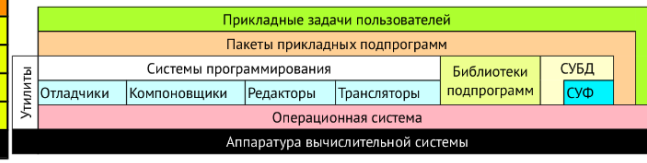
1) Вычислительная система. Обобщенная структура и иерархическое представление. Понятие ресурса и виртуализация

Вычислительная система рассматривается как многослойная структура из нескольких функциональных уровней. Каждый уровень определяется данными, выполняемыми функциями и результатами.
Уровни:

Обобщенная структура вычислительной системы (взгляд IT-специалиста)



Ресурс — всякий потребляемый объект, независимо от формы его существования, обладающий некоторой практической ценностью для потребителя.
Вычислительный ресурс — возможности, обеспечиваемые компонентами вычислительной системы, расходуемые или занимаемые в процессе её работы. (процессоры, память и т.д.)
Виртуализация — предоставление набора вычислительных ресурсов или их логического объединения, абстрагированное от аппаратной реализации, и обеспечивающее при этом логическую изоляцию вычислительных процессов, выполняемых на одном физическом ресурсе.

2) Операционная система. Классификация и структура

Операционная система — с точки зрения программиста это программа, которая добавляет множество новых инструкций и особенностей помимо тех, которые обеспечиваются уровнем архитектуры набора инструкций. Это:
- виртуальная память;
- виртуализация оборудования;
- виртуализация инструкций ввода/вывода;
- виртуализация инструкций параллельной обработки.

Операционная система — комплекс управляющих и обрабатывающих программ, который
- выступает как интерфейс между аппаратурой компьютера и пользователем;
- предназначен для эффективного и безопасного использования ресурсов ВС.

Любой из компонентов прикладного программного обеспечения обязательно работает под управлением ОС.
Ни один из компонентов программного обеспечения, за исключением самой ОС, не имеет непосредственного доступа к аппаратуре компьютера. Пользователи взаимодействуют со своими программами через интерфейс ОС. Любые их команды, прежде чем попасть в прикладную программу, сначала про-ходят через ОС.

Классификацию операционных систем можно осуществлять несколькими способами
1. По способу организации вычислений:
- системы пакетной обработки (batch processing operating systems) — целью является выполнение максимального количества вычислительных задач за единицу времени; при этом из нескольких задач формируется пакет, который обрабатывается системой;
- системы разделения времени (time-sharing operating systems) — целью является возможность одновременного использования одного компьютера несколькими пользователями; реализуется посредством поочередного предоставления каждому пользователю интервала процессорного времени;
- системы реального времени (real-time operating systems) — целью является выполнение каждой задачи за строго определённый для данной задачи интервал времени.

2. По типу ядра:
- системы с монолитным ядром (monolithic operating systems);
- системы с микроядром (microkernel operating systems);
- системы с гибридным ядром (hybrid operating systems).

3. По количеству одновременно решаемых задач:
- однозадачные (single-tasking operating systems);
- многозадачные (multitasking operating systems).

4. По количеству одновременно работающих пользователей:
- однопользовательские (single-user operating systems);
- многопользовательские (multi-user operating systems).

5. По количеству поддерживаемых процессоров:
- однопроцессорные (uniprocessor operating systems);
- многопроцессорные (multiprocessor operating systems).

6. По поддержке сети:
- локальные (local operating systems) — автономные системы, не предназначенные для работы в компьютерной сети;
- сетевые (network operating systems) — системы, имеющие компоненты, позволяющие работать с компьютерными сетями.

7. По роли в сетевом взаимодействии:
- серверные (server operating systems) — операционные системы, предоставляющие доступ к ресурсам сети и управляющие сетевой инфраструктурой;
- клиентские (client operating systems) — операционные системы, которые могут получать доступ к ресурсам сети.

8. По типу лицензии:
- открытые (open-source operating systems) — операционные системы с открытым исходным кодом, доступным для изучения и изменения;
- проприетарные (proprietary operating systems) — операционные системы, которые имеют конкретного правообладателя; обычно поставляются с закрытым исходным кодом.

9. По области применения:
- операционные системы мэйнфреймов — больших компьютеров (mainframe operating systems);
- операционные системы серверов (server operating systems);
- операционные системы персональных компьютеров (personal computer operating systems);
- операционные системы мобильных устройств (mobile operating systems);
- встроенные операционные системы (embedded operating systems);
- операционные системы маршрутизаторов (router operating systems).

3) Мультизадачность и многопоточность. Концепция процесса. Диаграмма состояний и операции над процессами.

Многозадачность — это один из основных параметров всех современных ВС. Фундаментальным понятием для изучения работы современных ВС является понятие процесса — основного динамического объекта, над которыми ОС выполняет определенные действия. По ходу работы программы ВС обрабатывает различные команды и преобразует значения переменных. Для этого ОС должна выделить определенное количество оперативной памяти, закрепить за ней определенные устройства ВВ или файлы (откуда должны поступать входные данные и куда нужно доставить полученные результаты), то есть зарезервировать определенные ресурсы из общего числа ресурсов всей ВС.

Количество и конфигурация ресурсов могут изменяться с течением времени. Для описания таких активных объектов внутри вычислительной системы используется термин «процесс» — программа, загруженная в память и выполняющаяся.

Процесс — некоторая совокупность набора исполняющихся команд, ассоциированных с ним ресурсов и текущего момента его выполнения, находящуюся под управлением ОС.

Многопоточность — свойство платформы (например, операционной системы, виртуальной машины или приложения, состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся «параллельно», то есть без предписанного порядка во времени. При выполнении некоторых задач такое разделение может достичь более эффективного использования ресурсов вычислительной машины.

Диаграмма состояний показывает, как объект переходит из одного состояния в другое. Диаграммы состояний служат для моделирования динамических аспектов системы. Данная диаграмма полезна при моделировании жизненного цикла объекта. От других диаграмм диаграмма состояний отличается тем, что описывает процесс изменения состояний только одного экземпляра определенного класса – одного объекта, причем объекта реактивного, то есть объекта, поведение которого характеризуется его реакцией на внешние события.

Процесс не может сам перейти из одного состояния в другое. Изменением состояния процессов занимается ОС, совершая над ними операции:
Создание — завершение
Приостановка — запуск
Блокирование — разблокирование
Изменение приоритета

4) Структуры управления процессами. Контексты. Переключение контекста

Мультизадачные возможности — это когда процессор выполняет какую-то одну программу (процесс или задача).
По истечении некоторого времени (микросекунды), операционная система переключает процессор на другую программу. При этом все регистры текущей программы (состояние, контекст) сохраняются. Че-рез некоторое время вновь передается управление этой программе. Программа при этом не замечает ка-ких-либо изменений — для нее процесс переключения незаметен.

Понятие процесса охватывает:
– некоторую совокупность исполняющихся команд;
– совокупность ассоциированных с процессом ресурсов — выделенная для исполнения память или ад-ресное пространство, стеки, используемые файлы, устройства ввода-вывода, и т. д (прикладной кон-текст, системный контекст).
– текущий момент выполнения — значения регистров и программного счетчика, состояние стека, зна-чения переменных (регистровый контекст).
Информацию, для хранения которой предназначен блок управления процессом, можно разделить на две части:
– регистровый контекст — содержимое всех регистров процессора (включая значение программного счетчика);
– системный контекст — все остальное.

Знания регистрового и системного контекстов процесса достаточно для того, чтобы управлять его пове-дением в операционной системе, совершая над ним операции, однако недостаточно, чтобы полностью характеризовать процесс.
Пользовательский контекст — код и данные, находящиеся в адресном пространстве процесса.
Контекст процесса — совокупность регистрового, системного и пользовательского контекстов.
В любой момент времени процесс полностью характеризуется своим контекстом.

5) Планирование процессов. Долгосрочное и краткосрочное планирование. Критерии и требования к алгоритмам.

Планирование заданий используется в качестве долгосрочного планирования процессов. Оно отвечает за порождение новых процессов в системе, определяя ее степень мультипрограммирования, т. е. коли-чество процессов, одновременно находящихся в ней. Если степень мультипрограммирования системы поддерживается постоянной, т. е. среднее количество процессов в компьютере не меняется, то новые процессы могут появляться только после завершения ранее загруженных. Поэтому долгосрочное пла-нирование осуществляется достаточно редко, между появлением новых процессов могут проходить ми-нуты и даже десятки минут.

Планирование использования процессора применяется в качестве краткосрочного планирования про-цессов. Оно проводится, к примеру, при обращении исполняющегося процесса к устройствам ввода-вывода или просто по завершении определенного интервала времени. Поэтому краткосрочное планиро-вание осуществляется, как правило, не реже одного раза в 100 миллисекунд.

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

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

6) Алгоритмы планирования. First-Come, First-Served (FCFS). Round Robin (RR). Shortest-Job-First (SJF).

First Come First Serve (FCFS) — это алгоритм планирования операционной системы, который автома-тически выполняет запросы и процессы в очереди в порядке их поступления. Это самый простой и про-стой алгоритм планирования процессора. В алгоритме этого типа процессы, которые сначала запраши-вают ЦП, сначала получают распределение ЦП. Это управляется с помощью очереди FIFO. Полной формой FCFS является First Come First Serve.

Когда процесс входит в готовую очередь, его PCB (блок управления процессом) связывается с хвостом очереди и, когда ЦП становится свободным, его следует назначить процессу в начале очереди.

SCHED_FIFO

Потоки, запланированные в соответствии с этой политикой, выбираются из списка потоков, который упорядочен по времени того, сколько они находятся в списке, но не выполняются.

Как правило, в голове списка поток, который был в списке дольше всех, а в хвосте — который был в списке самое короткое время.

В соответствии с политикой **SCHED_FIFO**, изменение определяющих списков потоков выглядит следующим образом:

1. Когда выполняющийся поток вытесняется (preempted), он становится в голову списка потоков своего приоритета.
2. Когда заблокированный поток становится готовым к исполнению потоком, он становится в хвост списка потоков своего приоритета.
3. Когда выполняющийся поток вызывает функцию **sched_setscheduler()**, процесс, указанный в вызове функции, изменяется в соответствии с указанной политикой и приоритетом, указанным в аргументе **param**.
4. Когда выполняющийся поток вызывает функцию **sched_setparam()**, приоритет процесса, указанного в вызове функции, изменяется до приоритета, указанного в аргументе **param**.

---

5. Когда выполняющийся поток вызывает функцию **pthread_setschedparam()**, поток, указанный в вызове функции, изменяется на указанную политику и приоритет, указанный аргументом **param**.
6. Когда выполняющийся поток вызывает функцию **pthread_setschedprio()**, поток, указанный в вызове функции, изменяется до приоритета, указанного аргументом **prio**.
7. Если поток, политика или приоритет которого были изменены не с помощью **pthread_setschedprio()**, является выполняющимся потоком или может быть запущен на выполнение, тогда он становится хвостом списка потоков для своего нового приоритета.
8. Если поток, приоритет которого был изменен с помощью **pthread_setschedprio()**, является выполняющимся потоком или может быть запущен на выполнение, влияние на его позицию в списке потоков зависит от направления *модификации*, как показано ниже:
    a. если приоритет повышен, поток становится хвостом списка потоков;
    б. если приоритет не изменяется, поток не меняет позицию в списке потоков;
    с. если приоритет понижен, поток становится головой соответсвующего списка потоков.
9. Когда запущенный поток вызывает функцию **sched_yield()**, поток становится хвостом списка потоков по своему приоритету.

Допустимые пределы приоритетов для политики можно получить, используя функции **sched_get_priority_max()** и **sched_get_priority_min()**, предоставляя в качестве параметра **SCHED_FIFO**.

Минимальный диапазон приоритетов **SCHED_FIFO** содержит не менее 32 приоритетов.

Round Robin

Название этого алгоритма происходит от принципа циклического перебора, когда каждый человек по-лучает равную долю чего-то по очереди. Это самый старый, самый простой алгоритм планирования, который в основном используется для многозадачности. В циклическом планировании каждая готовая задача выполняется по очереди только в циклической очереди в течение ограниченного промежутка времени. Этот алгоритм также предлагает выполнение процессов без голодания.

SCHED_RR

Политика планирования с циклическим перебором (round-robbin — карусель).

Эта политика идентична политике **SCHED_FIFO** с дополнительным условием, что, когда обнаруживается, что исполняющийся поток выполнялся как работающий поток в течение периода времени, равного длительности, возвращаемой функцией **sched_rr_get_interval()** или дольше, этот поток становится в конец своего списка потоков, голова этого списка потоков делается выполняющимся потоком и удаляется из списка.

Результатом политики **SCHED_RR** является гарантия того, чтобы при наличии нескольких потоков с одинаковым приоритетом, какая-нибудь из них не монополизирует процессор.

Поток, который вытесняется и впоследствии возобновляет выполнение в соответствии с данной политикой, завершает неистекшую часть своего временного интервала, выделенного в рамках стратегии планирования **RR**.

Приоритетный диапазон — не менее 32 приоритетов.

Shortest Job First (SJF) — это алгоритм, в котором процесс с наименьшим временем выполнения выби-рается для следующего выполнения. Этот способ планирования может быть упреждающим или не упреждающим. Это значительно сокращает среднее время ожидания для других процессов, ожидающих выполнения. Полная форма SJF — самая короткая работа в первую очередь.

SCHED_SPORADIC

Политика спорадического обслуживания основана, в основном, на двух временных параметрах:
- период пополнения[1] (replenishment period);
- доступный объем исполнения (execution capacity).

Политика спорадического обслуживания идентична политике **SCHED_FIFO** с некоторыми дополни тельными условиями, которые вызывают переключение назначенного приоритета потока между значениями, задаваемыми элементами **sched_priority** и **sched_ss_low_priority** в структуре **sched_param**, в зависимости от вышеназванных временных параметров.

SCHED_OTHER

Соответствующие стандарту реализации включают еще одну политику планирования, идентифицируемую как **SCHED_OTHER** (которая может работать идентично политике планирования **FIFO** или RR).

Планирование потоков с политикой **SCHED_OTHER** в системе, в которой другие потоки выполняют· ся под **SCHED_FIFO**, **SCHED_RR** или **SCHED_SPORADIC**, определяется реализацией.

Эта политика предоставлена, чтобы позволить строго соответствующим стандарту приложениям иметь возможность переносимым образом указывать, что им больше не нужна политика планирования в реальном времени.

7) Алгоритмы планирования. Гарантированное планирование. Приоритетное планирование. Гарантированное планирование

При интерактивной работе N пользователей в вычислительной системе можно применить алгоритм планирования, который гарантирует, что каждый из пользователей будет иметь в своем распоряжении ~1/N часть процессорного времени. Пронумеруем всех пользователей от 1 до N. Для каждого пользова-теля с номером i введем две величины: Ti — время нахождения пользователя в системе или, другими словами, длительность сеанса его общения с машиной и — суммарное процессорное время уже выде-ленное всем его процессам в течение сеанса. Справедливым для пользователя было бы получение Ti/N процессорного времени. Если то i – й пользователь несправедливо обделен процессорным временем. Если же то система явно благоволит к пользователю с номером i. Вычислим для процессов каждого пользователя значение коэффициента справедливости и будем предо-ставлять очередной квант времени готовому процессу с наименьшей величиной этого отношения. Предложенный алгоритм называют алгоритмом гарантированного планирования. К недостаткам этого алгоритма можно отнести невозможность предугадать поведение пользователей. Если некоторый поль-зователь отправится на пару часов пообедать и поспать, не прерывая сеанса работы, то по возвращении его процессы будут получать неоправданно много процессорного времени.

Приоритетное планирование

Алгоритмы SJF и гарантированного планирования представляют собой частные случаи приоритетного планирования. При приоритетном планировании каждому процессу присваивается определенное чис-ловое значение — приоритет, в соответствии с которым ему выделяется процессор. Процессы с одинако-выми приоритетами планируются в порядке FCFS. Для алгоритма SJF в качестве такого приоритета вы-ступает оценка продолжительности следующего CPU burst. Чем меньше значение этой оценки, тем бо-лее высокий приоритет имеет процесс. Для алгоритма гарантированного планирования приоритетом служит вычисленный коэффициент справедливости. Чем он меньше, тем больше у процесса приоритет.

8) Алгоритмы планирования. Многоуровневые очереди (Multilevel Queue). Многоуровневые очереди с обратной связью (Multilevel Feedback Queue).

Многоуровневые очереди (Multilevel Queue)
В многоуровневых очередях процессы сортируются по группам. Организация очередей с помощью групп способствует повышению гибкости планирования. Для каждой группы процессов создается своя очередь процессов, находящихся в состоянии «готовность» Этим очередям приписываются фиксирован-ные приоритеты. Например, приоритет очереди системных процессов устанавливается выше, чем при-оритет очередей пользовательских процессов, приоритет очереди процессов, запущенных студентами, ниже, чем очереди процессов, запущенных преподавателями. Процессы с более низкими приоритетами не могут быть выбраны до тех пор, пока имеются высокоприоритетные процессы.
Многоуровневые очереди с обратной связью (Multilevel Feedback Queue)
В многоуровневых очередях с обратной связью процесс может переходить из одной очереди в другую в зависимости от своего поведения. Для этого нужны следующие данные:

- Количество очередей процессов, находящихся в состоянии готовность.
- Алгоритм планирования, действующий между очередями.
- Алгоритмы планирования, действующие внутри очередей.
- Правила помещения родившегося процесса в одну из очередей.
- Правила перевода процессов из одной очереди в другую.

+ Многоуровневые очереди с обратной связью представляют собой наиболее общий подход к планиро-ванию процессов. Они наиболее трудны в реализации, но в то же время обладают наибольшей гибко-стью

9) Логическая и физическая память. Основные функции ОС по управлению памятью. Простые схемы управления памятью.

Программа пользователя «видит» память не совсем так, как это обеспечивает аппаратура процессора (RAM/ROM), а лишь то, что обеспечивает ей ОС — абстракция виртуальной памяти. ОС предоставляет программе пользователя определенный интерфейс управления самым нижним слоем системной аб-стракции, называемой «виртуальное адресное пространство процесса». Традиционно «адресное про-странство процесса» определяется как «диапазон доступных процессу адресов памяти». Не утверждает-ся, что все эти адреса доступны безусловно — возможно, что для корректного доступа к каким-то из них требуется некая специальная процедура (выделение памяти).
В большинстве систем, использующих абстракцию виртуальной памяти, реализована страничная орга-низация памяти. В любой системе программы обращаются к набору адресов памяти. Эти адреса гене-рируются с помощью базовых регистров, индексной адресации, сегментных регистров или какими-то другими способами.
S*I + B + Disp -> EA

Стратегии управления страничной памятью

Стратегия выборки

Стратегия выборки определяет в какой момент следует переписать страницу из вторичной памяти в первичную. Существуют 2 основных варианта выборки:

- по запросу;
- с упреждением.

Алгоритм выборки по запросу

Вступает в действие в тот момент, когда процесс обращается к отсутствующей странице, содержимое которой находится на диске. Его реализация заключается в загрузке страницы с диска в свободную физическую страницу и коррекции соответствующей записи таблицы страниц.

Алгоритм выборки с упреждением

Осуществляет опережающее чтение, т.е кроме страницы, вызвавшей исключительную ситуацию в памяти (#PF), также загружается несколько страниц окружающих её. Такой алгоритм призван уменьшить накладные расходы, связанные с большим количеством исключительных ситуаций, возникших при работе со значительными объемами данных или кода. Кроме того, оптимизируется работа с диском.

Стратегия размещения

Стратегия размещения определяет, в какой участок первичной памяти поместить поступающую страницу.

В системе со страничной организацией она помещается в любой свободный страничный кадр.

В случае системы с сегментной организацией необходима стратегия, аналогичная стратегии с динамическим распределением.

Стратегия замещения

Стратегия замещения определяет, какую страницу надо вытолкнуть во внешнюю память, чтобы освободить место в ОП.

Стратегия позволяет хранить в памяти самую нужную информацию и тем самым снизить частоту страничных нарушений.

Замещение должно происходить с учетом выделенного каждому процессу количества страниц.

Кроме того, надо решить должна ли замещаемая страница принадлежать процессу, который вызвал замещение или она должна быть выбрана среди всех страниц основной памяти.

10) Кооперация (взаимодействие) процессов. Категории средств межпроцессного обмена информацией.

5 Лекция

Для достижения поставленной цели различные процессы (возможно, принадлежащие разным пользова-телям) могут исполняться псевдопараллельно на одной ВС или параллельно на разных ВС, взаимодей-ствуя между собой. Причины для их кооперации:

1) Повышение скорости работы.
2) Совместное использование данных.
3) Модульная конструкция какой-либо системы.
4) Просто для удобства. Пользователь может желать, например, редактировать и отлаживать программу одновременно. В этой ситуации процессы редактора и отладчика должны уметь взаимодействовать друг с другом.

Категории средств обмена информацией

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

1) Сигнальные (signal)

Передается минимальное количество информации — один бит, «да» или «нет».

Степень воздействия на поведение процесса, получившего информацию, минимальна — все зависит от того, знает ли он, что означает полученный сигнал, надо ли на него реагировать и каким образом.

Неправильная реакция на сигнал или его игнорирование могут привести к трагическим последствиям.
2) Канальные (pipe)
Общение процессов происходит через коммуникационные линии, предоставляемые операционной си-стемой.
3) Совместно используемая память (shared memory)
Два или более процессов могут совместно использовать некоторую область адресного пространства.
Созданием совместно используемой памяти занимается операционная система (по запросу процесса). Возможность обмена информацией максимальна, как, и влияние на поведение другого процесса, но требует повышенной осторожности.
Использование совместно используемой памяти для передачи/получения информации осуществляется с помощью средств обычных языков программирования, в то время как сигнальным и канальным сред-ствам коммуникации для этого необходимы специальные системные вызовы.
Совместно используемая память представляет собой наиболее быстрый способ взаимодействия процес-сов в одной вычислительной системе.

11) Кооперация (взаимодействие) процессов. Логическая организация механизма передачи информации.

5 Лекция
1) Установление связи
Можно ли использовать средство связи непосредственно для обмена информацией сразу после созда-ния процесса или первоначально необходимо предпринять некоторые действия по инициализации об-мена?
- Сигнальные

Для передачи сигнала от одного процесса к другому никакая инициализация не нужна.
- Канальные

Передача информации по линиям связи может потребовать первоначального резервированиятакой ли-нии для процессов, желающих обменяться информацией.
- Общая память

Для использования совместно используемой памяти различными процессами потребуется специальное обращение к операционной системе, которая выделит требуемую область адресного пространства.
2) Адресация
- Прямая адресация

В случае прямой адресации взаимодействующие процессы непосредственно общаются друг с другом, при каждой операции обмена данными явно указывая имя или номер процесса, которому информация предназначена или от которого она должна быть получена.
- Непрямая адресация

При непрямой адресации данные помещаются передающим процессом в некоторый промежуточный объект для хранения данных.
3) Информационная валентность процессов и средств связи
Слово валентность здесь использовано по аналогии с химией:
– сколько процессов может быть одновременно ассоциировано с конкретным средством связи?
– сколько таких средств связи может быть задействовано между двумя процессами?
4) Направленность связи
При однонаправленной связи каждый процесс, ассоциированный с ней, может использовать средство связи либо только для приема информации, либо только для ее передачи.
При двунаправленной связи каждый процесс, участвующий в общении, может использовать связь как для приема, так и для передачи данных.

12) Кооперация (взаимодействие) процессов. Особенности межпроцессной передачи информации. Бу-феризация, потоки и сообщения.

9 Лекция

Буферизация — возможность линии связи сохранять информацию, переданную одним процессом, до ее получения другим процессом или помещения в промежуточный объект. Здесь можно выделить три принципиальных варианта:

1) буфер нулевой емкости или отсутствует;

Никакая информация не может сохраняться на линии связи.

2) буфер ограниченной емкости;

Линия связи не может хранить до момента получения более чем некоторый фиксированный объем ин-формации, который определяется размером буфера.

3) буфер неограниченной емкости.

Теоретически это возможно, но практически вряд ли реализуемо. Процесс, посылающий информацию, никогда не ждет окончания ее передачи и приема другим процессом.

Существует две модели передачи данных по каналам связи — поток ввода-вывода и сообщения. Примерами потоковых каналов связи мо-гут служить pipe и FIFO.

Pipe – Представим себе, что в ВС есть некоторая труба, в один из концов которой процессы могут сли-вать информацию, а из другого принимать полученный поток. Информацией о расположении такой трубы в операционной системе обладает только один процесс — ее создавший. Этой информацией он может поделиться исключительно со своими наследниками — процессами-потомками и их потомками. Для связи между собой могут только родственные процессы, имеющие общего предка, создавшего дан-ный канал связи.

FIFO (именованный канал) – Если процесс, создавший канал, сообщит о ее точном расположении в си-стеме другим процессам, сделав вход и выход трубы каким-либо образом видимыми для всех осталь-ных, например, зарегистрировав ее в операционной системе под определенным именем, получится объект, который принято называть FIFO или именованным каналом. Именованный канал может исполь-зоваться для связи между любыми процессами в системе.

Сообщения

В модели сообщений процессы налагают на передаваемые данные некоторую структуру. Все сообщения могут иметь одинаковый фиксированный размер или могут быть переменной длины. В вычислительных системах используются разнообразные средства связи для передачи сообщений:

- очереди сообщений;
- sockets (гнезда или сокеты);
- и т. д.

13) Нити исполнения. Многопоточность

14 Лекция

Нити процесса совместно используют его программный код, глобальные и переменные и системные ресурсы, но каждая нить имеет свой собственный программный счетчик, свое содержимое регистров и свой собственный стек. Теперь процесс представляется как совокупность взаимодействующих нитей и выделенных ему ресурсов.Многопоточность — свойство платформы (например, операционной системы, виртуальной машины или приложения, состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся «параллельно», то есть без предписанного порядка во времени. При выполнении некоторых задач такое разделение может достичь более эффективного использования ресурсов вычислительной машины. В POSIX используются вызовы:

Вызовы, связанные с потоком	Описание
`pthread_create()`	Создание нового потока
`pthread_exit()`	Завершение работы вызвавшего потока
`pthread_join()`	Ожидание выхода из указанного потока
`pthread_yield()`	Освобождение центрального процессора, позволяющее выполняться другому потоку
`pthread_attr_init()`	Создание и инициализация структуры атрибутов потока
`pthread_attr_destroy()`	Удаление структуры атрибутов потока

14) Синхронизация активностей. Достаточные условия Бернстайна

8 Лекция
Активность (сущ.) — последовательное выполнение некоторых действий, направленных на достижение определенной цели.
Активности могут иметь место в программном и техническом обеспечении, в обычной деятельности людей и животных. Активности можно разбить на некоторые неделимые или атомарные операции.
Пусть у нас есть две активности P и Q, состоящие из двух атомарных операций каждая:
P: x = 2
y = x - 1
Q: x = 3
y = x + 1
В результате их псевдопараллельного выполнения, если переменные x и y являются общими, для актив-ностей, возможны шесть вариантов чередования и четыре разных результата для пары (x, y): (3, 4), (2, 1), (2, 3) и (3, 2). Говорят, что набор активностей (например, программ) детерминирован, если всякий раз при псевдопараллельном исполнении для одного и того же набора входных данных он дает одина-ковые выходные данные. В противном случае он недетерминирован.
Выше приведен пример недетерминированного набора программ.
Условие Брейнсайна: Имеем R/read(P), W/write(P), R(Q), W(Q). Например, для программы
P: x = u + v
y = x * w
R(P) = {u, v, x, w}, W(P) = {x, y}.
Следует заметить, что переменная x присутствует как в R(P), так и в W(P).
Итак, условия Бернстайна.
Если для двух данных активностей P и Q:
1) пересечение W(P) и W(Q) пусто, (P и Q записывают разные данные)
2) пересечение W(P) с R(Q) пусто, (Q не читает того, что записывает P)
3) пересечение R(P) и W(Q) пусто, (P не читает того, что записывает Q)
тогда выполнение P и Q детерминировано. Не соблюдены — не известно.

15) Взаимное исключение и критическая секция.

8 Лекция
Если не важна очередность доступа к совместно используемым данным, задачу упорядоченного к ним доступа (устранение race condition) можно решить, если обеспечить каждому процессу эксклюзивное право доступа к этим данным.
Каждый процесс, обращающийся к совместно используемым ресурсам, исключает для всех других процессов возможность общения с этими ресурсами одновременно с ним, если это может при-вести к недетерминированному поведению набора процессов.
Такой прием называется взаимоисключением (mutual exclusion).

Если же для получения правильных результатов очередность доступа к совместно используемым ресур-сам важна, то одними взаимоисключениями уже не обойтись.
Критическая секция — это часть программы, исполнение которой может привести к возникновению состояния гонок.
Чтобы исключить эффект гонок по отношению к некоторому ресурсу, необходимо организовать работу так, чтобы в каждый момент времени только один процесс мог находиться в своей критической секции, связанной с этим ресурсом.
Иными словами, необходимо обеспечить реализацию взаимоисключения для критических секций про-грамм. (Пример про пиво).

16) Пять требований к алгоритмам организации взаимодействия процессов.

8 Лекция
1) Задача должна быть решена чисто программным способом на обычной машине, не имеющей специ-альных команд взаимоисключения (на самом деле сегодня такую машину найти трудно).
При этом предполагается, что основные инструкции языка программирования (такие примитивные ин-струкции как load, store, test) являются атомарными операциями.
2) Не должно существовать никаких предположений об относительных скоростях выполняющихся про-цессов или количестве процессоров, на которых они исполняются.
3) Если процесс исполняется в своей критической секции, то не существует никаких других процессов, которые исполняются в своих соответствующих критических секциях.
Это условие получило название условия взаимоисключения (mutual exclusion).
4) Процессы, которые находятся вне своих критических секций и не собираются входить в них, не могут препятствовать другим процессам входить в их собственные критические секции.
Если нет процессов в критических секциях, и имеются процессы, желающие войти в них, то только те процессы, которые не исполняются в регулярной секции (regular section), должны принимать решение о том, какой процесс войдет в свою критическую секцию.
Такое решение не должно приниматься бесконечно долго. Это условие получило название условия прогресса (progress).
5) Не должно возникать бесконечного ожидания для входа процесса в свою критическую секцию. От того момента, когда процесс запросил разрешение на вход в критическую секцию, и до того момента, когда он это разрешение получил, другие процессы могут пройти через свои критические секции лишь ограниченное число раз.
Это условие получило название условия ограниченного ожидания (bound waiting).

17) Алгоритмы организации взаимодействия процессов. Запрет прерываний, переменная-замок, строгое чередование и флаги готовности.

8 Лекция
Запрет прерываний
```
regular_section
    interrupts_disable — запретить все прерывания / CLI
        critical_section
    interrupts_enable — разрешить все прерывания / STI
regular_section
```

+: внутри критической секции никто не может вмешаться в его работу.
−: разрешает процессу пользователя разрешать и запрещать прерывания во всей вычислительной систе-ме. (Запретил прерывания и не возвратил их)
Переменная-замок (lock)
Процесс может войти в критическую секцию только тогда, когда значение переменной-замка равно 0, одновременно изменяя ее значение на 1 — закрывая замок.

При выходе из критической секции процесс сбрасывает ее значение в 0 — замок открывается.
```
shared int lock = 0; // свободно/занято
    regular section
    while(lock); lock = 1; // не атомарное действие
        critical section
        lock = 0;
    regular section
```
–: сожалению, такое решение, не удовлетворяет условию взаимоисключения, так как действие while(lock); lock = 1; не является атомарным.
Строгое чередование
```
// turn == 0, в КС входит P0, если turn == 1, в КС входит P1.
shared int turn = 0;          // чья очередь (turn — чья сейчас очередь исполнения)
    regular section
    while(turn != i);    // ждем, пока второй не выйдет
        critical section
        turn = 1–i;
    regular section
```
+: взаимоисключение гарантируется и процессы входят в критическую секцию строго по очереди: P0, P1, P0, P1, ...
–: процессы, которые находятся вне своих критических участков и не собираются входить в них, не должны препятствовать другим процессам входить в их собственные критические участки.
Флаги готовности
Пусть два наши процесса имеют совместно используемый массив флагов готовности входа процессов в критический участок shared int ready[2] = {0, 0};
Когда i–й процесс готов войти в КС, он присваивает своему элементу массива ready[i] значение равное 1. После выхода из КС он сбрасывает это значение в 0. Процесс не входит в критическую секцию, если другой процесс уже готов ко входу в критическую секцию или находится в ней.
```
...
ready[i] = 1;
while(ready[1–i]);
    critical_section
ready[i] = 0;
...
```
+: обеспечивает взаимоисключение, позволяет процессу, готовому к входу в критический участок, войти в него сразу после завершения эпилога в другом процессе
–: все равно нарушает условие прогресса.
Пусть процессы практически одновременно подошли к выполнению пролога. После выполнения при–сваивания ready[0] = 1 планировщик передал процессор от процесса 0 процессу 1, который также вы–полнил присваивание ready[1] = 1. После этого оба процесса бесконечно долго ждут друг друга на входе в критическую секцию. Возникает ситуация, которую принято называть тупиковой (deadlock).

18) Алгоритмы организации взаимодействия процессов. Алгоритмы Петерсона и регистратуры.

8 Лекция
Алгоритм Петерсона
Это решение удовлетворяет всем пяти требованиям к алгоритмам синхронизации.
Пусть оба процесса имеют доступ к массиву флагов готовности и к переменной очередности.
```
Shared int ready[2] = {0, 0};
shared int turn;
...
ready[i] = 1;        // (1)
```

```
turn = 1-i;          // (2)
while(ready[1-i] && turn == 1-i);
    critical section
ready[i] = 0;
...
```

При исполнении пролога критической секции процесс заявляет о своей готовности выполнить критиче-ский участок (1) и одновременно предлагает другому процессу приступить к его выполнению (2).
Если оба процесса подошли к прологу практически одновременно, то они оба объявят о своей готовно-сти и предложат выполняться друг другу.
При этом одно из предложений всегда последует после другого. Тем самым работу в критическом участке продолжит процесс, которому было сделано последнее предложение.
Все пять требований данным алгоритмом удовлетворяются. Алгоритм работает только для 2 процессов!

Алгоритм регистратуры (Bakery algorithm)
Соответствующий алгоритм (как и Петерсона) для n взаимодействующих процессов получил название алгоритм регистратуры (булочной).
Каждый вновь прибывающий клиент (процесс) получает талончик на обслуживание с номером.
Клиент с наименьшим номером на талончике обслуживается следующим. Из-за неатомарности опера-ции вычисления следующего номера не гарантируется, что у всех процессов будут талончики с разны-ми номерами. В случае равенства номеров на талончиках у двух или более клиентов первым обслужи-вается клиент с меньшим значением имени (имена можно сравнивать в лексикографическом порядке). Совместно используемые структуры данных для алгоритма — два массива
shared enum {false, true} choosing[n]; // получает талончик
shared int number[n]; // номер талончика
Изначально элементы этих массивов инициируются значениями false и 0 соответственно.
Введем следующие обозначения:
1) (a,b) < (c,d), если a < c или если a = c и b < d
2) sup(a0, a1, ..., an)— это число k такое, что k >= ai и для всех i = 0, ..., n
Структура алгоритма для процесса P[k]

```
shared enum {false, true} choosing[n]; // получает талончик
shared int number[n];   // номер талончика
do {
    choosing[i] = true;
    number[i] = sup(number[0], number[1], ..., number[n - 1]) + 1; (автоинкремент)
    choosing[i] = false;

    for (j = 0; j < n; j++) {        // ждем, пока не подойдет очередь
        while (choosing[j]);         // ждем, пока j получает талончик
        while ((number[j] != 0) &&  // ждем если j в очереди и
        (number[j],j)<(number[i],i));  // очередь I (наша) после j
    }
    critical section
    number[i] = 0;
    remainder section
} while (1);
```

19) Аппаратная поддержка взаимоисключений. Команды test_and_set и swap.

Многие вычислительные системы помимо комманд CLI STI имеют специальные команды процессора, которые позволяют проверить и изменить значение машинного слова или поменять местами значения двух машинных слов в памяти, выполняя эти действия как атомарные операции. Это и называется аппа-ратной поддержкой взаимоисключений

Команда Test-and-Set (проверить и установить в 1)
О выполнении команды Test-and-Set, осуществляющей проверку значения логической переменной с од-новременной установкой ее значения в 1, можно думать, как о выполнении функции
```
int Test_and_Set(int *target) {
    int tmp = *target;
    *target = 1;
    return tmp;
}
```
С использованием этой атомарной команды мы можем модифицировать алгоритм для переменной-замка, так чтобы он обеспечивал взаимоисключения
```
shared int lock = 0;
...
while(Test_and_set(&lock));
    critical section
lock = 0;
...
```
Минусы: алгоритм не удовлетворяет условию ограниченного ожидания.
Процесс может ждать неограниченно долго, пока переменная-замок освободится.

Команда Swap (Обменять значения)
Выполнение команды Swap, обменивающей два значения, находящихся в памяти, можно проиллюстрировать следующей функцией
```
void Swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```
Применяя атомарную команду Swap, мы можем реализовать предыдущий алгоритм, введя дополни-тельную логическую переменную key локальную для каждого процесса:
```
shared int lock = 0;
int key;
...
key = 1;
do
    Swap(&lock, &key);
while (key);
    critical section
lock = 0;
...
```

20) Семафоры и мьютексы. Задача «производитель-потребитель».

8 Лекция
Семафор (semaphore) — объект, ограничивающий количество процессов/потоков, которые могут войти в заданный участок кода. Определение введено Эдсгером Дейкстрой.
Семафоры используются для синхронизации и защиты передачи данных через совместно использу-емую память, а также для синхронизации работы процессов и потоков.

Семафор представляет собой целую переменную, принимающую неотрицательные значения, доступ любого процесса к которой, за исключением момента ее инициализации, может осуществляться только через две атомарные операции:
Операция P(S) или down(s) выясняет, отличается ли значение семафора S от 0. Если отличается, она уменьшает это значение на 1 и выполнение продолжается (входит в критическую секцию).
V(S) — увеличивает значение семафора на единицу.
Одной из типовых задач, требующих организации взаимодействия процессов, является задача producer-consumer (производитель-потребитель).
Пусть два процесса обмениваются информацией через буфер ограниченного размера.
Производитель закладывает информацию в буфер, а потребитель извлекает ее оттуда

```
Producer                Consumer
while(1) {              while(1) {
    produce_item;               get_item;
    put_item;               consume_item;
}                       }
```

Если буфер заполнен, то производитель должен ждать, пока в нем появится место, чтобы положить туда новую порцию информации.
Если буфер пуст, то потребитель должен дожидаться, пока в буфере не появится сообщение.

Задача решается с помощью семафоров:

```
Semaphore mutex = 1;
Semaphore free_space = N; // емкость буфера;
Semaphore items = 0; // пусто
Producer            Consumer
while(1) {          while(1) {
    produce_item;           down(items);
    down(free_space);       down(mutex);
    down(mutex);            get_item;
    put_item;           up(mutex);
    up(mutex);              up(free_space);
    up(items);          consume_item;
}                   }
```

Мьютекс (mutex, mutual exclusion — «взаимное исключение») — аналог одноместного семафора, слу-жащий для синхронизации одновременно выполняющихся потоков.
Мьютекс отличается от семафора тем, что только владеющий им поток может его освободить.
Мьютексы — это один из вариантов семафорных механизмов для организации взаимного исключения. Мьютексы могут находиться в одном из двух состояний — отмеченном или неотмеченном (открыт и закрыт соответственно).

21) Мониторы. Задача «производитель-потребитель».

8 Лекция
Монитор — высокоуровневый механизм взаимодействия и синхронизации процессов, обеспечивающий доступ к общим ресурсам (аппаратура или набор переменных).
Компилятор или интерпретатор прозрачно для программиста вставляет код блокировки-разблокировки в оформленные соответствующим образом процедуры, избавляя программиста от явного обращения к примитивам синхронизации.
Монитор — это специальный объектный тип данных, состоящий из:
- переменных, связанных с общим ресурсом;
- набора процедур, взаимодействующих с этим ресурсом (функции-методы);
- мьютекса;

– инварианта, определяющего условия, позволяющие избежать состояние гонки.
На абстрактном уровне можно описать структуру монитора следующим образом:
monitor monitor_name {
variable declarations; // переменные состояния монитора
void m1(...) {...}
...
void mn(...) {...}
{конструктор -- блок инициализации внутренних переменных;}
}
Мониторы и задача производителя-потребителя

```
monitor Producer_Consumer {
  condition free_space, items;
  int count;

  void put() {
    if(count == N) free_space.wait;
    put_item;
    count += 1;
    if(count == 1) items.signal;
  }
  void get() {
    if (count == 0) items.wait;
    get_item();
    count -= 1;
    if(count == N-1) free_space.signal;
  }
  {count = 0;}
}
```

```
Producer:
while(1) {
  produce_item;
  ProducerConsumer.put();
}

Consumer:
while(1) {
  ProducerConsumer.get();
  consume_item;
}
```

22) Сообщения. Задача «производитель-потребитель»

8 Лекция
Для прямой и непрямой адресации достаточно двух примитивов, чтобы описать передачу сообщений по линии связи — send и receive.
В случае прямой адресации их обозначают обычно так:
send(P, message) — послать сообщение message процессу P;
receive(Q, message) — получить сообщение message от процесса Q.
В случае непрямой адресации их обозначают обычно так:
send(A, message) — послать сообщение message в почтовый ящик A;
receive(A, message) — получить сообщение message из почтового ящика A.
Примитивы send и receive уже имеют скрытый механизм взаимоисключения. Более того, в большинстве систем они уже имеют и скрытый механизм блокировки при чтении из пустого буфера и при записи в полностью заполненный буфер.
Реализация решения задачи producer-consumer для таких примитивов становится неприлично тривиальной. (Producer отправляет сообщения, consumer получает).

23) Файловый ввод/вывод. Файлы и каталоги.

4 Лекция
Файл (Именованный файл) — некий набор хранимых на внешнем носителе данных, имеющий имя (набор данных — data set).
Каталоги — это файлы специального типа. Они хранят имена и номера индексных дескрипторов. Все, чем отличается каталог от обычного файла на низком уровне — это значение признака типа в индекс-ном дескрипторе.
UNIX поддерживает файлы специального типа:
- файлы байт-ориентированных устройств;
- файлы блок-ориентированных устройств;

– имена сокетов;
– именованные каналы (FIFO);
– ссылки.
ИНДЕКСНЫЙ ДЕСКРИПТОР — хранимая на внешнем ЗУ (диске) структура данных, содержащая всю информацию о файле, исключая его имя.
Индексный дескриптор содержит:
– номер (уникальный в рамках файловой системы данного диска);
– тип файла;
– права доступа к файлу;
– количество связей (ссылок на файл в каталогах) файла;
– идентификатор пользователя и группы–владельца;
– размер файла в байтах;
– время последнего доступа к файлу;
– время последнего изменения файла;
– время последнего изменения индексного дескриптора файла;
– указатели на блоки данных файла (обычно 10);
– указатели на косвенные блоки (обычно 3).
С точки зрения программиста, файл — это объект, в который можно писать, читать или и то и другое.
Для открытия файла используется вызов open(const char *pathname, int flags). Он получает дескриптор файла (НЕ ИНДЕКСНЫЙ ДЕСКРИПТОР, а объект данный ядра ОС). В последующем этот дескриптор используется для операций чтения (read) и записи (write).
read(int fd, void *buf, size_t count); — читает «count» байт из файла. Если количество прочитанных бай–тов меньше, чем количество запрошенных, то это не считается ошибкой — данные, например, могли быть почти в конце файла, в канале, на терминале, или read() был прерван сигналом.
write(int fd, const void *buf, size_t count) — записывает count байт в файл.
close(int fd) — просто закрывает файл.

24) Файловая система ОС UNIX. Типы файлов и права доступа

4 Лекция
Файловая система UNIX:
– Единое древо файлов
– В имя файла не входит имя устройства, на котором файл находится
– Если в системе несколько устройств прямого доступа — один из них становится root–ом и все осталь–ные монтируются (mount) к нему.
– для указания полного пути к файлу на примонтированном устройстве спереди к имени файла добав–ляется полный путь к точке монтирования. Если на /media/floppy есть каталог work с файлов foo.c, то он ищется как /media/floppy/work/foo.c
– Каталоги хранят только имя файла и его индексный дескриптор.

Индексный дескриптор — хранимая часть на диске, которая содержит всю информацию о файле, без его имени. Размер — обычно 128 байт.
Индексный дескриптор содержит:
– номер (уникальный в рамках файловой системы данного диска);
– тип файла;
– права доступа к файлу;
– количество связей (ссылок на файл в каталогах) файла;
– идентификатор пользователя и группы–владельца;
– размер файла в байтах;
– время последнего доступа к файлу;
– время последнего изменения файла;

– время последнего изменения индексного дескриптора файла;
– указатели на блоки данных файла (обычно 10);
– указатели на косвенные блоки (обычно 3).
Типы файлов:
– Обычный файл
– Каталог
Специальные типы:
– файлы байт–ориентированных устройств;
– файлы блок–ориентированных устройств;
– имена сокетов;
– именованные каналы (FIFO);
– символические ссылки.
Жёсткая ссылка указывает на индексный дескриптор файла, символьная на его полный путь.

Восьмеричное значение	Вид в столбце прав доступа	Право или назначение бита
4000	`---s------`	Установленный эффективный идентификатор владельца (бит SUID)
2000	`------s---`	Установленный эффективный идентификатор группы (бит SGID)
1000	`---------t` `---------T`	*Клейкий* (sticky) бит. Вид для каталогов и выполняемых файлов, соответственно.
0400	`-r--------`	Право владельца на чтение
0200	`--w-------`	Право владельца на запись
0100	`---x------`	Право владельца на выполнение
0040	`----r-----`	Право группы на чтение
0020	`-----w----`	Право группы на запись
0010	`------x---`	Право группы на выполнение
0004	`-------r--`	Право всех прочих на чтение
0002	`--------w-`	Право всех прочих на запись
0001	`---------x`	Право всех прочих на выполнение

Для расчета прав доступа необходимо сложить восьмеричные значения всех необходимых установлен–ных битов.
Например:
Чтение для владельца: 0400
Запись для владельца: 0200
Выполнение для владельца: 0100
Чтение для группы: 0040
Выполнение для группы: 0010
Выполнение для прочих: 0001
Сумма: 0751

25) Структура ФС ОС UNIX. Суперблок и индексные дескрипторы.

4 Лекция
Физическая файловая система UNIX занимает раздел диска и состоит из следующих основных компо–нентов:
– Суперблок (superblock). Содержит общую информацию о файловой системе.
– Массив индексных дескрипторов (ilist).
Содержит метаданные всех файлов файловой системы.
Индексный дескриптор (inode) содержит информацию о статусе файла и указывает на расположение данных этого файла. Ядро обращается к индексному дескриптору по индексу в массиве. Размер массива индексных дескрипторов является фиксированным и задается при создании физической файловой си–стемы.
– Блоки хранения данных.

Данные обычных файлов и каталогов хранятся в блоках. Обработка файла осуществляется через ин-дексный дескриптор, содержащий ссылки на блоки данных.
Суперблок содержит:
— тип файловой системы;
— размер файловой системы в логических блоках, включая сам суперблок, массив индексных дескрипторов и блоки хранения данных;
— размер массива индексных дескрипторов;
— количество свободных блоков;
— количество свободных индексных дескрипторов;
— флаги;
— размер логического блока файловой системы (512, 1024, 2048, 4096, 8192).
— список номеров свободных индексных дескрипторов;
— список адресов свободных блоков.

Индексный дескриптор, или inode, содержит информацию о файле, необходимую для обработки дан-ных, т.е. метаданные файла. Каждый файл ассоциирован с одним индексным дескриптором.
Индексный дескриптор содержит:
— номер;
— тип файла;
— права доступа к файлу;
— количество связей (ссылок на файл в каталогах) файла;
— идентификатор пользователя и группы-владельца;
— размер файла в байтах;
— время последнего доступа к файлу;
— время последнего изменения файла;
— время последнего изменения индексного дескриптора файла;
— указатели на блоки данных файла (обычно 10);
— указатели на косвенные блоки (обычно 3).

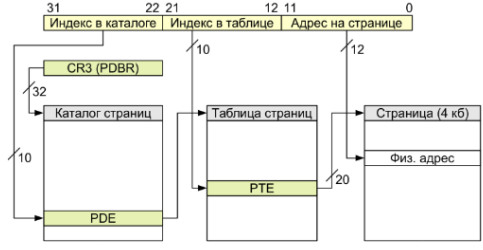
26) Виртуальная память. Страничная организация памяти.

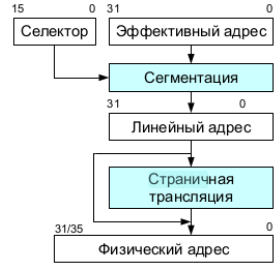
23 Лекция
ОС предоставляет программе пользователя определенный интерфейс управления самым нижним слоем системной абстракции, называемой «виртуальное адресное пространство процесса».
Традиционно «адресное пространство процесса» определяется как «диапазон доступных процессу ад-ресов памяти». Не утверждается, что все эти адреса доступны безусловно — возможно, что для кор-ректного доступа к каким-то из них требуется некая специальная процедура (выделение памяти). В большинстве систем, использующих абстракцию виртуальной памяти, реализована страничная органи-зация памяти. В любой системе программы обращаются к набору адресов памяти. Эти адреса генери-руются с помощью базовых регистров, индексной адресации, сегментных регистров или какими-то дру-гими способами.

S*I + B + Disp -> EA

Например, архитектура x86-64 поддерживает следующие размеры для страниц — 4 Кбайт, 2 Мбайт и 1 Гбайт[2]. Пользовательские приложения обычно используют страницы размером 4 Кбайт. Ядро системы обычно использует одну большую страницу размером 1 Гбайт.

При каждом обращении к памяти виртуальный адрес делится на три части:





Первая часть — индекс (PDE — Page Directory Entry) элемента в каталоге страниц (Page Directory). Из этого элемента извлекается физический адрес таблицы страниц (Page Table).

Вторая часть — индекс (PTE — Page Table Entry) в таблице страниц. Из этого элемента извлекается физический адрес страницы.

Третья часть интерпретируется как смещение в этой странице.

27) Синхронизация структуры ФС. Журнали[зи]руемые ФС.

В System V.4 появились средства, позволяющие процессам синхронизировать параллельный доступ к файлам. Было бы логично связать синхронизацию доступа к файлу как к единому целому с системным вызовом open (т.е., например, открытие файла в режиме записи или обновления могло бы означать его монопольную блокировку соответствующим процессом, а открытие в режиме чтения — совместную блокировку). Так поступают во многих операционных системах (начиная с ОС Multics). Однако, по от-ношению к ОС UNIX такое решение принимать было слишком поздно, поскольку многочисленные со-зданные за время существования системы прикладные программы опирались на свойство отсутствия автоматической синхронизации.

Поэтому разработчикам пришлось пойти «обходным путем». Ядро ОС UNIX поддерживает дополни-тельный системный вызов fcntl, обеспечивающий такие вспомогательные функции, относящиеся к фай-ловой системе, как получение информации о текущем режиме открытия файла, изменение текущего режима открытия и т.д. В System V.4 именно на системный вызов fcntl нагружены функции синхрони-зации.

Цель создания журналируемых файловых систем заключается в том, чтобы обеспечить быстрое восста-новление системы после сбоев (например, после потери питания).

В системе произошёл сбой. Восстанавливать программой очень долго, особенно если это сервер (и он в это время не будет доступен!!), и нету гарантии, что всё восстановится нормально. Для этого использу-ются журналируемые ФС. В них если действие не завершено — данные не будут записаны на диск. Чтобы сбои не приводили к необратимым последствиям, все действия протоколируются. Если сбой всё-таки происходит, то по этому протоколу система возвращается в безошибочное состояние.

28) Файлы устройств и операции над устройствами

4 Лекция

ОС UNIX обобщает концепцию файла как универсальную абстракцию.

Практически любое внешнее устройство может быть представлено на пользовательском уровне, как файл специального типа. Это справедливо для жестких дисков, всевозможных параллельных и последо-вательных портов и виртуальных терминалов.

Устройства, которые могут быть представлены в виде файлов, делятся на два типа — символьные (или потоковые) и блочные. Иногда используются термины «байт-ориентированные» и «блок-ориентированные» устройства.
Основными операциями над байт-ориентированными устройствами являются запись и чтение одного (очередного) символа (байта).
Блок-ориентированные устройства воспринимаются как хранилища данных, разделенных на блоки фиксированного размера. Основными операциями, соответственно, являются чтение и запись заданного блока.
Файлы устройств могут быть открыты на чтение и запись. В результате, используя полученный де-скриптор, можно писать на устройство и читать с него.
Блок-ориентированные устройства поддерживают позиционирование с помощью вызова lseek().
Байт-ориентированные устройства операцию позиционирования не поддерживают.
Это основное отличие устройств одного типа от другого.
Над устройствами определены дополнительные операции, специфические для конкретного устройства.
Все операции данной категории выполняются с помощью системного вызова управления устройствами — ioctl()

29) Очереди сообщений UNIX

9 Лекция
Очереди сообщений, как и семафоры, и разделяемая память, являются средством связи с непрямой ад-ресацией, требуют инициализации для организации взаимодействия процессов и специальных дей-ствий для освобождения системных ресурсов по окончании взаимодействия.
Пространством имен оче-редей сообщений является то же самое множество значений ключа, генерируемых с помощью функции ftok().
Очереди сообщений располагаются в адресном пространстве ядра операционной системы в виде одно-направленных списков и имеют ограничение по объему информации, хранящейся в каждой очереди. Каждый элемент списка представляет собой отдельное сообщение.
Сообщения имеют атрибут, называ-емый типом сообщения . Выборка сообщений из очереди может осуществляться тремя способами.
1. В порядке FIFO, независимо от типа сообщения.
2. В порядке FIFO для сообщений конкретного типа.
3. Первым выбирается сообщение с минимальным типом, не превышающим некоторого заданного значения, пришедшее раньше других сообщений с тем же типом.

Для управления такими очередями используются вызовы msgget, msgset и msgrcv. С помощью них можно создавать очереди, отправлять и получать сообщения.

30) Семафоры ОС UNIX

9 Лекция
Семафоры POSIX предоставляют более простой и продуманный интерфейс чем семафоры System V, с другой стороны, семафоры POSIX не так широко распространены (особенно в старых си-стемах), по сравнению с семафорами System V.
Семафоры POSIX позволяют синхронизировать свою работу как процессам, так и нитям.
Семафор представляет собой целое число, значение которого никогда не будет меньше нуля.
Над семафорами выполняются две операции — увеличение значения семафора на единицу.
Есть два вида семафоров — именованные и безымянные.
Именованные семафоры отличают по именам вида /имя — строка (с null в конце) длиной до NAME_MAX-4 (т. Е., 251) символов, состоящая из начальной косой черты и одного или нескольких символов (символ косой черты не допускается).

Безымянные семафоры не имеют имени. Семафор размещается в области памяти, которая доступна не-скольким нитям (общий семафор для нитей) или процессам (общий семафор для процессов).
Общий семафор для нитей размещается в области памяти, которая доступна из нитей процесса, напри-мер в глобальной переменной.
Операции:
sem_post(3) и уменьшение значения семафора на единицу sem_wait(3).
Если значение семафора равно нулю, то операция sem_wait(3) блокирует работу до тех пор, пока значение не станет больше нуля.
Sem_init(sem_t *sem, int pshared, unsigned int value) — инициализирует безымянный семафор.
ЕСЛИ СЕМАФОР УЖЕ БЫЛ СОЗДАН — БУДЕТ БОЛЬНО.
Sem_destroy(sem_t *sem) — уничтожает безымянный семафор
sem_open(const char *name, int oflag) — инициализирует и открывает именованный семафор
sem_wait(sem_t *sem), sem_trywait(sem_t *sem), sem_timedwait(sem_t *sem, const struct timespec *abs_timeout) — блокирует семафор
sem_post(sem_t *sem) — разблокирует семафор
sem_close(sem_t *sem) — закрывает именованный семафор
sem_unlink(const char *name) — удаляет именованный семафор
sem_getvalue(sem_t *sem, int *sval) — возвращает значение семафора

31) Управление открытыми файлами: возможности системного вызова fcntl()

11 Лекция
Для начала следует открыть файл вызовом int open(const char *pathname, nt flags, mode_t mode) или вы-зовом int creat(const char *pathname, mode_t mode). Эти вызовы преобразуют имена файлов в дескрип-торы ФАЙЛА (НЕ ИНДЕКСНЫЕ). Дескрипторы с номерами 0, 1, 2 связаны с потоками ввода-вывода-ошибок.
Команда fcntl(int fd, int cmd, ...) — манипуляции с файловым дескриптором. Может принимать третий аргумент. Необходимость его значения зависит от значения cmd.
Системный вызов fcntl(2) позволяет выполнять следующие операции:
– создание дубликата файлового дескриптора; команда F_DUPFD с аргументом меньше retfd
ищет наименьший доступный номер файлового дескриптора, который больше retfd и делает его копией дескриптора fd.

– управление флагами файлового дескриптора; команды FD_CLOEXEC, F_GETFD, F_SETFD
С файловым дескриптором в настоящее время определён только один флаг FD_CLOEXEC — флаг close-on-exec. Он определяет что произойдет с дескриптором при вызове exec().
– управление флагами состояния файла; команды F_GETFL, F_SETFL
Каждый дескриптор открытого файла имеет несколько связанных с ним флагов состояния, которые инициализируются вызовом open(2).
– совместная/рекомендательная (advisory) блокировка;
Если на сегменте файла установлена совместная блокировка, другие процессы на данном сегменте или его части могут тоже устанавливать совместные блокировки. Исключительная препятствует другому процессу устанавливать блокировки на заблокированную часть.
– обязательная (mandatory) блокировка; F_SETLK, F_GETLK
Обязательные блокировки влияют на все процессы.
– управление сигналами; F_SETSIF, F_GETSIG
Установить сигнал, который будет послан, когда станет возможен ввод или вывод, равным значению, указанному в arg.
– аренда (лизинг, владение); F_SET(GET)LEASE, F_RDLCK, F_WRLCK, F_UNLCK

Предоставляет механизм, посредством которого процесс, который удерживает аренду («арендатор»), уведомляется сигналом, если процесс («нарушитель аренды») пытается выполнить вызов open(2) или truncate(2) для этого файла/дескриптора.
– уведомления об изменении файла и каталога; F_NOTIFY
Уведомлять при изменении каталога, на который указывает fd или когда изменились файлы, которые в нём содержатся.
– изменение ёмкости канала. F_SETPIPE_SZ
Изменяет ёмкость канала fd

32) Управление открытыми файлами: блокировки файлов и областей (fcntl).

11 Лекция
С помощью вызова fcntl можно заблокировать часть файла или весь файл. Для этого применяются ко-манды F_GETLCK, F_SETLCK, F_SETCLKW.
Блокировки могут быть совместными и исключительными. Совместная блокировка позволяет другим процессам блокировать область файла, которая уже заблокирована. Исключительная не позволяет.
F_GETLK (struct flock *lock) — получить информацию о возможности установить блокировку. При вы-зове параметр lock должен описывать блокировку, которую мы хотели бы установить на файл. Возвра-щенная информация может уже быть устаревшей к моменту ее проверки и использования.
После успешного запроса F_GETLK, когда блокирующая блокировка обнаружена, значения, возвраща-емые в структуре flock, должны быть следующими:
l_type — тип обнаруженной блокирующей блокировки;
l_whence — SEEK_SET;
l_start — начало блокировки блокировки;
l_len — длина блокировки;
l_pid — ID процесса, удерживающего блокирующую блокировку.
F_SETLK (struct flock *) — устанавливает или снимает блокировку в соответствии с типом операции, указанным в поле
l_type третьего аргумента lock (F_RDLCK, F_WRLCK, F_UNLCK), на область байт, указанную полями l_whence, l_start и l_len там же (в структуре lock).
F_SETLKW (struct flock *) — эквивалент SETLCK, только если блокировки конфликтуют, то поток бу-дет ждать, пока запрос не будет удовлетворён.
Взаимная блокировка (deadlock)
Если процесс, управляющий заблокированной областью, переводится в состояние ожидания, пытаясь заблокировать заблокированную область другого процесса, возникает опасность взаимоблокировки (deadlock, тупик).

33) Флаги состояния файла и управление ими. Системные вызовы open() и fcntl().

11 Лекция
Каждый дескриптор открытого файла имеет несколько связанных с ним флагов состояния, которые инициализируются вызовом open(2).
Эти флаги совместно используются копиями файловых дескрипторов (сделанными с помощью dup(2), fcntl(F_DUPFD), fork(2) и т.д.), которые указывают на одно и то же описание открытого файла. Эти фла-ги состояния и их смысл описаны в open(2).
Некоторые из этих флагов могут быть изменены вызовом fcntl().
O_RDONLY -------режим доступа по чтению
O_WRONLY -------режим доступа по записи
O_RDWR -------режим доступа по чтению и записи
O_CREAT -------создается если не существует

O_EXCL ------- если файл существует и O_CREAT, open() вернет ошибку
O_NOCTTY ------- если fname -> терминал, он не станет управляющим
O_TRUNC ------- если файл существует, будет урезан до 0
O_APPEND ------- файл открывается в режиме добавления
O_NONBLOCK --- файл открывается в режиме non-blocking (FIFO)
O_SYNC ------- файл открывается в режиме синхронного ввода-вывода
O_NOFOLLOW --- если fname символьная ссылка, open() вернет ошибку
O_DIRECTORY --- если fname не является каталогом, open() вернет ошибку
O_LARGEFILE ---- позволяет открывать файлы, длина которых больше 31 бита

F_GETFL (void) — Получить права доступа к файлу и флаги состояния файла. Флаги состояния файла и режимы доступа к файлу связаны с описанием файла и не влияют на другие дескрипторы файлов, кото-рые относятся к одному и тому же файлу с разными описаниями открытых файлов.
F_SETFL (int arg)
Установить флаги состояния файла, определенные в <fcntl.h>, согласно значению, указанному в третьем аргументе, взятом как тип int.
Int open(const char *pathname, nt flags, mode_t mode) — преобразовывает имя файла в его ФАЙЛОВЫЙ ДЕСКРИПТОР (не индексный). 0, 1, 2 — дескрипторы потоков ввода, вывода, ошибок.
Команда fcntl(int fd, int cmd, ...) — манипуляции с файловым дескриптором. Может принимать третий аргумент. Необходимость его значения зависит от значения cmd.
Системный вызов fcntl(2) позволяет выполнять следующие операции:
- создание дубликата файлового дескриптора;
- управление флагами файлового дескриптора;
- управление флагами состояния файла;
- совместная/рекомендательная (advisory) блокировка;
- обязательная (mandatory) блокировка;
- управление сигналами;
- аренда (лизинг, владение);
- уведомления об изменении файла и каталога;
- изменение ёмкости канала.

34) Системные вызовы read(), write() и lseek().

4 Лекция
ssize_t read(int fd, void *buf, size_t count).
Пытается записать count байт из фала, с которым связан файловый дескриптор fd в буфер, адрес которого начинается с buf. Если количество count равно нулю, то read() возвращает это нулевое значение и завершает свою работу. Если количество прочитанных байтов меньше, чем количество за-прошенных, то это не считается ошибкой — данные, например, могли быть почти в конце файла, в ка-нале, на терминале, или read() был прерван сигналом.
ssize_t write(int fd, const void *buf, size_t count).
Записывает до count байтов из буфера buf в файл, на который ссылается файловый дескриптор fd. Воз-вращает количество записанных байт, если всё хорошо или -1, если произошла ошибка. Если count ра-вен нулю, а файловый описатель ссылается на обычный файл, то будет возвращен ноль и больше не бу-дет произведено никаких действий.
off_t lseek(int fd, off_t offset, int whence).
Устанавливает смещение для файлового дескриптора fd в значение аргумента offset в соот-ветствии с параметром whence который может принимать одно из следующих значений:
SEEK_SET Смещение устанавливается в offset байт от начала файла.
SEEK_CUR Смещение устанавливается как текущее смещение плюс offset байт.
SEEK_END Смещение устанавливается как размер файла плюс offset байт.

SEEK_DATA Подогнать файловое смещение к следующему расположению, большему или равному зна-чению offset, по которому в файле есть данные. Если значение offset указывает на
данные, то файловое смещение устанавливается в offset.
SEEK_HOLE Подогнать файловое смещение к следующему промежутку, большему или равному значе-нию offset. Если значение offset указывает в середину промежутка, то файловое смещение устанавлива-ется в offset. Если перед offset нет промежутка, то файловое смещение подгоняется к концу файла (т.е., это скрытый промежуток, который есть в конце любого файла).

35) Доступность ввода вывода. Системный вызов fcntl() — управление сигналами и аренда.

11 Лекция
Управление сигналами
Функция sigaction() указывает, как должен обрабатываться сигнал процессом.
Для управления сигналами доступности ввода/вывода используются команды
F_GETOWN
F_SETOWN
F_GETOWN_EX
F_SETOWN_EX

F_SETSIG (int) — сигнал, который будет послан, когда станет возможен ввод или вывод, равным значе-нию, указанному в arg. Нулевое значение означает отправку сигнала SIGIO по умолчанию.
F_GETSIG (void) — Возвращает (как результат функции fcntl()) сигнал, отправляемый тогда, когда ста-новится возможным ввод или вывод. Нулевое значение означает отправку SIGIO.
F_GETOWN (void) — Вернуть (в качестве результата функции fcntl()) идентификатор процесса или группы процессов, которые в настоящее время получают сигналы SIGIO и SIGURG для событий на файловом дескрипторе fd.
F_SETOWN (int) — Установить идентификатор процесса или идентификатор группы процессов, кото-рые будут принимать сигналы SIGIO и SIGURG для событий на файловом дескрипторе fd.

Аренда файла предоставляет механизм, посредством которого процесс, который удерживает аренду («арендатор»), уведомляется сигналом, если процесс («нарушитель аренды») пытается выполнить вызов open(2) или truncate(2) для этого файла/дескриптора.
Для установки новой и получения текущей аренды открытого дескриптора файла используются, соот-ветственно, команды fcntl() F_SETLEASE и F_GETLEASE.
Существуют два типа аренды:
- аренда чтения;
- аренда записи.
Когда процесс («нарушитель аренды») выполняет вызов open(2) или truncate(2), который конфликтует с арендой, установленной через F_SETLEASE, то системный вызов блокируется ядром и ядро уведомляет арендатора сигналом (по умолчанию SIGIO).

36) Системные вызовы POSIX управления каталогами (opendir, closedir, readdir, ...).

Уведомления об из-менении файла и каталога.
11 Лекция
DIR* opendir(const char *name) — открыть поток директории и вернуть указатель на этот поток. В слу-чае неудачи возвращает NULL.
Int closedir(DIR* dirp) — закрывает поток директории dirp. В случае удачи возвращает файловый де-скриптор dirp, в случае неудачи — 0.

Struct dirent *readdir(DIR* dirp) — считать файл из директории и вернуть указатель на него. В против-ном случае вернуть NULL.
Структура dirent содержит поля:
```
struct dirent {
        ino_t          d_ino;       /* Инода */
        off_t          d_off;       /* Возвращаемое значение telldir (НЕ ОФФСЕТ) */
        unsigned  short d_reclen;   /* Длинна записи */
        unsigned  char  d_type;     /* Тип файла (DT_CHR, DT_REG, DT_FIFO, ...)*/
        char           d_name[256]; /* Имя файла */
    };
```
F_NOTIFY (int) — уведомлять при изменении каталога, на который указывает fd или когда изменились файлы, которые в нём содержатся.
События, о наступлении которых делается уведомление, задаются в аргументе arg, который является битовой маской, получаемой сложением (OR) одного или более следующих бит:
DN_ACCESS — был произведен доступ к файлу (read(2), pread(2), readv(2), ...);
DN_MODIFY — файл был изменён (write(2), pwrite(2), writev(2), truncate(2), ftruncate(2), ...);
DN_CREATE — файл был создан (open(2), creat(2), mknod(2), mkdir(2), link(2), symlink(2), rename(2));
DN_DELETE — файл был удалён (unlink(2), rename to another directory, rmdir(2));
DN_RENAME — файл был переименован внутри каталога (rename(2));
DN_ATTRIB — у файла были изменены атрибуты (chown(2), chmod(2), utime[s](2), ...).
Чтобы не уведомлять о событиях в arg следует передать 0.

37) Файлы, отображаемые в памяти. Системные вызовы POSIX mmap(), mremap(), mlock(), munlock().

13 Лекция
С появлением концепции виртуальной памяти, когда физические размеры памяти перестали играть роль сдерживающего фактора в развитии вычислительных систем, стало возможным отображать файлы непосредственно в адресное пространство процессов.
Иными словами, появилась возможность работать с файлами как с обычной памятью, заменив выпол-нение базовых операций над ними с помощью системных вызовов на использование операций обычных языков программирования.

С точки зрения программиста работа с такими файлами выглядит следующим образом:
1) Вначале файл необходимо открыть, используя обычный системный вызов open(). (отобразить в дис-ковое а потом в адресное).
2) Отображение файла в адресное пространство процесса — системный вызов mmap().
(Файл после этого можно и закрыть, выполнив системный вызов close())
3. После с содержимым файла можно работать, как с содержимым обычной области памяти.
4. По окончании работы с содержимым файла освободить дополнительно выделенную процессу область памяти — системный вызов munmap().

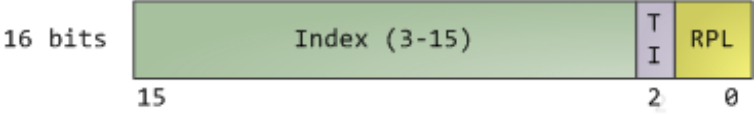
```
void *mmap(  void *start,   // адрес начала отображения
        size_t length,      // количество отображаемых байтов
        int prot,  // желаемый режим защиты памяти
        int flags,      // тип отражаемого объекта и опции
        int fd,         // дескриптор открытого файла
        off_t offset  // смещение в файле
);
```
mmap() служит для отображения предварительно открытого файла в адресное пространство вычисли-тельной системы.

```
void *mremap(void *old_address, // старый адрес виртуальной памяти
             size_t old_size,   // старый размер блока виртуальной памяти
             size_t new_size,   // требуемый размер блока виртуальной памяти
             unsigned long flags     // флаги переопределения
);
```
mremap() увеличивает или уменьшает размер текущего отражения памяти, одновременно перемещая его при необходимости, что контролируется параметром flags и доступным виртуальным адресным пространством.
```
int mlock(     const void *addr,   // начало области
          size_t len          // размер области
);
```
mlock() запрещает страничный обмен памяти в области, начинающейся с адреса addr длиной len бай-тов.

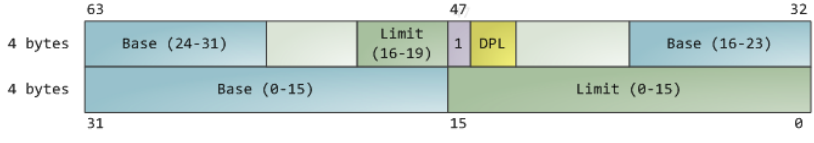
```
int munlock(   const void *addr,   // начало области
          size_t len          // размер области
);
```
munlock() разрешает страничный обмен в областях памяти, указание на которую начинается с адреса addr длиной len байтов.

38) Сегментная организация памяти защищенного режима x86. Дескриптор сегмента.

Сегментная адресация памяти — схема логической адресации памяти компьютера в архитектуре x86. Линейный адрес конкретной ячейки памяти, который в некоторых режимах работы процессора будет совпадать с физическим адресом, делится на две части: сегмент и смещение. В 32-битном защищенном режиме также используется сегментированная модель памяти, однако уже организованная по другому принципу: расположение сегментов описывается специальными структурами (таблицами дескрипто-ров), расположенными в оперативной памяти.
Сегменты памяти также выбираются все теми же сегментными регистрами. Значение сегментного ре-гистра (сегментный селектор) больше не является сырым адресом, но вместо этого представляет собой структуру такого вида:



Существует два типа дескрипторных таблиц: глобальная (GDT) и локальная (LDT). Глобальная таблица описывает сегменты операционной системы и разделяемых структур данных, у каждого ядра своя. Ло-кальная таблица может быть определена для каждой конкретной задачи (процесса). Бит TI равен 0 для GDT и 1 для LDT. Индекс задаёт номер дескриптора в таблице дескрипторов сегмента. Поле RPL рас-шифровывается как Requested Privilege Level.
Сама таблица представляет собой просто массив, содержащий 8-байтные записи (дескрипторы сегмен-та), где каждая запись описывает один сегмент и выглядит так:



39) Параметры функции main() в языке C. Переменные среды и аргументы командной строки. Систем-ные вызовы setenv(), getenv().

У функции main() в языке программирования C существует три параметра, которые могут быть переда-ны ей операционной системой. Полный прототип функции main() выглядит следующим образом:
int main(int argc, char *argv[], char *envp[]);
Первые два параметра при запуске программы на исполнение командной строкой позволяют узнать полное содержание командной строки. Вся командная строка рассматривается как набор слов, разде-ленных пробелами. Через параметр argc передается количество слов в командной строке, которой была запущена программа
Третий параметр — envp — является массивом указателей на параметры окружающей среды процесса. Начальные параметры окружающей среды процесса задаются в специальных конфигурационных фай-лах для каждого пользователя и устанавливаются при входе пользователя в систему. В дальнейшем они могут быть изменены с помощью специальных команд операционной системы UNIX. Каждый пара-метр имеет вид: переменная=строка. Такие переменные используются для изменения долгосрочного поведения процессов, в отличие от аргументов командной строки. Например, задание параметра TERM=vt100 может говорить процессам, осуществляющим вывод на экран дисплея, что работать им придется с терминалом vt100.
Переменная среды — текстовая переменная операционной системы, хранящая какую-либо информа-цию — например, данные о настройках системы.
Int setenv(const char *name, const char *value, int overwrite) — устанавливает или изменяет значение пе-ременной среды. Name — имя переменной среды, value — значение. Если имя уже существует, то зна-чение переменной изменяется на value, если overwrite не равно нулю. В противном случае не меняется.
Char *getenv(const char *name) — получить переменную окружения под именем name. Возвращает ука-затель на строку с переменной value.

40) Формирование адреса памяти в x86 (адрес эффективный, логический, линейный, виртуальный, фи-зический).

Эффективный адрес — это начало пути. Он задаётся в аргументах индивидуальной машинной инструк-ции, и вычисляется из значений регистров, смещений и масштабирующих коэффициентов, заданных в ней явно или неявно.
Например, для инструкции (ассемблер в AT&T-нотации)
addl %eax, 0x11(%ebp, %edx, 8)
эффективный адрес операнда-назначения будет вычислен по формуле:
eff_addr = EBP + EDX * 8 + 0x11

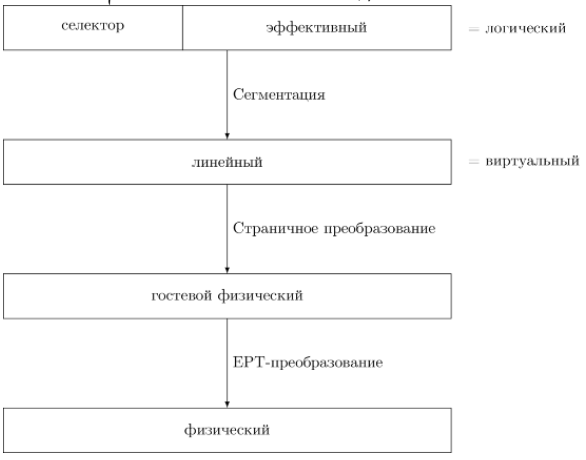
Без знания номера и параметров сегмента, в котором указан эффективный ад-рес, последний бесполезен. Сам сегмент выбирается ещё одним числом, именуемым селектором. Пара чисел, записываемая как selector:offset, получила имя логический адрес.

Эффективный адрес — это смещение от начала сегмента — его базы. Если сло-жить базу и эффективный адрес, то получим число, называемое линейным адре-сом:
lin_addr = segment.base + eff_addr

В литературе и в документации других архитектур встречается ещё один тер-мин — виртуальный адрес. Он не используется в документации Intel на IA-32, однако встречается, например, в описании Intel® Itanium, в котором сегмен-тация не используется. Можно смело считать, что для IA-32 виртуальный == линейный.

Следующее после сегментации преобразование адресов: линейный ⟶ физический — имеет м ножество вариаций в своём алгоритме, в зависимости от того, в каком режиме (32-битном, PAE или 64-битном) находится процессор.

В целом организация памяти выглядит так:



41) Жесткие и символические ссылки. Системные вызовы создания и удаления.

4 Лекция
Жесткой ссылкой (hard link) считается элемент каталога, указывающий непосредственно на некоторый индексный дескриптор. Это «настоящее имя файла». Жесткие ссылки очень эффективны, но у них су-ществуют определенные ограничения, так как они могут создаваться только в пределах одной физиче-ской файловой системы. Когда создается такая ссылка, связываемый файл должен уже существовать. Кроме того, жесткой ссылкой не могут связываться каталоги.
Символическая ссылка (Symbolic link) — файл специального типа, содержащий путь к другому файлу. Указание на то, что данный элемент каталога является символической ссылкой, находится в индексном дескрипторе. Обычные команды доступа к файлу вместо получения данных из физического файла, бе-рут их из файла, имя которого приведено в ссылке.
Этот путь может указывать на что угодно — это может быть каталог, он может даже находиться в дру-гой физической файловой системе, более того, указанного файла может и вовсе не быть.
Файл создается с помощью вызовов open() или create() всегда с одним именем.
Дополнительные имена можно назначить файлу используя системный вызов link(). Этот вызов является проекцией команды оболочки ln.
Файл удаляется функцией unlink(), которая удаляет одну из жестких ссылок. Когда будет удалена по-следняя, файл считается удаленным и ресурсы, которые он занимает на носителе освобождаются.
Создание и/или удаление символической ссылки никогда не затрагивают ни имени файла, на который она ссылается, ни его индексного дескриптора.
Файл может быть удален, а ссылка остается («висящая»).
Файл может не существовать в момент создания символической ссылки.

42) Получение метаданных файла. Системные вызовы stat(), fstat(), lstat().

4 Лекция
Чтобы получить метаданные о файле используются вызовы stat(), fstat(), lstat().
Данные системные вызовы возвращают информацию о файле в буфер, на который указывает statbuf, которая содержит следующие поля
```
struct stat {
dev_t         st_dev;   // ID устройства с файлом
ino_t         st_ino;         // номер иноды
mode_t  st_mode; // тип файла и режим доступа
nlink_t  st_nlink; // количество жёстких ссылок
uid_t         st_uid;         // идентификатор пользователя-владельца
gid_t         st_gid;         // идентификатор группы-владельца
dev_t         st_rdev;  // идентификатор устройства
off_t         st_size;   // общий размер в байтах
blksize_t     st_blksize;     // размер блока ввода-вывода ФС
blkcnt_t      st_blocks;      // количество выделенных 512Б блоков

struct timespec st_atim;      // время последнего доступа
struct timespec st_mtim;      // время последнего изменения
struct timespec st_ctim;      // время последней смены состояния
};
```

Вызов int stat(const char *pathname, struct stat *statbuf) и fstatat возвращают информацию о файле, ука-занном в pathname.
Вызов int fstat(int fd, struct stat *statbuf);идентичен stat(), но файл задаётся файловым дескриптором fd.
Вызов int lstat(const char *pathname, struct stat *statbuf); идентичен stat(), но в случае, если pathname яв-ляется символьной ссылкой, то возвращается информация о самой ссылке, а не о файле, на который она указывает.

43) Каталоги. Системные вызовы чтения, смены, создания и удаления.

4 Лекция
Каталоги — это файлы специального типа. Они хранят имена и номера индексных дескрипторов. Все, чем отличается каталог от обычного файла на низком уровне — это значение признака типа в индекс-ном дескрипторе.
В остальном хранение каталогов на диске не отличается от хранения обычных файлов. При создании на диске файловой системы создается один каталог — корневой каталог. При необходимости в корневом каталоге можно создавать другие каталоги, а их имена записать в корневой — каталоги первого уровня вложенности.
Struct dirent *readdir(DIR *dir) — возвращает указатель на структуру dirent, представляющую запись ка-талога в потоке каталога, указанного в dir.

```
Struct dirent {
    ino_t              d_ino;         // номер иноды файла
    off_t              d_off;         // это не смещение!!!
    Unsigned short          d_reclen;      // длина этой записи
    unsigned char      d_type;  // тип файла; поддерживается не во всех ФС
    char          d_name[256];      // имя файла с null в конце
};
```
d_off — значение, возвращаемое в d_off, тоже самое что и после вызова telldir(3) в текущем положении курсора в потоке каталога.

Void rewinddir(DIR *dirp) — меняет позицию поиска каталога dirp на начало этого каталога.
Int mkdir(const char *pathname, mode_t mode) — создаёт директорию по пути pathname.
Int unlink(const char *pathname) — удаляет имя файла и, по возможности, сам файл.

44) Два типа устройств и операции над ними. Системный вызов ioctl()

4 Лекция
ОС UNIX обобщает концепцию файла как универсальную абстракцию.
Практически любое внешнее устройство может быть представлено на пользовательском уровне, как файл специального типа.
Устройства, которые могут быть представлены в виде файлов, делятся на два типа — символьные (или потоковые) и блочные. Иногда используются термины «байт-ориентированные» и «блок-ориентированные» устройства.
Основными операциями над байт-ориентированными устройствами являются запись и чтение одного (очередного) символа (байта).
Блок-ориентированные устройства воспринимаются как хранилища данных, разделенных на блоки фиксированного размера. Основными операциями, соответственно, являются чтение и запись заданного блока.
Байт-ориентированные — терминал, клавиатура, мышь, принтер, звуковая карта, /dev/null, /dev/zero, /dev/random, ...
Блок-ориентированные — диски, разделы дисков.
Файлы устройств могут быть открыты на чтение и запись. В результате, используя полученный де-скриптор, можно писать на устройство и читать с него.
Блок-ориентированные устройства поддерживают позиционирование с помощью вызова lseek().
Байт-ориентированные устройства операцию позиционирования не поддерживают.
Это основное отличие устройств одного типа от другого.

Над устройствами определены дополнительные операции, специфические для конкретного устройства. Все операции данной категории выполняются с помощью системного вызова управления устройствами — ioctl()

int ioctl(int d, int request, ...) — манипулирует базовыми параметрами устройств, представленных в ви-де специальных файлов.
В частности, многими оперативными характеристиками специальных символьных файлов (например терминалов) можно управлять через ioctl запросы. В качестве аргумента d должен быть ука-зан открытый файловый дескриптор.
Второй аргумент является кодом запроса, который зависит от устройства.
Третий аргумент является указателем на память, который не имеет типа.
Традиционно это char *argp или void *argp.
Ioctl запрос request кодирует в себе либо аргумент, который является параметром in либо аргумент, который является параметром out и кроме того размер аргумента argp в байтах.

45) Системные вызовы read(), write() и lseek().

4 Лекция
ssize_t read(int fd, void *buf, size_t count).
Пытается записать count байт из фала, с которым связан файловый дескриптор fd в буфер, адрес которого начинается с buf. Если количество count равно нулю, то read() возвращает это нулевое значение и завершает свою работу. Если количество прочитанных байтов меньше, чем количество за-прошенных, то это не считается ошибкой — данные, например, могли быть почти в конце файла, в ка-нале, на терминале, или read() был прерван сигналом.

Ssize_t write(int fd, const void *buf, size_t count).
Записывает до count байтов из буфера buf в файл, на который ссылается файловый дескриптор fd. Воз-вращает количество записанных байт, если всё хорошо или -1, если произошла ошибка. Если count ра-вен нулю, а файловый описатель ссылается на обычный файл, то будет возвращен ноль и больше не бу-дет произведено никаких действий.
Off_t lseek(int fd, off_t offset, int whence).
Устанавливает смещение для файлового дескриптора fd в значение аргумента offset в соот-ветствии с параметром whence который может принимать одно из следующих значений:
SEEK_SET Смещение устанавливается в offset байт от начала файла.
SEEK_CUR Смещение устанавливается как текущее смещение плюс offset байт.
SEEK_END Смещение устанавливается как размер файла плюс offset байт.
SEEK_DATA Подогнать файловое смещение к следующему расположению, большему или равному зна-чению offset, по которому в файле есть данные. Если значение offset указывает на
данные, то файловое смещение устанавливается в offset.
SEEK_HOLE Подогнать файловое смещение к следующему промежутку, большему или равному значе-нию offset. Если значение offset указывает в середину промежутка, то файловое смещение устанавлива-ется в offset. Если перед offset нет промежутка, то файловое смещение подгоняется к концу файла (т.е., это скрытый промежуток, который есть в конце любого файла).

46) Семафоры UNIX. Создание, управление и операции (semget, semctl, semop).

9 Лекция
Над семафорами выполняются две операции — увеличение значения семафора на единицу sem_post(3) и уменьшение значения семафора на единицу sem_wait(3). Если значение семафора равно нулю, то операция sem_wait(3) блокирует работу до тех пор, пока значение не станет больше нуля.

Создание набора семафоров.
```
int semget(key_t key,    // ключ
           int nsems,    // число семафоров в наборе
           int semflg); // флаги
```
Если значение key не равно IPC_PRIVATE, а semflg равно нулю, key можно использовать для получения идентификатора уже созданного набора семафоров.

Управление набором семафоров.
```
int semctl(int semid,     // идентификатор набора семафоров, выданный semget()
           int semnum, // номер семафора из набора
           int cmd,    // операция (команда)
           ...);       // опциональный параметр arg
```
Вызов semctl() выполняет операцию, определённую аргументом cmd, над набором семафо-ров System V, указанном в semid, или над семафором с номером semnum из этого набора. semnum — номер семафора в множестве. Данный вызов имеет три или четыре аргумента, в зависимо-сти от значения cmd.
Возможные значения аргумента cmd (Общие для всех IPC-механизмов)
IPC_STAT — поместить информацию о состоянии множества семафоров, содержащуюся в структуре данных, ассоциированной с идентификатором semid, в пользовательскую структуру, на кото-рую указывает arg.buf.
IPC_SET — в структуре данных, ассоциированной с идентификатором semid, переустановить значения действующих идентификаторов пользователя и группы, а также прав на операции.
IPC_RMID — удалить из системы идентификатор semid, ликвидировать множество семафоров и ассо-циированную с ним структуру данных.

Операции над множеством семафоров.
int semop(int semid, // идентификатор набора семафоров
struct sembuf *sops, // структура с параметрами операции
size_t nsops); //
Набор операций из sops выполняется в порядке появления в массиве и является атомарным.
sem_op > 0 — если значение sem_op является положительным целым числом, то оно добавляется к значению семафора semval.
sem_op == 0 — если значение sem_op равно нулю, то процесс должен иметь права на чтение набора семафоров.
sem_op < 0 — если значение sem_op меньше нуля, то процесс должен иметь права на измене-
ние набора семафоров.

47) Сегменты общей памяти UNIX.

9-10 Лекция
Сегмент общей памяти уникально идентифицируется положительным целым (shmid) и имеет связанную структуру данных struct shmid_ds.
```
Struct shmid_ds {
    struct ipc_perm   msg_perm;
    size_t            shm_segsz;
    pid_t             shm_cpid;
    pid_t             shm_lpid;
    shmatt_t      shm_nattch;
    time_t            shm_atime;
    time_t            shm_dtime;
    time_t            shm_ctime;
};
```
Работа с совместно используемой памятью начинается с того, что процесс при помощи системного вы-зова shmget() создает совместно используемый сегмент, указывая первоначальные права доступа к это-му сегменту (чтение и/или запись), а также его размер в байтах.
Чтобы затем получить доступ к совместно используемому сегменту, его нужно присоединить посредством системного вызова shmat(), который разместит сегмент в виртуальном пространстве про-цесса. После присоединения, в соответствии с правами доступа, процессы могут читать данные из сег-мента и записывать их (быть может, синхронизируя свои действия с помощью семафоров).
Когда разделяемый сегмент становится ненужным, его следует отсоединить, воспользовавшись систем-ным вызовом shmdt().
Для выполнения управляющих действий над разделяемыми сегментами памяти служит системный вы-зов shmctl().

48) Работа с общей памятью. Системные вызовы shmget(), shmctl(), shmat(), shmdt().

10 Лекция
Работа с совместно используемой памятью начинается с того, что процесс при помощи системного вы-зова shmget() создает совместно используемый сегмент, указывая первоначальные права доступа к это-му сегменту (чтение и/или запись), а также его размер в байтах.
Чтобы затем получить доступ к совместно используемому сегменту, его нужно присоединить посредством системного вызова shmat(), который разместит сегмент в виртуальном пространстве про-цесса. После присоединения, в соответствии с правами доступа, процессы

могут читать данные из сег-мента и записывать их (быть может, синхронизируя свои действия с помощью семафоров).
Когда разделяемый сегмент становится ненужным, его следует отсоединить, воспользовавшись систем-ным вызовом shmdt().
Для выполнения управляющих действий над разделяемыми сегментами памяти служит системный вы-зов shmctl().

```
int shmget(   key_t key,   // ключ
          int size,  // размер сегмента
          int shmflg     // флаги создания
);
```
Возвращает идентификатор сегмента общей памяти, соответствующий значению аргумента key.

```
int shmctl( int shmid,
            int cmd,
            struct shmid_ds *buf );
```
shmctl() позволяет пользователю:
- получать информацию о общих сегментах памяти;
- устанавливать владельца, группу общего сегмента, права на него;
- удалить сегмент.

shmid — идентификатор общего сегмента памяти, полученный при помощи shmget().

```
void *shmat(                  // операция присоединения сегментов (attach)
          int shmid,      // id сегмента общей памяти
          const void *shmaddr, // адрес присоединения
          int shmflg          // флаги )
```
Функция shmat() подсоединяет сегмент общей памяти shmid к адресному пространству вызывающего процесса.
shmid — идентификатор сегмента общей памяти, предварительно полученный при помощи системного вызова shmget().
shmaddr — адрес присоединенного сегмента.

```
int shmdt(                // операция отсоединения сегментов (detach)
    const void *shmaddr    // адрес присоединения )
```
Функция shmdt() отсоединяет сегмент общей памяти, находящийся по адресу shmaddr, от адресного пространства вызвающего процесса. Эта функция освобождает занятую ранее этим сегментом область памяти в адресном пространстве процесса.

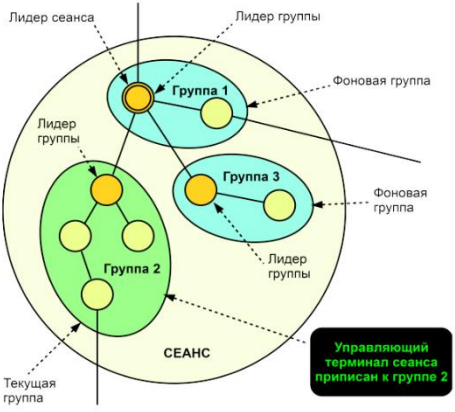
49) Структура ФС ОС UNIX. Хранение данных.

4 Лекция
- единое дерево каталогов.
- в имя файла ни в каком виде не входит имя устройства, на котором файл находится.
- если в системе присутствует несколько устройств прямого доступа (dasd), один их них объявляется корневым (root), и все остальные «монтируются» (mount) в тот или иной каталог, называемый точкой монтирования (mount point).
- для указания полного пути к файлу на примонтированном устройстве спереди к имени файла добав-ляется полный путь к точке монтирования. Например, если у нас есть дискета, смонтированная в /media/myfloppy, а на ней есть каталог work, а в нем файл foo.c, то полный путь к этому файлу будет /media/myfloppy/work/foo.c.

В ОС UNIX каталоги хранят только имя файла и некоторый номер, позволяющий идентифицировать соответствующий файл. Вся остальная информация о файле, как то размер, расположение на диске, да-ты создания, модификации и последнего обращения, данные о владельце и о правах доступа к нему свя-зываются не с именем файла, а с этим самым номером, который связан индексным дескриптором фай-ла.
ИНДЕКСНЫЙ ДЕСКРИПТОР — хранимая на внешнем ЗУ (диске) структура данных, содержащая всю информацию о файле, исключая его имя. Размер индексного дескриптора обычно составляет 128 бай-тов.
Допускается, чтобы несколько имен файлов, в том числе и в разных каталогах, ссылались на один и тот же номер индексного дескриптора.

50) Иерархия процессов в операционной системе. Группы и сеансы. Системные вызовы POSIX порож-дения и управления процессами.

5 Лекция
В операционной системе *NIX все процессы кроме одного, создающегося при старте операционной си-стемы, могут быть порождены только какими-либо другими процессами.
В качестве процесса прародителя всех остальных процессов в разных *NIX-ообразных системах могут выступать процессы с номерами 1 или 0.



Группа процессов включает в себя один или более процессов и существует, пока в группе присутствует хотя бы один процесс. Каждый процесс обязательно включен в какую-нибудь группу. При рождении но-вого процесса он попадает в ту же группу процессов, в которой находится его родитель.
Группы процессов объединяются в сеансы. Понятие сеанса изначально было введено в UNIX для логи-ческого объединения групп процессов, созданных в результате каждого входа и последующей работы пользователя в системе.

Pid_t fork(void);
fork создает процесс-потомок, который отличается от родительского только значениями PID (идентификатор процесса) и PPID (идентификатор родительского процесса), а также тем фактом, что счетчики использования ресурсов установлены в 0. Блокировки файлов и сигналы, ожидающие обра-ботки, не наследуются.
Существует два способа корректного завершения процесса в программах, написанных на языке C.
1) процесс корректно завершается по достижении конца функции main() или при выполнении оператора return в функции main();

2) второй способ применяется при необходимости завершить процесс в каком-либо другом месте программы. Для этого применяется функция exit(3) из стандартной библиотеки функций для языка C.

Pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
Функция wait приостанавливает выполнение текущего процесса до тех пор, пока какой нибудь из до-черних процессов не завершится, или до появления сигнала, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик.

51) Жесткие и символические ссылки. Системные вызовы создания и удаления.

4 Лекция
Жесткой ссылкой (hard link) считается элемент каталога, указывающий непосредственно на некоторый индексный дескриптор. Это «настоящее имя файла». Жесткие ссылки очень эффективны, но у них су-ществуют определенные ограничения, так как они могут создаваться только в пределах одной физиче-ской файловой системы. Когда создается такая ссылка, связываемый файл должен уже существовать. Кроме того, жесткой ссылкой не могут связываться каталоги.
Символическая ссылка (Symbolic link) — файл специального типа, содержащий путь к другому файлу.
Указание на то, что данный элемент каталога является символической ссылкой, находится в индексном дескрипторе. Обычные команды доступа к файлу вместо получения данных из физического файла, бе-рут их из файла, имя которого приведено в ссылке.
Этот путь может указывать на что угодно — это может быть каталог, он может даже находиться в дру-гой физической файловой системе, более того, указанного файла может и вовсе не быть.
Файл создается с помощью вызовов open() или create() всегда с одним именем.
Дополнительные имена можно назначить файлу используя системный вызов link(). Этот вызов является проекцией команды оболочки ln. Файл удаляется функцией unlink(), которая удаляет одну из жестких ссылок. Когда будет удалена последняя, файл считается удаленным и ресурсы, которые он занимает на носителе освобож-даются. Создание и/или удаление символической ссылки никогда не затрагивают ни имени файла, на который она ссылается, ни его индексного дескриптора. Файл может быть удален, а ссылка остается («висящая»). Файл может не существовать в момент создания символической ссылки.

52) Системные вызовы read(), write() и lseek().

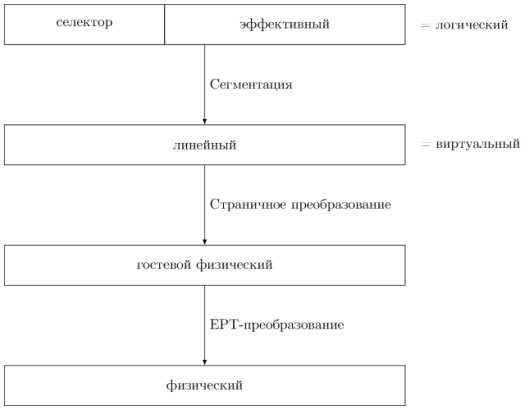
4 Лекция
ssize_t read(int fd, void *buf, size_t count).
Пытается записать count байт из фала, с которым связан файловый дескриптор fd в буфер, адрес которого начинается с buf. Если количество count равно нулю, то read() возвращает это нулевое значение и завершает свою работу. Если количество прочитанных байтов меньше, чем количество за-прошенных, то это не считается ошибкой — данные, например, могли быть почти в конце файла, в ка-нале, на терминале, или read() был прерван сигналом.
Ssize_t write(int fd, const void *buf, size_t count — Записывает до count байтов из буфера buf в файл, на который ссылается файловый дескриптор fd. Возвращает количество записанных байт, если всё хорошо или -1, если произошла ошибка. Если count равен нулю, а файловый описатель ссылается на обычный файл, то будет возвращен ноль и больше не будет произведено никаких действий.

Off_t lseek(int fd, off_t offset, int whence).

Устанавливает смещение для файлового дескриптора fd в значение аргумента offset в соответствии с параметром whence который может принимать одно из следующих значений:
SEEK_SET Смещение устанавливается в offset байт от начала файла.
SEEK_CUR Смещение устанавливается как текущее смещение плюс offset байт.
SEEK_END Смещение устанавливается как размер файла плюс offset байт.
SEEK_DATA Подогнать файловое смещение к следующему расположению, большему или равному зна-чению offset, по которому в файле есть данные. Если значение offset указывает на
данные, то файловое смещение устанавливается в offset.
SEEK_HOLE Подогнать файловое смещение к следующему промежутку, большему или равному значе-нию offset. Если значение offset указывает в середину промежутка, то файловое смещение устанавлива-ется в offset. Если перед offset нет промежутка, то файловое смещение подгоняется к концу файла (т.е., это скрытый промежуток, который есть в конце любого файла).

53) Формирование адреса памяти в x86 (адрес эффективный, логический, линейный, виртуальный, фи-зический).

Эффективный адрес — это начало пути. Он задаётся в аргументах индивидуальной машинной инструкции, и вычисляется из значений регистров, смещений и масштабирующих коэффициентов, за-данных в ней явно или неявно.
Например, для инструкции (ассемблер в AT&T-нотации)
addl %eax, 0x11(%ebp, %edx, 8)
эффективный адрес операнда-назначения будет вычислен по формуле:
eff_addr = EBP + EDX * 8 + 0x11
Без знания номера и параметров сегмента, в котором указан эффективный адрес, последний бесполезен. Сам сегмент выбирается ещё одним числом, име-нуемым селектором. Пара чисел, записываемая как selector:offset, получила имя логический адрес.
Эффективный адрес — это смещение от начала сегмента — его базы. Если сложить базу и эффективный адрес, то получим число, называемое линейным адресом: lin_addr = segment.base + eff_addr
В литературе и в документации других архитектур встречается ещё один термин — виртуальный адрес. Он не используется в документации Intel на IA-32, однако встречается, например, в описании Intel® Itanium, в котором сегментация не используется. Можно смело считать, что для IA-32 виртуаль-ный == линейный.
Следующее после сегментации преобразование адресов: линейный ⟶ физический — имеет м ножество вариаций в своём алгоритме, в зависимости от того, в каком режиме (32-битном, PAE или 64битном) находится процессор. В целом организация памяти выглядит так:



54) Потоки исполнения (нити), их характеристики (атрибуты) и применение.

14 Лекция
У потока есть счетчик команд, отслеживающий, какую очередную инструкцию нужно выполнять. У не-го есть регистры, в которых содержатся текущие рабочие переменные. У него есть стек с протоколом выполнения, содержащим по одному фрейму для каждой вызванной, но еще не возвратившей управле-ние процедуры. Хотя поток может выполняться в рамках какого-нибудь процесса, сам поток и его про-цесс являются разными понятиями и должны рассматриваться отдельно.
Процессы используются для группировки ресурсов в единое образование, а потоки являются «сущностью», распределяемой для выполнения на центральном процессоре.
Потоки добавляют к модели процесса возможность реализации нескольких в значительной степени независимых друг от друга выполняемых задач в единой среде процесса.
Наличие нескольких потоков, выполняемых параллельно в рамках одного процесса, является аналогией наличия нескольких процессов, выполняемых параллельно на одном компьютере. В первом случае потоки используют единое адресное пространство и другие ресурсы. А в последнем случае про-цессы используют общую физическую память, диски, принтеры и другие ресурсы.
Поскольку потоки обладают некоторыми свойствами процессов, их иногда называют облегченными процессами. Термин «многопоточный режим» также используется для описания ситуации, при которой допускается работа нескольких потоков в одном и том же процессе.

Атрибуты включают размер стека, параметры планирования и другие элементы, необходимые при использовании потока.
Каждый поток имеет собственный стек. Стек каждого потока содержит по одному фрейму для каждой уже вызванной, но еще не возвратившей управление процедуры.
Необходимость в подобных мини-процессах, называемых потоками, обусловливается целым рядом причин. Основная причина использования потоков заключается в том, что во многих приложени-ях одновременно происходит несколько действий, часть которых может периодически блокироваться. Вторым аргументом в пользу потоков является легкость (то есть быстрота) их создания и ликвидации по сравнению с более «тяжеловесными» процессами. Во многих системах создание потоков осуществляет-ся в 10–100 раз быстрее, чем создание процессов. Третий аргумент в пользу потоков — производитель-ность. И наконец, потоки весьма полезны для систем, имеющих несколько центральных процессоров (ядер), где есть реальная возможность параллельных вычислений.

55) Преимущества и недостатки потоков по сравнению с процессами. Использование объектов потока-ми и общий программный интерфейс (create(), exit(), join(), yield()).

14 Лекция
+: Каждый поток может иметь доступ к любому адресу памяти в пределах адресного пространства про-цесса, один поток может считывать данные из стека другого потока, записывать туда свои данные и да-же стирать оттуда данные. Защита между потоками отсутствует, потому что ее невозможно осуществить и в ней нет необходимости.
В отличие от различных процессов, которые могут принадлежать различным пользователям и которые могут конфликтовать друг с другом, один поток всегда принадлежит одному и тому же пользо-вателю, который, по-видимому, и создал несколько потоков для их совместной работы.
–: Различные потоки в процессе не обладают той независимостью, которая есть у различных процессов. У всех потоков абсолютно одно и то же адресное пространство, а значит, они так же совместно исполь-зуют одни и те же глобальные переменные. Главный недостаток потоков в ядре состоит в весьма суще-ственных затратах времени на системный вызов,

поэтому если операции над потоками (создание, уда-ление и т. П.) выполняются довольно часто, это влечет за собой более существенные издержки.

В параметре thread_create() обычно указывается имя процедуры, которая запускается в новом потоке. Создающий поток обычно получает идентификатор потока, который дает имя новому потоку.
Когда поток завершает свою работу, выход из него может быть осуществлен за счет вызова библиотеч-ной процедуры, к примеру thread_exit(). После этого он прекращает свое существование и больше не фигурирует в работе планировщика.
В некоторых случаях какой-нибудь поток для выполнения выхода может ожидать выхода из какого-нибудь другого (указанного) потока, для чего используется thread_join().
Другой распространенной процедурой, вызываемой потоком, является thread_yield().
Она позволяет потоку добровольно уступить центральный процессор для выполнения другого потока. Важность вызова такой процедуры обусловливается отсутствием прерывания по таймеру, которое есть у процессов и благодаря которому фактически задается режим многозадачности.

56) Потоки (нити) исполнения POSIX и их атрибуты. Понятие потокобезопасности и уступчивости. Об-работчики очистки.

15 Лекция
В POSIX.1 определён набор интерфейсов для работы с потоками, более известными как потоки POSIX или Pthreads. В одном процессе может быть несколько потоков, которые выполняют одну программу. Эти потоки работают с общей глобальной памятью (сегментами данных и кучи), но у каждого потока есть собственный стек (автоматические переменные).
Также, в POSIX.1 требуется, чтобы потоки имели общий диапазон других атрибутов (т. е., эти атрибуты процесса, а не потоков):
- идентификатор процесса;
- идентификатор родительского процесса;
- идентификатор группы процессов и сеанса;
- управляющий терминал;
- идентификаторы пользователя и группы;
- открытые файловые дескрипторы;
- обычные блокировки (смотрите fcntl(2));
- обработчики сигналов;
- маска создания режима доступа к файлу (umask(2));
- текущий каталог (chdir(2)) и корневой каталог (chroot(2));
- интервальные таймеры (setitimer(2)) и таймеры POSIX (timer_create(2));
- значение уступчивости (setpriority(2));
- ограничения по ресурсам (setrlimit(2));
- измерители потребления времени ЦП (times(2)) и ресурсов (getrusage(2));

Как и для стека, в POSIX.1 определены другие атрибуты, которые уникальны в каждом потоке:
- идентификатор потока (тип данных pthread_t);
- маска сигналов (pthread_sigmask(3));
- переменная errno (определена как макрос, задающий для каждого потока собственное l-value);
- альтернативный стек сигнала (sigaltstack(2));
- алгоритм и приоритет планирования реального времени (sched(7));

Следующие свойства есть только в Linux и также уникальны в каждом потоке:
- мандаты (смотрите capabilities(7))
- привязка к ЦП (sched_setaffinity(2))

Потокобезопасная функция — это функция, которую можно безопасно (т. е., это приведёт к единым ре-зультатам независимо от окружения) вызывать из нескольких потоков одновременно. Примеры потоко-безопасных функций: getopt, ctime, setenv, rand,...
Уступчивость
В ядре работают специальные алгоритмы, которые управляют переключением процессов/потоков и распределяют между ними системные ресурсы. Есть возможность влиять на этот механизм.
При распределении системных ресурсов ядро по умолчанию рассматривает процессы/потоки как равные. Но можно сообщить ядру, что некоторый процесс/поток желает уступить часть своего права на системные ресурсы. В таком случае говорят, что уступчивость процесса увеличивается.
Так же встречается понятие «приоритет процесса/потока», характеризующий величину, обратную уступчивости.
Любые обработчики очистки (clean-up), установленные pthread_cleanup_push3(3), которые еще не были вызваны (pop), вызываются (в обратном порядке, в котором они были назначены (push)) и выполняются.
Если поток имеет какие-либо специфичные для потока данные, то после выполнения обработчиков очистки вызываются соответствующие функции-деструкторы в неопределенном порядке.

57) Организация программы, использующей потоки

15 Лекция
Новый поток создается функцией pthread_create(), которой передается имя функции, с которой начнется выполнение потока, — потоковая функция. Далее, вызывающая сторона продолжает выполнять какие-то свои действия параллельно потоковой функции.
Поток завершает выполнение задачи когда:
- потоковая функция выполняет return и возвращает результат произведенных вычислений;
- в результате вызова завершения исполнения потока pthread_exit();
- в результате вызова отмены потока pthread_cancel();
- один из потоков совершает вызов exit();
- основной поток в функции main() выполняет return, и в таком случае все нити процесса резко сворачи-ваются.

Если основной поток в функции main() выполнит pthread_exit() вместо просто exit() или вместо return, остальные потоки продолжат исполняться.
Ожидание потока
Чтобы синхронизировать потоки, используется функция pthread_join(). Она ожидает завершения ука-занного потока. Если этот поток к тому времени был уже завершен, то функция немедленно возвращает значение.
Для досрочного завершения потока можно воспользоваться функцией pthread_cancel(). Функция pthread_cancel() возвращается сразу, но этот не означает, что поток будет в обязательном порядке завер-шен досрочно — поток не только может самостоятельно выбрать момент завершения в ответ на вызов pthread_cancel(), но и его проигнорировать.
Вызов функции pthread_cancel() следует рассматривать как запрос на выполнение досрочного заверше-ния потока. Поэтому, если важно, чтобы поток был удален, нужно дождаться его завершения функцией pthread_join().
Потоки бывают двух типов — к которым можно подсоединиться с помощью pthread_join() пожидать его завершения, и к которым подсоединиться нельзя. По умолчанию потоки создаются подсоединяемыми. Для отсоединения потока используется вызов pthread_detach(). После это перехватить поток с помощью вызова pthread_join() становится невозможно, также невозможно получить статус его завершения и прочее. Отменить отсоединенное состояние также невозможно.

Если завершение потока не перехватить вызовом pthread_join() в результате получим зомби-поток. Если завершился отсоединенный поток, все будет нормально.

58) Потоки (нити) исполнения POSIX. Системные вызовы pthread_create(), pthread_join(), pthread_cancel(). Подсоединение к потоку и отсоединение потока.

15 Лекция
В POSIX.1 определён набор интерфейсов для работы с потоками, более известными как потоки POSIX или Pthreads. В одном процессе может быть несколько потоков, которые выполняют одну программу. Эти потоки работают с общей глобальной памятью (сегментами данных и кучи), но у каждого потока есть собственный стек (автоматические переменные).

```
int pthread_create(pthread_t *restrict thread,  // здесь возвращается ID
                   const pthread_attr_t *restrict attr, // атрибуты потока
                   void *(*start_routine)(void*),      // функция, начинающая поток
                   void *restrict arg);  // аргумент для start_routine
```

Функция pthread_create() запускает новый поток в вызывающем процессе. Новый поток начинает вы-полнение с функции start_routine(), при этом этой функции передается единственный аргумент arg.

```
int pthread_join(pthread_t thread, void **retval);
```

Функция pthread_join() ожидает завершения потока, указанного потоком. Если этот поток уже завершился, pthread_join() немедленно возвращается. Поток, указанный в pthread_join(), должен быть подсоединяемым.

```
int pthread_cancel(pthread_t thread);
```

Функция pthread_cancel() отправляет запрос отмены потоку thread. Отреагирует ли целевой поток на запрос отмены, и когда, зависит от двух атрибутов, находящихся под контролем этого потока: его состояния отмены (cancelability state) и типа.

Для отсоединения потока используется вызов pthread_detach(). После это перехватить поток с помощью вызова pthread_join() становится невозможно, также невозможно получить статус его завершения и прочее. Отменить отсоединенное состояние также невозможно. Если завершение потока не перехватить вызовом pthread_join() в результате получим зомби-поток. Если завершился отсоединенный поток, все будет нормально. Любому потоку по умолчанию можно присоединиться вызовом pthread_join() и ожидать его завершения. Однако в некоторых случаях статус завершения потока и возврат значения нам не интересны.

59) Завершение нитей POSIX. Системные вызовы pthread_cancel(), pthread_testcancel().

15 Лекция
```
void pthread_exit(void *retval);
```

При компиляции и компоновке неоходимо использовать опцию -pthread.
Функция pthread_exit() завершает вызывающий поток и возвращает значение через retval, которое (если поток присоединяется) доступно другому потоку в том же процессе, который вызывает pthread_join(3). Если завершение потока не перехватить вызовом pthread_join() в результате получим зомби-поток. Если завершился отсоединенный поток, все будет нормально.

```
int pthread_cancel(pthread_t thread);
```

Функция pthread_cancel() отправляет запрос отмены потоку thread. Отреагирует ли целевой поток на запрос отмены, и когда, зависит от двух атрибутов, находящихся под контролем этого потока: его состояния отмены (cancelability state) и типа.

void pthread_testcancel(void) — Создает точку отмены внутри вызывающий поток, так что поток, кото-рый в противном случае выполняется код, который не содержит точек отмены, будет реагировать на за-прос на отмену. Если возможность отмены отключена (с помощью pthread_setcancelstate(3)), или за-прос на отмену не ожидается, то вызов pthread_testcancel() не действует.

60) Системные вызовы POSIX управления процессами getpgrp(), setpgrp(), getpgid(), setpgid(), getsid(), setsid().

pid_t getpgid(pid_t pid); Возвращает ID группы вызывающего процесса.

int setpgid(pid_t pid, pid_t pgid); Устанавливает PGID процесса, указанного в pid, равным pgid. Если pid равен нулю, то идентификатор процесса вызывающего процесса использовал. Если pgid равен нулю, то PGID процесса, указанного параметром pid делается таким же, как его идентификатор процесса.

В целом вызовы getpid и setpgid одинаковы с getpgrp и setpgrp. Отличия только в системе, на которой они были реализованы.

```
int setpgid(pid_t pid, pid_t pgid);
pid_t getpgid(pid_t pid);
pid_t getpgrp(void);                /* POSIX.1 version */
pid_t getpgrp(pid_t pid);           /* BSD version */
int setpgrp(void);                  /* System V version */
int setpgrp(pid_t pid, pid_t pgid); /* BSD version */
```

61) Стандартные сигналы. Базовая обработка и функция signal().

15 Лекция
С точки зрения пользователя получение процессом сигнала выглядит как возникновение прерывания. Процесс прекращает свое регулярное исполнение, и управление передается механизму обработки сиг-нала. По окончании обработки сигнала процесс может возобновить регулярное исполнение.
Процесс может получить сигнал от:
1) hardware (при возникновении исключительной ситуации);
2) другого процесса, выполнившего системный вызов передачи сигнала;
3) операционной системы (при наступлении некоторых событий);
4) терминала (при нажатии определенной комбинации клавиш);
5) системы управления заданиями при выполнении команды kill.
Передачу сигналов процессу в случаях его генерации источниками 2, 3 и 5, т.е. в конечном счете каким-либо другим процессом, можно рассматривать как реализацию в UNIX сигнальных средств связи.

Существует три варианта реакции процесса на сигнал:
1) принудительно проигнорировать сигнал;
2) произвести обработку по умолчанию (проигнорировать, остановить процесс (перевести в состояние ожидания до получения другого специального сигнала), либо завершить работу с образование core файла или без него);
3) выполнить обработку сигнала, указанную пользователем.

sighandler_t signal(int SIGNUM, sighandler_t ACTION) — установить обработчик сигнала

Функция signal() устанавливает ACTION как действие для сигнала SIGNUM.

SIGNUM — номер сигнала, который мы хотим контролировать.
Второй аргумент, ACTION, указывает действие, используемое для сигнала SIGNUM. Он может быть од-ним из следующих:
SIG_DFL — определяет действие по умолчанию для конкретного сигнала.
SIG_IGN — указывает, что сигнал следует игнорировать. (Сигналы SIGKILL или SIGSTOP не возможно игнорировать. Так же, игнорирование пользовательских запросов, таких как SIGINT, SIGQUIT и SIGTSTP, является неприемлемым.)
HANDLER — указывает адрес функции обработчика в программе, чтобы указать запуск этого обработ-чика, как способ доставки сигнала.

62) Расширенная обработка сигналов и функция sigaction()

7 Лекция
Функция sigaction() имеет тот же основной эффект, что и signal() — указывает, как должен обрабаты-ваться сигнал процессом. Однако sigaction() предлагает больший контроль за счет большей сложности. В частности, sigaction() позволяет указать дополнительные флаги для управления, когда генерируется сигнал и как вызывается обработчик.
Для определения всей информации о том, как обрабатывать конкретный сигнал, в функции sigaction() используются структуры типа struct sigaction.

```
int sigaction(  int sig,
        const struct sigaction *restrict act,
        struct sigaction *restrict oldact)
```

Аргумент act используется для установки нового действия для сигнала sig, а аргумент oldact использу-ется для возврата информации о действии, ранее связанном с этим сигналом.
Параметр oldact имеет ту же цель, что и возвращаемое значение функции signal() — можно проверить, какое старое действие действовало для сигнала, и, если необходимо, восстановить его позже.
Либо act, либо oldact могут быть нулевым указателем (NULL). Если нулевым указателем является oldact, это просто подавляет возврат информации о старом действии.
Если нулевым указателем является act, действие, связанное с сигналом sig, не изменяется.
Функция sigaction() возвращает: 0 в случае успеха и −1 в случае неудачи.

Структура sigaction определяется примерно так:

```
struct sigaction {
    void     (*sa_handler)(int); // тип функции-обработчика сигналов
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;// набор сигналов, которые должны быть заблокированы во
          время работы обработчика
    int      sa_flags;// Битовая маска флагов, которые могут влиять на
                      различное поведение сигналов
    void     (*sa_restorer)(void);
};
```

sa_sigaction - обработчик с тремя параметрами для флага SA_SIGINFO

```
void func( int signo,     // более продвинутый обработчик sa_sigaction
        siginfo_t *info, // Полезная нагрузка
        void *context); //
```

63) Искусственная генерация сигналов. Системные вызовы abort(), raise(), kill(), pthread_kill().

Сигналы можно посылать в процессы или потоки с помощью других функций.

Abort() — разблокировать SIGABRT и инициировать этот сигнал для вызывающего процесса (как если бы была вызвана функция raise(3)). Это приводит к аварийному завершению процесса, если только сиг-нал SIGABRT не перехватывается и обработчик сигнала не возвращается.

Raise(int sig) — функция посылает сигнал вызывающему процессу или потоку.
В однопоточных программах является эквивалентом kill(getpid(), sig). В многопоточных — pthread_kill(pthread_self(), sig).
Если сигнал вызывает вызов обработчика, функция raise() вернется только после того, как обработчик сигнала вернется.

Int kill(pid_t pid, int sig) — послать сигнал процессу.
Если значение pid является положительным, то сигнал sig посылается процессу с идентификатором pid.
Если значение pid равно 0, то sig посылается каждому процессу, который входит в группу вызывающего процесса.
Если значение pid равно -1, то sig посылается каждому процессу, которым вызывающий процесс имеет право отправлять сигналы, за исключением процесса с номером 1 (init).
Если значение pid меньше -1, то sig посылается каждому процессу, который входит в группу процессов, чей ID равен -pid.
Если значение sig равно 0, то никакой сигнал не посылается, но выполняется проверка существования и права.
Чтобы процесс мог посылать сигнал, он должен быть привилегированным, либо реальный или эффективный идентификатор пользователя посылающего процесса должен быть равен реальному или сохранённому идентификатору пользователя процесса, которому отправляется сигнал.

Int pthread_kill(pthread_t thread, int sig) — послать сигнал потоку. Отправляет sig сигнала в поток, поток в том же процессе, что и вызывающий. Сигнал асинхронно направляется в поток.

64) Сигналы в UNIX. Типы и жизненный цикл. Блокирование

6-7 Лекция
Типы сигналов (их принято задавать номерами, как правило, в диапазоне от 1 до 31 включительно или специальными символьными обозначениями) и способы их возникновения в системе жестко регламен-тированы.
Процесс может получить сигнал от:
1) hardware (при возникновении исключительной ситуации);
2) другого процесса, выполнившего системный вызов передачи сигнала;
3) операционной системы (при наступлении некоторых событий);
4) терминала (при нажатии определенной комбинации клавиш);
5) системы управления заданиями при выполнении команды kill.
Передачу сигналов процессу в случаях его генерации источниками 2, 3 и 5, т.е. в конечном счете каким-либо другим процессом, можно рассматривать как реализацию в UNIX сигнальных
средств связи.
Существует три варианта реакции процесса на сигнал:
1) принудительно проигнорировать сигнал;
2) произвести обработку по умолчанию (проигнорировать, остановить процесс (перевести в состояние ожидания до получения другого специального сигнала), либо завершить работу с образование core файла или без него);
3) выполнить обработку сигнала, указанную пользователем.
Изменить реакцию процесса на сигнал можно с помощью специальных системных вызовов.

С момента, как обработчик начинает выполнение, до момента его завершения, необходимо блокировать все сигналы, которые могут помешать его работе или испортить его данные.
Когда для сигнала вызывается функция-обработчик, этот сигнал в дополнение к любым другим сигна-лам, которые находятся в маске сигналов процесса, на время работы обработчика автоматически блоки-руется. Например, если установлен обработчик для «SIGTSTP», то прибытие этого сигнала заставляет дальнейшие сигналы «SIGTSTP» ожидать во время выполнения обработчика.
Однако другие виды сигналов по умолчанию не блокируются, поэтому они могут поступать во время выполнения обработчика.
Надежный способ заблокировать другие виды сигналов во время выполнения обработчика — использовать элемент «sa_mask» структуры «sigaction»
sigaddset(&block_mask, signum), где block_mask — маска, используемая для блокирования сигнала, а signum — номер сигнала.

65) Каналы и конвейеры FIFO. Характеристики и особенности их использования.

12 Лекция
Каналы и FIFO (также известные как именованные каналы) обеспечивают однонаправленный канал связи между процессами. Канал имеет конец для чтения и конец для записи. Данные, записанные в ко-нец канала для записи, могут быть прочитаны из конца канала для чтения.
Канал создается с помощью pipe(2), которая создает новый канал и возвращает два файловых дескрип-тора, один из которых относится к концу канала для чтения, а другой — к концу для записи. Каналы можно использовать для создания канала связи между связанными процессами.
Любой процесс может открыть FIFO, если права доступа к файлу позволяют это сделать.
Чтобы FIFO работал, он должен быть открыт как на запись, так и на чтение. Конец для чтения открывается с помо-щью флага O_RDONLY. Конец для записи открывается с помощью флага O_WRONLY.
Если процесс попытается прочитать из пустого канала, read(2) заблокируется до тех пор, пока в канале не станут доступны данные.
Если процесс пытается записать в заполненный канал (см. ниже), то блокируется write(2) до тех пор, пока из канала не будет прочитано достаточно данных, чтобы запись можно было завершить.
Канал имеет ограниченную емкость. Если канал заполнен, то write(2) заблокируется или завершится ошибкой, в зависимости от того, установлен ли флаг O_NONBLOCK. Различные реализации имеют разные ограничения на емкость канала.

67) Системные вызовы fork(), exec..(), exit() и их связь с функцией main()

3 Лекция
pid_t fork(void);
fork создает процесс-потомок, который отличается от родительского только значениями PID (идентификатор процесса) и PPID (идентификатор родительского процесса), а также тем фактом, что счетчики использования ресурсов установлены в 0.
Блокировки файлов и сигналы, ожидающие обработки, не наследуются.
Int exec..(const char *filename,
char *const argv[], ..); - выполняет программу, передавая в неё аргументы и окружение (если надо). Exec..() не возвращает управление при успешном выполнении, а код, данные, bss и стек вызвавшего процесса перезаписываются кодом, данными и стеком загруженной программы.
_Noreturn void exit(int status);
Функция exit() вызывает обычное завершение программы.
Функции, зарегистрированные функцией at_quick_exit(), не вызываются.

Если программа вызывает функцию выхода более одного раза или вызывает функцию quick_exit() в до-полнение к функции выхода, поведение не определено. Сначала вызываются все функции, зарегистри-рованные функцией atexit(), в порядке, обрат-ном их регистрации.
Для корректного вызова программы из другой программы следует выполнить следующие действия:
1) Переопределить прерывания, вызываемые в child-программе.
2) Создать процесс-потомок, в котором будет выполняться другая программа.
3) Перейти в режим ожидания main-процессом и вызвать другую программу child-процессом.
4) Ждать прерывания и реагировать на них.

68) Очистка при завершении программы. Функции стандартной библиотеки exit() и atexit().

3 Лекция
int atexit(void (*func)(void));
Функция atexit() регистрирует функцию, на которую указывает func, для вызова без аргумен-тов при нормальном завершении программы.
_Noreturn void exit(int status);
Функция exit() вызывает обычное завершение программы.
Функции, зарегистрированные функцией at_quick_exit(), не вызываются.
Если программа вызывает функцию выхода более одного раза или вызывает функцию quick_exit() в до-полнение к функции выхода, поведение не определено.
Сначала вызываются все функции, зарегистрированные функцией atexit(), в порядке, обрат-ном их регистрации.
При завершении программы все открытые файлы закрываются.

69) Алгоритмы замещения страниц.

23 Лекция (rip)
Алгоритмы замещения страниц
При возникновении ошибки отсутствия страницы операционная система должна выбрать выгружаемую (удаляемую из памяти) страницу, чтобы освободить место для загружаемой страницы.

Оптимальный (невозможный) алгоритм замещения страниц
На момент возникновения ошибки отсутствия страницы в памяти находится определенный набор стра-ниц. К некоторым из этих страниц будет осуществляться обращение буквально из следующих команд (например, эти команды содержатся на странице). К другим страницам обращения может не быть и че-рез 10, 100 или даже 1000 команд. Каждая страница может быть помечена количеством команд, которые должны быть выполнены до первого обращения к странице. Оптимальный алгоритм замещения стра-ниц гласит, что должна быть удалена страница, имеющая пометку с наибольшим значением. Если ка-кая-то страница не будет использоваться на протяжении двадцати млн. команд, а другая какая-нибудь страница не будет использоваться на протяжении пяти млн. команд, то удаление первой из них приве-дет к ошибке #PF в самом отдаленном будущем.
Проблема: невозможность его реализации — к тому времени, когда произойдет ошибка #PF, у опера-ционной системы нет способа узнать, когда каждая из страниц будет востребована в следующий раз.

Алгоритм FIFO (First In, First Out — «первым пришел, первым ушел»)
Выталкивание первой пришедшей (самой старой) страницы
Простейший алгоритм. Каждой странице присваивается временная метка. Реализуется это просто со-зданием очереди страниц, в конец которой страницы попадают, когда загружаются в

физическую па-мять, а из начала берутся, когда требуется освободить память. Для замещения выбирается самая старая страница. В случае замещения активных страниц все сработает корректно, но тут же за этим сразу по-следует #PF. Поэтому принцип FIFO в чистом виде используется довольно редко.

Алгоритм NRU (Not Recently Used)
Замещается не использовавшаяся в последнее время страница.
Чтобы позволить операционной системе осуществить сбор полезной статистики востребованности страниц, большинство компьютеров, использующих виртуальную память, имеют два бита состояния R и M, связанных с каждой страницей.
Бит R устанавливается при каждом обращении к странице (при чтении или записи). Бит M устанавли-вается, когда в страницу ведется запись (то есть когда она модифицируется).
Эти биты присутствуют в каждой записи таблицы страниц и они должны обновляться при каждом об-ращении к памяти, поэтому необходимо, чтобы их значения устанавливались аппаратной частью.
После установки бита в 1 он сохраняет это значение до тех пор, пока не будет сброшен операционной системой.

Алгоритм «вторая попытка»
Подобен FIFO, но если R=1, то страница переводится в конец очереди, если R=0, то страница выгружа-ется.
В таком алгоритме часто используемая страница никогда не покинет память. Но в этом алгоритме при-ходится часто перемещать страницы по списку. Заключается в проверке бита R самой старой страницы.
Если его значение равно нулю, значит, страница не только старая, но и невостребованная, поэтому она тут же удаляется. Если бит R имеет значение 1, он сбрасывается, а страница помещается в конец списка страниц и время ее загрузки обновляется, как будто она только что поступила в память. Затем поиск продолжается.

Алгоритм «часы»
При возникновении ошибки отсутствия страницы #PF проверяется та страница, на которую указывает стрелка.
Если ее бит R имеет значение 0, страница выгружается, на ее место в списке вставляется новая страни-ца и указатель смещается вперед на одну страницу. Если значение бита R равно 1, то он сбрасывается и указатель перемещается на следующую страницу. Этот процесс повторяется до тех пор, пока не будет найдена страница с R = 0.

Алгоритм LRU (Least Recently Used)
Выталкивание дольше всего не использовавшейся страницы.
Алгоритм позволяет сократить количество страничных нарушений. В основе лежит использование прошлого для аппроксимации (экстраполяции) будущего — замещается страница, которая не использо-валась в течение самого долгого времени.

Алгоритм NFU (Not Frequently Used)
Выталкивание редко используемой страницы.
Алгоритм NFU(Not Frequently Used) — программная реализация алгоритма, близкого к LRU. Для него требуются программные счетчики, по одному на каждую страницу, которые сначала равны нулю. При каждом прерывании по времени операционная система сканирует все страницы в памяти и у каждой страницы с установленным флагом обращения увеличивает на единицу значение счетчика, а флаг об-ращения сбрасывает.

70) Спинлоки и их использование.

17 Лекция
Большинство программ должны использовать мьютексы вместо спин-блокировок. Спин-блокировки в первую очередь полезны в сочетании с политиками планирования в реальном вре-мени.

pthread_spin_init()/destroy() — инициализировать/уничтожить спинлок
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
int pthread_spin_destroy(pthread_spinlock_t *lock);
Поддержка спин-блокировок с совместным использованием несколькими процессами — это опция POSIX. В реализации glibc эта опция поддерживается

int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
Функция pthread_spin_lock() блокирует спин-блокировку, на которую ссылается lock. Если спин-блокировка в настоящее время разблокирована, вызывающий поток получает блокировку немедленно.
Если спин-блокировка в настоящее время заблокирована другим потоком, вызывающий поток «враща-ется», проверяя блокировку, пока она не станет доступной, после чего вызывающий поток получает блокировку.

Спинлоки применяются для синхронизации небольших участков кода, когда использование более сложных механизмов неоправданно или невозможно. Реализация примитивов синхронизации и дис-петчера потоков обязательно требует блокировок, защищающих списки потоков, готовых к исполне-нию, и списки потоков, ожидающих на объектах.

71) Блокировки чтения/записи и их использование в программах. Протокол обновления файла в много-пользовательской среде.

17 Лекция
Прежде чем можно пользоваться блокировкой rwlock, она должна быть инициализирована. Для иници-ализации используется вызов rwlock_init(), который инициализирует блокировку чтения-записи, на ко-торую ссылается объект rwlock, с атрибутами, на которые ссылается объект attr. Как всегда, если объект attr имеет значение NULL, используются атрибуты блокировки чтения-записи по умолчанию. Блоки-ровку можно использовать любое количество раз без повторной инициализации.
Вызов rwlock_destroy() уничтожает объект блокировки чтения-записи, на который ссылается объект rwlock, и освобождает все ресурсы, используемые блокировкой. Уничтоженный объект блокировки чте-ния-записи может быть повторно инициализирован с помощью вызова pthread_rwlock_init().

Вызов rwlock_rdlock() выполняет блокировку чтения объекта, на который ссылается rwlock.
1) Вызывающий поток получает блокировку чтения, если эту блокировку не удерживает «писатель» и нет ни одного «писателя», заблокированного в ее ожидании.
2) Получит ли вызывающий поток блокировку, когда «писатель» не удерживает блокировку, но есть «писатели», ее ожидающие, не указано — это прерогатива реализации;
3) Если блокировку удерживает «писатель», вызывающий поток блокировку чтения не получит.
Если блокировка чтения не установлена сразу, вызывающий поток не возвращается из вызова rwlock_rdlock(), пока не получит блокировку.

72) Условные переменные и их использование потоками — ожидание и сигнализация.

18 Лекция
Условная переменная — примитив синхронизации, обеспечивающий блокирование одного или не-скольких потоков в состоянии ожидания выполнения некоторого условия до момента поступления сиг-нала от другого потока о том, что данное условие выполнено (или до истечения максимального проме-жутка времени ожидания).
Условные переменные используются вместе с ассоциированным с ними мьютексом и фактически пред-ставляют собой совокупность объекта синхронизации cond, предиката P и мьютекса mutex. Типичный шаблон использования переменных состояния:

```
// безопасно проверим состояние/условие предиката SOME-CONDITION, чтобы защитить в
// в этот момент предикат от изменения другими потоками захватываем мьютекс
lock(mutex);
while (SOME-CONDITION is false) {
    wait(cond, mutex);  // блокируемся на переменной cond, mutex, связанный с ней
}                       // при этом освобождается и может быть получен другим потоком
do_stuff();             // нас разблокировали и вернули мьютекс — делаем наши дела
unlock(mutex);      // и отдаем мьютекс
```

С другой стороны, поток, сигнализирующий о переменной условия, обычно выглядит так:
lock(mutex); //нужен эксклюзивный доступ, каким бы условие не было
ALTER-CONDITION// делаем наши дела и меняем предикат состояния, после чего
signal(cond); // разбудим по крайней мере один поток из ожидающих изменения состояния и заблоки-рованных на условной переменной cond
unlock(mutex)   // отдаем мьютекс

Концептуально, условная переменная — это очередь потоков, ассоциированных с совместно использу-емым объектом данных, которые ожидают выполнения некоторого условия, накладываемого на состоя-ние данных.

Сигнализация
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond); Эти функции разблокируют потоки, заблокированные на условной переменной. Функция pthread_cond_broadcast() разблокирует все потоки, заблокированные в данный момент для указанной условной переменной cond. Функция pthread_cond_signal() разблокирует по крайней мере один из потоков, которые заблокированы по указанной условной переменной cond (ес-ли есть какие-либо потоки, которые заблокированы по cond).

Ожидание
Ожидание условия, будь то ограниченное по времени или нет, является точкой отмены – функции pthread_cond_[timed]wait() являются точками, где может быть замечен запрос отмены.
Причина такого положения состоит в том, что в этих точках возможно неопределенное ожидание — ка-кое бы событие ни ожидалось, даже если программа полностью верна, оно может никогда не произой-ти.

73) Методы ввода/вывода — блокирующий, неблокирующий, мультиплексированный и асинхронный.

19 Лекция
Блокирующий метод

Любая пользовательская программа запускается внутри процесса, а код выполняется в контексте потока. Предположим, мы пишем программу, которой нужно читать информацию из файла. С блокирующим вводом-выводом мы просим ОС «усыпить» читающий поток и «разбудить» его после того, как данные из файла будут доступны для чтения.

Неблокирующий метод
Иногда программе больше и не надо ничего делать. В противном же случае во время ожидания ввода-вывода было бы полезно выполнять другие задачи. Один из способов осуществить это — использовать неблокирующий ввод-вывод. Идея заключается в том, что когда программа делает вызов на чтение фай-ла, ОС не будет блокировать поток, а просто вернёт программе либо готовые данные, либо информацию о том, что ввод-вывод ещё не завершен. В этом случае поток не будет заблокирован, но программе при-дётся позже проверять, завершён ли ввод-вывод. Это означает, что можно по-разному реагировать в за-висимости от того, завершён ли ввод-вывод и выполнять другие задачи.
Когда же программе снова понадобится ввод-вывод, она сможет повторно попробовать прочесть содер-жимое файла, и если ввод-вывод завершён, то получит содержимое файла. В противном случае ПО сно-ва получит сообщение о том, что операция ещё не завершена и сможет заняться другими задачами.

Мультиплексированный метод
Проблема с неблокирующим вводом-выводом в том, что с ним не удобно работать, если задачи, которые выполняет программа, ожидая ввода-вывода, сами из себя представляют ввод-вывод. Во избежание про-блем можно использовать мультиплексированный ввод-вывод. Он тоже блокирует поток на операциях ввода-вывода, но вместо того, чтобы производить блокировку по очереди, можно запланировать все опе-рации ввода-вывода, которые нужно сделать, чтобы продолжить выполнение, и блокировать их все. Операционная система разбудит поток, когда какая-нибудь из операций завершится. В некоторых реали-зациях мультиплексированного ввода-вывода можно даже точно указать, чтобы поток был разбужен только тогда, когда будет завершён заданный набор операций ввода-вывода, например, когда будут гото-вы файлы A и C, или файлы B и D.

Асинхронный метод
Проблема мультиплексированного ввода-вывода в том, что поток всё-таки спит, пока ввод-вывод не бу-дет готов для обработки. Например читаем файл и вычисляем цифры числа π. К огда какой-нибудь из файлов будет прочитан, ПО выполнит обработку и продолжит вычисл ять цифры числа π дальше, пока ещё один файл не будет прочитан.Чтобы это работало, нужн о, чтобы вычисление цифр числа π могло быть прервано вводом-выводом, когда он завершает ся. Это можно сделать с помощью обратных вызо-вов, связанных с событиями. Вызов на чтение принимает функцию обратного вызова и возвращается немедленно.
Когда ввод-вывод завершается, операционная система остановит ваш поток и выполнит обратный вызов. Когда обратный вызов завершится, система возобновит работу вашего потока.

74) Циклы опроса poll и select. Функции обратного вызова и очереди завершения.

19 Лекция
Опрос (Polling)
Опрос предоставляет неблокирующий синхронный API, который можно использовать для реализации некоторого асинхронного API.
Доступно в традиционных Unix и Windows. Его основная проблема заключается в том, что он тратит время ЦП на многократные опросы,
если процессу, инициировавшему ввод/вывод больше нечего делать — это сокращает время, доступное для других процессов.

Кроме того, поскольку приложение для опроса по существу является однопоточным, оно может быть не в состоянии полностью использовать параллелизм ввода-вывода, на который способно оборудование.

Циклы select(poll)
Цикл select использует системный вызов select() для перехода в спящий режим до тех пор, пока:
- в дескрипторе файла не возникнет условие, например, станут доступны для чтения данные;
- не истечет время ожидания;
- не будет получен сигнал (например, когда дочерний процесс завершается).
Возвращаемые параметры вызова select(), анализируются в цикле и определяется, какой дескриптор файла был изменен, и выполняется соответствующий код.
Цикл select основан на предположении, что можно задействовать весь ввод/вывод в одном вызове select(), и это может создавать проблемы при использовании библиотек, которые выполняют собственный ввод-вывод.

Функции обратного вызова (Callback functions)
Ввод-вывод выполняется асинхронно, и по его завершении генерируется сигнал. Каждый запрос ввода-вывода обычно может иметь свою собственную функцию завершения, тогда как сигнальная система имеет единственный обратный вызов.
Потенциальная проблема использования обратных вызовов заключается в том, что стек может неуправ-ляемо расти, поскольку очень часто после завершения одного ввода-вывода планируется другой.
На практике, однако, это обычно не проблема, потому что новая операция ввода/вывода обычно возвра-щается, как только новый ввод/вывод запускается, что позволяет стеку "раскручиваться".
Проблему также можно предотвратить, избегая дальнейших обратных вызовов с помощью очереди до тех пор, пока не вернется первый обратный вызов.

Очереди завершения/порты (Completion queues/ports)
Запросы ввода-вывода отправляются асинхронно, но уведомления о завершении предоставляются через механизм очереди синхронизации в порядке их выполнения. Обычно ассоциируется с конечным авто-матом (программирование, управляемое событиями), что может создать трудности программирования процессов, не использующих асинхронный ввод-вывод или использующих одну из других форм. Не требует дополнительных специальных механизмов синхронизации или поточно-ориентированных биб-лиотек, а также не разделены текстовые (код) и временные (события) потоки.

75) Асинхронный ввод/вывод в POSIX и уведомления о завершении операций

19 Лекция
Интерфейс POSIX AIO состоит из следующих функций:

```
int aio_read(struct aiocb *aiocbp);
```
Функция aio_read() ставит в очередь запрос ввода-вывода, описанный в буфере, на который указывает aiocbp. Эта функция является асинхронным аналогом вызова read(2).

```
int aio_write(struct aiocb *aiocbp);
```
Функция aio_write() ставит в очередь запрос ввода-вывода, описанный в буфере, на который указывает aiocbp. Эта функция является асинхронным аналогом вызова write(2).

```
int lio_listio(  int mode,
        struct aiocb *const aiocb_list[],
```

int nitems,
struct sigevent *sevp);
Функция lio_listio() запускает на выполнение список операций ввода-вывода, описанных в массиве aiocb_list.

int aio_error(const struct aiocb *aiocbp);
Функция aio_error() возвращает состояние ошибки запроса асинхронного ввода-вывода для указанного блока управления aiocbp.

ssize_t aio_return(struct aiocb *aiocbp);
Функция aio_return() возвращает окончательное значение завершения запроса асинхронного ввода-вывода, задаваемого указателем на контрольный блок aiocbp.

Структура sigevent используется в различных программных интерфейсах для описания способа, кото-рым нужно уведомлять процесс о событии (например, окончание асинхронного запроса, истечение таймера или поступление сообщения).

struct sigevent {
int sigev_notify; // метод уведомления
int sigev_signo; // сигнал уведомления
union sigval sigev_value; // данные, передаваемые с уведомлением
(SIGEV_THREAD) // функция, используемая для нити уведомления
void (*sigev_notify_function)(union sigval);
void *sigev_notify_attributes; // атрибуты для нити уведомления (SIGEV_THREAD)
pid_t sigev_notify_thread_id; // ID нити для уведомления (SIGEV_THREAD_ID)
};
Уведомления приходят по окончании ввода/вывода и являются сигнализацией окончания.

76) Основной интерфейс асинхронного ввода/вывода в POSIX и списки запросов.

19 Лекция
Интерфейс POSIX AIO состоит из следующих функций:

int aio_read(struct aiocb *aiocbp);
Функция aio_read() ставит в очередь запрос ввода-вывода, описанный в буфере, на который указывает aiocbp. Эта функция является асинхронным аналогом вызова read(2).

int aio_write(struct aiocb *aiocbp);
Функция aio_write() ставит в очередь запрос ввода-вывода, описанный в буфере, на который указывает aiocbp. Эта функция является асинхронным аналогом вызова write(2).

int lio_listio(int mode,
struct aiocb *const aiocb_list[],
int nitems,
struct sigevent *sevp);
Функция lio_listio() запускает на выполнение список операций ввода-вывода, описанных в массиве aiocb_list.
Значение операции mode может быть одним из следующих:
LIO_WAIT — вызов не завершается до тех пор, пока не будут выполнены все операции. Аргумент sevp при этом игнорируется.
LIO_NOWAIT — операции ввода-вывода ставятся в очередь на обработку и вызов сразу завершается. После выполнения всех операций ввода-вывода посылается асинхронное уведомление, задаваемое в ар-гументе sevp. Если значение sevp равно NULL, то асинхронные

уведомления не посылаются. Аргумент aiocb_list представляет собой массив указателей на структуры aiocb, в которых описаны операции вво-да-вывода. Эти операции выполняются в произвольном порядке

int aio_error(const struct aiocb *aiocbp);
Функция aio_error() возвращает состояние ошибки запроса асинхронного ввода-вывода для указанного блока управления aiocbp.

ssize_t aio_return(struct aiocb *aiocbp);
Функция aio_return() возвращает окончательное значение завершения запроса асинхронного ввода-вывода, задаваемого указателем на контрольный блок aiocbp.

77) Сокеты — основные концепции, типы и адреса

20 Лекция
Когда создается сокет, необходимо указать тип связи, который предполагается использовать, и тип про-токола, который должен ее реализовать.
Тип связи
«Тип (стиль) связи» сокета определяет семантику отправки и получения данных на уровне пользовате-ля через сокет. Выбор типа связи определяет ответы на следующие вопросы:
- Каковы единицы передачи данных?
- Могут ли данные быть потеряны во время нормальной работы?

Тип протокола
Для осуществления связи необходимо выбрать «протокол».
Протокол определяет, какой низкоуровневый механизм используется для передачи и приема данных. Нужно знать о протоколах следующее:
1) Для связи между двумя сокетами они должны указать один и тот же протокол.
2) Каждый протокол имеет смысл с определенными комбинациями стиля/пространства имен и не может использоваться с несоответствующими комбинациями. Например, протокол TCP соот-ветствует только стилю связи с потоком байтов и пространству имен Internet.
3) Для каждой комбинации стиля и пространства имен существует «протокол по умолчанию», который можно запросить, указав 0 в качестве номера протокола. И это то, что обычно следует делать — исполь-зовать в качестве номера протокола значение по умолчанию.

SOCK_STREAM
Тип SOCK_STREAM похож на канал (pipes и FIFO). Он работает через соединение с конкретным уда-ленным сокетом и надежно передает данные в виде потока байтов.
SOCK_DGRAM
Тип SOCK_DGRAM используется для ненадежной отправки пакетов с индивидуальной адресацией. Это полная противоположность SOCK_STREAM.
SOCK_RAW
Этот тип обеспечивает доступ к сетевым протоколам и интерфейсам низкого уровня. Обычные пользо-вательские программы обычно не нуждаются в использовании этого стиля.
SOCK_SEQPACKET
Обеспечивает работу последовательного двустороннего канала для передачи дейтаграмм на основе со-единений. Дейтаграммы имеют постоянный размер. От получателя требуется за один раз прочитать це-лый пакет.
SOCK_RDM
Обеспечивает надежную доставку дейтаграмм без гарантии, что они будут расположены по порядку.

Имя сокета обычно называется «адресом».
Функции и символы для работы с адресами сокетов были названы непоследовательно, иногда с использованием термина «имя», а иногда с использованием термин «адрес». Когда речь идет о сокетах, можно рассматривать эти термины как синонимы.

78) Пять основных системных вызовов для работы с сокетами и их использование.

20 Лекция
```
int socket(    int socket_family, // семейство протоколов, которое будет использоваться
        int socket_type,   // семантика соединения (stream, datagram, raw, ...)
        int protocol);     // протокол, используемый с сокетом

int bind( int sockfd,     // файловый дескриптор сокета const struct sockaddr *addr,    // адрес сокета sock-len_t addrlen); // длина адреса

int listen( int sockfd,    // файловый дескриптор сокета int backlog); // максимальный размер очереди ожидающих соединений

int accept(int sockfd,    // файловый дескриптор слушающего сокета struct sockaddr *addr,// адресс соке-та или NULL socklen_t *addrlen);    // длина адреса или же
int connect(int sockfd,  // сокет инициатора соединения const struct sockaddr *addr, // адрес соедине-ния socklen_t addrlen);      // размер addr

send()/recv()
```
На стороне сервера:
1) Вызывается socket для создания объекта и связанного с ним файлового дескриптора.
2) Вызывается bind, для назначения сокету адреса (имени) .
3) Вызывается listen, чтобы пометить сокет, как предназначенный для приема запросов на соединение.
4) Вызывается accept, чтобы дождаться и принять входящее соединение. После этого вызов accept созда-ет второй сокет с новым файловым дескриптором, который и используется для обмена данными.
5) В операциях ввода-вывода (системные вызовы read и write) участвует созданный (второй) файловый дескриптор.

На стороне клиента:
1) Вызывается socket для создания объекта и связанного с ним файлового дескриптора.
2) Для установления соединения с сервером вызывается connect, которому в качестве аргумента переда-ется имя сервера, (этот шаг, не обязательно должен быть синхронизирован со вторым шагом на стороне сервера).
3) Файловый дескриптор сокета используется для выполнения операций ввода-вывода.

79) Ввод/вывод с сокетами, функции отправки и получения сообщений.

21 Лекция
Ввод/вывод осуществляется путём отправки и получения сообщений. Сокет отправляет сообщение, дру-гой же сокет изначально ждёт принятия сообщения вызовом recv(). Ожидает готовности операций вво-да-вывода

Вызов select() позволяет программам отслеживать изменения нескольких файловых дескрипторов ожи-дая, когда один или более файловых дескрипторов станут «готовы» для операции ввода-вывода опреде-лённого типа (например, ввода).
```
Int select(    int nfds,       // количество всех файловых дескрипторов
        fd_set *readfds,   // набор дескрипторов для чтения
```

```
        fd_set *writefds,  // набор дескрипторов для записи
        fd_set *exceptfds, // набор дескрипторов для обнаружения
                           // исключительных условий.
        Struct timeval *timeout); // время ожидания

ssize_t send( int sockfd,          // отправляющий сокет
        const void *buf,   // буфер с данными
        size_t len,        // длина данных
        int flags); //
```

Системные вызовы send(), sendto() и sendmsg() используются для пересылки сообщений в другой сокет. Вызов send() можно использовать, только если сокет находится в состоянии соединения, то есть если известен получатель.

```
Ssize_t recv( int sockfd, // получающий сокет void *buf,     // буфер с данными size_t len, //
длина данных int flags);
```

Системные вызовы recv(), recvfrom() и recvmsg() используются для получения сообщений из сокета. Они могут использоваться для получения данных, независимо от того, является ли сокет ориентирован-ным на соединения или нет. Вызов recv() отличается от read(2) только наличием аргумента flags. Если значение flags равно нулю, то вызов recv() эквивалентен read(2) с некоторыми отличиями.

80) Сокеты для локального межпроцессного взаимодействия AF_LOCAL, три типа адреса.

21 Лекция
AF_UNIX — сокеты для локального межпроцессного взаимодействия

```
unix_socket  = socket(PF_UNIX, type, 0);
error        = socketpair(PF_UNIX, type, 0, int *sv);
```

Семейство сокетов AF_UNIX (AF_LOCAL) используется для эффективного взаимодействия между про-цессами на одной машине.
Допустимые типы сокета для домена UNIX:

SOCK_STREAM — потоковый сокет;
SOCK_DGRAM — датаграмный сокет, сохраняющий границы сообщений в большинстве реализаций UNIX. Датаграмные сокеты домена UNIX всегда надёжны и не меняют порядок датаграмм.
SOCK_SEQPACKET — сокет, ориентированный на соединение. Задает последовательность пакетов, со-храняет границы сообщений и доставляет сообщения в том же порядке, в каком они были отправлены.

Адрес доменного сокета UNIX представляет собой следующую структуру:

```
struct sockaddr_un {
    sa_family_t sun_family; // AF_UNIX
    char sun_path[108]; // имя пути
};
```

В sockaddr_un структуре различают три типа адресов:
1) с именем пути (путевые сокеты) — доменный сокет UNIX может быть привязан к имени пути (с завершающимся null) в файловой системе с помощью bind(2). При возврате адреса имени пути сокета (одним и системных вызовов, упомянутых выше), его длина равна

offsetof(struct sockaddr_un, sun_path)4 + strlen(sun_path) + 1 и sun_path содержит путь, оканчивающийся нулем.
В Linux, указанное выше выражение offsetof() равно sizeof(sa_family_t), но в некоторых реализациях включаются другие поля перед sun_path, поэтому выражение offsetof() описывает размер адресной структуры более переносимым способом).
4) offsetof(type, member)2) безымянный — потоковый сокет, который не привязан к имени пути с по-мощью bind(), не имеет имени.
Два сокета, создаваемые socketpair(), также не имеют имён. При возврате адреса сокета его длина равна sizeof(sa_family_t), а значение sun_path не используется.
3) абстрактный: абстрактный адрес сокета отличается (от имени пути сокета) тем, что значением sun_path[0] является байт \0.

81) Безымянные сокеты AF_LOCAL и функция socketpair(), путевые сокеты.

21 Лекция
Безымянный сокет — потоковый сокет, который не привязан к имени пути с помощью bind(), не имеет имени.
Два сокета, создаваемые socketpair(), также не имеют имён. При возврате адреса сокета его длина равна sizeof(sa_family_t), а значение sun_path не используется.
Int socketpair(int domain, int type, int protocol, int sv[2]);
Вызов socketpair() создает пару неименованых присоединённых сокетов в заданном семействе адреса-ции (домене) domain заданного типа взаимодействия type, используя (при необходимости) заданный протокол protocol. Данные аргументы имеют тот же смысл и значения, что и для системного вызова socket(2). Файловые дескрипторы, используемые как ссылки на новые сокеты, возвращаются в sv[0] и sv[1]. Никаких различий между этими двумя сокетами нет.

Путевые сокеты — доменный сокет UNIX может быть привязан к имени пути (с завершающимся null) в файловой системе с помощью bind(2).
При привязке сокета к пути для максимальной переносимости и простоте кодирования нужно учесть несколько правил:
- Имя пути в sun_path должно завершаться null.
- Длина имени пути, включая завершающий байт null, не должна превышать размер sun_path.
- Аргумент addrlen, описывающий включаемую структуру sockaddr_un, должен содержать значение, как минимум: offsetof(struct sockaddr_un, sun_path) + strlen(addr.sun_path) + 1 или, проще говоря, для addrlen можно использовать sizeof(struct sockaddr_un).

82) Вспомогательные сообщения, их передача и прием с использованием функций *msg().

21 Лекция
Вспомогательные сообщения
Вспомогательные данные отправляются и принимаются с помощью sendmsg(2) и recvmsg(2). В силу исторических причин перечисленные типы вспомогательных сообщений относятся к типу SOL_SOCKET, даже если они относятся к AF_UNIX.
Для того, чтобы их отправить, следует установить значение поля cmsg_level структуры cmsghdr равным SOL_SOCKET, а в значении поля cmsg_type указать его тип.
Дополнительная информация приведена в cmsg(3).
SCM_RIGHTS
Передать или принять набор открытых файловых дескрипторов из другого процесса. Часть с данными содержит целочисленный массив файловых дескрипторов.
Эта операция эквивалентна дублированию (dup(2)) файлового дескриптора в таблицу файловых де-скрипторов другого процесса.
SCM_CREDENTIALS

Передать или принять учётные данные UNIX. Может быть использована для аутентификации. Учётные данные передаются в виде структуры struct ucred вспомогательного сообщения.
SCM_SECURITY
Получить контекст безопасности SELinux (метку безопасности) однорангового сокета. Полученные вспомогательные данные представляют собой строку (с null в конце) с контекстом безопасности.
Получатель должен выделить не менее NAME_MAX байт под эти данные в в части данных вспомога-тельного сообщения.

При отправке вспомогательных данных с помощью sendmsg(2) посылаемое сообщение может содер-жать только по одному элементу каждого типа, из представленных выше. При отправке вспомогатель-ных данных по крайней мере должен быть отправлен один байт реальных данных.
При отправке вспомогательных данных через дейтаграммный доменный сокет UNIX в Linux необяза-тельно отправлять какие-либо реальные сопровождающие данные. Однако переносимые приложения должны также включать, по крайней мере, один байт реальных данных при отправке вспомогательных данных через дейтаграммный сокет.
При получении из потокового сокета вспомогательные данные формируют своего рода барьер для по-лученных данных.

83) Права на каталоги и файлы, функции umask(), chown() и chmod().

21 Лекция
mode_t umask(mode_t mask); Системный вызов umask() устанавливает в вызывающем процессе значе-ние маски (umask) режима доступа к файлу равным mask & 0777 (т.е. из mask используются только би-ты прав доступа к файлу) и возвращает предыдущее значение маски. Значение umask используется в open(2), mkdir(2) и других системных вызовах, которые создают файлы, для изменения прав, назначае-мых на создаваемые файлы или каталоги.
int chown(const char *pathname, uid_t owner, gid_t group); chown() изменяет владельца для файла, зада-ваемого параметром pathname, который разыменовывается, если является символьной ссылкой.
Только привилегированный процесс может сменить владельца файла. Владелец файла может сменить группу файла на любую группу, в которой он числится. Привилегированный процесс может задавать произвольную группу.
int chmod(const char *pathname, mode_t mode); Вызов chmod() изменяет режим файла, задаваемого пу-тём из параметра pathname, который разыменовывается, если является символьной ссылкой. Новый ре-жим файла указывается в mode и представляет собой битовую маску, создаваемую побитовым сложени-ем нуля или более следующих констант — S_ISUID, S_ISGID, S_ISVTX, S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH, S_IXOTH.

84) Семиуровневая модель взаимодействия открытых систем.

21 Лекция
Называется «The Open Systems Interconnection model» или «семиуровневая модель». Основана на эта-лонной модели взаимосвязи открытых систем, разработанной Международной организацией по стан-дартизации ISO и принята в качестве международного стандарта ISO/IEC 7498.

Уровни модели OSI

Уровень (layer)		Тип данных (PDU[1])	Функции	Примеры	Оборудование
Уровни хоста[2]	7. Прикладной (application)	Данные	Доступ к сетевым службам	HTTP, FTP, SSH POP3, WebSocket	Хосты (клиенты сети)
	6. Представления (presentation)		Представление и шифрование данных	ASCII, EBCDIC	
	5. Сеансовый (session)		Управление сеансом связи	RPC, PAP, L2TP	
	4. Транспортный (transport)	Сегменты/Датаграммы (segment/datagram)	Прямая связь между конечными пунктами и надёжность	TCP, UDP, SCTP, PORTS	
Уровни среды передачи[3]	3. Сетевой (network)	Пакеты (packet)	Определение маршрута и логическая адресация	IPv4, IPv6, IPsec, AppleTalk, ICMP	Маршрутизатор
	2. Канальный (data link)	Биты (bit)/ Кадры (frame)	Физическая адресация	PPP, IEEE 802.22, Ethernet, DSL, ARP, сетевая карта.	Коммутатор, точка доступа
	1. Физический (physical)	Биты (bit)	Работа со средой передачи, сигналами и двоичными данными	USB, кабель («витая пара», коаксиальный, оптоволоконный), радиоканал	Концентратор Повторитель (сетевое оборудование)

1) PDU — сокращение от англ. protocol data units, единица измерения информации (данных), которой оперирует протокол.
2) Host layers – уровни хоста
3) Media layers

85) Общие функции уровней семиуровневой модели

21 Лекция

86) Уровни хоста семиуровневой модели, типы данных и функции.

21 Лекция

Уровень (layer)		Тип данных (PDU[1])	Функции	Примеры	Оборудование
Уровни хоста[2]	7. Прикладной (application)	Данные	Доступ к сетевым службам	HTTP, FTP, SSH POP3, WebSocket	Хосты (клиенты сети)
	6. Представления (presentation)		Представление и шифрование данных	ASCII, EBCDIC	
	5. Сеансовый (session)		Управление сеансом связи	RPC, PAP, L2TP	
	4. Транспортный (transport)	Сегменты/Датаграммы (segment/datagram)	Прямая связь между конечными пунктами и надёжность	TCP, UDP, SCTP, PORTS	

Прикладной уровень
Прикладной уровень (уровень приложений; англ. application layer) — верхний уровень модели, обеспе-чивающий взаимодействие пользовательских приложений с сетью:
позволяет приложениям использовать сетевые службы:
- удалённый доступ к файлам и базам данных,
- пересылка электронной почты;
- отвечает за передачу служебной информации;
- предоставляет приложениям информацию об ошибках;
- формирует запросы к уровню представления.

Уровень представления (англ. presentation layer) обеспечивает преобразование протоколов и кодирова-ние/декодирование данных.
Запросы приложений, полученные с прикладного уровня, на уровне представления преобразуются
в формат для передачи по сети, а полученные из сети данные преобразуются в формат приложений.
На этом уровне может осуществляться сжатие/распаковка или шифрование/дешифрование, а также пе-ренаправление запросов другому сетевому ресурсу, если они не могут быть обработаны локально.
Таким образом, уровень 6 обеспечивает организацию данных при их пересылке.

Сеансовый уровень
Сеансовый уровень (англ. session layer) модели обеспечивает поддержание сеанса связи, позволяя при-ложениям взаимодействовать между собой длительное время. Уровень управляет создани-ем/завершением сеанса, обменом информацией, синхронизацией задач, определением права на переда-чу данных и поддержанием сеанса в периоды неактивности приложений.

Транспортный уровень (англ. transport layer) модели предназначен для обеспечения надёжной передачи данных от отправителя к получателю. При этом уровень надёжности может варьироваться в широких пределах.

87) Уровни среды передачи семиуровневой модели, типы данных, функции и примеры оборудвоания.

21 Лекция

Уровни среды передачи[3]					
	3. Сетевой (network)	Пакеты (packet)	Определение маршрута и логическая адресация	IPv4, IPv6, IPsec, AppleTalk, ICMP	Маршрутизатор
	2. Канальный (data link)	Биты (bit)/ Кадры (frame)	Физическая адресация	PPP, IEEE 802.22, Ethernet, DSL, ARP, сетевая карта.	Коммутатор, точка доступа
	1. Физический (physical)	Биты (bit)	Работа со средой передачи, сигналами и двоичными данными	USB, кабель («витая пара», коаксиальный, оптоволоконный), радиоканал	Концентратор Повторитель (сетевое оборудование)

Сетевой уровень (англ. Network layer) модели предназначен для определения пути передачи данных. Отвечает за трансляцию логических адресов и имён в физические, определение кратчайших маршрутов, коммутацию и маршрутизацию, отслеживание неполадок и «заторов» в сети.
Протоколы сетевого уровня маршрутизируют данные от источника к получателю. Работающие на этом уровне устройства (маршрутизаторы) условно называют устройствами третьего уровня (по номеру уровня в модели OSI).

Канальный уровень (англ. Data link layer) предназначен для обеспечения взаимодействия сетей на физи-ческом уровне и контроля ошибок, которые могут возникнуть. Полученные с физического уровня дан-ные, представленные в битах, он упаковывает в кадры, проверяет их на целостность и, если нужно, ис-правляет ошибки (либо формирует повторный запрос повреждённого кадра) и отправляет на сетевой уровень. Канальный уровень может взаимодействовать с одним или несколькими физическими уровня-ми, контролируя и управляя этим взаимодействием.

Спецификация IEEE 802 разделяет этот уровень на два подуровня:
MAC (англ. Media access control) регулирует доступ к разделяемой физической среде; LLC (англ. Logical link control) обеспечивает обслуживание сетевого уровня.

Физический уровень (англ. Physical layer) — нижний уровень модели, который определяет метод пере-дачи данных, представленных в двоичном виде, от одного устройства (компьютера) к другому.
Составлением таких методов занимаются разные организации, в том числе:
- Институт инженеров по электротехнике и электронике (IEEE);
- Альянс электронной промышленности (EIA — Electronic Industries Alliance (R.I.P.-2010));
- Европейский институт телекоммуникационных стандартов (ETSI — (European Telecommunications Standards Institute);
- CCSDS и другие.
Осуществляют передачу электрических или оптических сигналов в кабель или в радиоэфир и, соответ-ственно, их приём и преобразование в биты данных в соответствии с методами кодирования цифровых сигналов.

88) Классификация IP-сетей — классовая и бесклассовая адресация.

22 Лекция
Для адресации IP-сетей используется 4 байта.
Существуют два типа адресации IP-сетей — классовая и бесклассовая.

Классовая адресация
При классовой адресации сетевой адрес IPv4 разделяется на узловой и сетевой компоненты по байтовой границе следующим образом:
Класс A
На данный тип адреса указывает 0 на месте старшего бита (сетевой порядок байтов) адреса. Адрес сети (сетевой адрес) содержится в самом старшем байте, а адреса узлов занимают оставшиеся три байта.
Класс B
На данный тип адреса указывает двоичное значение 10 на месте двух самых старших битов (сетевой порядок байтов) адреса. Адрес сети содержится в двух старших байтах, а адреса узлов занимают остав-шиеся два байта.
Класс C
На данный тип адреса указывает двоичное значение 110 на месте самых трех старших битов (сетевой порядок байтов) адреса. Адрес сети содержится в первых трёх старших байтах, а адреса узлов занимают оставшийся байт.
Классовая адресация в настоящее время устарела и была заменена на бесклассовую адресацию (CIDR), при которой компоненты сети и узла в адресе могут занимать произвольное число битов (а не байтов).

Бесклассовая адресация — метод IP-адресации, позволяющий гибко управлять пространством IP-адресов, не используя жёсткие рамки классовой адресации. Использование этого метода позволяет эко-номно использовать ограниченный ресурс IP-адресов, поскольку возможно применение различных ма-сок подсетей к различным подсетям. IP-адрес трактуется, как массив бит. Маска подсети задаёт какие биты в IP-адресе являются адресом сети.
Блок адресов задаётся указанием начального адреса и маски подсети.
Бесклассовая адресация основывается на переменной длине маски подсети (англ. variable length subnet mask, VLSM), в то время, как в классовой адресации длина маски подсети имела всего лишь 3 фиксиро-ванных значения.

Пример подсети 192.0.2.16/28 с применением бесклассовой адресации:

Октеты IP-адреса	192								0								2								16							
Биты IP-адреса	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0
Биты маски подсети	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
Октеты маски подсети	255								255								255								240							

В маске подсети 28 бит слева — единицы. В таком случае говорят о длине префикса подсети в 28 бит и указывают через косую черту (символ /) после базового адреса.
192.0.2.16 — адрес подсети
192.0.2.17...30 — адреса хостов (14 штук)
192.0.2.31 — широковещательный адрес
192.0.2.32— адрес следующей подсети

89) Сокеты AF_INET и их типы. Адреса IPv4, их типы и формат адреса

21 Лекция
AF_INET — реализация протокола IPv4 в Linux
tcp_socket = socket(AF_INET, SOCK_STREAM, 0);
udp_socket = socket(AF_INET, SOCK_DGRAM, 0);
raw_socket = socket(AF_INET, SOCK_RAW, protocol);
В Linux реализован протокол Интернета.
Программный интерфейс совместим с интерфейсом сокетов BSD (socket(), bind(), listen(), connect(), accept(), gethostbyname(), gethostbyaddr()).
Сокет IP создаётся с помощью системного вызова socket():
socket(AF_INET, socket_type, protocol);
socket_type — возможные типы сокета:
SOCK_STREAM — для открытия сокета tcp(7);
SOCK_DGRAM — для открытия сокета udp(7);
SOCK_RAW — для открытия сокета raw(7) с прямым доступом к протоколу IP

protocol — задаётся протокол IP, который указывается в IP-заголовке принимаемых или отправляемых пакетов. Допустимые значения для параметра protocol:
0 и IPPROTO_TCP — для сокетов TCP,
0 и IPPROTO_UDP — для сокетов UDP.

Адрес IP-сокета определяется как комбинация IP-адреса интерфейса и номера порта.
В самом протоколе IP нет номеров портов, они реализуются протоколами более высокого уровня, например udp(7) и tcp(7).
У неструктурированных (raw) сокетов номер протокола IP указывается в sin_port.

```
struct sockaddr_in {
sa_family_t sin_family;     // семейство адресов: AF_INET
in_port_t sin_port;         // порт сокета в сетевом порядке байт
struct in_addr sin_addr;    // Интернет-адрес
};
struct in_addr {     // Интернет-адрес
    uint32_t s_addr;  // адрес в сетевом порядке байт
};
```

Значение sin_family всегда устанавливается в AF_INET. Это обязательно.
В sin_port указывается номер порта в сетевом порядке байт.
Порты, номера которых меньше 1024, называются привилегированными портами (зарезервированными портами). Только привилегированные процессы могут быть связаны с этими сокетами с помощью bind(2).

90) Динамическое связывание, преимущества и недостатки по сравнению со статическим.

24 Лекция
С одной стороны программы не могут изменять код — для операционной системы исполняемый код считается «только для чтения».
С другой стороны, программы имеют большие объемы общего кода, поэтому вместо того, чтобы репли-цировать его для каждого исполняемого файла, это общий код может совместно использоваться многи-ми исполняемыми файлами. Виртуальная память позволяет это сделать легко — на физические страни-цы памяти (страничные блоки), в которые загружен общий код (библиотеки), можно легко ссылаться с любого количества виртуальных страниц в любом количестве адресных пространств. Таким образом, несмотря на то, что в памяти системы существует только одна физическая копия кода библиотеки, каж-дый процесс может иметь доступ к этому коду библиотеки по любому виртуальному адресу, который ему нравится.
Преимущества библиотек динамической компоновки:
– только один экземпляр библиотеки динамической компоновки загружается в память, если несколько программ, работающих одновременно, используют одну и ту же библиотеку.
– библиотеку динамической компоновки можно обновлять без изменения исполняемого файла.
– библиотеку динамической компоновки можно вызвать из большинства языков программирования, та-ких как Pascal, C# и Visual Basic. Можно даже вызвать из Java, но сложно.
Недостатки динамически подключаемых библиотек:
– вся библиотека загружается в память, даже если требуется лишь небольшая ее часть.
– загрузка библиотеки динамической компоновки требует дополнительного времени при загрузке ис-полняемого файла программы.
– вызов функции в библиотеке динамической компоновки менее эффективен, чем вызов в статической библиотеке, из-за дополнительных накладных расходов на вызов и из-за менее эффективного использо-вания кэша кода.
– библиотека динамической компоновки должна распространяться вместе с исполняемым файлом.
– несколько программ, установленных на одном компьютере, должны использовать одну и ту же версию библиотеки динамической компоновки, если разные версии имеют одинаковое имя, что может вызвать множество проблем с совместимостью.