**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; взаимодействие родственных процессов; управление процессами посредством сигналов.**

|  |  |  |
| --- | --- | --- |
| Студент гр. 1384 |  | Сочков И.С. |
| Преподаватель |  | Душутина Е.В. |

Санкт-Петербург

2023

## **Цель работы.**

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

**Задание.**

Порождение и запуск процессов

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.

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 отличить от прочих, перечислите назначение самых  
важных из них.

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

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

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

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

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

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

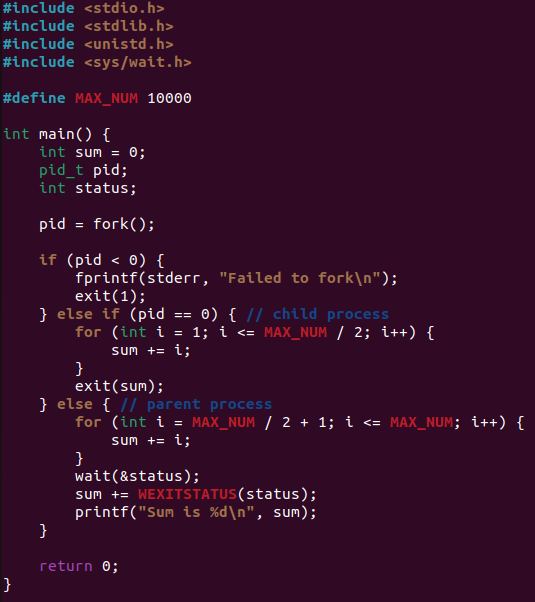
24. Проанализируйте процедуру планирования для процессов и потоков одного процесса. 24.1. Обоснуйте результат экспериментально. 24.2. Попробуйте процедуру планирования изменить. Подтвердите экспериментально, если изменение возможно. 24.3. Задайте нитям разные приоритеты программно и извне (объясните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

**Ход работы:**

**Порождение и запуск процессов**

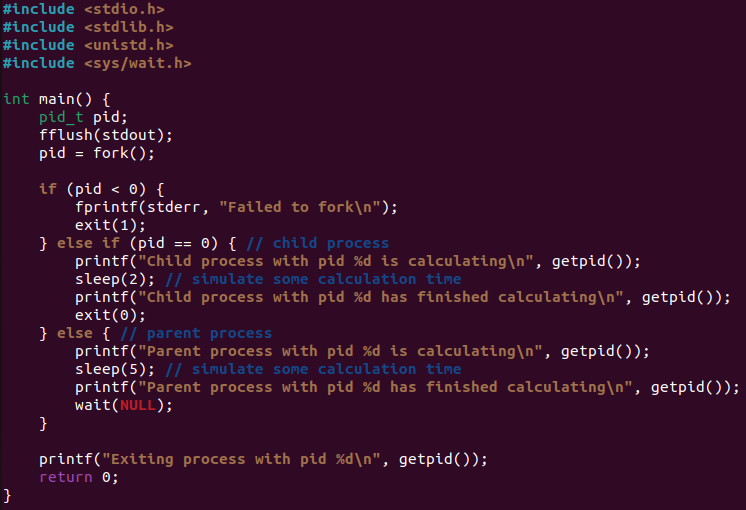
1. Создадим программу, которая использует функции fork(), wait(), exit() для создания процесса-потомка и выполняет вычисления с использованием псевдораспараллеливания.

Ниже приведен пример программы на языке C, которая вычисляет сумму всех чисел от 1 до 10000. Она использует функцию fork() для создания процесса-потомка, который будет обрабатывать половину чисел, а родительский процесс будет обрабатывать другую половину. После завершения вычислений процесс-потомок отправляет результат родительскому процессу, который выводит общий результат на экран.



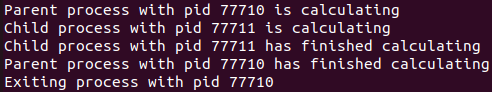
В этом примере родительский процесс и процесс-потомок используют разные части цикла для вычисления суммы чисел. После завершения вычислений процесс-потомок использует функцию exit() для отправки результата родительскому процессу. Родительский процесс ждет завершения процесса-потомка с помощью функции wait() и затем добавляет результаты вычислений и выводит общий результат на экран.

1. Создаем программу, чтобы сначала выполнялись однократные вычисления в каждом процессе, а затем родительский процесс выводил сообщение о завершении программы. Каждый процесс также будет выводить сообщение, идентифицирующее его.



В этой программе мы используем функцию getpid() для вывода идентификатора текущего процесса. Каждый процесс сначала выводит сообщение о начале вычислений, затем задерживается на 2 секунды, чтобы симулировать некоторое время вычислений, и затем выводит сообщение о завершении вычислений. После этого каждый процесс выводит сообщение о завершении.

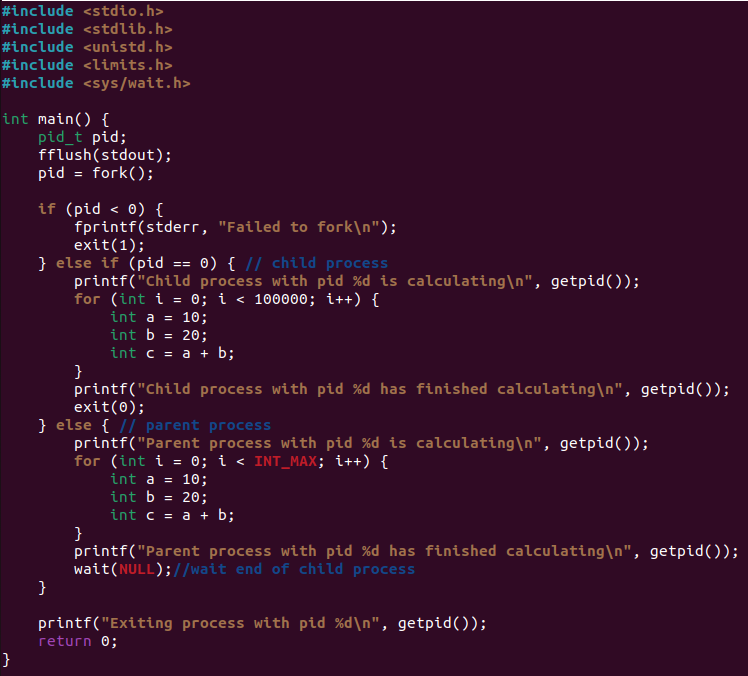
При запуске программы мы увидим следующий вывод:



Как мы видим, сначала запускается родительский процесс, который выводит сообщение о начале вычислений. Затем он создает процесс-потомок, который также выводит сообщение о начале вычислений. Затем дочерний процесс задерживается на 2 секунды, а родительский – на 5 секунд, чтобы симулировать некоторое время вычислений, и затем каждый процесс выводит сообщение о завершении вычислений. Наконец, каждый процесс завершается и выводит сообщение о завершении.

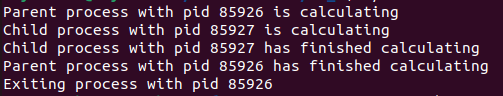
Из этого вывода мы видим, что процесс-потомок завершился раньше, чем родительский процесс, и это произошло потому, что процесс-потомок выполнялся на другом процессорном ядре.

1. Меняем однократные вычисления на циклы, длительность  
   исполнения которых достаточна для наблюдения конкуренции процессов за процессорный ресурс:



Каждый процесс по-прежнему выводит сообщения о начале и завершении вычислений, идентифицирующие текущий процесс.

При запуске программы мы увидим следующий вывод:



Как и ранее, родительский процесс сначала выводит сообщение о начале вычислений, затем создает процесс-потомок, который также выводит сообщение о начале вычислений. Затем каждый процесс выполняет свой цикл вычислений, который занимает некоторое время. Поскольку процессы конкурируют за процессорный ресурс, то каждый процесс будет задерживаться на некоторое время, чтобы ожидать выделения процессорного времени. Наконец, каждый процесс завершится и выведет сообщение о завершении.

Из этого вывода мы видим, что процесс-потомок снова завершился раньше, чем родительский процесс, и это произошло потому, что процесс-потомок выполнялся на другом процессорном ядре и получал более высокий приоритет из-за конкуренции за процессорный ресурс.

1. Функция sched\_setscheduler устанавливает алгоритм и параметры планирования процесса с номером pid. Если pid равен нулю, то будет задан алгоритм вызывающего процесса. Тип и значение аргумента p зависят от алгоритма планирования.

Сегодня в Unix подобных ОС, в частности в Linux, и других ОС, следующих стандарту POSIX, поддерживаются три базовые политики планирования: SCHED\_FIFO, SCHED\_RR, и SCHED\_OTHER: одна для обычных процессов и две для процессов «реального» времени. Их реализация обеспечивается ядром, а точнее, планировщиком. Каждому процессу присваивается статический приоритет sched\_priority, который можно изменить только при помощи системных вызовов. Ядро хранит в памяти списки всех работающих процессов для каждого возможного значения sched\_priority, а это значение может находиться в определенном интервале, заданном для данной конкретной реализации ОС. Для того, чтобы определить, какой процесс будет выполняться следующим, планировщик ищет непустой список (очередь) с наибольшим статическим приоритетом и запускает первый процесс из этого списка. Алгоритм планирования определяет, как процесс будет добавлен в список-очередь с тем же статическим приоритетом, и как он будет перемещаться внутри этого списка. SCHED\_OTHER – это используемый по умолчанию алгоритм со стандартным разделением времени, с которым работает большинство процессов. SCHED\_FIFO и SCHED\_RR предназначены для процессов, которым необходим более четкий контроль над порядком исполнения процессов. Для каждой политики – свои значения и диапазон статических приоритетов, которые могут зависеть от конкретного типа системы, с которой вы работаете. Например, для Linux Debian статический приоритет процессов с алгоритмом SCHED\_OTHER равен нулю, а статические приоритеты процессов с алгоритмами SCHED\_FIFO и SCHED\_RR могут находиться в диапазоне от 1 до 99. Статический приоритет, больший, чем 0, может быть установлен только у суперпользовательских процессов, то есть только эти процессы могут иметь алгоритм планировщика SCHED\_FIFO или SCHED\_RR.

Для того, чтобы узнать возможный диапазон значений статических приоритетов данного алгоритма планировщика, необходимо использовать функции sched\_get\_priority\_min и sched\_get\_priority\_max. Планирование является упреждающим: если процесс с большим статическим приоритетом готов к запуску, то текущий процесс будет приостановлен и помещен в соответствующий список ожидания. Политика планирования определяет лишь поведение процесса в очереди (списке) работающих процессов с тем же статическим приоритетом.

1) SCHED\_FIFO: планировщик FIFO (First In-First Out). Алгоритм SCHED\_FIFO можно использовать только со значениями статического приоритета, большими нуля. Это означает, что если процесс с алгоритмом SCHED\_FIFO готов к работе, то он сразу запустится, а все обычные процессы с алгоритмом SCHED\_OTHER будут приостановлены. SCHED\_FIFO - это простой алгоритм без квантования времени. Процессы, работающие согласно алгоритму SCHED\_FIFO подчиняются следующим правилам: процесс с алгоритмом SCHED\_FIFO, приостановленный другим процессом с большим приоритетом, останется в начале очереди процессов с равным приоритетом, и его исполнение будет продолжено сразу после того, как закончатся процессы с большими приоритетами. Когда процесс с алгоритмом SCHED\_FIFO готов к работе, он помещается в конец очереди процессов с тем же приоритетом.

Вызов функции sched\_setscheduler или sched\_setparam, который посылается процессом под номером pid с алгоритмом SCHED\_FIFO приведет к тому, что процесс будет перемещен в конец очереди процессов с тем же приоритетом. Как следствие, он может очистить текущий запущенный процесс, если он имеет такой же приоритет. (POSIX 1003.1 рекомендует процессу перейти в конец списка). Процесс, вызывающий sched\_yield, также будет помещен в конец списка. Других способов перемещения процесса с алгоритмом SCHED\_FIFO в очереди процессов с одинаковыми статическими приоритетами не существует. Процесс с алгоритмом SCHED\_FIFO работает до тех пор, пока не будет заблокирован запросом на ввод/вывод, приостановлен процессом с большим статическим приоритетом или не вызовет sched\_yield.

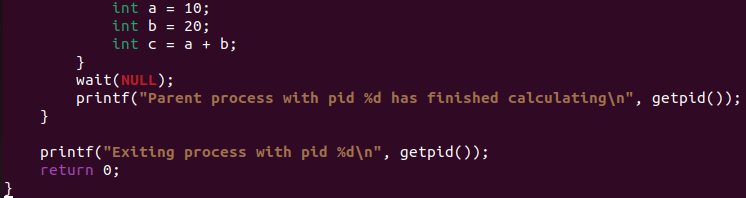
2) SCHED\_RR: циклический алгоритм планирования. Все, относящееся к алгоритму SCHED\_FIFO, справедливо и для SCHED\_RR за исключением того, что каждому процессу разрешено работать непрерывно не дольше некоторого времени, называемого квантом. Если процесс с алгоритмом SCHED\_RR работал столько же или дольше, чем квант, то он помещается в конец очереди процессов с тем же приоритетом. Процесс с алгоритмом SCHED\_RR, приостановленный процессом с большим приоритетом, возобновляя работу, использует остаток своего кванта. Длину этого кванта можно узнать, вызвав функцию sched\_rr\_get\_interval.

3) SCHED\_OTHER: стандартный алгоритм планировщика с разделением времени. Алгоритм SCHED\_OTHER можно использовать только со значениями статического приоритета, равными нулю. SCHED\_OTHER – это стандартный алгоритм планирования Linux с разделением времени, предназначенный для процессов, не требующих специальных механизмов реального времени со статическими приоритетами.

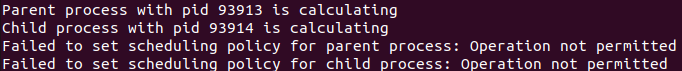
Порядок предоставления процессорного времени процессам со статическим приоритетом, равным нулю, основывается на динамических приоритетах, существующих только внутри этого списка. Динамический приоритет основан на уровне nice (установленном при помощи системных вызовов nice или setpriority) и увеличивается с каждым квантом времени, при котором процесс был готов к работе, но ему было отказано в этом планировщиком. Это приводит к тому, что, рано или поздно, всем процессам с приоритетом SCHED\_OTHER выделяется процессорное время.

Попытаемся изменить приоритеты и текущую политику планирования программы из предыдущего задания. Пример кода программы:

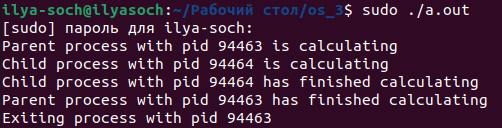




При попытке исполнения программы от имени обычного пользователя, появляется сообщение об ошибке – о том, что операция изменения приоритета запрещена.



Для корректного исполнения необходимы права суперпользователя, тогда запустим программу на правах суперпользователя через sudo:



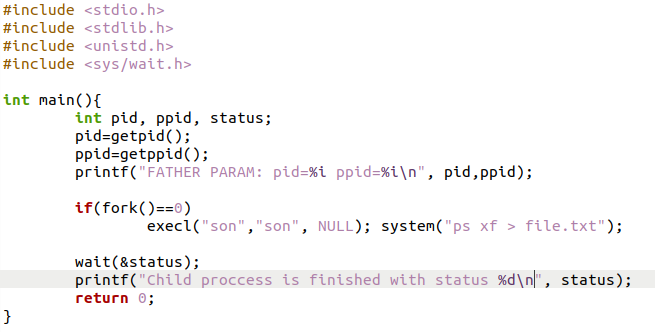
Данный код запускает два процесса: родительский и дочерний. В каждом процессе выполняется цикл вычислений, который занимает некоторое время. После выполнения цикла каждый процесс выводит сообщение о завершении вычислений.

Однако, перед запуском цикла вычислений каждый процесс изменяет свою политику планирования на RR (Round-Robin) и устанавливает приоритет 1. Это означает, что процессы будут выполняться в порядке очереди, а если два процесса готовы к выполнению, то процессор будет переключаться между ними через определенные промежутки времени.

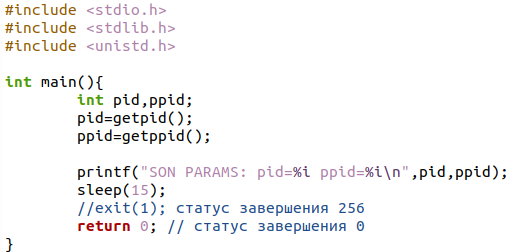
Таким образом, код позволяет демонстрировать работу планировщика задач в операционной системе и влияние на нее изменения политики планирования и приоритета процессов.

1. 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 -aux > file").

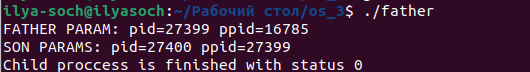
Father.c



Son.c



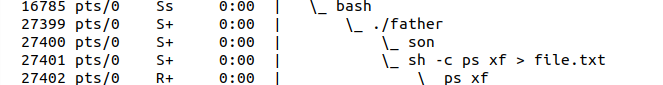
1. Запущен исполнительный файл father, получена информация о процессах, запущенных с терминала.



В соответствии с выводимыми идентификаторами родителем процесса son является процесс father. Эти же параметры записываются в файл в результате выполнения команды

ps –xf (строка программы system("ps xf > file.txt"); ):





Назначение полей:

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:

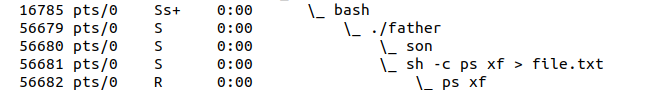
S : процесс ожидает (т.е. спит менее 20 секунд)

s : лидер сессии

R : процесс выполняется в данный момент

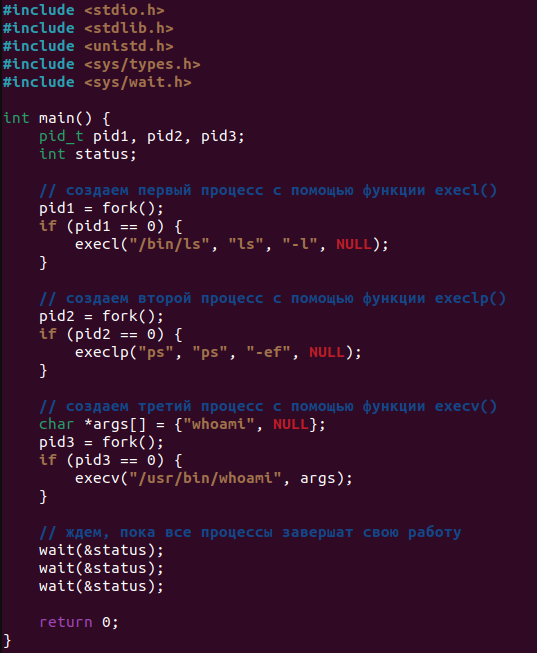
+: порожденный процесс

1. При запуске «родителя» в фоновом режиме, таблица процессов покажет следующую статистику:



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.

1. Для выполнения создания процессов с использованием функций семейства exec() мы можем использовать следующий код на языке программирования C:

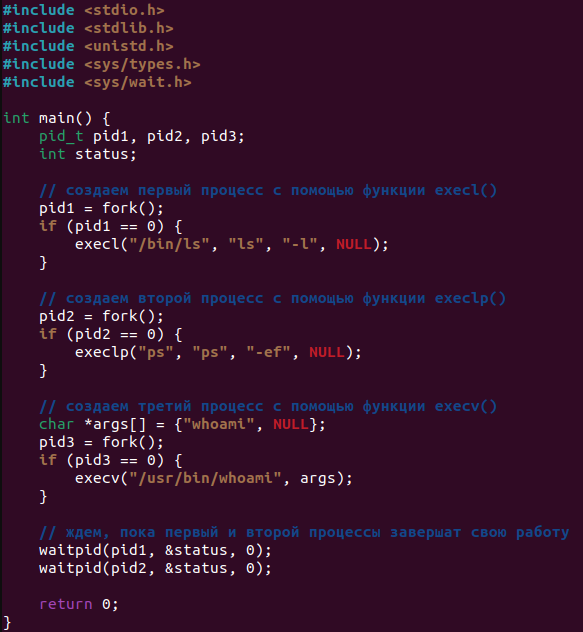


В этом примере мы создаем три процесса с помощью функций семейства exec(). Первый процесс запускает команду "ls -l" с помощью функции execl(), второй процесс запускает команду "ps -ef" с помощью функции execlp(), а третий процесс запускает команду "whoami" с помощью функции execv(). После создания процессов мы используем функцию wait() для ожидания их завершения.

1. Значение, возвращаемое функцией wait(&status), является статусом завершения дочернего процесса. Этот статус может быть использован родительским процессом для определения того, завершился ли дочерний процесс успешно или произошла ошибка. Если статус завершения равен 0, то процесс завершился успешно, в противном случае произошла ошибка.

Чтобы отслеживать только подмножество порожденных потомков, можно использовать функции waitpid() или waitid(). Функция waitpid() позволяет родительскому процессу ожидать завершения конкретного дочернего процесса, указанного по идентификатору процесса (PID). Функция waitid() позволяет ожидать завершения любого дочернего процесса, идентификатор которого соответствует определенным критериям (например, процесс, который первым завершил свою работу).

Вот пример кода, использующего функцию waitpid() для ожидания завершения только двух из трех созданных процессов:



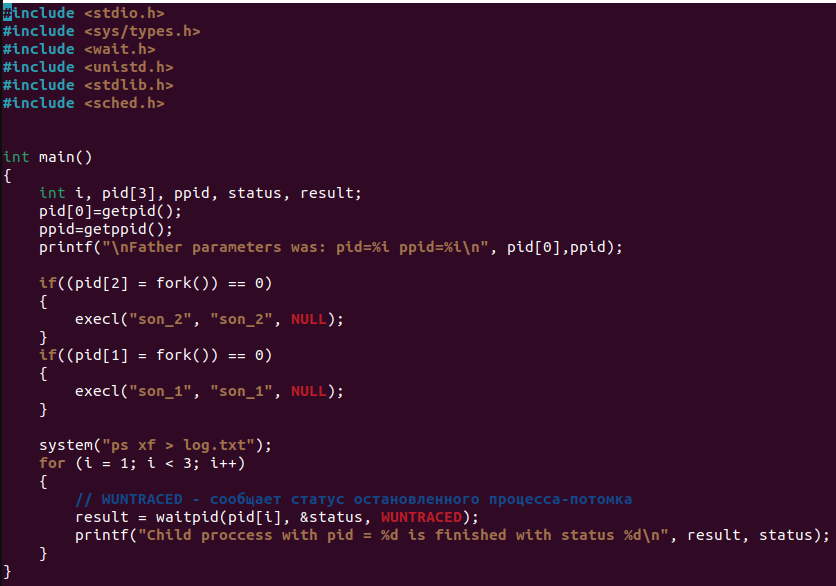
В этом примере мы использовали функцию waitpid() для ожидания завершения только первого и второго процессов. Для этого мы передали идентификаторы процессов pid1 и pid2, которые были возвращены функциями fork(), в качестве первого аргумента функции waitpid(). Второй аргумент функции &status является указателем на переменную, в которую будет записан статус завершения процесса. Наконец, третий аргумент 0 указывает на то, что мы ждем любого статуса завершения процесса.

Таким образом, функции waitpid() и waitid() могут быть использованы для отслеживания завершения только определенных дочерних процессов, а не всех процессов, созданных родительским процессом.

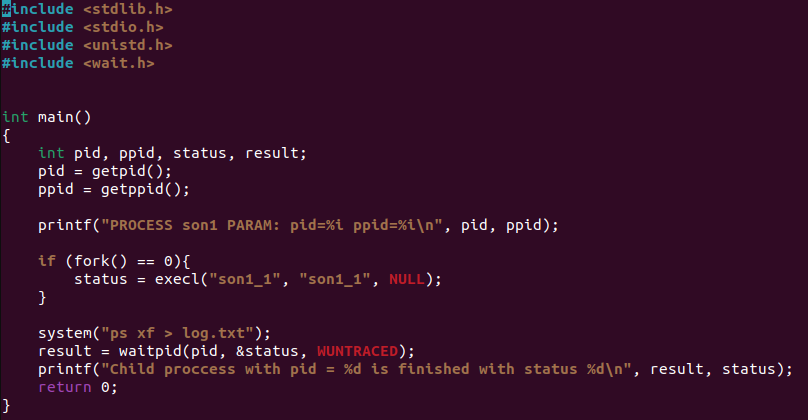
10. Порядок выполнения операций при использовании функций семейства exec() определяется тем, какой процесс был запущен первым и какие процессы зависят от него. Иными словами, приоритет будет у процесса, который был указан первым.

10.1. При вызове функции fork() создается новый дочерний процесс, который является точной копией родительского процесса. Затем дочерний процесс продолжает выполнение кода, следующего за вызовом fork(), в то время как родительский процесс продолжает выполнение кода, предшествующего вызову fork(). Если процесс, который был создан вызовом fork() еще раз вызывает fork() внутри себя, то создается другой дочерний процесс и так далее. Таким образом, порядок выполнения процессов, созданных через вложенные вызовы fork(), будет иметь структуру дерева, где каждый родительский процесс создает свой дочерний процесс, и каждый дочерний процесс может создать другой дочерний процесс. Для проверки этого были созданы 4 программных модуля: father.c, son1.c, son1\_1.c, son2.c. Родительский процесс создает два дочерних процесса: son1 и son2, а son1 в свою очередь создает son1\_1.c:

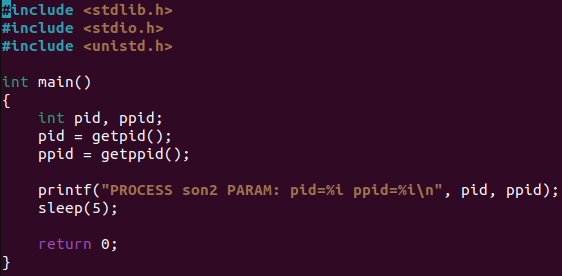
Father.c



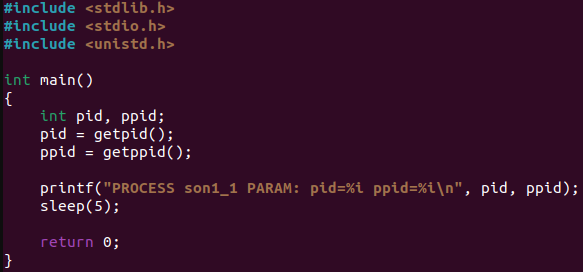
Son\_1.c



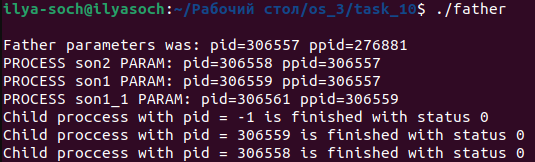
Son\_2.c



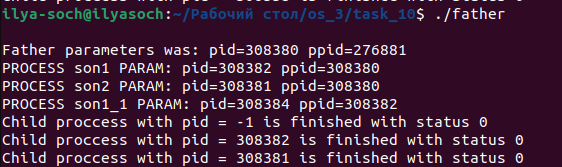
Son\_11.c (порождается son\_1.c)



Если father вызывает сначала son1, то лог исполняемого файла выглядит следующим образом:



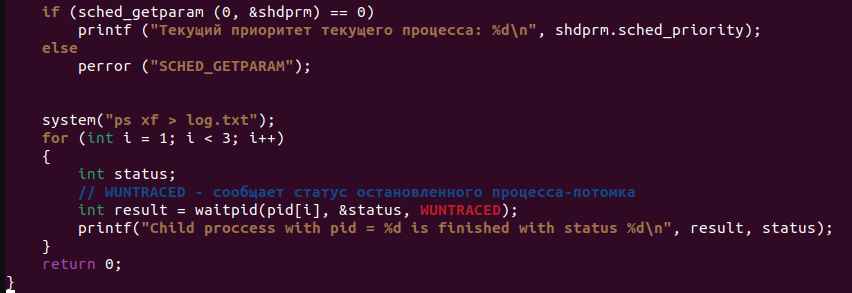
В ином случае, когда сначала вызывается son2.c, лог следующий:



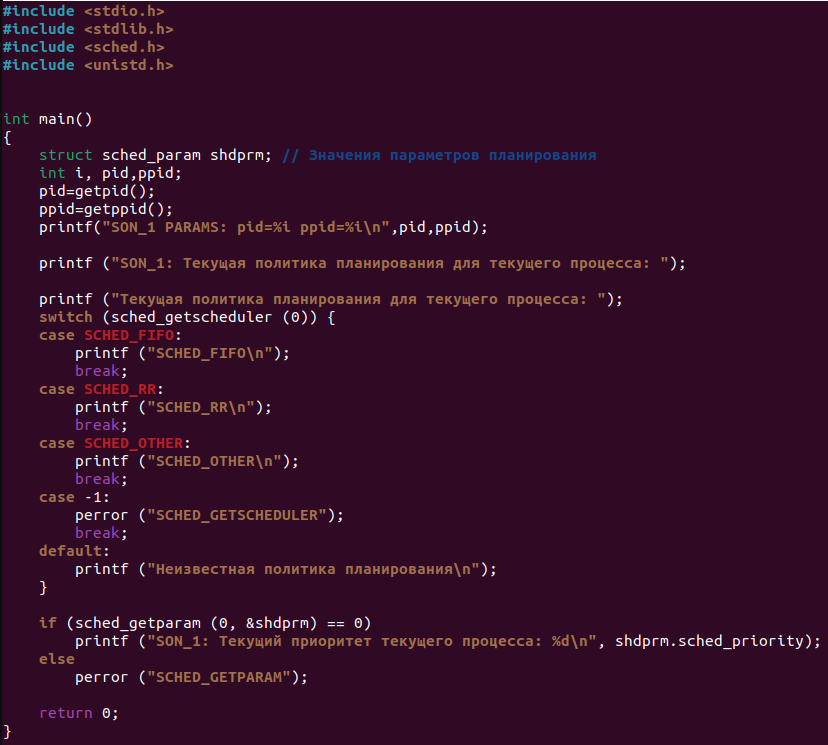
10.2. Изменим процедуру планирования с помощью функции sched\_setscheduler() и повторим эксперимент:

Father.c

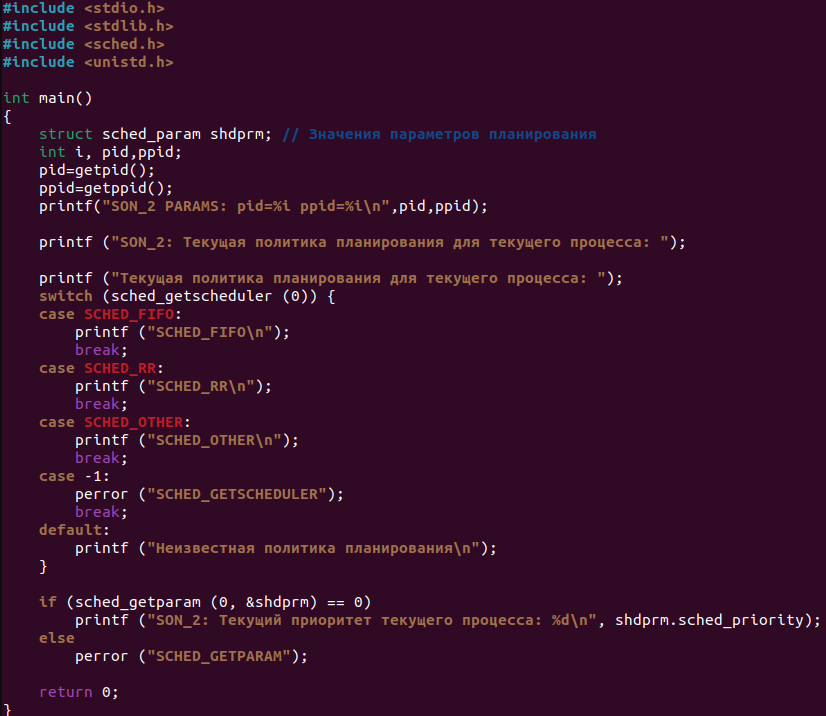




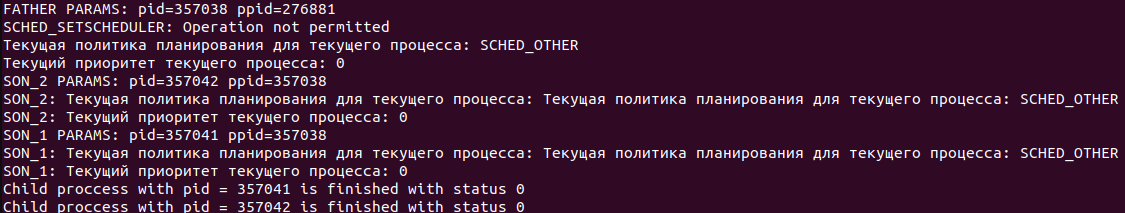
Son1.c



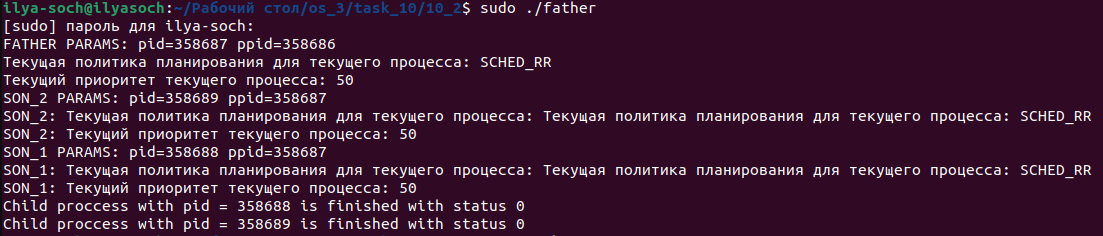
Son2.c



Вывод без прав суперпользователя выглядит следующим образом:



В ином случае (с правами суперпользователя):

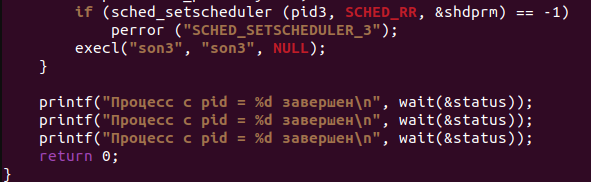


Таким образом, можно утверждать, что потомки наследуют политику планирования и приоритет их родительских процессов.

10.3. Поменяем порядок очереди в RR-процедуре:

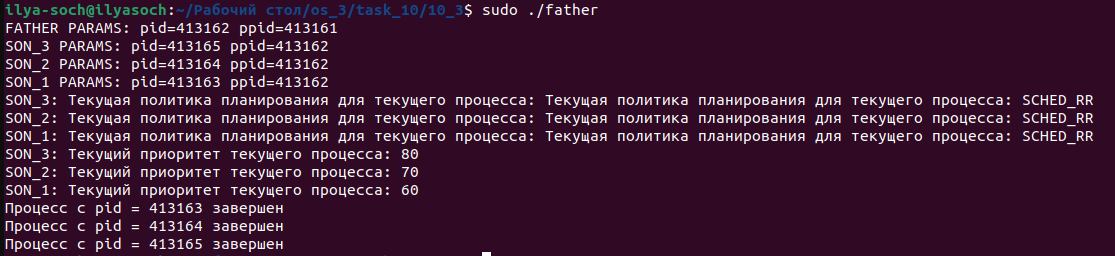
Father.c





Программы, описывающие дочерние процессы, будут аналогичны в задании 10.2.

RR (Round Robin) - это алгоритм планирования процессов, где каждый процесс имеет фиксированное время выполнения, называемое квантом времени. Процессы выполняются по очереди, и если процесс не успевает выполниться за отведенное ему время, то он ставится в конец очереди и ждет своей очереди на выполнение.



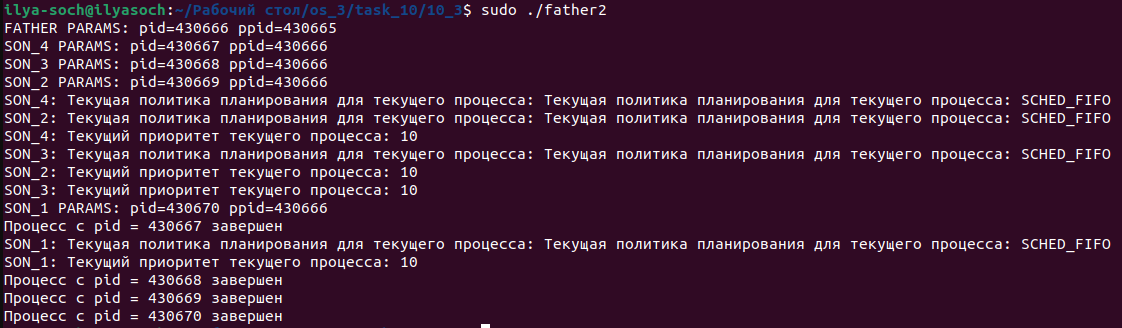
Для вышеуказанного лога мы поменяли в очереди приоритеты для SON3 и SON2, хотя запускаются они в другом порядке - видим, что процесс SON3 не успевает выполниться за отведенное ему время, поэтому встал в конец очереди и завершился последним (pid=413165).

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

Father\_2.c



Заметим, что мы поменяли порядок создания процессов - начиная с son4, заканчивая son1.



10.4. Был проведен эксперимент с планированием FIFO и RR: для этого было создано 15 сыновей, каждый из которых выполняет некоторое арифметическое действие и выводит свой номер на экран. Код программ представлен ниже:

Father\_rr.c



Father\_fifo.c

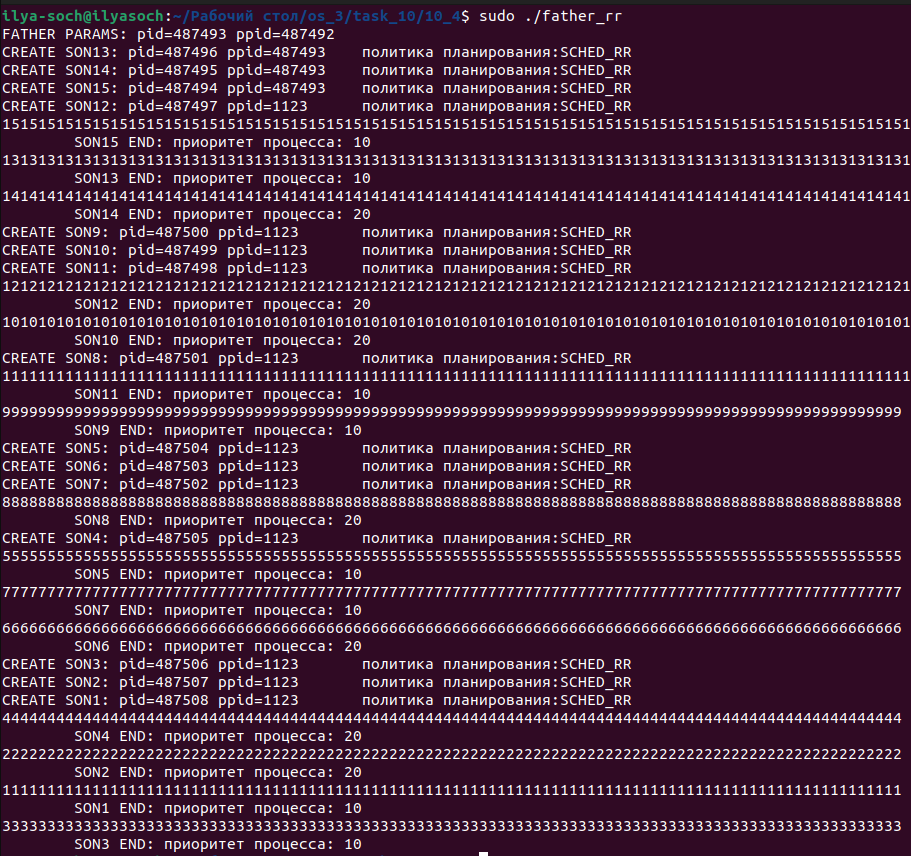


SonN.c (все сыновья аналогичны, их параметры number и name задаются в начале программы main)





Запустив программы, заметим, что в случае политики FIFO сыновья последовательно начинаются и заканчиваются в одном и том же порядке, в то время как при политике RR некоторые процессы меняются местами. Лог запуска программ представлен ниже:



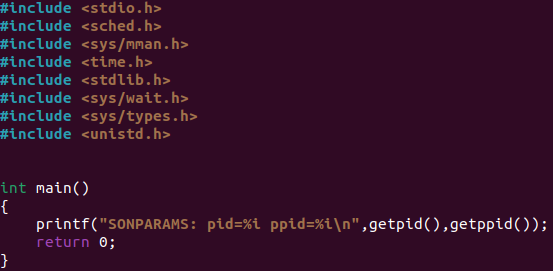


1. Определим величину кванта с помощью функции (POSIX) sched\_rr\_get\_interval():

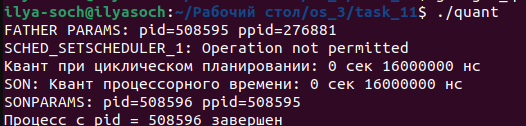
Get\_quant.c



Son.c



Вывод программы:

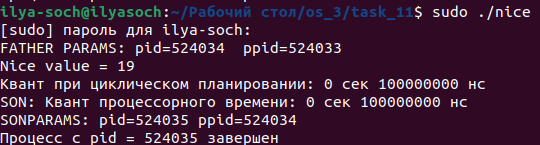


Как можно заметить, длительность кванта равна 0,1 секунды или 10^8 тактов процессора. В настоящее время, в отличие от более ранних версий операционной системы Linux, существует ограничение на установку значения кванта процессорного времени для RR-планировщика приложениями, так как в современных версиях ОС Linux нет специального механизма для этого. В прошлом квант можно было настраивать, регулируя параметр nice процесса: отрицательное значение nice увеличивало квант, а положительное уменьшало его длительность. Однако, в разных версиях ядра системы, степень влияния параметра nice на квант была разной. Начиная с версии Linux 2.6.24, изменение кванта SCHED\_RR стало невозможным документированными средствами. Это можно проверить экспериментально, используя системную функцию nice().

Nice.c



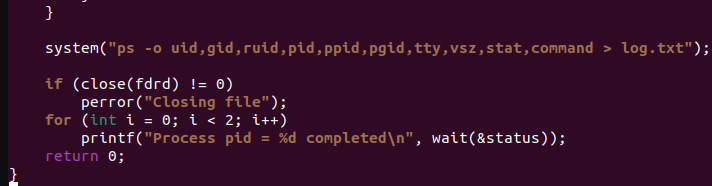
Вывод программы:



1. Давайте рассмотрим, как происходит наследование дескрипторов файлов на этапах вызова функций fork() и exec(). Для этого проведем эксперимент и проверим, имеют ли потомки доступ к файлам, которые были открыты в родительском процессе. Рассмотрим пример кода, в котором дескрипторы открытых файлов (file.txt и out.txt), созданные в родительском процессе, передаются как аргументы дочерним процессам:

Father.c



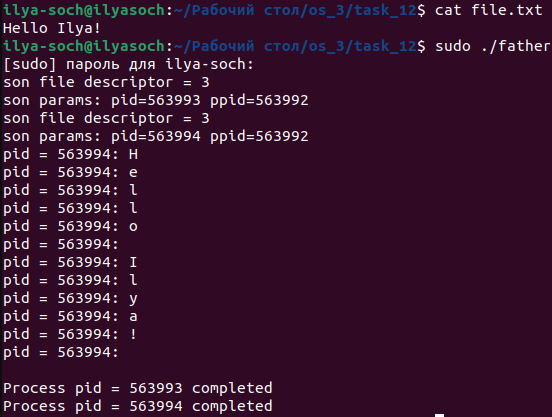


Son.c



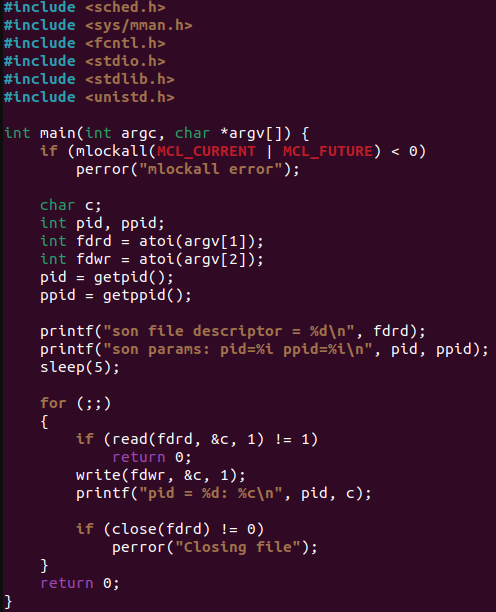
Для проведения эксперимента, мы анализируем наследование на этапах вызова функций семейства fork() и exec(). В данном примере, мы рассматриваем случай, где потомки передаются дескрипторы открытого файла (fdrd) и созданного файла (fdwr), и оба потомка имеют доступ к записям в таблице файлов, соответствующим исходному файлу и файлу вывода соответственно. Кроме того, оба процесса разделяют одно адресное пространство в результате наследования, поэтому они используют одну переменную для хранения считанного из входного файла символа.

Важно отметить, что ядро смещает внутрифайловые указатели после каждой операции чтения или записи, таким образом, оба процесса не обратятся вместе на чтение или запись по одному и тому же указателю или смещению внутри файла. Это особенно важно в случае использования RR-политики с равными приоритетами процессов. В результате выполнения программы, мы получаем определенный результат.

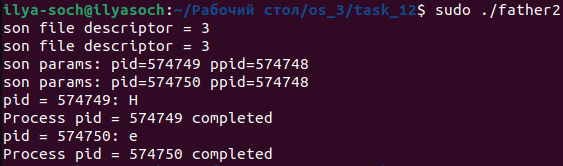


Попробуем закрыть в одном из процессов файл с заданным дескриптором, например, fdrd в son.c:

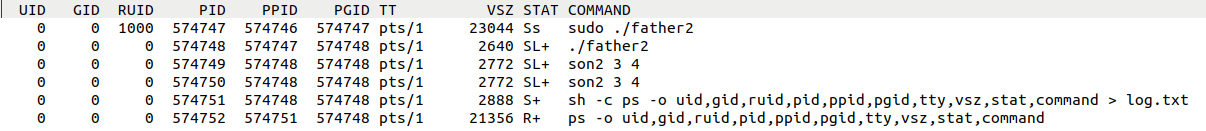
Son2.c



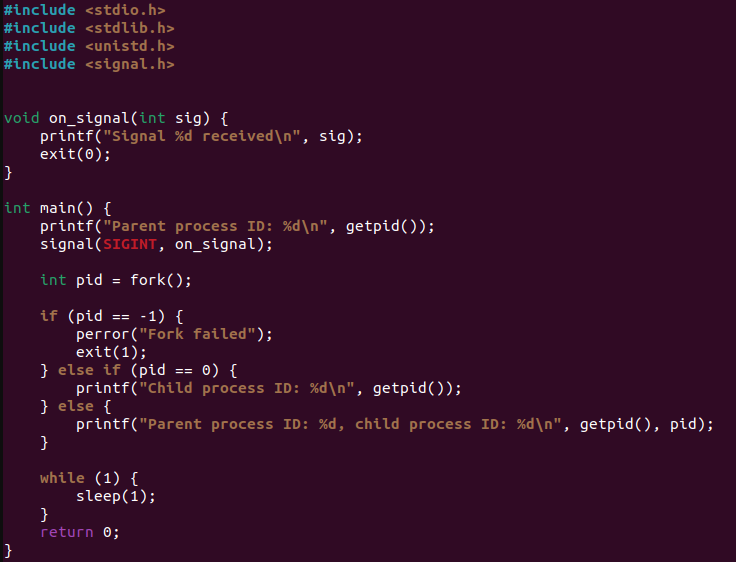
Соответствующий вывод:



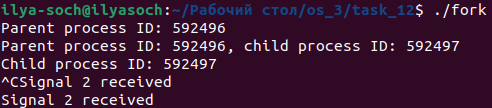
При вызове функций fork() и exec() наследуются следующие параметры: таблица файловых дескрипторов, текущая и корневая директории. Наследование других параметров, таких как UID, GID, RUID, PGID, TTY, при порождении потомков можно проверить, используя утилиту ps с опцией -o uid,gid,ruid,pid,ppid,pgid,tty,vsz,stat,command. Ранее отмечалось, что приоритеты и политика планирования процессов также наследуются от родительского процесса.



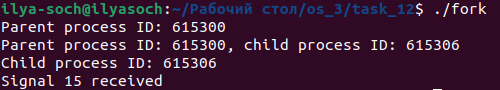
Давайте проанализируем наследование сигналов при вызове функций fork() и exec(). При вызове fork() процесс-потомок наследует все запущенные сигналы от процесса-родителя, включая установленные обработчики. Если обработчик для сигнала не будет установлен в процессе-потомке, то управление будет передано обработчику, установленному в родительском процессе. Однако при вызове exec() происходит полная замена процесса-родителя на новый процесс, созданный в результате выполнения исполняемого файла. В этом случае все установленные обработчики сигналов будут утеряны, и новый процесс получит только дефолтные обработчики сигналов. Если необходимо передать какие-то сигналы в новый процесс, то их нужно устанавливать явно внутри него. Для подтверждения данного утверждения, можно провести эксперимент, поменяв обработчик по умолчанию для сигналов SIGTERM и SIGINT при вызове функции fork(). Затем можно вызвать сигналы командами Ctrl+C и kill из другого терминала и проанализировать результат.



Отправка Ctrl+C:



Отправка команды kill 615300:



Как видно из лога, сигнал получен дважды, хотя обработчик есть только для процесса-родителя. Таким образом, сигналы наследуются при fork().

**Взаимодействие родственных процессов**

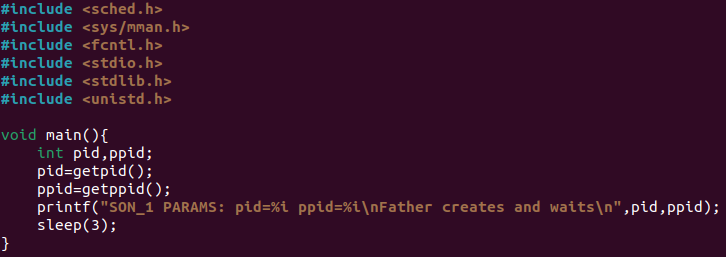
13.1. Мы будем исследовать взаимодействие процессов, которые являются родственными друг другу, то есть процесс порождает другой процесс, а потомок в свою очередь может породить еще один процесс. Взаимодействие этих процессов основано на наследовании определенных характеристик. Для исследования этого взаимодействия мы проанализируем таблицу процессов в трех различных случаях.

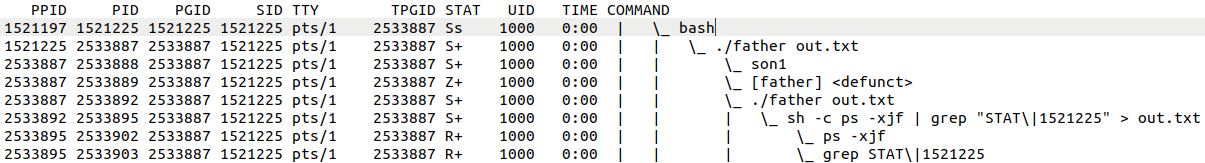
1) Процесс-отец запускает процесс-сын и ожидает его завершения;

father.c



son1.c

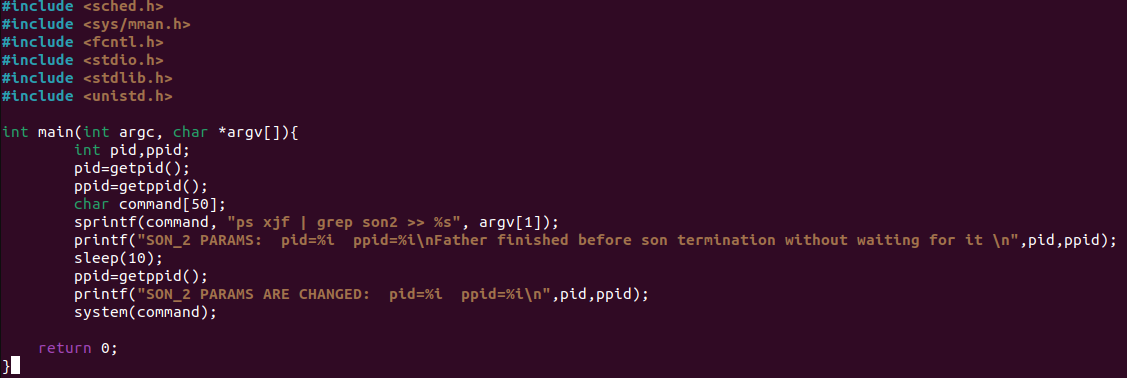


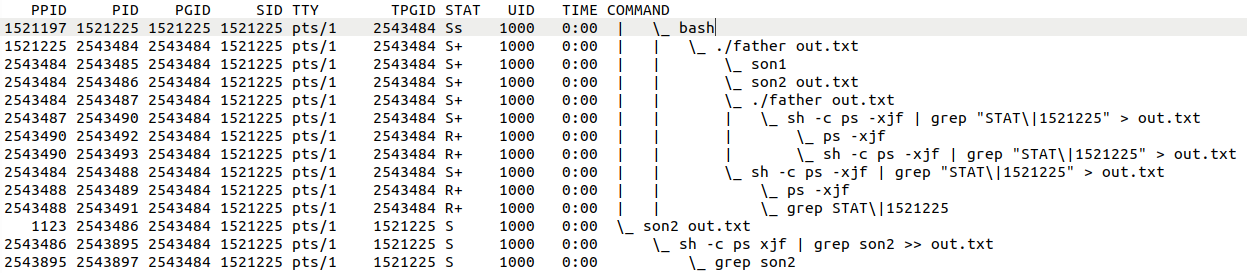


Когда процесс-родитель завершается, на консоли сразу же появляется приглашение на ввод команды, а процесс-потомок son2 продолжает выполняться в фоновом режиме. Поскольку время выполнения son2 значительно дольше, результаты его работы появятся только после появления приглашения на ввод.

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

son2.c

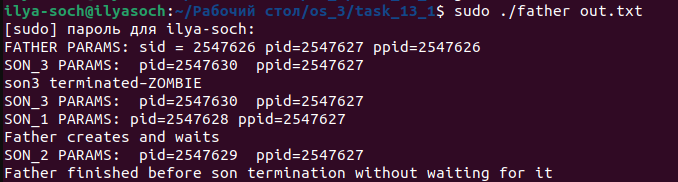




3) Процесс-родитель запускает процесс-потомка и не ожидает его завершения; процесс-сын завершает свое выполнение. Зафиксируйте появление процесса-зомби, для этого включите команду ps в программу father.c.

son3.c

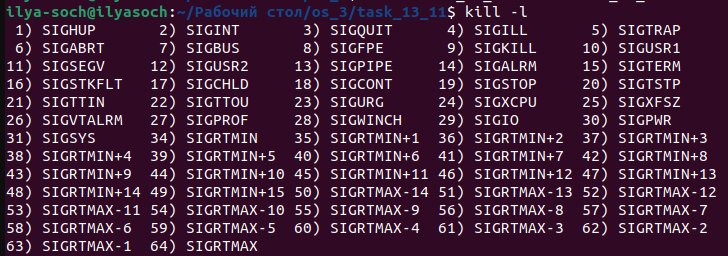




Ранее было замечено, что в таблицах процессов для son3 отображается процесс-зомби с пометкой Z и подписью <defunct>. Процесс-зомби - это процесс, который уже завершил свою работу, но его родительский процесс еще не получил информацию о завершении. Такой процесс остается в системе, но не может выполнять никаких задач, и при этом занимает память и ресурсы. Поэтому он отображается в таблице процессов при использовании утилиты ps.

**Управление процессами посредством сигналов**

13.1. С помощью системного вызова kill можно отправлять сигналы определенным процессам. Если не указан конкретный тип сигнала, по умолчанию отправляется сигнал SIGTERM. Для использования этой утилиты необходимо указать идентификатор процесса в качестве аргумента. Если идентификатор больше нуля, то сигнал будет отправлен процессу с указанным ID. Если идентификатор равен нулю, то сигнал будет отправлен всем процессам, принадлежащим текущему пользователю. Если идентификатор меньше нуля, то он будет рассматриваться как идентификатор группы процессов, и сигнал будет отправлен всем процессам в этой группе.



Функция signal в операционной системе позволяет установить определенное поведение программы при получении сигнала. Для этого можно задать три значения: SIG\_DFL, SIG\_IGN или указатель на пользовательскую функцию обработки. SIG\_DFL означает, что процесс будет реагировать на сигнал стандартным образом, который определен по умолчанию (обычно это приводит к завершению процесса). SIG\_IGN (нельзя использовать для SIGSTOP и SIGKILL) означает, что сигнал будет игнорироваться.

а) Процесс father порождает процессы son1, son2 и son3, и запускает соответствующие исполнительные файлы для их выполнения.

б) Затем родительский процесс управляет потомками, отправляя каждому пользовательскому процессу сигнал.

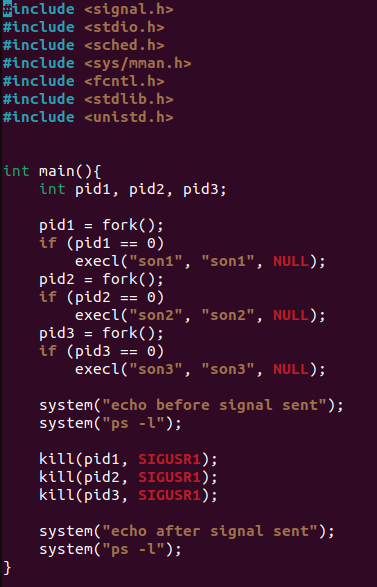
в) В пользовательских процессах-потомках требуется обеспечить:

Для son1 - реагирование на сигнал по умолчанию.

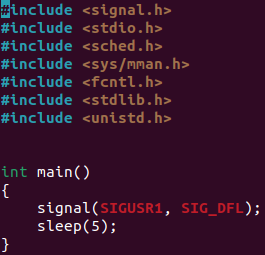
Для son2 - игнорирование сигнала.

Для son3 - перехват и обработку сигнала.

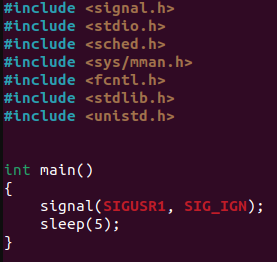
Father.c



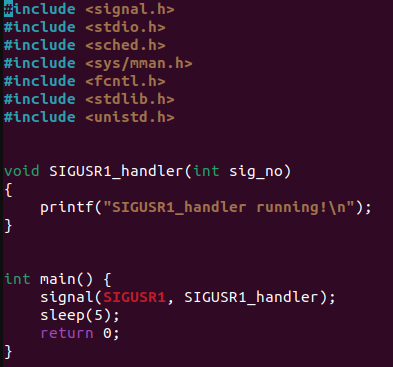
Son1.c



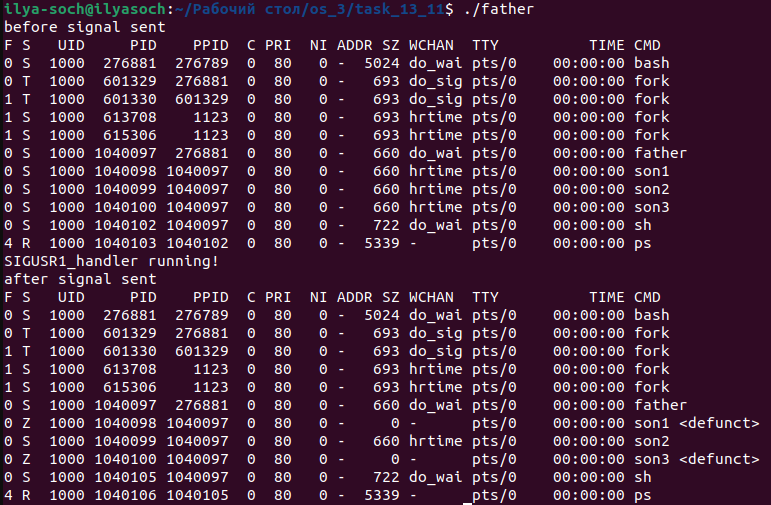
Son2.c



Son3.c

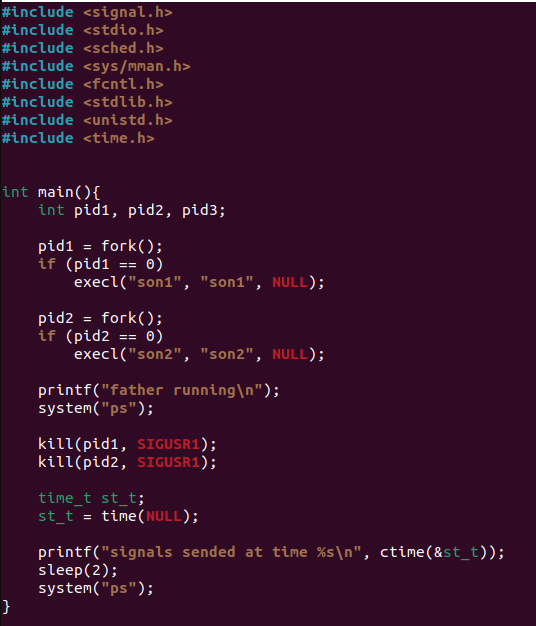


Итоговый вывод:

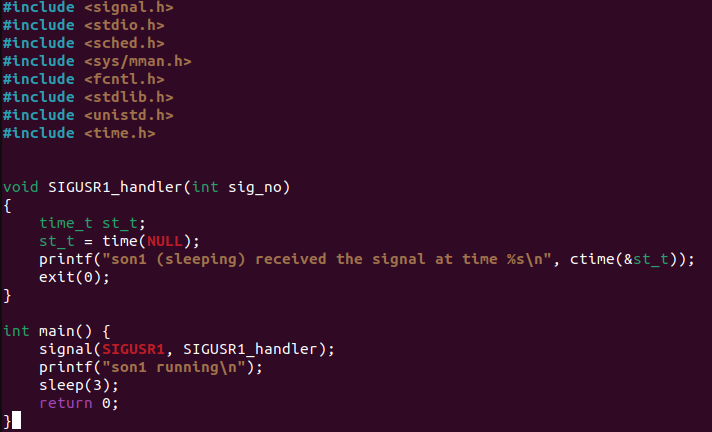


13.2. Для организации посылки сигналов любым двум процессам, находящимся в разных состояниях, был написан следующий код:

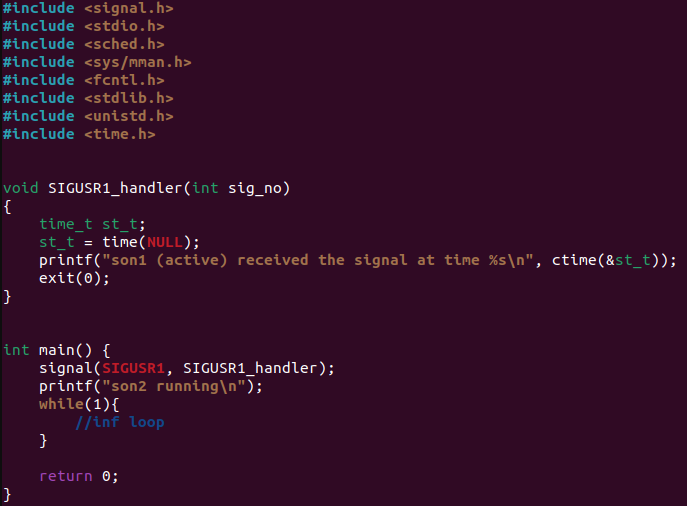
Father.c



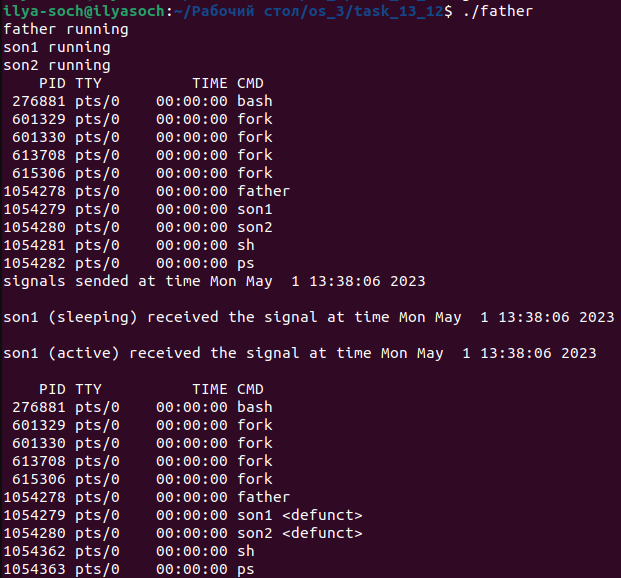
Son1.c



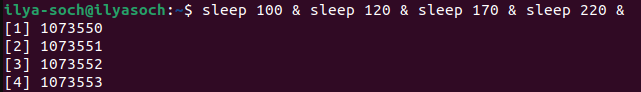
Son2.c



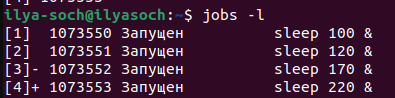
Итоговый вывод:



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



Можно заметить, что задаче "sleep 100" был присвоен порядковый номер выполнения 1 и идентификатор процесса (pid) 1073550, указанные в квадратных скобках. Для того чтобы получить список текущих процессов и их состояния, можно использовать команду jobs -l.



Из приведенной информации можно заметить, что порядковый номер выполнения (job number) и идентификатор процесса (pid) первой задачи (sleep 100) совпадают и указаны в квадратных скобках. Командой "jobs -l" можно получить список запущенных процессов, их статус выполнения (в данном случае "выполняется") и сами команды. Если необходимо вывести информацию о конкретной задаче по ее номеру, можно использовать следующий синтаксис: %номер\_задачи.



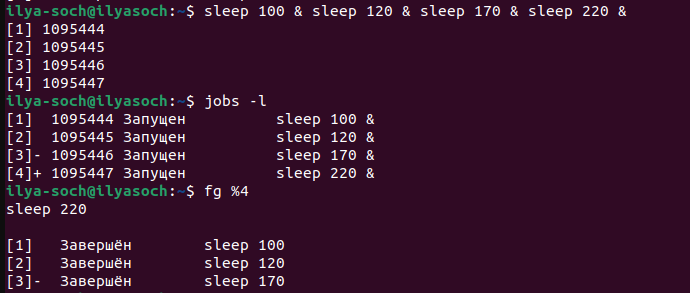
Чтобы вывести последний процесс можно передать вместо номера % .



Давайте повторно вызовем команды в режиме фона и используем команду notify с указанием временных интервалов. Это позволит нам получать уведомления о завершении работы каждого процесса в фоновом режиме в терминале.

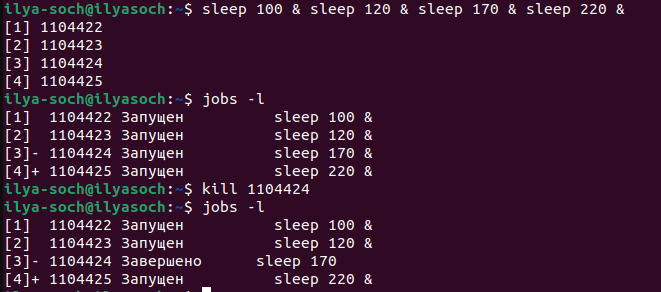


Создадим несколько процессов в фоновом режиме, выведем их список и вернем задачу одну из задач в приоритетном режиме с помощью fg.



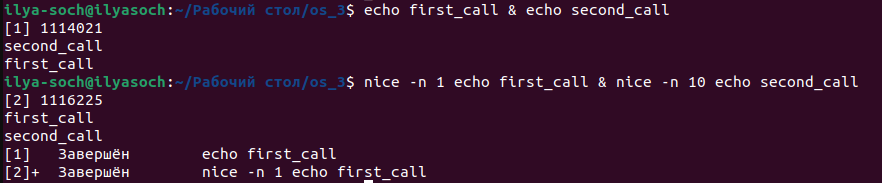
Видно, что при использовании fg задача возвращается в приоритетном режиме - то есть выполняется непосредственно.

Мы создадим несколько задач и запустим их в фоновом режиме, после чего выведем список задач, которые выполняются в фоновом режиме, и с помощью команды kill завершим нужный процесс, передав его pid в качестве аргумента.



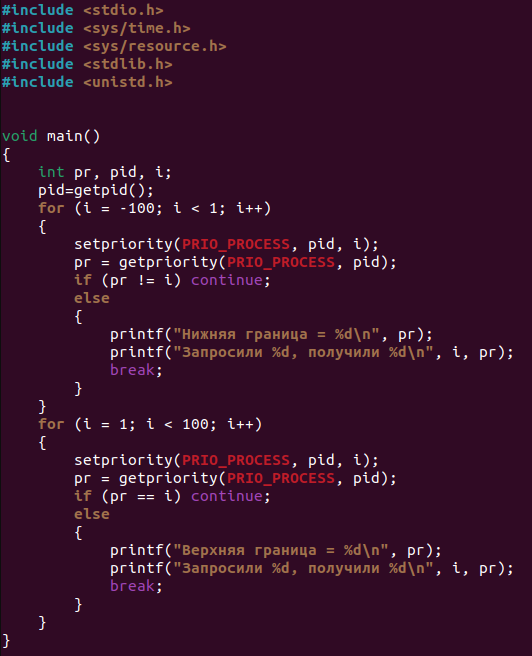
Видно, что статус процесса, после применения утилиты kill изменился на «Завершено», то есть процесс был принудительно завершен.

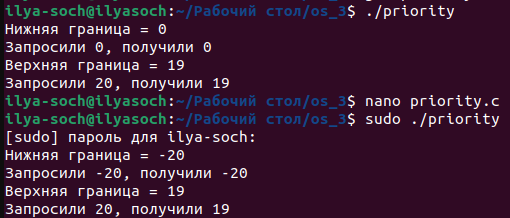
15. Утилита nice используется для запуска программы с измененным приоритетом. Если не указаны аргументы, команда выведет текущий унаследованный приоритет. Если указан приоритет, то команда nice запустит программу с указанным значением. Если не указано смещение приоритета, то приоритет программы увеличится на 10. Утилита nice может смещать приоритет в диапазоне от -20 до 19 включительно, при использовании прав суперпользователя. Когда команда выполняется обычным пользователем, диапазон изменяется от 0 до 19.



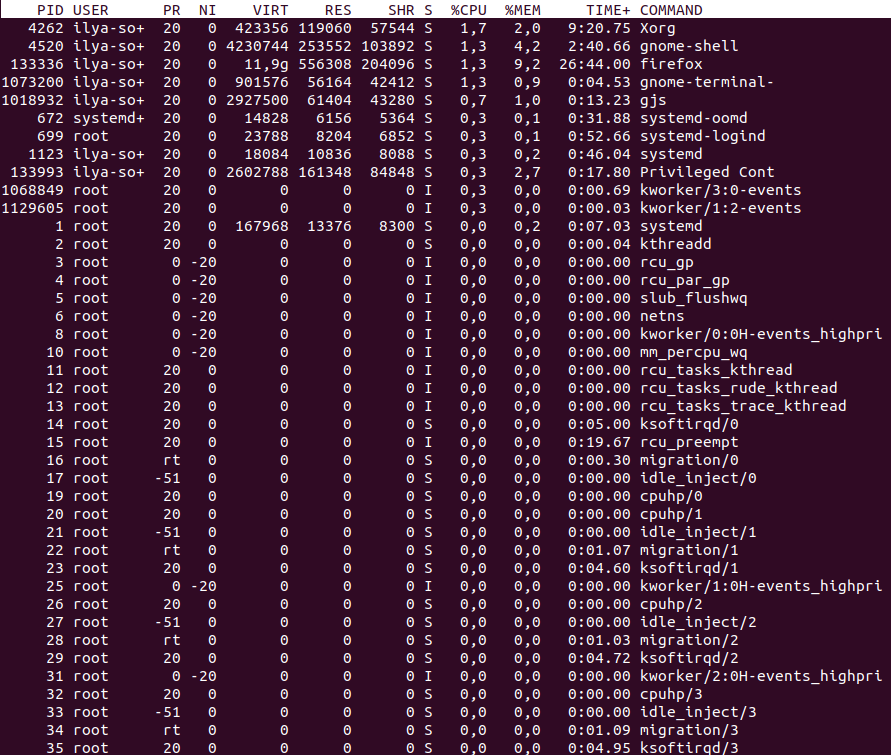
Видно, что при использовании nice меняется приоритет задач.

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



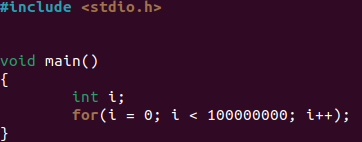


Утилита top позволяет выводить информацию о системе:



Из вышеупомянутых данных сделать вывод, что при запуске приложений из shell пользовательский приоритет равен 20, а системные процессы имеют обычно более высокий приоритет, чем пользовательские, поскольку они обеспечивают работу ядра операционной системы. Приоритеты реального времени используются для некоторых системных процессов, которые должны выполняться в строго заданные временные рамки, но их использование может привести к проблемам с производительностью и стабильностью системы.

16. Утилита nohup(1) предоставляет возможность запуска команды, которая будет продолжать выполняться независимо от потери связи (hungup), а также направляет вывод этой команды в файл nohup.out вместо терминала. Это позволяет запустить команду в фоновом режиме и продолжать ее выполнение даже после выхода пользователя из системы.



Скомпилируем и выполним:



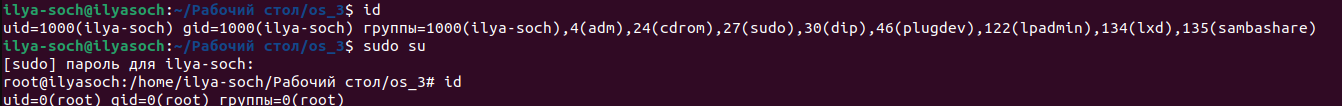
Процесс с идентификатором (PID) 18871 был запущен с использованием команды "nohup", которая игнорирует входные данные и записывает выходные данные в файл "nohup.out", если таковые будут выводиться программой. После выхода из системы, можно использовать команду "ps xa | grep "./nohup"" для поиска процессов, связанных с выполнением данной команды.



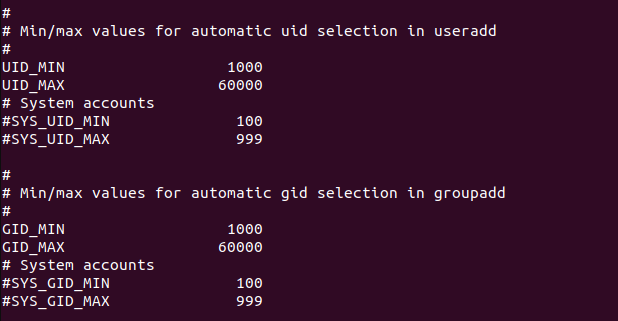
Даже при выходе из системы процесс остается.

17. Допустимые значения UID зависят от конкретной операционной системы. В большинстве случаев для UID разрешается использовать значения от 0 до 65535, с учетом следующих правил: UID суперпользователя всегда равен 0; пользователю nobody обычно присваивается максимальное возможное значение UID или одно из системных значений (nobody - это имя пользователя, который не является владельцем ни одного файла, не принадлежит ни к одной из привилегированных групп и не имеет никаких дополнительных прав, кроме стандартных для обычных пользователей). UID в диапазоне от 1 до 100 обычно зарезервированы для системных нужд, некоторые операционные системы рекомендуют резервировать UID с 101 по 499 или 999.

Для того, чтобы узнать UID процесса, можно использовать утилиту ps. Кроме того, для просмотра текущих параметров пользователя и суперпользователя можно воспользоваться командой id:



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

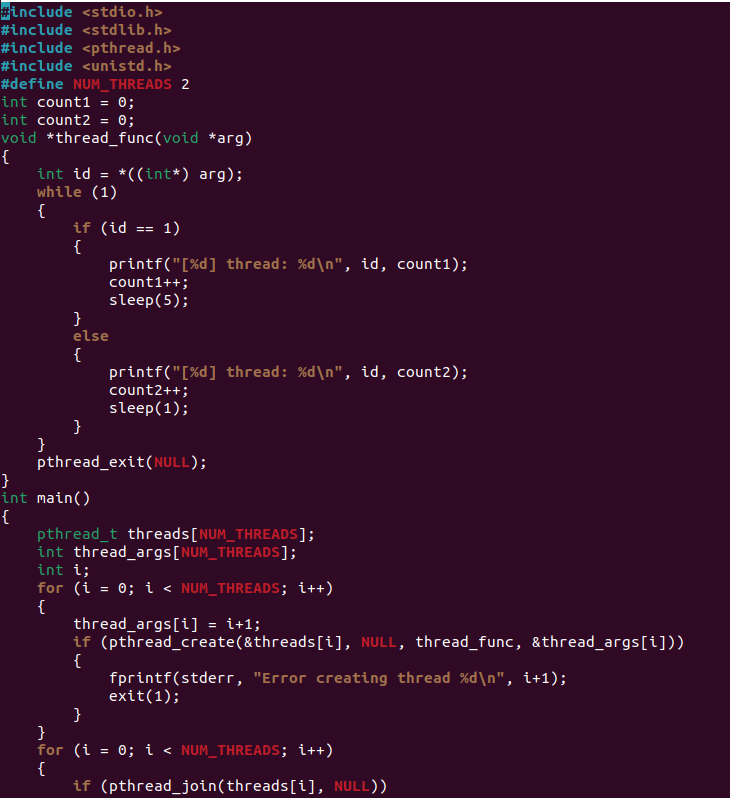


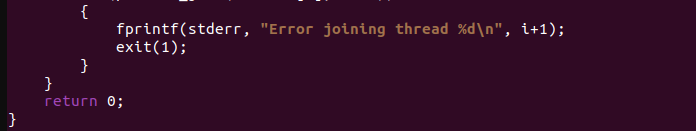
Обычно первый процесс, запускаемый ядром операционной системы, имеет pid равный 1 и называется init. Максимальное значение pid ограничено максимальным значением целочисленного типа, используемого для хранения pid. В Linux, например, максимальное значение pid может быть до 4194303.

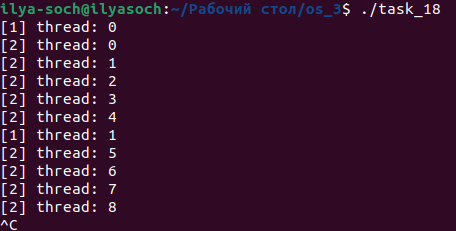
Принадлежность процессов к группам может указывать на то, являются ли они системными или пользовательскими. Обычно системные процессы запускаются от имени пользователя root. Некоторые из важнейших системных процессов включают в себя:

* init: первый процесс, который инициализирует систему и запускает другие процессы;
* systemd: системный менеджер, управляющий процессами и службами в Linux;
* kernel: ядро операционной системы, которое устанавливает взаимодействие между аппаратным и программным обеспечением;
* cron: системный планировщик задач, позволяющий запускать задачи в определенное время или с определенной периодичностью;
* sshd: демон, обеспечивающий удаленный доступ к системе по протоколу SSH;
* syslogd: демон, отвечающий за запись системных сообщений в журналы;
* udevd: демон, отвечающий за автоматическое определение и настройку устройств в системе;
* networkd: демон, отвечающий за управление сетевыми интерфейсами и соединениями в Linux.

18. Разработана программа, формирующая 2 нити. Программный код следующий:

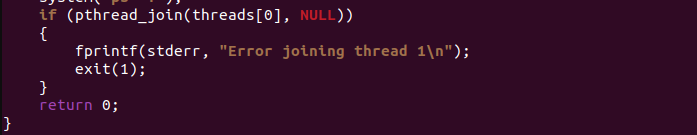




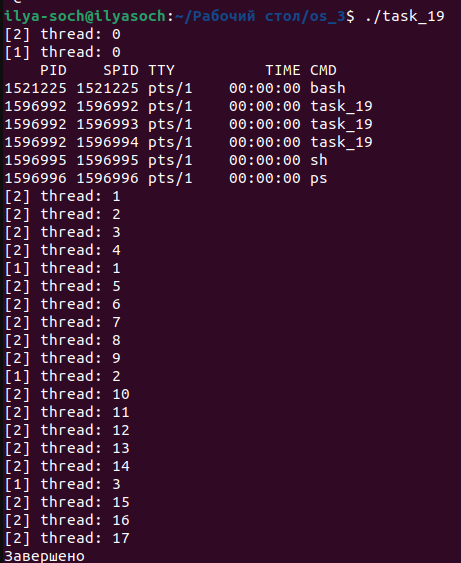


19. Видоизменим код, добавив вывод в таблицу ps:



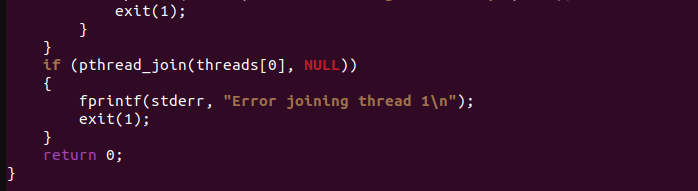


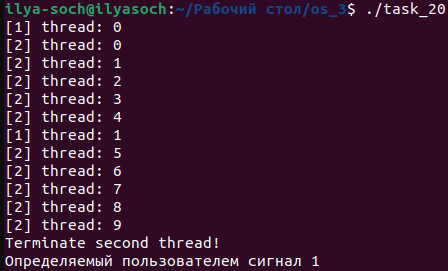
Запускаем программу и закончим процесс командой kill PID из другого терминала:



20. Изменена программа из пункта 18 таким образом, чтобы управление второй нитью осуществлялось по сигналу SIGUSR1, передаваемому из первой нити. На 5-й секунде работы приложения вторая нить удаляется с помощью функции pthread\_kill(t2, SIGUSR1).



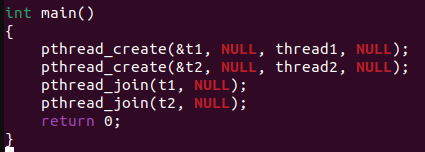


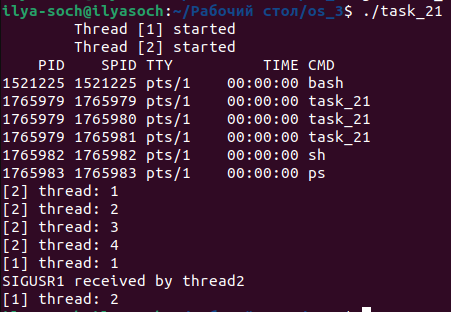


В независимости от выбранного способа удалить одну нить не удалось - удаляется весь процесс в целом.

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





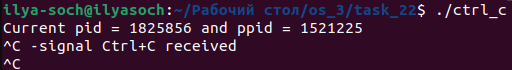


С помощью такого управления удалось корректно организовать удаление одной нити: все остальные нити данного процесса процесса сохранились и продолжили выполняться.

22. Написана программа, позволяющая перехватывать сигнал, генерируемый в результате нажатия клавиш Ctrl+C:

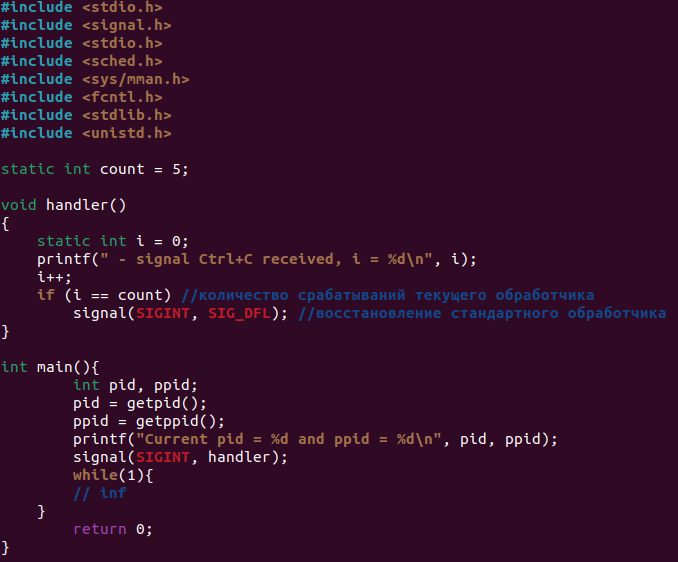


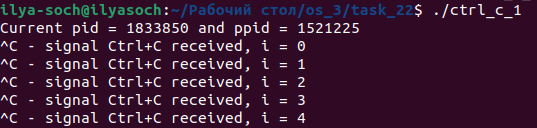
Работа программы представлена ниже:



После перехвата сигнала SIGINT, его обработчик выводит в терминал сообщение о получении сигнала и возвращает оригинальный обработчик. Это приводит к принудительному завершению работы программы. Если повторно нажать на сочетание клавиш, текущая программа будет прервана.

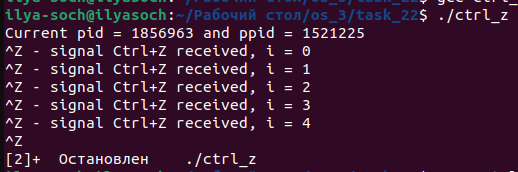
Немного скорректируем функцию обработчика handler() для многократного применения:





Проделаем аналогичную работу для другой комбинации клавиш, например, Ctrl+Z:





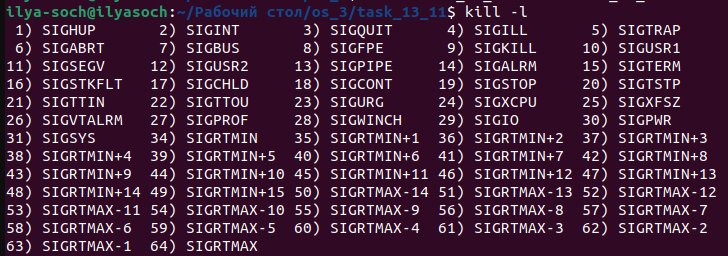
Попробуем удалить нить с использованием команды Ctrl+C:





Запустив данную программу убеждаемся, что удаление одной нитки командой Ctrl+C невозможна - прерывается весь процесс.

23. Вывести весь список сигналов можно с помощью команды kill -l:



Приведен список некоторых сигналов и краткая информация о них.

SIGHUP (сигнал под номером 1) информирует программу о потере связи с управляющим терминалом, и программа по умолчанию завершается после получения этого сигнала.

SIGINT (сигнал под номером 2) посылается программе, выполняемой на терминале, с помощью комбинации клавиш Ctrl+C для прерывания программы.

SIGILL (сигнал под номером 4) возникает, когда передача управления происходит через неинициализированный или поврежденный указатель.

SIGFPE (сигнал под номером 8) происходит из-за переполнения типа данных.

SIGKILL (сигнал под номером 9) вызывает немедленное завершение процесса, и он не может быть перехвачен или проигнорирован.

SIGUSR1 и SIGUSR2 (сигналы под номерами 10 и 12 соответственно) используются для межпроцессной (или межпоточной) синхронизации, управления фоновыми процессами и т.д.

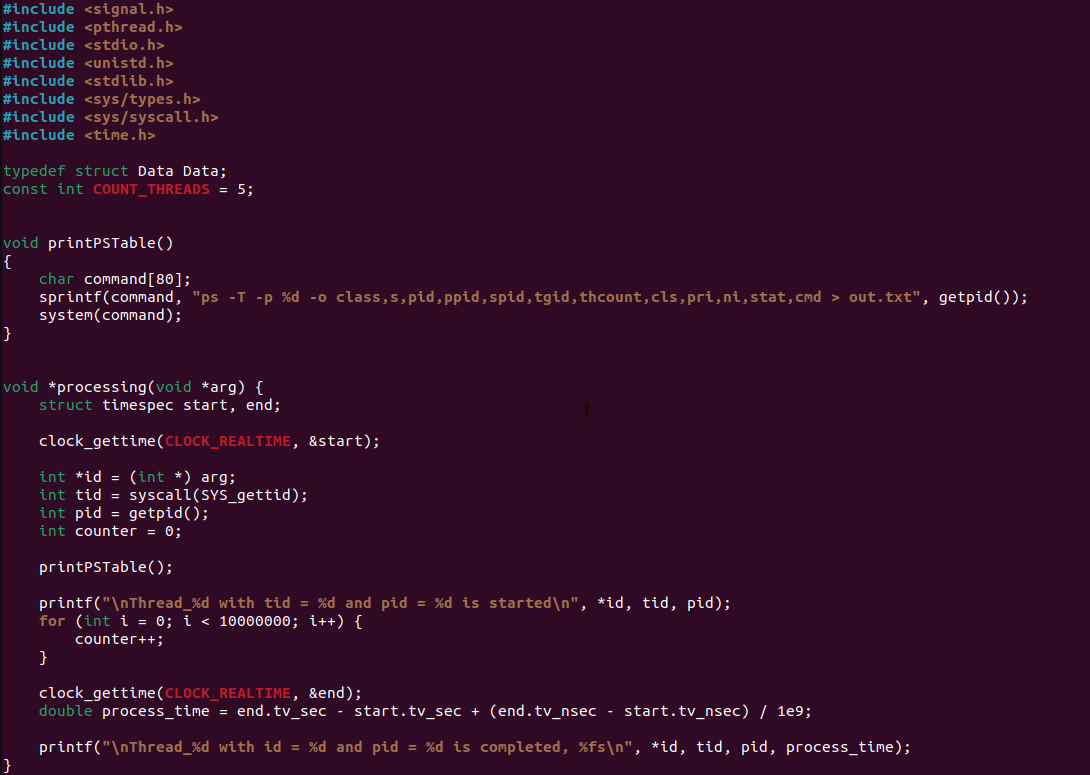
SIGSTOP (сигнал под номером 19), посланный процессу приостанавливает его выполнение. Выполнение процесса будет возобновлено только после получения сигнала SIGCONT.

SIGTSTP (сигнал под номером 20) посылается процессу при нажатии комбинации «стоп» (обычно Ctrl-Z) на управляющем терминале, и, по умолчанию, приостанавливает его выполнение. Выполнение процесса будет возобновлено только после получения сигнала SIGCONT.

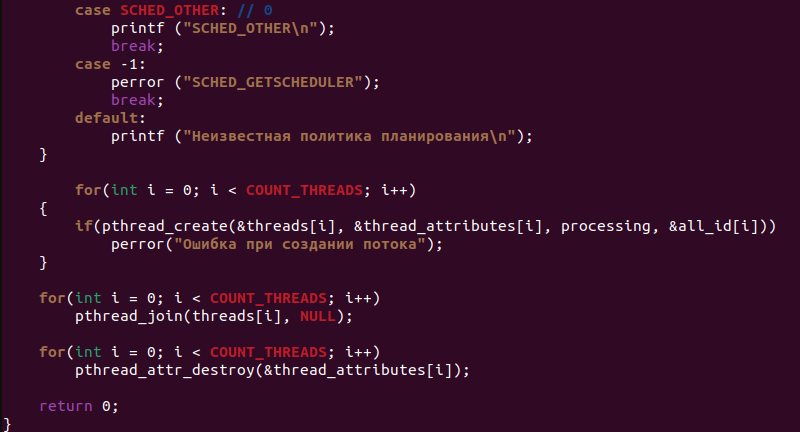
Сигналы реального времени, начиная с 32-го и до 64-го, используются для межпроцессорного взаимодействия и синхронизации в многопоточных системах и системах реального времени. Они ставятся в очередь и обрабатываются в порядке поступления.

Например, SIGPWR (номер сигнала 33) отправляется системой при возникновении событий, связанных с электропитанием, таких как сбои питания или низкий уровень заряда батареи. Программа может перехватывать этот сигнал, чтобы корректно завершить свою работу в случае отключения питания.

24. Для анализа процедуры планирования процессов и потоков внутри одного процесса была разработана программа, которая считывает приоритеты и политики планирования для каждого потока. Результат выводится в файл out.txt в формате, аналогичном выводу команды ps.



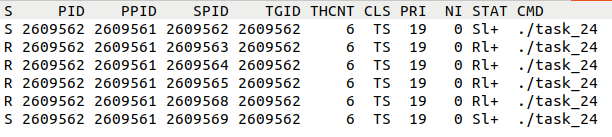




Для политики SCHED\_FIFO результат следующий:



Таблица выглядит следующим образом:



Для политики планирования SCHED\_RR вывод следующий:

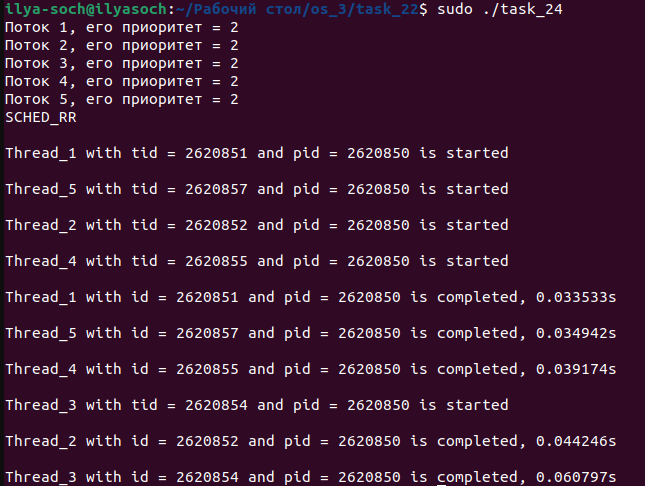
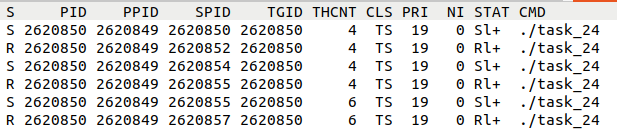


Таблица выглядит следующим образом:



Рассматривая таблицы различных политик планирования и разных приоритетов, мы можем увидеть, что у потоков одного процесса есть общие характеристики, такие как: идентификатор процесса и родительского процесса (PID и PPID), идентификатор группы потоков (TGID) и количество потоков, которые были запущены в рамках этого процесса (THCNT).