1 Операционная система. Определение, выполняемые функции. Примеры.

OS - комплекс системных и управляющих программ, предназначенных для наиболее эффективного использования всех ресурсов вычислительной системы.

Операционная система представляет абстракции для работы с аппаратными ресурсами:
ОЗУ, диск, устройства ввода/вывода (клавиатура, мышь, монитор, принтер).
ОС скрывает особенности работы с конкретной аппаратурой, что позволяет прикладным программам быть переносимыми между различными компьютерами, если на этих компьютерах работает нужная операционная система. Таким образом, операционную систему можно рассматривать как некоторую виртуальную машину.

Управление ресурсами включает в себя мультиплексирование (распределение) ресурсов двумя различными способами: во времени и в пространстве. Когда ресурс разделяется во времени, различные программы или пользователи используют его по очереди: сначала ресурс получают в пользование одни, потом другие и т. д. Определение того, как именно ресурс будет разделяться во времени — кто будет следующим потребителем и как долго, — это задача операционной системы. Другим видом разделения ресурсов является пространственное разделение. Вместо поочередной работы каждый клиент получает какую-то часть разделяемого ресурса.

Основные функции ОС:

- Управление памятью
- Управление дисковым пространством
- Поддержка многозадачности (разделение использования памяти, времени выполнения)
- Ограничение доступа, многопользовательский режим работы
- Обеспечение интерфейса прикладного программирования (API) для доступа к ресурсам

Примеры OS:

1. UNIX — семейство переносимых, многозадачных и многопользовательских операционных систем.
2. Windows — семейство проприетарных операционных систем (ОС) корпорации Microsoft, ориентированных на применение графического интерфейса при управлении.

2 Процессы и потоки. Определение, назначение, взаимосвязь.

Процесс - программа и все необходимые ей данные на этапе исполнения в OS.
Процесс характеризует некоторую совокупность набора исполняющихся команд, ассоциированных с ним ресурсов (выделенная для исполнения память или адресное пространство, стеки, используемые файлы и устройства ввода-вывода и т. д.) и текущего момента его выполнения (значения регистров, программного счетчика, состояние стека и значения переменных), находящегося под управлением операционной системы.
Процесс находится под управлением операционной системы, поэтому в нем может выполняться часть кода ее ядра, как в случаях, специально запланированных авторами программы (например, при использовании системных вызовов), так и в непредусмотренных ситуациях (например, при обработке внешних прерываний).

Потоки исполнения - Thread - наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы. Реализация потоков выполнения и процессов в разных операционных системах отличается друг от друга, но в большинстве случаев поток выполнения исполняется в рамках процесса. Несколько потоков выполнения могут существовать в рамках одного и того же процесса и совместно использовать ресурсы, такие как память, тогда как процессы не разделяют этих ресурсов. В частности, потоки выполнения разделяют инструкции процесса (его код) и его контекст (значения переменных, которые они имеют в любой момент времени).
Для понимания работы потоков необходимо помнить, что каждый поток имеет независимый от других потоков стек выполнения.

В рамках процесса могут находиться один или несколько потоков, каждый из которых обладает следующими характеристиками:

- Состояние выполнения потока (выполняющийся, готовый к выполнению и т.д.)
- Сохраненный контекст не выполняющегося потока; один из способов рассмотрения потока — считать его независимым счетчиком команд, работающим в рамках процесса
- Сохраненный контекст не выполняющегося потока; один из способов рассмотрения потока — считать его независимым счетчиком команд, работающим в рамках процесса
- Стек выполнения
- Статическая память, выделяемая потоку для локальных переменных
- Доступ к памяти и ресурсам процесса, которому этот поток принадлежит, этот доступ разделяется всеми потоками данного процесса

Потоки выполнения отличаются от традиционных процессов многозадачной операционной системы тем, что:

- процессы, как правило, независимы, тогда как потоки выполнения существуют как составные элементы процессов
- процессы несут значительно больше информации о состоянии, тогда как несколько потоков выполнения внутри процесса совместно используют информацию о состоянии, а также память и другие вычислительные ресурсы
- процессы имеют отдельные адресные пространства, тогда как потоки выполнения совместно используют их адресное пространство
- процессы взаимодействуют только через предоставляемые системой механизмы связей между процессами
- переключение контекста между потоками выполнения в одном процессе, как правило, быстрее, чем переключение контекста между процессами

Процессы, системные вызовы:

- pid_t pid = fork(); //создание нового процесса;
  В случае успешного выполнения функции fork(), она возвращает PID процесса-потомка родительскому процессу и нуль — процессу-потомку. Если порождение процесса-потомка закончилось неудачей, функция fork() возвращает значение -1).

- pid_t cpid = getpid();// значение идентификатора текущего процесса
- pid_t ppid = getppid();//начение идентификатора родительского процесса для текущего процесса
- waitpid(pid, &status, 0);//приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс, указанный в параметре pid, не завершит выполнение, или пока не появится сигнал, который либо завершает текущий процесс либо требует вызвать функцию-обработчик
- void exit()(int status);//служит для нормального завершения процесса. Возврата из функции в текущий процесс не происходит, и функция ничего не возвращает. Значение параметра status – кода завершения процесса – передается ядру операционной системы и может быть затем получено процессом, породившим завершившийся процесс.

Системные вызовы для работы с потоками:

- int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void*(*func) (void*), void *arg);
  pthread_create(&(thread), NULL, write_thr, NULL); //Создает новый поток
- pthread_t id = pthread_self(); //Возвращает идентификатор текущего потока
- void pthread_exit(void *status); //Функция pthread_exit служит для завершения нити исполнения (thread'a) текущего процесса.
- int pthread_detach(pthread_t thread); //Отсоединяет поток. По умолчанию все потоки создаются присоединенными. Это означает, что когда поток завершается его идентификатор и статус завершения сохраняются до тех пор, пока какой-либо поток данного процесса не вызовет pthread_join.
- int pthread_join (pthread_t thread, void **status_addr);
  pthread_join(thread, NULL); //Блокирует вызывающий поток до завершения потока

3 Взаимодействие процессов и потоков. Определение. Типы взаимодействий.

Межпроцессное взаимодействие - (англ. inter-process communication, IPC) — обмен данными между потоками одного или разных процессов. Реализуется посредством механизмов, предоставляемых ядром ОС или процессом, использующим механизмы ОС и реализующим новые возможности IPC. Может осуществляться как на одном компьютере, так и между несколькими компьютерами сети.
Условно все IPC можно разделить на обмен данными и синхронизацию обмена. Более детально IPC делятся на следующие:

- механизмы обмена сообщениями;
- механизмы синхронизации;
- механизмы разделения памяти;
- механизмы удалённых вызовов (RPC).

Межпроцессное взаимодействие, наряду с механизмами адресации памяти, является основой для разграничения адресного пространства между процессами.
Сигнальные IPC - Передается минимальное количество информации – один бит, "да"или "нет". Используются, как правило, для извещения процесса о наступлении какого-либо события. 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.

Канальные IPC - Обмен данными между процессами происходит через линии связи, предоставленные операционной системой, и напоминает общение людей по телефону, с помощью записок, писем или объявлений. Объем передаваемой информации в единицу времени ограничен пропускной способностью линий связи. С увеличением количества информации возрастает и возможность влияния на поведение другого процесса.

Разделяемая помять:
Для оценки производительности различных механизмов IPC используют следующие параметры:

- пропускная способность (количество сообщений в единицу времени, которое ядро ОС или процесс способна обработать);
- задержки (время между отправкой сообщения одним потоком и его получением другим потоком).

Типы Межпроцессного взаимодействия:

1. Разделяемая память - используется, когда два или более процесса могут совместно использовать некоторую область адресного пространства.
2. Семафоры - средство синхронизации доступа нескольких процессов к разделяемым ресурсам (функция разрешения/запрещения использования ресурса).
3. Мьютексы - бинарный семафор.
4. Неименованные каналы (pipe) - Обмен данными между процессами порождает программный канал, обеспечивающий однонаправленную передачу между двумя процессами (задачами).
5. Именованные каналы (FIFO) - В отличие от неименованных каналов (pipe) – возможен обмен данными не только между родственными процессами, так как буферизация происходит в рамках файловой системы с именованием -> специальный файл (тип FIFO). Чтение происходит в порядке записи.
6. Сигналы - способ передачи уведомления о событии, произошедшем либо между процессами, либо между процессом и ядром. Сигналы очень ресурсоемки. Ограничены с точки зрения системных средств. Они малоинформативны. Являются простейшим способом IPC. Используются для генерации простейших команд, уведомлений об ошибке. Обработка сигнала похожа на обработку прерывания. Сигнал имеет собственное имя и уникальный номер.
7. Очереди сообщений
8. Сокеты

4 Разделяемая память. Определение, назначение, функции работы с разделяемой памятью.

Два или более процессов могут совместно использовать некоторую область адресного пространства. Созданием разделяемой памяти занимается операционная система (если, конечно, ее об этом попросят). Использование разделяемой памяти для передачи/получения информации осуществляется с помощью средств обычных языков программирования, в то время как сигнальным и канальным средствам коммуникации для этого необходимы специальные системные вызовы. Разделяемая память представляет собой наиболее быстрый способ взаимодействия процессов в одной вычислительной системе.
Разделяемую память *(англ. Shared memory)* применяют для того, чтобы увеличить скорость прохождения данных между процессами. В обычной ситуации обмен информацией между процессами проходит через ядро. Техника разделяемой памяти позволяет осуществить обмен информацией не через ядро, а используя некоторую часть виртуального адресного пространства, куда помещаются и откуда считываются данные. После создания разделяемого сегмента памяти любой из пользовательских процессов может подсоединить его к своему собственному виртуальному пространству и работать с ним, как с обычным сегментом памяти. Недостатком такого обмена информацией является отсутствие каких бы то ни было средств синхронизации, однако для преодоления этого недостатка можно использовать технику семафоров.

Назначение:

- Метод межпроцессного взаимодействия (IPC), то есть способ обмена данными между программами, работающими одновременно. Один процесс создаёт область в оперативной памяти, которая может быть доступна для других процессов.
- Метод экономии памяти, путём прямого обращения к тем исходным данным, которые при обычном подходе являются отдельными копиями исходных данных.

Для работы с разделяемой памятью используются системные вызовы:

- shmget — создание сегмента разделяемой памяти;
  int shmget(key_t key, int size, int shmflg); - возвращает идентификатор разделяемому сегменту памяти, соответствующий значению аргумента *key*

  создание сегмента разделяемой памяти, 128 байт, 0666 - чтение и запись разрешены для всех
  shmid = shmget(key, 128, 0666 | IPC_CREAT | IPC_EXCL);

  Чтобы получить ключ:
  if((key = ftok(pathname, 1)) < 0)

  printf("Ключ не сгенерирован");
  exit(-1);

  printf("Ключ сгенерирован");

  *key_t ftok(const char *path, int id);*
  использует файл с именем pathname (которое должно указывать на существующий файл к которому есть доступ) и младшие 8 бит id (который должен быть отличен от нуля) для создания ключа с типом key_t. Возвращаемое значение одинаково для всех имен, указывающих на один и тот же файл при одинаковом значении id. Возвращаемое значение должно отличаться, когда (одновременно существующие) файлы или идентификаторы проекта различаются.
- *shmctl — установка параметров; int shmctl(int shmid, int cmd, struct shmid_ds *buf);*
  shmctl() позволяет пользователю получать информацию о разделяемых сегментах памяти, устанавливать владельца, группу разделяемого сегмента, права на него; эта функция может также удалить сегмент.

  shmctl(shmid, IPC_RMID, NULL);
- shmat — подсоединение сегмента памяти;
  void *shmat(int shmid, const void *shmaddr, int shmflg);

Функция shmat подстыковывает сегмент разделяемой памяти *shmid* к адресному пространству вызывающего процесса.Адрес подстыковываемого сегмента определяется *shmaddr* с помощью одного из перечисленных ниже критериев:

- Если *shmaddr* равен **NULL,** то система выбирает для подстыкованного сегмента подходящий (неиспользованный) адрес.
- Если *shmaddr* не равен **NULL,** а в поле *shmflg* включен флаг SHM_RND, то подстыковка производится по адресу *shmaddr*, округленному вниз до ближайшего кратного SHMLBA. В противном случае *shmaddr* должен быть округленным до размера страницы адресом, к которому производится подстыковка.
- Если в поле *shmflg включен флаг SHM_RDONLY, то подстыковываемый сегмент будет доступен только для чтения, и вызывающий процесс должен иметь права на чтение этого сегмента. Иначе, сегмент будет доступен для чтения и записи, и у процесса должны быть соответствующие права. Сегментов "только-запись"не существует.*
- *Флаг SHM_REMAP (специфичный для Linux) может быть указан в shmflg* для обозначения того, что распределение сегмента должно замещать любые существующие распределения в диапазоне, начиная с *shmaddr* и до размера сегмена. (Обычно выдается ошибка EINVAL если уже существует распределение в этом диапазоне адресов.) В этом случае *shmaddr* не должно быть равно **NULL.**
- Значение *brk* вызывающего процесса подстыковкой не изменяется. При завершении работы процесса сегмент будет отстыкован. Один и тот же сегмент может быть подстыкован в адресное пространство процесса несколько раз, как "только для чтения так и в режиме "чтение-запись".

shmat(shmid, NULL, 0));

- shmdt — отсоединение сегмента;
int shmdt(const void *shmaddr);
Функция shmdt отстыковывает сегмент разделяемой памяти, находящийся по адресу *shmaddr,* от адресного пространства вызвающего процесса. Отстыковываемый сегмент должен быть среди пристыкованных ранее функцией shmat. Параметр *shmaddr* должен быть равен значению, которое возвратила соответствующая функция shmat.

  shmdt(shm);

5 Семафоры и мьютексы. Определение, назначение. Примеры совместного использования.

Семафор - объект, ограничивающий количество потоков, которые могут войти в заданный участок кода.(предложен Дейкстрой)
Назначение:

- запрет одновременного выполнения заданных участков кода (критические секции - участок исполняемого кода программы, в котором производится доступ к общему ресурсу (данным или устройству), который не должен быть одновременно использован более чем одним потоком исполнения);
- поочерёдный доступ к критическому ресурсу (важному ресурсу, для которого невозможен (или нежелателен) одновременный доступ);
- синхронизация процессов и потоков (например, можно инициировать обработку события отпусканием семафора).

Семафор в ОС UNIX состоит из следующих элементов:

- значение семафора;
- идентификатор процесса, который хронологически последним работал с семафором;
- число процессов, ожидающих увеличения значения семафора;
- число процессов, ожидающих нулевого значения семафора.

Для работы с семафорами поддерживаются три системных вызова:

- semget для создания и получения доступа к набору семафоров;
- semop для манипулирования значениями семафоров (это именно тот системный вызов, который позволяет процессам синхронизоваться на основе использования семафоров);
- semctl для выполнения разнообразных управляющих операций над набором семафоров.

Мьютексы можно рассматривать как двоичные семафоры.
Блокировка, которая устанавливается (запирается) перед обращением к разделяемому ресурсу и снимается (отпирается) после выполнения требуемой последовательности операций. Если мьютекс заперт, то любой другой поток, который попытается запереть его, будет заблокирован до тех пор, пока мьютекс не будет отперт.
Если в момент, когда отпирается мьютекс, заблокированными окажутся несколько потоков, все они будут запущены и первый из них, который успеет запереть мьютекс, продолжит работу. Все остальные потоки обнаружат, что мьютекс по-прежнему заперт, и опять перейдут в режим ожидания. Таким образом, доступ к ресурсу сможет получить одновременно только один поток.
Такой механизм взаимоисключений будет корректно работать только при условии, что все потоки приложения будут соблюдать одни и те же правила доступа к данным. Операционная система никак не упорядочивает доступ к данным. Если мы позволим одному потоку производить действия с разделяемыми данными, предварительно не ограничив доступ к ним, то остальные потоки могут обнаружить эти данные в противоречивом состоянии, даже если перед обращением к ним будут устанавливать блокировку.
Переменные-мьютексы определяются с типом pthread_mutex_t. Прежде чем использовать переменную мьютекс, мы должны сначала инициализировать ее, записав в нее значение константы
PTHREAD_MUTEX_INITIALIZER (только для статически размещаемых мьютексов) или вызвав функцию pthread_mutex_init.
Назначение - защита объекта от доступа к нему других потоков, отличных от того, который завладел мьютексом.
Системные вызовы работы с mutex:

- pthread_mutex_init - Инициализирует мьютекс переданный параметром mutex. Для того, чтобы использовать параметры по умолчанию необходимо вместо attr передать NULL.
  int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);
  static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
- pthread_mutex_destroy - Уничтожает мьютекс.

- pthread_mutex_lock - Блокирует мьютекс. Если мьюеткс уже заблокирован, то вызывающий поток блокируется. pthread_mutex_lock(&mutex);
- pthread_mutex_trylock - Если к моменту вызова этой функции мьютекс будет отперт, функция запрет мьютекс и вернет значение 0. В противном случае pthread_mutex_trylock вернет код ошибки EBUSY. Вызывающий поток при в этом случае не плокируется.
- pthread_mutex_unlock - Разблокирует мьютекс. pthread_mutex_unlock(&mutex);

6 Условные блокировки. Определение, назначения. Примеры использования.

Условная переменная — примитив синхронизации, обеспечивающий блокирование одного или нескольких потоков до момента поступления сигнала от другого потока о выполнении некоторого условия или до истечения максимального промежутка времени ожидания.
Условные переменные используются для того, чтобы заблокировать потоки до выполнения определенных условий. Условные переменные применяются в сочетании мьютексам, чтобы несколько потоков могли ожидать момента выполнения одного условия. Это можно сделать несколькими способами. Сначала поток блокирует мьютекс, но и сам блокируется до момента выполнения условия. На то время, пока поток заблокирован, установленная им блокировка мьютекса автоматически снимается. Когда другой поток выполняет поставленное условие, он дает условной переменной сигнал (не имеющий отношения к сигналам unix) о разблокировании первого потока. После блокировки потока мьютекс автоматически устанавливается и первый поток повторно проверяет условие. Если оно не выполняется, поток опять блокируется переменной. Если условие выполняется, поток разблокирует мьюеткс и выполняется дальше.

Системные вызовы для работы с условными переменными:

- pthread_cond_init;
  int pthread_cond_init (pthread_cond_t *cond, const pthread_condattr_t *attr);
  pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
  Инициализирует атрибутный объект условной переменной, заданный параметром attr, значениями, действующими по умолчанию для всех атрибутов, определенных реализацией.

- pthread_cond_wait;
  int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);
  При вызове pthread_cond_wait мутекс должен быть захвачен, в противном случае результат не определен. Pthread_cont_wait освобождает мутекс и блокирует нить до момента вызова другой нитью pthread_cond_signal. После пробуждения wait пытается захватить мутекс; если это не получается, он блокируется до того момента, пока мутекс не освободят.

- pthread_cond_timedwait;
  int pthread_cond_timedwait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex, const struct timespec *restrict abstime);
  Аналогично pthread_cond_wait, но задается время ожидания разблокировки.

- pthread_cond_signal;
  int pthread_cond_signal(pthread_cond_t *cond);
  Разблокирует поток, заблокированный вызовом pthread_cond_wait().

- pthread_cond_broadcast;
  int pthread_cond_broadcast(pthread_cond_t *cond); Разблокирует все потоки, заблокированные вызовом pthread_cond_wait().

- pthread_cond_destroy;
  int pthread_cond_destroy(pthread_cond_t *cond);
  Уничтожает условную переменную.

7 Блокировки чтения-записи. Определение назначение, примеры использования.

Блокировки чтения-записи (read-write locks) похожи на мьютексы, но отличаются от них тем, что имеют два режима захвата - для чтения и для записи. Блокировку для чтения могут удерживать несколько нитей одновременно. Блокировку для записи может удерживать только одна нить; при этом никакая другая нить не может удерживать эту же блокировку для чтения.

Блокировки чтения-записи похожи на мьютексы, за исключением того, что - они допускают более высокую степень параллелизма. Мьютексы могут иметь всего два состояния, закрытое и открытое, и только один поток может владеть мьютексом в каждый момент времени. Блокировки чтения-записи могут иметь три состояния: *режим блокировки для чтения, режим блокировки для записи и отсутствие блокировки.* Режим блокировки для записи может установить только один поток, но установка режима блокировки для чтения доступна нескольким потокам одновременно.

Если блокировка чтения-записи установлена в режиме блокировки для записи, все потоки, которые будут пытаться захватить эту блокировку, будут приостановлены до тех пор, пока блокировка не будет снята. Если блокировка чтения-записи установлена в режиме блокировки для чтения, все потоки, которые будут пытаться захватить эту блокировку для чтения, получат доступ к ресурсу, но если какой-либо поток попытается установить режим блокировки для записи, он будет приостановлен до тех пор, пока не будет снята последняя блокировка для чтения. Различные реализации блокировок чтения-записи могут значительно различаться, но обычно, если блокировка для чтения уже установлена и имеется поток, который пытается установить блокировку для записи, то остальные потоки, которые пытаются получить блокировку для чтения, будут приостановлены. Это предотвращает возможность блокирования пишущих потоков непрекращающимися запросами на получение блокировки для чтения.

Блокировки чтения-записи прекрасно подходят для ситуаций, когда чтение данных производится намного чаще, чем запись. Когда блокировка чтения-записи установлена в режиме для записи, можно безопасно выполнять модификацию защищаемых ею данных, поскольку только один поток может владеть блокировкой для записи. Когда блокировка чтения-записи установлена в режиме для чтения, защищаемые ею данные могут быть безопасно прочитаны несколькими потоками, если эти потоки смогли получить блокировку для чтения.

Блокировки чтения-записи еще называют совместноисключающими блокировками. Когда блокировка чтения-записи установлена в режиме для чтения, то говорят, что блокировка находится в режиме совместного использования.

Когда блокировка чтениязаписи установлена в режиме для записи, то говорят, что блокировка находится в режиме исключительного использования. Функции:

- pthread_rwlock_init;
  int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr);
  pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
  Функция pthread_rwlock_init инициализирует блокировку чтения-записи. Если в аргументе attr передается пустой указатель, блокировка инициализируется с атрибутами по умолчанию.

- pthread_rwlock_destroy;
  int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
  Перед освобождением памяти, занимаемой блокировкой чтения-записи, нужно вызвать функцию pthread_rwlock_destroy, чтобы освободить все занимаемые блокировкой ресурсы.

- pthread_rwlock_rdlock;
  int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
  Заблокироваь ресурс для чтения.

- pthread_rwlock_wrlock;
  int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
  Заблокироваь ресурс для записи.

- pthread_unlock_wrlock;

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
Снять блокировку чтения-записи.

- pthread_tryrdlock_wrlock;
  int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
  Попытаться заблокировать ресурс для чтения.

- pthread_trywrlock_wrlock;
  int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
  Попытаться заблокировать ресурс для записи.

8 Именованные и неименованные каналы. Определение, назначение, примеры использования.

Для организации потокового взаимодействия любых процессов в операционной системе UNIX применяется средство связи, получившее название FIFO (от First Input First Output) или именованный pipe. FIFO во всем подобен pipe, за одним исключением: доступ к FIFO процессы могут получать не через родственные связи, а через файловую систему. Для этого при создании FIFO на диске создается файл специального типа, обращаясь к которому процессы могут обмениваться информацией. Для создания FIFO используется системный вызов *mknod()* или существующая в некоторых версиях UNIX функция *mkfifo()*.

Важно! При работе этих системных вызовов не происходит действительного выделения области адресного пространства операционной системы под именованный pipe, а только заводится файл-метка, существование которой позволяет осуществить реальную организацию FIFO в памяти при его открытии с помощью системного вызова *open()*.

После открытия FIFO ведет себя точно так же, как и pipe. Для дальнейшей работы с ним применяются системные вызовы read(), write() и close(). Время существования FIFO в адресном пространстве ядра операционной системы, как и в случае с pipe, не может превышать время жизни последнего из использовавших его процессов. Когда все процессы, работающие с FIFO, закрывают все файловые дескрипторы, ассоциированные с ним, система освобождает ресурсы, выделенные под FIFO. Вся непрочитанная информация теряется. В то же время файл-метка остается на диске и может использоваться для новой реальной организации FIFO в дальнейшем.

Важно! Системный вызов lseek() к FIFO не применим. Для обмена между родственными процессами используют pipe. Pipe представляет собой два файловых дескриптора: один для чтения, другой для записи.

Программные каналы не имеют имен, и их главным недостатком является невозможность передачи информации между неродственными процессами. Два неродственных процесса не могут создать канал для связи между собой (если не передавать дескриптор).

Неименованный канал — один из методов межпроцессного взаимодействия (IPC) в операционной системе, который доступен связанным процессам — родительскому и дочернему. Представляется в виде области памяти на внешнем запоминающем устройстве, управляемой операционной системой, которая осуществляет выделение взаимодействующим процессам частей из этой области памяти для совместной работы.

Системные вызовы для работы с FIFO:

1. int mknod(char *path, int mode, int dev);
   Параметр *dev* является несущественным в нашей ситуации, и мы будем всегда задавать его равным 0.
   Параметр *path* является указателем на строку, содержащую полное или относительное имя файла, который будет являться меткой FIFO на диске. Для успешного создания FIFO файла с таким именем перед вызовом существовать не должно.
   Параметр *mode* устанавливает атрибуты прав доступа различных категорий пользователей к FIFO, аналогичные атрибутам задающим права доступа для файлов.

2. int mkfifo(char *path, int mode);
   Функция mkfifo() предназначена для создания FIFO в операционной системе.
   Параметр *path* является указателем на строку, содержащую полное или относительное имя файла, который будет являться меткой FIFO на диске. Для успешного создания FIFO файла с таким именем перед вызовом функции не должно существовать.
   Параметр *mode* соответствует параметру mode системного вызова mknode().

   Возвращаемые значения:
   При успешном создании FIFO функция возвращаетзначение 0, при ошибке – отрицательное значение.

3. Системные вызовы read() и write() при работе с FIFO имеют те же особенности поведения, что и при работе с pipe.
   Системный вызов open() при открытии FIFO также ведет себя несколько иначе, чем при открытии других типов файлов, что связано с возможностью блокирования выполняющих его процессов.
   Если FIFO открывается только для чтения, и флаг O_NDELAY не задан, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет FIFO на запись.
   Если флаг O_NDELAY задан, то возвращается значение файлового дескриптора, ассоциированного с FIFO.
   Если FIFO открывается только для записи, и флаг O_NDELAY не задан, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет FIFO на чтение.
   Если флаг O_NDELAY задан, то констатируется возникновение ошибки и возвращается значение -1.
   Задание флага O_NDELAY в параметрах системного вызова open() приводит и к тому, что процессу, открывшему FIFO, запрещается блокировка при выполнении последующих операций чтения из этого потока данных и записи в него.

   Если процесс записывает данные в FIFO, с которым не взаимодействует в режиме чтения ни один другой процесс, то ядро посылает в него сигнал SIGPIPE, для уведомления. Если процесс пытается прочитать данные из FIFO, с которым ни один процесс не взаимодействует в режиме записи, то он прочитает оставшиеся (если они там были) данные и признак конца файла.
   Таким образом, если два процесса взаимодействуют через FIFO записывающий процесс после завершения работы должен закрыть свой дескриптор FIFO для того, чтобы читающий процесс получил признак конца файла.

Для создания неименованного канала используется системный вызов pipe(int *fd).
Аргументом данного системного вызова является массив fd из двух целочисленных элементов. Если системный вызов pipe() прорабатывает успешно, то он возвращает код ответа, равный нулю, а массив будет содержать два открытых файловых дескриптора. Соответственно, в fd[0] будет содержаться дескриптор чтения из канала, а в fd[1] — дескриптор записи в канал. После этого с данными файловыми дескрипторами можно использовать всевозможные средства работы с файлами, поддерживающие стратегию FIFO, т.е. любые операции работы с файлами, за исключением тех, которые касаются перемещения файлового указателя.
Неименованные каналы в общем случае предназначены для организации взаимодействия родственных процессов, осуществляющегося за счет передачи по наследству ассоциированных с каналом файловых дескрипторов. Но иногда встречаются вырожденные случаи использования неименованного канала в рамках одного процесса.