UNIT 2: PROCESS MANAGEMENT

Process Concept

- A process is the unit-of-work.
- A system consists of a collection of processes:
    1. **OS process** can execute system-code and
    2. **User process** can execute user-code.

The Process

- A process is a program in execution.
- It also includes (Figure 2.1):
    1. **Program counter** to indicate the current activity.
    2. **Registers content** of the processor.
    3. **Process stack** contains temporary data.
    4. **Data section** contains global variables.
    5. **Heap** is memory that is dynamically allocated during process run time.
- A program by itself is not a process.
    1. A process is an active-entity.
    2. A program is a passive-entity such as an executable-file stored on disk.
- A program becomes a process when an executable-file is loaded into memory.
- If you run many copies of a program, each is a separate process.
    The text-sections are equivalent, but the data-sections vary.

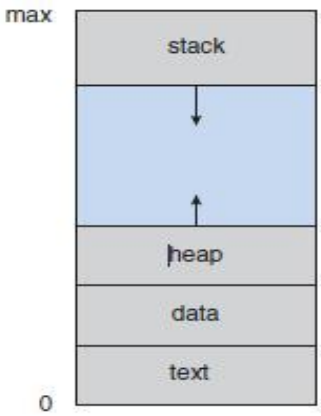


Figure 2.1 Process in memory

Process State

- As a process executes, it changes state.
- Each process may be in one of the following states (Figure 2.2):
    1. **New**: The process is being created.
    2. **Running**: Instructions are being executed.
    3. **Waiting:** The process is waiting for some event to occur (such as I/0 completions).
    4. **Ready**: The process is waiting to be assigned to a processor.
    5. **Terminated**: The process has finished execution.
- Only one process can be *running* on any processor at any instant.

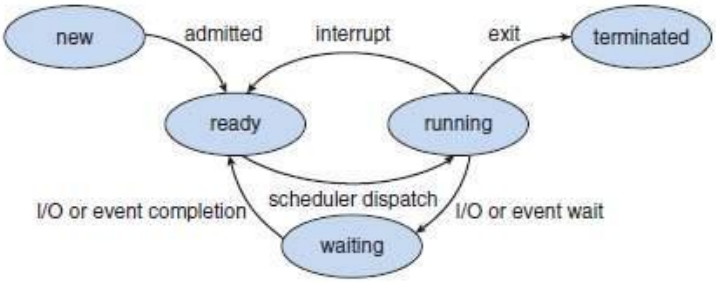


Figure 2.2 Diagram of process state

Process Control Block

• In OS, each process is represented by a PCB (Process Control Block).

Figure 2.3 Process control block (PCB)

• PCB contains following information about the process (Figure 2.3):

- **Process State:** The current state of process may be new, ready, running, waiting or halted.
- **Program Counter**: This indicates the address of the next instruction to be executed for the process.
- **CPU Registers:** These include accumulators (AX), index registers (SI, DI), stack pointers (SP) and general-purpose registers (BX, CX, DX).
- **CPU Scheduling Information:** This includes priority of process, pointers to scheduling-queues and scheduling-parameters. (Fig. 2.3.1 showing CPU switch from process to process)
- **Memory Management Information:** This includes value of base- & limit-registers and value of page-tables(or segment-tables).
- **Accounting Information:** This includes amount of CPU time, time-limit and process-number
- **I/O Status Information:** This includes list of I/O devices, list of open files.

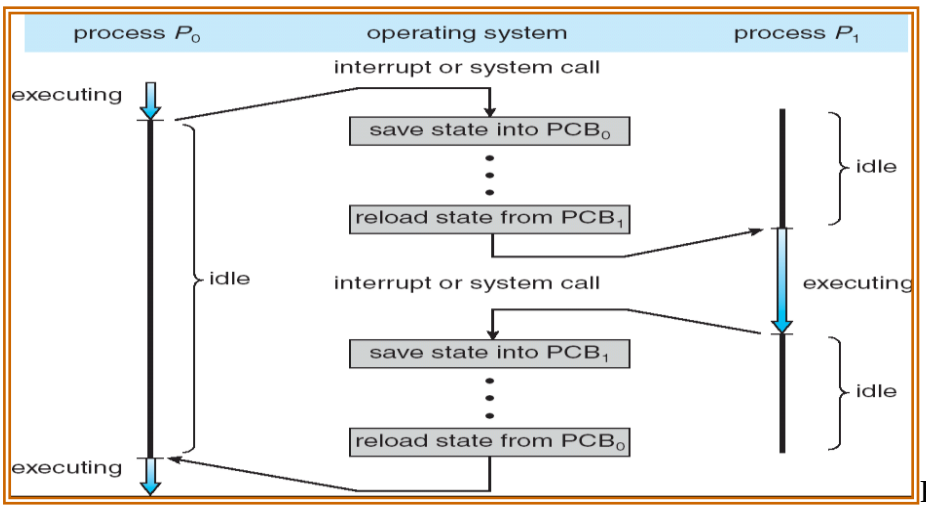


Fig (2.3.1)

Process Scheduling

• Objective of multiprogramming:

- To have some process running at all times to maximize CPU utilization. Objective of time-sharing:
- To switch the CPU between processes so frequently that users can interact with each program while it is running.

• To meet above 2 objectives: **Process scheduler** is used to select an available process for program-execution on the CPU.

Scheduling Queues

• Three types of scheduling-queues:

1. Job Queue

- This consists of all processes in the system.
- As processes enter the system, they are put into a job-queue.

2. Ready Queue: This consists of the processes that are residing in main-memory and ready & waiting to execute (Figure 2.4).

- 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 has a pointer to the next PCB in the ready-queue.

3. Device Queue

- This consists of the processes that are waiting for an I/O device.
- Each device has its own device-queue.

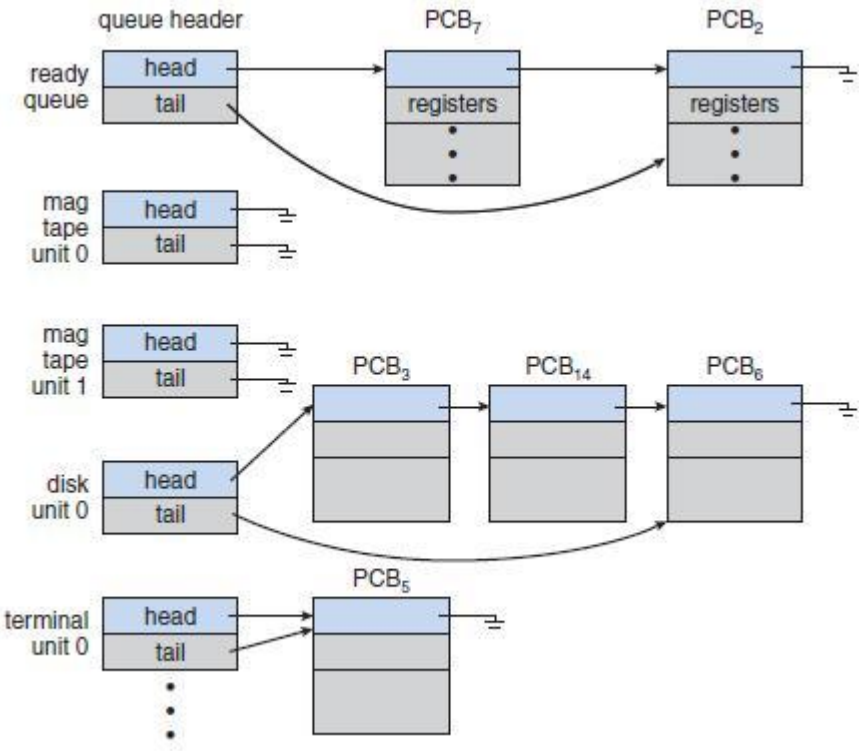


Figure 2.4 The ready-queue and various I/O device-queues

Queueing diagram : (Figure 2.5):

- Process is initially put in ready queue.
- Each rectangular box represents a queue.
- Circle represents resources that serves the queues.
- Arrows indicates the flow of processes in the system.

When the process is executing, one of following events could occur :

1. The process could issue an I/0 request and then be placed in an I/0 queue.
2. The process could create a new subprocess and wait for the subprocess's termination.
3. The process could be interrupted and put back in the ready-queue.

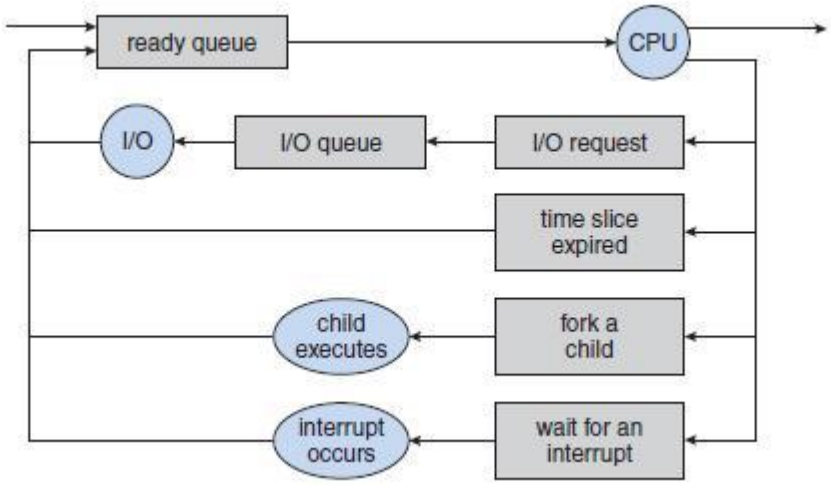


Figure 2.5 Queuing-diagram representation of process scheduling

Schedulers

• Three types of schedulers:

1. Long-term scheduler
2. Short-term scheduler and
3. Medium-term schedulers

Long-term Scheduler	***Short-term Scheduler***
Also called job scheduler.	Also called CPU scheduler.
Selects processes from disk that should be brought into the ready-queue.	Selects processes from ready queue that should be executed next and allocates CPU.
Need to be invoked only when a process leaves the system and therefore executes much less frequently.	Need to be invoked to select a new process for the CPU and therefore executes much more frequently.
May be slow ‘,’ minutes may separate the creation of one new process and the next.	Must be fast ‘,’ a process may execute for only a few milliseconds.
Controls the degree of multiprogramming by maintaining good process mix of CPU bound and I/O bound processes.	No process mix is maintained.

• Processes can be described as either:

1. I/O-bound Process

- Spends more time doing I/O operation than doing computations.
- Many short CPU bursts.

2. CPU-bound Process

- Spends more time doing computations than doing I/O operation.
- Few very long CPU bursts.

• Why long-term scheduler should select a good process mix of I/O-bound and CPU-bound

processes ? Ans: 1) If all processes are I/0 bound, then

i) Ready-queue will almost always be empty, and

ii) Short-term scheduler will have little to do.

2) If all processes are CPU bound, then

i) I/0 waiting queue will almost always be empty (devices will go unused) and

ii) System will be unbalanced.

- Some time-sharing systems have **medium-term scheduler** (Figure 2.6).
  - The scheduler removes processes from memory and thus reduces the degree of multiprogramming.
  - 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 scheduler.

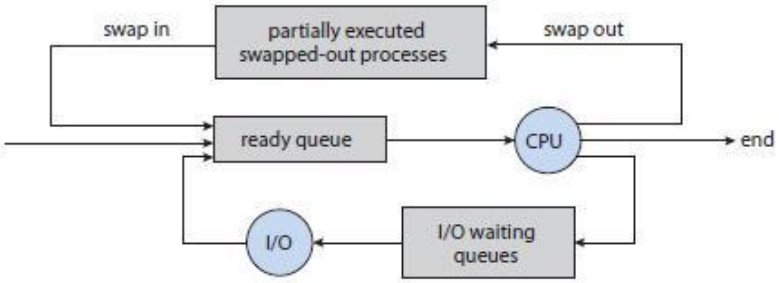


Figure 2.6 Addition of medium-term scheduling to the queueing diagram

Context Switch

- Context-switch means **state save** of the curent process and **state restore** i.e load the state of other process **.**
- Kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.
- The context of a process is represented in the PCB of the process; it includes
  - →value of CPU registers
  - →process-state and
  - →memory-management information.
- Disadvantages:
  1. Context-switch time is pure overhead, because the system does no useful work while switching.
  2. Context-switch times are highly dependent on hardware support.

Operations on Processes

1. Process Creation and
2. Process Termination

Process Creation

- A process may create a new process via a **create-process** system-call.
- The creating process is called a *parent-process*.

  The new process created by the parent is called the *child-process* (Sub-process).

- OS identifies processes by pid (process identifier), which is typically an integer-number.
- A process needs following resources to accomplish the task:
  - →CPU time
  - →memory and
  - → I/0 devices.
- Child-process may
  - → get resources directly from the OS or
  - → get resources of parent-process. This prevents any process from overloading the system.
- Two options exist when a process creates a new process:
  1. The parent & the children execute concurrently.
  2. The parent waits until all the children have terminated.
- Two options exist in terms of the address-space of the new process:
  1. The child-process is a duplicate of the parent-process (it has the same program and data as the parent).
  2. The child-process has a new program loaded into it.

Process creation in UNIX

- In UNIX, each process is identified by its process identifier (pid), which is a unique integer.
- A new process is created by the **fork()** system-call (Figure 2.7 & 2.8).
- The new process consists of a copy of the address-space of the original process.
- Both the parent and the child continue execution with one difference:
  1. The return value for the fork()

     is **zero** for the new

     (child) process.
  2. The return value for the fork() is

     nonzero pid of the child for the parent-process.
- Typically, the **exec()** system-call is used after a fork() system-call by one of the two processes to replace the process's memory-space with a new program.
- The parent can issue **wait()** system-call to move itself off the ready-queue.

```
#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;
}
```

Figure 2.7 Creating a separate process using the UNIX fork() system-call

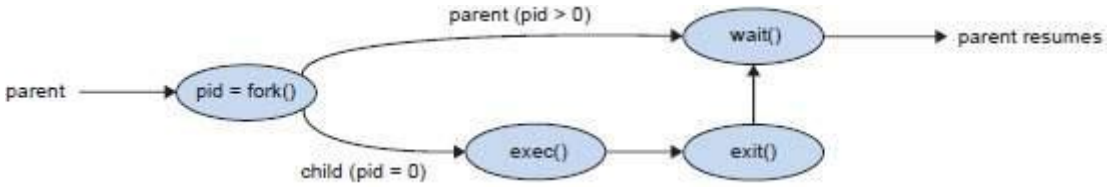


Figure 2.8 Process creation using the fork() system-call

Process Termination

- A process terminates when it executes the last statement (in the program).
- Then, the OS deletes the process by using **exit()** system-call.
- Then, the OS de-allocates all the resources of the process. The resources include
  - →memory
  - →open files and
  - → I/0 buffers.
- Process termination can occur in following cases:
  - → A process can cause the termination of another process via **TerminateProcess()** system-call.
  - →Users could arbitrarily **kill** the processes.
- A parent terminates the execution of children for following reasons:
  1. The child has exceeded its usage of some resources.
  2. The task assigned to the child is no longer required.
  3. The parent is exiting, and the OS does not allow a child to continue.
- In virtual systems, if a process terminates, then all its children must also be terminated. This phenomenon is referred to as **cascading termination**.

Inter Process Communication (IPC)

- Processes executing concurrently in the OS may be either
  1. Independent processes or 2. Co-operating processes.

1. A process is ***independent*** if
   - i) The process cannot affect or be affected by the other processes.
   - ii) The process does not share data with other processes.
2. A process is ***co-operating*** if
   - i) The process can affect or be affected by the other processes.
   - ii) The process shares data with other processes.

- Advantages of process co-operation:

1. Information Sharing

- Since many users may be interested in same piece of information (ex: shared file).

2. Computation Speedup

- We must break the task into subtasks.
- Each subtask should be executed in parallel with the other subtasks.
- The speed can be improved only if computer has multiple processing elements such as
  - →CPUs or
  - → I/O channels.

3. Modularity

- Divide the system-functions into separate processes or threads.

4. Convenience

- An individual user may work on many tasks at the same time.
- For ex, a user may be editing, printing, and compiling in parallel.

- Two basic models of IPC (Figure 2.9):
  1) Shared-memory and
  2) Message passing.

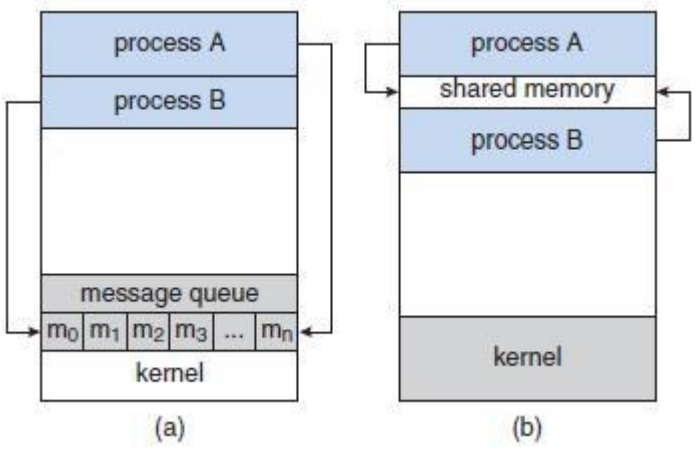


Figure 2.9 Communications models. (a) Message passing. (b) Shared-memory

Shared-Memory Systems

- Communicating-processes must establish a region of shared-memory.
- A shared-memory resides in address-space of the process creating the shared-memory. Other processes must attach their address-space to the shared-memory.
- The processes can then exchange information by reading and writing data in the shared-memory.
- The processes are also responsible for ensuring that they are not writing to the same location simultaneously.
- For ex, **producer-consumer problem**:
  - Producer-process produces information that is consumed by a consumer-process
  - 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:

1. ***Unbounded-buffer*** places no practical limit on the size of the buffer.
2. ***Bounded-buffer*** assumes that there is a fixed buffer-size.

Shared data (by producer & consumer) in shared memory

```
#define BUFFER_SIZE 10 // can only use BUFFER_SIZE-1 elements
typedef struct {
      ...
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

```
item nextProduced;

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

Figure 3.14 The producer process.

```
item nextConsumed;

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

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

Figure 3.15 The consumer process.

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

- Advantages:
    1. Allows maximum speed and convenience of communication.
    2. Faster.

Message-Passing Systems

- These allow processes to communicate and to synchronize their actions without sharing the same address-space.
- For example, a chat program used on the WWW.
- Messages can be of either
    1) Fixed size or 2) Variable size.
        1. If ***fixed-sized messages*** are used, the system-level implementation is simple.

        - However, the programming task becomes more difficult.

        2. If ***variable-sized messages*** are used, the system-level implementation is complex.

        - However, the programming task becomes simpler.
- A communication-link must exist between processes to communicate
- Three methods for implementing a link:
    1. Direct or indirect communication.
    2. Symmetric or asymmetric communication.
    3. Automatic or explicit buffering.
- Two operations:
    1. send(P,message): Send a message to process P.
    2. receive(Q,message): Receive a message from process Q.
- Advantages:
    1. Useful for exchanging smaller amounts of data (‘.’ No conflicts need be avoided).
    2. Easier to implement.
    3. Useful in a distributed environment.

Naming

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

Direct Communication	***Indirect Communication***
Each process must explicitly name the recipient/sender.	Messages are sent to/received from mailboxes (or ports).
Properties of a communication link: • A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate. • A link is associated with exactly two processes. • Exactly one link exists between each pair of processes.	Properties of a communication link: • A link is established between a pair of processes only if both members have a shared mailbox. • A link may be associated with more than two processes. • A number of different links may exist between each pair of communicating processes.
Symmetric addressing: • Both sender and receiver processes must name the other to communicate.	Mailbox owned by a process: • The owner can only receive, and the user can only send. • The mailbox disappears when its owner process terminates.
Asymmetric addressing: • Only the sender names the recipient; the recipient needn't name the sender.	Mailbox owned by the OS: • The OS allows a process to: 1. Create a new mailbox 2. Send & receive messages via it 3. Delete a mailbox.

Synchronization

Synchronous Message Passing	***Asynchronous Message Passing***
Blocking send: • The sending process is blocked until the message is received by the receiving process or by the mailbox.	Non-blocking send: • The sending process sends the message and resumes operation.
Blocking receive: • The receiver blocks until a message is available.	Non-blocking receive: • The receiver retrieves either a valid message or a null.

• Message passing may be either blocking or non-blocking (also known as synchronous and asynchronous).

Buffering

• Messages exchanged by processes reside in a temporary queue.

• Three ways to implement a queue:

1. Zero Capacity

- The queue-length is zero.
- The link can't have any messages waiting in it.
- The sender must block until the recipient receives the message.

2. Bounded Capacity

- The queue-length is finite.

- If the queue is not full, the new message is placed in the queue.
- The link capacity is finite.
- If the link is full, the sender must block until space is available in the queue.

3. Unbounded Capacity

- The queue-length is potentially infinite.
- Any number of messages can wait in the queue.
- The sender never blocks.

MULTITHREADED PROGRAMMING

Multithreaded Programming

• A thread is a basic unit of CPU utilization.

• It consists of

→thread ID

→PC

→register-set and

→stack.

• It shares with other threads belonging to the same process its code-section & data-section.

• 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. Such a process is called **multithreaded process** (Figure 2.10).

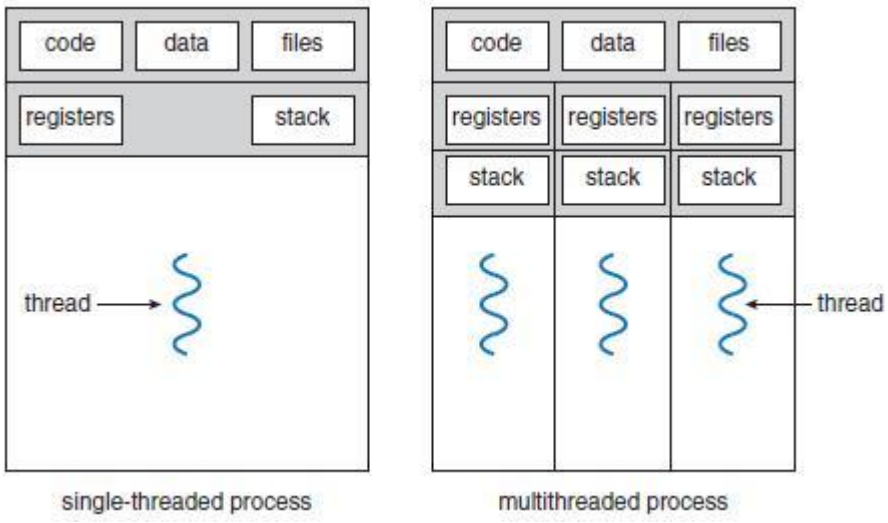


Figure 2.10 Single-threaded and multithreaded processes

Motivation for Multithreaded Programming

1. The software-packages that run on modern PCs are multithreaded.
   - An application is implemented as a separate process with several threads of control. For ex: A word processor may have
     →first thread for displaying graphics
     →second thread for responding to keystrokes and
     →third thread for performing grammar checking.

2. In some situations, a single application may be required to perform several similar tasks. For ex: A web-server may create a separate thread for each client request. This allows the server to service several concurrent requests.
3. 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.
4. Most OS kernels are multithreaded;
   - Several threads operate in kernel, and each thread performs a specific task, such as
     →managing devices or interrupt handling.

Benefits of Multithreaded Programming

1. Responsiveness

• A program may be allowed to continue running even if part of it is blocked. Thus, increasing responsiveness to the user.

2. Resource Sharing

• By default, threads share the memory (and resources) of the process to which they belong. Thus, an application is allowed to have several different threads of activity within the same address-space.

3. Economy

• Allocating memory and resources for process-creation is costly. Thus, it is more economical to create and context-switch threads.

4. Utilization of Multiprocessor Architectures

• In a multiprocessor architecture, threads may be running in parallel on different processors. Thus, parallelism will be increased.

Multithreading Models

• Support for threads may be provided at either
  1. The user level, for **user threads** or
  2. By the kernel, for **kernel threads**.

• User-threads are supported above the kernel and are managed without kernel

support. Kernel-threads are supported and managed directly by the OS.

- Three ways of establishing relationship between user-threads & kernel-threads:
  1. Many-to-one model
  2. One-to-one model and
  3. Many-to-many model.

Many-to-One Model

- Many user-level threads are mapped to one kernel thread (Figure 2.11).
- Advantage:
  - Thread management is done by the thread library in user space, so it is efficient.
- Disadvantages:
  1. The entire process will block if a thread makes a blocking system-call.
  2. Multiple threads are unable to run in parallel on multiprocessors.
- For example:
  →Solaris green threads
  →GNU portable threads.

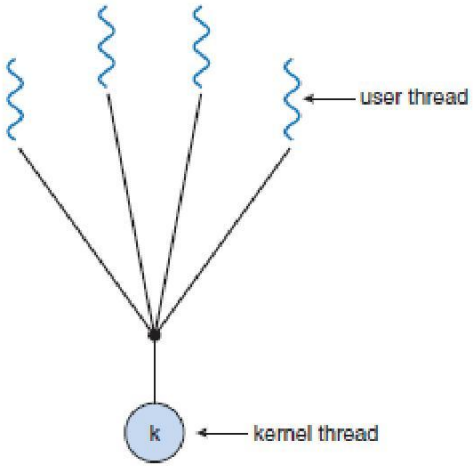


Figure 2.11 Many-to-one model

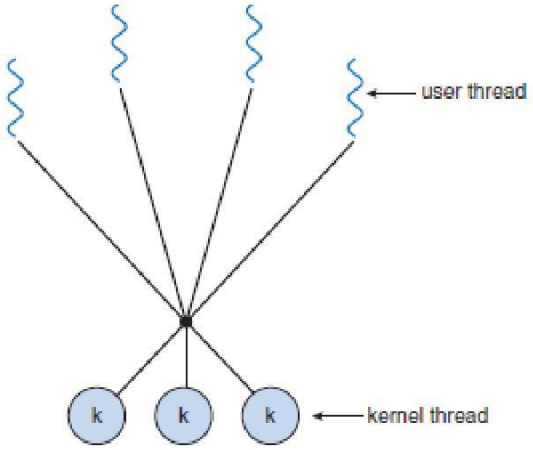


Figure 2.12 One-to-one model

One-to-One Model

- Each user thread is mapped to a kernel thread (Figure 2.12).
- Advantages:
  1. It provides more concurrency by allowing another thread to run when a thread makes a blocking system-call.
  2. Multiple threads can run in parallel on multiprocessors.
- Disadvantage:
  1. Creating a user thread requires creating the corresponding kernel thread.
- For example:
  →Windows NT/XP/2000, Linux

Many-to-Many Model

- Many user-level threads are multiplexed to a smaller number of kernel threads (Figure 2.13).
- Advantages:
  1. Developers can create as many user threads as necessary
  2. The kernel threads can run in parallel on a multiprocessor.
  3. When a thread performs a blocking system-call, kernel can schedule another thread for execution.

Two Level Model

- A variation on the many-to-many model is the two level-model (Figure 2.14).
- Similar to M:N, except that it allows a user thread to be bound to kernel thread.

- For example:
  - →HP-UX
  - →Tru64 UNIX

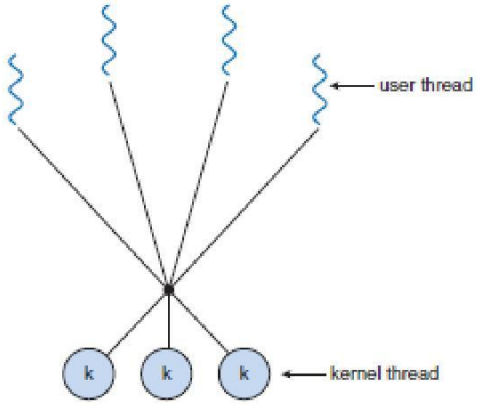


Figure 2.13 Many-to-many model

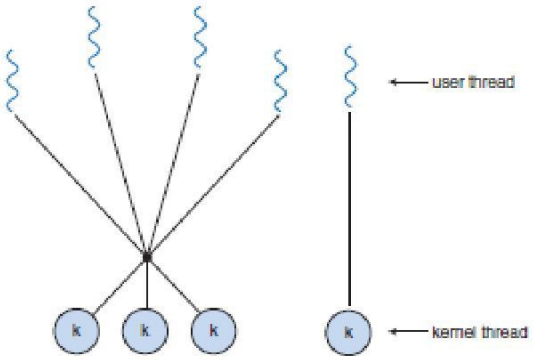


Figure 2.14 Two-level model

Thread Libraries

- It provides the programmer with an API for the creation and management of threads.
- Two ways of implemention:

1. First Approach

- Provides a library entirely in user space with no kernel support.
- All code and data structures for the library exist in the user space.

2. Second Approach

- Implements a kernel-level library supported directly by the OS.
- Code and data structures for the library exist in kernel space.

- Three main thread libraries:

1. POSIX Pthreads
2. Win32 and
3. Java.

Pthreads

- This is a POSIX standard API for thread creation and synchronization.
- This is a specification for thread-behavior, not an implementation.
- OS designers may implement the specification in any way they wish.
- Commonly used in: UNIX and Solaris.

```c
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }

   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Win32 threads

- Implements the one-to-one mapping
- Each thread contains
    - A thread id
    - Register set
    - Separate user and kernel stacks
    - Private data storage area

The register set, stacks, and private storage area are known as the context of the threads
The primary data structures of a thread include:

- ETHREAD (executive thread block)
- KTHREAD (kernel thread block)
- TEB (thread environment block)

Java Threads

- Threads are the basic model of program-execution in
  →Java program and
  →Java language.
- The API provides a rich set of features for the creation and management of threads.

- All Java programs comprise at least a single thread of control.
- Two techniques for creating threads:
    1. Create a new class that is derived from the Thread class and override its run() method.
    2. Define a class that implements the Runnable interface. The Runnable interface is defined as follows:

```
public interface Runnable
{
    public abstract void run();
}
```

Threading Issues

fork() and exec() System-calls

- fork() is used to create a separate, duplicate process.
- If one thread in a program calls fork(), then
    1. Some systems duplicates all threads and
    2. Other systems duplicate only the thread that invoked the forkO.
- If a thread invokes the exec(), the program specified in the parameter to exec() will replace the entire process including all threads.

Thread Cancellation

- This is the task of terminating a thread before it has completed.
- Target thread is the thread that is to be canceled
- Thread cancellation occurs in two different cases:
    1. ***Asynchronous cancellation***: One thread immediately terminates the target thread.
    2. ***Deferred cancellation***: The target thread periodically checks whether it should be terminated.

Signal Handling

- In UNIX, a signal is used to notify a process that a particular event has occurred.
- All signals follow this pattern:
    1. A signal is generated by the occurrence of a certain event.
    2. A generated signal is delivered to a process.
    3. Once delivered, the signal must be handled.
- A signal handler is used to process signals.
- A signal may be received either synchronously or asynchronously, depending on the source.
    1. Synchronous signals
    - Delivered to the same process that performed the operation causing the signal.
    - E.g. illegal memory access and division by 0.

2. Asynchronous signals

- Generated by an event external to a running process.
- E.g. user terminating a process with specific keystrokes <ctrl><c>.

• Every signal can be handled by one of two possible handlers:

1. A Default Signal Handler

- Run by the kernel when handling the signal.

2. A User-defined Signal Handler

- Overrides the default signal handler.

• In ***single-threaded programs***, delivering signals is simple (since signals are always delivered to a process).

In ***multithreaded programs***, delivering signals is more complex. Then, the following options exist: 1. Deliver the signal to the thread to which the signal applies.

2. Deliver the signal to every thread in the process.

3. Deliver the signal to certain threads in the process.

4. Assign a specific thread to receive all signals for the process.

Thread Pools

• The basic idea is to
→ create a no. of threads at process-startup and
→ place the threads into a pool (where they sit and wait for work).

• Procedure:

1. When a server receives a request, it awakens a thread from the pool. 2. If any thread is available, the request is passed to it for service.
3. Once the service is completed, the thread returns to the pool.

• Advantages:

1. Servicing a request with an existing thread is usually faster than waiting to create a thread. 2. The pool limits the no. of threads that exist at any one point.

• No. of threads in the pool can be based on factors such as → no. of CPUs
→ amount of memory and
→ expected no. of concurrent client-requests.

Thread Specific data

- Threads belonging to a process share the data of the process.
- Indeed, this sharing of data provides one of the benefits of multithreaded programming.
- In some circumstances, each thread might need its own copy of certain data.
  - We will call such data thread-specific data.
- For example, in a transaction-processing system, we might service each transaction in a separate thread.

- Furthermore, each transaction may be assigned a unique identifier. to associate each thread with its unique identifier, we could use thread-specific data.

Scheduler activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application.
- Scheduler activations provide **upcalls** a communication mechanism from the kernel to the thread library
- This communication allows an application to maintain the correct number kernel threads
- One scheme for communication between the user-thread library and the kernel is known as **scheduler activation.**

PROCESS SCHEDULING

Basic Concepts

- In a single-processor system,
  →only one process may run at a time.
  →other processes must wait until the CPU is rescheduled.
- Objective of multiprogramming:
  →to have some process running at all times, in order to maximize CPU utilization.

CPU-I/0 Burst Cycle

- Process execution consists of a cycle of
  →CPU execution and
  → I/O wait (Figure 2.15 & 2.16).
- Process execution begins with a CPU burst, followed by an I/O burst, then another CPU burst, etc…
- Finally, a CPU burst ends with a request to terminate execution.
- An I/O-bound program typically has many short CPU bursts.
  A CPU-bound program might have a few long CPU bursts.

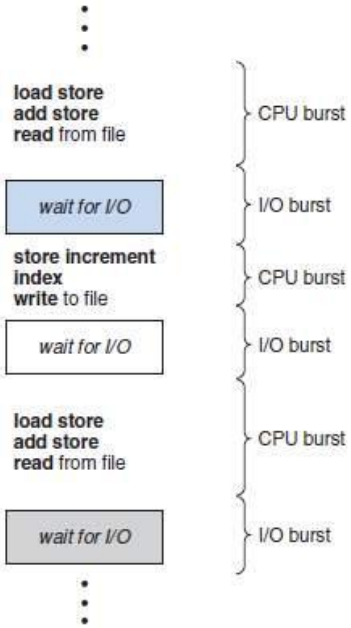


Figure 2.15 Alternating sequence of CPU and I/O bursts

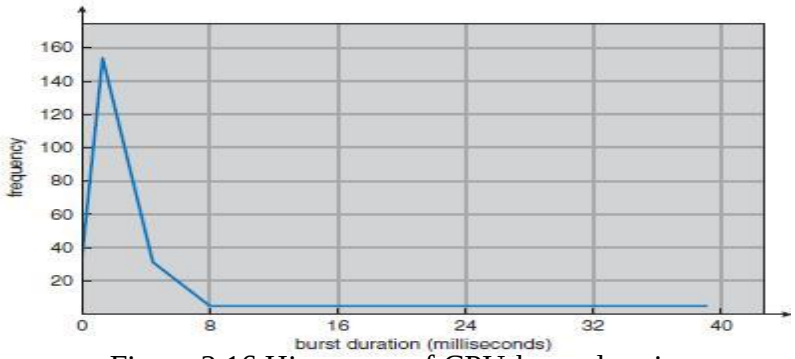


Figure 2.16 Histogram of CPU-burst durations

CPU Scheduler

- This scheduler
  - →selects a waiting-process from the ready-queue and
  - →allocates CPU to the waiting-process.
- The ready-queue could be a FIFO, priority queue, tree and list.
- The records in the queues are generally process control blocks (PCBs) of the processes.

CPU Scheduling

- Four situations under which CPU scheduling decisions take place:
  1. When a process switches from the running state to the waiting state. For ex; I/O request.
  2. When a process switches from the running state to the ready state. For ex: when an interrupt occurs.
  3. When a process switches from the waiting state to the ready state. For ex: completion of I/O.
  4. When a process terminates.
- Scheduling under 1 and 4 is non-preemptive.
  Scheduling under 2 and 3 is preemptive.

Non Preemptive 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.

Preemptive Scheduling

- This is driven by the idea of prioritized computation.
- Processes that are runnable may be temporarily suspended
- Disadvantages:
  1. Incurs a cost associated with access to shared-data.
  2. Affects the design of the OS kernel.

Dispatcher

- It gives control of the CPU to the process selected by the short-term scheduler.
- The function involves:
  1. Switching context
  2. Switching to user mode &
  3. Jumping to the proper location in the user program to restart that program.
- It should be as fast as possible, since it is invoked during every process switch.
- ***Dispatch latency*** means the time taken by the dispatcher to
  - →stop one process and
  - →start another running.

Scheduling Criteria:

In choosing which algorithm to use in a particular situation, depends upon the properties of the various algorithms.

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

CPU utilization: We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).

Throughput: 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.

Turnaround time. This is the important criterion which tells 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.

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.

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.

Scheduling Algorithms

- CPU scheduling deals with the problem of deciding which of the processes in the ready-queue is to be allocated the CPU.
- Following are some scheduling algorithms:
    1. FCFS scheduling (First Come First Served)
    2. Round Robin scheduling
    3. SJF scheduling (Shortest Job First)
    4. SRT scheduling
    5. Priority scheduling
    6. Multilevel Queue scheduling and
    7. Multilevel Feedback Queue scheduling

FCFS Scheduling

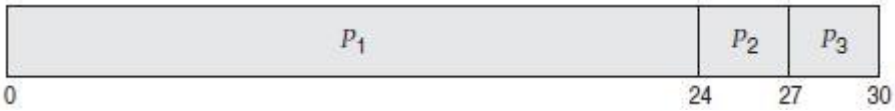
- The process that requests the CPU first is allocated the CPU first.
- The implementation is easily done using a FIFO queue.
- Procedure:

1. When a process enters the ready-queue, its PCB is linked onto the tail of the queue.
2. When the CPU is free, the CPU is allocated to the process at the queue's head.
3. The running process is then removed from the queue.

- Advantage:
  1. Code is simple to write & understand.
- Disadvantages:
  1. **Convoy effect:** All other processes wait for one big process to get off the CPU.
  2. Non-preemptive (a process keeps the CPU until it releases it).
  3. Not good for time-sharing systems.
  4. The average waiting time is generally not minimal.
- Example: Suppose that the processes arrive in the order P1, P2, P3.

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt Chart for the schedule is as follows:



- Waiting time for *P1* = 0; *P2* = 24; *P3* = 27
  Average waiting time: (0 + 24 + 27)/3 = 17ms

- Suppose that the processes arrive in the order P2, P3, P1.
- The Gantt chart for the schedule is as follows:

- Waiting time for P1 = 6;P2 = 0; P3 = 3
  Average waiting time: (6 + 0 + 3)/3 = 3ms

SJF Scheduling

- The CPU is assigned to the process that has the smallest next CPU burst.
- If two processes have the same length CPU burst, FCFS scheduling is used to break the tie.
- For long-term scheduling in a batch system, we can use the process time limit specified by the user, as the 'length'
- SJF can't be implemented at the level of short-term scheduling, because there is no way to know the length of the next CPU burst
- Advantage:
  1. The SJF is optimal, i.e. it gives the minimum average waiting time for a given set of processes.
- Disadvantage:
  1. Determining the length of the next CPU burst.
- SJF algorithm may be either 1) non-preemptive or 2) preemptive.

1. Non preemptive SJF

- The current process is allowed to finish its CPU burst.

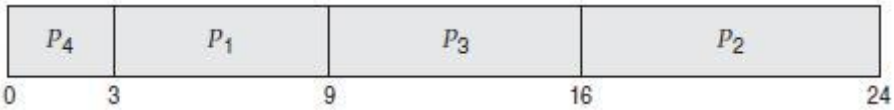
2. Preemptive SJF

- If the new process has a shorter next CPU burst than what is left of the executing process, that process is preempted.
- It is also known as **SRTF** scheduling (Shortest-Remaining-Time-First).

- Example (for non-preemptive SJF): Consider the following set of processes, with the length of the CPU-burst time given in milliseconds.

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- For non-preemptive SJF, the Gantt Chart is as follows:



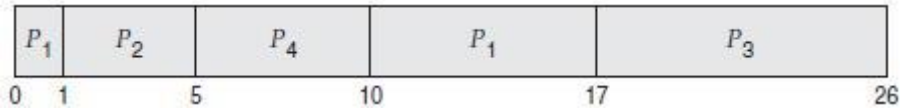
- Waiting time for P1 = 3; P2 = 16; P3 = 9; P4=0

  Average waiting time: (3 + 16 + 9 + 0)/4 = 7

- Example (preemptive SJF): Consider the following set of processes, with the length of the CPU-burst time given in milliseconds.

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

- For preemptive SJF, the Gantt Chart is as follows:



- The average waiting time is ((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4 = 6.5.

Priority Scheduling

- A priority is associated with each process.
- The CPU is allocated to the process with the highest priority.
- Equal-priority processes are scheduled in FCFS order.
- Priorities can be defined either internally or externally.

  1. ***Internally-defined*** priorities.

  - Use some measurable quantity to compute the priority of a process.
  - For example: time limits, memory requirements, no. of open

  files. 2. ***Externally-defined*** priorities.

  - Set by criteria that are external to the OS
  - For example:

    →importance of the process

    →political factors

- Priority scheduling can be either preemptive or nonpreemptive.

  1. ***Preemptive***

  - The CPU is preempted if the priority of the newly arrived process is higher than the priority of the currently running process. 2. ***Non Preemptive***
  - The new process is put at the head of the ready-queue

- Advantage:

  1. Higher priority processes can be executed first.

• Disadvantage:

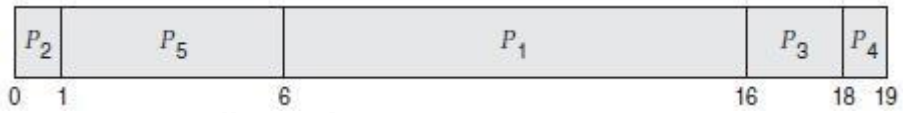
1. Indefinite blocking, where low-priority processes are left waiting indefinitely for CPU.

Solution: ***Aging*** is a technique of increasing priority of processes that wait in system for a long time.

• Example: Consider the following set of processes, assumed to have arrived at time 0, in the order PI, P2, ..., P5, with the length of the CPU-burst time given in milliseconds.

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

• The Gantt chart for the schedule is as follows:



• The average waiting time is 8.2 milliseconds.

Round Robin Scheduling

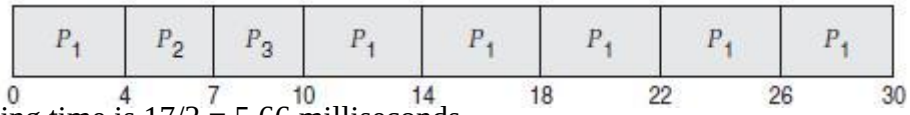
• Designed especially for timesharing systems.
• It is similar to FCFS scheduling, but with preemption.
• A small unit of time is called a **time quantum** *(or time slice).*
• Time quantum is ranges from 10 to 100 ms.
• The ready-queue is treated as a **circular queue**.
• The CPU scheduler
→goes around the ready-queue and
→allocates the CPU to each process for a time interval of up to 1 time quantum.
• To implement:
The ready-queue is kept as a FIFO queue of processes
• CPU scheduler
1. Picks the first process from the ready-queue.
2. Sets a timer to interrupt after 1 time quantum and
3. Dispatches the process.

• One of two things will then happen.
1. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily.
2. 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 OS. The process will be put at the tail of the ready-queue.
• Advantage:
1. Higher average turnaround than SJF.
• Disadvantage:
1. Better response time than SJF.

• Example: Consider the following set of processes that arrive at time 0, with the length of the CPU-burst time given in milliseconds.

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart for the schedule is as follows:



- The average waiting time is 17/3 = 5.66 milliseconds.
- *The RR scheduling algorithm is preemptive.*

  No process is allocated the CPU for more than *1* time quantum in a row. If a process' CPU burst exceeds *1* time quantum, that process is preempted and is put back in the ready-queue..
- The performance of algorithm depends heavily on the size of the time quantum (Figure 2.17 & 2.18).
  1. If time quantum=very large, RR policy is the same as the FCFS policy.
  2. If time quantum=very small, RR approach appears to the users as though each of n processes has its own processor running at l/n the speed of the real processor.
- In software, we need to consider the effect of context switching on the performance of RR scheduling
  1. Larger the time quantum for a specific process time, less time is spend on context switching.
  2. The smaller the time quantum, more overhead is added for the purpose of context-switching.

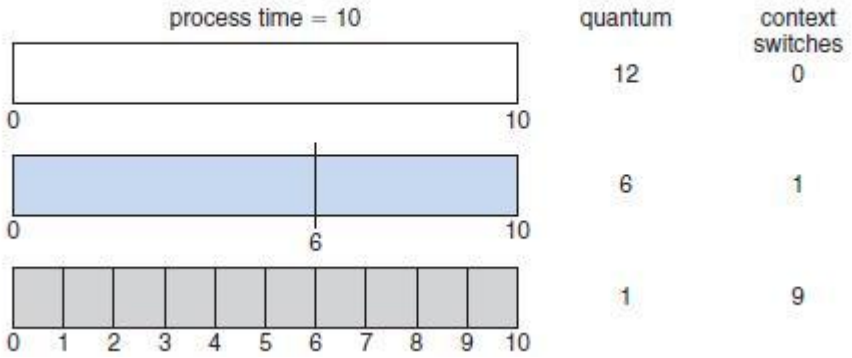
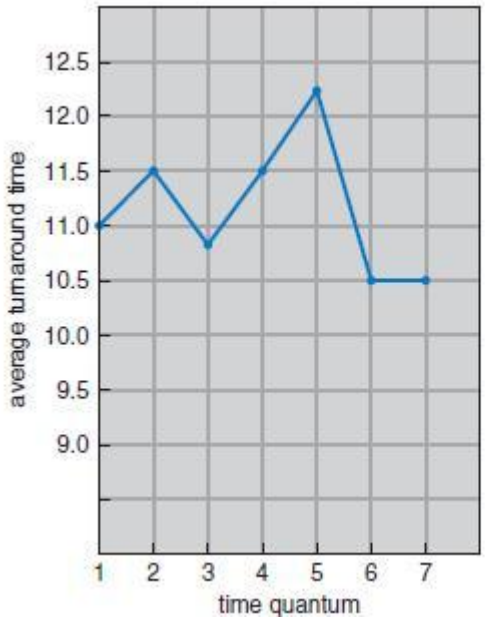


Figure 2.17 How a smaller time quantum increases context switches



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Figure 2.18 How turnaround time varies with the time quantum

Multilevel Queue Scheduling

- Useful for situations in which processes are easily classified into different groups.
- For example, a common division is made between
  - →foreground (or interactive) processes and
  - →background (or batch) processes.
- The ready-queue is partitioned into several separate queues (Figure 2.19).
- The processes are permanently assigned to one queue based on some property like
  - →memory size
  - →process priority or
  - →process type.
- Each queue has its own scheduling algorithm.
  For example, separate queues might be used for foreground and background processes.

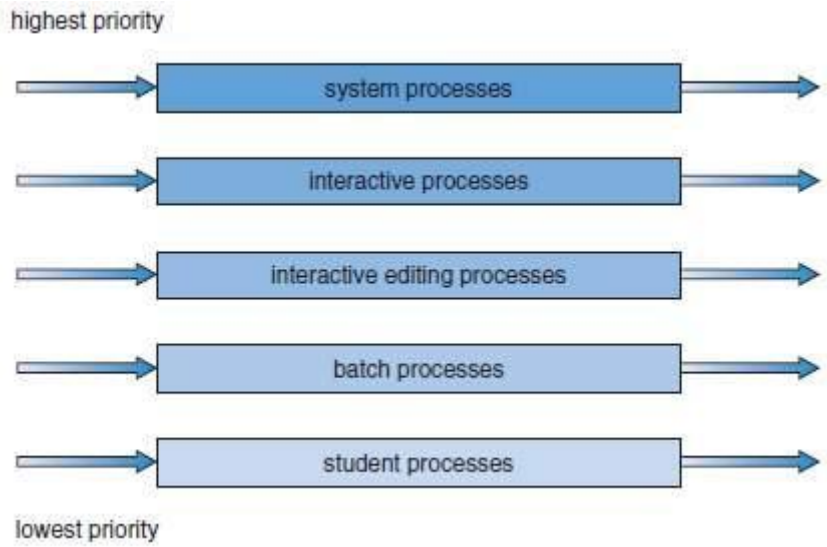


Figure 2.19 Multilevel queue scheduling

- There must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling.
  For example, the foreground queue may have absolute priority over the background queue.
- **Time slice**: each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR 20% to background in FCFS

Multilevel Feedback Queue Scheduling

- A process may move between queues (Figure 2.20).
- The basic idea: Separate processes according to the features of their CPU bursts. For example

1. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues.
2. If a process waits too long in a lower-priority queue, it may be moved to a higher-priority queue This form of aging prevents starvation.

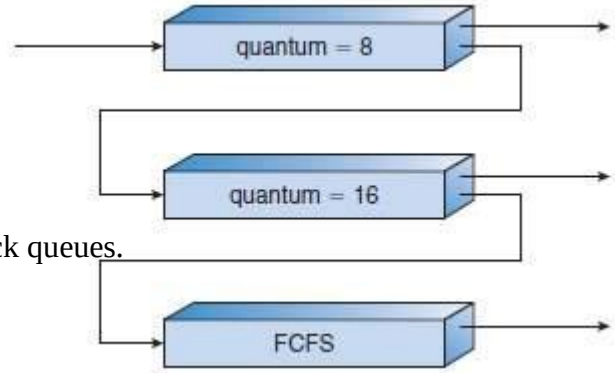


Figure 2.20 Multilevel feedback queues.

- In general, a multilevel feedback queue scheduler is defined by the following parameters:
    1. The number of queues.
    2. The scheduling algorithm for each queue.
    3. The method used to determine when to upgrade a process to a higher priority queue.
    4. The method used to determine when to demote a process to a lower priority queue.
    5. The method used to determine which queue a process will enter when that process needs service.

Multiple Processor Scheduling

- If multiple CPUs are available, the scheduling problem becomes more complex.
- Two approaches:
    1. Asymmetric Multiprocessing
    - The basic idea is:
        1. A master server is a single processor responsible for all scheduling decisions, I/O processing and other system activities.
        2. The other processors execute only user code.
    - Advantage:
        1. This is simple because only one processor accesses the system data structures, reducing the need for data sharing.

    2. Symmetric Multiprocessing
    - The basic idea is:
        1. Each processor is self-scheduling.
        2. To do scheduling, the scheduler for each processor
            i. Examines the ready-queue and
            ii. Selects a process to execute.

Restriction: We must ensure that two processors do not choose the same process and that processes are not lost from the queue.

Processor Affinity

- In SMP systems,
    1. Migration of processes from one processor to another are avoided and
    2. Instead processes are kept running on same processor. This is known as processor affinity.
- Two forms:
    1. Soft Affinity
    - When an OS try to keep a process on one processor because of policy, but cannot guarantee it will happen.
    - It is possible for a process to migrate between processors.

    2. Hard Affinity
    - When an OS have the ability to allow a process to specify that it is not to migrate to other processors. Eg: Solaris OS

Load Balancing

- This attempts to keep the workload evenly distributed across all processors in an SMP system.
- Two approaches:
    1. Push Migration
    - A specific task periodically checks the load on each processor and if it finds an imbalance, it evenly distributes the load to idle processors.

    2. Pull Migration
    - An idle processor pulls a waiting task from a busy processor.

Symmetric Multithreading

- The basic idea:
    1. Create multiple logical processors on the same physical processor.
    2. Present a view of several logical processors to the OS.
- Each logical processor has its own architecture state, which includes general-purpose and machine-state registers.
- Each logical processor is responsible for its own interrupt handling.
- SMT is a feature provided in hardware, not software.

Thread Scheduling

- On OSs, it is kernel-level threads but not processes that are being scheduled by the OS.
- User-level threads are managed by a thread library, and the kernel is unaware of them.
- To run on a CPU, user-level threads must be mapped to an associated kernel-level thread.

Contention Scope

- Two approaches:

    1. Process-Contention scope

    - On systems implementing the many-to-one and many-to-many models, the thread library schedules user-level threads to run on an available LWP.
    - Competition for the CPU takes place among threads belonging to the same process.

    2. System-Contention scope

    - The process of deciding which kernel thread to schedule on the CPU.
    - Competition for the CPU takes place among all threads in the system.
    - Systems using the one-to-one model schedule threads using only SCS.

Pthread Scheduling

- Pthread API that allows specifying either PCS or SCS during thread creation.
- Pthreads identifies the following contention scope values:
    1. PTHREAD_SCOPEJPROCESS schedules threads using PCS scheduling.
    2. PTHREAD-SCOPE_SYSTEM schedules threads using SCS scheduling.
- Pthread IPC provides following two functions for getting and setting the contention scope policy:
    1. pthread_attr_setscope(pthread_attr_t *attr, int scope)
    2. pthread_attr_getscope(pthread_attr_t *attr, int *scope)