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

ОТЧЕТ

по лабораторной работе №3-4

по дисциплине «Операционные системы»

Тема: Системное программирование в ОС семейства Unix

Студентка гр. 2384 _____ Соц Е.А.

Преподаватель _____ Душутина Е.В.

Санкт-Петербург
2024

Цель работы.

Целью данной работы является изучение основных принципов управления процессами и потоками в операционных системах.

Задание.

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.

Выполнение работы.

Информация о системе:

Linux katya 6.5.0-28-generic #29~22.04.1-Ubuntu SMP PREEMPT_DYNAMIC Thu Apr 4 14:39:20 UTC 2 x86_64 x86_64 x86_64 GNU/Linux

Замечание: задания с конкуренцией запускались на одном ядре, хотя это и не совсем реально для практических целей, это позволяет теоретически лучше понять процессы планирования

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

Была написана программа task1.c, наброски которой были взяты из методических указаний:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main(){
   int pid, n;
   n = 1;
   printf("начальное число:%d\n", n);
   pid = fork();
   if(pid==-1){
       perror("fork");
       exit(1);
   }
   if(pid == 0){
       printf("new pid = %d, pprid = %d\n", getpid(), getppid());
       //здесь размещаются вычисления, выполняемые
процессом-потомком
       n *= 3;
   }
   else{
       printf("parrent pid = %d, ppid = %d\n", getpid(),
getppid());
       //здесь размещаются вычисления, выполняемые порождающим
процессом
       n += 10;
   }
   printf("итоговое число:%d\n", n);
   printf("завершение процесса\n");
```

```
    exit(1);
    return 0;
}
```

Данная программа создает процесс с помощью функции fork() и позволяет различать, в каком процессе выполняется код: проверяется значение pid. В консоль выводится информация о PID текущего и родительского процессов и значение переменной n, которое меняется в ходе выполнения программы: если процесс родительский, к переменной добавляется 10, если дочерний, переменная умножается на 3.

Задание 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.

Результаты работы команды:

```
(base) katya@katya:~/os/lb34$ taskset 0x1 ./task1.out
начальное число:1
parrent pid = 6930, ppid = 6396
итоговое число:11
завершение процесса
new pid = 6931, pprid = 1501
итоговое число:3
завершение процесса
```

Видно, что выводятся идентификаторы каждого процесса и его родителя, а также дважды выводится сообщение о завершении процесса, что свидетельствует об исполнении одного и того же кодового сегмента обоими процессами. Распараллеливание - условное, если оба процесса выполняются на одном процессоре или ядре (т. е. в режиме разделения времени при многозадачности).

Можно заметить, что на первом этапе выполняются вычисления для родительского процесса, а затем для дочернего, значит, каждый процесс владеет своим процессорным ресурсом.

В данном коде создается один дочерний процесс с помощью системного вызова fork(). После создания дочернего процесса в каждом из них выполняются разные вычисления с переменными n. Важное замечание: в Unix-подобных средах, когда процесс создает дочерний процесс с помощью fork(), дочерний процесс получает преобразование адресного пространства родительского процесса. Это означает, что оба процесса (родительский и дочерний) имеют свои собственные копии технологии и памяти, которые изначально определены, но могут быть изменены независимо от другого. Важно понимать, что, хотя дочерний процесс и получает преобразование адресного пространства родительского процесса, эти копии являются изолированными. Изменения в одной копии не отражаются в другой, что является ключевым аспектом работы с процессами в Unix-подобных компонентах.

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

Программа task3.c:

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


void work() {
    int n = 0;
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 1000000000; j++)
            n += 1;
               printf("pid=%d, ppid=%d, policy=", getpid(),
getppid());
        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;
            }
        }
    }

    int main() {
        // направление вывода в текстовый файл
        freopen("task3.txt", "w", stdout);
        pid_t pid;

        for (int i = 0; i < 5; i++) {
            pid = fork();
            if (pid == 0) {
                // дочерний процесс
                    printf("START: pid=%d, ppid=%d, policy=",
getpid(), getppid());
                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;
                }
                // трудоемкая задача
                work();
                 printf("END: pid=%d, ppid=%d, policy=", getpid(),
getppid());
                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;
                }
                exit(EXIT_SUCCESS);
            }
        }
```

```
            // Для наблюдения конкуренции дочерних процессов
родительский дожидается их выполнения
        int status;
        for (int i = 0; i < 5; i++) {
            wait(&status);
        }
        return 0;
    }
```

SCHED_FIFO: политика планирования реального времени первый вошёл, первый вышел (First-In First-Out). Алгоритм планирования не использует никаких интервалов времени. Процесс SCHED_FIFO выполняется до завершения, если он не заблокирован запросом ввода/вывода, вытеснен высокоприоритетным процессом, или он добровольно отказывается от процессора.

SCHED_RR: циклическая (Round-Robin) политика планирования реального времени. Она похожа на SCHED_FIFO с той лишь разницей, что процессу SCHED_RR разрешено работать как максимум время кванта. Если процесс SCHED_RR исчерпывает свой квант времени, он помещается в конец списка с его приоритетом. Процесс SCHED_RR, который был вытеснен процессом с более высоким приоритетом, завершит оставшуюся часть своего кванта времени после возобновления выполнения.

SCHED_OTHER: стандартный планировщик Linux с разделением времени для процессов, работающих не в реальном времени.

(Информация взята из https://dmilvdv.narod.ru/Translate/ELSDD/elsdd_process_scheduling.html)

Результат выполнения программы:

```
START: pid=8136, ppid=8131, policy=SCHED_OTHER
START: pid=8135, ppid=8131, policy=SCHED_OTHER
START: pid=8134, ppid=8131, policy=SCHED_OTHER
START: pid=8133, ppid=8131, policy=SCHED_OTHER
START: pid=8132, ppid=8131, policy=SCHED_OTHER
pid=8136, ppid=8131, policy=SCHED_OTHER
pid=8135, ppid=8131, policy=SCHED_OTHER
pid=8134, ppid=8131, policy=SCHED_OTHER
pid=8133, ppid=8131, policy=SCHED_OTHER
```

```
pid=8132, ppid=8131, policy=SCHED_OTHER
pid=8136, ppid=8131, policy=SCHED_OTHER
pid=8135, ppid=8131, policy=SCHED_OTHER
pid=8134, ppid=8131, policy=SCHED_OTHER
pid=8133, ppid=8131, policy=SCHED_OTHER
pid=8132, ppid=8131, policy=SCHED_OTHER
pid=8136, ppid=8131, policy=SCHED_OTHER
pid=8134, ppid=8131, policy=SCHED_OTHER
pid=8133, ppid=8131, policy=SCHED_OTHER
pid=8135, ppid=8131, policy=SCHED_OTHER
pid=8132, ppid=8131, policy=SCHED_OTHER
pid=8136, ppid=8131, policy=SCHED_OTHER
pid=8135, ppid=8131, policy=SCHED_OTHER
pid=8134, ppid=8131, policy=SCHED_OTHER
pid=8132, ppid=8131, policy=SCHED_OTHER
pid=8133, ppid=8131, policy=SCHED_OTHER
pid=8136, ppid=8131, policy=SCHED_OTHER
pid=8135, ppid=8131, policy=SCHED_OTHER
pid=8134, ppid=8131, policy=SCHED_OTHER
pid=8132, ppid=8131, policy=SCHED_OTHER
pid=8133, ppid=8131, policy=SCHED_OTHER
pid=8136, ppid=8131, policy=SCHED_OTHER
pid=8135, ppid=8131, policy=SCHED_OTHER
pid=8134, ppid=8131, policy=SCHED_OTHER
pid=8132, ppid=8131, policy=SCHED_OTHER
pid=8133, ppid=8131, policy=SCHED_OTHER
pid=8135, ppid=8131, policy=SCHED_OTHER
pid=8134, ppid=8131, policy=SCHED_OTHER
pid=8133, ppid=8131, policy=SCHED_OTHER
pid=8136, ppid=8131, policy=SCHED_OTHER
pid=8132, ppid=8131, policy=SCHED_OTHER
pid=8135, ppid=8131, policy=SCHED_OTHER
pid=8133, ppid=8131, policy=SCHED_OTHER
pid=8134, ppid=8131, policy=SCHED_OTHER
pid=8136, ppid=8131, policy=SCHED_OTHER
pid=8132, ppid=8131, policy=SCHED_OTHER
pid=8135, ppid=8131, policy=SCHED_OTHER
pid=8133, ppid=8131, policy=SCHED_OTHER
pid=8134, ppid=8131, policy=SCHED_OTHER
pid=8136, ppid=8131, policy=SCHED_OTHER
pid=8132, ppid=8131, policy=SCHED_OTHER
pid=8135, ppid=8131, policy=SCHED_OTHER
END: pid=8135, ppid=8131, policy=SCHED_OTHER
pid=8134, ppid=8131, policy=SCHED_OTHER
END: pid=8134, ppid=8131, policy=SCHED_OTHER
pid=8132, ppid=8131, policy=SCHED_OTHER
END: pid=8132, ppid=8131, policy=SCHED_OTHER
pid=8136, ppid=8131, policy=SCHED_OTHER
END: pid=8136, ppid=8131, policy=SCHED_OTHER
pid=8133, ppid=8131, policy=SCHED_OTHER
END: pid=8133, ppid=8131, policy=SCHED_OTHER
```

Таким образом можно заметить, что, если однократные вычисления в коде заменить на циклическое исполнение, то можно будет наблюдать конкуренцию процессов за процессорный ресурс.

Задание 4. Измените процедуру планирования и повторите эксперимент.

Написана программа task4.c:

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


void work() {
    int n = 0;
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 1000000000; j++)
            n += 1;
               printf("pid=%d,  ppid=%d,  policy=",  getpid(),
getppid());
        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;
        }
    }
}

int main() {
    // направление вывода в текстовый файл
    //freopen("task3.txt", "w", stdout);
    pid_t pid;

    ////////////////////////////////////////
    // изменение процедуры планирования
    struct sched_param param;
    param.sched_priority = 1;
    if (sched_setscheduler(0, SCHED_FIFO, &param) == -1) {
        perror("sched_setscheduler");
        exit(EXIT_FAILURE);
    }
    ////////////////////////////////////////
```

```
        for (int i = 0; i < 5; i++) {
            pid = fork();
            if (pid == 0) {
                // дочерний процесс
                       printf("START: pid=%d, ppid=%d, policy=",
getpid(), getppid());
                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;
                }
                // трудоемкая задача
                work();
                 printf("END: pid=%d, ppid=%d, policy=", getpid(),
getppid());
                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;
                }
                exit(EXIT_SUCCESS);
            }
        }
             // Для наблюдения конкуренции дочерних процессов
родительский дожидается их выполнения
        int status;
        for (int i = 0; i < 5; i++) {
            wait(&status);
        }
        return 0;
    }
```

Изменение политики планирования: Перед созданием дочерних процессов в родительском процессе происходит изменение политики планирования с помощью функции sched_setscheduler(). Устанавливается

политика ФИФО с приоритетом 1. Это означает, что процесс будет приоритетным по получению выполнения перед другими процессами

Результат работы программы:

```
START: pid=18431, ppid=18430, policy=SCHED_FIFO
pid=18431, ppid=18430, policy=SCHED_FIFO
pid=18431, ppid=18430, policy=SCHED_FIFO
pid=18431, ppid=18430, policy=SCHED_FIFO
pid=18431, ppid=18430, policy=SCHED_FIFO
pid=18431, ppid=18430, policy=SCHED_FIFO
pid=18431, ppid=18430, policy=SCHED_FIFO
pid=18431, ppid=18430, policy=SCHED_FIFO
pid=18431, ppid=18430, policy=SCHED_FIFO
pid=18431, ppid=18430, policy=SCHED_FIFO
END: pid=18431, ppid=18430, policy=SCHED_FIFO
START: pid=18432, ppid=18430, policy=SCHED_FIFO
pid=18432, ppid=18430, policy=SCHED_FIFO
pid=18432, ppid=18430, policy=SCHED_FIFO
pid=18432, ppid=18430, policy=SCHED_FIFO
pid=18432, ppid=18430, policy=SCHED_FIFO
pid=18432, ppid=18430, policy=SCHED_FIFO
pid=18432, ppid=18430, policy=SCHED_FIFO
pid=18432, ppid=18430, policy=SCHED_FIFO
pid=18432, ppid=18430, policy=SCHED_FIFO
pid=18432, ppid=18430, policy=SCHED_FIFO
END: pid=18432, ppid=18430, policy=SCHED_FIFO
START: pid=18433, ppid=18430, policy=SCHED_FIFO
pid=18433, ppid=18430, policy=SCHED_FIFO
pid=18433, ppid=18430, policy=SCHED_FIFO
pid=18433, ppid=18430, policy=SCHED_FIFO
pid=18433, ppid=18430, policy=SCHED_FIFO
pid=18433, ppid=18430, policy=SCHED_FIFO
pid=18433, ppid=18430, policy=SCHED_FIFO
pid=18433, ppid=18430, policy=SCHED_FIFO
pid=18433, ppid=18430, policy=SCHED_FIFO
pid=18433, ppid=18430, policy=SCHED_FIFO
END: pid=18433, ppid=18430, policy=SCHED_FIFO
START: pid=18434, ppid=18430, policy=SCHED_FIFO
pid=18434, ppid=18430, policy=SCHED_FIFO
pid=18434, ppid=18430, policy=SCHED_FIFO
pid=18434, ppid=18430, policy=SCHED_FIFO
pid=18434, ppid=18430, policy=SCHED_FIFO
pid=18434, ppid=18430, policy=SCHED_FIFO
pid=18434, ppid=18430, policy=SCHED_FIFO
pid=18434, ppid=18430, policy=SCHED_FIFO
pid=18434, ppid=18430, policy=SCHED_FIFO
pid=18434, ppid=18430, policy=SCHED_FIFO
END: pid=18434, ppid=18430, policy=SCHED_FIFO
START: pid=18435, ppid=18430, policy=SCHED_FIFO
pid=18435, ppid=18430, policy=SCHED_FIFO
```

```
pid=18435, ppid=18430, policy=SCHED_FIFO
pid=18435, ppid=18430, policy=SCHED_FIFO
pid=18435, ppid=18430, policy=SCHED_FIFO
pid=18435, ppid=18430, policy=SCHED_FIFO
pid=18435, ppid=18430, policy=SCHED_FIFO
pid=18435, ppid=18430, policy=SCHED_FIFO
pid=18435, ppid=18430, policy=SCHED_FIFO
pid=18435, ppid=18430, policy=SCHED_FIFO
pid=18435, ppid=18430, policy=SCHED_FIFO
END: pid=18435, ppid=18430, policy=SCHED_FIFO
```

В предыдущем коде не было никакого изменения процессов планирования политики, поэтому все процессы используют политику планирования по умолчанию, которая, как правило, является SCHED_OTHER обычной для Unix-подобных процессов. Это означает, что процессы получают время выполнения в соответствии с алгоритмом планирования, который может включать в себя кванты времени и приоритеты, установленные для других процессов в системе.

В данном коде, в отличие от применения, явно задается планирование политики для процесса с помощью системного вызова sched_setscheduler(). Политика установки устанавливается SCHED_FIFO с приоритетом 1. Это означает, что процесс будет получать приоритетное выполнение перед другими процессами, которые могут использовать другие политические планы, такие как SCHED_OTHER или SCHED_RR.

Задание 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").

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 process is finished with status %d\n",
status);

        return 0;
    }
```

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
    }
```

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

Запуск программ происходит в пункте ниже.

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

Результат работы программы:

```
(base) katya@katya:~/os/lb34$ taskset 1 ./father.out
FATHER PARAM: pid=20482  ppid=18401
```

```
SON PARAMS: pid=20483  ppid=20482
Child process is finished with status 0
```

Проанализировав текстовый файл после запуска программы, видим, что процесс son является дочерним для father (иерархия процессов выведена в древовидной форме):

```
  18401 pts/2    Ss     0:00      \_ bash
  20482 pts/2    S+     0:00          \_ ./father.out
  20483 pts/2    S+     0:00              \_ son
  20484 pts/2    S+     0:00              \_ sh -c ps -xf >
file.txt
  20485 pts/2    R+     0:00                  \_ ps -xf
```

Назначение полей:

- PID —идентификатор процесс
- TTY —терминал, с которым связан данный процесс
- STAT—состояние, в котором на данный момент находится процесс-родитель
- 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
- +: порожденный процесс

Задание 7. Выполните программу father.out в фоновом режиме father&

Получите таблицу процессов, запущенных с вашего терминала (включая отцовский и сыновний процессы).

```
  18401 pts/2    Ss+    0:00  |   \_ bash
  20572 pts/2    S      0:00  |       \_ ./father.out
  20573 pts/2    S      0:00  |           \_ son
   20574 pts/2     S       0:00  |             \_ sh -c ps -xf >
file.txt
  20575 pts/2    R      0:00  |               \_ ps -xf
```

Можно заметить, что разница есть только в выводе программ, а таблица процессов полностью одинакова (только в фоновом режиме + стоит у bash, так как он - на переднем плане).

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

Функция exec() и её семейство функций в C используются для замены текущего процесса новым процессом. Это означает, что после вызова exec(), текущий процесс полностью заменяется новым процессом, включая его адресное пространство, стековую область, глобальные переменные и т.д.

execl позволяет передать список аргументов отдельными аргументами друг за другом,

execv аналогично, но список аргументов может быть массивом строк,

execlp заменяет текущий процесс на процесс исполняемого файла, указанного в аргументах,

execle задает окружение для дочернего процесса, по умолчанию это будет родительское огружение.

файл task8.c:

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

int main(int argc, char **argv){
    int pid = fork();
    char* const vector_args[] = {"task8_1", "first_arg",
"sec_arg", NULL};
    char* const env[] = {"ENV_VARIABLE=env_variable", NULL};
    if(pid == 0){
        switch (argv[1][0])
        {
        case '1':
        // замена дочернего процесса программой с аргументами
```

```
                execl("task8_1", "task8_1", "first_arg",
"sec_arg", NULL);
                break;

            case '2':
            // замена дочернего процесса программой с аргументами
в виде массива
                execv('task8_1', vector_args);
                break;

            case '3':
            // возможность использ переменную среду PATH для
поиска исполняемого файла
                execlp("echo", "echo", "some_info", NULL);
                break;

            case '4':
            // замена дочернего процесса программой, передавая ее
окружение в массиве env
                execle("task8_1", "task8_1", "some_info", NULL,
env);
                break;
            }
        }
        wait(NULL);
        return 0;
    }
```

Файл task8_1.c:

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

extern char **environ;

int main(int argc, char **argv){
    system("ps -ft > 8file.txt");
    for(int i=0; environ[i]; i++)
        puts(environ[i]);
    for(int i=1; argv[i]; i++)
        puts(argv[i]);
    return 0;
}
```

Вариации запуска программы:

```
(base) katya@katya:~/os/lb34$ ./task8.out 1
SHELL=/bin/bash
SESSION_MANAGER=local/katya:@/tmp/.ICE-unix/1813,unix/katya:/tmp/.
ICE-unix/1813
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
SSH_AGENT_LAUNCHER=gnome-keyring
XDG_MENU_PREFIX=gnome-
```

```
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
GTK_IM_MODULE=ibus
CONDA_EXE=/home/katya/anaconda3/bin/conda
_CE_M=
GNOME_SHELL_SESSION_MODE=ubuntu
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
GTK_MODULES=gail:atk-bridge
PWD=/home/katya/os/lb34
LOGNAME=katya
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=x11
CONDA_PREFIX=/home/katya/anaconda3
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
SYSTEMD_EXEC_PID=1836
XAUTHORITY=/run/user/1000/gdm/Xauthority
WINDOWPATH=2
LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libstdc++.so.6
HOME=/home/katya
USERNAME=katya
LANG=ru_RU.UTF-8
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:
bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;
41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.a
rc=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*.lz4=01;31:*.lzh=01;
31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.
zip=01;31:*.z=01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=01;31:*
.lzo=01;31:*.xz=01;31:*.zst=01;31:*.tzst=01;31:*.bz2=01;31:*.bz=01
;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.
jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.alz=01
;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.c
ab=01;31:*.wim=01;31:*.swm=01;31:*.dwm=01;31:*.esd=01;31:*.jpg=01;
35:*.jpeg=01;35:*.mjpg=01;35:*.mjpeg=01;35:*.gif=01;35:*.bmp=01;35
:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm
=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;
35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.
m2v=01;35:*.mkv=01;35:*.webm=01;35:*.webp=01;35:*.ogm=01;35:*.mp4=
01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:
*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=
01;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.
xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.ogv=01;35:*.ogx=01
;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.m4a=00;36:*.mid=00;36:*.
midi=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00
;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.spx=00;36:*.xspf=00;36:
XDG_CURRENT_DESKTOP=ubuntu:GNOME
VTE_VERSION=6800
CONDA_PROMPT_MODIFIER=(base)
GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/screen/47c27a63_7c77_470
7_8726_616db8ac6022
LESSCLOSE=/usr/bin/lesspipe %s %s
XDG_SESSION_CLASS=user
TERM=xterm-256color
_CE_CONDA=
LESSOPEN=| /usr/bin/lesspipe %s
USER=katya
```

```
GNOME_TERMINAL_SERVICE=:1.646
CONDA_SHLVL=1
DISPLAY=:1
SHLVL=1
QT_IM_MODULE=ibus
PROJ_LIB=/home/katya/anaconda3/share/proj
CONDA_PYTHON_EXE=/home/katya/anaconda3/bin/python
XDG_RUNTIME_DIR=/run/user/1000
CONDA_DEFAULT_ENV=base
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/share/gnome:/usr/local/share/
:/usr/share/:/var/lib/snapd/desktop
PATH=/home/katya/anaconda3/bin:/home/katya/anaconda3/condabin:/usr
/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/game
s:/usr/local/games:/snap/bin:/snap/bin
GDMSESSION=ubuntu
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
OLDPWD=/home/katya
_=./task8.out
first_arg
sec_arg

(base) katya@katya:~/os/lb34$ ./task8.out 2
SHELL=/bin/bash
SESSION_MANAGER=local/katya:@/tmp/.ICE-unix/1813,unix/katya:/tmp/.
ICE-unix/1813
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
SSH_AGENT_LAUNCHER=gnome-keyring
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
GTK_IM_MODULE=ibus
CONDA_EXE=/home/katya/anaconda3/bin/conda
_CE_M=
GNOME_SHELL_SESSION_MODE=ubuntu
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
GTK_MODULES=gail:atk-bridge
PWD=/home/katya/os/lb34
LOGNAME=katya
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=x11
CONDA_PREFIX=/home/katya/anaconda3
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
SYSTEMD_EXEC_PID=1836
XAUTHORITY=/run/user/1000/gdm/Xauthority
WINDOWPATH=2
LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libstdc++.so.6
HOME=/home/katya
USERNAME=katya
LANG=ru_RU.UTF-8
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:
bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;
41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.a
rc=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*.lz4=01;31:*.lzh=01;
```

```
31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.
zip=01;31:*.z=01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=01;31:*
.lzo=01;31:*.xz=01;31:*.zst=01;31:*.tzst=01;31:*.bz2=01;31:*.bz=01
;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.
jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.alz=01
;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.c
ab=01;31:*.wim=01;31:*.swm=01;31:*.dwm=01;31:*.esd=01;31:*.jpg=01;
35:*.jpeg=01;35:*.mjpg=01;35:*.mjpeg=01;35:*.gif=01;35:*.bmp=01;35
:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm
=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;
35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.
m2v=01;35:*.mkv=01;35:*.webm=01;35:*.webp=01;35:*.ogm=01;35:*.mp4=
01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:
*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=
01;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.
xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.ogv=01;35:*.ogx=01
;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.m4a=00;36:*.mid=00;36:*.
midi=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00
;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.spx=00;36:*.xspf=00;36:
XDG_CURRENT_DESKTOP=ubuntu:GNOME
VTE_VERSION=6800
CONDA_PROMPT_MODIFIER=(base)
GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/screen/47c27a63_7c77_470
7_8726_616db8ac6022
LESSCLOSE=/usr/bin/lesspipe %s %s
XDG_SESSION_CLASS=user
TERM=xterm-256color
_CE_CONDA=
LESSOPEN=| /usr/bin/lesspipe %s
USER=katya
GNOME_TERMINAL_SERVICE=:1.646
CONDA_SHLVL=1
DISPLAY=:1
SHLVL=1
QT_IM_MODULE=ibus
PROJ_LIB=/home/katya/anaconda3/share/proj
CONDA_PYTHON_EXE=/home/katya/anaconda3/bin/python
XDG_RUNTIME_DIR=/run/user/1000
CONDA_DEFAULT_ENV=base
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/share/gnome:/usr/local/share/
:/usr/share/:/var/lib/snapd/desktop
PATH=/home/katya/anaconda3/bin:/home/katya/anaconda3/condabin:/usr
/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/game
s:/usr/local/games:/snap/bin:/snap/bin
GDMSESSION=ubuntu
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
OLDPWD=/home/katya
_=./task8.out
first_arg
sec_arg

(base) katya@katya:~/os/lb34$ ./task8.out 3
some_info

(base) katya@katya:~/os/lb34$ ./task8.out 4
ENV_VARIABLE=env_variable
```

```
some_info
```

Можно отметить, что теория совпала с практикой: по умолчанию наследовалось окружение родителя (заметно по выводу environ).

Также по выводу программы видно, что аргументы можно передавать и с помощью массива, и перечислением - они успешно принимаются дочерней программой.

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

Функция wait(&status) в Unix-подобных операционных системах используется для ожидания завершения любого из дочерних процессов, созданных вызывающим процессом. Параметр status позволяет родительскому процессу получить информацию о завершении дочернего процесса, включая его код завершения и сигнал, если процесс был завершен сигналом.

Анализ возвращаемого значения:

-1: Ошибка, например, если процесс не является родителем дочернего процесса или если дочерних процессов нет.

PID дочернего процесса: Если дочерний процесс завершился, функция возвращает PID завершенного процесса.

task9.c:

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

int main(){
    int pid[3];
    printf("parent pid=%d, ppid=%d\n", getpid(), getppid());

    if(pid[0]=fork() == 0)
        execl("task9_child", "child0", NULL);
    if(pid[1]=fork() == 0)
        execl("task9_child", "child1", NULL);
```

```
    if(pid[2]=fork() == 0)
        execl("task9_child", "child2", NULL);
    system("ps -ft > task9.txt");

    for(int i=0; i<3; i++){
        int status;
        int ret_pid = wait(&status);
            printf("pid %d ended, status %d\n", ret_pid, status);
    }
    return 0;
}
```

task9_child.c:

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

int main(int argc, char **argv){
    printf("%s pid=%d, ppid=%d\n", argv[0], getpid(), getppid());
    return (int)argv[0][5] - '0';
}
```

```
(base) katya@katya:~/os/lb34$ ./task9
parent pid=5568, ppid=5551
child0 pid=5569, ppid=5568
child1 pid=5570, ppid=5568
child2 pid=5571, ppid=5568
pid 5569 ended, status 0
pid 5570 ended, status 256
pid 5571 ended, status 512
(base) katya@katya:~/os/lb34$ cat task9.txt
    PID TTY      STAT   TIME COMMAND
   5551 pts/2    Ss     0:00 bash
   5568 pts/2    S+     0:00  \_ ./task9
   5569 pts/2    Z+     0:00      \_ [task9_child] <defunct>
   5570 pts/2    Z+     0:00      \_ [task9_child] <defunct>
   5571 pts/2    Z+     0:00      \_ [task9_child] <defunct>
   5572 pts/2    S+     0:00      \_ sh -c ps -ft > task9.txt
   5573 pts/2    R+     0:00          \_ ps -ft
```

Изменим task9.c:

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

int main(){
    int pid[3];
    printf("parent pid=%d, ppid=%d\n", getpid(), getppid());

    if(pid[0]=fork() == 0)
        execl("task9_child", "child0", NULL);
    if(pid[1]=fork() == 0)
        execl("task9_child", "child1", NULL);
```

```
    if(pid[2]=fork() == 0)
        execl("task9_child", "child2", NULL);
    system("ps -ft > task9.txt");

    for(int i=0; i<3; i++){
        int status;
        int ret_pid = wait(&status);
                    printf("pid %d ended, status %d\n", ret_pid,
WEXITSTATUS(status));
    }
    return 0;
}
```

```
(base) katya@katya:~/os/lb34$ ./task9
parent pid=5732, ppid=5551
child0 pid=5733, ppid=5732
child1 pid=5734, ppid=5732
child2 pid=5735, ppid=5732
pid 5733 ended, status 0
pid 5734 ended, status 1
pid 5735 ended, status 2
(base) katya@katya:~/os/lb34$ cat task9.txt
    PID TTY      STAT   TIME COMMAND
   5551 pts/2    Ss     0:00 bash
   5732 pts/2    S+     0:00  \_ ./task9
   5733 pts/2    Z+     0:00      \_ [task9_child] <defunct>
   5734 pts/2    Z+     0:00      \_ [task9_child] <defunct>
   5735 pts/2    Z+     0:00      \_ [task9_child] <defunct>
   5736 pts/2    S+     0:00      \_ sh -c ps -ft > task9.txt
   5737 pts/2    R+     0:00          \_ ps -ft
```

WEXITSTATUS(status) возвращает восемь младших битов значения, которое вернул завершившийся потомок. Эти биты могли быть установлены в аргументе функции exit() или в аргументе оператора return функции main().

Можно заметить, что без WEXITSTATUS возвращаемое значение было умножено на 256.

Проанализировав ps, можно сказать, что дочерние процессы к моменту вызова wait были в состоянии зомби. При завершении процесс в любом случае освобождает все свои ресурсы и становится зомби, то есть пустой записью в таблице процессов, которая хранит статус завершения, предназначенный для чтения процессом-родителем. Зомби-процесс существует до тех пор, пока процесс-родитель не прочитает его статус с помощью системного вызова wait(), поэтому запись в таблице процессов

будет освобождена. Зомби-процессы не занимают памяти, но блокируют записи в таблице процессов, размер которой является ограниченной.

Для управления группой процессов можно использовать waitpid() с аргументом pid равным 0, что позволит ожидать завершения всех дочерних процессов в группе вызывающего процесса. Это может быть полезно для сценариев, когда необходимо синхронизировать выполнение нескольких процессов, созданных одним родительским процессом.

Задание 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.

При вызове команды fork() создается новый процесс-потомок, который является копией родительского. Затем дочерний процесс выполняет код, следующий за вызовом fork(), в то время как родительский процесс продолжает выполнять код, предшествующий вызову fork().

Если внутри процесса-потомка также вызывается команда fork(), то создается еще один процесс-потомок и так далее. Каждый новый дочерний процесс наследует копию адресного пространства и контекста выполнения родительского процесса. Таким образом, можно сказать, что вызов fork() следует древовидной структуре, где внутри каждого дочернего процесса можно создать еще один.

task10.c:

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


void work() {
    int n = 0;
    for (int i = 0; i < 10; i++) {
```

```
        for (int j = 0; j < 1000000000; j++)
            n += 1;
        printf("pid=%d, ppid=%d, policy=", getpid(), getppid());
        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;
        }
    }
}

int main() {
    // направление вывода в текстовый файл
    //freopen("task10.txt", "w", stdout);

    // для сына создается потомок, т е внук родительского
    if (!fork()) fork();
          printf("START:  pid=%d,  ppid=%d,  policy=",  getpid(),
getppid());
    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;
            }
            // трудоемкая задача
            work();
                printf("END: pid=%d, ppid=%d, policy=", getpid(),
getppid());
            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;
            }
```

```
     // Для наблюдения конкуренции дочерних процессов родительский
дожидается их выполнения
    wait(NULL);
    return 0;
}
```

```
(base) katya@katya:~/os/lb34$ taskset 1 ./task10.out
START: pid=8572, ppid=5551, policy=SCHED_OTHER
START: pid=8573, ppid=8572, policy=SCHED_OTHER
START: pid=8574, ppid=8573, policy=SCHED_OTHER
pid=8572, ppid=5551, policy=SCHED_OTHER
pid=8573, ppid=8572, policy=SCHED_OTHER
pid=8574, ppid=8573, policy=SCHED_OTHER
pid=8572, ppid=5551, policy=SCHED_OTHER
pid=8574, ppid=8573, policy=SCHED_OTHER
pid=8573, ppid=8572, policy=SCHED_OTHER
pid=8572, ppid=5551, policy=SCHED_OTHER
pid=8573, ppid=8572, policy=SCHED_OTHER
pid=8574, ppid=8573, policy=SCHED_OTHER
pid=8573, ppid=8572, policy=SCHED_OTHER
pid=8574, ppid=8573, policy=SCHED_OTHER
pid=8572, ppid=5551, policy=SCHED_OTHER
pid=8574, ppid=8573, policy=SCHED_OTHER
pid=8573, ppid=8572, policy=SCHED_OTHER
pid=8572, ppid=5551, policy=SCHED_OTHER
pid=8574, ppid=8573, policy=SCHED_OTHER
pid=8572, ppid=5551, policy=SCHED_OTHER
pid=8573, ppid=8572, policy=SCHED_OTHER
pid=8572, ppid=5551, policy=SCHED_OTHER
pid=8574, ppid=8573, policy=SCHED_OTHER
pid=8573, ppid=8572, policy=SCHED_OTHER
pid=8573, ppid=8572, policy=SCHED_OTHER
pid=8574, ppid=8573, policy=SCHED_OTHER
pid=8572, ppid=5551, policy=SCHED_OTHER
pid=8573, ppid=8572, policy=SCHED_OTHER
pid=8574, ppid=8573, policy=SCHED_OTHER
pid=8572, ppid=5551, policy=SCHED_OTHER
pid=8572, ppid=5551, policy=SCHED_OTHER
END: pid=8572, ppid=5551, policy=SCHED_OTHER
pid=8574, ppid=8573, policy=SCHED_OTHER
END: pid=8574, ppid=8573, policy=SCHED_OTHER
pid=8573, ppid=1524, policy=SCHED_OTHER
END: pid=8573, ppid=1524, policy=SCHED_OTHER
```

Изменение политики планирования в task10.c:

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

```
void work() {
    int n = 0;
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 1000000000; j++)
            n += 1;
        printf("pid=%d, ppid=%d, policy=", getpid(), getppid());
        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;
        }
    }
}

int main() {
    // направление вывода в текстовый файл
    //freopen("task10.txt", "w", stdout);

    //изменение политики планирования
    struct sched_param param;
    param.sched_priority = 1;
    if(sched_setscheduler(0, SCHED_FIFO, &param) == -1){
        perror("sched_setscheduler");
        exit(EXIT_FAILURE);
    }

    // для сына создается потом, т е внук родительского
    if (!fork()) fork();
          printf("START:  pid=%d,  ppid=%d,  policy=",  getpid(),
getppid());
    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;
            }
            // трудоемкая задача
            work();
                 printf("END: pid=%d, ppid=%d, policy=", getpid(),
getppid());
            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;
            }


     // Для наблюдения конкуренции дочерних процессов родительский
дожидается их выполнения
    wait(NULL);
    return 0;
}

(base) katya@katya:~/os/lb34$ sudo taskset 1 ./task10.out
START: pid=8767, ppid=8766, policy=SCHED_FIFO
pid=8767, ppid=8766, policy=SCHED_FIFO
pid=8767, ppid=8766, policy=SCHED_FIFO
pid=8767, ppid=8766, policy=SCHED_FIFO
pid=8767, ppid=8766, policy=SCHED_FIFO
pid=8767, ppid=8766, policy=SCHED_FIFO
pid=8767, ppid=8766, policy=SCHED_FIFO
pid=8767, ppid=8766, policy=SCHED_FIFO
pid=8767, ppid=8766, policy=SCHED_FIFO
pid=8767, ppid=8766, policy=SCHED_FIFO
END: pid=8767, ppid=8766, policy=SCHED_FIFO
START: pid=8768, ppid=8767, policy=SCHED_FIFO
pid=8768, ppid=8767, policy=SCHED_FIFO
pid=8768, ppid=8767, policy=SCHED_FIFO
pid=8768, ppid=8767, policy=SCHED_FIFO
pid=8768, ppid=8767, policy=SCHED_FIFO
pid=8768, ppid=8767, policy=SCHED_FIFO
pid=8768, ppid=8767, policy=SCHED_FIFO
pid=8768, ppid=8767, policy=SCHED_FIFO
pid=8768, ppid=8767, policy=SCHED_FIFO
pid=8768, ppid=8767, policy=SCHED_FIFO
END: pid=8768, ppid=8767, policy=SCHED_FIFO
START: pid=8781, ppid=8768, policy=SCHED_FIFO
pid=8781, ppid=8768, policy=SCHED_FIFO
pid=8781, ppid=8768, policy=SCHED_FIFO
pid=8781, ppid=8768, policy=SCHED_FIFO
pid=8781, ppid=8768, policy=SCHED_FIFO
pid=8781, ppid=8768, policy=SCHED_FIFO
pid=8781, ppid=8768, policy=SCHED_FIFO
pid=8781, ppid=8768, policy=SCHED_FIFO
pid=8781, ppid=8768, policy=SCHED_FIFO
pid=8781, ppid=8768, policy=SCHED_FIFO
END: pid=8781, ppid=8768, policy=SCHED_FIFO
```

Таким образом, при вложенном fork() с политикой планирования SCHED_OTHER процессы выполняются чередуясь, начиная с родителя до потомка сына (внука).

Политика SCHED_FIFO демонстрирует такую же очередность, но родительские процессы, захватив процессорный ресурс, не освобождают его до выполнения задачи.

Задание 10.3. Поменяйте порядок очереди в RR-процедуре.

task 10_3.c:

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


void work() {
    int n = 0;
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 1000000000; j++)
            n += 1;
        printf("pid=%d, ppid=%d, policy=", getpid(), getppid());
        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;
        }

    }
}

int main()
{
    // Изменение политики планирования
    struct sched_param param;
    param.sched_priority = 1;
    if (sched_setscheduler(0, SCHED_RR, &param) == -1) {
        perror("sched_setscheduler");
        exit(EXIT_FAILURE);
    }
```

```
    // для сына создается потомок, т е внук родительского
    if (!fork()) {
        param.sched_priority = 99;
        sched_setscheduler(0, SCHED_RR, &param);
        if (!fork()) {
            param.sched_priority = 99;
            sched_setscheduler(0, SCHED_RR, &param);
        } else {
            sched_yield();
        }
    }
    sched_getparam(0, &param);
        printf("START:  pid=%d,  ppid=%d,  priority=%d,  policy=",
getpid(), getppid(), param.sched_priority);
    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;
    }
    // запускается трудоемкая задача
    work();
    printf("END: pid=%d, ppid=%d, policy=", getpid(), getppid());
    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;
    }
    // родители дожидаются детей
    wait(NULL);
    return 0;
}

(base) katya@katya:~/os/lb34$ sudo taskset 1 ./task10_3.out
START: pid=9916, ppid=9915, priority=1, policy=SCHED_RR
START: pid=9918, ppid=9917, priority=99, policy=SCHED_RR
START: pid=9917, ppid=9916, priority=99, policy=SCHED_RR
pid=9918, ppid=9917, policy=SCHED_RR
pid=9917, ppid=9916, policy=SCHED_RR
pid=9918, ppid=9917, policy=SCHED_RR
pid=9917, ppid=9916, policy=SCHED_RR
pid=9918, ppid=9917, policy=SCHED_RR
pid=9917, ppid=9916, policy=SCHED_RR
```

```
pid=9917, ppid=9916, policy=SCHED_RR
pid=9918, ppid=9917, policy=SCHED_RR
pid=9917, ppid=9916, policy=SCHED_RR
pid=9918, ppid=9917, policy=SCHED_RR
pid=9917, ppid=9916, policy=SCHED_RR
pid=9918, ppid=9917, policy=SCHED_RR
pid=9918, ppid=9917, policy=SCHED_RR
pid=9917, ppid=9916, policy=SCHED_RR
pid=9917, ppid=9916, policy=SCHED_RR
pid=9918, ppid=9917, policy=SCHED_RR
pid=9917, ppid=9916, policy=SCHED_RR
pid=9918, ppid=9917, policy=SCHED_RR
pid=9917, ppid=9916, policy=SCHED_RR
END: pid=9917, ppid=9916, policy=SCHED_RR
pid=9918, ppid=9917, policy=SCHED_RR
END: pid=9918, ppid=9917, policy=SCHED_RR
pid=9916, ppid=9915, policy=SCHED_RR
pid=9916, ppid=9915, policy=SCHED_RR
pid=9916, ppid=9915, policy=SCHED_RR
pid=9916, ppid=9915, policy=SCHED_RR
pid=9916, ppid=9915, policy=SCHED_RR
pid=9916, ppid=9915, policy=SCHED_RR
pid=9916, ppid=9915, policy=SCHED_RR
pid=9916, ppid=9915, policy=SCHED_RR
pid=9916, ppid=9915, policy=SCHED_RR
pid=9916, ppid=9915, policy=SCHED_RR
END: pid=9916, ppid=9915, policy=SCHED_RR
```

После изменения приоритетов для политики SCHED_RR внук стал приоритетным, родитель стал самым неприоритетным, в исполнении пропало чередование, так как статический приоритет внука выше и он работает до своего завершения.

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

task10_4.c:

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


void work() {
    int n = 0;
```

```
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 1000000000; j++)
            n += 1;
        printf("pid=%d, ppid=%d, policy=", getpid(), getppid());
        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;
        }

    }
}

int main(){
    pid_t pid;
    struct sched_param param;
    for(int i=0; i<4; i++){
        pid = fork();
        if (pid==0){//дочерний процесс
            //чет номер RR, нечет FIFO, приоритет одинаковый
            param.sched_priority = 50;
            if(i%2)
                sched_setscheduler(0, SCHED_FIFO, &param);
            else sched_setscheduler(0, SCHED_RR, &param);
               printf("START: pid=%d, ppid=%d, policy=", getpid(),
getppid());
            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;
            }
            // трудоемкая задача
            work();
                 printf("END: pid=%d, ppid=%d, policy=", getpid(),
getppid());
            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;
            }
            exit(EXIT_SUCCESS);
        }
    }
     // Для наблюдения конкуренции дочерних процессов родительский
дожидается их выполнения
    int status;
    for (int i = 0; i < 4; i++) {
        wait(&status);
    }
    return 0;
}
```

```
(base) katya@katya:~/os/lb34$ gcc task10_4.c -o task10_4.out
(base) katya@katya:~/os/lb34$ sudo taskset 1 ./task10_4.out
START: pid=15444, ppid=15440, policy=SCHED_FIFO
pid=15444, ppid=15440, policy=SCHED_FIFO
pid=15444, ppid=15440, policy=SCHED_FIFO
START: pid=15443, ppid=15440, policy=SCHED_RR
START: pid=15442, ppid=15440, policy=SCHED_FIFO
START: pid=15441, ppid=15440, policy=SCHED_RR
pid=15444, ppid=15440, policy=SCHED_FIFO
pid=15444, ppid=15440, policy=SCHED_FIFO
pid=15444, ppid=15440, policy=SCHED_FIFO
pid=15444, ppid=15440, policy=SCHED_FIFO
pid=15444, ppid=15440, policy=SCHED_FIFO
pid=15444, ppid=15440, policy=SCHED_FIFO
pid=15444, ppid=15440, policy=SCHED_FIFO
pid=15444, ppid=15440, policy=SCHED_FIFO
END: pid=15444, ppid=15440, policy=SCHED_FIFO
pid=15442, ppid=15440, policy=SCHED_FIFO
pid=15442, ppid=15440, policy=SCHED_FIFO
pid=15442, ppid=15440, policy=SCHED_FIFO
pid=15442, ppid=15440, policy=SCHED_FIFO
pid=15442, ppid=15440, policy=SCHED_FIFO
pid=15442, ppid=15440, policy=SCHED_FIFO
pid=15442, ppid=15440, policy=SCHED_FIFO
pid=15442, ppid=15440, policy=SCHED_FIFO
pid=15442, ppid=15440, policy=SCHED_FIFO
pid=15442, ppid=15440, policy=SCHED_FIFO
END: pid=15442, ppid=15440, policy=SCHED_FIFO
pid=15443, ppid=15440, policy=SCHED_RR
pid=15441, ppid=15440, policy=SCHED_RR
pid=15443, ppid=15440, policy=SCHED_RR
pid=15441, ppid=15440, policy=SCHED_RR
pid=15443, ppid=15440, policy=SCHED_RR
pid=15441, ppid=15440, policy=SCHED_RR
pid=15443, ppid=15440, policy=SCHED_RR
pid=15441, ppid=15440, policy=SCHED_RR
pid=15443, ppid=15440, policy=SCHED_RR
pid=15441, ppid=15440, policy=SCHED_RR
pid=15443, ppid=15440, policy=SCHED_RR
```

```
pid=15441, ppid=15440, policy=SCHED_RR
pid=15443, ppid=15440, policy=SCHED_RR
pid=15441, ppid=15440, policy=SCHED_RR
pid=15443, ppid=15440, policy=SCHED_RR
pid=15441, ppid=15440, policy=SCHED_RR
pid=15443, ppid=15440, policy=SCHED_RR
pid=15441, ppid=15440, policy=SCHED_RR
pid=15443, ppid=15440, policy=SCHED_RR
END: pid=15443, ppid=15440, policy=SCHED_RR
pid=15441, ppid=15440, policy=SCHED_RR
END: pid=15441, ppid=15440, policy=SCHED_RR
```

В ходе эксперимента было запущено 4 процесса: 2 SCHED_FIFO, 2 SCHED_RR с одинаковыми приоритетами. Сначала FIFO процессы захватывают процессорное время и не конкурируют между собой, после их завершения происходят процессы RR, которые уже конкурируют между собой. Таким образом, более приоритетной политикой будет SCHED_FIFO при равных статических приоритетах.

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

Квант времени — это численное значение, которое характеризует, как долго может выполняться задание до того момента, пока оно не будет вытеснено.

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. Начиная с версии Linux 2.6.24, квант SCHED_RR не может быть изменен документированными средствами.

task11.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;
}


(base) katya@katya:~/os/lb34$ ./task11.out
FATHER PARAMS: pid=17250 ppid=15258
Nice value = 19
SCHED_SETSCHEDULER_1: Operation not permitted
Квант при циклическом планировании: 0 сек 20000000 нс
SON: Квант процессорного времени: 0 сек 20000000 нс
SON PARAMS: pid=17251  ppid=17250
Процесс с pid = 17251 завершен
```

Таким образом мы можем узнать размер кванта при циклическом планировании, но не можем его изменить.

У SCHED_FIFO размер 0, так как данная политика вообще не подразумевает квантование.

Задание 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.

task12.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() {
    int status;
    int fdrd, fdwr;
    char str1[10], str2[10];
    struct sched_param param;
    param.sched_priority = 1;
    sched_setscheduler(0, SCHED_RR, &param);

     // Заблокировать все страницы памяти процесса в оперативной
памяти
    if (mlockall(MCL_CURRENT | MCL_FUTURE) < 0)
        perror("mlockall error");


    // Открыть файл для чтения и файла для записи
    if ((fdrd = open("task12_input.txt", O_RDONLY)) == -1)
        perror("Opening file");
    if ((fdwr = creat("task12_output.txt", 0666)) == -1)
        perror("Creating file");
    // Преобразовать дескрипторы файлов в строковые значения
```

```
    itoa(str1, fdrd);
    itoa(str2,fdwr);
    // Создать два процесса-потомка
    for (int i = 0; i < 2; i++) {
        if (fork() == 0) {
            param.sched_priority = 90;
            sched_setscheduler(0, SCHED_RR, &param);
            execl("12son.out", "son12", str1, str2, NULL);
        }
    }
        // Ждем завершения детей
    for (int i = 0; i < 2; i++)
        printf("Process pid = %d completed\n", wait(&status));
    // Закрыть файл для чтения
    if (close(fdrd) != 0)
        perror("Closing file");

    return 0;
}
```

12son.c:

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


int main(int argc, char *argv[]) {
     // Заблокировать все страницы памяти процесса в оперативной
памяти
    if (mlockall(MCL_CURRENT | MCL_FUTURE) < 0)
        perror("mlockall error");

    char c;
    char buff[3];
    int pid, ppid;
     int fdrd = atoi(argv[1]); // Преобразовать строковый параметр
входного файла в целочисленный дескриптор файла
     int fdwr = atoi(argv[2]); // Преобразовать строковый параметр
выходного файла в целочисленный дескриптор файла
    pid = getpid();
    ppid = getppid();
    printf("son file descriptor = %d\n", fdrd);
    printf("son params: pid=%i ppid=%i\n", pid, ppid);

    // Работа с файлами
    for(;;)
    {
        sleep(2);
```

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

```
(base) katya@katya:~/os/lb34$ sudo taskset 1 ./task12.out
son file descriptor = 3
son params: pid=18197 ppid=18196
son file descriptor = 3
son params: pid=18199 ppid=18196
pid = 18197: k
pid = 18199: a
pid = 18197: t
pid = 18199: y
pid = 18197: a
pid = 18199:
pid = 18197: s
pid = 18199: o
pid = 18197: t
pid = 18199: s
pid = 18197:

Process pid = 18199 completed
Process pid = 18197 completed
(base) katya@katya:~/os/lb34$ cat task12_input.txt
katya sots
(base) katya@katya:~/os/lb34$ cat task12_output.txt
katya sots
```

Изменим 12son.c:

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


int main(int argc, char *argv[]) {
     // Заблокировать все страницы памяти процесса в оперативной
памяти
    if (mlockall(MCL_CURRENT | MCL_FUTURE) < 0)
        perror("mlockall error");

    char c;
    char buff[3];
```

```
    int pid, ppid;
     int fdrd = atoi(argv[1]); // Преобразовать строковый параметр
входного файла в целочисленный дескриптор файла
     int fdwr = atoi(argv[2]); // Преобразовать строковый параметр
выходного файла в целочисленный дескриптор файла
    pid = getpid();
    ppid = getppid();
    printf("son file descriptor = %d\n", fdrd);
    printf("son params: pid=%i ppid=%i\n", pid, ppid);
     system("ps -o uid,gid,ruid,pid,ppid,pgid,tty,vsz,stat,command
> 12ps_info.txt");
    // Работа с файлами
    for(;;)
    {
        sleep(2);
        if (read(fdrd,&c,1) != 1)
            return 0;
        write(fdwr,&c,1);
        printf("pid = %d: %c\n", pid, c);
        if (close(fdrd) != 0)
            perror("Closing file!");
    }
}


(base) katya@katya:~/os/lb34$ sudo taskset 1 ./task12.out
son file descriptor = 3
son params: pid=18396 ppid=18395
son file descriptor = 3
son params: pid=18398 ppid=18395
pid = 18396: k
pid = 18398: a
Process pid = 18396 completed
Process pid = 18398 completed
(base) katya@katya:~/os/lb34$ cat task12_output.txt
ka(base) katya@katya:~/os/lb34$ cat 12ps_info.txt
  UID    GID   RUID      PID    PPID    PGID TT           VSZ STAT
COMMAND
     0      0  1000    18394    18393    18394 pts/3      14428 Ss
sudo taskset 1 ./task12.out
     0      0      0    18395    18394    18395 pts/3       2644 SL+
./task12.out
     0      0      0    18396    18395    18395 pts/3       2776 SL+
son12 3 4
     0      0      0    18398    18395    18395 pts/3       2776 SL+
son12 3 4
     0      0      0   18409   18398   18395 pts/3      2892 R+   sh
-c   ps   -o   uid,gid,ruid,pid,ppid,pgid,tty,vsz,stat,command   >
12ps_info.txt
     0      0      0   18410   18409   18395 pts/3     12712 R+   ps
-o uid,gid,ruid,pid,ppid,pgid,tty,vsz,stat,command
```

Таким образом подтверждается наследование таблицы файлов и возможность работы с дескрипторами файлов в дочерних процессах, ядро смещает внутрифайловые указатели после каждой операции чтения или записи, поэтому оба процесса никогда не обратятся вместе на чтение или запись по одному и тому же указателю или смещению внутри файла. Можно заметить, как при изменении программы на закрытие файла внутри одного из потомков происходит очевидное завершение, так как другой потомок не может читать дальше. Вывод ps показывает немало наследуемых параметров при использовании fork(). Текущая рабочая директория наследуется, наследуется окружение.

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); 86 сигналов процесса-родителя, требующих ответа; блокированных файлов (record locking).

Литература:

1) https://elib.spbstu.ru/dl/2/s17-71.pdf/download/s17-71.pdf
   Методическое пособие «Практические вопросы разработки системных приложений», Душутина Е.В
2) https://www.opennet.ru/docs/RUS/lnx_process/process2.html
   Управление процессами в Linux
3) https://studfile.net/preview/7707813/page:13/ дополнительная информация о функциях семейства exec