**«Санкт-Петербургский Государственный Политехнический Университет»**

Кафедра компьютерных систем и программных технологий

**Отчёт по лабораторной работе №3**

Тема: «Процессы Unix»

Дисциплина: «Операционные системы»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Выполнил студент гр.3530901/70203 | \_\_\_\_\_\_\_\_\_\_\_ | | | Иванов И.Д. |
| Преподав­атель, к.т.н., доц. | \_\_\_\_\_\_\_\_\_\_\_ | | | Душутина Е. В. |
|  |  | “\_\_\_”\_\_\_\_\_\_\_\_\_\_\_\_ 2019 г. | | |
|  |  | |  | |

**1. Цели работы**

Изучение принципов управления процессами в среде ОС UNIX на примере Linux.

**2. Программа работы**

1. Создайте программу на основе одного исходного (а затем исполняемого) файла с псевдораспараллеливанием вычислений посредством порождения процесса-потомка.

2. Выполните сначала однократные вычисления в каждом процессе, обратите внимание, какой процесс на каком этапе владеет процессорным ресурсом. Каждый процесс должен иметь вывод на терминал, идентифицирующий текущий процесс. Последняя исполняемая команда функции main должна вывести на терминал сообщение о завершении программы. Объясните результаты. Сделайте выводы об использовании адресного пространства.

3. Затем однократные вычисления замените на циклы, длительность исполнения которых достаточна для наблюдения конкуренции процессов за процессорный ресурс.

4. Измените процедуру планирования и повторите эксперимент.

5. Разработайте программы родителя и потомка с размещением в файлах father.c и son.c Для фиксации состояния таблицы процессов в файле целесообразно использовать системный вызов system("ps -abcde > file").

6. Запустите на выполнение программу father.out, получите информацию о процессах, запущенных с вашего терминала;

7. Выполните программу father.out в фоновом режиме father & Получите таблицу процессов, запущенных с вашего терминала (включая отцовский и сыновний процессы).

8. Выполните создание процессов с использованием различных функций семейства exec() с разными параметрами функций семейства, приведите результаты эксперимента.

9. Проанализируйте значение, возвращаемое функцией wait(&status). Предложите эксперимент, позволяющий родителю отслеживать подмножество порожденных потомков, используя различные функции семейства wait().

10. Проанализируйте очередность исполнения процессов.

10.1. очередность исполнения процессов, порожденных вложенными вызовами fork(). 10.2. Измените процедуру планирования с помощью функции с шаблоном scheduler в ее названии и повторите эксперимент.

10.3. Поменяйте порядок очереди в RR-процедуре.

10.4. Можно ли задать разные процедуры планирования разным процессам с одинаковыми приоритетами. Как они будут конкурировать, подтвердите экспериментально.

11. Определите величину кванта. Можно ли ее поменять? – для обоснования проведите эксперимент.

12. Проанализируйте наследование на этапах fork() и exec(). Проведите эксперимент с родителем и потомками по доступу к одним и тем же файлам, открытым родителем. Аналогичные эксперименты проведите по отношению к другим параметрам.

13.1. Изменяя длительности выполнения процессов и параметры системных вызовов, рассмотрите 3 ситуации и получите соответствующие таблицы процессов:

а) процесс-отец запускает процесс-сын и ожидает его завершения;

б) процесс-отец запускает процесс-сын и, не ожидая его завершения, завершает свое выполнение. Зафиксируйте изменение родительского идентификатора процесса-сына;

в) процесс-отец запускает процесс-сын и не ожидает его завершения; процесс-сын завершает свое выполнение. Зафиксируйте появление процесса-зомби, для этого включите команду ps в программу father.c

13.2. Перенаправьте вывод не только на терминал, но и в файл. Организуйте программу многопроцессного функционирования так, чтобы результатом ее работы была демонстрация всех трех ситуаций с отображением в итоговом файле

Управление процессами посредством сигналов

13.1. С помощью команды kill -l ознакомьтесь с перечнем сигналов, поддерживаемых процессами. Ознакомьтесь с системными вызовами kill(2), signal(2).

Подготовьте программы следующего содержания:

а.) процесс father порождает процессы son1, son2, son3 и запускает на исполнение программные коды из соответствующих исполнительных файлов;

б.) далее родительский процесс осуществляет управление потомками, для этого он генерирует сигнал каждому пользовательскому процессу;

в.) в пользовательских процессах-потомках необходимо обеспечить:

для son1 - реакцию на сигнал по умолчанию;

для son2 - реакцию игнорирования;

для son3 - перехватывание и обработку сигнала.

Сформируйте файл-проект из четырех файлов, откомпилируйте, запустите программу. Проанализируйте таблицу процессов до и после посылки сигналов с помощью системного вызова system("ps -s >> file"). Обратите внимание на реакцию, устанавливаемую для последнего потомка.

13.2. Организуйте посылку сигналов любым двум процессам, находящимся в разных состояниях: активном и пассивном, фиксируя моменты посылки и приема каждого сигнала с точностью до секунды. Приведите результаты в файле результатов.

14. Запустите в фоновом режиме несколько утилит, например: cat \*.c > myprog & lpr myprog & lpr intro& Воспользуйтесь командой jobs для анализа списка заданий и очередности их выполнения. Позаботьтесь об уведомлении о завершении одного из заданий с помощью команды notify. Аргументом команды является номер задания. Верните невыполненные задания в приоритетный режим командой fg. Например: fg %3 Отмените одно из невыполненных заданий.

15. Ознакомьтесь с выполнением команды и системного вызова nice(1) и getpriority(2). Приведите примеры их использования в приложении. Определите границы приоритетов (создайте для этого программу). Есть ли разница в приоритетах для системных и пользовательских процессов, используются ли приоритеты реального времени? Каков пользовательский приоритет для запуска приложений из shell? Все ответы подкрепляйте экспериментально.

16. Ознакомьтесь с командой nohup(1). Запустите длительный процесс по nohup(1). Завершите сеанс работы. Снова войдите в систему и проверьте таблицу процессов. Поясните результат.

17. Определите uid процесса, каково минимальное значение и кому оно принадлежит. Каково минимальное и максимальное значение pid, каким процессам принадлежат? Проанализируйте множество системных процессов, как их отличить от прочих, перечислите назначение самых важных из них.

*Многонитевое функционирование*

18. Подготовьте программу, формирующую несколько нитей. Нити для эксперимента могут быть практически идентичны. Например, каждая нить в цикле: выводит на печать собственное имя и инкрементирует переменную времени, после чего "засыпает" (sleep(5); sleep(1); - для первой и второй нитей соответственно), на экран (в файл) должно

выводиться имя нити и количество пятисекундных (для первой) и секундных (для второй) интервалов функционирования каждой нити.

19. После запуска программы проанализируйте выполнение нитей, распределение во времени. Используйте для этого вывод таблицы процессов командой ps -axhf Попробуйте удалить нить, зная ее идентификатор, командой kill. Приведите и объясните результат.

20. Модифицируйте программу так, чтобы управление второй нитью осуществлялось посредством сигнала SIGUSR1 из первой нити. На пятой секунде работы приложения удалите вторую нить. Для этого воспользуйтесь функцией pthread\_kill(t2, SIGUSR); (t2 - дескриптор второй нити). В остальном программу можно не изменять. Проанализируйте полученные результаты.

21. Последняя модификация предполагает создание собственного обработчика сигнала, содержащего уведомление о начале его работы и возврат посредством функции pthread\_exit(NULL); Сравните результаты, полученные после запуска этой модификации программы с результатами предыдущей.

22. Перехватите сигнал «CTRL C» для процесса и потока однократно, а также многократно с восстановлением исходного обработчика после нескольких раз срабатывания. Проделайте аналогичную работу для переназначения другой комбинации клавиш.

23. С помощью утилиты kill выведите список всех сигналов и дайте их краткую характеристику на основе документации ОС. Для чего предназначены сигналы с 32 по 64-й. Приведите пример их применения.

24. Проанализируйте процедуру планирования для процессов и потоков одного процесса. 24.1. Обоснуйте результат экспериментально. 24.2. Попробуйте процедуру планирования изменить. Подтвердите экспериментально, если изменение возможно. 24.3. Задайте нитям разные приоритеты программно и извне (объясните результат).

25. Создайте командный файл (скрипт), выполняющий вашу лабораторную работу автоматически при наличии необходимых С-файлов.

## **Система, на которой производится работа:**

asdf@BigAwesomeTurtle:/home/asdf/lab\_2$uname -a

Linux BigAwesomeTurtle 5.2.0-kali2-686-pae #1 SMP Debian 5.2.9-2kali1 (2019-08-22) i686 GNU/Linux

Дистрибутив – Kali Linux

**3. Выполнение работы**

**3.1. Создайте программу на основе одного исходного (а затем исполняемого) файла с псевдораспараллеливанием вычислений посредством порождения процесса-потомка.**

При вызове fork() порождается новый процесс (процесс-потомок), который почти идентичен порождающему процессу-родителю. Весь код после fork() выполняется дважды, как в процессе-потомке, так и в процессе-родителе.

У процесса есть две характеристики PID и PPID

PID – это номер процесса

PPID – это номер процесса-родителя для для данного процесса

**Текст программы:**

#include <stdio.h>

#include <math.h>

#include <sys/resource.h>

void main(int arc, char\* argv[]){

if(fork() == 0){

printf("Son done: PID %d, PPID %d\n",getpid(),getppid());

}

else{

printf("Father done: PID %d, PPID %d\n",getpid(),getppid());

}

printf("Programm done\n");

exit(0);

}

**Результат выполнения программы:**

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/1$./a.out  
Father done: PID 13307, PPID 1109  
Programm done  
Son done: PID 13309, PPID 13307  
Programm done

**3.2 Выполните сначала однократные вычисления в каждом процессе,**

**обратите внимание, какой процесс на каком этапе владеет процессорным**

**ресурсом. Каждый процесс должен иметь вывод на терминал,**

**идентифицирующий текущий процесс. Последняя исполняемая команда**

**функции main должна вывести на терминал сообщение о завершении**

**программы. Объясните результаты. Сделайте выводы об использовании**

**адресного пространства.**

**Текст программы:**

#include <stdio.h>

#include <math.h>

#include <sys/resource.h>

#include <stdlib.h>

void main(int argc, char\* argv[]){

int m, n, pid;

m=5000;

n=1;

pid = fork();

if(pid == -1){

perror("fork error");

exit(1);

}

printf("pid=%i\n", pid);

if(pid !=0){

int j;

for(j = 1; j <= 5000; j++){

m-=1;

}

printf("father: %i\n\n", m);

}

else{

int i;

for(i = 1; i <= 5000; i++){

n+=1;

}

printf("son: %d\n\n", n);

}

printf("Program finished\n");

exit(1);

}

**Результат выполнения программы:**

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/2$./a.out  
pid=13564  
father: 0  
  
Program finished  
pid=0  
son: 5001  
  
Program finished

Так как вывод информации о процессе потомке происходит уже после выполнения вычислений в процессе родителе, можно заключить, что на инициализацию нового процесса затрачивается значительное время.

**3.3. Затем однократные вычисления замените на циклы, длительность**

**исполнения которых достаточна для наблюдения конкуренции процессов**

**за процессорный ресурс.**

**Текст измененной программы:**

#include <stdio.h>

#include <stdlib.h>

int main(){

int pid, n, m;

pid = fork();

n = 500;

m = 0;

if(pid == -1){

perror("fork error");

exit(1);

}

while(1){

if(pid == 0){

printf("new pid = %d, ppid =%d \n", getpid(),getppid() );

n--;

if(n == 0){

break;

}

}

else {

printf("parent pid = %d, ppid =%d \n", getpid(),getppid() );

m++;

if(m == 500){

break;

}

}

}

printf("Process finished");

printf("n=%i, m=%i\n", n, m);

exit(1);

}

**Результат выполнения программы:**

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/3$./a.out

parent pid = 13619, ppid =1109

parent pid = 13619, ppid =1109

parent pid = 13619, ppid =1109

parent pid = 13619, ppid =1109

parent pid = 13619, ppid =1109

parent pid = 13619, ppid =1109

…

new pid = 13621, ppid =13619

new pid = 13621, ppid =13619

new pid = 13621, ppid =13619

new pid = 13621, ppid =13619

new pid = 13621, ppid =13619

new pid = 13621, ppid =13619

…

parent pid = 13619, ppid =1109

parent pid = 13619, ppid =1109

parent pid = 13619, ppid =1109

parent pid = 13619, ppid =1109

parent pid = 13619, ppid =1109

parent pid = 13619, ppid =1109

parent pid = 13619, ppid =1109

…

Process finishedn=500, m=500

new pid = 13621, ppid =13619

new pid = 13621, ppid =13619

new pid = 13621, ppid =13619

new pid = 13621, ppid =13619

new pid = 13621, ppid =13619

new pid = 13621, ppid =13619

…

Process finished n=0, m=0

В выводе программы видно, как два процесса борются за ресурсы, по очереди получая к ним доступ. Родительский процесс завершился раньше, борьба за ресурсы между этими двумя процессами прекратилась и, спустя некоторое время, работу завершил и процесс-потомок.

**3.4 Измените процедуру планирования и повторите эксперимент.**

По-умолчанию процессам присваивается процедура планирования SCHED\_OTHER, однако в данных примерах она изменена на SCHED\_FIFO и SCHED\_RR.

**Текст программы FIFO:**

#include <stdio.h>

#include <stdlib.h>

#include <sched.h>

int main()

{

int pid, n, m;

struct sched\_param shdprm;

pid = fork();

n = 500;

m = 0;

if(pid == -1)

{

perror("fork error");

exit(1);

}

shdprm.sched\_priority = 1;

if(sched\_setscheduler (getpid(), SCHED\_FIFO, &shdprm) == -1) {

perror ("ERRROR”)

}

while(1)

{

if(pid == 0)

{

printf("new pid = %d, ppid =%d \n", getpid(),getppid() );

n--;

if(n == 0)

{

break;

}

}

else {

printf("parent pid = %d, ppid =%d \n", getpid(),getppid() );

m++;

if(m == 500)

{

break;

}

}

}

printf("Process finished\n");

printf("n=%i, m=%i\n", n, m);

exit(1);

}

**Текст программы RR:**

#include <stdio.h>

#include <stdlib.h>

#include <sched.h>

int main()

{

int pid, n, m;

struct sched\_param shdprm;

pid = fork();

n = 500;

m = 0;

if(pid == -1)

{

perror("fork error");

exit(1);

}

shdprm.sched\_priority = 1;

if(sched\_setscheduler (getpid(), SCHED\_RR, &shdprm) == -1) {

perror ("ERROR");

}

while(1)

{

if(pid == 0)

{

printf("new pid = %d, ppid =%d \n", getpid(),getppid() );

n--;

if(n == 0)

{

break;

}

}

else {

printf("parent pid = %d, ppid =%d \n", getpid(),getppid() );

m++;

if(m == 500)

{

break;

}

}

}

printf("Process finished\n");

printf("n=%i, m=%i\n", n, m);

exit(1);

}

Так как изменение процедуры планирования позволено только суперпользователю, запуская программу не от его имени, мы получим сообщение об ошибке.

**Результат работы FIFO:**

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/4$sudo ./fifo

parent pid = 13947, ppid =13945

parent pid = 13947, ppid =13945

parent pid = 13947, ppid =13945

parent pid = 13947, ppid =13945

parent pid = 13947, ppid =13945

parent pid = 13947, ppid =13945

Process finished

n=500, m=500

new pid = 13948, ppid =13947

new pid = 13948, ppid =13947

new pid = 13948, ppid =13947

new pid = 13948, ppid =13947

new pid = 13948, ppid =13947

new pid = 13948, ppid =13947

…

Process finished

n=0, m=0

Так как процедура планирования теперь SCHED\_FIFO, процессы не борются за ресурсы, а выполняются один за другим.

**Результат работы RR:**

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/4$sudo ./rr

parent pid = 13947, ppid =13945

parent pid = 13947, ppid =13945

parent pid = 13947, ppid =13945

parent pid = 13947, ppid =13945

parent pid = 13947, ppid =13945

parent pid = 13947, ppid =13945

Process finished

n=500, m=500

new pid = 13948, ppid =13947

new pid = 13948, ppid =13947

new pid = 13948, ppid =13947

new pid = 13948, ppid =13947

new pid = 13948, ppid =13947

new pid = 13948, ppid =13947

…

Process finished

n=0, m=0

При изменении процедуры планирования на SCHED\_RR процессы также выполняются один за другим. Это означает, что выделенный квант времени больше, чем требуется процессу для завершения.

**3.5. Разработайте программы родителя и потомка с размещением в файлах father.c и son.c Для фиксации состояния таблицы процессов в файле целесообразно использовать системный вызов system("ps -abcde > file").**

**Содержимое father.c:**

#include <stdio.h>

int main()

{

int pid, ppid, status;

pid=getpid();

ppid=getppid();

printf("\n\nfather param: pid=%i ppid=%i\n", pid,ppid);

if(fork()==0)

{

execl("son.out","son.out", NULL);

}

system("ps xf > file.txt");

wait(&status);

printf("\n\nChild proccess is finished with status %d\n\n", status);

return 0;

}

**Содержимое son.c:**

#include <stdio.h>

int main()

{

int pid,ppid;

pid=getpid();

ppid=getppid();

printf("\n\nson param: pid=%i ppid=%i\n\n",pid,ppid);

sleep(15);

return 0;

}

Родителем процесса son действительно является процесс father:

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/5$./father\_out  
son param: pid=14139 **ppid=14137**  
father param: **pid=14137** ppid=13637  
Child proccess is finished with status 0

**3.6. Запустите на выполнение программу father.out, получите информацию о процессах, запущенных с вашего терминала;**

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/5$./father\_out  
son param: pid=14139 ppid=14137  
father param: pid=14137 ppid=13637  
Child proccess is finished with status 0

**Информация о процессах, запущенных с терминала:**

PID TTY STAT TIME COMMAND

647 ? Ss 0:00 /bin/sh /etc/xdg/xfce4/xinitrc -- /etc/X11/xinit/xserverrc

676 ? Ss 0:00 \\_ /usr/bin/ssh-agent x-session-manager

686 ? Sl 0:05 \\_ xfce4-session

701 ? Sl 1:23 \\_ xfwm4 --display :0.0 --sm-client-id 23d0ca590-df7a-4ac2-873c-6ead021fd190

703 ? Sl 0:48 \\_ xfce4-panel --display :0.0 --sm-client-id 2a3345f91-4103-4cfc-a333-69b46082fbfc

850 ? S 0:00 | \\_ /usr/lib/i386-linux-gnu/xfce4/panel/wrapper-1.0 /usr/lib/i386-linux-gnu/xfce4/panel/plugins/libsystray.so 19 10485809 systray Notification Area Area where notification icons appear

851 ? Sl 0:14 | \\_ /usr/lib/i386-linux-gnu/xfce4/panel/wrapper-2.0 /usr/lib/i386-linux-gnu/xfce4/panel/plugins/libxfce4powermanager.so 24 10485810 power-manager-plugin Power Manager Plugin Display the battery levels of your devices and control the brightness of your display

852 ? S 0:00 | \\_ /usr/lib/i386-linux-gnu/xfce4/panel/wrapper-1.0 /usr/lib/i386-linux-gnu/xfce4/panel/plugins/libactions.so 20 10485811 actions Action Buttons Log out, lock or other system actions

853 ? Sl 1:02 | \\_ /usr/lib/i386-linux-gnu/xfce4/panel/wrapper-2.0 /usr/lib/i386-linux-gnu/xfce4/panel/plugins/libpulseaudio-plugin.so 2 10485812 pulseaudio PulseAudio Plugin Adjust the audio volume of the PulseAudio sound system

706 ? Sl 0:30 \\_ Thunar --sm-client-id 2750c700d-7799-4951-8246-d36238b248c6 --daemon

739 ? Sl 0:16 \\_ xfdesktop --display :0.0 --sm-client-id 2db8e108c-bca8-4213-a4c7-857579757180

750 ? Sl 14:24 \\_ /usr/lib/firefox-esr/firefox-esr --sm-client-id 2d42bd800-2405-4884-9036-8497f70fef8c

909 ? Sl 1:05 | \\_ /usr/lib/firefox-esr/firefox-esr -contentproc -childID 1 -isForBrowser -prefsLen 1 -prefMapSize 185099 -parentBuildID 20191022215001 -greomni /usr/lib/firefox-esr/omni.ja -appomni /usr/lib/firefox-esr/browser/omni.ja -appdir /usr/lib/firefox-esr/browser 750 true tab

944 ? Sl 12:55 | \\_ /usr/lib/firefox-esr/firefox-esr -contentproc -childID 2 -isForBrowser -prefsLen 241 -prefMapSize 185099 -parentBuildID 20191022215001 -greomni /usr/lib/firefox-esr/omni.ja -appomni /usr/lib/firefox-esr/browser/omni.ja -appdir /usr/lib/firefox-esr/browser 750 true tab

969 ? Sl 0:21 | \\_ /usr/lib/firefox-esr/firefox-esr -contentproc -childID 3 -isForBrowser -prefsLen 6090 -prefMapSize 185099 -parentBuildID 20191022215001 -greomni /usr/lib/firefox-esr/omni.ja -appomni /usr/lib/firefox-esr/browser/omni.ja -appdir /usr/lib/firefox-esr/browser 750 true tab

1410 ? Sl 0:02 | \\_ /usr/lib/firefox-esr/firefox-esr -contentproc -childID 4 -isForBrowser -prefsLen 7435 -prefMapSize 185099 -parentBuildID 20191022215001 -greomni /usr/lib/firefox-esr/omni.ja -appomni /usr/lib/firefox-esr/browser/omni.ja -appdir /usr/lib/firefox-esr/browser 750 true tab

802 ? Sl 0:00 \\_ /usr/lib/policykit-1-gnome/polkit-gnome-authentication-agent-1

803 ? Sl 0:01 \\_ light-locker

811 ? Sl 0:14 \\_ nm-applet

13316 ? Sl 7:16 /opt/sublime\_text/sublime\_text /home/asdf/logfile\_3.txt

13328 ? Sl 0:10 \\_ /opt/sublime\_text/plugin\_host 13316 --auto-shell-env

1101 ? Sl 0:39 xfce4-terminal

1105 pts/0 Ss 0:00 \\_ bash

13637 pts/0 S+ 0:00 | \\_ bash

**14137 pts/0 S+ 0:00 | \\_ ./father\_out**

**14139 pts/0 S+ 0:00 | | \\_ son\_out**

14140 pts/0 S+ 0:00 | | \\_ sh -c ps xf > file.txt

14141 pts/0 R+ 0:00 | | \\_ ps xf

14138 pts/0 S+ 0:00 | \\_ tee -a /home/asdf/logfile\_3.txt

1450 ? Zs 0:00 \\_ [bash] <defunct>

14062 ? Zs 0:00 \\_ [bash] <defunct>

757 ? Ssl 0:02 xfce4-power-manager --restart --sm-client-id 22b7a270b-28fc-4adf-9b52-60c374d26f14

704 ? Ssl 0:01 xfsettingsd --display :0.0 --sm-client-id 23a012914-4e0f-4a17-9d4b-bd66e9755221

630 ? Ss 0:00 /lib/systemd/systemd --user

631 ? S 0:00 \\_ (sd-pam)

641 ? S<sl 0:27 \\_ /usr/bin/pulseaudio --daemonize=no

650 ? Ss 0:18 \\_ /usr/bin/dbus-daemon --session --address=systemd: --nofork --nopidfile --systemd-activation --syslog-only

692 ? Sl 0:01 \\_ /usr/lib/i386-linux-gnu/xfce4/xfconf/xfconfd

699 ? SLs 0:00 \\_ /usr/bin/gpg-agent --supervised

714 ? Ssl 0:00 \\_ /usr/lib/gvfs/gvfsd

823 ? Sl 0:10 | \\_ /usr/lib/gvfs/gvfsd-trash --spawner :1.12 /org/gtk/gvfs/exec\_spaw/0

722 ? Ssl 0:00 \\_ /usr/lib/at-spi2-core/at-spi-bus-launcher

727 ? S 0:01 | \\_ /usr/bin/dbus-daemon --config-file=/usr/share/defaults/at-spi2/accessibility.conf --nofork --print-address 3

751 ? Ssl 0:00 \\_ /usr/lib/gvfs/gvfs-udisks2-volume-monitor

764 ? Ssl 0:01 \\_ /usr/lib/i386-linux-gnu/xfce4/notifyd/xfce4-notifyd

769 ? Sl 0:02 \\_ /usr/lib/at-spi2-core/at-spi2-registryd --use-gnome-session

813 ? Sl 0:00 \\_ /usr/lib/dconf/dconf-service

846 ? Ssl 0:00 \\_ /usr/lib/gvfs/gvfsd-metadata

**3.7 Выполните программу father.out в фоновом режиме father & Получите таблицу процессов, запущенных с вашего терминала (включая отцовский и сыновний процессы).**

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/5$./father\_out &

son param: pid=17514 ppid=17513

father param: pid=17513 ppid=1

Child proccess is finished with status 0

**Информация о процессах, запущенных с терминала:**

PID TTY STAT TIME COMMAND

644 ? Ss 0:00 /bin/sh /etc/xdg/xfce4/xinitrc -- /etc/X11/xinit/xserverrc

672 ? Ss 0:00 \\_ /usr/bin/ssh-agent x-session-manager

682 ? Sl 0:00 \\_ xfce4-session

697 ? S 0:10 \\_ xfwm4 --display :0.0 --sm-client-id 23d0ca590-df7a-4ac2-873c-6ead021fd190

701 ? Sl 0:05 \\_ xfce4-panel --display :0.0 --sm-client-id 2a3345f91-4103-4cfc-a333-69b46082fbfc

857 ? S 0:00 | \\_ /usr/lib/i386-linux-gnu/xfce4/panel/wrapper-1.0 /usr/lib/i386-linux-gnu/xfce4/panel/plugins/libsystray.so 19 10485809 systray Notification Area Area where notification icons appear

860 ? Sl 0:03 | \\_ /usr/lib/i386-linux-gnu/xfce4/panel/wrapper-2.0 /usr/lib/i386-linux-gnu/xfce4/panel/plugins/libxfce4powermanager.so 24 10485810 power-manager-plugin Power Manager Plugin Display the battery levels of your devices and control the brightness of your display

861 ? S 0:00 | \\_ /usr/lib/i386-linux-gnu/xfce4/panel/wrapper-1.0 /usr/lib/i386-linux-gnu/xfce4/panel/plugins/libactions.so 20 10485811 actions Action Buttons Log out, lock or other system actions

862 ? Sl 0:05 | \\_ /usr/lib/i386-linux-gnu/xfce4/panel/wrapper-2.0 /usr/lib/i386-linux-gnu/xfce4/panel/plugins/libpulseaudio-plugin.so 2 10485812 pulseaudio PulseAudio Plugin Adjust the audio volume of the PulseAudio sound system

704 ? Sl 0:08 \\_ Thunar --sm-client-id 22a8fc8b7-8c79-44df-a673-a522703b9f35 --daemon

736 ? Sl 0:06 \\_ xfdesktop --display :0.0 --sm-client-id 2db8e108c-bca8-4213-a4c7-857579757180

748 ? Sl 2:09 \\_ /usr/lib/firefox-esr/firefox-esr --sm-client-id 2a1ba9cda-65ba-430a-bfcc-948635451308

918 ? Sl 0:21 | \\_ /usr/lib/firefox-esr/firefox-esr -contentproc -childID 1 -isForBrowser -prefsLen 1 -prefMapSize 185158 -parentBuildID 20191022215001 -greomni /usr/lib/firefox-esr/omni.ja -appomni /usr/lib/firefox-esr/browser/omni.ja -appdir /usr/lib/firefox-esr/browser 748 true tab

997 ? Sl 0:14 | \\_ /usr/lib/firefox-esr/firefox-esr -contentproc -childID 3 -isForBrowser -prefsLen 6094 -prefMapSize 185158 -parentBuildID 20191022215001 -greomni /usr/lib/firefox-esr/omni.ja -appomni /usr/lib/firefox-esr/browser/omni.ja -appdir /usr/lib/firefox-esr/browser 748 true tab

1077 ? Sl 1:09 | \\_ /usr/lib/firefox-esr/firefox-esr -contentproc -childID 5 -isForBrowser -prefsLen 7355 -prefMapSize 185158 -parentBuildID 20191022215001 -greomni /usr/lib/firefox-esr/omni.ja -appomni /usr/lib/firefox-esr/browser/omni.ja -appdir /usr/lib/firefox-esr/browser 748 true tab

1105 ? Sl 0:12 | \\_ /usr/lib/firefox-esr/firefox-esr -contentproc -childID 6 -isForBrowser -prefsLen 7355 -prefMapSize 185158 -parentBuildID 20191022215001 -greomni /usr/lib/firefox-esr/omni.ja -appomni /usr/lib/firefox-esr/browser/omni.ja -appdir /usr/lib/firefox-esr/browser 748 true tab

750 ? Sl 0:15 \\_ xfce4-terminal --geometry=100x28 --display :0.0 --role=xfce4-terminal-1574529129-1417824128 --show-menubar --show-borders --hide-toolbar --active-tab --working-directory /home/asdf/Documents/yacc/ind3 --sm-client-id 274c6962e-29a6-4832-8c0b-2cfd3b23c61d

869 ? Zs 0:00 | \\_ [bash] <defunct>

17426 pts/1 Ss 0:00 | \\_ bash

17434 pts/1 S+ 0:00 | \\_ bash

17512 pts/1 S+ 0:00 | \\_ tee -a /home/asdf/logfile\_3.txt

801 ? Sl 0:00 \\_ /usr/lib/policykit-1-gnome/polkit-gnome-authentication-agent-1

805 ? Sl 0:00 \\_ light-locker

820 ? Sl 0:03 \\_ nm-applet

**17513 pts/1 S+ 0:00 ./father\_out**

**17514 pts/1 S+ 0:00 \\_ son\_out**

17515 pts/1 S+ 0:00 \\_ sh -c ps xf > file.txt

17516 pts/1 R+ 0:00 \\_ ps xf

17461 ? Sl 0:36 /opt/sublime\_text/sublime\_text /home/asdf/lab\_3/5/file.txt

17475 ? Sl 0:01 \\_ /opt/sublime\_text/plugin\_host 17461 --auto-shell-env

757 ? Ssl 0:00 xfce4-power-manager --restart --sm-client-id 242f9b279-8997-470b-986c-957c91b8d4b0

702 ? Ssl 0:00 xfsettingsd --display :0.0 --sm-client-id 23a012914-4e0f-4a17-9d4b-bd66e9755221

626 ? Ss 0:00 /lib/systemd/systemd --user

627 ? S 0:00 \\_ (sd-pam)

637 ? S<sl 0:00 \\_ /usr/bin/pulseaudio --daemonize=no

643 ? Ss 0:01 \\_ /usr/bin/dbus-daemon --session --address=systemd: --nofork --nopidfile --systemd-activation --syslog-only

686 ? Sl 0:00 \\_ /usr/lib/i386-linux-gnu/xfce4/xfconf/xfconfd

695 ? SLs 0:00 \\_ /usr/bin/gpg-agent --supervised

712 ? Ssl 0:00 \\_ /usr/lib/gvfs/gvfsd

825 ? Sl 0:08 | \\_ /usr/lib/gvfs/gvfsd-trash --spawner :1.12 /org/gtk/gvfs/exec\_spaw/0

720 ? Ssl 0:00 \\_ /usr/lib/at-spi2-core/at-spi-bus-launcher

725 ? S 0:00 | \\_ /usr/bin/dbus-daemon --config-file=/usr/share/defaults/at-spi2/accessibility.conf --nofork --print-address 3

749 ? Ssl 0:00 \\_ /usr/lib/gvfs/gvfs-udisks2-volume-monitor

765 ? Ssl 0:00 \\_ /usr/lib/i386-linux-gnu/xfce4/notifyd/xfce4-notifyd

773 ? Sl 0:00 \\_ /usr/lib/at-spi2-core/at-spi2-registryd --use-gnome-session

819 ? Sl 0:00 \\_ /usr/lib/dconf/dconf-service

853 ? Ssl 0:00 \\_ /usr/lib/gvfs/gvfsd-metadata

**3.8. Выполните создание процессов с использованием различных функций семейства exec() с разными параметрами функций семейства, приведите результаты эксперимента.**

Функции семейства exec() имеют следующие прототипы:

int execlp(const char \*file,const char \*arg0,…const char \*argN,(char \*)NULL );

int execvp(const char \*file, char \*argv[]);

int execl(const char \*path,const char \*arg0,…const char \*argN, (char \*)NULL );

int execv(const char \*path, char \*argv[]);

int execle(const char \*path, const char \*arg0,…const char \*argN,(char \*)NULL, char \*envp[]);

int execve(const char \*path, char \*argv[], char \*envp[]) и отличаются принимаемыми аргументами, на что указывает суффикс в названии.

Суффиксы l, v, p, e , а также их сочетания в именах функций определяют формат и объем аргументов, а также каталоги, в которых нужно искать загружаемую программу:

l (список) – аргументы командной строки передаются в форме списка arg0, arg1.... argN, NULL. Эту форму используют, если количество аргументов известно;

v (vector) – аргументы командной строки передаются в форме вектора argv[]. Отдельные аргументы адресуются через argv [0], argv [1]... argv [n]. Последний аргумент (argv [n]) должен быть указателем NULL;

p (path) – обозначенный по имени файл ищется не только в текущем каталоге, но и в каталогах, определенных переменной среды PATH;

e (среда или окружение) – функция ожидает список переменных окружения в виде вектора (envp []).

**Текст программы для испытания:**

#include <stdio.h>

#include <math.h>

#include <sys/resource.h>

int main(int argc, char \* argv[]) {

char\* file = "ls";

char \*path = "/bin/ls";

char \*args[] = {"ls", "-a", "-l", NULL };

char \* env[] = {(char\*)NULL };

char \* envp[] = {(char\*)NULL };

if (fork() == 0) {

switch ( (int)argv[1][0] ) {

case (int)'1':

execl("/bin/pwd", "/bin/pwd", (char \*)NULL);break;

case (int)'2':

execlp("ls", "ls", "-l", (char \*)NULL); break;

case (int)'3':

execle("/bin/whoami", "whoami", (char \*)NULL, envp); break;

case (int)'4':

execv("/bin/ls", args); break;

case (int)'5':

execvp("ls", args); break;

case (int)'6':

execvpe("ls", args, (char \*)NULL, env); break;

}

}

system("ps fx>filex8.txt");

return 0;

}

**Результат работы программы с различными аргументами**:

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/8$./8.out 1  
/home/asdf/lab\_3/8

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/8$./8.out 2  
total 20  
-rw-r--r— 1 asdf asdf 670 Nov 26 13:57 8.c  
-rwxr-xr-x 1 asdf asdf 15836 Nov 26 13:57 8.out

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/8$./8.out 3  
asdf

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/8$./8.out 4  
total 28  
drwxr-xr-x 2 asdf asdf 4096 Nov 26 13:57 .  
drwxr-xr-x 8 asdf asdf 4096 Nov 26 13:34 ..  
-rw-r--r— 1 asdf asdf 670 Nov 26 13:57 8.c  
-rwxr-xr-x 1 asdf asdf 15836 Nov 26 13:57 8.out

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/8$./8.out 5  
total 28  
drwxr-xr-x 2 asdf asdf 4096 Nov 26 13:57 .  
drwxr-xr-x 8 asdf asdf 4096 Nov 26 13:34 ..  
-rw-r--r— 1 asdf asdf 670 Nov 26 13:57 8.c  
-rwxr-xr-x 1 asdf asdf 15836 Nov 26 13:57 8.out

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/8$./8.out 6  
total 28  
drwxr-xr-x 2 asdf asdf 4096 Nov 26 13:57 .  
drwxr-xr-x 8 asdf asdf 4096 Nov 26 13:34 ..  
-rw-r--r— 1 asdf asdf 670 Nov 26 13:57 8.c  
-rwxr-xr-x 1 asdf asdf 15836 Nov 26 13:57 8.out

**3.9. Проанализируйте значение, возвращаемое функцией wait(&status). Предложите эксперимент, позволяющий родителю отслеживать подмножество порожденных потомков, используя различные функции семейства wait().**

Программа-родитель порождает 3 потока, каждый из которых имеет различное завершение, а затем в цикле вызывает функцию waitpid() с pid каждого из процессов-потомков, после чего выводит результат завершения потомка.

**Содержимое father.c:**

#include <stdio.h>

#include <sys/types.h>

#include <wait.h>

int main()

{

int i, pid[4], ppid, status, result;

pid[0]=getpid();

ppid=getppid();

printf("\nFATHER PARAMS: pid=%i ppid=%i\n", pid[0],ppid);

if((pid[1] = fork()) == 0)

execl("son1.out", "son1.out", NULL);

if((pid[2] = fork()) == 0)

execl("son2.out", "son2.out", NULL);

if((pid[3] = fork()) == 0)

execl("son3.out", "son3.out", NULL);

system("ps xf > file.txt");

for (i = 1; i < 4; i++)

{

result = waitpid(pid[i], &status, WUNTRACED);

printf("\n%d) Child proccess with pid = %d is finished with status %d\n", i, result, status);

}

return 0;

}

**Содержимое son1.c:**

#include <stdio.h>

int main()

{

int pid,ppid;

pid=getpid();

ppid=getppid();

printf("\nSON PARAMS: pid=%i ppid=%i\n",pid,ppid);

sleep(15);

return 0;

}

**Содержимое son2.c:**

#include <stdio.h>

#include <stdlib.h>

int main()

{

int pid,ppid;

pid=getpid();

ppid=getppid();

printf("\nSON PARAMS: pid=%i ppid=%i\n",pid,ppid);

sleep(15);

exit(1);

}

**Содержимое son3.c:**

#include <stdio.h>

#include <stdlib.h>

int main()

{

int pid,ppid;

pid=getpid();

ppid=getppid();

printf("\nSON PARAMS: pid=%i ppid=%i\n",pid,ppid);

sleep(15);

exit(-1);

}

**Результат работы программы:**

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/9$./father.out  
SON PARAMS: pid=18092 ppid=18090  
SON PARAMS: pid=18094 ppid=18090  
SON PARAMS: pid=18093 ppid=18090

father params: pid=18090 ppid=17434

1) Child proccess with pid = 18092 is finished with status 0

2) Child proccess with pid = 18093 is finished with status 256

3) Child proccess with pid = 18094 is finished with status 65280

Можно заметить, что вместо 1 и -1 были получены числа 256 и 65280. Если перевести их в двоичный код, получатся числа 100000000 и 1111111100000000 соответственно. Эти числа являются числами 1 и -1 в дополнительном коде, к которым было присоединено еще 8 нулевых разрядов.

**3.10. Проанализируйте очередность исполнения процессов.**

**3.10.1 Очередность исполнения процессов, порожденных вложенными вызовами fork().**

Были созданы 5 файлов: father.c, son1.c, son2.c, son3.c, son4.c.

father.c с помощью функции fork() вызывает процессы son1 И son2, каждый из которых с помощью fork() вызывает еще по одному son3 и son4 соответственно. В конце каждого из файлов есть вывод, сообщающий о том, что файл выполняет свои вычисления. Для каждого из процессов это происходит уже после того, как все функции fork() вызваны.

**Содержимое father.c:**

#include <stdio.h>

int main()

{

int i, pid[3], ppid, status, result;

pid[0]=getpid();

ppid=getppid();

printf("\nFATHER PARAMS: pid=%i ppid=%i\n", pid[0],ppid);

if((pid[1] = fork()) == 0)

execl("son1.out", "son1.out", NULL);

if((pid[2] = fork()) == 0)

execl("son2.out", "son2.out", NULL);

system("ps xf > file.txt");

printf("father process is doing its calculations");

return 0;

}

**Содержимое son1.c:**

#include <stdio.h>

int main()

{

int pid[2],ppid;

pid[0]=getpid();

ppid=getppid();

printf("\nSON PARAMS: pid=%i ppid=%i\n",pid[0],ppid);

if((pid[1] = fork()) == 0)

execl("son3.out", "son3.out", NULL);

printf("son, called from first fork is doing its calculations\n");

return 0;

}

**Содержимое son2.c:**

#include <stdio.h>

int main()

{

int pid[2],ppid;

pid[0]=getpid();

ppid=getppid();

printf("\nSON PARAMS: pid=%i ppid=%i\n",pid[0],ppid);

if((pid[1] = fork()) == 0)

execl("son4.out", "son4.out", NULL);

printf("son, called from second fork is doing its calculations\n");

return 0;

}

**Содержимое son3.c:**

#include <stdio.h>

int main()

{

int pid,ppid;

pid=getpid();

ppid=getppid();

printf("\nSON PARAMS: pid=%i ppid=%i\n",pid,ppid);

printf("son, called from fork in first son is doing its calculations\n");

return 0;

}

**Содержимое son4.c:**

#include <stdio.h>

int main()

{

int pid,ppid;

pid=getpid();

ppid=getppid();

printf("\nSON PARAMS: pid=%i ppid=%i\n",pid,ppid);

printf("son, called from fork in second son is doing its calculations\n");

return 0;

}

**Результат работы программы:**

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/10$./father.out  
  
SON PARAMS: pid=18522 ppid=18519  
son, called from second fork is doing its calculations  
son, called from first fork is doing its calculations  
  
SON PARAMS: pid=18526 ppid=1  
son, called from fork in first son is doing its calculations  
  
SON PARAMS: pid=18525 ppid=1  
son, called from fork in second son is doing its calculations  
  
FATHER PARAMS: pid=18519 ppid=17434  
father process is doing its calculations

Можно сделать вывод, что до части с вычислениями процессы дошли в таком порядке:

son2->son1->son3->son4->father

Из этого можно сделать вывод, что в первую очередь выполняются процессы, вызванные первым fork(), а затем процессы, вызванные вложенными fork() и наконец последним завершается процесс-родитель.

**3.10.2. Измените процедуру планирования с помощью функции с шаблоном scheduler в ее названии и повторите эксперимент.**

Программы son3.c и son4.c остались неизменными, а в father.c, son1.c и son2.c были добавлены функции для изменения вызываемых процессов. Для всех процессов была назначена процедура планирования FIFO, для процесса father приоритет был назначен 10, для son1 - 2, для остальных - 1.

**Содержимое father.c:**

#include <stdio.h>

#include <sched.h>

int main()

{

int i, pid[3], ppid, status, result;

struct sched\_param shdprm;

pid[0]=getpid();

ppid=getppid();

printf("\nFATHER PARAMS: pid=%i ppid=%i\n", pid[0],ppid);

shdprm.sched\_priority = 10;

if (sched\_setscheduler (0, SCHED\_FIFO, &shdprm) == -1) {

perror ("SCHED\_SETSCHEDULER");

}

if((pid[1] = fork()) == 0)

{

shdprm.sched\_priority = 2;

if (sched\_setscheduler (pid[1], SCHED\_FIFO, &shdprm) == -1) {

perror ("SCHED\_SETSCHEDULER");

}

execl("son1.out", "son1.out", NULL);

}

if((pid[2] = fork()) == 0)

{

shdprm.sched\_priority = 1;

if (sched\_setscheduler (pid[2], SCHED\_FIFO, &shdprm) == -1) {

perror ("SCHED\_SETSCHEDULER");

}

execl("son2.out", "son2.out", NULL);

}

int unused=1;

for (int i=0;i<50000;i++){

unused=unused+3;

}

system("ps xf > file.txt");

printf("father process is doing its calculations");

return 0;

}

**Содержимое son1.c:**

#include <stdio.h>

#include <sched.h>

int main()

{

int pid[2],ppid;

struct sched\_param shdprm;

pid[0]=getpid();

ppid=getppid();

printf("\nSON PARAMS: pid=%i ppid=%i\n",pid[0],ppid);

if((pid[1] = fork()) == 0)

{

shdprm.sched\_priority = 1;

if (sched\_setscheduler (pid[1], SCHED\_FIFO, &shdprm) == -1) {

perror ("SCHED\_SETSCHEDULER");

}

execl("son3.out", "son3.out", NULL);

}

int unused=1;  
 for (int i=0;i<50000;i++){  
 unused=unused+3;  
 }

printf("son, called from first fork is doing its calculations\n");

return 0;

}

**Содержимое son2.c:**

#include <stdio.h>

#include <sched.h>

int main()

{

int pid[2],ppid;

struct sched\_param shdprm;

pid[0]=getpid();

ppid=getppid();

printf("\nSON PARAMS: pid=%i ppid=%i\n",pid[0],ppid);

if((pid[1] = fork()) == 0)

{

shdprm.sched\_priority = 1;

if (sched\_setscheduler (pid[1], SCHED\_FIFO, &shdprm) == -1) {

perror ("SCHED\_SETSCHEDULER");

}

execl("son4.out", "son4.out", NULL);

}

int unused=1;  
 for (int i=0;i<50000;i++){  
 unused=unused+3;  
 }

printf("son, called from second fork is doing its calculations\n");

return 0;

}

**Результат работы программы:**

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/10/10\_2$sudo ./father.out

FATHER PARAMS: pid=18682 ppid=18680  
father process is doing its calculations

SON PARAMS: pid=18683 ppid=18682  
son, called from first fork is doing its calculations

SON PARAMS: pid=18686 ppid=1  
son, called from fork in first son is doing its calculations

SON PARAMS: pid=18684 ppid=18682  
son, called from second fork is doing its calculations

SON PARAMS: pid=18687 ppid=1  
son, called from fork in second son is doing its calculations

Теперь порядок выполнения стал следующим:

father->son1->son3->son2->son4

Ожидаемо, процессы выполнялись по порядку их приоритетов.

**3.10.3 Поменяйте порядок очереди в RR-процедуре.**

**Содержимое father.c:**

#include <stdio.h>

#include <sched.h>

int main()

{

int i, pid[3], ppid, status, result;

struct sched\_param shdprm;

pid[0]=getpid();

ppid=getppid();

printf("\nFATHER PARAMS: pid=%i ppid=%i\n", pid[0],ppid);

shdprm.sched\_priority = 90;

if (sched\_setscheduler (0, SCHED\_RR, &shdprm) == -1) {

perror ("SCHED\_SETSCHEDULER");

}

if((pid[1] = fork()) == 0)

{

shdprm.sched\_priority = 2;

if (sched\_setscheduler (pid[1], SCHED\_RR, &shdprm) == -1) {

perror ("SCHED\_SETSCHEDULER");

}

execl("son1.out", "son1.out", NULL);

}

if((pid[2] = fork()) == 0)

{

shdprm.sched\_priority = 80;

if (sched\_setscheduler (pid[2], SCHED\_RR, &shdprm) == -1) {

perror ("SCHED\_SETSCHEDULER");

}

execl("son2.out", "son2.out", NULL);

}

int unused=1;

for (int i=0;i<50000;i++){

unused=unused+3;

}

system("ps xf > file.txt");

printf("father process is doing its calculations");

return 0;

}

**Содержимое son1.c:**

#include <stdio.h>

#include <sched.h>

int main()

{

int pid[2],ppid;

struct sched\_param shdprm;

pid[0]=getpid();

ppid=getppid();

printf("\nSON PARAMS: pid=%i ppid=%i\n",pid[0],ppid);

if((pid[1] = fork()) == 0)

{

shdprm.sched\_priority = 1;

if (sched\_setscheduler (pid[1], SCHED\_RR, &shdprm) == -1) {

perror ("SCHED\_SETSCHEDULER");

}

execl("son3.out", "son3.out", NULL);

}

int unused=1;  
 for (int i=0;i<50000;i++){  
 unused=unused+3;  
 }

printf("son, called from first fork is doing its calculations\n");

return 0;

}

**Содержимое son2.c:**

#include <stdio.h>

#include <sched.h>

int main()

{

int pid[2],ppid;

struct sched\_param shdprm;

pid[0]=getpid();

ppid=getppid();

printf("\nSON PARAMS: pid=%i ppid=%i\n",pid[0],ppid);

if((pid[1] = fork()) == 0)

{

shdprm.sched\_priority = 1;

if (sched\_setscheduler (pid[1], SCHED\_RR, &shdprm) == -1) {

perror ("SCHED\_SETSCHEDULER");

}

execl("son4.out", "son4.out", NULL);

}

int unused=1;  
 for (int i=0;i<50000;i++){  
 unused=unused+3;  
 }

printf("son, called from second fork is doing its calculations\n");

return 0;

}

**Результат работы программы:**

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/10/10\_3$sudo ./father.out  
  
FATHER PARAMS: pid=19435 ppid=19433  
father process is doing its calculation

SON PARAMS: pid=19437 ppid=19435  
son, called from second fork is doing its calculations

SON PARAMS: pid=19439 ppid=1  
son, called from fork in second son is doing its calculations  
  
SON PARAMS: pid=19436 ppid=19435  
son, called from first fork is doing its calculations  
  
SON PARAMS: pid=19440 ppid=1  
son, called from fork in first son is doing its calculations

С помощью SCHED\_RR удалось изменить порядок выполнения процессов, теперь он следующий:

father->son2->son4->son1->son3

**3.10.4 Можно ли задать разные процедуры планирования разным процессам с одинаковыми приоритетами. Как они будут конкурировать, подтвердите экспериментально.**

Для проверки были модифицированы файлы father.c, son1.c и son2.c. Теперь для всех процессов приоритет равен 10, но при этом процесс father является SCHED\_OTHER, процессы son1 и son4 - SCHED\_FIFO, а son2 и son3 - SCHED\_RR.

**Содержимое father.c:**

#include <stdio.h>

#include <sched.h>

int main()

{

int i, pid[3], ppid, status, result;

struct sched\_param shdprm;

pid[0]=getpid();

ppid=getppid();

printf("\nFATHER PARAMS: pid=%i ppid=%i\n", pid[0],ppid);

shdprm.sched\_priority = 10;

if((pid[1] = fork()) == 0)

{

shdprm.sched\_priority = 10;

if (sched\_setscheduler (pid[1], SCHED\_ FIFO, &shdprm) == -1) {

perror ("SCHED\_SETSCHEDULER");

}

execl("son1.out", "son1.out", NULL);

}

if((pid[2] = fork()) == 0)

{

shdprm.sched\_priority = 10;

if (sched\_setscheduler (pid[2], SCHED\_RR, &shdprm) == -1) {

perror ("SCHED\_SETSCHEDULER");

}

execl("son2.out", "son2.out", NULL);

}

int unused=1;

for (int i=0;i<50000;i++){

unused=unused+3;

}

system("ps xf > file.txt");

printf("father process is doing its calculations");

return 0;

}

**Содержимое son1.c:**

#include <stdio.h>

#include <sched.h>

int main()

{

int pid[2],ppid;

struct sched\_param shdprm;

pid[0]=getpid();

ppid=getppid();

printf("\nSON PARAMS: pid=%i ppid=%i\n",pid[0],ppid);

if((pid[1] = fork()) == 0)

{

shdprm.sched\_priority = 10;

if (sched\_setscheduler (pid[1], SCHED\_RR, &shdprm) == -1) {

perror ("SCHED\_SETSCHEDULER");

}

execl("son3.out", "son3.out", NULL);

}

int unused=1;  
 for (int i=0;i<50000;i++){  
 unused=unused+3;  
 }

printf("son, called from first fork is doing its calculations\n");

return 0;

}

**Содержимое son2.c:**

#include <stdio.h>

#include <sched.h>

int main()

{

int pid[2],ppid;

struct sched\_param shdprm;

pid[0]=getpid();

ppid=getppid();

printf("\nSON PARAMS: pid=%i ppid=%i\n",pid[0],ppid);

if((pid[1] = fork()) == 0)

{

shdprm.sched\_priority = 10;

if (sched\_setscheduler (pid[1], SCHED\_ FIFO, &shdprm) == -1) {

perror ("SCHED\_SETSCHEDULER");

}

execl("son4.out", "son4.out", NULL);

}

int unused=1;  
 for (int i=0;i<50000;i++){  
 unused=unused+3;  
 }

printf("son, called from second fork is doing its calculations\n");

return 0;

}

**Результат работы программы:**

FATHER PARAMS: pid=19548 ppid=19546

SON PARAMS: pid=19550 ppid=19548  
son, called from second fork is doing its calculations

SON PARAMS: pid=19553 ppid=1  
son, called from fork in second son is doing its calculations

SON PARAMS: pid=19549 ppid=19548  
son, called from first fork is doing its calculations  
  
SON PARAMS: pid=19552 ppid=1  
son, called from fork in first son is doing its calculations  
  
father process is doing its calculations

Видно, что несмотря на разные процедуры планирования, все процессы были выполнены. Порядок выполнения процессов:

son2->son4->son1->son3->father

При изменении процедуры планирования так, чтобы son1 и son4 имели SCHED\_FIFO, а son2 и son3 - SCHED\_RR получаются полностью идентичные результаты из чего можно сделать вывод, что при одинаковом приоритете не имеет значения выбрана процедура планирования SCHED\_RR или SCHED\_FIFO, так как в данном случае на выполнение процесса уходит времени меньше, чем величина кванта.

**3.11. Определите величину кванта. Можно ли ее поменять? – для обоснования проведите эксперимент**

Для определения длины кванта можно воспользоваться функцией

int sched\_rr\_get\_interval(pid\_t, struct timespec \*);

**Текст программы для определения величины кванта:**

#include <stdio.h>

#include <sched.h>

#include <sys/mman.h>

int main(void)

{

struct timespec qp;

struct sched\_param shdprm;

shdprm.sched\_priority = 50;

if (sched\_setscheduler (0, SCHED\_RR, &shdprm) == -1)

perror ("SCHED\_SETSCHEDULER\_1");

if (sched\_rr\_get\_interval (0, &qp) == 0)

printf ("Квант при циклическом планировании: %g сек\n",qp.tv\_sec + qp.tv\_nsec / 1000000000.0);

else

perror ("SCHED\_RR\_GET\_INTERVAL");

return 0;

}

**Результат работы программы:**

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/11$sudo ./11.out  
Квант при циклическом планировании: 0.1 сек

В рамках одной политики планирования для большинства ОС Linux на значение кванта не влияет ни количество процессов, ни их загруженность, ни их иные характеристики, для всех процессов величина кванта одинакова и постоянна.

Определим величину кванта для SCHED\_OTHER:

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/11$sudo ./11.out  
Квант при циклическом планировании: 0.008 сек

Величина кванта для процедуры планирования SCHED\_OTHER оказалась значительно меньше, чем для SCHED\_RR.

Определим теперь ее для SCHED\_FIFO:

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/11$sudo ./11.out  
Квант при циклическом планировании: 0 сек

Ожидаемо для процедуры SCHED\_FIFO величина кванта равна 0.

Проверим возможность изменения кванта с помощью функции nice()

**Текст программы для изменения величины кванта:**

#include <stdio.h>

#include <sched.h>

#include <sys/mman.h>

int main(void)

{

struct timespec qp;

struct sched\_param shdprm;

shdprm.sched\_priority = 50;

int n;

if (sched\_setscheduler (0, SCHED\_RR, &shdprm) == -1)

perror ("SCHED\_SETSCHEDULER\_1");

if (sched\_rr\_get\_interval (0, &qp) == 0)

printf ("Квант при циклическом планировании: %g сек\n",qp.tv\_sec + qp.tv\_nsec / 1000000000.0);

else

perror ("SCHED\_RR\_GET\_INTERVAL");

if ((n = nice(100000)) == -1)

perror("NICE");

else

printf ("Nice value = %d\n", n);

if (sched\_rr\_get\_interval (0, &qp) == 0)

printf ("Квант при циклическом планировании: %g сек\n",qp.tv\_sec + qp.tv\_nsec / 1000000000.0);

else

perror ("SCHED\_RR\_GET\_INTERVAL");

return 0;

}

**Результат работы программы:**

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/11$sudo ./11\_2.out  
Квант при циклическом планировании: 0.1 сек  
Nice value = 19  
Квант при циклическом планировании: 0.1 сек

Несмотря на то, что функция nice() не возвращает -1, изменить величину кванта не удается.

**3.12. Проанализируйте наследование на этапах fork() и exec(). Проведите эксперимент с родителем и потомками по доступу к одним и тем же файлам, открытым родителем. Аналогичные эксперименты проведите по отношению к другим параметрам.**

Для проверки наследования доступа к файлам созданы программы father.c и son.c. father запускает 2 процесса son, каждый из которых считывает входной файл и записывает считанное в выходной файл.

**Содержание father.c:**

#include <stdio.h>

#include <sched.h>

#include <sys/mman.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

void itoa(char \*buf, int value) {

sprintf(buf, "%d", value);

}

int main (void)

{

int i, pid, ppid, status;

int fdrd, fdwr;

char str1[10], str2[10];

char c;

struct sched\_param shdprm;

if (mlockall((MCL\_CURRENT | MCL\_FUTURE)) < 0)

perror("mlockall error");

pid = getpid();

ppid = getppid();

shdprm.sched\_priority = 1;

if (sched\_setscheduler (0, SCHED\_RR, &shdprm) == -1)

perror ("SCHED\_SETSCHEDULER\_1");

if ((fdrd = open("infile.txt",O\_RDONLY)) == -1)

perror("Openning file");

if ((fdwr = creat("outfile.txt",0666)) == -1)

perror("Creating file");

itoa(str1, fdrd);

itoa(str2, fdwr);

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

if(fork() == 0)

{

execl("son.out", "son.out", str1, str2, NULL);

}

if (close(fdrd) != 0)

perror("Closing file");

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

printf("Процесс с pid = %d завершен\n", wait(&status));

return 0;

}

**Содержание son.c:**

#include <sched.h>

#include <sys/mman.h>

#include <fcntl.h>

#include <stdio.h>

int main(int argc, char \*argv[])

{

if (mlockall((MCL\_CURRENT | MCL\_FUTURE)) < 0)

perror("mlockall error");

char c;

int pid, ppid, buf;

int fdrd = atoi(argv[1]);

int fdwr = atoi(argv[2]);

pid=getpid();

ppid=getppid();

printf("son file decriptor = %d\n", fdrd);

printf("son params: pid=%i ppid=%i\n",pid,ppid);

for(;;)

{

sleep(5);

if (read(fdrd,&c,1) != 1)

return;

write(fdwr,&c,1);

printf("pid = %d: %c\n", pid, c);

//if (close(fdrd) != 0)

// perror("Closing file");

}

return 0;

**Содержание input.txt:**

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/12$cat input.txt  
Something here!

**Результат работы программы:**

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/12$sudo ./father.out  
son file decriptor = 3  
son params: pid=19901 ppid=19900  
son file decriptor = 3  
son params: pid=19902 ppid=19900  
pid = 19901: S  
pid = 19902: o  
pid = 19901: m  
pid = 19902: e  
pid = 19901: t  
pid = 19902: h  
pid = 19901: i  
pid = 19902: n  
pid = 19901: g  
pid = 19902:  
pid = 19901: h  
pid = 19902: e  
pid = 19901: r  
pid = 19902: e  
pid = 19901: !  
Процесс с pid = 19901 завершен  
Процесс с pid = 19902 завершен

Видно, что процессы по очереди считывают текст, при этом так как дескрипторы fdrd в обоих потомках указывают на запись в таблице файлов, соответствующую исходному файлу, ядро смещает внутрифайловые указатели после каждой операции чтения, а значит каждый процесс считывает лишь половину файла. Так как процессы вызывают функции записи и чтения попеременно в выходной файл записывается всё содержимое входного файла:

Something here!

От родителя наследуются UID, GID, RUID, PGID и политика планирования процессов.

**3.13.1. Изменяя длительности выполнения процессов и параметры системных вызовов, рассмотрите 3 ситуации и получите соответствующие таблицы процессов:**

**а) процесс-отец запускает процесс-сын и ожидает его завершения;**

**Содержимое father.c:**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <wait.h>

#include <string.h>

int main(int argc, char \*argv[])

{

int sid, pid, pid1, ppid, status;

char command[50];

pid = getpid();

ppid = getppid();

sid = getsid(pid);

sprintf(command, "ps xjf | grep \"STAT\\|%d\" > %s", sid, argv[1]);

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

if((pid1=fork())==0) execl("son.out","son.out", NULL);

system(command);

waitpid(pid1, &status, WNOHANG);

}

**Содержимое son.c:**

#include <stdio.h>

void main()

{

int pid,ppid;

pid=getpid();

ppid=getppid();

printf(" SON\_1 PARAMS : pid=%i ppid=%i\n Father creates and waits \n",pid, ppid);

sleep(3);

}

**Результат выполнения программы:**

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/13$./father.out output.txt  
 FATHER PARAMS: sid = 17426 pid=20195 ppid=17434  
 SON\_1 PARAMS : pid=20197 ppid=20195  
 Father creates and waits

Процесс сына создался выполнил свою работы и завершился после завершился процесс отца

**Содержимое файла output.txt:**

PPID PID PGID SID TTY TPGID STAT UID TIME COMMAND

750 17426 17426 17426 pts/1 17434 Ss 1000 0:00 | \\_ bash

17426 17434 17434 17426 pts/1 17434 S+ 1000 0:00 | \\_ bash

17434 20195 17434 17426 pts/1 17434 S+ 1000 0:00 | \\_ ./father.out output.txt

20195 20197 17434 17426 pts/1 17434 S+ 1000 0:00 | | \\_ son.out

20195 20198 17434 17426 pts/1 17434 S+ 1000 0:00 | | \\_ sh -c ps xjf | grep "STAT\|17426" > output.txt

20198 20199 17434 17426 pts/1 17434 R+ 1000 0:00 | | \\_ ps xjf

20198 20200 17434 17426 pts/1 17434 S+ 1000 0:00 | | \\_ grep STAT\|17426

17434 20196 17434 17426 pts/1 17434 S+ 1000 0:00 | \\_ tee -a /home/asdf/logfile\_3.txt

**б) процесс-отец запускает процесс-сын и, не ожидая его завершения, завершает свое выполнение. Зафиксируйте изменение родительского идентификатора процесса-сына**

**Содержимое father.c:**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <wait.h>

#include <string.h>

int main(int argc, char \*argv[])

{

int sid, pid, pid1, ppid, status;

char command[50];

pid = getpid();

ppid = getppid();

sid = getsid(pid);

sprintf(command, "ps xjf | grep \"STAT\\|%d\" > %s", sid, argv[1]);

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

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

system(command);

}

**Содержимое son.c:**

#include <stdio.h>

void main(int argc, char \*argv[])

{

int pid,ppid;

pid=getpid();

ppid=getppid();

printf("SON\_2 PARAMS: pid=%i ppid=%i\n Father finished before son termination without waiting for it \n",pid,ppid);

sleep(5);

ppid = getppid();

printf("SON\_2 PARAMS ARE CHANGED: pid=%i ppid=%i\n",pid,ppid);

}

**Результат выполнения программы**:

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/13/13\_2$./father.out output.txt  
 FATHER PARAMS: sid = 17426 pid=20363 ppid=17434  
 SON\_2 PARAMS: pid=20365 ppid=20363  
 Father finished before son termination without waiting for it  
 SON\_2 PARAMS ARE CHANGED: pid=20365 ppid=1

Процесс отца был завершен до выполнения работы потомка, следовательно, потомку присвоен новый PPID.

**Содержимое файла output.txt:**

PPID PID PGID SID TTY TPGID STAT UID TIME COMMAND

750 17426 17426 17426 pts/1 17434 Ss 1000 0:00 | \\_ bash

17426 17434 17434 17426 pts/1 17434 S+ 1000 0:00 | \\_ bash

17434 20363 17434 17426 pts/1 17434 S+ 1000 0:00 | \\_ ./father.out output.txt

20363 20365 17434 17426 pts/1 17434 S+ 1000 0:00 | | \\_ son.out

20363 20366 17434 17426 pts/1 17434 S+ 1000 0:00 | | \\_ sh -c ps xjf | grep "STAT\|17426" > output.txt

20366 20367 17434 17426 pts/1 17434 R+ 1000 0:00 | | \\_ ps xjf

20366 20368 17434 17426 pts/1 17434 S+ 1000 0:00 | | \\_ grep STAT\|17426

17434 20364 17434 17426 pts/1 17434 S+ 1000 0:00 | \\_ tee -a /home/asdf/logfile\_3.txt

**в) процесс-отец запускает процесс-сын и не ожидает его завершения; процесс-сын завершает свое выполнение. Зафиксируйте появление процесса-зомби, для этого включите команду ps в программу father.с**

**Содержимое father.c:**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <wait.h>

#include <string.h>

int main(int argc, char \*argv[])

{

int sid, pid, pid1, ppid, status;

char command[50];

pid = getpid();

ppid = getppid();

sid = getsid(pid);

sprintf(command, "ps xjf | grep \"STAT\\|%d\" > %s", sid, argv[1]);

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

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

system(command);

}

**Содержимое son.c:**

#include <stdio.h>

void main()

{

int pid,ppid;

pid=getpid();

ppid=getppid();

printf("SON\_3 PARAMS: pid=%i ppid=%i\nson3 terminated – ZOMBIE\n",pid,ppid);

ppid=getppid();

printf("SON\_3 PARAMS: pid=%i ppid=%i\n",pid,ppid);

}

**Результат Выполнения программы:**

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/13/13\_3$./father.out output.txt

FATHER PARAMS: sid = 17426 pid=20497 ppid=1743

SON\_3 PARAMS: pid=20499 ppid=20497

son3 terminated – ZOMBIE

SON\_3 PARAMS: pid=20499 ppid=20497

Видно появление процесса зомби

**Содержимое файла output.txt:**

PPID PID PGID SID TTY TPGID STAT UID TIME COMMAND

750 17426 17426 17426 pts/1 17434 Ss 1000 0:00 | \\_ bash

17426 17434 17434 17426 pts/1 17434 S+ 1000 0:00 | \\_ bash

17434 20497 17434 17426 pts/1 17434 S+ 1000 0:00 | \\_ ./father.out output.txt

20497 20499 17434 17426 pts/1 17434 Z+ 1000 0:00 | | \\_ [son.out] <defunct>

20497 20500 17434 17426 pts/1 17434 S+ 1000 0:00 | | \\_ sh -c ps xjf | grep "STAT\|17426" > output.txt

20500 20501 17434 17426 pts/1 17434 R+ 1000 0:00 | | \\_ ps xjf

20500 20502 17434 17426 pts/1 17434 S+ 1000 0:00 | | \\_ grep STAT\|17426

17434 20498 17434 17426 pts/1 17434 S+ 1000 0:00 | \\_ tee -a /home/asdf/logfile\_3.txt

**3.13.2. Перенаправьте вывод не только на терминал, но и в файл. Организуйте программу многопроцессного функционирования так, чтобы результатом ее работы была демонстрация всех трех ситуаций с отображением в итоговом файле.**

**Содержимое father.c:**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <wait.h>

#include <string.h>

int main(int argc, char \*argv[])

{

int sid, pid, pid1, ppid, status;

char command[50];

if (argc < 2)

return -1;

pid = getpid();

ppid = getppid();

sid = getsid(pid);

sprintf(command, "ps xjf | grep \"STAT\\|%d\" > %s", sid, argv[1]);

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

if((pid1=fork())==0) execl("son1.out","son1.out", NULL);

if(fork()==0) execl("son2.out","son2.out", argv[1], NULL);

if(fork()==0) execl("son3.out","son3.out", NULL);

system(command);

waitpid(pid1, &status, WNOHANG);

}

**Результат выполнения программы:**

asdf@BigAwesomeTurtle:/home/asdf/lab\_3/13/13\_final$./father.out output.txt  
 FATHER PARAMS: sid = 17426 pid=20617 ppid=17434

SON\_3 PARAMS: pid=20621 ppid=20617  
 son3 terminated – ZOMBIE  
 SON\_3 PARAMS: pid=20621 ppid=20617  
 SON\_1 PARAMS : pid=20619 ppid=20617  
 Father creates and waits  
 SON\_2 PARAMS: pid=20620 ppid=20617  
 Father finished before son termination without waiting for it  
 SON\_2 PARAMS ARE CHANGED: pid=20620 ppid=1

**Содержимое файла output.txt:**

PPID PID PGID SID TTY TPGID STAT UID TIME COMMAND

750 17426 17426 17426 pts/1 17434 Ss 1000 0:00 | \\_ bash

17426 17434 17434 17426 pts/1 17434 S+ 1000 0:00 | \\_ bash

17434 20617 17434 17426 pts/1 17434 S+ 1000 0:00 | \\_ ./father.out output.txt

20617 20619 17434 17426 pts/1 17434 S+ 1000 0:00 | | \\_ son1.out

20617 20620 17434 17426 pts/1 17434 S+ 1000 0:00 | | \\_ son2.out output.txt

20617 20621 17434 17426 pts/1 17434 Z+ 1000 0:00 | | \\_ [son3.out] <defunct>

20617 20622 17434 17426 pts/1 17434 S+ 1000 0:00 | | \\_ sh -c ps xjf | grep "STAT\|17426" > output.txt

20622 20623 17434 17426 pts/1 17434 R+ 1000 0:00 | | \\_ ps xjf

20622 20624 17434 17426 pts/1 17434 S+ 1000 0:00 | | \\_ grep STAT\|17426

17434 20618 17434 17426 pts/1 17434 S+ 1000 0:00 | \\_ tee -a /home/asdf/logfile\_3.txt

**3.13.3. С помощью команды kill -l ознакомьтесь с перечнем сигналов, поддерживаемых процессами. Ознакомьтесь с системными вызовами kill(2), signal(2). Подготовьте программы следующего содержания: а.) процесс father порождает процессы son1, son2, son3 и запускает на исполнение программные коды из соответствующих исполнительных файлов; б.) далее родительский процесс осуществляет управление потомками, для этого он генерирует сигнал каждому пользовательскому процессу; в.) в пользовательских процессах-потомках необходимо обеспечить: для son1 - реакцию на сигнал по умолчанию; для son2 - реакцию игнорирования; для son3 - перехватывание и обработку сигнала. Сформируйте файл-проект из четырех файлов, откомпилируйте, запустите программу. Проанализируйте таблицу процессов до и после посылки сигналов с помощью системного вызова system("ps -s >> file"). Обратите внимание на реакцию, устанавливаемую для последнего потомка**

**Содержание father.c:**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <wait.h>

#include <string.h>

#include <signal.h>

int main(int argc, char \*argv[])

{

int sid, pid, pid1, pid2, pid3, ppid, status;

char command[50];

pid = getpid();

ppid = getppid();

sid = getsid(pid);

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

if((pid1=fork())==0) execl("son1.out","son1.out", NULL);

if((pid2 = fork())==0) execl("son2.out","son2.out", argv[1], NULL);

if((pid3 = fork())==0) execl("son3.out","son3.out", NULL);

system("ps -s>>output.txt");

kill(pid1,SIGUSR2);

kill(pid2,SIGUSR2);

kill(pid3,SIGUSR2);

sleep(2);

system("ps -s>>output.txt");

wait(&status);

wait(&status);

wait(&status);

}

**Содержание son1.c:**

#include <stdio.h>

#include <signal.h>

void main()

{

int pid,ppid;

pid=getpid();

ppid=getppid();

printf(" SON\_1 PARAMS : pid=%i ppid=%i\nFather creates and waits \n", pid,

ppid);

signal(SIGUSR2, SIG\_DFL);

sleep(1);

}

Обработка сигнала по умолчанию

**Содержание son2.c:**

#include <stdio.h>

#include <signal.h>

void main()

{

int pid,ppid;

pid=getpid();

ppid=getppid();

printf(" SON\_2 PARAMS : pid=%i ppid=%i\nFather creates and waits \n", pid,

ppid);

signal(SIGUSR2, SIG\_IGN);

sleep(1);

}

Игнорирование сигнала

**Содержание son3.c:**

#include <stdio.h>

#include <signal.h>

void handler()

{

puts("^C - signal received");

signal(SIGINT, SIG\_DFL);

}

void main()

{

int pid,ppid;

pid=getpid();

ppid=getppid();

printf(" SON\_3 PARAMS : pid=%i ppid=%i\nFather creates and waits \n", pid,

ppid);

signal(SIGUSR2, handler);

while(1);

sleep(1);

}

Ловля сигнала ^C

**Результат выполнения программы:**

asdf@BigAwesomeTurtle:~/lab\_3/13\_2$ ./father.out

FATHER PARAMS: sid = 17426 pid=20797 ppid=17426

SON\_1 PARAMS : pid=20798 ppid=20797

Father creates and waits

SON\_2 PARAMS : pid=20799 ppid=20797

Father creates and waits

SON\_3 PARAMS : pid=20800 ppid=20797

Father creates and waits

^C - signal received

**Содержимое файла output.txt:**

UID PID PENDING BLOCKED IGNORED CAUGHT STAT TTY TIME COMMAND

1000 17426 0000000000000000 0000000000010000 0000000000380004 000000004b817efb Ss pts/1 0:00 bash

1000 17434 0000000000000000 0000000000010000 0000000000000004 0000000008010002 S+ pts/1 0:01 bash

1000 20786 0000000000000000 0000000000010000 0000000000000006 0000000000000000 S+ pts/1 0:00 ./father.out

1000 20787 0000000000000000 0000000000000000 0000000000000000 0000000000000000 S+ pts/1 0:00 tee -a /home/asdf/logfile\_3.txt

1000 20788 0000000000000000 0000000000000000 0000000000000000 0000000000000000 S+ pts/1 0:00 son1.out

1000 20789 0000000000000000 0000000000000000 0000000000000800 0000000000000000 S+ pts/1 0:00 son2.out

1000 20790 0000000000000000 0000000000000000 0000000000000000 0000000000000800 R+ pts/1 0:00 son3.out

1000 20791 0000000000000000 0000000000000000 0000000180000000 0000000000010002 S+ pts/1 0:00 sh -c ps -s>>output.txt

1000 20792 0000000000000000 0000000000000000 0000000000000000 00000001f3d1fef9 R+ pts/1 0:00 ps -s

UID PID PENDING BLOCKED IGNORED CAUGHT STAT TTY TIME COMMAND

1000 17426 0000000000000000 0000000000010000 0000000000380004 000000004b817efb Ss pts/1 0:00 bash

1000 17434 0000000000000000 0000000000010000 0000000000000004 0000000008010002 S+ pts/1 0:01 bash

1000 20786 0000000000000000 0000000000010000 0000000000000006 0000000000000000 S+ pts/1 0:00 ./father.out

1000 20787 0000000000000000 0000000000000000 0000000000000000 0000000000000000 S+ pts/1 0:00 tee -a /home/asdf/logfile\_3.txt

1000 20788 0000000000000800 0000000000000000 0000000000000000 0000000000000000 Z+ pts/1 0:00 [son1.out] <defunct>

1000 20789 0000000000000000 0000000000000000 0000000000000800 0000000000000000 Z+ pts/1 0:00 [son2.out] <defunct>

1000 20790 0000000000000000 0000000000000000 0000000000000000 0000000000000800 R+ pts/1 0:02 son3.out

1000 20793 0000000000000000 0000000000000000 0000000180000000 0000000000010002 S+ pts/1 0:00 sh -c ps -s>>output.txt

1000 20794 0000000000000000 0000000000000000 0000000000000000 00000001f3d1fef9 R+ pts/1 0:00 ps -s

UID PID PENDING BLOCKED IGNORED CAUGHT STAT TTY TIME COMMAND

1000 17426 0000000000000000 0000000000010000 0000000000380004 000000004b817efb Ss pts/1 0:00 bash

1000 20797 0000000000000000 0000000000010000 0000000000000006 0000000000000000 S+ pts/1 0:00 ./father.out

1000 20798 0000000000000000 0000000000000000 0000000000000000 0000000000000000 S+ pts/1 0:00 son1.out

1000 20799 0000000000000000 0000000000000000 0000000000000800 0000000000000000 S+ pts/1 0:00 son2.out

1000 20800 0000000000000000 0000000000000000 0000000000000000 0000000000000800 R+ pts/1 0:00 son3.out

1000 20801 0000000000000000 0000000000000000 0000000180000000 0000000000010002 S+ pts/1 0:00 sh -c ps -s>>output.txt

1000 20802 0000000000000000 0000000000000000 0000000000000000 00000001f3d1fef9 R+ pts/1 0:00 ps -s

UID PID PENDING BLOCKED IGNORED CAUGHT STAT TTY TIME COMMAND

1000 17426 0000000000000000 0000000000010000 0000000000380004 000000004b817efb Ss pts/1 0:00 bash

1000 20797 0000000000000000 0000000000010000 0000000000000006 0000000000000000 S+ pts/1 0:00 ./father.out

1000 20798 0000000000000800 0000000000000000 0000000000000000 0000000000000000 Z+ pts/1 0:00 [son1.out] <defunct>

1000 20799 0000000000000000 0000000000000000 0000000000000800 0000000000000000 Z+ pts/1 0:00 [son2.out] <defunct>

1000 20800 0000000000000000 0000000000000000 0000000000000000 0000000000000800 R+ pts/1 0:02 son3.out

1000 20803 0000000000000000 0000000000000000 0000000180000000 0000000000010002 S+ pts/1 0:00 sh -c ps -s>>output.txt

1000 20804 0000000000000000 0000000000000000 0000000000000000 00000001f3d1fef9 R+ pts/1 0:00 ps –s

**3.13.4. Организуйте посылку сигналов любым двум процессам, находящимся в разных состояниях: активном и пассивном, фиксируя моменты посылки и приема каждого сигнала с точностью до секунды. Приведите результаты в файле результатов.**

**Содержимое father.c:**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <wait.h>

#include <string.h>

#include <signal.h>

int main(int argc, char \*argv[])

{

int sid, pid, pid1, pid2, ppid, status;

pid = getpid();

ppid = getppid();

sid = getsid(pid);

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

if((pid1 = fork())==0) execl("son1.out","son1.out", argv[1], NULL);

if((pid2 = fork())==0) execl("son2.out","son2.out", NULL);

system("ps -s>>output.txt");

printf("Signal 1\n");

system("date");

kill(pid1,SIGUSR2);

sleep(2);

printf("Signal 2\n");

system("date");

kill(pid2,SIGUSR2);

system("ps -s>>output.txt");

wait(&status);

wait(&status);

}

**Содержимое son1.c:**

#include <stdio.h>

#include <signal.h>

void sign(int a){

printf("recive signal\n");

system("date");

}

void main()

{

int pid,ppid;

pid=getpid();

ppid=getppid();

printf(" SON\_1 PARAMS : pid=%i ppid=%i\nFather creates and waits \n", pid,

ppid);

signal(SIGUSR2, sign);

sleep(5);

}

**Содержимое son2.c:**

#include <stdio.h>

#include <signal.h>

void sign(int a){

printf("recive signal\n");

system("date");

}

void main()

{

int pid,ppid;

pid=getpid();

ppid=getppid();

printf(" SON\_2 PARAMS : pid=%i ppid=%i\nFather creates and waits \n", pid,

ppid);

signal(SIGUSR2, sign);

sleep(5);

}

**Результат выполнения программы :**

asdf@BigAwesomeTurtle:~/lab\_3/13\_3$ ./father.out  
FATHER PARAMS: sid = 17426 pid=20927 ppid=17426  
SON\_1 PARAMS : pid=20928 ppid=20927  
Father creates and waits  
SON\_2 PARAMS : pid=20929 ppid=20927  
Father creates and waits  
Signal 1  
Tue 26 Nov 2019 07:00:44 PM MSK  
recive signal  
Tue 26 Nov 2019 07:00:44 PM MSK  
Signal 2  
Tue 26 Nov 2019 07:00:46 PM MSK  
recive signal  
Tue 26 Nov 2019 07:00:46 PM MSK

**Содержимое файла output.txt:**

UID PID PENDING BLOCKED IGNORED CAUGHT STAT TTY TIME COMMAND

1000 17426 0000000000000000 0000000000010000 0000000000380004 00000001cb817efb Ss pts/1 0:00 bash

1000 20927 0000000000000000 0000000000010000 0000000000000006 0000000000000000 S+ pts/1 0:00 ./father.out

1000 20928 0000000000000000 0000000000000000 0000000000000000 0000000000000800 S+ pts/1 0:00 son1.out

1000 20929 0000000000000000 0000000000000000 0000000000000000 0000000000000800 S+ pts/1 0:00 son2.out

1000 20930 0000000000000000 0000000000000000 0000000180000000 0000000000010002 S+ pts/1 0:00 sh -c ps -s>>output.txt

1000 20931 0000000000000000 0000000000000000 0000000000000000 00000001f3d1fef9 R+ pts/1 0:00 ps -s

UID PID PENDING BLOCKED IGNORED CAUGHT STAT TTY TIME COMMAND

1000 17426 0000000000000000 0000000000010000 0000000000380004 00000001cb817efb Ss pts/1 0:00 bash

1000 20927 0000000000010000 0000000000010000 0000000000000006 0000000000000000 S+ pts/1 0:00 ./father.out

1000 20928 0000000000000000 0000000000000000 0000000000000000 0000000000000800 Z+ pts/1 0:00 [son1.out] <defunct>

1000 20929 0000000000000000 0000000000000000 0000000000000000 0000000000000800 Z+ pts/1 0:00 [son2.out] <defunct>

1000 20939 0000000000000000 0000000000000000 0000000180000000 0000000000010002 S+ pts/1 0:00 sh -c ps -s>>output.txt

1000 20941 0000000000000000 0000000000000000 0000000000000000 00000001f3d1fef9 R+ pts/1 0:00 ps -s

**3.14. Запустите в фоновом режиме несколько утилит, например: cat \*.c > myprog & lpr myprog & lpr intro& Воспользуйтесь командой jobs для анализа списка заданий и очередности их выполнения. Позаботьтесь об уведомлении о завершении одного из заданий с помощью команды notify. Аргументом команды является номер задания. Верните невыполненные задания в приоритетный режим командой fg. Например: fg %3 Отмените одно из невыполненных заданий.**

asdf@BigAwesomeTurtle:~/lab\_3/13\_3$ sleep 100 & sleep 110 & sleep 120 & sleep 130 & jobs -l  
[1] 21072 Running sleep 100 &  
[2] 21073 Running sleep 110 &  
[3]- 21074 Running sleep 120 &  
[4]+ 21075 Running sleep 130 &

asdf@BigAwesomeTurtle:~/lab\_3/13\_3$ kill 21072

asdf@BigAwesomeTurtle:~/lab\_3/13\_3$ jobs -l  
[1] 21072 Terminated sleep 100   
[2] 21073 Running sleep 110 &  
[3]- 21074 Running sleep 120 &  
[4]+ 21075 Running sleep 130 &  
  
asdf@BigAwesomeTurtle:~/lab\_3/13\_3$ fg %2  
sleep 110

[3] Done sleep 120

[4] Done sleep 130

**3.15. Ознакомьтесь с выполнением команды и системного вызова nice(1) и getpriority(2). Приведите примеры их использования в приложении. Определите границы приоритетов (создайте для этого программу). Есть ли разница в приоритетах для системных и пользовательских процессов, используются ли приоритеты реального времени? Каков пользовательский приоритет для запуска приложений из shell? Все ответы подкрепляйте экспериментально**

**Текст программы:**

#include <stdio.h>

#include <sys/time.h>

#include <sys/resource.h>

void main()

{

int pr, pid, i;

pid=getpid();

for (i = -100; i < 1; i++)

{

setpriority(PRIO\_PROCESS, pid, i);

pr = getpriority(PRIO\_PROCESS, pid);

if (pr != i) continue;

else

{

printf("Нижняя граница = %d\n", pr);

printf("Запросили %d, получили %d\n", i, pr);

break;

}

}

for (i = 1; i < 100; i++)

{

setpriority(PRIO\_PROCESS, pid, i);

pr = getpriority(PRIO\_PROCESS, pid);

if (pr == i) continue;

else

{

printf("Верхняя граница = %d\n", pr);

printf("Запросили %d, получили %d\n", i, pr);

break;

}

}

}

**Результат выполнения программы:**

asdf@BigAwesomeTurtle:~/lab\_3/15$ ./15.out  
Нижняя граница = 0  
Запросили 0, получили 0  
Верхняя граница = 19  
Запросили 20, получили 19  
asdf@BigAwesomeTurtle:~/lab\_3/15$ sudo ./15.out  
[sudo] password for asdf:  
Нижняя граница = -20  
Запросили -20, получили -20  
Верхняя граница = 19  
Запросили 20, получили 19

Мы так же можем поменять приоритет при запуске:

asdf@BigAwesomeTurtle:~/lab\_3/15$ echo 1 & echo 2

[1] 21169

2

asdf@BigAwesomeTurtle:~/lab\_3/15$ 1

asdf@BigAwesomeTurtle:~/lab\_3/15$ nice -n 1 echo 1 & nice -n 10 echo 2

[1] 21171

1

2

В первом случае сначала вывелось 2, а потом уже 1. Однако, если поставить у echo 1 приоритет больше (цифра меньше) чем у echo 2, то 1 выведется раньше.

**3.16. Ознакомьтесь с командой nohup(1). Запустите длительный процесс по nohup(1). Завершите сеанс работы. Снова войдите в систему и проверьте таблицу процессов. Поясните результат.**

**Текст программы:**

#include <stdio.h>

void main()

{

int i;

for(i = 0; i < 999999999999; i++);

}

Запустим программу с помощью nohup:

asdf@BigAwesomeTurtle:~/lab\_3/16$ nohup ./16 &  
[1] 1105  
asdf@BigAwesomeTurtle:~/lab\_3/16$ nohup: ignoring input and appending output to 'nohup.out'  
ps  
PID TTY TIME CMD  
1101 pts/0 00:00:00 bash  
1105 pts/0 00:00:03 16  
1106 pts/0 00:00:00 ps

После выхода и захода в систему найдем этот процесс:

asdf@BigAwesomeTurtle:~/lab\_3/16$ ps –xa

В выводе присутствует строчка:

1112 ? R 0:35 ./16

Что говорит нам о том, что nohup запускает команду с игнорированием сигнала потери связи.

**3.17. Определите uid процесса, каково минимальное значение и кому оно принадлежит. Каково минимальное и максимальное значение pid, каким процессам принадлежат? Проанализируйте множество системных процессов, как их отличить от прочих, перечислите назначение самых важных из них.**

asdf@BigAwesomeTurtle:~/lab\_3/16$ ps afo ruid,euid,pid,tty,stat,cmd  
RUID EUID PID TT STAT CMD  
1000 1000 1369 pts/0 Ss bash  
1000 1000 1578 pts/0 R+ \\_ ps afo ruid,euid,pid,tty,stat,cmd  
0 0 1135 tty7 Ssl+ /usr/lib/xorg/Xorg :0 -seat seat0 -auth /var/run  
0 0 481 tty1 Ss+ /sbin/agetty -o -p — \u —noclear tty1 linux

asdf@BigAwesomeTurtle:~/lab\_3/16$ sudo ps afo ruid,euid,pid,tty,stat,cmd  
[sudo] password for asdf:  
RUID EUID PID TT STAT CMD  
1000 1000 1369 pts/0 Ss bash  
0 0 1604 pts/0 S+ \\_ sudo ps afo ruid,euid,pid,tty,stat,cmd  
0 0 1605 pts/0 R+ \\_ ps afo ruid,euid,pid,tty,stat,cmd  
0 0 1135 tty7 Ssl+ /usr/lib/xorg/Xorg :0 -seat seat0 -auth /var/run  
0 0 481 tty1 Ss+ /sbin/agetty -o -p — \u —noclear tty1 linux

При запуске с правами суперпользователя UID становится равен 0.

Множество допустимых значений UID зависит от выбранной системы. В общем случае для UID допускается использование значений от 0 до 65535 и требуется соблюдение следующих правил: UID суперпользователя всегда равен нулю (UID = 0); пользователю nobody обычно присваивается или наибольший из возможных UID, или один из системных UID (nobody – имя пользователя, не являющегося владельцем ни одного файла, не состоящего ни в одной привилегированной группе и не имеющего никаких полномочий, кроме стандартных для обычных пользователей); UID в диапазоне от 1 до 100, как правило, резервируются на системные нужды.

В ОС Linux присвоение PID начинается с 0, такой идентификатор получает процесс ядра (kernel) при старте ОС. PID равный 1 всегда имеет init.

При создании каждого нового процесса ОС пытается присвоить ему следующий по возрастанию свободный номер. Если таких свободных номеров не остается (достигли максимума), то ОС выбирает минимальный из всех свободных. Освобождаются идентификаторы по мере завершения процессов. Максимально возможное значение для PID в Linux на базе процессоров Intel составляет 2^31-1.

asdf@BigAwesomeTurtle:~/lab\_3/16$ ps -eaf  
UID PID PPID C STIME TTY TIME CMD  
root 1 0 0 19:58 ? 00:00:05 /sbin/init  
root 2 0 0 19:58 ? 00:00:00 [kthreadd]  
root 3 2 0 19:58 ? 00:00:00 [rcu\_gp]  
root 4 2 0 19:58 ? 00:00:00 [rcu\_par\_gp]  
root 6 2 0 19:58 ? 00:00:00 [kworker/0:0H-kblockd]

…

…

asdf 1616 1369 0 20:18 pts/0 00:00:00 ps -eaf

**3.18. Подготовьте программу, формирующую несколько нитей. Нити для эксперимента могут быть практически идентичны. Например, каждая нить в цикле: выводит на печать собственное имя и инкрементирует переменную времени, после чего "засыпает" (sleep(5); sleep(1); - для первой и второй нитей соответственно), на экран (в файл) должно выводиться имя нити и количество пятисекундных (для первой) и секундных (для второй) интервалов функционирования каждой нити**

**Текст программы:**

#include <signal.h>

#include <pthread.h>

#include <stdio.h>

void\* thread1() {

int i,cnt = 0;

int tid,pid;

pid = getpid();

printf("Thread\_1 with thread pid = %d is started\n", pid);

for (i = 0; i<10; i++) {

cnt++;

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

sleep(5);

}

}

void\* thread2() {

int i,cnt = 0;

int tid,pid;

pid = getpid();

printf("Thread\_2 with thread pid = %d is started\n", tid, pid);

for (i = 0; i<10; i++) {

cnt++;

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);

}

**Результат выполнения программы:**

asdf@BigAwesomeTurtle:~/lab\_3/18$ ./18.out

Thread\_1 with thread pid = 1801 is started

thread1: 1

Thread\_2 with thread pid = 0 is started

thread2: 1

thread2: 2

thread2: 3

thread2: 4

thread2: 5

thread1: 2

thread2: 6

thread2: 7

thread2: 8

thread2: 9

thread2: 10

thread1: 3

thread1: 4

thread1: 5

thread1: 6

thread1: 7

thread1: 8

thread1: 9

thread1: 10

**3.19. После запуска программы проанализируйте выполнение нитей, распределение во времени. Используйте для этого вывод таблицы процессов командой ps -axhf Попробуйте удалить нить, зная ее идентификатор, командой kill. Приведите и объясните результат.**

asdf@BigAwesomeTurtle:~/lab\_3/18$ ./18.out&

asdf@BigAwesomeTurtle:~/lab\_3/18$ps –Lf

UID PID PPID LWP C NLWP SZ RSS PSR STIME TTY TIME CMD  
asdf 1895 1891 1895 0 1 2043 4784 3 20:49 pts/0 00:00:00 bash  
asdf 1908 1895 1908 0 3 4981 612 2 20:50 pts/0 00:00:00 ./18.  
asdf 1908 1895 1909 0 3 4981 612 3 20:50 pts/0 00:00:00 ./18.  
asdf 1908 1895 1910 0 3 4981 612 0 20:50 pts/0 00:00:00 ./18.  
asdf 1911 1895 1911 0 1 2114 3036 2 20:51 pts/0 00:00:00 ps –L

asdf@BigAwesomeTurtle:~/lab\_3/18$ kill -s 9 1908

asdf@BigAwesomeTurtle:~/lab\_3/18$ ps -LF  
UID PID PPID LWP C NLWP SZ RSS PSR STIME TTY TIME CMD  
asdf 1895 1891 1895 0 1 2043 4784 3 20:49 pts/0 00:00:00 bash  
asdf 1912 1895 1912 0 1 2114 2964 3 20:51 pts/0 00:00:00 ps -L

У главного процесса и его нитей одинаковый PID. При попытке удалить нить удаляется и процесс, и его нити.

**3.20. Модифицируйте программу так, чтобы управление второй нитью осуществлялось посредством сигнала SIGUSR1 из первой нити. На пятой секунде работы приложения удалите вторую нить. Для этого воспользуйтесь функцией pthread\_kill(t2, SIGUSR); (t2 - дескриптор второй нити). В остальном программу можно не изменять. Проанализируйте полученные результаты**

**Текст модифицированной программы:**

#include <signal.h>

#include <pthread.h>

#include <stdio.h>

#include <signal.h>

pthread\_t t1, t2;

void\* thread1() {

int i,cnt = 0;

int tid,pid;

pid = getpid();

printf("Thread\_1 with thread pid = %d is started\n", pid);

for (i = 0; i<10; i++) {

cnt++;

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

sleep(5);

if (cnt == 1) pthread\_kill(t2, SIGUSR2);

}

}

void\* thread2() {

int i,cnt = 0;

int pid;

pid = getpid();

printf("Thread\_2 with thread pid = %d is started\n", pid);

for (i = 0; i<10; i++) {

cnt++;

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);

}

**Результат выполнения программы:**

asdf@BigAwesomeTurtle:~/lab\_3/20$ ./20.out  
Thread\_1 with thread pid = 1937 is started  
thread1: 1  
Thread\_2 with thread pid = 1937 is started  
thread2: 1  
thread2: 2  
thread2: 3  
thread2: 4  
thread2: 5  
User defined signal 2

После однократного завершения первого sleep был послан сигнал SIGUSR2 из первой нити, что привело к удалению второй нити и завершению программы в целом.

**3.21. Последняя модификация предполагает создание собственного обработчика сигнала, содержащего уведомление о начале его работы и возврат посредством функции pthread\_exit(NULL); Сравните результаты, полученные после запуска этой модификации программы с результатами предыдущей.**

**Текст программы:**

#include <signal.h>

#include <pthread.h>

#include <stdio.h>

#include <signal.h>

pthread\_t t1, t2;

void\* thread1() {

int i,cnt = 0;

int tid,pid;

pid = getpid();

printf("Thread\_1 with thread pid = %d is started\n", pid);

for (i = 0; i<10; i++) {

cnt++;

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

sleep(5);

if (cnt == 1) pthread\_kill(t2, SIGUSR2);

}

}

void\* thread2() {

int i,cnt = 0;

int pid;

pid = getpid();

printf("Thread\_2 with thread pid = %d is started\n", pid);

for (i = 0; i<10; i++) {

cnt++;

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);

}

**Результат выполнения программы:**

asdf@BigAwesomeTurtle:~/lab\_3/21$ ./21.out  
Thread\_1 with thread pid = 1986 is started  
thread1: 1  
Thread\_2 with thread pid = 1986 is started  
thread2: 1  
thread2: 2  
thread2: 3  
thread2: 4  
thread2: 5  
thread1: 2  
thread2 is stopped by signal SIGSUR2 from thread1  
thread1: 3  
thread1: 4  
thread1: 5  
thread1: 6  
thread1: 7  
thread1: 8  
thread1: 9  
thread1: 10

Теперь при отправке сигнала первой нитью, вторая нить ловит его и запускает обработчик сигнала, который выполняет заданный код. В данном случае выводит информацию о том, что вторая нить была приостановлена и останавливает её. Программа в целом в данном случае уже не завершается.

**3.22. Перехватите сигнал «CTRL C» для процесса и потока однократно, а также многократно с восстановлением исходного обработчика после нескольких раз срабатывания. Проделайте аналогичную работу для переназначения другой комбинации клавиш.**

**Текст программы:**

#include <signal.h>

#include <pthread.h>

#include <stdio.h>

#include <signal.h>

pthread\_t t1, t2;

void sigHandler2(int sigNo) {

static int i = 0;

printf(" we catch i = %d times\n",i+1);

i++;

if (i == 3)

{

printf("now signal ^Z is default \n");

signal(SIGTSTP,SIG\_DFL);

}

}

void sigHandler(int sigNo) {

static int i = 0;

printf(" we catch i = %d times\n",i+1);

i++;

if (i == 6)

{

printf("now signal ^C is default \n");

signal(SIGINT,SIG\_DFL);

}

}

void\* thread1() {

int i,cnt = 0;

int tid,pid;

pid = getpid();

signal(SIGINT,sigHandler);

signal(SIGTSTP,sigHandler2);

printf("Thread\_1 with thread pid = %d is started\n", pid);

for (i = 0; i<10; i++) {

cnt++;

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

sleep(5);

}

}

void\* thread2() {

int i,cnt = 0;

int pid;

pid = getpid();

signal(SIGINT,sigHandler);

signal(SIGTSTP,sigHandler2);

printf("Thread\_2 with thread pid = %d is started\n", pid);

for (i = 0; i<10; i++) {

cnt++;

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);

signal(SIGINT,sigHandler);

signal(SIGTSTP,sigHandler2);

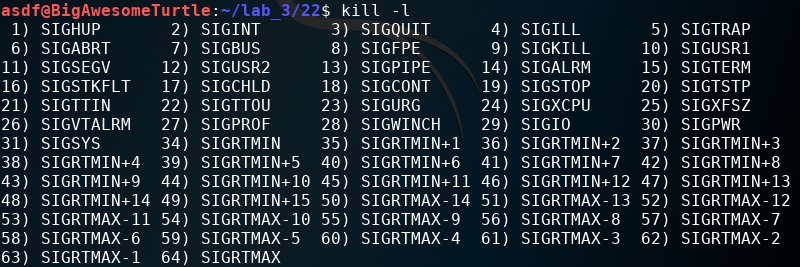
}

**Результат выполнения программы:**

asdf@BigAwesomeTurtle:~/lab\_3/22$ ./22.out  
Thread\_1 with thread pid = 2039 is started  
thread1: 1  
Thread\_2 with thread pid = 2039 is started  
thread2: 1  
thread2: 2  
thread2: 3  
^C we catch i = 1 times  
thread2: 4  
^C we catch i = 2 times  
^C we catch i = 3 times  
^C we catch i = 4 times  
thread2: 5  
^C we catch i = 5 times  
^C we catch i = 6 times  
now signal ^C is default  
thread1: 2  
thread2: 6  
^C

В данном случае мы перехватываем сигнал <CTRL+C> и при его получении инкрементируем счётчик и выводим сообщение с текущей величиной счётчика на экран. После 6 обработок такого сигнала мы перестаём его перехватывать и следующее использование этого сигнала останавливает программу.

**3.23. С помощью утилиты kill выведите список всех сигналов и дайте их краткую характеристику на основе документации ОС. Для чего предназначены сигналы с 32 по 64-й. Приведите пример их применения**





Сигналы с 32 по 64 - это сигналы реального времени. В отличии от обычных сигналов, они дают гарантию упорядоченной доставки сигналов – если поочерёдно послать несколько сигналов реального времени, то можно быть уверенным в том, что они в полном порядке дойдут до адресата и будут упорядочены – сигналы с меньшим номером придут первыми, перед сигналами с большими номерами. Также, вместе с сигналом реального времени можно передать целочисленное значение или указатель.

**3.24. Проанализируйте процедуру планирования для процессов и потоков одного процесса. 24.1. Обоснуйте результат экспериментально. 24.2. Попробуйте процедуру планирования изменить. Подтвердите экспериментально, если изменение возможно. 24.3. Задайте нитям разные приоритеты программно и извне (объясните результат).**

**Текст программы:**

#include <signal.h>

#include <pthread.h>

#include <stdio.h>

#include <sys/types.h>

#include <linux/unistd.h>

#include <sys/syscall.h>

#include <sched.h>

pthread\_t t1, t2, t3;

void\* thread1()

{

int i, count = 0;

int tid, pid;

tid = syscall(SYS\_gettid);

pid = getpid();

printf("Thread\_1 with thread id = %d and pid = %d is started\n", tid, pid);

for (i = 0; i < 10; i++)

{

printf("Thread\_1: %d\n", i);

}

}

void\* thread2()

{

int i, count = 0;

int tid, pid;

tid = syscall(SYS\_gettid);

pid = getpid();

printf("Thread\_2 with thread id = %d and pid = %d is started\n", tid, pid);

for (i = 0; i < 10; i++)

{

printf("Thread\_2: %d\n", i);

}

}

void\* thread3()

{

int i, count = 0;

int tid, pid;

tid = syscall(SYS\_gettid);

pid = getpid();

printf("Thread\_3 with thread id = %d and pid = %d is started\n", tid, pid);

for (i = 0; i < 10; i++)

{

printf("Thread\_3: %d\n", i);

}

}

void main()

{

int policy;

struct sched\_param param;

pthread\_attr\_t attr\_1, attr\_2, attr\_3;

pthread\_attr\_init(&attr\_1);

pthread\_attr\_init(&attr\_2);

pthread\_attr\_init(&attr\_3);

pthread\_attr\_setschedpolicy(&attr\_1, SCHED\_OTHER);

pthread\_attr\_setschedpolicy(&attr\_2, SCHED\_RR);

pthread\_attr\_setschedpolicy(&attr\_3, SCHED\_FIFO);

//значения приоритетов лучше задавать извне – из командной строки или файла

param.sched\_priority = 10;

pthread\_attr\_setschedparam(&attr\_1, &param);

param.sched\_priority = 95;

pthread\_attr\_setschedparam(&attr\_2, &param);

param.sched\_priority =99;

pthread\_attr\_setschedparam(&attr\_3, &param);

// Стратегия планирования и связанные с ней атрибуты должны быть взяты из описателя //атрибутов, на который указывает аргумент attr

pthread\_attr\_setinheritsched (&attr\_1,PTHREAD\_EXPLICIT\_SCHED);

pthread\_attr\_setinheritsched (&attr\_2,PTHREAD\_EXPLICIT\_SCHED);

pthread\_attr\_setinheritsched (&attr\_3,PTHREAD\_EXPLICIT\_SCHED);

pthread\_attr\_getschedparam(&attr\_1, &param);

pthread\_attr\_getschedpolicy(&attr\_1, &policy);

printf("Thread\_1's priority = %d\n", param.sched\_priority);

pthread\_attr\_getschedparam(&attr\_2, &param);

pthread\_attr\_getschedpolicy(&attr\_2, &policy);

printf("Thread\_2's priority = %d\n", param.sched\_priority);

pthread\_attr\_getschedparam(&attr\_3, &param);

pthread\_attr\_getschedpolicy(&attr\_3, &policy);

printf("Thread\_3's priority = %d\n", param.sched\_priority);

switch (policy) {

case SCHED\_FIFO:

printf ("policy SCHED\_FIFO\n");

break;

case SCHED\_RR:

printf ("policy SCHED\_RR\n");

break;

case SCHED\_OTHER:

printf ("policy SCHED\_OTHER\n");

break;

case -1:

perror ("policy SCHED\_GETSCHEDULER");

break;

default:

printf ("policy Неизвестная политика планирования\n");

}

pthread\_create(&t1, &attr\_1, thread1, NULL);

pthread\_create(&t2, &attr\_2, thread2, NULL);

pthread\_create(&t3, &attr\_3, thread3, NULL);

pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

pthread\_join(t3, NULL);

pthread\_attr\_destroy(&attr\_1);

pthread\_attr\_destroy(&attr\_2);

pthread\_attr\_destroy(&attr\_3);

}

**Результат выполнения программы:**

asdf@BigAwesomeTurtle:~/lab\_3/24$ sudo ./24.out  
[sudo] password for asdf:  
Thread\_1's priority = 0  
Thread\_2's priority = 95  
Thread\_3's priority = 99  
policy SCHED\_FIFO  
Thread\_2 with thread id = 1171 and pid = 1169 is started  
Thread\_2: 0  
Thread\_2: 1  
Thread\_2: 2  
Thread\_2: 3  
Thread\_2: 4  
Thread\_2: 5  
Thread\_2: 6  
Thread\_2: 7  
Thread\_2: 8  
Thread\_2: 9  
Thread\_3 with thread id = 1172 and pid = 1169 is started  
Thread\_3: 0  
Thread\_3: 1  
Thread\_3: 2  
Thread\_3: 3  
Thread\_3: 4  
Thread\_3: 5  
Thread\_3: 6  
Thread\_3: 7  
Thread\_3: 8  
Thread\_3: 9

Thread\_1 with thread id = 1170 and pid = 1169 is started  
Thread\_1: 0  
Thread\_1: 1  
Thread\_1: 2  
Thread\_1: 3  
Thread\_1: 4  
Thread\_1: 5  
Thread\_1: 6  
Thread\_1: 7  
Thread\_1: 8  
Thread\_1: 9

Потоки выполнились в порядке 2-3-1. Thread\_2 выполнился первым, т.к. у него приоритет выше, чем у Thread\_3. Последним выполнился поток с процедурой планирования SCHED\_OTHER.

**4. Вывод**

В ходе выполнения данной лабораторной работы было осуществлено знакомство с процессами в unix, рассмотрены различные политики планирования распределения процессорного времени между разными процессами, а также рассмотрены потоки и сигналы.

**Дополнение  
  
3.10.4**

Изменим процедуры планирования так, чтобы все процессы имели процедуру планирования SCHED\_FIFO.

**Результат выполнения программы:**

asdf@BigAwesomeTurtle:~/lab\_3/10/10\_reworked$ ./father.out

FATHER PARAMS: pid=1694 ppid=1693

SON PARAMS: pid=1696 ppid=1694  
son, called from second fork is doing its calculations

SON PARAMS: pid=1698 ppid=1  
son, called from fork in second son is doing its calculations

SON PARAMS: pid=1695 ppid=1694  
son, called from first fork is doing its calculations  
  
SON PARAMS: pid=1699 ppid=1  
son, called from fork in first son is doing its calculations  
  
father process is doing its calculations

Порядок выполнения процессов:

son2->son4->son1->son3->father

Сначала выполняются процессы, вызванные последними (вместе с порожденными ими процессами), а затем процессы, вызванные ранее, также в порядке, обратном порядку их вызова.

При изменении процедур планирования всех процессов на SCHED\_RR результаты не изменились.

**3.23.**

**Текст программы:**

#include <pthread.h>   
#include <signal.h>   
#include <stdio.h>   
#include <sys/types.h>   
#include <linux/unistd.h>   
#include <sys/syscall.h>   
pthread\_t t1, t2;   
void hnd (int sig) {   
if(sig == SIGUSR1) {   
printf("Catched SIGUSR1 %d\n",sig);   
signal(sig,hnd); //восстановление диспозиции обработчиком   
return;   
}   
if(sig == SIGUSR2) {   
printf("Catched SIGUSR2 %d\n",sig);   
signal(sig, hnd);   
return;   
}   
if(sig == SIGRTMIN+1) {   
printf("Catched SIGRTMIN+1 %d\n",sig);   
signal(sig, hnd);   
return;   
}   
if(sig == SIGRTMIN+2) {   
printf("Catched SIGRTMIN+2 %d\n",sig);   
signal(sig, hnd);   
return;   
}   
if(sig == SIGRTMIN+3) {   
printf("Catched SIGRTMIN+3 %d\n",sig);   
signal(sig, hnd);   
return;   
}   
if(sig == SIGRTMIN+4) {   
printf("Catched SIGRTMIN+4 %d\n",sig);   
signal(sig, hnd);   
return;   
}   
if(sig == SIGRTMIN+5) {   
printf("Catched SIGRTMIN+5 %d\n",sig);   
signal(sig, hnd);   
return;   
}  
}   
void\* thread1() {   
int i, count = 0;   
int tid, pid;   
tid = syscall(SYS\_gettid);   
pid = getpid();   
printf("Thread\_1 with thread id = %d and pid = %d is started\n", tid, pid);   
int n=1; //можно установить любым   
for (i = 0; i < n; i++) {   
count += 1;   
printf("Thread\_1: step %d \n", count);   
sleep(5);   
// отправка обычных сигналов из первой нити во вторую   
pthread\_kill(t2, SIGUSR1);   
pthread\_kill(t2, SIGUSR2);   
pthread\_kill(t2, SIGUSR1);   
pthread\_kill(t2, SIGRTMIN+3);// отправка сигналов реального времени   
pthread\_kill(t2, SIGRTMIN+1);   
pthread\_kill(t2, SIGRTMIN+5);   
pthread\_kill(t2, SIGRTMIN+1);   
}   
}  
 void\* thread2() {   
int i, count = 0;   
int tid, pid;   
tid = syscall(SYS\_gettid);   
pid = getpid();   
printf("Thread\_2 with thread id = %d and pid = %d is started\n", tid, pid);   
int m=10;   
for (i = 0; i < m; i++) {   
signal(SIGUSR1,hnd);   
signal(SIGUSR2,hnd);   
signal(SIGRTMIN+1,hnd);   
signal(SIGRTMIN+3,hnd);   
signal(SIGRTMIN+5,hnd);   
count += 1;   
sleep(1);   
printf("Thread\_2: step %d\n", count);   
}   
}   
void main() {   
pthread\_create(&t1, NULL, thread1, NULL);   
pthread\_create(&t2, NULL, thread2, NULL);   
pthread\_join(t1, NULL);   
pthread\_join(t2, NULL);   
}

**Результат выполнения программы:**

asdf@BigAwesomeTurtle:~/lab\_3/23$ ./23.out

Thread\_2 with thread id = 1120 and pid = 1118 is started  
Thread\_1 with thread id = 1119 and pid = 1118 is started  
Thread\_1: step 1  
Thread\_2: step 1  
Thread\_2: step 2  
Thread\_2: step 3  
Thread\_2: step 4  
Catched SIGRTMIN+5 39  
Catched SIGRTMIN+3 37  
Catched SIGRTMIN+1 35  
Catched SIGRTMIN+1 35  
Catched SIGCHLD 17  
Catched SIGUSR2 12  
Catched SIGUSR1 10  
Catched SIGINT 2  
Thread\_2: step 5  
Thread\_2: step 6  
Thread\_2: step 7  
Thread\_2: step 8  
Thread\_2: step 9  
Thread\_2: step 10

При одновременной отправке нескольких сигналов, они приходят в порядке приоритетов – сначала сигналы реального времени (от большего порядкового номера к меньшему), затем обычные сигналы (также от большего порядкового номера в таблице к меньшему).

**3.24.**

Установим одинаковые процедуры планирования(SCHED\_RR) и одинаковые приоритеты для всех потоков.

**Результат выполнения программы:**

asdf@BigAwesomeTurtle:~/lab\_3/24\_reworked$ sudo ./24.out  
Thread\_1's priority = 95  
Thread\_2's priority = 95  
Thread\_3's priority = 95  
policy SCHED\_RR  
Thread\_1 with thread id = 1274 and pid = 1273 is started  
Thread\_1: 0  
Thread\_1: 1  
Thread\_1: 2  
Thread\_1: 3  
Thread\_1: 4  
Thread\_1: 5  
Thread\_1: 6  
Thread\_1: 7  
Thread\_1: 8  
Thread\_1: 9  
Thread\_2 with thread id = 1275 and pid = 1273 is started  
Thread\_2: 0  
Thread\_2: 1  
Thread\_2: 2  
Thread\_2: 3

Thread\_2: 4  
Thread\_2: 5  
Thread\_2: 6  
Thread\_2: 7  
Thread\_2: 8  
Thread\_2: 9  
Thread\_3 with thread id = 1276 and pid = 1273 is started  
Thread\_3: 0  
Thread\_3: 1  
Thread\_3: 2  
Thread\_3: 3  
Thread\_3: 4  
Thread\_3: 5  
Thread\_3: 6  
Thread\_3: 7  
Thread\_3: 8  
Thread\_3: 9

Потоки выполнились в порядке их запуска.