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

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Операционные системы»

Тема: Системное программирование в ОС семействах UNIX

Студентка гр. 1304 _____ Чернякова В.А.

Преподаватель _____ Душутина Е.В.

Санкт-Петербург

2023

Цель работы.

Изучить системное программирование в ОС семейства UNIX.

Выполнение работы.

Модель ОС:

```
Linux Valeriya 4.15.0-142-generic #146~16.04.1-Ubuntu SMP Tue Apr 13 09:27:15 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
```

1. Создадим программу с псевдораспараллеливанием вычислений посредством порождения процесса-потомка.

task1.c

```c
#include <stdio.h>
#include <math.h>
#include <sys/resource.h>
#include <stdlib.h>

void main(int argc, char* argv[])
{
    int pid;
    //Создание нового процесса с помощью fork(). pid = PID процесса.
    pid = fork();
    //При вызове функции fork() вернулась ошибка (-1)
    if(pid == -1)
    {
        //Вывод ошибки
        perror("fork error");
        //Выход из программы с кодом соотвествующей ошибки
        exit(1);
    }
    //Вывод значения полученного PID
    printf("Полученный pid с помощью fork():\n");
    printf("pid=%i\n",pid);
    //PID != 0 => родительский процесс. Значение PID равно дочернему процессу
    if(pid != 0)
    {
        //Вывод pid и ppid родителя
        printf("РОДИТЕЛЬ\n");
        printf("pid = %d, ppid = %d\n", getpid(), getppid());
    }
    //PID = 0 => процесс - потомок.
    else
    {
        //Вывод pid и ppid потомка
```

```
        printf("ПОТОМОК\n");
        printf("pid = %d, ppid = %d\n", getpid(), getppid());
        //Изменение перменной, так как процесс потомок
    }
    printf("Программа завершена\n\n");
    exit(1);
}
```

Работа программы task1.c

```
LERA2003@VALERIYA:~/OS_LAB34/LB3$ ./TASK1
ПОЛУЧЕННЫЙ PID С ПОМОЩЬЮ FORK():
PID=6255
РОДИТЕЛЬ
PID = 6254, PPID = 6230
ПРОГРАММА ЗАВЕРШЕНА

ПОЛУЧЕННЫЙ PID С ПОМОЩЬЮ FORK():
PID=0
ПОТОМОК
PID = 6255, PPID = 6254
ПРОГРАММА ЗАВЕРШЕНА
```

При вызове fork() порождается новый процесс (процесс-потомок), который почти идентичен порождающему процессу-родителю.

При успешном завершении родителю возвращается PID процесса-потомка, а процессу-потомку возвращается 0. При ошибке родительскому процессу возвращается - 1.

Функции getpid() - получить идентификатор текущего процесса, getppid() - получить идентификатор родительского процесса. Соответственно, идентификатор getpid() в родительском процессе, совпадает с идентификатором getppid() в дочернем процессе. Ну а значение getpid() в дочернем процессе совпадет со значением, получаемой первой раз функцией fork().

Когда дочерний процесс завершается, связь его с родителем сохраняется, пока родительский процесс не завершится или не вызовет функцию wait. Т.е. дочерний процесс остается в системе.

Заметим, что ppid потомка равен pid родителя. Фраза “Программа завершена” выводится дважды, что свидетельствует о том, что один и тот же сегмент кода был исполнен обоими процессами.

Проведем эксперимент: будем создавать потомков внутри других потомков.

task1_2.c

```
#include <stdio.h>
#include <math.h>
#include <sys/resource.h>
#include <stdlib.h>

void main(int argc, char* argv[])
{
    int pid1, pid2;
    //Создание нового процесса с помощью fork(). pid = PID процесса.
    pid1 = fork();
    //При вызове функции fork() вернулась ошибка (-1)
    if(pid1 == -1)
    {
        //Вывод ошибки
        perror("fork error");
        //Выход из программы с кодом соотвествующей ошибки
        exit(1);
    }
    //PID != 0 => родительский процесс. Значение PID равно дочернему
процессу
    if(pid1 != 0)
    {
        //Вывод pid и ppid родителя
        printf("РОДИТЕЛЬ\n");
        printf("pid = %d, ppid = %d\n", getpid(), getppid());
    }
    //PID = 0 => процесс - потомок.
    else
    {
        printf("РОДИТЕЛЬ|ПОТОМОК1\n");
        printf("pid = %d, ppid = %d\n", getpid(), getppid());
        pid2 = fork();
```

```
        if(pid2 == -1)
        {
            //Вывод ошибки
            perror("fork error");
            //Выход из программы с кодом соотвествующей ошибки
            exit(1);
        }
        //PID = 0 => процесс - потомок.
        if (pid2 == 0)
        {
            //Вывод pid и ppid потомка
            printf("ПОТОМОК2\n");
            printf("pid = %d, ppid = %d\n", getpid(), getppid());
        }
    }
    printf("Программа завершена\n\n");
    exit(1);
}
```

Работа программы task1_2.c

```
lera2003@Valeriya:~/OS_lab34/lb3$ ./task1_2
РОДИТЕЛЬ
РОДИТЕЛЬ|ПОТОМОК1
PID = 6326, PPID = 6230
PID = 6327, PPID = 6326
Программа завершена
Программа завершена
ПОТОМОК2
PID = 6328, PPID = 6327
Программа завершена
```

“Программа завершена” выводится 3 раза, что свидетельствует о том, что все 3 процессы выполнены. PPID потомка, порожденного внутри другого потомка равен PID первого потомка. То есть всеми процессами был исполнен один и тот же сегмент кода.

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.

task2.c

```c
#include <stdio.h>
#include <math.h>
#include <sys/resource.h>
#include <stdlib.h>

void main(int argc, char* argv[])
{
    int pid;
    //Переменная для отслеживания работы процесса
    int n=1;
    printf("Начальное значение n: %d\n", n);
    //Создание нового процесса с помощью fork(). pid = PID процесса.
    pid = fork();
    //При вызове функции fork() вернулась ошибка (-1)
    if(pid == -1)
    {
        //Вывод ошибки
        perror("fork error");
        //Выход из программы с кодом соотвествующей ошибки
        exit(1);
    }
    //PID != 0 => родительский процесс. Значение PID равно дочернему
процессу
    if(pid != 0)
    {
        //Вывод pid и ppid родителя
        printf("Значение n до изменения: %d\n", n);
        n += 5;
        printf("РОДИТЕЛЬ: pid = %d, ppid = %d. Значение n: %i\n", getpid(),
getppid(), n);
        printf("Значение n: %d\n", n);
        //Изменение перменной, так как процесс родитель
    }
    //PID = 0 => процесс - потомок.
    else
    {
        //Вывод pid и ppid потомка
        printf("Значение n до изменения: %d\n", n);
        n -= 5;
        printf("ПОТОМОК: pid = %d, ppid = %d. Значение n: %i\n", getpid(),
getppid(), n);
```

```
        printf("Значение n: %d\n", n);
    }
    printf("Программа завершена\n\n");
    exit(1);
}
```

Работа программы task2.c

```
LERA2003@VALERIYA:~/OS_LAB34/LB3$ ./TASK2
НАЧАЛЬНОЕ ЗНАЧЕНИЕ N: 1
ЗНАЧЕНИЕ N ДО ИЗМЕНЕНИЯ: 1
РОДИТЕЛЬ: PID = 6366, PPID = 6230. ЗНАЧЕНИЕ N: 6
ЗНАЧЕНИЕ N: 6
ПРОГРАММА ЗАВЕРШЕНА

ЗНАЧЕНИЕ N ДО ИЗМЕНЕНИЯ: 1
ПОТОМОК: PID = 6367, PPID = 6366. ЗНАЧЕНИЕ N: -4
ЗНАЧЕНИЕ N: -4
ПРОГРАММА ЗАВЕРШЕН
```

В программе переменной pid присваивается значение фукнции fork(). Данная функция возвращается дважды - в родительском и дочернем процессе. В данной программе получается так, что в начале полностью работает родительский процесс, а после его завершения начинает работать дочерний. То есть в начале процессорным ресурсом владеет родитель, потом потомок. При этом в обоих процессах изменение перменнной n для отслеживания адресного пространства, происходит независимо.

Проведем эксперимент с 5 порожденными процессами.

task2_1.c

```
#include <stdio.h>
#include <math.h>
#include <sys/resource.h>
#include <stdlib.h>

void main(int argc, char* argv[])
{
    int pid, pid1, pid2, pid3, pid4;
    //Переменная для отслеживания работы процесса
    int n=1;
```

```
    printf("Начальное значение n: %d\n", n);
    //Создание нового процесса с помощью fork(). pid = PID процесса.
    pid = fork();
    //При вызове функции fork() вернулась ошибка (-1)
    if(pid == -1)
    {
        //Вывод ошибки
        perror("fork error");
        //Выход из программы с кодом соотвествующей ошибки
        exit(1);
    }
    //PID != 0 => родительский процесс. Значение PID равно дочернему
процессу
    if(pid != 0)
    {
        //Вывод pid и ppid родителя
        printf("Значение n до изменения %i\n", n);
        n += 5;
        printf("РОДИТЕЛЬ: pid = %d, ppid = %d. Значение n: %i\n", getpid(),
getppid(), n);
        //Изменение перменной, так как процесс родитель
    }
    //PID = 0 => процесс - потомок.
    else
    {
        pid1 = fork();
        if(pid1 == -1)
        {
            perror("fork error");
            exit(1);
        }
        if (pid1 != 0)
        {
            printf("Значение n до изменения %i\n", n);
            n -= 5;
            printf("РОДИТЕЛЬ|ПОТОМОК1: pid = %d, ppid = %d. Значение n:
%i\n", getpid(), getppid(), n);
        }
        else
        {
            pid2 = fork();
            if(pid2 == -1)
            {
                perror("fork error");
                exit(1);
            }
            if(pid2 != 0)
            {
```

```
                printf("Значение n до изменения %i\n", n);
                n *= 13;
                printf("РОДИТЕЛЬ|ПОТМОК2: pid = %d, ppid = %d. Значение n:
%i\n", getpid(), getppid(), n);
            }
            else
            {
                pid3 = fork();
                if(pid3 == -1)
                {
                    perror("fork error");
                    exit(1);
                }
                if(pid3 != 0)
                {
                    printf("Значение n до изменения %i\n", n);
                    n -= 100;
                    printf("РОДИТЕЛЬ|ПОТОМОК3: pid = %d, ppid = %d. Значение
n: %i\n", getpid(), getppid(), n);
                }
                else
                {
                    pid4 = fork();
                    if(pid4 == -1)
                    {
                        perror("fork error");
                        exit(1);
                    }
                    if(pid4 != 0)
                    {
                        printf("Значение n до изменения %i\n", n);
                        n /= 2;
                        printf("РОДИТЕЛЬ|ПОТОМОК4: pid = %d, ppid = %d.
Значение n: %i\n", getpid(), getppid(), n);
                    }
                    else
                    {
                        printf("Значение n до изменения %i\n", n);
                        n += 100001;
                        printf("ПОТОМОК5: pid = %d, ppid = %d. Значение n:
%i\n", getpid(), getppid(), n);
                    }
                }
            }
        }
    }
    printf("Программа завершена\n\n");
    exit(1);
```

```
}
```

Работа программы task2_1.c

```
lera2003@Valeriya:~/OS_lab34/lb3$ ./task2_1
Начальное значение n: 1
Значение n до изменения 1
РОДИТЕЛЬ: pid = 6433, ppid = 6230. Значение n: 6
Программа завершена

Значение n до изменения 1
РОДИТЕЛЬ|ПОТОМОК1: pid = 6434, ppid = 1337. Значение n: -4
Программа завершена

Значение n до изменения 1
РОДИТЕЛЬ|ПОТМОК2: pid = 6435, ppid = 1337. Значение n: 13
Программа завершена

Значение n до изменения 1
РОДИТЕЛЬ|ПОТОМОК3: pid = 6436, ppid = 1337. Значение n: -99
Программа завершена

Значение n до изменения 1
Значение n до изменения 1
РОДИТЕЛЬ|ПОТОМОК4: pid = 6437, ppid = 1337. Значение n: 0
Программа завершена

ПОТОМОК5: pid = 6438, ppid = 6437. Значение n: 100002
Программа завершена
```

Заметим, что при добавлении пораждения потомков внутри потомков программы также выполняются последовательно и используют начальное значение n равное 1. Это подверждает, что при вычислениях адресные пространства друг друга не затрагиваются.

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

task3.c

```c
#include <stdio.h>
#include <math.h>
#include <sys/resource.h>
#include <stdlib.h>
```

```c
void main(int argc, char* argv[])
{
    int pid;
    //Переменная для отслеживания работы процесса
    //Создание нового процесса с помощью fork(). pid = PID процесса.
    pid = fork();
    //При вызове функции fork() вернулась ошибка (-1)
    if(pid == -1)
    {
        //Вывод ошибки
        perror("fork error");
        //Выход из программы с кодом соотвествующей ошибки
        exit(1);
    }
    while (1){
        //PID != 0 => родительский процесс. Значение PID равно дочернему
процессу
        if(pid != 0)
        {
            printf("РОДИТЕЛЬ: pid = %d, ppid = %d\n", getpid(), getppid());
        }
        //PID = 0 => процесс - потомок.
        else
        {
            printf("ПОТОМОК: pid = %d, ppid = %d\n", getpid(), getppid());
        }
    }
    printf("Программа завершена\n\n");
    exit(1);
}
```

Работа программы task3.c

```
РОДИТЕЛЬ: pid = 6670, ppid = 6230
ПОТОМОК: pid = 6671, ppid = 6670
РОДИТЕЛЬ: pid = 6670, ppid = 6230
ПОТОМОК: pid = 6671, ppid = 6670
РОДИТЕЛЬ: pid = 6670, ppid = 6230
ПОТОМОК: pid = 6671, ppid = 6670
РОДИТЕЛЬ: pid = 6670, ppid = 6230
ПОТОМОК: pid = 6671, ppid = 6670
РОДИТЕЛЬ: pid = 6670, ppid = 6230
ПОТОМОК: pid = 6671, ppid = 6670
РОДИТЕЛЬ: pid = 6670, ppid = 6230
ПОТОМОК: pid = 6671, ppid = 6670
РОДИТЕЛЬ: pid = 6670, ppid = 6230
ПОТОМОК: pid = 6671, ppid = 6670
```

```
РОДИТЕЛЬ: pid = 6670, ppid = 6230
^C
```

При замене однократных вычислений на циклы можно заметить, что процессы начинаются выполняться попеременно - конкурировать. Это связано с тем, что процессы хотят использовать один и тот же ресурс, хотят получить к нему доступ во время выполнения.

То есть если имеется совокупность процессов, каждый из которых в своем контексте содержит общий объект - процессор, но в каждый момент времени его может использовать только один из процессов. В этом случае говорят, что процессы находятся в состоянии *конкуренции* за обладание ресурсом.

Продолжим эксперимент. Теперь в каждый из процессов добавим циклы разной длины.

task3_1.c

```c
#include <stdio.h>
#include <math.h>
#include <sys/resource.h>
#include <stdlib.h>

void main(int argc, char* argv[])
{
    int pid;
    //Переменная для отслеживания работы процесса
    //Создание нового процесса с помощью fork(). pid = PID процесса.
    pid = fork();
    //При вызове функции fork() вернулась ошибка (-1)
    if(pid == -1)
    {
        //Вывод ошибки
        perror("fork error");
        //Выход из программы с кодом соотвествующей ошибки
        exit(1);
    }
    while (1){
        //PID != 0 => родительский процесс. Значение PID равно дочернему
процессу
        if(pid != 0)
        {
            printf("РОДИТЕЛЬ: pid = %d, ppid = %d\n", getpid(), getppid());
```

```
            for (int i = 0; i < 10; i++) {
                continue;
            }
        }
        //PID = 0 => процесс - потомок.
        else
        {
            printf("ПОТОМОК: pid = %d, ppid = %d\n", getpid(), getppid());
            for (int j = 0; j < 1000; j++) {
                continue;
            }
        }
    }
    printf("Программа завершена\n\n");
    exit(1);
}
```

Результат работы task3_1.c

```
РОДИТЕЛЬ: PID = 6744, PPID = 6230
РОДИТЕЛЬ: PID = 6744, PPID = 6230
РОДИТЕЛЬ: PID = 6744, PPID = 6230
РОДИТЕЛЬ: PID = 6744, PPID = 6230
РОДИТЕЛЬ: PID = 6744, PPID = 6230
РОДИТЕЛЬ: PID = 6744, PPID = 6230
РОДИТЕЛЬ: PID = 6744, PPID = 6230
РОДИТЕЛЬ: PID = 6744, PPID = 6230
РОДИТЕЛЬ: PID = 6744, PPID = 6230
РОДИТЕЛЬ: PID = 6744, PPID = 6230
РОДИТЕЛЬ: PID = 6744, PPID = 6230
РОДИТЕЛЬ: PID = 6744, PPID = 6230
ПОТОМОК: PID = 6745, PPID = 6744
РОДИТЕЛЬ: PID = 6744, PPID = 6230
РОДИТЕЛЬ: PID = 6744, PPID = 6230
РОДИТЕЛЬ: PID = 6744, PPID = 6230
РОДИТЕЛЬ: PID = 6744, PPID = 6230
РОДИТЕЛЬ: PID = 6744, PPID = 6230
РОДИТЕЛЬ: PID = 6744, PPID = 6230
РОДИТЕЛЬ: PID = 6744, PPID = 6230
РОДИТЕЛЬ: PID = 6744, PPID = 6230
ПОТОМОК: PID = 6745, PPID = 6744
РОДИТЕЛЬ: PID = 6744, PPID = 6230
РОДИТЕЛЬ: PID = 6744, PPID = 6230
```

```
РОДИТЕЛЬ: PID = 6744, PPID = 6230
РОДИТЕЛЬ: PID = 6744, PPID = 6230
РОДИТЕЛЬ: PID = 6744, PPID = 6230
РОДИТЕЛЬ: PID = 6744, PPID = 6230
РОДИТЕЛЬ: PID = 6744, PPID = 6230
РОДИТЕЛЬ: PID = 6744, PPID = 6230
РОДИТЕЛЬ: PID = 6744, PPID = 6230
ПОТОМОК: PID = 6745, PPID = 6744
```

Заметим, что цикл у процесса-потомка больше, чем у родителя. Процесс родитель появляется чаще на экране, что свидетельствует о том, что из-за меньшего числа операций процесс выполняется быстрее. НО борьба за процессорный ресурс продолжается.

4. Измените процедуру планирования и повторите эксперимент.

task4.c

```c
#include <stdio.h>
#include <math.h>
#include <sys/resource.h>
#include <stdlib.h>
#include <stdio.h>
#include <sched.h>

void main(int argc, char* argv[])
{
    // Параметры диспетчеризации определены в структуре
    struct sched_param shdprm;
    int pid;
    //Создание нового процесса с помощью fork(). pid = PID процесса.
    pid = fork();
    //При вызове функции fork() вернулась ошибка (-1)
    if(pid == -1)
    {
        //Вывод ошибки
        perror("fork error");
        //Выход из программы с кодом соотвествующей ошибки
        exit(1);
    }
    // Этот приоритет задается целым числом, при этом чем выше значение,
    // тем выше приоритет потока при планировании.
    // Создаваемые потоки получают этот приоритет.
    shdprm.sched_priority = 50;
    //  устанавливает алгоритм и параметры планирования процесса с номером
pid.
```

```
    // Если pid равен нулю, то будет задан алгоритм вызывающего процесса.
    if (sched_setscheduler (0, SCHED_FIFO, &shdprm) == -1) {
        perror ("SCHED_SETSCHEDULER");
    }
    printf ("Текущая политика планирования для текущего процесса: ");
    // получает алгоритм диспетчеризации процесса с номером pid.
    // Если pid равен нулю, то возвращается алгоритм планирования
вызывающего 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");
    }
    while (1){
        //PID != 0 => родительский процесс. Значение PID равно дочернему
процессу
        if(pid != 0)
        {
            printf("РОДИТЕЛЬ: pid = %d, ppid = %d\n", getpid(), getppid());
            for (int i = 0; i < 10; i++) {
                continue;
            }
        }
        //PID = 0 => процесс - потомок.
        else
        {
            printf("ПОТОМОК: pid = %d, ppid = %d\n", getpid(), getppid());
            for (int j = 0; j < 1000; j++) {
                continue;
            }
        }
    }
    printf("Программа завершена\n\n");
    exit(1);
}
```

Результат работы программы task4.c

```
РОДИТЕЛЬ: PID = 6796, PPID = 6230
РОДИТЕЛЬ: PID = 6796, PPID = 6230
ПОТОМОК: PID = 6797, PPID = 6796
РОДИТЕЛЬ: PID = 6796, PPID = 6230
РОДИТЕЛЬ: PID = 6796, PPID = 6230
РОДИТЕЛЬ: PID = 6796, PPID = 6230
ПОТОМОК: PID = 6797, PPID = 6796
РОДИТЕЛЬ: PID = 6796, PPID = 6230
РОДИТЕЛЬ: PID = 6796, PPID = 6230
РОДИТЕЛЬ: PID = 6796, PPID = 6230
ПОТОМОК: PID = 6797, PPID = 6796
РОДИТЕЛЬ: PID = 6796, PPID = 6230
РОДИТЕЛЬ: PID = 6796, PPID = 6230
ПОТОМОК: PID = 6797, PPID = 6796
РОДИТЕЛЬ: PID = 6796, PPID = 6230
РОДИТЕЛЬ: PID = 6796, PPID = 6230
```

Была изменена процедура планирования на FIFO. Процессы, работающие согласно алгоритму SCHED_FIFO подчиняются следующим правилам: процесс с алгоритмом SCHED_FIFO, приостановленный другим процессом с большим приоритетом, останется в начале очереди процессов с равным приоритетом, и его исполнение будет продолжено сразу после того, как закончатся процессы с большими приоритетами. Когда процесс с алгоритмом SCHED_FIFO готов к работе, он помещается в конец очереди процессов с тем же приоритетом.

Изначальная процедура планирования SCHED_OTHER – это используемый по умолчанию алгоритм со стандартным разделением времени, с которым работает большинство процессов.

При изменении процедуры планирования процессы продолжают борьбу за ресурс.

5. Разработайте программы родителя и потомка с размещением в файлах father.c и son.c

father5.c

```c
#include <stdio.h>

int main()
{
     int pid, ppid, status;
     pid=getpid();
     ppid=getppid();
     printf("\n\nFATHER PARAM: pid=%i ppid=%i\n", pid,ppid);
     if(fork()==0){
        //Заменяет текущий образ процесса новым образом процесса
        execl("son5","son5", NULL);
     }
     //x - Отсоединённые от терминала
     //f - Показать дерево процессов с родителями
     //system() выполняет команды, указанные в string, вызывая в свою
очередь команду /bin/sh -c string
     system("ps xf > file.txt");
     //Функция wait приостанавливает выполнение текущего процесса до тех
пор, пока дочерний процесс не завершится
     wait(&status);
     printf("\n\nChild proccess is finished with status %d\n\n", status);
     return 0;
}
```

son5.c

```c
#include <stdio.h>
int main()
{
    int pid,ppid;
    pid=getpid();
    ppid=getppid();
    printf("\n\nSON PARAMS: pid=%i ppid=%i\n\n",pid,ppid);
    //переход в режим ожидания на указанное количество секунд
    sleep(15);
    return 0; // статус завершения 0
}
```

```
int execl(const char *path, const char *arg, ...);
```

Параметр *const char *arg* и аналогичные записи в функциях execl, execlp, и execle подразумевают параметры *arg0*, *arg1*, ..., *argn*. Все вместе они описывают один или нескольких указателей на строки, заканчивающиеся NULL, которые представляют собой список параметров, доступных исполняемой программе. Первый параметр, по соглашению,

должен указывать на имя, ассоциированное с файлом, который надо исполнить. Список параметров *должен* заканчиваться NULL.

Родительский процесс с исходным кодом в файле father5.c порождает процесс-потомок с помощью функции fork(). Затем, с помощью функции execl("son5","son5",NULL); запускается исполняемый файл son5, выполнение начинается с точки входа – функции main. При этом фиксируются идентификаторы запущенных процессов, а также состояние таблицы процессов в файле file.txt. Родительский процесс дожидается выполнения потомка с помощью команды wait(&status), а статус завершения этого процесса записывается по адресу &status.

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

Результат работы father5.c

```
LERA2003@VALERIYA:~/OS_LAB34/LB3$ ./FATHER5
FATHER PARAM: PID=6864 PPID=6230
SON PARAMS: PID=6865 PPID=6864
CHILD PROCCESS IS FINISHED WITH STATUS 0
```

file.txt

```
6162 PTS/3    Ss     0:00      \_ BASH
 6227 PTS/3    S+     0:00          \_ /BIN/BASH ./WRITE.SH
 6229 PTS/3    S+     0:02              \_ SCRIPT -F SESSION.LOG
 6230 PTS/11   Ss     0:00                  \_ BASH -I
 6396 PTS/11   T      0:00                      \_ NANO TASK2_1.C
 6864 PTS/11   S+     0:00                      \_ ./FATHER5
 6865 PTS/11   S+     0:00                          \_ SON5
 6866 PTS/11   S+     0:00                          \_ SH -C PS XF > FILE.TXT
 6867 PTS/11   R+     0:00                              \_ PS XF
```

Содержимое файла следующее:

PID — идентификатор процесс

TTY — терминал, с которым связан данный процес

STAT — состояние, в котором на данный момент находится процессродитель

TIME — процессорное время, занятое этим процессом

COMMAND — команда, запустившая данный процесс-отец Состояния **STAT**, представленные выше:

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

s : лидер сессии

R : процесс выполняется в данный момент

+: выполняется на переднем плане, то есть это не фоновый процесс.

Процесс 6230 – ожидает и является лидером сессии.

Процессы 6864, 6865, 6866 – ожидают, являются порожденными процессы и выполняются на переднем плане, не фоновые.

Процесс 6867 – процесс выполняется в данный момент на переднем плане, не фоновый.

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

Результат работы father5.c в фоновом режиме

```
lera2003@Valeriya:~/OS_lab34/lb3$ ./father5&
[2] 6901
lera2003@Valeriya:~/OS_lab34/lb3$
FATHER PARAM: PID=6901 PPID=6230
SON PARAMS: PID=6902 PPID=6901
Child proccess is finished with status 0
```

file.txt

```
6162 pts/3    Ss     0:00      \_ bash
 6227 pts/3    S+     0:00         \_ /bin/bash ./write.sh
 6229 pts/3    S+     0:02            \_ script -f session.log
 6230 pts/11   Ss+    0:00               \_ bash -i
 6396 pts/11   T      0:00                  \_ nano task2_1.c
```

```
6901 pts/11   S      0:00                  \_ ./father5
6902 pts/11   S      0:00                     \_ son5
6903 pts/11   S      0:00                     \_ sh -c ps xf > file.txt
6904 pts/11   R      0:00                        \_ ps xf
```

Процесс 6230 – ожидает и является лидером сессии, выполняется на переднем плане.

Процессы 6901, 6902, 6903 – ожидают, являются порожденными процессы и выполняются в фоне.

Процесс 6904 – процесс выполняется в данный момент в фоном режиме.

Запуск программы father.out в фоновом режиме позволяет запускать с терминала новые процессы, не дожидаясь завершения father.out.

Командный интерпретатор (в данном случае bash) запускает программу ./father, «распараллеливает» процессы и порождает son. Программа запускается в фоновом режиме, а параллельно ей — команда ps -xf.

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

На втором этапе создания процесса осуществляется заполнение содержимого контекста, загрузка исполняемого кода новой программы, коррекция дескриптора и запуск новой программы на исполнение. Используется при этом какая-либо из функций семейства exec().

Функции семейства exec() имеют следующие прототипы:

```
int execlp(const char *file,const char *arg0,…const char *argN,(char *)NULL );
int execvp(const char *file, char *argv[]); int execl(const char *path,const char *arg0,…const char *argN, (char *)NULL );
int execv(const char *path, char *argv[]); int execle(const char *path, const char *arg0,…const char *argN,(char *)NULL, char *envp[]);
int execve(const char *path, char *argv[], char *envp[])
```

и отличаются принимаемыми аргументами, на что указывает суффикс в названии. Суффиксы l, v, p, e , а также их сочетания в именах функций определяют формат и объем аргументов, а также каталоги, в которых нужно искать загружаемую программу:

l - execl(), execlp(), execle()

Const char *arg и последующие многоточия можно рассматривать как arg0, arg1, ..., argn. Вместе они описывают список из одного или нескольких указателей на строки, заканчивающиеся нулем, которые представляют список аргументов, доступных выполняемой программе. Первый аргумент, по соглашению, должен указывать на имя файла, связанное с выполняемым файлом. Список аргументов должен заканчиваться нулевым указателем, и, поскольку это переменные функции, этот указатель должен быть приведен (char *) к нулю.

v - execv(), execvp(), execvpe()

Аргумент char *const argv[] - это массив указателей на строки, заканчивающиеся нулем, которые представляют список аргументов, доступных новой программе. Первый аргумент, по соглашению, должен указывать на имя файла, связанное с выполняемым файлом. Массив указателей должен завершаться нулевым указателем.

e - execle(), execvpe()

Среда нового образа процесса задается с помощью аргумента envp. Аргумент envp представляет собой массив указателей на строки, заканчивающиеся нулем, и должен завершаться нулевым указателем.

Все остальные функции exec() (которые не включают 'e' в суффикс) берут среду для нового образа процесса из внешней переменной environ в вызывающем процессе.

p - execlp(), execvp(), execvpe()

Эти функции дублируют действия командной строки при поиске исполняемого файла, если указанное имя файла не содержит символа косой

черты (/). Файл ищется в списке путей к каталогу, разделенных двоеточием, указанном в переменной окружения PATH. Если эта переменная не определена, список путей по умолчанию представляет собой список, включающий каталоги, возвращаемые confstr(_CS_PATH) (который обычно возвращает значение "/bin:/usr/bin") и, возможно, также текущий рабочий каталог.

Создадим программу для создания процессов с использованием различных функций семейства exec().

task8.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 = 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, (char *)NULL, env);
             break;
        }
    }
}
```

Выполним программу с различными ключами. Соответственно:

```
1 - execel
2 - execlp
3 - execle
4 - execv
5 - execvp
6 - execvpe
```

Работа программы task8.c с различными ключами

```
lera2003@Valeriya:~/OS_lab34/lb3$ ./task8 3
lera2003@Valeriya:~/OS_lab34/lb3$ total 44764
-rwxrwxr-x 1 lera2003 lera2003    8976 Apr 20 08:23 father5
-rw-rw-r-- 1 lera2003 lera2003     986 Apr 20 08:23 father5.c
-rw-rw-r-- 1 lera2003 lera2003    6743 Apr 20 08:35 file.txt
-rw-rw-r-- 1 lera2003 lera2003 45656300 Apr 20 08:46 session.log
-rwxrwxr-x 1 lera2003 lera2003    8760 Apr 20 08:23 son5
-rw-rw-r-- 1 lera2003 lera2003     328 Apr 20 08:23 son5.c
-rwxrwxr-x 1 lera2003 lera2003    8912 Apr 20 06:59 task1
-rw-rw-r-- 1 lera2003 lera2003    1401 Apr 20 06:59 task1.c
-rwxrwxr-x 1 lera2003 lera2003    8912 Apr 20 07:13 task1_2
-rw-rw-r-- 1 lera2003 lera2003    1661 Apr 20 07:12 task1_2.c
-rwxrwxr-x 1 lera2003 lera2003    8912 Apr 20 07:21 task2
-rw-rw-r-- 1 lera2003 lera2003    1692 Apr 20 07:21 task2.c
-rwxrwxr-x 1 lera2003 lera2003    8912 Apr 20 07:29 task2_1
-rw-rw-r-- 1 lera2003 lera2003    3655 Apr 20 07:29 task2_1.c
-rwxrwxr-x 1 lera2003 lera2003    8864 Apr 20 07:42 task3
-rw-rw-r-- 1 lera2003 lera2003    1055 Apr 20 07:42 task3.c
-rwxrwxr-x 1 lera2003 lera2003    8864 Apr 20 07:59 task3_1
-rw-rw-r-- 1 lera2003 lera2003    1229 Apr 20 07:59 task3_1.c
-rwxrwxr-x 1 lera2003 lera2003    9040 Apr 20 08:08 task4
-rw-rw-r-- 1 lera2003 lera2003    1938 Apr 20 08:08 task4.c
-rwxrwxr-x 1 lera2003 lera2003    8976 Apr 20 08:45 task8
-rw-rw-r-- 1 lera2003 lera2003     887 Apr 20 08:45 task8.c
-rwxrwxr-x 1 lera2003 lera2003     122 Apr 20 06:56 write.sh
lera2003@Valeriya:~/OS_lab34/lb3$ ./task8 1
lera2003@Valeriya:~/OS_lab34/lb3$ итого 44764
```

```
-RWXRWXR-X 1 LERA2003 LERA2003    8976 АПР 20 08:23 FATHER5
-RW-RW-R-- 1 LERA2003 LERA2003     986 АПР 20 08:23 FATHER5.C
-RW-RW-R-- 1 LERA2003 LERA2003    6743 АПР 20 08:35 FILE.TXT
-RW-RW-R-- 1 LERA2003 LERA2003 45657955 АПР 20 08:46 SESSION.LOG
-RWXRWXR-X 1 LERA2003 LERA2003    8760 АПР 20 08:23 SON5
-RW-RW-R-- 1 LERA2003 LERA2003     328 АПР 20 08:23 SON5.C
-RWXRWXR-X 1 LERA2003 LERA2003    8912 АПР 20 06:59 TASK1
-RWXRWXR-X 1 LERA2003 LERA2003    8912 АПР 20 07:13 TASK1_2
-RW-RW-R-- 1 LERA2003 LERA2003    1661 АПР 20 07:12 TASK1_2.C
-RW-RW-R-- 1 LERA2003 LERA2003    1401 АПР 20 06:59 TASK1.C
-RWXRWXR-X 1 LERA2003 LERA2003    8912 АПР 20 07:21 TASK2
-RWXRWXR-X 1 LERA2003 LERA2003    8912 АПР 20 07:29 TASK2_1
-RW-RW-R-- 1 LERA2003 LERA2003    3655 АПР 20 07:29 TASK2_1.C
-RW-RW-R-- 1 LERA2003 LERA2003    1692 АПР 20 07:21 TASK2.C
-RWXRWXR-X 1 LERA2003 LERA2003    8864 АПР 20 07:42 TASK3
-RWXRWXR-X 1 LERA2003 LERA2003    8864 АПР 20 07:59 TASK3_1
-RW-RW-R-- 1 LERA2003 LERA2003    1229 АПР 20 07:59 TASK3_1.C
-RW-RW-R-- 1 LERA2003 LERA2003    1055 АПР 20 07:42 TASK3.C
-RWXRWXR-X 1 LERA2003 LERA2003    9040 АПР 20 08:08 TASK4
-RW-RW-R-- 1 LERA2003 LERA2003    1938 АПР 20 08:08 TASK4.C
-RWXRWXR-X 1 LERA2003 LERA2003    8976 АПР 20 08:45 TASK8
-RW-RW-R-- 1 LERA2003 LERA2003     887 АПР 20 08:45 TASK8.C
-RWXRWXR-X 1 LERA2003 LERA2003     122 АПР 20 06:56 WRITE.SH
```

Результат работы программы одинаковый, поэтому проанализируем разницу передаваемых аргументов.

```
1 - execl("/bin/ls", "/bin/ls", "-l", (char *)NULL)
```

l (список) – аргументы командной строки передаются в форме списка arg0, arg1.... argN, NULL. Эту форму используют, если количество аргументов известно.

```
2 - execlp("ls", "ls", "-l", (char *)NULL)
```

l – колчиество аргументов известно.

p (path) – обозначенный по имени файл ищется не только в текущем каталоге, но и в каталогах, определенных переменной среды PATH.

char* path = "/bin/ls";

```
3 - execle("/bin/ls", "ls", "-l", (char *)NULL, env)
```

l – колчиество аргументов известно.

e (среда или окружение) – функция ожидает список переменных окружения в виде вектора (envp []).

```
4 - execv("/bin/ls", args)
```

v (vector) – аргументы командной строки передаются в форме вектора argv[]. Отдельные аргументы адресуются через argv [0], argv [1]... argv [n]. Последний аргумент (argv [n]) должен быть указателем NULL.

char *args[] = {"ls", "-l", NULL }

```
5 - execvp("ls", args)
```

v – передача аргументов командной строки в виде вектора.

p – поиск файла и в каталогах, определенных переменной среды PATH.

```
6 -  execvpe("ls", args, (char *)NULL, env)
```

v – передача аргументов командной строки в виде вектора.

p – поиск файла и в каталогах, определенных переменной среды PATH.

e – список переменных окружения в виде вектора.

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

Прототипы функций рассматримаемого семейства:

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
int waitid(idtype_t idtype, id_t id, siginfo_t * infop , int options );
```

При успешном завершении системного вызова fork процессы порождающий и порожденный равноправно сосуществуют в системе. Они выполняются, разделяя процессорное время, конкурируя за ресурсы на основе своих приоритетов. Выполнение порождающего процесса может

быть приостановлено до завершения потомка системным вызовом wait. Системный вызов wait возвращает родителю идентификатор того потомка, который завершился первым после последнего обращения к wait. Если у родителя несколько потомков, то чтобы узнать о завершении каждого из них, нужно выполнить несколько системных вызовов wait с проверкой их возвращаемых значений. Если процесс не имеет потомков, wait возвращает код (-1).

Системный вызов waitpid() блокирует выполнение текущего процесса до тех пор, пока либо не завершится порожденный им процесс, определяемый значением параметра pid, либо пока текущий процесс не получит сигнал, для которого установлена реакция по умолчанию "завершить процесс" или реакция обработки пользовательской функцией. Если порожденный процесс, заданный параметром pid, уже в стадии завершения к моменту системного вызова управление немедленно возвращается без блокирования текущего процесса.

Параметр pid определяет порожденный процесс, завершения которого дожидается процесс-родитель, следующим образом:

- если pid > 0 ожидаем завершения процесса с идентификатором pid;
- если pid = 0, то ожидаем завершения любого порожденного процесса в группе, к которой принадлежит процесс-родитель;

- если pid = -1, то ожидаем завершения любого порожденного процесса;

- если pid < 0, но не (-1), то ожидаем завершения любого порожденного процесса из группы, идентификатор которой равен абсолютному значению параметра pid.

Функции wait и waitpid сохраняют информацию о статусе в переменной, на которую указывает status, если status не равен NULL.

father9.c

```c
#include <stdio.h>
#include <sys/types.h>
#include <wait.h>
int main()
{
	int i, pid[4], ppid, status, result;
	pid[0]=getpid();
	ppid=getppid();
	printf("\nFATHER 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("\n%d) Child proccess with pid = %d is finished with
status %d\n", i,result, status);
	}
	return 0;
}
```

son*.c

```c
#include <stdio.h>
int main()
{
    int pid,ppid;
    pid=getpid();
    ppid=getppid();
    printf("\n\nSON PARAMS: pid=%i ppid=%i\n\n",pid,ppid);
    //переход в режим ожидания на указанное количество секунд
    sleep(15);
    return 0; // статус завершения 0
}
```

Результат работы программы father9.c

```
lera2003@Valeriya:~/OS_lab34/lb3$ ./father9
FATHER PARAMS: pid=7069 ppid=6230
SON PARAMS: pid=7072 ppid=7069
SON PARAMS: pid=7071 ppid=7069
SON PARAMS: pid=7070 ppid=7069
1) Child proccess with pid = 7070 is finished with status 0
```

```
2) CHILD PROCCESS WITH PID = 7071 IS FINISHED WITH STATUS 0
3) CHILD PROCCESS WITH PID = 7072 IS FINISHED WITH STATUS 0
```

file.txt

```
7069 PTS/11   S+    0:00                  \_ ./FATHER9
 7070 PTS/11   S+    0:00                      \_ SON1
 7071 PTS/11   S+    0:00                      \_ SON2
 7072 PTS/11   S+    0:00                      \_ SON3
 7073 PTS/11   S+    0:00                      \_ SH -C PS XF > FILE.TXT
 7074 PTS/11   R+    0:00                         \_ PS XF
```

Создается 3 процесса потомка и с помощью waitpid() ожидается их завершение. Системный вызов waitpid() блокирует выполнение текущего процесса до тех пор, пока либо не завершится порожденный им процесс, определяемый значением параметра pid, либо пока текущий процесс не получит сигнал, для которого установлена реакция по умолчанию "завершить процесс" или реакция обработки пользовательской функцией. Если порожденный процесс, заданный параметром pid, уже в стадии завершения к моменту системного вызова управление немедленно возвращается без блокирования текущего процесса.

10. Проанализируйте очередность исполнения процессов.

Сегодня в Unix подобных ОС, в частности в Linux, и других ОС, следующих стандарту POSIX, поддерживаются три базовые политики планирования: SCHED_FIFO, SCHED_RR, и SCHED_OTHER: одна для обычных процессов и две для процессов «реального» времени. Их реализация обеспечивается ядром, а точнее, планировщиком. Каждому процессу присваивается статический приоритет sched_priority, который можно изменить только при помощи системных вызовов. Ядро хранит в памяти списки всех работающих процессов для каждого возможного значения sched_priority, а это значение может находиться в определенном интервале, заданном для данной конкретной реализации ОС. Для того, чтобы определить, какой процесс будет выполняться следующим,

планировщик ищет непустой список (очередь) с наибольшим статическим приоритетом и запускает первый процесс из этого списка.

Алгоритм планирования определяет, как процесс будет добавлен в список-очередь с тем же статическим приоритетом, и как он будет перемещаться внутри этого списка.

Для каждой политики – свои значения и диапазон статических приоритетов, которые могут зависеть от конкретного типа системы, с которой вы работаете. Например, для Linux Debian статический приоритет процессов с алгоритмом SCHED_OTHER равен нулю, а статические приоритеты процессов с алгоритмами SCHED_FIFO и SCHED_RR могут находиться в диапазоне от 1 до 99. Статический приоритет, больший, чем 0, может быть установлен только у суперпользовательских процессов, то есть только эти процессы могут иметь алгоритм планировщика SCHED_FIFO или SCHED_RR.

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

1) SCHED_FIFO: планировщик FIFO (First In-First Out)

Алгоритм SCHED_FIFO можно использовать только со значениями статического приоритета, большими нуля. Это означает, что если процесс с алгоритмом SCHED_FIFO готов к работе, то он сразу запустится, а все обычные процессы с алгоритмом SCHED_OTHERбудут приостановлены. SCHED_FIFO - это простой алгоритм без квантования времени. Процессы, работающие согласно алгоритму SCHED_FIFO подчиняются следующим

правилам: процесс с алгоритмом SCHED_FIFO, приостановленный другим процессом с большим приоритетом, останется в начале очереди процессов с равным приоритетом, и его исполнение будет продолжено сразу после того, как закончатся процессы с большими приоритетами.

2) SCHED_RR: циклический алгоритм планирования

Все, относящееся к алгоритму SCHED_FIFO, справедливо и для SCHED_RR за исключением того, что каждому процессу разрешено работать непрерывно не дольше некоторого времени, называемого квантом. Если процесс с алгоритмом SCHED_RR работал столько же или дольше, чем квант, то он помещается в конец очереди процессов с тем же приоритетом. Процесс с алгоритмом SCHED_RR, приостановленный процессом с большим приоритетом, возобновляя работу, использует остаток своего кванта. Длину этого кванта можно узнать, вызвав функцию sched_rr_get_interval.

3) SCHED_OTHER: стандартный алгоритм планировщика с разделением времени

Алгоритм SCHED_OTHER можно использовать только со значениями статического приоритета, равными нулю. SCHED_OTHER – это стандартный алгоритм планирования Linux с разделением времени, предназначенный для процессов, не требующих специальных механизмов реального времени со статическими приоритетами. Порядок предоставления процессорного времени процессам со статическим приоритетом, равным нулю, основывается на динамических приоритетах, существующих только внутри этого списка. Динамический приоритет основан на уровне nice (установленном при помощи системных вызовов nice или setpriority) и увеличивается с каждым квантом времени, при котором процесс был готов к работе, но ему было отказано в этом планировщиком. Это приводит к тому, что, рано или поздно, всем процессам с приоритетом SCHED_OTHER выделяется процессорное время.

10.1. Очередность исполнения процессов, порожденных вложенными вызовами fork().

task10_11.c

```c
#include <stdio.h>
#include <sched.h>
int main (void) {
    struct sched_param shdprm; // Значения параметров планирования
    printf ("диапазоны приоритетов для разных политик планирования\n");
    printf ("SCHED_FIFO : от %d до %d\n", sched_get_priority_min
(SCHED_FIFO), sched_get_priority_max (SCHED_FIFO));
    printf ("SCHED_RR : от %d до %d\n", sched_get_priority_min (SCHED_RR),
sched_get_priority_max (SCHED_RR));
    printf ("SCHED_OTHER: от %d до %d\n", sched_get_priority_min
(SCHED_OTHER), sched_get_priority_max (SCHED_OTHER));
    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;
}
```

Результат работы task10_11.c

```
LERA2003@VALERIYA:~/OS_LAB34/LB3$ ./TASK10_11
ДИАПАЗОНЫ ПРИОРИТЕТОВ ДЛЯ РАЗНЫХ ПОЛИТИК ПЛАНИРОВАНИЯ
SCHED_FIFO : ОТ 1 ДО 99
```

```
SCHED_RR : от 1 до 99
SCHED_OTHER: от 0 до 0
Текущая политика планирования для текущего процесса: SCHED_OTHER
Текущий приоритет текущего процесса: 0
```

С помощью программы task10_11.c была выяснена текущая политика планирования и диапазоны приоритетов для разных политик.

SCHED_OTHER - порядок предоставления процессорного времени процессам со статическим приоритетом, равным нулю, основывается на динамических приоритетах, существующих только внутри этого списка. Динамический приоритет основан на уровне nice и увеличивается с каждым квантом времени, при котором процесс был готов к работе, но ему было отказано в этом планировщиком.

Теперь согласно заданию, определим очередность исполнения процессов, порожденных вложенными вызовами fork().

Очередность исполнения процессов, порожденных вложенными вызовами fork(), зависит от того, какой процесс первым завершит свою работу.

При вызове fork() создается точная копия родительского процесса, и оба процесса начинают исполняться параллельно. Это означает, что оба процесса будут продолжать работу одновременно вплоть до тех пор, пока один из них не завершится раньше другого.

Если родительский процесс завершится первым, то его дочерний процесс станет сиротой и будет передан под опеку другого процесса (init, в большинстве случаев), который вызовет wait() для завершения дочернего процесса.

Если же дочерний процесс завершится первым, то родительский процесс продолжит работу в своем обычном режиме, не обращая внимания на то, что дочерний процесс завершился.

Таким образом, очередность исполнения процессов, порожденных вложенными вызовами fork(), зависит от того, какой процесс первым завершит свою работу, и может быть не определена заранее.

Напишем программный код, который позволит сделать такой анализ.

task10_12.c

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

int main() {
    pid_t pid;

    printf("Parent process. Parent's PID: %d\n", getpid());

    pid = fork();

    if (pid == 0) {
        printf("Child process 1. PID: %d, Parent's PID: %d\n", getpid(),
getppid());

        pid = fork();

        if (pid == 0) {
            printf("Child process 2. PID: %d, Parent's PID: %d\n", getpid(),
getppid());
        } else {
            wait(NULL);
            printf("Child process 1 finished. PID: %d, Parent's PID: %d\n",
getpid(), getppid());
        }
    } else {
        wait(NULL);
        printf("Parent process finished. PID: %d\n", getpid());
    }

    return 0;
}
```

Код сначала выведет идентификатор родительского процесса, затем создаст первый дочерний процесс и выведет его идентификатор и идентификатор родительского процесса. Затем он создаст еще один

дочерний процесс внутри первого дочернего процесса и выведет его идентификатор и идентификатор родительского процесса.

Когда процесс 2 завершит свою работу, процесс 1 будет завершаться, и родительский процесс продолжит работу. Когда родительский процесс завершится, программа закончит работу.

Результат работы программы task10_12.c

```
lera2003@Valeriya:~/OS_lab34/lb3$ ./task10_12
Parent process. Parent's PID: 7153
Child process 1. PID: 7154, Parent's PID: 7153
Child process 2. PID: 7155, Parent's PID: 7154
Child process 1 finished. PID: 7154, Parent's PID: 7153
Parent process finished. PID: 7153
```

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

Приведем прототипы системных функций, позволяющих считывать и устанавливать политики планирования и соответствующие им параметры:

```
int sched_setscheduler(pid_t pid, int policy, const struct sched_param *p); int sched_getscheduler(pid_t pid);
struct sched_param
{ …
 int sched_priority;
... };
```

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

father10_2.c

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

```c
int main (void) {
    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);
    //Этот приоритет задается целым числом, при этом чем выше значение,
    //тем выше приоритет потока при планировании.
    //Создаваемые потоки получают этот приоритет.
    shdprm.sched_priority = 50;
    //устанавливает или получает алгоритм планировщика (и его параметры)
    //SCHED_RR: циклический алгоритм планировщика
    if (sched_setscheduler(0, SCHED_RR, &shdprm) == -1)
    {
        perror ("SCHED_SETSCHEDULER");
    }
    if((pid1=fork()) == 0){
        execl("son10_21", "son10_21", NULL);
    }
    if((pid2=fork()) == 0){
        execl("son10_22", "son10_22", NULL);
    }
    //получение алгоритма планирования
    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;
}
```

son10_21.c (аналогично son10_22.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <sched.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;
}
```

Результат работы father10_2.c

```
lera2003@Valeriya:~/OS_lab34/lb3$ sudo ./father10_2
[sudo] пароль для lera2003:
FATHER PARAMS: pid=2568 ppid=2567
SCHED_RR
Текущий приоритет текущего процесса: 50
SON_1 PARAMS: pid=2569 ppid=1340
SON_1: Текущая политика планирования для текущего процесса: SCHED_RR
```

```
SON_1: Текущий приоритет текущего процесса: 50
SON_2 PARAMS: pid=2570 ppid=1340
SON_2: Текущая политика планирования для текущего процесса: SCHED_RR
SON_2: Текущий приоритет текущего процесса: 50
```

Политика планирования была изменена на SCHED_RR, с помощью функции с шаблоном scheduler.

Все приоритеты равны 50.

SCHED_RR - политика в которой, каждому процессу разрешено

работать непрерывно не дольше некоторого времени. Если процесс с алгоритмом SCHED_RR работал столько же или дольше, чем квант, то он помещается в конец очереди процессов с тем же приоритетом. Процесс с алгоритмом SCHED_RR, приостановленный процессом с большим приоритетом, возобновляя работу, использует остаток своего кванта.

Таким образом, из результатов следует, что потомки наследуют политику планирования и приоритет родительского процесса.

Изменим политику планирования на FIFO и повторим эксперимент

Результат работы father10_2.c после изменения политики планирования

```
lera2003@Valeriya:~/OS_lab34/lb3$ sudo ./father10_2
FATHER PARAMS: pid=2580 ppid=2579
SCHED_FIFO
Текущий приоритет текущего процесса: 50
SON_1 PARAMS: pid=2581 ppid=1340
SON_1: Текущая политика планирования для текущего процесса: SCHED_FIFO
SON_1: Текущий приоритет текущего процесса: 50
SON_2 PARAMS: pid=2582 ppid=1340
SON_2: Текущая политика планирования для текущего процесса: SCHED_FIFO
SON_2: Текущий приоритет текущего процесса: 50
```

Политика планирования была изменена на SCHED_FIFO (First in first out), с помощью функции с шаблоном scheduler.

Как видно из вывода программы у всех процессов одинаковый приоритет равный 50.

10.3. Поменяйте порядок очереди в RR-процедуре.

Создадим программу, позволяющую вручную задавать приоритеты.

father10_3.c

```c
#include <stdio.h>
#include <sched.h>
int main (void) {
    struct sched_param shdprm; // значения параметров планирования
    int pid, pid1, pid2, pid3, ppid, status;
    int n, m, l, k; // переменные для задания значений приоритетов
    n=50; m=60; l=10; k=4; // заданные значения приоритетов с политикой RR
    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("son10_31", "son10_31", NULL);
    }
    if((pid2=fork()) == 0)
    {
        shdprm.sched_priority = l;
        if (sched_setscheduler (pid2, SCHED_RR, &shdprm) == -1)
            perror ("SCHED_SETSCHEDULER_2");
        execl("son10_32", "son10_32", NULL);
    }
    if((pid3=fork()) == 0)
    {
        shdprm.sched_priority = k;
        if (sched_setscheduler (pid3, SCHED_RR, &shdprm) == -1)
            perror ("SCHED_SETSCHEDULER_3");
        execl("son10_33", "son10_33", 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;
}
```

Результат работы father10_3.c

```
lera2003@Valeriya:~/OS_lab34/lb3$ sudo ./father10_3
FATHER PARAMS: pid=2700 ppid=2699
SON_1 PARAMS: pid=2701 ppid=2700
SON_1: Текущая политика планирования для текущего процесса: SCHED_RR
SON_1: Текущий приоритет текущего процесса: 60
SON_2 PARAMS: pid=2702 ppid=2700
SON_3 PARAMS: pid=2703 ppid=2700
SON_3: Текущая политика планирования для текущего процесса: SCHED_RR
SON_3: Текущий приоритет текущего процесса: 4
Процесс с pid = 2701 завершен
SON_2: Текущая политика планирования для текущего процесса: SCHED_RR
SON_2: Текущий приоритет текущего процесса: 10
Процесс с pid = 2702 завершен
Процесс с pid = 2703 завершен
```

В результате эксперимент сначала был завершен процесс с приоритетом 60 (son1), после него завершил работу son2 с приоритетом 10, самым последним завершил свою работу son3 с приоритетом 4.

Повторим эксперимент, изменив политику планирования на FIFO.

Результат работы father10_3.c после изменения политики планирования.

```
lera2003@Valeriya:~/OS_lab34/lb3$ sudo ./father10_3
FATHER PARAMS: pid=2723 ppid=2722
SON_1 PARAMS: pid=2724 ppid=2723
SON_2 PARAMS: pid=2725 ppid=2723
SON_1: Текущая политика планирования для текущего процесса: SCHED_FIFO
SON_2: Текущая политика планирования для текущего процесса: SCHED_FIFO
SON_1: Текущий приоритет текущего процесса: 60
SON_2: Текущий приоритет текущего процесса: 10
SON_3 PARAMS: pid=2726 ppid=2723
SON_3: Текущая политика планирования для текущего процесса: SCHED_FIFO
SON_3: Текущий приоритет текущего процесса: 4
Процесс с pid = 2724 завершен
Процесс с pid = 2725 завершен
Процесс с pid = 2726 завершен
```

Процесс с приоритетом 60 первым завершил работу, после него завершил работу процесс с приоритетом 10, самым последним завершил

свою работу процесс с приоритетом 4. Также наблюдается не последовательный вывод.

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

father10_4.c

```
#include <stdio.h>
#include <sched.h>
int main (void) {
    struct sched_param shdprm; // значения параметров планирования
    int pid, pid1, pid2, pid3, ppid, status;
    int n, m, l, k; // переменные для задания значений приоритетов
    n=50; m=60; l=60; k=60; // заданные значения приоритетов с политикой RR
    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_FIFO, &shdprm) == -1)
            perror ("SCHED_SETSCHEDULER_1");
        execl("son10_31", "son10_31", NULL);
    }
    if((pid2=fork()) == 0)
    {
        shdprm.sched_priority = l;
        if (sched_setscheduler (pid2, SCHED_RR, &shdprm) == -1)
            perror ("SCHED_SETSCHEDULER_2");
        execl("son10_32", "son10_32", NULL);
    }
    if((pid3=fork()) == 0)
    {
        shdprm.sched_priority = k;
        if (sched_setscheduler (pid3, SCHED_FIFO, &shdprm) == -1)
            perror ("SCHED_SETSCHEDULER_3");
        execl("son10_33", "son10_33", 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;
}
```

Результат работы father10_4.c

```
lera2003@Valeriya:~/OS_lab34/lb3$ sudo ./father10_4
FATHER PARAMS: pid=2753 ppid=2752
SON_2 PARAMS: pid=2755 ppid=2753
SON_2: Текущая политика планирования для текущего процесса: SCHED_RR
SON_2: Текущий приоритет текущего процесса: 60
SON_1 PARAMS: pid=2754 ppid=2753
SON_1: Текущая политика планирования для текущего процесса: SCHED_FIFO
SON_1: Текущий приоритет текущего процесса: 60
Процесс с pid = 2754 завершен
Процесс с pid = 2755 завершен
SON_3 PARAMS: pid=2756 ppid=2753
SON_3: Текущая политика планирования для текущего процесса: SCHED_FIFO
SON_3: Текущий приоритет текущего процесса: 60
Процесс с pid = 2756 завершен
```

Были заданы одинаковые приоритеты для первого и второго процессов с разной политикой (RR и FIFO). В результате сначала завершается процесс с той политикой которая указана у первого процесса.

Повторим эксперимент, поменяв местами политики планирования.

Результат работы father10_4.c после изменения политик планирвоания

```
FATHER PARAMS: pid=2775 ppid=2774
SON_2 PARAMS: pid=2777 ppid=2775
SON_2: Текущая политика планирования для текущего процесса: SCHED_FIFO
SON_2: Текущий приоритет текущего процесса: 60
SON_1 PARAMS: pid=2776 ppid=2775
SON_1: Текущая политика планирования для текущего процесса: SCHED_RR
SON_1: Текущий приоритет текущего процесса: 60
SON_3 PARAMS: pid=2778 ppid=2775
SON_3: Текущая политика планирования для текущего процесса: SCHED_FIFO
SON_3: Текущий приоритет текущего процесса: 60
Процесс с pid = 2776 завершен
Процесс с pid = 2777 завершен
```

```
Процесс с pid = 2778 завершен
```

Результат эксперимента остался прежним.

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

Определить величину кванта можно с помощью функции (POSIX)

```
int sched_rr_get_interval(pid_t, struct timespec *);
```

father11.c

```c
#include <stdio.h>
#include <sched.h>
#include <sys/mman.h>
int main (void) {
        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: %g сек\n",
qp.tv_sec + qp.tv_nsec / 1000000000.0);
        else
                perror ("SCHED_RR_GET_INTERVAL");
        if((pid1=fork()) == 0)
        {
                if (sched_rr_get_interval (pid1, &qp) == 0)
                        printf ("SON: Квант процессорного времени: %g
сек\n",qp.tv_sec + qp.tv_nsec / 1000000000.0);
                execl("son11", "son11", NULL);
        }
        printf("Процесс с pid = %d завершен\n", wait(&status));
        return 0;
}
```

Результат работы father11.c

```
lera2003@Valeriya:~/OS_lab34/lb3$ sudo ./father11
FATHER PARAMS: pid=2812 ppid=2811
Квант при циклическом планировании: 0.1 сек
SON: Квант процессорного времени: 0.1 сек
```

```
SON PARAMS: PID=2813 PPID=2812
Процесс с PID = 2813 завершен
```

Таким образом, размер кванта родительского процесса составляет 0.1 сек, более того, данный размер наследуется процессом-потомком.

Теперь попробуем изменить размер кванта, изменив приоритет порождаемых процессов.

Экспериментально это можно проверить, используясистемную функцию nice():

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

Результат работы father11.c после попытки изменить величину кванта

```
lera2003@Valeriya:~/OS_lab34/lb3$ sudo ./father11_1
FATHER PARAMS: PID=2840 PPID=2839
Nice value = 4196048
Квант при циклическом планировании: 0.1 сек
SON: Квант процессорного времени: 0.1 сек
SON PARAMS: PID=2841 PPID=2840
Процесс с PID = 2841 завершен
```

После выполнения получаем те же самые значения для размера кванта.

Современные ОС linux не имеют специального механизма, который позволял бы устанавливать величину кванта процессорного времени для RR—планировщика из приложений в отличие от более старых версий, где квантом можно было управлять, регулируя параметр процесса nice. Отрицательное значение nice — квант длиннее, положительное — короче. Степень влияния значения nice на квант в разных версиях ядра была различной. Начиная с версии Linux 2.6.24, квант SCHED_RR не может быть изменен документированными средствами.

Текущая версия ядра Linux:

```
lera2003@Valeriya:~/OS_lab34/lb3$ sudo uname -sv
Linux #146~16.04.1-Ubuntu SMP Tue Apr 13 09:27:15 UTC 2021
```

Величина кванта может быть при FIFO - 0 сек, а при алгоритме OTHER — 0.016001 сек, при RR - 0.100006 сек.

12. Проанализируйте наследование на этапах fork() и exec(). Проведите эксперимент с родителем и потомками по доступу к одним и тем же файлам, открытым родителем.

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

father12.c

```c
#include <stdio.h>
#include <sched.h>
#include <sys/mman.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
void itoa(char *buf, int value) {
    sprintf(buf, "%d", value);
}
int main (void) {
    int i, pid, ppid, status;
    int fdrd, fdwr;
    char str1[10], str2[10];
    char c;
    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("Openning file");
    if ((fdwr = creat("outfile.txt",0666)) == -1)
        perror("Creating file");
    itoa(str1, fdrd);
    itoa(str2, fdwr);
    for (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("son12", "son12", str1, str2, NULL);
        }
    if (close(fdrd) != 0)
        perror("Closing file");
    for (i = 0; i < 2; i++)
        printf("Процесс с pid = %d завершен\n", wait(&status));
    return 0;
}
```

son12.c

```
#include <sched.h>
#include <sys/mman.h>
#include <fcntl.h>
int main(int argc, char *argv[])
{
    if (mlockall((MCL_CURRENT | MCL_FUTURE)) < 0)
        perror("mlockall error");
    char c;
    int pid, ppid, buf;
    int fdrd = atoi(argv[1]);
    int fdwr = atoi(argv[2]);
    pid=getpid();
    ppid=getppid();
    printf("son file decriptor = %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;
}
```

infile.txt

```
lera2003@Valeriya:~/OS_lab34/lb3$ cat infile.txt
This is lb 3, operation systems!
```

Результат работы программы task12.c

```
lera2003@Valeriya:~/OS_lab34/lb3$ sudo ./father12
son file decriptor = 3
son params: pid=2886 ppid=2885
son file decriptor = 3
son params: pid=2887 ppid=2885
pid = 2886: T
pid = 2887: H
pid = 2887: S
pid = 2886: I
pid = 2887:
pid = 2887: S
pid = 2887:
pid = 2886: I
pid = 2887: L
pid = 2886: B
pid = 2887:
pid = 2886: 3
pid = 2887: ,
pid = 2886:
pid = 2886: O
pid = 2887: P
pid = 2886: E
pid = 2887: R
pid = 2886: A
pid = 2887: T
pid = 2886: I
pid = 2887: O
pid = 2886: N
pid = 2887:
```

```
PID = 2886: S
PID = 2887: Y
PID = 2886: S
PID = 2887: T
PID = 2886: E
PID = 2887: M
PID = 2886: S
PID = 2887: !
PID = 2886:

PID = 2887:

Процесс с PID = 2887 завершен
Процесс с PID = 2886 завершен
```

outfile.txt

```
lera2003@Valeriya:~/OS_lab34/lb3$ cat outfile.txt
This is lb 3, operation systems!
```

При выполнении функции fork() ядро создает потомка как копию родительского процесса, процесс-потомок наследует от родителя:

- сегменты кода, данных и стека программы
- таблицу файлов, в которой находятся состояния флагов
- дескрипторов файла, указывающие допустимые операции над файлом.

Кроме того, в таблице файлов содержится текущая позиция указателя записи-чтения; рабочий и корневой каталоги; реальный и эффективный идентификатор пользователя и номер группы; приоритеты процесса (администратор может изменить их через nice); терминал; маску сигналов; ограничения по ресурсам; сведения о среде выполнения; разделяемые сегменты памяти.

Потомок не наследует от родителя:

- идентификатора процесса (PID, PPID);

- израсходованного времени ЦП (оно обнуляется);
- сигналов процесса-родителя, требующих ответа;
- блокированных файлов (record locking).

Убедиться в наследовании других параметров при порождении потомков можно, проанализировав вывод утилиты:

```
ps -o uid,gid,ruid,pid,ppid,pgid,tty,vsz,stat,command
```

```
UID   GID  RUID   PID  PPID  PGID TT         VSZ STAT COMMAND
   0  1000    0  2899  2163  2899 pts/11    56376 S    sudo su
   0    0    0  2900  2899  2899 pts/11    55892 S    su
   0    0    0  2901  2900  2901 pts/11    22736 S    bash
   0    0    0  2925  2901  2925 pts/11     4220 SL+  ./father12
   0    0    0  2926  2901  2925 pts/11    30432 R+   ps -o
uid,gid,ruid,pid,ppid,pgid,tty,vsz,stat,comm
   0    0    0  2927  2925  2925 pts/11     4356 SL+  son12 3 4
   0    0    0  2928  2925  2925 pts/11     4356 SL+  son12 3 4
```

То есть от родителя наследуются UID, GID, RUID, PGID, TTY и, как было показано ранее, приоритеты и политика планирования процессов.

Продолжая эксперимент с программой демонстрирующей наследование файловых дескрипторов открытых файлов и указателей на позицию при чтении и записи в файл, попробуем закрыть в одном из процессов файл с заданным дескриптором, например, fdrd в son.c:

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

Результат работы father12.c при закрытии в одном из процессов файла

```
root@Valeriya:/home/lera2003/OS_lab34/lb3# ./father12
son file decriptor = 3
```

```
SON FILE DECRIPTOR = 3
SON PARAMS: PID=2968 PPID=2967
SON PARAMS: PID=2969 PPID=2967
PID = 2968: T
PID = 2969: H
Процесс с PID = 2968 завершен
Процесс с PID = 2969 завершен
```

outfile.txt

```
root@Valeriya:/home/lera2003/OS_lab34/lb3# cat outfile.txt
TH
```

Так как один из процессов закрывает файл на чтение, запись состоит всего из 2 символов.

Вывод.

Изучен процесс порождения процессов-потомков.

Проанализирвоано владение адресным пространством каждого из процессов.

Проведено наблюдение с помощью создания циклов внутри процессов для отслеживания конкуренции процессов за процессорный ресурс.

На практике использованы функции fork() и exec() с разными суффиксами.

Проанализировано значение, возвращаемое функцией wait(&status). Системный вызов wait возвращает родителю идентификатор того потомка, который завершился первым после последнего обращения к wait.

Изучены различные политик планирвоания: SHED_FIFO, SHED_RR, SHED_OTHER.

Определина величина кванта. Проведена попытка изменить величину кванта, но из-за текущей версии Ubuntu, установленной на ноутбуке это невозможно.

Проведите эксперимент с родителем и потомками по доступу к одним и тем же файлам, открытым родителем. Процессы по очереди считывают данные из файла.

Список источников.

1. «Системное программное обеспечение. Практические вопросы разработки системных приложений. Учебное пособие» Душутина Е.В.
2. Сайт fork(2) - Справочная страница Linux (man7.org)
3. Сайт Создание процессов с помощью вызова fork(). (opennet.ru)
4. Сайт Ubuntu Manpage: Welcome
5. Сайт https://www.opennet.ru/docs/RUS/linux_parallel/node7.html