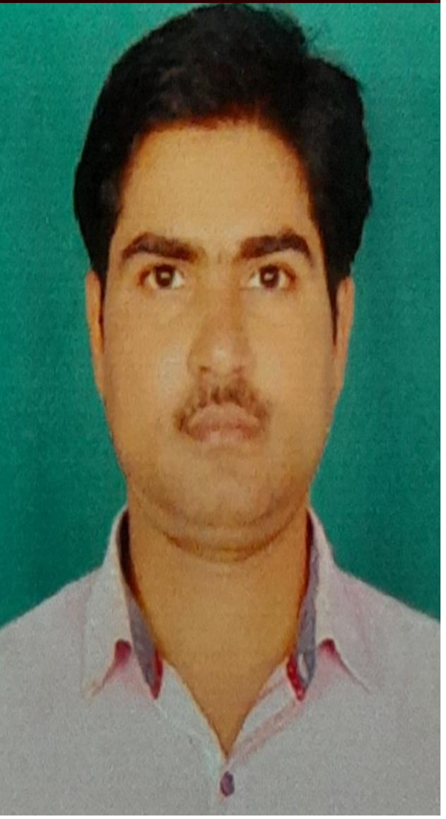
Operating system Class-BCA IV Semester



Dr. Vijay Kant Sharma

Assistant professor ,

Department Of Computer Application

Jagatpur P.G. College, Varanasi

Affiliated to Mahatma Gandhi Kashi vidhyapith Varanasi

Email-mzp.vijay@gmail.com

OUTLINE-

UNIT :- II

Operations on process

Process creation

Process termination

Interprocess communication

Shared memory ,Producer-Consumer problem

Buffering

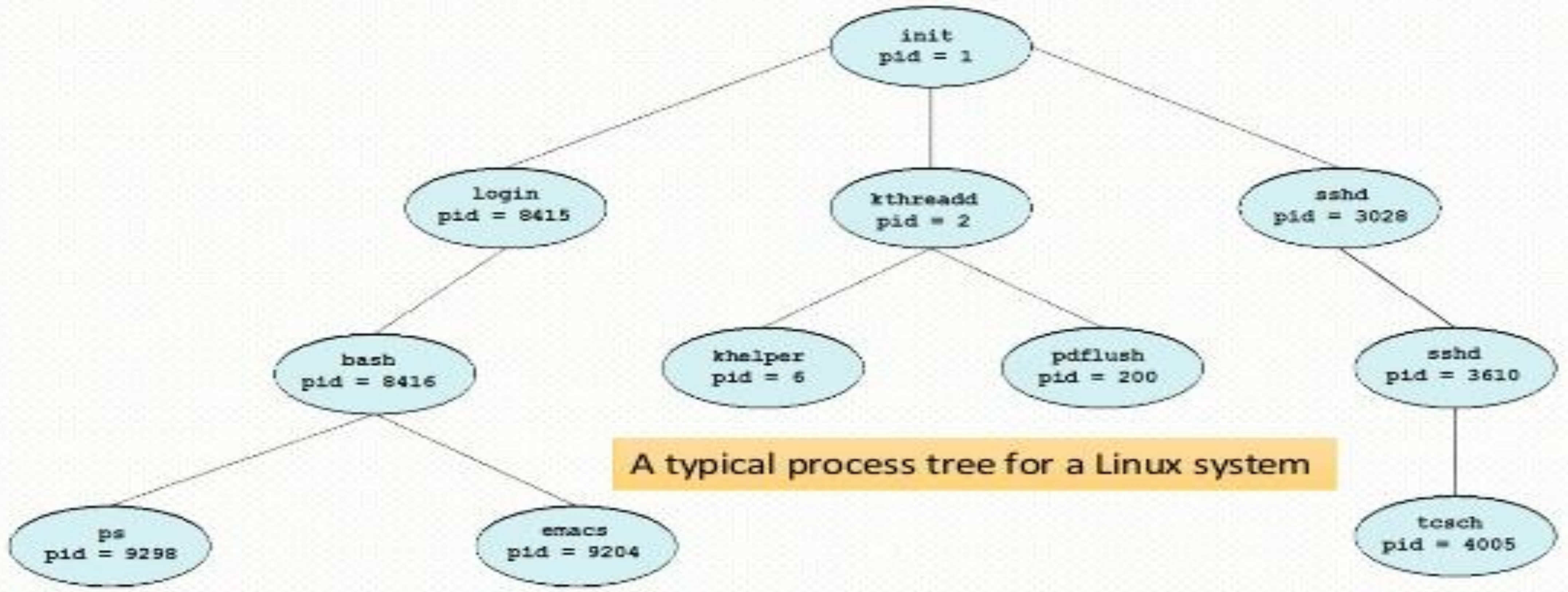
Operations on Processes

- The processes in the system can execute concurrently, and they must be created and deleted dynamically. Thus, the operating system must provide a mechanism (or facility) for process creation and termination.
- Other kinds of operations on processes include Deletion, Suspension, Resumption, Cloning, Inter-Process Communication and Synchronization

Process Creation

- During the course of execution, a process may create several new processes.
- The creating process is called a ***parent process***, and the new processes are called the ***children*** of that process. Each of these new processes may in turn create other processes, forming a tree of processes.
- Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique **process identifier (or pid)**, which is typically an integer number.
- The pid provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel.

- On typical UNIX systems the process scheduler is termed sched, and is given PID 0. The first thing it does at system startup time is to launch init, which gives that process PID 1.
- Init then launches all system daemons and user logins, and becomes the ultimate parent of all other processes.



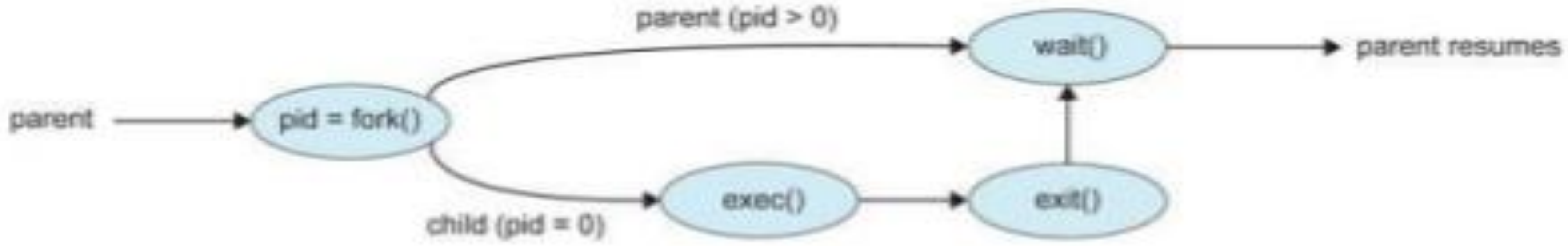
A typical process tree for a Linux system

- In general, when a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task.
- A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process.
- The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children.
- Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes.
- In addition to supplying various physical and logical resources, the parent process may pass along initialization data (input) to the child process.

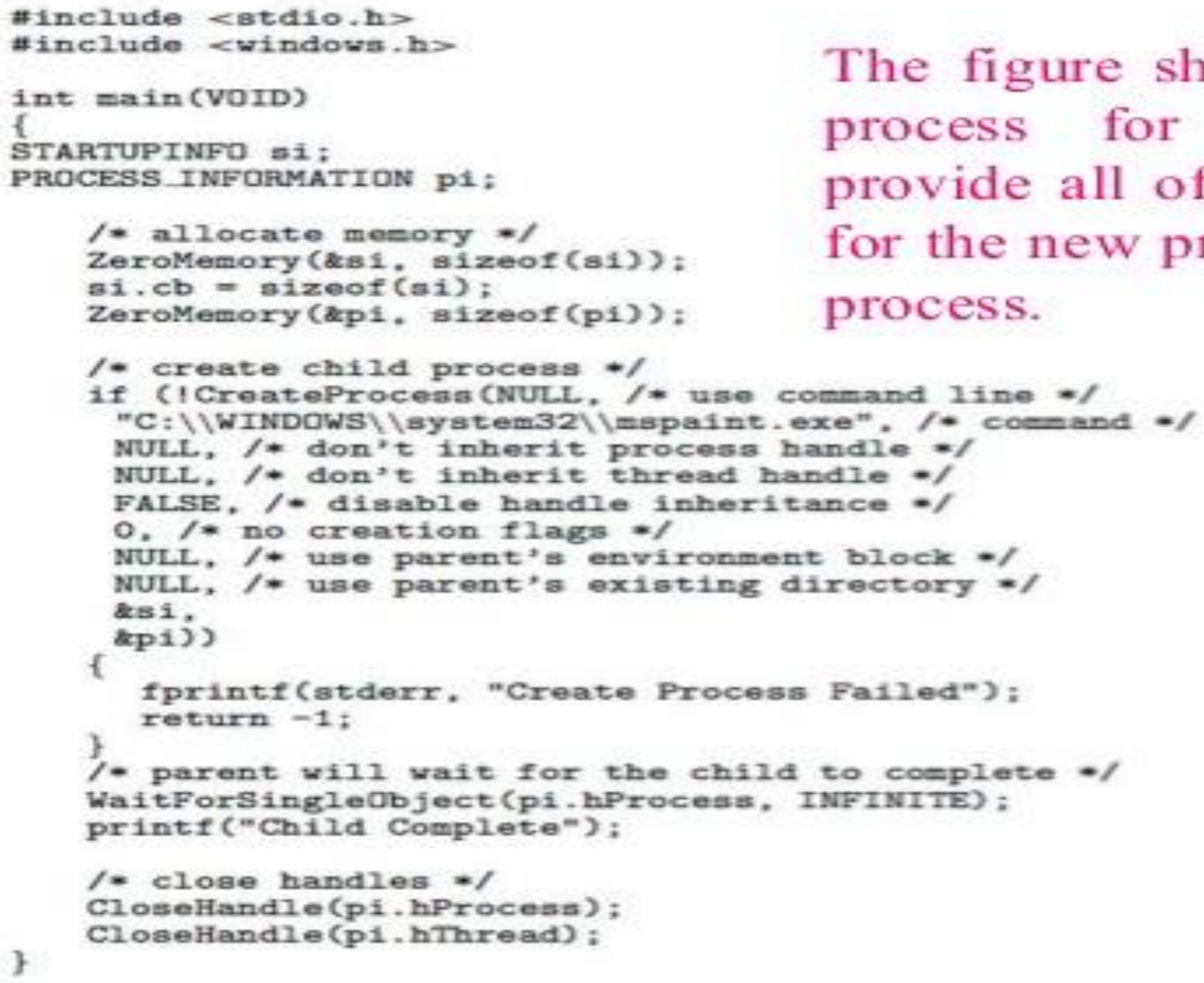
- There are two options for the parent process after creating the child:
  - Wait for the child process to terminate before proceeding. The parent makes a wait( ) system call, for either a specific child or for any child, which causes the parent process to block until the wait( ) returns. UNIX shells normally wait for their children to complete before issuing a new prompt.
  - Run concurrently with the child, continuing to process without waiting. This is the operation seen when a UNIX shell runs a process as a background task. It is also possible for the parent to run for a while, and then wait for the child later, which might occur in a sort of a parallel processing operation. (E.g. the parent may fork off a number of children without waiting for any of them, then do a little work of its own, and then wait for the children.)
- Two possibilities for the address space of the child relative to the parent:
  - The child may be an exact duplicate of the parent, sharing the same program and data segments in memory. Each will have their own PCB, including program counter, registers, and PID. This is the behavior of the fork in UNIX.
  - The child process may have a new program loaded into its address space, with all new code and data segments. This is the behavior of the spawn system calls in Windows. UNIX systems implement this as a second step, using the exec system call.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

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

   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }
   else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
   }
   else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
   }

   return 0;
}
```

- The figure shows the fork and exec process on a UNIX system.
- The fork system call returns the PID of the processes.
- It returns a zero to the child process and a non-zero child PID to the parent, so the return value indicates which process is which.
- Process IDs can be looked up any time for the current process or its direct parent using the getpid( ) and getppid( ) system calls respectively.



The figure shows the more complicated process for Windows, which must provide all of the parameter information for the new process as part of the forking process.



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

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION 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", /* command */
      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 */
      &si,
      &pi))
    {
       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);
}
```

Creating a Separate Process via Windows API

Process Termination

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call.
- All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.
- A process can cause the termination of another process via an appropriate system call. Such system calls are usually invoked only by the parent of the process that is to be terminated. A parent needs to know the identities of its children if it is to terminate them.

- A Parent may terminate the execution of one of its children for a variety of reasons, such as these:
   1. The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
   2. The task assigned to the child is no longer required.
   3. The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Some systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as cascading termination, is normally initiated by the operating system.

- When a process terminates, all of its system resources are freed up, open files flushed and closed, etc.
- The process termination status and execution times are returned to the parent if the parent is waiting for the child to terminate, or eventually returned to init if the process becomes an orphan.
- A process that has terminated, but whose parent has not yet called wait(), is known as a **zombie process**.
- If a parent did not invoke wait() and instead terminated, thereby leaving its child processes as **orphans**. Linux and UNIX address this scenario by assigning the init process as the new parent to orphan processes.
- The init process periodically invokes wait(), thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.

Interprocess Communication

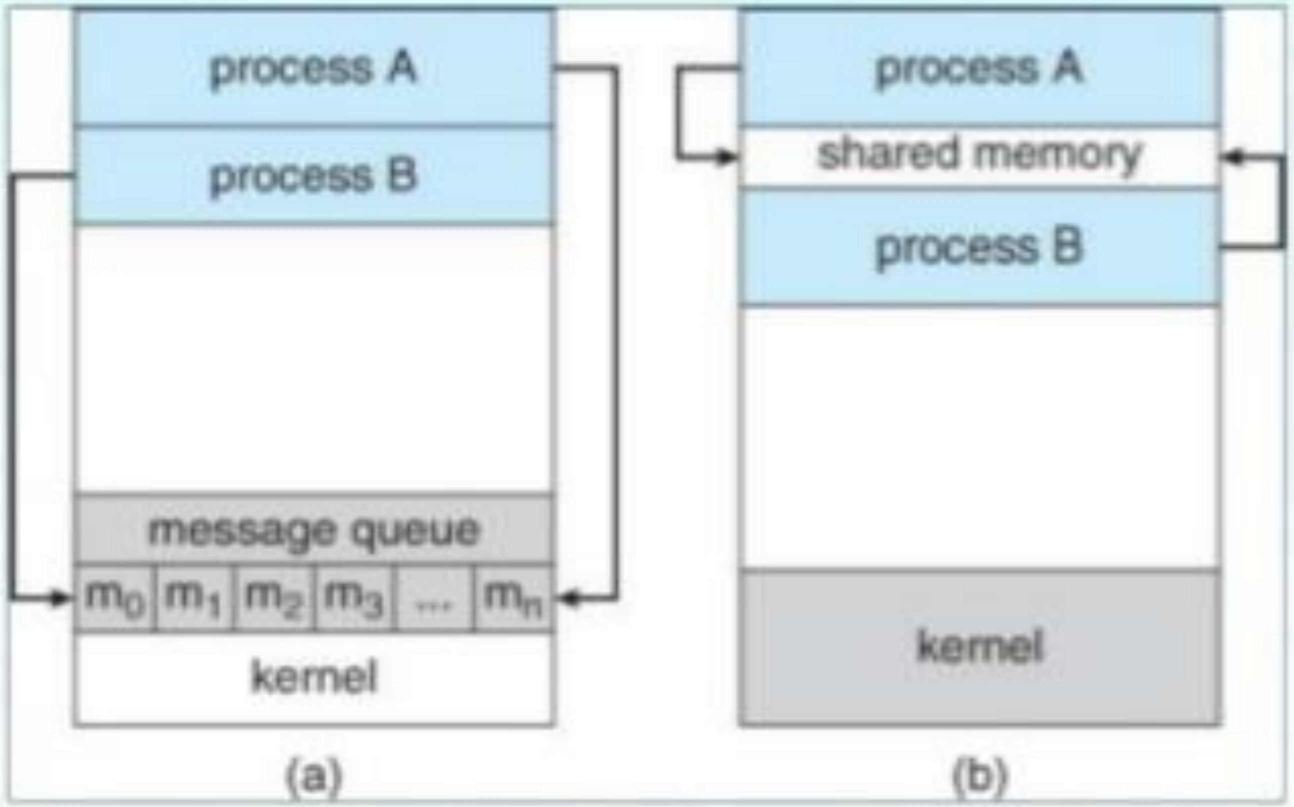
- Processes executing concurrently in the operating system may be either independent processes or cooperating processes.
  - *A process is* ***independent*** *if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent.*
  - *A process is* ***cooperating*** *if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.*

There are several reasons for providing an environment that allows process cooperation:

- **Information Sharing** - There may be several processes which need access to the same file. (e.g. pipelines.)
- **Computation speedup** - Often a solution to a problem can be solved faster if the problem can be broken down into sub-tasks to be solved simultaneously (particularly when multiple processors are involved.)
- **Modularity** - The most efficient architecture may be to break a system down into cooperating modules. (E.g. databases with a client-server architecture.)
- **Convenience** - Even a single user may be multi-tasking, such as editing, compiling, printing, and running the same code in different windows.

Cooperating processes require some type of inter-process communication, which is most commonly one of two types: **Message Passing systems (a) or Shared Memory systems(b)**

- Message Passing requires system calls for every message transfer, and is therefore slower, but it is simpler to set up and works well across multiple computers.
- Message passing is generally preferable when the amount and/or frequency of data transfers is small, or when multiple computers are involved.



- Shared Memory is faster once it is set up, because no system calls are required and access occurs at normal memory speeds. However it is more complicated to set up, and doesn't work as well across multiple computers. Shared memory is generally preferable when large amounts of information must be shared quickly on the same computer.

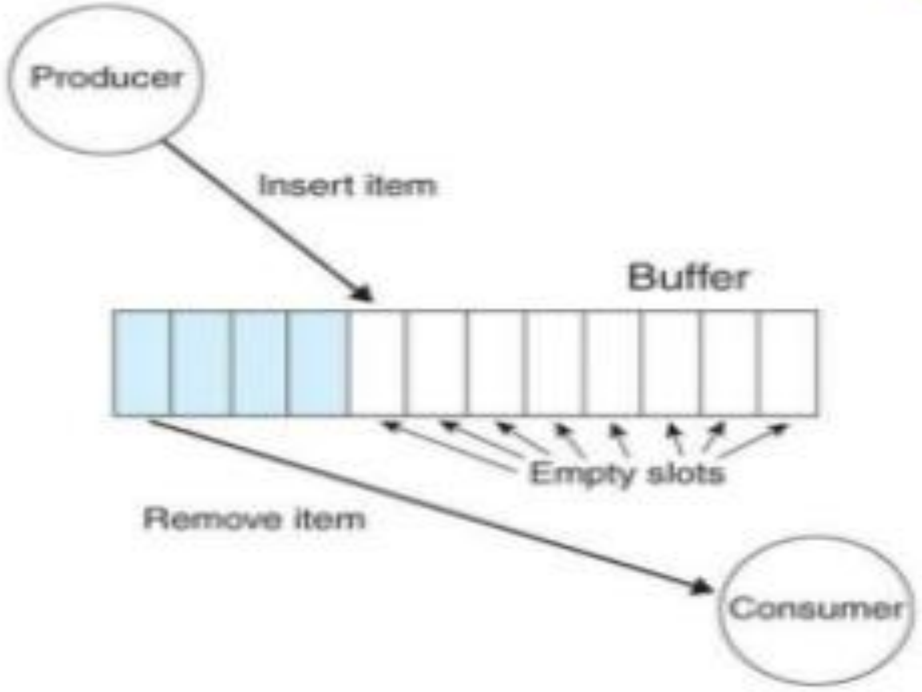
Shared Memory Systems

- Interprocess communication using shared memory requires communicating processes to establish a region of shared memory.
- Typically, a shared-memory region 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.
- Shared memory requires that two or more processes agree to remove the restriction of preventing one process accessing another processes memory.
- They can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by these processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

Producer-Consumer Example Using Shared Memory

- Producer-Consumer problem is a common paradigm for cooperating processes in which one process is producing data and another process is consuming the data.
- The producer–consumer problem provides a useful metaphor for the client–server paradigm. A server is thought as a producer and a client as a consumer.
- One solution to the producer–consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
- This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

- Two types of buffers can be used. The unbounded buffer places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.
- The bounded buffer assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.
- A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.



Producer Process:

```
while (true)
{
        /* produce an item in nextProduced */
        while (counter == BUFFER_SIZE)
           ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer Process:

```
while (true)
{
        while (counter == 0)
           ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in nextConsumed */
}
```

The producer process has a local variable next produced in which the new item to be produced is stored. The consumer process has a local variable next consumed in which the item to be consumed is stored.

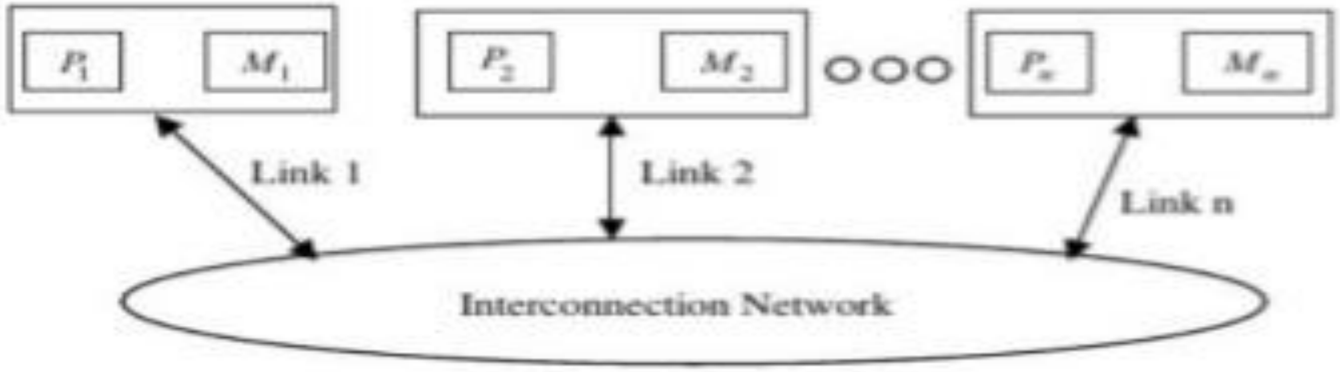
Message-Passing Systems

- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.
- It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.
- A message-passing facility provides at least two operations:

  send(message) **receive(message)**

- Messages sent by a process can be either fixed or variable in size.
  - If only fixed-sized messages can be sent, the system-level implementation is straightforward. However, makes the task of programming more difficult.
  - If it is variable-sized messages then it require a more complex system-level implementation, but the programming task becomes simpler.

- If processes P and Q want to communicate, they must send messages to and receive messages from each other: a communication link must exist between them. This link can be implemented in a variety of ways.
- Here are several methods for logically implementing a link and the send()/receive() operations:
  - Direct or indirect communication (Naming)
  - Synchronous or asynchronous communication
  - Automatic or explicit buffering



Message passing systems.

I P C

Naming

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

Under **direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, 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.*

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.*
- ➢*Between each pair of processes, there exists exactly one link.*

I P C

Naming

The previous scheme exhibits symmetry in addressing; that is, both the sender process and the receiver process must name the other to communicate.

A variant of this scheme employs asymmetry in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the send() and receive() primitives are defined as follows:

- *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.*

- The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions.
- Any such hard-coding techniques, where identifiers must be explicitly stated, are less desirable than techniques involving indirection.

Naming – Indirect Communication

I
P
C

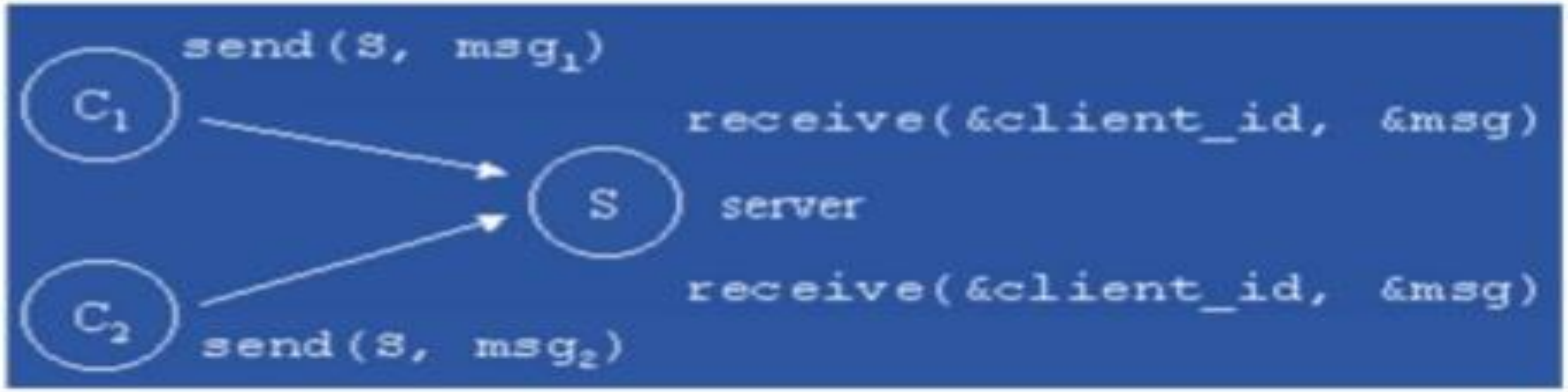
- With indirect communication, the messages are sent to and received from **mailboxes, or ports**.
- A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification.
- A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox. The send() and receive() primitives are defined as follows:
  - **send(A, message)—Send a message to mailbox A.**
  - **receive(A, message)—Receive a message from mailbox A.**
- In this scheme, a communication link has the following properties:
  - ➢ 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, a number of different links may exist, with each link corresponding to one mailbox

Naming – Indirect Communication

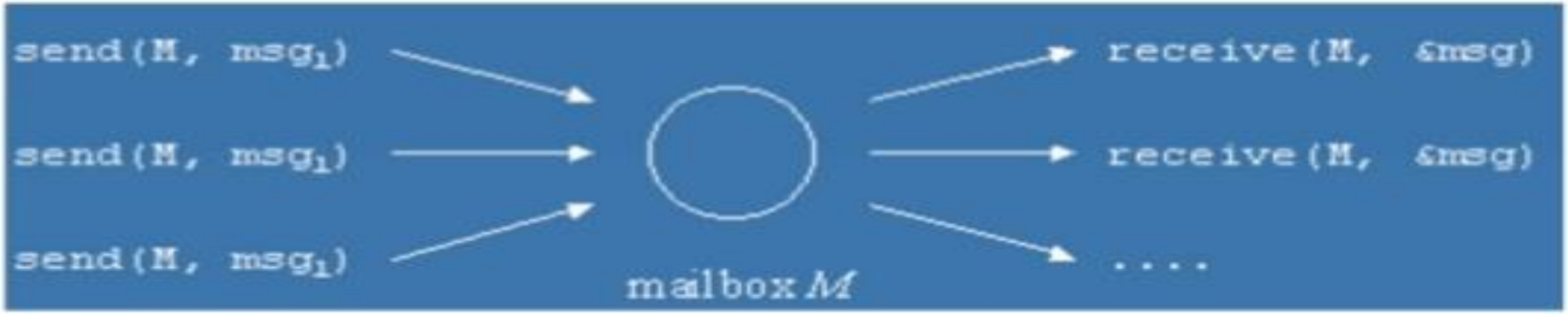
- A mailbox may be owned either by a process or by the OS.
- If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (which can only receive messages through this mailbox) and the user (which can only send messages to the mailbox).
- Since each mailbox has a unique owner, there can be no confusion about which process should receive a message sent to this mailbox.
- When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.
- A mailbox owned by an OS is independent and is not attached to any process. Then the OS needs to provide a mechanism to do the following:
    - Create a new mailbox.
    - Send and receive messages through the mailbox.
    - Delete a mailbox.

IPC

Direct communication



Indirect communication



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.*

I
P
C

I
P
C

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, and the sender can continue execution without waiting. 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.*

Reference Books:

1. Silberschatz and Galvin, “Operating System Concepts”, Person, 5
th
Ed. 2001
2. Madnick E., Donovan J., “Operating Systems:, Tata McGraw Hill,2001
3. Tanenbaum, “Operating Systems”, PHI, 4
th
Edition, 2000
4. Dietel, “Operating Systems”, TMH.

Declaration

The content is exclusively meant for academic purpose and for enhancement teaching and learning.Any other use of economic/commercial purpose is strictly prohibited. The users of the content shall not distribute ,disseminate or share it with anyone else and its use is restricted to advancement of individual knowledge.The information provided in this e-content is authentic and best as per my knowledge.

Dr.Vijay kant Sharma

Assistant professor

Department of computer Application

jagatpur P.G. College Varanasi Affiliated to Mahatma Gandhi Kashi Vidyapith Varanasi

Thanks