**Содержание**

[**1. Цели работы** 2](#_Toc86611982)

[**2. Задачи** 2](#_Toc86611983)

[**3. Ход выполнения работы** 2](#_Toc86611984)

[**4. Вывод** 16](#_Toc86611985)

# **Лабораторная работа №2 – Работа с процессами в OS Linux**

# **1. Цели работы**

Знакомство с характерной для Linux схемой порождения и завершения процессов, с отношениями типа потомок-родитель, со способами передачи информации о событии завершения процесса.

**2. Задачи**

1. Войти в систему и скопировать в свой HOME-каталог с разделяемого ресурса набор исходных файлов для второй лабораторной работы.

2. Скомпилировать и выполнить примеры программ forkdemo.cpp, tinymenu.cpp, tinyexit.cpp, procgroup.cpp, wait\_parent.cpp.

3. Модифицировать программу forkdemo.cpp так, чтобы ввод/вывод на терминал отсутствовал, а при переходе по циклу была временная задержка. Запустить модифицированную программу.

4. Исследовать, что произойдет, если процесс-потомок сменит текущий каталог: проверить, будет ли изменен текущий каталог для родителя. Создать программу, подтверждающую ответ и привести её в отчете.

5. Проиллюстрировать как процесс-родитель и процесс потомок разделяют один и тот же дескриптор и смещение текстового файла.

**3. Ход выполнения работы**

1. Войду в систему и скопирую в свой HOME-каталог с разделяемого ресурса набор исходных файлов для второй лабораторной работы. Содержимое файлов приведено в листингах 2.1 – 2.6, соответственно.

|  |
| --- |
| Листинг 2.1 Содержимое файла forkdemo.cpp |
| #include<stdio.h>  #include<sys/types.h>  #include<unistd.h>  main()  {  int i;  if (fork()) { /\* I must be the parent \*/  for(i=0; i<1000 ; i++)  printf("\t\tPARENT %d\n", i);  }  else { /\* I must be the child \*/  for(i=0; i<1000 ; i++)  printf("CHILD %d\n",i);  }  } |

|  |
| --- |
| Листинг 2.2 Содержимое файла tinymenu.cpp |
| /\* The program tinymenu.cpp \*/  #include<stdio.h>  #include<unistd.h>  main()  {  /\* Hard-wired list of commands \*/  static char \*cmd[]={(char \*)"who",(char \*)"ls",(char \*)"date"};  int i;  /\* Prompt for read a command number \*/  printf("0=who, 1=ls, 2=date:");  scanf("%d",&i);  /\* Execute the selected command \*/  execlp(cmd[i], cmd[i], (char \*)0);  printf("Command not found\n");  /\* exec failed \*/  } |

|  |
| --- |
| Листинг 2.3 Содержимое файла tinyexit.cpp |
| /\* The program tinyexit.cpp \*/  #include<stdio.h>  #include<unistd.h>  #include<stdlib.h>  #include<sys/types.h>  #include<sys/wait.h>  main()  {  /\* Hard-wired list of commands \*/  static char \*cmmd[3];  cmmd[0]=(char \*)"who";  cmmd[1]=(char \*)"ls";  cmmd[2]=(char \*)"date";  int i;  while(1){  /\* Prompt for read a command number \*/  printf("0=who, 1=ls, 2=date:");  scanf("%d",&i);  /\* If selection is invalid, parent terminates \*/  if(i<0 || i>2)  exit(2);  if (fork()==0){ /\* Child \*/  /\* The child executes the selected command \*/  execlp(cmmd[i], cmmd[i], NULL);  printf("Command not found\n");  /\* exec failed \*/  exit(1);  }  else  { /\* The parent waits for the child to finish \*/  wait(0);  }  }  } |

|  |
| --- |
| Листинг 2.4 Содержимое файла procgroup.cpp |
| /\*  \* Displaying process group ID information  \*/  #include <stdio.h>  #include<sys/types.h>  #include<unistd.h>  main(){  int i;  printf("\n\nInitial process \t PID %6d \t PPID %6d \t GID %6d\n\n",getpid(), getppid(), getpgid(0));  for (i=0; i<3 ; ++i)  if (fork() == 0) /\* Generate some processes \*/  printf("New process \t\t PID %6d \t PPID %6d \t GID %6d\n", getpid(), getppid(), getpgid(0));  } |

|  |
| --- |
| Листинг 2.5 Содержимое файла wait\_child.cpp |
| /\* The program wait\_child.cpp \*/  #include<stdio.h>  #include<unistd.h>  #include<stdlib.h>  #include<sys/types.h>  #include<signal.h>  main(int argc, char \*argv[])  {  pid\_t pid;  int ret\_value;  pid = getpid();  ret\_value = (int) (pid % 256);  srand((unsigned) pid);  sleep(rand() %5);  if(atoi(\*(argv+1)) % 2){  /\* assuming argv[1] exists! \*/  printf("Child %d is terminating with signal 0009\n", pid);  kill(pid, 9); /\* hara-kiri \*/  }  else{  printf("Child %d is terminating with exit (%04X)\n", pid, ret\_value);  exit(ret\_value);  }  } |

|  |
| --- |
| Листинг 2.6 Содержимое файла wait\_parent.cpp |
| /\* The program wait\_parent.cpp \*/  /\* A parent process waits for a child to finish \*/  #include<stdio.h>  #include<unistd.h>  #include<stdlib.h>  #include<sys/types.h>  #include<sys/wait.h>  main()  {  pid\_t pid, w;  int i, status;  char value[3]; /\*place to store index as string \*/  for(i=0; i<3; ++i)  { /\* Gener. 3 child processes \*/  if ((pid=fork())==0){  sprintf(value, "%d", i);  execl("wait\_child", "wait\_child", value, (char \*)0);  }  else /\* assuming no falures here ...\*/  printf("Forked child %d\n", pid);  }  /\* Wait for the children \*/  while((w=wait(&status)) && w!=-1){  if(w!=-1)  printf("Wait on PID: %d returns status of: %04X\n", w, status);  }  exit(0);  } |

2. Скомпилирую и выполню примеры программ. Начну с forkdemo.cpp с помощью опции g++. Полученный результат программы forkdemo.cpp записывается в файл a.out. Выведу его в терминал. Вывод файла a.out приведен на рисунке 2.1-2.

|  |  |
| --- | --- |
| Рис. 2.1 – Начало a.out | Рис. 2.2 – Фрагмент 1 a.out |
| Рис. 2.3 – Фрагмент 2 a.out | Рис. 2.4 – Фрагмент 3 a.out |
| Рис. 2.5 – Фрагмент 4 a.out | Рис. 2.5 – Конец файла a.out |

Как видно из полученных данных присутствует конкурентность и асинхронность процессов, так как во фрагментах 2, 3, 4, приведенных на рисунках 2.2, 2.3, 2.4 соответственно, наблюдается переключение вывода от процесса к процессу. Важно уточнить, хоть и известен код программы, но невозможно рассчитать то, что будет подано на вывод в определенный момент времени, так как переключение зависит от буферизации, от характеристик компьютера и от кванта времени, который выделяется планировщиком определенному процессу. В коде программы производится вызов fork(), который порождает новый процесс (процесс-потомок), который почти идентичен порождающему процессу-родителю, данный процесс потомок наследует множество параметров, например, сегменты кода, данных и стека программы, таблицу файлов, рабочий и корневой каталоги, реальный и эффективный номер пользователя и номер группы, и другие. Важно отметить, что процесс потомок не наследует от родителя идентификатор процесса (PID, PPID), израсходованное время ЦП (оно обнуляется), сигналов процесса родителя, требующих ответа, блокированных файлов. После вызова возникают два полностью идентичных процесса, весь код после вызова метода fork(), выполняется дважды. В коде есть два цикла с условием for(fork()) {}, так как fork() может возвращать три параметра: -1 – при вызове fork() возникла ошибка, 0 – это код процесса-потомка, 1 – код процесса-родителя.

Продолжу процесс компиляции файл tinymenu.cpp. Выполню файл a.out и при этом выведу результат в терминал. Полученные данные в терминале приведены на рисунке 2.6. В программе tinymenu используется вызов системного метода exec(), который позволяет передавать параметры командной строки, но в то же время переменные окружения будут наследоваться, их может быть несколько, по соглашению первый параметр должен указывать на имя, ассоциируемое с файлом, который необходимо исполнить. С помощью вызова exelp() в примере запускаются команды интерпретатора. Содержимое терминала при введении параметров равных 0, 1, 2 представлено на рисунке 2.7.

|  |
| --- |
| Рис. 2.6 – Содержимое файла a.out при компиляции tinymenu.cpp |
| Рис. 2.7 – Вывод программы при введении параметров |

Далее произведу компиляцию файла tinyexit.cpp. Так как tinyexit содержит системный процесс wait, то родительский процесс переводится в состояние ожидания, после, если выполняется вызов метода exit(), то происходит выход по событию завершению родительского процесса. Вывод файла a.out приведен на рисунке 2.9. Содержимое терминала представлено на рисунке 2.10 при введении параметров 0, 1, 2, 3.

|  |
| --- |
| Рис. 2.9 – Вывод файла a.out после компиляции tinyexit.cpp |
| Рис. 2.10 – Вывод в терминал при введении параметров |

Далее скомпилирую и выполню программу procgroup.cpp. Данная программа отображает информацию о процессах. Вывод терминала представлен на рисунке 2.11. Вызов метода getpgid() аналогичен методу getpgid(0) – получает группу процессов.

|  |
| --- |
| Рис. 2.11 – Вывод работы программы procgroup.cpp |

Далее скомпилирую и выполню программу wait\_parent.cpp. В данной программе запускается три потомка из родительского процесса и отслеживается завершения каждого из них с помощью вызова системного метода wait(). Вывод в терминал представлен на рисунке 2.12. Как видно также наблюдается информация о том, какой код завершения процесса был передан. В данном примере wait\_child – процесс потомок, которая запускается из wait\_parent при помощи вызова метода exelp c разными параметрами. В процессах wait\_child формируется код завершения. Важно уточнить, что при завершении по сигналу, номер сигнала передается wait\_parent в младшем байте статуса завершения.

|  |
| --- |
| Рис. 2.12 – Вывод в терминал программы wait\_parent.cpp |

Далее скомпилирую и выполню программу wait\_child.cpp. Содержимое вывода wait\_parent представлено на рисунке 2.13. Как видно, что при завершении по сигналу, номер сигнала передается в младшем байте статуса разряда.

|  |
| --- |
| Рис. 2.13 – Вывод выполнения программы wait\_child.cpp и wait\_parent.cpp |

3. Модифицирую программу forkdemo().cpp так, чтобы не было ввода и вывода в терминал, а при проходе по циклу установлю временную задержку sleep(10). Запущу данную программу в фоновом режиме (background), введя при запуске символ & после пробела и зафиксировав значение PID, назначенное системой фоновому процессу при запуске. Выполню в терминале команды ps, top, uptime, pstree. Код модифицированной программы приведен в листинге 2.7.

|  |
| --- |
| Листинг 2.7 – Код программы forkdemo\_m.cpp |
| #include<stdio.h>  #include<sys/types.h>  #include<unistd.h>  main()  {  pid\_t PID;  PID = fork();  int i;  if (PID) { /\* I must be the parent \*/  for(i=0; i<1000 ; i++)  sleep(10);  }  else { /\* I must be the child \*/  for(i=0; i<1000 ; i++)  sleep(10);  }  } |

Запущу программу в фоновом режиме с помощью инструкции, описанной выше. Выполнение программы и результаты представлены на рисунке 2.14. Как можно убедиться программа ничего не выводит и не требует ввести что-либо.

|  |
| --- |
| Рис. 2.14 – Содержимое терминала при выполнении forkdemo\_m.cpp |

Выполню команды ps и ps aux, которые выводят список процессов в терминал. Результаты представлены на рисунках 2.15 и 2.16 соответственно.

|  |
| --- |
| Рис. 2.15 – Выполнение команды ps |
| Рис. 2.16 – Выполнение команды ps aux |

Выполню команду top, которая позволяет наблюдать за процессами в режиме реального времени. Результат выполнения приведен на рисунке 2.17.

|  |
| --- |
| Рис. 2.17 – Выполнение команды top |

Выполню команду uptime, которая позволяет посмотреть на информацию о загрузки системы. Результат выполнения приведен на рисунке 2.18.

|  |
| --- |
| Рис. 2.18 – Выполнение команды uptime |

Выполню команду pstree, чтобы посмотреть все запущенные процессы. Результат выполнения представлен на рисунке 2.19.

|  |
| --- |
| Рис. 2.19 – Выполнение команды pstree |

Сниму собственный процесс с помощью команды kill и параметров процесса, и с помощью команды w можно посмотреть список активных процессов для всех пользователей – можно убедиться, что процесс завершен. Результаты представлены на рисунках 2.20 и 2.21.

|  |
| --- |
| Рис. 2.20 – Выполнение команды kill |
| Рис. 2.21– Выполнение команды w |

4. Исследую ситуацию, когда процесс-потомок меняет текущий каталог. Создам программу, моделирующую данную ситуацию. Код программы представлен в листинге 2.8.

|  |
| --- |
| Листинг 2.8 – Код программы DIR\_change.cpp |
| #include<stdio.h>  #include<sys/types.h>  #include<unistd.h>  main()  {  int i;  if (fork()) { /\*I must be the parent \*/  printf("\n %s \n", get\_current\_dir\_name());  for(i=0; i<20 ; i++)  printf("\t\tPARENT %d\n", i);  sleep(10);  }  else { /\*I must be the child \*/  printf("\n %d \n", chdir("home/alex/OS"));  printf("\n %s \n", get\_current\_dir\_name());  for(i=0; i<20 ; i++)  printf("CHILD %d\n",i);  sleep(10);  }  return 0;  } |

Скомпилирую и запущу программу DIR\_change.cpp. Результаты выполнения представлены на рисунке 2.22. Как видно из полученных результатов программы при корректном перемещении процесса-потомка каталог процесса родителя не изменился, что и было описано выше при описании команды fork(). Поэтому данная операция не влияет на потоки процессов.

|  |
| --- |
| Рис. 2.22 – Результаты выполнения программы DIR\_change.cpp |

5. Необходимо проиллюстрировать, как процесс-родитель и процесс-потомок разделяют один и тот же дескриптор и смещение текстового файла. Чтобы исследовать данную ситуацию, необходимо составить программу, где процесс-родитель открывает некий текстовый файл и запускает потомка, в то же время процесс-потомок читает из данного файла фрагмент данных и выводит их в консоль. По завершению потомка родитель должен читать из того же файла и выводить результат в консоль. Чтобы синхронизировать доступ родителя и потомка к файлу буду использовать метод sleep(). Код программы приведен на листинге 2.9. Содержимое input.txt приведено на рисунке 2.23. Результат выполнения приведен на рисунке 2.24. Как можно наблюдать, программа работает корректно, а это означает, что при определении процесса-потомка ему наследуется таблица файлов, в которой находятся состояния флагов дескрипторов файла, указывающие, читается ли файл или пишется. Кроме того, в таблице файлов содержится текущая позиция указателя записи-чтения.

|  |
| --- |
| Листинг 2.23 – Код программы test\_input.cpp |
| #include <stdio.h>  #include <stdlib.h>  #include <sys/types.h>  #include <sys/wait.h>  #include <unistd.h>  int main() {  FILE \*ptr;  size\_t length = 0;  ssize\_t up;  char\* buf = (char\*)malloc (100\*sizeof(char));  if ((ptr = fopen("input.txt", "r")) == NULL) {  printf("File error occured\n");  exit(1);  }  if (fork() == 0){  if((up = getline(&buf, &length, ptr)) != -1) printf("Child: %s", buf);  exit(0);  } else {  wait(NULL);  if((up = getline(&buf, &length, ptr)) != -1) printf("Parent: %s", buf);  fclose(ptr);  }  free(buf);  return 0;  } |

|  |
| --- |
| Рис. 2.23 – Содержимое файла input.txt |
| Рис. 2.24 – Выполнение программы test\_input.cpp |

**4. Вывод**

В ходе выполнения работы были изучены и испытаны на практике основные методы работы с процессами. Осуществлен запуск, порождение процессов-потомков, вызов методов ожидания, и прерывания процесса. Были изучены основные их свойства, а именно свойства директорий текущих процессов, работа с идентификаторами процессов и файлов