MONASH University

MONASH
INFORMATION
TECHNOLOGY

INTERPROCESS COMMUNICATION

LECTURE 10 / FIT2100 / SEMESTER 2
2019

WEEK 11

INTERPROCESS COMMUNICATION

INTRODUCTION

LEARNING OUTCOMES

- Discuss various mechanisms for interprocess communication (IPC).
- Understand the implications of different IPC mechanisms for **synchronization** and **concurrency**.

READING

- Stallings:
  - Chapter 5 sections 5.6, 5.7.
  - Chapter 6 sections 6.7, 6.8.
- Further reading: Curry, *Unix Systems Programming for SVR4*
  - Chapter 13 – IPC, Chapter 10 – Signals

WHAT IS IPC?

INTERPROCESS COMMUNICATION

COMMUNICATION BETWEEN TWO OR MORE PROCESSES

- Multiple applications/utilities often need to communicate
- Communication may happen over a network, but often communication happens among processes running on the same computer.

APPROACHES TO BE DISCUSSED

- Signals
- Shared memory
- Streams (pipes)
- Sockets (client/server)
- Message queues.

WHY DO WE NEED IPC?

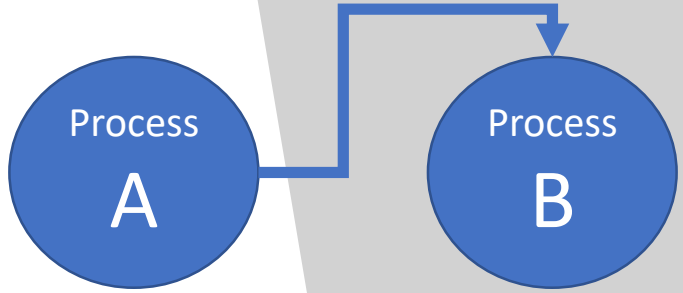
LET'S COMPARE PROCESSES TO THREADS…

THREADS

- Memory is 'shared' between threads
  - Well actually, there is only one memory space
  - The threads are all part of the same process
  - This is only communication **within** a process

PROCESSES

- Often, multiple related or unrelated processes need to exchange data
  - Most common case: a program often streams output to a console utility using a **'print'** statement.
- Different processes do not typically share the same memory space.
- The OS provides a number of communication mechanisms here.



SIGNALS

THE MOST PRIMITIVE KIND OF IPC

Image credit: Original mikz, Creative Commons: BY-SA 4.0

MONASH University

UNIX SIGNALS

SIGNALS ARE LIKE INTERRUPTS FOR USER PROCESSES

USED TO INFORM A PROCESS OF AN ASYNCHRONOUS EVENT

- A signal is ‘delivered’ by updating a field (bit flag) in the process table for the process that receives the signal.
- There are no priorities for signals: all treated equally.
- A signal has a signal number and that’s it. This number identifies the **type** of signal sent.

A PROCESS MAY RESPOND TO A SPECIFIED SIGNAL NUMBER BY…

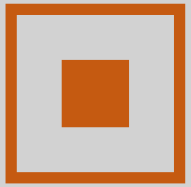
- Jumping into a signal-handler function, or
- Choosing to ignore a signal, or,
- Performing the operating system’s default action for that signal (e.g. process termination).

SOME COMMON SIGNALS

FOR EACH SIGNAL NUMBER, A CONSTANT IS DEFINED IN <signal.h>

SIGNALS USED FOR PROCESS MANAGEMENT

- **`SIGINT`:** The *interrupt* signal that is sent when you press Ctrl+C in a terminal window.
  - **Default action:** terminate the process.
- **`SIGTSTP`:** The *stop* signal that is sent when you press Ctrl+Z in a terminal window
  - **Default action:** suspend (pause) the process
  - In **bash**, the **`fg`** command can be used to resume a stopped process.



SOME SIGNALS CANNOT BE HANDLED OR IGNORED BY PROCESSES.

- **`SIGKILL`:** Used to *kill* a process without any way for the process to ignore the signal or handle it in a different way
  - **Default (only) action:** terminate the process.

For more signals, check the man page: ***`man 7 signal`***

DEALING WITH SIGNALS

UNIX SYSTEM CALLS (defined in <signal.h>)

HANDLING A SIGNAL

- The handler must be set up **before** the signal arrives:

```
sighandler_t signal(int signal_number, sighandler_t handler)
```

- The value of **`handler`** may be a pointer to a handler function to be called, or **SIG_IGN** (ignore signal) or **SIG_DFL** (reset to default action).
- Refer to man page: **`man 2 signal`** for more information.

SENDING A SIGNAL TO ANOTHER PROCESS

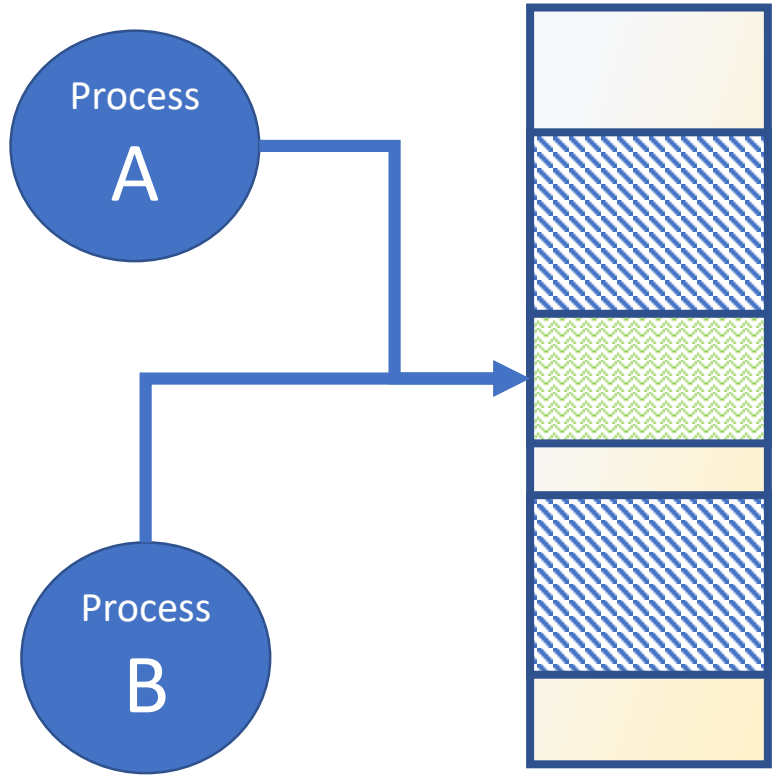
- You only need the PID and the signal type you wish to send:

```
int kill(pid_t process, int signal_number)
```

- Why such a morbid name, ‘kill’?
  - Originally the signalling system was only used to terminate processes.
  - Nowadays, a variety of harmless signals can be sent too!

SHARED MEMORY (SHM)

– AN IPC APPROACH THAT WORKS THROUGH THE VIRTUAL MEMORY SYSTEM



SHARED MEMORY

FROM LAST WEEK'S LECTURE…

A SEGMENT IN VITUAL MEMORY THAT IS ALLOCATED TO MULTIPLE PROCESSES

- A memory segment may be shared among multiple processes
- Like with multi-threading, multiple processes can access each other's memory
- Unlike with multi-threading, it does not happen automatically.
    - Must manually request a new block of shared memory to be allocated.
    - Required number of bytes must be specified.
    - Must manually assign a data structure (e.g. **struct** or **array**) to the shared block of bytes.
- There is no concurrency protection offered. Mutual exclusion mechanisms like semaphores must be used.
- Using multiple processes with shared memory can be **more robust** than using multiple threads. e.g. if one process crashes, the parent process can restart it.
    - e.g. tabs in Firefox web browser and Google Chrome

TWO WAYS TO USE SHARED MEMORY

LINUX SUPPORTS TWO TRADITIONAL UNIX APPROACHES

SYSTEM V STYLE (FROM OLD SVR4 UNIX SYSTEMS)

- Library functions in **`<sys/shm.h>`** include **`shmget(…)`** and **`shmat(…)`**
- Processes are required to have knowledge of a common **key** value in order to get access to the same segment.
- The key value is an integer that might be stored in a file for other processes to look up, or simply generated prior to forking a child process.

POSIX STYLE (NEWER 'STANDARD' APPROACH)

- Library functions in **`<sys/mman.h>`** include **`shm_open(…)`** and **`mmap(…)`**
- A **filename** in the filesystem is associated with a shared memory segment.
- When opened, the **file descriptor** can be read or written to just like a file.
- Can also be mapped to a logical memory address using **`mmap(…)`**

SHARED MEMORY APPROACHES

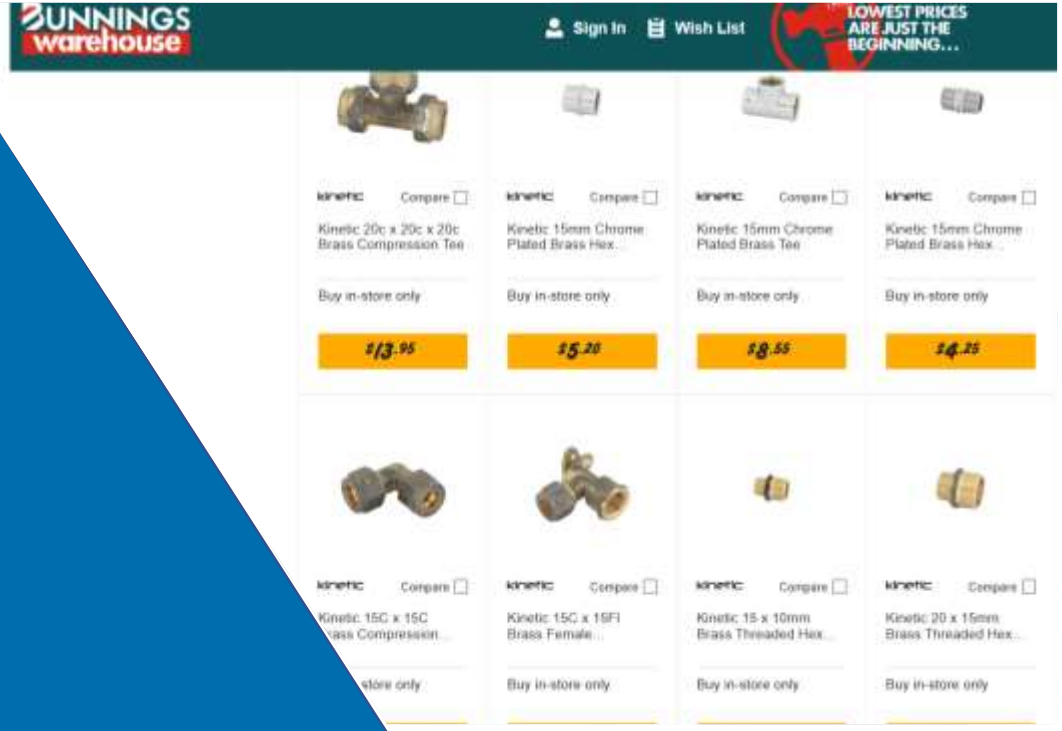
SYSTEM V VS POSIX

BEWARE!

- For either shared memory approach, there is **no** built-in concurrency protection
  - Shared memory has the same hazards as multithreading.
- Mutual exclusion mechanisms such as **semaphores, mutexes,** etc. must be used when accessing shared data.
- These mutual exclusion resources can be stored in the shared memory itself.

STREAM-BASED IPC

UNNAMED PIPES AND FIFOS



– AN APPROACH TO IPC THAT WORKS THROUGH THE KERNEL DIRECTLY

WHAT ARE PIPES?

BYTES GO IN ONE END, COME OUT AT THE OTHER END

A BUFFER MAINTAINED WITHIN THE KERNEL

- A pipe is a resource where information written in at one ‘end’ can be read out at the other ‘end’.
- Slower than SHM since all data must pass through the kernel.
- A pipe is a bit like a file but with **two** file descriptors. One file descriptor is for reading only, while the other is for writing only.

A PIPE ENFORCES SYNCHRONISATION BETWEEN READER AND WRITER

- As long as only one process reads, and only one process writes to pipe, concurrency protection is guaranteed.
  - First-in-first-out behavior

The reader cannot read past what has been written, and the writer cannot overwrite data in the pipe before it has been read

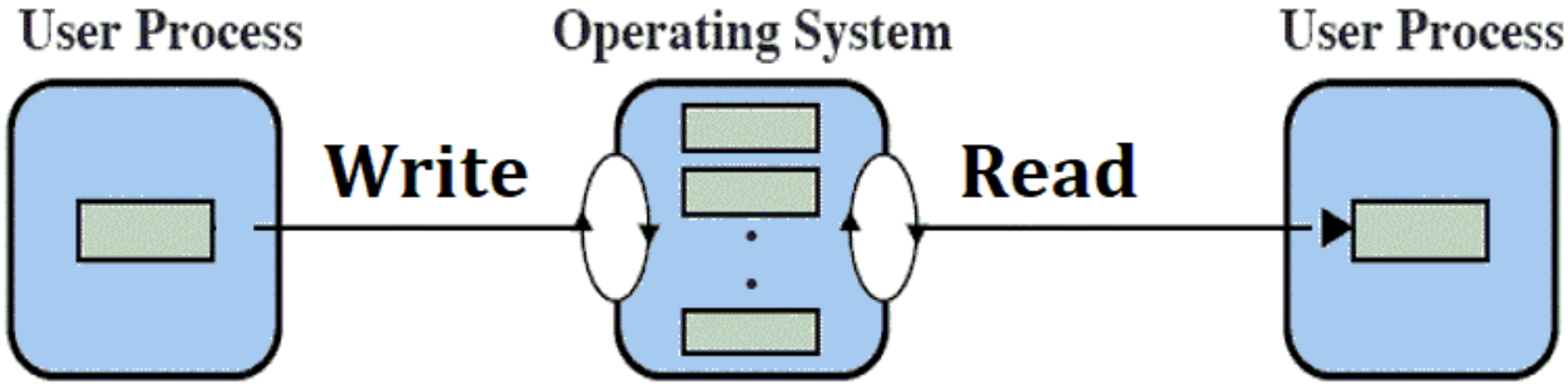
- The kernel implements a **circular buffer** to make this happen.

CIRCULAR BUFFERS (QUEUES)

RECALL: LECTURE 3 (WEEK 3)

A CIRCULAR BUFFER IS A SET OF BUFFERS ACCESSED IN A 'CIRCULAR' WAY

- The stream of bytes spans multiple buffers (character arrays)
- At any one time, one buffer is selected for reading while another is being written to.
- **Writer** fills buffers while the **reader** empties them, before moving on to the next buffer in the circle.
- If the reader 'catches up' to the writer, further reads will **block** until the next buffer becomes available (has been written to).
- If the writer **catches up** to the reader, further writes will **block** until the next buffer becomes available for writing (has been completely read).



TWO KINDS OF PIPES

UNNAMED AND NAMED

UNNAMED PIPES

- Can only be shared between related processes.
- Created using the **`pipe`** system call.
    - Creates a **pipe** and allocates **two file descriptors**.
    - After a **`fork`**, the child will inherit the parent's open file descriptors.
    - So one process can write to the pipe to talk to the other process.

NAMED PIPES (also known as FIFOs)

- A special kind of pipe with a **filename** in the filesystem.
- Can be used between unrelated processes.
- A special 'file' created using the **`mkfifo`** system call.
- One process then opens the 'file' read-only, while the other process opens the same file 'write-only'.

SOCKETS

– ONE SERVER PROCESS ACCEPTS CONNECTIONS FROM MULTIPLE CLIENT PROCESSES

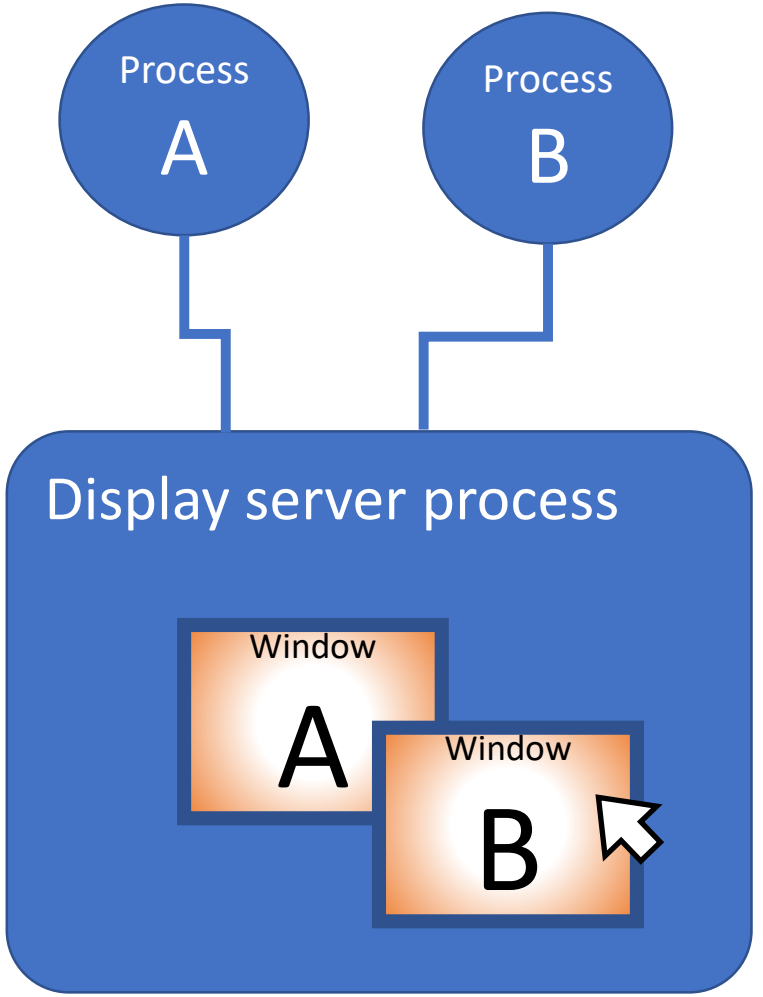
EXAMPLE

DISPLAY SERVER UTILITY

ONLY <u>ONE</u> PROCESS CAN CONTROL THE DISPLAY HARDWARE AT A TIME

How does a desktop system like ***Windows*** work?

- One utility called the **display server** takes responsibility for drawing all graphics on the screen.
- Applications (**clients**) connect to the display server and send instructions for what should be drawn on the screen. (Through an API.)
- The display server interprets instructions from different applications to produce a montage of **windows** on the screen.
- When the user interacts with the mouse and keyboard, the display server figures out which **client** to send the information back to.



If all processes were allowed to write to the display directly, the result would be a mess!

SOCKETS IN LINUX

‘UNIX DOMAIN SOCKETS’

HOW IS A SOCKET CONNECTION CREATED?

1. The server process creates a **`socket`**, which is mapped to a **filename** in the filesystem.
2. Client processes must **`connect`** to this file to reach the server.
3. The server receives information about a new connection request, and chooses to **`accept`** the connection.
4. When accepted, the OS provides an open file descriptor resource to both the client and server process.

- Just like a pipe, a socket connection enforces FIFO behaviour when it is being read and written to concurrently.
- Unlike a pipe, a socket connection can send data in both directions.
    - The kernel provides **two** circular buffers. One for bytes sent from the client to the server, and another for bytes sent from the server to the client.

MESSAGE QUEUES

– A SYSTEM FOR SENDING FIXED-LENGTH MESSAGES BETWEEN PROCESSES

WHAT IS A MESSAGE QUEUE?

MORE STRUCTURED MESSAGE PASSING

A RESOURCE PROVIDED BY THE KERNEL

Once a queue has been created, multiple processes can send messages into the queue, or read them out of the queue.

Multiple processes can share a single queue.

MESSAGES COME WITH ATTACHED INFORMATION

By defining a set of different **message types** (assigning a different integer to different kinds of messages), a process can request the type of message it is waiting for to be delivered next.

Unlike streams (pipes, etc.) a message queue always serves a **whole message** at a time. It does not work on a per-character basis.

MESSAGE FORMAT

MESSAGES ARE SENT AS FIXED-LENGTH CHUNKS

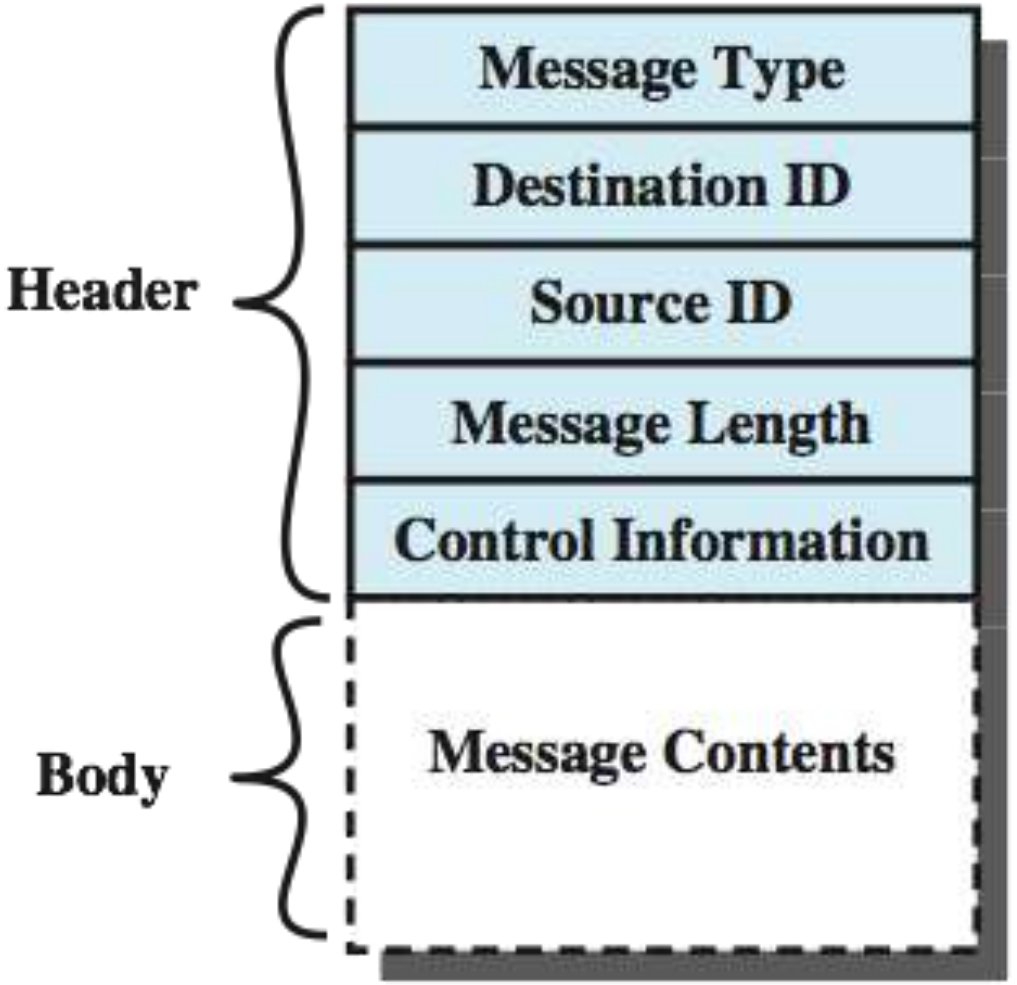
MESSAGE CONSISTS OF TWO PARTS

Header

- Contains information about the message: e.g…
  - Who sent it?
  - Who is meant to receive it?
  - What type of message is it?

Body

- A block of bytes containing the message itself.
- As with shared memory and streams, it is up to the programmer to structure the attached data in a sensible way.



CONCURRENCY IMPLICATIONS

CAN IPC BE USED AS A MUTUAL EXCLUSION TOOL?

IPC MECHANISMS PROVIDED BY KERNEL BUFFERING…

SYNCHRONISING PROCESSES WITH IPC

- Stream based IPC (pipes, etc.) as well as sockets, provide enforcement of FIFO behavior between readers and writers.
- Together with message queues, they share the following important property:
  - **Data can never be received before it has been sent!**
- This means they can be used for synchronizing concurrent processes, as tools for mutual exclusion.
  - e.g. one process may block waiting to read a message from another process, indicating that it is safe to proceed.
  - The kernel guards access to the IPC resource to enforce mutual exclusion without the need for the programmer to use semaphores/mutexes.

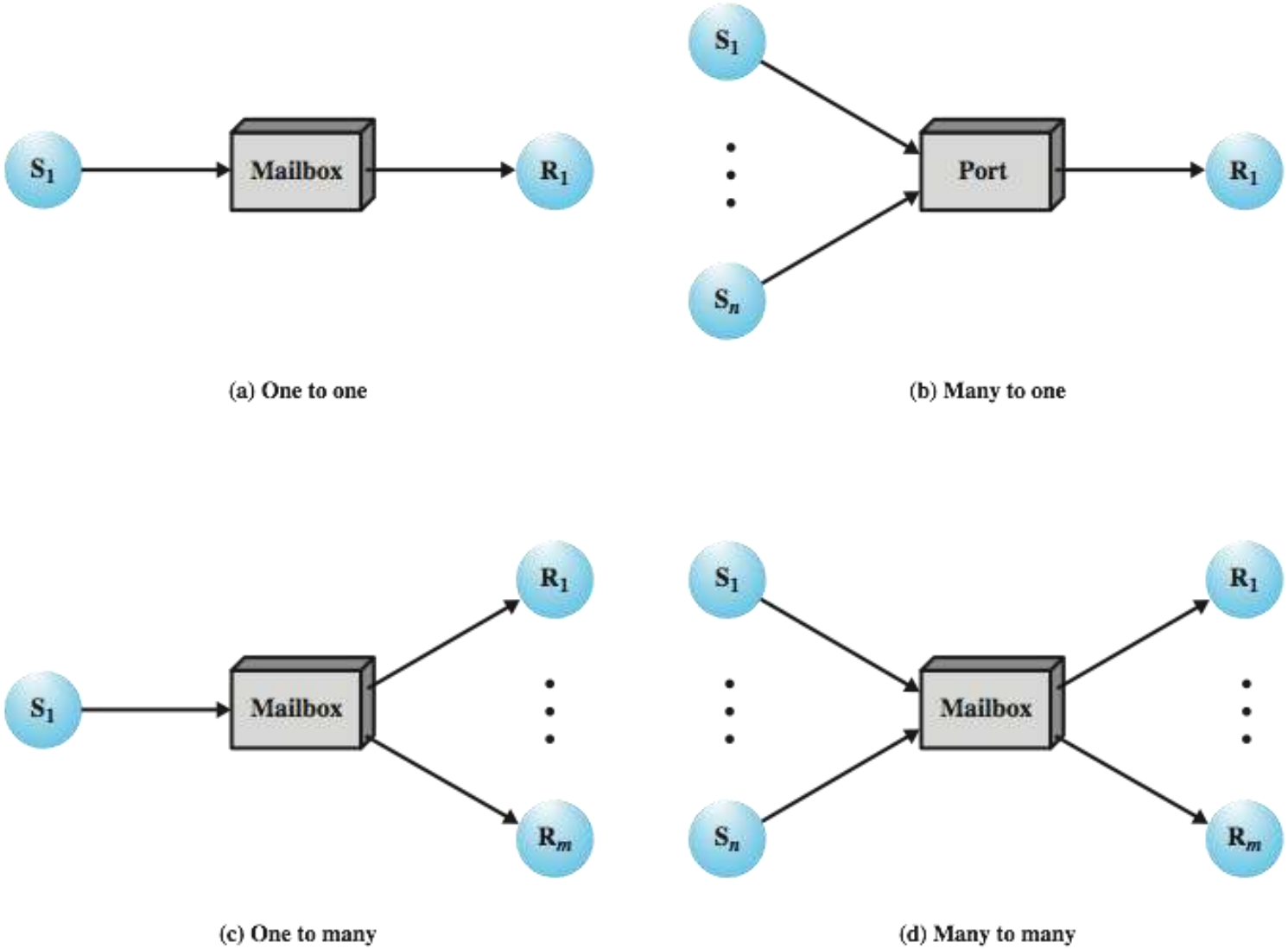
INDIRECT PROCESS COMMUNICATION

STREAMS, MESSAGE QUEUES, ETC.

For IPC mechanisms such as pipes, sockets, and message queues, all data must go through the kernel.

This slows things down a little, but enables the kernel to enforce concurrency protections.

(As discussed earlier.)



WHAT ABOUT SHARED MEMORY?

NO CONCURRENCY PROTECTIONS AVAILABLE

NOT BUFFERED BY KERNEL

- Shared memory does not pass through buffers in the kernel.
- It is a segment in virtual memory that multiple processes may access directly at any time.
  - Fast, but hazardous!
- Additional synchronization tools (semaphores/mutexes) are needed when dealing with shared memory, to guard access to shared data.

SUMMARY (LECTURE 10)

INTERPROCESS COMMUNICATION

- We have discussed several mechanisms for interprocess communication.
  - Signals are the most primitive
  - Other approaches include pipes, sockets and message queues
- We now have a better understanding of the implications for synchronisation and mutual exclusion, of using IPC tools in an environment where concurrent processes share data.

- **Next week: Security**