## Язык С как стандарт взаимодействия с ОС

**Замечание**

Хотя 32-битное ПО может работать на процессорах х86-64, *64-битное ПО не может работать на процессорах х86*.

**Общеизвестно**, что язык ***С*** рассматривается в двух контекстах:

* *как высокоуровневый инструмент* для написания самих ОС;
* *как низкоуровневый инструмент* для написания прикладных программ, взаимодействующих с ядром ОС.

Синтаксис языка не содержит даже собственных конструкций ввода-вывода, функцио- нальное окружение его, реализованное прежде всего в виде библиотеки *libc*, явля- ется неотъемлемой частью ОС и *изменяется после перекомпиляции ее ядра*.

**Как следствие** указанных особенностей языка ***С***, его реализация для целей программирования сильно привязана как к аппаратной платформе ЭВМ, так и к архитектуре ПО ОС. Это, в свою очередь, порождает проблемы переносимости уже разработанного ПО на разные архитектуры, а также разработку инструментальных средств для этого языка.

**Стремление повысить мобильность ПО**, которое пришется на языке С, привело к разработе стандарта **POSIX** (*Portable Operating System Interface*), призванного стабилизировать описание интерфейсов различных операционных систем.

Само название **POSIX** (1988 год) было предложено известным специалистом, являющимся основателем «*Фонда свободного программного обеспечения*», - Ричар- дом Столмэном. Наиболее современная версия стандарта POSIX, ***в редакции 2003 г.***, основана на Техническом стандарте *Open Group IEEE Std 1003.1* и на междуна- родном стандарте *ISO/IEC 9945* (версии до 2001 года являются устаревшими).

**По состоянию на 2001 год**, стандарт содержал следующие *четыре части*:

1. *Основные определения* (термины, концепции и интерфейсы, общие для всех частей);
2. *Описание прикладного программного C-интерфейса* к системным сервисам;
3. *описание интерфейса* к системным сервисам на уровне командного языка и служебных программ;
4. *Детальное разъяснение* положений стандарта, обоснование принятых реше- ний;
5. *В дальнейшем*, многие мелкие исправления накапливались и были внесены *в редакцию 2003 года*.

Основные идеи этого стандарта описываются *множеством базовых, системных сервисов*, необходимых для функционирования прикладных программ. **Доступ к этим сервисам** предоставляется посредством интерфейса, который был специфи- цирован для языка ***C***, командного языка и других общеупотребительных служебных программ. Здесь следует подчеркнуть, что у каждого интерфейса есть две стороны: *вызывающая* и *вызываемая*.

**Стандарт POSIX** ориентирован в первую очередь на *вызывающую сторону*, что делает его полезным как для системных, так и прикладных программистов.

**Цель стандарта POSIX** - сделать приложения мобильными на уровне исход- ного кода языка. Это значит, что, при переносе программ языка ***С*** на другую опера-ционную платформу, *потребуется только новая компиляция исходных текстов программ*.

**Замечание** Поскольку, стандарт POSIX определяет только интерфейс к системным сервисам, то *он оставляет за рамками рассмотрения саму реализацию интерфейса*. В частности:

* *не различаются* системные вызовы и библиотечные функции;
* *не являются* объектом стандартизации средства администрирования, а также аппаратные ограничения и функции, которые необходимы только суперполь-

зователю.

**Ориентация POSIX** на международный стандарт языка ***C*** определила также и направление развития спецификаций POSIX, в плане синхронизации обоих стан- дартов:

* В стандарте проведено разделение на *обязательные* и *дополнительные функ- ции*.
* Особое внимание уделяется *способам реализации стандартизуемых функ- ций*, как в "классической" Unix-среде, так и на других операционных плат- формах, в сетевых и распределенных конфигурациях.

**В редакции 2003-го года**, стандарт POSIX рассматривает следующие катего- рии системных компонентов:

* средства разработки;
* сетевые средства;
* средства реального времени;
* потоки управления;
* математические интерфейсы;
* пакетные сервисы;
* заголовочные файлы;
* унаследованные интерфейсы.

Именно такой, общий перечень интерфейсов должна предоставлять каждая

операционная система для работы любого приложения.

*Реализация* или *операционная система*, соответствующая стандарту POSIX, должна поддерживать *все обязательные служебные программы, функции, а также заголо-вочные файлы с обеспечением специфицированного в стандарте поведения*.

Для этих целей используется константа ***\_POSIX\_VERSION***, которая имеет значение ***200112L***.

*ОС также может* предоставлять *возможности, помеченные стандартом в качестве дополнительных* или *содержать нестандартные функции*. Если утверж- дается, что поддерживается некоторое расширение, то это должно производиться *непротиворечивым образом*, для всех необходимых частей и так, *как описано в стандарте*. Для этого, в заголовочном файле **<unistd.h>** должны быть *определены константы, соответствующие всем поддерживаемым необязательным возмож- ностям*. Например, константа ***\_POSIX2\_C\_DEV*** обслуживает средства разработки программ на языке C.

**Анализируя** эти константы, *во время компиляции*, система разработки выяс- нит возможности используемой ОС и подстроится под них.

**Аналогичные действия** могут быть выполнены с помощью:

* функции *long sysconf(int name)*, - во время выполнения программы;
* служебной программой *getconf*, - посредством запуска ее в командной строке или в сценарии языка shell.

**Для минимизации** размеров ОС и приложений, стандартом POSIX была пре- дусмотрена весьма мелкая гранулярность *необязательных возможностей*. Было проведено объединение *взаимосвязанных необязательных возможностей* в группы:

* шифрование;
* средства реального времени;
* продвинутые средства реального времени;
* потоки реального времени;
* продвинутые потоки реального времени;
* трассировка;
* ПОТОКИ;
* унаследованные возможности.

**В документации** на ОС должны быть отражены вопросы соответствия стан-

дарту POSIX, описаны поддерживаемые дополнительные и нестандартные возмож- ности.

**Применительно непосредственно к ОС**, определены ряд основных поня- тий, соответствующих стандарту POSIX:

* пользователь;
* файл;
* процесс;
* терминал;
* хост;
* узел сети;
* время;
* языково-культурная среда.

**Это** - первичные понятия, которые строго не определяются, а поясняются с

помощью других понятий и отношений. Для каждого из них описаны присущие им атрибуты и применимые к ним операции. Содержатся пояснения следующих *ос- новных понятий*:

1. *У пользователя* есть имя и числовой идентификатор.
2. *Файл* - объект, допускающий чтение и/или запись и имеющий такие атрибуты, как права доступа и тип. К числу последних относятся обычный файл, символьный и блочный специальные файлы, канал, символьная ссылка, сокет и каталог. Реализация может поддерживать и другие типы файлов.
3. *Процесс* - адресное пространство вместе с выполняемыми в нем потоками управления, а также системными ресурсами, которые этим потокам требуются.
4. *Терминал* (или терминальное устройство) - символьный специальный файл, подчиняющийся спецификациям общего терминального интерфейса.
5. *Сеть* - совокупность взаимосвязанных хостов.
6. *Языково-культурная среда* - часть пользовательского окружения, которая зависит от языковых и культурных соглашений.

**Для работы**, с большим числом сущностей, предоставляются *механизмы группирования* и *построения иерархий*. Существует:

* иерархия файлов;
* группы пользователей и процессов;
* подсети и другие.

**Для написания программ**, оперирующих с сущностями POSIX-совмести-

мых систем, применяются или командный интерпретатор (языка shell) и/или компи- лируемый язык C:

* *в первом случае*, приложение может пользоваться разными служебными прог- раммами (утилитами);
* *во втором*, - функциями.

**Функциональный интерфейс ОС** всегда считается *первичным*, но, в POSIX- совместимых ОС, определены объекты, которые считаются *вспомогательными*. Такие объекты обеспечивают организацию взаимодействия между основными сущ- ностями. Примерами таких объектов являются *средства межпроцессного взаимо- действия*, которые *выполняются в определенном окружении*. Частью такого окру- жения является *языково-культурная среда (Locale)*, которая образованная такими категориями, как:

* *символы и их свойства;*
* *форматы сообщений;*
* *дата и время;*
* *числовые и денежные величины*.

*Каждый пользовательский процесс ОС ассоциирован*, по крайней мере, на три файла:

* стандартный ввод;
* стандартный вывод;
* стандартный протокол.

**Обычно**, стандартный ввод назначается на клавиатуру терминала, а стан-

дартный вывод и стандартный протокол - на экран:

* со стандартного ввода читаются команды и (иногда) исходные данные для них;
* на стандартный вывод поступают результаты выполнения команд;
* в стандартный протокол помещаются диагностические сообщения.

**Замечание** К ОС, также могут предъявляться качественные требования, например, требование под- держки реального времени: способность обеспечить необходимый сервис в течение за- данного отрезка времени.

**Стандарт POSIX** также определяет ряд требований к среде компиляции POSIX-совместимых приложений. Часто, разработка приложений ведется в *кросс- режиме*. Поэтому на каждой инструментальной платформе создается такая среда компи-ляции приложений, чтобы результат этой компиляции можно было перенес- ти для последующего выполнения на целевую платформу.

**Важнейшая часть среды компиляции** - заголовочные (или включаемые) файлы, содержащие прототипы функций, определения символических констант, макросов, типов данных, структур и т.п.

**Для каждой**, описанной в стандарте POSIX функции, определено, где и какие заголовочные файлы должны быть включены использующим ее приложением. Пос- редством символических констант, определенных в заголовочном файле ***<****unistd.h****>***, операционная система предоставляет приложению информацию о поддерживаемых возможностях.

**Стандартом POSIX**, также предусмотрен симметричный механизм, называе- мый механизмом макросов проверки возможностей. Он позволяет приложениям объявлять о своем желании получить доступ к определенным прототипам и име- нам.

**С целью** не допустить пересечения имен, в заголовочных файлах исполь- зуются префиксы ***posix\_***, ***POSIX\_*** и ***\_POSIX\_***, которые зарезервированы для нужд стандарта.

**Замечание**

С подчеркивания, за которым следует еще одно подчеркивание или заглавная латинская буква, могут начинаться только системные, но не прикладные имена.

**Для включаемых файлов** описаны префиксы используемых в них имен.

Например:

* *для операций управления файлами*, фигурирующих в *<fcntl.h>*, в качестве префиксов задействованы ***F\_, O\_, S\_***;
* *у средств межпроцессного взаимодействия*, описанных в файле *<sys/ipc.h>*, префиксом служит ***IPC\_***;
* *для манипулирования характеристиками терминалов* в файле *<termios.h>* определено множество разнообразных имен: EXTB, VDSUSP, DEFECHO, FLUSHO и другие;
* еще имеется *четыреста семнадцать* имен типа *\_Exit, abort, abs, acos* и дру- гие, которые могут участвовать в редактировании внешних связей приклад- ной программы.

*Мобильность приложений*, соответствующих стандарту POSIX, принципи- ально достижима благодаря двум основным факторам:

* *во-первых*, - наличию огромного числа стандартизованных системных серви- сов;
* *во-вторых*, - возможности динамического выяснения характеристик целевой платформы и подстройки под них приложения.

Приложения, соответствующие стандарту POSIX, могут быть *одно****-*** и *много****-*** *процессными*, с возможностью динамической адаптации конфигурации к свойствам целевой платформы. Далее перечислим основные важные элементы стандартиза- ции без подробного объяснения.

**Стандартизованы средства** *порождения и завершения процессов, смены их программ, опроса и/или изменения разнообразных характеристик*. Процессы мож- но приостанавливать и активизировать в заданное время.

Необходимая *степень детерминизма выполнения приложений* достигается благодаря *средствам поддержки реального времени*, к которым относятся управ- ление дисциплиной выделения процессоров, сигналы реального времени, удержа- ние страниц в оперативной памяти, таймеры высокого разрешения и другие.

**Функции для работы с файлами** удовлетворяют потребности приложений в чтении и записи долговременных данных, защите таких данных от несанкциониро- ванного доступа:

* *Механизм блокировки фрагментов файлов* позволяет обеспечить *атомар- ность транзакций*.
* *Асинхронный ввод/вывод* дает возможность совмещать операции обмена, оп- тимизируя тем самым приложения.
* *С помощью множества служебных программ* можно относительно легко организовать сложную обработку данных.
* *Тщательно проработаны вопросы доступа* к внешним устройствам, которые подсоединены по последовательным линиям, особенно к терминалам.

**Стандартизованный командный язык** shell обеспечивает адекватное сред- ство для написания небольших мобильных процедур и их быстрой интерактивной отладки.

**Выделены механизмы конвейеров**, позволяющие объединять команды в цепочки с фильтрацией промежуточных результатов.

**Служебные программы** образуют развитую среду выполнения для shell- процедур.

**Фоновый режим** выполнения программ позволяет организовать одновре- менное выполнение нескольких программ и взаимодействие с ними.

**Стандарт POSIX** стандартизует интерфейс командной строки. Вероятно, в будущих версиях стандарта будет регламентирован графический интерфейс.

**Для многопользовательских систем**, POSIX регламентирует различные средства *непосредственного и почтового обмена информацией*.

**Стандарт POSIX** - обязательный элемент современной дисциплины разра- ботки прикладных систем.

## Системные операции для работы с файловой системой

Теперь перейдем к изучению основных *системных вызовов*, разделенных на две группы

* управление файлами;
* управление каталогами и файловыми системами.

Изучение назначения и возможностей этих функций является предметной целью

данной темы.

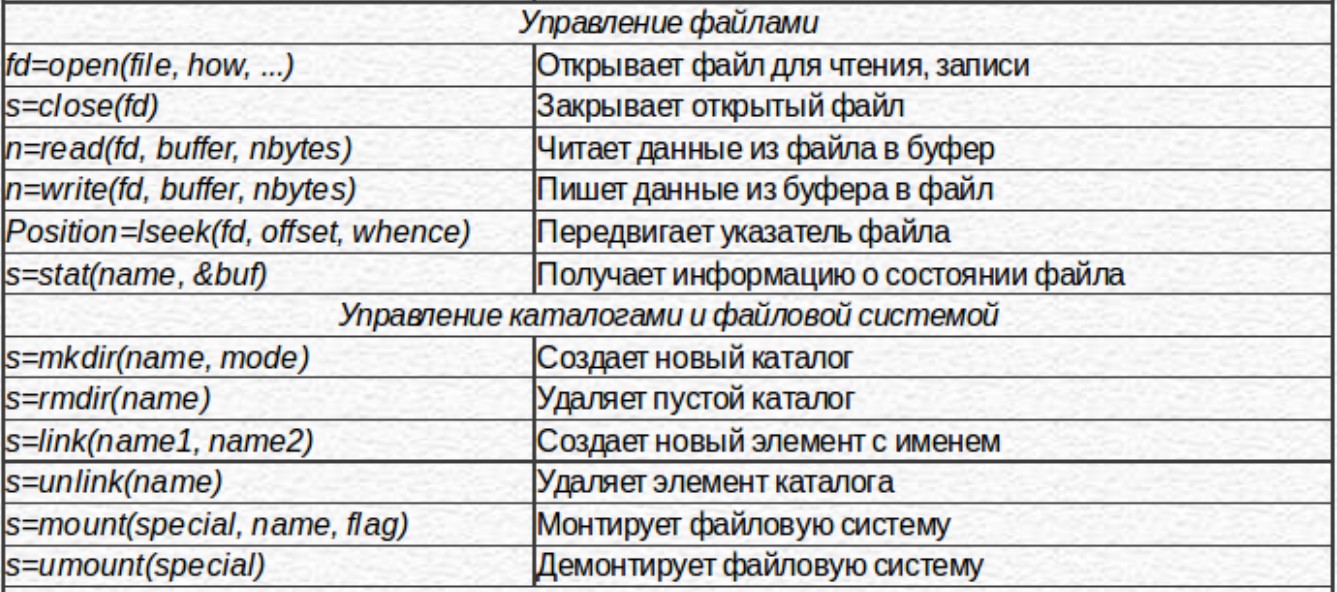


Рисунок 1.5 — Системные функции подсистемы ввода-вывода

**Замечание**

Отметим, что наряду с *низкоуровневыми системными вызовами*, приведенными на ри- сунке 1.5, для работы с файлами и файловыми системами используются *более высо- коуровневые функции*, которые обеспечивают буферизованный ввод-вывод и подробно изложены в любом учебнике по языку С. Считается, что студент уже имеет опыт работы с ними и, при необходимости, использует учебник [5] как справочное пособие.

В данном подразделе будут изучены только первые шесть функций, которые касаются *непосредственно управления файлами*.

Все системные функции ввода-вывода используют *целочисленное значение дескриптора файла*. Это значение формируется в ядре ОС и индивидуально привя- зывается к конкретному процессу.

Нумерация значений дескрипторов для каждого процесса начинается с нуля и определяется в момент открытия файлов, из условия *минимального значения дос- тупного (незанятого) дескриптора*.

Когда файл закрывается, то соответсвующий ему *номер дескриптора осво- бождается* и *используется для открытия других файлов*.

Все остальные системные функции, кроме функции *stat(...)*, используют уже открытый дескриптор файла.

### Системные вызовы open() и close()

Системный вызов *open(...)* открывает некоторый файл и, в случае успешного завершения, возвращает целочисленный номер дескриптора.

Допускается использование следующих аргументов:

*pathname* — строковый указатель на имя файла, заданного в соответсвии с требова- ниями конкретной ОС;

*flags* — целочисленное значение, определяющее режимы открытия файла, которые обычно задаются *конкатенацией битовых масок*:

* *O\_APPEND* — добавление информации в конец файла;
* *O\_CREAT* — создание файла, если он отсутсвует;
* *O\_RDONLY* — только для чтения из файла;
* *O\_WRONLY* — только для записи в файл;
* *O\_RDWR* — для чтения и записи в файл;
* *O\_TMPFILE* — после закрытия файл будет удален, а также *многие другие*. *mode* — целочисленное значение, определяющее режимы доступа к файлу,

например, восьмеричное значение *0666* разрешает доступ на чтение и запись всем пользователям.

В случае ошибочного завершения, функция *open(...)* возвращает отрицатель- ное значение -1, а переменная *errno* содержит код ошибки, котрый следует анали- зировать отдельно.

**Замечание**

Функция *open(...)* содержит множество режимов открытия файлов, которые могут быть несовместимы друг с другом, а также множество вариантов сообщений об ошибках,

Системный вызов *close(...)* закрывает открытый файл и освобождает номер дескриптора для последующего использования.

ние 0.

В случае успешного завершения вызова, возвращается целочисленное значе-

В случае ошибки — целочисленное значение -1, а переменная *errno* содержит

код ошибки:

* *EBADF* — аргумент функции не является правильным дескриптором файла;
* *EINTR* — закрытие файла было прервано сигналом;
* *EIO* — общая ошибка ввода-вывода.

**Замечание**

Несмотря на кажущуюся простоту, системный вызов *close(...)* выполняет очень важные

функции:

* *он универсален*, поскольку применим ко всем видам файлов;
* *корректно завершает работу с файлом*, записывая при необходимости все данные на устройство ввода-вывода;
* *неявно вызывается* ядром ОС, когда процесс завершает свою работу.

### Системные вызовы read() и write()

Системный вызов *read(...)* читает заданное число байт из файла, определен- ного его дескриптором. В случае успешного завершения операции чтения, данные помещаются во внешний буфер и возвращается реальное число прочитанных байт.

Для вызова этой функции используется три аргумента:

* *fd* — целочисленный номер дескриптора открытого файла;
* *buf* — указатель на внешний буфер, в который записываюся читаемые байты;
* *count* — количество байт запрашиваемых для чтения из файла. Функция *read(...)* читает байты с текущей позиции файла:
* если смещение текущей позиции равно или больше размера файла, то возвра- щается значение 0;
* если данные недоступны, но конец файла не достигнут, то ожидается поступ- ление данных, что в общем случае определяется режимом открытия файла;
* в случае ошибки чтения, возвращается значение -1 и необходимо анализиро- вать переменную *errno*;
* если размер параметра *count* превышает значение константы *SSIZE\_MAX*, то результат выполнения операции не специфицирован.

**Замечание**

При использовании функции *read(...)* необходимо анализировать системные ограниче- ния конкретной ОС: максимальный размер выделяемого буфера для записи читаемых данных и максимальное число читаемых данных за одну операци.

Системный вызов *write(...)* записывает заданное число байт из буфера в файл, определенный его дескриптором. В случае успешного завершения операции, дан- ные помещаются в файл, начиная с текущей позиции, и возвращается реальное чис- ло записанных байт.

В этом системном вызове используются теже аргументы, что и в рассмот- ренной ранее функции *read(...)*. В случае ошибки записи, возвращается значение -1 и необходимо анализировать переменную *errno*.

Если данные нужно поместить в файл с нужной позиции, то предварительно нужно использовать системный вызов

Дополнительно, на результат вывода могут влиять различные системные ограничения ОС, например, *максимальный размер файлов*, поддерживаемых той или иной файловой системой. Ряд таких значений можно посмотреть с помощью утилиты ***prlimit***, которая позволяет как получать так и устанавливать ряд лимитов процессов ОС.

Демонстрацию применения изученных в данном подразделе системных функций проведем на примере задачи «*Чтение и анализ структуры MBR*».

Напомним, что ***MBR*** является *заголовочной структурой* каждого накопителя типа «*винчестер*», совместимая с функциональными возможностями BIOS ЭВМ. В новых системах, которые cодерживают ***UEFI*** вместо ***BIOS***, возможна поддержка накопителей со структурой ***GPT***, куда MBR входит как вспомогательная структура, предназначенная для защиты блочного устройства.

MBR содержит *6 основных частей*:

* *код* System Bootstrap — 446 байт;
* *четыре описателя* разделов по 16 байт;
* *сигнатуру* MBR — 2 байта.

**Замечание** Запуск всех программ для работы с устройствами должен проводиться от имени пользователя *root*, иначе они будут завершаться из-за отсутствия прав доступа к этим устройствам.

### Системный вызов lseek()

Системный вызов *lseek(...)* устанавливает позицирование в файле, опреде- ленном его дескриптором. В случае успешного завершения операции, возвращается *место новой позиции в байтах* от начала файла.

Для вызова этой функции используется три аргумента: *fd* — целочисленный номер дескриптора открытого файла; *offset* — целочисленное значение смещения в байтах; *whence* — директива, задающая интерпретацию смещения:

* *SEEK\_SET* = 0 - смещение устанавливается в *offset* байт;
* *SEEK\_CUR* = 1 - смещение устанавливается как текущее смещение плюс

*offset* байт;

* *SEEK\_END* = 2 - смещение устанавливается как размер файла плюс *offset*

байт.

В случае ошибки, функция *lseek(...)* возвращает значение -1 и необходимо анализировать переменную *errno*:

**Замечание** Функция *lseek()* позволяет задавать смещения, которые будут находиться за сущест- вующим концом файла, *но это не изменяет размер файла*. Если позднее по этому смещению будут записаны данные, то последующее чтение в промежутке от конца файла до этого смещения, будет возвращать нулевые байты, пока в этот промежуток не будут фактически записаны данные.

## Создание специальных файлов

Сама ***VFS*** реализована в ядре ОС и все операции с файлами осуществляются в пространстве ее представлений. Фактически, ***VFS***, является *виртуальным коммутатором* всех внешних файловых систем, подклю- ченных (монтированных) к ней.

**В идейном плане**, структура ***VFS*** очень близка к структуре ***EXT2FS***, которая и является прямым ее прототипом.

**Напомним**, что структурную основу ***EXT2FS***, как и других файловых систем Linux/UNIX, играет *таблица индексных дескрипторов* (Inode Tables), каждая строка которой содержит полное описание файла, кроме его имени:

* *когда файл создается* в файловой системе, то ему выделяется отдельная строка в Inode Tables;
* *номер этой строки* — *i-узел* (*inode*) и используется для дальнейшей работы с файлом, а *имя файла* и *значение его i-узла* записываются в соответствующий

файл директории, имеющий записи переменной длинны (см. далее таблицу 1.1);

* *когда файл открывается*, то ***VFS*** находит по имени файла значение *i-узла*,

запоминает его значение и возвращает пользователю *номер дескриптора файла*;

* *пользователь*, обращаясь к файлу, указывает его дескриптор файла, а ***VFS*** по номеру дескриптора находит значение *i-узла* файла.

Наряду с изученной ранее функцией *open(...)*, которая позволяет как откры- вать, так и создавать файлы, в стандарте POSIX имеется функция *mknod(...)*, обес- печивающая возможность создавать и различные специальные файлы.

**Замечание** Специальные файлы, после их создания, получают свои номера *i-узлов*, но, в отличие от обычных файлов, они в действительности есть только указатели на соответствующие драйверы устройств в ядре. По сравнению с обычными файлами файлы устройств имеют три дополнительных атрибута, которые характеризуют устройство, соответствую- щее данному файлу: *класс устройства*, *старший номер устройства* и *младший номер устройства*.

*Класс устройства* фактически сообвествует понятию тип устройства, опре- деленному выше.

*Старший номер устройства*, группирует типы устройств, например, жест- кий диск или звуковая плата.

Файлы устройств одного типа имеют одинаковые имена и различаются по но- меру, прибавляемому к имени. Например, все файлы сетевых плат Ethernet имеют имена, начинающиеся на **eth:** eth0, eth1 и другие.

*Младший номер устройства* применяется для нумерации устройств одного типа, другими словами, - устройств с одинаковыми старшими номерами.

## Запрос информации о статусе файлов

Изучив основные особенности файлов, перейдем к рассмотрению системного вызова *stat(...)*, который *по заданному имени файла* позволяет получить информа- цию о свойствах любых типов файлов.

Дополнительно, стандарт POSIX предоставляет функции:

* *fstat(...)* - которая вместо имени файла использует *номер его дескриптора*;
* *lstat(...)* - которая использует *имя ссылки на файл* и возвращяет информацию о самой ссылке, а не о файле, на который она ссылается.

Ряд полей этой структуры требует уточнения:

* *st\_dev* — описывает устройство, на котором находится этот файл; поскольку это поле имеет тип *dev\_t*, то для работы с ним необходимо пользоваться мак- росами *major(…)*, *miner(...)* и *mkdev(...)*, описанными ниже;
* *st\_rdev* - описывает устройство, которое представляет этот файл (inode);
* *st\_size* - задает размер файла в байтах, если он обычный или является сим- вольной ссылкой; *размер символьной ссылки* - длина пути файла на который

она сылается, без конечного NUL;

* *st\_blocks* - задает размер файла *в 512-байтных блоках*: оно может быть мень- ше, чем *st\_size*/512, например, когда в файле есть пропуски;
* *st\_blksize* - задает "предпочтительный" размер блока для эффективного ввода/ вывода в файловой системе: запись в файл более мелкими порциями может

привести к некорректному *чтению/изменению/повторной* записи информации;

* *более точную семантику других полей* следует изучать по руководству ***man***.

**В стандарте** POSIX предусмотрены макросы, которые проверяют, является ли файл:

* S\_ISLNK(m) — символьной ссылкой;
* S\_ISREG(m) — обычным файлом;
* S\_ISDIR(m) — каталогом;
* S\_ISCHR(m) — символьным устройством;
* S\_ISBLK(m) — блочным устройством;
* S\_ISFIFO(m) — каналом FIFO;
* S\_ISSOCK(m) — сокетом.

**Для поля** *st\_mode*, в стандарте POSIX для программирования на языке С пре- дусмотрены также флаги

**В случае** успеха, все функции возвращают *ноль*.

**При обнаружении** ошибки возвращается *-1*, а переменной *errno* присваива- ется номер ошибки:

Поскольку поля *st\_dev* и *st\_rdev* имеют тип *dev\_t*, который может отличаться для разных аппаратных платформ, следует пользоваться макросами, присутству- ющими в система разработки большинства ОС:

## Каналы

Операционные системы предоставляют программисту два типа каналов:

* *полудуплексные каналы* UNIX;
* *именованные каналы* FIFO.

### Полудуплексные каналы UNIX

Изучая командный интерпретатор shell, мы уже сталкивались с понятием каналов. Например, если в командной строке запустить: ***ls -l | grep src***, то ин- терпретатор *sh* или *bash* будут выполнять следующую последовательность дейст- вий:

* будут одновременно запускатся утилиты *ls*, обеспечивающая вывод подроб- ного списка файлов текущей директории, и утилита *grep*, фильтрующая вход-

ной поток данных по наличию слова *src*, в каждой строке;

* перед вызовом утилит, shell создаст полудуплексный каканал, обеспечив воз- можность записи в него для утилиты *ls* и чтение из него для утилиты *grep*.

**Каналы** - старейший из инструментов IPC, существующий приблизительно со времени появления самых ранних версий оперативной системы UNIX.

Они предоставляют метод односторонних коммуникаций между процессами: отсюда появился термин *half-duplex channel*.

Сам канал создается в ядре ОС. Когда процесс создает канал, ядро уста- навливает *два файловых дескриптора* для пользования этим каналом. Один такой дескриптор используется, чтобы открыть *путь ввода в канал* (запись), в то время как другой применяется *для получения данных из канала* (чтение).

Если открыт массив дескрипторов *int fd[2]* и процесс посылает данные через канал *fd[1]*, то он имеет возможность получить эту информацию из деск- риптора *fd[0]*.

Создание неименованного канала выполняется с помощью функции *pipe(...)*

Основное назначение *неименованных полудуплексных каналов* — *взаимодей- ствие между родительским и дочерними процессами*. В этом случае используется то свойство, что дочерний процесс наследует все ресурсы, открытые родительским

процессом, в том числе и открытые каналы.

**Если предположить**, что родительский процесс должен передать некоторое сообщение дочернему процессу, а потом дождаться его завершения, то в этом случае:

* родителскому процессу *не нужет дескриптор чтения из канала*;
* дочернему процессу *не нужен дескриптор записи в канал*.

**Замечание**

Следует помнить:

* с каналами не работает функция *lseek(...)*;
* системные вызовы не всегда корректно работают с консолями систем разработки, поэтому программы лучше запускать в терминале из командной строки.

### Именованные каналы FIFO

*Неименованные каналы* UNIX позволяют общаться только родственным друг другу процессам, которые получены с помощью *fork()*.

Для целей взаимодействия не родственных («*чужих*») процессов предназ- начены *именованные каналы FIFO*, которые также создаются в ядре ОС, но имеют *имена, отображаемые в файловой системе*.

Типичное применение каналов FIFO - разработка приложений «*клиент — сервер*», когда:

* *несколько процессов* могут записывать или читать FIFO *одновременно*;
* *режим работы* с FIFO - *полудуплексный*, что позволяет процессам общаться только в одном из направлений.

**Создать** канал FIFO можно с помощью функции *mknod(...)*, если во втором параметре *mode* установить тип файла *S\_IFIFO*, а третьему параметру *dev* присвоить значение *0*.

Стандарт POSIX предоставляет для создания именованных каналов специ- альный системный вызов *mkfifo(...)*

Перед использованием каналы FIFO требуют своего открытия с помощью системного вызова *open(…)*. Если при этом не указать режим *O\_NONBLOCK*, то:

* *открытие* FIFO блокируется и для записи, и для чтения;
* *при записи* канал блокируется до тех пор, пока другой процесс не откроет FIFO для чтения;
* *при чтении* канал снова блокируется до тех пор, пока другой процесс не запишет данные;
* *если FIFO закрыть для записи* через *close(...)* или *fclose(...)*, то это значит, что для чтения в FIFO помещается *EOF*.

Флаг открытия канала *O\_NONBLOCK* может использоваться *только при доступе для чтения*. При попытке открыть FIFO с *O\_NONBLOCK* для записи возникает ошибка открытия.

Если несколько процессов пишут в один и тот же FIFO, необходимо обратить внимание на то, чтобы сразу не записывалось больше, чем *PIPE\_BUF* байтов. Это необходимо, чтобы данные не смешивались друг с другом.

**Чтобы** не запускать множество программ, рассмотрим работу каналов FIFO на примере двух родственных процессов:

* *программа запускается* и создает канал FIFO с именем */dev/home/cfifo*, а за- тем выполняет *fork()*, создавая дочерний процесс;
* *родительский процесс* открывает канал на чтение, читает данные и выводит их на терминал; если прочитано менее 2-х байт, то — ожидает завершение дочернего процесса, а затем завершается сама;
* *дочерний процесс*, в цикле, читает строку данных с клавиатуры, открывает канал для записи, пишет в него данные и закрывает канал; если с клавиатуры

прочитано менее 2-х байт, то, после передачи их в канал, завершает свою ра- боту.

## Дублирование дескрипторов файлов

Известно, что:

* *интерпретатор shell*, запуская процесс, передает ему три дескриптора фай- лов: 0 — для чтения; 1 и 2 — для записи;
* *функция open(...)*, в случае нормального завершения, возвращает наименьший доступный номер дескриптора;

В практических задачах часто возникает необходимость дублирования деск- рипторов или открытие дескриптора с заданным номером. Например, серверные *программы*, принимая внешние к ним запросы, могут сами не иметь алгоритмов обработки файлов, а обращаются к другим программам. В этом случае:

* *программа обработки алгоритма* пищется и отлаживается при условии, что ввод осущесвляется из дескриптора с номером 0, а вывод осуществляется в

файл с дескриптором 1;

* *серверная программа*, для обработки запроса, создает дочерний процесс, ко- торый: открывает файл ввода информации, закрывает дескриптор 0 и дубли-

рует дескриптор открытого файла, обеспечивая ввод данных по дескриптору 0; аналогичные действия проводятся с дескриптором файла вывода; после этого, посредством одного из системных вызовов *exec(…)*, осуществляется запуск *программы обработки алгоритма*.

Стандарт POSIX предоставляет два системных вызова дублирования деск- рипторов файлов:

#include <unistd.h>

int dup (int *oldfd*);

int dup2(int *oldfd*, int *newfd*);

где oldfd — старый дескриптор файла; newfd — новый дескриптор файла.

Обе функции, в случае нормального завершения вызова:

* возвращают новый дескриптор файла;
* функция *dup2(…)*, если требуется, предварительно закрывает старый деск- риптор *oldfd*.

Старый и новый дескрипторы можно использовать друг вместо друга. Они имеют общие блокировки, указатель позиции в файле и флаги; например, если позиция в файле была изменена с помощью **lseek**, на одном из дескрипторов, то эта позиция также меняется и на втором.

Если произошла ошибка, то возвращается значение -1 и переменная *errno*

устанавливается должным образом..

## Монтирование и демонтирование ФС

Все ОС имеют системные вызовы, предназначенные для *подключения* и *отключения* внешних файловых систем к *корневой файловой системе*. Стандарт POSIX определяет эти функции как:

#include <sys/mount.h>

int mount(const char \*source, const char \*target, unsigned mountflags);

int umount(const char \*special);

где *source* — раздел блочного устройства или файл, содержащий файловую систе- му, известную ядру ОС;

*target* — директория, присутствующая в файловой системе, к которой монтируется внешняя файловая система;

*mountflags* — опции, с которыми монтируется файловая система;

*special* — это *source* или *target*, которые специфицируют точку монтирования.

При удачном завершении вызова возращаемое значение равно нулю. При ошибке возвращается -1, а переменной *errno* присваивается номер ошибки.

**Замечание** В предыдущих темах были подробно изучены утилиты ***mount*** и ***umount***, предназначен- ные для аналогичных целей.

Программисту, *без большой необходимости*, не следует не следует использовать эти системные вызовы, потому что:

* *основные привилегии* создания структуры ФС и защита такой структуры обеспе- чивается администратором ОС, что исключает необходимость оперативного мон-

тирования и демонтирования из программы;

* *синтаксис и реализация* этих системных вызовов всегда специфичен для конк- ретной ОС, можно убедится при изучении ***man 2 mount*** и ***man 2 umount***;
* *в крайнем случае*, можно с помощью дочернего процесса вызвать «родные» ути- литы ОС, что повысит переносимость разработанного ПО.

## Ссылки на имена файлов

Программисту и обычному пользователю, любая отдельная файловая система ОС представляется в виде иерархии имен файлов, в которой особое место зани- мают *файлы типа директорий* (*каталогов*):

* именно *с корневого каталога* начинается каждая файловая система, разме- щенная на внешних носителях информации;
* именно *к каталогам монтируются* разные файловые системы, образуя еди- ное дерево файловой системы ОС, вершина которой (*корневой каталог VFS*)

формируется в ядре ОС;

* именно *в записях каталогов* имя файла (*name*) связано с i-узлом (*inode*).

В стандарте POSIX для создания и удаления каталогов определяет два сис-

темных вызова: *mkdir(...)* и *rmdir(...)*.

Системный вызов *mkdir(...)* имеет синтаксис:

#include <sys/stat.h> #include <sys/types.h>

int mkdir(const char \*pathname, mode\_t mode);

где *pathname* — имя создаваемой каталога;

*mode* — задает права доступа, которые получит зданный каталог; эти права стан- дартным образом модифицируются с помощью *umask*: права доступа оказываются равны *mode & ~umask*.

Созданный каталог принадлежит фактическому владельцу процесса и насле- дует его права. При успешном завершении возвращается ноль или -1, если про- изошла ошибка. В этом случае *errno* устанавливается следующим образом:

**Замечание** Создать файл каталог можно и спомощью системного вызова *open(...)* или, напри-мер, *mknod(...)*,но такой каталог будет «*неполноценным*», поскольку в нем будут от- сутствовать файлы «.» и «..».

Системный вызов *rmdir(...)* имеет синтаксис:

#include <unistd.h>

int rmdir(const char \*pathname);

где *pathname* — имя удаляемого каталога, который должен быть пустым, кроме файлы «.» и «..».

**В случае успеха** возвращается ноль. При ошибке возвращается -1, а *errno*

устанавливается следующим образом:

**Для удаления имен файлов**, в стандарте POSIX, предусмотрен системный вызов *unlink(...)*, имеющий следующий синтаксис:

#include <unistd.h>

int unlink(const char \*pathname);

где *pathname* — имя удаляемого файла.

**В случае успеха** возвращается ноль. При ошибке возвращается -1, а *errno*

устанавливается следующим образом:

**Замечание**

Удаляя имя из файловой системы, *unlink(...)* учитывает, что:

* если это имя было последней ссылкой на файл и больше нет процессов, которые держат этот файл открытым, данный файл удаляется и место, которое он зани- мает освобождается для дальнейшего использования;
* если имя было посленей ссылкой на файл, но какие-либо процессы все еще дер- жат этот файл открым, файл будет оставлен пока последний файловый дескрип- тор, указывающий на него, не будет закрыт;
* если имя указывает на символьную ссылку, ссылка будет удалена;
* если имя указывает на сокет, FIFO или устройство, имя будет удалено, но про- цессы, которые открыли любой из этих объектов могут продолжать его использо-

вать.

**Таким образом**, мы видим, что в общем случае, *один файл может иметь несколько имен*.

Для создания новых имен, которые интерпретируются как имена файлов, стандарт POSIX предоставляет два системных вызова:

* *link()* - создает *новую ссылку* на существующий файл (*на i-узел*) внутри одной файловой системы, известную также как "*жесткая*" ссылка; она имеет теже

свойства, что и старое имя, поэтому их оригинальность их установить невоз- можно;

* *symlink()* - создает *символьную ссылку на имя* предположительно существую- щего файла, независимо от файловой системы, в которой оно должно нахо-

диться; если такого файла не существует, то ссылка называется «*висячей*». Системный вызов *link(...)* имеет следующий синтаксис:

#include <unistd.h>

int link(const char \*oldpath, const char \*newpath);

где *oldpath* — имя существующего файла;

*newpath* — новое имя файла, в тойже файловой системе, что и старое.

**В случае успеха** возвращается ноль. При ошибке возвращается -1, а *errno*

устанавливается следующим образом:

Системный вызов *symlink(...)* имеет следующий синтаксис:

#include <unistd.h>

int symlink(const char \*oldpath, const char \*newpath);

где *oldpath* — имя существующего файла;

*newpath* — новое имя файла, в тойже файловой системе, что и старое.

**В случае успеха** возвращается ноль. При ошибке возвращается -1, а *errno*

устанавливается следующим образом:

**Замечание**

При использовании *symlink(...)*:

* *не производится* никакой проверки *oldpath;*
* *удаление файла*, на который ссылается символьная ссылка, действительно уда- лит файл, если только у него нет других жестких ссылок; если такое поведение

нежелательно, то следует использовать *link(...)*.

**Замечание**

На практике, часто используются и другие функции для работы с файловой системой ОС, которые можно отнести к системным вызовам. Например, *chdir(...)*, *chroot(...)*, *rename(…)* и другие.

## Классификация способов управления ОЗУ

В первых вариантах компьютеров, *ОС как таковые — отсутствовали*:

* + - *программа пользователя* загружалась в ОЗУ с адреса, который указывал оператор ЭВМ;
    - *после загрузки* программы, оператор указывал на пульте ЭВМ стартовый адрес программы и нажимал кнопку «Пуск»;
    - *после выполнения* текущей программы, в ЭВМ загружадась новая программа. Фактически, в таком (*пакетном*) режиме работы ЭВМ, управление памятью не проводилось или проводилось в упрощенных вариантах.

С появлением и совершенствованием первых ОС, которые также, как и прик- ладное ПО, являются программным обеспечением, стали применять *три способа загрузки*,

*Развите этих способов* загрузки ПО показано слева на право:

* + - *певый способ* предполагает первоначальную загрузку ОС, обычно в младшие адреса ОЗУ; такой способ применялся в *мэйнфреймах* и первых миником-

пьютерах; программы пользователя загружались с помощью ПО ОС в ос- тальную часть ОЗУ;

* + - *второй способ* стал применяться в некоторых карманных компьютерах и

*встроенных системах*, которые обладают малыми ресурсами и, как правило,

разными конструктивными особенностями; здесь ОС пишется в ПЗУ, которое соответствует верхним адресам памяти, оставляя ОЗУ полностью для ПО пользователей;

* + - *третий способ* соответствует *большинству современных систем*, подобных IBM PC; в ПЗУ записывается ПО BIOS, которое работает только при включе-

нии питания компьютера; ПО BIOS находит и загружает в младшие адреса ОЗУ ПО ОС; программам пользователей предоставляется оставшееся ОЗУ *до 640 Кбайт*, в младших адресах, и вся оставшаяся память *свыше первого Мбайт*; память ОЗУ между 640 Кб и 1 Мб (*скрытая область*) распреде- ляется между ПО BIOS и памятью прямого доступа для внешних устройств.

Для компьютера, память (ОЗУ) всегда была и остается *дефицитом*:

* + - *всегда имеются программы*, которым, по тем или иным причинам, недоста- точно выделенного объема ОЗУ ЭВМ;
    - *проблемы реализации языков программирования* требуют решать многие воп- росы, связанные с адресацией памяти и ее распределением.

Для решения этих и многих других вопросов было введено *понятие виртуальной памяти*.

**Виртуальная память**, призвана *разрешить противоречие в адресации команд и данных*, связанных с потребностями прикладного ПО и ресурсами ОЗУ, которые имеет конкретная ЭВМ.

Идея состоит в использовании *таблицы виртуальных адресов*, которая указы- вает адресам всех команд и данных, выставленных компилятором, соответствие физических адресов ОЗУ.

Реализация такого соответствия может выполняться:

* + - специальным загрузчиком (*статически*);
    - или специальными аппаратными средствами (*динамически*).

Появление ОС, поддерживающих *мультипрограммирование*, потребовало ре- шения вопросов распределения памяти ОЗУ между многими одновременно выпол- няемыми программами (*процессами*). На рисунке 1.5, приведена классификация различных подходов, решающих поставленную проблему.

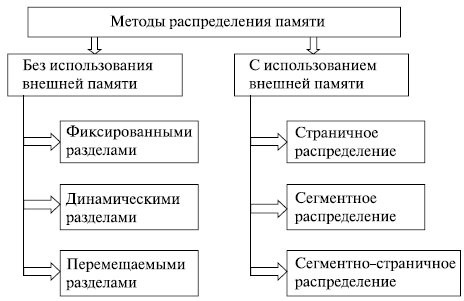


Рисунок 1.5 - Классификация способов распределения памяти Здесь можно выделить две большие группы подходов:

* + - методы, *не использующие внешнюю память*, обычно, накопители на жестких дисках;
    - методы, *использующие внешнюю память*, которая теоретически продолжает (расширяет, увеличивает) ОЗУ ЭВМ.

*Методы, не использующие* внешнюю память, предполагают, что *ОЗУ доста- точно для решения самой крупной задачи*, или, что программу можно исполнять по частям, для которых ОЗУ достаточно. Здесь, в большинстве случаев, задачи распре- деления памяти ставятся как *оптимальное разбиение ОЗУ на сегменты* (*отдельные и не пересекающиеся участки памяти*), обеспечивающие максимальную эффектив- ность использования процессорного времени ЭВМ.

*Методы, использующие* внешнюю память, предполагают, что ОЗУ расширя- ется за счет памяти медленных внешних устройств и становится возможным однов- ременное выполнение необходимого числа «параллельно» работающих процессов.

Здесь, в большинстве случаев, задачи распределения памяти ставятся как ограниче- ния на скорость выполнения отдельных процессов или эффективность использова- ния ОЗУ ЭВМ по управлению самим процессом распределения памяти ЭВМ.

В любом из указанных случаев, ОЗУ расходуется не идеально, а эффектив- ность используемых идей сильно зависит как от величины соотношения *объем ОЗУ/размер процессов*, так и от *аппаратной реализации поддержки* указанных подходов.

**Основными причинами** неэффективности использования ОЗУ являются:

* + - *сильная фрагментация памяти* как во время загрузки, так и во время завер- шения процессов;
    - *затраты, связанные с устранением последствий* такой фрагментации.

**Замечание** Фрагментация вызывается не только из-за различных размеров запускаемых про- цессов, но и, по-причине, *дополнительных требований ряда процессов к динами- ческому выделению дополнительной памяти ОЗУ*.

**В целом**, можно выделить ряд общих задач, по распределению памяти ЭВМ, которые должны решаться каждой ОС:

* + - отслеживание (учет) свободной и занятой памяти;
    - первоначальное и динамическое выделение памяти процессам приложений и самой операционной системе, а также освобождение памяти по завершении

процессов;

* + - настройка адресов программы на конкретную область физической памяти;
    - полное или частичное вытеснение кодов и данных процессов из ОЗУ на диск, когда размеры ОЗУ недостаточны для размещения всех процессов, а также

возвращение этих процессов в ОЗУ;

* + - защита памяти, выделенной процессу, от возможных вмешательств со сторо- ны других процессов;
    - дефрагментация памяти.

Решение всех указанных задач возложено на «*Подсистему распределения памяти*»,

которая, в свою очередь, входит в «*Подсистему управления процессами*» ядра ОС.

## Программный и аппаратный способы адресации памяти

Нормальная работа компьютера обеспечивается двумя способами адресации памяти ЭВМ: *программным* и *аппаратным*.

*Программный способ адресации неявно присутствует* в любом исходном тексте программы в виде операторов команд программы, которые обращаются к объявленным переменным данных и осуществляют различные управляющие дейст- вия по определению порядка выполнения самих команд. В последствии, компиля- тор обеспечит *явное присутствие* как адресов команд, так и адресов данных.

Соответственно, в каждом отдельном процессе, *минимально* можно выде-

лить:

* + - управляющий блок процесса;
    - сегмент кода (программы);
    - сегмент данных;
    - стек;
    - «куча», если процесс использует динамическое выделение памяти.

Аппаратный способ адресации заложен сам процессор:

* + - *специализацией регистров процессора*: сегментные, смещения и индексные;
    - *методами адресации команд процессора*;
    - *шинами* (каналами) *процессора*.

Реализация такой схемы имеет *недостатки*, поскольку:

* + - *каждым, из выделенных блоков*, необходимо управлять;
    - *каждый, из выделенных блоков*, использует относительную адресацию, кото- рая требует использования сумматоров для каждой команды и данных про-

цесса;

* + - *каждая команда или обращение данных* требуют проверки на возможность нарушения границ, выделенных областей.

Таким образом, эффективность использования процессора значительно снижается, по-причине использования *сложной системы адресации памяти*.

Чтобы устранить эти недостатки, во всех разви-тых архитектурах ЭВМ используются *аппаратные средства виртуализации памяти* (***MMU***). Это освобождает процессор от указанной выше рутинной работы.

**MMU** — *Memory Managment Unit* — *диспетчер памяти* уже был рассмотрен ранее, при анализе прямого доступа к памяти внешних устройств. В его обязан- ности входит:

* + - виртуализация адреса, при взаимодействии процессора с ОЗУ;
    - аппаратное обеспечение алгоритмов распределения блоков ОЗУ в различных режмах адресации.

## Страничная и сегментная адресации памяти

Исторически, для управления памятью ОЗУ ЭВМ, в ОС стала разрабаты- ваться *сегментная адресация*.

Сегментная адресация ОЗУ реализуется с помощью *специальных регистров* процессора, которые так и называются: *регистры сегментов* и *регистры смещения*. *Сегментный способ адресации ОЗУ* всегда использует таблицу сегментов, которая сама размещается в ОЗУ и предназначена для решения двух основных

задач:

* + - виртуализация относительных адресов программы относительно физической памяти ЭВМ;
    - управление свопингом и перемещение сегментов в физическом пространстве ОЗУ.

**Задачи виртуализации памяти** могут решаться как с привлечением, так и без привлечения специальных аппаратных средств (***MMU***).

**Управление свопингом** — управление перемещением сегментов программ из ОЗУ в специальные файлы или разделы свопинга, расположенные на внешних ЗУ, и обратно.

Для этой цели, таблица сегментов должна содержит управляющую информа- цию, которая указывает не только на наличие или отсутствие сегментв в ОЗУ, но и используется для целей защиты или разделения памяти между процессами.

**Замечание** *Основная проблема* сегментгой адресации ОЗУ заключается в ее *фрагментации*, кото- рая вызвана *разным размером* загружаемых и выгружаемых сегментов:

* + - *при равных размерах* сегментов, фрагментация была бы минимальна или ее удалось бы устранить;
    - *чем меньше разделы сегментов*, тем меньше становятся непроизводительные потери ОЗУ.

Идея использования блоков одинакового размера, для адресации физической и логической частей ОЗУ, стала называться *страничной организацией памяти*.

*Основное достоинство* страничной адресации ОЗУ - *отсутствие фрагмен- тации*, что не требует затрат ресурса ЭВМ на ее устранение.

схема адресации ОЗУ с помощью страниц очень похожа на схему адресации ОЗУ с помощью сегментов, поскольку в обоих случаях используются таблицы переадресации, которые должны размещаться в той же ОЗУ.

**Замечание** В связи с малым размером страницы, обычно 4 Кбайт, *увеличивается скорость пэйджинга*: перемещения страницы на внешний носитель ЗУ и обратно.

**Пэйджинг** (*pagging*) - свопинг (*swapping*) страниц.

**По традиции**, файлы или разделы винчестера, *куда осуществляется pagging*, называ- ются файлами или разделами свопинга.

**Очевидно**, что аппратные средства виртуализации (***MMU***) способны обеспечивать как

*pagging*, так и *swapping*.

*Основная проблема* адресации с помощью страниц - *большой размер таблиц адресации*, которые должны храниться в ОЗУ.

Чтобы частично устранить эту проблему, стали использовать *двухуровневую схему страничной адресации*, ставшую основой адресации 32-битных ЭВМ:

* + - *в ОЗУ постоянно находится* только корневая таблица страниц, содержащая не более 1024 записи;
    - *когда программа загружается* на выполнение, в корневую таблицу страниц заностися запись о местоположении и числе требуемых ей страниц;
    - *внутри самой программы*, создается своя собственная таблица, которая заполняется во время загрузки программы и удаляется после ее завершения.

*Другая проблема*, с которой сталкиется страничный способ адресации — значительные затраты времени, связанные с поиском информации о странице в больших таблицах страниц.

Чтобы разрешить эту проблему, стали использовать *таблицу быстрой транс- ляции адресов* или — *буфер TLB*.

**TLB** — *Translation Lookaside Buffer* — ассоциативная кэш-память, в которую записывается реально используемая страница, перед тем как команды и данные будет использовать сам процессор.

Алгоритм использования TLB — прост:

* + - когда процессор обращается к странице и *она присутствует в TLB*, то используется команды и данные, считанные из нее;
    - когда процессор обращается к странице и *она отсутствует в TLB*, возникает прерывание к соответствующей программе ядра ОС;
    - обработка прерываний осуществляется «*Подсистемой управления памятью ОС*», которая производит поиск нужной страницы в ОЗУ или на внешнем

носителе, решает вопрос об удалении ненужной страницы и записи нужной, перезапускает команду процессора, вызвавшую прерывание по отсутствию страницы в TLB.

### Замечание

К сожалению, проблема управления страницами не решается так просто, например, для 64-разрядных ЭВМ создание таблицы страниц становится *не реализуемым*.

Чтобы теоретически закрыть изучаемую тему, рассмотрим комбинированный способ адресации.

## Комбинированный способ адресации памяти

Преимущества адресации имеются у каждого способа:

* + - сегментная адресация памяти — требует относительно малое число записей в таблице сегментов.
    - страничная адресация памяти — устраняет проблемы фрагментации, пос- кольку ОЗУ адресуется с помощью *произвольного доступа*, следовательно, свободную страницу ОЗУ можно найти в любом месте памяти.

Отсюда возникает *идея смешанной (комбинированной) адресации*, которая состоит в том, что:

* + - *отдельная программа* (процесс) отображается в виде нескольких записей в общей таблице сегментов;
    - *кажный сегмент*, выделенный программе, отображается внутри нее в виде

одной или нескольких таблиц страниц.

**Таким образом**, сегментно-страничная адресация памяти похожа на *подоб- ную двухуровневую страничную адресацию*. При этом:

* + - *существенно уменьшается размер* корневой таблицы, постоянно хранящейся в ОЗУ и как следствие ускоряется поиск в ней нужной страницы;
    - *трехкомпонентная адресация* (номер сегмента, номер страницы, смещение) требует аппаратной поддержки виртуализации памяти с помощью MMU, без которой она становится не эффективной;
    - *появляется возможность оптимизации* схем адресации на базе учета разных свойств разных сегментов программы; например, *сегменты кода предназна- чены только для чтения*.

**В общем случае**, применяемая схема адресации ОЗУ зависит не только от конструктивных возможностей физической памяти или аппаратной поддержки вируализации памяти с помощью устройств MMU, но и от *конструктивных особенностей самого процессора*.

*Такая зависимость*, стала поддерживаться процессорами компании Intel, начиная с архитектуры i386. Здесь выделяются:

* + - глобальная таблица дескрипторов (***GDT***), на которую должен указывать

специальный регистр процессора — ***GDTR***;

* + - локальные таблицы дескрипторов (***LDT***), на которые должен указывать специальный регистр процессора — ***LDTR***.

*Глобальная дескрипторная таблица* является единственной и контролизуется ядром ОС, адресуя сегменты процесса и значения регистра LDTR.

*Локальные дескрипторные таблицы* связаны с конкретными испольняемыми процессами и формируются при их запуске.

Указанная схема адресации, широко используется в ОС MS Windows, по крайней мере, начиная с Windows 2000, выполненной по технологии NT.

**Замечание** В качестве указателей на GDT и LDT, используются *16-разрядные слова*, в которых младшие три бита имеют специальное значение:

* + - *0-й и 1-й биты* — уровень защиты (уровень привилегий) работающей задачи;
    - *2-й бит* используется для различения GDT и LDT.

следует понимать, что *ОС «скрывает»* от процессов детали способов адресации памяти, возлагая реализацию ее распределения на «*Подсистему управления памятью*».

## Системные вызовы ОС по управлению памятью

## Структура поцесса

Как было отмечено ранее, основная память (ОП) или ее рассматриваемя часть ОЗУ занимает особое положение в плане теории и практики ее применения.

Причина такой особоенности состоит в том, что для управления чем-либо, включая распределение ОЗУ, необходимо программное обеспечение, которое само должно быть загружено в ОЗУ. Поэтому для программ (процессов), выполняющих- ся в пользовательском режиме, многие аспекты распределения памяти или скрыты совсем, или проявляются только косвенно.

*Виртуальная структура процесса* состоит из отдельных частей, которые, по- традиции, также называются сегментами:

* *cегмент text* - содержит машинные инструкции и константы; его содержание не изменяется при выполнения процесса;
* *сегмент data* - содержит внешние и статические переменные, *которые ини- циализированы*, при компиляции программы процесса;
* *сегмент bss* - содержит внешние и статические переменные, *которые неини- циализированы*, при компиляции программы процесса;
* *сегмент stack* - используется для хранения значений автоматических пере- менных и параметров функций, а также для организации рекурсивных вызовов функций.

**Замечание**

Между нижней границей стека и сегментов данных обычно существует *адресный зазор*, который может быть использован при выполнении процесса для расширения сегментов стека и данных. Такое расширение возможно, пока не исчерпано пространство свобод- ных адресов.

Когда процесс создан, то измение размера его сегментов осуществляется в случаях загрузки *нового тела процесса*. Если новое тело не загружается, то:

* изменение размера *сегмента кода процесса* не происходит, поскольку этот сегмент предназначен только для чтения;
* изменение размера *сегментов данных* осуществляется за счет сегмента BSS и, обычно, реализуется системными вызовами динамического распределения памяти *brk(...)* и *sbrk(...)*;
* изменение размеров *сегментов стеков* осуществляется автоматически, во время выполнения процесса.

**Замечание**

*Считается*, что ОС Linux придерживается идеологии ОС UNIX и, поэтому, должна иметь

одинаковую структуру сегментов процесса.

*На самом деле*, - это не так, хотя общая идея выделения *сегмента кода* и *сегмента данных + стека* — присутствует.

В общем случае, учесть все детали распределения сегментов процессов — довольно проблематично, тем более, что эти детали могут меняться от одной ОС к другой.

## Определяемые сегменты процесса

Как отмечено ранее, любой программе пользователя для своей работы необ- ходима память ОЗУ. Память ОЗУ выделяется программе в виде сегментов или в виде страниц, или другим комбинированным способом. Все разнообразие способов выделения памяти программы учитывается в ядре ОС и доступна программистам, занимающимся разработкой ПО ядра. Кроме того, имеются три параметра любого процесса, которые доступны любой программе на языке ***С***. Такими параметрами (*внешними переменными*) являются:

* *etext* — первый адрес, следующий за последним адресом сегмента кода про- цесса (*text segment*);
* *edata* — первый адрес, следующий за последним адресом сегмента иници- ализированных данных процесса (*initialized data segment*);
* *end* — первый адрес, следующий за последним адресом сегмента не иници- ализированных данных процесса (*uninitialized data segment*), известный также как *BSS*.

Чтобы подробно изучить назначение описанных выше внешних переменных, сле- дует воспользоваться командой: ***man etext***. Там же, приведены примеры их исполь- зования.

**Замечание**

Естественно, что стандартные средства языка ***С*** не отражают многих характеристик реальных процессов ОС, поскольку эти характеристики связаны с особенностями реализации конкретных систем.

В следующем подразделе, мы рассмотрим характеристики процессов 64-битных ОС Linux, которой также является ОС УПК АСУ.

Значения многих параметров процесса можно посмотреть непосредственно, используя возможности «просмотрищика» файлого менеджера «*Midnight Comman- der*».

## Создание и удаление процессов из памяти

Структура процессов ОС Linux следует современным тенденциям построе- ния операционных систем, в плане использования ими основной памяти ЭВМ:

* *они полностью используют* виртуальную адресацию памяти;
* *ядро ОС фиксирует* основные характеристики всех запущенных процессов.

В настоящее время, ядро ОС использует *32-битную виртуальную адресацию*

пространства процесса, которая составляет размер в 4 ГБ, как показано на рисунке 2.3:

* *первые 3 ГБ* соответствуют пространству пользователя (*User space*);
* *последний 1 ГБ* находится в распоряжении ядра ОС (*Kernel space*).

В ядре ОС, для каждого процесса отведена *директория страниц*, что подра- зумевает возможность доступа к *1КВ таблицам страниц*, которые указывают на *1МВ 4-х килобайтных страниц*. Это позволяет адресовать 4 ГБ памяти.

Каждый пользовательский процесс имеет свою *локальную таблицу дескрип- тора*, которая адресует различные *сегменты кода* и *сегменты данные-стек*.

В пользовательском пространстве, линейные адреса и логические адреса идентичны.

Процесс получает свои таблицы страниц от родителя, при выполнении пос- ледним системного вызова *fork()*, со входами, помеченными как *READ-ONLY* или *замещаемые*.

Если процесс пытается писать в некоторую область памяти и страница явля- ется *COPY-ON-WRITE* страницей, то она копируется и помечается как *READ- WRITE*.

В любом случае, «*Cистема управления памятью*» Linux осуществляет *подкачку страниц по обращению*, в соответствии со стратегией *COPY-ON-WRITE*, которая *основана на механизме подкачки* и поддерживается процессором i386.

Современные ядра ОС Linux псевдоустройство *proc*, с файловой системой типа *proc*, которая монтируется к директории /proc корневой ФС.

Эта файловая система содержит много информации о каждом запущенном процессе:

* директория */proc/<номер процесса>* - содержит файлы (псевдофайлы) о раз- ных характеристиках процесса с конкретным номером ***PID***;
* директория */proc/self* — является уникальной ссылкой для процесса, который обращается к информации о себе и который, как любой дочерний процесс, не знает свой ***PID***.
  1. **Динамическое выделение и освобождение памяти**

**процесса**

В предыдущих подразделах были рассмотрены случаи выделения памяти, которые осуществляются ядром ОС.

В данном подразделе, мы изучим системные вызовы ОС, обеспечивающие выделение памяти для нужд программиста. Для этих целей существует ряд функ- ций, позволяющих *выделять и освобождать память ОЗУ динамически*, во время выполнения программы.

Семантика этих функций - следующая:

* *malloc(...)* - выделяет *область неинициализированной памяти*, размер которой указан в байтах, в качестве аргумента, и возвращает значение указателя на

эту область; если значение аргумента равно нулю, то возвращается значение указателя *NULL*;

* *free(...)* — освобождает память ОЗУ, которая *ранее была выделена* одной из функций: *malloc(...)*, *calloc(...)* или *realoc(...)*; если аргумент функции равен

*NULL*, то никаких действий не производится;

* *calloc(...)* - выделяет память ОЗУ *под массив данных*, размерность которого задана первым аргументом, а размер каждого элемента массива в байтах,

задан вторым аргументом; при этом, значение каждого байта выделенной памяти обнуляется; возвращает указатель на выделенный массив данных; если значение любого из аргументов функции равно нулю, то память ОЗУ процессу не выделяется и возвращается значение *NULL*;

* *realloc(...)* - изменяет *размер уже выделенного блока памяти*, используя пер- вый аргумент как указатель, а второй — как нужный размер; если размер

области уменьшается, то значения байт оставшейся области не изменяется; если размер области увеличивается, то добавляемая область не инициали- зируетсся; если значение аргумента указателя равно *NULL*, то результат функции, соответствует действию функции *malloc(...)*; если требование на новый размер памяти равно нулю, а указатель не равен *NULL*, то действие эквивалентно функции *free(...)*.

Кроме перечисленных, имеются также системные функции *brk(...)* и *sbrk(...)*, с помощью которых можно непосредственно изменять *конец сегмента неинициа- лизированных данных*.

Семантика системных вызовов:

* *brk(...)* - устанавливает конец сегмента данных в значение, указанное в аргу- менте *addr*, когда это значение является приемлимым; указание адреса памя- ти должно быть в пределах максимально возможного размера сегмента дан-

ных (см. *setrlimit(2)*);

* *sbrk(...)* - увеличивает пространство данных программы на *increment* байт; *sbrk(...)* не является системным вызовом, он просто является оберткой (wrapper), которую использует библиотека ***C*** для вызова *brk(...)*; вызов

*sbrk(...)* с инкрементом *0* может быть использован, чтобы найти текущее местоположение прерывания программы.

В случае успеха *brk(...)* возвращает ноль, а *sbrk(...)* возвращает указатель *на начало новой области*. В случае ошибки, возвращается *-1* и значение *errno* устанавли- вается в значение *ENOMEM*.

## Подсистема управления процессами

Подсистема управления процессами находится в ядре ОС. Основная ее функ- ция — *обеспечение мультипрограммного режима работы ОС*, что связано с:

* *созданием* процессов в системе и удаление их из системы, так как это было рассмотрено в теме 8; предполагает управление основной памятью ЭВМ;
* *переключением* процессов в режимы «*Готовность*», «*Выполнение*» и «*Ожи- дание*».

Качественное выполнение этой функции требует *планирования подсистемой своих действий*, что в общем случае не является однозначно решаемой задачей.

Чтобы оценить сложность решаемой задачи планирования, рассмотрим пере- чень требований предъявляемых к ней, в зависимости от целевых аспектов различ- ных прикладных систем.

**Все системы** должны обеспечить:

* *Справедливость* — предоставление каждому процессу справедливой доли процессорного времени.
* *Принудительное применение политики* - контроль за выполнением принятой политики;
* *Баланс* — поддержка занятости всей системы.

**Системам пакетной обработки данных** необходима:

* *Пропускная способность* — максимальное количество задач в час.
* *Оборотное время* — минимизация времени, затрачиваемого на ожидание об- служивания и обработку задачи.
* *Использование процессора* — поддержка постоянной занятости процессора.

**Интерактивным системам** важно:

* *Время отклика* — быстрая реакция на запросы.
* *Соразмерность* — выполнение пожеланий пользователя.

**Системам реального времени** требуется:

* *Окончание работы к сроку* — предотвращение потери данных.
* Предстказуемость - предотвращение деградации качества в мультимедийных системах.

Перечисленные выше требования к различным системам показывают, что можно построить большое число алгоритмов планирования, но все они окажутся обоснованными *только в ограниченных условиях*.

Для примера, рассмотрим некоторые из них.

**Планирование** в системах *пакетной обработки данных*:

* *Первым пришел — первым обслужен*. Является наиболее простым алгоримом планирования, который выделяет первому процессу, запросившему процес- сор, все время, необходимое для его завершения.
* *Кратчайшая задача — первая*. Позволяет очень быстро выполнять маленькие

задачи, но требует знания времени их выполнения.

* *Наименьшее оставшееся время выполнения*. Если имеется задача, время вы- полнения которой меньше, чем время завершения текущей, то текущая задача

останавливается, а минимальная, по времени исполнения, запускается. Здесь требуется также знать время выпонения процессов.

* *Трехуровневое планирование*. Здесь имеется *впускной планировщик*, который выбирает задачи из общей очереди и передает их процессору на выполнение.

Возможны разные варианты, учитывающие возможности процессора и уст- ройств ввода-вывода. Второй уровень планирования определяет: какие про- цессы можно хранить в памяти, а какие — на диске. Этим занимается *плани- ровщик памяти*.

**Планирование** в системах *интерактивной обработки данных* также обладает большим разнообразием. Наиболее известных два алгоритма:

* *Циклическое планирование*, когда каждому запускаемому процессу *выделя- ется квант времени*, по истечении которого или по запросу устройств ввода-

вывода процесс останавливается и помещается в конец очереди.

* *Приоритетное планирование*, когда каждому запускаемому процессу *присва- ивается приоритет* и *управление передается готовому к работе процессу, с наивысшим приоритетом*.

**Планирование** в системах *реального времени*, которые подразделяются на:

* *Жесткие системы реального времени*, требующие жестких сроков реакции на запросы каждой задачи.
* *Мягкие системы реального времени*, для которых нарушение сроков выпол- нения задач - *нежелательно, но допустимо*.

**Замечание** Как правило, все ОС используют разные и сложные алгоритмы планирования, подроб- ности которых мы изучать не будем, но широкоизвестные ОС, такие как MS Windows или Linux Desktop, относятся к системам интерактивной обработки данных.

## Синхронизация процессов

Вопросы *синхронизации процессов* описывают проблематику взаимодействия активных элементов ПО, связанных единым алгоритмом реализации их работы.

Основы такой синхронизации заложены в самой модели процесса:

* *процесс создается* на основе родительского процесса, наследуя от него прог- раммный код и все открытые ресурсы;
* *родительский процесс* отслеживает завершение дочернего процесса, тем са- мым синхронизируя иерархию процессов и разгружая ядро ОС от приклад- ных аспектов взаимодействия процессов.

**Замечание**

В ОС MS Windows не реализована полная модель процесса, поэтому синхронизация

*родитель-дочерний* в ней только имитируется, что естественным образом вносит свою специфику в разработу ПО этой ОС.

Следующей по важности проблемой является *реакция процессов на события*, которые по своей природе являются асинхронными (случайными) и не могут быть эффективно реализованы в прикладном алгоритме программы. Более того, реакция на события должна распространяться на все процессы, которые не связаны между собой «*родственными*» отношениями. «Механизмом» такой синхронизации про- цессов являются сигналы, перечень которых должен поддерживаться ядром ОС.

Модель сигнала, по многим причинам, является самой сложной «конструк- цией» ОС:

* сигналы *должны доставляться всем* запущенным процессам ОС, многие из которых находятся в состояниях «Готовность» или «Ожидание»;
* реакция процесса на сигнал *должна быть своевременной*, иначе возможна ошибочная обработка его процессом;
* реализация «механизма» сигналов *должна быть эффективной*, покольку в среде ОС возникает и обрабатывается множество сигналов;
* реакция на сигналы *должна быть стандартизирована*, чтобы она могла быть реализована в языках программирования.

**Замечание**

Несмотря на универсальное средство синхронизации процессов, сигналы являются дос- таточно сложным инструментом реализации программного обеспечения, поскольку тре- буют:

* знания особенностей их использования, в контексте работы ядра ОС;
* знания особенностей их реализации в конкретной ОС, что снижает переноси- мость прикладного ПО.

Очевидно, что рассмотренных выше средств недостаточно для «приемлемой» синхронизации многих прикладных задач, поскольку сама «природа» процесса ори- ентирована на разделение кода и данных приложений, реализуемых в виде процес- сов. В частности, невозможно определить последовательность активации ядром ОС дочернего и родительского процессов.

Дальнейшее развитие средств синхронизации процессов связано с *понятем канала*, которое было изучено в теме 7, как реализация части функций «*Подсисте- мы ввода-вывода*». Очевидно, что реализация этих каналов тесно связана и с «*Под- системой управления процессами*», подчеркивая сложность реализации функций ядра ОС.

Общий «механизм» синхронизации процессов с помощью каналов основан на *функциях блокирования операций* чтения из канала и записи в канал:

* *неименованные* или полудуплексные каналы UNIX обеспечивают синхрони- зацию только родственных процессов;
* *именованные каналы*, в которых задействована файловая система ОС, обеспе-

чивает синхронизацию всех процессов, даже еще не запущенных в системе.

Недостатки такой синхронизации:

* *неформатированный обмен данными*, что требует выделения переданного со- общения на каждой взаимодействующей стороне;
* *необходима дополнительная синхронизация*, даже при реализации схемы

*один-ко-многим*.

Принципиальное для многих задач решение синхронизации связано с *мо- делью потоков* (нитей, threads), которые обеспечивают прикладной программе об- щее адресное пространство как для кода, так и для данных. Сама синхронизация возлагается на алгоритм программы, но, в отличие от неименованных каналов, не требует организации средств передачи сообщений, а также специального формати- рования передаваемых данных.

Существенный недостаток использования каналов - *невозможность взаимо- действия произвольных процессов ОС*.

**Замечание** Ядро ОС Linux организует модель потоков в виде отдельных процессов, которые раз- деляют общие сегменты кода и данных, но имеют разные сегменты стека.

Общим недостатком всех базовых средств взаимодействия процессов являет- ся *отсутствие полных гарантий* на заданную последовательность выполняемых операций в системе. Причина состоит в том, что все указанные средства синхрони- зации реализуются ядром ОС на фоне действий планировщика процессов. Как следствие, невозможно определить какой из процессов первым захватит нужный многим процессам ресурс или изменит какие-либо данные.

Устранение этих общих недостатков обеспечивается дополнительными сред- ствами синхронизации, которые будут рассмотрены в последующих темах данной дисциплины.

## Стандарты POSIX

Мы знаем, что процессы являются *основными функциональными элементами* операционных систем (ОС). В стандарте POSIX-2001, формальное определение процесса дается, *через определение его атрибутов*.

Общий список таких атрибутов приведен в таблице 1.1. Следует хорошо зау- чить эти определения, поскольку далее, мы будем использовать их по-умолчанию.

*Таблица 1.1 - Атрибуты, уточняющие понятие процесса*

|  |  |
| --- | --- |
| ***Понятие*** | ***Определение*** |
| Процесс | Адресное пространство вместе с выполняемыми в нем потоками управления, а также системными ресурсами, которые этим пото- кам требуются. |
| Идентификатор процесса | Положительное целое число, которое однозначно идентифици- рует процесс в течение времени его жизни. |
| Время жизни процесса | Период времени от его создания до возврата идентификатора операционной системе. |
| Активный процесс | Процесс, созданный с помощью функции *fork(...)* до его завершения и имеющий, по крайней мере, один поток управления и собственное адресное пространство. |
| Зомби-процесс | Завершившийся процесс, подлежащий ликвидации после того, как код его завершения будет передан ожидающему этого друго- му процессу. |
| Родительский процесс | Процесс, создавший данный процесс. |
| Группа процессов | Совокупность процессов, допускающая согласованную доставку сигналов. У каждой группы имеется уникальный положитель- ный целочисленный идентификатор, представляющий ее в тече- ние времени ее жизни. В такой роли выступает идентификатор процесса, именуемого *лидером группы*. |
| Время жизни группы процессов | Период от создания группы до момента, когда ее покидает пос- ледний процесс (по причине завершения или смены группы). |
| Задание | Набор процессов, составляющих конвейер, а также порожден- ных ими процессов, входящих в одну группу. |
| Управление заданиями | Предоставленные пользователям средства выборочно приоста- навливать и затем продолжать (возобновлять) выполнение про- цессов. На отдельные задания ссылаются с помощью идентифи- каторов. |
| Сеанс (сессия) | Множество групп процессов, сформированное для целей управ- ления заданиями. Каждая группа принадлежит некоторому *сеансу*; считается, что все процессы группы принадлежат тому же сеансу. Вновь созданный процесс присоединяется к сеансу своего создателя; в дальнейшем принадлежность сеансу может быть изменена. |
| Время жизни | Период от создания сеанса до истечения времени жизни всех |

|  |  |
| --- | --- |
| сеанса | групп процессов, принадлежавших сеансу. |
| Лидер сеанса | Процесс, создавший данный сеанс. |
| Управляющий терминал | Терминал, ассоциированный с сеансом. У сеанса может быть не более одного управляющего терминала, а тот, в свою очередь, ассоциируется ровно с одним сеансом. Некоторые последова- тельности символов, вводимые с управляющего терминала, вызывают посылку сигналов всем процессам группы, ассоции- рованной с данным управляющим терминалом. |
| Управляющий процесс | Лидер сеанса, установивший соединение с управляющим терми- налом. Если в дальнейшем терминал перестанет быть управляю- щим для сеанса, лидер сеанса утратит статус управляющего процесса. |
| Приоритетные (переднего плана) процессы | Имеют некоторые привилегии при доступе к управляющему терминалу. В сеансе, установившем соединение с неким управ- ляющим терминалом, может быть не более одной группы про- цессов, приоритетной по отношению к данному управляющему терминалу. |
| Фоновые процессы | Неприоритетные к данному управляющему терминалу процес- сы. |
| Реальный идентификатор пользователя процесса | Идентификатор пользователя, создавшего процесс. |
| Реальный идентификатор группы процессов | Идентификатор группы пользователя, создавшего процесс. |

## Системные вызовы ОС по управлению процессами

Управление процессами имеет достаточно развитый набор системных сред- ств для управления ими. Мы рассмотрим только наиболее важные из них, которые к тому же соответствуют стандартам POSIX. В таблице 1.2 приведен перечень таких системных вызовов, которые разделены на три группы управления:

* атрибутами процессов;
* порождения и завершения процессов;
* потоки и сигналы процессов.

*Таблица 1.2 — Перечень системных вызовов ОС по управлению процессами*

|  |  |
| --- | --- |
|  | ***Функции управления атрибутами процессов*** |
| *getpid* | Возвращает идентификатор ID *текущего* процесса. |
| *getppid* | Возвращает идентификатор ID *родительского* процесса. |
| *getpgrp* | Возвращает идентификатор ID *группы* процессов, к которой принад- лежит текущий процесс. |
| *getsid* | Возвращает идентификатор (ID) *сессии*, для процесса заданного его идентификатором ID. |
| *setpgid* | Присваивает идентификатор группы процессов тому процессу, кото- рый задан своим номером ID. |
| *setsid* | Создает новый сеанс для текущего процесса, в котором этот процесс становится ведущим группы. |
|  | ***Функции порождения и завершения процессов*** |
| *fork* | Создание нового процесса. |
| *wait* | Ожидание завершения дочерних процессов. |
| *waitpid* | Ожидание завершения дочернего процесса, по заданному ID. |
| *exec\** | Набор системных вызовов для смены «тела» текущего процесса. |
| *exit* | Нормальное завершение процесса с заданным статусом. |
| *atexit* | Регистрация функции, которая будет вызвана при нормальном завер- шении процесса (при вызове функции *exit(...)*). |
|  | ***Функции потоков и сигналов процессов*** |
| *pthread\_\** | Набор системных вызовов для организации и управления потоками. |
| *signal* | Установка нового обработчика сигнала текущего процесса. |
| *kill* | Посылка сигнала процессу по его идентификатору ID. |
| *raise* | Посылает сигнал текущему процессу. |
| *pause* | Остановка процесса до получения сигнала. |
| *sleep* | Остановка текущего процесса на заданное число секунд, либо до по- лучения сигнала, который процесс не может игнорировать. |

В данном подразделе будут рассмотрены системные вызовы, относящиеся к первой группе: *управление атрибутами процессов*.

Как было изучено ранее, каждый процесс ОС имеет свой уникальный цело- численный идентификатор ID (*PID*).

Такие же идентификаторы емеются и у групп процессов — *GID*.

Чтобы из программы, написанной на языке ***С***, определить нужные идентифи- каторы атрибутов процессов, предусмотрены три системные функции *getpid(...)*, *getppid(...)*, *getpgrp(...)*,

Семантика этих функций следующая:

* *getpid(...)* - возвращает PID текущего процесса;
* *getppid(...)* - возвращает PID родительского процесса;
* *getpgrp(...)* - возвращает GID текущего процесса;
* *getsid(...)* - возвращает идентификатор сессии (SID) для процесса, заданного идентификатором PID.

По стандарту POSIX, первые три функции *всегда завершаются успешно*, поэтому ошибочных кодов возврата не предусмотрено.

Функция *getsid(...)*, в случае ошибки, возвращает значение *-1*, а значение переменной *errno* будет содержать код ошибки

Последние два системных вызова предназначены *для установки идентифи- каторов групп*.

**Системный вызов** *setpgid(...)* предназначен для установки идентификатора

группы процессов, например, в целях управления заданиями. Его выполнение вле- чет либо присоединение к существующей группе процессов, либо создание новой группы в рамках сеанса, в которую входит вызывающий процесс.

При удачном выполнении *setpgid(...)* возвращает значение равное *нулю*. При ошибке возвращается *-1*, а переменной *errno* присваивается номер ошибки:

**Замечание**

Процесс может установить идентификатор группы *только для себя или порожденного*

процесса. *Нельзя изменить* идентификатор группы процессов *лидера сеанса*.

**Системный вызов** *setsid(...)* служит для создания *нового сеанса (сессии)* и *установки идентификатора группы сессии* текущему процессу. В результате, вызывающий процесс становится:

* ведущим процессом *в группе сессии*;
* ведущим процессом *нового сеанса (сессии)*;
* и не имеет *контролирующего терминала*.

Функция возвращает *идентификатор сеанса* вызывающего процесса. При ошибке

возвращается *-1*, а переменной *errno* присваивается номер ошибки *EPERM*.

**Замечание**

Идентификаторы группы процессов и сеанса равными идентификатору вызывающего (текущего) процесса. Вызывающий процесс будет единственным в этой группе и сеансе.

## Системный вызов fork() и каналы процесса

В данном подразделе, мы рассмотрим группу системных вызовов, связанных с *порождением* и *завершением процессов*.

Прежде всего, взаимодействие процессов на этом уровне определяется функциями:

* *fork(...)* - порождает новый дочерний процесс;
* *wait(...)* - ожидает завершение любого дочернего процесса, регистрируя их в системе и завершая сам процесс удаления из системы;
* *waitpid(...)* - ожидает завершение процесса с конкретным номером PID.

**Новый (***порожденный, дочерний***) процесс** является точной копией (*роди- тельского*) процесса, вызвавшего *fork(...)*, за исключением следующих моментов:

* у порожденного процесса *свой идентификатор*, отличный от идентификато- ра родительского процесса;
* у порожденного процесса *собственная копия файловых дескрипторов*, ссыла- ющихся на те же описания открытых файлов, что и соответствующие деск- рипторы родительского процесса;
* порожденный процесс *не наследует блокировки файлов*, установленные ро- дительским процессом.
* порожденный процесс *создается с одним потоком управления* – копией того, что вызвал *fork(...)*;
* имеются также некоторые *тонкости, связанные с обработкой сигналов*, на которых мы здесь останавливаться не будем.

В случае успешного завершения функция *fork(...)* возвращает порожденному процессу *0*, а родительскому процессу – *идентификатор порожденного процесса*. После этого оба процесса начинают независимо выполнять инструкции, располо- женные за обращением к функции *fork(...)*.

При неудаче, родительскому процессу возвращается *-1*, *новый процесс не создается*, а значение *errno* устанавливается код ошибки

**Замечание**

В ОС Linux родительский и дочерний процессы разделяют сегмент кода, а сегменты дан- ных и стека у них — различны.

**Родительский процесс** реализует ожидание завершения процессов-потомков и получает информацию о их статусе завершения с помощью функций семейства *wait(...)*.

Функция *wait(...)* приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс не завершится, или до появления сигнала, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик. Если дочерний процесс к моменту вызова функции уже завершился (*так называемый процесс "зомби" ("zombie")*), то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются.

Функция *wait(...)* возвращает *идентификатор дочернего процесса*, который завершил выполнение, или *-1* в случае ошибки и переменной *errno* присваивается номер ошибки.

В частности, -1 возвращается, когда дочерних процессов *еще небыло* или они

*все уже обработаны*.

**Замечание**

Целочисленная переменная *status* содержит в себе статус завершения дочернего про- цесса, которое установленно им с помощью системного вызова *exit(...)* или оператора *return*, в функции ***main(...)***. Чтобы правильно извлечь это значение, следует воспользо- ваться двумя макросами:

* *WIFEXITED***(***status***)** - не равно нулю, если дочерний процесс успешно завершился.
* *WEXITSTATUS***(***status***)** - возвращает восемь младших битов значения, которое вер- нул завершившийся дочерний процесс; этот макрос можно использовать только,

если макрос *WIFEXITED* вернул ненулевое значение.

**Замечание** Системный вызов *wait(...)* может возвращать значение *-1* и в других случаях, например, при обработке сигналов, что будет рассмотрено далее.

Системный вызов *waitpid()* эквивалентен вызову *wait(...)*, когда аргумент *pid*

равен *(pid\_t) (-1)*, а аргумент *options* имеет *нулевое значение*.

В общем случае, аргумент *pid* может принимать несколько значений:

* *< -1* - означает, что нужно ждать любого дочернего процесса, идентификатор группы процессов которого равен абсолютному значению *pid;*
* *-1* - означает ожидание любого дочернего процесса; функция *wait(...)* ведет себя точно так же;
* *0* - означает ожидание любого дочернего процесса, идентификатор группы процессов которого равен идентификатору текущего процесса;
* *> 0* - означает ожидание дочернего процесса, чей идентификатор равен *pid*.

Значение *options* создается путем логического сложения нескольких следую- щих констант, определенных в заголовочном файле *<sys/wait.h>*:

* *WNOHANG* - означает немедленное возвращение управления, если ни один дочерний процесс не завершил выполнение;
* *WUNTRACED* - означает возврат управления и для остановленных (но не отслеживаемых) дочерних процессов, о статусе которых еще не было сооб-

щено. Статус для отслеживаемых остановленных подпроцессов также обес- печивается без этой опции.

**Замечание**

При анализе переменной *status*, кроме рассмотренных выше двух макросов, можно вос- пользоваться макросами:

* *WIFSIGNALED***(***status***)** - возвращает истинное значение, если дочерний процесс завершился из-за необработанного сигнала.
* *WTERMSIG***(***status***)** - возвращает номер сигнала, который привел к завершению дочернего процесса; этот макрос можно использовать, только если *WIFSIGNALED* вернул ненулевое значение;
* *WIFSTOPPED***(***status***)** - возвращает истинное значение, если дочерний процесс, из- за которого функция вернула управление, в настоящий момент остановлен; это возможно, только если использовался флаг *WUNTRACED* или когда подпроцесс

отслеживается;

* *WSTOPSIG***(***status***)** - возвращает номер сигнала, из-за которого дочерний процесс был остановлен; этот макрос можно использовать, только если *WIFSTOPPED* вернул ненулевое значение.

Теперь рассмотрим системные вызовы, которые непосредственно связаны с самим завершением процесса. Синтаксис этих вызовов имеет вид:

#include <stdlib.h>

void exit (int status); void \_Exit (int status);

#include <unistd.h>

void \_exit (int status);

Все эти вызовы немедленно завершают процесс и посылают родительскому процессу значение аргумента *status*, в качестве которого могут служить константы *0*, *EXIT\_SUCCESS*, *EXIT\_FAILURE* или любое другое значение, но ожидающему родительскому процессу будет доступено только значение (*status & 0377*). Также родительскому процессу посылается сигнал *SIGCHLD*.

**Замечание**

Сами эти вызовы переводят процесс в состояние «*зомби*», пока родительский процесс не выполнит функции *wait(...)* или *waitpid(...)*.

Если процесс является *лидером сеанса* и его управляющий терминал является управ- ляющим терминалом сеанса, то каждому процессу в группе процессов этого управляю- щего терминала посылается сигнал *SIGHUP*, и терминал отсоединяется от этого сеанса, что позволяет захватить его новому управляющему процессу.

Если завершение процесса *приводит группу процессов к потере родителя*, и если лю- бой член такой группы приостанавливается, то каждому процессу группы посылается сигнал *SIGHUP*, за которым следует сигнал *SIGCONT*.

Завершающийся процесс может использовать функцию *atexit(...) для регист- рации других функций*, которые будут вызываться, если процесс завершается, обра- щаясь к *exit(...)* или возвращаясь из ***main(...)***.

Синтаксис вызова:

#include <stdlib.h>

int atexit (void (\*func) (void));

Функция *atexit(...)* возвращает значение *0*, если действие выполнено, иначе возвращается *ненулевое значение*, а значение переменной *errno* не меняется.

**Замечание**

Реализация ОС должна поддерживать регистрацию *по крайней мере тридцати двух функций* и вызывать их в обратном порядке.

### Пример использования каналов процессов

Чтобы продемонстрировать *прикладные аспекты* взаимодействия процессов, рассмотрим задачу «*Взаимодействие процессов через неименованные каналы*»:

* обычно, многие приложения читают входные данные с устройства 0 и выво- дят информацию на устройво 1;
* если общий алгоритм приложения организован так, что каждый процесс вы- полняет только часть обработки данных и передает результат другому про- цессу, то можно организовать взаимодействие процессов через неименован-

ные полудуплексные каналы UNIX; при этом, на ***N*** взаимодействующих процессов потребуется ***N-1*** канал.

Такое взаимодействие называется *конвейером процессов* и может рассматриваться как типовое решение *метода последовательной обработки данных*.

В качестве примера, рассмотрим программу, в которой:

* *родительский процесс* создает ***N-1*** каналов, а затем запускает ***N*** дочерних процессов и ожидает их завершения; после завершения работы всех процес- сов, все открытые каналы закрываются и работа родителького процесса — за- вершается;
* *первый дочерний процесс* читает строку данных с клавиатуры и по первому каналу передает ее второму процессу;
* *последний дочерний процесс* (*N*) читает данные из канала *N-1* и печатает результат на терминал;
* *все остальные дочерние процессы* (*i*): читают из *i-канала* и пишут в (*i+1*)-

*канал*.

Дополнительно:

* с целю унификации, *все дочерние процессы*: читают из устройства *0* и пишут на устройство *1*;
* последний дочерний процесс регистрирует *функцию обработки завершения*.

**Замечание**

Полудуплексные каналы UNIX были уже изучены в теме 7 и рассматривались как сред- ство подсистемы управления файлами. Вспомним, что:

* каналы создаются с помощью сисемного вызова *pipe(...)*, который возвращает целочисленный массив дескрипторов *fd[2]*;
* дескриптор *fd[0]* — предназначен для чтения из канала;
* дескриптор *fd[1]* — предназначен для записи в канал.

**Замечание** Поскольку родительский и дочерний процессы запускаются независимо друг от друга (*асинхронно*), то родителький процесс должен дать поработать дочернему. Делается это с помощью системных вызовов *wait(...)* или *sleep(...)*.

На рисунке 1.4 приведен пример передачи сообщений через последователь- ность каналов. Обратите внимание, что для передачи сообщений *не обязательно выделять буфер в памяти ОС*, поскольку таким буфером является сам канал.

### Имитация конвейеров языка shell

Любая программа, написанная на языке ***С***, должна иметь *особую точку входа*

для запуска ее в среде ОС.

Такую точку входа обеспечивает функция *main(...)*,

**Замечание**

В соответствии с принятыми соглашениями, значение *argc* не меньше единицы, а *пер-*

*вый элемент* массива *argv* указывает на цепочку символов, содержащую имя выпол- няемого файла. Дополнительно, разрешается использовать и другие формы опреде- ления функции ***main(...)***:

Поскольку все процессы ОС, кроме первого, создаются посредством систем- ного вызова *fork(...)*, то для запуска программ имеется набор системных функций семейства *exec(...)*:

#include <unistd.h> extern char \*\*environ;

int execl (const char \*path, const char \*arg0, ..., (char \*) 0); int execv (const char \*path, char \*const argv []);

int execle (const char \*path, const char \*arg0, ..., (char \*)0, char \*const envp []);

int execve (const char \*path, char \*const argv [], char \*const envp []);

int execlp (const char \*file, const char \*arg0, ..., (char \*) 0); int execvp (const char \*file, char \*const argv []);

где внешняя переменная *environ* - указатель на массив указателей, которые адресуют переменные среды ОС, в виде: **Имя=Значение**

*path* — указатель на маршрутное имя файла, с новым образом процесса;

*file* — указатель на имя файла загружаемой программы;

*arg0, ...*, - указатели на соответствующие аргументы, при вызове нового образа процесса; причем, последним в списке располагается пустой указатель, а аргумент *arg0* должен указывать на имя файла-образа;

*envp* — то же, что и внешняя переменная *environ*.

**Замечание**

Все массивы *argv*, *envp* и *environ* должны завершаться *пустым указателем* ***NULL***.

Во многих системах разрабоки ОС Linux, как и в ОС УПК АСУ, вместо переменной

*environ* используется переменная *environ*.

Все функции семейства *exec(...)*:

* *заменяют* текущий образ процесса новым;
* *случае успешного завершения*, возврат в вызывающий процесс *невозможен*; новый образ создается на основе выполнимого файла, который называется

*файлом образа процесса*;

* *в случае ошибки*, возвращаемым значением будет *-1* и глобальной перемен- ной *errno* будет присвоен код: (см. ***man execve***).

**Замечание**

Файловые дескрипторы остаются открытыми в новом образе, если только они не были снабжены флагом *FD\_CLOEXEC*.

Действующий идентификатор пользователя процесса *переустанавливается* равным идентификатору владельца файла (аналогично для группы).

Следующие атрибуты процесса *остаются неизменными*:

* идентификатор процесса;
* идентификатор родительского процесса;
* идентификатор группы процессов;
* членство в сеансе;
* реальные идентификаторы пользователя и группы процесса;
* идентификаторы дополнительных групп;
* текущий и корневой каталоги;
* маска режима создания файлов;
* атрибуты, связанные с обработкой сигналов.

Для демострации возможностей системных вызовов по смене образов про- цессов, рассмотрим задачу «*Имитация конвейеров языка shell*»:

* *родителский процесс* запрашивает ввод строки, содержащей не менее двух команд, например, **ls -l /home/upk | grep d**, а затем, - проводит форми-

рование массивов указателей *для аргументов вызова образов* дочерних про- цессов;

* *далее*, родительский процесс создает массив каналов, число которых на единицу меньше числа заявленных команд, и, *в цикле по числу команд*,

запускает дочерние процессы, отслеживая их завершение;

* *каждый дочерний процесс*, в зависимости от своего номера, *связывает свои стандартные ввод и вывод с номером канала*, - как уже было рассмотрено в программе на листинге 1.4;
* *далее*, дочерний процесс *загружает новый образ*, используя в качестве аргу- ментов соотвествующий массив указателей, подготовленный для него роди- тельским процессом, или аварийно завершается, в случае ошибки.

## Нити (Threads)

Взаимодействие процессов посредством каналов, является мощным инстру- ментальным средством ОС, но имеет ряд существенных недостатков:

* *необходимость синхронизации* операций чтения и записи в каналы для раз- ных *асинхнонно выполняющихся* процессов;
* *необходимость форматирования* передаваемых в канал данных;
* *необходимость дешифровки* читаемых из канала данных.

Во многих современных ОС существует *расширенная реализация* понятия процесс,

когда *исполняемый образ программы* представляет собой не только совокупность выделенных ему ресурсов данных и каналов взаимодействия, но и *набор нитей исполнения*.

В данном подразделе, применительно к стандарту *POSIX*, будут рассмотрены

*наборы нитей исполнения*, еще известные как ***потоки процессов*** или ***threads***.

Каждый, запущенный ОС, процесс имеет *одну нить* исполнения, которая на-

зывается *главной* или *начальной нитью*. *Дополнительные нити* (*threads*) создаются программистом из главной нити посредством набора системных вызовов.

Все дополнительные нити процесса *разделяют*:

* его программный код;
* глобальные переменные;
* системные ресурсы.

Но каждая дополнительная нить *имеет*:

* собственный программный счетчик;
* свое содержимое регистров;
* свой стек.

Поскольку, все глобальные переменные у нитей исполнения являются общими, они

могут использоваться как общие элементы данных, поэтому нет необходимости прибегать к «механизму» каналов, описанных выше.

**Замечание**

В различных версиях ОС UNIX существуют различные *интерфейсы*, обеспечивающие работу с нитями исполнения. Нити исполнения, удовлетворяющие стандарту POSIX, при- нято называть *POSIX threads* или кратко, - *pthreads*.

Главная проблема реализации нитей — *каким образом реализовать механизм их параллельного исполнения*.

В ОС Linux, создание новой нити обеспечивается с помощью системного вызова *clone(...)*, который мы рассматривать не будем, но который обеспечивает разделение с родительским процессом:

* его ресурсов;
* программного кода;
* данных, расположенных вне стека. Такая нить процесса имеет:
* свои *управляющие структуры*, включая новый идентификатор нити;
* свой *собственный стек* для вызова функций.

Таким образом, созданный новый поток (*нить*) имеет тот же идентификатор процесса, что и главная нить.

Каждая нить исполнения, как и процесс, имеет в системе *уникальный номер* – *иден- тификатор thread'a*. Поскольку традиционный процесс, в концепции нитей, исполнения трактуется как процесс, содержащий единственную нить исполнения, мы можем узнать идентификатор конкретной нити только в процессе ее исполнения. Для этого исполь- зуется функция *pthread\_self()*.

**Замечание**

Для работы с нитями разработано большое количесво функций. Далее, мы рассмотрим только функции, которые реализуются следующими *системными вы- зовами ОС*:

* *pthread\_create()* - создание нити;
* *pthread\_exit()* - завершение работы нити;
* *pthread\_join()* - другой вариант завершения работы нити;
* *pthread\_self()* - получение идентификатора нити.

**Замечание**

Далее, говоря о создании или удалении нити, мы будем иметь ввиду *дополнительную*

*нить*, поскольку главная нить в процессе существует всегда.

Новая (дополнительная) нить создается вызовом *pthread\_create(...)*, который имеет синтаксис:

#include <pthread.h>

*int* pthread\_create(*pthread\_t* \*thread,

*pthread\_attr\_t \**attr,

*void \** (*\*start\_routine*)(*void \**), *void \**arg);

где *thread* - указатель на создаваемый идентификатор новой дополнительной нити; *attr* - указатель на атрибуты создаваемой нити, значение которого *NULL* предпола- гает создание нити с атрибутами по умолчанию;

*start\_routine* — указатель на функцию, которая вызывается при создании нити и работа которой ассоциируется с алгоритмом работы самой нити;

*arg* — указатель на аргумент, передаваемый в функцию *start\_routine*.

В случае успешного завершения, возвращается значение *0*, иначе положи- тельное значение, соответствующее:

* *EAGAIN* — в системе отсутствуют ресурсы, необходимые для создания нити, или процесс превысил лимиты количества создаваемых нитей, заданный

системным значением *PTHREAD\_THREADS\_MAX*;

* *EINVAL* — ошибочное значение аргумента *attr*;
* *EPERM* — ошибка доступа к средствам создаваемой нити.

Функционирование новой (дополнительной) нити ассоциируется с алгорит- мом работы вызываемой функциии *start\_routine(...)*, прототип которой имеет вид:

*void \**start\_routine(*void \** arg) { ... };

где *arg* — указатель на статически расположенный аргумент, передаваемый этой функции или значение *NULL*.

Работа нити завершется выходом из этой функции, что может осуществлять- ся тремя способами:

* с помощью выполнения функции *pthread\_exit(...)*;
* с помощью оператора *return*, который указывает статический адрес, возвра- щаемый функцией, или значение *NULL*;
* в процессе общего возврата из функции *main(...)*.

**Замечание**

Возврат *start\_routine(...)* аналогичен завершению процесса с помощью системной функ-

ции *exit(...)*. Он никогда не приводит к возврату в главную нить процесса. Кроме того, результаты завершения ее могут быть изучены из других, еще работающих, нитей про- цесса.

Для нормального завершения работы нити предусмотрен системный вызов

*pthread\_exit(...)*, который во многом похож на системный вызов *exit(...)*:

#include <pthread.h>

*void* pthread\_exit(*void \**status);

где *status* — указатель на статический адрес данных, которые могут быть изучены другими нитями.

Чтобы дождаться завершения работы конкретной нити и узнать адрес возвра- щенных ей данных, используется системный вызов:

#include <pthread.h>

*int* pthread\_join (*pthread\_t* thread, *void \*\**status\_addr);

где *thread* — идентификатор ожидаемой завершения нити;

*status\_addr* — указатель на указатель адреса, возвращаемых нитью данных; если нас не интересует, что вернула нить исполнения, то в качестве этого параметра можно использовать значение *NULL*.

Функция возвращает значение *0*, при успешном завершении, иначе - положи- тельное значение:

* *EDEADLK* — выполнение заблокировано, например, когда предлагается ожидать собственного завершения;
* *EINVAL* — указанный идентификатор ссылается на отключенную нить;
* *ESRCH* — отсутствует нить с заданным идентификатором.

Любая нить может узнать свой идентификатор с помощью вызова:

#include <pthread.h>

*pthread\_t* pthread\_self(*void*);

**Замечание**

Тип данных *pthread\_t* является синонимом для одного из целочисленных типов языка ***C***.

Для демонстрации работы нитей, рассмотрим пример программы, показан- ной на листинге 1.6.

Алгоритм работы данной программы следующий:

* *из главной нити* процесса создаются две дополнительные нити, которые вы-

полняются на базе функции *mythread(void \*arg)*, которой, в качестве аргумен- та *arg*, передается номер создаваемой нити;

* *после создания* дополнительных нитей, главная нить переходит в состояние ожидания их завершения и печатает на терминал сообщение об этом;
* *каждая созданная нить* выполняет по 10 циклов, «засыпая» на случайный промежуток времени, и изменяет разделяемый массив *a[3]* следующим обра- зом:
  + *a[0]* - сумманное число запусков нитей №1 и №2;
  + *a[1]* - количество циклов, выполненных нитью №1;
  + *a[2]* - количество циклов, выполненных нитью №2;
* *выполнив работу*, каждая дополнительная нить печатает сообщение о своем завершении и осуществляет выход оператором *return NULL*.

## Сигналы POSIX

Формально, стандарт POSIX-2001 под *сигналом* понимает «*механизм*», с помощью которого процесс или поток управления уведомляют о некотором собы- тии, произошедшем в системе, или подвергают воздействию этого события.

Примерами подобных событий могут служить аппаратные исключительные ситуации и специфические действия процессов.

Термин "***сигнал***" используется также *для обозначения самого события*.

**Говорят**, что *сигнал генерируется* (или *посылается*) для процесса (*потока управления*), когда происходит вызвавшее его событие:

* выявлен аппаратный сбой;
* отработал таймер;
* пользователь ввел с терминала специфическую последовательность симво- лов;
* процесс обратился к функции *kill(...)* и другие.

Иногда, по одному событию генерируются сигналы *для нескольких процес- сов*, например, для группы процессов, ассоциированных с некоторым *управля- ющим терминалом*.

В момент генерации сигнала определяется, посылается ли он процессу или конкретному потоку управления в процессе. Сигналы, сгенерированные в резуль- тате действий, приписываемых отдельному потоку управления, таких как возник- новение аппаратной исключительной ситуации, посылаются этому потоку.

Сигналы, генерация которых ассоциирована с идентификатором процесса или группы процессов, а также с асинхронным событием, например, пользова- тельский ввод с терминала, посылаются процессу. В каждом процессе определены действия, предпринимаемые в ответ на все предусмотренные системой сигналы.

**Говорят**, что *сигнал доставлен процессу*, когда взято для выполнения дейст- вие, соответствующее данным процессу и сигналу.

В интервале, *от генерации до доставки* или принятия сигнал называется *ждущим*. Обычно, он невидим для приложений, однако доставку сигнала потоку управления можно блокировать. Если действие, ассоциированное с заблокирован- ным сигналом, отлично от игнорирования, он будет ждать разблокирования.

У каждого потока управления есть *маска сигналов*, определяющая набор блокируемых сигналов. Обычно она достается в наследство от родительского потока.

В данном подразделе мы завершаем рассмотрение базовых системных вызо- вов управления процессами, объединенными в группу «*Функции потоков и сигна- лов процессов*», где будут изучены следующие системные вызовы:

* *signal()* - установка нового обработчика сигнала текущего процесса;
* *kill()* - посылка сигнала процессу по его иденитфикатору PID;
* *raise()* - посылка сигнала текущему процессу;
* *pause()* - остановка процесса до получения сигнала;
* *sleep()* - остановка процесса на заданное число секунд, либо до получения сигнала.

Установка реакции процесса на получаемые сигналы осуществляется с по- мошью системного вызова:

#include <signal.h>

typedef void (\*sighandler\_t)(int);

sighandler\_t signal(int *signum*, sighandler\_t *handler*);

где *signum* — номер обрабатываемого сисгнала;

*handler* — ассоциируется с действем одного из трех типов:

* *SIG\_DFL* — выполняются подразумеваемые действия, зависящие от сигнала,

которые описаны в заголовочном файле *<signal.h>*;

* *SIG\_IGN* - игнорировать сигнал; доставка сигнала не оказывает воздействия на процесс;
* *указатель на функцию* - обработать сигнал, выполнив при его доставке задан- ную функцию; после завершения функции обработки процесс возобновляет выполнение с точки прерывания.

Обычно, функция обработки вызывается в соответствии со следующим заголовком языка C:

void func (int signo);

где ***signo*** - номер доставленного сигнала.

**Замечание**

Первоначально, до входа в функцию ***main(...)***, реакция на все сигналы установлена как

*SIG\_DFL* или *SIG\_IGN*.

Функция обработки называется *асинхронно-сигнально-безопасной* (АСБ), если ее можно вызывать без каких-либо ограничений при обработке сигналов.

В стандарте POSIX-2001, имеется список функций, которые должны быть либо *повтор- но входимыми*, либо непрерываемыми сигналами, что превращает их в АСБ-функции. В этот список включены 117 функций.

Сигнал процессу может быть послан либо из командной строки с помощью служебной программы ***kill***, либо из процесса с помощью системных вызовов:

#include <signal.h>

int kill (pid\_t pid, int sig); int raise(int *sig*);

где *pid* — может принимать следующие значения:

* *pid > 0* - сигнал *sig* посылается процессу с идентификатором *pid;*
* *pid = 0* - *sig* посылается каждому процессу, который входит в группу текущего процесса;
* *pid = -1* - *sig* посылается каждому процессу, за исключением процесса с номером *1*;
* *pid < -1* - *sig* посылается каждому процессу, который входит в группу процесса с идентификатором -*pid*.

*sig* — номер обрабатываемого сигнала.

**Замечание**

Если *sig=0*, то действия сводятся к проверке допустимости значения *pid*.

В случае успеха, возвращается *ноль*. При ошибке, возвращается *-1* и значение

*errno* устанавливается:

* *EINVAL* -задан неправильный сигнал;
* *ESRCH* - идентификатор процесса *pid* или группа процесса не существуют. Заме-

тим, что существующий процесс может быть *зомби - процессом*, который уже находится в состоянии завершения, но пока в котором пока ещё не выполнился *wait(...)*.

* *EPERM* - текущий процесс не имеет прав на посылку сигнала к любому из ука- занных процессов; процессы, которые имеют права на посылку сигнала процессу

с номером *pid* должны иметь привелегии суперпользователя или, реальный или эффективный идентификатор пользователя процесса, посылающего сигнал, должен быть таким же как реальный или эффективный идентификатор пользова- теля процесса, принимающего сигнал; в случае, когда посылающий и принимаю- щий процессы относятся к одной сессии, становится доступным сигнал *SIGCONT*.

**Замечание**

Процесс имеет право послать сигнал адресату, заданному аргументом *pid*, если он (про- цесс) имеет соответствующие привилегии или его реальный или действующий иденти- фикатор пользователя совпадает с реальным или сохраненным идентификатором адресата.

Вызов служебной программы ***kill*** в виде: ***kill -l***, позволяет вывести на терми- нал весь список сигналов, поддерживаемых используемой ОС:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1) SIGHUP | | 2) | SIGINT | 3) | SIGQUIT | 4) | SIGILL |
| 5) SIGTRAP | | 6) | SIGABRT | 7) | SIGBUS | 8) | SIGFPE |
| 9) SIGKILL | | 10) | SIGUSR1 | 11) | SIGSEGV | 12) | SIGUSR2 |
| 13) | SIGPIPE | 14) | SIGALRM | 15) | SIGTERM | 17) | SIGCHLD |
| 18) | SIGCONT | 19) | SIGSTOP | 20) | SIGTSTP | 21) | SIGTTIN |
| 22) | SIGTTOU | 23) | SIGURG | 24) | SIGXCPU | 25) | SIGXFSZ |
| 26) | SIGVTALRM | 27) | SIGPROF | 28) | SIGWINCH | 29) | SIGIO |
| 30) | SIGPWR | 31) | SIGSYS | 32) | SIGRTMIN | 33) | SIGRTMIN+1 |
| 34) | SIGRTMIN+2 | 35) | SIGRTMIN+3 | 36) | SIGRTMIN+4 | 37) | SIGRTMIN+5 |
| 38) | SIGRTMIN+6 | 39) | SIGRTMIN+7 | 40) | SIGRTMIN+8 | 41) | SIGRTMIN+9 |
| 42) | SIGRTMIN+10 | 43) | SIGRTMIN+11 | 44) | SIGRTMIN+12 | 45) | SIGRTMIN+13 |
| 46) | SIGRTMIN+14 | 47) | SIGRTMIN+15 | 48) | SIGRTMAX-15 | 49) | SIGRTMAX-14 |
| 50) | SIGRTMAX-13 | 51) | SIGRTMAX-12 | 52) | SIGRTMAX-11 | 53) | SIGRTMAX-10 |
| 54) | SIGRTMAX-9 | 55) | SIGRTMAX-8 | 56) | SIGRTMAX-7 | 57) | SIGRTMAX-6 |
| 58) | SIGRTMAX-5 | 59) | SIGRTMAX-4 | 60) | SIGRTMAX-3 | 61) | SIGRTMAX-2 |
| 62) | SIGRTMAX-1 | 63) | SIGRTMAX |  |  |  |  |

**Стандарт языка C** определяет имена всего шести сигналов: *SIGABRT*, *SIGFPE, SIGILL, SIGINT, SIGSEGV и SIGTERM*.

ОС Linux поддерживает *сигналы режима реального времени*, включенные в POSIX 1003.1-2001, которые соответствуют 32 сигналам, начиная с номера 32 (*SIGRTMIN*) до номера 63 (*SIGRTMAX*).

В отличие от стандартных сигналов, *сигналы режима реального времени* не имеют предопределенных заранее значений: весь набор сигналов режима реального времени приложения могут использовать так, как им будет нужно.

**Замечание**

Программы должны всегда ссылаться на сигналы режима реального времени, используя записи **SIGRTMIN**+n, так как диапазон номеров таких сигналов варьируется в системах Unices.

**Стандарт POSIX-2001** определяет как обязательные для реализации сигна- лы, представленные в таблице 1.3.

*Таблица 1.3 - Сигналы, определенные стандартом POSIX*

|  |  |
| --- | --- |
| ***Сигнал*** | ***Описание сигнала*** |
| *SIGABRT* | Сигнал аварийного завершения процесса. Подразумеваемая реакция предусматривает, помимо аварийного завершения, создание файла с образом памяти процесса. |
| *SIGALRM* | Срабатывание будильника. Подразумеваемая реакция - аварийное завершение процесса. |
| *SIGBUS* | Ошибка системной шины как следствие обращения к неопределенной области памяти. Подразумеваемая реакция - аварийное завершение и создание файла с образом памяти процесса. |
| *SIGCHLD* | Завершение, остановка или продолжение порожденного процесса. Подразумеваемая реакция - игнорирование. |
| *SIGCONT* | Продолжение процесса, если он был остановлен. Подразумеваемая реакция - продолжение выполнения или игнорирование, если процесс не был остановлен. |
| *SIGFPE* | Некорректная арифметическая операция. Подразумеваемая реакция - аварийное завершение и создание файла с образом памяти процесса. |
| *SIGHUP* | Сигнал разъединения. Подразумеваемая реакция - аварийное завершение процесса. |
| *SIGILL* | Некорректная команда. Подразумеваемая реакция - аварийное завершение и создание файла с образом памяти процесса. |
| *SIGINT* | Сигнал прерывания, поступивший с терминала. Подразумеваемая реакция - аварийное завершение процесса. |
| *SIGKILL* | Уничтожение процесса (этот сигнал нельзя перехватить для обработки или проигнорировать). Подразумеваемая реакция - аварийное завершение процесса. |
| *SIGPIPE* | Попытка записи в канал, из которого никто не читает. Подразумеваемая реакция - аварийное завершение процесса. |
| *SIGQUIT* | Сигнал выхода, поступивший с терминала. Подразумеваемая реакция - аварийное завершение и создание файла с образом памяти процесса. |
| *SIGSEGV* | Некорректное обращение к памяти. Подразумеваемая реакция - аварийное завершение и создание файла с образом памяти процесса. |
| *SIGSTOP* | Остановка выполнения (этот сигнал нельзя перехватить для обработки или проигнорировать). Подразумеваемая реакция - |

|  |  |
| --- | --- |
|  | остановка процесса. |
| *SIGTERM* | Сигнал терминирования. Подразумеваемая реакция - аварийное завершение процесса. |
| *SIGTSTP* | Сигнал остановки, поступивший с терминала. Подразумеваемая реакция - остановка процесса. |
| *SIGTTIN* | Попытка чтения из фонового процесса. Подразумеваемая реакция  - остановка процесса. |
| *SIGTTOU* | Попытка записи из фонового процесса. Подразумеваемая реакция  - остановка процесса. |
| *SIGUSR1, SIGUSR2* | Определяемые пользователем сигналы. Подразумеваемая реакция  - аварийное завершение процесса. |
| *SIGPOLL* | Опрашиваемое событие. Подразумеваемая реакция - аварийное завершение процесса. |
| *SIGPROF* | Срабатывание таймера профилирования. Подразумеваемая реакция - аварийное завершение процесса. |
| *SIGSYS* | Некорректный системный вызов. Подразумеваемая реакция - аварийное завершение и создание файла с образом памяти процесса. |
| *SIGTRAP* | Попадание в точку трассировки/прерывания. Подразумеваемая реакция - аварийное завершение и создание файла с образом памяти процесса. |
| *SIGURG* | Высокоскоростное поступление данных в сокет. Подразумеваемая реакция - игнорирование. |
| *SIGVTALRM* | Срабатывание виртуального таймера. Подразумеваемая реакция - аварийное завершение процесса. |
| *SIGXCPU* | Исчерпан лимит процессорного времени. Подразумеваемая реакция - аварийное завершение и создание файла с образом памяти процесса. |
| *SIGXFSZ* | Превышено ограничение на размер файлов. Подразумеваемая реакция - аварийное завершение и создание файла с образом памяти процесса. |

В целом, технология использования сигналов достаточно сложна и требует знания многих аспектов работы ОС.

Мы рассмотрим простой пример, предполагающий, что дочерние процессы могут быть остановлены по тем или иным причинам, например внешним воздей- ствием. В таком случае, родительский процесс может бесконечно долго ожидать их завершения.

Если остановка дочернего процесса вызвана получением соответствующего сигнала, то родительский процесс может узнать об этом с помощью системного вы- зова *waitpid(...)*.

В приложениях так же часто используются системные вызовы, останавли- вающие выполнение процесса. К ним относятся:

#include <unistd.h>

unsigned int sleep(unsigned int seconds); int pause(void);

где *seconds* — число секунд, на которое процесс «засыпает».

Эти вызовы не посылают родительскому процессу сигналов, но сами реаги- руют на сигнал *SIGCONT*. Поэтому, когда они блокируют выполнение программы, родительский процесс может снять их действие по тайм-ауту, посылая сигнал.

## Проблемы распределения ресурсов ОС

Многие задачи взаимодействия процессов можно свести к *задачам борьбы за ресурсы ЭВМ*. В частности, таким ресурсом может быть память ЭВМ или, напри- мер, очередь процессов на печать.

В борьбе за общий разделяемый ресурс, процессы входят в *состояние состя- зания*.

**Состояние состязания** - *ситуация*, в которой два или более процессов счи- тывают или записывают данные *одновременно*, а результат зависит от того, *кто из них был первым*.

Основной способ решения этих проблем - *запрет* на одновременное исполь- зование разделяемого ресурса или *взаимное исключение*.

**Взаимное исключение** предполагает, что одновременно не могут выполнять- ся две *критические секции* разных процессов.

**Критической секцией** (*критической областью*) называется *часть кода про- цесса*, который обеспечивает доступ к разделяемым ресурсам.

Чтобы сформулировать общую проблематику использования любых разделя- емых ресурсов, были сформулированы *четыре необходимых условия*:

* два процесса *не должны находиться* в критических областях;
* в программе *не должно быть предположений* о скорости или количестве процессов;
* процесс, находящийся в критической области, *не может блокировать* другие процессы;
* невозможна ситуация, когда процесс *вечно ждет попадания* в критическую область.

Среди известных способов, реализующих *взаимное исключение*, можно выде- лить следующие:

* *запрет всех прерываний*, когда процесс вошел в критическую область;
* *переменные блокировки*, когда процесс, прежде чем войти в критическую об- ласть, использует значение переменной, которая доступна всем процессам.

Если значение переменной *равно 0,* то процесс изменяет ее значение на 1 и входит в критическую область, а если значение переменной *равно 1*, то про- цесс ждет, пока ее значение не *изменится на 0*;

* *строгое чередование*, когда процесс имеющий и использующий критическую область становится в очередь и получает номер этой очереди, в цикле про-

веряя некоторую общую переменную, значение которой указывает номер оче- реди с разрешенным входом в критическую область. Когда процесс выходит из критической области, он изменяет значение этой переменной, тем самым, разрешая другому процессу войти в критическую область.

* *алгоритм Петерсона*, который использует две функции *enter\_region(int process)* и *leave\_region(int process)*. Вызов первой, из этих функций, является

обязательным условием, при входе процесса в критическую область. Она так- же сохраняет дополнительные данные о процессах, вызвавших ее, поэтому только один процесс может войти в критическую область, а другому необ- ходимо ждать, когда выходящий из критической области процесс вызовет функцию *leave\_region(int process)*, освобождающую доступ к нужной облас- ти.

* *команда TSL* (*Test and Set Lock — проверить и заблокировать*), которая на аппаратном уровне гарантирует неделимость операций чтения и записи слова

памяти, обеспечивая однозначность отражения блокировки процессом разде- ляемого ресурса.

Не вдаваясь в детали возможных реализаций приведенных выше алгоритмов, мы рассмотрим пример *простейшей проблемы* разделения общих ресурсов.

**Проблема производителя и потребителя**, которая соотвествует взаимодей- ствию двух процессов, использующих *общий буфер данных* (*проблема ограничен- ного буфера*).

Более конкретно:

* *один* из процессов, *производитель*, помещает данные в буфер;
* *другой* процесс, *потребитель*, читает данные из буфера;
* *трудности начинаются*, когда производитель хочет поместить данные в бу- фер, а он уже заполнился; тогда производитель должен ждать, когда второй

процесс, полностью или частично, не освободит буфер;

* *аналогично*, когда потребитель хочет прочитать данные из буфера, а он пуст, процесс уходит в ожидание, пока другой процесс не положит данные в буфер и не разбудит потребителя.

**Замечание** Указанная задача очень часто встречается даже в прикладном программировании, не го- воря уже о системном, и требует *разработки примитивов*, обеспечивающих такое меж- процессное взаимодействие.

Ситуация значительно осложняется, когда процессу необходимо захватить

несколько разделяемых ресурсов. В этом случае, возникает *состояние взаимной блокировки*.

**Состояние взаимной блокировки** возникает тогда, когда процесс успел зах- ватить только часть разделяемых ресурсов, необходимых ему для выполнения рабо- ты, а другую часть захватили другие процессы. В такой ситуации, все процессы переходят в *состояние бесконечного ожидания* необходимых им ресурсов.

Примером задачи, приводящей к взаимным блокировкам, является «*Задача об обедающих философах*», которую мы подробно рассмотрим в подразделе 1.5. Здесь же отметим, что в общем случае, необходимы дополнительные системные средства, обеспечивающие процессы инструментами надежной фиксации фактов входа в критические области и выхода из них.

Одним из первых системных средств, обеспечивающих программистов на- дежными инструментами работы с взаимодействующими процессами, стал пакет, имеющий название *System V IPC*. В последующем, его идеи были положены в сред- ства сетевого взаимодействия ЭВМ.

**Замечание** Следует хорошо осознать и запомнить, что все проблемы синхронизации процессов не могут быть решены алгоритмически *только на уровне отдельного процесса*. Это свя- зано с тем, что алгоритм работы процесса прерывается ядром ОС, а захват нескольких нужных ресурсов процесса может осуществляться только некоторой последователь- ностью команд (системных вызовов), которые могут не завершиться до остановки про- цесса и передачи управления другому процессу.

## Системный пакет IPC

С целью обеспечить системных и прикладных программистов надежным инструментом, устранения проблем взаимодействующих процессов, было разрабо- тано несколько системных средств ядра ОС, известных под названием *System V IPC*.

**IPC** (*Inter Process Communication*) — ориентирован на решение трех проб-

лем:

* *надежная и простая передача данных и сообщений* от одного процесса дру- гому;
* *контроль над деятельностью* процессов в критических ситуациях;
* *согласование действий* между процессами.

Непосредственная реализация пакета разделена на *четыре части*:

* *средства адресации IPC*
* *семафоры*,
* *разделяемая память* и *очереди сообщений*

Создавая пакет IPC, разработчики сразу же предусмотрели возможность асинхронного взаимодействия процессов. Для этих целей была разработана систе- ма адресации, которой должен пользоваться каждый процесс, использующий сред- ства пакета IPC.

Необходимость такой адресации обусловлена тем, что *процессы должны раз- личать* разные виды взаимодействия, обусловленные их алгоритмами работы и прикладной направленностью самих процессов.

В качестве информационной основы формирования таких адресов исполь- зуются следующие два объекта:

* *имена файлов*, доступные в файловой системе ОС;
* *произвольное целое число*, отличное от нуля, интерпретируемое как номер проекта.

В случае удачного завершения вызова возвращается значение созданного ключа типа *key\_t*. При ошибке возвращается *-1*, а в переменную *errno* записывается код ошибки, согласно системному вызову *stat*(2).

**Замечание** Тип данных *key\_t* обычно представляет собой 32-битовое знаковое целое — возвращае- мое значение, которое — одинаково для всех имен, указывающих на один и тот же файл, при одинаковом значении аргументов *proj\_id* и *pathname*. Возвращаемые значения функ- ции должны различаться, когда одновременно существующие файлы или идентификато- ры проекта различаются.

Ядро ОС хранит информацию обо всех средствах System V IPC, используе- мых в системе вне контекста пользовательских процессов. В прикладном плане, ключ, генерируемый функией *ftok(...)*, рассматривается как *средство связи* между взаимодействующими процессами.

При создании нового средства связи или получении доступа к уже существу- ющему, процесс получает неотрицательное целое число — *дескриптор* (иденти- фикатор) *этого средства связи*, которое однозначно идентифицирует его во всей вычислительной системе. Этот дескриптор *должен передаваться в качестве пара- метра* всем системным вызовам, осуществляющим дальнейшие операции над со- ответствующими средствами System V IPC.

## Утилиты управления средствами пакета IPC

У пользователя ОС, для работы со средствами межпроцессного взаимодей- ствия *System V IPC*, имеются три основные утилиты, которыми следует пользо- ваться по мере необходимости:

* *ipcmk* — создание различных ресурсов средств IPC;
* *ipcs* - вывод отчета о состоянии средств межпроцессного взаимодействия;
* *ipcrm* - удаление очередей сообщений, наборов семафоров и разделяемых сег- ментов памяти.

### Утилита ipcmk

ipcmk [options]

*ipcmk* позволяет открывать разделяемые сегменты памяти, очереди сообще- ний и массивы семафоров.

**Имеются следующие опции**:

-M, --shmem size

Открытие разделяемого сегмента памяти размером *size* байт;

-Q, --queue

Открытие очереди сообщений;

-S, --semaphore number

Открытие массива семафоров с *number* элементами;

-p, --mode mode

Установка доступа к ресурсу; по-умолчанию = *0644*;

-V, --version

Отображение версии пакета;

-h, --help

### Утилита ipcs

ipcs [-abcmopqstMQSTy] [-C дамп] [-N система] [-u пользователь]

*ipcs* выводит информацию о системных средствах межпроцессного взаимо- действия *System V IPC*.

**Имеются следующие опции** (используйте *ipcs -h*, для конкретной системы):

-a Показать максимально возможное количество информации во время вывода данных об активных семафорах, очередях сообщений и разделяемых сегмен- тах памяти. Это эквивалентно указанию опций -b, -c, -o, -p и -t.

-b Показать максимально допустимые размеры активных семафоров, очередей сообщений и разделяемых сегментов памяти. «Максимальный допустимый размер» -- это максимальное количество байт в сообщении в очереди сообщений, размер разделяемого сегмента памяти или количество семафо-

ров в наборе семафоров.

-c Показать имя и группу создателя активных семафоров, очередей сообщений и разделяемых сегментов памяти.

-m Вывести информацию об активных сегментах разделяемой памяти.

-o Показать пиковое использование активных очередей сообщений и раз- деляемых сегментов памяти. «Пиковое использование» -- это количество сообщений в очереди сообщений, или количество процес- сов, подключенных к разделяемому сегменту памяти.

-p Показать информацию об идентификаторе процесса для активных сема-

форов, очередей сообщений и разделяемых сегментов памяти.

«Идентификатором процесса» является последний процесс, отпра- вивший или получивший сообщение из очереди сообщений, процесс, создавший семафор, или последний процесс, подключившийся или отключившийся от разделяемого сегмента памяти.

-q Вывести информацию об активных очередях сообщений.

-s Вывести информацию об активных семафорах.

-t Показать время доступа к активным семафорам, очередям сообщений и разделяемым сегментам памяти. Время доступа -- это время послед- ней операции управления IPC объектом, последняя отправка или приём сообщения, последнее подключение или отключение от разделя- емого сегмента памяти, или последняя операция с семафором.

-C дамп

Извлечь значения из списка имён (namelist) указанного дампа памяти ядра, вместо определённого по умолчанию /dev/kmem. Подразумевает -y.

-M Вывести системную информацию о разделяемой памяти.

-N система

Извлечь список имён из указанной системы, вместо определённой по умолчанию /boot/kernel/kernel. Подразумевает -y.

-Q Вывести системную информацию об очередях сообщений.

-S Вывести системную информацию о семафорах.

-T Вывести системную информацию о разделяемой памяти, очередях сооб- щений и семафорах.

-y Использовать интерфейс kvm вместо интерфейса sysctl для извлечения необходимой информации. Если ipcs запущена на работающей системе, использование kvm(3) потребует привилегии чтения из /dev/kmem.

-u пользователь

Вывести информацию о механизмах IPC для указанного пользователя. Пользователь может быть задан либо числовым идентификатором UID, либо регистрационным именем.

**Замечание** Если не указана ни одна из опций -M, -m, -Q, -q, -S, или -s, то выводится информация обо всех активных средствах IPC.

### Утлита ipcrm

ipcrm [-q msqid] [-m shmid] [-s semid] [-Q msgkey] [-M shmkey] [-S semkey] ...

*ipcrm* удаляет указанные очереди сообщений, семафоры и разделяемые сег- менты памяти из системы. Требуемые объекты System V IPC задаются идентифика- тором их создания или любым связанным с ними ключом.

Для выбора объектов IPC, которые будут удалены, используются следующие опции, которых может быть задано любое число и любая комбинация:

-q msqid

Удалить из системы очередь сообщений, связанную с идентификатором msqid.

-m shmid

Пометить для удаления разделяемый сегмент памяти, связанный с идентификатором shmid. Этот помеченный сегмент будет уничтожен после отключения от него последнего процесса.

-s semid

Удалить из системы набор семафоров, связанный с идентификатором semid.

-Q msgkey

Удалить из системы очередь сообщений, связанную с ключом msgkey.

-M shmkey

Пометить для удаления разделяемый сегмент памяти, связанный с ключом shmkey. Этот помеченный сегмент будет уничтожен после отключения от него последнего процесса.

-S semkey

Удалить из системы набор семафоров, связанный с ключом semkey.

**Замечание** Идентификаторы и ключи, связанные с этими объектами System V IPC, могут быть най- дены с помощью утилиты ***ipcs***.

## Семафоры

Для решения проблем *взаимого исключения* и *взаимной блокировки* были предприняты значительные усилия. Естественно, что указанные проблемы косну- лись и стандарта POSIX, в частности, стандарта *POSIX-2001*. Со временем, вопро- сы согласования действий между взаимодействующими процессами стали называть задачами *синхронизации*.

Одним из первых механизмов, предложенных для синхронизации поведения процессов, стали *семафоры*, концепцию которых описал Дейкстра (Dijkstra) в 1965 году. При разработке средств System V IPC семафоры вошли в их состав как неотъ- емлемая часть.

**Замечание**

Следует отметить, что набор операций над семафорами System V IPC отличается от

классического набора операций, предложенного Дейкстрой.

**Набор действий** над семафорами System V IPC включает три операции:

1. *A(S, n) — семафор S увеличивается на n;*
2. *D(S, n) – пока значение семафора S < n, процесс блокируется. Иначе процес не блокируется и выполняется операция S = S - n;*
3. *Z(S) – процесс блокируется до тех пор, пока значение семафора S не станет равным 0.*

**Основная идея** реализации этого механизма, предполагает, что:

* + семафор - это *минимальный примитив синхронизации*, служащий основой для более сложных механизмов синхронизации, определенных в прикладной программе;
  + у семафора есть *значение*, которое представляется целым числом в диапа- зоне *от 0 до 32767*;
  + прикладная реализация механизма синхронизации обеспечивается *набором (массивом) семафоров*, операции над этими наборами, для приложений яв- ляются *атомарными*;
  + *гарантом атомарности* операций на наборами является *ядро ОС*, в котором и реализованы механизмы синхронизации.

Непосредственно в пакете IPC, работа с семафорами осуществляется с помощью трех системных вызовов:

//Необходимо для совместимости со старыми версиями

//#include <sys/types.h>

//#include <sys/ipc.h>

#include <sys/sem.h>

int *semget* (key\_t key,

int

nsems, int

semflg);

int *semop* (int semid, struct sembuf \*sops, size\_t nsops); int *semctl* (int semid, int semnum, int cmd, ...);

Рассмотрим отдельно каждый из этих вызовов.

Системный вызов *semget(...)* предназначен для выполнения операции доступа к массиву IPC-семафоров или создание такого массива.

В случае успешного завершения функции, возвращается дескриптор System V IPC для этого массива, который является целым неотрицательным числом, одно- значно характеризующим массив семафоров внутри вычислительной системы и ис- пользуется в дальнейших операциях над ним:

int *semget* (key\_t key, int nsems, int semflg);

где *key* — ключ, ассоциированный с семафором и генерируемый функцией *ftok(...)*; *nsems* - задает число семафоров в наборе;

*semflg* — флаг семафора; обычно ассоциируется с константами:

* *IPC\_CREAT* — если массив для указанного ключа не существует, то он дол- жен быть создан;
* *IPC\_EXCL* — применяется совместно с флагом *IPC\_CREAT*; при совместном их использовании и существовании массива с указанным ключом, доступ к массиву не производится и констатируется ошибка; при этом, переменная

*errno*, описанная в файле *<errno.h>*, примет значение *EEXIST*.

**Дополнительные права доступа** могут иметь значения:

* 0400 — разрешено чтение для пользователя, создавшего массив;
* 0200 — разрешена запись для пользователя, создавшего массив;
* 0040 — разрешено чтение для группы пользователя, создавшего массив;
* 0020 — разрешена запись для группы пользователя, создавшего массив;
* 0004 — разрешено чтение для всех остальных пользователей;
* 0002 — разрешена запись для всех остальных пользователей.

**В случае ошибки**, возвращается *-1*, а переменной *errno* присваивается ее номер:

* *EACCES* - набор семафоров существует для ключа *key*, но вызывающий про- цесс не имеет прав на доступ к набору;
* *EEXIST* - набор семафоров существует для ключа *key*, а в *semflg* не включены флаги *IPC\_CREAT* и *IPC\_EXCL*;
* *ENOENT* - набора семафоров для ключа *key* не существует, а в *semflg* не включен флаг *IPC\_CREAT*;
* *EINVAL* - значение *nsems* меньше *0* или больше максимально возможного для набора количества семафоров (*SEMMSL*), или набор семафоров, соответст- вующий *key* уже существует и *nsems* больше, чем количество семафоров в

этом наборе;

* *ENOMEM* - набор семафоров должен быть создан, но недостаточно памяти для создания новой структуры;
* *ENOSPC* - набор семафоров должен быть создан, но при этом будет превы- шен системный лимит количества наборов семафоров (*SEMMNI*) или систем- ный лимит количества семафоров (*SEMMNS*).

**Замечание**

Вновь созданные семафоры *инициируются нулевым значением*.

Системный вызов *semop(...)* предназначен для выполнения операций ***A***, ***D*** и ***Z***

на основе ключа, который получен системным вызовом *semget(...)*:

int *semop* (int semid, struct sembuf \*sops, size\_t nsops);

где *semid* — идентификатор набора семафоров, созданный функцией *semget(...)*; *sops* — указатель на массив структур, с числом элементов *nsops*, состоящим из структур типа *sembuf*;

*nsops* - число семафоров, указанное при вызове *semget(...)*.

При успешном завершении возвращает *0*, иначе — возвращает *-1*, а перемен- ной *errno* присваивается номер ошибки:

Действия, которые выполняются над набором семафоров, определяются вто- рым аргументом системного вызова *semop(...)*, являющимся указателем на массив сируктур типа *sembuf*.

**Каждая структура** типа *sembuf* содержит, по крайней мере, следующие три

поля:

unsigned short sem\_num; // Номер семафора в наборе (нумерация с нуля)

short

sem\_op; // Запрашиваемая операция над семафором

short

sem\_flg; // Флаги операции: 0 — операции с блокировкой

**Операция** над конкретным семафором определяется значением поля *sem\_op*:

* *положительное значение* предписывает увеличить значение семафора на ука- занную величину, соответствует операции ***A(S,n)***;
* *отрицательное* - уменьшить, соответствует операции ***D(S,n)***;
* *нулевое* - сравнить с нулем, соответствует операции ***Z(S)***.

**Замечание** Вторая операция не может быть успешно выполнена, если в результате значение сема- фора становится отрицательным, *а третья* - если значение семафора ненулевое. В та- ких случаях, процесс выполняющий операцию — *блокируется*.

С точки зрения пользовательского процесса, выполнение операций над мас- сивом семафоров является неделимым действием. Это значит:

* *если операции выполняются*, то только все вместе;
* *никакой другой процесс не может получить доступ* к промежуточному сос- тоянию набора семафоров, когда часть операций из массива уже выполни- лась, а другая еще не успела.

Сама ОС выполняет операции из массива семафоров - по очереди, причем по- рядок их обработки - не оговаривается.

Если очередная операция не может быть выполнена, то:

* эффект предыдущих операций *аннулируется*;
* вызов функции *semop(...)* приостанавливается (операция с блокировкой, когда

*sem\_flg=0*) или немедленно завершается неудачей, когда выполняется *опера-*

*ция без блокировки*.

**Замечание**

В случае неудачного завершения вызова *semop(...)* значения всех семафоров в наборе останутся неизменными.

Последний изучаемый в данной теме системный вызов *semctl(...)* предназна- чен *для управления набором семафоров*. Он может иметь три или четыре аргумен- та:

int *semctl* (int semid, int semnum, int cmd, [union semun arg]);

где *semid* - идентификатор набора семафоров;

*semnum* - номер семафора в наборе, определяющий объект, над которым выполня- ется управляющее действие, задаваемое значением третьего аргумента *cmd*; если объектом является набор, значение *semnum* — *игнорируется*;

*cmd* - набор команд выполняемых на семафорами; для некоторых команд исполь- зуется четвертый аргумент *arg*, имеющий вид:

union semun { int

val;

// Значение для SETVAL

struct semid\_ds \*buf;

unsigned short

} arg;

\*array;

// Буфер

// Массив

для IPC\_STAT, IPC\_SET

для GETALL,

SETALL

Полный перечень значений аргумента *cmd* следует изучать по руководству:

***man semctl***.

Далее рассмотрим наиболее важные значения *cmd*.

**Одиночные операции над семафором**:

* *GETVAL* - получить значение семафора и выдать его в качестве результата;
* *SETVAL* - установить значение семафора равным **arg.val**.

val = semctl (semid, semnum, GETVAL);

arg.val = ...;

if (semctl (semid, semnum, SETVAL, arg) == -1) ...;

**Групповые операции над набором семафоров**:

* *GETALL* - прочитать значения всех семафоров набора и поместить их в мас- сив **arg.array**;
* *SETALL* - установить значения всех семафоров набора равными значениям элементов массива.

arg.array = (unsigned short \*) malloc (nsems \* sizeof (unsigned short));

err = semctl (semid, 0, GETALL, arg);

for (i = 0; i < nsems; i++) arg.array [i] = ...; err = semctl (semid, 0, SETALL, arg);

**Информационные операции**:

* *GETPID* - узнать PID процесса, выполнившего последнюю операцию над се- мафором;
* *GETNCNT/GETZCNT* — узнать число процессов, ожидающих *увеличения / обнуления* значения семафора.

lpid = semctl (semid, semnum, GETPID);

ncnt = semctl (semid, semnum, GETNCNT); zcnt = semctl (semid, semnum, GETZCNT);

**Управляющие команды над семафорами**:

* *IPC\_STAT* - получить информацию о состоянии набора семафоров ;
* *IPC\_SET* - переустановить характеристики;
* *IPC\_RMID* - удалить набор семафоров.

arg.buf = (struct semid\_ds \*) malloc (sizeof (struct semid\_ds); err = semctl (semid, 0, IPC\_STAT, arg);

arg.buf->sem\_perm.mode = 0644;

err = semctl (semid, 0, IPC\_SET, arg);

err = semctl (semid, 0, IPC\_RMID);

В случае ошибки, *semctl(...)* вернет значение *-1*, а переменная *errno* будет иметь одно из значений:

* *EACCES* - вызывающий процесс не имеет права доступа, необходимого для запуска *cmd*;
* *EFAULT* - адрес, указанный *arg***.buf** или *arg***.array** , недоступен;
* *EIDRM* - набор семафоров был удален;
* *EINVAL* - неверное значение *cmd* или *semid*;
* *EPERM* - аргумент *cmd* имеет значение *IPC\_SET* или *IPC\_RMID*, но вызыва- ющий процесс не имеет достаточных привилегий на выполнение команды;
* *ERANGE* - аргумент *cmd* имеет значение *SETALL* или *SETVAL*, или значение, присваиваемое *semval*, для некоторых семафоров в наборе, меньше нуля или больше, чем стандартное значение *SEMVMX*.

## Задача об обедающих философах

Как уже было отмечено ранее, семафоры:

* *обеспечивают* программиста наддежными *средствами синхронизации* про- цессов;
* *требуют* от программиста разработки и реализации алгоритмов, обеспечива- ющих *взаимное исключение* процессов и устранение их *взаимных блокировок*.

В сложных реальных задачах, в которых взаимодействуют множество асинх- ронно выполняющихся процессов, разработка надежных алгоритмов синхрони- зации также превращается в самостоятельную проблему.

В таких случаях, программисту требуется правильно провести декомпозицию алгоритма работы приложений и реализовать его по частям.

В данном подразделе, на классическом примере «*Задачи об обедающих фило- софах*», мы рассмотрим основные этапы подготовки и реализации проекта по син- хронизации процессов, которые могут приводить к *взаимным блокировкам*.

### Описание задачи

**Класический вариант** «*Задачи об обедающих философах*» демонстрируется рисунком 1.2 и следующим описанием:

* За круглым столом сидит несколько философов.
* В каждый момент времени каждый из них *либо беседует*, *либо ест*.
* Для процесса еды одновременно *требуются две вилки*. Поэтому, прежде чем в очередной раз перейти от беседы к приему пищи, философу *необходимо до-*

*ждаться, пока освободятся обе вилки* - слева и справа от него, и взять их в руки.

* Немного поев, философ *кладет вилки на стол* и вновь присоединяется к бе- седе.
* Требуется разработать программную модель обеда философов.

**Главная проблема в этой задаче,** - *корректная дисциплина захвата и осво- бождения вилок*, иначе например, если каждый из философов, одновременно с дру- гими, возьмется за вилку, лежащую слева от него, и будет ждать освобождения пра- вой, то - *обед не завершится никогда*.

Прежде, чем приступить к реализации алгоритма синхронизаци *процессов- философов*, определимся с общей стратегией решения задачи.

### Выбор стратегии решения

Одним из общих подходов решения задач синхронизации множества асинх- ронно взаимодействующих процессов является разделение структуры всего алго- ритма на *два уровня иерархии*:

* первый — *нижний уровень* реализует сами действующие процессы, выделяя те общие ресурсы, которые нужны ему для обеспечия функционирования;
* второй — *верхний уровень* (программа-монитор) обеспечивает нижний уро- вень необходимыми данными, проводит первоначальную подготовку разделя- емых ресурсов и, по возможности, устраняет возникающие проблемы синх-

ронизации процессов нижнего уровня.

Для нашего случая:

* *нижний уровень* — программы *процессов-философов*, которые борятся за ре- сурс в виде двух вилок;
* *верхний уровень* — *программа-монитор*, которая подготавливает набор сема- форов для синхронизации всех процессов-философов, создает и запускает эти процессы и отслеживает их завершение.

Далее, переходим к построению и реализации моделей каждого уровня.

### Модель философа

Каждый философ ведет себя независимо от других: он ест и беседует, причем время, необходимое для беседы и поедания порции (части) своего обеда, не привя- зано к аналогичным действиям других философов.

Для моделирования такой независимости *предполагаем*, что:

* время *одной порции разговора* философа - случайное значение ***trnd***;
* время *одной порции поедания* обеда философа - случайное значение ***ernd***;
* общее *время поедания всего обеда* — ограничено, поэтому философ может завершить обед в несколько приемов.

**Описание действий философа**:

* философ какое-то время беседует (***trnd***), затем пытается взять вилки слева и справа от себя;
* когда ему это удается, он некоторое время ест (**ernd**), после чего освобождает вилки;
* так продолжается до тех пор, пока не будет съеден весь обед, после чего про- грамма, моделирующая действия философа, закончит свою работу.

**Таким образом**, мы видим, что для нормальной работы философа необходи- мы два ресурса, которые обеспечиваются двумя семафорами.

Чтобы реализовать корректную модель *программы-философа*, необходимо уточнить исходные данные, передаваемые ей в качестве аргументов.

Учитывая, что отдельный философ использует только два разделяемых ре- сурса из общего их колличества, равного общему числу философов, то для запуска программы достаточно только трех аргументов:

* *argv[1]* — идентификатор набора семафоров;
* *argv[2]* — общее число философов;
* *argv[3]* — порядковый номер конкретного философа за столом.

**Замечание**

Философы нумеруются, начиная с 1.

Вилки нумеруются, начиная 0. Номер вилки — номер семафора.

Значение семафора = 1, когда вилка — свободна. Значение семафора = 0, когда вилка — занята.

### Программа-монитор

Для рассматриваемой задачи, учитывая, что количество обедающих филосо- фов задано некоторой константой ***QPH***, алгоритм работы достаточно прост:

* *порождается* набор семафоров ***QPH***: по одному семафору на каждую вилку;
* *устанавливаются* начальные значения семафоров: занятой вилке будет соот- ветствовать значение 0, свободной — 1;
* запускаются ***QPH*** процессов, каждый из которых представляет одного фило- софа, передавая им в качестве аргументов: идентификатор набора семафоров,

общее количество всех философов и место конкретного философа за столом: философы нумеруются от 1 до ***QPH***;

* *ожидается* завершение работы всех процессов: когда все философы съедят свой обед;
* *удаляется* весь набор семафоров.