**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МОЭВМ**

отчет

**по лабораторной работе №4**

**по дисциплине «Операционные системы»**

Тема: Взаимодействие родственных процессов. Управление процессами посредством сигналов. Многонитевое функционирование.

|  |  |  |
| --- | --- | --- |
| Студент гр. 1304 |  | Кривоченко Д.И. |
| Преподаватель |  | Душутина Е.В. |

Санкт-Петербург

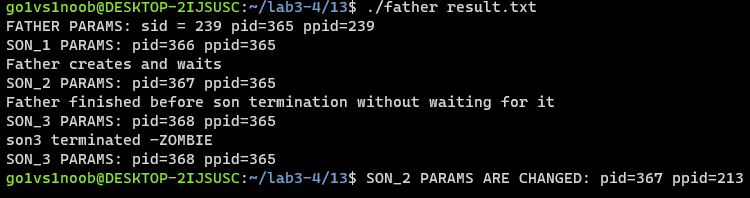
2023

**Цель работы.**

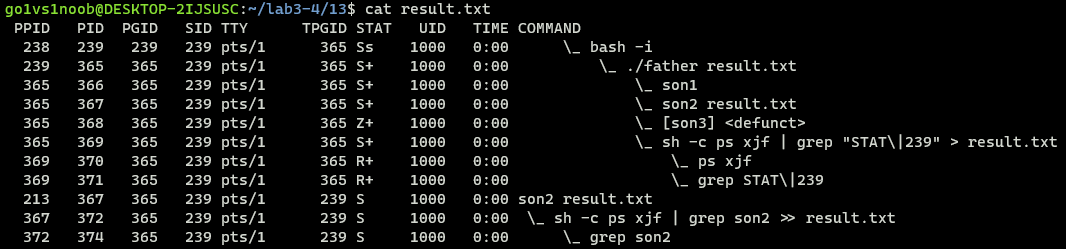
**Ход работы.**

1. Изменяя длительность выполнения процессов и параметры системных

вызовов рассмотрим 3 ситуации и получим соответствующие таблицы процессов: процесс-отец запускает процесс-сын и ожидает его завершения; процесс-отец запускает процесс сын и завершает своё выполнение; процесс-отец запускает процесс-сын и не ожидает его завершения. Организуем программы с перенаправлением результатов в файл. *Son1* будет отвечать за первую ситуацию, *son2* – за вторую, *son3* – за третью, все они будут вызываться из *father.c*. Результат работы – на рисунках 1 и 2.

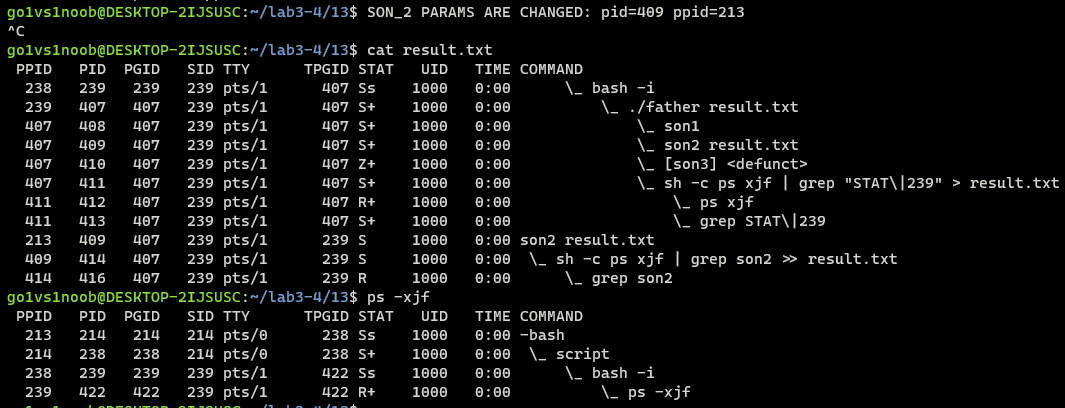


*Рисунок 1 – результат работы программы*



*Рисунок 2 – содержание таблицы процессов во время выполнения программы*

В коде son2 выполняется задержка в 20 секунд, что обеспечивает то, что он будет работать дольше процесса-родителя. В результате видно, что во *PPID son2* во время работы программы и через 20 секунд изменились (стал таким же, как у *bash*, т. е. процесс стал самостоятельным), это видно на рисунке 3.

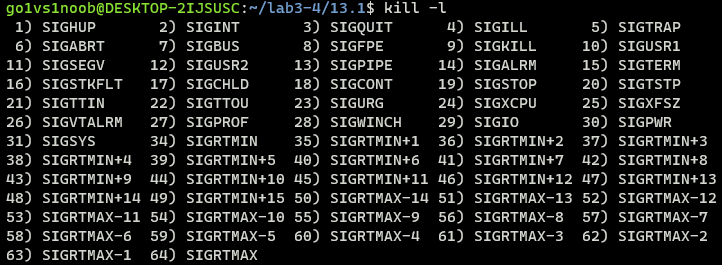


*Рисунок 3 – изменение PPID процесса*

Рассмотрим, что произошло с процессом *son3*. Поскольку процесс-отец не дождался завершения дочернего процесса, он находится в состоянии *Zombie* (что видно на рисунке 3, STAT = Z+). Это означает, что процесс остается формально существующим, но ресурсы, отведенные для него освобождены. Такие процессы остаются в таблице на случай, если кто-то запросит статистику использования ресурсов этим потомком или статус о его завершении.

1. С помощью команды *kill -l* ознакомимся с перечнем сигналов,

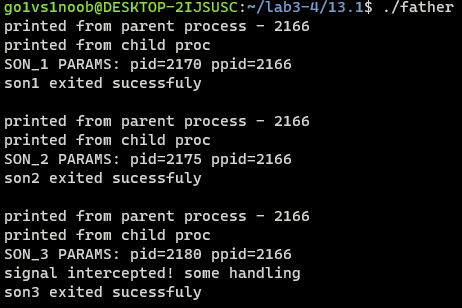
поддерживаемых процессами. Для начала ознакомимся с системными вызовами *kill()*, *signal()*. Системный вызов *kill()* позволяет отправить сигнал процессу (или группе процессов). При успешном выполнении (т. е. хотя-бы один сигнал отправлен) возвращает 0, при ошибке – -1. Все сигналы можно вывести командой *kill -l* (рисунок 4). Их можно задавать как числом, так и в формате текста.



*Рисунок 4 – список сигналов, поддерживаемых kill*

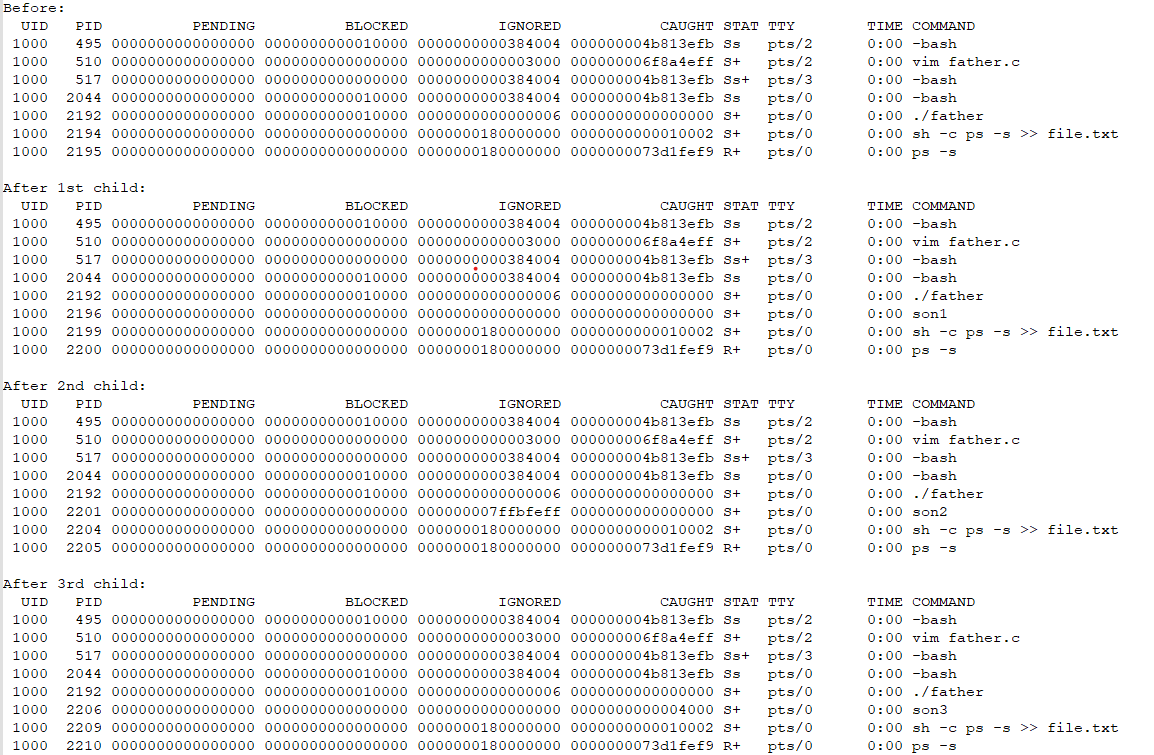
Системный вызов *signal(signum, handler)* позволяет установить свой обработчик сигнала: устанавливает signum в обработчик handler, который может быть нами написанной функцией (обработчиком сигнала).

Напишем программу *father.c*, порождающий *son1, son2, son3*. Сделаем так, что *son1* реагирует на сигнал по умолчанию, *son2* – игнорирует сигнал, *son3* – перехватывает и обрабатывает сигнал. Результат работы программы на рисунке 5.



*Рисунок 5 – результат работы программы, написанной для демонстрации обработки сигналов*

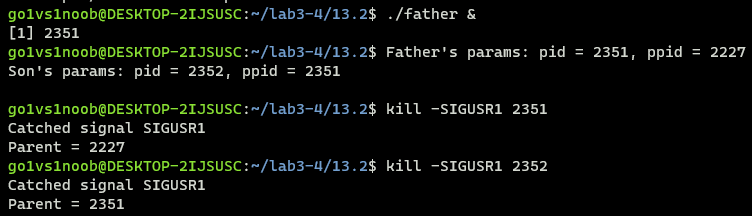
Каждой программе подаётся сигнал *SIGTERM*. Первая программа обрабатывает его стандартно и завершает свою работу. Вторая программа игнорирует его (в результате чего *sleep(10)* выполняется все десять секунд), третья – перехватывает, выводя в консоль строку, говорящую о том, что сигнал обработан. Рассмотрим таблицу процессов до и после посылки сигналов (рисунок 6).



*Рисунок 6 – ps -s после отправки сигналов*

Из вывода *ps -s* видно, что после вызова 1-го дочернего процесса, никакие сигналы не были пойманы/игнорированы/заблокированы, что и ожидалось увидеть, поскольку 1-я программа обрабатывает сигнал. После вызова второй программы, видно, что напротив строки с *son2* везде нули, кроме столбца *IGNORED*, таким образом убеждаемся, что наш сигнал был проигнорирован *son2*. После запуска 3-го дочернего процесса, видим, что столбец с *CAUGHT*, как и ожидалось, ненулевой (поскольку третья подпрограмма перехватывает процесс и обрабатывает его).

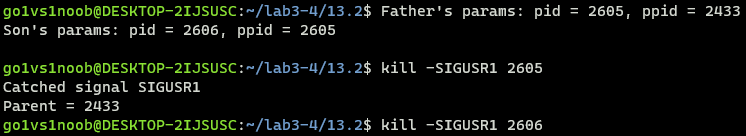
Проанализируем наследование диспозиции сигналов при создании процессов на этапе *fork()* и *exec()*. Напишем программу *sig\_father.c*, которая меняет диспозицию сигналов. Она задаёт обработчик многопользовательских сигналов *SIGUSR1* и *SIGUSR2*. Родительский процесс порождает процесс-копию с помощью *fork()* и уходит в ожидание сигналов. Процесс-потомок, при этом, не ждёт никаких сигналов, не назначает им обработчиков. Обработчик *SIGUSR1* и *SIGUSR2* содержит восстановление диспозиции и оповещение на экране об удачно или неудачно полученном сигнале и идентификаторе родительского процесса. Сигналы генерируются из командной строки. Рассмотрим работу программы (Рисунок 7).



*Рисунок 7 – результат работы программы, демонстрирующей наследование диспозиции*

Как видно, при отправке сигнала разным процессам результат совпадает, что говорит о том, что потомок использует один и тот же обработчик, что и родитель. Это свидетельствует о наследовании диспозиции при порождении потомка на этапе *fork()*. Диспозиция сигналов для дочернего процесса, созданного с помощью *fork()* сохраняется даже после завершения процесса-родителя.

Повторим эксперимент для процесса-родителя, порождающего дочерний процесс с помощью *fork()* и *exec()*. Теперь потомок загружается с помощью *execl()*.

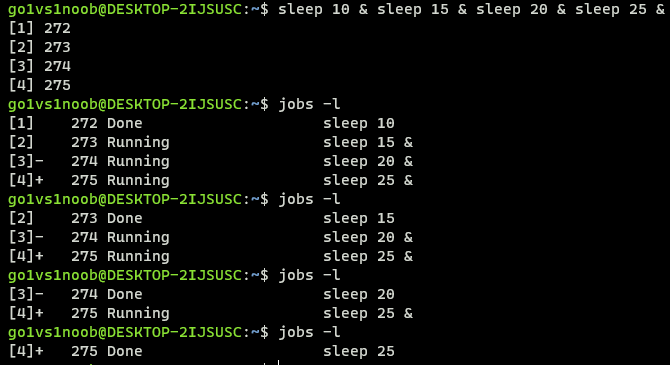


*Рисунок 8 – результат работы программы, демонстрирующей отсутствие наследования диспозиции*

Из рисунка 8, при отправке сигнала процессу-отцу срабатывает его обработчик, а при отправке процессу-потомку диспозиция этого сигнала не сохраняется и срабатывает обработчик по умолчанию (ничего не происходит). Из этого можно сделать вывод, что при создании процесса с помощью *fork()* и *exec()* диспозиция сигналов не наследуется.

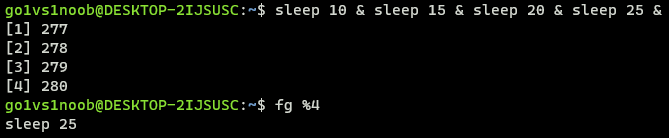
1. Запустим в фоновом режиме несколько утилит в фоновом режиме.

Воспользуемся командой *jobs* для анализа списка заданий и очередности их выполнений. Вернём невыполненные задание в приоритетный режим командой *fg*. Отменим одно из невыполненных заданий.



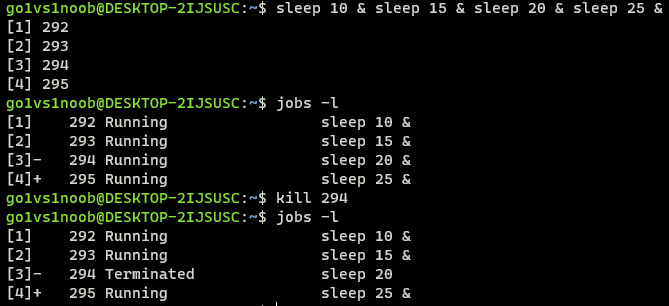
*Рисунок 9* *– результат команды jobs*

Из рисунка 9, выполнение задач начинается с начала. С помощью утилиты *fg* повысим приоритет задачи 4. В результате сразу начинается выполняться задача 4, не в фоновом режиме (рисунок 10).



*Рисунок 10 – демонстрация команды fg*

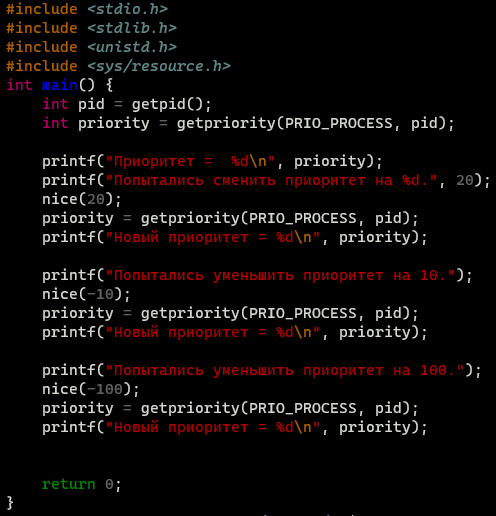
С помощью команды *kill* отменим одно из невыполненных заданий (рисунок 11):



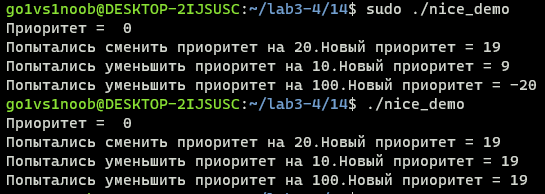
*Рисунок 11 – демонстрация команды kill*

1. Рассмотрим команду *nice()* и системный вызов *getpriority()*. Функция

*nice()* позволяет менять приоритет процессу. *Getpriority()* позволяет получить приоритет нужного процесса. Рассмотрим их применение. Устройство программы на рисунке 12, результат работы – на рисунке 13.



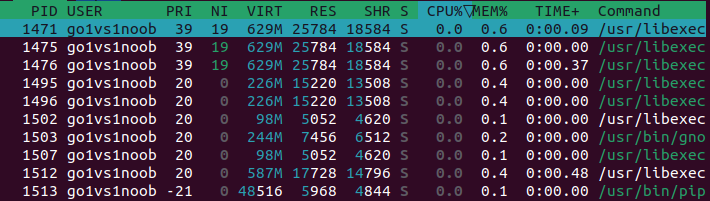
*Рисунок 12 – содержимое программы для демонстрации nice() и getpriority()*



*Рисунок 13 – результат работы программы для демонстрации nice() и getpriority()*

Как видно из рисунка выше, уменьшать приоритет можно только имея права суперпользователя. Для увеличения приоритета такие права не нужны. Также видно, что изменять приоритет можно в рамках от -20 до 19.

Разница в приоритетах для системных и пользовательских процессов есть: обычно большинство системных процессов, отвечающих за управление системы имеют более высокий приоритет, чем пользовательские (рисунок 14). Это обусловлено тем, что системные процессы обеспечивают более низкоуровневые функции.



*Рисунок 14 – демонстрация разницы в приоритетах*

Приоритеты реального времени используются для процессов, которые должны быть выполнены в строго определенные моменты времени, чтобы обеспечить отзывчивость системы на события внешней среды.



*Рисунок 15 – пример процесса с приоритетом реалного времени*

При запуске из оболочки пользовательский приоритет по умолчанию равен 20. Он может быть изменен используя, например, утилиту nice.



*Рисунок 16 – пример процесса при запуске из оболочки*

1. Ознакомимся с командой *nohup*. Запустим длительный процесс по

*nohup()*. Завершим сеанс работы, войдем в систему и снова проверим таблицу процессов.

*Nohup()* – утилита, позволяющая запустить команду, невосприимчивую к сигналам потери связи, и чей вывод будет направлен не в терминал, а в файл *nohup.out*. Таким образом, команда будет выполняться в фоновом режиме даже когда пользователь выйдет из системы. Запустим длительный процесс с помощью *nohup()*:

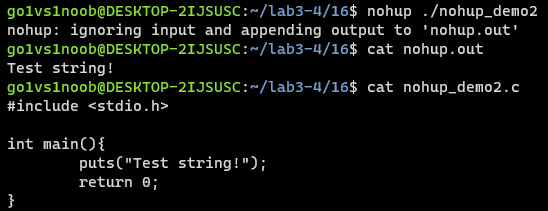


*Рисунок 17 – запуск скрипта с nohup*



*Рисунок 18 – список процессов после nohup*

Таким образом, при выходе из системы процесс *nohup\_demo* не завершился, так как команда *nohup* позволила этому процессу игнорировать сигнал *SIGHUP*, который высылается при выходе из системы. Рассмотрим ещё один пример с командой *nohup*, выводящей на экран строку.

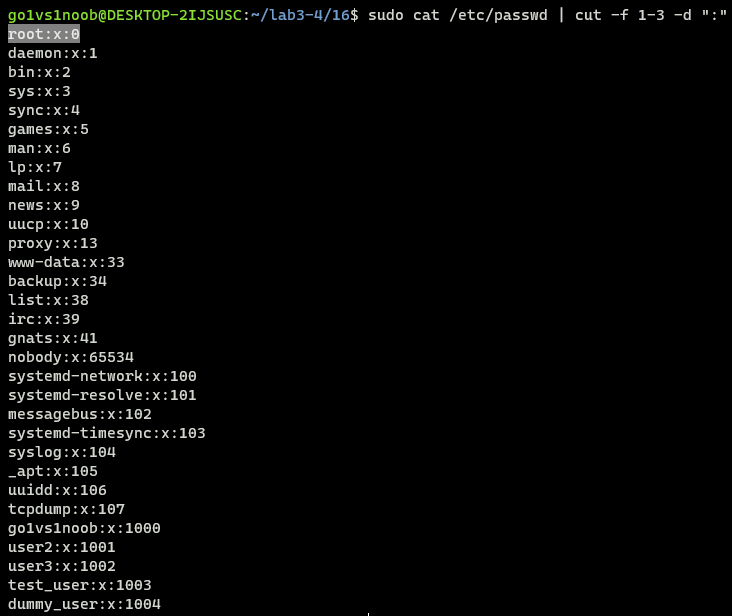


*Рисунок 19 – результат работы nohup с записью строки*

Как видно, в результате работы команды *nohup* запись производилась не в консоль, а в файл *nohup.out*.

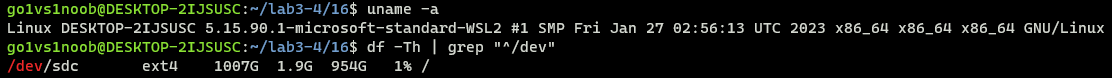
1. Определим минимальное значение *uid*, кому оно принадлежит.

Определим минимальное и максимальное значение *pid*, каким процессам они принадлежат. Проанализируем множество системных процессов. Перечислим назначение самых важных из них. Минимальное значение *uid* равно нулю и принадлежит *root*. Это видно на рисунке 20 (в файле */etc/passwd* на 1-м месте пользователь, а на третьем – его *uid*).



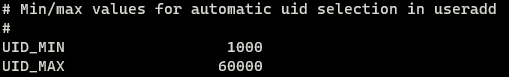
*Рисунок 20 – демонстрация uid root*

Максимальное значение *uid* зависит от используемой файловой системы и версии ядра. Рассмотрим файловую систему и версию ядра на моей машине (рисунок 21).



*Рисунок 21 – демонстрация версии ядра и файловой системы на машине*

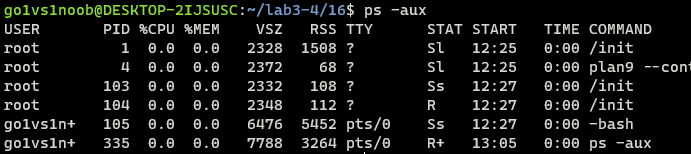
Для файловой системы *ext4* максимальное значение *uid* составляет 4294967295 (2^32 – 1). Однако в самой ОС это значение может быть другим (рисунок 22).



*Рисунок 22 – минимальные и максимальные значения uid*

Это минимальное и максимальное значения, которые могут быть выданы пользователю. Остальные зарезервированы под нужды ОС.

Минимальное значение *PID* равно одному и принадлежит процессу *init*. Этот процесс запускается после загрузки системы и является родительским процессом для всех остальных. *PID* *0* зарезервирован ядром. Таким образом, любой новый процесс будет иметь *pid* больше либо равный двум.



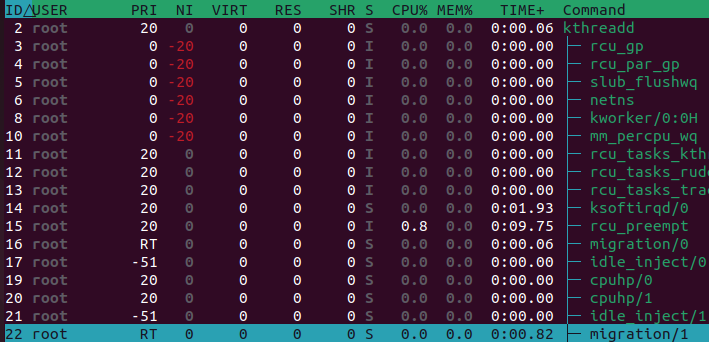
*Рисунок 23 – pid процессов*

Максимальное значение *PID* может варьироваться, при том его можно изменять. Оно находится в файле */proc/sys/kernel/pid\_max*



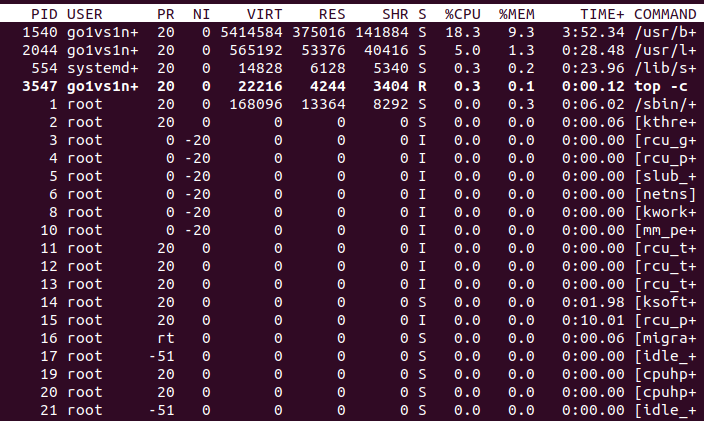
*Рисунок 24 – максимальное значение pid*

Посмотреть на множество системных процессов можно с помощью утилиты *htop*. Все эти процессы будут наследниками *kthreadd*.



*Рисунок 25 – множество системных процессов*

Также можно использовать утилиту *top -c*. Различаются системные процессы от обычных тем, что их имя показано в квадратных скобках.



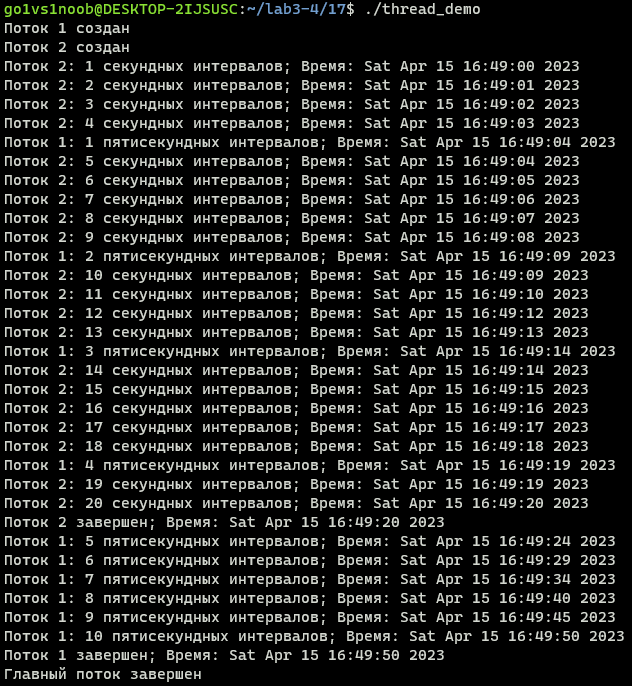
*Рисунок 26 – множество системных процессов*

1. Напишем программу, формирующую несколько нитей. Каждая нить

Выводит на печать собственное имя и инкрементирует переменную *counter*, соответствующую своему счётчику, выводит на экран сколько в текущий момент времени прошло односекундных или пятисекундных интервалов и текущее время.



*Рисунок 27 – устройство программы, создающей несколько нитей*



*Рисунок 28 – результат работы программы, создающей несколько нитей*

1. Проанализируем выполнение нитей, распределение во времени. Для

этого используем вывод таблицы *ps -axhf* в разных местах программы (когда, например, будет работать первый и второй поток и когда будет работать только второй поток).

*Листинг 1 – результат модифицированной программы с ps -axhf*

*Поток 1 создан*

*Поток 2 создан*

*Работает первый и второй поток*

*1 ? Sl 0:00 /init*

*4 ? Sl 0:01 plan9 --control-socket 5 --log-level 4 --server-fd 6 --pipe-fd 8 --log-truncate*

*103 ? Ss 0:00 /init*

*104 ? S 0:00 \\_ /init*

*105 pts/0 Ss+ 0:00 \\_ -bash*

*337 ? Ss 0:00 /init*

*338 ? S 0:01 \\_ /init*

*339 pts/1 Ss 0:01 \\_ -bash*

*2514 pts/1 Sl+ 0:00 \\_ ./thread\_demo*

*2518 pts/1 S+ 0:00 \\_ sh -c ps -axhf*

*2519 pts/1 R+ 0:00 \\_ ps -axhf*

*1900 ? Ss 0:00 /init*

*1901 ? S 0:00 \\_ /init*

*1902 pts/2 Ss+ 0:00 \\_ -bash*

*Поток 2: 1 секундных интервалов; Время: Sat Apr 15 17:56:53 2023*

*Поток 2: 2 секундных интервалов; Время: Sat Apr 15 17:56:54 2023*

*Поток 2: 3 секундных интервалов; Время: Sat Apr 15 17:56:55 2023*

*Продолжение листинга 1*

*Поток 2: 4 секундных интервалов; Время: Sat Apr 15 17:56:56 2023*

*Поток 1: 1 пятисекундных интервалов; Время: Sat Apr 15 17:56:57 2023*

*Поток 2: 5 секундных интервалов; Время: Sat Apr 15 17:56:57 2023*

*Поток 2: 6 секундных интервалов; Время: Sat Apr 15 17:56:58 2023*

*Поток 2: 7 секундных интервалов; Время: Sat Apr 15 17:56:59 2023*

*Поток 2: 8 секундных интервалов; Время: Sat Apr 15 17:57:00 2023*

*Поток 2: 9 секундных интервалов; Время: Sat Apr 15 17:57:02 2023*

*Поток 1: 2 пятисекундных интервалов; Время: Sat Apr 15 17:57:02 2023*

*Поток 2: 10 секундных интервалов; Время: Sat Apr 15 17:57:03 2023*

*Поток 2: 11 секундных интервалов; Время: Sat Apr 15 17:57:04 2023*

*Поток 2: 12 секундных интервалов; Время: Sat Apr 15 17:57:05 2023*

*Поток 2: 13 секундных интервалов; Время: Sat Apr 15 17:57:06 2023*

*Поток 1: 3 пятисекундных интервалов; Время: Sat Apr 15 17:57:07 2023*

*Поток 2: 14 секундных интервалов; Время: Sat Apr 15 17:57:07 2023*

*Поток 2: 15 секундных интервалов; Время: Sat Apr 15 17:57:08 2023*

*Поток 2: 16 секундных интервалов; Время: Sat Apr 15 17:57:09 2023*

*Поток 2: 17 секундных интервалов; Время: Sat Apr 15 17:57:10 2023*

*Поток 2: 18 секундных интервалов; Время: Sat Apr 15 17:57:11 2023*

*Поток 1: 4 пятисекундных интервалов; Время: Sat Apr 15 17:57:12 2023*

*Поток 2: 19 секундных интервалов; Время: Sat Apr 15 17:57:12 2023*

*Поток 2: 20 секундных интервалов; Время: Sat Apr 15 17:57:13 2023*

*Поток 2 завершен; Время: Sat Apr 15 17:57:13 2023*

*Поток 1: 5 пятисекундных интервалов; Время: Sat Apr 15 17:57:17 2023*

*Поток 1: 6 пятисекундных интервалов; Время: Sat Apr 15 17:57:22 2023*

*Поток 1: 7 пятисекундных интервалов; Время: Sat Apr 15 17:57:27 2023*

*Поток 1: 8 пятисекундных интервалов; Время: Sat Apr 15 17:57:32 2023*

*Поток 1: 9 пятисекундных интервалов; Время: Sat Apr 15 17:57:37 2023*

*Поток 1: 10 пятисекундных интервалов; Время: Sat Apr 15 17:57:42 2023*

*Только первый поток работает*

*1 ? Sl 0:00 /init*

*4 ? Sl 0:01 plan9 --control-socket 5 --log-level 4 --server-fd 6 --pipe-fd 8 --log-truncate*

*103 ? Ss 0:00 /init*

*104 ? S 0:00 \\_ /init*

*105 pts/0 Ss+ 0:00 \\_ -bash*

*337 ? Ss 0:00 /init*

*338 ? S 0:01 \\_ /init*

*339 pts/1 Ss 0:01 \\_ -bash*

*2514 pts/1 Sl+ 0:00 \\_ ./thread\_demo*

*2521 pts/1 S+ 0:00 \\_ sh -c ps -axhf*

*2522 pts/1 R+ 0:00 \\_ ps -axhf*

*1900 ? Ss 0:00 /init*

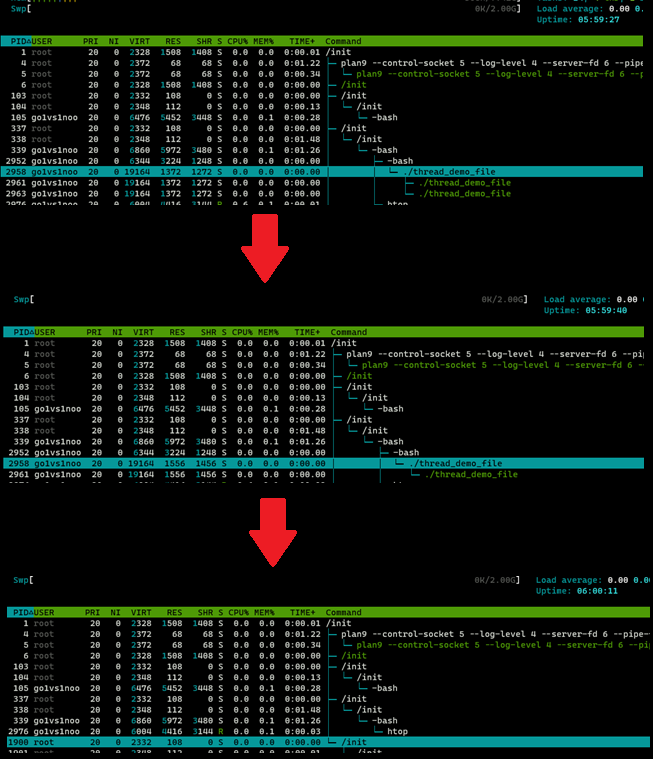
*1901 ? S 0:00 \\_ /init*

*1902 pts/2 Ss+ 0:00 \\_ -bash*

*Поток 1 завершен; Время: Sat Apr 15 17:57:42 2023*

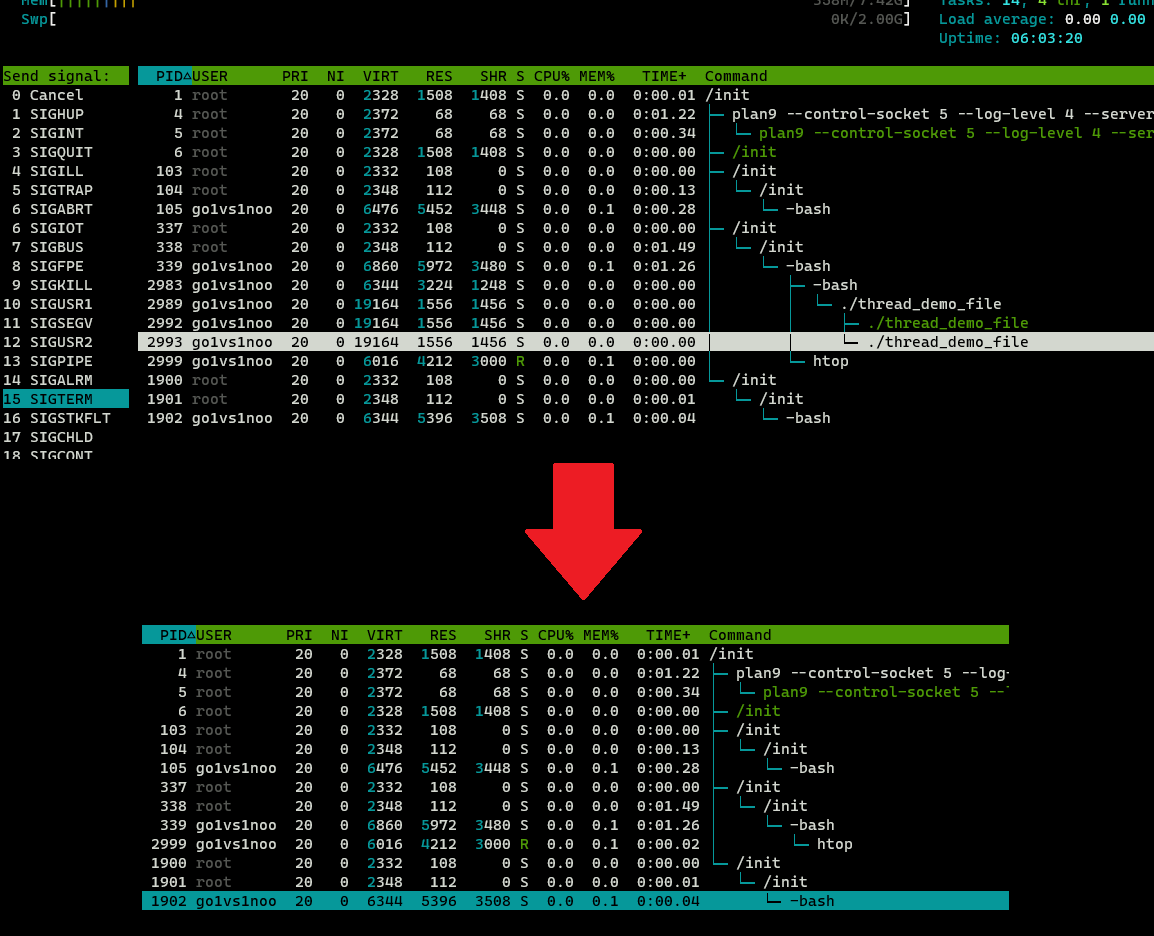
*Главный поток завершен*

Для наглядности изменим программу так: все выводы перенаправим в файл (чтобы можно было без проблем запустить её в фон), а анализировать работу будем через утилиту *htop* (рисунок 29).



*Рисунок 29 – демонстрация работы программы через htop (список процессов)*

Как видно, сначала работало два потока, далее один закончил свою работу и, наконец, второй закончил работу, и программа завершилась. Попробуем завершить один из потоков, послав ему *SIGTERM*.



*Рисунок 30 – результат посыла одному из процессов SIGTERM*

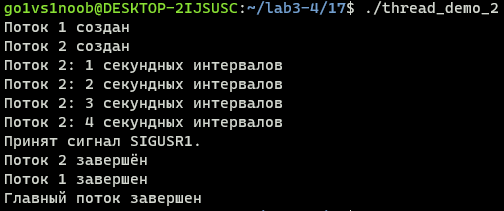
Как видно, после послания одному потоку *SIGTERM* завершилась вся программа, что подтверждает информацию из теоретических сведений.

1. Модифицируем программу так, чтобы управление второй нитью

передавалось посредством сигнала *SIGUSR1* из первой нити. Для этого напишем программу так: в *main()* передадим управление сигналом *SIGUSR1* в функцию *sigusr1\_handler* (внутри неё выведется сообщение об успешном принятии сигнала и завершится поток 2). Также внутри головной функции вызовем поток 1. В потоке 1 создаём поток 2, ждём 5 секунд и посылаем сигнал *SIGUSR1*, в результате чего поток завершает свою работу. Устройство программы на рисунке 31, результат работы – на рисунке 32.



*Рисунок 31 – устройство программы, создающей поток из потока*

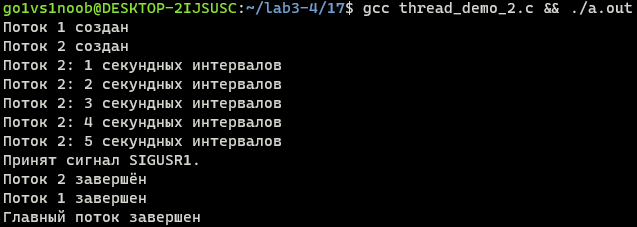


*Рисунок 32 – результат программы, создающей поток из потока*

Как и ожидалось, поток 2 завершён из потока 1 в течение 5 секундных интервалов.

1. Модифицируем программу так, чтобы внутри обработчика сигнала

поток завершался посредством функции *pthread\_exit(NULL)*. Сравним результат с предыдущим (рисунок 33).



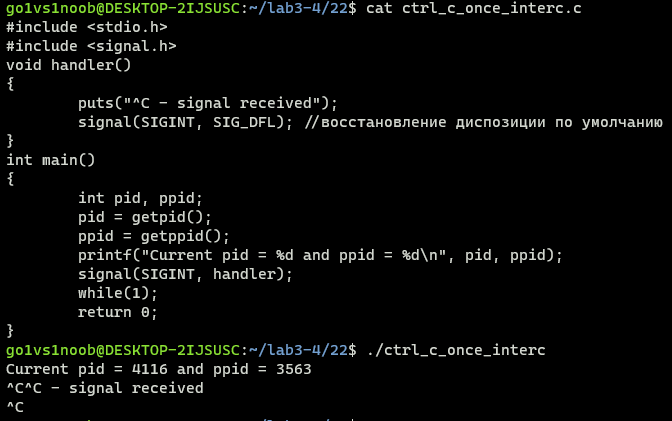
*Рисунок 33 – модификация обработчика сигнала*

Как видно из рисунка, успело пройти 5 секундных интервалов, в отличие от четырех в предыдущей версии программы. Это связано с небольшими различиями *pthread\_exit()* и *pthread\_cancel*. Различие в точке выхода: для *pthread\_cancel* это может быть любой точкой в потоке. *Pthread\_exit()*, в свою очередь, позволяет потоку выполнить все предварительные или завершающие действия для корректной остановки, в то время как *pthread\_cancel()* может прервать поток в любой точке его выполнения. Можно сказать, что аналог *pthread\_exit()* для простых функций – *exit()*, а аналог *pthread\_cancel()* – *abort()*.

Именно поэтому вывод может различаться (т. е. *pthread\_exit()* немного дольше работает).

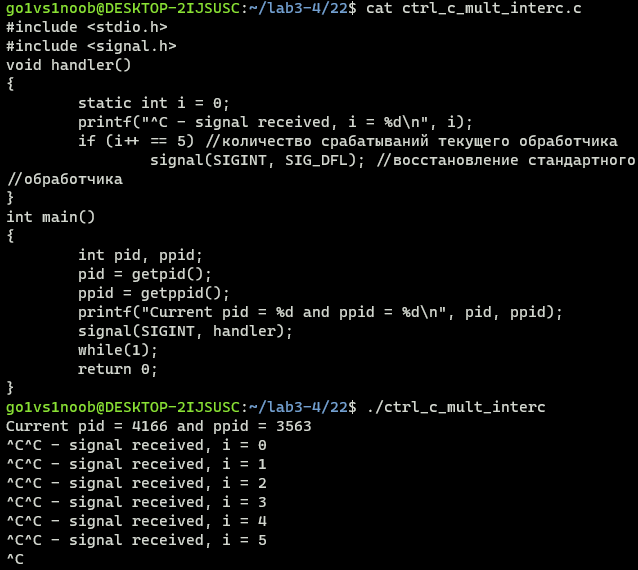
1. Перехватим сигнал *Ctrl C* для процесса и потока однократно, а также

многократно с восстановлением исходного обработчика после нескольких раз срабатывания. Содержание программы и результат работы на рисунке 34.



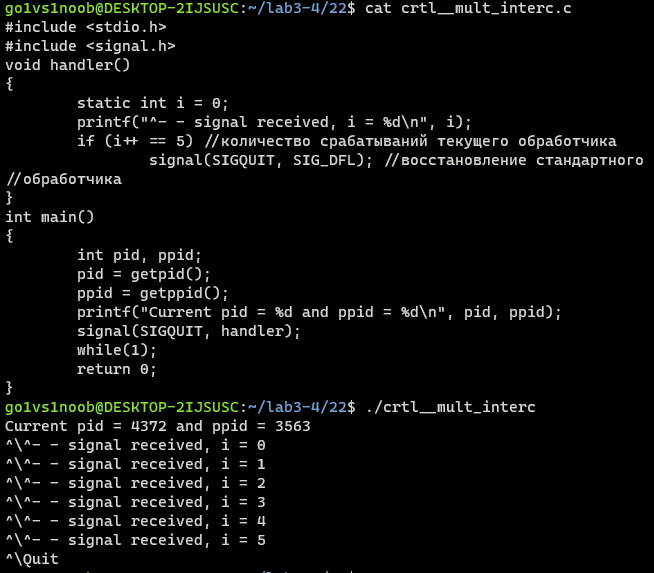
*Рисунок 34 – единичный перехват Ctrl C*

Сигнал *^C* перехватился и однократно вызвался обработчик *handler*, который вывел строку, оповещающую о получении сигнала, после чего обработчик *SIGINT* возвращается по умолчанию, результат которого – принудительное завершение программы. Скорректируем программу для многократного нажатия. Содержание программы и демонстрация работы на рисунке 35.



*Рисунок 35 – многократный перехват Ctrl C*

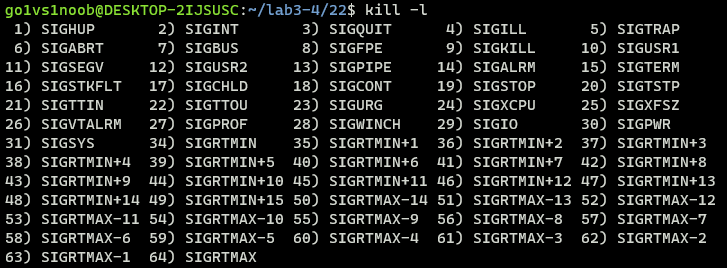
Проделаем аналогичную работу для другой комбинации клавиш (*Ctrl + \*). Результат на рисунке 36.



*Рисунок 36 – многократный перехват Ctrl \*

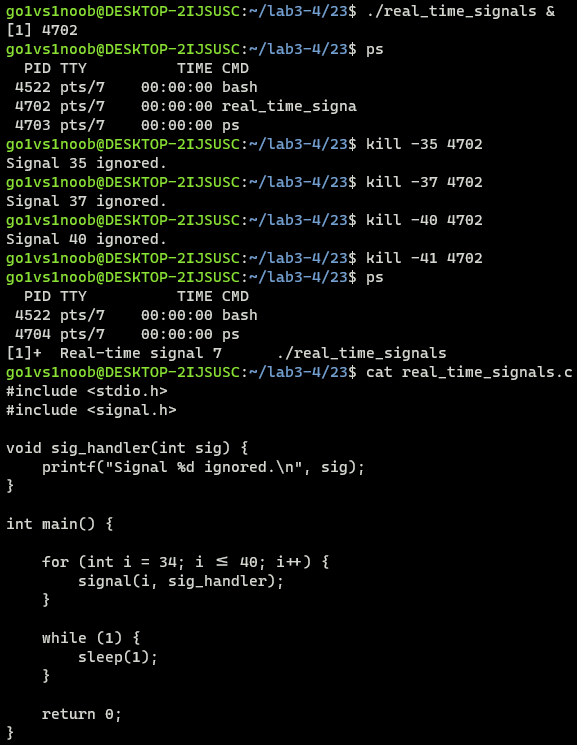
1. С помощью утилиты *kill* выведем список сигналов и дадим краткую

характеристику.



*Рисунок 37 – список сигналов kill*

Команда выводит на экран список всех сигналов, которые могут быть отправлены процессу с помощью команды *kill* или других инструментов управления процессами. Процессы от 1 до 31 заданы стандартом *POSIX* и выполняют определенные функции. Например, завершение процесса (*SIGTERM*), неожиданное прерывание процесса (*SIGINT*), остановка процесса (*SIGSTOP*). Сигналы 32 и 33 не отображаются, так как они заняты реализацией *POSIX*-ядер. Сигналы с номерами от 34 и выше – переменные. *SIGRTMIN* и *SIGRTMAX* используются как минимальное и максимальное значение для этих сигналов (эти значения могут быть изменены), которые могут быть определены динамически в программах. Из мануала, эти сигналы называются сигналами реального времени. Их предназначение не обязательно определено, как говорилось раннее, они могут быть использованы при разработке. По умолчанию сигналы от 34 и выше завершают работу программы. Приведём пример программы, использующей сигналы от 34 до 40. Результат работы программы и устройство на рисунке 38.



*Рисунок 38 – пример программы, используемой сигналы реального времени*

Как видно, при посылке сигналов от 34 до 40 сигналы игнорируются нашим обработчиком, а при посылке сигнала 41 программа завершается (т.к. по умолчанию, как и ожидалось, сигнал 41 завершает работу).

1. Исследуем характеристики планирования процессов. Создадим

программу, которая вызывает процесс son1 с разными политиками планирования. Проведём несколько экспериментов по-разному нагружая вычисления в RR и FIFO процессах. Содержание программ в листинге 2.

*Листинг 2 – содержание программы, порождающей RR и FIFO процессы*

*│ File: prog\_sched.c*

*1 │ #include <stdio.h>*

*2 │ #include <stdlib.h>*

*3 │ #include <sched.h>*

*4 │ #include <unistd.h>*

*5 │ #define PROC\_AMOUNT 10*

*6 │ int main() {*

*7 │ int i;*

*8 │ char\* program = "son1";*

*9 │ struct sched\_param params;*

*10 │ params.sched\_priority = PROC\_AMOUNT;*

*11 │ int sched;*

*12 │ for (i = 0; i < PROC\_AMOUNT; i++) {*

*13 │ pid\_t pid = fork();*

*14 │ if (pid < 0) {*

*15 │ printf("Ошибка при создании дочернего процесса\n");*

*16 │ return 1;*

*17 │ } else if (pid == 0) {*

*18 │ sched = (i % 2 == 0) ? SCHED\_FIFO : SCHED\_RR;*

*19 │ if (sched\_setscheduler(0, sched, &params) < 0) {*

*20 │ printf("Ошибка при установке политики планирования\n");*

*21 │ return 1;*

*22 │ }*

*23 │ execl(program, program, NULL);*

*24 │ printf("Ошибка при вызове функции execl\n");*

*25 │ return 1;*

*26 │ }*

*27 │ //else {*

*28 │ // wait(NULL);*

*29 │ //}*

*30 │ }*

*31 │ for (int i = 0; i<PROC\_AMOUNT;i++){*

*32 │ wait(NULL);*

*33 │ }*

*34 │*

*35 │ return 0;*

*36 │ }*

*│ File: son1.c*

*1 │ #include <stdio.h>*

*2 │ #include <stdlib.h>*

*3 │ #include <sched.h>*

*4 │ #include <time.h>*

*5 │ #include <unistd.h>*

*6 │ int main()*

*7 │ {*

*8 │ struct sched\_param shdprm; // Значения параметров планирования*

*9 │ int i, pid, ppid;*

*10 │ pid = getpid();*

*11 │ int k=0;*

*12 │ ppid = getppid();*

*13 │ // printf("pid=%i ppid=%i ", pid, ppid);*

*14 │ //printf("pol: ");*

*15 │ int schedule = sched\_getscheduler(0);*

*16 │ switch (schedule)*

*17 │ {*

*18 │ case SCHED\_FIFO:*

*19 │ printf("SCHED\_FIFO\n");*

*20 │ // sleep(1);*

*21 │ break;*

*22 │ case SCHED\_RR:*

*23 │ //printf("SCHED\_RR\n");*

*24 │ //sleep(5);*

*25 │ for (int i =0;i<100000000;i++){*

*26 │ k+=1;*

*27 │ }*

*28 │ printf("SCHED\_RR\n");*

*29 │ break;*

*30 │ case SCHED\_OTHER:*

*31 │ printf("SCHED\_OTHER\n");*

*32 │ break;*

*33 │ case -1:*

*34 │ perror("SCHED\_GETSCHEDULER");*

*35 │ break;*

*36 │ default:*

*37 │ printf("Неизвестная политика планирования\n");*

*38 │ }*

*39 │ //if (sched\_getparam(0, &shdprm) == 0)*

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

*41 │ // else*

*42 │ // perror ("SCHED\_GETPARAM");*

*43 │ return 0;*

*44 │ }*



*Рисунок 39 – результат работы программы, когда в вызываемом процессе мало вычислений и для FIFO, и для RR*

**

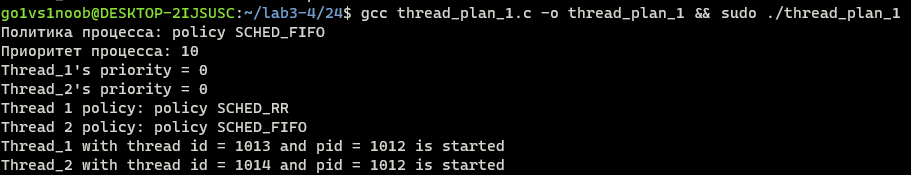
*Рисунок 40 – результат работы программы, когда в вызываемом процессе мало вычислений и для FIFO, много для RR*

**

*Рисунок 41 – результат работы программы, когда в вызываемом процессе много вычислений и для FIFO, много для RR*

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

Исследуем характеристики планирования потоков. Изменять стратегию планирования потока можно с помощью команды *pthread\_attr\_setschedpolicy*. Перед этим нужно инициализировать описатель атрибутов потока управления командой *pthread\_attr\_init*. Иначе получим *SEGFAULT*. Поменяем стратегию планирования потоков. Первому потоку зададим стратегию планирования *SCHED\_RR*, а второму – *SCHED\_FIFO*. Самому же процессу зададим стратегию планирования *FIFO*. Результат работы программы и её устройство на рисунке 42.



*Рисунок 42 – установка процедур планирования на потоки*

Как видно, процессу 1 задана процедура планирования *RR*, а процессу 2 – *FIFO*. Приоритеты процессы не были явно заданы, поэтому они равны нулю. Разница между *pthread\_attr\_setschedpolicy()* и *sched\_setscheduler()* в том, что первая команда используется для установки стратегии планирования новому потоку, а вторая – для установки планирования уже существующему потоку или процессу. Рассмотрим разницу на примере. В данной программе самому процессу устанавливается процедура планирования *FIFO*, а двум созданным потокам *RR* и *FIFO* соответственно. Устройство программы в листинге 3. Результат работы на рисунке 43.

*Листинг 3 – устройство программы, демонстрирующей комбинацию двух команд*

*│ File: thread\_plan\_1.c*

*1 │ #include <signal.h>*

*2 │ #include <pthread.h>*

*3 │ #include <stdio.h>*

*4 │ #include <sys/types.h>*

*5 │ #include <linux/unistd.h>*

*6 │ #include <sys/syscall.h>*

*7 │ #include <sched.h>*

*8 │ #include <unistd.h>*

*9 │*

*10 │ pthread\_t t1, t2;*

*11 │ void \*thread1();*

*12 │ void \*thread2();*

*13 │ void switch\_policy(int policy);*

*14 │*

*15 │ void main()*

*16 │ {*

*17 │ int policy1;*

*18 │ int policy2;*

*19 │ struct sched\_param param;*

*20 │ struct sched\_param shdprm;*

*21 │ pid\_t pid = getpid();*

*22 │ pthread\_attr\_t attr\_1, attr\_2;*

*23 │ param.sched\_priority = 5;*

*24 │ shdprm.sched\_priority = 10;*

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

*26 │ {*

*27 │ perror("SCHED\_SETSCHEDULER");*

*28 │ }*

*29 │*

*30 │ printf("Политика процесса: ");*

*31 │ switch\_policy(sched\_getscheduler(pid));*

*32 │ printf("Приоритет процесса: %d\n", shdprm.sched\_priority);*

*33 │*

*34 │*

*35 │ pthread\_attr\_init(&attr\_1);*

*36 │ pthread\_attr\_init(&attr\_2);*

*37 │ pthread\_attr\_setschedparam(&attr\_1, &shdprm);*

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

*39 │ pthread\_attr\_setschedpolicy(&attr\_1, SCHED\_RR);*

*40 │ pthread\_attr\_setschedpolicy(&attr\_2, SCHED\_FIFO);*

*41 │ pthread\_attr\_getschedparam(&attr\_1, &shdprm);*

*42 │ pthread\_attr\_getschedpolicy(&attr\_1, &policy1);*

*43 │*

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

*45 │ pthread\_attr\_getschedpolicy(&attr\_2, &policy2);*

*46 │*

*47 │ printf("Thread 1 policy: ");*

*48 │ switch\_policy(policy1);*

*49 │ printf("Thread 2 policy: ");*

*50 │ switch\_policy(policy2);*

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

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

*53 │ pthread\_join(t1, NULL);*

*54 │ pthread\_join(t2, NULL);*

*55 │ pthread\_attr\_destroy(&attr\_1);*

*56 │ pthread\_attr\_destroy(&attr\_2);*

*57 │ }*

*58 │*

*59 │ void switch\_policy(int policy)*

*60 │ {*

*61 │ switch (policy)*

*62 │ {*

*63 │ case SCHED\_FIFO:*

*64 │ printf("policy SCHED\_FIFO\n");*

*65 │ break;*

*66 │ case SCHED\_RR:*

*67 │ printf("policy SCHED\_RR\n");*

*68 │ break;*

*69 │ case SCHED\_OTHER:*

*70 │ printf("policy SCHED\_OTHER\n");*

*71 │ break;*

*72 │ case -1:*

*73 │ perror("policy SCHED\_GETSCHEDULER");*

*74 │ break;*

*75 │ default:*

*76 │ printf ("policy Неизвестная политика планирования\n");*

*77 │ }*

*78 │ }*

*79 │ void \*thread2()*

*80 │ {*

*81 │ int i, count = 0;*

*82 │ int tid, pid;*

*83 │ tid = syscall(SYS\_gettid);*

*84 │ pid = getpid();*

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

*86 │ for (i = 0; i < 10; i++)*

*87 │ {*

*88 │ count += 1;*

*89 │ }*

*90 │ }*

*91 │ void \*thread1()*

*92 │ {*

*93 │ int i, count = 0;*

*94 │ int tid, pid;*

*95 │ tid = syscall(SYS\_gettid);*

*96 │ pid = getpid();*

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

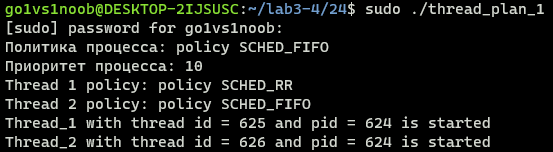
*98 │ for (i = 0; i < 10; i++)*

*99 │ {*

*100 │ count += 1;*

*101 │ }*

*102 │ }*



*Рисунок 43 – результат работы программы в листинге 3*

Модифицируем программы так, чтобы создавалось несколько потоков, и этим потокам выдавались разные политики планирования. Внутри потоков вставим некоторые вычисления. Рассмотрим содержание программы:

*Листинг 4 –программа для демонстрации политик планирования потоков*

*│ File: thread\_plan\_sched.c*

*1 │ #include <stdio.h>*

*2 │ #include <stdlib.h>*

*3 │ #include <pthread.h>*

*4 │ #include <sched.h>*

*5 │ #include <errno.h>*

*6 │*

*7 │ long int OPS\_AMOUNT = 1000000000;*

*8 │*

*9 │ void\* func\_fifo(void\* arg) {*

*10 │ int dummy = 0;*

*11 │ for (int i = 0; i < 100; i++) {*

*12 │ dummy++;*

*13 │ }*

*14 │ printf("Thread with SCHED\_FIFO END\n");*

*15 │ pthread\_exit(NULL);*

*16 │ }*

*17 │*

*18 │ void\* func\_rr(void\* arg) {*

*19 │ int dummy = 0;*

*20 │ for (int i = 0; i < OPS\_AMOUNT; i++) {*

*21 │ dummy++;*

*22 │ }*

*23 │ printf("Thread with SCHED\_RR END\n");*

*24 │ pthread\_exit(NULL);*

*25 │ }*

*26 │*

*27 │ int main() {*

*28 │ const int NUM\_THREADS = 10;*

*29 │ pthread\_t threads[NUM\_THREADS];*

*30 │ pthread\_attr\_t attrs[NUM\_THREADS];*

*31 │ struct sched\_param params[NUM\_THREADS];*

*32 │ int priorityFIFO = 50;*

*33 │ int priorityRR = 50;*

*34 │ for (int i = 0; i < NUM\_THREADS; i++) {*

*35 │ pthread\_attr\_init(&attrs[i]);*

*36 │ pthread\_attr\_setinheritsched(&attrs[i], PTHREAD\_EXPLICIT\_SCHED);*

*37 │ if (i % 2 == 0)*

*38 │ params[i].sched\_priority = priorityRR;*

*39 │ else*

*40 │ params[i].sched\_priority = priorityFIFO;*

*41 │ pthread\_attr\_setschedpolicy(&attrs[i], (i % 2) ? SCHED\_RR : SCHED\_FIFO);*

*42 │ pthread\_attr\_setschedparam(&attrs[i], &params[i]);*

*43 │ int err = pthread\_create(&threads[i], &attrs[i], (i % 2) ? func\_rr : func\_fifo, NULL);*

*44 │ if (err != 0) {*

*45 │ printf("Error creating thread %d", i);*

*46 │ }*

*47 │ }*

*48 │ puts("");*

*49 │ for (int i = 0; i < NUM\_THREADS; i++) {*

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

*51 │ }*

*52 │*

*53 │ printf("All threads finished\n");*

*54 │*

*55 │ return 0;*

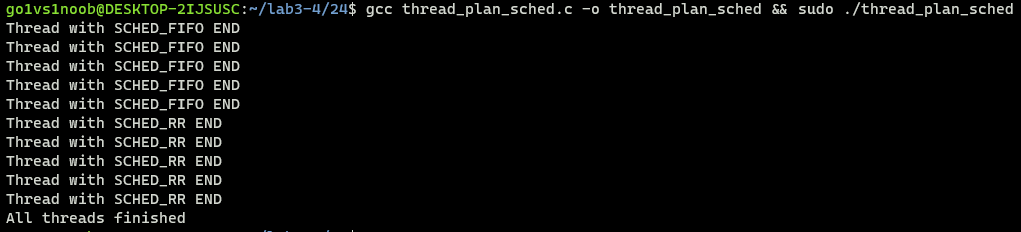
*56 │ }*

Рассмотрим примеры работы. Для начала количество операций для RR-потоков и FIFO-потоков оставим одинаковыми. Как и ожидалось, каждый раз разный порядок выполнения, так как количество вычислений одинаковое.



*Рисунок 44 – эксперимент 1*

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



*Рисунок 45 – эксперимент 2*

Модифицируем программу. Теперь все запускаемые потоки будут RR, ещё добавим маркеры для каждого из процессов. Содержание изменённой программы в листинге. Проведём эксперименты с изменённой программой.

*Листинг 5 – модифицированная программа для демонстрации политик планирования потоков*

*│ File: thread\_plan\_sched.c*

*1 │ #include <stdio.h>*

*2 │ #include <stdlib.h>*

*3 │ #include <pthread.h>*

*4 │ #include <sched.h>*

*5 │ #include <errno.h>*

*6 │*

*7 │ long long int OPS\_AMOUNT = 1000000000;*

*8 │*

*9 │ void\* func\_fifo(void\* arg) {*

*10 │ int marker = \*((int\*) arg);*

*11 │ int dummy = 0;*

*12 │ for (int i = 0; i < OPS\_AMOUNT; i++) {*

*13 │ dummy++;*

*14 │ }*

*15 │ printf("Thread with SCHED\_FIFO END: %d\n", marker);*

*16 │ pthread\_exit(NULL);*

*17 │ }*

*18 │*

*19 │ void\* func\_rr(void\* arg) {*

*20 │ int marker = \*((int\*) arg);*

*21 │ int dummy = 0;*

*22 │ for (int i = 0; i < OPS\_AMOUNT; i++) {*

*23 │ dummy++;*

*24 │ }*

*25 │ printf("Thread with SCHED\_RR END: %d\n", marker);*

*26 │ pthread\_exit(NULL);*

*27 │ }*

*28 │*

*29 │ int main() {*

*30 │ const int NUM\_THREADS = 5;*

*31 │ pthread\_t threads[NUM\_THREADS];*

*32 │ pthread\_attr\_t attrs[NUM\_THREADS];*

*33 │ struct sched\_param params[NUM\_THREADS];*

*34 │ int priorityFIFO = 50;*

*35 │ int priorityRR = 50;*

*36 │ for (int i = 0; i < NUM\_THREADS; i++) {*

*37 │ pthread\_attr\_init(&attrs[i]);*

*38 │ pthread\_attr\_setinheritsched(&attrs[i], PTHREAD\_EXPLICIT\_SCHED);*

*39 │ if (i % 2 == 0)*

*40 │ params[i].sched\_priority = priorityRR;*

*41 │ else*

*42 │ params[i].sched\_priority = priorityFIFO;*

*43 │ // pthread\_attr\_setschedpolicy(&attrs[i], (i % 2) ? SCHED\_RR : SCHED\_FIFO);*

*44 │ pthread\_attr\_setschedpolicy(&attrs[i], SCHED\_RR);*

*45 │ pthread\_attr\_setschedparam(&attrs[i], &params[i]);*

*46 │ // int err = pthread\_create(&threads[i], &attrs[i], (i % 2) ? func\_rr : func\_fifo, NULL);*

*47 │ int err = pthread\_create(&threads[i], &attrs[i], func\_rr, &i);*

*48 │ if (err != 0) {*

*49 │ printf("Error creating thread %d", i);*

*50 │ }*

*51 │ }*

*52 │ // puts("");*

*53 │ for (int i = 0; i < NUM\_THREADS; i++) {*

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

*55 │ }*

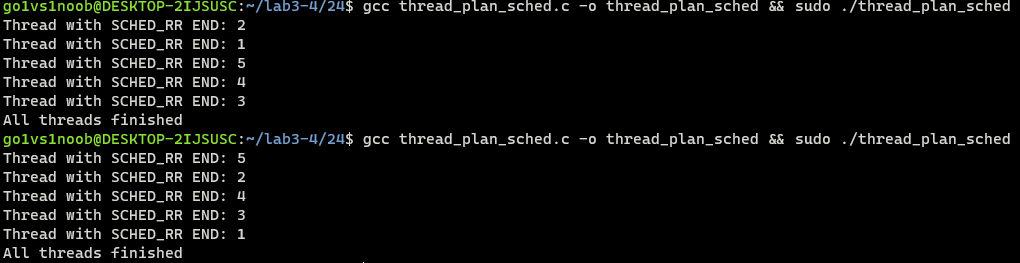
*56 │*

*57 │ printf("All threads finished\n");*

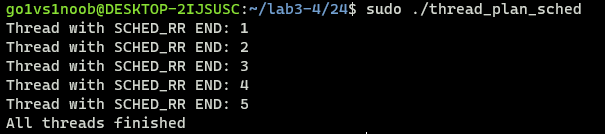
*58 │*

*59 │ return 0;*

*60 │ }*



*Рисунок 46 – результат работы программы с большим количеством вычислений*



*Рисунок 47 – результат работы программы с малым количеством вычислений*

Как видно, при большом количестве вычислений величина кванта превышается, потоки передают управление друг другу, следовательно порядок завершение произволен. При малом количестве вычислений потоки завершаются друг за другом.

Рассмотрим команду *pthread\_attr\_setinheritsched()* сначала с *PTHREAD\_INHERIT\_SCHED* (т. е. наследуемся от родительского), а потом – *PTHREAD\_EXPLICIT\_SCHED* (устанавливаем явно отдельно). Листинг программы в листинге 6. Результат работы на рисунке 48-49.

*Листинг 6 – программа для демонстрации разницы pthread\_inherit\_sched и pthread\_explicit\_sched*

*│ File: thread\_dummy.c*

*1 │ #include <stdio.h>*

*2 │ #include <stdlib.h>*

*3 │ #include <pthread.h>*

*4 │ int i = 0;*

*5 │*

*6 │ void switch\_policy(int policy)*

*7 │ {*

*8 │ switch (policy)*

*9 │ {*

*10 │ case SCHED\_FIFO:*

*11 │ printf("policy SCHED\_FIFO\n");*

*12 │ break;*

*13 │ case SCHED\_RR:*

*14 │ printf("policy SCHED\_RR\n");*

*15 │ break;*

*16 │ case SCHED\_OTHER:*

*17 │ printf("policy SCHED\_OTHER\n");*

*18 │ break;*

*19 │ case -1:*

*20 │ perror("policy SCHED\_GETSCHEDULER");*

*21 │ break;*

*22 │ default:*

*23 │ printf ("policy Неизвестная политика планирования\n");*

*24 │ }*

*25 │ }*

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

*27 │ printf("Child thread started.\n");*

*28 │ /\* Do some work here \*/*

*29 │ printf("Thread's ");*

*30 │ switch\_policy(sched\_getscheduler(0));*

*31 │ while (i<10){*

*32 │ i++;*

*33 │ }*

*34 │ printf("Child thread finished.\n");*

*35 │ pthread\_exit(NULL);*

*36 │ }*

*37 │*

*38 │ int main() {*

*39 │ pthread\_t child\_thread;*

*40 │ pthread\_attr\_t attr;*

*41 │ struct sched\_param shdprm;*

*42 │ int ret;*

*43 │*

*44 │ pid\_t pid = getpid();*

*45 │ shdprm.sched\_priority = 1;*

*46 │*

*47 │ if (sched\_setscheduler(pid, SCHED\_FIFO, &shdprm) == -1)*

*48 │ {*

*49 │ perror("SCHED\_SETSCHEDULER");*

*50 │ }*

*51 │*

*52 │ printf("Политика процесса: ");*

*53 │ switch\_policy(sched\_getscheduler(pid));*

*54 │*

*55 │ /\* Initialize thread attributes \*/*

*56 │ ret = pthread\_attr\_init(&attr);*

*57 │ if (ret != 0) {*

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

*59 │ exit(EXIT\_FAILURE);*

*60 │ }*

*61 │*

*62 │ /\* Set thread scheduling to inherit from parent \*/*

*63 │ ret = pthread\_attr\_setinheritsched(&attr, PTHREAD\_INHERIT\_SCHED);*

*64 │ //ret = pthread\_attr\_setinheritsched(&attr, PTHREAD\_EXPLICIT\_SCHED);*

*65 │ if (ret != 0) {*

*66 │ fprintf(stderr, "pthread\_attr\_setinheritsched() failed.\n");*

*67 │ exit(EXIT\_FAILURE);*

*68 │ }*

*69 │*

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

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

*72 │ if (ret != 0) {*

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

*74 │ exit(EXIT\_FAILURE);*

*75 │ }*

*76 │*

*77 │ int thread\_policy;*

*78 │ pthread\_attr\_getschedpolicy(&attr, &thread\_policy);*

*79 │*

*80 │*

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

*82 │ pthread\_join(child\_thread, NULL);*

*83 │*

*84 │ /\* Cleanup thread attributes \*/*

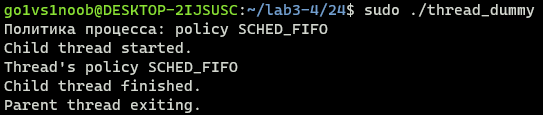
*85 │ pthread\_attr\_destroy(&attr);*

*86 │*

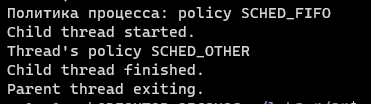
*87 │ printf("Parent thread exiting.\n");*

*88 │ return 0;*

*89 │ }*



*Рисунок 48 – результат с INHERIT*



*Рисунок 49 – результат с EXPLICIT*

Как видно, если применять *EXPLICIT* политика потока – *SCHED\_OTHER*, поскольку не была задана явно. Если же *INHERIT* – политика наследуется от политики процесса (который его породил).

Теперь не будем задавать процессу процедуру планирования. Зададим её потоку 1, который, в свою очередь, породит поток 2. Рассмотрим наследование при таком варианте. Содержание программы в листинге 7, результат работы – на рисунке 50.

*Листинг 7 – наследование процедур планирования при создании потока из другого потока*

*│ File: thread\_dummy\_2.c*

*1 │ #include <stdio.h>*

*2 │ #include <stdlib.h>*

*3 │ #include <pthread.h>*

*4 │ #include <unistd.h>*

*5 │ #include <signal.h>*

*6 │*

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

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

*9 │ void sigusr1\_handler(int signal\_number);*

*10 │ pthread\_t thread2;*

*11 │*

*12 │ void switch\_policy(int policy)*

*13 │ {*

*14 │ switch (policy)*

*15 │ {*

*16 │ case SCHED\_FIFO:*

*17 │ printf("policy SCHED\_FIFO\n");*

*18 │ break;*

*19 │ case SCHED\_RR:*

*20 │ printf("policy SCHED\_RR\n");*

*21 │ break;*

*22 │ case SCHED\_OTHER:*

*23 │ printf("policy SCHED\_OTHER\n");*

*24 │ break;*

*25 │ case -1:*

*26 │ perror("policy SCHED\_GETSCHEDULER");*

*27 │ break;*

*28 │ default:*

*29 │ printf("policy Неизвестная политика планирования\n");*

*30 │ }*

*31 │ }*

*32 │*

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

*34 │ {*

*35 │ pthread\_t thread1;*

*36 │ int result1;*

*37 │ pthread\_attr\_t attr1;*

*38 │*

*39 │ // Установка обработчика сигнала SIGUSR1*

*40 │ signal(SIGUSR1, sigusr1\_handler);*

*41 │*

*42 │ pthread\_attr\_init(&attr1);*

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

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

*45 │ // Установка параметров планирования для первого потока*

*46 │ struct sched\_param shdprm;*

*47 │ shdprm.sched\_priority = 1;*

*48 │ pthread\_attr\_setschedparam(&attr1, &shdprm);*

*49 │*

*50 │ // Создание первого потока*

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

*52 │*

*53 │*

*54 │*

*55 │ // Ожидание завершения первого потока*

*56 │ pthread\_join(thread1, NULL);*

*57 │ printf("Главный поток завершен\n");*

*58 │ exit(EXIT\_SUCCESS);*

*59 │ }*

*60 │*

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

*62 │ {*

*63 │*

*64 │ printf("Поток 1 создан\n");*

*65 │ printf("Политика потока 1: ");*

*66 │ switch\_policy(sched\_getscheduler(0));*

*67 │ printf("Приоритет потока 1: %d", );*

*68 │*

*69 │*

*70 │ pthread\_attr\_t attr2;*

*71 │*

*72 │ pthread\_attr\_init(&attr2);*

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

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

*75 │*

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

*77 │*

*78 │ // Установка параметров планирования для второго потока*

*79 │ struct sched\_param param;*

*80 │ param.sched\_priority = 2;*

*81 │ pthread\_attr\_setschedparam(&attr2, &param);*

*82 │*

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

*84 │*

*85 │ // Ждем 5 секунд и затем останавливаем второй поток*

*86 │ sleep(5);*

*87 │ pthread\_kill(thread2, SIGUSR1);*

*88 │*

*89 │ // Ожидание завершения второго потока*

*90 │ pthread\_join(thread2, NULL);*

*91 │ printf("Поток 1 завершен\n");*

*92 │*

*93 │ pthread\_attr\_destroy(&attr2);*

*94 │ return NULL;*

*95 │ }*

*96 │*

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

*98 │ {*

*99 │ printf("Поток 2 создан\n");*

*100 │ printf("Политика потока 2: ");*

*101 │ switch\_policy(sched\_getscheduler(0));*

*102 │ while (1)*

*103 │ {*

*104 │ sleep(1);*

*105 │ }*

*106 │ return NULL;*

*107 │ }*

*108 │*

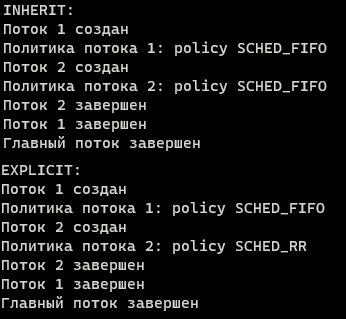
*109 │ void sigusr1\_handler(int signal\_number)*

*110 │ {*

*111 │ printf("Поток 2 завершен\n");*

*112 │ pthread\_exit(NULL);*

*113 │ }*



*Рисунок 50 – результат работы программы, демонстрирующей наследование процедур планирования при создании потока из другого потока*

Программа работает как ожидалось, поведение наследования от процесса и наследования от потока не различаются.

Напишем программы, задающие приоритет извне (листинг 8) и внутри программы (листинг 9) для двух разных потоков. Результат работы программ на рисунке 51.

*Листинг 8 – программа, задающая приоритет извне*

*│ File: thread\_dummy\_4.c*

*1 │*

*2 │ #include <pthread.h>*

*3 │ #include <sched.h>*

*4 │ #include <stdio.h>*

*5 │ #include <stdlib.h>*

*6 │ #include <unistd.h>*

*7 │*

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

*9 │ int prio = \*((int\*) arg);*

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

*11 │ // sleep(1);*

*12 │ return NULL;*

*13 │ }*

*14 │*

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

*16 │ pthread\_t thread1, thread2;*

*17 │ pthread\_attr\_t attr;*

*18 │ struct sched\_param param;*

*19 │*

*20 │*

*21 │*

*22 │ if (argc < 2) {*

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

*24 │ return 1;*

*25 │ }*

*26 │*

*27 │ int priority1 = atoi(argv[1]);*

*28 │ int priority2 = atoi(argv[2]);*

*29 │*

*30 │ // Set the scheduling policy and priority for the newly created pthread*

*31 │ pthread\_attr\_init(&attr);*

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

*33 │ param.sched\_priority = priority1;*

*34 │ pthread\_attr\_setschedparam(&attr, &param);*

*35 │*

*36 │ pthread\_attr\_init(&attr);*

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

*38 │ param.sched\_priority = priority2;*

*39 │ pthread\_attr\_setschedparam(&attr, &param);*

*40 │*

*41 │*

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

*43 │ if (pthread\_create(&thread1, &attr, thread\_function, &priority1) != 0) {*

*44 │ perror("pthread\_create");*

*45 │ return 1;*

*46 │ }*

*47 │*

*48 │ if (pthread\_create(&thread2, &attr, thread\_function, &priority2) != 0) {*

*49 │ perror("pthread\_create");*

*50 │ return 1;*

*51 │ }*

*52 │*

*53 │*

*54 │ printf("Main started\n");*

*55 │ // Now join the pthread*

*56 │ pthread\_join(thread1, NULL);*

*57 │ pthread\_join(thread2, NULL);*

*58 │ printf("Main end\n");*

*59 │ return 0;*

*60 │ }*

*Листинг 9 – программа, задающая приоритет программно*

*│ File: thread\_dummy\_3.c*

*1 │ #include <pthread.h>*

*2 │ #include <stdio.h>*

*3 │*

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

*5 │ int prio = \*((int\*) arg);*

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

*7 │ return NULL;*

*8 │ }*

*9 │*

*10 │ int main(void) {*

*11 │ pthread\_t thread1, thread2;*

*12 │ int prio1 = 1, prio2 = 99;*

*13 │*

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

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

*16 │*

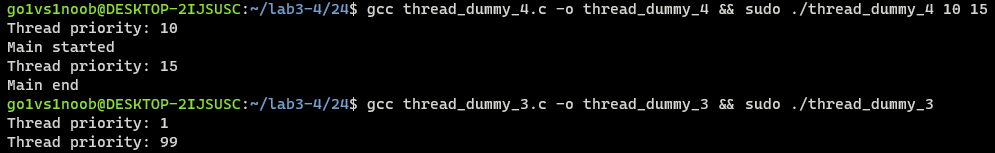
*17 │ pthread\_join(thread1, NULL);*

*18 │ pthread\_join(thread2, NULL);*

*19 │*

*20 │ return 0;*

*21 │ }*



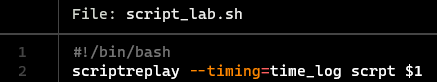
*Рисунок 51 – результат работы программы, задающей приоритет извне/программно*

В первой программе задаются приоритеты 10 и 15, а во второй – 1 и 99 для потока 1 и 2 соответственно. Как видно, принципиальных различий в задаче приоритетов нет.

Подытожив, можно сказать, что на данной системе (WSL, Ubuntu) нет ограничений на изменение процедур планирования, приоритета для потоков одного процесса.

1. Создадим скрипт, выполняющий нашу работу автоматически. Для

этого воспользуемся утилитой Linux *script*. С помощью команды *script --timing=time\_log scrpt* все действия в терминале (с соблюдением временных промежутков) будет записан в scrpt (временные промежутки в time\_log). Можем его воспроизвести с помощью команды *scriptreplay --timing=time\_log scrpt* варьируя скорость воспроизведения. Напишем небольшой bash скрипт, воспроизводящий лабораторную работу с задаваемой скоростью. Его содержание на рисунке 52.



*Рисунок 52 – bash скрипт, выполняющий лабораторную работу*

**Вывод.**

В ходе выполнения лабораторной работы ознакомились с возможностью управлением процессов посредством сигналов, многопотоковым функционированием. Рассмотрен перечень сигналов, поддерживаемых процессами. Организованы различные посылки сигналов, их фиксация, обработка, перехватывание, переопределение. Ознакомились с выполнением команды *nice()* и *getpriority()*, *nohup()*. Определили некоторые системные параметры (максимальные, минимальные *uid*, *pid*). Рассмотрено многопотоковое программирование (управление потоками из процесса, из другого потока, создание обработчиков сигнала, перехват сигналов). Проанализированы процедуры планирования для процессов и потоков одного процесса. Создан скрипт, выполняющий работу автоматически.

**список использованных источников**

1. Geeks for Geeks – <https://www.geeksforgeeks.org/>

2. Linux.org – <https://www.linux.org.ru/forum/general/>

3. Linux Documentation – [linux.die.net/man](file:///C:\Users\go1vs1noob\Documents\Учеба\ОС\linux.die.net\man)