MONASH University

Information Technology

FIT2100 Operating Systems

INTER PROCESS COMMUNICATION - PART 1

Week 10 - Part 2
Semester 2 2024
(Reading: Tanenbaum: Chapter 2 and Stallings: Chapter 7, 8)

Dr Charith Jayasekara
Faculty of Information Technology

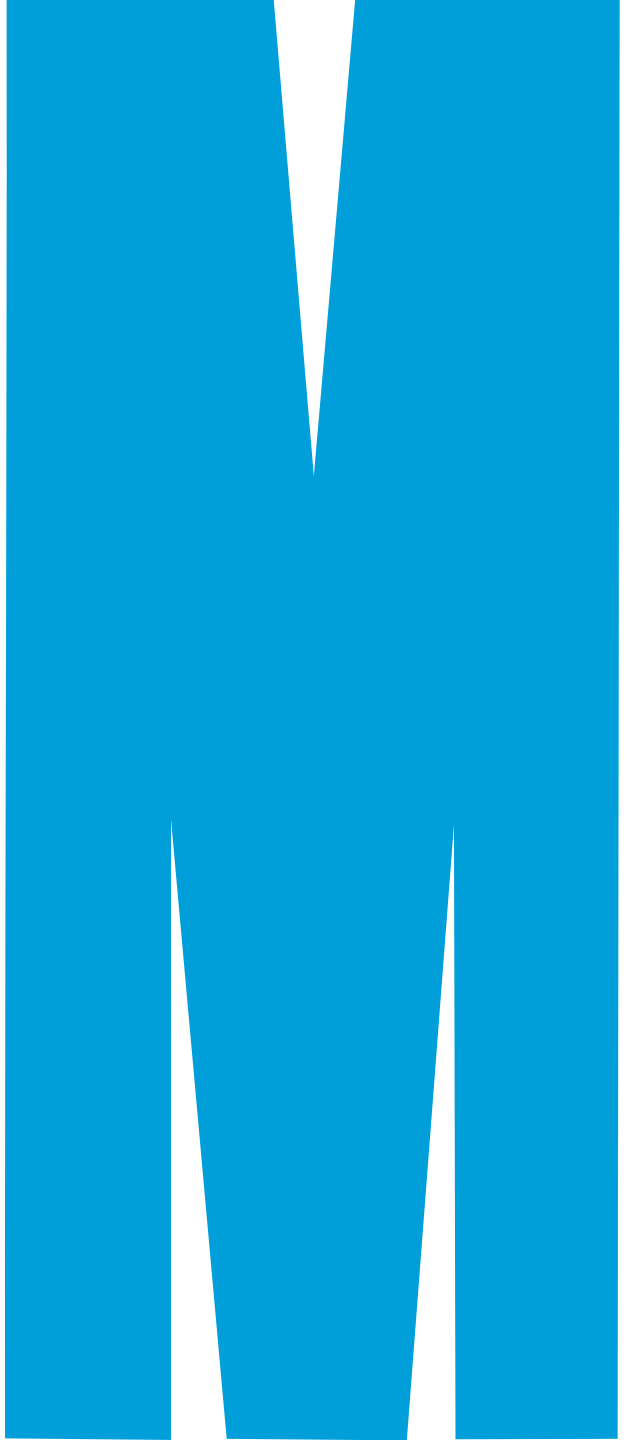
OUTLINE

- Inter Process Communication (IPC)
- Message Queues
- Concurrency Implications
- Signals
- Shared Memory
- Stream-based IPC: Unnamed and Named Pipes
- Sockets

LEARNING OUTCOMES

- Upon the completion of Week 10 Part 2 and Week 11, you should be able to:
  - Discuss various **mechanisms** for interprocess communication (IPC).
  - Understand the **implications** of different IPC mechanisms for **synchronization** and **concurrency**.

INTER PROCESS COMMUNICATION (IPC)



INTERPROCESS COMMUNICATION (IPC)

WHAT IS IPC?

- **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.

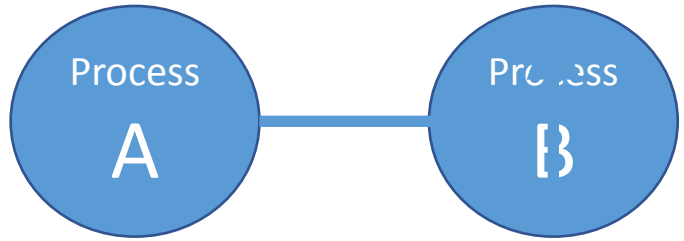
- **Approaches** we will discuss
  - Signals
  - Shared memory
  - Streams (pipes)
  - Sockets (client/server)
  - Message queues

INTERPROCESS COMMUNICATION (IPC)

WHY DO WE NEED IPC (THREADS VS PROCESSES)

❒ THREADS

- Memory is **'shared'** between threads
- 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
- Example: 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 to provide for specific styles of communication

SIGNALS
THE MOST BASIC TYPE OF IPC

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

SIGNALS

UNIX SIGNALS

- Signals are like **interrupts** for user processes
- Signals may be sent by a process to a process, or from the kernel to a process
- 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).

SIGNALS

SOME COMMON SIGNALS

- For each signal number in *nix, a **constant** is defined in **<signal.h>**
- Signals are used for process management:
  - `SIGINT`: The ***interrupt*** signal that is sent when you press Ctrl+C in a terminal window (your shell process sends the signal to the target process).
    - **Default action:** terminate the process.
  - `SIGTSTP`: The ***stop*** signal that is sent when you press Ctrl+Z in a terminal window (your shell process sends the signal to the target process).
    - **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

SIGNALS

BSD SIGNALS (MacOSX 10.14 Mojave)

No	Name	Default Action	Description
1	SIGHUP	terminate process	terminal line hangup
2	SIGINT	terminate process	interrupt program
3	SIGQUIT	create core image	quit program
4	SIGILL	create core image	illegal instruction
5	SIGTRAP	create core image	trace trap
7	SIGEMT	create core image	emulate instruction executed
8	SIGFPE	create core image	floating-point exception
9	SIGKILL	terminate process	kill program
10	SIGBUS	create core image	bus error
11	SIGSEGV	create core image	segmentation violation
12	SIGSYS	create core image	non-existent system call invoked
13	SIGPIPE	terminate process	write on a pipe with no reader
14	SIGALRM	terminate process	real-time timer expired
15	SIGTERM	terminate process	software termination signal
16	SIGURG	discard signal	urgent condition present on socket
17	SIGSTOP	stop process	stop (cannot be caught or ignored)
18	SIGTSTP	stop process	stop signal generated from keyboard
19	SIGCONT	discard signal	continue after stop
20	SIGCHLD	discard signal	child status has changed
21	SIGTTIN	stop process	background read attempted from control terminal
22	SIGTTOU	stop process	background write attempted to control terminal
23	SIGIO	discard signal	I/O is possible on a descriptor (see fcntl(2))
24	SIGXCPU	terminate process	cpu time limit exceeded (see setrlimit(2))
25	SIGXFSZ	terminate process	file size limit exceeded (see setrlimit(2))
26	SIGVTALRM	terminate process	virtual time alarm (see setitimer(2))
27	SIGPROF	terminate process	profiling timer alarm (see setitimer(2))
28	SIGWINCH	discard signal	Window size change
29	SIGINFO	discard signal	status request from keyboard
30	SIGUSR1	terminate process	User defined signal 1
31	SIGUSR2	terminate process	User defined signal 2

SIGNALS

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!

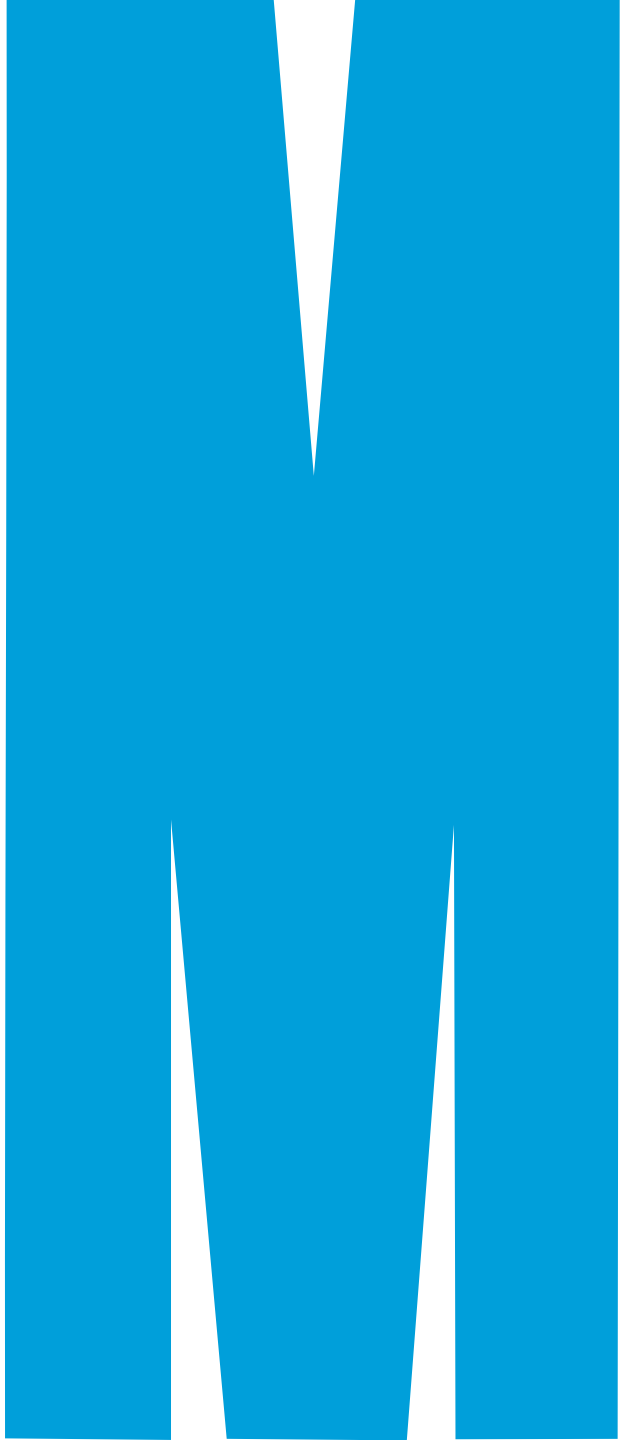
SIGNALS

ADVANTAGES AND DISADVANTAGES OF SIGNALS

- **Advantage**: Signals are simple to implement
- **Advantage**: Signals have a clearly defined meaning and actions
- **Advantage**: Signals are easy to implement in user shells

- **Disadvantage**: Signals are slow to transmit
- **Disadvantage**: Signals cannot transmit user defined messages
- **Disadvantage**: Signals have trivial bandwidth – eq. to 5 bits per message

MESSAGE QUEUES



MESSAGE QUEUES

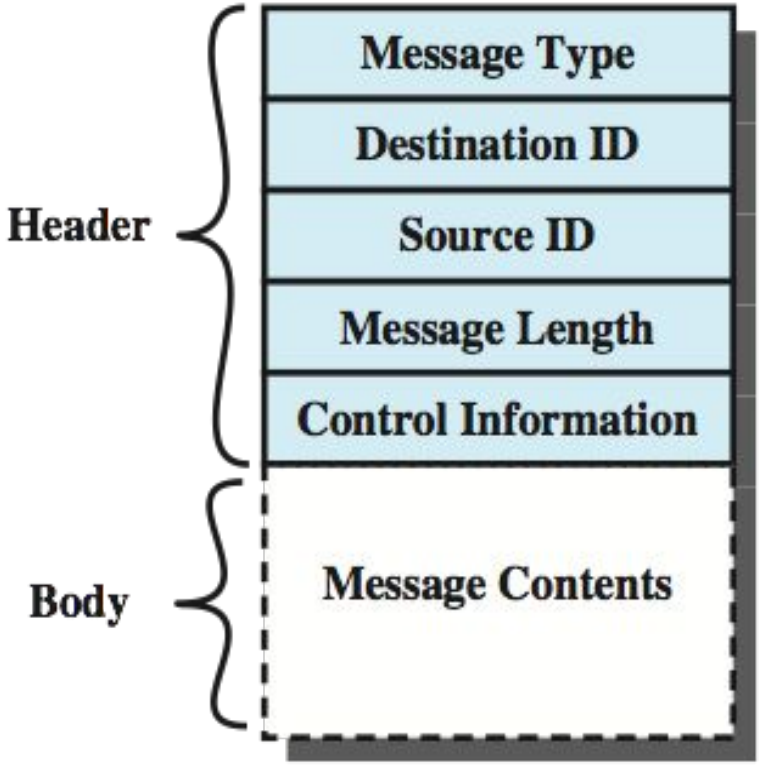
WHAT ARE MESSAGE QUEUES

- 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 QUEUES

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.



MESSAGE QUEUES

ADVANTAGES AND DISADVANTAGES OF MESSAGE QUEUES

- **Advantage**: Message Queues provide structured IPC

- **Disadvantage**: Message Queues are often slow to transmit
- **Disadvantage**: Message Queues are usually proprietary and not portable

MESSAGE QUEUES

EXAMPLE: SVR4 UNIX MESSAGE QUEUES (Curry, 2014, p127)

- Each queue has a unique identifier, called a `queue_id`.
- Queues described with a `msqid_ds` struct, in `sys/msg.h; sys/types.h`:

```
struct msqid_ds {
    struct ipc_perm msg_perm; /* permissions    */
    struct msg *msg_first;    /* 1st message    */
    struct msg *msg_last;     /* last message   */
    ushort     msg_cbytes;    /* # bytes on q   */
    ushort     msg_qnum;      /* # of msgs on q */
    ushort     msg_qbytes;    /* max # bytes/q  */
    ushort     msg_lspid;     /* last send proc */
    ushort     msg_lrpid;     /* last recv proc */
    time_t     msg_stime;     /* last send time */
    time_t     msg_rtime;     /* last recv time */
    time_t     msg_ctime;     /* last chg time  */
};
```

MESSAGE QUEUES

EXAMPLE: SVR4 UNIX MESSAGE QUEUES (Curry, 2014, p127)

- Queue permissions are described with an `ipc_perm` struct in `sys/ipc.h`.

```
struct ipc_perm {
    ushort    uid;      /* owner's user id       */
    ushort    gid;      /* owner's group id      */
    ushort    cuid;     /* creators' user id     */
    ushort    cgid;     /* creator's group id    */
    ushort    mode;     /* access permissions    */
    ushort    seq;      /* slot sequence number */
    key_t     key;      /* key (queue name)      */
};
```

- To create a new queue, or get the identity of a queue, you must execute a `msgget` system call
- To get and modify the attributes of an existing queue, you must execute a `msgctl` system call
- To send a message to a queue, you must execute a `msgsnd` system call, using a *queue id*, a *msgbuf* struct pointer, an integer message size, and a *flags* word
- To receive a message from a queue, you must execute a `msgrcv` system call, using a *queue id*, a *msgbuf* struct pointer, integer *maximum message size* and *type*, and a *flags* word

MESSAGE QUEUES

EXAMPLE: SVR4 UNIX MESSAGE QUEUES (Curry, 2014, p127)

- The `msgbuf` struct is defined thus:

```
struct msgbuf {
    long    mtype;
    char    *mtext;
};
```

- Curry provides a simple server program to demonstrate queue operations (next page)

MESSAGE QUEUES

EXAMPLE: SVR4 UNIX MESSAGE QUEUES (Curry, 2014, p127)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#define MSGSZ     128
/*
 * Declare the message structure.
 */
struct msgbuf {
    long    mtype;
    char    mtext[MSGSZ];
};
main()
{
    int msqid;
    key_t key;
    struct msgbuf sbuf, rbuf;
    /*
     * Create a message queue with "name"
     * 1234.
     */
    key = 1234;

    /*
     * We want to let everyone read and
     * write on this message queue, hence
     * we use 0666 as the permissions.
     */
```

MESSAGE QUEUES

EXAMPLE: SVR4 UNIX MESSAGE QUEUES (Curry, 2014, p127)

```
if ((msqid = msgget(key, IPC_CREAT | 0666)) < 0) {
        perror("msgget");
        exit(1);
    }
    /*
     * Receive a message.
     */
    if (msgrcv(msqid, &rbuf, MSGSZ, 0, 0) < 0) {
        perror("msgrcv");
        exit(1);
    }
    /*
     * We send a message of type 2.
     */
    sbuf.mtype = 2;
    sprintf(sbuf.mtext, "I received your message.");
    /*
     * Send an answer.
     */
    if (msgsnd(msqid, &sbuf, strlen(sbuf.mtext) + 1, 0) < 0) {
        perror("msgsnd");
        exit(1);
    }
    exit(0);
}
```

- Curry also provides a simple client program to demonstrate queue operations (next page)

MESSAGE QUEUES

EXAMPLE: SVR4 UNIX MESSAGE QUEUES (Curry, 2014, p127)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#define MSGSZ     128
/*
 * Declare the message structure.
 */
struct msgbuf {
    long    mtype;
    char    mtext[MSGSZ];
};
main()
{
    int msqid;
    key_t key;
    struct msgbuf sbuf, rbuf;
    /*
     * Get the message queue id for the
     * "name" 1234, which was created by
     * the server.
     */
    key = 1234;
    if ((msqid = msgget(key, 0666)) < 0) {
        perror("msgget");
        exit(1);
    }
```

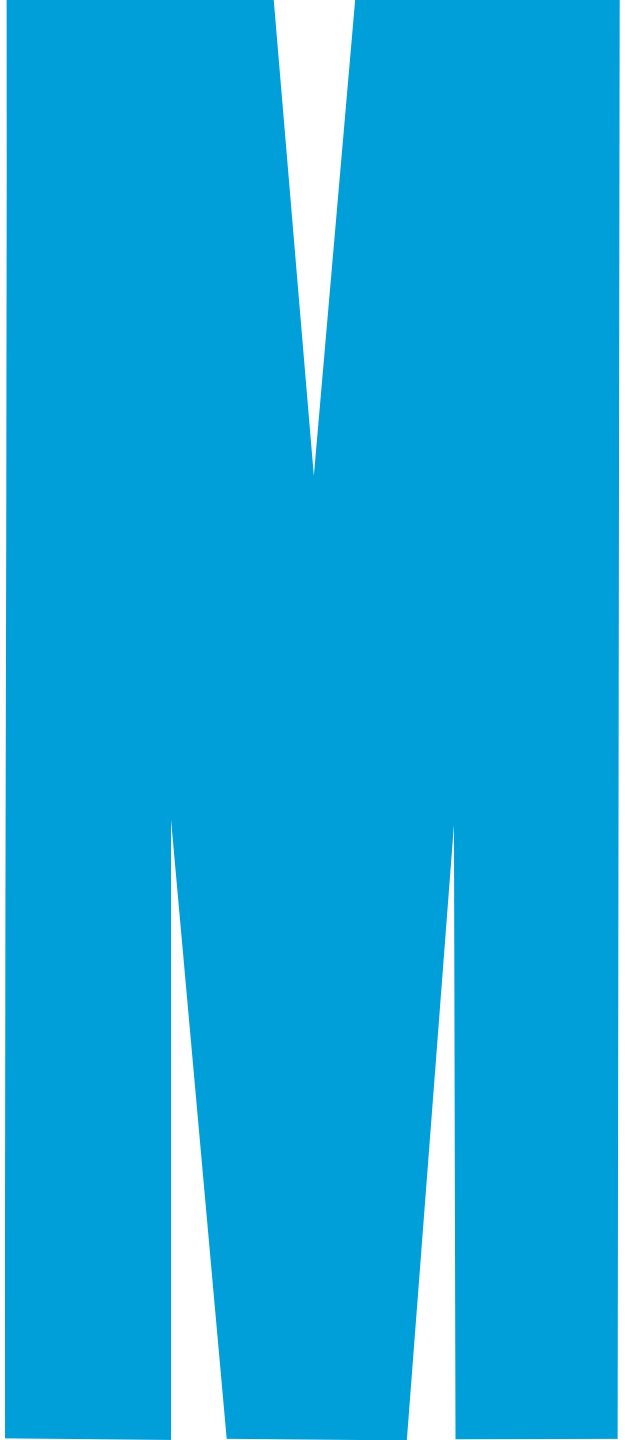
MESSAGE QUEUES

EXAMPLE: SVR4 UNIX MESSAGE QUEUES (Curry, 2014, p127)

```
 /*
     * We'll send message type 1, the server
     * will send message type 2.
     */
    sbuf.mtype = 1;
    sprintf(sbuf.mtext, "Did you get this?");
    /*
     * Send a message.
     */
    if (msgsnd(msqid, &sbuf, strlen(sbuf.mtext) + 1, 0) < 0) {
        perror("msgsnd");
        exit(1);
    }
    /*
     * Receive an answer of message type 2.
     */
    if (msgrcv(msqid, &rbuf, MSGSZ, 2, 0) < 0) {
        perror("msgrcv");
        exit(1);
    }
    /*
     * Print the answer.
     */
    printf("%s\n", rbuf.mtext);
    exit(0);
}
```

- End of example

CONCURRENCY IMPLICATIONS



CONCURRENCY IMPLICATIONS

CAN IPC BE USED FOR MUTUAL EXCLUSION?

- IPC mechanisms provided by kernel buffers can be used for mutual exclusion
- 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 synchronising 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.

CONCURRENCY IMPLICATIONS

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

- So far we have discussed
  - Several mechanisms for interprocess communication.
  - The implications for synchronisation and mutual exclusion, of using IPC tools in an environment where concurrent processes share data.
- Next week
  - More IPC
- Reading
  - Stallings (7th Edition):
    - Chapter 5 sections 5.5, 5.6.
    - Chapter 6 sections 6.7, 6.8.
  - Further reading: Curry, Unix Systems Programming for SVR4,Chapter 13 – IPC, Chapter 10 – Signals
  - Further reading: Curry, Using C on the UNIX System, Chapter 11 IPC