

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Преподаватели Рязанова Н. Ю.

Задание №1

Процессы-сироты. В программе создаются не менее двух потомков системным вызовом `fork()`. В потомках вызывается `sleep()`, чтобы предок гарантированно завершился раньше своих потомков. В предке вывести собственный идентификатор, идентификатор группы и идентификаторы потомков. В процессе-потомке вывести собственный идентификатор, идентификатор предка и идентификатор группы. Убедиться, что при завершении процесса-предка потомок, который продолжает выполняться, получает идентификатор предка (PPID), равный 1 или идентификатор процесса-посредника. Продемонстрировать с помощью соответствующего вывода информацию об идентификаторах процессов и их группе. Продемонстрировать «усыновление». Для этого надо в потомках вывести идентификаторы: собственный, предка, группы до блокировки и после блокировки.

В программе добавлен `sleep` в предке перед его завершением для того, чтобы предок не завершился до того, как будет выведена информация о процессах-потомках до блокировки. При этом время блокировки предка меньше, чем время блокировки потомков, поэтому предок гарантированно завершится раньше своих потомков.

Текст программы приведён на листинге 1.

Листинг 1: Процессы-сироты

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #define ERROR_FORK 1
4 #define OK 0
5 int main() {
6     int childpids[2];
7     for (int i = 0; i < 2; i++) {
8         int pid = fork();
9
10        if (pid == -1) {
11            return ERROR_FORK;
12        }
13
14        if (pid == 0) {
15            printf("\nCHILD    %d BEFORE BLOCK: pid: %d, ppid: %d, grp: %d\n",
16                  i + 1, getpid(), getppid(), getpgrp());
17            sleep(2);
18            printf("\nCHILD    %d AFTER BLOCK: pid: %d, ppid: %d, grp: %d\n",
19                  i + 1, getpid(), getppid(), getpgrp());
20            return OK;
21        }
22
23        childpids[i] = pid;
24    }
25    printf("PARENT: pid: %d grp: %d, child's pids: %d, %d\n", getpid(),
26          getpgrp(), childpids[0], childpids[1]);
27    sleep(1);
28    return OK;
29 }
```

На рисунке 1 приведён результат работы программы. Видно, что до завершения процесса-предка `ppid` у потомков был равен идентификатору предка. Затем, когда процесс-предок завершился, потомки были "усыновлены" процессом с идентификатором 1.

```
kate@MacBook-Pro-Ekaterina ~/D/o/l/s/lab_04_01 (master)> ./app
PARENT: pid: 15695 grp: 15695, child's pids: 15696, 15697

CHILD №1 BEFORE BLOCK: pid: 15696, ppid: 15695, grp: 15695

CHILD №2 BEFORE BLOCK: pid: 15697, ppid: 15695, grp: 15695
kate@MacBook-Pro-Ekaterina ~/D/o/l/s/lab_04_01 (master)>
CHILD №1 AFTER BLOCK: pid: 15696, ppid: 1, grp: 15695

CHILD №2 AFTER BLOCK: pid: 15697, ppid: 1, grp: 15695
```

Рис. 1: Демонстрация работы программы (задание №1).

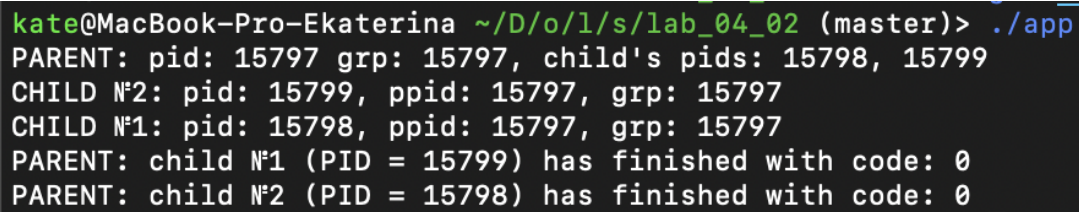
Задание №2

Предок ждет завершения своих потомков, используя системный вызов `wait()`. Вывод соответствующих сообщений на экран. В программе необходимо, чтобы предок выполнял анализ кодов завершения потомков. Текст программы приведён на листинге 2.

Листинг 2: Системный вызов `wait()`

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/types.h>
4 #include <sys/wait.h>
5 #define ERROR_FORK 1
6 #define OK 0
7 int main() {
8     int chldpids[2];
9     for (int i = 0; i < 2; i++) {
10         int pid = fork();
11         if (pid == -1) {
12             return ERROR_FORK;
13         }
14         if (pid == 0){
15             sleep(2);
16             printf("CHILD    %d: pid: %d, ppid: %d, grp: %d\n", i + 1,
17                    getpid(), getppid(), getpgrp());
18             return OK;
19         }
20         chldpids[i] = pid;
21     }
22     printf("PARENT: pid: %d grp: %d, child's pids: %d, %d\n", getpid(),
23            getpgrp(), chldpids[0], chldpids[1]);
24     for (int i = 0; i < 2; i++)
25     {
26         int status;
27         pid_t chldpid = wait(&status);
28         if (WIFEXITED(status)) {
29             printf("PARENT: child    %d (PID = %d) has finished with code: %d\n", i + 1, chldpid, WEXITSTATUS(status));
30         }
31         else if (WIFSIGNALED(status)) {
32             printf("PARENT: child    %d (PID = %d) has finished because of signal: %d\n", i + 1, chldpid, WTERMSIG(status));
33         }
34         else if (WIFSTOPPED(status)) {
35             printf("PARENT: child    %d (PID = %d) has been stopped because of signal: %d\n", i + 1, chldpid, WSTOPSIG(status));
36         }
37     }
38     return OK;
39 }
```

Результат работы программы приведён на рисунке 2. Видно, что в отличие от программы из первого задания процесс-предок дождался завершения дочерних процессов, о чём свидетельствуют коды завершения дочерних процессов, перехваченные процессом-предком.

A terminal window with a black background and white text. The prompt is 'kate@MacBook-Pro-Ekaterina ~/D/o/l/s/lab_04_02 (master)>'. The command './app' has been executed. The output shows the parent process (pid: 15797, grp: 15797) spawning two child processes. Child #2 (pid: 15799, ppid: 15797, grp: 15797) finishes first with code 0. Then child #1 (pid: 15798, ppid: 15797, grp: 15797) finishes with code 0. The parent process then prints the completion status for both children.

```
kate@MacBook-Pro-Ekaterina ~/D/o/l/s/lab_04_02 (master)> ./app
PARENT: pid: 15797 grp: 15797, child's pids: 15798, 15799
CHILD №2: pid: 15799, ppid: 15797, grp: 15797
CHILD №1: pid: 15798, ppid: 15797, grp: 15797
PARENT: child №1 (PID = 15799) has finished with code: 0
PARENT: child №2 (PID = 15798) has finished with code: 0
```

Рис. 2: Демонстрация работы программы (задание №2).

Задание №3

Потомки переходят на выполнение других программ, которые передаются системному вызову `exec()` в качестве параметра. Потомки должны выполнять разные программы. Прерок ждет завершения своих потомков с анализом кодов завершения. На экран выводятся соответствующие сообщения. Текст программы приведён на листинге 3.

Листинг 3: Системный вызов `exec()`

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/types.h>
4 #include <sys/wait.h>
5 #define ERROR_FORK 1
6 #define ERROR_EXEC 2
7 #define OK 0
8 int main() {
9     int childpids[2];
10    for (int i = 0; i < 2; i++) {
11        int pid = fork();
12        if (pid == -1) {
13            return ERROR_FORK;
14        }
15        if (pid == 0) {
16            printf("CHILD    %d: pid: %d, ppid: %d, grp: %d\n", i + 1,
17                  getpid(), getppid(), getpgrp());
18            if (i == 0) execl("sort", "sort", "1", "3", "2", "0", "4", "5",
19                             NULL);
20            else execl("max", "max", "1", "3", "2", "0", "4", "5", NULL);
21            return ERROR_EXEC;
22        }
23        childpids[i] = pid;
24    }
25    printf("PARENT: pid: %d grp: %d, child's pids: %d, %d\n", getpid(),
26          getpgrp(), childpids[0], childpids[1]);
27    for (int i = 0; i < 2; i++) {
28        int status;
29        pid_t childpid = wait(&status);
30        if (WIFEXITED(status)) {
31            printf("PARENT: child    %d (PID = %d) has finished with code: %d\n", i + 1, childpid, WEXITSTATUS(status));
32        } else if (WIFSIGNALED(status)) {
33            printf("PARENT: child    %d (PID = %d) has finished because of signal: %d\n", i + 1, childpid, WTERMSIG(status));
34        } else if (WIFSTOPPED(status)) {
35            printf("PARENT: child    %d (PID = %d) has been stopped because of signal: %d\n", i + 1, childpid, WSTOPSIG(status));
36        }
37    }
38    return OK; }
```

Результат работы программы приведён на рисунке 3. Видно, что в отличие от программы из второго задания дочерние процессы переходят на выполнение других программ, которые передаются системному вызову `exec()` в качестве параметра. В данном случае выполняются программы "sort" и "max" с аргументами в виде элементов обрабатываемого массива (массив: [1, 3, 2, 0, 4, 5]). Программа sort выводит отсортированный массив, программа max выводит максимальный элемент в массиве. Исходные коды программ сортировки и поиска максимума приведены в приложениях в листингах 6 и 7 соответственно.

```
kate@MacBook-Pro-Ekaterina ~/D/o/l/s/lab_04_03 (master)> ./app
PARENT: pid: 18673 grp: 18673, child's pids: 18674, 18675
CHILD №1: pid: 18674, ppid: 18673, grp: 18673
CHILD №2: pid: 18675, ppid: 18673, grp: 18673
max in array: 5
sorted array: 0 1 2 3 4 5
PARENT: child №1 (PID = 18675) has finished with code: 0
PARENT: child №2 (PID = 18674) has finished with code: 0
```

Рис. 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 должны посылать 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. Вывод соответствующих сообщений на экран. Текст программы приведён на листинге 4.

Листинг 4: Программные каналы

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/types.h>
4 #include <sys/wait.h>
5 #include "string"
6 #define ERROR_FORK 1
7 #define ERROR_EXEC 2
8 #define ERROR_PIPE 3
9 #define OK 0
10 int main() {
11     int fd[2];
12     const char *messages[2] = { "msg\n", "msg msg\n"};
13     int len[2] = {strlen(messages[0]), strlen(messages[1])};
14     if (pipe(fd) == -1) {
15         return ERROR_PIPE;
16     }
17     int childpids[2];
18     for (int i = 0; i < 2; i++) {
19         int pid = fork();
20         if (pid == -1) {
21             return ERROR_FORK;
22         }
23
24         if (pid == 0) {
25             close(fd[0]);
26             write(fd[1], messages[i], len[i]);
27             printf("CHILD    %d (pid: %d, ppid: %d, grp: %d) sent message to\n", i + 1, getpid(), getppid(), getpgrp());
28             return OK;
29         }
30         childpids[i] = pid;
31     }
32     printf("PARENT: pid: %d grp: %d, child's pids: %d, %d\n", getpid(), getpgrp(), childpids[0], childpids[1]);
33     for (int i = 0; i < 2; i++) {
34         int status;
35         pid_t childpid = wait(&status);
36         if (WIFEXITED(status)) {
37             printf("PARENT: child    %d (PID = %d) has finished with code: %d\n", i + 1, childpid, WEXITSTATUS(status));
38         }
39     }
```

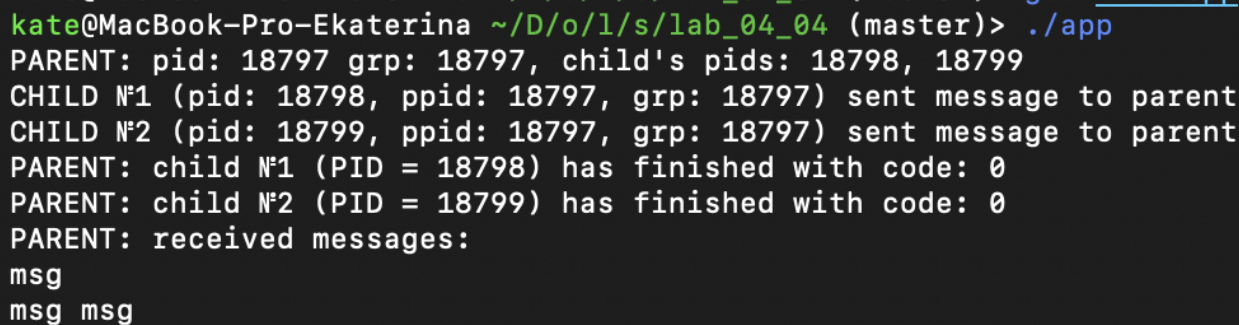


```

38         d\n", i + 1, childpid, WEXITSTATUS(status));
39     } else if (WIFSIGNALED(status)) {
40         printf("PARENT: child    %d (PID = %d) has finished because of
41             signal: %d\n", i + 1, childpid, WTERMSIG(status));
42     } else if (WIFSTOPPED(status)) {
43         printf("PARENT: child    %d (PID = %d) has been stopped because
44             of signal: %d\n", i + 1, childpid, WSTOPSIG(status));
45     }
46 }
47 char buf[len[0] + len[1]];
48 close(fd[1]);
49 read(fd[0], buf, len[0] + len[1]);
50 buf[len[0] + len[1]] = '\0';
51 printf("PARENT: received messages:\n%s", buf);
52 return OK;
53 }

```

Результат работы программы приведён на рисунке 4.



```

kate@MacBook-Pro-Ekaterina ~/D/o/l/s/lab_04_04 (master)> ./app
PARENT: pid: 18797 grp: 18797, child's pids: 18798, 18799
CHILD №1 (pid: 18798, ppid: 18797, grp: 18797) sent message to parent
CHILD №2 (pid: 18799, ppid: 18797, grp: 18797) sent message to parent
PARENT: child №1 (PID = 18798) has finished with code: 0
PARENT: child №2 (PID = 18799) has finished with code: 0
PARENT: received messages:
msg
msg msg

```

Рис. 4: Демонстрация работы программы (задание №4).

Задание №5

Предок и потомки обмениваются сообщениями через неименованный программный канал. В программу включается собственный обработчик сигнала. 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. Вывод соответствующих сообщений на экран. Вывод соответствующих сообщений на экран. Текст программы приведён на листинге 5.

Листинг 5: Использование сигналов

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/types.h>
4 #include <sys/wait.h>
5 #include "string"
6 #define ERROR_FORK 1
7 #define ERROR_EXEC 2
8 #define ERROR_PIPE 3
9 #define OK 0
10 bool sendSig = 0;
11 void empty(int sig){ }
12 void sendSigSwitch(int sig) {
13     sendSig = 1;
14 }
15 int main() {
16     signal(SIGINT, empty);
17     int fd[2];
18     const char *messages[2] = { "msg\n", "msg msg\n" };
19     int len[2] = { strlen(messages[0]), strlen(messages[1]) };
20     if (pipe(fd) == -1) {
21         return ERROR_PIPE;
22     }
23     int childpids[2];
24     for (int i = 0; i < 2; i++) {
25         int pid = fork();
26         if (pid == -1) {
27             return ERROR_FORK;
28         }
29         if (pid == 0) {
30             signal(SIGINT, sendSigSwitch);
31             sleep(4);
32             if (sendSig) {
33                 close(fd[0]);
34                 write(fd[1], messages[i], len[i]);
35                 printf("CHILD    %d: (pid: %d, ppid: %d, grp: %d) sent\n", i + 1, getpid(), getppid(), getpgrp());
36             }
37             else {
```

```

38         printf("CHILD    %d: didn't send a message\n", i + 1);
39     }
40     return OK;
41 }
42     childpids[i] = pid;
43 }
44     printf("PARENT: pid: %d grp: %d, child's pids: %d, %d\n", getpid(),
45         getpgrp(), childpids[0], childpids[1]);
46     for (int i = 0; i < 2; i++) {
47         int status;
48         pid_t childpid = wait(&status);
49         if (WIFEXITED(status)) {
50             printf("PARENT: child    %d (PID = %d) has finished with code: %d\n", i + 1, childpid, WEXITSTATUS(status));
51         }
52         else if (WIFSIGNALED(status)) {
53             printf("PARENT: child    %d (PID = %d) has finished because of signal: %d\n", i + 1, childpid, WTERMSIG(status));
54         }
55         else if (WIFSTOPPED(status)) {
56             printf("PARENT: child    %d (PID = %d) has been stopped because of signal: %d\n", i + 1, childpid, WSTOPSIG(status));
57         }
58     }
59     char buf[len[0] + len[1]];
60     close(fd[1]);
61     read(fd[0], buf, len[0] + len[1]);
62     buf[len[0] + len[1]] = '\0';
63     printf("PARENT: received messages:\n%s", buf);
64     return OK;
65 }

```

Результат работы программы приведён на рисунке 7. При первом запуске был испущен сигнал SIGINT, при втором не был.

```
kate@MacBook-Pro-Ekaterina ~/D/o/l/s/lab_04_05 (master)> ./app
PARENT: pid: 18944 grp: 18944, child's pids: 18945, 18946
^CCHILD №2: (pid: 18946, ppid: 18944, grp: 18944) sent message to parent
CHILD №1: (pid: 18945, ppid: 18944, grp: 18944) sent message to parent
PARENT: child №1 (PID = 18946) has finished with code: 0
PARENT: child №2 (PID = 18945) has finished with code: 0
PARENT: received messages:
msg msg
msg
kate@MacBook-Pro-Ekaterina ~/D/o/l/s/lab_04_05 (master)> ./app
PARENT: pid: 18958 grp: 18958, child's pids: 18959, 18960
CHILD №2: didn't send a message
CHILD №1: didn't send a message
PARENT: child №1 (PID = 18960) has finished with code: 0
PARENT: child №2 (PID = 18959) has finished with code: 0
PARENT: received messages:
```

Рис. 5: Демонстрация работы программы (задание №5).

Дополнительное задание

При выполнении вызова fork():

1. Создается пространство выполнения для дочерних и стека процесса-потомка;
2. Назначается идентификатор процесса PID и структура proc потомка;
3. Инициализируется структура proc потомка. Некоторое поле этой структуры копируется от процесса-родителя; идентификатор 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. Создается карта трансляции адресов для процесса-потомка;
5. Выделяется область u потомка и в нее копируется область u процесса-родителя;
6. Изменяются ссылки области u на новые карты адресации и пространство символа;
7. Потомок добавляется в набор процессов, которые разделяют область кода программы, выполняемой процессом-родителем;
8. Постранично дублируются области данных и строка родителя и модифицируется карта 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. Процесс-потомок помещается в очередь готовых процессов;
12. Возвращается PID в потоке возврата из системного вызова в родительском процессе и 0 - в дочерском-потомке.

Рис. 6: Последовательность действий fork

Системой вызов ехес выполняет след. 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 две райлы установлены бито $SUID$ или $SBID$ то эррективные идентификакторы UID и GID вызывающего процесса изменяет на UID и GID , соответствующие владельцу файла.
5. Копирует аргументы, передаваемые в ехес, а также переменные среды в пространство ядра, после чего текущее пользовательское пространство готово к уничтожению.
6. Возвращает пространство свопинга для областей данных и стека.
7. Возвращает старое адресное пространство и связанное с ним пространство свопинга. Если же процесс был создан при помощи $vfork()$, то производится возврат старого адресного пространства родительскому процессу.
8. Возвращает карту размещения 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 индивидуализоваться при выполнении файла. Процесс в системе UNIX обязан разбить на страницы, что означает, что каждая страница считывается в память только по мере необходимости.
10. Копирует аргументы и переменные среды обратно в новый стек программы.
11. Обрабатывает все обработчики сигналов в действие, определяемое по умолчанию, так как текущие обработчики сигналов не существуют в новой программе. Сигналы, которые были проигнорированы или зафиксированы перед вызовом ехес, остаются в тех же состояниях.
12. Индивидуализирует аппаратный контекст. При этом большинство регистров сбрасывается в 0, а указатель команды получает значение точки входа программы.

Рис. 7: Последовательность действий ехес

Приложения

Листинг 6: Сортировка

```
1 #include <iostream>
2 #define ERROR 1
3 void sort(int *data, int length, int (*cmp)(int, int))
4 {
5     bool fl;
6     for (size_t j = 1; j < length; j++) {
7         fl = false;
8         for (size_t i = 0; i < length - j; i++) {
9             if (cmp(data[i], data[i + 1]) > 0) {
10                 std::swap(data[i], data[i + 1]);
11                 fl = true;
12             }
13         }
14         if (!fl)
15             break;
16     }
17     return;
18 }
19 int compare_int(int f, int s)
20 {
21     return f - s;
22 }
23 int main(int argc, const char *argv[])
24 {
25     if (argc - 1 <= 0)
26         return ERROR;
27     int *a = (int *)malloc((argc - 1) * sizeof(int));
28     if (a == NULL)
29         return ERROR;
30     for (int i = 1; i < argc; i++)
31         a[i - 1] = atoi(argv[i]);
32
33     sort(a, argc - 1, compare_int);
34
35     printf("sorted array: ");
36     for (int i = 0; i < argc - 1; i++)
37         printf("%d ", a[i]);
38     printf("\n");
39
40     free(a);
41
42     return 0;
43 }
```

Листинг 7: Поиск максимума

```
1 #include <stdio.h>
2 #include <cstdlib>
3 #define ERROR 1
4 int main(int argc, const char *argv[])
5 {
6     if (argc < 2)
7         return ERROR;
8     int max = atoi(argv[1]), tmp;
9     for (int i = 2; i < argc; i++)
10         if ((tmp = atoi(argv[i])) > max)
11             max = tmp;
12     printf("max in array: %d\n", max);
13     return 0;
14 }
```