Содержание по темам

2.7.2.11. Нижние половины.
2.7.2.12. Структура данных.
2.7.2.13. Время на таймер.
2.7.3. Процессы-демоны.
2.7.3.1. Понятие о демоны.
2.7.3.2. Основные демоны UNIX.
2.7.3.3 Примеры программы демону.
3. Средства, механизмы, подсистемы ОС.
3.1. Системы управления процессами.
3.1.1. Двухуровневая система управления процессами.
3.1.2. Уровень долгосрочного планирования.
3.1.3. Схема долгосрочного уровня планирования.
3.1.4. Уровень краткосрочного планирования.
3.1.5. Структуры данных процессов.
3.1.5.1. Состояние процессов в UNIX.
3.1.6. Особенности планировщика UNIX (Linux).
3.1.7. Вычисление значения goodness.
3.1.8 Дескрипторы процессов.
3.1.9. Схемы работы краткосрочного планировщика.
3.1.10. Взаимодействие между процессами в UNIX.
3.1.10.1. Каналы.
3.1.10.2.FIFO (Fi rst InFi rst Out).
3.1.10.3. Сообщение (очереди сообщений).
3.1.10.4. Семафоры. Общая теория.
3.1.10.4.1. Задачи синхронизации.
3.1.10.4.2. Архитектура и основные вопросы построения механизмов синхронизации.
3.1.10.4.3. Семофорна техника синхронизации и упорядочения процессов.
3.1.10.5. Семофоры в UNIX.
3.1.10.6. Разделяемую память. Теоретические основы.
3.1.10.7. Память разделяемой в UNIX.

3.2. Подсистема управления ведением / выводом.
3.2.1. Основные концепции, лежащие в основе подсистемы управления ведением / выводом в ОС UNIX.
3.2.2. Подсистема ввода / вывода системы UNIX.
3.2.2.1. Драйверы устройств.
3.2.2.1.1. Типы драйверов.
3.2.2.1.2. Базовая архитектура драйверов.
3.2.3. Блочные устройства.
3.2.4. Символьные устройства.
3.2.4.1. Интерфейс доступа низкого уровня.
3.2.4.2. Буферизация.
3.2.5. Архитектура терминального доступа.
3.2.5.1. Псевдотерминалы.
3.3. Система управления данными.
3.3.1. Файловая подсистема UNIX.
3.3.1.1. Базовая ФС System.
3.3.1.1.1. Суперблок.
3.3.1.1.2. Индексные дескрипторы.
3.3.1.1.3. имена файлов
3.3.1.1.4. Недостатки и ограничения.
3.3.2. ФС BSD UNIX.
3.3.2.1. Каталоги.

4. Построение подсистем ядра мультипрограммных ОС.
4.1. Организация виртуальной ОП.

4.2. Основные понятия и принципы виртуализации памяти.
4.3. Схема структоризации адресных пространств.
4.4. Основы логической организации виртуальной ОП.
4.5. Схемы функционирования виртуальбнои ОП.
4.5.1. Частотно страничная схема функционирования виртуальной ОП
4.5.2. Страничная (по требованию) схема функционирования виртуальной ОП
4.5.3. Сегментная схема функционирования виртуальной ОП
4.5.4. Сегментносторинкова схема функционирования виртуальной ОП
4.6. Четыре задачи управления виртуальной памятью
4.7. Задача размещения
4.8. Алгоритмы распределения адресного пространства ОП
4.9. Задача перемещения
4.9.1. Распределение адресного пространства архивного среды хранения
4.9.2. Особенности алгоритмов передачи данных
4.10. Задача преобразования адресов
4.10.1. Механизм преобразования адресов в виртуальные ОП на основе сигментную адресации

4.10.2 Механизм преобразования адреса виртуальной ОП на основе страничной по требованию, организации

4.10.3. Положительные вопрос уменьшения влияния обращение к ТС на время выполнения команды за счет использования регистровой памяти ограниченной емкости
4.10.4. механизм преобразования адрес в виртуальные ОП на основе сегментносторивнковоий организации
4.11. Задача замещения
4.11.1. стратегия замещения
4.11.2. Явление "пробуксовки". Способы уменьшения потерь
5. Принципы управления памятью в UNIX
5.1. Виртуальная и физическая память
5.1.1. сегменты
5.1.2. страничный механизм
5.2. Адресное пространство процессов
5.2. Управление памятью процесса
5.2.1. области
5.2.2. замещение страниц
5.3. Планирование выполнения процессе
5.3.1. Обработка прерываний таймера
5.3.2. отложенный вызов
5.3.3. алармы
5.3.4. контекст процесса
6. Архитектура виртуальной ФС
6.1. Виртуальные индексные дескрипторы
6.2. Монтаж ФС
6.3. трансляции имен
6.4. Доступ к ФС
6.5. файловая таблица
6.6. Блокировка доступа к файлу
6.7. буферный кэш
6.7.1. Внутренняя структура буферного кэша
6.7.2. Операции ввода / вывода
6.8. целостность ФС

Лекция 1

Введение

На сегодняшний день не существует ЭВМ, которая не имела или была оснащена якоюнебудь ОС. Независимо от класса современные ЭВМ в большинстве имеют ОС ориентированы на работу с различными накопителями, сетевыми картами и у большинства есть мультипрограммными.

Развитие вычислительной техники предполагает быструю смену чипов (ЭВМ), это автоматически предполагает изменчивость ОС.

Проблему в большинстве можно решить, если ввести определенный стандарт на ОС. Такие работы велись и ведутся до сих пор. Например, ОС UNIX является наиболее вероятной для назначения ее "стандартной ОС". Но в стандартизации есть свои недостатки. Это такие проблемы как при создании языков программирования "все для всех". В особенно тяжелом положении находятся студенты. Возникает вопрос: что и как изучать в области ОС? Этот вопрос отнюдь не праздный и носит методологический характер. В этом случае возникает вопрос: делать студента "узкоспециализированным" и поэтому изучать досконально только одну ОС или давать некоторые общие представления, принципы построения ОС, предусматривая дальнейшую детализацию на этапе практической деятельности будущего специалиста по конкретной системе?

По некоторым авторитетными специалистами [4], лучшим является последний подход. Базовой ОС при изучении на сегодняшний день можно (лучше) принять ОС LINUX.

Во-первых, это разновидность одной из старейших и наиболее совершенных ОС UNIX. Во-вторых, ОС LINUX имеет открытый код ядра и всех дополнительных системных и несистемных программ.

В-третьих, свободно распространяется и не является коммерческой. эти ее свойства позволяют рассматривать ОС полностью. Студенты могут экспериментировать с системой на уровне ядра и надстроек не имея страха перед законом лицензирования.

И самое главное, все современные ОС персональных ЭВМ, имеющих ОС от компании Microsoft есть ничто иное, как сокращенным и упрощенным вариантом ОС UNIX.

1. Принципы построения ОС

Исторически ОС разделяются на:

- системы, ориентированные на перфокарты;
- системы, ориентированные на магнитные ленты;
- системы пакетной обработки данных;
- мультипрограммный пакетный режим и Мультипрограммные системы.

На сегодняшний день в чистом виде ни один из этих принципов построения ОС не используется, но некоторые из них принципов могут быть обнаружены в различных ОС.

Согласно анализу ОС [4] можно дать следующее определение ОС. Операционная система - это упорядоченная последовательность системных управляющих программ вместе с нужными информационными массивами, которые предназначены для планирование выполнения пользовательских программ и управления всеми ресурсами вычислительной машины (программами, данными, аппаратурой, оператором и другими объектами распределяемых и руководствуются) с целью предоставления возможности пользователям эффективно (в некотором смысле) решать задачи, которые сформированы в терминах вычислительной системы.

1.1. Основные принципы построения ОС

ОС различают по назначению, функциями которые они выполняют, формами реализации. В этом смысле каждая ОС - это уникальная сложная программная система. Одновременно можно говорить о тождественном уравнении ОС в смысле использования некоторых общих принципов, которые положены в основу их разработки.

1.1.1. Частотный принцип

Наиболее общий принцип реализации системных программ. Основан на выделении в алгоритмах программ и в массивах обрабатываемых действий и данных по частоте использования. Для действий, которые часто встречаются при работе ОС, обеспечиваются условия их быстрого выполнения. Например, такие программные тексты постоянно находятся в оперативной памяти, активно поддерживаются специальными средствами; к данным, которые активно поддерживаются специальными средствами; к данным, к которым выполняется частое обращение, обеспечивается наиболее быстрый доступ. К тому же, как правило, "частые" операции пытаются делать и более короткими.

наиболее существенные последствия от применения частотного принципа - использование многоуровневого планирования при организации работы ОС.

На уровень долгосрочности планирования выносятся жидкие и "длинные" операций й управления деятельностью системы. Минимальным объектом управления на этом уровне есть программы без детализации особенностей их выполнения, например, задание от пользователя на трансляцию и редактирования его программы. на уровень краткосрочного планирования выносятся следующие часто используемых и "короткие" операции по обеспечению выполнения на данном уровне отдельных программ. Система инициализирует или прерывает использования программ, предоставляя или отбирая ресурсы, которые требуются динамично и, прежде всего, центральный процессор.

1.1.2. принцип модульности

Этот принцип в уровне степени отражает технологические и эксплуатационные свойства. При этом наибольший эффект от его использования достигается в случае, когда принцип распространен одновременно на ОС, положительные программы и аппаратуру.

Под модулем в общем случае понимают функциональный элемент системы рассматриваемого имеющий оформлены, законченные и выполнены в рамках требований системы, и средства сопряжения с подобными элементами или элементами более высокого уровня данной или другой системы.

По своему определению модуль предполагает легкий способ его замены на другой при том что имеются заданные интерфейсы. Способы обособления сборочных частей ОС в отдельные модули могут быть существенно разными. Чаще всего разделения выполняется по функциональным признакам. В значительной степени разделения системы на модули определяется методом проектирования ОС используется. Как правило, модули строят с учетом характера их последующего использования. Модуль может быть построен так, что после очередного своего исполнения становится непригодным для следующего своего исполнения. Это наступает, например, если при использовании были изменены и не восстановлены в исходное состояние или совсем исчезли некоторые команды или данные модуля. Так, при работе Двухпроходный транслятора его часть, которая реализует первый проход,

Очевидно, что если такой транслятор закончил работать и возникла необходимость повторной работы с ним, то нужно повторно произвести копирование исходного текста транслятора к оперативной памяти. Модули, которые могут испортить сами себя после окончания работы и не восстанавливаться до исходного состояния, называются однократных. Если же модули могут испортить себя, но в конце работы восстанавливаются, то их называют многократными.

Особенно важное значение при построении ОС имеют модули, называется такими параллельно используются или реентерабельным. Каждый реерентабельний модуль можно использовать параллельно (одновременно) несколькими программами при их использовании.

Достижения реентерабельности реализуется в разные средства. Можно достичь реентерабельности автоматически, благодаря незминяемости кодовых частей программ при исполнении (из-за особенностей системы команд машины), а также автоматическому распределению регистров, автоматическом отделению кодовых долей программ от данных и размещению последних в математическую память.

1.1.3. Принцип функциональной видбирковости

Этот принцип является логическим продолжением частотного и модульного принципов. В ОС выделяется некоторая часть важных модулей, которые должны быть постоянно «под рукой» для эффективной организаци й вычислительного процесса. Эту часть в ОС называют ядром. При формировании состава ядра нужно удовлетворить двум требованиям, имеют противоречия. В состав ядра должны войти те системные модули, наиболее используются. Количество модулей должна быть такой, чтобы объем памяти, который занимает ядро, был не очень большим. В его состав, как правило, входят модули по управлению системой прерываний, средства по переводу программ по состоянию счета в состояние ожидания, готовности и наоборот, средства по распределению таких основных ресурсов, как оперативная память и процессор. Программы, которые входят в состав ядра, размещают перед работой ОС в оперативную память,

Кроме них существуют транзитивные системные программы, которые постоянно хранятся в памяти на магнитных носителях (МГД, ЖМГД, СД и т.д.), загружаются в оперативную память только при необходимости и в случае отсутствия свободного пространства могут перекрывать один другого.

1.1.4. принцип генеруемости

Основные положения этого принципа определяет такой способ исходного представления системной программы ОС, который позволял бы настраивать эту системную программу исходя из конкретной конфигурации конкретной машины и круга проблем, которые решаются. Эта процедура проводится редко перед достаточно длинными периодами эксплуатации ОС. Процесс генерации осуществляется с помощью специальной программы, которая позволяет описывать программные возможности системы и конфигурации машины. В результате генерации получается полная версия ОС. Она чаще всего представлена на магнитном носителе и называется дистрибутивной. Сгенерирована версия ОС представляет собой совокупность системных наборов данных, которые размещаются на томах внешней памяти. Для запуска ОС требуется осуществить ряд дополнительных процедур.

Рассмотрен принцип модульности положительно проявляется при генерации ОС. Он существенно упрощает настройку ОС на нужную конфигурацию вычислительной системы.

1.1.5. Принцип функциональной избыточности

Этот принцип учитывает возможность проведения одной и той же работы различными средствами. К примеру. Функциональную избыточность можно получить при организации пакетной мультипрограммной обработке, которая допускает три альтернативных конфигурации ОС:

- обеспечение мультипрограммирования с фиксированным количеством задач;
- обеспечение мультипрограммирования с переменным количеством задач;

• обеспечение мультипрограммирования с переменным количеством задач на основе виртуальной памяти.

Такой системой может быть ОС с мониторами - управляющими программами ОС. Каждый монитор предусматривает свою организацию обработки пользовательских программ и альтернативен к другим.

Наличие нескольких типов мониторов позволяет пользователям быстро адаптировать ОС к определенной конфигурации ЭВМ, обеспечить максимально эффективное загрузки технических средств при решении конкретного класса задач.

1.1.6. Принцип "по умолчанию"

Применяется для облегчения организации связей с системой как на стадии генерации, так и при работе с уже готовой ОС. Принцип основан на хранении в системе некоторых базовых описаний структур процессов, модулей, конфигураций оборудования и данных, которые определяют прогнозируемый объемы нужной ОП, времени счета программ и т.д. , Характеризующих пользовательские программы и условия их выполнения.

Эту информацию ОС использует как заданную если пользователь, оператор или администратор забудет или специально не конкретизирует ее. В данном случае ОС сама устанавливает соответствующие значения.

В целом использование принципа позволяет сократить количество параметров, которые устанавливает пользователь, когда он общается с системой.

1.1.7. принцип перемищуемости

Этот принцип предполагает построение модулей, использование которых не зависит от места расположения в оперативной памяти. Настройка текста модуля в соответствии с его расположением в памяти может осуществляться специальным механизмом или непосредственно перед выполнением программы, или по времени ее выполнения. Настройка заключается в определении фактических адрес, какие используются в адресных частях команд модуля, и определяется способом адресации используется в конкретной ЭВМ, а также алгоритмом распределения оперативной памяти, используемой в составе ОС.

Программный модуль, который имеет свойство перемещения, может по налаштовкы выполняться в любом месте ОП. Принцип может быть распространен как на ОС, так и на положительные программы.

1.1.8. принцип защиты

Этот принцип определяет необходимость разработки мер, которые оберегают программы и данные пользователей от искажения или нежелательных воздействий друг на друга, а также пользователей на ОС и наоборот. программы должны гарантированно быть защищенными как в своем исполнении, так и в режиме хранения. Способов утечки, как умышленных, так и неумышленно, очень много.

Не все из них можно предупредить чисто техническими средствами. Особенно трудно обеспечить защиту при использовании разделения ресурсов.

принцип защиты реализуется в той или иной форме в каждой мультипрограммной ОС.

1.2. Двохконтекстнисть работы процессору

В каждый момент времени процессор может выполнять или программу состав ОС, или любую положительную или служебную программу, не входит в состав ОС. Поскольку в составе ОС реализован принцип распределения всех ресурсов системы, нужно гарантировать невозможность непосредственного доступа к любому ресурсе разделяемой со стороны пользовательских и служебных программ.

Это осуществляется путем включения в состав команд ЭВМ команд, называются привилегированных, которые управляют распределением и использованием ресурсов. При этом запрещается использование привилегированных команд в составе программ пользователей. Они выполняются только в системных программах ОС. Наиболее часто такое решение сопровождается контролем, который осуществляется аппаратными средствами. Каждый раз, когда выполняются пользовательские или утилиты, процессор переводится в специальный (непривилегированное) режим. Программа может выполнять все команды кроме привилегированных.

При попытке выполнить команду такого рода возникает прерывание и процессор аппаратно переводится в новый режим (привилегированный). В этом режиме работают программы ОС, которые и обрабатывают прежде всего прерывания и запросы поступающие от пользователей на распределение и увеличение ресурсов. При очередном выделении любой пользовательской программе процессора он автоматически устанавливается в непривилегированное режим.

Существует большое количество принципов и форм реализации механизмов защиты данных и текстов программ, которые находятся в оперативной памяти. Такие механизмы являются производными алгоритмов распределения памяти, какие используются ОС. Самый вид защиты такого рода - контекстный защиту. Наиболее простая форма такой защиты определяется известным адресным пространством непрерывной области оперативной памяти, куда размещаются программы и данные, которые относятся к одному пользователю. Предусмотрен или непредвиденный выход за пределы этой области запрещено и контролируется при производстве каждого адреса при выполнении программ. Механизмы контроля реализуются аппаратными средствами и основаны на использовании предельных регистров или ключи памяти.

Очень разнообразные способы защиты данных, долговременно сохраняются (файлов). Традиционный способ защиты - давать пользователю или группе пользователей некоторое пароль. Способы задания и изменения паролей, идентификации по паролю прав доступа пользователя к файлам очень разнообразны. Например, над файлом в системе UNIX можно выполнить три типа операций: чтение, запись и исполнение. Это могут потенциально делать обладатель файла, группа пользователей, в которую входит и обладатель файла, другие пользователи. С помощью специальных команд каждому уровню привилегий могут быть поставлены в соответствие нужные операции над файлом. Пользователь может определять режимы доступа только для тех файлов, которыми он владеет.

1.3. Принцип независимости программы от внешних устройств

Этот принцип позволяет одинаково осуществлять операции управления внешними устройствами независимо от их конкретных физических характеристик. Принцип заключается в том, что связь программ с конкретным устройством выполняется не на уровне трансляции программы, а в период планирования й й исполнения. Перекомпиляция при работе программы с новым устройством, на котором располагаются данные, не нужны. Например, программе, которая содержит операционные обработки последовательного набора данных, все равно на каком носителе эти данные будут расположены. Изменение носителя и данных, размещенных на них (при неизменности структурных характеристик данных), не принесет каких-либо изменений в программу, если в системе реализован принцип независимости. Наиболее последовательно этот принцип реализован в ОС UNIX.

1.4. Принцип открытой и такой ОС что наращивается

Открытая ОС доступна для анализа пользователями, специалистами, какие обслуживают ЭВМ. ОС которая наращивается (такая что модефикуеться, развивается) позволяет не только использовать возможность генерации, но и вводить в ее состав новые модули, совершенствовать существующие и т.д.

лекция 2

2 Концептуальные основы ОС

2.1 Ресурс

2.1.1.Визначення ресурса

Среди многих и различных направлений, которые должна выполнять любые - какая ОС, имеется общий - обеспечить эффективный и бесконфликтный способ распределения ресурсов ЭВМ между пользователями (процессами, задачами) .с точки зрения программирования, ресурс нужно воспринимать не как где - какой показатель объекта , а как сам объект. Когда речь идет о ресурсе в этом смысле, то имеется в виду запас где - либо материальных предметов или энергетических, структурных или каких-либо внутренних характеристик предмета в составе где - либо объекта.

Понятие ресурса в контексте ОС понимает сам объект, а не показатель объекта. Понимание ресурса, как запаса чего - одновременно понимает два свойства этого понятия: полезность и полноту. Действительно, когда говорят о где - предмет, как о ресурсе, то подразумевается, что он нужен (красный) другим объектам, которые в повседневной жизни называют пользователями или потребителями ресурсов. Без потребителя не было бы и понятия ресурса. Ресурс нужен потребителю для поддержания процесса его жизнедеятельности. Когда ресурс отсутствует, процесс приостанавливается. В обычной практике потребитель получает ресурс, что этому выделяется, воспринимает его таким, исчерпывается. Иными словами, запас, который выделяется потребителям - запас предметов (внутренних свойств предметов) по мере выделения может иссякнуть.

Нетрудно представить себе запасы, также такими, которые потребляются, но и такими, что пополняются, также может возникнуть ситуация, когда единицы ресурса распределены пользователями (процессами, задачами). При этом ресурс не может быть выделен очередном пользователю. Но, по мере восстановления, одной или более единиц ресурса пользователь снова имеет право обратиться к ресурсу и, возможно, получит его для использования.

Границы, которые определяют область действия понятия "ресурс", достаточно условны. Поэтому будем считать, что любой - объект, потребляется (независимо от формы его существования), который имеет некоторую практическую ценность для потребителя, является ресурсом.

Ресурсы различаются по запасам единиц ресурса, выделяемого и бывают в этом смысле исчерпывающими и неисчерпаемыми. Полнота ресурса, как правило, приводит к жизненных конфликтов в среде потребителей. Для регулирования конфликтов ресурсы должны распределяться между пользователями по каким правилам, в самом случае их удовлетворяющими.

Вычислительную систему или ЭВМ можно представить, как ограниченную последовательность функциональных элементов, которые имеют потенциальные возможности выполнять с их помощью или над ними действий, связанных с обработкой, хранением или передачей данных. Такие элементы "пользуются спросом", или потребляются другими элементами, которые являются в общем случае пользовательскими.

Уровень детализации элементов, которые выделяются по запросам для использования в системе, может быть различным. Можно, в качестве элемента выделяется для использования, рассматривать всю ЭВМ в целом. В равном степени отдельный разряд ячейки памяти можно практиковать, как отдельный элемент, распределяется. Степень детализации зависит от того, кто и каким образом требует и использует элемент системы, выделяется. Поэтому для того, определенности, будем считать, что потребителем элементов, выделено вычислительной системой, являются процессы, которые в ней развиваются. Все элементы, которые выделяются по запросам от процессов системы отождествляют с понятием ресурса. В этом смысле будем трактовать понятие ресурса. В соответствии с ГОСТ 19781 - 83 ресурсом является средство вычислительной системы, который может быть выделен процессом на определенный интервал времени.

2.1.2.Властивости и классификация ресурсов

Упорядоченность ресурсов за де - какой классификационным признаком нужна в ОС для определения тождественных действий в отношении ресурсов данного класса - действий по учету и распределению ресурсов, предотвращения конфликтов в их пользовании и т.д.

Объединять ресурсы в тот или иной класс можно по - разному: по одной или иному признаку или их совокупности. Описанные далее свойства заимствованы из [1].

Первая классификационный признак - "реальность существования". Она касается факта существования ресурса. Факт раздела ресурсов на физические и виртуальные (кажущиеся) - самый распространенный и перспективный способ в распределении ресурсов. Под физическим понимают ресурс, который реально существует и при распределении его между пользователями имеет все присущие ему физические характеристики. Виртуальный ресурс похож многим своим характеристикам с де - каким физическим, но по многим свойствам имеет отличия.

По сути - это где - какая модель физического ресурса. Виртуальный ресурс не существует в том виде в котором он проявляет себя перед пользователем. Свойство виртуализации ресурсов - одна из важнейших свойств при построении систем управления ресурсами. По важности - это одна из важнейших концепций при построении современных ОС.

Выделение пользователю виртуального ресурса в известной степени схоже с ситуацией, когда пользователю вместо реального объекта предоставляется модель. Если пользователь, работая с моделью, получает нужные ему параметры объекта, то ему все равно, что выделено - модель или объект.

Виртуальный ресурс - это своего рода модель конкретного ресурса. Как модель, он реализуется в где - либо программно - аппаратном форме. В этом смысле виртуальный ресурс существует. Но виртуальный ресурс может оказывать пользователю при работе с ним, не только часть тех свойств, которые присущи объекта моделирования, или физическом ресурса, но и свойства, которые он не имеет. Такие дополнительные свойства не имеет физический ресурс, но пользователь имеет право утверждать обратное, так как он имеет дело не с физическим, а с виртуальным ресурсом, но он этого не подозревает.

Построение каждого виртуального ресурса осуществляется на базе где - либо физического. Имея всего один физический ресурс, можно построить на его основе несколько виртуальных. Это дает возможность существенно экономическое использовать соответствующий физический ресурс, а также увеличить гибкость политики распределения ресурсов, включая в большинстве случаев, конфликтные ситуации. (Пример игры с гроссмейстером). Признак "возможность расширения свойства" характеризует ресурс с точки зрения возможности построения на его основе где - либо виртуального ресурса. Физический ресурс, который допускает "визуализацию", или воспроизведение и (или) расширение своих свойств, называется эластичного. Неэластичным или жестким ресурсом называют физический ресурс, который по своим внутренним свойствам не допускает визуализации.

В соответствии с признаком "степень активности" различают "активные" и "пассивные" ресурсы. При использовании активного ресурса он может выполнять действия по отношению к другим ресурсам (или даже по отношению к себе) или процессам, которые, в общем случае, приводят к изменению последних. Пассивный ресурс не имеет таких свойств. Над таким объектом можно проводить допустимые для него действия, которые могут привести к изменению его состояния, или к изменению его внутренних или внешних характеристик. ЦБ является примером активного ресурса. Область памяти, выделяемой по требованию, - это пример пассивного ресурса, так как над этой областью можно выполнять операции записи или считывания информации. Очевидно, что логика распределения активных ресурсов, которая реализуется в соответствующих механизмах распределения, должна отличаться от логики распределения пассивных ресурсов.

Различение ресурсов по признаку "время существования" обусловлено динамикой ресурсов в отношении процессов, которые их используют. Если ресурс существует в системе до момента порождения процесса и доступен для использования в течение всего интервала существования процесса, то такой ресурс рассматривают как постоянный для данного процесса. Временной ресурс может появляться и (или) уничтожаться в системе динамично в течение времени существования процесса, рассматривается. Причем, создание и уничтожение может проводиться, как самим процессом, так и другими процессами - системными или пользовательскими. Понятно, что ресурсы распределяются по определенным правилам: системой взаимосвязанных процессов. Поэтому ресурсы, которые являются постоянными для одних процессов, могут быть временными для других, и наоборот.

Необходимость различения ресурсов по признаку "степень важности" обусловлена двумя причинами: обеспечение нужной работоспособности и увеличение гибкости управления процессами и распределением ресурсов. Для этого различают главные и второстепенные ресурсы. Ресурс является главным по отношению к конкретному процессу, если без его выделения процесс принципиально не может развиваться. К этим ресурсам относятся, прежде всего, центральный процессор и оперативная память. Ресурсы, которые допускают где - какой альтернативный развитие процесса, если они не будут выделены, являются второстепенными. Например, для хранения набора данных могут быть использованы гибкий магнитный диск, жесткий магнитный диск, CDROM и МЛ. Любой - какой из процессов должен провести распределение ресурсов, которые он использует по этому признаку, если известно, что развитие процессов может

выполняться в условиях отказов и сбоев оборудования, а система распределения гарантирует сохранность только главных ресурсов. При выходе из аномальной ситуации, создавшейся процесс может работать только с главными ресурсами, не достигая при этом, всех запланированных целей, которые справедливы для нормальных условий работы.

Распределение ресурсов на дорогие и дешевые связано с реализацией принципа функционального избытка при распределении ресурсов. В большинстве случаев, средства за использование ресурсов, при выполнении процесса, является функцией времени. Чаще всего, система предусматривает различный сервис, или различные условия использования одного и того же ресурса, или предусматривает альтернативные ресурсы. В обоих случаях перед пользователем стоит задача выбора - получить быстро нужный ресурс и дорого заплатить за такую услугу или подождать выделения нужного ресурса и, после его использования, заплатить меньше. В системе альтернативных ресурсов вводятся и различные цены за их использование. Примером последнего случая может быть присутствие в системе двух или нескольких трансляторов одной и той же речи.

Каждый транслятор - это отдельный альтернативный ресурс, который отличается от друг от друга нужным временем на трансляцию, расходы памяти, а отсюда, и ценой за его использование.

Средства, учитывают при распределении ресурсов признаков "Степень важности" и "Функциональный избыток", достаточно разнообразны и присутствуют практически в какой - либо ОС.

Структурная признак устанавливает ли или отсутствует у ресурса где - какая структура. Ресурс является простым, если не содержит сборочных элементов и рассматривается при распределении, как одно целое. Сборочный ресурс характеризуется где - либо структурой. Он содержит в своем составе ряд однотипных элементов, которые имеют, с точки зрения пользователей, одинаковые характеристики. При каждом разовом распределении сборочного ресурса пользователь может получить один или более таких сборочных элементов. Процессам - пользователям все равно, какой из элементов сборочного ресурса будет выделяться для них при удовлетворении их запросов на ресурс.

Простой и сборочный ресурсы отличаются количеством состояний. Простой, по структуре, ресурс может быть либо "занятым", когда выделены для использования каком процесса, или "свободный". Сборочный ресурс находится в состоянии "свободный", если ни один из его составляющих элементов не распределены для использования. Если же все элементы такого ресурса выделены для использования, то он находится в состоянии "занято". Если часть элементов распределены, а другие нет, то ресурс находится в промежуточном состоянии "частично занят".

При построении механизмов распределения ресурсов на основе использования той или иной дисциплины, особенно важным является учет использования ресурсов, которые распределяются. По этому признаку учитывается и суть ресурса - возможность, в этой связи, ресурса восстанавливаться в системе после его использования. По возможности восстанавливаться ресурсы делятся на те, которые воспроизводятся и на те, которые потребляются.

При централизованном распределении ресурсов соответствующими механизмами ОС, по отношению к каждому ресурса предполагается, что процесс - пользователь выполняет три типа действий: запрос, использования и освобождения. При выполнении действия "запрос" в ответ на требование процесса - пользователя система выделяет ресурс или отказывает в распределении. Отказ может быть вызвана тем, что ресурс, который распределяется находится в состоянии "занят" или обусловлен какой-то другой причиной. Если ресурс, после выполнения действия "запрос" переведены в состояние "занято", то процесс может использовать его. Выполняется действие использования. Действие "освобождение" выполняется по требованию процесса и сводится, в итоге, к переводу ресурса в состояние "свободный".

Если при распределении системного ресурса, допускается многократное выполнение действия в последовательности запрос - использование - освобождение, то такой ресурс называют тем, что воспроизводится.

В отношении определенной категории ресурсов правомерное использование действий в следующем порядке: увольнение - запрос - использование, после чего ресурс, который в данном случае называют тем, потребляемой изымается из сферы потребления. Такая ситуация возможна в определенных областях использования, где в отношении процессов, которые разделяют ресурс справедливое отношение типа "производитель - потребитель". Например, если один процесс производит ("освобождает") сообщения, а другой потребляет их, то каждое разовое сообщение рассматривают, как ресурс, потребляется. При этом предполагают, что процессы параллельные, развиваются с произвольными скоростями и обмениваются сообщениями через какого посредника. Процесс - посредник буферизуе сообщение, синхронизируя, при этом, действия параллельных процессов по приемке и выдаче сообщений из обобщенного буфера в соответствии с правилами взаимного исключения. выполнение

действия "освобождение" сводится к тому, что процесс - производитель производит (или делает доступным, свободным для использования) очередное сообщение и размещает его с помощью процесса - посредника в буфер. Действие "запрос" инициируется по инициативе процесса - потребителя и заключается в изъятии из буфера одного из хранимых и еще не обработанных сообщений, если таковые имеются. Это равно выделению ресурса по требованию из числа свободных. После обработки принятого сообщения, равна действия "использование", оно теряет права, или изымается из системы, как ресурс.

Таким образом, срок жизни ресурса, потребляемой который определяется периодом между выполнением действий "освобождение" и "использование", имеющий конец. В отношении процесса - производителя и процесса - потребителя потребляемые ведут себя, как временные.

Природа ресурса и (или) правило распределения ресурса, используемого обусловленные параллельной или последовательной схеме использования распределенного между несколькими процессами ресурса. Последовательная схема предполагает, что в отношении некоторого ресурса, который называют таким, что используется последовательно, допустим последовательное во времени выполнения цепочек действий "запрос - использование - освобождение" каждым процессом - потребителем этого ресурса. Для параллельных процессов такие цепочки действий являются критическими областями и должны выполняться так, чтобы удовлетворять правилу взаимного исключения. Поэтому ресурс, который используется последовательно и распределяется несколькими параллельными процессами, чаще называют критическим ресурсом. Буфер - критический ресурс для процесса - производителя и процесса - потребителя. Последовательный ресурс АЦПУ. Файл, который находится на каком-то из носителей и, который используется одним процессом для чтения с него данных, а другим для записи в него данных, становится ресурсом, последовательно используется при наложении на порядок его использования следующего требования: данные из файла можно считывать только после того, как файл полностью сформирован. Для этого нужна специальная организация работы процессов с файлом. Если бы такого ограничения не было, то файл можно было бы использовать параллельно или параллельные процессы могли бы писать и считывать информацию в произвольном порядке. Для этого нужна специальная организация работы процессов с файлом. Если бы такого ограничения не было, то файл можно было бы использовать параллельно или параллельные процессы могли бы писать и считывать информацию в произвольном порядке. Для этого нужна специальная организация работы процессов с файлом. Если бы такого ограничения не было, то файл можно было бы использовать параллельно или параллельные процессы могли бы писать и считывать информацию в произвольном порядке.

Параллельная схема предусматривает параллельное или одновременное использование одного ресурса, который так и называют параллельно используемый более чем одним процессом. Массив данных, который находится в где - либо области ОТ и такой, что допускает только чтение данных с него - пример ресурса, который используется параллельно. Другим примером может быть регистровая память процессора, которая состоит с не большого количества ресурсов общего назначения и используется каждым из внутренних процессов. При переходе любого из процессов из активного в какое - либо из пассивных состояний (ожидания или готовности) выполняется обязательное копирование содержимого регистровой памяти в где - либо область сохранения или фиксируется состояние регистровой памяти на момент переключения процессора. При обратном переходе к активному состоянию с де - либо пассивного выполняется восстановление значений регистров, которые были зафиксированы для данного процесса в момент окончания его предыдущего интервала активности. Таким образом, регистровая память параллельных процессов выступает, как ресурс, используемый параллельно.

По форме реализации различают "твердые" и "мягкие" ресурсы. Под "твердыми" понимают аппаратные компоненты ЭВМ, а также человеческие ресурсы. Все другие виды ресурсов относятся к разряду "мягких"

В классе "мягких" ресурсов выделяют два типа - программные и информационные. Это деление достаточно условно. Если "мягкий" ресурс допускает копирования и эффект от использования ресурса - оригинала и ресурса - копии идентичен, то такой ресурс называют программным "мягким" ресурсом. И наоборот, его нужно отнести к информационному типу. Примером "мягких" программных ресурсов могут служить программные модули, различные информационные структуры: массивы, файлы, дескрипторы процессов и прочее.

"Мягкие" информационные ресурсы принципиально не допускают копирование или допускают копирования, но оно является функцией времени. Это различного вида потребляемые: сообщения, сигналы, сигналы прерывания, запросы к ОС и разного рода услуги, сигналы синхронизации. Такие сообщения и сигналы информационно значимые (доступные и ценные, как ресурс) только в течение некоторого конечного интервала времени.

Из выше сказанного можно составить результирующую схему (одну из возможных), которая является наиболее общей для различных ОС:

Рис.2.1 Классификационная схема ресурсов ОС

лекция 3

2.2. Теоретические основы процесса

Для определения процессов существует большое количество определений как формального, так и не формального вида. Неоднозначность в определении возникает потому, что понятие "процесс" является определенным видом абстракции, которую по-разному используют, а отсюда и толкуй и различные категории людей. Так, точки зрения положительных и системных программистов расходятся в деталях, в формах восприятия и реализации этого понятия.

архитектуру современной ЭВМ можно считать многопроцессорной (Многопрограммного). На самом деле, процессор - это любое устройство составе ЭВМ, которое может автоматически выполнять допустимые для него ди й в некотором определенном порядке, или хранимой в памяти и непосредственно доступной таком активном устройства. Тогда кроме ЦБ (одного или нескольких) можно назвать процессором канал или устройство, которое работает с каналом. В данной трактовке оператор также подпадает под определение процессора. Между процессорами в системе существуют информационные и управляющие связи.

Каждый процессор - это такой объект в системе, который, в общем случае, хотели бы воспользоваться одновременно несколько пользователей для выполнения своей программы на процессоре. По отношению к каждому пользователю, который претендует на выполнение программы на некотором процессоре, и системы, которые распределяют этот процессор среди многих пользователей, вводится понятие "процесс".

В общем случае процесс - это некоторая деятельность, связанная с выполнением программы на процессоре.

Согласно ГОСТ 19781-83 процесс - это система действий, которая реализует определенную функцию в вычислительной системе и оформлена таким образом, что управляющая программ вычислительной системы может перераспределять ресурсы этой системы в целях обеспечения мультипрограммирования.

Деятельность может протекать по-разному. Программное в некоторый момент может предоставить процессор или забрать его. При выполнении программе могут понадобиться результаты работы других процессоров или какие-либо другие ресурсы. Иными словами, деятельностью, или ходом развития, процесса нужно управлять. Управление процессами как в отношении каждого, так и в отношении их совокупности - это функции ОС.

При выполнении программ ЦП чаще всего различают следующие характерные отдельные состояния:

- порождение - пидготовляютья условия для первого использования на процессоре;
- активное состояние, или состояние "Счет" - программа выполняется на процессоре;
- ожидания - программа не выполняется на процессоре по причине занятости любого нужного ресурса;
- готовность - программа не выполняется, но для выполнения предоставляются все нужны в текущий момент ресурсы, кроме центрального процессора;
- окончание - нормальное или аварийное окончания выполнения программы, после которого процессор и другие ресурсы ей не подаются.

Иногда используют более детальное представление состояния процессов, отличное от приведенного.

Процесс находится в каждом из своих допустимых состояний в течение некоторого времени, после чего переходит в некоторое другое состояние. состав допустимых состояний, а также допустимые переходы из одного состояния в состояние обычно задают в форме графа существования процесса.

Рис.2.1. Граф существования процесса.

Для ОС процесс в такой трактовке рассматривается как объект, в отношении которого нужно обеспечить реализацию каждого из допустимых состояний, а также допустимые переходы из одного состояния в состояние, которые могут быть причиной таких переходов. такие события

могут инициироваться и самими процессами, которые способны потребовать процессор или какой-нибудь другой ресурс, необходимый для выполнения программы.

2.2.1. Свойства и классификация процессов

Для построения средств и механизмов, реализующих совместно систему управления процессами в составе ОС, нужно определить свойства процессов в соответствии с этими свойствами.

- Процесс реального времени

Процесс определяются рядом временных характеристик. В некоторый момент времени процесс может быть порожден, а через некоторое время закончено. Интервал между этими моментами называют интервалом существования процесса. В момент порождения последовательность и длина пребывания процесса в каждом из своих состояний (трасса процесса) в общем случае непредсказуемы.

Поэтому, непредсказуемая и длина интервала существования. Но отдельные виды процессов требуют такого планирования, чтобы гарантировать окончания процесса до наступления некоторого конкретного момента времени.

Интерактивные: Это процессы, время существования которых должен быть не более интервала времени допускающий реакция ЭВМ на запросы пользователя.

Пакетные. Процессы не входящих в первых двух классов.

В любой ОС по требованию существующего или такого существовавший процесса проводится работа по порождению процессов.

порождающий - это процесс, который задает данные на порождение процессов.

рожденный - это процесс, который создан по этим данным.

Рожденный в свою очередь может выдавать требования на порождение другого процесса и в свою очередь, стать порождающим.

При управлении процессами нужно обеспечить производительность результатов работы каждого процесса, учитывать и управлять этой ситуацией, которая состояла при развитии процесса.

С этих позиций ОС должна быть способна сравнивать процессы с динамическими свойствами. Сравнение можно выполнять, используя понятие "трасса" - порядок и длина пребывания процесса в допустимых состояниях на интервале существования.

эквивалентные - это процессы, которые имеют одинаковый конечный результат обработки одних и тех же исходных данных на одной и той же или даже разных программах. Трассы эквивалентных процессов в общем случае не совпадают. Если в каждом из эквивалентных процессов обработка данных выполняется по одной и той же программе, но трассы при этом в общем не совпадают, то такие процессы называют

тождественными.

Такие равные друг другу процессы возникают при совпадении трасс в тождественных процессах.

Последовательны. Это процессы, интервалы которых не пересекаются во времени.

Параллельны. Это процессы, интервалы которых существуют во времени одновременно.

Комбинированные. Это процессы в которых на интервале рассматриваемого найдется хотя бы одна точек, в которой существует один процесс, но не существует другой, и хотя бы одна точка, в которой оба процесса существуют одновременно.

В ОС принято различать процессы не только по времени, но и по месту их развития или на каком из процессов выполняется программа процесса. Точкой отсчета принято считать ЦБ, на котором развиваются процессы.

Программным или внутренними называют процессы, выполняемые на центральном ЦБ.

Внешние это процессы, развитие которых выполняется под контролем или управлением ОС на процессорах, отличных от центрального. Например, процессы ввода-вывода, которые развиваются в канале. Деятельность оператора, который обслуживает ЭВМ и вводит информацию для выполнения одной или нескольких программ, может рассматриваться как внешний процесс.

Программные процессы принято делить на системные и пользовательские. При развитии системного процесса используется программа с состав ОС. При развитии пользовательского процесс используется пользовательская (Положительная) программа.

Взаимосвязанные процессы это процессы, которые поддерживаются между собой с помощью системы управления процесса любого рода связь: функциональные, пространственно-временные, управляющие, информационные и т.д.

В противном случае процессы называются изолированных. Если имеется между процессами управляющий связь может устанавливаются отношения вида «порождающий - порожденный".

Информационно-независимые это процессы, при развитии которые используют некоторые ресурсы, но информационно которые между собой не связаны, либо не обмениваются информацией.

взаимодействующие процессы имеют связь или функциональный, или пространственно-временных. Как правило взаимодействующие имеют информационную связь, причем схемы, а отсюда и механизмы установления таких связей могут быть разными.

Особенность, во-первых, обусловлена динамикой процессов (или есть взаимодействующие процессы последовательными, параллельными или комбинированными) во-вторых, выбранным способом связи (явным, с помощью обмена сообщениями между процессами, или неявными, с помощью структур данных разделяемых).

конкурирующими называют процессы, которые взаимосвязаны и имеют связь между собой по ресурсам.

Управление взаимосвязанными процессами в составе ОС основано на контроле и задовольнени определенных ограничений состоящих на порядок выполнения таких процессов. Данные ограничения определяют виды отношений, которые допустимы между процессами, и составляют в совокупности синхронизирующих правил.

Отношение переддии (предшествования).
Это отношение означает, что первый процесс должен переходить в активное состояние всегда раньше второго.

Отношение приоритетности. Процесс с приоритетом Р может быть переведенным в активное состояние только при выполнении двух условий: в состоянии готовности к процессу рассматриваемого отсутствуют процессы с большим приоритетом; процессор или свободный, или используется процессом с меньшим, чем Р, приоритетом.

Отношение взаимного исключения. В этом случае процессы используют обобщенный ресурс. При этом совокупность действий над этим ресурсом в составе одного процесса называют критической областью. Критическая область одного процесса не должна выполняться одновременно с критической областью над этим же ресурсом в составе другого процесса.

Рассматриваемую классификацию можно отобразить следующим рисунком.

Рис.2.1. Классификационная схема пакетных ресурсов.

В разных мультипрограммных пакетных ОС имеются соответствующие механизмы и средства, которые учитывают перечисленные свойства при их управлении.

лекция 4

2.2.2 Процессы ОС UNIX

Для системы UNIX процесс это программа, которая находится в стадии выполнения. Например, shell в UNIX - это процесс, который создается при входе пользователя в систему. Более того, когда пользователь вводит команду cat foo, shell создает новый процесс.

Согласно терминологии UNIX, это - порожденный процесс, который выполняет командую, например, cat от имени пользователя. Процесс, который создал порожденный процесс (или процесс-потомок), становится родительским процессом. порожденный процесс наследуют от родительского много атрибутов и планируется ядром UNIX к исполнению независимо от родительского.

Способность процесса создавать в процессе выполнения новые процессы дает следующие преимущества:

- Любой пользователь имеет право создавать многозадачные приложения;
- Поскольку порожденный процесс выполняется в собственном виртуальном адресном пространстве, успех или неудача при его выполнении на родительский процесс не влияет. Родительский процесс может после завершения порожденного им процесса пригласить статус завершения и статистические данные, относящиеся к периоду выполнения порожденного процесса.

- Очень часто процессы создают порожденные процессы, которые выполняют новые программы (например, программу spell). Это позволяет пользователям писать программы, которые могут путем вызова других программ расширять свои функциональные возможности, которые не требуют ввода нового исходного кода.

2.2.3 Поддержка процессов ядром ОС UNIX

Структура данных и механизмов выполнения процессов зависит от реализаци й ОС. Рассмотрим структуру данных процесса в поддержку его на уровне ОС и примере ОС UNIX System V.

Рис.2.3. Структура данных UNIX-процесса.

Из рис. 2.3. видно, что UNIX-процесс состоит как минимум из сегмента текста. Сегмент - это область памяти, которой система управляет как единым целым. Сегмент текста содержит текст программы процесс в формате машинных кодов команд. Сегмент данных содержит текст программы процесса в формате машинных кодов команд. Сегмент данных содержит статические и глобальные переменные с соответствующими данными. Сегмент стека содержит динамический стек. В стеке сохраняются аргументы функций, изменений и адреса возвращение всех функций, активных в процессе в каждый данный момент времени.

В ядре UNIX имеется таблица процессов, в которой отслеживаются все активные процессы. Каждый элемент таблицы процессов имеет указатели на сегменты текста, данных стека и U-область процесса. U-область - это разрешение записи в таблице процессов, которые содержат дополнительные данные о процессах, отдельно таблицу дескрипторов файлов, номер индексных дескрипторов текущего коренного и рабочего каталогов, набор установленных системой лимитов ресурсов процессов и т.д.

Все процессы в UNIX-системе, кроме самого первого (процесс с ID равным Ø), создаваемый программой начальной загрузки системы, создаются с помощью системного вызова fork. После завершения свое исполнение.

Рис.2.4. Структура данных родительского и порожденного процессов после завершения системного вызова fork

2.2.4 Компоненты процесса UNIX

Процессы состоят из адресного пространства и набора структурных данных, содержащихся внутри ядра. Адресное пространство это совокупность страниц памяти (страницы это "кусочки" памяти размером, как правило, от 1 Кб до 4 Кб), которые ядро выделяло для выполнения процесса. Адресное пространство содержит сегменты для кода программы, которую выполняет процесс, переменные используемые процессом, стек процесса и различную вспомогательную информацию, необходимую ядру во время работы процесса. Адресное пространство в системах с виртуальной памятью, адресное пространство в конкретный момент времени может находиться в физической памяти в целом или частично, либо вовсе отсутствовать.

В структуре данных ядра хранится различная информация о каждом процесс. К более важным относятся:

- таблица распределения памяти процесса;
- текущий статус процесса;
- приоритет выполнения процесса;
- информация о ресурсах, используемых процесс;
- маска обработки процесса;
- обладатель процесса.

2.2.5 Наиболее важные характеристики процессов UNIX

Идентификатор процесса (PID).

Каждому новому процессу, который создает ядро, присваивается уникальный номер (PID). Фактическое значение PID большой роли не играет. Идентификационные номера процессам присваиваются по порядку, начиная с нуля. Когда номера в ядре заканчиваются, они вновь возвращаются к нулю и снова присваиваются за их порядком, пропуская те PID, которые еще используются.

Идентификатор родительского процесса PPID.

В UNIX отсутствует системный вызов, который создавал бы новый процесс для выполнения конкретной программы. Новый процесс создается путем клонирования одного из уже существующих процессов, после чего текст клона заменяется текстом (в данном контексте - последовательность команд, которые выполняет ЦП) программы, которую должен выполнять процесс. Кроме собственного идентификатора, каждый процесс имеет атрибут PPID, или идентификатор своего родительского процесса.

Идентификатор пользователя (UID) и эффективный идентификатор пользователя EUID.

UID - это идентификационный номер пользователя, который создал данный процесс. Вносить изменения в процесс могут только кто его создал и привилегированный пользователь. Система учета относит на счет того, кто создал процесс все ресурсы, которые использует его процесс.

EUID - это "эффективный" UID процесса. EUID используется для того, чтобы определить, к каким ресурсам и файлов в процесса есть право доступа. В большинстве процессов UID и EUID будут одинаковыми. Исключение составляют программы, в которых установлен бит изменения идентификатора пользователя.

Идентификатор группы (GID) и эффективный идентификатор группы (EGID) GID - это идентификатор новой группы данного процесса. идентификаторы что допускаются для групп указываются в файле / etc / group и в поле GID файла / etc / passner. Когда процесс запускается, его GID устанавливается равным GID родительского процесса.

EGID связано с GID также, как EUID с UID. Если процесс пытается обратиться к файлу, на который не имеет прав обладатель, ядро автоматически проверяет, можно ли предоставлять разрешение на основе данного EGID.

2.2.6 Жизненный цикл процесса

Новые процессы в системе не порождаются сами по себе, они порождаются другими процессами.

Для создания нового процесса существующий процесс копирует самого себя с помощью fork. вызов fork создае
копию исходного процесса, идентичном
отцу, но такую, которая имеет следующие отмены:

- у нового процесса свой PID;
- PPID нового процесса равна PID отца;
- Информация учета нового процесса обнуления;
- у нового процесса имеется свой собственный экземпляр дескрипторов файлов. Последняя из этих различий может привести к некоторой путанице. Номера индексных дескрипторов ядро выделяет процессам при открытии файла или гнезда. Эти номера по сути дела являются индексами небольшой таблицы, содержащей указания на структуры данных ядра. Когда вызывается fork копируется именно эта таблица, а не соответствующие структуры ядра. В результате операции, с этими структурами выполняет порожденный процесс могут оказуваты непосредственное влияние на родительский процесс.

Пусть процесс не имеет с дескриптора файла некоторые данные. Когда в следующий раз родительский процесс будет пытаться прочитать информацию с этого дескриптора, он начнет чтение с того места, где остановил чтение порожденный процесс, а не оттуда, откуда он начал бы это делать до вызова fork.

Системный вызов fork обладает уникальными свойствами: он возвращает сразу два значения. С точки зрения порожденного процесса эта операция всегда выдает Ø. С другой стороны, родительскому процессу возвращается PID снова созданного порожденного процесса. Таким образом эти два процесса могут различать друг друга. Это можно продемонстрировать с помощью программы:

```
 int kidpid; kidpid = fork
();  if (kidpid == Ø) {

/ * Это порожденный процесс * /} else {

    / * Это родительский /}
```

после выполнения fork новый процесс запускает новую программу с помощью одного из системных вызовов семейства exec.

Все вызовы семейства exec выполняет примерно одинаково функции: изменяет текст программы, которую выполняет процесс, и устанавливает сегменты данных и стека и исходное состояние. когда вызывается одна из подпрограмм exec, она записывает в адресное пространство процесса содержание нового программного файла, а после возобновляет выполнение с указанной точки входа нового текста. Формы вызова exec различаются только способом указания аргументов командной строки и среды передается новому процессу.

Рассмотрим пример использования fork и exec для порождения нового процесса

```
if (fork () == Ø) {/ * Есть
потомка * /
execl ( "/ bin / ls", "ls", "/ usr / bin", (char *) Ø) }
```

В этом примере программа выполняемой запускает новый процесс, который становится вызовом команды ls. Этот эффект идентичен ввода команды / usr / bin / ls с shell. По согласованию имя программы - первый аргумент, передаваемый в новый текст. Интерпретатор команд обычно сам зебезпечуе соблюдения этого правила, но его нужно делать при программировании.

Когда система загружается, ядро самостоятельно создает несколько процессов. Наиболее важный из них - процесс ini t, PID которого всегда равен 1. Этот процесс отвечает за вызов shell для выполнения сценариев запуска rc. Все процессы, кроме тех, которые создает ядро, являются потомками init.

Процесс init играет важную роль и в управлении процессами. Когда процесс завершается, он вызывает подпрограмма _exit, чтобы сообщить ядро о своей готовность "умереть". По параметры подпрограмме _exit передается код завершения процесса. По согласованию нулевой код завершения значит, что процесс был "успешным".

Код завершения требуется родительском процесса, поэтому ядро должно хранить его системным вызовом wait. Код завершения - единственная информация, которую хранит ядро о завершении процессы. Если процесс был завершен снаружи (Сигналу), то его родительский процесс может выяснить, что это был за сигнал и был выполнен дамп оперативной памяти.

Дело в том, что когда процесс завершается, его адресное пространство увеличивается, время ЦБ этому процессу не выделяется, но в таблице процессов ядра запись о нем сохраняется. Процесс в этом состоянии называют "зомби".

лекция 5

2.2.7 APE процессов (Application program interfase) (создание и завершение)

2.2.7.1 Функции fork

Системный вызов fork, как указывалось, используется для создания порожденного процесса. Прототип функции fork имеет следующий вид:

```
# i fdef _POSIX _SOURCE
# include <sys / stdtypes.h>
# else
# include <sys / types.h>
# endi f

pid_t fork (void)
```

Функция fork не принимает аргументов и возвращает значение типа pid_t (которое определяется в <sys / types.h>). Этот вызов может давать один из следующих результатов:

успешное выполнение. Создается порожденный процесс, и функция возвращает идентификатор этого порожденного процесса родительскому. Рожденный процесс получает от fork нулевой код возврата;

неудачное исполнение. Рожденный процесс не создается, а функция присваивает переменный errno код ошибки и возвращает значение -1. Основные причины неудачи выполнения fork и соответствующие errno.

ENOMEM - Для создания нового процесса не хватает свободной памяти. EAGAIN - Количество текущих процессов в системе превышает установленное системой ограничения, нужно повторить вызов позже.

Существуют ограничения, устанавливаемые системой на максимальное количество процессов, которые создает один пользователь (CHILD_MAX), и максимальное количество процессов, которые одновременно могут существовать во всей системе (MAXPID). Если любое из этих ограничений при вызове fork нарушается, то функция возвращает код неудачного завершения. Символы MAXPID и CHILD_MAX определяются соответственно в заголовках <sys / param.h> и <limi ts.h>. Кроме того, процесс может получить значение CHILD_MAX с помощью sysconf.

```
int child_max = sysmkonf (_SC_CHILD_MAX)
```

При успешном вызова fork создается порожденный процесс. Как рожден, так и родительский процессы планируются ядром UNIX для выполнения независимо, а очередность запуска этих процессов зависит от реализации ОС. Пример реализации fork в программе tast_fork.c.

```
# include <iostream.h>
# include <stdio.h>
# include <unistd.h> int main
() {

switch (fork ())        {
case (pid_t) 1: perror ( "fork") / * Fork fails * / break; case (pid_t) •: cout <<
"Процесс потомок ств. \ N "; return •;

defaul t: cout << "Эт. к родителям. проц. \ N "; }
```

```
    return •;
}
```

Родительский процесс вызывает fork для создания порожденного процесса. Если fork возвращает -1, значит, системный вызов не исполнился, и родительский процесс вызывает perror для вывода диагностического сообщения в стандартный поток ошибок. Если же fork выполнено успешно, то порожденный процесс после своего выполнения выдает на стандартный вывод сообщения "Процесс потомок создан". Затем он завершается с помощью оператора return. Между тем родительский процесс выводит на экран сообщение "Эт. к родительского процесса после fork "и также завершает свою работу.

2.2.7.2. функция _exit

Системный вызов _exit завершает процесс. Отдельно, этот APE вызывает освобождение сегмента данных процесса вызываемого сегментов стека и U-области и закрытия всех открытых дескрипторов файлов. Но запись в таблице процессов для этого процесса остается нетронутою, с тем чтобы там регистрировался статус завершения процесса, а также отражалась статистика его выполнения (например, информация об общем времени выполнения, количество пересылаемых блоков ввода-вывода данных и т.д.). Теперь процесс называют зомби-процессом, так как его выполнение уже не может быть запланировано. Родительский процесс может выбрать данные, которые хранятся в записи таблицы процессов с помощью системного вызова wai t или waitpid. Эти APE освобождают также запись в таблице процессов, которая относится к порожденного процесса.

Если процесс создает порожденный процесс и заканчивается к концу последнего, то ядро назначить процесс init как управляющий для порожденного процесса (этот процесс, создается после первоначального скачивания порожденного ОС UNIX. Его идентификатор всегда равен 1). После завершения порожденного процесса соответствующая запись в таблице процессов будет истреблен процессом init.

Прототип функции _exit имеет следующий вид:

```
 #  include <unistd.h> void _exit (int
exit_code)
```

Целочисленный аргумент функции _exit - это код завершения процесса. Родительском процесса передаются только младшие 8 бит этого кода. Код завершения Ø означает успешное выполнение процесса, а ненулевой код - на удачу исполнения. Библиотечная функция exit является оболочкой для _exit. Отдельно exit сначала очищает буфера и закрывает все открытые потоки процесса вызывающего. Затем она вызывает все функции, которые были зарегистрированы с помощью функции atexit, и наконец, вызывает _exit. Для завершения процесса. Пример программы test_exit

```
• • •    # include <iostream.h>
   # include <unistd.h> int main
   () {
       cout << "Test program for _exit" << end; _exit ( •)

       return •;
     }
```

Когда программа запускается, она объявляет о своем существовании и затем завершается с помощью вызова _exit. Для указания успешного выполнения программы она передает нулевое значение кода завершения.

2.2.8. сигналы

Сигнал - это способ указать процесса о некоторых событиях, при которых процесс должен перейти к какому-то состояния. В ОС UNIX определено более тридцати различных видов сигналов. Сигналы отсылаются процесса с помощью команды kill. Когда поступает сигнал, возможен один из двух вариантов развития событий. Если процесс назначил данном сигнала подпрограмму обработки, то вызывается эта программа и предоставляется информация о контексте, в котором был издан сигнал.

И наоборот, ядро выполняет от имени процесса действия, определенные по умолчанию. Эти действия различны для разных сигналов. Многие сигналов завершают процесс, а некоторые при этом выполняют дамп оперативной памяти.

Вызов подпрограммы обработки сигнала называется перех сигнала. Когда завершается подпрограммы обработки, процесс возобновляется с той точки, где был получен сигнал.

Для того чтобы сигнал не поступал в программу, можно указать, что они должны игнорироваться или блокироваться. Сигнал, который игнорируется просто должен откидываться и не оказуваты на процесс никакого влияния. Блокирован сигнал ставится в очередь на издание, но ядро не требует от процесса никаких действий до явного разблокирования сигнала. Подпрограмма обработки разблокированного сигнала вызывается только один раз, даже если в течение периода блокировки было получено несколько экземпляров этого сигнала.

Сигналы с именем KILL и STOP нельзя ни перехватить, ни блокировать, ни игнорировать. Сигнал KILL разрушает процесс принимающей, а сигнал STOP приостанавливает его исполнение до получения сигнала CONT.Сигнал CONT можно перехватить и игнорировать, но нельзя блокировать. Существует много способов использования сигналов. Они могут ссылаться одним процессом в другой для обеспечения связи между процессами, которые управляют терминалами процесса для предоставления пользователю возможности прерывания или приостановки этого процесса, ядром в очень многих ситуациях и даже процессом самого себя.

Сигналы еще называются - системных вызовов (fork, _exit).

2.2.9 Функции wait, waitpid

Родительский процесс использует системный вызов wai t и waitpid для перехода в режим ожидания завернення порожденного процесса и для выборки его статуса завершения (присвоенного порожденным процессом с помощью функцией й _exit). Кроме того, эти вызовы освобождают ячейку таблицы процессов порожденного процесса с тем, чтобы она могла использоваться новым процессом. Прототипы этих функций имеют следующий вид:

```
#  include <sys / wait .h> pid_t wai t
 (int * status_p)
 pid_t wai tpid (pid_t child_pid, int * status_p, int options)
```

Функция wai t приостанавливает выполнение родительского процесса до тех пор, пока ему не будет отправлено сигнал, или пока один из его порожденных процессов не завершится или не будет установлено (а его статус еще не будет сообщено). Если порожденный процесс уже завершился или был установлен до вызова wait, функция wai t сразу вернется со статусом завершения порожденного процесса (его значение содержится в аргументе status_p), а значению возвращающиеся функции будет PID порожденного процесса. Но, если, родительский процесс не имеет рожденным процессов, завершение которых он ожидает, или был проверен сигналом при выполнении wait, функция возвращает значение -1, а переменная errno будет содержать код ошибки. Надо понимать, что если родительский процесс создал более одного рожденного, функция wai t будет ожидать завершения любого из них.

Функция waitpid более универсальной по сравнению с wai t. Аналогично wai t waitpid сообщает код завершения и идентификатор порожденного процесса по его завершением. Но в случае с waitpid в процессе вызывающий можно указать, завершение которого с порожденных процессов нужно ожидать. Для этого нужно аргумента child_pid присвоить одно из следующих значений:

Такое равной ID процесса
- Порождения процесса с данным идентификатором
- 1
- Любого порожденного процесса •
•
- Любого порожденного процесса, который принадлежит к той же группе процессов, и родительский

Отрицательное значение, но не -1
- Любого порожденного процесса, индефикатора группы (GID)

Кроме того, процесс вызывающий может дать функции waitpid указание быть блокирующей либо не блокирующей, а также ожидать завершения порожденного процесса, приостановленного или непризупиненого механизмом управления задачами. Эти опции указываются в аргументе options. Так, отдельно если аргумент options установлено значение WNOHNG (этот флаг определен в <sys / wait .h>, вызов будет не блокирующим (или функция мгновенно вернет управление, если отсутствует порожденный процесс, который соответствует критериям ожидания). В обратном случае вызов будет таким блокирующий а родительский процесс будет установлено, как при вызове wait.

Дали, если аргумент options установлено значение WUNTRACED, функция также будет ожидать завершения порождено процесса, остановленного в результате действия механизма управления задачами (но о статусе которого не было сообщений ранее).

Если фактическое значение аргумента status_p вызывает wait или waitpid равен NULL, нет необходимости запрашивать статус завершения порожденного

процесса. Но если его фактическое значение является адресом целочисленной переменной, функция присвоит этой переменной код завершения (указанный в API _exit).

После этого родительский процесс может проверить этот код завершения с помощью следующих макрокоманд, которые определены в заголовке <sys / wait .h:

WIFEXITED (* status_p) Возвращает нулевое значение, если порожденный процесс был завершен с помощью вызова _exit, в обратному случае возвращается нулевое значение.

WEXITSTATUS (* status_p) Возвращает значение кода завершения порожденного процесса, которое присвоено вызовом _exit. Эту микрокоманду нужно вызвать только в том случае, если WIFEXITED возвращает НЕ нулевое значение.

WIFSIGNALED (* status_p) Возвращает значения не нулевое, если порожденный процесс был завершен по причине прерывания сигналом.

WTERMSIG (* status_p) Возвращает номера сигнала, который завершает порожденный процесс. Эту макрокоманду нужно вызвать только в том случае, если WIFSIG-NALED возвращает НЕ нулевое значение.

WIFSTOPPED (* status_p) Возвращает значения не нулевое, если порожденный процесс был остановлен механизмом управления задачами.

WSTOPSIG (* status_p) Возвращает номера сигнала, который завершает порожденный процесс.

Эту микрокоманду нужно вызвать только тогда, когда WIFSTOPPED возвращает НЕ нулевое значение.

Приведенную выше информацию можно получить непосредственно с * status_p. Отдельно семь младших битов (биты с • по 6-й включительно) * status_p содержит ноль, если порожденный процесс был завершен с помощью функцией й _exit, или номер сигнала, процесса, который завершился. Восьмой бит * status_p равен 1, если от прерывания порожденного процесса сигналом было создано дамп образа процесса (файл core).

В противном случае он равен нулю. Далее, если порожденный процесс был завершен с помощью API _exit, тогда биты 8-го по 15-й * status_p является кодом завершения порожденного процесса, который передан по допомогою_exit. Следующая схема демонстрирует использование битов данных * status_p.

Рис. 2.6. Файл создания файла core

Если возвращаемого значения wai t или waitpid является положительным целым значением, то это - идентификатор порожденного процесса, в противном случае - это (pid_t) - 1. Последнее означает, что либо ни один из порожденных процессов не соответствует критериям ожидания, или функция была прервана перехваченным сигналом. В этом случае errno может быть присвоено одно из следующих значений:

EINTER wait або waitpid возвращает управление процесса, вызываемого так как системный вызов был прерван сигналом

ECHILD Для wait это значит, что процесс, который вызывает не имеет порожденного процесса, завершение которого ожидается. В случае waitpid это значит, что либо значение child_pid недопустимо, или процесс не может находиться в состоянии определенном значением options.

EFAULT Аргумент status_p указывает на недоступную адрес.

EINVAL Значение options недопустимо.

Как wait, так и waitpid являются стандартными POSIX.1. Пример демонстрирующий использование API waitpid

```
# include <iostream.h>
# include <stdio.h>
# include <sys / types.h>
# include <sys / wait .h>
# include <unistd.h> int main
() {

    pid_t child_pid, pid; int status;

    switch (child_pid = fork ()) {

      case (pid_t) 1: perror ( "fork") / * Fork tailse * / break; case (pid_t) •: cout <<
      "Child process created \ n"; _exit (15);
                          / * Terminate child * /
      defaul t: cout << "Parent process after fork \ n"; pid = waitpid (child_pid,
      & status, WUNTRACED) }

    if (WIFEXITED (status))
       cerr << child_pid << "exits:" << WEXITSTATUS (status) << endle; else if (WIFSTOPPED
       (status))
       cerr << child_pid << stopped by: «<WSTOPSIG (status) << endl; else if
       (WIFSIGNALED (status))
       cerr << child_pid << "killed by:" << WTERMSIG (status) << endle; else perror ( "Waipid")
       _exit ( •)

       return •;
```

Эта программа создает порожденный процесс, который подтверждает свое производство, а затем завершается с кодом завершения 15. Между тем родительский процесс приостанавливает свое выполнение с помощью вызова функции waitpid.

2.3.5.2 Функция exec

Системный вызов exec заставляет процесс, вызываемый изменить свой контекст и выполнить другую программу. Существует шесть версий системного вызова exec. все они выполняют одну и ту же функция, но отличаются друг от друга аргументами.

```
# include <unistd.h>
int execl (const char * path, const char * arg ...) int execlp (const
char * file, const char * arg ...)
int execlp (const char * path, const char * arg ..., const char ** env) int execl (const char * path,
const char ** argv, ...); int execlp (const char * file, const char ** argv, ...);

int execlp (const char * path, const char * argv, ..., const char ** env)
```

Полный аргумент функции является или полным дорожным именем или именем файла программы, которая подлежит выполнению. Если вызов проходит успешно, данные и команды процесса вызывающего, который хранится в памяти, перекрывается данными и командами новой команды. Процесс начинает выполнять новую программу по ее первой строки. После завершения новой программы код завершения процесса передается обратно родительском процесса. В отличие от fork, который создает

порожденный процесс, который выполняется независимо от родительского exec меняет содержание процесса, вызываемого для выполнения другой программы.

Вызов exec может быть неудачным, если в программу, которая подлежит выполнению, отсутствует доступ или если она не имеет разрешения на выполнение. Далее программа, которая указана в первом аргументе вызова exec, должна быть файлом, выполняется. В UNIX, но, можно указать имя сценария shell для вызовов execlp и execvp, чтобы ядро UNIX для интерпретации сценария shel l было активизировано.

Аргументы arg и argv являются аргументами также для программы, которую вызвано функцией exec. Они отражаются в переменную argv функции новой программы. Для функций execl, execlp и execle аргумент arg отражается в arg [•]• значение, указанное после arg - в argv [1] и т.д. Список аргументов, которые имеются в вызове exec, должен завершиться значением NULL, чтобы указать функции, где нужно завершить значение аргумента. для функции

execv и execvp аргументов argv является массивом (Вектором) указателей на символьные строки, где каждая строка является одним значение аргумента. Аргумент argv отражается непосредственно в переменную argv функции main новой программы. Так, символ l в имени функции exec указывает на то, что значение аргумента передается в виде списка, а символ v в этом имени значит, что аргументы передаются в векторном формате.

Лекция 7

2.4 Концепция виртуализации

В любой мультипрограммной системе используется та или иная форма процесса обработки входных и выходных данных, которые называют Спулинг. Основу такого процесса составляет виртуализация перефирийных устройств которые есть по типу такие, которые используются последовательно. Это прежде всего устройства ввод и печатающие устройства вывода. Количество этих устройств в в комп'ютеров ограничено и вместе с этим они активно должны использоваться, по сути, каждым из процессов, которые развиваются параллельно одним к другу. Вводить и выводить информацию с таких устройств можно только последовательно законченными частями. Поэтому эти устройства введеннявиведення могут стать узким местом при организации параллельного развития процессов, если не предпринятому то меры. Виртуализация клавиатуры и печатающих устройств является наиболее действующей мере. Каждый процесс или совокупность процессов,

какие составляют автономную работу, получают при необходимости собственный (Виртуальный) устройство ввода или вывода, откуда возникает ввода или вывода нужной информации без влияния других процессов. Причем в конце концов данные будут вводится в установленном порядке с одной реального устройства ввода, а выводится на печать также один печатающее устройство. При этом на листинга печатающего устройства каждый пользователь получит свои функциянально отделены в поле листинг текстами, а не смесь строк текста произвольно чередуются и выдаются из разных процессов.

- Реализация спульнга в своей основе достаточно проста. каждый виртуальный устройство моделируется некоторой областью внешней памяти, как правило дисковой. По смыслу каждая такая область является буфером между реальным устройством ввода - вывода и соответствующим процессом. Как правило, входной буфер заполняется данными, которые читаются с реального устройства считывания еще до инициализации соответствующей работы. Выходной буфер, накапливает данные по мере выполнения процессом команд вывода. По Этими командамы данные пересылаются не в печатая устройства, а в выходной буфер. Но для отвечающего процесса возникает иллюзия того, что он совершил выдачу на реальное устройство. Физическая передача информации на печатающее устройство выполняется только после того, как закончится процесс (или совокупность процессов), которой является поставщиком информации в выходной буфер. При мультипрограмёний обработке в системе существует одновременно несколько выходной и входных и выходных буферов. Причем их заполнения, моментами готовности использования процессами выходных буферов и моменты окончания процессами заполнения выходных буферов постоянно меняются. Поэтому входные и выходные буферы связывают в ответе очереди, которые обслуживаются в установленном порядке.

- В результате использование виртуализации для организации спулинга добиваются очень существенных результатов в работе ОС. Операция ввода - вывода (виртуальные по своей сути) выполняются быстрей и не задерживает развитие процесса из-за недоступности реального устройства ввода - вывода. Наконец, можно спланировать и обеспечить равномерную загрузку устройств ввода - вывода.

- Приведенный пример иллюстрирует возможность построения и использования виртуальных ресурсов на базе реальных ресурсов. В конце концов каждый виртуальный ресурс подобного рода моделируется вторым процессом. Управление таким процессом выполняется в составе специальной системы, якя является по смыслу распределителем эластичного ресурса, на базе которого допускается один или более виртуальных ресурсов. Для построения виртуальных ресурсов используются дополнительные ресурсы, которые розподилюються специальным образом в зависимости от назначения виртуальных ресурсов. Если строится не один, а больше виртуальных ресурсов, тогда в составе распределителя реализуются дисциплины распределения таких ресурсов среди процессов - пользователей.

- Наиболее характерными представителями виртуального ресурса, которой построено на базе эластичного пассивного ресурса является виртуальная память.

2.4.1. виртуальная машина

- Наиболее законченным и естественным проявлением виртуальности является понятие виртуальной машины. По сути любая ОС, является средством распределения ресурсов которая организует по определенным правилам управления процессами на базе скрытой аппаратной части, создает у пользователей видимость виртуальной машины.
- Сущность восприятия характеристики виртуальной машины у пользователей может быть существенно отличной. Некоторые из них видят и используют виртуальную машину как некое устройство которой может воспринимать его программы, написанные на определенной языке программирования, выполнять их и выдавать результаты. При таком языковом представлены пользователя совершенно не интересует структура машины, а способы эффективного использования ее частей.

- Он мыслит и работает с машиной в терминах языка которую он использует.
- Чаще всего виртуальная машина, которая предоставляется пользователю воспроизводит архитектуру реальной машины, но архитектурные элементы в таком представлены выступают с новыми или улучшенными характеристиками. Характеристики могут быть произвольными, но чаще всего пользователи хотят иметь собственную "идеальную" машину по архитектурным характеристикам в следующем составе:

- - бесконечная по объекту память с таким произвольно выбирается, наиболее удобным для пользователя способом доступа к объектам, которые хранятся в памяти;

- один или несколько процессоров, которые могут выполнить действия, которые пользователь может выразить в терминах некоторых удобных для него языков программирования;
- произвольное количество внешних устройств с удобным способом доступа и представлением информации, которая передается через эти устройства или такой хранящейся ими без каких-либо слышимых ограничений на объем информации.
- Степень приближения к "иделальнои" машины может быть больше или меньшей в каждом случае. Чем больше машина, которая реализуется средствами конкретной ОС на базе конкретной аппаратной части, приближенная к "идеальной" по характеристикам машины и, следовательно, чем больше ее архитектурно-логические характеристики отличные от тех, что существуют реально, тем больше степень вертуальности в полученной пользователем машины.
- Хорошим примером обобщенным характеристикам виртуальной машины может быть мультипроцессорный вычислительный комплекс "Эльбрус".
- В состав цого виртуальной машины входят:
- 1. единообразно по логике работы память (виртуальная) практически неограниченного объема. Среднее время доступа сравним со значением этого параметра ОП. Организация работы с информацией в такой памяти выполняется в терминах обработки данных - в терминах работы с сегментами данных на уровне выбранного языка программирования.

- 2. Произвольное количество процессоров (виртуальных), которые могут работать параллельно и взаимодействовать, в том числе синхронизация и информационных взаимодействий й, реализованы и доступном пользователям уровне языка, используемого в терминах управления процессами.

- 3. Произвольное количество внешних устройств (виртуальных), которые могут работать с памятью виртуальной машины параллельно или последовательно, асинхронно по отношению к работе того или иного виртуального процесса, которые инициируют работу этих устройств.

информация, которая передается или хранится на виртуальных устройствах, не ограничено допустимыми размерами. Доступ к такой информации осуществляется на основе или последовательного или прямого способа доступа в терминах системы управления файлами. Предусмотрено расширение информационных структур данных, которые хранятся на виртуальных устройствах.

- Каждый пользователь может получить виртуальную машину с указанными свойствами. Причем семья таких машин реализуется одновременно на единый аппаратном конфегурации, которая характеризуется фиксированный составом: ОП объемом до 4,6 Мбайт; ЦП-до 10; ЗП; ПВВ - до 4; ПВО - до 16.

- Концепция виртуальности нашла широкое применение при проектировании и реализации ОС. Наиболее рациональной представить структуру системы в виде определенного набора планировщиков процессоров и распределителей ресурсов.
- Последние часто называют мониторами. В данном случае под монитором будем понимать распределитель некоторого ресурса, который может на основе некоторой организации работы обеспечить ту или иную степень виртуальности при распределении эластичного ресурса.

- Модель, которой реализовано в системе как монитор, является не только средством виртуализации ресурса, скрытый в модуле, но и средством "крупномасштабного программирования". Изменения внесенные в организацию распределения ресурса не сопряжено в системе с видимиму затрудненіями. Меняться должен только один модуль (его внутреннее содержание) при неизменности интерфейса с ним. Окружающую среду, или процессы, которые взаимодействуют с модулем, такие изменения никак не касаются. Это дает возможность автоновно разрабатывать отдельные мониторы в составе ОС и, если это нужно, достаточно просто вносить при эксплуатации ОС нужны перемены.

- Использование концепции виртуальности положено в основу зходжувального метода проектирования и разработки ОС. ОС строят как иерархию вложенных одна в другую виртуальных машин.
- Низким уровнем иерархии является аппаратные средства компьютера. следующий уровень - программный, который совместно с нижним уровнем что спрягается обеспечивает достижение компьютером новых свойств. Получается виртуальная машина первого уровня. Далее относительно этой, уже виртуальной машины разрабатывается новый программный слой. В результате получается виртуальная машина второго уровня. Такое последовательное, все более отвлеченное относительно аппаратной части дает возможность построить виртуальный компьютер нужного уровня.

- Каждый новый уровень при данном способе построения ОС дает возможность расширять функциональные возможности по обработке данных, позволяет достаточно просто организовать доступ к низших уровней.

2.4.2 Свойства уровней абстракции иерархического построения ОС

.

Использование метода иерархического упорядочения виртуальных компьютеров кроме преимуществ, которые связаны прежде всего с систематичностью проекта, увеличивает надежность сложной программной системы, которой является ОС, уменьшить сроки разработки, привело к очень существенных проблем. Главная из них - определение свойств и количества уровней абстрактным свойств и количества уровней абстракции, определение правил виненсення на каждый уровень нужных частей ОС. Наиболее последовательные свойства уровней абстракции определены Г.Майерсом.

1. На каждом уровне ничего неизвестно о свойствах (и даже при существовании) более высоких уровней.
2. На каждом уровне ничего неизвестного о внутреннем строении других уровней. Связь между условиями осуществляется только через жесткие, ранее выделенные сопряжения.
3. Каждый уровень определяет собой группу модулей, некоторые из которых является внутренним для уровня, или недоступные для других уровней. Имена других модулей известны на следующей, более высоком уровне и является сопряжением с этим уровнем.
4. Каждый уровень розпоряжаеться определенными ресурсами и или скрывает от других уровней, или предоставляет другим уровням некоторые их обстракции (виртуальные ресурсы).
5. Каждый уровень может обеспечить некоторую абстракцию данных в системе.
6. Предположение, которые на каждом уровне делаются относительно других уровней, должны быть минимальными.

7. Связь между уровнями ограничен явными аргументами, которые передаются из одного уровня на другой. Недопустимо совместное использование несколькими уровнями глобальных данных. Желательно полностью исключить использование глобальных данных. Под глобальными понимают данные, которые не описано в модуле, но доступны для использование.

8. Каждый уровень должен иметь высокую прочность и слабое обеспечение с другими уровнями. Любая функция, которая выполняется уровнем абстракции, должна быть представлена единым входом.

Таким образом, концепция виртуальности в современном ее толковании стала одной из ведущих как при построении отдельных механизмов, так и всей ОС в целом. Последовательное ее использования является основой в достижении как системной, так и пользовательской эффективности при организации мультипрограммной ЭВМ.

Лекция 8

2.6 Дисциплины распределения ресурсов, которые используются в ОС

Идея мультипрограммирования непосредственно связана с началом очередного процесса. Так процессор - основной ресурс многопрограммной ЭВМ поочередно предоставляется процессам. Подобные очередь неменуче имеют место при обращении к каналам, которые широко используются внешним приставом (например, к устройствам печати), наборам

данных, модулям ОС.

Использование многими процессами того или иного ресурса, который в каждый момент времени может обслуживать только один процесс, осуществляется с помощью дисциплин распределения ресурса.

Их основой являются:

- Дисциплины формирования очередей на ресурсы или совокупность правил, которые определяют размещения процессов в очереди;
- Дисциплины обслуживания очереди или совокупность правил извлечения одного из процессов очереди с последующим представлением выбранном процесса ресурсов для использования.

Основным конструктивным, согласовывая элементом при реализации этой или иной дисциплины диспетчеризации есть очередь, в которую по определенным правилам заносятся или вытягиваются запросы.

Определенное влияние на содержание дисциплины формирования очередей производят:

- информация о классах и приоритеты задач и шаги заданий, информация о необходимости обращения к тем или иным устройств, массивы данных, которые зафиксированы в операторах языка управления заданиями;

- согласования о приоритете уровней запросов прерывания и прерывая программ, принимаемых при проектировании и разработке компьютеров;
- наборы согласований, которые принимаются администратором сети;
- дисциплина обслуживания очередей которая используется, очень часто определяет и дисциплину формирования очереди.

Дисциплины формирования очередей делятся на два класса:

- Статический, где приоритетным назначаются для выполнения пакета заданий; большинство фактов рассматриваемых определяют содержание статических дисциплин;
- Динамический, при котором приоритетным определяются в процессе использования пакета.

Оба класса широко применяются в практике вычислительного процесса на компьютере.

В компьютерах не только многопрограммных, но и в однопрограммных, однопроцессорных, многопроцессорных широко применяется ряд дисциплин обслуживания очередей, которые стали классическими. Эти дисциплины распределения ресурсов часто называют базовыми.

2.6.1. Дисциплины наиболее часто встречаются на практике

2.6.1.1 Дисциплина обслуживания в порядке поступления

Первый пришел - первый обслуживается. В литературе эта дисциплина обозначается как FIFO (First in - First out).

Самая и широко используется на практике.

Все заявки попадают в конец очереди. Первым обслуживаются заявки, которые находятся в начале очереди.

Рис. 2.6 Схема дисциплины обслуживания процессов первый пришел - первый обслуживается

2.6.1.2. Дисциплина обслуживания в порядке, обратном поступлению

Последняя пришла - первая обслуживается.

Сказывается LIFO (Last in - Fi rst out).

Также, как и FIFO, проста в реализации и широко используется на практике. Данная дисциплина является основой построения стековой памяти.

Рис. 2.7 Схема дисциплины обслуживания процессов последний пришел - первый обслуживается

Общим для вказиних дисциплин является простота их реализации и определена "справедливость" в обслуживании всего потока запросов, поступающих в систему. Среднее время ожидания запросов в очереди при некотором темпе обслуживании установившийся и темпе поступления одинаков независимо от характеристик процессов - пользователей. Например, если некоторые процессы предусматривают длинные использования ресурсов (отрабатываются "длинные" запросы), а другие, наоборот,

-

нетревали (отрабатываются "короткие" запросы), тогда и "длинные" и "короткие" запросы будут ожидать в очереди в среднем одинаково.

Дисциплина FIFO кроме кроме функциональной отмены обеспечивает минизацию дисперсии времени ожидания.

2.6.1.3. Круговой циклический алгоритм

В основе данной дисциплины лежит дисциплина FIFO. Но время обслуживания каждого процесса ограничено и определяется так называемым квантом времени t_k. Если запрос на использование ресурса с начала очереди обслуживается до конца за время t_k (например, программа процесса за время t_k полностью выполнен на процессоре), он покидает очередь. Если этот запрос не успевает обслужиться до конца, то его обслуживание прерывается и он поступает в конец очереди.

Рис. 2.8. Схема багаточерговои дисциплины обслуживания

Дисциплина широко используется на практике, в частности при реализации режима раздела времени.

Хотя в данной дисциплине отсутствуют явные приоритеты, в ней автоматически выполняется дискриминация "длинных" и "коротких" запросов. В наиболее благоприятных условиях находятся короткие запросы, или запросы от процемив, которым нужно меньше времени использования ресурсов. Короткие запросы обслуживаются быстрее, или имеют меньшие средние времена ожидания в системе, чем длиннее запросы. Определение коротких и длинных запросов выполняется по мере последовательного циклического мультиплексирования процессами ресурса. Степень содействия коротким запросам тем больше, чем ченше длина кванты мультиплексирования, чем ближе она к длине интервала номинального использования ресурса процессом. Но уменьшение длины кванта ведет к увеличению накладных расходов,

которые нужны для обработки прерываний и переразподилу ресурса. Это возникает из-за роста части прерываний, особенно неблагоприятно может сказаться на обработку "длинных" запросов. Поэтому на практике используют различные модификации данного метода.

Все дисциплины которое было рассмотрено является одночерговимы. В компьютерах, в ОС широко используют багаточергови дисциплины.

В таких дисциплинах организуется N очередей. Все новые запросы поступают в конец первой очереди. Первый запрос из очереди и (1 • и • N) поступает на обслуживание только тогда, когда все очереди от 1 до (и - 1) - й пустые. На обслуживание выделяется квант времени t_k. Если за это время обслуживания запроса завершается полностью, то он покидает систему. В обратном случае запрос который недообслуговано поступает в конец очереди с номером i + 1.

После обслуживания из очереди и система вирбирае для обслуживания запрос с пустая очереди с самым младшим номером. Таким запросу может быть следующий запрос из очереди и либо из очереди и + 1 (при условии, что после обслуживания запроса из очереди и последняя оказалась пустой). Новый запрос поступает в первую очередь (и = 1). В такой ситуации по истечении времени t_k которое выделено для обслуживания запроса из очереди и, начнется обслуживания запроса 1-й очереди.

Рис. 2.9. Схема приоритетной багаточерговои дисциплины обслуживания

Если система выходит на обслуживание заявок из очереди N, то они обслуживаются или по дисциплинам FIFO (каждая заявка обслуживается до конца), или круговым циклическим алгоритмом.

Данная система наиболее быстро обслуживает все короткие по времени обслуживания запросов. Недостаток системы заключается в непроизводительных затратах времени на перемещение запросов из одной очереди в другую.

В приведенной на рис 2.9 схеме реализовано багаточергова дисциплина, имеет определенный приоритет на обслуживание того или иного ресурса. основу этой

дисциплины представляет рассмотрена ранее багаточергова дисциплина. Новые запросы поступающие в систему не обязательно попадают в первый очередь. Они попадают в очередь в соответствии с приоритетами которых, определяемых параметрами процессов обслуживаемых.

В багаточергових дисциплинах возможны различные стратегии и системы по отношению к новым запросам, которые поступают в систему. Эти стратегии определяются дисциплинами обслуживания запросов с абсолютными и относительными приоритетами.

2.6.1.4 Обслуживание с абсолютным приоритетом

В багаточерговий дисциплине где запрос вновь поступает имеет определенный приоритет, используется обслуживания с абсолютным приоритетом. здесь приоритет определяется очередной (ее номером), и первыми обслуживаются запросы, которые имеют высокие приоритеты (с очереди с меньшим номером). Допустим, система обслуживает запрос из очереди с номером и, где 1 • и • N и в систему поступает более приоритетный запрос в очередь, к примеру, с номером и -1. В таких условиях обслуживание и го уровня прерывается и система начинает обслуживать запрос и-1 уровня. После окончания его обслуживание оказывается дообслуговування прерван запроса и го уровня.

В настоящее дисциплине это дильше увеличивается степень дискриминации по среднему времени ожидания в очереди между высоко - и низькоприоритетнимы запросами. Время ожидания високоприоритетних заявок сокращается, но опять за счет большего ухудшения в обслуживании низкоприоритетного заявок.

Стоимость за увеличение степени дискриминации, осложнения логики системы, а отсюда и ее реализации. Кроме того, с появляется проблема прерывания. Во - первых, появляется накладные расходы, необходимые для отработки прерываний процесса выполнения и перерасподилу ресурса.

Эти расходы при достаточной интенсивности прерываний могут стать очень слышны. Во - вторых, нужно выбрать наиболее правомочно правило о дообслуговування процессов прерываемых - когда выделять им снова ресурс, учитывать или нет, что ресурс уже используется прерывание, и т.д.

2.6.1.5 Обслуживание с относительным приоритетом

При даной дисциплине заявка, которая входит в систему, не вызывает прерываний обслуживания заявки, даже если последняя и менее приоритетна. В данном случае только после окончания обслуживания менее приоритетной заявки начнется обслуживание более приоритетной. Все сказанное касается дисциплины распределения ресурсов без учета взаимосвязи процессов. процессы, как правило, используют различные ресурсы и этот факт оказывает влияние на рассмотрение дисциплины распределения. На практике должна быть построена стратегия распределения, которая бы удовлетворяла не только в отношении некоторого конкретного ресурса, но и согласована с стратегией распределения других ресурсов.

Наряду с дисциплинами распределения, мы рассматриваем существуют и другие, содержание которых определяется спецификой ресурса распределяемой. Много таких оригинальных дисциплин имеет место при статическом и динамисному распределении между процессами ОП.

Лекция 9

2.7. концепция прерываний

2.7.1 Теория прерываний

Организация многопрограммного режима работы компьютера бозуеться на использовании прерываний. так, программа, которая обслуживается процессором, прерывается из-за отсутствия данных, которые нужно обрабатывать в оперативной памяти. Программа, которая обслуживается процессором, может быть прервана более приоритетной программой. С всего ризномаиття разным причинам прерываний нужно выделять два вида системных причин прерывания: первого и второго рода.

Системные причины прерываний первого рода, возникают в том случае, когда в процесса, который находится в, активном состоянии, возникает потребность получить некоторый ресурс или отказаться от него или выполнить над ресурсом какие - либо действия. Эти причины с появляются и тогда, когда процесс порождает, уничтожает и выполняет любые действия по отношению других процессов. При таких прерываниям возникает потребность в явной форме выражать требование на прерывание процессом самого себя. Требуется установление связи типа "что вызывает - либо вызывают".

Установление такой связи реализуется, как правило, в форме макрокоманд, которые представлены в пользовательской программе. При выполнении таких макропрограммы выполняется переключение процессора по обработке программы пакета на работу ОС, подготавливает и обеспечивает выполнение соответствующего прерывания. К этой группе относятся и так называемые внутренние прерывания, которые связаны с работой процессора и является синхронная с его операциями. К таким прерываний относится арифметическое переполнение, снижение порядка в операциях с плавающей запятой, обращение к защищенному массива ОП и другое.

системная причина прерывания второго рода обусловлена необходимостью проведения синхронизации между параллельными процессами. Процессы, порожденные и подчиненные ОС, по мере их окончания или при какой-то другой ситуации вырабатывают сигнал прерывания. К примеру, по мере окончания внешнего процесса пересылки массива данных с МД в ОП вырабатывается сигнал прерывания. этот сигнал прерывает выполнение программы обслуживаемого процессором, которое выполняется "без ведома", или асинхронным.

Пусть в момент времени t_1 возникло прерывание второго рода по причине окончания работы канала по передаче массива данных в оперативной памяти. Было прервано процесс Р1, который не имеет никакого отношения к этому массиву данных. Пересылка этого массива запретил ранее процесс Р2, прервав самого себя по системной причине первого рода. Предположим, что в системе есть еще один процесс Р3, который находится в момент рассматриваемого в состоянии готовности.

После отработки прерывания процесс Р1 будет находиться в состоянии готовности, так как у него есть все ресурсы, кроме процессора, которой у него отобрали без его желания. Процесс Р2 также будет переведен в состояние готовности по состоянию ожидания, так как осуществилась событие, которого он ожидал - массив записано в ОП и дальше ему доступен. Отсюда, система управления процессами некоторым правилом должна выбрать, какой из процессов перевести в активное состояние.

При обработке каждого прерывания нужно выполнять следующие действия:

- восприятие запроса на прерывание;
- запоминания состояния прерванного процесса, которое определяется прежде все значением счетовода команд. Оно должно отражать и признак команды, после выполнения которой возникло прерывание, содержание регистров общего назначения, режим работы процессора (с позиции обслуживания процессором пользовательской или системной программы в момент прерывания)

• • передача пользования программы которая прерывается, для чего в счетовод команд СК должна быть занесена адрес, который, как правило, является уникальной для каждого типа прерывания;

• • обработка прерываний;

• • восстановление нормальной работы.

В большинстве компьютеров этапы 1-3 реализуемых аппаратными средствами, а этапы 4 и 5 - ОП, и й блоком программ обработки прерываний. Наряду с описанными функциями, которые реализуются на аппаратном уровне, обработка прерываний реализуются в ОС на уровне системных программ. такие функции представлены в программах прерывающих. Их количество определяется количеством уровней прерываний. В терминологии, принятой в некоторых ОС, эти прерывая программы называют обработчиками или супервизорами прерываний. К примеру, обработчик внешних прерываний, обработчик программных прерываний. Прерывая программа, которая обслуживает запросы по уровню ввода - вывода называется супервизора ввода - вывода. Прерывая программы по уровню СО называют программы супервизора.

Каждая прерывая программа соответствует определенному уровню прерываний. Последний объединяет ряд запросов прерываний, какие обслуживаются одной прерывая программой.

При таком решении кроме выбора прерывая программы нужно выбирать на уровне наиболее приоритетные запросы, которые вимогають обслуживания. Эта функция часто реализуется в системных прерывая программах. Структура такой программы можно проиллюстрировать следующей схеме.

На ней реализовано дисциплину циклического опроса для выбора наиболее приоритетного запроса для обслуживания. Легко реализуется в этом варианте и дисциплина выбора запросу с относительным приоритетом. Реализация дисциплины абсолютным приоритетом будет вимогаты обращения к другой структуры прерывая программы.

После восстановления нормальной работы компьютера. Это восстановление выполняется передачей управления системной программе, которые выполняет функции диспетчера. Последняя получает управление в тех случаях, когда результатом обработки прерывания является выведение некоторой задачи по состоянию ожидания в состояние готовности или установлением текущей задачи в состояние ожидания.

Рис. 2.10 Структура прерывая программы, которая обеспечивает выбор запросов в уровне прерывания

Программа «Диспетчер" анализирует состояние задач в системе и передает управление готовой к выполнению задачи с наивысшим приоритетом. Если задачи в состоянии готовности

отсутствуют, тогда ЦБ переводится в состояние ожидания до возникновения очередного прерывания.

2.7.2. Сигналы, прерывания и время в Unix

Сигналы (Signal) представляют собой одну из форм взаимодействия й между процессами (Interprocess Communication, или IPC) - способа передачи информации от одного процесса другому. Но передавать возможно не очень много информации - вместе с сигналом нельзя передавать сообщения и даже итендификацию отправника; все, что в данной ситуаций й имеется, - это факт отправки сигнала. Сигналы практически непригодные для детализации двунаправленных взаимодействий. Кроме того, в рамках существующих ограничений получатель сигнала не должен каким - либо образом отвечать на него и даже может позволить себе проигнорировать большую часть сигналов.

Тем не менее, несмотря на такие ограничения, сигналы остаются мощным и полезным механизмом, который используется, даже чаще других. Каждый раз когда завершается некоторый процесс или розименовуеться указательных со значением NULL, при каждом нажатии Ct rl + C или применения программы Kill, генерируется какой-то сигнал.

Прерывания (Intrrupts), как пояснено в коде Linux, похожие на сигналы для ядра. Прерывания направляются ядру от аппаратных устройств похожих на диске с целью сообщения ядра о, что то или иное устройство вимогають к себе некоторого внимания. Одним из важных источников аппаратных прерываний есть таймер, который Периодические сообщает ядро о прохождении времени. Прерывания могут также генерироваться пользовательскими процессами, или программно.

2.7.2.1 Короткие тезисы о блокировании

Главная идея, которая связана с блокировкой, заключается в защите доступа к ресурсам используемых совместно - файла, области памяти или фрагмента кода, который должен выполняться строго одним ЦБ в один и тот же момент времени. В случае однопроцессорной (ЦР) системы ядру Linux блокировки не нужны, так как при его написании как раз и ставилась цель избежать нужного рода ситуаций. Но, в случае с мультипроцессорными (SMP) системами иногда один процессор хочет освободиться от нежелательного воздействия со стороны другого процессора.

Вместо размещения всех обращений к функциям блокировки с #ifdef применен другой подход и были разработаны отдельно макросы для UP (по большей степени пустые) и для SMP - систем (содержащие реальный код), после чего все макросы были сведены в файл include (fsm - i386) spin-lock.h. В результате другой код, если иметь в виду блокировки, выглядит одинаково как для UP -, так и для SMP - систем, но приводит к совершенно разным эффектов.

2.7.2.2 Сигналы

Ядро Linux разделяет сигналы на две категории:

- Сигналы нереального времени (NoRealtime). Наиболее традиционные сигналы Unix, такие как SIGSEGV, SIGHUP и SIGKILL.
- Сигналы реального времени (Realtime). Они имеют несколько отличные характеристики по сравнению с их аналоги не реального времени. Сигналы реального времени имеют смысл, связанный с конфигурированием процессов, как в случае с сигнала реального SIGUSR1 и SIGUSR2, кроме того, вместе с ними поступает и дополнительная информация. Они также подвергаются постановке в очередь, так что если много екземплярив сигнала поступает до того, как обробляется первый поступивший то все сигналы будут доставлены по назначению. Последнее не справедливо для сигналов нереального времени.

Все Unix - системы и язык программирования ANSI C поддерживают API signal, с помощью которого можно определить методы обработки отдельных сигналов. Прототип функции API имеет следующий вид:

```
#  include <signal .h>
```

vaid (* signal (int signal_num, vaid (* handler) (int))) (int) Аргументами данного API является signal_num - идентификатор сигнала (например, SIGINT, SIGTERM и т.д.), который определен в заголовке <signal .h> и handler - указатель на определенную пользователем функцию - обработчик сигнала. Эта функция должна принимать целочисленный аргумент. Никаких значений воне не возвращает.

Например программы где выполняется попытка перехвата сигнала SIGTERM, игнорируется сигнал SIGINT, а по сигналу SIGSERV выполняется действие, предусмотренное по заказу. API pouse приостанавливает выполнение процесса вызывающего до тех пор, пока он не будет прерван якимнебудь сигналом и соответствующий обработчик сигнала вернет результат:

```
# include <iostream.h>
# include <signal .h>
   / * Функция - обработчик сигналов * / void
   catch_sid (int sig_num)
        {
         signal (sig_num, catch_sig) count << "catch_sig:" <<
           sig_num << endl; }

        / * Главная функция * / int
main () {

         signal (SIGTERM, catch_sig) signal
         (SIGINT, SIG_IGN) signal (SIGSEGV,
         SIG_DFL) pause ();
                          / * Ожидание сигнала * /
}
```

SIG_IGN и SIG_DFL - это макросы, определенные в заголовке <signal.h>

```
# define SIG_DEL void (*) (int) 0
# define SIG_IGN void (*) (int) 1
```

SIG_IGN - задает сигнал, будет проигнорировано. Будучи отправленном в процесс, такой сигнал просто отбрасывается, а сам процесс не прерывается.

SIG_DEL - сигнализирует, что нужно выполнить действие по заказу. значение API signal, что возвращается - указатель на предыдущий обработчик данного сигнала. Его можно использовать для восстановления обработчика сигнала.

```
#  include <signal .h> int
main () {

void (* old_handler) (int) = signal (SIGINT, SIG_IGN) / * Выполнить
важные операции обработки * / signal (SIGINT, old_handler)

/ * Восстановить предыдущий обработчик сигнала * /
```

2.7.2.3. сигнальная маска

Каждый процесс в UNIX - системе и в POSIX.1 имеет сигнальную маску, которая определяет, какие сигналы с отосланных процессов блокируются. Разблокировка и обработку блокированного сигнала выполняют процессы - получатели. Если сигнал задан как таковой игнорируется и блокируется, то решение вопроса о том, будет ли он при передаче в процесс отвергнутым или переведенным в режим ожидания, зависит от реализации ОС.

При создании процесса следует сигнальную маску своего отца, но ни один сигнал, который ожидает обработку родительским процессом, к порожденного процесса не пропускается. Процесс может запрашивать и создавать сигнальную маску с помощью API sigprocmask:

```
#  include <signal .h>
int sigrocmask (int cmd, const sigset_t * new_mask, sigset_t * old_mask)
```

Аргумент new_mask определяет набор сигналов, которые будут добавлены к сигнальной маски процесса вызывающего или удалены из нее, если сигнал задан как таковой игнорируется и блокируется, то решение вопроса о том, будет ли он при передаче в процесс отвергнутым или переведенным в режим ожидания зависит от реализации ОС. При создании процесс следует сигнальную маску своего отца, но ни один сигнал, который ожидает обработку родительским процессом, к продленного процесса не пропускает.

В случае успешного завершения вызова sigprocmask возвращает 0, в противном случае возвращает -1. причиной неудачи может стать неправильное задание адресов в аргументах new_mask и (или) old_mask.

Тип данных sigset_t определяется в заголовке <signal .h>. Данные этого типа является набором битов, каждый из которых является флагом, который соответствует одному определенному в данной системе сигнала.

2.7.2.4. функция sigaction

Интерфейс дополнительного программирования sigaction заменяет API signal. Подобно API signal, он вызывается процессом для задания метода обработки каждого сигнала, с которым собираются работать. оба интерфейсы положительного программирования возвращают указатель на предыдущий метод обработки данного сигнала. Кроме того, API sigaction позволяет определить дополнительные сигналы, которые также будут блокироваться при обработке сигнала signal_num.

Прототип API sigaction имеет следующий вид:

```
# include <signal .h>
```

```
int sigaction (int signal_num, struct sigaction * action, struct
sigaction * old_action)

Тип данных struct sigaction определяется в заголовке <signal .h> Struct sigaction {


void (* sa_handler) (int); sigaset_t
sa_mask; int sa_flag; };
```

поле sa_handler соответствует второму аргументу функции signal. У него может быть занесет но значение SIG_ SIG_DFL, как определенная пользователем функция - обработчик сигнала. поле sa_mask рядом с сигналами, указанными в данный момент в сигнальной маске процесса, и сигналом signal_num задает дополнительные сигналы, процесс будет блокировать при обработке сигнала signal_num.

Аргумент signal_num показывает, какое действие по обработке сигнала определен в аргументе action. Предыдущий метод обработки сигнала для signal_num будет возвращен с помощью аргумента old_action, если это не NULL - указатель. Если аргумент action является NULL - указателем, то метод обработки сигнала процесса вызывающая для signal_num остается прежним.

Пример использования функции sigaction. В этом примере сигнальная маска процесса содержит сигнал SIGTERM. Затем процессом определяется обработчик для сигнала SIGINT.

Указывается, что сигнал SIGSEGV должен быть блокированным при обработке процессом сигнала SIGINT. Затем процесс приостанавливает свое выполнение с помощью API pause.

```
# include <iostream.h>
# include <stdio.h>
# include <unistd.h>
# include <signal .h> void
calime () {

cout << "catch signal" << endl; }

 int main () {

sigset_t              sigmask;
struct sigaction action, old_action; sigemptyset (&
sigmask)
if (sigaddset (& sigmask, SIGTERM == - 1)) sigprocmack
(SIG_SETMASK, & sigmask 0) == - 1) perror ( "set signal mask")
sigemptyset (& action.sa_mask) sigaddset (& action.sa_mask,
SIGSEGV)

# ifdef SOLARIS_25
action.sa_handler = (void (*) (int)) callme;
# else
action.sa_handler = callme;
# endif
action.sa_flags = 0
if (sigaction (SIGILL, & action, & old_action) == - 1) perror ( "sigaction") pause (); / * Ожидать
прерывания сигналом * / returm 0;
```

Если в процессе генерируется сигнал SIGINT, тогда ядро сигнала устанавливает сигнальную маску процесса и блокировки сигналов SIGTERM, SIGINT и SIGSEGV. Затем оно настраивает процесс на выполнение функции - обработчика сигнала callme. Когда функция callme возвращает результат сигнальная маска процесса восстанавливается и содержит только сигнал SIGTERM, а процесс продолжает перехватывать сигнал SIGILL.

Поле sa_flag в структуре st ruct sigaction используется для задания специальных методов обработки определенных сигналов. В стандарте POXIX. Установлено только два значения этого поля: 0 или SA_NOCHLDSTOP. Флаг NOCHLDSTOP - это целое число, определенное в заголовке <signal .h>. данный флаг может использоваться, когда аргумент signal_num важно SIGCHLD. Действие флага SA_NOCHLDSTOP заключается в том, что ядро посылает сигнал SIGHLD в процесс тогда, когда порожденный им процесс завершен, а не тогда, когда был остановлен. С другой стороны, если значение sa_flag в вызове sigaction равен 0, тогда ядро посылает сигнал SIGCHLD в процесс вызывающего вне зависимости от того, завершено порожденный процесс или оставлено.

2.7.2.5. Сигнал SIGCHLD и API waitpid

Когда порожденный процесс завершается или останавливается, ядро посылает сигнал SIGCHLD в его родительский процесс. В зависимости от того, какой способ обработки сигнала SIGCHLD выбирает родительский процесс, могут происходить различные события и:

1. Родительский процесс выполняет для сигнала SIGCHLD действие по умолчанию: на отличие от большинства сигналов SIGCHLD Не прерывайте родительский процесс. Он действует на родительский процесс только в том случае, если поступает в то же время, когда данный процесс приостанавливает системным вызовом waitpid.

Если это произошло, то родительский процесс "просыпается", API возвращает ему код завершения и идентификатор порожденного процесса, а ядро освобождает позицию в таблице процессов, которая была выделена для порожденного процесса.

Таким образом, при каждой такой настройке родительский процесс может многократно вызывать API waitpid чтобы дождаться завершения каждого порожденного процесса который им создано.

2. Родительский сигнал игнорирует сигнал SIGHLD: последний отбрасывается и родительский процесс не будет задет даже если он выполняет системный вызов wai tpid. Эффект такой настройки заключается в том, что если родительский процесс вызывает waitpid, то этот API будет приостанавливать родительский процесс до тех пор, пока не завершатся все порожденные им процессы. Затем ядро освобождает позиции в таблице порожденных процессов и API возвращает -1 в родительский процесс.

3. процесс перехватывает сигнал SIGHLD: функция-обработчик сигнала вызывается в родительском процессе при завершении порожденного процесса. Если сигнал SIGCHLD поступает, когда родительский процесс выполняет системный вызов wai tpid, то после возврата из функции обработчика сигнала этот API может быть перезагружен для определения статуса завершения порожденного процесса и освобождение в таблице процессов соответствующей ему позиции. С другой стороны, в зависимости от того, какой метод обработки для сигнала SIGCHILD выбрано в родительском процессе, API может быть прерван, а позиция порожденного процесса в таблице процессов освобождена не будет.

Взаимодействие между SIGCHLD и API wait осуществляется таким же образом, как между SIGCHLD и API waitpid.

2.7.2.6. API sigsetjmp и siglongjmp

API sigsetjmp и siglongjmp имеют то же назначение, что и API setjmp, а также longjmp и setjmp, и sigsetjmp делают пометку одной или нескольких позиций в пользовательской программе. Эта программа затем может вызвать API longjmp или siglongjmp для перехода в одну из отмеченных позиций. Таким образом, данные интерфейсы обеспечивают возможность передавать управление от одной функции к другой.

Пример прототипа функции API

```
#  include <setjmp.h>
int sigsetjmp (sigjmppbuf env, int save_sigmask) int siglongjmp
(sigjmpbuf env, int ret_val)
```

API sigsetjmp и siglongjmp создан для обработки сигнальных масок. Решение вопроса о том, будет ли сигнальная маска сохранена и восстановлена при вызове соответственно API setjmp и longjmp, зависит от реализации ОС.

API sigsetjmp работает почти также, как API set jmp, за исключением того, что он второй аргумент, save_sigmask должна сигнальная маска процесса, вызывающего быть сохранена в переменной env.

Так, если аргумент save_sigmask не равно 0, то сигнальная маска процесса вызываемого сохраняется; в обратном случае она не сохраняется.

API siglongjmp выполняет все те же операции, что и API longjmp, но кроме того, восстанавливает сигнальную маску процесса вызывающего, если и была сохранена в переменной env. аргумент ret_val задает значение возвращающей API sigsetjmp, когда он вызывается функцией siglongjmp. Его значение должно быть ненулевым числом; если же это значение равно 0, то API siglongjmp сбрасывает его в 1.

API siglongjmp обычно вызывается из определенных пользователем функций обработки сигналов. Сигнальная маска процесса модифицируется при вызове обработчика сигнала и API siglongjmp должен быть вызван (если пользователь не хочет восстановить вызывания с того места, где программа была прервана сигналом) для того, чтобы обеспечить нужное восстановления сигнальной маски процесса при "прыжке" из функции обработки сигнала.

Пример программы. В ней используется API sigset jmp и siglongjmp. Программа устанавливает свою сигнальную маску, так, чтобы она пропускала сигнал SIGTERM, а затем настраивает ловушку для этого сигнала. Далее программа вызывает sigsetjmp для сохранения в глобальной переменной env позиции, в которой было прервано выполнение программы. При этом sigsetjmp возвращает ненулевое значение, когда он вызван непосредственно в пользовательской программе, а не с применением sigglongjmp. Программа приостанавливает свое выполнение с помощью API pause. Когда пользователь прерывает процесс с помощью клавиатуры, тогда вызывается функция callme. Функция cal lme вызывает API siglongjmp для передачи управления программой обратно в функцию sigsetjmp (ее вызов выполняется в функцию main), которая теперь возвращает значение, равное 2.

```
#  include <iogtream.h>
#  include <stdio.h>
#  include <unistd.h>
#  include <signal .h>
#  include <setjmp.h>
sigjmp_buf env;
void callme (int sig_num) {

 cout << "catch signall:" << sig_num << endl; siglongjmp (env,
2);} int main () {


 sigset_t sigmask;
 struct sigaction action, old_action;

 sigemptyset (& sigmask)
 if (sigaddset (& sigmask, SIGTERM) == - 1
       sigprocmask (SIG_SETMASK, & sigmask 0) == - 1)
```

```
   perror ( "set signal mask") sigemptyser (&
  action.sa_mask) sigaddser (& action.sa_mask,
  SIGSEGV)
# ifdef SOLARIS_25
   action.sa_handler = (void (*) (int)) callme;
# else
   action.sa_handler = (void (*) ()) callme;
# endif
  action.sa_flage = 0
  if (sigaction (SIGINT, & action, & old_action, = -1) perror ( "sigaction") if (sigsetjmp (env, 1)! = 0) {


      cerr << "Return from signal interruptio \ n"; return 0; }


  else cerr << "Return from first time sigsetjmp is called \ n";
  pause (); // ожидать прерывания сигналом (например вводимые с помощью клавиатуры
```

2.7.2.7. API kill

Процесс может отсылать сигнал в родительский процесс с помощью API kill. Это простое средство предназначено для осуществления взаимодействия между процессами и управления процессами. Процесс - отправитель и процесс-получатель должны быть связаны таким образом, чтобы реальные или эффективный идентификатор владельца процесса - отправителя совпадал с реальным или эффективным идентификатором владельца процесса-получателя или чтобы процесс-отправителя имел права привилегированного пользователя. Отдельно, родительский и порожденный процессы могут отсылать сигналы друг другу с помощью интерфейса ki ll.

API kill определена в большинстве систем UNIX и входит в стандарт PSIX.1. Прототип функции этого API имеет следующий вид:

```
#  include <signal .h>
int kill (pid_t pid, int signal_num)
```

Аргумент signal_num - номер сигнала который должен быть отправлен в один или несколько процессов, обозначенных аргументом pid. При успешном выполнении API kill возвращает 0, а в случае неудачи - 1.

Пример программы показана возможность реализации API kill:

```
#  include <iostream.h>
#  include <stdio.h>
#  include <unistd.h>
#  include <string.h>
#  include <signal .h>

int main (int argc, charr ** argv) {

 int pid, sig = SIGTERM if (argc
 == 3) {

    if (sscanf (arv [1], "% d", & sig)! = 1) {/ *
     получить номер сигнала * /
      perr << "Invalid signal:" << argv [1] << endl; return 1; }


  argv ++, argc -; }

 while (- argc> 0)
  if (sscanf (* ++ argv, "% d", & pid) == 1) {/ * получить
     идентификатор процесса * / if (kill (pid, sig) == - 1)
     perror ( "kill") }


  else cerr << "Invalid pid:" << argv [0] << endl; return 0; }
```

Синтаксис вызова команды kill: Kill [-
<signal_num>] <pid> ...

где <signal_num> может быть целым числом или именем сигнала, которое определяется в заголовке <signal .h>. Аргумент <Pid> - это целое число идентификатора процесса. может

быть указано одно или несколько значений <Pid и команда kill будет отсылать сигнал <signal_num> в процессы с указанным <Pid>.

2.7.2.8. API alarm

API alarm может быть вызван процессом для того, чтобы требовать от ядра отправить через определенное количество секунд реального времени сигнал SIGALARM. Однако функции API alarm определены следующим образом:

```
#  include <signal .h>
unsigned int alarm (unsigned int time_interval)
```

Аргумент time_interval - это время в секундах, по истечении которого ядро отправит сигнал SIGALARM в процесс вызывающее. Если значение time_interval равен 0, будильник выключается.

Значение которое возвращает API alarm - это число секунд, которое осталось до срабатывания таймера процесса и соответствующих установке, которая была сделана в предыдущем системном вызове alarm. Установление выполненные в результате предыдущего вызова API alarm, отменяются, и тацмер процесса срабатывает при каждом новом вызове alarm. Таймер процесса не передается в порожденный процесс, созданный функцией fork, но процесс, созданный функцией exec сохраняет значение таймера, которое было задано до вызова API exec.

API alarm может быть использован для реализации API sleep:

```
#  include <stdio.h>
#  include <signal .h>
#  include <unistd.h>

void wakeup () {};
unsigned int sleep (unsigned int timer) {

   struct sigaction action;
# ifdef SOLARIS_25
          action.sa_handler = (void (*) (int) wakeup)
# else
          action.sa_handler = wakeup;
# endif
          action.sa_flags = 0
          sigemptyset (& action.sa_mask) if (sigaction (SIGALARM, &
          action 0) == - 1) {

           perror ( "sigaction") return 1; }

(Void) alarm (timer) (Void)
 pause (); return 0; }
```

API sleep приостанавливает процесс вызывающее на определенное количество секунд. Процесс "просыпается", когда в прошедший превышает значение timer или когда он прерывается сигналами.

В приведенном примере функция sleep настраивает обработчик сигнала на сигнал SIGALARM, вызывается API alarm для передачи в ядро запроса на отсылка указанного сигнала (после некоторого времени, заданного аргументом timer) и, наконец, приостанавливает его исполнения с помощью системного вызова pause. функция -

обработчик сигнала wakeup вызывается тогда, когда SIGALARM передается в процесс. После возвращения из этой функции системный вызов pause прерывается, выполняется возврат в процесс вызываемое с функцией sleep.

2.7.2.9. прерывание

Прерывание в Linux можно называть весьма условно, поскольку они прерывают нормальную работу системы.

Как и в случае прерываний системных функций, аппаратные прерывания могут вызвать переход в режим ядра и обратно.

Когда прерывания выполняется при выполнении процесса пользователя, система переходит в режим ядра, и ядро отвечает на прерывание. Затем ядро возвращает управление процесса пользователя, продолжается с той точки, где он был прерван.

Одна отмена аппаратных прерываний от прерываний системных функций заключается в том, что аппаратные прерывания могут выполняться в то время, когда ядро уже работает в режиме ядра. С системными вызовами такое выполняется редко - обычно ядро не затрудняет себя запуском прерываний системного вызова, поскольку может просто непосредственно вызвать целевую функцию ядра. Если прерывания выполняется в то время, когда система находится в режиме ядра, результат во многом аналогичен тому, как если бы оно возникло в пользовательском режиме. Разница лишь в том, что временно прерывается выполнения процесса самого ядра,

а не процесса пользователя.

Когда ядро не хочет быть временно прерванным, оно выключает и вызывает прерывание с помощью функции cli и sti

```
#  define _sti () _asm_ _volatile_ ( "sti" ::: "memory")
#  define _cli () _asm_ _volatile_ ( "cli" ::: "memory")
```

для UP - версии

```
void _global_cli (void) {

unsigned int flags;
_save_flags (flags)
if (flags & (1 << EFLAGS_IF_SHIFT)) {

int cpu = cmp_processor_id (); _cli ();

if (! local_irg_count [cpu]) get_irglock
 (cpu) }}


void _global_sti (void) {

int cpu = smp_processor_id (); if (!
  local_irg_count [cpu]) releace_irglock
  (cpu) _sti ()}
```

Для SMP версии эти функции названы в соответствии с базовыми инструкциями x86: cli значит "Clear the Interrupt flag" ("Очистить флаг прерываний»), а sti - "Set the Interrupts flag" ("Установить флаг прерываний»).

Их работа соответствует названию: процессор имеет флаг "прерывания разрешены", который разрешает прерывание, если установлен, и запрещает их, если снят. Таким образом, прерывание выключаются посредством выделения флага с помощью

функции cli и включаются после этого с помощью функции sti. В коде UP вместо этих функции можно вызывать макросы __cli и __sti, приведенные ранее.

Естественно порты ядра для платформы, какие отличные от x86, будут использовать другие базовые инструменты - в этих архитектурах функции cli и sti реализовано по другому.

2.7.2.10. IRQ

IRQ - interrupt request (запрос прерываний) - это сообщение о прерывании, которое определяется с аппаратного устройства ЦБ. В ответ на IRQ ЦБ выполняет переход к специальной адреса - interrupt service routine (Подпрограмма обслуживания прерываний) (ISR), что чаще называется обработчика прерываний - который был ранее зарегистрирован с IRQ ядром. Обработчиком прерываний - это функция, которую ядро выполняет для обслуживания прерывания; возврат из обработчика прерывания приводит к продолжению выполнения с того места, где оно было прервано.

IRQ нумеруется и каждый аппаратное устройство в системе связывается с номером IRQ.

Например, в архитектуре IBM PC IRQ 0 связан с аппаратным таймером, который генерирует 100 прерываний в секунду. Связывание номера IRQ с устройством позволяет ЦБ установить, устройство сгенерированный каждое прерывание, и, отсюда, позволяет ему выполнить переход к нужному обработчике прерываний. (В некоторых случаях номер IRQ может совместно использоваться устройствами в системе, хотя это и не очень распространено.)

2.7.2.11. нижние половины

Нижняя половина обработчика прерываний - та его часть, которую нужно выполнять мгновенно. Для некоторых прерываний может совсем не понадобится ее выполнять.

Нужно отметить, что каждый данный обработчик прерываний делится на верхнюю и нижнюю половины; верхняя половина вычислений выполняется мгновенно в момент прерывания, в то время как нижняя половина (если она есть) откладывается. Это достигается путем реализации верхней и нижней половины отдельными функциями и путем различной обработке.

В целом, верхняя половина принимает решение о необходимости выполнения нижней половины. Очевидно, что все не может быть отложено, не входит в нижнюю половину, но все, что может быть отложено, является кандидатом для исключения в ней.

Видно что такое осложнение вызывает задержку. Одной из причин такого осложнения является попытка минимизировать общую продолжительность прерываний. Ядро Linux определяет два вида прерываний: быстрый и медленный, и одна из разниц между ними заключается в том, что медленные прерывания сами могут быть прерванными, а быстрые - нет. Итак, во время обработки быстрых прерываний все остальные поступающих - как медленные, так и быстрые - должны просто ожидать своей очереди. Чтобы обработать эти прерывания как можно раньше, ядро откладывает как можно больше вычислений в ниже половины.

Вторая половина заключается в том, что на самом нижнем уровне чипа контроллера прерываний указывается отключить конкретный IRQ, который подлежит обработке, пока ядро выполняет верхнюю половину (это имеет отличие от отключения на уровне ЦБ, отличают быстрые и медленные прерывания). Нежелательно, чтобы эта ситуация продолжалась дольше, чем нужно, поэтому только наиболее критическая в смысле времени выполняется в верхней половине, и все остальное переносится в нижнюю половину.

Нижняя половина не обязательно вызывается один раз для каждого прерывания. Вместо этого, верхняя половина (или, иногда, которая либо другая часть кода) просто "намишуе" нижнюю половину, устанавливая разряд, указывающий на

необходимость выполнения нижней половины. Если нижняя половина замечается снова, когда уже есть замеченной, она просто не изменилось: ядро обслуживает ее когда может это сделать. Если для данного устройства 100 прерываний исполнится раньше, чем ядро получит возможность выполнить нижнюю половину, верхняя половина будет выполнена 100 раз, а нижняя - 1 раз.

Иногда нижние половины имеют в ядре название "мягких IRQ" или "обработчиков мягких прерываний" что помогает объяснить некоторые имена файлов и другие сроки.

2.7.2.12. структуры данных

Рассмотрим важные структуры данных прерываний и нижние половины. В Linux имеется независимый от архитектуры файл заголовке

linux / interrupt .h который определяет структуру st ruct i rqaction

```
struct irqaction {
void (* handler) (int, void *, struct pt_regs *) unsigned long
flags; unsigned long mask; const char * name; void * dev_id;



struct irqaction * next;
};
```

которая представляет действие, которое ядро повино начать при получении определенного IRQ. Эта структура имеет следующие члены:

handler - указатель на функцию, которая начинает определенное действие в ответ на прерывание.

flags - это член получается из того же набора, и sa_flags

mask - это член не используется ни одним из кодов x86 или архитектурно - независимыми кодами (исключая установления его таким равной 0);

name - имя, которое связано с устройством, который генерирует прерывание. Поскольку IRQ может использоваться совместно несколькими устройствами, этот член может помочь различить их при выводе списка для читателя - человека.

dev_id - уникальный идентификатор типа устройство - каждая модель каждого аппаратного устройства, поддерживается Linux, имеется идентификатор устройства, присвоенный производителем и записан в его член. Он состоит из большого количества определений в блоке # define который имеет примерно следующий вид:

```
#  define PCI_DEVICE_ID_S3_836 0x8880
#  define PCI_DEVICE_ID_S3_928 0x88h0
#  define PCI_DEVICE_ID_S3_864_1                0x88c0
#  define PCI_DEVICE_ID_S3_864_2                0x88c1
(Идентификат устройств для PCI видиоплат на основе чипов S3)
```

Хотя dev_id является указателем, он ни на что не указывает и попытки раскрыть его не будет ошибкой. Значение имеет только его разрядный шаблон.

next - указатель на следующую структуру st ruct i rqaction в очереди, если IRQ используется совместно, и, отсюда, этим членом является NULL.

Рис. 2.11 Структура данных связанные с прерываниями

2.7.2.13. Время на таймер

Время и Linux (UNIX) и управления таймером нужно рассматривать на двух уровнях - ядро и API.

На уровне ядра функция прерывания таймера timer_interupt связывается с IRQ 0

```
. . .
setup_x86_irq (co_IRQ_TIMER, & irg 0);
#  else
    setup_x86_irq (0, & irq 0);
#  endif ...
```

Переменная, которая здесь используется irq 0 определено как static struct

```
irqaction irq0 =
 {Timer_interrupt, SA_INTERRUPT, 0, "tumer", NULL, NULL};
```

С помощью использования функции init_bh extern inline void

```
initbh (int nr, void (* routine) (void)) {

bh_base [nr] = routime;
atomic_set (& bh_mask_count [nr], 0); bh_mask | =
1 << nr; }
```

которая регистрируется как нижняя половина таймера init_bh

```
(TIMER_BH, timer_bh)
```

когда IRQ 0 запускается, timer_interrupt считывает некоторые значения с счетовода отметок таймера ЦБ, если таковые имеются, а затем вызывает функцию

```
do_timer_iterrupt
.  .  .
static inline void do_timer_interrupt (int irq, void * dev_id, struct pt_regs * regs) {

#  ifdef CONFIG_VISWS / * clear
the interrupt * /
co_cpu_write (CO_CPU_STAT, co_cpu_read (CO_CPU_STAT) & ~ CO_STAT_TIMEINTR)

# enif
  do_timer (regs)
# ifndef _SMP_
  if (! User_mode (regs)) x86 do_profile
  (regs-> eip)
# else
  if (! smp_found_config)
  smp_local_timer_interrupt (regs)
# endif
  if ((time_status & STA_UNSYNC) == 0 && xtime.tv_sec> last_rtc_update + 660 && xtime. tv_usec> = 500000 -
((unsigned) tick) / 2 && xtime. tv_usec <= 500000 + ((unsigned) tick / 2)

{If (set_rtc_mmss (xtime. Tv_sec) == 0)
 last_rtc_update = xtime. tv_sec; else

 last_rtc_update = xtime. tv_sec-600; }

.  .  .
```

Помимо выполнения других задач, эта функция вызывает функцию do_timer, которая интересна частью прерывания таймера

```
void do_timer (struct pt_regs * regs) {

1. (* (Unsigned long *) & jiffies) ++;
2. lost_ticks ++;
3.mark_bh (TIMER_BH)
4. if (! User_mode (regs))
lost_ticks_system ++;
5. if (tg_timer)
mack_bh (TQUEUE_BH) }
```

1. Обновляет глобальную переменную ji ffies, которая отслеживает количество тиков системных часов, которые возникли с момента начальной загрузки. В действительности выполняется подсчет тиков таймера, которые возникли с момента установления прерываний таймера, которые не совпадают с моментом загрузки системы.
2. Устанавливает количество "потерянных" тиков таймера - тиков, которые не обработанные нижней половиной.
3. Верхняя половина викконуеться, поэтому ее так нижняя половина помичуеться для скорейшего выполнения.
4. Кроме отслеживания общего количества тиков таймера, которые возникли с момента последнего выполнения нижней половины таймера, необходимо знать, сколько с этих тиков возникло в режиме системы. Значение lost_ticks_system увеличивается, если процесс выполняется в режиме системы (ядра), а не в режиме пользователя.
5. Если любые задачи ждут в очереди таймера, нижняя половина очереди таймера получает пометку как готова к выполнению.

```
timer_bh
static void timer_bh (void) {

   update_times ();
  run_old_timers ();
  run_timer_list (); }
```

Эта нижняя половина таймера. Она вызывает функции, предназначенные для восстановления связанных со временем статистических данных процесса и самого ядра и для обслуживания ядра, как в старом, так и в новом формате.

Коротко о некоторых функции ядра, связанные с таймером:

update_times - в основном эта функция обновляет статистические данные: она вычисляет среднее загрузки системы, обновляет глобальную переменную, которая отслеживает текущее время, и обновляет исчисленный ядром объем времени ЦП, который может потребоваться текущем процесса

update_wall_time- обновляет глобальную переменную xtime, по возможности синхронизируя ее с реальным временем с целью соответствии с протоколом Network Time Protocol

calc_load - вызывается из верхней половины таймера и обновляет примерный время загрузки системы. Функция, в основном используется для сохранения в памяти этого числа.

run_old_timers - ядро имеет устаревшую возможность, подобный нижним половинкам, которая заключается в следующем: функции ядра могут регистрироваться в специальной таблице и вызываться при достижении этого времени. Это обеспечивает функция, которая вызывает другие функции связано с поддержкой старого кода. Все что она делает, проходит по

списком элементов массива timer_table и обращается к функциям, время вызова которых наступил.

API уровень таймера использует специальные APE функции. Функции й sleep, приостанавливает процесс на определенное время - это один из вариантов использования API alarm. Чаще alarm используется для установки в процесс интервального таймера (interval timer). С помощью интервального таймера планируется выполнение процессом определенных задач через фиксированные интервалы времени, осуществляется синхронизация выполнения той или иной задачи.

Пример программы, которая устанавливает интервальный таймер реального времени, используя интерфейс alarm:

```
# include <stdio.h>
# include <unist .h>
# include <signal .h>
# define INTERVAL 5 void callme
(int sig_no) {

alarm (INTERVAL)
/ * Выполнить запланированное задание * /}

int main () {

struct sigaction action; sigemptyset (&
action.samask)
# ifdef SOLARIS_25
    action.sa_handler = (void (*) (int)) callme;
# else
    action.sa_handler = (void (*) ()) callme;
# endif
action.sa_flags = SA_RESTART; if (sigaction
(SIGALRM, & action 0) == - 1) {

 perror ( "sigaction") return 1;
}

if (alarm (INTERVAL) == - 1) perror (
 "alarm") else while (1) {


 / * Выполнить обычную работу * /}

 return 0; }
```

В приведенном примере API sigaction вызываются для установления callme как функции обработки сигнала SIGALRM. Затем программа вызывает API alarm для передачи самой себе сигнала SIGALRM через 5 секунд реального времени. Затем программа входит в бесконечный цикл для выполнения обычных операций.

Когда время таймера заканчивается, вызывается функция cal lme, которая стирает запускает таймер еще 5 секунд, а затем выполняет запланированные задачи. По возвращении из функции callme программа продолжает свою "обычную" работу к следующему срабатывания таймера. Благодаря этой программе можно выполнить синхронизации времени и каждый раз, когда вызывается функция callme, она приглашает настоящее время в отделенного хост-компьютера и затем вызывает API st ime для установления часов локальной системы в соответствии с часами хост-компьютера.

POSIX.1b определяет ряд интерфейсов к положительному программирования, какие предназначены для манипулирования интервальными таймерами. таймеры POSIX.1b принято считать более гибкими и мощными чем таймеры UNIX по следующим причинам

- пользователи могут определить многие независимые таймеров на одни системные времена;
- таймеры POSIX.1b ведут отсчет времени в наносекундах;
- пользователи могут указывать для каждого таймера сигнал, подлежащий генерации после срабатывания таймера;
- интервал таймера может быть задан в форме абсолютного или относительного времени.

На количество установленных на один процесс POSIX - таймеров существует ограничения. Пределом является константа TIMER_MAX, определенной в заголовке <l imits.h>. Таймеры POSIX, созданные процессом, а не наследуются порожденными их процессами, но сохраняются с помощью системного вызова exec. Но в отличие от таймеров UNIX таймер POSIX.1b когда срабатывает не отсылает сигнал SIGALRM и может свободно использоваться в одной программе с API sleep.

В POSIX.1b предусмотрены наступлении API, предназначены для манипулирования таймерами:

```
# include <signal .h>
# include <time.h>
int timer_create (clockid_t clock, struct sigevent * spec, timer_t * timer_hdr) int timer_settime (timor_t timer_hdr, int flag, struct itimerspec * val, struct itimerspec * old)

int timer_getoverrun (timer_t timer_hdr) int timer_delete
(timer_t timer_hdr)
```

API t imer_create динамично создает таймер и возвращает указатель на него. Аргумент clock указывает, показатель которого системного таймера будет использовать новый таймер. Аргумент clock может принимать значения CLOCK_REALTIME, которое позволяет задавать установки таймера в реальном времени. Это значение определяется стандартом POSIX.1b. Другие значения аргументов clock зависят от системы.

Аргумент spec определяет, какое действие нужно выполнить после срабатывания таймера. Тип данных struct sigevent определяется так:

```
struct sigevent {

int sigev_notify; int
sigev_signo;
union sigval sigev_value; }
```

Пример программы которая демонстрирует способ установления таймера абсолютного времени, который должен сработать 1 апреля 2011 в 10:27

```
# include <iostream.h>
# include <stdio.h>
# include <untistd.h>
# include <signal .h>
# include <time.h>

void callme (int signo, siginfo_t * evp, void * uncontext) {

time_t tim = time (0);
```

```
        cerr << "callme:" << evp
si_value.sival_int << ", signo:" << signo << "," << ctime (& tim)
        }
        int main () {

        struct sigaction sig v; struct
        sigevent sig x; struct itimerspec
        val; struct tm do_time; timer_t
        t_id;

        sigemptyset (& sigv.sa_mask)
        sigv.sa_flags = SA_SIGINFO;
        sigv.sa_sigaction = calime;
        if (sigaction (SIGUSRI, & sigv 0) == - 1) {

        perror ( "sigaction") return 1; }


        sigx / sigev_notity = SIGEV_SIGNAL;
        sigx.sigev_signo = SIGUSR1;
        sigx.sigev_value.sival_int = 12;
        if (timer_create (CLOCK_REALTIME, & sigx, & t_id) == - 1) {

        perror ( "timer_create") return 1; }


        / * Встаовиты таймер на срабатывание 1 апреля 2009 г. в 10: 27 * / do_time.tm_hour =
        10; do_time.tm_min = 27; do_time.tm_sec = 30; do_time.tm_mon = 3; do_time.tm_year = 11;
        do_time.tm_mday = 1;




        val. it_value.tv_sec = mktime (& do_time) val.
        it_value.tv_nsec = 0 val. it_interval .tv_sec = 15; val.
        it_interval .tv_nsec = 0 '

        cerr << "timer will go off at:" << ctime (& val. it_value.tv_sec) if (timer_settime (t_id,
        TIMER_ABSTIME, & val 0) == - 1) {

        perror ( "timer_settime") return 2; }


        / * Выполнять другие операции, а затем ждать, пока время таймера не закончится два раза * /

        for (int i = 0; i <2; i ++) pause
          ();
        if (timer_delete (t_id) == - 1) {

         perror ( "timer_delete") return 3; }


        return 0;
        }
```

Данная программа всегда сначала функцию callme как обработчик сигнала SIGUSR1. Затем она создает таймер, используя для этого системный таймер реального времени. Программа указывает, что после срабатывания таймера должен быть послан сигнал SIGUSR1, а данные таймера, которые нужно отправить вместе с сигналом, - это TIMER_TAG. Указатель на таймер, который возвращает API t imer_create, сохраняется в переменной t_id.

В C ++ существует класс t imer. Конструктор класса timer задает также механизм обработки сигналов таймера (с помощью API sigaction). Объявление этого класса может иметь следующий вид: (timer.h)

```
# indef TIMER_H
# define TIMER_H

# include <signal .h>
# include <time.h>
# include <errno.h>

typedef void (* SIGFUNC) (int, siginfo * void *) class timer {


timer_t             timer_id;
int                 status;
struct itimerspec val; public

/ * Конструктор установки таймера * /
timer (int signo, SIGFUNC action, int timer_val, clock_t
        sys_clock = CLOCK_REALTIME) {

struct sigaction sigv; status = 0

sigemptyset (& sigv.sa_mask)
sigv.sa_flags = SA_SIGINFO;
sigv.sa_sigaction = action;
if (sigaction (signo, & sigv 0) == -1) {

perror ( "sigaction") status =
errno; } Else {



struct sigevent sigx;
sigx.sigev_notify = SIGEV_SIGNAL; sigx.sigev_signo
= signo;
sigx.sigev_value.sival_int = timer_val;
if (timer_create (sys_clock, & sigx, & timer_id) == -1) {

perror ( "timer_create") status =
errno; }}};



/ * Дескруптор: уничтожение таймера * / ~ timer () {
```

```
if (status == 0) {

 stop ();
 if (timer_delete (timer_id) == -1) {

    perror ( "timer_delete") }}};



/ * Проверить статус таймера * / int operator!
() {

return status? 1: 0; };

/ * Настроить таймер на относительное время * /
int run (long start_sec, long start_nsec, long reload_sec, long reload_nsec) {

if (status) return -1;
val. it_value.tv_sec = start_sec; val. it_value.tv_nsec = start_nsec;
 val. it_interval .tv_sec = reload_sec; val. it_interval .tv_nsec =
 reload_nsec; if (timer_settime (timer_id 0, & val 0) == -1) {



   perror ( "timer_settime") status =
  errno; return 1; }


 return 0; }

/ * Настроить таймер на абсолютное время * / int run (time_t start_time, long
reload_sec, long reload_nsec) {

if (status) return -1;
val. it_value.tv_sec = start_time; val. it_value.tv_nsec =
0 val. it_interval .tv_sec = reload_sec; val. it_interval
.tv_nsec = reload_nsec;

if (timer_settime (timer_id, TIMER_ABSTIME, & val 0) == -1) {

   perror ( "timer_settime") status =
  errno; return 1; }


 return 0; };

/ * Остановить таймер * / int
stop () {

if (status return 1; val. it_value.tv_sec =
0 val. it_value.tv_nsec = 0 val.
it_interval .tv_sec = 0 val. it_interval
.tv_nsec = 0

if (timer_settime (timer_id 0, & val 0) == -1)
```

```
{
perror ( "timer_settime") status =
errno; return 1; }


return 0;
/ * Получить статистику потерь сигналов * / int overrun () {


if (status) return -1;
   return timer_getoverrun (timer_id) };

/ * Получить оставшееся до срабатывания * / int values (long & sec, long &
nsec) {

if (status) return return 1;
if (timer_gettime (timer_id, & val) == -1) {

perror ( "timer_gettime") status =
errno; return 1; }


sec = val. it_value.tv_sec; nsec = val.
it_value.tv_nsec; return 0; };


/ * Перегрузиты операцию << для объектов класса timer * / friend ostream &
operator << (ostream & os, timer & obj) {

 long sec, nsec;
 obj, values (sec, nsec)
double tval = sec + ((double) nsec / 1000000000.0, os << "time left:" tval; return os;};}; #endif
```

Пример демонстрирующий преимущества и простоту применения класса timer

```
# include <iostream.h>
# include <stdio.h>
# include <stdlib.h>
# include <unistd.h>
# include "timer.h"

void callme (int signo, siginfo_t * evp, void * ucontext) {

long sec, nsec; time sec, nsec;
time_t tim = time (0);

cerr << "timer Id:" << evp-si value.sival int << ", signo:" << signo << "," << stime (& tim)

}
int main () {

timer t1 (SIGINT, callme, 1); timer t2
(SIGUSR1, callme, 2); timer t3 (SIGUSR2,
callme, 3); if (! t1 | |! t2 | |! t3) return 1;
```

```
t1.run (2, 0, 2, 0);
t2.run (3, 500 000 000, 3, 500000000) t3.run (5, 0, 5,
0);
/ * Ждать десятикратного срабатывания таймеров * / for (int I = 0; I <10;
I ++) {

 / * Выполнять другую работу в срабатывания таймеров * / pause ();

 / * Показать время, оставшееся до срабатывания таймера * / if (timer_create
(CLOCK_REALTIME, & sigx, & t_id) == - 1)
            {
                perror ( "timer_create") return 1; }


/ * Установить таймер на срабатывание 1 апреля 2002 в 10:27 * /
            do_time.tm_hour = 10;
            do_time.tm_min = 27;
            do_time.tm_sec = 30;
            do_time.tm_mon = 3;
            do_time.tm_year = 02;
            do_time.tm_mday = 1;

            val. it_value.tv_sec = mktime (& do_time) val.
            it_value.tv_nsec = 0 val. it_interval .tv_sec = 15; val.
            it_interval .tv_nsec = 0


            cerr << "timer will go off at:" << ctime (& val. it_value.tv_sec)

            if (timer_settime (t_id, TIMER_ABSTIME, & val 0) == - 1) {

                perror ( "timer_settime") return 2; }


/ * Выполнять другие операции, а затем ждать, пока время таймера не закончится два раза *
/
            for (int i = 0; i <2; i ++)
                pause ();
            if (timer_delete (t_id) == - 1) {

                perror ( "timer_delete") return 3; }


                return 0; }
```

Данная программа записывает сначала функцию callme как обработчик сигнала SIGUSR1. Затем она создает таймер, используя для этого системный таймер реального времени. Программа указывает, что после срабатывания таймера должен быть послан сигнал SIGUSR1, а данные таймера, которые нужно отправить вместе с сигналом - это TIMER_TAG. Указатель на таймер, который возвращает APE timer_create, сохраняется в переменной t_id.

В C ++ существует класс timer задает также механизм обработки сигналов таймера (с помощью APE sigaction). Объявление этого класса может иметь следующий вид: (timer.h)

```
# ifndef TIMER_H
# define TIMER_H

# include <signal .h>
# include <time.h>
# include <errno.h>

typedef void (* SIGFUNC) (int, siginfo * void *)

class timer
        {
                timer_t         timer_id;
                int             status;
                struct itimerspec val; public

                / * Конструктор установки таймера * / timer (int signo, SIGFUNC action, int
                timer_val, clock_t
sys_clock = CLOCK_REALTIME)
                {
                        struct sigaction sigv; status = 0

                        sigemptyset (& sigv.sa_mask) sigv.sa_flags =
                         SA_SIGINFO; sigv.sa_sigaction = action; if
                         (sigaction (signo, & sigv 0) == - 1) {


                                perror ( "sigaction") status =
                            errno; } Else {



                                struct sigevent sigx; sigx.sigev_notify = SIGEV_SIGNAL sigx.sigev_signo =
                                 signo; sigx.sigev_value.sival_int = timer_val; if (timer_create (sys_clock, &
                                 sigx, & timer_id) == - 1) {



                                        perror ( "timer_create") status =
                                    errno; }}};



                / * Деструктор: уничтожение таймера * / timer (); {


                        if (status == 0) {

                                stop ();
                                if (timer_delete (timer_id) == - 1) {

                                        perror ( "timer_delete") }}};



                / * Проверить статус таймера * /
```

```
int operator! () {

        return status? 1: 0; };

/ * Настроить таймер на относительное время * / int run (long start_sec, long
start_nsec, long reload_sec, long
reload_nsec)
  {
        if (status) return -1; val. it_value.tv_sec = start_sec; val.
        it_value.tv_nsec = start_nsec; val. it_interval .tv_sec =
        reload_sec; val. it_interval .tv_nsec = reload_nsec; if
        (timer_settime (timer_id 0, & val 0) == - 1) {



                perror ( "timer_settime") status =
           errno; return 1; } Return 0; };



/ * Настроить таймер на абсолютное время * / int run (time_t start_time, long
 reload_sec, long reload_nsec) {

        if (status) return -1; val. it_value.tv_sec = start_time;
        val. it_value.tv_nsec = 0 val. it_interval .tv_sec =
        reload_sec; val. it_interval .tv_nsec = reload_nsec;


        if (timer_settime (timer_id, TIMER_ABSTIME, & val 0) == - 1) {

                perror ( "timer_settime") status =
           errno; return 1; } Return 0; };



/ * Остановить таймер * / int
 stop () {

        if (status) return -1 val.
        it_value.tv_sec = 0 val.
        it_value.tv_nsec = 0 val. it_interval
        .tv_sec = 0 val. it_interval .tv_nsec =
        0
        if (timer_settime (timer_id 0, & val 0) == - 1) {

                perror ( "timer_settime") status =
           errno; return 1; } Return 0; };



/ * Получить статистику потерь сигналов * / int overrun ()
```

```
                    {
                        if (status) return -1;
                        return timer_getoverrun (timer_id) };

                    / * Получить оставшееся до срабатывания * / int values (long & sec, long & nsec)
                     {

                        if (status) return return 1; if (timer_gettime (timer_id, &
                         val) == - 1) {

                            perror ( "timer_gettime") status =
                          errno; return 1; }


                            sec = val. it_value.tv_sec; nsec = val.
                          it_value.tv_nsec; return 0; };


                    / * Перегрузиты операцию << для объектов класса timer * / friend ostream &
                     operator << (ostream & os, timer & obj) {

                        long sec, nsec; obj, values (sec,
                        nsec)
                        double tval = sec + ((double) nsec / 10000000000) os << "time
                       left:" tval; return os; };


      };
    #  endif
```

Пример демонстрирующий преимущества и простоту применения класса t imer

```
    #  include <iostream.h>
    #  include <stdio.h>
    #  include <stdlib.h>
    #  include <unistd.h>
    #  include "timer.h"

    void callme (int signo, sigsnfo_t * evp, void * ucontext) {

                    long sec, nsec; time sec,
                    nsec; time_t tim = time (0);

                    cerr. << "timer Id:" << evp-si value.sivac int << ", signo:" << signo
<< "," << ctime (& tim)
        }
    int main () {

                    timer t1 (SIGINT, callme, 1); timer t2
                    (SIGUSR1, callme, 2); timer t3 (SIGUSR2,
                    callme, 3); if (! t1 | |! t2 | |! t3) return 1; t1.run
                    (2, 0, 2, 0);

                    t2.run (3, 500 000 000, 3, 500000000) t3.run (5, 0, 5,
                    0);
```

```
                    / * Дождаться десятикратного срабатывания таймеров * / for (int i = 0; i <10; i ++) {


                        / * Выполнять другую работу до срабатывания таймера * / pause ();

                        / * Показать время оставшееся до срабатывания
таймеров * /
                        cerr << "t1:" << t1 << endl; cerr << "t2:"
                      << t2 << endl; cerr << "t3:" << t3 << endl; }


                    / * Показать статистику потерь сигналов таймеров * / cerr << "t1
                    overrun:" << t1.overrun () << endl; cerr << "t2 overrun:" << t2.overrun () <<
                    endl; cerr << "t3 overrun:" << t3.overrun () << endl;


                    return 0;
            };
```

В этой программе создаются три таймера. Первый таймер срабатывает каждые 2 секунды и генерирует сигнал SIGINT; второй таймер срабатывает каждые 3.5 секунды и генерирует сигнал SIGUSR1; третий таймер срабатывает каждые 5 секунд и генерирует сигнал SIGUSR2. Обработчик для всех этих сигналов - функция cal lme.

Лекция 12

2.7.3. Процессы - демоны

2.7.3.1. Понятие о демоны

Демоны - это фоновый процесс, который выполняет системную задачу. В полном соответствии с UNIX принципу модульности демоны есть программы, а не части ядра. Многие демонов запускается при начальной загрузки и продолжают работать все время, пока систему включено. Другие демоны запускаются при необходимости и работают столько, сколько предусмотрено их функциями.

Демоны играют важную роль в работе ОС. Достаточно сказать, что возможность терминального входа пользователей в систему, доступ к сети, использование системы печати и электронной почты - все это обеспечивается соответствующими демонами

- интерактивными программами, которые составляют собственные сеансы (и группы) и такие которые не принадлежат ни одному из пользовательских сеансов (группам).

2.7.3.2. Основные демоны UNIX

Некоторые демоны работают постоянно, наиболее ярким примером такого демона

- процесс init (), который является прародителем всех положительных процессов в системе.

Init - это первый процесс, который запускается после начальной загрузки системы, и во многих отношениях это самый важный демон. Он всегда PID 1 и является предком всех пользовательских и почти всех системных процессов.

Во время начального загрузки init или переводит систему в однопользовательский режим, или продолжает shel l для чтения файлов запуска. Когда система загружается в однопользовательском режиме, init начинает читать файлы запуска после того, как однопользовательский shell завершается нажатием комбинации клавиш [Ctrl + D].

После обработки файлов запуска ini t обращается к файлу конфигурации (Letcl ttytab, letclttys или letclini ttab, в зависимости от системы) и получает оттуда список портов, через которые нужно ожидать вхождения в систему. Демон init активизирует эти порты и порождает для каждого из них процесс getty. Если порт открыть невозможно, init периодически выдает на системный пульт сообщение, пока порт не будет открыта или удален с списке активных портов.

Демон init, кроме того, выполняет достаточно неприемлемую задачу: выгоняет еще живы процессы - 30 Mgb, которые накапливаются в системе. Остановка системы осуществляется путем отсылки демону init соответствующего сигнала (обычно это SIGTERM), который заставляет его перевести систему в однопользовательский режим. Эта последняя операция в большинстве сценариев остановки. Демон init играет столь существенную роль в работе системы, в случае его "смерти" инициируется ее автоматическое перегрузки.

Другим примером является cron (), который позволяет запускать программы в определенный момент времени inetd (), который обеспечивает доступ к сервисной системы из сети, и sendmail (), который обеспечивает получение и отправка почты.

Демон cron () отвечает за выполнение команд по установленному графику. Он обрабатывает файлы с расписанием задач (crontab - файлы) созданы как пользователями, так и администратором.

По отношению демонов можно сформулировать ряд правил, которые определяют их нормальное функционирование, которое нужно учитывать при разработке таких программ:

- демон не должен реагировать на сигналы управления задачами, которые отсылаются при попытке операции в-в с управляющим терминалом. Начиная с некоторого времени, демон снимает ассоциации с управляющим терминалом, на самом начальном запуске ему может потребоваться вывести то или иное сообщение на экран.
- нужно закрыть все открытые файлы (файловые дескрипторы), особенно стандартные потоки в-в. Многие из этих файлов представляют собой терминальные устройства которые

должны быть закрыты, например при выходе пользователя из системы. Предполагается, что демон остается работать и после того, как пользователь "покинул" CIN8X.

- нужно знать его ассоциации с группой процессов и управляющим терминалом. Это позволит бесу освободиться от сигналов, генерируемых терминалом (SIGINT или SIGHUP), например при нажатии определенных клавиш или выходе пользователя из системы.

- сообщение о работе демону нужно направлять в специальный журнал с помощью функции sylog () - это наиболее корректный способ передачи сообщений от демона.

- нужно изменить текущий каталог на корневой, допустим, находиться на смонтированную файловую систему, последнюю нельзя будет размонтировать. Самым надежным выбором является корневой каталог, который всегда принадлежит корневой ФС.

2.7.3.3. Пример программы - демону

```
#  include <stdio.h>
#  include <syslog.h>
#  include <signal .h>
#  include <sys / types.h>
#  include <sys / param.h>
#  include <sys / resource.h>

main (int argc, char ** argv)
        {
                int fd;
                struct rlimt flim;
```

/ * Если родительский процесс - init, можно не беспокоиться за терминальные сигналы. Если нужно игнорировать сигналы, связанные с в / в на терминал фонового процесса:

```
SIGTTOU, SIGTTIN, SIGTSTP * /
                if (getppid ()! = 1) {

                        signal (SIGTTOU, SIG_TGN) signal
                        (SIGTTIN, SIG_TGN) signal (SIGTSTP,
                        SIG_TGN)
```

/ * Теперь нужно организовать соответствующую группу и сеанс, не имеющие управляющего терминала.

Но лидером группы и сеанса может стать процесс, если он не является лидером. Поскольку предыстория запуска данной программы известна, нужна гарантия, что наш процесс не является лидером. Для этого породжуемо процесс - потомок. Так как его PID уникальный, то ни группы, ни сеанса с таким идентификатором не существует, а значит отсутствует и лидер. При этом родительский процесс мгновенно заворачивает выполнения, поскольку он уже не нужен.

Существует еще одна причина необходимости порождения процесса - потомка. Если демон был запущен из командной строки командного интерпретатора shell не у фоновом режиме, последний будет ожидать завершения выполнения демону, и таким

образом, терминал будет заблокирован. Порождая процесс и заканчивая выполнения отца, имитируем для командного интерпретатора завершения работы демона, после чего shell выведет свое приглашение. * /

```
if (fork ()! = 0)
        exit (0); / * Отец заканчивает работу * /

/ * Процесс потомок с помощью системного вызова setsid () становится лидером новой группы, сеанса и не имеет ассоциированного терминала * /
        setsind (); }


/ * Теперь нужно закрыть открытые файлы. Закрыть все возможные файловые дескрипторы. Максимальное количество открытых файлов получим с помощью функции getrlimit () * /

getrlimit (RLIMIT_NOFILE, & flim) for (fd = 0 fd <flim,
rlim_max; fd ++)
                close (fd);

/ * Закончим текущий каталог на корневой. * /
                chdir ( "/")

/ * Заявит о себе в системном журнале. Для этого сначала установим опции ведения журнала: каждая запись будет передуватися идентификатором PID демона, при невозможности записи в журнал сообщение будет выводиться на консоль, источник сообщений определим как "системный демон". * /

openlog ( "Склет демону", LOG_PID \ LOG_CONS, LOG_DAEMON)

/ * Отметимся * /
syslog (LOG_INFO, "Демон начал плодотворную работу .... '); closelog ();


/ * Далее может идти текст программы, которая реализует полезные функции демона. Эта часть разрабатывается свободная * /
                . . . . . .
        }
```

В программе использовалось еще возможность системного журнала сообщений программ выполняемой работы. Функцией генерации сообщений является syslog (), который в свою очередь или выводит на их консоль, или перенаправляет в соответствии со списком пользователей данной
или удаленной системы. Конкретный пункт назначения определяется конфигурационным файлом (letc / syslog.cont).

Функция имеет определение:

```
#  include <syslog.h>
void syslog (int priority, char * logstring, / * параметры * /...)
```

Каждому сообщению logstring назначаются приоритеты, указанный параметром priority. Возможные значения этого параметра включают:

LOG_EMERG идентифицирует состояние "Паника" в системе. конечно рассылается всем пользователям.

LOG_ALERT Идентифицирует ненормальное состояние который должен быть исправлен мгновенно, например, нарушение цельности системной базы данных.

LOG_CRIT Идентифицирует критические события, например, ошибку дискового устройства.

LOG_ERR Идентифицирует различные ошибки.

LOG_WARNING Идентифицирует предупреждения.

LOG_NOTICE Идентифицирует события, которые не являются ошибками, но требуют внимания.

LOG_INFO Идентифицирует информационные сообщения как, например, использования в приведенной программе.

LOG_DEBUG Идентифицирует сообщение, конечно такое использующий при отладке программы.

Последний тип сообщений указывает еще одну возможность использования системного журнала - для отладки программы, особенно неинтерактивных.

Строка logstring может включать элементы форматирования, такие же, какие в функции printf () с одним дополнительным выражением% m, которое заменяется сообщением, что соответствует ошибке errno. При этом может осуществляться вывода значений дополнительных параметров.

Функция openlog () позволяет определить ряд опций ведения журнала. Она имеет следующее определение:

void openlog (char * indent, int logopt, int facility)

Строка ident будет предыдущим каждому сообщению программы. Аргумент logopt задает дополнительные опции в том числе:

LOG_PID Позволяет указывать идентификатор процесса в каждому сообщении. Эта опция полезна при журнала нескольких демонов с одним и тем же значением ident, например, когда демоны порождаются вызовом fork ().

LOG_CONS позволяет выводить сообщения на консоль при невозможности записать в журнал наконец, аргумент facility позволяет определить источник сообщений:

LOG_KERN Указывает, что сообщение определяются ядром.

LOG_USER Указывает, что сообщение определены прикладным процессом (Используется по умолчанию).

LOG_MAIL Указывает, что инициатором сообщений является система электронной почты.

LOG_DAEMON Указывает, что инициатором сообщение является системный демон. LOG_NEWS указывает, что инициатором сообщений есть система телеконференций USENET.

LOG_CROW Указывает, что инициатором сообщений является система cron ().

Закончив работу с журналом, нужно аккуратно закрыть ее с помощью функции closelog ().

void closelog (void);

Лекция 13

3. Средства, механизм, подсистемы ОС

3.1. Система управления процессами

3.1.1. Двухуровневая схема управления процессами

При построении системы управления процессами в большинстве современных ОС принято двухуровневую схему. Это значит, что в системе различают два вида планировщиков процессов, которые выполняют соответственно функции долгосрочного и краткосрочного планирования по использованию ЦБ для развития на этом определенной множества процессов. Эти планировщики носят разные названия в разных ОС и составляют в совокупности с определенными информационными структурами в каждой из них систему управления процессами.

Наличие двух уровней управления процессами обусловлено частотными принципами построения ОС. На уровень долгосрочного планирования выносятся действия жидкие в системе, но которые требуют больших системных затрат.

На уровень краткосрочности планирования выносятся частые и более короткие по длине действия по управлению процессами. На каждом из уровней существует свой объект и собственные средства управления им.

На уровне долгосрочного планирования объектами управления являются не отдельные процессы, а некоторые их объединения, которые рассматриваются как единое целое, объединение единым функциональным назначением. Такие объекты в разных ОС имеют разное название. Такое объединение процессов будем называть работой.

Каждая работа рассматривается как независимая от других робот деятельность, связанная с выполнением одной или нескольких программ, как пользовательских, так и системных, на одну или нескольких процессорах для достижения определенного результата. Именно поэтому каждую работу можно представить как совокупность процессов.

Работа имеет конец. Начавшись в некоторый момент времени, она заканчивается нормально (достигнутый конечный результат) или ни через некоторый интервал времени. Работа должна выполняться по выдающейся программе на одном или нескольких ЦП. Такая программа составляется с помощью специальных языков.

Для выполнения некоторой программы работы предоставляется виртуальная машина, которая имеет в своем составе следующие компоненты. ОП, которая может хранить программу работы по мере ее выполнения. Другие виды памяти, которые могут хранить и обеспечивать доступ к входных и выходных данных для каждой элементарной действия. Процессор, который может различать и выполнять элементарные действия программы в установленном порядке. Тогда работа по отношению такого рода процессору выступает ни чем иным, как процессом. Как раз такого рода обобщенное, более абстрактное представление работы как процесса и принято на уровне долгосрочного планирования. На этом уровне планировщик планирует распределение пор не ЦБ, а виртуальной машины. Каждый раз по мере порождение новой работы создается собственная виртуальная машина для ее выполнения. Это значит, что распределение возникает однократно в отличие от уровня краткосрочного планирования, где время ЦБ для каждого из процессов распределяется многократно. Именно поэтому на уровне выполнения работ принято говорить о долгосрочном планировании.

На уровне краткосрочного планирования объектом управления являются процессы, которые выступают как потребители или ЦБ, если речь идет о внутренних процессах или других процессоров для внешних процессов. Распределение процессора выполняет планировщик данного уровня в соответствии с некоторой стратегией планирования, которая определяет правило выбора одного из процессоров, которые находятся в состоянии «готовности» для последующей его активиза й при увольнении процессора. Кроме планировщика на данный уровень выносятся и допустимые для использования процессами механизмы и средства, которые обеспечивают системные действия о порождению и уничтожению процессов, по переводу процессов с состояния в состояние, по решению нужных задач синхронизации при взаимодействии процессов.

Нужно отметить два важных свойства выполнения работ ОС на уровне краткосрочного планирования. Во - первых, причины, обуславливающие порождения и окончания процессов обрабатываемых на этом уровне, являются внутренними. Иными словами, заявки или требования на порождение и уничтожение процессов поступают или от процессов, управляемых на данном уровне, или с уровня долгосрочного планирования. Во - вторых, особенность стратегии распределения процессора на данном уровне, которую часто называют диспетчеризацией, необходимость многократного распределения единого процессора для каждого из запланированных процессов на интервалах их существования с целью достижения нужного эффекта мультипрограммирования. Именно поэтому планирование распределения

времени процессора по отношении каждого процесса называют краткосрочными.

3.1.2. Уровень долгосрочного планирования

Любую работу, подобно процесса, можно рассматривать как совокупность состояний по выполнению программы работы на виртуальной машине. Больше всего используется следующие из них.

Состояние «порождение», когда в ответ на требование выполнить работу планировочных верхнего уровня (долгосрочное планирование) создает нужную виртуальную машину. Одной из характерных особенностей данного уровня планирования является то, что источник требований на порождение работы является внешним по отношению к ЦБ. Требование представляет собой программу работы, составленную пользователем на командной языке используется.

Порождение работы, заключается в создании виртуальной машины и сводится к выполнению планировщиком верхнего уровня следующих действий:

1. Резервируются все ресурсы, указанные в задании и необходимые для выполнения программы работы. Это относится и к данным. Обеспечивается организация доступа к этим данным, которые находятся на определенных носителях и руководствуются соответствующими средствами ОС.

2. Резервируется память, в которую будет размещено программа работ по мере й й выполнения. В разных ОС вид памяти резервируется может отличаться.

3. Наконец, заводится структура данных, которая выполняет роль дескриптора работы и содержит нужную такую что модифицируется информацию для управления выполнением работы. Дескрипторы работ в свою очередь объединятся в определенные списки, создавая в результате управляющую информационную структуру, которую используют для работы долгосрочным планировщиком. Структура и название дескриптора работы различных ОС разные.

Состояние «Готовность» для работы означает, что для выполнения программы работы предоставлены все ресурсы виртуальной машины, кроме виртуального процессора. Деятельность виртуального процессора по выполнению каждой элементарной действия программы работы моделирует ЦП планировщик нижнего уровня (краткосрочного планирования), организуя развитие соответствующих процессов. Отсюда, для выполнения элементарной действия программы работ планировщик верхнего уровня должен выработать в установленном виде заявку на порождение определенного процесса на нижнем уровне. Порождение такой заявки и составляет содержание системных действий по переводу работы по состоянию «Готовность» в состояние «Активность».

Состояние «Активность» определяет выполнение работы на виртуальном процессоре. Каждый раз по мере окончания выполнения очередной элементарной действия на уровне краткосрочного планирования планировщик этого уровня передает сообщение в установленной форме планировщике верхнего уровня (и тот и другой - это системные параллельные процессы). В ответ на это сообщение планировщик верхнего уровня подготавливает условия для выполнения следующей элементарной действия со склада программ работы и затем снова производит заявку на порождение процесса, который будет развиваться на уровне краткосрочного планирования.

Состояние «Окончание» определяет совокупность действий, выполняемых под управлением планировщика верхнего уровня при окончании выполнения программы работ. Это прежде

все суммирования учетной информации по затратам, которые служат основой для начисления счета за выполненную работу. Суммируется, например, расход «чистого» процессорного времени, или общего времени занятости ЦП на фактическое выполнение пользовательских и системных программ по мере выполнения работы. Кроме других действий выполняется передача результирующих выходных данных на указанные в задании носителя информации. Например, организуется вывод результатов работы программы пользователя на печать. Наконец, освобождаются все ресурсы, которые были использованы для построения виртуальной машины для выполнения работы завершившуюся и уничтожается дескриптор работы.

3.1.3. Схема долгосрочного уровня планирования

Программа работы или задачи формируется пользователем на специальном языке. Поэтому основу долгосрочного планирования предоставляет своего рода «полный процессор», который должен вводить эти программы в компьютер и обеспечивать условия для их дальнейшей реализации. Существует два принципа реализации такого речевого процессора: компилятивное и интерпретационный.

Компиляционный принцип. В этом случае речевой процессор объединяет несколько автономных и больших программных подсистем. Эта совокупность системных программных средств обеспечивает ввод текстов задач - программ работ, записанных на языке управления заданиями, выполняет компилируя функции, или осуществляет трансляцию текстов задач в определенное внутреннее представление, конкретизирует условия выполнения и планирования порядка выполнения отдельных работ, выраженные в текстах задач.

Речевой процессор в ОС UNIX построено по интерпретационной схеме. Роль процессора выполняет в системе специальная Shel l - программа, которая рассматривается в системе как служебная интерактивная программа (программная утилита). Название утилиты своеобразно подчеркивает ее место в системе (Shell - оболочка).

Эта утилита по отношению к пользователю выступает как оболочка ядра системы. Через эту утилиту пользователь взаимодействует с ядром. Shell - программа является роздиляемою программой среди пользователей и построена так как нерентабельна. При общении с единственным пользователем она ведет с ним диалог. Выразительным средством при этом выступает Shell - язык, который выполняет не только командные функции в плане общего управления системой, но одновременно и развитой языке программирования Shell - программа позволяет обеспечить интерпретацию команд, поступающих от пользователя по мере ведения с ним диалога, а также обеспечивает выполнение на фоне диалога нужных пользовательских программ.

3.1.4. Уровень краткосрочного планирования

Процесс - это совокупность отдельных состояний, поэтому в составе ОС нужны механизмы, с помощью которых можно переводить процесс с одной допустимого состояния.

Переход процесса в состояние «рожден» обусловлено деятельностью системы управления процессами в ответ на соответствующую заявку, сформированная на этапе долгосрочного планирования.

Результатом такой деятельности является интерпретация содержания заявки на порождение. Считается, что система создала процесс, если по отношению к этому объекту она:

- определила в результате операции имя процесса или программу, которую нужно выполнить;
- обеспечила доступ к ресурсам, которые должны быть выделены перед выполнением программы, в том числе обеспечила передачу программе необходимые параметры; построила специальную структуру данных, в которой будет отображаться информация, необходимая для фиксации состояния процесса, для отслеживания порядке перехода из одного состояния в состояние, для управления распределением ресурсов. Такую структуру называют дескриптором процесса.

Деятельность по переводу процесса в состояние «закончено» проводится системой управления процессами в ответ на требование об окончании выполнения программы. Окончание может быть либо нормальным, обусловлено алгоритмом программы, или аварийным по каким-то причинам, не допускают дальнейшего выполнения.

При переводе в состояние «закончено» над процессом выполняются действия, обратные действиям, которые выполняются при порожденные процесса. Передается, или сохраняется, если это необходимо, информация о результатах работы процесса, закончено, тем процессам, которые потребляют эту информацию. Освобождаются ресурсы, распределенные процесса заканчивающийся. Чаще всего они передаются тому процессу, который является продолжающим в отношении того, что заканчивается и выделял последнем в момент порождения ресурсы. При этом, как правило, ОС выполняет окончательный подсчет и фиксацию расходов, связанных

с использованием ресурсов на интервале существования процесса.

Окончательной действием является уничтожение дескриптора своей программы.

Уничтожение процессов требует решения ряда проблем. Процесс, уничтожается, может быть связан с другими процессами. Например, процесс который уничтожается может быть рожденным по отношению одного или более порожденных им процессов и контролировать их развитие. В этом случае возникает вопрос: что делать с потомками процесса что уничтожается? Чаще всего используют решения, когда при необходимости уничтожить некоторый процесс будет уничтожаться и все поддерево, которое определяет родительские связи процесса. Корнем этого поддерева является процесс, который нужно уничтожить. Иногда предъявляют требование, чтобы процесс - отец не мог быть уничтоженным ранее какого-то его процесса - потомка.

Чтобы выполнить это требование, в состав системы управления процессами вводятся средства, которые позволяют соответственно делать пометки об окончании процессов и контролировать моменты когда эти события наступят. В ОС UNIX - это системные вызовы wai t и exit (wait - задержка, exit - завершение просинь.).

Последний вызов отрабатывает действия, связанные с уничтожением процесса, а также извещает процесс - отец что ожидает, что процесс - потомок уничтожено. Процесс - отец после этого переводится из одного состояния ожидания в этап «готовности».

Еще одна проблема, которая возникает при уничтожении процессов. Уничтожен процесс, как правило, не является изолированным, а взаимодействует прямо или косвенно с другими процессами. Например, имеет информационные связи с другими процессами - передает или сдерживает информацию, разделяет с другими процессами ресурсы, среди которых, возможно, есть критические. Поэтому при уничтожены процесса взаимодействующего нужно корректно «закрывать» связь, в котором он задействован. Для этого используют различные средства. Например, при уничтожении процесса направляются специальные «пустые» сообщения тем процессам, которые ждут поступления информаци й от того кто уничтожается.

Делается искусственный выход из критических областей, если процесс, уничтожается использовал критические ресурсы и т.д. Процесс сам не может перейти из состояния «рожденный в состояние« готовности »или любой другой. В любом состоянии, кроме «активного» процесса является пассивным объектом. Он сам не может выполнять действий.

Переход с «активного» состояния в какой - либо другой может выполняться как "по инициативе" самого процесса, так и по внешней инициативе.

Образ процесса состоит из двух частей: данных режима ядра и режима задачи. Образ процесса в режиме задачи состоит из сегмента кода, данных стека, библиотек и других структур данных, к которым он может получить непосредственный доступ. Образ процесса в режиме ядра состоит из структур данных, которые недоступны процесса в режиме

задачи, используемых ядром для управления процессом. сюда относятся данные, которые диктуются аппаратным уровнем, например состояние регистров, таблицы для отображения памяти и т.д. , А также структуры данных, необходимые ядру для обслуживания процесса. Обще говоря, в режиме ядра процесс имеет доступ к любой области памяти.

Рис. 3. Инфраструктура процесса ОС UNIX.

Лекция 14

3.1.5. Структуры данных процессов

Каждый процесс представлен в системе двумя основными структурами данных proc и user, описанными соответственно, в файлах <sys / proc.h> и <sys / user.h>. Содержание и формат этих структур может быть различным для разных версий UNIX.

Приведем некоторые поля структуры proc для платформы и386.

Таблица 3.1. Структура proc char

	p_stat	Состояние процесса (выполнения работ, приост., Сон и т.д.)
char	p_pri	Текущий приоритет процесса
unsigned int	p_flag	Флаги которые определяют дополнительную информацию о состоянии процесса
unsigned short	p_unid	UID процесса
unsigned short	p_suid	EUID процесса
int	p_sid	идентификатор сеанса
short	p_pgrp	Идентификатор группы процессов (равен идентификатору лидера гр.)
short	p_pid	Идентификатор процесса PID
short	p_ppid	Идентификатор родительского процесса PPID
sigset_t	p_sig	Сигналы что ждут доставки
unsigned int	p_size	Размер адресного пространства процесса на страницах
time_t	p_utime	Время выполнения в режиме задачи
time_t	p_stime	Время выполнения в режиме ядра
carddr_t	p_ldt	Указания на LDT процесса
struct pregion	p_region	Список областей памяти процесса
short	p_xstat	Код возврата, передается родительскому процессу
unsigned int	p_utbl []	Массив записей в таблице страниц для u-area

И любой времени данные структур proc для всех процессов должны присутствовать в памяти, хотя другие структуры данных, включая образ процесса, могут быть перемещены во вторичную память, - область подкачки. Это позволяет ядру иметь под рукой минимальную информацию, необходимую для определения места нахождения других данных, относящихся к процессу, даже если они отсутствуют в памяти.

Структура proc является записью системной таблицы процессов, которая всегда находится в ОП. Запись этой таблицы для процесса выполняемой в данный момент времени адресуется системной переменной curproc. Каждый раз при переключении контекста, когда ресурсы процессора передаются другому процессу, соответственно изменяется значение переменной curproc, которая теперь указывает на структуру proc активного процесса.

Вторая структура - user, также называется u-area или u_block, содержит дополнительные данные содержит данные о процессе, который нужен ядру только при выполнении процесса (или когда процессор выполняет инструкции процесса в режиме ядра или задачи. В отличие от структуры proc, которая адресована указателем curproc, данные user размещаются (точнее, отображаются) в определенном месте виртуальной памяти ядра и адресуются переменной u (рис 3.2.)

В u_area хранятся данные, которые используются многими подсистемами ядра и не только для управления процессом. Отдельно, там содержится информация об открытых файловые дескрипторы, диспозиция сигналов, статистика выполнения процесса, а также сохранения значения регистров, когда выполнение процесса приостановлено. Очевидно, что процесс не должен иметь модифицировать эти данные произвольным образом, поэтому u-area защищена от одступу в режиме задачи.

Как видно из рисунка, U - area также содержит стек фиксированного размера - системный стек или стек ядра (kernel stack). При выполнении процесса в режиме ядра, ОП использует этот стек, а не обычный стек процесса.

3.1.5.1 Состояние процессов в UNIX

exit () wait ()

- fork ()

sleep ()

wakenp ()

рис.3.3 Состояние процесса в UNIX.

1. Процесс выполняется в режиме задачи. При этом, процессором выполняются положительные инструкции данного процесса.

2. Процесс выполняется в режиме ядра. При этом, процессором выполняются системные инструкции ядра ОС от имени процесса.

3. Процесс не выполняется но готов к записи, как только планировщик выберет его (состояние runnable). Процесс находится в очереди на выполнение и имеет все необходимые ему ресурсы, кроме вычислительных.

4. Процесс находится в состоянии сна (asleep), ожидая недостижимого, в данный момент ресурса, например, завершения операции ввода / вывода.

5. Процесс возвращается из режима ядра в режим задачи но ядро прерывает его и выполняет переключение контекста для запуска более

высокопроизводительного процесса.

6. Процесс только, что создано вызовом fork () и находится в переходном состоянии: он существует, но не готов к запуску и не находится в состоянии сна.

7. Процесс выполнил системный вызов exit () и перешел в зомби (zombie, defunct). Если такой процесс не существует но остаются записи, содержащие код возврата и временную статистику его выполнения, которая доступна для родительского процесса. Это состояние является конечным в жизненном цикле процесса.

3.1.6 Особенности планировщика Linux

Планировщик Linux можно назвать быстрым очередным планировщиком или краткосрочным планировщиком. Основная функция планирования ядра называется schedule.

```
asmlinkage void shedule (void) {

  struct schedule_data * shed_data; struct
  task_struct * prev, * next; int this_cpu;

  run_task_queue (& tg_scheduler) prev =
  current;
  this_cpu = prev -> processor;
/ * 'Sched_data' is protected by the fact that we can run only on process per CPU * /
   sched_dat = & aligned_data [this_cpu]. Schedule_data;
```

```
if (in_interrupt ())
goto scheduling_in_interrupt;
release_kernel_lock (prev, this_cpu)
/ * Do "administrative" work here while we do not hold any locks * /
if (bh_active & bh_mask) do_bottom_half (); spin_lock (&
sheduler_lock) spin_lock_irq (& runqueue_lock) / * Move an
exhausted RR process to be last .... * /

prev -> need_resched = 0
if (! prev -> counter && prev -> policy = SCHED_RR) {

prev -> counter = prev -> priority;
move_last_runqueue (prev) }
```

Алгоритм, который используется в конкретном случае, зависит от процесса. Алгоритм планирования, используемый для данного процесса, называют его политикой планирования и отражается в члене policy структуры struct task_struct процесса. Обычно, член policy имеет установленным один из разрядов SHED_OTHER, SHED_FIFO, или SHED_RR. Но он может также иметь разряд, установлено SHED_YIEID, если процесс решит освободить ЦБ, например, вызвав системную функцию shed_yieid:

```
asmlinkege int sys_sheld_yield (void) {

spin_lock (& scheduler_lock) spin_lock_irq (&
runqeue_lock) if (current -> policy =
SCHED_OTHER) current -> policy! =
SCHED_YIEID; current -> need_resched = 1;
move_last_runqueue (current) spin_unlock_irq (&
runqueue_lock) spin_unlock (& scheduler_lock)
return 0; }
```

SCHED_XXX определяется через #define SCHED_OTHER, который определяет, что применяется традиционное планирование. Unix - процесс не является процессом реального времени - 0.
SHED_FIFO отмечает, что этот процесс является процессом реального времени и объектом для планирования FIFO. Он должен порождаться до тех пор, пока не заблокируется ввода / вывода, явным образом не освободит либо не будет вытеснен другим процессом реального времени с более высоким приоритетом
rt_priority (1). SHED_RR отмечает, что процесс является процессом реального времени и объектом планировщика RR (round - robin) POSIX.1b. Он аналогичен SHED_FIFO за исключением того, что временные кванты имеют другое значение. Когда временной квант процесса SHED_ RR следует, что он перемещается в конец процессов SHED_FIFO и SHED_RR с таким же приоритетом rt_priority (2). SHELD_YIELD является не политикой планирования, а дополнительным разрядом, который противоречит политике планирования. Как уже было сказано ранее, этот разряд предписывает планировщике освободить процессор, если он нужен каком Другому. Отдельно, это может привести даже к тому, что процесс реального времени освободит процессор для процесса не реального времени.

3.1.7. Вычисление значения goodness

Goodness процесса вычисляется функцией goodness.

```
Static inline int goodness (struct task_struct * p, struct task_struct * prev, int this_cpu) {

  int policy = p -> policy;
```

....

Эта функция возвращает значение, которое относится к одному из двух классов: менее 1000 и более 1000. Значение от 1000 и больше работают только процессом "реального времени", а значение от 0 до 999 только зичайним процессом. В действительности значение goodness для обычных процессов занимают только самую нижнюю часть этого диапазона от 0 до 41 (или от 0 до 56 для SMP поскольку режим SMP придает процессу дополнительную возможность оставаться в процессоре, который используется. И для SMP и для UP значение goodness процессов реального времени изменяются от 1001 до 1099).

Лекция 15

3.1.8 Дескрипторы процессов

Процесс - это абстракция, которая введена для лучшего восприятия работы системы, построения механизмов ОС на едином концептуальном базисе. Любой - либо процесса свойственны две вещественные части: программа, по которой будет развиваться процесс в активном состоянии, и дескриптор процесса, который представляет собой информационную структуру, в которой сосредоточена управляющая информация, которая нужна и достаточна для системы управления процессом.

Процесс - это динамический объект, который может менять свое состояние и может, по мере своего развития, приглашать, использовать и освобождать ресурсы. Динамика изменения состояний, а также связей с ресурсами обусловливают динамический характер информации, который сосредоточено в дескрипторе какого - либо процесса. В той или иной степени в дескрипторе отражаются классификационные признаки процессов. Прежде всего, отражены связи, которые изменяются динамически, с другими процессами. Все процессы, которые находятся в текущий момент времени в одном и том же состоянии, объединены через дескрипторы в одну цепь ("прошивают" из дескрипторы) и создают списочную структуру. Такой же прием проводят с дескрипторами тех процессов, которые ждут доступа к одному и тому же ресурсу. В таком случае прош и вку, как правило, осуществляют в соответствии с приоритетами на право использовать ресурс. Дескрипторы прошивают также для отображения "родственных" связей между процессами. Бесспорно одно из главных ссылок, которые хранятся в дескрипторе - это ссылка на область памяти, в которой будут использоваться программы при видшук овув ни процесса в активном состоянии.

Таким образом, информацию, которая находится в дескрип т ори, можно разделить на несколько групп по функциональному назначению:

- информация по идентификации, содержит уникальное имя процесса для реализации операций управление над процессами, как над именуемыми объектами;
- информация о ресурсах, содержит информацию о ресурсах, которые требуются или используются процессом в настоящее время;
- информация о состоянии процесса, содержит информацию о текущем состоянии процесса, позволяет определить текущее состояние и возможность перехода на следующий. Например, в состояниях, отличных от «активного», хранится информация которая является информацией, которая находится в слове состояния программы. Сохраняется содержимое регистров, хранящиеся или зсилкина области те области, где хранится данная информация;
- информация о родственных связях, используется для конкретного окончания процессов, связанных родственными связями, передачи для совместного или автоматического использования где
- каких ресурсов, для установления информационных связей;
- информация, необходимая для учета планирования процессов, содержит адресные ссылки на другие процессы в случае, если процесс находится в каких-то очередях, приоритет или место в соответствующей очереди, ссылки на информацию для реализации доступа к внешним данным по отношению к процессу, ссылки на средства синхронизации между процессами и т.д.

Физическое отражение дескрипторов в каждой ОС выполняется по - разному.

В ОС UNIX процесс представлено, во - первых, областью данных пользователя, которая в момент существования процесса может быть выделена в оперативную память и сохраняет данные и программу пользователя или находится на внешней памяти, если процесс получил подкачки - его временно вытеснены из ОП. Во - вторых, процесс представлен дескриптором процесса. Дескрипторы объединены в таблице процессов. С каждого дескриптора имеется ссылка на таблицу пользователя, которая еще называется контекст процесса. Каждая таблица пользователя - это продолжение дескриптора. В этой таблице хранится менее актуальная информация, чем в таблице процессов. Таблицы имеют еще то отличие, что они постоянно находятся в оперативной памяти в известном месте и размер ее фиксировано, а таблица

пользователя требуется системе только в том случае, когда процесс находится в активном состоянии. Поэтому эта таблица, будучи частью области да процесса, может быть перемещена, при необходимости, на диск, когда процесс находится в состояниях, отличных от активного. В данной ситуации, она становится недоступной для модификации.

В таблице пользователя сосредоточено информацию, которая ориентированная на организацию работы процесса с внешними данными (файлами). Это - программный интерфейс процесса с файловой системой.

Каждый дескриптор в таблице процессов, резервируется при порождении процесса и освобождается при его завершении. В нем содержится статусная информация или процесс, которые нужны, независимо от того, находится ли процесс в ОП, или его вигружено.

Отдельно в н ь ом содержатся адресные ссылки на область ОП, где располагается адресное ссылки на область внешней памяти, куда вытесняется процесс в результате подкачки. В дескрипторе содержится информация, которая нужна для синхронизации процессов. В нем также содержатся идентификаторы процесса, а также процесса, его породил.

3.1.9 Схема работы краткосрочного планировщика

Реализация краткосрочного планировщика представлена по - разному в разных системах. Как правило - это подсистема управления задачами. Она, в свою очередь, состоит из отдельных программных единиц, выполнение которых реализуется в форме параллельных процессов, взаимодействующих. Основные составляющие элементы подсистемы управления см. на рис.3.3

Супервизор задач - системная программа, которая выполняет функции по управлению процессами. В ее состав входит супервизор связей, осуществляющий функции по созданию, уничтожению и перевода задач из состояния в состояние. Супервизор задач также поддерживает механизм синхронизации параллельных процессов (задач) и механизм статического и динамического назначения приор задачам, которые учитываются при распределении главного ресурса в системе - ЦБ. Второй основной компонент подсистемы управления задачами - диспетчер задач. В обязанности этой системной программы, при ее выполнении, планирования использования процессора.

Рис.3.3 Структура системы управления задачами.

Другими словами, диспетчер задач выполняет, совместно с программно - аппаратной системой прерываний, функции краткосрочного планировщика в системе. Каждый раз, по мере обработки где - либо прерывания выполняется перепланировка использования ЦП до момента, когда наступит следующее прерывание. Диспетчер задач обслуживает две очереди: очередь готовности и очередь ожидания. В первой находятся благоустроенные, по приоритетам задачи, которые находятся в состоянии готовности. Во второй очереди находятся задачи в состоянии ожидания выполнения где - события. Какое - либо прерывания свидетельствует об осуществлении того или иного события. Поэтому, после определения, которой события (событиям) соответствуют прерывания, возникшие в системе, проверяется наличие задач (в очереди ожидания), которые ждали бы осуществление данного события. Если таковые находятся, то они переводятся в очередь готовности и располагаются там, в соответствии со своим текущим приоритетом. Диспетчер задач, обращаясь к очереди готовности, выбирает из нее этот процесс (задачу), который, в текущий момент перепланировки, имеет наибольший приоритет. Эта задача переводится в активное состояние, или ей передается для использования ЦП до следующего прерывания.

Кроме названных компонентов в состав подсистемы управления задачами могут входить еще супервизор памяти и супервизор интервального таймера. Первый выполняет распределение ОП, второй руководит обращениями к интервального таймера по требованиям параллельных задач.

В ОС UNIX эти функции выполняет один из двух секций ядра этой системы. Эта секция, по назначению, является центральной. Она программно реализована на языке С. Основные функции, которые выполняет секция:

- резервирования ресурсов;
- определения последовательности выполнения процессов;
- принятие запросов на обслуживание.

3.1.10. Взаимодействие между процессами в UNIX

В UNIX процессы выполняются в собственном адресном пространстве и, по сути изолированы друг от друга. Тем самым сводится к минимуму влияние процессов дин на одн и и, что необходимо в многозадачных ОС. Но от единичного изолированного процесса мало пользы. Сама концепция UNIX заключается в модульности, или ее основано на взаимодействии между отдельными процессами.

Для реализации взаимодействия нужно:

- обеспечить средства взаимодействия между процессами;
- одновременно исключить нежелательное воздействие одного процесса на другой.

Взаимодействие между процессами нужна для решения следующих задач:

- передача данных. Один процесс передает данные другому процессу, при этом их объем может меняться от десятков байт до нескольких мегабайт;
- совместное использование данных. Вместо копирования информации от одного процесса к другой, процессы могут совместно использовать одну копию данных, причем изменения, сделанные одним процессом, будут сразу заметны для другого. Количество процессов, взаимодействующих может быть больше двух. При совместном использовании ресурсов процессами может потребоваться где - протокол взаимодействия для сохранения целостности данных и исключения конфликтов, при доступе к ним;
- сообщения. Процесс может сообщать другой процесс или группу процессов о том, что где - какое событие произошло. Это может понадобиться, например, для синхронизации выполнения нескольких процессов.

В UNIX используют специальный механизм между процессорной взаимодействия (Inter - Process Communication, IPC).

Три типа IPC: сообщения, семафоры, разделяемую память называется System V IPC. Во многих версиях имеется еще одно средство IPC - сокеты.

3.1.10.1 Каналы

Каналы известны из команд shell, например

```
cat myfile | wc
```

При этом, вывод программы cat (), которая выводит содержимое файла myfile, передает на вывод программу wc, которая, в свою очередь, подсчитывает количество строк, слов и символов. Таким образом, два процесса обменялись данными. При этом используется программный канал, который обеспечивал равно направленную передачу данных между двумя задачами.

Для создания канала, используется системный вызов pipe ();

```
int pipe (int * filedes)
```

который возвращает два файловых дескриптора fildes [0] для записи в канал и fildes [1] для чтения из канала. Теперь, если один процесс записывает данные в fildes [0] другой может получить эти данные с fildes [1]. Вопрос только в том, как другой процесс может получить сам файловый дескриптор fildes [1]? Это подобно тому, как наследуются атрибуты при создании процесса. Процесс - потомок наследует и разделяет все назначении файловые дескрипторы родительского процесса. Иначе, доступ к дескрипторов fildes канала может получить сам процесс, который вызвал pipe () и его процессы - потомки.

В этом заключается серьезный недостаток каналов, поскольку они могут быть использованы для передачи данных только между родственными процессами. Каналы не могут быть использованы, как средства между процессорной взаимодействия между зависимыми процессами.

Как раз в примере процессов cat () и wc () не являются независимыми, на самом деле, оба создаются процессом shell и являются родственниками.

pipe ()

fork () fork () exec ()

exec ()

3.4 Создание канала cat () и wc ().

3.1.10.2. FIFO (First In First Out) / * (Second In Second In) * /

FIFO очень похожи на каналы, поскольку является однонаправленным средств передачи данных, причем, чтение данных выполняется в порядке их записи. Но, на отм и ну от программных каналов, FIFO имеет имена, которые позволяют независимым процессам получить доступ к этим объектам. Поэтому, иногда FIFO также называют именуемым каналом. FIFO является средством SystemV и не используется в BSD.

FIFO является отдельным типом файла (в команде ls - l). Для создания FIFO используют системный вызов mknod () int mknod (char * pathname, int mode, int dev)

Где pathname - имя файла в ФС (имя FIFO),

mode - флаг владения прав доступа и т.д., dev - при создании FIFO игнорируется.

FIFO можно создать из командной строки shell: $ mknod name p

После создания, FIFO может быть открыт на запись и чтение, причем запись и чтение могут выполняться в различных независимых процессах.

Каналы FIFO и обычные каналы работают по следующим правилам:

• • При чтении меньшего количества байтов, находящихся в канале или в FIFO, возвращается нужное количество байтов, остаток сохраняется для последующих чтений.

• • При чтении большого количества байтов, чем находится в канале или в FIFO, возвращается доступное количество байтов. Процесс, который читает из канала, должен соответствующим образом обработать ситуацию, когда прочитано меньше, чем заказано.

• • Если канал пустой и ни один процесс не открыл его на запись, при чтении с канала будет получено 0 байтов. Если один или более процессов открыли канал для записи, вызов read () блокируется до тех пор пока появятся данные (если для канала или FIFO не установлено флаг блокировки O_NDELAY).

• • Запись количества байтов, меньшей емкости канала или FIFO, гарантированно атомарно. Это означает, что в случае, когда несколько процессов одновременно записывают к каналу, порции данных от этих каналов не перемешиваются.

• • При записи большего количества байтов, чем это позволяет канал или FIFO, вызов write () блокируется до освобождения нужного места. При этом, атомарность операции не гарантируется. Если процесс пытается записать данные в канал, не открыто ни одним процессом на чтение, процесса

генерируется сигнал SIGPIPE, а вызов write () возвращает 0 с установлением ошибки (errno = EPIPE) (если процесс не установил обработку сигнала SIGPIPE, выполняется обработка по умолчанию - процесс завершается).

Пример программы приложения клиент - сервер, которая использует FIFO для обмена данными.

сервер

```
#include <sys / types.h>
#include <sys / stat.h>
#define FIFO "fifo.1"
#define MAXBUF 80 main
 () {

    int readfd, n;
    char buff [MAXBUFF]; / * Буфер для чтения данных с FIFO * / / * Создание
   спец файла FIFO с открытыми для всех правами доступа на чтение и запись *
   / if (mknod (FIFO, S_FIFO | 0666, 0) <o) {


    printf ( "Невозможно создать FIFO \ n"); exit (1); }


  / * Получено доступ к FIFO * / if ((read = open
   (FIFO, O_RDONLY)) <0) {

      printf ( "Невозможно открыть FIFO \ n"); exit (1); }


/ * Прочитать сообщение ( "Передача сообщения через FIFO") и вывод его на экран
  * /
    while ((n = read (readfd, buff, MAXBUFF))> 0) if (write (1,
    buff, n)! = n) {

      printf ( "Ошибка ввода \ n"); exit (1); }


/ * Закрываем FIFO, Удаляем FIFO-дело клиента * / close (readfd)
exit (0); }
```

клиент

```
#include <sys / types.h>
#include <sys / stat.h>
/ * Согласование об имени FIFO * /
#define FIFO "fifo.1" main
()
{Int writefd, n;
/ * Доступа к FIFO * / if ((writefd = open (FIFO, O_WRONLY)) <0)
{printf ( "Наименование открытого FIFO \ n"); exit (1);


/ * Передача сообщения сервера FIFO * /
if (write (writefd, "Передача сообщения через FIFO \ n", 18)! = 18)
```

{Printf ("Ошибка записи \ n"); exit (1); }

/ * Закрыть FIFO * / close
(writefd) / * Удалим FIFO *
/ if (unlink (FIFO) <0)

{Printf ("Невозможно удалить FIFO \ n"); exite (1); }

exite (0); }

3.1.10.3. Сообщение (Очереди сообщений)

Очереди сообщений является составной частью UNIX System V, они обслуживаются ОС, располагаются в адресном пространстве ядра и является системным ресурсом который разделяется.

Очереди сообщений имеют простую реализацию, но при этом демонстрируют некоторые архитектурные особенности общие для всех трех механизмов System VIPS. Каждая очередь имеет свой уникальный идентификатор. Процессы могут записывать и считывать сообщения из разных очередей, могут не ждать чтения этого сообщения любым процессом. Он может закончить свое исполнение, оставив в очереди сообщение, которое будет прочитано другим процессом позже.

Данная возможность позволяет процессам обмениваться структурированными данными, которые имеют следующие атрибуты:

- тип сообщения (позволяет мультиплекса ваты сообщение одной очереди).
- длина данных сообщений байтах (может быть нулевой).
- собственные данные (если длина нулевая, могут быть структурированы).

Очередь сообщений сохраняется в виде внутреннего однонаправленного связанного списка в адресном пространстве ядра. Для каждой очереди ядро создает заголовок очереди (msgid_ds), где содержится информация о правах доступа к очереди (msg_perm), ее текущее состояние (msg_cbytes - количество байтов и msg_gnum - количество сообщений в очереди), а также указательными на первое (msg_first) и последнее (msg_last) сообщения, хранящиеся в выгладит н "связной списка. Каждый элемент этого списка является отдельным сообщением.

Для создания новой очереди сообщений или для доступа к существующей используется вызов msgget ():

```
# include <sys / types.h>
# include <sys / ips.h>
# include <sys / msg.h>
int msgget (key_t key, int msgflag)
```

Функция возвращает дескриптор объекта очереди, или -1 в случае ошибки. Подобно файловом дескриптору, этот идентификатор используется процессом для работы с очередью сообщений.

Процесс может:

- размещать в очереди сообщения с помощью функции msgsnd ();
- получать уведомления определенного типа из очереди с помощью функции msgrcv ();
- управлять сообщениями с помощью функции msgctl ().

Перечисленные системные вызовы манипулирования сообщениями имеют следующий вид:

```
# include <sys / types.h>
# include <sys / ips.h>
# include <sys / msg.h>
```

int msgsnd (int msgid, const void * msgp, size_t msgsz, int msgflag) int msgrev (int msgid,
void * msgp, size_t msgsz, long msgtyp, int msgflag)

Tym msgid является дескриптором объекта, полученного в результате вызова msgget (). Параметр msgp указывает на буфер, который содержит тип сообщения и его данные, размер которого равен msgsz байт. Буфер имеет следующие поля:

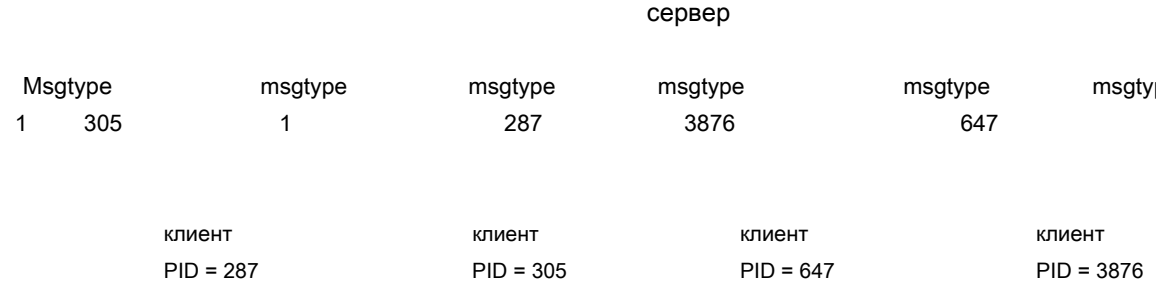
Lony msgtype тип сообщения Char
msgtext данные сообщения

Аргумент msgtyp указывает на тип сообщения и используется для их выборочного получения. Если msgtyp равен 0, функция msgrcv () получит первое сообщение из очереди. Если значение msgtyp меньше 0, функция msgrcv () получает сообщение с минимальным значением типа, меньше или такое равной абсолютному значению msgtyp. Если значение msgtyp выше 0, будет получено первое сообщение указанного типа.

Очереди сообщений имеют полезные свойства - в одной очереди можно мультиплекса ваты сообщение от различных процессов. Для демультиплексирования атрибут msgtype, на основе которого любой процесс может фильтровать сообщения с помощью функции msgrcv ().

Рассмотрим типичную ситуацию взаимодействия процессов, когда серверный процесс связывается с несколькими клиентами. Свойство мультиплексирования позволяет использовать для такого обмена одну очередь сообщений. Для этого сообщениям, которые направлении от кого-либо из клиентов сервера, будим присваивать значения типа, скажем такое равной 1. Если в теле сообщения клиент каким-то образом идентифицирует себя (например, передает свой PИD), тогда сервер сможет передать сообщение конкретному клиенту, присваивая тип сообщения равный идентификатору.

Поскольку функция msgrcv () позволяет принимать сообщения определенного типа (типов), сервер будет принимать сообщения с типом 1, а клиенты - сообщение с типами, которые равны идентификаторам их процессов.



Атрибут msgtype можно использовать для изменения порядка изъятия сообщений из очереди. Стандартный порядок получения сообщений есть аналогичный принципу FIFO - сообщение получаются в порядке их записи. Но используется тип, например, для назначения авторитета сообщений, этот порядок легко изменить.

пример:

1) файл описания mesg.h

```
# define MAXBUFF 80
# define PERM 0666
```

/ * Определим структуру сообщений. Она может отличаться от структуры msgbuf, но и должна содержать поле mtype. В данном случае структура сообщение состоит из буфера обмена * /

```
typedef struct our_msgbuf {

    long mtype;
    char buff [MAXBUFF]; }
```

```
    Message;
```

2) Сервер

```
 #  include <sys / types.h>
 #  include <sys / ips.h>
 #  include <sys / msg.h>
 main () {
/ * Структура сообщения (может отличаться от структуры msgbuf * / Message message;
key_t key;

int msgid, length, n;

/ * Получим ключ * /
if ((key = ftok ( "Server", "A")) <0) {printf ( "Невозможно
 получить ключ \ n"); exit (1); }



/ * Тип сообщений принимаемых * /
message.mtype = 1L;

/ * Создается очередь сообщений * /
if ((msgid = msgget (key, PERM | IPC _ CREAT)) <0) {printf
                ( "Невозможно создать очередь \ n"); exit (1); }



/ * Чтение сообщения * /
n = msgrcv (msgid, & message, sizeof (message), message, mtype 0)

/ * Если сообщение поступило, выведем его содержание на терминал * / if (n> 0)

 {If (write (1, message.buff, n)! = N) {printf (
 "Ошибка вывода / n"); exit (1); }}



else (printf ( "Ошибка чтения сообщения / n")); exit (1); {



  / * Удалить очередь поручено клиенту * / exit (0); }



3) Клиент /3.3/
 #  include <sys / types.h>
 #  include <sys / ips.h>
 #  include <sys / msg.h>
 main () {

/ * Структура сообщения (может отличаться от структуры msgbuf) * / Message message;
```

```
key_t key; int msgid,
length;

/ * Тип сообщение что отсылается, может использоваться для мультиплексирования * /
message.mtype = 1L;

/ * Получим ключ * /
if ((key = ftok ( "Server", "A")) <0) {printf ( "Невозможно
 получить ключ / n"); exit (1); }


/ * Получить доступ к очереди сообщений, очередь уже должна быть создана сервером * / if ((msgid = mgget
 (key 0)) <0
  {Printf ( "Невозможно получить доступ к очереди \ n"); exit (1); }



/ * Разместить строку в сообщении * /
if ((length = spintf (message.buff, "Работа с очередью сообщений \ n")) <0 {printf ( "Ошибка
 копирования в буфер \ n"); exit (1);}



/ * Передача сообщений * /
if (msgsnd (msgid, (void *) & message, length 0)! = 0) {printf (
 "Ошибка записи сообщения в записи \ n"); exit (1); }



/ * Удаление очереди сообщений * / if (msgctl
 (msgid, IPC_RMID 0) <0) {printf ( "Ошибка удаления
 очереди \ n"); exit (1); }


exit (0); }
```

Лекция 16

3.1.10.4. Семафоры. Общая теория.

Семафоры используются для синхронизации доступа нескольких процессов к ресурсам разделяемых. По обычаю принято рассматривать семафор, как сигнальный флажок (отсюда его название
), но, видимо, его лучше описывать как ключ.

3.1.10.4.1. Задачи синхронизации.

При параллельном выполнении задач возникает потребность в общении их между собой. Для обеспечения такого рода взаимодействия нужно специальным образом планировать и управлять ими в составе любой ОС. Для этого используют специальные средства и ограничения на последовательность выполнения взаимосвязанных параллельных процессов. Эти средства обеспечивают синхронизацию в общении. В каждом конкретном случае синхронизация задается с помощью синхронизирующих правил, устанавливаемых системой между процессами и определяют порядок их выполнения с целью обеспечения нужной взаимодействия. Существуют, как известно, отношение предварительности, отношение приоритетности, отношение взаимного исключения. Эти отношения и определяют сущность синхронизирующих правил. Реализация синхронизирующих правил осуществляется с помощью механизмов (средств) синхронизации.

Механизмы синхронизации различают по степени близости их к машинных команд и полноте свойств, которые они выполняют. Средства, которые имеют машинно - ориентированный характер и такие требующие от пользователей всякий раз составления специальных программных текстов для решения этой или иной задачи синхронизации называют низкоуровневыми средствами.

высокоуровневые средства представляют собой некоторую программную систему. Она предназначена для решения конкретной задачи синхронизации. Высокоуровневые средства централизованно доступны пользователям через определенный интерфейс. Особенности решения конкретной задачи синхронизации высокоуровневыми средствами скрытые от пользователей.

Различные механизмы синхронизации могут быть использованы для решения одной и той же конкретной задачи синхронизации. Для определения конкретного механизма нужно выполнять сравнение по ряду свойств и признаков. Чаще всего их определяют на качественном уровне. Наиболее такие используемых есть два класса свойств.

Первый класс определяет требования к средствам программирования, или к средствам алгоритмического упорядочения примитивов (базовых операций), с помощью которых выполняется обращение к механизмам синхронизации. Этот класс содержит такие свойства, как простота, гибкость, возможность проверки и понимания.

Второй класс определяет требования к реализации механизмов синхронизации. Он включает эффективность и надежность.

Простота использования определяется простотой синтаксиса и семантики механизма синхронизации. Предполагается, что структура примитива должна быть наглядной, а результатом проделанной им действия - понятным.

Гибкость определяет способность механизма быть использован для реализации известных типов отношений между процессами в явной и недвосмислений форме.

Перевиряемисть и понимание должны способствовать каком формальном типа анализа. Проверки могут быть разного типа: возможность возникновения тупиков или "голодовок", на достижение правильного взаимного исключения, на Неразрешенность потери управления процессом и.т.д.

Эффективность предполагает простоту реализации средств, имеются в составе рассматриваемых вопросов. Введение механизмов синхронизации не должно приводить к через мерной задержки в развитии процессов, которые их используют. Это требует установления ограничений на стратегии принятия решений при синхронизации, а также минимизации частоты включения и длины примитивов.

Надежность определяет вероятность правильной работы механизма не только в условиях, предусмотренных спецификациями пользователя согласно назначению механизма, но и в аномальных, отличных от ранее установленных.

Эти свойства взаимосвязаны и во многом имеют противоречия. Поэтому трудно построить механизм который удовлетворяет всем требованиям. Наиболее часто за основные выделяют требования простоты и

гибкости, а также эффективности. В настоящее время первое за все выделяют надежность, даже при потере эффективности.

3.1.10.4.2. Архитектура и основные вопросы построения механизмов синхронизации.

Согласно теоретическим и практическими данными (отдельно Т. Дейкстра в книге Логическое проектирование ОС - М. Мир 1981) установлено, что с помощью машинных команд, или используя средства языков программирования, принципиально возможно решить задачи синхронизации. Но нахождения правильного решения конкретной задачи - очень трудоемкий процесс. Еще больше сложнее процесс доказательства правильности найденного решения. В наше время известны только очень простые решения задач синхронизации.

Особенности каждой конкретной взаимодействия между двумя или более параллельными процессами определяются задачей синхронизации. Количество различных задач синхронизации ограничено. К ним относятся:

- взаимного исключения;
- производители - потребители;
- читатели - писатели;
- философы обедающих и.т.д.

Большинство задач в реальных ОС по согласованию параллельных процессов можно решить или с помощью этих типовых задач, или при помощи их модификации.

Сформируем в общем виде решение типовых задач синхронизации без формализации.

Задача взаимного исключения. Это фундаментальная по своему значению задача. Какая - либо ОС, руководит параллельными процессами, должна обеспечить тот или иной вариант реализации ее решения. Нужно согласовать работу n> = 2 параллельных процессов при использовании некоторого критического ресурса таким образом, чтобы удовлетворить следующие требования:

- одновременно в середине критической области должно находиться не более одного процесса;
- критические области не должны иметь приоритеты по отношению друг к другу;
- остановки какого - либо процесса снаружи его критической области при одинаковом времени поступления запросов на такое вхождение и равноприоритетности процессов не откладывается на неопределенный срок, а является конечным во времени;
- относительные скорости развития процессов неизвестные и произвольные;
- любой - процесс может переходить в какой - состояние, отличный от активного вне своей критической области;
- освобождение критического ресурса и выход из критической области должны быть выполнены процессом, который использует критический ресурс, за конечное время.

Задача "производитель - потребитель". Имеется большое количество вариантов постановления и решения такой задачи в рамках конкретной ОС. Наиболее простой случай, когда взаимодействуют два процесса с жестко распределены между ними функциями. Один процесс производит сообщения, предназначенные для восприятия и обработки другим процессом. Процесс, который производит сообщения называют производителем, а то что воспринимает сообщение - потребителем. Процессы взаимодействуют через некоторую обобщенную область памяти, которая по смыслу является критическим ресурсом. В эту область процесс - производитель должен размещать очередное сообщение (в простейшем случае предполагается, что область способна хранить только одно сообщение, и сообщения фиксированной длины), а процесс - потребитель должен считывать очередное сообщение.

- выполнять требования задачи взаимного исключения по отношению к критическому ресурса - обобщенной памяти для хранения сообщения;
- учитывая состояние обобщенной области памяти, которая характеризует возможность либо не возможность отсылки (получение) очередного сообщения.

Попытки процесса - производителя разместить очередное сообщение в области, из которой не было прочитано предварительное уведомление процессом - потребителем, должно быть блокировано. Процесс - производитель должен получить сообщение о невозможности размещения сообщения или быть переведенным в состояние ожидания возможности разместить очередное сообщение через некоторое время в

область памяти, по мере ее освобождения. Аналогично должно быть блокировано попытки процесса - потребителя считать сообщение из области в ситуации, когда процесс - производитель не разместил туда очередного сообщения. Возможны также несколько вариантов реакции на эту ситуацию - сообщение о невозможности считывания очередного сообщения или перевод процесса - потребителя в состояние ожидания поступления очередного сообщения. Если используется вариант с ожиданием изменения состояния успильненои области для сохранения сообщение, нужно обеспечить перевод ожидая процессов в состояние готовности каждый раз, когда меняется состояние области. Или процесс - производитель разместит очередного сообщение области. Оно теперь может быть считанное процессом - потребителем вызове.

Большая численность постановления задачи "производитель - потребитель" определяется тем, что количество как процессов - потребителей, так и процессов - производителей может быть больше одного. Каждый из таких процессов может устанавливать не только односторонний, но и двустороннюю связь через одну и ту же обобщенную область или другие области. Области могут хранить не одно, а большее количество сообщений.

Задача "читатели - писатели". Эта задача также имеет много вариантов. Наиболее характерная область использования этой задачи - при построении файловых систем ОС. В отношении некоторой области памяти, которая является по смыслу критическим ресурсом для параллельных процессов, которые работают с ней, выделяется два типа процессов:

первый тип - процессы - читатели. Они считывают одновременно информацию из области, если это допускается при работе с конкретным устройством памяти.

второй тип процессы - писатели. Они записывают информацию в область и могут делать это, только как исключая друг друга, так и процессы - читатели, запись должна довольствоваться на основе решения задачи взаимного исключения. Имеются различные варианты взаимодействия между процессами

- писателями и процессами - читателями. Если хотя бы один процесс - читатель использует ресурс, тогда он закрыт для использования всем процессам - писателям и доступен для использования всеми процессами - читателями. Во втором варианте, наоборот, больший приоритет у процессов - писателей. При появлении запроса от процесса - писателя нужно закрыть ресурс для использования всем процессам - читателям, которые выдают запрос позже его.

Задача "философы обедающих". В рамках этой задачи формируется требование на синхронизацию работы процессов, которые совместно используют группы ресурсов которые пересекаются.

К примеру. Пусть имеется три параллельных процесса X, Y, Z и три ресурса: P1 - устройство ввода с клавиатуры, P2 - печатающее устройство, P3 - накопитель на MD. Особенность развития процессов такова, что для пребывания процесса X в активном состоянии ему нужно выделить одновременно ресурсы P1 и P2, для пребывания процесса Y в активном состоянии ему нужно выделить одновременно ресурсы P2 и P3, для пребывания процесса Z в активном состоянии ему нужно выделить одновременно ресурсы P3 и P1. Скорость развития процессов произвольная. Переход из активного в другое состояние выполняется в непредсказуемые моменты. Надо обеспечить максимально параллельный и правильное процессов. Синхронизация в данном случае заключается в определенном упорядочении действий процессов по захвату ресурсов во избежание возможных блокировок одними процессами других. В данном случае возможны две опасности: возникновение тупиковой ситуации при распределении ресурсов и возникновения ситуации "голодания" по отношению некоторого процесса при распределении ресурса. Объяснение этой задачи дал Э. Дейкстра в соответствующей терминологии в 1971 году.

За круглым столом расстановке стулья, каждый из которых занимает определенный философ (в Дейкстры - 5). В центре стола - большое блюдо спагетти, а на столе лежит 5 вилок - каждая между двумя соседними тарелками. Каждый философ находится только в двух состояниях - либо он размышляет, или ест спагетти. Начать думать после еды философу ничего не мешает. Но, чтобы начать есть нужно выполнить ряд условий. Предполагается, что любой - философ, прежде чем начать есть, должен положить с общего блюда спагетти себе в тарелку. Для этого он должен держать в левой и правой руке по вилке, набрать спагетти в тарелку с их помощью, и не выпуская вилок с рук, начать есть. (Для упрощения в этом случае вопрос гигиены не рассматривается). Закончив есть философ кладет вилку слева и справа от своей тарелки и снова начинает розмишляты до тех пор, пока снова станет голодным. Нетрудно заметить, что вилки выступают в данной ситуации как ресурсы, которые пересекаются. Неприятная ситуация может возникнуть в случае, когда философы одновременно станут голодными и одновременно будут пытаться взять, например, свою левую вилку. В данном случае возникает тупиковая ситуация, так как

никто из них не может по условию начать есть, не имея второй вилки. Но вторая вилка может появиться для какого - либо философа только от соседа справа и одновременно не хочет положить свою левую вилку на стол и.т.д. Вторая неприятность (голодание) может уже по отношению не всех, а только одного процесса. В обществе философов такая ситуация возникает в случае мятежа двух соседей слева и справа против философа, по отношению к которому устраиваются козни. Каждый раз, когда последний хочет удовлетворить голод, мятежники, опережая его, поочередно забирают вилки то слева, то справа от него. Так согласованные действия злоумышленников приводят жертву к вынужденному голоданию, так как он никогда не может воспользоваться обеими вилками.

Отсюда, для реализации каких - либо механизма синхронизации требуется специальная аппаратная поддержка. В наиболее простом случае используется система прерываний. На одно процессорных машинах активный процесс может перейти в какой - промежуточный состояние только по мере возникновения прерываний в ответ на некоторое событие, возникшее в машине. Тогда при решении взаимного исключения, которое является обязательным для решения всех задач синхронизации, можно использовать следующую схему управления системой прерываний.

Если у процесса возникает необходимость использовать некоторый критический ресурс, тогда при его использовании никакой другой процесс не должен иметь возможности обратиться к этому ресурсу. Обеспечить такую гарантию очень просто - процесс, которому нужен критический ресурс, нужно блокировать обработку всех прерываний и тем самым выйти из под контроля ОС. Процесс становится монопольным "хозяином" ЦБ до тех пор, пока он сам нерозмаскеруе систему прерываний. Поскольку на интервале между блокировкой и деблокирования прерывания ни один процесс, кроме того что захватил ресурс, не может развиваться, то тем самым одновременно и решается задача взаимного исключения. Один процесс исключил все остальные в их попытке использовать критический ресурс.

Для такой схемы механизма синхронизации текст программы процесса, ограничено командами блокировки и деблокирования прерываний, и состоит из команд обработки некоторого критического ресурса. Поэтому это является критической областью. В данном случае для реализации взаимного исключения критической области делится свойством непрерывности.

Схема механизма синхронизации на основе использования прерываний имеет отрицательные стороны. Данный механизм является не гибким средством, имеет низкую эффективность. Поскольку при использовании команд критической области компьютер никак не реагирует на внешние и внутренние события, другие процессы в системе не могут развиваться. Отсюда, при долгом временном использовании критической области и частом обращении к ним, мультипрограммный режим работы компьютера может быть существенно нарушено и, в конце концов, приблизиться к однопрограммные.

Этого можно избежать, если ввести еще один элемент при обращении к критическому ресурса. В предварительную схему можно ввести переменную (переменную состояния), которая могла бы в любой момент характеризовать состояние критического ресурса. Эта переменная должна быть доступной всем процессам, которые с ее помощью хотят использовать критический ресурс. Причем в отношении этой переменной процессы должны не только осуществлять операции чтения, но и изменять ее значение. При этом эта переменная становится также критическим ресурсом. Чтобы захватить основной критический ресурс, надо предварительно захватить вспомогательный - переменную состояния. Предварительно нужно выйти из критической области по отношению переменной состояния. При таком решении можно использовать, например систему прерываний и достичь при этом гораздо большей эффективности, чем по предыдущей схеме.

Так, для реализации критической вспомогательной области можно, как и раньше, воспользоваться блокировкой и деблокирования прерываний. Но при этом очевидно, что длина использования вспомогательной критической области будет малой. Нужно только проверить и установить, если это допустимо, нужное значение переменной состояния. Тогда проблему взаимного исключения можно решить при следующих условиях. Все процессы будут попадать в свою основную критическую область только через вспомогательную критическую область при условии, что вход в основную область будет проходить только тогда, когда переменная состояния указывает на незанятость основного критического ресурса.

Теперь уже не возникает необходимости блокировать прерывания на период выполнения основной критической области. Гарантом взаимного исключения является изменение состояния, значение которой должен проверять каждый процесс, построив для этого вспомогательную критическую область. Предполагается, что после обработки основного критического ресурса процесс должен войти в вспомогательной критической области, но уже с другой целью. Нужно с помощью переменной состояния отметить, что основной критический ресурс стал свободен и его могут пытаться захватить другие процессы.

Данная упрощенная двухэтапная схема построения механизма синхронизации для решения задачи взаимного исключения показывает, что основная критическая область для достижения свойства взаимного исключения размещается в своеобразные логические скобки - вспомогательные критические области по проверке и переустановке переменной состояния. Эти скобки можно рассматривать как примитивы. Эти примитивы являются неделимыми и взаимно исключают друг друга. При реализации рассматриваемой схемы примитивы выполняют в составе ОС в форме макрокоманд, указывая в качестве параметров изменении состояния.

При более детальном рассмотрении двухэтапной схемы решения задачи взаимного исключения возникает вопрос о том, как поступить тем процессам, которые при вхождении в вспомогательной критической области находят с помощью анализа значение переменной состояния, то основной критический ресурс в текущий момент занято каким-то процессом? На практике используются два решения: или реализуется режим "занятого", или "пассивного ожидания". В случае использования "занятого" режима ожидания процесс, который нашел, что критический ресурс занят, периодически организует сам или с помощью системы проверку заявленного ресурса. Процесс повторно и периодически пытается выполнить заметил восторга ресурса. Режим "пассивного ожидания" предусматривает, что если процесс выполняет примитив восторга ресурса и оказывается, что он в текущий момент времени уже занят,

Для реализации режима пассивного ожидания в состав механизма синхронизации нужно ввести новые элементы. Нужно реализовать очередь процессов, которые пытались, но не смогли успешно выполнить заметил восторга критического ресурса. Как какую - либо очередь, ее требуется формировать и обслуживать по определенным правилам. Поэтому в состав механизма синхронизации включаются средства для решения задачи упорядочения процессов в очереди, а также средства для последующего извлечения процессов из очереди при увольнении критического ресурса. Последнее средство - это основа для решения задачи ре активации процессов, которые ждут возможности монопольно использовать критический ресурс.

3.1.10.4.3. Семафорная техника синхронизации и упорядочения процессов

Понятие семафорного механизма ввел Э. Дейкстра в 1965 Семафор - это переменная специального типа, которая доступна параллельным процессам для проведения над ней только двух видов операций "закрытия" и "открытие", которые называют соответственно P - и V - операциями. Они примитивами в отношении семафора, который указывается в качестве параметра операции. Семафор в данном случае выполняет роль вспомогательного критического ресурса.

P - и V - операции неделимые в своем исполнении и взаимно исключают один другого. Семафорный механизм работает по двухэтапной схеме, и использует при этом режим "пассивного" ожидания.

Поэтому в состав механизма включают средства формирования и обслуживания очереди в ожидании процессов, которым не удается выполнить с успехом примитив "закрытия" семафора.

Используют на практике семафоры различной модификации и механизмов. Рассмотрим некоторые, наиболее распространенные виды семафорных механизмов. В силу взаимного исключения примитивов попытки в разных параллельных процессах (например, в мультипроцессорной системе) одновременно выполнять заметил на одном и том же семафоре приведет к успеху только одного из процессов. Причем невозможно сказать, какой именно из конкурирующих процессов это сделает. Все остальные процессы будут данным процессом взаимно исключены на время выполнения примитива.

Простые семафоры. две модификации

Рассмотрим семафоры, допустимыми значениями которых являются целые числа. При этом возможны два варианта построения допустимого диапазона значений.

Первый вариант - допустимы только целые положительные числа, включая ноль. Особый случай в таком варианте, когда допустимые значения семафора - только 0 и 1. Такой семафор называется двоичный.

Второй вариант - допустимы не только положительные, но и отрицательные целые числа как значение семафора. Каждый из вариантов в свою очередь имеет модификации, обусловленные прежде всего особенностью построения примитивов по "открытию" и "закрытию" семафора. Обобщенный смысл операции закрытия

- проверить текущее значение семафора, и если оно больше нуля, тогда перейти к выполнению следующей за примитивом операции, предварительно уменьшив на единицу значение семафора. Если это значение семафора равно или меньше нуля, или семафор закрыто, тогда процесс, который пытается выполнить заметил, должен быть переведенным в состояние "пассивного" ожидания. Чтобы получать более наглядно эту характерную ситуацию, будим говорить, что процесс "засыпает" на семафоре, если он при проверке оказался закрытым. Операция закрытия связана с обязательным увеличением на

единицу значение семафора и кроме того выполняется "побудка" одного или нескольких процессов, которые "спят" на семафоре. Как правило, "пробуждение" состоит в переводе процесса по очереди ожидания до семафора в очередь готовности к ЦБ.

Рассмотрим первый вариант P - и V - операций. Основное допущение - семафор может принимать положительные и отрицательные целые значения. При этом возможны модификации в зависимости от начального значения семафора.

Для описания параллельных процессов в алгоритмах используется нотация "параллельного Паскаля" для лучшего понимания. Программные тексты, взятые в скобки COBEGIN и COEND и имеют цифровые пометки, представляют собой описание программ параллельных процессов. Алгоритмы примитивов имеют следующий вид:

{Алгоритмы P (S)} S =
S-1,
IF S <0 THEN <остановить процесс и разместить в очереди ожидания до семафора S> ELSE <продолжить процесс>;

{Алгоритм V (S)}
IF S <0 THEN <поместить один из ожидая процессов очереди семафора S в очередь готовности>;

S = S + 1;

Использование примитивов: BEGIN
S: = 1; COBEGIN

1: P (S); KO; V (S) ... .. 2: P
(S); KO; V (S) COEND END.

В приведенном примере используется семафор для решения задачи взаимного исключения. В программе КО обозначают критическую область, строится в составе процесса для обеспечения обработки некоторого критического ресурса. Каждая критическая область в параллельных процессах, согласно постановлению задачи, должна исключать друг друга. Переменная S - это семафор, которому присваивается начальное значение, равное 1. Имя семафора указывается в качестве параметра при обращениях к P - и V - примитивам. Показаны только два параллельных процесса, которые обозначены цифрами 1 и 2. Если процессы 1 и 2 будут пытаться одновременно выполнить заметил P, то это удается успешно сделать только одному из них (которому - неизвестно). Пусть это сделал процесс 2, тогда он не закрывает семафор S, после чего начинается выполнение критической области в составе. Процесс 1 в ситуации, которая рассматривается заснет на семафоре S. Тем самым будет гарантировано взаимное исключение. Процесс 1 временно выбыл из борьбы за критический ресурс. После выполнения процессом 2 примитива V семафор S открывается, тем самым указывая на возможность захвата каким-то (только одним) процессом критического ресурса освободившееся. При этом процессом 2 проводится побудка процесса 1.

На реальном уровне возможно одно из двух решений в отношении процессов, которые переводятся из очереди ожидания семафора в очередь готовности при выполнении примитива V. Это два возможных решения задачи ре активации:

- процесс при его активизации (при избирателе из очереди готовности) снова пытается выполнить заметил P, считая предыдущую попытку неудачной;
- процесс при размещении его в очереди готовности отмечается таким, успешно выполнил заметил P. Тогда при его активизации (при извлечении процесса из очереди готовности) управления будет выполнено не на повторное выполнение примитива P, а на команду, которая идет за ним.

При первом решении задачи реактивации возможно возникновение тупиков.

Продемонстрируем это на примере задачи взаимного исключения.

Пусть процесс 2 в некоторый момент времени пытается выполнить и выполняет примитив P (S). Тогда S становится таким равной 0. Пусть дальше процесс 1 пытается выполнить заметил P (S). Он не сможет это сделать, так как семафор закрыто., Или S = 0. Процесс 1 "засыпает" или размещается в очереди к семафора,

а семафор становится отрицательным, или S = 1. После выполнения критической области процесс 2 выполняет примитив V (S), при этом S = 0, а процесс 1 переводится в очередь готовности. Пусть через некоторое время процесс 1 будет активизирована (выбранный для выполнения из очереди готовности) он во второй пытается выполнить заметил P (S), но это ему не удается, так как S = 0. Процесс один "засыпает" на семафоре, а значение семафора при этом стало S = 1. Если через некоторое время процесс 2 пытается выполнить заметил P (S), он тоже "уснет". Пробуждать процессы некому, попали в тупике.

При использовании второго решения задачи ре активации тупике в данной последовательности действий не будет. На самом деле все выполняется таким образом до момента выдачи процессом 2 примитива V (S). Пусть заметил V (S) выполнено и S = 0. Рассмотрим наиболее неблагоприятно случай. Процесс 2 снова пытается войти в свою критическую область и закрыть семафор, или выдает P (S). Войти в критическую область ему не удается (S = 0) и процесс "засыпает", а значение семафора переустанавливается в 1. Пусть через некоторое время готов к выполнению процесс 1 переводится в активное состояние. Согласно второму решению задачи ре активации он сразу попадает в свою критическую область. При этом никакой другой процесс одновременно не будете иметь доступ к критическому ресурсу (семафор закрыто, а единственный пресс-конкурент, рассматривается в примере, спит на семафоре). После выполнения своей критической области процесс 1 выдает примитив V (S). Процесс 2 выводится из очереди ожидания до семафора в очередь готовности, а S = 0. Через некоторое время процесс 2 может с успехом войти в свою критическую область. Тупике нет, решение корректно. Путем аналогичных рассуждений можно показать, что оно правильно и для n> 2 параллельных процессов.

Многочисленные семафоры. Особенностью такого механизма является возможность проверки в одном примитиве не одного, а нескольких семафоров и обработки их по определенным правилам. В отношении семафоров, по-прежнему строятся очереди в ожидании процессов. Но проблема в этом случае существенно сложнее, чем в случае использования простых семафоров, так как отсутствует тривиальный связь между одним семафором и очередь ожидания. Очередь ожидания может быть связана с составляющей условием, которое проверяется в соответствующих примитив. Возможности многочисленных семафоров можно продемонстрировать на решении задачи синхронизации типа "философы обедающих".

```
VAR
   FORKS: ARRAY 1..S OF SEMAPHORE; I:
  INTEGER; BEGIN I = 5; REPEAT



     FORKS [I]: = 1; I = I-1 UNTIL I = 0;
COBEGIN
       1: BEGIN ... <ТЕЛО ПРОЦЕССА> ...
          END;
..... I: BEGIN

   VAR LEFT, RIGTH: 1..5;
  BEGIN
   LEFT = (I - 1) MOD 5, RIGTH
   = (I + 1) MOD 5, REPEAT

     <РАЗМЫШЛЕНИЯ>
     P (FORKS [LEFT]; FORKS [RIGTH])
  <ИСПОЛЬЗОВАНИЕ РЕСУРСОВ ИЛИ ИЖа>; V (FORKS
  [LEFT]; FORKS [RIGTH]) FOREVE END END COEND
  END
```

(В алгоритме параллельные процессы циклические, для их описания используется конструкция, ограниченная операторами REPEAT и FOREVE).

В этой программе каждый философ - это отдельный процесс с номером I = 1..5. Зная свой номер, философ (процесс) в состоянии определить номер соседа слева: LEFT = (И-1) MOD 5 и номер соседа справа: RIGTH = (I + 1) MOD 5. Каждой вилке (их всего 5, по количеству философов) поставлено в соответствие отдельный семафор. Все семафоры составляют отдельный массив FORKS. Если первая вилка в настоящее время занята философом, тогда значение семафора FORKS [1] равна 0. Каждый примитив P - и V проводит обработку сразу двух семафоров из массива. Понятно, примитивы неделимые и исключают друг друга во времени. При выполнении примитива P процесс может "заснуть" или на одном семафоре, или на другом, или на обоих сразу. Такие блокировки развития процесса возникают, если в момент выполнения примитива P соответствующие семафоры проверяемых на равенство 0. В предложенном решении исключено активное ожидание, которое пришлось бы реализовать, если для решения данной задачи синхронизации использовались бы не многочисленные, а простые семафоры. Реализация механизма многочисленных семафоров сложнее, так как нужен средство обслуживания в одном примитиве не одной, а двух очередей с учетом различных стеков семафоров. При использовании многочисленных семафоров решение полученные нагляднее и естественнее, чем при использовании простых семафоров.

Архивные семафоры, тест - семафоры. В таком механизме синхронизации при каждом выполнении примитивов P - и V допускается изменение значения семафора S не в 1, а на любое значение R, обозначается P (S, R) и V (S, R). Если в результате выполнения примитива P полученное значение семафора остается положительным, тогда процесс продолжается. В противном случае процесс "засыпает" на семафоре. Задача ре активации решается путем использования пассивного ожидания проверяемого.

Рассмотрим использование такого семафора на примере решения задачи синхронизации типа "читачи-писатели ", которые работают с некоторой файловой системой.

Задача решается при следующих исходных условиях. Количество процессов-писателей и процессов-читателей произвольная. С файловой системой могут одновременно работать (читать информацию) сколько угодно процессов-читателей. Но одновременная работа по записи и считыванию информации недопустима. Отсюда, с некоторым файлом одновременно не может работать любой процесс-писатель и процесс-читатель. Также недопустима работа процессов-писателей над файлами. Решение задачи:

```
VAR S: SEMAPHORE;
Q, R: INTEGER;
BEGIN R = 1; Q = n; {Инициализация семафора и инкремента} COBEGIN

 СПОЖИВАЧ1: REPEAT <произвольные
операторы>; P (S, R); <Чтения из базы
данных>; V (S, R);

FOREVER; ..... потребительница:
REPEAT <произвольные операторы>; P (S, R); <Чтения из базы
данных>; V (S, R); FOREVER;


  ВИРОБНИК1: REPEAT <произвольные
операторы>; P (S, Q) <Запись в базу
данных>; V (S, Q) FOREVER;


..... ВИРОБНИКn: REPEAT
<произвольные операторы>; P (S, Q) <Запись в базу данных>; V
(S, Q) FOREVER;
```

COEND
END

В этой программе семафора S при инициализации присваивается значение, равное n. При построении механизма, который называется тест - семафорами, вводится более усложнена логика работы P. По параметры этого примитива указывается: имя семафора S; значение на которое нужно изменить S; значение Q, с которой нужно поменять новое значение семафора (после его модификации). Если полученное значение S такое, что значение Q - S положительное, тогда процесс, который выполнил заметил, продолжается. В противном случае процесс "засыпает" на семафоре.

3.1.10.5. Семафоры в UNIX

Для синхронизации процессов в ОС UNIX используют семафороподибни средства. Этот механизм непосредственно доступен для использования только системными процессами. Пользовательским процессам доступны другие средства синхронизации обмена информацией (каналы, сигналы и.т.д.). В UNIX события рассматриваются как адрес структуры данных, связанной каким-то образом с произошедшим.

Семафоры UNIX является одной из форм ИРС и для SYSTEM V имеют следующие характеристики:

- семафоры есть не один счетовод, а группа, состоящая из нескольких счетоводов, объединенных общими признаками (например, деструктором объекта, правами доступа и.т.д.)
- Каждое из этих чисел может принимать любое неотъемлемое значение в пределах, определенных системой (а не только значение 0 и 1).

Для каждой группы семафоров ядро поддерживает структуру данных semid_ds, которые содержат следующие поля:

Struct ipc_perm sem_perm	описание прав доступа
Struct sem * sem_base	указатель на первый ел.мас.сем.
Ushort sem_nsems	количество семафоров в группе
time_t sem_otime	время последней операции
time_t sem_ctime	время последнего изменения

Значение конкретного семафора из набора сохраняется во внутренней структуре sem: Ushort semval
значение семафора
Pid_t sempid Идентификатор процесса, который выполнил последнюю операцию над семафором Ushort semncnt
Количество процессов, ожидающих увеличения значения семафора Ushort semrcnt Количество процессов,
ожидающих обнуления семафора

Кроме собственного значения семафора, в структуре sem сохраняется идентификатор процесса, который вызвал последнюю операцию над семафором, количество процессов, ожидающих увеличения значения семафора, на количество процессов, ожидающих, когда значение семафора станет таким равной 0. Эта информация позволяет ядру выполнять операции над семафорами.

Для получения доступа к семафора (и для его создания если он не существует) используется системный вызов semget ():

```
# include <sys / types.h>
# include <sys / ips.h>
# include <sys / sem.h>

int semget (key_t key, int nsems, int semflag)
```

В случае успешного завершения операции функция возвращает дескриптор объекта, в случае неудачи - 1. Аргументы nsems задает количество семафоров в группе. В случае, когда не создается, а получается доступ к существующему семафора, этот аргумент игнорируется. Аргумент semflag определяет право доступа к семафора и флажки для его создания (IPS_GREAT, IPC_EXCL).

После получения дескриптору объекта процесс может выполнять операции над семафорами, подобно тому, как после получения файлового дескриптору процесс может читать и записывать данные в файл. Для этого используется системный вызов semop ():

```
# include <sys / types.h>
# include <sys / ips.h>
# include <sys / sem.h>

int semop (int semid, struct sembuf * semop, size_t nops)
```

За второй аргумент функции передается указатель на структуру данных, определяет операции, которые нужно выполнить над семафором с дескриптором semid. Операций может быть несколько, а их количество указывается в последнем аргументе nops. Важно, что ядро обеспечивает атомарность выполнения критических участков операций (например, проверка значений - изменение значений) по отношению к другим процессам.

Каждый элемент набора операций semop имеет вид:

```
struct sembuf {short
sem_num;                    / * Номер семафора в группе * /
   short sem_op;            / * Операция * /
  short sem_flg;             / * Флаг операции * /
};
```

UNIX допускает три возможные операции над семафором, которые определяются полем semop:

- Если значение semop положительное, тогда текущее значение семафора увеличивается на это значения.
- Если значение semop равен 0, процесс ожидает, пока семафор НЕ обнулится.
- Если значение semop отрицательное, процесс ожидает, пока значение семафора не станет большим или таким равной абсолютному значению semop. Затем абсолютное значение semop вычитается из значения семафора.

Можно заметить, что первая операция меняет значение семафора (безусловное исполнение), а третья - проверяет, а затем меняет значение семафора (условное выполнение).
При работе с семафорами процессы взаимодействующих должны договориться об их использовании и кооперативно проводить операции над семафорами. ОС не накладывает ограничений на использование семафоров. Отдельно процессы свободно решают, какое значение семафора является разрешающая на какое значение меняется семафор и.т.д.

Таким образом, при работе с семафорами процессы используют различные комбинации из трех операций, определенных системой, по-своему трактуя значение семафоров.

В качестве примера рассмотрим два случая использования бинарного семафора (или значение которого могут принимать только 0 и 1). В первом примере значение 0 является таким что позволяет, а 1 запирает некоторый ресурс разделяемой (файл, память разделяемой ...), ассоциированный с семафором. Определим операции освобождающие ресурс и запирают его:

```
Static struct sembuf sop_lock [2] = {

  0,0,0 / * ожидать обнуления семафора * /
  0,1,0 / * затем увеличивает значение семафора на 1 * /}; static struct
 sembuf sop_unlock [1] = {

  0, -1,0 / * обнулить значение семафора * /};
```

Таким образом, для запирания ресурса процесс выполняет вызов:
semop (semid, & sop_lock, 2);

который обеспечивает атомарно выполнения двух операций:

1. Ожидания доступности ресурса. В случае, если ресурс уже занято (значение семафора равно 1), выполнение процесса будет приостановлено до освобождения ресурса (значение семафора равно 0).

2. Запирание ресурса. Значение семафора устанавливается таким равной 1. Для освобождения ресурса процесс должен выполнить вызов:

```
Semop (semid, & sop_unlock [0], 1);
```

а освобождает:

```
semop (semid, & sop_unlock [0], 1);
```

Во второй ситуации операции вышли простые (код стал короче), но этот подход имеет потенциальную опасность: при создании семафора, его значение устанавливается таким равной 0, и во втором случае он сразу запирает ресурс. Для преодоления данной ситуации процесс, который первым создал семафор должен вызвать операцию sop_unlock, но в этом случае процесс инициализации семафора перестанет быть атомарным и может быть прерван другим процессом, который, в свою очередь, изменит значение семафора. В конце концов, значение семафора станет таким равной 2, что навредит нормальной работе с ресурсом разделяемой.

Проблему можно решить следующим образом:

```
/ * Создаем семафор, если он уже существует semget возвращает ошибку, поскольку указано флаг IPC_EXCL * /


if ((semid = semget (key, nsems, perms | IPC_GREAT | IPC_EXCL)) <0) {

   if (errno = EEXIST) {

   / * Действительно, ошибка вызвана существованием объекта * / if ((semid =
   semget (key, msems, perms)) <0)
   return (-1); / * Возможно, не хватило системных ресурсов * /}

   else return (-1); / * Возможно, не хватило системных ресурсов * /}

   / * Если семафор создан, анализируем его * / else semop
   (semid, & sop_unlock [0], 1);
```

лекция 17

3.1.10.6. Память разделяемой. теоретические основы

Общая схема работы с ресурсами разделяемых следующая - есть определенный процесс-автор, создающий ресурс с какими-либо параметрами. При создании ресурса разделяемой памяти задаются три параметра - ключ, права доступа и размер области памяти. После создания ресурса к нему могут быть подключены процессы, желающие работать с этой памятью. Соответственно, имеется действие подключения к ресурсу с помощью ключа, который генерируется по тем же правилам, что и ключ для создания ресурса. Понятно, что здесь имеется момент некоторой рассинхронизации, который связан с тем, что потребитель ресурса (процесс, который будет работать с ресурсом, но не является его автором) может быть запущен и начать подключаться к запуску автора ресурса.

Рассмотрим функции, необходимые для работы с разделяемыми ресурсами.

- Создание общей памяти.
  int shmget (key_t key, int size, int shmemflg)

key - ключ памяти разделяется
size - размер раздела памяти, который должен быть создан shmemflg -
флаги

Данная функция возвращает идентификатор ресурса, который ассоциируется с созданным по данному запросу разделяемым ресурсом. То есть в рамках процесса по аналогии с файловыми дескрипторами каждому ресурса определяется его идентификатор.
Ключ - это общесистемный атрибут; идентификатор - внутрипроцессорное понятия. С помощью этой функции можно как создать **новый разделяемый ресурс "память" (в этом случае во флагах должен быть указан IPC_CREAT), а также можно подключиться к существующему разделяемому ресурсу. Кроме того, в возможных флагах может быть указан флаг IPC_EXECL, он позволяет** проверить и подключиться к существующему ресурсу - если ресурс существует, то функция подключает к нему процесс и возвращает код идентификатора, если же ресурс не существует, то функция возвращает -1.

Доступ к разделяемой памяти

char * shmat (int shmid, char * shmaddr, int shmflg) shmid -
идентификатор разделяется ресурса
shmaddr - адрес, с которого мы хотели бы разместить разделяемую память я

При этом, если значение shmaddr - адрес, то память будет подключена, начиная с этого адреса, если его значение - ноль, то система сама подберет адрес начала. Также в качестве значений этого аргумента могут быть некоторые предопределенные константы, которые позволяют организовать, в частности выравнивание адреса по странице или началу сегмента памяти.

shmflg - флаги. Они определяют различные режимы доступа, в частности, есть флаг SHM_RDONLY.

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

Открепление разделяемой памяти:

- int shmdt (char * shmaddr)

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

4. Управление памятью:
int shmctl (int shmid, int cmd, struct shmid_ds * buf); shmid -
идентификатор разделяемой памяти cmd - команда управления.
В частности, могут быть команды:

- IPC_SET (изменить права доступа и владельца ресурса - для этого надо иметь идентификатор автора данного ресурса или суперпользователя),

- IPC_STAT (пригласить состояние ресурса - в этом случае заполняется информация в структуру, указатель на которую передается третьим параметром,

- IPC_RMID (уничтожение ресурса - после того, как автор создал процесс - с ним работают процессы, которые подключаются и отключаются, но не уничтожают ресурс, а с помощью данной команды мы уничтожаем ресурс в системе)

3.1.10.7. Память разделяемой в Unix

В составе любой ОС имеется распределитель ОП, поскольку ни один из процессов не может развиваться без ресурса ОП. Вместе с этим алгоритмы, по которым выполняется распределение памяти, очень розноманитни.

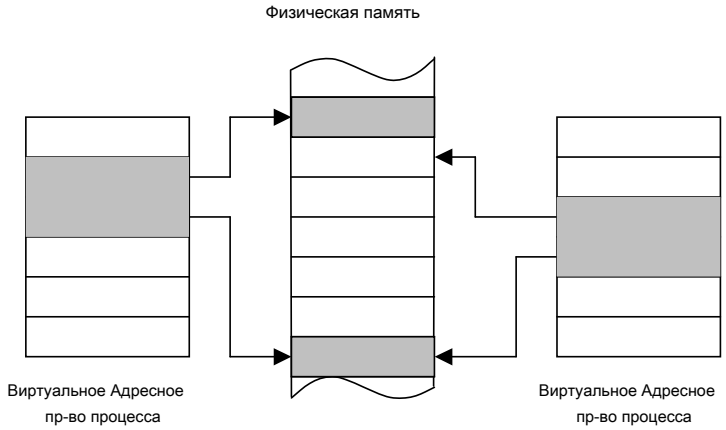


Рис. 19 Коллективная память

Механизм разделяемой памяти позволяет нескольким процессам получить отражение некоторых страниц со своей виртуальной памяти на общую область физической памяти. Благодаря этому, данные, находящиеся в этой области памяти, будут доступны для чтения и модификации всем процессам, которые подключились к данной области памяти.

Процесс, который подключился к разделяемой памяти, может затем получить указатель на некоторую адрес в своем виртуальном адресном пространстве, который соответствует данной области разделяемой памяти. После этого он может работать с этой областью памяти аналогично тому, как если бы она была выделена динамически (например, путем обращения к malloc ()), однако, как уже говорилось, сама по себе область памяти разделяется не уничтожается автоматически даже после того, как процесс, который создал или который использовал ее, перестанет с ней работать.

Рассмотрим набор системных вызовов для работы с памятью. Пример создания общей памяти.

```
#include <sys / types.h>
#include <sys / ipc.h>
#include <sys / shm.h>
int shmget (key_t key, int size, int shmemflg)
```

аргументы этого вызова:

key - ключ для доступа к разделяемой памяти;

size задает размер области памяти, к которой процесс желает получить доступ. Если в результате вызова shmget () буде создана новая область разделяемой памяти, то ее размер будет соответствовать значению size. Если же процесс подключается к существующей области разделяемой памяти, то значение size должно быть в пределах ее размера, иначе вызов вернет -1. Заметим, что если процесс при подключении к существующей области разделяемой памяти указал в аргументе size

значение, меньше ее фактического размера, то в дальнейшем он сможет получить доступ только к первым size байт этой области.

Отметим, что в заглавном файле < sys / shm.h> определены константы SHMMIN и SHMMAX, задающий минимально возможный и максимально возможный размер области разделяемой памяти. Если процесс пытается создать область разделяемой памяти, размер которой не удовлетворяет этим границам, системный вызов shmget () закончится неудачей.

Третий параметр определяет флаги, управляющие поведением вызова. Подробнее алгоритм создания / подключения ресурсу, был описан выше.

В случае успешного завершения вызов возвращает положительное число - дескриптор области памяти, в случае неудачи - 1.

Пример доступа к разделяемой памяти.

```
#include <sys / types.h>
#include <sys / ipc.h>
#include <sys / shm.h>
        char * shmat (int shmid, char * shmaddr, int shmflg)
```

С помощью этого вызова процесс подсоединяет область разделяемой памяти, дескриптор которой указан в ***shmid,***

к своему виртуальному адресному пространству. после выполнения этой операции процесс сможет читает ТЫ и модифицировать данные, которые находятся в области разделяемой памяти, адресуя ее как любую другую область в своем собственном виртуальном адресном пространстве.

В качестве второго аргумента процесс может указать виртуальный адрес в своем адресном пространстве, начиная с которого необходимо подсоединить разделяемую память. Чаще всего, однако, как значение этого аргумента передается 0, что означает, что система сама может выбрать адрес начала разделяемой памяти. Передача конкретного адреса в этом параметре имеет смысл в том случае, если, например, в разделяемую память записываются указатели на нее же (например, в ней сохраняется связанный список) в этой ситуации для того, чтобы использование этих указателей мало смысл и было корректным для всех процессов, подключенных к памяти, важно, чтобы во всех процессах адреса начала области разделяемой памяти совпадал.

Третий аргумент является комбинацией флагов. как значение этого аргумента может быть указан флаг ***SHM_RDONLY,*** указывающий на то, что область, которая подсоединяется будет использоваться только для чтения.

Эта функция возвращает адрес, начиная с которой будет отображаться и что присоединяется Коллективная память. В случае неудачи вызов возвращает -1.

Пример открепление разделяемой памяти.

```
#include <sys / types.h>
#include <sys / ipc.h>
#include <sys / shm.h> int shmdt (char
    * shmaddr)
```

Данный вызов позволяет отсоединить память, что разделяется, ранее присоединенную помощью вызова ***shmat ().***

параметр ***shmaddr -*** адрес прикрепленной к процессу памяти, который был получен при вызове shmat ().

В случае успешного выполнения функция возвращает 0, в случае неудачи -1

Пример управления памятью, що.роздиляється

```
#include <sys / types.h>
#include <sys / ipc.h>
#include <sys / shm.h>
int shmctl (int shmid, int cmd, struct shmid_ds * buf)
```

Данный вызов используется для получения или изменения процессом управляющих параметров, связанных с областью разделяемой памяти, наложение и снятие блокировки на нее и ее уничтожения. Аргументы вызова - дескриптор области памяти, команда, которую необходимо выполнить, и структура, описывает управляющие параметры области памяти. Тип shmid_ds описан в заголовочном файле < sys / shm.h>, и представляет собой структуру, в полях которой хранятся права доступа к области памяти, ее размер, количество процессов, присоединенных к ней в данный момент, и статистика обращений к области памяти.

Возможные значения аргумента cmd: IPC_STAT - скопировать структуру, описывает управляющие параметры области памяти по адресу, указанному в параметре buf

IPC_SET - заменить структуру, описывает управляющие параметры области памяти, на структуру, находящуюся по адресу, указанному в параметре buf. Выполнить эту операцию может процесс, у которого эффективный идентификатор пользователя совпадает с владельцем или создателем очереди, или процесс с правами привилегированного пользователя, при этом процесс может изменить только владельца области памяти и права доступа к ней.

IPC_RMID - удалить очередь. Как уже говорилось, удалить очередь может только процесс, у которого эффективный идентификатор пользователя совпадает с владельцем или создателем очереди, или процесс с правами привилегированного пользователя.

SHM_LOCK, SHM_UNLOCK - блокировать или разблокировать область памяти. Выполнить эту операцию может только процесс с правами привилегированного пользователя.

Общая схема работы с общей памятью в рамках одного процесса.

```
#include <sys / types.h>
#include <sys / ipc.h>
#include <sys / shm.h>

  int putm (char *) int
waitprocess (void);

 int main (int argc, char ** argv) {

        key_t key; int shmid;
        char * shmaddr;


        key = ftok ( "/ tmp / ter ',' S '); shmid = shmget (key, 100, 0666 |
        IPC_CREAT)
```

```
shmaddr = shmat (shmid, NULL, 0); / * Подключение к памяти * /

putm (shmaddr) / * Работа с ресурсом * / waitprocess ();


shmctl (shmid, IPC_RMID, NULL); / * Уничтожение ресурса * / return 0; }
```

В данном примере считается, что ***putm ()*** и ***waitprocess () -*** какие функции пользователя, определенные в другом месте.

лекция 18

3.2. Подсистема управления ведением / выводом.

Одной из главных функций ОС является управление всеми устройствами введеннявиведення компьютера. ОС должна передавать устройствам команды, перехватывать прерывания и обрабатывать ошибки; она также должна обеспечивать интерфейс между устройствами и остальной частью системы. В целях развития интерфейс должен быть одинаковым для всех типов устройств (независимость от устройств). Физическая организация устройств ввода-вывода

Устройства ввода-вывода делятся на два типа: блок-ориентированные устройства и байт-ориентированные устройства. Блок-ориентированные устройства хранят информацию в блоках фиксированного размера, каждый из которых имеет свой собственный адрес. Самое распространенное блок-ориентированное устройство - диск. Байт-ориентированные устройства не адресуемая и не позволяют производить операцию поиска, они генерируют или потребляют последовательность байтов. Примерами являются терминалы, строчные принтеры, сетевые адаптеры. Однако некоторые внешние устройства не относятся ни к одному классу, например, часы, которые, с одной стороны, не адресуются, а с другой стороны, не порождают потока байтов. Это устройство только выдает сигнал прерывания в некоторые моменты времени.

Внешнее устройство обычно состоит из механического и электронного компонента. Электронный компонент называется контроллером устройства или адаптером. Механический компонент представляет собственно устройство. Некоторые контроллеры могут управлять несколькими устройствами. Если интерфейс между контроллером и устройством стандартизован, то независимые производители могут выпускать совместимые как контроллеры, так и устройства.

Операционная система обычно имеет дело не с устройством, а с контроллером. Контроллер, как правило, выполняет простые функции, например, преобразует поток бит в блоки, состоящие из байт, и осуществляют контроль и исправление ошибок. Каждый контроллер имеет несколько регистров, используемых для взаимодействия с центральным процессором. В некоторых компьютерах эти регистры являются частью физического адресного пространства. В таких компьютерах нет специальных операций ввода-вывода. В других компьютерах адреса регистров ввода-вывода, называемых часто портами, образуют собственное адресное пространство за счет введения специальных операций ввода-вывода (например, команд IN и OUT в процессорах i86).

ОС выполняет ввода-вывода, записывая команды в регистры контроллера. Например, контроллер гибкого диска IBM PC принимает 15 команд, таких как READ, WRITE, SEEK, FORMAT и т.д. Когда команда принята, процессор оставляет контроллер и занимается другой работой. при завершении команды контроллер организует прерывания для того, чтобы передать управление процессором операционной системе, которая должна проверить результаты операции. Процессор получает результаты и статус устройства, читая информацию из регистров контроллера.

Организация программного обеспечения ввода-вывода

Основная идея организации программного обеспечения ввода-вывода заключается в разбивке его на несколько уровней, причем нижние уровни обеспечивают экранирование особенностей аппаратуры от верхних, а те, в свою очередь, обеспечивают удобный интерфейс для пользователей.

Ключевым принципом является независимость от устройств. Вид программы не должен зависеть от того, читает она данные с гибкого диска или с жесткого диска.

Очень близкой к идее независимости от устройств является идея одинакового именования, то есть для именования устройств должны быть приняты единые правила.

Другим важным вопросом для программного обеспечения ввода-вывода является обработка ошибок. Вообще говоря, ошибки следует обрабатывать как можно ближе к аппаратуре. Если контроллер обнаруживает ошибку чтения, то он должен попытаться ее так скорректировать. Если же это ему не удается, то исправлением ошибок должен

заняться драйвер устройства. Много ошибок могут исчезать при повторных попытках выполнения операций ввода-вывода, например, ошибки, вызванные наличием пылинок на головках чтения или на диске. И только если нижний уровень не может справиться с ошибкой, он сообщает об ошибке верхнему уровню.

Еще один ключевой вопрос - это использование блокирующих (синхронных) и неблокирующих (асинхронных) передач. Большинство операций физического введеннявиведення выполняется асинхронно - процессор начинает передачу и переходит на другую работу, пока не наступает прерывание. Пользовательские программы намного легче писать, если операции ввода-вывода блокирующие - после команды READ программа автоматически приостанавливается до тех пор, пока данные не попадут в буфер программы. ОС выполняет операции ввода-вывода асинхронно, но представляет их для пользовательских программ в синхронной форме.

Последняя проблема заключается в том, что одни устройства являются разделяемыми, а другие выделенными. Диски - это коллективные устройства, так как одновременный доступ нескольких пользователей к диску не представляет собой проблему. Принтеры - это выделенные устройства, потому что нельзя смешивать строки, печатаются различными пользователями. Наличие выделенных устройств создает для операционной системы некоторые проблемы.

Для решения поставленных проблем целесообразно разделить программное обеспечение ввода-вывода в четыре слоя (рисунок 2.30):

Обработка прерываний,
Драйверы устройств,
Независимый от устройств слой операционной системы,
Пользовательский слой программного обеспечения.

Многоуровневая организация подсистемы ввода-вывода

3.2.1. Основные концепции, лежащие в основе подсистемы управления ведением / выводом в ОС UNIX.

Мы уже обсуждали проблемы организации ввода / вывода в ОС UNIX. При этом нужно понимать, что в любом случае мы остаемся на концептуальном уровне. Если вам нужно написать драйвер некоторого внешнего

устройства для некоторого конкретного варианта ОС UNIX, то неизбежно придется внимательно читать документацию. Однако знание общих принципов будет полезно.

Традиционно в ОС UNIX выделяются три типа организации ввода / вывода и, соответственно, три типа драйверов. Блочный ввод / вывод главным образом предназначен для работы с каталогами и обычными файлами файловой системы, на базовом уровне имеют блочную структуру. В ЛС. 2.4.5 и 3.1.2 указывалось, что на уровне пользователя теперь можно работать с файлами, прямо отражая их в сегменты виртуальной памяти. Эта возможность рассматривается как верхний уровень блочного ввода / вывода. На нижнем уровне блочный ввод / вывод поддерживается блочными драйверами. Блочный ввод / вывод, кроме того, поддерживается системной буферизацией (см. П 3.3.1).

Символьное ввод / вывод служит для прямого (без буферизации) выполнения обменов между адресным пространством пользователя и соответствующим устройством. Общей для всех символьных драйверов поддержкой ядра является обеспечение функций пересылки данных между пользовательскими и ядерным адресными пространствами.

Наконец, потоковый ввод / вывод (который мы не будем рассматривать в этом курсе слишком подробно по причине большого количества технических деталей) похож на символьный ввод / вывод, но по причине наличия возможности включения в поток промежуточных обрабатывающих модулей имеет существенно большей гибкостью.

Принципы системной буферизации ввода / вывода

Традиционным способом снижения накладных расходов при выполнении обменов с устройствами внешней памяти, имеют блочную структуру, является буферизация блочного ввода / вывода. Это означает, что любой блок устройства внешней памяти считывается прежде всего в некоторый буфер области основной памяти, называемой в ОС UNIX системным кэшем, и уже оттуда полностью или частично (в зависимости от вида обмена) копируется в соответствующее пользовательское пространство.

Принципами организации традиционного механизма буферизации является, во-первых, то, что копия содержимого блока содержится в системном буфере до тех пор, пока не возникнет необходимость его замещения по причине нехватки буферов (для организации политики замещения используется разновидность алгоритма LRU, см. П 3.1 .1) . Во-вторых, при выполнении записи любого блока устройства внешней памяти реально выполняется только обновления (или образование и наполнения) буфера кэша. Действительный обмен с устройством выполняется или при выталкивании буфера вследствие замещения его содержания, либо при выполнении специального системного вызова sync (или fsync), поддерживаемого специально для насильственного выталкивания во внешнюю память обновленных буферов кэша.

Эта традиционная схема буферизации вошла в противоречие с развитыми в современных вариантах ОС UNIX средствами управления виртуальной памятью и особенно с механизмом отображения файлов в сегменты виртуальной памяти. (Мы не будем подробно объяснять тут суть этих противоречий и предложим читателям поразмышлять над этим.) Поэтому в System V Release 4 появилась новая схема буферизаци й, пока используемая параллельно со старой схеме.

Суть новой схемы заключается в том, что на уровне ядра фактически воспроизводится механизм отображения файлов в сегменты виртуальной памяти. Во-первых, напомним о том, что ядро ОС UNIX действительно работает в собственной виртуальной памяти. Эта память имеет более сложную, но принципиально такую же структуру, что и пользовательская виртуальная память.

Иными словами, виртуальная память ядра является сегментно-страничной, и наравне с виртуальной памятью пользовательских процессов поддерживается общей подсистемой управления виртуальной памятью. Из этого следует, во-вторых, что практически любая функция, обеспечивается ядром для пользователей, может быть обеспечена одними компонентами ядра для других его компонентов. В частности, это относится и к возможностям отображения файлов в сегменты виртуальной памяти.

Новая схема буферизации в ядре ОС UNIX главным образом основывается на том, что для организации буферизации можно не делать почти ничего специального. Когда один из пользовательских процессов открывает не открытие до сих пор файл, ядро образует новый сегмент и подключает к этому сегменту открываемый файл. После этого (независимо от того, будет ли пользовательский процесс работать с файлом в традиционном режиме с использованием системных вызовов read и write или подключит файл к сегменту своей виртуальной памяти) на уровне ядра работа будет проводиться с тем ядерным сегментом, к которому подключен файл на уровне ядра. Основная идея нового подхода заключается в том, что устраняется разрыв между управлением виртуальной памятью и общесистемной буферизацией (это нужно было бы сделать давно, поскольку очевидно,

Почему же нельзя отказаться от старого механизма буферизации? Все дело в том, что новая схема предполагает наличие некоторой непрерывной адресации внутри объекта внешней памяти (должен существовать изоморфизм между отображаемым и отображенным объектами). Однако, при организации файловых систем ОС UNIX достаточно сложно распределяет внешнюю память, особенно касается i-узлам. Поэтому некоторые блоки внешней памяти приходится считать изолированными, и для них оказывается выгоднее использовать старую схему буферизации (хотя, возможно, в завтрашних вариантах UNIX и удастся полностью перейти к унифицированной новой схеме).

Системные вызовы для управления вводом / выводом

Для доступа (то есть для получения возможности дальнейшего выполнения операций ввода / вывода) к файлу любого вида (включая специальные файлы) пользовательский процесс должен выполнить предварительное подключение к файлу с помощью одного из системных вызовов open, creat, dup или pipe. Программные каналы и соответствующие системные вызовы мы рассмотрим в п. 3.4.3, а пока немного более подробно, чем в п. 2.3.3, рассмотрим другие "инициализируем" системные вызовы.

Последовательность действий системного вызова open (pathname, mode) следующая: анализируется непротиворечивость входных параметров (Главным образом, касаются флагам режима доступа к файлу)

выделяется или находится пространство для описателя файла в системной области данных процесса (u-области);

в общесистемной области выделяется или находится существующее пространство для размещения системного описателя файла (структуры file)

проводится поиск в архиве файловой системы объекта с именем "pathname" и образуется или оказывается описатель файла уровня файловой системы (vnode в терминах UNIX V System 4);

выполняется связывание vnode ранее образованной структурой fi le. Системные вызовы open и creat (почти) функционально эквивалентны. Любой существующий файл можно открыть с помощью системного вызова creat, и любой новый файл можно создать с помощью системного вызова open. Однако, применительно к системному вызову creat мы должны подчеркнуть, что в случае своего природного применения (для создания файла) этот системный вызов создает новый элемент соответствующего каталога (в соответствии с заданным значением pathname), а также создает и соответствующим образом инициализирует новый i -вузол.

Наконец, системный вызов dup (duplicate - скопировать) приводит к образованию нового дескриптора уже открытого файла. Этот специфический для ОС UNIX системный вызов служит исключительно для целей перенаправления ввода / вывода. Его выполнение заключается в том, что в u-области системного пространства пользовательского процесса образуется новый описатель открытого файла, содержащего новообразованный дескриптор файла (целое число), но ссылается на уже существующую системное структуру file и содержит те же признаки и флаги, которые соответствуют открытому файлу-образцу.

Другими важными системными вызовами являются системные вызовы read и write. Системный вызов read выполняется следующим образом:

в общесистемной таблице файлов находится дескриптор указанного файла, и определяется, законно обращение от данного процесса к данному файлу в указанном режиме;

на некоторое (короткий) время устанавливается Синхронизационные блокировки на vnode данного файла (содержание описателя не должно меняться в критические моменты операции чтения)

выполняется собственно чтения с использованием старого или нового механизма буферизации, после чего данные копируются, чтобы стать доступными в пользовательском адресном пространстве.

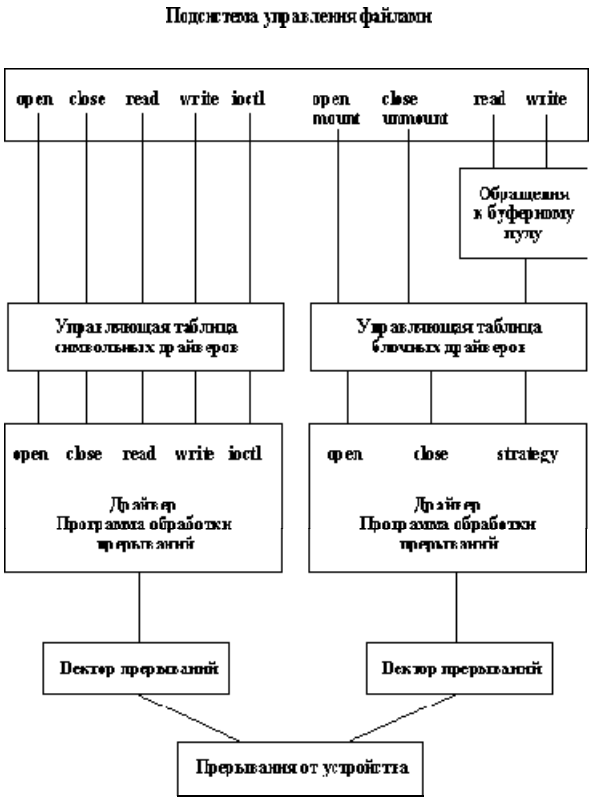
Операция записи выполняется аналогичным образом, но меняет содержимое буфера буферного пула.

Системный вызов close приводит к тому, что драйвер обрывает связь с соответствующим пользовательским процессом и (в случае последнего по времени закрытия устройства) устанавливает общесистемный флаг "драйвер свободен".

Наконец, для специальных файлов поддерживается еще один "специальный" системный вызов ioctl. Это единственный системный вызов, который обеспечивается для специальных файлов и не обеспечивается для остальных разновидностей файлов. Фактически, системный вызов ioctl позволяет произвольным образом расширить интерфейс любого драйвера. Параметры ioctl включают код операции и указатель на некоторую область памяти пользовательского процесса. Всю интерпретацию кода операции и соответствующих специфических параметров проводит драйвер.

Естественно, что поскольку драйверы главным образом предназначены для управления внешними устройствами, программный код драйвера должен содержать соответствующие средства для обработки прерываний от устройства. Вызов индивидуальной программы обработки прерываний в драйвере происходит из ядра операционной системы. Подобным же образом в драйвере может быть объявлен вход "timeout", к которому обращается ядро при окончании ранее заказанного драйвером времени (такой временной контроль является необходимым при управлении не слишком интеллектуальными устройствами).

Общая схема интерфейсной организации драйверов показана на рисунке 3.5. Как показывает этот рисунок, с точки зрения интерфейсов и общесистемного управления различаются два вида драйверов - символьные и блочные. С точки зрения внутренней организации выделяется еще один вид драйверов - потоковые (st ream) драйверы. Однако по своему внешнему интерфейсу потоковые драйверы не отличаются от символьных.



Блочные драйверы предназначаются для обслуживания внешних устройств с блочной структурой (магнитных дисков, лент и т.д.) и отличаются от других тем, что они разрабатываются и выполняются с использованием системной буферизации. Иными словами, такие драйверы всегда работают через системный буферный пул. Как видно на рисунке 3.5, любое обращение к блочному драйверу для чтения или записи всегда проходит через предварительную обработку, которая заключается в попытке найти копию нужного блока в буферном пуле.

В случае, если копия необходимого блока не находится в буферном пуле или если по какойлибо причине нужно заменить содержимое некоторого обновленного буфера, ядро ОС UNIX обращается к процедуре strategy соответствующего блочного драйвера. Strategy обеспечивает стандартный интерфейс между ядром и драйвером. С использованием библиотечных подпрограмм, предназначенных для написания драйверов, процедура strategy может организовывать очереди обменов с устройством, например, с целью оптимизации движения магнитных головок на диске. Все обмены, выполняемые блочным драйвером, выполняются с буферной памятью. Перепись нужной информации в память соответствующего пользовательского процесса производится программами ядра, заведующими управлением буферами. символьные драйверы

Символьные драйверы главным образом предназначены для обслуживания устройств, обмены с которыми выполняются посимвольно, или строчками символов переменного размера. Типичным примером символьного устройства является простой принтер, который принимает один символ за один обмен.

Символьные драйверы не используют системную буферизацию. Они напрямую копируют данные из памяти пользовательского процесса при выполнении операций записи или в память пользовательского процесса при выполнении операций чтения, используя собственные буфера.

Следует отметить, что имеется возможность обеспечить символьный интерфейс для блочного устройства. В этом случае блочный драйвер использует дополнительные возможности процедуры strategy, позволяющие выполнять обмен без применения системной буферизации. Драйвера, который обладает одновременно блочным и символьным интерфейсами, в файловой системе заводится два специальных файлы, блочный и символьный. При каждом обращении драйвер получает информацию о том, в каком режиме он используется. поточные драйверы

Как отмечалось в, основным назначением механизма потоков (streams) является повышение уровня модульности и гибкости драйверов со сложной внутренней логикой (более всего это относится к драйверам, которые реализуют развитые сетевые протоколы). Спецификой таких драйверов является то, что большая часть программного кода не зависит от особенностей аппаратного устройства. Более того, часто оказывается выгодно по-разному комбинировать части кода.

Все это привело к появлению поточной архитектуры драйверов, которые представляют собой двунаправленный конвейер обрабатывающих модулей. В начале конвейера (ближе к пользователю процесса) находится заголовок потока, к которому в первую очередь поступают обращения по инициативе пользователя. В конце конвейера (ближе к устройству) находится обычный драйвер устройства. В промежутке может располагаться произвольное число обрабатывающих модулей, каждый из которых оформляется в соответствии с обязательным потоковым интерфейсом.

3.2.4.2. Буферизация. подсистема буферизации

Любой запрос на ввод-вывод в блок-ориентированном устройству превратится в запрос к подсистеме буферизации, которая представляет собой буферный пул и комплекс программ управления этим пулом.

Буферный пул состоит из буферов, которые находятся в области ядра. Размер отдельного буфера равен размеру блока данных на диске.

С каждым буфером связана специальная структура - заголовок буфера, в котором содержится следующая информация:

Данные о состоянии буфера:

- занят / свободен,
- чтение / запись,
- признак отложенной записи,
- ошибка ввода-вывода.

Данные об устройстве - источнике информации, находящейся в этом буфере:

- тип устройства,
- номер устройства,
- номер блока на устройстве.

Адрес буфера.

Ссылка на следующий буфер в очереди свободных буферов, предназначенных для ввода-вывода каком устройству.

Упрощенный алгоритм выполнения запроса к подсистеме буферизации приведен на рисунке 5.14. Данный алгоритм реализуется набором функций, наиболее важные из которых рассматриваются ниже.

Функция bwrite - синхронная запись. В результате выполнения данной функции немедленно инициируется физический обмен с внешним устройством. Процесс, который выдал запрос, ожидает результат выполнения операции ввода-вывода. В данном случае в процессе может быть предусмотрена собственная реакция на ложную ситуацию. Такой тип записи используется тогда, когда необходима гарантия правильного завершения операции ввода-вывода.

Функция bawrite - асинхронная запись. При таком типе записи также немедленно инициируется физический обмен с устройством, однако завершения операции ввода-вывода процесс не ждет. В этом случае возможны ошибки ввода-вывода не могут быть переданы в процесс, который выдал запрос. Такая операция записи целесообразна при поточной обработке файлов, когда ожидания завершения операции ввода-вывода не обязательно, но есть уверенность в повторении этой операции.

Функция bdwrite - отложенная запись. При этом передача данных из системного буфера не производится, а в заголовке буфера делается отметка о том, что буфер заполнен и может быть выгружен, если потребуется освободить буфер.

Функции bread и getblk - получить блок. Каждая из этих функций ищет в буферном пуле буфер, содержащий указанный блок данных. Если такого блока в буферном пуле нет, то в случае использования функции getblk осуществляется поиск любого свободного буфера, при этом возможна выгрузка на диск буфера, содержащего в заголовке признак отложенной записи. При использовании функции bread при отсутствии заданного блока в буферном пуле организуется его загрузки в какой-нибудь свободный буфер. Если свободных буферов нет, то также производится выгрузка буфера с отложенной записью. Функция getblk используется тогда, когда содержание зарезервированного блока не существенно, например, при записи на устройство данных, объем которых равен одному блоку.

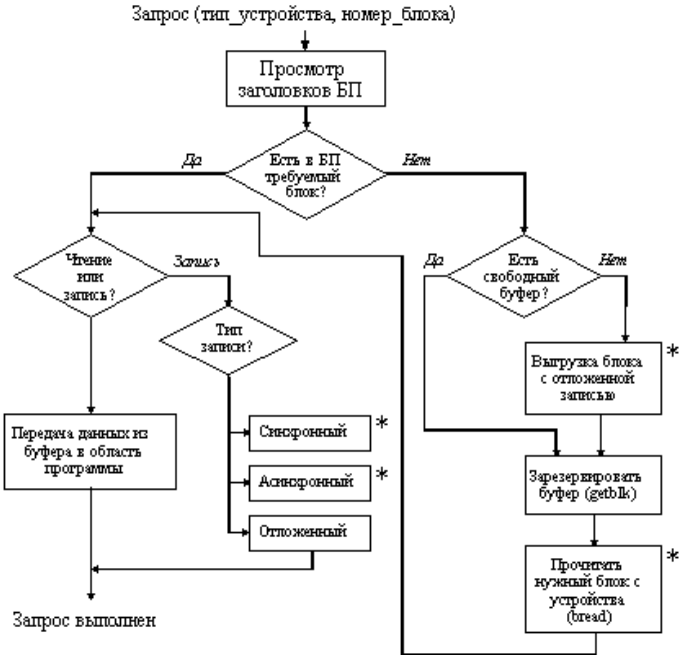


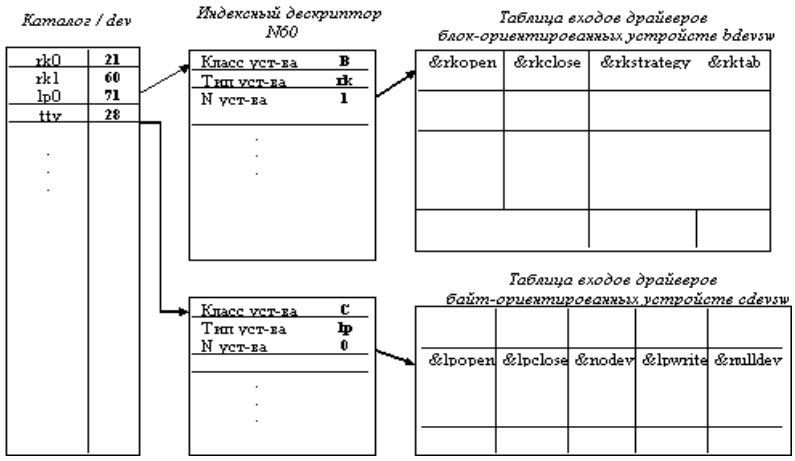
Рис. 5.14. Упрощенная схема выполнения запросов подсистемой буферизации

Таким образом, за счет отложенной записи в системном буферном пуле задерживается некоторое число блоков данных. При возникновении запроса к внешней памяти просматривается содержание буферного пула. При этом вероятность обнаружения данных в системном пуле достаточно велика. Это обусловлено объективными свойствами пространственной и временной локальности данных. Согласно описанному алгоритму буферизации, в системном буферном пуле оседает наиболее часто используемая информация. Таким образом, система буферизаци й выполняет роль кэш-памяти по отношению к диску. Кэширования диска уменьшает среднее время доступа к данным на диске, однако при этом снижается надежность файловой системы, так как в случае внезапной потери питания или отказа диска может произойти потеря блоков, содержащихся в системном буфере.

Выше был описан механизм старого буферного кэша, который использовался в предыдущих версиях UNIX System V в качестве основного дискового кэша. В UNIX System V Release 4 используется новый механизм, основанный на отражении файлов в физическую память. Однако старый механизм кэширования также сохранен, так как новый кэш используется только для блоков данных файлов, но непригоден для кэширования административной информаци й диска, такой как inode, каталог и т.д.

Новый буферный кэш

Расположение данных в файле характеризуется их смещением от начала файла. Так как ядро ссылается на любой файл с помощью структуры vnode, то расположение данных в файле определяется парой vnode / offset. Доступ к файлу по адресу vnode / offset достигается с помощью сегмента виртуальной памяти типа segmap, подобный сегмента segvn, используемом системой виртуальной памяти. Метод доступа к файлам, основанный на сегментах segmap, называется новым буферным кэшем. Этот способ кэширования использует модель страничного доступа к памяти для ссылок на блоки файлов. Размер страницы, используемой новым буферным кэшем, машиннозависим. Для отображения блоков файлов используется адресное пространство ядра, которое также как и пользовательское виртуальное пространство описывается структурой as.



драйверы

Драйвер - это совокупность программ (секций), предназначенная для управления передачей данных между внешним устройством и оперативной памятью.

Связь ядра системы с драйверами (рисунок 5.15) обеспечивается с помощью двух системных таблиц:

bdevsw - таблица блок-ориентированных устройств и cdevsw -
таблица байт-ориентированных устройств.

Для связи используется следующая информация из индексных дескрипторов специальных файлов:

класс устройства (байт-ориентированное или блок-ориентированное),
тип устройства (лента, гибкий диск, жесткий диск, устройство печати, дисплей, канал связи и т.д.)

номер устройства.

Класс устройства определяет выбор таблицы bdevsw или cdevsw. Эти таблицы содержат адреса программных секций драйверов, причем одна строка таблицы соответствует одному драйверу. Тип устройства определяет выбор драйвера. Типы устройств пронумерованы, то есть тип определяет номер строки выбранной таблицы. Номер устройства передается драйверу в качестве параметра, так как в ОС UNIX драйверы спроектированы в расчете на обслуживание нескольких устройств одного типа.

Такая организация логической связи между ядром и драйверами позволяет легко настраивать систему на новую конфигурацию внешних устройств путем модификации таблиц bdevsw и cdevsw.

Драйвер байт-ориентированного устройства в общем случае состоит из секции открытия, чтения и записи файлов, а также секции управления режимом работы устройства. В зависимости от типа устройства некоторые секции могут отсутствовать. Это определенным образом отражено в таблице cdevsw. Секции записи и чтения обычно используются совместно с модулями обработки прерываний ввода-вывода от соответствующих устройств.

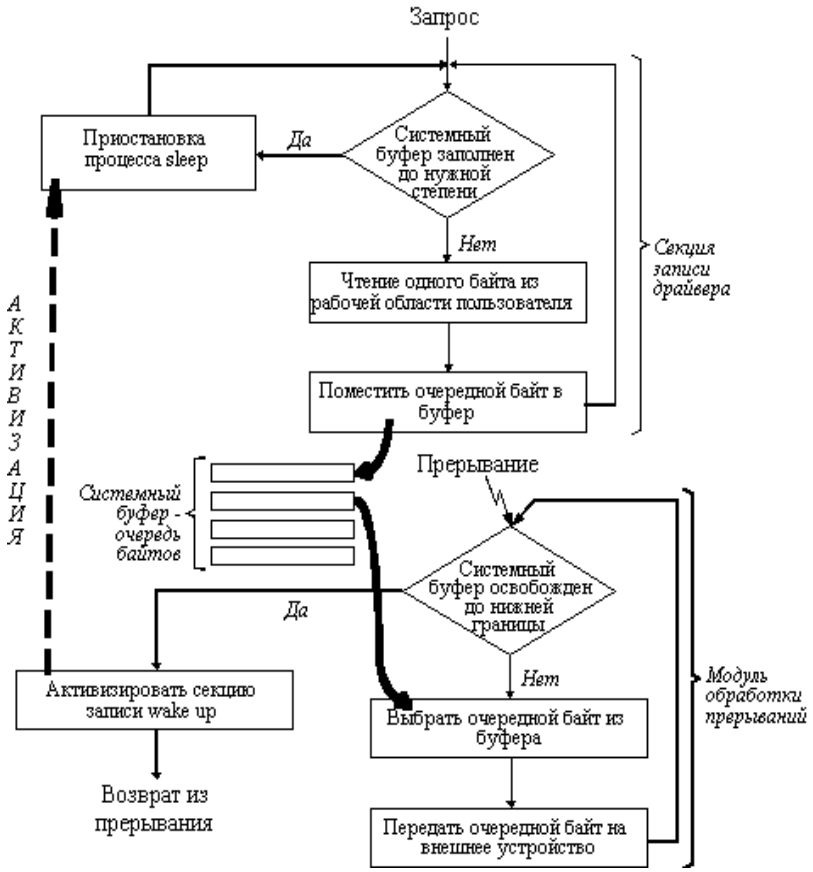


Рис. 5.16. Взаимодействие секции записи драйвера с модулем обработки прерывания

На рисунке 5.16 показано взаимодействие секции записи драйвера байт-ориентированного устройства с модулем обработки прерываний. Секция записи осуществляет передачу байтов из рабочей области программы, выдавшей запрос на обмен, в системный буфер, организованный в виде очереди байтов. Передача байтов идет до тех пор, пока системный буфер не заполнится до некоторого, заранее определенного в драйвере, уровня. В результате секция записи драйвера приостанавливается, выполнив системную функцию sleep (аналог функций типа wait). Модуль обработки прерываний работает асинхронно секции записи. Он вызывается в моменты времени, определяются готовностью устройства принять следующий байт. Если при очередном прерывании оказывается, что очередь байтов уменьшилась до определенной нижней границы, то модуль обработки прерываний активизирует секцию записи драйвера путем обращения к системной функции wakeup.

Аналогично организована работа при чтении данных с устройства.

Драйвер блок-ориентированного устройства заключается в общем случае из секций открытия и закрытия файлов, а также секции стратегии. Кроме адресов этих секций, в таблице bdevsw указаны адреса так называемых таблиц устройств (rktab). Эти таблицы содержат информацию о состоянии устройства - занято или свободно, указатели на буфера, для которых активизированы операции обмена с устройством, а также указатели на цепочку буферов, в которых находятся блоки данных, предназначенные для обмена с устройством.

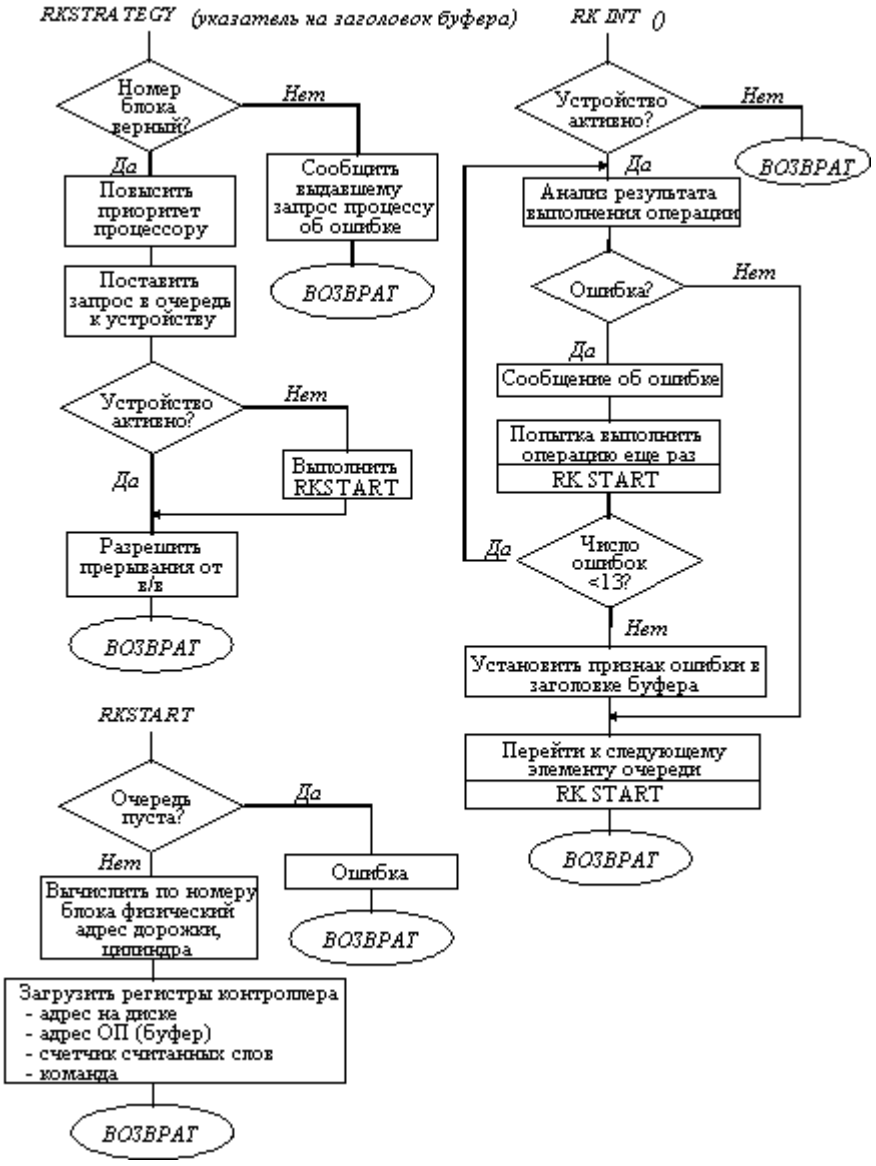


Рис 5.17. Структурная схема драйвера диска типа RK

На рисунке 5.17 приведена упрощенная схема драйвера жесткого диска. Секция стратегии - rkstrategy - выполняет постановку запроса на ввод-вывод в очередь к устройству путем присоединения указанного буфера к цепочке буферов, уже предназначенных для обмена с устройством. В случае необходимости секция стратегии запускает устройство (программа rkstart) для выполнения чтения или записи блока с устройства. Вся информация о необходимой операции может быть получена из заголовка буфера, указатель на который передается секции стратегии как аргумент.

После запуска устройства управления возвращается процесса, который выдал запрос драйвера.

Об окончании ввода-вывода каждого блока устройство оповещает операционную систему сигналом прерывания. Первое слово вектора прерываний на аппарате адрес секции драйвера - модуля обработки прерываний rkintr. Модуль обработки прерываний проводит анализ правильности выполнения ввода-вывода. Если зафиксирована ошибка, то несколько раз повторяется запуск этой же операции, после чего драйвер переходит к вводу-выводу следующего блока данных из очереди к устройству.

3.3. Система управления данными.
3.3.1. Файловая подсистема UNIX.
3.3.1.1. Базовая ФС System.
3.3.1.1.1. Суперблок.
3.3.1.1.2. Индексные дескрипторы.
3.3.1.1.3. имена файлов
3.3.1.1.4. Недостатки и ограничения.
3.3.2. ФС BSD UNIX.