Санкт-Петербургский Политехнический Университет Петра Великого

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

Операционные системы

Отчет по лабораторной работе №3

Процессы UNIX

Работу выполнил:

Иванов А.А.

Группа: 43501/1

Преподаватель:

Малышев И..

Санкт-Петербург

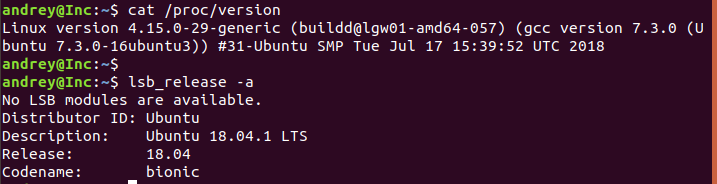
2018

# 1. Программа работы

1. Порождение и запуск процессов
   1. Создайте программу на основе одного исходного (а затем исполняемого) файла с псевдо распараллеливанием вычислений посредством порождения процесса-потомка
      1. Выполните сначала однократные вычисления в каждом процессе, обратить внимание, какой процесс на каком этапе владеет процессорным ресурсом. Каждый процесс должен иметь вывод на терминал, идентифицирующий текущий процесс. Последняя исполняемая команда функции main должна вывести на терминал сообщение о завершении программы. Объяснить результаты. Сделайте выводы об использовании адресного пространства.
      2. Затем однократные вычисления заменить на циклы, длительность исполнения которых достаточна для наблюдения конкуренции процессов за процессорный ресурс.
      3. Изменить процедуру планирования и повторить эксперимент.
   2. Разработать программы родителя и потомка father.c и son.c. Для фиксации состояния таблицы процессов в файле целесообразно использовать системный вызов system ”ps -abcde > file”
      1. Запустите на выполнение программу father. С терминала узнайте о процессах, запущенных с вашего терминала.
      2. Выполните программу father в фоновом режиме (father &). Получите таблицу процессов, запущенных с вашего терминала (включая отцовский и сыновий процессы)
   3. Выполнить создание процессов с использованием различных функций семейства exec() с разными параметрами функций семейства, привести результаты эксперимента.
   4. Проанализируйте значение, возвращаемое функцией wait(&status). Предложите эксперимент, позволяющий родителю отслеживать подмножество по- рожденных потомков, используя различные функции семейства wait().
   5. Проанализируйте очередность исполнения процессов.
      1. Очередность процессов, порожденных вложенными вызовами fork().
      2. Измените процедуру планирования с помощью функции с шаблоном scheduler в ее названии и повторите эксперимент.
      3. Поменяйте порядок очереди в RR-процедуре.
      4. Можно ли задать разные процедуры планирования разным процессам с одинаковыми приоритетами. Как они будут конкурировать, подтвердите экспериментально.
   6. Определите величину кванта. Можно ли ее поменять? Для обоснования проведите эксперимент.
   7. 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.
2. Взаимодействие родственных процессов
   1. Изменяя длительности выполнения процессов и параметры системных вызовов, рассмотреть 3 ситуации и вывести соответствующие таблицы процессов:
      1. процесс-отец запускает процесс-сын и ожидает его завершения
      2. процесс-отец запускает процесс-сын и, не ожидая его завершения, завершает свое выполнение. Зафиксируйте изменение родительского идентификатора процесса-сына
      3. процесс-отец запускает процесс-сын и не ожидает его завершения; процесс-сын завершает свое выполнение. Зафиксируйте появление процесса-зомби, для этого включите команду ps в программу father.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.
3. Управление процессами посредством сигналов
   1. С помощью команды kill -l ознакомьтесь с перечнем сигналов, поддерживаемых процессами. Ознакомьтесь с системными вызовами kill(2), signal(2).
   2. Подготовьте программы следующего содержания:
      1. процесс father порождает процессы son1, son2, son3 и запускает на исполнение программные коды из соответствующих исполнительных файлов;
      2. далее родительский процесс осуществляет управление потомками, для этого он генерирует сигнал каждому пользовательскому процессу;
      3. в пользовательских процессах-потомках необходимо обеспечить: для son1 - реакцию на сигнал по умолчанию; для son2 - реакцию игнорирования; для son3 - перехватывание и обработку сигнала.
   3. 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.
   4. 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. Позаботьтесь об уведомлении о завершении одного из заданий с помощью команды notify. Аргументом ко- манды является номер задания. Верните невыполненные задания в приоритетный режим командой fg. Например: fg Отмените одно из невыполненных заданий.
   5. 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). Есть ли разница в приоритетах для системных и пользовательских процессов? Используются ди-приоритеты реального времени? Каков пользовательский приоритет для запуска приложений из shell? Все ответы подкрепляйте экспериментально
   6. Ознакомьтесь с командой nohup(1). Запустите длительный процесс по nohup(1). Завершите сеанс работы. Снова войдите в систему и проверьте таблицу процессов. Поясните результат.
   7. 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.
4. Многонитевое функционирование
   1. Подготовьте программу, формирующую несколько нитей. Каждая нить в цикле: выводит на печать собственное имя и инкрементирует переменную времени, после чего "засыпает"(sleep(5); sleep(1); - для первой и второй нитей соответственно).
   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. Приведите и объясните результат.
   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, SIGUSR); (t2 - дескриптор второй нити). В остальном программу можно не изменять. Проанализируйте полученные результаты.
   4. Последняя модификация предполагает создание собственного обработчика сигнала, содержащего уведомление о начале его работы и возврат посредством функции pthread\_exit(NULL)
   5. Перехватите сигнал «CTRL C»
      1. Перехватите сигнал «CTRL C» для процесса(однократно)
      2. Перехватите сигнал «CTRL C» для потока(однократно)
      3. Перехватите сигнал «CTRL C» многократно с восстановлением исходного обработчика после нескольких раз срабатывания.
      4. 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.
   7. Проанализируйте процедуру планирования для процессов и потоков одного процесса.
      1. Обоснуйте результат экспериментально.
      2. Попробуйте изменить процедуру планирования. Подтвердите экспериментально, если изменение возможно
      3. Задайте нитям разные приоритеты программно и извне (объясните результат).
   8. Создайте командный файл (скрипт), выполняющий Вашу лабораторную работу автоматически при наличии необходимых С-файлов.

# 2. Ход работы

Узнаем систему и версию ядра, а также информацию о дистрибутиве:



## 2.1. Порождение и запуск процессов

Ознакомимся с выполнением системных вызовов fork(); execl(); wait(); exit(); sleep().

**fork**:используется для создания нового процесса, дублирующего вызывающий. Вызов fork является основным примитивом создания процессов.

**exec**:семейство библиотечных процедур и один системный вызов, выполняющих одну и ту же функцию - смену задачи процесса, за счет перезаписи его пространства записи памяти новой программой. Различие между вызовами exec в основном лежит в способе задания списка их аргументов.

**wait**:этот вызов обеспечивает синхронизацию процессов. Он позволяет процессу ожидать завершения другого процесса, обычно логически связанного с ним (скажем, глобальной переменной, к коей оба процесса имеют доступ). *По сути – явно вызывается при использовании семафоров (ф-ция sem\_wait()).*

* pid\_t wait (int \*status\_location) принуждает родительский процесс ждать завершения работы процесса потомка. wait() возвращает уникальный идентификационный номер процесса-потомка или -1 в случае ошибки. Код возврата потомка сохраняется в status\_location.
* pid\_t waitpid (pid\_t pid, int \*status\_location, int options) принуждает родительский процесс ждать завершения работы процесса потомка, обладающего идентификационным номером pid. waitpid() возвращает уникальный идентификационный номер процесса-потомка или -1 в случае ошибки. Код возврата потомка сохраняется в status\_location.

**exit**: прекращает процесс, из которого эта функция была вызвана. Кодом возврата этого процесса становиться значение *status (0 – нормальное завершение)*

**execl**: в рамках процесса выполняет исполняемый файл, а затем прекращает процесс.

**sleep**: вызов задержки при выполнении программы на указанное время.

Все вышеперечисленные ф-ции используются далее в лабораторной, поэтому здесь не приведены примеры их применения.

### 2.1.1. Псевдо распараллеливание вычислений посредством порождения процесса-потомка

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

#### 2.1.1.1. Однократные вычисления в каждом процессе

Выполните сначала однократные вычисления в каждом процессе, обратить внимание, какой процесс на каком этапе владеет процессорным ресурсом. Каждый процесс должен иметь вывод на терминал, идентифицирующий текущий процесс. Последняя исполняемая команда функции main должна вывести на терминал сообщение о завершении программы. Объяснить результаты. Сделайте выводы об использовании адресного пространства.

При вызове fork() порождается новый процесс (процесс-потомок), который почти идентичен порождающему процессу-родителю. Весь код после fork() выполняется дважды, как в процессе-потомке, так и в процессе-родителе. Между процессом-потомком и процессом-родителем существуют различия:

* PID процесса-потомка отличен от PID процесса-родителя
* значению PPID процесса-потомка присваивается значение PID процесса-родителя
* процесс-потомок получает собственную таблицу файловых дескрипторов, являющуюся копией таблицы процесса-родителя на момент вызова fork(). Это означает, что открытые файлы наследуются, но если процесс-потомок, например, закроет какой-либо файл, то это не повлияет на таблицу дескрипторов процесса-родителя
* для процесса-потомка очищаются все ожидающие доставки сигналы
* временная статистика выполнения процесса-потомка в таблицах ОС обнуляется
* блокировки памяти и записи, установленные в процессе-родителе, не наследуются

Ниже приведен листинг программы ch1\_p1.cpp:

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  int main(void){  if(fork()==0) // порождение процесса потомка  printf("Son process,pid=%d, ppid=%d\n",getpid(),getppid());  else // ветка для процесса родителя  printf("Father process,pid=%d, ppid=%d\n",getpid(),getppid());  printf("Process finished, pid=%d\n",getpid());  } |

Выполнение:

|  |
| --- |
| ana@AnaLinux ~/lab3 $ g++ -o ch1\_p1 ch1\_p1.cpp  ana@AnaLinux ~/lab3 $ ./ch1\_p1  Father process,pid=32517, ppid=14497  Process finished, pid=32517  ana@AnaLinux ~/lab3 $ Son process,pid=32518, ppid=1  Process finished, pid=32518 |

Сперва был выполнен родительский процесс, затем процесс потомок. Ppid процесса-потомка был равен pid процесса-родителя.

Возвращаемые fork значения – 0, другое – ненулевое значение. Процесс, который получает 0, называется порожденным процессом (потомок), а ненулевое значение достается исходному процессу, который является родительским процессом. Возвращаемые значения используются для того, чтобы определить, где какой процесс. Поскольку оба процесса возобновляют выполнение в одной и той же области, единственный возможный определитель процесса - это возвращаемые значения fork.

В программе ch1\_p2.cpp имеется общая переменная counter равная 0 перед началом работы процессов. Данная переменная однократно инкрементируется в процессе-родителе и однократно декрементируется в процессе-потомке. По логике должен получится 0 в конечном итоге.

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  int main(void){  int counter=0; // переменная для изменения  if(fork()==0){ // в процессе-потомке декрементируем  counter--;  printf("Son process,pid=%d, ppid=%d, counter=%d\n",getpid(),getppid(),counter);  }else{ // в процессе-родителе инкрементируем  counter++;  printf("Father process,pid=%d, ppid=%d, counter=%d\n",getpid(),getppid(),counter);  }  printf("Process finished, pid=%d, counter=%d\n",getpid(), counter);  } |

Результат выполнения:

|  |
| --- |
| ana@AnaLinux ~/lab3 $ g++ -o ch1\_p2 ch1\_p2.cpp  ana@AnaLinux ~/lab3 $ ./ ch1\_p2  Father process,pid=16299, ppid=1950, counter=1  Process finished, pid=16299, counter=1  ana@AnaLinux ~/lab3 $ Son process,pid=16300, ppid=1, counter=-1  Process finished, pid=16300, counter=-1 |

Процесс-родитель успешно инкрементировал переменную, и она стала равна 1, но после декремента процессом-потомком её итоговое значение стало равно -1. Это произошло из-за того, что процесс-потомок имеет собственную память, поэтому изменение в памяти процесса-родителя никак не повлияло на содержимое в памяти процесса-потомка.

#### 2.1.1.2. Циклы в каждом процессе

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

В программе ch1\_p3.cpp в каждом из процессов присутствует цикл, который каждые 1000000 итераций выводит сообщение в консоль.

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  #include "onecore.h"  int main(void){  int pid;  if(fork()==0){  pid=getpid(); // получение pid процесса  one\_core(pid); // установка работы на одно ядро  printf("Son process,pid=%d, ppid=%d\n",getpid(),getppid());  for(int i=0;i<10000000;i++) // цикл для того чтобы показать  if(i%1000000==0) // конкуренцию за процессорное  printf("son\n"); // время  }else{  pid=getpid(); // получение pid процесса  one\_core(pid); // установка работы на одно ядро  printf("Father process,pid=%d, ppid=%d\n",getpid(),getppid());  for(int i=0;i<10000000;i++) // цикл для того чтобы показать  if(i%1000000==0) // конкуренцию за процессорное  printf("father\n"); // время  }  printf("Process finished, pid=%d\n",getpid());  } |

|  |
| --- |
| ana@AnaLinux ~/lab3 $ ./ ch1\_p3  Father process,pid=19099, ppid=1950  father  Son process,pid=19100, ppid=19099  son  son  father  father  father  son  son  son  father  father  father  son  son  son  father  father  father  son  son  Process finished, pid=19100  Process finished, pid=19099 |

Как видно по результату выполнения, процессы конкурируют между собой за процессорное время, в среднем процессорное время было распределено поровну.

#### 2.1.1.3. Смена процедуры планирования

Изменить процедуру планирования и повторить эксперимент.

Для того чтобы произвести процедуру планирования процессорного времени между параллельными процессами, им необходимо присвоить приоритет sched\_priority. Значение sched\_priority может находиться в интервале от 0 до 99. Для того, чтобы определить, какой процесс будет работать следующим, диспетчер Linux ищет непустой список с наибольшим статическим приоритетом и запускает первый процесс из этого списка. Алгоритм планирования определяет, как процесс будет добавлен в список с тем же статическим приоритетом и как он будет перемещаться внутри этого списка

Существует следующие алгоритмы: SCHED\_OTHER, SCHED\_FIFO и SCHED\_RR.

* SCHED\_OTHER - это 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.

Статический приоритет процессов с алгоритмом SCHED\_OTHER равен нулю, а статические приоритеты процессов с алгоритмами SCHED\_FIFO и SCHED\_RR могут находиться в диапазоне от 1 до 99. Статический приоритет, больший, чем 0, может быть установлен только у суперпользовательских процессов.

Программа ch1\_p4.cpp:

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  #include <sched.h>  #include <stdlib.h>  #include "onecore.h"  using namespace std;  int main(int argc, char \*\*argv) {  struct sched\_param shdprm; // значение параметров планирования  int pid1, pid2, pid3, pid4; // pid различных процессов  int ppid1, ppid2, ppid3, ppid4; // ppid различных процессов  int p0,p1,p2,p3; // переменные для задания значения приоритетов  int policy; // переменная для значения алгоритма планирования  if(atoi(argv[1])==1) // в зависимости от первого входного  policy=SCHED\_FIFO; // аргумента, устанавливаем алгоритм  else // планировани  policy=SCHED\_RR;    p0 = atoi(argv[2]); // считывание коэффициентов для их  p1 = atoi(argv[3]); // последующей установки процессам  p2 = atoi(argv[4]);  p3 = atoi(argv[5]);    pid1=getpid(); // получение pid  ppid1=getppid(); // получение ppid  one\_core(pid1); // установка работы на одно ядра  printf("pid=%d\n",pid1); // родительский pid  printf("ppid=%d\n",ppid1); // родительский ppid    shdprm.sched\_priority=p0; // установка приоритета  if(sched\_setscheduler(0,policy,&shdprm) == -1) // если ошибка  printf("ERROR\n");  if((fork()) == 0){ // создание 1 процесса  pid2=getpid();  ppid2=getppid();  one\_core(pid2); // установка работы на одно ядра  shdprm.sched\_priority = p1; // установка приоритета  if(sched\_setscheduler(getpid(),policy,&shdprm) == -1)  printf("ERROR\n");  printf("pid=%d, ppid=%d, proc=1 stop\n", pid2, ppid2);  }  if((fork()) == 0){ // создание 2 процесса  pid3=getpid();  ppid3=getppid();  one\_core(pid3); // установка работы на одно ядра  shdprm.sched\_priority = p2; // установка приоритета  if(sched\_setscheduler(getpid(), policy, &shdprm) == -1)  printf("ERROR\n");  printf("pid=%d, ppid=%d, proc=2 stop\n", pid3, ppid3);  }  if((fork()) == 0){ // создание 3 процесса  pid4=getpid();  ppid4=getppid();  one\_core(pid4); // установка работы на одно ядра  shdprm.sched\_priority = p3; // установка приоритета  if(sched\_setscheduler(getpid(), policy, &shdprm) == -1)  printf("ERROR\n");  printf("pid=%d, ppid=%d, proc=3 stop\n", pid4, ppid4);  }  sleep(1);  } |

Переменные p0, p1, p2, p3 отвечают за приоритеты выполнения процессов. Для запуска данной программы, в качестве входных параметров требуется ввести 5 переменных. Первая для определения алгоритма, 1 = SCHED\_FIFO иначе SCHED\_RR. Оставшиеся 4 переменные отвечают за приоритеты выполнения.

Для наглядности планирования процессов будем изменять приоритеты, а также алгоритм планирования. Выполнение программы ch1\_p4.cpp без прав суперпользователя приведет к ошибкам.

**Алгоритм SCHED\_FIFO**

Пример 1: выполнения ch1\_p4.cpp при p0=99, p1=80, p2=50, p3=30

|  |
| --- |
| ana@AnaLinux ~/lab3 $ sudo ./ch1\_p4 1 99 80 50 30  pid=17880  ppid=17879  pid=17881, ppid=17880, proc=1 stop  pid=17884, ppid=17881, proc=2 stop  pid=17882, ppid=17880, proc=2 stop  pid=17887, ppid=17882, proc=3 stop  pid=17886, ppid=17884, proc=3 stop  pid=17885, ppid=17881, proc=3 stop  pid=17883, ppid=17880, proc=3 stop |

Пример 2: выполнения ch1\_p4.cpp при p0=99, p1=50, p2=80, p3=99

|  |
| --- |
| ana@AnaLinux ~/lab3 $ sudo ./ch1\_p4 1 99 50 80 99  pid=17889  ppid=17888  pid=17892, ppid=17889, proc=3 stop  pid=17891, ppid=17889, proc=2 stop  pid=17893, ppid=17891, proc=3 stop  pid=17890, ppid=17889, proc=1 stop  pid=17894, ppid=17890, proc=2 stop  pid=17896, ppid=17894, proc=3 stop  pid=17895, ppid=17890, proc=3 stop |

Пример 3: выполнения ch1\_p4.cpp при p0=99, p1=50, p2=50, p3=50

|  |
| --- |
| ana@AnaLinux ~/lab3 $ sudo ./ch1\_p4 1 99 50 50 50  pid=17899  ppid=17898  pid=17902, ppid=17899, proc=3 stop  pid=17901, ppid=17899, proc=2 stop  pid=17900, ppid=17899, proc=1 stop  pid=17903, ppid=17901, proc=3 stop  pid=17904, ppid=17900, proc=2 stop  pid=17905, ppid=17900, proc=3 stop  pid=17906, ppid=17904, proc=3 stop |

Как только появляется процесс с более высоки приоритетом, он начинает выполняться, а остальные ждут его завершения. После завершения этого процесса, начинает выполнятся менее приоритетный процесс. Так в примере №2 первым выполнился процесс 3, после него процесс 2 и затем снова 3. Тогда как в примере №1, процесс 3 имея низкий приоритет выполнился позднее других процессов.

**Алгоритм SCHED\_RR**

Пример 1: выполнения ch1\_p4.cpp при p0=99, p1=80, p2=50, p3=30

|  |
| --- |
| ana@AnaLinux ~/lab3 $ sudo ./ch1\_p4 2 99 80 50 30  pid=17908  ppid=17907  pid=17909, ppid=17908, proc=1 stop  pid=17912, ppid=17909, proc=2 stop  pid=17910, ppid=17908, proc=2 stop  pid=17915, ppid=17910, proc=3 stop  pid=17914, ppid=17912, proc=3 stop  pid=17913, ppid=17909, proc=3 stop  pid=17911, ppid=17908, proc=3 stop |

Пример 2: выполнения ch1\_p4.cpp при p0=99, p1=50, p2=80, p3=99

|  |
| --- |
| ana@AnaLinux ~/lab3 $ sudo ./ch1\_p4 2 99 50 80 99  pid=17917  ppid=17916  pid=17920, ppid=17917, proc=3 stop  pid=17919, ppid=17917, proc=2 stop  pid=17921, ppid=17919, proc=3 stop  pid=17918, ppid=17917, proc=1 stop  pid=17922, ppid=17918, proc=2 stop  pid=17924, ppid=17922, proc=3 stop  pid=17923, ppid=17918, proc=3 stop |

Пример 3: выполнения ch1\_p4.cpp при p0=99, p1=50, p2=50, p3=50

|  |
| --- |
| ana@AnaLinux ~/lab3 $ sudo ./ch1\_p4 2 99 50 50 50  pid=17926  ppid=17925  pid=17929, ppid=17926, proc=3 stop  pid=17928, ppid=17926, proc=2 stop  pid=17927, ppid=17926, proc=1 stop  pid=17930, ppid=17928, proc=3 stop  pid=17931, ppid=17927, proc=2 stop  pid=17932, ppid=17927, proc=3 stop  pid=17933, ppid=17931, proc=3 stop |

Результаты SCHED\_RR оказались аналогичны результатам SCHED\_FIFO.

**Алгоритм SCHED\_OTHER**

SCHED\_OTHER - это используемый по умолчанию алгоритм со стандартным разделением времени, с которым работает большинство процессов. В отличии от SCHED\_FIFO и SCHED\_RR у данной процедуры статический приоритет процесса равен 0, что означает процессы имеют одинаковые приоритеты.

ch1\_p4\_3.cpp:

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  #include <sched.h>  #include <stdlib.h>  #include "onecore.h"  using namespace std;  int main(int argc, char \*\*argv) {  struct sched\_param shdprm; // значение параметров планирования  int pid1, pid2, pid3, pid4; // pid различных процессов  int ppid1, ppid2, ppid3, ppid4; // ppid различных процессов  int policy; // переменная для значения алгоритма планирования  policy=SCHED\_OTHER; // устанавливаем алгоритм планировани    pid1=getpid(); // получение pid  ppid1=getppid(); // получение ppid  one\_core(pid1); // установка работы на одно ядра  printf("pid=%d\n",pid1); // родительский pid  printf("ppid=%d\n",ppid1); // родительский ppid    shdprm.sched\_priority=0; // установка приоритета  if(sched\_setscheduler(0,policy,&shdprm) == -1) // если ошибка  printf("ERROR\n");  if((fork()) == 0){ // создание 1 процесса  pid2=getpid();  ppid2=getppid();  one\_core(pid2); // установка работы на одно ядра  if(sched\_setscheduler(getpid(),policy,&shdprm) == -1)  printf("ERROR\n");  printf("pid=%d, ppid=%d, proc=1 stop\n", pid2, ppid2);  }  if((fork()) == 0){ // создание 2 процесса  pid3=getpid();  ppid3=getppid();  one\_core(pid3); // установка работы на одно ядра  if(sched\_setscheduler(getpid(), policy, &shdprm) == -1)  printf("ERROR\n");  printf("pid=%d, ppid=%d, proc=2 stop\n", pid3, ppid3);  }  if((fork()) == 0){ // создание 3 процесса  pid4=getpid();  ppid4=getppid();  one\_core(pid4); // установка работы на одно ядра  if(sched\_setscheduler(getpid(), policy, &shdprm) == -1)  printf("ERROR\n");  printf("pid=%d, ppid=%d, proc=3 stop\n", pid4, ppid4);  }  sleep(1);  } |

Результат выполнения:

|  |
| --- |
| ana@AnaLinux ~/lab3 $ ./ch1\_p4\_3  pid=19875  ppid=1950  pid=19878, ppid=19875, proc=3 stop  pid=19877, ppid=19875, proc=2 stop  pid=19876, ppid=19875, proc=1 stop  pid=19881, ppid=19876, proc=3 stop  pid=19879, ppid=19877, proc=3 stop  pid=19880, ppid=19876, proc=2 stop  pid=19882, ppid=19880, proc=3 stop |

Как видно из результатов, процессы, имея равные приоритеты выполнения, выполнялись, как и ожидалось. Сперва были выполнены процессы 3, 2, 1 которых породил родительский процесс, во время их выполнения процесс 2 породил процесс 3, который выполняется после этих трех процессов и т.д. их связь можно проследить по pid и ppid.

### 2.1.2. Программы родителя и потомка

Разработать программы родителя и потомка father.c и son.c. Для фиксации состояния таблицы процессов в файле целесообразно использовать системный вызов system ”ps -abcde > file”.

Были написаны программы родителя и потомка father.c и son.c.

Программа father.c:

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  #include <sys/wait.h>  #include <stdlib.h>  int main(void){  int status;  printf("Father params: pid=%d, ppid=%d\n",getpid(),getppid());  if(fork()==0)  execl("son", "son", NULL);  else{  //system("ps xf");  printf("Son pid=%d\n",wait(&status));  }  return 0;  } |

Программ son.c:

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  int main(void){  printf("Son params: pid=%d, ppid=%d\n",getpid(),getppid());  sleep(1);  return 0;  } |

Команда execl позволяет выполнять выполнять исполняемые файлы. Sleep - приостанавливает на некоторое время процесс. Wait принуждает родительский процесс ожидать завершения работы процесса-потомка. Wait возвращает pid процесса-потомка или -1 в случае ошибки. Код возврата потомка сохраняется в переменной переданной в данную функцию.

#### 2.1.2.1. Выполнение программы father

Запустите на выполнение программу father. С терминала узнайте о процессах, запущенных с вашего терминала.

В программу father была добавлена строчка (system("ps -xf");) для вызова списка текущих процессов

|  |
| --- |
| ana@AnaLinux ~/lab3 $ g++ -o father father.c  ana@AnaLinux ~/lab3 $ g++ -o son son.c  ana@AnaLinux ~/lab3 $ ./father  Father params: pid=17970, ppid=1950  Son params: pid=17971, ppid=17970  Son pid=17971  PID TTY STAT TIME COMMAND  1950 pts/0 Ss 0:00 \\_ bash  17970 pts/0 S+ 0:00 \\_ ./father  17972 pts/0 S+ 0:00 \\_ sh -c ps -xf  17973 pts/0 R+ 0:00 \\_ ps -xf |

После запуска программы, были выведены pid и ppid процесса-родителя и потомка, к которых ppid потомка равен pid родителя. Далее было отражено дерево процессов (обрезано для лучшей читаемости), в котором для процесса son родителем является процесс father, а для процесса father - bash.

#### 2.1.2.2. Выполнение программы father в фоновом режиме

Выполните программу father в фоновом режиме (father &). Получите таблицу процессов, запущенных с вашего терминала (включая отцовский и сыновий процессы)

Запустим программу father в фоновом режиме:

|  |
| --- |
| ana@AnaLinux ~/lab3 $ ./father &  [1] 17979  Father params: pid=17979, ppid=1950  Son params: pid=17980, ppid=17979  Son pid=17980  PID TTY STAT TIME COMMAND  1387 ? Ssl 0:00 cinnamon-session --session cinnamon  1510 ? Ss 0:00 \\_ /usr/bin/ssh-agent /usr/bin/dbus-launch --exit-with-session /usr/bin/im-laun  1560 ? Sl 0:07 \\_ /usr/lib/i386-linux-gnu/cinnamon-settings-daemon/cinnamon-settings-daemon  1635 ? S 0:00 \\_ /usr/bin/python3 /usr/bin/cinnamon-launcher  1652 ? Sl 15:37 | \\_ cinnamon --replace  1660 ? Sl 0:00 \\_ /usr/lib/policykit-1-gnome/polkit-gnome-authentication-agent-1  1662 ? Sl 0:18 \\_ nemo -n  1665 ? Sl 0:00 \\_ nm-applet  1666 ? Sl 0:00 \\_ /usr/bin/python3 /usr/bin/cinnamon-killer-daemon  1842 ? Sl 1:04 \\_ cinnamon-screensaver  1966 ? S 0:00 \\_ /usr/bin/python3 /usr/bin/mintupdate-launcher  1969 ? S 0:00 | \\_ sh -c /usr/lib/linuxmint/mintUpdate/mintUpdate.py  1970 ? Sl 0:00 | \\_ /usr/bin/python3 /usr/lib/linuxmint/mintUpdate/mintUpdate.py  1996 ? Sl 0:00 \\_ /usr/bin/python3 /usr/share/system-config-printer/applet.py  1946 ? Sl 0:03 /usr/lib/gnome-terminal/gnome-terminal-server  1950 pts/0 Ss+ 0:00 \\_ bash  17979 pts/0 S 0:00 \\_ ./father  17981 pts/0 S 0:00 \\_ sh -c ps -xf  17982 pts/0 R 0:00 \\_ ps -xf  1735 ? Sl 0:00 /usr/lib/gvfs/gvfsd-trash --spawner :1.1 /org/gtk/gvfs/exec\_spaw/0  1728 ? Sl 0:00 /usr/lib/gvfs/gvfsd-metadata  1642 ? Sl 0:00 /usr/lib/i386-linux-gnu/cinnamon-settings-daemon/csd-printer  1631 ? Sl 0:00 /usr/lib/gvfs/gvfs-mtp-volume-monitor  1626 ? Sl 0:00 /usr/lib/gvfs/gvfs-gphoto2-volume-monitor  1620 ? Sl 0:00 /usr/lib/gvfs/gvfs-afc-volume-monitor  1615 ? Sl 0:00 /usr/lib/gvfs/gvfs-goa-volume-monitor  1602 ? Sl 0:00 /usr/lib/gvfs/gvfs-udisks2-volume-monitor  1598 ? Sl 0:00 /usr/lib/dconf/dconf-service  1571 ? S<l 0:00 /usr/bin/pulseaudio --start --log-target=syslog  1555 ? Sl 0:00 /usr/bin/gnome-keyring-daemon --start --components=pkcs11  1549 ? Sl 0:00 /usr/lib/at-spi2-core/at-spi2-registryd --use-gnome-session  1542 ? Sl 0:00 /usr/lib/at-spi2-core/at-spi-bus-launcher  1547 ? S 0:00 \\_ /usr/bin/dbus-daemon --config-file=/etc/at-spi2/accessibility.conf --nofork  1532 ? Sl 0:00 /usr/lib/gvfs/gvfsd-fuse /run/user/1000/gvfs -f -o big\_writes  1527 ? Sl 0:00 /usr/lib/gvfs/gvfsd  1515 ? Ss 0:00 /usr/bin/dbus-daemon --fork --print-pid 6 --print-address 9 --session  1514 ? S 0:00 /usr/bin/dbus-launch --exit-with-session /usr/bin/im-launch cinnamon-session-cin  1507 ? S 0:00 /usr/bin/VBoxClient --draganddrop  1508 ? Sl 3:16 \\_ /usr/bin/VBoxClient --draganddrop  1500 ? S 0:00 /usr/bin/VBoxClient --seamless  1501 ? Sl 0:00 \\_ /usr/bin/VBoxClient --seamless  1491 ? S 0:00 /usr/bin/VBoxClient --display  1492 ? S 0:00 \\_ /usr/bin/VBoxClient --display  1480 ? S 0:00 /usr/bin/VBoxClient --clipboard  1481 ? Sl 0:00 \\_ /usr/bin/VBoxClient --clipboard  1353 ? Ss 0:00 /lib/systemd/systemd --user  1361 ? S 0:00 \\_ (sd-pam)  ^C  [1]+ Готово ./father |

В данном случае управление возвращается командной оболочке после прерывания процесса.

### Выполнить создание процессов с использованием различных функций семейства exec() с разными параметрами функций семейства, привести результаты эксперимента.

Семейство функций exec загружает и запускает другие программы, известные как "дочерние" процессы. Если вызов функции exec… завершается успешно, "дочерний" процесс накладывается на "родительский" процесс; причем должно быть достаточно памяти для загрузки и выполнения "дочернего" процесса.

Данное семейство имеет следующие прототипы:

* int execl(const char \*path, const char \*arg, ...);
* int execlp(const char \*file, const char \*arg, ...);
* int execle(const char \*path, const char \*arg, ..., char \* const envp[]);
* int execv(const char \*path, char \*const argv[]);
* int execvp(const char \*file, char \*const argv[]);
* int execvpe(const char \*file, char \*const argv[], char \*const envp[]);

Вышеприведенные прототипы отличаются принимаемыми аргументами. Если в названии функции содержится буква l, значит, аргументы для вызова программы передается через указатели на эти аргументы. Если в названии функции содержится буква v, то аргументы для вызова программы передается через указатель на массив аргументов. Если в названии функции есть буква p, значит, в качестве названия выполняемой программы принимается только имя файла, не содержащее слэш. Если в названии функции есть буква e, значит последним из аргументов, передаваемых в функцию, будет указатель на массив переменных окружения.

Все аргументы, списки параметров, массивы указателей должны заканчиваться NULL.

Программа ch1\_p8.cpp:

|  |
| --- |
| #include <unistd.h>  #include <stdlib.h>  int main(int argc, char \*\* argv){  char \*args[] = {"ls", "-l", NULL }; // список аргументов  char \* env[] = {(char\*)NULL}; // массив переменных окружения  switch ( atoi(argv[1]) ) { // в зависимости от введенного  case 1: // значения, выполняем ту или иную команду  execl("/bin/ls", "/bin/ls", "-l", (char \*)NULL);  break;  case 2:  execlp("ls", "ls", "-l", (char \*)NULL);  break;  case 3:  execle("/bin/ls", "ls", "-l", (char \*)NULL, env);  break;  case 4:  execv("/bin/ls", args);  break;  case 5:  execvp("ls", args);  break;  case 6:  execvpe("ls", args, env);  break;  default:  break;  }  return 0;  } |

Запуская программу с параметром (цифрой от 1 до 6), можно задать, какая функция из семейства exec будет использоваться. Результат при этом будет одинаковый.

Запуск функции execvp (case 5):

|  |
| --- |
| ana@AnaLinux ~/lab3 $ g++ -o ch1\_p8 ch1\_p8.cpp  ch1\_p8.cpp: In function ‘int main(int, char\*\*)’:  ch1\_p8.cpp:5:35: warning: deprecated conversion from string constant to ‘char\*’ [-Wwrite-strings]  char \*args[] = {"ls", "-l", NULL }; // список аргументов  ^  ch1\_p8.cpp:5:35: warning: deprecated conversion from string constant to ‘char\*’ [-Wwrite-strings]  ana@AnaLinux ~/lab3 $ ./ch1\_p8 5  итого 136  -rw-r--r-- 1 ana ana 8600 дек 19 2016 ch1\_p1  -rw-r--r-- 1 ana ana 367 дек 19 2016 ch1\_p1.cpp  -rw-r--r-- 1 ana ana 8600 дек 19 2016 ch1\_p2  -rw-r--r-- 1 ana ana 550 дек 19 2016 ch1\_p2.cpp  -rw-r--r-- 1 ana ana 8792 дек 19 2016 ch1\_p3  -rw-r--r-- 1 ana ana 957 дек 19 2016 ch1\_p3.cpp  -rwxr-xr-x 1 ana ana 7748 дек 5 01:05 ch1\_p4  -rw-r--r-- 1 ana ana 2714 дек 19 2016 ch1\_p4.cpp  -rwxr-xr-x 1 ana ana 7624 дек 5 01:26 ch1\_p8  -rw-r--r-- 1 ana ana 802 дек 19 2016 ch1\_p8.cpp  -rwxr-xr-x 1 ana ana 7620 дек 5 01:13 father  -rw-r--r-- 1 ana ana 325 дек 5 01:13 father.c  -rwxr-xr-x 1 ana ana 7464 ноя 21 12:46 lab3\_1  -rw-r--r-- 1 ana ana 369 ноя 21 12:44 lab3\_1.cpp  -rwxr-xr-x 1 ana ana 7464 ноя 29 23:03 lab3\_2  -rw-r--r-- 1 ana ana 552 ноя 29 23:02 lab3\_2.cpp  -rwxr-xr-x 1 ana ana 7628 ноя 29 23:32 lab3\_3  -rw-r--r-- 1 ana ana 1180 ноя 29 23:32 lab3\_3.cpp  -rw-r--r-- 1 ana ana 238 дек 5 01:04 onecore.h  -rwxr-xr-x 1 ana ana 7460 дек 5 01:11 son  -rw-r--r-- 1 ana ana 140 дек 19 2016 son.c |

### Проанализируйте значение, возвращаемое функцией wait(&status). Предложите эксперимент, позволяющий родителю отслеживать подмножество по- рожденных потомков, используя различные функции семейства wait().

Функция wait приостанавливает выполнение текущего процесса до тех пор, пока какой-либо 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"), то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются.

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 (так называемый "зомби"), то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются.

Данные функции имеют следующие прототипы:

* wait(int \*status);
* waitpid(pid\_t pid, int \*status, int options);

Параметр status может принимать несколько значений:

* < -1 означает, что нужно ждать любого дочернего процесса, идентификатор группы процессов которого равен абсолютному значению pid
* -1 означает ожидание любого дочернего процесса; функция wait ведет себя точно так же
* 0 означает ожидание любого дочернего процесса, идентификатор группы процессов которого равен идентификатору текущего процесса
* > 0 означает ожидание дочернего процесса, чей идентификатор равен pid

Значение options создается путем логического сложения нескольких следующих констант:

* 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 под процессов также обеспечивается без этой опции.

Используя различные макросы можно получить больше информации:

* 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 до- чернего 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.

Программа wait.cpp:

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  #include <sys/wait.h>  #include <stdlib.h>  int main() {  int pid = fork(); // create process  if (pid != 0) { // parent  int status = 0;  int res = wait(&status);    printf("WIFEXITED\t%i\n",WIFEXITED(status));  printf("WEXITSTATUS\t%i\n",WEXITSTATUS(status));  printf("WIFSIGNALED\t%i\n",WIFSIGNALED(status));  printf("WTERMSIG\t%i\n",WTERMSIG(status));  printf("WIFSTOPPED\t%i\n",WIFSTOPPED(status));  printf("WSTOPSIG\t%i\n",WSTOPSIG(status));  printf("Status\t\t%i\n",status);  printf("Pid\t\t%i\n",pid);  printf("Wait result\t%i\n",res);  } else {  printf("Son finished\n");  exit(123);  }  } |

Выполнение wait.cpp:

|  |
| --- |
| ana@AnaLinux ~/lab3 $ g++ -o wait wait.cpp  ana@AnaLinux ~/lab3 $ ./wait  Son finished  WIFEXITED 1 // процесс завершился успешно  WEXITSTATUS 123 // код выхода  WIFSIGNALED 0 // завершился по обработанному сигналу  WTERMSIG 0 // не используятся в данной ситуации  WIFSTOPPED 0 // не используется в данной ситуации  WSTOPSIG 123 // номер сигнала, из-за которого дочерний процесс был остановлен  Status 31488 // статус завершения  Pid 18084 // идентификатор потомка  Wait result 18084 // идентификатор дочернего процесса |

Программа порождает процесс потомок и в ней выводится сообщение о том, что это дочерний процесс. Далее программа завершается со статусом 123, а в родительской программе в это время идет ожидание изменения статуса дочернего процесса, в данном случае его окончание. После завершения ожидания окончания дочернего процесса в переменную status записывается состояние этого процесса.

**Функция waitpid**

Для исследования функции waitpid была написана программа waitpid, которая запускает программы son, daughter, wife.

Программа son.c:

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  int main(void){  printf("Son params: pid=%d, ppid=%d\n",getpid(),getppid());  sleep(1);  return 0;  } |

Программа daughter.c:

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  int main(void){  printf("Daughter params: pid=%d, ppid=%d\n",getpid(),getppid());  sleep(2);  return 0;  } |

Программа wife.c:

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  int main(void){  printf("Wife params: pid=%d, ppid=%d\n",getpid(),getppid());  sleep(3);  return 0;  } |

Программа waitpid.cpp:

|  |
| --- |
| #include <sys/wait.h>  #include <unistd.h>  #include <stdio.h>  #include <string.h>  #include <stdlib.h>  int main() {  int i, pid[4], status[3], result[3];  char \*son[] = {"son","daughter","wife"};  // флаги исполнения  int option[] = {WNOHANG, WUNTRACED, WNOHANG}; // wait 1  //int option[] = {WUNTRACED, WUNTRACED, WUNTRACED}; // wait 2  //int option[] = {WNOHANG, WUNTRACED, WUNTRACED}; // wait 3  printf("father parametrs: pid= %i, ppid =%i\n", getpid(), getppid());  for(i=0;i<3;i++)  if((pid[i] = fork()) == 0)  execl(son[i],son[i],NULL); // create 8 processes    system("ps xf > log1.txt"); // write about runing process  for(i = 0; i < 3; i++){  result[i] = waitpid(pid[i], &status[i], option[i]); // останавливает текущий процесс, пока поражденный им процесс не завершится. Информация о завершении запишется в статус, в options задаются настройки waitpid  printf("%i child pid = %i finished with status %i\n",(i+1), result[i], status[i]);  }  for(i =0; i< 3; i++){  if(WIFEXITED(status[i]) == 0){ // проверка на успешность завершения процесса  printf("process pid= %i was faild\n", pid[i]);  }else{  printf("process pid= %i was success\n", pid[i]);  }  }  return 0;  } |

Запустим waitpid.cpp с различными значениями option.

Пример 1: запуск waitpid.cpp при option = WNOHANG, WUNTRACED, WNOHANG:

|  |
| --- |
| ana@AnaLinux ~/lab3 $ g++ -o waitpid waitpid.cpp  ana@AnaLinux ~/lab3 $ ./waitpid  father parametrs: pid= 18151, ppid =1950  Wife params: pid=18154, ppid=18151  Daughter params: pid=18153, ppid=18151  Son params: pid=18152, ppid=18151  1 child pid = 0 finished with status 36183  2 child pid = 18153 finished with status 0  3 child pid = 0 finished with status 47  process pid= 18152 was faild  process pid= 18153 was success  process pid= 18154 was faild |

Пример 2: запуск waitpid.cpp при option = WUNTRACED, WUNTRACED, WUNTRACED

|  |
| --- |
| ana@AnaLinux ~/lab3 $ ./waitpid  father parametrs: pid= 18168, ppid =1950  Wife params: pid=18171, ppid=18168  Daughter params: pid=18170, ppid=18168  Son params: pid=18169, ppid=18168  1 child pid = 18169 finished with status 0  2 child pid = 18170 finished with status 0  3 child pid = 18171 finished with status 0  process pid= 18169 was success  process pid= 18170 was success  process pid= 18171 was success |

Пример 3: запуск waitpid.cpp при option = WNOHANG, WUNTRACED, WUNTRACED

|  |
| --- |
| ana@AnaLinux ~/lab3 $ ./waitpid  father parametrs: pid= 18185, ppid =1950  Wife params: pid=18188, ppid=18185  Daughter params: pid=18187, ppid=18185  Son params: pid=18186, ppid=18185  1 child pid = 0 finished with status 19799  2 child pid = 18187 finished with status 0  3 child pid = 18188 finished with status 0  process pid= 18186 was faild  process pid= 18187 was success  process pid= 18188 was success |

WNOHANG – означает вернуть управление немедленно, если ни один дочерний процесс не завершил выполнение.

WUNTRACED – означает возвращать управление также для остановленных дочерних процессов, о чьем статусе еще не было сообщено. В первом опыте 2 процесса завершились с ошибками. Во втором опыте все три процесса завершились успешно, т.к. управление возвращается даже для остановленных дочерних процессов. В третьем опыте один процесс снова с ошибкой, т.к. WNOHANG не вернул ему управление, и процесс так и остался ждать уже остановленный дочерний процесс.

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

Уберем в коде son.c задержку на возврат результата (son.c без задержки):

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  int main(void){  printf("Son params: pid=%d, ppid=%d\n",getpid(),getppid());  return 0;  } |

Вновь запустим waitpid.cpp с параметрами option = WNOHANG, WUNTRACED, WNOHANG.

|  |
| --- |
| ana@AnaLinux ~/lab3 $ ./waitpid  father parametrs: pid= 18222, ppid =1950  Daughter params: pid=18224, ppid=18222  Wife params: pid=18225, ppid=18222  Son params: pid=18223, ppid=18222  1 child pid = 18223 finished with status 0  2 child pid = 18224 finished with status 0  3 child pid = 0 finished with status 47  process pid= 18223 was success  process pid= 18224 was success  process pid= 18225 was faild |

Дополнительно рассмотрим процессы после запуска waitpid.cpp:

|  |
| --- |
| PID TTY STAT TIME COMMAND  ...  1950 pts/0 Ss 0:00 \\_ bash  18259 pts/0 S+ 0:00 \\_ ./waitpid  18260 pts/0 Z+ 0:00 \\_ [son] <defunct>  18261 pts/0 S+ 0:00 \\_ daughter  18262 pts/0 S+ 0:00 \\_ wife  18263 pts/0 S+ 0:00 \\_ sh -c ps xf > log1.txt  18264 pts/0 R+ 0:00 \\_ ps xf  ... |

Как видно у процесса son имеется приставка это означает что процесс уже выполнился, но все еще остался в списке процессов. Использование WHOHANG для данного процесса подтверждает корректное возращение управления.

### Проанализируйте очередность исполнения процессов.

#### Очередность процессов, порожденных вложенными вызовами fork().

#### Измените процедуру планирования с помощью функции с шаблоном scheduler в ее названии и повторите эксперимент.

#### Поменяйте порядок очереди в RR-процедуре.

Изменим программу ch1\_p4.cpp, для реализации очередности процессов, порожденных вложенными вызовами fork().

Полученная программа ch1\_p10.cpp:

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  #include <sched.h>  #include <stdlib.h>  #include "onecore.h"  using namespace std;  int main(int argc, char \*\*argv) {  struct sched\_param shdprm; // значение параметров планирования  int pid1, pid2, pid3, pid4;  int ppid1, ppid2, ppid3, ppid4;  int p0,p1,p2,p3; // переменные для задания значения приоритетов  int policy;  if(atoi(argv[1])==1)  policy=SCHED\_FIFO;  else  policy=SCHED\_RR;    p0 = atoi(argv[2]);  p1 = atoi(argv[3]);  p2 = atoi(argv[4]);  p3 = atoi(argv[5]);    pid1=getpid();  ppid1=getppid();  printf("pid=%d\n",pid1); // родительский pid  printf("ppid=%d\n",ppid1); // родительский ppid  one\_core(pid1);    shdprm.sched\_priority=p0; // установка приоритета  if(sched\_setscheduler(0,policy,&shdprm) == -1) // если ошибка  printf("ERROR\n");  if((fork()) == 0){ // 1 уровень вложенности  pid2=getpid();  one\_core(pid2);  ppid2=getppid();  shdprm.sched\_priority = p1; // установка приоритета  if((fork()) == 0){ // 2 уровень вложенности  pid3=getpid();  one\_core(pid3);  ppid3=getppid();  shdprm.sched\_priority = p2; // установка приоритета  if((fork()) == 0){ // 3 уровень вложенности  pid4=getpid();  one\_core(pid4);  ppid4=getppid();  shdprm.sched\_priority = p3; // установка приоритета  if(sched\_setscheduler(getpid(), policy, &shdprm) == -1)  printf("ERROR\n");  sleep(1);  printf("3 lvl, pid=%d, ppid=%d\n", pid4, ppid4);  }    if(sched\_setscheduler(getpid(), policy, &shdprm) == -1)  printf("ERROR\n");  sleep(1);  printf("2 lvl, pid=%d, ppid=%d\n", pid3, ppid3);  }    if(sched\_setscheduler(getpid(),policy,&shdprm) == -1)  printf("ERROR\n");  sleep(1);  printf("1 lvl, pid=%d, ppid=%d\n", pid2, ppid2);  }  sleep(3);  } |

Переменные p0, p1, p2, p3 отвечают за приоритеты выполнения процессов. Для запуска данной программы, в качестве входных параметров требуется ввести 5 переменных. Первая для определения алгоритма, 1 = SCHED\_FIFO иначе SCHED\_RR. Оставшиеся 4 переменные отвечают за приоритеты выполнения.

**Алгоритм SCHED\_FIFO**

Пример 1: выполнения ch1\_p10.cpp при p0=99, p1=99, p2=50, p3=1

|  |
| --- |
| ana@AnaLinux ~/lab3 $ sudo ./ch1\_p10 1 99 99 50 1  [sudo] пароль для ana:  pid=18283  ppid=18282  1 lvl, pid=18284, ppid=18283  2 lvl, pid=18285, ppid=18284  3 lvl, pid=18286, ppid=18285  1 lvl, pid=18284, ppid=18283  2 lvl, pid=18285, ppid=18284  1 lvl, pid=18284, ppid=18283 |

Пример 2: выполнения ch1\_p10.cpp при p0=99, p1=50, p2=50, p3=50

|  |
| --- |
| ana@AnaLinux ~/lab3 $ sudo ./ch1\_p10 1 99 50 50 50  pid=18288  ppid=18287  3 lvl, pid=18291, ppid=18290  2 lvl, pid=18290, ppid=18289  1 lvl, pid=18289, ppid=18288  2 lvl, pid=18290, ppid=18289  1 lvl, pid=18289, ppid=18288  1 lvl, pid=18289, ppid=18288 |

Пример 3: выполнения ch1\_p10.cpp при p0=99, p1=1 p2=50, p3=99

|  |
| --- |
| ana@AnaLinux ~/lab3 $ sudo ./ch1\_p10 1 99 1 50 99  pid=18293  ppid=18292  3 lvl, pid=18296, ppid=18295  2 lvl, pid=18295, ppid=18294  1 lvl, pid=18294, ppid=18293  2 lvl, pid=18295, ppid=18294  1 lvl, pid=18294, ppid=18293  1 lvl, pid=18294, ppid=18293 |

**Алгоритм SCHED\_RR**

Пример 1: выполнения ch1\_p10.cpp при p0=99, p1=99, p2=50, p3=1

|  |
| --- |
| ana@AnaLinux ~/lab3 $ sudo ./ch1\_p10 2 99 99 50 1  pid=18304  ppid=18303  1 lvl, pid=18305, ppid=18304  2 lvl, pid=18306, ppid=18305  3 lvl, pid=18307, ppid=18306  1 lvl, pid=18305, ppid=18304  2 lvl, pid=18306, ppid=18305  1 lvl, pid=18305, ppid=18304 |

Пример 2: выполнения ch1\_p10.cpp при p0=99, p1=50, p2=50, p3=50

|  |
| --- |
| ana@AnaLinux ~/lab3 $ sudo ./ch1\_p10 2 99 50 50 50  pid=18309  ppid=18308  3 lvl, pid=18312, ppid=18311  2 lvl, pid=18311, ppid=18310  1 lvl, pid=18310, ppid=18309  2 lvl, pid=18311, ppid=18310  1 lvl, pid=18310, ppid=18309  1 lvl, pid=18310, ppid=18309 |

Пример 3: выполнения ch1\_p10.cpp при p0=99, p1=1, p2=50, p3=99

|  |
| --- |
| ana@AnaLinux ~/lab3 $ sudo ./ch1\_p10 2 99 1 50 99  pid=18314  ppid=18313  3 lvl, pid=18317, ppid=18316  2 lvl, pid=18316, ppid=18315  1 lvl, pid=18315, ppid=18314  2 lvl, pid=18316, ppid=18315  1 lvl, pid=18315, ppid=18314  1 lvl, pid=18315, ppid=18314 |

Результаты SCHED\_RR оказались аналогичны результатам SCHED\_FIFO. Как видно из вышеприведенных примеров, при вложенных процессах, первым выполняется процесс с наибольшим приоритетом. При равных приоритетах, процессы выполняются по принципу FIFO.

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

В следующей программе ch1\_p10\_4.cpp приоритеты выполнения для процессов одинаковы, но будет отличаться процедура планирования.

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  #include <sched.h>  #include <stdlib.h>  #include "onecore.h"  using namespace std;  void delay(){  int sum=0;  for(int k = 0; k< 100000000; k++){  sum = sum + k;  }  }  int main(int argc, char \*\*argv) {  struct sched\_param shdprm; // значение параметров планирования  int pid,pid1,pid2,pid3,status;  int p=50; // переменная для задания значения пиоритетов    pid = getpid(); // get id process  one\_core(pid);  int ppid = getppid(); // get parent`s process id  printf("Parent params: pid=%d, ppid=%d\n",pid,ppid);    shdprm.sched\_priority=p; // set priority  if(sched\_setscheduler(0,SCHED\_RR,&shdprm) == -1)  perror("SCHED\_SETSCHEDULER");  if((pid1 = fork()) == 0){ // create process 1  shdprm.sched\_priority = p;  if(sched\_setscheduler(pid1,SCHED\_FIFO,&shdprm) == -1)  perror("SCHED\_SETSCHEDULER\_1");  delay(); // delay  printf("1 lvl stop, pid=%d\n",getpid());  }  if((pid2 = fork()) == 0){ // create process 2  shdprm.sched\_priority = p;  if(sched\_setscheduler(pid2, SCHED\_RR, &shdprm) == -1)  perror("SCHED\_SETSCHEDULER\_2");  delay(); // delay  printf("2 lvl stop, pid=%d\n",getpid());  }  if((pid3 = fork()) == 0){ // create process 3  shdprm.sched\_priority = p;  if(sched\_setscheduler(pid3, SCHED\_RR, &shdprm) == -1)  perror("SCHED\_SETSCHEDULER\_3");  delay(); // delay  printf("3 lvl stop, pid=%d\n",getpid());  }  sleep(5);  } |

Запустим данную программу изменяя процедуру планирования для процессов, при равных приоритетах.

Выполнения ch1\_p10\_4.cpp при: 1=SCHED\_FIFO, 2=SCHED\_FIFO, 3=SCHED\_FIFO

|  |
| --- |
| ana@AnaLinux ~/lab3 $ sudo ./ch1\_p10\_4 50 50 50 50  Parent params: pid=18353, ppid=18352  1 lvl stop, pid=18354  2 lvl stop, pid=18355  3 lvl stop, pid=18356  2 lvl stop, pid=18357  3 lvl stop, pid=18358  3 lvl stop, pid=18359  3 lvl stop, pid=18360 |

Выполнения ch1\_p10\_4.cpp при: 1=SCHED\_FIFO, 2=SCHED\_RR, 3=SCHED\_FIFO

|  |
| --- |
| ana@AnaLinux ~/lab3 $ sudo ./ch1\_p10\_4 50 50 50 50  Parent params: pid=18373, ppid=18372  1 lvl stop, pid=18374  3 lvl stop, pid=18376  3 lvl stop, pid=18378  2 lvl stop, pid=18375  2 lvl stop, pid=18377  3 lvl stop, pid=18379  3 lvl stop, pid=18380 |

Выполнения ch1\_p10\_4.cpp при: 1=SCHED\_RR, 2=SCHED\_RR, 3=SCHED\_FIFO

|  |
| --- |
| ana@AnaLinux ~/lab3 $ sudo ./ch1\_p10\_4 50 50 50 50  Parent params: pid=18392, ppid=18391  3 lvl stop, pid=18395  1 lvl stop, pid=18393  2 lvl stop, pid=18394  3 lvl stop, pid=18397  3 lvl stop, pid=18398  2 lvl stop, pid=18396  3 lvl stop, pid=18399 |

Выполнения ch1\_p10\_4.cpp при: 1=SCHED\_FIFO, 2=SCHED\_RR, 3=SCHED\_RR

|  |
| --- |
| ana@AnaLinux ~/lab3 $ sudo ./ch1\_p10\_4 50 50 50 50  Parent params: pid=18333, ppid=18332  1 lvl stop, pid=18334  2 lvl stop, pid=18335  3 lvl stop, pid=18336  2 lvl stop, pid=18337  3 lvl stop, pid=18338  3 lvl stop, pid=18339  3 lvl stop, pid=18340 |

Было проведено 4 эксперимента.

* В первом опыте у всех процессов одинаковая политика планирования, поэтому они выполняются согласно порядку попадания в очередь.
* Во втором опыте, второй процесс выполнился позже, чем первый и третий процессы.
* В третьем опыте первый и вторые процессы делили кванты между собой, из-за чего третий процесс их обошли.

Из проведенных опытов можно сделать вывод, что политика квантования (RR) медленнее политики без квантования (FIFO), однако она имеет свои преимущества, поэтому не стоит считать ее хуже. В зависимости от поставленной задачи — выбирается политика планирования. Если все процессы равноправны (например, добавление в БД большого кол-ва однотипных запросов), то лучше использовать FIFO — добавили и забыли. Если же процессы длительные (например идет передача информации различным пользователям), то лучше использовать политику квантования и разделять «скорость передачи» между клиентами. Только при правильном выборе политики планирования можно выиграть время при выполнении процессов.

### Определите величину кванта. Можно ли ее поменять? Для обоснования проведите эксперимент.

Воспользуемся программой из методички [1], для того, чтобы определить длину кванта.

Программа ch1\_p11.cpp:

|  |
| --- |
| #include <stdio.h>  #include <sched.h>  #include <sys/wait.h>  #include <sys/mman.h>  #include <unistd.h>  int main(void){  struct sched\_param shdprm;  struct timespec qp;  int pid, pid1, ppid, status;  pid = getpid(); // father pid  ppid = getppid(); // grandfather pid  printf("FATHER PARAM: pid=%i ppid=%i \n", pid,ppid);  shdprm.sched\_priority = 50; // set priority  if(sched\_setscheduler(0,SCHED\_RR,&shdprm) == -1)// set RR  perror("SCHED\_SETSCHEDULER\_1");  if(sched\_rr\_get\_interval(0,&qp) == 0) // get current interval  printf("Квант при циклическом планировании: %g сек\n", qp.tv\_sec + qp.tv\_nsec/1000000000.0);  else  perror("SCHED\_RR\_GET\_INTERVAL");  if((pid1=fork()) == 0) // create process  {  if(sched\_rr\_get\_interval(pid1, &qp) == 0)// get son interval  printf("SON: Квант процессорного времени: %g сек\n", qp.tv\_sec + qp.tv\_nsec/1000000000.0);  execl("son","son",NULL); // exe son process  }  printf("Процесс с = %d завершен \n", wait(&status));  return 0;  } |

Код son представлен в одном из предыдущих пунктов. Результатом работы программы стало:

|  |
| --- |
| ana@AnaLinux ~/lab3 $ sudo ./ch1\_p11  FATHER PARAM: pid=18408 ppid=18407  Квант при циклическом планировании: 0.1 сек  SON: Квант процессорного времени: 0.1 сек  Son params: pid=18409, ppid=18408  Процесс с = 18409 завершен |

Теперь попробуем изменить длину кванта, для этого реализуем другую программу.

Программа ch1\_p11\_2.cpp:

|  |
| --- |
| #include <stdio.h>  #include <math.h>  #include <sys/resource.h>  #include <unistd.h>  #include <sched.h>  int main(){    struct sched\_param param;  struct timespec ts;    param.sched\_priority = 99;    nice(-20); // приоритет -20, для увеличения длины кванта  if (sched\_setscheduler(0, SCHED\_RR, &param) == -1) { // установили политику планирования с квантованием  perror("sched\_setscheduler failed\n");  return 3;  }  sched\_rr\_get\_interval(0, &ts); // получить длину кванта  printf ("max timeslice = %ld msec\n", ts.tv\_nsec/1000000); // выводим    nice(39); // приоритет 39, для уменьшения длины кванта  if (sched\_setscheduler(0, SCHED\_RR, &param) == -1) { // установка политики планирования  perror("sched\_setscheduler failed\n");  return 3;  }  sched\_rr\_get\_interval(0, &ts); // получение кванта  printf ("min timeslice = %ld msec\n", ts.tv\_nsec/1000000); // выводим    return 1;  } |

Сначала попробовали сделать квант длиннее, записав положительно число, затем меньше

Выполнение ch1\_p11\_2.cpp

|  |
| --- |
| ana@AnaLinux ~/lab3 $ sudo ./ch1\_p11\_2  max timeslice = 100 msec  min timeslice = 100 msec |

Как видно, квант времени не изменился.

### 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.

Файловые дескрипторы полностью наследуются при системном вызове fork(). Более того фай- ловые дескрипторы остаются открытыми при системном вызове execve(). Закрываются только те дескрипторы. Которые помечены как close-on-exec. Проведем эксперимент при помощи программы. Откроем в основном процессе файл. Создадим два потомка при помощи разных системных вызовов. Один потомок создается при помощи fork(), а другой при помощи после- довательности вызовов fork() и exec(). Затем попытаемся в обоих потомках прочитать файл, открытый в родительской программе.

Файл для чтения содержит следующий текст: "Hello, World".

Программа ch1\_p12\_father.cpp:

|  |
| --- |
| #include <unistd.h>  #include <fcntl.h>  #include <stdio.h>  #include <stdlib.h>  int main(int argc, char \* argv[]) {  int filelen = 14; // file`s lenght  int fd = open("read\_test", O\_RDONLY); // open by read  if (fd < 0) { // check  perror("open");  return 0;  }  char line;  int pid = fork(); // create process  if (pid == 0) { // son  printf("Readed by fork son: ");  int i = 0;  while (i < filelen) {  pread(fd, &line, 1, i++); // read  printf("%c", line); // write in console  }  printf("\n");  char \*name="./ch1\_p12\_daughter";  char \*arg[3]; // create array arg and targ  char targ[10];  sprintf(targ, "%d", fd); // write in targ file`s descriptor  arg[0]=name; // name executed file  arg[1]=targ; // in arg[1] write targ[ ]  arg[2]=NULL;  execvp(name,arg); // exe daughter process  } else {  printf("Readed by parent: "); // parent  int i = 0;  while (i < filelen) { // read  pread(fd, &line, 1, i++);  printf("%c", line);  }  printf("\n");  close(fd); // close file  }  sleep(1);  } |

Программа ch1\_p12\_daughter.cpp

|  |
| --- |
| #include <unistd.h>  #include <stdio.h>  #include <stdlib.h>  int main(int argc, char \* argv[]) {  int fd = atoi(argv[1]); // file`s discriptor  char line;  printf("Readed by exec daughter from file: "); // message  int i = 0;  while (i < 14) { // read information  pread(fd, &line, 1, i++);  printf("%c", line);  }  printf("\n");  } |

Результаты выполнения ch1\_p12\_father.cpp:

|  |
| --- |
| ana@AnaLinux ~/lab3 $ ./ch1\_p12\_father  Readed by parent: Hello, World  Readed by fork son: Hello, World  Readed by exec daughter from file: Hello, World |

Как видно, все процессы смогли прочитать содержимое файла. Это потому, что при создании нового процесса процесс копируется полностью со всеми данными, включая дескрипторы. При вызове дочери — чтение возможно благодаря тому, что файловые дескрипторы остаются открытыми. Файловый дескриптор можно передать как параметр для программы и считать содержимое из уже открытого в родительском процессе файла.

## Взаимодействие родственных процессов

### 2.2.1. Изменяя длительности выполнения процессов и параметры системных вызовов, рассмотреть 3 ситуации и вывести соответствующие таблицы процессов

#### 2.2.1.1. процесс-отец запускает процесс-сын и ожидает его завершения

Напишем программы процесса-отца и процесса-потомка.

Программа ch2\_p1\_a\_father.cpp:

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  #include <sys/wait.h>  #include <stdlib.h>  int main(void){  int status;  printf("FATHER PARAM: pid=%i ppid=%i \n", getpid(),getppid());  if(fork()==0)  execl("ch2\_p1\_a\_son", NULL); // создание процесса-потомка  system("ps xf > log"); // запись состояни процессов в файл  wait(&status); // ожидание завершение потомка  printf("father finished\n"); // сообщение об окончании работы  return 0;  } |

Программа ch2\_p1\_a\_son.cpp:

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  main(){  sleep(5); // пауза  printf("SON PARAM: pid=%i ppid=%i \n",getpid(),getppid());  printf("son finished\n");// сообщение  } |

Запустим процесс-отца:

|  |
| --- |
| ana@AnaLinux ~/lab3 $ ./ch2\_p1\_a\_father  FATHER PARAM: pid=18461 ppid=1950  SON PARAM: pid=18462 ppid=18461  son finished  father finished |

Процесс-отец дождался завершения работы процесса-потомка, затем завершился сам.

#### 2.2.1.2. процесс-отец запускает процесс-сын и, не ожидая его завершения, завершает свое выполнение. Зафиксируйте изменение родительского идентификатора процесса-сына

Код программ аналогичен программам из пункта а, за исключением того что в процессе-отце убрана строчка(wait&(status);) с ожиданием-процесса потомка.

Программа ch2\_p1\_b\_father.cpp:

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  #include <sys/wait.h>  #include <stdlib.h>  int main(void){  int status;  printf("FATHER PARAM: pid=%i ppid=%i \n", getpid(),getppid());  if(fork()==0)  execl("ch2\_p1\_a\_son", NULL); // создание процесса-потомка  printf("father finished\n"); // сообщение об окончании работы  return 0;  } |

Выполнение ch2\_p1\_b\_father.cpp

|  |
| --- |
| ana@AnaLinux ~/lab3 $ ./ch2\_p1\_b\_father  FATHER PARAM: pid=18482 ppid=1950  father finished  SON PARAM: pid=18483 ppid=1  son finished |

После того, как родитель прекратил свое существование, идентификатор родителя у процесса потомка сменился на 1. Это корневой процесс init который запускается ядром при загрузке системы. Одной из его задач является "усыновлять" процессы которые остались без родительского процесса. Это означает что процесс init становиться родительским процессом, несмотря на то, что в реальности они не были им порождены.

#### 2.2.1.3. процесс-отец запускает процесс-сын и не ожидает его завершения; процесс-сын завершает свое выполнение. Зафиксируйте появление процесса-зомби, для этого включите команду ps в программу father.c

Программа ch2\_p1\_v\_father.cpp:

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  #include <sys/wait.h>  #include <stdlib.h>  int main(void){  int status;  printf("FATHER PARAM: pid=%i ppid=%i \n", getpid(),getppid());  if(fork()==0)  execl("ch2\_p1\_a\_son", NULL); // создание процесса-потомка  system("ps xf > log"); // запись в лог  sleep(6); // пауза  system("ps xf > log2"); // запись в лог  printf("father finished\n"); // сообщение об окончании работы  return 0;  } |

Выполнение ch2\_p1\_v\_father.cpp:

|  |
| --- |
| ana@AnaLinux ~/lab3 $ ./ch2\_p1\_v\_father  FATHER PARAM: pid=18491 ppid=1950  SON PARAM: pid=18492 ppid=18491  son finished  father finished |

Состояния процессов во время выполнения ch2\_p1\_v\_father.cpp:

|  |
| --- |
| PID TTY STAT TIME COMMAND  ...  1946 ? Sl 0:07 /usr/lib/gnome-terminal/gnome-terminal-server  1950 pts/0 Ss 0:00 \\_ bash  18491 pts/0 S+ 0:00 \\_ ./ch2\_p1\_v\_father  18492 pts/0 S+ 0:00 \\_ [ch2\_p1\_a\_son]  18493 pts/0 S+ 0:00 \\_ sh -c ps xf > log  18494 pts/0 R+ 0:00 \\_ ps xf  ...  //sleep 6  ...  1946 ? Sl 0:07 /usr/lib/gnome-terminal/gnome-terminal-server  1950 pts/0 Ss 0:00 \\_ bash  18491 pts/0 S+ 0:00 \\_ ./ch2\_p1\_v\_father  18492 pts/0 Z+ 0:00 \\_ [ch2\_p1\_a\_son] <defunct>  18496 pts/0 S+ 0:00 \\_ sh -c ps xf > log2  18497 pts/0 R+ 0:00 \\_ ps xf  ... |

Сначала оба процесса работают, затем сын завершает свою задачу. У него отнимаются ресурсы, и он становится зомби-процессом. Полностью он удалится только после того, как завершится родительский процесс, который связан с процессом-сыном.

### 2.2.2. 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.

Был написан скрипт, который запускает программы процессов-отцов из пунктов выше и выводит результаты их исполнения в консоль и в файл.

Скрипт ch2\_p2.script:

|  |
| --- |
| #!/bin/bash  echo 'Results of punkt a:' | tee tempFile  ./ch2\_p1\_a\_father | tee -a tempFile  echo 'Results of punkt b:' | tee -a tempFile  ./ch2\_p1\_b\_father | tee -a tempFile  sleep 6  echo 'Results of punkt v:' | tee -a tempFile  ./ch2\_p1\_v\_father | tee -a tempFile |

Результат выполнения (Содержимое tempFile):

|  |
| --- |
| Results of punkt a:  SON PARAM: pid=18561 ppid=18559  son finished  FATHER PARAM: pid=18559 ppid=18556  father finished  Results of punkt b:  FATHER PARAM: pid=18568 ppid=18556  father finished  SON PARAM: pid=18570 ppid=1  son finished  Results of punkt v:  SON PARAM: pid=18578 ppid=18576  son finished  FATHER PARAM: pid=18576 ppid=18556  father finished |

Все три программы были успешно выполнены, а результат их выполнения соответствует ожиданиям.

## Управление процессами посредством сигналов

### 2.3.1. С помощью команды kill -l ознакомьтесь с перечнем сигналов, поддерживаемых процессами. Ознакомьтесь с системными вызовами kill(2), signal(2).

С помощью команды kill -l был выведен список существующих сигналов.

|  |
| --- |
| ana@AnaLinux ~/lab3 $ 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 |

Системный вызов kill может быть использован для посылки какого-либо сигнала какому-либо процессу или группе процесса. Имеет следующий прототип:

int kill(pid\_t pid, int sig);

Так-же имеются следующий особенности:

* Если значение pid является положительным, сигнал sig посылается процессу с идентификатором pid.
* Если pid равен 0, то sig посылается каждому процессу, который входит в группу текущего процесса.
* Если pid равен -1, то sig посылается каждому процессу, за исключением процесса с номером 1 (init), но есть нюансы, которые описываются ниже.
* Если pid меньше чем -1, то sig посылается каждому процессу, который входит в группу процесса -pid.
* Если sig равен 0, то никакой сигнал не посылается, а только выполняется проверка на ошибку

signal(int sig, void (\*func)(int)) назначает действие, предпринимаемое при получении программой сигнала sig. Если значение func равно SIG\_DFL, то происходит обработка по умолчанию для указанного сигнала. Если значение func равно SIG\_IGN, то сигнал игнорируется. В остальных случаях func указывает на вызываемую функцию обработчик сигнала при получении сигнала. Функция func может завершиться выполнением операции возврата или вызовом функций самозавершения, выхода или перехода.

### 2.3.2. Подготовьте программы следующего содержания:

#### процесс father порождает процессы son1, son2, son3 и запускает на исполнение программные коды из соответствующих исполнительных файлов;

#### далее родительский процесс осуществляет управление потомками, для этого он генерирует сигнал каждому пользовательскому процессу;

#### в пользовательских процессах-потомках необходимо обеспечить: для son1 - реакцию на сигнал по умолчанию; для son2 - реакцию игнорирования; для son3 - перехватывание и обработку сигнала.

#### сформируйте файл-проект из четырех файлов, откомпилируйте, запустите программу. Проанализируйте таблицу процессов до и после посылки сигналов с помощью системного вызова system("ps -s » file").

#### обратите внимание на реакцию, устанавливаемую для последнего потомка.

Проведем такой эксперимент, который бы выполнил все три пункта сразу. Для этого процесс father будет порождать процессы son1, son2, son3 и запускать на исполнение программные коды из соответствующих исполнительных файлов. Далее родительский процесс осуществляет управление потомками, для этого он генерирует сигнал каждому пользовательскому процессу.

Программа ch3\_p1\_father.cpp:

|  |
| --- |
| #include <unistd.h>  #include <signal.h>  #include <stdio.h>  #include <sys/wait.h>  #include <stdlib.h>  int main() {  int pid1, pid2, pid3, status;  printf("Father params: pid=%d, ppid=%d\n",getpid(),getppid());  if ((pid1 = fork()) == 0)  execl("ch3\_p1\_son1", "son1", NULL); // порождение son1  if ((pid2 = fork()) == 0)  execl("ch3\_p1\_son2", "son2", NULL); // порождение son2  if ((pid3 = fork()) == 0)  execl("ch3\_p1\_son3", "son3", NULL); // порождение son3  system("ps -s>log1"); // запись текущих процессов  kill(pid1, SIGUSR1); // kill сигнал для son1  system("ps -s>log2"); // запись текущих процессов  kill(pid2, SIGUSR1); // kill сигнал для son2  system("ps -s>log3"); // запись текущих процессов  kill(pid3, SIGUSR1); // kill сигнал для son3  system("ps -s>log4"); // запись текущих процессов  return 0;  } |

Программа ch3\_p1\_son1.cpp:

|  |
| --- |
| #include <unistd.h>  #include <signal.h>  #include <stdio.h>  int main() {  printf("Son1 params: pid=%d, ppid=%d\n",getpid(),getppid());  signal(SIGUSR1, SIG\_DFL); // обработка по умолчанию  sleep(10);  } |

Программа ch3\_p1\_son2.cpp:

|  |
| --- |
| #include <unistd.h>  #include <signal.h>  #include <stdio.h>  int main() {  printf("Son2 params: pid=%d, ppid=%d\n",getpid(),getppid());  signal(SIGUSR1, SIG\_IGN); // игнорирование  sleep(10);  } |

Программа ch3\_p1\_son3.cpp:

|  |
| --- |
| #include <unistd.h>  #include <signal.h>  #include <stdio.h>  typedef void (\*sighandler\_t)(int);  void signalHandler(int i) {  printf("Some message\n");  }  int main() {  printf("Son3 params: pid=%d, ppid=%d\n",getpid(),getppid());  if (signal(SIGUSR1, (sighandler\_t)signalHandler) == SIG\_ERR) // перехват сигнала  printf("Signal error occured\n");  sleep(10);  } |

Результат выполнения ch3\_p1\_father.cpp:

|  |
| --- |
| ana@AnaLinux ~/lab3 $ g++ -o ch3\_p1\_son1 ch3\_p1\_son1.cpp  ana@AnaLinux ~/lab3 $ g++ -o ch3\_p1\_son2 ch3\_p1\_son2.cpp  ana@AnaLinux ~/lab3 $ g++ -o ch3\_p1\_son3 ch3\_p1\_son3.cpp  ana@AnaLinux ~/lab3 $ g++ -o ch3\_p1\_father ch3\_p1\_father.cpp  ana@AnaLinux ~/lab3 $ ./ch3\_p1\_father  Father params: pid=18622, ppid=1950  Son3 params: pid=18625, ppid=18622  Son2 params: pid=18624, ppid=18622  Son1 params: pid=18623, ppid=18622  Some message |

Содержимое log1:

|  |
| --- |
| UID PID PENDING BLOCKED IGNORED CAUGHT STAT TTY TIME COMMAND  1000 1950 0000000000000000 0000000000010000 0000000000380004 000000004b817efb Ss pts/0 0:00 bash  1000 18622 0000000000000000 0000000000010000 0000000000000006 0000000000000000 S+ pts/0 0:00 ./ch3\_p1\_father  1000 18623 0000000000000000 0000000000000000 0000000000000000 0000000000000000 S+ pts/0 0:00 son1  1000 18624 0000000000000000 0000000000000000 0000000000000200 0000000000000000 S+ pts/0 0:00 son2  1000 18625 0000000000000000 0000000000000000 0000000000000000 0000000000000200 S+ pts/0 0:00 son3  1000 18626 0000000000000000 0000000000000000 0000000000000000 0000000000010002 S+ pts/0 0:00 sh -c ps -s>log1  1000 18627 0000000000000000 0000000000000000 0000000000000000 00000001f3d1fef9 R+ pts/0 0:00 ps –s |

Содержимое log2:

|  |
| --- |
| UID PID PENDING BLOCKED IGNORED CAUGHT STAT TTY TIME COMMAND  1000 1950 0000000000000000 0000000000010000 0000000000380004 000000004b817efb Ss pts/0 0:00 bash  1000 18622 0000000000010000 0000000000010000 0000000000000006 0000000000000000 S+ pts/0 0:00 ./ch3\_p1\_father  1000 18623 0000000000000200 0000000000000000 0000000000000000 0000000000000000 Z+ pts/0 0:00 [ch3\_p1\_son1] <defunct>  1000 18624 0000000000000000 0000000000000000 0000000000000200 0000000000000000 S+ pts/0 0:00 son2  1000 18625 0000000000000000 0000000000000000 0000000000000000 0000000000000200 S+ pts/0 0:00 son3  1000 18628 0000000000000000 0000000000000000 0000000000000000 0000000000010002 S+ pts/0 0:00 sh -c ps -s>log2  1000 18629 0000000000000000 0000000000000000 0000000000000000 00000001f3d1fef9 R+ pts/0 0:00 ps –s |

Содержимое log3:

|  |
| --- |
| UID PID PENDING BLOCKED IGNORED CAUGHT STAT TTY TIME COMMAND  1000 1950 0000000000000000 0000000000010000 0000000000380004 000000004b817efb Ss pts/0 0:00 bash  1000 18622 0000000000000000 0000000000010000 0000000000000006 0000000000000000 S+ pts/0 0:00 ./ch3\_p1\_father  1000 18623 0000000000000200 0000000000000000 0000000000000000 0000000000000000 Z+ pts/0 0:00 [ch3\_p1\_son1] <defunct>  1000 18624 0000000000000000 0000000000000000 0000000000000200 0000000000000000 S+ pts/0 0:00 son2  1000 18625 0000000000000000 0000000000000000 0000000000000000 0000000000000200 S+ pts/0 0:00 son3  1000 18631 0000000000000000 0000000000000000 0000000000000000 0000000000010002 S+ pts/0 0:00 sh -c ps -s>log3  1000 18632 0000000000000000 0000000000000000 0000000000000000 00000001f3d1fef9 R+ pts/0 0:00 ps –s |

Содержимое log4:

|  |
| --- |
| UID PID PENDING BLOCKED IGNORED CAUGHT STAT TTY TIME COMMAND  1000 1950 0000000000000000 0000000000010000 0000000000380004 000000004b817efb Ss pts/0 0:00 bash  1000 18622 0000000000010000 0000000000010000 0000000000000006 0000000000000000 S+ pts/0 0:00 ./ch3\_p1\_father  1000 18623 0000000000000200 0000000000000000 0000000000000000 0000000000000000 Z+ pts/0 0:00 [ch3\_p1\_son1] <defunct>  1000 18624 0000000000000000 0000000000000000 0000000000000200 0000000000000000 S+ pts/0 0:00 son2  1000 18625 0000000000000000 0000000000000000 0000000000000000 0000000000000200 Z+ pts/0 0:00 [ch3\_p1\_son3] <defunct>  1000 18633 0000000000000000 0000000000000000 0000000000000000 0000000000010002 S+ pts/0 0:00 sh -c ps -s>log4  1000 18634 0000000000000000 0000000000000000 0000000000000000 00000001f3d1fef9 R+ pts/0 0:00 ps –s |

В log1 все три процесса-потомка запущены.

В log2 son1 завершает работу и становится зомби-процессом (действие по умолчанию)

В log3 son2 игнорирует сигнал и продолжает свою работу.

В log4 son3 вызывает обработчик сигнала, печатается сообщение, после чего son3 завершает работу и также становится зомби-процессом.

Рассмотрим подробнее столбцы таблицы из log4. C помощью системного вызова ps -s процессом-родителем были выведены зомби-процессы.В данном листинге присутствует графа STAT, она может иметь следующие значения:

* R — выполнимый процесс, ожидающий только момента, когда планировщик задач выделит ему очередной квант времени
* S — процесс "спит"
* D — процесс находится в состоянии подкачки на диске
* T — остановленный процесс
* Z — процесс-зомби
* X — мертвый процесс

Рядом с указателем статуса могут стоять дополнительные символы из следующего набора:

* W — процесс не имеет резидентных страниц
* < — высоко-приоритетеный процесс
* N — низко-приоритетный процесс
* L — процесс имеет страницы, заблокированные в памяти
* + — находится вначале группы процессов
* s — лидер сессии

Дополнительно была изменена команда ps -s на ps l. log4\_1:

|  |
| --- |
| F UID PID PPID PRI NI VSZ RSS WCHAN STAT TTY TIME COMMAND  0 1000 1950 1946 20 0 8368 4684 wait Ss pts/0 0:00 bash  0 1000 18778 1950 20 0 2068 512 signal T pts/0 0:00 ./ch3\_p3  0 1000 19931 1950 20 0 2200 556 wait S+ pts/0 0:00 ./ch3\_p1\_father  0 1000 19932 19931 20 0 0 0 exit Z+ pts/0 0:00 [ch3\_p1\_son1] <defunct>  0 1000 19933 19931 20 0 2200 544 hrtime S+ pts/0 0:00 son2  0 1000 19934 19931 20 0 0 0 exit Z+ pts/0 0:00 [ch3\_p1\_son3] <defunct>  0 1000 19943 19931 20 0 2368 564 wait S+ pts/0 0:00 sh -c ps l>log4\_1  0 1000 19944 19943 20 0 8716 1396 - R+ pts/0 0:00 ps l |

Появились новые графы:

* PRI — приоритет планирования;
* NI — значение nice (см. описание команды nice ниже);
* VSZ — виртуальный размер процесса (в килобайтах);
* RSS — размер резидентного набора (количество 1K-страниц в памяти);

Таким образом можно заметить, что виртуальный размер зомби-процессов равен 0.

### 2.3.3. 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.

Создадим для этого две программы father и son. father запускает другую программу son, назначает сигналу SIGUSR1 специальный обработчик сигнала, выводящий уведомление о прекращении работы программы и время, после чего зацикливается. Программа son переводит себя в фоновый режим при помощи функции daemon, назначает сигналу SIGUSR1 специальный обработчик сигнала, аналогичный предыдущему, и так же зацикливается.

Программа ch3\_p2\_father.cpp:

|  |
| --- |
| #include <unistd.h>  #include <signal.h>  #include <stdio.h>  #include <stdlib.h>  typedef void (\*sighandler\_t)(int);  void signalHandler(int i) { // обработчик прерывания  system("echo 'Father killed ' & date +\"%Y-%m-%d %H:%M:%S,%3N\""); // момент прерывания  exit(0);  }  int main(int argc, char \* argv[]) {  printf("Father pid= %d\n", getpid()); // информация о родителе  if (fork() == 0) { // создаем дочерний процесс  execl("ch3\_p2\_son", "son", NULL); // запускаем код сына  } else {  // ждем прерывания  if (signal(SIGUSR1, (sighandler\_t)signalHandler) == SIG\_ERR)  printf("Signal error occured\n");  while(1); // вечный цикл  }  return 0;  } |

Программа ch3\_p2\_son.cpp:

|  |
| --- |
| #include <unistd.h>  #include <signal.h>  #include <stdio.h>  #include <stdlib.h>  typedef void (\*sighandler\_t)(int);  void signalHandler(int i) {  system("echo 'Son killed ' &date +\"%Y-%m-%d %H:%M:%S,%3N\""); // момент прерывания  exit(0);  }  int main() {  daemon(1, 1);  if (signal(SIGUSR1, (sighandler\_t)signalHandler) == SIG\_ERR) {  printf("Signal error occured\n");  }  printf("Son pid= %d\n", getpid());  while(1);  } |

Результаты выполнения ch3\_p2\_father.cpp приведен ниже. Сперва программа father была запущена в первом окне терминала.

|  |
| --- |
| ana@AnaLinux ~/lab3 $ ./ch3\_p2\_father  Father pid= 18717  Son pid= 18719 |

В консоль были выведены pid родительского процесса и процесса-потомка. Далее было открыто второе окно консоли, в котором данным pid были высланы пользовательские сигналы 10(SIGUSR1).

Посылка пользовательских сигналов

|  |
| --- |
| ana@AnaLinux ~ $ date +"%Y-%m-%d %H:%M:%S,%3N" & sudo kill 18719 -s 10 & date +"%Y-%m-%d %H:%M:%S,%3N" & sudo kill 18717 -s 10  2017-12-05 02:50:32,496  2017-12-05 02:50:32,497 |

Результат получения сигналов

|  |
| --- |
| 2017-12-05 02:50:32,570  Son killed  2017-12-05 02:50:32,574  Father killed |

Как видно, по результатам, процессы были убиты спустя 74 и 77 миллисекунд после команды отсылки сигналов.

### 2.3.4. 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. Позаботьтесь об уведомлении о завершении одного из заданий с помощью команды notify. Аргументом команды является номер задания. Верните невыполненные задания в приоритетный режим командой fg. Например: fg Отмените одно из невыполненных заданий.

Были созданы следующие файлы:

* read\_test - данный файл содержит строчку "text from file".
* ch3\_p3.script - скрипт который выводит сообщение в консоль и уведомляет о своем завершении с помощью notify.
* ch3\_p3.cpp - программ которая выводит сообщение в консоль.

Скрипт ch3\_p3.script:

|  |
| --- |
| #!/bin/bash  echo "text from script" & |

Программа ch3\_p3.cpp:

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  int main(){  sleep(10);  printf("text from program\n");  return 0;  } |

Выполнение нескольких утилит:

|  |
| --- |
| ana@AnaLinux ~/lab3 $ g++ -o ch3\_p3 ch3\_p3.cpp  ana@AnaLinux ~/lab3 $ sudo chmod ugo+x ch3\_p3.script  ana@AnaLinux ~/lab3 $ cat read\_test & ./ch3\_p3 & ./ch3\_p3.script &  [1] 18746  [2] 18747  [3] 18748  text from file  text from script  ana@AnaLinux ~/lab3 $ kill -sigstop %2  [1] Готово cat read\_test  [2]+ Остановлено ./ch3\_p3  [3]+ Готово ./ch3\_p3.script  ana@AnaLinux ~/lab3 $ bg %2  [2]+ ./ch3\_p3 &  text from program |

Все задачи выполнились в заданном порядке. Скрипт сообщил с помощью notify о том, что он завершил работу. Что касается программы ch3\_p3, то она была приостановлена, что было выведено в jobs. Далее работы программы ch3\_p3 была вновь возноблена, и она вывела сообщение в консоль.

### 2.3.5. 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). Есть ли разница в приоритетах для системных и пользовательских процессов? Используются ди-приоритеты реального времени? Каков пользовательский приоритет для запуска приложений из shell? Все ответы подкрепляйте экспериментально

Используя команду nice, можно запустить процесс на выполнение с определенным приоритетом от -19 до 20. Меньшее число означает высший приоритет. Большее число означает низший приоритет. Попробуем это сделать: запустим одну и ту же программу, инкрементирующую внутренний счетчик и выводящую его и свой идентификатор.

Программа ch3\_p4\_1.cpp:

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  #include <stdlib.h>  int main(void){  for(int counter=0;counter<1000000;counter++)  if(counter%100000==0)  printf("pid=%d, counter=%d\n",getpid(),counter);  return 0;  } |

Запустим данный счетки с одинаковыми приоритетами.

|  |
| --- |
| ana@AnaLinux ~/lab3 $ nice -10 ./ch3\_p4\_1 & nice -10 ./ch3\_p4\_1  pid=18797, counter=0  pid=18798, counter=0  pid=18798, counter=100000  pid=18797, counter=100000  pid=18797, counter=200000  pid=18797, counter=300000  pid=18798, counter=200000  pid=18798, counter=300000  pid=18798, counter=400000  pid=18797, counter=400000  pid=18797, counter=500000  pid=18797, counter=600000  pid=18798, counter=500000  pid=18798, counter=600000  pid=18798, counter=700000  pid=18797, counter=700000  pid=18797, counter=800000  pid=18797, counter=900000  pid=18798, counter=800000  pid=18798, counter=900000 |

Как видно из результата, процессы конкурировали между собой за право выполнения. Теперь сделаем приоритет одного из процессов больше чем у другого.

|  |
| --- |
| ana@AnaLinux ~/lab3 $ nice -0 ./ch3\_p4\_1 & nice -19 ./ch3\_p4\_1  pid=18800, counter=0  pid=18800, counter=100000  pid=18800, counter=200000  pid=18800, counter=300000  pid=18800, counter=400000  pid=18800, counter=500000  pid=18800, counter=600000  pid=18800, counter=700000  pid=18800, counter=800000  pid=18800, counter=900000  pid=18801, counter=0  pid=18801, counter=100000  pid=18801, counter=200000  pid=18801, counter=300000  pid=18801, counter=400000  pid=18801, counter=500000  pid=18801, counter=600000  pid=18801, counter=700000  pid=18801, counter=800000  pid=18801, counter=900000 |

По результатам видно, что из-за неравности приоритетов, процесс с pid 18800 заканчивает свою работу раньше.

Используя команду getpriority, можно получить текущее значение приоритета для процесса, группы или пользователя. Эта команда принимает на вход два параметра: один для определения категории — пользователь, процесс или группа процессов, второй для определения конкретного элемента из категории. Определить минимальное и максимальное значение для каждого и режимов диспетчеризации можо при помощи функций sched\_get\_priority\_min() и sched\_get\_priority\_max(), принимающих в качестве параметра тип диспетчеризации.

Определим текущую политику планирования, а также возможные диапазоны. Для этого реализуем программу определения возможных процедур планирования ch3\_p4\_2.cpp:

|  |
| --- |
| #include <stdio.h>  #include <iostream>  #include <math.h>  #include <sys/resource.h>  #include <unistd.h>  #include <sched.h>  using namespace std;  int main(int argc, char\* argv[]) {  struct sched\_param shdprm;  cout<<"Диапазоны:"<<endl;  cout<<"SCHED\_FIFO от "<<sched\_get\_priority\_min(SCHED\_FIFO); // минимальный приоритет FIFO  cout<<" до "<<sched\_get\_priority\_max(SCHED\_FIFO)<<endl; // максимальный  cout<<"SCHED\_RR от "<<sched\_get\_priority\_min(SCHED\_RR); // минимальный приоритет RR  cout<<" до "<<sched\_get\_priority\_max(SCHED\_RR)<<endl; // максимальный  cout<<"SCHED\_OTHER от "<<sched\_get\_priority\_min(SCHED\_OTHER); // минимальный приоритет OTHER  cout<<" до "<<sched\_get\_priority\_max(SCHED\_OTHER)<<endl; // максимальный  cout<<"текущая политика: ";  switch(sched\_getscheduler(0)){ // получение текущего алгоритма планирования  case SCHED\_FIFO:  cout<<"SCHED\_FIFO"<<endl;break;  case SCHED\_RR:  cout<<"SCHED\_RR"<<endl;break;  case SCHED\_OTHER:  cout<<"SCHED\_OTHER"<<endl;break;  case -1:  cout<<"error"<<endl;break;  default:  cout<<"Неизвестная политика планирования.";break;  }    if(sched\_getparam(0,&shdprm) == 0){ // приоритет текущего процесса  cout<<"текущий приоритет текущего процесса: "<<shdprm.sched\_priority<<endl;  }else{  cout<<"ошибка приоритета определения"<<endl;  }    return 0;  } |

Выполнение ch3\_p4\_2.cpp:

|  |
| --- |
| ana@AnaLinux ~/lab3 $ ./ch3\_p4\_2  Диапазоны:  SCHED\_FIFO от 1 до 99  SCHED\_RR от 1 до 99  SCHED\_OTHER от 0 до 0  текущая политика: SCHED\_OTHER  текущий приоритет текущего процесса: 0 |

Как видно приоритет запущенного приложения из shell равен 0. Рассмотрим процессы текущего пользователя и пользователя root.

|  |
| --- |
| ana@AnaLinux ~/lab3 $ ps -lu ana --sort ni  F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD  4 S 1000 1353 1 0 80 0 - 1593 ep\_pol ? 00:00:00 systemd  5 S 1000 1361 1353 0 80 0 - 2974 - ? 00:00:00 (sd-pam)  4 S 1000 1387 1075 0 80 0 - 30709 poll\_s ? 00:00:00 cinnamon-sessio  …  ana@AnaLinux ~/lab3 $ ps -lu 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 12 2 0 60 -20 - 0 - ? 00:00:00 netns  1 S 0 13 2 0 60 -20 - 0 - ? 00:00:00 perf  1 S 0 15 2 0 60 -20 - 0 - ? 00:00:00 writeback  1 S 0 18 2 0 60 -20 - 0 - ? 00:00:00 crypto  1 S 0 19 2 0 60 -20 - 0 - ? 00:00:00 kintegrityd  ... |

Процессы были отсортированы во возрастание, но самый большой приоритет у пользователя ana так и остался равен 0, в то время как у пользователся root было достаточно много процессов с приоритетом -20. Это связано с тем, что root необходимо «контролировать» работу системы, для этого используются все возможные приоритеты.

### 2.3.6. Ознакомьтесь с командой nohup(1). Запустите длительный процесс по nohup(1). Завершите сеанс работы. Снова войдите в систему и проверьте таблицу процессов. Поясните результат.

nohup — UNIX-утилита, запускающая указанную команду с игнорированием сигналов потери связи (SIGHUP). Таким образом, команда будет продолжать выполняться в фоновом режиме и после того, как пользователь выйдет из системы. ch3\_p5.cpp:

|  |
| --- |
| int main(int argc, char\* argv[]) {  while(1);  } |

Выполнение ch3\_p5.cpp

|  |
| --- |
| ana@AnaLinux ~/lab3 $ nohup ./ch3\_p5  nohup: ввод игнорируется, вывод добавляется в 'nohup.out'  // Открываем другую консоль и просматриваем процесс  ana@AnaLinux ~ $ ps -x | grep ch3\_p5  18821 pts/0 R+ 0:07 ./ch3\_p5  // Выйдем из системы и снова войдем  ana@AnaLinux ~ $ ps -x | grep ch3\_p5  18821 pts/0 R+ 0:15 ./ch3\_p5 |

### 2.3.7. 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.

С помощью глобальной переменной UID в интерпретаторе командной строки bash возможно определить идентификатор текущего пользователя (просмотр UID и pid).

|  |
| --- |
| ana@AnaLinux ~/lab3 $ echo $UID  1000  // определяем идентификатор текущего пользователя  ana@AnaLinux ~/lab3 $ id -u ana  1000  // определяем идентификатор пользователя root  ana@AnaLinux ~/lab3 $ id -u root  0  // С помощью вызова ps -A демонстрирующего все процессы в системе, можно узнать диапазон значений идентификаоров процессах  ana@AnaLinux ~/lab3 $ ps -A  PID TTY TIME CMD  1 ? 00:00:02 systemd  ...  18953 pts/0 00:00:00 ps |

Системные процессы можно идентифицировать как запущенные от пользователя root.

## Многонитевое функционирование

### 2.4.1. Подготовьте программу, формирующую несколько нитей. Нити для эксперимента могут быть практически идентичны. Каждая нить в цикле: выводит на печать собственное имя и инкрементирует переменную времени, после чего "засыпает"(sleep(5); sleep(1); - для первой и второй нитей соответственно). На экран (в файл) должно выводиться имя нити и количество пятисекундных (для первой) и секундных (для второй) интервалов функционирования каждой нити.

Напишем программу ch4\_p1.cpp, создающую две нити. Обе нити практически идентичны, отличающиеся только длиной паузы.

|  |
| --- |
| #include <pthread.h>  #include <stdio.h>  #include <unistd.h>  void\* thread1(void\* par) { // первый поток  while(1) { // бесконечный цикл  printf("1\n"); // вывод сообщения  sleep(1); // пауза  }  }  void\* thread2(void\* par) { // второй поток  while(1) { // бесконечный цикл  printf("2\n"); // вывод сообщения  sleep(5); // пауза  }  }  int main() {  pthread\_t t1, t2; // инициализация  pthread\_create(&t1, NULL, &thread1, NULL); // создание первого потока  pthread\_create(&t2, NULL, &thread2, NULL); // создание второго потока  pthread\_join(t1, NULL); // ожидание завершение потока t1  pthread\_join(t2, NULL); // ожидание завершение потока t2  } |

Результаты выполнения:

|  |
| --- |
| ana@AnaLinux ~/lab3 $ g++ -pthread ch4\_p1.cpp -o ch4\_p1  ana@AnaLinux ~/lab3 $ ./ch4\_p1  2  1  1  1  1  1  2  1  1  1  1  1  2  1  1  1  1  1  2  1  1  1  1  ^C |

Для компиляции был добавлен ключ -pthread. Как видно из листинга, на 5 сообщений первого потока, приходится 1 сообщение второго потока.

### 2.4.2. 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. Приведите и объясните результат.

Была написана программа ch4\_p2.cpp, которая создает два потока, каждый из которых выводит сообщение в консоль со значением своего pid. Также во время работы, в файл выводится таблица процессов.

|  |
| --- |
| #include <pthread.h>  #include <stdio.h>  #include <unistd.h>  #include <stdlib.h>  void\* thread1(void\* par) { // первый поток  while(1) { // бесконечный цикл  // вывод pid первого потока  printf("Thread 1. pid=%d\n", getpid());  sleep(1); // пауза  }  }  void\* thread2(void\* par) { // второй поток  while(1) { // бесконечный цикл  // вывод pid второго потока  printf("Thread 2. pid=%d\n", getpid());  sleep(5); // пауза  }  }  int main() {  pthread\_t t1, t2; // инициализация  system("ps -axhf > log\_ch4\_p2"); // для анализа выполения нитей по времени  pthread\_create(&t1, NULL, &thread1, NULL); // создание первого потока  system("ps -axhf >> log\_ch4\_p2");  pthread\_create(&t2, NULL, &thread2, NULL); // создание второго потока  system("ps -axhf >> log\_ch4\_p2");  pthread\_join(t1, NULL); // ожидание завершение потока t1  pthread\_join(t2, NULL); // ожидание завершение потока t2  system("ps -axhf >> log\_ch4\_p2");  } |

Во время выполнения данной программы, был открыт второй терминал, в котором была выполнена команда kill. Результаты выполнения:

|  |
| --- |
| ana@AnaLinux ~/lab3 $ g++ -pthread ch4\_p2.cpp -o ch4\_p2  ana@AnaLinux ~/lab3 $ ./ch4\_p2  Thread 1. pid=18985  Thread 2. pid=18985  Thread 1. pid=18985  Thread 1. pid=18985  Thread 1. pid=18985  Thread 1. pid=18985  Thread 1. pid=18985  Thread 2. pid=18985  Thread 1. pid=18985  Thread 1. pid=18985  Thread 1. pid=18985  Thread 1. pid=18985  ...  // Был открыт второй терминал  ana@AnaLinux ~ $ kill 18985  // Возращаемся к первому  ...  Thread 1. pid=18985  Thread 1. pid=18985  Thread 2. pid=18985  Завершено |

Отрывок файла списка процессов во время выполнения программы:

|  |
| --- |
| ...  1950 pts/0 Ss 0:00 \\_ bash  18985 pts/0 S+ 0:00 \\_ ./ch4\_p2  18986 pts/0 S+ 0:00 \\_ sh -c ps -axhf > log\_ch4\_p2  18987 pts/0 R+ 0:00 \\_ ps -axhf  ... |

Как видно разделения процесса на потоки не произошло, оба потока являются одним процессом. Применение команды kill приводит к завершению работы программы.

### 2.4.3. 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, SIGUSR); (t2 - дескриптор второй нити). В остальном программу можно не изменять. Проанализируйте полученные результаты.

Была написана следующая программа ch4\_p3.cpp:

|  |
| --- |
| #include <pthread.h>  #include <stdio.h>  #include <unistd.h>  #include <stdlib.h>  #include <sys/wait.h>  bool working=true;  typedef void (\*sighandler\_t)(int);  void signalHandler(int i) {  printf("Handler\n"); // перехватчик сигнала  working=false; // установка флага остановки  }  void\* thread1(void\* par) { // первая нить  sleep(5); // 5 сек пауза  pthread\_t \*t2 = (pthread\_t \*)par; // ииниц. указатель на вторую нить  pthread\_kill(\*t2, SIGUSR1); // посылаем сигнал на завершение  while(1) {  printf("Thread 1. PID:%i\n", getpid()); // выводим инф. о 1 нити  sleep(1); // каждую секунду  }  }  void\* thread2(void\* par) { // вторая нить  if (signal(SIGUSR1, (sighandler\_t)signalHandler) == SIG\_ERR)  printf("Signal error occured\n");// ожидаем сигнал на завершение  while(working) {  printf("Thread 2. PID:%i\n", getpid());// выводим инф. о 2 нити  sleep(1); // каждые 5 секунд  }  printf("End of work thread 2\n");  return NULL;  }  int main() {  pthread\_t t1, t2; // инициализация  pthread\_create(&t1, NULL, &thread1, (void\*)&t2); // создание 1 нити+ передача указателя на вторую  pthread\_create(&t2, NULL, &thread2, NULL);// создание 2 нити  pthread\_join(t1, NULL);  pthread\_join(t2, NULL);  } |

Результат работы:

|  |
| --- |
| ana@AnaLinux ~/lab3 $ g++ -pthread ch4\_p3.cpp -o ch4\_p3  ana@AnaLinux ~/lab3 $ ./ch4\_p3  Thread 2. PID:19044  Thread 2. PID:19044  Thread 2. PID:19044  Thread 2. PID:19044  Thread 2. PID:19044  Thread 1. PID:19044  Handler  End of work thread 2  Thread 1. PID:19044  Thread 1. PID:19044  Thread 1. PID:19044 |

Была добавлена переменная working, которая отвечала за работу второй нити, спустя 5 секунд первая нить послала сигнал о завершении второй нити. Перехватчик сигнала принял сигнал и изменил значение переменной working, в случае чего работа второй нити была закончена.

### 2.4.4. 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.

Была написана следующая программа ch4\_p4.cpp:

|  |
| --- |
| #include <pthread.h>  #include <stdio.h>  #include <unistd.h>  #include <stdlib.h>  #include <sys/wait.h>  typedef void (\*sighandler\_t)(int);  void signalHandler(int i) {  printf("Handler\n"); // перехватчик сигнала  pthread\_exit(NULL); // завершение нити  }  void\* thread1(void\* par) { // первая нить  sleep(5); // 5 сек пауза  pthread\_t \*t2 = (pthread\_t \*)par; // ииниц. указатель на вторую нить  pthread\_kill(\*t2, SIGUSR1); // посылаем сигнал на завершение  while(1) {  printf("Thread 1. PID:%i\n", getpid()); // выводим инф. о 1 нити  sleep(1); // каждую секунду  }  }  void\* thread2(void\* par) { // вторая нить  if (signal(SIGUSR1, (sighandler\_t)signalHandler) == SIG\_ERR)  printf("Signal error occured\n");// ожидаем сигнал на завершение  while(1) {  printf("Thread 2. PID:%i\n", getpid());// выводим инф. о 2 нити  sleep(1); // каждые 5 секунд  }  }  int main() {  pthread\_t t1, t2; // инициализация  pthread\_create(&t1, NULL, &thread1, (void\*)&t2); // создание 1 нити+ передача указателя на вторую  pthread\_create(&t2, NULL, &thread2, NULL);// создание 2 нити  pthread\_join(t1, NULL);  pthread\_join(t2, NULL);  } |

Результат работы:

|  |
| --- |
| ana@AnaLinux ~/lab3 $ g++ -pthread ch4\_p3.cpp -o ch4\_p3  ana@AnaLinux ~/lab3 $ ./ch4\_p3  Thread 2. PID:19049  Thread 2. PID:19049  Thread 2. PID:19049  Thread 2. PID:19049  Thread 2. PID:19049  Thread 1. PID:19049  Handler  End of work thread 2  Thread 1. PID:19049  Thread 1. PID:19049  Thread 1. PID:19049  Thread 1. PID:19049  Thread 1. PID:19049... |

Вторая нить выведет 5 сообщений, после чего приходит сигнал, срабатывает обработчик — устанавливается флаг прерывания, и работа второй нити прекращается. В отличии от прошлого решения, решение с pthread\_exit(NULL) является более правильным, так как нет нужды указывать какую нить закончить, будет закончена вызвавшая процедура.

### 2.4.5. Перехватите сигнал «CTRL C»

#### 2.4.5.1. Перехватите сигнал «CTRL C» для процесса(однократно)

Была написана следующая программа ch4\_p5\_1.cpp:

|  |
| --- |
| #include <pthread.h>  #include <stdio.h>  #include <signal.h>  #include <unistd.h>  typedef void (\*sighandler\_t)(int);  void \*signalHandler(int i) { // обработчик прерывания  printf("\nCtrl+C pressed\n"); // сообщение  return NULL;  }  int main() {  printf("Press Ctrl+C\n");  if (signal(SIGINT, (sighandler\_t)signalHandler) == SIG\_ERR) {// перехватили сигнал  printf("Signal error occured\n");  }  sleep(10); // задержка  } |

Результат выполнения:

|  |
| --- |
| ana@AnaLinux ~/lab3 $ g++ -pthread ch4\_p5\_1.cpp -o ch4\_p5\_1  ana@AnaLinux ~/lab3 $ ./ch4\_p5\_1  Press Ctrl+C  ^C  Ctrl+C pressed |

Перехватчик CTRL+C был реализован с помощью сигнала SIGINT

#### 2.4.5.2. Перехватите сигнал «CTRL C» для потока(однократно)

Была написана следующая программа ch4\_p5\_2.cpp:

|  |
| --- |
| #include <pthread.h>  #include <stdio.h>  #include <unistd.h>  #include <stdlib.h>  #include <sys/wait.h>  bool working=true;  typedef void (\*sighandler\_t)(int);  void \*signalHandler(int i) { // обработчик прерывания  printf("\nCtrl+C pressed\n");  working=false; // установка флага остановки работы нити  }  void\* thread1(void\* par) {  // перехватчик сигнала  if (signal(SIGINT, (sighandler\_t)signalHandler) == SIG\_ERR) {  printf("Signal error occured\n");  }  while(working) { // тело потока  printf("Thread 1. PID:%i\n", getpid());  sleep(1);  }  }  int main() {  pthread\_t t1; // инициализация  pthread\_create(&t1, NULL, &thread1, NULL); // создание нити  pthread\_join(t1, NULL);  } |

Результат выполения:

|  |
| --- |
| ana@AnaLinux ~/lab3 $ g++ -pthread ch4\_p5\_2.cpp -o ch4\_p5\_2  ana@AnaLinux ~/lab3 $ ./ch4\_p5\_2  Thread 1. PID:19067  Thread 1. PID:19067  Thread 1. PID:19067  Thread 1. PID:19067  ^C  Ctrl+C pressed |

По нажатию клавиш ctrl+c обработчик прерывания изменил значение флага работы нити, после чего нить закончила свою работу

#### 2.4.5.3. Перехватите сигнал «CTRL C» многократно с восстановлением исходного обработчика после нескольких раз срабатывания.

Была написана следующая программа ch4\_p5\_3.cpp:

|  |
| --- |
| #include <pthread.h>  #include <stdio.h>  #include <unistd.h>  #include <stdlib.h>  #include <sys/wait.h>  typedef void (\*sighandler\_t)(int);  int count = 0;  void signalHandler(int i) { // обработчик прерывания  count++; // счетчик  printf("Ctrl+C pressed %i times\n", count);  if (count >= 5)  signal(SIGINT, SIG\_DFL);// изменяем диспозицию сигнала  }  void\* thread1(void\* par) {  if (signal(SIGINT, (sighandler\_t)signalHandler) == SIG\_ERR) // перехватчик сигнала  printf("Signal error occured\n");  // тело потока  while(1) {  printf("Thread 1. PID:%i\n", getpid());  sleep(1);  }  }  int main() {  pthread\_t t1; // объявление переменных  pthread\_create(&t1, NULL, &thread1, NULL); // создание потока  pthread\_join(t1, NULL);  } |

Результат выполнения:

|  |
| --- |
| ana@AnaLinux ~/lab3 $ g++ -pthread ch4\_p5\_3.cpp -o ch4\_p5\_3  ana@AnaLinux ~/lab3 $ ./ch4\_p5\_3  Thread 1. PID:19076  Thread 1. PID:19076  Thread 1. PID:19076  ^CCtrl+C pressed 1 times  Thread 1. PID:19076  Thread 1. PID:19076  ^CCtrl+C pressed 2 times  Thread 1. PID:19076  ^CCtrl+C pressed 3 times  ^CCtrl+C pressed 4 times  Thread 1. PID:19076  ^CCtrl+C pressed 5 times  Thread 1. PID:19076  ^C |

После пятой отправки SIGINT для него назначается обработчик по умолчанию. Поэтому после получения следующего сигнала SIGINT процесс завершается, что и является действием по умолчанию.

#### 2.4.5.4. Проделайте аналогичную работу для переназначения другой комбинации клавиш.

Был взял код из предыдущего пункта, и все записи "SIGINT"были заменены на "SIGQUIT теперь программа реагирует на комбинацию клавиш ctrl+ \

Выполнение ch4\_p5\_4.cpp

|  |
| --- |
| ana@AnaLinux ~/lab3 $ g++ -pthread ch4\_p5\_4.cpp -o ch4\_p5\_4  ana@AnaLinux ~/lab3 $ ./ch4\_p5\_4  Thread 1. PID:19087  ^\Pressed 1 times  Thread 1. PID:19087  Thread 1. PID:19087  ^\Pressed 2 times  Thread 1. PID:19087  ^\Pressed 3 times  ^\Pressed 4 times  Thread 1. PID:19087  ^\Pressed 5 times  ^\Выход |

### 2.4.6. 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. Приведите пример их применения.

С помощью команды kill -l был выведен список существующих сигналов.

|  |
| --- |
| ana@AnaLinux ~/lab3 $ 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 |

Далее приведена таблица с сигналами, их действиями, описанием и типом.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Название | Код | Действие по умолчанию | Описание | Тип |
| SIGHUP | 1 | Завершение | Закрытие терминала | Уведомление |
| SIGINT | 2 | Завершение | Сигнал прерывания (Ctrl-C) с терминала | Управление |
| SIGQUIT | 3 | Завершение с дампом памяти | Сигнал «Quit» с терминала (Ctrl-\) | Управление |
| SIGILL | 4 | Завершение с дампом памяти | Недопустимая инструкция процессора | Исключение |
| SIGTRAP | 5 | Завершение с дампом памяти | Ловушка трассировки или брейкпоинт | Отладка |
| SIGABRT | 6 | Завершение с дампом памяти | Сигнал посылаемый функцией abort() | Управление |
| SIGFPE | 8 | Завершение с дампом памяти | Ошибочная арифметическая операция | Исключение |
| SIGKILL | 9 | Завершение | Безусловное завершение | Управление |
| SIGBUS | 10 | Завершение с дампом памяти | Неправильное обращение в физическую память | Исключение |
| SIGSEGV | 11 | Завершение с дампом памяти | Нарушение при обращении в память | Исключение |
| SIGSYS | 12 | Завершение с дампом памяти | Неправильный системный вызов | Исключение |
| SIGPIPE | 13 | Завершение | Запись в разорванное соединение (пайп, сокет) | Уведомление |
| SIGALRM | 14 | Завершение | Сигнал истечения времени, заданного alarm() | Уведомление |
| SIGTERM | 15 | Завершение | Сигнал завершения (сигнал по умолчанию для утилиты kill) | Управление |
| SIGUSR1 | 16 | Завершение | Пользовательский сигнал № 1 | Пользовательский |
| SIGUSR2 | 17 | Завершение | Пользовательский сигнал № 2 | Пользовательский |
| SIGCHLD | 18 | Игнорируется | Дочерний процесс завершен или остановлен | Уведомление |
| SIGTSTP | 20 | Остановка процесса | Сигнал остановки с терминала (Ctrl-Z). | Управление |
| SIGURG | 21 | Игнорируется | На сокете получены срочные данные | Уведомление |
| SIGPOLL | 22 | Завершение | Событие, отслеживаемое poll() | Уведомление |
| SIGSTOP | 23 | Остановка процесса | Остановка выполнения процесса | Управление |
| SIGCONT | 25 | Продолжить выполнение | Продолжить выполнение ранее остановленного процесса | Управление |
| SIGTTIN | 26 | Остановка процесса | Попытка чтения с терминала фоновым процессом | Управление |
| SIGTTOU | 27 | Остановка процесса | Попытка записи на терминал фоновым процессом | Управление |
| SIGVTALRM | 28 | Завершение | Истечение «виртуального таймера» | Уведомление |
| SIGPROF | 29 | Завершение | Истечение таймера профилирования | Отладка |
| SIGXCPU | 30 | Завершение с дампом памяти | Процесс превысил лимит процессорного времени | Исключение |
| SIGXFSZ | 31 | Завершение с дампом памяти | Процесс превысил допустимый размер файла | Исключение |

Сигналы с 32 по 64 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.

### 2.4.7. Проанализируйте процедуру планирования для процессов и потоков одного процесса.

#### Обоснуйте результат экспериментально.

#### Попробуйте изменить процедуру планирования. Подтвердите экспериментально, если изменение возможно

#### Задайте нитям разные приоритеты программно и извне (объясните результат).

Была написана следующая программа ch4\_p7.cpp:

|  |
| --- |
| #include <signal.h>  #include <pthread.h>  #include <stdio.h>  #include <sys/types.h>  #include <linux/unistd.h>  #include <sys/syscall.h>  #include <sched.h>  #include <unistd.h>  #include <math.h>  #include <syslog.h>  #include "onecore.h"  pthread\_t t1, t2, t3;  void delay(){ // дилей для замедления работы потоков  int sum=0;  for(int k = 0; k< 1000000000; k++){  sum = sum + k;  }  }  void prnt\_policy(int thread\_number){ // функция для вывода  switch(sched\_getscheduler(0)){ // узнаем какая процедура  case SCHED\_FIFO: // планирования  printf("Thread %d, policy=SCHED\_FIFO\n", thread\_number);  break;  case SCHED\_RR:  printf("Thread %d, policy=SCHED\_RR\n", thread\_number);  break;  case SCHED\_OTHER:  printf("Thread %d, policy=SCHED\_OTHER\n", thread\_number);  break;  default:  break;  }    }  void\* thread1(void \*ptr){ // поток нити номер один  prnt\_policy(1); // вывод процедуры планирования  printf("Thread\_1, pid=%d is started \n",getpid()); // сообщение  while(true){ // бесконечный цикл  delay();  prnt\_policy(1);;  }  }    void\* thread2(void \*ptr){ // поток нити номер два  prnt\_policy(2); // вывод процедуры планирования  printf("Thread\_2, pid=%d is started \n",getpid()); // сообщение  while(true){ // бесконечный цикл  delay();  prnt\_policy(2);  }  }  void\* thread3(void \*ptr){ // поток нити номер три  prnt\_policy(3); // вывод процедуры планирования  printf("Thread\_3, pid=%d is started \n",getpid()); // сообщение  while(true){ // бесконечный цикл  delay();  prnt\_policy(3);  }  }  int main(void){  //one\_core(getpid());  prnt\_policy(0);  struct sched\_param param; // деклорировани структуры  pthread\_attr\_t attr\_1, attr\_2, attr\_3; // декларировани аттрибутов      pthread\_attr\_init(&attr\_1); // инициализация атрибута 1 потока  pthread\_attr\_init(&attr\_2); // инициализация атрибута 2 потока  pthread\_attr\_init(&attr\_3); // инициализация атрибута 3 потока  pthread\_attr\_setschedpolicy(&attr\_1, SCHED\_FIFO);// устанавливаем политку для 1 потока  pthread\_attr\_setschedpolicy(&attr\_2, SCHED\_RR);// устанавливаем политку для 2 потока  pthread\_attr\_setschedpolicy(&attr\_3, SCHED\_FIFO);// устанавливаем политку для 3 потока      param.sched\_priority = 1; // устанавливаем приоритет для 1 потока  pthread\_attr\_setschedparam(&attr\_1, &param);  param.sched\_priority = 99; // устанавливаем приоритет для 2 потока  pthread\_attr\_setschedparam(&attr\_2, &param);  param.sched\_priority = 1; // устанавливаем приоритет для 3 потока  pthread\_attr\_setschedparam(&attr\_3, &param);      pthread\_attr\_setinheritsched(&attr\_1, PTHREAD\_EXPLICIT\_SCHED); // установить приоритеты из объекта attr\_1  pthread\_attr\_setinheritsched(&attr\_2, PTHREAD\_EXPLICIT\_SCHED);// установить приоритеты из объекта attr\_2  pthread\_attr\_setinheritsched(&attr\_3, PTHREAD\_EXPLICIT\_SCHED);// установить приоритеты из объекта attr\_2  // установка атрибута наследования от родителя  pthread\_attr\_setinheritsched(&attr\_1, PTHREAD\_INHERIT\_SCHED);    pthread\_create(&t1,&attr\_1,thread1,NULL); // создание первого потока  pthread\_create(&t2,&attr\_2,thread2,NULL); // создание второго потока  pthread\_create(&t3,&attr\_3,thread3,NULL); // создание третьего потока    pthread\_join(t1, NULL); // ожидание завершения работы нитей  pthread\_join(t2, NULL);  pthread\_join(t3, NULL);    pthread\_attr\_destroy(&attr\_1); // удаляем аттрибуты  pthread\_attr\_destroy(&attr\_2);  pthread\_attr\_destroy(&attr\_3);  } |

Данная программа создает три нити, каждая из которых со своей процедурой планирования и своим приоритетом. Каждая из нитей выводит сообщение о своей процедуре планирования в консоль, а затем производит некоторые математические вычисления для осуществления задержки.

Далее приведен листинг выполнения, согласно настройкам вышенаписанной программы.

|  |
| --- |
| ana@AnaLinux ~/lab3 $ g++ -pthread ch4\_p7.cpp -o ch4\_p7  ana@AnaLinux ~/lab3 $ sudo ./ch4\_p7  Thread 0, policy=SCHED\_OTHER  Thread 1, policy=SCHED\_OTHER  Thread\_1, pid=1181 is started  Thread 3, policy=SCHED\_FIFO  Thread\_3, pid=1181 is started  Thread 2, policy=SCHED\_RR  Thread\_2, pid=1181 is started  Thread 3, policy=SCHED\_FIFO  Thread 1, policy=SCHED\_OTHER  Thread 2, policy=SCHED\_RR  Thread 3, policy=SCHED\_FIFO  Thread 2, policy=SCHED\_RR  Thread 1, policy=SCHED\_OTHER  Thread 3, policy=SCHED\_FIFO  Thread 3, policy=SCHED\_FIFO  Thread 2, policy=SCHED\_RR  Thread 1, policy=SCHED\_OTHER  Thread 3, policy=SCHED\_FIFO  Thread 2, policy=SCHED\_RR  Thread 1, policy=SCHED\_OTHER  Thread 3, policy=SCHED\_FIFO |

Первоначально процедура планирования для первой нити была SCHED\_FIFO, но далее эта процедура была переприсвоена путем её наследования от родителя. В случае если и вовсе нигде не указывать процеду планирования, то она будет наследоваться от родительского процесса (по умолчанию SCHED\_OTHER).

Как и требовалось ожидать, третья нить опережает другии нити по количеству выполнения, из-за более высокого приоритета выполнения.

Изменить политику планирования можно только обладая правами root. Изменить приоритет ветвей извне невозможно, так как все они имеют одинаковый идентификатор процесса, требующийся при изменении приоритета.

### 2.4.8. Создайте командный файл (скрипт), выполняющий Вашу лабораторную работу автоматически при наличии необходимых С-файлов.

Скрипт ch4\_p8.script:

|  |
| --- |
| #!/bin/bash  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_CHAPTER I\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch1\_p1.cpp"  g++ ch1\_p1.cpp -o ch1\_p1  echo 'executing ch1\_p1'  ./ch1\_p1  sleep 1  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch1\_p2.cpp"  g++ ch1\_p2.cpp -o ch1\_p2  echo 'executing ch1\_p2'  ./ch1\_p2  sleep 1  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch1\_p3.cpp"  g++ ch1\_p3.cpp one\_core.cpp -o ch1\_p3  echo 'executing ch1\_p3'  ./ch1\_p3  sleep 1  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch1\_p4.cpp"  g++ ch1\_p4.cpp one\_core.cpp -o ch1\_p4  echo 'executing ch1\_p4'  echo 'policy=SCHED\_FIFO, p0=99, p1=1, p2=50, p3=99'  ./ch1\_p4 1 99 1 50 99  sleep 1  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling father.c"  g++ father.c -o father  echo "Compiling son.c"  g++ son.c -o son  echo 'executing father'  ./father  sleep 2  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch1\_p8.cpp"  g++ ch1\_p8.cpp -o ch1\_p8  echo 'executing ch1\_p8'  ./ch1\_p8 5  sleep 1  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling wait.cpp"  g++ wait.cpp -o wait  echo 'executing wait'  ./wait  sleep 1  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling son.c"  g++ son.c -o son  echo "Compiling daughter.c"  g++ daughter.c -o daughter  echo "Compiling wife.c"  g++ wife.c -o wife  echo "Compiling waitpid.cpp"  g++ waitpid.cpp -o waitpid  echo 'executing waitpid'  ./waitpid  sleep 4  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch1\_p10.cpp"  g++ ch1\_p10.cpp one\_core.cpp -o ch1\_p10  echo 'executing ch1\_p10'  echo 'policy=SCHED\_FIFO, p0=99, p1=1, p2=50, p3=99'  ./ch1\_p10 1 99 1 50 99  sleep 2  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch1\_p10\_4.cpp"  g++ ch1\_p10\_4.cpp one\_core.cpp -o ch1\_p10\_4  echo 'executing ch1\_p10\_4'  ./ch1\_p10\_4 50 50 50 50  sleep 2  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch1\_p11.cpp"  g++ ch1\_p11.cpp -o ch1\_p11  echo 'executing ch1\_p11'  ./ch1\_p11  sleep 1  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch1\_p11\_2.cpp"  g++ ch1\_p11\_2.cpp -o ch1\_p11\_2  echo 'executing ch1\_p11\_2'  ./ch1\_p11\_2  sleep 1  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch1\_p12\_father.cpp"  g++ ch1\_p12\_father.cpp one\_core.cpp -o ch1\_p12\_father  echo "Compiling ch1\_p12\_daughter.cpp"  g++ ch1\_p12\_daughter.cpp -o ch1\_p12\_daughter  echo 'executing ch1\_p12\_father'  ./ch1\_p12\_father  sleep 1  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_CHAPTER II\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch2\_p1\_a\_son.cpp"  g++ ch2\_p1\_a\_son.cpp -o ch2\_p1\_a\_son  echo "Compiling ch2\_p1\_a\_father.cpp"  g++ ch2\_p1\_a\_father.cpp -o ch2\_p1\_a\_father  echo 'executing ch2\_p1\_a\_father'  ./ch2\_p1\_a\_father  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch2\_p1\_b\_father.cpp"  g++ ch2\_p1\_b\_father.cpp -o ch2\_p1\_b\_father  echo 'executing ch2\_p1\_b\_father'  ./ch2\_p1\_b\_father  sleep 6  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch2\_p1\_v\_father.cpp"  g++ ch2\_p1\_v\_father.cpp -o ch2\_p1\_v\_father  echo 'executing ch2\_p1\_v\_father'  ./ch2\_p1\_v\_father  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch2\_p1\_v\_father.cpp"  g++ ch2\_p1\_v\_father.cpp -o ch2\_p1\_v\_father  echo 'executing ch2\_p1\_v\_father'  ./ch2\_p1\_v\_father  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_CHAPTER III\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch3\_p1\_son1.cpp"  g++ ch3\_p1\_son1.cpp -o ch3\_p1\_son1  echo "Compiling ch3\_p1\_son2.cpp"  g++ ch3\_p1\_son2.cpp -o ch3\_p1\_son2  echo "Compiling ch3\_p1\_son3.cpp"  g++ ch3\_p1\_son3.cpp -o ch3\_p1\_son3  echo "Compiling ch3\_p1\_father.cpp"  g++ ch3\_p1\_father.cpp -o ch3\_p1\_father  echo 'executing ch3\_p1\_father'  ./ch3\_p1\_father  sleep 1  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch3\_p3.cpp"  g++ ch3\_p3.cpp -o ch3\_p3  cat read\_test & ./ch3\_p3 & ./ch3\_p3.script &  sleep 11  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch3\_p4\_1.cpp"  g++ ch3\_p4\_1.cpp -o ch3\_p4\_1  nice -10 ./ch3\_p4\_1 & nice -10 ./ch3\_p4\_1  sleep 1  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch3\_p4\_2.cpp"  g++ ch3\_p4\_2.cpp -o ch3\_p4\_2  ./ch3\_p4\_2  sleep 1  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_CHAPTER IV\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch4\_p1.cpp"  echo 'executing ch4\_p1'  g++ ch4\_p1.cpp -pthread -o ch4\_p1  timeout 5s ./ch4\_p1  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch4\_p3.cpp"  echo 'executing ch4\_p3'  g++ ch4\_p3.cpp -pthread -o ch4\_p3  timeout 10s ./ch4\_p3  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch4\_p4.cpp"  echo 'executing ch4\_p4'  g++ ch4\_p4.cpp -pthread -o ch4\_p4  timeout 10s ./ch4\_p4  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch4\_p5\_1.cpp"  echo 'executing ch4\_p5\_1'  g++ ch4\_p5\_1.cpp -o ch4\_p5\_1  ./ch4\_p5\_1 &  sleep 1  pid=$(echo $!)  kill -s sigint $pid  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch4\_p5\_2.cpp"  echo 'executing ch4\_p5\_2'  g++ ch4\_p5\_2.cpp -pthread -o ch4\_p5\_2  ./ch4\_p5\_2 &  sleep 1  pid=$(echo $!)  kill -s sigint $pid  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch4\_p5\_3.cpp"  echo 'executing ch4\_p5\_3'  g++ ch4\_p5\_3.cpp -pthread -o ch4\_p5\_3  ./ch4\_p5\_3 &  pid=$(echo $!)  sleep 1  kill -s sigint $pid  sleep 1  kill -s sigint $pid  sleep 1  kill -s sigint $pid  sleep 1  kill -s sigint $pid  sleep 1  kill -s sigint $pid  sleep 1  kill -s sigint $pid  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch4\_p5\_4.cpp"  echo 'executing ch4\_p5\_4'  g++ ch4\_p5\_4.cpp -pthread -o ch4\_p5\_4  ./ch4\_p5\_4 &  pid=$(echo $!)  sleep 1  kill -s sigquit $pid  sleep 1  kill -s sigquit $pid  sleep 1  kill -s sigquit $pid  sleep 1  kill -s sigquit $pid  sleep 1  kill -s sigquit $pid  sleep 1  kill -s sigquit $pid  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'  echo "Compiling ch4\_p7.cpp"  echo 'executing ch4\_p7'  g++ ch4\_p7.cpp -pthread -o ch4\_p7  timeout 10s ./ch4\_p7  echo '\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_' |

Вышенаписанный скрипт компилирует и запускает программы, с которыми ранее были произведены эксперименты.

# 3. Вывод

Порождение процесса-потомка можно осуществить с помощью функции fork(). Для каждого из процессов будут доступны все структуры данных находящиеся в программе, однако каждый из процессов будет работать лишь со своей копией данных. Это позволяет организовать распараллеливание процессов. При этом, узнать о процессе-родителе можно по параметру PPID, который равен идентификатору процесса-родителя PID.

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. Важно отметить, что при выполнении fork и exec наследуются файловые дескрипторы, что позволяет считать в процессе-потомке из файла, открытого в процессе-родителе.

Изменив процедуру планирования выполнения процессов, можно уравнять программы в правах на процессорный ресурс. А, изменив приоритет процесса, можно изменить порядок выполнения и дать преимущество в борьбе за процессорный ресурс. Границы приоритетов процесса зависят от выбранной процедуры планирования. Так же возможно устанавливать различные процедуры планирования для процессов-потомков, а также устанавливать одинаковый приоритет при разных методах планирования. Важно заметить, что чтобы увидеть результат данных действий необходимо сделать ограничение выполнения программы на одно ядро процессора, иначе полученные результаты будет затруднительно объяснить.

Процесс, породивший другой процесс, имеет возможность узнать о его завершении, с помощью функции wait(), а так же узнать результат, который вернул порожденный процесс. Это дает возможность контролировать и ожидать выполнения порожденных процесств. Если не ждать завершения процесса-потомка, то значение PPID станет равным 1 для всех процессов- потомков. Это корневой процесс init который запускается ядром при загрзуке системы. Одной из его задач является "усыновлять"процессы которые остались без родительского процесса.

В системе Linux существуют сигналы, который 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. Это позволяет процессам реагировать на изменения в системе необходимым образом.

Также для распараллеливания вычислений существует многонитевое программирование. Нити могут создаваться и отслеживаться в основном процессе, им могут быть переданы параметры, назначены обработчики сигналов, а также изменена процедура планирования и приоритеты. Отличие нитей от процессов состоит в том, что у нитей PID одинаковый для всех, а значит, влиять на нить извне становится проблематично.

# 4. Список литературы

1. Программа с псевдо-распараллеливанием. Источник: «Практические вопросы разработ- ки системных приложений» Душутина Е.В. 62