Глава 8

ПРОЦЕССЫ

8.1. Основные понятия

В общем случае программа представляет собой набор инструкций процессора, представленный в виде файла. Чтобы программа могла быть запущена на выполнение, ОС должна сначала создать окружение или среду выполнения задачи, включающую в себя ресурсы памяти, возможность доступа к системе ввода/вывода и т. п. Совокупность окружения и области памяти, содержащей код и данные исполняемой программы, называется *процессом*. Процесс в ходе своей работы может находиться в различных состояниях (см. подразд. 8.3), в каждом из которых он особым образом использует ресурсы, предоставляемые ему ОС.

Два основных состояния процесса — это выполнение либо в режиме задачи, либо в режиме ядра. В первом случае происходит выполнение программного кода процесса, а во втором — системных вызовов, находящихся в адресном пространстве ядра.

Для управления процессами ОС использует системные данные, которые существуют в течение всего времени выполнения процесса. Вся совокупность этих данных образует *контекст процесса*, в котором он выполняется. Контекст процесса определяет состояние процесса в заданный момент времени.

С точки зрения структур, поддерживаемых ядром ОС, контекст процесса включает в себя следующие составляющие [15]:

- *пользовательский контекст* — содержимое памяти кода процесса, данных, стека, разделяемой памяти, буферов ввода/вывода;
- *регистровый контекст* — содержимое аппаратных регистров (регистр счетчика команд, регистр состояния процессора, регистр указателя стека и регистры общего назначения);
- *контекст системного уровня* — структуры данных ядра, характеризующие процесс. Контекст системного уровня состоит из статической и динамической части.

В статическую часть входят дескриптор процесса и пользовательская область (U-область).

Дескриптор процесса включает в себя системные данные, используемые ОС для идентификации процесса. Эти данные используются при построении таблицы процессов, содержащей информацию обо всех выполняемых в текущий момент времени процессах.

Дескриптор процесса содержит следующую информацию:

• *расположение и занимаемый процессом объем памяти* — обычно указывается в виде базового адреса и размера при непрерывном распределении процесса в памяти или списка начальных адресов и размеров блоков памяти, если процесс располагается в памяти несколькими фрагментами;

• *идентификатор процесса PID (Process IDentifier)* — уникальное целое число, находящееся обычно в диапазоне от 1 до 65 535, которое присваивается процессу в момент его создания;

• *идентификатор родительского процесса PPID (Parent Process IDentifier)* — идентификатор процесса, породившего данный. Все процессы в UNIX-системах порождаются другими процессами, например, при запуске программы на исполнение из командного интерпретатора ее процесс считается порожденным от процесса командного интерпретатора;

• *приоритет процесса* — число, определяющее относительное количество процессорного времени, которое может использовать процесс. Процессам с более высоким приоритетом управление передается чаще;

• *реальный идентификатор пользователя и группы*, запустивших процесс.

U-область содержит следующую информацию:

• указатель на дескриптор процесса;

• счетчик времени, в течение которого процесс выполнялся (т.е. использовал процессорное время) в режиме пользователя и режиме ядра;

• параметры последнего системного вызова;

• результаты последнего системного вызова;

• таблица дескрипторов открытых файлов;

• максимальные размеры адресного пространства, занимаемого процессом;

• максимальные размеры файлов, которые может создавать процесс.

Динамическая часть контекста системного уровня — один или несколько стеков, которые используются процессом при его выполнении в режиме ядра.

Для просмотра таблицы процессов может быть использована команда `ps`. Запущенная без параметров командной строки, она выведет все процессы, запущенные текущим пользователем. Достаточно полную для практического использования информацию

о таблице процессов можно получить, вызвав ps с параметрами aux, при этом будет выведена информация о процессах всех пользователей (a), часть данных, входящих в дескриптор и контекст процесса (u), а также будут выведены процессы, для которых не определен терминал (x):

```
$ ps aux
USER  PID  %CPU %MEM  VSZ  RSS  TTY      STAT START TIME COMMAND
root    1   0.0  0.0 1324  388  ?          S   Jul06 0:25 init[3]
root    2   0.0  0.0    0    0  ?          SW  Jul06 0:00 [eventd]
lp    594   0.0  0.0 2512  268  ?          S   Jul06 0:00  lpd
nick  891   0.1  0.1 2341  562 /dev/tty1 S   Jul06 0:18  bash
```

В этом примере команда ps aux выводит следующую информацию о процессах: имя пользователя, от лица которого запущен процесс (USER), идентификатор процесса (PID), процент процессорного времени, используемого процессом в данный момент времени (%CPU), процент занимаемой памяти (%MEM), общий объем памяти в килобайтах, занимаемый процессом (VSZ), объем постоянно занимаемой процессом памяти, которая может быть освобождена только по его завершению (RSS), файл терминала (TTY), состояние процесса (см. подразд. 8.3), дату старта процесса (START), количество используемого процессорного времени (TIME), полную строку запуска программы (COMMAND).

При работе процесса ему предоставляется доступ к ресурсам ОС — оперативной памяти, файлам, процессорному времени. Для распределения ресурсов между процессами и управления доступом к ресурсам в состав ядра ОС входит планировщик задач, а также используются механизмы защиты памяти и блокировки файлов и устройств.

Основная функция планировщика задач — балансировка нагрузки на систему между процессами, распределение процессорного времени согласно приоритету процессов.

Механизм защиты памяти запрещает доступ процесса к области оперативной памяти, занятой другими процессами (за исключением случая межпроцессного взаимодействия с использованием общей памяти).

Механизм блокировки файлов и устройств работает по принципу уникального доступа — если какой-либо процесс открывает файл на запись, то на этот файл ставится блокировка, исключающая запись в этот файл данных другим процессом.

8.2. Создание процесса. Наследование свойств

Запуск нового процесса в ОС UNIX возможен только другим, уже выполняющимся процессом. Запущенный процесс при этом

называется *процессом-потомком*, а запускающий процесс — *родительским процессом*. Процесс-потомок хранит информацию о родительском процессе в своем дескрипторе. Единственный процесс, не имеющий родительского процесса — это головной процесс ядра ОС — процесс `init`, имеющий `PID = 1`. Запуск этого процесса происходит при начальной загрузке ядра ОС.

В ОС UNIX существуют механизмы для создания процесса и для запуска новой программы. Для этого используется системный вызов `fork()`, создающий новый процесс, который является почти точной копией родительского:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

Между процессом-потомком и процессом-родителем существуют следующие различия [8, 15]:

- потомку присваивается уникальный идентификатор `PID`, отличный от родительского;
- значение `PPID` процесса-потомка устанавливается в значение `PID` родительского процесса;
- процесс-потомок получает собственную таблицу файловых дескрипторов, т.е. файлы, открытые родителем, не являются таковыми для потомка;
- для процесса-потомка очищаются все ожидающие доставки сигналы (см. подразд. 9.3);
- временна́я статистика выполнения процесса-потомка в таблицах ОС обнуляется;
- блокировки памяти и записи, установленные в родителе, не наследуются.

При этом процесс наследует следующие свойства:

- аргументы командной строки программы;
- переменные окружения;
- реальный (UID) и эффективный (EUID) идентификаторы пользователя, запустившего процесс;
- реальный (GID) и эффективный (EGID) идентификатор группы, запустившей процесс;
- приоритет;
- установки обработчиков сигналов (см. подразд. 9.3).

Функция `fork()` возвращает значение `−1` в случае, если порождение процесса-потомка окончилось неудачей. Причиной этого может служить одна из следующих ситуаций:

- ограничения максимального числа процессов для текущего пользователя, накладываемые командной оболочкой. Текущие ограничения, например, при использовании оболочки BASH можно узнать или изменить с помощью команды `ulimit −u`;

• переполнение максимально допустимого количества дочерних процессов. Узнать ограничения, накладываемые системой на максимальное количество дочерних процессов можно с помощью команды `getconf CHILD_MAX`;

• переполнение максимального числа процессов в системе. Это ограничение накладывается самой системой и узнать его можно с помощью команды `sysctl fs.file-max` или просмотрев содержимое файла `/proc/sys/fs/file-max`;

• исчерпана виртуальная память, определяемая размерами оперативной памяти и раздела подкачки.

В случае успешного выполнения функции `fork()` она возвращает PID процесса-потомка родительскому процессу и 0 процессу-потомку. Для хранения идентификаторов процессов используется тип переменной `pid_t`. В большинстве систем этот тип идентичен типу `int`, но непосредственное использование типа `int` не рекомендуется из соображений обеспечения совместимости с будущими версиями UNIX.

Для получения значения PID и PPID используются функции `getpid()` и `getppid()` соответственно. Тип возвращаемого ими значения — `pid_t`:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

Поскольку выполнение обоих процессов — и родителя, и потомка — после вызова функции `fork()` продолжается со следующей за ней команды, необходимо разграничивать программный код, выполняемый каждым из этих процессов. Самый очевидный способ для этого — выполнять нужные участки кода в разных ветках условия, проверяющего код возврата функции `fork()`. Ставший уже классическим пример такой проверки приведен ниже:

```
#include <sys/types.h>
#include <unistd.h
int main(void)
{
 pid_t pid;
 switch (pid = fork())
 {
  case -1:
   /* Неудачное выполнение fork() — pid равен -1 */
   perror("Unsuccessful fork() execution\n");
   break;
```

```
   case 0:
    /* Тело дочернего процесса */
    /* pid = 0 — это дочерний процесс. */
    /* В нем значение pid инициализируется нулем */
    sleep(1);
    printf("CHILD: Child spawned with PID = %d\n",
      getpid());
    printf("CHILD: Parent PID = %d\n", getppid());

    /* Завершение работы дочернего процесса */
    _exit(0);
   default:
    /* Тело процесса-родителя */
    /* так как pid>0, выполняется родительский */
    /* процесс, получивший pid потомка */
    printf("PARENT:Child spawned with PID=%d\n",
       pid);
    printf("PARENT:Parent PID=%d\n", getpid());
  }
  /* Тело процесса-родителя после обработки fork() */
  /* Если бы в case 0 не был указан _exit(0), то */
  /* потомок бы тоже выполнил идущие следом команды */
exit(0);
}
```

Программа выведет следующие строки (идентификаторы процессов могут различаться):

```
PARENT:Child spawned with PID=2380
PARENT:Parent PID=2368
CHILD: Child spawned with PID = 2380
CHILD: Parent PID = 1
```

В приведенном примере программы необходимо обратить внимание на следующий момент: если не обеспечить выход из процесса-потомка внутри соответствующего варианта `case`, то потомок, закончив выполнение программного кода внутри `case`, продолжит выполнение кода, находящегося после закрывающей скобки конструкции `switch()`. В большинстве случаев такое поведение необходимо пресекать. Для явного выхода из процесса с заданным кодом возврата используется функция `_exit(<код возврата>)`. Вызов этой функции вызывает уничтожение процесса и выполнение следующих действий:

- отключаются все сигналы, ожидающие доставки (см. подразд. 9.3);
- закрываются все открытые файлы;

• статистика использования ресурсов сохраняется в файловой системе proc;

• извещается процесс-родитель и переставляется PPID у потомков;

• состояние процесса переводится в «зомби» (см. подразд. 8.3).

Другие способы завершения работы процесса рассмотрены в следующих разделах.

Для завершения процесса-потомка рекомендуется использовать именно функцию _exit() вместо функции exit(). Функция _exit() очищает структуры ядра, связанные с процессом — дескриптор и контекст процесса. В отличие от нее функция exit() в дополнение к указанным действиям устанавливает все структуры данных пользователя в начальное состояние. В результате этого может возникнуть ситуация, в которой структуры данных процесса-родителя будут повреждены. Функция exit() может быть вызвана только в функциях, выполняемых процессом-родителем.

В большинстве случаев функция fork() используется совместно с одной из функций семейства exec...(). При последовательном вызове этих функций возможно создание нового процесса и запуск из него новой программы.

В результате выполнения функции exec...() программный код и данные процесса заменяются на программный код запускаемой программы. Неизменными остаются только идентификатор процесса PID и идентификатор родительского процесса PPID.

Если выполнение функции семейства exec...() завершилось неуспешно, это может быть обусловлено следующими причинами:

• путь к исполняемому файлу превышает значение системного параметра PATH_MAX, элемент пути к файлу превышает значение системного параметра NAME_MAX или в процессе разрешения имени файла выяснилось, что число символических ссылок в пути превысило число SYMLOOP_MAX. Узнать значения этих параметров можно с помощью команды getconf –a;

• файл, для которого вызывается функция, не является исполняемым, не является обычным файлом или не существует.

Список параметров, передаваемых в запускаемый процесс, слишком большой. Максимально допустимое количество параметров вычисляется на основе максимально допустимой длины командной строки. Согласно стандарту POSIX данное значение должно быть не меньше 4 096. В реальных системах обычно это число больше на несколько порядков. Чтобы узнать это ограничение для системы, можно использовать команду getconf ARG_MAX. Параметр ARG_MAX и содержит максимально допустимое количество символов в командной строке. В реальности накладываются дополнительные ограничения на максимальную длину

выполняемой команды. Согласно стандарту POSIX, максимально допустимую длину выполняемой команды можно вычислить с помощью команды `expr `getconf ARG_MAX` - `env|wc -c` - 2048`. Кроме того, системой накладываются и ограничения на длину одного параметра.

В приведенном ниже примере процесс-родитель порождает новый процесс, который запускает на выполнение программу `/bin/ls` с параметром `-l`. Поскольку происходит полная замена кода процесса-потомка, то вызывать функцию `_exit()` не обязательно:

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

int main(void)
{
  pid_t pid;
  int status;
  if (fork() == 0)
  {

    /* Тело потомка */
    execl("/bin/ls", "/bin/ls", "-l", 0);
  }
  /* Продолжение тела родителя */
  wait(&status);
  printf("Child return code %d\n", WEXITSTATUS(status));
  return 0;
}
```

Программа выведет следующее:

```
total 17
-rwxr-xr-x  1 nick users 9763 Oct 11 15:41 a.out
-rwxrwxrwx  1 nick users 986 Oct 11 15:20 fork1.c
-rwxrwxrwx  1 nick users 321 Oct 11 15:40 fork2.c
Child return code 0
```

Процесс-родитель после создания потомка ожидает его завершения при помощи функции `wait()`:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

Эта функция ожидает окончания выполнения процесса-потомка (если их было порождено несколько — любого из них). Возвра-

щаемое функцией значение — PID завершившегося процесса. Передаваемое функции по ссылке значение статуса представляет собой число, кодирующее информацию о коде возврата потомка и его состоянии на момент завершения. Для просмотра интересующей информации необходимо воспользоваться одним из макроопределений `WEXIT…()`. Например, макроопределение `WEXISTATUS()` возвращает номер кода возврата, содержащегося в значении статуса.

Если процесс порождает множество потомков и возникает необходимость в ожидании завершения конкретного потомка, используется функция `waitpid()`:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

Первый аргумент этой функции — `pid` процесса, завершения которого мы ожидаем, второй — значение статуса. Третий параметр определяет режим работы функции.

Если третий параметр равен 0, то выполнение процесса приостанавливается до тех пор, пока хотя бы один потомок не будет завершен. Если в качестве третьего параметра передается константа `WNOHANG`, то значение статуса присваивается только в том случае, если потомок уже завершил свое выполнение, в противном случае выполнение родителя продолжается. Если указан параметр `WNOHANG` и потомок еще не завершен, то функция `waitpid()` вернет 0:

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
void main(void)
{
  pid_t childPID;
  pid_t retPID = 0;
  int status;

  if ( (childPID = fork()) == 0)
  {
    /* Тело потомка */
    execl("/bin/ls", "/bin/ls", "-l", 0);
  }
  while (!retPID) /* Продолжение тела родителя */
  {
    retPID = waitpid(childPID, &status, WNOHANG);
  }
```

```
  printf("Child return code %d", WEXITSTATUS(status));
}
```

Программа выведет следующее:

```
total 17
-rwxr-xr-x   1 nick users 9763 Oct 11 15:41 a.out
-rwxrwxrwx   1 nick users 986 Oct 11 15:20 fork1.c
-rwxrwxrwx   1 nick users 321 Oct 11 15:40 fork2.c
Child return code 0
```

8.3. Состояния процесса. Жизненный цикл процесса

Между созданием процесса и его завершением процесс находится в различных состояниях в зависимости от наступления некоторых событий в ОС. Сразу же после создания процесса при помощи функции `fork()` он находится в состоянии «Создан» — запись в таблице процессов для него уже существует, но внутренние структуры данных процесса еще не инициализированы. Как только первоначальная инициализация процесса завершается, он переходит в состояние «Готов к запуску». В этом состоянии процессу доступны все необходимые ресурсы, кроме процессорного времени, и он находится в очереди задач, ожидающих выполнения. Как только процесс выбирается планировщиком, он переходит в состояние «Выполняется в режиме ядра», т. е. выполняет программный код ядра ОС, обрабатывающий последнее изменение состояния процесса.

Из этого состояния процесс может перейти в состояние «Выполняется в режиме задачи», в котором он будет выполнять уже свой собственный программный код. При каждом системном вызове процесс будет переходить в состояние «Выполняется в режиме ядра» и обратно (рис. 8.1).

Поскольку системные вызовы могут выполняться для получения доступа к определенным ресурсам, а ресурсы могут оказаться недоступными, то в этом случае из состояния «Выполняется в режиме ядра» процесс переходит в состояние «Ожидание», в котором он освобождает процессорное время и ожидает освобождения ресурса. После того, как ресурс становится доступным процессу, процесс захватывает его и переходит в состояние «Готов к запуску» и опять начинает ожидать выбора планировщиком процессов.

В состояниях «Выполняется в режиме ядра» и «Выполняется в режиме задачи» планировщик может передать управление другому процессу. При этом процесс, у которого забрали управление, будет переведен в состояние «Готов к запуску».

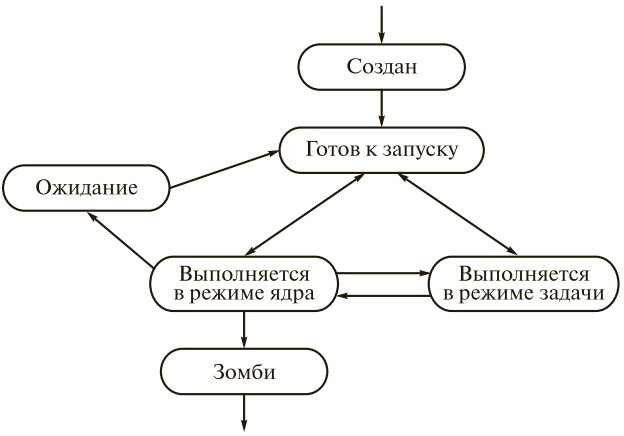

Рис. 8.1. Жизненный цикл процесса

При переключении активного процесса происходит переключение контекста, при котором ядро сохраняет контекст старого процесса и восстанавливает контекст процесса, которому передается управление. После завершения процесса он будет переведен в состояние «Зомби», т.е. память, занимаемая процессом, будет освобождена, а в таблице процессов останется информация о коде возврата процесса. Полностью завершить процесс можно при помощи вызова функции `wait()` процессом-родителем.

8.4. Терминал. Буферизация вывода

По умолчанию системная библиотека ввода/вывода передает данные на устройство терминала не сразу по выполнению системного вызова (например, `printf()`), а по мере накопления в специальной области памяти некоторого количества текста.

Такая область памяти получила название *буфера вывода*, а процесс накопления данных перед непосредственной их передачей на устройство называется *буферизацией*.

Для каждого процесса выделяется свой буфер ввода/вывода. Данные из него передаются на устройство по поступлению команды от ядра ОС или по заполнению буфера. Использование промежуточного буфера позволяет сильно сократить количество обращений к физическому устройству терминала и в целом ускоряет работу с ним. Системная библиотека использует три типа буферизации:

• *полная буферизация* — передача данных на физическое устройство терминала выполняется только после полного заполнения буфера;

• *построчная буферизация*, при которой передача данных на физическое устройство терминала производится после вывода одной строки текста (т.е. последовательности символов, оканчивающейся переводом строки или последовательности символов, длина которой равна ширине терминала);

• *отсутствие буферизации*, при котором данные сразу передаются на устройство, не накапливаясь в буфере.

При параллельном выводе текста несколькими процессами на терминал выводимый ими текст накапливается в отдельном буфере каждого процесса, а момент поступления данных на терминал определяется моментом посылки ОС команды сброса содержимого буферов на устройство. В результате неодинаковой длины выводимых разными процессами текстов и неравномерного предоставления им процессорного времени при включенной буферизации вывод таких параллельно выполняемых процессов может произвольно перемешиваться. То, каким образом данные будут перемешаны, зависит от момента передачи данных из буферов процессов на устройство терминала.

Например, на рис. 8.2 на линии времени показано последовательное изменение состояния буферов ввода/вывода двух процессов, последовательно выводящих на экран арабские цифры от 1 до 9 и латинские буквы от *A* до *F*.

Вывод каждого символа в этих процессах производится отдельным оператором `printf()`. Жирными стрелками вниз на ри-

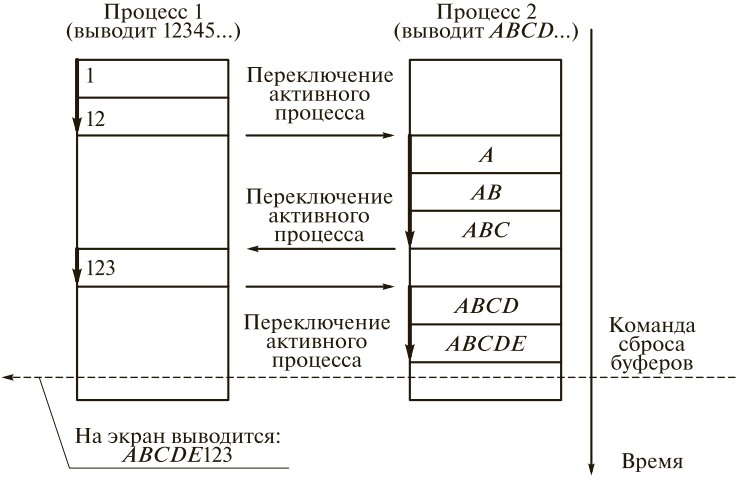

Рис. 8.2. Сброс буферов ввода/вывода параллельно выполняемых процессов:

— время выполнения процесса

сунке показано время, в течение которого процесс активен (использует процессорное время), высота каждого прямоугольника задает общее время жизни процесса.

Поскольку накопление данных в буферах происходит постепенно и между отдельными вызовами функции `printf()` может происходить переключение активного процесса, данные будут накапливаться в буферах неравномерно. Объем текста, выводимого на экран в момент поступления команды сброса буфера, будет зависеть от того, какой объем успел накопиться в буфере, а последовательность вывода фрагментов текста на экран будет определяться последовательностью сброса буферов на экран.

Для того чтобы избежать этой проблемы, можно отключить буферизацию. Это минимизирует задержку между выполнением команды вывода данных и реальным появлением текста на терминале.

Управление буферизацией производится функцией `setvbuf()`:

```
#include <stdio.h>
int setvbuf (FILE *stream, char *buf, int type, size_t
size);
```

Аргумент `stream` задает имя потока ввода/вывода, для которого изменяется режим буферизации, `buf` — задает указатель на область памяти, в которой хранится буфер, `type` — определяет тип буферизации и может принимать следующие значения:

`_IOFBF` — полная буферизация;
`_IOLBF` — построчная буферизация;
`_IONBF` — отсутствие буферизации.

Аргумент `size` задает размер буфера, на который ссылается указатель `buf`.

Для отключения буферизации стандартного потока вывода можно использовать буфер нулевого размера и вызвать функцию `setvbuf()` следующим образом:

```
setvbuf(stdout,(char*)NULL,_IONBF,0);
```

Рекомендуется вызывать эту функцию в начале работы любой программы, порождающей процессы, которые выводят данные на один и тот же терминал.

Контрольные вопросы

1. Как идентифицируется и какими параметрами характеризуется процесс в ОС?
2. Что происходит при выполнении функции `fork()`?
3. В чем различия системных вызовов `wait()` и `waitpid()`?

4. В каких случаях процесс находится в состоянии «Готов к запуску»?

5. Для чего нужно управление буферизацией при выводе данных несколькими процессами на один терминал?

6. Напишите программу, которая порождает 10 процессов-потомков, каждый из которых завершает свое выполнение с кодом возврата, равным номеру своего процесса. После завершения всех потомков программа должна выводить сумму кодов возврата всех потомков.

7. Напишите программу, которая последовательно порождает 10 процессов-потомков (т.е. каждый порожденный потомок порождает следующего потомка). При этом завершение выполнения потомков должно производиться в обратном порядке.