ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

Государственное образовательное учреждение высшего профессионального образования

Санкт-Петербургский государственный университет аэрокосмического приборостроения

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к выполнению лабораторной работы

"Процессы и сигналы ОС Linux"

Прокофьева Т.Л.

Санкт-Петербург 2019

Лабораторная работа № 3 посвящена изучению процессов и сигналов ОС UNIX как средства организации межпроцессного взаимодействия, приобретению знаний и навыков написания программ работы со взаимодействующими процессами. Приведены системные вызовы для работы с процессами и сигналами и примеры программ.

# Содержание работы

1. Изучить правила использования системных вызовов **signal()**, **waitpid ()**, **kill (), raise (), pause(), alarm (), sigaction ()**.
2. Ознакомиться с системными вызовами **sigemptyset(), sigaddset (), sigprocmas ()**.
3. Изучить средства работы с сигналами и каналами в ОС LINUX.
4. Ознакомиться с заданием к лабораторной работе.
5. Для указанного варианта составить программу на языке Си, реализующую задание. Обработчик сигналов должен быть реентерабельным
6. Отладить и оттестировать составленную программу, используя инструментарий ОС LINUX.
7. Защитить лабораторную работу, ответив на контрольные вопросы.

# 1. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Работу ОС UNIX можно представить в виде функционирования множества взаимосвязанных процессов.

Процесс – это задание в ходе его выполнения. Он представляет собой исполняемый образ программы, включающий отображение в памяти исполняемого файла, полученного в ходе компиляции, стек, код и данные библиотек, а также ряд структур данных ядра, необходимых для управления процессом. Выполнение процесса заключается в точном следовании замкнутому набору инструкций, который не передает управление набору инструкций другого процесса; он считывает и записывает информацию в раздел данных и в стек, но ему недоступны данные и стеки других процессов.

ОС UNIX является многозадачной системой, поэтому в ней параллельно выполняется множество процессов, их выполнение планируется ядром.

Несколько процессов могут быть экземплярами одной программы. Процессы взаимодействуют с другими процессами и с вычислительными ресурсами только посредством обращений к операционной системе, которая эффективно распределяет системные ресурсы между активными процессами.

## 1.1. Выполнение процесса

Выполнение процесса осуществляется ядром системы. Подсистема управления процессами отвечает за синхронизацию процессов, взаимодействие процессов, распределение памяти и планирование выполнения процессов.

С практической точки зрения процесс в системе UNIX является объектом, создаваемым в результате выполнения системного вызова fork(). Каждый процесс, за исключением нулевого, порождается в результате запуска другим процессом операции fork(). Процесс, запустивший операцию fork(), называется родительским, а вновь созданный процесс – порожденным. Каждый процесс имеет одного родителя, но может породить много процессов. Ядро системы идентифицирует каждый процесс по его номеру, который называется идентификатором процесса (PID). Нулевой процесс является особенным процессом, который создается "вручную" в результате загрузки системы.

Процесс 1, известный под именем init, является предком любого другого процесса в системе и связан с каждым процессом. Пользователь, транслируя исходный текст программы, создает исполняемый файл, который состоит из нескольких частей:

• набора "заголовков", описывающих атрибуты файла; • текста программы;

• представления на машинном языке данных, имеющих начальные значения при запуске программы на выполнение, и указания на то, сколько пространства памяти ядро системы выделит под неинициализированные данные, так называемые bss ("block started by symbol" – "блок, начинающийся с символа");

• некоторых других секций, таких как информация символических таблиц.

Ядро загружает исполняемый файл в память при выполнении системной операции exec, при этом загруженный процесс состоит по меньшей мере из трех частей, так называемых областей: текста, данных и стека. Области текста и данных соответствуют секциям текста и bss-данных исполняемого файла, а область стека создается автоматически и ее размер динамически устанавливается ядром системы во время выполнения.

Процесс в системе UNIX может выполняться в двух режимах – режиме ядра или режиме задачи. В режиме задачи процесс выполняет инструкции прикладной программы, системные структуры данных ему недоступны.

Когда процесс выполняет специальную инструкцию (системный вызов), он переключается в режим ядра. Каждой системной операции соответствует точка входа в библиотеке системных операций; библиотека системных операций написана на языке ассемблера и включает специальные команды прерывания, которые, выполняясь, порождают "прерывание", вызывающее переключение аппаратуры в режим ядра. Процесс ищет в библиотеке точку входа, соответствующую отдельной системной операции, подобно тому, как он вызывает любую из функций.

Соответственно и образ процесса состоит из двух частей: данных режима ядра и режима задачи. В режиме ядра образ процесса включает сегменты кода, данных, библиотек и других структур, к которым он может получить непосредственный доступ. Образ процесса в режиме ядра состоит из структур данных, недоступных процессу в режиме задачи (например, состояния регистров, таблицы для отображения памяти и т.п.).

Каждому процессу соответствует точка входа (запись) в таблице процессов ядра, кроме того, каждому процессу выделяется часть оперативной памяти, отведенной под задачи пользователей. Таблица процессов включает в себя указатели на промежуточную таблицу областей процессов, точки входа в которую служат в качестве указателей на собственно таблицу областей.

Областью называется непрерывная зона адресного пространства, выделяемая процессу для размещения текста, данных и стека. Точки входа в таблицу областей описывают атрибуты области, как например, хранятся ли в области текст программы или данные, закрытая ли эта область или же совместно используемая, и где конкретно в памяти размещается содержимое области. Внешний уровень косвенной адресации (через промежуточную таблицу областей, используемых процессами, к собственно таблице областей) позволяет независимым процессам совместно использовать области.

Когда процесс запускает системную операцию exec, ядро системы выделяет области оперативной памяти (ОП) под новый процесс и освобождает области памяти вызывающего процесса. Если процесс запускает операцию fork, ядро удваивает размер адресного пространства процесса, позволяя новому и старому процессам совместно использовать области ОП, когда это возможно, и, с другой стороны, производя физическое копирование. Если процесс запускает операцию exit, ядро освобождает области, которые использовались процессом. На Рис.1 изображены информационные структуры, связанные с запуском процесса позволяющие процессам разделять области ОП.

Таблица процессов ссылается на промежуточную таблицу областей, используемых процессом, в которой содержатся указатели на записи в собственно таблице областей, соответствующие областям для текста, данных и стека процесса.

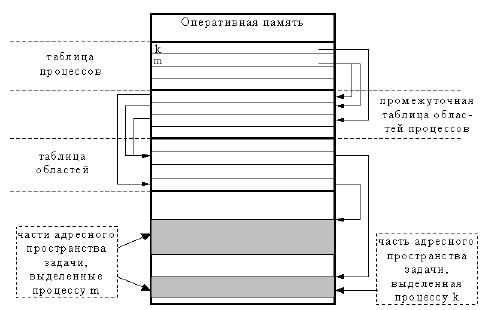


Рис. 1. Системные структуры для разделения областей памяти между процессами

UNIX является системой разделения времени. Это означает, что каждому процессу вычислительные ресурсы выделяются на ограниченный промежуток времени, после чего они предоставляются другому процессу. Максимальный временной интервал, на который процесс может захватить процессор, называется временным квантом. Таким образом, создается иллюзия того, что процессы работают одновременно, хотя в действительности на однопроцессорной машине одновременно может выполняться только один процесс.

Процессы предъявляют различные требования к системе с точки зрения их планирования и общей производительности. Можно выделить три основных класса приложений:

• интерактивные приложения (командные интерпретаторы, редакторы и проч.). Большую часть времени они проводят в ожидании пользовательского ввода, но для них критично время отклика (реакции системы на ввод данных).

• фоновые приложения (не требующие вмешательства пользователя). Основной показатель эффективности для них – минимальное суммарное время выполнения в системе.

• приложения реального времени. Они обычно привязаны к таймеру и требуют гарантированного времени совершения той или иной операции и времени отклика.

В основе планирования выполнения процессов лежат два понятия: квант времени и приоритет. Каждый процесс имеет два атрибута приоритета: текущий, на основании которого осуществляется планирование, и относительный, который задается при порождении процесса и влияет на текущий. Номера приоритетов разбиваются на несколько групп: для процессов в режиме задачи, в режиме ядра, для процессов реального времени (группы указаны в соответствии с повышением приоритета).

Обработчик прерываний от таймера, в частности, проверяет истечение временного кванта для процессов и пересчитывает приоритеты процессов: чем дольше процесс занимает процессор, тем ниже (в пределах группы процессов) становится его приоритет.

Выполнение процесса может быть прервано:

1. планировщиком процессов по истечении временного кванта или в том случае, если в очереди готовых к исполнению процессов есть процесс с более высоким приоритетом.

2. ядром системы, если процесс ожидает недоступного ресурса или окончания длительной операции ввода/вывода.

В режиме ядра приоритет процесса повышается для того, чтобы его выполнение не могло быть прервано, так как это может привести к нарушению целостности структур данных ядра. Таким образом, выполнение системных вызовов осуществляется в непрерывном режиме (за исключением некоторых аппаратных прерываний).

## 1.2. Контекст процесса

Под контекстом процесса понимается вся информация, необходимая для описания процесса. Контекст процесса состоит из нескольких частей:

• адресное пространство процесса в режиме задачи (код, данные и стек процесса, а также разделяемая память и данные динамических библиотек);

• управляющая информация (структуры proc и user - запись таблицы процессов и дополнительная информация соответственно);

• окружение процесса (системные переменные, например, домашний каталог, имя пользователя и др.);

• аппаратный контекст (значения используемых машинных регистров).

Контекстом процесса является его состояние, определяемое текстом, значениями глобальных переменных пользователя и информационными структурами, значениями используемых машинных регистров, значениями, хранимыми в таблице процессов и в адресном пространстве задачи, а также содержимым стеков задачи и ядра, относящихся к процессу. Текст операций системы и ее глобальные информационные структуры совместно используются всеми процессами, но не являются составной частью контекста процесса.

Принято говорить, что при запуске процесса система исполняется в контексте процесса. Когда ядро системы решает запустить другой процесс, оно выполняет переключение контекста с тем, чтобы система исполнялась в контексте другого процесса. Ядро осуществляет переключение контекста только при определенных условиях (см. п.1.3). Выполняя переключение контекста, ядро сохраняет информацию, достаточную для того, чтобы позднее переключиться вновь на прерванный процесс и возобновить его выполнение.

Аналогичным образом, при переходе из режима задачи в режим ядра, ядро системы сохраняет информацию, достаточную для того, чтобы позднее вернуться в режим задачи и продолжить выполнение с прерванного места.

Однако, переход из режима задачи в режим ядра является сменой режима, но не переключением контекста. Ядро меняет режим выполнения с режима задачи на режим ядра и наоборот, оставаясь в контексте одного процесса.

Ядро обрабатывает прерывания в контексте прерванного процесса, пусть даже оно и не вызывало никакого прерывания. Прерванный процесс мог при этом выполняться как в режиме задачи, так и в режиме ядра. Ядро сохраняет информацию, достаточную для того, чтобы можно было позже возобновить выполнение прерванного процесса, и обрабатывает прерывание в режиме ядра.

Ядро не порождает и не планирует порождение какого-то особого процесса по обработке прерываний.

## 1.3. Состояния процесса

Рис. 2. Состояния процесса и переходы между нимиВремя жизни процесса можно разделить на несколько состояний, каждое из которых имеет определенные характеристики, описывающие процесс. На рис. 2 показаны состояния, в которых процесс может находиться с момента создания до завершения выполнения:

1. Процесс выполняется в режиме задачи. Процессор выполняет прикладные инструкции данного процесса.

2. Процесс выполняется в режиме ядра. Процессор выполняет системные инструкции ядра ОС от имени процесса.

3. Процесс не выполняется, но готов к выполнению, находится в очереди готовых к исполнению процессов и ждет, когда планировщик выберет его. Естественно, в этом состоянии может находиться много процессов, и алгоритм планирования устанавливает, какой из процессов будет выполняться следующим.

4. Процесс приостановлен ("спит"). Процесс "впадает в сон", когда он не может больше продолжать выполнение, например, когда ждет завершения ввода-вывода или освобождения занятого ресурса.

5. Процесс возвращается из режима ядра в режим задачи, но ядро прерывает его и производит переключение контекста для запуска более высокоприоритетного процесса.

6. Процесс только что создан вызовом fork и находится в переходном состоянии: он существует, но не готов к запуску и не находится в состоянии сна.

7. Процесс выполнил системный вызов exit и перешел в состояние "зомби".

Как такового процесса не существует, но осталась запись с системной таблице процессов, содержащая код возврата и временную статистику, доступные для родительского процесса.

Поскольку процессор в каждый момент времени выполняет только один процесс, в состояниях 1 и 2 на однопроцессорной машине может находиться самое большее один процесс.

Процессы непрерывно переходят из состояния в состояние в соответствии с определенными правилами. Диаграмма переходов (Рис.2) представляет собой ориентированный граф. Вершинами этого графа являются состояния, в которые может перейти процесс, а дугами – события, которые стали причинами перехода процесса из одного состояния в другое. Переход между двумя состояниями разрешен, если существует дуга из начального состояния в конечное. Несколько дуг может выходить из одного состояния, однако процесс переходит только по одной из них в зависимости от того, какое событие произошло в системе.

В режиме разделения времени может выполняться одновременно несколько процессов, и все они могут одновременно работать в режиме ядра. Если им разрешить свободно выполняться в режиме ядра, то они могут испортить глобальные информационные структуры, принадлежащие ядру. Запрещая произвольное переключение контекста и управляя возникновением событий, ядро защищает свою целостность. Ядро разрешает переключение контекста только тогда, когда процесс переходит из состояния "запуск в режиме ядра" в состояние "сна в памяти". Процессы, запущенные в режиме ядра, не могут быть выгружены другими процессами; поэтому иногда говорят, что ядро невыгружаемо, при этом процессы, находящиеся в режиме задачи, могут выгружаться системой. Ядро поддерживает целостность своих информационных структур, поскольку оно невыгружаемо, таким образом решая проблему "взаимного исключения" – обеспечения того, что критические секции программы выполняются в каждый момент времени в рамках самое большее одного процесса.

# 2. СИГНАЛЫ

Сигнал - это программное средство, с помощью которого может быть прервано функционирование процесса ОС UNIX. Сигналы сообщают процессам о возникновении асинхронных событий. Механизм сигналов позволяет процессам реагировать на различные события, которые могут происходить в ходе работы процесса внутри него самого или во внешней среде.

Сигналы описаны в файле <signal.h>, каждому из них ставится в соответствие мнемоническое обозначение. Количество и семантика сигналов зависят от версии ОС UNIX.

UNIX System V сигналы имеют номера от 1 до 19:

#define NSIG 20

#define SIGHUP 1 /\* разрыв связи \*/

#define SIGINT 2 /\* прерывание \*/

#define SIGQUIT 3 /\* аварийный выход \*/

#define SIGILL 4 /\* неверная машинная инструкция \*/

#define SIGTRAP 5 /\* прерывание-ловушка \*/

#define SIGIOT 6 /\* прерывание ввода-вывода \*/

#define SIGEMT 7 /\* программное прерывание EMT \*/

#define SIGFPE 8 /\* авария при выполнении операции с плавающей точкой \*/

#define SIGKILL 9 /\* уничтожение процесса \*/

#define SIGBUS 10 /\* ошибка шины \*/

#define SIGSEGV 11 /\* нарушение сегментации \*/

#define SIGSYS 12 /\* ошибка выполнения системного вызова \*/

#define SIGPIPE 13 /\* запись в канал есть, чтения нет \*/

#define SIGALRM 14 /\* прерывание от таймера \*/

#define SIGTERM 15 /\* программ. сигнал завершения от kill \*/

#define SIGUSR1 16 /\* определяется пользователем \*/

#define SIGUSR2 17 /\* определяется пользователем \*/

#define SIGCLD 18 /\* процесс-потомок завершился \*/

#define SIGPWR 19 /\* авария питания \*/

#define SIG\_DFL (int(\*)())0 /\* все установки “по умолчанию” \*/

#define SIG\_IGN (int(\*)())1 /\* игнорировать этот сигнал \*/

## 2.1. Причины возникновения сигналов

В версии System V UNIX возникновение сигналов можно классифицировать следующим образом:

• введение пользователем управляющего символа с терминала всем процессам, ассоциированным с данным терминалом (SIGINT, SIGQUIT, SIGHUP);

• возникновение аварийной ситуации при функционировании пользовательского процесса (SIGILL, SIGTRAP, SIGFPE, SIGBUS, SIGSEGV, SIGSYS, SIGPIPE);

• возникновение непредусмотренного или не поддающегося идентификации события (SIGTERM, SIGCLD, SIGPWR);

• возникновение некоторого заранее описанного события (SIGALRM).

Посылка сигналов производится процессами - друг другу, с помощью функции kill, - или ядром. Для каждого процесса определен бинарный вектор, длина которого равна количеству сигналов в системе. При получении процессом сигнала I соответствующий i-й разряд этого вектора становится равным 1.

Каждому сигналу соответствует адрес функции, которая будет вызвана для обработки данного сигнала.

## 2.2. Обработка сигналов

Ядро обрабатывает сигналы в контексте того процесса, который получает их, поэтому чтобы обработать сигналы, нужно запустить процесс.Существует три способа обработки сигналов:

• реакция по умолчанию,

• игнорирование сигнала,

• выполнение особой (пользовательской) функции по его получении.

Реакцией по умолчанию со стороны процесса, исполняемого в режиме ядра, обычно является вызов функции exit(), т.е. завершение процесса. Но вместе с тем реакция процесса на принимаемый сигнал зависит от того, как сам процесс определил свое поведение в случае приема данного сигнала: процесс может проигнорировать сигнал, вызвать на выполнение другой процесс и т.д. При этом способ обработки сигналов одного типа не влияет на обработку сигналов других типов.

Обрабатывая сигнал, ядро определяет тип сигнала и очищает (гасит) разряд в записи таблицы процессов, соответствующий данному типу сигнала и установленный в момент получения сигнала процессом. Таким образом, когда процесс получает любой неигнорируемый им сигнал (за исключением SIGILL и SIGTRAP), ОС UNIX автоматически восстанавливает реакцию “по умолчанию” на всякое последующее получение этого сигнала.

Замечание 1: Если необходима многократная обработка одного и того же сигнала, процесс должен каждый раз осуществлять системный вызов signal для установления требуемой реакции на данный сигнал.

Замечание 2: Процесс не в состоянии узнать, сколько однотипных сигналов им было получено. В том случае, если процесс не успевает обработать все поступившие сигналы, происходит потеря информации.

Замечание 3: При системном вызове fork() все установленные реакции на сигналы наследуется порожденным процессом. При системном вызове exec() сохраняются реакции только для тех сигналов, которые игнорировались или обрабатывались по умолчанию. Получение любого сигнала, который до вызова exec() обрабатывался пользователем, приведет к завершению процесса.

Если функции обработки сигнала присвоено значение по умолчанию, ядро в отдельных случаях перед завершением процесса сбрасывает на внешний носитель (дампирует) образ процесса в памяти. Дампирование удобно для программистов тем, что позволяет установить причину завершения процесса и посредством этого вести отладку программ. Ядро дампирует состояние памяти при поступлении сигналов, которые сообщают о каких-нибудь ошибках в выполнении процессов, как например, попытка исполнения запрещенной команды или обращение к адресу, находящемуся за пределами виртуального адресного пространства процесса. Ядро не дампирует состояние памяти, если сигнал не связан с программной ошибкой, за исключением внешнего сигнала о выходе (quit), обычно вызываемого одновременным нажатием клавиш Ctrl+|.

Если процесс получает сигнал SIGINT, который было решено игнорировать (signal(SIGINT,SIG\_IGN)), выполнение процесса продолжается так, словно сигнала и не было. Поскольку ядро не сбрасывает значение соответствующего поля, свидетельствующего о необходимости игнорирования сигнала данного типа, то когда сигнал поступит вновь, процесс опять не обратит на него внимание.

В том случае, если процесс получает сигнал, реагирование на который установлено системным вызовом signal, сразу по возвращении процесса в режим задачи выполняется заранее условленное действие, описанное в вызове signal. После выполнения функции обработки сигнала управление будет передано на то место в программе пользователя, где было произведено обращение к системной функции или произошло прерывание.

Если во время исполнения системной функции приходит сигнал, а процесс приостановлен с допускающим прерывания приоритетом, этот сигнал побуждает процесс выйти из приостанова, вернуться в режим задачи и вызвать функцию обработки сигнала. Когда функция обработки сигнала завершается, процесс выходит из системной функции с ошибкой, сообщающей о прерывании ее выполнения. После этого пользователь может запустить системную функцию повторно.

## 2.3. Группы процессов

Несмотря на то, что в системе UNIX процессы идентифицируются уникальным кодом (PID), системе иногда приходится использовать для идентификации процессов номер “группы”, в которую они входят. Например, процессы, имеющие общего предка в лице регистрационного интерпретатора shell, взаимосвязаны, и поэтому когда пользователь нажимает клавиши “delete” или “break”, или когда терминальная линия “зависает”, все эти процессы получают соответствующие сигналы. Ядро использует код группы процессов для идентификации группы взаимосвязанных процессов, которые при наступлении определенных событий должны получать общий сигнал. Код группы запоминается в таблице процессов. При выполнении функции fork процесс-потомок наследует код группы своего родителя.

Для того, чтобы образовать новую группу процессов, следует воспользоваться системной функцией setpgrp:

grp = setpgrp()

где grp - новый код группы процессов, равный его коду идентификации процесса, осуществившего вызов setpgrp().

# 3. СИСТЕМНЫЕ ВЫЗОВЫ

## 3.1. Системные вызовы для работы с процессами

Рассмотрим системные вызовы, используемые при работе с процессами в ОС UNIX и описанные в библиотеке <fcntl.h>.

Системный вызов fork создает новый процесс, копируя вызывающий, вызов exit завершает выполнение процесса, wait дает возможность родительскому процессу синхронизировать свое продолжение с завершением порожденного процесса, а sleep приостанавливает на определенное время выполнение процесса. Системный вызов exec дает процессу возможность запускать на выполнение другую программу.

**FORK** Создание нового процесса:

int fork(void)

pid = fork();

В ходе выполнения функции ядро производит следующую последовательность действий:

1. Отводит место в таблице процессов под новый процесс.

2. Присваивает порождаемому процессу уникальный код идентификации.

3. Создает логическую копию контекста родительского процесса. Поскольку те или иные составляющие процесса, такие как область команд, могут разделяться другими процессами, ядро может иногда вместо копирования области в новый физический участок памяти просто увеличить значение счетчика ссылок на область.

4. Увеличивает значения счетчиков числа файлов, связанных с процессом как в таблице файлов, так и в таблице индексов.

5. Возвращает родительскому процессу код идентификации порожденного процесса, а порожденному процессу – 0.

В результате выполнения функции fork пользовательский контекст и того, и другого процессов совпадает во всем, кроме возвращаемого значения функции fork. Если процесс не может быть порожден, функция возвращает отрицательный код ошибки.

Процесс, вызывающий функцию fork, называется родительским (процесс-предок), вновь создаваемый процесс называется порожденным (процесс-потомок). Процесс-потомок всегда имеет более высокий приоритет, чем процесс-предок, так как приоритет процесса является самым высоким в момент порождения и уменьшается по мере нахождения в состоянии выполнения.

Нулевой процесс, возникающий внутри ядра при загрузке системы, является единственным процессом, не создаваемым с помощью функции fork.

**EXIT** Завершение выполнения процесса:

void exit(int status)

где status – значение, возвращаемое функцией родительскому процессу.

Процессы могут вызывать функцию exit как в явном, так и в неявном виде (по окончании выполнения программы функция exit вызывается автоматически с параметром 0). Также ядро может вызывать функцию exit по своей инициативе, если процесс завершается по сигналу. В этом случае значение параметра status равно номеру сигнала.

Выполнение вызова exit приводит к "прекращению существования" процесса, освобождению ресурсов и ликвидации контекста. Система не накладывает никакого ограничения на продолжительность выполнения процесса.**WAIT** Ожидание завершения выполнения процесса-потомка:

int wait(int \*stat)

pid = wait(stat\_addr);

где pid – значение кода идентификации (PID) завершившегося потомка, stat\_addr – адрес переменной целого типа, в которую будет помещено возвращаемое функцией exit значение.

С помощью этой функции процесс синхронизирует продолжение своего выполнения с моментом завершения потомка. Ядро ведет поиск потомков процесса, прекративших существование, и в случае их отсутствия возвращает ошибку. Если потомок, прекративший существование, обнаружен, ядро передает его код идентификации и значение, возвращаемое через параметр функции exit, процессу, вызвавшему функцию wait. Таким образом, через параметр функции exit (status) завершающийся процесс может передавать различные значения, кодирующие информацию о причине завершения процесса, однако на практике этот параметр используется по назначению довольно редко. Если процесс, выполняющий функцию wait, имеет потомков, продолжающих существование, он приостанавливается до получения ожидаемого сигнала. Ядро не возобновляет по своей инициативе процесс, приостановившийся с помощью функции wait: такой процесс может возобновиться только в случае получения сигнала о "гибели потомка".

**SLEEP** Приостанов работы процесса на определенное время:

void sleep(unsigned seconds)

где seconds – количество секунд, на которое требуется приостановить работу процесса.

Сначала ядро повышает приоритет работы процесса так, чтобы заблокировать все прерывания, которые могли бы (путем создания конкуренции) помешать работе с очередями приостановленных процессов, и запоминает старый приоритет, чтобы восстановить его, когда выполнение процесса будет возобновлено. Процесс получает пометку “приостановленного”, адрес приостанова и приоритет запоминаются в таблице процессов, а процесс помещается в хеш-очередь приостановленных процессов. При этом в простейшем случае (когда приостанов не допускает прерываний) процесс выполняет переключение контекста и благополучно "засыпает". На время приостанова процесс помещается в хеш-очередь приостановленных процессов.

По истечение указанного периода времени приостановленный процесс "пробуждается" и ядро осуществляет его запуск. Нельзя гарантировать, что приостановленный процесс сразу возобновит свою работу: он может быть выгружен на время приостанова и тогда требуется его подкачка в память; в это время на выполнении может находится процесс с более высоким приоритетом или процесс, не допускающий прерываний (например, находящийся в критическом интервале) и т.д.

Параметр seconds устанавливает минимальный интервал, в течение которого процесс будет приостановлен, а реальное время приостанова в любом случае будет несколько больше, хотя бы за счет времени, необходимого для переключения процессов.

**EXEC** Запуск программы.

Системный вызов exec осуществляется несколькими библиотечными функциями – execl, execv, execle и др.. Приведем формат одной из них:

int execv(char \*path, char \*argv[])

res = execv(path, argv);

где path – имя исполняемого файла, argv – указатель на массив параметров, которые передаются вызываемой программе. Этот массив аналогичен параметру argv командной строки функции main.

Список argv должен содержать минимум два параметра: первый – имя программы, подлежащей выполнению (отображается в argv[0] функции main новой программы), второй – NULL (завершающий список аргументов).

Системный вызов exec дает возможность процессу запускать другую программу, при этом соответствующий этой программе исполняемый файл будет располагаться в пространстве памяти процесса. Содержимое пользовательского контекста после вызова функции становится недоступным, за исключением передаваемых функции параметров, которые переписываются ядром из старого адресного пространства в новое.

Вызов exec возвращает 0 при успешном завершении и -1 при аварийном. В последнем случае управление возвращается в вызывающую программу.

В качестве примера использования этого вызова можно привести работу командного интерпретатора shell: при выполнении команды он сначала порождает свою копию (fork), а затем запускает соответствующую указанной команде программу с помощью exec.

## 3.2. Системные вызовы для работы с файлами

Стандартная библиотека Си содержит набор функций для работы с файлами. Эти функции описаны в стандарте ANSI. Отметим, что файловый ввод-вывод не является частью языка Си, и ANSI-функции - не единственное средство ввода-вывода. Так, в операционной системе Unix более популярен другой набор функций ввода-вывода, который можно использовать не только для работы с файлами, но и для обмена по сети. В C++ часто используются библиотеки классов для ввода-вывода. Тем не менее, функции ANSI-библиотеки поддерживаются всеми Си-компиляторами, и потому программы, применяющие их, легко переносятся с одной платформы на другую. Прототипы функций ввода-вывода и используемые для этого типы данных описаны в стандартном заголовочном файле «stdio.h»

Открытие файла: функция fopen

Для доступа к файлу применяется тип данных FILE. Это структурный тип, имя которого задано с помощью оператора typedef в стандартном заголовочном файле "stdio.h". Программисту не нужно знать, как устроена структура типа файл: ее устройство может быть системно зависимым, поэтому в целях переносимости программ обращаться явно к полям струтуры FILE запрещено.

Тип данных "указатель на структуру FILE используется в программах как черный ящик: функция открытия файла возвращает этот указатель в случае успеха, и в дальнейшем все файловые функции применяют его для доступа к файлу.

Прототип функции открытия файла выглядит следующим образом:

FILE \*fopen(const char \*path, const char \*mode);

Здесь path - путь к файлу (например, имя файла или абсолютный путь к файлу), mode - режим открытия файла. Строка mode может содержать несколько букв.

Буква " r " (от слова read) означает, что файл открывается для чтения (файл должен существовать). Буква " w " (от слова write) означает запись в файл, при этом старое содержимое файла теряется, а в случае отсутствия файла он создается. Буква " a " (от слова append) означает запись в конец существующего файла или создание нового файла, если файл не существует.

В некоторых операционных системах имеются различия в работе с текстовыми и бинарными файлами (к таким системам относятся MS DOS и MS Windows; в системе Unix различий между текстовыми и бинарными файлами нет). В таких системах при открытии бинарного файла к строке mode следует добавлять букву " b " (от слова binary), а при открытии текстового файла -- букву " t " (от слова text). Кроме того, при открытии можно разрешить выполнять как операции чтения, так и записи; для этого используется символ + (плюс).

Порядок букв в строке mode следующий: сначала идет одна из букв " r ", " w ", " a ", затем в произвольном порядке могут идти символы " b ", " t ", " + ". Буквы " b " и " t " можно использовать, даже если в операционной системе нет различий между бинарными и текстовыми файлами, в этом случае они просто игнорируются.

Значения символов в строке mode сведены в следующую таблицу:

r Открыть существующий файл на чтение

w Открыть файл на запись. Старое содержимое файла теряется, в случае отсутствия файла он создаётся.

a Открыть файл на запись. Если файл существует, то запись производится в его конец.

t Открыть текстовый файл.

b Открыть бинарный файл.

+ Разрешить и чтение, и запись.Несколько примеров открытия файлов:

FILE \*f, \*g, \*h;

. . .

// 1. Открыть текстовый файл "abcd.txt" для чтения

f = fopen("abcd.txt", "rt");

// 2. Открыть бинарный файл "tmp.dat" для чтения и записи

g = fopen("tmp.dat", "wb+");

// 3. Открыть текстовый файл "abcd.log" для дописывания в конец файла

h = fopen("abcd.log", "at");

Обратите внимание, что во втором случае мы используем обычную косую черту / для разделения директорий, хотя в системах MS DOS и MS Windows для этого принято использовать обратную косую черту \. Дело в том, что в операционной системе Unix и в языке Си, который является для нее родным, символ \ используется в качестве экранирующего символа, т.е. для защиты следующего за ним символа от интерпретации как специального. Поэтому во всех строковых константах Си обратную косую черту надо повторять дважды, как это и сделано в третьем примере. Впрочем, стандартная библиотека Си позволяет в именах файлов использовать нормальную косую черту вместо обратной; эта возможность была использована во втором примере.

В случае удачи функция fopen открытия файла возвращает ненулевой указатель на структуру типа FILE, описывающую параметры открытого файла. Этот указатель надо затем использовать во всех файловых операциях. В случае неудачи (например, при попытке открыть на чтение несуществующий файл) возвращается нулевой указатель. При этом глобальная системная переменная errno, описанная в стандартном заголовочном файле "errno.h, содержит численный код ошибки. В случае неудачи при открытии файла этот код можно распечатать, чтобы получить дополнительную информацию:

#include <stdio.h>

#include <errno.h>

. . .

FILE \*f = fopen("filnam.txt", "rt");

if (f == NULL) {

printf(

"Ошибка открытия файла с кодом %d\n",

errno

);

. . .

}

Диагностика ошибок: функция perror

Использовать переменную errno для печати кода ошибки не очень удобно, поскольку необходимо иметь под рукой таблицу возможных кодов ошибок и их значений. В стандартной библиотеке Си существует более удобная функция perror, которая печатает системное сообщение о последней ошибке вместо ее кода. Печать производится на английском языке, но есть возможность добавить к системному сообщению любой текст, который указывается в качестве единственного аргумента функции perror. Например, предыдущий фрагмент переписывается следующим образом:

#include <stdio.h>

. . .

FILE \*f = fopen("filnam.txt", "rt");

if (f == 0) {

perror("Не могу открыть файл на чтение");

. . .

}

Функция perror печатает сначала пользовательское сообщение об ошибке, затем после двоеточия системное сообщение. Например, при выполнении приведенного фрагмента в случае ошибки из-за отсутствия файла будет напечатано

Не могу открыть файл на чтение: No such file or directory

Функции бинарного чтения и записи fread и fwrite

После того как файл открыт, можно читать информацию из файла или записывать информацию в файл. Рассмотрим сначала функции бинарного чтения и записи fread и fwrite. Они называются бинарными потому, что не выполняют никакого преобразования информации при вводе или выводе (с одним небольшим исключением при работе с текстовыми файлами, которое будет рассмотрено ниже): информация хранится в файле как последовательность байтов ровно в том виде, в котором она хранится в памяти компьютера.

Функция чтения fread имеет следующий прототип:

size\_t fread(

char \*buffer, // Массив для чтения данных

size\_t elemSize, // Размер одного элемента

size\_t numElems, // Число элементов для чтения

FILE \*f // Указатель на структуру FILE

);

Здесь size\_t определен как беззнаковый целый тип в системных заголовочных файлах. Функция пытается прочесть numElems элементов из файла, который задается указателем f на структуру FILE, размер каждого элемента равен elemSize. Функция возвращает реальное число прочитанных элементов, которое может быть меньше, чем numElems, в случае конца файла или ошибки чтения.

Указатель f должен быть возвращен функцией fopen в результате успешного открытия файла. Пример использования функции fread:FILE \*f;

double buff[100];

size\_t res;

f = fopen("tmp.dat", "rb"); // Открываем файл

if (f == 0) { // При ошибке открытия файла

// Напечатать сообщение об ошибке

perror("Не могу открыть файл для чтения");

exit(1); // завершить работу с кодом 1

}

// Пытаемся прочесть 100 вещественных чисел из файла

res = fread(buff, sizeof(double), 100, f);

// res равно реальному количеству прочитанных чисел

В этом примере файл " tmp.dat " открывается на чтение как бинарный, из него читается 100 вещественных чисел размером 8 байт каждое. Функция fread возвращает реальное количество прочитанных чисел, которое меньше или равно, чем 100.

Функция fread читает информацию в виде потока байтов и в неизменном виде помещает ее в память. Следует различать текстовое представление чисел и их бинарное представление! В приведенном выше примере числа в файле должны быть записаны в бинарном виде, а не в виде текста. Для текстового ввода чисел надо использовать функции ввода по формату, которые будут рассмотрены ниже.

Внимание! Открытие файла как текстового с помощью функции fopen, например,

FILE \*f = fopen("tmp.dat", "rt");

вовсе не означает, что числа при вводе с помощью функции fopen будут преобразовываться из текстовой формы в бинарную! Из этого следует только то, что в операционных системах, в которых строки текстовых файлов разделяются парами символами " \r\n " (они имеют названия CR и LF - возврат каретки и продергивание бумаги, Carriage Return и Line Feed), при вводе такие пары символов заменяются на один символ " \n " (продергивание бумаги). Обратно, при выводе символ " \n " заменяется на пару " \r\n ". Такими операционными системами являются MS DOS и MS Windows. В системе Unix строки разделяются одним символом " \n " (отсюда проистекает обозначение " \n ", которое расшифровывается как new line). Таким образом, внутреннее представление текста всегда соответствует системе Unix, а внешнее - реально используемой операционной системе. Отметим также, что создатели операционной системы компьютеров Apple Macintosh выбрали, чтобы жизнь не казалась скучной, третий, отличный от двух предыдущих, вариант: текстовые строки разделяются одним символом " \r " возврат каретки!

Такое представление текстовых файлов восходит к тем уже далеким временам, когда еще не было компьютерных мониторов и для просмотра текста использовались электрифицированные пишущие машинки или посимвольные принтеры. Текстовый файл фактически представлял собой программу печати на пишущей машинке и, таким образом, содержал команды возврата каретки и продергивания бумаги в конце каждой строки.

Функция бинарной записи в файл fwrite аналогична функции чтения fread. Она имеет следующий прототип:

size\_t fwrite(

char \*buffer, // Массив записываемых данных

size\_t elemSize, // Размер одного элемента

size\_t numElems, // Число записываемых элементов

FILE \*f // Указатель на структуру FILE

);

Функция возвращает число реально записанных элементов, которое может быть меньше, чем numElems, если при записи произошла ошибка - например, не хватило свободного пространства на диске. Пример использования функции fwrite:

FILE \*f;

double buff[100];

size\_t num;

. . .

f = fopen("tmp.res", "wb"); // Открываем файл "tmp.res"

if (f == 0) { // При ошибке открытия файла напечатать сообщение об ошибке

perror("Не могу открыть файл для записи");

exit(1); // завершить работу программы с кодом 1

}

// Записываем 100 вещественных чисел в файл

num = fwrite(buff, sizeof(double), 100, f);

// В случае успеха num == 100

Закрытие файла: функция fclose

По окончании работы с файлом его надо обязательно закрыть. Система обычно запрещает полный доступ к файлу до тех пор, пока он не закрыт. (Например, в нормальном режиме система запрещает одновременную запись в файл для двух разных программ.) Кроме того, информация реально записывается полностью в файл лишь в момент его закрытия. До этого она может содержаться в оперативной памяти (в так называемой файловой кеш-памяти), что при выполнении многочисленных операций записи и чтения значительно ускоряет работу программы.

Для закрытия файла используется функция fclose с прототипом

int fclose(FILE \*f);

В случае успеха функция fclose возвращает ноль, при ошибке – отрицательное значение (точнее, константу конец файла EOF, определенную в системных заголовочных файлах как минус единица). При ошибке можно воспользоваться функцией perror, чтобы напечатать причину ошибки. Отметим, что ошибка при закрытии файла - явление очень редкое (чего не скажешь в отношении открытия файла), так что анализировать значение, возвращаемое функцией fclose, в общем-то, не обязательно. Пример использования функции fclose:

FILE \*f;

f = fopen("tmp.res", "wb"); // Открываем файл "tmp.res"

if (f == 0) { // При ошибке открытия файла

// Напечатать сообщение об ошибке

perror("Не могу открыть файл для записи");

exit(1); // завершить работу программы с кодом 1

}

. . .

// Закрываем файл

if (fclose(f) < 0) {

// Напечатать сообщение об ошибке

perror("Ошибка при закрытии файла");

}

## 3.3. Системные вызовы для работы с неименоваными каналами

Все операционные системы семейства UNIX предоставляют простейшее средство одно-направленной пересылки данных между процессами — анонимные каналы (часто их называют просто каналами — pipe). Канал создаётся системным вызовом pipe.

#include <unistd.h>

int pipe(int filedes[2]);

При успешном завершении системный вызов возвращает 0, а при ошибке — -1, и переменная errno устанавливается в код ошибки. Неуспешное завершение pipe скорее всего означает, что переполнилась таблица открытых файлов у процесса либо у всей системы. Системному вызову передаётся массив из двух элементов, в который этот системный вызов записывает номера двух файловых дескрипторов. Два файловых дескриптора связаны друг с другом. Данные записываются в файловый дескриптор filedes[1]. Записанные данные можно прочитать из файлового дескриптора filedes[0]. Вообще, filedes[1] ссылается на начало канала, filedes[0] — на конец канала (если мы предполагаем, что данные «текут» от начала к концу).

В результате создания нового процесса с помощью fork, либо копирований файлового дескриптора с помощью dup, dup2, на начало или конец канала может ссылаться несколько файловых дескрипторов у разных процессов.

Начало канала (в которую ведётся запись) считается закрытым, когда закрыты все файловые дескрипторы, которые ссылаются на него. Аналогично, конец канала (чтение из канала) считается закрытым, когда закрыты все файловые дескрипторы, которые ссылаются на него. Чтобы установить момент, когда какой-то конец канала закрывается, ядро подсчитывает количество ссылок на канал. После открытия канала читать и писать в него можно с помощью системных вызовов read и write.

## 3.3. Системные вызовы для работы с сигналами

Рассмотрим наиболее часто используемые системные вызовы при работе с сигналами в ОС UNIX, описанные в библиотеке <signal.h>.

**KILL** Посылка всем или некоторым процессам любого сигнала:

int kill(pid\_t pid, int sig)

где sig - номер сигнала, pid - идентификатор процесса, определяющий группу родственных процессов, которым будет послан данный сигнал:

• если pid - положительное целое число, ядро посылает сигнал процессу с идентификатором pid.

• если значение pid равно 0, сигнал посылается всем процессам, входящим в одну группу с процессом, вызвавшим функцию kill.

• если значение pid равно -1, сигнал посылается всем процессам, у которых реальный код идентификации пользователя совпадает с тем, под которым исполняется процесс, вызвавший функцию kill. Если процесс, пославший сигнал, исполняется под кодом идентификации суперпользователя, сигнал рассылается всем процессам, кроме процессов с идентификаторами 0 и 1.

• если pid - отрицательное целое число, но не -1, сигнал посылается всем процессам, входящим в группу с номером, равным абсолютному значению pid.

Вызов kill возвращает 0 при успешном завершении и -1 при аварийном (например, спецификация несуществующего в ОС UNIX сигнала или несуществующего процесса).

Во всех случаях, если процесс, пославший сигнал, исполняется под кодом идентификации пользователя, не являющегося суперпользователем, или если коды идентификации пользователя (реальный и исполнительный) у этого процесса не совпадают с соответствующими кодами процесса, принимающего сигнал, kill завершается неудачно.

Посылка сигнала может сопровождать возникновение любого события.

Сигналы SIGUSR1, SIGUSR2 и SIGKILL могут быть посланы только с помощью системного вызова kill.

**RAISE** Послать сигнал самому себе.

Вызов

raise(sig);

эквивалентен вызову

kill(getpid(), sig);

Естественно, каждый из рассмотренных системных вызовов возвращает ноль в случае успешного выполнения и ненулевое значение, если произошла какая-нибудь ошибка.

**SIGNAL** Позволяет процессу самому определить свою реакцию на получение того или иного сигнала:

void (\*signal(int signum, void (\*handler)(int)))(int)После определения реакции на сигнал signal при получении процессом этого сигнала будет автоматически вызываться функция handler(тип ее - void handler(int)), которая, естественно, должна быть описана или объявлена прежде, чем будет осуществлен системный вызов signal.

При многократной обработке одного и того же сигнала, процесс должен каждый раз осуществлять системный вызов signal для установления требуемой реакции на данный сигнал. Использование констант SIG\_DFL и SIG\_IGN позволяет упростить реализацию двух часто встречающихся реакций процесса на сигнал:

signal(SIGINT,SIG\_IGN) игнорирование сигнала;

signal(SIGINT,SIG\_DFL) восстановление стандартной реакции на сигнал.

Аргументом функции-обработчика является целое число – номер обрабатываемого сигнала. Значение его устанавливается ядром.

Основным недостатком системного вызова signal() является его низкая надежность.

Во многих вариантах операционной системы UNIX установленная при его помощи обработка сигнала пользовательской функцией выполняется только один раз, после чего автоматически восстанавливается реакция на сигнал по умолчанию. Для постоянной пользовательской обработки сигнала необходимо каждый раз заново устанавливать реакцию на сигнал прямо внутри функции-обработчика.

В системных вызовах и пользовательских программах могут существовать критические участки, на которых процессу недопустимо отвлекаться на обработку сигналов. Мы можем выставить на этих участках реакцию <игнорировать сигнал> с последующим восстановлением предыдущей реакции, но если сигнал все-таки возникнет на критическом участке, то информация о его возникновении будет безвозвратно потеряна.

Наконец, последний недостаток связан с невозможностью определения количества сигналов одного и того же типа, поступивших процессу, пока он находился в состоянии готовность. Сигналы одного типа в очередь не ставятся! Процесс может узнать о том, что сигнал или сигналы определенного типа были ему переданы, но не может определить их количество. Этот недостаток мы можем проиллюстрировать, слегка изменив программу с асинхронным получением информации о статусе завершившихся процессов, рассмотренную нами ранее в разделе <Изучение особенностей получения терминальных сигналов текущей и фоновой группой процессов>. Пусть в новой программе 13-14-6.c процесс-родитель порождает в цикле пять новых процессов, каждый из которых сразу же завершается со своим собственным кодом, после чего уходит в бесконечный цикл.

**SIGACTION** позволяет процессу узнать или указать реакцию на сигнал.

Прототип системного вызова sigaction():

#include <signal.h>

int sigaction (int signum, struct sigaction \*action, struct sigaction \*oldaction );

где signum - номер сигнала, action - реакция процесса на сигнал, oldaction- предыдущая реакция процесса на сигнал. Аргумент signum задает сигнал (см. список сигналов, определенных в файле <signal.h>).

Структура sigaction , используемая в описании реакции процесса на сигнал, имеет следующие поля:

struct signal() {

\_\_sighandler\_t sa\_handler;

sigset\_t sa\_mask;

int sa\_flags;

};

sa\_handler - указатель на обработчик сигнала, причем обработчик должен быть объявлен следующим образом:

void signal\_handler(int signo);

где единственный параметр - номер сигнала, который попал в обработчик. sa\_handler также может быть равным SIG\_IGN - сигнал игнорируется процессом, и SIG\_DFL - сигнал вызывает действие по умолчанию, например, прерывание процесса.

sa\_mask - дополнительное множество сигналов, которые должны блокироваться при вызове заданного в этой же структуре обработчика сигнала.

int sa \_ flags - специальные флаги, управляющие передачей сигнала.

Если переменная action не является нулевым указателем, она указывает структуру типа sigaction, которая специфицирует реакцию на сигнал. Если переменная oldaction не является нулевым указателем, предыдущая реакция на сигнал запоминается в структуре, которую указывает oldaction.

Если переменная action - нулевой указатель, то реакция на сигнал не изменяется, таким образом sigaction может использоваться для получения информации о текущей установленной реакции на сигнал.

Поле sa \_ handler структуры sigaction указывает реакцию на данный сигнал. Если это поле определяет указатель на перехватывающую функцию, поле sa \_ mask указывает множество сигналов, которые должны быть добавлены к сигнальной маске процесса перед входом в перехватывающую функцию. В добавляемое множество сигналов не могут входить сигналы SIGKILL и SIGSTOP.

Поле sa \_ flags может быть использовано для управления передачей сигнала. В этом поле может быть установлен флаг SA \_ NOCLDSTOP, определенный в файле <signal.h>. Если signum - это сигнал SIGCHLD и флаг SA \_ NOCLDSTOP обнулен, сигнал SIGCHLD посылается процессу каждый раз, когда какой-либо из порожденных им процессов останавливается. Если signum - это сигнал SIGCLD и флаг SA \_ NOCLDSTOP равен единице, то сигнал SIGCHLD не генерируется.

Когда сигнал перехватывается функцией, указанной в sigaction, вычисляется новая сигнальная маска, которая действует до завершения перехватывающей функции или до ближайшего вызова sigprocmask или sigsuspend. Эта маска получается объединением текущей сигнальной маски, множества из поля sa \_ mask структуры, связанной с данной реакцией на сигнал, и используемого сигнала. После нормального завершения перехватывающей функции восстанавливается прежняя сигнальная маска.

Реакция, установленная на определенный сигнал, сохраняется до тех пор, пока она не будет изменена явно функцией sigaction или неявно функцией семейства exec (функции семейства exec заменяют текущий образ процесса новым образом и передают управление содержащейся в нем программе).

Если предыдущая реакция на сигнал была задана функцией signal, значения, возвращаемые в поля структуры по указателю oldaction, не определены. В частности, поле oldaction->sa \_ handler не обязано совпадать с реакцией, установленной функцией signal. Однако, если указатель на эту структуру задается в качестве аргумента action функции sigaction, будет установлена та же реакция на сигнал, что была в свое время установлена функцией signal.

Возвращаемое значение:

sigaction при успешном завершении возвращает значение 0. В случае ошибки системный вызов sigaction возвращает значение -1, и новая реакция не устанавливается.

**WAIT, WAITPID** - ожидает завершения процесса.

#include <sys/types.h>

#include <sys/wait.h>

pid\_t wait(int \*status);

pid\_t waitpid(pid\_t pid, int \*status, int options);

Функция wait приостанавливает выполнение текущего процесса до тех пор, пока 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 "зомби" ("zombie")), то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются. Функция waitpid приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс, указанный в параметре pid, не завершит выполнение, или пока не появится сигнал, который либо завершает текущий процесс либо требует вызвать функцию-обработчик. Если указанный дочерний процесс к моменту вызова функции уже завершился (так называемый "зомби"), то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются. Параметр 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 следующих констант:

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 сообщено. Статус для отслеживаемых остановленных подпроцессов также обеспечивается без этой опции.

Если status не равен NULL, то функции wait и waitpid сохраняют информацию о статусе в переменной, на которую указывает status. Этот статус можно проверить с помощью нижеследующих макросов (они принимают в качестве аргумента буфер (типа int), --- а не указатель на буфер!):

WIFEXITED(status) не равно нулю, если дочерний процесс успешно завершился.

WEXITSTATUS(status) возвращает восемь младших битов значения, которое вернул завершившийся дочерний процесс. Эти биты могли быть установлены в аргументе функции exit() или в аргументе оператора return функции main(). Этот макрос можно использовать, только если WIFEXITED вернул ненулевое значение.

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 (см. ptrace(2)).

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

Некоторые версии Unix (например Linux, Solaris, но не AIX, SunOS) также определяют макрос WCOREDUMP(status) для проверки того, не вызвал ли дочерний процесс ошибку ядра. Используйте это только в структуре #ifdef WCOREDUMP ... #endif.

ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

Возвращает идентификатор дочернего процесса, который завершил выполнение, или ноль, если использовался WNOHANG и ни один дочерний процесс пока еще недоступен, или -1 в случае ошибки (в этом случае переменной errno присваивается соответствующее значение).

ЗАМЕЧАНИЯ ПО LINUX

В ядре Linux задачи, управляемые ядром, по внутреннему устройству не отличаются от процесса. Задача (thread) -- это простой процесс, который создан специфичным для Linux системным вызовом clone(2); другие процедуры, типа портируемой pthread\_create(3), также реализованы с помощью clone(2).

До Linux 2.4, задачи были частным случаем процесса, и последовательность одной группы задач не могла ожидать дочерний процесс или другую группу задач, даже если воследняя принадлежит к этой-же группе задач. Однако, POSIX предопределяет такие особенности, и с Linux 2.4 группа задач может, и по умолчанию будет ожидать дочерний процесс другой группы задач в этой-же группе задач.

Следующие специфичные для Linux параметры существуют для использования с дочерними процессами и clone(2).

\_\_WCLONE только ожидает дочерние процессы. Если отменяется, то ожидает только дочерних неклонированных процессов. ("клонированным" дочерним процессом является подпроцесс, не отправляющий сигналов, или выдающий сигналы, отличающиеся от SIGCHLD своему родителю до завершения.) Эта опция игнорируется, если определено \_\_WALL.

\_\_WALL (с Linux 2.4) Ожидает все дочерние процессы, независимо от их типа ("клон" или "не-клон").

\_\_WNOTHREAD (С Linux 2.4) Не ожидать дочерние процессы или остальные задачи в идентичной группе задач. Было параметром по умолчанию до Linux 2.4.

**PAUSE** Приостанавливает функционирование процесса до получения им некоторого сигнала:

void pause()

Этот системный вызов не имеет параметров. Работа процесса возобновляется после получения им любого сигнала, кроме тех, которые игнорируются этим процессом.

**ALARM** Посылка процессу сигнала побудки SIGALARM:

unsigned alarm(unsigned secs)

Этим системным вызовом процесс информирует ядро ОС о том, что ядро должно послать этому процессу сигнал побудки через secs секунд. Вызов alarm возвращает число секунд, заданное при предыдущем осуществлении этого системного вызова.

Если secs равно 0, то специфицированная ранее посылка процессу сигнала SIGALARM будет отменена.

Маска сигналов процесса

Добавление сигналов в структуру sigset\_t sa\_mask, ее очистка и т.п. осуществляются при помощи набора функций sigemptyset(), sigfillset(), sigaddset(), sigdelset(). Первые две функции принимают один параметр - указатель на структуру sigset\_t. Эти функции очищают и заполняют всеми возможными сигналами структуру sigset\_t соответственно.

Последние две функции, соответственно, добавляют и удаляют один определенный сигнал из структуры и имеют по два параметра. Их первый параметр - указатель на структуру sigset\_t, а второй - номер сигнала.

Все рассмотренные выше функции возвращают 0 при успешном завершении и число, не равное нулю, - при ошибке.

Кроме того, существует еще одна функция, проверяющая, находится ли указанный сигнал в указанном наборе - sigismember(). Ее параметры совпадают с параметрами sigaddset(). Функция возвращает 1, если сигнал находится в наборе, 0 - если не находится, и отрицательное число - при возникшей ошибке.

Помимо всего прочего, мы можем задать список сигналов, доставка которых процессу будет заблокирована. Это выполняется при помощи функции sigprocmask(int how, const sigset\_t \* set, sigset\_t \* oldset).

Первый ее параметр описывает то, что должно выполняться:

1. SIG\_BLOCK - сигналы из набора set блокируются;

2. SIG\_UNBLOCK - сигналы из набора set разблокируются;

3. SIG\_SETMASK - сигналы из набора set блокируются, остальные разблокируются.

Второй параметр является указателем на тот самый набор, сигналы из которого блокируются/разблокируются. Если он равен NULL, то значение первого параметра игнорируется системным вызовом.

Третий параметр - указатель на уже используемую маску сигналов; его можно поставить в NULL, если эти данные не нужны.

Для получения списка ожидающих сигналов можно использовать функцию sigpending(), которая принимает единственный параметр - указатель на структуру sigset\_t, куда будет записан набор ожидающих сигналов.

Принципы написания обработчиков сигналов

Одно из самых главных правил написания обработчиков сигналов - обработчик должен быть реентерабельным, т.е. он должен допускать свой повторный вызов, когда процесс уже находится в обработчике. Нужно заботиться о том, чтобы обработчик сигналов не использовал глобальные структуры данных или медленные системные вызовы. Если избежать этого, увы, невозможно, то стоит позаботиться о защите от повторного вызова обработчика во время работы со структурой данных или с системным вызовом. Добиться этого можно, заблокировав на время доставку сигнала, обработчик которого сейчас работает, при помощи системного вызова sigprocmask(). Например, мы имеем обработчик сигнала SIGCHLD, выполняем в нем блокировку так:

void chld\_handler(int signum) {

sigset\_t set;

if (sigemptyset(&set)) {

return;

)

if (sigaddset(&set, SIGCHLD)) {

return;

}

if (sigprocmask(SIG\_BLOCK, &set, NULL)) {

return;

}

/\* делаем здесь что-то важное \*/

if (sigprocmask(SIG\_UNBLOCK, &set, NULL)) {

return;

}

return;

}

Кроме всего перечисленного выше, считается, что обработчик должен быть максимально простым - в идеале он должен выставлять некий флаг и завершаться, а все остальное должна выполнять основная часть программы.

# 4. ПРИМЕРЫ ПРОГРАММ

**Пример** **1**. Порождение процессов. Программа в результате выполнения породит три процесса (процесс-предок 1 и процессы-потомки 2 и 3).

#include <sys/types.h>

#include <fcntl.h>

#include <stdio.h>

void main(void)

{

int pid2, pid3, st;

printf("Process 1, pid = %d:\n", getpid());

pid2 = fork();

if (pid2 == 0) /\* process 2 \*/

{

printf("Process 2, pid = %d:\n", getpid());

pid3 = fork(); if (pid3 == 0) /\* process 3 \*/

{ printf("Process 3, pid = %d:\n", getpid());

sleep(2);

printf("Process 3: end\n");

} /\* process 2 \*/

if (pid3 < 0) printf("Cann't create process 3: error %d\n", pid3);

wait(&st);

printf("Process 2: end\n");

}

else /\* process 1 \*/

{ if (pid2 < 0) printf("Cann't create process 2: error %d\n", pid2);

wait(&st);

printf("Process 1: end\n");

}

exit(0);

}

В соответствии с программой первоначально будет создан процесс 1 (как потомок интерпретатора shell), он сообщит о начале своей работы и породит процесс 2. После этого работа процесса 1 приостановится и начнет выполняться процесс 2 как более приоритетный. Он также сообщит о начале своей работы и породит процесс 3. Далее начнет выполняться процесс 3, он сообщит о начале работы и "заснет". После этого возобновит свое выполнение либо процесс 1, либо процесс 2 в зависимости от величин приоритетов и от того, насколько процессор загружен другими процессами. Так как ни один из процессов не выполняет никакой работы, они, вероятнее всего, успеют завершится до возобновления процесса 3, который в этом случае завершится последним.

Пример 2. Порождение процессов и их синхронизация. Данная программа в результате выполнения породит два процесса причем процесс-предок закончится после процесса-потомка.

#include <sys/types.h>

#include <fcntl.h>

#include <stdio.h>

void main(void)

{

int pid2, pid3, st; /\* proc 1 \*/

printf("Process 1, pid = %d: begin\n", getpid());

pid2 = fork();

if (pid2 < 0) printf("Cann't create process 2: error %d\n", pid2);

else

{

if (pid2 == 0) /\* process 2 \*/

{

printf("Process 2, pid = %d: begin\n", getpid());

sleep(1);

printf("Process 2: end\n");

}

else /\* process 1 \*/

{

wait(&st);

printf("Process 1: end\n");

}

}

exit(0);

}От предыдущей данная программа отличается наличием синхронизации выполнения процессов: процесс-предок ожидает завершения процесса-потомка (функция wait). Сначала выполняется процесс 1, который порождает процесс 2. После начинает выполняться процесс 2, который после выдачи сообщения "заснет". Тогда возобновит выполнение процесс 1, который приостановится до получения сигнала об окончании процесса 2. По истечение указанного периода процесс 2 возобновит свое выполнение, выдаст сообщение и завершится. После этого будет возобновлен процесс 1, который также выдаст сообщение и завершится.

Пример 3. Синхронизация работы процессов с использованием каналов.

Использование неименованных каналов для организации передачи сообщений между процессами.

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

int main(void)

{

int fd[2], nbytes;

pid\_t childpid;

char string[] = "Hello, world!\n";

char readbuffer[80];

pipe(fd);

if((childpid = fork()) == -1)

{

perror("fork");

exit(1);

}

if(childpid == 0)

{

/\* Child process closes up input side of pipe \*/

close(fd[0]);

/\* Send "string" through the output side of pipe \*/

write(fd[1], string, (strlen(string)+1));

exit(0);

}

else

{

/\* Parent process closes up output side of pipe \*/

close(fd[1]);

/\* Read in a string from the pipe \*/

nbytes = read(fd[0], readbuffer, sizeof(readbuffer));

printf("Received string: %s", readbuffer);

}

return(0);

}Пример 4. Синхронизация работы процессов с использованием сигналов.

Программа породит три процесса (процесс-предок 0 и процессы-потомки 1 и 2). Процессы 1 и 2 будут обмениваться сигналами и выдавать соответствующие сообщения на экран, а процесс 0 через определенное количество секунд отправит процессам 1 и 2 сигнал завершения и сам прекратит свое функционирование.

#include <sys/types.h>

#include <fcntl.h>

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#define TIMEOUT 10

int f1(int), f2(int), f3(int);

int pid0, pid1, pid2;

void main(void)

{

setpgrp();

pid0 = getpid();

pid1 = fork();

if (pid1 == 0) /\* process 1 \*/

{

signal(SIGUSR1, f1);

pid1 = getpid();

pid2 = fork();

if (pid2 < 0 ) puts("Fork error");

if (pid2 > 0) for(;;);

else /\* process 2 \*/

{

signal(SIGUSR2, f2);

pid2 = getpid();

kill(pid1,SIGUSR1);

for (;;);

}

}

else /\* process 0 \*/

{

signal(SIGALRM, f3);

alarm(TIMEOUT);

pause();

}

exit(0);

}

int f1(int signum)

{

signal(SIGUSR1, f1);

printf("Process 1 (%d) has got a signal from process 2 (%d)\n",pid1,pid2);

sleep(1);

kill(pid2, SIGUSR2);

return 0;

}

int f2(int signum)

{

signal(SIGUSR2, f2);

printf("Process 2 (%d) has got a signal from process 1 (%d)\n",pid2,pid1);

sleep(1);

kill(pid1, SIGUSR1);return 0;

}

int f3(int signum)

{

printf("End of job - %d\n", pid0);

kill(0, SIGKILL);

return 0;

}

# БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Дансмур М., Дейвис Г. Операционная система UNIX и программирование на языке Си: Пер. с англ. – М.: Радио и связь, 1989. – 192 с.

2. Керниган Б.В., Пайк Р. UNIX – универсальная среда программирования: Пер. с англ. – М.: Финансы и статистика, 1992. – 304 с.

3. А.М.Робачевский, С.А.Немнюгин, О.Л.Стесик. Операционная система UNIX. – СПб.: BHV – Санкт-Петербург, 2010. – 656 с.4. Cross-Referencing Linux (Электронный источник – http://lxr.linux.no) 28. The 5. Linux Kernel Archives ([www.kernel.org](http://www.kernel.org))

6. Материалы сайтов www.linux.org, www.linuxdoc.ru, www.opennet.ru, www.citforum.ru, www.sysinternals.com и многих других.