Pliki, potoki, sygnały

dr inż. Sławomir Samolej
Katedra Informatyki i Automatyki
Politechnika Rzeszowska

Program przedmiotu oparto w części na materiałach opublikowanych na:
http://wazniak.mimuw.edu.pl/

oraz

Na materiałach opracowanych przez
dr inż. Jędrzeja Ułasiewicza:
jedrzej.ulasiewicz.staff.iiar.pwr.wroc.pl

Pliki

- W Linuksie wszystko jest plikiem (prawie)
- Dostęp do większości urządzeń (plików dyskowych, drukarek, konsoli, portów szeregowych i wielu innych) jest taki sam i odbywa się jak dostęp do plików
- Nieco inaczej odbywa się dostęp do sieci (gniazda)
- W dostępie do wymienionych urządzeń można się posłużyć zbiorem tzw. niskopoziomowych funkcji dostępu do plików: open, close, read, write i ioctl.

Specjalne pliki

- /dev/console domyślna konsola
- /dev/tty terminal
- /dev/null puste urządzenie

Podstawowe funkcje niskiego poziomu do obsługi plików i urządzeń

- open: otwórz plik lub urządzenie
- read: czytaj z otwartego pliku lub urządzenia
- write: zapisz coś do otwartego urządzenia lub pliku
- close: zamknij plik lub urządzenie
- ioctl: przekaż informacje sterujące do sterownika urządzenia

Domyślne deskryptory plików

- Każdy uruchomiony program ma powiązane ze sobą deskryptory plików.
- Domyślnie są to:
  - 0: standardowe wejście
  - 1: standardowe wyjście
  - 2: standardowe błędy

write()

```
#include <unistd.h>
size_t write(int fildes, const void *buf, size_t nbytes);
```

- fildes – deskryptor pliku (uzyskany z open)
- buf – bufor z danymi do zapisu
- nbytes – ilość danych do zapisu
- Funkcja zwraca ilość zapisanych bajtów, 0 gdy koniec pliku, -1 gdy błąd

Przykład:

```
#include <unistd.h>
#include <stdlib.h>
int main()
{ if ((write(1, “Here is some data\n”, 18)) != 18)
  write(2, “A write error has occurred on file descriptor 1\n”,46);
  exit(0);
}
```

```
$ simple_write
Here is some data
$
```

read()

```
#include <unistd.h>
size_t read(int fildes, void *buf, size_t nbytes);
```

- fildes – deskryptor pliku
- buf – bufor na dane
- nbytes – maksymalna ilość bajtów do odczytania
- Funkcja zwraca ile bajtów odczytała, 0-nic nie przeczytano koniec pliku, -1 – błąd.

```
#include <unistd.h>
#include <stdlib.h>
int main()
{  char buffer[128];
   int nread;
   nread = read(0, buffer, 128);
   if (nread == -1)
      write(2, "A read error has occurred\n", 26);
   if ((write(1,buffer,nread)) != nread)
      write(2, "A write error has occurred\n",27);
   exit(0);
}
```

Przekierowanie strumienia wyjściowego jednego polecenia na drugie

```
$ echo hello there | simple_read
hello there
$ simple_read < draft1.txt
Files
In this chapter we will be looking at files and
directories and how to manipulate
them. We will learn how to create files, o$
```

open()

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
int open(const char *path, int oflags);
int open(const char *path, int oflags, mode_t mode);
```

- path – ścieżka dostępu do pliku

oflags:

Tryb	Opis
O_RDONLY	Otwórz tylko do odczytu
O_WRONLY	Otwórz tylko do zapisu
O_RDWR	Otwórz do zapisu i odczytu
O_APPEND	Ustaw się na końcu pliku
O_CREAT	Utwórz plik jeśli potrzeba
O_EXCL	W połączeniu z O_CREAT zapewnia, że tylko wywołujący utworzy plik

mode (gdy użyjemy opcji O_CREAT):

- ❑ S_IRUSR: prawo czytania, właściciel
- ❑ S_IWUSR: prawo pisania, właściciel
- ❑ S_IXUSR: prawo wykonywania, właściciel
- ❑ S_IRGRP: prawo czytania, grupa
- ❑ S_IWGRP: prawo pisania, grupa
- ❑ S_IXGRP: prawo wykonywania, grupa
- ❑ S_IROTH: prawo czytania, inni
- ❑ S_IWOTH: prawo pisania, inni
- ❑ S_IXOTH: prawo wykonywania, inni

przykład:

```
open (“myfile”, O_CREAT, S_IRUSR|S_IXOTH);
```

close()/ ioctl()

```
#include <unistd.h>
int close(int fildes);
```

```
#include <unistd.h>
int ioctl(int fildes, int cmd, ...);
```

Wykonaj komendę cmd na pliku wskazywanym Przez fildes uwzględniając ew. dodatkowe parametry.

Przykład zastosowania

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
int main()
{
char c;
int in, out;
in = open(“file.in”, O_RDONLY);
out = open(“file.out”, O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
while(read(in,&c,1) == 1)
            write(out,&c,1);
exit(0);
}
```

Blokowanie dostępu do pliku

```
#include <sys/file.h>
int lockf(int fd, int cmd, off_t len);
```

Parametry funkcji:

fd	Uchwyt pliku
cmd	Specyfikacja operacji: F_LOCK, F_ULOCK, F_TEST,F_TLOCK
len	Zakres blokowania (o ile bajtów od bieżącego położenia plik ma być zablokowany) (0 - blokowany jest cały plik)

Wartości zwracane:

-1	Błąd
>0	Sukces

Specyfikacje operacji:

F_LOCK	Zablokuj dostęp do pliku na długości zakres od pozycji bieżącej
F_ULOCK	Zwolnij dostęp do pliku.
F_TEST	Testuj czy fragment pliku jest zablokowany przez inny proces
F_TLOCK	Testuj czy fragment pliku jest zablokowany przez inny proces. Gdy nie to zajmij plik

Kontrolowany zapis 2 procesów do 1 pliku - przykład

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

int main()
{
   int fd, i; pid_t pid;
   char buf1[]="To jest tekst1\n";
   char buf2[]="To jest tekst2\n";
   printf("Tworze proces potomny...\n");
   fd=open("plik1.txt",O_CREAT|O_WRONLY,0777);
   pid=fork();
   switch(pid)
   {   case -1: perror("Blad...\n");
                exit(1);
       case  0: for(i=0;i<10;i++)
                { lockf(fd,F_LOCK,0);
                  write(fd,buf1,sizeof(buf1));
                  sleep(2);
                  lockf(fd,F_ULOCK,0);  }  exit(112);
```

```
default: { int stat_val;
                for(i=0;i<10;i++)
                {
                  if(lockf(fd,F_TEST,0)==-1)
                  {  printf("Zablokowany...\n");
                  }
                  else
                  {   lockf(fd,F_LOCK,0);
                      write(fd,buf2,sizeof(buf2));
                      lockf(fd,F_ULOCK,0);
                  }
                  sleep(1);
                }
                wait(&stat_val);
                exit(0);
               }
}}
```

Standardowa biblioteka wejścia-wyjścia - przypomnienie

- fopen, fclose
- fread, fwrite
- fflush
- fseek
- fgetc, getc, getchar
- fputc, putc, putchar
- fgets, gets
- printf, fprintf, and sprintf
- scanf, fscanf, and sscanf

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
int c;
FILE *in, *out;
in = fopen(“file.in”,”r”);
out = fopen(“file.out”,”w”);
while((c = fgetc(in)) != EOF)
fputc(c,out);
exit(0);
}
```

Definicja potoku

- Terminu potok (pipe) używa się na określenie przepływu danych z jednego procesu do innego. Można powiedzieć, że potoki łączą wyjście jednego procesu z wejściem innego.
- Istnieje możliwość wywołania polecenia powłoki, które wiąże:
  - Standardowe wejście polecenia2 z klawiaturą terminala
  - Standardowe wyjście polecenia2 ze standardowym wejściem polecenia1
  - Standardowe wyjście polecenie1 z ekranem komputera:

```
polecenie2 | polecenie1
```

Przykład do przetestowania potoków ustalonych z linii poleceń

```
// polecenie2
#include <stdio.h>
#include <stdlib.h>

int main()
{
   printf("Hello world!\n");
   return 0;
}
```

```
// polecenie1
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main()
{
   char txt[100];
   int i=0;
   gets(txt);
   while(txt[i]!=0)
   {
      txt[i]=toupper(txt[i]);
      i++;
   }
   puts(txt);
   return 0;
}
```

Potoki procesowe

```
#include <stdio.h>
FILE *popen(const char *command, const char *open_mode);
int pclose(FILE *stream_to_close);
```

- popen
  - uruchamia inny programu jako nowy proces oraz przekazuje lub odbiera od niego dane
  - parametr *command* jest nazwą programu, który należy uruchomić wraz ze wszystkimi parametrami wywołania, parametr *open_mode* musi mieć wartość *r* lub *w.*
- pclose
  - czeka do momentu zakończenia uruchomionego procesu
  - zwraca kod wyjściowy uruchomionego procesu
  - jeśli proces uruchamiający wykona funkcję wait tracimy wówczas kod wyjściowy, *pclose* zwróci *-1*, a zamienna errno będzie zawierała wartość ECHILD

Przykład – użycie popen i pclose

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{  FILE *read_fp;
   char buffer[BUFSIZ + 1];
   int chars_read;
   memset(buffer, '\0', sizeof(buffer));
   read_fp = popen("uname -a", "r");
   if (read_fp != NULL)
   { chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
     if (chars_read > 0)
     { printf("Output was:-\n%s\n", buffer);
     }
     pclose(read_fp);
     exit(EXIT_SUCCESS);
   }
   exit(EXIT_FAILURE);
}
```

Output was:-
Linux gw1 2.4.20-8 #1 Thu Mar 13 17:54:28 EST 2003 i686 i686 i386 GNU/Linux

Wysyłanie danych ze pomocą popen

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main()
{ FILE *write_fp;
  char buffer[BUFSIZ + 1];
  sprintf(buffer, “Once upon a time, there was...\n”);
  write_fp = popen(“od -c”, “w”);
  if (write_fp != NULL)
  { fwrite(buffer, sizeof(char), strlen(buffer), write_fp);
    pclose(write_fp);
    exit(EXIT_SUCCESS);
  }
  exit(EXIT_FAILURE);
}
```

```
$ ./popen2
0000000 O n c e u p o n a t i m e
0000020 , t h e r e w a s . . . \n
0000037
```

Przekazywanie większej ilości danych

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{ FILE *read_fp;
  char buffer[BUFSIZ + 1];
  int chars_read;
  memset(buffer, '\0', sizeof(buffer));
  read_fp = popen("ps -ax", "r");
  if (read_fp != NULL)
  { chars_read = fread(buffer, sizeof(char),
                 BUFSIZ, read_fp);
    while (chars_read > 0)
    {  buffer[chars_read - 1] = '\0';
       printf("Reading:-\n %s\n", buffer);
       chars_read = fread(buffer, sizeof(char),
                          BUFSIZ, read_fp);
    }
    pclose(read_fp);
    exit(EXIT_SUCCESS);
  }
  exit(EXIT_FAILURE);
}
```

```
$ ./popen3
Reading:-
PID TTY STAT TIME COMMAND
1 ? S 0:04 init
2 ? SW 0:00 [kflushd]
3 ? SW 0:00 [kpiod]
4 ? SW 0:00 [kswapd]
5 ? SW< 0:00 [mdrecoveryd]
...
240 tty2 S 0:02 emacs draft1.txt
Reading:-
368 tty1 S 0:00 ./popen3
369 tty1 R 0:00 ps -ax
...
```

Funkcja pipe (potok nie nazwany)

```
#include <unistd.h>
int pipe(int file_descriptor[2]);
```

- Udostępnia ona taki mechanizm przekazywania danych pomiędzy dwoma programami, który nie wymaga uruchomienia powłoki w celu interpretacji żądanego polecenia
- Do pipe przekazywana jest tablica (wskaźnik do tablicy) dwóch liczb całkowitych będących deskryptorami plików.
- Funkcja wypełnia tablicę dwoma nowymi deskryptorami plików i zwraca zero w przypadku błędy zwraca -1 i ustawia zmienną errno.
- Dwa zwrócone deskryptory plików są połączone w specjalny sposób. Wszystkie dane zapisane do file_descriptor[1], mogą być odczytane przez file_descriptor[0]. Dane są przetwarzane według zasady FIFO
- Przesyłanie danych z i do takiego potoku musi się odbywać z zastosowaniem funkcji read, write a nie fread, fwrite.

Pierwsze zastosowanie funkcji pipe

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{ int data_processed;
  int file_pipes[2];
  const char some_data[] = “123”;
  char buffer[BUFSIZ + 1];
  memset(buffer, ‘\0’, sizeof(buffer));
  if (pipe(file_pipes) == 0)
  { data_processed = write(file_pipes[1], some_data, strlen(some_data));
    printf(“Wrote %d bytes\n”, data_processed);
    data_processed = read(file_pipes[0], buffer, BUFSIZ);
    printf(“Read %d bytes: %s\n”, data_processed, buffer);
    exit(EXIT_SUCCESS);
   }
   exit(EXIT_FAILURE);
}
```

```
$ ./pipe1
Wrote 3 bytes
Read 3 bytes: 123
```

Potoki i fork

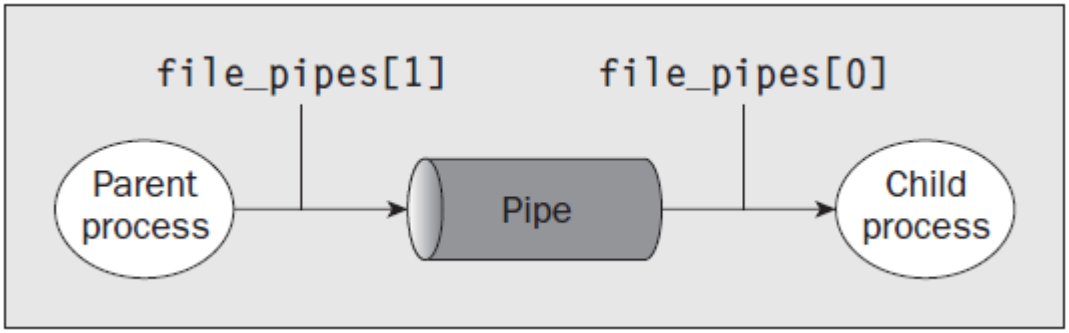
```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{ int data_processed;
  int file_pipes[2];
  const char some_data[] = “123”;
  char buffer[BUFSIZ + 1];
  pid_t fork_result;
  memset(buffer, ‘\0’, sizeof(buffer));
  if (pipe(file_pipes) == 0)
  { fork_result = fork();
    if (fork_result == -1)
    { fprintf(stderr, “Fork failure”);
      exit(EXIT_FAILURE);
    }
```

```
if (fork_result == 0)
{ data_processed = read(file_pipes[0], buffer,
                                    BUFSIZ);
   printf(“Read %d bytes: %s\n”, data_processed,
            buffer);
   exit(EXIT_SUCCESS);
}
else
{ data_processed = write(file_pipes[1],
                          some_data,strlen(some_data));
   printf(“Wrote %d bytes\n”, data_processed);
}
}
exit(EXIT_SUCCESS);
}
```

```
$ ./pipe2
Wrote 3 bytes
Read 3 bytes: 123
```

Jak to działa?

- Najpierw program tworzy potok, używając funkcji pipe
- Po sprawdzeniu, że funkcja fork zakończyła się pomyślnie proces macierzysty zapisuje dane do potoku, a proces potomny je odczytuje
- Proces macierzysty i potomny kończą pracę po pojedynczym wykonaniu instrukcji write/read
- Proces potomny dziedziczy w momencie wykonywania funkcji fork utworzone deskryptory plików
- Uzyskano możliwość przekazywania danych pomiędzy procesami.



Potoki i exec

Producent:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{ int data_processed; int file_pipes[2];
  const char some_data[] = “123”;
  char buffer[BUFSIZ + 1]; pid_t fork_result;
  memset(buffer, ‘\0’, sizeof(buffer));
  if (pipe(file_pipes) == 0)
  {  fork_result = fork();
     if (fork_result == (pid_t)-1)
     {   fprintf(stderr, “Fork failure”); exit(EXIT_FAILURE);}
     if (fork_result == 0)
     { sprintf(buffer, “%d”, file_pipes[0]);
       (void)execl(“pipe4”, “pipe4”, buffer, (char *)0);
       exit(EXIT_FAILURE);}
     else
     { data_processed = write(file_pipes[1], some_data,
       strlen(some_data));
       printf(“%d - wrote %d bytes\n”, getpid(), data_processed);}
   }
exit(EXIT_SUCCESS);}
```

Konsument:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{
int data_processed;
char buffer[BUFSIZ + 1];
int file_descriptor;
memset(buffer, ‘\0’, sizeof(buffer));
sscanf(argv[1], “%d”, &file_descriptor);
data_processed = read(file_descriptor,
buffer, BUFSIZ);
printf(“%d - read %d bytes: %s\n”,
getpid(), data_processed, buffer);
exit(EXIT_SUCCESS);
}
```

```
$ ./pipe3
980 - wrote 3 bytes
981 - read 3 bytes: 123
```

Jak to działa?

- Program producenta korzysta z funkcji pipe, aby stworzyć potok, a następnie z funkcji fork, aby utworzyć nowy proces
- Zapisuje w buforze „bufor” deskryptor pliku do odczytu otrzymany z funkcji pipe
- W ramach procesu potomnego wywoływana jest funkcja exec uruchamiająca inny program (konsument), do której jako parametry wywołania przekazywany jest deskryptor pliku do odczytu.
- Program konsumenta pobiera deskryptor pliku do odczytu i z niego czyta.

Czytanie zamkniętych potoków

- Zwykle dane z potoków odczytuje się porcjami w pętli
- Funkcja read blokuje proces – oczekuje aż będą dostępne następne dane
- Może się okazać, że potok od strony nadawcy zostaje zamknięty
- Wtedy funkcja read zwraca 0 (-1, gdy wykryto błąd otwarcia pliku) i program może zapobiec blokowaniu
- Jeśli korzystamy z potoku utworzonego dla dwu procesów z zastosowaniem funkcji fork, to w programie istnieją 2 deskryptory plików do zapisu w potoku.
- Abu potok uznał, że jest zamknięty oba z nich muszą zostać zamknięte.

Zamykanie nienazwanych potoków

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

    int
    main(int argc, char *argv[])
    {
       int pipefd[2];
       pid_t cpid;
       char buf;

       if (argc != 2) {
          fprintf(stderr, "Usage: %s <string>\n", argv[0]);
          exit(EXIT_FAILURE);
       }

       if (pipe(pipefd) == -1) {
          perror("pipe");
          exit(EXIT_FAILURE);
       }

       cpid = fork();
       if (cpid == -1) {
          perror("fork");
          exit(EXIT_FAILURE);
       }
```

```
       if (cpid == 0) {            /* Child reads from pipe */
          close(pipefd[1]);        /* Close unused write end */

          while (read(pipefd[0], &buf, 1) > 0)
             write(STDOUT_FILENO, &buf, 1);

          write(STDOUT_FILENO, "\n", 1);
          close(pipefd[0]);
          _exit(EXIT_SUCCESS);

       } else {                    /* Parent writes argv[1] to pipe */
          close(pipefd[0]);        /* Close unused read end */
          write(pipefd[1], argv[1], strlen(argv[1]));
          close(pipefd[1]);        /* Reader will see EOF */
          wait(NULL);              /* Wait for child */
          exit(EXIT_SUCCESS);
       }
    }
```

Nazwane potoki: FIFO

- Umożliwiają komunikację pomiędzy procesami, które nie są ze sobą spokrewnione
- Są specjalnym rodzajem pliku, który istnieje w systemie plików, ale zachowuje się jak nie nazwane potoki, które były omawiane do tej pory
- Polecenie systemowe tworzące nazwany potok: **mkfifo filename**
- Z poziomu programu można utworzyć nazwany potok z zastosowaniem funkcji:

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *filename, mode_t mode);
int mknod(const char *filename, mode_t mode | S_IFIFO, (dev_t) 0);
```

Tworzenie nazwanego potoku

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
int main()
{  int res = mkfifo(“/tmp/my_fifo”, 0777);
   if (res == 0) printf(“FIFO created\n”);
   exit(EXIT_SUCCESS);
}
```

```
ubuntu@ubuntu:~$ mkfifo /tmp/fifo1
ubuntu@ubuntu:~$ cat < /tmp/fifo1 &
[1] 13337
ubuntu@ubuntu:~$ echo "Ala ma kota" > /tmp/fifo1
ubuntu@ubuntu:~$ Ala ma kota

[1]+ Done cat                    < /tmp/fifo1
```

Otwieranie FIFO funkcją open

- open(const char *path, O_RDONLY);
  Open się zablokuje, to znaczy nie powróci, dopóki inny proces nie otworzy tego samego potoku do zapisu
- open(const char *path, O_RDONLY | O_NONBLOCK);
  Funkcja open zakończy się teraz pomyślnie i natychmiast powróci, nawet jeśli żaden proces nie otworzy FIFO do zapisu
- open(const char *path, O_WRONLY);
  W tym przypadku funkcja open zablokuje się, dopóki inny proces nie otworzy tego samego FIFO do odczytu
- open(const char *path, O_WRONLY | O_NONBLOCK);
  To wywołanie zawsze natychmiast wraca, ale jeśli wcześniej żaden proces nie otworzył FOFO do odczytu, open zwróci -1 i FIFO nie zostanie otwarte.
- Funkcja open może zostać zastosowana do **synchronizacji** pracy procesów.

FIFO: odczyt i zapis

- Użycie trybu O_NONBLOCK ma wpływ na zachowanie funkcji write i read.
  - Odczyt (read) z pustego, blokującego się FIFO (otwartego bez znacznika O_NONBLOCK) będzie oczekiwał, aż będzie można odczytać jakieś dane, natomiast odczyt z nieblokującego się FIFO, w którym nie ma żadnych danych, zwróci 0.
  - Zapis (write) do pełnego, blokującego się FIFO będzie oczekiwał, aż można będzie zapisać dane. Zapis do FIFO, które nie może przyjąć wszystkich bajtów, może zadziałać na 2 sposoby:
    - Spowodować błąd, ponieważ nie można przesłać wszystkich danych
    - Zapisać część danych, zwracając liczbę rzeczywiście wysłanych danych

Komunikacja międzyprocesowa przy użyciu FIFO – producent (fifo3)

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF
#define TEN_MEG (1024 * 1024 * 10)
int main()
{ int pipe_fd; int res;
  int open_mode = O_WRONLY;
  int bytes_sent = 0;
  char buffer[BUFFER_SIZE + 1];
  if (access(FIFO_NAME, F_OK) == -1)
  { res = mkfifo(FIFO_NAME, 0777);
    if (res != 0)
    {  fprintf(stderr, "Could not create fifo %s\n",
                                          FIFO_NAME);
       exit(EXIT_FAILURE);
    }
  }
```

```
  printf("Process %d opening FIFO O_WRONLY\n",
                                          getpid());
  pipe_fd = open(FIFO_NAME, open_mode);
  printf("Process %d result %d\n", getpid(),
                                          pipe_fd);
  if (pipe_fd != -1)
  {  while(bytes_sent < TEN_MEG)
     {   res = write(pipe_fd, buffer, BUFFER_SIZE);
         if (res == -1)
        {  fprintf(stderr, "Write error on pipe\n");
           exit(EXIT_FAILURE);
        }
        bytes_sent += res;
     }
     (void)close(pipe_fd);
  }
  else
  {  exit(EXIT_FAILURE);
  }
  printf("Process %d finished\n", getpid());
  exit(EXIT_SUCCESS);
}
```

Komunikacja międzyprocesowa przy użyciu FIFO – konsument (fifo4)

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF
int main()
{   int pipe_fd; int res;
    int open_mode = O_RDONLY;
    char buffer[BUFFER_SIZE + 1];
    int bytes_read = 0;
    memset(buffer, '\0', sizeof(buffer));
    printf("Process %d opening FIFO
    O_RDONLY\n", getpid());
```

```
    pipe_fd = open(FIFO_NAME,
                   open_mode);
    printf("Process %d result %d\n",
                  getpid(), pipe_fd);
    if (pipe_fd != -1)
    {  do
      {   res = read(pipe_fd, buffer,
                     BUFFER_SIZE);
          bytes_read += res;
       } while (res > 0);
      (void)close(pipe_fd);
    }
    else
    {   exit(EXIT_FAILURE);
    }
    printf("Process %d finished, %d bytes
    read\n", getpid(), bytes_read);
    exit(EXIT_SUCCESS);
}
```

Przykładowy rezultat pracy podanych 2 programów

```
$ ./fifo3 &
[1] 375
Process 375 opening FIFO O_WRONLY
$ time ./fifo4
Process 377 opening FIFO O_RDONLY
Process 375 result 3
Process 377 result 3
Process 375 finished
Process 377 finished, 10485760 bytes read

real    0m0.053s
user    0m0.020s
sys     0m0.040s

[1]+ Done fifo3
```

- Oba programy korzystają z FIFO w trybie blokującym
- Najpierw uruchamiany jest producent, który blokuje się, czekając na otwarcie FIFO przez program odczytujący
- Kiedy uruchomiony zostanie konsument, program zapisujący odblokowuje się i zaczyna wprowadzać dane do potoku
- W tym samym czasie program odczytujący rozpoczyna czytanie danych z potoku

Sygnały - wprowadzenie

- Sygnał to zdarzenie w systemie Linux powstałe w odpowiedzi na zaistnienie pewnych okoliczności
- Po otrzymaniu sygnału proces może podjąć określone czynności
- Sygnały są generowane przez niektóre błędy (naruszenie segmentów pamięci, błędy jednostki zmiennoprzecinkowej)
- Mogą być generowane przez powłokę i programy obsługi terminala, aby wykonać przerwanie.
- Mogą być też jawnie wysyłane z jednego procesu do drugiego w celu przesłania informacji lub modyfikacji pracy procesu.
- Na poziomie interfejsu można:
  - Generować sygnały
  - Przechwytywać sygnały
  - Wykonywać na ich podstawie pewne czynności
  - Zignorować (ale nie wszystkie).

Tablica sygnałów

Nazwa sygnału	Opis
SIGABORT	*Przerwanie procesu
SIGALRM	Zegar alarmowy
SIGFPE	*Wyjątek związany z jednostką zmiennoprzecinkową
SIGHUP	Zawieszenie
SIGILL	*Nielegalna instrukcja
SIGINT	Przerwanie z terminala
SIGKILL	„Zabójstwo” (nie można go przechwycić ani zignorować)
SIGPIPE	Zapis do potoku bez odbiorcy
SIGQUIT	Sygnał zakończenia terminala
SIGSEGV	*Dostęp do niewłaściwego segmentu pamięci
SIGTERM	Zakończenie
SIGUSR1	Sygnał 1 zdefiniowany przez użytkownika
SIGUSR2	Sygnał 1 zdefiniowany przez użytkownika

Reakcja procesów na sygnały

- Jeśli proces otrzyma jeden z wymienionych wyżej sygnałów i nie jest przygotowany na jego przechwycenie, to zostaje natychmiast zakończony
- W przypadkach sygnałów oznaczonych * mogą zostać podjęte w systemie czynności zależne od implementacji
- W chwili takiego zakończenia procesu w jego bieżącym katalogu tworzony jest plik „core” zawierający zrzut pamięci

Sygnały dodatkowe

Nazwa sygnału	Opis
SIGCHLD	Proces potomny zatrzymał się lub zakończył pracę
SIGCONT	Wznowienie wykonania, o ile proces był zatrzymany
SIGSTOP	Zatrzymanie wykonywania (nie można go przechwycić ani zignorować)
SIGTSTP	Sygnał zatrzymania terminala
SIGTTIN	Proces pracujący w tle próbuje przeprowadzić odczyt
SIGTTOU	Proces pracujący w tle próbuje przeprowadzić zapis

- SIGCHILD można zastosować do sterowania procesami potomnymi
- Pozostałe, poza SIGCONT powodują zatrzymanie otrzymujących je procesów
- Normalnie skonfigurowany terminal po przyciśnięciu kombinacji Ctrl+C powoduje wysłanie sygnału SIGINT do pierwszoplanowego procesu uruchomionego na tym terminalu.

Manipulowanie sygnałami

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

- Funkcja signal przyjmuje 2 parametry:
  - Sygnał, który należy przechwycić lub zignorować (sig)
  - Funkcję, którą należy wywołać po otrzymaniu sygnału, która:
    - Musi przyjmować pojedynczy argument tupu int, a sama być typu void
- Funkcja signal zwraca z kolei funkcję tego samego typu – poprzednią wartość funkcji ustawionej do obsługi sygnału, albo jedną z wartości specjalnych:
  - SIG_IGN = zignorować sygnał
  - SIG_DFL = przywrócić domyślne zachowanie
- W systemie Linux domyślne zachowanie przywracane jest automatycznie

Prosta obsługa sygnałów - przykład

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void ouch(int sig)
{
printf("OUCH! - I got signal
%d\n", sig);
(void) signal(SIGINT, SIG_DFL);
}
int main()
{
(void) signal(SIGINT, ouch);
while(1) {
printf("Hello World!\n");
sleep(1);
}
}
```

```
$ ./ctrlc1
Hello World!
Hello World!
Hello World!
Hello World!
^C
OUCH! - I got signal 2
Hello World!
Hello World!
Hello World!
Hello World!
^C
$
```

Sygnał jest domyślnie obsługiwany tylko raz. Do ponownej obsługi trzeba procedurę obsługi ponownie „ustanowić" (pojawia się możliwość niezdążenia z ponowną obsługą).

Wysyłanie sygnałów

Wysłanie sygnału do dowolnego procesu:

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

Funkcja alarm planuje dostarczenie sygnału *SIGALRM* za *seconds* sekund

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

Symulator funkcji alarm - budzik

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
static int alarm_fired = 0;
void ding(int sig)
{ alarm_fired = 1;
}
int main()
{ pid_t pid;
  printf("alarm application starting\n");
  pid = fork();
  switch(pid) {
  case -1:   /* Failure */
             perror("fork failed");
             exit(1);
  case 0:    /* child */
             sleep(5);
             kill(getppid(), SIGALRM);
             exit(0);
  }
```

```
/* if we get here we are the parent process */
printf("waiting for alarm to go off\n");
(void) signal(SIGALRM, ding);
pause();
if (alarm_fired) printf("Ding!\n");
printf("done\n");
exit(0);
}
```

```
$ ./alarm
alarm application starting
waiting for alarm to go off
<5 second pause>
Ding!
done
$
```

Jeśli zastosujemy pause, a sygnał już został wysłany do naszego procesu, to możemy spowodować „zawieszenie" procesu.

Odporny interfejs sygnałowy

```
#include <signal.h>
int sigaction( int sig, const struct sigaction *act,
               struct sigaction *oact);
```

- Struktura sigaction jest zdefiniowana w pliku nagłówkowym signal.h i musi się składać przynajmniej z następujących elementów:
  - void (*) (int) sa_handler — funkcja, SIG_DFL lub SIG_IGN
  - sigset_t sa_mask — sygnały, które należy zablokować w sa_handler
  - int sa_flags — modyfikatory reakcji na sygnał
- Funkcja sigaction ustala reakcję na sygnał sig. Jeśli oact ma wartość inną niż null, sigaction zapisuje poprzednią reakcję na sygnał w lokacji wskazanej przez oact
- Jeśli act ma wartość null, to jest to jedyny rezultat działania sigaction; jeśli zaś act jest różne od null, wówczas ustawiana jest reakcja na określony sygnał
- Wewnątrz struktury sigaction, na którą wskazuje argument act, sa_handelr jest wskaźnikiem do funkcji wywoływanej po otrzymaniu sygnału sig.

Przykładowy program stosujący sigaction

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

void ouch(int sig)
{ printf("OUCH! - I got signal %d\n", sig);
}

int main()
{
  struct sigaction act;
  act.sa_handler = ouch;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;
  sigaction(SIGINT, &act, 0);
  while(1) {
            printf("Hello World!\n");
            sleep(1);
           }
}
```

```
$ ./ctrlc2
Hello World!
Hello World!
Hello World!
^C
OUCH! - I got signal 2
Hello World!
Hello World!
^C
OUCH! - I got signal 2
Hello World!
Hello World!
^\
Quit
$
```