MODULE 2
PROCESS MANAGEMENT

CHAPTER 1 – PROCESSES BASIC CONCEPT

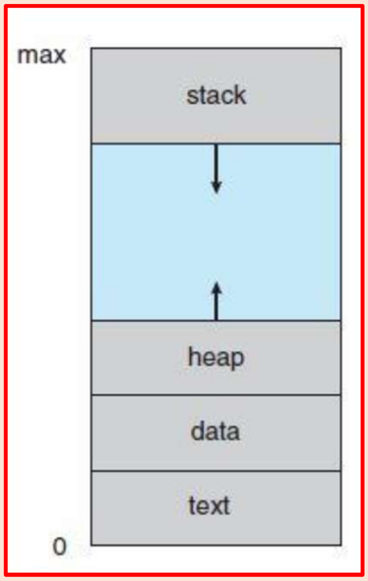
CO – Students will be able to discuss process basic concepts and process communication methods.

PROCESS

- A process is a program in execution.
- A process is a unit of work in a modern time sharing system.
- Process has a task and uses many resources such as CPU time, memory, files etc.
- Hardware resources and software resources are needed. These resources are allocated by the OS. Resources can be allocated in two manner.
  - One is, before the execution started. In this case, a problem arises, because the I/O devices are allocated for a long time up to the completion of the program execution. So the waiting time of I/O device is increased

- Secondly, the devices are allocated only when the need is arised. Here resource utilization is increased and waiting time is reduced.
- The allocation and deallocation of resources is performed by the OS. In normal case, the deallocation is performed only when the execution of process is completed. The two types of processes are
- **User executed process**
- **System executed process (Executed by the OS)**
- Process can be executed in concurrently. In concurrent execution, two or more processes are active, but at a time only one process is executed by the CPU.
- Concurrency control is needed for concurrent execution

A process is represented in memory as follows:



- **Text :** The actual code of the process is stored.
- **Data:** Global variables are stored
- **Heap:** Which is memory that is dynamically allocated during process run time.
- **Stack:** Which contains temporary data (such as function parameters, return addresses, and local variables)

- A program is a passive entity, such as a file containing a list of instructions stored on disk (often called an executable file).
- A process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources.
- A program becomes a process when an executable file is loaded into memory.
- Different copies of a program can be used by different users. For example, several users may be running different copies of the mail program. Similarly, Different copies of the same program can be used by one user. For example, same user may invoke many copies of the web browser program. Each of these is a separate process, and, although the text sections are equivalent, the data sections vary.

PROCESS STATES

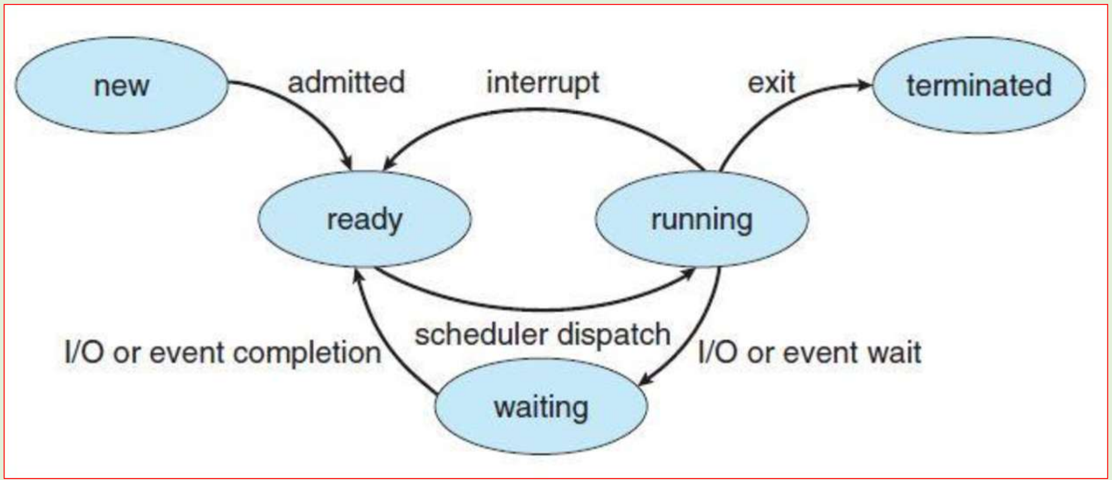


Figure: Diagram of process state

➢As a process executes, it changes state. Each process may be in one of the following states:

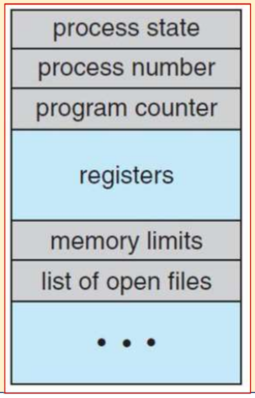
1. **New:** The process is being created.
2. **Running:** Instructions are being executed.
3. **Waiting:** The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
4. **Ready:** The process is waiting to be assigned to a processor.
5. **Terminated:** The process has finished execution.

- Only one process can be running on any processor at any instant, although many processes may be ready and waiting.

PROCESS CONTROL BLOCK(PCB)

➢Each process is represented in the operating system by a process control block (PCB) — also called a **task control block**. It contains many pieces of information associated with a specific process

- **Process state:** The state may be new, ready, running, waiting, halted, and so on.
- **Program counter:** The counter indicates the address of the next instruction to be executed for this process.



process state
process number
program counter
registers
memory limits
list of open files
• • •

- **CPU registers:** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward
- **CPU-scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information:** This information may include the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system

- **Accounting information:** This information includes time limits, account numbers, job or process numbers, and so on.
- **I/O status information:** list of I/O devices allocated to this process, a list of open files, and so on.

PROCESS SCHEDULING

- The objective of multiprogramming is to have some process running at all times, so as to maximize CPU utilization (reduce the idle time of CPU).
- The objective of time-sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.
- A uniprocessor system can have only one running process. If 5 more processes exist, the rest must wait until the CPU is free and can be rescheduled.
- If more process exists in memory, we must select the process. Here scheduling is done.

- As processes enter the system, they are put into a job queue. This queue consists of all processes in the system.
- The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ready queue. This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list.
- In the case of an I/O request, such a request may be to a shared device, such as a disk. Since the system has many processes, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a device queue. Each device has its own device queue

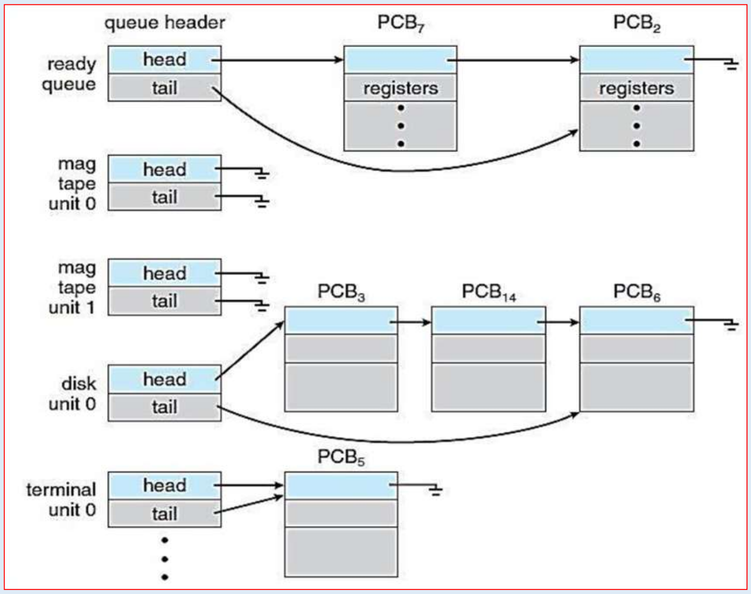
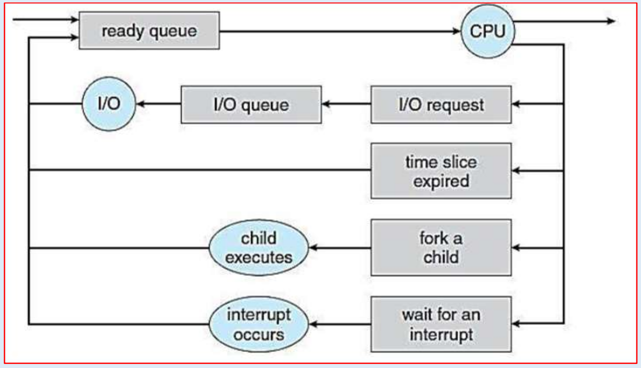


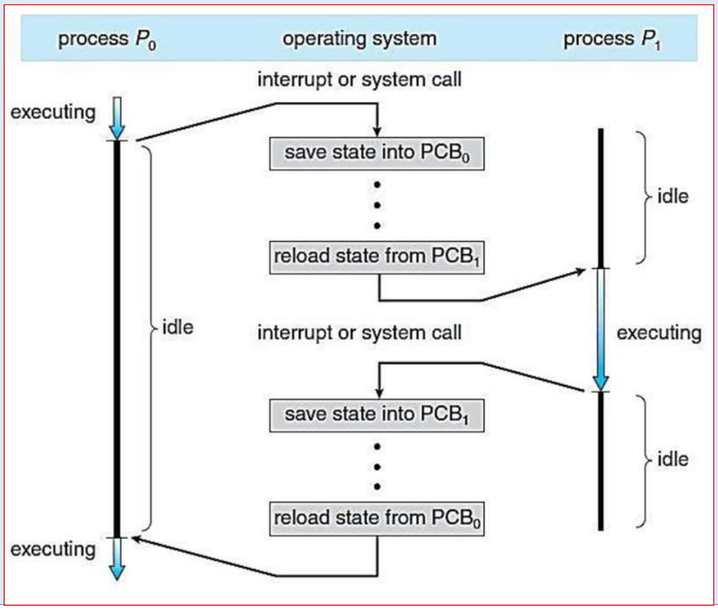
Figure: The ready queue and various I/O device queues.

❖**QUEUEING DIAGRAM** - Queueing diagram is used to represent process scheduling. Consider the following figure. Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.



- ➢A new process is initially put in the ready queue. It waits in the ready queue until it is selected for execution (or dispatched).
- ➢Once the process is assigned to the CPU and is executing, one of several events could occur:
  - ▪ The process could issue an I/O request, and then be placed in an I/O queue.
  - ▪ The process could create a new sub process (child process) and wait for its termination.
  - ▪ The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

Diagram showing CPU switches from process to process



SCHEDULERS

- All the jobs that enter in the system are kept in the job pool. The all processes in the job pool are in a job queue.
- The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate scheduler.
- **Different schedulers are:**
- **Long-term scheduler:** The long-term scheduler, or job scheduler, selects processes from job pool and loads them into memory for execution.

- **Short-term scheduler:** The short-term scheduler or CPU scheduler selects the process from main memory and allocates the CPU for executing the job.
- There is a difference between these two schedulers is the frequency of execution. Short term scheduler is more frequent because, When CPU selects a process, that process must have an I/O operation. So the CPU must select the next process quickly. So the frequency of CPU scheduler is high.
- The long-term scheduler, on the other hand, executes much less frequently. The long-term scheduler controls the degree of multiprogramming (the number of processes in memory).

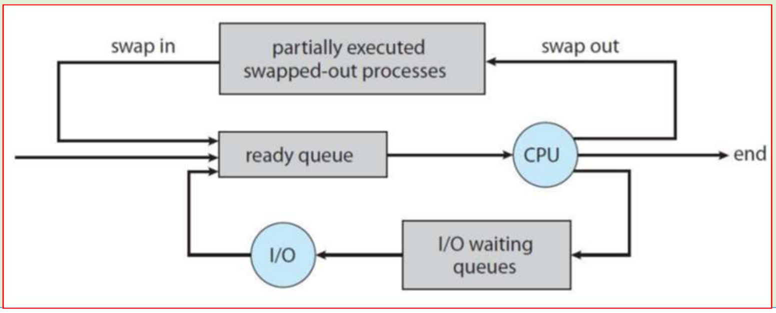
- If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.
- Thus, the long-term scheduler may need to be invoked only when a process leaves the system. Because of the longer interval between executions, the long-term scheduler can afford to take more time to select a process for execution.
- Most processes can be described as either I/O bound or CPU bound. An I/O bound process spends more of its time doing I/O than it spends doing computations. A CPU-bound process using more of its time doing computation than an I/O-bound process uses.

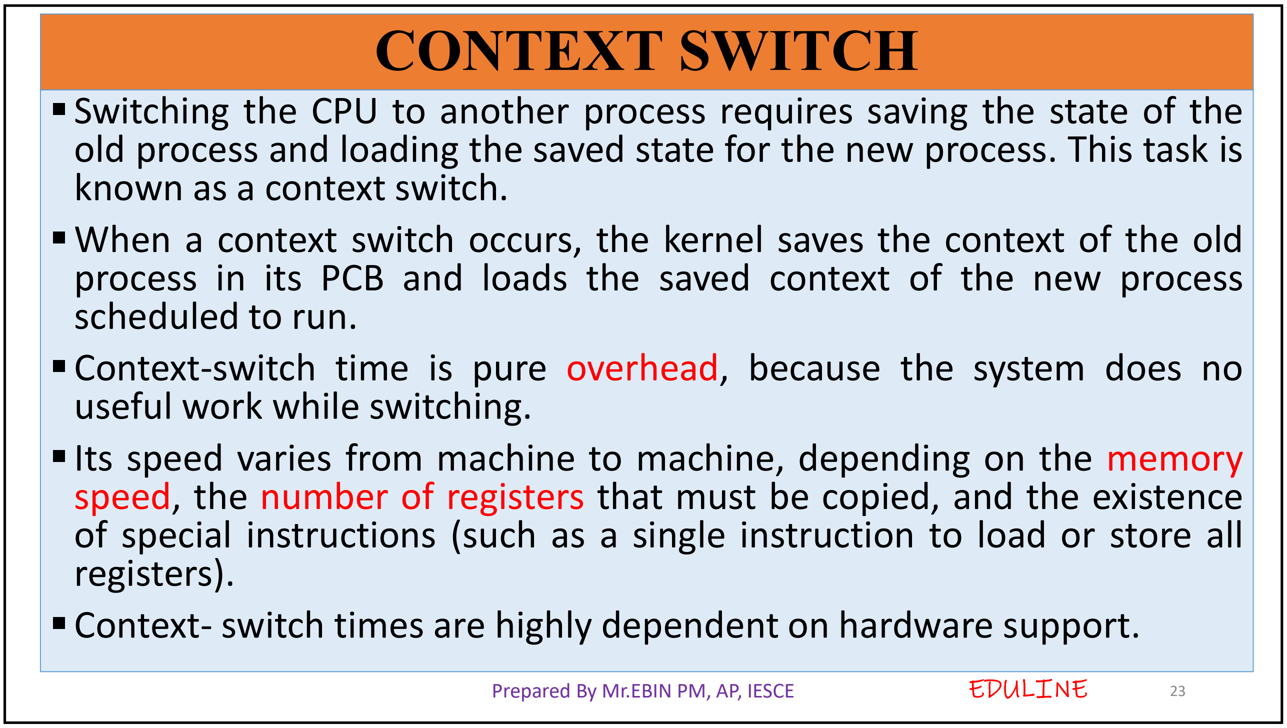
- The long-term scheduler should select a good process mix of I/O-bound and CPU-bound processes.
- If all processes are I/O bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do.
- If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced.
- The system with the best performance will have a combination of CPU-bound and I/O-bound processes

Medium-term scheduler

- On some systems, the long-term scheduler may be absent or minimal. For example, time-sharing systems such as UNIX often have no long-term scheduler.
- It introduces an additional scheduler called Medium-term scheduler, removes processes from memory (due to some performance reason) and thus reduces the degree of multiprogramming.
- Later on this process may be put in to the memory again and its execution can be again started from where it was left off. This process of moving a process in and out of the memory is called swapping

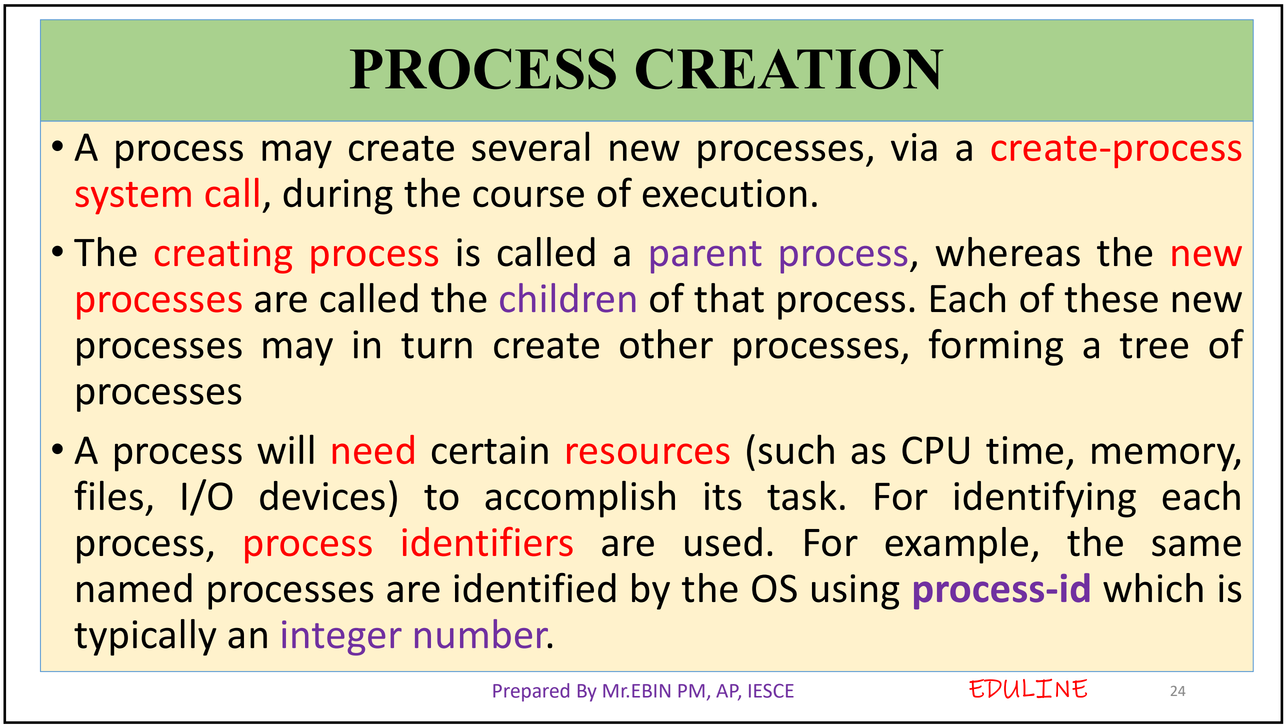
- Sometimes swapping becomes essential to deal with the situation when a running process requests more RAM. It can be given by moving (swapping) a ready process to the disk and making RAM available to the requesting process.
- The main function of medium term scheduler is to perform swap-in and swap-out of processes. Swapping is necessary to improve the process mix. All versions of windows OS uses swapping.





CONTEXT SWITCH

- Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process. This task is known as a context switch.
- When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.
- Context-switch time is pure overhead, because the system does no useful work while switching.
- Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers).
- Context- switch times are highly dependent on hardware support.



PROCESS CREATION

- A process may create several new processes, via a create-process system call, during the course of execution.
- The creating process is called a parent process, whereas 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
- A process will need certain resources (such as CPU time, memory, files, I/O devices) to accomplish its task. For identifying each process, process identifiers are used. For example, the same named processes are identified by the OS using **process-id** which is typically an integer number.

- A process may create several subprocesses. When a process creates a subprocess, that subprocess may be able to obtain its resources. The resources are allocated in two ways
  - The resources can be obtained directly from OS
  - Parent process gives the resources.
- The parent process gives its resources in two ways.
  - Parent divide its resources and give one part to the child
  - The parent and the child share the available resources.
- The execution of the processes is done in two ways
  - The parent continues to execute concurrently with its children.
  - The parent waits until some or all of its children have terminated.

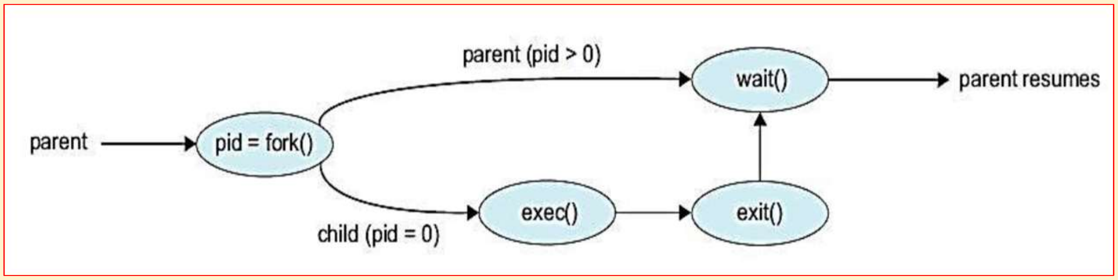
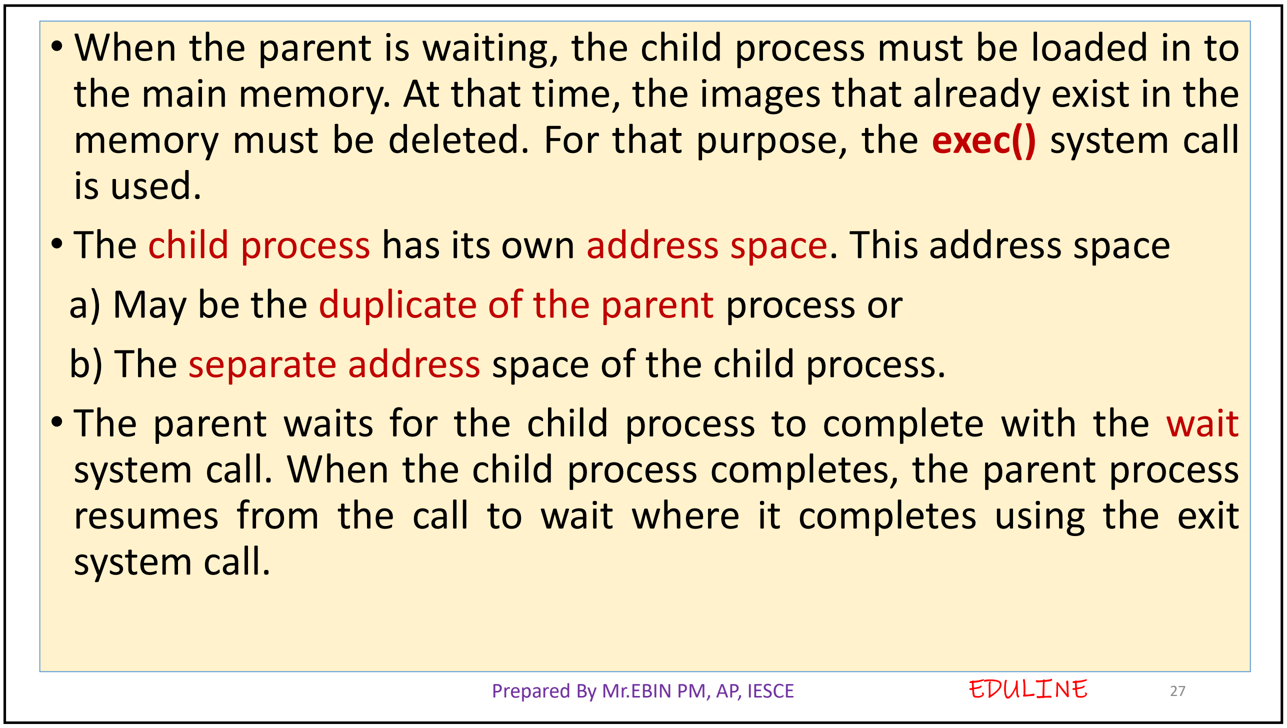


Figure: Process creation using the **fork()** system call

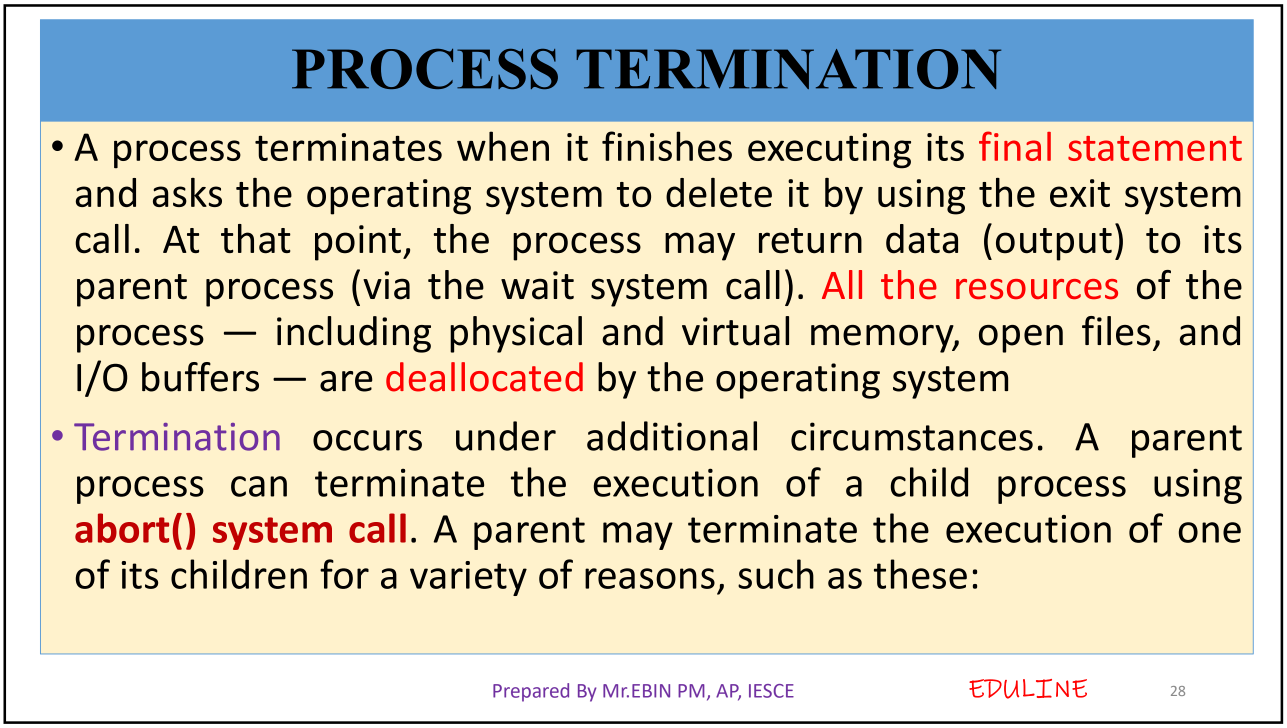
- In UNIX, each process is identified by its process identifier, which is a unique integer. A new process is created by the fork system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process



- When the parent is waiting, the child process must be loaded in to the main memory. At that time, the images that already exist in the memory must be deleted. For that purpose, the **exec()** system call is used.
- The child process has its own address space. This address space

  a) May be the duplicate of the parent process or

  b) The separate address space of the child process.
- The parent waits for the child process to complete with the wait system call. When the child process completes, the parent process resumes from the call to wait where it completes using the exit system call.



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. At that point, the process may return data (output) to its parent process (via the wait 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
- Termination occurs under additional circumstances. A parent process can terminate the execution of a child process using **abort() system call**. A parent may terminate the execution of one of its children for a variety of reasons, such as these:

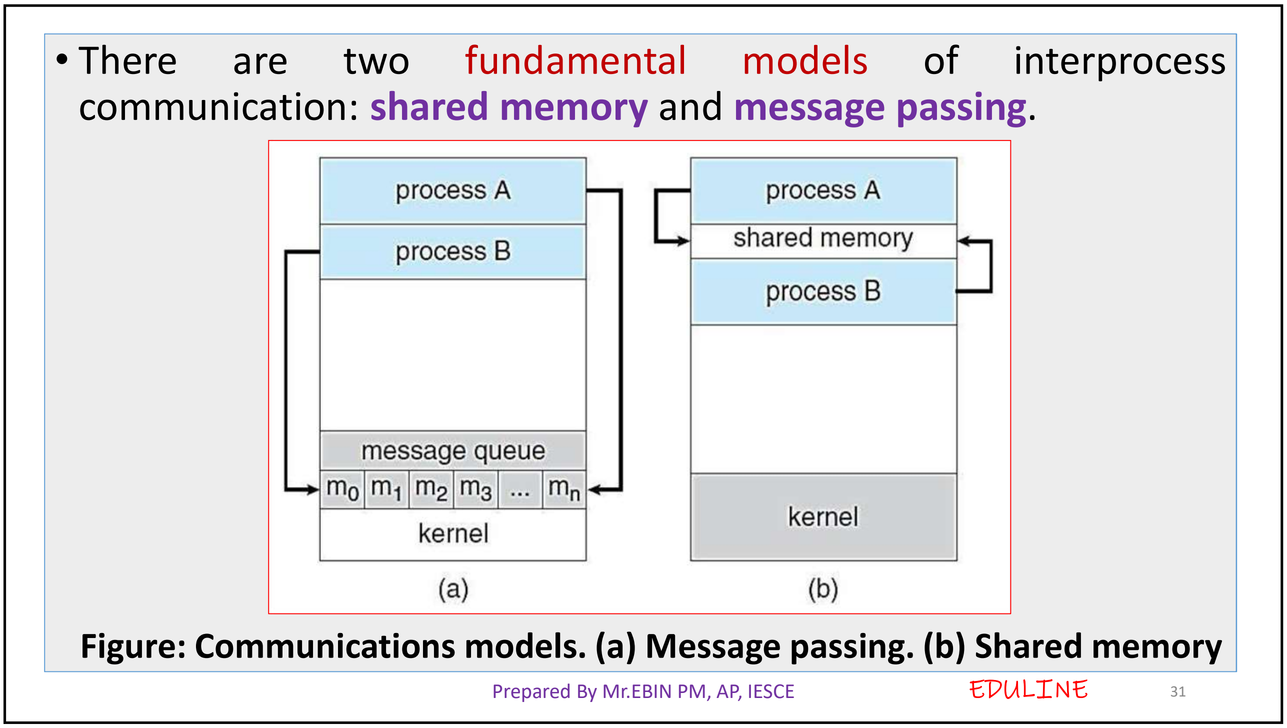
a)The child has exceeded its usage of some of the resources that it has been allocated. This requires the parent to have a mechanism to inspect the state of its children.

b) The task assigned to the child is no longer required. The parent process may create additional children for helping the parent. After some times, if the parent can succeed without the help of the child process, then the parent aborts the process.

c) The parent is exiting, and the operating system does not allow a child to continue if its parent terminates. On 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**.

INTERPROCESS COMMUNICATION(IPC)

- 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.
- Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information.



- There are two fundamental models of interprocess communication: **shared memory** and **message passing**.

Figure: Communications models. (a) Message passing. (b) Shared memory

- In the shared-memory model, a region of memory that is shared by cooperating processes . Processes can then exchange information by reading and writing data to the shared region.
- In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.
- Message passing is useful for exchanging smaller amounts of data. Message passing is easier to implement in a distributed system.
- Shared memory can be faster than message passing, since message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention.

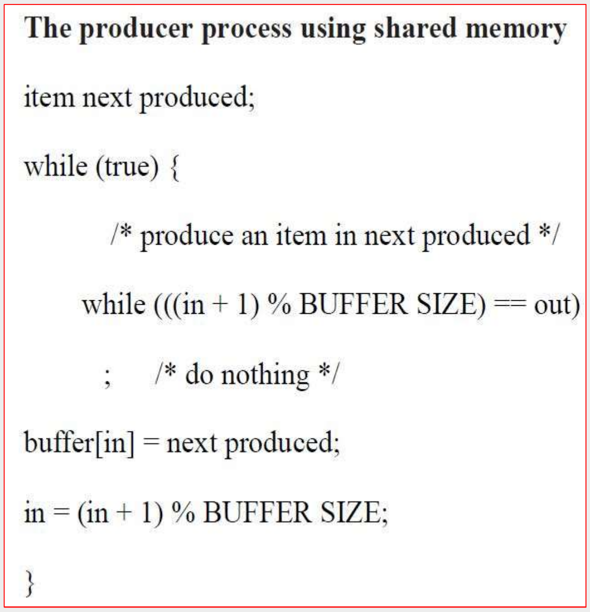
- In shared-memory systems, system calls are required only to establish shared memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

❖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.

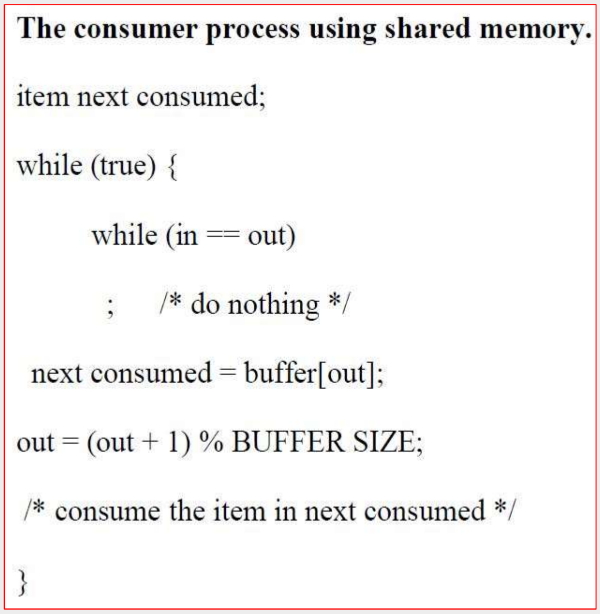
- To illustrate the concept of cooperating processes, let’s consider the producer–consumer problem, which is a common paradigm for cooperating processes.
- A producer process produces information that is consumed by a consumer process. For example, a compiler may produce assembly code that is consumed by an assembler.
- 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.

- 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 bounded buffer assumes a fixed buffer size
- The shared buffer is implemented as a circular array with two logical pointers: **in** and **out**. The variable **in** points to the next free position in the buffer; **out** points to the first full position in the buffer.
- The buffer is empty when **in==out**;
- The buffer is full when **((in + 1) % BUFFER SIZE) == out**.



The producer process using shared memory

```
item next produced;
while (true) {
        /* produce an item in next produced */
    while (((in + 1) % BUFFER SIZE) == out)
        ;   /* do nothing */
buffer[in] = next produced;
in = (in + 1) % BUFFER SIZE;
}
```



The consumer process using shared memory.

```
item next consumed;
while (true) {
    while (in == out)
        ;   /* do nothing */
 next consumed = buffer[out];
out = (out + 1) % BUFFER SIZE;
 /* consume the item in next consumed */
}
```

❖Message-Passing Systems

- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.
- For example, an Internet chat program could be designed so that chat participants communicate with one another by exchanging messages.
- 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.

- There are several methods for logically implementing a link

➢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 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.
- This scheme exhibits symmetry in addressing.
- A variant of this scheme employs asymmetry in addressing.

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.
- With indirect communication, the messages are sent to and received from mailboxes.
- 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.

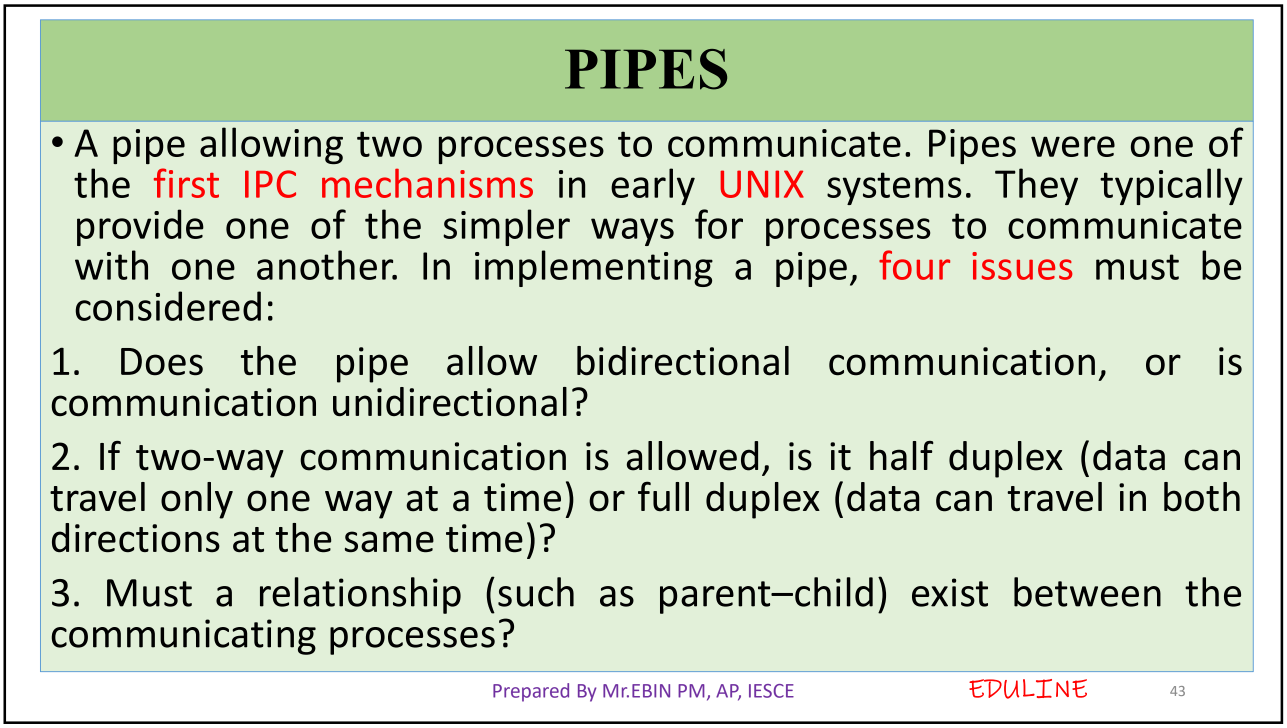
receive(A, message)—Receive a message from mailbox A.

❖Synchronization

- Message passing may be either **blocking** or **non-blocking** 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.
  - Non-blocking send: The sending process sends the message and resumes operation.
  - Blocking receive: The receiver blocks until a message is available.
  - Non-blocking 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. 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.



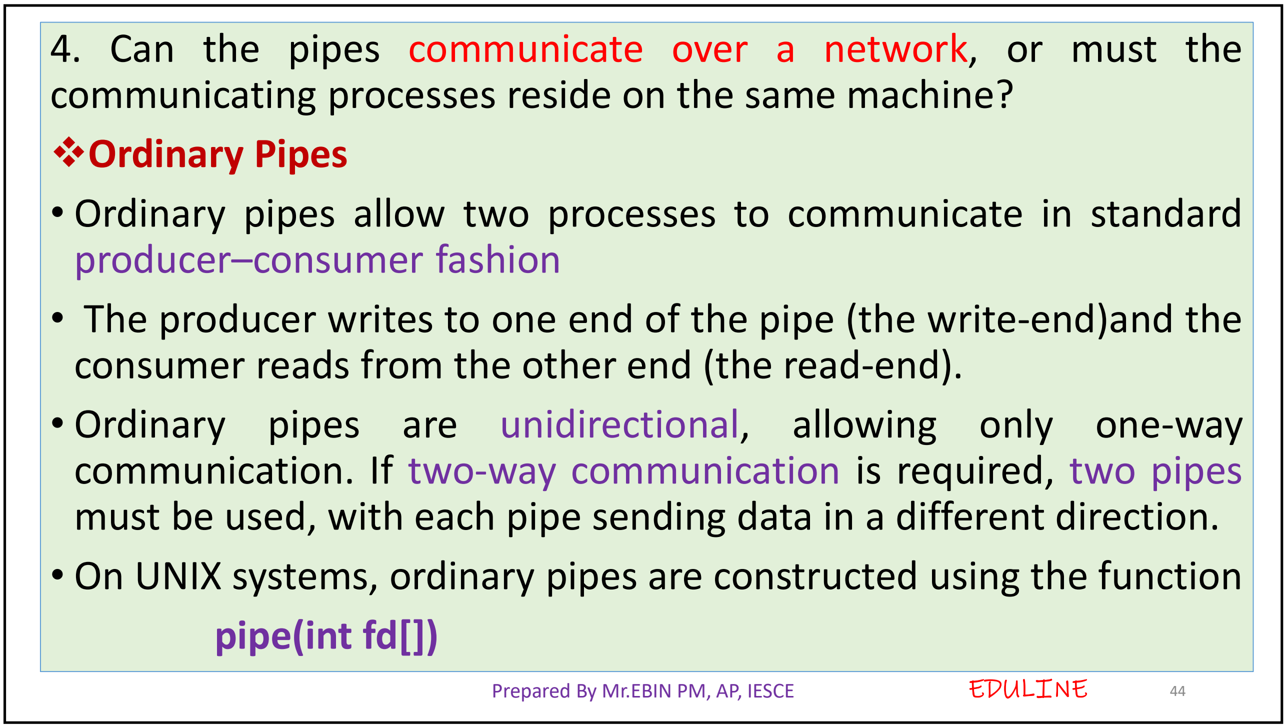
PIPES

- A pipe allowing two processes to communicate. Pipes were one of the first IPC mechanisms in early UNIX systems. They typically provide one of the simpler ways for processes to communicate with one another. In implementing a pipe, four issues must be considered:

1. Does the pipe allow bidirectional communication, or is communication unidirectional?

2. If two-way communication is allowed, is it half duplex (data can travel only one way at a time) or full duplex (data can travel in both directions at the same time)?

3. Must a relationship (such as parent–child) exist between the communicating processes?



4. Can the pipes communicate over a network, or must the communicating processes reside on the same machine?

❖Ordinary Pipes

- Ordinary pipes allow two processes to communicate in standard producer–consumer fashion
- The producer writes to one end of the pipe (the write-end)and the consumer reads from the other end (the read-end).
- Ordinary pipes are unidirectional, allowing only one-way communication. If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction.
- On UNIX systems, ordinary pipes are constructed using the function

pipe(int fd[])

- UNIX treats a pipe as a special type of file. Thus, pipes can be accessed using ordinary read() and write() system calls.
- A parent process creates a pipe and uses it to communicate with a child process that it creates via fork().
- Ordinary pipes on Windows systems are termed anonymous pipes.
- Ordinary pipes require a parent–child relationship between the communicating processes on both UNIX and Windows systems.
- These pipes can be used only for communication between processes on the same machine.

❖Named Pipes

- Named pipes provide a much more powerful communication tool.
- Communication can be bidirectional, and no parent–child relationship is required.
- Once a named pipe is established, several processes can use it for communication.
- A named pipe has several writers.
- Named pipes continue to exist after communicating processes have finished.
- Both UNIX and Windows systems support named pipes.

- Named pipes on Windows systems provide Full-duplex communication is allowed, and the communicating processes may reside on either the same or different machines.
- Named pipes are created with the CreateNamedPipe() function, and a client can connect to a named pipe using ConnectNamedPipe().
- Communication over the named pipe can be accomplished using the ReadFile() and WriteFile() functions.

THREADS

- A thread, sometimes called a lightweight process (LWP), is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack.
- If the process has multiple threads of control, it can do more than one task at a time.
- Many software packages that run on modern desktop PCs are multithreaded.
- A word processor may have a thread for displaying graphics, another thread for reading keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

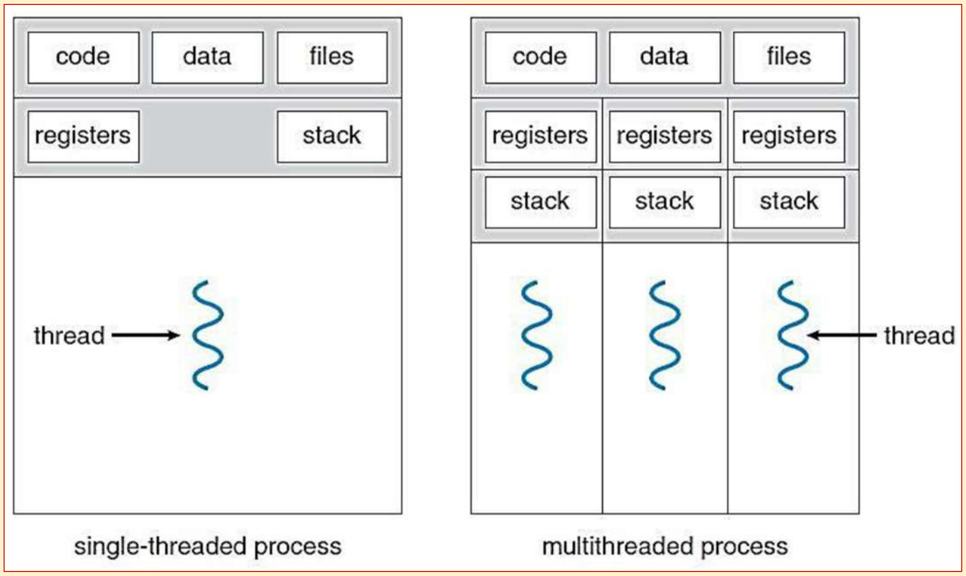


Figure: Single-threaded and multithreaded processes.

- Processes are heavy weight task that require their own separate address space. Threads on the other hand are light weight.
- Thread share the same address space.
- Thread shares the resources. So the thread creation is simple.

❖**Benefits**

- **Responsiveness:** Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
- **Resource sharing:** threads share the memory and resources of the process.

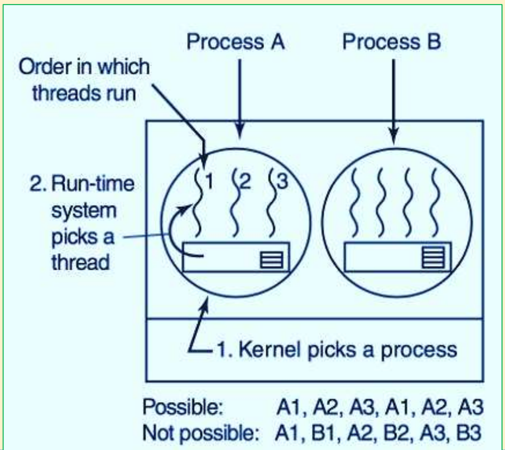
- **Economy:** Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.
- **Scalability:** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.

❖User Level Threads

- Here threads are implemented in user level libraries
- These are fast to create and manage because no kernel intervention

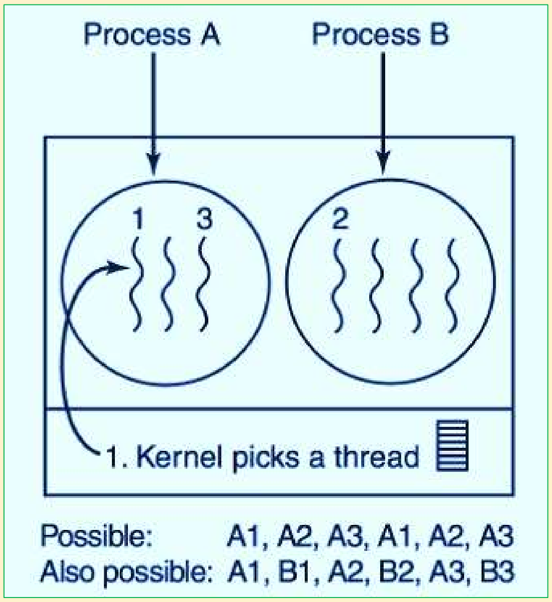
- Here kernel is not aware of the existence of thread. It pick a process and give control. The thread scheduler inside the process decides which thread to run.
- Since there are no clock interrupts to multiprogrammed threads, the thread may continue running as long as it wants to. If it uses up the process's entire time quantum , the kernel will select another process to run.
- The scheduling algorithm used by the runtime system can be Round-Robin scheduling and priority scheduling.
- **Drawback -** If kernel is a single threaded, then any user level thread performing a blocking system call will cause the entire process to block.

- Run-time system do the operations related to the thread. It contains a collection of Procedures for thread manipulation.
- Thread table is used for keeping the information about thread PC value, stack information's etc.



❖Kernel Level Threads

- Kernel does thread management
- Kernel performs thread creation, scheduling and management in kernel space.
- The OS is aware of the presence of threads in the processes, therefore even if one thread of a process gets blocked, the OS chooses the next one to run, either from the same process or from the different one.
- Kernel threads are generally slower to create and manage.
- The OS will handle blocking system call in kernel level.
- Kernel level thread is efficient in multiprocessing environment.



Possible: A1, A2, A3, A1, A2, A3

Also possible: A1, B1, A2, B2, A3, B3