Chapter 7

Inter-Process Communication (IPC) Mechanisms

Pipes, Message Queues, Shared Memory

Print Version of Lectures Notes of *Operating Systems*

Technical University of Cluj-Napoca (UTCN)
Computer Science Department

Adrian Coleșa

May 13, 2020

7.1

Purpose and Contents

The purpose of this chapter

- Presents the main mechanisms for communication between processes
- Presents different examples of using them

7.2

Bibliography

- A. Colesa, I. Ignat, Z. Somodi, *Sisteme de Operare. Chestiuni Teoretice si Practice*, 2007, Chapters 8 and 11, p. 105 – 117, p. 149 – 169 (Romanian only — see the pdf file on moodle page). For the english version see the html pages also on moodle page at Lecture resources.

7.3

Contents

7.4

IPC Mechanisms

- indirect communication
  - synchronization mechanisms, e.g. semaphores
- direct communication
  - wait/exit system calls
  - pipes
  - message queues
  - shared memory
  - sockets 7.5

1 Pipe (FIFO Files)

1.1 Characteristics and Communication Principles

Characteristics of a pipe

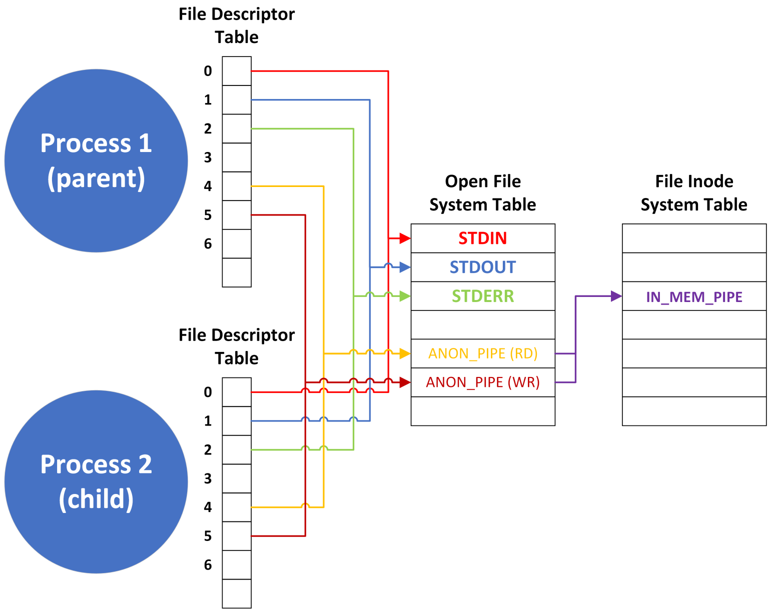
- provided **as a file**
  - accessed as a regular file (open, read, write, close)
- though, it is a **special file**
  - controlled and managed by the OS
  - exposed in a special format (FIFO)
  - imposed special rules to work with it
- it is **an IPC mechanism**
  - more processes (any number) can communicate through it
  - in a synchronized way 7.6

Communication Principles

- **based on producer/consumer** synchronization pattern
- **data access particularities** (differences by regular files)
  - **FIFO** principle ⇒ there is **no seek** in the pipe
  - once read, data from pipe is no more available
  - circular fix-sized buffer ⇒ **no end of file**
- **synchronization rules**
  1. reading from an empty pipe blocks the calling processes (consumer)
     - though, trying to read more bytes than available will not block
     - `read` returns the number of read bytes
  2. writing into a full pipe blocks the calling processes (producer)
- **synchronization exceptions**
  1. reading from an empty pipe with no writer connected returns immediately as if end of file was detected
  2. writing into a full pipe with no reader connected returns immediately with an error 7.7

Uni- and Bi-directional Pipes

- normally **a user convention**, but sometime an OS support
- consider the simple case of one producer and one consumer
- communication patterns, i.e. "**pipe usage types**"
  - uni-directional: used for a one-way communication
  - bi-directional: used for both directions
- in practice, though, there could be any number of consumers and producers
  - ⇒ a sort of **N-directional pipes**
  - though, the OS could not manage making distinction between "directions", i.e. communication between two particular processes 7.8



Types of Pipes

- **nameless** (**anonymous**) pipes
- pipes **with name** (FIFO files)

7.9

Nameless (Anonymous) Pipes

- created with a special system call
- **have no name**
  - ⇒ not visible as an element (file) in the file system
  - cannot be opened (like other visible files)
- not accessible, but to their creator process
  - just an in-memory OS data structure (and memory buffer)
  - invisible file automatically opened by the OS for the creator process
  - accessible through file handles (i.e. descriptors)
- used normally for **communication between processes in parent-child relationship**
  - based on the inheritance property
  - i.e. child inherits from its parent pipe's handles (file descriptors)
- ⇒ **anonymous pipes must be created before child creation**

7.10

Anonymous Pipe Management in File System Tables

7.11

Pipes With Name (FIFO Files)

- created by a special system call
- **have a name**
  - ⇒ visible in the file system
  - can be opened like any other files
- **any process** that wants to use them should **open them**

7.12

1.2 Examples

Linux Anonymous Pipes

- system call

```
int pipe(int fd[2]);
```

- usage

```
int fd[2];
int nr1 = 10, nr2;
pipe(fd);    // ex. fd[0] = 3 (for read)
             // ex. fd[1] = 4 (for write)
if (fork() == 0) { // child
    close(fd[0]);  // close access to pipe for read
    write(fd[1], &nr1, sizeof(int));
} else {           // parent
    close(fd[1]);  // close access to pipe for write
    read(fd[0], &nr2, sizeof(int));
}
```

7.13

Windows Anonymous Pipes

- Windows Win32 Function

```
BOOL WINAPI CreatePipe(
__out     PHANDLE hReadPipe,
__out     PHANDLE hWritePipe,
__in_opt  LPSECURITY_ATTRIBUTES lpPipeAttributes,
__in      DWORD nSize
);
```

- usage

```
HANDLE hReadPipe = NULL;
HANDLE hWritePipe = NULL;
DWORD dwRead, dwWritten;
CHAR chBuf[BUFSIZE];
SECURITY_ATTRIBUTES saAttr;
saAttr.nLength = sizeof(SECURITY_ATTRIBUTES);
saAttr.bInheritHandle = TRUE;
saAttr.lpSecurityDescriptor = NULL;

CreatePipe(&hReadPipe, &hWritePipe, &saAttr, 0);

WriteFile(hWritePipe, chBuf, 10, &dwWritten, NULL);
ReadFile(hReadPipe, chBuf, 10, &dwRead, NULL);
```

7.14

Linux FIFO Files

- system call

```
int mkfifo(char *name, mode_t permissions);
```

- usage

```
int fdPipe;
int nr1 = 10, nr2;

mkfifo("FIFO", 0644);
fdPipe = open("FIFO", O_RDWR);

write(fdPipe, &nr1, sizeof(int));
read(fdPipe, &nr2, sizeof(int));
```

7.15

Windows FIFO Files. Server Process

```
LPTSTR lpszPipename = TEXT("\\\\.\\pipe\\mynamedpipe");

hPipe = CreateNamedPipe(
            lpszPipename,              // pipe name
            PIPE_ACCESS_DUPLEX,        // read/write access
            PIPE_TYPE_MESSAGE |        // message type pipe
            PIPE_READMODE_MESSAGE |    // message-read mode
            PIPE_WAIT,                 // blocking mode
            PIPE_UNLIMITED_INSTANCES,  // max. instances
            BUFSIZE,                   // output buffer size
            BUFSIZE,                   // input buffer size
            0,                         // client time-out
            NULL);                     // default security attribute

fConnected = ConnectNamedPipe(hPipe, NULL);

ReadFile(
    hPipe,          // handle to pipe
    pchRequest,     // buffer to receive data
    BUFSIZE*sizeof(TCHAR), // size of buffer
    &cbBytesRead,   // number of bytes read
    NULL);          // not overlapped I/O
```

7.16

Windows FIFO Files. Client Process

```
LPTSTR lpszPipename = TEXT("\\\\.\\pipe\\mynamedpipe");

hPipe = CreateFile(
            lpszPipename,   // pipe name
            GENERIC_READ |  // read and write access
            GENERIC_WRITE,
            0,              // no sharing
            NULL,           // default security attributes
            OPEN_EXISTING,  // opens existing pipe
            0,              // default attributes
            NULL);

WriteFile(
    hPipe,                  // pipe handle
    lpvMessage,             // message
    (lstrlen(lpvMessage)+1)*sizeof(TCHAR), // message length
    &cbWritten,             // bytes written
    NULL);                  // not overlapped
```

7.17

Linux Uni-directional Pipe

```
// First Process
main()
{
    // create the pipe
    // must be done before any process try opening the pipe
    mkfifo("FIFO", 0600);

    // open the pipe for WRITE only
    // the process will block until the second process open the pipe for READ
    int fdW = open("FIFO", O_WRONLY);

    // write into pipe
    // unblock second process
    write(fdW, "1", 1);
}

// Second Process
main()
{
    // open the pipe for READ only
    // unblock the first process
    int fdR = open("FIFO", O_RDONLY);

    // read from pipe
    // block until first process succeeds writing into
    read(fdR, &c, 1);
}
```

7.18

Linux Bi-directional Pipe. Intended Scenario

```
// First Process
main()
{
    mkfifo("FIFO", 0600);
    int fdW = open("FIFO", O_WRONLY);
    int fdR = open("FIFO", O_RDONLY);

    write(fdW, "1", 1);

    read(fdR, &c, 1);
}

// Second Process
main()
{
    int fdR = open("FIFO", O_RDONLY);
    int fdW = open("FIFO", O_WRONLY);

    read(fdR, &c, 1);
    // do something with the read data
    write(fdW, "2", 1);
}
```

7.19

Linux Bi-directional Pipe. Wrong Scenario

```
// First Process
main()
{
    mkfifo("FIFO", 0600);
    int fdW = open("FIFO", O_WRONLY);
    int fdR = open("FIFO", O_RDONLY);

    write(fdW, "1", 1);

    read(fdR, &c, 1);  // problem: could read what it has just written
}

// Second Process
main()
{
    int fdR = open("FIFO", O_RDONLY);
    int fdW = open("FIFO", O_WRONLY);
```

```
    read(fdR, &c, 1);  // problem: could be blocked forever
    // do something with the read data
    write(fdW, "2", 1);
}
```

7.20

Linux Bi-directional Communication. Functional Solution

```
// First Process
main()
{
    mkfifo("FIFO1", 0600);
    mkfifo("FIFO2", 0600);
    int fdW = open("FIFO1", O_WRONLY);
    int fdR = open("FIFO2", O_RDONLY);

    write(fdW, "1", 1); // writes on FIFO1

    read(fdR, &c, 1); // blocks until data becomes available on FIFO2
}

// Second Process
main()
{
    int fdR = open("FIFO1", O_RDONLY);
    int fdW = open("FIFO2", O_WRONLY);

    read(fdR, &c, 1); // blocks until data becomes available on FIFO1
    // do something with read data
    write(fdW, "2", 1); // writes on FIFO2
}
```

7.21

Command Line Pipe (2 commands)

```
// Command interpreter's handling of command lines like
// cmd_0 | cmd_1
// e.g. "cat file.txt | wc -l"

// prepare for handling an anonymous pipe
// used between cmd_0 and cmd_1
int fd[2];

// command paths (names) and arguments
char cmd[2][256];
char argv[2][256][256];

prepare_cmds_and_args(cmd, argv);

// pipe creation must be done
// before creating processes that use that pipe
pipe(fd);
```

7.22

Command Line Pipe (2 commands) (cont.)

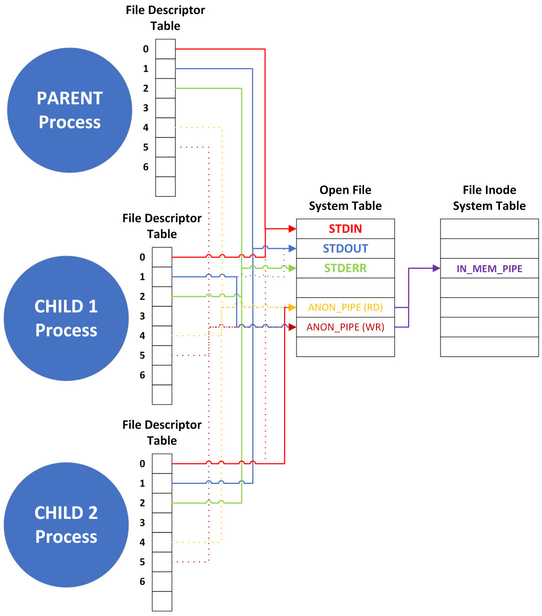
```
// first child process creation
if (fork() == 0) {
    // child executing cmd_0
    close(fd[0]);     // not reading from pipe
    dup2(fd[1], 1);   // redirect 1 (STDOUT) to pipe
    close(fd[1]);     // not using anymore the pipe explicitly

    // load command code
    execvp(cmd[0], argv[0]);
    exit(1);          // executed only when execvp fails
}
```

7.23

Command Line Pipe (2 commands) (cont.)

```
// second child process creation
if (fork() == 0) {
    // child executing cmd_1
    close(fd[1]);     // not writing into pipe
    dup2(fd[0], 0);   // redirect 0 (STDIN) to pipe
    close(fd[0]);     // not using anymore the pipe explicitly
                      // critical to be done
```




```
    // load command code
    execvp(cmd[1], argv[1]);
    exit(1);          // executed only when execvp fails
}

// parent
// not using the pipe, so close it
close(fd[0]);
close(fd[1]); // critical to be done

wait(NULL);
wait(NULL);
```

7.24

Two Commands Linked by an Anonymous Pipe 7.25

Command Line Pipe (n commands)

- command line syntax

```
cmd_0 | cmd_1 | ... | cmd_n-1
```

- command interpreter (shell) code

```
// prepare for handling n-1 pipes
// pipe_0 between cmd_0 and cmd_1
//  fd[0][0] for reading from pipe_0
//  fd[0][1] for writing into pipe_0
// ....
int fd[n][2];

// command paths (names) and arguments
char cmd[n][256];
char argv[n][256][256];

prepare_cmds_and_args(cmd, argv);

for (i=0; i<n; i++) {

    // pipe creation must be done
    // before creating processes that use that pipe
    // e.g. pipe_0 before cmd_0 and cmd_1
    if (i < n-1)
        pipe(fd[i]);

    // child process creation
    if (fork() == 0) {
        // if not cmd_0
        if (i > 0) {
```

```
                close(fd[i-1][1]);
                dup2(fd[i-1][0], 0);
                close(fd[i-1][0]);
            }

            // if not cmd_n-1
            if (i < n-1) {
            close(fd[i][0]);
            dup2(fd[i][1], 1);
            close(fd[i][1]);
            }

            // load command code
            execvp(cmd[i], argv[i]);
        }
        else {
            close(fd[i][0]);
            close(fd[i][1]);
        }
    }
```

7.26

Practice (1)

1. There are three processes running into the system, whose code is given below. Change and complete the code of the three processes (writing all the code in just one C file and adding the code to create the processes), such that the communication to be possible and the contents of the buffer to be
   (a) "ab" or
   (b) "ba"

   at the end of the execution of the three processes.

```
// Process P1
int fd[2];

pipe(fd);
...
write(fd[1], "a", 1);
```

```
// Process P2

...
...
write(fd[1], "b", 1);
...
```

```
// Process P3
char buf[2];
...
read(fd[0],&buf[0],1);
read(fd[0],&buf[1],1);
...
```

7.27

Practice (2)

2. Write the C code/pseudo-code to find out the size of
   (a) an anonymous pipe;
   (b) a named pipe.

7.28

Practice (3)

3. Implement the semaphore's primitives `P()` and `V()` using pipes.

7.29

2 Message Queues

2.1 Characteristics and Communication Principles

Characteristics

- a more specialized pipe
- group bytes in messages, making distinction between messages
- different types (labels) of messages
- processes can get messages of specified type from queue

7.30

Communication Principles

- same as pipe, though sometimes FIFO order can be broken, when a message of a specified type is requested 7.31

2.2 Examples

Two-Way Communication

- first process

```
struct msg {
    long type;
    int data;
} msg1, msg2;

int msgId = msgget(10000, IPC_CREAT | 0600);
msg1.type = 1;
msg1.data = getpid();

// send a message of type 1
msgsnd(msgId, &msg1, sizeof(msg1) - sizeof(long), 0);

// gets a message of type 2
msgrcv(msgID, &msg2, sizeof(msg1) - sizeof(long), 2, 0);
```

- second process

```
struct msg {
    long type;
    int data;
} msg1, msg2;

int msgId = msgget(10000, 0);
msg2.type = 2;
msg2.data = getpid();

// gets a message of type 1
msgrcv(msgID, &msg2, sizeof(msg1) - sizeof(long), 1, 0);

// send a message of type 2
msgsnd(msgId, &msg1, sizeof(msg1) - sizeof(long), 0);
```

7.32

3 Shared Memory

3.1 Characteristics and Communication Principles

Characteristics

- a common (shared) memory area belonging to more process address spaces
- created with a special system call
- managed by the OS
- the fastest IPC mechanism 7.33

Principles

- no need for special system calls for communication (only for creation)
- used as any process memory area referenced by a pointer
- sequence of bytes, whose structure is known by the collaborating processes
- the concurrent accesses to it is NOT synchronized by the OS ⇒ the processes SHOULD make this 7.34

3.2 Examples

System V Shared Memory

- first process

```
int shm_id = shmget(10000, sizeof(int), IPC_CREAT | 0600);
int *i = shmat(shm_id, 0, 0);
*i = 10; // write shared memory -> "send" information
```

- second process

```
int shmId = shmget(10000, 0, 0);
int *j = shmat(shm_id, 0, 0);
printf("Received␣info␣is␣%d\n", *j);  // read the shared memory -> "receive" information
```

7.35

POSIX Shared Memory

- first process

```
int main (int argc, char **argv)
{
	int shm_id, *pInt = NULL;

	shm_unlink("/my_shm");
	shm_id = shm_open("/my_shm", O_CREAT | O_EXCL | O_RDWR, 0666);
	if (shm_id < 0) {
		perror("Cannot create the shared memory");
		exit(1);
	}

	ftruncate(shm_id, sizeof(int));

	pInt = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, shm_id, 0);
	if (pInt == NULL) {
		perror("Cannot map the shared memory");
		exit(2);
	}

	close(shm_id);

	*pInt = 100;

	munmap(pInt, sizeof(int));
}
```

- second process

```
int main (int argc, char **argv)
{
	int shm_id;
	int *pInt = NULL;

	shm_id = shm_open("/my_shm", O_RDWR, 0);
	if (shm_id < 0) {
		perror("Cannot open the shared memory");
		exit(1);
	}

	pInt = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, shm_id, 0);
	if (pInt == NULL) {
		perror("Cannot map the shared memory");
		exit(2);
	}

	close(shm_id);

	printf("shm = %d\n", *pInt);

	munmap(pInt, sizeof(int));
}
```

7.36

4 Conclusions

What we talked about

- inter-process communication mechanisms (IPC)
- pipes
  - data handled based on FIFO principle
  - synchronized based on the producer-consumer pattern
  - with name vs anonymous (i.e. nameless)
  - unidirectional vs bidirectional
- message queues
  - similar to pipes, yet more specialized

  - make distinction between messages
  - messages could be labeled
- shared memory
  - explicitly share memory between processes
  - accessed using pointers, like dynamically allocated memory
  - communication: write and read from the shared memory

7.37

Lessons Learned

1. by default processes are isolated, i.e. share nothing
2. yet, they could communicate using explicitly designed mechanisms, i.e. IPC mechanisms
3. pipes and message queues are synchronized IPC mechanisms
4. shared memory is the fastest IPC mechanism, yet provides no synchronization

7.38