Section II
Process

- **Section-II *(Weightage – 70%)***
  Process - Introduction, Threads, CPU Scheduling algorithms, Inter-process communication, Critical section problem, Semaphores, Classical process coordination problem.
- **Deadlock -**Definition, Necessary and sufficient conditions for Deadlock, Deadlock Prevention, Deadlock Avoidance: Banker's algorithm, Deadlock detection and Recovery.
- **Memory Management –** Concept of Fragmentation**,** Swapping, Paging, Segmentation.
- **Virtual memory**-Demand Paging, Page replacement algorithm, Thrashing.

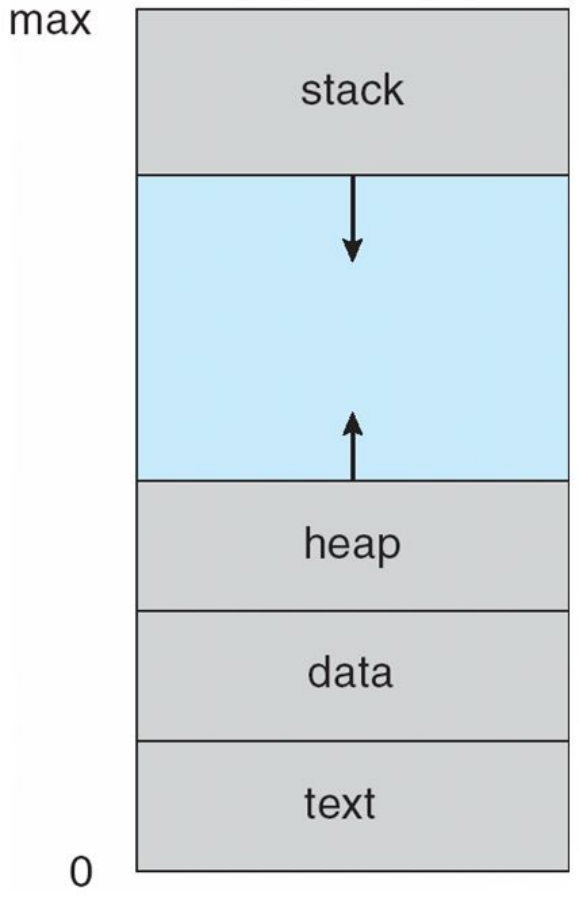
Background

- Early computer systems allowed only one program to be executed at a time.
- Modern computer systems allow to execute multiple programs at a time.
- In early days executing programs in -
  - Batch system / Multiprogramming – **jobs**
  - Time-shared systems – **user programs** or **tasks**
  - **Process is a program in execution**
- Program is ***passive*** entity stored on disk
- process is ***active***
  - Program becomes process when executable file loaded into memory
- One program can be several processes
  - Same user can work on multiple copies of browser.

Process Concept

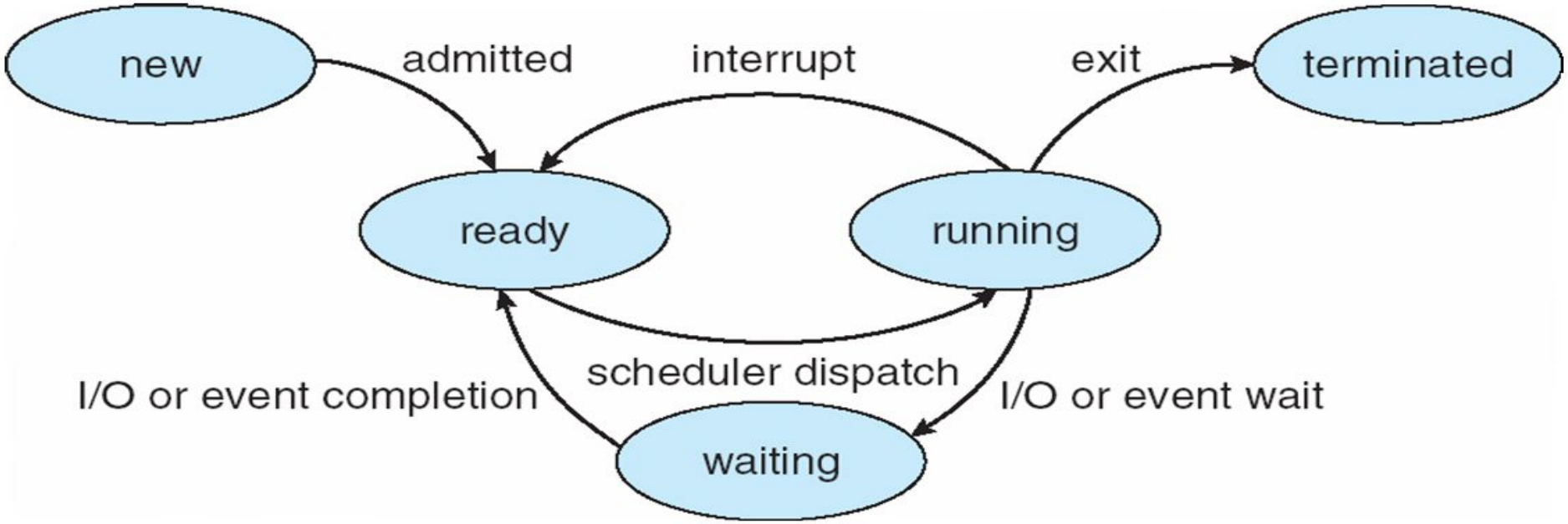
- **Process** – a program in execution;
  - It is a ready to execute program in ram.
  - Process is an active entity
  - It is more than a program code, include activities of the running process.
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

Process in Memory



Process State

- As a process executes, it changes **state**
- **Process activity is defined by its current activity**
  - **new**: The process is about to created, but still on hdd.
  - **ready**: all resources(memory) are allocated and he process is waiting to be assigned to a processor.
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur (e.g. completion of i/o operation, reception of a signal)
  - **terminated**: The process has finished execution

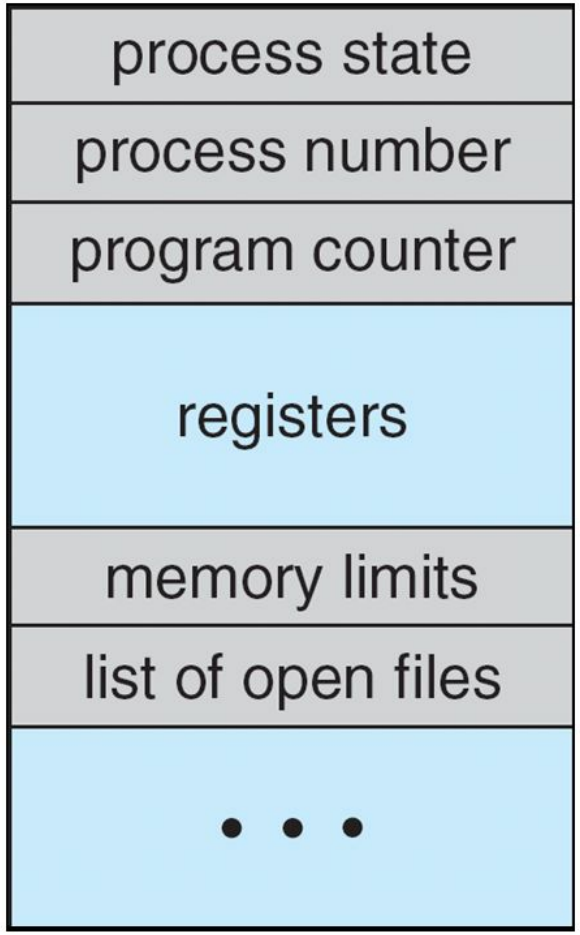


Process Control Block (PCB / TCB)

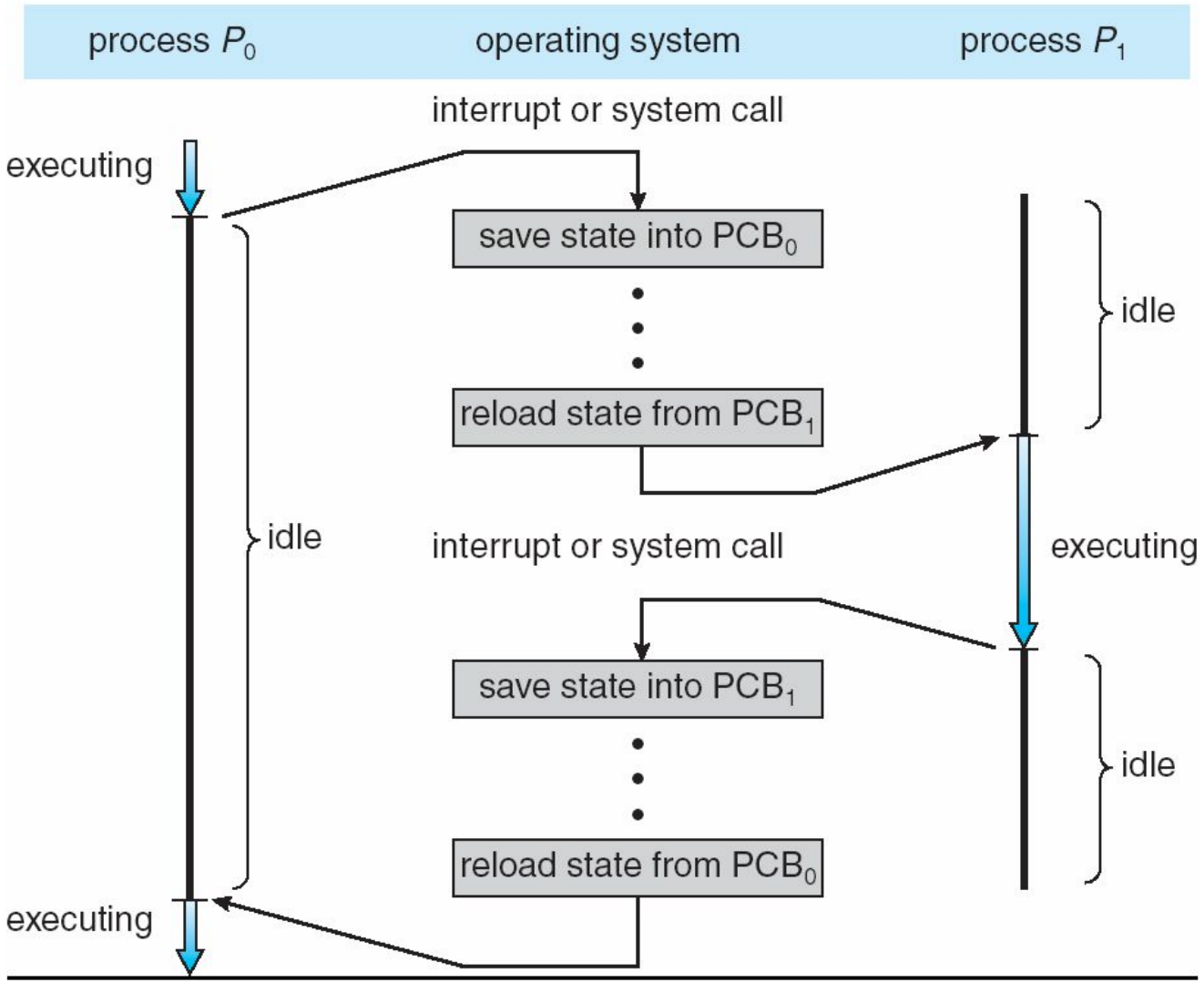
In OS information of each process is represented by PCB

(also called **task control block**)

- Process number - PID
- Process state – running, waiting, etc
- Program counter – Address of next instruction to be executed
- CPU registers – number & type of registres vary from architecture to architecture. PCB maintains contents of all process-centric registers
- CPU scheduling information- priorities, scheduling parameters
- Memory-management information – memory allocated to the process
- Accounting information – CPU time, CPU Cycles, time limits etc.
- I/O status information – list of I/O devices allocated to process, list of open files etc.



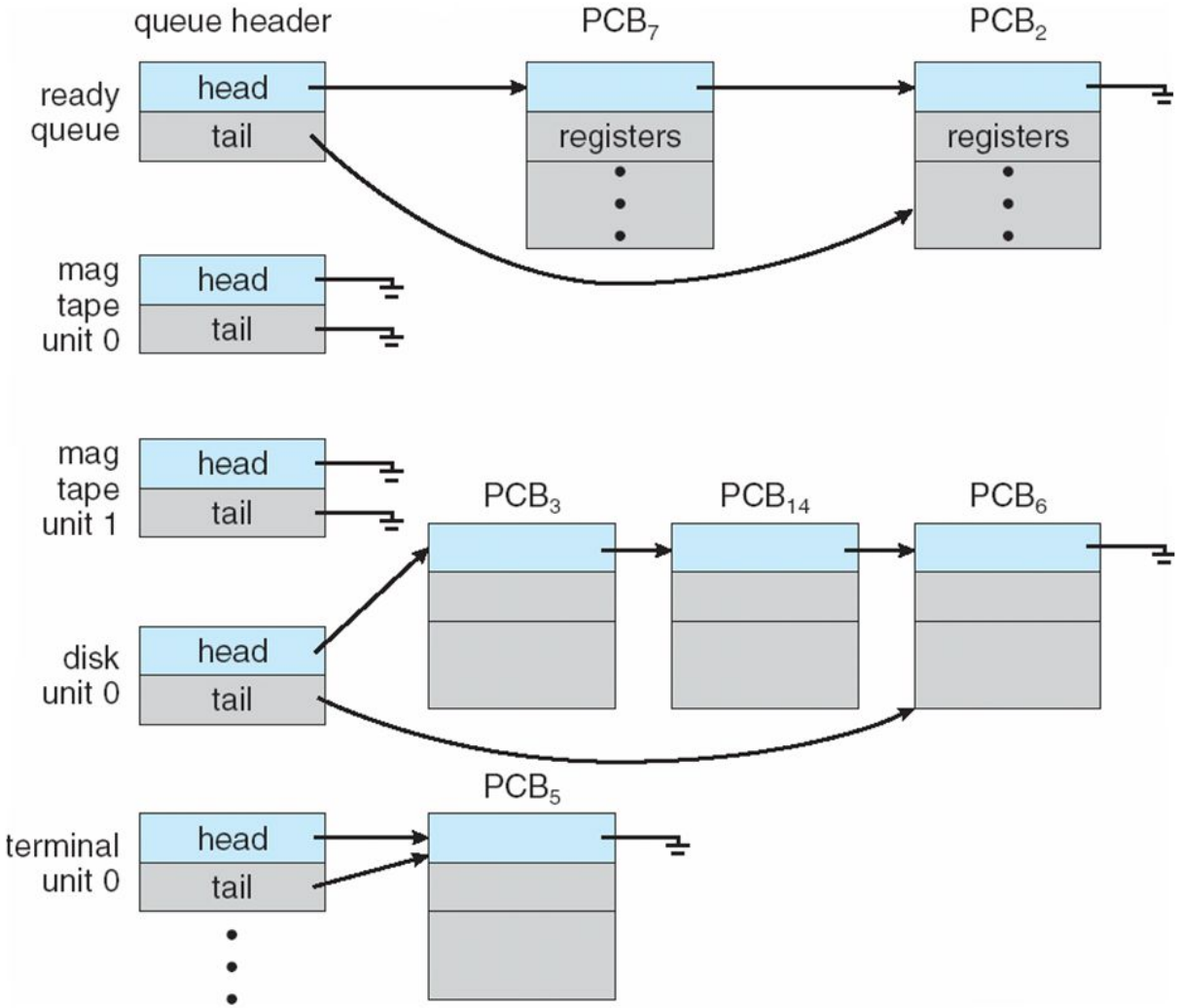
CPU Switch From Process to Process Context Switch



Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing are the objectives of **multiprogramming and time sharing systems.**
- **Process scheduler** selects among available processes for next execution on CPU
- **Scheduling queues** of processes
  - **Job queue** – when process enters the system they are placed in Job queue. Job queue consist of set of all processes in the system
  - **Ready queue** – when all resources(memory) available process is placed in ready queue. Ready queue consist of set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – when executing process issue i/o request, it is placed in device queue till i/o completion. Device queue consist of set of processes waiting for an I/O device
  - Processes continue this cycle till terminates and resources are deallocated.

Ready Queue And Various I/O Device Queues



(Figure 3.6)

- 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. When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/0 request.
- Suppose the process makes an I/O request to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/0 request of some other process.
- The process therefore may have to wait for the disk.
- The list of processes waiting for a particular I/0 device is called a device queue.
- Each device has its own device queue (Figure 3.6).
- A common representation of process scheduling is a queueing diagram, such as that in Figure 3.7.
- 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.

Representation of Process Scheduling

- **Queueing diagram** represents queues, resources, flows.

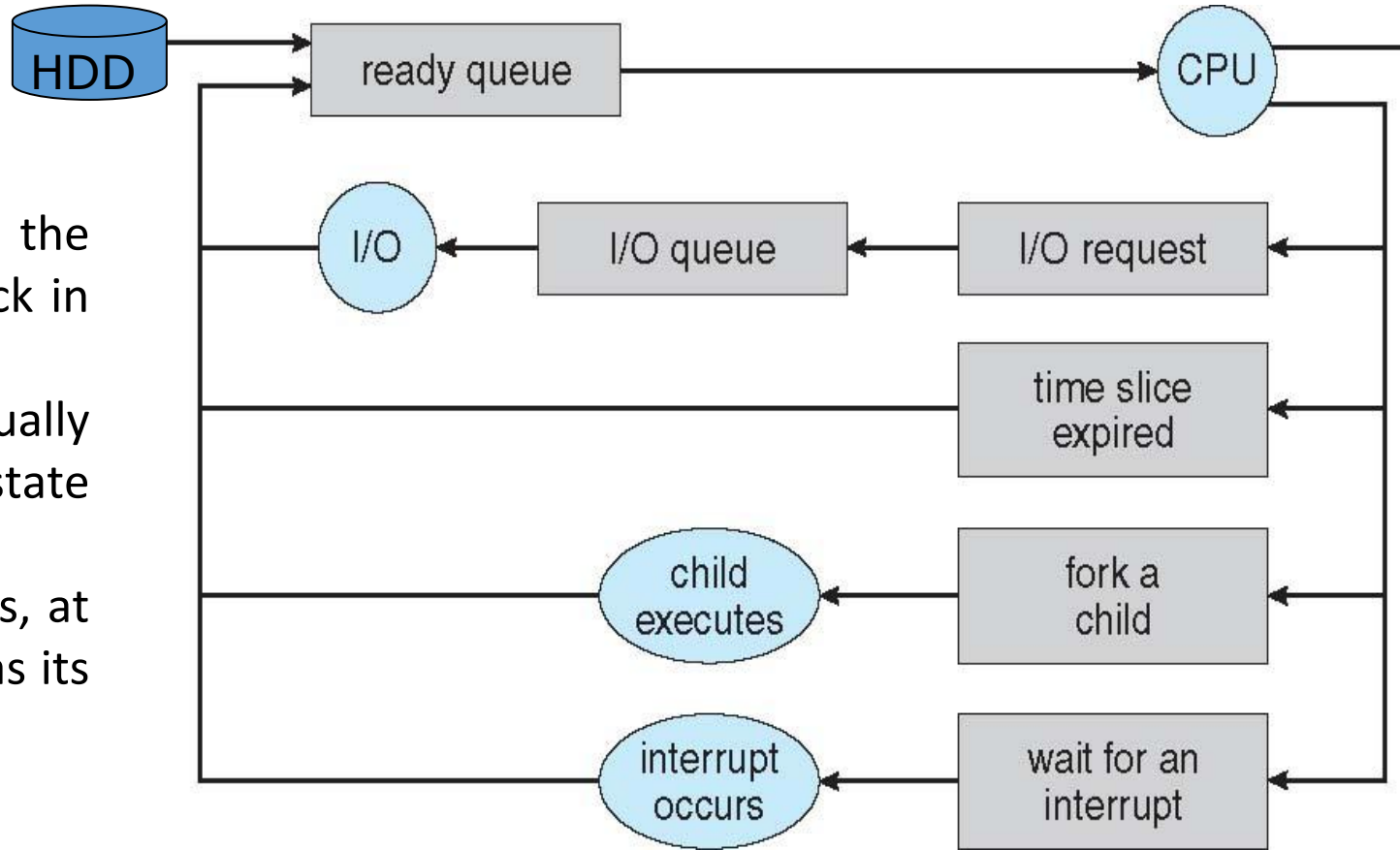
A new process is initially put in the ready queue. It waits there until it is selected for execution, or is dispatched.

Once the process is allocated the CPU and is executing, one of several events could occur:

The process could issue an I/0 request and then be placed in an I/0 queue.

The process could create a new subprocess and wait for the subprocess's termination.

Queueing diagram



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.

Schedulers

- A process migrates among the various scheduling queues throughout its lifetime.
- The operating system must select processes from these queues in some fashion.
- The selection process is carried out by the appropriate scheduler.
- 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. They differ in frequency of execution.
- The short-term scheduler must select a new process for the CPU frequently.
- A process may execute for only a few milliseconds before waiting for an I/0 request. Often, the short-term scheduler executes at least once every 100 milliseconds.
- Because of the short time between executions, the short-term scheduler must be fast.
- If it takes 10 milliseconds to decide to execute a process for 100 milliseconds, then 10 I (100 + 10) = 9 percent of the CPU is being used (wasted) simply for scheduling the work.
- The long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next. 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.

Schedulers

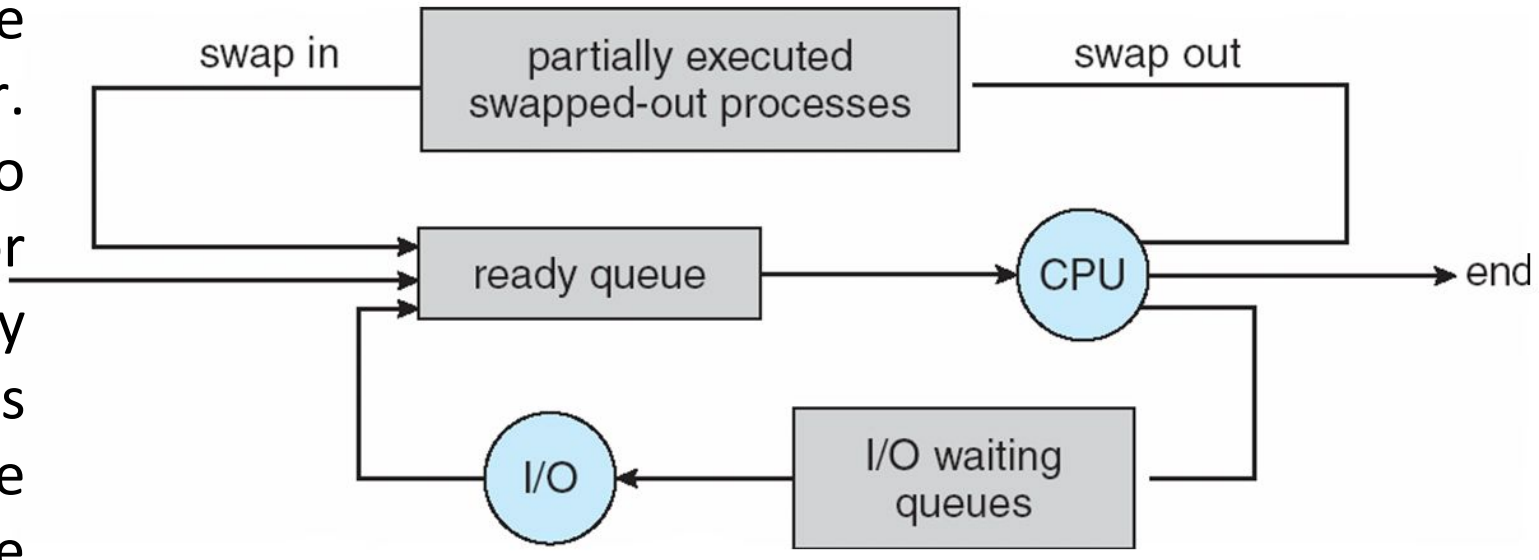
- Thus, the long-term scheduler may need to be invoked only when a process leaves the system.
- Because of the longer interval between executions, the long-term scheduler can afford to take more time to decide which process should be selected for execution.The long-term scheduler makes a careful selection.
- In general, most processes can be described as either I/O bound or CPU bound.
- An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations.
- A CPU-bound process, in contrast, generates I/0 requests infrequently, using more of its time doing computations.
- The 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/0 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.
- On some systems, the long-term scheduler may be absent or minimal.
- For example, time-sharing systems such as UNIX and Microsoft Windows systems often have no long-term scheduler but simply put every new process in memory for the short-term scheduler. The stability of these systems depends either on a physical limitation (such as the number of available terminals) or on the self-adjusting nature of human users. If performance declines to unacceptable levels on a multiuser system, some users will simply quit.

Addition of Medium Term Scheduling

- Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. This medium-term scheduler is diagrammed in Figure 3.8. The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove processes from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramrning. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called swapping.

- The process is swapped out, and is later swapped in, by the medium-term scheduler. Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be

Figure 3.8.



Context Switch

- Interrupts cause the operating system to change a CPU from its current task and to run a kernel routine. Such operations happen frequently on general-purpose systems.
- 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 its processing is done, essentially suspending the process and then resuming it.
- 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. Generically, we perform a state save of the current state of the CPU, be it in kernel or user mode, and then a to state restore to resume operations.
- Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as Context switch.

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.
- A context switch here simply requires changing the pointer to the current register set.
- Of course, if there are more active processes than there are register sets, the system resorts to copying register data to and from memory, as before.
- Also, the more complex the operating system, the more work must be done during a context switch.

Operations on Processes

- The processes in most systems can execute concurrently, and they may
- be created and deleted dynamically.

- System must provide mechanisms for:
  - process creation,
  - process termination,
  - Process communication

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, 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 and the Windows family of operating systems) identify processes according to a unique process identifier (or pid), which is typically an integer number.
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

A tree of processes on Solaris system

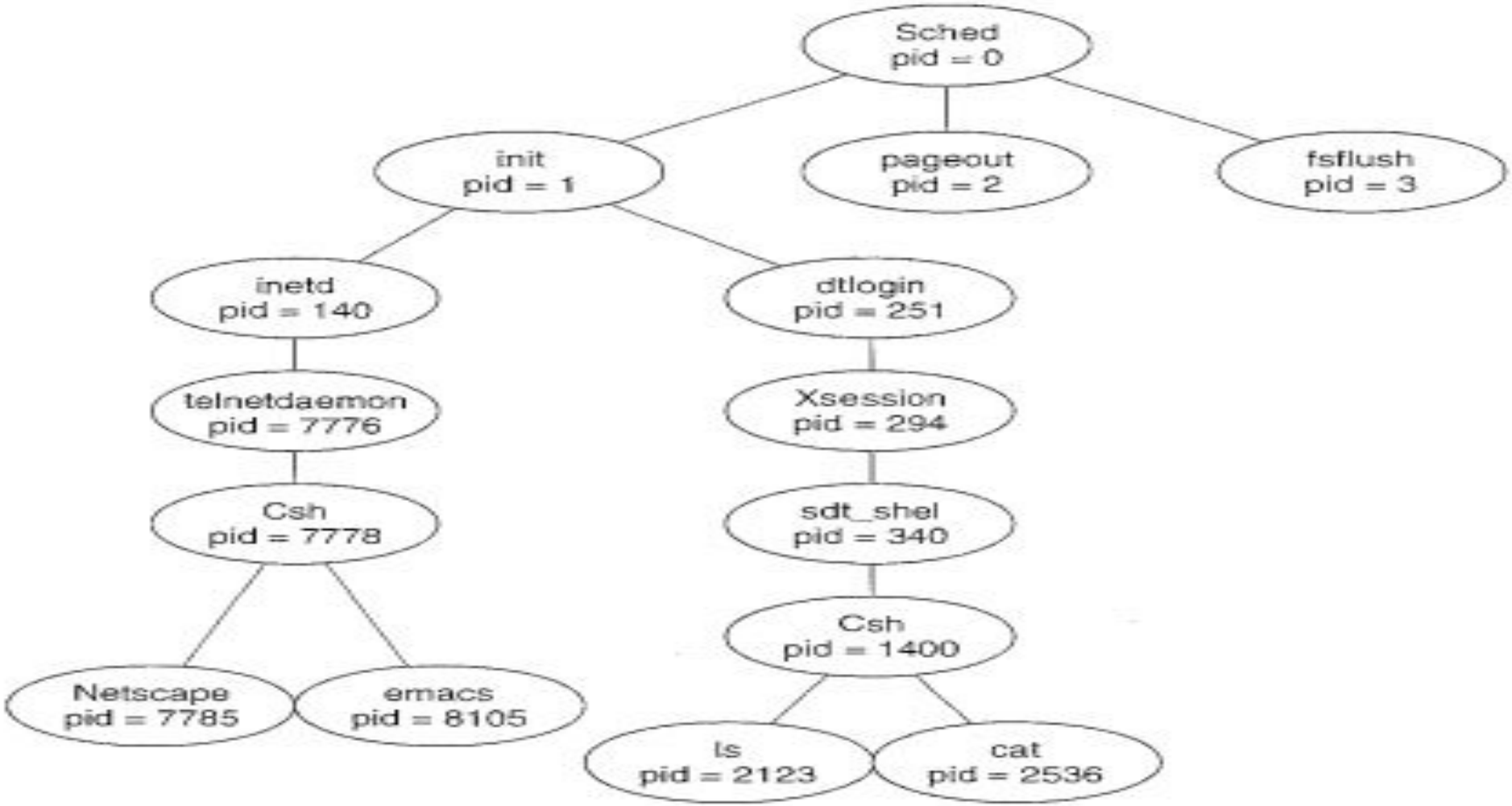
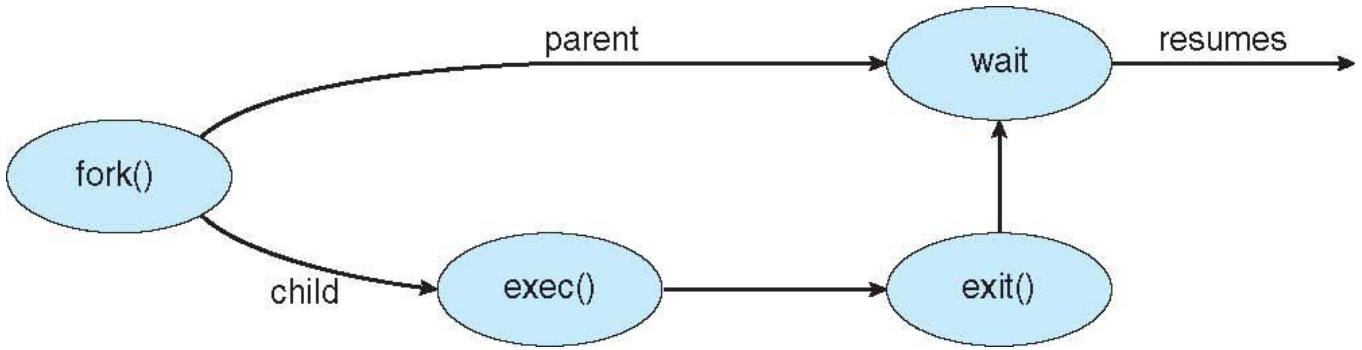


Figure 3.9 A tree of processes on a typical Solaris system.

- Figure 3.9 illustrates a typical process tree for the Solaris operating system, showing the name of each process and its pid.
- In Solaris, the process at the top of the tree is the sched process, with pid of 0.
- The sched process creates several children processes-including pageout and fsflush. These processes are responsible for managing memory and file systems.
- The sched process also creates the init process, which serves as the root parent process for all user processes.
- In Figure 3.9, we see two children of init are inetd and dtlogin.
- inetd is responsible for networking services such as telnet and ftp;
- dtlogin is the process representing a user login screen.
- When a user logs in, dtlogin creates an X-windows session (Xsession), which in turns creates the sdt_shel process.
- Below sdt_shel, a user's command-line shell-the C-shell or csh-is created.
- In this commandline interface, the user can then invoke various child processes, such as the ls and cat commands.

Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- 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



Abort()

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;
}
```

Process Termination

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

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.

Interprocess Communication

- There are several 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. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others.
- Modularity. We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
- Convenience. Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.
- 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 and (2) message passing.

Interprocess Communication

- 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.
- Message passing is useful for exchanging smaller amont of data, because no conflicts need be avoided.
- Message passing is also easier to implement than is shared memory for intercomputer communication.
- Shared memory allows maximum speed and convenience of communication.
- Shared memory is faster than message passing, as message passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel irttervention.
- In contrast, 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.

Communications Models

- From figure 3.13 b) Shared memory Process A and process B are executing simultaneously and they share some resources. Process A generates data based on computations in the code. Process A stores this data in shared memory. When process B needs to use the shared data, it will check in the shared memory segment and use the data that processA placed there. Processes can use shared memory for extracting information as a record from another process as well as for delivering any specific information to other processes.

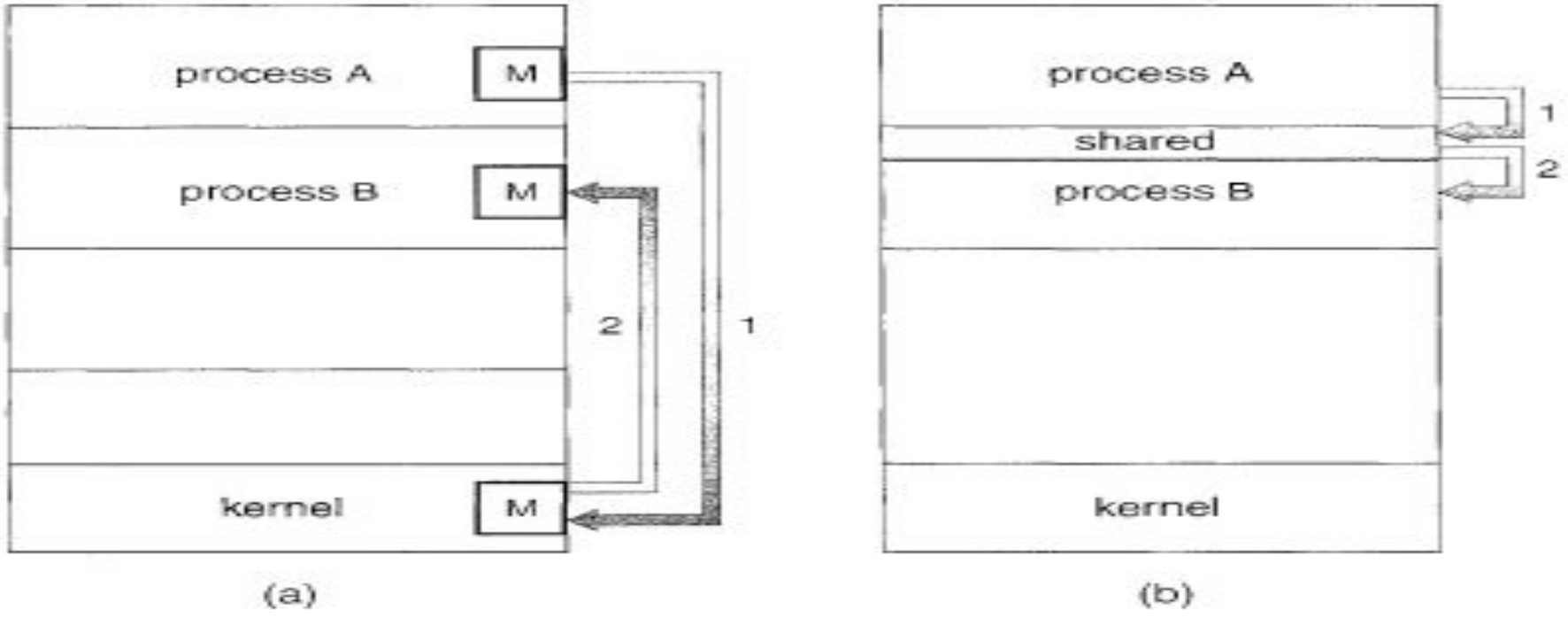


Figure 3.13 Communications models. (a) Message passing. (b) Shared memory.

Shared-memory

- To illustrate the concept of cooperating processes, let's consider the producer-consumer problem, A producer process produces information that is consumed by a consumer process.
- For example, a compiler may produce assembly code, which is consumed by an assembler.
- The assembler, in turn, may produce object modules, which are consumed by the loader.
- We generally think of a server as a producer and a client as a consumer. For example, a Web server produces (that is, provides) HTML files and images, which are consumed (that is, read) by the client Web browser requesting the resource.
- 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.

Message-passing

- Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via a message-passing facility.
- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.
- For example, a chat program used on the World Wide Web 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) and receive(message).
- Messages sent by a process can be of either fixed or variable size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult. 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
- Synchronous or asynchronous communication
- Automatic or explicit buffering

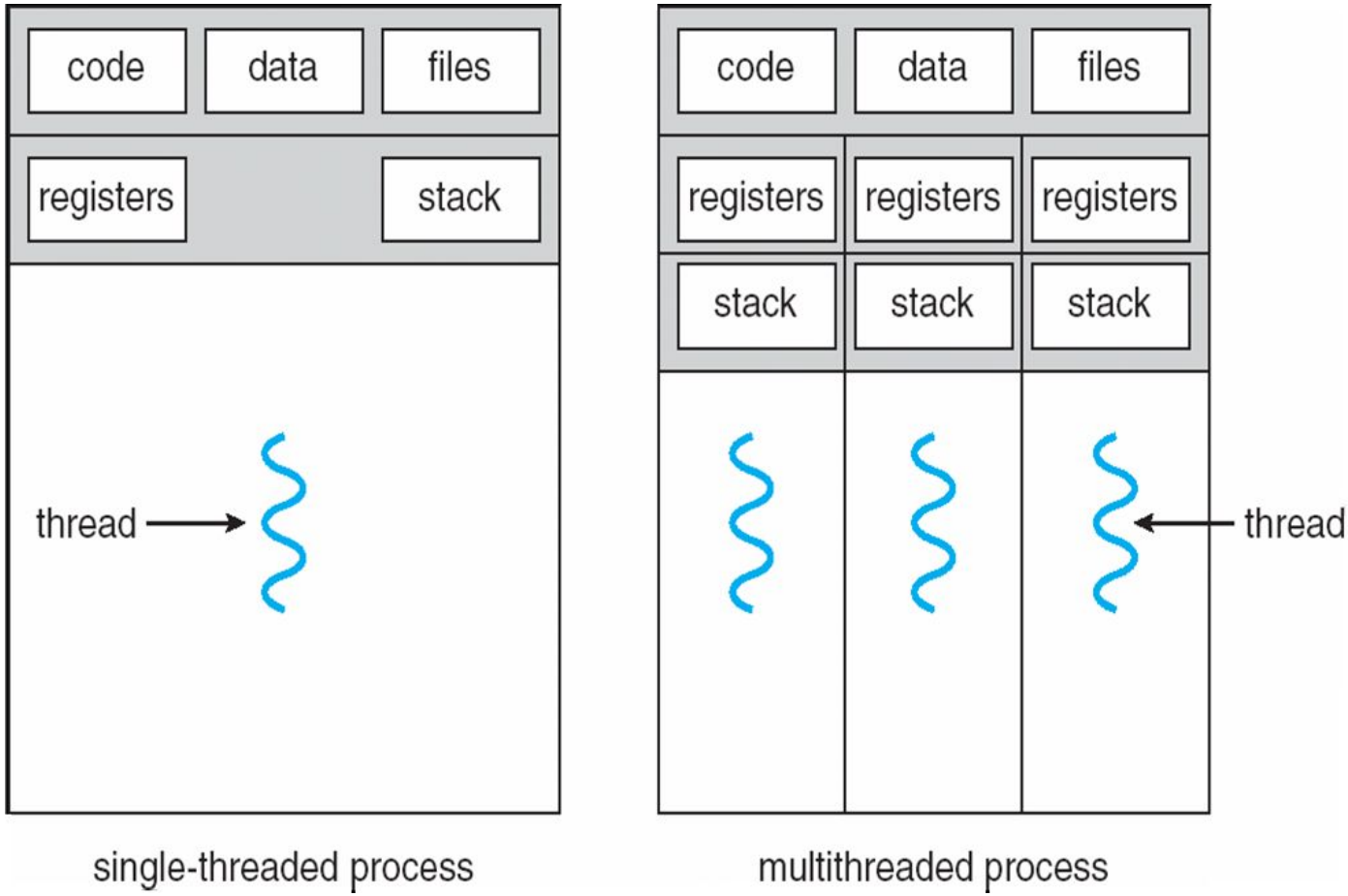
Threads: Overview

- A process is a program that performs a single thread of execution.
- A thread is a basic unit of CPU utilization;
- It comprises a thread ID, a program counter, a register set, and a stack.
- It shares its code section, data section, and other operating-system resources with the other processes, such as open files and signals.
- A traditional (or heavy weight:) process has a single thread of control.
- If a process has multiple threads of control, it can perform more than one task at a time. Many software packages that run on modern desktop PCs are multithreaded.
- An application typically is implemented as a separate process with several threads of control.
- A Web browser might have one thread display images ortext while another thread retrieves data from the network.
- A word processor may have a thread for displaying graphics, another thread for responding to key strokes from the user, and a third thread for performing spelling and grammar checking in the background.
- In certain situations, a single application may be required to perform several similar tasks. For example, a Web server accepts client requests for Web pages, images, sound, and so forth. A busy Web server may have several (perhaps thousands of) clients concurrently accessing it. If the Web server ran as a traditional single-threaded process, it would be able to service only one client at a time, artd a client might have to wait a very long time for its request to be serviced.

Process	Thread
Process means any program is in execution.	Thread means segment of a process.
Process takes more time to terminate.	Thread takes less time to terminate.
It takes more time for creation.	It takes less time for creation.
It also takes more time for context switching.	It takes less time for context switching.
Process is less efficient in term of communication.	Thread is more efficient in term of communication.
Multi programming holds the concepts of multi process.	We don’t need multi programs in action for multiple threads because a single process consists of multiple threads.
Process is isolated.	Threads share memory.
Process is called heavy weight process.	A Thread is lightweight as each thread in a process shares code, data and resources.
Process switching uses interface in operating system.	Thread switching does not require to call a operating system and cause an interrupt to the kernel.
If one process is blocked then it will not effect the execution of other process	Second thread in the same task could not run, while one server thread is blocked.
Process has its own Process Control Block, Stack and Address Space.	Thread has Parents’ PCB, its own Thread Control Block and Stack and common Address space.
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.
Changes to the parent process does not affect child processes.	Since all threads of the same process share address space and

Single and Multithreaded Processes

- One solution is to have the server run as a single process that accepts requests.
- When the server receives a request, it creates a separate process to service that request.
- Process creation is time consuming and resource intensive.
- It is generally more efficient to use one process that contains multiple threads.
- If the Web-server process is multithreaded, the server will create a separate thread that listens for client requests.

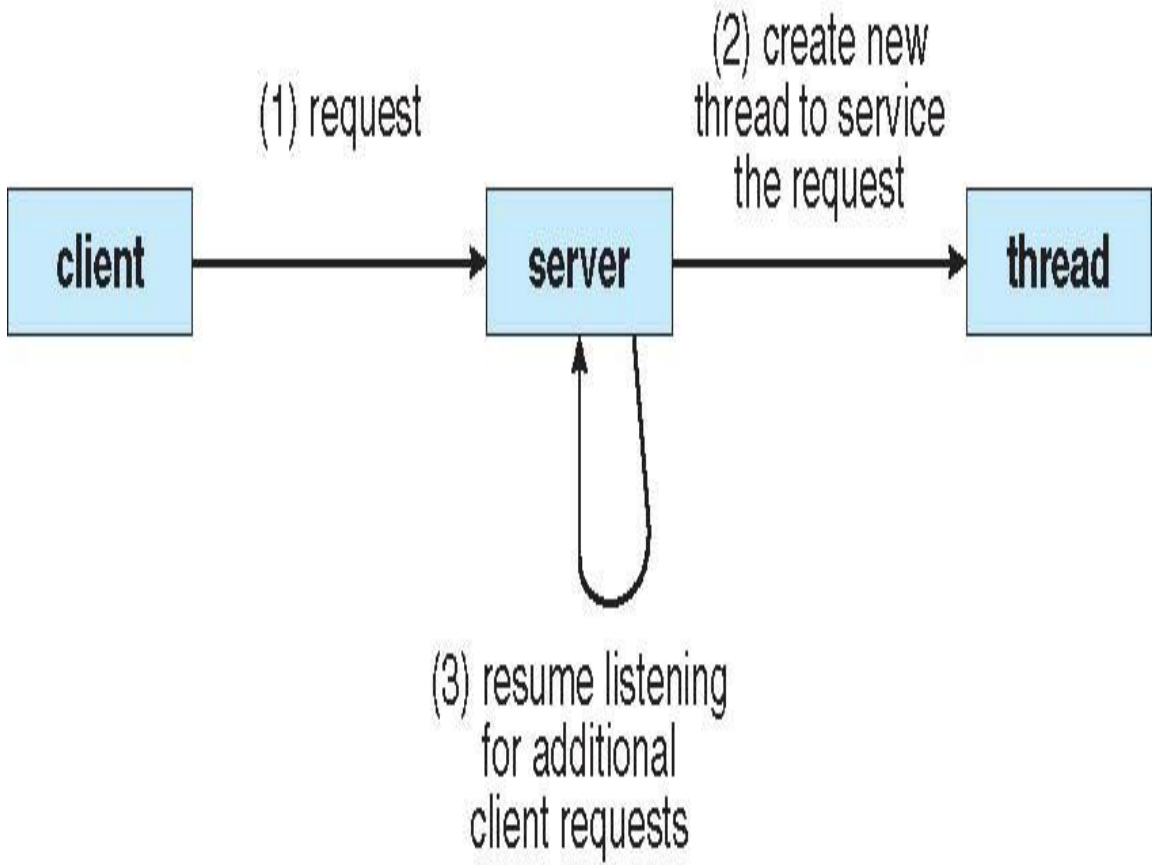


single-threaded process

multithreaded process

Multithreaded server architecture.

- When a request is made, rather than creating another process, the server will create a new thread to service the request and resume listening for additional requests.
- This is illustrated in Figure.
- Threads also play a vital role in remote procedure call (RPC) systems.
- RPC servers are multithreaded.
- When a server receives a message, it services the message using a separate thread.
- This allows the server to service several concurrent requests.
- Finally, most operating system kernels are now multithreaded; several threads operate in the kernel, and each thread performs a specific task, such as managing devices or interrupt handling.
- For examplef Solaris creates a set of threads in the kernel specifically for interrupt handling; Linux uses a kernel thread for managing the amount of free memory in the system.



Benefits of Multithreaded Programming

- **Responsiveness:** Multithreading may allow a program to continue running even if part of it is blocked or is performing a lengthy operation. Hence increases responsiveness to the user.
- For instance, a multithreaded Web browser could allow user interaction in one thread while an image was being loaded in another thread.
- **Resource sharing:** Processes may only share resources through techniques such as shared memory or message passing. Such techniques must be explicitly arranged by the programmer.
- However, threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
- **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. Empirically gauging the difference in overhead can be difficult, but in general it is much more time consuming to create and manage processes than threads. In Solarisf for example, creating a process is about thirty times slower than is creating a thread, and context switching is about five times slower.
- **Scalability:** The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors. A single-threaded process can only run on one processor, regardless how many are available. Multithreading on a multi-CPU machine increases parallelism.

Multicore Programming

- A recent trend in system design has been to place multiple computing cores on a single chip, where each core appears as a separate processor to the operating system.
- Multithreaded programming provides a mechanism for more efficient use of multiple cores and improved concurrency.
- Consider an application with four threads. On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time (Figure 4.3), as the processing core is capable of executing only one thread at a time.
- On a system with multiple cores, however, concurrency means that the threads can run in parallel, as the system can assign a separate thread to each core (Figure 4.4).
- The trend towards multicore systems has placed pressure on system designers as well as application programmers to make better use of the multiple computing cores.
- Designers of operating systems must write scheduling algorithms that use multiple processing cores to allow the parallel execution shown in Figure 4.4.
- For application programmers, the challenge is to modify existing programs as well as design new programs that are multithreaded to take advantage of multicore systems.

Multicore Programming

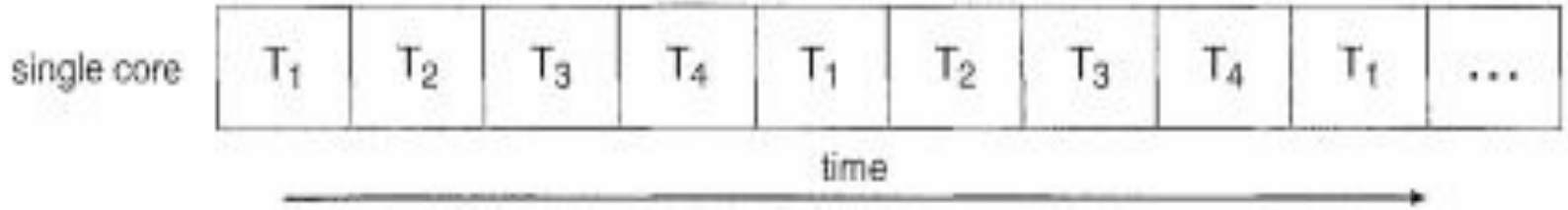


Figure 4.3 Concurrent execution on a single-core system.

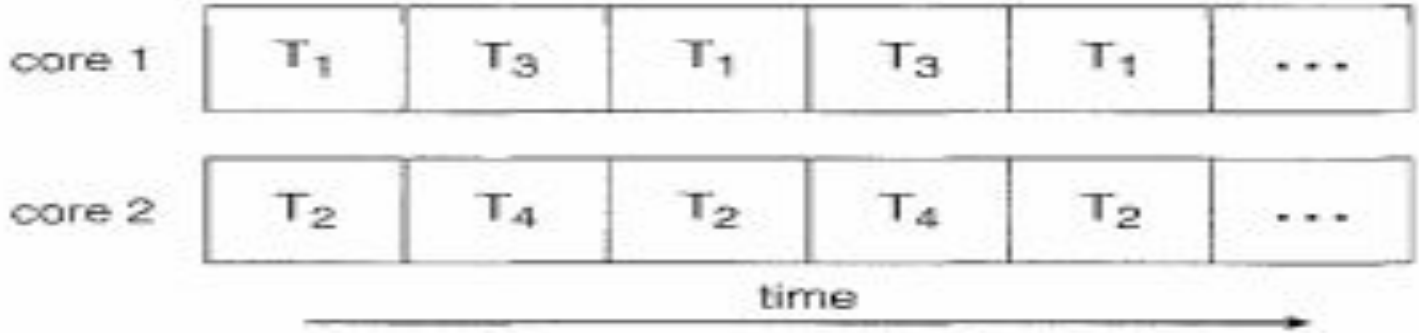


Figure 4.4 Parallel execution on a multicore system.

Multicore Programming

- **In general, five areas present challenges in programming for multicore systems:**
- **Dividing activities:** Examining applications to find areas that can be divided into separate, concurrent tasks and thus can run in parallel on individual cores.
- **Balance:** While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value. In some instances, a certain task may not contribute as much value to the overall process as other tasks; using a separate execution core to run that task may not be worth the cost.
- **Data splitting:** Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.
- **Data dependency:** The data accessed by the tasks must be examined for dependencies between two or more tasks. In instances where one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency.
- **Testing and debugging:** When a program is running in parallel on multiple cores, there are many different execution paths. Testing and debugging such concurrent programs is inherently more difficult than testing and debugging single-threaded applications.

- **Because of these challenges, many software developers argue that the advent of multicore systems will require an entirely new approach to designing software systems in the future.**

Multithreading Models

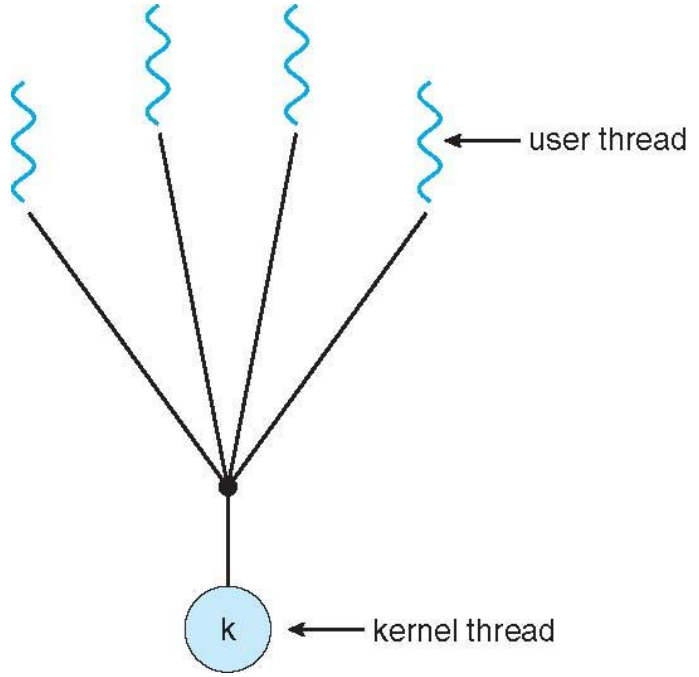
- Support for threads may be provided either at the user level, for user threads or by the kernel, for kernel threads. User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system.
- **Three common ways of establishing relationship between user threads and kernel threads.**

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

•The many-to-one model maps many user-level threads to one kernel thread.

- Thread management is done by the thread library in userspace, so it is efficient; but the entire process will block if a thread makes a blocking system call.
- Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.



Examples:

Solaris Green Threads

GNU Portable Threads

As the thread management is done in user space:

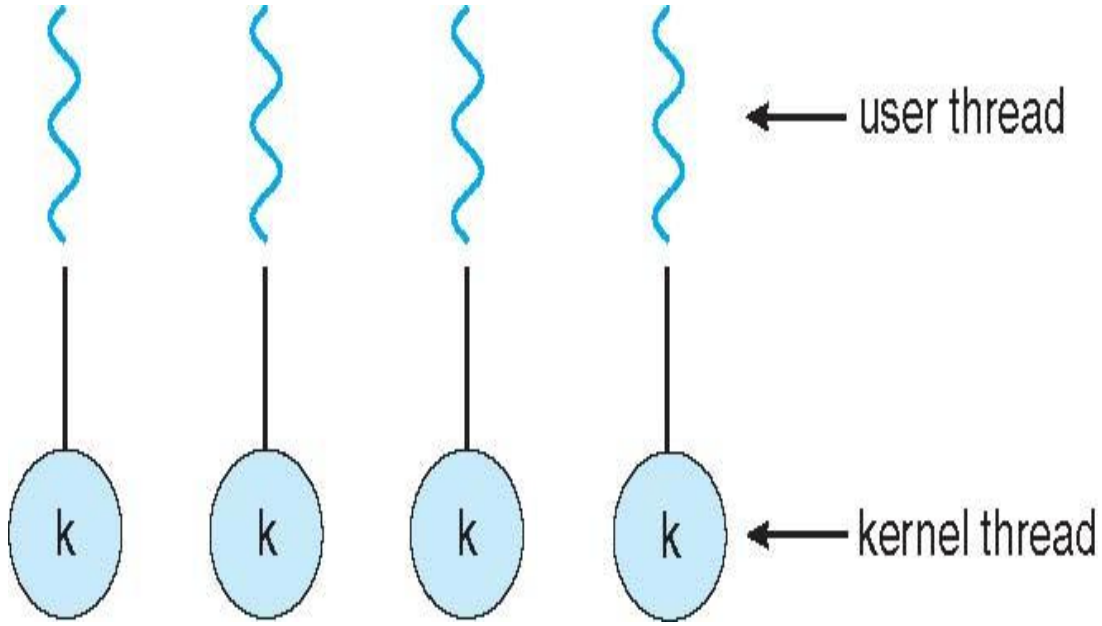
Advantages: Efficient thread management.

Disadvantages: Bad resource management.

One-to-One

•The one-to-one model maps each user thread to a kernel thread.

- It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call;
- It also allows multiple threads to run in parallel on multiprocessors.
- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.
- Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system.
- Linux, along with the family of Windows operating systems, implement the one-to-one model.
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later



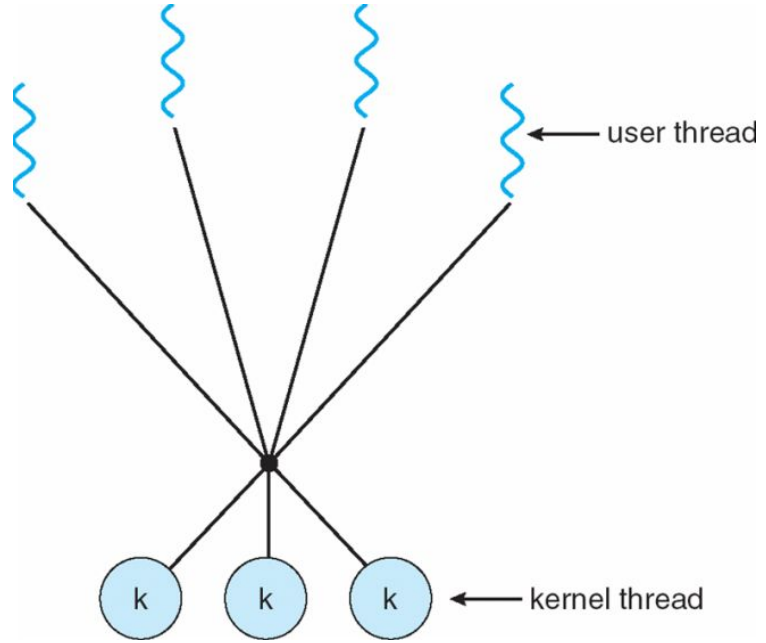
Advantages:
- Good resource management.
- High concurrency.

Disadvantages:
Each user thread **creating a user thread requires creating the corresponding kernel thread:**
- which makes the system **slow at thread creation time.**
- The system **resources** may **exhaust** when many threads are created.

Many-to-Many Model

•The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.

- The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than on a uniprocessor). Whereas the many-to-one model allows the developer tocreate as many user threads as she wishes, true concurrency is not gained because the kernel can schedule only one thread at a time.
- The one-to-one model allows for greater concurrency, but the developer has to be careful not to create too many threads within an application. The many-to-many model suffers from neither of these shortcomings: developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.
- Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.



Advantages :

- **Developers** can create as many user threads **as necessary** in the application.
- The corresponding kernel threads can run in **parallel** on a multiprocessor.
- Also, when a thread performs a **blocking** system call, the kernel can schedule another thread for execution.

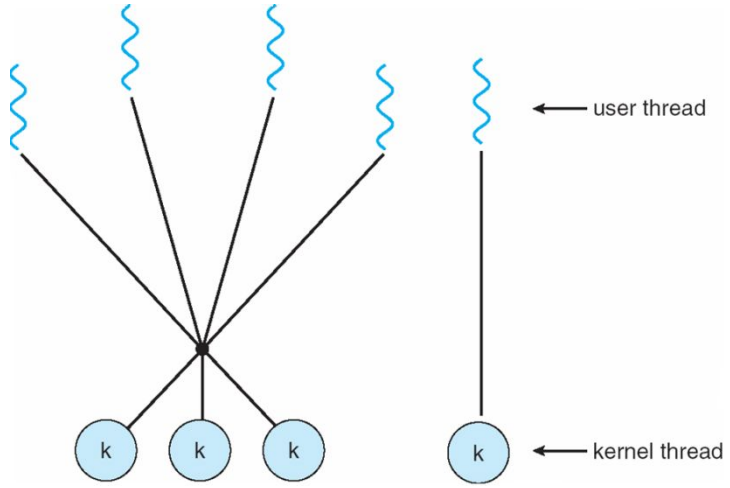
Disadvantages :

- Very complex to implement
- If kernel threads amount is not enough then the performance will be low.

Two-level Model

- One popular variation on the many-to-many model still multiplexes many user-level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread.
- This variation, sometimes referred to as the two-level model (Figure 4.8), is supported by operating systems such as IRlX, HP-UX, and Tru64 UNIX.
- The Solaris operating system supported the two-level model in versions older than Solaris 9.
- However, beginning with Solaris 9, this system uses the one-to-one model.
- We can use M:M,1:1 model

- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



CPU Scheduling/ Process Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Algorithm Evaluation

Basic Concepts

- CPU scheduling is the basis of multiprogrammed operating systems.
- In a single-processor system, only one process can run at a time. The objective of multiprogramming is to maximize CPU utilization. With multiprogramming, we try to 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.
- Scheduling of this kind is a fundamental operating-system function.
- Almost all computer resources are scheduled before use.
- The CPU is one of the primary computer resources.
- Thus, its scheduling is central to operating-system design.

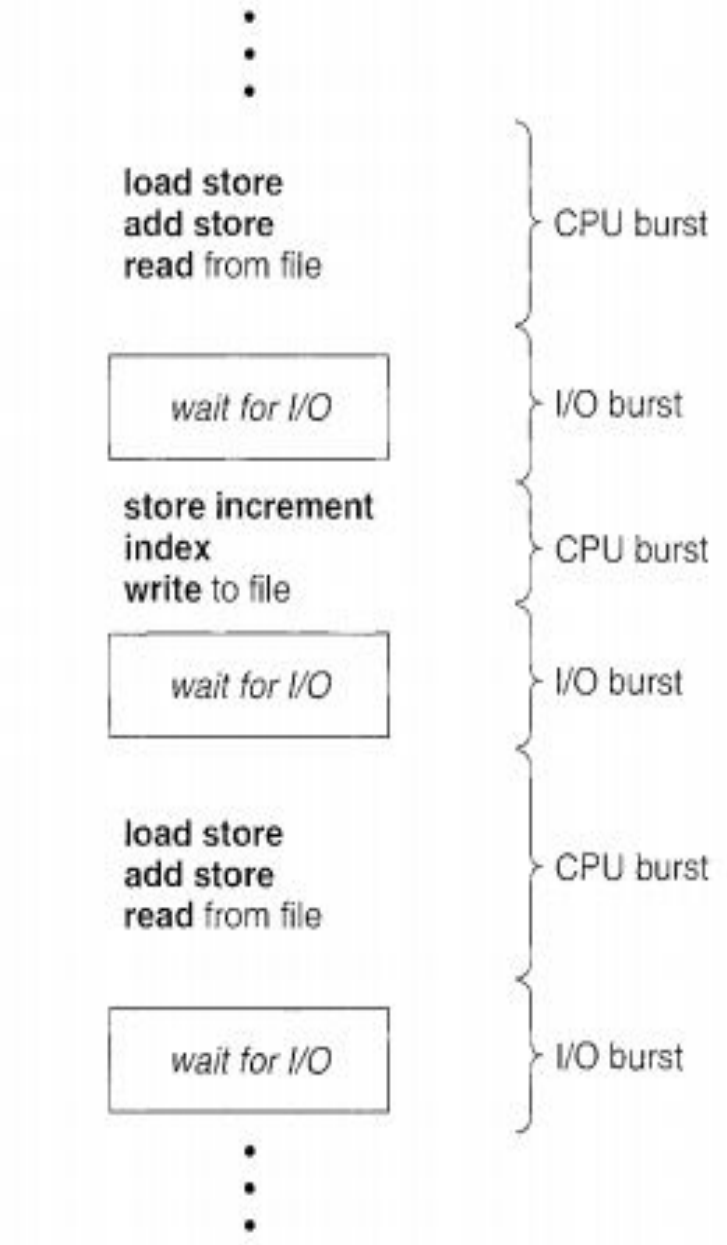


Figure 5.1 Alternating sequence of CPU and I/O bursts.

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/0 wait.
- Processes alternate between these two states.
- Process execution begins with a CPU burst followed by an I/O burst, which is followed by another CPU burst, then another I/0 burst, and so on.
- Eventually, the final CPU burst ends with a system request to terminate execution.
- The durations of CPU bursts have been measured extensively.
- Although they vary greatly from process to process and from computer to computer, they tend to have a frequency curve similar to that shown in Figure 5.2.

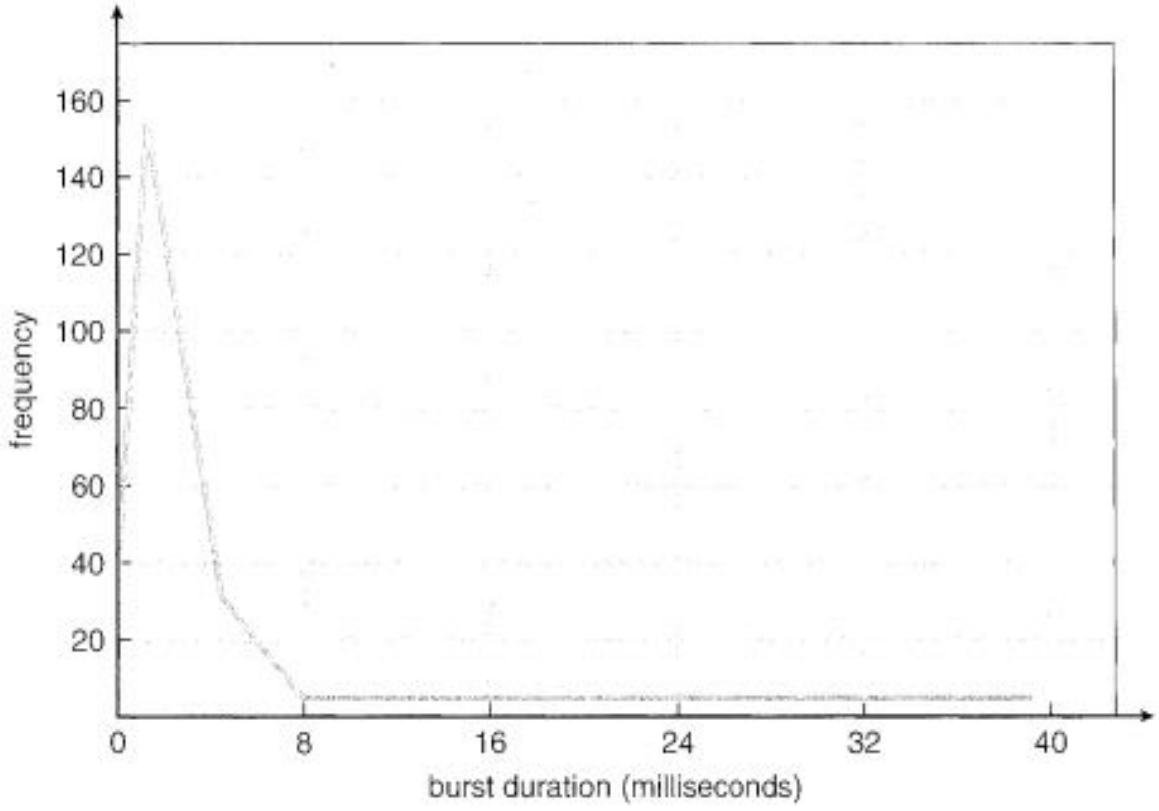


Figure 5.2 Histogram of CPU-burst durations.

- **The curve is generally characterized as exponential or hyperexponential, with a large number of short CPU bursts and a small number of long CPU bursts.**
- **An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts. This distribution can be important in the selection of an appropriate CPU-scheduling algorithm.**

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.
- Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue.
- As we shall see when we consider the various scheduling algorithms, a ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list. The records in the queues are generally process control blocks (PCBs) of the processes.

Preemptive 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/0 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/0)
4. When a process terminates

- For situations 1 and 4, there is no choice in terms of scheduling.
- A new process (if one exists in the ready queue) must be selected for execution.
- There is a choice for situations 2 and 3.
- When scheduling takes place only under circumstances **1 and 4**, we say that the scheduling scheme is **nonpreemptive or cooperative; otherwise, it is preemptive.**
- **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. This scheduling method was used by Microsoft Windows 3.x;

Preemptive Scheduling

- Windows 95 introduced **preemptive scheduling**, and all subsequent versions of Windows operating systems have used preemptive scheduling.
- The Mac OS X operating system for the Macintosh also uses preemptive scheduling; previous versions of the Macintosh operating system relied on cooperative scheduling.

- **Cooperative scheduling** is the only method that can be used on certain hardware platforms, because it does not require the special hardware (for example, a timer) needed for **preemptive scheduling.**
- Unfortunately, **preemptive scheduling** incurs a cost associated with access to shared data.
- Consider the case of two processes that share data. While one is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state. In such situations, we need new mechanisms to coordinate access to shared data. Preemption also affects the design of the operating-system kernel.
- During the processing of a system call, the kernel may be busy with an activity on behalf of a process. Such activities may involve changing important kernel data (for instance, I/0 queues).

Process	Arrival time	CPU Burst Time (in millisecond
P0	2	8
P1	3	6
P2	0	9
P3	1	4

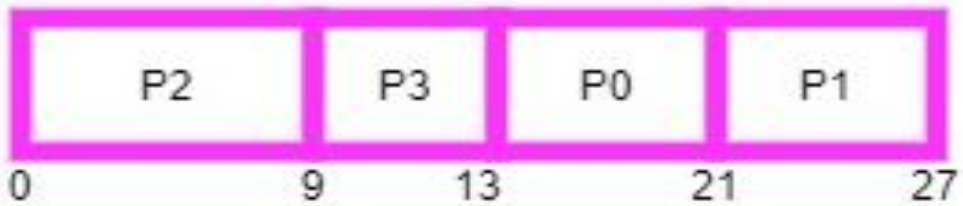


Figure: Non-Preemptive Scheduling

Process	Arrival time	CPU Burst Time (in millisecond
P0	2	3
P1	3	5
P2	0	6
P3	1	5

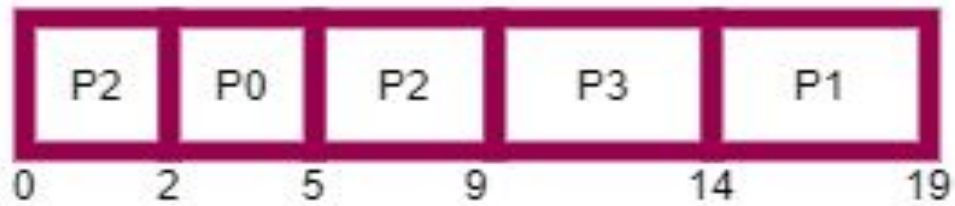


Figure: Preemptive Scheduling

Preemptive Scheduling	Non-Preemptive Scheduling
The resources are assigned to a process for a particular time period.	Once resources are assigned to a process, they are held until it completes its burst period or changes to the waiting state.
The process can be interrupted, even before the completion.	The process is not interrupted until its life cycle is complete.
When a high-priority process continuously comes in the ready queue, a low-priority process can starve.	When a high burst time process uses a CPU, another process with a shorter burst time can starve.
It is flexible.	It is rigid.
It is cost associated.	It does not cost associated.
It has overheads associated with process scheduling.	It doesn't have overhead.
It affects the design of the operating system kernel.	It doesn't affect the design of the OS kernel.
Its CPU utilization is very high.	Its CPU utilization is very low.
Examples: Round Robin and Shortest Remaining Time First	FCFS and SJF are examples of non-preemptive scheduling.

Dispatcher

- The dispatcher is the module that gives control of the CPU to the process selected by the short-term 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
- The dispatcher should be as fast as possible, since it is invoked during every process switch.
- The time it takes for the dispatcher to stop one process and start another running is known as the dispatch latency.

Scheduling Criteria

- **CPU utilization.** We want to keep the CPU as busy as possible. 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). **Maximize CPU utilization**

- **Throughput**. If the CPU is busy executing processes, then work is being done. One measure of work is the 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 ten processes per second. **Maximize throughput**

- **Turnaround time.** From the point of view of a particular process, the important 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/0. **Minimize turnaround time**

- **Waiting time.** The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/0; 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. **Minimize waiting time**

- **Response time.** In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the 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, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device. **Minimize 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

Scheduling Algorithms

- First Come, First Served (FCFS)
- Shortest Job First (SJF)
- Priority
- Round Robin (RR)

First Come, First Served (FCFS) Scheduling

- Simplest scheduling algorithm that schedules according to arrival times of processes.
- First come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first.
- It is implemented by using the 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.
- FCFS is a non-preemptive scheduling algorithm.
- First come first serve suffers from convoy effect.
- Poor in performance as average wait time is high.

First-Come, First-Served (FCFS) Scheduling

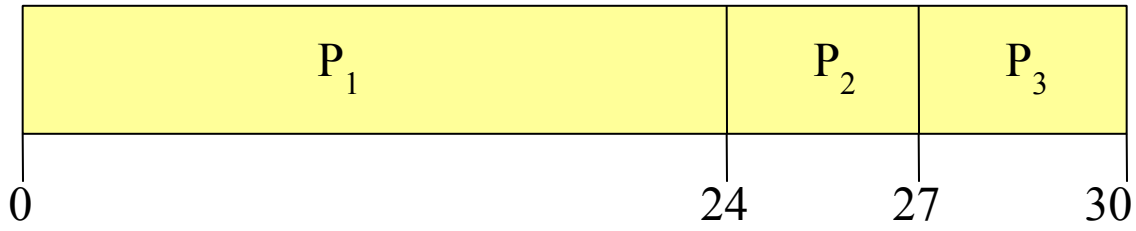
Process	Burst Time in ms
P_1	24
P_2	3
P_3	3

Process	Burst Time	Completion time	TAT	WT	RT
P1	24	24	24	0	0
P2	3	27	27	24	24
P3	3	30	30	27	27

- With FCFS, the process that requests the CPU first is allocated the CPU first
- Case #1: Suppose that the processes arrive in the order: P_1 , P_2 , P_3 at time t0
  Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$ ms
- Average waiting time: (0 + 24 + 27)/3 = 17 ms
- Average turn-around time: (24 + 27 + 30)/3 = 27 ms
-

Turn Around Time = Completion Time – Arrival Time
Waiting Time = Turn Around Time – Burst Time
Response time= CPU FIRST TIME-ARRIVAL TIME

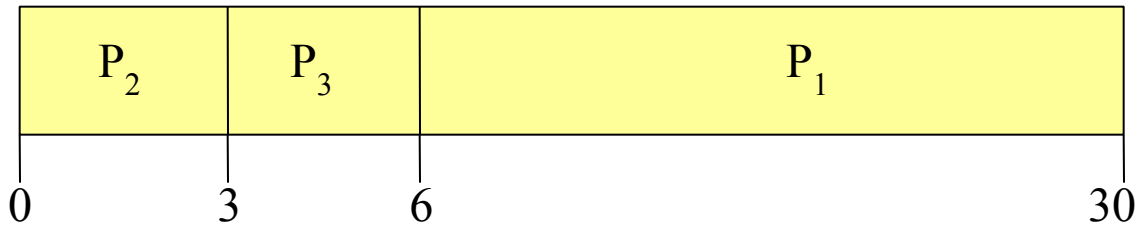
The Gantt Chart for the schedule is:



FCFS Scheduling (Cont.)

- Case #2: Suppose that the processes arrive in the order: P_2 , P_3 , P_1
  The Gantt chart for the schedule is:

Process	Burst Time
P_1	24
P_2	3
P_3	3



- Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$ ms
- Average waiting time: (6 + 0 + 3)/3 = 3 (Much better than Case #1)
- Average turn-around time: (30+3+6)/3 = 13 ms

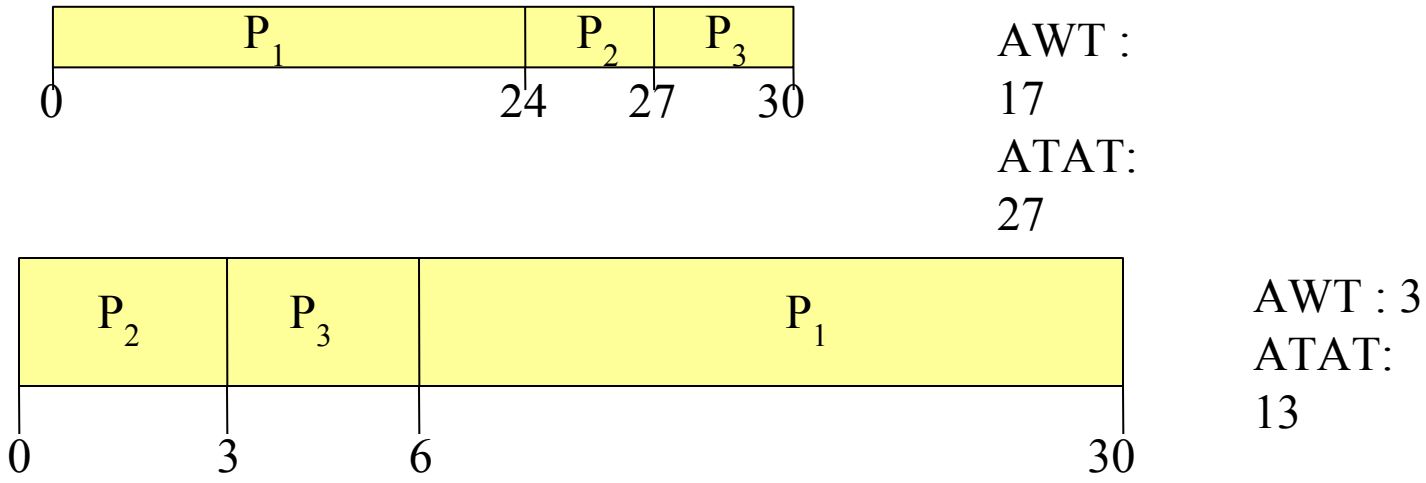
Turn Around Time = Completion Time – Arrival Time
Waiting Time = Turn Around Time – Burst Time
Response time= CPU FIRST TIME-ARRIVAL TIME

Process	Burst Time	Completion time	TAT	WT	RT
P1	24	30	30	6	6
P2	3	3	3	0	0
P3	3	6	6	3	3

CONVOY EFFECT

- Case #1 is an example of the **convoy effect;**
  - all the other processes wait for one long-running process to finish using the CPU
  - This problem results in **lower CPU and device utilization;**

  - Case #2 shows that higher utilization might be possible if the short processes were allowed to run first



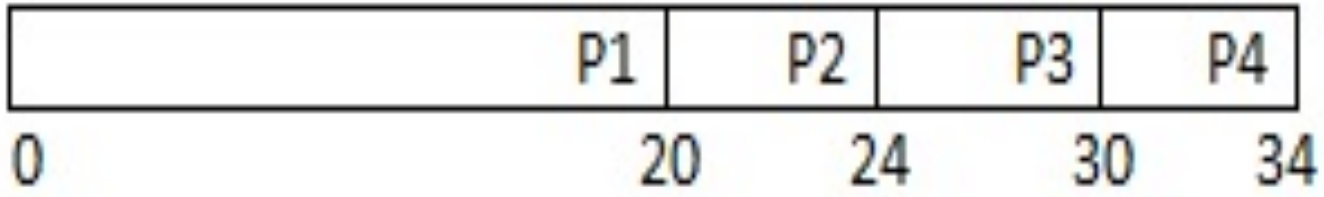
FCFS disadvantages

- The FCFS scheduling algorithm is non-preemptive
- Once the CPU has been allocated to a process, that process keeps the CPU until it releases it either by terminating or by requesting I/O.
- It is a troublesome algorithm for time-sharing systems
- Not optimal Average Waiting Time.
- Resources utilization in parallel is not possible, which leads to Convoy Effect, and hence poor resource(CPU, I/O etc) utilization.

First Come, First Served (FCFS) Scheduling

- Let us discuss an example, some process p with the CPU burst.
- Process CPU Burst
- P1 20
- P2 4
- P3 6
- P4 4
- Gantt Chart Show the result as below

Process	Burst Time	Completion time	TAT	WT	RT
P1	20	20	20	0	0
P2	4	24	24	20	20
P3	6	30	30	24	24
P4	4	34	34	30	30



Turn Around Time = Completion Time – Arrival Time
Waiting Time = Turn Around Time – Burst Time
Response time= CPU FIRST TIME-ARRIVAL TIME

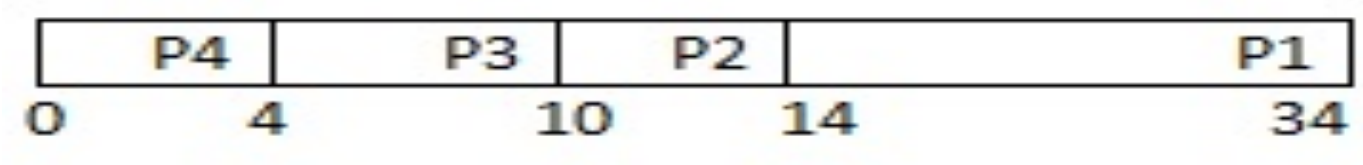
Waiting time :
Waiting time for process P1=0, P2=20, P3=24 and P4=30.
Average Waiting time = (0+20+24+30)/4=18.5.

First Come, First Served (FCFS) Scheduling

- Example2 . If we Change the execution Gantt chart shown the result.
- Process CPU Burst
- P1 20
- P2 4
- P3 6
- P4 4
- Gantt Chart Shown the result as below

Process	Burst Time	Completion time	TAT	WT	RT
P1	20	34	34	14	14
P2	4	14	14	10	10
P3	6	10	10	4	4
P4	4	4	4	0	0

Turn Around Time = Completion Time – Arrival Time
Waiting Time = Turn Around Time – Burst Time
Response time= CPU FIRST TIME-ARRIVAL TIME



- Waiting time : Waiting time for process P1=14, P2=10, P3=4, and P4=0.
- Now Average Waiting Time = (0+4+10+14)/4=7

First Come, First Served (FCFS) Scheduling

PROCESS WITH ITS ID AND BURST TIME

Process ID	Burst Time (ms)
P_1	10
P_2	2
P_3	8
P_4	6

- Waiting time :
- Waiting time for process P1=0, P2=10, P3=12 and P4=20.
- Average Waiting time = (0+10+12+20)/4=10.5.

a. First Come First Serve

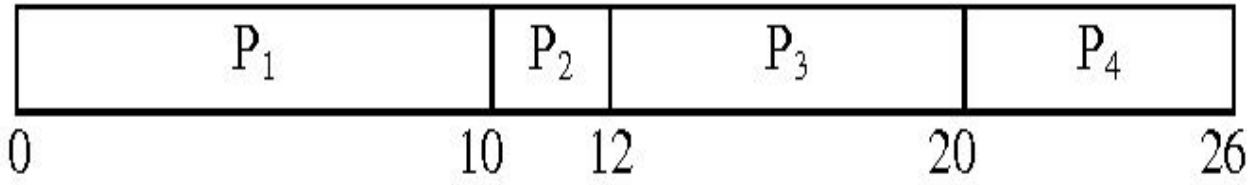


Figure II: Gantt chart for FCFS

Shortest-Job-First (SJF) Scheduling

- The SJF algorithm associates with each process the length of its next CPU burst.
- **When the CPU becomes available, it is assigned to the process that has the smallest next CPU burst.**
- If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.
- The term for this scheduling method would be the shortest-next-CPU-burst algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length.
- The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes.
- Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.
- The real difficulty with the SJF algorithm is knowing the length of the next CPU request.

Shortest-Job-First (SJF) Scheduling

- For long-term (job) scheduling in a batch system, we can use as the length the process time limit that a user specifies when he submits the job.
- Thus, users are motivated to estimate the process time limit accurately, since a lower value may mean faster response. (Too low a value will cause a time-limit-exceeded error and require resubmission.)
- SJF scheduling is used frequently in long-term scheduling.
- Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling.
- The SJF algorithm can be either preemptive or nonpreemptive.
- The choice arises when a new process arrives at the ready queue while a previous process is still executing.
- The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process.
- A preemptive SJF algorithm will preempt the currently executing process, whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called shortest-remaining-time-first scheduling.

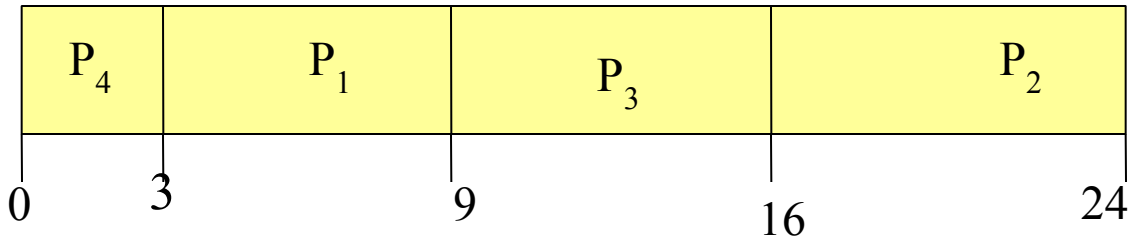
Example #1 Shortest-Job-First (SJF) Scheduling

Process	Burst Time in ms
P_1	6
P_2	8
P_3	7
P4	3

Process	Burst Time	Completion time	TAT	WT	RT
P1	6	9	9	3	3
P2	8	24	24	16	16
P3	7	16	16	9	9
P4	3	3	3	0	0

- Case #1: Suppose that the processes arrive in the order: P_1 , P_2 , P_3 , P_4 at time t0

  The Gantt Chart for the schedule is:



- Waiting time for P_1 = 3; P_2 = 16; P_3 = 9; P_4 = 0 ms
- Average waiting time: (3 + 16 + 9 + 0)/4 =28/4=7ms
- Average turn-around time: (9 + 24 + 16 + 3)/4 =52/4=13ms

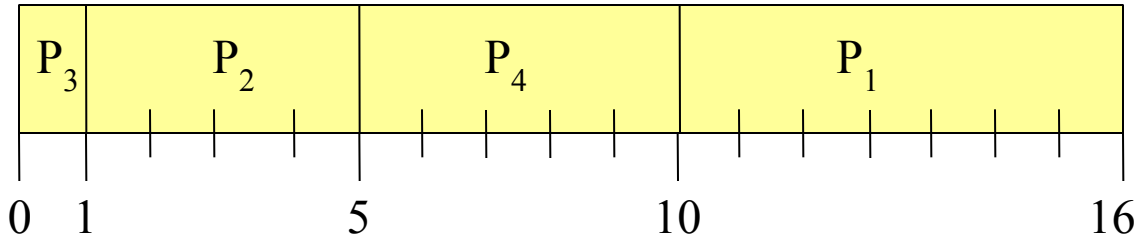
Turn Around Time = Completion Time – Arrival Tim
Waiting Time = Turn Around Time – Burst Time
Response time= CPU FIRST TIME-ARRIVAL TIME

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

Process	Arrival Time	Burst Time
P_1	0	6
P_2	0	4
P_3	0	1
P_4	0	5

Process	Burst Time	Completion time	TAT	WT	RT
P1	6	16	16	10	10
P2	4	5	5	1	1
P3	1	1	1	0	0
P4	5	10	10	5	5

- SJF (non-preemptive, simultaneous arrival)



- Average waiting time = (10 +1 + 0 + 5)/4 = 16/4 = 4 ms
- Average turn-around time = (16 + 5 + 1 + 10) / 4 = 32/4 = 8 ms

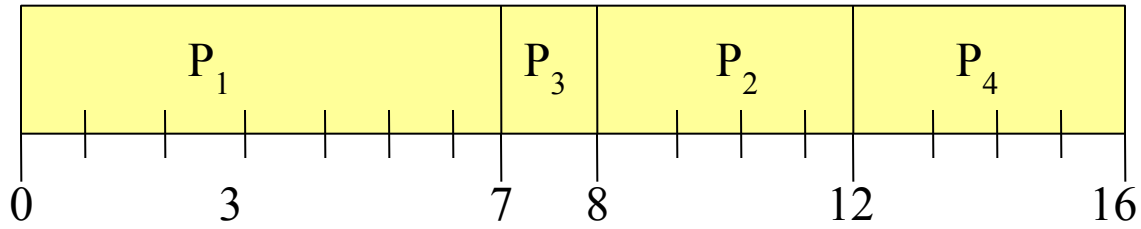
Turn Around Time = Completion Time – Arrival Time
Waiting Time = Turn Around Time – Burst Time
Response time= CPU FIRST TIME-ARRIVAL TIME

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

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

Process	Arrival Time	Burst Time	Completion time	TAT	WT	RT
P1	0	7	7	7	0	0
P2	2	4	12	10	6	6
P3	4	1	8	4	3	3
P4	5	4	16	11	7	7

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

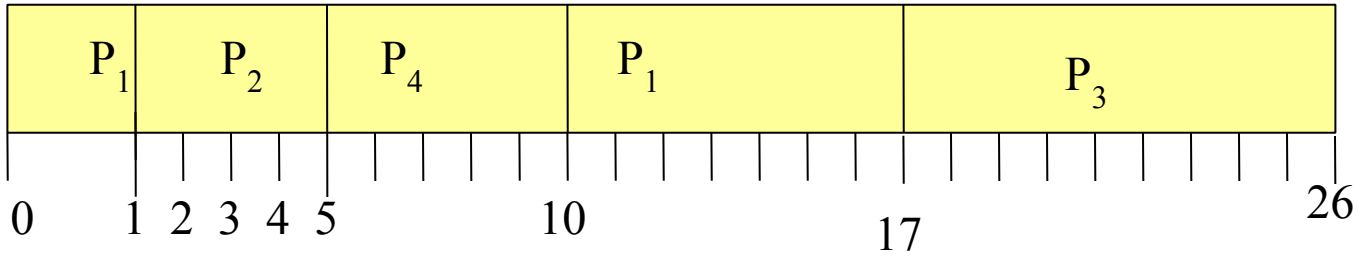
Turn Around Time = Completion Time – Arrival Time
Waiting Time = Turn Around Time – Burst Time
Response time= CPU FIRST TIME-ARRIVAL TIME

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

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Process	Arrival Time	Burst Time	Completion time	TAT	WT	RT
P1	0	8	17	17	9	0
P2	1	4	5	4	0	0
P3	2	9	26	24	15	15
P4	3	5	10	7	2	2

- SJF (preemptive, varied arrival times)



Turn Around Time = Completion Time – Arrival Time
Waiting Time = Turn Around Time – Burst Time
Response time= CPU FIRST TIME-ARRIVAL TIME

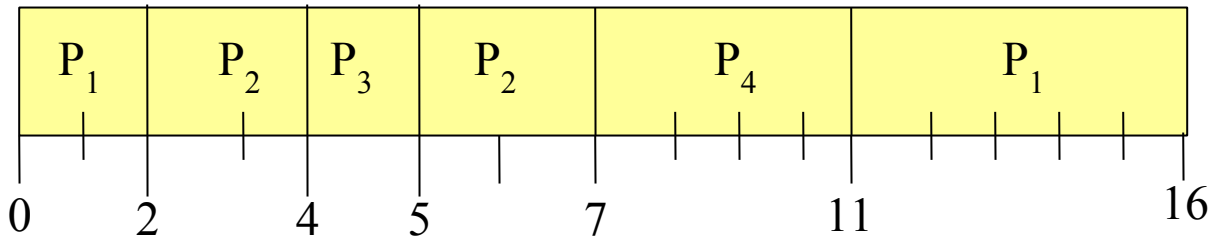
- Average waiting time
  = ([(0 – 0) + (10 - 1)] + [(1 – 1)] + [(17 – 2)] + [(5 - 3)])/4 = 9 + 0 + 15 + 2)/4 = 26/4 = 6.5 ms

- Average turn-around time = (17-0)+(5-1)+(26-2)+(10-3))/4 = (17+4+24+7)/4 = 52/4 = 13 ms

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

Process	Arrival Time	Burst Time
P_1	0	7:5:0
P_2	2	4:2:0
P_3	4	1:0
P_4	5	4:0

- SJF (preemptive, varied arrival times)



Process	Arrival Time	Burst Time	Completion time	TAT	WT	RT
P1	0	7	16	16	9	0
P2	2	4	7	5	1	0
P3	4	1	5	1	0	0
P4	5	4	11	6	2	2
				avg= 7	avg =3	

Turn Around Time = Completion Time – Arrival Time
Waiting Time = Turn Around Time – Burst Time
Response time= CPU FIRST TIME-ARRIVAL TIME

- 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
- = (16+5+1+6)/4=28/4=7 ms

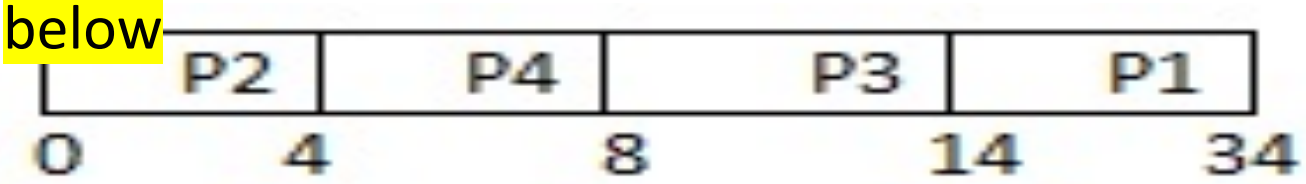
Shortest-Job-First (SJF) Scheduling

- Let us discuss an example, some process p1, p2, p3, and p4 arrived at time 0, with the CPU burst.
- Process CPU Burst
- P1 20
- P2 4
- P3 6
- P4 4
- In above example, two processes P2, P4 have the same length.
- Now to break their tie FCFS algorithm is being used by system. P2 will process first and P4 follow it.

Process	Burst Time	Completion time	TAT	WT	RT
P1	20	34	34	14	14
P2	4	4	4	0	0
P3	6	14	14	8	8
P4	4	8	8	4	4

Waiting time for process
P1=14, P2=0, P3=8 and P4=4.
Average Waiting time = (14+0+8+4)/4 = 6.5.
Average Turn Around Time
=(34+4+14+8)/4 =62/4 =15.5

Gantt Chart Shown the result as below

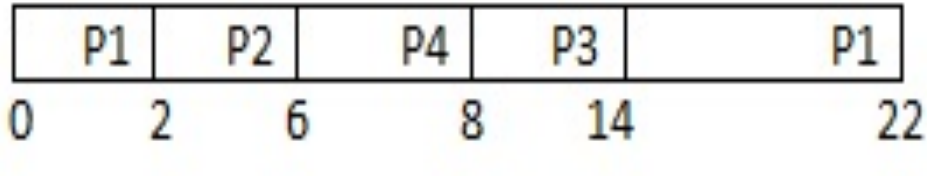


Turn Around Time = Completion Time – Arrival Time
Waiting Time = Turn Around Time – Burst Time
Response time= CPU FIRST TIME-ARRIVAL TIME

Preemptive Scheduling Algorithm example:

Process	CPU Burst	Arrival Time
P1	10	0
P2	4	2
P3	6	4
P4	2	5

Gantt Chart Shown the result as below



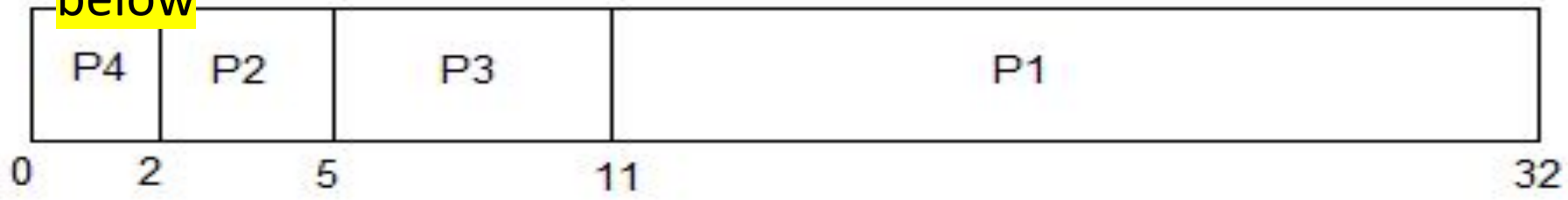
In above example, Process P1 is running after 2 ms. P2 enter in the ready queue with CPU Burst length 4.Now P1 is preempted, P2 gets CPU. In mid between of execution P3 and P4 get to enter with CPU burst 6,2.But P2 is not preempted, because of small burst. Don't think that P4 process has small CPU burst 2, because till p4 entered into ready queue P2 has only one length CPU burst remain. Now p4, p3, and P1 get executed.

Non Pre-emptive Shortest Job First

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

- Waiting time :
- Waiting time for process P1=11, P2=2, P3=5 and P4=0.
- Average Waiting time = (0+2+5+11)/4=4.5ms

Gantt Chart Shown the result as below



Priority Scheduling

- The SJF algorithm is a special case of the general priority scheduling algorithm.
- A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order.
- An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.
- Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority.
- Some systems use low numbers to represent low priority; others use low numbers for high priority.
- This difference can lead to confusion.
- In this text, we assume that low numbers represent high priority.
- Priority scheduling can be either preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.
- A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
- A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue. SJF is a priority scheduling algorithm where priority is the predicted next CPU burst time.

- The main problem with priority scheduling is starvation, that is, low priority processes may never execute.
- A solution to the problem of indefinite blockage of low-priority processes is aging.
- Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.
- For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes.
- Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed.
- In fact, it would take no more than 32 hours for a priority-127 process to age to a priority-0 process.

Priority Scheduling

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

- The Gantt chart is:

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

Average turnaround time=(16+1+18+19+6)/5 = 12 ms

Round Robin (RR) Scheduling

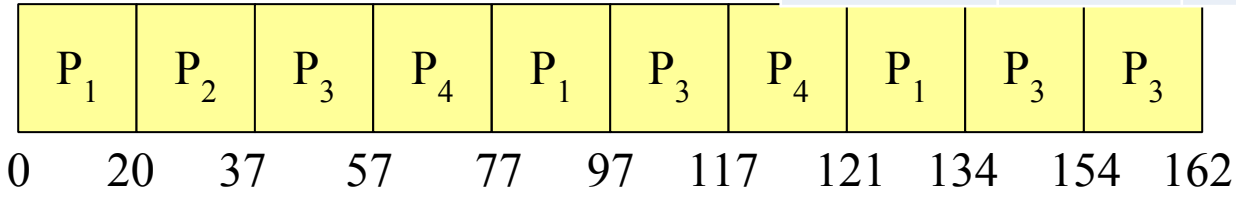
- The round-robin (RR) scheduling algorithm is designed especially for time sharing systems.
- It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes.
- A small unit of time, called a time quantum or time slice, is defined.
- A time quantum is generally from 10 to 100 milliseconds in length.
- The ready queue is treated as a circular queue.
- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.
- To implement RR scheduling, we keep the ready queue as a FIFO queue of processes.
- New processes are added to the tail of the ready queue.
- The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.
- One of two things will then happen. The process may have a CPU burst of less than 1 time quantum.
- In this case, the process itself will release the CPU voluntarily.
- The scheduler will then proceed to the next process in the ready queue.
- Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system.
- A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue. The average waiting time under the RR policy is often long.

Example of RR with Time Quantum = 20ms

Process	Burst Time
P_1	53:33 :13:0
P_2	17:0
P_3	68:48 :28:08
P_4	24:04 :00

Process	Burst Time	Completion time	TAT	WT	RT
P1	53	134	134	81	
P2	17	37	37	20	
P3	68	162	162	94	
P4	24	121	121	97	

- The Gantt chart is:



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

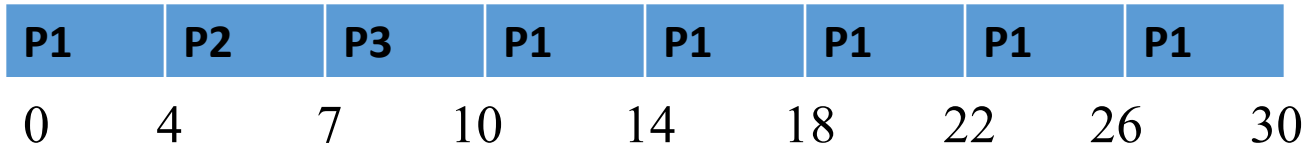
Average waiting time
= (**[(0 – 0) + (77 - 20) + (121 – 97)]** + (20 – 0) + [(37 – 0) + (97 - 57) + (134 – 117)] + **[(57 – 0) + (117 – 77)]**) / 4
= (0 + 57 + 24) + 20 + (37 + 40 + 17) + (57 + 40)) / 4
= (81 + 20 + 94 + 97)/4
= 292 / 4 = 73

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

Example of RR with Time Quantum = 4 ms

Process	Burst Time in ms
P_1	24:20:16:12:8:4:0
P_2	3
P_3	3

- The Gantt Chart for the schedule is:



find

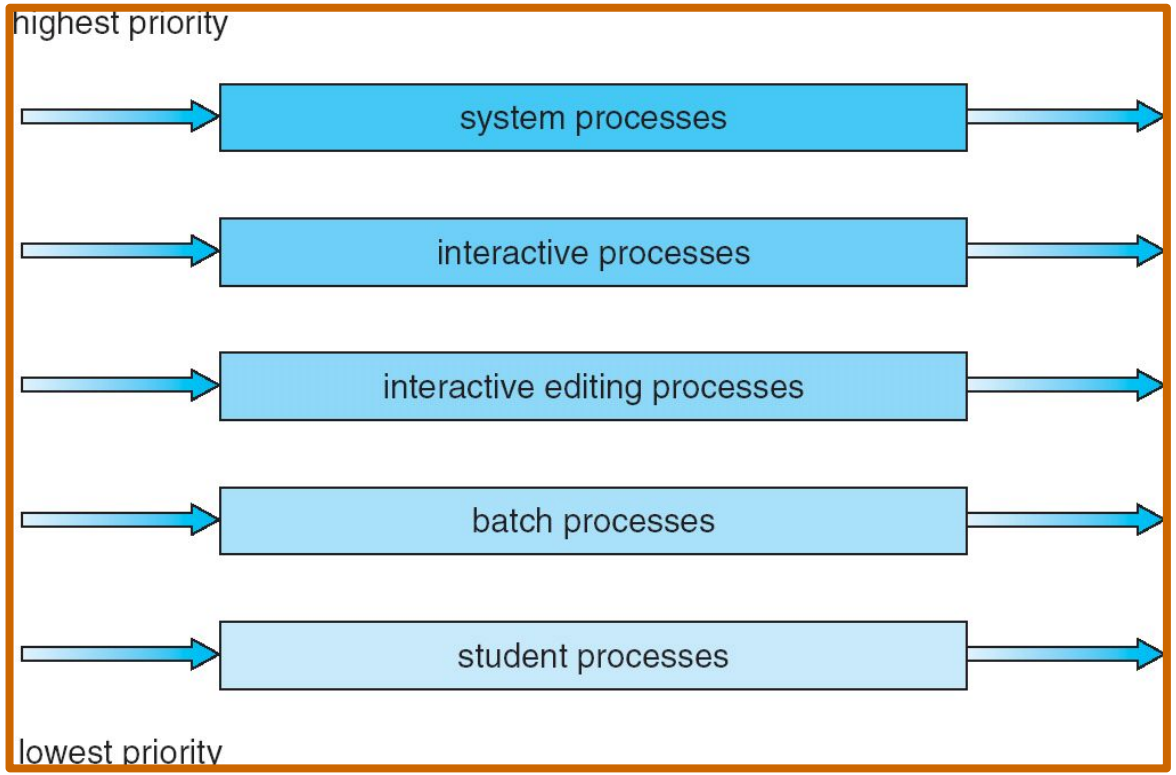
- Waiting time ?
- Average waiting time?
- Average turn-around time?

Multi-level Queue Scheduling

- Multi-level queue scheduling is used when processes can be classified into groups
- For example, **foreground** (interactive) processes and **background** (batch) processes
  - The two types of processes have different response-time requirements and so may have different scheduling needs
  - Also, foreground processes may have priority (externally defined) over background processes
- A multi-level queue scheduling algorithm partitions the ready queue into several separate queues
- The processes are permanently assigned to one queue, generally based on some property of the process such as memory size, process priority, or process type
- Each queue has its own scheduling algorithm
  - The foreground queue might be scheduled using an RR algorithm
  - The background queue might be scheduled using an FCFS algorithm
- In addition, there needs to be scheduling among the queues, which is commonly implemented as fixed-priority pre-emptive scheduling
  - The foreground queue may have absolute priority over the background queue

Multi-level Queue Scheduling

- One example of a multi-level queue are the five queues shown below
- Each queue has absolute priority over lower priority queues
- For example, no process in the batch queue can run unless the queues above it are empty
- However, this can result in starvation for the processes in the lower priority queues



Multilevel Queue Scheduling

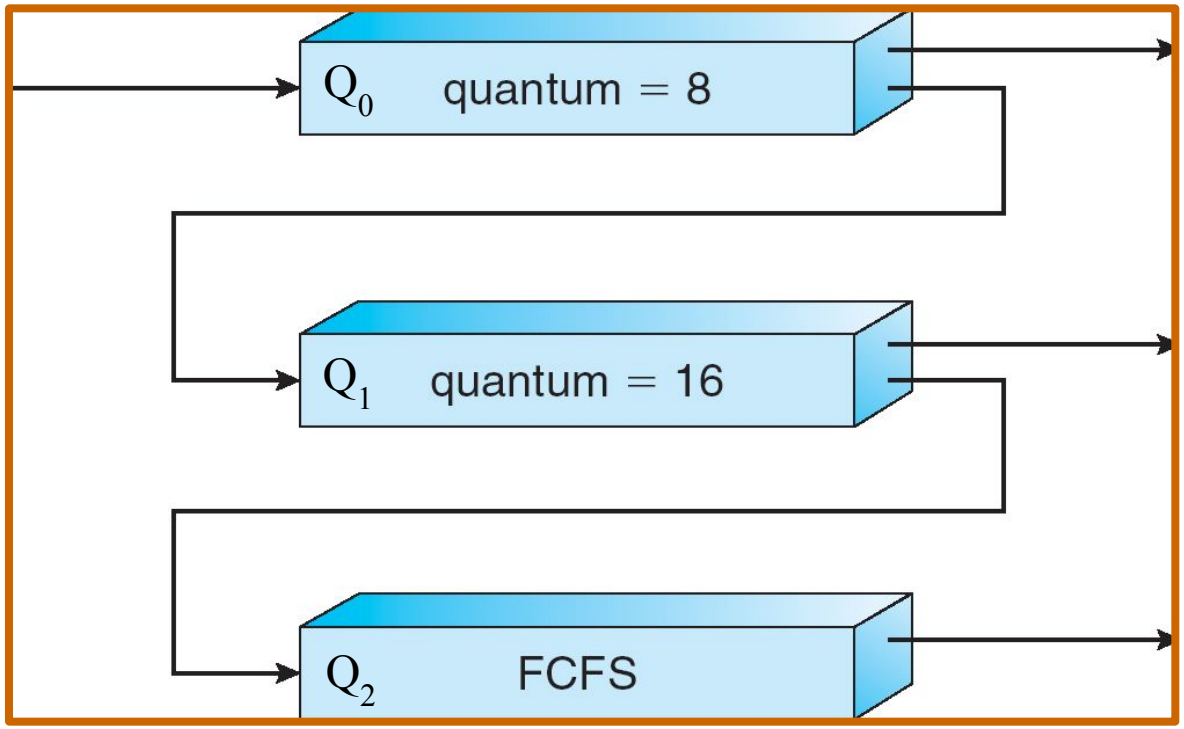
- Another possibility is to time slice among the queues
- Each queue gets a certain portion of the CPU time, which it can then schedule among its various processes
  - The foreground queue can be given 80% of the CPU time for RR scheduling
  - The background queue can be given 20% of the CPU time for FCFS scheduling

Multilevel Feedback Queue Scheduling

- In multi-level feedback queue scheduling, a process can move between the various queues; aging can be implemented this way
- A multilevel-feedback-queue scheduler is defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine when to promote a process
  - Method used to determine when to demote a process
  - Method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

- Scheduling
  - A new job enters queue Q_0 *(RR)* and is placed at the end. When it gains the CPU, the job receives 8 milliseconds. If it does not finish in 8 milliseconds, the job is moved to the end of queue Q_1.
  - A Q_1 (RR) job receives 16 milliseconds. If it still does not complete, it is preempted and moved to queue Q_2 (FCFS).



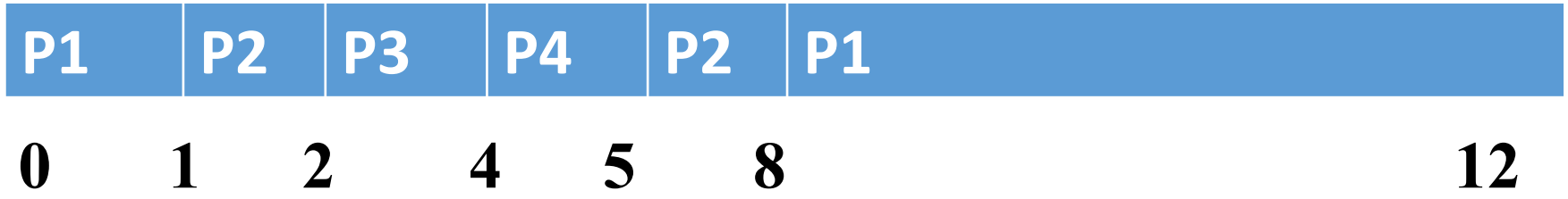
Priority Scheduling Preemptive

Process	Arrival time	Burst Time	Priority
P1	0	5:4	10
P2 1	4 :3	20	
P3	2	2 :1	30
P4	4	1	40

Higher the number implies higher priority

- The Gantt chart is:

Process	ARRIVAL Time	Burst Time	Completion time	TAT	WT
P1	0	5	12	12	7
P2	1	4	8	7	3
P3	2	2	4	2	0
P4	4	1	5	1	0



Average turnaround time=(12+7+2+1)/4=5.5ms

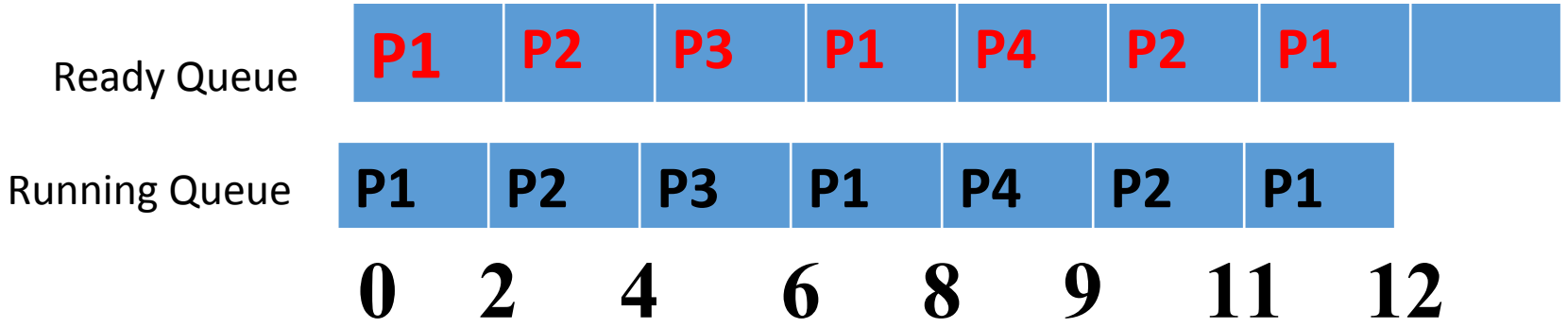
Average waiting time=(7+3+0+0)/4=2.5ms

Example of RR with Time Quantum = 2 ms

Process	Arrival time	Burst Time in ms
P_1	0	5:3:1
P_2	1	4:2:0
P_3	2	2:0
P4	4	1:0

Process	ARRIVAL Time	Burst Time	Completion time	TAT	WT
P1	0	5	12	12	7
P2	1	4	11	10	6
P3	2	2	6	4	2
P4	4	1	9	5	4

- The Gantt Chart for the schedule is:



- Find
- **Average turnaround time**
- **Average waiting time**

- Q1 Consider the following set of processes, with the length of the CPU burst given in milliseconds:
- Process Burst Time Priority
- P1 5 2
- P2 10 1
- P3 7 3
- P4 12 3
- P5 6 4
- The processes are assumed to have arrived in the order P1, P2, P3, P4, P5, all at time 0.
- a. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, nonpreemptive priority (a smaller priority number implies a higher priority), and RR (quantum = 2).
- b. What is the turnaround time of each process for each of the scheduling algorithms in part a?
- c. What is the waiting time of each process for each of the scheduling algorithms?
- d. Which of the algorithms results in the minimal average waiting time (over all processes)?

- **Q2** Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time	Priority
P1	0	40	3
P2	15	55	2
P3	15	35	1
P4	35	10	5
P5	40	30	4

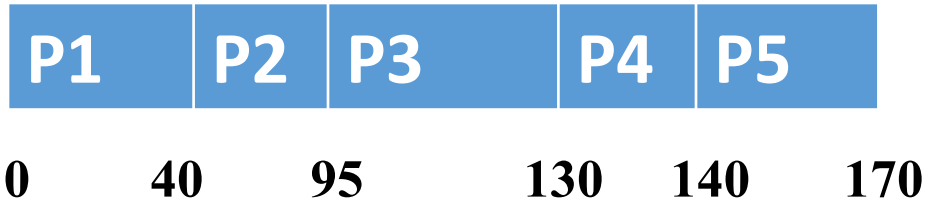
- The processes are assumed to have arrived in the order P1, P2,P3, P4, P5.
- a. Draw Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, SRTF, Preemptive Priority (a smaller priority number implies a higher priority) and RR (time quantam=10)
- b. What is the average turnaround time of each process for each of the scheduling algorithms in part a?
- c. What is the average waiting time of each process for each of these scheduling algorithm?

Solution Q2 FCFS

Process	Arrival Time	Burst Time	Priority
P1	0	40	3
P2	15	55	2
P3	15	35	1
P4	35	10	5
P5	40	30	4

Process	ARRIVAL Time	Burst Time	Completion time	TAT	WT
P1	0	40	40	40	0
P2	15	55	95	80	25
P3	15	35	130	115	80
P4	35	10	140	105	95
P5	40	30	170	130	100

The Gantt chart is:



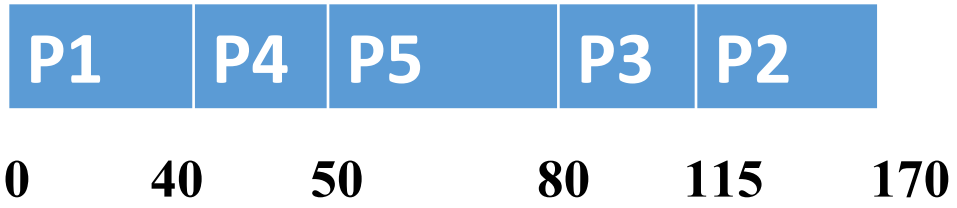
Average turnaround time=(40+80+115+105+130)/5=94

Average waiting

SJF

Process	Arrival Time	Burst Time	Priority
P1	0	40	3
P2	15	55	2
P3	15	35	1
P4	35	10	5
P5	40	30	4

The Gantt chart is:



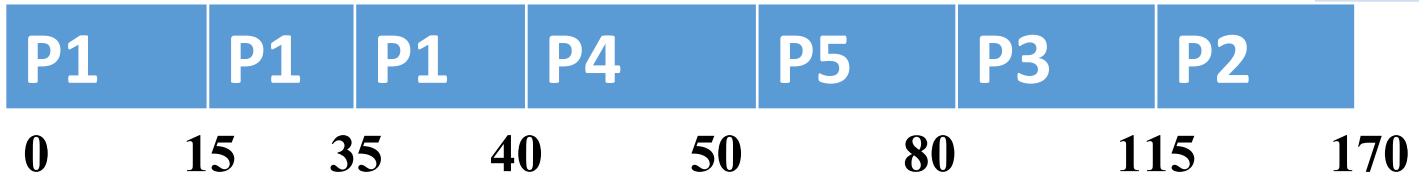
- Find
- **Average turnaround time**
- **Average waiting time**

Process	ARRIVAL Time	Burst Time	Completion time	TAT	WT
P1	0	40	40	40	0
P2	15	55	170	155	100
P3	15	35	115	100	65
P4	35	10	50	15	5
P5	40	30	80	40	10

SRTF (PREEMPTIVE SJF)

Process	Arrival Time	Burst Time	Priority
P1	0	40	3
P2	15	55	2
P3	15	35	1
P4	35	10	5
P5	40	30	4

The Gantt chart is:



Process	ARRIVAL Time	Burst Time	Completion time	TAT	WT
P1	0	40	40	40	0
P2	15	55	170	155	100
P3	15	35	115	100	65
P4	35	10	50	15	5
P5	40	30	80	40	10

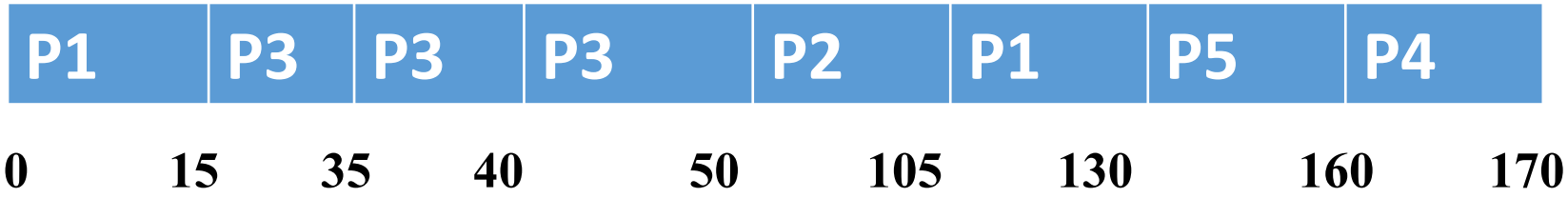
Find
Average turnaround time
Average waiting time

PRIORITY

Process	Arrival Time	Burst Time	Priority
P1	0	40	3
P2	15	55	2
P3	15	35	1
P4	35	10	5
P5	40	30	4

Process	ARRIVAL Time	Burst Time	Completion time	TAT	WT
P1	0	40:25			
P2	15	55			
P3	15	35:15:10			
P4	35	10			
P5	40	30			

The Gantt chart is:



Find

Average turnaround time

Average waiting time

RR QUANTUM=10

Process	Arrival Time	Burst Time	Priority
P1	0	40	3
P2	15	55	2
P3	15	35	1
P4	35	10	5
P5	40	30	4

Find

Average turnaround time

Average waiting time

Process	ARRIVAL Time	Burst Time	Completion time	TAT	WT
P1	0	40:30:20:10:0	100	100	60
P2	15	55:45:35:25:15:5:0	170	155	100
P3	15	35:25:15:5:0	155	140	105
P4	35	10:0	70	35	25
P5	40	30:20:10	150	110	80

Ready Queue

P1	P1	P2	P3	P1	P2	P4	P5	P3	P1	P2	P5	P3	P2	P5	P3	P2	P2

Running Queue

P1	P1	P2	P3	P1	P2	P4	P5	P3	P1	P2	P5	P3	P2	P5	P3	P2	P2

0 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 155 165 170

- Consider following five process with given CPU Burst Time and Quantum 10

Process	Burst Time	Priority
P1	10	4
P2	29	1
P3	3	2
P4	7	2
P5	12	3

- Calculate Average Waiting Time and Average Turn Around Time for SJF and priority algorithm. State which one is Optimal.

PROCESS SYNCHRONIZATION

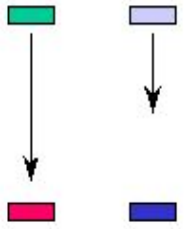
- **<u>Section-II</u>** ***(Weightage – 70%)***
  Process - Introduction, Threads, CPU Scheduling algorithms, Inter-process communication, Critical section problem, Semaphores, Classical process coordination problem.
- **Deadlock -**Definition, Necessary and sufficient conditions for Deadlock, Deadlock Prevention, Deadlock Avoidance: Banker's algorithm, Deadlock detection and Recovery.
- **Memory Management –** Concept of Fragmentation**,** Swapping, Paging, Segmentation.
- **Virtual memory**-Demand Paging, Page replacement algorithm, Thrashing.

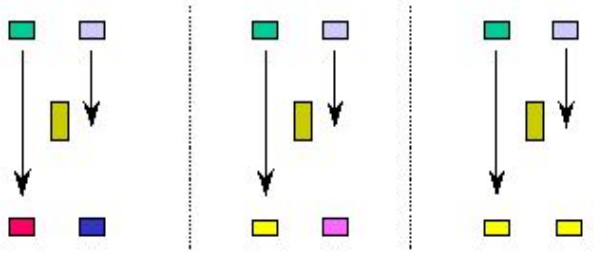
Synchronization

- A cooperating process is one that can affect or be affected by other processes executing in the system.
- Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages.
- Concurrent access to shared data may result in data inconsistency.
- In this section, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

Independent vs. Cooperating Processes

- Independent
  - no shared state
  - deterministic
  - each process can proceed at arbitrary rate
- Cooperating
  - Shared state
    - like a common variable
  - non-deterministic execution
    - hard to reproduce, and subject to race conditions





- Let's return to our consideration of the bounded buffer.
- As we pointed out, our original solution allowed at most BUFFER_SIZE - 1 items in the buffer at the same time. Suppose we want to modify the algorithm to remedy this deficiency. One possibility is to add an integer variable counter, initialized to 0.
- counter is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer.
- The code for the producer process can be modified as follows:

Synchronization

PRODUCER

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

CONSUMER

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

Assumptions

Bounded Buffer

counter : shared variable

Provide number of items in a buffer

count=0

count++: increase

- The producer-consumer problem is an example of a multi-process synchronization problem.
- The problem describes two processes, the producer and the consumer that shares a common fixed-size buffer use it as a queue.

- The producer's job is to generate data, put it into the buffer, and start again.
- At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time.

- Problem: Given the common fixed-size buffer, the task is to make sure that the producer can't add data into the buffer when it is full and the consumer can't remove data from an empty buffer.

- Solution: The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same manner, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

Synchronization

- Although both the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently.
- As an illustration, suppose that the value of the variable counter is currently 5 and that the producer and consumer processes execute the statements "counter++" and "counter--" concurrently.
- Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6!
- The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately.
- We can show that the value of counter may be incorrect as follows.
- The statement "counter++" may be implemented in machine language (on a typical machine) as
- *register1* = counter
  register1= *register1*+ 1
  counter= *register1*
  where *register1* is one of the local CPU registers.
- Similarly, the statement *register2"counter--"* is implemented as follows:
  register2 = counter
  register2= *register2* -1
  counter= *register2*
- where again *register2* is one of the local CPU registers. Even though *register1*and *register2* may be the same physical register (an accumulator, say), remember that the contents of this register will be saved and restored by the interrupt handler.

Synchronization

- The concurrent execution of "counter++" and "counter--" is equivalent to a sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order (but the order within each high-level statement is preserved). One such interleaving is

T0: *producer* execute *register1* =counter *{register1* = 5}

T1: *producer* execute *register1* = *register1* + 1 *{register1* = 6}

T2: *consumer* execute *register2* = counter *{register2* = 5}

T3: *consumer* execute *register2* = *register2*- 1 *{register2* = 4}

T4: *producer* execute counter= *register1 {counter* = 6}

T5: *consumer* execute counter = *register2 {counter* = 4}

Race Condition

- We have arrived at the incorrect state "counter == 4", indicating that four buffers are full, when, in fact, five buffers are full.
- If we reversed the order of the statements at T4 and *T5* , we would arrive at the incorrect state "counter== 6".
- We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently.
- A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition.
- To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter.
- To make such a guarantee, we require that the processes be synchronized in some way.

Producer

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

Consumer

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

Counter++

$register_1 = counter$

$register_1 = register_1 + 1$

$counter = register_1$

Counter--

$register_2 = counter$

$register_2 = register_2 - 1$

$counter = register_2$

T_0: *producer* $register_1 = counter$

T_1: *producer* $register_1 = register_1 + 1$

T_2: *consumer* $register_2 = counter$

T_3: *consumer* $register_2 = register_2 - 1$

T_4: *producer* $counter = register_1$

T_5: *consumer* $counter = register_2$

Result is 4,5,6
Problem because of interleaving instruction

Race Condition Problems

- Produce unpredictable results
- Create inconsistency

- Accessing joint account (Withdraw & deposit amount in account)

- Multithreaded applications running in parallel on **multi-core systems** and sharing data

- **Need of co-operation/synchronization**

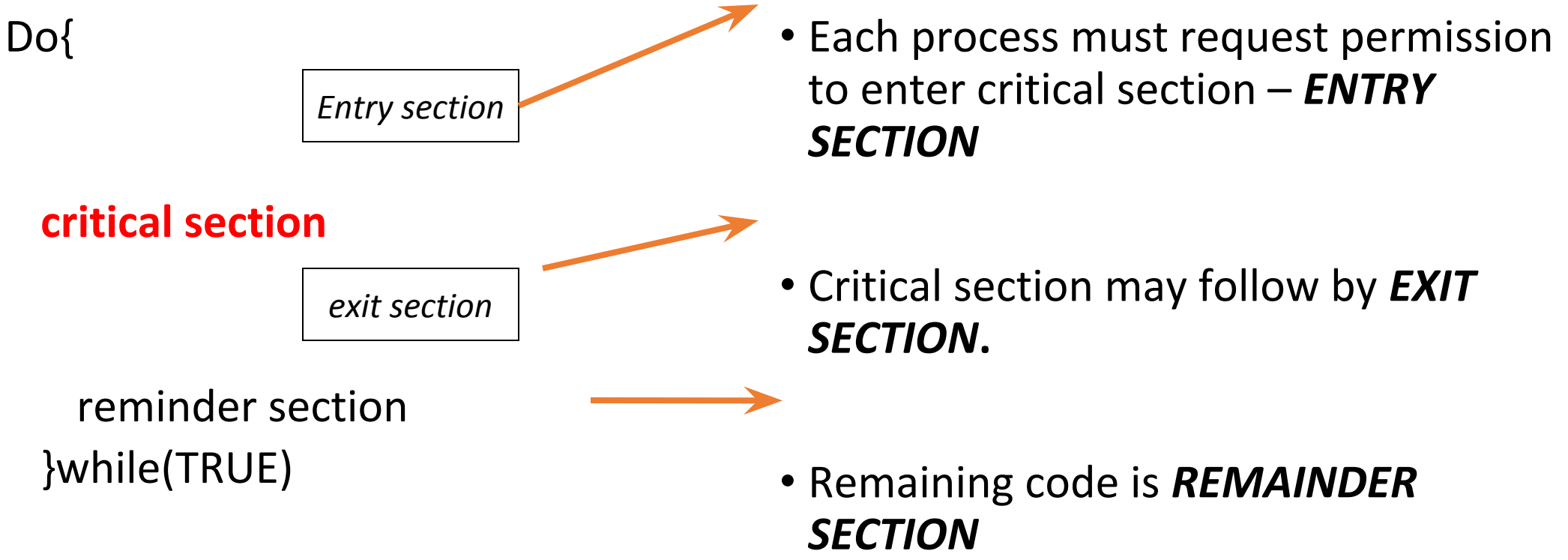
CRITICAL SECTION

- Process Synchronization controls the execution of processes running concurrently so as to produce the consistent results.
- Critical section is a part of the program where shared resources are accessed by the process.
- Critical Section Problem-
- If multiple processes access the critical section concurrently, then results produced might be inconsistent.
- This problem is called as critical section problem.
- Synchronization mechanisms allow the processes to access critical section in a synchronized manner to avoid the inconsistent results.

CRITICAL SECTION

- Consider a system consisting of n processes {Po, P1 , ... , P_{n-1} }.
- Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on.
- The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time.
- The critical-section problem is to design a protocol that the processes can use to cooperate.
- Each process must request permission to enter its critical section. The section of code implementing this request is the entry section.
  - It acts as a gateway for a process to enter inside the critical section.
  - It ensures that only one process is present inside the critical section at any time.
  - It does not allow any other process to enter inside the critical section if one process is already present inside it.

CRITICAL SECTION ENVIRONMENT



GENERAL STRUCTURE OF A TYPICAL PROCESS Pi

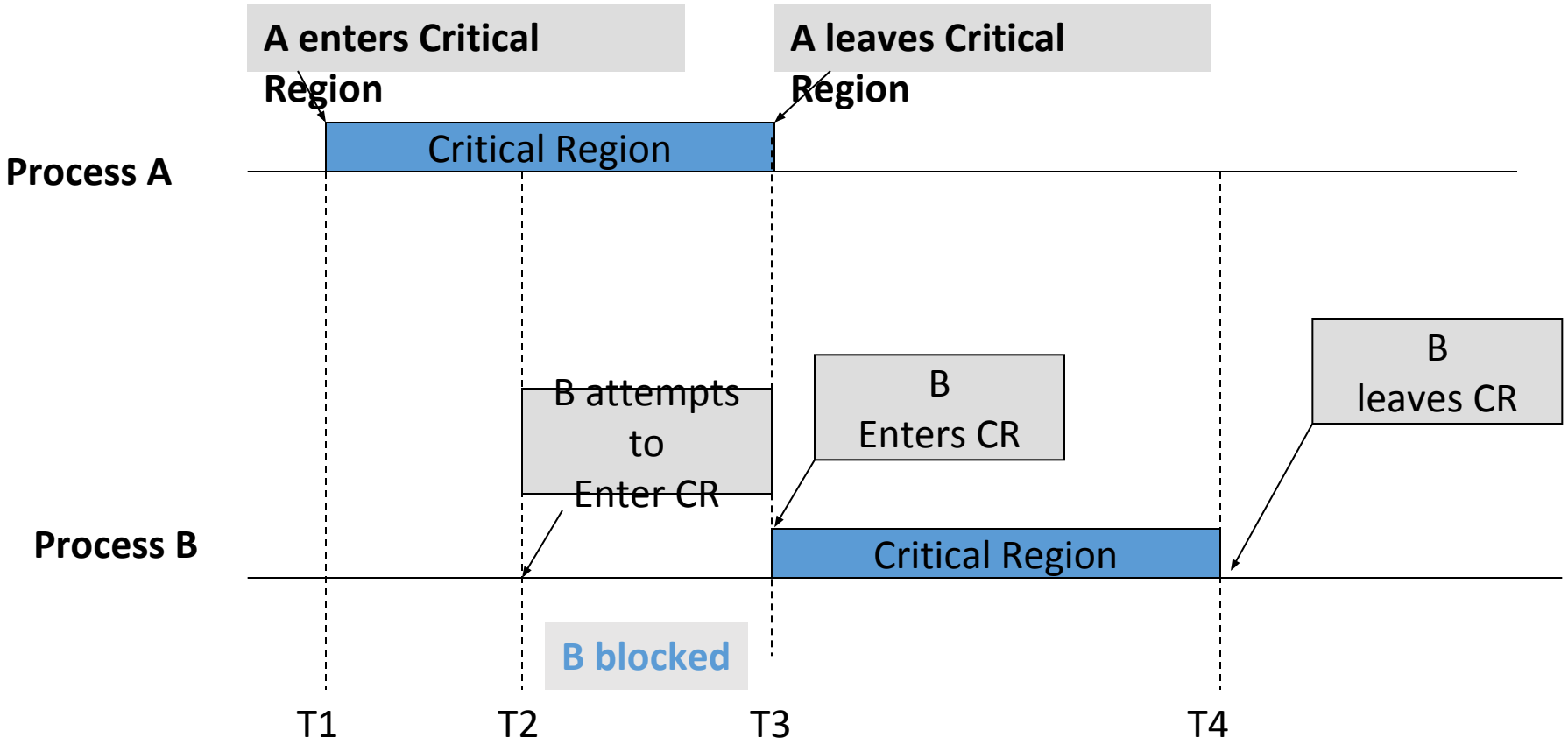
CRITICAL SECTION

- The critical section may be followed by an exit section.
  - It acts as an exit gate for a process to leave the critical section.
  - When a process takes exit from the critical section, some changes are made so that other processes can enter inside the critical section.
- The remaining code is the remainder section. The general structure of a typical process Pi is shown in Figure 6.1. The entry section and exit section are enclosed in boxes to highlight these important segments of code.
- A solution to the critical-section problem must satisfy the following three requirements:

Solution to the Critical Section Problem must meet three conditions...

- **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections (disabling interrupt,lock variables,algo)
- **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter into its critical section and this selection can not be postponed indefinitely.
- **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Mutual Exclusion using Critical Regions



Critical Section Solutions

- Race condition occurs in CS. So protection is needed to CS
- s/w solution
  - Peterson’s solution (two process solution)
  - Semaphore (n process solution)
- h/w solution
  - TestAndSet
  - Swap

Peterson Solution

- A classic software-based solution to the critical-section problem known as Peterson's solution.
- Because of the way modern computer architectures perform basic machine-language instructions, such as load and store, there are no guarantees that Peterson's solution will work correctly on such architectures. However we present the solution because it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.
- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered Po and P1. For convenience, when presenting Pi, we use Pj to denote the other process; that is, j equals 1 - i.
- The eventual value of turn determines which of the two processes is allowed to enter its critical section first.
- We now prove that this solution is correct. We need to show that:

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

TWO-PROCESS SOLUTION

-

- Meets all three requirements;
- solves the critical-section problem for two processes .
  - *Mutual exclusion*
  - *Progress*
    - *No strictness. Pi enter into CS till Pj is not interested*
    - Turn variable breaks any deadlock possibility
  - Bounded waiting
    - Initially Pi in CS
    - Then If Pj try Pj has to wait
    - Then if P0 tries again it has to wait
    - Then if P1 try it can enter

Synchronization Hardware

- Software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures.
- Instead, we can generally state that any solution to the critical-section problem requires a simple tool-a lock.
- Race conditions are prevented by requiring that critical regions be protected by locks. That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section.
- This is illu

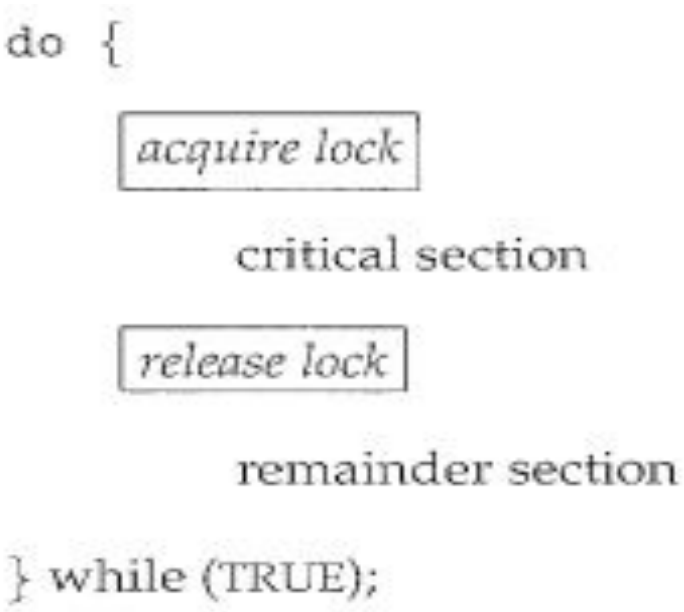


Figure 6.3 Solution to the critical-section problem using locks.

Synchronization Hardware

- Hardware features can make any programming task easier and improve system efficiency.
- The critical-section problem could be solved simply in a uniprocessor environment if we could prevent interrupts from occurring while a shared variable was being modified.
- In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption.
- No other instructions would be run, so no unexpected modifications could be made to the shared variable. This is often the approach taken by nonpreemptive kernels.
- Unfortunately, this solution is not as feasible in a multiprocessor environment.
- Disabling interrupts on a multiprocessor can be time consuming, as the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases. Also consider the effect on a system's clock if the clock is kept updated by interrupts.
- Many modern computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words automatically that is, as one uninterruptible unit.
- We can use these special instructions to solve the critical-section problem in a relatively simple manner.
- Rather than discussing one specific instruction for one specific machine, we abstract the main concepts behind these types of instructions by describing the TestAndSet () and Swap() instructions.

Synchronization Hardware

TestAndSet () instruction

- The TestAndSet () instruction can be defined as shown in Figure 6.4.
- This instruction is executed atomically. Thus, if two TestAndSet () instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order. If the machine supports the TestAndSet () instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false.
- The structure of process P; is shown in Figure 6.5.
- Process P0 arrives.
- It executes the test-and-set(Lock) instruction.
- Since lock value is set to 0, so it returns value 0 to the while loop and sets the lock value to 1.
- The returned value 0 breaks the while loop condition.
- Process P0 enters the critical section and executes.
- Now, even if process P0 gets preempted in the middle, no other process can enter the critical section.
- Any other process can enter only after process P0 completes and sets the lock value to 0.

```
boolean TestAndSet(boolean *target)
{
boolean rv = *target;
*target = TRUE;
return rv;
}
```

Figure 6.4 The definition of the TestAndSet () instruction.

```
do {
while (TestAndSet(&lock))
; //do nothing
//critical section
lock = FALSE;
// remainder section
} while (TRUE);
```

Figure 6.5 Mutual-exclusion implementation with TestAndSet ().

Synchronization Hardware

TestAndSet () instruction

- Here, the shared variable is lock which is initialized to false.
- TestAndSet(lock) algorithm works in this way – it always returns whatever value is sent to it and sets lock to true.
- The first process will enter the critical section at once as TestAndSet(lock) will return false and it'll break out of the while loop.
- The other processes cannot enter now as lock is set to true and so the while loop continues to be true. Mutual exclusion is ensured.
- Once the first process gets out of the critical section, lock is changed to false. So, now the other processes can enter one by one.
- Progress is also ensured.
- However, after the first process any process can go in.
- There is no queue maintained, so any new process that finds the lock to be false again, can enter.
- So bounded waiting is not ensured.

```
boolean TestAndSet(boolean *target)
{
boolean rv = *target;
*target = TRUE;
return rv;
}
```

Figure 6.4 The definition of the TestAndSet () instruction.

```
do {
while (TestAndSet(&lock))
; //do nothing
//critical section
lock = FALSE;
// remainder section
} while (TRUE);
```

Figure 6.5 Mutual-exclusion implementation with TestAndSet ().

Synchronization Hardware

Swap() instruction

- The Swap() instruction, in contrast to the TestAndSet () instruction, operates on the contents of two words; it is defined as shown in Figure 6.6.
- Like the TestAndSet () instruction, it is executed atomically.
- If the machine supports the Swap() instruction, then mutual exclusion can be provided as follows.
- A global Boolean variable lock is declared and is initialized to false.
- In addition, each process has a local Boolean variable key. The structure of process P; is shown in Figure 6.7.
- Although these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement.
- TestAndSet & Swap instructions satisfy Mutual Exclusion but not bounded waiting

```
void Swap(boolean *a, boolean *b)
{
boolean temp = *a;
*a =*b;
*b = temp;
}
```

Figure 6.6 The definition of the Swap () instruction.

```
do {
key = TRUE;
while (key == TRUE)
Swap(&lock, &key);
// critical section
lock = FALSE;
//remainder section
} while (TRUE);
```

Figure 6.7 Mutual-exclusion implementation with the Swap() instruction.

Synchronization Hardware

Bounded-waiting mutual exclusion with TestAndSet ()

- In Figure 6.8, we present another algorithm using the TestAndSet () instruction that satisfies all the critical-section requirements. The common data structures are
- boolean waiting[n];
- boolean lock;
- These data structures are initialized to false.
- To prove that the mutual exclusion requirement is met, we note that process P; can enter its critical section only if either waiting [i] == false or key == false.
- The value of key can become false only if the TestAndSet () is executed.
- The first process to execute the TestAndSet () will find key== false; all others must wait.
- The variable waiting [i] can become false only if another process leaves its critical section; only one waiting [i] is set to false,

```
do {
waiting[i] = TRUE;
key = TRUE;
while (waiting[i] && key)
key= TestAndSet(&lock);
waiting[i] = FALSE;
//critical section
j = (i + 1) % n;
while ((j != i) && !waiting[j])
j = (j + 1) % n;
if (j == i)
lock = FALSE;
else
waiting[j] = FALSE;
// remainder section
} while (TRUE) ;
```

Figure 6.8 Bounded-waiting mutual exclusion with TestAndSet ().

```
do {
waiting[i] = TRUE;
key = TRUE;
while (waiting[i] && key)
key= TestAndSet(&lock);
waiting[i] = FALSE;
//critical section
j = (i + 1) % n;
while ((j != i) && !waiting[j])
j = (j + 1) % n;
if (j == i)
lock = FALSE;
else
waiting[j] = FALSE;
// remainder section
} while (TRUE) ;
```

Figure 6.8 Bounded-waiting mutual exclusion with TestAndSet ().

N=NUMBER OF PROCESSES

Process P2 wants to enter in CS. According to algorithm , waiting[2]=TRUE and Key=TRUE.
Then it can proceed.

waiting	P 0	P 1	P2	P3	...	Pn
key	F	F	F	F	F	F

waiting [i]	P0	P1	P2	P3	P4	
key	F	F	F	F	F	

I	J
2	3
2	0
2	1
2	2

Bounded-waiting mutual exclusion with TestAndSet ()

- To prove that the progress requirement is met, we note that the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets lock to false or sets waiting[j] to false.
- Both allow a process that is waiting to enter its critical section to proceed.
- To prove that the bounded-waiting requirement is met, we note that, when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i + 1, i + 2, ... , n -1, 0, ... , i- 1).
- It designates the first process in this ordering that is in the entry section (waiting[j] ==true) as the next one to enter the critical section.
- Any process waiting to enter its critical section will thus do so within n - 1 turns.

Bounded-waiting mutual exclusion with TestAndSet (

- Unlock and Lock Algorithm uses TestAndSet to regulate the value of lock but it adds another value, waiting[i], for each process which checks whether or not a process has been waiting.
- A ready queue is maintained with respect to the process in the critical section.
- All the processes coming in next are added to the ready queue with respect to their process number, not necessarily sequentially.
- Once the ith process gets out of the critical section, it does not turn lock to false so that any process can avail the critical section now, which was the problem with the previous algorithms.
- Instead, it checks if there is any process waiting in the queue.
- The queue is taken to be a circular queue.
- j is considered to be the next process in line and the while loop checks from jth process to the last process and again from 0 to (i-1)th process if there is any process waiting to access the critical section.
- If there is no process waiting then the lock value is changed to false and any process which comes next can enter the critical section. If there is, then that process' waiting value is turned to false, so that the first while loop becomes false and it can enter the critical section.
- This ensures bounded waiting. So the problem of process synchronization can be solved through this algorithm

Hardware Synchronization

- Advantages
  - Applicable to any number of processes on either a single processor or multiple processors sharing main memory
  - Simple and therefore easy to verify
  - Can be used to support multiple critical section; each CS has its own variable
- Disadvantages
  - Busy waiting is employed, consumes processor time
  - Starvation is possible since selection of the next process to run waiting on that variable is arbitrary
  - Deadlock is possible on a single processor machine – P1 executes special instruction and enters CS. P1 is interrupted and P2, which has a higher priority, gets the CPU. P2 can't execute its CS since P1 is still in its CS – P2 goes into a busy waiting loop. P1,however, will not be dispatched because it has a lower priority than another ready process, P2.

SEMAPHORE: A synchronization tool

- The hardware-based solutions to the critical-section problem presented are complicated for application programmers to use.
- To overcome this difficulty, we can use a synchronization tool called a SEMAPHORE.

Semaphores

- Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.
- The definitions of wait and signal are as follows −
- Wait

```
wait(S)
{
  while (S<=0);    //no op
  S--;
}
```

Signal

The signal operation increments the value of its argument S.

```
signal(S)
{
  S++;
}
```

- The wait operation decrements the value of its argument S, if it is positive.
- If S is negative or zero, then no operation is performed.

All modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly.

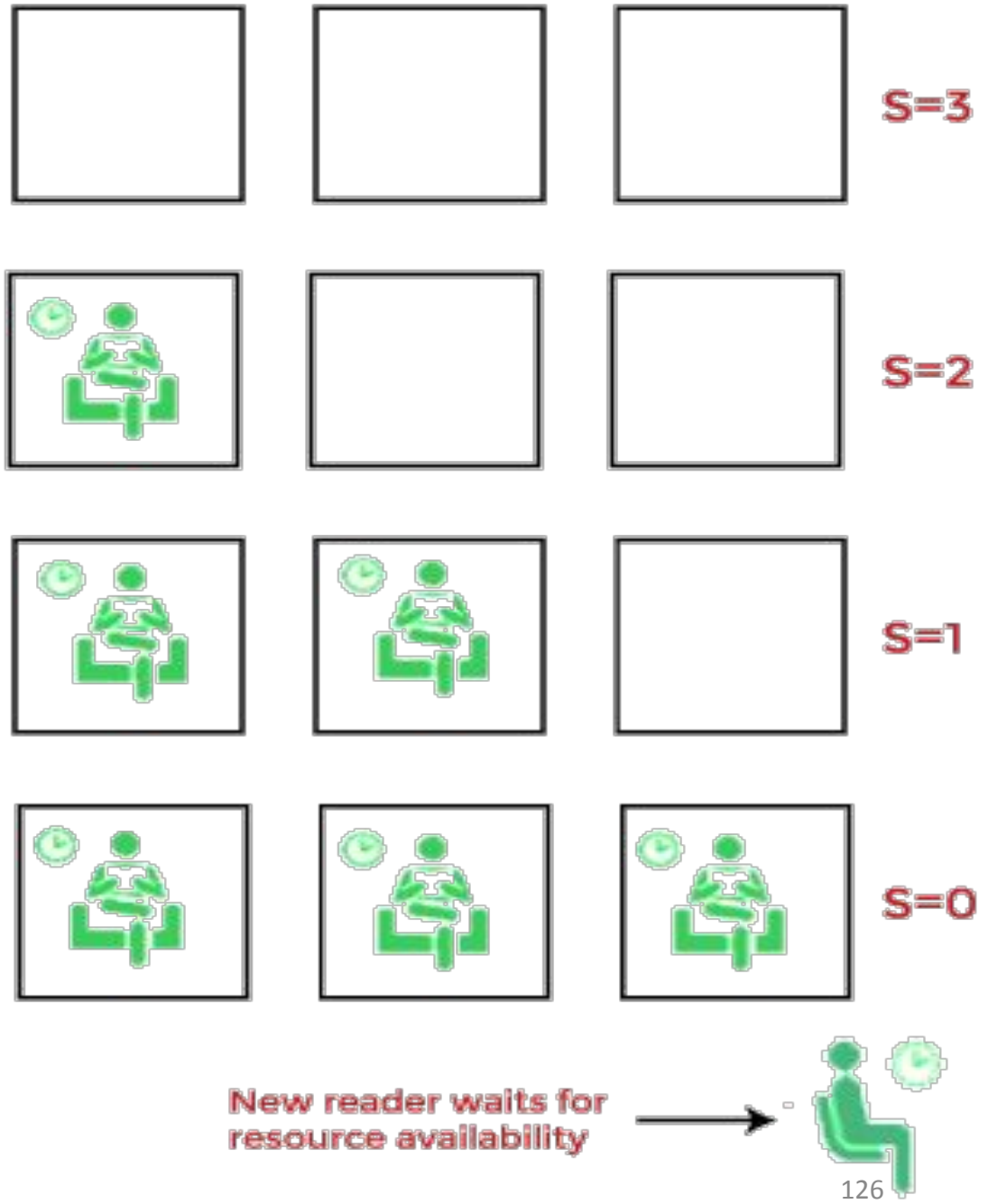
That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of wait (S), the testing of the integer value of S (S<=0), as well as its possible modification (S--), must be executed without interruption.

Characteristic of Semaphore

- Semaphore carries a non-negative integer value always.
- We use semaphore mechanism to offer synchronization of tasks.
- It is a low-level synchronization mechanism.

Usage:Types of Semaphores

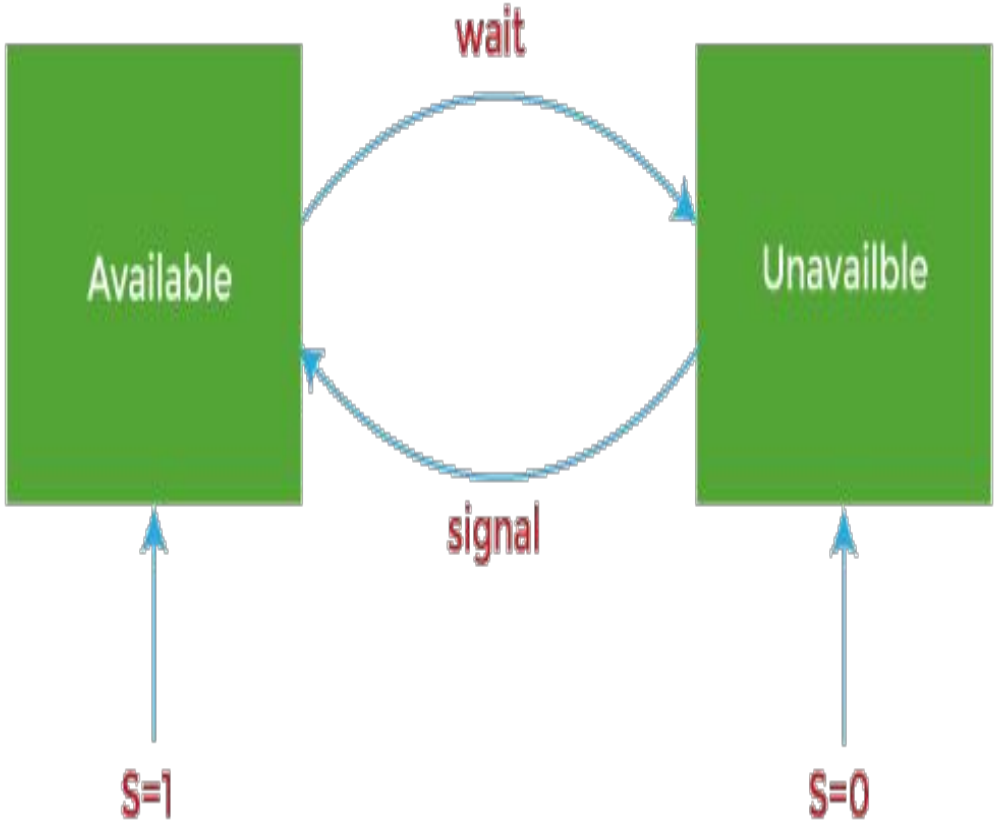
- There are two main types of semaphores i.e. counting semaphores and binary semaphores. Details about these are given as follows −
- 1. Counting Semaphores
- The value of a counting semaphore can range over an unrestricted domain.
- These are integer value semaphores . These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.



Usage:Types of Semaphores

2. Binary Semaphores

The binary semaphores are like counting semaphores. The value of a binary semaphore can range only between 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores.



Usage:

- On some systems, binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion.
- We can use binary semaphores to deal with the critical-section problem for multiple processes. Then processes share a semaphore, mutex, initialized to 1.
- Each process Pi is organized as shown in Figure 6.9.
- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
- The semaphore is initialized to the number of resources available.
- Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count).
- When a process releases a resource, it performs a signal() operation (incrementing the count).
- When the count for the semaphore goes to 0, all resources are being used.
- After that, processes that wish to use a resource will block until the count becomes greater than 0.

```
do {

wait (mutex) ;

// critical section

signal(mutex);

//remainder section

} while (TRUE);
```

Figure 6.9 Mutual-exclusion implementation with semaphores.

We can also use semaphores to solve various synchronization problems.
For example, consider two concurrently running processes: P1 with a statement s1 and P2 with a statement s2 .
Suppose we require that s2 be executed only after s1 has completed.
We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0, and by inserting the statements

```
S1;
signal(synch) ;
```

in process P1 and the statements

```
wait(synch);
S2;
```

in process P2. Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal (synch), which is after statement S1 has been executed.

Implementation

- The main disadvantage of the semaphore definition given here is that it requires busy waiting .
- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.
- This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a spinlock because the process "spins" while waiting for the lock. (Spinlocks do have an advantage in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlocks are useful; they are often employed on multiprocessor systems where one thread can "spin" on one processor while another thread performs its critical section on another processor.)
- To overcome the need for busy waiting, we can modify the definition of the wait() and signal() semaphore operations.
- When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

Implementation

- A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation.
- The process is restarted by a wakeup () operation, which changes the process from the waiting state to the ready state.
- The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)
- To implement semaphores under this definition, we define a semaphore as
- a "C' struct:
- typedef struct {
- int value;
- struct process *list;
- } semaphore;

Implementation

- Each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes. A signal() operation removes one process from the list of waiting processes and awakens that process.
- The wait() semaphore operation can now be defined as

```
wait(semaphore *S) {
S->value--;
if (S->value < 0) {
add this process to S->list;
block();
}
}
```

- The signal () semaphore operation can now be defined as

```
signal(semaphore *S) {
S->value++;
if (S->value <= 0) {
remove a process P from< S->list;
wakeup(P);
}
}
```

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        sleep();
    }
}
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
     remove a process P from S->list;
        wakeup(P);
    }
}
```

- **Sleep( )** – suspends the process that invokes it and places the process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state
- **Wakeup(P)** – change the process from waiting state to ready state and place it in the ready queue
- Wait()/signal() are critical sections, they must be executed atomically
- Busy waiting is limited only to the critical sections of **wait** and **signal** operations, and these are short

Implementation

- The block() operation suspends the process that invokes it.
- The wakeup(P) operation resumes the execution of a blocked process P.
- These two operations are provided by the operating system as basic system calls.
- In this implementation, semaphore values may be negative, although semaphore values are never negative under the classical definition of semaphores with busy waiting.
- If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore.
- This fact results from switching the order of the decrement and the test in the implementation of the wait () operation. The list of waiting processes can be easily implemented by a link field in each process control block (PCB).
- Each semaphore contains an integer value and a pointer to a list of PCBs. One way to add and remove processes from the list so as to ensure bounded waiting is to use a FIFO queue, where the semaphore contains both head and tail pointers to the queue. It is critical that semaphores be executed atomically.
- We must guarantee that no two processes can execute wait() and signal() operations on the same semaphore at the same time.
- It is important to admit that we have not completely eliminated busy waiting with this definition of the wait () and signal () operations. Rather, we have moved busy waiting from the entry section to the critical sections of application programs. Furthermore, we have limited busy waiting to the critical sections of the wait () and signal () opera times, and these sections are short (if properly coded, they sbould be no more than about ten instructions).

Advantages and Disadvantages of Semaphores

- **Advantages of Semaphore**
- In the Semaphore, only one process is allowed to enter into the critical section. In this, the principle of mutual exclusion is to be followed strictly. And the semaphore mechanism is a more efficient mechanism than other methods which we use in process synchronization.
- It is machine-independent because it is implemented in the microkernel's machine-independent code.
- With the help of the semaphore, the resources are managed flexibly.
- In semaphore, there is a busy waiting, so there is no wastage of resources and process time.
- In the semaphore, more threads are permitted to enter into the critical section.
- **Disadvantages of Semaphore**
- There is a priority inversion in the semaphore, and this is the biggest drawback of the semaphore.
- In the semaphore, due to program error violation of mutual exclusion, deadlock can be happened.
- In the semaphore, there are more chances of programmer errors.
- For the purpose of large-scale use, the semaphore is not a practical method, and due to this, there may be loss of modularity.
- The Programming of the semaphore is tough, so there may be a possibility of not achieving mutual exclusion.
- In the semaphore, the OS has to preserve all calls to wait and signal semaphore.

Counting Semaphore vs. Binary Semaphore

Counting Semaphore	Binary Semaphore
In counting semaphore, there is no mutual exclusion.	In binary semaphore, there is mutual exclusion.
In the counting semaphore, any integer value can be possible.	It contains only two integer values that are 1 and 0.
In counting semaphore, there are more than one slots.	In the binary semaphore, there is only one slot.
The counting process offers a set of processes.	Binary semaphore contains mutual exclusion mechanism.

Difference between Semaphore vs. Mutex

Parameter	Semaphore	Mutex
Data Type	It is an integer variable.	It is an object.
Mechanism	Semaphore is a signaling mechanism.	Mutex is a type of locking mechanism.
Types	There are two types of semaphore, which are binary semaphore and counting semaphore.	There are no types of mutex.
Operation	In semaphore, wait() and signal() operations are performed to modify the value of semaphore.	In mutex, locked or unlocked operation is performed.
Thread	In semaphore, there may be multiple program threads.	In mutex, there may also be a multiple program thread but not simultaneously.

Deadlocks and Starvation

- The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal() When such a state is reached, these processes are said to be deadlocked.
- To illustrate this, we consider a system consisting of two processes, Po and P1, each accessing two semaphores, S and Q, set to the value 1:

P0	P1

wait(S);
wait(Q);
.
.
signal(S);
signal(Q);
	wait(Q);
	wait(S);
	.
	.
	signal(Q);
	signal(S);

P0	P1
S=1,Q=1	
S=0	Q=0
Q=-1	S=-1

- Suppose that Po executes wait (S) and then P1 executes wait (Q).
- **When Po executes wait (Q), it must wait until P1 executes signal (Q).**
- **Similarly, when P1 executes wait (S), it must wait until Po executes signal(S).**
- Since these signal() operations cannot be executed, Po and P1 are deadlocked.
- We say that a set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set.
- The events with which we are mainly concerned here are resource acquisition and release.
- Another problem related to deadlocks is indefinite blocking or starvation, a situation in which processes wait indefinitely within the semaphore.
- Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

Priority Inversion

- A scheduling challenge arises when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process-or a chain of lower-priority processes.
- Since kernel data are typically protected with a lock, the higher-priority process will have to wait for a lower-priority one to finish with the resource.
- The situation becomes more complicated if the lower-priority process is preempted in favor of another process with a higher priority.
- As an example, assume we have three processes, L, M, and H, whose priorities follow the order L < M < H. Assume that process H requires resource R, which is currently being accessed by process L.
- Ordinarily, process H would wait for L to finish using resource R. However, now suppose that process M becomes runnable, thereby preempting process L. Indirectly, a process with a lower priority-process M-has affected how long process H must wait for L to relinquish resource R.
- This problem is known as Priority Inversion.
- It occurs only in systems with more than two priorities, so one solution is to have only two priorities.
- That is insufficient for most general-purpose operating systems, however. Typically these systems solve the problem by implementing a priority inheritance protocol. According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question.
- When they are finished, their priorities revert to their original values. In the example above, a priority-inheritance protocol would allow process L to temporarily inherit the priority of process H, thereby preventing process M from preempting its execution. When process L had finished using resource R, it would relinquish its inherited priority from Hand assume its original priority. Because resource R would now be available, process H-not M-would run next.

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

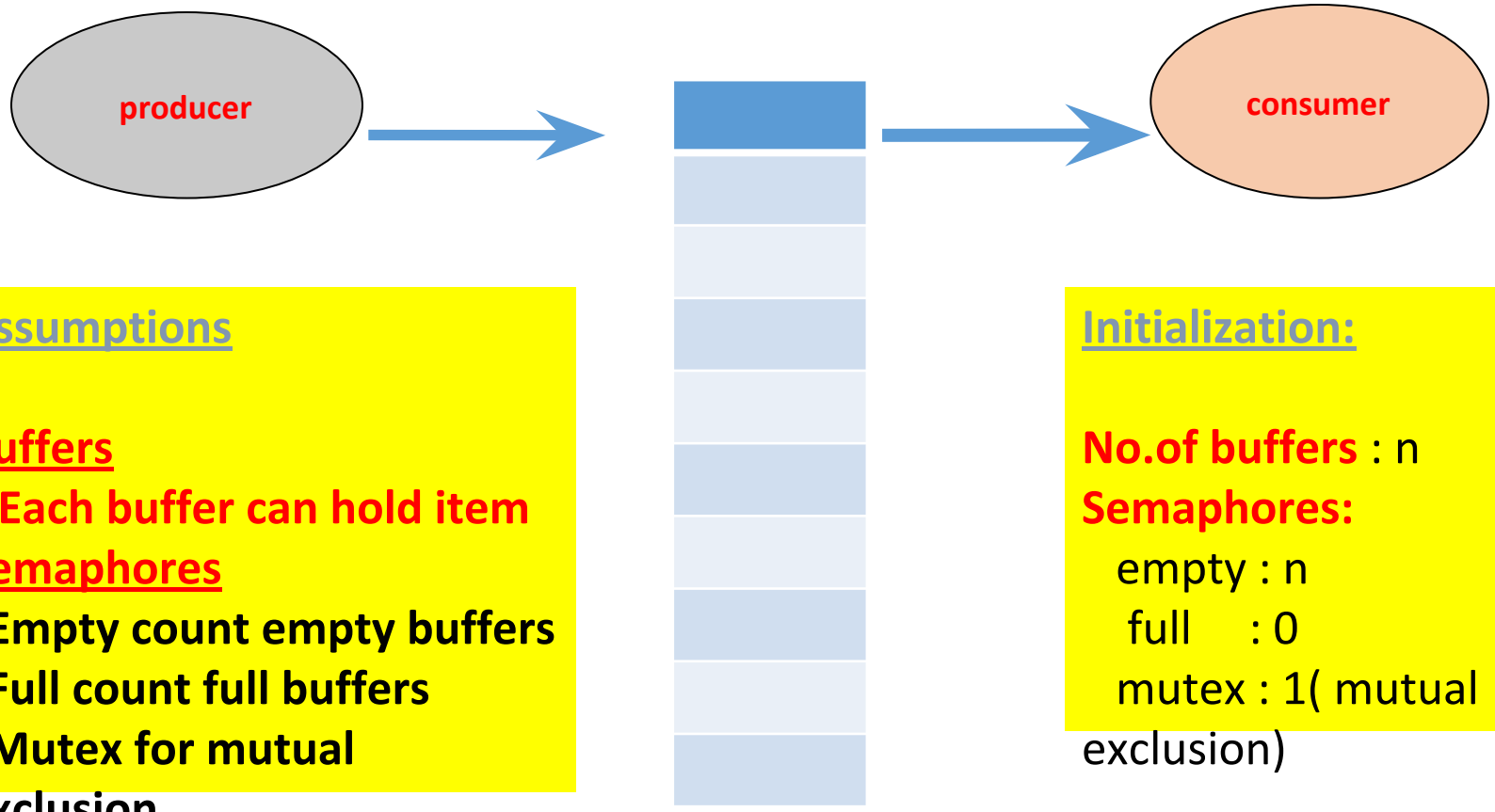
P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
  - Indefinite blocking may occur if we add and remove processes from the list associated with a semaphore in LIFO order.
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process ⇒ no progress!
  - Solved via **priority-inheritance protocol**

Classic Problems of Synchronization

- The Bounded-Buffer Problem
- The Readers-Writers Problem
- The Dining-Philosophers Problem

Classic Problem of Synchronization
The Bounded-Buffer Problem



Assumptions

Buffers

Each buffer can hold item

Semaphores

Empty count empty buffers

Full count full buffers

Mutex for mutual exclusion

Pool of n buffers

Initialization:

No.of buffers : n

Semaphores:

empty : n

full : 0

mutex : 1(mutual exclusion)

The Bounded-Buffer Problem

- The bounded-buffer problem is commonly used to illustrate the power of synchronization primitives.
- We assume that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores comct the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.
- The code for the producer process is shown in Figure 6.10; the code for the consumer process is shown in Figure 6.11.
- We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

The Bounded-Buffer Problem

Producer

	IN	OUT
EMPTY	5	4
MUTEX	1	0
MUTEX	0	1
FULL	0	1

buffers = n mutex=1 empty = n full = 0

Producer

```
do {  ……..
    // produce an item in nextp
      ……..
      wait(empty);
       wait(mutex);
       …….
       // add nextp to buffer
       …….
       signal(mutex);
       signal(full);
} while(True);
```

Consumer

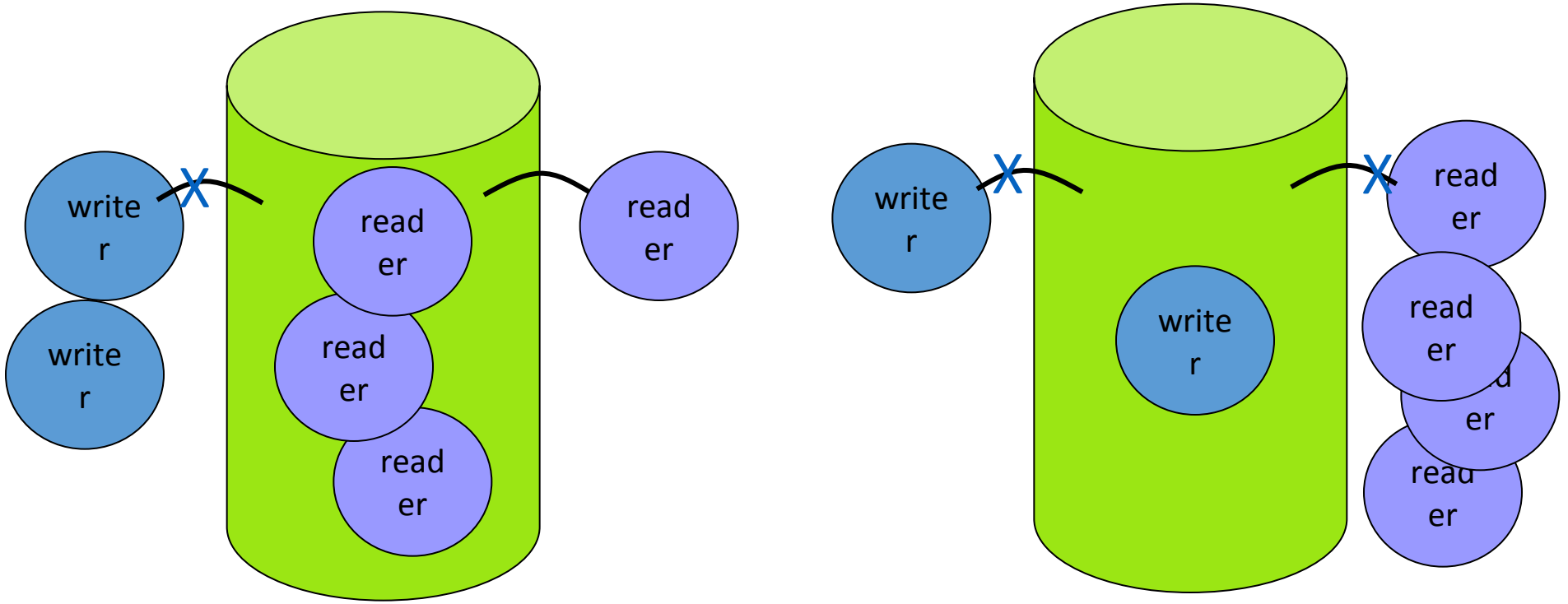
```
do {
       wait(full);
       wait(mutex);
       ……..
     // remove an item from
  buffer to nextc
       ……..
       signal(mutex);
       signal(empty);
       ……..
      // consume the item in nextc
} while(True);
```

Consumer

	IN	OUT
FULL	1	0
MUTEX	1	0
MUTEX	0	1
EMPTY	4	5

The Readers-Writers Problem

- Reader performs only read operation
- Writer can perform read or write or read+write operation
- Writers should have exclusive access to database



For reader / writer it is a section

The Readers-Writers Problem

- Suppose that a database is to be shared among several concurrent processes.
- Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.
- We distinguish between these two types of processes by referring to the former as readers and to the latter as writers. Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may occur.
- To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database.
- This synchronization problem is referred to as the readers-writers problem.

The Readers-Writers Problem

- The first readers-writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object.
- In other words, no reader should wait for other readers to finish simply because a writer is waiting.
- The second readers writers problem requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.
- A solution to either problem may result in starvation.
- In the first case, writers may starve; in the second case, readers may starve.
- For this reason, other variants of the problem have been proposed.
- Next, we present a solution to the first readers-writers problem.
- In the solution to the first readers-writers problem, the reader processes share the following data structures:
- semaphore mutex, wrt;
- int readcount;

The Readers-Writers Problem

- The semaphores mutex and wrt are initialized to 1; readcount is initialized to 0.
- The semaphore wrt is common to both reader and writer processes.
- The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated.
- The readcount variable keeps track of how many processes are currently reading the object.
- The semaphore wrt functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.
- Note that, if a writer is in the critical section and n readers are waiting, then one reader is queued on wrt, and n- 1 readers are queued on mutex.
- Also observe that, when a writer executes signal (wrt), we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.
- The readers-writers problem and its solutions have been generalized to provide locks on

The Readers-Writers Problem

- Acquiring a reader-writer lock requires specifying the mode of the lock either read or write access.
- When a process wishes only to read shared data, it requests the reader-writer lock in read mode; a process wishing to modify the shared data must request the lock in write mode. Multiple processes are permitted to concurrently acquire a reader-writer lock in read mode, but only one process may acquire the lock for writing, as exclusive access is required for writers.
- Reader-writer locks are most useful in the following situations:
- In applications where it is easy to identify which processes only read shared data and which processes only write shared data.
- In applications that have more readers than writers. This is because readerwriter locks generally require more overhead to establish than semaphores or mutual-exclusion locks.
- The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the readerwriter lock.

Structure of a reader process & writer process

	IN	OUT
W(MUTEX)	1	0
READCOUNT	0	1
W(WRT)	1	0
S(MUTEX)	0	1
W(MUTEX)	1	0
READCOUNT	1	0
S(WRT)	0	1
S(MUTEX)	0	1

wrt = 1 mutex = 1 readcount = 0

READER

```
do {
wait(mutex); //mutual exclusion for readers
    readcount++ ; // reader prcoess=1
    if (readcount==1)
      wait(wrt);  // don't allow writer
signal(mutex) ;  // other reader can enter while the current reader is in CS

     //reading is performed

wait(mutex);
    readcount - -; // reader wants to leave
    if (readcount==0)  //no reader is in CS
      signal(wrt);      // writer can enter
signal(mutex);
} while(TRUE);
```

WRITER

```
do {
        wait(wrt);

        // writing is performed

        signal(wrt);
}while (TRUE);
```

	IN	OUT
w(WRT)	1	0
s(WRT)	0	1

Dining philosopher problem

- Consider five philosophers who spend their lives thinking and eating.
- The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.
- In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (Figure 6.14).
- When a philosopher thinks, she does not interact with her colleagues.
- From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors).
- A philosopher may pick up only one chopstick at a time.
- Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor.
- When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.
- The dining-philosophers problem is considered a classic synchronization problem neither because of its practical importance nor because computer scientists dislike philosophers but because it is an example of a large class of concurrency-control problems.

Dining philosopher problem

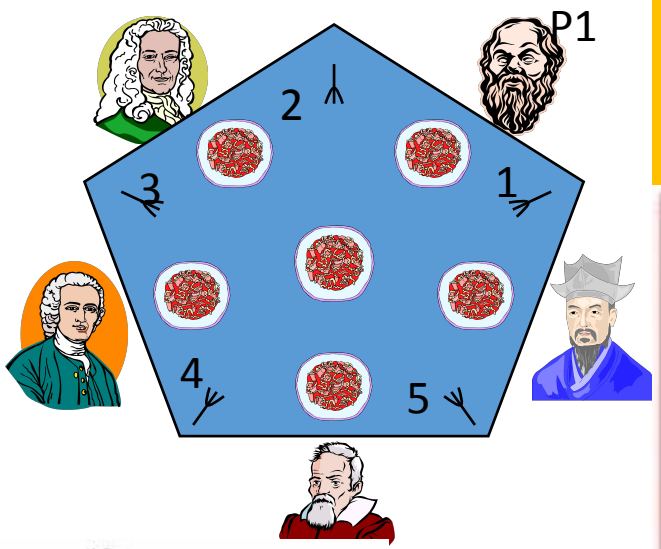
- It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner. One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing await () operation on that semaphore; she releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus, the shared data are
- semaphore chopstick[5];
- where all the elements of chopstick are initialized to 1.
- The structure of philosopher i is shown in Figure 6.15.
- Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock.
- Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick.
- All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Dining philosopher problem

- Several possible remedies to the deadlock problem are listed next.
- Allow at most four philosophers to be sitting simultaneously at the table. Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick In Section 6.7, we present a solution to the dining-philosophers problem that ensures freedom from deadlocks.

- Note, however, that any satisfactory solution to the dining-philosophers problem must guard against the possibility
- that one of the philosophers will starve to death.
- A deadlock-free solution does not necessarily eliminate the possibility of starvation.

Dining Philosopher's Problem



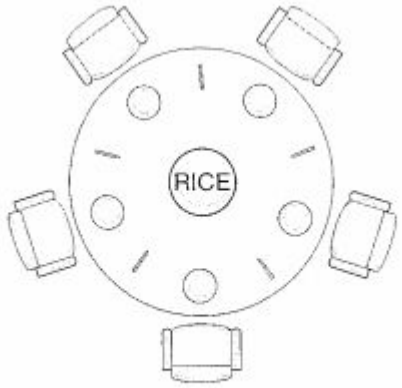


Figure 6.14 The situation of the dining philosophers.

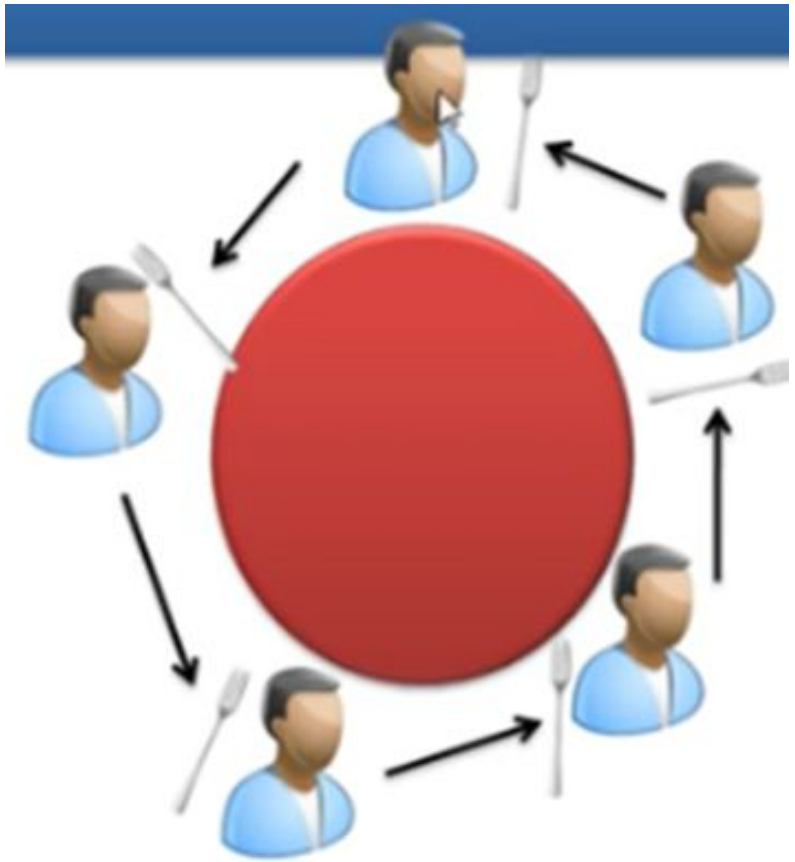
P[state={think,hungry,eat}
Shared data
Bowl of rice (data set)
Semaphore chopstick [5] initialized to 1

```
do {
wait(chopstick[i]);
wait(chopstick[(i+1)%5];

// Critical Section
eat

signal(chopstick[i]);
signal(chopstick[(i+1)%5];

think
}while(true);
```

Problems:
deadlock
starvation

Structure of Philosopher i

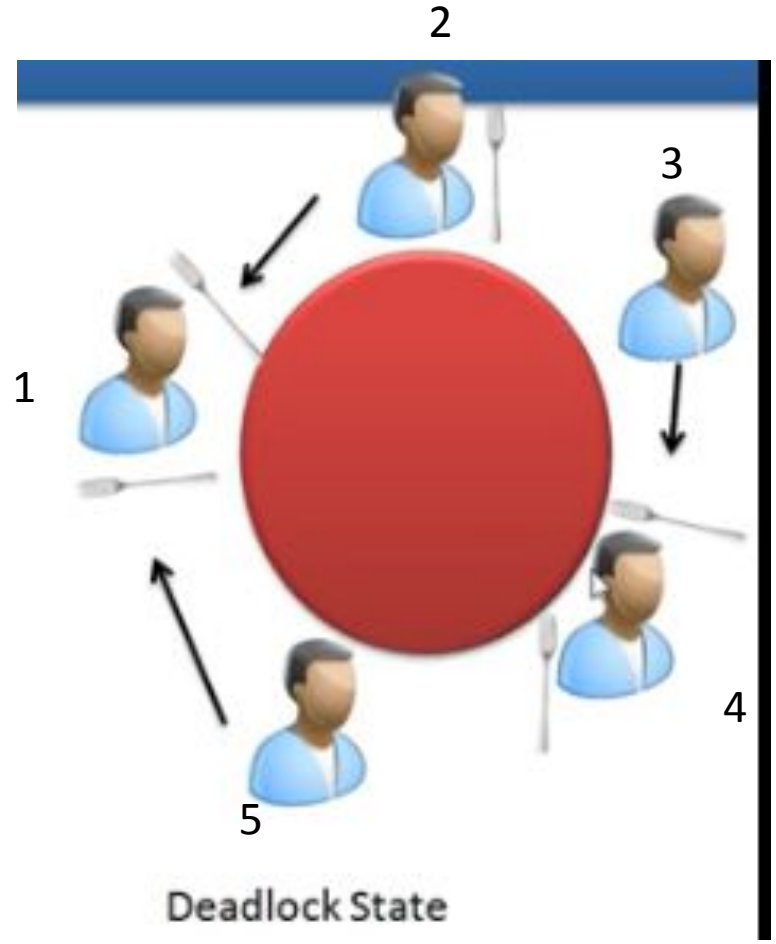
Problem 1



Deadlock state

- If all philosophers hungry & occupy their left chopstick at the same time their chopstick value is zero.
- Two neighbors cant eat at same time

Problem 2



- If two philosophers are faster than others, they think fast and eat fast, faster processes occupy chopsticks first
- Or greedy

starvatio
n

Solutions to DP problem

- Allow philosophers to eat when both chopsticks available

- Allow four philosophers at a time

- Allow odd philosopher to pick first left then right, while even one can pick first right , then left

- Asymmetric solution
- Odd ones use first left and then right chopstick while even can use right and then left chopstick.