# Computer Operating Systems, Practice Session 12

## Linux Pipe Structure

Mustafa Ersen (ersenm@itu.edu.tr)

Istanbul Technical University
34469 Maslak, Äřstanbul

07 May 2014

## Today

## What is Pipe?

- A one-way communication channel used for inter-process communication managed by the OS.
- Pipes can be considered as special files that may keep data up to specified limit with FIFO principle.
- In general: a process writes data onto a pipe and another process reads data from pipe.

## Pipe & Concurrency

OS ensures that processes using the pipe run concurrently.

- If pipe is full : Process trying to write onto pipe is suspended until sufficient data has been read from the pipe to allow the write to complete.
- If pipe is empty: Process trying to read from pipe is suspended until data is available.
- If a pipe's output descriptor is closed, reader sees EOF.
- If a pipe's input descriptor is closed, writer gives SIGPIPE signal.

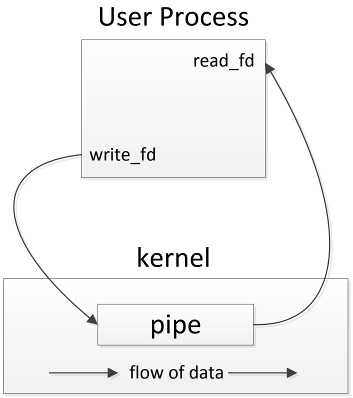
## Types of Pipes

- Most important restriction of pipes is that they have no name. This property necessitates their usage within the processes that are created by the same parent process.
- This situation has been tried to be overcome in Unix System III by the introduction of FIFO structure. FIFOs are the called ”named pipe”s. They can be used by the processes having no interaction/relation.

## Pipe/FIFO

- Pipe is destroyed with the last `close` command.
- FIFOs are deleted from the file system via `unlink` command.
- For creating and opening of a pipe: it is enough to call `pipe()` function.
- For creating and opening of a FIFO: `mkfifo()` and `open()` functions should be called in order.

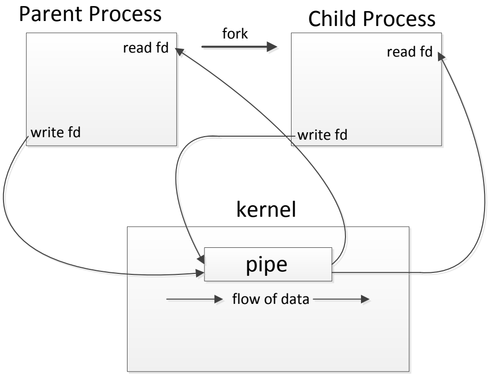
## Pipe Usage



When created within a single process.

## Pipe Usage

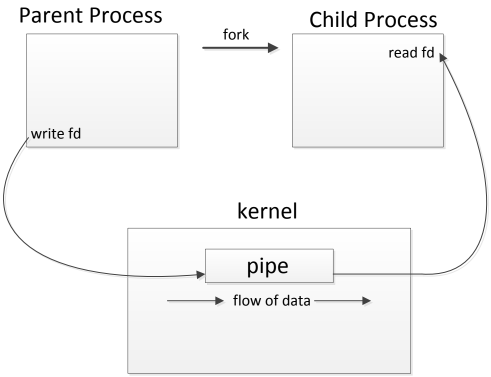
When parent process creates a child process with `fork()`: BOTH processes gain pipe's read (`pipe[0]`) and write (`pipe[1]`) descriptors.



## Pipe Usage

Afterwards, *Writer* process closes the reading end whereas *Reader* process closes the writing end.

One-way communication is set up ...



## Pipe Usage

```
<unistd.h>
int pipe(int filedes[2]);
int close(int fd);
```

- Has two flow paths.
- Normally one is used for reading whereas the other is used for writing (LINUX)
- If both are used for both reading & writing: full-duplex (SOLARIS)
- Returns 0 on successful completion of the operation, -1 on any error.
- Returns 2 file descriptors
  - `filedes[0]` : for reading
  - `filedes[1]` : for writing

## Pipe Example - 1

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#define NOFSEND 3 // number of messages
#define SOFSEND 4 // size of messages

int main(){
  int c, p[2], i;
  char send[NOFSEND][SOFSEND]={"Fee\0","Faa\0","Foo\0"}; // messages
  char rec[SOFSEND]; // buffer for receiver
  if (pipe(p) < 0) // creating pipe
    printf("Can't create a pipe.\n");
  if((c=fork()) < 0) // creating a child process
    printf("Can't fork.\n");
```

## Pipe Example - 1

```c
// parent process
else if (c > 0){
  close(p[0]); // closing reading end
  for(i=0;i<NOFSEND;i++){ // sending messages
    if (write(p[1], send[i], SOFSEND) < 0)
      printf("M: Can't write %d\n",i+1);
    else
      printf("M: I wrote %d.\n",i+1);
  }
  wait(NULL); // waiting for the child to terminate
  exit(0);
}
```

## Pipe Example - 1

```
  // child process
  else{
    sleep(1); // waiting for a second
    close(p[1]); // closing writing end
    for(i=0;i<NOFSEND;i++){ // reading messages
      if (read(p[0], &rec, SOFSEND) < 0)
        printf("C: Can't read %d\n",i+1);
      else
        printf("C: I read \"%s\"\n", rec);
    }
  }
}
```

## Pipe Example - 1, Output

```
M: I wrote 1.
M: I wrote 2.
M: I wrote 3.
 C: I read "Fee"
 C: I read "Faa"
 C: I read "Foo"
```

## Pipe Usage

Call to another process within a program:

- `popen` : Creates a pipe stream to a process within the process.
  `FILE *popen(const char *command, const char *mode);`
- `pclose` : Closes the pipe stream opened within the process.
  `int pclose(FILE *stream);`

## Pipe Example - 2

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

void main(){
  FILE *f;
  char line[80];

  // open pipe for reading
  // command: list files in current working directory
  // -l: in long format
  // -a: include . and ..
  if( (f=popen("ls -la", "r")) == NULL)
    printf("Can't open pipe.\n");

  // read data line by line and print out on the screen
  while(fgets(line, 80, f) != NULL)
    printf("%s", line);

  // close pipe
  pclose(f);
}
```

## Pipe Example - 2, Output

```
total 19
drwxrwx--- 1 root vboxsf 4096 Nis 16  2014 .
drwxr-xr-x 4 root root   4096 Sub 25 15:48 ..
-rwxrwx--- 1 root vboxsf 1040 Nis 16 12:30 1.c
-rwxrwx--- 1 root vboxsf   99 May  9  2011 2.bash
-rwxrwx--- 1 root vboxsf  413 Nis 16 13:12 2.c
-rwxrwx--- 1 root vboxsf    0 Nis 16  2014 2.txt
-rwxrwx--- 1 root vboxsf 7490 Nis 16 13:12 a.out
-rwxrwx--- 1 root vboxsf  430 Nis 16 13:01 deneme.txt
-rwxrwx--- 1 root vboxsf   90 Nis 16 12:23 exampleOutput1.txt
```

## Pipe Example - 3

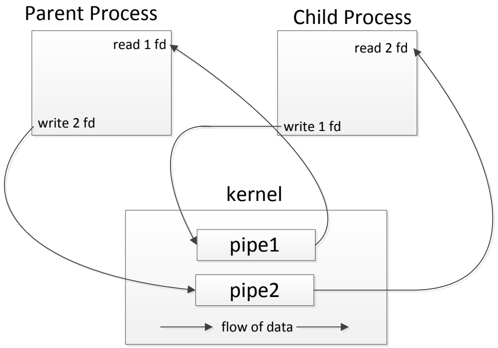
```c
#include <unistd.h>
#include <stdio.h>

void main(){
  FILE *f, *g;
  char line[80];
  // open pipe for reading
  // command: list files in current working directory.
  if( (f=popen("ls", "r")) == NULL)
    printf("Can't open pipe.\n");
  // open pipe for writing
  // command: grep (search for a pattern)
  // -i: case insensitive
  if( (g=popen("grep -i *.c", "w")) == NULL)
    printf("Can't open pipe.\n");
  // read data line by line from pipe f and write on pipe g
  while(fgets(line, 80, f) != NULL){
        printf("Read: %s", line);
    fputs(line, g);
  }
  // close pipes
  pclose(f);
  pclose(g);
}
```

## Pipe Example - 3, Output

```
Read:  3.c
Read:  3.txt
Read:  a.out
3.c
```

## Full-duplex (two-way) Pipe Usage



## Full-duplex (two-way) Pipe Usage (Fork)

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(){
  int c, p[2], q[2];
  // creating two pipes
  if (pipe(p) < 0 || pipe(q) < 0) printf("Can't create pipes.\n");
  // creating a child process
  if((c=fork()) < 0) printf("Can't fork.\n");
  else if (c > 0){ // parent process
    close(p[0]); // closing reading end of pipe p
    close(q[1]); // closing writing end of pipe q
    char r[4];
    // writing to pipe p
    if (write(p[1], "Foo\0", 4) < 0) printf("M: Can't write\n");
    printf("M: I wrote Foo.\n");
    // reading from pipe q
    if (read(q[0], &r, 4) < 0) printf("M: Can't read\n");
    printf("M: I read \"%s\"\n", r);
    wait(NULL); // waiting for the child to terminate
    exit(0);
  }
```

## Full-duplex (two-way) Pipe Usage (Fork)

```
  else{ // child process
    close(p[1]); // closing writing end of pipe p
    close(q[0]); // closing reading end of pipe q
    char r[4];
    // writing to pipe q
    if (write(q[1], "Bar\0", 4) < 0) printf("C: Can't write\n");
    printf("C: I wrote Bar.\n");
    // reading from pipe p
    if (read(p[0], &r, 4) < 0) printf("C: Can't read\n");
    printf("C: I read \"%s\"\n", r);
  }
}
```

## Full-duplex (two-way) Pipe Usage (Fork), Output

```
M: I wrote Foo.
M: I read "Bar"
C: I wrote Bar.
C: I read "Foo"
```

M: I read "Bar" before C: I wrote "Bar" -> synchronization problem (need to use mutex to `printf` just after writing to pipe)

## Full-duplex (two-way) Pipe Usage (Thread)

```c
#include <stdio.h>
#include <pthread.h>
#define NOFSEND 3 // number of messages
#define SOFSEND 4 // message size
#define NOFITER 10 // max. number of iterations
int p[2], q[2]; // pipes

void* sender(void *arg){ // sender thread handling function
  char* me=(char*)arg;
  int i;
  char send[NOFSEND][SOFSEND]={"Fee\0","Faa\0","Foo\0"};
  if((*me)=='M'){ // if arg = 'M' (mother), use p to write message
    for(i=0;i<NOFITER;i++){ // start from "Fee"
      if (write(p[1], send[i%NOFSEND], SOFSEND) < 0)
        printf("M: Can't write\n");
      printf("M: I wrote %s.\n", send[i%NOFSEND]);
    }
  }
  else{ // else (child) use q to write message
    for(i=2;i<NOFITER+2;i++){ // start from "Foo"
      if (write(q[1], send[i%NOFSEND], SOFSEND) < 0)
        printf("C: Can't write\n");
      printf("C: I wrote %s.\n", send[i%NOFSEND]);
    }
  }
}
```

## Full-duplex (two-way) Pipe Usage (Thread)

```
void* reciever(void *arg){ // receiver thread handling function
  char* me=(char*)arg;
  int i; char rec[SOFSEND];
  if((*me)=='M'){ // if arg = 'M' (mother), read message from q
    for(i=0;i<NOFITER;i++){
      if (read(q[0], &rec, SOFSEND) < 0)
        printf("M: Can't read\n");
      printf("M: I read %s.\n",rec);
    }
  }
  else{ // else (child) read message from p
    for(i=0;i<NOFITER;i++){
      if (read(p[0], &rec, SOFSEND) < 0)
        printf("C: Can't read\n");
      printf("C: I read %s.\n",rec);
    }
  }
}
```

## Full-duplex (two-way) Pipe Usage (Thread)

```
int main(){
  int c;
  pthread_t mSend,mRecv,cSend,cRecv;
  char mother='M',child='C';
  if (pipe(p) < 0 || pipe(q) < 0) // create two pipes
    printf("Can't create pipes.\n");
  if((c=fork()) < 0) printf("Can't fork.\n"); // create a child
  else if (c > 0){ // parent process
    close(p[0]); // closing reading end of pipe p
    close(q[1]); // closing writing end of pipe q
    // create two threads: a sender and a receiver
    if( pthread_create(&mSend,NULL,sender,&mother) ||
    pthread_create(&mRecv,NULL,reciever,&mother)){
      printf("error creating thread");
      return 1;
    }
    // wait until both threads terminate
    if( pthread_join(mSend,NULL) || pthread_join(mRecv,NULL) ){
      printf("error joining thread");
      return 1;
    }
    wait(NULL); // wait until child process terminates
    return 0;
  }
```

## Full-duplex (two-way) Pipe Usage (Thread)

```
  else{ // child process
    close(p[1]); // closing writing end of pipe p
    close(q[0]); // closing reading end of pipe q
    // create two threads: a sender and a receiver
    if( pthread_create(&cSend,NULL,sender,&child) ||
    pthread_create(&cRecv,NULL,reciever,&child)){
      printf("error creating thread");
      return 1;
    }
    // wait until both threads terminate
    if( pthread_join(cSend,NULL) || pthread_join(cRecv,NULL) ){
      printf("error joining thread");
      return 1;
    }
  }
}
```

## Full-duplex (two-way) Pipe Usage (Thread), Output

```
M: I wrote Fee.
M: I wrote Faa.
M: I wrote Foo.
M: I wrote Fee.
M: I wrote Faa.
M: I wrote Foo.
M: I wrote Fee.
M: I wrote Faa.
M: I wrote Foo.
M: I wrote Fee.
C: I read Fee.
C: I read Faa.
C: I read Foo.
C: I read Fee.
C: I read Faa.
M: I read Foo.
C: I read Foo.
C: I read Fee.
C: I read Faa.
C: I read Foo.
```

## Full-duplex (two-way) Pipe Usage (Thread), Output (Continues)

```
C: I read Fee.
C: I wrote Foo.
M: I read Fee.
C: I wrote Fee.
M: I read Faa.
C: I wrote Faa.
M: I read Foo.
C: I wrote Foo.
M: I read Fee.
C: I wrote Fee.
M: I read Faa.
C: I wrote Faa.
M: I read Foo.
C: I wrote Foo.
M: I read Fee.
C: I wrote Fee.
M: I read Faa.
C: I wrote Faa.
M: I read Foo.
C: I wrote Foo.
```

## FIFO Usage

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

void main(){
  int f; FILE *a, *b; char r[7];
  // creating a FIFO
  mkfifo("myFifo", 0777);
  // creating a child process
  if( (f=fork()) < 0) printf("Can't fork.\n");
  else if(f > 0){ // parent process
    a = fopen("myFifo", "w"); // write
    fputs("FooBar\0", a);
    fclose(a);
    // wait for child process to exit
    wait(NULL);
  }else{ // child process
    b = fopen("myFifo", "r"); // read
    fgets(r, 7, b);
    fclose(b);
    printf("Read: %s\n",r);
  }
  // deleting FIFO
  unlink("myFifo");
}
```

## FIFO Usage, Output

```
Read:  FooBar
```

## FIFO Usage From Command Line

**From a terminal console:**

```
musty@musty-VirtualBox:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
musty@musty-VirtualBox:~$ mkfifo myFIFO
musty@musty-VirtualBox:~$ ls > myFIFO
musty@musty-VirtualBox:~$ rm myFIFO
musty@musty-VirtualBox:~$
```

**From another terminal console:**

```
musty@musty-VirtualBox:~$ cat < myFIFO
Desktop
Documents
Downloads
Music
myFIFO
Pictures
Public
Templates
Videos
musty@musty-VirtualBox:~$
```