**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №3-4**

**по дисциплине «Операционные системы и среды»**

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

|  |  |  |
| --- | --- | --- |
| Студент гр. 1384 |  | Усачева Д.В. |
| Преподаватель |  | Душутина Е. В. |

Санкт-Петербург

2023

## Цель работы.

Целью данной работы является изучение основных принципов управления процессами и потоками в операционных системах.

## 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.

## Выполнение работы

Задание 1.

Была создана программа на основе одного исходного (а затем исполняемого) файла с псевдораспараллеливанием вычислений посредством порождения процесса-потомка.



Рисунок 1 – Код программы для задания 1 (взят из методических указаний).

Задание 2.

Были выполнены однократные вычисления в каждом процессе. Каждый процесс имеет вывод на терминал, идентифицирующий текущий процесс. Последняя исполняемая команда функции main выводит на терминал сообщение о завершении программы.

Листинг 1 – выполнение задания 2:

gcc 1.c && ./a.out

Текущее число:1

parent pid = 5341, ppid =3318

Текущее число:12

Завершение процесса

Текущее число:1

child pid = 5342, ppid =831

Текущее число:5

Завершение процесса

Можно заметить, что на первом этапе выполняются вычисления для первого (родительского процесса), а затем для второго (дочернем).

Задание 3.

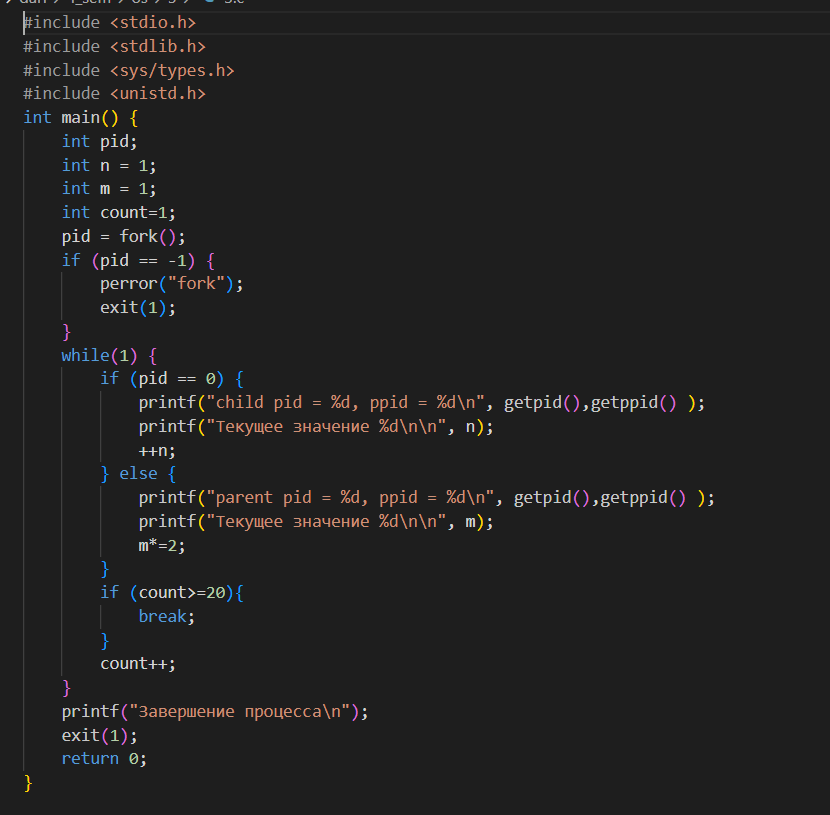


Рисунок 2 – Код программы для задания 3

(частично взят из методических указаний)

Листинг 2 – выполнение задания 3:

parent pid = 258, ppid = 9

Текущее значение 1

parent pid = 258, ppid = 9

Текущее значение 2

parent pid = 258, ppid = 9

child pid = 259, ppid = 258

Текущее значение 4

parent pid = 258, ppid = 9

Текущее значение 1

Текущее значение 8

parent pid = 258, ppid = 9

child pid = 259, ppid = 258

Текущее значение 16

Текущее значение 2

parent pid = 258, ppid = 9

Текущее значение 32

child pid = 259, ppid = 258

parent pid = 258, ppid = 9

Текущее значение 3

Текущее значение 64

child pid = 259, ppid = 258

parent pid = 258, ppid = 9

Текущее значение 4

Текущее значение 128

child pid = 259, ppid = 258

parent pid = 258, ppid = 9

Текущее значение 5

Текущее значение 256

child pid = 259, ppid = 258

parent pid = 258, ppid = 9

Текущее значение 6

Текущее значение 512

child pid = 259, ppid = 258

parent pid = 258, ppid = 9

Текущее значение 7

Текущее значение 1024

child pid = 259, ppid = 258

parent pid = 258, ppid = 9

Текущее значение 8

Текущее значение 2048

child pid = 259, ppid = 258

parent pid = 258, ppid = 9

Текущее значение 9

Текущее значение 4096

child pid = 259, ppid = 258

Текущее значение 10

child pid = 259, ppid = 258

parent pid = 258, ppid = 9

Текущее значение 11

Текущее значение 8192

child pid = 259, ppid = 258

parent pid = 258, ppid = 9

Текущее значение 12

Текущее значение 16384

child pid = 259, ppid = 258

parent pid = 258, ppid = 9

Текущее значение 13

Текущее значение 32768

parent pid = 258, ppid = 9

child pid = 259, ppid = 258

Текущее значение 65536

Текущее значение 14

parent pid = 258, ppid = 9

Текущее значение 131072

child pid = 259, ppid = 258

parent pid = 258, ppid = 9

Текущее значение 15

Текущее значение 262144

child pid = 259, ppid = 258

parent pid = 258, ppid = 9

Текущее значение 16

Текущее значение 524288

child pid = 259, ppid = 258

Завершение процесса

Текущее значение 17

child pid = 259, ppid = 258

Текущее значение 18

child pid = 259, ppid = 258

Текущее значение 19

child pid = 259, ppid = 258

Текущее значение 20

Завершение процесса

Здесь можно заметить, что, если однократные вычисления в этом коде заменить на циклическое исполнение, то можно будет наблюдать конкуренцию процессов за процессорный ресурс.

Задание 4.

Листинг 3 – выполнение задания 4:

pid=3498

родитель:4000

pid=0

Программа завершена

потомок:1001Программа завершена

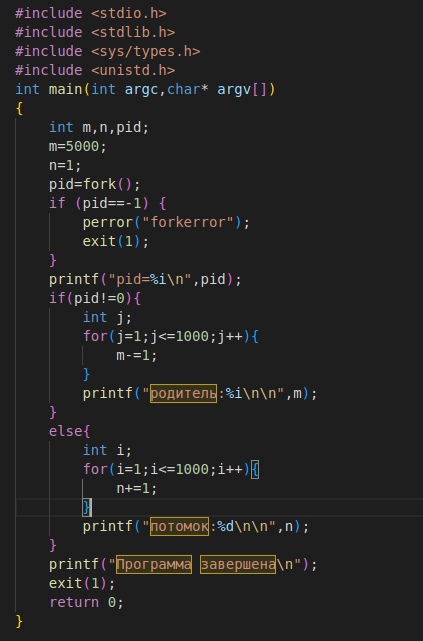


Рисунок 3 – Код программы для задания 4 (взят из методических указаний).

Программа выполняет вызов fork(), выводит идентификатор выполняющегося процесса. В случае, если текущий процесс – потомок (pid =0), то производится 1000 операций вычитания 1 из n, иначе (текущий процесс–родитель) – 1000 операций прибавления 1 к числу m. После завершения вычислений, выводится результат и сообщение о завершении вычислений.

Задание 5.

Разработаны 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 -xf > file.txt").

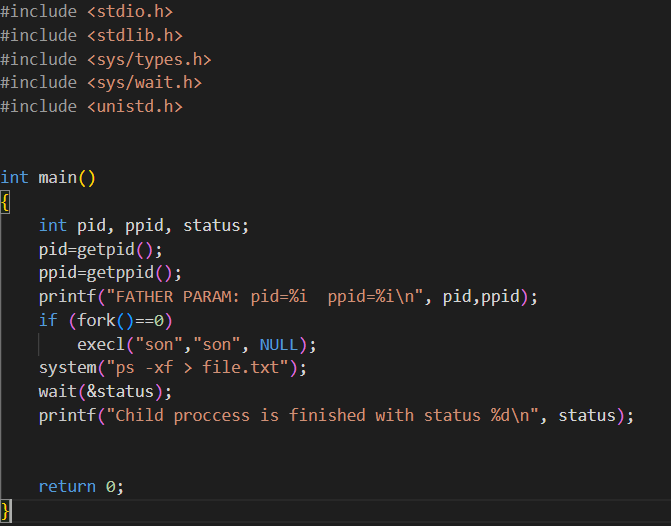


Рисунок 4 – Код программы father.c

(взят из методических указаний).

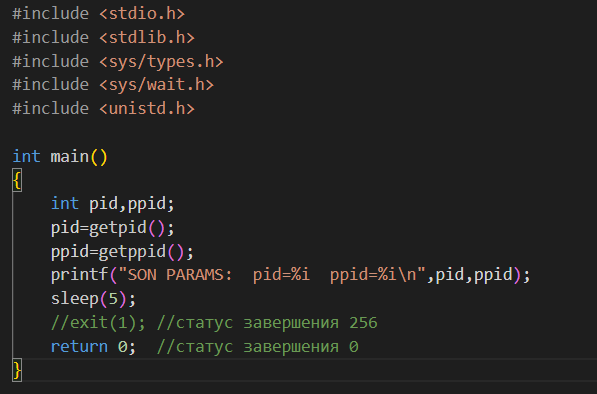


Рисунок 5 – Код программы son.c.

(взят из методических указаний).

Задание 6.

Запустим на выполнение программу father.out.

Листинг 4– Вывод программы для задания 6:

$ ./father

FATHER PARAM: pid=280 ppid=9

SON PARAMS: pid=281 ppid=280

Child proccess is finished with status 0

Получим информацию о процессах из пункта 5, запущенных с терминала. Для этого посмотрим состояние нужной части таблицы из файла file:

Листинг 5– таблица процессов для задания 6:

9 pts/0 Ss 0:00 -bash

280 pts/0 S+ 0:00 \\_ ./father

281 pts/0 S+ 0:00 \\_ son

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

283 pts/0 R+ 0:00 \\_ ps –xf

PID — идентификатор процесс

TTY — терминал, с которым связан данный процес

STAT — состояние, в котором на данный момент находится процесс-

родитель

TIME — процессорное время, занятое этим процессом

COMMAND — команда, запустившая данный процесс-отец

Состояния STAT, представленные выше:

S: процесс ожидает (т.е. спит менее 20 секунд)

s: лидер сессии

R: процесс выполняется в данный момент

+: порожденный процесс

Задание 7.

Выполнена программа father.out в фоновом режиме father &. Была получена таблица процессов, запущенных с терминала (включая отцовский и сыновний процессы):

Листинг 6– таблица процессов для задания 7:

PID TTY STAT TIME COMMAND

9 pts/0 Ss+ 0:00 -bash

318 pts/0 S 0:00 \\_ ./father

319 pts/0 S 0:00 \\_ son

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

321 pts/0 R 0:00 \\_ ps –xf

Командный интерпретатор (в данном случае bash) запускает программу ./father, «распараллеливает» процессы и порождает son. Программа запускается в фоновом режиме, а параллельно ей - команда ps -xf.

Задание 8.

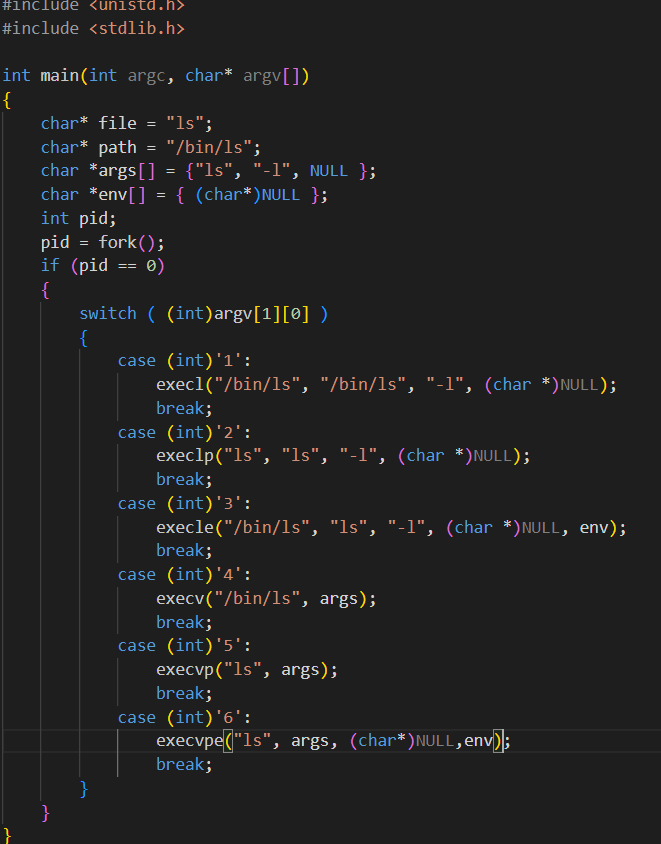


Рисунок 6 – Код программы для выполнения задания 8

(взят из методических указаний).

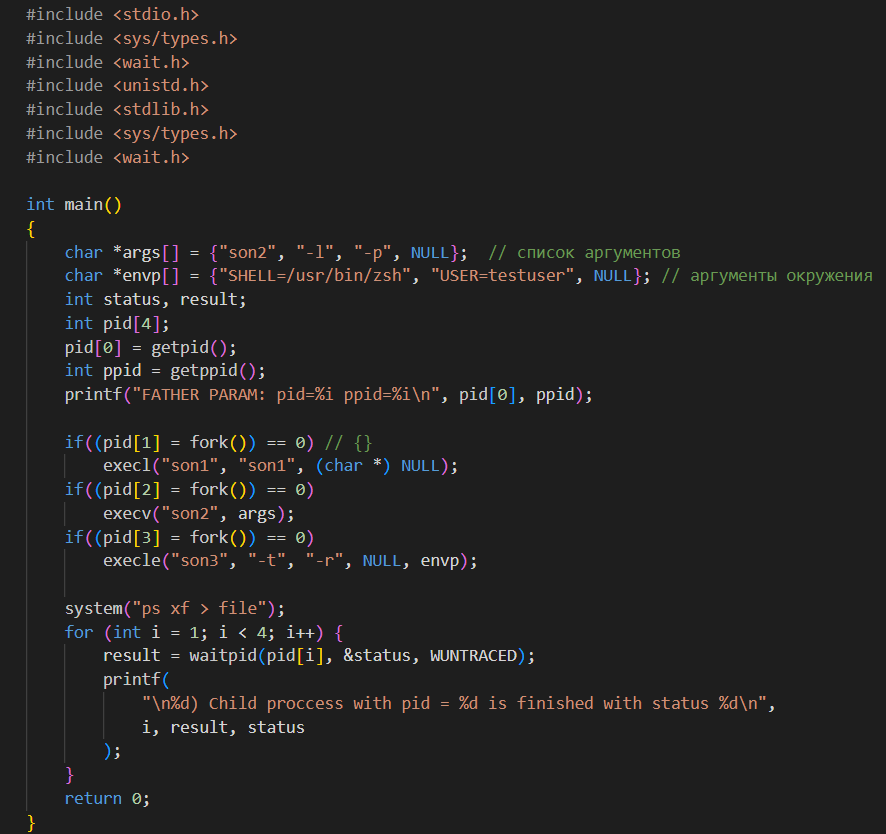


Рисунок 6.2 – Код программы father.c для выполнения задания 8

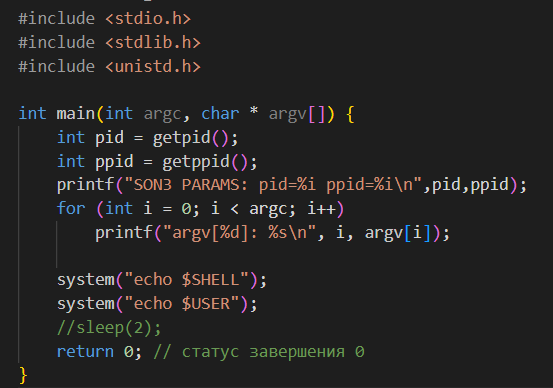


Рисунок 6.3 – Код программы son3.c для выполнения задания 8

Листинг 7– выполнение программы для задания 8:

./8 1

dari@DESKTOP-7CDTIKD:~/4\_sem/os/3$ total 76

-rwxr-xr-x 1 dari dari 16952 Apr 25 12:48 1

-rw-r--r-- 1 dari dari 566 Apr 25 12:43 1.c

-rwxr-xr-x 1 dari dari 16952 May 1 16:54 3

-rw-r--r-- 1 dari dari 779 Apr 25 15:33 3.c

drwxr-xr-x 2 dari dari 4096 May 1 16:42 5

-rwxr-xr-x 1 dari dari 17008 May 1 17:09 8

-rw-r--r-- 1 dari dari 940 May 1 17:08 8.c

./8 3

dari@DESKTOP-7CDTIKD:~/4\_sem/os/3$ total 76

-rwxr-xr-x 1 dari dari 16952 Apr 25 12:48 1

-rw-r--r-- 1 dari dari 566 Apr 25 12:43 1.c

-rwxr-xr-x 1 dari dari 16952 May 1 16:54 3

-rw-r--r-- 1 dari dari 779 Apr 25 15:33 3.c

drwxr-xr-x 2 dari dari 4096 May 1 16:42 5

-rwxr-xr-x 1 dari dari 17008 May 1 17:09 8

-rw-r--r-- 1 dari dari 940 May 1 17:08 8.c

1. execl() - запускает новую программу с заданным именем и аргументами, переданными в виде списка строк

2. execlp() - запускает новую программу с заданным именем и аргументами, переданными в виде списка строк, но ищет исполняемый файл в пути поиска.

3. execvp() - запускает новую программу с заданным именем и аргументами, переданными в виде массива строк, но ищет исполняемый файл в пути поиска.

4. execle() - принимает имя исполняемого файла и список аргументов в виде отдельных строк, но дополнительно позволяет указать массив переменных окружения. После списка аргументов в функцию передается массив переменных окружения, который заканчивается NULL.

5. execv() - запускает новую программу с заданным именем и аргументами, переданными в виде массива строк.

6. execvpe() - работает аналогично execvp(), но дополнительно позволяет указать массив переменных окружения. После списка аргументов в функцию передается массив переменных окружения, который заканчивается NULL.

Задание 9.

Рассмотрим wait(&status) При успешном завершении системного вызова fork процессы порождающий и порожденный равноправно сосуществуют в системе. Они выполняются, разделяя процессорное время, конкурируя за ресурсы на основе своих приоритетов. Выполнение порождающего процесса может быть приостановлено до завершения потомка системным вызовом wait. Системный вызов wait возвращает родителю идентификатор того потомка, который завершился первым после последнего обращения к wait. Если у родителя несколько потомков, то чтобы узнать о завершении каждого из них, нужно выполнить несколько системных вызовов wait с проверкой их возвращаемых значений. Если процесс не имеет потомков, wait возвращает код (-1).

Проведем эксперимент, позволяющий родителю отслеживать подмножество порожденных потомков, используя различные функции семейства wait().

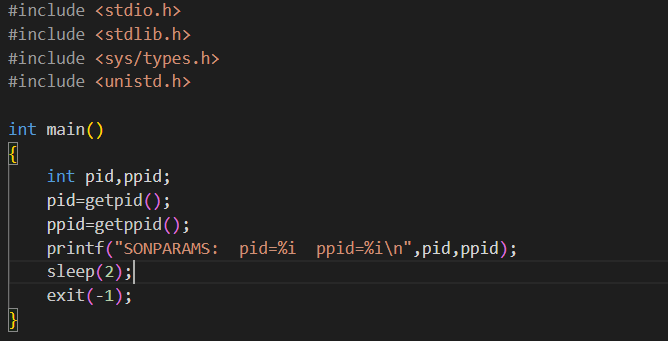


Рисунок 7 – Код программы son1.c для выполнения задания 9

(взят из методических указаний).



Рисунок 8 – Код программы son2.c для выполнения задания 9

(взят из методических указаний).

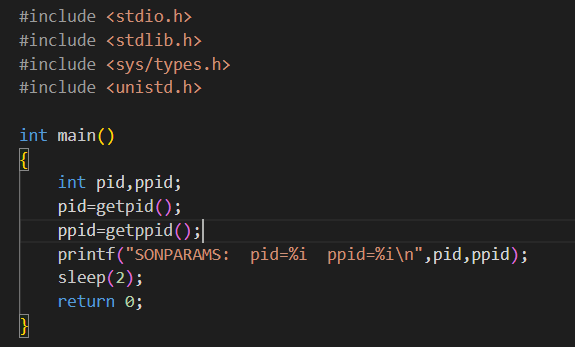


Рисунок 9 – Код программы son3.c для выполнения задания 9

(взят из методических указаний).

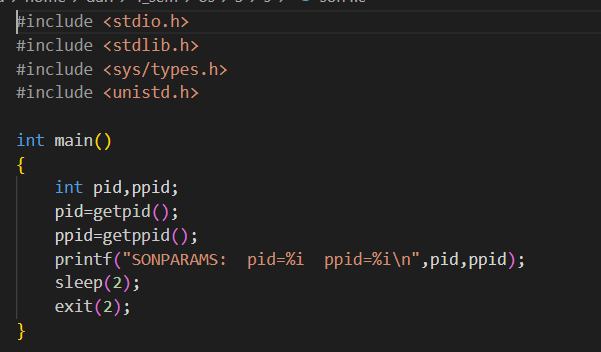


Рисунок 10 – Код программы son4.c для выполнения задания 9

(взят из методических указаний).

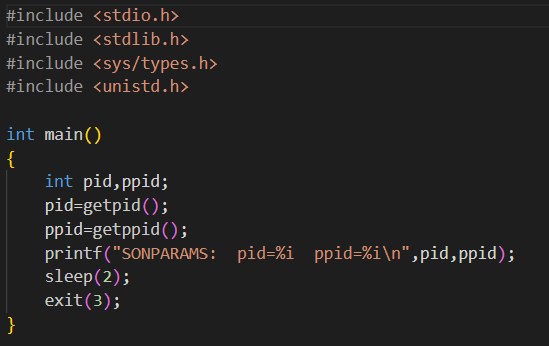


Рисунок 11 – Код программы son5.c для выполнения задания 9

(взят из методических указаний).

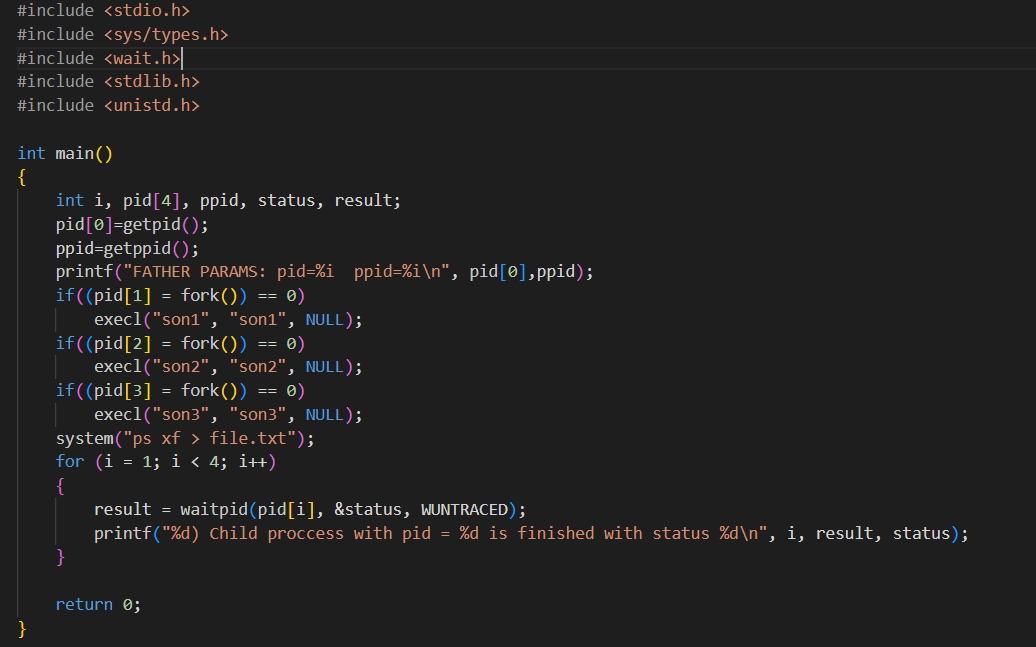


Рисунок 12 – Код программы father1.c для выполнения задания 9

(взят из методических указаний).

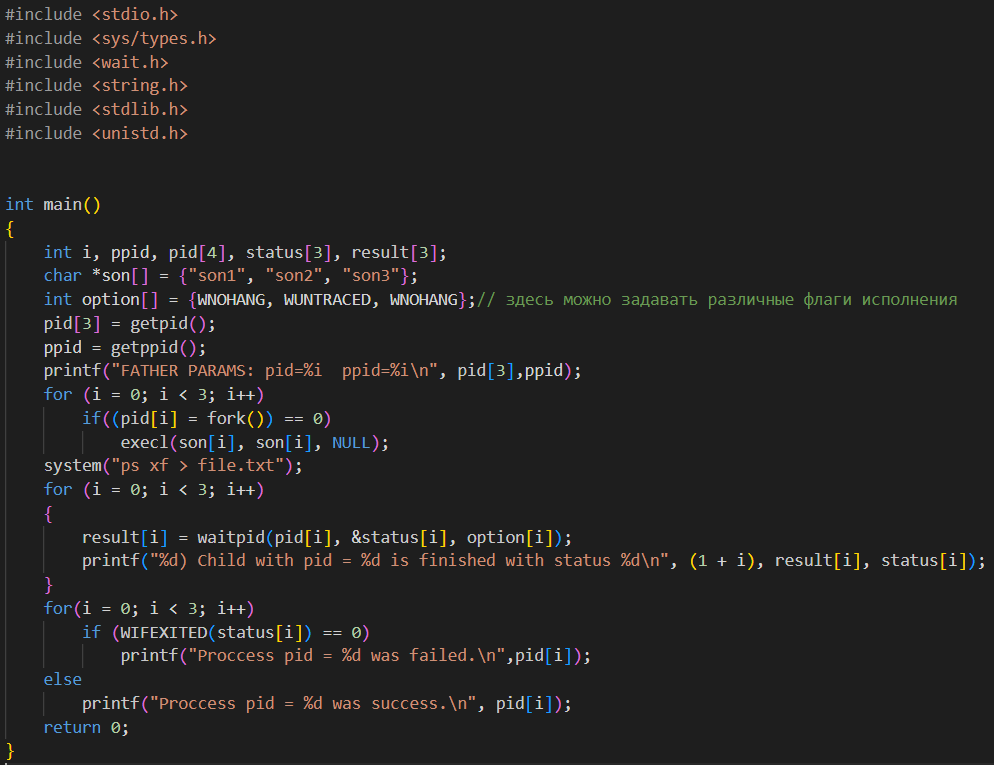


Рисунок 13 – Код программы father2.c для выполнения задания 9

(взят из методических указаний).

Листинг 8– выполнение программы father2.с(**WNOHANG, WUNTRACED, WNOHANG**):

FATHER PARAMS: pid=428 ppid=9

SONPARAMS: pid=429 ppid=428

SONPARAMS: pid=430 ppid=428

SONPARAMS: pid=431 ppid=428

SONPARAMS: pid=432 ppid=428

SONPARAMS: pid=433 ppid=428

1) Child with pid = 0 is finished with status 832

2) Child with pid = 430 is finished with status 256

3) Child with pid = 0 is finished with status 832

4) Child with pid = -1 is finished with status 832

5) Child with pid = -1 is finished with status 832

Proccess pid = 429 was failed.

Proccess pid = 430 was success.

Proccess pid = 431 was failed.

Proccess pid = 432 was failed.

Proccess pid = 433 was failed.

Листинг 9– выполнение программы father2.с(**WUNTRACED, WUNTRACED, WUNTRACED**):

FATHER PARAMS: pid=525 ppid=9

SONPARAMS: pid=526 ppid=525

SONPARAMS: pid=527 ppid=525

SONPARAMS: pid=528 ppid=525

SONPARAMS: pid=529 ppid=525

SONPARAMS: pid=530 ppid=525

1) Child with pid = 526 is finished with status 0

2) Child with pid = 527 is finished with status 256

3) Child with pid = 528 is finished with status 65280

4) Child with pid = -1 is finished with status 832

5) Child with pid = -1 is finished with status 832

Proccess pid = 526 was success.

Proccess pid = 527 was success.

Proccess pid = 528 was success.

Proccess pid = 529 was failed.

Proccess pid = 530 was failed.

Листинг 10– выполнение программы father2.с(**WNOHANG, WUNTRACED, WUNTRACED**):

FATHER PARAMS: pid=540 ppid=9

SONPARAMS: pid=541 ppid=540

SONPARAMS: pid=542 ppid=540

SONPARAMS: pid=543 ppid=540

SONPARAMS: pid=544 ppid=540

SONPARAMS: pid=545 ppid=540

1) Child with pid = 0 is finished with status 832

2) Child with pid = 542 is finished with status 256

3) Child with pid = 543 is finished with status 65280

4) Child with pid = -1 is finished with status 832

5) Child with pid = -1 is finished with status 832

Proccess pid = 541 was failed.

Proccess pid = 542 was success.

Proccess pid = 543 was success.

Proccess pid = 544 was failed.

Proccess pid = 545 was failed.

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. Именно поэтому, в выводе мы наблюдаем информацию о каждом сыне, хоть и код возврата у всех разный

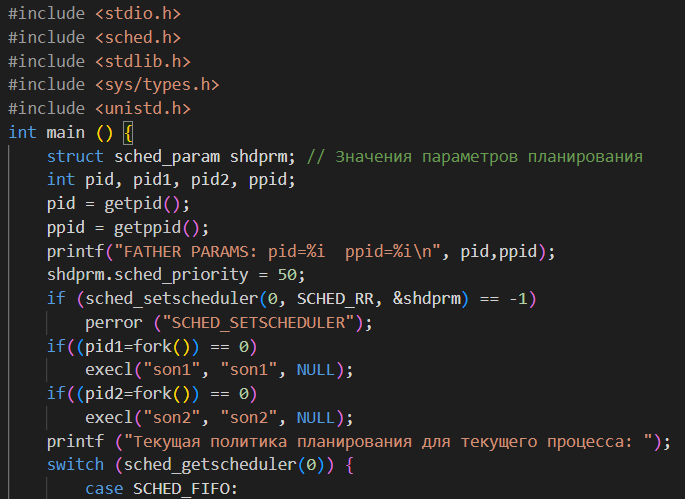
Задание 10.

Системный вызов fork используется для создания нового процесса, называемого дочерним процессом, который выполняется одновременно с процессом, вызвавшим вызов fork() (родительский процесс). Порядок выполнения процессов, генерируемых вызовами fork(), недетерминирован и зависит от различных факторов. Родительский и дочерний процессы выполняются одновременно, и их выполнение может перекрываться, что приводит к сложному и непредсказуемому поведению.

10.1. 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:



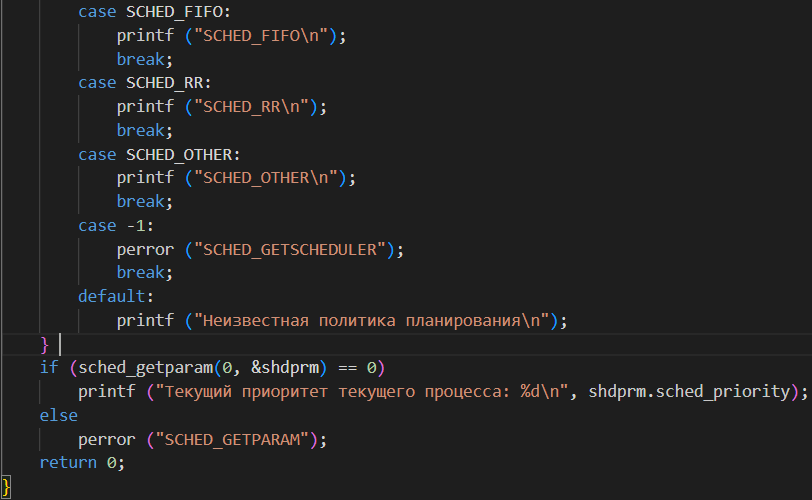


Рисунок 14.1-14.2 – Код программы father.c для выполнения задания 10.2

(взят из методических указаний).

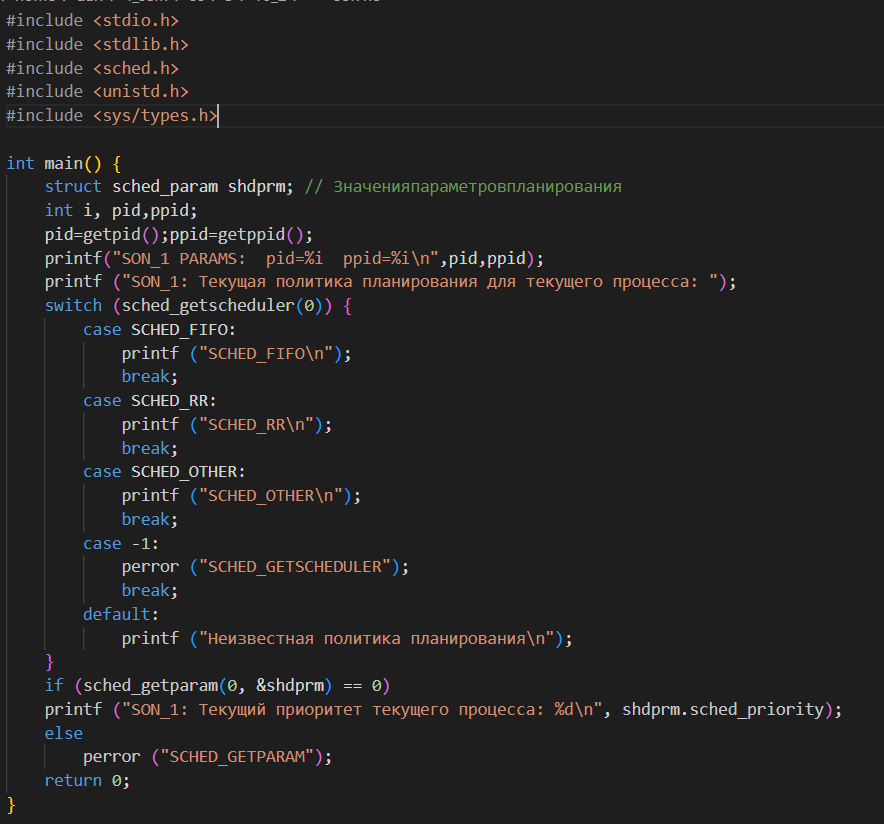


Рисунок 15 – Код программы son1-2.c для выполнения задания 10.2

(взят из методических указаний).

При попытке исполнения программы от имени обычного пользователя, появляется сообщение об ошибке – о том, что операция изменения приоритета запрещена:

Листинг 11– выполнение программы father.с от имени обычного пользователя:

FATHER PARAMS: pid=696 ppid=9

SCHED\_SETSCHEDULER: Operation not permitted

Текущая политика планирования для текущего процесса: SCHED\_OTHER

Текущий приоритет текущего процесса: 0

SON\_1 PARAMS: pid=697 ppid=8

SON\_1: Текущая политика планирования для текущего процесса: SCHED\_OTHER

SON\_1: Текущий приоритет текущего процесса: 0

SON\_2 PARAMS: pid=698 ppid=8

SON\_2: Текущая политика планирования для текущего процесса: SCHED\_OTHER

SON\_2: Текущий приоритет текущего процесса: 0

Для корректного исполнения необходимы права суперпользователя,

тогда пример результатов работы программы:

Листинг 12– выполнение программы father.с от имени супер пользователя:

FATHER PARAMS: pid=713 ppid=706

SON\_1 PARAMS: pid=714 ppid=713

Текущая политика планирования для текущего процесса: SCHED\_RR

Текущий приоритет текущего процесса: 50

SON\_1: Текущая политика планирования для текущего процесса: SCHED\_RR

SON\_1: Текущий приоритет текущего процесса: 50

root@DESKTOP-7CDTIKD:/home/dari/4\_sem/os/3/10/2# SON\_2 PARAMS: pid=715 ppid=8

SON\_2: Текущая политика планирования для текущего процесса: SCHED\_RR

SON\_2: Текущий приоритет текущего процесса: 50

Таким образом, из результатов следует, что потомки наследуют политику планирования и приоритет родительского процесса.

10.3 Поменяем порядок очереди в RR-процедуре.

В алгоритме планирования RR каждому процессу дается фиксированный интервал времени для выполнения, после чего он вытесняется, а запуск следующего процесса планируется. Порядок, в котором процессы ставятся в очередь, определяет порядок их выполнения. Изменяя порядок очереди в процедуре RR, мы можем изменить порядок выполнения процессов. Например, мы можем расставить приоритеты процессов с более высокими значениями приоритета, поместив их в начало очереди, или мы можем дать одинаковый приоритет всем процессам, используя очередь (FIFO).

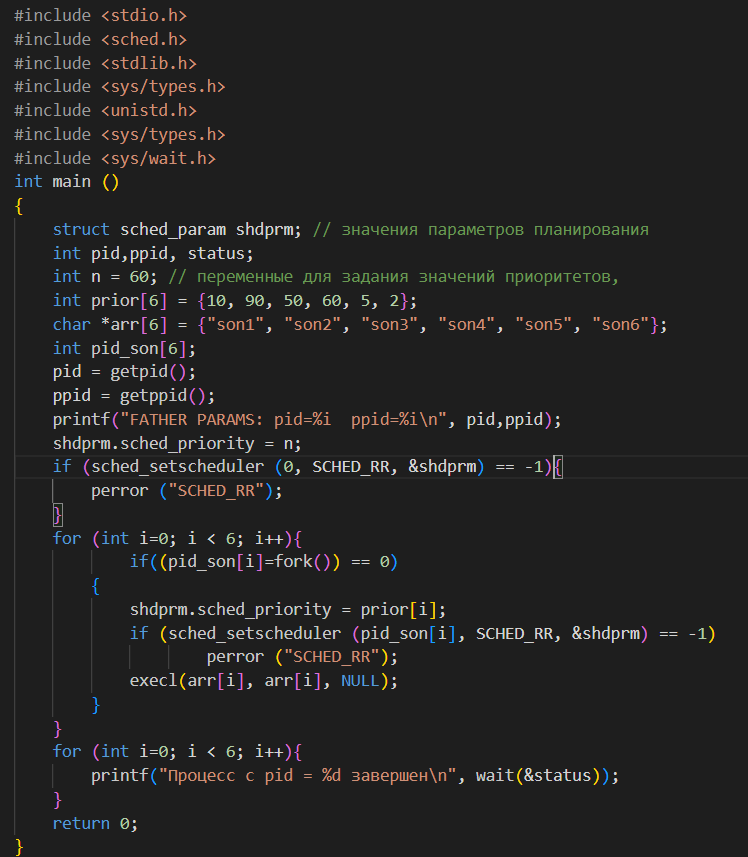


Рисунок 16 – Код программы father\_RR.c для выполнения задания 10.3

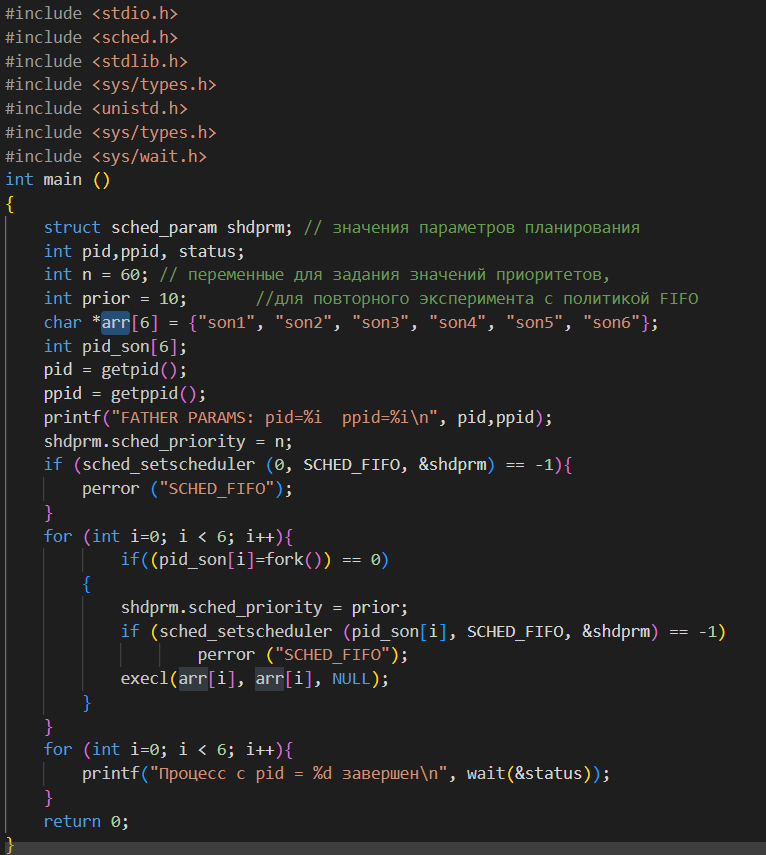


Рисунок 17 – Код программы father\_fifo.c для выполнения задания 10.3

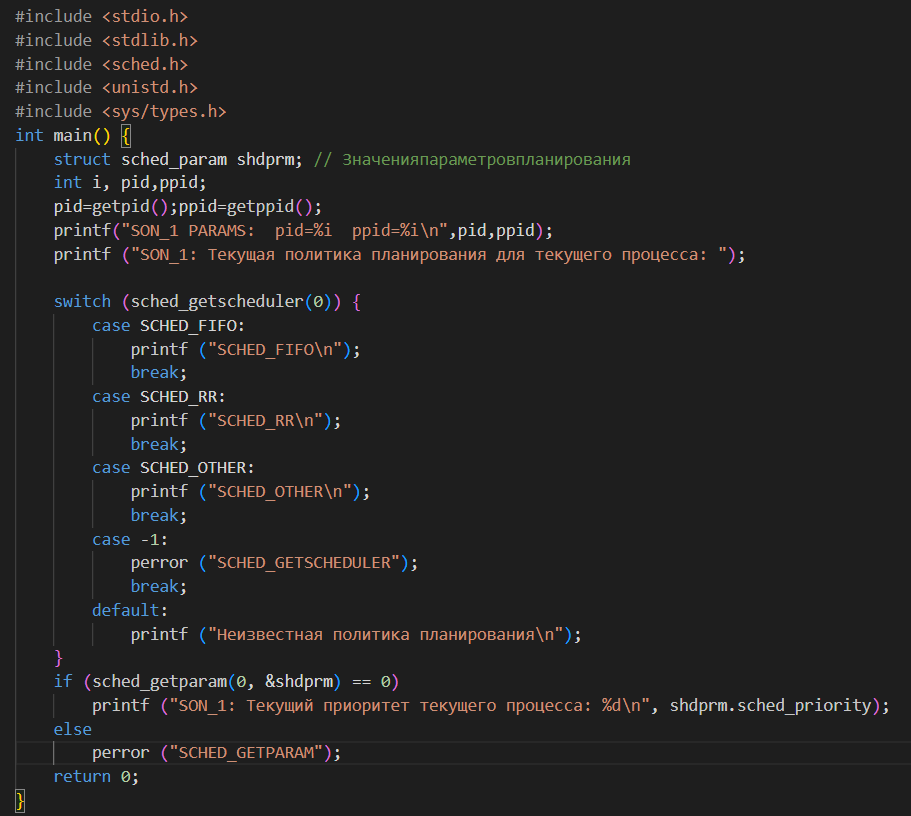


Рисунок 18 – Код программы son1-6.c для выполнения задания 10.3

(взят из методических указаний).

Листинг 13– выполнение программы father\_RR.с от имени обычного пользователя:

taskset -c 0 ./father\_RR

FATHER PARAMS: pid=1401 ppid=1007

SCHED\_RR: Operation not permitted

SCHED\_RR: Operation not permitted

SON\_6 PARAMS: pid=1407 ppid=1401

SON\_6: Текущая политика планирования для текущего процесса: SCHED\_OTHER

SON\_6: Текущий приоритет текущего процесса: 0

Процесс с pid = 1407 завершен

SCHED\_RR: Operation not permitted

SON\_5 PARAMS: pid=1406 ppid=1401

SON\_5: Текущая политика планирования для текущего процесса: SCHED\_OTHER

SON\_5: Текущий приоритет текущего процесса: 0

Процесс с pid = 1406 завершен

SCHED\_RR: Operation not permitted

SON\_4 PARAMS: pid=1405 ppid=1401

SON\_4: Текущая политика планирования для текущего процесса: SCHED\_OTHER

SON\_4: Текущий приоритет текущего процесса: 0

Процесс с pid = 1405 завершен

SCHED\_RR: Operation not permitted

SON\_3 PARAMS: pid=1404 ppid=1401

SON\_3: Текущая политика планирования для текущего процесса: SCHED\_OTHER

SON\_3: Текущий приоритет текущего процесса: 0

Процесс с pid = 1404 завершен

SCHED\_RR: Operation not permitted

SON\_2 PARAMS: pid=1403 ppid=1401

SON\_2: Текущая политика планирования для текущего процесса: SCHED\_OTHER

SON\_2: Текущий приоритет текущего процесса: 0

Процесс с pid = 1403 завершен

SCHED\_RR: Operation not permitted

SON\_1 PARAMS: pid=1402 ppid=1401

SON\_1: Текущая политика планирования для текущего процесса: SCHED\_OTHER

SON\_1: Текущий приоритет текущего процесса: 0

Процесс с pid = 1402 завершен

Листинг 14– выполнение программы father\_RR.с от имени суперпользователя:

sudo taskset -c 0 ./father\_RR

FATHER PARAMS: pid=1394 ppid=1393

SON\_2 PARAMS: pid=1396 ppid=1394

SON\_2: Текущая политика планирования для текущего процесса: SCHED\_RR

SON\_2: Текущий приоритет текущего процесса: 90

SON\_4 PARAMS: pid=1398 ppid=1394

SON\_4: Текущая политика планирования для текущего процесса: SCHED\_RR

SON\_4: Текущий приоритет текущего процесса: 60

Процесс с pid = 1396 завершен

Процесс с pid = 1398 завершен

SON\_3 PARAMS: pid=1397 ppid=1394

SON\_3: Текущая политика планирования для текущего процесса: SCHED\_RR

SON\_3: Текущий приоритет текущего процесса: 50

Процесс с pid = 1397 завершен

SON\_1 PARAMS: pid=1395 ppid=1394

SON\_1: Текущая политика планирования для текущего процесса: SCHED\_RR

SON\_1: Текущий приоритет текущего процесса: 10

Процесс с pid = 1395 завершен

SON\_5 PARAMS: pid=1399 ppid=1394

SON\_5: Текущая политика планирования для текущего процесса: SCHED\_RR

SON\_5: Текущий приоритет текущего процесса: 5

Процесс с pid = 1399 завершен

SON\_6 PARAMS: pid=1400 ppid=1394

SON\_6: Текущая политика планирования для текущего процесса: SCHED\_RR

SON\_6: Текущий приоритет текущего процесса: 2

Процесс с pid = 1400 завершен

Листинг 15– выполнение программы father\_fifo.с от имени обычного пользователя:

FATHER PARAMS: pid=1412 ppid=1007

SCHED\_FIFO: Operation not permitted

SCHED\_FIFO: Operation not permitted

SON\_6 PARAMS: pid=1418 ppid=1412

SON\_6: Текущая политика планирования для текущего процесса: SCHED\_OTHER

SON\_6: Текущий приоритет текущего процесса: 0

Процесс с pid = 1418 завершен

SCHED\_FIFO: Operation not permitted

SON\_5 PARAMS: pid=1417 ppid=1412

SON\_5: Текущая политика планирования для текущего процесса: SCHED\_OTHER

SON\_5: Текущий приоритет текущего процесса: 0

Процесс с pid = 1417 завершен

SCHED\_FIFO: Operation not permitted

SON\_4 PARAMS: pid=1416 ppid=1412

SON\_4: Текущая политика планирования для текущего процесса: SCHED\_OTHER

SON\_4: Текущий приоритет текущего процесса: 0

Процесс с pid = 1416 завершен

SCHED\_FIFO: Operation not permitted

SON\_3 PARAMS: pid=1415 ppid=1412

SON\_3: Текущая политика планирования для текущего процесса: SCHED\_OTHER

SON\_3: Текущий приоритет текущего процесса: 0

Процесс с pid = 1415 завершен

SCHED\_FIFO: Operation not permitted

SON\_2 PARAMS: pid=1414 ppid=1412

SON\_2: Текущая политика планирования для текущего процесса: SCHED\_OTHER

SON\_2: Текущий приоритет текущего процесса: 0

Процесс с pid = 1414 завершен

SCHED\_FIFO: Operation not permitted

SON\_1 PARAMS: pid=1413 ppid=1412

SON\_1: Текущая политика планирования для текущего процесса: SCHED\_OTHER

SON\_1: Текущий приоритет текущего процесса: 0

Процесс с pid = 1413 завершен

Листинг 16– выполнение программы father\_fifo.с от имени суперпользователя:

FATHER PARAMS: pid=1421 ppid=1420

SON\_6 PARAMS: pid=1427 ppid=1421

SON\_6: Текущая политика планирования для текущего процесса: SCHED\_FIFO

SON\_6: Текущий приоритет текущего процесса: 10

Процесс с pid = 1427 завершен

SON\_5 PARAMS: pid=1426 ppid=1421

SON\_5: Текущая политика планирования для текущего процесса: SCHED\_FIFO

SON\_5: Текущий приоритет текущего процесса: 10

Процесс с pid = 1426 завершен

SON\_4 PARAMS: pid=1425 ppid=1421

SON\_4: Текущая политика планирования для текущего процесса: SCHED\_FIFO

SON\_4: Текущий приоритет текущего процесса: 10

Процесс с pid = 1425 завершен

SON\_3 PARAMS: pid=1424 ppid=1421

SON\_3: Текущая политика планирования для текущего процесса: SCHED\_FIFO

SON\_3: Текущий приоритет текущего процесса: 10

Процесс с pid = 1424 завершен

SON\_2 PARAMS: pid=1423 ppid=1421

SON\_2: Текущая политика планирования для текущего процесса: SCHED\_FIFO

SON\_2: Текущий приоритет текущего процесса: 10

Процесс с pid = 1423 завершен

SON\_1 PARAMS: pid=1422 ppid=1421

SON\_1: Текущая политика планирования для текущего процесса: SCHED\_FIFO

SON\_1: Текущий приоритет текущего процесса: 10

Процесс с pid = 1422 завершен

10.4 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

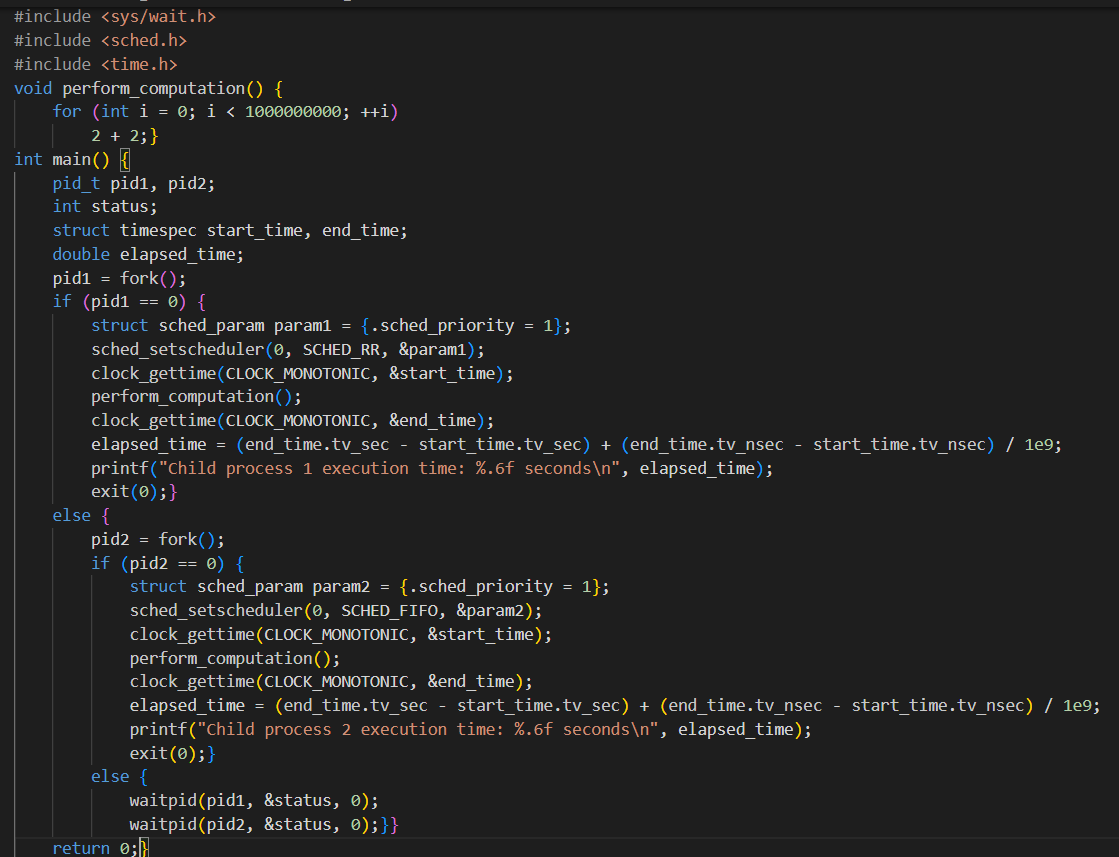


Рисунок 19 – Код программы 4.c для выполнения задания 10.4

Листинг 17– выполнение программы 4.с :

Child process 1 execution time: 1.988816 seconds

Child process 2 execution time: 1.998998 seconds

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

Задание 11.

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

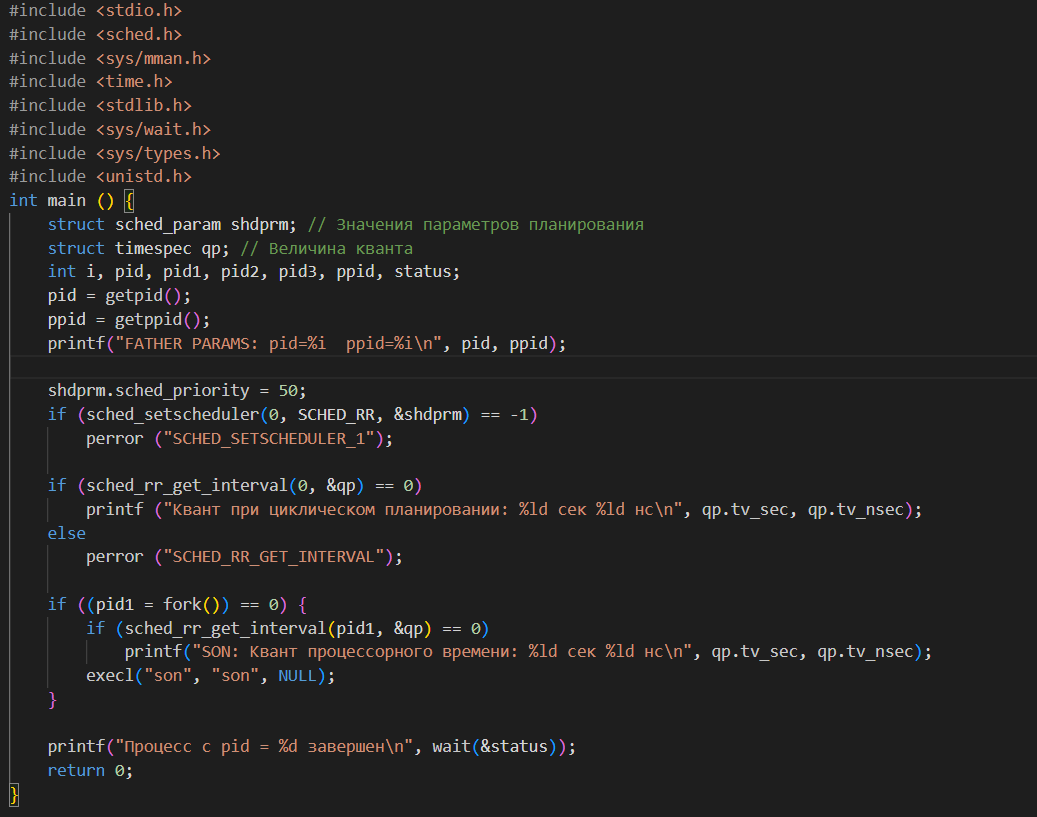


Рисунок 20 – Код программы father.c для выполнения задания 11

(взят из методических указаний).

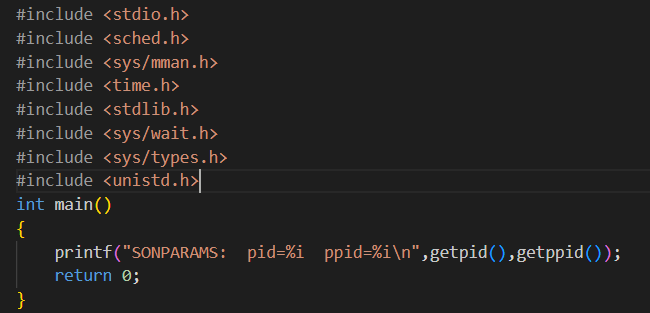


Рисунок 21 – Код программы son.c для выполнения задания 11

(взят из методических указаний).

Листинг 18– выполнение программы father.с :

FATHER PARAMS: pid=1562 ppid=1561

Квант при циклическом планировании: 0 сек 100000000 нс

SON: Квант процессорного времени: 0 сек 100000000 нс

SONPARAMS: pid=1563 ppid=1562

Процесс с pid = 1563 завершен

Экспериментально проверим, можно ли изменить величину кванта, используя системную функцию nice():

Листинг 19–программа father.с с изменением величины кванта :

if ((nice = nice(1000)) == -1)

perror("NICE");

else

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

Листинг 20– выполнение программы father.с с изменением величины кванта :

FATHER PARAMS: pid=105 ppid=10

Nice value = 19

SCHED\_SETSCHEDULER\_1: Operation not permitted

Квант при циклическом планировании:0 сек200000нс

SON: Квант процессорного времени: 0 сек 20000000 нс

SONPARAMS: pid=106 ppid=105

Процесс с pid = 106 завершен

Задание 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.

Листинг 21–содержимое файла infile.txt:

cat infile.txt

HelLo world!

Листинг 22–– выполнение программы father.с:

son file descriptor = 3

son file descriptor = 3

son params: pid=293 ppid=292

son params: pid=294 ppid=292

pid = 293: H

pid = 294: e

pid = 293: l

pid = 294: L

pid = 293: o

pid = 294:

pid = 293: w

pid = 294: o

pid = 293: r

pid = 294: l

pid = 293: d

pid = 294: !

Process pid = 293 completed

Process pid = 294 completed

Листинг 23–– содержимое файла outfile.txt:

cat outfile.txt

HelLo world!

Продолжая эксперимент с программой, демонстрирующей наследование файловых дескрипторов открытых файлов и указателей на позицию при чтении и записи в файл, попробуем закрыть в одном из процессов файл с заданным дескриптором, например, fdrd в son.c. Был немного исправлен код файла son.c

Листинг 24–– выполнение программы father.с после внесения изменений:

son file descriptor = 3

son file descriptor = 3

son params: pid=451 ppid=450

son params: pid=452 ppid=450

pid = 451: H

Process pid = 451 completed

pid = 452: e

Process pid = 452 completed

Листинг 25–– содержимое файла outfile.txt:

cat outfile.txt

He

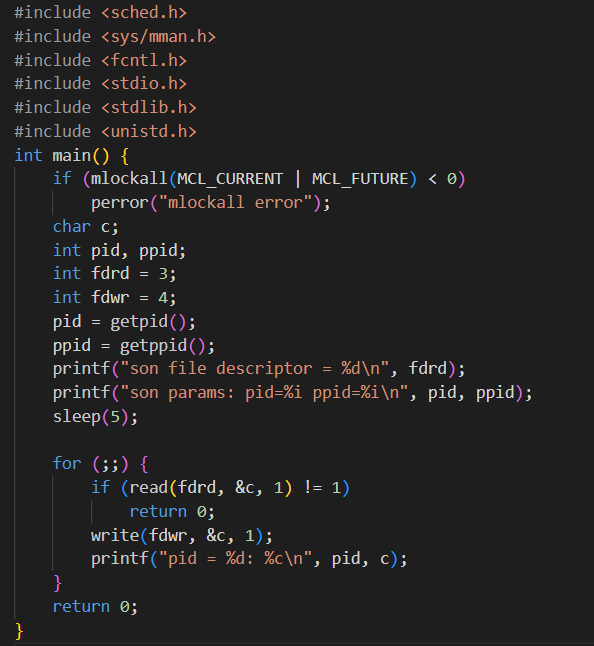


Рисунок 22 – Код программы son.c для выполнения задания 12

(взят из методических указаний).

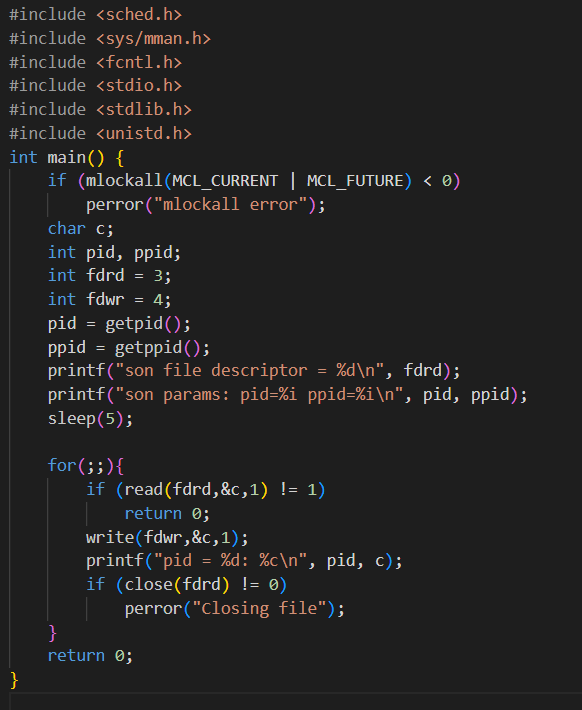


Рисунок 23 – Измененный код программы son.c для выполнения задания 12

(взят из методических указаний).

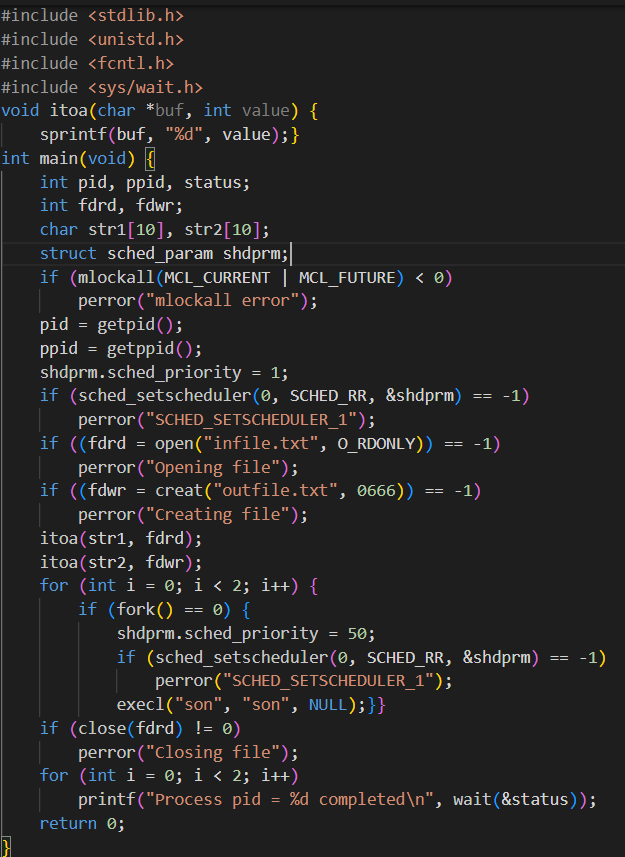


Рисунок 24 – Код программы father.c для выполнения задания 12

(взят из методических указаний).

Убедиться в наследовании других параметров при порождении

потомков можно, проанализировав вывод утилиты

Листинг 26–– Вывод утилиты

ps -o uid,gid,ruid,pid,ppid,pgid,tty,vsz,stat,command

UID GID RUID PID PPID PGID TT VSZ STAT COMMAND

0 0 0 588 10 588 pts/0 11060 S+ sudo ./father

0 0 0 589 588 588 pts/0 2360 SL+ ./father

0 0 0 590 589 588 pts/0 2492 SL+ son

0 0 0 591 589 588 pts/0 2492 SL+ son

0 0 0 592 589 588 pts/0 2612 S+ sh -c ps -o uid,gid,ruid,pid,ppid,pgid,tty,vsz,stat,command

0 0 0 593 592 588 pts/0 10524 R+ 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. Что касается наследования сигналов, рассмотрим это далее при обсуждении сигналов, как средства IPC.

Задание 13.

Поставим простой эксперимент: процесс-родитель создает трех потомков, выполняющихся с различной длительностью по отношению к породившему их процессу:

13.1.a) процесс-отец запускает процесс-сын, ожидает и дожидается его завершения (независимо от длительности выполнения потомка);

13.1.б) процесс-отец запускает процесс-сын и, не ожидая его завершения, завершается сам;

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

Листинг 27–– выполнение программы father.с:

sudo ./father res.txt

FATHER PARAMS: sid = 10 pid=703 ppid=702

SON\_1 PARAMS: pid=704 ppid=703

Father creates andwaits

SON\_3 PARAMS: pid=706 ppid=703

son3terminated–ZOMBIE

SON\_2 PARAMS: pid=705 ppid=703

Father finished before son terminationwithout waiting for it

SON\_3 PARAMS: pid=706 ppid=703

dari@DESKTOP-7CDTIKD:~/4\_sem/os/3/13\_1$ SON\_2 PARAMS ARE CHANGED: pid=705 ppid=9

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. Так как мы не используем ни одну из функций семейства wait для третьего сына, то его состояние переходит в Zombie. Это можно отследить в файле вывода res.txt:

Листинг 28–– содержимое файла res.txt:

cat res.txt

PPID PID PGID SID TTY TPGID STAT UID TIME COMMAND

10 702 702 10 pts/0 702 S+ 0 0:00 sudo ./father res.txt

702 703 702 10 pts/0 702 S+ 0 0:00 \\_ ./father res.txt

703 704 702 10 pts/0 702 S+ 0 0:00 \\_ son1

703 705 702 10 pts/0 702 S+ 0 0:00 \\_ son2 res.txt

703 706 702 10 pts/0 702 Z+ 0 0:00 \\_ [son3] <defunct>

703 707 702 10 pts/0 702 S+ 0 0:00 \\_ sh -c ps -xjf | grep "STAT\|10" > res.txt

707 708 702 10 pts/0 702 R+ 0 0:00 \\_ ps -xjf

707 709 702 10 pts/0 702 R+ 0 0:00 \\_ grep STAT\|10

9 705 702 10 pts/0 10 S 0 0:00 \\_ son2 res.txt

705 710 702 10 pts/0 10 S 0 0:00 \\_ sh -c ps xjf | grep son2 >> res.txt

710 712 702 10 pts/0 10 S 0 0:00 \\_ grep son2

В выводе зафиксированы: нормальное выполнение потомка son1; смена родителя son2 (PPID =1) и его переход в самостоятельную ветку; а также состояние zombie для son3 (ситуация, когда потомок выполняется быстрее процесса-отца, при этом отец не дожидается и никак не фиксирует завершение потомка).

Ознакомиться с полным перечнем сигналов можно с помощью команды kill -l в командном интерпретаторе.

Листинг 29 – Результат выполнения команды 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

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.

Функция системного вызова signal заключается в том, чтобы 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), 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.

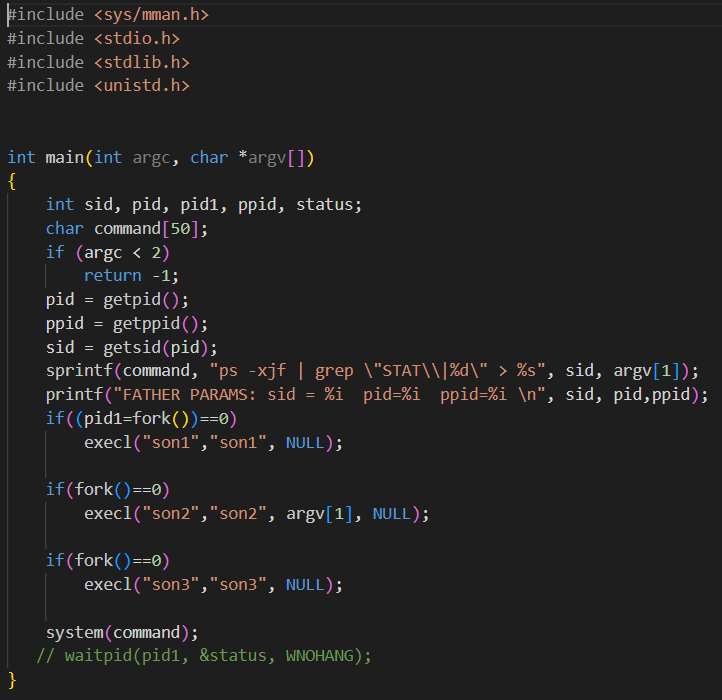


Рисунок 25 – Код программы father.c для выполнения задания 13.1

(взят из методических указаний).

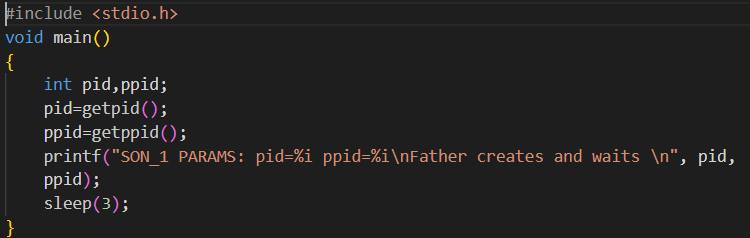


Рисунок 26 – Код программы son1.c для выполнения задания 13.1

(взят из методических указаний).

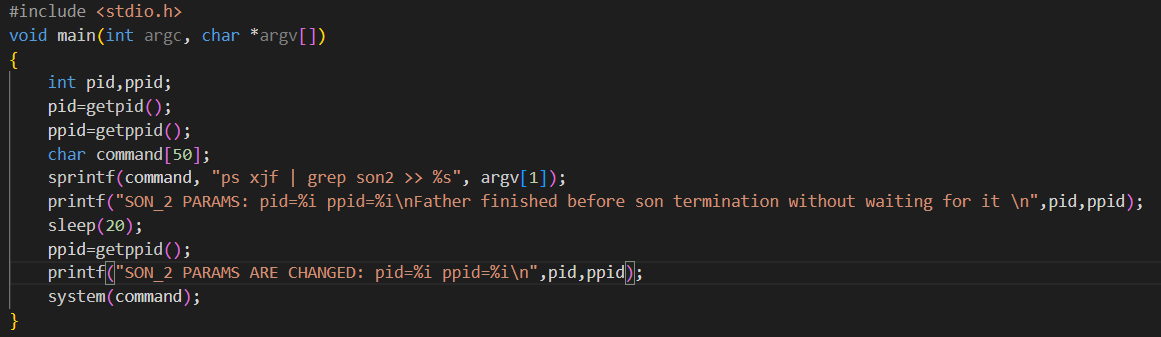


Рисунок 27 – Код программы son2.c для выполнения задания 13.1

(взят из методических указаний).

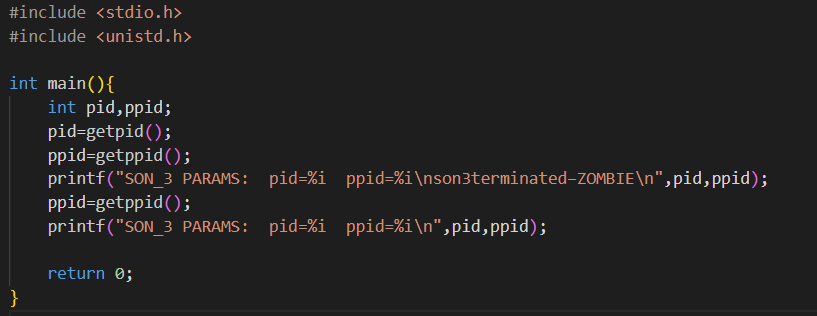


Рисунок 28 – Код программы son3.c для выполнения задания 13.1

(взят из методических указаний).

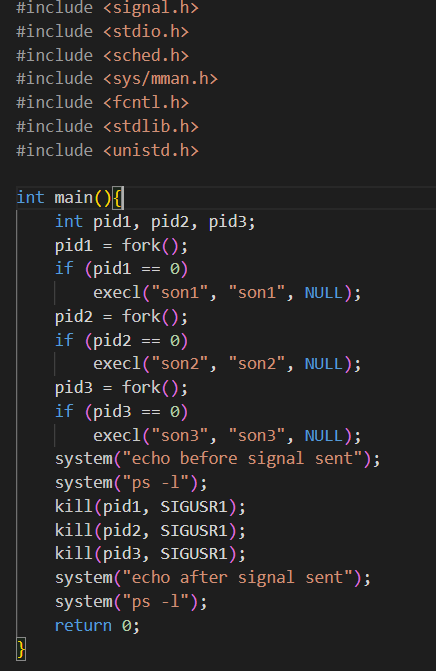


Рисунок 29 – Код программы father.c для выполнения задания 13.1

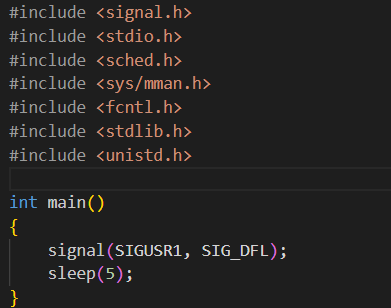


Рисунок 30 – Код программы son1.c для выполнения задания 13.1

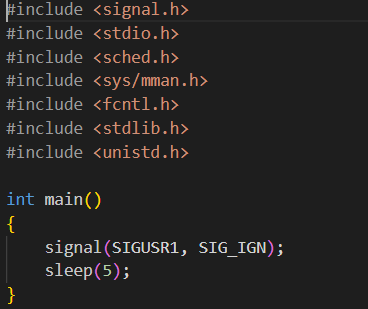


Рисунок 31 – Код программы son2.c для выполнения задания 13.1

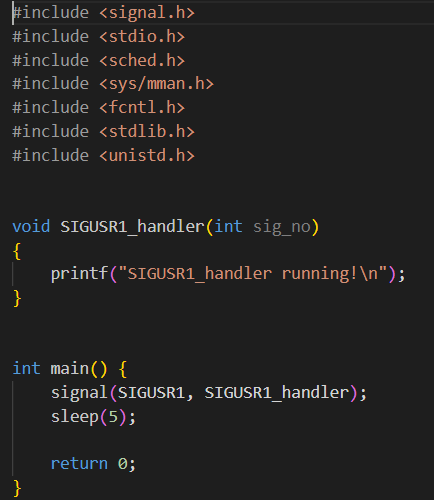


Рисунок 32 – Код программы son3.c для выполнения задания 13.1

Листинг 30 – Результат выполнения father.c:

before signal sent

F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD

4 S 1000 905 904 0 80 0 - 2574 do\_wai pts/0 00:00:00 bash

0 S 1000 1400 905 0 80 0 - 623 do\_wai pts/0 00:00:00 father

0 S 1000 1401 1400 0 80 0 - 590 hrtime pts/0 00:00:00 son1

0 S 1000 1402 1400 0 80 0 - 590 hrtime pts/0 00:00:00 son2

0 S 1000 1403 1400 0 80 0 - 590 hrtime pts/0 00:00:00 son3

0 S 1000 1405 1400 0 80 0 - 653 do\_wai pts/0 00:00:00 sh

0 R 1000 1406 1405 0 80 0 - 2631 - 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

4 S 1000 905 904 0 80 0 - 2574 do\_wai pts/0 00:00:00 bash

0 S 1000 1400 905 0 80 0 - 623 do\_wai pts/0 00:00:00 father

0 Z 1000 1401 1400 0 80 0 - 0 - pts/0 00:00:00 son1 <defunct>

0 S 1000 1402 1400 0 80 0 - 590 hrtime pts/0 00:00:00 son2

0 Z 1000 1403 1400 0 80 0 - 0 - pts/0 00:00:00 son3 <defunct>

0 S 1000 1408 1400 0 80 0 - 653 do\_wai pts/0 00:00:00 sh

0 R 1000 1409 1408 0 80 0 - 2631 - pts/0 00:00:00 ps

13.2. Для организации посылки сигналов любым двум процессам, находящимся в разных состояниях, был написан следующий код:

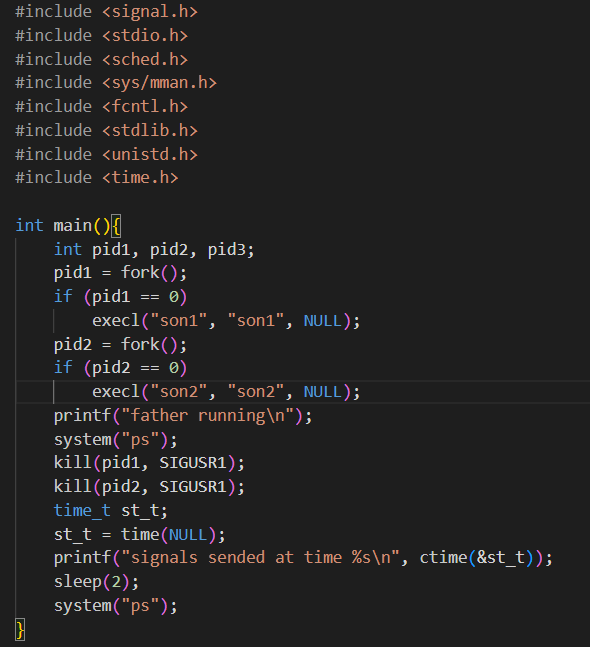


Рисунок 33 – Код программы father.c для выполнения задания 13.2

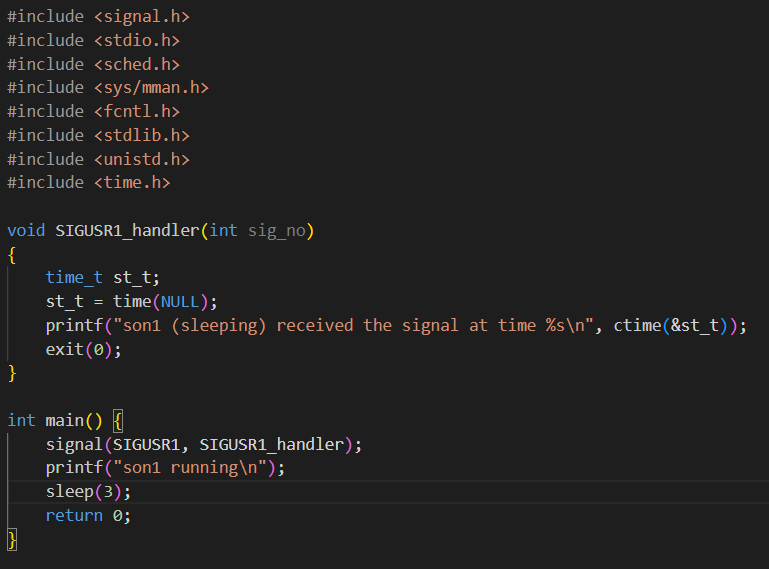


Рисунок 34 – Код программы son1.c для выполнения задания 13.2

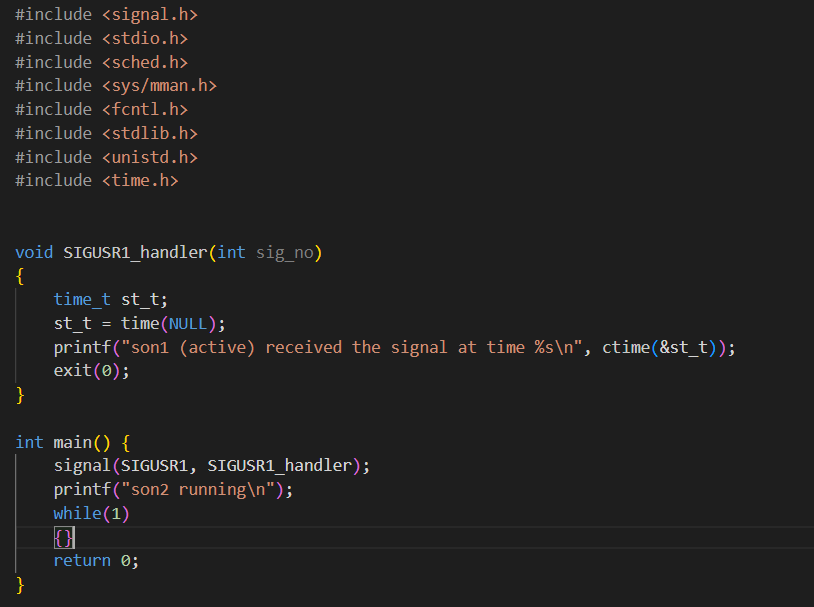


Рисунок 35 – Код программы son2.c для выполнения задания 13.2

Листинг 31 – Результат выполнения father.c:

father running

son1 running

son2 running

PID TTY TIME CMD

905 pts/0 00:00:00 bash

1478 pts/0 00:00:00 father

1479 pts/0 00:00:00 son1

1480 pts/0 00:00:00 son2

1481 pts/0 00:00:00 sh

1482 pts/0 00:00:00 ps

signals sended at time Tue May 16 12:06:57 2023

son1 (sleeping) received the signal at time Tue May 16 12:06:57 2023

son1 (active) received the signal at time Tue May 16 12:06:57 2023

PID TTY TIME CMD

905 pts/0 00:00:00 bash

1478 pts/0 00:00:00 father

1479 pts/0 00:00:00 son1 <defunct>

1480 pts/0 00:00:00 son2 <defunct>

1483 pts/0 00:00:00 sh

1484 pts/0 00:00:00 ps

Задание 14.

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

Чтобы вывести список запущенных процессов их статус, воспользуемся командой jobs.

Листинг 32 – Результат выполнения трех команд в фоновом режиме и отслеживание задач:

sleep 100 & sleep 1000 & sleep 10 &

[1] 26

[2] 27

[3] 28

jobs

[1] Running sleep 100 &

[2]- Running sleep 1000 &

[3]+ Done sleep 10

Видно, что порядок выполнения и pid совпадают, далее указан статус выполнения (выполняется) и сама команда. Можно вывести конкретную задачу по номеру передав % и номер. Последний процесс можно отследить, если вместо номера написать %.

jobs %2

[2]+ Running sleep 1000 &

Осведомиться о завершении какого-либо процесса можно с помощью команды notify. Тогда при каждом завершении работы процесса в фоновом режиме будет сообщаться в терминале:

Листинг 33 – Результат выполнения:

notify %

[4]- Завершён sleep 1000

[7]+ Завершён sleep 1000

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

Листинг 34 – Результат выполнения:

sleep 300 & sleep 330 & sleep 360 & sleep 150 &

[3] 112

[4] 113

[5] 114

[6] 115

dari@DESKTOP-7CDTIKD:~/4\_sem/os$ jobs -l

[2] 27 Running sleep 1000 &

[3] 112 Running sleep 300 &

[4] 113 Running sleep 330 &

[5]- 114 Running sleep 360 &

[6]+ 115 Running sleep 150 &

dari@DESKTOP-7CDTIKD:~/4\_sem/os$ fg %3

sleep 300

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.

Листинг 35 – Результат выполнения:

jobs -l

[2] 27 Running sleep 1000 &

[4] 113 Running sleep 330 &

[5]- 114 Running sleep 360 &

[6]+ 115 Running sleep 150 &

dari@DESKTOP-7CDTIKD:~/4\_sem/os$ kill 27

dari@DESKTOP-7CDTIKD:~/4\_sem/os$ jobs -l

[4] 113 Running sleep 330 &

[5]- 114 Running sleep 360 &

[6]+ 115 Running sleep 150 &

По итогу процесс 27 был принудительно завершен

Задание 15.

Вызов getpriority() позволяет получить приоритет процесса. Команда nice позволяет запустить программу с измененным приоритетом.

Синтаксис:

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

nprioritylevels – это число, на которое будет уменьшен приоритет программы по сравнению с текущим приоритетом. Оно может быть, как положительным, так и отрицательным. 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.

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

Листинг 35 – Код программы lim.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;

}

}

}

Листинг 36 – Результат выполнения

./lim

Нижняя граница = 0

Запросили 0, получили 0

Верхняя граница = 19

Запросили 20, получили 19

dari@DESKTOP-7CDTIKD:~/4\_sem/os/3/15$ sudo ./lim

Нижняя граница = -20

Запросили -20, получили -20

Верхняя граница = 19

Запросили 20, получили 19В

приоритетах для системных и пользовательских процессов может существовать следующая разница: для особо важных или критичных системных процессов используются приоритеты класса реального времени.

Приоритет реального времени имеют такие процессы, как migration/0-3. Процесс ядра migration распределяет рабочую нагрузку между ядрами ЦП. Таких процессов столько же, сколько ядер (потоков) у CPU.

Листинг 37 – Результат выполнения команды top-b | grep -w “migration”

top -b | grep -w "migration"

15 root rt 0 0 0 0 S 0,0 0,0 0:00.04 migration/0

21 root rt 0 0 0 0 S 0,0 0,0 0:00.22 migration/1

27 root rt 0 0 0 0 S 0,0 0,0 0:00.21 migration/2

С помощью команды top можно узнать информацию о процессах системы, с помощью команды top -b | grep -w “shell” можно узнать о приоритете для запуска приложений из shell.

Листинг 38 – Результат выполнения команды top -b

top -b | grep -w "shell"

1634 text 20 0 4524616 364484 126952 S 18,8 9,1 19:47.59 gnome-shell

1688 text 20 0 581076 10904 10032 S 0,0 0,3 0:00.07 gnome-shell-cal

Системным процессам обычно присваивается более высокий приоритет, чем пользовательским, поскольку они отвечают за управление критически важными системными функциями, такими как управление памятью, операции ввода-вывода и планирование процессов.

Приоритеты в реальном времени часто используются для процессов, требующих немедленного и предсказуемого реагирования. Этим процессам присваивается более высокий приоритет, чем другим процессам, чтобы гарантировать, что они получают достаточные системные ресурсы и могут выполняться своевременно.

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

Задание 16.

SIGHUP – это сигнал, посылаемый процессу для уведомления о потере соединения с управляющим терминалом пользователя. Потеря соединения в частности возникает при выходе пользователя из системы. Этот сигнал может быть перехвачен или проигнорирован программой.

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.

Игнорирование можно установить, начав выполнять процесс с утилитой nohup. Помимо настройки игнорирования она обеспечивает запуск программы (не в фоновом режиме), перенаправляя весь вывод в файл nohup.out в текущей директории или, если его невозможно создать, в домашнем каталоге пользователя (если и там его невозможно создать, то команда просто не запустится). После потери связи с терминалом программа продолжит выполняться в фоновом режиме.

Листинг 39– Код программы 16.c

#include <stdio.h>

void main()

{

int i;

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

Листинг 40 – Выполнение программы до выхода из системы

nohup ./16 &

[1] 192

dari@DESKTOP-7CDTIKD:~/4\_sem/os/3/16$ nohup: ignoring input and appending output to 'nohup.out'

При выходе из системы процесс выполняется.

Листинг 41 – Выполнение команды после выхода из системы

ps xa

PID TTY STAT TIME COMMAND

1 ? Sl 0:01 /init

7 ? Ss 0:00 /init

8 ? R 0:00 /init

9 pts/0 Ss 0:00 -bash

192 pts/0 R 0:07 ./16

193 pts/0 R+ 0:00 ps xa

Задание 17.

UID процесса - это идентификатор пользователя, от имени которого был запущен процесс. Минимально возможное значение UID - 0, оно принадлежит суперпользователю (root), который имеет полный доступ ко всем ресурсам системы. Максимальное значение UID определяется конкретной операционной системой и может быть очень большим.

Минимальное значение PID (идентификатор процесса) обычно равно 1, а максимальное - определяется конкретной операционной системой. Процессы с PID от 1 до 999 обычно являются системными процессами, которые запускаются при загрузке операционной системы и выполняют административные задачи. Процессы с PID от 1000 и выше представляют пользовательские процессы, которые запускаются от имени конкретного пользователя.

Чтобы определить UID процесса в системе на базе Unix, можно использовать команду "ps" вместе с опцией "u". Например, чтобы отобразить UID всех запущенных процессов, можно использовать следующую команду: 47 ps uax В Linux минимальное значение UID обычно равно 0, которое принадлежит пользователю root. Максимальное значение UID обычно равно 4294967294 (2^32 - 2), которое зарезервировано как максимальное значение UID и используется для представления пользователя nobody. Однако фактическое максимальное значение UID может быть настроено во время установки дистрибутива Linux или может быть изменено позже системным администратором. Максимальное значение UID обычно определяется в файле "/etc/login.defs" или в файле "/etc/nsswitch.conf"

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

Листинг 42 – содержимое "/etc/login.defs"

#

# Min/max values for automatic uid selection in useradd

#

UID\_MIN 1000

UID\_MAX 60000

# System accounts

#SYS\_UID\_MIN 100

#SYS\_UID\_MAX 999

#

# Min/max values for automatic gid selection in groupadd

#

GID\_MIN 1000

GID\_MAX 60000

# System accounts

#SYS\_GID\_MIN 100

#SYS\_GID\_MAX 999

Важно отметить, что максимальное значение PID может повлиять на производительность системы и использование ресурсов, поскольку более высокие значения PID требуют больше ресурсов памяти и процессора. Кроме того, значения PID уменьшаются после завершения процесса, поэтому два процесса могут иметь одинаковый PID, если они не выполняются одновременно.

Чтобы отличить системные процессы от пользовательских, можно посмотреть на принадлежность процессов пользователю и группе. Системные процессы обычно принадлежат пользователю root или системному пользователю, в то время как пользовательские процессы принадлежат обычным пользователям.

Вот некоторые общие системные процессы и их цели:

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.

dbus-daemon: демон, который обеспечивает межпроцессное взаимодействие между различными приложениями.

cupsd: Демон диспетчера очереди печати, который управляет службами печати в системе.

acpid: Демон, который обрабатывает события ACPI (Advanced Configuration and Power Interface) в системе.

Задание 18.

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 первой и второй нитей соответственно), на экран должно выводиться имя нити и количество пятисекундных (для первой) и секундных (для второй) интервалов функционирования каждой нити. Код программы представлен ниже.

Листинг 43 – Код 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;

}

Листинг 44 – Результат выполнения программы 18.c

sudo ./18

[sudo] password for dari:

Thread 1: 0

Thread 2: 0

Thread 2: 1

Thread 2: 2

Thread 2: 3

Thread 2: 4

Thread 1: 1

Thread 2: 5

Thread 2: 6

Thread 2: 7

Задание 19.

При попытке удаления нити, удаляется процесс в целом, поскольку все нити имеют одинаковый идентификатор. Код программы, подтверждающей это, представлен ниже.

Листинг 45– Код программы 19.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;

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;

// 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);

}

}

system("ps -T");

// join remaining child thread

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

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

exit(1);

}

return 0;

}

Результат выполнения программы представлен ниже.

Листинг 46 – Результат выполнения 19.c

sudo ./19

Thread 1: 0

Thread 2: 0

PID SPID TTY TIME CMD

383 383 pts/0 00:00:00 sudo

384 384 pts/0 00:00:00 19

384 385 pts/0 00:00:00 19

384 386 pts/0 00:00:00 19

387 387 pts/0 00:00:00 sh

388 388 pts/0 00:00:00 ps

Thread 2: 1

Thread 2: 2

Thread 2: 3

Как можно заметить, все потоки имеют одинаковый PID, поэтому при попытке удалить один поток, удалится весь процесс. Смотреть что будет если от двух разных потоков создать

Задание 20.

Модифицирована программа так, чтобы управление второй нитью осуществлялось посредством сигнала SIGUSR1 из первой нити. На пятой секунде работы приложения удалена вторая нить. Для этого воспользовалась функцией pthread\_kill(t2, SIGUSR);

Листинг 47 – Код программы 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;

}

Листинг 49 – Результат выполнения

sudo ./20

Thread 1: 0

Thread 2: 0

Thread 2: 1

Thread 2: 2

Thread 2: 3

Thread 2: 4

Определяемый пользователем сигнал 1

Как видно, удалить только вторую нить не удалось, вновь удалился весь процесс

Задание 21.

Для данного задания был создан собственный обработчик сигнала, в итоге получилось организовать удаление по 1 нити, в то время как другие продолжали выполняться и процесс не прекращался.

Листинг 50 – Код программы для задания 21

#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;

}

Листинг 51 – Результат выполнения программы

sudo ./21

[sudo] password for dari:

Thread 1 started

Thread 2 started

PID SPID TTY TIME CMD

427 427 pts/0 00:00:00 sudo

428 428 pts/0 00:00:00 21

428 429 pts/0 00:00:00 21

428 430 pts/0 00:00:00 21

431 431 pts/0 00:00:00 sh

432 432 pts/0 00:00:00 ps

Thread 2: slept 1 times

Thread 2: slept 2 times

Thread 2: slept 3 times

Thread 2: slept 4 times

Thread 1: slept 1 times

SIGUSR1 received by thread2

Thread 1: slept 2 times

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

Задание 22.

Для данного задания была написана программа, которая позволяет перехватить сигнал, генерируемый в результате нажатия комбинации клавиш (Ctrl+C).

Листинг 52 – Код программы 22.c

#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;

}

Листинг 53 – Результат выполнения программы

sudo ./22

Current pid = 839 and ppid = 838

^C^C -signal received

^C

Теперь изменим программу таким образом, чтоб можно было несколько раз отлавливать сигнал. Результат представлен ниже.

Листинг 54 – Изменение программы 22.c(22\_1.c)

#include <stdio.h>

#include <signal.h>

int count = 0;

void handler(){

count++;

puts("\nSignal Ctrl+C was catched");

if (count == 3){

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;

}

Значение переменной count можно задавать любое значение. Результат выполнения представлен ниже.

Листинг 55 – Результат выполнения измененной программы

sudo ./22\_1

Current pid = 854 and ppid = 853

^CSignal Ctrl+C was catched

^C

Signal Ctrl+C was catched

^CSignal Ctrl+C was catched

^C

Теперь проделаем то же самое, что и в предыдущий раз, только создадим нить, которая будет отлавливать сигналы. Для однократного срабатывания в аргумент max нужно передать 1.

Листинг 56 – Код программы 22\_2.c

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#include <signal.h>

int count = 0;

int max;

void \*thread() {

printf("Thread 1 was created\n");

while(1);

}

void handler(){

count++;

puts("\nSignal Ctrl+C was catched");

if (count == max){

signal(SIGINT, SIG\_DFL);

}

}

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

max = atoi(argv[1]);

pthread\_t t;

signal(SIGINT, handler);

pthread\_create(&t, NULL, thread, NULL);

pthread\_join(t, NULL);

return 0;

}

Результат выполнения представлен ниже.

Листинг 57 – Результат выполнения 22\_2.c

sudo ./22\_2 5

Thread 1 was created

^C

Signal Ctrl+C was catched

^C

Signal Ctrl+C was catched

^C

Signal Ctrl+C was catched

^C

Signal Ctrl+C was catched

^C

Signal Ctrl+C was catched

^C

Теперь попробуем написать программу, которая будет отлавливать другой сигнал(ctrl+Z). Ниже представлен код для отлавливания сигнала для процесса и сразу же результат выполнения данной программы.

Листинг 58 – Отлавливание другого сигнала

#include <stdio.h>

#include <signal.h>

#include <stdlib.h>

int count = 0;

int max;

void handler(){

    count++;

    puts("\nSignal Ctrl+Z was catched");

    if (count == max){

        signal(SIGTSTP, SIG\_DFL);

    }

}

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

    max = atoi(argv[1]);

    int pid, ppid;

    pid = getpid();

    ppid = getppid();

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

    signal(SIGTSTP, handler);

    while(1){

    }

return 0;

}

Листинг 59 – Результат выполнения 22\_3.c

sudo ./22\_3 3

Current pid = 906 and ppid = 905

^Z

Signal Ctrl+Z was catched

^Z

Signal Ctrl+Z was catched

^ZSignal Ctrl+Z was catched

^Z

[2]+ Stopped sudo ./22\_3 3

Теперь повторим то же самое для нити. Код и результат выполнения представлен ниже.

Листинг 60 – Отлавливание другого сигнала для потока 22\_4.c

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#include <signal.h>

int count = 0;

int max;

void \*thread() {

printf("Thread 1 was created\n");

while(1);

printf("Thread 1 terminated\n");

}

void handler(){

count++;

puts("\nSignal Ctrl+Z was catched");

if (count == max){

signal(SIGTSTP, SIG\_DFL);

}

}

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

max = atoi(argv[1]);

pthread\_t t;

signal(SIGTSTP, handler);

pthread\_create(&t, NULL, thread, NULL);

pthread\_join(t, NULL);

return 0;

}

Листинг 61 – Результат выполнения 22\_4.c

sudo ./22\_4 2

Thread 1 was created

^Z

Signal Ctrl+Z was catched

^Z

Signal Ctrl+Z was catched

^Z

[3]+ Stopped sudo ./22\_4 2

Задание 23.

С помощью утилиты kill был выведен список всех сигналов.

Листинг 62 – Результат выполнения kill -l

sudo kill -l

HUP INT QUIT ILL TRAP ABRT BUS FPE KILL USR1 SEGV USR2 PIPE ALRM TERM STKFLT

CHLD CONT STOP TSTP TTIN TTOU URG XCPU XFSZ VTALRM PROF WINCH POLL PWR SYS

Рассмотрим некоторые из сигналов базового списка:

1)SIGHUP предназначен для того, чтобы информировать программу о потере связи с управляющим терминалом, так же и в том случае, если процесс-лидер сессии завершил свою работу. Многие программы-демоны, у которых нет лидера сессии, так же обрабатывают этот сигнал. В ответ на получение SIGHUP демон обычно перезапускается. По умолчанию программа, получившая этот сигнал, завершается.

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

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

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

11)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.

15)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.

17)SIGCHLD посылается процессу в том случае, 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.

18)SIGCONT возобновляет выполнение процесса, остановленного сигналом SIGSTOP.

19)SIGSTOP приостанавливает выполнение процесса. Как и SIGKILL, этот сигнал невозможно перехватить или игнорировать.

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

29)SIGIO сообщает процессу, что на одном из дескрипторов, открытых асинхронно, появились данные. По умолчанию этот сигнал завершает работу программы.

10) и 12) SIGUSR1 и SIGUSR2 предназначены для прикладных зада и передачи ими произвольной информации.

Сигналы с 32 по 64 известны как «сигналы реального времени» и используются для межпроцессного взаимодействия и синхронизации в многопоточных системах и системах реального времени. В отличие от стандартных сигналов (от 1 до 31), сигналы реального времени ставятся в очередь, что позволяет принимающему процессу или потоку получать их в порядке поступления.

Задание 24.

Проанализирована процедура планирования для процессов и потоков одного процесса. Совершены попытки процедуру планирования изменить. Заданы нитям разные приоритеты программyно и извне

Листинг 63 – Код программы для 24 задания

#include <signal.h>

#include <pthread.h>

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/syscall.h>

#include <time.h>

#define BILLION 1000000000.0

typedef struct Data Data;

const int COUNT = 1500000;

const int COUNT\_THREADS = 6;

const int LENGHT\_FILE\_LINE = 10;

void printPS()

{

char command[80];

sprintf(command, "ps -T -p %d -o f,s,pid,ppid,spid,cls,pri,ni,stat,cmd,rtprio > file.txt", getpid());

system(command);

}

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

struct timespec start, end;

clock\_gettime(CLOCK\_REALTIME, &start);

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

int tid = syscall(SYS\_gettid);

int pid = getpid();

int a = 0;

printPS();

printf("\nThread\_%d with tid = %d and pid = %d is started\n",

\*id, tid, pid);

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

a++;

}

clock\_gettime(CLOCK\_REALTIME, &end);

double time\_spent = (end.tv\_sec - start.tv\_sec) +

(end.tv\_nsec - start.tv\_nsec) / BILLION;

printf("\nThread\_%d with id = %d and pid = %d is completed, %fs\n",

\*id, tid, pid, time\_spent);

}

int read\_from\_file(int \*priors, int \*policies, int \*isInherits)

{

FILE \*file = fopen("params.txt", "r");

char line[LENGHT\_FILE\_LINE];

if(file)

{

int i = 0;

while(fgets(line, LENGHT\_FILE\_LINE, file) != NULL)

{

if(i < 6)

{

priors[i] = atoi(line);

} else if(i < 12) {

policies[i - 6] = atoi(line);

} else

\*isInherits = atoi(line);

i++;

}

}

else

return -1;

fclose(file);

return 0;

}

int main()

{

pthread\_t threads[COUNT\_THREADS];

pthread\_attr\_t thread\_attributes[COUNT\_THREADS];

int priorities[COUNT\_THREADS], policies[COUNT\_THREADS];

int inherit = 0;

int IDs[COUNT\_THREADS];

for(int i = 0; i < COUNT\_THREADS; i++)

IDs[i] = i + 1;

int policy;

struct sched\_param param;

for(int i = 0; i < COUNT\_THREADS; i++) // инициализируем

pthread\_attr\_init(&thread\_attributes[i]); // описателей атрибутов

read\_from\_file(priorities, policies, &inherit);

for(int i = 0; i < COUNT\_THREADS; i++)

{

pthread\_attr\_setschedpolicy(&thread\_attributes[i], policies[i]);

param.sched\_priority = priorities[i];

pthread\_attr\_setschedparam(&thread\_attributes[i], &param);

}

if(inherit == 1)

{

for(int i = 0; i < COUNT\_THREADS; i++)

pthread\_attr\_setdetachstate(&thread\_attributes[i], PTHREAD\_INHERIT\_SCHED);

}

else

{

for(int i = 0; i < COUNT\_THREADS; i++)

pthread\_attr\_setinheritsched(&thread\_attributes[i], PTHREAD\_EXPLICIT\_SCHED);

}

for(int i = 0; i < COUNT\_THREADS; i++)

{

pthread\_attr\_getschedparam(&thread\_attributes[i], &param);

pthread\_attr\_getschedpolicy(&thread\_attributes[i], &policy);

printf("Поток№%d, его приоритет = %d\n", i + 1, param.sched\_priority);

}

switch (policy) {

case SCHED\_FIFO:

printf ("Политика процесса: SCHED\_FIFO\n");

break;

case SCHED\_RR:

printf ("Политика процесса: SCHED\_RR\n");

break;

case SCHED\_OTHER:

printf ("Политика процесса: SCHED\_OTHER\n");

break;

case -1:

perror ("Политика процесса: SCHED\_GETSCHEDULER");

break;

default:

printf ("Политика процесса: Неизвестная политика планирования\n");

}

for(int i = 0; i < COUNT\_THREADS; i++)

{

if(pthread\_create(&threads[i], &thread\_attributes[i], Thread\_func, &IDs[i]))

perror("Статус создания потока");

}

for(int i = 0; i < COUNT\_THREADS; i++)

pthread\_join(threads[i], NULL);

for(int i = 0; i < COUNT\_THREADS; i++)

pthread\_attr\_destroy(&thread\_attributes[i]);

return 0;

}

Листинг 64 – Вывод результата программы при FIFO

sudo ./24

[sudo] password for dari:

Поток 1, его приоритет = 50

Поток 2, его приоритет = 50

Поток 3, его приоритет = 50

Поток 4, его приоритет = 50

Поток 5, его приоритет = 50

Поток 6, его приоритет = 50

Поток 7, его приоритет = 50

Поток 8, его приоритет = 50

Поток 9, его приоритет = 50

Поток 10, его приоритет = 50

Поток 11, его приоритет = 50

Поток 12, его приоритет = 50

Поток 13, его приоритет = 50

Поток 14, его приоритет = 50

Поток 15, его приоритет = 50

Поток 16, его приоритет = 50

Поток 17, его приоритет = 50

Поток 18, его приоритет = 50

SCHED\_FIFO

Thread\_8 with tid = 2089 and pid = 2072 is started

Thread\_6 with tid = 2085 and pid = 2072 is started

Thread\_2 with tid = 2074 and pid = 2072 is started

Thread\_4 with tid = 2081 and pid = 2072 is started

Thread\_3 with tid = 2075 and pid = 2072 is started

Thread\_5 with tid = 2082 and pid = 2072 is started

Thread\_7 with tid = 2087 and pid = 2072 is started

Thread\_1 with tid = 2073 and pid = 2072 is started

Thread\_3 with id = 2075 and pid = 2072 is completed, 0.860949s

Thread\_6 with id = 2085 and pid = 2072 is completed, 0.864106s

Thread\_11 with tid = 2103 and pid = 2072 is started

Thread\_10 with tid = 2099 and pid = 2072 is started

Thread\_8 with id = 2089 and pid = 2072 is completed, 0.865814s

Thread\_9 with tid = 2097 and pid = 2072 is started

Thread\_4 with id = 2081 and pid = 2072 is completed, 0.872054s

Thread\_2 with id = 2074 and pid = 2072 is completed, 0.895305s

Thread\_1 with id = 2073 and pid = 2072 is completed, 0.901636s

Thread\_5 with id = 2082 and pid = 2072 is completed, 0.896873s

Thread\_7 with id = 2087 and pid = 2072 is completed, 0.897057s

Thread\_18 with tid = 2117 and pid = 2072 is started

Thread\_15 with tid = 2112 and pid = 2072 is started

Thread\_14 with tid = 2109 and pid = 2072 is started

Thread\_12 with tid = 2106 and pid = 2072 is started

Thread\_17 with tid = 2116 and pid = 2072 is started

Thread\_11 with id = 2103 and pid = 2072 is completed, 1.718775s

Thread\_13 with tid = 2108 and pid = 2072 is started

Thread\_10 with id = 2099 and pid = 2072 is completed, 1.729000s

Thread\_16 with tid = 2113 and pid = 2072 is started

Thread\_9 with id = 2097 and pid = 2072 is completed, 1.733971s

Thread\_17 with id = 2116 and pid = 2072 is completed, 0.879802s

Thread\_15 with id = 2112 and pid = 2072 is completed, 0.888038s

Thread\_18 with id = 2117 and pid = 2072 is completed, 0.894902s

Thread\_14 with id = 2109 and pid = 2072 is completed, 0.905035s

Thread\_12 with id = 2106 and pid = 2072 is completed, 0.928855s

Thread\_13 with id = 2108 and pid = 2072 is completed, 1.697099s

Thread\_16 with id = 2113 and pid = 2072 is completed, 1.708834s

Листинг 65 – Вывод результата программы при RR

sudo ./24

Поток 1, его приоритет = 50

Поток 2, его приоритет = 40

Поток 3, его приоритет = 44

Поток 4, его приоритет = 20

Поток 5, его приоритет = 33

Поток 6, его приоритет = 32

Поток 7, его приоритет = 31

Поток 8, его приоритет = 40

Поток 9, его приоритет = 44

Поток 10, его приоритет = 20

Поток 11, его приоритет = 33

Поток 12, его приоритет = 32

Поток 13, его приоритет = 31

Поток 14, его приоритет = 40

Поток 15, его приоритет = 44

Поток 16, его приоритет = 20

Поток 17, его приоритет = 33

Поток 18, его приоритет = 32

SCHED\_RR

Thread\_5 with tid = 1953 and pid = 1944 is started

Thread\_3 with tid = 1949 and pid = 1944 is started

Thread\_2 with tid = 1946 and pid = 1944 is started

Thread\_7 with tid = 1963 and pid = 1944 is started

Thread\_1 with tid = 1945 and pid = 1944 is started

Thread\_6 with tid = 1956 and pid = 1944 is started

Thread\_8 with tid = 1965 and pid = 1944 is started

Thread\_9 with tid = 1969 and pid = 1944 is started

Thread\_1 with id = 1945 and pid = 1944 is completed, 0.835666s

Thread\_4 with tid = 1951 and pid = 1944 is started

Thread\_2 with id = 1946 and pid = 1944 is completed, 0.836466s

Thread\_7 with id = 1963 and pid = 1944 is completed, 0.836825s

Thread\_8 with id = 1965 and pid = 1944 is completed, 0.836951s

Thread\_9 with id = 1969 and pid = 1944 is completed, 0.840975s

Thread\_3 with id = 1949 and pid = 1944 is completed, 0.848799s

Thread\_5 with id = 1953 and pid = 1944 is completed, 0.848790s

Thread\_10 with tid = 1972 and pid = 1944 is started

Thread\_11 with tid = 1974 and pid = 1944 is started

Thread\_12 with tid = 1976 and pid = 1944 is started

Thread\_13 with tid = 1978 and pid = 1944 is started

Thread\_6 with id = 1956 and pid = 1944 is completed, 0.849408s

Thread\_14 with tid = 1984 and pid = 1944 is started

Thread\_15 with tid = 1986 and pid = 1944 is started

Thread\_17 with tid = 1992 and pid = 1944 is started

Thread\_18 with tid = 1996 and pid = 1944 is started

Thread\_4 with id = 1951 and pid = 1944 is completed, 1.687793s

Thread\_12 with id = 1976 and pid = 1944 is completed, 0.892059s

Thread\_13 with id = 1978 and pid = 1944 is completed, 0.892658s

Thread\_14 with id = 1984 and pid = 1944 is completed, 0.889557s

Thread\_16 with tid = 1988 and pid = 1944 is started

Thread\_15 with id = 1986 and pid = 1944 is completed, 0.896506s

Thread\_11 with id = 1974 and pid = 1944 is completed, 0.905226s

Thread\_17 with id = 1992 and pid = 1944 is completed, 0.955555s

Thread\_18 with id = 1996 and pid = 1944 is completed, 0.848465s

Thread\_10 with id = 1972 and pid = 1944 is completed, 1.600318s

Thread\_16 with id = 1988 and pid = 1944 is completed, 1.735823s

Можно заметить, что в FIFO потоки выполняются ровно в том порядке, в котором они создались, в отличие от RR, в котором если поток не успевает завершиться за квант времени, выполнение передается следующему потоку.

Можно провести эксперимент с разными приоритетами для политики RR, результат представлен ниже. В данном случае можно лучше продемонстрировать конкуренцию нитей.

В результате эксперимента можно увидеть, как нити с меньшим приоритетом выполняются среди последних, что как раз и показывает конкуренцию нитей.

## Вывод.

В процессе исследования были изучены различные аспекты управления процессами и потоками, включая их создание, приоритеты, планирование и синхронизацию. Были рассмотрены различные подходы к управлению процессами и потоками.

Выводы из данной работы показывают, что управление процессами и потоками играет ключевую роль в эффективном функционировании операционных систем. Необходимо правильно настроить приоритеты и распределение ресурсов между процессами и потоками, чтобы максимизировать производительность и избежать конфликтов между ними.

## Список литературы.

Душутина Е.В. Системное программное обеспечение. Межроцессные взаимодействия в операционных системах: учеб. пособие. – СПб.: Изд-во Политехн. ун-та, 2016. – 180 с.