**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МОЭВМ**

отчет

**по лабораторной работе №3**

**по дисциплине «Операционные системы»**

Тема: Процессы

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

Санкт-Петербург

2023

**Цель работы.**

**Ход работы.**

1. Создадим программу на основе исходного файла с

псевдораспределением вычислений посредством порождения процесса-потомка командой *fork()*. После выполнения системного вызова  
*fork* оба процесса продолжают выполнение с одной и той же точки.  
*fork* возвращает в породивший процесс идентификатор порожденного  
процесса, а в порожденный процесс — *NULL*.   
В результате выполнения программы будут выведены  
идентификаторы каждого процесса и его родителя, а также дважды  
фраза «Завершение процесса», что свидетельствует об исполнении  
одного и того же кодового сегмента обоими процессами (рисунок 1):

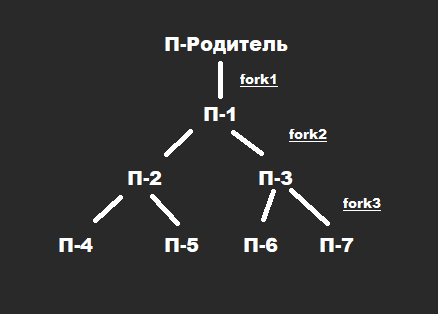


*Рисунок 1 – код программы для демонстрации системного вызова fork()*

Чтобы лучше понять принцип наследования процессов в *fork()* рассмотрим следующий пример кода (рисунок 2). В нём трижды вызывается *fork()*, и выводится на экран строка. В итоге получится 8 процессов с иерархией, показанной на рисунке 3. Запустим программу, чтобы подтвердить рассуждения (рисунок 2):



*Рисунок 2 – код программы для демонстрации иерархии fork()*



*Рисунок 3 – иерархия процессов fork()*

Рассмотрим то, как между родственными процессами делится память. Для этого составим программу и запустим её. Устройство программы и результат работы на рисунке 4:

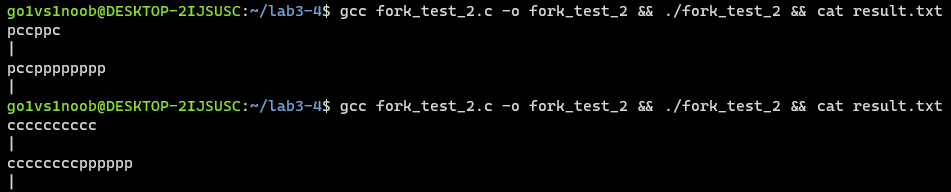


*Рисунок 4 – демонстрация разделения памяти между процессами fork()*

В результате видно: несмотря на то, что адрес *globalVar* у обоих процессов один – этот адрес указывает на разные участки в физической памяти.

1. Поскольку *printf()* будет приводить к передаче управления другому

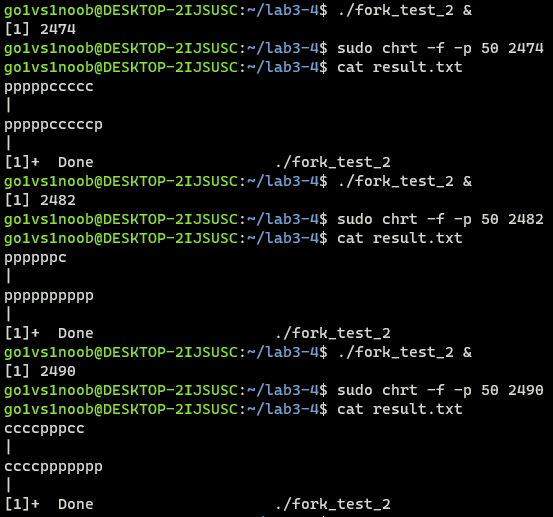
процессу, порядок исполнения может не соответствовать порядку вывода на терминал. Для более точных результатов перепишем программу так, чтобы результат вызовов записывался в массив символов *operationsArr*. Программа составлена так, что и потомок, и родитель имеют доступ к одному и тому же массиву. Изначально же, несмотря на то что, и для процесса-родителя, и для процесса-ребёнка адрес на массив совпадал – этому адресу соответствовали разные места в физической памяти (т.е. совпадало пространство виртуальных адресов, но не физических). В массив символов будем поочерёдно записывать символы ‘с’ и ‘p’ (c – child, p – parent), содержание массива запишем в файл. Помимо этого, заменим единичные вычисления на циклы. Поскольку процессы «делят» один массив между собой – происходит конкуренция процессов, а вывод непредсказуем. Содержимое программы находится в приложении А (*fork\_test\_2.c*). Результат работы – на рисунке 5.



*Рисунок 5 – результат работы программы, в которой у процессов есть доступ к одинаковым физическим местам памяти*

Как видно, каждый раз может получится разный порядок вызова процессов. При этом несмотря на то, что операций десять, а процессов два (т. е. должно быть 20 элементов в массиве) – число элементов в конечной строке тоже непредсказуемо, потому что процессы выполняются параллельно.

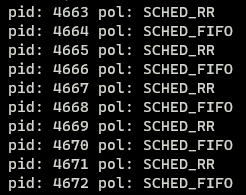
Изменим процедуру планирования на *FIFO* и повторим эксперимент (рисунок 3). Для этого немного изменим программу. В самом начале *main()* вызовем функцию *sleep(10)*, чтобы успеть сменить процедуру планирования. Результат на рисунке 6:



*Рисунок 6 – результат работы программы, в которой у процессов есть доступ к одинаковым физическим местам памяти с планировщиком FIFO*

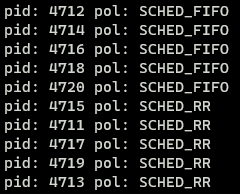
Как видно, результат работы всё также непредсказуем из-за того, что процессы делят один и тот же физический адрес.

Напишем программу (*another\_sched.c*), в которой рассмотрим конкуренцию множества процессов при задании разных политик на них. Для этого организуем цикл, в котором будут вызываться подпрограммы с разными алгоритмами планирования: *FIFO* и *RR*. Далее будем выводить их *pid* по мере завершения (по очереди). Для начала не будем нагружать программу вычислениями (чтобы не влияла величина кванта в *RR*). В итоге получим ожидаемую картину: процессы завершаются в таком же порядке, в котором были вызваны.

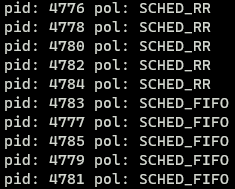
**

*Рисунок 7 – конкуренция политик без вычислительной нагрузки в процессах*

Изменим вызываемую программу (*son1.c*) таким образом: если у неё политика *RR* – вызовем *sleep(1)*. По идее, ввиду того что величина кванта будет превышена, сначала выполнятся все *FIFO* – подпрограммы, что и происходит на рисунке 7. Попробуем теперь вызвать *sleep(1)* в случае, если у подпрограммы политика *FIFO*.

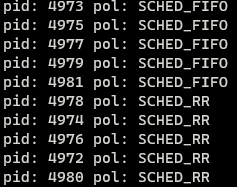
**

*Рисунок 8 – конкуренция политик с вычислительной нагрузкой в RR-процессах*

**

*Рисунок 9 – конкуренция политик с вычислительной нагрузкой в FIFO-процессах*

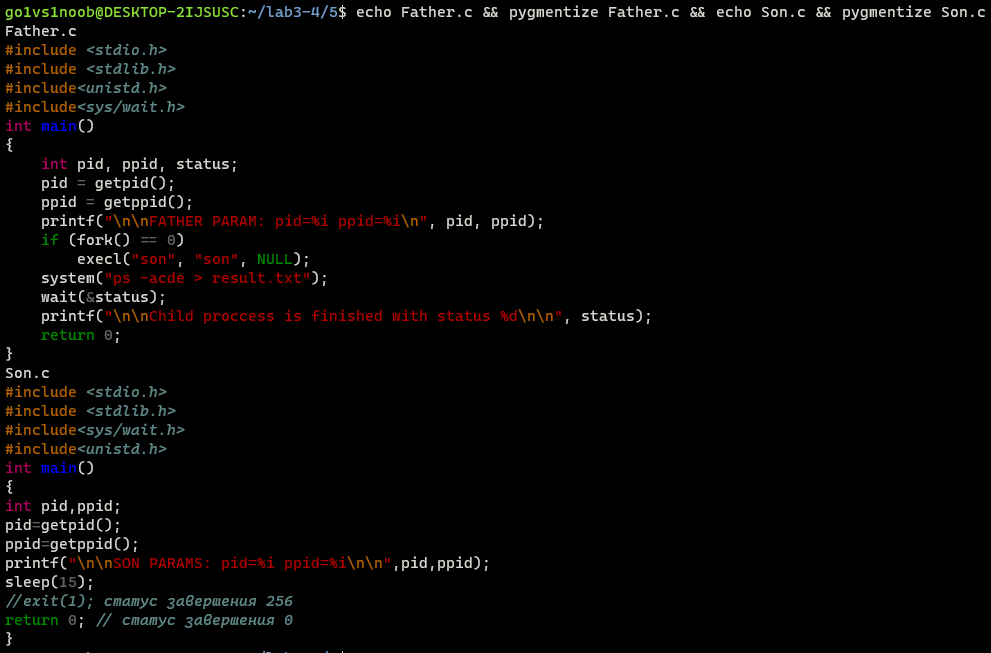
Как видно, хоть и у *FIFO* нет такого понятия, как величина кванта, если программы с политикой *FIFO* занимают больше времени, чем программы с политикой *RR* – планировщиком на выполнение в первую очередь будут избраны последние. Для полноты эксперимента попробуем «помешать» планировщику, добавим для *RR*-процессов задержку в 5 секунд (*sleep 5*), но при этом высокий приоритет (98), а *FIFO*-процессам – низкий приоритет (1). В результате планировщик приоритезировал более «лёгкий» процесс более «тяжёлому», но с высоким приоритетом, выставленным пользователем.

**

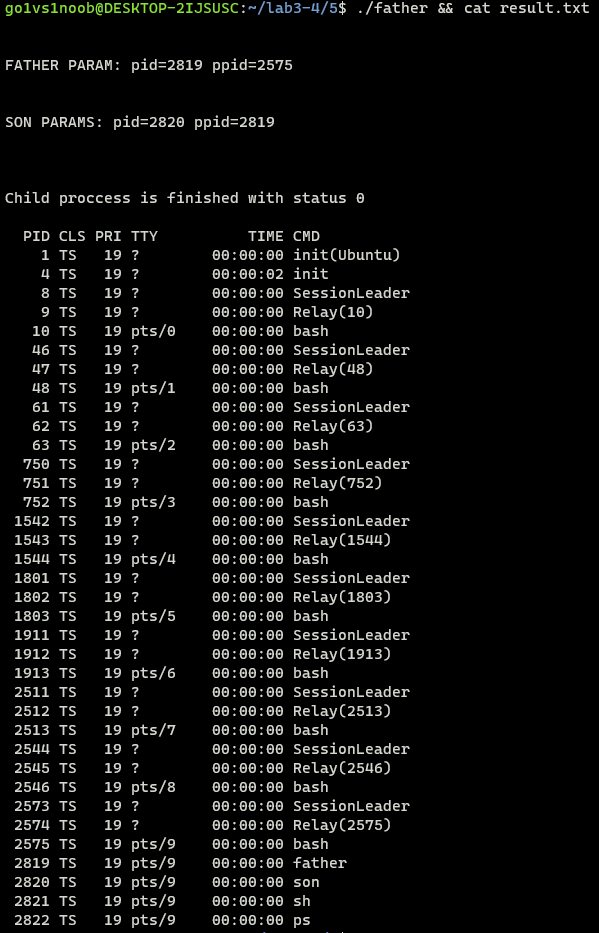
*Рисунок 10 – эксперимент, показывающий приоритезацию планировщиком время, требуемое для завершения программы приоритету, выставленному пользователем вручную*

1. Разработаем программу родителя и потомка с размещением в файлах

*Father.c* и *Son.c*. Для фиксации таблицы используем системный вызов *system (“ps -acde > result.txt”)*. Содержание программ представлены на рисунке 11. Родительский процесс с исходным кодом в файле  
*father.c* порождает процесс-потомок с помощью функции *fork()*. Затем,  
с помощью функции *execl("son","son",NULL);* запускается  
исполняемый файл *son*, выполнение начинается с точки входа –  
функции *main*. При этом фиксируются идентификаторы запущенных  
процессов, а также состояние таблицы процессов в файле *file.txt*.  
Родительский процесс дожидается выполнения потомка с помощью  
команды *wait(&status)*, а статус завершения этого процесса  
записывается по адресу *&status*. Запустим программу *Father*, получим информацию о процессах, запущенных с терминала (рисунок 12).



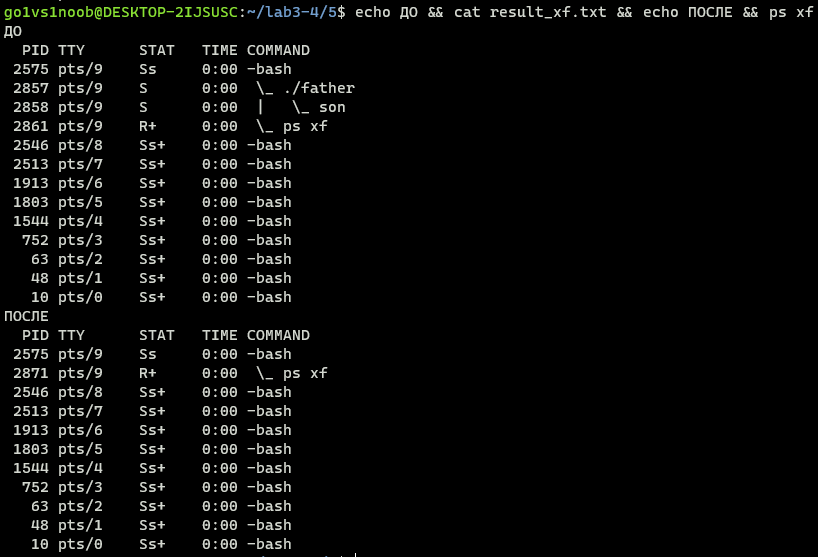
*Рисунок 11 – устройство программы-родителя и программы-потомка*



*Рисунок 12 – результат работы (вызова подпрограммы son родителем Father)*

В соответствии с выводимыми идентификаторами родителем  
процесса *son* является процесс *father*.

Выполним программу *Father* в фоновом режиме и получим таблицу процессов, запущенных с терминала с помощью *ps xf > result\_xf.txt* (рисунок 13)



*Рисунок 13 – вывод таблицы процессов до запуска Father и после*

Так, видно, что son был вызван с помощью *father*, также наглядно видно состояние списка процессов во время выполнения программы и после. *Bash* запускает *father*, распараллеливает процессы и порождает *son*.

Назначение полей:

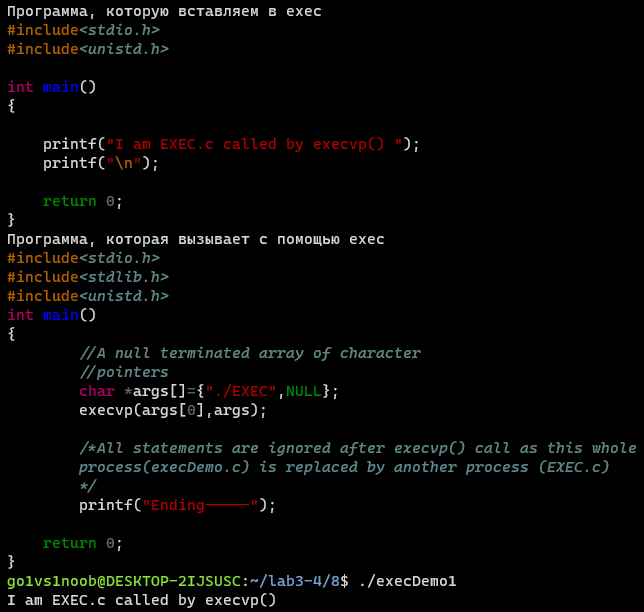
* PID – идентификатор процесса
* TTY – терминал, с которым связан данный процесс
* STAT – состояние, в котором на данный момент находится процесс (S – процесс ожидает, s – лидер сессии, R – процесс выполняется, + – порождённый процесс).
* TIME – процессорное время, занятое этим процессом
* COMMAND – команда, запустившая данный процесс

1. Выполним создание процессов с использованием различных функций

Семейства *exec()* с разными параметрами функций семейства, приведём результаты эксперимента. Для начала рассмотрим вариации команды *exec()*:

* *execl()* выполняет файл по заданному пути, аргументы – в виде списка переменных(arg0, …, NULL). Эту форму используют если количество аргументов известно
* *execlp()* подобно execl(), только файл ищется не только в текущем каталоге, но и в $PATH (переменная среды, которая содержит пути к директориям, где хранятся исполняемые файлы)
* *execle()* подобно execl(), при этом функция ожидает список переменных окружения в виде вектора
* *execv()* выполняет файл по заданному пути, аргументы – в форме вектора argv[]. Последний аргумент должен быть указателем NULL
* *execvp()* подобно execv(), только файл ищется не только в текущем каталоге, но и в $PATH (переменная среды, которая содержит пути к директориям, где хранятся исполняемые файлы)
* *execvpe* подобно execvp(), только функция также ожидает список переменных окружения в виде вектора

Рассмотрим пример применения *execvp()*. После вызова исполнительного файла *EXEC* строка *Ending* не выводится, так как после *exec()* в программе-родителе все команды игнорируются. Это видно на рисунке 14:



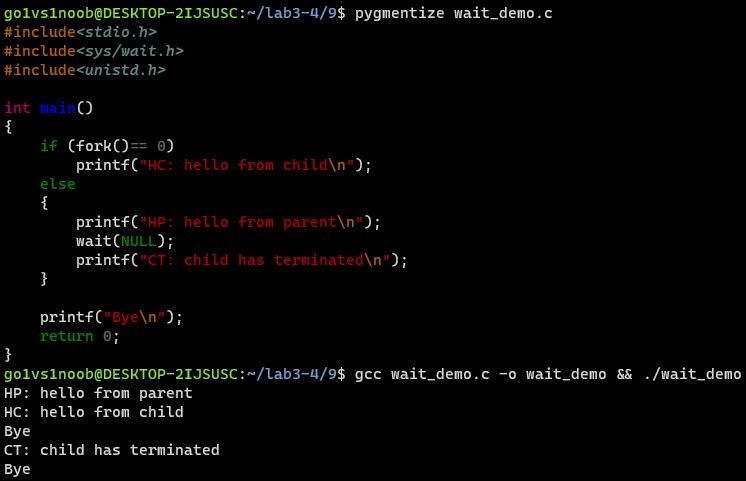
*Рисунок 14 – демонстрация работы программы с execvp*

При этом, очевидно, никакие переменные, не будут наследованы программой, вызванной из программы-родителя. После выполнения *exec()* машинный код, данные в куче и стеке будут заменены новой программой.

Подведём небольшой итог различий *exec()* и *fork()*: *fork()* «разбивает» текущую программу на две идентичные части (создавая новый процесс), по сути – программа становится двумя (или более) программами, исполняющими один и тот же код. При этом, дочерний процесс наследует данные процесса, который только что создал его, пусть и не на физическом уровне. С другой стороны, *exec()* запускает процесс с помощью исполнительного файла, полностью заменяя текущую программу той, что была подана в *exec()*.

1. Изучим утилиту *wait.* Вызов *wait()* приостанавливает вызов следующих

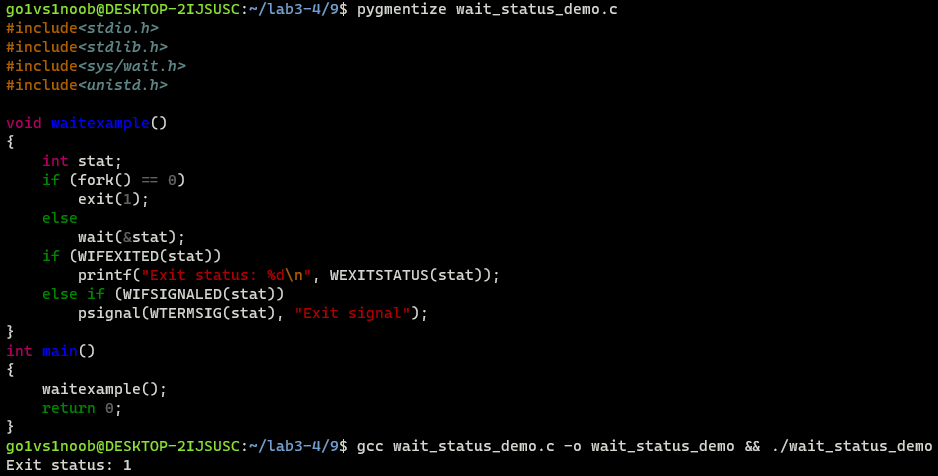
инструкций, пока один из дочерних процессов не завершит свою работу. На вход ей подаётся адрес переменной status, в которую будет возвращена информация о возвращении дочернего процесса. Сама *wait()* возвратит *PID* процесса, который был завершён. Если у процесса больше одного дочернего процесса, то *wait()* закончит свою работу, когда любой из дочерних закончит свою работу (будет возвращён *PID* первого дочернего элемента). Рассмотрим работу *wait()* на рисунке 15.



*Рисунок 15 – демонстрация работы программы с wait()*

Чтобы достать информацию о статусе, возвращённом из *wait()*, используются WIF-макро. Рассмотрим, что они значат и приведём пример программы, использующей их (рисунок 16).

* *WIFEXITED(status)* – дочерний процесс завершил свою работу нормально. *WEXITSTATUS(status)* вернёт код, который вернул дочерний процесс.
* *WIFSIGNALES(status)* – потомок завершился из-за необработанного сигнала. *WTERMSIG(status)* вернёт номер сигнала, который привёл к завершению потомка.
* *WIFSTOPPED(status)* – потомок остановлен. *WSTOPSIG(status)* вернёт номер сигнала остановки.

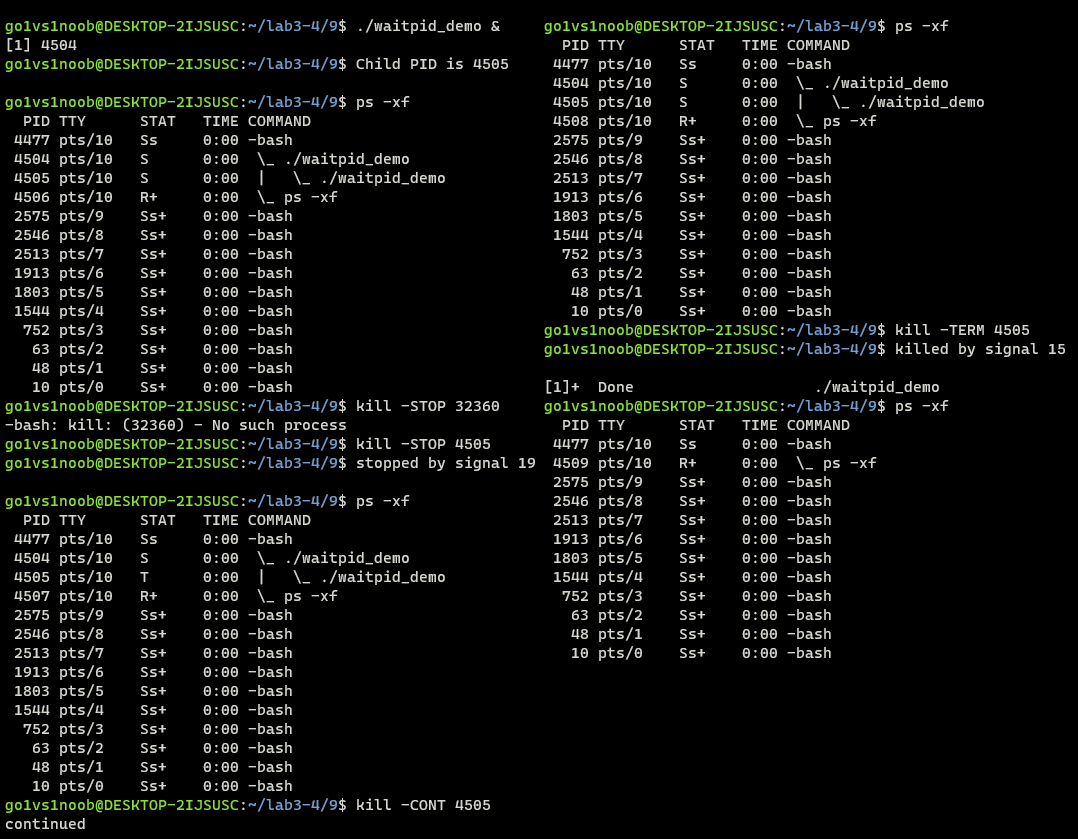


*Рисунок 16 – демонстрация программы с макро WIF*

Программа отработала согласно описанным выше макросам. Рассмотрим работу системного вызова *waitpid()*. Она работает как *wait()*, только ожидание кончается, если завершил свою работу процесс-потомок с поданным в вызов *PID*. Помимо этого, третьим аргументом туда можно передать опции (0 – без опций, родительский процесс должен дождаться завершения дочернего, *WNOHANG* – немедленное возвращение управления, если ни один из потомков не завершил выполнение). Рассмотрим программу на рисунке 17-18. Программа работает так: она создаёт дочерний процесс. Если не поданы аргументы командной строки – то дочерний процесс сразу завершает свою работу с сигналом, поданным как аргумент. Иначе – процесс приостанавливает свою работу с помощью *pause()* и позволяет пользователю посылать ему сигналы. Процесс – родитель мониторит дочерний процесс с помощью *waitpid()* и описанных выше макро.

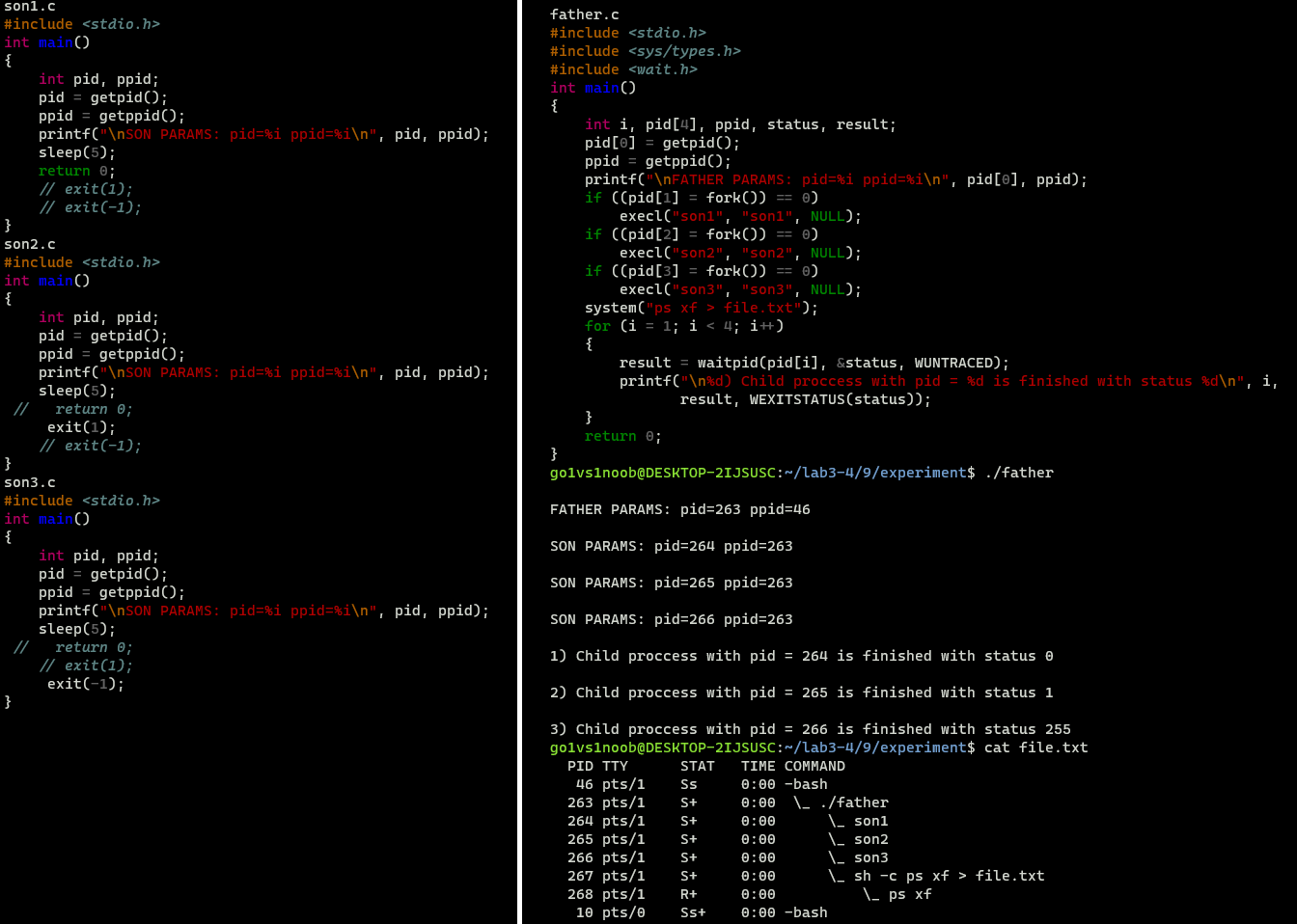


*Рисунок 17 – устройство программы с WIF-макро и управлением фоновым процессом с помощью -kill*

**

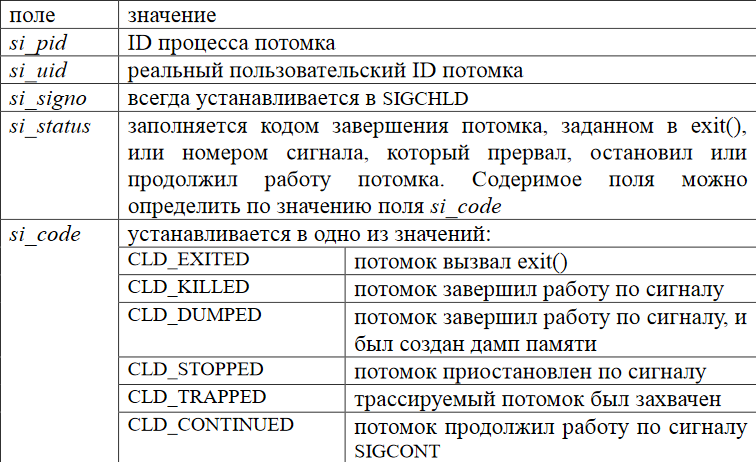
*Рисунок 18 – демонстрация работы программы, показанной на рисунке 17*

Предложим эксперимент, позволяющий родителю отслеживать подмножество порожденных потомков, используя *waitpid()*. Программа *father.c* порождает с помощью *fork()* и запускает *son1, son2, son3*. У *son1, son2, son3* возвращаются разные значения, которые мы записываем в массив *pid,* а после выводим на экран. Устройство файлов *father.c, son1.c, son2.c, son3.c*, а также результат работы эксперимента представлены на рисунке 19. Помимо этого представлен файл, в который запишется *ps -xf* (видно, как father породил *son1, son2, son3*).

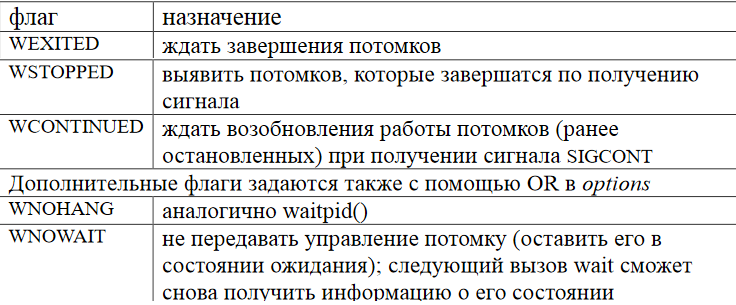


*Рисунок 19 – демонстрация работы программы, позволяющей отслеживать множество потомков процесса*

Системный вызов *waitid()*, по сути, является расширением *waitpid()*. *Waitid()* принимает на вход idtype (каких потомков отслеживать : P\_PID – только потомка, у которого PID = ID, G\_PID – потомков, у которых ID группы процесса совпадает с id, P\_ALL – любого потомка), id – PID потомка, infop – адрес структуры данных, которая будет заполняться информацией, представленной на рисунке 20, options – настройки, задающиеся флагами (перечислены на рисунке 21).



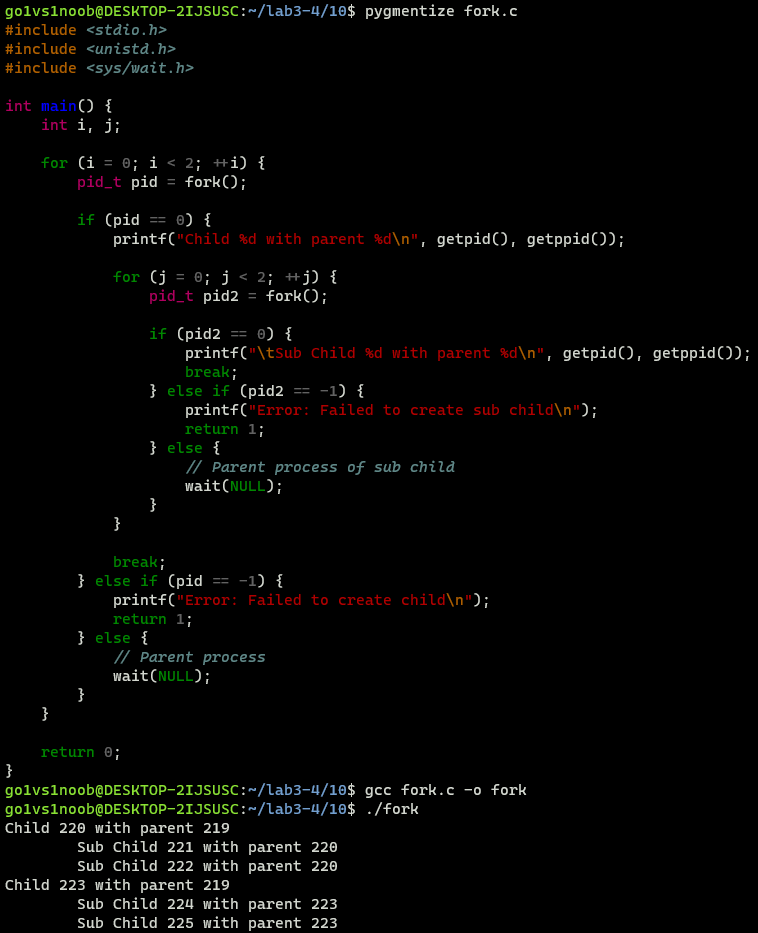
*Рисунок 20 – структура данных с информацией о потомке*



*Рисунок 21 – WIF-флаги и их значения*

1. Проанализируем очередность процессов, порождённых вложенными

вызовами *fork()*. Мы уже рассматривали структуру вложенных вызовов *fork()* – она древовидна. Процессы-потомки получают копию контекста родительского процесса и продолжает работу независимо друг от друга. В случае вложенных вызовов потомок также выполняет вложенный вызов и порождает собственных потомков. Процессы-потомки, порожденные одним процессом, но не являющиеся потомками друг друга могут выполняться параллельно. Очередность исполнения, при этом, зависит от работы планировщика процессов. Рассмотрим ещё одну программу с вложенными вызовами *fork()*. Этот пример создаёт для каждого дочернего процесса ещё два дочерних процесса, для каждого процесса вводится его *PID*, *PPID* (рисунок 22)

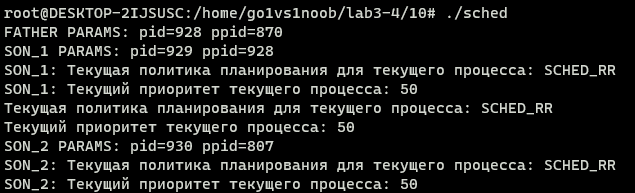


*Рисунок 22 – демонстрация программы с вложенными вызовами fork()*

Изменим процедуру планирования с помощью функции с шаблоном *scheduler* в ее названии. Рассмотрим устройство программы, результат её работы (рисунок 23-24). В *father.c* изменяется приоритет на 50, в результате чего приоритеты у дочерних процессов также изменяются (на 50), т.е. потомки наследуют политику планирование и приоритет родительского процесса.

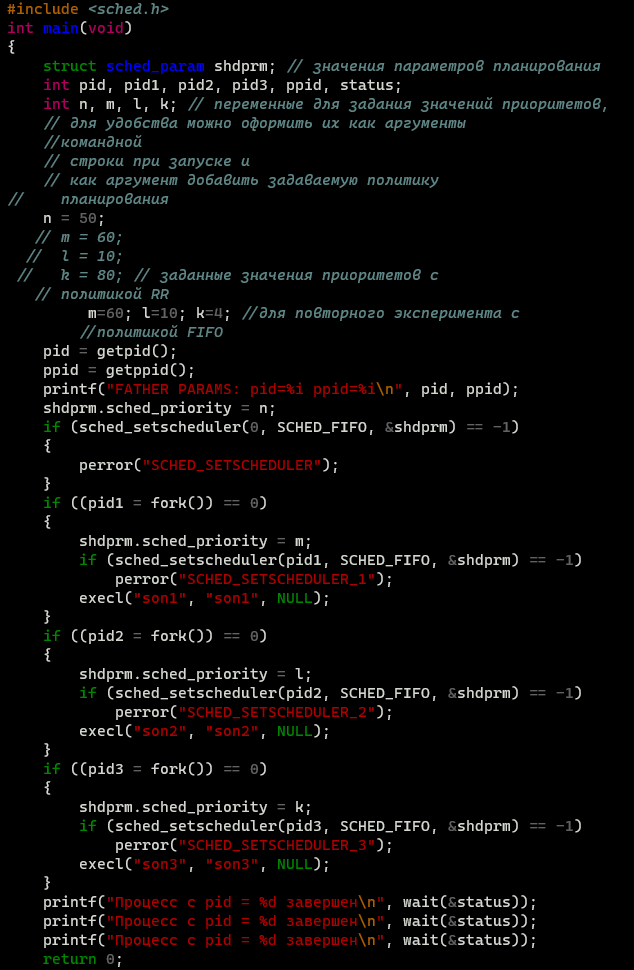


*Рисунок 23 – устройство программ, демонстрирующих наследование приоритетов*

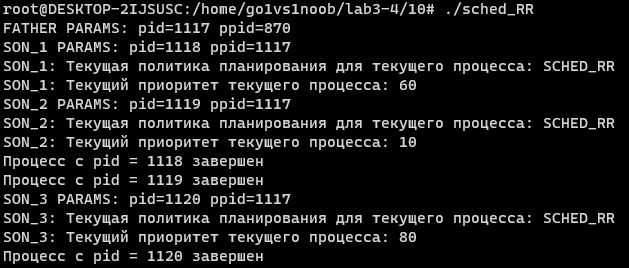


*Рисунок 24 – демонстрация работы программы, демонстрирующей наследование приоритетов*

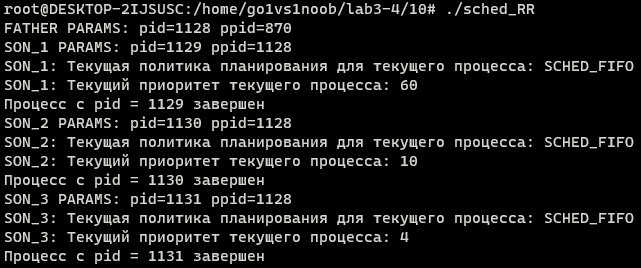
Теперь явным образом зададим политику планирования с помощью *sched\_setscheduler().* Устройство программы представим на рисунке 25. Сначала поставим *SCHED\_RR* (рисунок 26), потом – *SCHED\_FIFO* (рисунок 27). Как и говорили, потомки наследуют политику планирования.



*Рисунок 25 – устройство программ, демонстрирующих наследование политики планирования*

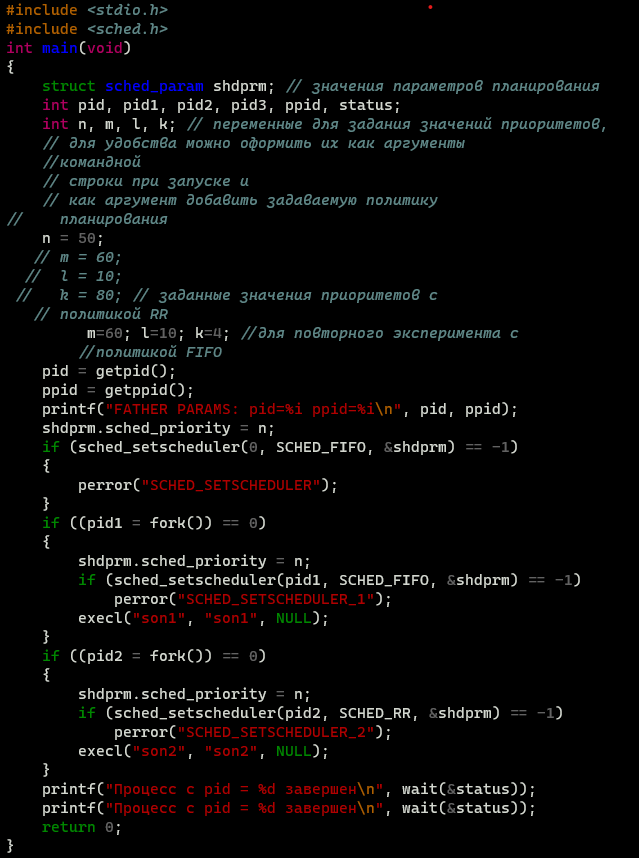


*Рисунок 26 – демонстрация наследования политики планирования RR*

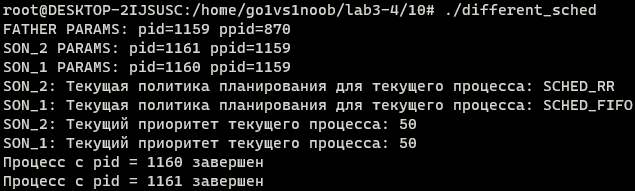


*Рисунок 27 – демонстрация наследования политики планирования FIFO*

Зададим разные процедуры планирования разным процессам с одинаковыми приоритетами n = 50 (устройство программы на рисунке 28). Рассмотрим, как они будут конкурировать (рисунок 29).

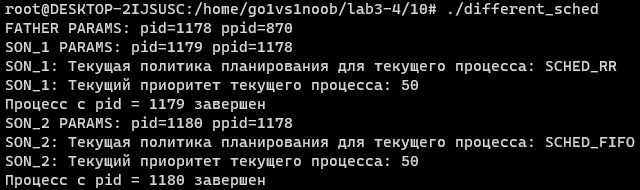


*Рисунок 28 – программа для демонстрации конкуренции двух процессов*



*Рисунок 29 – результат работы программы, показывающий конкуренцию двух процессов с одинаковыми приоритетами*

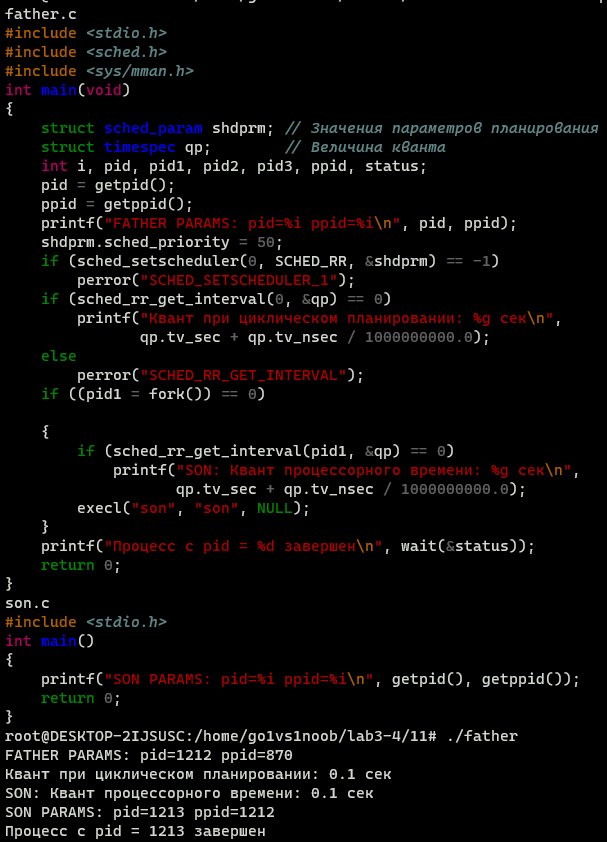
Поменяем порядок вызова *fork()* местами (т. е. сначала форкнем процесс с *sched\_RR*, а потом – *sched\_FIFO*). Получим ожидаемый результат – процесс, который был форкнут первым – завершился первым (рисунок 30).



*Рисунок 30 – результат работы программы, показывающий конкуренцию двух процессов с одинаковыми приоритетами*

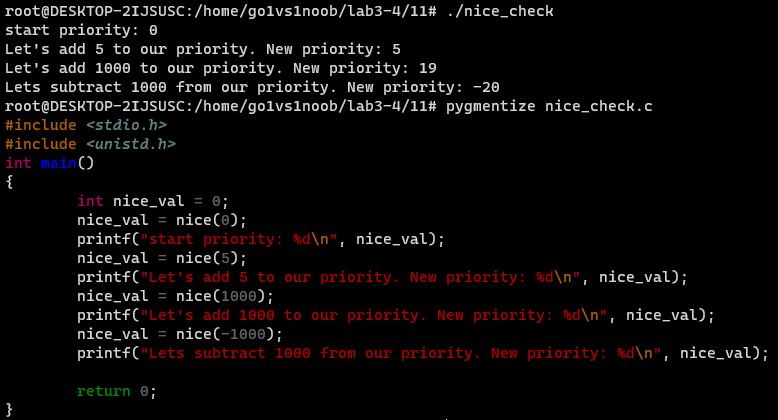
1. Процесс с алгоритмом *SCHED\_RR* работает не дольше времени,

называемого квантом. Если процесс работает дольше, чем квант, то он помещается в конец очереди процессов с тем же приоритетом. Попробуем определить величину кванта и выяснить, можно ли её менять. Определить величину кванта можно с помощью функции *sched\_rr\_get\_interval()*. Приведём пример её использования (рисунок 31).



*Рисунок 31 – устройство и демонстрация работы программы, позволяющей узнать величину кванта*

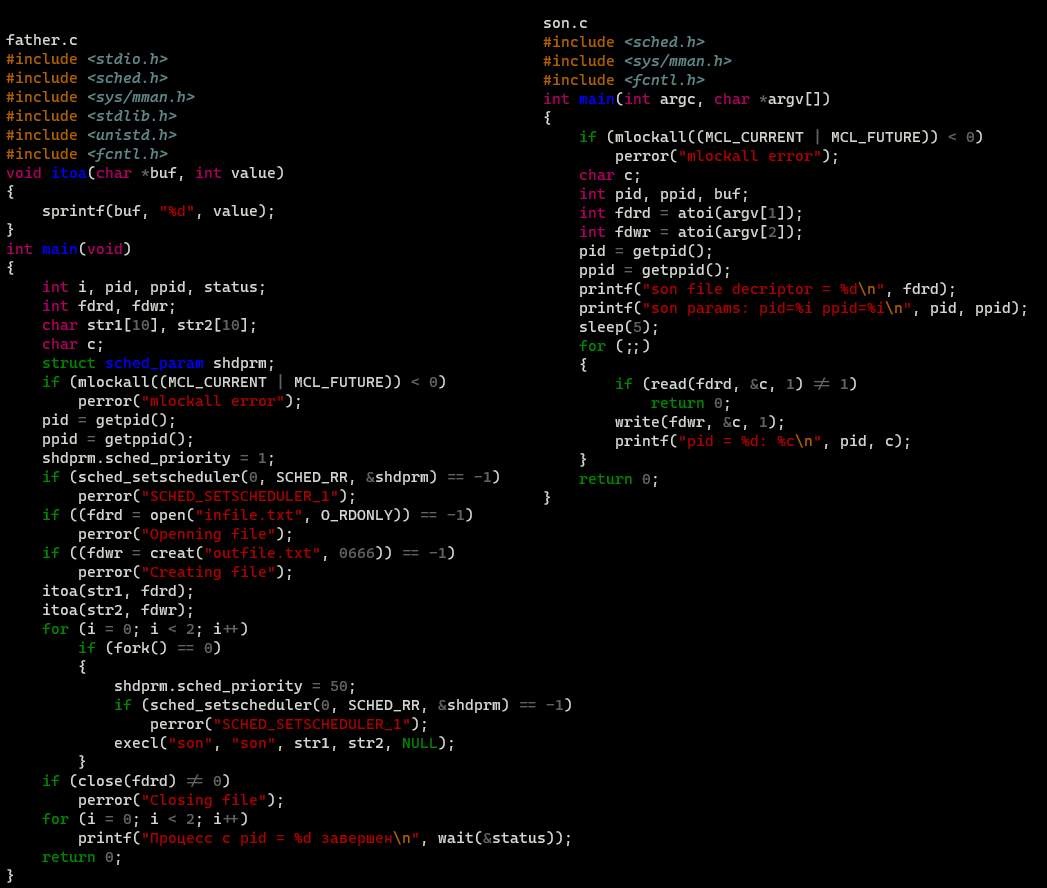
Таким образом, величина кванта – 0.1 секунды. В современной ОС Linux нельзя менять величину кванта. В старых можно было с помощью процесса *nice()*. Из мануала Linux можно узнать, что в старых версиях с помощью *nice()* можно было изменять приоритет процесса от -inf до 15. Сейчас же можно изменять от -20 (высокий приоритет) до +19 (низкий приоритет). Если попробуем выставить приоритет вне этих рамок – будет выставлен максимальный или минимальный соответственно. Проверим это экспериментально (рисунок 32):



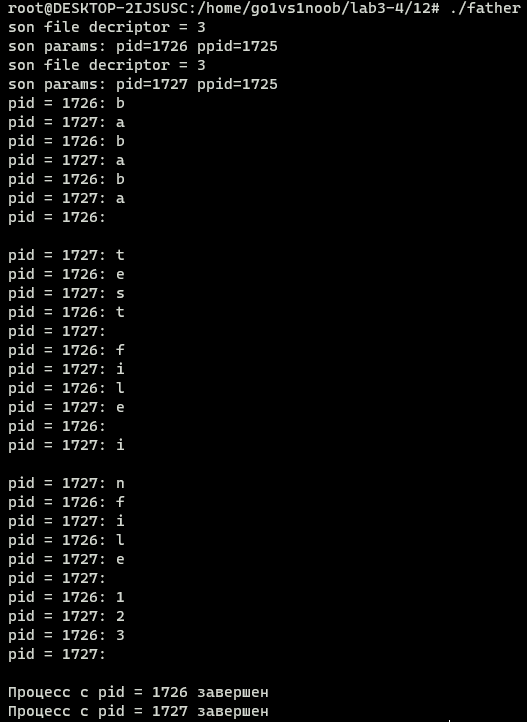
*Рисунок 32 – изменение приоритета с помощью nice()*

1. Проанализируем наследование на этапах *fork()* и *exec()*. Для

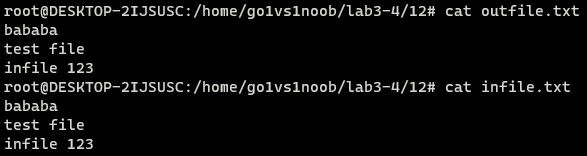
этого проведем эксперимент по проверке доступа потомков к файлам, открытым породившим их процессом. Рассмотрим пример кода, в котором в качестве аргументов процессам-потомкам передаются дескрипторы открытого и созданного родительским процессом файлов (в данном примере это *infile.txt* и *outfile.txt* соответственно). Порожденные процессы независимо друг от друга вызывают функции read и write, и в цикле считывают по одному байту информацию из исходного файла и переписывают ее в файл вывода. Устройство программ и результат работы на рисунках 33-35.



*Рисунок 33 – устройство программы, демонстрирующей наследование на этапах fork() и exec()*



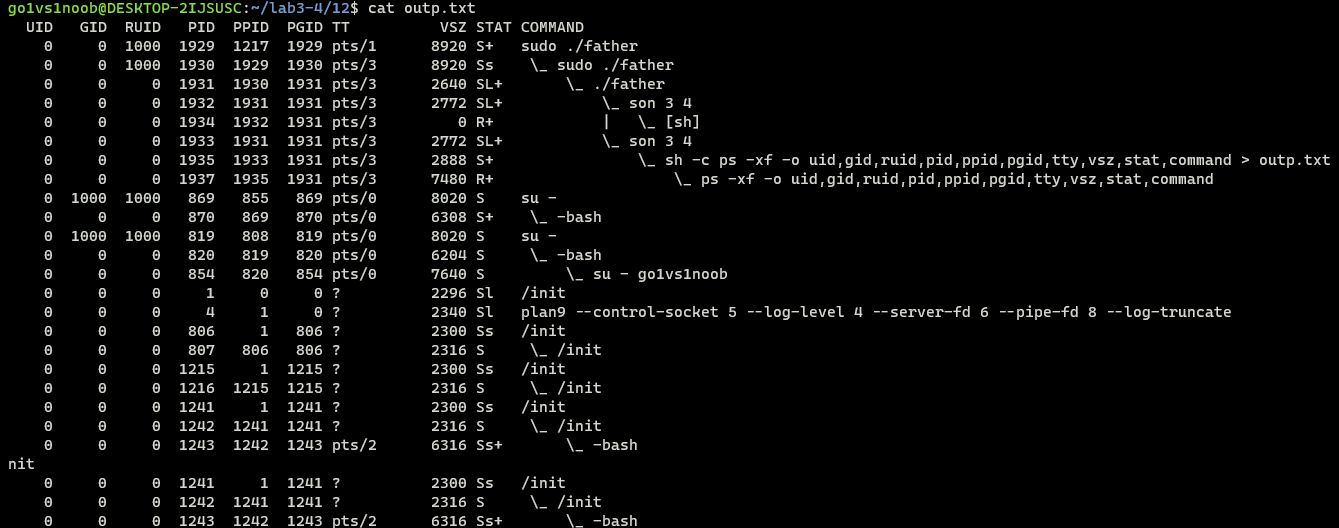
*Рисунок 34 – демонстрация работы программы, демонстрирующей наследование на этапах fork() и exec()*

**

*Рисунок 35 – результат, записанный в файл*

Дескрипторы *fdrd* в обоих потомках указывают на запись в таблице  
файлов, соответствующую исходному файлу, а дескрипторы,  
подставляемые в качестве *fdwr*, на запись, соответствующую файлу  
вывода. Ядро смещает внутрифайловые указатели после каждой  
операции чтения или записи, поэтому оба процесса никогда не  
обратятся вместе на чтение или запись по одному и тому же  
указателю или смещению внутри файла.  
Содержимое выходного файла зависит от очередности, в которой  
ядро ставит процессы на выполнение. Если очередность такая, что  
процессы исполняют системные функции попеременно (чередуя пары  
вызовов функций *read-write*), содержимое выходного файла будет  
совпадать с содержимым входного файла (что мы и видим на рисунке 34).

Наследование при выполнении функции *fork()* было рассмотрено в предыдуших пунктах. Можно подытожить следующим: процесс-потомок наследует от родителя: сегменты кода, данных и стека программы; таблицу файлов, в которой находятся состояния флагов дескрипторов файла, указывающие допустимые операции над файлом. Кроме того, в таблице файлов содержится текущая позиция указателя записи-чтения; рабочий и корневой каталоги; реальный и эффективный идентификатор пользователя и номер группы; приоритеты процесса (администратор может изменить их через *nice*); терминал; маску сигналов; ограничения по ресурсам; сведения о среде выполнения; разделяемые сегменты памяти. Потомок не наследует от родителя: идентификатора процесса (PID, PPID); израсходованного времени ЦП (оно обнуляется); сигналов процесса-родителя, требующих ответа; блокированных файлов (record locking).

В наследовании других параметров можно убедиться благодаря выводу программы *ps -xf -o uid,gid,ruid,pid,ppid,pgid,tty,vsz,stat,command* (т.е. от родителя наследуются *UID, GID, RUID, PGID, TTY*, приоритеты, политика планирования процессов (рисунок 36)). 

*Рисунок 36 – наследование прочих параметров программы*

**Вывод.**

В ходе выполнения лабораторной работы ознакомились с возможностью порождения и запуска процессов. Для этого использованы функции *fork()*, функции семейства *exec(), wait(), exit(), sleep()*. Созданы программы с псевдораспараллеливанием вычислений, программы, создающие процессы-потомки, изучены принципы их работы. Также написаны программы, запускающие другие программы с помощью *exec()*. Проанализированы функции семейства *wait()*. Проанализирована очередность исполнения процессов, определена величина кванта, продемонстрирована возможность её изменения.

**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)