Санкт-Петербургский Государственный Политехнический Университет

Факультет Технической Кибернетики

Кафедра Компьютерных Систем и Программных Технологий

**ОТЧЁТ О ЛАБОРАТОРНОЙ РАБОТЕ №3**

Дисциплина: С**истемное программное обеспечение**

Тема: **Управление процессами и нитями**

Д.А. Киселёв Е.В. Душутина

Выполнил студент гр. 4081/10

Преподаватель

Санкт-Петербург

2012

***Цель работы:***

Изучить основные принципы управления процессами и нитями.

***Программа работы:***

1. Порождение и запуск процессов
2. Взаимодействие родственных процессов
3. Управление процессами посредством сигналов
4. Многонитевое функционирование

**1. Порождение и запуск процессов**

**1.1. Выполнение системных вызовов fork(); execl(); wait(); exit(); sleep().**

* **fork()** создаёт новый процесс (потомок), который является копией, вызывающего процесса (родителя), но имеет другие значения PID и PPID. Также не наследуются ожидающие сигналы и файловые дескрипторы. Родителю возвращается PID потомка, а потомку возвращается 0.
* **execl()** заменяет образ вызывающего процесса на образ, полученный из указанного исполняемого файла. При этом сохраняются PID, PPID, UID, GID, EUID, EGID (если для исполняемого файла на установлен флаг SUID или SGID), переменные окружения.
* **wait()** приостанавливает вызывающий процесс до тех пор, пока не завершится выполнение любого его потомка. Возвращает ID этого потомка.
* **void exit(int status)** завершает выполнение вызвавшего процесса и возвращает родительскому процессу значение аргумента status.
* **unsigned int sleep(unsigned int seconds)** приостанавливает вызывающий процесс на указанное время. Возвращает 0, если процесс проспал все указанное время, или количество оставшихся секунд сна, если процесс был разбужен сигналом.

**1.2. Программы родителя и потомка father.c и son.c**

Файл father.c:

#include <stdio.h>

#include <stddef.h>

#include <process.h>

main() {

int pid, ppid, status;

pid=getpid();

ppid=getppid();

printf("\n FATHER PARAM: pid=%i ppid=%i \n", pid, ppid);

if(fork()==0) execl("son.out","son.out", NULL);

wait(&status);

}

Файл son.c:

#include <stdio.h>

main(){

int pid, ppid;

pid=getpid();

ppid=getppid();

printf("\n SON PARAM: pid=%i ppid=%i \n\n", pid, ppid);

sleep(1);

system("ps -a -o uid,pid,ppid,pgid,time,cmd");

}

Запустим программу father.out:

# gcc -o father.out father.c

# gcc -o son.out son.c

# ./father.out

FATHER PARAM: pid=1871919 ppid=565291

SON PARAM: pid=1876016 ppid=1871919

UID PID PPID PGID TIME CMD

0 507932 376865 507932 00:00:01 pfm

0 557087 507932 507932 00:00:00 /usr/photon/bin/pterm

0 376865 1 249887 00:00:18 shelf

0 397346 1 249887 00:00:00 bkgdmgr

0 397347 1 249887 00:00:01 wmswitch

0 397348 1 249887 00:00:00 saver

0 561191 507932 507932 00:00:00 /usr/photon/bin/pterm

0 565290 376865 249887 00:00:18 pterm

0 1667116 507932 507932 00:00:00 /usr/photon/bin/pterm

0 1871919 565291 1871919 00:00:00 ./father.out

0 1876016 1871919 1871919 00:00:00 son.out

0 1880113 1876016 1871919 00:00:00 sh

0 1880114 1880113 1871919 00:00:00 ps

Последние 4 процесса принадлежат к одной группе процессов, которая имеет ID такой же, как у процесса, выполняющего программу father.out. Процесс 1880114, выполняющий программу ps, был вызван процессом 1880113, выполняющим программу sh, который был вызван процессом 1876016, выполняющим программу son.out и вызванным процессом 1871919, выполняющим father.out.

Запуск программы father.out в фоновом режиме позволяет запускать с терминала новые процессы, не дожидаясь завершения father.out.

**2. Взаимодействие родственных процессов**

Рассмотрим 3 ситуации:

1. Родительский процесс запускает дочерний, и оба выполняются.
2. Родительский процесс запускает дочерний и, не ожидая его завершения, завершает свое выполнение.
3. Родительский процесс запускает дочерний и не ожидает его завершения; дочерний процесс завершается.

Для этого напишем программу father, которая будет демонстрировать все три ситуации с выводом результатов в итоговый файл.

Файл father2.c:

#include <stdio.h>

#include <stddef.h>

#include <process.h>

main() {

int pid, ppid;

pid=getpid();

ppid=getppid();

FILE \*f;

f = fopen("output.txt", "w");

fprintf( f, "Father start\nFATHER PARAM: pid=%i ppid=%i \n", pid, ppid);

fclose(f);

if(fork()==0) execl("son2.out","son2.out", NULL);

sleep(1);

f = fopen("output.txt", "a");

fprintf( f, "\nS2: FATHER END: \n");

fclose(f);

printf("Wait a few seconds please\n");

}

Файл son2.c:

#include <stdio.h>

#include <stddef.h>

#include <process.h>

main() {

FILE \*f;

f = fopen("output.txt", "a");

fprintf( f, "Son start\nSON PARAM: pid=%i ppid=%i \nS1: FATHER AND SON ARE RUNNING: \n", getpid(), getppid());

fclose(f);

system("ps -a -o uid,pid,ppid,pgid,time,cmd >> output.txt");

sleep(2);

system("ps -a -o uid,pid,ppid,pgid,time,cmd >> output.txt");

f = fopen("output.txt", "a");

fprintf( f, "\nS3: FATHER PARAM: pid=%i ppid=%i name=son2.out\n", getpid(), getppid());

fclose(f);

if(fork()==0) execl("grandson.out", "grandson.out", NULL);

sleep(1);

system("pidin -f aehn | grep -e ppid -e Zombie -e son >> output.txt");

printf("Now you can read the output file\n");

}

В QNX утилита ps не выводит информацию о процессах-зомби, даже если они присутствуют. Для того, чтобы увидеть такой процесс, была использована утилита pidin, которая является аналогом posix-команды ps, но создана специально для QNX. Файл grandson.c:

#include <stdio.h>

main(){

int pid, ppid;

pid=getpid();

ppid=getppid();

FILE \*f;

f = fopen("output.txt", "a");

fprintf( f, "Son start\nSON PARAM: pid=%i ppid=%i name=grandson.out \nSon end: \n", pid, ppid);

fclose(f);

}

Скомпилируем тексты программ GNU-компилятором, запустим файл father2.out, дождёмся завершения всех трёх программ и прочитаем файл output.txt:

Father start

FATHER PARAM: pid=1146921 ppid=434207

Son start

SON PARAM: pid=1151018 ppid=1146921

S1: FATHER AND SON ARE RUNNING:

UID PID PPID PGID TIME CMD

0 434204 376865 249887 00:00:11 pterm

0 376865 1 249887 00:00:13 shelf

0 397346 1 249887 00:00:00 bkgdmgr

0 397347 1 249887 00:00:01 wmswitch

0 397348 1 249887 00:00:00 saver

0 434213 376865 434213 00:00:01 pfm

0 446502 434213 434213 00:00:00 /usr/photon/bin/pterm

0 1146921 434207 1146921 00:00:00 ./father2.out

0 1151018 1146921 1146921 00:00:00 son2.out

0 1155115 1151018 1146921 00:00:00 sh

0 1155116 1155115 1146921 00:00:00 ps

S2: FATHER END:

UID PID PPID PGID TIME CMD

0 434204 376865 249887 00:00:11 pterm

0 376865 1 249887 00:00:13 shelf

0 397346 1 249887 00:00:00 bkgdmgr

0 397347 1 249887 00:00:01 wmswitch

0 397348 1 249887 00:00:00 saver

0 434213 376865 434213 00:00:01 pfm

0 446502 434213 434213 00:00:00 /usr/photon/bin/pterm

0 1171497 1151018 1146921 00:00:00 sh

0 1151018 1 1146921 00:00:00 son2.out

0 1171499 1171497 1146921 00:00:00 ps

S3: FATHER PARAM: pid=1151018 ppid=1 name=son2.out

Son start

SON PARAM: pid=1183785 ppid=1151018 name=grandson.out

Son end:

pid ppid thread name name

1151018 1 1 son2.out

1183785 1151018 (Zombie)

Первая ситуация совпадает с рассмотренной выше. При завершении родительского процесса его дочерний процесс становится дочерним процессом процесса init (изменился ppid процесса 1151018). Когда процесс завершается, он становится процессом-зомби, освобождаются все его ресурсы, сохраняются только его pid и ppid в системной статистике. Процесс-зомби существует до тех пор, пока родительский процесс не считает его код возврата или пока не завершится родительский процесс.

**3. Управление процессами посредством сигналов**

**3.1.** Перечень сигналов, поддерживаемых процессами:

# kill -l

1 HUP Hangup 29 PROF Profiling timer expired

2 INT Interrupt 30 XCPU CPU time limit exceeded

3 QUIT Quit 31 XFSZ File size limit exceeded

4 ILL Illegal instruction 32 32 signal 32

5 TRAP Trace trap 33 33 signal 33

6 ABRT Abort 34 34 signal 34

7 EMT EMT trap 35 35 signal 35

8 FPE Floating point exception 36 36 signal 36

9 KILL Killed 37 37 signal 37

10 BUS Bus error 38 38 signal 38

11 SEGV Memory fault 39 39 signal 39

12 SYS Bad system call 40 40 signal 40

13 PIPE Broken pipe 41 41 rtsig 41

14 ALRM Alarm clock 42 42 rtsig 42

15 TERM Terminated 43 43 rtsig 43

16 USR1 User defined signal 1 44 44 rtsig 44

17 USR2 User defined signal 2 45 45 rtsig 45

18 CHLD Child exited 46 46 rtsig 46

19 PWR Power-fail/Restart 47 47 rtsig 47

20 WINCH Window size change 48 48 rtsig 48

21 URG Urgent I/O condition 49 49 rtsig 49

22 IO I/O possible 50 50 rtsig 50

23 STOP Stopped (signal) 51 51 rtsig 51

24 TSTP Stopped 52 52 rtsig 52

25 CONT Continued 53 53 rtsig 53

26 TTIN Stopped (tty input) 54 54 rtsig 54

27 TTOU Stopped (tty output) 55 55 rtsig 55

28 VTALRM Virtual timer expired 56 56 rtsig 56

Системный вызов kill отправляет сигнал процессу или группе процессов. Формат вызова:

#include <sys/types.h>

#include <signal.h>

int kill( pid\_t pid, int sig );

Если pid > 0, то сигнал посылается процессу с идентификатором pid. Если pid = 0, то сигнал посылается всем процессам из группы вызывающего процесса. Если pid < 0, то сигнал посылается группе процессов с идентификатором –pid. Аргумент sig задаёт номер отправляемого сигнала.

Системный вызов signal позволяет определить действия, выполняемые при получении сигнала. Формат вызова:

#include <signal.h>

void ( \* signal( int sig, void ( \* func)(int) ) )( int );

Номер сигнала определяется аргументом sig. Аргумент func может принимать следующие значения:

* SIG\_DFL – выполнять действие по умолчанию.
* SIG\_IGN – игнорировать указанный сигнал.
* Указатель на функцию. При получении сигнала будет выполнена эта функция и установлено действие по умолчанию для данного сигнала.

Сигналы SIGSTOP и SIGKILL нельзя перехватить или проигнорировать.

Напишем программы для демонстрации следующих событий: процесс father3 порождает процессы son3, son4, son5 и запускает на исполнение программные коды из соответствующих исполнительных файлов; далее родительский процесс осуществляет управление потомками, для этого он генерирует сигнал каждому пользовательскому процессу; son3 реагирует на сигнал по умолчанию, son4 игнорирует сигнал, son5 перехватывает и обрабатывает сигнал. Файл father3.c:

#include <stdio.h>

#include <stddef.h>

#include <process.h>

#include <sys/types.h>

#include <signal.h>

main() {

char \*names[3] = {"son3", "son4", "son5"};

int i, sons[3];

printf("FATHER PARAM: pid=%i ppid=%i \n", getpid(), getppid());

for(i=0; i<3; i++) {

sons[i] = fork();

if(sons[i]==0) execl(names[i], names[i], NULL);

}

sleep(1);

printf("PIDIN before sending signals:\n");

system("pidin -f aehn | grep -e ppid -e father -e son");

for(i=0; i<3; i++) kill(sons[i], SIGUSR1);

sleep(1);

printf("\nPIDIN after sending signals:\n");

system("pidin -f aehn | grep -e ppid -e father -e son -e Zombie");

for(i=1; i<3; i++) kill(sons[i], SIGKILL);

}

Файл son3.c:

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

main() {

printf("SON3 PARAM: pid=%i ppid=%i \n", getpid(), getppid());

signal( SIGUSR1, SIG\_DFL );

while(1) pause();

}

Файл son4.c:

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

main() {

printf("SON4 PARAM: pid=%i ppid=%i \n", getpid(), getppid());

signal( SIGUSR1, SIG\_IGN );

while(1) pause();

}

Файл son5.c:

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

void mySigHandler(int sigNo) {

printf("\nson5 received the SIGUSR1\n");

}

main() {

printf("SON5 PARAM: pid=%i ppid=%i \n", getpid(), getppid());

signal( SIGUSR1, mySigHandler );

while(1) pause();

}

Функция pause() приостанавливает поток до получения сигнала. Запустим father3:

# gcc -o father3 father3.c

# gcc -o son3 son3.c

# gcc -o son4 son4.c

# gcc -o son5 son5.c

# ./father3

FATHER PARAM: pid=4304949 ppid=434207

SON3 PARAM: pid=4309046 ppid=4304949

SON4 PARAM: pid=4309047 ppid=4304949

SON5 PARAM: pid=4309049 ppid=4304949

PIDIN before sending signals:

pid ppid thread name name

4304949 434207 1 ./father3

4309046 4304949 1 son3

4309047 4304949 1 son4

4309049 4304949 1 son5

son5 received the SIGUSR1

PIDIN after sending signals:

pid ppid thread name name

4304949 434207 1 ./father3

4309046 4304949 (Zombie)

4309047 4304949 1 son4

4309049 4304949 1 son5

В результате получения сигнала SIGUSR1 son3 выполнил действие по умолчанию (завершил выполнение), son4 проигнорировал сигнал, son5 перехватил сигнал, выполнил функцию по обработке сигнала, которая выводит на стандартный вывод сообщение о том, что получен сигнал, и продолжил выполнение функции main().

**3.2.** Отправим сигналы двум процессам, находящимся в разных состояниях: активном и пассивном. Файл father4.c:

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <stddef.h>

#include <process.h>

#include <sys/types.h>

#include <signal.h>

main() {

char \*names[2] = {"active", "passive"};

int i, sons[2];

time\_t curTime;

for(i=0; i<2; i++) {

sons[i] = fork();

if(sons[i]==0) execl(names[i], names[i], NULL);

}

sleep(1);

for(i=0; i<2; i++) {

kill(sons[i], SIGUSR1);

curTime = time(NULL);

printf("SIGUSR1 has been sent to %s process at %s", names[i], ctime(&curTime));

}

}

Файл active.c:

#include <stdio.h>

#include <signal.h>

#include <stdlib.h>

#include <time.h>

void mySigHandler(int sigNo) {

time\_t curTime = time(NULL);

printf("SIGUSR1 has been delivered to active process at %s", ctime(&curTime));

exit(EXIT\_SUCCESS);

}

main() {

signal(SIGUSR1, mySigHandler);

int c=0;

while(1) c++;

}

Файл passive.c:

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#include <stdlib.h>

#include <time.h>

void mySigHandler(int sigNo) {

time\_t curTime = time(NULL);

printf("SIGUSR1 has been delivered to passive process at %s", ctime(&curTime));

}

main() {

signal(SIGUSR1, mySigHandler);

pause();

}

Запустим father4:

# gcc -o father4 father4.c

# gcc -o active active.c

# gcc -o passive passive.c

# ./father4

SIGUSR1 has been sent to active process at Sun Jan 6 17:54:57 2013

SIGUSR1 has been sent to passive process at Sun Jan 6 17:54:57 2013

SIGUSR1 has been delivered to active process at Sun Jan 6 17:54:57 2013

SIGUSR1 has been delivered to passive process at Sun Jan 6 17:54:57 2013

**3.3.** Если в фоновом режиме запущены одновременно несколько утилит, то можно просмотреть их список и очередность выполнения командой jobs:

# ./son4 & ./son5 & ./passive &

[1] 5439538

[2] 5439539

[3] 5439541

# SON4 PARAM: pid=5439538 ppid=434207

SON5 PARAM: pid=5439539 ppid=434207

# jobs

[3] + Running ./passive

[2] - Running ./son5

[1] Running ./son4

С помощью команды fg можно перевести задание из фонового режима в приоритетный. При этом терминал становится неактивным. Чтобы продолжить работу можно остановить процесс son4 нажатием клавиш CTRL+Z. После этого можно снова запустить этот процесс в фоновом режиме командой bg:

# fg %1

./son4

[1] + Stopped ./son4

# bg

[1] ./son4

# jobs

[1] + Running ./son4

[3] - Running ./passive

[2] Running ./son5

Также приоритетный процесс можно не останавливать, а убить нажатием CTRL+C. Процессы, запущенные в фоновом режиме, можно убить командой kill по их номеру в списке заданий или командой slay по их имени (аналог posix-команды kill, созданный специально для QNX):

# slay son5

[2] Terminated ./son5

# jobs

[1] + Running ./son4

[3] - Running ./passive

# kill %3

# jobs

[1] + Running ./son4

[3] - Terminated ./passive

# jobs

[1] + Running ./son4

**3.4.** Команда nice позволяет запустить программу с измененным приоритетом. Синтаксис:

nice [-nprioritylevels] command [arguments]...

prioritylevels – это число, на которое будет уменьшен приоритет программы по сравнению с текущим приоритетом. Оно может быть как положительным, так и отрицательным. Если оно не указано, то приоритет будет понижен на единицу. Обычный пользователь может установить приоритет от 1 до 63, суперпользователь – от 1 до 255. Если получается приоритет, выходящий за эти границы, то команда не выполняется.

Системный вызов nice() позволяет изменить приоритет процесса. Формат вызова:

#include <unistd.h>

int nice( int incr );

Приоритет вызывающего процесса увеличивается на incr. Приоритет может быть задан от 0 до (2\*NZERO-1). Значение NZERO определено в файле <limits.h>, по умолчанию оно равно 20. При задании приоритета, выходящего за установленные границы, он принимает граничное значение. Только процессы, выполняющиеся от имени суперпользователя, могут понижать свой приоритет. Но эта функция не работает, если в QNX Neutrino присутствуют какие-либо средства планирования. В этом случае можно использовать аналогичную функцию sched\_setparam(). Её формат вызова:

#include <sched.h>

int sched\_setparam( pid\_t pid, const struct sched\_param \*param );

Здесь можно указать, приоритет какого процесса должен быть изменён (если pid=0, то будет изменен приоритет вызывающего процесса). param – это указатель на структуру sched\_param, в которой содержится поле sched\_priority, в котором содержится приоритет, который будет назначен процессу. Кроме того, можно изменить приоритет потока с помощью функции pthread\_setschedprio(), формат вызова:

#include <pthread.h>

int pthread\_setschedprio( pthread\_t thread, int prio );

Аргументами являются идентификатор потока и новое значение приоритета. Идентификатор вызывающего потока можно получить с помощью функции pthread\_self(). Для каждого уровня приоритета есть своя очередь потоков с данным приоритетом готовых к исполнению. Если повысить приоритет выполняющегося или готового к выполнению потока, то он окажется в конце очереди потоков, у которых приоритет совпадает с новым приоритетом этого потока. Если же понизить приоритет, то поток окажется в начале соответствующей очереди.

Системный вызов getprio() возвращает приоритет указанного процесса. Формат вызова:

#include <sched.h>

int getprio( pid\_t pid );

Возвращает приоритет потока 1 в указанном процессе. Если pid=0, то возвращает приоритет вызвавшего потока.

**3.5. Утилита nohup**

Синтаксис:

nohup command [argument...]

Запускает указанную команду в режиме игнорирования сигнала SIGHUP. Таким образом, запущенный процесс останется в списке процессов даже после завершения сеанса и повторного входа в систему. Если стандартным выводом является терминал, то все данные, выводимые запущенным процессом на стандартный вывод, будут добавлены в файл nohup.out в текущем каталоге (или, если это невозможно, в домашнем каталоге, а если и там не удастся это сделать, то указанная команда не будет запущена). Если запустить nohup в фоновом режиме, то SIGQUIT тоже будет игнорироваться.

**4. Многонитевое функционирование**

Программа, формирующая несколько нитей (потоков):

#include <stdio.h>

#include <pthread.h>

void \*thread1(void \*t) {

int cnt = 0;

while(1) {

printf("thread1: %i\n", cnt++);

sleep(5);

}

}

void \*thread2(void \*t) {

int cnt = 0;

while(1) {

printf("thread2: %i\n", cnt++);

sleep(1);

}

}

main() {

pthread\_t t1, t2;

pthread\_create(&t1, NULL, thread1, NULL);

pthread\_create(&t2, NULL, thread2, NULL);

pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

}

Первая нить в цикле выводит на экран своё имя и количество пятисекундных интервалов своего функционирования; вторая нить – количество секундных интервалов. Запустим программу и проанализируем таблицу процессов (с другого терминала):

# gcc -o threads threads.c

# ./threads

thread1: 0

thread2: 0

thread2: 1

thread2: 2

thread2: 3

thread2: 4

thread1: 1

thread2: 5

thread2: 6

thread2: 7

thread2: 8

thread2: 9

thread1: 2

thread2: 10

thread2: 11

thread2: 12

thread2: 13

thread2: 14

thread1: 3

thread2: 15

thread2: 16

thread2: 17

# pidin | grep -e tid -e thread

pid tid name prio STATE Blocked

5808172 1 ./threads 10r JOIN 2

5808172 2 ./threads 10r NANOSLEEP

5808172 3 ./threads 10r NANOSLEEP

Если попытаться удалить нить по её идентификатору командой kill, то будет удалён весь процесс полностью. Модифицируем программу так, чтобы управление второй нитью осуществлялось посредством сигнала SIGUSR1 из первой нити:

#include <stdio.h>

#include <pthread.h>

#include <signal.h>

pthread\_t t1, t2;

void \*thread1(void \*t) {

int cnt = 0;

while(1) {

printf("thread1: %i\n", cnt++);

sleep(5);

if(cnt==1) pthread\_kill(t2, SIGUSR1);

}

}

void \*thread2(void \*t) {

int cnt = 0;

while(1) {

printf("thread2: %i\n", cnt++);

sleep(1);

}

}

main() {

pthread\_create(&t1, NULL, thread1, NULL);

pthread\_create(&t2, NULL, thread2, NULL);

pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

}

Запустим:

# gcc -o threads2 threads2.c

# ./threads2

thread1: 0

thread2: 0

thread2: 1

thread2: 2

thread2: 3

thread2: 4

thread1: 1

User defined signal 1

Несмотря на то, что сигнал был адресован конкретной нити, процесс был удалён полностью. Одну нить в отдельности можно убить функцией pthread\_exit(), если она будет вызвана из этой нити. Перепишем программу, чтобы продемонстрировать это:

#include <stdio.h>

#include <pthread.h>

#include <signal.h>

pthread\_t t1, t2;

void sigHndlr(int sigNo) {

printf("SIGUSR1 was here\n");

pthread\_exit(NULL);

}

void \*thread1(void \*t) {

int cnt = 0;

while(1) {

printf("thread1: %i\n", cnt++);

sleep(5);

if(cnt==1) pthread\_kill(t2, SIGUSR1);

}

}

void \*thread2(void \*t) {

signal(SIGUSR1, sigHndlr);

int cnt = 0;

while(1) {

printf("thread2: %i\n", cnt++);

sleep(1);

}

}

main() {

pthread\_create(&t1, NULL, thread1, NULL);

pthread\_create(&t2, NULL, thread2, NULL);

pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

}

И запустим:

# gcc -o threads3 threads3.c

# ./threads3

thread1: 0

thread2: 0

thread2: 1

thread2: 2

thread2: 3

thread2: 4

thread1: 1

SIGUSR1 was here

thread1: 2

thread1: 3

thread1: 4

thread1: 5

thread1: 6

После получения сигнала вторая нить больше не выводит сообщения на терминал, но первая всё ещё работает. Посмотрим таблицу процессов:

# pidin | grep -e tid -e thread

pid tid name prio STATE Blocked

6168620 1 ./threads3 10r JOIN 2

6168620 2 ./threads3 10r NANOSLEEP

6168620 3 ./threads3 10r DEAD

**5. Выводы**

Были изучены основные принципы управления процессами и нитями в QNX. Новый процесс может быть создан копированием существующего и заменой его образа на образ из исполняемого файла. Процессы могут взаимодействовать друг с другом путём отправки сигналов, причём стандартная реакция на сигнал может быть изменена. У каждого процесса есть определенный уровень приоритета, их также можно изменять. Каждый процесс работает в своём собственном виртуальном адресном пространстве. Процесс может состоять из нескольких потоков или нитей. Можно послать сигнал отдельной нити.