KTU NOTES

The learning companion.

KTU STUDY MATERIALS | SYLLABUS | LIVE NOTIFICATIONS | SOLVED QUESTION PAPERS

Website: www.ktunotes.in

Module II

Processes and Process Scheduling

CONTENTS

Processes - Process states, Process control block, threads, scheduling, Operations on processes - process creation and termination – Inter-process communication - shared memory systems, Message passing systems.

- **Process Scheduling –** Basic concepts- Scheduling criteria, Scheduling algorithms, First come First Served, Shortest Job First, Priority scheduling, Round robin scheduling

Processes

- A process is a program in execution.
- A process is just an instance of an executing program, including the current values of the program counter, registers, and variables.
- Conceptually, each process has its own virtual CPU.
- In reality, the real CPU switches back and forth from process to process.
- This rapid switching back and forth is called **multiprogramming.**
- **Batch System:** Executes **Jobs**
- **Time Shared System:** Has User Program or **Tasks**

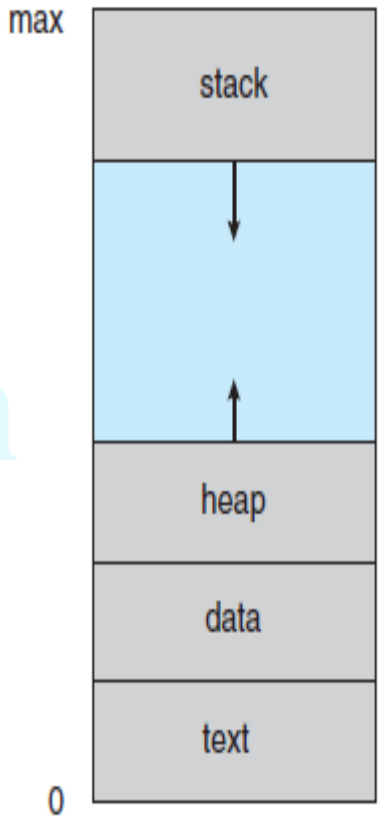
Process

A process has the program code, which is known as the **text section.**

Process also includes the current activity, as represented by the value of the **program counter and the contents of the processor's registers.**

A process generally also includes the process **stack, which contains temporary data (such as function** parameters, return addresses, and local variables), **Data section, which** contains global variables.

A process may also include a **heap, which is memory** that is dynamically allocated during process run time.



Process in memory.

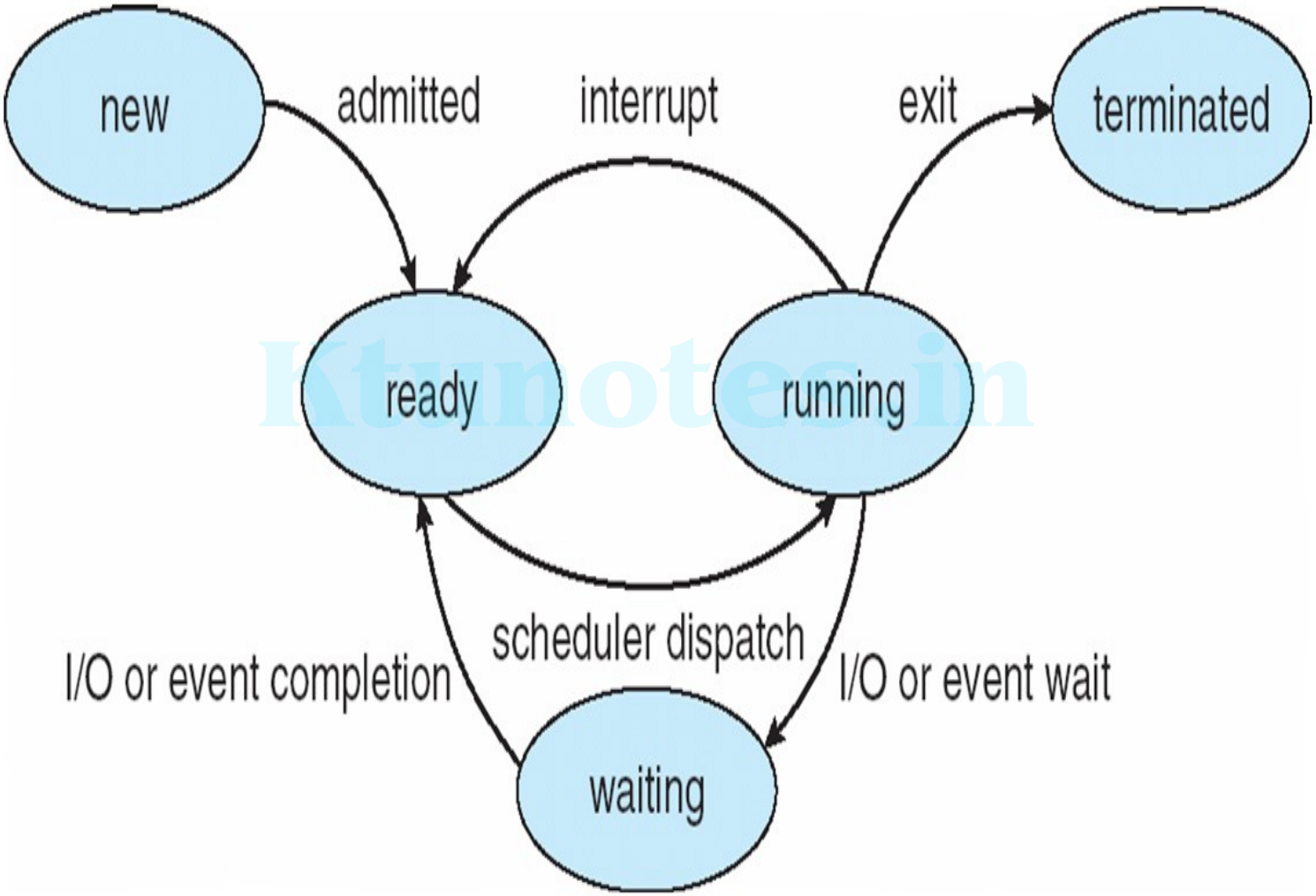
Process Concept (Cont.)

- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program.
- For instance, several users may be running different copies of the mail program, or the 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, heap, and stack sections vary.

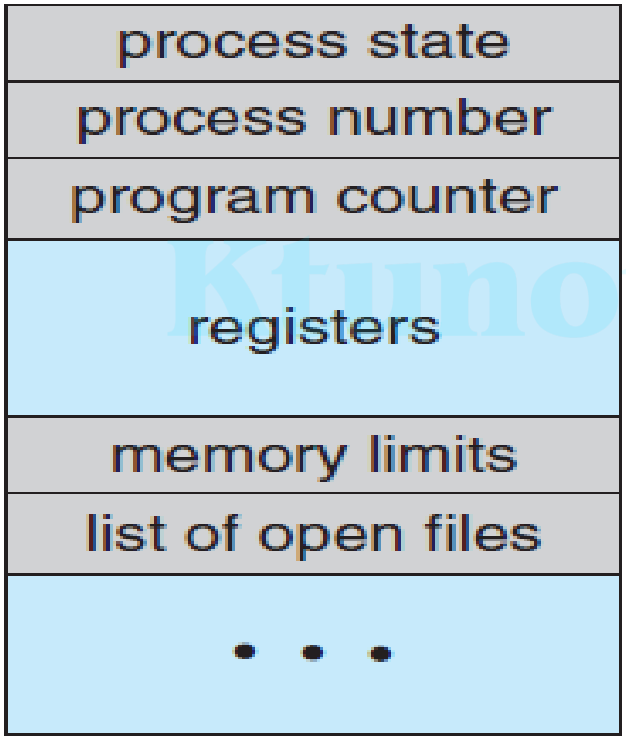
Process State

- As a process executes, it changes **state**
  - **new:** The process is being created
  - **running:** Instructions are being executed
  - **waiting:** The process is waiting for some event to occur
  - **ready:** The process is waiting to be assigned to a processor
  - **terminated:** The process has finished execution

Diagram of Process State



- Each process is represented in the operating system by a process control block (PCB) also called a task control block.



Process control block (PCB)

Process Control Block(cont..)

Process state : The state may be new, ready, running, waiting, terminated, and so on.

Program counter: The counter indicates the address of the next instruction to be executed for this process.

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, any condition-code information.

CPU-scheduling information: This information includes a process priority , pointers to scheduling queues, and any other scheduling parameters.

Process Control Block(cont..)

Memory-management information : This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system

Accounting information: This information includes the amount of CPU and real time used, time limits, job or process numbers, and so on.

I/O status information: This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

Process Control Block(Context switch)

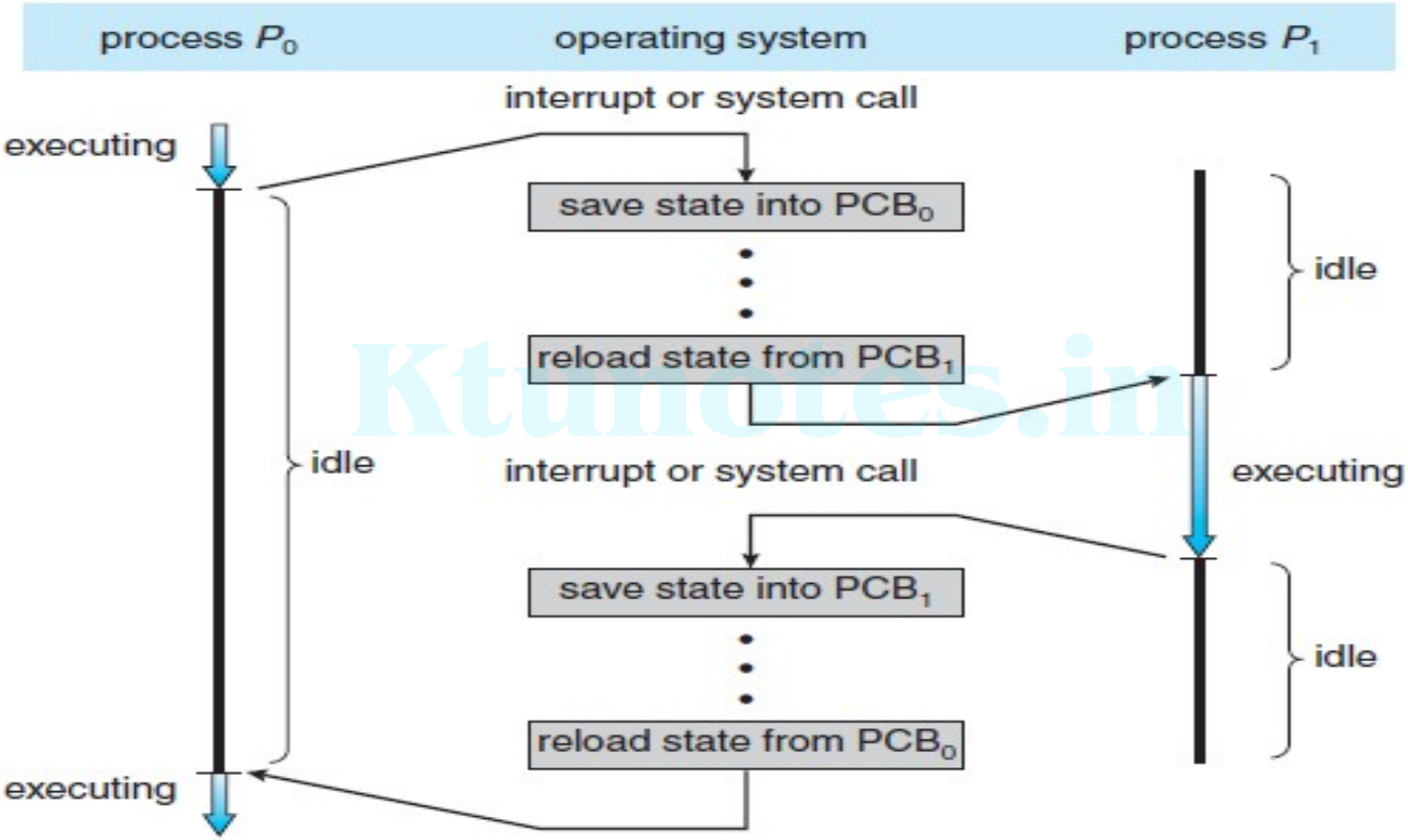


Diagram showing CPU switch from process to process.

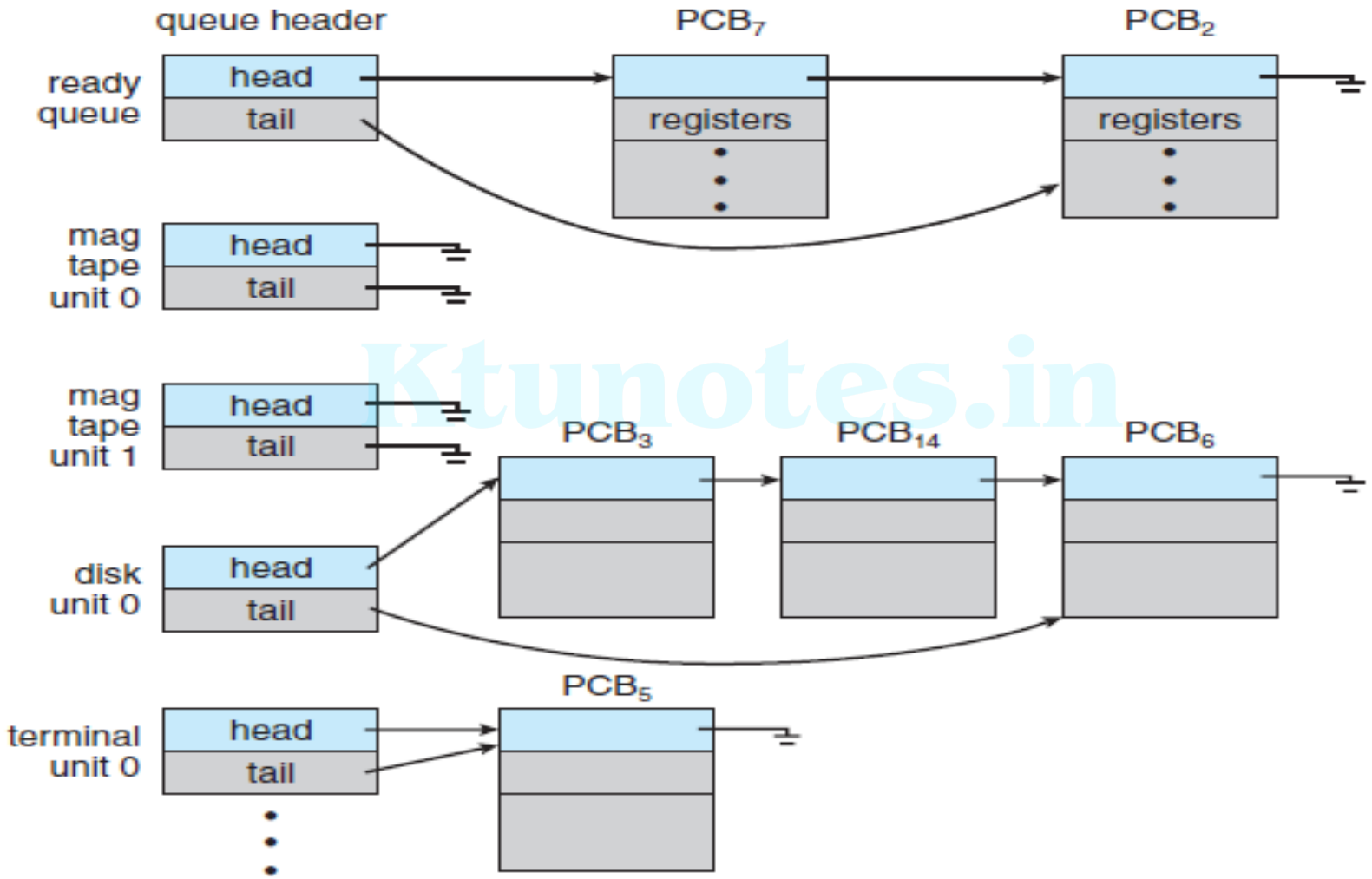
Process Scheduling

- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- 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.
- To meet these objectives, the **process scheduler selects** an available process (possibly from a set of several available processes) for program execution on the CPU.
- For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

Scheduling Queues

- As processes enter the system, they are put into a **job queue,** which 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.
- Each PCB includes a pointer field that points to the next PCB in the ready queue.
- The system also includes other queues. Each device has its own device queue.

Scheduling Queues



The ready queue and various I/O device queues.

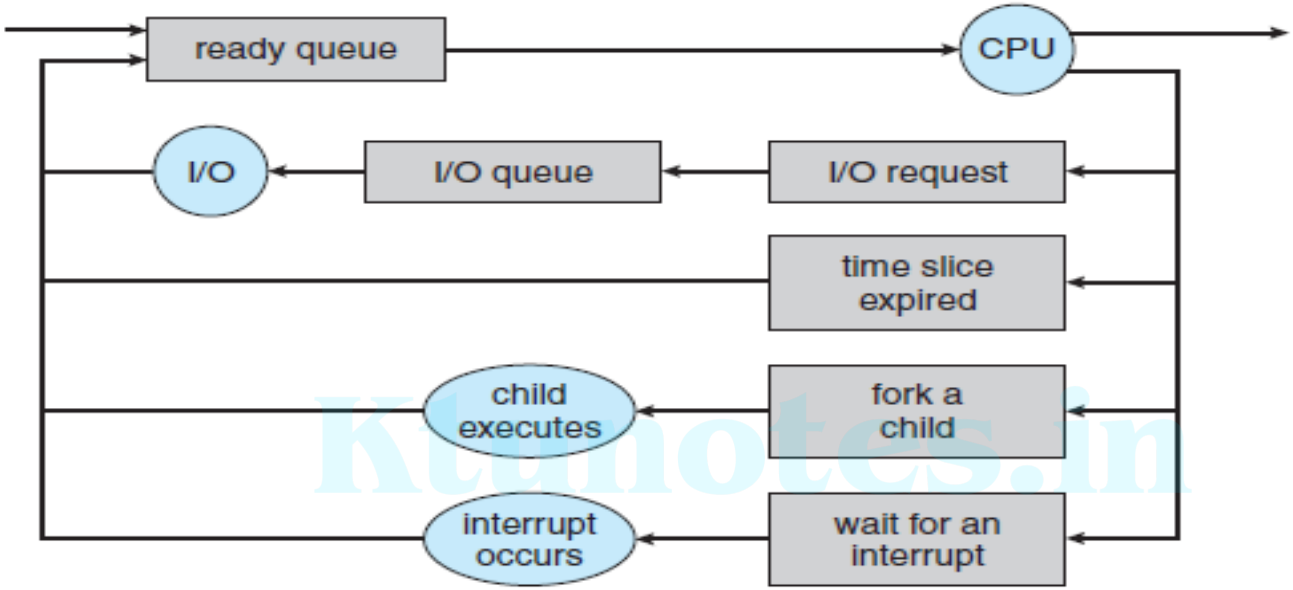
Scheduling Queues

- A new process is initially put in the ready queue.
- It waits there until it is selected for execution, or **dispatched.**
- **Once the process is allocated the CPU** and is executing, one of several events could occur:

1. The process could issue an I/O request and then be placed in an I/O queue.
2. The process could create a new child process and wait for the child's termination.
3. The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

- In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue.
- A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

Scheduling Queues



Queueing-diagram representation of process scheduling.

- 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.

Schedulers

- A process migrates among the various scheduling queues throughout its lifetime.
- 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.**
- In a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution. The **long-term scheduler, or job scheduler, selects processes from this pool and loads them into memory for** execution.
- The **short-term scheduler, or CPU scheduler, selects from among** the processes that are ready to execute and allocates the CPU to one of them.

Schedulers

- The primary distinction between these two schedulers lies in frequency of execution.
- The short-term scheduler must select a new process for the CPU frequently.
- The long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next.

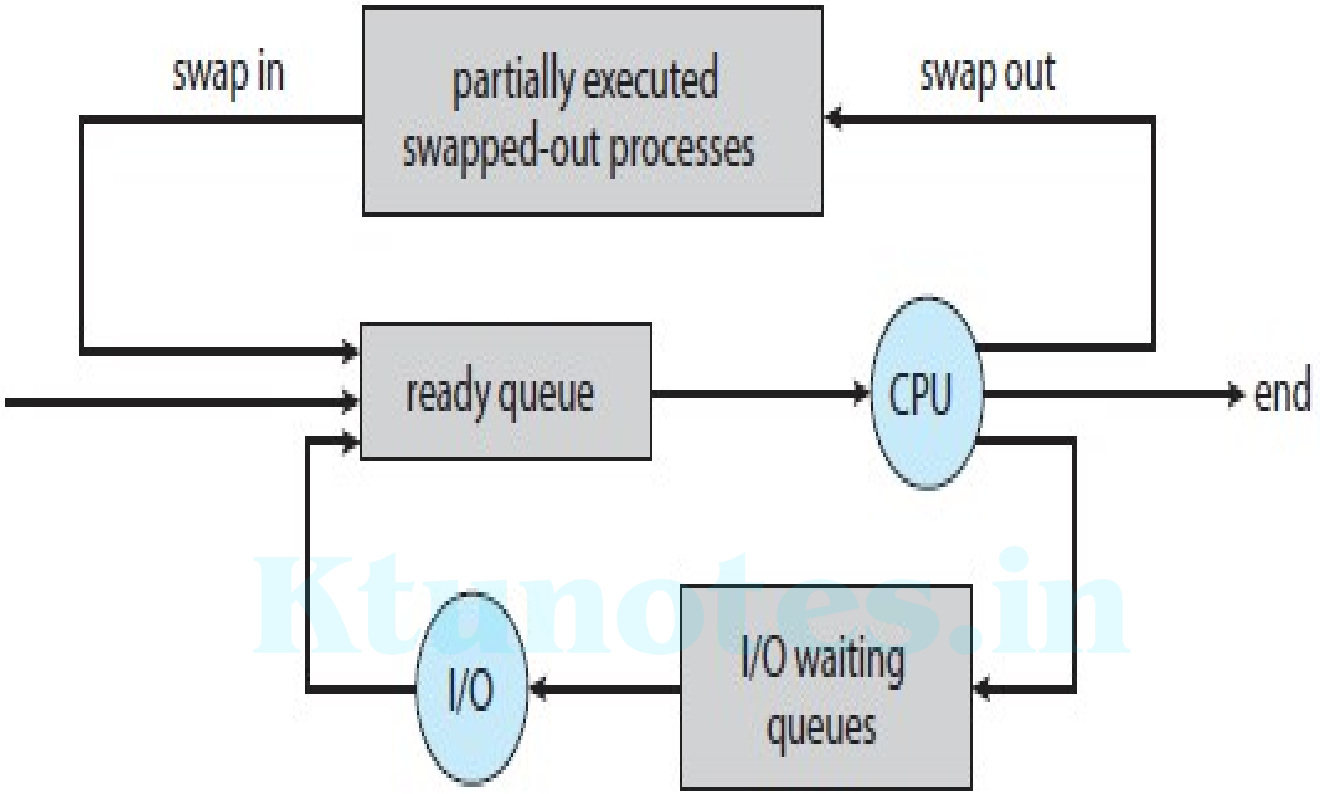
Schedulers

- Processes can be described as either **I/O bound** or **CPU bound**.
  - **I/O-bound process : Spends more of its time doing I/O** than it spends doing computations.
  - **CPU-bound process:** Generates I/O requests infrequently, using more of its time doing computations.
- Long-term scheduler 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 thus have a combination of CPU-bound and I/O-bound processes.

Schedulers

Medium-term scheduler

- **Medium-term scheduler can** remove a process from memory (and from active contention for the CPU) and later the process can be reintroduced into memory, and its execution can be continued where it left off.
- This scheme is called **swapping.**
- Swapping is necessary to improve the process mix(CPU bound I/O bound)



Addition of medium-term scheduling to the queueing diagram.

Context Switch

- When an interrupt occurs, the system needs to save the current context of the process running on the CPU so that it can restore that context when process resumes.
- The context is represented in the PCB of the process.
- It includes the value of the CPU registers, the process state and memory-management information.
- a **context switch** is the process of storing and restoring the state (more specifically, the execution context) of a process so that execution can be resumed from the same point at a later time. This enables multiple processes to share a single CPU and is an essential feature of a multitasking operating system.

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.

- Switching speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied etc.

Context Switch

- Context-switch times are highly dependent on hardware support.
- For instance, some processors (such as the Sun Ultra SPARC) provide multiple sets of registers.
- A context switch here simply requires changing the pointer to the current register set.

Context Switch

Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process.

State Save

State Restore

This task is known as a **context switch**.

Operations on Processes

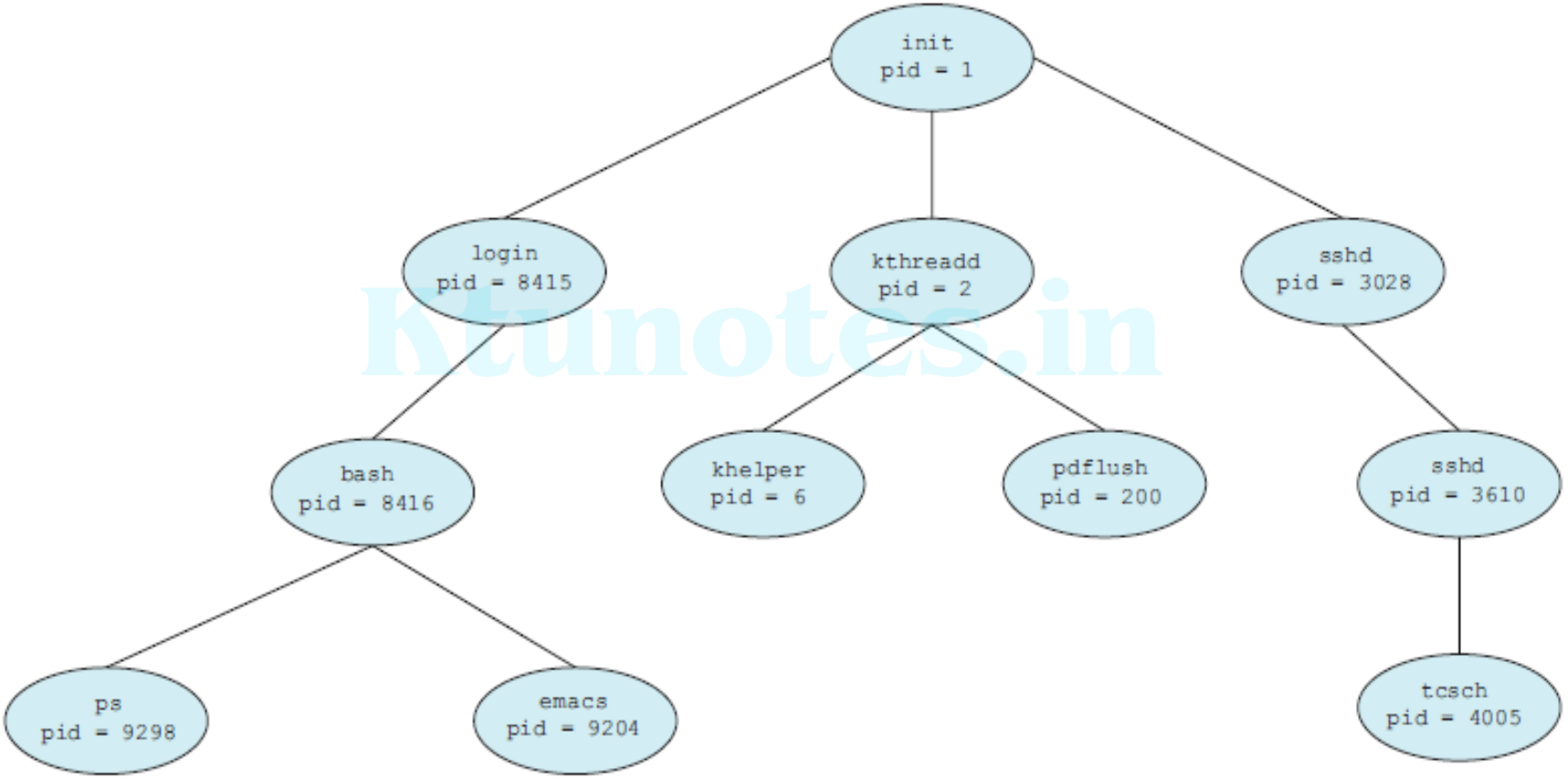
System must provide mechanisms for:

- **process creation**
- **process termination**

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.**
- **Process identifier (or pid)(** which is typically an integer number**)** , is associated with each process.
- 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.

Eg: A typical process tree for the Linux operating system



A tree of processes on a typical Linux system.

Eg: A typical process tree for the Linux operating system

- The init process (which always has a pid of 1) serves as the root parent process for all user processes.

- Once the system has booted, the init process can also create various user processes, such as a web or print server, an ssh server etc.

- In Figure there is three children of init—login,kthreadd and sshd.

- The kthreadd process is responsible for creating additional processes that perform tasks on behalf of the kernel (in this situation, khelper and pdflush).

Eg: A typical process tree for the Linux operating system

- The sshd process is responsible for managing clients that connect to the system by using ssh (which is short for ***secure shell).***

- ***The login process is responsible for managing clients that directly*** log onto the system.

- In this example, a client has logged on and is using the bash shell, which has been assigned pid 8416.

- Using the bash command-line interface, this user has created the process ps as well as the emacs editor.

Resource usage of Child processes

- 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.

- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources

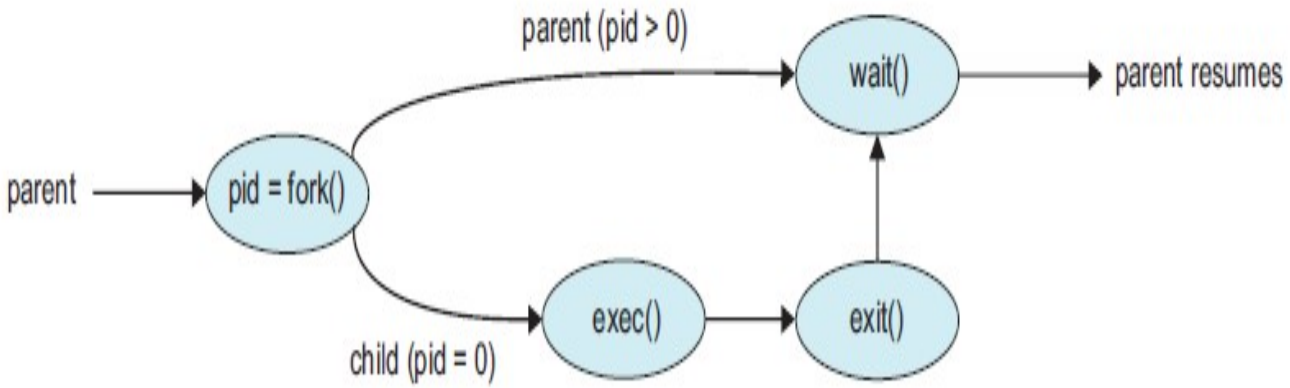
Execution of a child process

- When a process creates a new process, two possibilities for execution exist:
    1. **The parent continues to execute concurrently with its children.**
    2. **The parent waits until some or all of its children have terminated.**
- There are also two address-space possibilities for the new process:
    1. **The child process is a duplicate of the parent process (**it has the same program and data as the parent).
    2. **The child process has a new program loaded into it.**

Process Creation Example

UNIX examples

- **fork()** system call creates new process
- **exec()** system call used after a **fork()** to replace the process' memory space with a new program.
- **wait()** system call issued by a parent process and move itself off the ready queue until the termination of the child



Process creation using the `fork()` system call.

C Program Forking Separate Process

```
#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;
}
```

How many times does the following program print “yes”?

```
#include<stdio.h>
 #include <unistd.h>
main()
 {
fork();
fork();
printf("yes");
 }
```

Ans:4 times

How many times does the following program print “Hello World”?

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

int main() {
int i;
for (i = 0; i < 3; i++)
    fork();
printf("Hello World\n");
return 0;
}
```

Ans:8 times

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 a status value (typically an integer) to its parent process (via the wait() system call).

- All the resources of the process like physical and virtual memory, open files, and I/O buffers are deallocated by the operating system.

Process Termination

- A process can cause the termination of another process by system call (for example, TerminateProcess() in Windows).
- Usually, such a system call can be invoked only by the parent of the process that is to be terminated.
- Parent needs to know the identities of its children if it is to terminate them.
- When one process creates a new process, the identity of the newly created process is passed to the parent.

Process Termination

- A 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.
  - The task assigned to the child is no longer required.
  - The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

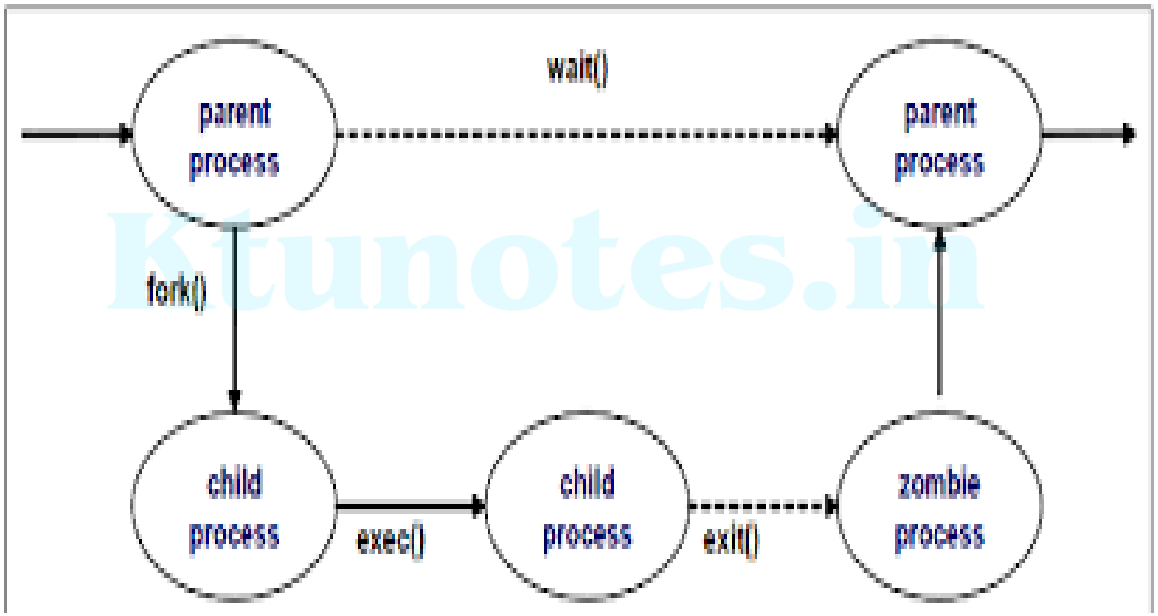
Process Termination- Cascading termination

- Some systems do not allow a child to exist if its parent has terminated.
- 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.

Process Termination- Zombie process

- When a process terminates, its resources are deallocated by the operating system.
- However, its entry in the process table must remain there until the parent calls wait(), because the process table contains the process's exit status.
- A process that has terminated, but whose parent has not yet called wait(), is known as a **zombie process.**
- Once the parent calls wait(), the process identifier of the zombie process and its entry in the process table are released.

Process Termination- Zombie process



Process Termination- Orphans.

- If a parent did not invoke wait() and instead terminated, its child processes left as **orphans.**

- Linux and UNIX handles orphans 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.

Process Termination- Orphans.-Example

```
int main()
{
  int pid = fork();
  if (pid > 0)
     printf("in parent process");
  else if (pid == 0)
    {
        sleep(30);
        printf("in child process");
    }
}
```

Interprocess Communication

Interprocess Communication

- Processes executing concurrently in the operating system may be either independent processes or cooperating processes.

Independent Process

- A process which 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.

Cooperating Process

- Process which *can affect or be affected by the other processes* executing in the system.
- Any process that shares data with other processes is a cooperating process.

Interprocess Communication

Reasons for providing an environment that allows process cooperation:

- **Information sharing.**

  Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.

- **Computation speedup.**

  In order to run a particular task fast, break it into subtasks, each of which will be executing in parallel with the others. Such a speedup can be achieved only if the computer has multiple processing cores.

- **Modularity.**

  In order to construct the system in a modular fashion, dividing the system functions into separate processes or threads,

- **Convenience.**

  Usually individual users may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

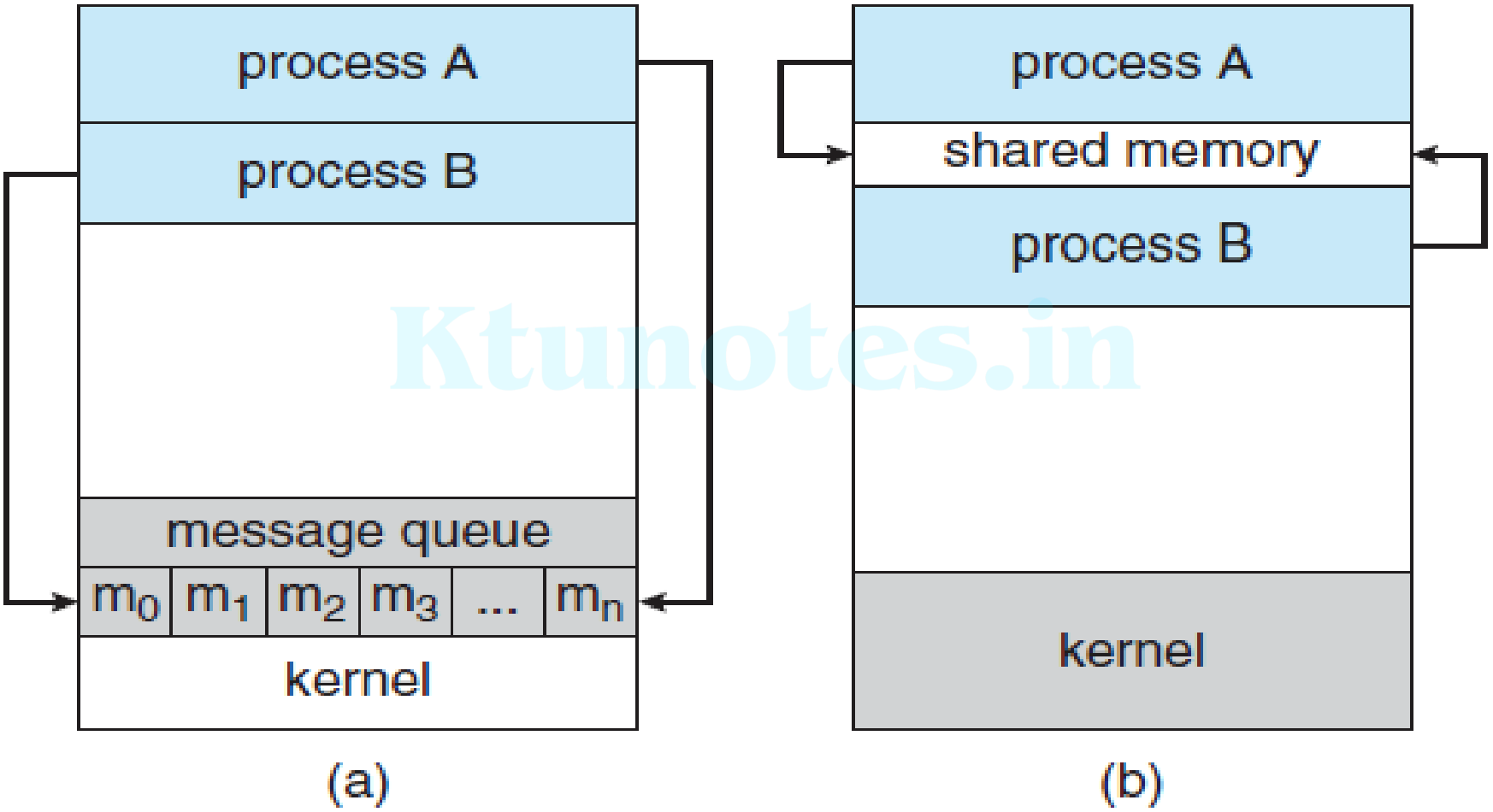
Interprocess Communication

- 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:

  1) shared memory

  2)message passing.
- In the shared-memory model, a region of memory that is shared by cooperating processes is established. 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.

Interprocess Communication



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

Shared-Memory Systems

- Interprocess communication using shared memory requires communicating processes to establish a region of shared memory.
- 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.
- They can then exchange information by reading and writing data in the shared areas.
- The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

Shared Memory Systems-Producer–consumer problem

Producer–consumer problem:

- A producer process produces information that is consumed by a consumer process.
- Eg:A server as a producer and a client as a consumer. A web server produces (that is, provides) HTML files and images, which are consumed (that is, read) by the client web browser
- Producer–consumer problem uses shared memory.
- 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.

Shared-Memory Systems-Producer–consumer problem

Two types of buffers can be used:

- **Unbounded buffer**
- **Bounded buffer**

Unbounded buffer

- No limit on the size of the buffer.
- The consumer may have to wait for new items,but the producer can always produce new items.

Bounded buffer

- A fixed buffer size.
- The consumer must wait if the buffer is empty
- The producer must wait if the buffer is full.

Shared-Memory Systems-Producer–consumer problem using Bounded Buffer

- 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**.

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Shared-Memory Systems-Producer–consumer problem using Bounded Buffer

```
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 producer 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 */
}
```

The consumer process using shared memory.

Message-Passing Systems

- Message passing allows processes to communicate and to synchronize their actions without using shared memory
- Useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.
- Eg: An Internet chat program where chat participants communicate with one another by exchanging messages.
- A message-passing facility provides at least two operations:

  send(message)

  receive(message)

Message-Passing Systems

- 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.

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.

Message-Passing Systems

- 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***; *that is*, *both the sender* process and the receiver process must name the other to communicate.

Message-Passing Systems

Asymmetry in addressing:

- 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.
- Variable id is set to the name of the process with which communication has taken place.

Message-Passing Systems

Indirect communication:

- *The messages are sent to and received from mailboxes, or ports.*
- *A mailbox is 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.

Message-Passing Systems

The **send()** and **receive()** primitives are defined as follows:

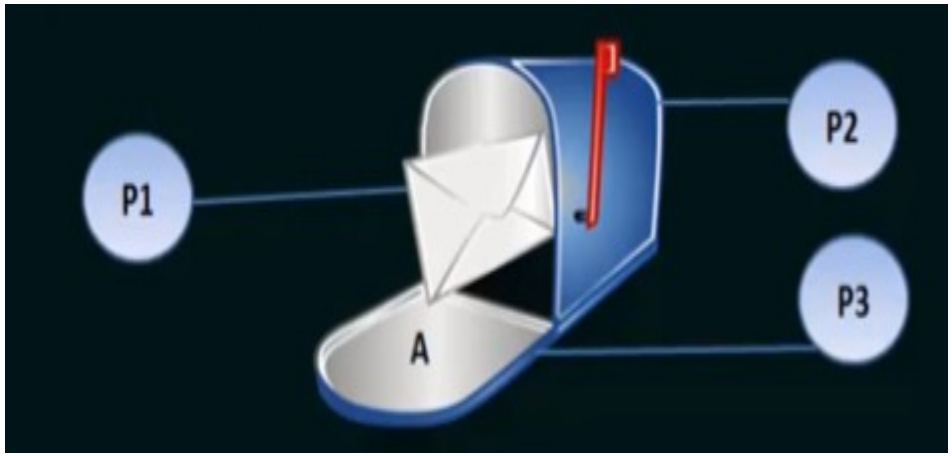
send(A, message)-Send a message to mailbox A.

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.

- Let processes *P1, P2, and P3 all share mailbox A.*
- *Process P1 sends a message to A, while both P2 and P3 execute a receive() from A.*
- **Which process will receive the message sent by *P1 ?***
  - Allow a link to be associated with two processes at most.
  - Allow at most one process at a time to execute a receive() 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).*



Message-Passing Systems

Synchronization

- Communication between processes takes place through calls to send() and receive() primitives.
- 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.
- **Non blocking 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.

Message-Passing Systems

Buffering

- Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue.
- 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.

Message-Passing Systems

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.

CPU SCHEDULING

CPU SCHEDULING

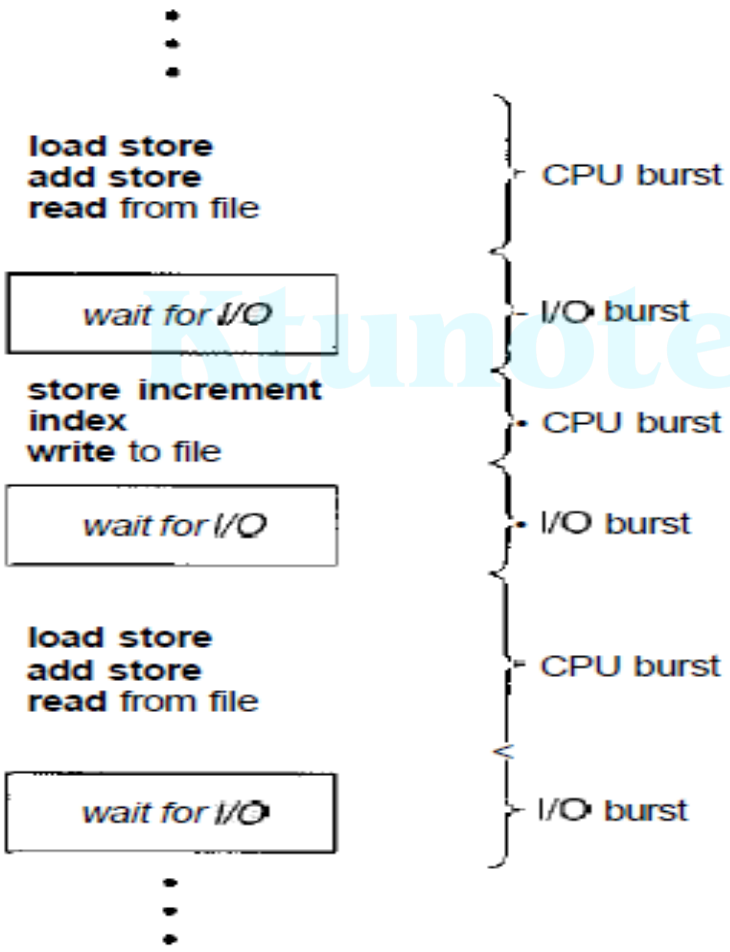
- When a computer is multiprogrammed, it frequently has multiple processes competing for the CPU at the same time. This situation occurs whenever two or more of them are simultaneously in the ready state.
- If only one CPU is available, a choice has to be made which process to run next.
- The part of the operating system that makes the choice is called the scheduler, and the algorithm it uses is called the scheduling algorithm.

CPU SCHEDULING

CPU-I/O Burst Cycle

- The success of CPU scheduling depends on an observed property of processes:
- Process execution consists of a **cycle of CPU execution and I/O wait.**
- **Processes** alternate between these two states.
- Process execution begins with a CPU **burst(**Burst time**).** That is followed by an **I/O burst, which is followed by another CPU burst, then** another I/O burst, and so on.
- Eventually, the final CPU burst ends with a system request to terminate execution.

CPU SCHEDULING…



CPU SCHEDULING….

- A process is executed until it must wait, typically for the completion of some I/O request.
- In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished.
- Multiprogramming, use this time productively. Several processes are kept in memory at one time.
- When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues.
- Every time one process has to wait, another process can take over use of the CPU.

CPU SCHEDULING

CPU Scheduler

- Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.
- The selection process is carried out by the **short-term scheduler (or CPU scheduler).**
- The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

CPU SCHEDULING

CPU-scheduling decisions may take place under the following four circumstances:

1. When a process switches from the **running state to the waiting state** (for example, as the result of an I/O request or an invocation of wait for the termination of one of the child processes)
2. When a process switches from the running state to the ready state *(for* example, when an interrupt occurs).
3. When a process switches from the waiting state to the ready state (for example, at completion of I/O).
4. When a **process terminates.**

Preemptive Scheduling

- A scheduling discipline is preemptive if, once a process has been **given the CPU ,the CPU can taken away** from that process.
- The scheduling which takes place when a process switches from running state to ready state or from waiting state to ready state is called Preemptive Scheduling.

Non Preemptive Scheduling

- A scheduling discipline is non preemptive if, once a process has been given the CPU, **the CPU cannot be taken away** from that process.
- **Under nonpreemptive scheduling, once the CPU has been** allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state

CPU SCHEDULING

Dispatcher

Dispatcher is the module that gives control of the CPU to the process selected by the CPU scheduler.

This function involves the following:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

Scheduling Criteria

Scheduling Criteria

Many criteria have been suggested for comparing CPU scheduling algorithms. The criteria include the following:

- **CPU utilization:** We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
- **Throughput:** Number of processes that are completed per time unit, called *throughput*. *For long processes, this rate may be one* process per hour; for short transactions, it may be 10 processes per second.
- **Turnaround time.** From the point of view of a particular process, theimportant criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

Scheduling Criteria cont...

- **Waiting time. The CPU-scheduling algorithm does not affect the amount** of time during which a process executes or does I/O. It affects only the amount of time that a process spends waiting in the ready queue.Waiting time is the sum of the periods spent waiting in the ready queue.

- **Response time:** Time from the submission of a request until the first response is produced. This measure, called *response time, is the time it takes to start responding*.
- **Fairness:** Degree to which each process is getting an equal chance to execute.

- It is desirable **to maximize CPU utilization and throughput** and to **minimize turnaround time, waiting time, and response time**

Arrival Time: *Time at which the process arrives in the ready queue.*

Completion Time: *Time at which process completes its execution.*

Burst Time: *Time required by a process for CPU execution.*

Turn Around Time: *Time Difference between completion time and arrival time.*

Turn Around Time = Completion Time – Arrival Time

Waiting Time(W.T): *Time Difference between turn around time and burst time.*
Waiting Time = Turn Around Time – Burst Time

CPU SCHEDULING

Arrival Time-AT

Burst Time-BT

Completion Time-CT

Turnaround Time-TAT(TT)

Waiting Time-WT

TT=CT-AT

WT=TT-BT

Scheduling Algorithms

First-Come, First-Served Scheduling(FCFS):

- Simplest CPU-scheduling algorithm is the **first-come, first-served (FCFS) scheduling algorithm.**
- The process that requests the CPU first is allocated the CPU first.
- The implementation of the FCFS policy is easily managed with a FIFO queue.
- When a process enters the ready queue, its PCB is linked onto the tail of the queue.
- When the CPU is free, it is allocated to the process at the head of the queue.
- The running process is then removed from the queue.
- The average waiting time under the FCFS policy, is often quite long.

First-Come, First-Served Scheduling contd…

- The FCFS scheduling algorithm is non preemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.

- The FCFS algorithm troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals.

First-Come, First-Served Scheduling contd...

Convoy effect:

- In a multiprogramming environment, if multiple processes are waiting for the CPU time for execution in “first come first serve” method and a slow process is utilizing the CPU keeping the fast process on wait. It will lead to the convoy effect.
- Consider 100 I/O bound processes and 1 CPU bound job in system.
- These I/O bound processes will quickly pass through the ready queue and suspend themselves (will wait for I/O).
- Now the CPU-bound process (slow) will get and hold the CPU.
- During this time, all the other I/O bound processes will finish their I/O and will move into the ready queue, waiting for the CPU.
- While the I/O bound processes wait in the ready queue, the I/O devices are idle.

First-Come, First-Served Scheduling contd...

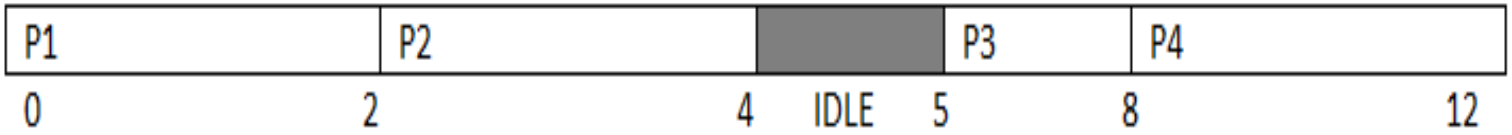
- Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device.
- All the I/O-bound processes, which have short CPU bursts, execute quickly and move back to the I/O queues.
- At this point, the CPU sits idle.
- The CPU-bound process will then move back to the ready queue and be allocated the CPU.
- Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done.
- There is a convoy effect as all the other processes wait for the one big process to get off the CPU.
- A long CPU-bound job may take the CPU and may force shorter (or I/O-bound) jobs to wait prolonged periods.

Example:FCFS

Q:Find the average turn around time and waiting time.

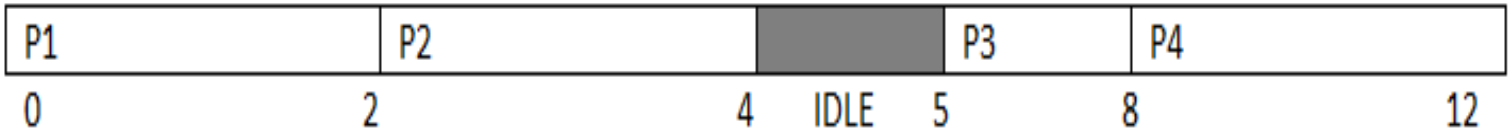
Process No.	Arrival Time	Burst Time
P1	0	2
P2	1	2
P3	5	3
P4	6	4

GANTT CHART:



Process No.	Arrival Time	Burst Time	Completion Time
P1	0	2	2
P2	1	2	4
P3	5	3	8
P4	6	4	12

GANTT CHART:

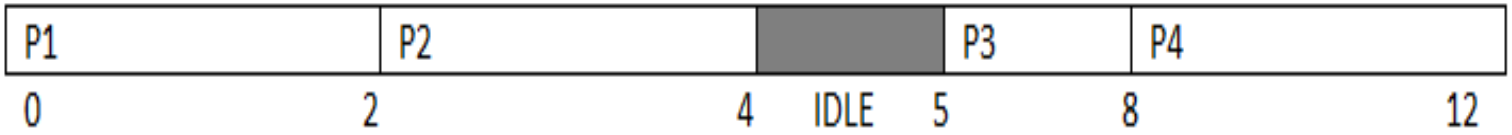


Process No.	Arrival Time	Burst Time	Completion Time	TAT(CT-AT)
P1	0	2	2	2
P2	1	2	4	3
P3	5	3	8	3
P4	6	4	12	6

Process No.	Arrival Time	Burst Time	Completion Time	TAT(CT-AT)	Waiting Time(TAT-BT)
P1	0	2	2	2	0
P2	1	2	4	3	1
P3	5	3	8	3	0
P4	6	4	12	6	2

Process No.	Arrival Time	Burst Time	Completion Time	TAT(CT-AT)	Waiting Time(TAT-BT)	Response Time
P1	0	2	2	2	0	0
P2	1	2	4	3	1	1
P3	5	3	8	3	0	0
P4	6	4	12	6	2	2

GANTT CHART:



First-Come, First-Served Scheduling contd..

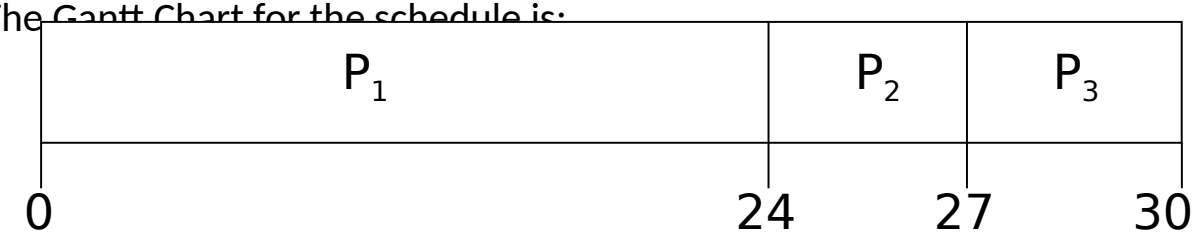
Q)Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order:

  P1 , P2 , P3

- The Gantt Chart for the schedule is:



Waiting time for P1 = 0; P2 = 24; P3 = 27

Average waiting time: (0 + 24 + 27)/3 = 17

First Come First Served

- Suppose that the processes arrive in the order : P_2 , P_3 , P_1 ($P1$:24, $P2$:3, $P3$:3)
- The Gantt chart for the schedule is:

- Waiting time for $P_1 = 6$; $P_2 = 0$, $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$

Q:Find the average turn around time and waiting time

Process No.	Arrival Time	Burst Time
P0	0	10
P1	1	6
P2	3	2
P3	5	4

Process No.	Arrival Time	Burst Time	Completion Time	TAT(CT-AT)	Waiting Time(TAT-BT)	Response Time
P0	0	10	10	10	0	0
P1	1	6	16	15	9	9
P2	3	2	18	15	13	13
P3	5	4	22	17	13	13

GANTT CHART:

Average TAT=(10+15+15+17)/4=14.25

Average waiting time=(0+9+13+13)/4=8.75

Shortest Job First

CONT...

- This algorithm that assumes the run times are known in advance.
- Associate with each process the length of its CPU burst.
- Use these lengths to schedule the process with the shortest time.
- Two schemes:
  - **Non-Preemptive:** once CPU given to the process it cannot be preempted until completes its CPU burst.
  - **Preemptive:** if a new process arrives with CPU burst length less than remaining time of current executing process, preempt the current process. This scheme is know as the Shortest-Remaining-Time-First (SRTF)(SRT).
- **SJF is optimal:** gives minimum average waiting time for a given set of processes.

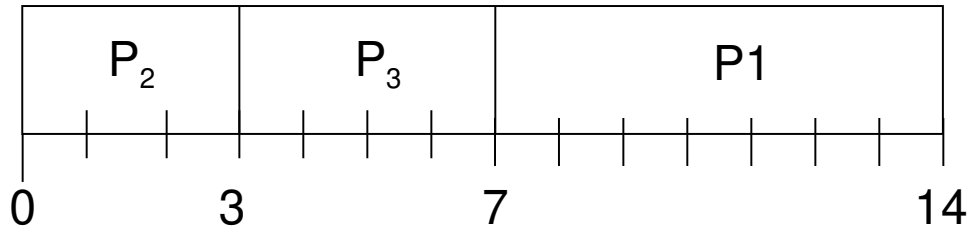
Example #1: Non-Preemptive SJF (simultaneous arrival)

Normal SJF

Process	Burst Time
P_1	7
P_2	3
P_3	4

Arrival Time of all processes is 0

The Gantt Chart for SJF (Normal) is:

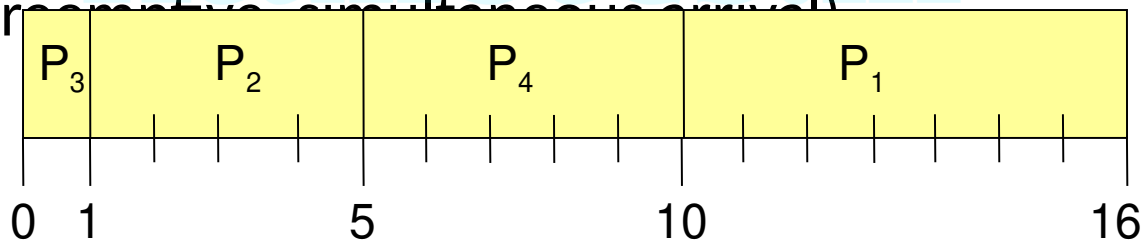


- **Average waiting time = (0 + 3 + 7)/3 = 3.33**

Example #2: Non-Preemptive SJF (simultaneous arrival)

Process	Arrival Time	Burst Time
P_1	0.0	6
P_2	0.0	4
P_3	0.0	1
P_4	0.0	5

- SJF (non-preemptive, simultaneous arrival)

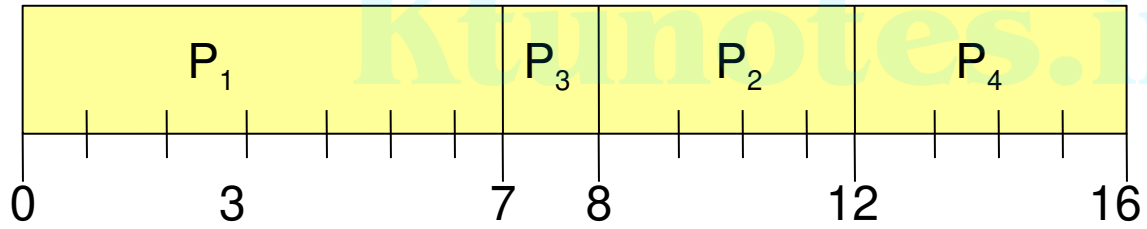


- Average waiting time = (0 + 1 + 5 + 10)/4 = 4
- Average turn-around time = (1 + 5 + 10 + 16)/4 = 8

Example #3: Non-Preemptive SJF (varied arrival times)

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive, varied arrival times)



- Average waiting time

  = ((0 – 0) + (8 – 2) + (7 – 4) + (12 – 5))/4

  = (0 + 6 + 3 + 7)/4 = 4

- Average turn-around time:

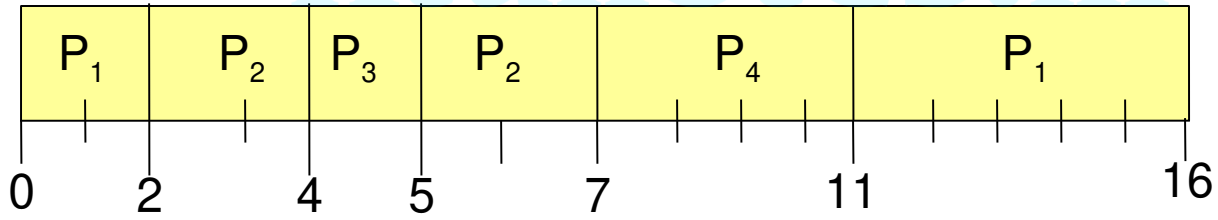
  = ((7 – 0) + (12 – 2) + (8 - 4) + (16 – 5))/4

  = (7 + 10 + 4 + 11)/4 = 8

Example #4: Preemptive SJF (Shortest-remaining-time-first)

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive, varied arrival times)



- Average waiting time

  = ([(0 – 0) + (11 - 2)] + [(2 – 2) + (5 – 4)] + (4 - 4) + (7 – 5))/4

  = 9 + 1 + 0 + 2)/4

  = 3

- Average turn-around time = [(16 -0)+ (7-2) +(5 -4)+ (11-5))/4 = 7

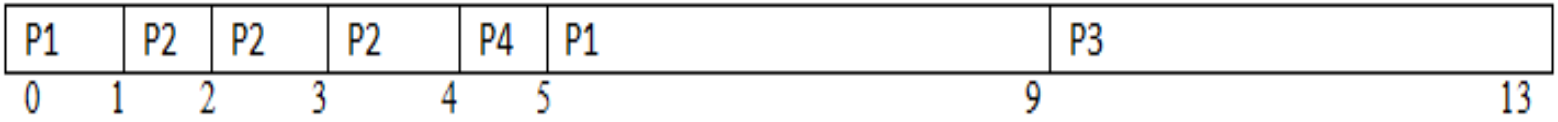
Shortest-Remaining-Time-First (SRTF)- SJF PREEMPTIVE

Q:Find the average turn around time and waiting time

Process No.	Arrival Time	Burst Time
P1	0	5
P2	1	3
P3	2	4
P4	4	1

Process No.	Arrival Time	Burst Time	Completion Time	TAT(CT-AT)	Waiting Time(TAT-BT)	Response Time
P1	0	5	9	9	4	0
P2	1	3	4	3	0	0
P3	2	4	13	11	7	7
P4	4	1	5	1	0	0

GANTT CHART:



Priority Scheduling

CONT...

- A priority number (integer) is associated with each process.
- Lager the CPU burst ,lower the priority.
- The CPU is allocated to the process with the highest priority (smallest integer highest priority)
- **Equal priority processes are scheduled in FCFS**
- Priorities can be defined either internally or externally.
- Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, memory requirements, number of open files , ratio of average I/O burst to average CPU burst have been used in computing priorities.
- Externally priority is set by criteria that is external to the O.S.; such as importance of the process.

Priority Scheduling

- Priority can be either preemptive or nonpreemptive.
- A preemptive priority will preempt the CPU if the newly arrived process priority is higher than the priority of the currently running process.
- A non preemptive priority will simply put the new highest priority process at the head of the ready queue.
- Problem :***Starvation*** – *low priority processes may never get the chance to* execute.

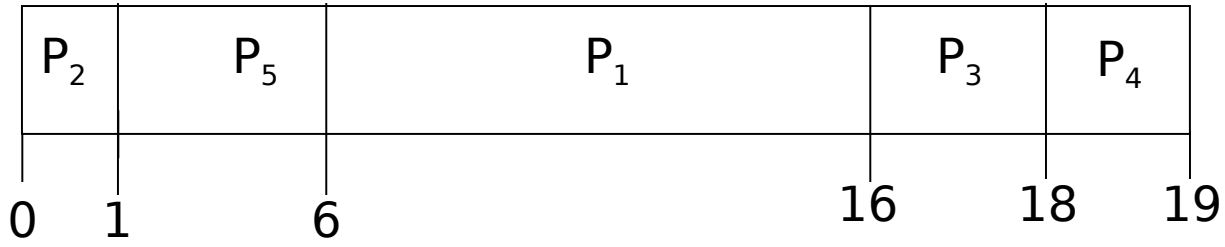
Solution : ***Aging*** – *is a technique of gradually, increasing the priority* of the processes that wait in the system for a long time. so eventually the process will become the highest priority and will gain the CPU (i.e., the more time is spending a process in ready queue waiting, its priority becomes higher and higher).

Priority Scheduling Example:

Non preemptive

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

- **Gantt Chart**



- **Average waiting time = (6 + 0 + 16 + 18 + 1)/5 = 8.2**

PRIORITY SCHEDULING –NON PREEMPTIVE

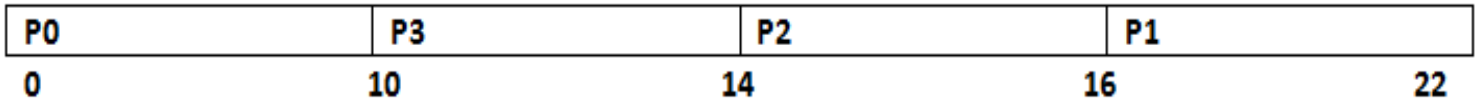
Q:Find the average turn around time and waiting time

Process No.	Arrival Time	Burst Time	Priority
P0	0	10	5
P1	1	6	4
P2	3	2	2
P3	5	4	0

Process No.	Arrival Time	Burst Time	Priority	Completion Time	TAT(CT-AT)	Waiting Time(TAT-BT)	Response Time
P0	0	10	5	10	10	0	0
P1	1	6	4	22	21	15	15
P2	3	2	2	16	13	11	11
P3	5	4	0	14	9	5	5

I

Gantt Chart:



Priority Scheduling Example: Preemptive

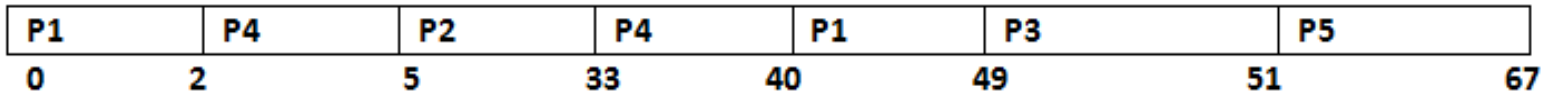
Q:Find the average turn around time and waiting time

Process ID	Arrival Time	Burst Time	Priority
P1	0	11	2
P2	5	28	0
P3	12	2	3
P4	2	10	1
P5	9	16	4

Priority Scheduling Example: Preemptive(cont..)

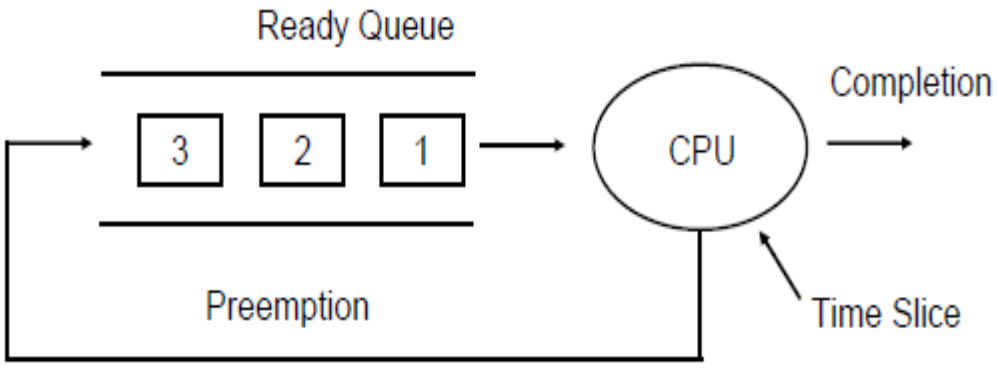
Process No.	Arrival Time	Burst Time	Priority	Completion Time	TAT(CT-AT)	Waiting Time(TAT-BT)	Response Time
P1	0	11/9	2	49	49	38	0
P2	5	28	0	33	28	0	0
P3	12	2	3	51	39	37	37
P4	2	10/7	1	40	38	28	0
P5	9	16	4	67	58	42	42

Gantt Chart:



Round Robin Scheduling

- The Round Robin is designed for time sharing systems.
- The RR is similar to FCFS, but preemption is added to switch between processes.
- Each process gets a small unit of CPU time (*time quantum or time slice*), *usually 10-100 milliseconds. After this time has elapsed*, the process is preempted and added to the end of the ready queue.
- That is, after the time slice is expired an interrupt will occur and a context switch.



Round Robin Scheduling

- If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units at once. No process waits more than (n-1)q time units.
- The performance of RR depends on the size of the time slice q :
  - If q very large (infinite) ⇒ FIFO
  - If q very small ⇒ RR is called processor sharing, and context switch increases. So, q must be large with respect to context switch time, otherwise overhead is too high.

Round Robin Scheduling(example)

Q:Find the average turn around time and waiting time

Process No.	Arrival Time	Burst Time
P1	0	5
P2	1	4
P3	2	2
P4	4	1

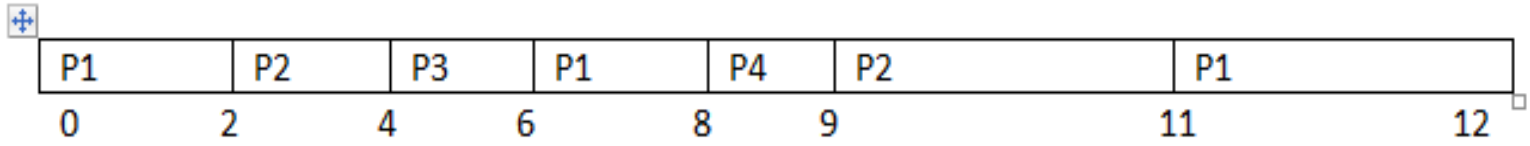
Time Quantum-2

Process No.	Arrival Time	Burst Time	Completion Time	TAT(CT-AT)	Waiting Time(TAT-BT)	Response Time
P1	0	5	12	12	7	0
P2	1	4	11	10	6	1
P3	2	2	6	4	2	2
P4	4	1	9	5	4	4

Ready Queue:

P1	P2	P3	P1	P4	P2	P1	

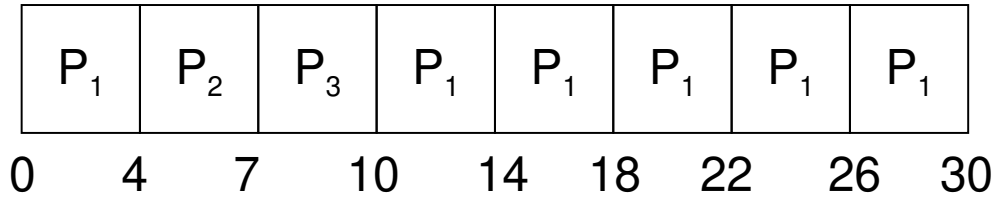
GANTT CHART:



Round Robin Scheduling: Example time Quantum = 4 milliseconds

Process	Burst Time
P1	24
P2	3
P3	3

- **Quantum time = 4 milliseconds**
- **The Gantt chart is:**

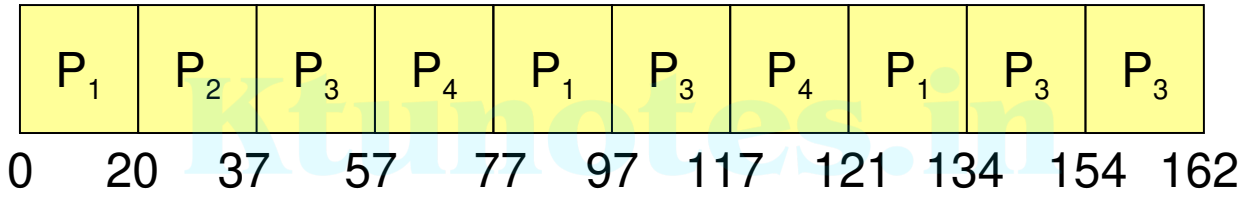


- **Average waiting time = {[0+(10-4)] +4+7}/3 = 5.6**

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better *response time*
- Average waiting time

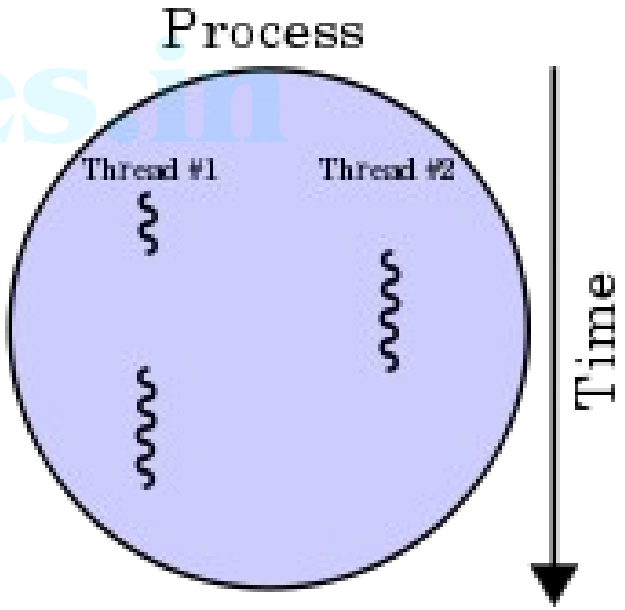
= ([(0 – 0) + (77 - 20) + (121 – 97)] + (20 – 0) + [(37 – 0) + (97 - 57) + (134 – 117)] + [(57 – 0) + (117 – 77)]) / 4
= (0 + 57 + 24) + 20 + (37 + 40 + 17) + (57 + 40)) / 4
= (81 + 20 + 94 + 97)/4
= 292 / 4 = 73

- Average turn-around time = 134 + 37 + 162 + 121) / 4 = 113.5

THREAD

- A thread is contained inside a process and different threads in the same process share some resources (most commonly memory), while different processes do not.

Process and threads



Thread Overview

- Threads are mechanisms that permit an application to perform multiple tasks concurrently.

- Thread sometimes called a lightweight process is a basic unit of CPU utilization.
- Thread comprises:
  - Thread ID
  - Program counter
  - Register set
  - Stack

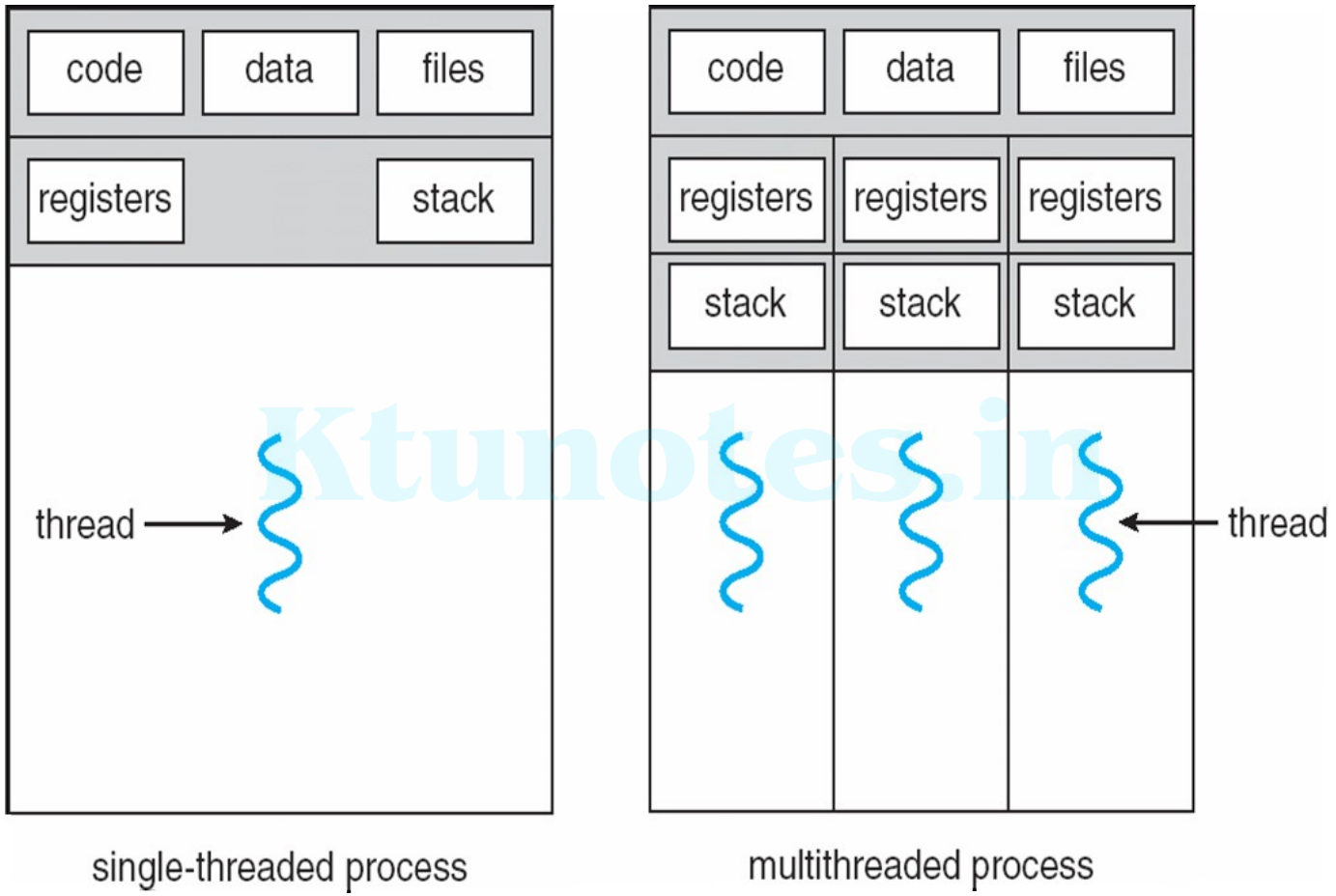
Threads

- Most software applications that run on modern computers are multithreaded.
- An application typically is implemented as a separate process with several threads of control.
- **Eg:** A web browser might have one thread display images or text while another thread retrieves data from the network,.
- A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

Threads

- A single program can contain multiple threads
  - Threads share with other threads belonging to the same process
    - Code, data, open files…….

Single and Multithreaded Processes



heavyweight process

lightweight process

Advantages of Thread

- Threads minimize the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

Difference between Process and Thread

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.