Системные вызовы взаимодействия с процессами в Linux.

Процессы — это вторая, после файлов, фундаментальная абстракция в системе *Linux* и других *POSIX* системах. Понятие процессы (*process*) связано с выполняющимся кодом конечных программ. Процессы состоят из данных, ресурсов и состояния. Процессы как исполняемые конечные программы. Исполняемый формат содержит метаданные, а также несколько разделов (section) кода и данных. Примерами таких разделов являются:

- текстовый раздел (*text section*) - содержатся исполняемый код и данные (разел доступен для чтения и исполняемый);
- раздел данных (*data section*) - инициализированные данные, такие, как переменные *C* с уже определенными значениями (раздел доступен для чтения и записи);
- раздел *bss* (*bss section*) - неинициализированные глобальные данные.

Процесс ассоциируется с рядом системных ресурсов, которые выделяются и управляются ядром операционной системы. Как правило процессы запрашивают ресурсы и манипулируют ими только при помощи системных вызовов. Ресурсы включают в себя таймеры, ожидающие сигналы, открытые файлы, сетевые соединения, аппаратные ресурсы и механизмы межпроцессорного взаимодействия IPC. Ресурсы процесса вместе с относящимися к процессу данными и статистикой хранятся внутри ядра в дескрипторе процесса (process descriptor).

Каждый процесс состоит из одного или нескольких потоков выполнения (*thread of execution,* поток*, thread*). Большинство процессов включают в себя только один поток (однопоточные процессы, *singlethreaded*). Процессы, содержащие несколько потоков, называются многопоточными (*multithreaded*).

РАБОТА В ОПЕРАЦИОННОЙ СИСТЕМЕ LINUX УФА 2014 О.Д. Лянцев, А.В. Казанцев

Каждый процесс в *Linux* помечается уникальным идентификатором (*PID, process identifer*). Идентификаторы – это 16-разрядные числа, назначаемые последовательно по мере создания процессов. У всякого процесса имеется также родительский процесс за исключением специального суперсервера *init* с идентификатором *1*. Таким образом, все процессы *Linux* организованы в виде сложной иерархии – т.н. дерево процессов, на вершине которой находится процесс *init*. Иерархию процессов можно увидеть, выполнив команду *ps –axf*. К атрибутам процесса относится идентификатор его родительского процесса (*PPID, parent process identifer*). Гарантируется, что в любой конкретный момент времени значение *pid* уникально.

Также отметит, что у процесса бездействия (*idle process*), который ядро «выполняет», когда нет никаких других доступных для выполнения процессов, идентификатор pid всегда равен 0.

Для управления процессами предусмотрены следующие функции:

- создание, завершение процесса, получение информации о процессе:
  fork, exit, getpid, getppid и т. д.;
- синхронизация процессов:
  signal, kill, sleep, alarm, wait, paus,semop, semctl, semcreate и т. д.;
- создание информационного канала, разделяемой памяти, очереди сообщений и работа с ними: *pipe, mkfifo, read, write, msgget, shmget, msgctl, shmctl* и т. д.

Для системных вызовов работы с процессами в *Linux* предусмотрен тип данных *pid_t* (определяется в <*sys/types.h*>). Программа может узнать идентификатор своего собственного процесса с помощью системного вызова *getpid*(), а идентификатор своего родительского процесса с помощью системного вы-зова *getppid*(). Отметим, что при каждом вызове программа сообщает о разный идентификатор, так как каждый раз запускается новый процесс. Если программа вызывается из одного и того же интерпретатора команд, то родительский идентификатор оказывается одинаковым. Для создания новых процессов предусмотрено два способа, первый способ основан на применении функции *system*(), а второй на применении функций *fork*() и *exec*(). Первый способ менее безопасен и используется редко.

Функция *fork*(), создает дочерний процесс, который является точной копией родительского процесса. Однако, для дочернего выделяется новый идентификатор *pid*, значение которого отличается от значения *pid* его предка; в качестве *ppid* устанавливается *pid* его родительского процесса; статистика ресурсов для потомка сбрасывается до нуля; любые ожидающие сигналы удаляются и не наследуются потомком; никакие захваченные блокировки файлов не наследуются потомком.

Задание 1.

Изучите работу следующего кода.

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main() {
        pid_t child_pid;
        printf("ID процесса основной программы: %d\n", (int) getpid());

        child_pid = fork();
```

```
        if (child_pid){
                printf(«Это родительский процесс, с ID %d\n», (int)getpid());
                printf("Дочерний процесс, с ID %d\n", (int)child_pid);
        }else
                printf(«Дочерний процесс с ID %d\n», (int)getpid());
        return 0;
}
```

1. Напишите комментарии к коду.

Семейство функций *exec*() (<*unistd.h*>), создает процессы, которые превращаются в экземпляры другой программы. Когда программа вызывает функцию *exec()*, ее выполнение немедленно прекращается и начинает работу новая программа. Предусмотрены несколько вариантов функций *exec*().

- Функции, в название которых присутствует суффикс '*p*' (*execvp(), execlp()*), принимают в качестве аргумента имя программы и ищут эту программу в каталогах, определяемых переменной среды *PATH*. Всем остальным функциям нужно передавать полное путевое имя программы.
- Функции, в названии которых присутствует суффикс '*v*' (*execv(), execvp(), execve()*), принимают список аргументов программы в виде массива строковых указателей, оканчивающегося *NULL*-указателем.
- Функции с суффиксом '*l*' (*excevl(), execlp(), execlve()*), принимают список аргументов переменного размера.
- Функции, в названии которых присутствует суффикс '*e*' (*execve(), execle()*), в качестве дополнительного аргумента принимают массив переменных среды. Этот массив содержит строковые указатели и оканчивается пустым указателем. Каждая строка должна иметь вид «Переменная = значение».

Отметим, что на практике для запуска одной программы из другой: сначала с помощью функции *fork*() создается дочерний процесс, затем в нем вызывается функция *exec*(). Это позволяет главной программе продолжать выполнение в родительском процессе.

Также отметим семейство функций *wait*() необходимое для работы с процессами.

Функция *pid_t wait (int *status)* приостанавливает выполнение родительского процесса до тех пор, пока дочерний процесс не прекратит выполнение или до появления сигнала, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик ошибок. Если дочерний процесс к моменту вызова функции уже завершился (так называемый «зомби»), то функция немедленно возвращается, а системные ресурсы, связанные с дочерним процессом, освобождаются.

Функция *pid_t waitpid (pid_t pid, int *status, int options)* позволяет дождаться завершения конкретного дочернего процесса; *wait3(…)* возвращает информацию о статистике использования центрального процессора завершившемся дочерним процессом; *wait4(…)* позволяет задать дополнительную информацию о том, каких процессов следует дождаться.

При использовании *waitpid* параметр *pid* может принимать несколько значений:

- *pid < -1* означает, что нужно ждать любой дочерний процесс, чей идентификатор группы процессов равен абсолютному значению *pid*.
- *pid = -1* означает ожидать любой дочерний процесс; функция *wait* ведет себя точно так же.
- *pid = 0* означает ожидать любой дочерний процесс, чей идентификатор группы процессов равен таковому у текущего процесса.
- *pid > 0* означает ожидать дочерний процесс, чей идентификатор равен *pid*.

Значение *options* создается путем битовой операции ИЛИ над константами *WNOHANG* (вернуть управление немедленно, если ни один дочерний процесс не завершил выполнение) и *WUNTRACED* (возвращать управление также для остановленных дочерних процессов, о чьем статусе еще не было сообщено).

В *Linux* определяется стандартная функция *void exit (int status)* для завершения текущего процесса (<*stdlib.h*>). Параметр *status* используется для обозначения статуса выхода процесса. Значения *EXIT_SUCCESS* и *EXIT_FAILURE* определяются в качестве способов представления успеха и неудачи. В *Linux* 0 обычно представляет успех; ненулевое значение, например, 1 или -1, соответствует неудаче. Следовательно, успешный выход определяется одной простой строкой: *exit (EXIT_SUCCESS)*.

Задание 2.

1. Изучите следующий код и дайте к нему комментарии.

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<unistd.h>
int spawn(char* program, char** arg_list){
        pid_t child_pid;
        child_pid = fork();
        if(child_pid)
                return child_pid;
        else{
                execvp (program, arg_list);
                fprintf (stderr, "an error process in execvp\n");
                abort();
        }
}
int main(){
        int child_status;
        char* arg_list[] = {"ls","-l","/",NULL};
        spawn ("ls", arg_list);
```

```
    wait (&child_status);
    printf("done\n");
    return 0;
}
```

2. Написать программу, создающую два дочерних процесса с использованием двух вызовов *fork*(). Родительский и два дочерних процесса должны выводить на экран свой pid и pid родительского процесса и текущее время в формате: часы : минуты : секунды : миллисекунды. Используя вызов *system*(), выполнить команду *ps -x* в родительском процессе.
3. Найти запущенные приложением процессы в списке запущенных процессов.
4. Завершите текущий процесс в приложении.

Задание 3

1. Разработать и отладить процедуру, выполняющую следующие действия:
    a. последовательный запуск четырех процессов соответственно системными вызовами *execl , execlp , execv , execvp*;
    b. вывод на консоль идентификаторов этих процессов;
    c. закрыть процессы в обратном порядке.
2. Разработать и отладить процедуру, выполняющую следующие действия:
    a. ответвление текущего процесса;
    b. запуск трех процессов потомков;
    c. вывод на консоль идентификаторов этих процессов;
    d. закрытие процессов в том же порядке.

Межпроцессорное взаимодействие.

Все процессы в *Linux* выполняются в раздельных адресных пространствах. В ряде случаев необходимо объединение процессов или организация некоторой связи между ними. Для организации взаимодействия между процессами (межпроцессного взаимодействия, *IPC*) необходимо использовать специальные методы, среди которых следующие.

- Общие файлы – для ускорения работы следует использовать файлы, отображаемые в памяти при помощи системного вызова *mmap()* (<*sys/mman.h*>).
- Разделяемая память – память к которой могут обратиться все процессы, системные вызовы *shm_open* и *shm_unlink* (<*sys/mman.h*>).
- Очереди сообщений (queue) – запись сообщений от одного процесса (клиент) другому или другим (приемники) при помощи механизма *FIFO* или других. Открытие очереди *mq_open*() (<*mqueue.h*>); создает описание открытой очереди и ссылающийся на него дескриптор типа *mqd_t*. Процесс, который использует очередь, должен предварительно получить разрешение на

использование общего сегмента памяти с помощью системных запросов, для работы с ней обращение к ОС.

- Сигналы – при вызове механизма процесс прерывает исполнение, и управление передается функции-обработчику сигнала. По окончании обработки сигнала процесс может возобновить регулярное исполнение. Для передачи сиголов по *PID* используется функция *kill*() (<*signal.h*>). Сигналы можно передавать только в рамках полномочий пользователя (или от лица суперпользователя). Для установки собственного обработчика сигналов предусмотрены функция *void (*signal (int sig, void (*handler) (int)))(int)* - для изменения реакции процесса (*handler*) на сигнал (*sig*); и *int sigaction(int sig, const struct sigaction *act, struct sigaction *oldact)* для изменения действий (*act*) процесса при получении соответствующего сигнала.
- Каналы (*pipe*) - аналог файла, однако, в канале прочитанная информация немедленно удаляется из него и не может быть прочитана повторно. Каналы используются для однонаправленной передачи сообщений (например, от процесса родителя своим потомкам). Канал представляет собой область памяти, недоступную пользовательским процессам напрямую, зачастую организованную в виде кольцевого буфера (хотя существуют и другие виды организации). При этом в канале есть два указателя: на запись и на чтение. Для создания нового экземпляра такого кольцевого буфера внутри операционной системы используется системный вызов *pipe*() (<*unistd.h*>). Для чтения и записи используются вызовы *read*() и *write*() как при работе с обычными файлами и вызов *close*() для освобождения системных ресурсов.
- Именованные каналы *FIFO* (именованный *pipe*). В отличии от обычного канала данные о расположении *FIFO* в адресном пространстве и его состоянии процессы могут получать через файловую систему, а не через родственные связи. Для этого при создании именованного *pipe* создается *FIFO* файл, обращаясь к которому процессы могут получить интересующую их информацию. Для создания FIFO используется системный вызов *mknod*() (в некоторых версиях *UNIX mkfifo()*). Для открытия и работы с файлом *FIFO* используются те же функции, как и для обычных файлов. Важно понимать, что файл типа *FIFO* не служит для размещения на диске информации, которая записывается в именованный *pipe*. Эта информация располагается внутри адресного пространства операционной системы, а файл является только меткой, создающей предпосылки для ее размещения.

- Семафоры – счетчики доступа к общим ресурсам со стороны разных процессов. Цель использования семафоров возможность синхронизации процессов при доступе к совместно используемым ресурсам. Возможны бинарный семафор и семафор со счетчиком. Цель первого дать доступ к ресурсу для ограниченного числа процессов. Бинарный светофор блокирует ресурс если тот уже занят кем-то.

В случае сигнального обмена передается минимальное количество информации, достаточное для извещения процесса о наступлении события. При канальном обмене объем передаваемой информации в единицу времени ограничен пропускной способностью доступа к файлу. При использовании разделяемой памяти обмен информацией наиболее быстрый в одной вычислительной системе, но требует повышенной осторожности.

Существуют различные причины кооперации процессов

- повышение скорости работы (один процесс в ожидании, другой выполняет полезную работу, направленную на решение общей задачи);
- совместное использование данных (использование различными процессами одной и той же динамической базы данных или разделяемого файла);
- модульная конструкция какой-либо системы (например, микроядерный способ построения ОС, когда взаимодействие процессов осуществляется путем передачи сообщений через микроядро);
- для удобства работы пользователя (например, при одновременном редактировании и отладке программы, процессы редактора и отладчика должны взаимодействовать).

Наиболее общим понятием *IPC* является ключ, хранимый в общесистемной таблице и обозначающий объект межпроцессного взаимодействия, доступный нескольким процессам. Обозначаемый ключом объект может быть очередью сообщений, набором семафоров или сегментом разделяемой памяти. Ключ имеет тип *key_t* (<*sys*/*types*.*h*>). Ключ используется для создания объекта межпроцессного взаимодействия или получения доступа к существующему объекту.

Задание 4.

1. Создайте файл, в который напишите некоторый текст в том числе заданное слово (например, ваше имя).
2. Выполните процедуру команды

   cat “filename” | grep “name”,

   где *filename* и *name* имя файла и заданное слово соответственно.

3. Данный тип взаимодействия относится к неименованным каналам. Дайте объяснение почему.
4. Выполните команды

   who|sort

   а затем

   sort|who

   объясните разницу в результате с точки зрения работы каналов.

Задание 5.

1. Изучите код следующей программы.

```
#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#define TIMEOUT 10

int f1(int);
int f2(int);
int f3(int);

int pid0, pid1, pid2;

void main(void)
{
        setpgrp();
        pid0 = getpid();
        pid1 = fork();
        if (pid1 == 0) { /* process 1 */
            signal(SIGUSR1, f1);
                pid1 = getpid();
                pid2 = fork();
                if (pid2 < 0 )
                        puts("Fork error");
                if (pid2 > 0)
                        for(;;);
                else{ /* process 2 */
                    signal(SIGUSR2, f2);
                    pid2 = getpid();
                    kill(pid1,SIGUSR1);
                    for (;;);
                }
        }
        else{ /* process 0 */
            signal(SIGALRM, f3);
                alarm(TIMEOUT);
                pause();
        }
        exit(0);
}

int f1(int signum) {
        signal(SIGUSR1, f1);
        printf("Process 1 (%d) has got a signal from process 2 (%d)\n",pid1,pid2);
        sleep(1);
        kill(pid2, SIGUSR2);
        return 0;
}
```

```
int f2(int signum) {
        signal(SIGUSR2, f2);
        printf("Process 2 (%d) has got a signal from process 1 (%d)\n",pid2,pid1);
        sleep(1);
        kill(pid1, SIGUSR1);
        return 0;
}

int f3(int signum) {
        printf("End of job - %d\n", pid0);
        kill(0, SIGKILL);
        return 0;
}
```

2. Ответьте на вопросы, что делает данная программа, какой тип межсетевого взаимодействия используется.
3. Напишите комментарии в коде.

Задание 6.

1. Изучите следующий код

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#define BUF_SIZE 1024
int main (int argc, char * argv[])
{
        int descriptors[2]; //оба дескриптора канала; 0 ? для чтения, 1 ? для записи
        int pid; //переменная для вызова fork()
        pipe(descriptors); //возврат дескрипторов файлов для чтения и данных из канала
        // и записи данных в канал
        pid = fork(); //вызов функции ?раздвоения? процесса на родительский и дочерний
        if ( pid > 0 ) { //родительский процесс
                char symb[] = "Hello!\n";
                int length=sizeof(symb); // размер массива символов
                close(descriptors[0]); //закрытие дескриптора для чтения
                write(descriptors[1], symb, length + 1); //запись строки в файл, по дескриптору для
                записи
                close(descriptors[1]); //разрыв связи с дескриптором для записи
        }else{ //дочерний процесс
                char buf[BUF_SIZE];
                int len;
                close(descriptors[1]); //закрытие дескриптора для записи
                while ((len = read(descriptors[0], buf, BUF_SIZE)) != 0) //чтение файла, куда была
                //была записана строка
                        write(2, buf, len); //вывод строки на экран
                close(descriptors[0]); //разрыв связи с дескриптором чтения
        }
        return 0; //код возврата
}
```

2. Напишите комментарии и опишите алгоритм, по которому работает программа.
3. Реализуйте систему меж процессного взаимодействия, когда родительский процесс осуществляет вывод данных, являющихся результатом работы запущенной в дочернем процессе утилиты.