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"

Тема Процессы. Системные вызовы fork() и exec()

Студент Хамзина Р. Р.

Группа ИУ7-53Б

Преподаватель Рязанова Н. Ю.

Москва — 2021 г.

Задание 1

При помощи системного вызова fork() создается два процесса-потомка. Для того, чтобы они завершились после процесса-предка, в них вызывается sleep(). При этом процессы-потомки становятся сиротами. Их усыновляет процесс-посредник systemd -user, который является потомком процесса с идентификатором 1.

Листинг 1 – Создание процессов-сирот

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/types.h>
4  #include <unistd.h>
5
6  #define PAUSE       4
7
8  #define TASK        "\n<<<<< Task 1 : Creating orphan processes
       >>>>>\n\n"
9
10
11 int main(void)
12 {
13     pid_t child_pid1, child_pid2;
14
15     if ((child_pid1 = fork()) == -1)
16     {
17         perror("\nCan't fork child 1.\n");
18         exit(1);
19     }
20     else if (child_pid1 == 0)
21     {
22         printf("\nBEFORE: \
23                 \nChild 1: PID = %d, PPID = %d, GPID = %d.\n",
                       getpid(), getppid(), getpgrp());
24
25         sleep(PAUSE);
26
27         printf("\nAFTER: \
```

```
28                  \nChild 1: PID = %d, PPID = %d, GPID = %d.\n",
                        getpid(), getppid(), getpgrp());
29
30          exit(0);
31      }
32      else
33      {
34          printf("\nParent: PID = %d, GPID = %d, child 1 PID =
              %d.\n", getpid(), getpgrp(), child_pid1);
35      }
36
37
38      if ((child_pid2 = fork()) == -1)
39      {
40          perror("\nCan't fork child 2.\n");
41          exit(1);
42      }
43      else if (child_pid2 == 0)
44      {
45          printf("\nBEFORE: \
46                  \nChild 2: PID = %d, PPID = %d, GPID = %d.\n",
                        getpid(), getppid(), getpgrp());
47
48          sleep(PAUSE);
49
50          printf("\nAFTER: \
51                  \nChild 2: PID = %d, PPID = %d, GPID = %d.\n",
                        getpid(), getppid(), getpgrp());
52
53          exit(0);
54      }
55      else
56      {
57          printf("\nParent: PID = %d, GPID = %d, child 2 PID =
              %d.\n", getpid(), getpgrp(), child_pid2);
58      }
59
60      return 0;
61  }
```

```
regina@regina-acer:~/bmstu/sem5/bmstu-os/sem5/lab_04/src$ ./main_01.exe

<<<<< Task 1 : Creating orphan processes >>>>>


Parent: PID = 55759, GPID = 55759, child 1 PID = 55760.

BEFORE:
Child 1: PID = 55760, PPID = 55759, GPID = 55759.

Parent: PID = 55759, GPID = 55759, child 2 PID = 55761.

BEFORE:
Child 2: PID = 55761, PPID = 55759, GPID = 55759.
regina@regina-acer:~/bmstu/sem5/bmstu-os/sem5/lab_04/src$
AFTER:
Child 1: PID = 55760, PPID = 1505, GPID = 55759.

AFTER:
Child 2: PID = 55761, PPID = 1505, GPID = 55759.
```

Рисунок 1 – Демонстрация работы программы

```
    1    1484    1484    1484 ?           -1 Ssl    120   0:00 /usr/bin/whoo
    1    1487    1487    1487 ?           -1 Ss     116   0:02 /usr/sbin/ker
    1    1496    1496    1496 ?           -1 Ss     116   0:02 /usr/sbin/ker
    1    1505    1505    1505 ?           -1 Ss    1000   0:01 /lib/systemd/
 1505    1506    1505    1505 ?           -1 S     1000   0:00 (sd-pam)
 1505    1511    1511    1511 ?           -1 S<sl  1000  24:11 /usr/bin/puls
 1505    1513    1513    1513 ?           -1 SNsl  1000   0:00 /usr/libexec/
```

Рисунок 2 – Процесс, усыновивший процессы-сироты

Задание 2

Системный вызов wait() блокирует процесс-предок, поэтому он ждет завершения процессов-потомков. Процесс-предок анализирует коды завершения процессов-потомков.

Листинг 2 – Системный вызов wait()

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/wait.h>
5 #include <unistd.h>
6
7 #define PAUSE       4
8
9 #define TASK        "\n<<<<< Task 2 : Parent is waiting childs
      with wait() >>>>>\n\n"
10
11 void check_status(int status)
12 {
13     if (WIFEXITED(status))
14     {
15         printf("Child exited correctly with code %d.\n",
             WEXITSTATUS(status));
16
17         return;
18     }
19     else if (WIFSIGNALED(status))
20     {
21         printf("Child exited with non-interceptable signal
             %d.\n", WTERMSIG(status));
22
23         return;
24     }
25     else if (WIFSTOPPED(status))
26     {
27         printf("Child stopped with signal %d.\n",
             WSTOPSIG(status));
28
29         return;
30     }
31 }
32
33 int main(void)
34 {
35     printf(TASK);
36
```

```
37      pid_t child_pid1, child_pid2, child_pid;
38      int status;
39
40      if ((child_pid1 = fork()) == -1)
41      {
42          perror("\nCan't fork child 1.\n");
43          exit(1);
44      }
45      else if (child_pid1 == 0)
46      {
47          printf("\nBEFORE: \
48                  \nChild 1: PID = %d, PPID = %d, GPID = %d.\n",
                       getpid(), getppid(), getpgrp());
49
50          sleep(PAUSE);
51
52          printf("\nAFTER: \
53                  \nChild 1: PID = %d, PPID = %d, GPID = %d.\n",
                       getpid(), getppid(), getpgrp());
54
55          exit(0);
56      }
57      else
58      {
59          child_pid = wait(&status);
60          printf("\nChild 1 has fihished: PID = %d, status =
              %d.\n", child_pid, status);
61
62          printf("\nParent: PID = %d, GPID = %d, child 1 PID =
              %d.\n", getpid(), getpgrp(), child_pid1);
63          check_status(status);
64      }
65
66
67      if ((child_pid2 = fork()) == -1)
68      {
69          perror("\nCan't fork child 2.\n");
70          exit(1);
71      }
72      else if (child_pid2 == 0)
73      {
```

```
74          printf("\n\n\nBEFORE: \
75                  \nChild 2: PID = %d, PPID = %d, GPID = %d.\n",
                       getpid(), getppid(), getpgrp());
76
77          sleep(PAUSE);
78
79          printf("\nAFTER: \
80                  \nChild 2: PID = %d, PPID = %d, GPID = %d.\n",
                       getpid(), getppid(), getpgrp());
81
82          exit(0);
83      }
84      else
85      {
86          child_pid = wait(&status);
87          printf("\nChild 2 has fihished: PID = %d, status =
              %d.\n", child_pid, status);
88
89          printf("\nParent: PID = %d, GPID = %d, child 2 PID =
              %d.\n", getpid(), getpgrp(), child_pid1);
90          check_status(status);
91      }
92
93      return 0;
94  }
```

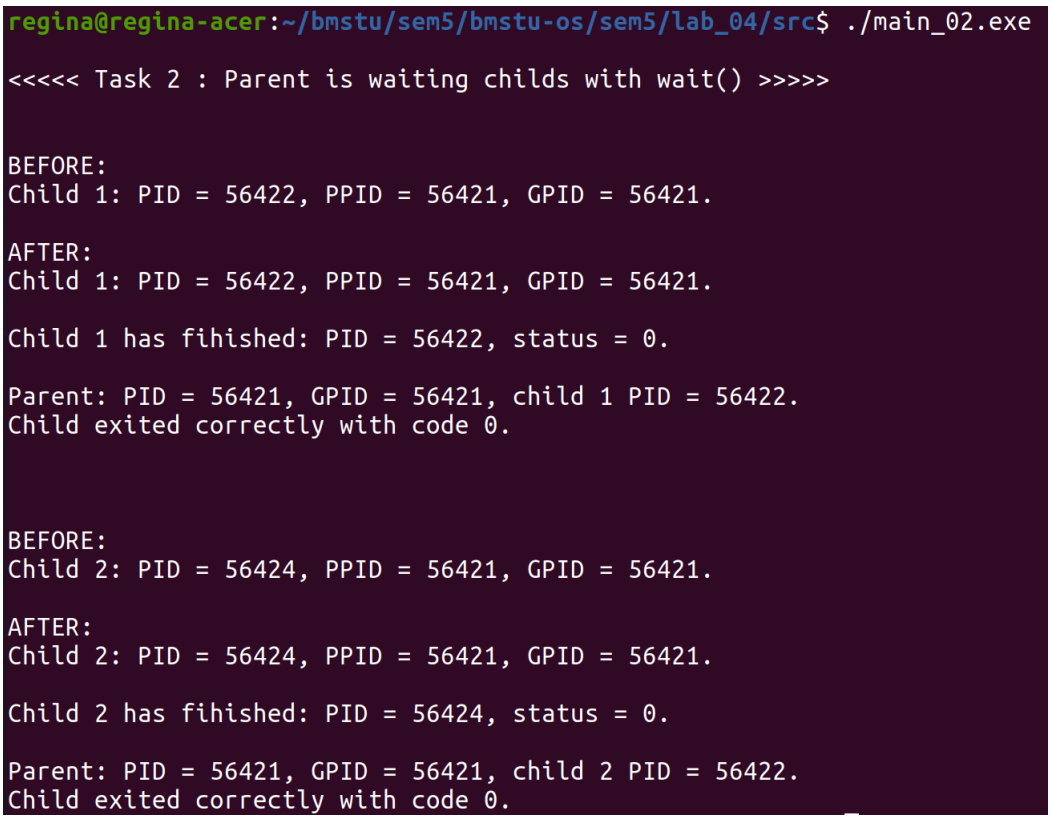


Рисунок 3 – Демонстрация работы программы

Задание 3

Процессы-потомки переходят на выполнение следующих программ:

- в первой программе (./sort.exe) выполняется чтение массива целых чисел из файла (array.txt), сортировка массива и его вывод на экран (лабораторная работа по курсу программирования на языке C);
- во второй программе (./palindrome.exe) выполняется чтение последовательности символов из файла (./string) и вывод сообщения о том, является ли введенная последовательность палиндромом (лабораторная работа по курсу программирования на языке C).

Программы передаются системному вызову execlp() в качестве параметра.

Листинг 3 – Системный вызов execlp

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/wait.h>
5 #include <unistd.h>
6
7 #define TASK        "\n<<<<< Task 3 : Children do other programs
      with execlp() >>>>>\n\n"
8
9
10 void check_status(int status)
11 {
12     if (WIFEXITED(status))
13     {
14         printf("Child exited correctly with code %d.\n",
             WEXITSTATUS(status));
15
16         return;
17     }
18     else if (WIFSIGNALED(status))
19     {
20         printf("Child exited with non-interceptable signal
             %d.\n", WTERMSIG(status));
21
22         return;
23     }
24     else if (WIFSTOPPED(status))
25     {
26         printf("Child stopped with signal %d.\n",
             WSTOPSIG(status));
27
28         return;
29     }
30 }
31
32
33 int main(void)
34 {
35     printf(TASK);
36
```

```
37      pid_t child_pid1, child_pid2, child_pid;
38      int status;
39
40      if ((child_pid1 = fork()) == -1)
41      {
42          perror("\nCan't fork child 1.\n");
43          exit(1);
44      }
45      else if (child_pid1 == 0)
46      {
47          printf("\nChild 1 START: PID = %d, PPID = %d, GPID =
              %d.\n", getpid(), getppid(), getpgrp());
48          printf("\n- to sort array\n");
49
50          if (execlp("./sort.exe", "./sort.exe", "array.txt", NULL)
              == -1)
51          {
52              printf("\nERROR: child 1 can not execute exec().\n");
53
54              exit(1);
55          }
56
57          exit(0);
58      }
59      else
60      {
61          child_pid = wait(&status);
62          printf("\nChild 1 END: PID = %d, status = %d.\n",
              child_pid, status);
63
64          printf("\nParent: PID = %d, GPID = %d, child 1 PID =
              %d.\n", getpid(), getpgrp(), child_pid1);
65          check_status(status);
66      }
67
68
69      if ((child_pid2 = fork()) == -1)
70      {
71          perror("\nCan't fork child 2.\n");
72          exit(1);
73      }
```

```
74      else if (child_pid2 == 0)
75      {
76          printf("\n\n\nChild 2 START: PID = %d, PPID = %d, GPID =
              %d.\n", getpid(), getppid(), getpgrp());
77          printf("\n- to check string is palindrome\n");
78
79          if (execlp("./palindrome.exe", "./palindrome.exe",
              "string.txt", NULL) == -1)
80          {
81              printf("\nERROR: child 2 can not execute exec().\n");
82
83              exit(1);
84          }
85
86          exit(0);
87      }
88      else
89      {
90          child_pid = wait(&status);
91          printf("\nChild 2 END: PID = %d, status = %d\n",
              child_pid, status);
92
93          printf("\nParent: PID = %d, GPID = %d, child 2 PID =
              %d\n", getpid(), getpgrp(), child_pid1);
94          check_status(status);
95      }
96
97      return 0;
98  }
```

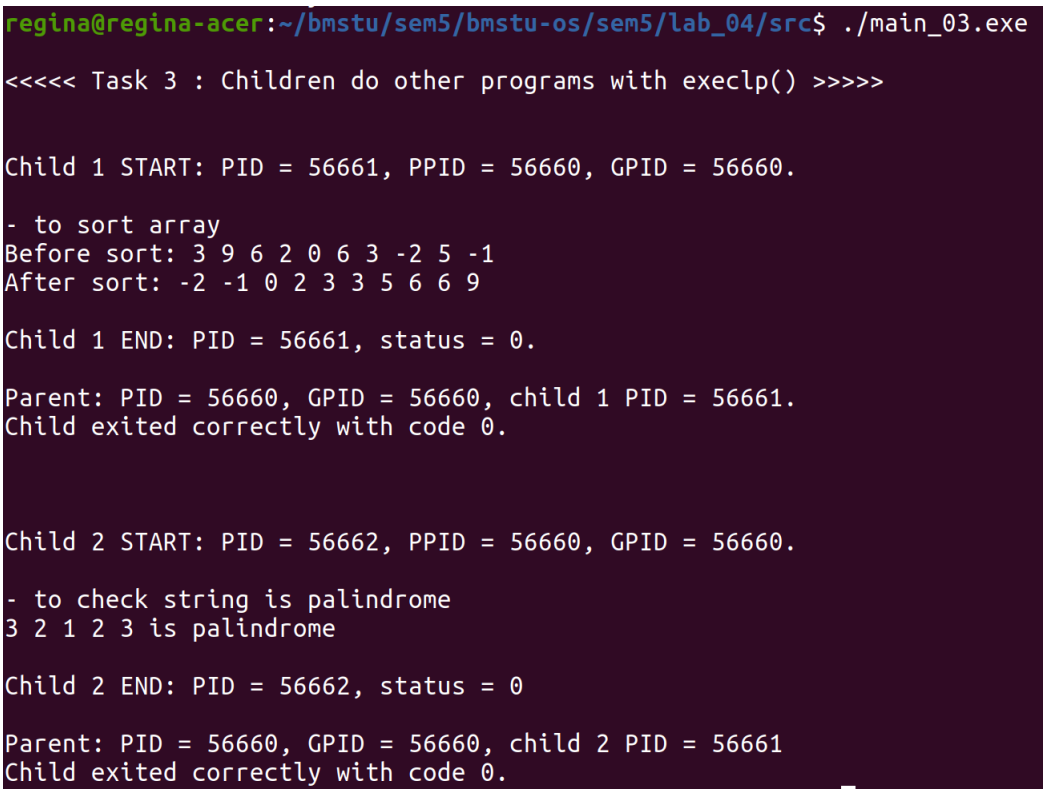


Рисунок 4 – Демонстрация работы программы

Задание 4

Процессы-потомки пишут разные сообщения в один неименованный программный канал, созданный системным вызовом pipe(). Процесс-предок считывает сообщения процессов-потомков и выводит сообщения на экран.

Листинг 4 – Системный вызов pipe()

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/wait.h>
5 #include <unistd.h>
6 #include <string.h>
7
8 #define TASK        "\n<<<<< Task 4 : Messaging with pipe()
     >>>>>\n\n"
9
10 #define MSG1  "Hello, child 2.\n"
11 #define MSG2  "How are you, child 1?\n"
12 #define LEN   50
13
14
15 void check_status(int status)
16 {
17     if (WIFEXITED(status))
18     {
19         printf("Child exited correctly with code %d.\n",
             WEXITSTATUS(status));
20
21         return;
22     }
23     else if (WIFSIGNALED(status))
24     {
25         printf("Child exited with non-interceptable signal
             %d.\n", WTERMSIG(status));
26
27         return;
28     }
29     else if (WIFSTOPPED(status))
30     {
31         printf("Child stopped with signal %d.\n",
             WSTOPSIG(status));
32
33         return;
34     }
35 }
36
```

```
37 int main(void)
38 {
39     printf(TASK);
40
41     int fd[2];
42
43     pid_t child_pid1, child_pid2, child_pid;
44     int status;
45
46     char msgs[LEN] = {0};
47
48     if (pipe(fd) == -1)
49     {
50         perror("\nCan't pipe.\n");
51         exit(1);
52     }
53
54     if ((child_pid1 = fork()) == -1)
55     {
56         perror("\nCan't fork child 1.\n");
57         exit(1);
58     }
59     else if (child_pid1 == 0)
60     {
61         printf("\nChild 1: PID = %d, PPID = %d, GPID = %d.\n",
               getpid(), getppid(), getpgrp());
62
63         close(fd[0]);
64         write(fd[1], MSG1, strlen(MSG1));
65
66         exit(0);
67     }
68
69
70     if ((child_pid2 = fork()) == -1)
71     {
72         perror("\nCan't fork child 2.\n");
73         exit(1);
74     }
75     else if (child_pid2 == 0)
76     {
```

```
77          printf("Child 2: PID = %d, PPID = %d, GPID = %d.\n",
                getpid(), getppid(), getpgrp());
78
79          close(fd[0]);
80          write(fd[1], MSG2, strlen(MSG2));
81
82          exit(0);
83      }
84
85      child_pid = wait(&status);
86      printf("\n\nChild 1 has fihished: PID = %d, status = %d.\n",
            child_pid, status);
87      check_status(status);
88
89      child_pid = wait(&status);
90      printf("\nChild 2 has fihished: PID = %d, status = %d.\n",
            child_pid, status);
91      check_status(status);
92
93      close(fd[1]);
94      read(fd[0], msgs, LEN);
95      printf("\nChilds wrote :\n%s\n", msgs);
96
97      return 0;
98  }
```

```
regina@regina-acer:~/bmstu/sem5/bmstu-os/sem5/lab_04/src$ ./main_04.exe

<<<<< Task 4 : Messaging with pipe() >>>>>


Child 1: PID = 17699, PPID = 17698, GPID = 17698.
Child 2: PID = 17700, PPID = 17698, GPID = 17698.


Child 1 has fihished: PID = 17699, status = 0.
Child exited correctly with code 0.

Child 2 has fihished: PID = 17700, status = 0.
Child exited correctly with code 0.

Childs wrote :
Hello, child 2.
How are you, child 1?
```

Рисунок 5 – Демонстрация работы программы

Задание 5

Процесс-предок и процессы-потомки обмениваются сообщениями аналогично заданию 4. При помощи сигнала меняется ход выполнения программы: при получении сигнала от нажатия Ctrl + Z первый процесс-потомок записывает сообщение в канал, при получении сигнала от нажатия Ctrl + C второй процесс-потомок записывает сообщение в канал.

Листинг 5 – Системный вызов signal()

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/wait.h>
5 #include <unistd.h>
6 #include <string.h>
7 #include <signal.h>
8
9 #define PAUSE       5
10
11 #define TASK        "\n<<<<< Task 5 : Signal handler with
       signal() >>>>>\n\n"
12
13 #define MSG1  "Hello, child 2.\n"
14 #define MSG2  "How are you, child 1?\n"
15 #define LEN   50
16
17 #define SIGNAL_MSG "\nPress: CTRL + Z - for msgs from childs\n"
18
19
20 int flag = 0;
21
22
23 void check_status(int status)
24 {
25     if (WIFEXITED(status))
26     {
27         printf("Child exited correctly with code %d.\n",
              WEXITSTATUS(status));
28
```

```
29          return;
30      }
31      else if (WIFSIGNALED(status))
32      {
33          printf("Child exited with non-interceptable signal
              %d.\n", WTERMSIG(status));
34
35          return;
36      }
37      else if (WIFSTOPPED(status))
38      {
39          printf("Child stopped with signal %d.\n",
              WSTOPSIG(status));
40
41          return;
42      }
43  }
44
45
46  void catch_ctrlz(int signal)
47  {
48      flag = 1;
49      printf("\nCatched signal = %d.\n", signal);
50  }
51
52
53  int main(void)
54  {
55      printf(TASK);
56      printf(SIGNAL_MSG);
57
58      signal(SIGTSTP, catch_ctrlz);
59      sleep(PAUSE);
60
61      int fd[2];
62
63      pid_t child_pid1, child_pid2, child_pid;
64      int status;
65
66      char msgs[LEN] = {0};
67
```

```
68      if (pipe(fd) == -1)
69      {
70          perror("\nCan't pipe.\n");
71          exit(1);
72      }
73
74
75      if ((child_pid1 = fork()) == -1)
76      {
77          perror("\nCan't fork child 1.\n");
78          exit(1);
79      }
80      else if (child_pid1 == 0)
81      {
82          printf("\nChild 1: PID = %d, PPID = %d, GPID = %d.\n",
               getpid(), getppid(), getpgrp());
83
84          if (flag == 1)
85          {
86              close(fd[0]);
87              write(fd[1], MSG1, strlen(MSG1));
88          }
89
90          exit(0);
91      }
92
93
94      if ((child_pid2 = fork()) == -1)
95      {
96          perror("\nCan't fork child 2.\n");
97          exit(1);
98      }
99      else if (child_pid2 == 0)
100     {
101         printf("Child 2: PID = %d, PPID = %d, GPID = %d.\n",
               getpid(), getppid(), getpgrp());
102
103         if (flag == 1)
104         {
105             close(fd[0]);
106             write(fd[1], MSG2, strlen(MSG2));
```

```
107         }
108
109         exit(0);
110     }
111
112     child_pid = wait(&status);
113     printf("\n\nChild 1 has fihished: PID = %d, status = %d.\n",
            child_pid, status);
114     check_status(status);
115
116     child_pid = wait(&status);
117     printf("\nChild 2 has fihished: PID = %d, status = %d.\n",
            child_pid, status);
118     check_status(status);
119
120     close(fd[1]);
121     read(fd[0], msgs, LEN);
122     printf("\nChilds wrote :\n%s\n", msgs);
123
124     return 0;
125 }
```

```
regina@regina-acer:~/bmstu/sem5/bmstu-os/sem5/lab_04/src$ ./main_05.exe

<<<<< Task 5 : Signal handler with signal() >>>>>


Press: CTRL + Z - for msgs from childs
^Z
Catched signal = 20.

Child 1: PID = 20931, PPID = 20930, GPID = 20930.
Child 2: PID = 20932, PPID = 20930, GPID = 20930.


Child 1 has fihished: PID = 20931, status = 0.
Child exited correctly with code 0.

Child 2 has fihished: PID = 20932, status = 0.
Child exited correctly with code 0.

Childs wrote :
Hello, child 2.
How are you, child 1?
```

Рисунок 6 – Демонстрация работы программы: сигнал от Ctrl + Z

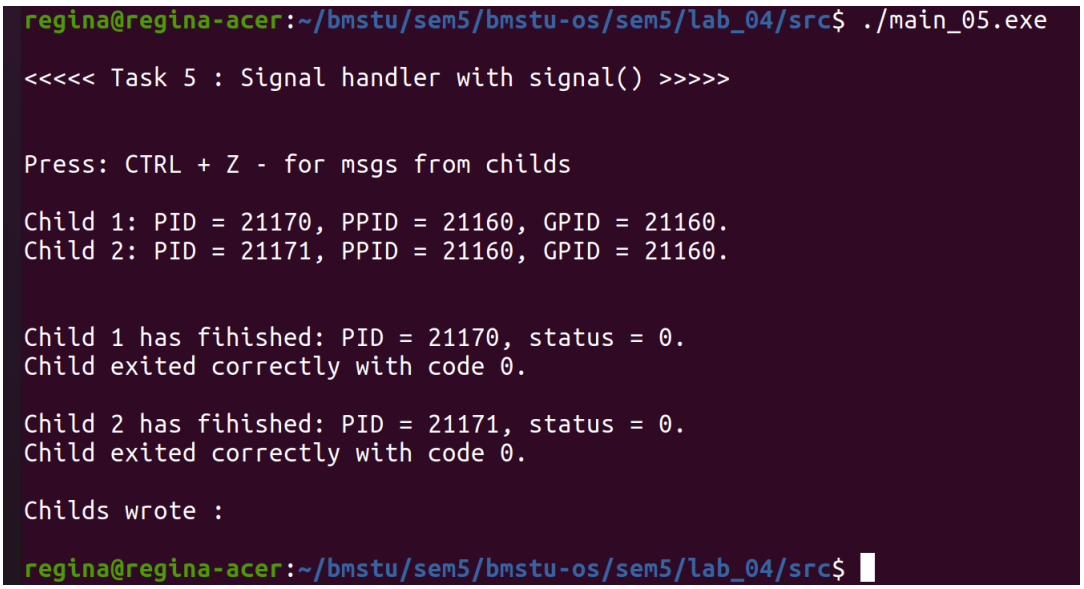
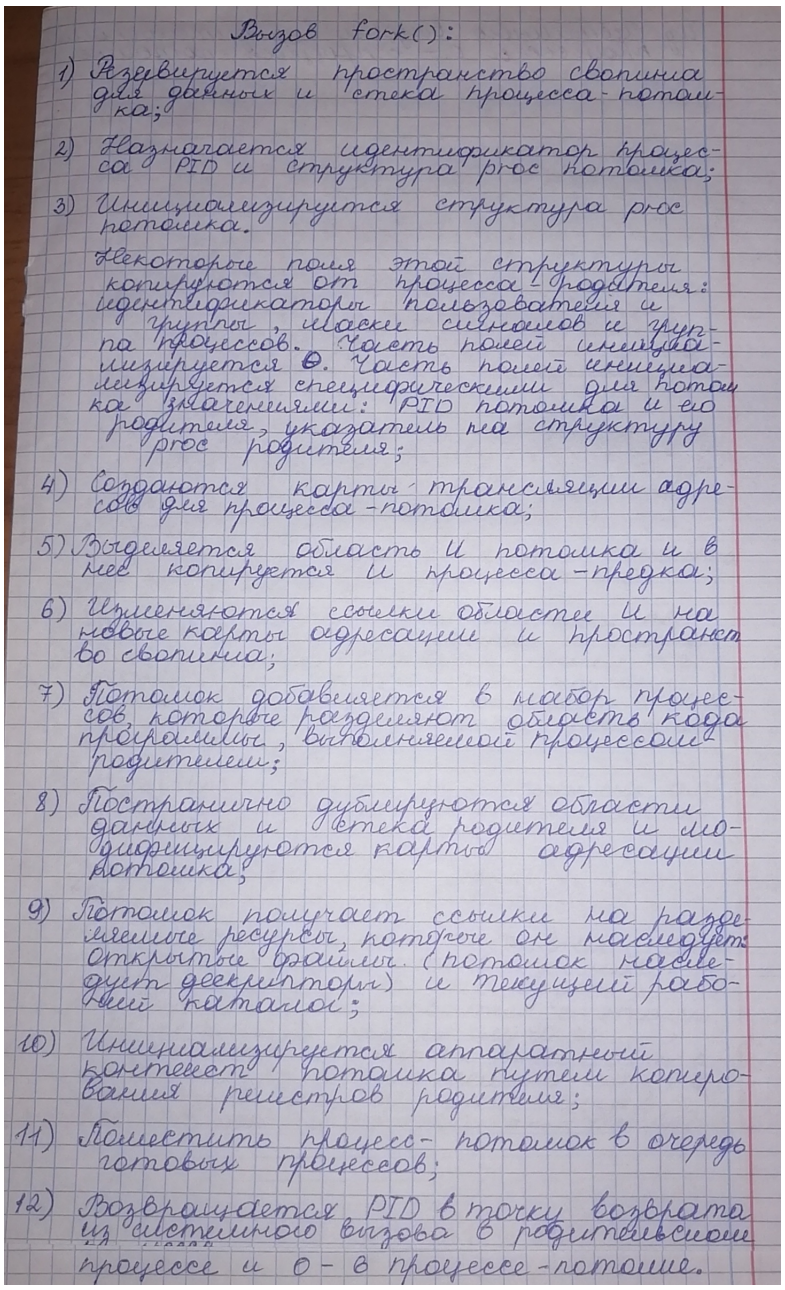


Рисунок 7 – Демонстрация работы программы: сигнал не вызван

Последовательность действий при вызове fork()



Вызов fork():

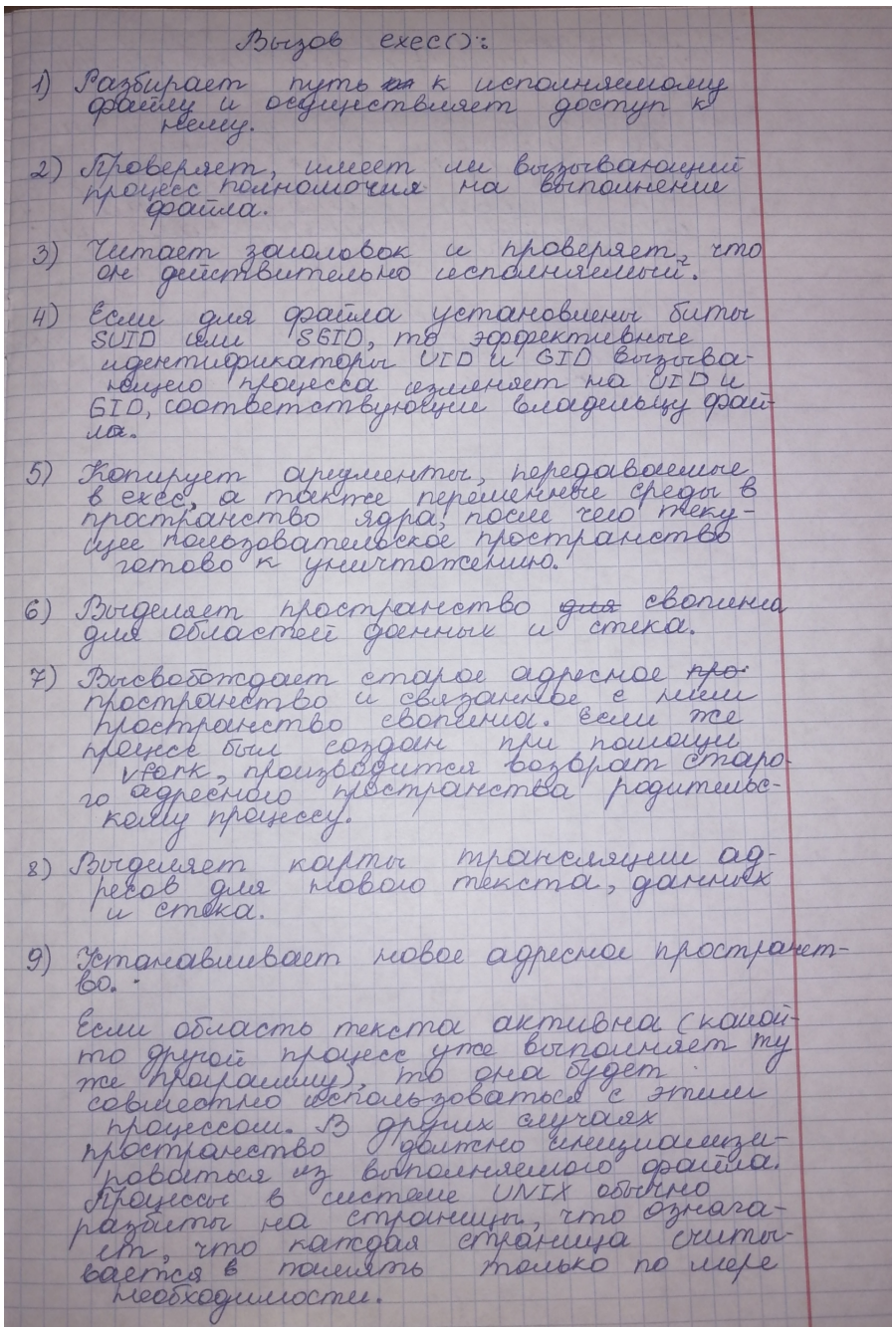
1) Резервируется пространство свопинга для данных и стека процесса-потомка;
2) Назначается идентификатор процесса PID и структура proc потомка;
3) Инициализируется структура proc потомка.

Некоторые поля этой структуры копируются от процесса-родителя: идентификаторы пользователя и группы, маски сигналов и группа процессов. Часть полей инициализируется 0. Часть полей инициализируется специфическими для потомка значениями: PID потомка и его родителя, указатель на структуру proc родителя;

4) Создаются карты трансляции адресов для процесса-потомка;
5) Выделяется область U потомка и в нее копируется U процесса-предка;
6) Изменяются ссылки областей U на новые карты адресации и пространство свопинга;
7) Потомок добавляется в набор процессов, которые разделяют область кода программы, выполняемой процессом-родителем;
8) Постранично дублируются области 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 8 – Вызов fork()

Последовательность действий при вызове exec()



Вызов exec():

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 установлены биты SUID или SGID, то эффективные идентификаторы UID и GID вызывающего процесса изменяет на UID и GID, соответствующие 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 для свопинга для областей данных и стека.

7) Высвобождает старое адресное пространство и связанное с ним пространство свопинга. Если же процесс был создан при помощи vfork, 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 ту же программу), то она будет совместно использоваться с этими процессами. В других случаях пространство должно специализироваться из выполняемого файла. Процессы в системе UNIX обычно разбиты на страницы, что означает, что каждая страница считывается в память только по мере необходимости.

Рисунок 9 – Вызов exec() - 1

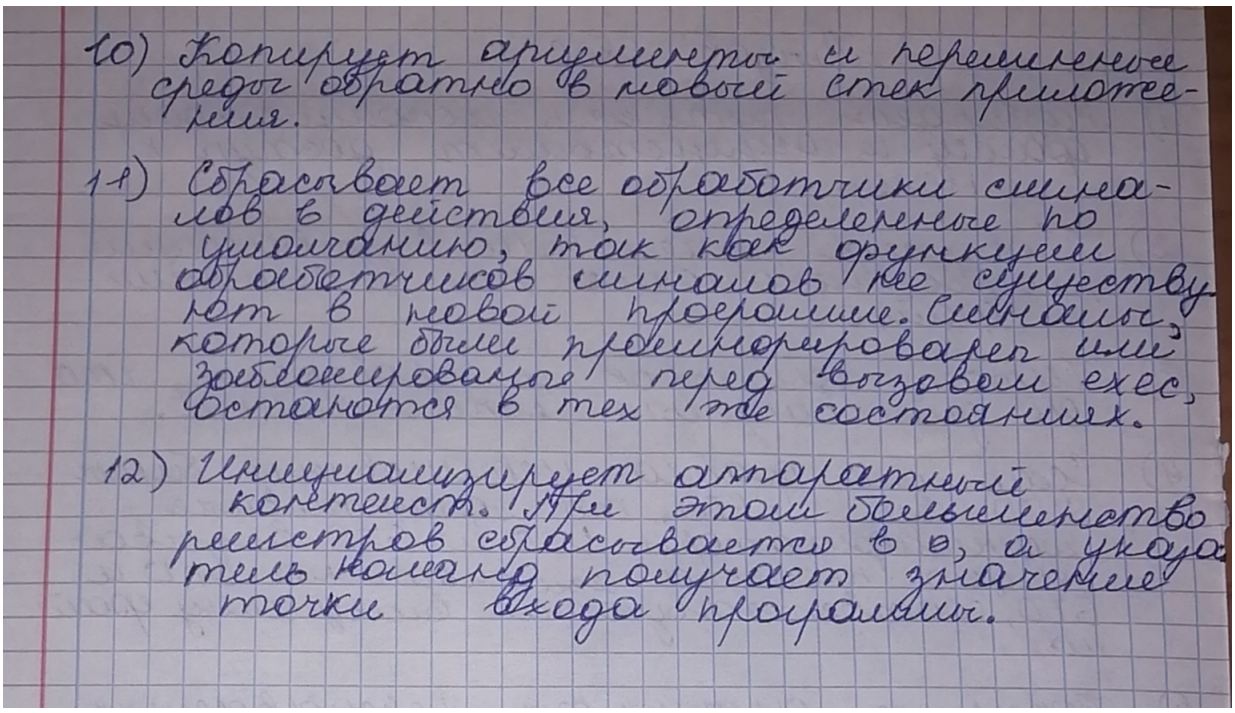


Рисунок 10 – Вызов exec() - 2