Санкт-Петербургский государственный политехнический университет

Факультет технической кибернетики

Кафедра компьютерных систем и программных технологий

Отчёт по лабораторной работе №3

«Управление процессами и нитями в ОС UNIX»

Работу выполнил студент группы № 4081/12

Дорофеев Юрий Владимирович

Работу принял преподаватель _____

Малышев Игорь Алексеевич

г. Санкт-Петербург

2012

1. Содержание отчета

2. Цель работы

Изучить основные принципы управления процессами и нитями с ОС LINUX.

3. Выполнение работы

3.1. Порождение и запуск процессов

Системные вызовы:

fork() – системный вызов, с помощью которого создается процесс, являющийся точной копией породившего его процесса (с теми же инструкциями и данными).

exec() – системный вызов для загрузки новой программы, при котором новый процесс не порождается, а исполняемый код процесса полностью замещается кодом запускаемой программы.

wait() – системный вызов, при котором процесс приостанавливается до тех пор, пока один из непосредственно порожденных им процессов не завершится, или пока порожденный процесс, который находится в режиме трассировки, не остановится при достижении точки прерывания. Преждевременный выход из системного вызова *wait* происходит, если был получен сигнал; если же порожденный процесс остановился или завершился раньше вызова *wait*, то возврат происходит немедленно.

exit() – системный вызов для завершения программы.

sleep() – системный вызов, задерживающий выполнение программы на n секунд.

Разработаны следующие программы родителя и потомка (*father.c* и *son.c*):

father.c:

```
#include <stdio.h>
main(){
int pid, ppid, status;
pid=getpid();
ppid=getppid();
printf("\n\n FATHER PARAM: pid=%i ppid=%i \n", pid,ppid);
if(fork()==0) execl("son.out","son.out", NULL);
system("ps xf");
wait(&status);
}
son.c
#include <stdio.h>
main(){
int pid,ppid;
pid=getpid();
ppid=getppid();
printf("\n SON PARAM: pid=%i ppid=%i \n",pid,ppid);
sleep(15);
}
```

Выполнение работы программы *father.out*:

```
FATHER PARAM: pid=2401 ppid=2318
SON PARAM: pid=2402 ppid=2401
PID TTY STAT TIME COMMAND
2012 ? S 0:00 \_ /usr/bin/python /usr/share/system-config-printer/
2035 ? Sl 0:00 \_ update-notifier
2121 ? Sl 0:11 /usr/lib/firefox-3.6.18/firefox-bin /media/KINGSTON/O
2094 ? Sl 0:01 gnome-terminal
2097 ? S 0:00 \_ gnome-pty-helper
2098 pts/0 Ss 0:00 \_ bash
2316 pts/0 S+ 0:00 \_ script
2317 pts/0 S+ 0:00 \_ script
2318 pts/1 Ss 0:00 \_ bash -i
2401 pts/1 S+ 0:00 \_ ./father.out
2402 pts/1 S+ 0:00 \_ son
2403 pts/1 S+ 0:00 \_ sh -c ps xf
2404 pts/1 R+ 0:00 \_ ps xf
2041 ? Sl 0:00 /usr/lib/notify-osd/notify-osd
```

Идентификатор процесса – 2402.
Идентификатор родительского процесса – 2401.
Father.out запускается из интерпретатора *bash*, поэтому *bash* является родителем по отношению к *father.out*. *father.out* в свою очередь запускает *son*, поэтому является родителем для него.
Запустим программу *father.out* в фоновом режиме:

```
$ ./father.out &
[1]2405
FATHER PARAM: pid=2405 ppid=2318
:~/Roshchina/OS/labProc$
SON PARAM: pid=2406 ppid=2405
PID TTY STAT TIME COMMAND
2012 ? S 0:00 \_ /usr/bin/python /usr/share/system-config-printer/
2035 ? Sl 0:00 \_ update-notifier
2121 ? Sl 0:11 /usr/lib/firefox-3.6.18/firefox-bin /media/KINGSTON/O
2094 ? Sl 0:01 gnome-terminal
2097 ? S 0:00 \_ gnome-pty-helper
2098 pts/0 Ss 0:00 \_ bash
2316 pts/0 S+ 0:00 \_ script
2317 pts/0 S+ 0:00 \_ script
2318 pts/1 Ss+ 0:00 \_ bash -i
2405 pts/1 S 0:00 \_ ./father.out
2406 pts/1 S 0:00 \_ son
2407 pts/1 S 0:00 \_ sh -c ps xf
2408 pts/1 R 0:00 \_ ps xf
2041 ? Sl 0:00 /usr/lib/notify-osd/notify-osd
```

При запуске в фоновом режиме управление сразу возвращается bash.

3.2. Взаимодействие родственных процессов

Изменяя длительности выполнения процессов и параметры системных вызовов, рассмотрим 3 ситуации и выведем соответствующие таблицы процессов:

1. процесс-отец запускает процесс-сын и ожидает его завершения;

Фактически такая ситуация рассмотрена в предыдущем пункте работы.

2. процесс-отец запускает процесс-сын и, не ожидая его завершения, завершает свое выполнение.

Зафиксируем изменение родительского идентификатора процесса-сына;

father.c :

```
#include <stdio.h>
main(){
int pid, ppid, status;
pid=getpid();
ppid=getppid();
printf("\n\n FATHER PARAM: pid=%i ppid=%i \n", pid,ppid);
if(fork()==0) execl("son.out","son.out", NULL);
system("ps xf");
//wait(&status);
}
```

son.c:

```
#include <stdio.h>
main(){
int pid,ppid;
pid=getpid();
ppid=getppid();
printf("\n SON PARAM: pid=%i ppid=%i \n",pid,ppid);
sleep(15);
printf("\n SON PARAM: pid=%i ppid=%i \n",getpid(),getppid());
}
```

```
$./father.out
FATHER PARAM: pid=8927 ppid=6671
SON PARAM: pid=8928 ppid=8927
PID TTY STAT TIME COMMAND
4431 ? Ss 0:00 /bin/sh /usr/bin/startkde
4678 ? S 0:00 \_ kwrapper4 ksmserver
7810 ? S 0:00 /usr/lib64/notification-daemon
6669 ? Sl 0:05 /usr/bin/konsole
6671 pts/1 Ss 0:00 \_ /bin/bash
8927 pts/1 S+ 0:00 \_ ./father.out
8928 pts/1 S+ 0:00 \_ son
8929 pts/1 R+ 0:00 \_ ps xf
$
SON PARAM: pid=8928 ppid=1
```

Здесь рассматривается ситуация потери процессом son родителя. Сначала его родителем был процесс *father*, но после того, как он завершился, не дождавшись завершения *son, son* стал дочерним процессом процесса с идентификатором 1, т.е. *init*.

3. процесс-отец запускает процесс-сын и не ожидает его завершения; процесс-сын завершает свое выполнение.

Зафиксируем появление процесса-зомби, для этого включим команду *ps* в программу *father.c*.

son.c :

```
#include <stdio.h>
main(){
int pid,ppid;
pid=getpid();
ppid=getppid();
printf("\n SON PARAM: pid=%i ppid=%i \n",pid,ppid);
}
$ ./father.out
FATHER PARAM: pid=10421 ppid=6671
SON PARAM: pid=10422 ppid=10421
PID TTY STAT TIME COMMAND
4431 ? Ss 0:00 /bin/sh /usr/bin/startkde
4678 ? S 0:00 \_ kwrapper4 ksmserver
7810 ? S 0:00 /usr/lib64/notification-daemon
6669 ? Sl 0:05 /usr/bin/konsole
6671 pts/1 Ss 0:00 \_ /bin/bash
10421 pts/1 S+ 0:00 \_ ./father.out
10422 pts/1 Z+ 0:00 \_ [son.out] <defunct>
10423 pts/1 R+ 0:00 \_ ps xf
```

Здесь рассматривается ситуация завершения дочернего процесса, когда родительский процесс этого еще не ожидает. После завершения дочерний поток переводится в состояние «зомби», т.е. прекращает выполнение и ожидает освобождения своих структур родительским процессом.

Переводим результат программ не только в терминал, но и в файл:

```
$./father.out >> file.txt
```

3.3. Управление процессами посредством сигналов

С помощью команды *kill -l* ознакомимся с перечнем сигналов, поддерживаемых процессами.

```
$ kill -l
1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP
6) SIGABRT 7) SIGBUS 8) SIGFPE 9) SIGKILL 10) SIGUSR1
11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO 30) SIGPWR
31) SIGSYS 34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47)
SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-
12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
```

kill() – системный вызов, который посылает сигнал процессу или группе процессов, которые заданы идентификатором *pid*. Посылаемый сигнал определяется аргументом *sig* и является либо одним из списка сигналов, заданного в системном вызове *signal()*, либо 0.

signal() - системный вызов, который позволяет вызывающему процессу выбрать один из возможных способов реакции на получение определенного сигнала. Аргументы *sig* и *func* специфицируют, соответственно, сигнал и выбор.

Подготовьте программы следующего содержания:

1. процесс *father* порождает процессы *son1*, *son2*, *son3* и запускает на исполнение программные коды из соответствующих исполнительных файлов;
2. далее родительский процесс осуществляет управление потомками, для этого он генерирует сигнал каждому пользовательскому процессу;
3. в пользовательских процессах-потомках необходимо обеспечить:
    - для *son1* - реакцию на сигнал по умолчанию;
    - для *son2* - реакцию игнорирования;
    - для *son3* - перехватывание и обработку сигнала.

Тексты программ:

father.c :

```
#include <stdio.h>
int main(){
int son1, son2, son3;
if((son1=fork())==0) execl("son1","son1", NULL);
if((son2=fork())==0) execl("son2","son2", NULL);
if((son3=fork())==0) execl("son3","son3", NULL);
system ("ps s");
kill(son1, SIGUSR2);
kill(son2, SIGUSR2);
kill(son3, SIGUSR2);
system("ps s");
kill(son1, SIGTERM);
kill(son2, SIGTERM);
kill(son3, SIGTERM);
return 0;
}
```

son1.c :

```
#include <stdio.h>
#include <signal.h>
main()
{
signal (SIGUSR2, SIG_DFL);
while (1) {
pause();
}
}
son2.c
#include <stdio.h>
#include <signal.h>
main(){
signal (SIGUSR2, SIG_IGN);
while (1) {
pause();
}
}
```

son3.c :

```
#include <stdio.h>
#include <signal.h>
void temp_handler (int signum)
{
printf ("Task work!\n");
}
int main(){
signal (SIGUSR2, temp_handler);
while (1) {
pause();
}
}
```

Запуск программы *father.out*:

```
UID PID PENDING BLOCKED IGNORED CAUGHT STAT TTY TIME COMMAND
1018 2098 00000000 00010000 00384004 4b813efb Ss pts/0 0:00 bash
1018 2318 00000000 00010000 00384004 4b813efb Ss pts/1 0:00 bash -i
1018 3062 00000000 00010000 00000006 00000000 S+ pts/1 0:00 ./father
1018 3063 00000000 00000000 00000000 00000000 S+ pts/1 0:00 son1
1018 3064 00000000 00000000 00000800 00000000 S+ pts/1 0:00 son2
1018 3065 00000000 00000000 00000000 00000800 S+ pts/1 0:00 son3
1018 3066 00000000 00000000 00000000 00000002 S+ pts/1 0:00 sh -c p
1018 3067 00000000 00000000 00000000 73d3fef9 R+ pts/1 0:00 ps s
Task work!
UID PID PENDING BLOCKED IGNORED CAUGHT STAT TTY TIME COMMAND
1018 2098 00000000 00010000 00384004 4b813efb Ss pts/0 0:00 bash
1018 2318 00000000 00010000 00384004 4b813efb Ss pts/1 0:00 bash -i
1018 3062 00000000 00010000 00000006 00000000 S+ pts/1 0:00 ./father
1018 3063 00000000 00000000 00000000 00000000 S+ pts/1 0:00 son1
<defunct>
1018 3064 00000000 00000000 00000800 00000000 S+ pts/1 0:00 son2
1018 3065 00000000 00000000 00000000 00000800 S+ pts/1 0:00 son3
1018 3066 00000000 00000000 00000000 00000002 S+ pts/1 0:00 sh -c p
1018 3067 00000000 00000000 00000000 73d3fef9 R+ pts/1 0:00 ps s
```

Создается три дочерних процесса и им посылается сигнал SIGUSR2. В первом процессе этот сигнал обрабатывается обработчиком по умолчанию, поэтому он завершается. Второй процесс игнорирует этот сигнал, поэтому продолжает работать. Третий процесс переопределяет обработчик и выводит сообщение при получении сигнала и продолжает работу.

Организуем посылку сигналов любым двум процессам, находящимся в разных состояниях: активном и пассивном, фиксируя моменты посылки и приема каждого сигнала с точностью до секунды.

father.c :

```
#include <stdio.h>
#include <signal.h>
main(){
int son1, son2;
int pid, ppid, status;
pid=getpid();
ppid=getppid();
printf ("\n\n FATHER PARAM: pid=%i ppid=%i\n", pid, ppid);
if ((son1=fork())==0) execl("son1.out", "son1.out", NULL);
```

```
if ((son2=fork())==0) execl("son2.out", "son2.out", NULL);
system ("ps -s >> res.txt");
printf("Before son1\n");
system("date");
kill(son1, SIGUSR2);
sleep(2);
printf("Before son2\n");
system("date");
kill(son2, SIGUSR2);
system ("ps -s >> res.txt");
wait(&status);
wait(&status);
}
```

son1.c :

```
#include <stdio.h>
#include <signal.h>
void sig1 (int p)
{
printf("After son1\n");
system("date");
}
main() {
int pid, ppid;
pid=getpid();
ppid=getppid();
printf ("\n\n SON1 PARAM: pid=%i ppid=%i\n", pid,ppid);
signal (SIGUSR2, sig1);
sleep (5);
}
```

son2.c :

```
#include <stdio.h>
#include <signal.h>
void sig2 (int p)
{
printf("After son2\n");
system("date");
}
main() {
int pid, ppid;
pid=getpid();
ppid=getppid();
printf ("\n\n SON2 PARAM: pid=%i ppid=%i\n", pid,ppid);
signal (SIGUSR2, sig2);
sleep (5);
}
```

```
$ ./father.out
FATHER PARAM: pid=2522 ppid=1634
SON1 PARAM: pid=2523 ppid=2522
SON2 PARAM: pid=2524 ppid=2522
Before son1
Сб. нояб. 19 15:01:00 SAMT 2012
After son1
Сб. нояб. 19 15:01:00 SAMT 2012
Before son2 8
Сб. нояб. 19 15:01:02 SAMT 2012
After son2
Сб. нояб. 19 15:01:02 SAMT 2012
```

Реакция на сигналы появляется через несколько секунд.

```
UID PID PENDING BLOCKED IGNORED CAUGHT STAT TTY TIME COMMAND
1000 1634 0000000000000000 0000000000010000 0000000000384004 000000004b813efb
Ss pts/0 0:00 bash
1000 2544 0000000000000000 0000000000010000 0000000000000006 0000000000000000
S+ pts/0 0:00 ./father1.out
1000 2545 0000000000000000 0000000000000000 0000000000000000 0000000000000800
S+ pts/0 0:00 son1.out
1000 2546 0000000000000000 0000000000000000 0000000000000000 0000000000000800
R+ pts/0 0:00 son2.out
1000 2547 0000000000000000 0000000000000000 0000000000000000 0000000000000002
S+ pts/0 0:00 sh -c ps -s >> res.txt
1000 2548 0000000000000000 0000000000000000 0000000000000000 0000000073d3fef9
R+ pts/0 0:00 ps -s
UID PID PENDING BLOCKED IGNORED CAUGHT STAT TTY TIME COMMAND
1000 1634 0000000000000000 0000000000010000 0000000000384004 000000004b813efb
Ss pts/0 0:00 bash
1000 2544 0000000000000000 0000000000010000 0000000000000006 0000000000000000
S+ pts/0 0:00 ./father1.out
1000 2545 0000000000000000 0000000000000000 0000000000000000 0000000000000800
Z+ pts/0 0:00 [son1.out] <defunct>
1000 2546 0000000000000000 0000000000000000 0000000000000000 0000000000000800
R+ pts/0 0:02 son2.out
1000 2556 0000000000000000 0000000000000000 0000000000000000 0000000000000002
S+ pts/0 0:00 sh -c ps -s >> res.txt
1000 2558 0000000000000000 0000000000000000 0000000000000000 0000000073d3fef9
R+ pts/0 0:00 ps -s
```

После реакции на сигнал активный процесс продолжает своё выполнение, а пассивный оказывается в состоянии зомби.

Запуск 4 задач в фоновом режиме:

```
$ sleep 100 & sleep 200 & sleep 300 & sleep 400 &
```

Вывод запущенных задач:

```
$ jobs
[1] Выполняется sleep 100 &
[2] Выполняется sleep 200 &
[3]- Выполняется sleep 300 &
[4]+ Выполняется sleep 400 &
```

Перевод задачи 3 в приоритетный режим:

```
$ fg %3
sleep 300
```

Завершение задачи 2:

```
$ kill -s kill %2
[2]- Убито sleep 200
```

Системный вызов *nice* увеличивает поправку к приоритету вызывающего процесса на величину *incr*. Поправка к приоритету - неотрицательное число; чем оно больше, тем ниже приоритет процесса в смысле использования процессора.

Вызов *getpriority* возвращает наивысший приоритет (наименьшее числовое значение), из приоритетов всех указанных процессов.

father.c

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>
main (void)
{
int pr;
pr = getpriority(PRIO_PROCESS, 0);
printf("Prio_PROCESS = %d\n", pr);
pr = getpriority(PRIO_USER, 0);
printf("Prio_USER = %d\n", pr);
pr = getpriority(PRIO_PGRP, 0);
printf("Prio_GROUP = %d\n", pr);
}
```

Запустим процесс father.out:

```
$ ./father.out
Prio_PROCESS = 0
Prio_USER = -11
Prio_GROUP = 0
```

Видно, что приоритет процесса и группы = 0, а приоритет пользователя, запустившего процесс = -11.

Команда nohup позволяет запускать процессы, таким образом, чтобы они не зависели от терминала в котором запущены. При закрытии терминала процесс продолжает выполняться.

father.c :

```
#include <stdio.h>
main (void)
{
int i;
while (i<9999999999999)
{
i=i+1;
}
}
```

Запустим длительный процесс father по nohup:

```
$ nohup ./father.out &
[1] 2974
$ nohup: ignoring input and appending output to `nohup.out'
```

Выведем процесс:

```
$ ps -a
PID TTY TIME CMD
2974 pts/0 00:01:28 father.out
2976 pts/0 00:00:00 ps
```

Завершим сеанс. Снова войдем в систему и проверим таблицу процессов:

```
$ ps -A
PID TTY TIME CMD
...
2996 ? 00:03:38 father.out
...
```

Процесс продолжает выполняться, но уже не привязан к терминалу.

3.4. Многонитевое функционирование

Программа, формирующая несколько нитей:

```
#include <stdio.h>
#include <pthread.h>
void* thread1 (void* );
void* thread2 (void* );
void * thread_function1(void * arg) {
int time = 0;
while (1) {
printf("Thread 1: %d\n", time++);
sleep(5);
}
}
void * thread_function2(void * arg) {
int time = 0;
while (1) {
printf("Thread 2: %d\n", time++);
sleep(1);
}
}
int main() {
pthread_t t1, t2;
pthread_create(&t1, NULL, thread_function1, NULL);
pthread_create(&t2, NULL, thread_function2, NULL);
pthread_join(t1, NULL);
pthread_join(t2, NULL);
return 0;
}
```

```
$ gcc -lpthread -o thread.out thread.c
$ ./thread.out
Thread 1: 0
Thread 2: 0
Thread 2: 1
Thread 2: 2 11

Thread 2: 3
Thread 2: 4
```

```
Thread 1: 1
Thread 2: 5
Thread 2: 6
Thread 2: 7
Thread 2: 8
Thread 2: 9
Thread 1: 2
Thread 2: 10
Thread 2: 11
Thread 2: 12
Thread 2: 13
Thread 2: 14
Thread 1: 3
Thread 2: 15
```

Программа создает две параллельно выполняющиеся нити. Они не являются процессами, поэтому не имеют собственных PID и завершить отдельную нить с помощью kill не получится.

Модифицируем программу так, чтобы управление второй нитью осуществлялось посредством сигнала SIGUSR1 из первой нити. На пятой секунде работы приложения удаляем вторую нить.

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <signal.h>
pthread_t t1, t2;
void * thread_function1(void * arg) {
int time = 0;
while (1) {
printf("Thread 1: %d\n", time++);
sleep(5);
pthread_kill(t2, SIGUSR1);
}
}
void * thread_function2(void * arg) {
int time = 0;
while (1) {
printf("Thread 2: %d\n", time++);
sleep(1);
}
}
int main() {
pthread_create(&t1, NULL, thread_function1, NULL);
pthread_create(&t2, NULL, thread_function2, NULL);
pthread_join(t1, NULL);
pthread_join(t2, NULL);
return 0;
}
```

Результат запуска:

```
Thread 1: 0
Thread 2: 0
Thread 2: 1
Thread 2: 2
Thread 2: 3
Thread 2: 4
User defined signal 1
```

Первая нить по истечении 5 секунд отправляет SIGUSR1 второй нити. У нити нет идентификатора, поэтому доставка сигнала нити заключается в доставке сигнала всему процессу в контексте этой нити. Поэтому завершается весь процесс, а не только вторая нить.

Последняя модификация предполагает создание собственного обработчика сигнала, содержащего уведомление о начале его работы и возврат.

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <signal.h>
pthread_t t1, t2;
void int_handler(int signum) {
printf("Caught SIGUSR1\n");
pthread_exit(NULL);
}
void * thread_function1(void * arg) {
int time = 0;
while (1) {
printf("Thread 1: %d\n", time++);
sleep(5);
pthread_kill(t2, SIGUSR1);
}
}
void * thread_function2(void * arg) {
int time = 0;
while (1) {
printf("Thread 2: %d\n", time++);
sleep(1);
}
}
int main() {
signal(SIGUSR1, int_handler);
pthread_create(&t1, NULL, thread_function1, NULL);
pthread_create(&t2, NULL, thread_function2, NULL);
pthread_join(t1, NULL);
pthread_join(t2, NULL);
return 0;
}
```

Результат запуска:

```
Thread 1: 0
Thread 2: 0
Thread 2: 1
Thread 2: 2
Thread 2: 3
Thread 2: 4
Thread 1: 1
Caught SIGUSR1
Thread 1: 2
Thread 1: 3
```

Переопределенный обработчик SIGUSR1 выполняется в контексте второй нити, поэтому именно для нее вызывается pthread_exit(). 14

4. Выводы

От оптимальной настройки подсистемы управления процессами и числа одновременно выполняющихся процессов зависит загрузка процессора, что в свою очередь оказывает непосредственное влияние на производительность системы в целом. Ядро ОС предоставляет задачам базовый набор услуг, определяемый интерфейсом системных вызовов.

В системе UNIX одновременно могут выполняться несколько процессов. Выполнение процесса заключается в точном следовании набору инструкций, который никогда не передает управление набору инструкций другого процесса. В то же время процессы имеют возможность обмениваться друг с другом данными с помощью межпроцессного взаимодействия, предоставляемого системой. В UNIX существует набор средств взаимодействия между процессами, таких как сигналы, каналы, разделяемая память, семафоры, сообщения, но в остальном процессы изолированы друг от друга. (В данной лабораторной работе использовалось одно средство взаимодействия – сигналы.)

Процесс имеет несколько атрибутов, позволяющих ОС эффективно управлять его работой. Каждый процесс имеет уникальный идентификатор PID, позволяющий ядру системы различать процессы. Когда создается новый процесс, ядро присваивает ему следующий свободный идентификатор, а когда процесс завершает свою работу, ядро занятый им идентификатор освобождает. Идентификатор родительского процесса PPID – это идентификатор процесса, породившего данный процесс.

Сигналы являются способом передачи уведомления о возникновении определенного события. Также их можно рассматривать как программные прерывания, когда нормальное выполнение процесса может быть прервано. В системе UNIX перечень сигналов, поддерживаемых процессами, довольно большой, и у каждого есть свое назначение.

Основные отличия процесса от нити заключаются в том, что каждому процессу соответствует своя независимая от других область памяти, таблица открытых файлов, текущая директория и прочая информация уровня ядра. У всех нитей, принадлежащих данному процессу, всё вышеперечисленное общее, поскольку принадлежат они одному процессу. Кроме того, процесс всегда является понятием уровня ядра, то есть ядро знает о его существовании, в то время как нити зачастую являются понятиями уровня пользователя, и ядро может ничего не знать о них. В подобных реализациях все данные о нитях хранятся в пользовательской области памяти, и, соответственно, такие процедуры, как порождение или переключение между нитями, не требуют обращения к ядру и занимают на порядок меньше времени.