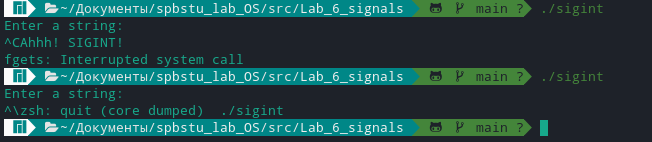
# **Лабораторная работа №6 «ГЕНЕРАЦИЯ И ОБРАБОТКА СИГНАЛОВ»**

**Цель работы**

Освоение простейшего средства управления процессами, позволяющего процессам передавать информацию о каких-либо событиях, отрабатывать реакции на различные события и взаимодействовать друг с другом.

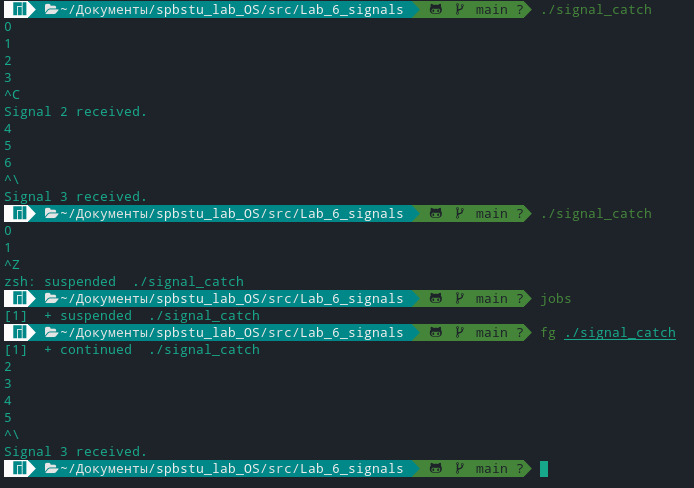
**Последовательность выполнения работы:**

1. Войдите в систему и скопируйте с разделяемого ресурса в свой HOME-каталог набор исходных файлов для шестого занятия.
2. Программа sigint.cpp осуществляет ввод символов со стандартного ввода. Скомпилируйте и запустите программу и отправьте ей сигналы SIGINT (нажатием Ctrl-C) и SIGQUIT (нажатием Ctrl-\). Проанализируйте результаты.



Программа перехватывает сигнал SIGINT, чтобы вывести соответствующее сообщение перед завершением.

1. Запустите программу signal\_catch.cpp , выполняющую вы-вод на консоль. Отправьте процессу сигналы SIGINT и SIGQUIT, а также SIGSTOP (нажатием Ctrl-Z) и SIGCONT (нажатием Ctrl-Q). Проанализируйте поведение процесса и вывод на консоль, а также сравните с программой из предыдущего пункта.



SIGINT – завершение процесса

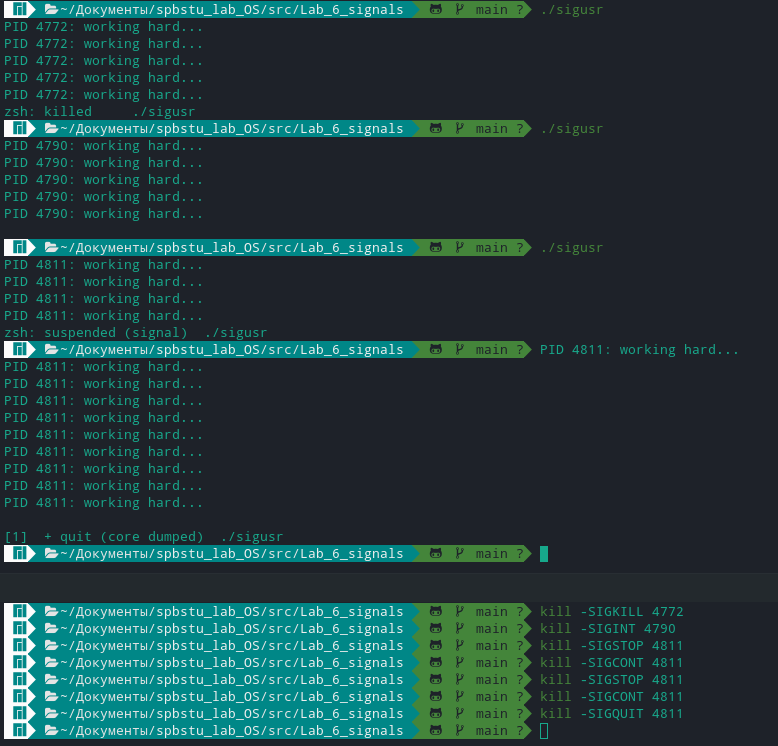
SIGQUIT – остановка процесса + выполнение дампа памяти для процесса (срез данных непосредственно из самой памяти, из ОЗУ)

SIGSTOP – остановка процесса

SIGCONT – запуск остановленного процесса

От прошлой программы эта отличается тем, что при получении SIGINT программа не завершается, то есть этот сигнал игнорируется.

1. Скомпилируйте и запустите программу sigusr.cpp. Программа выводит на консоль значение ее PID и зацикливается, ожидая получения сигнала. Запустите второй терминал и, отправляя с него командой kill различные сигналы, в том числе и SIGUSR1, проанализируйте реакцию на них.



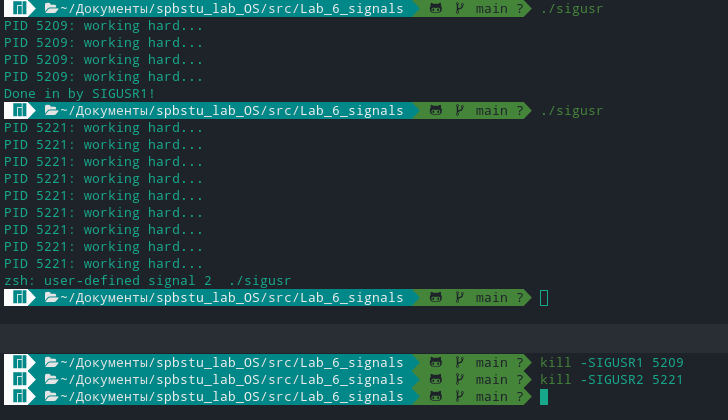
SIGKILL – убивает процесс

SIGINT – завершает процесс

SIGSTOP – приостанавливает процесс

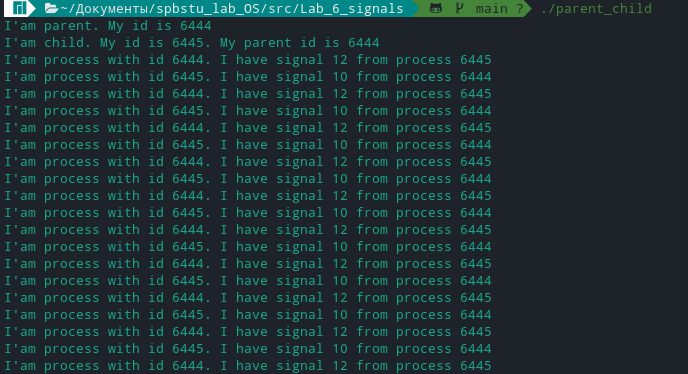
SIGCONT – возобновляет процесс в бэкграунде

SIGQUIT – остановка процесса + выполнение дампа памяти для процесса (срез данных непосредственно из самой памяти, из ОЗУ)



SIGUSR1 – этот сигнал ожидает программа и при получении его завершается сама

1. Составьте программу, запускающую процесс-потомок. Процессродитель и процесс-потомок должны генерировать (можно случайным образом) и отправлять друг другу сигналы (например, SIGUSR1, SIGUSR2). Каждый из процессов должен выводить на консоль информацию об отправленном и о полученном сигналах.
2. Для организации обработчиков сигналов предпочтительно использовать системный вызов sigaction() и соответствующую структуру данных. Обеспечьте корректное завершение процессов.



|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <errno.h>  #include <signal.h>  int process;  void sigusr1\_handler(int sig)  {  printf("I'am process with id %d. I have signal %d from process %d\n", getpid(), sig, process);  sleep(1);  }  int main()  {  int child = 0;  child = fork();  struct sigaction sa;  sa.sa\_handler = sigusr1\_handler;  sa.sa\_flags = 0;  sigemptyset(&sa.sa\_mask);  if ( child > 0)  {  if (sigaction(SIGUSR2, &sa, NULL) == -1) {  perror("sigaction");  exit(1);  }  process = child;  printf("I'am parent. My id is %d\n", getpid());  while(1) {  sleep(5);  kill(process, SIGUSR1);  }  }  else  {  if (sigaction(SIGUSR1, &sa, NULL) == -1) {  perror("sigaction");  exit(1);  }  process = getppid();  printf("I'am child. My id is %d. My parent id is %d\n", getpid(), getppid());  while(1) {  sleep(5);  kill(process, SIGUSR2);  }  }  return 0;  } |

Системный вызов signal() устанавливает новый обработчик сигнала с номером signum в соответствии с параметром sighandler, который может быть функцией пользователя, SIG\_IGN или SIG\_DFL  
  
Разница:

1. Функция signal() не обязательно блокирует поступление других сигналов во время выполнения текущего обработчика; функция sigaction() может блокировать другие сигналы до тех пор, пока текущий обработчик не вернется.
2. Функция signal() (обычно) сбрасывает действие сигнала обратно к SIG\_DFL (по умолчанию) почти для всех сигналов. Это означает, что обработчик signal() должен переустановить себя в качестве первого действия. Он также открывает окно уязвимости между временем обнаружения сигнала и переустановкой обработчика, в течение которого, если поступает второй экземпляр сигнала, происходит поведение по умолчанию (обычно завершается, иногда с предубеждением - он же дамп ядра).
3. sigaction() возвращает 0, если успешно и -1 в случае ошибки, signal() возвращает предыдущее значение обработчика сигнала или SIG\_ERR при ошибке.
4. Модифицируйте программу занятия 3 (файлы pipe\_server.cpp , pipe\_client.cpp и pipe\_local.h), сделав ее более стабильной в работе. В числе недостатков, которые желательно устранить, можно указать:
   * если клиентский процесс завершается по получению сигнала SIGINT (Ctrl+C), то private FIFO не удаляется из системы (исправляется посредством организации перехвата сигнала с выполнением необходимых действий);
   * клиентский процесс при его инициализации может обрушиться, если сервер окажется недоступен (исправляется путем попытки запуска сервера из клиента, если сервер не активен).

|  |
| --- |
| /\* The client program pipe\_client.cpp \*/  #include"pipe\_local.h"  FILE \*server;  void sigint\_handler(int sig) {  if (sig == SIGINT) {  exit(0);  }  }  int main(void)  {  int n, privatefifo, publicfifo;  static char buffer[PIPE\_BUF];  struct message msg;  /\* Make the name for the private FIFO \*/  sprintf(msg.fifo\_name, "/tmp/fifo %d", getpid());  struct sigaction sa;  sa.sa\_handler = sigint\_handler;  sa.sa\_flags = 0;  sigemptyset(&sa.sa\_mask);  if (sigaction(SIGINT, &sa, NULL) == -1) {  perror("sigaction");  exit(1);  }  /\* Generate the private FIFO \*/  if (mknod(msg.fifo\_name, S\_IFIFO | 0666, 0)<0){  perror(msg.fifo\_name);  exit(1);  }  /\* OPEN the public FIFO for writing \*/  int pid = 0;  if ((publicfifo=open(PUBLIC, O\_WRONLY))==-1){  pid = fork();  if (pid == 0) {  execl("./server", "&");  }  else {  sleep(2);  publicfifo=open(PUBLIC, O\_WRONLY);  }  }  while(1){ /\* FOREVER \*/  write(fileno(stdout), "\ncmd>", 6);  memset(msg.cmd\_line, 0x0, B\_SIZ); /\* Clear first \*/  n = read(fileno(stdin), msg.cmd\_line, B\_SIZ); /\* Get cmd \*/  if(!strncmp("quit", msg.cmd\_line, n-1))  break; /\* EXIT? \*/  write(publicfifo, (char \*) &msg, sizeof(msg)); /\* to PUBLIC \*/  /\* OPEN private FIFO to read returned command oputput \*/  if((privatefifo = open(msg.fifo\_name, O\_RDONLY))==-1){  perror(msg.fifo\_name);  exit(3);  }  /\* READ private FIFO and display on standard error \*/  while((n=read(privatefifo, buffer, PIPE\_BUF))>0){  write(fileno(stderr), buffer, n);  }  close(privatefifo);  }  close(publicfifo);  unlink(msg.fifo\_name);  } |

|  |
| --- |
| /\* The server program pipe\_server.cpp \*/  #include"pipe\_local.h"  int publicfifo;  void sigint\_handler(int sig) {  if (sig == SIGINT) {  close(publicfifo);  remove(PUBLIC);  exit(0);  }  }  int main(void)  {  int n, done, dummyfifo, privatefifo;  static char buffer[PIPE\_BUF];  FILE \*fin;  struct message msg;  /\* Generate the public FIFO \*/  mknod(PUBLIC, S\_IFIFO | 0666, 0);  struct sigaction sa;  sa.sa\_handler = sigint\_handler;  sa.sa\_flags = 0;  sigemptyset(&sa.sa\_mask);  if (sigaction(SIGINT, &sa, NULL) == -1) {  perror("sigaction");  exit(1);  }  /\* OPEN the public FIFO for reading and writing \*/  if ((publicfifo=open(PUBLIC, O\_RDONLY))==-1 ||  (dummyfifo=open(PUBLIC, O\_WRONLY | O\_NDELAY))==-1){  perror(PUBLIC);  exit(1);  }  /\* Message can be read from the PUBLIC pipe \*/  while(read(publicfifo, (char \*) &msg, sizeof(msg))>0){  n = done = 0; /\* Clear counters | flags \*/  do{ /\* Try OPEN of private FIFO \*/  if ((privatefifo=open(msg.fifo\_name, O\_WRONLY | O\_NDELAY))==-1)  sleep(3); /\* Sleep a while \*/  else{ /\* OPEN succesful \*/  fin = popen(msg.cmd\_line, "r"); /\* Execute the cmd \*/  write(privatefifo, "\n", 1); /\* Keep output pretty \*/  while((n=read(fileno(fin), buffer, PIPE\_BUF))>0){  write(privatefifo, buffer, n); /\*to private FIFO \*/  memset(buffer, 0x0, PIPE\_BUF); /\* Clear it out \*/  }  pclose(fin);  close(privatefifo);  done = 1; /\* Record succes \*/  }  }while(++n<5 && !done);    if(!done) /\* Indicate failure \*/  write(fileno(stderr), "\nNOTE: SERVER \*\* NEVER \*\* accessed private FIFO\n", 48);  }  return 0;  } |

**Вывод**

В ходе работы было освоены средства управления процессами, позволяющего процессам передавать информацию о каких-либо событиях,

отрабатывать реакции на различные события и взаимодействовать друг с другом.