**Лабораторная работа №2 - Запуск и завершение процессов**

**Цель работы**

Знакомство с характерной для Linux схемой порождения и завершения процессов, с отношениями типа потомок – родитель, со способами передачи информации о событии завершения процесса. В данной и в последующих работах понадобится компилировать исходные файлы с помощью строчного компилятора g++. В простейших случаях одномодульных проектов (до работы № 16) применяется команда g++ <имя.cpp файла>, исполняемый файл генерируется только в случае отсутствия ошибок компиляции и по умолчанию именуется a.out. Запускается исполняемый файл с терминала с указанием полного пути (например, /a.out, если он в текущем каталоге) или из Midnight Commander-а. Сведения о полной функциональности g++ можно почерпнуть в справке man.

**Пункт 1**

Войдите в систему и скопируйте в свой HOME-каталог с разделяемого ресурса набор исходных файлов для второй лабораторной работы.

|  |
| --- |
| Листинг 2.1 Код программы 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 Код программы 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 Код программы 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 Код программы 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 Код программы 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);  } |

|  |
| --- |
| Листинг 2.6 Код программы 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);  }  } |

**Пункт 2**

Скомпилируйте и выполните примеры программ forkdemo.cpp, tinymenu.cpp, tinyexit.cpp, procgroup.cpp, wait\_parent.cpp. Процесс wait\_parent при исполнении запускает процесс wait\_child. Программа wait\_child.cpp компилируется с опцией: g++ wait\_child.cpp-o wait\_child. Пояснения к данным программам можно найти в тексте лекций. При необходимости конвертации текстовых файлов из формата DOS в Linux и наоборот используйте команды dos2unix и unix2dos.

1. Скомпилируем и выполним программу forkdemo.cpp

|  |  |  |  |
| --- | --- | --- | --- |
|  | |  | |
|  |  | |  |

Рис. 2-1 Результат выполнения forkdemo.cpp

Асинхронность и конкурентность процессов в данном примере иллюстрируется случайным характером переключения вывода от процесса к процессу на этапе исполнения. Невозможно точно предугадать порядок вывода несмотря на то, что известен исходный текст программы.

Переключение ввода-вывода процессов зависит от длины кванта времени, выделяемого каждому из них планировщиком процессов, от буферизации, выполняемой библиотекой ввода-вывода, и даже от характеристик конкретного компьютера.

1. Скомпилируем и выполним программу tinymenu.cpp

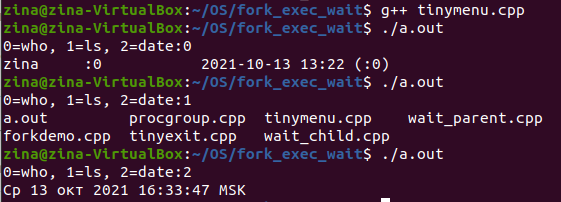


Рис. 2-2 Результат выполнения tinymenu.cpp

Для загрузки другого исполняемого файла из процесса потомка предназначен системный вызов exec(). В программе tinymenu используется версия системного вызова exec, позволяющая передавать параметры командной строки списком, а переменные окружения наследовать. С помощью вызова execlp в примере запускаются команды интерпретатора shell.

1. Скомпилируем и выполним программу tinyexit.cpp

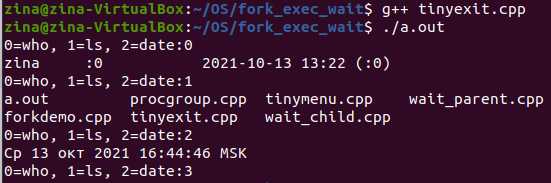


Рис. 2-3 Результат выполнения tinyexit.cpp

Посредством системного вызова wait() родительский процесс переводится в состояние ожидания, выход из которого происходит по событию завершения дочернего процесса, а именно, по выполнению в потомке вызова exit().

1. Скомпилируем и выполним программу procgroup.cpp

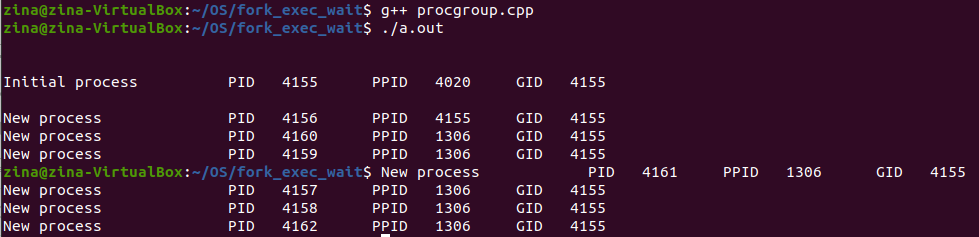


Рис. 2-4 Результат выполнения procgroup.cpp

Программа procgroup.cpp отображает информацию идентификатора группы процессов.

1. Скомпилируем и выполним программу wait\_parent.cpp

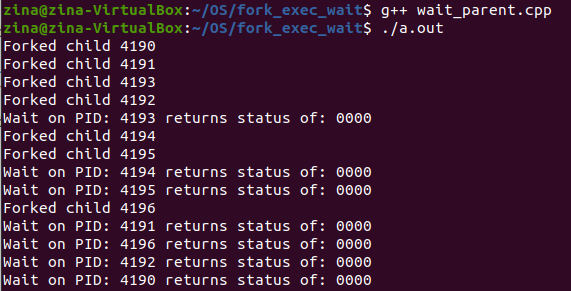


Рис. 2-5 Результат выполнения wait\_parent.cpp

В программе wait\_parent из процесса родителя запускаются три потомка, и затем системный вызов wait() отслеживает завершение каждого из них, сообщая, какой именно процесс закончил свое исполнение и какой код завершения был при этом передан. В качестве процесса-потомка выступает программа wait\_child, запускаемая из родителя с помощью execlp трижды с разными параметрами командной строки.

В потомках случайным образом формируется код завершения (для случаев нормального завершения), а также варьируется и сам способ завершения. При завершении по сигналу, номер сигнала передается процессу-родителю в младшем байте статуса завершения.

При нормальном же завершении, формируемый код завершения находится в передаваемой информации о статусе завершения, в байте, следующем после младшего (остальные байты обнуляются).

1. Скомпилируем и выполним программу wait\_child.cpp

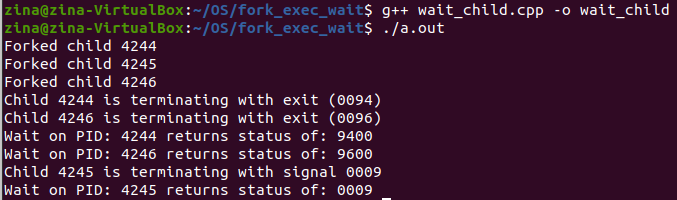


Рис. 2-6 Результат выполнения wait\_child.cpp

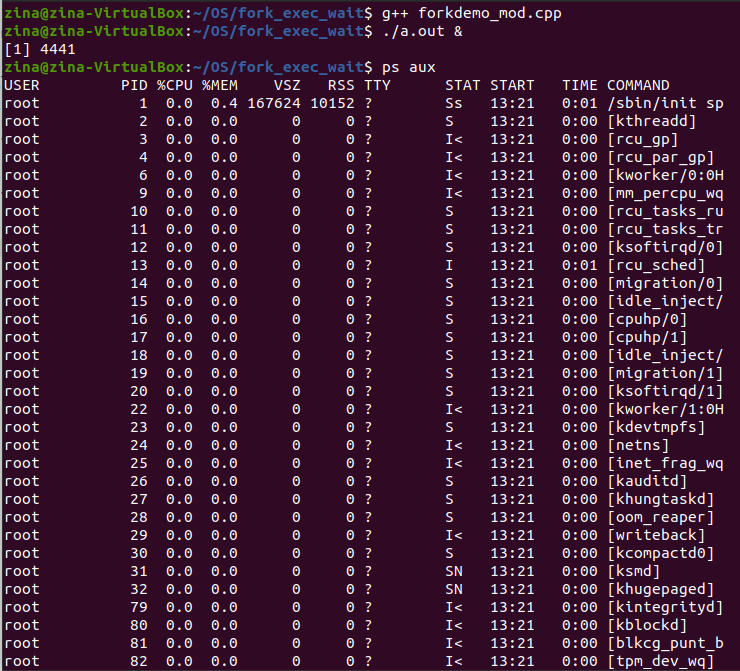
**Пункт 3**

Модифицируйте программу forkdemo.cpp (или создайте собственную), так чтобы ввод/вывод на терминал отсутствовал, а при проходе по циклу была временная задержка, например sleep (7). Запустите эту программу в фоновом режиме (background), введя при запуске символ & после пробела и зафиксировав значение PID, назначенное системой фоновому процессу при запуске. Выполните на терминале команды ps, top, uptime, pstre. Снимите свой фоновый процесс командой kill с соответствующими параметрами. Скриншоты вместе с пояснениями к выполнению процессов и команд, а также исходные тексты программ, составленных вами самостоятельно, приведите в отчете.

1. Модифицируем программу forkdemo.cpp:

|  |
| --- |
| Листинг 2.7 Код программы forkdemo\_mod.cpp |
| #include<stdio.h>  #include<sys/types.h>  #include<unistd.h>  int main()  {  pid\_t PID;  PID = fork();  int i;    if (PID) { /\* I must be the parent \*/  for(i=0; i<1000 ; i++) sleep(7);  }  else { /\* I must be the child \*/  for(i=0; i<1000 ; i++) sleep(7);  }  return 0;  } |

1. Запустим в фоновом режиме и выполним команду **ps**, которая выводит список процессов:



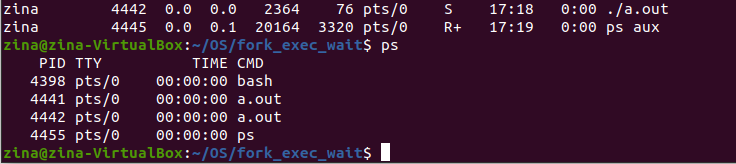


Рис. 2-7 Запуск forkdemo\_mod.cpp в фоновом режиме и

выполнение команды ps

1. Выполним команду **top**, чтобы интерактивно наблюдать за процессами:

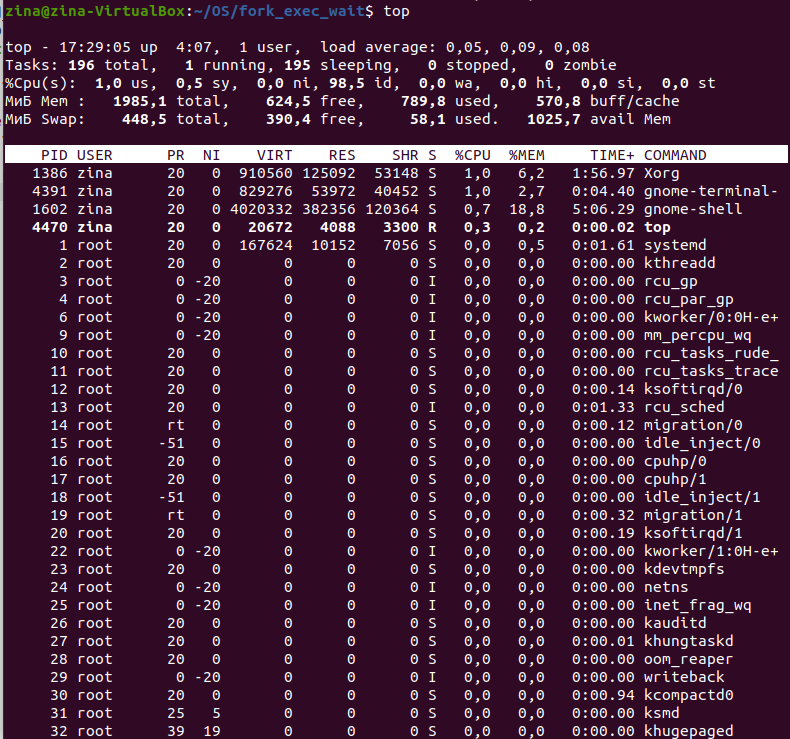


Рис. 2-8 Выполнение команды top

1. Выполним программу **uptime**, чтобы посмотреть загрузку системы:

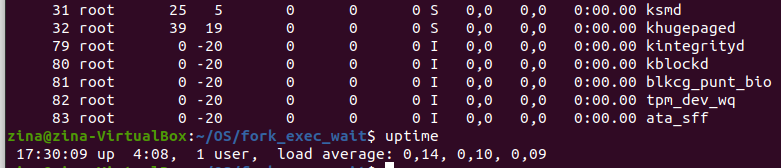


Рис. 2-9 Выполнение команды uptime

1. Выполним команду **pstree**, которая отображает все запущенные процессы в виде иерархии:

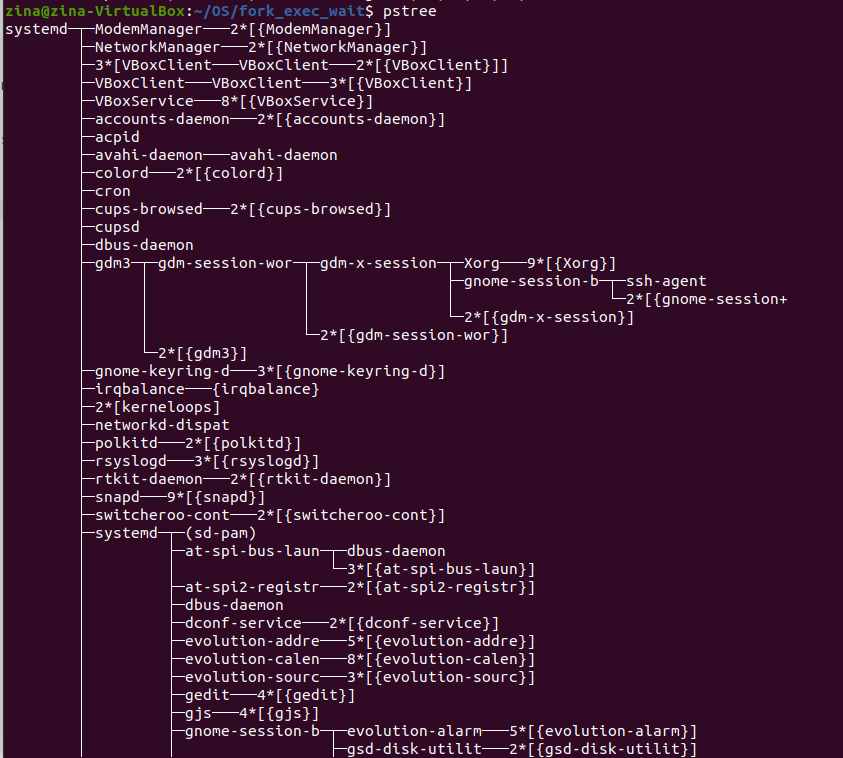


Рис. 2-10 Выполнение команды pstree

1. Снимем свой фоновый процесс командой **kill** с соответствующими параметрами, а затем с помощью команды **w** посмотрим список активных процессов для всех пользователей, где видим, что процесс завершен:

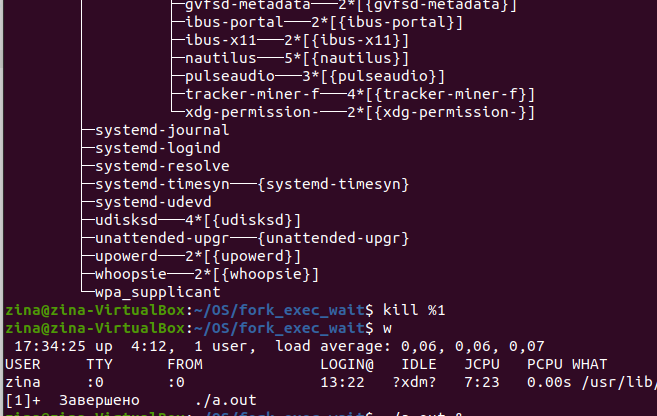


Рис. 2-11 Выполнение команд kill и w

**Пункт 4**

Исследуйте, что произойдет, если процесс-потомок сменит текущий каталог, будет ли изменен текущий каталог для родителя? Создайте программу, подтверждающую ответ, и приведите в отчете.

1. Составим программу, соответствующую заданию:

|  |
| --- |
| Листинг 2.8 Код программы change\_dir.cpp |
| #include<stdio.h>  #include<sys/types.h>  #include<unistd.h>  int main()  {  int i;  if (fork()) { /\*I must be the parent \*/  printf("\n %s \n", get\_current\_dir\_name());  for(i=0; i<15 ; i++)  printf("\t\tPARENT %d\n", i);  sleep(7);  }  else { /\*I must be the child \*/  printf("\n %d \n", chdir("home/zina/OS"));  printf("\n %s \n", get\_current\_dir\_name());  for(i=0; i<15 ; i++)  printf("CHILD %d\n",i);  sleep(7);  }  return 0;  } |

1. Скомпилируем и запустим программу change\_dir.cpp

|  |  |
| --- | --- |
|  |  |

Рис. 2-12 Результат выполнения change\_dir.cpp

Как видно из работы программы, при корректном перемещении процесса-потомка каталог процесса-родителя не изменился. Соответственно, данная операция не влияет на потоки процессов.

**Пункт 5**

Проиллюстрируйте как процесс-родитель и процесс-потомок разделяют один и тот же дескриптор и смещение текстового файла. Для этого составьте программу, в которой процесс-родитель должен открывать текстовый файл и запускать потомка. Потомок должен читать порцию данных из открытого файла и выводить на консоль. По завершению потомка родитель должен читать из того же файла и выводить результат на консоль. Можете использовать вызов sleep() для синхронизации доступа родителя и потомка к файлу.

1. Составим программу, соответствующую заданию:

|  |
| --- |
| Листинг 2.9 Код программы one\_des.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 upcome;  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((upcome = getline(&buf, &length, ptr)) != -1) printf("Child: %s", buf);  exit(0);  } else {  wait(NULL);  if((upcome = getline(&buf, &length, ptr)) != -1) printf("Parent: %s", buf);  fclose(ptr);  }  free(buf);  return 0;  } |

|  |
| --- |
| Содержание файла input.txt |
| This is the first line.  This is the second line. |

1. Скомпилируем и запустим программу one\_des.cpp

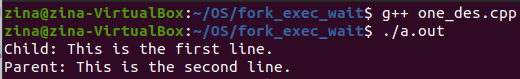


Рис. 2-13 Результат выполнения one\_des.cpp

Как видно из работы программы, она выполняет поставленную задачу.

**Вывод**

В ходе выполнения работы были изучены основные методы работы с процессами: запуск, создание процессов-потомков, ожидание и прерывание процесса. Были изучены свойства директорий текущих процессов, а также работа с идентификаторами процессов и с файлами.