

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ
по лабораторным работам №3, 4
по дисциплине «Операционные системы»
Тема: Управление процессами и потоками

Студентка гр. 1381

Рымарь М.И.

Преподаватель

Душутин Е.В.

Санкт-Петербург

2023

Цель работы.

Изучить основные принципы управления процессами и потоками в операционных системах.

Задание.

Порождение и запуск процессов

Используя системные функции `fork()`; семейства `exec...()`; `wait()`; `exit()`; `sleep()`; выполните следующее :

1. Создайте программу на основе одного исходного (а затем исполняемого) файла с псевдораспараллеливанием вычислений посредством порождения процесса-потомка.

2. Выполните сначала однократные вычисления в каждом процессе, обратите внимание, какой процесс на каком этапе владеет процессорным ресурсом. Каждый процесс должен иметь вывод на терминал, идентифицирующий текущий процесс. Последняя исполняемая команда функции `main` должна вывести на терминал сообщение о завершении программы. Объясните результаты. Сделайте выводы об использовании адресного пространства.

3. Затем однократные вычисления замените на циклы, длительность исполнения которых достаточна для наблюдения конкуренции процессов за процессорный ресурс.

4. Измените процедуру планирования и повторите эксперимент.

5. Разработайте программы родителя и потомка с размещением в файлах `father.c` и `son.c` Для фиксации состояния таблицы процессов в файле целесообразно использовать системный вызов `system("ps -abcde > file")`.

6. Запустите на выполнение программу `father.out`, получите информацию о процессах, запущенных с вашего терминала;

7. Выполните программу `father.out` в фоновом режиме `father &`. Получите таблицу процессов, запущенных с вашего терминала (включая отцовский и сыновний процессы).

8. Выполните создание процессов с использованием различных функций семейства `exec()` с разными параметрами функций семейства, приведите результаты эксперимента.

9. Проанализируйте значение, возвращаемое функцией `wait(&status)`. Предложите эксперимент, позволяющий родителю отслеживать подмножество порожденных потомков, используя различные функции семейства `wait()`.

10. Проанализируйте очередность исполнения процессов.

10.1. очередность исполнения процессов, порожденных вложенными вызовами `fork()`.

10.2. Измените процедуру планирования с помощью функции с шаблоном `scheduler` в ее названии и повторите эксперимент.

10.3. Поменяйте порядок очереди в RR-процедуре.

10.4. Можно ли задать разные процедуры планирования разным процессам с одинаковыми приоритетами. Как они будут конкурировать, подтвердите экспериментально.

11. Определите величину кванта. Можно ли ее поменять? — для обоснования проведите эксперимент.

12. Проанализируйте наследование на этапах `fork()` и `exec()`. Проведите эксперимент с родителем и потомками по доступу к одним и тем же файлам, открытым родителем. Аналогичные эксперименты проведите по отношению к другим параметрам.

Взаимодействие родственных процессов

13.1. Изменяя длительности выполнения процессов и параметры системных вызовов, рассмотрите 3 ситуации и получите соответствующие таблицы процессов:

а) процесс-отец запускает процесс-сын и ожидает его завершения;

б) процесс-отец запускает процесс-сын и, не ожидая его завершения, завершает свое выполнение. Зафиксируйте изменение родительского идентификатора процесса-сына;

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

13.2. Перенаправьте вывод не только на терминал, но и в файл. Организуйте программу многопроцессного функционирования так, чтобы результатом ее работы была демонстрация всех трех ситуаций с отображением в итоговом файле.

Управление процессами посредством сигналов

13.1. С помощью команды `kill -l` ознакомьтесь с перечнем сигналов, поддерживаемых процессами. Ознакомьтесь с системными вызовами `kill(2)`, `signal(2)`. Подготовьте программы следующего содержания:

а.) процесс `father` порождает процессы `son1`, `son2`, `son3` и запускает на исполнение программные коды из соответствующих исполнительных файлов;

б.) далее родительский процесс осуществляет управление потомками, для этого он генерирует сигнал каждому пользовательскому процессу;

в.) в пользовательских процессах-потомках необходимо обеспечить:

для `son1` - реакцию на сигнал по умолчанию;

для `son2` - реакцию игнорирования;

для `son3` - перехватывание и обработку сигнала.

Сформируйте файл-проект из четырех файлов, откомпилируйте, запустите программу. Проанализируйте таблицу процессов до и после отправки сигналов с помощью системного вызова `system("ps -s >> file")`. Обратите внимание на реакцию, устанавливаемую для последнего потомка.

13.2. Организуйте отсылку сигналов любым двум процессам, находящимся в разных состояниях: активном и пассивном, фиксируя моменты отсылки 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, SIGUSR1)`; (`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. Прodelайте 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).

25. Создайте командный файл (скрипт), выполняющий вашу лабораторную работу автоматически при наличии необходимых C-файлов.

Выполнение работы.

1. Была создана программа на основе исходного и исполняемого файла с псевдораспараллеливанием вычислений посредством порождения процесса-потомка. Исходный код программы *1.c* представлен ниже.

Исходный код программы *l.c*:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int pid;
    int n = 1;
    pid = fork();
    printf("Текущее число: %d\n", n);
    if (pid == -1)
    {
        perror("fork");
        exit(1);
    }
    if (pid == 0)
    {
        printf("child pid = %d, ppid = %d\n", getpid(), getppid());
        n *= 10;
        printf("Текущее число: %d\n", n);
    } else
    {
        printf("parent pid = %d, ppid = %d\n", getpid(), getppid());
        n += 13;
        printf("Текущее число: %d\n", n);
    }
    printf("Завершение процесса\n");
    exit(1);

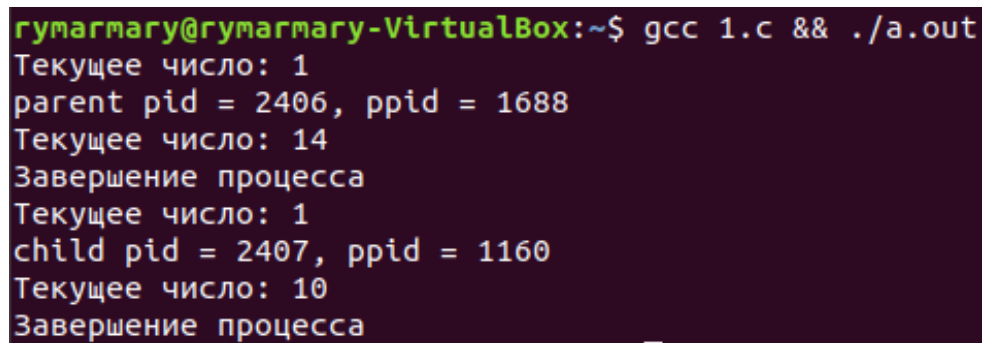
    return 0;
}
```

2. Сначала были выполнены однократные вычисления в каждом процессе. Каждый из них имеет свой вывод в терминал, самоидентифицирующий. Последняя команда, исполняемая основной функцией, выводит сообщение о завершении работы программы. Вывод в терминал представлен на рисунке 1.

Сначала был запущен порождающий процесс, который вывел результат своей работы. Затем был запущен процесс-потомок, результат его работы был выведен отдельно. Это позволяет сделать вывод, что при однократном запуске оба процесса работают последовательно, сначала выполняется порождающий процесс, а затем процесс-потомок. Следует отметить, что распараллеливание в

данном случае условное, так как оба процесса работают на одном процессоре или ядре, то есть в режиме деления времени при многозадачности.

По результатам выполнения программы будут выведены идентификаторы каждого процесса и его родителя, а также дважды фраза "Завершение процесса", что указывает на выполнение одного и того же кодового сегмента обоими процессами. Это показывает, что порожденный процесс полностью повторяет работу порождающего процесса, за исключением различий в идентификаторах процессов и их родителей.



```
gymarmary@gymarmary-VirtualBox:~$ gcc 1.c && ./a.out
Текущее число: 1
parent pid = 2406, ppid = 1688
Текущее число: 14
Завершение процесса
Текущее число: 1
child pid = 2407, ppid = 1160
Текущее число: 10
Завершение процесса
```

Рисунок 1 – Выполнение программы 1.c

3. Далее были заменены однократные вычисления циклами, длительность исполнения которых достаточна для наблюдения конкуренции процессов за процессорный ресурс. Вывод программы представлен на рисунке 2. Исходный код 3.c представлен ниже.

Исходный код 3.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    int pid;
    int n = 100;
    pid = fork();

    if (pid == -1) {
        perror("fork");
        exit(1);
    }

    while(1) {
        if (pid == 0) {
```



```

        printf("new pid = %d, ppid = %d\n", getpid(),getppid()
);
        /*здесь размещаются вычисления, выполняемые процессом-
потомком */
        printf("Текущее значение %d\n", n);
        ++n;
    } else {
        printf("parent    pid    =    %d,    ppid    =    %d\n",
getpid(),getppid() );
        /*здесь размещаются вычисления, выполняемые порождающим
процессом */
        printf("Текущее значение %d\n", n);
        --n;
    }
}
printf("Завершение процесса\n");
exit(1);

return 0;
}

```

```

parent pid = 2710, ppid = 1688
Текущее значение -6488
parent pid = 2710, ppid = 1688
Текущее значение -6489
parent pid = 2710, ppid = 1688
Текущее значение -6490
parent pid = 2710, ppid = 1688
Текущее значение -6491
parent pid = 2710, ppid = 1688
Текущее значение -6492
parent pid = 2710, ppid = 1688
Текущее значение -6493
parent pid = 2710, ppid = 1688
Текущее значение 6157
new pid = 2711, ppid = 2710
Текущее значение -6494
Текущее значение 6158
parent pid = 2710, ppid = 1688
new pid = 2711, ppid = 2710
Текущее значение 6159
Текущее значение -6495
new pid = 2711, ppid = 2710
parent pid = 2710, ppid = 1688
Текущее значение 6160
Текущее значение -6496
new pid = 2711, ppid = 2710

```

Рисунок 2 – Часть вывода программы 3.c

Здесь можно наблюдать, что если заменить однократные вычисления на цикл, то будет заметна конкуренция процессов за процессорный ресурс.

4. Изменили процедуру планирования и повторяем эксперимент. Исходный код программы 4.c приведён ниже. На рисунке 3 представлен вывод программы.

Исходный код программы 4.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
    int m,n,pid;
    m=5000;
    n=1;
    pid=fork();
    if (pid==-1) {
        perror("forkerror");
        exit(1);
    }
    printf("pid=%i\n",pid);
    if(pid!=0)
    {
        int j;
        for(j=1;j<=1000;j++)
        {
            m-=1;
        }
        printf("родитель:%i\n\n",m);
    }
    else
    {
        int i;
        for(i=1;i<=1000;i++)
        {
            n+=1;
        }
        printf("потомок:%d\n\n",n);
    }

    printf("Программа завершена\n");
    exit(1);
    return 0;
}
```

```

rymary@rymary-VirtualBox:~$ gcc 4.c && ./a.out
pid=3081
родитель:4000

Программа завершена
rymary@rymary-VirtualBox:~$ pid=0
потомок:1001

Программа завершена

```

Рисунок 3 – Вывод программы 4.c

Программа выполняет вызов *fork()*, выводит идентификатор выполняющегося процесса. В том случае, если текущий процесс является потомком (*pid=0*), то производится 1000 операций вычитания 1 из *n*. В противном случае, когда процесс является родителем, то происходит 1000 операций прибавления 1 к числу *m*. По завершении вычислений выводится результат и сообщение о завершении вычислений и работы программы.

5. Были разработаны две программы – родителя и потомка, которые были размещены соответственно в файлах *father.c* и *son.c*. Исходный код этих программ представлен ниже. Для фиксации состояния таблицы процессов в файле использовали системный вызов: *system("ps -xf > file.txt")*.

Исходный код программы *father.c*:

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main()
{
    int pid, ppid, status;
    pid=getpid();
    ppid=getppid();
    printf("FATHER PARAM: pid=%i ppid=%i\n", pid,ppid);
    if (fork()==0)
        execl("son","son", NULL);
    system("ps -xf > file.txt");
    wait(&status);
    printf("Child proccess is finished with status %d\n", status);

    return 0;
}

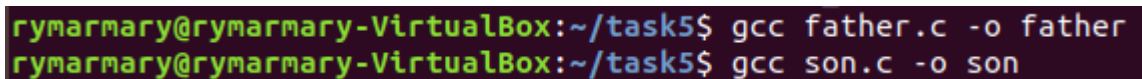
```

Исходный код программы *son.c*:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main()
{
    int pid,ppid;
    pid=getpid();
    ppid=getppid();
    printf("SON PARAMS:  pid=%i  ppid=%i\n",pid,ppid);
    sleep(5);
    //exit(1); //статус завершения 256
    return 0;  //статус завершения 0
}
```

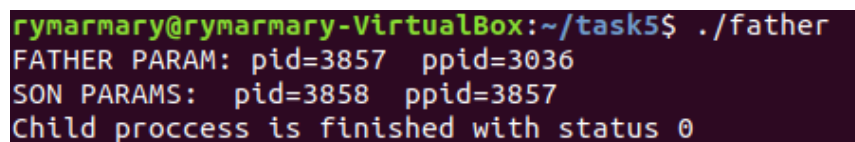
6. Скомпилируем программы из прошлого пункта, для удобства зададим такие же названия исполняемым файлам, как и исходникам. Компиляция показана на рисунке 4.



```
rymarmary@rymarmary-VirtualBox:~/task5$ gcc father.c -o father
rymarmary@rymarmary-VirtualBox:~/task5$ gcc son.c -o son
```

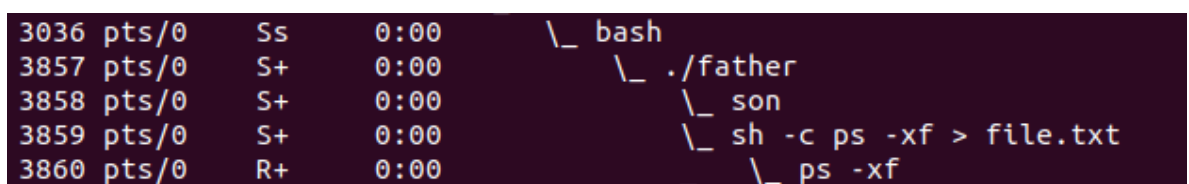
Рисунок 4 – Компиляция исходников

Далее командным интерпретатором (в нашем случае *bash*) запускаем программу *./father*, “распараллеливаем” процессы и порождаем *son*. Программа запускается в фоновом режиме, а параллельно ей – команда *ps -xf*. Важно отметить, что при однократном запуске от *father* запускается ещё один экземпляр интерпретатора, который в свою очередь запускает утилиту *ps*. Вывод программы показан на рисунке 5. Содержание получившегося файла *file.txt* с состояниями таблиц процессов показано на рисунке 6.



```
rymarmary@rymarmary-VirtualBox:~/task5$ ./father
FATHER PARAM: pid=3857  ppid=3036
SON PARAMS:  pid=3858  ppid=3857
Child process is finished with status 0
```

Рисунок 5 – Вывод программы



```
3036 pts/0    Ss      0:00      \_ bash
3857 pts/0    S+      0:00          \_ ./father
3858 pts/0    S+      0:00              \_ son
3859 pts/0    S+      0:00              \_ sh -c ps -xf > file.txt
3860 pts/0    R+      0:00                  \_ ps -xf
```

Рисунок 6 – Содержание *file.txt*

7. Исполняемый файл *father.out* был выполнен в фоновом режиме. Команда для выполнения и результат выполнения представлены на рисунке 7. Таблица процессов, запущенных с терминала (включая процессы родителя и потомка), получившихся в файле *file.txt* представлены на рисунке 8.

```

rymary@rymary-VirtualBox:~/task5$ ./father &
[1] 4100
rymary@rymary-VirtualBox:~/task5$ FATHER PARAM: pid=4100 ppid=3036
SON PARAMS: pid=4101 ppid=4100
Child process is finished with status 0
[1]+  Done                  ./father

```

Рисунок 7 – Выполнение программы в фоновом режиме

```

3036 pts/0    Ss+  0:00   \_ bash
4100 pts/0    S    0:00   \_ ./father
4101 pts/0    S    0:00       \_ son
4102 pts/0    S    0:00       \_ sh -c ps -xf > file.txt
4103 pts/0    R    0:00         \_ ps -xf

```

Рисунок 8 – Таблица процессов из *file.txt*

8. Было выполнено создание процессов с использованием различных функций семейства *exec()* с разными параметрами. Исходный код программы *8.c* представлен ниже. Результаты эксперимента приведены на рисунке 9.

Исходный код программы *8.c*:

```

#include <unistd.h>
#include <stdlib.h>

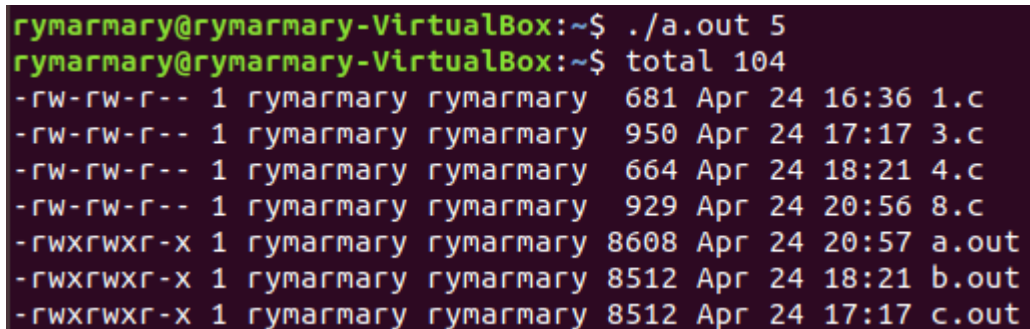
int main(int argc, char* argv[])
{
    char* file = "ls";
    char* path = "/bin/ls";
    char *args[] = {"ls", "-l", NULL };
    char *env[] = { (char*)NULL };
    int pid;
    pid = fork();
    if (pid == 0)
    {
        switch ( (int)argv[1][0] )
        {
            case (int)'1':
                execl("/bin/ls", "/bin/ls", "-l", (char *)NULL);
                break;
            case (int)'2':
                execlp("ls", "ls", "-l", (char *)NULL);
                break;

```

```

        case (int)'3':
            execle("/bin/ls", "ls", "-l", (char *)NULL, env);
            break;
        case (int)'4':
            execv("/bin/ls", args);
            break;
        case (int)'5':
            execvp("ls", args);
            break;
        case (int)'6':
            execvpe("ls", args, env);
            break;
    }
}
}

```



```

gymarmary@gymarmary-VirtualBox:~$ ./a.out 5
gymarmary@gymarmary-VirtualBox:~$ total 104
-rw-rw-r-- 1 gymarmary gymarmary 681 Apr 24 16:36 1.c
-rw-rw-r-- 1 gymarmary gymarmary 950 Apr 24 17:17 3.c
-rw-rw-r-- 1 gymarmary gymarmary 664 Apr 24 18:21 4.c
-rw-rw-r-- 1 gymarmary gymarmary 929 Apr 24 20:56 8.c
-rwxrwxr-x 1 gymarmary gymarmary 8608 Apr 24 20:57 a.out
-rwxrwxr-x 1 gymarmary gymarmary 8512 Apr 24 18:21 b.out
-rwxrwxr-x 1 gymarmary gymarmary 8512 Apr 24 17:17 c.out

```

Рисунок 9 – Пример работы программы 8.c

9. Описан системный вызов *wait(&status)*, который позволяет родительскому процессу ожидать завершения порожденного процесса и получить информацию о его статусе. При успешном завершении *fork*, процессы-родитель и порожденный работают параллельно, деля процессорное время и конкурируя за ресурсы на основе приоритетов. Выполнение порожденного процесса может быть приостановлено до завершения потомка системным вызовом *wait*. Если у родителя несколько потомков, необходимо выполнить несколько вызовов *wait*, чтобы узнать о завершении каждого из них. Если процесс не имеет потомков, вызов *wait* вернет код (-1).

Был предложен эксперимент, в котором использовались различные функции семейства *wait()* для отслеживания подмножества порожденных потомков. Ниже представлен исходный программный код для потомков *son1.c*, *son2.c*, *son3.c*, а также родителей *father1.c*, *father2.c*.

Исходный код программы *son1.c*:

```
#include <stdio.h>
```

```

#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    int pid,ppid;
    pid=getpid();
    ppid=getppid();
    printf("SONPARAMS: pid=%i ppid=%i\n",pid,ppid);
    sleep(2);
    return 0;
    // exit(1);
    // exit(-1);
}

```

Исходный код программы *son2.c*:

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    int pid,ppid;
    pid=getpid();
    ppid=getppid();
    printf("SONPARAMS: pid=%i ppid=%i\n",pid,ppid);
    sleep(2);
    // return 0;
    exit(1);
    // exit(-1);
}

```

Исходный код программы *son3.c*:

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    int pid,ppid;
    pid=getpid();
    ppid=getppid();
    printf("SONPARAMS: pid=%i ppid=%i\n",pid,ppid);
    sleep(2);
    // return 0;
    // exit(1);
    exit(-1);
}

```

Исходный код программы *father1.c*:

```

#include <stdio.h>
#include <sys/types.h>
#include <wait.h>

```

```

#include <stdlib.h>
#include <unistd.h>
int main()
{
    int i, pid[4], ppid, status, result;
    pid[0]=getpid();
    ppid=getppid();
    printf("FATHER PARAMS: pid=%i ppid=%i\n", pid[0],ppid);
    if((pid[1] = fork()) == 0)
        execl("son1", "son1", NULL);
    if((pid[2] = fork()) == 0)
        execl("son2", "son2", NULL);
    if((pid[3] = fork()) == 0)
        execl("son3", "son3", NULL);
    system("ps xf > file.txt");
    for (i = 1; i < 4; i++)
    {
        result = waitpid(pid[i], &status, WUNTRACED);
        printf("%d) Child proccess with pid = %d is finished with
status %d\n", i, result, status);
    }
    return 0;
}

```

Исходный код программы *father2.c*:

```

#include <stdio.h>
#include <sys/types.h>
#include <wait.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
    int i, ppid, pid[4], status[3], result[3];
    char *son[] = {"son1", "son2", "son3"};
    int option[] = {WNOHANG, WUNTRACED, WNOHANG}; // здесь можно
задавать различные флаги исполнения
    pid[4] = getpid();
    ppid = getppid();
    printf("FATHER PARAMS: pid=%i ppid=%i\n", pid[3],ppid);
    for (i = 0; i < 3; i++)
        if((pid[i] = fork()) == 0)
            execl(son[i], son[i], NULL);
    system("ps xf > file.txt");
    for (i = 0; i < 3; i++)
    {
        result[i] = waitpid(pid[i], &status[i], option[i]);
        printf("%d) Child with pid = %d is finished with status
%d\n", (1 + i), result[i], status[i]);
    }
    for(i = 0; i < 3; i++)
        if (WIFEXITED(status[i]) == 0)
            printf("Proccess pid = %d was failed.\n",pid[i]);
}

```



```

else
    printf("Process pid = %d was success.\n", pid[i]);
return 0;
}

```

Скомпилировав все программы и получив исполняемые файлы, запустим родительские исполняемые файлы. Вывод в терминал для father1 представлен на рисунке 10. Таблица процессов для него – на рисунке 11. Вывод в терминал для father2 представлен на рисунке 12. Таблица процессов для него – на рисунке 13.

```

rymary@rymary-VirtualBox:~/task9$ ./father1
FATHER PARAMS: pid=1938 ppid=1858
SONPARAMS: pid=1941 ppid=1938
SONPARAMS: pid=1940 ppid=1938
SONPARAMS: pid=1939 ppid=1938
1) Child process with pid = 1939 is finished with status 0
2) Child process with pid = 1940 is finished with status 256
3) Child process with pid = 1941 is finished with status 65280

```

Рисунок 10 – Выполнение father1

```

1938 pts/0    S+      0:00    \_ ./father1
1939 pts/0    S+      0:00    \_ son1
1940 pts/0    S+      0:00    \_ son2
1941 pts/0    S+      0:00    \_ son3
1942 pts/0    S+      0:00    \_ sh -c ps xf > file.txt
1943 pts/0    R+      0:00    \_ ps xf

```

Рисунок 11 – Таблица процессов для father1

```

rymary@rymary-VirtualBox:~/task9$ ./father2
FATHER PARAMS: pid=1945 ppid=1858
SONPARAMS: pid=1948 ppid=1945
SONPARAMS: pid=1946 ppid=1945
SONPARAMS: pid=1947 ppid=1945
1) Child with pid = 0 is finished with status 32623
2) Child with pid = 1947 is finished with status 256
3) Child with pid = 1948 is finished with status 65280
Process pid = 1946 was failed.
Process pid = 1947 was success.
Process pid = 1948 was success.

```

Рисунок 12 – Выполнение father2

```

1945 pts/0    S+      0:00    \_ ./father2
1946 pts/0    R+      0:00    \_ son1
1947 pts/0    R+      0:00    \_ son2
1948 pts/0    S+      0:00    \_ son3
1949 pts/0    S+      0:00    \_ sh -c ps xf > file.txt
1950 pts/0    R+      0:00    \_ ps xf

```

Рисунок 13 – Таблица процессов для father2

Комбинируя разные флаги, мы получаем каждый раз разный результат для `father2`.

`WNOHANG` – означает немедленное возвращение управления, если ни один дочерний процесс не завершил выполнение.

`WUNTRACED` – означает возврат управления и для остановленных (но не отслеживаемых) дочерних процессов, о статусе которых еще не было сообщено. Статус для отслеживаемых остановленных подпроцессов также обеспечивается без этой опции.

Именно поэтому, в выводе мы наблюдаем информацию о каждом сыне, хоть и код возврата у всех разный. Примеры выводов для различных комбинаций флагов представлены на рисунках 14, 15.

```
gymarmary@gymarmary-VirtualBox:~/task9$ ./father2
FATHER PARAMS: pid=1984 ppid=1858
SONPARAMS: pid=1987 ppid=1984
SONPARAMS: pid=1985 ppid=1984
SONPARAMS: pid=1986 ppid=1984
1) Child with pid = 0 is finished with status 32636
2) Child with pid = 0 is finished with status 827252360
3) Child with pid = 0 is finished with status 32764
Process pid = 1985 was failed.
Process pid = 1986 was failed.
Process pid = 1987 was failed.
```

Рисунок 14 – Вывод с флагами `WNOHANG`, `WNOHANG`, `WNOHANG`

```
gymarmary@gymarmary-VirtualBox:~/task9$ ./father2
FATHER PARAMS: pid=1998 ppid=1858
SONPARAMS: pid=2001 ppid=1998
SONPARAMS: pid=1999 ppid=1998
SONPARAMS: pid=2000 ppid=1998
1) Child with pid = 0 is finished with status 32739
2) Child with pid = 0 is finished with status 1530091944
3) Child with pid = 2001 is finished with status 65280
Process pid = 1999 was failed.
Process pid = 2000 was failed.
Process pid = 2001 was success.
```

Рисунок 15 – Вывод с флагами `WNOHANG`, `WNOHANG`, `WUNTRACED`

10. С помощью системного вызова `fork()` можно создать дочерний процесс, который будет выполняться одновременно с родительским процессом. Однако порядок выполнения процессов, порожденных вызовами `fork()`, недетерминирован и может зависеть от различных факторов, что может привести

к сложному и непредсказуемому поведению. Родительский и дочерний процессы могут выполняться одновременно и их выполнение может перекрываться, что может привести к неожиданным результатам.

10.1. Когда процесс вызывает функцию *fork()*, создается новый процесс-потомок, который является копией родительского процесса, включая его адресное пространство и контекст выполнения. Родительский процесс продолжает выполнение кода, предшествующего вызову *fork()*, в то время как дочерний процесс начинает выполнять код, следующий за вызовом *fork()*. Если дочерний процесс вызывает *fork()* снова, создается еще один дочерний процесс, и так далее, создавая древовидную структуру процессов. Порядок выполнения процессов, созданных вложенными вызовами *fork()*, недетерминирован и зависит от различных факторов.

10.2. Изменим приоритеты и текущую политику планирования. Приведём пример такой программы. Исходный код *father.c*, *son1.c*, *son2.c* приведён ниже.

Исходный код программы *father.c*:

```
#include <stdio.h>
#include <sched.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main () {
    struct sched_param shdprm; // Значения параметров планирования
    int pid, pid1, pid2, ppid;
    pid = getpid();
    ppid = getppid();
    printf("FATHER PARAMS: pid=%i ppid=%i\n", pid, ppid);
    shdprm.sched_priority = 50;
    if (sched_setscheduler(0, SCHED_RR, &shdprm) == -1)
    {
        perror ("SCHED_SETSCHEDULER");
    }
    if((pid1=fork()) == 0)
        execl("son1", "son1", NULL);
    if((pid2=fork()) == 0)
        execl("son2", "son2", NULL);
    printf ("Текущая политика планирования для текущего процесса:");
    switch (sched_getscheduler(0)) {
        case SCHED_FIFO:
```

```

        printf ("SCHED_FIFO\n");
        break;
    case SCHED_RR:
        printf ("SCHED_RR\n");
        break;
    case SCHED_OTHER:
        printf ("SCHED_OTHER\n");
        break;
    case -1:
        perror ("SCHED_GETSCHEDULER");
        break;
    default:
        printf ("Неизвестная политика планирования\n");
    }
    if (sched_getparam(0, &shdprm) == 0)
        printf ("Текущий приоритет текущего процесса: %d\n",
shdprm.sched_priority);
    else
        perror ("SCHED_GETPARAM");
    return 0;
}

```

Исходный код программы *son1.c*:

```

#include <stdio.h>
#include <stdlib.h>
#include <sched.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    struct sched_param shdprm; // Значения параметров планирования
    int i, pid, ppid;
    pid=getpid(); ppid=getppid();
    printf("SON_1 PARAMS: pid=%i ppid=%i\n", pid, ppid);
    printf ("SON_1: Текущая политика планирования для текущего
процесса: ");
    switch (sched_getscheduler(0)) {
        case SCHED_FIFO:
            printf ("SCHED_FIFO\n");
            break;
        case SCHED_RR:
            printf ("SCHED_RR\n");
            break;
        case SCHED_OTHER:
            printf ("SCHED_OTHER\n");
            break;
        case -1:
            perror ("SCHED_GETSCHEDULER");
            break;
        default:
            printf ("Неизвестная политика планирования\n");
    }
}

```

```

        if (sched_getparam(0, &shdprm) == 0)
            printf ("SON_1: Текущий приоритет текущего процесса: %d\n",
shdprm.sched_priority);
        else
            perror ("SCHED_GETPARAM");

    return 0;
}

```

Исходный код программы *son2.c*:

```

#include <stdio.h>
#include <stdlib.h>
#include <sched.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    struct sched_param shdprm; // Значения параметров планирования
    int i, pid, ppid;
    pid=getpid(); ppid=getppid();
    printf("SON_2 PARAMS: pid=%i ppid=%i\n", pid, ppid);
    printf("SON_2: Текущая политика планирования для текущего
процесса: ");
    switch (sched_getscheduler(0)) {
        case SCHED_FIFO:
            printf ("SCHED_FIFO\n");
            break;
        case SCHED_RR:
            printf ("SCHED_RR\n");
            break;
        case SCHED_OTHER:
            printf ("SCHED_OTHER\n");
            break;
        case -1:
            perror ("SCHED_GETSCHEDULER");
            break;
        default:
            printf ("Неизвестная политика планирования\n");
    }
    if (sched_getparam(0, &shdprm) == 0)
        printf ("SON_2: Текущий приоритет текущего процесса: %d\n",
shdprm.sched_priority);
    else
        perror ("SCHED_GETPARAM");

    return 0;
}

```

Скомпилировав и запустив исполняемый файл родителя, получим два разных вывода в зависимости от режимов. На рисунке 16 показан вывод в обычном режиме, на рисунке 17 – в режиме суперпользователя (от имени root).

Посмотрев на результаты, можно сделать вывод о том, что потомки наследуют политику планирования и приоритет родительского процесса.

```
rymarmary@rymarmary-VirtualBox:~/task10_2$ ./father
FATHER PARAMS: pid=2327 ppid=2140
SCHED_SETSCHEDULER: Operation not permitted
Текущая политика планирования для текущего процесса: SCHED_OTHER
Текущий приоритет текущего процесса: 0
rymarmary@rymarmary-VirtualBox:~/task10_2$ SON_2 PARAMS: pid=2329 ppid=1185
SON_2: Текущая политика планирования для текущего процесса: SCHED_OTHER
SON_2: Текущий приоритет текущего процесса: 0
SON_1 PARAMS: pid=2328 ppid=1185
SON_1: Текущая политика планирования для текущего процесса: SCHED_OTHER
SON_1: Текущий приоритет текущего процесса: 0
```

Рисунок 16 – Вывод в обычном режиме

```
rymarmary@rymarmary-VirtualBox:~/task10_2$ sudo -su root
[sudo] password for rymarmary:
root@rymarmary-VirtualBox:~/task10_2# ./father
FATHER PARAMS: pid=2363 ppid=2352
Текущая политика планирования для текущего процесса: SCHED_RR
Текущий приоритет текущего процесса: 50
root@rymarmary-VirtualBox:~/task10_2# SON_2 PARAMS: pid=2365 ppid=1185
SON_2: Текущая политика планирования для текущего процесса: SCHED_RR
SON_2: Текущий приоритет текущего процесса: 50
SON_1 PARAMS: pid=2364 ppid=1185
SON_1: Текущая политика планирования для текущего процесса: SCHED_RR
SON_1: Текущий приоритет текущего процесса: 50
```

Рисунок 17 – Вывод в режиме суперпользователя

10.3. Ниже представлен изменённый программный код для *father.c* в соответствии с заданием. На рисунках 18 и 19 представлены выводы при запусках программы от имени обычного пользователя и от суперпользователя, соответственно.

Исходный код программы *father.c*:

```
#include <stdio.h>
#include <sched.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main ()
{
    struct sched_param shdprm; // значения параметров планирования
    int pid, pid1, pid2, pid3, ppid, status;
    int n, m, l, k; // переменные для задания значений приоритетов,
```

```

// для
удобства можно оформить их как аргументы командной
//
строки при запуске и
// как
аргумент добавить задаваемую политику планирования
n=50; m=60; l=10; k=80; // заданные значения приоритетов
сполитикой RR//
//m=60; l=10; k=4; //для повторного эксперимента с
политикой FIFO
pid = getpid();
ppid = getppid();
printf("FATHER PARAMS: pid=%i ppid=%i\n", pid, ppid);
shdprm.sched_priority = n;
if (sched_setscheduler (0, SCHED_RR, &shdprm) == -1)
{
    perror ("SCHED_SETSCHEDULER");
}
if ((pid1=fork()) == 0)
{
    shdprm.sched_priority = m;
    if (sched_setscheduler (pid1, SCHED_RR, &shdprm) == -1)
        perror ("SCHED_SETSCHEDULER_1");
    execl("son1", "son1", NULL);
}
if ((pid2=fork()) == 0)
{
    shdprm.sched_priority = l;
    if (sched_setscheduler (pid2, SCHED_RR, &shdprm) == -1)
        perror ("SCHED_SETSCHEDULER_2");
    execl("son2", "son2", NULL);
}
if ((pid3=fork()) == 0)
{
    shdprm.sched_priority = k;
    if (sched_setscheduler (pid3, SCHED_RR, &shdprm) == -1)
        perror ("SCHED_SETSCHEDULER_3");
    execl("son3", "son3", NULL);
}
printf("Процесс с pid = %d завершен\n", wait(&status));
printf("Процесс с pid = %d завершен\n", wait(&status));
printf("Процесс с pid = %d завершен\n", wait(&status));
return 0;
}

```

```

rymarmary@rymarmary-VirtualBox:~/task10_3$ ./father
FATHER PARAMS: pid=3073 ppid=2140
SCHED_SETSCHEDULER: Operation not permitted
SCHED_SETSCHEDULER_3: Operation not permitted
Процесс с pid = -1 завершен
Процесс с pid = -1 завершен
Процесс с pid = -1 завершен
Процесс с pid = 3076 завершен
SCHED_SETSCHEDULER_2: Operation not permitted
SCHED_SETSCHEDULER_1: Operation not permitted
SON_2 PARAMS: pid=3075 ppid=3073
SON_2: Текущая политика планирования для текущего процесса: SCHED_OTHER
SON_2: Текущий приоритет текущего процесса: 0
Процесс с pid = 3075 завершен
SON_1 PARAMS: pid=3074 ppid=3073
SON_1: Текущая политика планирования для текущего процесса: SCHED_OTHER
SON_1: Текущий приоритет текущего процесса: 0
Процесс с pid = 3074 завершен

```

Рисунок 18 – Вывод при запуске программы в обычном режиме

```

root@rymarmary-VirtualBox:~/task10_3# ./father
FATHER PARAMS: pid=3091 ppid=3083
Процесс с pid = -1 завершен
Процесс с pid = -1 завершен
Процесс с pid = -1 завершен
SON_1 PARAMS: pid=3092 ppid=3091
SON_1: Текущая политика планирования для текущего процесса: SCHED_RR
SON_1: Текущий приоритет текущего процесса: 60
Процесс с pid = 3092 завершен
Процесс с pid = 3094 завершен
SON_2 PARAMS: pid=3093 ppid=3091
SON_2: Текущая политика планирования для текущего процесса: SCHED_RR
SON_2: Текущий приоритет текущего процесса: 10
Процесс с pid = 3093 завершен

```

Рисунок 19 – Вывод при запуске программы в режиме суперпользователя

В алгоритме планирования RR каждому процессу выделяется фиксированный интервал времени для выполнения, после чего процесс вытесняется и запускается следующий процесс в очереди. Порядок, в котором процессы добавляются в очередь, определяет порядок их выполнения.

Изменение порядка очереди в алгоритме RR позволяет изменять порядок выполнения процессов. Например, можно установить приоритеты процессов с более высоким значением приоритета, разместив их в начале очереди. Можно также дать всем процессам одинаковый приоритет, используя очередь FIFO. Далее приведу пример эксперимента с множеством процессов-сыновей. Ниже

добавлен исходный код программ родителя *father.c* и потомков *son1.c-son6.c*.

Программный вывод показан на рисунке 20.

Исходный код программы *father.c*:

```
#include <stdio.h>
#include <sched.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main ()
{
    struct sched_param shdprm; // значения параметров планирования
    int pid,ppid, status;
    int n = 60; // переменные для задания значений приоритетов,

    int prior = 10;          //для повторного эксперимента с политикой
FIFO
    char *arr[6] = {"son1", "son2", "son3", "son4", "son5", "son6"};
    int pid_son[6];
    pid = getpid();
    ppid = getppid();
    printf("FATHER PARAMS: pid=%i  ppid=%i\n", pid,ppid);
    shdprm.sched_priority = n;
    if (sched_setscheduler (0, SCHED_FIFO, &shdprm) == -1)
    {
        perror ("SCHED_FIFO");
    }

    for (int i=0; i < 6; i++){
        if((pid_son[i]=fork()) == 0)
        {
            shdprm.sched_priority = prior;
            if (sched_setscheduler (pid_son[i], SCHED_FIFO,
&shdprm) == -1)
                perror ("SCHED_FIFO");
            execl(arr[i], arr[i], NULL);
        }
    }
    for (int i=0; i < 6; i++){
        printf("Процесс с pid = %d завершен\n", wait(&status));
    }
    return 0;
}
```

Исходный код программ *son1.c-son6.c*:

```
#include <stdio.h>
#include <stdlib.h>
#include <sched.h>
#include <unistd.h>
```

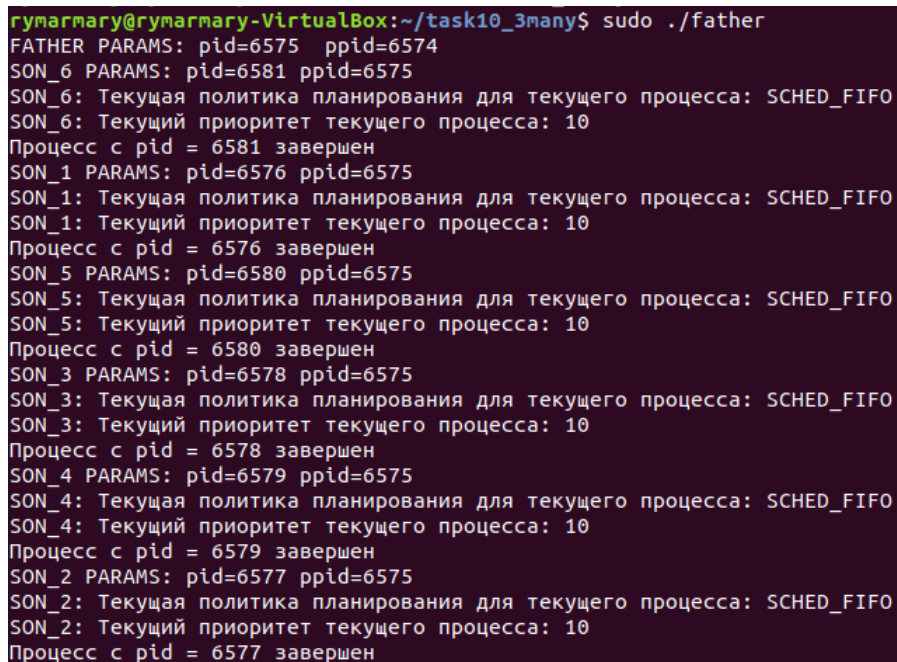
```

#include <sys/types.h>

int main() {
    struct sched_param shdprm; // Значения параметров планирования
    int i, pid,ppid;
    pid=getpid();ppid=getppid();
    printf("SON_1 PARAMS: pid=%i ppid=%i\n",pid,ppid);
    printf ("SON_1: Текущая политика планирования для текущего
процесса: ");

    switch (sched_getscheduler(0)) {
        case SCHED_FIFO:
            printf ("SCHED_FIFO\n");
            break;
        case SCHED_RR:
            printf ("SCHED_RR\n");
            break;
        case SCHED_OTHER:
            printf ("SCHED_OTHER\n");
            break;
        case -1:
            perror ("SCHED_GETSCHEDULER");
            break;
        default:
            printf ("Неизвестная политика планирования\n");
    }
    if (sched_getparam(0, &shdprm) == 0)
        printf ("SON_1: Текущий приоритет текущего процесса: %d\n",
shdprm.sched_priority);
    else
        perror ("SCHED_GETPARAM");
    return 0;
}

```



```

rymary@rymary-VirtualBox:~/task10_3many$ sudo ./father
FATHER PARAMS: pid=6575 ppid=6574
SON_6 PARAMS: pid=6581 ppid=6575
SON_6: Текущая политика планирования для текущего процесса: SCHED_FIFO
SON_6: Текущий приоритет текущего процесса: 10
Процесс с pid = 6581 завершен
SON_1 PARAMS: pid=6576 ppid=6575
SON_1: Текущая политика планирования для текущего процесса: SCHED_FIFO
SON_1: Текущий приоритет текущего процесса: 10
Процесс с pid = 6576 завершен
SON_5 PARAMS: pid=6580 ppid=6575
SON_5: Текущая политика планирования для текущего процесса: SCHED_FIFO
SON_5: Текущий приоритет текущего процесса: 10
Процесс с pid = 6580 завершен
SON_3 PARAMS: pid=6578 ppid=6575
SON_3: Текущая политика планирования для текущего процесса: SCHED_FIFO
SON_3: Текущий приоритет текущего процесса: 10
Процесс с pid = 6578 завершен
SON_4 PARAMS: pid=6579 ppid=6575
SON_4: Текущая политика планирования для текущего процесса: SCHED_FIFO
SON_4: Текущий приоритет текущего процесса: 10
Процесс с pid = 6579 завершен
SON_2 PARAMS: pid=6577 ppid=6575
SON_2: Текущая политика планирования для текущего процесса: SCHED_FIFO
SON_2: Текущий приоритет текущего процесса: 10
Процесс с pid = 6577 завершен

```

Рисунок 20 – Вывод программы

10.4. Да, возможно назначить разные процедуры планирования разным процессам с одинаковыми приоритетами.

Для проведения эксперимента по изучению влияния разных процедур планирования на конкуренцию между процессами с одинаковыми приоритетами, можно создать несколько процессов с одинаковыми уровнями приоритета и разными процедурами планирования. Далее можно запустить эти процессы в одной системе и замерить время их выполнения.

Ниже приведен пример кода, который иллюстрирует, как можно назначить разные процедуры планирования разным процессам с одинаковыми приоритетами. Вывод программы представлен на рисунке 21. Так как у разных процедур одинаковый приоритет, время примерно одинаковое.

Исходный код программы *10_4.c*:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sched.h>
#include <time.h>

void perform_computation() {
    for (int i = 0; i < 1000000000; ++i)
        2 + 2;
}

int main() {
    pid_t pid1, pid2;
    int status;
    struct timespec start_time, end_time;
    double elapsed_time;

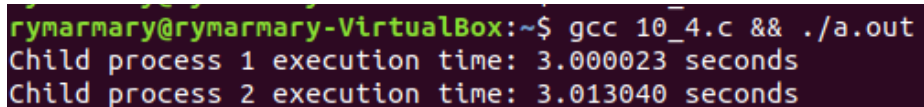
    // Create two child processes with the same priority level
    pid1 = fork();
    if (pid1 == 0) { // child process 1
        // Assign Round-Robin scheduling procedure
        struct sched_param param1 = {.sched_priority = 1};
        sched_setscheduler(0, SCHED_RR, &param1);
        // Perform some computation
        clock_gettime(CLOCK_MONOTONIC, &start_time);
        perform_computation();
        clock_gettime(CLOCK_MONOTONIC, &end_time);
        elapsed_time = (end_time.tv_sec - start_time.tv_sec) +
            (end_time.tv_nsec - start_time.tv_nsec) / 1e9;
    }
```

```

        printf("Child process 1 execution time: %.6f seconds\n",
elapsed_time);
        exit(0);
    }
    else {
        pid2 = fork();
        if (pid2 == 0) { // child process 2
            // Assign First-Come-First-Serve scheduling procedure
            struct sched_param param2 = {.sched_priority = 1};
            sched_setscheduler(0, SCHED_FIFO, &param2);
            // Perform some computation
            clock_gettime(CLOCK_MONOTONIC, &start_time);
            perform_computation();
            clock_gettime(CLOCK_MONOTONIC, &end_time);
            elapsed_time = (end_time.tv_sec - start_time.tv_sec) +
(end_time.tv_nsec - start_time.tv_nsec) / 1e9;
            printf("Child process 2 execution time: %.6f seconds\n",
elapsed_time);
            exit(0);
        }
        else { // parent process
            // Wait for both child processes to finish
            waitpid(pid1, &status, 0);
            waitpid(pid2, &status, 0);
        }
    }

    return 0;
}

```



```

rymary@rymary-VirtualBox:~$ gcc 10_4.c && ./a.out
Child process 1 execution time: 3.000023 seconds
Child process 2 execution time: 3.013040 seconds

```

Рисунок 21 – Вывод программы 10_4.c

11. В отличие от старых версий, современные операционные системы Linux не обладают специальным механизмом, позволяющим приложениям устанавливать длину кванта процессорного времени для планировщика RR. Ранее квант можно было регулировать путем изменения параметра процесса *nice*, где отрицательное значение увеличивало квант, а положительное - уменьшало. Для экспериментального подтверждения ниже приведён программный код родителя *father.c* и потомка *son.c*. Вывод программы представлен на рисунке 22.

Исходный код программы *father.c*:

```

#include <stdio.h>
#include <sched.h>

```

```

#include <sys/mman.h>
#include <time.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <unistd.h>

int main () {
    struct sched_param shdprm; // Значения параметров планирования
    struct timespec qp; // Величина кванта
    int i, pid, pid1, pid2, pid3, ppid, status;

    pid = getpid();
    ppid = getppid();
    printf("FATHER PARAMS: pid=%i  ppid=%i\n", pid, ppid);

    shdprm.sched_priority = 50;
    if (sched_setscheduler(0, SCHED_RR, &shdprm) == -1)
        perror ("SCHED_SETSCHEDULER_1");

    if (sched_rr_get_interval(0, &qp) == 0)
        printf ("Квант при циклическом планировании: %ld сек %ld
nc\n", qp.tv_sec, qp.tv_nsec);
    else
        perror ("SCHED_RR_GET_INTERVAL");

    if ((pid1 = fork()) == 0) {
        if (sched_rr_get_interval(pid1, &qp) == 0)
            printf("SON: Квант процессорного времени: %ld сек %ld
nc\n", qp.tv_sec, qp.tv_nsec);
        execl("son", "son", NULL);
    }

    printf("Процесс с pid = %d завершен\n", wait(&status));
    return 0;
}

```

Исходный код программы *son.c*:

```

#include <stdio.h>
#include <sched.h>
#include <sys/mman.h>
#include <time.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    printf("SONPARAMS:  pid=%i  ppid=%i\n",getpid(),getppid());
    return 0;
}

```

```

rymary@rymary-VirtualBox:~/task11$ gcc father.c -o father && ./father
FATHER PARAMS: pid=7252 ppid=2140
SCHED_SETSCHEDULER_1: Operation not permitted
Квант при циклическом планировании: 0 сек 4000000 нс
SON: Квант процессорного времени: 0 сек 4000000 нс
SONPARAMS: pid=7253 ppid=7252
Процесс с pid = 7253 завершен
rymary@rymary-VirtualBox:~/task11$ sudo ./father
[sudo] password for rymary:
FATHER PARAMS: pid=7255 ppid=7254
Квант при циклическом планировании: 0 сек 100000000 нс
SON: Квант процессорного времени: 0 сек 100000000 нс
SONPARAMS: pid=7256 ppid=7255
Процесс с pid = 7256 завершен

```

Рисунок 22 – Вывод программы

В отличие от более старых версий, современные ОС Linux не предоставляют специального механизма для установки величины кванта процессорного времени для RR-планировщика из приложений. В прошлом квант можно было регулировать, используя параметр процесса *nice*: отрицательное значение *nice* увеличивало квант, а положительное - уменьшало. В разных версиях ядра степень влияния значения *nice* на квант была разной. Однако начиная с версии Linux 2.6.24 квант SCHED_RR не может быть изменен с помощью документированных средств. Экспериментально это можно проверить, используя системную функцию *nice()*. Ниже представлен программный код родителя *father2.c* с этой функцией. Вывод программы показан на рисунке 23.

Исходный код программы *father2.c*:

```

#include <stdio.h>
#include <sched.h>
#include <sys/mman.h>
#include <time.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <unistd.h>

int main () {
    struct sched_param shdprm;
    struct timespec qp;
    int i, pid, pid1, pid2, pid3, ppid, status;

    pid = getpid();
    ppid = getppid();
    printf("FATHER PARAMS: pid=%i ppid=%i\n", pid, ppid);

```

```

if (nice(1000) == -1)
    perror("NICE");
else
    printf("Nice value = %d\n", nice(0));

shdprm.sched_priority = 50;
if (sched_setscheduler(pid, SCHED_RR, &shdprm) == -1)
    perror("SCHED_SETSCHEDULER_1");

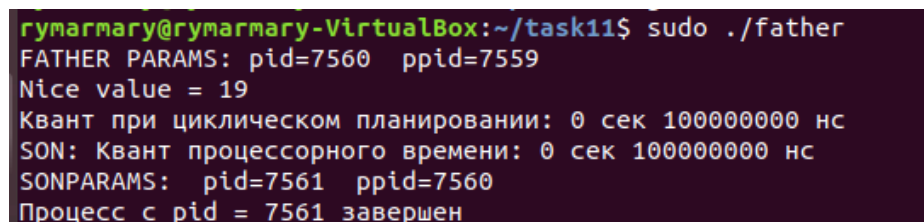
if (sched_rr_get_interval(pid, &qp) == -1)
    perror("SCHED_RR_GET_INTERVAL");
else
    printf("Квант при циклическом планировании: %ld сек %ld
нс\n", qp.tv_sec, qp.tv_nsec);

pid1 = fork();
if (pid1 == 0) {
    if (sched_rr_get_interval(pid1, &qp) == -1)
        perror("SCHED_RR_GET_INTERVAL");
    else
        printf("SON: Квант процессорного времени: %ld сек %ld
нс\n", qp.tv_sec, qp.tv_nsec);

    execl("./son", "son", NULL);
    exit(EXIT_FAILURE);
}

printf("Процесс с pid = %d завершен\n", wait(&status));
return 0;
}

```



```

rymarmary@rymarmary-VirtualBox:~/task11$ sudo ./father
FATHER PARAMS: pid=7560 ppid=7559
Nice value = 19
Квант при циклическом планировании: 0 сек 1000000000 нс
SON: Квант процессорного времени: 0 сек 1000000000 нс
SONPARAMS: pid=7561 ppid=7560
Процесс с pid = 7561 завершен

```

Рисунок 23 – Вывод программы с функцией `nice`

12. Проанализируем наследование на этапах `fork()` и `exec()`. Для этого проведем эксперимент по проверке доступа потомков к файлам, открытым породившим их процессом. Рассмотрим пример кода, в котором в качестве аргументов процессам-потомкам передаются дескрипторы открытого и созданного родительским процессом файлов (в данном примере это *infile.txt* и *outfile.txt*, соответственно). Вывод программы показан на рисунке 24.

Содержимое *infile.txt*: Hello world!

Содержимое *outfile.txt*: eL ol!

Исходный код программы *father.c*:

```
#include <stdio.h>
#include <sched.h>
#include <sys/mman.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

void itoa(char *buf, int value) {
    sprintf(buf, "%d", value);
}

int main(void) {
    int pid, ppid, status;
    int fdrd, fdwr;
    char str1[10], str2[10];
    struct sched_param shdprm;

    if (mlockall(MCL_CURRENT | MCL_FUTURE) < 0)
        perror("mlockall error");

    pid = getpid();
    ppid = getppid();
    shdprm.sched_priority = 1;

    if (sched_setscheduler(0, SCHED_RR, &shdprm) == -1)
        perror("SCHED_SETSCHEDULER_1");

    if ((fdrd = open("infile.txt", O_RDONLY)) == -1)
        perror("Opening file");

    if ((fdwr = creat("outfile.txt", 0666)) == -1)
        perror("Creating file");
    itoa(str1, fdrd);
    itoa(str2, fdwr);

    for (int i = 0; i < 2; i++) {
        if (fork() == 0) {
            shdprm.sched_priority = 50;
            if (sched_setscheduler(0, SCHED_RR, &shdprm) == -1)
                perror("SCHED_SETSCHEDULER_1");
            execl("son", "son", NULL);
        }
    }
    if (close(fdrd) != 0)
        perror("Closing file");
    for (int i = 0; i < 2; i++)
        printf("Process pid = %d completed\n", wait(&status));

    return 0;
}
```


Исходный код программы *son.c*:

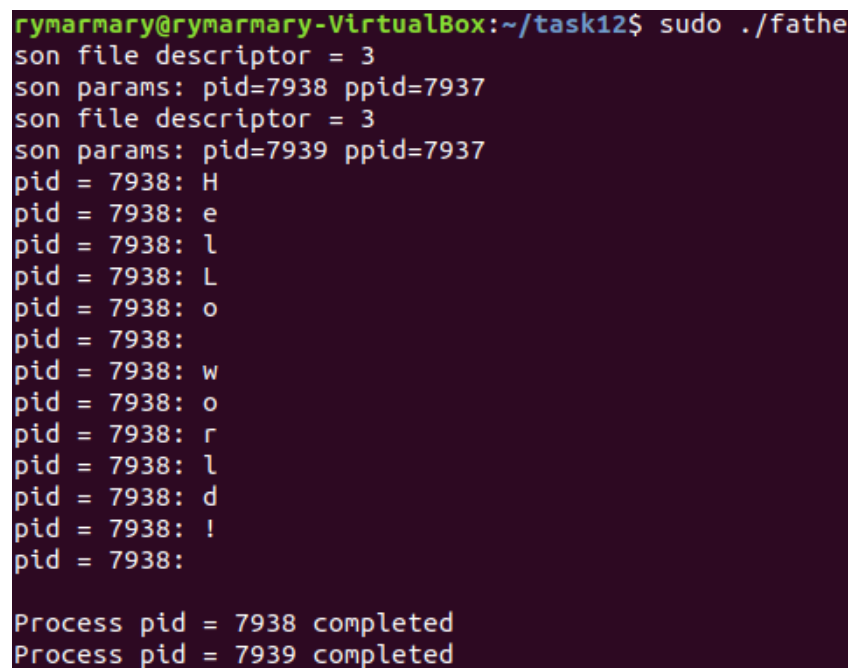
```
#include <sched.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    if (mlockall(MCL_CURRENT | MCL_FUTURE) < 0)
        perror("mlockall error");

    char c;
    int pid, ppid;
    int fdrd = 3;
    int fdwr = 4;
    pid = getpid();
    ppid = getppid();
    printf("son file descriptor = %d\n", fdrd);
    printf("son params: pid=%i ppid=%i\n", pid, ppid);

    sleep(5);

    for (;;) {
        if (read(fdrd, &c, 1) != 1)
            return 0;
        write(fdwr, &c, 1);
        printf("pid = %d: %c\n", pid, c);
    }
    return 0;
}
```



```
rymary@rymary-VirtualBox:~/task12$ sudo ./fathe
son file descriptor = 3
son params: pid=7938 ppid=7937
son file descriptor = 3
son params: pid=7939 ppid=7937
pid = 7938: H
pid = 7938: e
pid = 7938: l
pid = 7938: L
pid = 7938: o
pid = 7938:
pid = 7938: w
pid = 7938: o
pid = 7938: r
pid = 7938: l
pid = 7938: d
pid = 7938: !
pid = 7938:
Process pid = 7938 completed
Process pid = 7939 completed
```

Рисунок 24 – Вывод программы

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

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

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

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

Исходный код программы father.c:

```
#include <sys/mman.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int sid, pid, pid1, ppid, status;
    char command[50];
    if (argc < 2)
        return -1;
    pid = getpid();
    ppid = getppid();
    sid = getsid(pid);
    sprintf(command, "ps -xjf | grep \"STAT\\|\\|%d\\\" > %s", sid,
argv[1]);
    printf("FATHER PARAMS: sid = %i  pid=%i  ppid=%i  \n", sid,
pid, ppid);
    if ((pid1=fork())==0)
        execl("son1", "son1", NULL);

    if(fork()==0)
        execl("son2", "son2", argv[1], NULL);

    if(fork()==0)
        execl("son3", "son3", NULL);

    system(command);
    // waitpid(pid1, &status, WNOHANG);
}
```

Исходный код программы son1.c:

```
#include <stdio.h>
```

```
#include <unistd.h>

int main(){
    int pid,ppid;
    pid=getpid();
    ppid=getppid();
    printf("SON_1  PARAMS:      pid=%i      ppid=%i\nFather  creates
andwaits\n",pid,ppid);
    sleep(1);

    return 0;
}
```

Исходный код программы son2.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]){
    int pid,ppid;
    pid=getpid();
    ppid=getppid();
    char command[50];
    sprintf(command, "ps xjf | grep son2 >> %s", argv[1]);
    printf("SON_2  PARAMS:  pid=%i  ppid=%i\nFather finished before
son terminationwithout waiting for it \n",pid,ppid);
    sleep(20);
    ppid=getppid();
    printf("SON_2      PARAMS      ARE      CHANGED:      pid=%i
ppid=%i\n",pid,ppid);
    system(command);

    return 0;
}
```

Исходный код программы son3.c:

```
#include <stdio.h>
#include <unistd.h>

int main(){
    int pid,ppid;
    pid=getpid();
    ppid=getppid();
    printf("SON_3  PARAMS:      pid=%i      ppid=%i\nson3terminated-
ZOMBIE\n",pid,ppid);
    ppid=getppid();
    printf("SON_3  PARAMS:  pid=%i  ppid=%i\n",pid,ppid);

    return 0;
}
```

а). Как видно из результатов (на рисунке 25), как только процесс-отец завершается, на консоли сразу появляется приглашение на ввод команды. А son2 продолжает свое выполнение в фоновом режиме. Т.к. время выполнения son2 много дольше, то результат выполнения процесса-потомка, появляется уже после приглашения.

```
rymary@rymary-VirtualBox:~/task13/task13_1$ cat res.txt
PPID  PID  PGID  SID  TTY  TPGID  STAT  UID  TIME  COMMAND
2352  2742  2742  2352 pts/0  2742  S+    0    0:00 sudo ./father res.txt
2742  2743  2742  2352 pts/0  2742  S+    0    0:00 \_ ./father res.txt
2743  2744  2742  2352 pts/0  2742  R+    0    0:00 \_ ./father res.t
xt
2743  2745  2742  2352 pts/0  2742  S+    0    0:00 \_ son2 res.txt
2743  2746  2742  2352 pts/0  2742  Z+    0    0:00 \_ [son3] <defunc
t>
2743  2747  2742  2352 pts/0  2742  S+    0    0:00 \_ sh -c ps -xjf
| grep "STAT\|2352" > res.txt
2747  2748  2742  2352 pts/0  2742  R+    0    0:00 \_ ps -xjf
2747  2749  2742  2352 pts/0  2742  S+    0    0:00 \_ grep STAT\
|2352
rymary@rymary-VirtualBox:~/task13/task13_1$ SON_2 PARAMS ARE CHANGED: pi
d=2745 ppid=1272
```

Рисунок 25 – Выполнение 13.1 (а)

б). Раскомментировав строку с waitpid, получаем вывод, представленный на рисунке 26. Процесс-отец запускает процесс-сын и, не ожидая его завершения, завершает своё выполнение.

```
rymary@rymary-VirtualBox:~/task13/task13_1$ sudo ./father res_1.txt
FATHER PARAMS: sid = 2352 pid=2879 ppid=2878
SON_3 PARAMS: pid=2882 ppid=2879
son3terminated-ZOMBIE
SON_3 PARAMS: pid=2882 ppid=2879
SON_2 PARAMS: pid=2881 ppid=2879
Father finished before son termination without waiting for it
SON_1 PARAMS: pid=2880 ppid=2879
Father creates and waits
rymary@rymary-VirtualBox:~/task13/task13_1$ SON_2 PARAMS ARE CHANGED: pi
d=2881 ppid=1272

rymary@rymary-VirtualBox:~/task13/task13_1$ cat res_1.txt
PPID  PID  PGID  SID  TTY  TPGID  STAT  UID  TIME  COMMAND
2352  2878  2878  2352 pts/0  2878  S+    0    0:00 sudo ./father res_1.tx
t
2878  2879  2878  2352 pts/0  2878  S+    0    0:00 \_ ./father res_1.txt
2879  2880  2878  2352 pts/0  2878  S+    0    0:00 \_ son1
2879  2881  2878  2352 pts/0  2878  S+    0    0:00 \_ son2 res_1.txt
2879  2882  2878  2352 pts/0  2878  Z+    0    0:00 \_ [son3] <defunc
t>
2879  2883  2878  2352 pts/0  2878  S+    0    0:00 \_ sh -c ps -xjf
| grep "STAT\|2352" > res_1.txt
2883  2884  2878  2352 pts/0  2878  R+    0    0:00 \_ ps -xjf
2883  2885  2878  2352 pts/0  2878  S+    0    0:00 \_ grep STAT\
|2352
1272  2881  2878  2352 pts/0  2352  S    0    0:00 son2 res_1.txt
2881  2886  2878  2352 pts/0  2352  S    0    0:00 \_ sh -c ps xjf | gre
o son2 >> res_1.txt
```

Рисунок 26 – Выполнение 13.1 (б)

13.1. Системный вызов `kill` посылает сигналы указанным процессам. По умолчанию (если не указано имя или номер сигнала) посылается сигнал `SIGTERM`. Идентификатор процесса является аргументом для этой утилиты: если он больше нуля, то сигнал посылается процессу с указанным `pid`, если он равен нулю, то сигнал посылается всем процессам, принадлежащим пользователю, если он меньше нуля, то он воспринимается как идентификатор группы процессов, и тогда сигнал посылается всей группе. Результат выполнения команды представлен на рисунке 27.

```
gymarmy@gymarmy-VirtualBox:~/task13/task13_1$ kill -l
1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL      5) SIGTRAP
6) SIGABRT     7) SIGBUS     8) SIGFPE      9) SIGKILL     10) SIGUSR1
11) SIGSEGV    12) SIGUSR2    13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT  17) SIGCHLD   18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN    22) SIGTTOU   23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF   28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS     34) SIGRTMIN  35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
```

Рисунок 27 – Результат выполнения системного вызова `kill`

Функция системного вызова `signal` заключается в том, чтобы задать определенные действия для программы в ответ на пришедший сигнал. В качестве действий можно задать следующие значения: `SIG_DFL`, `SIG_IGN` или указатель на собственную функцию обработки.

`SIG_DFL` означает, что процесс должен реагировать на сигнал, как задано по умолчанию (чаще всего это завершение процесса), `SIG_IGN` (нельзя задать для `SIGSTOP` и `SIGKILL`) означает, что нужно игнорировать сигнал.

а). процесс `father` порождает процессы `son1`, `son2`, `son3` и запускает на исполнение программные коды из соответствующих исполнительных файлов;

б). далее родительский процесс осуществляет управление потомками, для этого он генерирует сигнал каждому пользовательскому процессу;

в). в пользовательских процессах-потомках необходимо обеспечить:

для `son1` – реакцию на сигнал по умолчанию;

для son2 – реакцию игнорирования;

для son3 – перехватывание и обработку сигнала.

Исходный код программы son1.c:

```
#include <signal.h>
#include <stdio.h>
#include <sched.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
```

```
int main()
{
    signal(SIGUSR1, SIG_DFL);
    sleep(5);
}
```

Исходный код программы son2.c:

```
#include <signal.h>
#include <stdio.h>
#include <sched.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
```

```
int main()
{
    signal(SIGUSR1, SIG_IGN);
    sleep(5);
}
```

Исходный код программы son3.c:

```
#include <signal.h>
#include <stdio.h>
#include <sched.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
```

```
void SIGUSR1_handler(int sig_no)
{
    printf("SIGUSR1_handler running!\n");
}
```

```
int main() {
    signal(SIGUSR1, SIGUSR1_handler);
    sleep(5);

    return 0;
}
```

```
}
```

Исходный код программы father.c:

```
#include <signal.h>
#include <stdio.h>
#include <sched.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(){
    int pid1, pid2, pid3;
    pid1 = fork();
    if (pid1 == 0)
        execl("son1", "son1", NULL);
    pid2 = fork();
    if (pid2 == 0)
        execl("son2", "son2", NULL);
    if (pid3 == 0)
        execl("son3", "son3", NULL);
    system("echo before signal sent");
    system("ps -l");
    kill(pid1, SIGUSR1);
    kill(pid2, SIGUSR1);
    kill(pid3, SIGUSR1);
    system("echo after signal sent");
    system("ps -l");
}
```

Результат выполнения работы показан на рисунке 28.

```
rymarmar@rymarmar-VirtualBox:~/task13/task13.1$ ./father
before signal sent
F S  UID  PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  2352  2342  0  80   0 -  5646 wait  pts/0    00:00:00 bash
0 S  1000  3225  2352  0  80   0 -  1096 wait  pts/0    00:00:00 father
0 S  1000  3226  3225  0  80   0 -  1096 hrtime pts/0    00:00:00 son1
0 S  1000  3227  3225  0  80   0 -  1096 hrtime pts/0    00:00:00 son2
0 S  1000  3228  3225  0  80   0 -  1096 hrtime pts/0    00:00:00 son3
0 S  1000  3230  3225  0  80   0 -  1159 wait  pts/0    00:00:00 sh
0 R  1000  3231  3230  0  80   0 -  7230 -      pts/0    00:00:00 ps
SIGUSR1_handler running!
after signal sent
F S  UID  PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  2352  2342  0  80   0 -  5646 wait  pts/0    00:00:00 bash
0 S  1000  3225  2352  0  80   0 -  1096 wait  pts/0    00:00:00 father
0 Z  1000  3226  3225  0  80   0 -    0 -      pts/0    00:00:00 son <defunct>
0 S  1000  3227  3225  0  80   0 -  1096 hrtime pts/0    00:00:00 son2
0 Z  1000  3228  3225  0  80   0 -    0 -      pts/0    00:00:00 son <defunct>
0 S  1000  3233  3225  0  80   0 -  1159 wait  pts/0    00:00:00 sh
0 R  1000  3234  3233  0  80   0 -  7230 -      pts/0    00:00:00 ps
```

Рисунок 28 – Результат выполнения программы 13.1

13.2. Организуем посылку сигналов любым двум процессам, находящимся в разных состояниях: активном и пассивном, фиксируя моменты посылки и приема каждого сигнала с точностью до секунды.

Запускаем два файла son1 и son2, после чего происходит вызов ps, далее сигналы уничтожаются, после чего засекается время, когда был отправлен сигнал kill потоку. Принимает сигнал и передает его в функцию по обработке, в которой происходит вывод времени получения. Результаты выполнения программы представлены на рисунке 29.

Исходный код программы father.c:

```
#include <signal.h>
#include <stdio.h>
#include <sched.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

int main(){
    int pid1, pid2, pid3;
    pid1 = fork();
    if (pid1 == 0)
        execl("son1", "son1", NULL);
    pid2 = fork();
    if (pid2 == 0)
        execl("son2", "son2", NULL);
    printf("father running\n");
    system("ps");
    kill(pid1, SIGUSR1);
    kill(pid2, SIGUSR1);
    time_t st_t;
    st_t = time(NULL);
    printf("signals sended at time %s\n", ctime(&st_t));
    sleep(2);
    system("ps");
}
```

Исходный код программы son1.c:

```
#include <signal.h>
#include <stdio.h>
#include <sched.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
```



```

void SIGUSR1_handler(int sig_no)
{
    time_t st_t;
    st_t = time(NULL);
    printf("son1 (sleeping) received the signal at time %s\n",
ctime(&st_t));
    exit(0);
}

int main() {
    signal(SIGUSR1, SIGUSR1_handler);
    printf("son1 running\n");
    sleep(3);
    return 0;
}

```

Исходный код программы son2.c:

```

#include <signal.h>
#include <stdio.h>
#include <sched.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

void SIGUSR1_handler(int sig_no)
{
    time_t st_t;
    st_t = time(NULL);
    printf("son1 (active) received the signal at time %s\n",
ctime(&st_t));
    exit(0);
}

int main() {
    signal(SIGUSR1, SIGUSR1_handler);
    printf("son2 running\n");
    while(1)
    {}

    return 0;
}

```

```

rymary@rymary-VirtualBox:~/task13/task13.2$ ./father
father running
son2 running
son1 running
  PID TTY          TIME CMD
 2352 pts/0        00:00:00 bash
 3294 pts/0        00:00:00 father
 3295 pts/0        00:00:00 son1
 3296 pts/0        00:00:00 son2
 3297 pts/0        00:00:00 sh
 3298 pts/0        00:00:00 ps
signals sent at time Mon May  1 18:42:10 2023
son1 (sleeping) received the signal at time Mon May  1 18:42:10 2023

son1 (active) received the signal at time Mon May  1 18:42:10 2023

  PID TTY          TIME CMD
 2352 pts/0        00:00:00 bash
 3294 pts/0        00:00:00 father
 3295 pts/0        00:00:00 son1 <defunct>
 3296 pts/0        00:00:00 son2 <defunct>
 3299 pts/0        00:00:00 sh
 3300 pts/0        00:00:00 ps

```

Рисунок 29 – Результат выполнения программы 13.2

14. Запущено в фоновом режиме несколько утилит. Использована команда `jobs` для анализа списка заданий и очередности их выполнения. Возвращены невыполненные задания в приоритетный режим командой `fg`. С помощью утилиты `fg` можно повысить приоритет задач. Благодаря ей, сразу начинает выполняться задача 1, причем не в фоновом режиме. Новые добавленные задачи добавляются в конец очереди и начинают выполняться первыми. Получено уведомление о завершении одного из заданий с помощью команды `notify`.

Результат выполнения работы программы показан на рисунке 30.

```

rymary@rymary-VirtualBox:~$ sleep 100 & sleep 110 & sleep 120 & sleep 130
&
[1] 3342
[2] 3343
[3] 3344
[4] 3345
rymary@rymary-VirtualBox:~$ jobs -l
[1] 3342 Running      sleep 100 &
[2] 3343 Running      sleep 110 &
[3]- 3344 Running      sleep 120 &
[4]+ 3345 Running      sleep 130 &
rymary@rymary-VirtualBox:~$ jobs -l %%
[4]+ 3345 Running      sleep 130 &
rymary@rymary-VirtualBox:~$ kill 3344
[3]- Terminated      sleep 120
rymary@rymary-VirtualBox:~$ jobs -l
[1] 3342 Running      sleep 100 &
[2]- 3343 Running      sleep 110 &
[4]+ 3345 Running      sleep 130 &

```

Рисунок 30 – Результат выполнения программ в фоновом режиме

15. Функция `nice` позволяет процессу менять свой приоритет. Аргумент команды – величина, которую нужно прибавить к приоритету процесса. В используемой версии QNX эта команда не работает (не меняет приоритета).

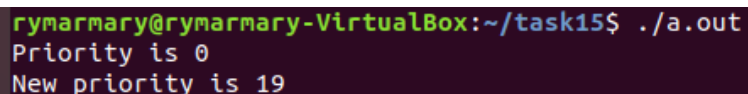
Для изменения приоритета процесса можно воспользоваться командой `setprio`, правда в отличие от команды `nice` она устанавливает приоритет, а не меняет (прибавляет, отнимает). Для процесса с идентификатором `pid` устанавливается приоритет `prio`. Возвращаемое значение – прошлый приоритет.

Функция `getprio` позволяет узнать приоритет процесса – он будет передан в возвращаемом значении. Аргументом является идентификатор процесса, приоритет которого мы хотим узнать. Вывод с `nice()` представлен на рисунке 31, вывод без `nice()` представлен на рисунке 32.

Исходный код программы 15_1.c:

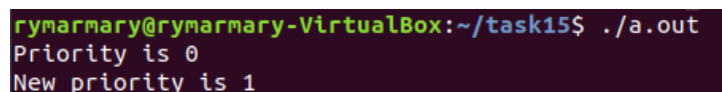
```
#include <signal.h>
#include <stdio.h>
#include <sched.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <sys/resource.h>

int main() {
    int pr, pid;
    pid = getpid();
    pr = getpriority(PRIO_PROCESS, pid);
    printf("Priority is %d\n", pr);
    nice(1000);
    //setpriority(PRIO_PROCESS, pid, pr + 1); // increase priority
by 1
    pr = getpriority(PRIO_PROCESS, pid);
    printf("New priority is %d\n", pr);
    return 0;
}
```



```
rymary@rymary-VirtualBox:~/task15$ ./a.out
Priority is 0
New priority is 19
```

Рисунок 31 – Вывод с `nice()`



```
rymary@rymary-VirtualBox:~/task15$ ./a.out
Priority is 0
New priority is 1
```

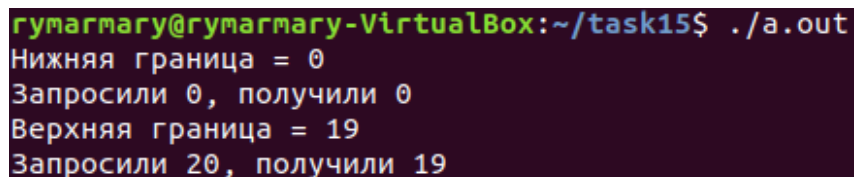
Рисунок 32 – Вывод без `nice()`

Исходный файл 15_2.c:

```
#include <signal.h>
#include <stdio.h>
#include <sched.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <sys/resource.h>

void main()
{
    int pr, pid, i;
    pid=getpid();
    for (i = -100; i < 1; i++)
    {
        setpriority(PRIO_PROCESS, pid, i);
        pr = getpriority(PRIO_PROCESS, pid);
        if (pr != i) continue;
        else
        {
            printf("Нижняя граница = %d\n", pr);
            printf("Запросили %d, получили %d\n", i, pr);
            break;
        }
    }
    for (i = 1; i < 100; i++)
    {
        setpriority(PRIO_PROCESS, pid, i);
        pr = getpriority(PRIO_PROCESS, pid);
        if (pr == i) continue;
        else
        {
            printf("Верхняя граница = %d\n", pr);
            printf("Запросили %d, получили %d\n", i, pr);
            break;
        }
    }
}
```

Результат работы программы показан на рисунке 33.



```
gymarmary@gymarmary-VirtualBox:~/task15$ ./a.out
Нижняя граница = 0
Запросили 0, получили 0
Верхняя граница = 19
Запросили 20, получили 19
```

Рисунок 33 – Результат работы программы 15_2.c

Утилита `top` позволяет выводить информацию о системе, а также список процессов динамически обновляя информацию о потребляемых ими ресурсах.

Видно, что пользовательский приоритет для запуска приложений из shell 20. Результат вывода показан на рисунке 34.

```
top - 19:31:39 up 3:28, 1 user, load average: 0.00, 0.01, 0.00
Tasks: 219 total, 1 running, 184 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.3 us, 0.3 sy, 0.0 ni, 99.3 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 1000372 total, 96908 free, 569316 used, 334148 buff/cache
KiB Swap: 728520 total, 194104 free, 534416 used. 274180 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1422	rymarma+	20	0	3009476	210344	77272	S	0.7	21.0	2:01.68	gnome-shell
948	gdm	20	0	2940740	46616	21060	S	0.3	4.7	0:07.86	gnome-shell
1292	rymarma+	20	0	415608	39480	16172	S	0.3	3.9	0:43.48	Xorg
2599	root	20	0	0	0	0	I	0.3	0.0	0:06.29	kworker/0:+
1	root	20	0	225620	6044	4104	S	0.0	0.6	0:02.71	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kthreadd
3	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	rcu_gp
4	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	rcu_par_gp
6	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker/0:+
8	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	mm_percpu_+
9	root	20	0	0	0	0	S	0.0	0.0	0:00.62	ksoftirqd/0
10	root	20	0	0	0	0	I	0.0	0.0	0:01.77	rcu_sched
11	root	rt	0	0	0	0	S	0.0	0.0	0:00.10	migration/0
12	root	-51	0	0	0	0	S	0.0	0.0	0:00.00	idle_injec+
14	root	20	0	0	0	0	S	0.0	0.0	0:00.00	cpuhp/0
15	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kdevtmpfs
16	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	netns
17	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcu_tasks_+
18	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kauditd
19	root	20	0	0	0	0	S	0.0	0.0	0:00.01	khungtaskd
20	root	20	0	0	0	0	S	0.0	0.0	0:00.00	oom_reaper

Рисунок 34 – Утилита top

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

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

Системные процессы обычно принадлежат пользователю root или системному пользователю имеют имена процессов, указывающие на их

назначение, в то время как пользовательские процессы принадлежат обычным пользователям.

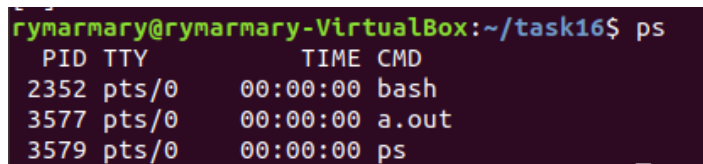
16. **SIGHUP** – это сигнал, посылаемый процессу для уведомления о потере соединения с управляющим терминалом пользователя. Потеря соединения, в частности, возникает при выходе пользователя из системы. Этот сигнал может быть перехвачен или проигнорирован программой.

Игнорирование можно установить, начав выполнять процесс с утилитой **nohup**. Помимо настройки игнорирования она обеспечивает запуск программы (не в фоновом режиме), перенаправляя весь вывод в файл **nohup.out** в текущей директории или, если его невозможно создать, в домашнем каталоге пользователя (если и там его невозможно создать, то команда просто не запустится). После потери связи с терминалом программа продолжит выполняться в фоновом режиме. Запуск команды **ps** до выхода из системы представлен на рисунке 35, после выхода – на рисунке 36.

Исходный код программы 16.c:

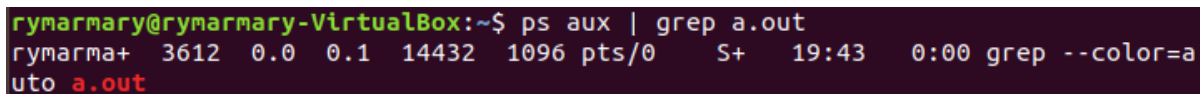
```
#include <signal.h>
#include <stdio.h>
#include <sched.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <sys/resource.h>

int main() {
    sleep(300);
    printf("some text");
    return 0;
}
```



```
gymarmary@gymarmary-VirtualBox:~/task16$ ps
  PID TTY          TIME CMD
 2352 pts/0        00:00:00 bash
 3577 pts/0        00:00:00 a.out
 3579 pts/0        00:00:00 ps
```

Рисунок 35 – До выхода из системы



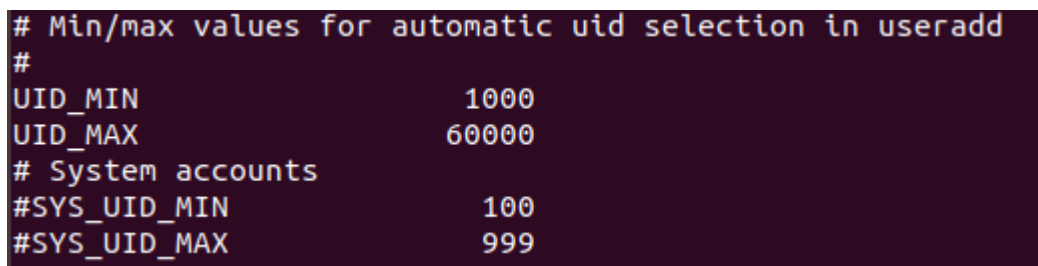
```
gymarmary@gymarmary-VirtualBox:~$ ps aux | grep a.out
gymarma+ 3612  0.0  0.1 14432 1096 pts/0    S+   19:43   0:00 grep --color=a
uto a.out
```

Рисунок 36 – После выхода из системы

17. Чтобы определить UID процесса в системе на базе Unix, можно использовать команду "ps" вместе с опцией "u". Например, чтобы отобразить UID всех запущенных процессов, можно использовать следующую команду: ps uax.

В Linux минимальное значение UID обычно равно 0, которое принадлежит пользователю root. Максимальное значение UID обычно равно 4294967294 ($2^{32} - 2$), которое зарезервировано как максимальное значение UID и используется для представления пользователя nobody.

Однако фактическое максимальное значение UID может быть настроено во время установки дистрибутива Linux или может быть изменено позже системным администратором. Максимальное значение UID обычно определяется в файле "/etc/login.defs" или в файле "/etc/nsswitch.conf". Эти данные представлены на рисунке 37.



```
# Min/max values for automatic uid selection in useradd
#
UID_MIN                1000
UID_MAX                60000
# System accounts
#SYS_UID_MIN           100
#SYS_UID_MAX           999
```

Рисунок 37 – Данные из файла /etc/login.defs

В Linux минимальное значение PID обычно равно 1, которое зарезервировано для процесса инициализации. Максимальное значение PID может варьироваться в зависимости от конфигурации ядра.

В более старых версиях Linux максимальное значение PID было ограничено 32 767. Однако в более поздних версиях ядра Linux максимальное значение PID намного выше и может быть настроено во время компиляции с помощью опции "CONFIG_PID_MAX".

По умолчанию в большинстве дистрибутивов Linux максимальное значение PID установлено равным 2^{22} или 4,194,304. Однако это значение может быть изменено в процессе сборки ядра или во время выполнения с использованием параметров ядра "pid_max".

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

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

Вот некоторые общие системные процессы и их цели:

init/systemd: первый процесс, запускаемый при загрузке системы на базе Unix. Он отвечает за запуск системных служб и управление системными ресурсами.

kernel: ядро операционной системы, которое управляет системными ресурсами и предоставляет низкоуровневые службы другим процессам.

sshd: демон, который обеспечивает безопасный shell (SSH) доступ к системе.

crond: демон, который выполняет запланированные задания с заданными интервалами.

syslogd/rsyslogd: демон системного журнала, который собирает и регистрирует сообщения от различных системных процессов.

httpd/nginx: демон веб-сервера, который обслуживает HTTP-запросы клиентам.

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

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

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

18. Подготовлена программа, формирующая несколько нитей. Каждая нить в цикле: выводит на печать собственное имя и инкрементирует переменную времени, после чего "засыпает" (sleep(5); sleep(1); -для первой и второй нитей соответственно), на экран должно выводиться имя нити и количество пятисекундных (для первой) и секундных (для второй) интервалов функционирования каждой нити.

Исходный код программы 18.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#define NUM_THREADS 2

int count1 = 0;
int count2 = 0;

void *thread_func(void *arg) {
    int id = *((int*) arg);

    while (1) {
        if (id == 1) {
            printf("Thread %d: %d\n", id, count1);
            count1++;
            sleep(5);
        } else {
            printf("Thread %d: %d\n", id, count2);
            count2++;
            sleep(1);
        }
    }

    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];
    int i;

    for (i = 0; i < NUM_THREADS; i++) {
        thread_args[i] = i+1;
        if (pthread_create(&threads[i], NULL, thread_func,
&thread_args[i])) {
```

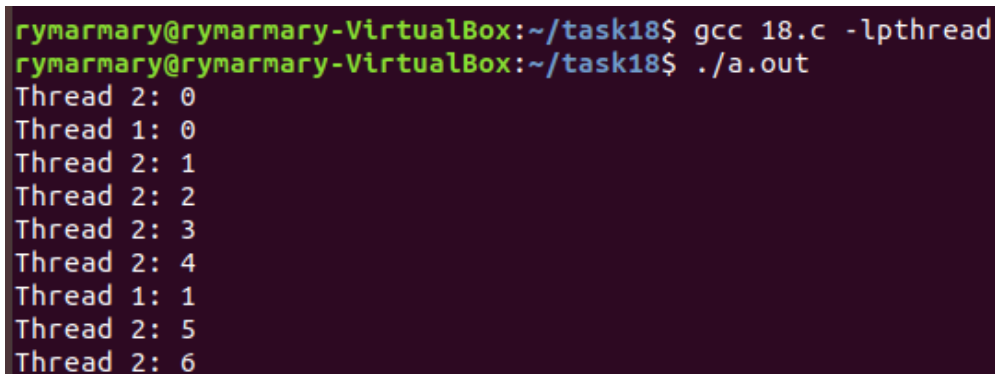
```

        fprintf(stderr, "Error creating thread %d\n", i+1);
        exit(1);
    }
}

for (i = 0; i < NUM_THREADS; i++) {
    if (pthread_join(threads[i], NULL)) {
        fprintf(stderr, "Error joining thread %d\n", i+1);
        exit(1);
    }
}
return 0;
}

```

Вывод программы представлен на рисунке 38.



```

gymarmary@gymarmary-VirtualBox:~/task18$ gcc 18.c -lpthread
gymarmary@gymarmary-VirtualBox:~/task18$ ./a.out
Thread 2: 0
Thread 1: 0
Thread 2: 1
Thread 2: 2
Thread 2: 3
Thread 2: 4
Thread 1: 1
Thread 2: 5
Thread 2: 6

```

Рисунок 38 – Программа 18.c

19. При попытке удаления нити, удаляется процесс в целом, поскольку все нити имеют одинаковый идентификатор. Вывод программы представлен на рисунке 39.

Исходный код программы 19.c:

```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <signal.h>
#define NUM_THREADS 2

int count1 = 0;
int count2 = 0;

void *thread_func(void *arg) {
    int id = *((int*) arg);
    while (1) {
        if (id == 1) {
            printf("Thread %d: %d\n", id, count1);
            count1++;
            sleep(5);
        }
    }
}

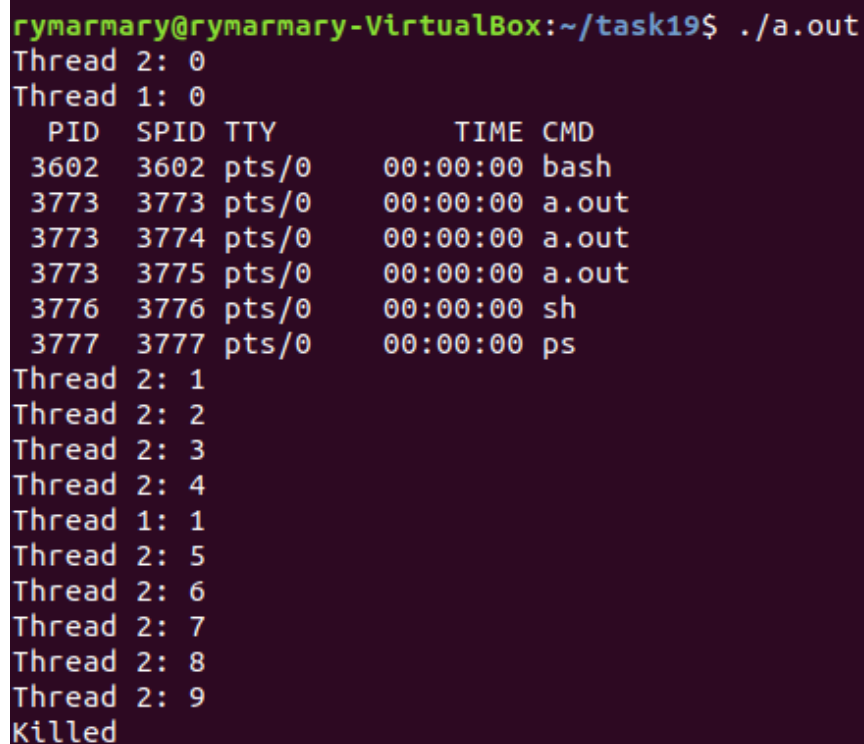
```

```

        } else {
            printf("Thread %d: %d\n", id, count2);
            count2++;
            sleep(1);
        }
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];
    int i;
    // create two child threads
    for (i = 0; i < NUM_THREADS; i++) {
        thread_args[i] = i+1;
        if (pthread_create(&threads[i], NULL, thread_func,
&thread_args[i])) {
            fprintf(stderr, "Error creating thread %d\n", i+1);
            exit(1);
        }
    }
    system("ps -T");
    // join remaining child thread
    if (pthread_join(threads[0], NULL)) {
        fprintf(stderr, "Error joining thread 1\n");
        exit(1);
    }
    return 0;
}

```



```

rymarmary@rymarmary-VirtualBox:~/task19$ ./a.out
Thread 2: 0
Thread 1: 0
  PID  SPID  TTY          TIME CMD
 3602  3602  pts/0        00:00 bash
 3773  3773  pts/0        00:00 a.out
 3773  3774  pts/0        00:00 a.out
 3773  3775  pts/0        00:00 a.out
 3776  3776  pts/0        00:00 sh
 3777  3777  pts/0        00:00 ps
Thread 2: 1
Thread 2: 2
Thread 2: 3
Thread 2: 4
Thread 1: 1
Thread 2: 5
Thread 2: 6
Thread 2: 7
Thread 2: 8
Thread 2: 9
Killed

```

Рисунок 39 – Вывод программы 19.c

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

Можно сделать вывод, что с помощью указанного способа удалить только вторую нить не удалось, удалился процесс в целом. Вывод программы показан на рисунке 40.

Исходный код программы 20.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <signal.h>
#define NUM_THREADS 2

int count1 = 0;
int count2 = 0;

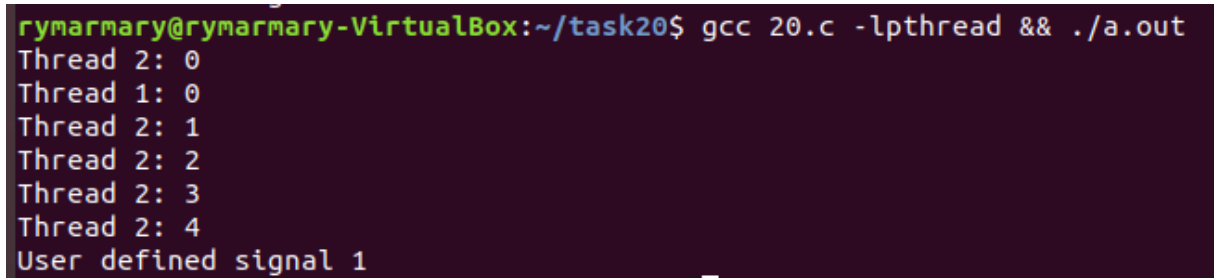
pthread_t threads[NUM_THREADS];
void *thread_func(void *arg) {
    int id = *((int*) arg);
    while (1) {
        if (id == 1) {
            printf("Thread %d: %d\n", id, count1);
            count1++;
            sleep(5);
            printf("Удаление второй нити.");
            pthread_kill(threads[1], SIGUSR1);
        } else {
            printf("Thread %d: %d\n", id, count2);
            count2++;
            sleep(1);
        }
    }
    pthread_exit(NULL);
}

int main() {
    int thread_args[NUM_THREADS];
    int i;
    for (i = 0; i < NUM_THREADS; i++) {
        thread_args[i] = i+1;
        if (pthread_create(&threads[i], NULL, thread_func,
&thread_args[i])) {
            fprintf(stderr, "Error creating thread %d\n", i+1);
        }
    }
}
```

```

        exit(1);
    }
}
if (pthread_join(threads[0], NULL)) {
    fprintf(stderr, "Error joining thread 1\n");
    exit(1);
}
return 0;
}

```



```

rymary@rymary-VirtualBox:~/task20$ gcc 20.c -lpthread && ./a.out
Thread 2: 0
Thread 1: 0
Thread 2: 1
Thread 2: 2
Thread 2: 3
Thread 2: 4
User defined signal 1

```

Рисунок 40 – Вывод программы 20.c

21. С помощью такого управления удалось корректно организовать удаление одной нити: все остальные нити данного процесса сохранились и продолжили выполняться. Вывод программы представлен на рисунке 41.

Последняя модификация предполагает создание собственного обработчика сигнала, содержащего уведомление о начале его работы и возврат посредством функции `pthread_exit(NULL)`.

Исходный код программы 21.c:

```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <signal.h>

pthread_t t1, t2;
void SIGUSR1_handler(int sig_no){
    printf("SIGUSR1 received by thread2\n");
    pthread_exit(NULL);
}

void* thread1(void* arg) {
    int count = 0;
    int i;
    printf("Thread 1 started\n");
    for (i = 0; i < 2; i++) {
        count++;
    }
}

```

```

        sleep(5);
        printf("Thread 1: slept %d times\n", count);
        pthread_kill(t2, SIGUSR1);
    }
    pthread_exit(NULL);
}
void* thread2(void* arg) {
    int count = 0;
    int i;
    printf("Thread 2 started\n");
    system("ps -T");
    signal(SIGUSR1, SIGUSR1_handler);
    for (i = 0; i < 10; i++) {
        count++;
        sleep(1);
        printf("Thread 2: slept %d times\n", count);
    }
    pthread_exit(NULL);
}
int main() {
    pthread_create(&t1, NULL, thread1, NULL);
    pthread_create(&t2, NULL, thread2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}

```

```

rymarmary@rymarmary-VirtualBox:~/task21$ gcc 21.c -lpthread && ./a.out
Thread 2 started
Thread 1 started
  PID  SPID  TTY          TIME CMD
 3602  3602  pts/0        00:00:00 bash
 3852  3852  pts/0        00:00:00 a.out
 3852  3853  pts/0        00:00:00 a.out
 3852  3854  pts/0        00:00:00 a.out
 3855  3855  pts/0        00:00:00 sh
 3856  3856  pts/0        00:00:00 ps
Thread 2: slept 1 times
Thread 2: slept 2 times
Thread 2: slept 3 times
Thread 2: slept 4 times
Thread 1: slept 1 times
SIGUSR1 received by thread2
Thread 1: slept 2 times

```

Рисунок 41 – Вывод программы 21.c

22. Пример простейшего кода, который позволяет перехватить сигнал, генерируемый в результате нажатия комбинации клавиш (Ctrl+C). Вывод программы показан на рисунке 42.

Исходный код программы 22.c:

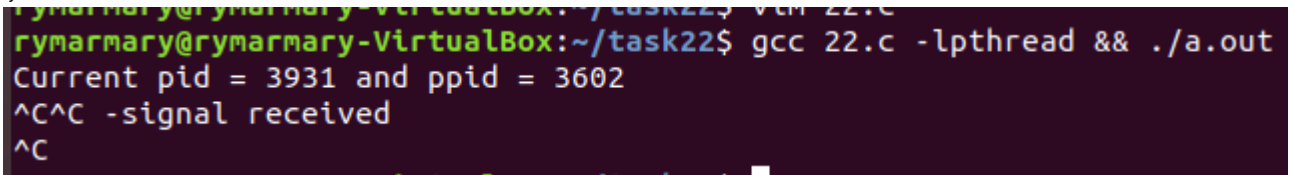
```

#include <stdio.h>
#include <signal.h>
#include <stdio.h>
#include <sched.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

void handler(){
    puts("^C -signal received");
    signal(SIGINT, SIG_DFL);
}

int main(){
    int pid, ppid;
    pid = getpid();
    ppid = getppid();
    printf("Current pid = %d and ppid = %d\n", pid, ppid);
    signal(SIGINT, handler);
    while(1);
    return 0;
}

```



```

rymarmary@rymarmary-VirtualBox:~/task22$ gcc 22.c -lpthread && ./a.out
Current pid = 3931 and ppid = 3602
^C^C -signal received
^C

```

Рисунок 42 – Вывод программы 22.c

Сигнал `^C` перехватывается и однократно вызывается обработчик `handler`, который выводит строку, оповещающую о получении сигнала, после чего возвращается обработчик `SIGINT` по умолчанию, результатом выполнения которого является принудительное завершение программы. Поэтому при повторном нажатии клавиш `^C`, текущая программа прерывается, и выводится приглашение командной строки.

Перехвачен сигнал «`CTRLC`» для процесса и потока многократно с восстановлением исходного обработчика после нескольких раз срабатывания.

23. Ознакомиться с полным перечнем сигналов можно с помощью команды `kill — 1` в командном интерпретаторной реализации ОС.

```
rymary@rymary-VirtualBox:~/task22$ kill -l
1) SIGHUP      2) SIGINT      3) SIGQUIT      4) SIGILL      5) SIGTRAP
6) SIGABRT     7) SIGBUS     8) SIGFPE      9) SIGKILL     10) SIGUSR1
11) SIGSEGV    12) SIGUSR2    13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT  17) SIGCHLD   18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN    22) SIGTTOU    23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF   28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS     34) SIGRTMIN   35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
```

Рисунок 43 – Вывод утилиты kill -l

Рассмотрим некоторые из сигналов базового списка:

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

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

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

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

11) SIGSEGV посылается процессу, который пытается обратиться к не принадлежащей ему области памяти. Если обработчик сигнала не установлен, программа завершается с сохранением на диске образа памяти.

15) SIGTERM вызывает «вежливое» завершение программы. Получив этот сигнал, программа может выполнить необходимые перед завершением операции (например, высвободить занятые ресурсы).

Получение SIGTERM свидетельствует не об ошибке в программе, а о желании ОС или пользователя завершить ее.

17) SIGCHLD посылается процессу в том случае, если его дочерний процесс завершился или был приостановлен. Родительский процесс также получит этот сигнал, если он установил режим отслеживания сигналов дочернего процесса и дочерний процесс получил какой-либо сигнал. По умолчанию сигнал SIGCHLD игнорируется.

18) SIGCONT возобновляет выполнение процесса, остановленного сигналом SIGSTOP.

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

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

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

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

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

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

Исходный код программы 24.c:

```
#include <signal.h>
#include <pthread.h>
#include <stdio.h>
```

```

#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/syscall.h>
#include <time.h>
#define BILLION 1000000000.0

typedef struct Data Data;
const int COUNT = 1500000;
const int COUNT_THREADS = 6;
const int LENGHT_FILE_LINE = 10;

void printPS()
{
    char command[80];
    sprintf(command, "ps -T -p %d -o f,s,pid,ppid,spid,cls,pri,ni,stat,cmd,rtprio > file.txt",
getpid());
    system(command);
}

void *Thread_func(void *arg) {
    struct timespec start, end;

    clock_gettime(CLOCK_REALTIME, &start);

    int *id = (int *) arg;
    int tid = syscall(SYS_gettid);
    int pid = getpid();
    int a = 0;
    printPS();
    printf("\nThread_%d with tid = %d and pid = %d is started\n",
        *id, tid, pid);
    for (int i = 0; i < COUNT; ++i) {
        a++;
    }
    clock_gettime(CLOCK_REALTIME, &end);
    double time_spent = (end.tv_sec - start.tv_sec) +
        (end.tv_nsec - start.tv_nsec) / BILLION;

    printf("\nThread_%d with id = %d and pid = %d is completed,
%fs\n",
        *id, tid, pid, time_spent);
}

int read_from_file(int *priors, int *policies, int *isInherits)
{
    FILE *file = fopen("params.txt", "r");
    char line[LENGHT_FILE_LINE];
    if(file)
    {
        int i = 0;

```

```

while(fgets(line, LENGHT_FILE_LINE, file) != NULL)
{
    if(i < 6)
    {
        priors[i] = atoi(line);
    } else if(i < 12) {
        policies[i - 6] = atoi(line);
    } else
        *isInherits = atoi(line);
    i++;
}
}
else
    return -1;
fclose(file);
return 0;
}

int main()
{
    pthread_t threads[COUNT_THREADS];
    pthread_attr_t thread_attributes[COUNT_THREADS];
    int priorities[COUNT_THREADS], policies[COUNT_THREADS];
    int inherit = 0;
    int IDs[COUNT_THREADS];
    for(int i = 0; i < COUNT_THREADS; i++)
        IDs[i] = i + 1;

    int policy;
    struct sched_param param;

    for(int i = 0; i < COUNT_THREADS; i++) //
        инициализируем //
        pthread_attr_init(&thread_attributes[i]); //
        описателей атрибутов

    read_from_file(priorities, policies, &inherit);

    for(int i = 0; i < COUNT_THREADS; i++)
    {
        pthread_attr_setschedpolicy(&thread_attributes[i],
        policies[i]);
        param.sched_priority = priorities[i];
        pthread_attr_setschedparam(&thread_attributes[i], &param);
    }

    if(inherit == 1)
    {
        for(int i = 0; i < COUNT_THREADS; i++)
            pthread_attr_setdetachstate(&thread_attributes[i],
            PTHREAD_INHERIT_SCHED);
    }
    else

```

```

    {
        for(int i = 0; i < COUNT_THREADS; i++)
            pthread_attr_setinheritsched(&thread_attributes[i],
PTHREAD_EXPLICIT_SCHED);
    }

    for(int i = 0; i < COUNT_THREADS; i++)
    {
        pthread_attr_getschedparam(&thread_attributes[i], &param);
        pthread_attr_getschedpolicy(&thread_attributes[i],
&policy);
        printf("Поток№%d, его приоритет = %d\n", i + 1,
param.sched_priority);
    }

    switch (policy) {
        case SCHED_FIFO:
            printf ("Политика процесса: SCHED_FIFO\n");
            break;
        case SCHED_RR:
            printf ("Политика процесса: SCHED_RR\n");
            break;
        case SCHED_OTHER:
            printf ("Политика процесса: SCHED_OTHER\n");
            break;
        case -1:
            perror ("Политика процесса: SCHED_GETSCHEDULER");
            break;
        default:
            printf ("Политика процесса: Неизвестная политика
планирования\n");
    }

    for(int i = 0; i < COUNT_THREADS; i++)
    {
        if(pthread_create(&threads[i], &thread_attributes[i],
Thread_func, &IDs[i]))
            perror("Статус создания потока");
    }
    for(int i = 0; i < COUNT_THREADS; i++)
        pthread_join(threads[i], NULL);
    for(int i = 0; i < COUNT_THREADS; i++)
        pthread_attr_destroy(&thread_attributes[i]);
    return 0;
}

```

Вывод представлен на рисунках 44 и 45.

```
rymarmary@rymarmary-VirtualBox:~$ sudo ./a.out
Поток№1, его приоритет = 50
Поток№2, его приоритет = 50
Поток№3, его приоритет = 50
Поток№4, его приоритет = 50
Поток№5, его приоритет = 50
Поток№6, его приоритет = 50
Политика процесса: SCHED_FIFO
```

Рисунок 44 – Политика планирования FIFO

```
rymarmary@rymarmary-VirtualBox:~$ sudo ./a.out
Поток№1, его приоритет = 10
Поток№2, его приоритет = 20
Поток№3, его приоритет = 5
Поток№4, его приоритет = 15
Поток№5, его приоритет = 30
Поток№6, его приоритет = 3
Политика процесса: SCHED_RR
```

Рисунок 45 – Политика планирования RR

Выводы.

В ходе выполнения лабораторной работы по теме "Управление процессами и потоками" в операционных системах на примере ОС Линукс, мы изучили различные аспекты управления процессами и потоками, такие как создание, приоритеты, планирование и синхронизация. Также были рассмотрены различные подходы к управлению процессами и потоками, в том числе с использованием многопоточности и многопроцессорности.

Анализ результатов показал, что управление процессами и потоками играет важную роль в эффективном функционировании операционных систем. Для достижения максимальной производительности и избегания конфликтов между процессами и потоками необходимо правильно настроить приоритеты и распределение ресурсов между ними.

В процессе выполнения лабораторной работы мы углубили свои знания в области управления процессами и потоками в операционных системах, а также узнали, какие методы и подходы можно использовать для оптимизации работы системы и повышения ее производительности. Кроме того, было выяснено, что для более эффективного использования ресурсов процессора и оперативной памяти следует использовать механизмы управления памятью и планирования процессов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Столяров С. Операционная система UNIX. - М.: Издательский дом "Вильямс", 2017. - 528 с.
2. Лав К. Поиск в Linux. Как найти все, что вам нужно на вашей системе. - СПб.: Питер, 2019. - 320 с.
3. Мауриер Д. Введение в UNIX. - М.: Издательский дом "Вильямс", 2015. - 576 с.
4. Комаров С. В., Комаров Д. С. Операционные системы: учебное пособие. - СПб.: БХВ-Петербург, 2016. - 592 с.
5. Орейлли Б., Маккрейни Дж. UNIX. Программное окружение. - СПб.: Питер, 2015. - 932 с.
6. Бач Л., Муир М. Linux. Руководство системного администратора. - СПб.: Питер, 2019. - 1200 с.
7. Официальная документация Linux:
<https://www.kernel.org/doc/html/latest/>
8. Руководства и документация по Linux на сайте Linux.org:
<https://www.linux.org/docs/>
9. Руководства и документация по Linux на сайте Ubuntu:
<https://help.ubuntu.com/>