Процессы и потоки

Андрей Боровский

Все процессы должны быть у нас под контролем
М.С. Горбачев

Понятие процесса играет ключевую роль в современных ОС. Существует много определений процессов, я воспользуюсь простым определением, данным в [1] – процесс, это выполняющийся экземпляр программы. Хотя это определение, в общем, «работает», оно не совсем подходит к многопоточным программам Linux, о которых мы поговорим в следующей статье. Важный аспект процессов с точки зрения ОС – неделимость процесса относительно выделения ресурсов. Система выделяет ресурсы процессу в целом, а не его частям. Внутри процесса задача управления ресурсами ложится на программиста. Для каждого процесса Linux ядро системы поддерживает специальную структуру данных task_struct, в которой хранятся важнейшие параметры процесса (процессы уровня пользователя не имеют доступа к этой структуре). В структуре task_struct есть специальное поле, в котором хранится численный идентификатор процесса (Process Identifier, PID). Именно этот идентификатор используется для обозначения процессов на уровне прикладного API Linux.

Под управлением процессами мы будем понимать, прежде всего, создание новых процессов и контроль их выполнения. Обычно новые экземпляры программы создаются в Unix-системах с помощью вызова функции fork(2), с которой мы уже многократно встречались в статьях этой серии. Помимо функции fork() Linux предоставляет нам еще один, весьма тонкий инструмент - функцию clone(2). Эта функция позволяет настраивать разделение ресурсов между процессами, но не умеет раздваивать процесс в точке вызова, как это делает fork(). Специально для того чтобы вас запутать существует еще и системный вызов sys_clone(), который умеет раздваивать процессы в точке вызова.

Вы, конечно, знаете, что, будучи вызвана один раз, функция fork() возвращает управления дважды, двум копиям исходного процесса (родительской и дочерней). Поскольку вы знаете так же, что эти две копии процесса похожи как две капли воды, будет полезнее перечислить ниже не сходства этих процессов, а их различия.

- Родительскому процессу функция fork() возвращает PID дочернего процесса, а дочернему процессу – значение 0. Именно по возвращенному fork() значению процесс может узнать, дочерний он или родительский. В иерархии процессов родительский процесс оказывается родителем (странно, не правда ли) дочернего процесса. Дочерний процесс может получить PID родительского процесса с помощью вызова getppid(2), в то время как родительский процесс может узнать PID своего дочернего процесса только из результата fork(). Именно поэтому fork() может позволить себе не возвращать значение PID дочернему процессу, но обязана возвращать значение PID дочернего процесса родительскому.
- Значения tms_utime, tms_stime, tms_cutime, и tms_cstime в дочернем процессе обнуляются.
- Блокировки на уровне файлов (file locks), установленные в родительском процессе, в дочернем процессе снимаются (иначе сразу два процесса оказались бы владельцами блокировок).
- Сигналы, обработка которых была отложена в родительском процессе в момент вызова fork(), в дочернем процессе сбрасываются.

Если вы хотите запустить из вашей программы экземпляр другой программы, вы должны пожертвовать для этого уже существующим процессом. Если вы хотите, чтобы новая программа работала одновременно со старой, нужно сначала раздвоить процесс с помощью fork(), а затем заместить образ программы в одной из копий образом новой программы. Для замены образа одной программы образом другой применяются функции семейства exec*. Функций этого семейства всего шесть: execl(), execlp(), execle(), execv(), execvp() и execv(). Если вы присмотритесь к названиям функций, то увидите, что первые три имеют имена вида execl*, а вторые три – имена вида execv*. Эти функции отличаются списками параметров, а также тем, как обрабатывается имя файла.

Рассмотрим, например, функции execve(2) и execvp(2). Заголовок функции execve(2) имеет вид:

```
int execve(const char *pathname, char *const argv[], char *const envp []);
```

Первый параметр функции execve(), - это имя исполнимого файла запускаемой программы (исполнимым файлом может быть двоичный файл или файл сценария оболочки, в котором в первой строке указан интерпретатор). Имя исполнимого файла для этой функции должно включать полный путь к файлу, (путь, начиная с корневого слеша, точки или тильды). Второй параметр функции, представляет собой список аргументов командной строки, которые должны быть переданы запускаемой программе. Формат этого списка должен быть таким же, как у списка argv[], который получает функция main(), то есть, первым элементом должно быть имя запущенной программы, а последним – нулевой указатель на строку, то есть

```
(char *) NULL
```

В последнем параметре функции execve() передается список переменных среды окружения формате

```
ИМЯ=ЗНАЧЕНИЕ
```

Массив переменных должен заканчиваться нулевым указателем на строку, как и argv. Копию набора переменных окружения, полученного вашей программой от оболочки, можно получить из внешней переменной environ, которую вы должны объявить в файле вашей программы:

```
extern char ** environ;
```

Передача списка переменных окружения явным образом позволяет вам, в случае необходимости, модифицировать список переменных, заданных по умолчанию. В параметре argv, как и в параметре envp, можно передавать значения NULL, если вы уверены, что вызываемой программе не нужны переменные окружения или командная строка. Заголовок функции execvp() выглядит проще:

```
int execvp(const char *file, char *const argv[]);
```

Первый параметр функции, это имя запускаемого файла. Функция execvp() выполняет поиск имен с учетом значения переменной окружения PATH, так что для программы, которая расположена в одной из директорий, перечисленных в PATH, достаточно указывать только имя исполнимого файла. Второй параметр функции execvp() соответствует второму параметру execve(). У функции execvp() нет параметра для передачи набора переменных среды окружения, но это не значит, что запущенная программа не получит эти переменные. Программе будет передана копия набора переменных окружения родительского процесса.

Вы можете изменить переменные среды окружения не только путем модификации списка environ, но и с помощью набора специальных функций. Для установки новой переменной окружения используется putenv(3). У функции putenv() один параметр типа char *. В этом параметре функции передается строка ИМЯ=ЗНАЧЕНИЕ, устанавливающая переменную окружения. Функция возвращает 0 в случае успеха и -1 в случае ошибки. Программа может прочитать

значение переменной окружения с помощью вызова getenv(3). У этой функции так же один параметр типа «строка с нулевым конечным символом». В этом параметре передается имя переменной, значение которой мы хотим прочитать. В случае успеха функция возвращает строку, содержащую значение переменной, а в случае неудачи (например, если запрошенной переменной не существует) – NULL.

Переменные среды окружения играют важную роль в работе процессов, поскольку многие системные функции используют их для получения различных параметров, необходимых для нормальной работы программ, однако, управление программами с помощью переменных окружения считается морально устаревшим методом. Если ваша программа должна получать какие-то данные извне, воздержитесь от создания новых переменных окружения. Современные программы чаще полагаются на файлы конфигурации, средства IPC и прикладные интерфейсы. Старайтесь так же не использовать переменные окружения для получения тех данных, которые вы можете получить с помощью специальных функций C, например, используйте getcwd(3) вместо обращения к переменной PWD. Полный список установленных переменных окружения и их значений вы можете получить с помощью команды env, а некоторые наиболее важные переменные приведены в таблице:

Переменная	**Описание**
DISPLAY	Имя дисплея X Window
HOME	Полный путь к домашней директории пользователя-владельца процесса
HOST	Имя локального узла
LANG	Текущая локаль
LOGNAME	Имя пользователя-владельца процесса
PATH	Список директорий для поиска имен файлов
PWD	Полный путь к текущей рабочей директории
SHELL	Имя оболочки, заданной для пользователя-владельца процесса по умолчанию
TERM	Терминал пользователя-владельца процесса по умолчанию
TMPDIR	Полный путь к директории для хранения временных файлов
TZ	Текущая временная зона

Помимо функций getenv() и putenv() есть еще несколько дополнительных функций для работы с переменными среды окружения. Функция setenv() может быть использована для созданяи новой переменной. В отличие от функции putenv(), эта функция позволяет установить флаг, благодаря которому значение уже существующей переменной не будет изменено. С помощью функции unsetenv() вы можете удалить переменную окружения. Наконец, если переменные среды окружения вам надоели, вы можете воспользоваться функцией clearenv() для удаления всего списка переменных. Под удалением переменных окружения подразумевается их удаление из среды окружения текущего процесса и его потомков. Это удаление никак не повлияет на переменные среды окружения других процессов.

Рассмотрим использование функции execvp() на примере программы, запускающей другую программу, имя которой передается ей в качестве аргумента командной строки. Назовем нашу программу exec (ее исходные тексты вы найдете на прилагаемом диске в файле exec.c). Если вы скомпилируете файл exec.c и скомандуете, например

```
./exec ls -al
```

программа выполнит команду ls –al и возвратит сведения о том, как был завершен соответствующий процесс. Ниже приводится полный исходный текст exec.

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

int main(int argc, char * argv[])
{
   int pid, status;
   if (argc < 2) {
     printf("Usage: %s command, [arg1 [arg2]...]\n", argv[0]);
     return EXIT_FAILURE;
   }
   printf("Starting %s...\n", argv[1]);
    pid = fork();
   if (pid == 0) {
     execvp(argv[1], &argv[1]);
     perror("execvp");
     return EXIT_FAILURE; // Never get there normally
   } else {
     if (wait(&status) == -1) {
       perror("wait");
       return EXIT_FAILURE;
     }
     if (WIFEXITED(status))
       printf("Child terminated normally with exit code %i\n",
         WEXITSTATUS(status));
     if (WIFSIGNALED(status))
       printf("Child was terminated by a signal #%i\n", WTERMSIG(status));
     if (WCOREDUMP(status))
       printf("Child dumped core\n");
     if (WIFSTOPPED(status))
       printf("Child was stopped by a signal #%i\n", WSTOPSIG(status));
    }
   return EXIT_SUCCESS;
}
```

Поскольку мы хотим, чтобы новая программа не заменяла старую, а выполнялась одновременно с ней, мы раздваиваем процесс с помощью fork(). В дочернем процессе, которому fork() возвращает 0, мы выполняем вызов execvp(). Первый параметр execvp() – значение argv[1], в котором должно быть передано имя запускаемой программы. В качестве второго параметра функции передается массив аргументов командной строки, полученных программой exec, начиная со второго элемента (элемент argv[1]). Например, если список аргументов программы exec имел вид "exec", "ls", "-al", то первым параметром функции execvp() будет строка "ls", а вторым параметром – массив из двух строк "ls" и "-al" (не забывайте, что первым элементом массива аргументов командной строки должно быть имя самой программы, по которому она была вызвана). Таким образом, вы можете, например, давать команды

```
./exec ls –al
```

```
./exec ./exec ls –al
```

```
./exec ./exec ./exec ls –al
```

и так далее. Чего вы не можете, однако, сделать, так это выполнить с помощью нашей программы команду типа

```
./exec "ls > log.txt"
```

Перенаправление ввода-вывода выполняется командной оболочкой. Для того, чтобы выполнять такие команды, необходимо запустить сначала оболочку, например:

```
exec sh –c "ls –al > log.txt"
```

Сразу после вызова функции execvp() в нашей программе следует вывод сообщения об ошибке. Вам может показаться странным, что мы не проверяем значение, возвращенное execlp(), но на самом деле этого и не требуется. Если вызов функции успешен, текущий образ процесса будет заменен другим образом, и, стало быть, при нормальном завершении exec() следующая за ней инструкция никогда не будет выполнена. Если программа перешла к инструкции после вызова execvp(), значит заменить образ программы не удалось и нам остается только вывести сообщение об ошибке.

Обратимся теперь к родительскому процессу. Мы хотим чтобы родительский процесс дождался завершения дочернего процесса и вывел сообщение о том, как был завершен дочерний процесс. Первую часть этой задачи мы могли бы решить с помощью уже известного нам механизма сигналов. Каждый раз, когда дочерний процесс приостанавливается или завершается, родительский процесс получает сигнал SIGCHLD. Мы могли бы приостановить работу программы до получения этого сигнала, но таким образом мы бы знали только то, что *один из* дочерних процессов программы завершился, но не знали бы, ни как он завершился, ни какой именно это был процесс. Причина этого конечно, в том, что сигналы сами по себе не несут никакой дополнительной информации. Однако, в нашем распоряжении есть функция wait(2), которая приостанавливает процесс до тех пор, пока один из его дочерних процессов не будет остановлен или не завершится, после чего возвращает информацию о том, какой процесс завершился и что стало причиной его завершения. Значение, возвращаемое функцией wait(), – это PID завершившегося процесса, а аргументом функции должен быть указатель на переменную status типа int. В этой переменной функция вернет дополнительные сведения о том, как завершился процесс. Вы могли подумать, что после того как мы создали два процесса с помощью fork(), не так уж важно, запускаем ли мы новую программу в дочернем или в родительском процессе, ведь разница между ними невелика. Теперь вы знаете как минимум одну причину придерживаться строгих правил, ведь родительский процесс может следить за дочерним, в то время как обратное невозможно.

Значение переменной status, в которой функция wait() передает дополнительные данные о завершившемся процессе, представляет собой маску из нескольких разных параметров. В файле <sys/wait.h> определены макросы, упрощающие «расшифровку» этой маски. Макрос WIFEXITED возвращает значение 1, если процесс завершился «добровольно», то есть в результате вызова exit() или _exit(). В этом случае с помощью макроса WEXITSTATUS можно узнать код завершения, возвращенный процессом. Макрос WIFSIGNALED возвращает 1, если выполнение процесса было завершено сигналом. Номер этого сигнала можно узнать с помощью макроса WTERMSIG. Макрос WIFSTOPPED возвращает значение 1, если выполнение процесса было приостановлено сигналом, номер которого возвращает макрос WSTOPSIG.

Помимо функции wait() в нашем распоряжении есть еще две функции, позволяющие приостановить процесс в ожидании изменения состояния дочерней программы. Это функции waitpid(2) и waitid(2). Функция waitpid() может приостановить процесс в ожидании изменения состояния определенного дочернего процесса, заданного значением PID. У функции waitpid() три параметра. Первый параметр – значение PID процесса, завершения которого ждет функция. Если передать в этом параметре значение -1, функция будет ждать изменения состояния любого процесса, аналогично wait(). Если первый параметр равен нулю, waitpid() ждет завершения любого процесса из той же группы, что и текущий. Второй параметр waitpid() аналогичен параметру wait(). Третий параметр позволяет указать дополнительные флаги функции. Например, если установить флаг NOHANG, функция вернет управление немедленно, даже если ни один дочерний процесс не завершился (подробнее об этом будет сказано ниже).

Результатом функции waitpid(), также как и в случае wait(), является идентификатор завершившегося процесса. Функция waitid(), появившаяся в Linux начиная с версии ядра 2.6.9, позволяет установить более тонкий контроль над параметрами ожидаемого события и получить более подробную информацию о сигнале, вызвавшем изменение состояния дочернего процесса. Для большинства задач возможности этой функции явно избыточны.

Исследуя работу программы exec, мы еще не коснулись одного важного момента, о котором часто забывают при отладке программ, создающих несколько процессов. Неявно мы все время предполагали, что вызов wait() в родительском процессе произойдет до завершения дочернего процесса. Чаще всего так и будет. Но что произойдет, если дочерний процесс завершится раньше вызова wait() в родительском процессе, например, вследствие ошибки при вызове execvp()? Программируя несколько потоков или процессов, мы всегда должны учитывать подобные варианты развития событий. В нашем конкретном случае беспокоиться не о чем. Если дочерний процесс завершится до завершения родительского процесса, система сохранит его «след», именуемый зомби. Зомби будет существовать до тех пор, пока его родительский процесс не вызовет wait() (waitpid(), waitid()), или пока родительский процесс не завершится. Система сохраняет зомби процессов специально для того, чтобы родительский процесс мог исследовать причины завершения дочернего процесса, однако это не всегда удобно. Рассмотрим, например, процесс init, который является «папой» всех пользовательских процессов сеанса Linux. Поскольку init существует на протяжении всего сеанса, любой из его дочерних процессов по завершении оставался бы в виде зомби до тех пор, пока init не вызвал бы одну из функций wait*. Еще один процесс, который может породить множество зомби, это демон... Иногда я должен напоминать себе, что пишу статью по программированию, а не сценарий фильма ужасов. Для того, чтобы процесс не оставлял зомби, можно вызывать функцию waitpid() периодически (с флагом NOHANG, чтобы она не блокировала вызвавший процесс если в системе нет зомби). Можно поступить и иначе, назначив сигналу SIGCHLD обработчик, в котором вызывалась бы функция wait(). Именно так поступает программа nozombies, которую вы найдете на диске.

```
int main(int argc, char * argv[])
{
   int i, pid;
   struct sigaction sa;
   sa.sa_handler = child_handler;
   sigaction(SIGCHLD, &sa, 0);

   for (i = 0; i < 10; i++) {
     pid = fork();
     if (pid == 0) {
       printf("I will leave no zombie\n");
       exit(0);
     }
     else
       printf("Created a process with the PID %i\n", pid);
   }
   while (1)
     sleep(1);
   return EXIT_SUCCESS;
}
```

Программа nozombies устанавливает обработчик сигнала SIGCHLD (функция child_handler()):

```
void child_handler(int i)
{
   int status;
```

```
   wait(&status);
}
```

Далее программа создает несколько дочерних процессов, каждый из которых выводит диагностическое сообщение и завершает свою работу. Затем программа nozombies переходит в бесконечный цикл (так что завершать ее придется с помощью Ctrl-C). Пока программа находится в бесконечном цикле, вы можете проверить, оставила ли она процессы-зомби. Для этого откройте другой терминал и скомандуйте

```
ps –al
```

Для сравнения, закоментируйте строку

```
wait(&status);
```

в теле функции child_handler(), перекомпилируйте и снова запустите программу. Теперь команда ps –al покажет наличие десяти зомби-процессов (они помечены символом Z).

Интерфейс функций exec* может показаться слишком сложным и, для многих задач, избыточным. Стандарт языка C включает описание функции system(), предназначенной для запуска внешних программ. Единственный аргумент этой функции – строка запуска программы. Поскольку для выполнения программы функция system() запускает копию оболочки (той, которая в вашей системе вызывается командой sh(1)), эта функция выполнит любую команду, которую вы могли бы ввести в командной строке оболочки. Функция system() (пример ее использования вы найдете в файле system.c) приостанавливает выполнение вызвавшей ее программы до тех пор, пока дочерний процесс не завершит работу и возвращает код завершения процесса.

Программируя управление процессами, мы часто идем по лезвию бритвы. Нет, дело тут вовсе не в ужасных зомби, а в том, что трудно заставить работать слаженно два независимых процесса. Труднее этого может быть только программирование потоков, которым мы и займемся в следующий раз.

Литература:

1. D. P. Bovet, M. Cesati, **Understanding the Linux Kernel, 3rd Edition,** O'Reilly, 2005
2. W. R. Stevens, S. A. Rago, **Advanced Programming in the UNIX® Environment: Second Edition,** Addison Wesley Professional, 2005