## 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.