**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра МОЭВМ**

**ОТЧЕТ**

**по лабораторной работе №3**

**по дисциплине «Операционные системы»**

**Тема: «Системное программирование в ОС семейства Unix»**

| Студент гр. 1303 |  | Чубан Д.В. |
| --- | --- | --- |
| Преподаватель |  | Душутина Е.В. |

Санкт-Петербург

2023

**Цель работы.**

Проанализировать функциональное назначение структурных элементов дерева ФС. Определить размещение корневого каталога (корневой ФС).

**Задание.**

Используя системные функции fork();семейства exec...(); wait(); exit(); sleep(); выполните следующее :

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.

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.

**Ход работы.**

**Задание 1.** На языке С++ была написана программа, порождающая процесс с использованием fork(). Программный код представлен в приложении А и ниже:

// Создайте программу на основе одного исходного (а затем исполняемого)

// файла с псевдораспараллеливанием вычислений посредством порождения

// процесса-потомка.

#include <iostream>

#include <unistd.h>

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

{

int pid;

int n = 0;

int m = 100;

pid = fork();

if(pid == -1)

{

perror("fork error");

exit(1);

}

std::cout << "pid: " << pid << '\n';

if(pid != 0)

{

for (int i = 0; i < 100; i++)

{

n += 1;

}

std::cout << "Parent: " << n << '\n';

}

else

{

for (int j = 0; j < 100; j++)

{

m -= 1;

}

std::cout << "Son: " << m << '\n';

}

std::cout << "Program done!" << '\n';

exit(1);

}

**Задание 2**. Для задания была написана программа, выводящая значение индекса в цикле для обоих процессов. В конце программы выводится сообщение о том, что текущий процесс закончил свою работу.

#include <iostream>

#include <unistd.h>

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

{

int pid;

pid = fork();

if(pid == -1)

{

perror("fork error");

exit(1);

}

std::cout << "pid: " << pid << '\n';

if(pid != 0)

{

for (int i = 0; i < 100; i++)

{

std::cout << "step = " << i << ",parent cycle with pid: " << pid << '\n';

}

}

else

{

for (int i = 0; i < 100; i++)

{

std::cout << "step = " << i << ",son cycle with pid: " << pid << '\n';

}

}

std::cout << "Program done from pid: " << pid << '\n';

exit(1);

}

Лог результата программы после запуска выглядит следующим образом (приведена его часть):

defrozen@defrozen-ubu:~/OSstuff/lb3/task2$ g++ task2.cpp -o task2  
defrozen@defrozen-ubu:~/OSstuff/lb3/task2$ ./task2   
pid: 10828  
step = 0,parent cycle with pid: 10828  
step = 1,parent cycle with pid: 10828  
step = 2,parent cycle with pid: 10828  
step = 3,parent cycle with pid: 10828  
step = 4,parent cycle with pid: 10828  
step = 5,parent cycle with pid: 10828  
step = 6,parent cycle with pid: 10828  
step = 7,parent cycle with pid: 10828

. . .  
step = 39,parent cycle with pid: 10828  
step = 40,parent cycle with pid: 10828  
pid: 0  
step = 41,parent cycle with pid: 10828  
step = 0,son cycle with pid: 0  
step = 42,parent cycle with pid: 10828  
step = 1,son cycle with pid: 0  
step = 43,parent cycle with pid: 10828  
step = 2,son cycle with pid: 0  
step = 44,parent cycle with pid: 10828  
step = 3,son cycle with pid: 0  
step = 45,parent cycle with pid: 10828  
step = 4,son cycle with pid: 0  
step = 46,parent cycle with pid: 10828  
step = 5,son cycle with pid: 0

. . .  
step = 98,parent cycle with pid: 10828  
step = 57,son cycle with pid: 0  
step = 99,parent cycle with pid: 10828  
step = 58,son cycle with pid: 0  
Program done from pid: 10828  
step = 59,son cycle with pid: 0  
step = 60,son cycle with pid: 0  
step = 61,son cycle with pid: 0  
. . .  
step = 98,son cycle with pid: 0  
step = 99,son cycle with pid: 0  
Program done from pid: 0  
defrozen@defrozen-ubu:~/OSstuff/lb3/task2$

По результатам работы программы можно сделать вывод о том, что процесс-потомок использует адресное пространство, выделенное для процесса-родителя. Это видно из того, что в самом начале работает основной процесс, затем, когда часть выделенного под него адресного пространства освобождается, начинает работать процесс-потомок. Процесс-родитель заканчивает работу - процесс-потомок продолжает работу в рамках того же пространства.

**Задание 3**. Для исследования борьбы процессов за ресурсы системы, предыдущая программа была немного видоизменена с использованием цикла while:

#include <iostream>

#include <unistd.h>

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

{

int pid;

int n = 0;

int m = 100;

int counter = 0;

pid = fork();

if(pid == -1)

{

perror("fork error");

exit(1);

}

while(true)

{

std::cout << "pid: " << pid << '\n';

if(pid != 0)

{

for (int i = 0; i < 100; i++)

{

n += 1;

}

std::cout << "Parent: " << n << '\n';

}

else

{

for (int j = 0; j < 100; j++)

{

m -= 1;

}

std::cout << "Son: " << m << '\n';

}

counter++;

if (counter == 50){

break;

}

}

std::cout << "Program done!" << '\n';

exit(1);

}

**Задание 4**. Изменив программу, запустим ее и проанализируем логи (ниже приведена часть лога):

defrozen@defrozen-ubu:~/OSstuff/lb3/task3$ g++ task3.cpp -o task3  
defrozen@defrozen-ubu:~/OSstuff/lb3/task3$ ./task3   
pid: 11768  
Parent: 100  
pid: 11768  
Parent: 200  
pid: 11768  
Parent: 300  
. . .  
Parent: 2300  
pid: 11768  
Parent: 2400  
pid: 11768  
pid: 0  
Parent: 2500  
pid: 11768  
Son: 0  
Parent: 2600  
pid: 0  
pid: 11768  
Son: -100  
. . .  
pid: 11768  
Son: -3300  
Parent: 4900  
pid: 0  
pid: 11768  
Son: -3400  
Parent: 5000  
pid: 0  
Program done!  
Son: -3500  
pid: 0

. . .  
pid: 0  
Son: -4900  
Program done!  
defrozen@defrozen-ubu:~/OSstuff/lb3/task3$

Исходя из лога, снова убеждаемся в том, что процесс-потомок использует адресное пространство процесса-родителя, и начинает работать тогда, когда часть выделенного адресного пространства освобождается.

**Задание 5.** Разработаны программные модули father.c и son.c. В первом файле создается процесс-потомок командой fork(), затем, с помощью функции execl(“son”,”son”,NULL) запускается второй файл, в котором вызывается команда sleep() для процесса-потомка. Состояние таблицы процессоров фиксируется в файле log.txt.

father.cpp

#include <iostream>

#include <wait.h>

#include <unistd.h>

int main()

{

int pid, ppid, status;

pid = getpid();

ppid = getppid();

std::cout << "FATHER PARAM: pid=" << pid << " ppid=" << ppid << "\n";

if (fork() == 0){

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

}

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

wait(&status);

std::cout << "Child process is finished with status: " << status << '\n';

return 0;

}

son.cpp

#include <iostream>

#include <unistd.h>

int main()

{

int pid, ppid;

pid = getpid();

ppid = getppid();

std::cout << "SON PARAMS: pid=" << pid << " ppid=" << ppid << '\n';

sleep(15);

return 0;

}

**Задание 6**. Был скомпилирован father.cpp и запущен файл исполняемый файл как ./father. Информация о процессах, запущенных с терминала, выглядит следующим образом:

defrozen@defrozen-ubu:~/OSstuff/lb3/task5$ g++ father.cpp -o father  
defrozen@defrozen-ubu:~/OSstuff/lb3/task5$ g++ son.cpp -o son  
defrozen@defrozen-ubu:~/OSstuff/lb3/task5$ ./father   
FATHER PARAM: pid=14083 ppid=9250  
SON PARAMS: pid=14084 ppid=14083  
Child process is finished with status: 0  
defrozen@defrozen-ubu:~/OSstuff/lb3/task5$ cat log.txt

9250 pts/1 Ss 0:00 \\_ bash  
14083 pts/1 S+ 0:00 \\_ ./father  
14084 pts/1 S+ 0:00 \\_ son  
14085 pts/1 S+ 0:00 \\_ sh -c ps xf > log.txt  
14086 pts/1 R+ 0:00 \\_ ps xf

Как видно из таблицы и вывода программы, были созданы потомки исходного процесса ./father - выполнение команды sleep() в son а также запись лога таблицы в файл. Информация в таблице следующая: PID - идентификатор процесса, TTY - терминал, с которым связан данный процесс, STAT - состояние, в котором находится процесс-родитель (S - процесс ожидает, s - лидер сессии, R - процесс выполняется в данный момент, + - порожденный процесс), TIME - процессорное время, занятие этим процессом.

**Задание 7**. Исполняемый файл ./father был запущен в фоновом режиме командой ./father &. Таблица процессов в логе выглядела следующим образом:

PID TTY STAT TIME COMMAND

9250 pts/1 Ss 0:00 \\_ bash  
14430 pts/1 S 0:00 \\_ ./father  
14431 pts/1 S 0:00 | \\_ son  
14435 pts/1 R+ 0:00 \\_ ps -xf  
defrozen@defrozen-ubu:~/OSstuff/lb3/task5$

Видно, что командный интерпретатор распараллелил процессы ps -xf и ./father, в свою очередь ./father породил процесс son. Результат работы выглядит следующим образом:

9250 pts/1 Ss 0:00 \\_ bash  
14437 pts/1 R+ 0:00 \\_ ps -xf  
[1]+ Done ./father  
defrozen@defrozen-ubu:~/OSstuff/lb3/task5$

**Задание 8**. Были написаны три программы с использованием различных функций семейства exec(). Их код и логи представлены ниже:

Функция execv():

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

/\* Синтаксис:

https://en.m.wikipedia.org/wiki/Exec\_(system\_call)#C\_language\_prototypes

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

path: указывает на путь к заменяемому процессу

argv: список аргументов

\*/

int main()

{

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

int pid = fork();

if (pid == 0)

{

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

}

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

printf("End from execv.c\n");

return 0;

}

Лог программы:

defrozen@defrozen-ubu:~/OSstuff/lb3/task8/execv$ gcc execv.c -o execv  
defrozen@defrozen-ubu:~/OSstuff/lb3/task8/execv$ ./execv  
total 32K  
drwxrwxr-x 2 defrozen defrozen 4,0K апр 11 14:25 .  
drwxrwxr-x 4 defrozen defrozen 4,0K апр 11 14:13 ..  
-rwxrwxr-x 1 defrozen defrozen 17K апр 11 14:25 execv  
-rw-rw-r— 1 defrozen defrozen 526 апр 11 14:25 execv.c  
-rw-rw-r— 1 defrozen defrozen 0 апр 11 14:25 log.txt  
End from execv.c

Функция execle():

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

/\* Синтаксис:

https://en.m.wikipedia.org/wiki/Exec\_(system\_call)#C\_language\_prototypes

int execle(char const \*path, char const \*arg0, ..., char const \*envp[]);

path: указывает на путь к исполняемому процессу

argv: список аргументов

envp: это массив указателей на настройки окружения. Вызовы exec с именем, оканчивающимся на e, изменяют среду для нового образа процесса, передавая список параметров среды через аргумент envp.

\*/

int main()

{

//find / -name "ros" 2>/dev/null

char\* env[] = { NULL };

int pid = fork();

if (pid == 0)

{

execle("/bin/find", "find", "/home", "-name", "\*.py", NULL, env);

}

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

printf("End from execle.c\n");

return 0;

}

Лог execle():

defrozen@defrozen-ubu:~/OSstuff/lb3/task8/execle$ gcc execle.c -o execle  
defrozen@defrozen-ubu:~/OSstuff/lb3/task8/execle$ ./execle  
/home/defrozen/Downloads/Telegram Desktop/[virtual\_joystick\_gui.py](https://vk.com/away.php?to=http://virtual_joystick_gui.py&cc_key=)  
/home/defrozen/Downloads/Telegram Desktop/virtual\_joystick\_gui (2).py  
/home/defrozen/Downloads/Telegram Desktop/virtual\_joystick\_gui (3).py  
/home/defrozen/Downloads/[run\_file.py](https://vk.com/away.php?to=http://run_file.py&cc_key=)  
/home/defrozen/Downloads/bot\_camera(2).py  
/home/defrozen/Downloads/[bot\_camera.py](https://vk.com/away.php?to=http://bot_camera.py&cc_key=)  
/home/defrozen/Downloads/bot\_camera(1).py

. . .

Функция execvp():

#include <unistd.h>

#include <stdio.h>

/\* Синтаксис:

https://en.m.wikipedia.org/wiki/Exec\_(system\_call)#C\_language\_prototypes

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

файл: указывает на имя файла, связанное с исполняемым файлом.

argv: массив указателей на символы с нулевым завершением.

\*/

int main()

{

char\* args[] = {"./exec1", NULL};

execvp(args[0], args);

printf("End from execvp.c\n");

return 0;

}

Функция к запуску функцией execvp():

#include <unistd.h>

#include <iostream>

int main()

{

std::cout << "I is replaced process exec1.cpp called by execvp()" << std::endl;

return 0;

}

Лог:

defrozen@defrozen-ubu:~/OSstuff/lb3/task8/execvp$ ./execvp   
I is replaced process exec1.cpp called by execvp()  
defrozen@defrozen-ubu:~/OSstuff/lb3/task8/execvp$

**Задание 9**. Для того, чтобы проанализировать значения wait(&status), была написана программа, отслеживающая состояние окончания порожденного процесса (father.c) и подпроцессы, которые возвращают различные коды окончания работы (sonN.c):

Father.c

#include <stdio.h>

#include <sys/types.h>

#include <wait.h>

#include <unistd.h>

#include <stdlib.h>

int main()

{

int i, pid[6], ppid, status, result;

pid[0]=getpid();

ppid=getppid();

printf("\nFather parameters was: pid=%i ppid=%i\n", pid[0],ppid);

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

{

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

}

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

{

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

}

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

{

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

}

if((pid[4] = fork()) == 0)

{

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

}

if((pid[5] = fork()) == 0)

{

execl("son5", "son5", NULL);

}

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

for (i = 1; i < 6; i++)

{

// WUNTRACED - сообщает статус остановленного процесса-потомка

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;

}

SonN.c:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main()

{

int pid, ppid;

pid = getpid();

ppid = getppid();

printf("\n Son parameters was: pid=%i, ppid=%i\n", pid, ppid);

sleep(10);

// в son1: return 0

// в son2: exit(1)

// в son3: exit(-1)

// в son4: return 5

// в son5: exit(-10)

return 0;

}

Все файлы скомпилированы. Был запущен исполняемый файл ./father1:

defrozen@defrozen-ubu:~/OSstuff/lb3/task9$ gcc son1.c -o son1  
defrozen@defrozen-ubu:~/OSstuff/lb3/task9$ gcc son2.c -o son2  
defrozen@defrozen-ubu:~/OSstuff/lb3/task9$ gcc son3.c -o son3  
defrozen@defrozen-ubu:~/OSstuff/lb3/task9$ gcc son4.c -o son4  
defrozen@defrozen-ubu:~/OSstuff/lb3/task9$ gcc son5.c -o son5  
defrozen@defrozen-ubu:~/OSstuff/lb3/task9$ gcc father1.c -o father1  
defrozen@defrozen-ubu:~/OSstuff/lb3/task9$ ./father1   
  
Father parameters was: pid=21576 ppid=7795  
  
Son parameters was: pid=21577, ppid=21576  
  
Son parameters was: pid=21580, ppid=21576  
Son parameters was: pid=21578, ppid=21576  
  
Son parameters was: pid=21579, ppid=21576  
  
Son parameters was: pid=21581, ppid=21576  
  
1) Child proccess with pid = 21577 is finished with status 0  
  
2) Child proccess with pid = 21578 is finished with status 256  
  
3) Child proccess with pid = 21579 is finished with status 65280  
  
4) Child proccess with pid = 21580 is finished with status 1280  
  
5) Child proccess with pid = 21581 is finished with status 62976  
defrozen@defrozen-ubu:~/OSstuff/lb3/task9$

Статус завершения процесса представлен числом, которое может быть интерпретировано с помощью макросов WIFEXITED(), WIFSIGNALED() и WEXITSTATUS(). 0 - процесс завершился без ошибок, в данном утверждении можно убедиться, посмотрев на значение макроса WIFEXITED(0) - оно вернёт истину, что соответствует тому, что процесс-потомок успешно завершился без ошибок. Все остальные статусы не соответствуют нормальному завершению - их можно проверить с помощью макроса WEXITSTATUS. Например, WEXITSTATUS(256) значит, что код возврата был 1 (так как данный макрос возвращает первые восемь младших битов значения, которое вернул завершившийся потомок). -1 соответствует 255\*256 в таком случае, как и для -10 это 246\*256.

Для отслеживания подмножества порожденных процессов использовались функции wait(): WNOHANG и WUNTRACED:

Father2.c:

#include <stdio.h>

#include <sys/types.h>

#include <wait.h>

#include <string.h>

#include <unistd.h>

#include <stdlib.h>

int main()

{

int i, ppid, pid[6], status[5], result[5];

char \*son[] = {"son1", "son2", "son3", "son4", "son5"};

int option[] = {WNOHANG, WUNTRACED, WNOHANG, WNOHANG, WNOHANG};

pid[5] = getpid();

ppid = getppid();

printf("Father parameters was: pid=%i ppid=%i\n", pid[5],ppid);

for (i = 0; i < 5; i++)

{

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

{

execl(son[i], son[i], NULL);

}

}

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

for (i = 0; i < 5; i++)

{

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

printf("%d) Child with pid = %d is finished with status %d\n", (1+i), result[i], status[i]);

}

for(i = 0; i < 5; i++)

if (WIFEXITED(status[i]) == 0)

{

printf("Proccess pid = %d was failed.\n", pid[i]);

}

else

{

printf("Proccess pid = %d was success.\n", pid[i]);

}

return 0;

}

WNOHANG – означает немедленное возвращение управления, если ни  
один дочерний процесс не завершил выполнение.  
 WUNTRACED – означает возврат управления и для остановленных (но  
не отслеживаемых) дочерних процессов, о статусе которых еще не было  
сообщено. Статус для отслеживаемых остановленных подпроцессов также  
обеспечивается без этой опции.

WNOHANG, WNOHANG, WNOHANG, WNOHANG, WNOHANG  
defrozen@defrozen-ubu:~/OSstuff/lb3/task9$ gcc father2.c -o father2  
defrozen@defrozen-ubu:~/OSstuff/lb3/task9$ ./father2  
Father parameters was: pid=24602 ppid=7795  
  
Son parameters was: pid=24605, ppid=24602  
  
Son parameters was: pid=24604, ppid=24602  
  
Son parameters was: pid=24603, ppid=24602  
  
Son parameters was: pid=24606, ppid=24602  
  
Son parameters was: pid=24607, ppid=24602  
1) Child with pid = 0 is finished with status 0  
2) Child with pid = 0 is finished with status 64  
3) Child with pid = 0 is finished with status 512  
4) Child with pid = 0 is finished with status 1024  
5) Child with pid = 0 is finished with status 0  
Proccess pid = 24603 was success.  
Proccess pid = 24604 was failed.  
Proccess pid = 24605 was success.  
Proccess pid = 24606 was success.  
Proccess pid = 24607 was success.  
defrozen@defrozen-ubu:~/OSstuff/lb3/task9$ ^C

Так как нет потомков в ожидаемом состоянии, то waitid() сразу возвращает 0 (так как для всех процессов указан параметр WHOHANG()).

WUNTRACED, WUNTRACED, WUNTRACED, WUNTRACED, WUNTRACED

defrozen@defrozen-ubu:~/OSstuff/lb3/task9$ gcc father2.c -o father2

defrozen@defrozen-ubu:~/OSstuff/lb3/task9$ ./father2

Father parameters was: pid=25543 ppid=7795

Son parameters was: pid=25544, ppid=25543

Son parameters was: pid=25546, ppid=25543

Son parameters was: pid=25545, ppid=25543

Son parameters was: pid=25547, ppid=25543

Son parameters was: pid=25548, ppid=25543

1) Child with pid = 25544 is finished with status 0

2) Child with pid = 25545 is finished with status 256

3) Child with pid = 25546 is finished with status 65280

4) Child with pid = 25547 is finished with status 256

5) Child with pid = 25548 is finished with status 65280

Proccess pid = 25544 was success.

Proccess pid = 25545 was success.

Proccess pid = 25546 was success.

Proccess pid = 25547 was success.

Proccess pid = 25548 was success.

defrozen@defrozen-ubu:~/OSstuff/lb3/task9$

Так как pid > 0, то происходит ожидание завершения процесса с идентификатором pid.

WNOHANG, WUNTRACED, WNOHANG, WNOHANG, WNOHANG  
defrozen@defrozen-ubu:~/OSstuff/lb3/task9$ gcc father2.c -o father2  
defrozen@defrozen-ubu:~/OSstuff/lb3/task9$ ./father2  
Father parameters was: pid=25803 ppid=7795  
  
Son parameters was: pid=25805, ppid=25803  
Son parameters was: pid=25806, ppid=25803  
  
Son parameters was: pid=25804, ppid=25803  
  
Son parameters was: pid=25807, ppid=25803  
  
Son parameters was: pid=25808, ppid=25803  
1) Child with pid = 0 is finished with status 0  
2) Child with pid = 25805 is finished with status 256  
3) Child with pid = 0 is finished with status 512  
4) Child with pid = 25807 is finished with status 256  
5) Child with pid = 25808 is finished with status 65280  
Proccess pid = 25804 was success.  
Proccess pid = 25805 was success.  
Proccess pid = 25806 was success.  
Proccess pid = 25807 was success.  
Proccess pid = 25808 was success.  
defrozen@defrozen-ubu:~/OSstuff/lb3/task9$

**Задание 10**. Порядок выполнения процессов при использовании функций семейства exec() зависит от того, какой процесс запускается первым и какие процессы зависят от него, то есть кто первее указан - тот и запускается первее.

**Задание 10.1.** Когда вызывается fork(), создается новый дочерний процесс, который является идентичной копией родительского процесса. Затем дочерний процесс выполняет код, следующий за вызовом fork(), в то время как родительский процесс продолжает выполнять код, предшествующий вызову fork().  
Если процесс снова вызывает fork() внутри дочернего процесса, создается другой дочерний процесс и так далее.

Таким образом, порядок выполнения процессов, созданных вложенными вызовами fork(), следует древовидной структуре, где каждый родительский процесс создает дочерний процесс, и каждый дочерний процесс может создавать другой дочерний процесс.

Для того, чтобы это экспериментально подтвердить, были написаны 4 программных модуля: father.c, son1.c, son1\_1.c, son2.c. Родительский процесс порождает два процесса: son1 и son2, а son1 в свою очередь порождает son1\_1.c:

Father.c:

#include <stdio.h>

#include <sys/types.h>

#include <wait.h>

#include <unistd.h>

#include <stdlib.h>

#include <sched.h>

int main()

{

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

pid[0]=getpid();

ppid=getppid();

printf("\nFather parameters was: pid=%i ppid=%i\n", pid[0],ppid);

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

{

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

}

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

{

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

}

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

for (i = 1; i < 3; i++)

{

// WUNTRACED - сообщает статус остановленного процесса-потомка

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

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

}

}

Son1.c:

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

#include <wait.h>

int main()

{

int pid, ppid, status, result;

pid = getpid();

ppid = getppid();

printf("PROCESS son1 PARAM: pid=%i ppid=%i\n", pid, ppid);

if (fork() == 0){

status = execl("son1\_1", "son1\_1", NULL);

}

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

result = waitpid(pid, &status, WUNTRACED);

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

return 0;

}

Son1\_1.c (Son2.c аналогичен):

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

int main()

{

int pid, ppid;

pid = getpid();

ppid = getppid();

printf("PROCESS son1\_1 PARAM: pid=%i ppid=%i\n", pid, ppid);

sleep(5);

return 0;

}

Если father вызывает сначала son1, то лог исполняемого файла выглядит следующим образом:  
defrozen@defrozen-ubu:~/OSstuff/lb3/task10/task10\_1$ gcc father.c -o father  
defrozen@defrozen-ubu:~/OSstuff/lb3/task10/task10\_1$ ./father  
  
Father parameters was: pid=30682 ppid=28310  
PROCESS son2 PARAM: pid=30684 ppid=30682  
PROCESS son1 PARAM: pid=30683 ppid=30682  
PROCESS son1\_1 PARAM: pid=30686 ppid=30683  
Child proccess with pid = -1 is finished with status 370258240  
Child proccess with pid = 30683 is finished with status 0  
Child proccess with pid = 30684 is finished with status 0  
defrozen@defrozen-ubu:~/OSstuff/lb3/task10/task10\_1$

В ином случае, когда сначала вызывается son2.c, лог следующий:

defrozen@defrozen-ubu:~/OSstuff/lb3/task10/task10\_1$ ./father  
  
Father parameters was: pid=30825 ppid=28310  
PROCESS son1 PARAM: pid=30827 ppid=30825  
PROCESS son2 PARAM: pid=30826 ppid=30825  
PROCESS son1\_1 PARAM: pid=30829 ppid=30827  
Child proccess with pid = -1 is finished with status -1098460864  
Child proccess with pid = 30827 is finished with status 0  
Child proccess with pid = 30826 is finished with status 0  
defrozen@defrozen-ubu:~/OSstuff/lb3/task10/task10\_1$

**Задание 10.2.** Изменим процедуру планирования с помощью функции sched\_setscheduler() и повторим эксперимент:

Код father.c

#include <stdio.h>

#include <sys/types.h>

#include <wait.h>

#include <unistd.h>

#include <stdlib.h>

#include <sched.h>

int main(){

struct sched\_param shdprm; // Значения параметров планирования

int pid[3], ppid;

pid[0] = getpid();

ppid = getppid();

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

shdprm.sched\_priority = 50;

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

perror ("SCHED\_SETSCHEDULER");

}

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

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

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

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

printf ("Текущая политика планирования для текущего процесса: ");

switch (sched\_getscheduler (0)){

case SCHED\_FIFO:

printf ("SCHED\_FIFO\n");

break;

case SCHED\_RR:

printf ("SCHED\_RR\n");

break;

case SCHED\_OTHER:

printf ("SCHED\_OTHER\n");

break;

case -1:

perror ("SCHED\_GETSCHEDULER");

break;

default:

printf ("Неизвестная политика планирования\n");

}

if (sched\_getparam (0, &shdprm) == 0)

printf ("Текущий приоритет текущего процесса: %d\n", shdprm.sched\_priority);

else

perror ("SCHED\_GETPARAM");

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

for (int i = 1; i < 3; i++)

{

int status;

// WUNTRACED - сообщает статус остановленного процесса-потомка

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

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

}

return 0;

}

Код son1.c

#include <stdio.h>

#include <stdlib.h>

#include <sched.h>

#include <unistd.h>

int main()

{

struct sched\_param shdprm; // Значения параметров планирования

int i, pid,ppid;

pid=getpid();

ppid=getppid();

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

printf ("SON\_1: Текущая политика планирования для текущего процесса: ");

printf ("Текущая политика планирования для текущего процесса: ");

switch (sched\_getscheduler (0)) {

case SCHED\_FIFO:

printf ("SCHED\_FIFO\n");

break;

case SCHED\_RR:

printf ("SCHED\_RR\n");

break;

case SCHED\_OTHER:

printf ("SCHED\_OTHER\n");

break;

case -1:

perror ("SCHED\_GETSCHEDULER");

break;

default:

printf ("Неизвестная политика планирования\n");

}

if (sched\_getparam (0, &shdprm) == 0)

printf ("SON\_1: Текущий приоритет текущего процесса: %d\n", shdprm.sched\_priority);

else

perror ("SCHED\_GETPARAM");

return 0;

}

Код son2.c

#include <stdio.h>

#include <stdlib.h>

#include <sched.h>

#include <unistd.h>

int main()

{

struct sched\_param shdprm; // Значения параметров планирования

int i, pid,ppid;

pid=getpid();

ppid=getppid();

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

printf ("SON\_2: Текущая политика планирования для текущего процесса: ");

printf ("Текущая политика планирования для текущего процесса: ");

switch (sched\_getscheduler (0)) {

case SCHED\_FIFO:

printf ("SCHED\_FIFO\n");

break;

case SCHED\_RR:

printf ("SCHED\_RR\n");

break;

case SCHED\_OTHER:

printf ("SCHED\_OTHER\n");

break;

case -1:

perror ("SCHED\_GETSCHEDULER");

break;

default:

printf ("Неизвестная политика планирования\n");

}

if (sched\_getparam (0, &shdprm) == 0)

printf ("SON\_2: Текущий приоритет текущего процесса: %d\n", shdprm.sched\_priority);

else

perror ("SCHED\_GETPARAM");

return 0;

}

Вывод без прав суперпользователя выглядит следующим образом:

defrozen@defrozen-ubu:~/OSstuff/lb3/task10/task10\_2$ ./father  
FATHER PARAMS: pid=9121 ppid=6165  
SCHED\_SETSCHEDULER: Operation not permitted  
Текущая политика планирования для текущего процесса: SCHED\_OTHER  
Текущий приоритет текущего процесса: 0  
SON\_1 PARAMS: pid=9122 ppid=9121  
SON\_1: Текущая политика планирования для текущего процесса: SCHED\_OTHER  
SON\_1: Текущий приоритет текущего процесса: 0  
SON\_2 PARAMS: pid=9123 ppid=9121  
SON\_2: Текущая политика планирования для текущего процесса: SCHED\_OTHER  
SON\_2: Текущий приоритет текущего процесса: 0  
defrozen@defrozen-ubu:~/OSstuff/lb3/task10/task10\_2$

В ином случае (с правами суперпользователя):

defrozen@defrozen-ubu:~/OSstuff/lb3/task10/task10\_2$ sudo ./father  
[sudo] password for defrozen:   
FATHER PARAMS: pid=9129 ppid=9128  
Текущая политика планирования для текущего процесса: SCHED\_RR  
Текущий приоритет текущего процесса: 50  
SON\_1 PARAMS: pid=9130 ppid=9129  
SON\_1: Текущая политика планирования для текущего процесса: SCHED\_RR  
SON\_1: Текущий приоритет текущего процесса: 50  
SON\_2 PARAMS: pid=9131 ppid=9129  
SON\_2: Текущая политика планирования для текущего процесса: SCHED\_RR  
SON\_2: Текущий приоритет текущего процесса: 50  
defrozen@defrozen-ubu:~/OSstuff/lb3/task10/task10\_2$

Таким образом, можно утверждать, что потомки наследуют политику планирования и приоритет их родительских процессов.

**Задание 10.3.** Поменяем порядок очереди в RR-процедуре:

Father.c:

#include <stdio.h>

#include <sys/types.h>

#include <wait.h>

#include <unistd.h>

#include <stdlib.h>

#include <sched.h>

int main() {

struct sched\_param shdprm; // значения параметров планирования

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

int n, m, l, k;

n=50; m=60; l=70; k=80;

//

pid = getpid();

ppid = getppid();

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

shdprm.sched\_priority = n;

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

perror ("SCHED\_SETSCHEDULER");

}

if((pid1=fork()) == 0)

{

shdprm.sched\_priority = m;

if (sched\_setscheduler (pid1, SCHED\_RR, &shdprm) == -1)

perror ("SCHED\_SETSCHEDULER\_1");

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

}

if((pid2=fork()) == 0)

{

shdprm.sched\_priority = l;

if (sched\_setscheduler (pid2, SCHED\_RR, &shdprm) == -1)

perror ("SCHED\_SETSCHEDULER\_2");

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

}

if((pid3=fork()) == 0)

{

shdprm.sched\_priority = k;

if (sched\_setscheduler (pid3, SCHED\_RR, &shdprm) == -1)

perror ("SCHED\_SETSCHEDULER\_3");

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

}

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

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

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

return 0;

}

Код для SonN.c аналогичен коду в задании 10.2.

Алгоритм планирования RR (Round Robin) - это алгоритм планирования процессов, в котором каждый процесс имеет фиксированное время выполнения (квант времени). Процессы выполняются по очереди, и если процесс не успевает выполниться за отведённый ему квант времени, он ставится в конец очереди и ждёт своей очереди.

defrozen@defrozen-ubu:~/OSstuff/lb3/task10/task10\_3$ sudo ./father   
FATHER PARAMS: pid=9369 ppid=9368  
SON\_1 PARAMS: pid=9370 ppid=9369  
SON\_1 Текущая политика планирования для текущего процесса: SCHED\_RR  
SON\_1: Текущий приоритет текущего процесса: 60  
Процесс с pid = 9370 завершен  
SON\_3 PARAMS: pid=9372 ppid=9369  
SON\_3 Текущая политика планирования для текущего процесса: SCHED\_RR  
SON\_2 PARAMS: pid=9371 ppid=9369  
SON\_3: Текущий приоритет текущего процесса: 80  
SON\_2 Текущая политика планирования для текущего процесса: SCHED\_RR  
SON\_2: Текущий приоритет текущего процесса: 70  
Процесс с pid = 9371 завершен  
Процесс с pid = 9372 завершен  
defrozen@defrozen-ubu:~/OSstuff/lb3/task10/task10\_3$

Для вышеуказанного лога мы поменяли в очереди приоритеты для SON3 и SON2, хотя запускаются они в другом порядке - видим, что процесс SON3 не успевает выполниться за отведенное ему время, поэтому встал в конец очереди и завершился последним (pid=9372).

Теперь расставим всем процессам одинаковый приоритет. Тогда получим политику планирования FIFO:

Father.c:

#include <stdio.h>

#include <sys/types.h>

#include <wait.h>

#include <unistd.h>

#include <stdlib.h>

#include <sched.h>

int main ()

{

struct sched\_param shdprm;

int pid,ppid, status;

int n = 60; // переменные для задания значений приоритетов,

int prior = 10; //для повторного эксперимента с политикой FIFO

char \*arr[4] = {"son1", "son2", "son3", "son4"};

int pid\_son[4];

pid = getpid();

ppid = getppid();

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

shdprm.sched\_priority = n;

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

{

perror ("SCHED\_FIFO");

}

for (int i=3; i >= 0; i--)

{

if((pid\_son[i]=fork()) == 0)

{

shdprm.sched\_priority = prior;

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

perror ("SCHED\_FIFO");

execl(arr[i], arr[i], NULL);

}

}

for (int i=0; i < 4; i++)

{

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

}

return 0;

}

Код для SonN.c аналогичен коду в задании 10.2. Заметим, что мы поменяли порядок создания процессов - начиная с son4, заканчивая son1.  
defrozen@defrozen-ubu:~/OSstuff/lb3/task10/task10\_4$ sudo ./father  
FATHER PARAMS: pid=10002 ppid=10001  
SON\_4 PARAMS: pid=10003 ppid=10002  
SON\_4 Текущая политика планирования для текущего процесса: SCHED\_FIFO  
SON\_4: Текущий приоритет текущего процесса: 10  
SON\_3 PARAMS: pid=10004 ppid=10002  
SON\_3 Текущая политика планирования для текущего процесса: SCHED\_FIFO  
SON\_3: Текущий приоритет текущего процесса: 10  
SON\_2 PARAMS: pid=10005 ppid=10002  
SON\_2 Текущая политика планирования для текущего процесса: SCHED\_FIFO  
SON\_2: Текущий приоритет текущего процесса: 10  
SON\_1 PARAMS: pid=10006 ppid=10002  
SON\_1 Текущая политика планирования для текущего процесса: SCHED\_FIFO  
SON\_1: Текущий приоритет текущего процесса: 10  
Процесс с pid = 10003 завершен  
Процесс с pid = 10004 завершен  
Процесс с pid = 10005 завершен  
Процесс с pid = 10006 завершен  
defrozen@defrozen-ubu:~/OSstuff/lb3/task10/task10\_4

**Задание 11.** Определим величину кванта с помощью функции (POSIX) sched\_rr\_get\_interval():

Quantum.c

#include <stdio.h>

#include <sched.h>

#include <sys/mman.h>

#include <time.h>

#include <stdlib.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <unistd.h>

int main () {

struct sched\_param shdprm; // Значения параметров планирования

struct timespec qp; // Величина кванта

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

pid = getpid();

ppid = getppid();

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

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: %ld сек %ld нс\n", qp.tv\_sec, qp.tv\_nsec);

else

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

if ((pid1 = fork()) == 0) {

if (sched\_rr\_get\_interval(pid1, &qp) == 0)

printf("SON: Квант процессорного времени: %ld сек %ld нс\n", qp.tv\_sec, qp.tv\_nsec);

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

}

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

return 0;

}

Son.c

#include <stdio.h>

#include <sched.h>

#include <sys/mman.h>

#include <time.h>

#include <stdlib.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

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

return 0;

}

Соответствующий вывод будет выглядеть следующим образом:

defrozen@defrozen-ubu:~/OSstuff/lb3/task11$ ./quantum   
FATHER PARAMS: pid=10497 ppid=6165  
SCHED\_SETSCHEDULER\_1: Operation not permitted  
Квант при циклическом планировании: 0 сек 16000000 нс  
SON: Квант процессорного времени: 0 сек 16000000 нс  
SONPARAMS: pid=10498 ppid=10497  
Процесс с pid = 10498 завершен  
defrozen@defrozen-ubu:~/OSstuff/lb3/task11$ sudo ./quantum   
FATHER PARAMS: pid=10505 ppid=10504  
Квант при циклическом планировании: 0 сек 100000000 нс  
SON: Квант процессорного времени: 0 сек 100000000 нс  
SONPARAMS: pid=10506 ppid=10505  
Процесс с pid = 10506 завершен  
defrozen@defrozen-ubu:~/OSstuff/lb3/task11

Как видно, величина кванта равна 10\*\*8 = 0.1 секунде. Современные ОС Linux не имеют специального механизма, который позволял бы устанавливать величину кванта процессорного времени для RR - планировщика из приложений в отличие от более старых версий, где квантом можно было управлять, регулируя параметр процесса nice. Отрицательное значение nice —квант длиннее, положительное —короче. Степень влияния значения nice на квант в разных версиях ядра была различной. Начиная с версии Linux 2.6.24, квант SCHED\_RR не может быть изменен документированными средствами. Экспериментально это можно проверить, используя системную функцию nice():

Код nice.c:

#include <stdio.h>

#include <sched.h>

#include <sys/mman.h>

#include <time.h>

#include <stdlib.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <unistd.h>

int main () {

struct sched\_param shdprm;

struct timespec qp;

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

pid = getpid();

ppid = getppid();

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

if (nice(1000) == -1)

perror("NICE");

else

printf("Nice value = %d\n", nice(0));

shdprm.sched\_priority = 50;

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

perror("SCHED\_SETSCHEDULER\_1");

if (sched\_rr\_get\_interval(pid, &qp) == -1)

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

else

printf("Квант при циклическом планировании: %ld сек %ld нс\n", qp.tv\_sec, qp.tv\_nsec);

pid1 = fork();

if (pid1 == 0) {

if (sched\_rr\_get\_interval(pid1, &qp) == -1)

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

else

printf("SON: Квант процессорного времени: %ld сек %ld нс\n", qp.tv\_sec, qp.tv\_nsec);

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

exit(EXIT\_FAILURE);

}

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

return 0;

}

Соответствующий вывод:

defrozen@defrozen-ubu:~/OSstuff/lb3/task11$ sudo ./nice  
FATHER PARAMS: pid=11027 ppid=11026  
Nice value = 19  
Квант при циклическом планировании: 0 сек 100000000 нс  
SON: Квант процессорного времени: 0 сек 100000000 нс  
SONPARAMS: pid=11028 ppid=11027  
Процесс с pid = 11028 завершен  
defrozen@defrozen-ubu:~/OSstuff/lb3/task11$

Как видим, изменение приоритета не повлияло на квант.

**Задание 12.** Проанализируем наследование на этапах вызова функций семейства fork() и exec(). Для этого проведём эксперимент по проверке доступа потомков к файлам, открытым породившим их процессом. Рассмотрим пример кода, в котором в качестве аргументов процессам-потомкам передаются дескрипторы открытого и созданного родительским процессом файлов (file.txt и out.txt):

Код father.c:

#include <stdio.h>

#include <sched.h>

#include <sys/mman.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/wait.h>

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

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

}

int main(void) {

int pid, ppid, status;

int fdrd, fdwr;

char str1[10], str2[10];

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("file.txt", O\_RDONLY)) == -1)

perror("Opening file");

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

perror("Creating file");

itoa(str1, fdrd);

itoa(str2, fdwr);

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

{

if (fork() == 0)

{

shdprm.sched\_priority = 50;

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

perror("SCHED\_SETSCHEDULER\_1");

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

}

}

system("ps -o uid,gid,ruid,pid,ppid,pgid,tty,vsz,stat,command > log.txt");

if (close(fdrd) != 0)

perror("Closing file");

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

printf("Process pid = %d completed\n", wait(&status));

return 0;

}

Файл son.c

#include <sched.h>

#include <sys/mman.h>

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

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

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

perror("mlockall error");

char c;

int pid, ppid;

int fdrd = atoi(argv[1]);

int fdwr = atoi(argv[2]);

pid = getpid();

ppid = getppid();

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

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

sleep(5);

for (;;) {

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

return 0;

write(fdwr, &c, 1);

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

}

return 0;

}

Дескрипторы fdrd в обоих потомках указывают на запись в таблице файлов, соответствующую исходному файлу, а дескрипторы, подставляемые в качестве fdwr, на запись, соответствующую файлу вывода. Ядро смещает внутрифайловые указатели после каждой операции чтения или записи, поэтому оба процесса никогда не обратятся вместе на чтение или запись по одному и тому же указателю или смещению внутри файла.

В примере рассматривается случай, когда используется RR-политика с равными приоритетами процессов, потомки пользуются одной переменной для считанного из входного файла символа, т.к. в результате наследования разделяют одно адресное пространство, и результат выполнения программы получается следующим:

defrozen@defrozen-ubu:~/OSstuff/lb3/task12$ cat file.txt  
Happy Easter!defrozen@defrozen-ubu:~/OSstuff/lb3/task12$ sudo ./father  
son file descriptor = 3  
son params: pid=11578 ppid=11577  
son file descriptor = 3  
son params: pid=11579 ppid=11577  
pid = 11578: H  
pid = 11578: a  
pid = 11578: p  
pid = 11578: p  
pid = 11578: y  
pid = 11578:   
pid = 11578: E  
pid = 11578: a  
pid = 11578: s  
pid = 11578: t  
pid = 11578: e  
pid = 11578: r  
pid = 11578: !  
Process pid = 11578 completed  
Process pid = 11579 completed  
defrozen@defrozen-ubu:~/OSstuff/lb3/task12$

Попробуем закрыть в одном из процессов файл с заданным дескриптором, например, fdrd в son.c:

Son2.c

#include <sched.h>

#include <sys/mman.h>

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

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

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

perror("mlockall error");

char c;

int pid, ppid;

int fdrd = atoi(argv[1]);

int fdwr = atoi(argv[2]);

pid = getpid();

ppid = getppid();

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

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

sleep(5);

for (;;)

{

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

return 0;

write(fdwr, &c, 1);

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

if (close(fdrd) != 0)

perror("Closing file");

}

return 0;

}

Соответствующий вывод:

defrozen@defrozen-ubu:~/OSstuff/lb3/task12$ sudo ./father2  
son file descriptor = 3  
son params: pid=11844 ppid=11843  
son file descriptor = 3  
son params: pid=11845 ppid=11843  
pid = 11844: H  
pid = 11845: a  
Process pid = 11845 completed  
Process pid = 11844 completed  
defrozen@defrozen-ubu:~/OSstuff/lb3/task12$

Таким образом, при создании родственных процессов функциями fork() и exec() наследуются следующие параметры: таблица открытых файловых дескрипторов (file descriptor table), текущая директория (current working directory), корневая директория (root directory).  
Убедиться в наследовании других параметров при порождении потомков можно, проанализировав вывод утилиты: ps -o uid,gid,ruid,pid,ppid,pgid,tty,vsz,stat,command, т.е. от родителя наследуются UID, GID, RUID, PGID, TTY и, как было показано ранее, приоритеты и политика планирования процессов.

UID GID RUID PID PPID PGID TT VSZ STAT COMMAND  
0 0 0 12055 6165 12055 pts/0 20740 S+ sudo ./father  
0 0 0 12056 12055 12055 pts/0 2364 SL+ ./father  
0 0 0 12057 12056 12055 pts/0 2496 SL+ son 3 4  
0 0 0 12058 12056 12055 pts/0 2496 SL+ son 3 4  
0 0 0 12059 12056 12055 pts/0 2616 S+ sh -c ps -o uid,gid,ruid,pid,ppid,pgid,tty,vsz,stat,command > log.txt  
0 0 0 12060 12059 12055 pts/0 20048 R+ ps -o uid,gid,ruid,pid,ppid,pgid,tty,vsz,stat,command

**Задание 13.1.** Родственными считаются процессы, ближайшие в дереве процессов, т.е. породивший и порождённые им процессы. Их взаимодействие основывается на наследовании. Проанализируем таблицу процессов для трёх случаев:

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

Код father.c:

#include <sched.h>

#include <sys/mman.h>

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sched.h>

#include <sys/wait.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","son1", NULL);

if(fork()==0)

execl("son2","son2", argv[1], NULL);

if(fork()==0)

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

system(command);

waitpid(pid1, &status, WNOHANG);

}

Код son1.c:

#include <sched.h>

#include <sys/mman.h>

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

void main(){

int pid,ppid;

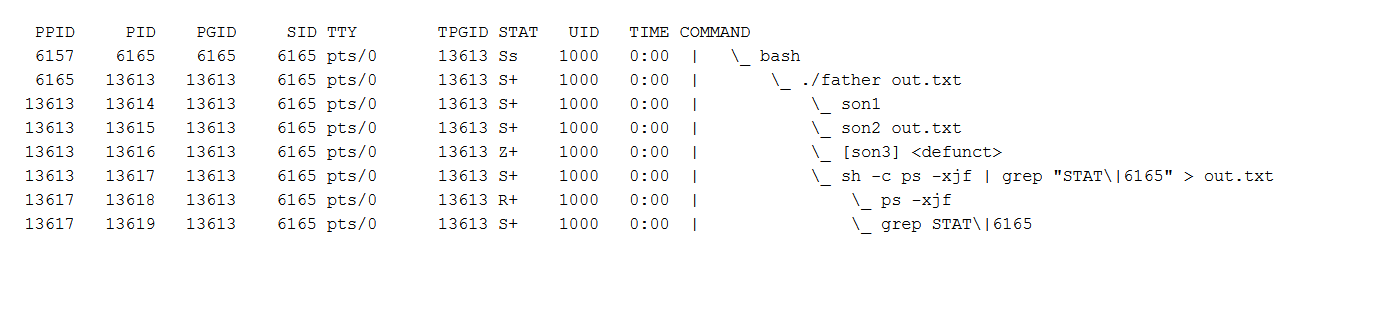
pid=getpid();

ppid=getppid();

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

sleep(3);

}



Как только процесс-родитель завершается, в консоли сразу появляется приглашение на ввод команды. А son2 продолжает своё выполнение в фоновом режиме. Т.к. время выполнения son2 много дольше, то результат выполнения процесса-потомка появляется уже после приглашения.

Б) процесс-родитель запускает процесс-потомка и, не ожидая его завершения, завершает свое выполнение. Зафиксируйте изменение родительского идентификатора процесса-сына;

Код son2.c

#include <sched.h>

#include <sys/mman.h>

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

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

int pid,ppid;

pid=getpid();

ppid=getppid();

char command[50];

sprintf(command, "ps xjf | grep son2 >> %s", argv[1]);

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

sleep(10);

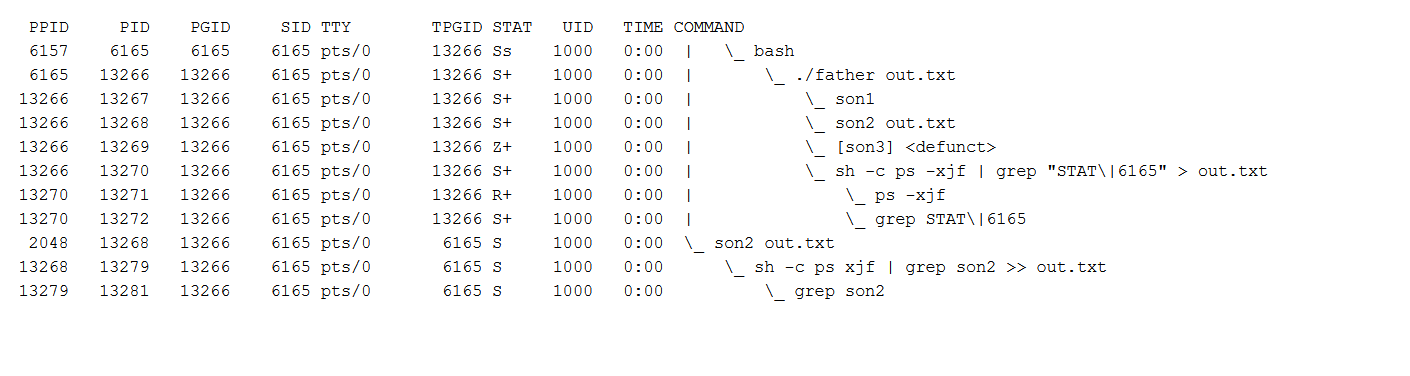
ppid=getppid();

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

system(command);

return 0;

}



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

Код son3.c:

#include <sched.h>

#include <sys/mman.h>

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int 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);

return 0;

}

defrozen@defrozen-ubu:~/OSstuff/lb3/task13/A$ ./father out.txt  
FATHER PARAMS: sid = 6165 pid=13266 ppid=6165   
SON\_1 PARAMS: pid=13267 ppid=13266  
Father creates and waits  
SON\_2 PARAMS: pid=13268 ppid=13266  
Father finished before son termination without waiting for it   
SON\_3 PARAMS: pid=13269 ppid=13266  
son3 terminated–ZOMBIE  
SON\_3 PARAMS: pid=13269 ppid=13266

Процесс-зомби фиксировался ранее в таблицах для son3 с подписью <defunct> и пометкой Z. Процесс-зомби - это процесс, который завершил свою работу, но его родительский процесс еще не получил информацию о том, что процесс завершился. Такой процесс все еще существует в системе, но не может выполнять никаких задач. Процессы-зомби занимают память и ресурсы системы, поэтому отображаются в таблице при вызове утилиты ps.

**Задание 13.1 (управление процессами посредством сигналов).** Системный вызов kill посылает сигналы указанным процессам. По умолчанию (если не указано имя или номер сигнала) посылается сигнал SIGTERM. Идентификатор процесса является аргументом для этой утилиты: если он больше нуля, то сигнал посылается процессу с указанным pid, если он равен нулю, то сигнал посылается всем процессам, принадлежащим пользователю, если он меньше нуля, то он воспринимается как идентификатор группы процессов, и тогда сигнал посылается всей группе.

defrozen@defrozen-ubu:~/OSstuff/lb3/task13/123$ kill -l  
1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP  
6) SIGABRT 7) SIGBUS 8) SIGFPE 9) SIGKILL 10) SIGUSR1  
11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM  
16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP  
21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ  
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO 30) SIGPWR  
31) SIGSYS 34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3  
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8  
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13  
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12  
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7  
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2  
63) SIGRTMAX-1 64) SIGRTMAX   
defrozen@defrozen-ubu:~/OSstuff/lb3/task13/123$

Функция системного вызова signal заключается в том, чтобы задать определённые действия для программы в ответ на пришедший сигнал. В качестве действий можно задать следующие значения: SIG\_DFL, SIG\_IGN или указатель на собственную функцию обработки. SIG\_DFL означает, что процесс должен реагировать на сигнал, как задано по умолчанию (чаще всего это завершение процесса), SIG\_IGN (нельзя задать для SIGSTOP и SIGKILL) означает, что нужно игнорировать сигнал.

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.

Код son1.c:  
#include <signal.h>

#include <stdio.h>

#include <sched.h>

#include <sys/mman.h>

#include <fcntl.h>

#include <stdlib.h>

#include <unistd.h>

int main()

{

signal(SIGUSR1, SIG\_DFL);

sleep(5);

}

Код son2.c:  
#include <signal.h>

#include <stdio.h>

#include <sched.h>

#include <sys/mman.h>

#include <fcntl.h>

#include <stdlib.h>

#include <unistd.h>

int main()

{

signal(SIGUSR1, SIG\_IGN);

sleep(5);

}

Код son3.c:  
#include <signal.h>

#include <stdio.h>

#include <sched.h>

#include <sys/mman.h>

#include <fcntl.h>

#include <stdlib.h>

#include <unistd.h>

void SIGUSR1\_handler(int sig\_no)

{

printf("SIGUSR1\_handler running!\n");

}

int main() {

signal(SIGUSR1, SIGUSR1\_handler);

sleep(5);

return 0;

}

Код father.c:  
#include <signal.h>

#include <stdio.h>

#include <sched.h>

#include <sys/mman.h>

#include <fcntl.h>

#include <stdlib.h>

#include <unistd.h>

int main(){

int pid1, pid2, pid3;

pid1 = fork();

if (pid1 == 0)

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

pid2 = fork();

if (pid2 == 0)

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

pid3 = fork();

if (pid3 == 0)

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

system("echo before signal sent");

system("ps -l");

kill(pid1, SIGUSR1);

kill(pid2, SIGUSR1);

kill(pid3, SIGUSR1);

system("echo after signal sent");

system("ps -l");

}

defrozen@defrozen-ubu:~/OSstuff/lb3/task13/123$ ./father   
before signal sent  
F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD  
0 S 1000 6165 6157 0 80 0 - 5078 do\_wai pts/0 00:00:00 bash  
0 S 1000 15223 6165 0 80 0 - 593 do\_wai pts/0 00:00:00 father  
0 S 1000 15224 15223 0 80 0 - 593 hrtime pts/0 00:00:00 son1  
0 S 1000 15225 15223 0 80 0 - 593 hrtime pts/0 00:00:00 son2  
0 S 1000 15226 15223 0 80 0 - 593 hrtime pts/0 00:00:00 son3  
0 S 1000 15228 15223 0 80 0 - 656 do\_wai pts/0 00:00:00 sh  
4 R 1000 15229 15228 0 80 0 - 5012 - pts/0 00:00:00 ps  
SIGUSR1\_handler running!  
after signal sent  
F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD  
0 S 1000 6165 6157 0 80 0 - 5078 do\_wai pts/0 00:00:00 bash  
0 S 1000 15223 6165 0 80 0 - 593 do\_wai pts/0 00:00:00 father  
0 Z 1000 15224 15223 0 80 0 - 0 - pts/0 00:00:00 son1 <defunct>  
0 S 1000 15225 15223 0 80 0 - 593 hrtime pts/0 00:00:00 son2  
0 Z 1000 15226 15223 0 80 0 - 0 - pts/0 00:00:00 son3 <defunct>  
0 S 1000 15231 15223 0 80 0 - 656 do\_wai pts/0 00:00:00 sh  
4 R 1000 15232 15231 0 80 0 - 5012 - pts/0 00:00:00 ps  
defrozen@defrozen-ubu:~/OSstuff/lb3/task13/123$

**Задание 13.2.** Для организации посылки сигналов любым двум процессам, находящимся в разных состояниях, был написан следующий код:  
Код father.c:  
#include <signal.h>

#include <stdio.h>

#include <sched.h>

#include <sys/mman.h>

#include <fcntl.h>

#include <stdlib.h>

#include <unistd.h>

#include <time.h>

int main(){

int pid1, pid2, pid3;

pid1 = fork();

if (pid1 == 0)

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

pid2 = fork();

if (pid2 == 0)

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

printf("father running\n");

system("ps");

kill(pid1, SIGUSR1);

kill(pid2, SIGUSR1);

time\_t st\_t;

st\_t = time(NULL);

printf("signals sended at time %s\n", ctime(&st\_t));

sleep(2);

system("ps");

}

Код son1.c:  
#include <signal.h>

#include <stdio.h>

#include <sched.h>

#include <sys/mman.h>

#include <fcntl.h>

#include <stdlib.h>

#include <unistd.h>

#include <time.h>

void SIGUSR1\_handler(int sig\_no)

{

time\_t st\_t;

st\_t = time(NULL);

printf("son1 (sleeping) received the signal at time %s\n", ctime(&st\_t));

exit(0);

}

int main() {

signal(SIGUSR1, SIGUSR1\_handler);

printf("son1 running\n");

sleep(3);

return 0;

}

Код son2.c:  
#include <signal.h>

#include <stdio.h>

#include <sched.h>

#include <sys/mman.h>

#include <fcntl.h>

#include <stdlib.h>

#include <unistd.h>

#include <time.h>

void SIGUSR1\_handler(int sig\_no)

{

time\_t st\_t;

st\_t = time(NULL);

printf("son1 (active) received the signal at time %s\n", ctime(&st\_t));

exit(0);

}

int main() {

signal(SIGUSR1, SIGUSR1\_handler);

printf("son2 running\n");

while(1){

//inf loop

}

return 0;

}

defrozen@defrozen-ubu:~/OSstuff/lb3/task13/subtask3$ ./father   
father running  
son1 running  
son2 running  
PID TTY TIME CMD  
6165 pts/0 00:00:00 bash  
15818 pts/0 00:00:00 father  
15819 pts/0 00:00:00 son1  
15820 pts/0 00:00:00 son2  
15821 pts/0 00:00:00 sh  
15822 pts/0 00:00:00 ps  
signals sended at time Sun Apr 16 02:23:41 2023  
son1 (sleeping) received the signal at time Sun Apr 16 02:23:41 2023  
  
son1 (active) received the signal at time Sun Apr 16 02:23:41 2023  
  
PID TTY TIME CMD  
6165 pts/0 00:00:00 bash  
15818 pts/0 00:00:00 father  
15819 pts/0 00:00:00 son1 <defunct>  
15820 pts/0 00:00:00 son2 <defunct>  
15823 pts/0 00:00:00 sh  
15824 pts/0 00:00:00 ps  
defrozen@defrozen-ubu:~/OSstuff/lb3/task13/subtask3$

**Задание 14.** Для того, чтобы отследить список заданий и порядок их выполнения, запустим в фоновом режиме sleep с разными значениями.

defrozen@defrozen-ubu:~/OSstuff/lb3$ sleep 100 & sleep 120 & sleep 170 & sleep 220 &  
[1] 14056  
[2] 14057  
[3] 14058  
[4] 14059

Видно, что первой задаче (sleep 100) присвоен в квадратных скобках порядок выполнения 1 и pid 14056. Чтобы вывести список запущенных процессов и их статус, воспользуемся командой jobs –l

defrozen@defrozen-ubu:~/OSstuff/lb3$ jobs -l  
[1] 14056 Running sleep 100 &  
[2] 14057 Running sleep 120 &  
[3]- 14058 Running sleep 170 &  
[4]+ 14059 Running sleep 220 &

Видно, что порядок выполнения и pid совпадают, далее указан статус выполнения (выполняется) и сама команда. Можно вывести конкретную задачу по номеру передав % и номер:

defrozen@defrozen-ubu:~/OSstuff/lb3$ jobs -l %2  
[2] 14057 Running sleep 120 &

Чтобы вывести последний процесс можно передать вместо номера % :

defrozen@defrozen-ubu:~/OSstuff/lb3$ jobs -l %%  
[4]+ 14059 Running sleep 220 &

Вызовем еще раз команды в фоновом режиме и воспользуемся notify с временными промежутками. Тогда при каждом завершении работы процесса в фоновом режиме будет сообщаться в терминале:

defrozen@defrozen-ubu:~/OSstuff/lb3$ notify %

[1] Done sleep 10

Создадим несколько процессов в фоновом режиме, выведем их список и вернем задачу одну из задач в приоритетном режиме с помощью fg:

defrozen@defrozen-ubu:~/OSstuff/lb3$ jobs -l  
[1] 14056 Running sleep 100 &  
[2] 14057 Running sleep 120 &  
[3]- 14058 Running sleep 170 &  
[4]+ 14059 Running sleep 220 &  
defrozen@defrozen-ubu:~/OSstuff/lb3$ fg %4  
sleep 220  
[1] Done sleep 100  
[2] Done sleep 120  
[3]- Done sleep 170  
defrozen@defrozen-ubu:~/OSstuff/lb3$ jobs -l

Видно, что при использовании fg задача возвращается в приоритетном режиме - то есть выполняется непосредственно.

Создадим несколько задач в фоновом режиме, выведем их список и с помощью kill, передав утилите pid процесса, завершим его.

defrozen@defrozen-ubu:~/OSstuff/lb3$ sleep 100 & sleep 110 & sleep 120 &  
[1] 17668  
[2] 17669  
[3] 17670  
defrozen@defrozen-ubu:~/OSstuff/lb3$ jobs -l  
[1] 17668 Running sleep 100 &  
[2]- 17669 Running sleep 110 &  
[3]+ 17670 Running sleep 120 &  
defrozen@defrozen-ubu:~/OSstuff/lb3$ kill 17669   
defrozen@defrozen-ubu:~/OSstuff/lb3$ jobs -l  
[1] 17668 Running sleep 100 &  
[2]- 17669 Terminated sleep 110  
[3]+ 17670 Running sleep 120 &  
defrozen@defrozen-ubu:~/OSstuff/lb3$

Видно, что статус процесса, после применения утилиты kill изменился на «Terminated», то есть процесс был принудительно завершен.

**Задание 15**. nice — утилита, запускающая программу с измененным приоритетом. Если не указано ни одного аргументы, команда выводит текущий унаследованный приоритет. В противном случае, nice запускает команду с указанным приоритетом. Если смещение не указано, то приоритет команды увеличивается на 10. команда nice может смещать приоритет в диапазоне от -20 до 19 включительно, когда используются права суперпользователя. Когда команда выполняется обычным пользователем, диапазон изменяется от 0 до 19.

defrozen@defrozen-ubu:~/OSstuff/lb3$ echo first\_call & echo second\_call  
[1] 18097  
second\_call  
first\_call  
defrozen@defrozen-ubu:~/OSstuff/lb3$ nice -n 1 echo first\_call & nice -n 10 echo second\_call  
[2] 18100  
first\_call  
second\_call

Видно, что при использовании nice меняется приоритет задач.

Для нахождения диапазона значений, принимаемых setpriority реализована программа priority.c:

#include <stdio.h>

#include <sys/time.h>

#include <sys/resource.h>

#include <stdlib.h>

#include <unistd.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;

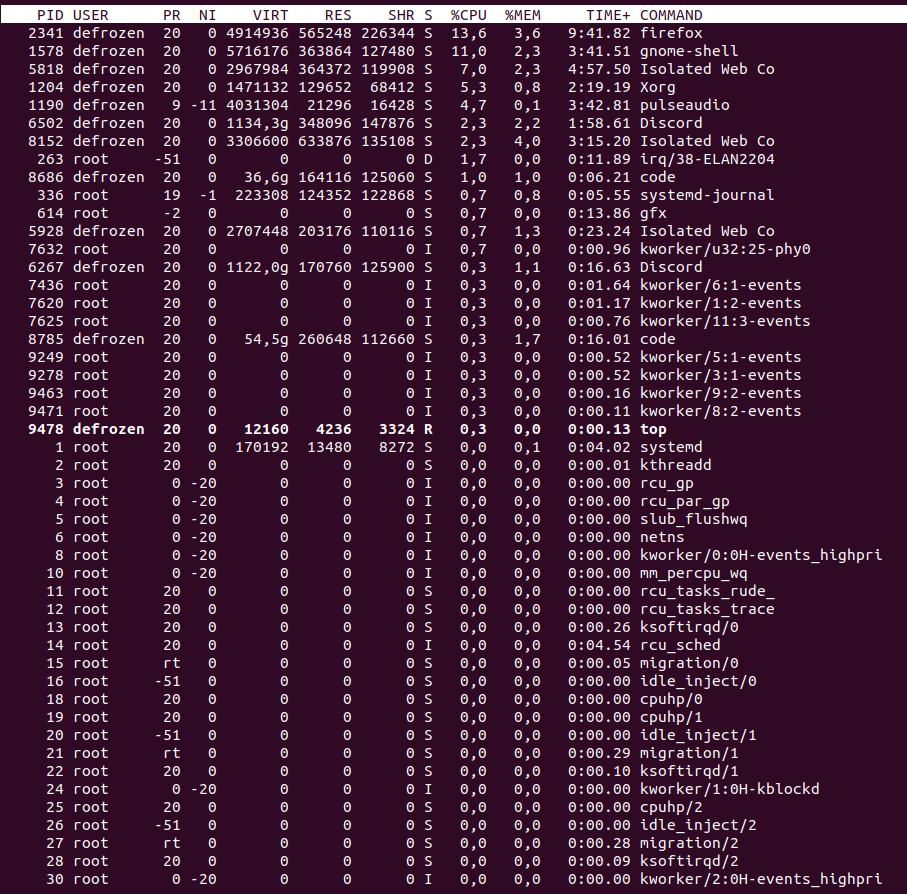
}

}

}

defrozen@defrozen-ubu:~/OSstuff/lb3/task15$ ./priority   
Нижняя граница = 0  
Запросили 0, получили 0  
Верхняя граница = 19  
Запросили 20, получили 19  
defrozen@defrozen-ubu:~/OSstuff/lb3/task15$ sudo ./priority   
[sudo] password for defrozen:   
Нижняя граница = -20  
Запросили -20, получили -20  
Верхняя граница = 19  
Запросили 20, получили 19  
defrozen@defrozen-ubu:~/OSstuff/lb3/task15$

Утилита top позволяет выводить информацию о системе:



Видим, что пользовательский приоритет для запуска приложений из shell равен 20. Также заметно, что существует разница в приоритетах для системных и пользовательских процессов. Обычно системные процессы имеют более высокий приоритет, чем пользовательские процессы, поскольку они обеспечивают работу ядра операционной системы.

Приоритеты реального времени могут использоваться для некоторых системных процессов, которые должны выполняться в строго заданные временные рамки. Однако, использование приоритетов реального времени может привести к проблемам с производительностью и стабильностью системы.

**Задание 16**. nohup(1) — утилита, позволяющая запустить команду, невосприимчивую к сигналам потери связи (hungup), и чей вывод будет направлен не на терминал, а в файл nohup.out. Таким образом, команда будет выполняться в фоновом режиме даже тогда, когда пользователь выйдет из системы.

Реализуем программу nohup.c:

#include <stdio.h>

void main()

{

int i;

for(i = 0; i < 100000000; i++);

}

Скомпилируем и выполним:

[1] 18871

nohup: ignoring input and appending output to 'nohup.out'

Pid процесса 18871, система сообщает, что программа игнорирует входные данные (если они будут) и записывает выходные в файл 'nohup.out' (если программа будет что-то выводить). Выйдем из системы и введем команду ps xa | grep "./nohup":

defrozen@defrozen-ubu:~$ ps xa | grep "./honup"   
18871 pts/0 S+ 0:00 grep —color=auto ./honup  
defrozen@defrozen-ubu:~$

Даже при выходе из системы процесс остается.

**Задание 17.** 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 (некоторые системы рекомендуют резервировать UID с 101 по 499, или 999).

Чтобы определить UID процесса, можно воспользоваться утилитой ps. Просмотрим текущие параметры пользователя и суперпользователя, используя команду id:

defrozen@defrozen-ubu:~/OSstuff/lb3/task16$ id  
uid=1000(defrozen) gid=1000(defrozen) groups=1000(defrozen),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),132(lxd),133(sambashare),998(docker)

defrozen@defrozen-ubu:~/OSstuff/lb3/task16$ sudo su  
[sudo] password for defrozen:   
root@defrozen-oumuamua:/home/defrozen/OSstuff/lb3/task16# id  
uid=0(root) gid=0(root) groups=0(root)  
root@defrozen-oumuamua:/home/defrozen/OSstuff/lb3/task16#

Посмотреть максимальные и минимальные значения для UID можно в файле "/etc/login.defs":

root@defrozen-oumuamua:/home/defrozen/OSstuff/lb3/task16# cat /etc/login.defs  
# Min/max values for automatic uid selection in useradd  
#  
UID\_MIN 1000  
UID\_MAX 60000  
# System accounts  
[#SYS\_UID\_MIN](https://vk.com/im?sel=341534830&st=%23SYS_UID_MIN) 100  
[#SYS\_UID\_MAX](https://vk.com/im?sel=341534830&st=%23SYS_UID_MAX) 999  
  
#  
# Min/max values for automatic gid selection in groupadd  
#  
GID\_MIN 1000  
GID\_MAX 60000  
# System accounts  
[#SYS\_GID\_MIN](https://vk.com/im?sel=341534830&st=%23SYS_GID_MIN) 100  
[#SYS\_GID\_MAX](https://vk.com/im?sel=341534830&st=%23SYS_GID_MAX) 999

Минимальное значение pid обычно равно 1 и принадлежит процессу init, который является первым процессом, запускаемым ядром операционной системы. Максимальное значение pid может быть достигнуто при достижении максимального значения целочисленного типа, используемого для хранения pid. Например, в Linux максимальное значение pid может быть до 4194303.

Системные процессы могут быть отличены от пользовательских по их принадлежности к той или иной группе. Системные процессы обычно запускаются от имени пользователя root.

Некоторые из самых важных системных процессов включают:

1. init - первый процесс, запускаемый ядром операционной системы, который выполняет инициализацию системы и запускает другие процессы.

2.systemd - системный менеджер, который управляет процессами и службами в Linux-системах.

3. kernel - ядро операционной системы, которое обеспечивает взаимодействие между аппаратным обеспечением и программным обеспечением.

4. cron - системный планировщик задач, который позволяет запускать задачи в заданное время или с определенной периодичностью.

5. sshd - демон, который обеспечивает удаленный доступ к системе по протоколу SSH.

6. syslogd - демон, который обеспечивает запись системных сообщений в журналы.

7. udevd - демон, который обеспечивает автоматическое определение и настройку устройств в системе.

8. networkd - демон, который управляет сетевыми интерфейсами и соединениями в Linux-системах.