства блоков памяти. Эти номера записаны со второго элемента по сорок девятый. Первый элемент массива содержит номер последней записи в этом массиве. Нулевой элемент этого списка содержит ссылку на блок пространства блоков файлов, в котором этот список продолжен, и т.д.

Если какому-то процессу требуется для расширения размера его файла дополнительный свободный блок, то система по указателю Номер Блока (НБ) выбирает элемент массива (копия суперблока всегда присутствует в оперативной памяти, поэтому почти при всех таких действиях система не нуждается в обращении к СВЗУ) и этот блок предоставляется соответствующему файлу для расширения (при этом корректируется указатель НБ). Если происходит сокращение размера файла или удаление всего файла, то высвободившиеся номера записываются в массив свободных блоков, при этом также происходит коррекция указателя НБ.

Так как размер массива равен пятидесяти элементам, то возможны две критические ситуации. Первая - когда при освобождении новых блоков не удается поместить их номера в этом массиве, так как он уже полон. В этом случае из файловой системы выбирается один свободный блок (при этом он удаляется из списка свободных блоков) и заполненный массив свободных блоков копируется в этот блок. После этого значение указателя НБ обнуляется, а в нулевой элемент массива записывается номер блока, который мы выбрали для копирования в него нашего массива. В итоге, при постоянном освобождении блоков образуется список, в котором будут размещены номера абсолютно всех свободных блоков файловой системы.

Вторая критическая ситуация - когда нужно получить свободный блок, но содержимое массива исчерпалось. В этом случае система действует так: если нулевой элемент списка равен нулю, то это означает, что исчерпалось все пространство файловой системы, и выдается сообщение об этом. Если он не равен нулю, то его содержимое равно адресу продолжения массива, и операционная система подчитывает соответствующий блок и размещает это продолжение на место массива в суперблоке.

Второй массив, который находится в суперблоке - это массив, состоящий из ста элементов и содержащий номера свободных индексных дескрипторов. Работа с этим массивом осуществляется просто. Пока есть место в этом массиве, то при освобождении индексных дескрипторов свободные индексные дескрипторы записываются на свободные места массива. Если массив заполнен полностью, то запись в этот массив прекращается. Если, наоборот, содержимое массива исчерпалось, то запускается процесс, который просматриваетобласть индексных дескрипторов и соответственно заполняет массив новыми значениями. Возможна ситуация, когда нужно создать файл, т.е. нужен новый индексный дескриптор, а в массиве нет ни одного элемента, и запущенный процесс также не нашел свободных индексных дескрипторов. Это вторая ситуация, когда система вынуждена заявить, что ресурс исчерпан (первая - когда заканчиваются свободные блоки файловой системы).

Лекция №8

**Индексные дескрипторы.** Индексные дескрипторы занимают несколько подряд идущих блоков на диске. Размер области индексных дескрипторов определяется параметром (который был определен при инсталляции системы), определяющим количество индексных дескрипторов для данной конкретной файловой системы. Размер области равен произведению этого количества на размер индексного дескриптора.

Индексный дескриптор - это объект UNIX-а, который ставится во взаимно однозначное соответствие с содержимым файла, за исключением случаев, когда файл есть некий специальный файл, ассоциированный с внешним устройством. В индексном дескрипторе существуют следующие поля:

1. Поле, определяющее тип файла (каталог или нет).
2. Поле кода защиты.
3. Количество ссылок к данному индексному дескриптору из всевозможных каталогов файловой системы (в ситуации нарушения дерева файловой системы). *Если значение этого поля равно нулю, то считается что этот индексный дескриптор свободен.*
4. Длина файла в байтах.
5. Статистика: поля, характеризующие дату и время создания и т.п.
6. Поле адресации блоков файлов.

Обратите внимание на то, что в индексном дескрипторе нет имени файла, хотя этот объект характеризует содержимое файла. Давайте посмотрим, как организована адресация блоков, в которых размещается содержимое файла.

В поле адресации находятся номера первых десяти блоков файлов. Если файл небольшой, то вся информация о размещении его блоков находится в индексном дескрипторе. Если файл превышает десять блоков, то начинает работать некая списочная структура. Одиннадцатый элемент поля адресации содержит номер блока из пространства блоков файлов, в котором размещены 128 ссылок на блоки данного файла.

В том случае, если файл еще больше, то используется двенадцатый элемент поля адресации. Он содержит номер блока, в котором содержится 128 записей о номерах блоков, содержащих по 128 номеров блоков файловой системы, т. е. здесь используется двойная косвенность. Если файл еще больше, то используется тринадцатый элемент и используется тройная косвенность (аналогично двойной, но добавляется еще один уровень). Предельный размер файла (при размере блока 512) будет равен (128 + 1282 + 1283) \* 512 байт = 1Гб + 8Мб + 64Кб > 1Гб.

Мы с вами договаривались, что в течение нашего курса мы будем обращать внимание на несоответствие скоростей и на сглаживание этого. Первая проблема - если бы файловая система не имела в суперблоке массива свободных блоков, то пришлось бы каким-либо образом каждый раз искать свободные блоки, и эта работа была бы сумасшедшей, и файловая система рухнула бы на идейном уровне. Аналогично со списком свободных индексных дескрипторов, хотя здесь поиск был бы проще, чем для свободных блоков, но тем не менее, здесь также есть элементы оптимизации. Косвенность в адресации блоков файлов позволяет нарастать накладным расходам по чтению блоков файлов соразмерно величине этого файла. То есть если файл маленький, то накладных расходов нет, потому что при открытии файла в оперативной памяти создается копия индексного дескриптора файла, и без дополнительных обращений к ВЗУ можно добраться к любому из десяти блоков файла сразу же. Если необходимо работать с блоками, размещенными на первом уровне косвенной адресации, то появляется один дополнительный обмен, но при этом доступ можно осуществить уже к 128-и блокам. Аналогичные рассуждения и для блоков второго и третьего порядка. Казалось бы, плохо то, что при обмене с большим файлом приходиться осуществлять множество дополнительных обменов, однако система UNIX хитрая - она использует глубокую эшелонированную буферизацию обменов с ВЗУ. То есть если мы и получаем некоторые накладные расходы на одном уровне, то они компенсируются на другом уровне оптимизации взаимодействия системы с внешней памятью.

**Блоки файлов.** Размер пространства блоков файлов определен однозначным образом за счет информации в суперблоке.

**Область сохранения процессов.** Хотя эта область изображена за блоками файлов, но она может быть размещена и в некотором файле файловой системы или на произвольном месте других ВЗУ. Это зависит от конкретной реализации системы. По сути, эта область является полезной областью, в которую происходит откачка процессов, а также эта область используется для оптимизации запуска наиболее часто используемых процессов с использованием, так называемого, t-бита файла (подробнее об этом будет рассказано позже).

Итак, мы с вами рассмотрели структуру файловой системы и ее организацию на системном устройстве. Как любая системная конструкция, структура файловой системы и связанные с ней алгоритмы работы просты настолько, чтобы при работе с ними накладные расходы не выходили за пределы разумного. Файловая система UNIX-а при реальной работе заведомо оптимальнее файловой системы Windows NT (сравните даты разработок!!!), за счет простоты и оптимизации, которая встречается на каждом шагу.

## Каталоги

Мы с вами говорили, что одним из свойств операционной системы UNIX является то, что вся информация размещается в файлах, т.е. нет каких-то специальных таблиц, которыми пользуется операционная система, за исключением тех таблиц, которые она создает, уже функционируя в пространстве оперативной памяти. Каталог, с точки зрения файловой системы, - это файл, в котором размещены данные о тех файлах, которые принадлежат каталогу.

В каталоге А содержаться файлы В, С и D, несмотря на то, что файлы В и С могут быть как файлами, так каталогами, а файл D является каталогом.

A

B

D

C

F

G

Каталог состоит из элементов, которые содержат два поля. Первое поле - номер индексного дескриптора, второе поле - это имя файла, которое ассоциировано с данным индексным дескриптором. Номера индексных дескрипторов (в пространстве индексных дескрипторов) начинаются с единицы. Первый индексный дескриптор - индексный дескриптор каталога. В общем случае в каталоге могут встречаться записи, ссылающиеся на один и тот же индексный дескриптор, но в каталоге не могут быть записи, имеющие одинаковые имена. Имя в пределах каталога уникально, но с содержимым файла может ассоциироваться произвольное количество имен. Поэтому есть некоторая неоднозначность в определении понятия файл в операционной системе UNIX. Файл оказывается не просто именованным набором данных: у него есть индексный дескриптор и может быть несколько имен (т.е. имя - вторичная компонента).

При создании каталога в нем всегда создаются две записи: запись на специальный файл с именем «.» (точка), с которым ассоциирован индексный дескриптор самого каталога, и файл «**..**» (две точки), с которым ассоциируется индексный дескриптор (ИД) родительского каталога. Для нашего примера каталог А имеет, например, ИД с номером 7, а каталог D имеет ИД с номером 5. Файл F имеет ИД №10, файл G имеет ИД №101. В этом случае файл-каталог D будет иметь следующее содержимое:

|  |  |  |
| --- | --- | --- |
| Имя | №ИД | |
| «**.**» | | 5 | Первая запись - запись на самого себя. |
| «**..**» | | 7 | Вторая запись - на родителя (каталог А). |
| «F» | | 10 | Далее перечислены файлы, которые находятся в этом каталоге. |
| «G» | | 101 | Вот таким будет содержимое каталога D. |

Отличие файла-каталога от обычных файлов пользователя заключается в содержимом поля типа файла в ИД. Для корневого каталога поле родителя будет ссылаться на него самого.

Теперь схематически рассмотрим, как могут использоваться полные имена и структура каталогов. В системе, в каждый момент времени работы пользователя определен текущий каталог, то есть каталог и весь путь от корня, связанный с этим каталогом, который по умолчанию подставляется ко всем именам файлов, не начинающихся с символа «/». Если текущий каталог D, то можно говорить просто о файлах F и G, а если надо добраться до файла В, то необходимо использовать полное имя или специальный файл «**..**», т.е., в данном случае, конструкцию «**../**В». Мы ссылаемся на файл «**..**» - это означает, что нужно прочесть ИД родителя и по нему добраться до содержимого каталога А. Затем в файле-каталоге А надо выбрать строку с именем В и определить ИД файла В, а затем произвести открытие файла. Вся эта операция довольно трудоемка, однако учитывая то, что файлы открываются не часто, это не будет сказываться на скорости работы системы.

Мы говорили, что с одним и тем же содержимым может ассоциироваться несколько имен, т.е. одновременно могут быть открыты файлы с одним и тем же ИД. Возникает проблема - как синхронизируется работа с содержимым файла в случае его открытия разными процессами или с разными именами. В UNIX-е это решается достаточно корректно (это мы рассмотрим несколько позже).

## Специальные файлы устройств

Мы уже знаем два типа файлов: файлы-каталоги и рабочие файлы, в которых хранятся данные. Есть третья разновидность - файлы устройств. Эта разновидность характеризуется типом, указанным в ИД. Содержимого у файлов устройств нет, а есть только ИД и имя. В ИД указывается информация о том, какой тип устройства ассоциирован с этим файлом: байт-ориентированное устройство или блок-ориентированное устройство. Байт-ориентированное устройство - это то устройство, обмен с которым осуществляется по одному байту (например, клавиатура). Блок-ориентированное устройство - это устройство, с которым обмен может осуществляться блоками.

Также имеется поле, определяющее номер драйвера, связанного с этим устройством (у одного устройства может быть несколько драйверов, но не наоборот). Это поле, на самом деле, есть номер в таблице драйверов соответствующего класса устройств. В системе имеются две таблицы: для блок- и для байт-ориентированных устройств. Также в ИД определен некоторый цифровой параметр, который может быть передан драйверу в качестве уточняющего информацию о работе.

## Организация обмена данными с файлами

Определим сначала, что является низкоуровневым вводом/выводом в системе. В файловой системе UNIX-а определены некоторые специальные функции, которые называются системными вызовами. Системные вызовы осуществляют непосредственное обращение к операционной системе, то есть это функции, выполняющие некоторые действия операционной системы. Реализация системных и библиотечных функций (например, математических) в корне отличается. Если библиотечная функция будет подгружена в тело процесса, который пользуется этой библиотекой, то все действия в большинстве случаев будут выполняться в пределах этого процесса, а системный вызов сразу же передает управление операционной системе и она выполняет заказанное действие. В UNIX-е для обеспечения низкоуровнего ввода/вывода, т.е. ввода/вывода, который реализуется посредством системных вызовов, имеется набор функций. Вот основные из них:

1. **open** - Открытие имеющегося файла. Одним из параметров этой функции является строка с именем файла, а возвращает она некоторое число, которое называется дескриптором файла. В теле процесса пользователя, а также в данных, ассоциированных с этим процессом, размещается (кроме кода и данных, разумеется) некоторая служебная информация, в частности, таблица файловых дескрипторов. Она, как и все таблицы в системе UNIX, позиционная, т.е. номер дескриптора соответствует номеру записи в этой таблице. С файловым дескриптором (ФД) ассоциировано имя файла и все необходимые атрибуты для работы с ним. Номера ФД уникальны в пределах одного процесса. Есть аналогичная функция **create** - функция открытия нового файла.
2. **read/write** - системные вызовы чтения/записи, параметрами которых является номер ФД и некоторые атрибуты, которые не так важны для нашего рассмотрения.
3. **close -** системный вызов завершения работы с файлом, параметром которого является номер ФД. После обращения к этой функции ФД становится свободным, а работа данного процесса с файлом завершается.

Вот некоторые системные вызовы, обеспечивающие ввод/вывод (кстати, они почти не добавляют кода к вашей программе). Подробности посмотрите самостоятельно. Я обратил ваше внимание, что это системные вызовы, потому что ввод/вывод можно осуществлять и через библиотеки ввода/вывода. Для этого существует, так называемый, файловый обмен и функции fopen, fread, и т.д. (с префиксом f). Это библиотечные функции. Эти функции сами обращаются к низкоуровневым функциям внутри себя.

Рассмотрим организацию обмена с системной точки зрения в операционной системе UNIX. При организации обмена операционная система подразделяет данные на две категории: данные, ассоциированные с процессом пользователя, и данные, ассоциированные с операционной системой.

**Таблица индексных дескрипторов открытых файлов**. Первая таблица данных, ассоциированных с операционной системой, - таблица индексных дескрипторов открытых файлов (ТИДОФ). Эта таблица содержит записи, каждая из которых содержит копию индексного дескриптора для каждого открытого в системе файла. Через эту копию осуществляется доступ к блокам файлов. Каждая записей таблицы содержит также поле, характеризующее количество открытых в системе файлов, использующих данный дескриптор (счетчик). То есть, если один и тот же файл открыт от имени двух процессов, то запись в ТИДОФ создается одна, но каждое дополнительное открытие этого файла увеличивает счетчик на единицу.

**Таблица файлов.** Таблица файлов (ТФ) содержит информацию об имени открытого файла, и имеет ссылку на ТИДОФ.

Лекция №9

Мы говорили, что система может работать с содержимым файла в том и только том случае, если процесс зарегистрировал свое желание работать с этим файлом. Факт такой регистрации называется открытием файла. При открытии файла в пределах процесса каждому имени открываемого файла (открываться может уже существующий файл, либо новый) ставится в соответствие уникальное целое число, которое называется файловым дескриптором (ФД). В пределах процесса ФД имеют нумерацию от 0 до k-1. Значение k - это параметр настройки операционной системы, определяющий, какое количество одновременно открытых файлов может быть у процесса. Здесь следует отметить, что мы говорим о количестве одновременно открытых файлов (так же написано в любой книжке по UNIX-у), однако, на самом деле, k - это максимальное количество ФД, которые могут быть ассоциированы с одним файлом, потому что один и тот же файл в пределах процесса можно открыть два раза, и образуется два ФД. К чему это приведет, мы рассмотрим несколько позже, но это вполне корректно. После открытия файла, все операции обмена осуществляются через указания файлового дескриптора (т.е. имя более нигде не указывается). С каждым файловым дескриптором ассоциирован ряд параметров (о них чуть позже).

Давайте посмотрим, как организуется ввод/вывод, а точнее обработка низкоуровнего обмена, с точки зрения операционной системы. Сейчас будет рассказано о логической схеме организации ввода/вывода, ибо реальная схема устроена несколько иначе, но это для нас не так важно.

Все данные, с которыми оперирует система, подразделяются на два класса. Первый тип данных - данные, ассоциированные с операционной системой, то есть общесистемные данные. К этим данным относится ТИДОФ. Размер таблицы фиксирован и определяется количеством одновременно открытых ФД. Каждая запись в этой таблице содержит некоторую информацию, среди которой нас будет интересовать следующая:

1. Копия ИД открытого файла. Для любого открытого файла, ИД, который характеризует содержимое этого файла, копируется и размещается в ТИДОФ. После этого все манипуляции с файлом (например, изменение адресации файла) происходят с копией ИД, а не с самим ИД на диске. ТИДОФ размещается в оперативной памяти, т.е. доступ к информации в ней осуществляется быстро.
2. Счетчик открытых в данный момент файлов, связанных с данным ИД. Это означает, что для любого количества открытий файла, связанного с данным ИД, система работает с единственной копией этого ИД.

Теперь перейдем к, так называемой, таблице файлов (ТФ). Таблица файлов состоит из фиксированного количества записей. Каждая запись ТФ соответствует открытому в системе файлу {{или точнее ФД}}. При этом в подавляющем большинстве случаев это есть взаимно однозначное соответствие. Тот случай, когда это не есть взаимно однозначное соответствие, мы рассмотрим ниже. Каждая запись ТФ содержит указатели чтения/записи по файлу. Это означает, что, если открыт один и тот же файл в двух процессах или дважды в одном процессе, то с каждым открытием связан свой указатель, и они друг от друга не зависят (почти всегда, за исключением некоторых случаев). Каждая запись ТФ содержит, так называемый, индекс наследственности - это есть некоторое целое число.

Это данные уровня операционной системы, т.е. данные, которые описывают состояние проблемы в системе в целом.

С каждым процессом связана, так называемая, таблица открытых файлов (ТОФ). Номер записи в данной таблице есть номер ФД. Каждая строка этой таблицы имеет ссылку на соответствующую строку ТФ. Это означает, что информация об указателях, связанных с ФД, как бы разорвана. С одной стороны, файловые дескрипторы - это данные, являющиеся атрибутом процесса, с другой стороны, указатель - это данные, являющиеся атрибутом операционной системы. Казалось бы, не логично, и сейчас мы рассмотрим, в чем эта нелогичность проявляется. Для этого кратко рассмотрим концептуальные вопросы, связанные с формированием процесса. Операционная система UNIX имеет функцию **fork()**. Это системный вызов. При обращении к этому системному вызову в системе происходит некоторое действие, которое для большинства из вас может показаться бессмысленным, - происходит копирование процесса, в котором встретилась эта функция, т.е. создается процесс-двойник. Для чего это нужно, я скажу несколько позже.

Формирование процесса-двойника обладает следующими свойствами. Первое свойство: процесс-сын, который будет сформирован после обращения к функции **fork(),**  имеет все те файлы, которые были открыты в процессе-отце. Второе - система позволяет некоторыми своими средствами, идентифицировать, где процесс-отец, а где процесс-сын, хотя в общем случае они абсолютно одинаковы.

Предположим, есть процесс №1, и с ним ассоциирована таблица открытых файлов №1. В этом процессе открыт файл с именем Name, и этому файлу поставлен в соответствие файловый дескриптор I. Это означает, что в соответствующей строке ТОФ будет запись, имеющая ссылку на ТФ. В ТФ определены какие-то атрибуты, связанные с открытием файла, а также имеется указатель чтения/записи, т.е. тот указатель, по которому мы работаем, обмениваясь информацией с файлом. Записи в ТФ имеют ссылку на ТИДОФ, в которой находится копия ИД, соответствующего файлу с именем Name.

Предположим, что в этом процессе еще раз открыт файл с именем Name. Система поставила ему в соответствие файловый дескриптор J. Т.е. этому открытию соответствует J-тая строка ТОФ первого процесса. В этой записи будет ссылка на запись ТФ, которая поставлена в соответствие второму открытию файла Name. И пока индексы наследственности для обоих случаев будут равны единице. В этой записи будут свои, связанные с этим открытием, указатели чтения/записи. Указатели файловых дескрипторов I и J независимы друг от друга, т.е. при чтении/записи через файловый дескриптор I, указатель файлового дескриптора J не изменится. Эта запись будет ссылаться на тот же самый индексный дескриптор из ТИДОФ, и значение счетчика будет равно двум.

ТОФ №1

Name

I: I:

Name

J:

rw

rw

ТФ

ТИДОФ

1

1

2

Блоки

Счетчик

Индекс Наследственности

Предположим, процесс №1 выполнил обращение к функции **fork()**, образовалась копия процесса, причем, обе копии начинают работать на выходе из **fork()**, и со вторым процессом будет ассоциирована ТОФ №2. Так же будет открыт файл Name по ИД I и по ИД J. Но в этом случае, когда процесс получил открытые файлы в наследство от родителя, то ссылки из соответствующих строк ТОФ будут происходить не на новые записи ТФ, а на те же самые, к которым ссылались соответствующие ФД у родителя. У этих процессов указатели чтения записи будут одинаковы, т.е. если передвинуть указатель в одном процессе, то он автоматически передвинется и для другого процесса. Этот случай, как раз тот, когда нет взаимно однозначного соответствия между строками ТФ и строками ТОФ. При порождении этих ссылок счетчик увеличивается на два. И, соответственно, из ИД, за счет адресации блоков, осуществляется доступ к блокам файлов. Такая информационная организация обмена означает, что обмен с содержимым каждого файла осуществляется централизованно, т.е. в конечном итоге, все заказы на обмен идут через одну единственную запись, сколько бы файлов, связанных с этим ИД, не было открыто в системе. Здесь нет никаких коллизий, когда во времени начинается путаница в выполненных, или невыполненных обменах, связанных с одним дескриптором.

ТОФ №1

Name

I: I:

Name

J:

ТОФ №2

Name

I: I

Name

J: J:

rw

rw

ТФ

ТИДОФ

2

2

4

Блоки

При любом формировании нового процесса, система априори устанавливает нулевой, первый и второй файловые дескрипторы из ТОФ, связывая их с предопределенными файлами. Нулевой ФД связан с системным файлом ввода, с ним обычно ассоциировано внешнее устройство клавиатура. Первый ФД - это стандартный файл вывода, обычно с ним ассоциирован экран монитора. Второй ФД - это стандартный файл вывода диагностических сообщений, с ним также обычно ассоциирован экран монитора.

Рассмотрим для примера типовые действия при обращении к тем или иным системным вызовам.

**Обращение к функции fork()**. Как известно, при обращении к этой функции система создает копию исходного процесса. При этом система дублирует ТОФ одного процесса в ТОФ процесса-наследника, а также увеличивает на единицу индекс наследственности в строках ТФ, ассоциированных с открытыми файлами исходного процесс, а также увеличивает счетчик открытых файлов, связанных с данным ИД, в ТИДОФ.

**Обращение к функции open().** При обращении к этой функции происходит следующее:

1. По полному имени определяется каталог, в котором размещен файл.
2. Определяется номер ИД. По номеру ИД осуществляется поиск в таблице ТИДОФ.
3. Если запись с заданным номером обнаружена, фиксируем номер соответствующей строки ТИДОФ и переходим к шагу 5.
4. В случае если строка не обнаружена, происходит формирование новой строки, соответствующей новому ИД и фиксируется ее номер.
5. Корректируем счетчик ссылок (стрелок) на запись ТИДОФ. Номер записи в ТИДОФ записывается в запись ТФ, а также в ТОФ устанавливается ссылка на соответствующую запись ТФ. После этого в программу возвращается номер сроки ТОФ, в которой находится ссылка на запись в ТФ.

При операциях ввода/вывода действия системы очевидны.

**Взаимодействие с устройствами.** Мы уже говорили, что все устройства, которые обслуживаются операционной системой UNIX, могут быть классифицированы на два типа - байт-ориентированные устройства и блок-ориентированные устройства. Следует отметить, что одно и то же устройство в системе может рассматриваться и как байт-ориентированное, и как блок-ориентированное (пример - оперативная память). Соответственно, есть драйверы блок-ориентированные и байт-ориентированные. На прошлой лекции мы рассматривали специальные файлы, ассоциированные с внешними устройствами, и говорили о том, что есть таблица драйверов блок-ориентированных устройств и таблица драйверов байт-ориентированных устройств. Соответственно, на эти таблицы имеются ссылки в ИД специальных файлов.

Основной особенностью организации работы с блок-ориентированными устройствами является возможность буферизации обмена. Суть заключается в следующем. В оперативной памяти системы организован пул буферов, где каждый буфер имеет размер в один блок. Каждый из этих блоков может быть ассоциирован с драйвером одного из физических блок-ориентированных устройств.

Рассмотрим, как выполняется последовательность действий при исполнении заказа на чтение блока. Будем считать, что поступил заказ на чтение N-ого блока из устройства с номером M.

1. Среди буферов буферного пула осуществляется поиск заданного блока, т.е. если обнаружен буфер, содержащий N-ый блок М-ого устройства, то фиксируем номер этого буфера. В этом случае, обращение к реальному физическому устройству не происходит, а операция чтения информации является представлением информации из найденного буфера. Переходим на шаг 4.
2. Если поиск заданного буфера неудачен, то в буферном пуле осуществляется поиск буфера для чтения и размещения данного блока. Если есть свободный буфер (реально, эта ситуация возможна только при старте системы), то фиксируем его номер и переходим к шагу 3. Если свободного буфера не нашли, то мы выбираем буфер, к которому не было обращений самое долгое время. В случае если в буфере имеется установленный признак произведенной записи информации в буфер, то происходит реальная запись размещенного в буфере блока на физической устройство. Затем фиксируем его номер и также переходим к пункту 3.
3. Осуществляется чтение N-ого блока устройства М в найденный буфер.
4. Происходит обнуление счетчика времени в данном буфере и увеличение на единицу счетчиков в других буферах.
5. Передаем в качестве результата чтения содержимое данного буфера.

Вы видите, что здесь есть оптимизация, связанная с минимизацией реальных обращений к физическому устройству. Это достаточно полезно при работе системы. Запись блоков осуществляется по аналогичной схеме. Таким образом организована буферизация при низкоуровневом вводе/выводе. Преимущества очевидны. Недостатком является то, что система в этом случае является критичной к несанкционированным выключениям питания, т.е. ситуация, когда буфера системы не выгружены, а происходит нештатное прекращение выполнения программ операционной системы, что может привести к потере информации.

Второй недостаток заключается в том, что за счет буферизации разорваны во времени факт обращения к системе за обменом и реальный обмен. Этот недостаток проявляется в случае, если при реальном физическом обмене происходит сбой. Т.е. необходимо, предположим, записать блок, он записывается в буфер, и получен ответ от системы, что обмен закончился успешно, но когда система реально запишет этот блок на ВЗУ, неизвестно. При этом может возникнуть нештатная ситуация, связанная с тем, что запись может не пройти, предположим, из-за дефектов носителя. Получается ситуация, при которой обращение к системе за функцией обмена для процесса прошло успешно (процесс получил ответ, что все записано), а, на самом деле, обмен не прошел.

Таким образом, эта система рассчитана на надежную аппаратуру и на корректные профессиональные условия эксплуатации. Для борьбы с вероятностью потери информации при появлении нештатных ситуаций, система достаточно «умна», и действует верно. А именно, в системе имеется некоторый параметр, который может оперативно меняться, который определяет периоды времени, через которые осуществляется сброс системных данных. Второе - имеется команда, которая может быть доступна пользователю, - команда SYNC. По этой команде осуществляется сброс данных на диск. И третье - система обладает некоторой избыточностью, позволяющей в случае потери информации, произвести набор действий, которые информацию восстановят или спорные блоки, которые не удалось идентифицировать по принадлежности к файлу, будут записаны в определенное место файловой системы. В этом месте их можно попытаться проанализировать и восстановить вручную, либо что-то потерять. Наш университет одним из первых в стране начал эксплуатировать операционную систему UNIX, и сейчас уже можно сказать, что проблем ненадежности системы, с точки зрения фатальной потери информации, не было.

Сегодня мы начинали разговор о том, что у нас есть системные вызовы и библиотеки ввода/вывода. Еще одно средство, которое позволяет оптимизировать работу системы, - это стандартная библиотека ввода/вывода, связанная с include-файлом stdio.h. Суть концептуального обмена та же самая, что и при организации низкоуровневого ввода/вывода. Разница в том, что, если open() возвращает номер файлового дескриптора, fopen() возвращает указатель на некоторую структуру специального типа FILE. Второе и основное - это библиотека функций. Многие функции сервиса, которые предоставляет эта библиотека, реализуются в пределах вашего адресного пространства. В частности, такой функцией сервиса является еще один уровень буферизации ввода/вывода. Суть его заключается в том, что на ресурсах процесса можно выделить буфер, который будет работать аналогично буферному пулу операционной системы и, который минимизирует обращение вашего процесса к системным вызовам ввода/вывода. Понятно, что эта библиотека ввода/вывода реализуется посредством программы, использующей системные вызовы ввода/вывода. Это означает, что появляется фактор двойной буферизации, хотя это увеличивает ненадежность.

Двойная буферизация, очевидно, вещь полезная. Она позволяет обращаться к чтению или записи через библиотечные функции объемами данных в полблока или в треть блока, и если эти части идут подряд, то система сама за счет буферизации, собирает эти части и вместо нескольких обращений к системному вызову выполняет только одно обращение. Это выгодно. Невыгодно то, что эта буферизация организуется в пределах адресного пространства процесса со всеми вытекающими последствиями (теряется синхронизация по обменам в том случае, если с данным файлом через эту библиотеку работают другие процессы, потому что в теле каждого процесса есть свой буфер, который может аккумулировать эти данные и никакого единообразия, которое есть в рассмотренной нами схеме, не получается). Тем не менее, стандартная библиотека ввода/вывода есть удобный инструмент; она имеет также средства блокировки этой буферизации.

Лекция №10

На прошлой лекции мы разобрали следующие моменты, связанные с организацией функционирования файловой системы. Это системная организация низкоуровнего обмена. Мы выяснили, что за счет организации данных, операционная система UNIX достаточно простыми и «прозрачными» средствами решает проблемы возможных конфликтов в случае нескольких открытий одного и того же файла. Мы видели, что все открытия одного и того же файла (под файлом мы понимаем не имя, а содержимое) в конечном итоге сводятся к работе с единственной копией ИД. Мы с вами выяснили, что почти все открытия файлов, связанные с одним ИД, порождают для процессов возможность работать со своими указателями чтения/записи по файлом, за исключением случаев, когда файл в процессе был получен через наследование, т.е. файл был получен от процесса-отца через функцию **fork()** процессом-сыном.

Мы с вами выяснили, что система подразделяет обслуживаемые устройства на два класса: блок-ориентированные и байт-ориентированные. Одно и то же устройство может одновременно быть и байт-ориентированным, и блок-ориентированным. Это зависит как от самого устройства, так и от наличия драйверов - программ управляющих этим устройством. Примером такого устройства является оперативная память.

Мы с вами рассмотрели принципы организации низкоуровнего обмена для блок-ориентированных устройств, и в контексте этого мы познакомились со средствами буферизации, которые применяются в ОС UNIX, суть которых заключается в том, что по аналогии с буферами чтения/записи из оперативной памяти (аппаратным средством), операционная система создает программные средства, которые позволяют минимизировать количество обращений к физическому устройству. Этот механизм выгодно отличал и отличает ОС UNIX. Здесь следует заметить, что буферизация обмена может быть многоуровневой. Первый дополнительный уровень может появиться за счет того, что устройство может иметь свои аппаратные буфера, реализованные по аналогии с буферами оперативной памяти.

Мы также говорили о том, что кроме низкоуровнего ввода/вывода, с которым связаны функции, обеспечивающие системные вызовы (open, read, write и т.д.), существуют высокоуровневые средства доступа - это стандартная библиотека ввода/вывода stdio.h, подключение которой позволяет использовать для организации обменов еще один уровень буферизации (это оптимизация обращений к системным вызовам), который ассоциирован с процессом, т.е. буферизация происходит за счет ресурсов процесса. Мы оценивали, что хорошо, что плохо. Очевидно, что буферизация сокращает количество обменов с медленным внешним устройством, и чем больше таких уровней, тем меньше происходит обменов. Однако плохо то, что за счет буферизации снижается надежность системы. Например, при неожиданном для системы выключении питания, все буфера теряют информацию. Момент обращения к обмену далек от реального обмена, и поэтому возможны неприятные ситуации. Но, несмотря на эти недостатки, опыт показывает, что фатальные потери информации происходят крайне редко.

Обращаю ваше внимание, насколько UNIX экономит обращения к ВЗУ. Суперблок находится в оперативной памяти, и реальные действия с информацией суперблока происходят не с диска, а из оперативной памяти, хотя здесь возникает та же проблема с несанкционированным выключением питания.

Открывая файл, мы работаем с ИД. Мы выяснили, что работа с ИД осуществляется через работу с его копией, размещенной в программных таблицах в оперативной памяти. Это означает, что почти нет накладных расходов, связанных с маленькими файлами, и эти накладные расходы малы при работе с огромными файлами. Получается так, что почти вся инфраструктура, поддерживающая работу файловой системы, работает за счет глубокой эшелонированной буферизации.

***Атрибуты файлов***

Мы с вами говорили об организации пользователей системы; она имеет иерархическую трехуровневую структуру.

Любой пользователь принадлежит к группе. В соответствии с иерархией пользователей, определена иерархия защиты файлов и прав пользователей. Определено понятие владельца файла. Изначально владельцем файла является пользователь (а точнее, процесс пользователя), создавший этот файл. Атрибут «владелец файла» может быть изменен командой **changeown**. Каждый файл имеет атрибуты защиты, связанные с иерархией. Есть права доступа к некоторым действиям файла со стороны владельца файла. Это права на чтение, на запись, на исполнение. У каждого файла, кроме прав, связанных с уровнем пользователя, имеются права, связанные с уровнем группы. Это права для всех пользователей группы, к которой принадлежит владелец файла, за исключением его самого (т.е. права владельца и его группы различны). Третья категория защиты - все остальные. Это те права, которые имеют все пользователи системы, за исключением владельца и его группы. В системе имеется команда изменения прав доступа **changemode**.

Кроме атрибутов доступа, каждый файл может иметь признаки, в частности, т.н. t-бит и s-бит, которые также устанавливаются некоторой командой. Мы, уже зная структуру файловой системы, понимаем, что в принципе файл может находиться в очень сильно фрагментированном виде. Кроме того, файл может быть большим, а при открытии большого файла, возникают накладные расходы, связанные с доступом к далеким блокам файла. Поэтому открытие файла - это длительный процесс. Чтобы оптимизировать это действие, в системе имеется возможность пометить исполняемые файлы t-битом. После этого происходит следующее: в том случае, если вызывается исполняемый файл, помеченный t-битом, то при первом вызове за сеанс работы системы происходит копирование тела файла в область сохранения. При каждом повторном вызове файла, сначала происходит просмотр каталога области сохранения, и в том случае, если искомый файл там есть, то загрузка файла происходит не с ВЗУ, а из этой области. То есть это еще один путь минимизации обращений к ВЗУ. Обычно возможность установки t-бита - это прерогатива системного администратора, и системный администратор сам выбирает те процессы (и соответственно, файлы), которые надо пометить t-битом. Обычно им помечаются те процессы, которые используются наиболее часто (если, например, идет практикум, то t-битом имеет смысл пометить файл компилятора).

S-бит мы рассмотрим несколько поверхностно, но вернемся к нему позже. Есть следующая проблема. Все средства системы принадлежат кому-то, т.к. все средства, все команды (за исключением некоторых встроенных) в конечном счете представляют из себя файлы, у которых имеется владелец. Какие-то из этих команд могут обращаться в те или иные системные файлы. Возникает проблема, связанная с тем, что с одной стороны, должна быть защита от несанкционированного доступа к файлу. С другой стороны, все команды имеют потенциальные права для всех категорий. Как быть? Есть возможность помечать некоторые файлы s-битом. Владелец файла с s-битом остается неизменным, но при запуске этого файла, владельцу процесса запустившего этот файл, предоставляются права по доступу к данным от владельца исходного файла.

Предположим, есть исполняемый файл с именем file, и он работает каким-то образом с файлом file2, в котором находится конфиденциальная информация. Предположим, file корректирует file2, в котором находится информация обо всех зарегистрированных пользователях и, в частности, file может менять пароль пользователя в системе. Если я запущу file от своего имени, то могут возникнуть две ситуации: либо я не смогу работать с file2, в котором есть учетная информация о пользователях, потому что он закрыт для всех остальных; либо он открыт для всех, тогда нет никакой защиты. В этом случае работает s-бит. Суть его работы заключается в следующем. Владельцем исходного файла является пользователь ROOT. Предположим, этот файл захотел запустить пользователь с именем MASH. Если MASH запускает этот файл и нет s-бита, то получается, что владельцем файла является ROOT, а владельцем процесса стал MASH. В этом случае, файлы, которые недоступны пользователю MASH, будут недоступны и его процессу, и MASH не сможет изменить свой пароль в системе. S-бит позволяет продлить права владельца (ROOT) файла на владельца (MASH) процесса (запущенного из этого файла), и на время сеанса работы процесса ему будут доступны все те файлы, которые были доступны владельцу файла (ROOT).

Следующий вопрос: как интерпретируются права доступа к каталогам (поскольку каталоги также являются файлами)? Разрешение на чтение из каталога означает, что разрешен вход в каталог и открытие файлов из этого каталога. Разрешение на запись предоставляет возможность создавать и уничтожать файлы в этом каталоге. Разрешение на исполнение - это возможность поиска в данном каталоге (например, с помощью команды ls).

***Файловая система с точки зрения пользователя.***

Давайте рассмотрим структуру файловой системы с точки зрения пользователя. Эта структура будет рассматриваться для обобщенной операционной системы, так как реальная ее структура может варьироваться.

В корневом каталоге есть файл с именем **unix**. Это тот самый файл, который запускается программным загрузчиком, и который формирует ядро системы.

***Каталог ETC***. В этом каталоге находятся стандартные файлы данных системы и команды, обеспечивающие некоторый уровень управления функционированием системы. 1. *Файл* ***passwd***. Все пользователи в системе зарегистрированы через этот файл. Это означает, что если пользователь может работать, то в файле **passwd** имеется строка, помеченная именем пользователя, которая содержит набор некоторых данных, разделенных символом разделителя. В частности, строка файла **passwd** содержит номер группы, к которой принадлежит пользователь, иногда может содержать закодированный пароль на вход пользователя в системе. Закодированный - означает то, что в системе используется взаимно неоднозначная возможность отображения последовательности символов в некоторый код, и в системе хранится отображение этого пароля. Современные UNIX-ы хранят пароли в отдельной защищенной базе данных (хотя файл **passwd** тоже присутствует), потому что файл **passwd** обычно открыт на чтение, алгоритм преобразования тоже обычно известен и есть возможность подобрать пароль.

Далее, строка содержит поле, в котором должен быть атрибут, характеризующий фамилию, имя и отчество пользователя; поле, в котором указывается статус пользователя; поле, в котором указан «домашний» каталог. В этой же строке указано (или может быть указано) с каким интерпретатором команд этот пользователь будет работать. Могут быть еще некоторые параметры.

2. *Файл* ***rc***. В этом файле в текстовом виде находится набор команд, которые будут выполнены при загрузке операционной системы. Например, при загрузке, операционная система может запускать процесс проверки сохранности файловой системы.

Обращаю ваше внимание, что операционная система UNIX, за исключением нескольких случаев, содержит всю свою системную информацию в обыкновенных текстовых файлах. Эта информация легко просматривается и легко корректируется. В свое время, это был революционный шаг.

3. В этом же каталоге находятся команды, которые позволяют изменять пароли пользователя (исполняемый файл passwd), позволяют «примонтировать» к файловой системе локальные файловые системы и отбазировать эти же локальные системы, позволяют запускать процесс тестирования и коррекции файловой системы. Этот процесс проверяет файловую систему по некоторому набору признаков, например, множество свободных файлов должно при объединении с множеством занятых файлов давать все множество файлов. И так далее.

**Каталог BIN**. В этом каталоге находится подавляющее число стандартных команд системы, доступных пользователю.

**Каталог MNT**. Это каталог, к которому можно «примонтировать» локальные файловые системы. До сегодняшнего дня мы считали, что файловая система размещена на одном устройстве, но реально это не так. Имеется основная файловая система на системном устройстве, и имеется произвольное (в разумных пределах) количество локальных файловых систем, которые монтируются к системе с помощью некоторой команды. Корнем локальной файловой системы будет каталог **MNT**.

**Каталог DER**. В этом каталоге размещаются файлы, ассоциированные с конкретными драйверами внешних устройств, например, драйверы консоли, линейной печати и т.д. Вы помните, что файлы, ассоциированные с драйверами внешних устройств, в ИД, который ассоциирован с их именем, имеют признак того, что это файл-устройство, и также имеют в ИД ссылки на соответствующие таблицы драйверов. Эти файлы не имеют содержимого.

**Каталог USR**. Этот каталог имеет подкаталог **LIB**, в котором обычно находятся библиотеки, реализующие некоторые групповые функции, предоставляемые пользователю, в т.ч. Си-компилятор с соответствующими библиотеками поддержки.

Также, здесь имеется подкаталог **BIN** (/USR/BIN), в котором размещаются администратором системы дополнительные «домотканные» команды, потому что их размещение в каталоге /BIN считается некорректным.

Подкаталог **INCLUDE**. Вы помните, как выглядит строка **include <stdio.h>**. Эта строка дает команду препроцессору Си взять файл из каталога /USR/INCLUDE. Этот каталог имеет свои подкаталоги, и для нас интересен подкаталог **SYS** (/USR/INCLUDE/SYS). В нем находятся include-файлы, ассоциированные с системными возможностями, в частности signal.h - это перечисление тех сигналов, которыми могут обмениваться два процесса.

Итак, мы закончили описание файловой системы, и можем заключить, что файловая система UNIX иерархическая, многопользовательская. Файловая система UNIX имеет глубокую, многоярусную буферизацию при обменах с реальными устройствами. Файловая система UNIX является информационной основой функционирования операционной системы. Это расширяемая файловая система, при этом сохраняется ее целостность, т.е. при этом всегда существует единственный путь от ее корня до любого узла (или листа). Файловая система UNIX, с точки зрения логической организации файлов, имеет свою понятную и прозрачную структуру. Это накладывает определенные условия на администрацию системы, т.к. имеются проблемы координации прав доступа к различным компонентам файловой системы, имеются проблемы размещения новой информации в пределах файловой системы.