**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №4**

**по дисциплине «Операционные системы»**

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

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

Санкт-Петербург

2023

## **Цель работы.**

* 1. Задание.

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.

*Многонитевое функционирование*

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).

* 1. Выполнение работы

1. Рассмотрим 3 ситуации:
   1. Процесс-родитель запускает процесс-потомок и ожидает его завершения
   2. Процесс-родитель запускает процесс-потомок и завершается
   3. Процесс-родитель запускает процесс-потомок и не дожидается его завершения

Для всех случаев получим соответствующие таблицы процессов, добьемся такого поведения от процессов меняя параметры системных вызовов и длительность выполнения процессов. Перенаправим выходные данные 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.

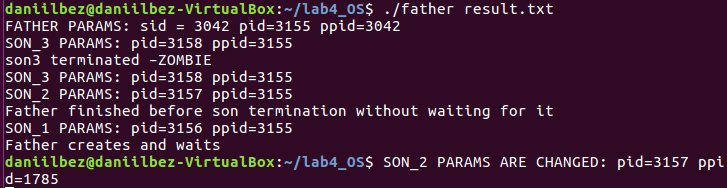


Рисунок 1 – результат работы программы father

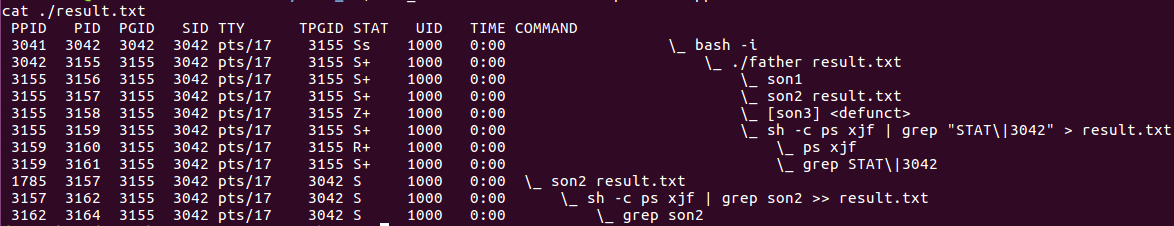


Рисунок 2 – таблица процессов во время выполнения программы

Из содержимого таблицы можно сделать следующие выводы – т.к. процесс son2 работал дольше, чем родительский процесс (в коде son2 выполняется задержка 20 с), то по истечению времени жизни родительского процесса он стал самостоятельным (ppid изменился на 1785, так же это иллюстрируется деревом процессов), при этом на момент создания он являлся процессом потомком.

Рассмотрим, что произошло с процессом son3. Как видно из таблицы он стал zombie-процессом (STAT = Z+), это означает, что ресурсы, выделенные для данного процесса были освобождены, но он остался существовать формально.

1. С помощью команды kill -l ознакомимся с перечнем сигналов, поддерживаемых процессами. Рассмотрим системные вызовы *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. Выведем все сигналы с помощью команды kill -l

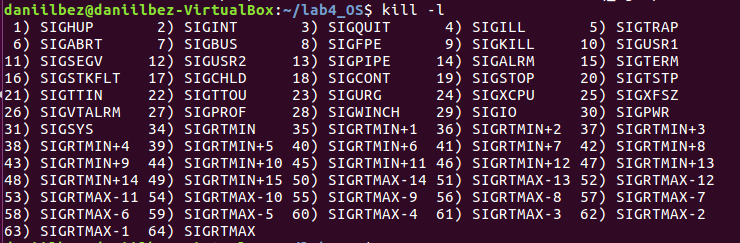


Рисунок 3 – список сигналов, поддерживаемых kill

Системный вызов *signal(signum, handler)* позволяет установить свой обработчик сигнала, (handler в том числе может быть написан самостоятельно): устанавливает signum в обработчик handler.

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.

Выполним данную программу:



Рисунок 4 – результат выполнения программы father

Каждой программе подаётся сигнал *SIGTERM*. Первая программа обрабатывает его стандартно и завершает свою работу. Вторая программа игнорирует его (в результате чего выводится строка “SON2: signal ignored”), третья – перехватывает, выводя в консоль строку, говорящую о том, что сигнал обработан. Рассмотрим таблицу процессов до и после посылки сигналов

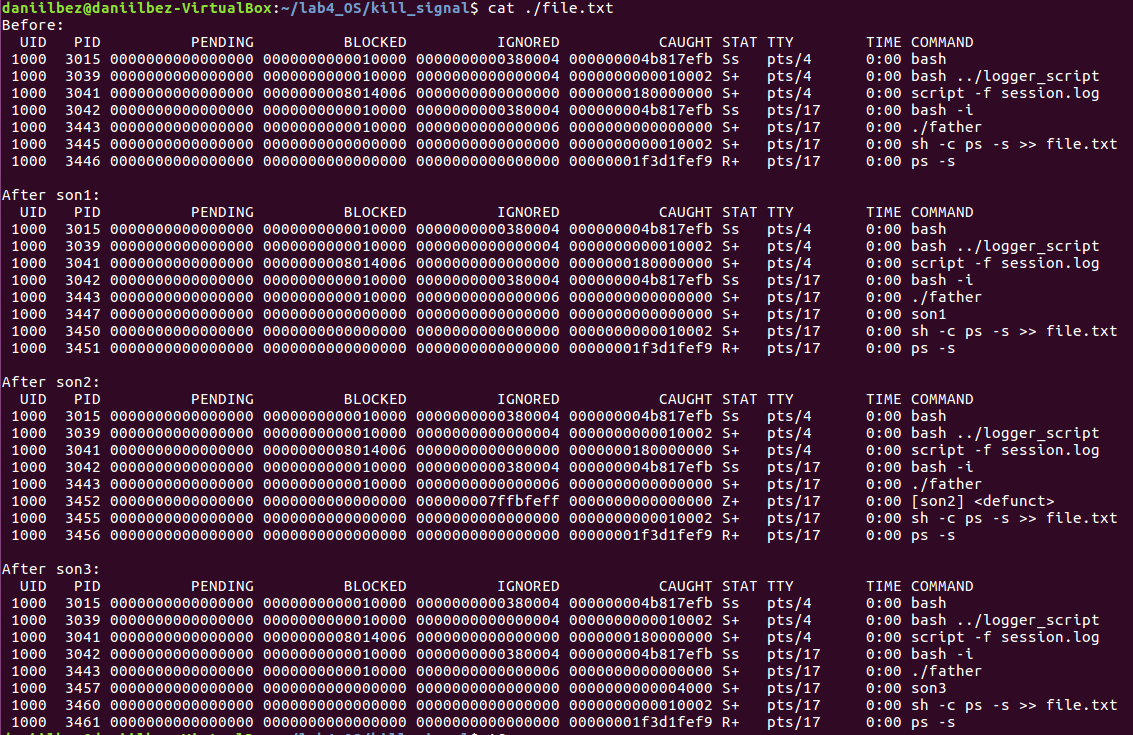


Рисунок 5 – ps -s после отправки сигналов

Рассмотрим данную таблицу – поля BLOCKED, IGNORED, CAUGHT показывают, каким образом процесс отреагировал на посланный сигнал. Таким образом нетрудно заметить, что процесс son1 обработал сигнал SIGTERM стандартным образом – завершился после его получения, son2 – проигнорировал его (поэтому в консоль была выведена надпись о том, что сигнал проигнорирован), а процесс son3 – перехватил сигнал и обработал его с помощью кастомного обработчика, который вывел в консоль строку о том, что сигнал был перехвачен и обработан.

Проанализируем наследование диспозиции сигналов при создании процессов на этапе *fork()* и *exec()*. Напишем программу *sig\_father\_copy.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

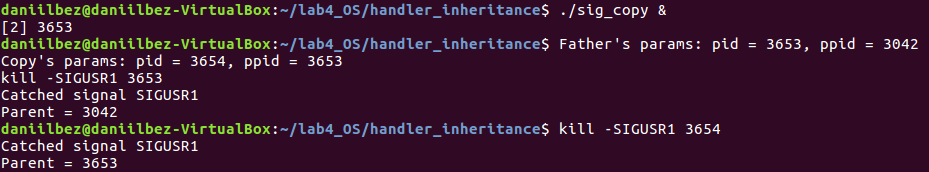


Рисунок 6 – результат работы программы sig\_copy

Как нетрудно заметить, несмотря на то, что мы не задавали обработчиков для процесса-потомка, он обработал полученный сигнал аналогичным процессу-родителю образом, это позволяет сделать вывод, что при ветвлении с помощью fork(), наследуется диспозиция сигналов, даже если родительский процесс завершен. Повторим эксперимент, но в этот раз организуем наследование с помощью fork() и execl().

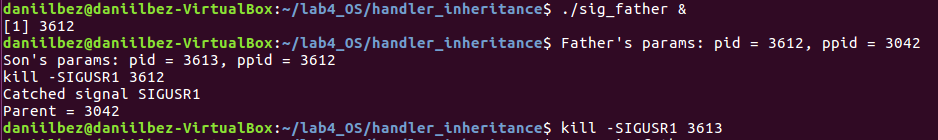


Рисунок 7 – результат работы программы sig\_father

В данном же случае для родительского процесса был вызван обработчик, но для процесса-потомка данный обработчик оказался не задан, следовательно, процесс просто завершился, ничего не выведя. Отсюда следует, что при создании процесса с помощью *fork()* и *exec()* диспозиция сигналов не наследуется.

1. Запустим 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.

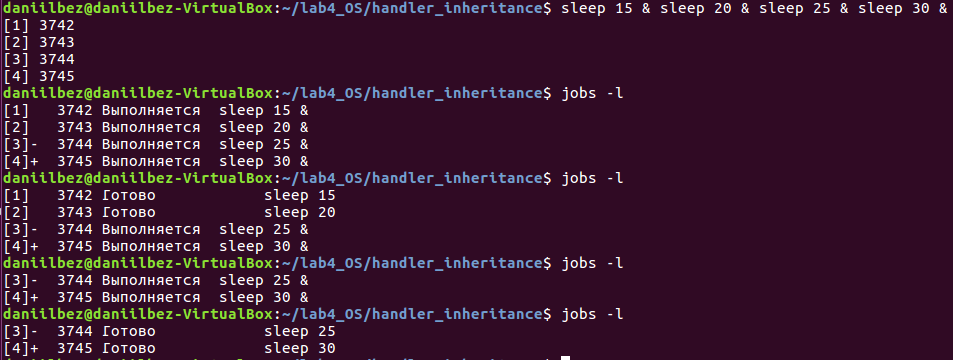


Рисунок 8 – результат команды jobs

Из рисунка 8, выполнение задач начинается по порядку, причем параллельно друг другу. С помощью утилиты *fg* повысим приоритет задачи 4. В результате сразу начинается выполняться задача 4, не в фоновом режиме

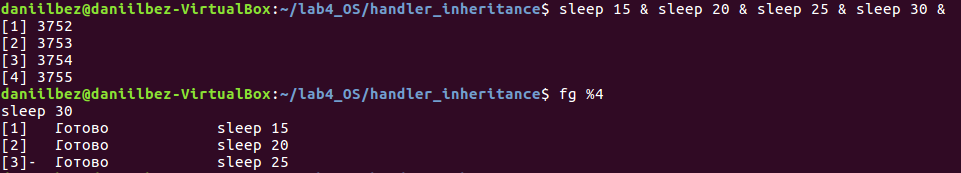


Рисунок 9 – результат изменения приоритета задачи 4

С помощью команды *kill* отменим одно из невыполненных заданий (рисунок 10):

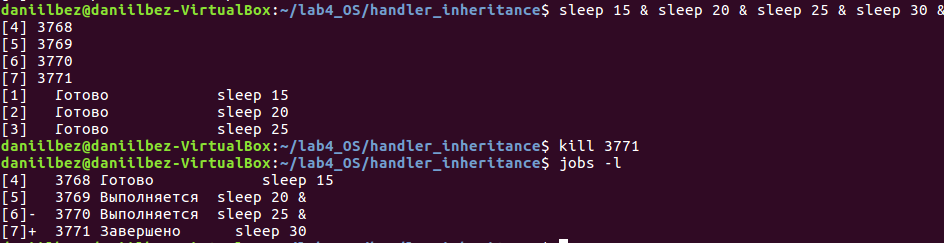


Рисунок 10 - результат отмены задания [7]

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 11, результат работы – на рисунке 12.

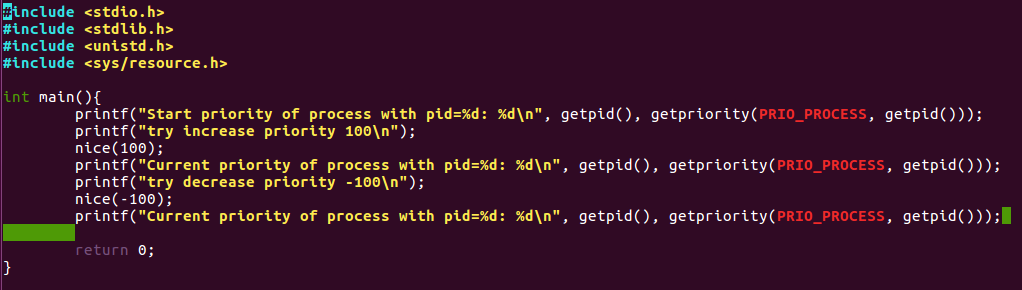


Рисунок 11 – структура программы

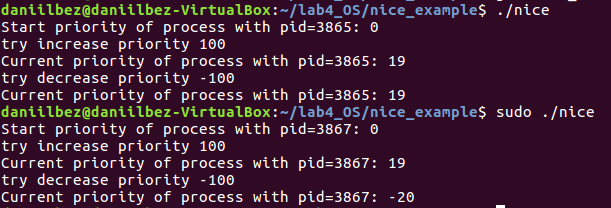


Рисунок 12 – результат выполнения

Запустили программу как от имени суперпользователя, так и в обыкновенном режиме. Нетрудно заметить, что для того, чтобы изменить приоритет, необходимо выполнять программу именно в режиме sudo. Кроме того, несмотря на то, что мы пытались увеличить и уменьшить приоритет на 100, границы для изменения приоритетов: -19…20.

Разница в приоритетах для системных и пользовательских процессов есть: обычно большинство системных процессов, отвечающих за управление системы имеют более высокий приоритет, чем пользовательские (рисунок 13). Это обусловлено тем, что системные процессы обеспечивают более низкоуровневые функции.



Рисунок 13 – приоритеты системных процессов

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



Рисунок 14 – пример процесса с приоритетом реального времени

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



Рисунок 15 – 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 16 – запуск скрипта с nohup

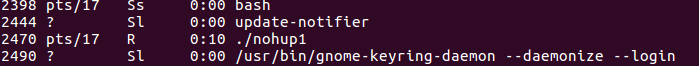


Рисунок 17 – список процессов после nohup

Таким образом, при выходе из системы процесс *nohup1*  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 18 – запуск второго примера

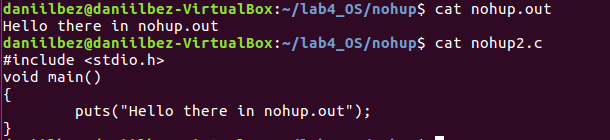


Рисунок 19 – результат работы nohup с записью строки

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).

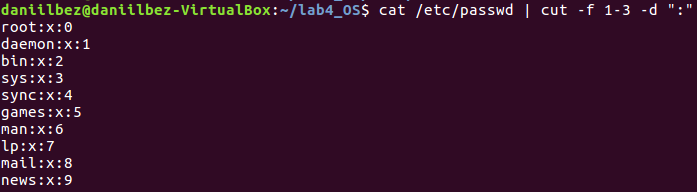


Рисунок 20 – демонстрация UID root

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

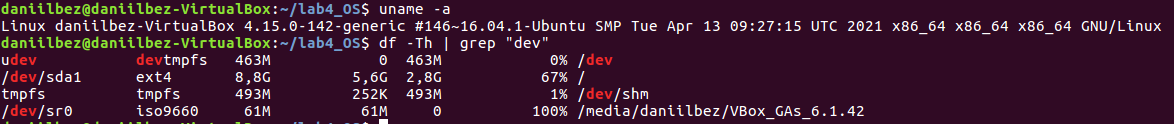


Рисунок 21 – параметры ФС и версия ядра

Для файловой системы *ext4* максимальное значение *uid* составляет 4294967295 (2^32 – 1). Однако в самой ОС это значение может быть другим (рисунок 22). Данный параметр можно найти в файле /etc/login.defs

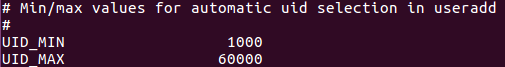


Рисунок 22 – минимальный и максимальный UID

Минимальное значение *PID* принадлежит процессу *init* и равняется 1. Этот процесс запускается после загрузки системы и является родительским процессом для всех остальных. *PID* *0* зарезервирован ядром.

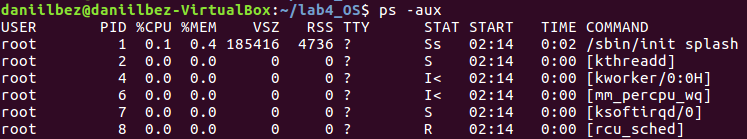


Рисунок 23 – pid процессов

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



Рисунок 24 – максимальный pid

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



Рисунок 25 – Пример системных процессов

7. 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 26 – структура программы, иллюстрирующей создание потоков

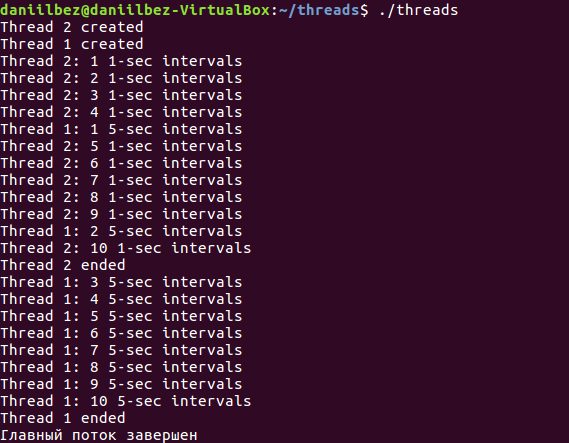


Рисунок 27 – результат выполнения представленной программы

8. 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). Вывод программы направим в файл, и будем анализировать процессы с помощью htop.

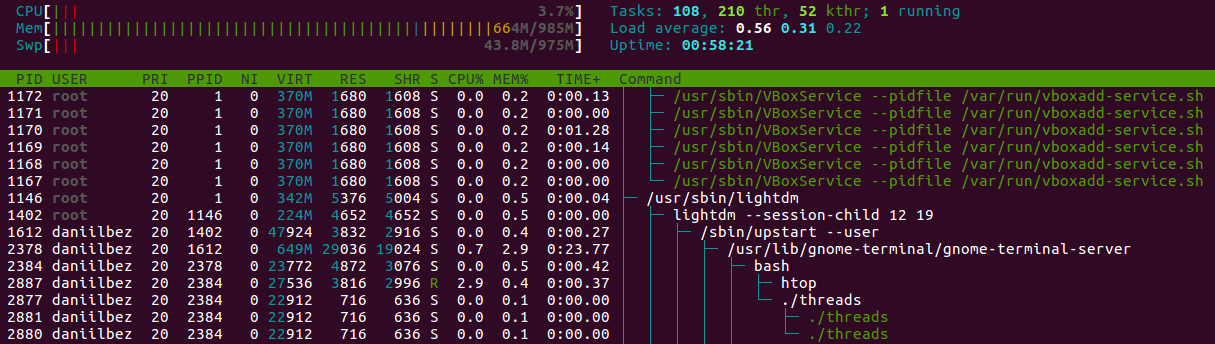


Рисунок 28 – потоки при запуске программы

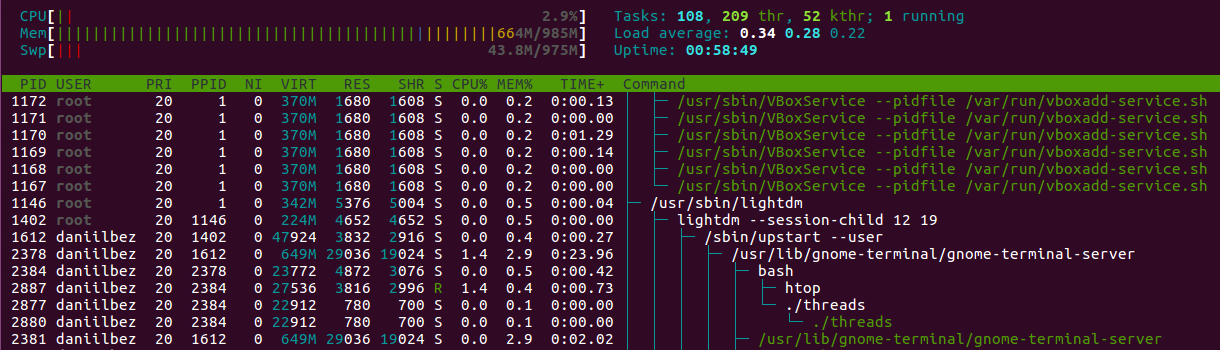


Рисунок 29 – потоки после завершения потока 2

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*.

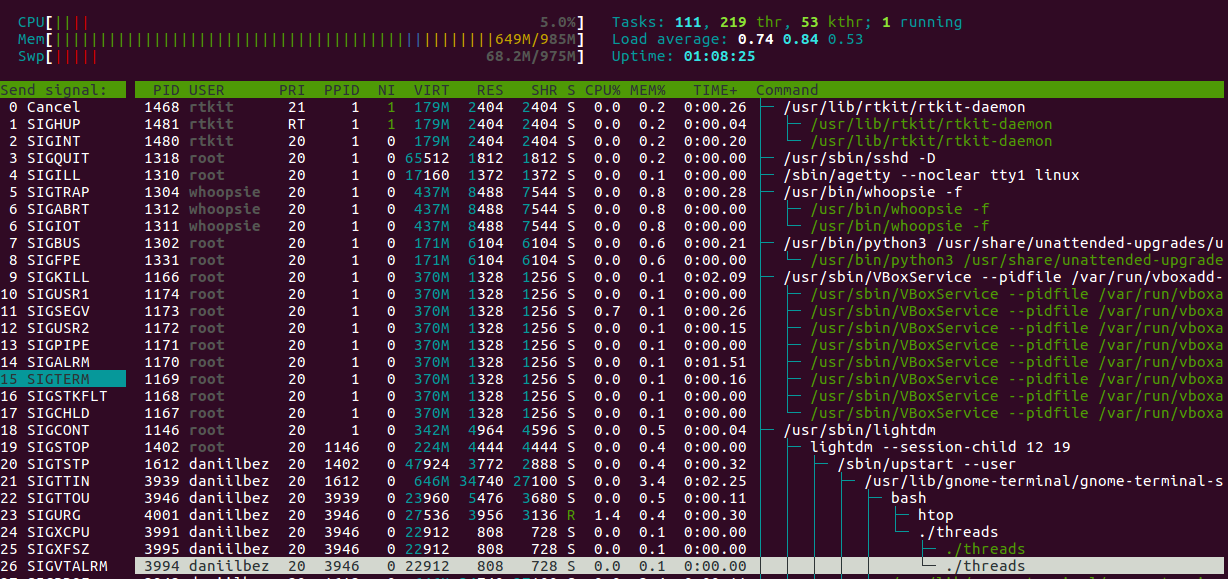


Рисунок 30 – kill второго потока

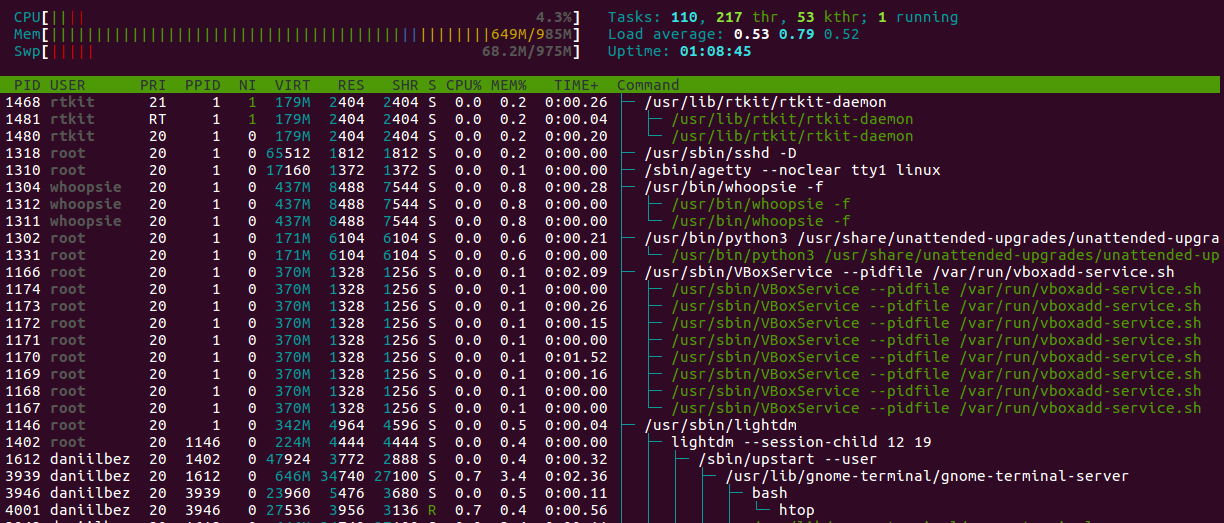


Рисунок 31 – результат отправления потока SIGTERM второму потоку

Как нетрудно заметить, после отправления данного сигнала в один из потоков -завершилась программа целиком.

9. Модифицируем программу так, чтобы управление второй нитью

передавалось посредством сигнала SIGUSR1 из первой нити. Для этого напишем программу так: зададим обработчик сигнала SIGUSR1 в функции sigusr1\_handler (внутри неё выведется сообщение об успешном принятии сигнала и завершится поток 2). В функцию создания потока 1 добавим блок с созданием, а затем и завершением через 5 секунд после создания потока 2.

В данном эксперименте ожидаем, что 4 раза будет выведена надпись из потока 2, затем он будет завершен (на 5 секундном интервале поток будет завершен).

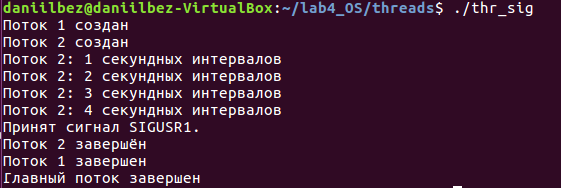


Рисунок 32 – результат выполнения эксперимента

Результат совпал с ожидаемым.

10. Модифицируем программу так, чтобы внутри обработчика сигнала

поток завершался посредством функции pthread\_exit(NULL). Сравним результат с предыдущим. Для этого изменим обработчик:

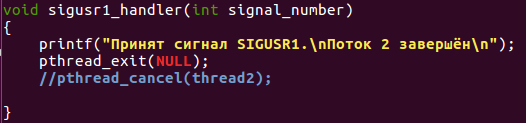


Рисунок 33 – измененный обработчик сигнала

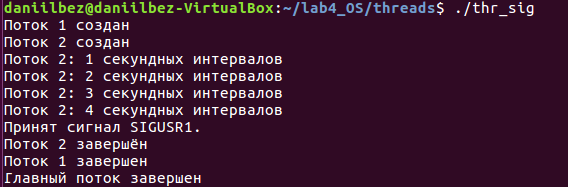


Рисунок 34 – результат выполнения модифицированной программы

Хотя результат работы совпал, между данными функциями есть некоторая разница, а именно то, в какой момент функции завершают поток. *Pthread\_exit()* 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.

11. 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 35.

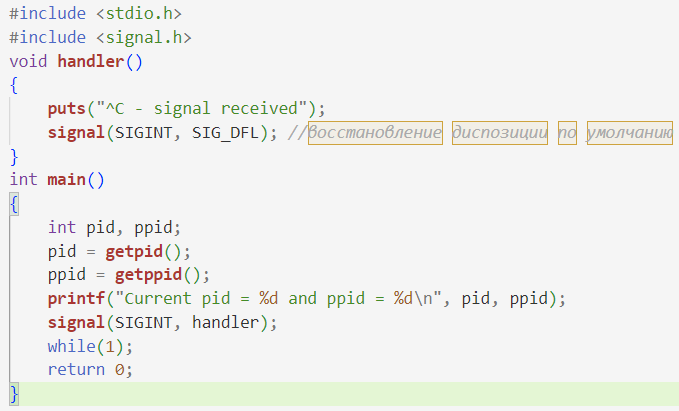


Рисунок 35 – программа, перехватывающая ^C единожды

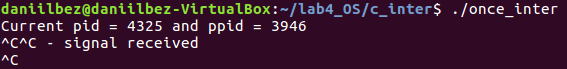


Рисунок 36 – результат работы и вызова 2 прерываний

Как видно из результата выполнения программы – после вызова прерывания в первый раз, сработал наш handler, который вывел сообщение и восстановил диспозицию по умолчанию, после срабатывания второго прерывания, обработчик уже не вызывался, поэтому программа прекратила свою работу.

Изменим нашу программу таким образом, что теперь срабатывание обработчика будет увеличивать значение статической переменной, после достижения ей значения 5, восстановим стандартную диспозицию.



Рисунок 37 – результат работы измененной программы

Проделаем аналогичную работу для другой комбинации клавиш (Ctrl + Z). С помощью данной команды мы отправляем сигнал SIGTSTP, который «ставит на паузу» программу.

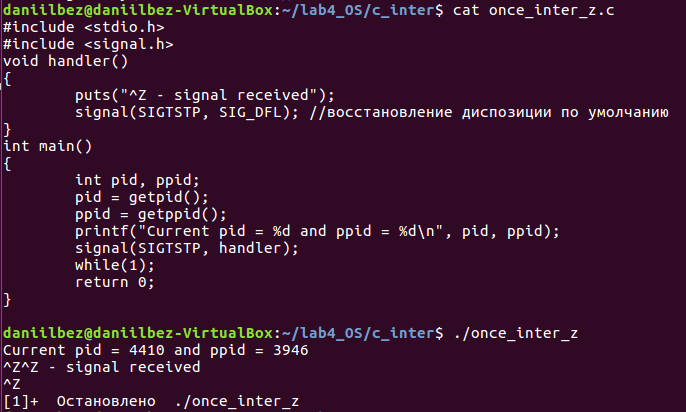


Рисунок 38 – текст программы с однократной обработкой сигнала и результат ее работы

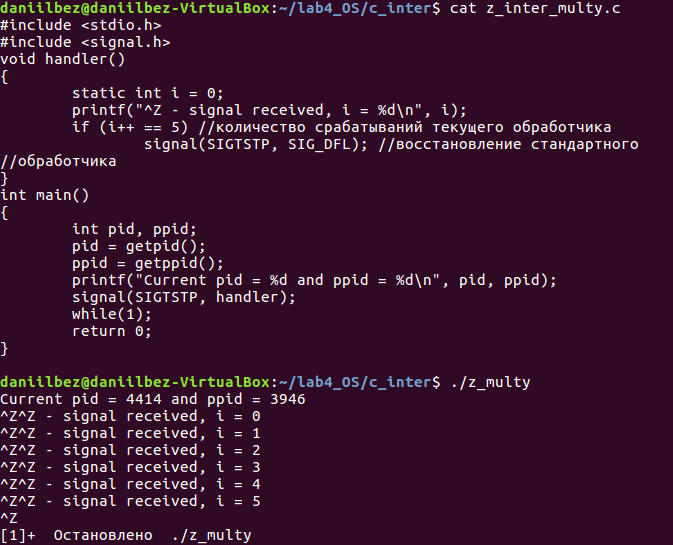


Рисунок 39 – текст программы со множественной обработкой сигнала и результат ее работы

12. С помощью утилиты kill выведем список сигналов и дадим краткую

характеристику.

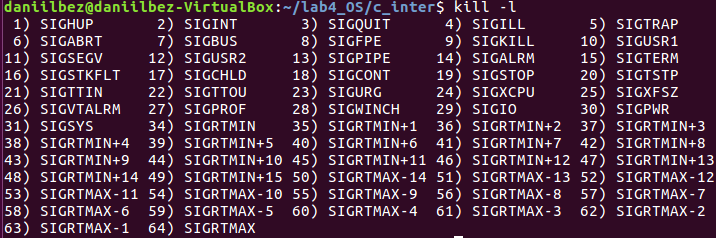


Рисунок 40 – список сигналов kill

* Процессы от 1 до 31 заданы стандартом *POSIX* и выполняют определенные функции. Например, завершение процесса (*SIGTERM*), неожиданное прерывание процесса (*SIGINT*), остановка процесса (*SIGSTOP*)
* Процессы 32 и 33 не отображаются – они используются POSIX-ядрами
* 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). Их предназначение не обязательно определено, как говорилось раннее, они могут быть использованы при разработке. По умолчанию сигналы от 34 и выше завершают работу программы.

Попробуем послать данные сигналы фоновому процессу, посмотрим, что произойдет.

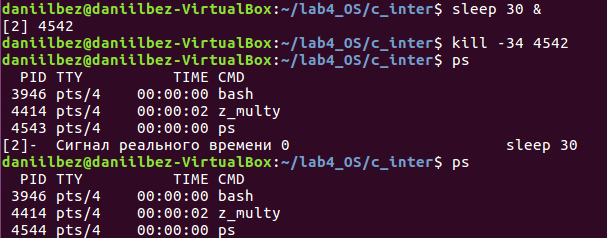
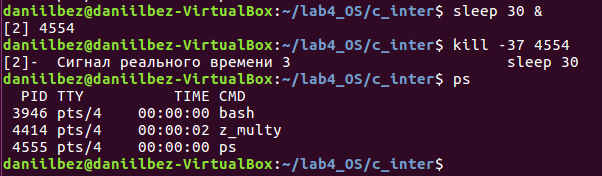
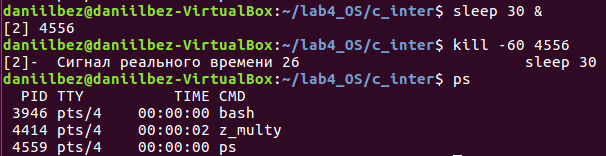


Рисунок 41 – результат использования SIGRTMIN

Нетрудно заметить, что после того, как мы послали данный сигнал, выполнение процесса прекратилось – подтверждает это вывод утилиты ps.

Приведем еще два примера – kill -37, kill -60

Результаты ожидаемы, и совпали с результатами первого эксперимента.

Вывод