Работа с процессами в Linux

Понятие *процесса* является очень важным понятием в *ОС Linux*. Процессы используют для организации многозадачной работы; они образуют иерархию, называемую деревом процессов. Во главе этой иерархии находится системный процесс init, используемый при загрузке системы. Практически все процессы, работающие в системе, прямо или косвенно порождаются от него. Процесс, порождающий другой процесс, называется *родителем* или *родительским процессом*; процесс, порожденный от него, называется *потомком*.

Работа с процессами в оболочке

Каждый процесс запускается от имени какого-то пользователя. Процессы, которые стартовали при загрузке, обычно выполняются от имени пользователей root или nobody.

Каждый пользователь может управлять поведением процессов, им запущенных. При этом пользователь root может управлять всеми процессами — как запущенными от его имени, так и процессами, порожденными другими пользователями операционной системы. Управление процессами осуществляется с помощью утилит, а также посредством некоторых команд командной оболочки (shell).

Каждый процесс в системе имеет уникальный номер — идентификационный номер процесса (*Process Identification*, *PID*). Этот номер используется ядром операционной системы, а также некоторыми утилитами для управления процессами.

Выполнение процесса на переднем плане и в фоновом режиме

Процессы могут выполняться на переднем плане (*foreground*) — режим по умолчанию и в фоновом режиме (*background*). На переднем плане в каждый момент для текущего терминала может выполняться только один процесс. Однако пользователь может перейти в другой виртуальный терминал и запустить на выполнение еще один процесс, а на другом терминале еще один и т. д. Процесс переднего плана – это процесс, с которым вы взаимодействуете, он получает информацию с клавиатуры (стандартный ввод) и посылает результаты на ваш экран (стандартный вывод).

Фоновый процесс после своего запуска благодаря использованию специальной команды командной оболочки отключается от клавиатуры и экрана, т. е. не ожидает ввода данных со стандартного ввода и не выводит информацию на стандартный вывод, а командная оболочка не ожидает окончания запущенного процесса, что позволяет пользователю немедленно запустить еще один процесс.

Обычно фоновые процессы требуют очень большого времени для своего завершения и не требуют вмешательства пользователя во время существования процесса. К примеру, компиляция программ или архивирование большого объема информации — кандидаты номер один для перевода процесса в фоновый режим.

Процессы так же могут быть отложенными. Отложенный процесс — это процесс, который в данный момент не выполняется и временно остановлен. После того как вы остановили процесс, в дальнейшем вы можете его продолжить как на переднем плане, так и в фоновом режиме. Возобновление приостановленного процесса не изменит его состояния — при возобновлении он начнется с того места, на котором был приостановлен.

Для выполнения программы в режиме переднего плана достаточно просто набрать имя программы в командной строке и запустить ее на выполнение.

Для запуска программы в качестве фонового процесса достаточно набрать в командной строке имя программы и в конце добавить знак амперсанта (&), отделенный пробелом от имени программы и ее параметров командной строки, если таковые имеются.

Например, команда для запуска команды yes (yes [string] — вывод с бесконечным повтором переданной строки.) в фоновом режиме с подавлением вывода имеет вид:

```
/home/user$ yes > /dev/null &
```

После команды выводится сообщение, состоящее из двух чисел. Первое число в скобках означает номер запущенного фонового процесса для пользователя в текущем сеансе, с его помощью можно производить манипуляции с этим фоновым процессом. Второе число показывает идентификационный номер (*PID*) процесса. Отличия этих двух чисел достаточно существенные. Номер фонового процесса уникален только для пользователя, запускающего данный фоновый процесс. То есть, если в системе три пользователя решили запустить фоновый процесс (первый для текущего сеанса), то в результате у каждого пользователя появится фоновый процесс с номером 1. Напротив, идентификационный номер процесса (*PID*) уникален для всей операционной системы и однозначно идентифицирует в ней каждый процесс. Нумерация фонового процесса для пользователя нужна для удобства. Номер фонового процесса хранится в переменных командной оболочки пользователя и позволяет не забивать голову цифрами типа 2693 или 1294, а использовать переменные типа 1 и т. д.

Для проверки состояния фоновых процессов можно воспользоваться командой командной оболочки — jobs.

```
/home/user$ jobs
[1]+    Running         yes >/dev/null &
/home/user$
```

Результат команды показывает, что у пользователя user в данный момент запущен один фоновый процесс, и он выполняется.

Обратите внимание, что подавление вывода команды осуществляется перенаправлением выходного потока на псевдоустройство /dev/null . Все, что записывается в этот файл, "*исчезает*" навсегда.

Остановка и возобновление процесса

Помимо прямого указания выполнять программу в фоновом режиме, существует еще один способ перевести процесс в фоновый режим. Для этого мы должны выполнить следующие действия:

1. Запустить процесс выполняться на переднем плане.
2. Приостановить выполнение процесса.
3. Продолжить процесс в фоновом режиме.

Для выполнения программы введем ее имя в командной строке и запустим на выполнение. Для остановки выполнения программы необходимо нажать на клавиатуре следующую комбинацию клавиш — <Ctrl>+<Z>. После этого вы увидите на экране примерно следующее:

```
/home/user$ yes > /dev/null
ctrl+Z
[1]+    Stopped         yes >/dev/null
/home/user$
```

Для того чтобы перевести выполнение этого процесса в фоновый режим, необходимо выполнить следующую команду:

```
/home/user$ bg %1
```

Причем необязательно делать это сразу после остановки процесса, главное правильно указать номер остановленного процесса.

Для того чтобы вернуть процесс из фонового режима выполнения на передний план, достаточно выполнить следующую команду:

```
/home/user$ fg %1
```

В том случае, если вы хотите перевести программу в фоновый или, наоборот, на передний план выполнения сразу после остановки процесса, можно выполнить соответствующую программу без указания номера остановленного процесса.

Существует большая разница между фоновым и остановленным процессом. Остановленный процесс не выполняется и не потребляет ресурсы процессора, однако занимает оперативную память или пространство свопинга. В фоновом же режиме процесс продолжает выполняться.

Завершение работы процесса

Рассмотрим возможные способы завершения процесса.

Вариант первый. Если процесс интерактивный, как правило, в документации или прямо на экране написано, как корректно завершить программу.

Вариант второй. В том случае, если вы не знаете, как завершить текущий процесс (не фоновый), можно воспользоваться клавиатурной комбинацией <Ctrl>+<C>. Попробуйте также комбинацию клавиш <Ctrl>+<Break>. А для остановки фонового процесса можно перевести его на передний план, а затем уже воспользоваться вышеприведенными клавиатурными комбинациями.

Вариант третий и самый действенный. В том случае, если вам не удалось прекратить выполнение процесса вышеприведенными способами – например, программа зависла – для завершения процесса можно воспользоваться следующими командами: kill, killall.

Команда kill может получать в качестве аргумента, как номер процесса, так и идентификационный номер (*PID*) процесса. Таким образом, команда:

```
/home/user$ kill 123
```

эквивалентна команде:

```
/home/user$ kill %1
```

Можно видеть, что не надо использовать "%", когда вы обращаетесь к работе по идентификационному номеру (*PID*) процесса.

С помощью команды killall можно прекратить выполнение нескольких процессов сразу, имеющих одно и то же имя. Например, команда

```
/home/user$ killall mc
```

прекратит работу всех программ mc, запущенных от имени данного пользователя.

Для того чтобы завершить работу процесса, вам надо быть его владельцем. Это сделано в целях безопасности. Если бы одни пользователи могли завершать процессы других пользователей, открылась бы возможность исполнения в системе множества злонамеренных

действий. Пользователь root может завершить работу любого процесса в операционной системе.

Программы, используемые для управления процессами

Существует достаточно большое количество команд, используемых для управления процессами, исполняемыми в операционной системе. Сейчас мы рассмотрим только самые основные.

at
: Выполняет команды в определенное время

batch
: Выполняет команды тогда, когда это позволяет загрузка системы

cron
: Выполняет команды по заранее заданному расписанию

crontab
: Позволяет работать с файлами crontab отдельных пользователей

kill
: Прекращает выполнение процесса

nice
: Изменяет приоритет процесса перед его запуском

nohup
: Позволяет работать процессу после выхода пользователя из системы

ps
: Выводит информацию о процессах

renice
: Изменяет приоритет работающего процесса

w
: Показывает, кто в настоящий момент работает в системе и с какими программами

nohup

Эта утилита позволяет организовать фоновый процесс, продолжающий свою работу даже тогда, когда пользователь отключился от терминала, в отличие от команды &, которая этого не позволяет. Для организации такого фонового процесса необходимо выполнить команду в форме:

```
nohup выполняемая_фоновая_команда &
```

Во вновь запущенном терминале процесс нельзя увидеть с помощью команды jobs, так как эта команда выводит список процессов текущего терминала, поэтому после подключения к терминалу необходимо использовать команду ps с ключом -A.

ps

Программа предназначена для получения информации о существующих в операционной системе процессах. У этой команды есть множество различных опций, но мы остановимся на самых часто используемых. Для получения подробной информации смотрите *man*-страницу этой программы.

Простой запуск ps без параметров выдаст список программ, выполняемых на текущем терминале. Обычно этот список очень мал, например:

```
PID   TTY    TIME        CMD
885    tty1  00:00:00    login
```

```
893    tty1  00:00:00      bash
955    tty1  00:00:00      ps
```

Первый столбец — *PID* (идентификационный номер процесса). Как уже упоминалось, каждый выполняющийся процесс в системе получает уникальный идентификатор, с помощью которого производится управление процессом. Каждому вновь запускаемому на выполнение процессу присваивается следующий свободный *PID*. Когда процесс завершается, его номер освобождается. Когда достигнут максимальный *PID*, следующий свободный номер будет взят из наименьшего освобожденного.

Следующий столбец — *TTY* — показывает, на каком терминале процесс выполняется.

Столбец *TIME* показывает, сколько процессорного времени выполняется процесс. Оно не является фактическим временем с момента запуска процесса, поскольку *Linux* — это многозадачная операционная система. Показывается время, реально потраченное процессором на выполнение процесса.

Столбец *CMD* показывает имя программы, опции командной строки не выводятся.

Для получения расширенного списка процессов, выполняемых в системе, используется следующая команда:

Для получения расширенного списка процессов, выполняемых в системе, используется следующая команда:

```
ps -ax
```

Как правило, список запущенных процессов в системе велик и достаточно сильно зависит от конфигурации операционной системы. Опции, заданные программе в этом примере, заставляют ее выводить не только имена программ, но и список опций, с которыми были запущены программы.

Появился новый столбец — *STAT*, в котором отображается состояние (*status*) процесса. Полный список состояний вы можете прочитать в описании программы `ps`. Опишем самые важные состояния:

- буква `R` обозначает запущенный процесс, исполняющийся в данный момент времени;
- буква `S` обозначает спящий (*sleeping*) процесс — процесс ожидает какое-то событие, необходимое для его активизации;
- буква `Z` используется для обозначения "*зомбированных*" процессов (*zombied*) — это процессы, родительский процесс которых прекратил свое существование, оставив дочерние процессы рабочими.

Обратите внимание на колонку *TTY*. Как вы, наверное, заметили, многие процессы, расположенные в верхней части таблицы, в этой колонке содержат знак `"?"` вместо терминала. Так обозначаются процессы, запущенные с более не активного терминала. Как правило, это всякие системные сервисы.

Если вы хотите увидеть еще больше информации о выполняемых процессах, попробуйте выполнить команду:

```
ps –aux
```

Появились еще столбцы:

- `USER` — показывает, от имени какого пользователя был запущен данный процесс;
- `%CPU`, `%MEM` — показывают, сколько данный процесс занимает соответственно процессорного времени и объем используемой оперативной памяти;
- `TIME` — время запуска программы.

Рассмотрим некоторые ключи программы ps.

a
: Показать процессы всех пользователей

c
: Имя команды из переменной среды

e
: Показать окружение

f
: Показать процессы и подпроцессы

h
: Вывод без заголовка

j
: Формат заданий

l
: "Длинный" формат вывода

m
: Вывод информации о памяти

n
: Числовой вывод информации

r
: Только работающие процессы

s
: Формат сигналов

S
: Добавить время использования процессора порожденными процессами

txx
: Только процессы, связанные с терминалом xx

u
: Формат вывода с указанием пользователя

v
: Формат виртуальной памяти

w
: Вывод без обрезки информации для размещения в одной строке

x
: Показать процессы без контролирующего терминала

top

Еще одна утилита, с помощью которой можно получать информацию о запущенных в операционной системе процессах. Для использования достаточно просто запустить команду top на выполнение. Эта утилита выводит на экран список процессов в системе, отсортированных в порядке убывания значений использованного процессорного времени.

Сначала идет общесистемная информация — из нее можно узнать время запуска операционной системы, время работы операционной системы от момента последнего перезапуска системы, количество зарегистрированных в данный момент в операционной системе пользователей, а также минимальную, максимальную и среднюю загрузку операционной системы. Помимо этого, отображается общее количество процессов и их состояние, сколько процентов ресурсов системы используют пользовательские процессы и системные процессы, использование оперативной памяти и свопа.

Далее идет таблица, во многом напоминающая вывод программы ps. Идентификационный номер процесса, имя пользователя — владельца процесса, приоритет процесса, размер процесса, его состояние, используемые процессом оперативная память и

ресурс центрального процесса, время выполнения и, наконец, имя процесса.

Утилита top после запуска периодически обновляет информацию о состоянии процессов в операционной системе, что позволяет нам динамически получать информацию о загрузке системы.

Утилита top полностью управляется с клавиатуры. Вы можете получить справку, нажав клавишу h. Вот еще несколько полезных команд:

k — используется для отправки сигнала процессу;

u — используется для вывода процессов указанного пользователя;

i — используется для вывода только работающих процессов;

r — используется для изменения приоритета выбранного процесса.

Можно переключать режимы отображения информации с помощью следующих команд:

<Shift>+<N> — сортировка по PID;

<Shift>+<A> — сортировать процессы по возрасту;

<Shift>+<P> — сортировать процессы по использованию ЦПУ;

<Shift>+<M> — сортировать процессы по использованию памяти;

<Shift>+<T> — сортировка по времени выполнения.

kill

Программа kill (в переводе с английского — *убить*) предназначена для посылки соответствующих сигналов указанному нами процессу. Как правило, это бывает тогда, когда некоторые процессы начинают вести себя неадекватно. Наиболее часто программа применяется, чтобы прекратить выполнение процессов.

Для того чтобы прекратить работу процесса, необходимо знать *PID* процесса либо его имя. Например, чтобы "убить" процесс 123, достаточно выполнить следующую команду:

```
kill 123
```

В этом случае по умолчанию процессу посылается сигнал SIGTERM (*terminate*, завершиться). Процесс, получивший данный сигнал, должен корректно завершить свою работу (закрыть используемые файлы, сбросить буферы ввода/вывода и т. п.).

Как обычно, чтобы прекратить работу процесса, вам необходимо быть его владельцем. Однако, пользователь root может прекратить работу любого процесса в системе.

Иногда стандартное выполнение программы kill не справляется с поставленной задачей. Обычно это объясняется тем, что данный процесс завис, либо выполняет операцию, которую с его точки зрения нельзя прервать немедленно. Для прерывания этого процесса можно воспользоваться следующей командой:

```
kill -9 123
```

Ключ -9 указывает посылать процессу другой тип сигнала — SIGKILL. Это приводит к тому, что процесс не производит корректного завершения, а немедленно прекращает свою жизнедеятельность. Помимо этих сигналов, в вашем распоряжении целый набор различных сигналов. Полный список сигналов можно получить, выполнив следующую команду:

```
kill -l
```

Вы увидите внушительный список сигналов:

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL
5) SIGTRAP	6) SIG7ABRT	7) SIGBUS	8) SIGFPE

9) SIGKILL	10) SIGUSR1	11) SIGSEGV	12) SIGUSR2
13) SIGPIPE	14) SIGALRM	15) SIGTERM	и т. д.

killall

Еще один вариант программы kill. Используется для того, чтобы завершить работу процессов, носящих одно и то же имя. К примеру, в нашей системе запущено несколько программ файлового менеджера mc. Для того чтобы одновременно завершить работу этих программ, достаточно выполнить команду:

```
killall mc
```

Конечно, этим не ограничивается использование данной команды. С ее помощью можно отсылать сигналы группе одноименных процессов. Для получения более подробной информации по этой команде обращайтесь к ее *man*-странице.

nice

В операционной системе *Linux* у каждого процесса есть свой приоритет исполнения. Поскольку операционная система многозадачная, то для выполнения каждого процесса выделяется определенное количество времени в соответствии с приоритетом.

Программа nice позволяет запустить команду с предопределенным приоритетом выполнения, который задается в командной строке. При обычном запуске все задачи имеют один и тот же приоритет, и операционная система равномерно распределяет между ними процессорное время. Однако с помощью утилиты nice можно понизить приоритет какой-либо задачи, таким образом, предоставляя другим процессам больше процессорного времени. Повысить приоритет той или иной задачи имеет право только пользователь root. Синтаксис использования nice следующий:

```
nice -number command
```

Уровень приоритета процесса определяется параметром number, при этом большее его значение означает меньший приоритет процесса. Значение по умолчанию — 0, и number представляет собой число, на которое должен быть уменьшен приоритет.

К примеру, процесс top имеет приоритет, равный -5. Для того чтобы понизить приоритет выполнения процесса на десять, мы должны выполнить следующую команду:

```
nice 10 top
```

В результате процесс top получит приоритет, равный 5.

Только пользователь root может поднять приоритет того или иного процесса, используя для этого отрицательное значение параметра number.

renice

Программа renice, в отличие от программы nice, позволяет изменить приоритет уже работающего процесса. Формат запуска программы следующий:

```
renice -number PID
```

В общем, программа renice работает точно так же, как и nice. Уровень приоритета процесса определяется параметром number, при этом большее его значение означает меньший приоритет процесса. Значение по умолчанию — 10, и number представляет собой число, на которое должен быть уменьшен приоритет процесса.

Только пользователь root может поднять приоритет того или иного процесса, используя для этого отрицательное значение параметра number.

at

Одна из основных задач автоматизации администрирования операционной системы — выполнение программ в заданное время или с заданной периодичностью.

Для запуска одной или более команд в заранее определенное время используется команда at. В этой команде вы можете определить время и дату запуска той или иной команды. Команда at требует, по меньшей мере, двух параметров — время выполнения программы и запускаемую программу с ее параметрами запуска.

Приведенный ниже пример запустит команду ls на выполнение в *1 час 5 минут*. Каждая команда записывается на отдельной строке, т. е. завершается нажатием клавиши <Enter>, а по окончании ввода всех команд — <Ctrl>+<D> .

```
at 1:05
ls > file
```

Помимо времени, в команде at может быть также определена и дата запуска программы на выполнение.

batch

Команда batch в принципе аналогична команде at. Более того, batch представляет собой псевдоним команды at -b. Для чего необходима эта команда? Представьте, вы хотите запустить резервное копирование вечером. Однако в это время система очень занята, и выполнение резервирования системы практически парализует ее работу. Для этого и существует команда batch — ее использование позволяет операционной системе самой решить, когда наступает подходящий момент для запуска задачи в то время, когда система не сильно загружена.

Формат команды batch представляет собой просто список команд для выполнения, следующих в строках за командой; заканчивается список комбинацией клавиш <Ctrl>+<D>. Можно также поместить список команд в файл и перенаправить его на стандартный ввод команды batch.

cron и crontab

cron — это программа, выполняющая задания по расписанию, но, в отличие от команды at, она позволяет выполнять задания неоднократно. Вы определяете времена и даты, когда должна запускаться та или иная программа. Времена и даты могут определяться в минутах, часах, днях месяца, месяцах года и днях недели.

Программа cron запускается один раз при загрузке системы. При запуске cron проверяет очередь заданий at и задания пользователей в файлах crontab. Если для запуска не было найдено заданий — следующую проверку cron произведет через минуту.

Для создания списка задач для программы cron используется команда crontab. Для каждого пользователя с помощью этой команды создается его собственный crontab-файл со списком заданий, имеющий то же имя, что и имя пользователя.

Каждая строка в файле crontab содержит шаблон времени и команду. Команда выполняется тогда, когда текущее время соответствует приведенному шаблону. Шаблон времени состоит из пяти частей, разделенных пробелами или символами табуляции, и имеет вид:

```
минуты   часы   день_месяца   месяц   день_недели
```

Эти поля обязательно должны присутствовать в файле. Для того чтобы программа cron игнорировала какое-то поле шаблона времени, поставьте в нем символ звездочки (*).

Например, шаблон 10 01 01 * * говорит о том, что команда должна быть запущена в десять минут второго каждого первого числа любого (*) месяца, каким бы днем недели оно ни было. Рассмотрим описание полей cron-файла.

минуты
: Указывает минуты в течение часа. Значения от 0 до 59

часы
: Указывает час запуска задания. Значения от 0 до 23, где 0 — полночь

день_месяца
: Указывает день месяца, в который должна исполняться команда

месяц
: Указывает месяц, в который необходимо запускать задание. Значения лежат в пределах от 1 до 12, где 1 — январь

день_недели
: Указывает день недели — или как цифровое значение от 0 до 7 (0 и 7 означают воскресенье), или используя первые три буквы, например Mon

задание
: Командная строка для запуска задания

Ниже приведены несколько примеров команд, исполняемых программой cron:

01 * * * * /usr/bin/script – команда запускается в 1 минуту каждого часа
20 8 * * * /usr/bin/script – команда запускается каждый день в 8:20
00 6 * * 0 /usr/bin/script – команда запускается в 6 часов каждое воскресенье
40 7 1 * * /usr/bin/script – команда запускается в 7:40 каждое первое число месяца

Для создания и редактирования файла заданий для программы cron используется команда crontab. Прямое редактирование файла заданий не допускается.

Команда crontab имеет следующие параметры командной строки:

- -e — позволяет редактировать компоненты файла (при этом вызывается редактор, определенный в переменной EDITOR);
- -r — удаляет текущий crontab-файл из каталога;
- -l — используется для вывода списка текущих заданий.

Общение между процессами

Процессы могут общаться между собой при помощи каналов, сокетов, разделяемой памяти и т. д. Рассмотрим только один вариант: каналы.

Именованные каналы (*FIFO*: *First In First Out*). Данный вид канала создаётся с помощью mknod или mkfifo, и два различных процесса могут обратиться к нему по имени. Пример работы с *FIFO*:

в первом терминале (создаем именованный канал в виде файла pipe и из канала направляем данные с помощью конвейера в архиватор):

```
[root@proxy 1]# mkfifo pipe
[root@proxy 1]# ls -l
total 0
prw-r--r-- 1 root root 0 Nov 9 19:41 pipe
[root@proxy 1]# gzip -9 -c < pipe > out
```

во втором терминале (отправляем в именованный канал данные):

```
[root@proxy 1]# cat /path/to/file > pipe
```

в результате это приведет к сжатию передаваемых данных *gzip*-ом.

Программное управление процессами

Создание процесса: функция system()

Самый простой способ создать в *Linux* новый процесс — вызвать библиотечную функцию system(), объявленную в заголовочном файле stdlib.h. Эта функция просто передает команду, представленную в виде *Cu* строки, в командную оболочку /bin/sh. Функция system() имеет такой прототип:

```
int system(const char * COMMAND);
```

Данная функция выполняет переданную ей команду COMMAND (передавать можно команду с опциями) и ожидает ее завершения. Она возвращает целое число, называемое *статусом завершения потомка*. Если в качестве аргумента COMMAND передается пустой указатель, то функция system() возвращает ненулевое значение только в том случае, если командный процессор доступен. Таким способом можно проверить, поддерживается ли функция system() в данной системе. Функция system() возвращает код 127, если командная оболочка не может быть запущена для выполнения команды, и -1 в случае другой ошибки. Иначе system() вернет код завершения команды.

Следует отметить, что из соображений безопасности данной функцией нужно пользоваться с осторожностью.

Рассмотрим пример применение функции system():

```
#include <stdlib.h>
#include <stdio.h>

int main (void) {

        fprintf (stderr, "Start the program\n");
        system("uname");
        system("sleep 2");
        system("ps ax");
        system("sleep 2");
        fprintf (stderr, "Stop the program\n");

        return EXIT_SUCCESS;
}
```

Задание:

1. Написать исходный код примера, создать исполняемый файл и запустить написанную программу.
2. Написать свою программу, которая получает из командной строки команды (либо одну команду, либо одну команду с параметрами, либо несколько команд, разделенный точной с запятой (;)) и передает их для выполнения функции system(). Изучить

работу программы.

Завершение процесса

Для завершения текущего процесса существует стандартная функция exit():

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

После своего вызова функция exit() выполняет некоторые завершающие действия, а затем заставляет ядро прекратить процесс. У этой функции нет возможности возвращать ошибки — она вообще никогда не возвращает результат. Таким образом, следом за вызовом exit() нет смысла использовать никакие другие инструкции.

Параметр status используется для обозначения статуса выхода процесса. Другие программы, а также пользователь, работающий в оболочке, могут при необходимости проверять это значение. Например, в оболочке *Борна* (bash) этот статус завершения доступен с помощью специальной переменной $?.

Стандарт языка *Си* определяет две константы, которые следует использовать для полной переносимости программы на различные системы. Константа EXIT_SUCCESS применяется для обозначения того, что программа завершилась без проблем. Константа EXIT_FAILURE используется для указания того, что в программе была какая-нибудь проблема. В *Linux* значение 0 обычно представляет успешное завершение; любое ненулевое значение, например 1 или -1, соответствует неудаче.

Перед завершением процесса библиотечная функция exit() выполняет в указанном порядке следующие шаги:

1. вызывает функции, зарегистрированные с помощью atexit() или on_exit(), в порядке, обратном порядку регистрации;
2. сбрасывает все открытые стандартные потоки ввода-вывода;
3. удаляет все временные файлы, созданные при помощи функции tmpfile().

Эти шаги завершают всю работу, которую процесс должен проделать в пользовательском пространстве, и после этого вызов библиотечной функции exit() делает системный вызов _exit(), сообщая ядру, что оно теперь может выполнить оставшуюся часть процесса завершения:

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

Когда процесс завершается, ядро очищает все ресурсы, которые были созданы от имени процесса и которые больше не используются. После очистки ядро разрушает процесс и уведомляет предка о кончине его потомка.

Приложения могут напрямую вызывать функцию _exit(), но это редко имеет смысл, так как большинству приложений необходимо производить какую-то «уборку», которую обеспечивает обычный выход, например сбрасывать поток stdout.

Следует отметить, что в стандарт языка *Си* была добавлена функция _Exit(), обладающая поведением, полностью идентичным системному вызову _exit():

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

Следует отметить, что для возвращения из функции main() можно использовать как указанные ранее функции, так и оператор return <значение>. Возвращенное из main() значение автоматически передается обратно системе, от которой родительский процесс может его впоследствии получить.

В *Linux* реализована стандартная библиотечная функция atexit(), предназначенная для регистрации функций, которые должны автоматически вызываться при завершении процесса:

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

Успешный вызов atexit() регистрирует указанную при вызове функцию, как обработчик. Эта функция будет автоматически вызвана во время обычного завершения процесса, то есть когда процесс завершится при помощи вызова exit() или возврата из функции main(). Если процесс запускает функцию exec, список зарегистрированных функций очищается (так как функции более не существуют в адресном пространстве нового процесса). Если процесс завершается при помощи сигнала, то зарегистрированные функции не вызываются.

Указанная функция-обработчик не должна принимать параметров и не должна возвращать никакого значения. Прототип этой функции выглядит так:

```
void my_function(void);
```

Функции вызываются в порядке, обратном регистрации. Это означает, что функции хранятся в стеке и последняя попавшая в него выходит первой. Зарегистрированные функции не должны вызывать exit(), чтобы не начать бесконечную рекурсию. Если функция должна раньше запланированного прервать процесс завершения, то это необходимо делать с использованием системного вызова _exit(). Подобное поведение, однако, нежелательно.

Стандарт *POSIX* требует, чтобы atexit() поддерживала, по крайней мере, ATEXIT_MAX зарегистрированных функций, и это значение должно быть не меньше, чем 32. Точное максимальное значение можно получить при помощи функции sysconf() и значения _SC_ATEXIT_MAX:

```
long atexit_max;

atexit_max = sysconf(_SC_ATEXIT_MAX);
printf("atexit_max=%ld\n", atexit_max);
```

В случае успеха atexit() возвращает значение 0. В случае ошибки она возвращает значение -1 и соответствующим образом устанавливает значение переменной errno.

Рассмотрим пример программы, которая не делает полезной работы, но демонстрирует, как работает atexit():

```
#include <stdio.h>
#include <stdlib.h>

void callback1(void) { printf("Callback1 invoked\n"); }
void callback2(void) { printf("Callback2 invoked\n"); }
void callback3(void) { printf("Callback3 invoked\n"); }

int main(int argc, char **argv) {
```

```
        printf("Callback1 registration\n");
        if(atexit(callback1)) fprintf(stderr, "Callback1 registration failed\n");
        printf("Callback2 registration\n");
        if(atexit(callback2)) fprintf(stderr, "Callback2 registration failed\n");
        printf("Callback3 registration\n");
        if(atexit(callback3)) fprintf(stderr, "Callback3 registration failed\n");

        return EXIT_SUCCESS; /* exit(0); */
}
```

Пример подтверждает, что функции, зарегистрированные с помощью atexit(), запускаются в порядке, обратном порядку их регистрации.

Задание:

Проверить работу приведенной программы.

Существует эквивалент функции atexit(), и обычно *Linux* библиотека glibc поддерживает этот эквивалент:

```
#include <stdlib.h>
int on_exit(void (*function)(int , void *), void *arg);
```

Эта функция работает аналогично функции atexit(), но прототип регистрируемой функции отличается:

```
void my_function (int status, void *arg);
```

Аргумент status — это значение, передаваемое exit() или возвращаемое main(). Аргумент arg — это второй параметр, передаваемый on_exit(). Необходимо с большой внимательностью относиться к использованию этой функции, чтобы гарантировать, что на момент, когда функция в конечном итоге вызывается, в памяти, на которую указывает arg, содержатся допустимые данные. В связи с этими проблемами, а также проблемами переносимости, лучше не использовать эту функцию. Вместо нее следует использовать совместимую со стандартами функцию atexit().

Задание:

Написать свою версию упрощенного командного процессора. В самом начале работы выводится сообщение о том, кто работает с командным процессором (выводится полное имя пользователя, который в данный момент работает с программой). Затем на экран выводится приглашение вида [user]$, (здесь user – регистрационное имя текущего пользователя). Программа готова к приему команд командной оболочки. Если команде не выполнена, то на экран выводится сообщение, в противном случае — результат выполненной команды. Для завершения работы программы предназначена команда stop. При преждевременном завершении программы в результате какой-либо ошибки, а также при нормальном завершении программы, на экран выводится сообщение Wait 3 seconds..., программа ждет 3 секунды, затем очищает экран и завершает свою работу. Кроме того, программа должна работать с двумя опциями:

- -h и --help — опция без аргумента, предназначена вывода справочной информации по работе с программой;
- -b и --buffer опция с аргументом — целым числом, которое определяет размер

символьного буфера для хранения вводимой пользователем команды; если пользователь не указал размер при запуске программы, в качестве значения по умолчанию взять 127.

При написании программы особое внимание обратить на обработку ошибок. Продумать, какие библиотечные функции понадобятся для работы. (Можно воспользоваться: gets() <stdio.h> для чтения команды, sleep() <unistd.h> для задержки на указанное количество секунд и команду оболочки clear для очистки экрана)