POSIX Pipes/FIFOs (mostly)

Last modification date: 18.02.2019

POSIX definitions

- **FIFO special file (or FIFO)**

A type of file with the property that data written to such a file is read on a first-in-first-out basis.

- **Pipe**

An object accessed by one of the pair of file descriptors created by the ***pipe( )*** function. Once created, the file descriptors can be used to manipulate it, and it behaves identically to a FIFO special file when accessed in this way. It has no name in the file hierarchy.

FIFO/pipe related defines

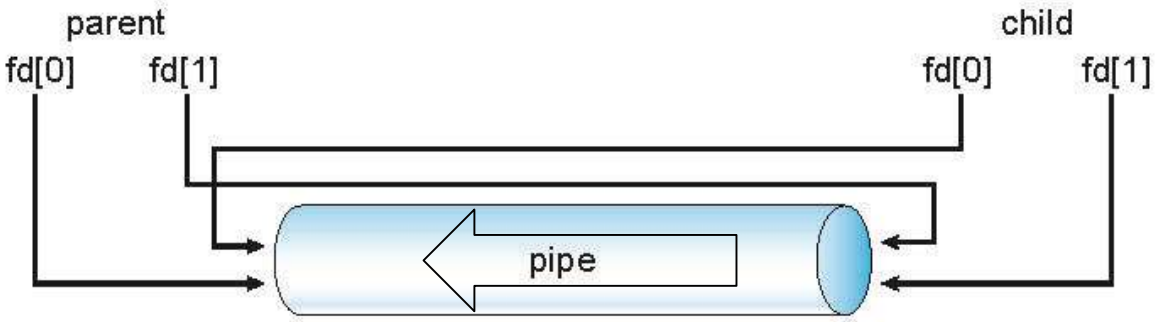
Header file	Symbols
<errno.h>	EPIPE, ESPIPE
<limits.h>, <unistd.h>	PIPE_BUF
<signal.h>	SIGPIPE
<sys/stat.h>	S_ISFIFO(*m*)

Pipes

- Act as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?
- **Ordinary pipes** – are not visible outside of the process that created it. However, a parent process, which created a pipe, can communicate with a child process that inherited access to the pipe.
- **Named pipes (Unix FIFOs)** – can be used for communication by processes which know the name, and have needed (read or wright) access rights.

Ordinary Pipes

- **Ordinary pipes** allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are typically unidirectional (but can be bi-directional, e.g. in some UNIX systems which support STREAM subsystem)



- Communication require passing descriptors to pipe endpoint(s), e.g.
  - Child process can inherit pipe descriptors of its parent
  - In Unix systems descriptors can be passed with local socket IPC.

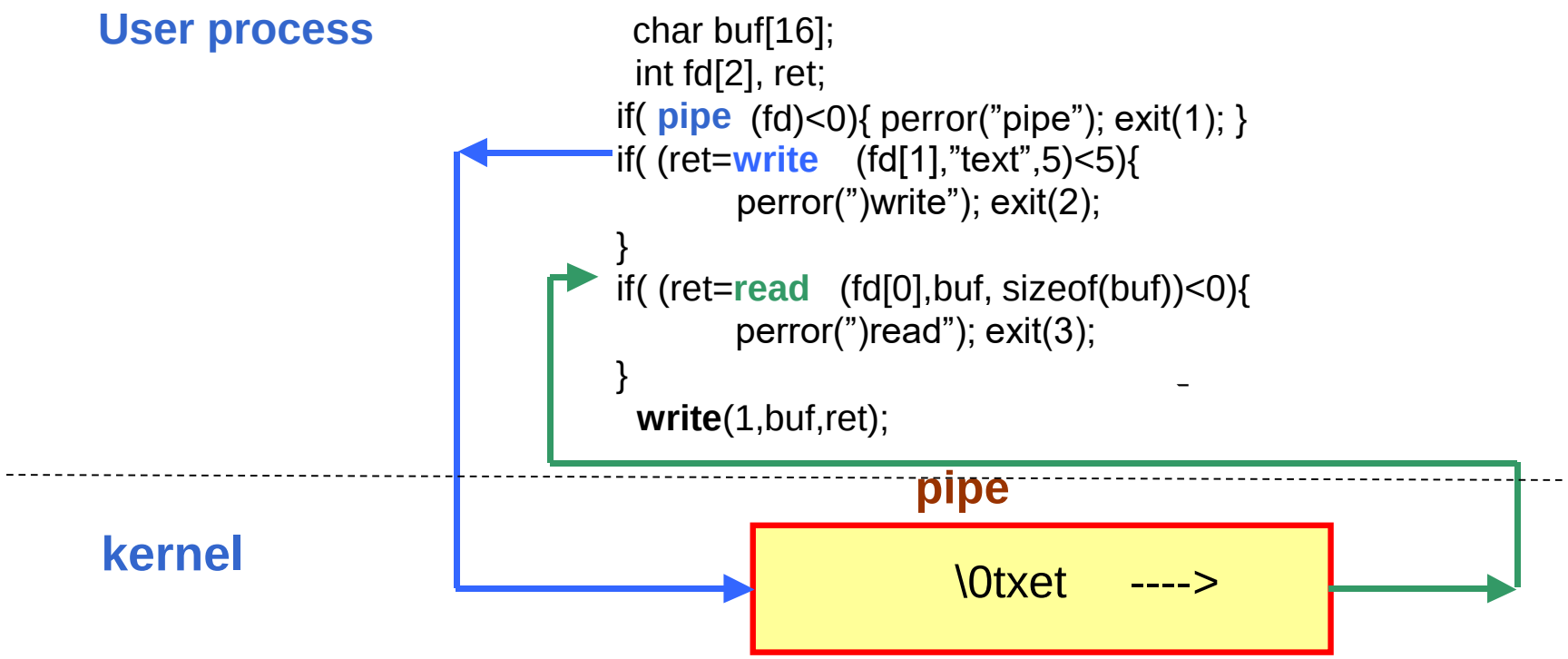
POSIX/UNIX pipe creation

```
#include <unistd.h>
int ret=pipe (int filedes[2])
```

creates uni-directional pipe, stores file descriptors for the reading and writing pipe ends (respectively) into **filedes**[0] and **filedes**[1]

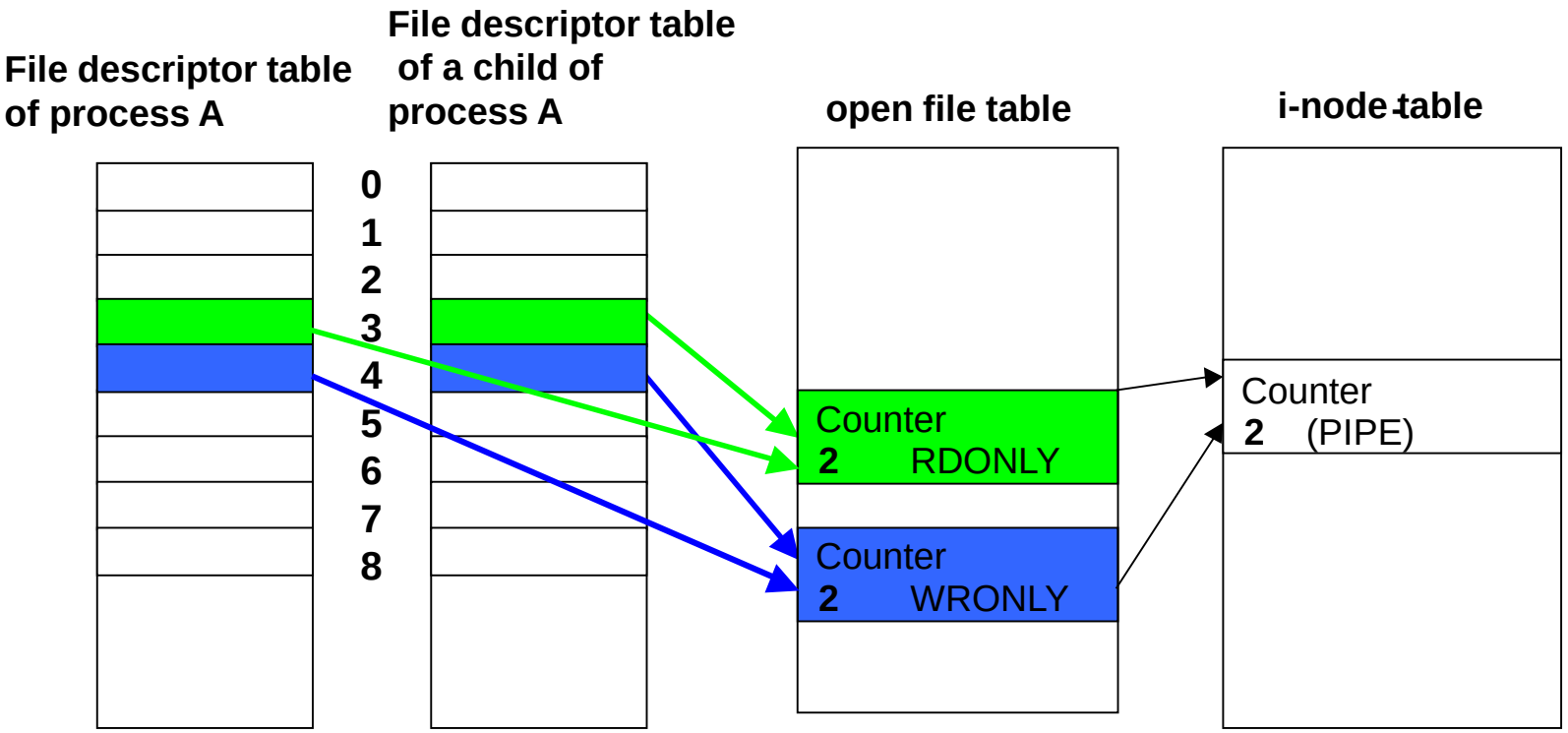
Upon success **ret==0**, otherwise **ret==-1** (error code in **errno**).

Illustration of pipe creation and trivial use



Unix/POSIX: pipes and fork()

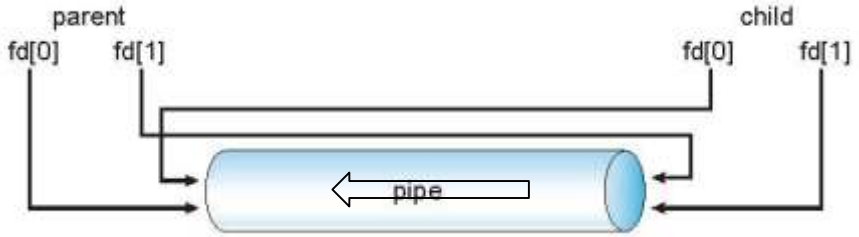
fork() makes the pipe descriptors available to the child process, thus enabling parent-child communication

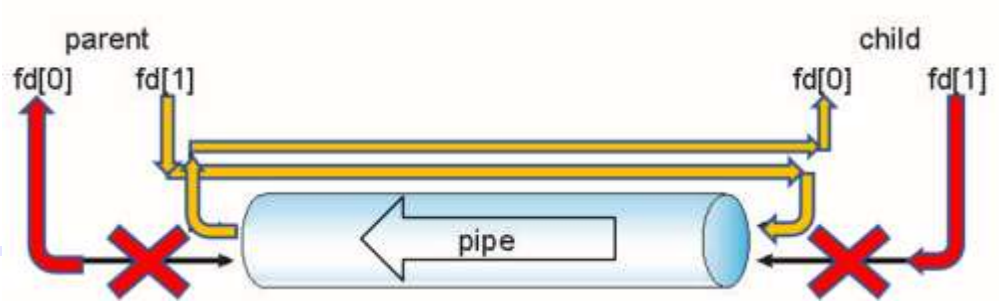


IPC with a pipe - 1

Example. The parent process sends data to its child process (**write**() / **read**())

```
...
int main(void){
char buf[16];
int fd[2], pid, ret;
  if( pipe(fd)<0){/* error handling */ }
  if( (pid=fork()) == -1 ){        ){ /* error handling */ }
  else if(pid>0){ /* parent process */
        close(fd[0]); /* closing unused pipe end */
        if( (ret=write(fd[1],"tekst",6)<6){/* error */}
        wait(NULL); /* waiting for child process to terminate */
  } else { /* child process */
        close(fd[1]); /* closing unused pipe end */
        if( (ret=read(fd[0],buf, sizeof(buf))<0){{/* error */}
        write(1,buf,ret);
  }
return EXIT_SUCCESS;
}
```

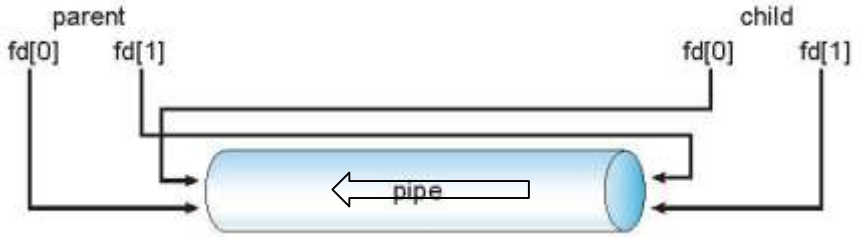


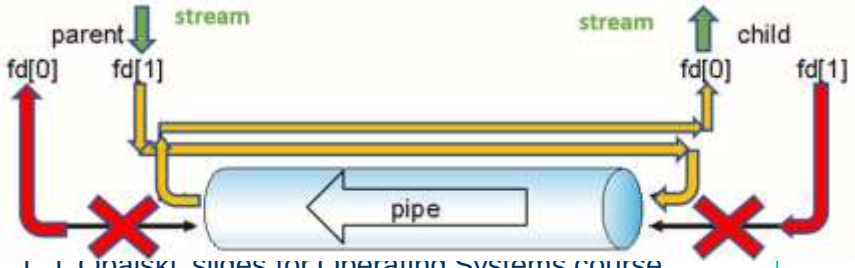


IPC with a pipe - 2

Example. The parent process sends data to its child process (using **stream**)

```
...
int main(void){
int fd[2], ret;
pid_t pid;
FILE *stream;
     if( pipe(fd)<0){/* error handling */ }
     if( (pid=fork()) == -1 ){ /* error handling */ }
     else if(pid>0){
        close(fd[0]); /* closing unused pipe end */
        stream=fdopen(fd[1],"w");
        fprintf(stream,"Message …");
        fclose(stream);
        if( wait(NULL)<0 ){ /* error handling */ }
     } else {
        close(fd[1]); /* closing unused pipe end */
        stream =fdopen(fd[0],"r");
        while( (ret=fgetc(stream)) != EOF )
                putchar(ret);
        fclose(stream);
     }
     return EXIT_SUCCESS;
}
```





Pipes – cont.

- Properties of a pipe:
  - Access to a pipe – only via its file descriptors (that are inherited).
  - One cannot get/set position. An attempt sets **errno=ESPIPE** (invalid seek)
  - An attempt to write to a closed pipe sets **errno=EPIPE** (broken pipe); also **SIGPIPE** signal is sent to the writer process.
  - Limited capacity (**PIPE_BUF** >=512B)
  - No logical record boundaries
  - While pipe file descriptor is in blocking mode (default)
    - Reading or writing pipe is atomic if the size of data to be sent is not greater than **PIPE_BUF**.
    - If a message of size <= **PIPE_BUF** would overflow the pipe – the process is waiting until enough free space is available. Messages longer than **PIPE_BUF** are sent in fragments.

    For discussion of non-blocking mode – see later.
  - When all file descriptors associated with a pipe are closed, any data remaining in the pipe shall be discarded.

Pipes – cont.

#include <stdio.h>

FILE * fp=popen (const char *cmd, const char *mode)

Creates a sub-process (using command **cmd**) and if mode is equal to:

- **„r”** – then **fp** is a stream connected to **stdout** of the sub-process
- **„w”** – then **fp** is a stream connected to **stdin** of the sub-process

int pclose(FILE *fp)

closes access to the stream that was created with **popen()**, waits for the command to terminate and returns the termination status of the process that was running the command language interpreter. For more details see **man pclose.**

Example popen usage

```
/* cmdlog.c - Executes shell command(cmd), duplicating
  standard output stream (to a file: log) */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
char buf[BUFSIZ];
FILE *fp1, *fp2;
size_t n;
  if(argc!=3){
       fprintf(stderr,"Usage: cmdlog  log  cmd"); exit(1);
   }
  if((fp1=fopen(argv[1],"w"))==NULL){ /* error */... }
  if((fp2=popen(argv[2],"r"))==NULL){ /* error */...}

  while((n=fread(buf,sizeof(char),sizeof(buf),fp2))>0){
       if(fwrite(buf,sizeof(char),n,fp1)!=n){/* error */...}
       if(fwrite(buf,sizeof(char),n,stdout)!=n){/* error */..}
  }
  (void) pclose(fp2); (void) fclose(fp1);
  return 0;
}
```

FIFOs

FIFO – a type of a file with the property that data written to such a file is read in the first-in-first-out order.

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

creates the FIFO type file system object (location specified with relative or absolute **path**), **mode** determines access mode (as for **open()**).

The FIFO can be used for communication in the same way as pipe, after it is opened **both** for reading and writing.

One FIFO can be used by more than 2 processes – if at least one of them opened it for writing and at least one opened it for reading.

Note: By default **open**() w.r.t. a FIFO, blocks all threads, until at least one of them attempts opening for reading, and at least one attempts to open FIFO for writing.

Example of FIFO usage

```
/* fifosv.c */
…
#define FIFO_FILE "MYFIFO"
int main(void){
  char buf[80];
  int fd, m, n;
  unlink(FIFO_FILE);
  umask(0);
  if(mkfifo(FIFO_FILE, 0666)){. .}
  if((fd=open(FIFO_FILE,O_RDONLY)) <0){
    . . .
  }
  while((n=read(fd,buf,80)) >0 ){
     if((m=write(1,buf,n))!=n){ . . .
     } else {
``````fprintf(stderr,"%d B read\n”,n);
     }
   } /* while() */
   if(n==0){
````  fputs("EOD\n",stderr); return 0;
   }
   if(errno) perror(„fifosv”);
   return 0;
}
```

```
/* fifocl.c */
…
#define FIFO_FILE "MYFIFO"
void handler(int sig){
  fputs("SIGPIPE\n",stderr); return;
}
int main(int argc, char*argv[]){
   char buf[80];// what if 8000?
   int fd, m, n;
   signal(SIGPIPE,handler); /* ☺ */
   if((fd=open(FIFO_FILE,O_WRONLY))<0){.}
   while((n=read(0,buf,80))>0 ){
      if( (m=write(fd,buf,n))!=n ){ . . .
      } else {
         fprintf(stderr,
              "%d B to FIFO\n",m);
      }
    }/* while() */
    if(n==0)fputs("EOD\n",stderr);
    else {
      if(errno==EINTR)
         fputs("EINTR\n",stderr);
      else perror("fifocl");
    }
    return 0;
}
```

Deadlock while opening FIFOs

Unfortunate coding two-way communication with two (created earlier) FIFOs: "F12", "F21" can result in **deadlock**. Example of bad coding follows.

```
/*  prog1.c */
...

int main(void){
int fd1, fd2;
...
if((fd1=open("F21",O_RDONLY))<0 ){
      perror("open");    return 1;
}
if((fd2=open("F12",O_WRONLY))<0 ){
      perror("open");    return 1;
}
/* code that is to read from fd1
 * and writes to fd2 */
...
 return 0;
}
```

```
/*  prog2.c */
...

int main(void){
int fd1, fd2;
...
if((fd1=open("F12",O_RDONLY))<0 ){
      perror("open");    return 1;
}
if((fd2=open("F21",O_WRONLY))<0 ){
      perror("open");    return 1;
}
/* code that is to read from fd1
 * and writes to fd2  */
...
 return 0;
}
```

Non-blocking use of FIFOs

- When opening a FIFO with O_RDONLY or O_WRONLY set:
  - If **O_NONBLOCK** flag of the file descriptor is set, an ***open( )*** for reading-only shall return **without delay**. An ***open( )*** for writing-only **shall return an error** if no process currently has the file open for reading.
  - If **O_NONBLOCK** is clear, an ***open( )*** for reading-only shall block the calling thread until a thread opens the file for writing. An ***open( )*** for writing-only shall block the calling thread until a thread opens the file for reading.

Example use of O_NONBLOCK to avoid deadlock on FIFO open.

```
for(i=0; i<20; i++){/* try 20 times (polling)*/
   fd = open(pname,O_RDONLY|O_NONBLOCK);/* returns -1 if failed */
   if(fd!=-1) break;
   if(errno!=ENXIO){
        perror("opening FIFO"); exit(1);
   } else {
        printf("waiting for a client");
        sleep(2);
   }
}
```

Non-blocking use of FIFOs – cont.

- When attempting to write to a pipe or FIFO:
  - If the **O_NONBLOCK** flag is set
    - The ***write( )*** function shall not block the thread.
    - A write request for {**PIPE_BUF**} (system defined constant) or fewer bytes shall have the following effect:
      - if there is sufficient space available in the pipe, ***write( )*** shall transfer all the data and return the number of bytes requested.
      - Otherwise, ***write( )*** shall transfer no data and return **−1** with ***errno*** set to [**EAGAIN**].
    - A write request for more than {**PIPE_BUF**} bytes shall cause one of the following:
      - When at least one byte can be written, transfer what it can and return the number of bytes written.
      - When no data can be written, transfer no data, and return **−1** with ***errno*** set to [**EAGAIN**].

Summary of writing conditions

Write to a Pipe or FIFO with O_NONBLOCK *clear*			
Immediately Writable:	**None**	**Some**	*nbyte*
nbyte≤{PIPE_BUF}	Atomic blocking *nbyte*	Atomic blocking *nbyte*	Atomic immediate *nbyte*
nbyte>{PIPE_BUF}	Blocking *nbyte*	Blocking *nbyte*	Blocking *nbyte*

Write to a Pipe or FIFO with O_NONBLOCK *set*			
Immediately Writable:	**None**	**Some**	*nbyte*
nbyte≤{PIPE_BUF}	−1, [EAGAIN]	−1, [EAGAIN]	Atomic *nbyte*
nbyte>{PIPE_BUF}	−1, [EAGAIN]	<*nbyte* or −1, [EAGAIN]	≤*nbyte* or −1, [EAGAIN]

Non-blocking use of FIFOs – cont.

- When attempting to read from an empty pipe or FIFO:
  - If no process has the pipe open for writing, ***read( )*** shall return **0** to indicate end-of-file.
  - If some process has the pipe open for writing and **O_NONBLOCK** is set, ***read( )*** shall return **−1** and set ***errno*** to [**EAGAIN**].
  - If some process has the pipe open for writing and **O_NONBLOCK** is clear, ***read( )*** shall block the calling thread until some data is written or the pipe is closed by all processes that had the pipe open for writing.

Remarks:

read returns **0:**

- In blocking mode (**O_NONBLOCK**) - to indicate end of data condition.
- In non-blocking mode (**O_NONBLOCK** is set) - to indicate empty pipe/FIFO (temporary condition) or no connection to that pipe/FIFO in write mode from any thread/process. Thus the end of data condition has to be inferred from data content.

Use of non-blocking FIFOs/pipes is **more challenging** than use of blocking ones.

Switching blocking/non-blocking mode

```
#include <fcntl.h>
int set_nonblock_flag (int desc, int value){
/* Set the O_NONBLOCK flag of desc if value is nonzero,
   or clear the flag if value is 0.
   Return 0 on success, or -1 on error with errno set.
   Source: glibc documentation
*/
   int oldflags = fcntl(desc, F_GETFL, 0);
/* If reading the flags failed, return error indication */
  if (oldflags == -1)
       return -1;
  /* Set just the flag we want to set. */
  if (value != 0)
       oldflags |= O_NONBLOCK;
  else
       oldflags &= ~O_NONBLOCK;
  /* Store modified flag word in the descriptor. */
  return fcntl(desc, F_SETFL, oldflags);
}
```

Remarks regarding MS Win pipes

- An ***anonymous pipe*** is an unnamed, one-way pipe that typically transfers data between a parent process and a child process. Anonymous pipes are always local; they cannot be used for communication over a network.
- Under MS Windows ***named pipe*** is a named, one-way or duplex pipe for communication between the pipe server and one or more pipe clients. The term pipe server refers to a process that creates a named pipe, and the term pipe client refers to a process that connects to an instance of a named pipe.
- Named pipes can be used to provide communication between processes on the same computer or between processes on different computers across a network.
- Pipe names follow UNC (Universal Naming Convention), embedding host name, e.g. **\\myhost\pipe\mypipe** points at pipe **mypipe** at the host **myhost**
- All instances of a named pipe share the same pipe name, but each instance has its own buffers and handles, and provides a separate conduit for client/server communication.
- Data can be transmitted through a named pipe as either a stream of bytes or as a stream of messages.

Note: Linux (>=3.4) also supports transport of messages of variable length via anonymous pipes (so called packet mode – see **man pipe**).