МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ»

Кафедра автоматизированных систем управления (АСУ)

УТВЕРЖДАЮ

Зав. кафедрой АСУ, профессор

 А.М. Кориков

АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ КОМПЛЕКСОВ

Лабораторная работы №2: «POSIX. Сигналы процессов»

Учебно-методическое пособие

для студентов уровня основной образовательной программы магистратура
направления подготовки 010400.68 «Прикладная математика и информатика»
профиля Математическое и программное обеспечение вычислительных комплексов и компьютерных сетей

Разработчик

доцент кафедры АСУ

В.Г. Резник

2014

Резник В.Г.

Архитектура вычислительных комплексов. Лабораторная работа №2: POSIX. Сигналы процессов. Учебно-методическое пособие. – Томск, ТУСУР, 2012. – 40 с.

Учебно-методическое пособие предназначено для выполнения лабораторной работы №2 по дисциплине «Архитектура вычислительных комплексов» для студентов уровня основной образовательной программы магистратура направления подготовки 010400.68 «Прикладная математика и информатика» профиля «Математическое и программное обеспечение вычислительных комплексов и компьютерных сетей».

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Данное учебно-методическое пособие содержит краткое описание назначения стандарта POSIX, а также методические рекомендации по его применению для выполнения лабораторных работ по дисциплине «Архитектура вычислительных комплексов» (АВК). При изложении материала, использовалась информация сайта [1].

Тема лабораторной работы: «***POSIX. Сигналы процессов***».

Знание основ стандарта POSIX входит в набор общекультурных компетенций студента. Умение студента разрабатывать программы, удовлетворяющие этому стандарту, составляет основу профессиональных компетенций по дисциплине АВК.

В целом, лабораторная работа №2 формирует базовые знания по разработке ПО на основе вычислительного комплекса УПК АСУ, а также вырабатывает умения, необходимые для выполнения последующих работ по данной дисциплине.

Последовательность и изложение материала данного пособия предполагает, что студент:

- успешно выполнил задания по лабораторной работе №1 на тему: «Подготовка работы на УПК АСУ»;
- владеет теоретическими знаниями и практическим умением, которые полученны при изучении дисциплины «Современные операционные системы»;
- имеет практические навыки разработки прикладного ПО на языке программирования С.

Учебный материал данного пособия разбит на пять разделов:

- ***первый раздел*** дает описание основных понятий стандарта POSIX. Эта информация содержит знания, которыми студент должен пополнить свою общекультурную компетенцию по данной дисциплине;
- ***во втором разделе*** даны основные понятия процесса, описаны его атрибуты и базовый список функций на языке С, позволяющий работать с атрибутами процессов;
- ***третий раздел*** содержит описание назначения сигналов и функций, обеспечивающих передачу и прием сигналов процессами;
- ***в четвертом разделе*** ставится задача на выполнение лабораторной работы и дается описание этапов, обеспечивающих успешное завершение учебного задания;
- ***пятый раздел*** пособия содержит рекомендации по подведению итогов лабораторной работы №2.

1. ОСНОВНЫЕ ПОНЯТИЯ СТАНДАРТА POSIX

Один из общепринятых способов повышения мобильности ПО - обеспечение стандартизации окружения приложений, что подразумевает стандартизацию ПО ОС, утилит и программных интерфейсов этих приложений. Таким средством является стандарт **POSIX** (***Portable Operating System Interface***), который описывает интерфейс операционной системы.

Название **POSIX** предложено известным специалистом, основателем Фонда свободного программного обеспечения, Ричардом Столмэном.

Наиболее современная версия стандарта POSIX, в редакции 2003 г., основана на Техническом стандарте ***Open Group IEEE Std 1003.1*** и на международном стандарте ***ISO/IEC 9945***.

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 С-интерфейса к системным сервисам;
3. описание интерфейса к системным сервисам на уровне командного языка и служебных программ ;
4. детальное разъяснение положений стандарта, обоснование принятых решений.

В дальнейшем накапливались и были внесены многие мелкие исправления, учтенные в редакции ***2003*** года.

Основные идеи стандарта POSIX описываются множеством базовых, системных сервисов, необходимых для функционирования прикладных программ. Доступ к этим сервисам предоставляется посредством интерфейса, специфицированного для языка С, командного языка и общеупотребительных служебных программ.

Подчеркнем, что у каждого интерфейса есть две стороны: ***вызывающая*** и ***вызываемая***. Стандарт POSIX ориентирован в первую очередь на вызывающую сторону.

Цель стандарта - сделать приложения мобильными на уровне исходного языка. Это значит, что при переносе программ языка С на другую операционную платформу потребуется только новая компиляция исходных текстов программ.

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

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

Ориентация POSIX на международный стандарт языка С определила также направление развития спецификаций POSIX в плане синхронизации обоих стандартов:

- В стандарте проведено разделение на обязательные и дополнительные функции.
- Особое внимание уделяется ***способам реализации стандартизуемых функций*** как в "классической" Unix-среде, так и на других операционных платформах, в сетевых и распределенных конфигурациях.

В редакции 2003-го года, стандарт POSIX рассматривает следующие категории 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, заголовочные 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***, а также ***содержать нестандартные функции***.

Если утверждается, 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 возможностям***.

Например, константа **_POSIX2_C_DEV** обслуживает средства разработки на языке С.

Анализируя эти константы во время компиляции, приложение выяснит возможности используемой ОС и подстроится под них. Аналогичные действия могут быть выполнены с помощью функции **sysconf()** и/или служебной программы **getconf**.

Для минимизации размеров ОС и приложений, стандартом POSIX была предусмотрена весьма мелкая гранулярность необязательных возможностей. 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 стандарту POSIX, описаны поддерживаемые дополнительные и нестандартные возможности.

Применительно непосредственно к ОС определены следующие основные понятия, соответствующие стандарту POSIX:

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

Это - первичные понятия. Они строго не определяются, а поясняются с помощью других понятий и отношений. Для каждого из них описаны присущие им атрибуты и применимые к ним операции.

В стандарте POSIX содержатся пояснения следующих основных 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-совместимых систем, применяются или командный интерпретатор (языка shell) и/или компилируемый язык C:

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

Функциональный интерфейс операционных систем естественно считать первичным, но в POSIX-совместимых ОС определены объекты, которые считаются вспомогательными. Они обеспечивают организацию взаимодействия между основными сущностями. Примерами таких объектов являются средства межпроцессного взаимодействия. Процессы выполняются в определенном окружении, частью которого является языково-культурная среда (Locale), образованная такими категориями, как символы и их свойства, форматы сообщений, дата и время, числовые и денежные величины.

Каждый процесс ассоциирован, по крайней мере, на три файла:

- стандартный ввод;
- стандартный вывод;
- стандартный протокол.

Обычно стандартный ввод назначается на клавиатуру терминала, а стандартный 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.

К ОС также могут предъявляться качественные требования, например, требование поддержки реального времени: способность обеспечить необходимый сервис в течение заданного отрезка времени.

Стандарт POSIX определяет ряд требований к среде компиляции POSIX-совместимых приложений.

Часто разработка приложений ведется в кросс-режиме. Поэтому на каждой инструментальной платформе создается такая среда компиляции приложений, чтобы результат компиляции можно было перенести 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, какие заголовочные файлы должны быть включены использующим ее приложением (обычно требуется один файл).

Посредством символических констант, определенных в заголовочном файле **<unistd.h>**, операционная система предоставляет приложению информацию о поддерживаемых возможностях.

Стандартом POSIX также предусмотрен симметричный механизм, называемый механизмом макросов проверки возможностей. Он позволяет приложениям объявлять о своем желании получить доступ к определенным прототипам и именам.

Замечание 1.
Основным требованием к приложениям, строго соответствующим стандарту POSIX, является необходимость определения символической константы ***_POSIX_C_SOURCE*** со значением ***200112L*** до включения каких-либо заголовочных файлов.
Таким образом POSIX-совместимое приложение заявляет, что ему нужны POSIX-имена.

Близкую по смыслу роль играет макрос ***_XOPEN_SOURCE*** (со значением 600).

В качестве примера может служить следующий фрагмент заголовочного файла:

```
#if defined(_REENTRANT) || (_POSIX_C_SOURCE - 0 >= 199506L)
#define LIBXML_THREAD_ENABLED
#endif
```

С целью не допустить пересечения имен, префиксы ***posix_***, ***POSIX_*** и ***_POSIX_***, которые зарезервированы для нужд стандарта.

Замечание 2.
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 т.п. Еще имеется четыреста семнадцать имен типа _Exit, abort, abs, acos и т.д., которые могут участвовать в редактировании внешних связей прикладной программы.

Мобильность приложений, соответствующих стандарту POSIX, принципиально достижима благодаря двум основным факторам:

- во-первых - наличию огромного числа стандартизованных системных сервисов,
- 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.

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

Необходимая степень детерминизма выполнения приложений достигается благодаря ***средствам поддержки реального времени***, к которым относятся управление дисциплиной выделение процессоров, сигналы реального времени, удержание страниц в оперативной памяти, таймеры высокого разрешения и т. д.

Функции для работы с файлами удовлетворяют потребности приложений в чтении и записи долговременных данных, защите таких данных от несанкционированного доступа. Механизм блокировки фрагментов файлов позволяет обеспечить ***атомарность транзакций***. Асинхронный ввод/вывод дает возможность совмещать операции обмена, оптимизируя тем самым

приложения. С помощью множества служебных программ можно относительно легко организовать сложную обработку данных.

Тщательно проработаны вопросы доступа к внешним устройствам, которые подсоедины по последовательным линиям, особенно к терминалам.

Стандартизованный командный язык shell обеспечивает адекватное средство 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 одновременное выполнение нескольких программ и взаимодействие с ними.

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

Важнейшим понятием стандарта POSIX является языково-культурная среда. Приложения способны определять нужную им среду и адаптироваться к потребностям пользователей.

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

Стандарт POSIX - обязательный элемент современной дисциплины разработки прикладных систем.

2. АТРИБУТЫ И ФУНКЦИИ ПРОЦЕССОВ

Процессы являются основными функциональными элементами операционных систем (ОС). В стандарте POSIX-2001 определение процесса дается, через определение его атрибутов. Данный раздел учебно-методического пособия содержит краткое описание атрибутов и системных функций, позволяющих управлять процессами.

Одним из средств, с помощью которых можно воздействовать на процессы, являются сигналы.

Замечание 3.
Сигналы и их свойства описаны в следующем разделе.

Далее, в таблице 2.1, даны определения основных атрибутов, уточняющих понятие процесса.

Таблица 2.1. Атрибуты, уточняющие понятие процесса

Понятие	**Определение**
Процесс	Адресное пространство вместе с выполняемыми в нем потоками управления, а также системными ресурсами, которые этим потокам требуются.
Идентификатор процесса	Положительное целое число, которое однозначно идентифицирует 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 другому процессу.
Родительский процесс	Процесс, создавший данный процесс.
Группа процессов	Совокупность процессов, допускающая согласованную доставку сигналов. У каждой группы имеется уникальный положительный целочисленный идентификатор, представляющий ее в течение 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 последний процесс (по причине завершения или смены группы).
Задание	Набор процессов, составляющих конвейер, а также порожденных ими процессов, входящих в одну группу.
Управление заданиями	Предоставленные пользователям средства выборочно приостанавливать и затем продолжать (возобновлять) выполнение процессов. На отдельные задания ссылаются с помощью идентификаторов.
Сеанс	Множество групп процессов, сформированное для целей управления 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 сеанса	Период от создания сеанса до истечения времени жизни всех 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 последовательности символов, вводимые с управляющего терминала, вызывают посылку сигналов всем процессам группы, ассоциированной с данным управляющим терминалом.
Управляющий процесс	Лидер сеанса, установивший соединение с управляющим терминалом. Если в дальнейшем терминал перестанет быть управляющим 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 управляющим терминалом, может быть не более одной группы процессов, приоритетной по отношению к данному управляющему терминалу.

Фоновые процессы	Неприоритетные к данному управляющему терминалу процессы.
Реальный идентификатор пользователя процесса	Идентификатор пользователя, создавшего процесс.
Реальный идентификатор группы процессов	Идентификатор группы пользователя, создавшего процесс.

Кроме приведенных в таблице 2.1, имеются и другие атрибуты процессов, несущественные для выполнения данной лабораторной работы.

Для выдачи информации о процессах служит утилита ***ps***:

```
ps [-aA] [-defl] [-G список_групп]
   [-o формат] ... [-p список_процессов]
   [-t список_терминалов]
   [-U список_пользователей]
   [-g список_групп]
   [-n список_имен]
   [-u список_пользователей]
```

По умолчанию, информация выдается обо всех процессах, имеющих тот же действующий идентификатор и тот же управляющий терминал, что и у текущего пользователя. При этом выводятся идентификатор процесса, имя управляющего терминала, истраченное к данному моменту процессорное время и имя программы (команды), выполняемой в рамках процесса. Например:

```
PID    TTY    TIME CMD

1594  ttyS4 00:00:02 sh
1645  ttyS4 00:00:00 sh
1654  ttyS4 00:02:45 rk.20.01
18356 ttyS4 00:00:00 prconecm
18357 ttyS4 00:00:00 sh
18358 ttyS4 00:00:00 ps
```

Если нужна более подробная информация о более широком наборе процессов, следует пользоваться опциями.

Для опроса идентификаторов процесса, родительского процесса и группы процессов из программы, написанной на языке С, предусмотрены функции ***getpid(...)***, ***getppid(...)***, ***getpgrp(...)***:

```
#include <unistd.h>
pid_t getpid (void);
```

```
#include <unistd.h>
pid_t getppid (void);

#include <unistd.h>
pid_t getpgrp (void);
```

По стандарту эти функции всегда завершаются успешно, поэтому ошибочных кодов возврата не предусмотрено. Для более подробного изучения этих и приведенных далее функций, следует использовать утилиту ***man***.

Для установки идентификатора группы процессов в целях управления заданиями предназначена функция ***setpgid()***:

```
#include <unistd.h>
int setpgid (pid_t pid, pid_t pgid);
```

Выполнение функции ***setpgid(...)*** влечет либо присоединение к существующей группе процессов, либо создание новой группы в рамках сеанса, в который входит вызывающий процесс. Процесс может установить идентификатор группы для себя или для порожденного процесса. Нельзя изменить идентификатор группы процессов лидера сеанса.

В случае успешного завершения функции ***setpgid(...)*** (результат при этом равен нулю) идентификатор группы процессов устанавливается равным ***pgid*** для заданного аргументом ***pid*** процесса. Если значение ***pid*** равно нулю, установка производится для вызывающего процесса. А если значение ***pgid*** равно нулю, то в качестве идентификатора группы процессов используется идентификатор процесса, заданного аргументом ***pid***.

Для создания сеанса и установки идентификатора группы процессов служит функция ***setsid(...)***:

```
#include <unistd.h>
pid_t setsid (void);
```

Если вызывающий процесс не является лидером группы, в результате выполнения функции ***setsid()*** будет создан новый сеанс, причем вызывающий процесс станет лидером этого сеанса, равно как и лидером новой группы процессов (без управляющего терминала и без других процессов в группе и сеансе).

Примером использования описанных выше функций, является программа:

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
int main (void) {
                pid_t ppid;
        pid_t pgid;
        /* Отменим буферизацию стандартного вывода */
        setbuf (stdout, NULL);
        printf ("Атрибуты текущего процесса: pid: %d,
                ppid: %d, pgid: %d\n",
                getpid (), ppid = getppid (), pgid =
                getpgrp ());
        /* Выделимся в новую группу */
        if (setpgid (0, 0) != 0) {
                perror ("setpgid (0, 0)");
        }
        printf ("Новая группа текущего процесса: %d\n",
        getpgrp ());
        /* Попробуем создать новый сеанс */
        if (setsid () == (pid_t) (-1)) {
                perror ("setsid от имени лидера группы");
        }
        /* Вернемся в прежнюю группу */
        if (setpgid (0, pgid) != 0) {
                perror ("setpgid (0, pgid)");
        }
        printf ("Группа текущего процесса после повторной
                смены: %d\n", getpgrp ());
        /* Повторим попытку создать новый сеанс */
        if (setsid () == (pid_t) (-1)) {
                perror ("setsid от имени не-лидера группы");
        }
        printf ("Группа текущего процесса после создания
                нового сеанса: %d\n", getpgrp ());
        /* Попробуем сменить группу родительского
        процесса */
        if (setpgid (ppid, 0) != 0) {
                perror ("setpgid (ppid, 0)");
        }
        /* Попробуем установить несуществующий */
        /* идентификатор группы процессов */
        if (setpgid (0, ppid) != 0) {
                perror ("setpgid (0, ppid)");
        }
        return (0);
}
```

Результат работы этой программы может быть таким:

```
Атрибуты текущего процесса: pid: 11726, ppid:
        11725, pgid: 1153
Новая группа текущего процесса: 11726
        setsid от имени лидера группы: Operation not permitted
Группа текущего процесса после повторной смены:
        1153
Группа текущего процесса после создания нового
        сеанса: 11726
setpgid (ppid, 0): No such process
setpgid (0, ppid): Operation not permitted
```

Новые процессы создаются при помощи функции ***fork(...)***:

```
#include <unistd.h>
pid_t fork (void);
```

Новый (порожденный) процесс является точной копией процесса, вызвавшего ***fork(...)*** (родительского), за исключением следующих моментов:

1. У порожденного процесса свой идентификатор, равно как и идентификатор родительского процесса.
2. У порожденного процесса собственная копия файловых дескрипторов, ссылающихся на те же описания открытых файлов, что и соответствующие дескрипторы родительского процесса.
3. Порожденный процесс не наследует блокировки файлов, установленные родительским процессом.
4. Порожденный процесс создается с одним потоком управления – копией того, что вызвал ***fork(...)***.
5. Имеются также некоторые тонкости, связанные с обработкой сигналов, на которых мы, однако, 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, расположенные за обращением к ***fork(...)***.

При неудаче родительскому процессу возвращается ***-1***, ***новый процесс не создается***.

Поскольку возвращаемые функцией ***fork(...)*** значения различны для обеих копий, родительский и порожденный процессы могут далее выполняться по-разному. Например, процесс-предок переходит в состояние ожидания завершения процесса-потомка либо, если процесс-потомок запущен асинхронно, продолжает выполнение параллельно с ним. Процесс-потомок при помощи функции семейства ***exec(...)*** подменяет программу, которая определяет поведение процесса, и передает ей управление и список аргументов.

Заголовок функции ***main(...)*** программы на языке С, в общем случае, имеет вид:

```
      int main (int argc, char *argv []);
```

Значение ***argc*** равно количеству аргументов;
argv – это массив указателей собственно на аргументы, которые определяются исходя из командной строки, запускающей программу на языке С.

В соответствии с принятым соглашением, значение ***argc*** не меньше единицы, а первый элемент массива ***argv*** указывает на цепочку символов, содержащую имя выполняемого файла.

Аналогичный смысл имеют аргументы 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 []);
```

Функции семейства ***exec(...)*** заменяют текущий образ процесса новым (и, следовательно, в случае успешного завершения возврат в вызывающий процесс невозможен). Новый образ создается на основе выполнимого файла, который называется **файлом образа процесса**.

Переменная **environ** инициализируется как указатель на массив указателей на составляющие окружение цепочки символов.

Замечание 4.
Массивы **argv** и **environ** завершаются пустым указателем.

Аргумент **path** указывает на маршрутное имя файла с новым образом процесса.

Аргумент **file** имеет аналогичный смысл, однако, если он задан как простое имя, то производится поиск в каталогах, заданных переменной окружения PATH.

Аргументы **arg0**, ..., являются указателями на цепочки символов, составляющие список аргументов нового образа процесса. Последним в списке располагается пустой указатель, а аргумент ***arg0*** должен указывать на имя файла-образа.

Аргумент **envp** имеет тот же смысл и назначение, что и переменная ***environ***.

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.

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

```
#include <sys/wait.h>
pid_t wait (int *stat_loc);
pid_t waitpid (pid_t pid, int *stat_loc, int  options);
```

Функция ***waitpid()*** эквивалентна ***wait()***, если аргумент **pid** равен ***(pid_t) (-1)***, а аргумент **options** имеет нулевое значение. Аргумент **pid** задает набор порожденных процессов, статус завершения которых запрашивается. Значение ***(pid_t) (-1)*** представляет произвольный элемент множества порожденных процессов. Если ***pid > 0***, имеется в виду один процесс с данным идентификатором. Нулевое значение специфицирует любого потомка из той же группы процессов, что и вызывающий. Наконец, при ***pid < (pid_t) (-1)*** запрашивается статус завершения любого порожденного процесса из группы, идентификатор которой равен абсолютной величине ***pid***.

Аргумент **options** задается как побитное ***ИЛИ*** следующих флагов, определенных в заголовочном файле <***sys/wait.h***>:

- ***WNOHANG*** - функция ***waitpid()*** не приостанавливает выполнение вызывающего потока управления, если запрашиваемый статус не доступен немедленно.
- ***WUNTRACED*** - наряду со статусом завершения запрашивать статус остановленных, но еще не опрошенных процессов.

Если запрос статуса порожденного процесса завершился успешно, функции ***wait()*** и ***waitpid()*** возвращают идентификатор этого процесса и размещают по указателю ***stat_loc*** (если он отличен от ***NULL***) значение, которое будет нулевым тогда и только тогда, когда выдан статус порожденного процесса, который завершился по одной из трех причин:

- произошел возврат из функции ***main(...)*** с нулевым результатом;
- порожденный процесс вызвал функцию ***_exit()*** или ***exit()*** с нулевым аргументом;
- завершились все потоки управления порожденного процесса.

Для анализа целочисленного значения ***stat_val***, на которое указывает аргумент ***stat_loc***, в файле <***sys/wait.h***> определен набор макросов.

Например, значение ***WIFEXITED (stat_val)*** будет ненулевым в случае нормального завершения порожденного процесса; при этом ***WEXITSTATUS (stat_val)*** возвращает младший байт статуса.
Макрос ***WIFSTOPPED (stat_val)*** возвращает ненулевое значение, если получен статус остановленного процесса.

Процесс может вызвать собственное завершение, обратившись к функциям семейства ***exit(...)***:

```
#include <stdlib.h>
void exit (int status);
void _Exit (int status);

#include <unistd.h>
void _exit (int status);
```

Значением аргумента **status** могут служить константы ***0***, ***EXIT_SUCCESS***, ***EXIT_FAILURE*** или любое другое значение, хотя ожидающему родительскому процессу будет доступен только младший байт статуса (***status & 0377***).

Если родительский процесс выполняет функции ***wait(...)*** или ***waitpid(...)***, ему передается младший байт значения ***status***. Если родительский процесс этого не делает, функции семейства ***exit(...)*** переводят вызывающий процесс **в состояние зомби**.

Если у завершающегося процесса были потомки, родительским для них становится системный процесс, определяемый реализацией.

Функция **atexit(...)** позволяет зарегистрировать функции, которые будут вызываться, если процесс завершается, обращаясь к ***exit(...)*** или возвращаясь из ***main(...)***.

```
#include <stdlib.h>
int atexit (void (*func) (void));
```

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

Ниже приведен пример использования функций порождения и завершения процессов:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>

static void atefunc (void) {

/* Перед завершением выдадим информацию о */
/* процессах */
```

```
printf ("Ситуация перед завершением
родительского процесса\n");
(void) system ("ps -o pid,ppid,vsz,args");
}

int main (void) {
        int pid;
        int stat;
        /* Отменим буферизацию стандартного вывода */
        setbuf (stdout, NULL);
        /* Зарегистрируем обработчик завершения процесса */
        if (atexit (atefunc) != 0) {
                perror ("ATEXIT");
                exit (1);
        }
        /* Создадим новый процесс */
        if ((pid = fork ()) < 0) {
                perror ("FORK");
                exit (2);
        } else if (pid == 0) {
                /* Порожденный процесс */
                /* Выполним служебную программу ps */
                printf ("Ситуация с точки зрения порожденного
                        процесса\n");
                (void) execl ("/bin/ps", "ps", "-o",
                        "pid,ppid,args", (char *) 0);
                perror ("EXEC");
                exit (3); /* execl() завершился неудачей */
        } else {
                /* Родительский процесс */
                sleep (1);
                /* Вероятно, порожденный процесс уже */
                /* завершился */
                /* Посмотрим на текущую ситуацию */
                printf ("Ситуация перед вызовом waitpid() в
                        родительском процессе\n");
                (void) system ("ps -o pid,ppid,vsz,args");
                (void) waitpid (pid, &stat, 0);
                printf ("Статус завершения порожденного
                процесса с идентификатором %d: %d\n", pid, stat);
        }
        return 0;
}
```

Результат работы этой программы может выглядеть так:

```
Ситуация с точки зрения порожденного процесса
PID PPID COMMAND
6123 1072 -sh
29568 6123 prog30
29569 29568 ps -o pid,ppid,args
Ситуация перед вызовом waitpid() в родительском
        процессе
PID PPID VSZ COMMAND
6123 1072 2260 -sh
29568 6123 1340 prog30
29569 29568 0 [ps <defunct>]
29570 29568 2584 ps -o pid,ppid,vsz,args
Статус завершения порожденного процесса с
        идентификатором 29569: 0
Ситуация перед завершением родительского процесса
```

```
PID PPID VSZ COMMAND
6123 1072 2260 -sh
29568 6123 1340 prog30
29571 29568 2584 ps -o pid,ppid,vsz,args
```

В примере, ***информация о зомби-процессе*** выведена с пометкой <***defunct***>.

Для терминирования процессов извне, предназначена служебная программа ***kill***, подверженная контролю прав доступа и вызванная следующим образом:

```
kill -s TERM идентификатор_процесса ...
```

или

```
kill -s KILL идентификатор_процесса ...
```

3. СИГНАЛЫ - КАК СРЕДСТВО УПРАВЛЕНИЯ ПРОЦЕССАМИ

С прикладной точки зрения, каждый ***процесс*** — это прикладная программа, выполняющая расчетную часть некоторой задачи или обеспечивающая решение этой задачи. Для ОС, ***процесс*** — это функциональный элемент, требующий ресурсов процессора, оперативной памяти, файловой системы или некоторых средств коммуникации.

В конечном итоге, необходимые ресурсы вычислительной системы (комплекса), определяются алгоритмом решения задачи и потребностями самой задачи в ресурсах.

Ожидание задачей ресурсов имеет два негативных следствия:

- увеличивает общее время решения задачи;
- приводит к неэффективному использованию задачей процессора ЭВМ.

В ряде случаев, удается частично устранить эти недостатки, разбив задачу на подзадачи, которые решаются отдельными процессами ОС, и синхронизировав процессы с помощью средств передачи сигналов между процессами.

В данном разделе приводится описание инструментальных средств языка С, которые обеспечивают управление сигналами.

Стандарт POSIX-2001, под сигналом понимает механизм, с помощью которого процесс или поток управления уведомляют о некотором событии, 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 символов;
- процесс обратился к функции ***kill()*** и другие.

Иногда по одному событию генерируются сигналы для нескольких процессов, например, для группы процессов, ассоциированных с некоторым управляющим терминалом.

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

Сигналы, генерация которых ассоциирована с идентификатором процесса или группы процессов, а также с асинхронным событием, например, пользовательский 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 действие, соответствующее данным процессу и сигналу.

Сигнал принят процессом, когда он выбран и возвращен одной из функций ***sigwait(...)***.

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 заблокированным 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.

С сигналом могут быть ассоциированы действия 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 заданную функцию. После завершения функции обработки процесс возобновляет выполнение с точки прерывания.

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

```
void func (int signo);
```

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

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

Функция 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 ***повторно входимыми***, либо непрерываемыми сигналами, что превращает их в АСБ-функции. В этот список включены 117 функций, в том числе почти все из рассматриваемых нами.

Если сигнал доставляется потоку, а реакция заключается в завершении, остановке или продолжении, весь процесс должен завершиться, остановиться или продолжиться.

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

```
#include <signal.h>
int kill (pid_t pid, int sig);
```

Сигнал задается аргументом ***sig***, значение которого может быть нулевым; в этом случае действия функции ***kill()*** сводятся к проверке допустимости значения **pid**; нулевой результат - признак успешного завершения ***kill()***.

Если ***pid > 0***, это значение трактуется как идентификатор процесса. При нулевом значении ***pid***, сигнал посылается всем процессам из той же группы, что и вызывающий.
Если значение ***pid*** равно ***-1***, адресатами являются все процессы, которым вызывающий имеет право посылать сигналы.
При прочих отрицательных значениях ***pid***, сигнал посылается группе процессов, чей идентификатор равен абсолютной величине **pid**.

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

Вызов служебной программы **kill** в виде: ***kill -l***, позволяет вывести на терминал весь список сигналов:

```
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.

Стандарт POSIX-2001 определяет как обязательные для реализации сигналы, представленные в таблице 3.1.

Таблица 3.1. 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.

Процесс (***поток управления***) может послать сигнал самому себе с помощью функции ***raise(..)***.

Вызов ***raise(...)*** эквивалентен вызову функции: **kill (getpid(), sig);**

```
#include <signal.h>
int raise (int sig);
```

Посылка сигнала самому себе использована также в функции ***abort()***, которая по умолчанию вызовет аварийное завершение процесса. Если же функция обработки сигнала ***SIGABRT*** не возвратит управление, то завершение процесса не будет. Кроме того, функция ***abort()*** отменяет блокирование или игнорирование сигнала ***SIGABRT***.

```
#include <stdlib.h>
void abort (void);
```

Опросить и изменить способ обработки сигналов позволяет функция ***sigaction()***:

```
#include <signal.h>
int sigaction (int sig, const struct sigaction
    *restrict act, struct sigaction
        *restrict oact);
```

Для описания способа обработки сигнала используется структура **sigaction**, которая должна содержать по крайней мере следующие поля:

```
void (*sa_handler) (int);
/* Указатель на функцию обработки сигнала   */
/* или один из макросов SIG_DFL или SIG_IGN */
sigset_t sa_mask;
/* Дополнительный набор сигналов, блокируемых */
/* на время выполнения функции обработки      */
int sa_flags;
/* Флаги, влияющие на поведение сигнала       */
void (*sa_sigaction) (int, siginfo_t *, void *);
/* Указатель на функцию обработки сигнала     */
```

Замечание 5.
Приложение, соответствующее стандарту, не должно одновременно использовать поля обработчиков ***sa_handler*** и ***sa_sigaction***.

Тип **sigset_t** может быть целочисленным или структурным и представлять набор сигналов.

Тип **siginfo_t** должен быть структурным по крайней мере со следующими полями:

```
int si_signo;   /* Номер сигнала */
int si_errno;
/* Значение переменной errno, ассоциированное
с данным сигналом */
int si_code;
/* Код, идентифицирующий причину сигнала */
pid_t si_pid;
/* Идентификатор процесса, пославшего сигнал */
uid_t si_uid;
/* Реальный идентификатор пользователя
процесса, пославшего сигнал */
void *si_addr;
/* Адрес, вызвавший генерацию сигнала */
int si_status;
/* Статус завершения порожденного процесса */
long si_band;
/* Событие, связанное с сигналом SIGPOLL */
```

В заголовочном файле <**signal.h**> определены именованные константы, предназначенные для работы с полем ***si_code***, значения которого могут быть как специфичными для конкретного сигнала, так и универсальными. К числу универсальных кодов относятся:

- **SI_USER** - Сигнал послан функцией ***kill()***.
- **SI_QUEUE** - Сигнал послан функцией ***sigqueue()***.
- **SI_TIMER** - Сигнал сгенерирован в результате срабатывания таймера, установленного функцией ***timer_settime()***.
- **SI_ASYNCIO** - Сигнал вызван завершением асинхронной операции ввода/вывода.
- **SI_MESGQ** - Сигнал вызван приходом сообщения в пустую очередь сообщений.

Из кодов, специфичных для конкретных сигналов, мы упомянем лишь несколько, чтобы дать представление о степени детализации диагностики, предусмотренной стандартом POSIX-2001.

Из имени константы должно быть ясно, к какому сигналу она относится:

- **ILL_ILLOPC** - Некорректный код операции.
- **ILL_COPROC** - Ошибка сопроцессора.
- **FPE_INTDIV** - Целочисленное деление на нуль.
- **FPE_FLTOVF** - Переполнение при выполнении операции вещественной арифметики.
- **FPE_FLTSUB** - Индекс вне диапазона.
- **SEGV_MAPERR** - Адрес не отображен на объект.
- **BUS_ADRALN** - Некорректное выравнивание адреса.
- **BUS_ADRERR** - Несуществующий физический адрес.

- **TRAP_BRKPT** - Процесс достиг точки прерывания.
- **TRAP_TRACE** - Срабатывание трассировки процесса.
- **CLD_EXITED** - Завершение порожденного процесса.
- **CLD_STOPPED** - Остановка порожденного процесса.
- **POLL_PRI** - Поступили высокоприоритетные данные.

Рассмотрим описание функции обработки сигналов sigaction():

- Если аргумент **act** отличен от ***NULL***, он указывает на структуру, специфицирующую действия, которые будут ассоциированы с сигналом ***sig***.
- По адресу **oact** (если он не ***NULL***) возвращаются сведения о прежних действиях.
- Если значение **act** есть ***NULL***, обработка сигнала остается неизменной; подобный вызов можно использовать для опроса способа обработки сигналов.

Следующие флаги в поле **sa_flags** влияют на поведение сигнала ***sig***:

- **SA_NOCLDSTOP** - Не генерировать сигнал SIGCHLD при остановке или продолжении порожденного процесса. Значение аргумента sig должно равняться SIGCHLD.
- **SA_RESETHAND** - При входе в функцию обработки сигнала sig установить подразумеваемую реакцию SIG_DFL и очистить флаг SA_SIGINFO.
- **SA_SIGINFO** - Если этот флаг не установлен и определена функция обработки сигнала sig, она вызывается с одним целочисленным аргументом - номером сигнала. Соответственно, в приложении следует использовать поле sa_handler структуры sigaction. При установленном флаге SA_SIGINFO функция обработки вызывается с двумя дополнительными аргументами, как **void func (int sig, siginfo_t *info, void *context);** второй аргумент указывает на данные, поясняющие причину генерации сигнала, а третий может быть преобразован к указателю на тип ucontext_t - контекст процесса, прерванного доставкой сигнала. В этом случае приложение должно использовать поле sa_sigaction и поля структуры типа siginfo_t. В частности, если значение si_code неположительно, сигнал был сгенерирован процессом с идентификатором si_pid и реальным идентификатором пользователя si_uid.
- **SA_NODEFER** - По умолчанию обрабатываемый сигнал добавляется к маске сигналов процесса при входе в функцию обработки; флаг **SA_NODEFER** предписывает не делать этого, если только sig не фигурирует явным образом в sa_mask.

Опросить и изменить способ обработки сигналов можно и на уровне командного интерпретатора, посредством специальной встроенной команды **trap**:

trap [действие условие ...]

Аргумент "условие" может задаваться как ***EXIT*** (завершение командного интерпретатора) или как имя доставленного сигнала (без префикса ***SIG***). При задании аргумента "действие" минус обозначает подразумеваемую реакцию, пустая цепочка ("") - игнорирование. Если в качестве действия задана команда, то при наступлении условия она обрабатывается как ***eval*** действие.

Команда **trap** без аргументов выдает на стандартный вывод список команд, ассоциированных с каждым из условий. Выдача имеет формат, пригодный для восстановления способа обработки сигналов.

```
save_traps=$(trap)
 . . .
eval "$save_traps"
```

Например, обеспечить выполнение утилиты **logout** из домашнего каталога пользователя во время завершения командного интерпретатора можно с помощью команды:

```
      trap '$HOME/logout' EXIT
```

При перенаправлении вывода в файл, приходится считаться с возможностью возникновения ошибок, специфичных для каналов. Чтобы защитить от них процедуры начальной загрузки, в ОС Linux применяются связки из игнорирования и последующего восстановления подразумеваемой реакции на сигнал **SIGPIPE**:

```
      trap "" PIPE
      echo "$INITLOG_ARGS -n $0 -s \"$1\" -e 1" >&21
      trap - PIPE
```

Технически работа с наборами сигналов выполняется посредством функций ***sigemptyset(...)*** и ***sigfillset(...)***, которые инициализируют набор, делая его, соответственно, пустым или "полным".

Функции:
sigaddset() добавляет сигнал ***signo*** к набору ***set;***
sigdelset() удаляет сигнал,
sigismember() проверяет вхождение в набор.

Обычно признаком завершения является нулевой результат, в случае ошибки возвращается -***1***.

Только ***sigismember()*** выдает ***1***, если сигнал ***signo*** входит в набор ***set***.

```
#include <signal.h>

int sigemptyset (sigset_t *set);
int sigfillset (sigset_t *set);
int sigaddset (sigset_t *set, int signo);
int sigdelset (sigset_t *set, int signo);
int sigismember (const sigset_t *set, int signo);
```

Функция **sigprocmask(...)** предназначена для опроса и/или изменения маски сигналов процесса, определяющей набор блокируемых сигналов:

```
#include <signal.h>
int sigprocmask (int how, const sigset_t
   *restrict set, sigset_t *restrict oset);
```

Если аргумент **set** отличен от ***NULL***, он указывает на набор, используемый для изменения текущей маски сигналов.
Аргумент **how** определяет способ изменения; он может принимать одно из трех значений:

- **SIG_BLOCK** - результирующая маска получается при объединении текущей и заданной аргументом ***set***;
- **SIG_SETMASK** - результирующая маска устанавливается равной ***set***;
- **SIG_UNBLOCK** - маска ***set*** вычитается из текущей.

По адресу **oset**, если он не ***NULL***, возвращается прежняя маска. Если значение ***set*** есть ***NULL***, набор блокируемых сигналов остается неизменным; подобный вызов можно использовать для опроса текущей маски сигналов процесса.

Если к моменту завершения ***sigprocmask()*** будут существовать ждущие не блокированные сигналы, по крайней мере один из них должен быть доставлен до возврата из ***sigprocmask()***.

Замечание 6.
Нельзя блокировать сигналы, не допускающие игнорирования.

Функция **sigpending(...)** позволяет выяснить набор блокированных сигналов, ожидающих доставки вызывающему процессу (потоку управления):

```
#include <signal.h>
int sigpending (sigset_t *set);
```

Дождаться появления подобного сигнала можно с помощью функции ***sigwait()***:

```
#include <signal.h>
int sigwait (const sigset_t *restrict set,
             int *restrict sig);
```

Функция **sigwait()** выбирает ждущий сигнал из заданного набора (он должен включать только блокированные сигналы), удаляет его из системного набора ждущих сигналов и помещает его номер по адресу, заданному аргументом ***sig***. Если в момент вызова ***sigwait()*** нужного сигнала нет, процесс (***поток управления***) приостанавливается до появления такового.

Стандарт POSIX-2001 не специфицирует воздействие функции ***sigwait()*** на обработку сигналов, включенных в набор ***set***.

Чтобы дождаться доставки обрабатываемого или терминирующего процесс сигнала, можно воспользоваться функцией ***pause()***:

```
#include <unistd.h>
int pause (void);
```

Функция **pause()** может ждать доставки сигнала неопределенно долго. Возврат из ***pause()*** осуществляется после возврата из функции обработки сигнала (результат при этом равен ***-1***). Если прием сигнала вызывает завершение процесса, возврата из функции ***pause()***, естественно, не происходит.

Замечание 7.
Несмотря на внешнюю простоту, использование функции ***pause()*** сопряжено с рядом тонкостей. При наивном подходе сначала проверяют некоторое условие, связанное с сигналом, и, если оно не выполнено (сигнал отсутствует), вызывают ***pause()***. К сожалению, сигнал может быть доставлен в промежутке между проверкой и вызовом ***pause()***, что нарушает логику работы процесса и способно привести к его зависанию.

Решить подобную проблему позволяет функция **sigsuspend()**, в сочетании с рассмотренной выше функцией ***sigprocmask()***:

```
#include <signal.h>
int sigsuspend (const sigset_t *sigmask);
```

Функция **sigsuspend()** заменяет текущую маску сигналов вызывающего процесса на набор, заданный аргументом ***sigmask***, а затем переходит в состояние ожидания, аналогичное функции ***pause()***. После возврата из ***sigsuspend()***, если таковой произойдет, восстанавливается прежняя маска сигналов.

Обычно парой функций ***sigprocmask()*** и ***sigsuspend()*** обрамляют критические интервалы. Перед входом в критический интервал посредством ***sigprocmask()*** блокируют некоторые сигналы, а на выходе вызывают ***sigsuspend()*** с маской, которую возвратила ***sigprocmask()***, восстанавливая, тем самым, набор блокированных сигналов и дожидаясь их доставки.

В качестве примера использования описанных выше функций работы с сигналами, рассмотрим упрощенную реализацию функции ***abort()***:

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>

void abort (void) {
 struct sigaction sact;
 sigset_t sset;

 /* Вытолкнем буфера */
 (void) fflush (NULL);

 /* Снимем блокировку сигнала SIGABRT */
 if ((sigemptyset (&sset) == 0) && (sigaddset (&sset, SIGABRT) == 0)) {
   (void) sigprocmask (SIG_UNBLOCK, &sset, (sigset_t *) NULL);
 }

 /* Пошлем себе сигнал SIGABRT.                      */
 /* Возможно, его перехватит функция обработки,      */
 /* и тогда вызывающий процесс может не завершиться */
 raise (SIGABRT);

 /* Установим подразумеваемую реакцию на сигнал SIGABRT */
 sact.sa_handler = SIG_DFL;
 sigfillset (&sact.sa_mask);
 sact.sa_flags = 0;
 (void) sigaction (SIGABRT, &sact, NULL);

 /* Снова пошлем себе сигнал SIGABRT */
 raise (SIGABRT);

 /* Если сигнал снова не помог, попробуем еще одно средство завершения */
 _exit (127);
}

int main (void) {
 printf ("Перед вызовом abort()\n");
 abort ();
 printf ("После вызова abort()\n");
 return 0;
}
```

В качестве нюанса, характерного для работы с сигналами, отметим, что до первого обращения к ***raise()*** нельзя закрыть потоки (можно только вытолкнуть буфера), поскольку функция обработки сигнала ***SIGABRT***, возможно, осуществляет вывод.

Еще одним примером использования механизма сигналов может служить упрощенная реализация функции ***sleep()***, предназначенной для "засыпания" на заданное число секунд.

```
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
#include <time.h>

/* Функция обработки сигнала SIGALRM.                  */
/* Она ничего не делает, но игнорировать сигнал нельзя */
static void signal_handler (int sig) {
/* В демонстрационных целях распечатаем номер обрабатываемого сигнала */
 printf ("Принят сигнал %d\n", sig);
}

/* Функция для "засыпания" на заданное число секунд        */
/* Результат равен разности между заказанной и фактической */
/* продолжительностью "сна"                                 */
unsigned int sleep (unsigned int seconds) {
 time_t before, after;
 unsigned int slept;
 sigset_t set, oset;
 struct sigaction act, oact;

 if (seconds == 0) {
   return 0;
 }

 /* Установим будильник на заданное время,    */
 /* но перед этим блокируем сигнал SIGALRM    */
 /* и зададим свою функцию обработки для него */
 if ((sigemptyset (&set) < 0) || (sigaddset (&set, SIGALRM) < 0) ||
     sigprocmask (SIG_BLOCK, &set, &oset)) {
   return seconds;
 }

 act.sa_handler = signal_handler;
 act.sa_flags = 0;
 act.sa_mask = oset;
 if (sigaction (SIGALRM, &act, &oact) < 0) {
   return seconds;
 }

 before = time ((time_t *) NULL);
 (void) alarm (seconds);

 /* Как атомарное действие восстановим старую маску сигналов */
 /* (в надежде, что она не блокирует SIGALRM)                */
 /* и станем ждать доставки обрабатываемого сигнала          */
 (void) sigsuspend (&oset);
 /* сигнал доставлен и обработан */

 after = time ((time_t *) NULL);

 /* Восстановим прежний способ обработки сигнала SIGALRM */
 (void) sigaction (SIGALRM, &oact, (struct sigaction *) NULL);

 /* Восстановим первоначальную маску сигналов */
 (void) sigprocmask (SIG_SETMASK, &oset, (sigset_t *) NULL);

 return ((slept = after - before) > seconds ? 0 : (seconds - slept));
}

int main (void) {
 struct sigaction act;
```

```
 /* В демонстрационных целях установим обработку прерывания с клавиатуры */
 act.sa_handler = signal_handler;
 (void) sigemptyset (&act.sa_mask);
 act.sa_flags = 0;
 (void) sigaction (SIGINT, &act, (struct sigaction *) NULL);

 printf ("Заснем на 10 секунд\n");
 printf ("Проснулись, не доспав %d секунд\n", sleep (10));
 return (0);
}
```

Следует обратить внимание на применение функции ***sigsuspend()***, которая реализует (неделимую) транзакцию снятия блокировки сигналов и перехода в режим ожидания.

Также следует отметить, что по умолчанию при входе в функцию обработки к маске добавляется принятый сигнал для защиты от бесконечной рекурсии.

Наконец, если происходит возврат из функции ***sigsuspend()*** (после возврата из функции обработки), то автоматически восстанавливается маска сигналов, существовавшая до вызова ***sigsuspend()***. В данном случае в этой маске блокирован сигнал **SIGALRM**, и потому можно спокойно менять способ его обработки.

Вызвать "недосыпание" приведенной программы можно выполнить, послав ей сигнал **SIGALRM**, например, посредством команды:

```
kill -s SIGALRM идентификатор_процесса
```

или **SIGINT** - путем нажатия на клавиатуре терминала комбинации клавиш ***Ctrl+C***.

4. РАЗРАБОТКА ПРОГРАММЫ, РЕАГИРУЮЩЕЙ НА СИГНАЛЫ

Используя полученные в процессе обучения навыки программирования, а также теоретический и методический материал, изложенный в разделах 1 - 3 данного учебно-методического пособия, студент должен:

- **написать**, отладить и запустить ***программу на языке С***, принимающую и обрабатывающую поступающие к ней сигналы;
- **продемонстрировать** и исследовать эту программу, посылая ей, с помощью системной утилиты **kill**, различные сигналы;
- **подготовить** и представить на проверку преподавателю письменный отчет о выполнении лабораторной работы, содержащий текст программы, результаты демонстрирующие исследовательскую часть работы и выводы о технологии и возможностях использования сигналов при разработке прикладных программ.

Методические указания по проведению данной лабораторной работы требуют от магистра последовательное выполнение следующих этапов:

1. Изучение раздела 1 учебно-методического пособия с целью повышения уровня своей общекультурной компетенции при разработке системного и прикладного программного обеспечения; студент должен осознать важность и необходимость ***стандарта POSIX*** для целей мобильности (переносимости) ПО.
2. Изучение раздела 2 обеспечивает студента теоретическими знаниями и инструментом создания процессов, как основных объектов ОС, которые подвержены воздействию сигналов и способны реагировать на эти сигналы.
3. Изучение раздела 3 обеспечивает студента теоретическими знаниями и инструментом создания сигналов, их посылки, приема и обработки в программах на языке С; приведены также конкретные примеры использования данного инструмента.
4. Запустив на ОС УПК АСУ инструментальную среду разработки EclipseC, студент должен создать новый проект, написать, отладить запустить на выполнение прикладную программу, согласно указанному выше заданию.
5. С помощью утилиты **kill** и информационного материала, который предоставлен утилитой ***man***, студент должен провести исследование, посылая разработанной программе различные сигналы, а также, наблюдая и фиксируя реакцию программы на эти сигналы.
6. По результатам исследования студент подготавливает письменный отчет, при необходимости, иллюстрируя его результатами исследования.
7. Подготовив отчет, студент докладывает о выполнении задания преподавателю и следует его указаниям по сдаче выполненной работы.

5. КОНТРОЛЬ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ №2

Обязательным требованием по контролю знаний и умений студента является наличие письменного отчета по выполненной лабораторной работе №2.

Отчет оформляется как второй раздел общего отчета по дисциплине «Архитектура вычислительных комплексов» и находится в определенном преподавателем месте учебного комплекса кафедры АСУ.

Отчет по лабораторной работе №2 ***должен содержать***, как минимум три подраздела: постановка задачи, описание работы, выводы.

Цель подготовки и сдачи отчета — формирование у студентов общекультурных компетенций по оформлению и представлению научных и исследовательских документов.

Порядок контроля навыков студента и сдача отчета проводятся в следующей последовательности.

1. Студент сообщает преподавателю о завершении выполнения задания и готовности студента к контролю навыков и сдаче отчета.
2. Преподаватель убеждается в наличии отчета и необходимых элементов его оформления, а затем делает замечания по устранению недостатков отчета или уточняет время и условия контроля навыков и приема результатов работы.
3. В процессе сдачи отчета, студент демонстрирует результаты работы и отвечает на вопросы преподавателя, при необходимости, устраняет ошибки или недоработки, отмеченные преподавателем.
4. Приняв отчет студента, преподаватель сообщает об этом факте устно, при необходимости оценивает результаты работы, а также определяет дальнейший процесс обучения студента.

ЛИТЕРАТУРА

1. Галатенко В.А. Прогаммирование в стандарте POSIX. Интернет ресурс: http://www.intuit.ru/department/se/pposix/8/.

Учебное издание

Резник Виталий Григорьевич

АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ КОМПЛЕКСОВ

Методические рекомендации для выполнения лабораторной работы №2 по дисциплине «Архитектура вычислительных комплексов» для студентов уровня основной образовательной программы магистратура направления подготовки 010400.68 «Прикладная математика и информатика» профиля «Математическое и программное обеспечение вычислительных комплексов и компьютерных сетей».

Учебно-методическое пособие

Усл. печ. л. . Тираж ___. Заказ .

Томский государственный университет

систем управления и радиоэлектроники

634050, г. Томск, пр. Ленина, 40