Основы управления процессами в ОС GNU/Linux

Цели работы:

1. Познакомиться с основами управления процессами в GNU/Linux, включая порождение, завершение дочернего процесса, а также основы синхронизации между родительскими и дочерними процессами.
2. Получить практический опыт создания многопроцессного приложения

1. Введение в GNU/Linux

1.1. Понятие процесса

В UNIX все, что выполняется, каждая вводимая команда и запускаемая программа является «процессом». Каждый процесс объединяет программный код, значения данных в переменных программы и другие элементы, необходимые для работы программы (значения регистров процессора, стек, и др.).

Любой процесс UNIX, в свою очередь, может “порождать” (то есть запускать) другие процессы. Это ставит два процесса в отношения “родитель – потомок”. В результате, среда процессов Unix-подобной системы имеет иерархическую структуру, аналогичную дереву каталогов файловой системы. На вершине дерева процессов находится процесс, представляющий собой экземпляр программы init (или её аналога), которая является прародительницей всех системных и пользовательских процессов.

Операционная система присваивает процессам пятизначные идентификационные номера, называемые pid (process identifier, то есть идентификатор процесса). Каждый процесс в системе имеет уникальный pid.

Пиды в конечном итоге повторяются, потому что все возможные числа израсходованы, и нумерация процессов может пойти по кругу. Но в любой момент времени в системе не существует двух процессов с одинаковым pid, поскольку именно pid используется Unix для отслеживания каждого процесса).

В стандартной библиотеке C определен тип `pid_t` – он представляет собой целое, способное вместить в себе pid (Process IDentifier — числовой ID процесса в системе).

Рассмотрим функцию, которая возвращает pid процесса, содержащего запущенную программу. Эта функция определена вместе с pid_t в unistd.h и sys/types.h:

```
pid_t getpid (void)
```

Напишем программу, которая выводит в стандартный вывод свой pid:

```
#include <unistd.h>
```

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

#include <stdio.h>

int main() {

   pid_t pid;

   pid = getpid();

   printf("pid присвоенный процессу - %d\n", pid);

   return 0;

}
```

Сохранив программу, скомпилируем ее.

```
gcc -o print_pid print_pid.c
```

Программа выведет положительное число, и, если продолжать запускать ее, это число будет постоянно увеличиваться на единицу. Если число отличается больше, чем на единицу, значит в перерыве между запусками был создан другой процесс. Это можно увидеть например, если выполнить `ps` между двумя запусками `print_pid`.

Системная утилита `ps` (сокращение от “process status”, состояние процесса) для просмотра информации, связанной с запущенными в системе процессами. Команда ps выводит список запущенных в данный момент процессов и их pid, а также некоторую дополнительную информацию, в зависимости от указанных команде параметров.

1.2 Порождение дочернего процесса

Функция, которая создает новый процесс, выглядит следующим образом:

```
pid_t fork(void)
```

При ее вызове происходит разветвление выполнения процесса. Число, которое она возвращает — это `pid`. Текущий процесс дублируется в родительском и дочернем, которые будут выполняться, чередуясь с другими выполняющимися процессами. Решение, какой процесс должен выполняться, принимается планировщиком, и он не принимает во внимание, является ли процесс родительским или дочерним.

Оба процесса содержат какой-то программный код, и после вызова `fork()` этот программный код у родительского и дочернего процесса один и тот же. однако очевидно, что процессы должны выполнить различный набор действий. Это достигается следующим алгоритмом:

```
РАЗВЕТВИТЬ
ЕСЛИ ТЫ ДОЧЕРНИЙ ПРОЦЕСС ВЫПОЛНИТЬ (...)
ЕСЛИ ТЫ РОДИТЕЛЬСКИЙ ПРОЦЕСС ВЫПОЛНИТЬ (...)
```

который представляет собой код нашей программы, написанный на некотором метаязыке.Функция fork возвращает '0' в дочерний процесс и pid дочернего процесса в родительский. На языке C мы получим

```
int main() {
   pid_t pid;
   pid = fork();
   if (pid == 0)
   {  КОД ДОЧЕРНЕГО ПРОЦЕССА }
    КОД РОДИТЕЛЬСКОГО ПРОЦЕССА
}
```

Напишем действующий пример кода:

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
int main()
{  pid_t pid;
   int i;
   pid = fork();
   if (pid == 0){
      for (i = 0; i < 8; i++){
         printf("-ДОЧЕРНИЙ-\n");
      }
      return(0);
   }
   for (i = 0; i < 8; i++){
      printf("+РОДИТЕЛЬСКИЙ+\n");
   }
   return(0);
}
```

Программа сделает разветвление и оба процесса, родительский и дочерний, выведут текст на экран.

Вставляя задержку случайной длины перед каждым вызовом printf, можно нагляднее увидеть эффект многозадачности: мы сделаем это при помощи функций sleep и rand.

```
sleep(rand()%4)
```

Это заставит программу "заснуть" на случайное число секунд: от 0 до 3 (напомним, что % возвращает остаток от целочисленного деления).

1.3 Синхронизация родительского и дочернего процессов

Часто родительскому процессу необходимо обмениваться информацией с дочерними или хотя бы синхронизироваться с ними, чтобы выполнять операции в нужное время. Первый способ синхронизации процессов — функции wait и waitpid:

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

`pid_t wait(int *status)` - приостанавливает выполнение текущего процесса до завершения какого-либо из его процессов-потомков.

`pid_t waitpid (pid_t pid, int *status, int options)` - приостанавливает выполнение текущего процесса до завершения заданного процесса или проверяет завершение заданного процесса.

Если процесс уже завершился, то приостановка текущего процесса не происходит.

Значение параметров функции `waitpid()`:

- `pid` — это pid ожидаемого процесса:
    - Если `pid` > 0, то он задает PID процесса, завершение которого ожидается/проверяется функцией `waitpid()`
    - Если `pid` = 0, то `waitpid()` ожидает/проверяет завершение любого процесса той группы[1], к которой принадлежит текущий процесс.
    - Если `pid` < 0, то waitpid ожидает/проверяет завершение любого процесса - своего потомка.
- `status` — указатель на целое, которое будет содержать статус дочернего процесса (NULL, если эта информация не нужна)
- `options` — это набор опций, задающих режим поведения waitpid. Может задаваться одним из следующих значений или их логическим ИЛИ:
    - `WNOHANG` - не приостанавливать текущий процесс, если проверяемый процесс не завершился;
    - `WUNTRACED` - не приостанавливать текущий процесс для потомков, которые завершились, но о состоянии которых еще не доложено

Если `status` не равен `NULL`, то функции `wait()` и `waitpid()` сохраняют информацию о статусе в переменной, на которую указывает `status`. Этот статус можно проверить с помощью следующих макросов (они принимают в качестве аргумента буфер (типа `int`), а не указатель на буфер:

- `WIFEXITED(status)` не равно нулю, если дочерний процесс успешно завершился.
- `WIFEXITEDSTATUS(status)` возвращает восемь младших битов значения, которое вернул завершившийся дочерний процесс. Эти биты могли быть установлены в аргументе функции `exit()` или в аргументе оператора return функции `main()`. Этот макрос можно использовать, только если `WIFEXITED` вернул ненулевое значение.
- `WIFSIGNALED(status)` возвращает истинное значение, если дочерний процесс завершился из-за необработанного сигнала.

---

[1] В одну группу включаются процессы, имеющие общего предка, идентификатор группы процесса можно изменить с помощью системного вызова setpgrp. (см. man 2 getgid)

- `WTERMSIG(status)` возвращает номер сигнала, который привел к завершению дочернего процесса. Этот макрос можно использовать, только если `WIFSIGNALED` вернул ненулевое значение.
- `WIFSTOPPED(status)` возвращает истинное значение, если дочерний процесс, из-за которого функция вернула управление, в настоящий момент остановлен; это возможно, только если использовался флаг `WUNTRACED`.
- `WSTOPSIG(status)` возвращает номер сигнала, из-за которого дочерний процесс был остановлен. Этот макрос можно использовать, только если `WIFSTOPPED` вернул ненулевое значение.

Значение `options = 0` определяет переход в ожидание, если проверяемый процесс не завершился.

Рассмотрим пример программы, где родительский процесс создает дочерний и ждет его завершения.

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
int main() {
   pid_t pid;
   int i;
   pid = fork();
   if (pid == 0) {
      for (i = 0; i < 14; i++) {
         sleep (rand()%2);
         printf("-ДОЧЕРНИЙ-\n");
      }
      return 0;
   }
   sleep (rand()%4);
    printf("+РОДИТЕЛЬСКИЙ+ Ожидаю завершения выполнения дочернего
процесса...\n");
   waitpid (pid, NULL, 0);
   printf("+РОДИТЕЛЬСКИЙ+ ...завершен\n");
   return 0;
}
```

ПРИМЕЧАНИЕ: Для любого порождённого процесса его *родитель должен рано или поздно вызвать* функции `wait()/waitpid()`, чтобы дождаться завершения дочернего процесса и получить его статус (см. `man 3 wait`). Если этого не сделать, и родительский процесс завершится, то в системе появится “зомби-процесс”: запись о процессе, который уже завершил свою работу, но его статус не был получен родителем. Такие “зомби” занимают память, и с ними ничего нельзя сделать без перезагрузки системы.

1.4 Функции и программы в процессах

Чтобы в качестве дочернего процесса вызвать функцию, достаточно вызвать ее после ветвления:

```
pid = fork();
   if (pid == 0)
       // если выполняется дочерний процесс, то вызовем функцию
pid=process(arg);
// выход из процесса
exit(0);
```

Часто в качестве дочернего процесса необходимо запускать другую программу. Для этого применяются функции семейства exec:

```
#include <unistd.h>

extern char **environ;
int execl( const char *path, const char *arg, ...);
int execlp( const char *file, const char *arg, ...);
int execle( const char *path, const char *arg , ..., char * const
envp[]);
int execv( const char *path, char *const argv[]);
int execvp( const char *file, char *const argv[]);
```

Семейство функций exec() заменяет образ текущей программы в памяти на образ запускаемой программы.

Первый аргумент всех функций является указателем на символьную строку, содержащую полное имя исполняемого файла (`path`). Для функций execlp и execvp имя файла может задаваться без пути (`file`). Если первый аргумент этих функций не содержит символа "`/`", то файл ищется в каталогах, определенных в переменной окружения PATH.

Аргументы `arg`,... функций `execl`, `exelp`, `execle` составляют список указателей на символьные строки, содержащие параметры, передаваемые программе. По соглашениям первый элемент этого списка должен содержать имя программного файла. Список параметров должен заканчиваться пустым указателем - NULL или `(char *)0`.

В функциях execv и execvp параметры, передаваемые программе, передаются через массив символьных строк. Аргумент argv является указателем на этот массив.

Аргумент envp функции `execle` также является массивом указателей на символьные строки. Эти строки представляют собой окружение – среду для нового образа процесса. Последний элемент массива envp должен быть пустым указателем.

При успешном выполнении системного вызова exec возвращаемое значение функции оказывается некому принять. При ошибках выполнения функция возвращает -1 и устанавливает `errno`.

Пример – программа подсчета пробелов.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <wait.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

// Название: main
// Описание: главная программа
// Входные параметры: список имен файлов для обработки
// Выходные параметры: нет
int main(int argc, char *argv[]) {
   int i, pid[argc], status, stat;
   char arg[20];
   // для всех файлов, перечисленных в командной строке
   if (argc<2) {
      printf("Usage: file textfile1 textfile2 ...\n");
      exit(-1);
   }
   for (i = 1; i< argc; i++) {
      // printf("File %s\n",argv[i]);
      // запускаем дочерний процесс
      strcpy(arg,argv[i]); // копируем строку
      pid[i] = fork();
      if (pid[i] == 0) {
         // если выполняется дочерний процесс
         // вызов функции счета количества пробелов в файле
         if (execl("./file","file",arg, NULL)<0) {
            printf("ERROR while start processing file
%s\n",argv[i]);
            exit(-2);
       }
       else printf( "processing of file %s started (pid=%d)\n",
argv[i],pid[i]);
       // выход из процесса
   }
   // если выполняется родительский процесс
   }
   sleep(1);
   // ожидание окончания выполнения всех запущенных процессов
   for (i = 1; i< argc; i++) {
      status = waitpid(pid[i], &stat, WNOHANG);
      if (pid[i] == status) {
         printf("File %s done,
result=%d\n",argv[i],WEXITSTATUS(stat));
```

```
         // if (pid != 0) while (wait(&status)>0);
      }
   }
   return 0;
}

//----------------------------
//file
//----------------------------
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
// Название: processFile
// Описание: обработка файла, подсчет кол-ва пробелов
// Входные параметры: fileName - имя файла для обработки
// Выходные параметры: кол-во пробелов в файле
int main(int argc, char *argv[]) {
   int handle, numRead, total= 0;
   char buf;
   if (argc<2) {
      printf("Usage: file textfile\n");
      exit(-1);
   }
   ...................
   printf("(PIO: %d), File %s, spaces = %d\n", getpid(), argv[1],
total);
 return( total);
}
```

Если необходимо узнать состояние порожденного процесса при его завершении и возвращенное им значение, то используют макрос `WEXITSTATUS`, передавая ему в качестве параметра статус дочернего процесса.

```
status=waitpid(pid,&status,WNOHANG);
if (pid == status) {
   printf("PID: %d, Result = %d\n", pid, WEXITSTATUS(status));
}
```

1.5 Уничтожение процессов

Для уничтожения процесса служит функция kill:

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

Значение параметров функции `kill()`:

- Если pid > 0, то он задает PID процесса, которому посылается сигнал. Если `pid = 0`, то сигнал посылается всем процессам той группы, к которой принадлежит текущий процесс.
- `sig` - тип сигнала. Некоторые типы сигналов в Linux:
    - `SIGKILL` – этот сигнал приводит к немедленному завершению процесса. Этот сигнал процесс не может игнорировать.
    - `SIGTERM` – этот сигнал является запросом на завершение процесса.
    - `SIGCHLD` – система посылает этот сигнал процессу при завершении одного из его дочерних процессов.

Задание для самостоятельного выполнения

Выполните следующее задание, **сохраняя в отчет текст выполняемых команд и/или скриншоты с результатами их работы**:

- Изучите теоретический материал, приведенный выше
    - При необходимости, воспользуйтесь статьями Википедии, более подробно разъясняющими концепцию строк в C (https://ru.wikipedia.org/wiki/%D0%9D%D1%83%D0%BB%D1%8C-%D1%82%D0%B5%D1%80%D0%BC%D0%B8%D0%BD%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B0 и https://ru.wikipedia.org/wiki/Strncpy).
    - Ознакомьтесь с особенностями работы команды ps (https://ru.wikipedia.org/wiki/Ps). Выведите в терминал список всех процессов, запущенных в системе, и поясните информацию, выданную командой для какого-либо из системных процессов
- Выполните запуск дочерних процессов по следующему алгоритму :
    - родительский процесс запускает параллельно 4 дочерних процесса, а затем через некоторое время принудительно их все завершает (для задержки использовать, например, функцию `sleep(n)`, где `n` - количество секунд);
    - ри из четырёх порождённых процессов должны в бесконечном цикле выводить на экран свой PID с поясняющим сообщением (текст по вашему выбору). Вывод информации выполнять с секундной задержкой после каждой итерации.
    - один из четырёх порождённых процессов (номер этого процесса по счёту порождённых указан в таблице ниже в столбце **fork**) вместо вывода своего PID выполняет с помощью функции `exec()` программу, указанную в таблице в столбце **exec.**
    - ** **(задание повышенной сложности)** модифицировать программу: передавать родительскому процессу через аргументы командной строки число порождаемых процессов и аргументы для команды, запускаемой одним из дочерних процессов.

No варианта	**fork**	**exec**

1	1	ls
2	2	ps
3	3	pwd
4	4	whoami
5	3	df
6	2	ls
7	4	ps
8	1	pwd
9	2	whoami
10	3	time
11	4	ls
12	2	ps
13	3	pwd
14	3	whoami
15	1	date
16	2	ls
17	3	ps
18	4	pwd
19	4	whoami
20	2	free
21	1	ls

Результат выполнения работы: компилируемый .c файл с текстом программы, PDF-файл с кратким описанием задания, текстом программы и демонстрацией её работы (скриншоты или скопированный текстовый вывод программы).