

Computer Operating Systems, Practice Session 11

Linux Pipe Structure

Mustafa Ersen (ersenm@itu.edu.tr)

Istanbul Technical University
34469 Maslak, İstanbul

06 May 2015

Today

Computer Operating Systems, PS 11

Pipe Structure

Pipe Examples

FIFO Examples

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.

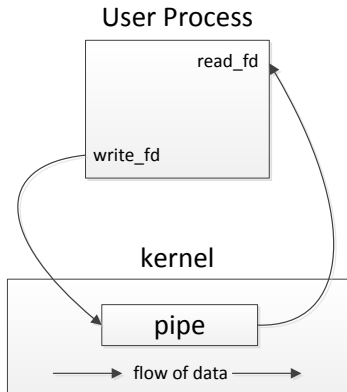
Types of Pipes

- ▶ 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.

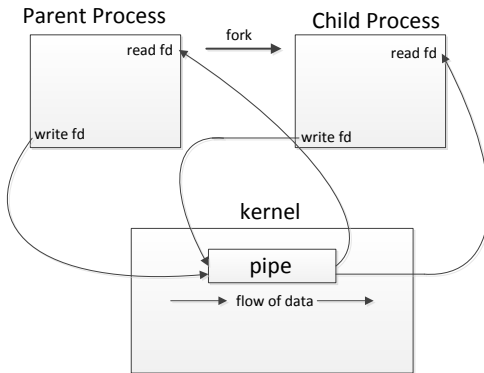
Pipe Usage



When created within a single process.

Pipe Usage

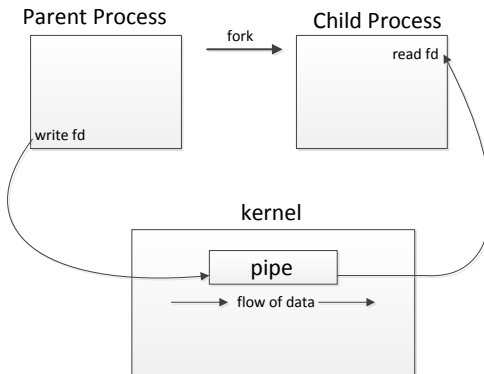
When parent process creates a child process with `fork()`: BOTH processes gain pipe's read (`pipe[0]`) and write (`pipe[1]`) descriptors.



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

```
1 #include <unistd.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4 #define NOFSEND 3 // number of messages
5 #define SOFSEND 4 // size of messages
6
7 int main(){
8     int c, p[2], i;
9     char send[NOFSEND][SOFSEND]={ "Fee\0", "Faa\0", "Foo\0"}; // messages
10    char rec[SOFSEND]; // buffer for receiver
11    if (pipe(p) < 0) // creating pipe
12        printf("Can't create a pipe.\n");
13    if((c=fork()) < 0) // creating a child process
14        printf("Can't fork.\n");
```

Pipe Example - 1

```
1  // parent process
2  else if (c > 0){
3      close(p[0]); // closing reading end
4      for(i=0;i<NOFSEND;i++){ // sending messages
5          if (write(p[1], send[i], SOFSEND) < 0)
6              printf("M: Can't write %d\n", i+1);
7          else
8              printf("M: I wrote %d.\n", i+1);
9      }
10     wait(NULL); // waiting for the child to terminate
11     exit(0);
12 }
```

Pipe Example - 1

```
1 // child process
2 else{
3     sleep(1); // waiting for a second
4     close(p[1]); // closing writing end
5     for(i=0;i<NOFSEND;i++){ // reading messages
6         if (read(p[0], &rec , SOFSEND) < 0)
7             printf("C: Can't read %d\n", i+1);
8         else
9             printf("C: I read \"%s\"\n", rec);
10    }
11 }
12 }
```

Pipe Example - 1, Output

```
1 M: I wrote 1.  
2 M: I wrote 2.  
3 M: I wrote 3.  
4 C: I read "Fee"  
5 C: I read "Faa"  
6 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

```
1 #include <unistd.h>
2 #include <stdio.h>
3
4 void main(){
5     FILE *f;
6     char line[80];
7
8     // open pipe for reading
9     // command: list files in current working directory
10    // -l: in long format
11    // -a: include . and ..
12    if( (f=popen("ls -la", "r")) == NULL)
13        printf("Can't open pipe.\n");
14
15    // read data line by line and print out on the screen
16    while(fgets(line, 80, f) != NULL)
17        printf("%s", line);
18
19    // close pipe
20    pclose(f);
21 }
```


Pipe Example - 2, Output

```

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

```

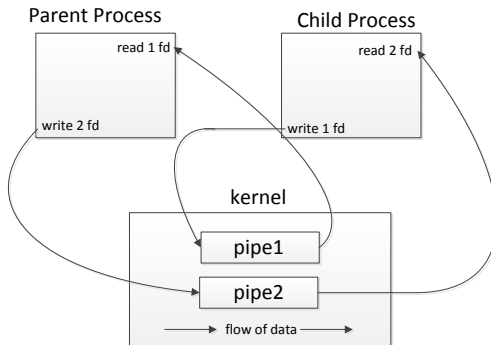
Pipe Example - 3

```
1 #include <unistd.h>
2 #include <stdio.h>
3
4 void main(){
5     FILE *f, *g;
6     char line[80];
7     // open pipe for reading
8     // command: list files in current working directory.
9     if( (f=popen("ls", "r")) == NULL)
10         printf("Can't open pipe.\n");
11     // open pipe for writing
12     // command: grep (search for a pattern)
13     // -i: case insensitive
14     if( (g=popen("grep -i *.c", "w")) == NULL)
15         printf("Can't open pipe.\n");
16     // read data line by line from pipe f and write on pipe g
17     while(fgets(line, 80, f) != NULL){
18         printf("Read: %s", line);
19         fputs(line, g);
20     }
21     // close pipes
22     pclose(f);
23     pclose(g);
24 }
```

Pipe Example - 3, Output

```
1 Read: 3.c  
2 Read: 3.txt  
3 Read: a.out  
4 3.c
```

Full-duplex (two-way) Pipe Usage



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

```
1 #include <unistd.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4
5 int main(){
6     int c, p[2], q[2];
7     // creating two pipes
8     if (pipe(p) < 0 || pipe(q) < 0) printf("Can't create pipes.\n");
9     // creating a child process
10    if((c=fork()) < 0) printf("Can't fork.\n");
11    else if (c > 0){ // parent process
12        close(p[0]); // closing reading end of pipe p
13        close(q[1]); // closing writing end of pipe q
14        char r[4];
15        // writing to pipe p
16        if (write(p[1], "Foo\0", 4) < 0) printf("M: Can't write\n");
17        printf("M: I wrote Foo.\n");
18        // reading from pipe q
19        if (read(q[0], &r, 4) < 0) printf("M: Can't read\n");
20        printf("M: I read \"%s\"\n", r);
21        wait(NULL); // waiting for the child to terminate
22        exit(0);
23    }
```

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

```
1  else{ // child process
2      close(p[1]); // closing writing end of pipe p
3      close(q[0]); // closing reading end of pipe q
4      char r[4];
5      // writing to pipe q
6      if (write(q[1], "Bar\0", 4) < 0) printf("C: Can't write\n");
7      printf("C: I wrote Bar.\n");
8      // reading from pipe p
9      if (read(p[0], &r, 4) < 0) printf("C: Can't read\n");
10     printf("C: I read \"%s\"\n", r);
11 }
12 }
```

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

```
1 M: I wrote Foo.  
2 M: I read "Bar"  
3 C: I wrote Bar.  
4 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)

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #define NOFSEND 3 // number of messages
4 #define SOFSEND 4 // message size
5 #define NOFITER 10 // max. number of iterations
6 int p[2], q[2]; // pipes
7
8 void* sender(void *arg){ // sender thread handling function
9     char* me=(char*)arg;
10    int i;
11    char send[NOFSEND][SOFSEND]={ "Fee\0", "Faa\0", "Foo\0"};
12    if((*me)=='M'){ // if arg = 'M' (mother), use p to write message
13        for(i=0;i<NOFITER;i++){ // start from "Fee"
14            if (write(p[1], send[i%NOFSEND], SOFSEND) < 0)
15                printf("M: Can't write\n");
16            printf("M: I wrote %s.\n", send[i%NOFSEND]);
17        }
18    }
19    else{ // else (child) use q to write message
20        for(i=2;i<NOFITER+2;i++){ // start from "Foo"
21            if (write(q[1], send[i%NOFSEND], SOFSEND) < 0)
22                printf("C: Can't write\n");
23            printf("C: I wrote %s.\n", send[i%NOFSEND]);
24        }
25    }
26 }
```


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

```
1 void* reciever(void *arg){ // receiver thread handling function
2     char* me=(char*)arg;
3     int i; char rec[SOFSEND];
4     if((*me)=='M'){ // if arg = 'M' (mother), read message from q
5         for(i=0;i<NOFITER;i++){
6             if (read(q[0], &rec, SOFSEND) < 0)
7                 printf("M: Can't read\n");
8             printf("M: I read %s.\n",rec);
9         }
10    }
11    else{ // else (child) read message from p
12        for(i=0;i<NOFITER;i++){
13            if (read(p[0], &rec, SOFSEND) < 0)
14                printf("C: Can't read\n");
15            printf("C: I read %s.\n",rec);
16        }
17    }
18 }
```

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

```

1  int main(){
2      int c;
3      pthread_t mSend,mRecv,cSend,cRecv;
4      char mother='M',child='C';
5      if (pipe(p) < 0 || pipe(q) < 0) // create two pipes
6          printf("Can't create pipes.\n");
7      if((c=fork()) < 0) printf("Can't fork.\n"); // create a child
8      else if (c > 0){ // parent process
9          close(p[0]); // closing reading end of pipe p
10         close(q[1]); // closing writing end of pipe q
11         // create two threads: a sender and a receiver
12         if( pthread_create(&mSend,NULL,sender,&mother) ||
13            pthread_create(&mRecv,NULL,receiver,&mother)){
14             printf("error creating thread");
15             return 1;
16         }
17         // wait until both threads terminate
18         if( pthread_join(mSend,NULL) || pthread_join(mRecv,NULL) ){
19             printf("error joining thread");
20             return 1;
21         }
22         wait(NULL); // wait until child process terminates
23         return 0;
24     }

```

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

```
1  else{ // child process
2      close(p[1]); // closing writing end of pipe p
3      close(q[0]); // closing reading end of pipe q
4      // create two threads: a sender and a receiver
5      if( pthread_create(&cSend, NULL, sender, &child) ||
6          pthread_create(&cRecv, NULL, reciever, &child)){
7          printf("error creating thread");
8          return 1;
9      }
10     // wait until both threads terminate
11     if( pthread_join(cSend, NULL) || pthread_join(cRecv, NULL) ){
12         printf("error joining thread");
13         return 1;
14     }
15 }
16 }
```

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

```
1 M: I wrote Fee.  
2 M: I wrote Faa.  
3 M: I wrote Foo.  
4 M: I wrote Fee.  
5 M: I wrote Faa.  
6 M: I wrote Foo.  
7 M: I wrote Fee.  
8 M: I wrote Faa.  
9 M: I wrote Foo.  
10 M: I wrote Fee.  
11 C: I read Fee.  
12 C: I read Faa.  
13 C: I read Foo.  
14 C: I read Fee.  
15 C: I read Faa.  
16 M: I read Foo.  
17 C: I read Foo.  
18 C: I read Fee.  
19 C: I read Faa.  
20 C: I read Foo.
```

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

```
1 C: I read Fee.  
2 C: I wrote Foo.  
3 M: I read Fee.  
4 C: I wrote Fee.  
5 M: I read Faa.  
6 C: I wrote Faa.  
7 M: I read Foo.  
8 C: I wrote Foo.  
9 M: I read Fee.  
10 C: I wrote Fee.  
11 M: I read Faa.  
12 C: I wrote Faa.  
13 M: I read Foo.  
14 C: I wrote Foo.  
15 M: I read Fee.  
16 C: I wrote Fee.  
17 M: I read Faa.  
18 C: I wrote Faa.  
19 M: I read Foo.  
20 C: I wrote Foo.
```

FIFO Usage

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 void main(){
5     int f; FILE *a, *b; char r[7];
6     // creating a FIFO
7     mkfifo("myFifo", 0777);
8     // creating a child process
9     if( (f=fork()) < 0) printf("Can't fork.\n");
10    else if(f > 0){ // parent process
11        a = fopen("myFifo", "w"); // write
12        fputs("FooBar\0", a);
13        fclose(a);
14        // wait for child process to exit
15        wait(NULL);
16    }else{ // child process
17        b = fopen("myFifo", "r"); // read
18        fgets(r, 7, b);
19        fclose(b);
20        printf("Read: %s\n", r);
21    }
22    // deleting FIFO
23    unlink("myFifo");
24 }
```

FIFO Usage, Output

```
1 Read: FooBar
```

FIFO Usage From Command Line

From a terminal console:

```
1 musty@musty-VirtualBox:~$ ls
2 Desktop Documents Downloads Music Pictures Public Templates Videos
3 musty@musty-VirtualBox:~$ mkfifo myFIFO
4 musty@musty-VirtualBox:~$ ls > myFIFO
5 musty@musty-VirtualBox:~$ rm myFIFO
6 musty@musty-VirtualBox:~$
```

From another terminal console:

```
1 musty@musty-VirtualBox:~$ cat < myFIFO
2 Desktop
3 Documents
4 Downloads
5 Music
6 myFIFO
7 Pictures
8 Public
9 Templates
10 Videos
11 musty@musty-VirtualBox:~$
```