**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**КАФЕДРА МО ЭВМ**

**ОТЧЕТ**

**по лабораторной работе №3**

**по дисциплине «Операционные системы»**

**Тема: Порождение и запуск процессов в Unix-подобных ОС**

Студент гр. 1303 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Герасименко Я.Д.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Душутина Е.В.

Санкт-Петербург

2023

**Цель работы**

Освоение механизмов порождения и запуска процессов в Unix-подобных операционных системах, а также приобретение навыков работы с командной оболочкой и основных концепций управления процессами, изучение различных способов запуска процессов, передачи параметров и управления ими, анализ их состояния и использование системных вызовов и API для более эффективного управления процессами.

**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().

щ

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*.

*Взаимодействие родственных процессов*

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

18. Подготовьте программу, формирующую несколько нитей. Нити для эксперимента могут быть практически идентичны. Например, каждая нить в цикле: выводит на печать собственное имя и 130 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).

**Выполнение работы.**

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

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**#include <sys/wait.h>**

**int main() {**

**pid\_t pid;**

**pid = fork(); // создаем процесс-потомок**

**if (pid == -1) {**

**perror("ошибка fork");**

**exit(EXIT\_FAILURE);**

**} else if (pid == 0) {**

**// код процесса-потомка**

**printf("Дочерний процесс %d запущен.\n", getpid());**

**execlp("ls", "ls", NULL); // исполняем команду ls**

**perror("ошибка exec"); // если execlp вернула ошибку**

**exit(EXIT\_FAILURE);**

**} else {**

**// код родительского процесса**

**printf("Родительский процесс %d запущен.\n", getpid());**

**wait(NULL); // ожидаем завершения процесса-потомка**

**printf("Родительский процесс %d завершен.\n", getpid());**

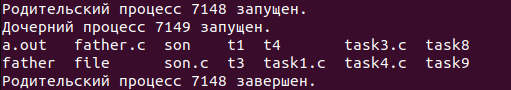
**exit(EXIT\_SUCCESS);**

**}**

**}**

Этот код создает новый процесс с помощью функции fork() и позволяет различать, в каком процессе выполняется код, с помощью проверки значения переменной pid. Код выводит в консоль информацию о PID текущего и родительского процессов, а также изменяет и выводит в консоль значение переменной n в зависимости от того, в каком процессе код выполняется. В конце процесс завершается с кодом выхода

Результат работы программы:



2.Выполните сначала однократные вычисления в каждом процессе, обратитевнимание, 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. Объяснитерезультаты. Сделайте выводы об использовании адресного пространства

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**#include <sys/wait.h>**

**int main() {**

**pid\_t pid;**

**pid = fork(); // создаем процесс-потомок**

**if (pid == -1) {**

**perror("ошибка fork");**

**exit(EXIT\_FAILURE);**

**} else if (pid == 0) {**

**// код процесса-потомка**

**printf("Дочерний процесс %d запущен.\n", getpid());**

**execlp("ls", "ls", NULL); // исполняем команду ls**

**perror("ошибка exec"); // если execlp вернула ошибку**

**exit(EXIT\_FAILURE);**

**} else {**

**// код родительского процесса**

**printf("Родительский процесс %d запущен.\n", getpid());**

**wait(NULL); // ожидаем завершения процесса-потомка**

**printf("Родительский процесс %d завершен.\n", getpid());**

**exit(EXIT\_SUCCESS);**

**}**

**}**

В данном случае процесс-родитель создает процесс-потомок, и каждый из них выполняет свою команду (в данном случае ls). При этом порядок выполнения команд не гарантируется. Однако, поскольку процесс-потомок создается после команды fork(), он обычно получает доступ к процессорному ресурсу позже, чем процесс-родитель. Это связано с тем, что при вызове fork() операционная система создает копию адресного пространства процесса-родителя, а затем передает его процессу-потомку. Поэтому процесс-потомок должен выполнить дополнительную работу, чтобы скопировать данные из адресного пространства процесса-родителя в свое собственное адресное пространство.

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

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**#include <sys/wait.h>**

**int main() {**

**pid\_t pid;**

**int i;**

**pid = fork(); // создаем процесс-потомок**

**if (pid == -1) {**

**perror("ошибка fork");**

**exit(EXIT\_FAILURE);**

**} else if (pid == 0) {**

**// код процесса-потомка**

**printf("Дочерний процесс %d запущен.\n", getpid());**

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

**printf("Дочерний процесс %d выполняет итерацию %d.\n", getpid(), i);**

**sleep(1); // замедляем выполнение цикла на 1 секунду**

**}**

**exit(EXIT\_SUCCESS);**

**} else {**

**// код родительского процесса**

**printf("Родительский процесс %d запущен.\n", getpid());**

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

**printf("Родительский процесс %d выполняет итерацию %d.\n", getpid(), i);**

**sleep(1); // замедляем выполнение цикла на 1 секунду**

**}**

**wait(NULL); // ожидаем завершения процесса-потомка**

**printf("Родительский процесс %d завершен.\n", getpid());**

**exit(EXIT\_SUCCESS);**

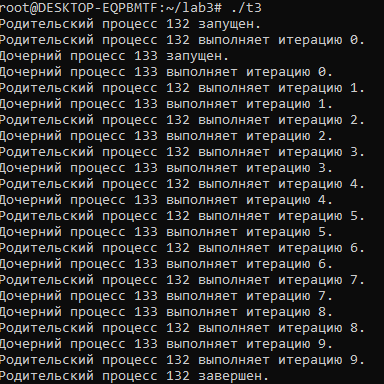
**}**

**}**

После замены однократных вычислений на циклы в каждом процессе, можно наблюдать конкуренцию за процессорный ресурс. В данном случае каждый процесс выполняет цикл из 10 итераций, замедляя выполнение каждой итерации на 1 секунду при помощи функции sleep(). Таким образом, каждый процесс занимает процессорный ресурс в течение 10 секунд, пока другой процесс находится в состоянии ожидания.

Использование системных функций fork(), exec(), wait() и sleep() позволяет создавать многопоточные программы на языке Си, которые используют конкуренцию за процессорный ресурс для повышения производительности

Результат работы программы:



4. Изменим процедуру планирования и повторим эксперимент.

GNU nano 6.2 task4.c #include <stdio.h>

#include <sys/resource.h>

#include <sys/types.h>

#include <sched.h>

#include <unistd.h>

#include <stdlib.h>

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

{

int n = 0;

struct sched\_param shdprm;

shdprm.sched\_priority = 1;

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

perror("SCHED\_SETSCHEDULER");

}

char command\_fifo[100];

sprintf(command\_fifo, "chrt -p %d", getpid());

system(command\_fifo);

int pid = fork();

if(pid == -1)

{

perror("fork error");

exit(1);

}

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

{

if (pid == 0)

{

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

n += 1;

}

else

{

printf("PARENT: pid = %d, ppid = %d \nValue n: %d\n", getpid(), getppid(), n);

n -= 1;

}

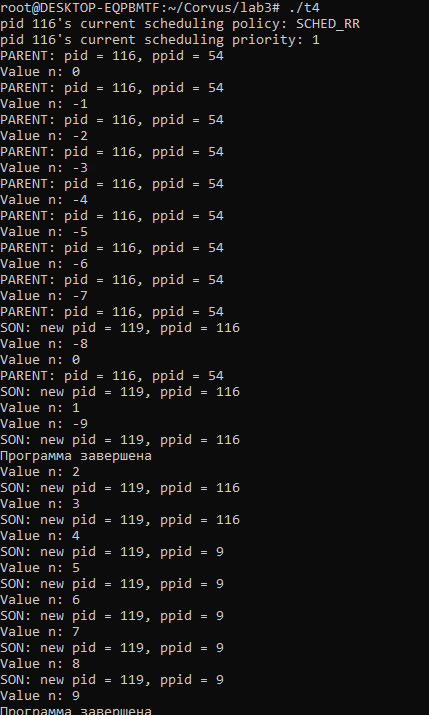
}

printf("Программа завершена\n");

exit(1);

}

Изменение процедуры планирования SCHED\_OTHER может привести к изменению порядка выполнения процессов и распределению процессорного времени между ними. Для изменения процедуры планирования можно использовать функцию sched\_setscheduler(), которая позволяет задать различные алгоритмы планирования, такие как FIFO или Round Robin.

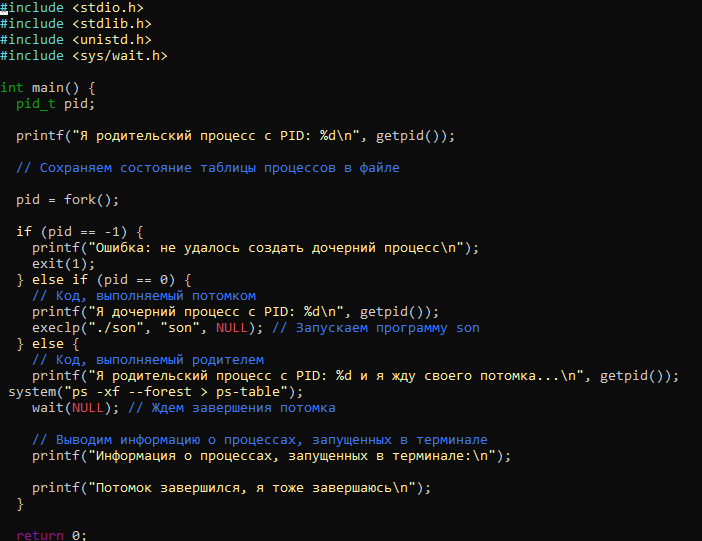
Результат работы программы:  


В выводе можно наблюдать, что когда устанавливается алгоритм Round Robin, родительский процесс и процесс-потомок выполняются поочередно, каждый на одну итерацию цикла.

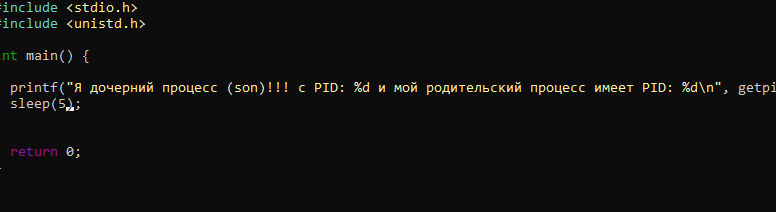
5,6.

были созданы две программы: father.c и son.c. Они были скомпилированы в отдельные исполняемые файлы father.out и son.out. Для того, чтобы получить информацию о процессах, запущенных с терминала, была использована системная функция system() с командой ps -abcde > file. Эта команда запускает процесс ps с опцией -abcde для получения таблицы процессов, после чего перенаправляет вывод в файл file.

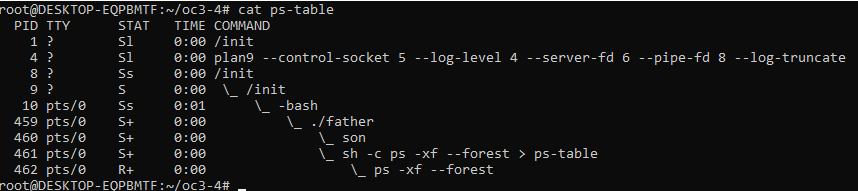
father.c



son.c

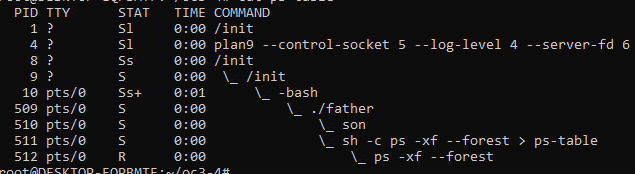


Проанализировав текстовый файл после запуска программы, видим, что процесс son является дочерним для father, т.к. иерархия процессов выведена в древовидной форме.



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

Сначала программа запускается обыкновенно, а далее фоново. Разница заключается только в выводе программ, а таблица процессов полностью одинакова.



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

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <stdlib.h>

void run\_child(const char \*exec\_func) {

pid\_t pid;

int status;

pid = fork();

if (pid < 0) {

perror("fork");

return;

}

if (pid == 0) { // Child process

execl("./son.out", "son.out", exec\_func, (char \*)NULL);

perror("execl");

\_exit(1);

} else { // Parent process

wait(&status);

printf("Parent: Child process with %s terminated with status %d\n", exec\_func, WEXITSTATUS(status));

}

}

int main() {

const char \*exec\_funcs[] = {"execl", "execlp", "execle", "execv", "execvp", "execvpe", NULL};

printf("Running 'ps' before starting child processes:\n");

system("ps -aux | grep father.out");

printf("\n");

for (int i = 0; exec\_funcs[i] != NULL; i++) {

run\_child(exec\_funcs[i]);

}

printf("\nRunning 'ps' after all child processes terminated:\n");

system("ps -aux | grep father.out");

return 0;

}

son.c

GNU nano 6.2 son.c #include <stdio.h>

#include <unistd.h>

#include <string.h>

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

if (argc < 2) {

fprintf(stderr, "Usage: %s <exec\_function>\n", argv[0]);

return 1;

}

if (strcmp(argv[1], "execl") == 0) {

printf("Using execl\n");

} else if (strcmp(argv[1], "execlp") == 0) {

printf("Using execlp\n");

} else if (strcmp(argv[1], "execle") == 0) {

printf("Using execle\n");

} else if (strcmp(argv[1], "execv") == 0) {

printf("Using execv\n");

} else if (strcmp(argv[1], "execvp") == 0) {

printf("Using execvp\n");

} else if (strcmp(argv[1], "execvpe") == 0) {

printf("Using execvpe\n");

} else {

fprintf(stderr, "Unknown exec function: %s\n", argv[1]);

return 1;

}

return 0;

}

execl(): выполняет файл по заданному пути, передавая ему аргументы в виде списка переменных (каждый аргумент передается в качестве отдельного аргумента функции).

execlp(): выполняет файл с именем, передавая ему аргументы в виде списка переменных (каждый аргумент передается в качестве отдельного аргумента функции). Эта функция ищет файл в $PATH (переменная среды, которая содержит пути к директориям, где хранятся исполняемые файлы).

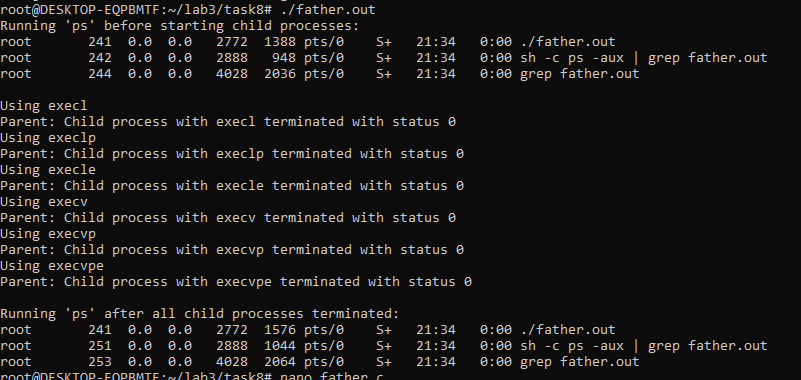
execle(): выполняет файл по заданному пути, передавая ему аргументы в виде списка переменных (каждый аргумент передается в качестве отдельного аргумента функции) и заданный массив переменных среды.

execv(): выполняет файл по заданному пути, передавая ему аргументы в виде массива (каждый аргумент представлен как элемент массива).

execvp(): выполняет файл с именем, передавая ему аргументы в виде массива (каждый аргумент представлен как элемент массива). Эта функция ищет файл в $PATH.

execvpe(): выполняет файл с именем, передавая ему аргументы в виде массива (каждый аргумент представлен как элемент массива) и заданный массив переменных среды. Эта функция ищет файл в $PATH.

Результат вывода аналогичен при каждом аргументе.



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

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <stdlib.h>

int main() {

pid\_t pid;

int status;

int num\_children = 6;

for (int i = 0; i < num\_children; i++) {

pid = fork();

if (pid < 0) {

perror("fork");

return 1;

}

if (pid == 0) { // Child process

if (i == 5) {

printf("Child %d (PID %d) executing 'ls -l'\n", i + 1, getpid());

execl("/bin/ls", "ls", "-l", NULL);

perror("execl");

exit(1);

} else {

printf("Child %d (PID %d) started\n", i + 1, getpid());

sleep(i + 1);

printf("Child %d (PID %d) exiting\n", i + 1, getpid());

exit(0);

}

}

}

// Parent process: отслеживаем только четные дочерние процессы и процесс 'ls -l'

for (int i = 0; i < num\_children; i++) {

if (i % 2 == 0 || i == 5) {

pid = waitpid(-1, &status, 0);

if (pid > 0) {

printf("Parent: Child %d exited\n", pid);

}

}

}

sleep(10); // Задержка перед выводом дерева процессов

system("ps -xf --forest > file");

return 0;

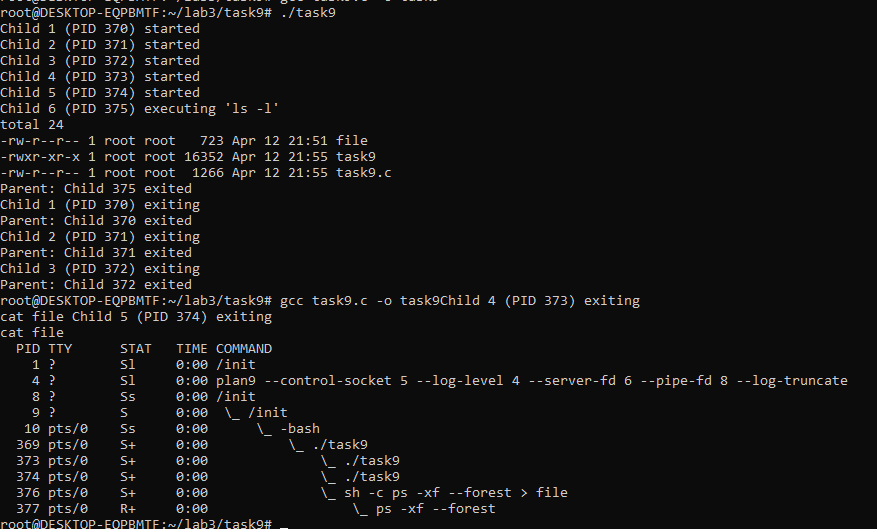
}

Программа создает родительский процесс (father), который в свою очередь создает шесть дочерних процессов. Первые пять дочерних процессов выполняют sleep на разное количество времени (от 1 до 5 секунд) и затем завершаются. Шестой дочерний процесс выполняет команду ls -l с использованием функции execl() и также завершается после выполнения.

Родительский процесс отслеживает завершение четных дочерних процессов (2, 4, 6) с помощью функции waitpid(). Функция waitpid() позволяет родителю ждать завершения определенных дочерних процессов, основываясь на их PID.

В данном случае, функция waitpid() используется для отслеживания завершения только четных дочерних процессов. Это демонстрирует, как родитель может выбирать и отслеживать определенные дочерние процессы, а не все порожденные потомки.

После завершения всех четных дочерних процессов, программа сохраняет текущую структуру дерева процессов в файл file с использованием команды ps -xf --forest. Затем программа завершает свою работу.



10)

10. Проанализируйте очередность исполнения процессов.

10.1. очередность исполнения процессов, порожденных вложенными вызовами fork(). Порядок выполнения процессов, генерируемых вложенными вызовами fork(): Когда процесс вызывает 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. Каждый новый дочерний процесс наследует копию адресного пространства и контекста выполнения родительского процесса. 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.

10.2 Попытаемся изменить приоритеты и текущую политику планирования. Пример кода программы:

#include <stdio.h>

#include <sched.h>

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

17

int main() {

struct sched\_param shdprm; // Значения параметров планирования

int pid, pid1, pid2, ppid;

pid = getpid();

ppid = getppid();

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

shdprm.sched\_priority = 50;

// Устанавливаем политику планирования и значение приоритета в 50

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

perror("SCHED\_SETSCHEDULER");

}// Выводим сообщение об ошибке, если sched\_setscheduler() вернула -1

// Создаем два дочерних процесса

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

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

// Запускаем исполняемый файл son1 в первом дочернем процессе

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

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

// Запускаем исполняемый файл son2 во втором дочернем процессе

// Выводим текущую политику планирования и значение приоритета 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("Текущий приоритет текущего процесса: %d\n",

shdprm.sched\_priority);

else

perror("SCHED\_GETPARAM");

// Выводим сообщение об ошибке, если sched\_getparam() вернула -1

return 0;

}

Код son1.c, son2.c

#include <stdio.h>

#include <stdlib.h>

#include <sched.h>

#include <unistd.h>

#include <sys/types.h>

int main() {

struct sched\_param shdprm; // Значенияпараметровпланирования 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: ");

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",

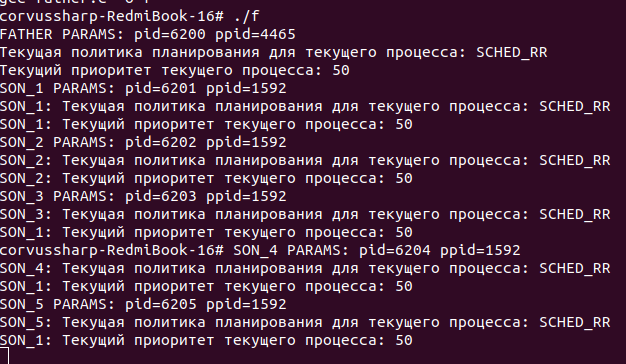
shdprm.sched\_priority);

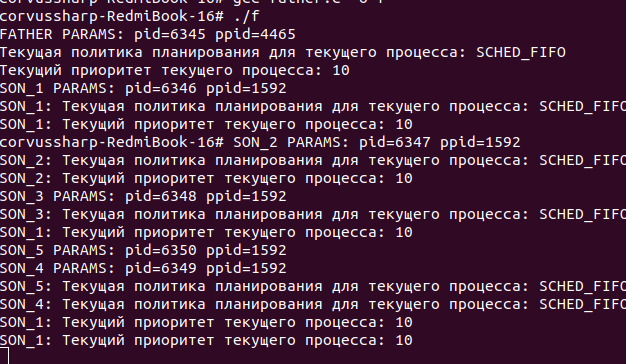
else

perror("SCHED\_GETPARAM");

return 0;

}





потомки наследуют политику планирования и приоритет родительского процесса при политиках планирования SHED\_RR и SHED\_FIFO

10.3. Поменяем порядок очереди в RR-процедуре.

#include <stdio.h>

#include <sched.h>

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

struct sched\_param shdprm; // значения параметров планирования

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

int n, m, l, k; // переменные для задания значений приоритетов,

n = 50; m = 60; l = 10; k = 80; // заданные значения приоритетов c политикой

RR

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");

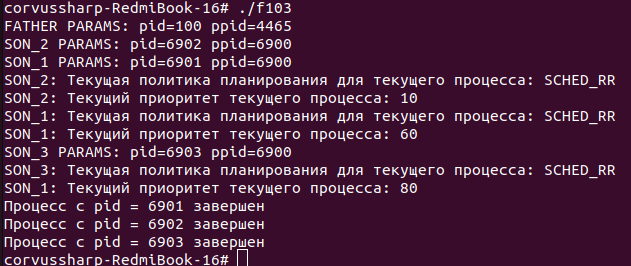
execl("son3", "son3", NULL);

}

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));



С процедурой планирования FIFO процессы выполняются в том порядке, в

котором были созданы.

10.4. Можно ли задать разные процедуры планирования разным процессам с

одинаковыми приоритетами? Как они будут конкурировать, подтвердите

экспериментально.

Да, можно назначать разные процедуры планирования разным

процессам с одинаковыми приоритетами.

Чтобы экспериментально подтвердить, как разные процедуры планирования

влияют на конкуренцию между процессами с одинаковыми приоритетами, мы

можем создать несколько процессов с одинаковыми уровнями приоритета и

разными процедурами планирования. Затем мы можем запустить эти процессы в

той же системе и измерить время их выполнения.

Вот пример кода, демонстрирующий, как назначить разные процедуры

планирования разным процессам с одинаковыми приоритетами:

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/wait.h>

#include <sched.h>

#include <time.h>

void perform\_computation() {

for (int i = 0; i < 1000000000; ++i)

2 + 2;

}

int main() {

pid\_t pid1, pid2;

int status;

struct timespec start\_time, end\_time;

double elapsed\_time;

// Create two child processes with the same priority level

pid1 = fork();

if (pid1 == 0) { // child process 1

// Assign Round-Robin scheduling procedure

struct sched\_param param1 = {.sched\_priority = 1};

sched\_setscheduler(0, SCHED\_RR, &param1);

// Perform some computation

clock\_gettime(CLOCK\_MONOTONIC, &start\_time);

perform\_computation();

clock\_gettime(CLOCK\_MONOTONIC, &end\_time);

elapsed\_time = (end\_time.tv\_sec - start\_time.tv\_sec) + (end\_time.tv\_nsec -

start\_time.tv\_nsec) / 1e9;

printf("Child process 1 execution time: %.6f seconds\n", elapsed\_time);

exit(0);

}

else {

pid2 = fork();

if (pid2 == 0) { // child process 2

// Assign First-Come-First-Serve scheduling procedure

struct sched\_param param2 = {.sched\_priority = 1};

sched\_setscheduler(0, SCHED\_FIFO, &param2);

// Perform some computation

clock\_gettime(CLOCK\_MONOTONIC, &start\_time);

perform\_computation();

clock\_gettime(CLOCK\_MONOTONIC, &end\_time);

elapsed\_time = (end\_time.tv\_sec - start\_time.tv\_sec) +

(end\_time.tv\_nsec - start\_time.tv\_nsec) / 1e9;

printf("Child process 2 execution time: %.6f seconds\n",

elapsed\_time);

exit(0);

}

else { // parent process

// Wait for both child processes to finish

waitpid(pid1, &status, 0);

waitpid(pid2, &status, 0);

}

}

return 0;

}

Алгоритм планирования RR (Round Robin) предполагает, что каждому

процессу выделяется фиксированный квант времени для выполнения. По

истечении этого времени процесс вытесняется, а запуск следующего процесса

планируется. Порядок, в котором процессы ставятся в очередь, определяет порядок

их выполнения.

Изменение порядка очереди в алгоритме RR позволяет изменить порядок

выполнения процессов. Например, можно расставить приоритеты процессов с

более высокими значениями приоритета, поместив их в начало очереди. Таким

образом, эти процессы будут запускаться раньше и иметь больший доступ к

процессорным ресурсам. Также можно использовать очередь FIFO (First-In-First-

Out), при которой процессы выполняются в порядке поступления в очередь. В этом

случае приоритеты игнорируются, и все процессы имеют одинаковый приоритет.

Повторим Эксперимент с большим кол-во процессов.

#include <stdio.h>

#include <sched.h>

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

struct sched\_param shdprm; // значения параметров планирования

int pid,ppid, status;

int n = 60;

int prior = 10;

char \*arr[5] = {"s1", "s2", "s3", "s4", "s5"};

int pid\_son[5];

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 = 0; i < 6; 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);

}

}

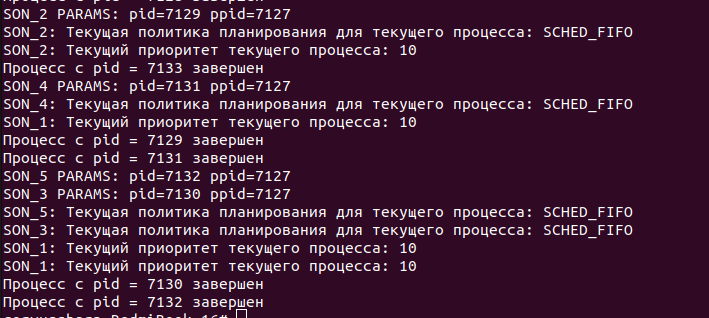
for (int i = 0; i < 6; i++) {

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

}

return 0;

}



10.4. Можно ли задать разные процедуры планирования разным процессам с

одинаковыми приоритетами? Как они будут конкурировать, подтвердите

экспериментально.

Да, можно назначать разные процедуры планирования разным

процессам с одинаковыми приоритетами.

Чтобы экспериментально подтвердить, как разные процедуры планирования

влияют на конкуренцию между процессами с одинаковыми приоритетами, мы

можем создать несколько процессов с одинаковыми уровнями приоритета и

разными процедурами планирования. Затем мы можем запустить эти процессы в

той же системе и измерить время их выполнения.

Вот пример кода, демонстрирующий, как назначить разные процедуры

планирования разным процессам с одинаковыми приоритетами:

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/wait.h>

#include <sched.h>

#include <time.h>

void perform\_computation() {

for (int i = 0; i < 1000000000; ++i)

2 + 2;

}

int main() {

pid\_t pid1, pid2;

int status;

struct timespec start\_time, end\_time;

double elapsed\_time;

// Create two child processes with the same priority level

pid1 = fork();

if (pid1 == 0) { // child process 1

// Assign Round-Robin scheduling procedure

struct sched\_param param1 = {.sched\_priority = 1};

sched\_setscheduler(0, SCHED\_RR, &param1);

// Perform some computation

clock\_gettime(CLOCK\_MONOTONIC, &start\_time);

perform\_computation();

clock\_gettime(CLOCK\_MONOTONIC, &end\_time);

elapsed\_time = (end\_time.tv\_sec - start\_time.tv\_sec) + (end\_time.tv\_nsec -

start\_time.tv\_nsec) / 1e9;

printf("Child process 1 execution time: %.6f seconds\n", elapsed\_time);

exit(0);

}

else {

pid2 = fork();

if (pid2 == 0) { // child process 2

// Assign First-Come-First-Serve scheduling procedure

struct sched\_param param2 = {.sched\_priority = 1};

sched\_setscheduler(0, SCHED\_FIFO, &param2);

// Perform some computation

clock\_gettime(CLOCK\_MONOTONIC, &start\_time);

perform\_computation();

clock\_gettime(CLOCK\_MONOTONIC, &end\_time);

elapsed\_time = (end\_time.tv\_sec - start\_time.tv\_sec) +

(end\_time.tv\_nsec - start\_time.tv\_nsec) / 1e9;

printf("Child process 2 execution time: %.6f seconds\n",

elapsed\_time);

exit(0);

}

else { // parent process

// Wait for both child processes to finish

waitpid(pid1, &status, 0);

waitpid(pid2, &status, 0);

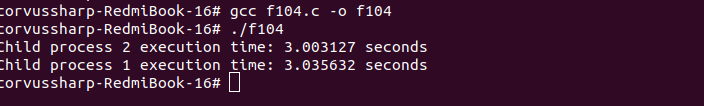
}

}

return 0;

}

вывод:



Так как у разных процедур одинаковый приоритет, время примерно одинаковое.

11. Определим величину кванта. Можно ли ее поменять? – для обоснования проведём эксперимент. 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():

#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);

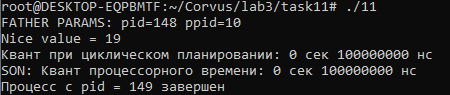
exit(EXIT\_FAILURE);

}

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

return 0;

}



По результату программы видим, что квант изменить нельзя (Ubuntu 20.04 LTS),

но можем узнать его размер.

**Выводы.**

В ходе исследования были рассмотрены различные аспекты управления

процессами и потоками, которые являются ключевыми элементами эффективной

работы операционных систем. Были изучены методы создания процессов и

потоков, а также принципы приоритетов, планирования и синхронизации.

В результате работы было установлено, что правильное управление

процессами и потоками является критически важным для обеспечения

эффективной работы системы. Необходимо правильно настроить приоритеты и

распределение ресурсов между процессами и потоками, чтобы максимизировать

производительность системы и избежать возможных конфликтов.

Также были рассмотрены различные подходы к управлению процессами и

потоками, которые могут быть применены в различных ситуациях и для различных задач. Изучение этих методов может помочь определить оптимальный подход к управлению процессами и потоками в конкретных условиях.

Таким образом, данная работа подчеркивает важность правильного

управления процессами и потоками для обеспечения эффективной работы

операционных систем, а также обозначает необходимость дальнейшего

**Список литературы**

<https://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html>

"Operating System Concepts" Авторы: А. Силбершатц, П. Байер, Г.

Гальвин. Издательство: Wiley.

<https://www.opennet.ru/docs/RUS/linux_parallel/node7.html>

"Synchronization Algorithms and Concurrent Programming" Автор:

Гедеон Липски. Издательство: Addison-Wesley Professional.

•

"Advanced Programming in the UNIX Environment" Автор: Ричард

Стивенс, Стивен Раго. Издательство: Addison-Wesley Professional.

•

"Multithreaded Programming with Java Technology" Автор: Наталья

Комарова, Йонатан Фитушел. Издательство: Prentice Hall.

12. Проанализируем наследование на этапах fork() и exec(). Для этого

проведем 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 (в данном примере это infile.txt и outfile.txt соответственно) Порожденные процессы независимо друг от друга вызывают функции read и write, и в цикле считывают по одному байту информацию из исходного файла и переписывают ее в файл вывода.

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;

// Заблокировать все страницы памяти процесса в оперативной памяти

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

perror("mlockall error");

// Получить идентификатор текущего и родительского процесса

pid = getpid();

ppid = getppid();

// Установить параметры планировщика задач для родительского процесса

shdprm.sched\_priority = 1;

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

perror("SCHED\_SETSCHEDULER\_1");

// Открыть файл для чтения и файла для записи

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

perror("Opening file");

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

perror("Creating file");

// Преобразовать дескрипторы файлов в строковые значения

itoa(str1, fdrd);

itoa(str2,fdwr);

// Создать два процесса-потомка

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

if (fork() == 0) {

// Установить параметры планировщика задач для процессов-потомков

shdprm.sched\_priority = 50;

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

perror("SCHED\_SETSCHEDULER\_1");

// Заменить текущий процесс на новый процесс, запущенный из файла "son" с параметрами-строками

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

}

}

// Закрыть файл для чтения

if (close(fdrd) != 0)

perror("Closing file");

// Дождаться завершения процессов-потомков

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

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

return 0;

}

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[]) {

// Заблокировать все страницы памяти процесса в оперативной памяти

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

perror("mlockall error");

// Объявить переменные

char c;

char buff[3];

int pid, ppid;

int fdrd = atoi(argv[1]); // Преобразовать строковый параметр входного файла в целочисленный дескриптор файла

int fdwr = atoi(argv[2]); // Преобразовать строковый параметр выходного файла в целочисленный дескриптор файла

pid = getpid(); // Получить идентификатор текущего процесса

ppid = getppid(); // Получить идентификатор родительского процесса

printf("son file descriptor = %d\n", fdrd); // Вывести дескриптор входного файла

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

sleep(5); // Задержать выполнение процесса на 5 секунд

// Начать бесконечный цикл

for (;;) {

if (read(fdrd, &c, 1) != 1) // Прочитать один символ из входного файла

return 0;

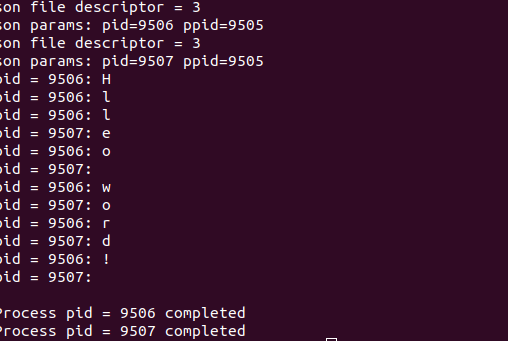
write(fdwr, &c, 1); // Записать один символ в выходной файл

printf("pid = %d: %c\n", pid, c); // Вывести текущий идентификатор процесса и символ

}

}

Вывод про граммы:



Так как 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.

Если сменить ее на FIFO, то результат будет:

Процессы выполняются в порядке очереди.

Если добавим в цикл son.c закрытие файла, то результат будет следующим:

for (;;) {

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

return 0;

write(fdwr, &c, 1);

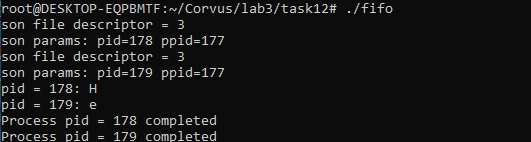
printf("pid = %d: %c\n", pid, c);

if (close(fdrd) != 0) {

perror("Closing file");

}

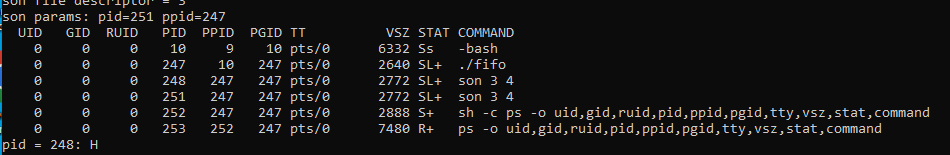
}



При закрытии файла прекращается работа с ним у всех процессов, что подтверждает то, что все процессы работают в одном адресном пространстве.

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, приоритеты и политика планирования процессов.

13)

13 Поставим простой эксперимент: процесс-родитель создает трех

потомков, выполняющихся с различной длительностью по отношению к

породившему их процессу:

а) процесс-родитель запускает дочерний процесс, ожидает и дожидается его

завершения(независимо от длительности выполнения потомка);

б)процесс-родитель запускает дочерний процесс и, не ожидая его завершения, завершается сам;

в) процесс-родитель запускает дочерний процесс и не ожидает его завершения; а процесс-сын завершает свое выполнение до завершения родителя.

**father.c**

#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 \"%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);

return 0;

}

**son.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\

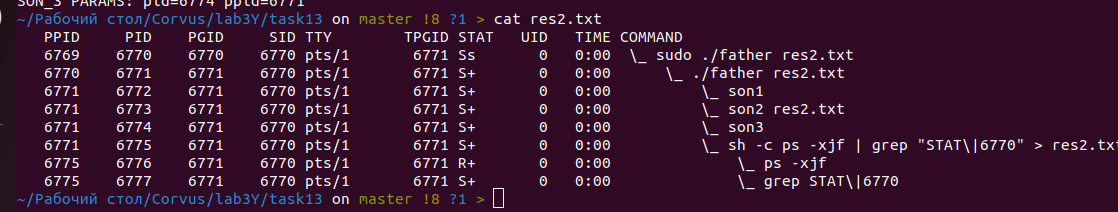
n",pid,ppid);

sleep(3);

}

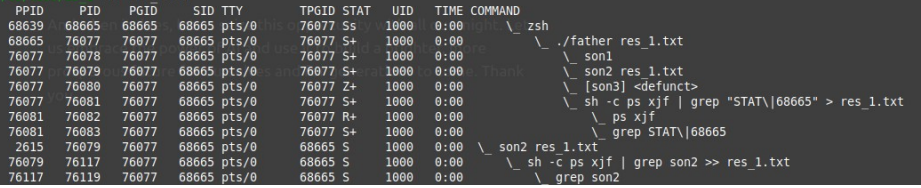
Результат работы программы:

a)



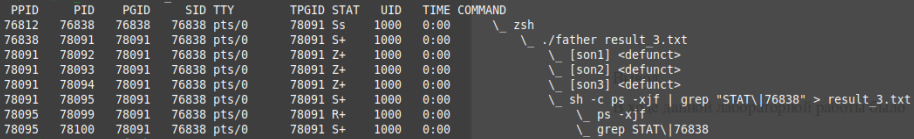
Как видно из результатов, как только процесс-отец завершается, на консоли сразу появляется приглашение на ввод команды. А son2 продолжает свое выполнение в фоновом режиме. Т.к. время выполнения son2 много дольше, то результат выполнения процесса-потомка, появляется уже после приглашения.

б)



Заметим, что процесс-отец запускает процесс-сын и, не ожидая его завершения, завершает свое выполнение. 36

в)



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

13.1.1)

Системный вызов "kill" используется для отправки сигналов указанным процессам. Если не указывать номер сигнала, то будет отправлен сигнал SIGTERM по умолчанию. Идентификатор процесса является аргументом команды. Если он больше нуля, то сигнал отправляется процессу с указанным ID, если он равен нулю, то сигнал отправляется всем процессам, принадлежащим пользователю, если он меньше нуля, то он воспринимается как идентификатор группы процессов, и сигнал отправляется всей группе.

Функция системного вызова "signal" позволяет задать определенные 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), а SIG\_IGN означает, что нужно игнорировать сигнал (за исключением SIGSTOP и SIGKILL).

Для выполнения задания:

a) процесс father порождает процессы son1, son2, son3 и запускает программные коды из соответствующих исполнительных файлов;

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

c) в пользовательских процессах-потомках необходимо обеспечить следующие действия: для son1 – реакцию на сигнал по умолчанию; для son2 – реакцию игнорирования; для son3 – перехватывание и обработку сигнала.

**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);

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");

}

**son1.c:**

#include <sched.h>

#include <sys/mman.h>

#include <fcntl.h>

#include <stdlib.h>

#include <unistd.h>

int main()

{

signal(SIGUSR1, SIG\_DFL);

sleep(5);

}

**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);

}

**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() {

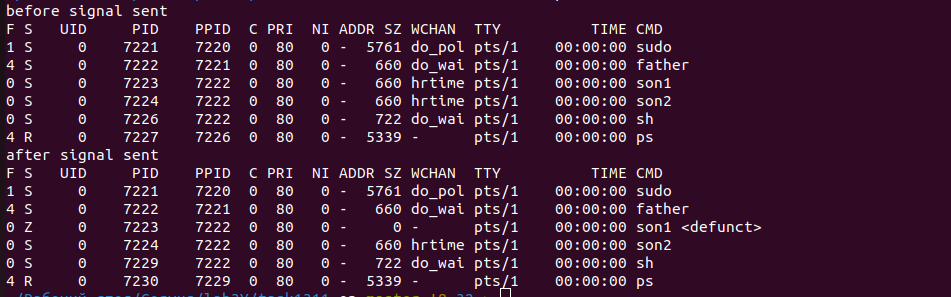
signal(SIGUSR1, SIGUSR1\_handler);

sleep(5);

return 0;

}

**ы**

**Вывод программы:  
**

13.2)

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:

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:

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);

}

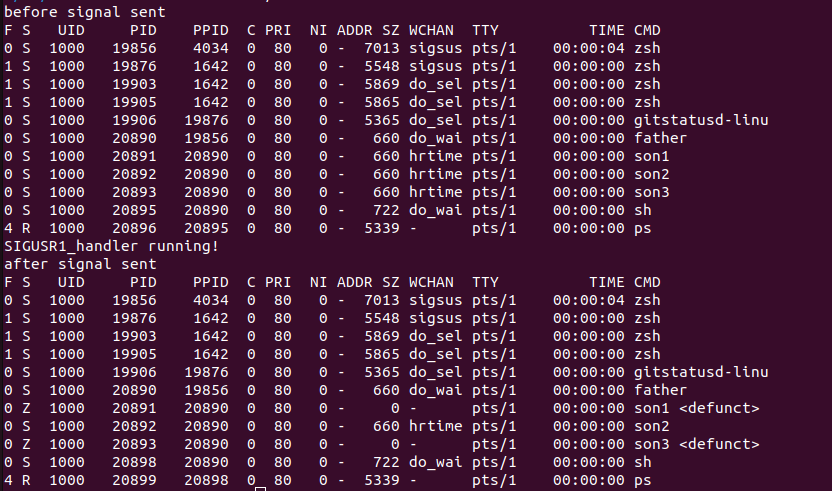
int main() {

signal(SIGUSR1, SIGUSR1\_handler);

printf("son2 running\n");

while (1) {}

return 0;



**Ctrl + c:**

#include <stdio.h>

#include <signal.h> #include <stdio.h> #include <sched.h> #include <sys/mman.h> #include <fcntl.h> #include <stdlib.h> #include <unistd.h>

void handler() {

puts("^C -signal received");

signal(SIGINT, SIG\_DFL);

//восстановление диспозиции по умолчанию

}

￼int main() {

int pid, ppid;

pid = getpid();

ppid = getppid();

printf("Current pid = %d and ppid = %d\n", pid, ppid);

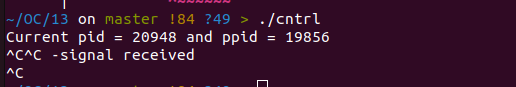
signal(SIGINT, handler);

while (1);

return 0;

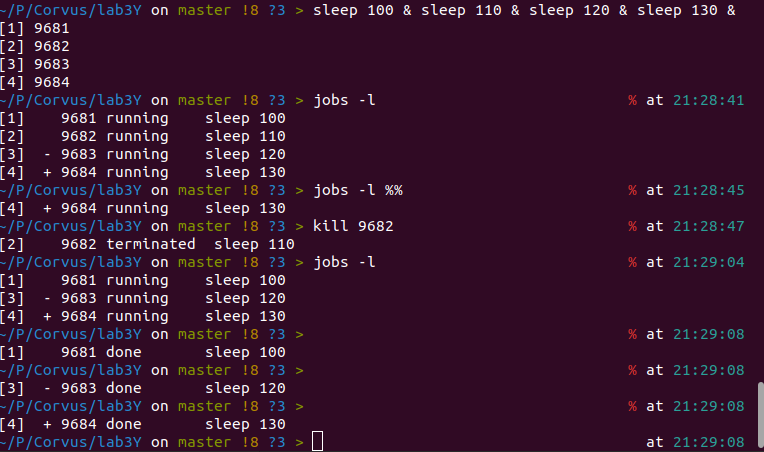
}

Вывод программы:



1. Запущено в фоновом режиме несколько утилит

Использована команда jobs для анализа списка заданий и очередности их выполнения.



С помощью утилиты fg можно повысить приоритет задач.

Благодаря ей, сразу начинает выполняться задача 1, причем не в фоновом режиме. Новые добавленные задачи добавляются в конец очереди и начинают выполняться первыми.

Получено уведомление о завершении одного из заданий с помощью команды notify.

1. **)**

Функция nice позволяет процессу менять свой приоритет. Аргумент команды – величина, которую нужно прибавить к приоритету процесса. В используемой версии QNX эта команда не работает (не меняет приоритета).

Для изменения приоритета процесса можно воспользоваться командой setprio, правда в отличие от команды nice она устанавливает приоритет, а не меняет (прибавляет, отнимает). Для процесса с идентификатором pid устанавливается приоритет prio. Возвращаемое значение – прошлый приоритет.

Функция getprio позволяет узнать приоритет процесса – он будет передан в возвращаемом значении. Аргументом является идентификатор процесса, приоритет которого мы хотим узнать.

#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>

#include <bits/resource.h>

int main() {

int pr, pid;

pid = getpid();

pr = getpriority(PRIO\_PROCESS, pid);

printf("Priority is %d\n", pr);

nice(1000);

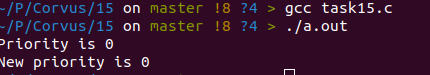
pr = getpriority(PRIO\_PROCESS, pid);

printf("New priority is %d\n", pr);

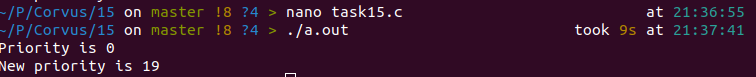
return 0;

}

Вывод без nice():



Вывод с nice():



Границы приоритетов:

#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>

#include <sys/resource.h>

int 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;

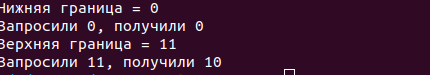
}

}

return 0;

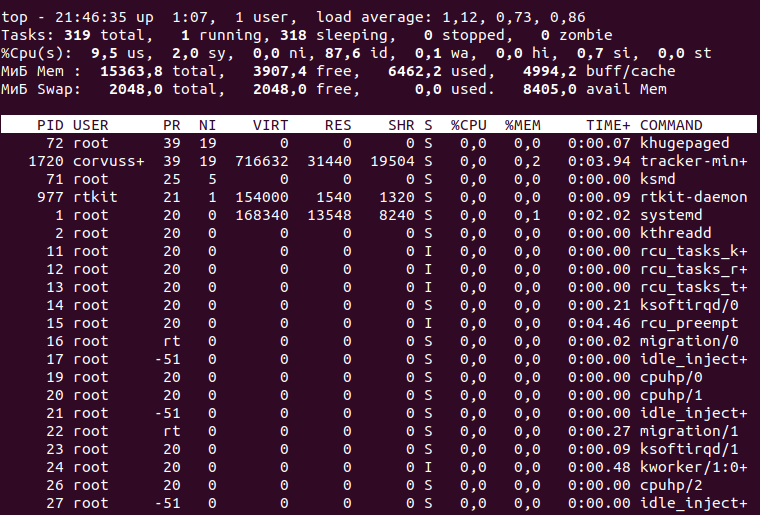
}

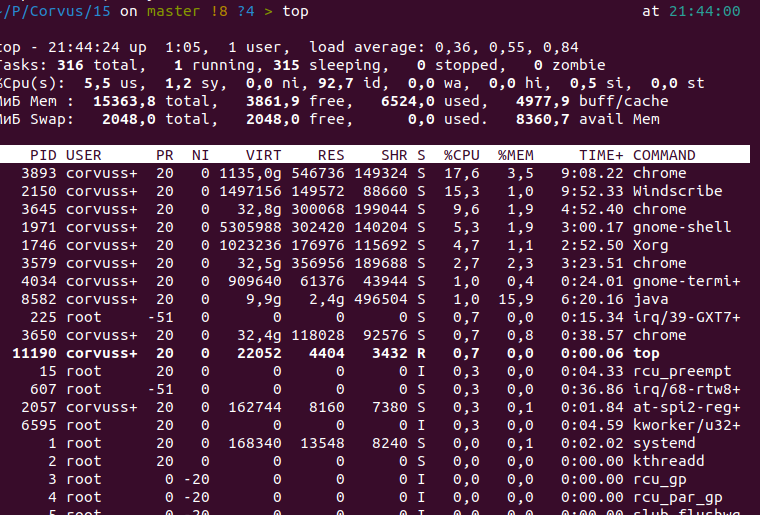
Результат программы:



Утилита top позволяет выводить информацию о системе, а также список процессов динамически обновляя информацию о потребляемых ими ресурсах.

Видно, что пользовательский приоритет для запуска приложений из терминала 19





Системным процессам в операционной системе обычно назначается более высокий приоритет, чем пользовательским процессам. Это связано с тем, что системные процессы отвечают за выполнение критически важных системных функций, таких как управление памятью, операции ввода-вывода и планирование процессов.

Приоритеты в реальном времени часто применяются для процессов, которым требуется немедленное и предсказуемое реагирование. Для таких процессов устанавливается более высокий приоритет, чтобы гарантировать, что они получат достаточные системные ресурсы и могут выполняться вовремя.

Системные процессы обычно принадлежат пользователю root или системному пользователю и имеют имена, которые указывают на их назначение, в то время как пользовательские процессы принадлежат обычным пользователям.

1. **)**

Сигнал SIGHUP используется для уведомления процесса о потере соединения с управляющим терминалом пользователя, например, при выходе пользователя из системы. Этот сигнал может быть перехвачен или проигнорирован программой.

Для игнорирования сигнала SIGHUP можно использовать утилиту nohup. Она не только устанавливает игнорирование сигнала, но также запускает программу в режиме, не зависимом от терминала, и перенаправляет весь вывод в файл nohup.out в текущей директории или в домашнем каталоге пользователя, если файл не может быть создан в текущей директории. После потери связи с терминалом программа продолжит выполняться в фоновом режиме.

Таким образом, использование утилиты nohup может обеспечить надежную и бесперебойную работу программы, даже если пользователь закрыл терминал или вышел из системы.

#include <unistd.h> // для функции sleep()

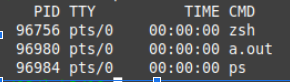
int main() {

sleep(200); // задержка на 200 секунд

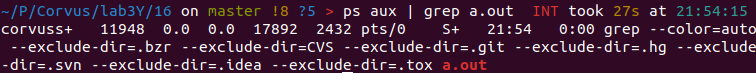
return 0;

}

До выхода из системы:

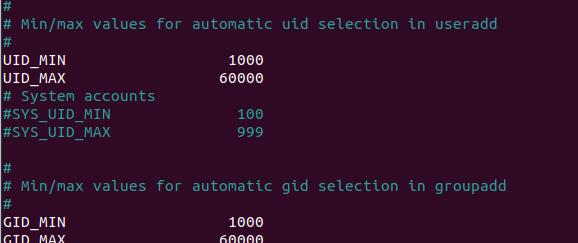


После выхода из системы:



1. **)**

Для определения UID процесса в системе на базе Unix можно использовать команду "ps" вместе с опцией "u". Например, команда "ps uax" отобразит UID всех запущенных процессов. В Linux минимальное значение UID равно 0, которое принадлежит пользователю root, а максимальное значение UID обычно равно 4294967294 (2^32 - 2), которое зарезервировано для пользователя nobody. Однако фактическое максимальное значение UID может быть настроено во время установки дистрибутива Linux или позже системным администратором. Для определения максимального значения UID можно обратиться к файлу "/etc/login.defs" или "/etc/nsswitch.conf".



В операционной системе Linux минимальное значение PID обычно равно 1, и оно зарезервировано для процесса инициализации. Однако максимальное значение PID может различаться в зависимости от конфигурации ядра. В старых версиях Linux максимальное значение PID было ограничено 32 767, но в более новых версиях ядра Linux это значение может быть настроено при компиляции с помощью опции "CONFIG\_PID\_MAX". По умолчанию в большинстве дистрибутивов Linux максимальное значение PID установлено равным 2^22 или 4,194,304, однако оно может быть изменено во время сборки ядра или с использованием параметров ядра "pid\_max".

Стоит отметить, что более высокие значения PID могут влиять на производительность системы и использование ресурсов, поскольку они требуют больше ресурсов памяти и процессора. Кроме того, значения PID могут пересекаться после завершения процесса, что может привести к тому, что два процесса будут иметь одинаковый PID, если они не выполняются одновременно.

Для различия системных и пользовательских процессов можно проверить принадлежность процессов пользователю и группе. Обычно системные процессы принадлежат пользователю root или системному пользователю, в то время как пользовательские процессы принадлежат обычным пользователям.

В системе Linux есть множество системных процессов, каждый из которых имеет свою задачу. Некоторые из них включают:

* init/systemd: первый процесс, запускаемый при загрузке системы на базе Unix. Он отвечает за запуск системных служб и управление системными ресурсами.
* kernel: ядро операционной системы, которое управляет системными ресурсами и предоставляет низкоуровневые службы другим процессам.
* sshd: демон, который обеспечивает безопасный shell (SSH) доступ к системе.
* crond: демон, который выполняет запланированные задания с заданными интервалами.
* syslogd/rsyslogd: Демон системного журнала, который собирает и регистрирует сообщения от различных системных процессов.
* httpd/nginx: демон веб-сервера, который обслуживает HTTP-запросы клиентам.

1. **)**

Подготовлена программа, формирующая несколько нитей. Каждая нить в цикле: выводит на печать собственное имя и инкрементирует

переменную времени, после чего "засыпает" (sleep(5); sleep(1); -для первойи второй нитей соответственно), на экран должно выводиться имя нити и количество пятисекундных (для первой) и секундных (для второй) интервалов функционирования каждой нити.

**18.c:**

Вывод программы:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#define NUM\_THREADS 2

int count1 = 0;

int count2 = 0;

void \*thread\_func(void \*arg) {

int id = \*((int\*) arg);

while (1) {

if (id == 1) {

printf("Thread %d: %d\n", id, count1);

count1++;

sleep(5);

} else {

printf("Thread %d: %d\n", id, count2);

count2++;

sleep(1);

}

}

pthread\_exit(NULL);

}

int main() {

pthread\_t threads[NUM\_THREADS];

int thread\_args[NUM\_THREADS];

int i;

for (i = 0; i < NUM\_THREADS; i++) {

thread\_args[i] = i+1;

if (pthread\_create(&threads[i], NULL, thread\_func, &thread\_args[i])) {

fprintf(stderr, "Error creating thread %d\n", i+1);

exit(1);

}

}

for (i = 0; i < NUM\_THREADS; i++) {

if (pthread\_join(threads[i], NULL)) {

fprintf(stderr, "Error joining thread %d\n", i+1);

exit(1);

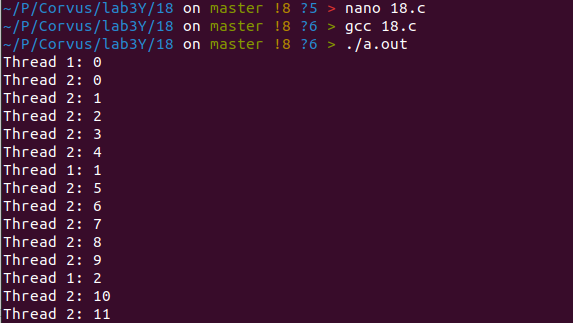
}

}

return 0;

}

вывод:



1. При попытке удаления нити, удаляется процесс в целом, поскольку все нити имеют одинаковый идентификатор.

19.c:

Вывод программы:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#include <signal.h>

#include <sys/syscall.h> // Add the header for SYS\_gettid

#define NUM\_THREADS 2

int count1 = 0;

int count2 = 0;

pthread\_t threads[NUM\_THREADS];

void \*thread\_func(void \*arg) {

int id = \*((int\*) arg);

while (1) {

if (id == 1) {

printf("Thread %d: %d\n", id, count1);

count1++;

sleep(5);

} else {

printf("Thread %d: %d\n", id, count2);

count2++;

sleep(1);

}

}

pthread\_exit(NULL);

}

int main() {

int thread\_args[NUM\_THREADS];

int i;

// Create two child threads

for (i = 0; i < NUM\_THREADS; i++) {

thread\_args[i] = i+1;

if (pthread\_create(&threads[i], NULL, thread\_func, &thread\_args[i])) {

fprintf(stderr, "Error creating thread %d\n", i+1);

exit(1);

}

}

sleep(5); // Wait for threads to run

// Output process table before terminating second thread

printf("Process Table Before Termination:\n");

system("ps -T");

// Terminate second thread using pthread\_cancel

int result = pthread\_cancel(threads[1]);

if (result == 0) {

printf("Second thread terminated successfully.\n");

} else {

printf("Error terminating second thread.\n");

}

sleep(2); // Give time for the second thread to terminate

// Output process table after terminating second thread

printf("\nProcess Table After Termination:\n");

system("ps -T");

// Terminate second thread using pthread\_cancel

int result = pthread\_cancel(threads[1]);

if (result == 0) {

printf("Second thread terminated successfully.\n");

} else {

printf("Error terminating second thread.\n");

}

sleep(2); // Give time for the second thread to terminate

// Output process table after terminating second thread

printf("\nProcess Table After Termination:\n");

system("ps -T");

// Check remaining threads

printf("\nRemaining threads:\n");

for (i = 0; i < NUM\_THREADS; i++) {

pthread\_t tid = threads[i];

printf("Thread %lu: ", tid);

char command[50];

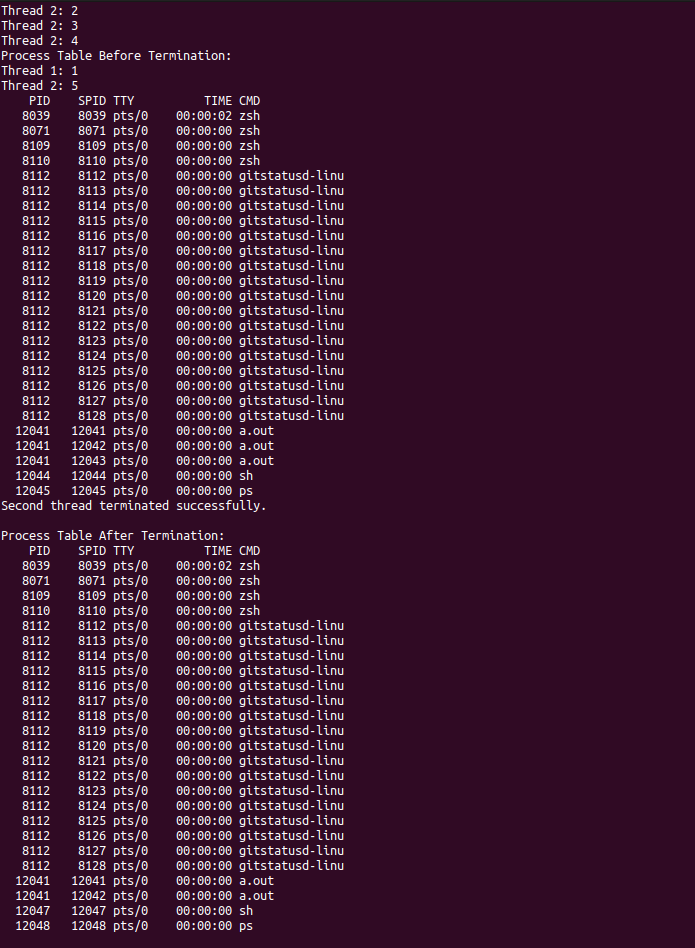
sprintf(command, "ps -T -p $(pidof a.out) | grep %lu", tid);

system(command);

}

return 0;

}



1. **)**

Измененная программа позволяет управлять второй нитью с помощью сигнала SIGUSR1, передаваемого из первой нити. На пятой секунде работы приложения была удалена вторая нить при помощи функции pthread\_kill(threads[1], SIGUSR1). Однако, данная модификация привела к удалению всего процесса вместо удаления только второй нити. Таким образом, можно сделать вывод о том, что указанный способ не позволяет удалить только выбранную нить.

20.c:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#include <signal.h>

#define NUM\_THREADS 2

int count1 = 0;

int count2 = 0;

pthread\_t threads[NUM\_THREADS];

void \*thread\_func(void \*arg) {

int id = \*((int\*) arg);

while (1) {

if (id == 1) {

printf("Thread %d: %d\n", id, count1);

count1++;

sleep(5);

printf("Удаление второй нити.");

pthread\_kill(threads[1], SIGUSR1);

} else {

printf("Thread %d: %d\n", id, count2);

count2++;

sleep(1);

}

}

pthread\_exit(NULL);

}

int main() {

int thread\_args[NUM\_THREADS];

int i;

for (i = 0; i < NUM\_THREADS; i++) {

thread\_args[i] = i+1;

if (pthread\_create(&threads[i], NULL, thread\_func, &thread\_args[i])) {

fprintf(stderr, "Error creating thread %d\n", i+1);

exit(1);

}

}

if (pthread\_join(threads[0], NULL)) {

fprintf(stderr, "Error joining thread 1\n");

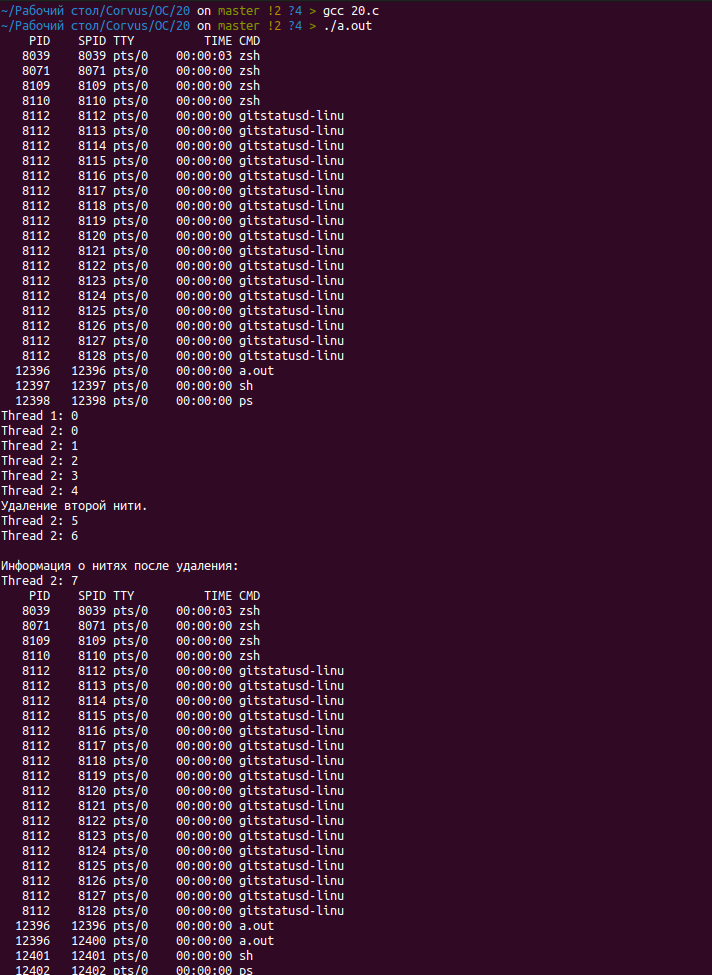
exit(1);

}

return 0;

}

Вывод программы:



С помощью такого управления удалось корректно организовать удаление одной нити: все остальные нити данного процесса процесса сохранились и продолжили выполняться.

Код 21.c:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#include <signal.h>

pthread\_t t1, t2;

void SIGUSR1\_handler(int sig\_no) {

printf("SIGUSR1 received by thread2\n");

pthread\_exit(NULL);

}

void\* thread1(void\* arg) {

int count = 0;

int i;

printf("Thread 1 started\n");

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

count++;

sleep(5);

printf("Thread 1: slept %d times\n", count);

pthread\_kill(t2, SIGUSR1);

}

pthread\_exit(NULL);

}

void\* thread2(void\* arg) {

int count = 0;

int i;

printf("Thread 2 started\n");

system("ps -T");

signal(SIGUSR1, SIGUSR1\_handler);

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

count++;

sleep(1);

printf("Thread 2: slept %d times\n", count);

}

pthread\_exit(NULL);

}

int main() {

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

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

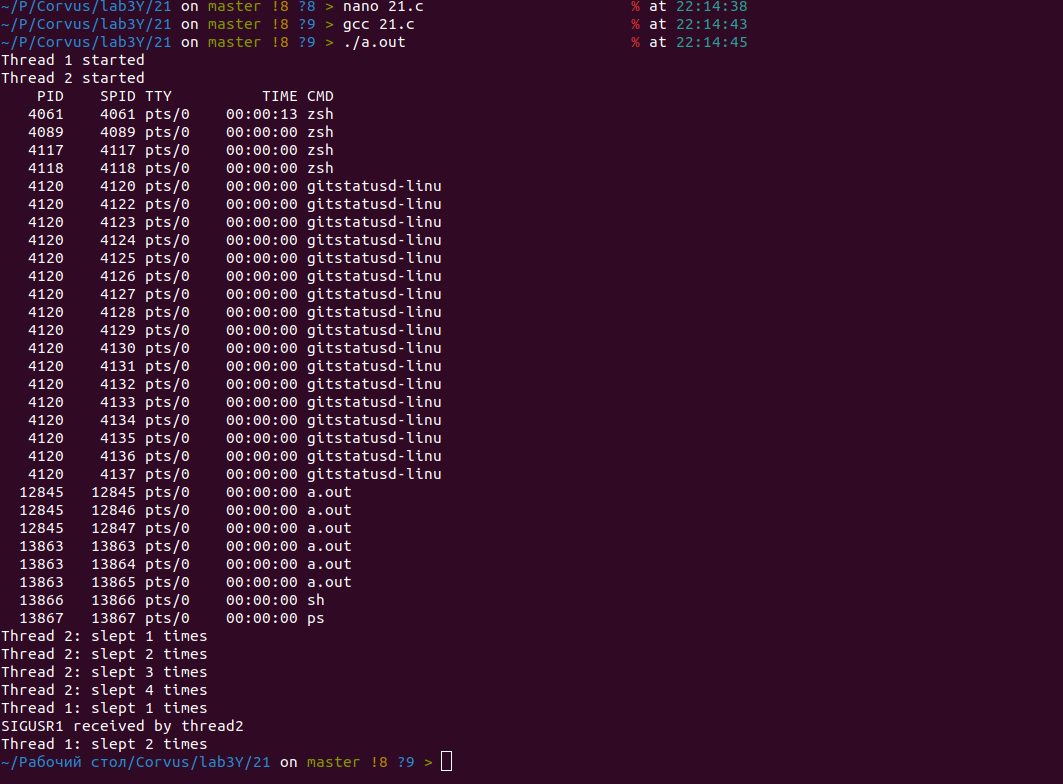
pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

return 0;

}

Вывод программы:



Последняя модификация предполагает создание собственного обработчика сигнала, содержащего уведомление о начале его работы и возврат посредством функции pthread\_exit(NULL);

1. Пример простейшего кода, который позволяет перехватить сигнал, генерирумый в результате нажатия комбинации клавиш (Ctrl+C).

Код ctrl+c.c:

#include <stdio.h>

#include <signal.h>

#include <stdio.h>

#include <sched.h>

#include <sys/mman.h>

#include <fcntl.h>

#include <stdlib.h>

#include <unistd.h>

void handler(){

puts("^C -signal received");

signal(SIGINT, SIG\_DFL);

}

int main(){

int pid, ppid;

pid = getpid();

ppid = getppid();

printf("Current pid = %d and ppid = %d\n", pid, ppid);

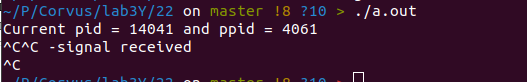
signal(SIGINT, handler);

while(1);

return 0;

}

Вывод программы:



Перехват сигнала ^C осуществляется в программе с помощью обработчика handler, который выводит сообщение о получении сигнала и возвращает управление обработчику SIGINT по умолчанию. Это приводит к принудительному завершению программы при следующем нажатии клавиши ^C. Однако, в данной модификации программы, обработчик перехватывает сигнал многократно с восстановлением исходного обработчика после нескольких раз срабатывания. Таким образом, программа не прерывается при повторном нажатии клавиши ^C, и можно продолжить работу с командной строкой.

1. **)**

Ознакомиться с полным перечнем сигналов можно с помощью команды kill — l в командном интерпретаторетой реализации ОС.

Для ознакомления с полным перечнем сигналов можно воспользоваться командой kill -l в командном интерпретаторе операционной системы.

Некоторые из сигналов базового списка:

1) SIGHUP используется для информирования программы о потере связи с управляющим терминалом или завершении работы процесса-лидера сессии. Многие программы-демоны, которые не имеют лидера сессии, также обрабатывают этот сигнал. Обычно в ответ на получение SIGHUP демон перезапускается. По умолчанию программа, получившая этот сигнал, завершается.

2)SIGINT посылается процессу, если пользователь с консоли отправил команду прервать процесс комбинацией клавиш (Ctrl+C).

3)SIGABRT посылается программе в результате вызова функции abort(3). В результате программа завершается с сохранением на диске образа памяти.

4)SIGKILL завершает работу программы. Программа не может ни обработать, ни игнорировать этот сигнал.

5) SIGSEGV посылается процессу, который пытается обратиться к не принадлежащей ему области памяти. Если обработчик сигнала не установлен, программа завершается с сохранением на диске образа памяти.

6) SIGTERM вызывает "вежливое" завершение программы. Получив этот сигнал, программа может выполнить необходимые перед завершением операции (например, высвободить занятые ресурсы). Получение SIGTERM свидетельствует не об ошибке в программе, а о желании ОС или пользователя завершить ее.

7) SIGCHLD посылается процессу, если его дочерний процесс завершился или был приостановлен. Родительский процесс также получит этот сигнал, если он установил режим отслеживания сигналов дочернего процесса и дочерний процесс получил какой-либо сигнал. По умолчанию сигнал SIGCHLD игнорируется.

8) SIGCONT возобновляет выполнение процесса, остановленного сигналом SIGSTOP.

9)SIGSTOP приостанавливает выполнение процесса. Как и SIGKILL, этот сигнал нельзя перехватить или игнорировать.

10) SIGTSTP приостанавливает процесс по команде пользователя (Ctrl+Z).

1. **)**

24. Проанализируйте процедуру планирования для процессов и потоков одного процесса.

Функции для задания приоритетов нитям и политики планирования:

* *pthread\_attr\_init* - инициализация описателя атрибутов потока управления;
* *pthread\_attr\_setinheritsched* – установка атрибута "наследование стратегии и параметров планирования" потока управления, например, атрибут *PTHREAD\_EXPLICIT\_SCHED*, при котором стратегия планирования и связанные с ней атрибуты должны быть взяты из описателя атрибутов, на который указывает аргумент attr, а не наследоваться от процесса-отца;
* *\_attr\_setschedparam* – установка атрибута "параметры планирования" потока управления, с помощью которых был задан приоритет;
* *pthread\_attr\_setschedpolicy* – установка атрибута "стратегия планирования" потока управления;
* *pthread\_attr\_getschedparam* — получение текущего атрибута "параметры планирования" потока управления;
* *pthread\_attr\_getschedpolicy* – получение текущего атрибута "политика планирования" потока управления;
* *pthread\_attr\_destroy* – удаление описателя атрибутов потока управления.Приоритет и политику планирования для потоков можно задать и другим способом -наследуя от процесса-родителя с помощью функции *pthread\_attr\_setinheritsched (&attr*,*PTHREAD\_INHERIT\_SCHED*) с атрибутом наследования.

Узнаем политику планирования потоков по умолчанию.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <sched.h>

void \*thread\_func(void \*arg) {

int policy;

struct sched\_param param;

pthread\_getschedparam(pthread\_self(), &policy, &param);

printf("Thread policy: %s\n", (policy == SCHED\_FIFO ? "FIFO" :

(policy == SCHED\_RR ? "Round Robin" :

(policy == SCHED\_OTHER ? "Other" :

"Unknown"))));

pthread\_exit(NULL);

}

int main() {

pthread\_t thread\_id;

int ret;

ret = pthread\_create(&thread\_id, NULL, &thread\_func, NULL);

if (ret != 0) {

perror("pthread\_create() error");

exit(EXIT\_FAILURE);

}

pthread\_join(thread\_id, NULL);

return 0;

}



Создадим программу, которая вызывает процесс son1 с разными политиками планирования. Проведём несколько экспериментов по-разному нагружая вычисления в RR и FIFO процессах.

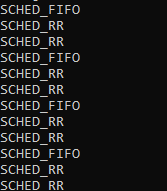
*father.c*

#include <stdio.h>  
#include <stdlib.h>  
#include <sched.h>  
#include <unistd.h>  
  
#define PROC\_AMOUNT 12  
  
*int* main() {  
 *int* i;  
 *char* \*program = "son1";  
 *struct* sched\_param params;  
 params.sched\_priority = PROC\_AMOUNT;  
 *int* sched;  
 *for* (i = 0; i < PROC\_AMOUNT; i++) {  
 pid\_t pid = fork();  
 *if* (pid < 0) {  
 printf("Ошибка при создании дочернего процесса\n");  
 *return* 1;  
 } *else if* (pid == 0) {  
 sched = (i % 2 == 0) ? SCHED\_FIFO : SCHED\_RR;  
 execl(program, program, NULL);  
 printf("Ошибка при вызове функции execl\n");  
 *return* 1;  
 }  
 }  
 *for* (*int* i = 0; i < PROC\_AMOUNT; i++) {  
 wait(NULL);  
 }  
 *return* 0;  
}

*son1.c*

#include <stdio.h>  
#include <stdlib.h>  
#include <sched.h>  
#include <time.h>  
#include <unistd.h>  
  
*int* main() {  
 *struct* sched\_param shdprm; *// Значения параметров планирования  
 int* i, pid, ppid;  
 pid = getpid();  
 *int* k = 0;  
 ppid = getppid();  
 *int* schedule = sched\_getscheduler(0);  
 *switch* (schedule) {  
 printf("SCHED\_FIFO\n");  
 *break*;  
 *case* SCHED\_RR:  
 *for* (*int* i = 0; i < 100000000; i++) {  
 k += 1;  
 }  
 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");  
 }  
 *return* 0;  
}

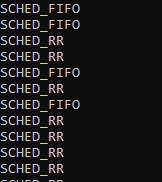
Работа программы, , когда в вызываемом процессе мало вычислений и для FIFO, и для RR:



Результат работы программы, когда в вызываемом процессе мало вычислений для FIFO и много для RR:



Работа программы, , когда в вызываемом процессе много вычислений и для FIFO, и для RR.

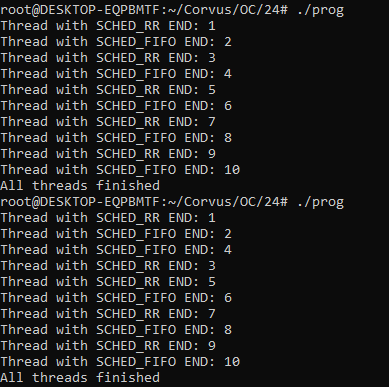


Из проведённых экспериментов сделаем вывод, что при малых вычислениях порядок вычисления процессов примерно равномерный. При сильном утяжелении вычислений для RR-процессов сначала выполняются все FIFO-процессы (так как содержат меньше вычислений, плюс у RR превышается величина кванта). При большом количестве вычислений и в RR, и в FIFO сначала выполняется большинство FIFO-процессов (опять же, в виду превышения кванта у RR).

Исследуем характеристики планирования потоков. Для этого напишем программу, создающую заданное количество потоков (10) с политиками планирования RR и FIFO.

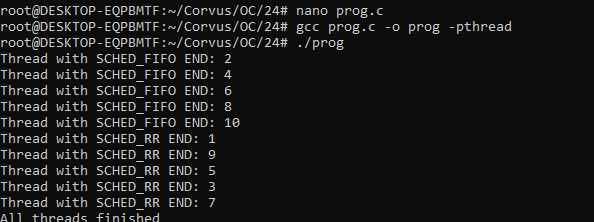
#include <stdio.h>  
#include <stdlib.h>  
#include <pthread.h>  
#include <sched.h>  
#include <errno.h>  
  
*long long int* OPS\_AMOUNT = 100;  
  
*void*\* func\_fifo(*void*\* arg) {  
 *int* marker = \*((*int*\*) arg);  
 *int* dummy = 0;  
 *for* (*int* i = 0; i < OPS\_AMOUNT; i++) {  
 dummy++;  
 }  
 printf("Thread with SCHED\_FIFO END: %d\n", marker);  
 pthread\_exit(NULL);  
}  
  
*void*\* func\_rr(*void*\* arg) {  
 *int* marker = \*((*int*\*) arg);   
 *int* dummy = 0;  
 *for* (*int* i = 0; i < 100; i++) {  
 dummy++;  
 }  
 printf("Thread with SCHED\_RR END: %d\n", marker);  
 pthread\_exit(NULL);  
}  
  
*int* main() {  
 *const int* NUM\_THREADS = 10;  
 pthread\_t threads[NUM\_THREADS];  
 pthread\_attr\_t attrs[NUM\_THREADS];  
 *struct* sched\_param params[NUM\_THREADS];  
 *int* priorityFIFO = 50;  
 *int* priorityRR = 50;  
 *for* (*int* i = 0; i < NUM\_THREADS; i++) {  
 pthread\_attr\_init(&attrs[i]);  
 pthread\_attr\_setinheritsched(&attrs[i], PTHREAD\_EXPLICIT\_SCHED);  
 *if* (i % 2 == 0){  
 params[i].sched\_priority = priorityRR;  
 pthread\_attr\_setschedpolicy(&attrs[i], SCHED\_RR);  
 pthread\_attr\_setschedparam(&attrs[i], &params[i]);  
 pthread\_create(&threads[i], &attrs[i], func\_rr, &i);  
 }  
 *else*{  
 params[i].sched\_priority = priorityFIFO;  
 pthread\_attr\_setschedpolicy(&attrs[i], SCHED\_FIFO);  
 pthread\_attr\_setschedparam(&attrs[i], &params[i]);  
 pthread\_create(&threads[i], &attrs[i], func\_fifo, &i);  
 }  
 *//pthread\_attr\_setschedpolicy(&attrs[i], SCHED\_RR);  
 //pthread\_attr\_setschedparam(&attrs[i], &params[i]);* }  
 *for* (*int* i = 0; i < NUM\_THREADS; i++) {  
 pthread\_join(threads[i], NULL);  
 }  
  
 printf("All threads finished\n");  
  
 *return* 0;  
}

Результат работы программы при одинаковом количестве вычислений:

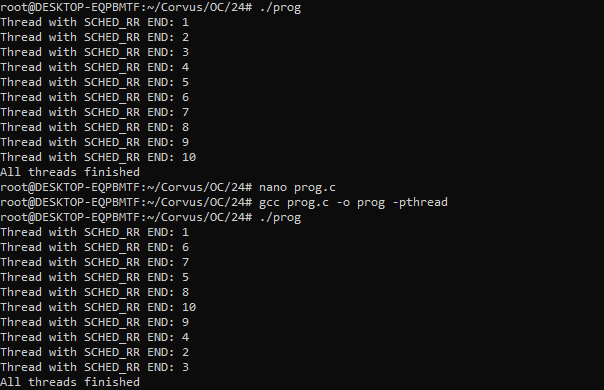


Наблюдаем разный порядок выполнения по причине одинаково малого количества вычислений.

Поменяем количество вычислений для SCHED\_FIFO (сильно их уменьшим). Результат ожидаем, сначала выполнились все вычисления SCHED\_FIFO (т. к. требуют меньше ресурсов), а потом уже – SCHED\_RR.



Модифицируем программу так, чтобы все потоки были RR, и протестируем её запуск с разным кол-вом вычислений.



При первом запуске вычислений было мало(10), а при втором много (100 млн.). Как видно, при большом количестве вычислений величина кванта превышается, потоки передают управление друг другу, следовательно порядок завершение произволен. При малом количестве вычислений потоки завершаются друг за другом.

Теперь исследуем наследование потоком политики планирования от процесса. Напишем программу, где процесс порождает поток и в зависимости от параметра команды *pthread\_attr\_setinheritsched() inheritsched.*

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

int i = 0;

void switch\_policy(int policy) {

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;

}

}

void\* thread\_func(void\* arg) {

printf("Child thread started.\n");

/\* Do some work here \*/

printf("Thread's ");

switch\_policy(sched\_getscheduler(0));

while (i < 10) {

i++;

}

printf("Child thread finished.\n");

pthread\_exit(NULL);

}

int main() {

pthread\_t child\_thread;

pthread\_attr\_t attr;

struct sched\_param shdprm;

int ret;

pid\_t pid = getpid();

shdprm.sched\_priority = 1;

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

perror("SCHED\_SETSCHEDULER");

}

printf("Политика процесса: ");

switch\_policy(sched\_getscheduler(pid));

/\* Initialize thread attributes \*/

ret = pthread\_attr\_init(&attr);

if (ret != 0) {

fprintf(stderr, "pthread\_attr\_init() failed.\n");

exit(EXIT\_FAILURE);

}

/\* Set thread scheduling policy to SCHED\_FIFO \*/

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

if (ret != 0) {

fprintf(stderr, "pthread\_attr\_setschedpolicy() failed.\n");

exit(EXIT\_FAILURE);

}

/\* Create child thread with inherited scheduling \*/

ret = pthread\_create(&child\_thread, &attr, thread\_func, NULL);

if (ret != 0) {

fprintf(stderr, "pthread\_create() failed.\n");

exit(EXIT\_FAILURE);

}

/\* Wait for child thread to finish \*/

pthread\_join(child\_thread, NULL);

/\* Cleanup thread attributes \*/

pthread\_attr\_destroy(&attr);

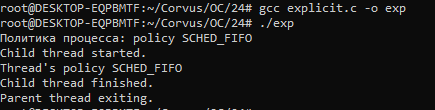
printf("Parent thread exiting.\n");

return 0;

}

p

Результат работы программы при значении *PTHREAD\_EXPLICIT\_SCHED:*



Политика планирования не наследуется, так как задаётся отдельно.

Политика планирования наследуется.

Теперь напишем программу, где один поток порождает другой и исследуем наследование политики планирования:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#include <signal.h>

void \*thread\_function1(void \*arg);

void \*thread\_function2(void \*arg);

void sigusr1\_handler(int signal\_number);

pthread\_t thread2;

void switch\_policy(int policy)

{

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;

}

}

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

{

pthread\_t thread1;

int result1;

pthread\_attr\_t attr1;

// Установка обработчика сигнала SIGUSR1

signal(SIGUSR1, sigusr1\_handler);

pthread\_attr\_init(&attr1);

pthread\_attr\_setinheritsched(&attr1, PTHREAD\_EXPLICIT\_SCHED);

pthread\_attr\_setschedpolicy(&attr1, SCHED\_FIFO);

// Установка параметров планирования для первого потока

struct sched\_param shdprm;

shdprm.sched\_priority = 1;

pthread\_attr\_setschedparam(&attr1, &shdprm);

// Создание первого потока

result1 = pthread\_create(&thread1, &attr1, thread\_function1, NULL);

// Ожидание завершения первого потока

pthread\_join(thread1, NULL);

printf("Главный поток завершен\n");

exit(EXIT\_SUCCESS);

}

void \*thread\_function1(void \*arg)

{

printf("Поток 1 создан\n");

printf("Политика потока 1: ");

switch\_policy(sched\_getscheduler(0));

//printf("Приоритет потока 1: %d", );

pthread\_attr\_t attr2;

pthread\_attr\_init(&attr2);

pthread\_attr\_setinheritsched(&attr2, PTHREAD\_EXPLICIT\_SCHED);

//pthread\_attr\_setinheritsched(&attr2, PTHREAD\_INHERIT\_SCHED);

pthread\_attr\_setschedpolicy(&attr2, SCHED\_RR);

// Установка параметров планирования для второго потока

struct sched\_param param;

param.sched\_priority = 2;

pthread\_attr\_setschedparam(&attr2, &param);

pthread\_create(&thread2, &attr2, thread\_function2, NULL);

// Ждем 5 секунд и затем останавливаем второй поток

sleep(5);

pthread\_kill(thread2, SIGUSR1);

// Ожидание завершения второго потока

pthread\_join(thread2, NULL);

printf("Поток 1 завершен\n");

pthread\_attr\_destroy(&attr2);

return NULL;

}

void \*thread\_function2(void \*arg)

{

printf("Поток 2 создан\n");

printf("Политика потока 2: ");

switch\_policy(sched\_getscheduler(0));

while (1)

{

sleep(1);

}

return NULL;

}

void sigusr1\_handler(int signal\_number)

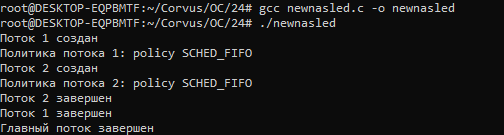
{

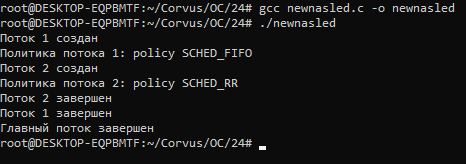
printf("Поток 2 завершен\n");

pthread\_exit(NULL);

}

При тех же значениях видим аналогичный наследованию от процесса результат:





Напишем программы, задающие приоритет извне и внутри программы для двух разных потоков:

#include <pthread.h>

#include <sched.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

void \*thread\_function(void \*arg) {

int prio = \*((int\*) arg);

printf("Thread priority: %d\n", prio);

return NULL;

}

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

pthread\_t thread1, thread2;

pthread\_attr\_t attr;

struct sched\_param param;

if (argc < 2) {

printf("Usage: %s <priority1> <priority2>\n", argv[0]);

return 1;

}

int priority1 = atoi(argv[1]);

int priority2 = atoi(argv[2]);

// Set the scheduling policy and priority for the newly created pthread

pthread\_attr\_init(&attr);

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

param.sched\_priority = priority1;

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

pthread\_attr\_init(&attr);

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

param.sched\_priority = priority2;

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

// Create the pthread with the specified SCHED\_FIFO policy and priority

pthread\_create(&thread1, &attr, thread\_function, &priority1);

pthread\_create(&thread2, &attr, thread\_function, &priority2);

printf("Main started\n");

// Now join the pthread

pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

printf("Main end\n");

return 0;

}

#include <pthread.h>

#include <stdio.h>

void\* thread\_func(void\* arg) {

int prio = \*((int\*) arg);

printf("Thread priority: %d\n", prio);

return NULL;

}

int main(void) {

pthread\_t thread1, thread2;

int prio1 = 1, prio2 = 99;

pthread\_create(&thread1, NULL, thread\_func, &prio1);

pthread\_create(&thread2, NULL, thread\_func, &prio2);

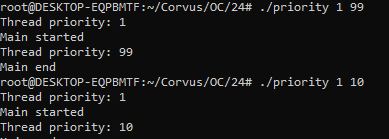
pthread\_join(thread1, NULL);

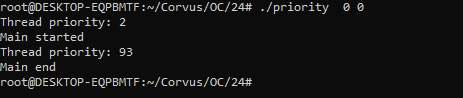
pthread\_join(thread2, NULL);

return 0;

}

Результаты работы программ:





Видим, что приоритет можно задать обоими способами.

**Выводы.**

В ходе исследования были рассмотрены различные аспекты управления процессами и потоками, которые являются ключевыми элементами эффективной работы операционных систем. Были изучены методы создания процессов и потоков, а также принципы приоритетов, планирования и синхронизации.

В результате работы было установлено, что правильное управление процессами и потоками является критически важным для обеспечения эффективной работы системы. Необходимо правильно настроить приоритеты и распределение ресурсов между процессами и потоками, чтобы максимизировать производительность системы и избежать возможных конфликтов.

Также были рассмотрены различные подходы к управлению процессами и потоками, которые могут быть применены в различных ситуациях и для различных задач. Изучение этих методов может помочь определить оптимальный подход к управлению процессами и потоками в конкретных условиях.

Таким образом, данная работа подчеркивает важность правильного управления процессами и потоками для обеспечения эффективной работы операционных систем, а также обозначает необходимость дальнейшего исследования этой темы для оптимизации производительности системы в различных сценариях.

**СПИСОК ЛИТУРАТУРЫ**

*Geeks for Geeks – https://www.geeksforgeeks.org/*

*Linux.org – https://www.linux.org.ru/forum/general/*

*Linux Documentation – linux.die.net/man*

# Multithreading in C *https://www.geeksforgeeks.org/multithreading-in-c/*