#

PS01CMCA54 - Operating Systems

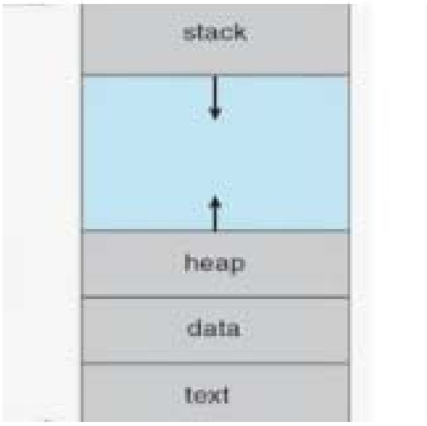
Process Management

* The concept of a process
* Scheduling of processes
* Interprocess communication
* Multithreading : concept, advantages, models
* Schedulers : long-term, medium-term, short-term
* CPU scheduling: criteria and algorithms
* Multiprocessor scheduling
* Introduction to process synchronization
* The critical section problem and Peterson’s solution
* The concepts of semaphores and monitors
* Introduction to deadlocks

The Process

- A ***process*** is a program in execution.
- Apart from the text section containing the ***program code***, it also includes the ***current activity***, as represented by a value of the program counter and the contents of the processor’s registers.
- A process generally also includes the process ***stack***, which contains temporary data (e.g., function parameters, return addresses, and local variables), and a ***data section***, which contains global variables. A process may also include a ***heap***, which is memory that is dynamically allocated during process run time.

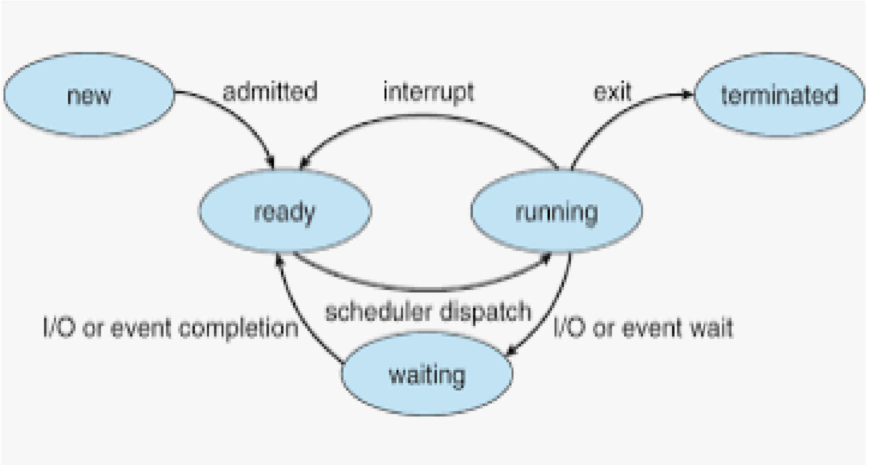
The structure of a process in memory



Program vs Process

- A ***program*** is a ***passive entity***, such as a file containing a list of instructions stored on disk (often called an executable file).
- A ***process*** is an ***active entity***, with a program counter specifying the next instruction to be executed and a set of associated resources.
- A program becomes a process when an executable file is loaded into memory.

Process State Diagram



- As a process executes, it changes state.
- The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states :

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

Process Control Block (PCB)

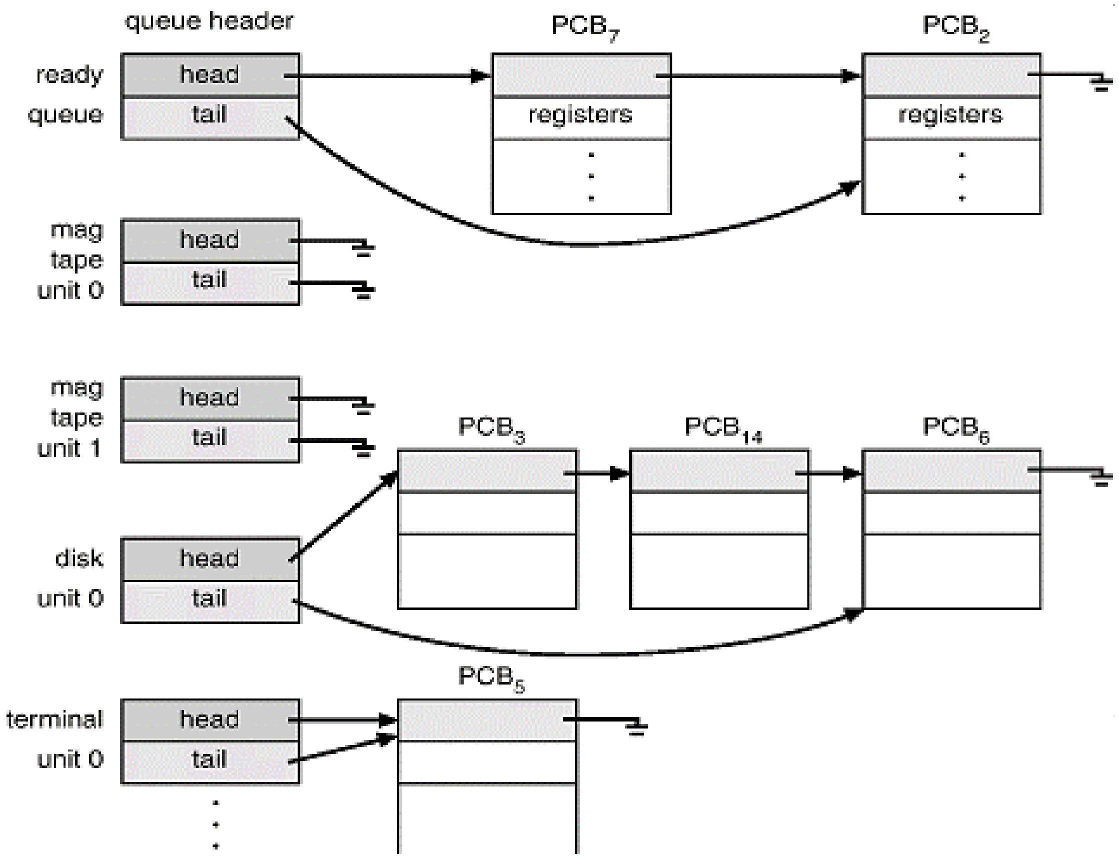
- Each process is represented in the operating system by a Process Control Block (PCB), which is also known as a Task Control Block.
- A PCB contains many pieces of information associated with a specific process, such as
  * Process state
  * Program counter
  * CPU registers
  * CPU scheduling information
  * Accounting information
  * I/O status information

Process Scheduling

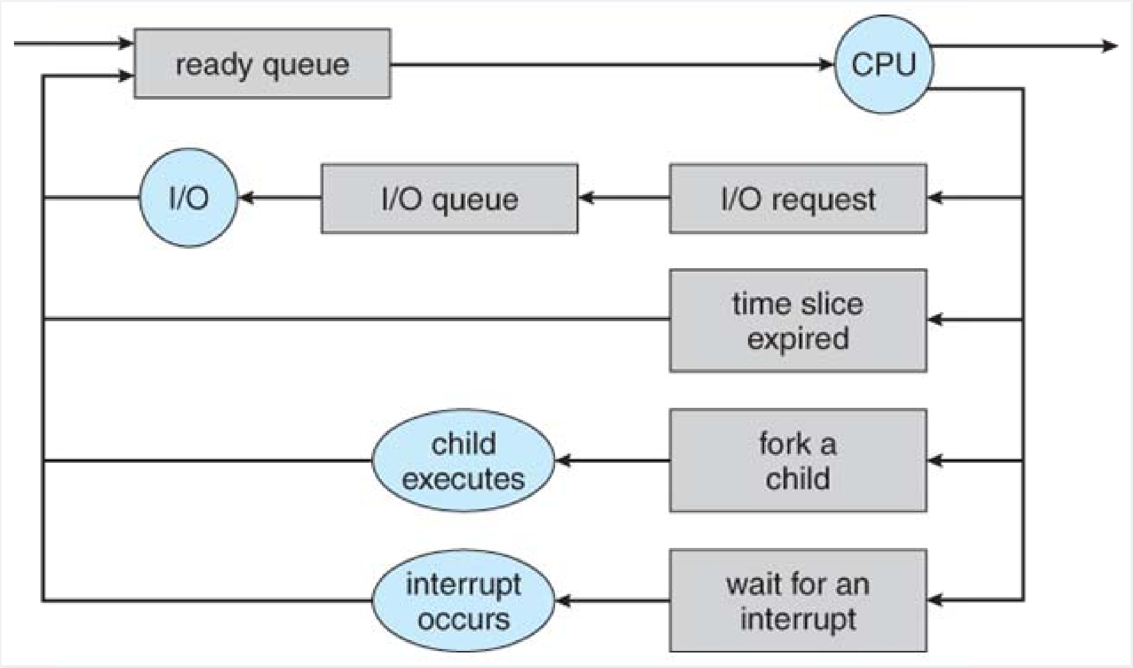
- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.
- To meet these objectives, the process scheduler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

Scheduling Queues

- As processes enter the system, they are put into a job queue, which consists of all processes in the system.
- ***Ready Queue*** : The processes that are residing in main memory and are ready and waiting to execute are kept on a list called a ready queue.
- The ready 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.
- ***Device Queue*** : A list of processes waiting for a particular I/O device is called a device queue.



Queueing Diagram representation of process scheduling



Queuing-diagram representation of process scheduling

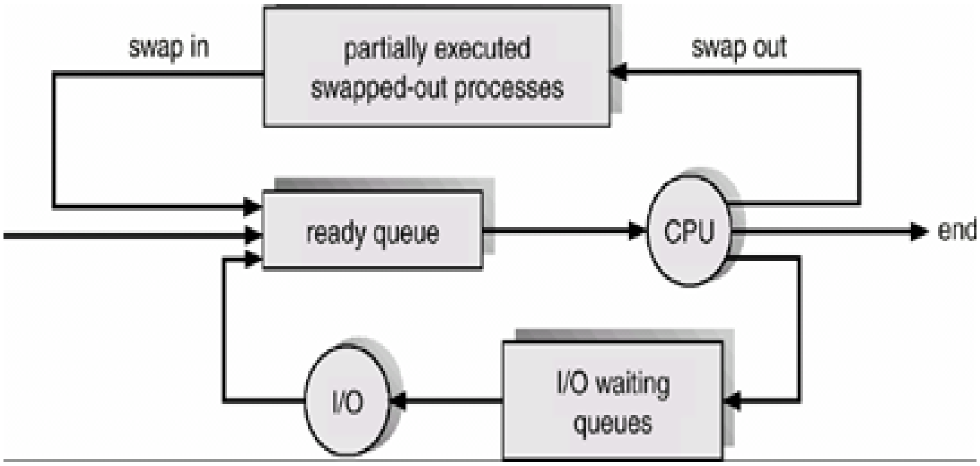
- A common representation of process scheduling is a queueing diagram.
- Each ***rectangular box*** in the diagram represents a ***queue***. Two types of queues are present : the ready queue and a set of device queues.
- The ***circles*** represent the ***resources*** that serve the queues, and the ***arrows*** indicate the flow of processes in the system.
- A new process is initially put in the ready queue. It waits 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/O request and then be placed in an I/O queue.
* The process could create a new subprocess and wait for the subprocess's termination.
* 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 various scheduling queues throughout its lifetime.
- The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by an appropriate **scheduler**.
- Often, in a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution.
- The ***long-term scheduler*** or ***job scheduler*** selects processes from the job pool kept on a mass-storage device and loads them into main 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.
- The primary distinction between the short-term scheduler and the long-term scheduler lies in the ***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/O 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.***
- ***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).***
- 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.***



- 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 in doing I/O than it spends doing computations. A CPU-bound process, in contrast, generates I/O requests infrequently, using more of its time doing computations.
- It is important that 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/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced.
- ***The system with the best performance will have a combination of CPU-bound and I/O-bound processes.***
- Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling.
- 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 multiprogramming. Later, the process can be re-introduced 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 over-committed available memory, requiring memory to be freed up.

Interprocess Communication (IPC)

- Processes executing concurrently in the OS 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. Any process that shares data with other processes is a cooperating process.

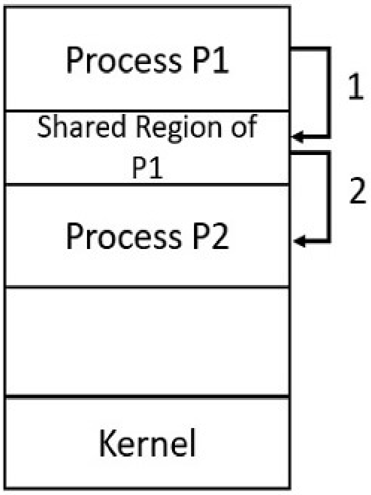
There are several reasons for providing an environment that allows process cooperation :

- Information sharing : Since several users may be interested in the same piece of information (e.g., 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. Such a speedup can be achieved only if the computer has multiple processing elements (CPUs or I/O channels).
- 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.

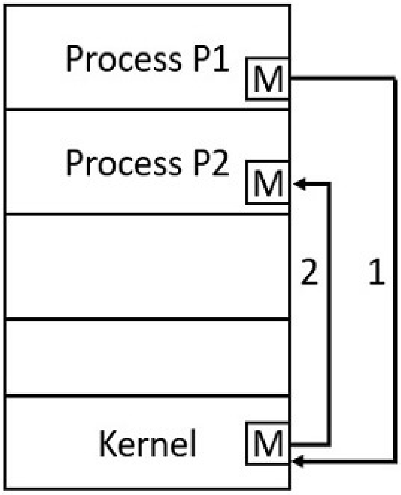
- Interprocess communication (IPC) mechanism is required to allow ***cooperating processes*** to ***exchange data and information***.
- There are ***two fundamental models of interprocess communication*** :

(1) Shared memory model
(2) Message passing model

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.
Message passing model : Communication takes place by means of messages exchanged between the cooperating processes.



Shared Memory System



Message Passing System

Message passing model :

- Useful for exchanging smaller amounts of data, since no conflicts need be avoided.
- Easier to implement than shared-memory model.
- Message-passing systems are typically implemented using system calls.
- Required more time-consuming task of kernel intervention.

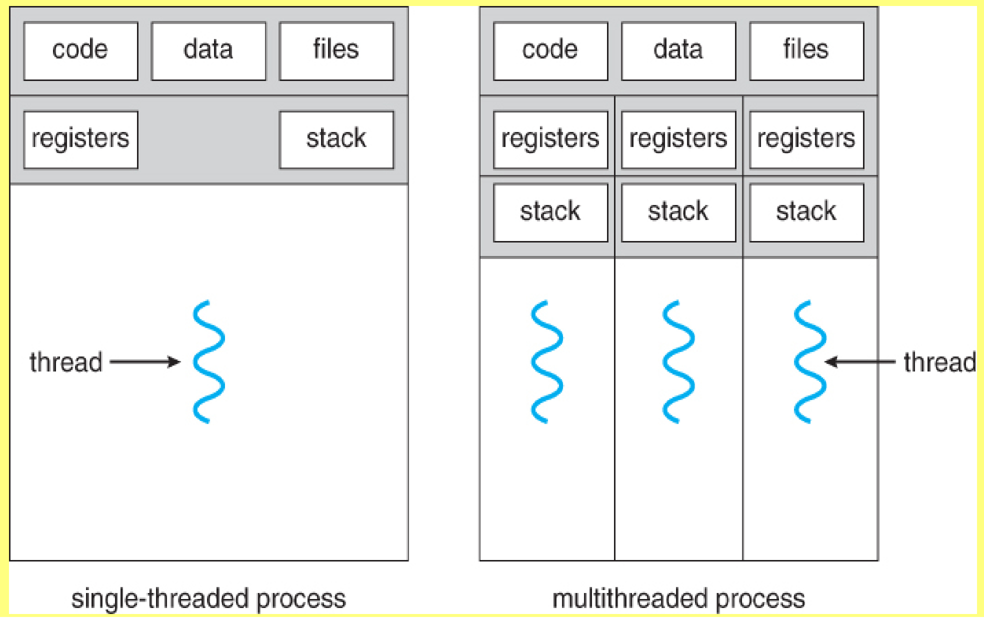
Shared memory model :

- Allows maximum speed and convenience of communication.
- Faster than message-passing
- System calls are required only to establish shared-memory regions. No assistance from the kernel is required thereafter.
- Processes exchange information by reading and writing data in the shared areas.

Multithreading

- A thread is a basic unit of CPU utilization.
- It comprises
  * a thread ID
  * a program counter
  * a register set
  * a stack
- A thread shares with other threads belonging to the same process its code section, data section, and other operating-system resources, 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.

Single-threaded and multithreaded processes



Benefits of Multithreaded Programming

(1) Responsiveness : Multithreading an interactive environment may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. For instance, a multithreaded Web browser could allow user interaction in one thread while an image was being loaded in another thread.

(2) Resource sharing : 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.

(3) 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. It is much more time consuming to create and manage processes than threads. In Solaris, for example, creating a process is about 30 times slower than is creating a thread, and context-switching is about 5 times slower.

(4) Scalability : The benefit 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.

Multithreading Models

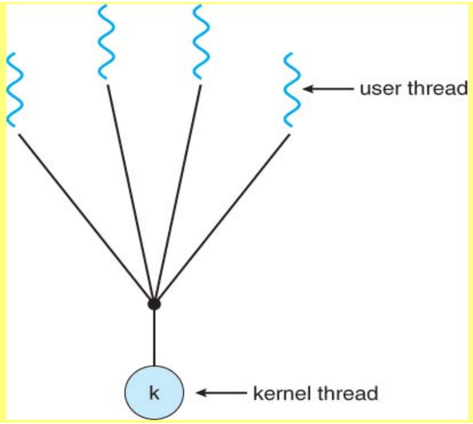
- Support for threads may be provided either at the user level, for user threads, or by the kernel, for kernel threads.
- User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system.
- Virtually all contemporary operating systems – including Windows XP, Linux, Mac OS X, Solaris, and True64 UNIX – support kernel threads.
- Ultimately, a relationship must exist between user threads and kernel threads.

There are three popular multithreading models :

1. Many-to-One model
2. One-to-One Model
3. Many-to-Many Model

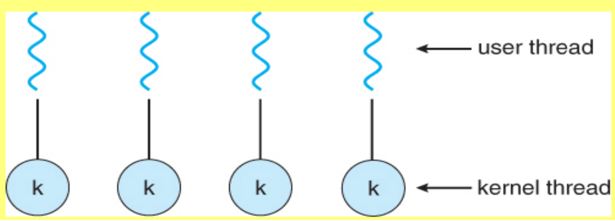
1. **Many-to-One Model :**

- The many-to-one model maps many user-level threads to one kernel thread.
- Thread management is handled by the thread library in user space, which is very efficient.
- However, if a blocking system call is made, then the entire process blocks.
- Because only a single kernel thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.
- Green threads for Solaris and GNU Portable Threads implemented the many-to-one model in the past, but few systems continue to do so today.



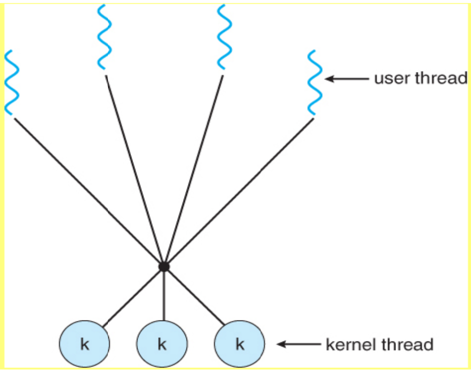
2. One-to-One Model

- 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.
- Drawback : 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, family of Windows OSs
- One-to-one model overcomes the problems listed above involving blocking system calls and the splitting of processes across multiple CPUs.
- However the overhead of managing the one-to-one model is more significant, involving more overhead and slowing down the system.
- Most implementations of this model place a limit on how many threads can be created.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.



3. Many-to-Many Model :

- The many-to-many model multiplexes many user-level threads to an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
- Developers have no restrictions on the number of threads created, but true concurrency is not gained since 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.
- Blocking kernel system calls do not block the entire process.



CPU scheduling: criteria and algorithms

CPU scheduling criteria

- CPU utilization : Keep CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In real systems : 40 percent to 90 percent.
- Throughput : The amount of work done in unit time. Here it refers to the number of processes that are completed per unit time. For long processes, the rate may be one process per hour; for short transactions, it may be ten processes per second.
- Turnaround time : The interval from the time of submission of a process to the time of completion. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- Waiting time : Waiting time is the sum of the periods spent waiting in the ready queue. The CPU scheduling algorithm does not affect the amount of time during which a process executes or does I/O. It affects only the amount of time that a process spends waiting in the ready queue.
- Response time : refers to the time from submission of a request until the first response is produced. Response time is the time it takes to start responding, not the time taken to output the response. 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.

CPU scheduling algorithms

- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-First (SJF) Scheduling
- Priority Scheduling
- Round-Robin (RR) Scheduling
- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling

First-Come, First-Served (FCFS) Scheduling

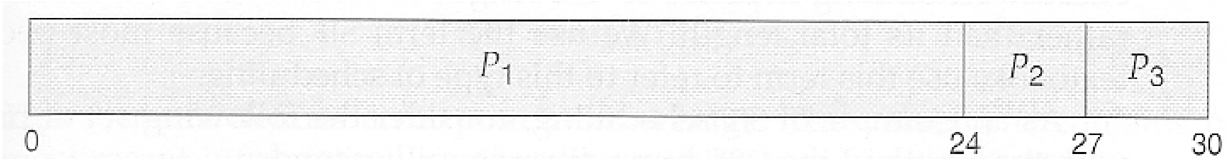
- The process that requests the CPU first is allocated the CPU first.
- Implementation of the FCFS policy is easily managed with a FIFO queue.
- When a process enters the ready queue, its PCB is linked onto the tail of the queue.
- When the CPU is free, it is allocated to the process at the head of the queue.
- The code for FCFS scheduling is simple to write and understand.
- The average waiting time under the FCFS policy is often quite long.

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

Process	Burst Time
P_1	24
P_2	03
P_3	03

Average Waiting Time = (0+24+27) / 3
= 17 milliseconds

If the processes arrive in the order P_1, P_2, P_3, and are served in FCFS order, we get the result shown in the following Gantt chart, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes :

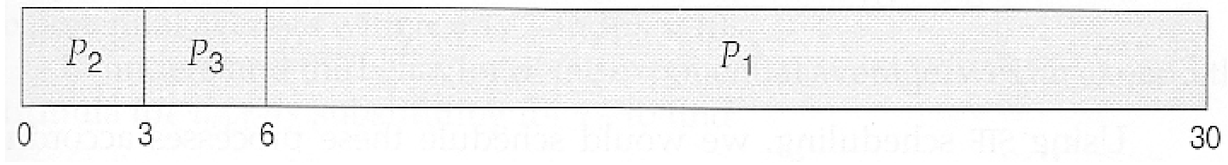


Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds :

Process	Burst Time
P_1	24
P_2	03
P_3	03

Average Waiting Time = (6+0+3) / 3
= 3 milliseconds

If the processes arrive in the order P_2, P_3, P_1, and are served in FCFS order, we get the result shown in the following Gantt chart, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes :



Shortest-Job-First (SJF) Scheduling

- The SJF algorithm associates with each process the length of the process's next CPU burst.
- When the CPU is 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 algorithm is also called the *shortest-next-CPU-burst* algorithm.

Example

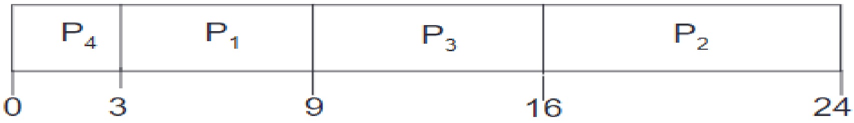
Consider the following set of processes, with the length of the CPU burst given in milliseconds :

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Average Waiting Time = (3+16+9+0) / 4
= 7 milliseconds

Using SJF scheduling, we would schedule these processes according to the following Gantt chart :

Gantt chart



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

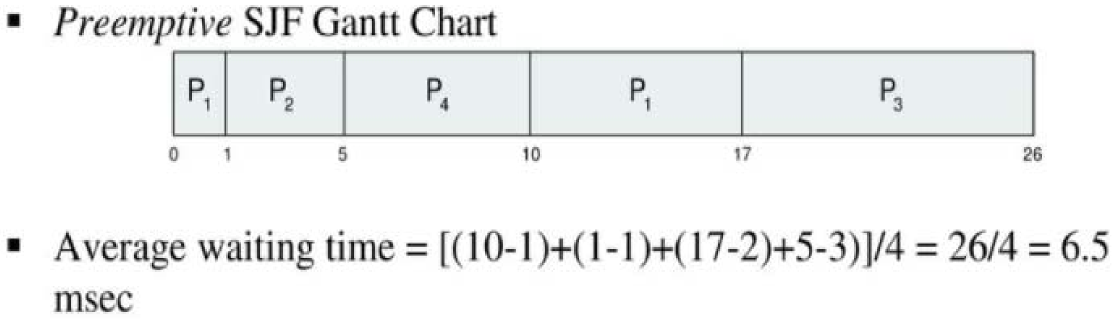
- The SJF scheduling algorithm is provably optimal.
- It gives the minimum average waiting time for a given set of processes.
- Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.
- The real difficulty with the SJF algorithm is knowing the length of the next CPU burst. 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.
- 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. With short-term scheduling, there is no way to know the length of the next CPU burst.
- One approach is to try to approximate SJF scheduling.
- We may not ***know*** the length of the next CPU burst, but we may be able to ***predict*** its value.
- We expect that the next CPU burst will be similar in length to the previous ones.
- By computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.
- The next CPU burst is generally predicted as an exponential average of the measured lengths of previous CPU bursts.
- We can define the exponential average with a formula.

- The SJF algorithm can be either preemptive or non-preemptive.
- 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 non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst.
- Preemptive SJF scheduling is sometimes called shortest-remaining-time-first scheduling.

Consider the following four processes, with the length of the CPU burst given in milliseconds :

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

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the ***resulting preemptive SJF schedule*** is as depicted in the following ***Gantt chart*** :



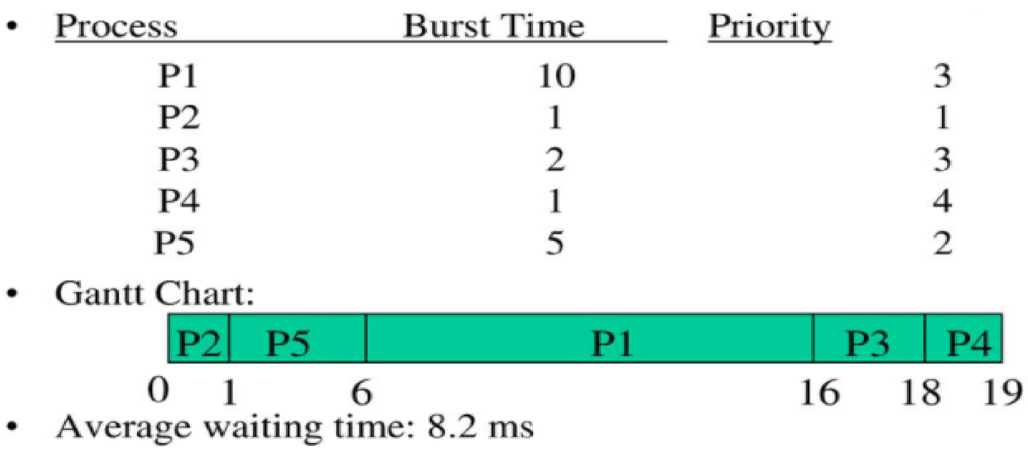
- *Preemptive* SJF Gantt Chart

- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

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 4095.
- 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.

- As an example, consider the following set of processes, assumed to have arrived at time 0 in the order P_1, P_2,…,P_5, with the length of the CPU burst given in milliseconds :



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

- Gantt Chart:

- Average waiting time: 8.2 ms

- Priorities can be defined either internally or externally.
- Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities.
- External priorities are set by criteria outside the operating system, such as the importance of a process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political factors.

- Priority scheduling can be either preemptive or non-preemptive.
- 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 ***non-preemptive priority scheduling algorithm*** will simply put the new process at the head of the ready queue.
- A major problem with the priority scheduling algorithm is ***indefinite blocking*** or ***starvation***.
- A process that is ready to run but waiting for the CPU can be considered blocked.
- ***A priority scheduling algorithm can leave some low-priority processes waiting indefinitely***.
- ***In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.***
- Generally, one of two things will happen. Either the process will eventually be run or the computer system will eventually crash and lose all unfinished low-priority processes.
- 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.

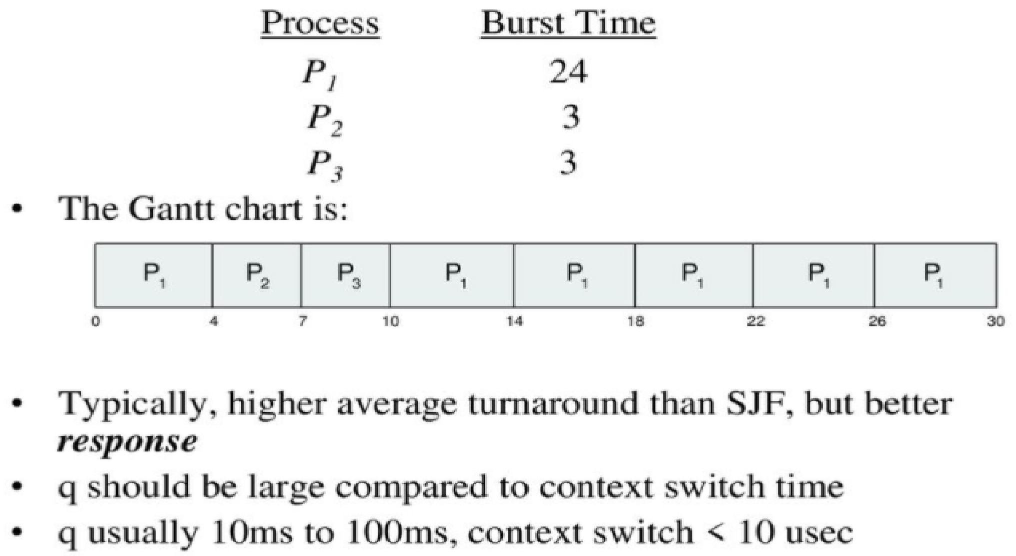
- For example, if priorities range from 127(low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed. In fact, it would take no more than 32 hours for a priority-127 process to age to a priority-0 process.

Round-Robin 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.
- A 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 OS. 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.

Example of RR with a Time Quantum of 4 milliseconds

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



Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

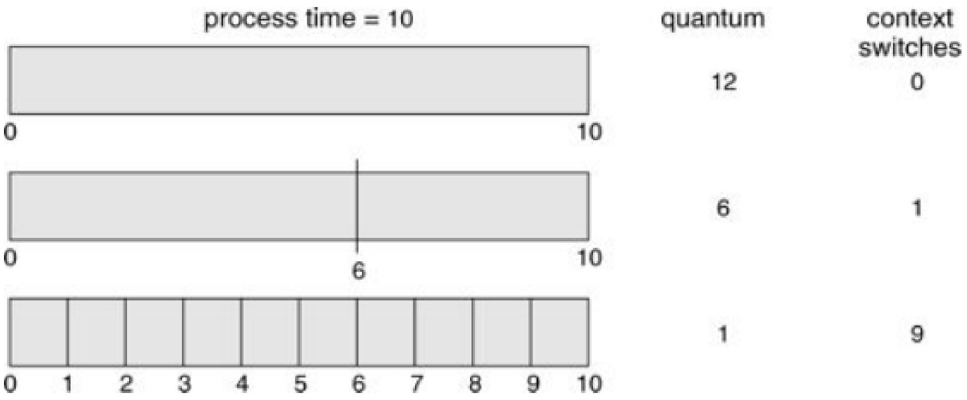
0 4 7 10 14 18 22 26 30

- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

Average waiting time = (6+4+7)/3 = 17/3 = 5.66 ms

- In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process).
- If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue.
- The RR scheduling algorithm is thus preemptive.
- If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units. Each process must wait no longer than (n-1) x q time units until its next time quantum.
- The performance of the RR algorithm depends heavily on the size of the time quantum.
- At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy.
- In contrast, if the time quantum is extremely small (say, 1 millisecond), the RR approach is called ***processor sharing*** and (in theory) creates the appearance that each of the n processes has its own processor running at 1/n the speed of the real processor.
- The ***effect of context switching on the performance of RR*** scheduling should be examined.
- ***The time quantum should be large with respect to the context-switch time.***

- Assume that we have only one process of 10 time units. If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead. If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch. If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly.

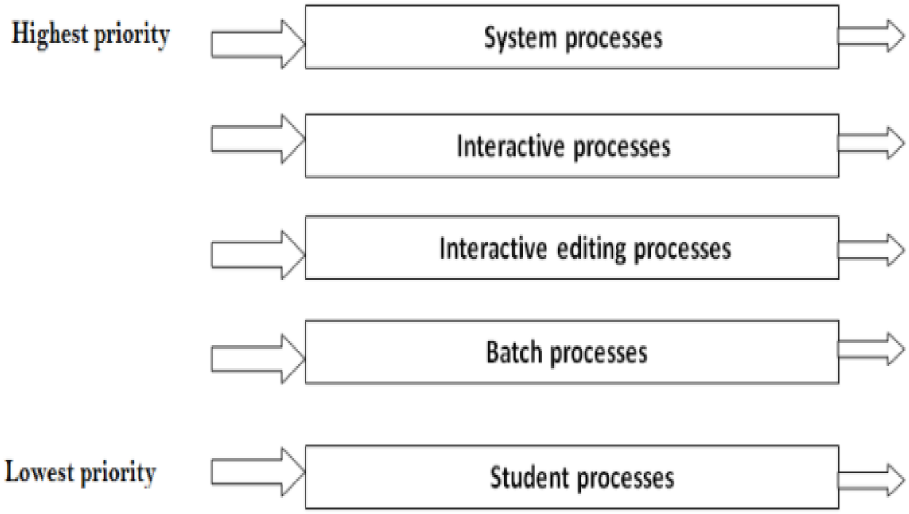


- If the context-switch time is approximately 10 percent of the time quantum, then about 10 percent of the CPU time will be spent in context switching.
- In practice, most modern systems have time quanta ranging from 10 to 100 milliseconds.
- Turnaround time also depends on the size of the time quantum. The average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases. In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum.
- Although the time quantum should be large compared with the context- switch time, it should not be too large. If the time quantum is too large, as mentioned earlier, RR scheduling degenerates to an FCFS policy. A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.

Multilevel Queue Scheduling

- Another class of scheduling algorithms has been created 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. These two types of processes have different response-time requirements, and so might have different scheduling needs. In addition, foreground processes may have priority 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.

- For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by the Round Robin algorithm, while the background queue is scheduled by an FCFS algorithm.

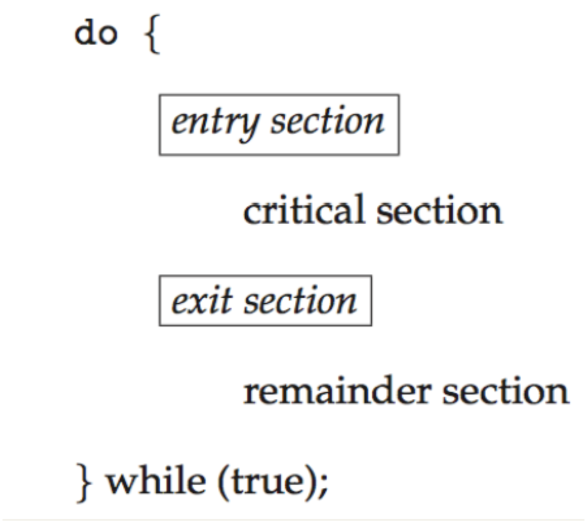


- In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, The foreground queue may have absolute priority over the background queue.
- Let us consider an example of a multilevel queue-scheduling algorithm with five queues:
- System Processes
- Interactive Processes
- Interactive Editing Processes
- Batch Processes
- Student Processes
- Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process will be preempted.
- Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes.

The Critical-Section Problem

- Consider a system consisting of n processes {P_0, P_1, ..., 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***. The critical section may be followed by an ***exit section***. The remaining code is the ***remainder section***.

The general structure of a typical process Pi is shown below :



```
do {

    entry section

        critical section

    exit section

        remainder section

} while (true);
```

A solution to the critical-section problem must satisfy the following three requirements:

1. Mutual exclusion : If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

2. Progress : If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. Bounded waiting : There exists a bound, or limit, 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.

- We assume that each process is executing at a nonzero speed. However, we can make no assumption concerning the relative speed of the n processes.

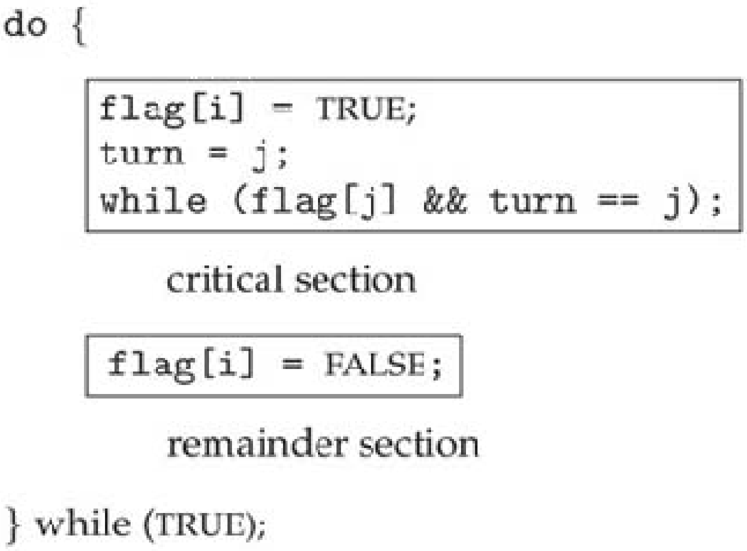
Peterson's Solution

- Next, we illustrate 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 P_0 and P_1. For convenience, when presenting P_i, we use P_j to denote the other process; that is, j equals 1 - i.
- Peterson's solution requires the two processes to share two data items :

  ```
  int  turn;
  boolean  flag [2];
  ```

- The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process Pi is allowed to execute in its critical section. The flag array is used to indicate if a process is *ready* to enter its critical section. For example, if flag[i] is true, this value indicates that P_i is ready to enter its critical section. With an explanation of these data structures complete, we are now ready to describe the algorithm shown in Figure.
- To enter the critical section, process P_i first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so. If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately.

The structure of process P_i in Peterson's Solution :



```
do {

    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

- 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.

- To prove property 1, we note that each P_i enters its critical section only if either flag[j] == false or turn == i. Also note that, if both processes can be executing in their critical sections at the same time, then flag [0] == flag [1] == true. These two observations imply that P_0 and P_1 could not have successfully executed their *while statements* at about the same time, since the value of turn can be either 0 or 1 but cannot be both. Hence, one of the processes - say, Pj - must have successfully executed the while statement, whereas P_i had to execute at least one additional statement ("turn == j"). However, at that time, flag[j] == true, and turn == j, and this condition will persist as long as P_j is in its critical section; as a result, mutual exclusion is preserved.
- To prove properties 2 and 3, we note that a process P_i can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag [j] == true and turn == j; this loop is the only one possible. If P_j is not ready to enter the critical section, then flag [j] == false, and P_i can enter its critical section. If Pj has set flag [j] to true and is also executing in its while statement, then either turn == i or turn == j . If turn == i, then P_i will enter the critical section. If turn == j, then Pj will enter the critical section. However, once P_j exits its critical section, it will reset flag[j] to false, allowing P_i to enter its critical section. If P_j resets flag [j] to true, it must also set turn to i. Thus, since P_i does not change the value of the variable *turn* while executing the while statement, P_i will enter the critical section (progress) after at most one entry by P_j(bounded waiting).

The Concepts of Semaphores and Monitors

- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations : wait() and signal().
- The wait() operation was originally termed P (from the Dutch proberen, “to test”).
- The signal() operation was originally called V (from verhogen, “to increment”).
- The definition of wait() is as follows :

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

- The definition of signal() is as follows:

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

- 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.

Monitors

- A monitor is a high-level synchronization construct.

Introduction to Deadlocks

- A set of processes is in a deadlocked 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.
- The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example, files, semaphores, and monitors).
- To illustrate a deadlocked state, consider a system with three CD RW drives. Suppose each of three processes holds one of these CD RW drives. If each process now requests another drive, the three processes will be in a deadlocked state. Each is waiting for the event “CD RW is released”, which can be caused only by one of the other waiting processes. This example illustrates a deadlock involving the same resource type.

- Deadlocks may also involve different resource types. For example, consider a system with one printer and one DVD drive. Suppose that process P_i is holding the DVD and process P_j is holding the printer. If Pi requests the printer and P_j requests the DVD drive, a deadlock occurs.

Deadlock Characterization

- In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting. Before we discuss the various methods for dealing with the deadlock problem, we look more closely at features that characterize deadlocks.

Necessary Conditions

- A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. Mutual Exclusion

- At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. Hold and Wait

- A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

3. No Preemption

- Resources cannot be preempted.; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. Circular Wait

- A set $\{P_0, P_1, ..., P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2,……, P_{n-1} is waiting for a resource held by $P_{n,}$ and P_n is waiting for a resource held by $P_{0.}$
- We emphasize that all four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent.

Memory Management and File Systems

- Basic concepts of memory management
- Paging
- Segmentation
- Virtual memory, demand paging
- Page replacement
- Introduction to file system management and directory structure
- File system mounting
- Disk scheduling

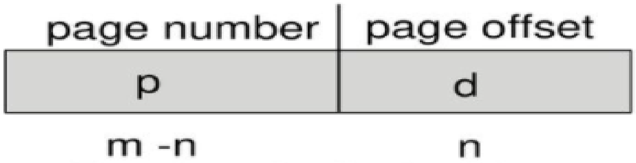
Paging

- Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous.
- Paging avoids external fragmentation and the need for compaction. It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store; most memory-management schemes used before the introduction of paging suffered from this problem. The problem arises because, when some code fragments or data residing in main memory need to be swapped out, space must be found on the backing store. The backing store also has the fragmentation problems discussed in connection with main memory; except that access is much slower, so compaction is impossible. Because of its advantages over earlier methods, paging in its various forms is commonly used in most operating systems. Traditionally, support for paging has been handled by hardware. However, recent designs have implemented paging by closely integrating the hardware and operating system, especially on 64-bit microprocessors.

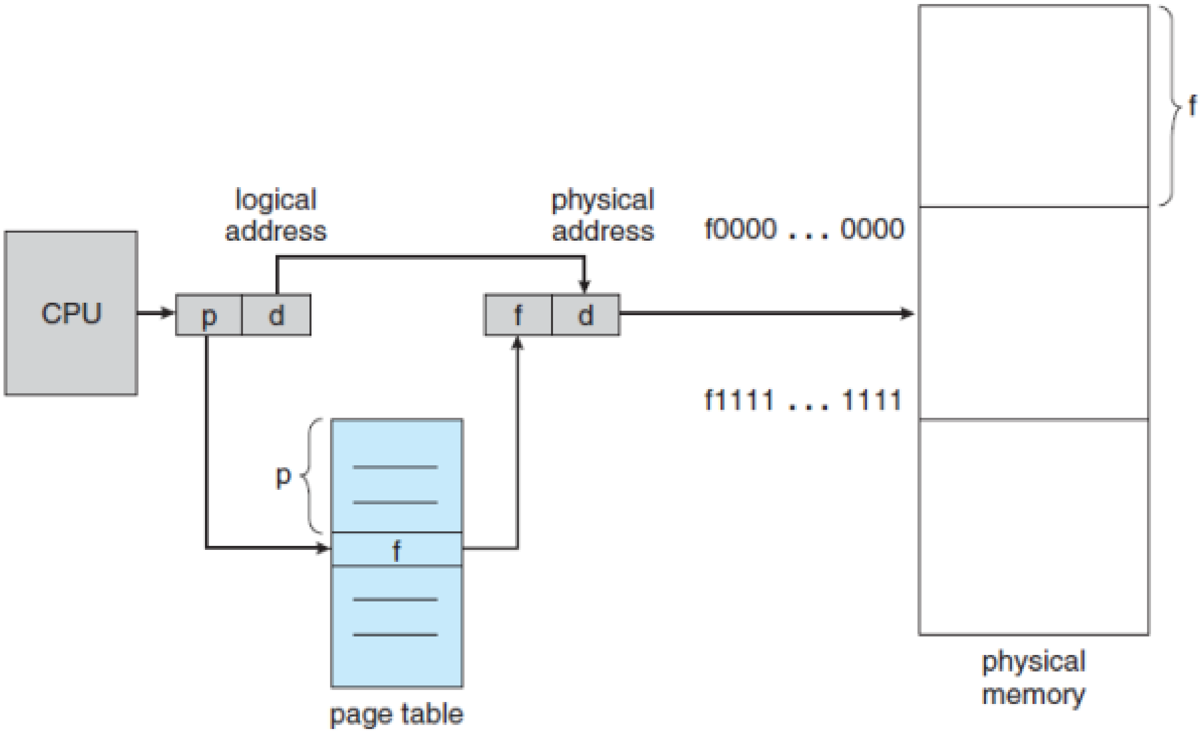
Basic Method

- The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called *frames* and breaking logical memory into blocks of the same size called pages. When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.
- The hardware support for paging is illustrated in the figure. Every address generated by the CPU is divided into two parts: a *page number (p)* and a *page offset (d).* The page number is used as an index into a page table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. The paging model of memory is shown in the figure.

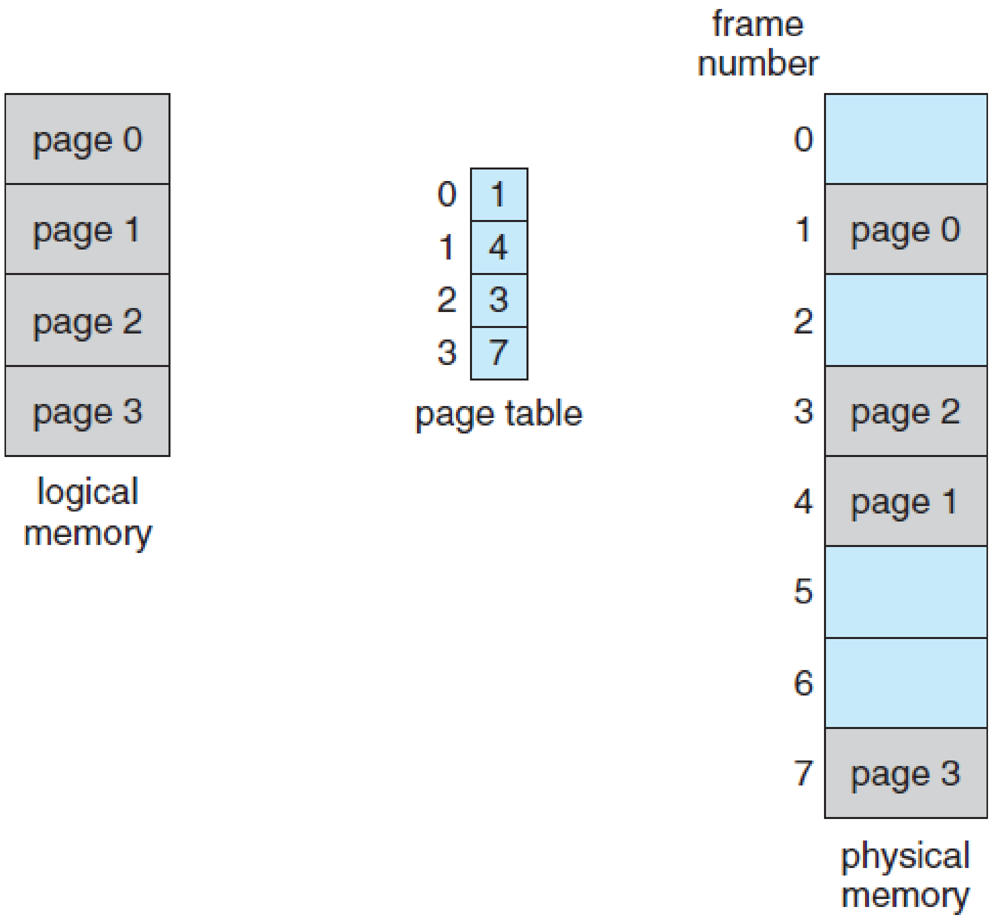
- The page size (like the frame size) is defined by the hardware. *The size of a page is typically a power of 2*, varying between 512 bytes and 16 MB per page, depending on the computer architecture. *The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy.* If the size of logical address space is 2^m and a page size is 2^n addressing units (bytes or words), then the high-order $m - n$ bits of a logical address designate the *page number*, and the n low-order bits designate the *page offset*. Thus, the logical address is as follows, where p is an index into the page table and d is the displacement within the page.



Paging Hardware

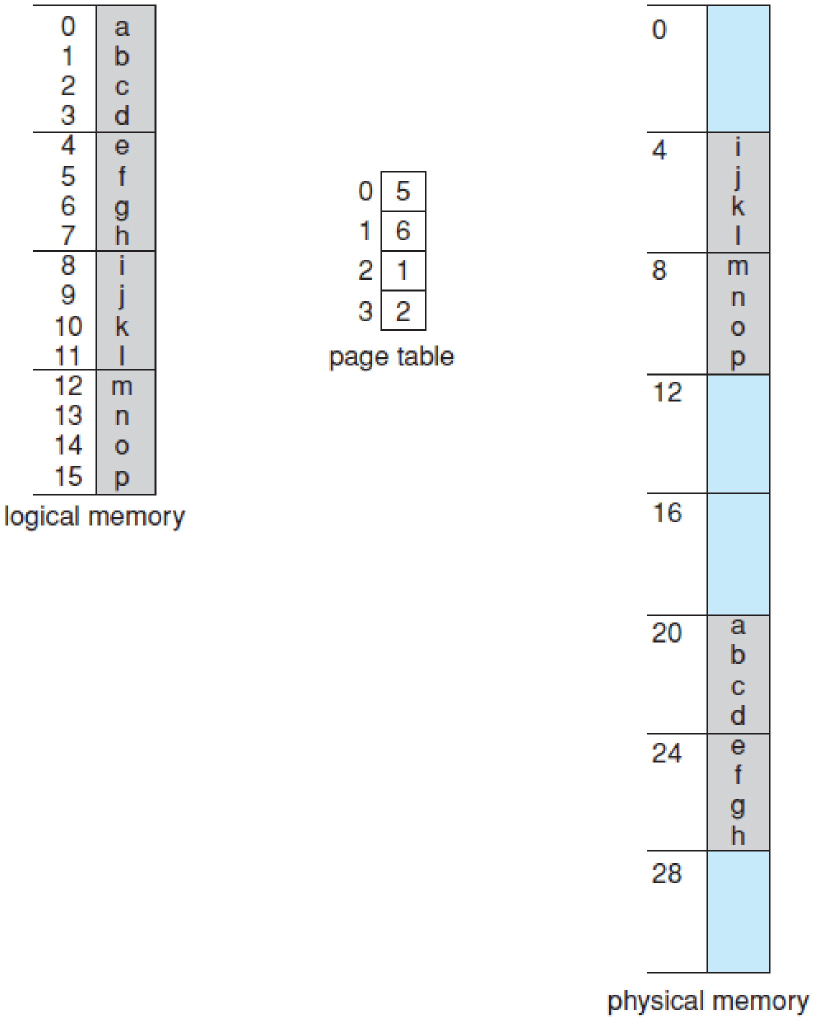


Paging hardware



- As a concrete (although minuscule) example, consider the memory in the next Figure. Here, in the logical address, n=2 and m=4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the *user's view of memory can be mapped into physical memory*.
- Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 x 4) + 0]. Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5x4) + 3]. Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6x4) + 0]. Logical address 13 maps to physical address 9.
- You may have noticed that *paging* itself is a form of *dynamic relocation*. Every *logical address* is bound by the paging hardware to some *physical address*. Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory.

Paging example for a 32-byte memory with 4-byte pages



- When we use a *paging scheme*, we have *no external fragmentation*: Any free frame can be allocated to a process that needs it.
- However, *we may have some internal fragmentation*. Notice that frames are allocated as units. If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full. For example, if page size is 2,048 bytes, a process of 72,766 bytes would need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in an internal fragmentation of 2,048 - 1,086 = 962 bytes. In the worst case, a process would need n pages plus 1 byte. It would be allocated n + 1 frames, resulting in an internal fragmentation of almost an entire frame.

- If *process size* is independent of *page size*, we expect *internal fragmentation* to average one-half page per process. This consideration suggests that *small page sizes are desirable*.
- However, *overhead* is involved in *each page-table entry*, and this overhead is reduced as the size of the pages increases. Also, *disk I/O is more efficient when the amount of data being transferred is larger*. Generally, page sizes have grown over time as processes, data sets, and main memory have become larger. Today, pages typically are between *4 KB and 8 KB* in size, and some systems support even larger page sizes. Some CPUs and kernels even support *multiple page sizes*. For instance, Solaris uses page sizes of 8 KB and 4 MB, depending on the data stored by the pages. Researchers are now developing support for *variable on-the-fly page-size*.
- Usually, each page-table entry is 4 bytes long, but that size can vary as well. A 32-bit entry can point to one of 2^{32} physical page frames. If frame size is 4 KB, then a system with 4-byte entries can address 2^{44} bytes (or 16 TB) of physical memory.

- When a process arrives in the system to be executed, its size, expressed in pages, is examined. *Each page of the process needs one frame*. Thus, if the process requires n pages, at least n frames must be available in memory. If n frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the *page table* for this process. The next page is loaded into another frame, and its frame number is put into the page table, and so on.
- *An important aspect of paging is the clear separation between the user's view of memory and the actual physical memory.* The user program views memory as one single space, containing only this one program. In fact, the *user program is scattered throughout physical memory, which also holds other programs*. The difference between the *user's view of memory* and the *actual physical memory* is reconciled by the *address-translation hardware*. The logical addresses are translated into physical addresses. This *mapping* is hidden from the user and is controlled by the operating system.

- Notice that the user process by definition is unable to access memory it does not own. It has no way of addressing memory outside of its page table, and the table includes only those pages that the process owns.
- Since the ***operating system*** is managing physical memory, it must be aware of the allocation details of physical memory - ***which frames are allocated, which frames are available, how many total frames there are***, and so on. This information is generally kept in a data structure called a ***frame table***. The frame table has one entry for each physical page frame, indicating whether the latter is ***free*** or ***allocated*** and, if it is allocated, to which page of which process or processes.

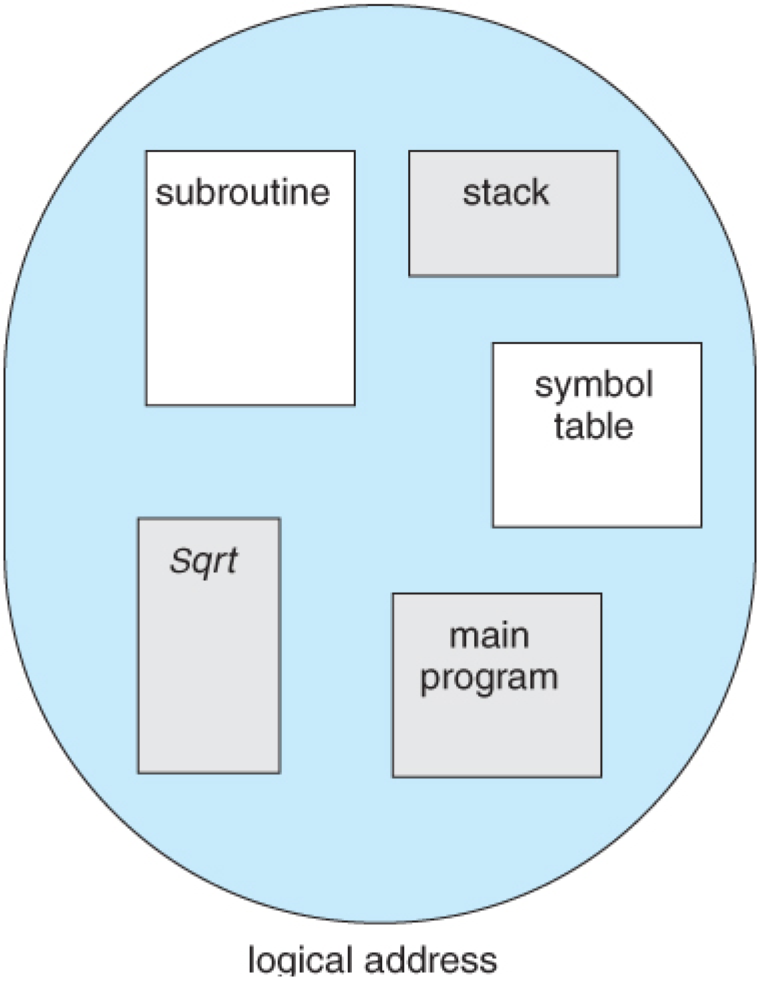
Segmentation

- An important aspect of memory management that became unavoidable with paging is the ***separation of the user's view of memory and the actual physical memory***. As we have already seen, the user's view of memory is not the same as the actual physical memory.
- ***The user's view is mapped onto physical memory.*** This mapping allows differentiation between ***logical memory*** and ***physical memory***.

Basic Method

- Do users think of memory as a linear array of bytes, some containing instructions and others containing data? Most people would say no. Rather, ***users prefer to view memory as a collection of variable-sized segments, with no necessary ordering among segments*** (See the next Figure).

User's view of a program



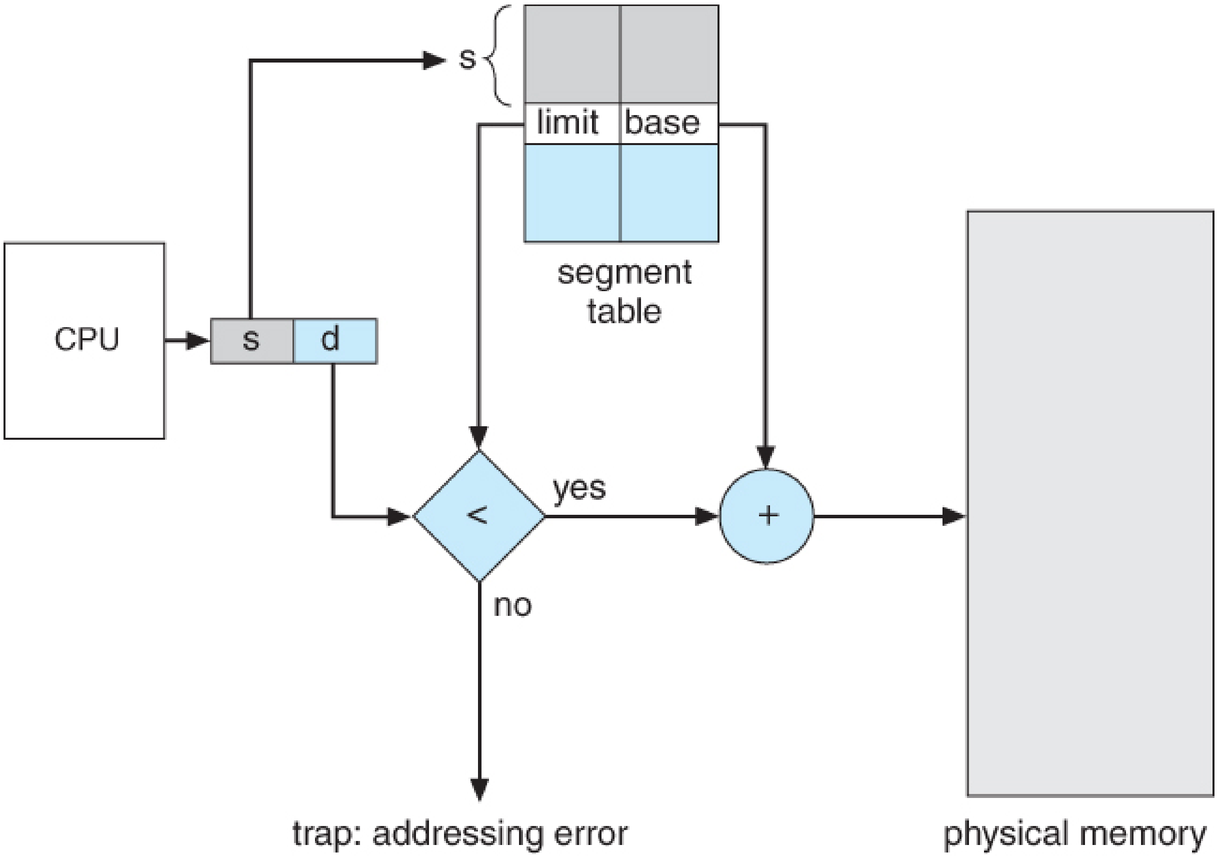
- Consider how you think of a program when you are writing it. You think of it as a main program with a set of methods, procedures, or functions. It may also include various data structures: objects, arrays, stacks, variables, and so on. Each of these ***modules*** or ***data elements*** is referred to by ***name***. You talk about "the stack," "the math library," "the main program," without caring ***what addresses in memory these elements occupy .***You are not concerned with whether the stack is stored before or after the sqrt () function. Each of these segments is of variable length; the length is intrinsically defined by the purpose of the segment in the program. ***Elements within a segment are identified by their offset from the beginning of the segment***: the first statement of the program, the seventh stack frame entry in the stack, the fifth instruction of the sqrt (), and so on.

- ***Segmentation*** is a memory-management scheme that supports this user view of memory. ***A logical address space is a collection of segments***.
- ***Each segment has a name and a length.*** The ***addresses specify both the segment name and the offset within the segment***. The user therefore specifies each address by two quantities: a segment name and an offset. (Contrast this scheme with the paging scheme, in which the user specifies only a single address, which is partitioned by the hardware into a page number and an offset, all invisible to the programmer.)
- For simplicity of implementation, ***segments are numbered*** and are referred to by a segment number, rather than by a segment name.
- Thus, a ***logical address*** consists of a two tuple:

  < *segment-number, offset* >.

- Normally, the user program is compiled, and the compiler automatically constructs segments reflecting the input program. A C compiler might create ***separate segments*** for the following:
  1. The code
  2. Global variables
  3. The heap, from which memory is allocated
  4. The stacks used, by each thread
  5. The standard C library
- Libraries that are linked in during compile time might be assigned separate segments. The loader would take all these segments and assign them segment numbers.

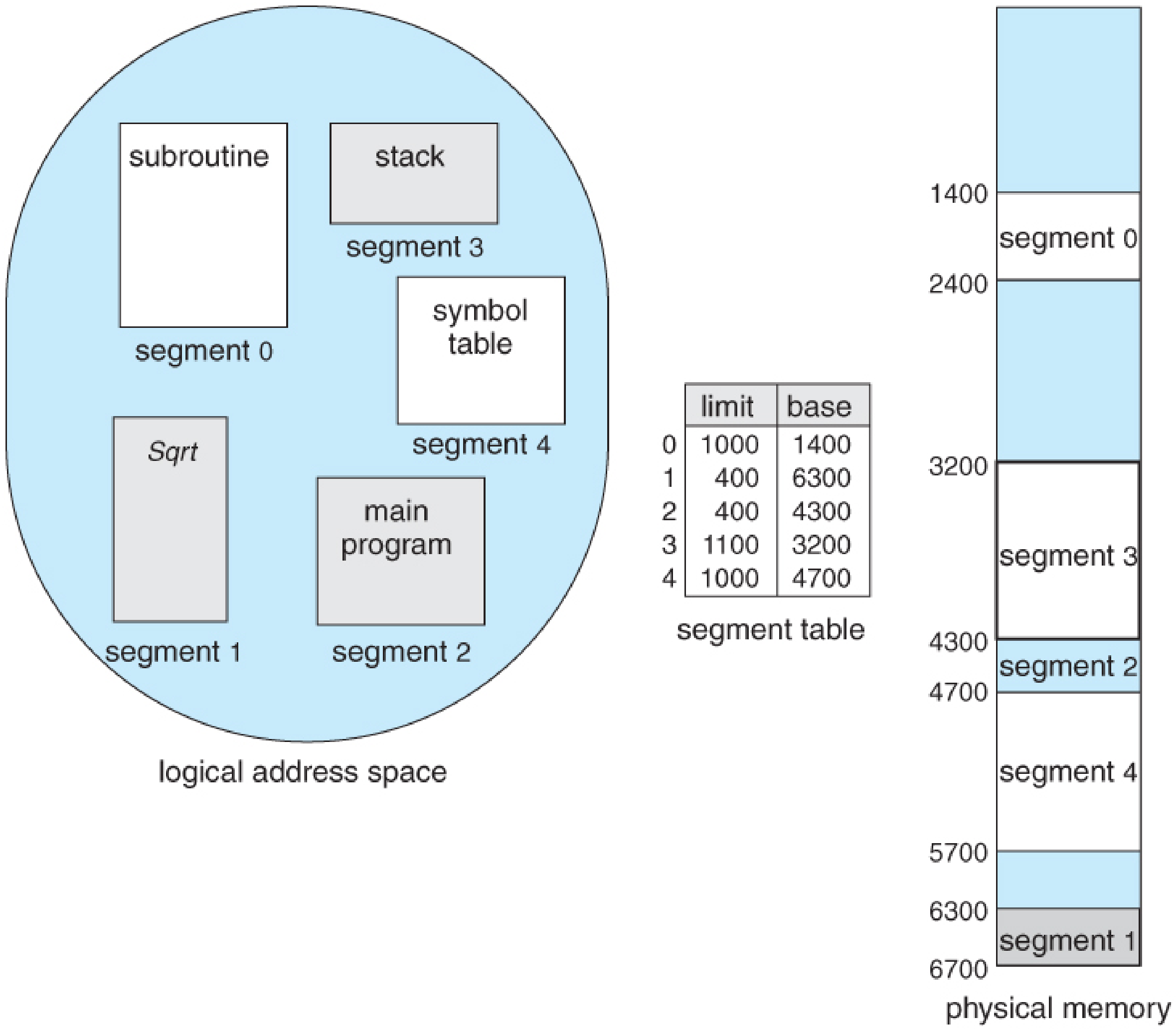
Segmentation Hardware



Hardware

- Although the user can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one-dimensional sequence of bytes. Thus, ***we must define an implementation to map two dimensional user-defined addresses into one-dimensional physical addresses***.
- This ***mapping*** is implemented through a ***segment table***. Each entry in the segment table has a ***segment base*** and a ***segment limit***. The segment base contains the ***starting physical address where the segment resides*** in memory, whereas the segment limit specifies the ***length of the segment***.
- The ***use of a segment table*** is illustrated in the Figure. ***A logical address consists of two parts: a segment number (s), and an offset into that segment (d).***
- The ***segment number*** is used as an ***index to the segment table***.
- The offset d of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond the end of a segment). ***When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte***. The segment table is thus essentially an array of base-limit register pairs.

Example of Segmentation



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

- As an example, consider the situation shown in the Figure.
- We have five segments numbered from 0 through 4.
- The segments are stored in physical memory as shown.
- The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit).
- For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353.
- A reference to byte 852 of segment 3, is mapped to 3200 (the base of segment 3) + 852 = 4052.
- A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.

Virtual Memory

Background

- ***The instructions being executed must be in physical memory.***
- Previous strategies require that an entire process should be in memory to execute it.
- There are cases where ***not entire program is needed*** to be placed in the physical memory :
  - Code to handle unusual error conditions is almost never executed.
  - Arrays, lists, and tables are often allocated more memory than they actually need.
  - Certain options and features of a program may be used rarely.
- Even in those cases where the entire program is needed, it may not all be needed at the same time.
- The ***ability to execute a program that is only partially placed in memory*** would offer many ***benefits*** :
- ***The program would no longer be constrained by the amount of physical memory that is available***. Users would be able to write programs for an extremely large virtual address space, simplifying the programming task.
- ***Because each user program could take less physical memory, more programs could be run at the same time***, with a corresponding increase in CPU utilization and throughput but with no increase in response time or turnaround time.
- ***Less I/O would be needed*** to load or swap user programs into memory, so each user program would run faster.
- Thus, ***running a program that is not entirely in memory would benefit both the system and the user.***

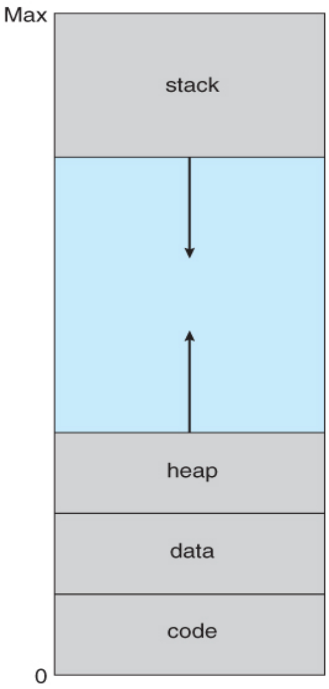
- ***Virtual memory involves the separation of logical memory as perceived by users from physical memory***.
- It allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.
- Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available. One can concentrate instead on the problem to be programmed.
- The ***virtual address space*** of a process refers to the logical (or virtual) view of how a process is stored in memory. Typically, this view is that a process begins at a certain logical address – say, address 0 – and exists in contiguous memory.

- In fact, the physical memory may be organized in page frames and the physical page frames assigned to a process may not be contiguous.
- It is up to the memory management unit (MMU) to map logical pages to physical page frames in memory.
- In the next Figure, we allow for the heap to grow upward in memory as it is used for dynamic memory allocation. Similarly, we allow for the stack to grow downward in memory through successive function calls. The large blank space (or hole) between the

heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows. Virtual address spaces that include holes are known as sparse address spaces.

- In addition to separating logical memory from physical memory, virtual memory allows files and memory to be shared by two or more processes through page sharing.

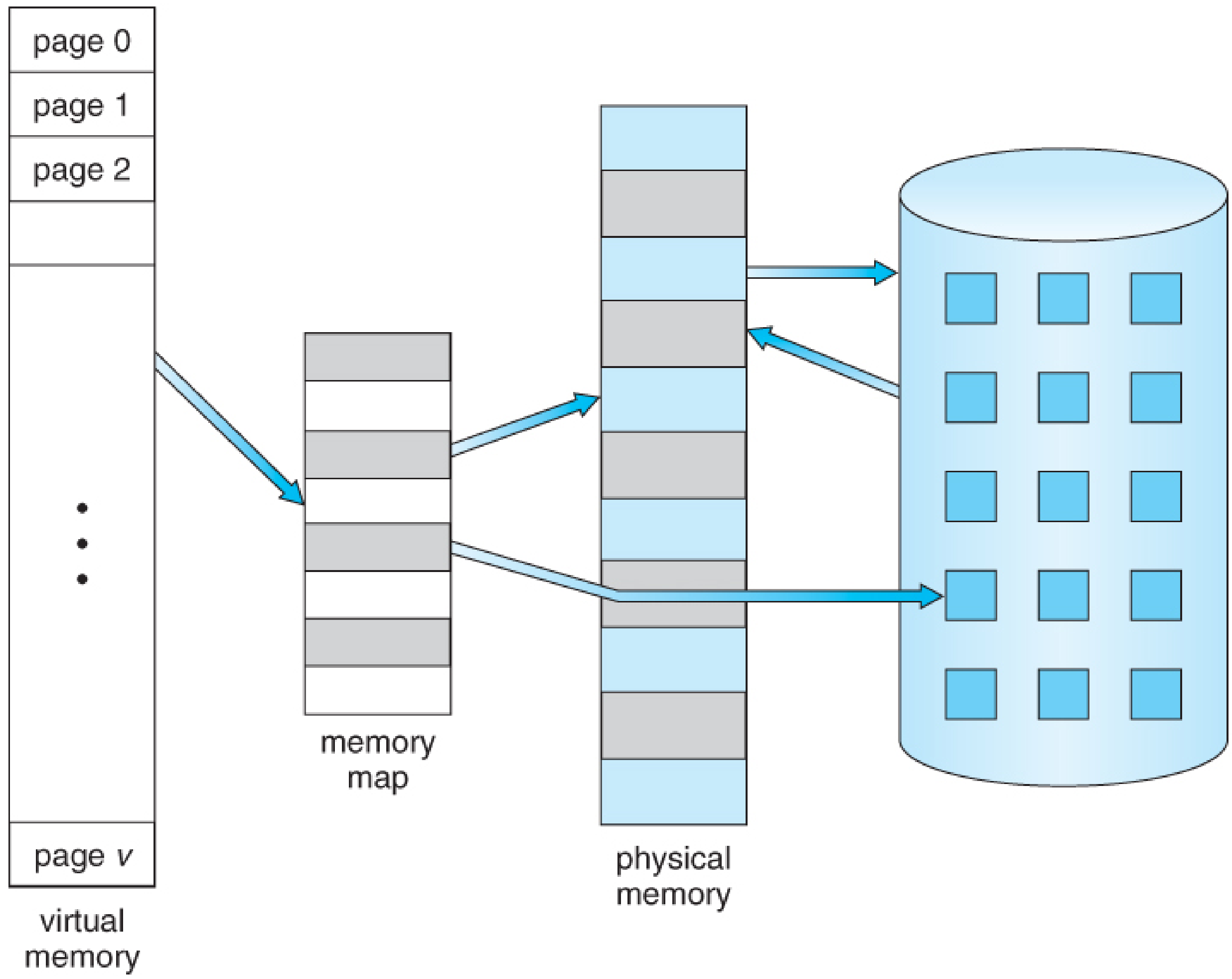
Figure shows ***virtual address space***, which is the programmers logical view of process memory storage. The actual physical layout is controlled by the process's page table.
Note that the address space shown in the figure is ***sparse*** - A great hole in the middle of the address space is never used, unless the stack and/or the heap grow to fill the hole.



- Virtual memory also allows the sharing of files and memory by multiple processes, with several benefits:
  - System libraries can be shared by mapping them into the virtual address space of more than one process.
  - Processes can also share virtual memory by mapping the same block of memory to more than one process.
  - Process pages can be shared during a fork( ) system call, eliminating the need to copy all of the pages of the original (parent) process.

Virtual Memory

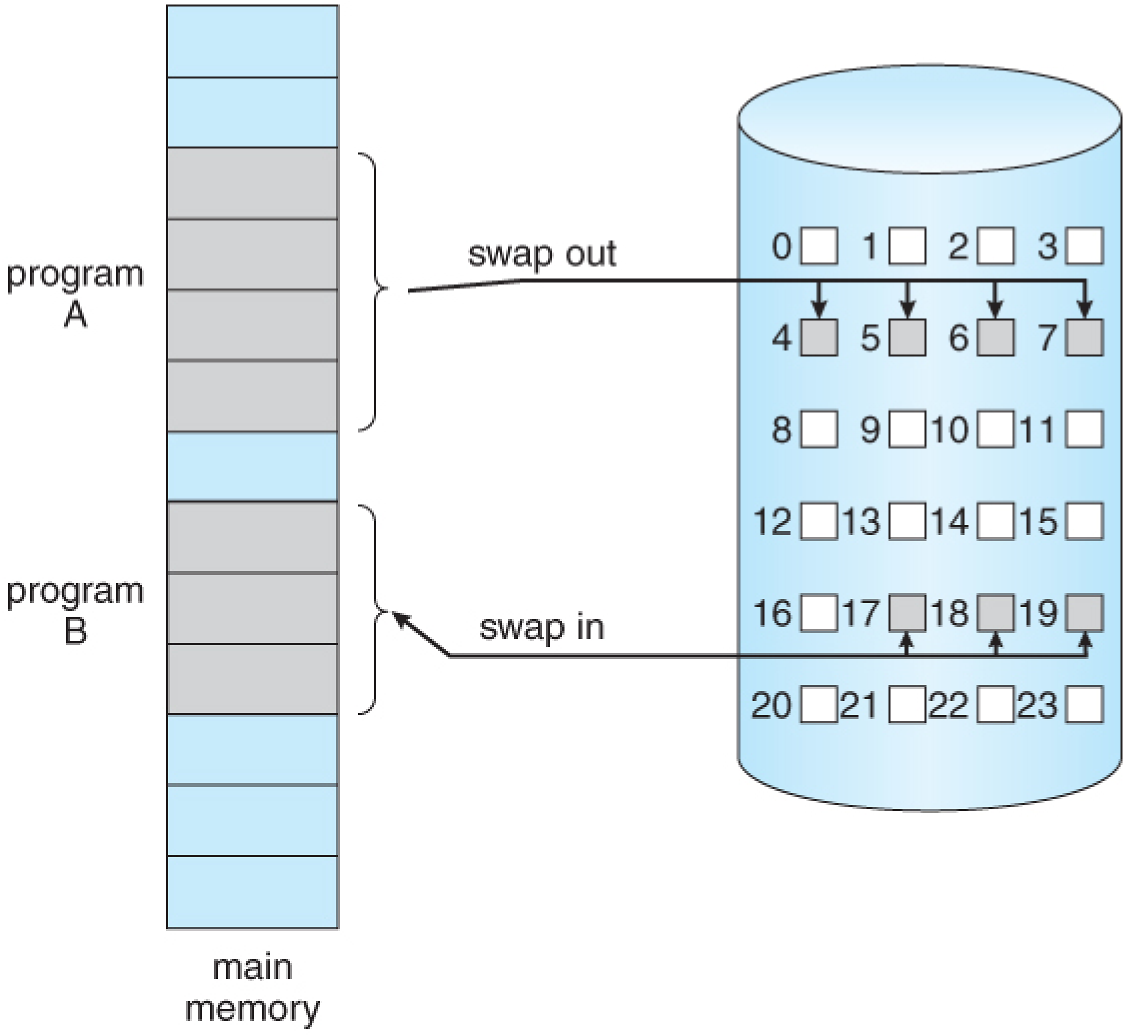
Diagram showing virtual memory that is larger than physical memory



<u>Demand Paging</u>

- A ***demand-paging system*** is similar to ***a paging system with swapping*** where processes reside in secondary memory (usually a disk).
- When we want to execute a process, we swap it into memory.
- Rather than swapping the entire process into memory, however, we use a **lazy swapper**.
- A ***lazy swapper*** never swaps a page into memory unless that page will be needed.
- Since we are now viewing a process as a sequence of pages, rather than as one large contiguous address space, use of the term ***swapper*** is technically incorrect. A swapper manipulates the entire process, whereas a ***pager*** is concerned with the individual pages of a process. We thus use ***pager***, rather than ***swapper***, in connection with ***demand paging***.

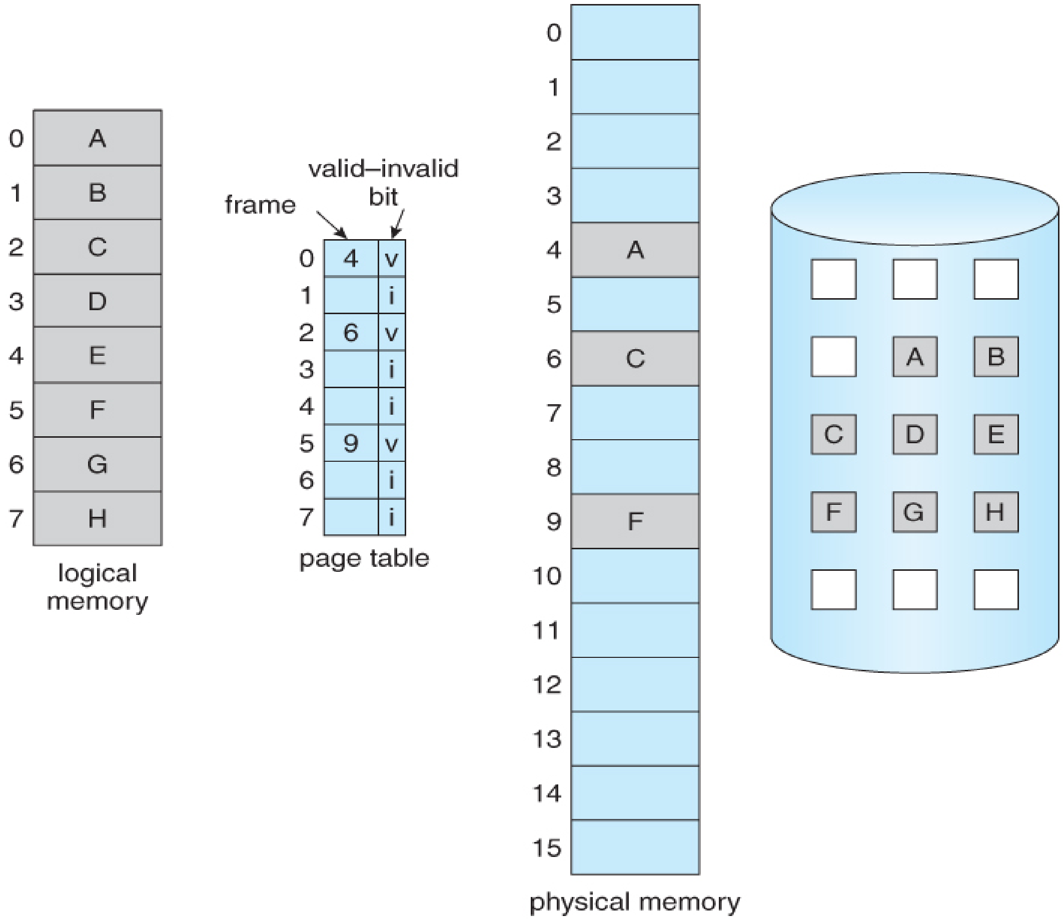
Transfer of a paged memory to contiguous disk space



Demand Paging

Basic Concepts

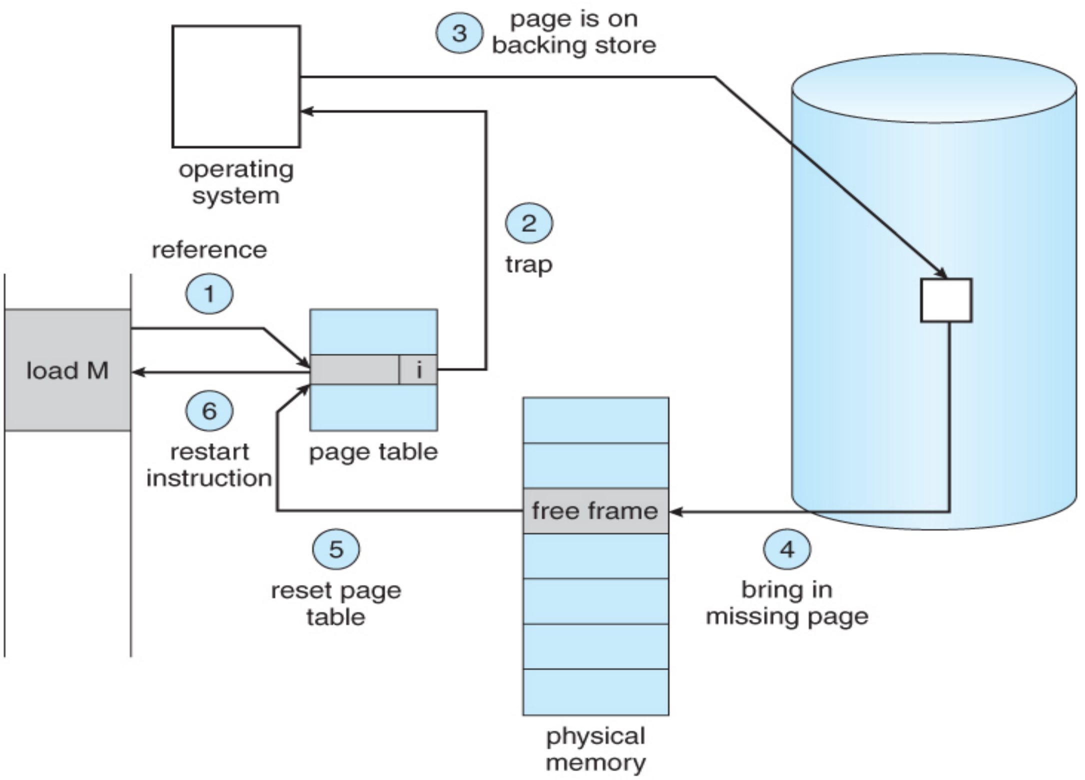
- When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again.
- Instead of swapping in a whole process, the pager brings only those pages into memory.
- Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.
- With this scheme, we need some form of **hardware support** to distinguish between the ***pages that are in memory*** and the ***pages that are on the disk***.
- The ***valid-invalid bit scheme*** can be used for this purpose.
- This time, however, when this bit is set to "***valid***", the ***associated page*** is both ***legal*** and ***in memory***.
- If the bit is set to "***invalid***", the ***page either is not valid*** (that is, ***not in the logical address space of the process***) or ***is valid but is currently on the disk.***
- The ***page-table entry*** for a page that is brought into memory is set as usual, but the ***page-table entry*** for a ***page that is not currently in memory*** is either simply marked ***invalid*** or ***contains the address of the page on disk***.
- This situation is depicted in the next Figure.
- Notice that marking a page invalid will have no effect if the process never attempts to access that page.



- Hence, if we guess right and page in all and ***only those pages*** that are ***actually needed***, the process will run exactly as though we had brought in all pages.
- While the process executes and accesses pages that are ***memory resident***, execution proceeds normally.
- But what happens if the process tries to access a page that was not brought into memory ?
- Access to a page marked invalid causes a page fault.
- The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory.
- The procedure for handling the page fault is shown next.

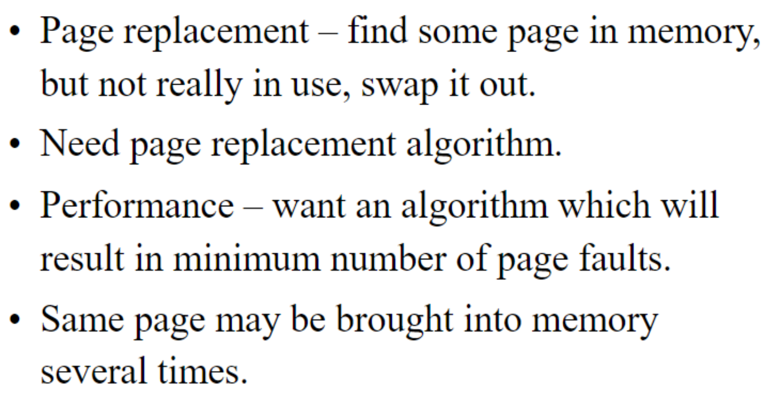
Steps in handling a page fault

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example).
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.



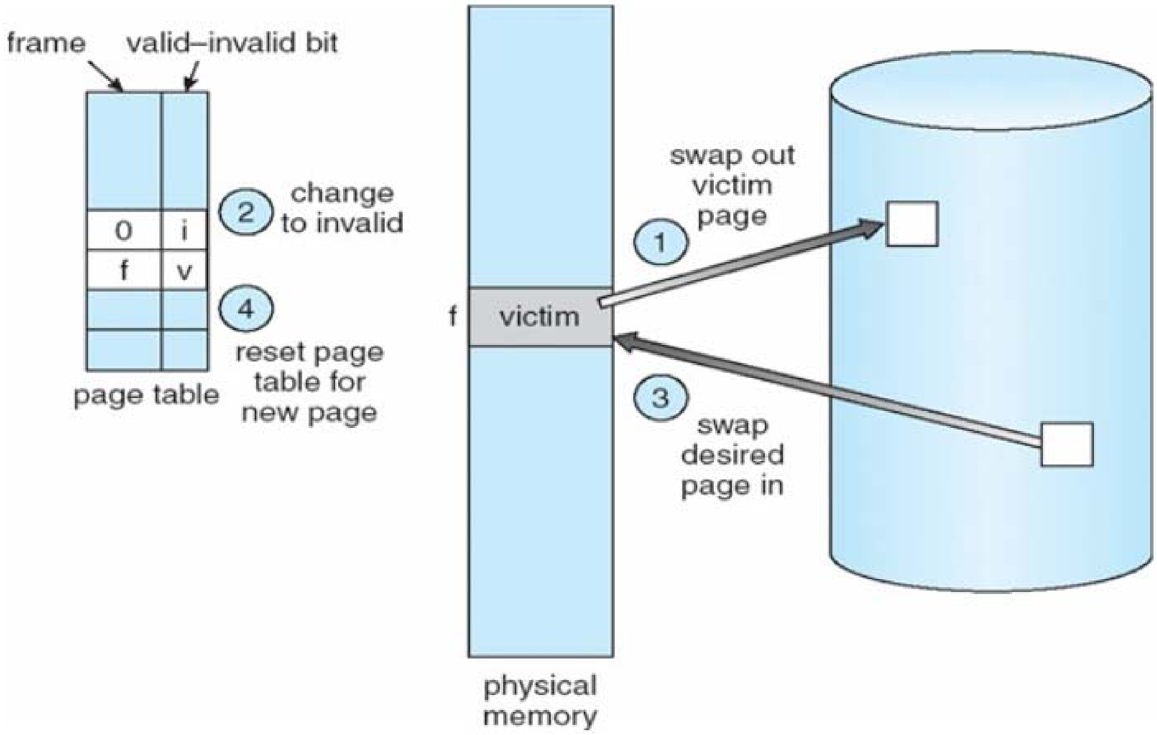
- In an extreme case, NO pages are swapped in for a process until they are requested by page faults. This is known as ***pure demand paging.***
- In theory each instruction could generate multiple page faults. In practice this is very rare, due to ***locality of reference***.
- The hardware necessary to support virtual memory is the same as for paging and swapping: A page table and secondary memory (***Swap space***).
- A crucial part of the process is that the instruction must be restarted from scratch once the desired page has been made available in memory. For most simple instructions this is not a major difficulty. However there are some architectures that allow a single instruction to modify a fairly large block of data, (which may span a page boundary), and if some of the data gets modified before the page fault occurs, this could cause problems. One solution is to access both ends of the block before executing the instruction, guaranteeing that the necessary pages get paged in before the instruction begins.

What happens if there is no free frame ?



- Page replacement – find some page in memory, but not really in use, swap it out.
- Need page replacement algorithm.
- Performance – want an algorithm which will result in minimum number of page faults.
- Same page may be brought into memory several times.

Steps in Handling Page Replacement



1. Find the location of the desired page on disk.
2. Find a free frame:
   - If there is a free frame, use it.
   - If there is no free frame, use a page replacement algorithm to select a victim page.
3. Bring the desired page into the (newly) free frame; update the page and frame tables.
4. Restart the process.

Page Replacement

Page Replacement Algorithms

- FIFO Page Replacement
- Optimal Page Replacement
- LRU Page Replacement

- If we trace a particular process, we might record the following address sequence :

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

At 100 bytes per page, this sequence is reduced to the following reference string :

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

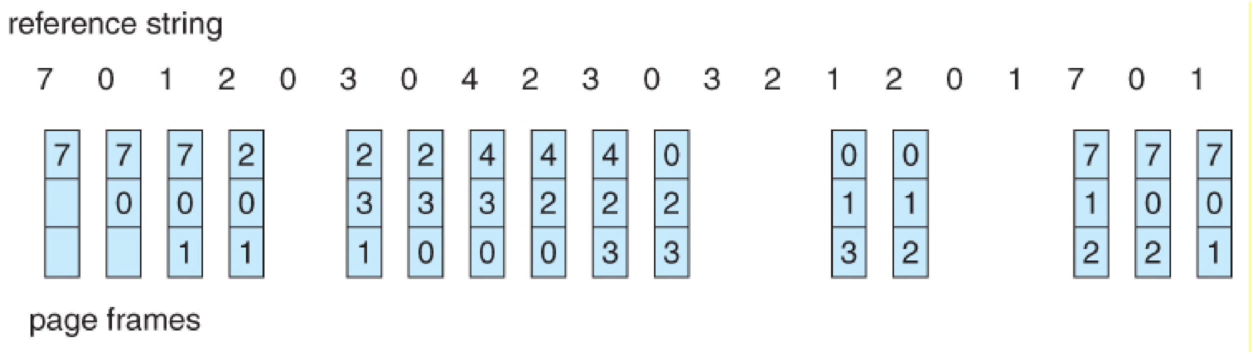
- To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the number of page frames available.

FIFO Page Replacement

- The simplest page-replacement algorithm is first-in, first-out (FIFO) algorithm.
- A FIFO page replacement algorithm associates with each page the time when that page was brought into memory.
- When a page must be replaced, the oldest page is chosen.
- Notice that it is not strictly necessary to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

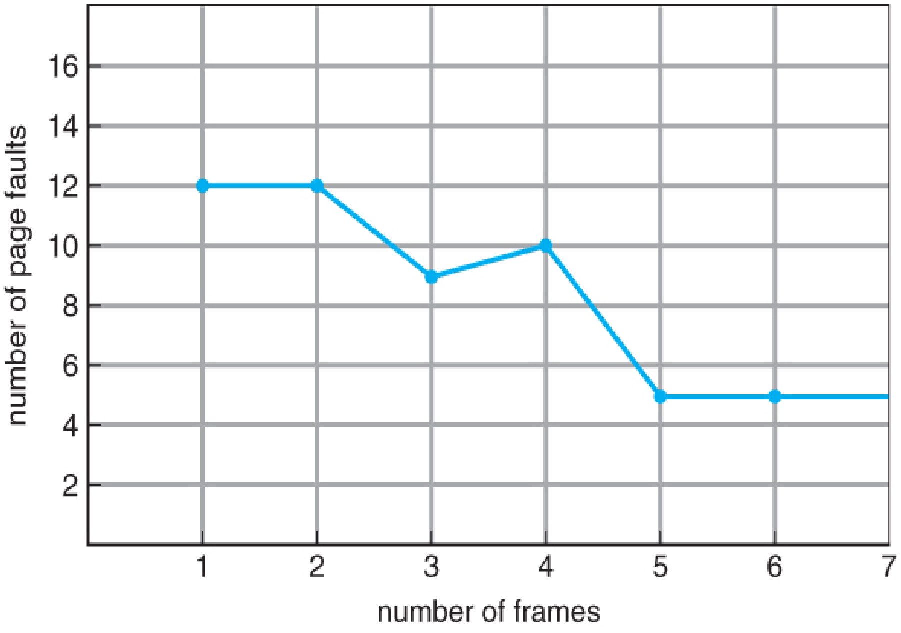
- Consider the following reference string for a memory with three frames.

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1



Total number of page faults :15

Page-fault curve for FIFO page replacement on a given reference string

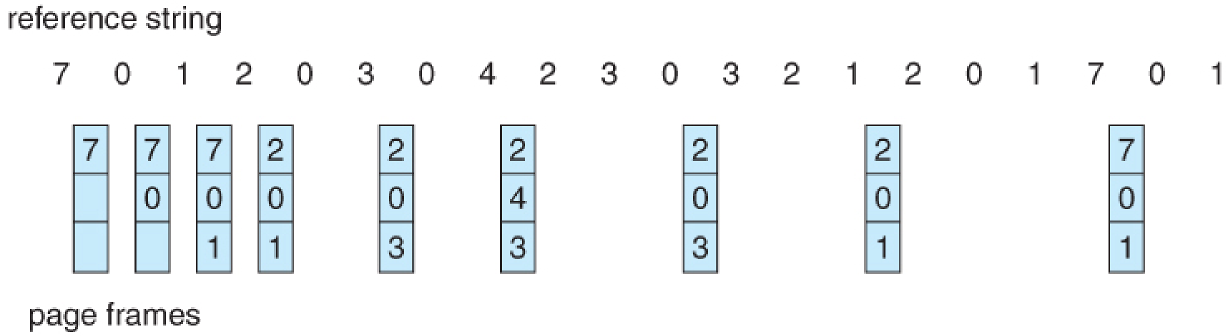


For some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases. This unexpected result is known as *Belady’s anomaly*.

Optimal Page Replacement Algorithm

- The optimal page replacement algorithm has the lowest page-fault rate of all algorithms.
- This algorithm never suffers from the Belady’s anomaly.
- Also known as OPT or MIN.

Replace the page that will not be used for the longest period of time.



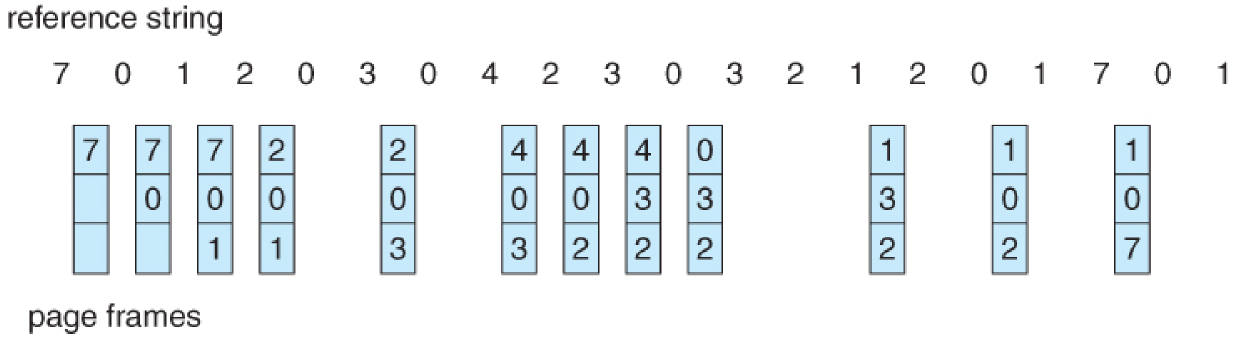
On our sample reference string, the optimal page-replacement algorithm would yield 9 page faults.

- The optimal page-replacement algorithm guarantees the lowest possible page-fault rate for a fixed number of frames.
- OPT is difficult to implement, because it requires future knowledge of the reference string.
- The optimal page-replacement algorithm is used mainly for comparison studies.

Least Recently Used (LRU) Page-Replacement Algorithm

- If the optimal page-replacement algorithm is not feasible, perhaps an approximation of the optimal algorithm is possible.
- The key distinction between the FIFO and OPT algorithms (other than looking backward versus forward in time) is that the FIFO algorithm uses the time when a page was brought into memory, whereas the OPT algorithm uses the time when a page is to be *used*.
- If we use the recent past as an approximation of the near future, then we can ***replace the page that has not been used for the longest period of time***.

- LRU replacement associates with each page the time of that page's last use.
- When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.
- We can think of this strategy as the optimal page-replacement algorithm looking backward in time.
- The result of applying LRU page-replacement to our example reference string is shown in the following figure :



Total number of page faults :12

Introduction to file system management and directory structure

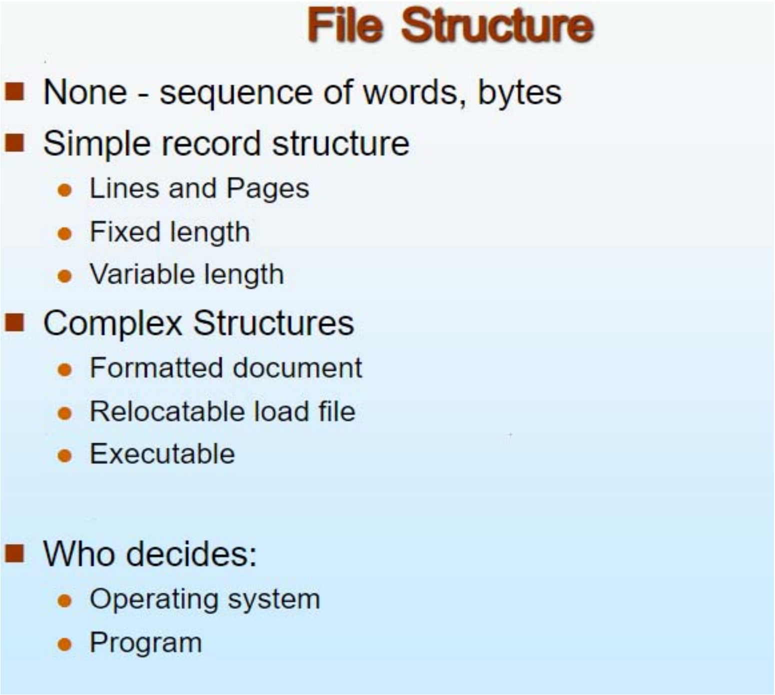
File attributes

File attributes may vary from one OS to another but typically consist of

- Name
- Identifier
- Type
- Location
- Size
- Protection

File Operations

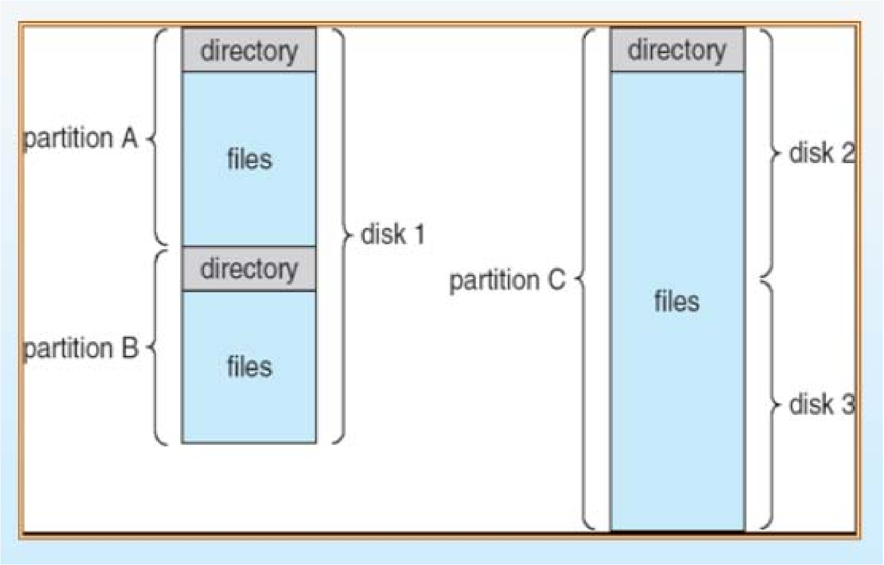
- Creating a file
- Writing a file
- Reading a file
- Repositioning within a file
- Deleting a file
- Truncating a file



File Structure

- None - sequence of words, bytes
- Simple record structure
  - Lines and Pages
  - Fixed length
  - Variable length
- Complex Structures
  - Formatted document
  - Relocatable load file
  - Executable
- Who decides:
  - Operating system
  - Program

A Typical File System Organization

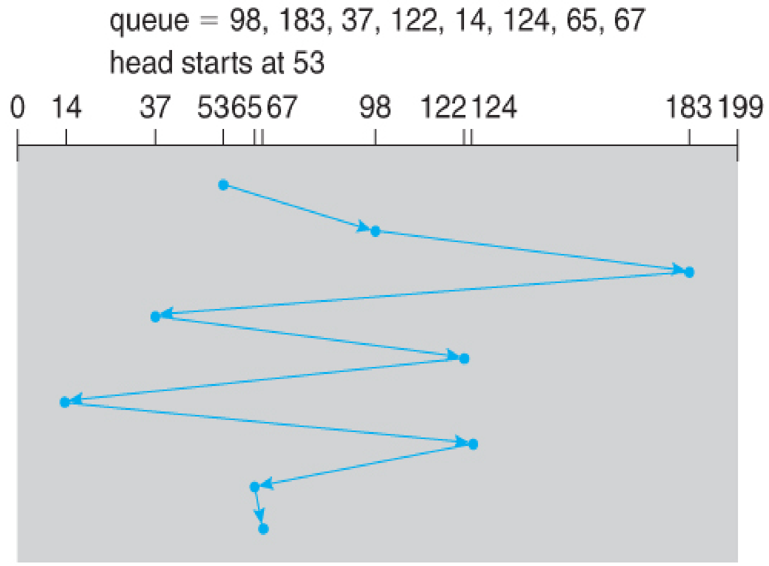


Disk Scheduling

- FCFS Scheduling
- SSTF Scheduling
- SCAN Scheduling
- C-SCAN Scheduling
- LOOK Scheduling
- C-LOOK Scheduling

FCFS Scheduling

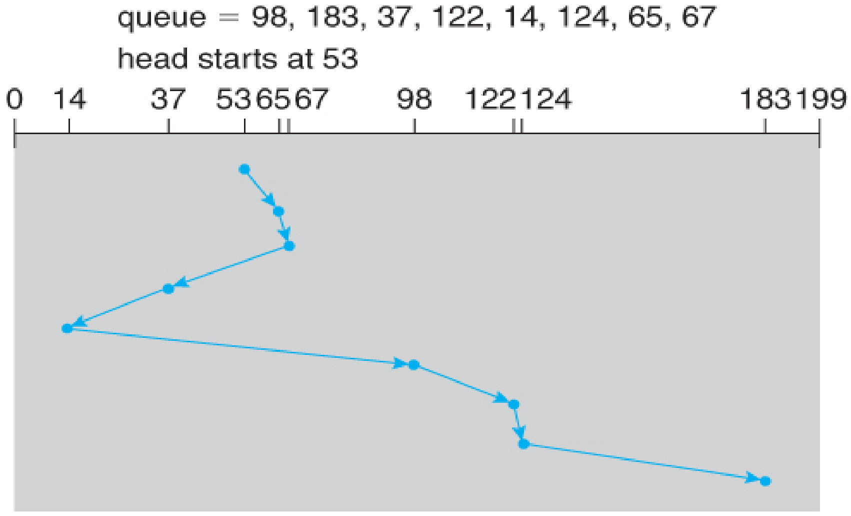
- The simplest form of disk scheduling is first-come, first-served (FCFS) algorithm.
- This algorithm generally does not provide the fastest service.
- Consider, for example, a disk queue with requests for I/O to blocks on cylinders
  98, 183, 37, 122, 14, 124, 65, 67
  in that order.
- If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for a total head movement of 640 cylinders.



Total head movement = (98-53) + (183-98) + (183-37) + (122-37) + (122-14)
+ (124-14) + (124-65) + (67-65)
= 640 cylinders.

Shortest-Seek-Time-First (SSTF) Disk Scheduling

- It seems reasonable to service all the requests close to the current head position before moving the head away to service other requests.
- The SSTF algorithm selects the request with the least seek time from the current head position.
- Since seek time increases with the number of cylinders traversed by the head, SSTF chooses the pending request closest to the current head position.
- For our example request queue, the closest request to the initial head position (53) is at cylinder 65. Once we are at cylinder 65, the next closest request is at cylinder 67.

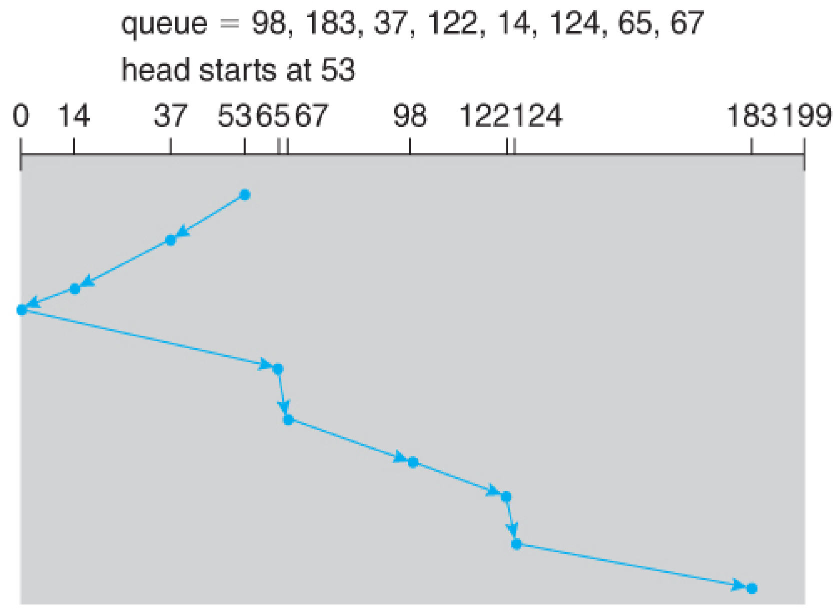


Total head movement = 236 cylinders

- SSTF gives a substantial improvement in performance.
- SSTF scheduling is essentially a form of shortest-job-first (SJF) scheduling.
- Like SJF scheduling, it may cause starvation of some requests.

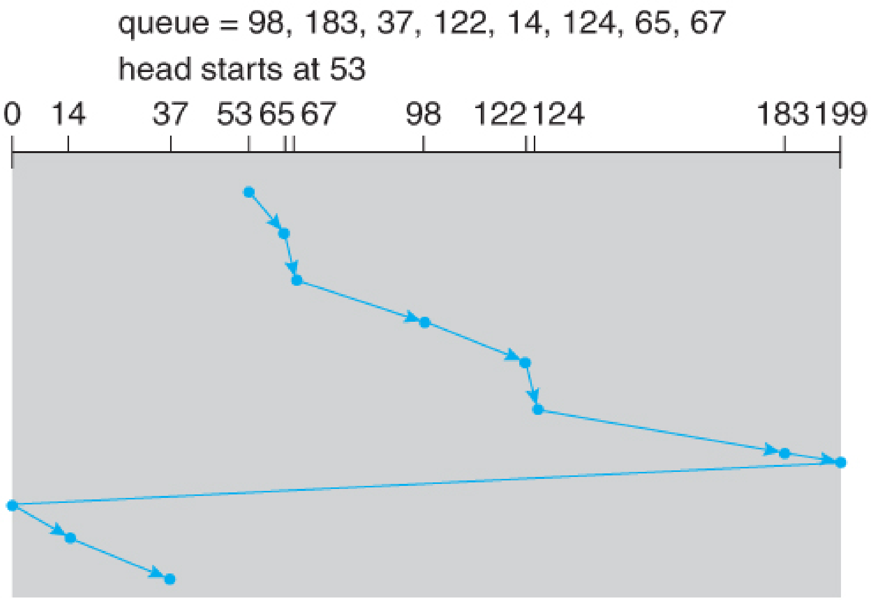
SCAN Disk Scheduling Algorithm

- In the SCAN disk scheduling algorithm, the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk.
- At the other end, the direction of head movement is reversed, and servicing continues.
- The head continuously scans back and forth across the disk.
- The SCAN algorithm is sometimes called the elevator algorithm, since the disk arm behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way.
- Let us apply the SCAN disk scheduling algorithm to the following disk queue with requests on cylinders 98, 183, 37, 122, 14, 124, 65, 67.
- We need to know the direction of head movement in addition to the head's current position.
- Assume that the disk arm is moving toward 0.
- Assume that the initial head position is 53.
- The head will next service 37 and then 14.
- At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183.
- If a request arrives in the queue just in front of the head, it will be serviced almost immediately. A request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses the direction, and comes back.



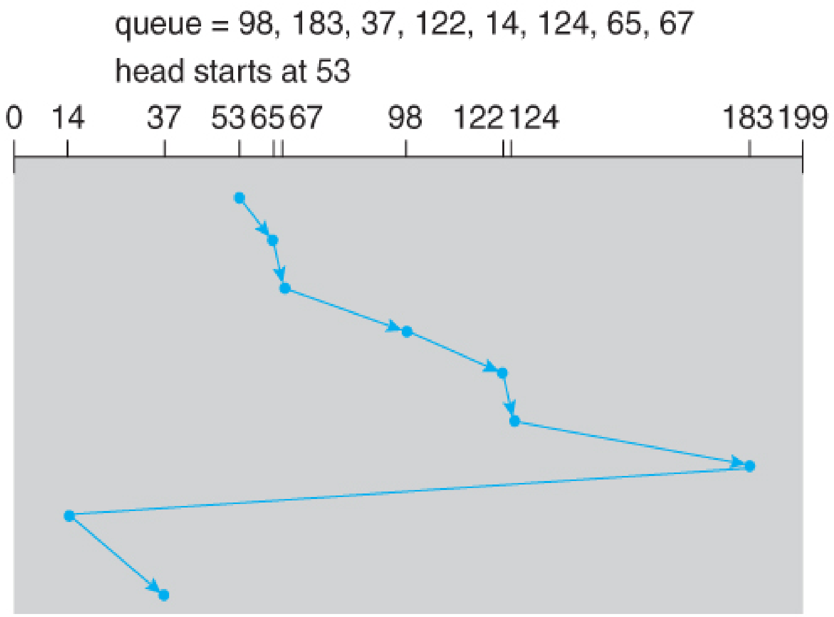
Circular SCAN (C-SCAN) scheduling

- C-SCAN scheduling is a variant of SCAN designed to provide a more uniform wait time.
- Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way.
- When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip.
- The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.



LOOK Disk Scheduling Algorithm

- Both SCAN and C-SCAN move the disk arm across the full width of the disk.
- In practice, neither algorithm is often implemented this way.
- More commonly, the arm goes only as far as the final request in each direction. Then, it reverses direction immediately, without going all the way to the end of the disk.
- Versions of SCAN and C-SCAN that follow this pattern are called LOOK and C-LOOK scheduling, because they look for a request before continuing to move in a given direction.


queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0
14
37
53
65
67
98
122
124
183
199