Lecture 7 — Inter-Process Communication

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

September 6, 2017

IPC Motivation

When 2+ processes would like to co-ordinate/exchange data the mechanism is called inter-process communication.

If a process shares data with another process in the system, the operating system will provide some facilities to make this possible.

The motivations for inter-process communication are fairly obvious.

IPC Preliminaries

Before proceeding, we need to define some things.

It is the transfer of data from one process to another.

The data being transferred is typically referred to as the *message*.

The process sending that message is the *sender*.

The process receiving it will be the *receiver*.

This terminology may seem (painfully) obvious.

IPC: What to Send

The processes involved must have some agreement on:
- What data a message should contain; and
- The way the data is formatted.

There may be defined standards, e.g., XML.

The processes themselves must be aware the message is in XML format.

How this agreement is made falls outside the purview of the OS.

Messages: (A)synchronous

Sending and receiving of messages may be either synchronous or asynchronous.

Synchronous Send: the sender sends the message and then is blocked from proceeding until the message is received.

Asynchronous Send: the sender can post the message and then carry on.

Synchronous Receive: the receiver is blocked until it receives a message.

Asynchronous Receive: the receiver is notified there is no message available and continues execution.

Messages: (A)synchronous

Thus there are four combinations to consider, three of which are common:

1. **Synchronous send, synchronous receive**
2. **Synchronous send, asynchronous receive**
3. **Asynchronous send, synchronous receive**
4. **Asynchronous send, asynchronous receive**

We may also have “acknowledgement” messages.

Producer-Consumer Problem

A general paradigm for understanding IPC is known as the *producer-consumer* problem.

The producer creates some information.

The information is later used by the consumer.

Example: the database produces data to be consumed by the shell.

This is a general problem and applicable to client-server situations.

IPC Implementation Strategies

There are three approaches we will consider on how we can accomplish IPC:

1. Shared memory.
2. The file system.
3. Message passing.

All are quite common.

Shared Memory

Conceptually, the idea of shared memory is very simple.

A region of memory is designated as being shared with some processes.

Those processes may read and write to that location.

To share an area of memory, the OS must be notified.

Shared Memory

Normally, a region of memory is associated with exactly one process (its owner).

That process may read and write that location.
Other processes may not.

If a second process attempts to do so, the operating system will intervene and that will be an error.

If a process wants to designate memory as shared, it needs to tell the operating system it is okay.

Shared Memory

The OS needs to know that the memory is referenced by two processes.

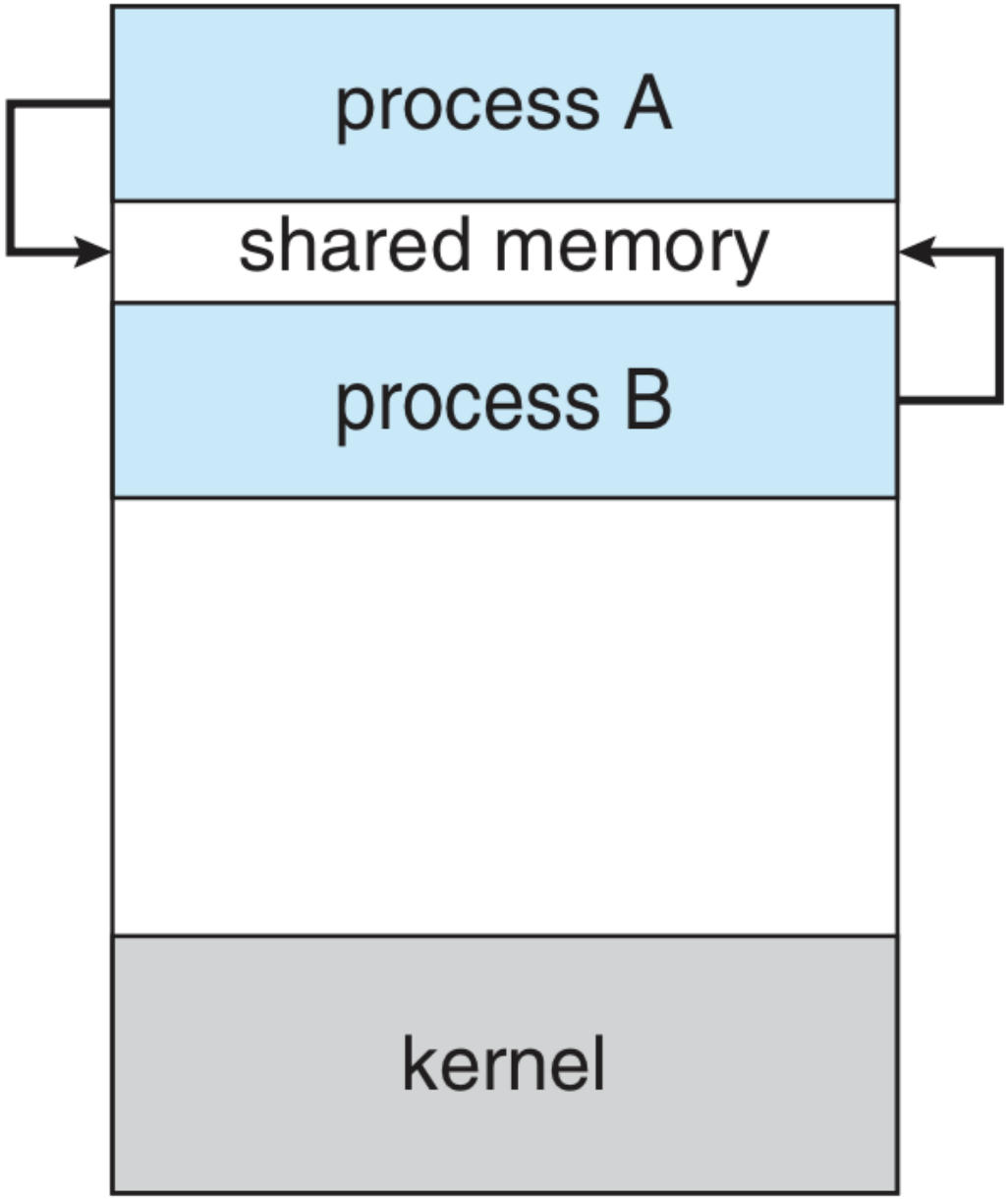
If the first one terminates and is reaped, the memory may still be in use by the second process.

The previously-shared region should not be considered free as long as the second process is still using it.

Once the area of memory is shared, when either process attempts to access it, it is just a normal memory access.

The kernel is only involved in the setup and cleanup of that shared area.

Shared Memory



Shared Memory: Risk

When a section of memory is shared, there is the possibility that one process overwrites another's changes.

To prevent this, we need a system of co-ordination.

...A subject we will return to later.

File System

Another way for 2 processes to communicate is through the file system.

Unlike shared memory, messages stored in the file system are persistent.

Can be used if the sender & receiver know nothing about one another.

File System

The producer may write to a file in an agreed upon location.

The consumer may read from that same location.

The operating system is still involved because of its role in file creation and manipulation.

File System: Co-ordination

If one file is being used then we still have the problem of co-ordination.

We can get around this, however, by using multiple files with unique IDs.

Example from a co-op work term: if the producer is generating XML data, it can write in a file in a designated `import/` directory.

The consumer program scans the directory, and imports files.

In this case, since one process writes files and another reads them, there is no possibility that one process overwrites the data of another.

If the sender chooses distinct file names, it will not overwrite a message if a second message is created before the receiver picks up the first.

Message Passing

Message passing is a service provided by the operating system.

The sender will give the message to the OS and ask that it be delivered to a recipient.

There are two basic operations: sending and receiving.

Messages can be of fixed or variable size.

Message Passing

Our experience with postal mail, or e-mail, suggests that to send a message successfully, the sender needs to indicate where the message should go.

Under *direct communication*, each process that wants to communicate needs to explicitly name the recipient or sender of the communication.

`send(A, message)` – Send a message to process *A*.
`receive(B, message)` – Receive a message from process *B*.

Symmetric addressing.

Message Passing

This does not fit our experience with postal mail.

Receiving an item does not require foreknowledge of the sender.

More common: asymmetric addressing.

Sender names the recipient; recipient can receive from anyone.

`send(`*A*`, message)` – Send a message to Process *A* (unchanged).
`receive(id, message)` – Receive a message from any process; the variable `id` is set to the sender.

In either case, we have to know some identifier for the other processes.

This is not very flexible.

If we want to replace process *B* with some alternative software...
- Change *A*, recompile it, and reinstall it?
- ‘Fake” the identifier of the new software?

What if the sender will produce the data but is not interested in who receives it?

Message Passing

What we would like is *indirect communication* where the messages are sent to mailboxes.

That makes our send and receive functions:

`send(M, message)` – Send a message to mailbox *M*.
`receive(M, message)` – Receive a message from mailbox *M*.

A mailbox may belong to a process or be set up by the OS.

If the mailbox belongs to the process:

Anyone can send to this mailbox.

Only the owning process may receive messages from that mailbox.

If the owner process has not started or has terminated, attempting to send to that mailbox will be an error for the sender.

If the mailbox is owned by the operating system, it is persistent and independent of any particular process.

There is no conceptual reason preventing an operating system mailbox from belonging to more than one process.

If mailbox M belongs to the operating system and processes P_1 and P_2 have access to it, which process will receive a message sent to M?

Message Passing

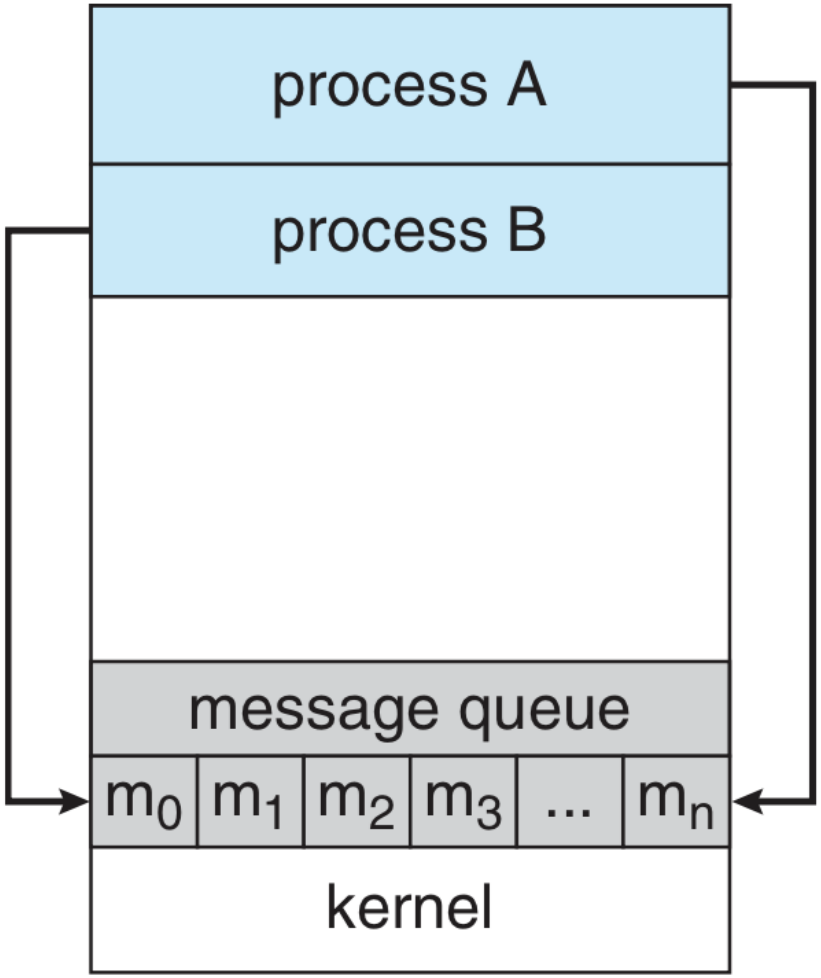
Two potential solutions:

1. Only one process may be the receiver at a time.

2. A system for determining whose turn it is.

Message Queue

A message queue for communication between processes *A* and *B*:



Message Queues

Thus far we have dealt with messages one at a time.

The sender wants to send one message and the receiver wants to receive one message.

If the sender wants to send a second message before the first message is received, the sender will have three choices:

1. Wait for the last message to be picked up (block).
2. Overwrite the last message (sometimes this is what you want).
3. Discard the current message (let the old one remain).

Message Queue

A queue may alleviate the problem or just “kick the can down the road”.

If a queue exists, when sending a message, that message is placed in the queue and when receiving a message, the first message is taken.

If the queue is of (effectively) unlimited size, no problem!

If the queue has a fixed size then the problem is put off but not solved.

The sender can keep adding messages until the queue is full.

If the queue is full, the sender has to face the same choices.

UNIX Pipes

In UNIX, we can create a *pipe* to set up communication.

The producer writes in one end; the consumer receives on the other.

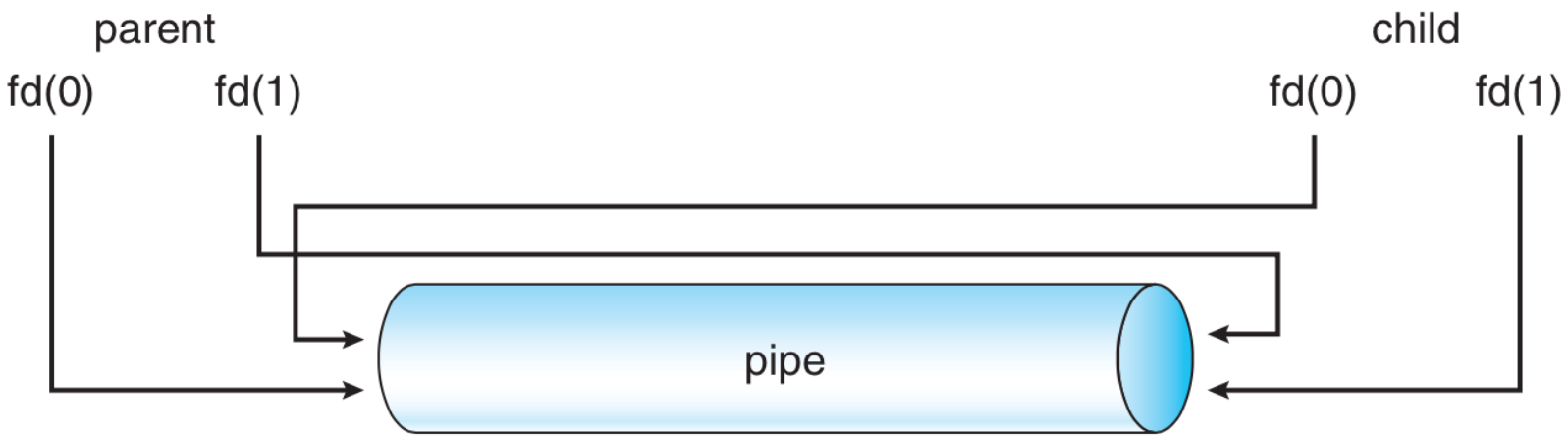
This is unidirectional, so if bidirectional communication is desired, two pipes must be used (going in different directions).

Constructing a UNIX Pipe

The method is `pipe` and it is constructed with the call:
`pipe( int fileDescriptors[])`
where `fileDescriptors[0]` is the read-end; and
`fileDescriptors[1]` is the write-end.

Yes, `fileDescriptors` means that UNIX thinks of a pipe as a file
even though it is in memory.

UNIX Pipes



UNIX Pipes

The pipe is a block of main memory interpreted as a circular queue.

Each entry in the queue is fixed in size and usually one character.

The sender may place the message into the queue in small chunks.

The receiver gets data one character at a time.

The sender and receiver need to know when the message is finished.

Solutions: termination character, or declared length at the start.

Named Pipes

A UNIX pipe may be stored on disk.

When this happens, we call it a named pipe.

Unless we make it a named pipe, a pipe exists only as long as the processes are communicating.

Regular pipes require a parent-child process relationship.
Named pipes do not.

Named pipes are also bidirectional, but one direction at a time.

UNIX Command Line Pipes

You may have worked with pipes on the UNIX command line.

A command like `cat fork.c | less` creates a pipe;.

It takes the output of the `cat` program and delivers it as input to `less`.

Pipe Code Example

Use `fork` to spawn a new child process and then setting up a communication pipe between the parent and child.

We will send a message “Greetings” from the parent to the child.

```
char_write msg[BUFFER_SIZE] = "Greetings";
char_read msg[BUFFER_SIZE];
int fd[2];
pid_t pid;

if (pipe(fd) == -1) {
  fprintf(stderr,"Pipe failed");
  return 1;
}
```

Pipe Code Example, Continued

```
/* fork a child process */
  pid = fork();

  if (pid < 0) {
    /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
  }
```

Pipe Code Example, Continued

```
if (pid > 0) { /* parent process */
   /* close the unused end of the pipe */
   close(fd[READ_END]);

   /* write to the pipe */
   write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

   /* close the write end of the pipe */
   close(fd[WRITE_END]);

 }
```

`READ_END` is defined as `0` in a `#define` directive.
`WRITE_END` is defined as `0` in a `#define` directive.

Pipe Code Example, Continued

```
  else { /* child process */
    /* close the unused end of the pipe */
    close(fd[WRITE_END]);

    /* read from the pipe */
    read(fd[READ_END], read_msg, BUFFER_SIZE);
    printf("read %s",read_msg);
     /* close the write end of the pipe */
     close(fd[READ_END]);
  }
  return 0;
}
```

Creating a Named Pipe

If we wanted to create a named pipe, the system call is `mkfifo`.

Sometimes a named pipe is called a FIFO.

As it is a file, it can be manipulated with the usual UNIX file system calls: `open`, `read`, `write`, and `close`.