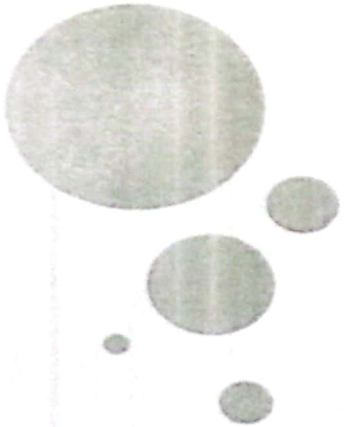
COURSE TITLE:

SYSTEM PROGRAMMING
Chapter I: Processes



By: Prof. Ebot Ebot Enaw

- **This Chapter will explore the following concepts:**
  - PROCESS CREATION AND TERMINATION
  - INTERPROCESS COMMUNICATION
  - REASONS FOR IPC
  - MESSAGE PASSING
  - SHARED MEMORY
  - SYNCHRONIZATION
  - BUFFERING

Process creation in Windows

- *createProcess()* function creates a new child process ;
- whereas *fork()* in Linux has the child process inheriting the address space of its parent ;
- *createProcess()* requires loading a specified program into the address space of the child process at the process creation, whereas *fork()* is passed no parameters *createProcess()* expects no fewer than **10** parameters.

C Program in which CreateProcess creates child process

- *CreateProcess()* function creates a child process that loads the application mspaint.exe.

```
#include <stdio.h>
#include <windows . h>

int main(VOID)
{
STARTUPINFO si;
PROCESSINFORMATION pi;

    // allocate memory ZeroMemory(&si,
    sizeof(si)) ; si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

// create child process
if (! CreateProcess (NULL,     // use command
line
  "C:\\WINDOWS\\system32\\mspaint.exe",      //
  comman.
  NULL, // don't inherit process handle
  NULL, // don't inherit thread handle
  FALSE, // disable handle inheritance 0,
  // no creation flags
  NULL, // use parent's environment block
  NULL, // use parent's existing
  directory


   fprintf(stderr, "Create Process Failed");
   return -1;
);
// parent will wait for the child to complete
WaitForSingleObject(pi.hProcess, INFINITE);
printf("Child Complete");


// close handles
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
```

Two parameters passed to CreateProcess() are instances of STARTUPINFO and PROCESS_INFORMATION.

Default parameters

First two parameters if application name is NULL (as is the case), the command line parameter specifies the application to load.

Provide two pointers to the STARTUPINFO and PROCESS_INFORMATION structures

Equivalent of wait() system call in Unix, which is passed a handle of the child process_pi.hProcess and waits for the process to complete.

Parameters

STARTUP INFO

- Specifies many properties of the new process such as:
  - window size
  - appearance
  - handles to standard I/O files

PROCESS_INFORMATION

- Contains:
  - a handle
  - identifiers of newly created process and its threads

ZeroMemory ( ) function

- invoked to allocate memory for each of these structures before proceeding with CreateProcess ( ).

Process termination

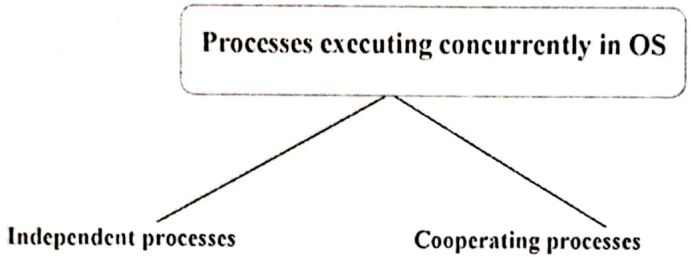
Once a process finishes executing its final statement:

- It asks the OS to delete it by using exit( ) system call in Unix.
- If the parent terminates however all its children are assigned as their new parent the init process.

Parent may terminate the execution of one of its children for a variety of reasons, such as:

- The child has exceeded its usage of some of the resources that it has been allocated. To determine if this has occurred, a parent must have a mechanism to inspect the state of its children.
- The task assigned to the child is no longer required
- The parent is exiting and the operating system doesn't allow a child to continue if its parent terminates
- Parent process may wait for the termination of a child process by using the wait( ) System call, which returns the process identifier of a terminated child so the parent can tell which of his children has terminated.

Interprocess Communication (IPC)



- Cannot affect or be affected by other processes executing in the OS.
- Doesn't share data with any other process.

- It can affect or be affected by the other process executing in the system.
- Shares data with other processes.

PROCESS COOPERATION REQUIRED FOR THE FOLLOWING REASONS

Information sharing:	Many users may be interested in a particular piece of information such as a shared file. Mechanism must be provided to allow concurrent access to the file
Computation speed:	If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Requirement- Computer has multiprocessing elements (CPU or I/O channels).
Modularity:	We may want to construct the system in a modular form, dividing the system functions into separate processes or threads.
Convenience:	Even an individual user may work on many tasks at the same time, a user may be editing, printing and compiling in parallel

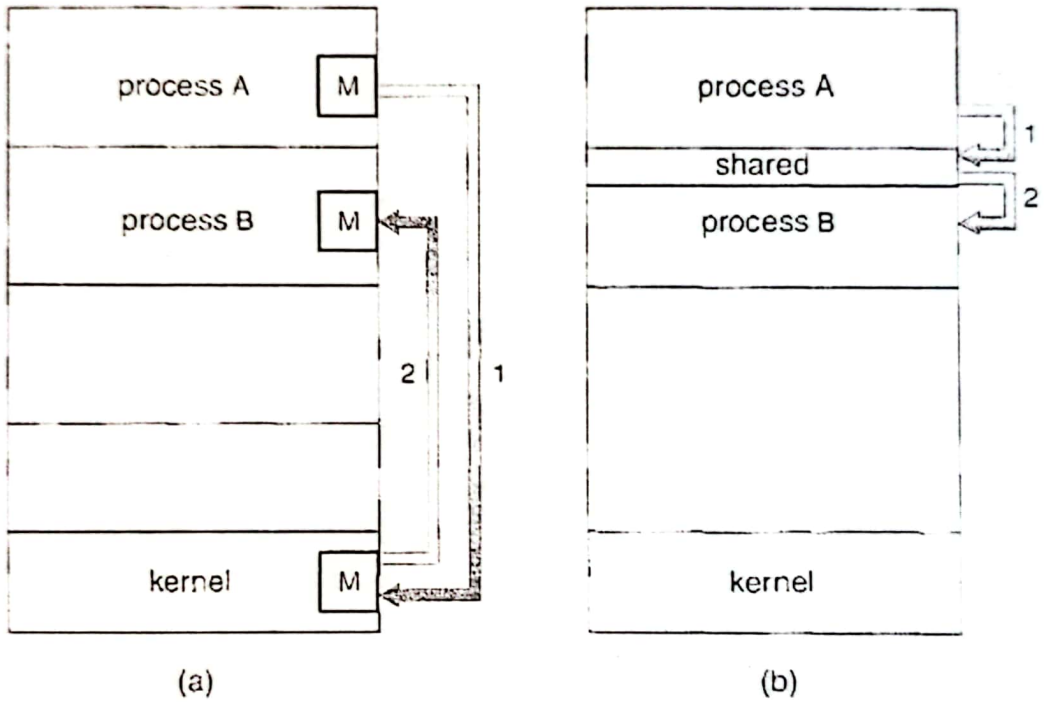
Interprocess communication (IPC) mechanism required by cooperating processes to exchange data and information.

A region of memory that is shared by cooperating processes is established.
Information is exchanged by reading and writing data to the shared region.
Allows maximum speed and convenience of communication, making it faster than message passing.
System calls are required only to establish shared memory regions.

All accesses are treated as routine memory accesses, with no assistance required from kernel.
Shared-memory region typically resides in the • address space of the process creating the shared-memory segment.
Other processes that wish to communicate using this shared memory segment must attach it to their address space.
They can then exchange information by reading and writing data in the shared area.
Form of data and location are determined by these processes and are not under the control of the OS. Processes need to ensure that they are not writing to the same location at the same time- simultaneously.
The code for accessing and manipulating the shared memory is written explicitly by the application programmer.

that required any kernel (system) intervention after the creation of shared memory.
so it's faster than message passing.

Communication is by means of messages exchanged between the cooperating processes. Useful for exchanging smaller amounts of data, because no conflicts need to be avoided, and is easier to implement.
Implement using system calls and thus requires the more time-consuming task of kernel intervention OS provides a mechanism for cooperating processes to communicate with each other via a message passing facility.
Communicate and synchronize actions without sharing the same address space
Useful in a distributed environment, where the communicating processes may reside on different computers connected by the network. E.g. WhatsApp.



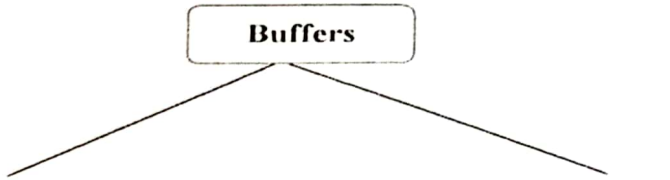
Interprocess communication (IPC) models (a) message passing (b) shared memory.

Cooperating processes- producer consumer problem illustrated

- Producer produces information that is consumed by a consumer process.
- Web server produces (that is, provides) HTML files and images, which are consumed (that is, read) by the client browser requesting the resource.

One solution to the producer consumer problem uses shared memory:

- Concurrent execution of producer and consumer processes requires a buffer of items filled by the producer and emptied by the consumer. The buffer resides in a region of memory shared by the producer and consumer processes.
- Synchronization is key so that the consumer does not try to consume an item that has not yet been produced.



Bounded buffer (fixed size) **Unbounded buffer**

- Consumer must wait if the buffer is empty.
- Producer must wait if the buffer is full.
- Buffer is empty when in==out
- Buffer is full when ((in+ 1)% BUFFER_SIZE)==out

Message passing system

- provides at least two operations: send (message) by producer and receive (message)
- messages sent by a process can be of either fixed or variable size.

Communication between P&Q

- They must send messages to and receive messages from each other.
- A communication link must exist between them, either physical implementation (shard memory, hardware bus or network or logical).

```
item nextConsumed;

while (true) {
    while (in == out)
; // do nothing

    nextConsumed =
    buffer[out]; out =
    (out + 1) %
    BUFFELSIZE;
    /* consume the item in nextConsumed */
}
```

Figure 1: The consumer process

```
#define BUFFER_SIZE 10
typedef struct
}item;
item buffer[BUFFER_SIZE];
int in = 0; int out = 0;
```

Shared buffer implemented as a circular array with two logical pointers in and out.

variable *in* points to the next free position in the buffer.
***out* points to the first full position in the buffer.**

```
item nextProduced;

while (true) {
    /* tredre m ::e:  :e:':Er:c::ec */  -
    while (((in + 1) %BITFER_SIZE)
    == out)
      ; /+ do nothing */ buffer[in] =
            nextProduced;
```

Figure 2: The producer process

Local variable nextProduced in which the new item to be produced is stored.

Maximum of BUFFER-1 items in buffer at the same time.

Logically implementing a link and the Send () / receive () operations

- Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering:

Naming

- Processes that want to communicate must have a way to refer to each other
- They can either use Direct or indirect communication

1. Direct Communication

- Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication i.e exhibits symmetry in addressing
- In this scheme the Send() & receive() primitives are defined as follows:
    - send (P, message)- Send a message to process P.
    - receive (Q, message)- Receive a message from process Q.

Properties

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

Variant of Symmetry in addressing

- A variant of direct communication scheme employs asymmetry in addressing: here, only the sender names the recipient while the recipient is not required to name the sender.
- Here, send() is defined normally while primitive receive() is defined by :
    - 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.

Disadvantages of direct communication (symmetric and asymmetric)

- Limited modularity of the resulting process definitions.
- Changing the identifier of a process may necessitate examining all other process definitions.
- All references to the old identifier must be found, so that they can be modified to the new identifier.
- In general, any such hard-coding techniques, where identifiers must be explicitly stated, are less desirable than techniques involving indirection.

2. Indirect Communication

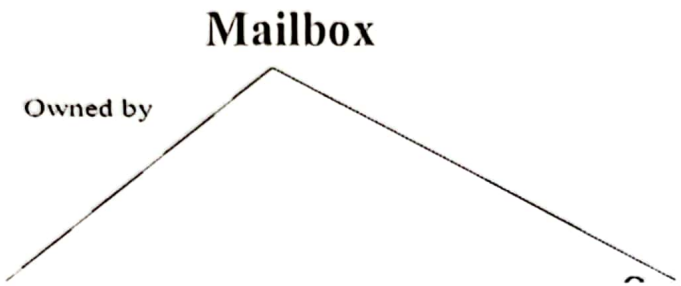
- Messages are sent to and received from mail boxes, or ports.
- A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
- Each mailbox has a unique identification.
- In this scheme, a process can communicate with some other processes via a number of mailboxes. Two processes can communicate only if the processes have a shared mailbox.

- In this scheme the Send() & receive() primitives are defined as follows:
  - send (A, message)- Send a message to mailbox A.
  - receive (QA, message)- Receive a message from mailbox A.

Properties of the communication link

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mailbox.

- Now suppose that processes P1, P2, and P3 all share mailbox A. Process P1 sends a message to A, while both P2 and P3 execute a receive 0 from A.
- Which process will receive the message sent by P1? The answer depends on which of the following methods we choose:
  - Allow a link to be associated with two processes at most.
  - Allow at most one process at a time to execute a receive 0 operation.
  - Allow the system to select arbitrarily which process will receive the message (that is, either P2 or P3, but not both, will receive the message).
    The system also may define an algorithm for selecting which process will receive the message (that is, round robin, where processes take turns receiving messages). The system may identify the receiver to the sender.



Process

- Part of the address space of process
- Distinguish between the owner (which can only receive messages through this mailbox) and the user (which can only send messages to this mailbox)
- When the process that owns the mailbox terminates mailbox disappears.
- Mailbox owner by default
- The OS then must provide a mechanism that allow a process to do the following:
  - Create a new mailbox
  - Send and receive messages through the mailbox
  - Delete the mailbox.

Operating system

- Has an existence of its own
- It is independent and not attached to any particular process

The process that creates the mailbox is that mailbox's owner by default, the owner is the only process that can receive messages through this mailbox.

SYNCHRONIZATION

Communication between processes takes place through calls to send() and receive () primitives. There are different design options for implementing each primitive. Message passing may be either blocking or nonblocking also known as synchronous and asynchronous

• **Blocking send:**	The sending process is blocked until the message is received by the receiving process or by the mailbox
• **Nonblocking send:**	The sending process sends the message and resumes operation
• **Blocking receive:**	The receiver blocks until a message is available
• **Nonblocking receive:**	The receiver retrieves either a valid message or a null

Buffering

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

- **Zero capacity**: The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
- **Bounded capacity**: The 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 message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.
- **Unbounded capacity**: The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

The zero-capacity case is sometimes referred to as a message system with no buffering; the other cases are referred to as systems with automatic buffering.

QUESTIONS/CLARIFICATIONS

Create a producer process, consumer process and the communication will establish between producer and consumer