Версия 0.9 release candidate 1 от 06.02.2014. Возможны незначительные изменения.

Тема 2. Многозадачность в ОС GNU/Linux. Часть 1

Главные вопросы, которые мы обсудим, представлены на СЛАЙДЕ 1.

Отметим предварительно, что программа, которая не взаимодействует с внешним миром, вряд ли может сделать что-то полезное. Вывести сообщение на экран, прочитать данные из файла, установить сетевое соединение – это примеры действий, которые программа не может совершить без помощи ОС. В Linux прикладной программный интерфейс ядра организован через системные вызовы. **Системный вызов** (*syscall*) можно рассматривать как функцию, которую для клиента выполняет ОС. Примеры наиболее популярных системных вызовов мы будем приводить в этом и последующих разделах лекционного курса.

2.1 Многозадачность: процессы

Выполняющийся экземпляр программы называется **процессом**. Если на экране отображаются два терминальных окна, то, скорее всего, одна и та же терминальная программа запущена дважды – просто ей соответствуют два процесса. В каждом окне работает «наш добрый друг» – интерпретатор команд, и это еще один процесс. Когда пользователь вводит команду в интерпретаторе, соответствующая ей программа запускается в виде процесса. По завершении работы программы управление вновь передается процессу интерпретатора.

В некоторых случаях в рамках одного приложения создаются несколько процессов, чтобы оно могло выполнить группу действий одновременно. Это повышает надежность приложения и предоставляет ему возможность использования уже написанных программ.

Большинство рассматриваемых далее функций управления процессами доступно и в других *NIX-систем. Обычно имеет смысл взглянуть на файл *unistd.h*, однако никогда не помешает проверить это в документации.

Каждый процесс в Linux помечается уникальным 16-битным **идентификатором** (PID). Идентификатор назначается последовательно по мере создания процессов.

У всех процессов, кроме *init*, имеется также родительский процесс. *Init* в этой иерархии процессов является корневым элементом. У каждого процесса, за исключением *init* есть **идентификатор предка** (PPID). СЛАЙД 2.

Работая с идентификаторами процессов в программах на C/C++, следует объявлять соответствующие переменные как имеющие тип *pid_t*, который определен наряду с другими в файле *sys/types.h*. Программа может узнать PID своего процесса с помощью системного вызова *getpid()*, а PPID – с помощью системного вызова *getppid()*.

В коде на СЛАЙДЕ 3 приведен листинг программы.

Для вывода информации о процессах обычно используется стандартная программа *ps* (располагается в */bin/ps* или */usr/bin/ps*).

Для уничтожения процесса предназначена команда *kill*, которая в качестве одного из параметров получает *PID* «убиваемого» процесса и сигнал *SIGTERM*.

Как и в любой другой UNIX-подобной системе, в ОС Linux поддерживается целая система **сигналов**, с помощью которых могут взаимодействовать процессы.

Самые распространенные сигналы приведены на СЛАЙДЕ 4.

Сигналы SIGKILL and SIGSTOP нельзя перехватить, заблокировать или игнорировать.

2.2 Создание процессов

Принято говорить о двух способах создания процессов. Первый из них относительно прост, однако применяется сравнительно редко, поскольку неэффективен и связан с риском для безопасности ОС. Второй способ сложнее, но избавлен от недостатков первого.

Функция system

Функция *system()* определена в стандартной библиотеке языка С и позволяет вызывать из программы системную команду таким образом, как если бы она была набрана в командной строке. Фактически, эта функция запускает стандартный интерпретатор команд (*/bin/sh*) и передает ему команду для выполнения. В коде на СЛАЙДЕ 5 вызывается команда *ls -l /*, отображающая содержимое корневого каталога.

Функция *system()* возвращает код завершения указанной команды. Если по каким-то причинам интерпретатор команд не может быть запущен, то возвращается значение 127, а при возникновении других ошибок – -1.

Поскольку эта функция запускает интерпретатор команд, она подвержена тем же ограничениям безопасности, что и системный интерпретатор. Рассчитывать на какую-то конкретную версию оболочки не приходится. Во многих версиях *NIX-систем программа представляет собой символическую ссылку на другой интерпретатор. В Linux – это обычно *bash*, причем в разных дистрибутивах присутствуют разные его версии. Вызов из *system()* программы с привилегиями *root* может иметь неодинаковые последствия в разных системах. Вывод прост: лучше создавать процессы с помощью *fork()* и *exec()*.

Функция fork()

Системами семейства Windows от семейства DOS-систем унаследована группа функций *spawn()*. Они принимают в качестве аргумента имя программы, создают новый экземпляр ее процесса и запускают его. В Linux нет функции, которая сделала бы это за один заход. Вместо этого предлагается с помощью функции *fork()* создать дочерний процесс, являющийся полной копией своего родителя, и семейство функций *exec()*. Они заставляют требуемый процесс перестать быть экземпляром одной программы и превратиться в экземпляр другой программы. Чтобы создать новый процесс, нужно сначала с помощью *fork()* создать копию текущего процесса, а затем с помощью *exec()* преобразовать одну из копий в экземпляр запускаемой программы.

Вызывая *fork()*, программа создает дубликат, называемый *дочерним процессом*. Родитель продолжает выполнять программу с той точки, где была вызвана *fork()*. То же самое делает дочерний процесс.

Очевидно, что родительский и дочерний процессы отличаются друг от друга значениями идентификаторов PID. Выше мы показали, как программа может узнать, в каком процессе она находится. Сама функция *fork()* реализует несколько иной подход. Она возвращает разные значения в родительском и дочернем процессах. Родитель получает идентификатор своего потомка, а дочернему процессу возвращается 0. В ОС нет процессов с нулевым PID, поэтому программа легко разбирается в ситуации.

На СЛАЙДЕ 6 приведен код ветвления программы с помощью *fork()*.

Здесь *true*-ветка оператора *if* выполняется родителем, а *false*-ветка – дочерним.

Функции exec()

Функции *exec()* заменяют программу, выполняющуюся в текущем процессе, другой программой. Когда программа вызывает функцию *exec()*, ее выполнение немедленно прекращается и начинает работу новая программа.

Функции из семейства *exec()* отличаются друг от друга по своим возможностям и способу вызова (СЛАЙД 7).

- Функции *execvp()* и *execlp()* принимают в качестве аргумента имя программы и ищут ее в каталогах, определенных переменной окружения *PATH*. Всем остальным функциям нужно передавать полное имя программы.
- Функции *execv()*, *execvp()* и *execve()* принимают список аргументов в виде массива указателей на строки, оканчивающегося NULL-указателем.
- Функции *execl()*, *execlp()* и *execle()* принимают список аргументов переменного размера.
- Функции *execve()* и *execle()* в качестве дополнительного аргумента принимают массив переменных среды. Он содержит строковые указатели и оканчивается пустым указателем. Каждая строка должна иметь вид «*ПЕРЕМЕННАЯ=Значение*».

Поскольку функции *exec()* заменяют одну программу другой, то они возвращают значения только в том случае, если вызов программы невозможен из-за ошибки.

Список аргументов программы аналогичен аргументам командной строки, указываемым при запуске в интерактивном режиме, т.е. с помощью *argc* и *argv*.

Программа, чей код приведен на СЛАЙДЕ 8, отображает содержимое корневого каталога, однако в отличие от кода со СЛАЙДА 5, команда выполняется напрямую, а не через интерпретатор командной строки.

2.3 Завершение процессов

Обычно процесс завершается одним из двух способов: либо выполняющаяся программа вызывает функцию *exit()*, либо функция *main()* заканчивается. У каждого процесса есть статус (код) завершения – число, возвращаемое родительскому процессу. Этот код передается в качестве аргумента функции *exit()* или возвращается оператором *return*.

Возможно аварийное завершение процесса, в частности в ответ на получение сигнала. Таковыми могут быть сигналы *SIGBUS*, *SIGSEGV*, *SIGFPE*. Есть сигналы, явно запрашивающие прекращение работы процесса, например, *SIGINT*, *SIGTERM* и т.д. Если программа вызывает функцию *abort()*, то она посылает сама себе сигнал *SIGABRT*. Самый «мощный» сигнал – это, конечно же, *SIGKILL*.

Любой сигнал можно послать функцией *kill()*. Ее первым аргументом является PID целевого процесса. Второй аргумент – номер сигнала (стандартному поведению команды *kill* соответствует *SIGTERM*). Завершение дочернего процесса из родительского может быть сделано так (СЛАЙД 9):

```
kill(childPid, SIGTERM);
```

По существующему соглашению код завершения указывает на успешность завершения программы. Нулевое значение говорит о том, что все в порядке, ненулевой код соответствует ошибке. Его конкретное значение может подсказать источник ошибки.

Следует помнить о том, что хотя тип параметра *exit()* и возвращаемый функцией *main()* код, равен *int*, Linux записывает код завершения в младший байт. Значит, код завершения может принимать значения из диапазона от 0 до 127. Коды, значения которых больше 128, интерпретируются особым образом: когда процесс уничтожается после приход сигнала, его код завершения равен 128 плюс номер сигнала.

Ожидание завершения процесса

Если не предпринять никаких дополнительных действий, то дочерний процесс планируется независимо от родительского. Linux – многозадачная ОС, процессы в ней выполняются одновременно, и заранее нельзя сказать, кто завершится раньше, а кто – позже.

Однако бывают ситуации, когда родительский процесс должен дождаться завершения одного, части или всех своих потомков. Это можно сделать с помощью функций семейства *wait()*. Они позволяют получить информацию о завершении процесса.

Это четыре функции, различающиеся объемом возвращаемой информации и способом задания дочернего процесса.

Самая простая функция – *wait()*. Она блокирует процесс до тех пор, пока один из его потомков не завершится (успешно или с ошибкой). Код состояния потомка возвращается через аргумент, представляющим собой указатель на *int*. В нем «зашифрована» информация о потомке. Макрос *WEXITSTATUS* возвращает код завершения. Макрос *WIFEXITED* позволяет узнать, как именно завершился процесс: обычным образом или аварийно (по сигналу). Номер сигнала в последнем случае можно извлечь макросом *WTERMSIG*. СЛАЙД 9.

На СЛАЙДЕ 10 приведен модифицированный код программы со СЛАЙДА 8.

Здесь программа вызывает *wait()*, чтобы дождаться завершения дочернего процесса, в котором выполняется команда *ls*.

Другие функции этого семейства не намного сложнее. Функция *waitpid()* позволяет дождаться завершения конкретного дочернего процесса, а не просто любого. Функция *wait3()* возвращает информацию о статистике использования ЦП завершившимся дочерним процессом. Функция *wait4()* позволяет задать дополнительную информацию о том, завершения каких именно процессов следует дождаться.

Процессы-зомби

Если дочерний процесс завершается в то время, когда родительский процесс заблокирован функцией *wait()*, он успешно удаляется из системных таблиц, а его код завершения передается предку. А что произойдет, если потомок завершился, а родитель так и не вызвал функцию *wait()*? Исчезнет ли дочерний процесс? Конечно, нет, ведь в этом случае информация о его завершении просто пропадет. Вместо этого дочерний процесс становится процессом-зомби.

Зомби – это процесс, который завершился, но не был удален. Удаление зомби возлагается на родительский процесс. Функция *wait()* тоже это делает, поэтому перед ее вызовом не нужно проверять, продолжает ли выполняться требуемый дочерний процесс. Допустим, программа создает дочерний процесс, выполняет нужные операции и затем вызывает функцию *wait()*. Если к тому времени дочерний процесс еще не завершился, эта функция заблокирует программу. В противном случае процесс на некоторое время превратится в зомби. Тогда *wait()* извлечет код его завершения, система удалит процесс, и функция немедленно завершится.

Но что же все-таки случится, если родительский процесс не удалит потомков? Они останутся в системе в виде зомби. Программа, код которой показан на СЛАЙДЕ 11, порождает дочерний процесс, который немедленно завершается, тогда как родитель берет 60-секундную паузу, после чего тоже заканчивает работу, так и не позаботившись об удалении потомка.

После сборки этой программы, запустим ее. Пока программа работает, перейдем с другой терминал и посмотрим список процессов. В списке должны отобразиться два процесса с именем *zombie*. Один из них предок, другой – потомок. У последнего PPID равен PID основного процесса *zombie*. Потомок при этом должен быть обозначен как <*defunct*>, а код его состояния равен *Z*.

Что будет, когда программа *zombie* завершится, не вызвав функцию *wait()*? Останется ли он зомби? Нет – достаточно выполнить команду *ps* и убедиться в этом: оба процесса *zombie* исчезли. Дело в том, что после завершения программы управление ее дочерними процессами принимает на себя *init*, т.е. процесс, чей PID равен 1. Он автоматически удаляет все унаследованные им дочерние процессы-зомби.

Асинхронное удаление дочерних процессов

Если дочерний процесс просто вызывает другую программу с помощью функции *exec()*, то в родительском процессе можно сразу же вызвать функцию *wait()* и пассивно

дожидаться завершения потомка. Однако иногда нужно, чтобы родитель продолжал выполняться параллельно с одним или несколькими потомками. Как в этом случае получать сигналы об их завершении? Для этого существует несколько способов.

Один подход заключается в периодическом вызове функции *wait3()* или *wait4()*. Обычная функция *wait()* не подходит из-за своего блокирующего характера. А названные функции принимают дополнительный флаг *WNOHANG*. Он переводит их в *неблокируемый* режим, в котором функция либо удаляет дочерний процесс, если он есть, либо просто завершается. В первом случае возвращается идентификатор процесса, во втором – 0.

Более изящный подход состоит в асинхронном уведомлении родительского процесса о завершении потомка. Существуют разные способы сделать это, но проще воспользоваться сигналом *SIGCHLD*, посылаемым как раз тогда, когда завершается дочерний процесс. По умолчанию программа не реагирует на этот сигнал, поэтому многие и не догадываются о его существовании.

Таким образом, нужно организовать удаление дочерних процессов в обработчике сигнала *SIGCHLD*. Код состояния придется сохранять в глобальной переменной, если эта информация необходима основной программе. В заготовке кода, представленной на СЛАЙДЕ 12, продемонстрирован данный прием.

Литература и дополнительные источники к Теме 2

1. Delve into UNIX process creation - http://www.ibm.com/developerworks/aix/library/au-unixprocess.html
2. YoLinux Tutorial: Fork, Exec and Process control - http://www.yolinux.com/TUTORIALS/ForkExecProcesses.html
3. Инструменты Linux для Windows-программистов - http://rus-linux.net/nlib.php?name=/MyLDP/BOOKS/Linux-tools/index.html
4. Лав, Р. Linux. Системное программирование/ Р.Лав. – СПб.: Питер, 2008. – 416 с.