



Министерство науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«Московский государственный технический университет  
имени Н.Э. Баумана  
(национальный исследовательский университет)»  
(МГТУ им. Н.Э. Баумана)

---

ФАКУЛЬТЕТ \_\_\_\_\_ «Информатика и системы управления»

КАФЕДРА \_\_\_\_\_ «Программное обеспечение ЭВМ и информационные технологии»

## ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №4 по курсу «Операционные системы»

«Процессы. Системные вызовы fork() и exec()»

Студент \_\_\_\_\_ Маслова Марина Дмитриевна

Группа \_\_\_\_\_ ИУ7-53Б

Оценка (баллы) \_\_\_\_\_

Преподаватель \_\_\_\_\_ Рязанова Наталья Юрьевна

2021 г.

# Задание №1

Процессы-сироты. В программе создаются не менее двух потомков. В потомках вызывается *sleep()*, чтобы предок гарантированно завершился раньше своих потомков. На рисунке 1 продемонстрирована информация об идентификаторах процессов и их группе, а также усыновление процессов-потомков, родительский процесс которых завершился раньше них, процессом с идентификатором 1 (процессом "открывшим" терминал).

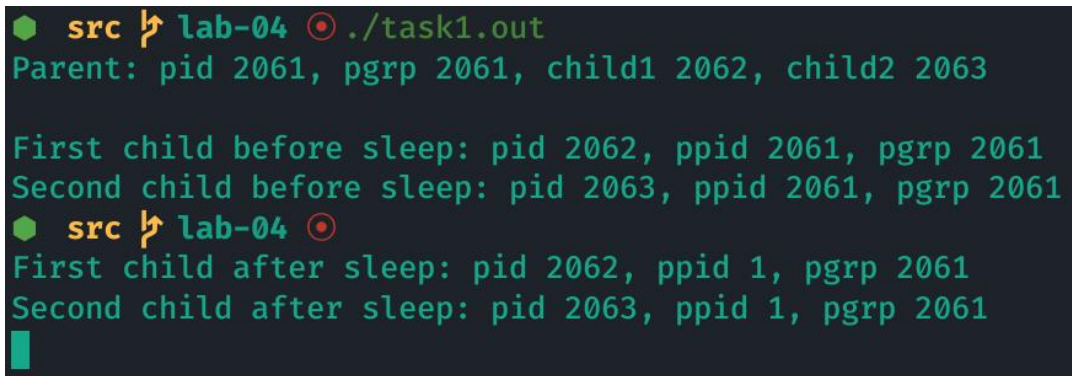
Листинг 1 – Процессы-сироты

```
1 #include <sys/types.h>
2 #include <unistd.h>
3
4 #include <stdio.h>
5
6 #define OK 0
7 #define ERROR 1
8 #define FORK_ERR -1
9
10 #define SLEEP_TIME 1
11
12 int main(void)
13 {
14     pid_t first_child_id, second_child_id;
15
16     if ((first_child_id = fork()) == FORK_ERR)
17     {
18         perror("Can't fork!\n");
19         return ERROR;
20     }
21
22     if (first_child_id == 0)
23     {
24         printf("First child before sleep: pid %d, ppid %d, pgrp %d\n",
25             getpid(), getppid(), getpgrp());
26
27         sleep(SLEEP_TIME);
28
29         printf("\nFirst child after sleep: pid %d, ppid %d, pgrp %d\n",
30             getpid(), getppid(), getpgrp());
31         return OK;
32     }
33
34     if ((second_child_id = fork()) == FORK_ERR)
35     {
36         perror("Can't fork!\n");
```

```

37     return ERROR;
38 }
39
40 if (second_child_id == 0)
41 {
42     printf("Second child before sleep: pid %d, ppid %d, pgrp %d\n",
43           getpid(), getppid(), getpgrp());
44
45     sleep(SLEEP_TIME);
46
47     printf("Second child after sleep: pid %d, ppid %d, pgrp %d\n",
48           getpid(), getppid(), getpgrp());
49     return OK;
50 }
51
52 printf("Parent: pid %d, pgrp %d, child1 %d, child2 %d\n\n",
53       getpid(), getpgrp(), first_child_id, second_child_id);
54
55 return OK;
56 }

```



```

src lab-04 ./task1.out
Parent: pid 2061, pgrp 2061, child1 2062, child2 2063

First child before sleep: pid 2062, ppid 2061, pgrp 2061
Second child before sleep: pid 2063, ppid 2061, pgrp 2061
src lab-04
First child after sleep: pid 2062, ppid 1, pgrp 2061
Second child after sleep: pid 2063, ppid 1, pgrp 2061

```

Рисунок 1 – Демонстрация работы программы (задание №1)

## Задание №2

Предок ждет завершения своих потомков, используя системный вызов *wait()*. Предок выполняет анализ кодов завершения потомков. На экран выводятся соответствующие сообщения.

Системный вызов *wait()* блокирует родительский процесс до момента завершения дочернего. При этом процесс-предок получает статус завершения процесса-потомка.

Листинг 2 – Системный вызов *wait()*

```
1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <unistd.h>
4
5 #include <stdio.h>
6
7 #define OK 0
8 #define ERROR 1
9 #define FORK_ERR -1
10
11 #define SLEEP_TIME 1
12
13 void check_status(const int status)
14 {
15     if (WIFEXITED(status))
16     {
17         printf("Child exited with code %d\n", WEXITSTATUS(status));
18     }
19     else if (WIFSIGNALED(status))
20     {
21         printf("Child was terminated by a signal with number %d\n",
22             WIFSIGNALED(status));
23     }
24     else if (WIFSTOPPED(status))
25     {
26         printf("Child was stopped by a signal with number %d\n",
27             WSTOPSIG(status));
28     }
29 }
30
31 int main(void)
32 {
33     pid_t first_child_id, second_child_id, child_pid;
34     int status;
35 }
```

```

36     if ((first_child_id = fork()) == FORK_ERR)
37     {
38         perror("Can't fork!\n");
39         return ERROR;
40     }
41
42     if (first_child_id == 0)
43     {
44         sleep(SLEEP_TIME);
45         printf("FIRST CHILD: pid %d, ppid %d, pgrp %d\n",
46             getpid(), getppid(), getpgrp());
47         return OK;
48     }
49
50     if ((second_child_id = fork()) == FORK_ERR)
51     {
52         perror("Can't fork!\n");
53         return ERROR;
54     }
55
56     if (second_child_id == 0)
57     {
58         sleep(SLEEP_TIME);
59         printf("SECOND CHILD: pid %d, ppid %d, pgrp %d\n",
60             getpid(), getppid(), getpgrp());
61         return OK;
62     }
63
64     printf("PARENT: pid %d, pgrp %d, child1 %d, child2 %d\n\n",
65         getpid(), getpgrp(), first_child_id, second_child_id);
66
67     child_pid = wait(&status);
68     printf("\nChild with pid = %d has finished\n", child_pid);
69     check_status(status);
70
71     child_pid = wait(&status);
72     printf("\nChild with pid = %d has finished\n", child_pid);
73     check_status(status);
74
75     return OK;
76 }

```

```
src lab-04 ./task2.out
PARENT: pid 2092, pgrp 2092, child1 2093, child2 2094

FIRST CHILD: pid 2093, ppid 2092, pgrp 2092
SECOND CHILD: pid 2094, ppid 2092, pgrp 2092

Child with pid = 2093 has finished
Child exited with code 0

Child with pid = 2094 has finished
Child exited with code 0
```

Рисунок 2 – Демонстрация работы программы (задание №2)

## Задание №3

Процессы-потомки переходят на выполнение других программ, которые передаются системному вызову *exec()*. Один процесс выполняет программу, осуществляющую поиск расстояния Левенштейна между двумя строками, другой – программу, осуществляющую поиск недостижимых вершин из данной в ориентированном графе. Предок ждет завершения своих потомков с анализом кодов завершения. На экран выводятся соответствующие сообщения.

Листинг 3 – Системный вызов *exec()*

```
1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <unistd.h>
4
5 #include <stdio.h>
6
7 #define OK 0
8 #define ERROR 1
9 #define FORK_ERR -1
10 #define EXEC_ERR -1
11
12 void check_status(const int status)
13 {
14     if (WIFEXITED(status))
15     {
16         printf("Child exited with code %d\n", WEXITSTATUS(status));
17     }
18     else if (WIFSIGNALED(status))
19     {
20         printf("Child was terminated by a signal with number %d\n",
21             WIFSIGNALED(status));
22     }
23     else if (WIFSTOPPED(status))
24     {
25         printf("Child was stopped by a signal with number %d\n",
26             WSTOPSIG(status));
27     }
28 }
29
30 int main(void)
31 {
32     pid_t first_child_id, second_child_id, child_pid;
33     int status;
34 }
```

```

35     if ((first_child_id = fork()) == FORK_ERR)
36     {
37         perror("Can't fork!\n");
38         return ERROR;
39     }
40
41     if (first_child_id == 0)
42     {
43         printf("FIRST CHILD: pid %d, ppid %d, pgrp %d\n",
44             getpid(), getppid(), getpgrp());
45
46         if (execl("./levenstein.out", "./levenstein.out",
47             "./data/test1.txt", 0) == EXEC_ERR)
48         {
49             printf("Can't exec!\n");
50             return ERROR;
51         }
52
53         return OK;
54     }
55
56     if ((second_child_id = fork()) == FORK_ERR)
57     {
58         perror("Can't fork!\n");
59         return ERROR;
60     }
61
62     if (second_child_id == 0)
63     {
64         printf("SECOND CHILD: pid %d, ppid %d, pgrp %d\n",
65             getpid(), getppid(), getpgrp());
66
67         if (execl("./graph.out", "./graph.out",
68             "./data/test2.txt", 0) == EXEC_ERR)
69         {
70             printf("Can't exec!\n");
71             return ERROR;
72         }
73
74         return OK;
75     }
76
77     printf("PARENT: pid %d, pgrp %d, child1 %d, child2 %d\n\n",
78         getpid(), getpgrp(), first_child_id, second_child_id);
79
80     child_pid = wait(&status);
81     printf("\nChild with pid = %d has finished\n", child_pid);
82     check_status(status);

```



```

83
84     child_pid = wait(&status);
85     printf("\nChild with pid = %d has finished\n", child_pid);
86     check_status(status);
87
88     return OK;
89 }

```

```

src lab-04 ./task3.out
PARENT: pid 1539, pgrp 1539, child1 1540, child2 1541

FIRST CHILD: pid 1540, ppid 1539, pgrp 1539
SECOND CHILD: pid 1541, ppid 1539, pgrp 1539

РАССТОЯНИЕ ЛЕВЕНШТЕЙНА

Первое слово: grate
Второе слово: giraffe
Расстояние Левенштейна между данными словами равно 3

ПОИСК НЕДОСТИЖИМЫХ ВЕРШИН В ОРИЕНТИРОВАННОМ ГРАФЕ

Заданный граф:
Количество вершин: 5
Количество ребер: 6
Ребра:
1 → 4
1 → 5
2 → 1
2 → 3
3 → 1
4 → 5
Вершина для поиска недостижимых вершин: 1
Недостижимые вершины:
2
3

Child with pid = 1540 has finished
Child exited with code 0

Child with pid = 1541 has finished
Child exited with code 0

```

Рисунок 3 – Демонстрация работы программы (задание №3)

## Задание №4

Предок и потомки обмениваются сообщениями через неименованный программный канал. Причем оба потомка пишут свои сообщения в один программный канал, а предок их считывает из канала. Потомки посылают предку разные сообщения по содержанию и размеру. Предок считывает сообщения от потомков и выводит их на экран. Предок ждет завершения своих потомков и анализирует код их завершения. На экран выводятся соответствующие сообщения.

Листинг 4 – Системный вызов pipe()

```
1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <unistd.h>
4
5 #include <stdio.h>
6 #include <string.h>
7
8 #define OK 0
9 #define ERROR 1
10 #define FORK_ERR -1
11 #define PIPE_ERR -1
12
13 #define TEXT1 "message 1\n"
14 #define TEXT2 "Maslova Marina IU7-53B\n"
15
16 #define BUF_SIZE 64
17
18 void check_status(const int status)
19 {
20     if (WIFEXITED(status))
21     {
22         printf("Child exited with code %d\n", WEXITSTATUS(status));
23     }
24     else if (WIFSIGNALED(status))
25     {
26         printf("Child was terminated by a signal with number %d\n",
27             WIFSIGNALED(status));
28     }
29     else if (WIFSTOPPED(status))
30     {
31         printf("Child was stopped by a signal with number %d\n",
32             WSTOPSIG(status));
33     }
34 }
```

```

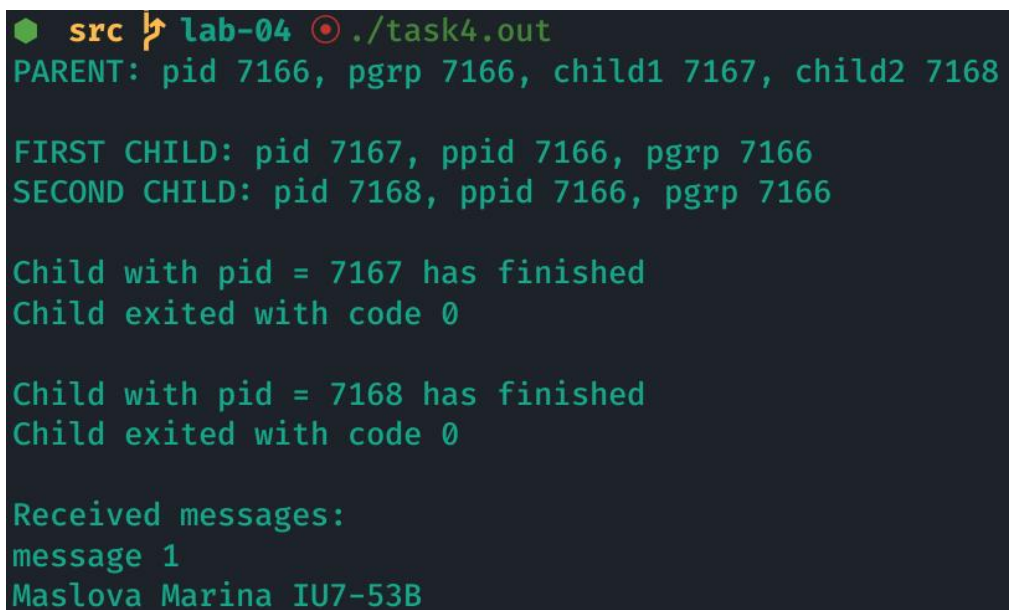
35
36 int main(void)
37 {
38     pid_t first_child_id, second_child_id, child_pid;
39     int status;
40     int fd[2];
41     char buf[BUF_SIZE] = "\0";
42
43     if (pipe(fd) == PIPE_ERR)
44     {
45         printf("Can't pipe!\n");
46         return ERROR;
47     }
48
49     if ((first_child_id = fork()) == FORK_ERR)
50     {
51         perror("Can't fork!\n");
52         return ERROR;
53     }
54
55     if (first_child_id == 0)
56     {
57         printf("FIRST CHILD: pid %d, ppid %d, pgrp %d\n",
58             getpid(), getppid(), getpgrp());
59
60         close(fd[0]);
61         write(fd[1], TEXT1, strlen(TEXT1));
62
63         return OK;
64     }
65
66     if ((second_child_id = fork()) == FORK_ERR)
67     {
68         perror("Can't fork!\n");
69         return ERROR;
70     }
71
72     if (second_child_id == 0)
73     {
74         printf("SECOND CHILD: pid %d, ppid %d, pgrp %d\n",
75             getpid(), getppid(), getpgrp());
76
77         close(fd[0]);
78         write(fd[1], TEXT2, strlen(TEXT2));
79
80         return OK;
81     }
82

```

```

83     printf("PARENT: pid %d, pgrp %d, child1 %d, child2 %d\n\n",
84           getpid(), getpgrp(), first_child_id, second_child_id);
85
86     child_pid = wait(&status);
87     printf("\nChild with pid = %d has finished\n", child_pid);
88     check_status(status);
89
90     child_pid = wait(&status);
91     printf("\nChild with pid = %d has finished\n", child_pid);
92     check_status(status);
93
94     close(fd[1]);
95
96     read(fd[0], buf, sizeof(buf));
97     printf("\nReceived messages:\n%s\n", buf);
98
99     return OK;
100 }

```



```

src lab-04 ./task4.out
PARENT: pid 7166, pgrp 7166, child1 7167, child2 7168

FIRST CHILD: pid 7167, ppid 7166, pgrp 7166
SECOND CHILD: pid 7168, ppid 7166, pgrp 7166

Child with pid = 7167 has finished
Child exited with code 0

Child with pid = 7168 has finished
Child exited with code 0

Received messages:
message 1
Maslova Marina IU7-53B

```

Рисунок 4 – Демонстрация работы программы (задание №4)

## Задание №5

Предок и потомки обмениваются сообщениями через неименованный программный канал. С помощью сигнала меняется ход выполнения программы. При получении сигнала потомки записывают сообщения в канал, если сигнал не поступает, то не записывают. Предок ждет завершения своих потомков и анализирует коды их завершений. На экран выводятся соответствующие сообщения.

Листинг 5 – Системный вызов signal()

```
1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <unistd.h>
4 #include <signal.h>
5
6 #include <stdio.h>
7 #include <string.h>
8 #include <stdbool.h>
9
10 #define OK 0
11 #define ERROR 1
12 #define FORK_ERR -1
13 #define PIPE_ERR -1
14
15 #define WAIT_SIGNAL_TIME 5
16
17 #define TEXT1 "message 1\n"
18 #define TEXT2 "Maslova Marina IU7-53B\n"
19
20 #define BUF_SIZE 64
21
22 _Bool flag = false;
23
24 void check_status(const int status)
25 {
26     if (WIFEXITED(status))
27     {
28         printf("Child exited with code %d\n", WEXITSTATUS(status));
29     }
30     else if (WIFSIGNALED(status))
31     {
32         printf("Child was terminated by a signal with number %d\n",
33             WIFSIGNALED(status));
34     }
35     else if (WIFSTOPPED(status))
```

```

36     {
37         printf("Child was stopped by a signal with number %d\n",
38             WSTOPSIG(status));
39     }
40 }
41
42 void switch_mode(int signal)
43 {
44     flag = true;
45 }
46
47 int main(void)
48 {
49     pid_t first_child_id, second_child_id, child_pid;
50     int status;
51     int fd[2];
52     char buf[BUF_SIZE] = "\0";
53
54     if (pipe(fd) == PIPE_ERR)
55     {
56         printf("Can't pipe!\n");
57         return ERROR;
58     }
59
60     signal(SIGINT, switch_mode);
61
62     if ((first_child_id = fork()) == FORK_ERR)
63     {
64         perror("Can't fork!\n");
65         return ERROR;
66     }
67
68     if (first_child_id == 0)
69     {
70         sleep(WAIT_SIGNAL_TIME);
71
72         if (flag)
73         {
74             close(fd[0]);
75             write(fd[1], TEXT1, strlen(TEXT1));
76         }
77
78         return OK;
79     }
80
81     if ((second_child_id = fork()) == FORK_ERR)
82     {
83         perror("Can't fork!\n");

```

```

84         return ERROR;
85     }
86
87     if (second_child_id == 0)
88     {
89         sleep(WAIT_SIGNAL_TIME);
90
91         if (flag)
92         {
93             close(fd[0]);
94             write(fd[1], TEXT2, strlen(TEXT2));
95         }
96
97         return OK;
98     }
99
100    printf("PARENT: pid %d, pgrp %d, child1 %d, child2 %d\n\n",
101           getpid(), getpgrp(), first_child_id, second_child_id);
102    printf("Press \"Ctrl+C\" to send messages\n");
103
104    child_pid = wait(&status);
105    printf("\nChild with pid = %d has finished\n", child_pid);
106    check_status(status);
107
108    child_pid = wait(&status);
109    printf("\nChild with pid = %d has finished\n", child_pid);
110    check_status(status);
111
112    close(fd[1]);
113
114    read(fd[0], buf, sizeof(buf));
115    printf("\nReceived messages:\n%s\n", buf);
116
117    return OK;
118 }

```

```
src lab-04 ./task5.out
PARENT: pid 10140, pgrp 10140, child1 10141, child2 10142

Press "Ctrl+C" to send messages

Child with pid = 10141 has finished
Child exited with code 0

Child with pid = 10142 has finished
Child exited with code 0

Received messages:
```

Рисунок 5 – Демонстрация работы программы, сигнал не поступает

```
src lab-04 ./task5.out
PARENT: pid 10234, pgrp 10234, child1 10235, child2 10236

Press "Ctrl+C" to send messages
^C
Child with pid = 10235 has finished
Child exited with code 0

Child with pid = 10236 has finished
Child exited with code 0

Received messages:
message 1
Maslova Marina IU7-53B
```

Рисунок 6 – Демонстрация работы программы, сигнал поступает



## Выполнение системных вызовов `fork()` и `exec()`

Системный вызов <code>fork()</code> выполняет следующие действия:	
1.	Резервируется пространство ящика для данных и стека процесса-потомка.
2.	Назначается идентификатор PID и структура <code>proc</code> потомка.
3.	Инициализируется структура <code>proc</code> потомка. Некоторые поля этой структуры копируются от процесса-родителя: идентификатор пользователя и группы, маск-символов и группа процессов. Часть полей инициализируется 0. Часть полей инициализируется специфическими для потомка значениями: PID потомка и его родителя, указатель на структуру <code>proc</code> родителя.
4.	Создаются карты трансляции адресов для процесса-потомка.
5.	Выделяется область и потомка и в нее копируется область и процесса-предка.
6.	Изменяются ссылки области и на первые карты адресации и пространство ящика.
7.	Потомок добавляется в набор процессов, которые разделяют область кода программы, выполняемой процессом-родителем.
8.	Пространство дублируются область данных и стека родителя и модифицируются карты адресации потомка.
9.	Потомок получает ссылки на разделяемые ресурсы, которые он наследует: открытые файлы (потомок наследует дескрипторы) и текущий рабочий каталог.
10.	Инициализируется аппаратный контекст потомка путем копирования регистров родителя.
11.	Поместить процесс-потомок в очередь готовых процессов.
12.	Возвращается PID в точку возврата из системного вызова в родительском процессе и 0 — в процессе-потомке.

Рисунок 7 – Системный вызов `fork()`



Системный вызов exec выполняет следующие действия:

1. Разбирает путь к исполняемому файлу и осуществляет доступ к нему.
2. Проверяет, имеет ли вызывающий процесс полномочия на выполнение файла.
3. Читает заголовок и проверяет, что он действительно исполняемый.
4. Если для файла установлен биты `SUID` или `SGID`, то эффективные идентификаторы `UID` и `GID` вызывающего процесса изменяет на `UID` и `GID` соответствующие ~~файлу~~ владельцу файла.
5. Копирует аргументы передаваемые в `exec` а также переменные среды в пространство ядра, после чего текущее пользовательское пространство готово к уничтожению.
6. Выделяет пространство свопинга для областей данных и стека.
7. Выбывает старое адресное пространство и связанное с ним пространство свопинга. Если же процесс был создан при помощи `fork`, производится возврат старого адресного пространства родительскому процессу.
8. Выделяет карты трансляции адресов для нового текста, данных и стека.
9. Устанавливает новое адресное пространство. Если область текста активна (какой-то другой процесс уже выполняет эту программу), то она будет совместно использоваться с этим процессом. В других случаях пространство должно инициализироваться из исполняемого файла. Процесс в системе `UNIX` обычно разбит на страницы, что означает, что каждая страница отбрасывается в память только по мере необходимости.
10. Копирует аргументы и переменные среды обратно в новый стек приложения.
11. Сбрасывает все обработчики сигналов в действия, определенные по умолчанию, так как функции обработчиков

Рисунок 8 – Системный вызов `exec()`

сигналов не существует в новой программе  
Сигналы которые были протестированы или  
заблокированы перед вызовом exes, остаются  
в тех же состояниях.

12. Инициализируется аппаратный процессор. При  
этом большинство регистров сбрасывается в 0,  
а указатель команд получает значение  
точки входа программы.

Рисунок 9 – Системный вызов exes() (продолжение)