Понятие процесса. Процесс как единица декомпозиции системы. Диаграмма состояний процесса. Контекст процесса. Процессы и потоки. Планирование и диспетчеризация. Классификация алгоритмов планирования. Алгоритм адаптивного планирования. Процессы в Unix: состояния, иерархия.

Виртуальная память: распределение страницами по запросам, свойство локальности, анализ страничного поведения процессов, рабочее множество. Схемы адресации. Достоинства и недостатки. Управление памятью.

Система прерываний: типы прерываний и их особенности, прерывания в последовательности ввода-вывода - обслуживание запроса процесса на ввод-вывод. Контроллер прерывания - задачи, структура. Прерывания в Linux: нижние и верхние половины, тасклеты, softirq. Обслуживание прерываний в ОС Windows и DPC.

Взаимодействие процессов. Сообщения. Три состояния процесса при передаче сообщений.

Взаимоисключение, монопольный доступ. Организация монопольного доступа с помощью команды test-and-set. Алгоритмы Деккера, Лампорта (“булочная”). Аппаратное взаимоисключение. Семафоры, мьютексы. Задача “обедающие философы”. Задача “производство-потребление”.

Мониторы: простые, с кольцевым буфером, Хоара (“читатели-писатели”).

Тупики - определение, условия возникновения тупиков, методы обхода тупиков. Алгоритмы обхода: алгоритм Дейкстры (банкира). Обнаружение тупиков для повторно используемых ресурсов методом редукции графа и методы восстановления работоспособности системы

ОС - определение, место ОС в системе ПО ЭВМ. Классификация ОС и их особенности. Ресурсы пользовательской системы. Режимы ядра и задачи - переключение в режим ядра - классификация событий.

Ядро системы - определение. Классификация ядер ОС. ОС с монолитным ядром. ОС с микроядром: реализация взаимодействия процессов по модели клиент-сервер. Достоинства и недостатки микроядерной архитектуры.

Файловые системы. Иерархическая структура файловой системы в Unix, задачи уровней. Открытые файлы - структуры описывающие файл. Особенности использования open() и fopen().

ФС Unix, vfs/vnode, inode, адресация больших файлов. Примеры спец. файловых систем. Описания структур dentry и superblock.

Файловая система proc. Функции для передачи данных (между процессом и ядром).

Процессы Unix: демоны, примеры системных демонов, правила создания демонов

Управление устройствами: физические принципы управления устройствами. Буферизация ввода-вывода - управление буферами. Буферный пул, кеширование.

Подсистема ввода-вывода (управления устройствами): задача, место в ОС. Основные функции ОС, связанные с управлением устройствами. Программные принципы управления. Драйверы. Точки входа в драйверы. Устройства в Unix. Модули ядра в Linux: особенности, пример.

Unix: команды - fork(); wait(); exec(); pipe(); signal().

Сокеты - определение, типы, семейства, виды сокетов. DGRAM сокеты. Примеры(программирование). Сетевые сокеты (с мультиплексированием и без), сетевой стек, установление соединения, порядок байтов. Сетевой стек (порядок вызова функций - схема), преобразование байтов (little-big endian), мультиплексирование.

Пять моделей ввода-вывода в Linux: диаграммы, особенности. Модель ввода-вывода с мультиплексированием.

Адресация прерываний в защищённом режиме

Что-то там про QNX.

Понятие процесса. Процесс как единица декомпозиции системы. Диаграмма состояний процесса. Контекст процесса. Процессы и потоки. Планирование и диспетчеризация. Классификация алгоритмов планирования. Алгоритм адаптивного планирования. Процессы в Unix: состояния, иерархия.

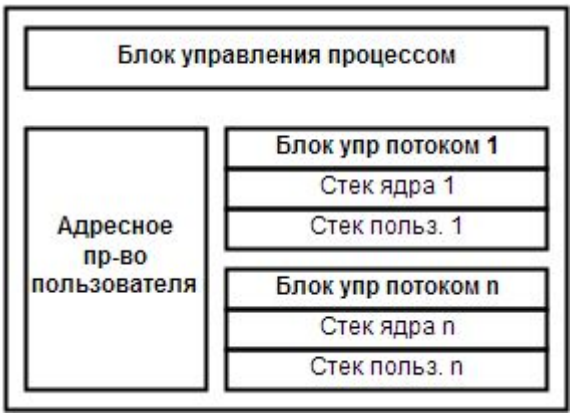
Процесс - программа в стадии выполнения. Является единицей декомпозиции системы. Является потребителем системных ресурсов.

Каждый процесс в системе обычно имеет свою строку в некой *таблице процессов*, где хранится информация о нем. Таблица одна на систему.

В течение своей жизни процесс может принимать различные состояния, переходить из одного в другое:

При *порождении* процессу выделяется строка в таблице процессов. После получения необходимых ресурсов процесс переходит в состояние *готовности*. Далее очевидно.

В каждый момент времени процесс обладает *контекстом*.



Полный контекст - аппаратный контекст + сведения о выделенных процессу ресурсах. *Аппаратный контекст* - значения всех регистров процессора.

Переключать полный контекст при переключении процесса затратно. Поэтому часто процессы делятся на *потоки* - непрерывные части кода этого процесса, которые могут выполняться (квази)параллельно с другими потоками. У каждого потока есть свой стек ядра, стек пользователя, аппаратный контекст (со счетчиком команд), но нет своего адресного пространства (выполняются в пространстве их процесса). Потоки бывают двух видов: потоки ядра (легковесные процессы), пользовательские потоки (нити) (обычно через библиотеку). Легковесные п. можно выделить в отдельный вид.

Такой подход дает выигрыш при переключении между потоками одного процесса (не надо переключать полный контекст, только аппаратный). При этом сохранение аппаратного контекста идет в стеке ядра, так как пользовательский динамический.

В многопроцессных системах постоянно происходит *планирование* - управление распределением процессорного времени между процессами. *Диспетчеризация* - поочередное выделение процессам процессорного времени планировщиком. Время выделяется *квантами* - за раз выделяется фиксированное количество. Планирование бывает с/без приоритетов, с/без вытеснения, с/без переключения. Приоритеты могут быть статическими и динамическими, абсолютными и относительными.

В интерактивных ОС (разделения времени) используются следующие алгоритмы планирования:

1. *Round Robin (RR)*: очередь процессов. Когда у текущего процесса истекает квант, он опять помещается в конец очереди.
2. *Адаптивное планирование (многоуровневые очереди):* несколько очередей с приоритетами. Новые и “проснувшиеся” процессы попадают в первую очередь. Если они не успевают завершиться или заблокироваться на вводе/выводе, то попадают в следующую очередь и т.д. Последняя очередь работает по схеме RR.
3. *Адаптивное планирование (по памяти):* дополнительно оценивается объем занимаемой в данный момент процессом физической памяти и необходимый объем для дальнейшего выполнения. В случае, если процесс может получить этот объем. то он выполняется. Если нет - откладывается.

В ОС *nix процессы создаются при помощи системного вызова fork(). При этом любой процесс может создать сколько угодно процессов, для которых он будет являться *предком,* а они для него - *потомками*. Таким образом получается иерархия из отношений “предок-потомок”. При этом потомок наследует адресное пространство и код предка, открытые им файловые дескрипторы и его сигнальную маску. Все потомки одного отдельно взятого процесса объединяются в *группу*. Нужно чтобы отсылать сигналы процессам во всей группе сразу.

Процессы в *nix описываются структурой *proc*, где содержится информация о процессе, его файловых дескрипторах и т.д. Иерархия реализована в виде связного спика: в *proc* присутствует указатель на последнего потомка, ID/указатель на предка и т.д.

В *nix, как правило, имеется один процесс с ИД = 1, являющийся предком любого другого процесса. Иногда называется *терминальным* (так как создает/открывает виртуальные терминалы tty*) или *init-процессом* (по названию программы init). В случае если предок какого-либо процесса завершится раньше чем он сам (т.е. процесс *осиротеет*), для сохранения иерархии этот процесс будет “усыновлен” терминальным, путем изменения соответствующих указателей в *proc* у нового предка и усыновляемого потомка.

В случае если потомок завершится раньше того, как предок вызовет *wait(в любом случае процесс проходит через состояние зомби)*, то он переведется в состояние *зомби* и у него будут отобраны все ресурсы, кроме его дескриптора в таблице процессов. Код завершения будет сохранен в *proc* предка и потомок продолжит существовать, пока предок не вызовет *wait() (см. ниже).*

Процесс в *nix (Linux) может находиться в следующих основных состояниях (смотреть можно в ps -aux):

R (running) - выполняется, S (sleep) - прерываемый сон (ожидает события), D - непрерываемый сон (блокирован на вв/выв), Z - зомби, T - приостановлен, X - уничтожен (не должно быть видно в выводе ps).

Виртуальная память: распределение страницами по запросам, свойство локальности, анализ страничного поведения процессов, рабочее множество. Схемы адресации. Достоинства и недостатки. Управление памятью.

Управление памятью - одна из основных функций ОС. Туда входит удовлетворение запросов на выделение и освобождение памяти. Обычно этим занимается т.н. *менеджер памяти*.

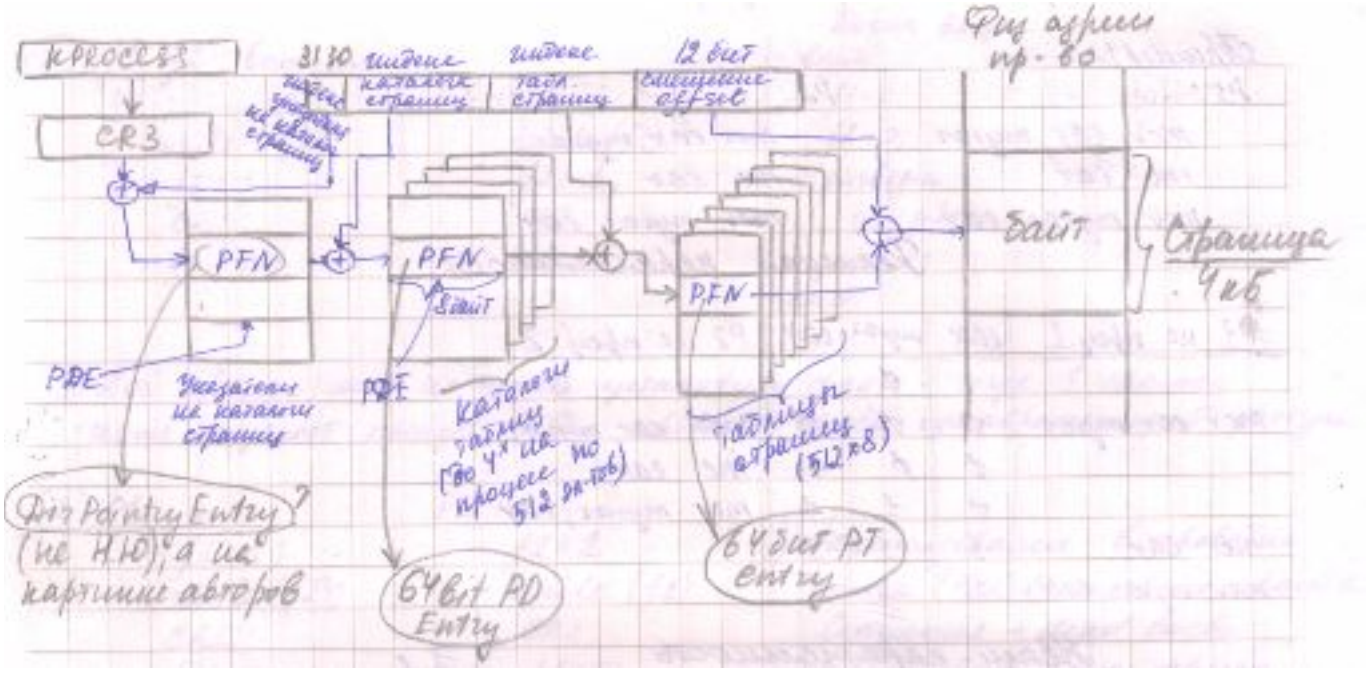
Для выделения памяти требуется информация об уже выделенных (занятых) областях и свободных областях. Обычно представляется в виде двух соотв. таблиц (таблица свободных областей (ТСО) и таблица выделенных областей (ТВО)). Так как освобождение и выделение происходит блоками различных размеров, неизбежна *фрагментация* памяти. Для уменьшения фрагментации часто при освобождении области памяти ее объединяют с соседними свободными участками, если это возможно.

Стратегии выделения памяти: а) первый блок из ТСО, подходящий по размеру; б) самый узкий из подходящих; в) самый широкий из подходящих (после выделения останется незанятый кусок большего размера).

Удобно, если каждая программа “считает”, что ее адресное пространство начинается с нуля и непрерывно. Поэтому вводим *логическую адресацию*. Можно это сделать двумя способами: 1) связное распределение: путем выделения какого-либо непрерывного блока по физическому адресу N и преобразованию логических адресов при помощи прибавления этого смещения (N); 2) несвязное с помощью страниц или сегментов.
Отсюда идея *виртуальной памяти*. Каждый (логический) ее адрес транслируется в физический с помощью какого-то отображения. Также вполне можно сделать частью ВП область на диске (своп).

Есть три схемы управления ВП:

Страницами по запросам:



Виртуальный адрес состоит из номера страницы и смещения в ней. Адреса страниц хранятся в таблице страниц (вида номер-физический адрес). В x86 для таблицы есть регистр CR3. При этом на каждый процесс одна таблица. Отсюда недостаток - большое количество больших таблиц в памяти.

Запрос - *страничное прерывание*, которое возникает при попытке доступа к незагруженной в память странице. При его обработке происходит *пэйджинг* - загрузка с диска новых и замещение старых страниц.

В этом случае есть три варианта отображения из виртуальной в физическую память:

а) прямое: физический адрес = таблица_страниц[номер_страницы] + смещение;

б) ассоциативное: предполагает наличие ассоциативной памяти (как ассоциативный массив). Ключ - виртуальный адрес. Ассоциативная память дорогая и требует поиска ключа, но при этом меньше размер самой таблицы, т.к. есть только нужные адреса;
в) ассоциативно-прямое: есть ассоциативный кэш, в котором несколько наиболее часто используемых страниц; при обращении по адресу сначала идет проверка на наличие его в кэше и если есть - оттуда берется физический, иначе преобразование по схеме а). В современных x86 есть специально для этого TLB-кэш, основанный на алгоритме LRU.

Но есть проблема: при смене процесса актуальность кэша падает в 0 (имеем дело с другими страницами), а кэш всего один. Для этого существует схема с *гиперстраницами*: каталогами таблиц страниц. На систему имеем один каталог таблиц страниц. Тогда виртуальный адрес состоит из номера таблицы в каталоге, номера страницы в ней и смещения. Кэш в таком случае не теряет актуальность при переключении.

Физическая память также делится на страницы, но они могут не совпадать с виртуальными. Для решения проблемы копирования адресных пространств (см. вопрос про fork()) в данной схеме используется флаг “copy-on-write”.

Сегментами по запросам:

Сегмент - логическое деление памяти. Имеется таблица сегментов, виртуальный адрес есть номер сегмента и смещение в нем. В таблице хранятся как минимум размер и оффсет сегмента - при формировании физического адреса можно проверять, выходит ли оффсет за границы сегмента.

В этом случае три варианта организации сегментации:

а) одна глобальная таблица - глобальные имена сегментов;
б) локальные таблицы на каждый процесс - сугубо локальные имена сегментов;
в) локальные + глобальная: локальные таблицы содержат ссылки на элементы глобальной.
Для реализации этого в x86 имеются регистры LDTR (локальной таблицы дескрипторов сегментов) и GDTR (глобальной таблицы).

Сегментами, поделенными на страницы, по запросам:

Предлагается в качестве решения фрагментации и того, что для устранения фрагментации требуется перемещать сегменты, меняя таблицы, что затратно. Размер сегмента кратен размеру страницы. Для каждого сегмента присутствует таблица страниц. В таблице сегментов адреса таблиц страниц. Итого виртуальный адрес состоит из номера сегмента, номера страницы в этом сегменте и смещения.

Обычно используется в совокупности с *алгоритмом отложенного связывания*:
если сегмент не в таблице сегментов, вызывается прерывание связывания, при обр. которого дескриптор сегмента добавляется в таблицу; если у него нет таблицы страниц, вызывается прерывание сегментирования,, при обр. которого создается эта таблица; если в этой таблице нет данной страницы, вызывается страничное прерывание, где грузится эта страница.

Эффективность этой схемы падает до нуля при использовании “copy-on-write”.

Система прерываний: типы прерываний и их особенности, прерывания в последовательности ввода-вывода - обслуживание запроса процесса на ввод-вывод. Контроллер прерывания - задачи, структура. Прерывания в Linux: нижние и верхние половины, тасклеты, softirq. Обслуживание прерываний в ОС Windows и DPC.

Прерывания.

Типы прерываний и их особенности

1. Системные вызовы
2. Исключения (устранимые/неустранимые). Например, страничное прерывание–устранимое исключение. 1 и 2 – синхронные по отношению к выполняемому процессу
3. Аппаратные прерывания:
    a. Системный таймер (выполняет важнейшую функцию – декремент кванта процессорного времени).
    b. Прерывания от действий оператора
    c. Аппаратные прерывания, которые генерируют устройства ввода/вывода по завершении операций ввода/вывода

Аппаратные прерывания являются асинхронными. Аппаратные прерывания возникают по завершении операций ввода/вывода, т.е. с помощью них процессор информируется о завершении ввода/вывода.

Точное прерывание – прерывание, оставляющее машину в строго определённом состоянии.

Свойства точного прерывания:

1. Счётчик команд(InstructionPointer - IP) указывает на команду, до которой все команды успешно выполнены.
2. Ни одна команда, после той, на которую указывает счётчик команд, не выполнена.
3. Состояние команды, на которую указывает счётчик команд, известно.

В определении точного прерывания ничего не говорится о том, что команды, после той, на которую указывает счётчик команд не начали выполняться.

Фактически, свойства точного прерывания требуют, чтобы все изменения, связанные с возможным выполнением команд, после той, на которую указывает счётчик команд должны быть отменены. Прерывания, которые не удовлетворяют перечисленным условиям называются неточными. Машины, с неточными прерываниями, обычно выгружают в стек огромное количество да нных, чтобы ОС смогла определить, что происходило (откат) в момент прерывания. Это требует затрат. Не все прерывания должны выполняться как точные. (Например, исключения в программе всё равно приведут к завершению всей программы, значит они могут быть неточными). Прерывания в последовательности ввода-вывода

обслуживание запроса процесса на ввод-вывод

Управление вводом-выводом.

1. Опрос. Polling (упорядоченный опрос).
2. Прерывания.
3. Прямой доступ к памяти (direct memory access, dma).

Программируемый ввода-вывод очень неэффективно тратит время работы процессора. В состав системы для распараллеливания введены специальные устройства — контроллеры, которые должны управлять работой внешних устройств. Когда устройство завершает ввод-вывод, устройство посылает на контроллер прерывания сигнал. Прерывание позволяет процессору отключиться от постоянного опроса готовности устройства. Получив по линии IRQ сигнал прерывания от устройства (сигнал означает готовность устройства к вводу-выводу), если это линия IRQ не замаскирована, контроллер прерывания посылает процессору сигнал int.

В конце цикла выполнения каждой команды процессор проверяет наличие сигнала прерывания. (см. последние лекции прошлого семестра). Если сигнал прерывания пришел, то процессор посылает ответный сигнал, получив который, контроллер выставляет на шину данных вектор прерывания (перевод компьютер в защищенный режим — лабораторная работа). Вектор прерывания (прерывание аппаратное) используется процессором как смещение в таблице дескрипторов прерываний. Вычислив адрес прерывания, процессор переходит на выполнение обработчика прерывания. В системе будет только таблиц дескрипторов прерываний, сколько процессоров. (Почему? Потому что каждый процессор должен иметь возможность обрабатывать прерывания, возникающие в процессе выполнения программ. Прерывания: системные вызовы (программные прерывания), аппаратные прерывания, исключения).

Контроллер не адресуется, не имеет связи с шиной адреса. Является неотъемлемой частью системы. Мы передаем вектор прерывания и получаем маску прерывания. Для этого есть регистр масок прерывания; получив соответствующую маску, система получает запрет на какие - то аппаратные прерывания. Есть немаскируемые прерывания, которые должны происходить в системе постоянно. Внутри контроллера есть регистры, шифратор приоритетов, поскольку системы реализуют так называемые вложенные прерывания (прошлый семестр, последние лекции). В наших системах прерывания имеют разные уровни приоритета. Если выполняемая программа была прервана в результате поступления прерывания и процессор перешел на выполнение обработчика этого прерывания, но пришло более высокоприоритетное прерывание, процессор должен переключиться на его обработчик. Информация о первом прерывании должна быть сохранена, для этого должен быть задействован стек ядра. Есть системы в которых для реализации вложенных прерываний используется отдельный стек – стек прерываний, другие системы используют стек ядра. (У каждого процесса есть стек ядра и стек пользователя.) Устройство поддерживает возможность установки прерываний различных приоритетов. Линии запросов прерываний не связаны с шинами, они заводятся непосредственно на ножке контроллера. Сигнал int – сигнал прерывания, который формирует контроллер, inta получает контроллер от процессора и в соответствии с ним устанавливает на шине данных прерывание.

Реализация механизма прерывания. Процессор проверяет наличие сигнала прерывания на своем отдельном входе после выполнения каждой команды. С точки зрения контроллера процесс выглядит следующим образом: сначала контроллер получает от процессора команду read и переходит к считыванию данных из связанного с ним периферийного устройства, как только данные поступают в регистр контроллера, контроллер посылает процессору сигнал int и ждет, когда процессор запросит эти данные. При получении запроса контроллер передает данные по шине данных и переходит в состояние готовности для новых операций ввода - вывода. С точки зрения процессора передача входных данных выглядит следующим образом: процессор выставляет на шину данных команду read, сохраняет содержимое счетчика команд и других регистров процессора, содержание которых соответствует выполняемой программе (сохраняет аппаратный контекст программы) и переходит на выполнение действий, не связанных с выполнением программы. В конце каждого цикла выполнения команды процессор проверяет наличие сигнала прерывания. При поступлении прерывания от контроллера (устройство посылает сигнал на контроллер прерываний, если прерывание не заблокировано, контроллер формирует прерывание) процессор сохраняет информацию о выполняющейся в данной момент программе (сохраняет аппаратный контекст) и переходит на выполнение обработчика прерывания.

?(Схема управления вводом - выводом). Задачей обработчика прерывания является инициализация работы драйвера. Под управлением драйвера получается из регистра данных контроллера внешнего устройства. Эта информация по шине данных будет направлена в соответствующую программу через соответствующую буферизацию. ISR – interrupt service routine. Информация о соответствующей подпрограмме прерывания передает изрегистра в контроллер по шине данных. Остается открытым вопрос, каким образом процессор узнает адрес устройства, завершившего ввод-вывод. Прерывание может вместе с вектором передавать адрес устройства, либо этот адрес может быть прописан в соответствующем драйвере в соответствующих системных таблицах.

Вывод: вся информация передается через регистры процессора, то есть в отличие от программируемого ввода-вывода, прерывания позволяют процессору отвлечься от проверки готовности устройства, но процессор сильно задействован, поскольку вся информация передается через его регистры.

Прерывания в лине:

Каждое устройство имеет один драйвер и регистрирует один обработчик прерывания. Драйверы могут регистрировать один обработчик прерываний и разрешать определенную линию прерываний, связанную с этим обработчиком.

Листинг 3.4 — listing

```
int request_irq(unsigned int irq, irqreturn_t (*handler)(int, void *,
    struct pr_regs *), unsigned long irqflags, const char *devname, void
    *dev_id)
```

Обработчик прерывания регистрируется регистрируется функцией 3.4. *irq* – определяет номер прерывания. Для некоторых устройств (legacy device: таких как системный таймер или клавиатура) эта величина обычно устанавливается аппаратно. Для большинства других устройств она определяется программно, и динамически (в соответствии с требованиями системы). *handler* – указатель на код обработчика прерывания который обслуживает данное прерывание.

Листинг 3.5 — listing

```
static irqreturn_t intr_handler(int irq, void *dev_id, struct pt_regs
    *regs);
```

Функция 3.5 вызывается когда происходит прерывание. Принимает 3 параметра:

`int irq` - числовое значение линии прерывания.

`dev_id` – общий указатель на `dev_id` который указывается в `request_irq`. Если значение это уникальное в системе, то оно выполняет роль дифференциатора между множеством устройств, которые потенциально могут использовать один обработчик прерывания. Может также указывать на структуру, используемую обработчиком прерывания, т.к. структура `device` уникальная для каждого устройства и используется в обработчике то она обычно передается для `dev_id`.

`regs` – содержит указатель на структу, содержащий регистры процессора, котрые определяют состояние до обслуживания прерывания. Этот параметр используется для отладки.

Возвращаемое значение имеет тип `irqreturn_t` это м.б. два значения:

`IRQ_NONE` если обработчик прерывания обнаруживает что устройство, за которым он озакреплено его не инициировало.

`IRQ_HANDLER` – когда обработчик прерывания был корректно вызван, т.е. устройства, за которым этот обработчик закреплен, действительно сформировало прерывание.

`irqflags` – м.б. 0 или битовой маской одного или нескольких флагов. Определены след. флаги:

`SA_INTERRUPT` – определяет обработчик прерывания, который является fast interrupt, т.е. быстрым прерыванием. Исторчиески linux различает быстрые и медленные обработчики прерываний. Быстрые должны выполняться быстро, но очень часто флаг устанавливается для выполнения прерывания, чтобы разрешить ему выполняться так быстро, как он может. В настоящее время есть только одно отличие: быстрые прерывания выполняеются при запрете всех прерываний на локальном процессоре, что немаловажно для быстрого завершения прерывания. Если флаг сброшен, то это не быстрое прерывание и все прерывания разрешены за исключением замаскированных на всех прцоессорах. Прерывание от системного таймера - это быстрое прерывание.

`SA_SAMPLE_RANDOM` – определяет, что прерываание, генерируемые устрйоством, должны вносить свой вклад в пулл ентропии. Этот пулл в ядре генерирует случайные числа. Большинство устройств генерируют прерывание в случайные промежутки времени (исключяя системный таймер и сетевые устройства, подверженные внешним атакам)

Про нижние и верхние половины

Некоторые прерывания (не быстрые) имеют довольно крупные обработчики

Так как во время работы обработчика другие прерывания заблокированы (а плохо когда это долго длится), обычно у обработчиков и в винде, и в лине есть две части: верхняя половина - это собственно сам обработчик, его задача состоит в том, чтобы принять данные от устройства и зарегистрировать в качестве отложенного действия вторую, более тяжелую, нижнюю половину, которая уже все обработает

Обычно во время нижней половины аппаратные прерывания разрешены и она выполняется почти как обычный процесс

В линуксе для этих отложенных действий есть два основных средства: softirq и тасклеты

На самом деле три, там еще очереди работ

Тасклет - обработчик отложенного дейсвтия. Обычная ошибка EBUSY означает, что данная линия прерывания уже используется или не указали irqfshared.
Ascii строка - текст, обозначающий соответствующее устройство. Этот текст - стринговая константа имени используется в */proc/irq*, а также в */proc/interrapt* для указания пользователя.
параметр №5) *devid* - используется для разделения линий irq. Важно понимать, что когда обработчик прерывания освобождается, именно поле *devid* обеспечивает возможность указания уникального нашего файла, чтобы удалить только нужный обработчик прерывания с линии прерывания. `freeirq(void *devid, int линии)`
`devid` - обработчик.

3 механизма реализации отложенного действия необходимы для завершения обработки прерывания.

а) softirq (гибкие);
б) тасклеты;
в) очереди работ (work queue).

3.3.1 Soft irq

определяются статически во время компиляции ядра. Описываются структурой SOFT_IRQ_ACTION. Эта структура представляет один вход.

```
// тут объявлен статический массив
// static struct softirq_action softirq_vec[32]
// те.. таких обработчиков не может быть больше 32
#include <kernel/softirq.c>

struct softirq_action {
    // выполняемая функция
    void (*action) (struct *softirq_action *);

    // данные для функции
    void *data;
}
```

Таблица 3.1 — Обработчики Softirq

Инндекс	приоритет	комментарий
HI_SOFTIRQ	0	высокоприоритетный тасклет
TIMER_SOFTIRQ	1	таймеры
NET_TX_SOFTIRQ	2	отправка сетевых пакетов
NET_RX_SOFTIRQ	3	прием сетевых пакетов
SCSI_SOFTIRQ	4	блочные устройства подситемы SCSI
TASKLET_SOFTIRQ	5	тасклеты с обычным приоритетом

Чтобы заработал обработчик SOFTIRQ его нужно зарегистрировать. Регистрация происходит с помощью функции OPEN_SOFTIRQ3 парамета):

а) индекс (один из таблицы)
б) функция обработчик
в) значения поля дата

В системе имеется демон $KSOFTIRQD$ предназначенный для выполнения отложенных действий. При этом обработка отложенных действий типа $SOFTIRQ$ и тасклетов осуществляется с помощью набора потоков ядра по одному на каждый процессор.

SOFTIRQ не могут переходить в состояние sleep. Если SOFTIRQ выполняется на процессоре, то его может вытеснить только аппаратное прерывание. Одно и тоже IRQ может параллельно выполняться на разных процессорах. Код должен быть реентабельным, следовательно код меньше простаивает в очереди.

Тасклеты

Тасклеты реализованы на основе softirq. Для них есть собственная структура. Тасклеты представлены двумя типами отложенных прерываний: HI_SOFTRQ и TASKLET_SOFTRIQ, Единственная разница между ними заключается в том, что тасклеты типа HI выполняются всегда раньше, чем тасклеты

типа TASKLET. В системе представляются структурой(tasklet_struct). Каждый экземпляр такой структуры представляет собой уникальный тасклет. Структура определена в linux/interrupt.h. Планирование тасклетов на выполнение. Запланированные (scheduled) на выполнение тасклеты хранятся в двух структурах, определенных для каждого процессора. В структуре tasklet_hi_ve высокоприоритетные тасклеты, в tasklet_ve– обычные тасклеты. Тасклеты могут быть запланированы на выполнение с помощью соответствующих функций: tasklet_hi_scheduled, tasklet_scheduled.

Отложенные прерывания и тасклеты (сравнение).

Отложенные прерывания используются для запуска самых важных и критичных по времени выполнения нижних половин (bottomhalf). В настоящий момент только в двух подсистемах, а именно в сетевой и подсистемы блочных устройств, напрямую используются softirq (отложенные прерывания). Кроме того, на основе softirq построены таймера ядра и сами тасклеты. Тасклеты имеют более простой интерфейс и упрощенные правила блокировок Это связано с тем, что два тасклета одного типа не могут выполняться одновременно на разных процессорах, но могут выполняться параллельно тасклеты разных типов. Тасклеты создаются динамически, в отличие от софтиэрку. Тасклеты являются хорошим компромиссом между производительностью и простотой использования. Тем не менее для задач, критичных ко времени выполнения и способных обеспечить эффективные блокировки, лучше (правильнее) использовать отложенные прерывания. При этом необходимо помнить, что софтиэрку выполняется при разрешенных прерываниях и не может переходить в состояние sleep. Другое отложенное прерывание не может прерывать выполнение отложенного прерывания. Отложенное прерывание может прервать только аппаратное прерывание. Поскольку софтиэрку одного типа могут одновременно выполняться на разных процессорах, взаимоисключение должно быть реализовано корректно и эффективно. Коротко можно определить тасклет следующим образом. Тасклет - это, по сути, отложенное прерывания, для которого обработчик не может выполняться параллельно на не скольких процессорах.

Про DPC:

Завершая передачу данных, контроллер устройства генерирует прерывание. Это прерывание в x86 приходит на выделенную ножку контроллера прерываний. После того, как прерывание получено, начинает действовать ядро, а именно диспетчер ввода/вывода и драйвер устройств. В результате процессор передает управление (в windows обработчик ловушки ядра или ISR) в соответствии с таблицей диспетчеризации прерываний. ISR должна выполнять минимум действий, связано это с тем, что часть действий – критические, и должны выполняться, как неделимые. Основное действие – сохранение информации о состоянии прерывания. Второе действие – инициализация отложенного действия. ДПС процедура начинает обработки следующего запроса из очереди устройства IRP (пакет), после чего заканчивает обработку прерывания. В системе столько очередей DPC, сколько процессоров. DPC – объект. По умолчанию ядро помечает DPC объекты в конец DPC очереди того процессора, на котором выполнялось ISR. Однако драйвер устройства может указать приоритет DPC низкий, средний, высокий, а также может направить DPC в очередь конкретного процессора. При каждом тике системного таймера генерируется прерывание IRQL clock (приоритет его выше всех девайсов Обработчик прерывания таймера обновляет системное время и уменьшает квант. Когда кванта обнуляется, ядру может понадобиться перераспределить процессорное время. Обработчик прерывания таймера ставит DPC в очередь для того, чтобы инициализировать диспетчеризацию потоков, после чего завершает свою работу и IRQL процессора понижается.

Взаимодействие процессов. Сообщения. Три состояния процесса при передаче сообщений.

Средства взаимодействия процессов: программные каналы (именованные и неименованные), сигналы, разделяемая память, очереди сообщений, семафоры, ввод/вывод с отображением в память

Программный канал – это специальный буфер, который создается в системной области памяти. Они описываются в соответствующей системной таблице.

3 состояния процесса при передаче сообщения(протокол обмена):

- запрос: клиент запрашивает сервер для обработки запроса
- ответ: сервер возвращает результат операции
- подтверждение: клиент подтверждает прием пакета от сервера

Для обеспечения надежности обмена в протокол обмена могут входить следующие действия:

- сервер доступен? (запрос клиента)
- сервер доступен (ответ сервера)
- перезвоните (ответ сервера о недоступности)

Разделяемые сегменты – средство взаимодействия процессов через разделяемое адресное пространство. Т.к. адресное пространство защищено, процессы могут взаимодействовать только через ядро. Подключается к адресному пространству процесса (виртуальный).
+ быстрота передачи информации. Осуществляется мэппинг, но не копирование => повыш. быстродействие.

Разделяемая память не имеет средств взаимоисключения. В ядре создается таблица разделяемых сегментов.

int shmget(key_t key, size_t size, int shmflg);

Возвращает идентификатор общего сегмента памяти, связанного с ключом, значение которого задано аргументом key. Если сегмента, связанного с таким ключом, нет и в параметре shmflg имеется значение IPC_CREATE или значение ключа задано IPC_PRIVATE, создается новый сегмент. Значение ключа IPC_PRIVATE гарантирует уникальность идентификации нового сегмента.

Присоединяет разделяемый сегмент памяти, определяемый идентификатором shmid к адресному пространству процесса. Если значение аргумента shmaddr равно нулю, то сегмент присоединяется по виртуальному адресу, выбираемому системой.

Если значение аргумента shmaddr ненулевое, то оно задает виртуальный адрес, по которому сегмент присоединяется. Если в параметре shmflg указано SHM_RDONLY, то присоединенный сегмент будет доступен только для чтения.

Для поддержки очереди сообщений в адресном пространстве ядра создается система сообщений.

Шаблон сообщения описывает Struct msgbuf {long mytype; char mytext[MSG_MAX];}

Ядро только выполняет размещение и выборку сообщений. Менеджер ресурсов отслеживает число очередей. При посылке сообщения производится копирование текста сообщение в адресное пространство ядра системы. При получении – обратная операция. Недостаток: двойное копирование. Когда процесс передает сообщение в очередь, ядро создает для него новую запись и помещает его в связный список указанной очереди. Процесс, отправивший сообщение, может завершиться. Когда какой - либо процесс выбирает сообщение из очереди, он может выбрать:

1. самое старое сообщение независимо от его типа
2. по указанному id. Если существует несколько сообщений, то берется самое старое.
3. взять сообщение, числовое значение типа которого является наименьшим из меньших или равным значению типа, указанного процессом. Т.о. ни отправитель, ни получатель не будут заблокированы. Msgget, msgctl, msgsnd, msgrev

Взаимоисключение, монопольный доступ. Организация монопольного доступа с помощью команды test-and-set. Алгоритмы Деккера, Лампорта (“булочная”). Аппаратное взаимоисключение. Семафоры, мьютексы. Задача “обедающие философы”. Задача “производство-потребление”.

Монопольное использование – если процесс получил доступ к разделяемому ресурсу, то др. процесс не м. получить доступ. Необходимо обеспечить монопольный доступ процесса к разделяемому ресурсу до тех пор, пока процесс его не освободит.

Монопольный доступ осуществляется взаимоисключением, т.е. процесс, получивший доступ к разделяемой переменной, исключает доступ к ней др. процессов.

Аппаратная реализация взаимоисключения (test-and-set).

Впервые test-and-set была введена в OS360 для IBM 370. Эта команда является машинной и неделимой, т.е. ее нельзя прервать. Она одновременно производит проверку и установку ячейки памяти, называемой ячейкой блокировки: читает значение логической переменной B, копирует его в A, а затем устанавливает для B значение «истина» (все это делается за счет одной шины данных). Она присутствует в наборе сист вызовов Win и Unix.

```
Test-and-set(a, b) : a = b; b = true;
```

В Windows это называется спин-блокировкой. /* спин-блокировкой по-рязаной, наз. проверка флага в цикле */

```
flag, c1, c2: logical;
P1: while(1)
 c1 = 1;
 while(c1 == 1)
  test_and_set(c1, flag);
 CR1;
 flag = 0;
 PR1;
end P1;
```

```
P2: while(1)
 c2 = 1;
 while(c2 == 1)
  test_and_set(c2, flag);
 CR2;
 flag = 0;
 PR2;
end P2;

flag = 0;
parbegin
P1;
P2;
parend;
```

Пусть P1 хочет войти в свой критический участок, когда P2 уже там. P1 уст в единицу и входит в цикл проверки. Поскольку P2 нах в критическом участке, то у P2 – 1. P1 будет находится в цикле активного ожидания, пока P2 не выйдет из своего критичнского участка.

Т.к. test-and-set машинная неделимая команда и выполняется очень быстро, то бесконечного откладывания не возникает. Бесконечное откладывание – ситуация, когда разделённый ресурс снова захватывается тем же процессом.

```
flag1, flag2: logical;
queue: int;

11 p1: while(1)
 flag1 = 1;
 while(flag2)
  if(queue == 2) then
  begin
   flag1 = 0;
   while(queue == 2);
   flag1 = 1;
  end;
 CR1;
 flag1 = 0;
 queue = 2;
 PR1;
end P1;
```

```
p2: while(1)
  flag2 = true;
 while(flag1)
  if(queue == 1) then
  begin
   flag2 = 0;
   while(queue == 1);
   flag2 = 1;
  end;
 CR2;
 flag2 = 0;
 queue = 1;
 PR2;
end P2;

flag1 = 0;
flag2 = 0;
parbegin
P1;
P2;
parend;
```

Программная реализация (алгоритм Деккера).

Деккер – голландский математик. Предложил способ свободный от бесконечного откладывания.

queue – очередь процесса входить в критическую секцию.

Недостаток обоих методов – активное ожидание на процессоре - ситуация, когда процесс занимает процессорное время, проверяя значение флага. Активное ожидание на процессоре является неэффективным использованием процессорного времени.

Мониторы: простые, с кольцевым буфером, Хоара (“читатели-писатели”).

Взаимодействие процессов: монопольное использование:

Процессам часто нужно взаимодействовать друг с другом, например, один процесс может передавать данные другому процессу, или несколько процессов могут обрабатывать данные из общего файла. Во всех этих случаях возникает проблема синхронизации процессов. Она связана с потерей доступа к параметрам из-за их некорректного разделения. В каждый момент времени на процессоре выполняется 1 процесс. Каждому процессу выделяется квант процесс. времени.

Разделяемый ресурс - переменная, к которой обращаются разные процессы.

Критическая секция – область кода, из которой осуществляется доступ к разделяемому ресурсу.

Монопольное использование – если процесс получил доступ к разделяемому ресурсу, то др. процесс не м. получить доступ. Необходимо обеспечить монопольный доступ процесса к разделяемому ресурсу до тех пор, пока процесс его не освободит. Все алгоритмы программной реализации обобщил Дейкстра, введя понятие семафора.

Активное ожидание на процессоре – ситуация, когда процесс занимает процессорное время, проверяя значение флага (занятости ресурса другим процессом). Активное ожидание на процессоре является неэффективным использованием процессорного времени.

Возможные варианты развития событий:

1. Возможно, что оба процесса пройдут цикл ожидания и попадут в свои критические секции -ок

2. Возможно бесконечное откладывание (зависание)– ситуация, когда разделённый ресурс снова захватывается тем же процессом.

3. Тупик (deadlock, взаимоблокировка)– ситуация, когда оба процесса установили флаги занятости и ждут. Т.е. каждый ожидает освобождения ресурса, занятого другим процессом

Монитор – языковая конструкция, состоящая из структур данных и подпрограмм,использующих данные структуры. Монитор защищает данные. Доступ к данным монитора могут получить только п/п монитора. В каждый момент времени в мониторе м. находиться только 1 процесс. Монитор является ресурсом.

Процесс, захвативший монитор,– процесс в мониторе, процесс, ожидающий в очереди,– процесс на мониторе.

Используется переменная типа «событие» для каждой причины перевода процесса в состояние блокировки. wait – откр. доступ к монитору, задержив. выполнение процесса. Оно д.б. восстановлено операцией signal др. процесса.

Если очередь перем. к типу усл. <> 0, то из очереди выбирается 1 из процессов и инициализируется его выполн..

Монитор «кольцевой буфер»– решает задачу «производство.потребление», то есть существуют процессы.производители и процессы.потребители, а также буфер – массив заданного размера, куда производители помещают данные, а потребители считывают оттуда данные в том порядке, в котором они помещались (FIFO).

```
resource: monitor circle_buffer;
const n = size;
var buffer: array [0..n-111]]] of <type>;
pos, write_pos, read_pos: 0..n-1;
full, empty: conditional;

procedure producer(p: process, data:<type>)
begin
if (pos = n) then
wait(empty);
buffer[write_pos] := data;
Inc(pos);
write_pos = (write_pos + 1) mod n;
signal(full);
end;
```

```
procedure consumer(p: process, data:<type>)
begin
if (pos = n) then
wait(full);
data:= buffer[read_pos];
Dec(pos);
read_pos = (read_pos + 1) mod n;
signal(empty);
end;
begin pos := 0; write_pos := 0; read_pos := 0; end.
```

```
#include <windows.h>
volatile LONG ActReadersCount = 0, ReadersCount = 0,
        WritersCount = 0;
volatile int Value = 0;
HANDLE CanRead, CanWrite, Writing;
void StartRead()
{       InterlockedIncrement(&ReadersCount);
        WaitForSingleObject(Writing, INFINITE);
        ReleaseMutex(Writing);
        if (WritersCount)
            WaitForSingleObject(CanRead, INFINITE);
        InterlockedDecrement(&ReadersCount);
        InterlockedIncrement(&ActReadersCount);
        if (ReadersCount)
                SetEvent(CanRead);                      }
void StopRead()
{       InterlockedDecrement(&ActReadersCount);
        if (!ReadersCount)
                SetEvent(CanWrite);                     }
void StartWrite()
{       InterlockedIncrement(&WritersCount);
        if (ActReadersCount)
            WaitForSingleObject(CanWrite, INFINITE);
        WaitForSingleObject(Writing, INFINITE);
        InterlockedDecrement(&WritersCount);            }
void StopWrite()
{       ReleaseMutex(Writing);
        if (ReadersCount)
                SetEvent(CanRead);
        else
                SetEvent(CanWrite);                     }
```

```
DWORD WINAPI Reader(PVOID pvParam)
{       for (;;)
        {       StartRead();
                Sleep(rand()/100.);
                printf("Reader %d: %d\n", (int)pvParam,
                        Value);
                StopRead();
                Sleep(rand()/10.);      }
        return 0;       }
DWORD WINAPI Writer(PVOID pvParam)
{       for (;;)
        {       StartWrite();
                Sleep(rand()/75.);
                printf("Writer %d: %d\n", (int)pvParam,
                        ++Value);
                StopWrite();
                Sleep(rand()/7.);       }
        return 0;       }
int main()
{       HANDLE Writers[3], Readers[3];
       CanRead = CreateEvent(NULL, false, false, NULL);
       CanWrite = CreateEvent(NULL, false, true, NULL);
        Writing = CreateMutex(NULL, false, NULL);
        for (int i = 0; i < 3; i++)
                Writers[i] = CreateThread(NULL, NULL,
                        Writer, (LPVOID)i, NULL, NULL);
        for (int j = 0; j < 3; j++)
                Readers[j] = CreateThread(NULL, NULL,
                        Reader, (LPVOID)j, NULL, NULL);
        getch();
        return 0;       }
```

Тупики - определение, условия возникновения тупиков, методы обхода тупиков. Алгоритмы обхода: алгоритм Дейкстры (банкира). Обнаружение тупиков для повторно используемых ресурсов методом редукции графа и методы восстановления работоспособности системы

Тупик - ситуация, возникающая в результате монопольного использования ресурсов: процесс, владея ресурсом, запрашивает другой ресурс, которой занят другим процессом, требующий ресурс , занятый первым процессом.

Условия возникновения (их четыре):

- Условие взаимоисключения - только один процесс может использовать один ресурс.
- Условие ожидания ресурсов - процесс может блочить ресурс и ждать освобождения другого ресурса
- Условие неперераспределяемости - система не может отбирать ресурсы у процесса
- Условие кругового ожидания - P1 ждёт P2, P2 ждёт P3, P3 ждёт P1

Методы борьбы (3 метода):

- Предотвращение тупиков (обход)
- Избегание тупиков
- Обнаружение тупиков

Предотвращение:

- Косвенное: не позволять трём первым условиям выполняться одновременно
    - Взаимоисключение (должно поддерживаться на уровне ОС)
    - Ожидание (процесс обязан запрашивать все необходимые ресурсы одновременно)
    - “Отбор” ресурсов системой (процесс должен освободить ресурс и запросить его снова)
- Прямое: запрет циклического ожидания

Избегание:

- Оценка текущей ситуации
- Необходимо знать будущие запросы системы

Подходы к избеганию:

- Запрет инициализации процесса, который может привести к взаимной блокировке
- Запрет предоставления ресурсов, если это может привести к взаимной блокировке
    - Алгоритм банкира

Алгоритм банкира:

Введём понятие: *Безопасное состояние* - это такое состояние, для которого имеется по крайней мере одна последовательность событий, не приводящая к взаимоблокировке.

Предположим, что у системы в наличии N устройств. ОС принимает запрос от пользовательского процесса, если его максимальная потребность не превышает N. Пользователь гарантирует, что если ОС в состоянии удовлетворить его запрос, то все устройства будут возвращены системе в течение конечного времени. Текущее состояние системы называется *надёжным,* если ОС может обеспечить всем процессам их выполнение в течение конечного времени. В соответствии с алгоритмом банкира выделение устройств возможно, только если состояние системы является надёжным

Рассмотрим пример безопасного состояния с 3 пользователями и 11 устройствами. Пусть текущая ситуация такова:

Польз.	**Требует**	**Выделено уже**
1	9	6
2	10	2
3	3	1

Последующие действия системы таковы. Вначале удовлетворить запросы 3-его пользователя, затем дождаться, когда он закончит работу и освободит свои 3 устройства. Затем можно обслужить первого и второго пользователей. Т.е. система удовлетворяет только те запросы, которые оставляют её в надёжном состоянии, и отклоняет остальные.

Ненадёжное состояние не предполагает, что обязательно возникнет тупик. Он говорит о том, что в случае неблагоприятной последовательности событий система может зайти в тупик.

Плюсы:

+ Не нужно отбирать ресурсы
+ Меньше ограничений по сравнению с предотвращением взаимной блокировки

Минусы:

− Максимальное количество необходимых ресурсов и процессов должно быть известно заранее
− Количество ресурсов должно быть конечным
− Процессы должны гарантировать, что ресурсы будут возвращены
− Процесс не может завершиться, пока он владеет ресурсом

Обнаружение

Формализуем задачу: будем рассматривать систему как декартово произведение множества состояний, где под состоянием понимается состояние ресурса (свободен или распределён). При этом состояние может измениться процессом в результате запроса и последующего получения ресурса, а также в результате освобождения процессом занимаемого им ресурса. Если в системе процесс не может ни получить, ни вернуть ресурс, то система находится в тупике, т.е. не может поменять своё состояние в результате выделения или освобождения ресурса. Определить, что какое-то количество процессов находится в тупике можно при помощи графовой модели Холдта.

Граф $L = (X, U, P)$ задан, если даны множества вершин $X \neq \varnothing$ и множество рёбер $U \neq \varnothing$, а также инцидентор (трехместный предикат) P, причём высказвание $P(x,u;y)$ означает высказывание «Ребро u соединяет вершину x с вершиной y», а также удовлетворяет двум условиям:

1) предикат P определён на всех таких упорядоченных тройках (x,u,y), для которых $x, y \in X$ и $u \in U$.
2) каждое ребро, соединяющее какую-либо упорядоченную пару вершин x и y кроме неё может соединять только обратную пару y, x.

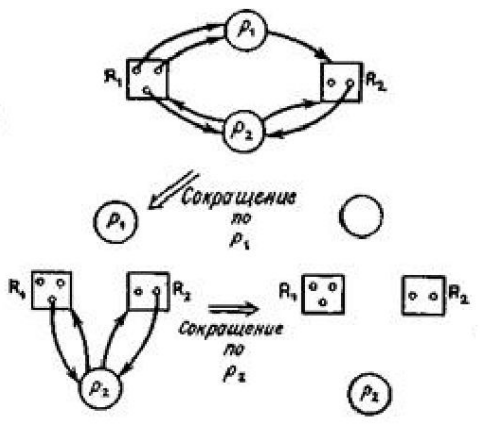
Дуга – ребро, соединяющее x с y, но не y с x.

Дуги бывают двух видов: запросы и выделения. Таким образом модель Холдта представляет собой двудольный (бихроматический) граф, где X разбивается на подмножество вершин-процессов $\pi = \{p_1, p_2, ..., p_n\}$ и подмножество вершин-ресурсов $\rho = \{r_1, r_2, ..., r_n\}$.

$\pi, \rho : X = \rho \cup \pi, \rho \cap \pi = \varnothing$.

Приобретение (выделение) – дуга (r, p), где $r \in \rho, p \in \pi$.

Запрос – дуга (p, r), где $r \in \rho, p \in \pi$.



Обнаружить процесс, попавший в тупик, можно методом редукции (сокращения) графа.

Формализуем процедуру сокращения:

1) Граф сокращается по вершине pi, если эта pi не является ни заблокированной, ни изолированной, путём удаления всех рёбер, входящих в pi и выходящих из неё.

2) Процедура сокращения соответствует действиям процессов по приобретению запрошенных ранее ресурсов и последующего освобождения всех занимаемых процессом ресурсов. В этом случае Pi становится изолированной вершиной.

● · Граф повторно используемых ресурсов несокращаем, если он не может быть сокращен ни одним процессом.

● · Граф повторно используемых ресурсов является полностью сокращаемым, если существует последовательность сокращений, которые устраняют все ребра графа.

Представление графов: две матрицы: выделенных ресурсов (где элемент ij - количество выделенных процессу j ресурсов класса i), запросов (ij - количество запрашиваемых процессом i ресурсов класса j). Их можно представить как есть в памяти, можно в виде связных списков векторов. Их можно сокращать так же как и граф. Можно ускорить сокращения, сравнивая сразу целые вектора.

В простейшем варианте двудольный (бихроматический) граф м.б. описан 2 матрицами:

1. матрица запросов (отраж. запросов проц.) – A = {p, r}
2. матрица распредел. (кол-во рес, выдел. к-л. проц.) – B = {r, p}

	запросы			распредел.		i = процесс, j = ресурс
A =			B =			Как табл, так и списки д.б. моноп. использ.
	a_{ij}			b_{ij}		

ОС - определение, место ОС в системе ПО ЭВМ. Классификация ОС и их особенности. Ресурсы пользовательской системы. Режимы ядра и задачи - переключение в режим ядра - классификация событий.

Операционная система (ОС) - набор программ, как обычных, так и микропрограмм, которые обеспечивают возможность использования аппаратуры компьютера, при этом аппаратура предоставляет сырую вычислительную мощность.

Главное назначение ОС - управление ресурсами системы, которое сводится к выделению ресурсов под процессы.

Особенности построения ОС:

- определение абстракций, зависят от разработчиков ОС;
- предоставляемые примитивные операции;
- защита, путем установки прав доступа;
- обеспечение совместного использования данных и ресурсов;
- изоляция отказов системы;
- управление аппаратурой.

Виды ОС:

- однопрограммные пакетной обработки - в ОП м.б. только 1 прикладная программа;
- мультипрограммные
    1. пакетной обработки — планирование на принципе распараллеливания функций; программа выполняется до тех пор, пока не запросят доп. ресурс системы, или не придет более приоритетный процесс; в случае поддержки вытеснения, может вытеснить;
    2. разделения времени — каждому процессу выделяется квант процессорного времени, процесс выполняется, пока квант не истек, не началась процедура ввода/вывода или не произошло вытеснение процесса другим процессом.
    3. реального времени — обеспечивает возможность обслуживать внешние по отношению к системе системы за определенный, строго заданный, промежуток времени.
- локальные и сетевые (серверные);
- однопроцессорные и многопроцессорные;
- однопользовательские и многопользовательские
- встроенные;
- ОС для смарт-карт.

Сложность проектирования ОС:

- большой объем кода;
- подсистемы ОС взаимодействуют друг с другом;
- параллелизм;
- совместное использование ресурсов системы и отдельных программ;
- высокая степень универсальности;
- хакеры;
- переносимость;
- совместимость с предыдущими версиями.

Ресурсы системы:

- процессорное время;
- объем ОП;
- внешние устройства;
- объекты ядра;
- коды ОС (реинтерабельность - свойство повторной входимости);

- данные.

3 события, переводящие систему в режим ядра:

- системные вызовы (программные прерывания) - вызов API - набор функций, которые система предоставляет пользователю в Unix; в системе всё построено единообразно;
- исключения - возникают при ошибках, 2 типа:
  - ❏ устранимые, например - страничное прерывание;
  - ❏ неустранимые.
- аппаратные прерывания - асинхронные события:
  - ❏ самое массовое прерывание - от внешних устройств;
  - ❏ прерывание от системного таймера (по тику) - прерывает всё;
  - ❏ прерывание от действия оператора.

Ядро системы - определение. Классификация ядер ОС. ОС с монолитным ядром. ОС с микроядром: реализация взаимодействия процессов по модели клиент-сервер. Достоинства и недостатки микроядерной архитектуры.

ОС разбивается на несколько уровней, причем обращение через уровень невозможно. В качестве ядра выделяется самый низкий уровень распределения аппаратных ресурсов процессам самой ОС. Существует 2 типа структур ядер: монолитное ядро и микроядро.

Монолитное ядро – программа, состоящая из подпрограмм (имеющая модульную структуру), содержащих в себе все функции ОС, включая планировщик, файловую систему, драйверы, менеджеры памяти. Поскольку это 1 программа, то она имеет 1 адресное пространство, все её подпрограммы имеют доступ ко всем её внутренним структурам. Такие ОС делятся на 2 части – резидентную и нерезидентную. Любые изменения приводят к необходимости перекомпилирования всей ОС.

Типы структур ядер:

1. Монолитное ядро

● − ядро – это все, что выполняется в режиме ядра

● − ядро представляет собой единую программу с модульной структурой (выделены функции, такие как

● планировщик, файловая система, драйверы, менеджеры памяти)

● − при изменении к.-л. функции нужно перекомпилировать все ядро

● − такие ОС делятся на две части – резидентную и нерезидентную

1) Выполнение вызова в ОС с монолитным ядром требует 2 переключения в режим ядра (приложение – ядро – приложение)

2. Микроядро

Специальный модуль нижнего уровня, который обеспечивает работу с аппаратурой на самом низком уровне и базовые функции работы с процессами. Остальные компоненты – самостоятельные процессы, которые могут работать в разных ядерных пространствах. Общение между компонентами происходит с помощью сообщений через адресное пространство микроядра. Микроядерная архитектура основана на модели клиент-сервер (например, ОС Mach, Hurd и Win2k(но не в классическом понимании))

Модель клиент-сервер

Система рассматривается как совокупность двух групп процессов

- процессы-серверы, предоставляющие набор сервисов
- процессы-клиенты, запрашивающие сервисы

Принято считать, что данная модель работает на уровне транзакций (запрос и ответ представляет неделимая операция)

2) Без учёта времени передачи самих сообщений, в ОС с микроядром системный вызов требует как минимум 4 переключения, плюс время, проводимое в блокировке при передаче сообщений (приложение – микроядро – сервер ОС – микроядро – приложение).

Достоинства и недостатки микроядерной архитектуры:

+ Высокая степень модульности ядра ОС, что существенно упрощает добавление в него новых компонентов. В микроядерной ОС можно, не прерывая ее работы, загружать и выгружать новые драйверы, файловые системы и т. д., т.о. упрощается процесс отладки компонентов ядра.

+ Компоненты ядра ОС принципиально не отличаются от пользовательских программ, поэтому для их отладки можно применять обычные средства.

+ Повышается надежность системы, поскольку ошибка на уровне непривилегированной программыменее опасна, чем отказ на уровне режима ядра.

− Микроядерная архитектура операционной системы вносит дополнительные накладные расходы,связанные с передачей сообщений, что существенно влияет на производительность.

− Для того чтобы микроядерная операционная система по скорости не уступала операционным системам на базе монолитного ядра, требуется очень аккуратно проектировать разбиение системы на компоненты, стараясь минимизировать взаимодействие между ними.

Файловые системы. Иерархическая структура файловой системы в Unix, задачи уровней. Открытые файлы - структуры описывающие файл. Особенности использования open() и fopen().

Файл— любая поименованная совокупность данных, хранящихся на диске.

Файловая система - это система хранения файлов и организации каталогов.

Иерархическая структура файловой системы в Unix: (определяется удаленностью от аппаратного обеспечения.)

Символьный уровень - именование файлов (уровень "удобства" для пользователя). Любое имя – набор символов. Полное имя файла – путь к файлу и его имя. Разные ОС накладывают разные требования на имена файлов. В Юникс имя файла не является его атрибутом. Т. н. «этикетка» к файлу - хард линк. На символьном уровне файл не имеет уникального имени.

Базовый уровень - Дескриптор файла. Структура или несколько структур, которые позволяют однозначно идент. Файл в системе. Базовый уровень обеспечивает идент-ю любого файла любой файловой системы.

Уровень проверки прав доступа.
В Юникс user, group, others. Они должны быть зафиксированы в структурах, описывающих файлы.

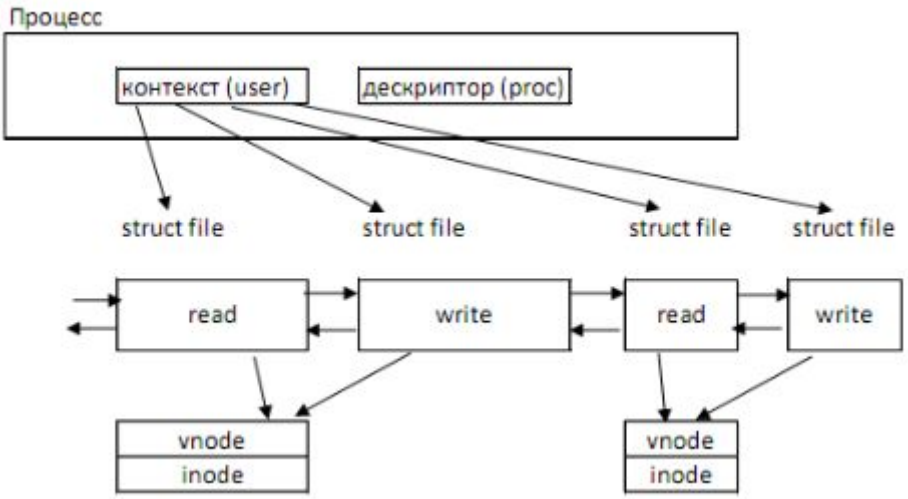
Логический уровень - позволяет преобразовать виртуальный адрес записи к физическому адресу. Делается это через подсистему ввода-вывода. Подсистема предназначена для взаимодействия определенных уровней системного ПО с низлежащими уровнями системного ПО и с аппаратной частью.

Физический уровень - физическое адресное устройство.
Диск – внешнее устройство. Работа осуществляется через систему ввода-вывода.

При каждом открытии процессом файла ядро создает в системной области памяти новую структуру типа file, которая описывает как открытый файл, так и операции, которые процесс собирается производить с файлом (например, чтение). Структура file содержит такие поля, как определение режима открытия (только для чтения, для чтения и записи и т.п.); указатель на структуру vnode; смещение в файле при операциях чтения/записи; указатель на структуру, содержащую права процесса, открывшего файл; а также указатели на предыдущую и последующую структуру типа file, связывающие все такие структуры в список.

Открытый файл

— файл, с которым процесс работает в данный момент времени. Как программа отличается от процесса, так файл отличается от открытого файла. Открытие файла — установка связи. В системе для работы с файлами создаются дополнительные структуры. Только процессы могут выполнять функции read/write. Процесс, заинтересованный в работе с файлом, открывает файл. При каждом открытии файла каким-либо процессом ядро создает в системной области памяти экземпляр структуры файл (struct file). Каждый файл имеет единственный vnode, inode и все процессы, открывшие данный файл получат указатель на одни и те же vnode inode. Struct file представляет из себя двусвязный список.



Файловый объект - представляет собой файл, открытый процессом. Создаётся системным вызовом open(), уничтожается close(); все функции работы с файлами - методы, определены в таблице файловых операций. Для одного файла может существовать несколько файловых объектов, т.к. он может быть открыт несколькими процессами. Файловый объект не соответствует никакой структуре на жёстком диске (как и dentry).

```
struct file
{
mode_t f_mode;
loff_t f_pos;
unsigned short f_flags;
unsigned short f_count;
unsigned long f_reada, f_ramax, f_raend, f_ralen, f_rawin;
struct file *f_next, *f_prev;
int f_oner;
struct inode *f_inode;
struct file_operations *f_op;
unsigned long f_version;
void *private_data;
};
```

Функции, которые система предоставляет для работы с файлом на самом деле являются методами этой структуры.

OPEN(), FOPEN()

Функция open() является системным вызовом, предназначенным для открытия файла и возвращающим файловый дескриптор. Функция fopen() возвращает указатель на структуру _IO_FILE и в своей реализации использует системный вызов open(). Основное отличие функций чтения и записи stdio.h от системных вызовов в том, что библиотечные функции при чтении или записи используют буферизацию

FILE *fopen(const char *filename, const char *mode);

Открытие файла с режиме добавления (в качестве первого символа в аргумент режима – “a”) приводит к тому, что все последующие операции записи в файл работать с current end-of-file, даже если вмешиваются вызовы FSEEK(3C). Если два независимых процесса откроют один и тот же файл для добавления данных, каждый процесс может свободно писать в файл без опасения нарушить вывод сохраненный другим процессом. Информация будет записана в файл в том порядке, в котором процессы записывали ее в файл.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname,int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
```

Вызов open() создает новый файловый дескриптор для открытого файла, запись в общесистемной таблице открытых файлов. Эта запись регистрирует смещение в файле и флаги состояния файла (модифицируемые с помощью the fcntl (2) операции F_SETFL). Дескриптор файла является ссылкой на одну из этих записей; эта ссылка не влияет, если путь впоследствии удален или изменен ссылаться на другой файл. Новый дескриптор открытого файла изначально не разделяется с любым другим процессом, но разделение может возникнуть через fork(2).

ФС Unix, vfs/vnode, inode, адресация больших файлов. Примеры спец. файловых систем. Описания структур dentry и superblock.

Структура ФС Unix.

Файловая система в Юникс композиция нескольких отдельных деревьев, каждое из которых является отдельной файловой системой.

На нижнем уровне происходит выделение физических устройств, в частности физических страниц. На верхнем уровне – управление данными. Файловая система предназначена для управления данными, которые хранятся на так называемой вторичной памяти. Сама файловая система имеет сложную иерархическую структуру. (рисунок выше, посл. лекции первого семестра)

Символьный уровень – именование файлов с точки зрения пользователя (интерфейсный).

Базовый уровень – уровень дескриптора файла (структура или несколько структур, которые позволяют идентифицировать файл и хранить информацию о нем).

Физический уровень – физическое адресное устройство.

VFS виртуальная файловая система, позволила скрыть от пользователя особенности реализации разных ФС. ВФС не ориентирована на конкретную ФС. Достигается это за счет использования всех особенностей объектноориентированного подхода в программировании.

vnode/VFS - интерфейс, позволяющий на одной машине использовать несколько ФС, которые могуть быть как локальные, так и удалённые. В ядре системы файловое представление абстрагирует vnode. VFS и vnode реализованы как абстрактный базовый класс, подклассы которого описывают разные файловые системы. Интерфейс vnode/vfs базируется на таких понятиях ООП как класс и объект.

На основе любого класса можно создать один или более порожденных классов – классовнаследников. Эти классы могут стать базовыми для порожденных от них классов, таким образом получается иерархия классов. Объект типа порожденный класс является также объектом типа базовый класс.

Файл (порожд.)> директория. Любой каталог является файлом. Атрибуты, добавленные в класснаследник не видны базовому классу. Базовый класс используется для создания основных абстракций, которые, как правило, реализованы в виде виртуальных функций. В классах наследниках определяются специфичные для каждого класса реализации таких виртуальных функций базового класса.

Организация виртуальной файловой системы (VFS).

Эта ФС охватывает файлы дисковой ФС и сетевой ФС.

Файлы VFS:
- принадлежащие дисковой или сетевой ФС
  - обычный файл
  - каталог
  - символьная ссылка
- принадлежащие специальной ФС
  - файл блочного устройства
  - файл символьного устройства
  - именованный канал (ФИФО)
  - сокет

VFS имеет 4 основных типа объектов:
1. Суперблок (superblock). Представляет определенную смонтированную ФС.
2. Файловый индекс (inode). Представляет определённый файл.
3. Элемент каталога (dentry – directory entry). Определяет путь.
4. Файл – представляет в системе открытый файл.

В ВФС информация о файлах разделена на две части: часть, которая не зависит от типа файловой системы, хранится в специальной структуре ядра vnode; часть, зависимая от ФС, в структуре inode.

Inode.

С каждым файлом связана таблица, которую называют inode. В этой таблице перечислены физические адреса и атрибуты блоков файла.

15 полей, каждое поле по 4 байта. 12 используются для непосредственной адресации.

12ое поле – простая косвенная адресация. (1024 записи)

13ое поле – двойная косвенная адресация.

14ый блок – тройная косвенная адресация.

Vnode.

```
struct vnode
    //данные
    v_count
    v_type
       v_vfsmountedhere
       v_date // закртые данные, зависящие от ФС
```

Для файлов S5FS и ufs – это структура inode.

Таким образом в Юникс системах поддерживается адресация очень больших файлов. Каждый файл имеет собственный индексный блок, который содержит адреса блоков данных. Запись в директории, относящаяся к файлу, содержит адрес индексного блока. По мере заполнения файла, соответствующие сектора диска заполняются, указатели на блоки диска в индексном узле принимают фактическое значение.

Структура strcut inode определена в файле <linux/fs.h>

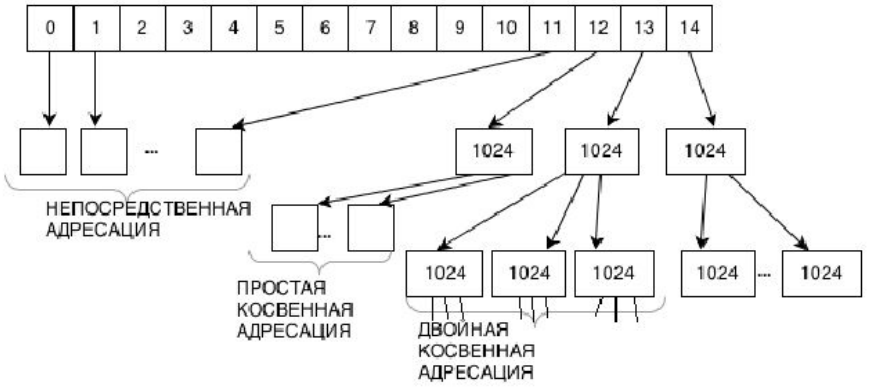
```
Struct inode
{
        Struct inode_operations *i_op; //указатель на таблицу операций с inode'ом
        …
        Union
        {
                Struct pipe_inode *i_pipe;
                Struct block_device *i_bdev;
                Strcut cdev *i_dev;
        };
```

(Такая организация позволяет создавать большие файлы. Блок-адресуемая двойная косвенная адресация единицы дискового адресного пространства. Файлам выделяются блоки. Блоки объединяются в группы (кластеры). Внутри них нумеруются последовательно. В результате начало каждой группы блоков имеет адрес: (Nгруппы блоков) -1) число блоков в гр.. Инф. хранящаяся в суперблоке, используется для доступа к остальным данным на диске)

Адресация больших файлов

РИС.30

Адресная информация файла:



Файловая система proc. Функции для передачи данных (между процессом и ядром).

ФС proc не является монтируемой файловой системой, это интерфейс ядра, который позволяет приложениям читать и изменять данные в адресном пространстве ядра, получать инфо о процессах и используемых ими ресурсах, при этом используется стандартный интерфейс системных вызовов.

- доступ к адресному пространству ядра осуществляется с помощью прав на чтение, запись, изменение.

По умолчанию чтение, запись разрешены только для владельцев процессов.

В системе каждый процесс имеет id – целое число, порядковый номер процесса по времени создания. Этот Pid является именем директории в ФС proc. (/proc/<PID>) В поддиректории каждого процесса находятся файлы и данные, относящиеся к данному процессу.

Элемент	Тип	Содержимое
cmdline	файл	указатель на директорию процесса
cwd	симолическая ссылка	-//- (путь к файлу)
environ	файл	список окружения процесса
exe	симолическая ссылка	указатель на образ процесса (путь к файлу)
fd	директория	ссылки на файлы, используемые процессом
root	soft link	указатель на корень ФС процесса
stat	файл	инфо о процессе

С помощью функции getpid процесс может узнать свой Pid и сконструировать путь к своей директории.

Использование ссылки self: /proc/self

Пример. Программа, печатающая инфо об окружении процесса.

```
#include <stdio.h>
#define BUF_SIZE 0x100
int main(int argc, char *argv[])
{
char buf[BUF_SIZE];
int len, i;
FILE *f;
f = fopen("/proc/self/environ", "r");
while ((len=fread(buf, 1, BUF_SIZE, f)>0))
{
for (i=0; i<len; i++)
{
if (buf[i]==0)
buf[i] = 10;
}
buf[len]=0;
printf("%s", buf);
}
fclose(f);
return 0;
}
```

В environ строки разделены не ‘/n’, а ‘/0’. Вся информация в ФС proc – это текстовый файл.

Также proc содержит информацию о разных частях системы.

Команды.

- lspci – информация об устройствах, подключенных к шине pci.
- apm – информация о состоянии батареи ноутбука.

Особенностью ФС proc является то, что файлы и поддиректории этой ФС могут создаваться, их легко регистрировать и дерегистривать, все это динамически. Поэтому их часто используют другие модули.

ФС proc использует структуру

```
struct proc_dir_entry
{
unsigned short low_ino; // номер индекса, inode файла
unsigned short namelen; // длина
const char *name; // указатель на имя файла
mode_t mode; // режим файла
nlink_t nkink; // количество ссылок на файл
uid_t uid; // id пользователя
//...
struct proc_dir_entry *next, *parent, *subdir;
void *data;
//...
int deleted;
};
```

Функции для работы с элементами ФС proc.

1. create_proc_entry(name,mode, parent);Создает файл с именем name, режимом mode и указателем на структуру proc_dir_entry parent, в которой появится файл. Обычно mode = 0.Получим указатель на структуру proc_dir_entry://proc/net/test// для инициализации файла test:

```
char[] name, test_entry
;test_entry=create_proc_entry("test",0600,proc_net);t
est_entry->nlink=1;
test_entry->data=(void*)&test_data;
test_entry->read_proc = test_read_proc;
test_entry->write_proc = test_write_proc;
```

2.Удаляет файл из proc. Для этого необходим указатель, относительный путь или структура proc_dir_entry.

remove_proc_entry(Name, parent);

3.Создание директории с именем name в parent, возвращает указатель на proc_dir_entry созданной поддиректории.proc_mkdir(name, parent);

4.create_proc_read_entry(); create_proc_read_entry(name,mode,base, get_info);

Создает в proc файл name и регистрирует функцию get_info – метод, который используется для чтения из файла с именем name. Для примера, приведенного выше, указанная функция может использоваться следующим образом:

test_entry= create_proc_read_entry("test", 0600, proc_net, test_get_info);

5. create_proc_info_entry(name,mode,base, read_proc,data);

Аналогично (4) и дополнительно в proc_dir_entry устанавливает параметр data

test_entry= create_proc_info_entry("test", 0600, proc_net, test_read_proc,&test_data);

6.struct proc_dir_entry *proc_symlink(constchar*name,structproc_dir_entry*parent,constchar*dest);

Создание символической ссылки.

7.unsignedlongcopy_to_user(void_user*to,constvoid*from,unsignedlongn);

8.unsignedlongcopy_from_user(void_user*to,constvoid*from,unsignedlongn);

Хорошим примером для демонстрации возможностей файловой системы /proc являются загружаемые модули ядра (LKM), позволяющие динамически добавлять и, при необходимости, удалять код из ядра Linux. LKM завоевали популярность как удобный механизм реализации в ядре Linux драйверов устройств и файловых систем.

Процессы Unix: демоны, примеры системных демонов, правила создания демонов

Демон - процесс, выполняющий какую-то фоновую задачу, не имеющий управляющего терминала и, как следствие, обычно не интерактивный по отношению к пользователю. «Демон» Unix примерно соответствует «сервису» Windows.

6 правил программирования процессов-демонов:

1) вызвать umask для сброса маски создания файлов (маска наследуется и может маскировать биты прав доступа (запись, чтение))

2) вызвать fork() и завершить предка

- чтобы командная оболочка думала, что команда была выполнена
- чтобы новый процесс гарантированно не был лидером группы, что позволит вызвать setsid (у дочернего процесса id отличный от родителя, а pgid наследуется)

Демон

```
int daemonize(void) {
switch (fork()) {
case 0:
return setsid();
case -1:
return -1;
default:
exit(0);}}
```

3) создать новую сессию, вызвав setsid, тогда процесс станет:

- лидером новой сессии
- лидером новой группы процессов
- лишится управляющего терминала (TTY = ?)

4) сделать корневой каталог текущим рабочим каталогом (если рабочий каталог на смонтированной файловой системе, то её нельзя будет отмонтировать, так как процессы-демоны обычно живут, пока система не перезагрузится). 5) закрыть все ненужные открытые файловые дескрипторы, которые процесс-демон может унаследовать и препятствовать их закрытию (для этого нужно сначала получить максимальный номер дескриптора (см. код))

6) такой процесс не связан ни с каким терминальным устройством и не может взаимодействовать с пользователем в интерактивном режиме, даже если он был запущен в рамках интерактивной сессии, он все равно будет переведен в фоновый режим (некоторые процессы-демоны открывают файловые дескрипторы 0 1 и 2 на dev/null - "пустые" stdin, stdout, stderr, что позволяет вызывать в них фунции стандартного ввода вывода, не получая при этом ошибок)

Примеры системных демонов (Стивен)

В ОС Linux демон **keventd** предоставляет контекст процесса для запуска задач из очереди планировщика. Демон **kapmd** обеспечивает поддержку расширенного управления питанием, которое доступно в некоторых компьютерных системах. Демон **kswapd** известен также как демон выгрузки страниц. Этот демон поддерживает подсистему виртуальной памяти, в фоновом режиме записывая на диск страницы, измененные со времени их чтения с диска (dirty pages), благодаря чему они могут быть использованы снова. Сбрасывание кэшированных данных на диск в ОС Linux производится с помощью двух дополнительных демонов **bdf lush** и **kupdated**. Демон **bdf lush** начинает сбрасывать измененные буферы из кэша на диск, когда объем свободной памяти уменьшается до определенного уровня. Демон **kupdated** сбрасывает измененные буферы из кэша на диск через регулярные интервалы времени, снижая тем самым риск потери данных в случае краха системы.

Демон **port map** осуществляет преобразование номеров программ RPC (Remote Procedure Call удаленный вызов процедур) в номера сетевых портов. Демон **syslogd** может использоваться программами для вывода системных сообщений в журнал для просмотра оператором. Сообщения могут выводиться на консоль, а также записываться в файл. (

Демон **crond** производит запуск команд в определенное время. Множество административных задач выполняется благодаря регулярному запуску программ с помощью демона **crond**. Демон **cupsd** это сервер печати, он обслуживает запросы к принтеру.

Управление устройствами: физические принципы управления устройствами. Буферизация ввода-вывода - управление буферами. Буферный пул, кеширование.

Основополагающие на физическом уровне принципы:

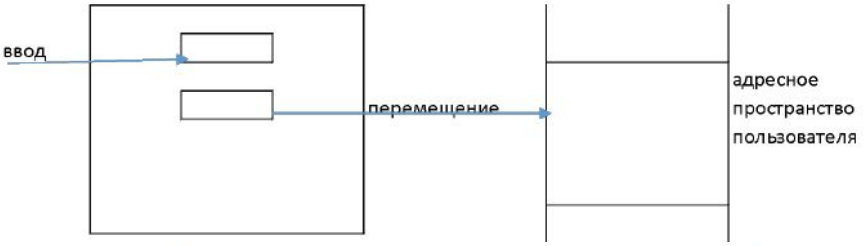
1. Все устройства делятся на символьные и блочные. Блочные устройства — это устройства, с которыми обмен информацией выполняется блоками фиксированного размера. Главным блочным устройством являются диски, а также ленточные запоминающие устройства. Стример — ленточное устройство очень большого объема. Символьные устройства. Обмен с ними осуществляется байтами. Системы (и Юникс, и Винд) поддерживают два типа файлов: блок-ориентированные файлы и байт-ориентированные файлы. Текстовыми файлы в системе являются файлы, в конце которых определен символ конца строки.Такое деление используется для достижения независимости от конкретных физических параметров отдельных устройств.
2. Принцип распараллеливания функций. Задача подключения к системе лежит внутри распараллеливания. Работа внешних устройств только инициируется центральным процессором, а управляет работой внешнего устройства другое специальное устройство. В вычислительной технике существует две глобальных архитектуры: канальная и шинная. В канальной архитектуре внешние устройства подключаются не непосредственно к ЦП, а к соответствующим каналам. Каналы бывают четырех типов: мультиплексорные, селекторные, блок-мультиплексорные и блок-селекторные. Канал является спецпроцессором, который фактически управляет внешними устройствами. Канал является программно-управляемым устройством и программу управления внешним устройством он получает от процессора (так называемая канальная программа).Шинная архитектура предполагает наличие специальных устройств, управляющих внешними устройствами, которые в одном случае называются контроллерами, а в другом адаптерами. Фактически в шинной архитектуре любое внешнее устройство состоит из двух частей. Собственно, устройство (функции, которые предписаны устройству) и контроллер устройства. Контроллер — устройство, предназначенное для управления работой внешнего устройства. Программно-управляемый, получает от процессора команду для управления.Задача адаптера фактически такая же (возможна совокупность адаптера и контроллера), но адаптер располагается на материнской плате. То есть то, что находится в устройстве — контроллер, на мат.плате — адаптер.На английском внешнее устройство может называться periferal device или external device.Трехшинная архитектура современных вычислительных систем. Безразлично, какая архитектура, шинная или канальная. В основу взаимодействия внешних устройств и процессора и оперативной памяти положено три вида сигналов. Сигналы — передаваемые данные, сигналы — передаваемые адреса, сигналы — передаваемые команды управления. Отсюда три шины: шина данных, шина адресов, шина управления. Это разделение является прямым следствием из архитектуры Фон Неймана. В современных системах эти сигналы разделены в пространстве. Это позволяет сократить время передачи информации. Предположим, что приложению надо считать из файла блок данных размером 512 байт. Это блок данных должен быть помещен в виртуальное адресное пространство процесса. Это непрерывная последовательность адресов размером 512 байт (некоторая непрерывная последовательность адресов размером 512 байт). Приложение выполняет команду fread(&block_adress, sizeof, blocks_number, file_desc).

Имеется три проблемы:

1. В ожидании завершения ввода процесс будет блокирован.
2. Очевидно, что для того, чтобы поместить данные с диска в виртуальное адресное пространство, должна быть задействована физическая память. Чтобы записать данные с диска по выделенным адресам, соответствующие страницы адресного пространства процесса должны находиться в физической памяти. Но процесс блокирован, значит система должна помечать эти страницы, как невыгружаемые. Проблема свопинга, то есть свопинг процесса оказывается невозможным.
3. Возможность взаимоблокировки. Процесс выполнил команду read и заблокирован. Соответственно, страницы процесса выгружаются в область свопинга до начала операции ввода-вывода. Затем процесс находится в состоянии блокировки до завершения операции ввода-вывода. Драйвер, который управляет запрошенным устройством, так же ждет по завершении операции ввода-вывода, когда процесс (на мой взгляд, здесь логичнее страницы процесса, а не процесс) будет загружен в основную память, чтобы загрузить данные в нужные ему адреса. Здесь возможна взаимоблокировка

(тупик). Взаимодействующие процессы в системе, каждый из которых ждет событие, которое может произойти в контексте другого процесса. Избежать взаимоблокировки можно заблокировав необходимые для ввода данных страницы в оперативной памяти.

Дисковое устройство работает с пулом буферов. В системе выполняется тотальная буферизация ввода-вывода. Потом они меняются ролями.Это все называется управлением буферами.

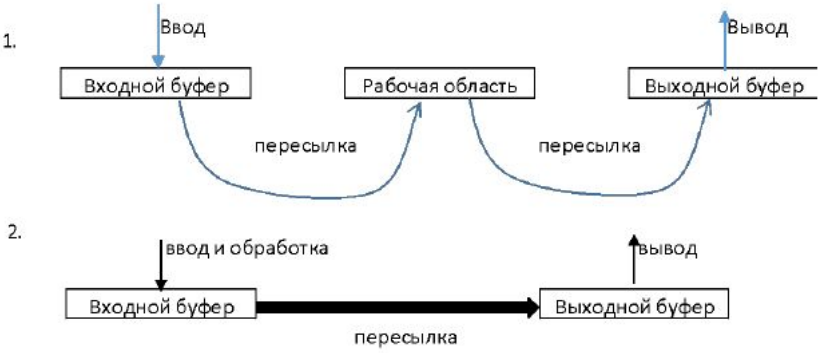


Схемы простой буферизации.

Простая буферизация. Связана с временными задержками. Метод применим, когда объем данных

относительно небольшой.

Альтернативой простой буферизации является обменная буферизация.

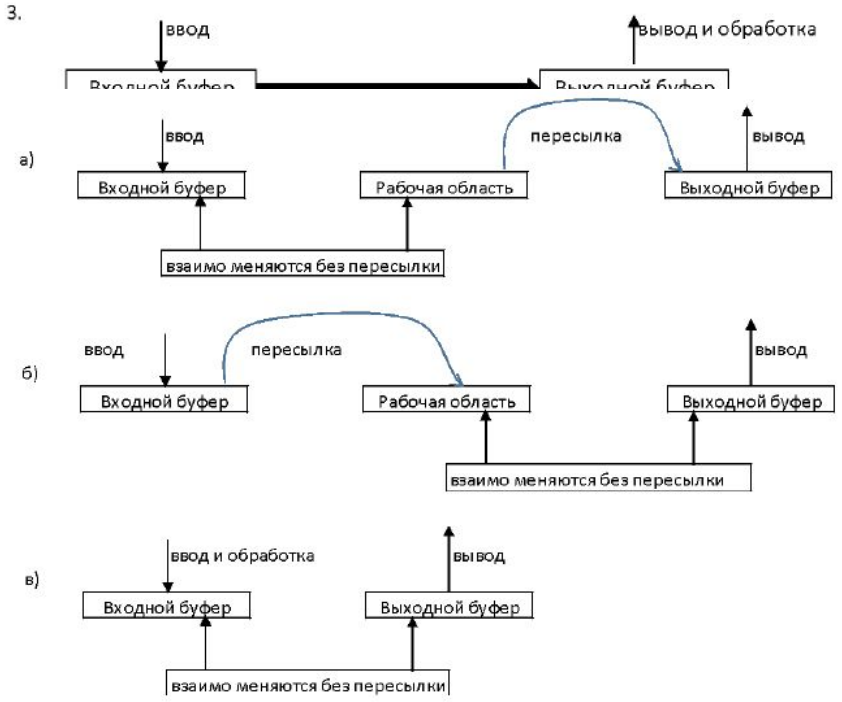


Схемы обменной буферизации
Для управления буферами в системе создается специальная таблица. Каждый элемент этой таблицы описывает один буфер.

Буферный пул или отдельный буфер могут быть созданы:

а) статически во время выделения памяти задаче. Такой буфер будет существовать все время существования задачи (под задачей в данном случае можно понимать процесс или нить)

б) динамически. Динамическое создание является более эффективным с точки зрения ресурсов системы. буферный пул создается перед началом обмена с внешним устройством. По завершении обмена буфера освобождаются



Буферный пул формируется одним из следующих способов:

1. Созданием в обрабатывающей программе специальной области памяти и последующим выполнением макро-команды build (BUILT?), которая связывает данный буферный пул с соответствующим набором (наборами) данных.
2. С помощью макро-команды GETPOOL, которая требует, чтобы операционная система создала буферный пул. Он будет освобожден командной FREEPOOL
3. Разрешением ОС автоматически создавать буферный пул при открытии набора данных.

Подсистема ввода-вывода (управления устройствами): задача, место в ОС. Основные функции ОС, связанные с управлением устройствами. Программные принципы управления. Драйверы. Точки входа в драйверы. Устройства в Unix. Модули ядра в Linux: особенности, пример.

Программные принципы.

Принцип невидимости (transparent). Все сложности, связанные с управлением устройствами, должны быть невидимы приложениям. Достигается это большим объёмом системного ПО. Основная задача ОС – обеспечение обмена информацией между устройствами.

Украшающие принципы – набор алгоритмов, скрывающих детали аппаратного обеспечения и создающих более приятное окружение. Т.о. менеджер памяти – скрытая трансляция адреса, реализованная таким образом, чтобы процессы могли жить в более приятном виртуальном мире.

Управление устройствами.

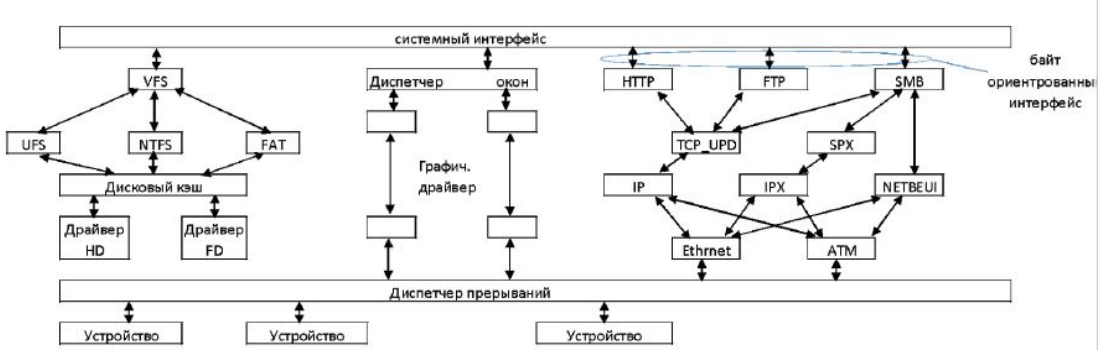
Основные функции управления устройствами:

1. Отслеживание состояния всех устройств, для чего требуется наличие специальных программных механизмов (т.е. ОС для управления устройством должна располагать информацией об устройстве, которая содержится в блоке управления устройствами (UCB)).
2. ОС принимает решение, кому выделить устройство, на какой срок и когда. Принятие решения зависит от принятой в системе стратегии распределения устройства. Существует 3 основных способа использования устройств:
    a. Закрепление устройства (т.е. монопольное использование устройства одним процессом)
    b. Разделение устройства (т.е. совместное использование устройства несколькими процессами)
    c. Виртуализация устройства
3. Непосредственное выделение устройства. Выделение устройства связано с физическим приписыванием устройства процессу. Кроме устройства процессу должны быть приписаны соответствующие буфера и каналы.
4. Освобождение устройства. Связано с соответствующим редактированием системной информации.

При это ключевой концепцией программного обеспечения ввода-вывода является концепция независимых процессов от устройств. Эта концепция означает возможность написания прог, способных получать доступ к любому устройству без предварительного указания конкретного устройства. Например, программа, работающая с файлами, должна с одинаковым успехом работать, читая данные с HDD, CD, Flash. Для решения этой задачи ОС обрабатывает запросы, как будто бы они адресованы файлом.

В Юникс для этого есть Device-file – специальный файл, ассоциированный с определенным устройством. Все файлы устройств находятся в /dev или его подкаталогах. С точки зрения пользователя файл устройства ничем не отличается от обычного файла. Пользователь может открывать, закрывать, читать, писать данные. Командный интерпретатор умеет направлять stdin, stdout, stderr в файл устройства. Причем данные преобразуются в определенные действия, например запись в файл /dev/lpt приведёт к печати данных на принтере.

Многослойная модель системы ввода-вывода.



Имеется горизонтальное и вертикальное деление. Это объясняется тем, что задачи невозможно решить единообразно для разных устройств.

Этот принцип для различных устройств реализуется по-разному.

В схеме: подсистема управления дисками, графическая (мониторы, принтеры), управление сетевыми адаптерами.

В каждой из этих подсистем существует несколько слоев, нижний – драйверы устройств (аппаратные драйверы). Они управляют непосредственно аппаратурой внешних устройств.

Задачи подсистем ввода-вывода:

1. Организация параллельной работы устройств ввода-вывода и процессора.
2. Согласование скоростей обмена и кеширование данных. Для этого используется буферизация, которая позволяется также сократить количество реальных операций ввода-вывода.
3. Разделение устройств и данных между процессами. ОС должна обеспечивать контроль доступа теми же способами, какими она обеспечивает доступ процессов к ресурсам ОС. При этом ОС должна обеспечивать доступ не только к устройству в целом, но и к отдельным порциям данных, хранимых или отображаемых устройством (например, диск – доступ к отдельным подкаталогам; графический дисплей – отдельные окна есть ресурсы, к которым необходимо обеспечить доступ с соответствующими правами). Устройства бывают разделяемые и монопольно используемые, отсюда устройства делятся на выделенные и закреплённые.
4. Обеспечение удобного логического интерфейса между устройствами и остальной частью ОС.
5. Поддержка широкого спектра драйверов с возможностью простого включения в систему нового драйвера.
6. Динамическая загрузка и выгрузка драйверов.
7. Поддержка нескольких ФС.
8. Поддержка синхронного и асинхронного ввода-вывода.

Пространство имен устройств.

Старший и младший номера.

Идентификация и обращение к устройству определяются пространством имен устройств. В Юникс существует три пространства имен устройств:

1. Аппаратное пространство – идентифицирует устройство по контроллеру, к которому оно присоединено, а также логическому номеру контроллера.
2. Ядро применяет для именования устройств их нумерацию.
3. Полные имена файлов – для пользователей простое и понятное

пространство имен. ОС должна как-то хранить информацию об устройствах аналогично хранению И. о файлах. Для этого отводятся отдельные inode’ы, которым соответствуют внешние устройства. Ядро идентифицирует каждое устройство по типу (блочное, символьное), а также по паре номеров, которые называются старший и младший номер устройства. Старший номер идентифицирует тип устройства (мышь, клавиатура), или, что более точно, его драйвер. Младший номер идентифицирует определенный экземпляр устройства.

Например, диск: имеет 1 старший номер, разделы диска имеют разные ФС, им присвоены разные младшие номера. В результате каждый раздел диска имеет старший номер и младший номер.

В разных ОС для хранения номера устройства отводится разное количество байтов. В POSIX1 определен тип для хранения номеров dev_t, но его размер и содержимое не оговаривается.

В <sys/types.h> определен ряд зависимых от реализации типов данных, которые называются элементарными системными типами данных. Они объявляются с использованием директивы type_dev.

Например, dev_t – номер устройства ст/мл, sigset_t – набор сигналов, time_t – счётчик секунд календарного времени. В Linux тип dev_t определён в файле <sys/sysmacroc.h>, который инклудится в файле <sys/types.h>. Обычно ст и мл номера можно получить с помощью макросов, находящихся в <linux/kdev_t.h> MAJOR(dev_t dev) и MINOR(dev_t dev). Для получения устройства по номерам можно использовать макрос MKDEV(int mj, int mn). В dev_t в поле st_dev для каждого файла хранится номер устройства файловой системы, в котором располагается файл и соответствующий индексный узел.

Таблица специальных файлов

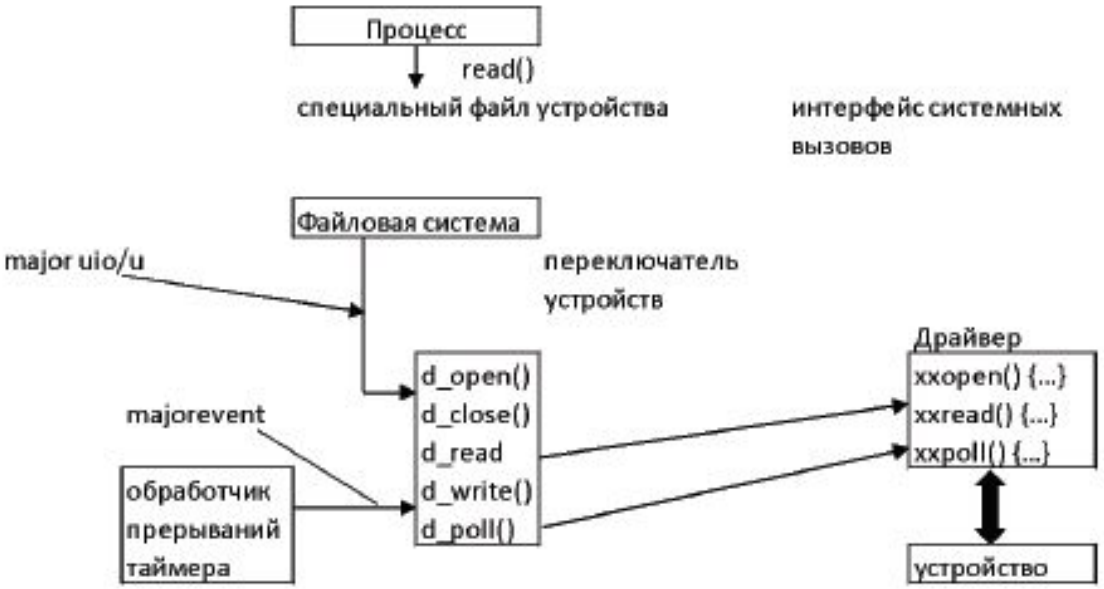
В ФС устройство имеет idode аналогично файлам. На уровне системы различие типов внешних устройств и возможное наличие нескольких устройств одного типа отражается в индексных дескрипторах специальных файлов структурой из двух полей: d_minor и d_major. Поля размещены в символьном массиве di_addr[40] адресного поля индексного дескриптора dinode, который у каталогов и файлов используется для адресации блоков данных на внешнем устройстве. di_addr[1] – это di_major. Доступ к драйверам внешних устройств осуществляется с помощью так называемых переключателей.

Блочное устройство определяется как bdevsw, символьное – cdevsw.

```
struct cdevsw
{
int (*d_open)();
int (*d_close)();
int (*d_read)();
int (*d_write)();
int (*d_ioctl)();
}
struct bdevsw
{
int (*d_open)();
int (*d_close)();
int (*d_strategy)();
int (*d_print)();
int (*d_size)();
}
```

Эти структуры переназначены для записи в системные таблицы, которые представляют собой массивы этих структур.

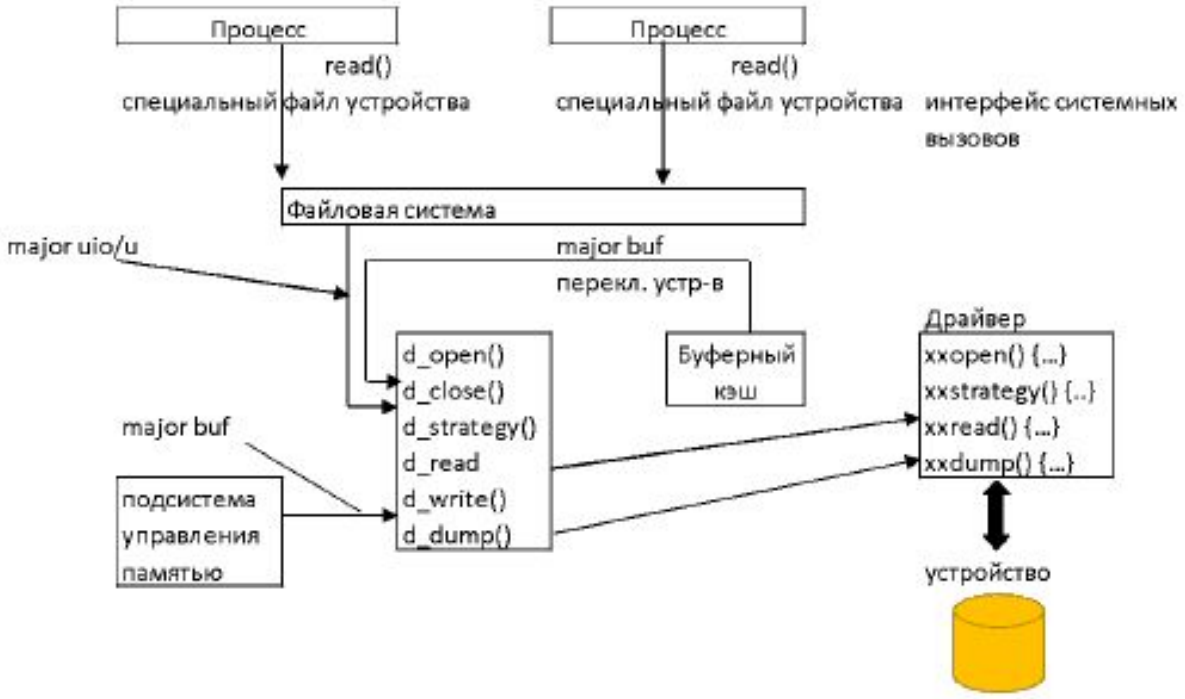
Доступ к драйверу символьного устройства.



Драйвер – часть ядра, предназначенная для управления устройством. Символьные драйверы предназначены для устройств, ориентированных на передачу и прием произвольной последовательности байтов.

Минимальный набор функций ядра Unix и их функциональность заключается в том, чтобы взять/поместить данные из/в виртуальное адресное пространство пользователя.

Доступ к драйверу блочного устройства



Размер буфера кратен 512. Блочные драйверы являются более сложными, работают с использованием возможности системной буферизации блочных обменов ядра. В число функций такого драйвера входит включение соответствующего блока данных в систему буферов ядра ОС или взятие содержимого буферной области, если необходимо.

В Юникс блочные устройства вообще не доступны пользовательским программам. Все операции с ними осуществляются посредством ФС, что упрощает логику используемых операций, так как драйверу не надо заботиться об обмене данными с пользовательским адресным пространством.

В отличие от обычных операций чтения/записи в которых допустим обмен пакетами данных произвольного размера, блочный драйвер передает данные блоками, размер которых кратен 512 байтам. Если все-таки понадобился доступ к диску, драйвер создаёт две минорные записи для этого устройства: блочную и символьную и все-таки предоставляет обычные символьные операции чтения/записи.

Ещё более радикально подошёл к проблеме взаимодействия с блочными устройствами Торвальдс. В Linux драйвера блочных устройств обязаны предоставлять символьные операции чт/зап, но им не нужно создавать вторую минорную запись, т.к. пользователям разрешено работать с блочными устройствами также, как с символьными.

Также в системе могут быть потоковые драйверы. В ОС Юникс они предназначены для доступа к сетевым устройствам. Должны работать с многоуровневыми сетевыми протоколами.

Существует два способа включить драйвер в состав ядра:

1. Включить драйвер на стадии генерации системы. Тогда драйвер статически объявляется частью ядра системы.
2. Позволяет обойтись минимальным номером статических объявлений на стадии генерации ядра, фактически фиксируются только необходимые элементы статической таблицы. Драйвера загружаются динамически в ядро в любой момент.

После загрузки все драйверы (и статические, и динамические) работают единообразно.

Мы рассмотрели два типа устройств: блочные и символьные. В тех иллюстрациях, которые были приведены, были перечислены драйверы соответствующих устройств. В системах есть еще третий тип устройств: прозрачные (row) драйверы/ устройства. У драйвера будет точка входа open, close, могут быть read/write, обязательно будет ioctl(). Обработчики прерываний входят в состав драйверов.

Типичные точки входа в драйвер устройства

У драйверов есть уникальная часть имени (приставка – здесь xx) и обязательная часть.

- xxopen()
- xxclose()
- xxread()
- xxwrite()
- xxioctl()
- xxintr() – interrupt handler, вызывается при поступлении прерывания, связанного с данным устройством. Может выполнять копирование данных в промежуточные буфера.
- xxpoll()
- xxhalt()
- xxstrategy()
- xxxize() – используется дисковыми устройствами для определения размеров дисковых разделов.
- xxgetmap() – отображает буферную память устройство в адресное пространство процесса Используется символьными устройствами, отображаемыми в память при применении системного вызова mmap()
- xxmmap() – применяется только если применяется getmap().

Unix: команды - fork(); wait(); exec(); pipe(); signal().

Unix создавалась как ОС разделения времени. Базовое понятие Unix – процесс (единица декомпозиции ОС, программа времени выполнения).

Процесс рассматривается как виртуальная машина с собств. адресным пространством, выполн. пользов. прогр., предоставл. набор услуг. Процесс может находиться в двух состояниях –«задача»(процесс выполняет собственный код) и «система» или «ядро»(выполняет реентерабельный код ОС). Процессы все время переходят «пользователь» <=>«система».

Unix – ОС с динамическим управлением процессами. Любой процесс может создавать любое число процессов с помощью системного вызова fork() – ветвление. Процессы образуют иерархию в виде дерева процессов, процессы связ. отношением потомок.предок. В результате вызова fork создается процесс.потомок, который является копией процесса.предка (наследует адресное пространство предка и дескрипторы всех открытых файлов (фактически наследует код)). В Unix все рассматривается как файл (файлы, директории, устройства). fork() возвращает 0– для потомка, **.-1 (- адын)**– если ветвление невозможно, для родителя возвращ. натуральное число (ID потомка).Любой процесс имеет предка, кроме демонов. Все процессы имеют прародителя –Init с ID =0 (порожд. в нач. работы и существует до окончания работы ОС). Все ост. порожд. по унифиц. схеме с помощью сист. вызова fork().

Системный вызов exec(), заменяет адресное пространство потомка на адресное пространство программы, указанной в системном вызове. exec () НЕ создает новый процесс!!! Бывает шести видов: execlp, execvp, execl, execv, execle, execve. Например, execl(”/bin/ls”, ”ls”, “.l”, 0)

Системный вызов wait(&status) вызывается в теле предка. Предок б. ждать завершения всех своих потомков. При их завершении он получит статус завершения. `

Все процессы в Unix объединены в группы, процессы одной группы получают одни и те же сигналы (события). Т.к. Unix система разделения времени, то существует понятие терминала. Процесс, запустивший терминал имеет PID = 1. Все процессы, запущенные на этом терминале, являются его потомками.

Неименованный программный канал создается системным вызовом pipe(). Труба (pipe) –это специальный буфер, который создается в системной области памяти. Информация в канал записывается по принципу FIFO и не модифицируется. Родственники могут обмениваться сообщениями с помощью неименованного программного канала (симплексная связь). Программные каналы описываются в соответствующей системной таблице. Канал имеет собственные средства синхронизации. Канал д.б. закрыт для чт/зап, если в/из него пишут/читают.

```
int mp[2];
pipe (mp);
…
close(mp[0]);
write(mp[1], msg, sizeof(msg));
…
```

Прогр. канал буфферизация. на 3.х уровнях: в сист. обл. памяти; при переполнении наибольш. врем. существ. пишется на диск.; если процесс пишет >4 Кб, то канал буфер. по времени, пока все данные не б. прочитаны.

Любое важное событие в ОС сопровожд соотв. сигналом. Процессы м. порождать, принимать и обрабатывать сигналы. М.б. синхронные и асинхронные. Средство передачи сигнала – kill(), приема сигнала – signal(). Для изменения хода выполнения программы. Необходимо написать свой обработчик (в зависимости от того был получен сигнал или нет выполняются разные действия)

kill(getpid(), SIGALARM); – передать самому себе сигнал будильника.

Системный вызов signal возвращает указатель на предыдущий обработчик данного сигнала.

```
void catch(int num)
{
if(ChangedMode == 0) ChangedMode = 1;
else ChangedMode = 0;
}
…
signal(SIGINT, catch); // обработать полученный сигнал
```

Сокеты - определение, типы, семейства, виды сокетов. DGRAM сокеты. Примеры(программирование). Сетевые сокеты (с мультиплексированием и без), сетевой стек, установление соединения, порядок байтов. Сетевой стек (порядок вызова функций - схема), преобразование байтов (little-big endian), мультиплексирование.

Сокет – это абстракция конечной точки взаимодействия (end point) Для работы с сокетами используют дескрипторы сокетов. Сокеты были созданы для организации взаимодействия процессов обезличено (любой процесс может соединиться с любым процессом без ограничений). Могут использоваться как для соединения процессов на одной машине, так и на разных через сеть. Был создан набор функций, который определил интерфейс сокетов. Это функции можно использовать как в сети, так и на одной машине. Обмен данными через сокеты является двунаправленным.

Семейства(домены):

Определены константы для следующих семейств адресов. Все имеют приставку AF (address family):

1. AF_UNIX – сокеты для межпроцессного взаимодействия на локальном компьютере, домен – Юниксы.
2. AF_INET – сокеты семейства протоколов TCP/IP. Основаны на протоколе интернета версии 4 IPv4.
3. AF_INET6 – семейство протоколов TCP/IP, основанное на IPv6.
4. AF_IPX – семейство протоколов IPX.
5. AF_UNSPEC – неопределенный домен, который может представлять любой домен

Типы:

Внутри семейства протоколов TCP/IP различают три типа:

1. SOCK_STREAM – потоковые сокеты. Это надежная упорядоченная полнодуплексная логическое соединение между двумя сокетами. Всегда TCP. Ориентированы на создание логического соединения, упорядоченность передачи данных, гарантируется доставка сообщений, двунаправленный поток байтов. Напоминают телефонный звонок - сначала надо установить соединение а потом только общаться.

2. SOCK_DATAGRAM – определяют ненадежную службу datagram без установления логического соединения, где пакеты могут передаваться без сохранения порядка – широковещательная передача данных. UDP. Доставка сообщений не гарантируется. Напоминаю отправку письма по почте – можно много отправить, но гарантировать доставку их и что они не потеряются по дороге нельзя.

3. SOCK_RAW – низкоуровневый интерфейс. Так называемый прямой сокет. По протоколу IP, не обязательно POSIX1. При использовании этого интерфейса вся ответственность за построение заголовков пакетов возлагается на приложения, так как сокеты этого типа не используют протоколы транспортного уровня(TCP, UDP)

4) SOCK_SEQPACKET Ориентированы на создание логического соединения, упорядоченность передачи данных, сообщения фиксированной длины, гарантируется доставка сообщений. Похожи на SOCK_STREAM, за исключение того, что вместо услуги приёма/передачи данных в виде потока байтов они реализуют услугу передачи отдельных сообщений. То есть объём данных полученных от сокета этого типа каждый раз в точности совпадает с объёмом отправленных данных.

мультиплексирование

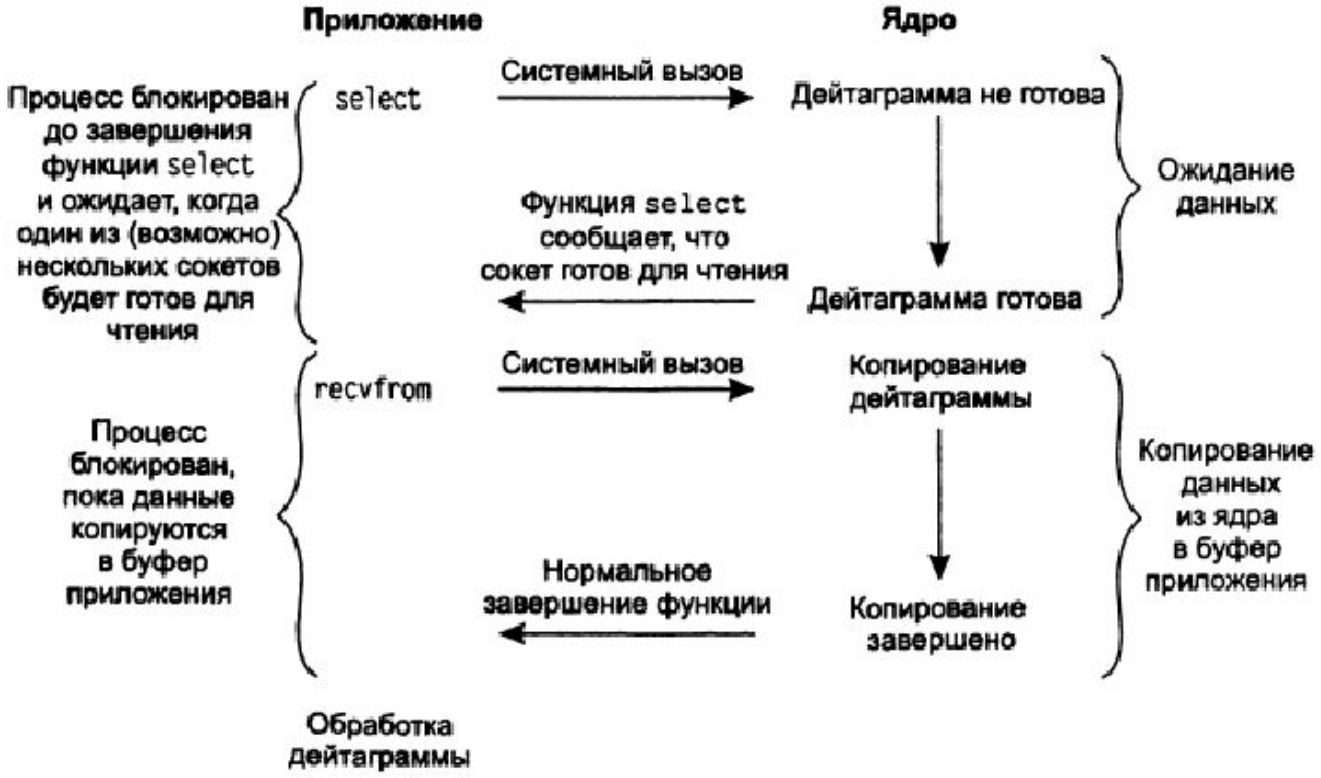


Рис. 6.3. Модель мультиплексирования ввода-вывода

Преобразование байтов

То есть порядок байтов - характеристика аппаратной архитектуры процессора, определяющая, в каком порядке следуют байты в данных длинных типов

1. Прямой порядок байтов(little-endian) // FreeBSD 5.2.1, Linux 2.4.22
2. Обратный порядок байтов(big-endian) // Mac OS X 10.3, Solaris 9

Чтобы не возникало путаницы при обмене данными между разнородными компьютерными системами, сетевые протоколы жёстко задают порядок байтов. TCP/IP использует обратный(big-endian) порядок байтов. Таким образом иногда возникает необходимость преобразовать порядок байтов, поддерживаемый аппаратной архитектурой в сетевой порядок байтов. Это производится через следующие 4 функции:

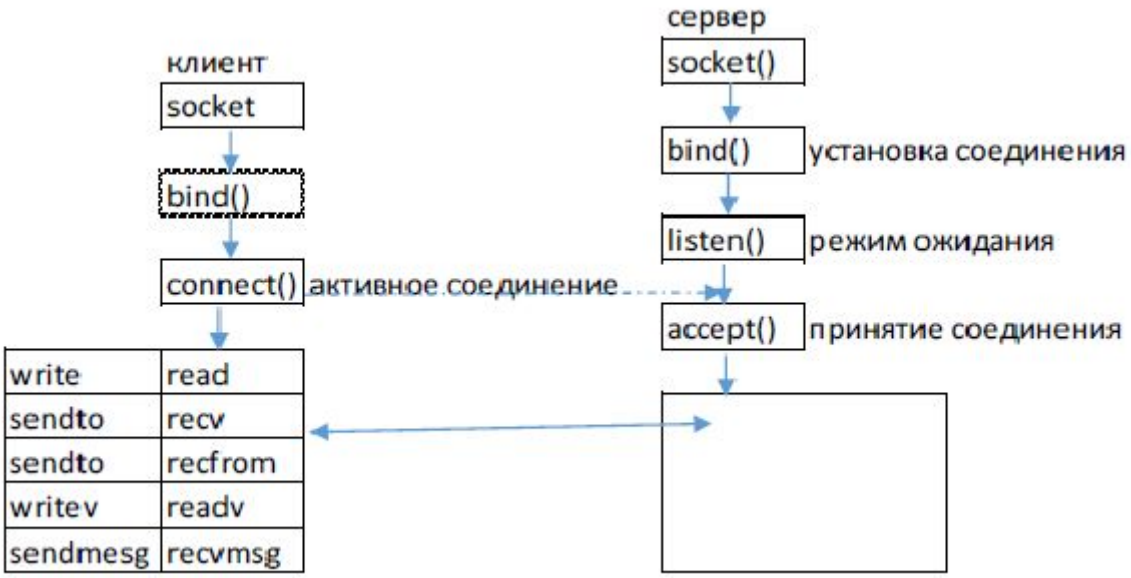
«include <arpa/inet.h> // В некоторых устаревших системах их определения в <neti-net/in.h>

- uint32_t htonl(uint32_t hostint32) // Возвращает 32-битное целое с сетевым порядком байтов
- uint16_t htons(uint16_t hostint16) // Возвращает 16-битное целое с сетевым порядком байтов
- uint32_t ntohl(uint32_t netint32) // Возвращает 32-битное целое с аппаратным порядком байтов
- uint16_t ntohs(uint16_t netint16) // Возвращает 16-битное целое с аппаратным порядком

// В именах функций h-host(аппаратный порядок байтов)

- // n-network(сетевой порядок байтов)
- // l - long(4)
- // s - short(2)

Сетевой стек:



Примеры:

Клиент:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define DATA "The sea is calm tonight, the tide is full . . ."

main(argc, argv)
     int argc;
     char *argv[];
{
     int sock;
     struct sockaddr_un name;

     sock = socket(AF_UNIX, SOCK_DGRAM, 0);
     if (sock < 0) {
          perror("opening datagram socket");
          exit(1);
     }

     name.sun_family = AF_UNIX;
     strcpy(name.sun_path, argv[1]);

     if (sendto(sock, DATA, sizeof(DATA), 0,
         &name, sizeof(struct sockaddr_un)) < 0) {
          perror("sending datagram message");
     }
     close(sock);
}
```

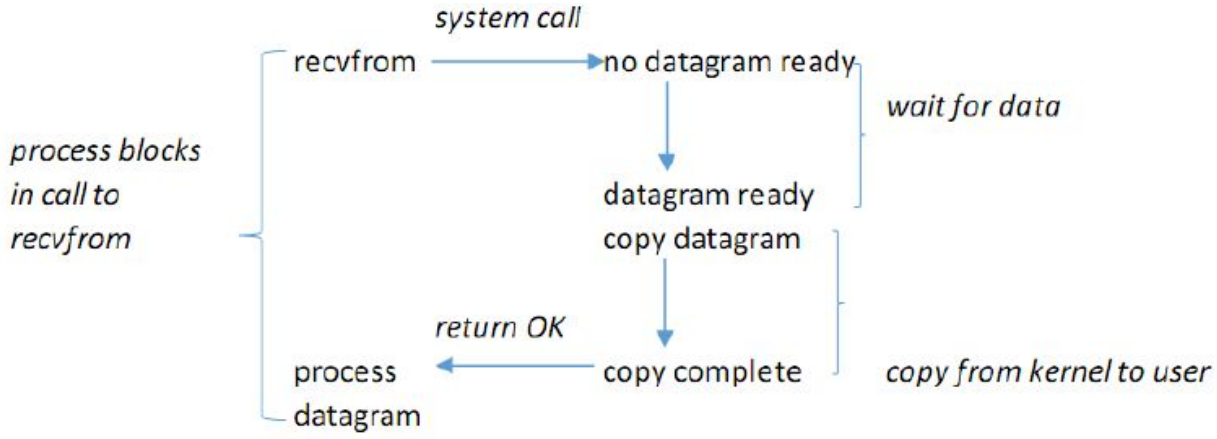
Сервер:

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

#include <stdio.h>

#define NAME "socket"

main()
{
     int sock, length;
     struct sockaddr_un name;
     char buf[1024];
     /* Create socket from which to read. */
     sock = socket(AF_UNIX, SOCK_DGRAM, 0);
     if (sock < 0) {
          perror("opening datagram socket");
          exit(1);
     }
     /* Create name. */
     name.sun_family = AF_UNIX;
     strcpy(name.sun_path, NAME);
     if (bind(sock, &name, sizeof(struct sockaddr_un))) {
          perror("binding name to datagram socket");
          exit(1);
     }
     printf("socket -->%s\n", NAME);
     /* Read from the socket */
     for (;;) {
          if (read(sock, buf, 1024) < 0)
               perror("receiving datagram packet");
               printf("-->%s\n", buf);
     }
}
```

Пять моделей ввода-вывода в Linux: диаграммы, особенности. Модель ввода-вывода с мультиплексированием.

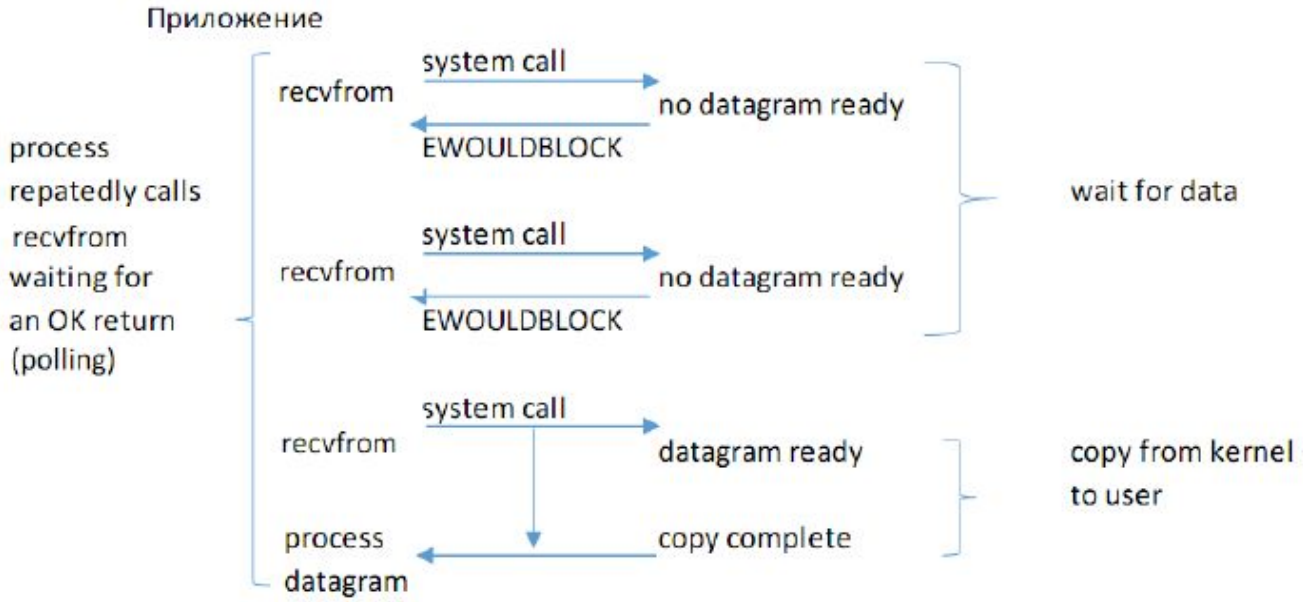
В Юникс-подобных системах программисту доступно 5+1 различных моделей ввода-вывода.

1. Первый способ ввода-вывода – блокирующий ввод-вывод (blocking IO). Почти все обсуждения связаны с таким вводом-выводом. (рис 1. Комментарий: recvfrom – и есть системный вызов, этот вызов переведет систему в режим ядра. Если данные не готовы, то процесс на recvfrom блокируется до того момента, пока данные не станут доступны приложению. Для этого система должна выполнить значительную последовательность действий. Когда данные готовы, их необходимо скопировать в буфер приложения. Когда данные скопированы в пользовательский буфер.)



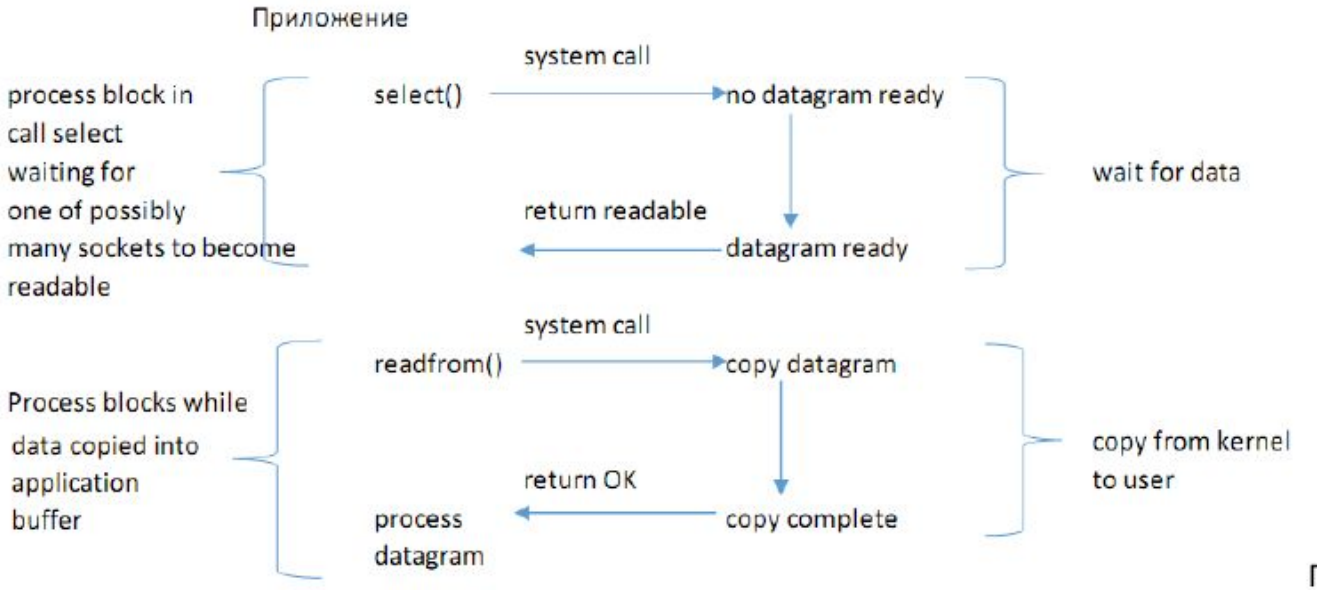
Recvfrom вызывается процессом, система переключается в режим ядра, система блокируется до тех пор, пока данные не поступят в буфер приложения. Пока процесс блокирован, он не отвечает на запросы. Для сетей это может стать проблемой. Время блокировки зависит от времени связи. В это время нет ответа на запросы. Система тормозится.

2. Неблокирующий ввод-вывод. (nonblocking IO). (лист 2)

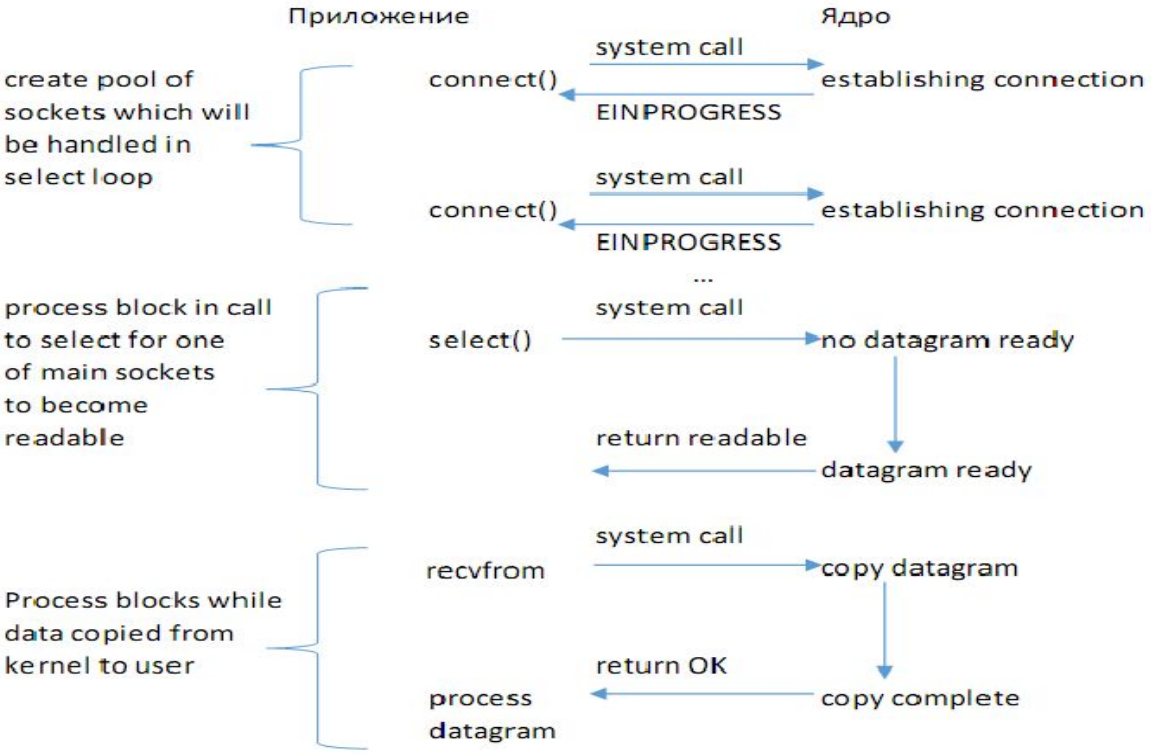


Процесс в цикле читает данные до тех пор, пока не будет возвращен результат. Ядро копирует данные в буфер процесса и они становятся доступны. Данный способ приводят к большим накладным расходам системы, так как во время опроса процессор занимается тем, что выполняет непроизводительные действия, но не блокируется. (аналог – опрос внешних устройств. Опрос может быть и более общим способом организации ввода-вывода.) data copied into application buffer

3. Мультиплексирование. При мультиплексировании используется один из возможных мультиплексоров: select, pselect, poll, epoll (рекомендован для Линукс). (лист 3).

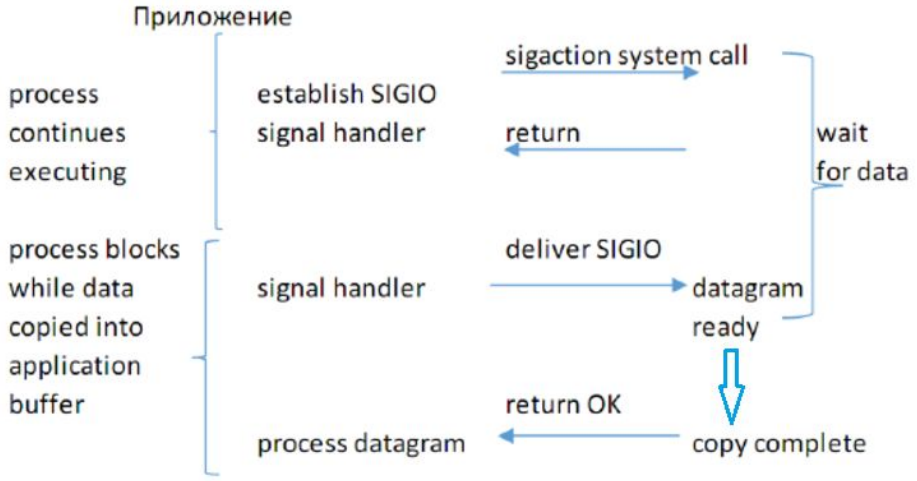


Процесс будет блокирован на select до тех пор пока на одном сокете из пола сокетов данные не станут доступны. Когда данные доступны (readable) выполняется системный вызов readfrom, который тоже блокирует процесс, но эта блокировка связана только с временем копирования данных из ядра в пространство пользователя. Преимущества перед блокируемым вводом-выводом состоит в том, что обрабатывается не один, а несколько дескрипторов. После получения статуса readable данные можно получить с помощью системного вызова recvfrom, но время блокировки здесь определяется только временем копирования данных из ядра пользователю. Время блокировки при мультиплексирование вводе-выводе за счет того, что готовность данных формируется не одним дескриптором, а пулом дескрипторов. Это можно развить. (лист 4).



В случае с мультиплексированием в цикле проверяются все сокеты и берется первый готовый. Пока обрабатывается первый, могут стать готовыми и другие, то есть могу стать доступны данные на других соединениях. В результате сокращается время простоя, то есть время блокировки процесса. Очевидно, что нельзя недооценивать мультиплексирование. (+1). Способ без названия (тот самый +1 из 5+1 способов). Похожий на мультиплексирование, но не мультиплексирование. Способ, при котором запускается несколько процессов или потоков, в каждом из которых выполняется блокирующий ввод-вывод. Надо обработать данные от нескольких источников, для этого мы запускаем или какое-то количество процессов, или какое-то количество потоков. Недостатки: а) в Юникс, как говорят, потоки «дорогие» - имеют высокие накладные расходы. б) относительно питона-GIL. В одном процессе только один поток. Решение – создавать дочерние процессы, но возникает проблема межпроцессорного взаимодействия.

4. Ввод-вывод, управляемый сигналами. (лист 5) (Signal-driver IO).



Запущен обработчик события, он должен прислать соответствующее сообщение (как в лабораторной в прошлом году, когда обработчик срабатывал на нажатие cntrl+c). Здесь сигнал будет послан в результате поступления данных. Все время, которое нужно ожидать данные, процесс будет работать. Сначала надо установить параметры сокета для работы с сигналами и установить обработчик сигнала системным вызовом sigaction. Результат возвращается сразу. Процесс не блокируется, всю работу

берет на себя ядро (отслеживает, когда данные будут готовы; после готовности данных ядро посылает сигнал sigio, в результате чего будет вызван установленный на этот сигнал обработчик. Соответственно сам вызов recvfrom можно сделать либо в обработчике сигнала, либо в основном потоке программы. Однако, обработчик сигнала типа sigio для каждого процесса может быть один. То есть, реакция на получения сигнала сигио должна быть однозначной. В результате за один раз можно соединиться только с одним файловым дескриптором.

5. Асинхронный ввод-вывод. (лист 6)

Некоторые системы предоставляют специальные системные вызовы. Идея асинхронного ввода-вывода состоит в том, что ядру дается команда «начать операцию», при этом процесс не блокируется, продолжает выполнять какие-то действия и ядро должно сообщить процессу или с помощью сигналов или как-то еще, что операция ввода-вывода завершена, после чего процесс должен перейти на обработку полученных данных. Модель выглядит следующим образом. Как видно из диаграммы, делается системный вызов aioread. Всю остальную работу выполняет ядро. Асинхронный – все действия по обработке ввода-вывода выполняются с собственной скоростью (ярчайший пример – прерывания). Проблема состоит в том, что необходимо получать асинхронное событие синхронно.

Адресация прерываний в защищённом режиме

Здесь всё очень весело (или нет…):

- Инициируется прерывание (аппаратное или при помощи инструкции INT).
- Посылается сигнал контроллеру прерываний
- Контроллер выбрасывает на шину данных вектор прерывания и через шину управления дёргает ногу INTERRUPT у проца
- Проц после выполненной инструкции видит, что у него нога дёрганная, получает вектор с шины и лезет в IDTR (там адрес IDT)
- Из IDT берётся дескриптор (если в байте доступа 0-ой бит = 0, то это прерывание, иначе это исключение но на это ПОФИГ). Первые 32 бита - это селектор:смещение.
- Проц лезет в GDTR, узнаёт адрес GDT и по селектору берёт адрес сегмента из дескриптора в таблице
- Потом прибавляется к этому адресу прибавляется смещение и-и-и-и…
- У нас получен физический адрес обработчика прерываний!
- И проц с радостью начинает его выполнять

Прилетел Druide и оставил это здесь:

Прерывания в защищенном режиме:

IDT (interrupt descriptor table) – таблица, предназначенная для хранения адресов обработчиков прерываний. Базовый адрес IDT помещен IDT Register. IDT столько, сколько процессоров.

Обработчик прерывания:

IDTR (указывает на начало IDT) + смещение из прерывания = дескриптор в IDT.

Из дескриптора в IDT берем селектор. С помощью селектора узнаем, с какой таблицей мы работаем.

Если работаем с GDT, то с помощью селектора получаем дескриптор сегмента, в котором находится наш обработчик, в этом сегменте с помощью смещения из дескриптора в IDT мы получаем точку входа в обработчик прерывания. Если работаем с LDT, то с помощью LDTR (в котором у нас смещение до дескриптора сегмента в GDT, в котором находится LDT) находим этот дескриптор, получаем сегмент. В этом сегменте находится нужная LDT, в ней с помощью селектора получаем дескриптор сегмента, в котором находится наш обработчик, в этом сегменте с помощью смещения из дескриптора в IDT получаем точку входа в обработчик прерывания.

Что-то там про QNX.

Операционные системы реального времени (ОСРВ)QNX.Применять понятие реального времени к физическим устройствам бессмысленно. Мы и они живут в реальном времени. Это понятие возникло для операционных систем (может быть и ПО спец назначения).

Ключевым отличием ядра ОСРВ является детерминированность (основанный на строгом контроле времени). Детерминированность определяется тем, что они управляются внешними системами/устройствами. Они завязаны на характеристиках внешних устройств. Это и есть детерминированность. Её следствием является требование обеспечения операционной системы соответствующих сервисов за определенные промежутки времени. Т.е. в ОСРВ главным критерием эффективность являетсяобеспечение временных характеристик вычислительного процесса

Различаются системы реального времени:

- гибкие (система резервирования билетов; если из-за временных задержек оператору не удается зарезервировать билет, это не страшно так как запрос может быть повторен);
- жесткие (система управления атомным реактором; временные задержки/отказ системы недопустимы)

В системах реального времени использовались статические приоритеты. В настоящее время в ОСРВ используется подходы:

- статические приоритеты в зависимости от значимости процессов;
- динамические приоритеты (пересчитываются).

Переключение потоков и процессов мало чем отличается. Процесс в QNX выполняется в собственном виртуальном адресном пространстве. Само ядро никогда не получает управление в результате диспетчеризации. Код ядра выполняется только как следствие системных вызовов, исключений или аппаратных прерываний.Единица диспетчеризации – поток.

Physical Address Extension (PAE) — режим работы встроенного блока управления памятью x86-совместимых процессоров, в котором используются 64-битные элементы таблиц страниц (из которых для адресации используются только 36 бит), с помощью которых процессор может адресовать 64 ГБ физической памяти (вместо 4 ГБ, адресуемых при использовании 32-разрядных таблиц), хотя каждая задача (программа) всё равно может адресовать максимум до 4 ГБ виртуальной памяти[1]. Также в новых моделях процессоров в PAE-режиме старший бит элемента таблицы страниц отвечает за запрет исполнения кода в странице, что затрудняет атаку по методу переполнения буфера.

Впервые расширение появилось в процессоре Pentium Pro. Для использования 36-разрядной адресации памяти необходима поддержка расширения физических адресов на программном уровне (включение режима PAE в ОС) и аппаратном: необходима поддержка как со стороны процессора, так и материнской платы (можно определить по команде CPUID). Материнские платы с поддержкой PAE, как правило, были дорогими и предназначенными для серверов.[2].