Изучение создания процесса UNIX

Анализ жизненного цикла процесса, запускаемого операционной системой UNIX

Шон Волберг

Опубликовано 01.03.2007

Одна из многочисленных обязанностей системного администратора – это обеспечивать правильный запуск программ пользователей. Эта задача усложняется наличием в системе других одновременно выполняющихся программ. По разным причинам эти программы могут не работать, зависать или быть неисправными. Понимание процесса создания, управления и уничтожения этих заданий в операционной системе UNIX® является важнейшим шагом к созданию более надежной системы.

Разработчики изучают, как ядро управляет процессами, еще и потому, что приложения, которые нормально работают с другими составляющими системы, требуют меньше ресурсов и не так часто вызывают проблемы у системных администраторов. Приложение, которое постоянно нужно перезапускать, потому что оно создает процессы-зомби (описываются дальше), естественно не желательно. Понимание системы UNIX означает, что управляемые процессы позволяют разработчикам создавать программы, которые спокойно выполняются в фоновом режиме. Необходимость в сеансе работы с терминалом, который должен отображаться на чьем-то экране, отпадает.

Основным компоновочным блоком управления этих программ является процесс. Процесс – это имя, присвоенное программе, выполняемой операционной системой. Если вы знаете команду `ps`, вам должен быть знаком список процессов, такой как в листинге 1.

Листинг1. Вывод команды `ps`

```
sunbox#ps -ef
     UID   PID  PPID   C    STIME TTY         TIME CMD
    root     0     0   0 20:15:23 ?           0:14 sched
    root     1     0   0 20:15:24 ?           0:00 /sbin/init
    root     2     0   0 20:15:24 ?           0:00 pageout
    root     3     0   0 20:15:24 ?           0:00 fsflush
  daemon   240     1   0 20:16:37 ?           0:00 /usr/lib/nfs/statd
...
```

Для рассмотрения важны первые три колонки. В первой находится список пользователей,от имени которых работают процессы, во второй перечисляются ID процессов, в третьей - ID родительских процессов. Последняя колонка содержит описание процесса, как правило, имя запущенной программы. Каждому процессу присвоен идентификатор, который называется идентификатор процесса (PID). Также у процесса есть родитель, в большинстве случаев указывается PID процесса, который запустил данный процесс.

Существование родительского PID (PPID), означает, что один процесс создается другим процессом. Исходный процесс, который запускается в системе, называется `init`, и ему всегда присваивается PID 1. `init` - это первый действительный процесс, запускаумый ядром при загрузке. Основная задача `init`запуск всей системы. `init` и другие процессы с PPID 0 являются процессами ядра.

Использование системного вызова fork

Системный вызов `fork(2)` создает новый процесс. В листинге 2 показан `fork` используемый в простом примере С-кода.

Листинг 2. Простое применение fork(2)

```
sunbox$ cat fork1.c
#include <unistd.h>
#include <stdio.h>

int main (void) {

        pid_t p; /* fork returns type pid_t */
        p = fork();
        printf("fork returned %d\n",  p);
}

sunbox$ gcc fork1.c -o fork1
sunbox$ ./fork1
fork returned 0
fork returned 698
```

Код в `fork1.c` просто вызывает `fork` и отображает целочисленный результат выполения `fork` через вызов `printf`. Делается только один вызов, но вывод отображается дважды. Это происходит потому, что новый процесс создается в рамках вызова `fork`. После вызова возвращаются два отдельных процесса. Это часто называют "вызванный единожды, возвращается дважды."

Возвращаемые `fork` значения очень интересны. Одно из них - 0; другое – ненулевое значение. Процесс, который получает 0, называется *порожденным процессом*, а ненулевое значение достается исходному процессу, который является *родительским процессом*. Вы используете возвращаемые значения, для того чтобы определить, где какой процесс. Поскольку оба процесса возобновляют выполнение в одной и той же области, единственный возможный дифференциатор это возвращаемые значения `fork`.

Логическим основанием для нулевого и ненулевого возвращаемого значения служит то, что порожденный процесс всегда может вычислить своего родителя с помощью запроса `getppid(2)`, однако родителю намного сложнее определить всех своих потомков. Таким образом, родитель узнает о своем новом потомке, и потомок при необходимости может отыскать своего родителя.

Теперь, зная о возвращаемом значении `fork`, код может различать порожденный и родительский процессы и вести себя соответствующе. В листинге 3 показана программа, которая отображает разные выводы, основанные на результатах `fork`.

Листинг 3. Более полный пример использования fork

```
sunbox$ cat fork2.c
#include <unistd.h>
#include <stdio.h>

int main (void) {

        pid_t p;

        printf("Original program,  pid=%d\n",  getpid());
        p = fork();
        if (p == 0) {
                printf("In child process,  pid=%d,  ppid=%d\n",
                        getpid(),  getppid());
        } else {
                printf("In parent,  pid=%d,  fork returned=%d\n",
                        getpid(),  p);
```

```
        }
}

sunbox$ gcc fork2.c -o fork2
sunbox$ ./fork2
Original program,  pid=767
In child process,  pid=768,  ppid=767
In parent,  pid=767,  fork returned=768
```

В листинге 3 распечатываются PID распечатываются на каждом шаге, и код проверяет возвращаемые `fork` значения, для того чтобы определить, какой процесс родительский, а какой порожденный (дочерний). Сравнивая распечатанные PID, вы можете увидеть, что исходный процесс – это родительский процесс (PID 767), и порождаемый процесс (PID 768) знает, кто его родитель. Обратите внимание на то, как потомок находит своего родителя при помощи `getppid`, и как родитель использует результат `fork` для поиска своих потомков.

Теперь, когда вы разобрались с методом дублирования процессов, давайте рассмотрим, как выполнять разные процессы. `fork` - это только половина уравнения. Семейство системных вызовов `exec` запускает определенную программу.

Использование семейства системных вызовов exec

Задачей `exec` является замена текущего процесса на новый процесс. Отметьте использование слова *заменить*. Как только вы вызываете `exec`, текущий процесс завершается и начинается новый. Если вы хотите создать отдельный процесс, сначала вы должны вызвать `fork`, затем вызвать `exec` для новой программы в дочернем процессе. В листинге 4 показан этот сценарий.

Листинг 4. Запуск разных программ посредством соединения `fork` с `exec`

```

sunbox$ cat exec1.c
#include <unistd.h>
#include <stdio.h>

int main (void) {

        /* Определить массив с завершающим нулем команды для запуска
         следующим за любым параметром,  в этом случае никаким */
        char *arg[] = { "/usr/bin/ls",  0 };

        /* fork и exec в порожденном процессе */
        if (fork() == 0) {
                printf("In child process:\n");
                execv(arg[0],  arg);
                printf("I will never be called\n");
        }
        printf("Execution continues in parent process\n");
}
sunbox$ gcc exec1.c -o exec1
sunbox$ ./exec1
В порожденном процессе:
fork1.c      exec1        fork2       exec1.c      fork1
fork2.c
Выполнение продолжается в родительском процессе

```

Код в листинге 4 прежде всего, определяет массив, первый элемент которого является путем к исполняемой программе, а остальные элементы представляют собой параметры командной строки. Массив заканчивается нулевым символом. После возврата от системного вызова `fork` дочерний процесс должен запустить новую программу с помощью `execv`.
При вызове `execv` в первую очередь получает указатель на строку с именем программы для запуска, а затем указатель на массив параметров, который вы задали ранее. Первый элемент массива фактически является именем программы, следовательно, параметры начинают перечисляться со второго элемента. Запомните, что порожденный процесс никогда не возвращается после вызова `execv`. Это означает, что выполняемый процесс заменяется на новый. Есть и другие системные вызовы `exec`. Они отличаются способом приема параметров и вопросом относительно необходимости передачи переменных окружения. `execv(2)` - это один из самых простых способов замены текущего образа процесса, поскольку он использует массив с завершающим null-символом и не требует информации об окружении. Другие варианты: `execl(2)`, который принимает параметры в отдельные аргументы или `execvp(2)`, который также принимает массив переменных окружения с завершающим null-символом. Причем, не каждая операционная система поддерживает все варианты. Выбор зависит от платформы, способа программирования и от того, нужно ли вам определять какие-либо переменные окружения.

Что происходит с открытыми файлами, когда вызывается fork?

Когда процесс дублируется, ядро создает копии всех дескрипторов открытых файлов. Дескриптор файла - это целое число, которое является ссылкой на открытый файл или устройство, и используется для чтения и записи. Если открытие файла находится в программе перед `fork`, то что произойдет, если оба процесса попытаются читать или записывать? Перезапишет один из процессов данные другого? Будут прочитаны две копии файла? Эти вопросы изучаются в листинге 5, посредством открытия двух файлов - одного для чтения, другого для записи, а также при помощи одновременного чтения и записи родителем и потомком.

Листинг 5. Два процесса одновременно записывают и читают один и тот же файл.

```
1     #include <stdio.h>
2     #include <strings.h>
3     #include <unistd.h>
4     #include <sys/types.h>
5     #include <sys/stat.h>
6     #include <fcntl.h>
7
8     int main(void) {
9
10            int fd_in,  fd_out;
11            char buf[1024];
12
13            memset(buf,  0,  1024); /* пустой буфер*/
14            fd_in = open("/tmp/infile",  O_RDONLY);
15            fd_out = open("/tmp/outfile",  O_WRONLY|O_CREAT);
16
17            fork(); /* Потомок против родителя значения не имеет */

              while (read(fd_in,  buf,  2) > 0) { /* Цикл через infile */
                      printf("%d: %s",  getpid(),  buf);
```

```
                /* Написать строку */
                sprintf(buf,  "%d Hello,  world!\n\r",  getpid());
                write(fd_out,  buf,  strlen(buf));
                sleep(1);
                memset(buf,  0,  1024); /* пустой буфер*/
        }
        sleep(10);
}

sunbox$ gcc fdtest1.c -o fdtest1
sunbox$ ./fdtest1
2875: 1
2874: 2
2875: 3
2874: 4
2875: 5
2874: 6
2874: 7
sunbox$ cat /tmp/outfile
2875 Hello, world!
2874 Hello, world!
2875 Hello, world!
2874 Hello, world!
2875 Hello, world!
2874 Hello, world!
2874 Hello, world!
```

Листинг 5 - это простая программа, открывающая файл и вызывающая `fork`. Каждый процесс использует для чтения один и тот же дескриптор файла (это просто текстовый файл с числами от 1 до 7), печатая то, что было прочитано вместе с PID. После прочтения строки PID записывается в выходной файл (out file). Цикл завершается, когда в файле больше не остается непрочитанных символов.

Результаты работы кода листинга 5 показывают, что, таким образом, когда один процесс читает из файла, указатель файла смещается для обоих процессов. Также, когда файл записывается, каждый следующий символ добавляется в конец файла. Это имеет смысл, поскольку ядро отслеживает информацию об открытом файле. Дескриптор файла это просто идентификатор для процесса.

Вы должно быть также знаете о том, что стандартный вывод (экран) тоже имеет дескриптор файла. Это видно во время вызова `fork`: оба процесса могут выводить информацию на экран.

Смерть родителя или потомка

В какой-то момент процессы должны завершаться. Вопрос только в том, какой завершится первым: родитель или потомок.

Родительский процесс завершается раньше потомка

Если родительский процесс умирает раньше своих потомков, осиротевшие потомки должны знать, кто их родитель. Вспомните о том, что у каждого процесса есть родитель, и вы можете полностью отследить дерево процессов, начиная с PID 1, имеющего также название `init`. Когда родитель умирает, `init` усыновляет всех его потомков, как показано в листинге 6.

Листинг 6. Родительский процесс умирает раньше потомка

```
1
2
3     #include <unistd.h>
4     #include <stdio.h>
5
6     int main(void) {
7
8             int i;
9             if (fork()) {
10                    /* Родитель */
11                    sleep(2);
12                    _exit(0);
13            }
14            for (i=0; i < 5; i++) {
15                    printf("My parent is   %d\n",  getppid());
16                    sleep(1);
17            }
18    }
19    sunbox$ gcc die1.c -o die1
20    sunbox$ ./die1
21    My parent is   2920
22    My parent is   2920
23    sunbox$ My parent is 1
      My parent is  1
      My parent is  1
```

В этом примере родительский процесс вызывает `fork`, ждет две секунды и завершается. Порожденный процесс продолжается, распечатывая PID своего родителя в течение пяти секунд. Вы можете видеть, что когда родитель умирает, PPID изменяется на 1. Также интересно возвращение управления командному процессору. Поскольку порожденный процесс выполняется в фоне, как только родитель умирает, управление возвращается к командному процессору.

Потомок умирает раньше родителя

Листинг 7 описывает процесс противоположный листингу 6: смерть потомка раньше родителя. Для того чтобы лучше показать, что происходит, непосредственно из процесса ничего не распечатывается. Вместо этого, интересная информация находится в списке процессов.

Листинг 7. Порожденный процесс умирает раньше родительского

```
1       sunbox$ cat die2.c
2     #include <unistd.h>
3     #include <stdio.h>
4
5     int main(void) {
6
7             int i;
8             if (!fork()) {
9                     /* Потомок немедленно завершается*/
10                    _exit(0);
11            }
12        /* Родитель ждет около минуты*/
13            sleep(60);
14    }
15
16    sunbox$ gcc die2.c -o die2
17    sunbox$ ./die2 &
      [1] 2934
      sunbox$ ps -ef | grep 2934
          sean  2934  2885   0 21:43:05 pts/1       0:00 ./die2
```

```
    sean  2935  2934   0        - ?           0:00 <defunct>
sunbox$ ps -ef | grep 2934
[1]+  Exit 199                ./die2
```

`die2` выполняется в фоновом режиме, используя оператор `&`, после этого на экран выводится список процессов, отображая только выполняемый процесс и его потомков. PID 2934 – родительский процесс, PID 2935 – процесс, который создается и немедленно завершается. Несмотря на преждевременный выход, порожденный процесс все еще находится в таблице процессов, уже как *умерший* процесс, который еще называется *зомби*. Когда через 60 секунд родитель умирает, оба процесса завершаются.
Когда порожденный процесс умирает, его родитель информируется при помощи сигнала, который называется `SIGCHLD`. Точный механизм всего этого сейчас не имеет значения. Что действительно важно, так это то, что родитель должен как-то узнать о смерти потомка. С момента смерти потомка и до того момента как родитель получает сигнал, потомок находится в состоянии зомби. Зомби не выполняется и не потребляет ресурсов CPU; он только занимает пространство в таблице процессов. Когда родитель умирает, ядро наконец-то может убрать потомков вместе с родителем. Значит, единственный способ избавиться от зомби - это убить родителя. Лучший способ справиться с зомби – гарантировать, что они не окажутся на первом месте. Код в листинге 8 описывает обработчик сигналов, для работы с входящим сигналом `SIGCHLD`.

Листинг 8. Обработчик сигналов в действии

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

void sighandler(int sig) {
        printf("In signal handler for signal  %d\n",  sig);
        /* wait() это основное для подтверждения SIGCHLD */
        wait(0);
}

int main(void) {

        int i;
        /* Установить обработчик сигнала к SIGCHLD  */
        sigset(SIGCHLD,  &sighandler);
        if (!fork()) {
                /* Потомок */
                _exit(0);
        }
        sleep(60);
}
sunbox$ gcc die3.c -o die3
sunbox$ ./die3 &
[1] 3116
sunbox$ In signal handler for signal  18
ps -ef | grep 3116
    sean  3116  2885   0 22:37:26 pts/1       0:00 ./die3
```

Листинг 8 немного сложнее, чем предыдущий пример, поскольку там есть функция `sigset`, которая устанавливает указатель функции на обработчик сигнала. Всякий раз, когда процесс получает обработанный сигнал, вызывается функция, заданная через `sigset`. Для сигнала `SIGCHLD`, приложение должно вызвать функцию `wait(3c)` для того, чтобы подождать завершения порожденного процесса. Поскольку процесс уже завершен, это необходимо для того, чтобы ядро получило подтверждение о смерти потомков. На самом деле, родителю следовало бы сделать больше, чем просто подтвердить сигнал. Ему следовало бы также очистить данные потомка.
После выполнения `die3`, проверяется список процессов. Обработчик сигнала получает значение 18 (`SIGCHLD`), подтверждение о завершении потомка сделано, и родитель возвращается в `состояние ожидания sleep(60)`.

Краткие выводы

Процессы UNIX создаются, когда один процесс вызывает `fork`, который разделяет выполняемый процесс на два. После этого процесс может выполнить один из системных вызовов в семействе `exec`, который заменяет текущий образ на новый. Когда родительский процесс умирает, всех его потомков усыновляет `init`, имеющий PID 1. Если потомок умирает раньше родителя, родительскому процессу передается сигнал, а потомок переходит в состояние зомби до тех пор, пока сигнал не подтвердится или родительский процесс не будет убит.
Теперь, когда вы знаете, как создаются и уничтожаются процессы, вам будет проще разобраться в процессах, работающих в вашей системе. Особенно это касается процессов со сложной структурой, которые усложняются множественным возовом других процессов, таких как Apache. Возможность прослеживать дерево процессов для какого-то отдельного процесса, позволяет отследить любое приложение до процесса