Sawp Instruction

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

Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key
- Solution:

```
do {
        key = TRUE;
        while ( key == TRUE)
                Swap (&lock, &key );

                //    critical section
        lock = FALSE;
                //      remainder section
} while (TRUE);
```

Bounded-waiting Mutual Exclusion with TestandSet()

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
```

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

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
    - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

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

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

- Counting semaphore – integer value can range over an unrestricted domain

- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
    - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

```
Semaphore mutex;    //  initialized to 1
do {
        wait (mutex);
      // Critical Section
    signal (mutex);
            // remainder section
} while (TRUE);
```

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the crtical section.
    - Could now have busy waiting in critical section implementation
        - But implementation code is short
        - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.
- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
    - value (of type integer)
    - pointer to next record in the list
- Two operations:

- block – place the process invoking the operation on the appropriate waiting queue.
- wakeup – remove one of processes in the waiting queue and place it in the ready queue.

- Implementation of wait:

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

- Implementation of signal:

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

Classical Problems of Synchronization

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

Bounded-Buffer Problem

The pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.
- The structure of the producer process

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

- The structure of the consumer process

```
do {
     wait (full);
     wait (mutex);
          //  remove an item from  buffer to nextc
     signal (mutex);
     signal (empty);
```

```
        //  consume the item in nextc
    } while (TRUE);
```

Readers-Writers Problem

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as ***readers*** *and to the latter as* ***writers****. Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously,*
chaos may ensue.

- A data set is shared among a number of concurrent processes
    - Readers – only read the data set; they do **not** perform any updates
    - Writers – can both read and write
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time
- Shared Data
    - Data set
    - Semaphore mutex initialized to 1
    - Semaphore wrt initialized to 1
    - Integer readcount initialized to 0
- The structure of a writer process

```
do {
  wait (wrt) ;
```

```
            //    writing is performed
            signal (wrt) ;
        } while (TRUE);
```

- The structure of a reader process

```
do {
            wait (mutex) ;
            readcount ++ ;
            if (readcount == 1)
                        wait (wrt) ;
            signal (mutex)

            // reading is performed
            wait (mutex) ;
            readcount  - - ;
            if (readcount  == 0)
                        signal (wrt) ;
            signal (mutex) ;
        } while (TRUE);
```

Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks). When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a

chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.

- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1
- The structure of Philosopher *i*:

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

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

             //  think
} while (TRUE);
```

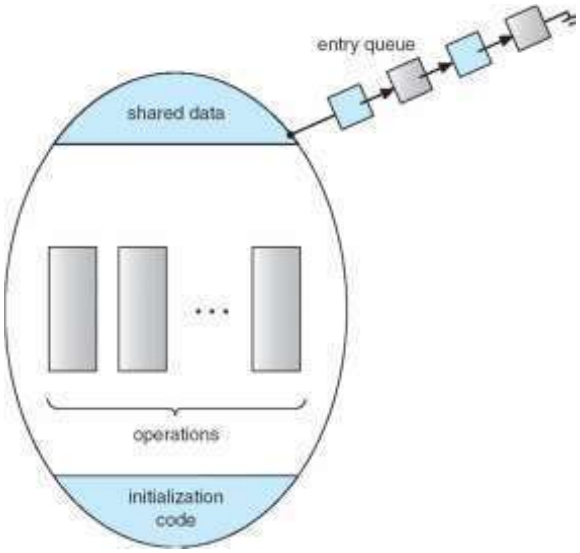
Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
        // shared variable declarations
        procedure P1 (…) { …. }
                …
        procedure Pn (…) {……}
    Initialization code ( ….) { … }
                …
        }
```

}

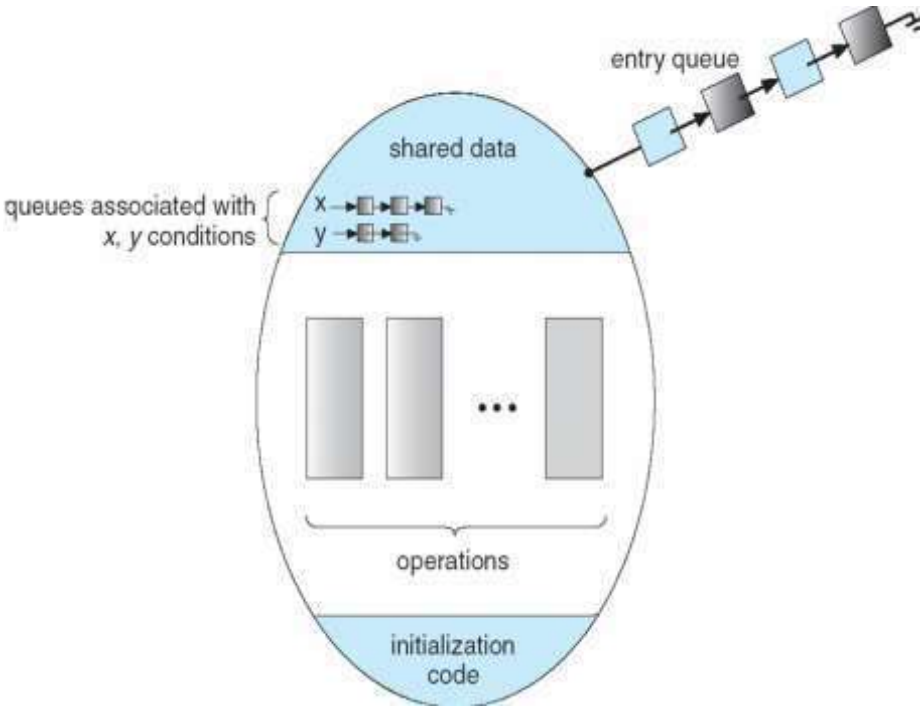
Schematic view of a Monitor



Condition Variables

- condition x, y;
- Two operations on a condition variable:
    - x.wait () – a process that invokes the operation is suspended.
        - x.signal () – resumes one of processes (if any) that invoked x.wait ()

Monitor with Condition Variables



Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;     // (initially  = 0)
int next-count = 0;
```

- Each procedure ***F*** will be replaced by

```
wait(mutex);
    …
            body of F;
    …
if (next_count > 0)
        signal(next)
else
        signal(mutex);
```

- Mutual exclusion within a monitor is ensured.

Monitor Implementation

For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially  = 0)
int x-count = 0;
```

The operation x.wait can be implemented as:

```
x-count++;
if (next_count > 0)
        signal(next);
else
        signal(mutex);
wait(x_sem);
x-count--;
```

The operation x.signal can be implemented as:

```
if (x-count > 0) {
```

```
            next_count++;
            signal(x_sem);
            wait(next);
            next_count--;
        }
```

Deadlock

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Example
    - System has 2 disk drives
    - P_1 and P_2 each hold one disk drive and each needs another one
- Example
    - semaphores *A* and *B*, initialized to 1

P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)

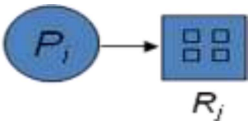
System Model

- Resource types R_1, R_2, . . ., R_m *(CPU cycles, memory space, I/O devices)*
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
    - **request**
    - **use**
    - **release**

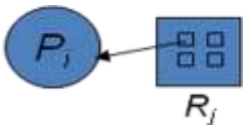
Deadlock Characterization

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_0 is waiting for a resource that is held by P_0.

Resource Allocation Graph

- A set of vertices V and a set of edges E.
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$
- Process 
- Resource Type with 4 instances 
- P_i requests instance of R_j 

- P_i is holding an instance of R_j



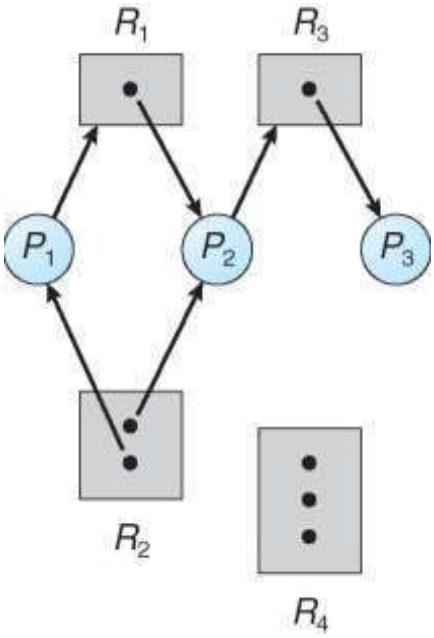


Fig: RAG

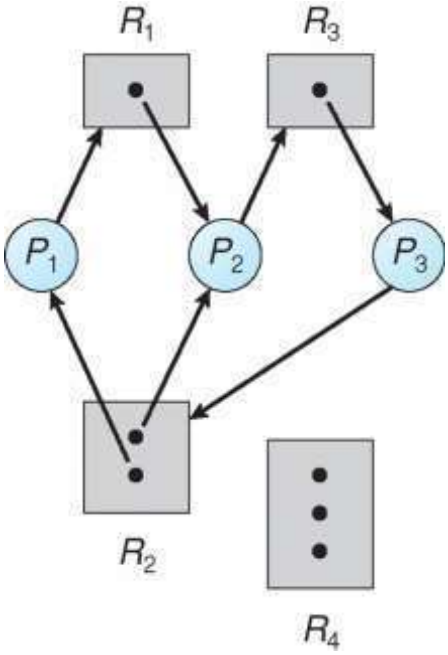


Fig: RAG with a deadlock

- If graph contains no cycles ⇒ no deadlock
- If graph contains a cycle ⇒
    - if only one instance per resource type, then deadlock
    - if several instances per resource type, possibility of deadlock

Methods for Handling Deadlock

- Ensure that the system will *never* enter a deadlock state
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

- **Mutual Exclusion –** not required for sharable resources; must hold for nonsharable resources

- **Hold and Wait –** must guarantee that whenever a process requests a resource, it does not hold any other resources
    - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none
    - Low resource utilization; starvation possible
- **No Preemption –**
    - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
    - Preempted resources are added to the list of resources for which the process is waiting
    - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait –** impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock Avoidance

- Requires that the system has some additional *a priori* information available
- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe state

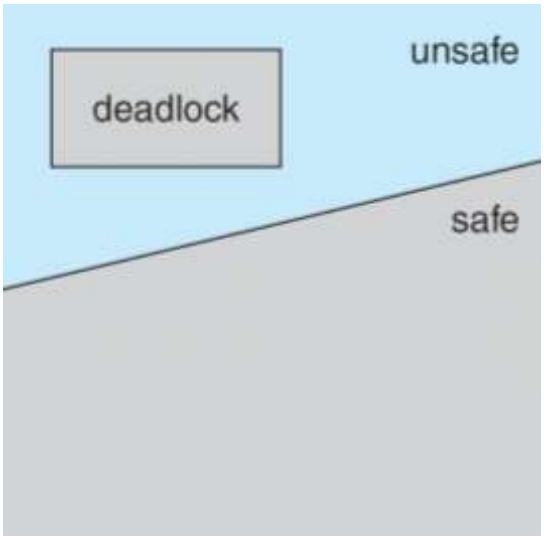
- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in safe state if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes is the systems such that for each P_i, the resources that P_i can still

request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$

- That is:
    - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
    - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
    - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Facts

- **If a system is in safe state ⇒ no deadlocks**
- **If a system is in unsafe state ⇒ possibility of deadlock**
- **Avoidance ⇒ ensure that a system will never enter an unsafe state.**

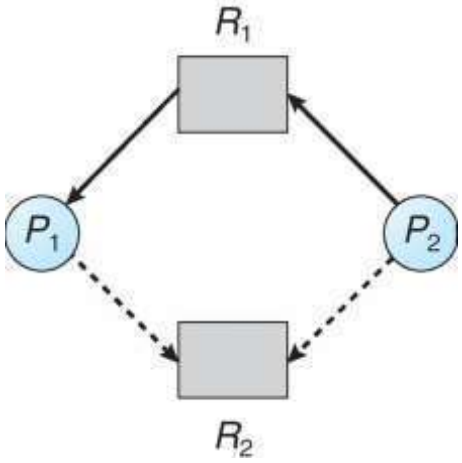


Deadlock Avoidance Algorithm

- Single instance of a resource type
    - Use a resource-allocation graph
- Multiple instances of a resource type
    - Use the banker's algorithm

RAG Scheme

- Claim edge $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system



Banker's Algorithm

Assumptions

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Data Structure for Bankers' Algorithm

Let n = number of processes, and m = number of resources types.

- **Available***:* Vector of length m. If available [j] = k, there are k instances of resource type R_j available
- **Max***: n x m* matrix. If *Max* [*i,j*] = k, then process P_i may request at most k instances of resource type R_j

- **Allocation**: n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j
- **Need**: n x m matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task

$$Need\ [i,j] = Max[i,j] - Allocation\ [i,j]$$

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize:

Work = *Available*

Finish [i] = *false* for i = 0, 1, …, n- 1

2. Find and i such that both:

(a) *Finish* [i] = *false*

(b) $Need_i \leq Work$

If no such i exists, go to step 4

3. $Work = Work + Allocation_i$
Finish[i] = *true*
go to step 2

4. If *Finish* [i] == true for all i, then the system is in a safe state

Resource Request Algorithm

Request = request vector for process P_i. If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available

3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$Available = Available - Request;$

$Allocation_i = Allocation_i + Request_i;$

$Need_i = Need_i - Request_i;$

- *If safe ⇒ the resources are allocated to Pi*
- *If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored*

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Recovery from Deadlock

A. Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
    - Priority of the process
    - How long process has computed, and how much longer to completion
    - Resources the process has used
    - Resources process needs to complete
    - How many processes will need to be terminated

B. Resource Preemption

- Selecting a victim – minimize cost
- Rollback – return to some safe state, restart process for that state
- Starvation – same process may always be picked as victim, include number of rollback in cost factor

Memory Management

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation
- A pair of **base** and **limit** registers define the logical address space

Logical vs Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
    - **Logical address –** generated by the CPU; also referred to as **virtual address**
    - **Physical address –** address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme

Address Binding

- Address binding of instructions and data to memory addresses can happen at three different stages
    - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
    - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
    - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers)

Memory Management Unit

- Hardware device that maps virtual to physical address
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses

Dynamic Loading

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded

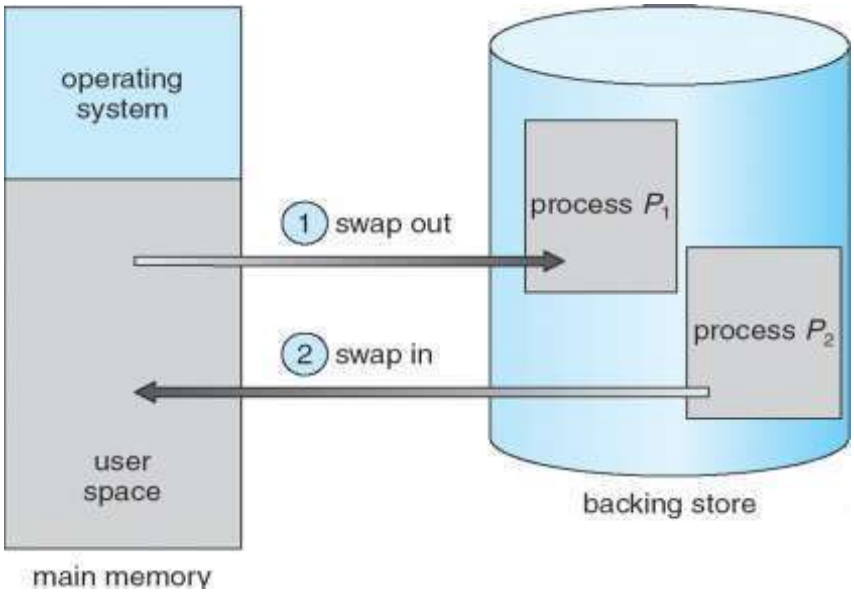
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required implemented through program design

Dynamic Linking

- Linking postponed until execution time
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**

Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk



Contiguous Allocation

- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*
- Multiple-partition allocation
  - Hole – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)

Dynamic Storage Allocation Problem

- **First-fit**: Allocate the *first* hole that is big enough
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
    - Produces the smallest leftover hole
- **Worst-fit**: Allocate the *largest* hole; must also search entire list
    - Produces the largest leftover hole

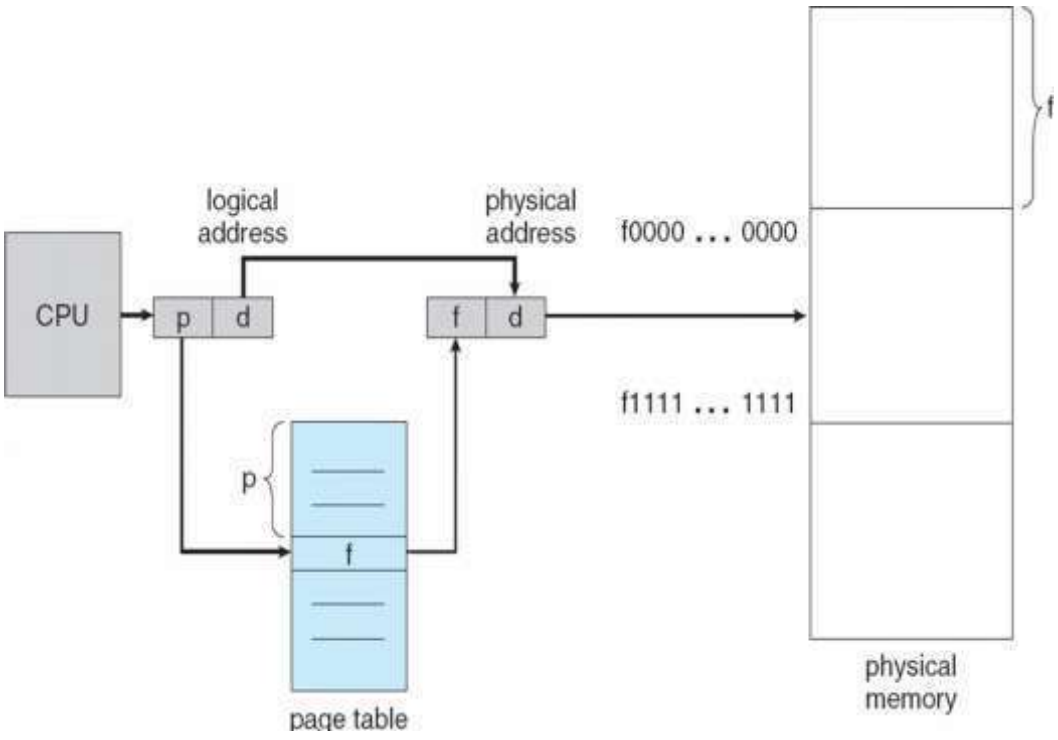
Fragmentation

- **External Fragmentation –** total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation –** allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation by **compaction**
    - Shuffle memory contents to place all free memory together in one large block
    - Compaction is possible *only* if relocation is dynamic, and is done at execution time
    - I/O problem
        - Latch job in memory while it is involved in I/O
        - Do I/O only into OS buffers

Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames

- To run a program of size ***n*** pages, need to find *n* free frames and load program
- Set up a page table to translate logical to physical addresses
- Internal fragmentation
- Address generated by CPU is divided into:
    - **Page number (*p*)** – used as an index into a *page table* which contains base address of each page in physical memory
    - **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit

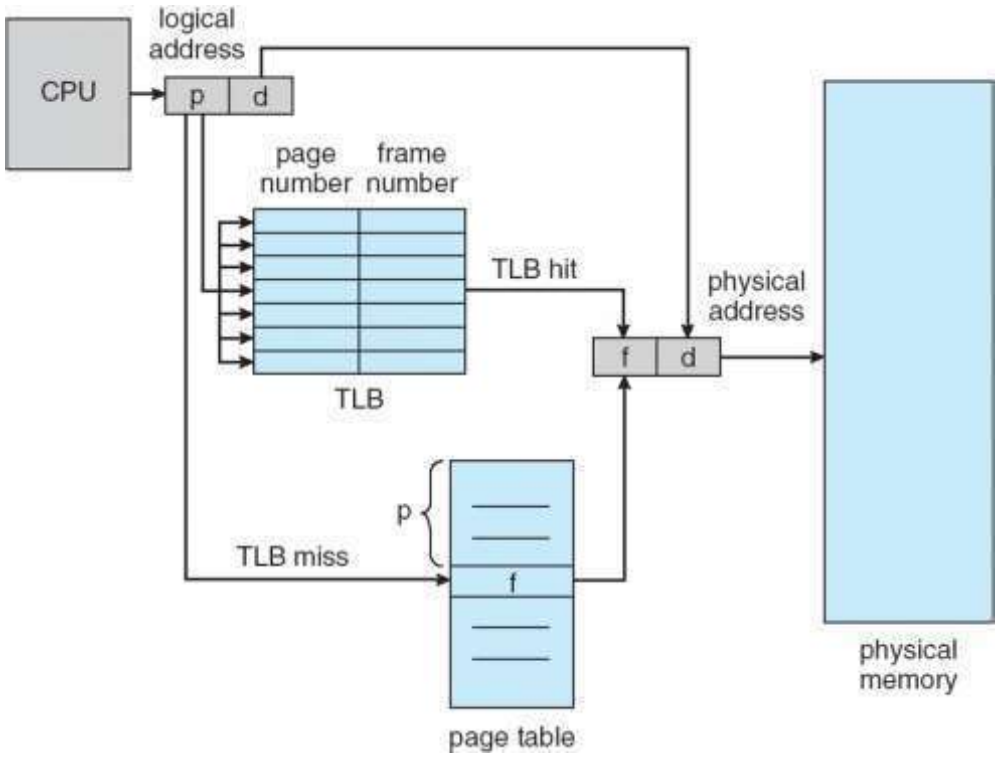


Implementation of Page table

- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PRLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.

- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

Paging with TLB



Memory Protection

- Memory protection implemented by associating protection bit with each frame
- **Valid-invalid** bit attached to each entry in the page table:
    - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
    - "invalid" indicates that the page is not in the process' logical address space

Shared Pages

- **Shared code**

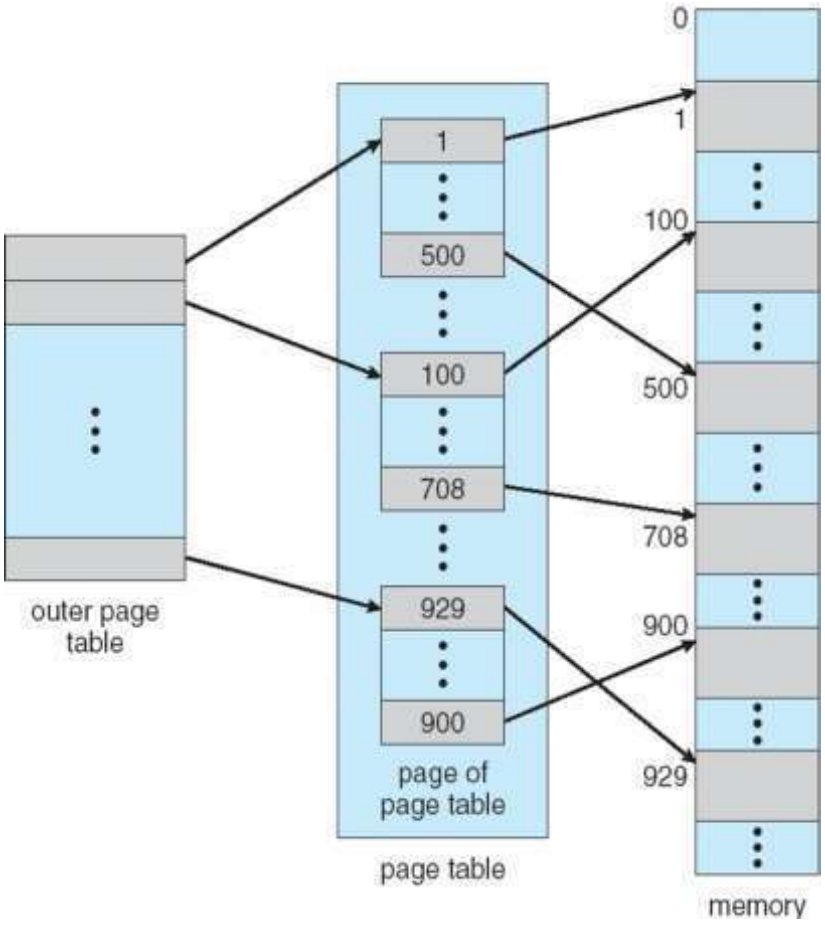
- One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
- Shared code must appear in same location in the logical address space of all processes

- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

Structure of Page table

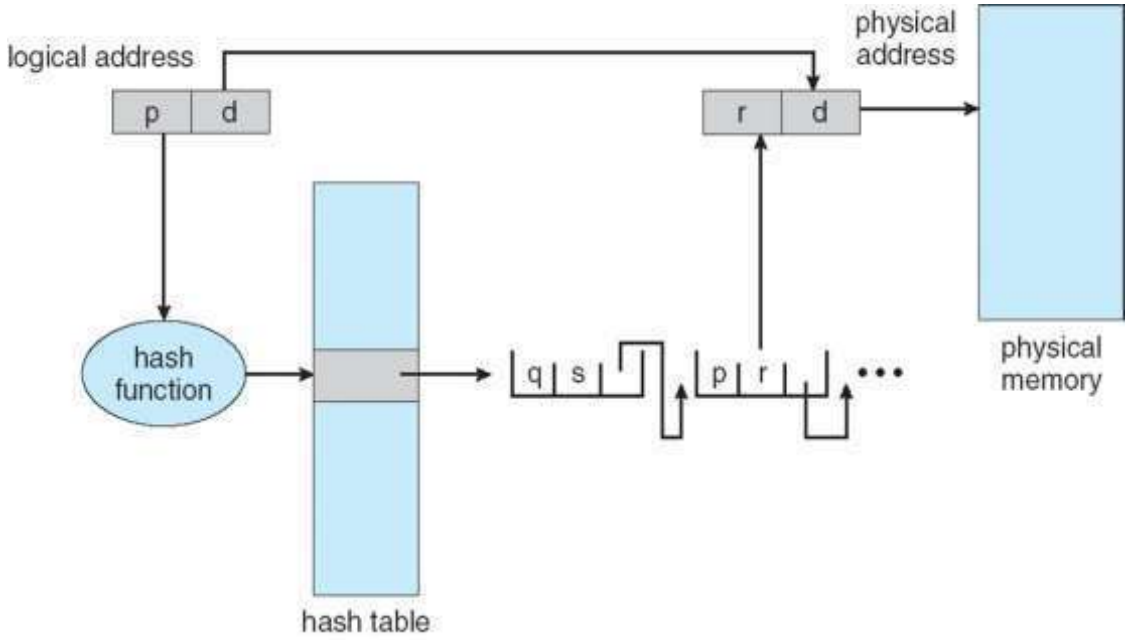
Hierarchical Paging

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table



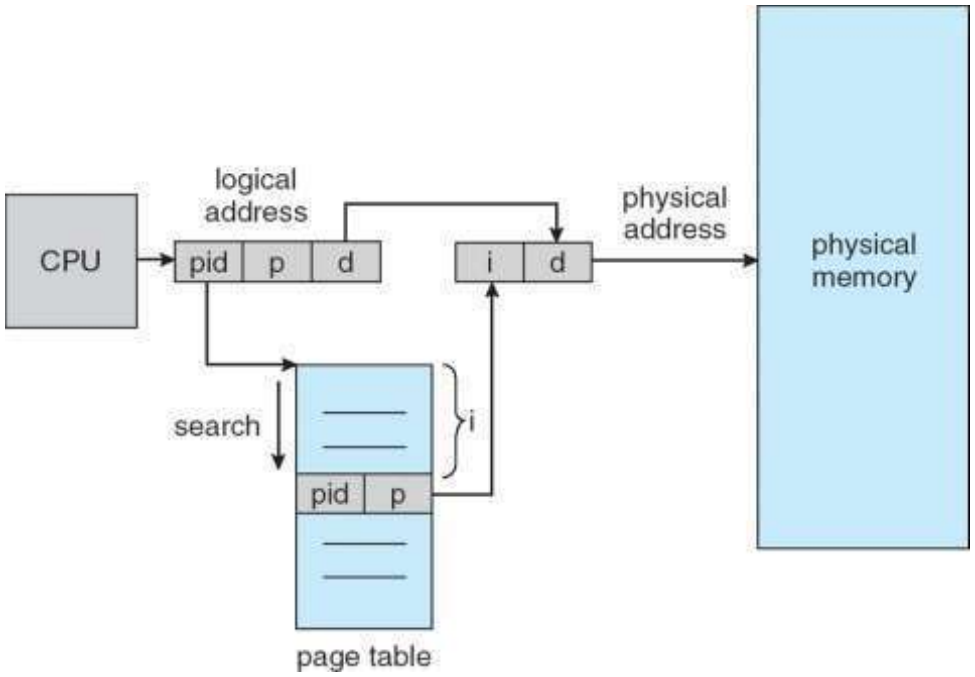
Hashed Page Tables

- The virtual page number is hashed into a page table
    - This page table contains a chain of elements hashing to the same location
- Virtual page numbers are compared in this chain searching for a match
    - If a match is found, the corresponding physical frame is extracted



Inverted Page Tables

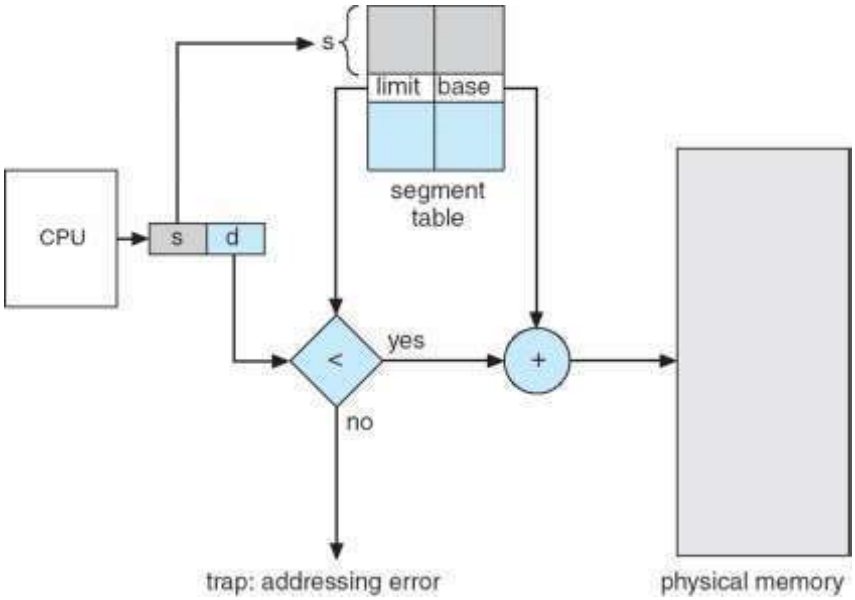
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries



Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments
    - A segment is a logical unit such as: main program, procedure, function, method, object, local variables, global variables, common block, stack, symbol table, arrays
- Logical address consists of a two tuple:
        - <segment-number, offset>,
- Segment table – maps two-dimensional physical addresses; each table entry has:
    - base – contains the starting physical address where the segments reside in memory
    - limit – specifies the length of the segment
- Segment-table base register (STBR) points to the segment table's location in memory
- Segment-table length register (STLR) indicates number of segments used by a program;
        - segment number *s* is legal if *s* < STLR
- Protection

- With each entry in segment table associate:
  - validation bit = 0 ⇒ illegal segment
  - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram



Virtual Memory Management

- **Virtual memory** – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Demand Paging

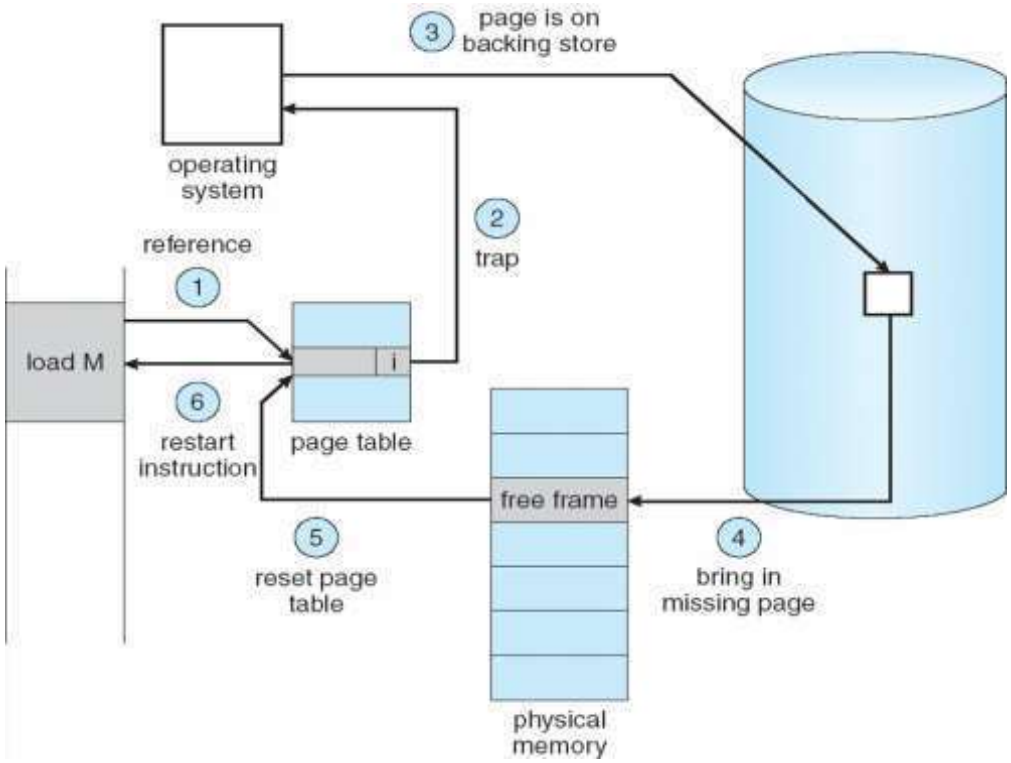
- Bring a page into memory only when it is needed
    - Less I/O needed
    - Less memory needed
    - Faster response
    - More users
- Page is needed ⇒ reference to it
    - invalid reference ⇒ abort
    - not-in-memory ⇒ bring to memory
- **Lazy swapper –** never swaps a page into memory unless page will be needed
    - Swapper that deals with pages is a **pager**
- With each page table entry a valid–invalid bit is associated
  (**v** ⇒ in-memory, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- During address translation, if valid–invalid bit in page table entry is **I** ⇒ page fault

Page Fault

If there is a reference to a page, first reference to that page will trap to operating system: **page fault**

1. Operating system looks at another table to decide:
    - Invalid reference ⇒ abort
    - Just not in memory
2. Get empty frame
3. Swap page into frame
4. Reset tables
5. Set validation bit = **v**

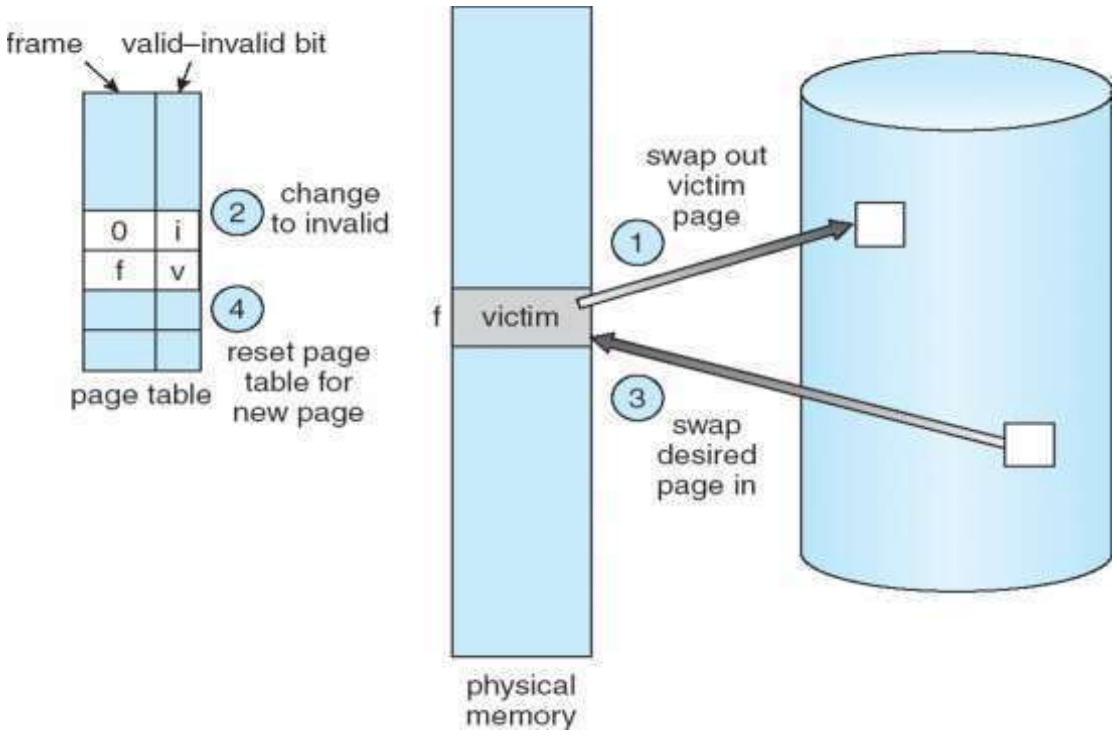
6. Restart the instruction that caused the page fault



Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Use modify (dirty) bit to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory
- Find the location of the desired page on disk
- 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 frame
- Bring the desired page into the (newly) free frame; update the page and frame tables

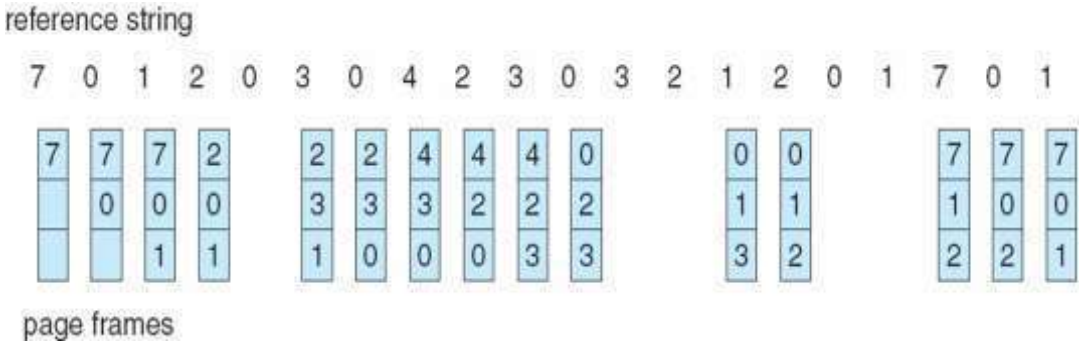
- Restart the process



Page Replacement algorithm

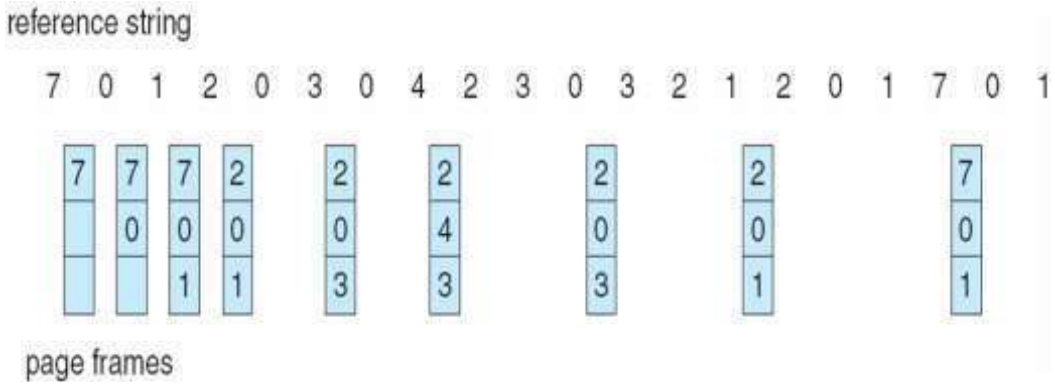
FIFO (First-in-First-Out)

- A FIFO 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.
- **Belady's Anomaly: more frames ⇒ more page faults (** for some page-replacement algorithms, the page-fault rate may ***increase*** as the number of allocated frames increases.)
- **Ex-**



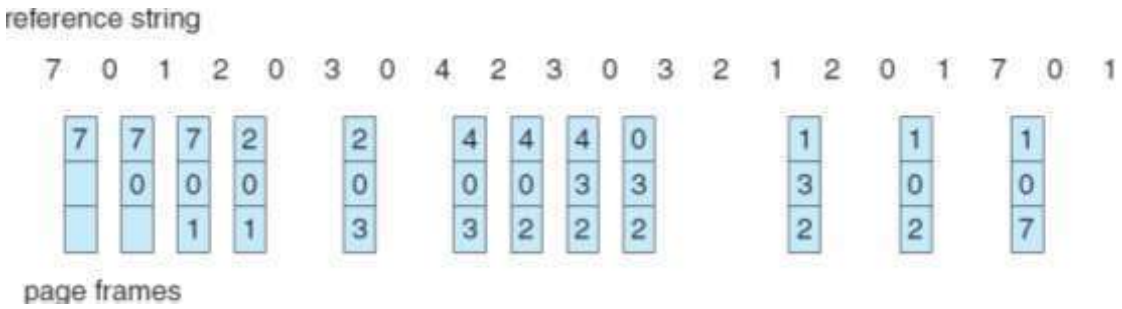
OPTIMAL PAGE REPLACEMENT

- Replace page that will not be used for longest period of time
- Ex-



LRU (LEAST RECENTLY USED)

- 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.
- Ex-



Allocation of Frames

- Each process needs *minimum* number of pages
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Equal allocation – For example, if there are 100 frames and 5 processes, give each process 20 frames.
- Proportional allocation – Allocate according to the size of process

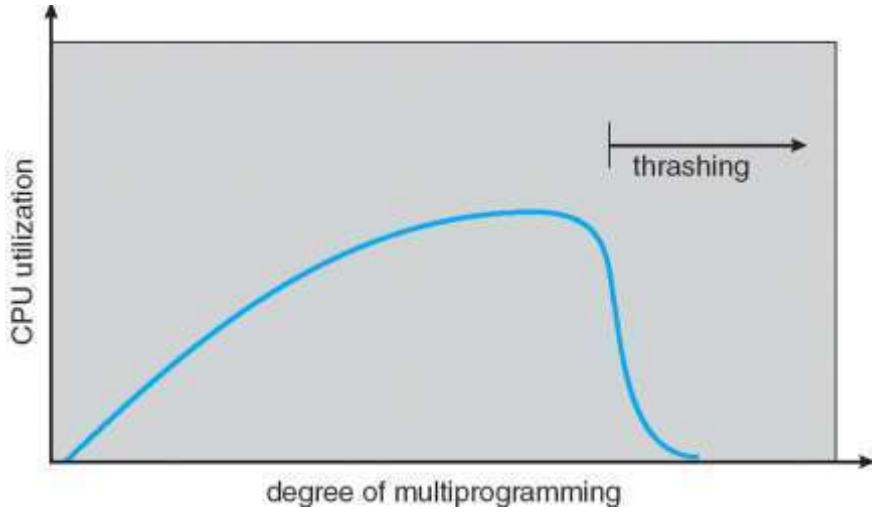
$$s_i = \text{size of process } p_i$$
$$S = \sum s_i$$
$$m = \text{total number of frames}$$
$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

Global vs Local Allocation

- Global replacement – process selects a replacement frame from the set of all frames; one process can take a frame from another
- Local replacement – each process selects from only its own set of allocated frames

Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system
- **Thrashing** ≡ a process is busy swapping pages in and out



MODULE-III

File System

File

- Contiguous logical address space
- Types:
    - Data
        - numeric
        - character
        - binary
    - Program

File Structure

- None - sequence of words, bytes
- Simple record structure
    - Lines
    - Fixed length
    - Variable length
- Complex Structures
    - Formatted document
    - Relocatable load file
- Can simulate last two with first method by inserting appropriate control characters
- Who decides:
    - Operating system
    - Program

File Attribute

- **Name –** only information kept in human-readable form
- **Identifier –** unique tag (number) identifies file within file system
- **Type –** needed for systems that support different types
- **Location –** pointer to file location on device
- **Size –** current file size
- **Protection –** controls who can do reading, writing, executing
- **Time, date, and user identification –** data for protection, security, and usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk

File Types

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

File Operations

- **Create, Write, Read, Reposition within file, Delete, Truncate**
- *Open(F_i)* – search the directory structure on disk for entry F_i, and move the content of entry to memory
- *Close (F_i)* – move the content of entry F_i in memory to directory structure on disk

File Access Methods

n **Sequential Access**

read next

write next

reset

no read after last write

(rewrite)

n **Direct Access**

read *n*

write *n*

position to *n*

read next

write next

rewrite *n*

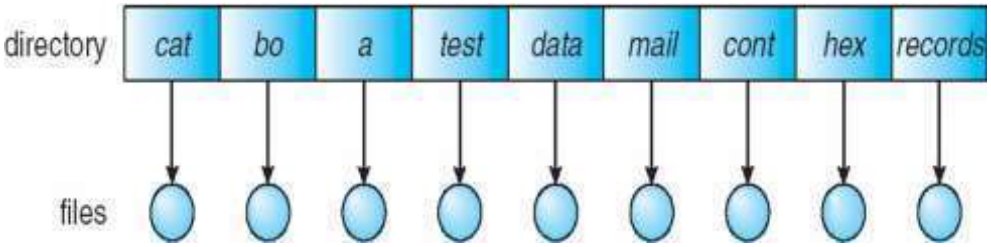
n = relative block number

sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

Directory Structure

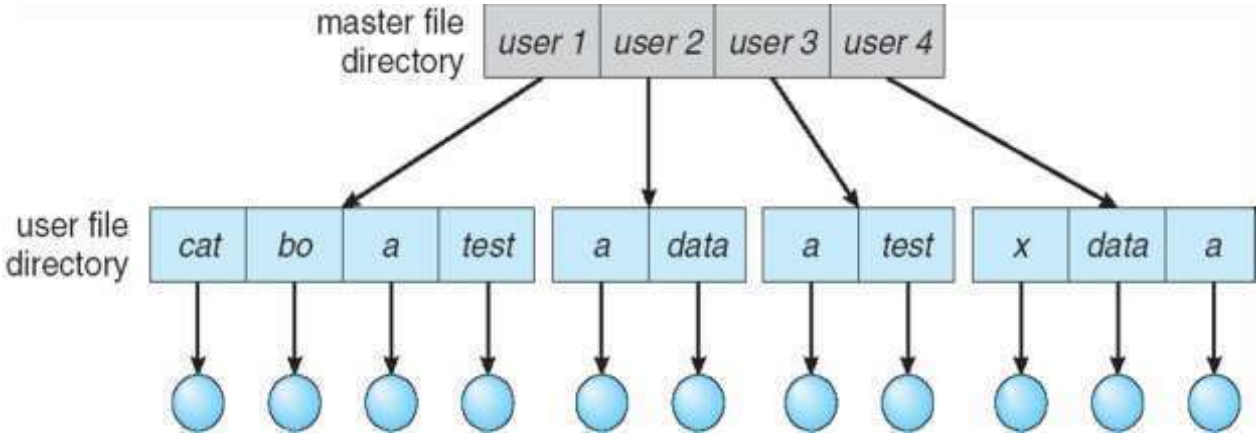
A. Single Level Directory

- A single directory for all users
- Naming problem
- Grouping problem



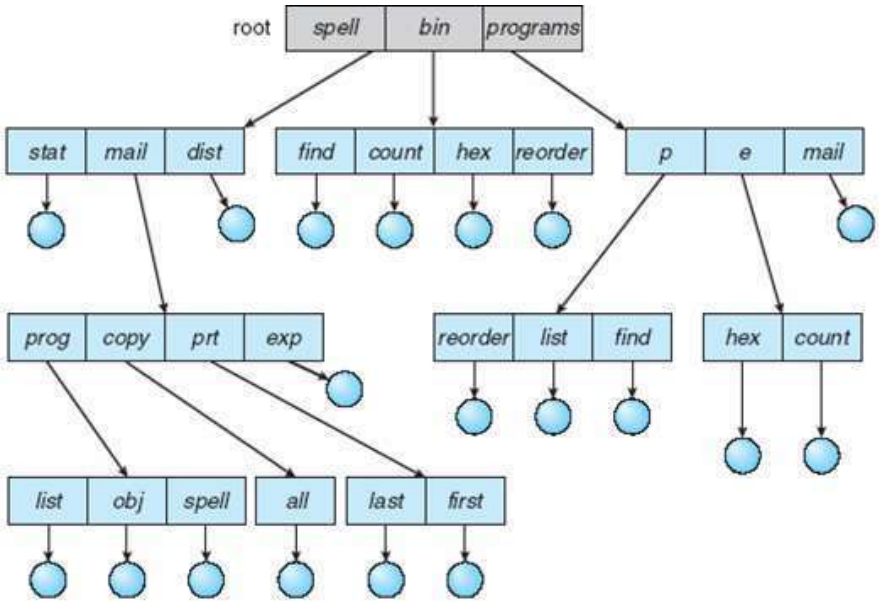
B. Two Level Directory

- Separate directory for each user
- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability

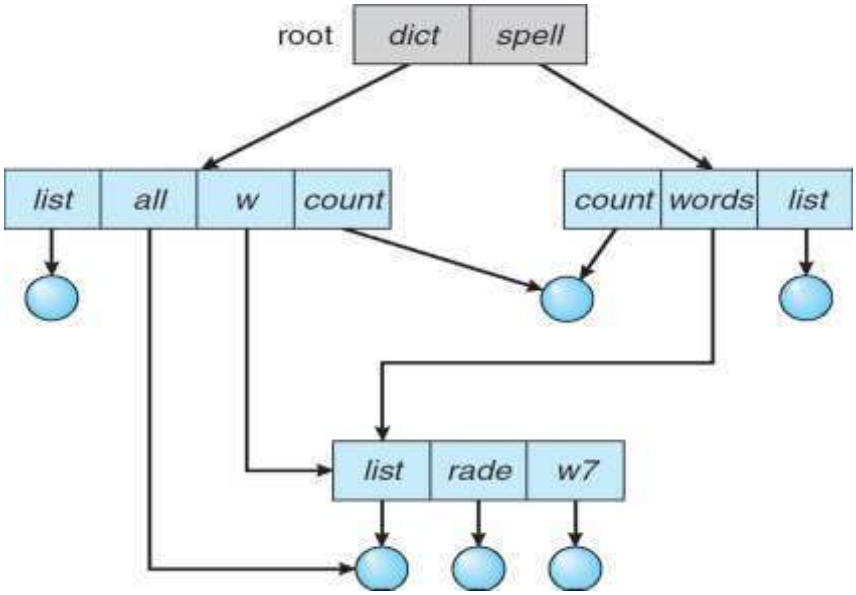


C. Tree Structure Directory

- Efficient searching
- Grouping Capability



D. Acyclic Graph Directories



- Have shared subdirectories and files

File Sharing

- Sharing of files on multi-user systems is desirable
- Sharing may be done through a **protection** scheme
- On distributed systems, files may be shared across a network
- Network File System (NFS) is a common distributed file-sharing method
- **User IDs** identify users, allowing permissions and protections to be per-user
- **Group IDs** allow users to be in groups, permitting group access rights
- Uses networking to allow file system access between systems
    - Manually via programs like FTP
    - Automatically, seamlessly using **distributed file systems**
    - Semi automatically via the **world wide web**
- **Client-server** model allows clients to mount remote file systems from servers
    - Server can serve multiple clients
    - Client and user-on-client identification is insecure or complicated
    - **NFS** is standard UNIX client-server file sharing protocol
    - **CIFS** is standard Windows protocol
    - Standard operating system file calls are translated into remote calls
- Distributed Information Systems **(distributed naming services)** such as LDAP, DNS, NIS, Active Directory implement unified access to information needed for remote computing
- Remote file systems add new failure modes, due to network failure, server failure
- Recovery from failure can involve state information about status of each remote request
- Stateless protocols such as NFS include all information in each request, allowing easy recovery but less security

- **Consistency semantics** specify how multiple users are to access a shared file simultaneously
    - Similar to Ch 7 process synchronization algorithms
        - Tend to be less complex due to disk I/O and network latency (for remote file systems
    - Andrew File System (AFS) implemented complex remote file sharing semantics
    - Unix file system (UFS) implements:
        - Writes to an open file visible immediately to other users of the same open file
        - Sharing file pointer to allow multiple users to read and write concurrently
    - AFS has session semantics
        - Writes only visible to sessions starting after the file is closed

File System Structure

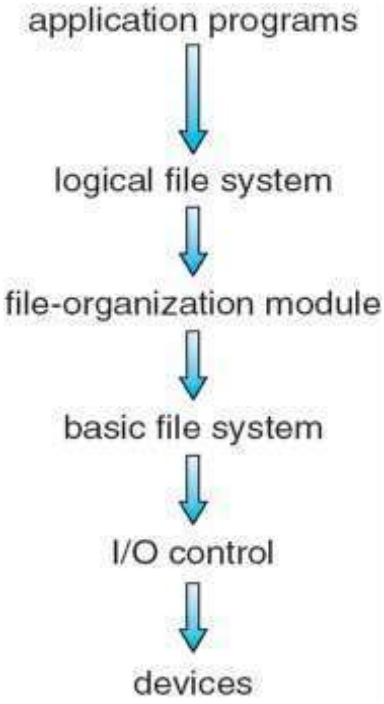
- File structure
    - Logical storage unit
    - Collection of related information

n File system resides on secondary storage (disks)

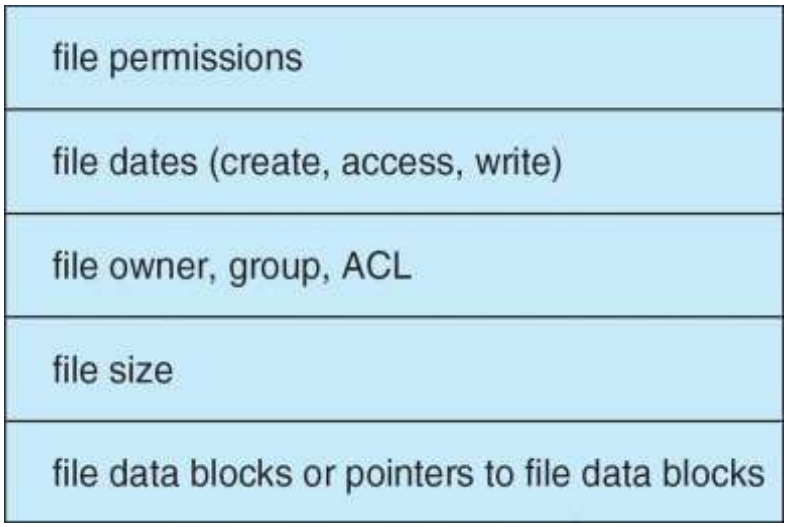
n File system organized into layers

n **File control block –** storage structure consisting of information about a file

Layered File System



File Control Block



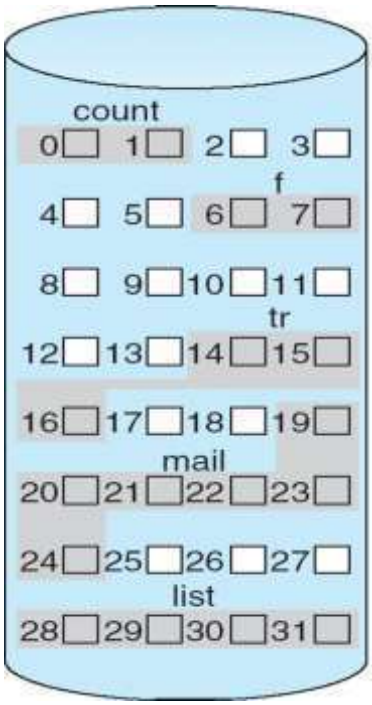
File Allocation Methods

An allocation method refers to how disk blocks are allocated for files:

A. Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk

- Simple – only starting location (block #) and length (number of blocks) are required
- Random access
- Wasteful of space (dynamic storage-allocation problem)
- Files cannot grow

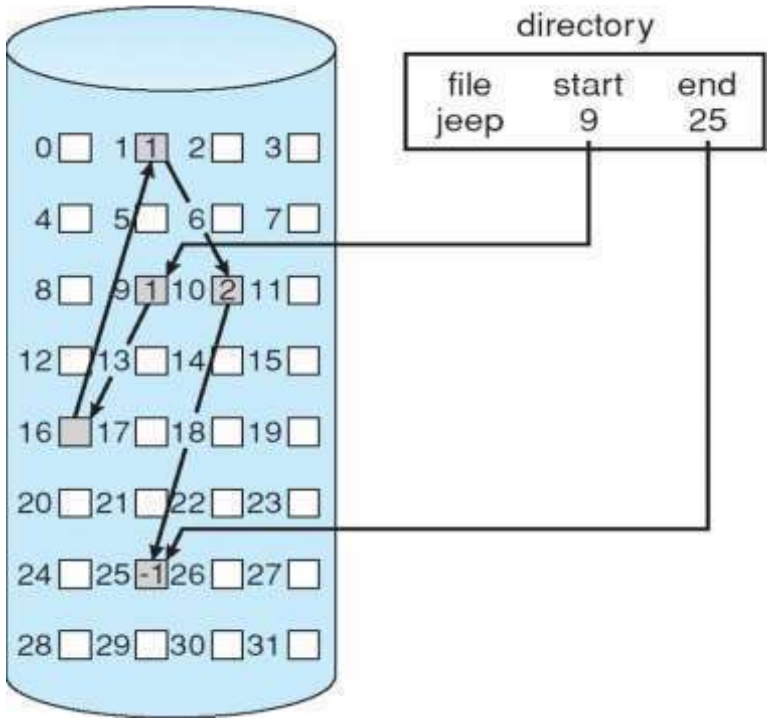


directory

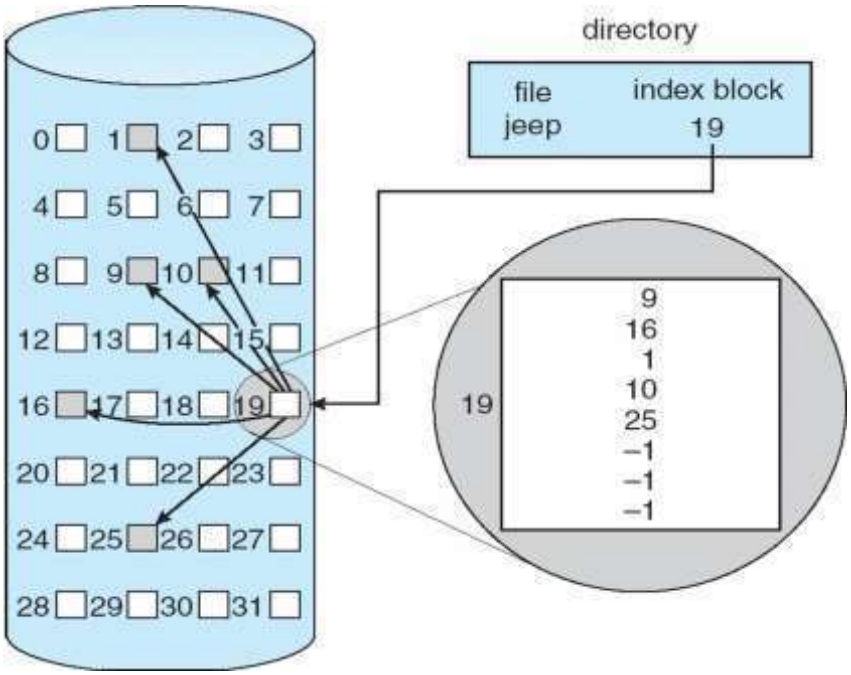
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

B. Linked Allocation

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.
- Simple – need only starting address
- Free-space management system – no waste of space
- No random access
- Mapping



C. Indexed Allocation



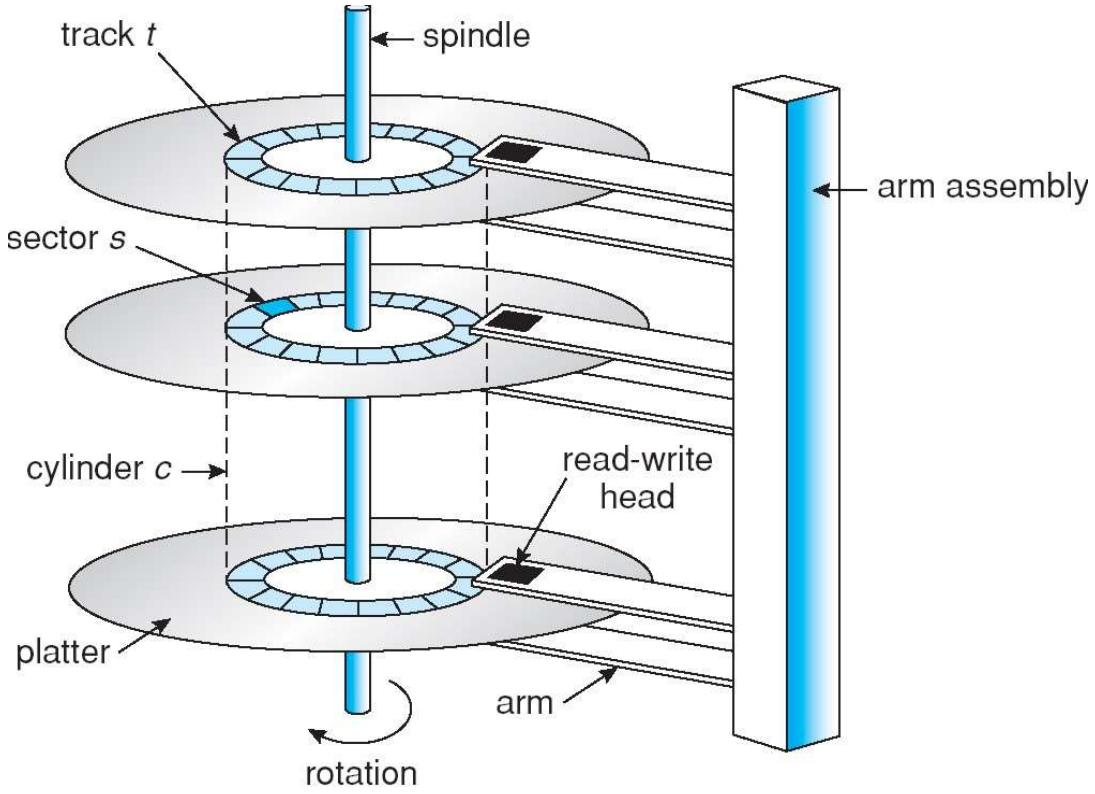
- Brings all pointers together into the *index block.*
- Need index table
- Random access

- Dynamic access without external fragmentation, but have overhead of index block.

Secondary Storage Structure

Magnetic Disk

- Magnetic disks provide bulk of secondary storage of modern computers
  - Drives rotate at 60 to 200 times per second
  - **Transfer rate** is rate at which data flow between drive and computer
  - **Positioning time** (**random-access time**) is time to move disk arm to desired cylinder (**seek time**) and time for desired sector to rotate under the disk head (**rotational latency**)
  - **Head crash** results from disk head making contact with the disk surface
    - That's bad
- Disks can be removable
- Drive attached to computer via **I/O bus**
  - Busses vary, including **EIDE, ATA, SATA, USB, Fibre Channel, SCSI**
  - **Host controller** in computer uses bus to talk to **disk controller** built into drive or storage array



Magnetic Tap

- Was early secondary-storage medium
- Relatively permanent and holds large quantities of data
- Access time slow
- Random access ~1000 times slower than disk
- Mainly used for backup, storage of infrequently-used data, transfer medium between systems
- Kept in spool and wound or rewound past read-write head
- Once data under head, transfer rates comparable to disk
- 20-200GB typical storage

Disk Structure

- Disk drives are addressed as large 1-dimensional arrays of *logical blocks*, where the logical block is the smallest unit of transfer.
- The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially.
    - Sector 0 is the first sector of the first track on the outermost cylinder.
    - Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

Disk Scheduling

- The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth.
- Access time has two major components
    - *Seek time* is the time for the disk are to move the heads to the cylinder containing the desired sector.

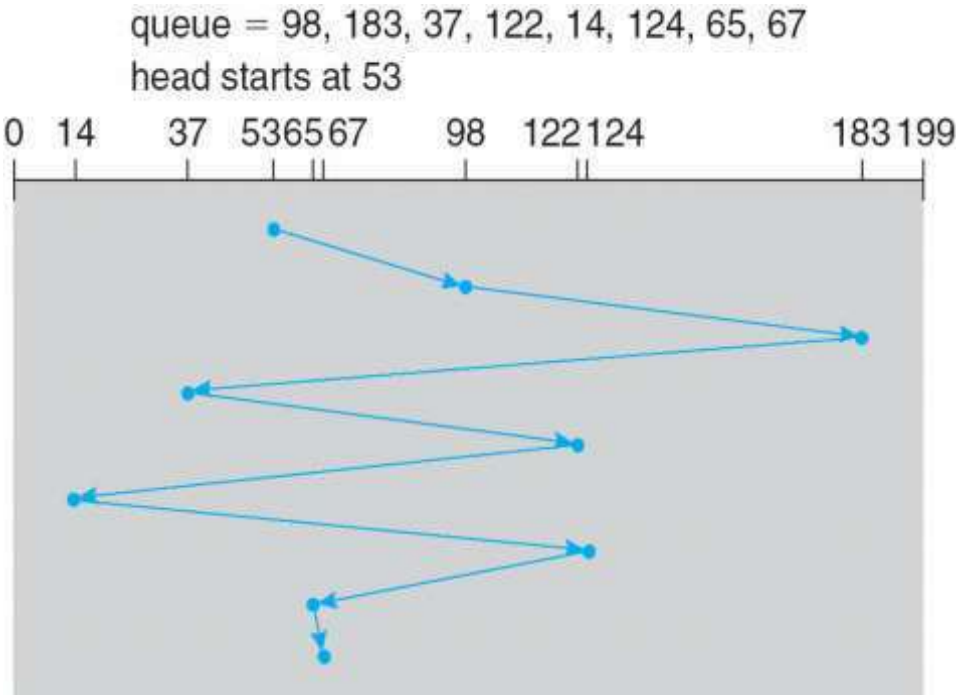
- *Rotational latency* is the additional time waiting for the disk to rotate the desired sector to the disk head.

- Minimize seek time
- Seek time ≈ seek distance
- Disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.

Disk Scheduling Algorithms

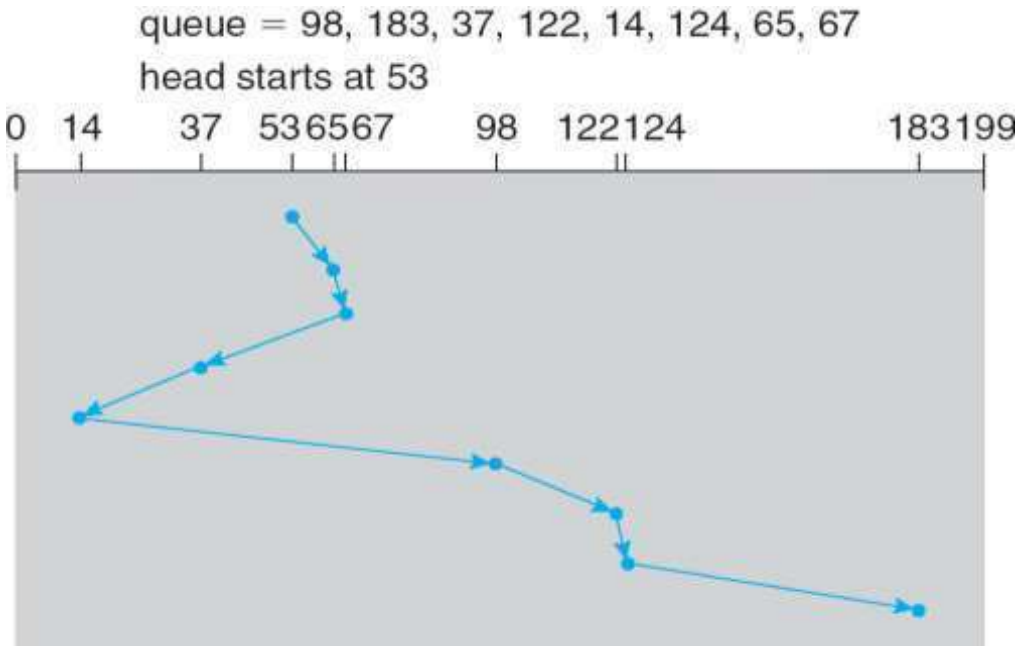
FCFS

- This algorithm is intrinsically fair, but it generally does not provide the fastest service.



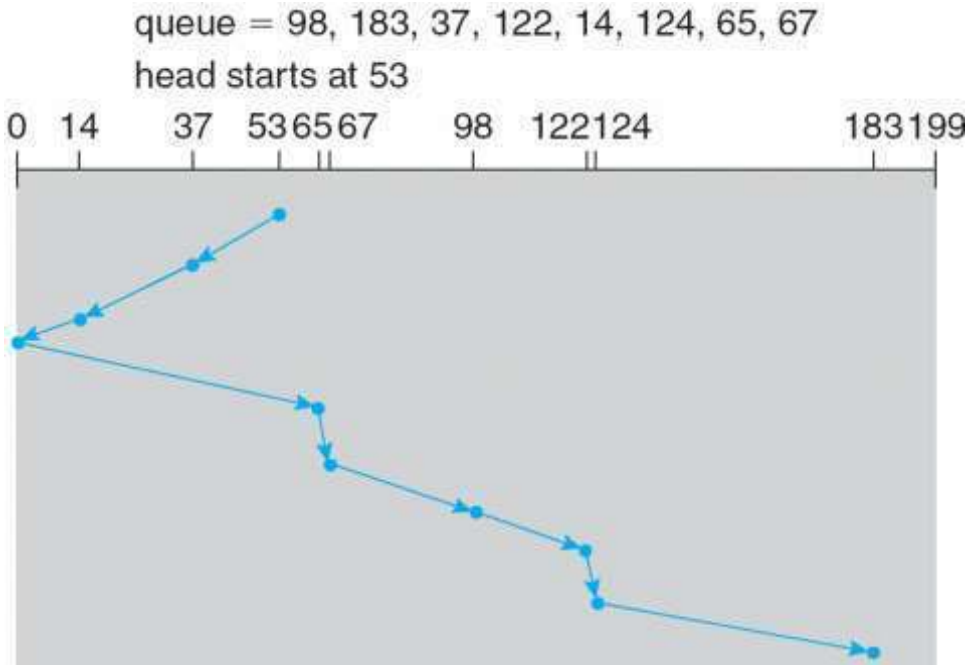
SSTF (Shortest Seek Time First)

- Selects the request with the minimum seek time from the current head position.
- SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests.



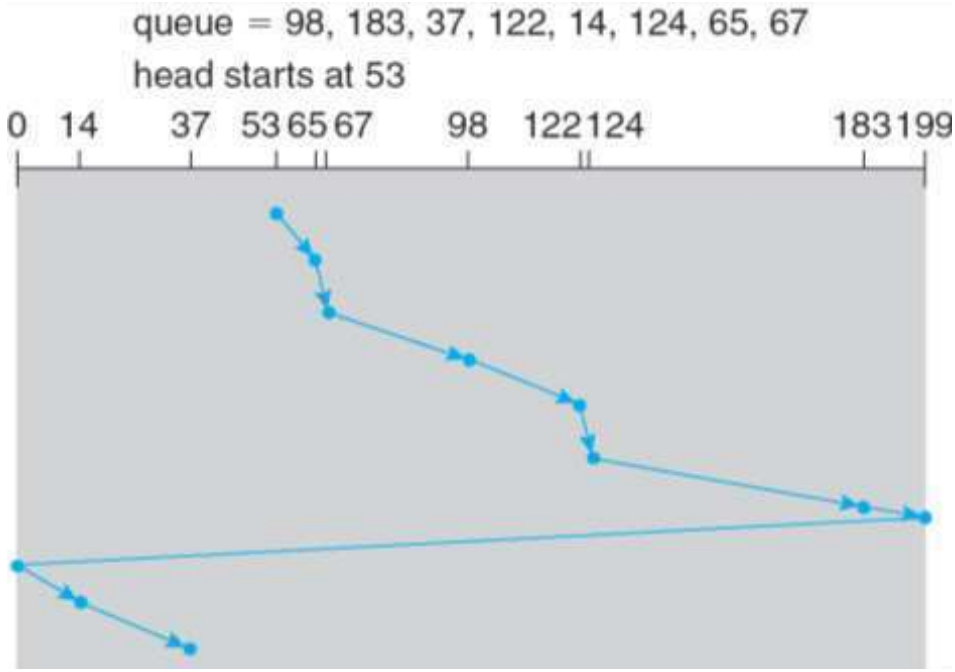
SCAN

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- Sometimes called the *elevator algorithm*.



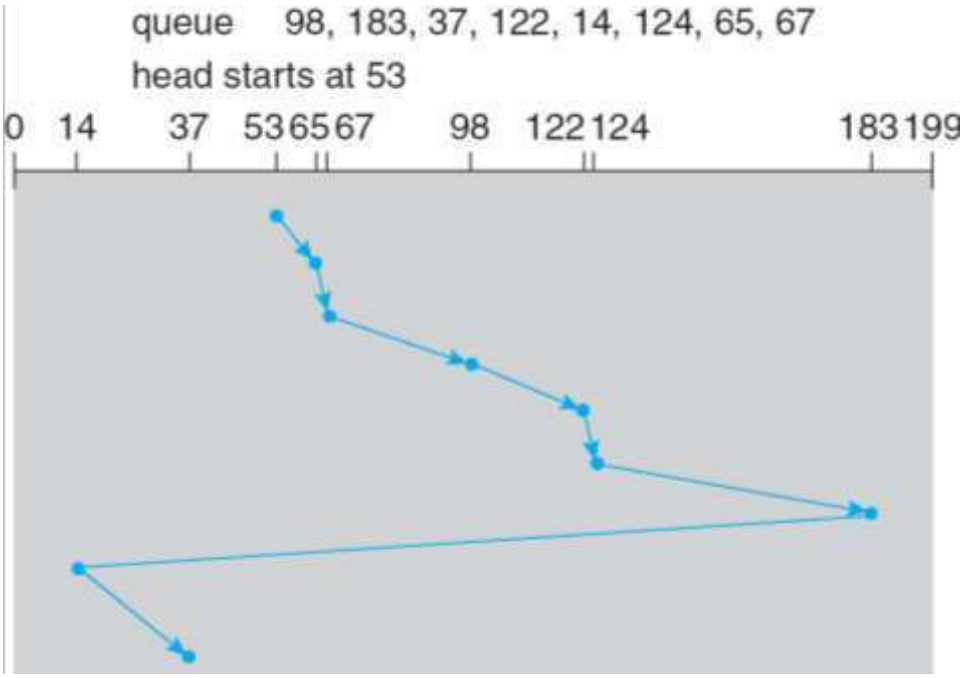
C-SCAN

- Provides a more uniform wait time than SCAN.
- The head moves from one end of the disk to the other. servicing requests as it goes. When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.
- Treats the cylinders as a circular list that wraps around from the last cylinder to the first one



C-LOOK

- Version of C-SCAN
- Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk.



Disk Management

- *Low-level formatting*, or *physical formatting* — Dividing a disk into sectors that the disk controller can read and write.
- To use a disk to hold files, the operating system still needs to record its own data structures on the disk.
    - *Partition* the disk into one or more groups of cylinders.
    - *Logical formatting* or “making a file system”.
- Boot block initializes system.
    - The bootstrap is stored in ROM.
    - *Bootstrap loader* program.
- Methods such as *sector sparing* used to handle bad blocks.
- The controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as **sector sparing** or **forwarding**.

Swap Space Management

- Swap-space — Virtual memory uses disk space as an extension of main memory.

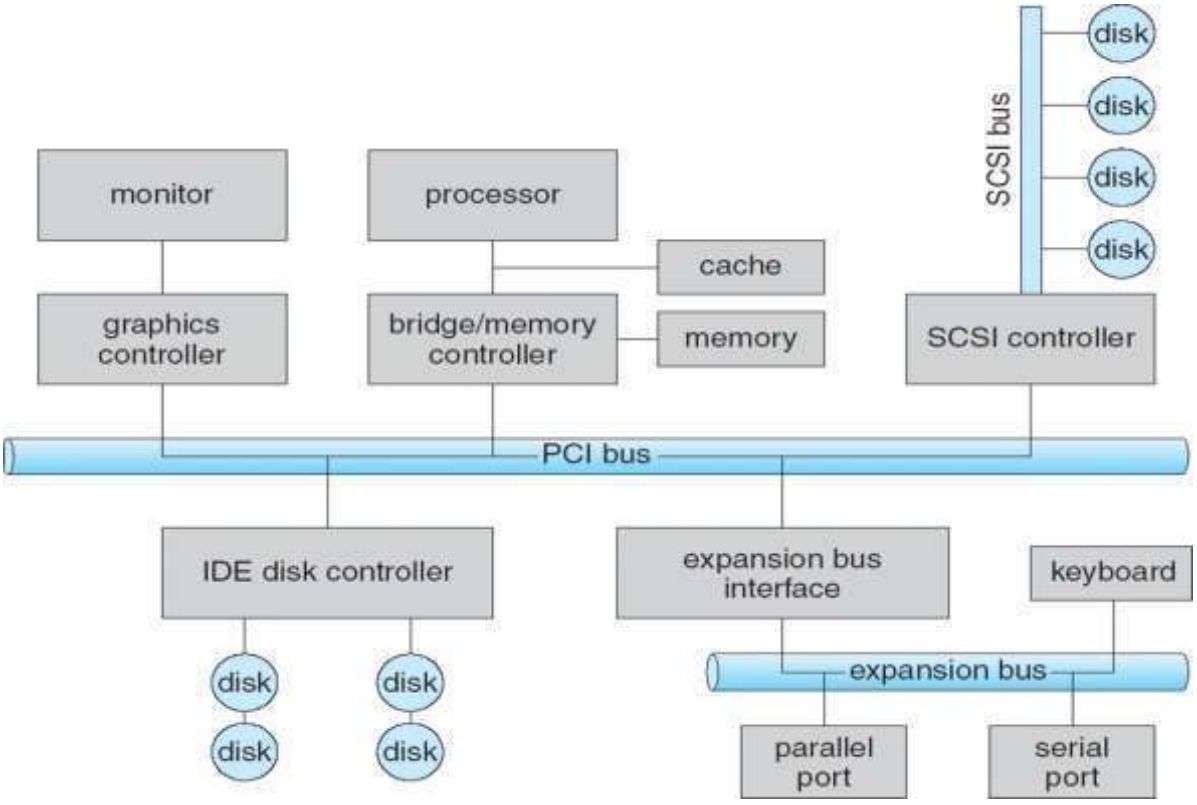
- Swap-space can be carved out of the normal file system or, more commonly, it can be in a separate disk partition.
- A swap space can reside in one of two places: it can be carved out of the normal file system, or it can be in a separate disk partition.
- If the swap space is simply a large file within the file system, normal file-system routines can be used to create it, name it, and allocate its space.
- Alternatively, swap space can be created in a separate **raw partition**. No file system or directory structure is placed in this space.
- A separate swap-space storage manager is used to allocate and deallocate the blocks from the raw partition.

I/O Systems

I/O Hardware

- A device communicates with a computer system by sending signals over a cable or even through the air. The device communicates with the machine via a connection point, or **port—**for example, a serial port.
- If devices share a common set of wires, the connection is called a bus.
- A **bus** is a set of wires and a rigidly defined protocol that specifies a set of messages that can be sent on the wires.
- When device *A* has a cable that plugs into device *B,* and device *B* has a cable that plugs into device *C,* and device *C* plugs into a port on the computer, this arrangement is called a **daisy chain.** A daisy chain usually operates as a bus.
- A **PCI bus** (the common PC system bus) connects the processor–memory subsystem to fast devices, and an **expansion bus** connects relatively slow devices, such as the keyboard and serial and USB ports.
- Disks are connected together on a **Small Computer System Interface (SCSI)** bus plugged into a SCSI controller.

- A **controller** is a collection of electronics that can operate a port, a bus, or a device.
- A serial-port controller is a simple device controller. It is a single chip (or portion of a chip) in the computer that controls the signals on the wires of a serial port.
- But the SCSI protocol is complex, the SCSI bus controller is often implemented as a separate circuit board (or a **host adapter**) that plugs into the computer. It typically contains a processor, microcode, and some private memory to enable it to process the SCSI protocol messages.

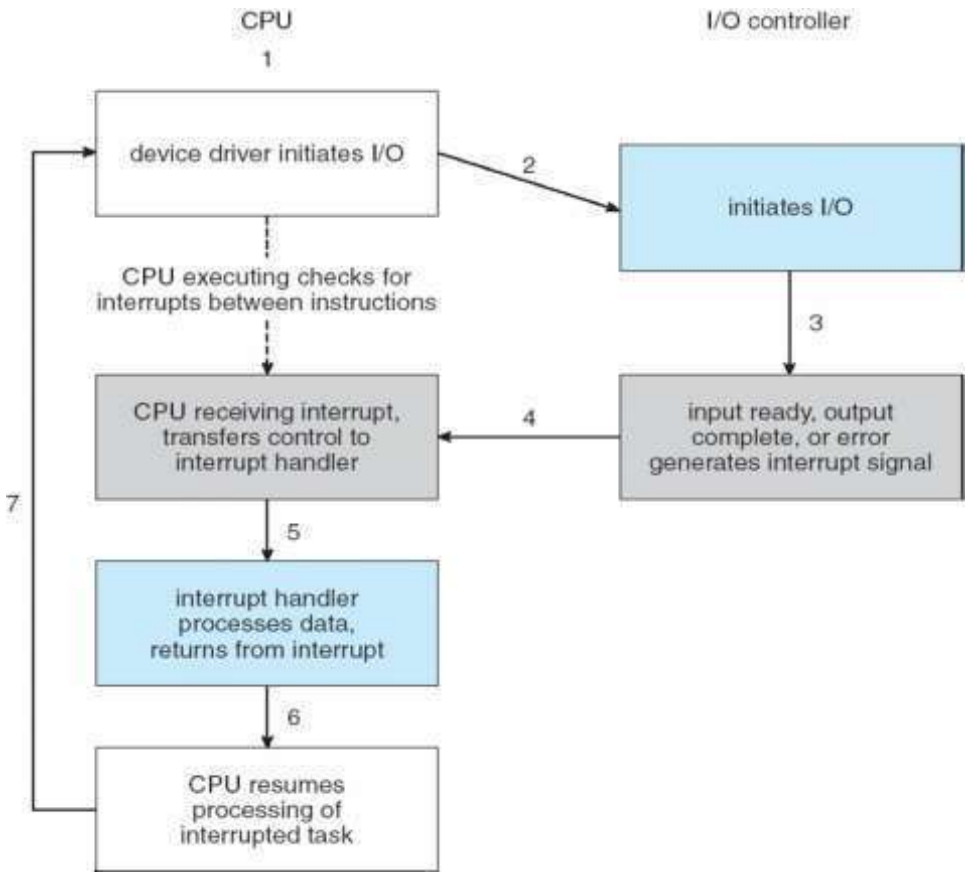


Polling

- Determines state of device
  - command-ready
  - busy
  - Error
- **Busy-wait** cycle to wait for I/O from device

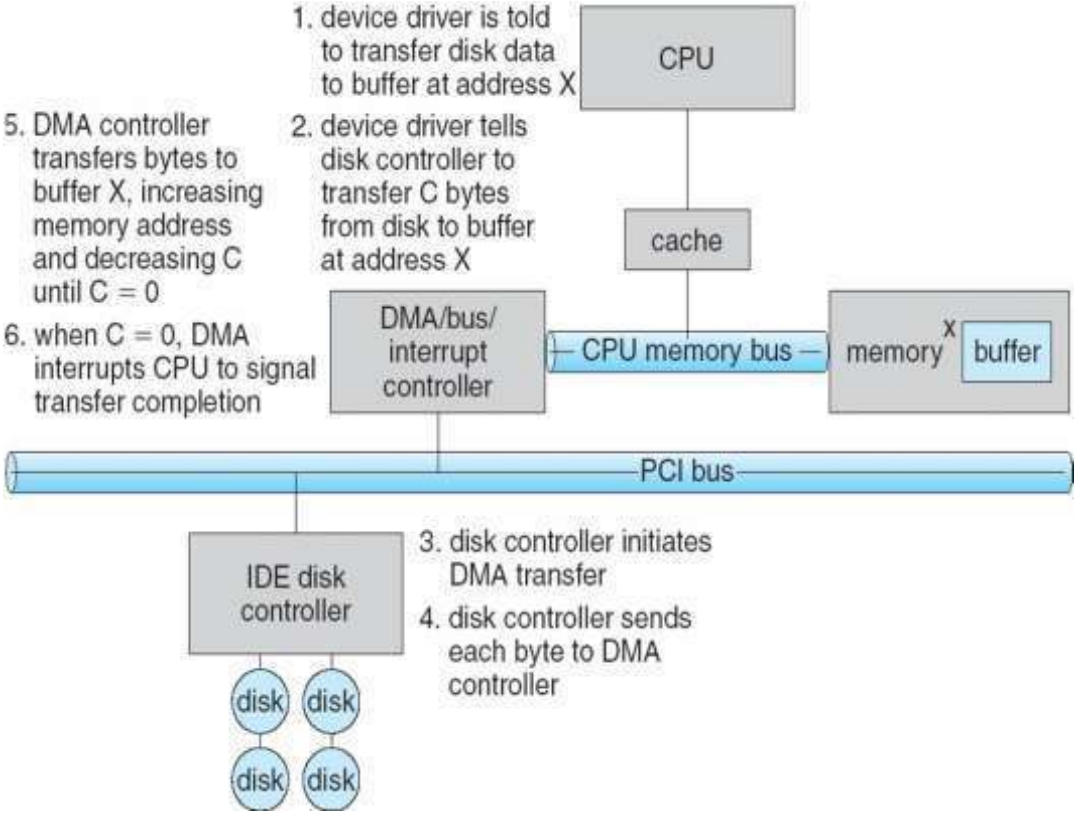
Interrupt

- CPU **Interrupt-request line** triggered by I/O device
- **Interrupt handler** receives interrupts
- **Maskable** to ignore or delay some interrupts
- Interrupt vector to dispatch interrupt to correct handler
- Most CPUs have two interrupt request lines. One is the **nonmaskable interrupt**, which is reserved for events such as unrecoverable memory errors.
- The second interrupt line is **maskable**: it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted.
- The maskable interrupt is used by device controllers to request service.
- Interrupt mechanism also used for exceptions



Direct Memory Access

- Used to avoid **programmed I/O** for large data movement
- Requires **DMA** controller
- Bypasses CPU to transfer data directly between I/O device and memory

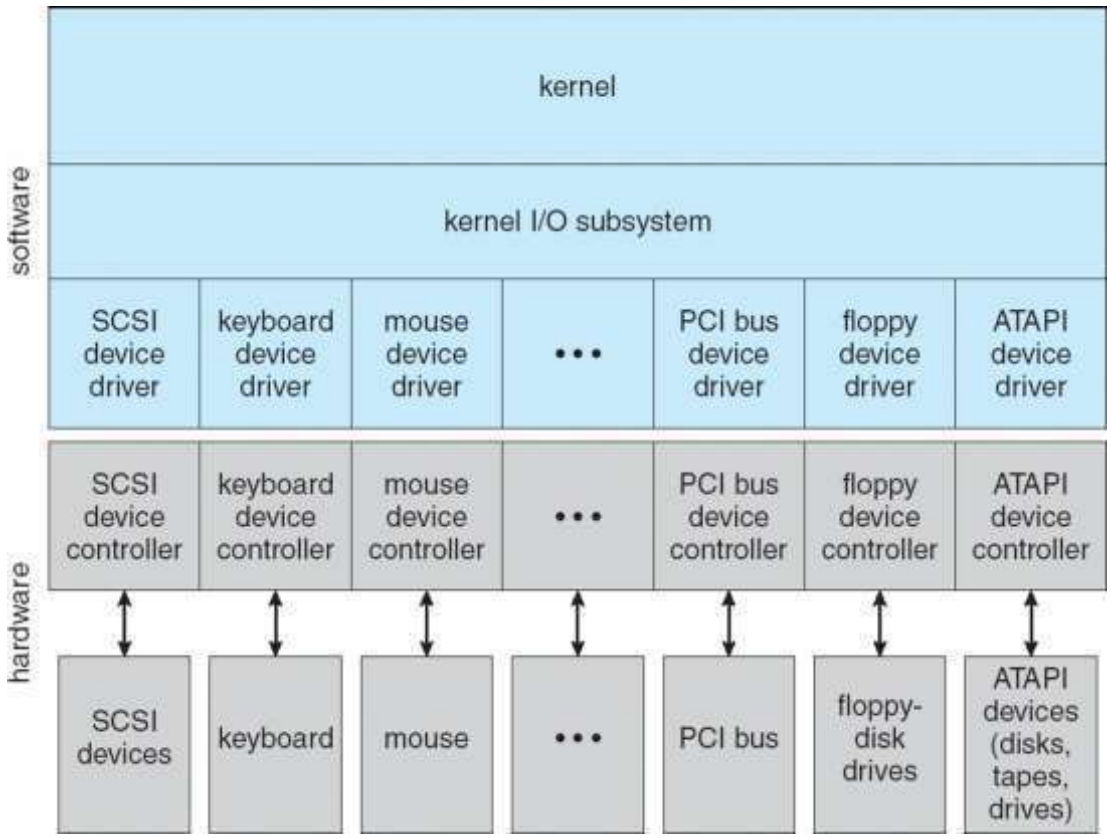


Application I/O Interface

- I/O system calls encapsulate device behaviors in generic classes
- Device-driver layer hides differences among I/O controllers from kernel
- Devices vary in many dimensions
    - **Character-stream** or **block**
    - **Sequential or random-access**
    - **Sharable or dedicated**
    - **Speed of operation**

- **read-write, read only,** or **write only**

Kernel I/O Structure



Characteristics of I/O Devices

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Block and Character Devices

- Block devices include disk drives
    - Commands include read, write, seek
    - Raw I/O or file-system access
    - Memory-mapped file access possible
- Character devices include keyboards, mice, serial ports
    - Commands include get, put
    - Libraries layered on top allow line editing

Network Devices

- Varying enough from block and character to have own interface
- Unix and Windows NT/9*x*/2000 include socket interface
    - Separates network protocol from network operation
    - Includes select functionality
- Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)

Clock and Timers

- Provide current time, elapsed time, timer
- **Programmable interval timer** used for timings, periodic interrupts

Blocking and Non-blocking I/O

- **Blocking** - process suspended until I/O completed
    - Easy to use and understand
    - Insufficient for some needs
- **Nonblocking** - I/O call returns as much as available
    - User interface, data copy (buffered I/O)
    - Implemented via multi-threading

    - Returns quickly with count of bytes read or written
- **Asynchronous** - process runs while I/O executes
    - Difficult to use
    - I/O subsystem signals process when I/O completed

Kernel I/O Subsystem

- **Scheduling**
    - Some I/O request ordering via per-device queue
    - Some OSs try fairness
- **Buffering -** store data in memory while transferring between devices
    - To cope with device speed mismatch
    - To cope with device transfer size mismatch
    - To maintain “copy semantics”
- **Caching** - fast memory holding copy of data
    - Always just a copy
    - Key to performance
- **Spooling** - hold output for a device
    - If device can serve only one request at a time
    - i.e., Printing
- **Device reservation** - provides exclusive access to a device
    - System calls for allocation and deallocation
    - Watch out for deadlock

Reference

Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition ",