Санкт-Петербургский Политехнический Университет Петра Великого

Кафедра компьютерных систем и программных технологий

Отчёт по лабораторной работе №3

Курс: «Операционные системы»

Тема: «Процессы UNIX»

Выполнил студент:

Волкова М.Д.
Группа: 43501/3

Проверил:

Малышев И.А.

Санкт-Петербург
2017 г.

Лабораторная работа №3

1.1 Цель работы

Изучить принципы управления и порождения процессов, создания потоков в ОС Linux.

1.2 Программа работы

Глава 1. Порождение и запуск процессов

1. Создание программы на основе исходного файла с псевдораспараллеливанием вычислений, посредством порождения процесса-потомка.
2. Выполнить сначала однократные вычисления в каждом процессе, обратить внимание, какой процесс на каком этапе владеет процессорным ресурсом.
3. Однократные вычисления заменить на циклы, длительность исполнения которых достаточна для наблюдения конкуренции процессов за процессорный ресурс.
4. Изменить процедуру планирования и повторить эксперимент.
5. Разработать программы родителя и потомка parent.c и child.c, с фиксацией состояния таблицы процессов.
6. Запустить на выполнение программу parent. Узнать о всех процессах, запущенных с этого терминала.
7. Запустить на выполнение программу parent в фоновом режиме. Получение таблицы процессов, запущенных с терминала.
8. Выполнить создание процессов с использованием различных функций семейства exec(), привести 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, порожденных вложенными вызовами fork().
    (a) Проанализируйте очередность исполнения процессов, порожденных вложенными вызовами fork().
    (b) Измените процедуру планирования с помощью функции с шаблоном scheduler в ее названии и повторите эксперимент.
    (c) Поменяйте порядок очереди в RR-процедуре.
    (d) 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.

Глава 2. Взаимодействие родственных процессов

1. Изменяя длительности выполнения процессов и параметры системных вызовов, рассмотреть 3 ситуации и вывести соответствующие таблицы процессов.
   (a) Процесс-предок запускает процесс-потомок и ожидает его завершения.
   (b) Процесс-предок запускает процесс-потомок и, не ожидая его завершения, завершает свое выполнение. Зафиксируйте изменение родительского идентификатора процесса-потомка.
   (c) Процесс-предок запускает процесс-сын и не ожидает его завершения. Процесс-предок завершает свое выполнение. Зафиксируйте появление процесса-зомби, для этого включите команду ps в программу parent.c
2. Перенаправьте вывод не только на терминал, но и в файл. Организуйте программу многопроцессного функционирования так, чтобы результатом ее работы была демонстрация всех трех ситуаций с отображением в итоговом файле.

Глава 3. Управление процессами посредством сигналов

1. С помощью команды kill -l ознакомьтесь с перечнем сигналов, поддерживаемых процессами.
   (a) Процесс parent порождает процессы child1, child2, child3 и запускает на исполнение программные коды из соответствующих исполнительных файлов.
   (b) Далее родительский процесс осуществляет управление потомками, для этого он генерирует сигнал каждому пользовательскому процессу.
   (c) В пользовательских процессах-потомках необходимо обеспечить: для child1 - реакцию на сигнал по умолчанию; для child2 - реакцию игнорирования; для child3 - перехватывание и обработку сигнала. Сформируйте файл-проект из четырех файлов, откомпилируйте, запустите программу.
2. Организуйте посылку сигналов любым двум процессам, находящимся в разных состояниях: активном и пассивном, фиксируя моменты посылки и приема каждого сигнала с точностью до секунды. Приведите результаты в файле результатов.
3. Запустите в фоновом режиме несколько утилит, например: cat *.c > myprog & lpr myprog & lpr intro&. Воспользуйтесь командой jobs для анализа списка заданий и очередности их выполнения.
4. Ознакомьтесь с выполнением команды и системного вызова nice(1) и getpriority(2). Приведите примеры их использования в приложении. Определите границы приоритетов (создайте для этого программу).
5. Ознакомьтесь с командой nohup(1). Запустите длительный процесс по nohup(1). Завершите сеанс работы. Снова войдите в систему и проверьте таблицу процессов. Поясните результат.
6. Определите uid процесса, каково минимальное значение и кому оно принадлежит. Каково минимальное и максимальное значение pid, каким процессам принадлежат. Проанализируйте множество системных процессов, как их отличить от прочих, перечислите назначение самых важных из них.

Глава 4. Многопоточное функционирование

1. Подготовьте программу, формирующую несколько потоков. Каждый поток выводит сообщение с разными интервалами.
2. После запуска программы проанализируйте выполнение нитей, распределение во времени. Используйте для этого вывод таблицы процессов командой ps -axhf. Попробуйте удалить нить, зная ее идентификатор, командой kill.
3. Модифицируйте программу так, чтобы управление второй нитью осуществлялось посредством сигнала SIGUSR1 из первой нити. На пятой секунде работы приложения удалите вторую нить. Для этого воспользуйтесь функцией pthread_kill(t2, SIGUSR1); (t2 - дескриптор второй нити). В остальном программу можно не изменять.
4. Последняя модификация предполагает создание собственного обработчика сигнала, содержащего уведомление о начале его работы и возврат посредством функции pthread_exit(NULL). Сравните результаты, полученные после запуска этой модификации программы с результатами предыдущей.
5. Перехватите сигнал «CTRL+C» для процесса и потока однократно и многократно с восстановлением исходного обработчика после нескольких раз срабатывания. Проделайте аналогичную работу для переназначения другой комбинации клавиш.

6. С помощью утилиты kill выведите список всех сигналов и дайте их краткую характеристику на основе документации ОС. Для чего предназначены сигналы с 32 по 64-й.

7. Проанализируйте процедуру планирования для процессов и потоков одного процесса. Обоснуйте результат экспериментально. Попробуйте процедуру планирования изменить. Подтвердите экспериментально, если изменение возможно. Задайте нитям разные приоритеты программно и извне (объясните результат).

8. Создайте скрипт, выполняющий вашу лабораторную работу автоматически при наличии необходимых исходных файлов.

1.3 Ход работы

1.3.1 Глава 1. Порождение и запуск процессов

Создание функции, перераспределяющей ресурсы процессора на одно ядро

Для отображения корректного результата в условиях многопоточных возможностей процессора, разработаем функцию, перераспределяющую ресурсы процессора на одно ядро:

```
#include <sched.h>
#include "singlecore.h"

/*
  Функция, перераспределяющая ресурсы процессора на одно ядро
  Если не получилось, то возвращает 0
*/

int disableMultithreading(int pid) {
  // Структура, сохраняющая набор процессоров
  cpu_set_t cpuSet;

  // Обнуление структуры cpu_set_t
  CPU_ZERO(&cpuSet);
  // Добавляем в структуру новый процессор для последующего маскирования
  CPU_SET(0, &cpuSet);

  // Устанавливаем маску для процесса и выводим результат операции
  return sched_setaffinity(pid, sizeof(cpu_set_t), &cpuSet);
}
```

Заголовочный файл для функции:

```
#ifndef SINGLECORE
#define SINGLECORE

int disableMultithreading(int);

#endif // SINGLECORE
```

1. Создание программы на основе исходного файла с псевдораспараллеливанием вычислений посредством порождения процесса-потомка

Системный вызов fork служит для создания нового процесса в операционной системе UNIX. Процесс, который инициировал системный вызов fork , принято называть родительским процессом (*parent process*). Вновь порожденный процесс принято называть процессом-потомком (*child process*). Процесс-потомок является почти полной копией родительского процесса. У порожденного процесса по сравнению с родительским изменяются значения следующих параметров:

- Идентификатор процесса (PID).
- Идентификатор родительского процесса (PPID).
- Время, оставшееся до получения сигнала SIGALRM.
- Сигналы, ожидавшие доставки родительскому процессу, не будут доставляться порожденному процессу.

Была написана простейшая программа, иллюстрирующая действие функции *fork()*:

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

int main() {
  printf("Process has been started, pid %d, ppid %d.\n", getpid(), getppid());

  // Порождаем процесспотомок–
  if(fork())
    printf("Parent process, pid %d, ppid %d.\n", getpid(), getppid());
  else
    printf("Child process, pid %d, ppid %d.\n", getpid(), getppid());


  printf("Process has been finished, pid %d, ppid %d.\n", getpid(), getppid());
  return 0x0;
}
```

Результат работы программы:

```
stakenschneider@stakenschneider:~/temp$ gcc p1.1.c -o p1.1
stakenschneider@stakenschneider:~/temp$ ./p1.1
Process has been started, pid 2384, ppid 2383.
Parent process, pid 2384, ppid 2383.
Process has been finished, pid 2384, ppid 2383.
Child process, pid 2385, ppid 2384.
Process has been finished, pid 2385, ppid 2384.
```

Можно наблюдать, что *PPID* процесса-потомка совпадает с *PID* родительского процесса. В зависимости от возвращаемого значения функции *fork* можно контролировать, какой код выполнится родителем и потомком.

2. Выполнение однократных вычислений в каждом процессе

Проведем эксперимент: процесс-родитель инкрементирует переменную, процесс-потомок декрементирует переменную. Если оба результата не изменятся (останутся равными нулю), то это значит, что родитель и потомок имеют общую память данных. Если одна переменная будет равна единице, а вторая минус единице, то процессы используют различную память данных.

Была написана простейшая программа, выполняющая вычисления в процессе-родителе и процессе-потомке:

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

int main() {
  // Обнуляем тестовую переменную
  int testValue = 0;
  printf("Process has been started #%d.\n", getpid());

  // Порождаем процесспотомок–
  if(fork()) {
    // Если родитель, то инкрементируем переменную
    ++testValue;
    printf("Parent process #%d.\n", getpid());
    printf("Test value: %d \n", testValue);
  }
  else {
    // Если потомок, то декрементируем переменную
    --testValue;
    printf("Child process #%d.\n", getpid());
    printf("Test value: %d \n", testValue);
  }

  printf("Process has been finished #%d.\n", getpid());
  return 0x0;
}
```

Результат работы программы:

```
stakenschneider@stakenschneider:~/temp$ gcc p1.2.c -o p1.2
stakenschneider@stakenschneider:~/temp$ ./p1.2
Process has been started #4169.
Parent process #4169.
Test value: 1
Process has been finished #4169.
Child process #4170.
Test value: -1
Process has been finished #4170.
```

Результат подтвердил, что процесс-родитель и процесс-потомок используют различную память данных.

3. Выполнение множественных вычислений, для наблюдения конкуренции родственных процессов

Разработаем программу, которая выводит несколько сообщений от родителя и потомка с заданной периодичностью:

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

#define ITERATIONS_COUNT ((int)1e6)
#define ITERATIONS_PERIOD ((int)1e5)

int main() {
  // Создаем временную переменную для идентификаторра процесса
  printf("Process has been starte.d #%d.\n", getpid());

  // Порождаем процесспотомок–
  if(fork()) {
    int pid = getpid();
    printf("Parent process #%d.\n", pid);

    // Если не получилось перераспределить ресурсы процессорра на одно ядро, то выходим с ошибкой
    if(disableMultithreading(pid) < 0)
      return 0x1;

    // Запускаем цикл с большим количеством итераций, и с каким то периодом выводим сообщение
    int iteration = ITERATIONS_COUNT;
    while(iteration != 0)
      if(--iteration % ITERATIONS_PERIOD == 0)
        printf("Parent win.\n");
  }
  else {
    int pid = getpid();
    printf("Child process #%d.\n", pid);

    // Если не получилось перераспределить ресурсы процессорра на одно ядро, то выходим с ошибкой
    if(disableMultithreading(pid) < 0)
      return 0x2;

    // Запускаем цикл с большим количеством итераций, и с каким то периодом выводим сообщение
    int iteration = ITERATIONS_COUNT;
    while(iteration != 0)
      if(--iteration % ITERATIONS_PERIOD == 0)
        printf("Child win.\n");
  }

  printf("Process has been finished #%d.\n", getpid());
  return 0x0;
}
```

Результат выполнения программы:

```
stakenschneider@stakenschneider:~/temp$ g++ p1.3.c singlecore.c -o p1.3
stakenschneider@stakenschneider:~/temp$ ./p1.3
Process has been started #4408.
Parent process #4408.
Child process #4409.
Parent win.
Parent win.
Parent win.
Parent win.
Parent win.
Child win.
Child win.
Child win.
Child win.
Child win.
Child win.
Parent win.
Parent win.
Parent win.
Parent win.
Child win.
Child win.
Child win.
Child win.
Process has been finished #4409.
Parent win.
Process has been finished #4408.
```

В результате выполнения программы видна конкуренция процессов за процессорное время.

4. Наблюдение конкуренции родственных процессов, в зависимости от алгоритма планирования

Для изменения приоритета планирования используется функция *sched_setscheduler*, принимающая следующие аргументы:

- Идентификатор процесса (PID).
- Политику планирования. Существует три политики планирования: *SCHED_OTHER*, *SCHED_FIFO* и *SCHED_RR*. *SCHED_OTHER* - используемый по умолчанию алгоритм со стандартным разделением времени, с которым работает большинство процессов.*SCHED_FIFO* и *SCHED_RR* предназначены для процессов, зависящих от возникновения задержек, которым необходим более четкий контроль над порядком исполнения процессов.
- Структуру с численным значением приоритета. Для *SCHED_OTHER* численное значение всегда нулевое. Для *SCHED_FIFO* и *SCHED_RR* численные значения варьируются от 0 до 99.

Разработаем программу, принимающую в качестве аргументов командной строки алгоритм планирования и численные значение приоритетов процесса для родителя и каждого из трех потомков:

```
#include <iostream>
#include <string>
#include <unistd.h>
#include <sched.h>
#include <stdlib.h>
#include "singlecore.h"

int main(int argc , char** argv) {
  // Проверка на количество аргументов, если неправильное, то выходим с ошибкой
  if(argc != 6) {
    std::cerr << "Invalid count of arguments." << std::endl;
    return 0x1;
  }

  int policy;
  int coeffParent, coeffChildFirst, coeffChildSecond, coeffChildThird;

```

```
18  try {
19    // Первый аргумент – алгоритм планирования
20    switch(std::stoi(argv[1])) {
21      case 0: policy = SCHED_FIFO; break;
22      case 1: policy = SCHED_RR; break;
23      case 2: policy = SCHED_OTHER; break;
24      default: throw std::invalid_argument(argv[2]);
25    }
26
27    // Следующие четыре аргумента – приоритеты
28    coeffParent = std::stoi(argv[2]);
29    coeffChildFirst = std::stoi(argv[3]);
30    coeffChildSecond = std::stoi(argv[4]);
31    coeffChildThird = std::stoi(argv[5]);
32  }
33  catch(const std::exception& exception) {
34    // Обработка ситуации неверного аргумента
35    std::cerr << "Invalid argument." << std::endl;
36    return 0x2;
37  }
38
39  struct sched_param schedParam;
40  schedParam.sched_priority = 0;
41
42  // Parent
43
44  int pid = getpid();
45  int ppid = getppid();
46
47  // Если не получилось перераспределить ресурсы процессорра на одно ядро, то выходим с ошибкой
48  if(disableMultithreading(pid) < 0)
49    return 0x3;
50
51  // Записываем алгоритм планирования в поле структуры
52  if(policy != SCHED_OTHER)
53    schedParam.sched_priority = coeffParent;
54  // Вызов планировщика
55  if(sched_setscheduler(0, policy, &schedParam) == -1) {
56    std::cerr << "It's impossible to set sheduler. Parent pid #" << pid << ", ppid #" <<
      ppid << "." << std::endl;
57    return 0x4;
58  }
59
60  std::cout << "Parent pid #" << pid << ", ppid #" << ppid << "." << std::endl;
61
62  if(!fork()) {
63    // First child
64
65    pid = getpid();
66    ppid = getppid();
67
68    // Если не получилось перераспределить ресурсы процессорра на одно ядро, то выходим с ошибкой
69    if(disableMultithreading(pid) < 0)
70      return 0x5;
71
72    // Записываем алгоритм планирования в поле структуры
73    if(policy != SCHED_OTHER)
74      schedParam.sched_priority = coeffChildFirst;
75    // Вызов планировщика
76    if(sched_setscheduler(pid, policy, &schedParam) == -1) {
77        std::cerr << "It's impossible to set sheduler. First child pid #" << pid << ",
      ppid #" << ppid << "." << std::endl;
78        return 0x6;
79    }
80
81    std::cout << "First child pid #" << pid << ", ppid #" << ppid << "." << std::endl;
```

```
82    }
83
84    if(!fork()) {
85      // Second child
86
87      pid = getpid();
88      ppid = getppid();
89
90      // Если не получилось перераспределить ресурсы процессорра на одно ядро, то выходим с ошибкой
91      if(disableMultithreading(pid) < 0)
92        return 0x7;
93
94      // Записываем алгоритм планирования в поле структуры
95      if(policy != SCHED_OTHER)
96        schedParam.sched_priority = coeffChildSecond;
97      // Вызов планировщика
98      if(sched_setscheduler(pid, policy, &schedParam) == -1) {
99          std::cerr << "It's impossible to set sheduler. Second child pid #" << pid << ",
      ppid #" << ppid << "." << std::endl;
100         return 0x8;
101     }
102
103     std::cout << "Second child pid #" << pid << ", ppid #" << ppid << "." << std::endl;
104   }
105
106   if(!fork()) {
107     // Third child
108
109     pid = getpid();
110     ppid = getppid();
111
112     // Если не получилось перераспределить ресурсы процессорра на одно ядро, то выходим с ошибкой
113     if(disableMultithreading(pid) < 0)
114       return 0x9;
115
116     // Записываем алгоритм планирования в поле структуры
117     if(policy != SCHED_OTHER)
118       schedParam.sched_priority = coeffChildThird;
119     // Вызов планировщика
120     if(sched_setscheduler(pid, policy, &schedParam) == -1) {
121         std::cerr << "It's impossible to set sheduler. Third child pid #" << pid << ",
      ppid #" << ppid << "." << std::endl;
122         return 0xA;
123     }
124
125     std::cout << "Third child pid #" << pid << ", ppid #" << ppid << "." << std::endl;
126   }
127
128   std::cin.get();
129   return 0x0;
130 }
```

Пример работы программы с алгоритмом планирования *SCHED_FIFO* (первый аргумент в командной строке 0):

```
1  stakenschneider@stakenschneider:~/temp$ g++ -std=c++11 p1.4.cpp singlecore.c -o p1.4
2
3  stakenschneider@stakenschneider:~/temp$ sudo ./p1.4 0 64 40 30 20
4  Parent pid #5241, ppid #5240.
5  First child pid #5242, ppid #2274.
6  Second child pid #5245, ppid #2274.
7  Second child pid #5243, ppid #2274.
8  Third child pid #5248, ppid #2274.
9  Third child pid #5247, ppid #2274.
10 Third child pid #5246, ppid #2274.
11 Third child pid #5244, ppid #2274.
12
```

```
stakenschneider@stakenschneider:~/temp$ sudo ./p1.4 0 64 31 40 64
Parent pid #5321, ppid #5320.
Third child pid #5324, ppid #5321.
Second child pid #5323, ppid #5321.
Third child pid #5325, ppid #5323.
First child pid #5322, ppid #5321.
Second child pid #5326, ppid #5322.
Third child pid #5328, ppid #5326.
Third child pid #5327, ppid #5322.

stakenschneider@stakenschneider:~/temp$ sudo ./p1.4 0 74 20 20 20
Parent pid #5349, ppid #5348.
Third child pid #5352, ppid #5349.
Second child pid #5351, ppid #5349.
First child pid #5350, ppid #5349.
Third child pid #5353, ppid #5351.
Second child pid #5354, ppid #5350.
Third child pid #5355, ppid #5350.
Third child pid #5356, ppid #5354.
```

Как только появляется более приоритетный процесс он выполняется, а все остальные ждут его завершения. В первом примере первым выполнился процесс-родитель с приоритетом 74, после этого выполнился первый потомок с приоритетом 40, потом второй потомок с приоритетом 30, потом третий потомок с приоритетом 20.

Во втором эксперименте родительский-процесс и третий потомок имеют наивысшие приоритеты, вследствие чего выполняются раньше других.

Пример работы программы с алгоритмом планирования *SCHED_RR* (первый аргумент в командной строке 1):

```
stakenschneider@stakenschneider:~/temp$ sudo ./p1.4 1 74 60 30 20
Parent pid #5415, ppid #5414.
First child pid #5416, ppid #5415.
Second child pid #5419, ppid #5416.
Second child pid #5417, ppid #5415.
Third child pid #5422, ppid #5417.
Third child pid #5421, ppid #5419.
Third child pid #5420, ppid #5416.
Third child pid #5418, ppid #5415.

stakenschneider@stakenschneider:~/temp$ sudo ./p1.4 1 74 50 60 74
Parent pid #5441, ppid #5440.
Third child pid #5444, ppid #5441.
Second child pid #5443, ppid #5441.
Third child pid #5445, ppid #5443.
First child pid #5442, ppid #5441.
Second child pid #5446, ppid #5442.
Third child pid #5448, ppid #5446.
Third child pid #5447, ppid #5442.

stakenschneider@stakenschneider:~/temp$ sudo ./p1.4 1 74 30 30 30
Parent pid #5488, ppid #5487.
Third child pid #5491, ppid #5488.
Second child pid #5490, ppid #5488.
First child pid #5489, ppid #5488.
Third child pid #5492, ppid #5490.
Second child pid #5493, ppid #5489.
Third child pid #5494, ppid #5489.
Third child pid #5495, ppid #5493.
```

Результаты *SCHED_RR* аналогичны *SCHED_FIFO*.

Пример работы программы с алгоритмом планирования *SCHED_OTHER* (первый аргумент в командной строке 2):

```
stakenschneider@stakenschneider:~/temp$ sudo ./p1.4 2 0 0 0 0
Parent pid #5956, ppid #5955.
Third child pid #5959, ppid #5956.
```

```
Second child pid #5958, ppid #5956.
First child pid #5957, ppid #5956.
Third child pid #5960, ppid #5958.
Third child pid #5962, ppid #5957.
Second child pid #5961, ppid #5957.
Third child pid #5963, ppid #5961.
```

При *SCHED_OTHER* все процессы выполняются в стандартной последовательности.

5-7. Разработка программ родителя и потомка с фиксацией состояния таблицы процессов

Разработаем программу родителя, которая вызывает другую программу в процессе-потомке. Для запуска программы используется функция *execl*, для системного вызова - функция *system*, для ожидания завершения процесса-потомка используется функция *wait*:

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

int main() {
  // Переменная для последующего использования wait
  int temp;

  printf("Parent process pid #%d.\n", getpid());

  // Выполняем исполняемый файл, в процессепотомке–
  if(!fork())
    execl("child", "child", NULL);

  // Ждем завершение процессапотомка–
  printf("Waiting child process pid #%d.\n", wait(&temp));
  return 0x0;
}
```

Программа потомок:

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

int main() {
  printf("Child process pid #%d.\n", getpid());

  // Вызываем список текущих процессов в виде дерева
  system("ps -xf");
  return 0x0;
}
```

Вывод программы до добавления системного вызова отображения процессов:

```
stakenschneider@stakenschneider:~/temp$ gcc child.c -o child
stakenschneider@stakenschneider:~/temp$ gcc parent.c -o parent
stakenschneider@stakenschneider:~/temp$ ./parent
Parent process pid #6214.
Child process pid #6215.
Waiting child process pid #6215.
```

Вывод программы после добавления системного вызова отображения процессов:

```
stakenschneider@stakenschneider:~/temp$ ./parent
Parent process pid #6276.
Child process pid #6277.
  PID TTY      STAT   TIME COMMAND
( ... )
 5390 pts/1    Ss     0:00  |   \_ bash
 6276 pts/1    S+     0:00  |      \_ ./parent
 6277 pts/1    S+     0:00  |         \_ child
 6278 pts/1    S+     0:00  |            \_ sh -c ps -xf
```

```
 6279 pts/1    R+     0:00   |                    \_ ps -xf
( ... )
Waiting child process pid #6277.
```

Вывод программы в фоновом режиме после добавления системного вызова отображения процессов:

```
stakenschneider@stakenschneider:~/temp$ ./parent &
[1] 6395
Parent process pid #6395.
Child process pid #6396.
  PID TTY      STAT   TIME COMMAND
( ... )
 5390 pts/1    Ss+    0:00   |   \_ bash
 6395 pts/1    S      0:00   |      \_ ./parent
 6396 pts/1    S      0:00   |         \_ child
 6397 pts/1    S      0:00   |            \_ sh -c ps -xf
 6398 pts/1    R      0:00   |               \_ ps -xf
( ... )
Waiting child process pid #6396.

[1]+  Done                    ./parent
```

Здесь можно наблюдать передачу управления командной оболочке до завершения процесса.

8. Создание процессов с использованием различных функций семейства exec()

Семейство функций *exec* содержит набор функций с различными сигнатурами для запуска дочерних процессов.

Семейство *exec* содержит следующие прототипы:

```
int execl(char * pathname, char * arg0, arg1, ..., argn, NULL);
int execle(char * pathname, char * arg0, arg1, ..., argn, NULL, char ** envp);
int execlp(char * pathname, char * arg0, arg1, ..., argn, NULL);
int execlpe(char *  pathname, char * arg0, arg1, ..., argn, NULL, char ** envp);
int execv(char * pathname, char * argv[]);
int execve(char * pathname, char * argv[], char ** envp);
int execvp(char * pathname, char * argv[]);
int execvpe(char * pathname, char * argv[], char ** envp);
```

Суффиксы *l*, *v*, *p* и *e*, добавляемые к имени семейства *exec* обозначают, что данная функция будет работать с некоторыми особенностями:

- Суффикс *p* - определяет, что функция будет искать дочернюю программу в директориях, определяемых переменной среды DOS PATH. Без суффикса *p* поиск будет производиться только в рабочем каталоге.
- Суффикс *l* - показывает, что адресные указатели (arg0, arg1, ..., argn) передаются, как отдельные аргументы. Обычно суффикс *l* употребляется, когда число передаваемых аргументов заранее вам известно.
- Суффикс *v* - показывает, что адресные указатели (arg[0], arg[1],...arg[n]) передаются, как массив указателей. Обычно, суффикс *v* используется, когда передается неизвестно число аргументов.
- Суффикс *e* - показывает, что дочернему процессу может быть передан аргумент *envp*, который позволяет выбирать среду дочернего процесса. Без суффикса *e* дочерний процесс унаследует среду родительского процесса.

Разработаем программу, которая запускает программу */bin/df* с ключем *-h* шестью различными функциями семейства *exec*. Функция выбирается аргументом командной строки:

```
#include <unistd.h>
#include <iostream>
#include <string>

// Константы для функции семейства exec()
const char* FULL_PATH = "/bin/df";
const char* SIMPLE_PATH = "df";
const char* FIRST_ARG = "df";
```

```
const char* SECOND_ARG = "-h";
const char* LAST_ARG = (char*) NULL;

int main(int argc, char** argv) {
  // Проверка на количество аргументов, если неправильное, то выходим с ошибкой
  if(argc != 2) {
    std::cerr << "Invalid count of arguments." << std::endl;
    return 0x1;
  }

  int value;
  try {
    value = std::stoi(argv[1]);
    if(value < 0 || value > 5)
      throw std::invalid_argument(argv[1]);
  }
  catch(const std::exception& exception) {
    // Обработка ситуации неверного аргумента
    std::cerr << "Invalid argument." << std::endl;
    return 0x2;
  }

  // Массив аргументов
  char* args[] = {(char*) FIRST_ARG, (char*) SECOND_ARG, (char*) LAST_ARG};

  switch(value) {
    case 0:
      // Полный путь, аргументы программы в аргументах функции
      execl(FULL_PATH, FIRST_ARG, SECOND_ARG, LAST_ARG);
      break;

    case 1:
      // Неполный путь, аргументы программы в аргументах функции
      execlp(SIMPLE_PATH, FIRST_ARG, SECOND_ARG, LAST_ARG);
      break;

    case 2:
      // Полный путь, аргументы программы в аргументах функции, добавлен массив переменных окружения
      execle(FULL_PATH, FIRST_ARG, SECOND_ARG, LAST_ARG, __environ);
      break;
    case 3:
      // Полный путь, аргументы программы указываются в массиве
      execv(FULL_PATH, args);
      break;
    case 4:
      // Неполный путь, аргументы программы указываются в массиве
      execvp(SIMPLE_PATH, args);
      break;
    case 5:
      // Неполный путь, аргументы программы указываются в массиве, добавлен массив переменных
    окружения
      execvpe(SIMPLE_PATH, args, __environ);
      break;
    default:
      // Обработка ситуации неверного аргумента
      std::cerr << "Impossible situation." << std::endl;
      return 0x3;
  }

  return 0x0;
}
```

Результаты выполнения программы ожидаемо идентичен, независимо от сигнатуры функции семейства *exec*:

```
stakenschneider@stakenschneider:~/temp$ g++ -std=c++11 p1.8.cpp -o p1.8

```

```
stakenschneider@stakenschneider:~/temp$ ./p1.8 0
Filesystem      Size  Used Avail Use% Mounted on
udev            976M     0  976M   0% /dev
tmpfs           199M  3,8M  196M   2% /run
/dev/sda1       292G  5,7G  271G   3% /
tmpfs           995M  220K  994M   1% /dev/shm
tmpfs           5,0M  4,0K  5,0M   1% /run/lock
tmpfs           995M     0  995M   0% /sys/fs/cgroup
tmpfs           199M   52K  199M   1% /run/user/1000

stakenschneider@stakenschneider:~/temp$ ./p1.8 1
( ... )

stakenschneider@stakenschneider:~/temp$ ./p1.8 5
Filesystem      Size  Used Avail Use% Mounted on
udev            976M     0  976M   0% /dev
tmpfs           199M  3,8M  196M   2% /run
/dev/sda1       292G  5,7G  271G   3% /
tmpfs           995M  220K  994M   1% /dev/shm
tmpfs           5,0M  4,0K  5,0M   1% /run/lock
tmpfs           995M     0  995M   0% /sys/fs/cgroup
tmpfs           199M   52K  199M   1% /run/user/1000
```

9.1. Анализ функции wait()

Функция *wait* приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс не завершится, или до появления сигнала, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик. Если дочерний процесс к моменту вызова функции уже завершился (так называемый "зомби"("zombie")), то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются.

Если аргумент *status* не равен *NULL*, то функции *wait* и *waitpid* сохраняют информацию о статусе в переменной, на которую указывает status. Этот статус можно проверить с помощью нижеследующих макросов:

- *WIFEXITED(status)* - не равно нулю, если дочерний процесс успешно завершился.
- *WEXITSTATUS(status)* - возвращает восемь младших битов значения, которое вернул завершившийся дочерний процесс. Эти биты могли быть установлены в аргументе функции *exit* или в аргументе оператора *return* функции *main*. Этот макрос можно использовать, только если *WIFEXITED* вернул ненулевое значение.
- *WIFSIGNALED(status)* - возвращает истинное значение, если дочерний процесс завершился из-за необработанного сигнала.
- *WTERMSIG(status)* - возвращает номер сигнала, который привел к завершению дочернего процесса. Этот макрос можно использовать, только если *WIFSIGNALED* вернул ненулевое значение.
- *WIFSTOPPED(status)* - возвращает истинное значение, если дочерний процесс, из-за которого функция вернула управление, в настоящий момент остановлен; это возможно, только если использовался флаг *WUNTRACED* или когда подпроцесс отслеживается.
- *WSTOPSIG(status)* - возвращает номер сигнала, из-за которого дочерний процесс был остановлен. Этот макрос можно использовать, только если *WIFSTOPPED* вернул ненулевое значение.

Разработаем программу, в которой процесс-родитель ожидает завершения процесса-потомка, а после анализирует статус, с помощью вышеописанных макросов:

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

int main() {
  // Порождаем процесспотомок–
  if(fork()) {
    int status = 0;
```

```
10
11      // Ждем завершение потомка, перезаписывая переменную статуса
12      const int result = wait(&status);
13
14      printf("Parent started, pid %d.\n", getpid());
15      printf("WAIT: %d\n", result);
16      printf("STATUS: %d\n", status);
17
18      // Используем различные макросы для получения большей информации о статусе
19      printf("WIFEXITED: %d\n", WIFEXITED(status));
20      printf("WEXITSTATUS: 0x%X\n", WEXITSTATUS(status));
21      printf("WIFSIGNALED: %d\n", WIFSIGNALED(status));
22      printf("WTERMSIG: %d\n", WTERMSIG(status));
23      printf("WIFSTOPPED: %d\n", WIFSTOPPED(status));
24      printf("WSTOPSIG: 0x%X\n", WSTOPSIG(status));
25    }
26    else {
27      printf("Child finished, pid %d, ppid %d.\n\n", getpid(), getppid());
28
29      // Завершаем процесспотомок– шестнадцатиричным кодом 0xA
30      return 0xA;
31    }
32
33    return 0x0;
34  }
```

Результат работы программы:

```
1  stakenschneider@stakenschneider:~/temp$ gcc p1.9.1.c -o p1.9.1
2  stakenschneider@stakenschneider:~/temp$ sudo ./p1.9.1
3  Child finished, pid 4835, ppid 4834.
4
5  Parent started, pid 4834.
6  WAIT: 4835
7  STATUS: 2560
8  WIFEXITED: 1
9  WEXITSTATUS: 0xA
10 WIFSIGNALED: 0
11 WTERMSIG: 0
12 WIFSTOPPED: 0
13 WSTOPSIG: 0xA
```

Процесс потомок завершился успешно с кодом выхода 0xA, номер сигнала из-за которого процесс был остановлен также 0xA.

9.2. Анализ функции waitpid()

Функция *waitpid* приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс, указанный в параметре *pid*, не завершит выполнение, или пока не появится сигнал, который либо завершает текущий процесс либо требует вызвать функцию-обработчик. Если указанный дочерний процесс к моменту вызова функции уже завершился, то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются.

Исследуем функцию *waitpid*. Она позволяет ожидать процесс по его *PID*, с учетом следующих опций:

- *WNOHANG* - означает немедленное возвращение управления, если ни один дочерний процесс не завершил выполнение.
- *WUNTRACED* - означает возврат управления и для остановленных (но не отслеживаемых) дочерних процессов, о статусе которых еще не было сообщено. Статус для отслеживаемых остановленных подпроцессов также обеспечивается без этой опции.

Разработаем программу, которая порождает три процесса-потомка и в зависимости от аргумента командной строки выполняет функцию *waitpid* с конкретным набором опций:

```
1  #include <iostream>
2  #include <string>
3  #include <unistd.h>
```

```cpp
#include <sys/wait.h>

const int CHILD_PROCESS_COUNT = 3;

int main(int argc, char** argv) {
  // Проверка на количество аргументов, если неправильное, то выходим с ошибкой
  if(argc != 2) {
    std::cerr << "Invalid count of arguments." << std::endl;
    return 0x1;
  }

  int value;
  try {
    value = std::stoi(argv[1]);
    if(value < 0 || value > 3)
      throw std::invalid_argument(argv[1]);
  }
  catch(const std::exception& exception) {
    // Обработка ситуации неверного аргумента
    std::cerr << "Invalid argument." << std::endl;
    return 0x2;
  }

  // Задаем флаги исполнения в зависимости от аргумента
  int* option;
  switch(value) {
    case 0:
      option = new int[CHILD_PROCESS_COUNT]{WNOHANG, WUNTRACED, WNOHANG};
      break;
    case 1:
      option = new int[CHILD_PROCESS_COUNT]{WUNTRACED, WUNTRACED, WUNTRACED};
      break;
    case 2:
      option = new int[CHILD_PROCESS_COUNT]{WNOHANG, WUNTRACED, WUNTRACED};
      break;
    default:
      // Обработка ситуации неверного аргумента
      std::cerr << "Impossible situation." << std::endl;
      return 0x3;
  }

  int childPid, childWait, childStatus;
  const char* childProgramName[] = {"p1.9.2.c1", "p1.9.2.c2", "p1.9.2.c3"};

  for(int index = 0; index < CHILD_PROCESS_COUNT; ++index) {
    // Порождаем процесспотомок–
    childPid = fork();
    if(!childPid) {
      // Запускаем программу
      execl(childProgramName[index], childProgramName[index], NULL);
      return 0x0;
    }

    // Выводим дерево процессов в файл
    system("ps xf > p1.9.2.p.ps.log");

    // Ждем выполнения процессапотомка–
    childWait = waitpid(childPid, &childStatus, option[index]);
    std::cout << "Child process #" << index + 1 << " has been finished ";

    // Проверка на успешность решения процесса
    if(WIFEXITED(childStatus) == 0)
      std::cout << " unsuccessfully." << std::endl;
    else
      std::cout << " successfully." << std::endl;
  }
```

```
70
71    return 0x0;
72 }
```

Создадим программы-потомки для запуска. Первый потомок:

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  #define SLEEP_DURATION 1
5
6  int main() {
7    printf("Child #1 started, pid %d, ppid %d.\n", getpid(), getppid());
8    sleep(SLEEP_DURATION);
9    return 0x0;
10 }
```

Второй потомок:

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  #define SLEEP_DURATION 2
5
6  int main() {
7    printf("Child #2 started, pid %d, ppid %d.\n", getpid(), getppid());
8    sleep(SLEEP_DURATION);
9    return 0x0;
10 }
```

Третий потомок:

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  #define SLEEP_DURATION 3
5
6  int main() {
7    printf("Child #3 started, pid %d, ppid %d.\n", getpid(), getppid());
8    sleep(SLEEP_DURATION);
9    return 0x0;
10 }
```

Проверим работоспособность программ-потомков:

```
1  stakenschneider@stakenschneider:~/temp$ gcc p1.9.2.c1.c -o p1.9.2.c1
2  stakenschneider@stakenschneider:~/temp$ gcc p1.9.2.c2.c -o p1.9.2.c2
3  stakenschneider@stakenschneider:~/temp$ gcc p1.9.2.c3.c -o p1.9.2.c3
4  stakenschneider@stakenschneider:~/temp$ ./p1.9.2.c1
5  Child #1 started, pid 5236, ppid 3623.
6  stakenschneider@stakenschneider:~/temp$ ./p1.9.2.c2
7  Child #2 started, pid 5239, ppid 3623.
8  stakenschneider@stakenschneider:~/temp$ ./p1.9.2.c3
9  Child #3 started, pid 5240, ppid 3623.
```

Выполним программу-родитель с разными наборами опций: аргумент 0 задает опции *WNOHANG*, *WUNTRACEI* *WNOHANG*, аргумент 1 задает опции *WUNTRACED*, *WUNTRACED*, *WUNTRACED*, аргумент 2 задает опции *WNOHANG*, *WUNTRACED*, *WUNTRACED*.

```
1  stakenschneider@stakenschneider:~/temp$ g++ -std=c++11 p1.9.2.p.cpp -o p1.9.2.p
2
3  stakenschneider@stakenschneider:~/temp$ ./p1.9.2.p 0
4  Child #1 started, pid 6057, ppid 6056.
5  Child process #1 has been finished  unsuccessfully.
6  Child #2 started, pid 6060, ppid 6056.
7  Child process #2 has been finished  successfully.
8  Child #3 started, pid 6063, ppid 6056.
9  Child process #3 has been finished  successfully.
```

```
stakenschneider@stakenschneider:~/temp$ ./p1.9.2.p 1
Child #1 started, pid 6067, ppid 6066.
Child process #1 has been finished  successfully.
Child #2 started, pid 6070, ppid 6066.
Child process #2 has been finished  successfully.
Child #3 started, pid 6074, ppid 6066.
Child process #3 has been finished  successfully.

stakenschneider@stakenschneider:~/temp$ ./p1.9.2.p 2
Child #1 started, pid 6078, ppid 6077.
Child process #1 has been finished  unsuccessfully.
Child #2 started, pid 6081, ppid 6077.
Child process #2 has been finished  successfully.
Child #3 started, pid 6084, ppid 6077.
Child process #3 has been finished  successfully.
```

Из опытов видно, чтобы процесс завершился корректно и не повис в ожидании дочернего процесса, необходимо использовать опцию *WUNTRACED*. Если дочерний процесс должен быть обязательно проконтроллирован по завершению, то необходимо использовать *WNOHANG*.

Рассмотрим дерево процессов перед ожиданием последнего дочернего процесса:

```
  PID TTY      STAT   TIME COMMAND
( ... )
 3616 ?        Sl     0:30  \_ /usr/lib/gnome-terminal/gnome-terminal-server
 5410 pts/18   Ss     0:00  |   \_ bash
 6077 pts/18   S+     0:00  |       \_ ./p1.9.2.p 2
 6078 pts/18   Z+     0:00  |           \_ [p1.9.2.c1] <defunct>
 6084 pts/18   S+     0:00  |           \_ p1.9.2.c3
 6085 pts/18   S+     0:00  |           \_ sh -c ps xf > p1.9.2.p.ps.log
 6086 pts/18   R+     0:00  |               \_ ps xf
( ... )
```

Приставка *<defunct>* у процесса *p1.9.2.c1* означает, что процесс уже выполнился, но все еще остался в списке процессов. Использование опции *WNOHANG* для данного процесса подтверждает корректное возвращение управления.

10.1. Анализ очередности процессов, порожденных вложенными вызовами fork()

Разработаем программу, принимающую в качестве аргументов командной строки алгоритм планирования и численные значение приоритетов процесса для родителя и каждого из трех потомков:

```
#include <iostream>
#include <unistd.h>
#include <sched.h>
#include "singlecore.h"

int main(int argc, char** argv) {
  // Проверка на количество аргументов, если неправильное, то выходим с ошибкой
  if(argc != 6) {
    std::cerr << "Invalid count of arguments." << std::endl;
    return 0x1;
  }

  int policy;
  int coeffParent, coeffChildFirst, coeffChildSecond, coeffChildThird;

  try {
    // Первый аргумент – алгоритм планирования
    switch(std::stoi(argv[1])) {
      case 0: policy = SCHED_FIFO; break;
      case 1: policy = SCHED_RR; break;
      case 2: policy = SCHED_OTHER; break;
      default: throw std::invalid_argument(argv[2]);
    }

    // Следующие четыре аргумента – приоритеты
```

```
26      coeffParent = std::stoi(argv[2]);
27      coeffChildFirst = std::stoi(argv[3]);
28      coeffChildSecond = std::stoi(argv[4]);
29      coeffChildThird = std::stoi(argv[5]);
30    }
31    catch(const std::exception& exception) {
32      // Обработка ситуации неверного аргумента
33      std::cerr << "Invalid argument." << std::endl;
34      return 0x2;
35    }
36
37    struct sched_param schedParam;
38    schedParam.sched_priority = 0;
39
40    // Parent
41
42    // Если не получилось перераспределить ресурсы процессорра на одно ядро, то выходим с ошибкой
43    if(disableMultithreading(getpid()) < 0)
44      return 0x3;
45
46    // Записываем алгоритм планирования в поле структуры
47    if(policy != SCHED_OTHER)
48      schedParam.sched_priority = coeffParent;
49
50    // Вызов планировщика
51    if(sched_setscheduler(0, policy, &schedParam) == -1) {
52      std::cerr << "It's impossible to set sheduler. Parent pid #" << getpid() << ", ppid #
        " << getppid() << "." << std::endl;
53      return 0x4;
54    }
55
56    std::cout << "Parent pid #" << getpid() << ", ppid #" << getppid() << "." << std::endl;
57
58    if(!fork()) {
59      // Child level one
60
61      // Если не получилось перераспределить ресурсы процессорра на одно ядро, то выходим с ошибкой
62      if(disableMultithreading(getpid()) < 0)
63        return 0x5;
64
65      // Записываем алгоритм планирования в поле структуры
66      if(policy != SCHED_OTHER)
67        schedParam.sched_priority = coeffChildFirst;
68
69      if(!fork()) {
70        // Child level two
71
72        // Если не получилось перераспределить ресурсы процессорра на одно ядро, то выходим с ошибкой
73        if(disableMultithreading(getpid()) < 0)
74          return 0x7;
75
76        // Записываем алгоритм планирования в поле структуры
77        if(policy != SCHED_OTHER)
78          schedParam.sched_priority = coeffChildSecond;
79
80        if(!fork()) {
81          // Child level three
82
83          // Если не получилось перераспределить ресурсы процессорра на одно ядро, то выходим с ошибкой
84          if(disableMultithreading(getpid()) < 0)
85            return 0x9;
86
87          // Записываем алгоритм планирования в поле структуры
88          if(policy != SCHED_OTHER)
89            schedParam.sched_priority = coeffChildThird;
90
```

```
91          // Вызов планировщика
92          if(sched_setscheduler(getpid(), policy, &schedParam) == -1) {
93            std::cerr << "It's impossible to set sheduler. Third child pid #" << getpid()
    << ", ppid #" << getppid() << "." << std::endl;
94            return 0xA;
95          }
96
97          std::cout << "Third child pid #" << getpid() << ", ppid #" << getppid() << "." <<
     std::endl;
98        }
99
100       // Вызов планировщика
101       if(sched_setscheduler(getpid(), policy, &schedParam) == -1) {
102         std::cerr << "It's impossible to set sheduler. Second child pid #" << getpid() <<
     ", ppid #" << getppid() << "." << std::endl;
103         return 0x8;
104       }
105
106       std::cout << "Second child pid #" << getpid() << ", ppid #" << getppid() << "." <<
    std::endl;
107     }
108
109     if(sched_setscheduler(getpid(), policy, &schedParam) == -1) {
110       std::cerr << "It's impossible to set sheduler. First child pid #" << getpid() << ",
     ppid #" << getppid() << "." << std::endl;
111       return 0x6;
112     }
113
114     std::cout << "First child pid #" << getpid() << ", ppid #" << getppid() << "." << std
    ::endl;
115   }
116
117   return 0x0;
118 }
```

Пример работы программы с алгоритмом планирования *SCHED_FIFO* (первый аргумент в командной строке 0):

```
1  stakenschneider@stakenschneider:~/temp$ g++ -std=c++11 p1.10.1.cpp singlecore.c -o p1
       .10.1
2
3  stakenschneider@stakenschneider:~/temp$ sudo ./p1.10.1 0 64 40 30 20
4  Parent pid #6626, ppid #6625.
5  First child pid #6627, ppid #2284.
6  Second child pid #6628, ppid #2284.
7  First child pid #6628, ppid #2284.
8  Third child pid #6629, ppid #2284.
9  Second child pid #6629, ppid #2284.
10 First child pid #6629, ppid #2284.
11
12 stakenschneider@stakenschneider:~/temp$ sudo ./p1.10.1 0 74 20 20 20
13 Parent pid #6676, ppid #6675.
14 Third child pid #6679, ppid #6678.
15 Second child pid #6679, ppid #6678.
16 First child pid #6679, ppid #6678.
17 Second child pid #6678, ppid #6677.
18 First child pid #6678, ppid #6677.
19 First child pid #6677, ppid #2284.
20
21 stakenschneider@stakenschneider:~/temp$ sudo ./p1.10.1 0 64 31 40 64
22 Parent pid #6699, ppid #6698.
23 Third child pid #6702, ppid #6701.
24 Second child pid #6702, ppid #6701.
25 First child pid #6702, ppid #6701.
26 Second child pid #6701, ppid #6700.
27 First child pid #6701, ppid #6700.
28 First child pid #6700, ppid #2284.
```

Пример работы программы с алгоритмом планирования *SCHED_RR* (первый аргумент в командной строке 1):

```
stakenschneider@stakenschneider:~/temp$ sudo ./p1.10.1 1 64 40 30 20
Parent pid #6740, ppid #6739.
First child pid #6741, ppid #2284.
Second child pid #6742, ppid #2284.
First child pid #6742, ppid #2284.
Third child pid #6743, ppid #2284.
Second child pid #6743, ppid #2284.
First child pid #6743, ppid #2284.

stakenschneider@stakenschneider:~/temp$ sudo ./p1.10.1 1 70 20 20 20
Parent pid #6755, ppid #6754.
Third child pid #6758, ppid #6757.
Second child pid #6758, ppid #6757.
First child pid #6758, ppid #6757.
Second child pid #6757, ppid #6756.
First child pid #6757, ppid #6756.
First child pid #6756, ppid #2284.

stakenschneider@stakenschneider:~/temp$ sudo ./p1.10.1 1 64 31 40 64
Parent pid #6768, ppid #6767.
Third child pid #6771, ppid #6770.
Second child pid #6771, ppid #6770.
First child pid #6771, ppid #6770.
Second child pid #6770, ppid #6769.
First child pid #6770, ppid #6769.
First child pid #6769, ppid #2284.
```

Результаты выполнения *SCHED_RR* оказались аналогичными с *SCHED_FIFO*.

Пример работы программы с алгоритмом планирования *SCHED_OTHER* (первый аргумент в командной строке 2):

```
stakenschneider@stakenschneider:~/temp$ sudo ./p1.10.1 2 0 0 0 0
Parent pid #6783, ppid #6782.
First child pid #6784, ppid #2284.
Second child pid #6785, ppid #2284.
First child pid #6785, ppid #2284.
Third child pid #6786, ppid #2284.
Second child pid #6786, ppid #2284.
First child pid #6786, ppid #2284.
```

При вложенных процессах первым выполняется процесс с наибольшим приоритетом. При равных приоритетах, процессы выполняются по принципу *FIFO*.

10.2. Разные процедуры планирования разным процессам с одинаковыми приоритетами

Разработаем программу, в которой у процессов-потомков одинаковый приоритет, а алгоритмы планирования задаются аргументами командной строки:

```
#include <iostream>
#include <unistd.h>
#include <sched.h>
#include "singlecore.h"

const int DEFAULT_COEFF = 50;
const int DELAY = 30;
const int EXIT_DELAY = DELAY * 20;

void setPolicy(int* policy, const char* argument) throw(std::invalid_argument);

int main(int argc, char** argv) {
  // Проверка на количество аргументов, если неправильное, то выходим с ошибкой
  if(argc != 5) {
    std::cerr << "Invalid count of arguments." << std::endl;
```

```
16      return 0x1;
17    }
18
19    int policyParent, policyChildFirst, policyChildSecond, policyChildThird;
20
21    try {
22      // лгоритмA планирования родителя
23      setPolicy(&policyParent, argv[1]);
24      // лгоритмA планирования первого потомка
25      setPolicy(&policyChildFirst, argv[2]);
26      // лгоритмA планирования второго потомка
27      setPolicy(&policyChildSecond, argv[3]);
28      // лгоритмA планирования третьего потомка
29      setPolicy(&policyChildThird, argv[4]);
30    }
31    catch(const std::exception& exception) {
32      // Обработка ситуации неверного аргумента
33      std::cerr << "Invalid argument." << std::endl;
34      return 0x2;
35    }
36
37    struct sched_param schedParam;
38    schedParam.sched_priority = DEFAULT_COEFF;
39
40    // Parent
41
42    int pid = getpid();
43    int ppid = getppid();
44
45    // Если не получилось перераспределить ресурсы процессорра на одно ядро, то выходим с ошибкой
46    if(disableMultithreading(pid) < 0)
47      return 0x3;
48
49    // Записываем алгоритм планирования в поле структуры
50    schedParam.sched_priority = (policyParent == SCHED_OTHER) ? 0 : DEFAULT_COEFF;
51
52    // Вызов планировщика
53    if(sched_setscheduler(0, policyParent, &schedParam) == -1) {
54      std::cerr << "It's impossible to set sheduler. Parent pid #" << pid << ", ppid #" <<
        ppid << "." << std::endl;
55      return 0x4;
56    }
57
58    std::cout << "Parent pid #" << pid << ", ppid #" << ppid << "." << std::endl;
59
60    if(!fork()) {
61      // First child
62
63      pid = getpid();
64      ppid = getppid();
65
66      // Если не получилось перераспределить ресурсы процессорра на одно ядро, то выходим с ошибкой
67      if(disableMultithreading(pid) < 0)
68        return 0x5;
69
70      // Записываем алгоритм планирования в поле структуры
71      schedParam.sched_priority = (policyChildFirst == SCHED_OTHER) ? 0 : DEFAULT_COEFF;
72      // Вызов планировщика
73      if(sched_setscheduler(pid, policyChildFirst, &schedParam) == -1) {
74          std::cerr << "It's impossible to set sheduler. First child pid #" << pid << ",
        ppid #" << ppid << "." << std::endl;
75          return 0x6;
76      }
77
78      usleep(DELAY);
79
```

```
80      std::cout << "First child pid #" << pid << ", ppid #" << ppid << "." << std::endl;
81    }
82
83    if(!fork()) {
84      // Second child
85
86      pid = getpid();
87      ppid = getppid();
88
89      // Если не получилось перераспределить ресурсы процессорра на одно ядро, то выходим с ошибкой
90      if(disableMultithreading(pid) < 0)
91        return 0x7;
92
93      // Записываем алгоритм планирования в поле структуры
94      schedParam.sched_priority = (policyChildSecond == SCHED_OTHER) ? 0 : DEFAULT_COEFF;
95      // Вызов планировщика
96      if(sched_setscheduler(pid, policyChildSecond, &schedParam) == -1) {
97          std::cerr << "It's impossible to set sheduler. Second child pid #" << pid << ",
        ppid #" << ppid << "." << std::endl;
98          return 0x8;
99      }
100
101     usleep(DELAY);
102
103     std::cout << "Second child pid #" << pid << ", ppid #" << ppid << "." << std::endl;
104   }
105
106   if(!fork()) {
107     // Third child
108
109     pid = getpid();
110     ppid = getppid();
111
112     // Если не получилось перераспределить ресурсы процессорра на одно ядро, то выходим с ошибкой
113     if(disableMultithreading(pid) < 0)
114       return 0x9;
115
116     // Записываем алгоритм планирования в поле структуры
117     schedParam.sched_priority = (policyChildThird == SCHED_OTHER) ? 0 : DEFAULT_COEFF;
118     // Вызов планировщика
119     if(sched_setscheduler(pid, policyChildThird, &schedParam) == -1) {
120         std::cerr << "It's impossible to set sheduler. Third child pid #" << pid << ",
        ppid #" << ppid << "." << std::endl;
121         return 0xA;
122     }
123
124     usleep(DELAY);
125
126     std::cout << "Third child pid #" << pid << ", ppid #" << ppid << "." << std::endl;
127   }
128
129   usleep(EXIT_DELAY);
130
131   return 0x0;
132 }
133
134 void setPolicy(int* policy, const char* argument) throw(std::invalid_argument) {
135   switch(std::stoi(argument)) {
136     case 0: *policy = SCHED_FIFO; break;
137     case 1: *policy = SCHED_RR; break;
138     case 2: *policy = SCHED_OTHER; break;
139     default: throw std::invalid_argument(argument);
140   }
141 }
```

Результаты выполнения программы с разными алгоритмами планирования:

```
stakenschneider@stakenschneider:~/temp$ g++ -std=c++11 p1.10.2.cpp singlecore.c -o p1
    .10.2

stakenschneider@stakenschneider:~/temp$ sudo ./p1.10.2 0 0 0 0
Parent pid #3780, ppid #3779.
First child pid #3781, ppid #3780.
Second child pid #3782, ppid #3780.
Third child pid #3783, ppid #3780.
Second child pid #3784, ppid #3781.
Third child pid #3785, ppid #3781.
Third child pid #3786, ppid #3782.
Third child pid #3787, ppid #3784.

stakenschneider@stakenschneider:~/temp$ sudo ./p1.10.2 2 1 0 1
Parent pid #4176, ppid #4175.
Third child pid #4179, ppid #4176.
Second child pid #4178, ppid #4176.
Third child pid #4180, ppid #4178.
First child pid #4177, ppid #2297.
Second child pid #4181, ppid #4177.
Third child pid #4182, ppid #4177.
Third child pid #4183, ppid #4181.

stakenschneider@stakenschneider:~/temp$ sudo ./p1.10.2 1 0 1 0
Parent pid #4227, ppid #4226.
First child pid #4228, ppid #4227.
Second child pid #4229, ppid #4227.
Third child pid #4230, ppid #4227.
Second child pid #4231, ppid #4228.
Third child pid #4232, ppid #4228.
Third child pid #4233, ppid #4229.
Third child pid #4234, ppid #4231.

stakenschneider@stakenschneider:~/temp$ sudo ./p1.10.2 1 0 0 1
Parent pid #4240, ppid #4239.
First child pid #4241, ppid #4240.
Second child pid #4242, ppid #4240.
Third child pid #4243, ppid #4240.
Second child pid #4244, ppid #4241.
Third child pid #4245, ppid #4241.
Third child pid #4246, ppid #4242.
Third child pid #4247, ppid #4244.

stakenschneider@stakenschneider:~/temp$ sudo ./p1.10.2 2 2 2 2
Parent pid #4264, ppid #4263.
Third child pid #4267, ppid #4264.
Second child pid #4266, ppid #4264.
First child pid #4265, ppid #2297.
Third child pid #4268, ppid #4266.
Second child pid #4269, ppid #2297.
Third child pid #4270, ppid #2297.
Third child pid #4271, ppid #4269.
```

Эксперимент №1: все четыре политики планирования *SCHED_FIFO*. Так как у всех процессов политика одинаковая, они выполняются согласно порядку попадания в очередь.

Эксперимент №2: политики планирования соответственно *SCHED_OTHER*, *SCHED_RR*, *SCHED_FIFO*, *SCHED_RR*. Второй и третий процесс выполнились раньше, чем первый.

Эксперимент №3: политики планирования соответственно *SCHED_RR*, *SCHED_FIFO*, *SCHED_RR*, *SCHED_FI* Процессы выполнились согласно порядку попадания в очередь.

Эксперимент №4: политики планирования соответственно *SCHED_RR*, *SCHED_FIFO*, *SCHED_FIFO*, *SCHED_RR*. Процессы выполнились согласно порядку попадания в очередь.

Эксперимент №5: политики планирования соответственно *SCHED_OTHER*, *SCHED_OTHER*, *SCHED_OTHER*, *SCHED_OTHER*. Второй и третий процесс выполнились раньше, чем первый.

Делаем вывод, что любые комбинации комбинации *SCHED_RR* и *SCHED_FIFO* без *SCHED_OTHER* никак не влияют на порядок выполнения процессов. Это объясняется тем, что *SCHED_RR* и *SCHED_FIFO*

во всех экспериментах до этого показали абсолютно одинаковые результаты.

11. Определение величины кванта, попытка ее изменения

Определить длину кванта можно с помощью функции:

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

Разработаем программу, определяющую длину кванта:

```
#include <iostream>
#include <sched.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <unistd.h>
#include "singlecore.h"

const int DEFAULT_COEFF = 50;
const float SECONDS_COEFF = 1e9;

int main() {
  struct sched_param schedParam;
  struct timespec time;

  // Parent

  int pid = getpid();
  int ppid = getppid();

  std::cout << "Parent pid #" << pid << ", ppid #" << ppid << "." << std::endl;

  // Если не получилось перераспределить ресурсы процессорра на одно ядро, то выходим с ошибкой
  if(disableMultithreading(pid) < 0)
    return 0x1;

  // Записываем алгоритм планирования в поле структуры
  schedParam.sched_priority = DEFAULT_COEFF;

  // Вызов планировщика
  if(sched_setscheduler(0, SCHED_RR, &schedParam) == -1) {
    std::cerr << "It's impossible to set sheduler. Parent pid #" << pid << ", ppid #" <<
    ppid << "." << std::endl;
    return 0x2;
  }

  // Определение длины кванта
  if(sched_rr_get_interval(0, &time) == 0) {
    std::cout << "Parent " << double(time.tv_nsec) / SECONDS_COEFF + time.tv_sec << "
    seconds." << std::endl;
  }
  else {
    std::cerr << "It's impossible to get sheduler interval. Parent pid #" << pid << ",
    ppid #" << ppid << "." << std::endl;
    return 0x3;
  }


  if(!fork()) {
    // Child

    pid = getpid();
    ppid = getppid();

    std::cout << "First child pid " << pid << ", ppid " << ppid << "." << std::endl;

    // Если не получилось перераспределить ресурсы процессорра на одно ядро, то выходим с ошибкой
    if(disableMultithreading(pid) < 0)
```

```
55        return 0x3;
56
57      // Определение длины кванта
58      if(sched_rr_get_interval(0, &time) == 0) {
59        std::cout << "Child " << double(time.tv_nsec) / SECONDS_COEFF + time.tv_sec << "
      seconds." << std::endl;
60        execl("p1.9.2.c1", "p1.9.2.c1", NULL);
61      }
62      else {
63        std::cerr << "It's impossible to get sheduler interval. Parent pid #" << pid << ",
      ppid #" << ppid << "." << std::endl;
64        return 0x4;
65      }
66    }
67
68    return 0x0;
69  }
```

Результат выполнения программы:

```
stakenschneider@stakenschneider:~/temp$ g++ -std=c++11 p1.11.cpp singlecore.c -o p1.11
stakenschneider@stakenschneider:~/temp$ sudo ./p1.11
Parent pid #4130, ppid #4129.
Parent 0.1 seconds.
First child pid 4131, ppid 2292.
Child 0.1 seconds.
Child #1 started, pid 4131, ppid 2292.
```

Попробуем изменить длину кванта, посредством задания ему наивысшего приоритета:

```
#include <iostream>
#include <sched.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <unistd.h>
#include "singlecore.h"

const int DEFAULT_COEFF = 99;
const int PRIORITY_FIRST_COEFF = -20;
const int PRIORITY_SECOND_COEFF = 40;
const float SECONDS_COEFF = 1e9;

int main() {
  struct sched_param schedParam;
  struct timespec time;

  // Parent

  int pid = getpid();
  int ppid = getppid();

  std::cout << "Parent pid #" << pid << ", ppid #" << ppid << "." << std::endl;

  // Если не получилось перераспределить ресурсы процессорра на одно ядро, то выходим с ошибкой
  if(disableMultithreading(pid) < 0)
    return 0x1;

  // Устанавливаем приоритет, для увеличения длины кванта
  nice(PRIORITY_FIRST_COEFF);

  // Записываем алгоритм планирования в поле структуры
  schedParam.sched_priority = DEFAULT_COEFF;

  // Вызов планировщика
  if(sched_setscheduler(0, SCHED_RR, &schedParam) == -1) {
    std::cerr << "It's impossible to set sheduler. Parent pid #" << pid << ", ppid #" <<
    ppid << "." << std::endl;
```

```
37      return 0x2;
38    }
39
40    // Определение длины кванта
41    if(sched_rr_get_interval(0, &time) == 0) {
42      std::cout << float(time.tv_nsec) / SECONDS_COEFF + time.tv_sec << " seconds." << std
        ::endl;
43    }
44    else {
45      std::cerr << "It's impossible to get sheduler interval. Parent pid #" << pid << ",
        ppid #" << ppid << "." << std::endl;
46      return 0x3;
47    }
48
49    // Устанавливаем приоритет, для увеличения длины кванта
50    nice(PRIORITY_SECOND_COEFF);
51
52    // Записываем алгоритм планирования в поле структуры
53    schedParam.sched_priority = DEFAULT_COEFF;
54
55    // Вызов планировщика
56    if(sched_setscheduler(0, SCHED_RR, &schedParam) == -1) {
57      std::cerr << "It's impossible to set sheduler. Parent pid #" << pid << ", ppid #" <<
        ppid << "." << std::endl;
58      return 0x4;
59    }
60
61    // Определение длины кванта
62    if(sched_rr_get_interval(0, &time) == 0) {
63      std::cout << float(time.tv_nsec) / SECONDS_COEFF + time.tv_sec << " seconds." << std
        ::endl;
64    }
65    else {
66      std::cerr << "It's impossible to get sheduler interval. Parent pid #" << pid << ",
        ppid #" << ppid << "." << std::endl;
67      return 0x5;
68    }
69
70    return 0x0;
71  }
```

Результат выполнения программы:

```
1 stakenschneider@stakenschneider:~/temp$ g++ -std=c++11 p1.11.1.cpp singlecore.c -o p1
      .11.1
2 stakenschneider@stakenschneider:~/temp$ sudo ./p1.11.1
3 Parent pid #4281, ppid #4280.
4 0.1 seconds.
5 0.1 seconds.
```

Квант изменить не удалось, однако в других *ОС*, поддерживающих *POSIX*, значение кванта можно менять, в том числе из системных приложений, оптимизируя функционирование прикладных задач. Особенно это существенно для ОС реального времени и систем технического обслуживания.

12. Анализ наследования на этапах fork() и exec()

Файловые дескрипторы наследуются при системном вызове функции *fork*, также они не закрываются при вызове функции *exec*.

Проведем эксперимент: попробуем прочитать содержимое файла в процессе-родителе, процессе-потомке и дочернем процессе. При этом дескриптор открывается один раз и не закрывается:

```
1 #include <iostream>
2 #include <fcntl.h>
3 #include <unistd.h>
4 #include <stdlib.h>
5 #include <stdio.h>
6 #include "singlecore.h"
```

```
7
8  #define FILE_LENGTH 15
9  #define STRING_LENGTH 15
10
11 int main(int argc, char* argv[]) {
12   // Открываем файл на чтение
13   int file = open("readfile.txt", O_RDONLY);
14   if(file < 0) {
15     // В случае неудачи выводим сообщение об ошибке
16     perror("It's impossible to open file.");
17     return 0x1;
18   }
19
20   // Если не получилось перераспределить ресурсы процессорра на одно ядро, то выходим с ошибкой
21   if(disableMultithreading(getpid()) < 0)
22     return 0x2;
23
24   if(!fork()) {
25     // Child
26
27     printf("Child pid %d, ppid %d. Read from file: ", getpid(), getppid());
28
29     // Считываем побайтово файл и выводим его содержимое
30     char symbol;
31     int index = 0;
32     while(index < FILE_LENGTH) {
33       pread(file, &symbol, 1, index++);
34       printf("%c", symbol);
35     }
36     printf("\n");
37
38     // Передаем дескриптор в качестве аргумента
39     char buffer[STRING_LENGTH];
40     sprintf(buffer, "%d", file);
41     execl("p1.12.ch", "p1.12.ch", buffer, NULL);
42   }
43   else {
44     // Parent
45
46     printf("Parent pid %d, ppid %d. Read from file: ", getpid(), getppid());
47
48     // Считываем побайтово файл и выводим его содержимое
49     char symbol;
50     int index = 0;
51     while(index < FILE_LENGTH) {
52       pread(file, &symbol, 1, index++);
53       printf("%c", symbol);
54     }
55     printf("\n");
56
57     // Закрываем дескриптор
58     close(file);
59   }
60
61
62 }
```

Дочерняя программа:

```
1  #include <iostream>
2  #include <unistd.h>
3
4  #define FILE_LENGTH 15
5  #define STRING_LENGTH 15
6
7  int main(int argc, char* argv[]) {
8    // Проверка на количество аргументов, если неправильное, то выходим с ошибкой
```

```
  if(argc > 2) {
    std::cerr << "Wrong count of arguments." << std::endl;
    return 0x1;
  }

  int file;
  try { file = std::stoi(argv[1]); }
  catch(const std::exception& exception) {
    // Обработка ситуации неверного аргумента
    std::cerr << "Wrong argument." << std::endl;
    return 0x2;
  }

  std::cout << "Process pid " << getpid() << ", ppid " << getppid() << ". Read from file:
     ";

  // Считываем побайтово файл и выводим его содержимое
  char symbol;
  int index = 0;
  while(index < FILE_LENGTH) {
    pread(file, &symbol, 1, index++);
    std::cout << symbol;
  }

  std::cout << std::endl;

  return 0x0;
}
```

Результат выполнения программы:

```
stakenschneider@stakenschneider:~/temp$ g++ p1.12.c singlecore.c -o p1.12
stakenschneider@stakenschneider:~/temp$ g++ -std=c++11 p1.12.ch.cpp -o p1.12.ch
Parent pid 5155, ppid 3961. Read from file: some text there
Child pid 5156, ppid 2292. Read from file: some text there
Process pid 5156, ppid 2292. Read from file: some text there
```

Все процессы считали содержимое файла, что подтверждает возможность единожды отрывать файловый дескриптор, которым могут пользоваться потомки и дочерние процессы.

1.3.2 Глава 2. Взаимодействие родственных процессов

1.a. Процесс-родитель запускает процесс-потомок и ожидает его завершения

Рассмотрим ситуацию, когда процесс-родитель несколько секунд ожидает дочерний процесс:

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

int main() {
  printf("Parent pid %d, ppid %d.\n", getpid(), getppid());

  if(!fork()) {
    // Создаем процесспотомок–
    execl("p2.1.1.ch", "p2.1.1.ch", NULL);
  }

  // Ожидаем завершение потомка
  int status;
  wait(&status);

  printf("Parent has been finished.\n");

  return 0x0;
}
```

Дочерняя программа:

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

#define DELAY 3

int main() {
  sleep(DELAY);
  printf("Child pid %d, ppid %d.\n", getpid(), getppid());
  printf("Child has been finished.\n");

  return 0x0;
}
```

Результат выполнения программы:

```
stakenschneider@stakenschneider:~/temp$ gcc p2.1.1.c -o p2.1.1
stakenschneider@stakenschneider:~/temp$ gcc p2.1.1.ch.c -o p2.1.1.ch
stakenschneider@stakenschneider:~/temp$ ./p2.1.1
Parent pid 5732, ppid 5671.
Child pid 5733, ppid 5732.
Child has been finished.
Parent has been finished.
```

Процесс-родитель дождался завершения процесса-потомка и только после этого завершился сам.

1.b. Процесс-родитель запускает процесс-потомок и не ожидает его завершения

Рассмотрим ситуацию, когда процесс-родитель запускает дочерний процесс, но не ожидает его завершения:

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

int main() {
  printf("Parent pid %d, ppid %d.\n", getpid(), getppid());

  if(!fork()) {
    // Создаем процесспотомок–
    execl("p2.1.1.ch", "p2.1.1.ch", NULL);
  }

  printf("Parent has been finished.\n");

  return 0x0;
}
```

Результат выполнения программы:

```
stakenschneider@stakenschneider:~/temp$ gcc p2.1.1.ch.c -o p2.1.1.ch
stakenschneider@stakenschneider:~/temp$ gcc p2.1.2.c -o p2.1.2
stakenschneider@stakenschneider:~/temp$ ./p2.1.2
Parent pid 5866, ppid 5671.
Parent has been finished.
Child pid 5867, ppid 1.
Child has been finished.
```

Так как родительский процесс завершился раньше дочернего, мы зафиксировали изменение *PID* дочернего процесса на единицу. Это значит что родительский процесс этого процесса теперь корневой процесс *init*, который запускается ядром при загрузке системы. Его задача - усыновлять процессы, которые остались без родительского процесса.

1.c. Процесс-родитель запускает процесс-потомок и не ожидает его завершения, фиксация зомби процесса

Модифицируем программу, добавив задержку, которая больше чем в дочернем процессе и системные вызовы таблицы процессов:

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

#define DELAY 5

int main() {
  printf("Parent pid %d, ppid %d.\n", getpid(), getppid());

  if(!fork()) {
    // Создаем процесспотомок–
    execl("p2.1.1.ch", "p2.1.1.ch", NULL);
  }

  system("ps xf > p2.1.3.z1.log");
  sleep(DELAY);
  system("ps xf > p2.1.3.z2.log");

  printf("Parent has been finished.\n");

  return 0x0;
}
```

Состояния процессов во время выполнения программы:

```
5671 pts/4    Ss     0:00  |   \_ bash
 6256 pts/4    S+     0:00  |       \_ ./p2.1.3
 6257 pts/4    S+     0:00  |           \_ p2.1.1.ch
 6258 pts/4    S+     0:00  |           \_ sh -c ps xf > p2.1.3.z1.log
 6259 pts/4    R+     0:00  |               \_ ps xf
 5795 ?        Sl     0:54  \_ gedit /home/nikita/temp/p2.1.1.log
 2286 ?        Ss     0:00 /lib/systemd/systemd --user
 2289 ?        S      0:00  \_ (sd-pam)

( ... )

 5671 pts/4    Ss     0:00  |   \_ bash
 6256 pts/4    S+     0:00  |       \_ ./p2.1.3
 6257 pts/4    Z+     0:00  |           \_ [p2.1.1.ch] <defunct>
 6260 pts/4    S+     0:00  |           \_ sh -c ps xf > p2.1.3.z2.log
 6261 pts/4    R+     0:00  |               \_ ps xf
 5795 ?        Sl     0:54  \_ gedit /home/nikita/temp/p2.1.1.log
 2286 ?        Ss     0:00 /lib/systemd/systemd --user
 2289 ?        S      0:00  \_ (sd-pam)
```

Дочерний процесс завершает задачу раньше родительского процесса, у него отнимаются ресурсы и он становится зомби-процессом. Он полностью удалится только после завершения родительского процесса.

2. Создание скрипта, запускающего все три программы

Создадим скрипт, запускающий все три программы:

```
#!/bin/bash

echo 'First program result: '
./p2.1.1
echo 'Second program result: '
./p2.1.2
echo 'Third program result: '
./p2.1.3
```

Запустим скрипт и перенаправим вывод в файл:

```
stakenschneider@stakenschneider:~/temp$ sudo sh p2.2.sh > p2.2.r.log
```

Файл с результатом выполнения программ:

```
First program result:
Child pid 3806, ppid 3805.
Child has been finished.
Parent pid 3805, ppid 3804.
Parent has been finished.

Second program result:
Parent pid 3807, ppid 3804.
Parent has been finished.
Child pid 3808, ppid 2362.
Child has been finished.

Third program result:
Child pid 3811, ppid 3810.
Child has been finished.
Parent pid 3810, ppid 3804.
Parent has been finished.
```

Отклонений в работе программ не обнаружено.

1.3.3 Глава 3. Управление процессами посредством сигналов

1. Выполнить команду kill с ключем -l, проанализировать результат

Выполним команду *kill* с ключем *-l*, которая выводит список всех сигналов для данной операционной системы:

```
stakenschneider@stakenschneider:~/temp$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Функция *kill* позволяет посылать любой из этих сигналов процессу по его *PID*. Функция имеет следующую сигнатуру:

```
int kill(pid_t pid, int sig);
```

Также есть утилита *kill*, выполняющая такую же функцию.

Функция *signal* позволяет задавать собственный обработчик для вышеперечисленных сигналов. Функция имеет следующую сигнатуру:

```
int signal(int sig, void (*func)(int));
```

Если значение передаваемой функции равно *SIG_DFL*, то устанавливается обработчик по умолчанию, если значение передаваемой функции равно *SIG_IGN*, то сигнал игнорируется.

1.a. - 1.c. Родительский процесс порождает три дочерних процесса. Первый дочерний процесс содержит обработчик сигнала по умолчанию, второй игнорирует сигнал, третий обрабатывает и выводит сообщение. Анализ, с помощью вызова ps -s

Первая программа обрабатывает сигнал по умолчанию:

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

int main() {
  printf("First child pid %d, ppid %d.\n", getpid(), getppid());

  // Обработка сигнала по умолчанию
  signal(SIGINT, SIG_DFL);

  // Бесконечный цикл, программа завершится только внешним сигналом
  while(1);

  return 0x0;
}
```

Вторая программа игнорирует сигнал:

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

#define DELAY 7

int main() {
  printf("Second child pid %d, ppid %d.\n", getpid(), getppid());

  // Игнорирование сигнала
  signal(SIGINT, SIG_IGN);

  // Задержка перед выходом программы
  sleep(DELAY);

  return 0x0;
}
```

Третья программа содержит собственный обработчик сигнала:

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

void handler(int signal);

int main() {
  printf("Third child pid %d, ppid %d.\n", getpid(), getppid());

  // Игнорирование сигнала
  signal(SIGINT, handler);

  // Бесконечный цикл, программа завершится только внешним сигналом
  while(1);

  return 0x0;
}

void handler(int signal) {
  printf("Signal handle");
  exit(0x0);
}
```

Программа родитель, запускающая все три дочерних процесса, и завершающая их функцией *kill*:

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

```

```
int main() {
  printf("Parent pid %d, ppid %d.\n", getpid(), getppid());

  int firstPid, secondPid, thirdPid;

  if(!fork()) {
    firstPid = getpid();
    // Вызов программы, которая обрабатывает сигнал по умолчанию
    execl("p3.1.1.ch", "p3.1.1.ch", NULL);
  }

  if(!fork()) {
    secondPid = getpid();
    // Вызов программы, которая игнорирует посылаемый сигнал
    execl("p3.1.2.ch", "p3.1.2.ch", NULL);
  }

  if(!fork()) {
    thirdPid = getpid();
    // Вызов программы, с собственным обработчиком сигнала
    execl("p3.1.3.ch", "p3.1.3.ch", NULL);
  }

  // Запись таблицы процессов в файл
  system("ps -s > p3.1.0.log");

  // Посылаем сигнал первой программе
  kill(firstPid, SIGINT);
  // Запись таблицы процессов в файл
  system("ps -s > p3.1.1.log");

  // Посылаем сигнал второй программе
  kill(secondPid, SIGINT);
  // Запись таблицы процессов в файл
  system("ps -s > p3.1.2.log");

  // Посылаем сигнал третьей программе
  kill(thirdPid, SIGINT);
  // Запись таблицы процессов в файл
  system("ps -s > p3.1.3.log");

  return 0x0;
}
```

Результат эксперимента:

```
stakenschneider@stakenschneider:~/temp$ gcc p3.1.p.c -o p3.1.p
stakenschneider@stakenschneider:~/temp$ gcc p3.1.1.ch.c -o p3.1.1.ch
stakenschneider@stakenschneider:~/temp$ gcc p3.1.2.ch.c -o p3.1.2.ch
stakenschneider@stakenschneider:~/temp$ gcc p3.1.3.ch.c -o p3.1.3.ch

stakenschneider@stakenschneider:~/temp$ ./p3.1.p
Parent pid 5142, ppid 5012.
First child pid 5143, ppid 5142.
Second child pid 5144, ppid 5142.
Third child pid 5145, ppid 5142.
Signal handle
```

Первый сигнал был обработан по умолчанию и процесс завершился. Второй сигнал был ожидаемо проигнорирован и процесс завершился после некоторой задержки. Третий процесс обработал сигнал и вывел сообщение, после чего завершился.

Рассмотрим результаты залогированных вызовов *ps -s*. Вызов до первого вызова *kill*:

```
  UID   PID  STAT TTY        TIME COMMAND
( ... )
    0  5250  S+   pts/1      0:00 sudo ./p3.1.p
    0  5251  S+   pts/1      0:00 ./p3.1.p
```

```
    0  5252  R+   pts/1      0:00 p3.1.1.ch
    0  5253  S+   pts/1      0:00 p3.1.2.ch
    0  5254  R+   pts/1      0:00 p3.1.3.ch
    0  5255  S+   pts/1      0:00 sh -c ps -s > p3.1.0.log
    0  5256  R+   pts/1      0:00 ps -s
```

Вызов *ps -s* после первого вызова *kill*:

```
  UID   PID  STAT TTY      TIME COMMAND
( ... )
    0  5250  S+   pts/1      0:00 sudo ./p3.1.p
    0  5251  S+   pts/1      0:00 ./p3.1.p
    0  5252  R+   pts/1      0:00 [p3.1.1.ch] <defunct>
    0  5253  S+   pts/1      0:00 p3.1.2.ch
    0  5254  R+   pts/1      0:00 p3.1.3.ch
    0  5255  S+   pts/1      0:00 sh -c ps -s > p3.1.1.log
    0  5256  R+   pts/1      0:00 ps -s
```

Вызов *ps -s* после второго вызова *kill*:

```
  UID   PID  STAT TTY      TIME COMMAND
( ... )
    0  5250  S+   pts/1      0:00 sudo ./p3.1.p
    0  5251  S+   pts/1      0:00 ./p3.1.p
    0  5252  R+   pts/1      0:00 [p3.1.1.ch] <defunct>
    0  5253  S+   pts/1      0:00 p3.1.2.ch
    0  5254  R+   pts/1      0:00 p3.1.3.ch
    0  5255  S+   pts/1      0:00 sh -c ps -s > p3.1.2.log
    0  5256  R+   pts/1      0:00 ps -s
```

Вызов *ps -s* после третьего вызова *kill*:

```
  UID   PID  STAT TTY      TIME COMMAND
( ... )
    0  5250  S+   pts/1      0:00 sudo ./p3.1.p
    0  5251  S+   pts/1      0:00 ./p3.1.p
    0  5252  R+   pts/1      0:00 [p3.1.1.ch] <defunct>
    0  5253  S+   pts/1      0:00 p3.1.2.ch
    0  5254  R+   pts/1      0:00 [p3.1.3.ch] <defunct>
    0  5255  S+   pts/1      0:00 sh -c ps -s > p3.1.3.log
    0  5256  R+   pts/1      0:00 ps -s
```

Первый и третий процесс завершились и стали зомби, в то время как второй процесс проигнорировал сигнал завершения и завершится только после некоторой задержки.

2. Посылка сигналов двум процессам, находящимся в активном и пассивных состояниях соответственно. Фиксация времени посылки и приема каждого сигнала

Программа родитель в активном режиме запускает дочерний процесс, обрабатывает сигнал прерывания и фиксирует время завершения:

```c
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>

volatile sig_atomic_t childPid;

void handler(int signal);

int main() {
  printf("Parent pid %d, ppid %d.\n", getpid(), getppid());

  if(!(childPid = fork())) {
    // Вызываем дочерний процесс
    execl("p3.2.ch", "p3.2.ch", NULL);
```

```
  }
  else {
    // Определяем обработчик прерывания
    signal(SIGINT, handler);
    // Бесконечный цикл, программа завершится только внешним сигналом
    while(1);
  }

  return 0x0;
}

void handler(int signal) {
  // Вывод текущего системного времени
  char buffer[100];
  struct timeval timeV;
  gettimeofday(&timeV, NULL);
  time_t currentTime = timeV.tv_sec;
  strftime(buffer, 100, "%T", localtime(&currentTime));
  printf("%s.%.3ld Parent signal handle.\n", buffer, timeV.tv_usec);

  // Завершение программы
  exit(0x0);
}
```

Дочерний процесс в пассивном режиме (с помощью вызова *daemon*) обрабатывает сигнал прерывания и фиксирует время завершения:

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

int childPid;

void handler(int signal);

int main() {
  printf("Child pid %d, ppid %d.\n", getpid(), getppid());

  // Перевод программы в фоновый режим
  daemon(1, 1);

  // Определяем обработчик прерывания
  signal(SIGINT, handler);
  // Бесконечный цикл, программа завершится только внешним сигналом
  while(1);

  return 0x0;
}

void handler(int signal) {
  // Вывод текущего системного времени
  char buffer[100];
  struct timeval timeV;
  gettimeofday(&timeV, NULL);
  time_t currentTime = timeV.tv_sec;
  strftime(buffer, 100, "%T", localtime(&currentTime));
  printf("%s.%.3ld Child signal handle.\n", buffer, timeV.tv_usec);

  // Завершение программы
  exit(0x0);
}
```

Запустим программу родитель и вручную из нового окна терминала завершим оба процесса с фиксацией времени посылки:

```
stakenschneider@stakenschneider:~/temp$ date +"%H:%M:%S.%5N" & sudo kill 3527 -s 2 & date
    +"%H:%M:%S.%5N" &  sudo kill 3526 -s 2
( ... )
03:27:19.60753
03:27:19.63116
( ... )
```

Лог с фиксацией времени приема:

```
stakenschneider@stakenschneider:~/temp$ sudo ./p3.2.p
Parent pid 3526, ppid 3525.
Child pid 3527, ppid 3526.
03:27:19.616428 Child signal handle.
03:27:19.642323 Parent signal handle.
```

Дочерний процесс в фоновом режиме завершился спустя 8.898 микросекунд после посылки сигнала, родительский процесс в активном режиме завершился спустя 11.163 микросекунд после посылки сигнала. Временные задержки завершения процессов в активном и фоновом режиме отличаются незначительно, и на основании этого эксперимента нельзя точно утверждать, что процесс в активном режиме завершается медленнее, чем в пассивном.

3. Запуск в фоновом режиме нескольких утилит

Текстовый файл для отображения утилитой *cat*:

```
Text
```

Скрипт, выводящий сообщение в консоль и уведомляющий о своем завершении при помощи *notify*:

```
#!/bin/bash

echo "Script" &
```

Программа выводящая сообщение в консоль:

```
#include <stdio.h>

int main() {
  printf("Program\n");
  return 0x0;
}
```

Результат запуска нескольких утилит в фоновом режиме:

```
stakenschneider@stakenschneider:~/temp$ cat p3.3.txt & ./p3.3 & sh p3.3.sh &
[1] 3346
[2] 3347
[3] 3348
Text
Program
Script

[1]   Done                    cat p3.3.txt
[2]-  Done                    ./p3.3
[3]+  Done                    sh p3.3.sh
```

Теперь приостановим работу программы и вновь возобновим ее:

```
stakenschneider@stakenschneider:~/temp$ cat p3.3.txt & ./p3.3 & sh p3.3.sh & kill -
    sigstop %2
[1] 3346
[2] 3347
[3] 3348
Text
Script

[1]   Done                    cat p3.3.txt
[2]+  Stopped                 ./p3.3
```

```
[3]-  Done                    sh p3.3.sh

stakenschneider@stakenschneider:~/temp$ fg %2
[2]+  ./p3.3 &
Program
```

В списке *jobs* выведена информация о том, что программа приостановлена, после этого программа была восстановлена командой *fg %2* и было выведено сообщение в консоль.

4. Ознакомление с системными вызовами nice() и getpriority()

Утилита *nice* запускает процесс на выполнение с некоторым приоритетом, который варьируется от -19 до 20. Меньшее число обозначает наивысший приоритет. Большее число означает низший приоритет.

Разработаем программу, которая выводит сообщение в консоль несколько раз с некоторой периодичностью:

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

#define COUNT (int)1e6
#define PERIOD (int)1e5

int main() {
  int pid = getpid();

  for(int index = 0; index < COUNT; ++index)
    if(index % PERIOD == 0)
      printf("Pid %d, Index %d.\n", pid, index);

  return 0x0;
}
```

Рассмотрим результаты запуска двух экземпляров программ с одинаковыми приоритетами:

```
stakenschneider@stakenschneider:~/temp$ nice -5 ./p3.4.1 & nice -5 ./p3.4.1
[1] 3466
Pid 3466, Index 0.
Pid 3467, Index 0.
Pid 3466, Index 100000.
Pid 3467, Index 100000.
Pid 3466, Index 200000.
Pid 3467, Index 200000.
Pid 3467, Index 300000.
Pid 3466, Index 300000.
Pid 3467, Index 400000.
Pid 3466, Index 400000.
Pid 3467, Index 500000.
Pid 3466, Index 500000.
Pid 3467, Index 600000.
Pid 3466, Index 600000.
Pid 3467, Index 700000.
Pid 3466, Index 700000.
Pid 3467, Index 800000.
Pid 3466, Index 800000.
Pid 3467, Index 900000.
Pid 3466, Index 900000.
```

Так как приоритет одинаковый, процессы конкурируют между собой за право выполнения и ни один из них не вырывается вперед образовывая неровность выполнения.

Рассмотрим результаты запуска двух экземпляров программ с разными приоритетами:

```
stakenschneider@stakenschneider:~/temp$ nice -15 ./p3.4.1 & nice -1 ./p3.4.1
[1] 3502
Pid 3502, Index 0.
Pid 3502, Index 100000.
Pid 3503, Index 0.
Pid 3503, Index 100000.
```

```
Pid 3502, Index 200000.
Pid 3503, Index 200000.
Pid 3502, Index 300000.
Pid 3503, Index 300000.
Pid 3502, Index 400000.
Pid 3503, Index 400000.
Pid 3502, Index 500000.
Pid 3503, Index 500000.
Pid 3503, Index 600000.
Pid 3502, Index 600000.
Pid 3503, Index 700000.
Pid 3502, Index 700000.
Pid 3503, Index 800000.
Pid 3502, Index 800000.
Pid 3503, Index 900000.
Pid 3502, Index 900000.
```

Видно, что первый процесс выполнил первую и вторую итерацию раньше, чем второй, это связано с тем что его приоритет выше.

Функция *getpriority* получает текущее значение приоритета для процесса, группы или пользователя. Функция имеет следующие сигнатуры:

```
int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int prio);
```

Для определения минимального и максимального значения приоритета для конкретного алгоритма планирования можно с помощью функций *sched_get_priority_min* и *sched_get_priority_max* соответственно. Функции имеет следующие сигнатуры:

```
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

Разработаем программу, определяющую текущую политику планирования и возможные диапазоны приоритетов для этих политик:

```
#include <iostream>
#include <sched.h>
#include <unistd.h>

int main() {
  int min, max;

  // SCHED_FIFO

  min = sched_get_priority_min(SCHED_FIFO);
  max = sched_get_priority_max(SCHED_FIFO);

  std::cout << "SCHED_FIFO [" << min << ", " << max << "]" << std::endl;

  // SCHED_RR

  min = sched_get_priority_min(SCHED_RR);
  max = sched_get_priority_max(SCHED_RR);

  std::cout << "SCHED_RR [" << min << ", " << max << "]" << std::endl;

  // SCHED_OTHER

  min = sched_get_priority_min(SCHED_OTHER);
  max = sched_get_priority_max(SCHED_OTHER);

  std::cout << "SCHED_OTHER [" << min << ", " << max << "]" << std::endl;

  // Получение текущей политики планирования
  int policy = sched_getscheduler(0);

```

```
32    std::cout << "Current policy: " << ((policy == SCHED_FIFO) ? "SCHED_FIFO" : ((policy ==
        SCHED_RR) ? "SCHED_RR": "SCHED_OTHER")) << std::endl;
33
34
35    struct sched_param schedParam;
36
37    // Получение текущего приоритета
38    if(sched_getparam(0, &schedParam) != 0) {
39      // Выводим сообщение об ошибке
40      std::cerr << "It's impossible to get param." << std::endl;
41      return 0x1;
42    }
43
44    std::cout << "Current priority: " << schedParam.sched_priority << std::endl;
45
46    return 0x0;
47  }
```

Результат работы программы:

```
stakenschneider@stakenschneider:~/temp$ ./p3.4.2
SCHED_FIFO [1, 99]
SCHED_RR [1, 99]
SCHED_OTHER [0, 0]
Current policy: SCHED_OTHER
Current priority: 0
```

Делаем вывод, что алгоритм планирования по умолчанию SCHED_OTHER, а приоритет равен 0.

Рассмотрим некоторые процессы текущего пользователя и суперпользователя:

```
stakenschneider@stakenschneider:~/temp$ ps -l -u stakenschneider --sort ni
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
1 S  1000  2806  2275  0  69 -11 - 110499 poll_s ?       00:00:00 pulseaudio
4 S  1000  2269     1  0  80   0 - 11341 ep_pol ?        00:00:00 systemd
5 S  1000  2272  2269  0  80   0 - 15849 -      ?        00:00:00 (sd-pam)
4 S  1000  2275  2263  0  80   0 - 13387 poll_s ?        00:00:01 upstart
1 S  1000  2448  2275  0  80   0 -  9982 poll_s ?        00:00:00 upstart-udev-b
1 S  1000  2460  2275  0  80   0 - 10955 ep_pol ?
( ... )

stakenschneider@stakenschneider:~/temp$ ps -l -u root --sort ni
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
1 S     0     5     2  0  60 -20 -     0 -      ?        00:00:00 kworker/0:0H
1 S     0    15     2  0  60 -20 -     0 -      ?        00:00:00 kworker/1:0H
1 S     0    20     2  0  60 -20 -     0 -      ?        00:00:00 kworker/2:0H
1 S     0    25     2  0  60 -20 -     0 -      ?        00:00:00 kworker/3:0H
1 S     0    28     2  0  60 -20 -     0 -      ?        00:00:00 netns
1 S     0    29     2  0  60 -20 -     0 -      ?
( ... )
```

Процессы были отсортированы по убыванию приоритетов (столбец *NI*). Заметим, что приоритеты у обычного пользователя за редкими исключениями отличаются от нуля, в то время как у суперпользователя большой набор процессов, запущенных с наивысшим приоритетом. Это связано с необходимостью контролирования работы системы.

5. Ознакомление с командой nohup

Команда *nohup* запускает указанную команду с игнорированием сигналов потери связи *SIGHUP*. Таким образом, команда продолжает выполнение в фоновом режиме и после выхода из системы.

Разработаем программу, которая не завершается самостоятельно:

```
int main() {
  // Бесконечный цикл
  while(1);

  return 0x0;
}
```

Запустим программу с опцией *nohup*:

```
stakenschneider@stakenschneider:~/temp$ gcc p3.5.c -o p3.5

stakenschneider@stakenschneider:~/temp$ nohup ./p3.5
nohup: ignoring input and appending output to 'nohup.out'
```

Зафиксируем наличие процесса до перезагрузки:

```
stakenschneider@stakenschneider:~/temp$ ps -A | grep p3.5
 4779 pts/2    00:00:43 p3.5
```

Зафиксируем наличие процесса после перезагрузки:

```
stakenschneider@stakenschneider:~/temp$ ps -A | grep p3.5
 4779 pts/2    00:02:31 p3.5
```

5. Определение UID процесса, зафиксировать минимальное и максимальное значение PID, анализ системных процессов

Для определения *UID* процесса воспользуемся следующим набором команд:

```
stakenschneider@stakenschneider:~/temp$ whoami
stakenschneider
stakenschneider@stakenschneider:~/temp$ id -u stakenschneider
1000
stakenschneider@stakenschneider:~/temp$ id -u root
0
stakenschneider@stakenschneider:~/temp$ ps -Al
F S   UID    PID   PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0      1      0  0  80   0 - 46329 -      ?        00:00:06 systemd
1 S     0      2      0  0  80   0 -     0 -      ?        00:00:00 kthreadd
( ... )
4 S     0   2009      1  0  80   0 - 54414 -      ?        00:00:00 snapd
4 S   106   2017      1  0  80   0 - 11076 -      ?        00:00:10 dbus-daemon
4 S   109   2043      1  0  80   0 - 96673 -      ?        00:00:00 whoopsie
0 S     0   2054      1  0  80   0 -  1100 -      ?        00:00:01 acpid
4 S   104   2056      1  0  80   0 - 64099 -      ?        00:00:00 rsyslogd
4 S     0   2060      1  0  80   0 - 74615 -      ?        00:00:00 accounts-daemo
( ... )
4 S  1000   2269      1  0  80   0 - 11341 ep_pol ?        00:00:00 systemd
5 S  1000   2272   2269  0  80   0 - 15849 -      ?        00:00:00 (sd-pam)
4 S  1000   2275   2263  0  80   0 - 13387 poll_s ?        00:00:01 upstart
4 S 65534   2392   2080  0  80   0 - 14984 -      ?        00:00:00 dnsmasq
( ... )
1 S     0   4809      2  0  80   0 -     0 -      ?        00:00:00 kworker/u8:0
0 R  1000   4916   4545  0  80   0 -  8996 -      pts/2
```

В первую очередь получаем *UID* пользователя и суперпользователя по их имени. После этого вызвали таблицу процессов с флагом *-l*, для получения информации о текущем *UID* процесса.

Зафиксируем текущие минимальное и максимальное *PID* процессов в системе:

```
stakenschneider@stakenschneider:~/temp$ ps -A
  PID TTY          TIME CMD
    1 ?        00:00:06 systemd
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00 ksoftirqd/0
  ( ... )
 5119 ?        00:00:00 kworker/u8:0
 5143 pts/2    00:00:00 ps

stakenschneider@stakenschneider:~/temp$ cat /proc/sys/kernel/pid_max
32768
```

Минимальное возможное значение *PID* равно единице и всегда принадлежит процессу *systemd*. Максимальное возможное значение *PID* было получено чтением системного файла, оно равно 32768.

Системные процессы легко определить от прочих: они имеют наивысший приоритет, имеют маленькие значения *PID* и принадлежат суперпользователю. Системными процессами, например, являются: *shed* (диспетчер свопинга), *vhand* (диспетчер страничного замещения), *kmadaemon* (диспетчер памяти ядра).

1.3.4 Глава 4. Многопоточное функционирование

1. Создание программы, формирующей несколько потоков, которые выводят сообщения с определенным интервалом

В настоящее время в GNU/Linux потоки отличаются от процессов в основном набором свойств и вещами вроде вызова функций семейства *exec* в одном из потоков. То есть и процессы, и потоки являются объектами планирования для планировщика ядра, могут независимо друг от друга блокироваться, получать сигналы и т.д. Даже порождение процессов и потоков происходит схожим образом при помощи функции *clone* перечислением нужных флагов, определяющих свойства порождаемого объекта.

Рассмотрим основные отличия потока от процесса:

- Потоки всегда работают в контексте какого-то процесса: потоков без процессов не бывает.
- Потоки одного процесса совместно используют адресное пространство этого процесса, что означает, что потоки могут работать с данными друг друга без использования *IPC*, просто средствами языка программирования как с обычными переменными внутри единой программы.
- Некоторые сигналы (вроде SIGSEGV) вызывают принудительное завершение всего процесса (появление SIGSEGV связано с обнаружением нарушения адресного пространства, а это влияет на все потоки процесса), тогда как появление SIGSEGV в одном процессе обычно не влияет на другой процесс.

Так как глобальные данные, используемые несколькими потоками, не содержат чего-либо, имеющего отношение к управлению своевременностью доступа, то используется такое понятие как критическая секция: это блок кода, в котором выполняется обращение к защищаемым данным. Защита в этом случае обеспечивается использованием мьютексов и/или семафоров. Это специальные системные объекты, состояние которых нужно проверять перед выполнением критической секции и реагировать соответственно ему.

Разработаем программу, порождающую два потока. Первый поток выводит сообщение один раз в пять секунд, второй поток выводит сообщение один раз в секунду. Основной поток блокируется до завершения этих потоков:

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

#define FIRST_DELAY 5
#define SECOND_DELAY 1

// Обработчик первого потока
void* firstThread() {
  while(1) {
    printf("First thread message.\n");
    sleep(FIRST_DELAY);
  }
}

// Обработчик второго потока
void* secondThread() {
  while(1) {
    printf("Second thread message.\n");
    sleep(SECOND_DELAY);
  }
}

int main() {
  pthread_t first, second;

  // Создание и запуск потоков
  pthread_create(&first, NULL, &firstThread, NULL);
  pthread_create(&second, NULL, &secondThread, NULL);

```

```
31    // Ожидаем завершение потоков
32    pthread_join(first, NULL);
33    pthread_join(second, NULL);
34
35    return 0x0;
36  }
```

Результат выполнения программы:

```
stakenschneider@stakenschneider:~/temp$ gcc -pthread p4.1.c -o p4.1
stakenschneider@stakenschneider:~/temp$ ./p4.1
First thread message.
Second thread message.
Second thread message.
Second thread message.
Second thread message.
Second thread message.
First thread message.
Second thread message.
Second thread message.
Second thread message.
Second thread message.
Second thread message.
( ... )
```

Из результатов видно, что на пять сообщений второго потока приходится одно сообщение первого потока, что полностью соответствует ожиданиям.

Выясним, как регистрируются потоки с точки зрения системы. Для этого выполним команду *ps* с флагом *-L*:

```
stakenschneider@stakenschneider:~$ ps -eLf
UID          PID  PPID LWP  C NLWP STIME TTY          TIME     CMD
root         1    0    1    0 1    22:21 ?        00:00:04 /sbin/init splash
(...)
stakenschneider 2677 2358 2677 0 3    23:48 pts/2 00:00:00 ./p4.1
stakenschneider 2677 2358 2678 0 3    23:48 pts/2 00:00:00 ./p4.1
stakenschneider 2677 2358 2679 0 3    23:48 pts/2 00:00:00 ./p4.1
stakenschneider 2687 2351 2687 1 1    23:48 pts/6 00:00:00 bash
stakenschneider 2698 2687 2698 0 1    23:49 pts/6 00:00:00 ps -eLf
```

Информация о потоках отображается в столбцах *LWP* (Light Weight Process - облегченные процессы) и *NLWP* (Number of LWP's - количество потоков). В нашем случае система регистрирует три потока *NLWP*, у которых различные идентификаторы *LWP*.

2. Создание программы, формирующей несколько потоков, которые выводят сообщения с определенным интервалом

Модифицируем программу из предыдущего пункта, добавив в нее логирование таблицы процессов:

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

#define FIRST_DELAY 5
#define SECOND_DELAY 1

// Обработчик первого потока
void* firstThread() {
  int pid = getpid();
  int ppid = getppid();

  while(1) {
    printf("First thread message, pid %d, ppid %d.\n", pid, ppid);
    sleep(FIRST_DELAY);
  }
}

```

```
// Обработчик второго потока
void* secondThread() {
  int pid = getpid();
  int ppid = getppid();

  while(1) {
    printf("Second thread message, pid %d, ppid %d.\n", pid, ppid);
    sleep(SECOND_DELAY);
  }
}

int main() {
  pthread_t first, second;

  system("ps -axhf > p4.2.1.log");

  // Создание и запуск первого потока
  pthread_create(&first, NULL, &firstThread, NULL);
  system("ps -axhf > p4.2.2.log");

  // Создание и запуск второго потока
  pthread_create(&second, NULL, &secondThread, NULL);
  system("ps -axhf > p4.2.3.log");

  // Ожидаем завершение первого потока
  pthread_join(first, NULL);

  // Ожидаем завершение второго потока
  pthread_join(second, NULL);

  return 0x0;
}
```

Запустим программу и завершим ее командой *kill* из соседнего терминала.

Рассмотрим логи таблиц процессов:

```
( ... )
5292 pts/2    Ss     0:00              \_ bash
5557 pts/2    S+     0:00              |   \_ ./p4.2
5558 pts/2    S+     0:00              |       \_ sh -c ps -axhf > p4.2.1.log
5559 pts/2    R+     0:00              |           \_ ps -axhf
5519 pts/6    Ss+    0:00              \_ bash
( ... )
```

```
( ... )
5292 pts/2    Ss     0:00              \_ bash
5557 pts/2    Sl+    0:00              |   \_ ./p4.2
5561 pts/2    S+     0:00              |       \_ sh -c ps -axhf > p4.2.2.log
5562 pts/2    R+     0:00              |           \_ ps -axhf
5519 pts/6    Ss+    0:00              \_ bash
( ... )
```

```
( ... )
5292 pts/2    Ss     0:00              \_ bash
5557 pts/2    Sl+    0:00              |   \_ ./p4.2
5565 pts/2    S+     0:00              |       \_ sh -c ps -axhf > p4.2.3.log
5566 pts/2    R+     0:00              |           \_ ps -axhf
5519 pts/6    Ss+    0:00              \_ bash
( ... )
```

Можно заметить, что ни на одном из этапов программы разделения на дочерние процессы не происходит, и оба потока являются одним процессом.

3. Модификация программы с завершением потока посредством сигнала SIGUSR1

Модифицируем программу, добавив код, который через определенное количество секунд завершит вторую нить из первой:

```c
#include <pthread.h>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

#define FIRST_DELAY 5
#define SECOND_DELAY 1
#define KILL_DELAY 5

int interrupt = 0;

void handler(int signal);
void* firstThread();
void* secondThread();

int main() {
  pthread_t first, second;

  // Создание и запуск первого потока
  pthread_create(&first, NULL, &firstThread, (void*) &second);

  // Создание и запуск второго потока
  pthread_create(&second, NULL, &secondThread, NULL);

  // Ожидаем завершение первого потока
  pthread_join(first, NULL);

  // Ожидаем завершение второго потока
  pthread_join(second, NULL);

  return 0x0;
}

void handler(int signal) {
  printf("Signal arrived.\n");
  // Прерываем второй поток
  interrupt = 1;
}

// Обработчик первого потока
void* firstThread(void* secondThread) {
  int pid = getpid();
  int ppid = getppid();

  sleep(KILL_DELAY);

  // Завершаем поток с сигналом SIGUSR1
  // Обращение к потоку происходят по указателю, перданному через аргумент функции
  pthread_kill(*((pthread_t*)secondThread), SIGUSR1);

  while(1) {
    printf("First thread message, pid %d, ppid %d.\n", pid, ppid);
    sleep(FIRST_DELAY);
  }
}

// Обработчик второго потока
void* secondThread() {
  int pid = getpid();
  int ppid = getppid();

  signal(SIGUSR1, handler);

  // Цикл работает пока не приходит прерывание
  while(!interrupt) {
```

```
66      printf("Second thread message, pid %d, ppid %d.\n", pid, ppid);
67      sleep(SECOND_DELAY);
68    }
69
70    printf("Second thread has been finished.\n");
71    return NULL;
72  }
```

Результат работы программы:

```
stakenschneider@stakenschneider:~/temp$ gcc -pthread p4.3.c -o p4.3
stakenschneider@stakenschneider:~/temp$ ./p4.3
Second thread message, pid 4749, ppid 4669.
Second thread message, pid 4749, ppid 4669.
Second thread message, pid 4749, ppid 4669.
Second thread message, pid 4749, ppid 4669.
Second thread message, pid 4749, ppid 4669.
First thread message, pid 4749, ppid 4669.
Signal arrived.
Second thread has been finished.
First thread message, pid 4749, ppid 4669.
First thread message, pid 4749, ppid 4669.
First thread message, pid 4749, ppid 4669.
( ... )
```

4. Модификация программы с добавлением функции pthread_exit()

Если в предыдущем пункте программа завершалась с помощью глобальной переменной, в этой программе используем специальную функцию завершения потока *pthread_exit*:

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

#define FIRST_DELAY 5
#define SECOND_DELAY 1
#define KILL_DELAY 5

void handler(int signal);
void* firstThread();
void* secondThread();

int main() {
  pthread_t first, second;

  // Создание и запуск первого потока
  pthread_create(&first, NULL, &firstThread, (void*) &second);

  // Создание и запуск второго потока
  pthread_create(&second, NULL, &secondThread, NULL);

  // Ожидаем завершение первого потока
  pthread_join(first, NULL);

  // Ожидаем завершение второго потока
  pthread_join(second, NULL);

  return 0x0;
}

void handler(int signal) {
  printf("Signal arrived.\n");
  printf("Second thread has been finished.\n");

  // Прерываем второй поток
```

```
37    pthread_exit(NULL);
38  }
39
40  // Обработчик первого потока
41  void* firstThread(void* secondThread) {
42    int pid = getpid();
43    int ppid = getppid();
44
45    sleep(KILL_DELAY);
46
47    // Завершаем поток с сигналом SIGUSR1
48    // Обращение к потоку происходят по указателю, перданному через аргумент функции
49    pthread_kill(*((pthread_t*)secondThread), SIGUSR1);
50
51    while(1) {
52      printf("First thread message, pid %d, ppid %d.\n", pid, ppid);
53      sleep(FIRST_DELAY);
54    }
55  }
56
57  // Обработчик второго потока
58  void* secondThread() {
59    int pid = getpid();
60    int ppid = getppid();
61
62    signal(SIGUSR1, handler);
63
64    // Цикл работает пока не приходит прерывание
65    while(1) {
66      printf("Second thread message, pid %d, ppid %d.\n", pid, ppid);
67      sleep(SECOND_DELAY);
68    }
69  }
```

Результат работы программы аналогичен предыдущему пункту:

```
1  stakenschneider@stakenschneider:~/temp$ ./p4.4
2  Second thread message, pid 4847, ppid 4669.
3  Second thread message, pid 4847, ppid 4669.
4  Second thread message, pid 4847, ppid 4669.
5  Second thread message, pid 4847, ppid 4669.
6  Second thread message, pid 4847, ppid 4669.
7  First thread message, pid 4847, ppid 4669.
8  Signal arrived.
9  Second thread has been finished.
10 First thread message, pid 4847, ppid 4669.
11 First thread message, pid 4847, ppid 4669.
12 First thread message, pid 4847, ppid 4669.
13 ( ... )
```

5.1. Перехват сигнала Ctrl+C из терминала однократно для процесса

Однократный перехватчик сигнала для процесса:

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <signal.h>
4  #include <unistd.h>
5
6  void handler(int signalCode);
7
8  int main() {
9    printf("Process started, pid %d, ppid %d.\n", getpid(), getppid());
10
11   // Устанавливаем обработчик прерывания
12   signal(SIGINT, handler);
```

```
13
14    // Выход из приложения только по внешнему прерыванию
15    while(1);
16    return 0x0;
17  }
18
19  void handler(int signalCode) {
20    printf("Signal arrived.\n");
21    exit(0x0);
22  }
```

Результат работы программы:

```
1  stakenschneider@stakenschneider:~/temp$ gcc -pthread p4.5.1.c -o p4.5.1
2  stakenschneider@stakenschneider:~/temp$ ./p4.5.1
3  Process started, pid 5556, ppid 5516.
4  ^CSignal arrived.
```

5.2. Перехват сигнала Ctrl+C из терминала однократно для потока

Однократный перехватчик сигнала для потока:

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <signal.h>
4  #include <unistd.h>
5
6  #define FIRST_DELAY 1
7
8  int interrupt = 0;
9
10 void handler(int signalCode);
11 void* firstThread();
12
13 int main() {
14   pthread_t first;
15
16   // Создание и запуск первого потока
17   pthread_create(&first, NULL, &firstThread, NULL);
18
19   // Ожидаем завершение первого потока
20   pthread_join(first, NULL);
21
22   printf("Program has been finished.\n");
23   return 0x0;
24 }
25
26 void handler(int signalCode) {
27   printf("Signal arrived.\n");
28
29   interrupt = 1;
30 }
31
32 // Обработчик первого потока
33 void* firstThread(void* secondThread) {
34   int pid = getpid();
35   int ppid = getppid();
36
37   signal(SIGINT, handler);
38
39   while(!interrupt) {
40     printf("First thread message, pid %d, ppid %d.\n", pid, ppid);
41     sleep(FIRST_DELAY);
42   }
43
44   printf("Thread has been finished.\n");
```

```
45    return NULL;
46  }
```

Результат работы программы:

```
stakenschneider@stakenschneider:~/temp$ gcc -pthread p4.5.2.c -o p4.5.2
stakenschneider@stakenschneider:~/temp$ ./p4.5.2
First thread message, pid 5652, ppid 5516.
First thread message, pid 5652, ppid 5516.
First thread message, pid 5652, ppid 5516.
^CSignal arrived.
Thread has been finished.
Program has been finished.
```

5.3. Перехват сигнала Ctrl+C из терминала многократно с восстановлением исходного обработчика

Многократный перехватчик сигнала с восстановлением исходного обработчика:

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

#define FIRST_DELAY 1
#define SIGNAL_COUNT 5

int count = 0;

void handler(int signalCode);
void* firstThread();

int main() {
  pthread_t first;

  // Создание и запуск первого потока
  pthread_create(&first, NULL, &firstThread, NULL);

  // Ожидаем завершение первого потока
  pthread_join(first, NULL);

  return 0x0;
}

void handler(int signalCode) {
  printf("Signal arrived. Counter: %d.\n", ++count);

  // Если количество прерываний больше критического значения, то воостанавливаем обработчик по
    умолчанию
  if(count >= SIGNAL_COUNT)
    signal(SIGINT, SIG_DFL);
}

// Обработчик первого потока
void* firstThread(void* secondThread) {
  int pid = getpid();
  int ppid = getppid();

  signal(SIGINT, handler);

  while(1) {
    printf("First thread message, pid %d, ppid %d.\n", pid, ppid);
    sleep(FIRST_DELAY);
  }
}
```

Результат работы программы:

```
stakenschneider@stakenschneider:~/temp$ gcc -pthread p4.5.3.c -o p4.5.3
stakenschneider@stakenschneider:~/temp$ ./p4.5.3
First thread message, pid 5292, ppid 4669.
First thread message, pid 5292, ppid 4669.
^CSignal arrived. Counter: 1.
First thread message, pid 5292, ppid 4669.
^CSignal arrived. Counter: 2.
First thread message, pid 5292, ppid 4669.
^CSignal arrived. Counter: 3.
First thread message, pid 5292, ppid 4669.
^CSignal arrived. Counter: 4.
First thread message, pid 5292, ppid 4669.
First thread message, pid 5292, ppid 4669.
^CSignal arrived. Counter: 5.
First thread message, pid 5292, ppid 4669.
^C
```

5.4. Перехват сигнала для другой комбинации клавиш (Ctrl+Z)

Многократный перехватчик сигнала с восстановлением исходного обработчика для комбинации клавиш *Ctrl+Z* (сигнал *SIGTSTP*):

```c
#include <pthread.h>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

#define FIRST_DELAY 1
#define SIGNAL_COUNT 5

int count = 0;

void handler(int signalCode);
void* firstThread();

int main() {
  pthread_t first;

  // Создание и запуск первого потока
  pthread_create(&first, NULL, &firstThread, NULL);

  // Ожидаем завершение первого потока
  pthread_join(first, NULL);

  return 0x0;
}

void handler(int signalCode) {
  printf("Signal arrived. Counter: %d.\n", ++count);

  // Если количество прерываний больше критического значения, то воостанавливаем обработчик по
    умолчанию
  if(count >= SIGNAL_COUNT)
    signal(SIGTSTP, SIG_DFL);
}

// Обработчик первого потока
void* firstThread(void* secondThread) {
  int pid = getpid();
  int ppid = getppid();

  signal(SIGTSTP, handler);

  while(1) {
    printf("First thread message, pid %d, ppid %d.\n", pid, ppid);
```

```
43      sleep(FIRST_DELAY);
44    }
45  }
```

Результат работы программы:

```
stakenschneider@stakenschneider:~/temp$ gcc -pthread p4.5.4.c -o p4.5.4
stakenschneider@stakenschneider:~/temp$ ./p4.5.4
First thread message, pid 5374, ppid 4669.
First thread message, pid 5374, ppid 4669.
First thread message, pid 5374, ppid 4669.
^ZSignal arrived. Counter: 1.
First thread message, pid 5374, ppid 4669.
^ZSignal arrived. Counter: 2.
First thread message, pid 5374, ppid 4669.
^ZSignal arrived. Counter: 3.
First thread message, pid 5374, ppid 4669.
^ZSignal arrived. Counter: 4.
First thread message, pid 5374, ppid 4669.
First thread message, pid 5374, ppid 4669.
^ZSignal arrived. Counter: 5.
First thread message, pid 5374, ppid 4669.
^Z
[1]+  Stopped                 ./p4.5.4
```

6. Перехват сигнала для другой комбинации клавиш (Ctrl+Z)

Выполним команду *kill* с ключем *-l*, которая выводит список всех сигналов для данной операционной системы:

```
stakenschneider@stakenschneider:~/temp$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Рассмотрим некоторые из них:

- *SIGABRT* - сигнал посылаемый функцией *abort()*, по умолчанию завершает процесс с дампом памяти.
- *SIGALRM* - сигнал истечения времени, заданного *alarm()*, по умолчанию завершает процесс.
- *SIGBUS* - неправильное обращение в физическую память, по умолчанию завершает процесс с дампом памяти.
- *SIGCHLD* - дочерний процесс завершен или остановлен, по умолчанию игнорируется.
- *SIGCONT* - по умолчанию продолжает выполнение ранее остановленного процесса.
- *SIGFPE* - ошибочная арифметическая операция, по умолчанию завершает процесс с дампом памяти.
- *SIGHUP* - закрытие терминала, по умолчанию завершает процесс.
- *SIGILL* - недопустимая инструкция процессора, по умолчанию завершает процесс с дампом памяти.
- *SIGINT* - сигнал прерывания *Ctrl-C* с терминала, по умолчанию завершает процесс.
- *SIGKILL* - безусловное завершение, по умолчанию завершает процесс.

- *SIGPIPE* - запись в разорванное соединение (пайп или сокет), по умолчанию завершает процесс.
- *SIGQUIT* - сигнал *Quit* с терминала *Ctrl-*, по умолчанию завершает процесс с дампом памяти.
- *SIGSEGV* - нарушение при обращении в память, по умолчанию завершает процесс с дампом памяти.
- *SIGSTOP* - остановка выполнения процесса.
- *SIGTERM* - сигнал завершения (сигнал по умолчанию для утилиты *kill*)
- *SIGTSTP* - сигнал остановки с терминала *Ctrl-Z*, по умолчанию останавливает процесс.
- *SIGTTIN* - попытка чтения с терминала фоновым процессом, по умолчанию останавливает процесс.
- *SIGTTOU* - попытка записи на терминал фоновым процессом, по умолчанию останавливает процесс.
- *SIGUSR1* - пользовательский сигнал № 1, по умолчанию завершает процесс.
- *SIGUSR2* - пользовательский сигнал № 2, по умолчанию завершает процесс.
- *SIGPOLL* - событие, отслеживаемое *poll()*, по умолчанию завершает процесс.
- *SIGPROF* - истечение таймера профилирования, по умолчанию завершает процесс.
- *SIGSYS* - неправильный системный вызов, по умолчанию завершает процесс с дампом памяти.
- *SIGTRAP* - ловушка трассировки или брейкпоинт, по умолчанию завершает процесс с дампом памяти.
- *SIGURG* - на сокете получены срочные данные, по умолчанию игнорируется.
- *SIGVTALRM* - истечение «виртуального таймера», по умолчанию завершает процесс.
- *SIGXCPU* - процесс превысил лимит процессорного времени, по умолчанию завершает процесс с дампом памяти.
- *SIGXFSZ* - процесс превысил допустимый размер файла, по умолчанию завершает процесс с дампом памяти.

Остальные сигналы (с кодами большими 32) - это сигналы реального времени. В отличие от обычных сигналов, сигналы реального времени имеют очередь, при использовании специальных функций для отправки сигнала (*sigqueue* - передача сигнала или данных процессу) могут передавать информацию (целое число или указатель), доставляются в том же порядке, в котором были отправлены.

7. Эксперимент с изменением процедуры планирования для потоков одного процесса

Разработаем программу, которая при помощи атрибутов *pthread_attr_t* задает потокам атрибуты, в которых указывается политика планирования и приоритет. В самих потоках, с помощью функции *sched_getscheduler* выводим политику планирования текущего потока.

```
#include <pthread.h>
#include <stdio.h>
#include <signal.h>
#include <sched.h>
#include <unistd.h>

#define THREAD_DELAY 1

void* threadHandler(void* threadNumber);

int main() {
  struct sched_param schedParam;
  pthread_t firstThread, secondThread, thirdThread;
  pthread_attr_t firstAttr, secondAttr, thirdAttr;
  int firstThreadNumber, secondThreadNumber, thirdThreadNumber;

  // Инициализация аттрибутов
  pthread_attr_init(&firstAttr);
  pthread_attr_init(&secondAttr);
  pthread_attr_init(&thirdAttr);
```

```
21
22    // Установка политики планирования
23    pthread_attr_setschedpolicy(&firstAttr, SCHED_FIFO);
24    pthread_attr_setschedpolicy(&secondAttr, SCHED_RR);
25    pthread_attr_setschedpolicy(&thirdAttr, SCHED_FIFO);
26
27    // Установка приоритетов планирования
28    schedParam.sched_priority = 1;
29    pthread_attr_setschedparam(&firstAttr, &schedParam);
30    schedParam.sched_priority = 1;
31    pthread_attr_setschedparam(&secondAttr, &schedParam);
32    schedParam.sched_priority = 99;
33    pthread_attr_setschedparam(&thirdAttr, &schedParam);
34
35    // Установка приоритетов для аттрибутов потоков
36    pthread_attr_setinheritsched(&firstAttr, PTHREAD_EXPLICIT_SCHED);
37    pthread_attr_setinheritsched(&secondAttr, PTHREAD_EXPLICIT_SCHED);
38    pthread_attr_setinheritsched(&thirdAttr, PTHREAD_EXPLICIT_SCHED);
39
40    // Установка приоритета наследования от родителя
41    pthread_attr_setinheritsched(&firstAttr, PTHREAD_INHERIT_SCHED);
42
43    // Создание и запуск потоков
44    firstThreadNumber = 1;
45    pthread_create(&firstThread, &firstAttr, &threadHandler, (void*) &firstThreadNumber);
46    secondThreadNumber = 2;
47    pthread_create(&secondThread, &secondAttr, &threadHandler, (void*) &secondThreadNumber)
        ;
48    thirdThreadNumber = 3;
49    pthread_create(&thirdThread, &thirdAttr, &threadHandler, (void*) &thirdThreadNumber);
50
51    // Ожидаем завершение потоков
52    pthread_join(firstThread, NULL);
53    pthread_join(secondThread, NULL);
54    pthread_join(thirdThread, NULL);
55
56    // Удаляем аттрибуты
57    pthread_attr_destroy(&firstAttr);
58    pthread_attr_destroy(&secondAttr);
59    pthread_attr_destroy(&thirdAttr);
60
61    return 0x0;
62  }
63
64  void* threadHandler(void* threadNumber) {
65    char* stringPolicy;
66
67    //pthread_attr_t currentAttr = (*((int*)threadNumber) == 1) ? firstAttr: ((*((int*)
        threadNumber) == 2) ? secondAttr: thirdAttr);
68
69    while(1) {
70      //int currentPolicy;
71      //pthread_attr_getschedpolicy(&currentAttr, &currentPolicy);
72
73      switch(sched_getscheduler(0)) {
74        case SCHED_FIFO:
75          stringPolicy = "SCHED_FIFO";
76          break;
77        case SCHED_RR:
78          stringPolicy = "SCHED_RR";
79          break;
80        case SCHED_OTHER:
81          stringPolicy = "SCHED_OTHER";
82          break;
83      }
84
```

```
85      printf("Thread number %d, thread policy %s.\n", *((int*)threadNumber), stringPolicy);
86      sleep(THREAD_DELAY);
87    }
88
89  }
```

Результат работы программы:

```
1  stakenschneider@stakenschneider:~/temp$ gcc -pthread p4.7.c -o p4.7
2  stakenschneider@stakenschneider:~/temp$ sudo ./p4.7
3  Thread number 1, thread policy SCHED_OTHER.
4  Thread number 3, thread policy SCHED_FIFO.
5  Thread number 2, thread policy SCHED_RR.
6  Thread number 2, thread policy SCHED_RR.
7  Thread number 3, thread policy SCHED_FIFO.
8  Thread number 1, thread policy SCHED_OTHER.
9  Thread number 3, thread policy SCHED_FIFO.
10 Thread number 2, thread policy SCHED_RR.
11 Thread number 1, thread policy SCHED_OTHER.
12 ^C
```

Согласно результатам, первый поток имеет политику планирования *SCHED_ OTHER*, в отличие от ожидаемой политики *SCHED_ FIFO*. это связано с тем, что первая политика была задана с опцией *PTHREAD_ INHERIT_* что означает, что политика наследуется от родительского потока.

Как и ожидалось, третий поток выполняется быстрее, чем должен. Это связано с тем, что третьему потоку задан наивысший приоритет.

Рассмотрим значения приоритет потоков с точки зрения системы:

```
1  stakenschneider@stakenschneider:~$ ps -eLfl
2  F S UID      PID  PPID LWP  C NLWP PRI  NI ADDR SZ     WCHAN STIME TTY   TIME     CMD
3  4 S root     1    0    1    0 1    80   0  -    46314  -     дек09 ?     00:00:04 /sbin/
       init splash
4  (...)
5  4 S root     3197 2358 3197 0 1    80   0  -    14004  -     00:56 pts/2 00:00:00 sudo ./
       p4.7
6  4 S root     3198 3197 3198 0 4    80   0  -    24161  -     00:56 pts/2 00:00:00 ./p4.7
7  1 S root     3198 3197 3199 0 4    80   0  -    24161  -     00:56 pts/2 00:00:00 ./p4.7
8  1 S root     3198 3197 3200 0 4    58   -  -    24161  -     00:56 pts/2 00:00:00 ./p4.7
9  1 S root     3198 3197 3201 0 4    -40  -  -    24161  -     00:56 pts/2 00:00:00 ./p4.7
10 4 R nikita   3205 2687 3205 0 1    80   0  -    9646   -     00:56 pts/6 00:00:00 ps -eLfl
```

Как и ожидалось, система регистрирует все четыре потока приложения, приоритеты которых (столбец *PRI*) отличаются. У основного потока и первого потока одинаковые приоритеты, равные 80, что объясняется наследованием приоритета от родительского потока. Третий поток имеет приоритет -40, то есть наивысший. Это объясняется заданием значения приоритета 99 в коде программы.

8. Скрипт, выполняющий лабораторную работу из исходных файлов

Был написан скрипт, выполняющий лабораторную работу из исходных файлов:

```
1  #!/bin/bash
2  sh p4.8.p1.sh
3  sh p4.8.p2.sh
4  sh p4.8.p3.sh
5  sh p4.8.p4.sh
```

Скрипт, выполняющий первую главу:

```
1  #!/bin/bash
2  echo '\n_____________________1)_POROJDENIE_I_ZAPUSK_PROCESSOV_____________________\n'
3
4  echo '\n___________p1.1___________\n'
5  gcc p1.1.c -o p1.1
6  ./p1.1
7  rm p1.1
8  sleep 1
9
```

```
echo '\n__________p1.2__________\n'
gcc p1.2.c -o p1.2
./p1.2
rm p1.2
sleep 1

echo '\n__________p1.3__________\n'
g++ p1.3.c singlecore.c -o p1.3
./p1.3
rm p1.3
sleep 1

echo '\n__________p1.4.fifo__________\n'
g++ -std=c++11 p1.4.cpp singlecore.c -o p1.4
./p1.4 0 64 40 30 20
sleep 1

echo '\n__________p1.4.rr__________\n'
./p1.4 1 74 60 30 20
sleep 1

echo '\n__________p1.4.other__________\n'
./p1.4 2 0 0 0 0
rm p1.4
sleep 1

echo '\n__________p1.5_p1.6_p1.7__________\n'
gcc child.c -o child
gcc parent.c -o parent
./parent
rm child
rm parent
sleep 1

echo '\n__________p1.8__________\n'
g++ -std=c++11 p1.8.cpp -o p1.8
./p1.8 0
rm p1.8
sleep 1

echo '\n__________p1.9.1__________\n'
gcc p1.9.1.c -o p1.9.1
./p1.9.1
rm p1.9.1
sleep 1

echo '\n__________p1.9.2.c1_p1.9.2.c2_p1.9.2.c3__________\n'
gcc p1.9.2.c1.c -o p1.9.2.c1
gcc p1.9.2.c2.c -o p1.9.2.c2
gcc p1.9.2.c3.c -o p1.9.2.c3
./p1.9.2.c1
./p1.9.2.c2
./p1.9.2.c3
sleep 1

echo '\n__________p1.9.2.p__________\n'
g++ -std=c++11 p1.9.2.p.cpp -o p1.9.2.p
./p1.9.2.p 0
rm p1.9.2.c1
rm p1.9.2.c2
rm p1.9.2.c3
rm p1.9.2.p
sleep 1

echo '\n__________p1.10.1.fifo__________\n'
g++ -std=c++11 p1.10.1.cpp singlecore.c -o p1.10.1
```

```
./p1.10.1 0 64 40 30 20
sleep 1

echo '\n__________p1.10.1.rr__________\n'
./p1.10.1 1 74 60 30 20
sleep 1

echo '\n__________p1.10.1.other__________\n'
./p1.10.1 2 0 0 0 0
rm p1.10.1
sleep 1

echo '\n__________p1.10.2__________\n'
g++ -std=c++11 p1.10.2.cpp singlecore.c -o p1.10.2
./p1.10.2 0 1 0 1
rm p1.10.2
sleep 1

echo '\n__________p1.11__________\n'
g++ -std=c++11 p1.11.cpp singlecore.c -o p1.11
./p1.11
rm p1.11
sleep 1

echo '\n__________p1.11.1__________\n'
g++ -std=c++11 p1.11.1.cpp singlecore.c -o p1.11.1
./p1.11.1
rm p1.11.1
sleep 1

echo '\n__________p1.12__________\n'
g++ p1.12.c singlecore.c -o p1.12
g++ -std=c++11 p1.12.ch.cpp -o p1.12.ch
./p1.12
rm p1.12
rm p1.12.ch
sleep 1
```

Скрипт, выполняющий вторую главу:

```
#!/bin/bash
echo '\n_____________________2)_VZAIMODEYSTVIYE_RODSTVENNIH_PROCESSOV_____________________\n'

echo '\n__________p2.1.1__________\n'
gcc p2.1.1.c -o p2.1.1
gcc p2.1.1.ch.c -o p2.1.1.ch
./p2.1.1
sleep 1

echo '\n__________p2.1.2__________\n'
gcc p2.1.2.c -o p2.1.2
./p2.1.2
sleep 3

echo '\n__________p2.1.3__________\n'
gcc p2.1.3.c -o p2.1.3
./p2.1.3
sleep 1

echo '\n__________p2.2__________\n'
sh p2.2.sh
rm p2.1.1
rm p2.1.1.ch
rm p2.1.2
rm p2.1.3
sleep 1
```

Скрипт, выполняющий третью главу:

```
#!/bin/bash
echo '\n_____________________3)
    _UPRAVLENIE_PROCESSAMI_POSREDSTVOM_SIGNALOV_____________________\n'

echo '\n___________p3.1.k___________\n'
kill -l
sleep 1

: <<'p3.1.1'
echo '\n___________p3.1.1___________\n'
gcc p3.1.p.c -o p3.1.0
gcc p3.1.1.ch.c -o p3.1.1.ch
gcc p3.1.2.ch.c -o p3.1.2.ch
gcc p3.1.3.ch.c -o p3.1.3.ch
./p3.1.0
sleep 4
rm p3.1.0
rm p3.1.1.ch
rm p3.1.2.ch
rm p3.1.3.ch
sleep 1
p3.1.1

echo '\n___________p3.3___________\n'
gcc p3.3.c -o p3.3
cat p3.3.txt & ./p3.3 & sh p3.3.sh &
rm p3.3
sleep 1

echo '\n___________p3.4.1.1___________\n'
gcc p3.4.1.c -o p3.4.1
nice -5 ./p3.4.1 & nice -5 ./p3.4.1
sleep 1

echo '\n___________p3.4.1.2___________\n'
nice -15 ./p3.4.1 & nice -1 ./p3.4.1
rm p3.4.1
sleep 1

echo '\n___________p3.4.2___________\n'
g++ -std=c++11 p3.4.2.cpp -o p3.4.2
./p3.4.2
rm p3.4.2

echo '\n___________p3.6___________\n'
whoami
id -u nikita
id -u root
ps -Al
```

Скрипт, выполняющий четвертую главу:

```
#!/bin/bash
echo '\n_____________________4)_MNOGONITEVOE_FUNKCIONIROVANIE_____________________\n'

echo '\n___________p4.1___________\n'
gcc -pthread p4.1.c -o p4.1
timeout 6s ./p4.1
rm p4.1
sleep 1

echo '\n___________p4.2___________\n'
gcc -pthread p4.2.c -o p4.2
timeout 6s ./p4.2
rm p4.2
```

```
sleep 1

echo '\n__________p4.3__________\n'
gcc -pthread p4.3.c -o p4.3
timeout 6s ./p4.3
rm p4.3
sleep 1

echo '\n__________p4.4__________\n'
gcc -pthread p4.4.c -o p4.4
timeout 6s ./p4.4
rm p4.4
sleep 1

echo '\n__________p4.6__________\n'
kill -l

echo '\n__________p4.7__________\n'
gcc -pthread p4.7.c -o p4.7
timeout 6s ./p4.7
rm p4.7
sleep 1
```

Скрипт, выполняющий всю лабораторную работу, был успешно запущен на лабораторном компьютере. Ввиду отсутствия *C++ 11* на лабораторных компьютерах, некоторые программы были отредактированы, для использования более старой версией компилятора. Также из-за отсутствия прав суперпользователя, эксперименты с планированием не работают. Все остальные эксперименты успешно выполнены.

1.4 Вывод

В ходе работы были изучены методы распараллеливания процессов, изменение приоритетов выполнения, программную обработку сигналов.

Процессы можно создавать с помощью функции *fork*, копирующей процесс со всеми структурами, возвращающей идентификатор процесса для процесса, откуда она была вызвана, и ноль для скопированного процесса. Это позволяет организовать псевдораспараллеливание процессов.

При помощи функций семейства *exec* можно заменить образ текущего процесса на новый образ процесса из файла, задать параметры вызываемого процесса и новые переменные окружения разными форматами: строкой или вектором.

Изменяя политику планирования, можно управлять принципами построения очереди. Изменяя приоритет процесса, можно изменить порядок выполнения и дать преимущество в борьбе за процессорный ресурс.

Кроме того, в существует набор сигналов, которые позволяют процессам обмениваться информацией между собой и позволяют системе уведомлять процессы о различных событиях в системе, будь то нажатие клавиш, окончание пользовательской сессии или множество других событий. Для каждого процесса можно установить реакцию на получение сигнала: действие по умолчанию, игнорирование и вызов определенной функции. Необдуманное использование сигналов может привести к сбою.

Альтернативой использования *fork* и *exec* для распараллеливания вычислений является многопоточное программирование. Потоки могут создаваться и отслеживаться в основном процессе, им могут быть переданы параметры, назначены обработчики сигналов, а так же изменена процедура планирования и приоритеты.