OPERATING SYSTEMS

Lecture # 6

Razi Uddin

INTER-PROCESS COMMUNICATION (IPC)

IPC provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.

MESSAGE PASSING SYSTEM

- The function of a message system is to allow processes to communicate without the need to resort to shared data.

- Messages sent by a process may be of either fixed or variable size.

- If processes P and Q want to communicate, a communication link must exist between them and they must send messages to and receive messages from each other through this link.

MESSAGE PASSING SYSTEM

Methods for logically implementing a link and the send and receive options:

- Direct or indirect communication
- Symmetric or asymmetric communication
- Automatic or explicit buffering
- Send by copy or send by reference
- Fixed-size or variable size messages

DIRECT COMMUNICATION

With direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication.

- The send and receive primitives are defined as:
- Send(P, message) – send a message to process P
- Receive(Q, message) – receive a message from process Q.

DIRECT COMMUNICATION

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate.
- The processes need to know only each other's identity to communicate
- A link is associated with exactly two processes.
- Exactly one link exists between each pair of processes.

DIRECT COMMUNICATION

Unlike this symmetric addressing scheme, a variant of this scheme employs asymmetric addressing, in which the recipient is not required to name the sender.

- Send(P, message) – send a message to process P
- Receive(id, message) – receive a message from any process; the variable id is set to the name of the process with which communication has taken place.

INDIRECT COMMUNICATION

- With indirect communication, messages can be sent to and received from mailboxes.
- Here, two processes can communicate only if they share a mailbox.
- The send and receive primitives are defined as:
- Send(A, message) – send a message to mailbox A.
- Receive(A, message) – receive a message from mailbox A.

INDIRECT COMMUNICATION

- A communication link in this scheme has the following properties:
- A link is established between a pair of processes only if both members have a shared mailbox.
- A link is associated with more than two processes.
- A number of different links may exist between each pair of communicating processes, with each link corresponding to one mailbox.

SYNCHRONIZATION

- Communication between processes takes place by calls to send and receive primitives (i.e., functions).
- Message passing may be either blocking or non-blocking also called synchronous and asynchronous.

Blocking send—The sending process is blocked until the receiving process or the mailbox receives the message.

Non-blocking send—The sending process sends the message and resumes operation.

Blocking receiver—The receiver blocks until a message is available.

Non-blocking receiver—The receiver receives either a valid message or a null.

BUFFERING

- Whether the communication is direct or indirect, messages exchanged by the processes reside in a temporary queue. This queue can be implemented in three ways:
- **Zero Capacity:** The queue has a maximum length zero, thus the link cannot have any messages waiting in it. In this case, the sender must block until the message has been received.
- **Bounded Capacity:** This queue has finite length n; thus at most n messages can reside in it. If the queue is not full when a new message is sent, the latter is placed in the queue and the sender resumes operation. If the queue is full, the sender blocks until space is available.
- **Unbounded Capacity:** The queue has infinite length; thus the sender never blocks.

UNIX/LINUX IPC TOOLS

UNIX and Linux operating systems provide many tools for inter-process communication.

- Pipe
- Named pipe (FIFO)
- BSD Socket
- TLI
- Message queue
- Shared memory
- Etc.

OPEN() SYSTEM CALL

- The open() system call is used to open or create a file. Its synopsis is as follows:

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

OPEN() SYSTEM CALL

- The call converts a pathname into a file descriptor (a small, non-negative integer for use in subsequent I/O as with read, write, etc.).
- When the call is successful, the file descriptor returned will be the lowest file descriptor not currently open for the process.
- This system call can also specify whether read or write will be blocking or non-blocking.
- The ‘oflag’ argument specifies the purpose of opening the file and ‘mode’ specifies permission on the file if it is to be created.
- ‘oflag’ value is constructed by ORing various flags: O_RDONLY, O_WRONLY, O_RDWR, O_NDELAY (or O_NONBLOCK), O_APPEND, O_CREAT, etc

READ SYSTEM CALL

read()—attempts to read up to count bytes from file descriptor fd into the buffer starting at buf. If count is zero, read() returns zero and has no other results. If count is greater than SSIZE_MAX, the result is unspecified. On success, read() returns the number of bytes read (zero indicates end of file) and advances the file position pointer by this number.

```
#include<unistd.h>

ssize_t  read(int fd, void *buf, size_t count);
```

WRITE SYSTEM CALL

write()—attempts to write up to count bytes to the file referenced by the file descriptor fd from the buffer starting at buf. On success, write() returns the number of bytes written (zero indicates nothing was written) and advances the file position pointer by this number. On error, write() returns -1, and errno is set appropriately. If count is zero and the file descriptor refers to a regular file, 0 will be returned without causing any other effect.

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

CLOSE SYSTEM CALL

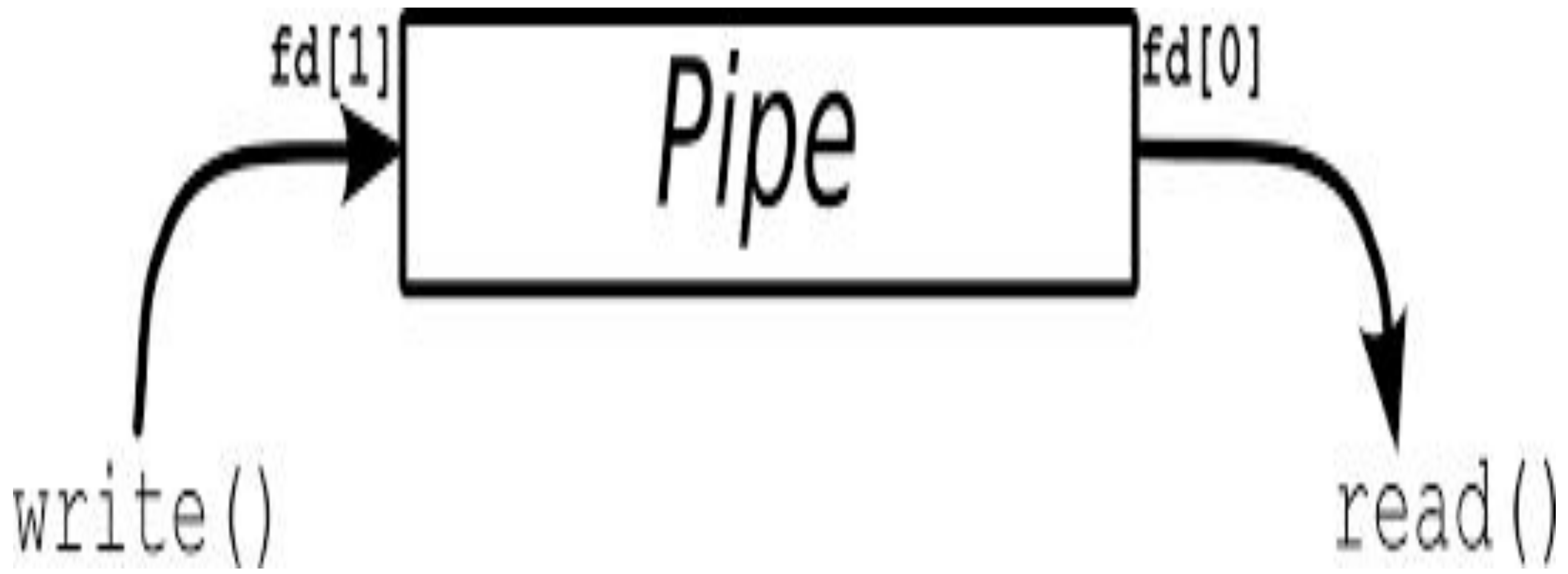
close()—closes a file descriptor, so that it no longer refers to any file and may be reused. If fd is the last copy of a particular file descriptor the resources associated with it are freed; if the descriptor was the last reference to a file which has been removed using unlink(2) the file is deleted. close() returns zero on success, or -1 if an error occurred.

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

PIPES

- A UNIX/Linux pipe can be used for IPC between related processes on a system.
- Communicating processes typically have sibling or parent-child relationship.
- At the command line, a pipe can be used to connect the standard output of one process to the standard input of another.
- Pipes provide a method of one-way communication and for this reason may be called half-duplex pipes.
- The pipe() system call creates a pipe and returns two file descriptors, one for reading and second for writing.
- The files associated with these file descriptors are streams and are both opened for reading and writing.
- Naturally, to use such a channel properly, one needs to form some kind of protocol in which data is sent over the pipe.
- Also, if we want a two-way communication, we'll need two pipes.

PIPES



PIPES

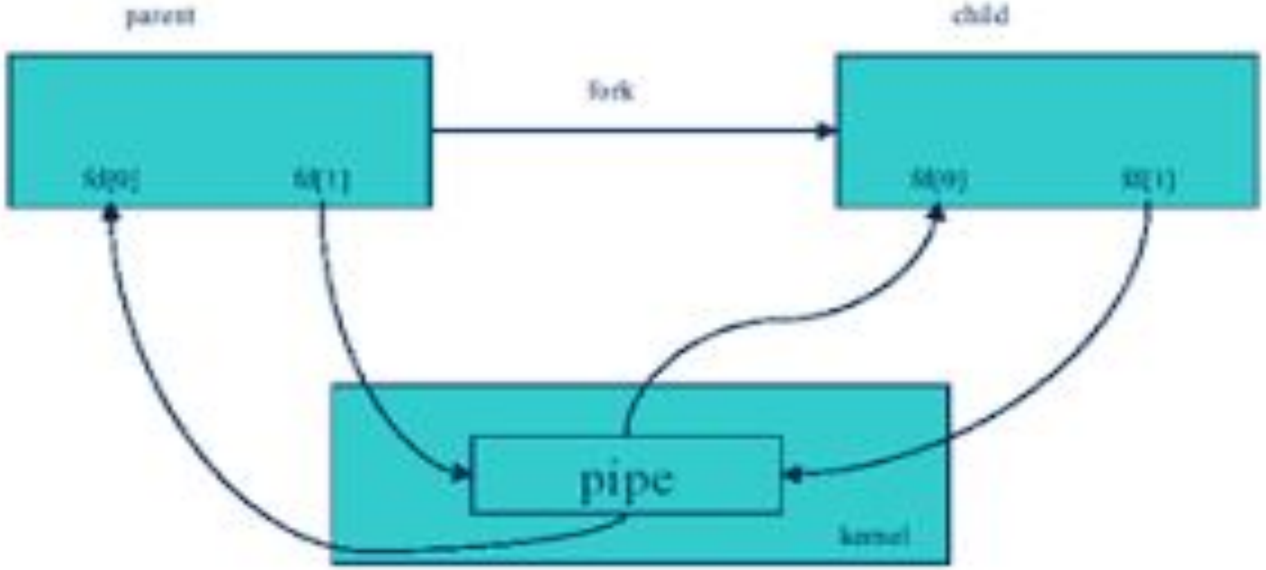
- The system assures us of one thing: the order in which data is written to the pipe, is the same order as that in which data is read from the pipe.
- The system also assures that data won't get lost in the middle, unless one of the processes (the sender or the receiver) exits prematurely.
- The pipe() system call is used to create a read-write pipe that may later be used to communicate with a process we'll fork off.
- The synopsis of the system call is:

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

PIPES

- fd[0] is the file descriptor for the read end of the pipe (i.e., the descriptor to be used with the read system call),
- whereas fd[1] is the file descriptor for the write end of the pipe. (i.e., the descriptor to be used with the write system call).
- The function returns -1 if the call fails.
- A pipe is a bounded buffer and the maximum data written is PIPE_BUF, defined in <sys/param.h> in UNIX and in <linux/param.h> in Linux as 5120 and 4096, respectively.

PIPES



Half-duplex pipe after a fork