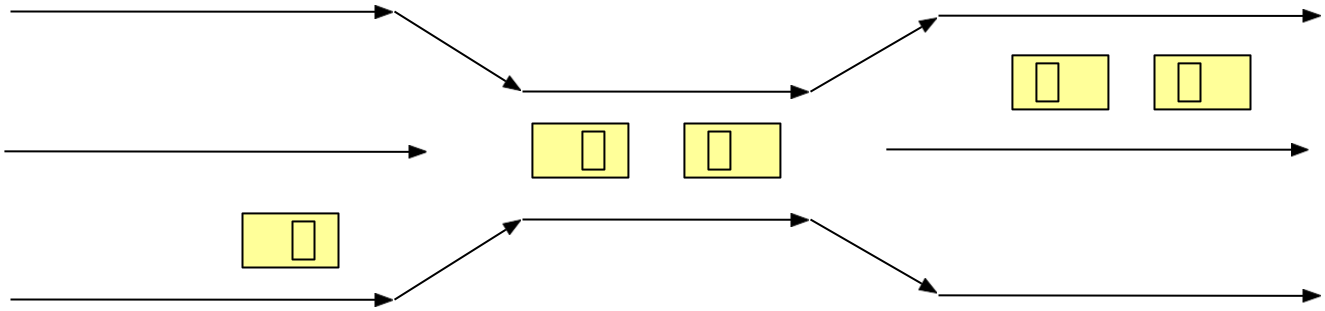
Deadlocks

The Deadlock Problem

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

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

Bridge Crossing Example



- Traffic only in one direction
- The resource is a one-lane bridge
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible

System Model

- Resource types R_1, R_2, . . ., R_m

  CPU cycles, memory space, I/O devices
- Each resource type R_i has *1 or more* instances
- Each process utilizes a resource as follows:
  - request
  - use
  - release

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

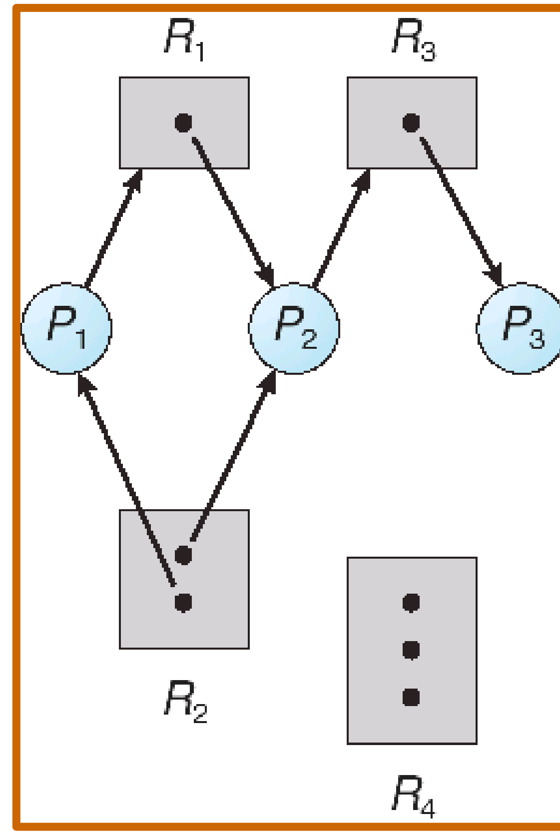
- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it after that process 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

  $P_2, \ldots, P_{n-1}$ is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0

Resource-Allocation Graph

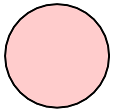
A set of vertices V and a set of edges E.

- 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 $R_j \rightarrow P_i$

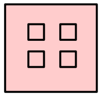


Resource-Allocation Graph (Cont.)

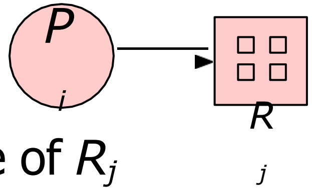
- Process



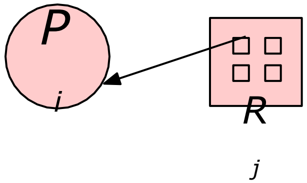
- Resource Type with 4 instances



- P_i requests instance of R_j

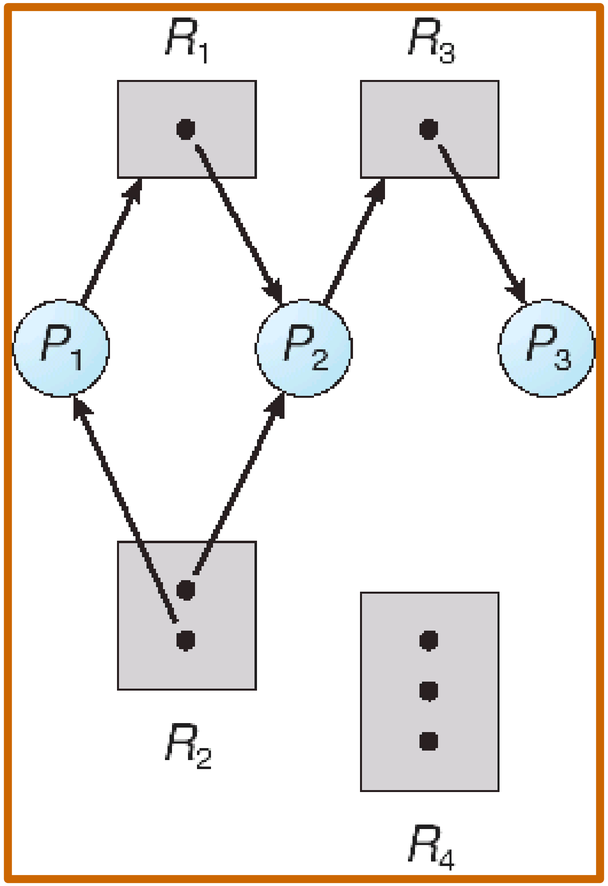


- P_i is holding an instance of R_j

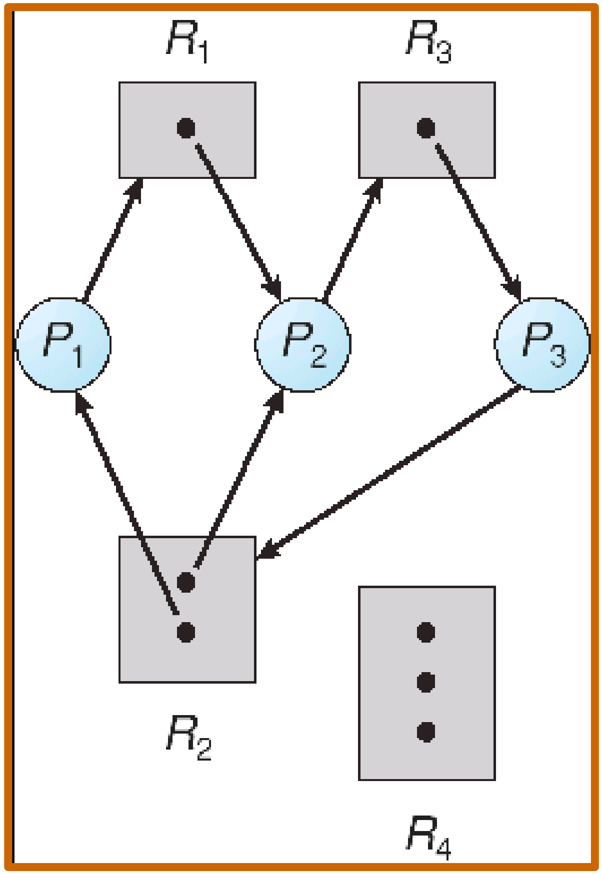


Resource Allocation Graph With A Deadlock

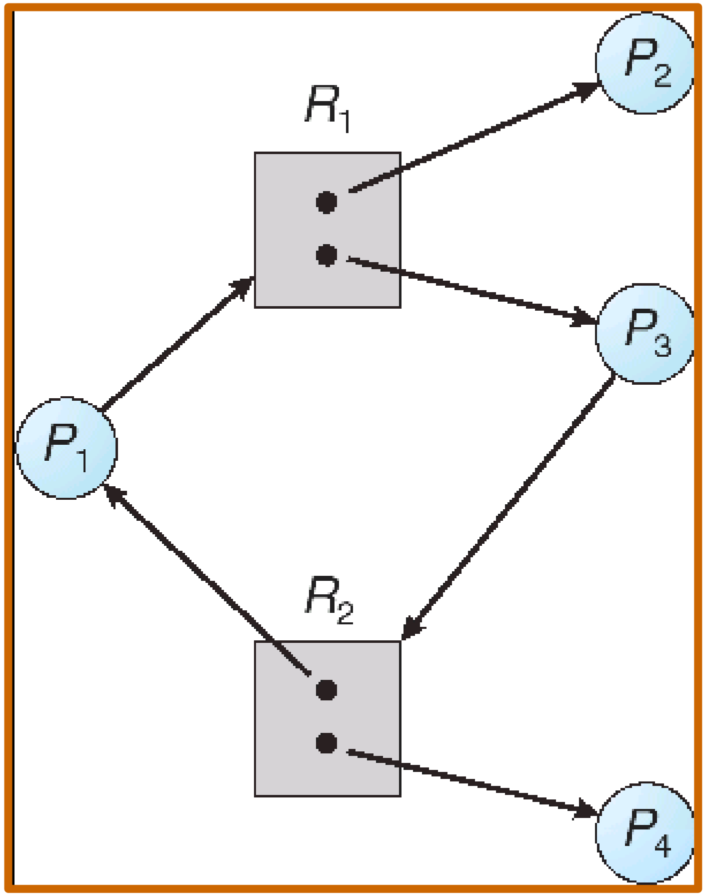
Before P_3 requested an instance of R_2



After P_3 requested an instance of R_2



Graph With A Cycle But No Deadlock



Process P_4 may release its instance of resource type R_2. That resource can then be allocated to P3, thereby breaking the cycle.

Relationship of cycles to deadlocks

- If a resource allocation graph contains <u>no</u> cycles ⇒ no deadlock
- If a resource allocation graph contains a cycle and if <u>only one</u> instance exists per resource type ⇒ deadlock
- If a resource allocation graph contains a cycle and and if <u>several</u> instances exists per resource type ⇒ possibility of deadlock

Methods for Handling Deadlocks

- **Prevention**
  - Ensure that the system will *never* enter a deadlock state
- **Avoidance**
  - Ensure that the system will *never* enter an unsafe state
- **Detection**
  - Allow the system to enter a deadlock state and then recover
- **Do Nothing**
  - Ignore the problem and let the user or system administrator respond to the problem; used by most operating systems, including Windows and UNIX

Deadlock Prevention

To prevent deadlock, we can restrain the ways that a request can be made

- **Mutual Exclusion** – The mutual-exclusion condition must hold for non-sharable resources

- **Hold and Wait** – we must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require a process to request and be allocated all its resources before it begins execution, or allow a process to request resources only when the process has none
  - Result: Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

- **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
  - A 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. For example:

  F(tape drive) = 1
  F(disk drive) = 5
  F(printer) = 12

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

- A resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

a priori: formed or conceived beforehand

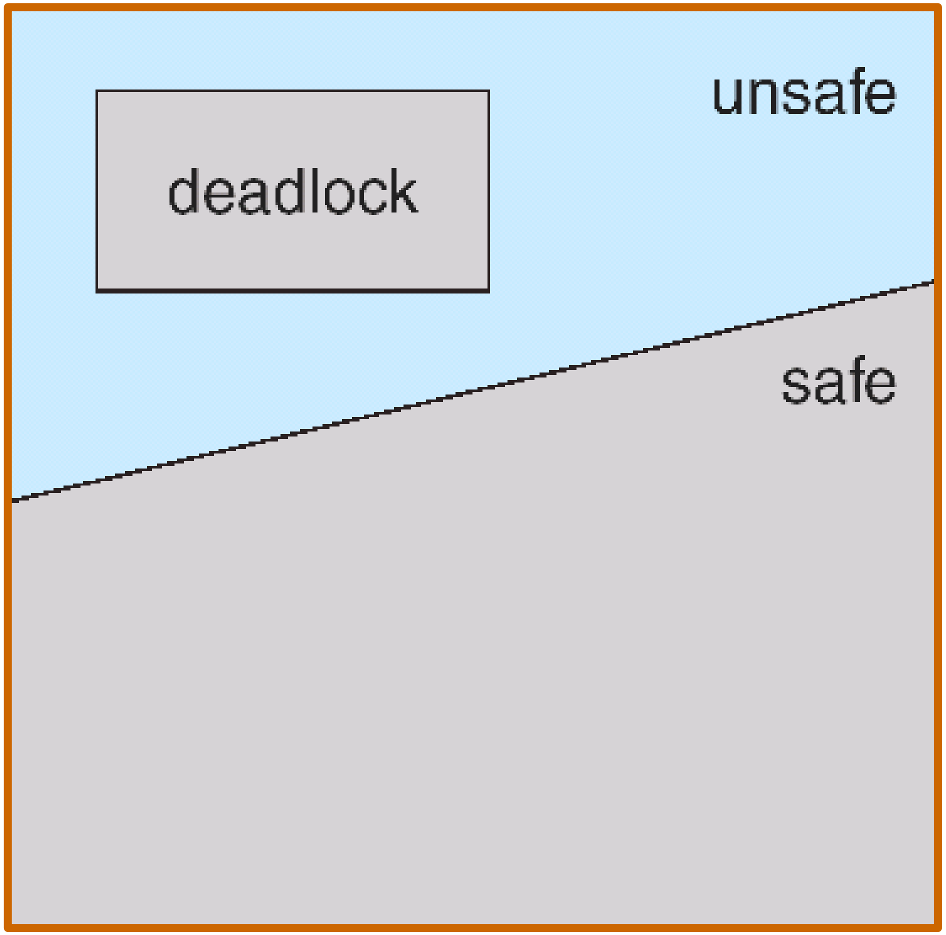
Safe State

- When a process requests an available resource, the system <u>must decide</u> if immediate allocation leaves the system in a <u>safe state</u>
- A system is in a safe state only if there exists a <u>safe sequence</u>
- A sequence of processes $<P_1, P_2, \ldots, P_n>$ is a safe sequence for the current allocation state if, for each P_i, the resource requests that P_i can still make, can be satisfied by currently available resources plus resources held by all P_j, with $j < i$.
- That is:
  - If the 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

Safe State (continued)

- If a system is in <u>safe</u> state ⇒ no deadlocks
- If a system is in <u>unsafe</u> state ⇒ possibility of deadlock
- Avoidance ⇒ ensure that a system will <u>never</u> enter an <u>unsafe</u> state

Safe, Unsafe , Deadlock State



Avoidance algorithms

- For a single instance of a resource type, use a resource-allocation graph

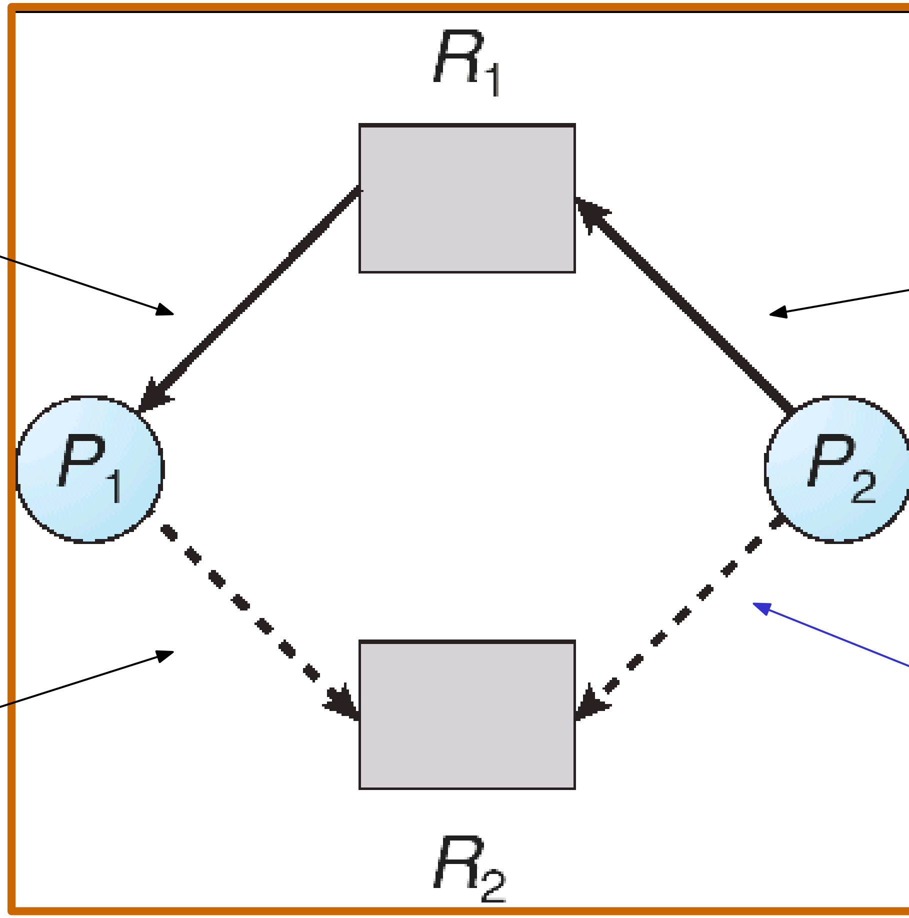
- For multiple instances of a resource type, use the banker's algorithm

Resource-Allocation Graph Scheme

- Introduce a new kind of edge called a <u>claim edge</u>
- *Claim edge* $P_i \longrightarrow R_j$ indicates that process P_j may request resource R_j; which is represented by a dashed line
- A <u>claim edge</u> converts to a <u>request edge</u> when a process **requests** a resource
- A <u>request edge</u> converts to an <u>assignment edge</u> when the resource is **allocated** to the process
- When a resource is **released** by a process, an <u>assignment edge</u> reconverts to a <u>claim edge</u>
- Resources must be **claimed *a priori*** in the system

Resource-Allocation Graph with Claim Edges

Assignm
e nt
edge

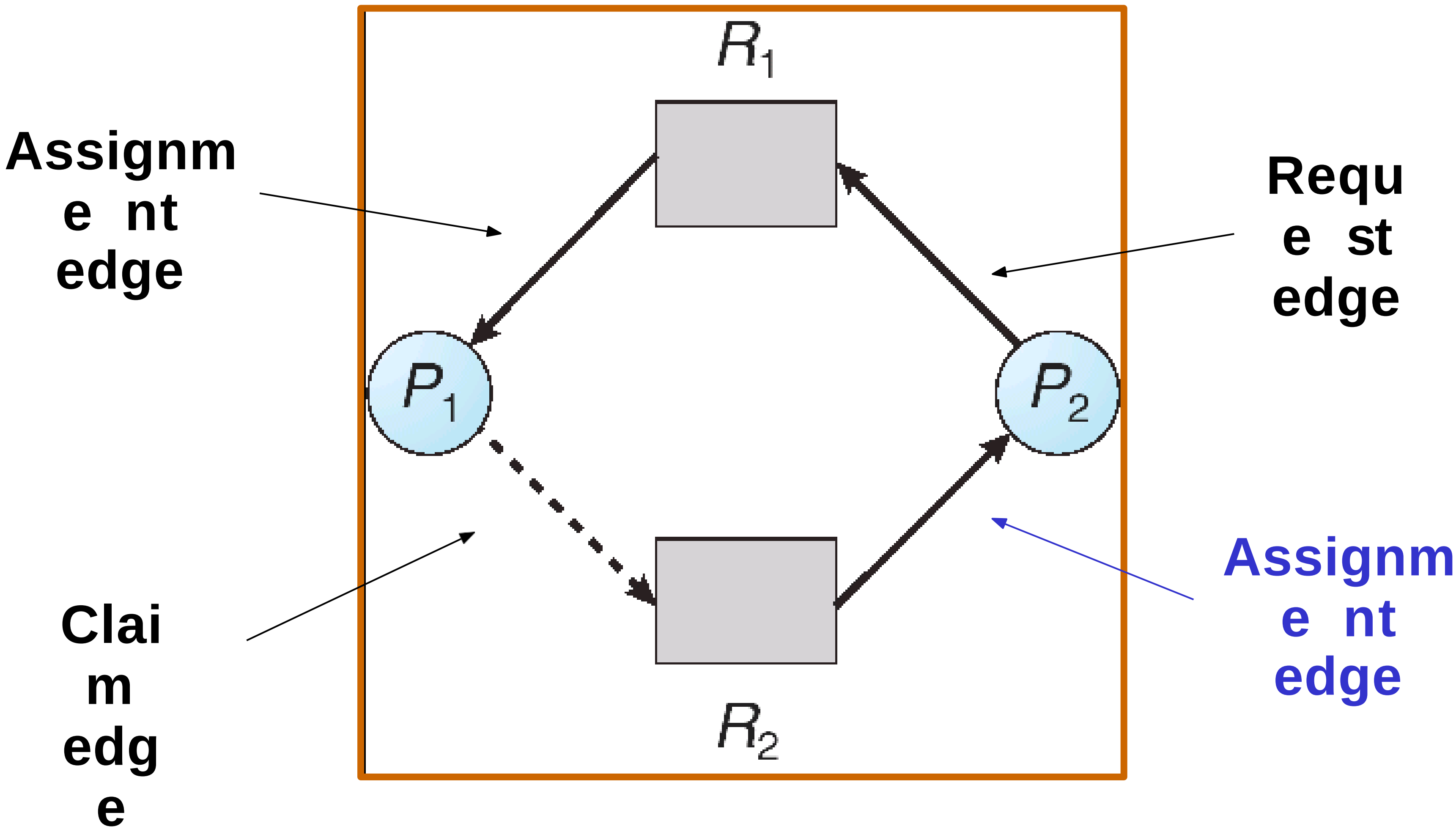


Requ
e st
edge

Clai
m
edg
e

Clai
m
edge

Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the <u>request edge</u> to an <u>assignment edge</u> does not result in the formation of a <u>cycle</u> in the resource allocation graph

Banker's Algorithm

- Used when there exists **multiple** instances of a resource type
- 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 Structures for the Banker's 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***: $n \times m$ matrix. If *Max* $[i,j] = k$, then process P_i may request at most k instances of resource type R_j.
- ***Allocation***: $n \times m$ matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j.
- ***Need***: $n \times m$ 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 - 1,3, …, n.

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 for Process P_i

$Request_i$ = 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:

$$Available := Available - Request_i;$$

$$Allocation_i := Allocation_i + Request_i;$$

$$Need_i := Need_i - Request_i;;$$

- *If safe $\Rightarrow$ the resources are allocated to P_i.*
- *If unsafe $\Rightarrow$ P_i must wait, and the old resource-allocation state is restored*

7.6 Deadlock Detection

Deadlock Detection

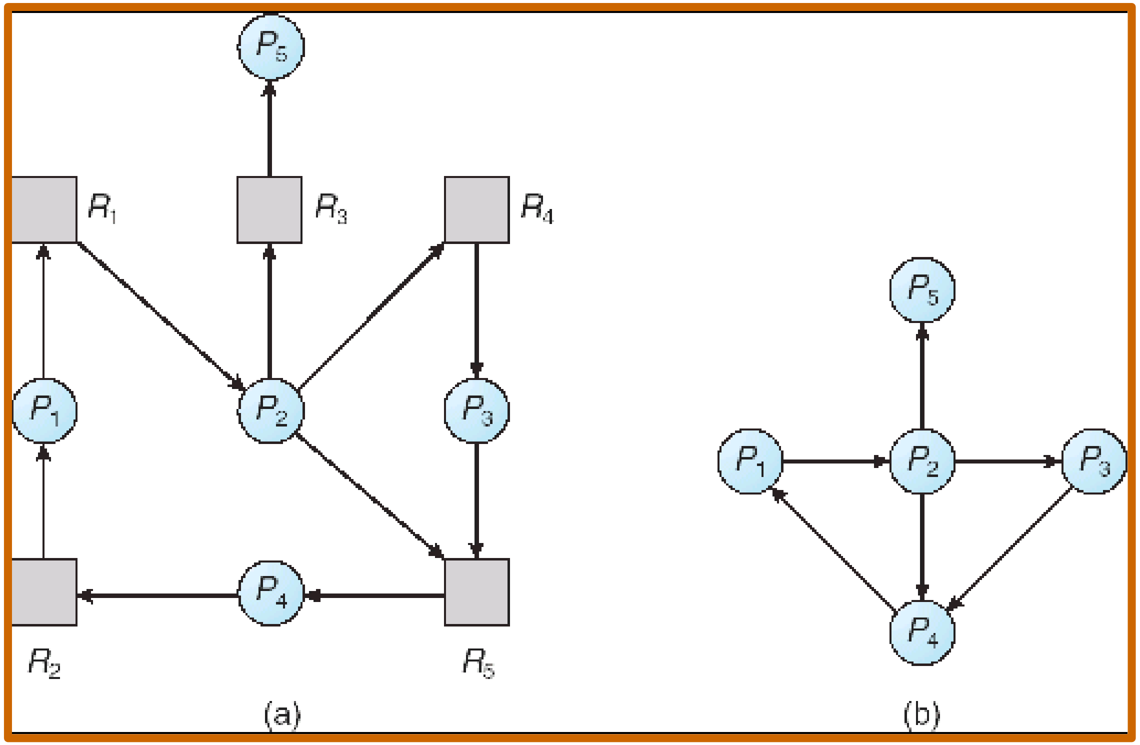
- For deadlock detection, the system must provide
  - An algorithm that examines the state of the system to detect whether a deadlock has occurred
  - And an algorithm to recover from the deadlock
- A detection-and-recovery scheme requires various kinds of overhead
  - Run-time costs of maintaining necessary information and executing the detection algorithm
  - Potential losses inherent in recovering from a deadlock

Single Instance of Each Resource Type

- Requires the creation and maintenance of a *wait-for* graph
  - Consists of a variant of the resource-allocation graph
  - The graph is obtained by **removing** the resource nodes from a resource-allocation graph and **collapsing** the appropriate edges
  - Consequently; all nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j.

- Periodically invoke an algorithm that searches for a cycle in the graph
  - If there is a cycle, there exists a deadlock
  - An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph

Corresponding wait-for graph

Multiple Instances of a Resource Type

Required data structures:

- ***Available:*** A vector of length *m* indicates the number of available resources of each type.
- ***Allocation:*** An n x m matrix defines the number of resources of each type currently allocated to each process.
- ***Request:*** An n x m matrix indicates the current request of each process. If *Request* $[i_j] = k$, then process P_i is requesting k more instances of resource type. R_j.

Detection-Algorithm Usage

- When, and how often, to invoke the detection algorithm depends on:
  - How often is a deadlock likely to occur?
  - How many processes will be affected by deadlock when it happens?

- If the detection algorithm is invoked arbitrarily, there may be **many** cycles in the resource graph and so we would not be able to tell **which one** of the many deadlocked processes "caused" the deadlock
- If the detection algorithm is invoked for every resource request, such an action will incur a considerable **overhead** in computation time
- A less expensive alternative is to invoke the algorithm when CPU utilization drops **below 40%**, for example
  - This is based on the observation that a deadlock eventually cripples system throughput and causes CPU utilization to drop

7.7 Recovery From Deadlock

Recovery from Deadlock

- Two Approaches
  - Process termination
  - Resource preemption

Recovery from Deadlock: Process Termination

- **Abort all deadlocked processes**
  - This approach will break the deadlock, but at great expense

- **Abort one process at a time until the deadlock cycle is eliminated**
  - This approach incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be re-invoked to determine whether any processes are still deadlocked

- **Many factors may affect which process is chosen for termination**
  - What is the priority of the process?
  - How long has the process run so far and how much longer will the process need to run before completing its task?
  - How many and what type of resources has the process used?
  - How many more resources does the process need in order to finish its task?
  - How many processes will need to be terminated?
  - Is the process interactive or batch?

Recovery from Deadlock: Resource Preemption

- With this approach, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken
- When preemption is required to deal with deadlocks, then three issues need to be addressed:
  - **Selecting a victim** – Which resources and which processes are to be preempted?
  - **Rollback** – If we preempt a resource from a process, what should be done with that process?
  - **Starvation** – How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

Summary

- Four necessary conditions must hold in the system for a deadlock to occur
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait
- Four principal methods for dealing with deadlocks
  - Use some protocol to (1) **prevent** or (2) **avoid** deadlocks, ensuring that the system will never enter a deadlock state
  - Allow the system to enter a deadlock state, (3) **detect** it, **and** then **recover**

    Recover by **process termination** or **resource preemption**
  - **(4) Do nothing;** ignore the problem altogether and pretend that deadlocks never occur in the system (used by Windows and Unix)
- To prevent deadlocks, we can ensure that **at least one** of the four necessary conditions **never holds**

File Management

File System Architecture

- Field
- Records
- Files
- Database

Field

Roll No.	Name	Address	Telephone	…….
865472	Jhony	North East P.	985647234	….
….	…..	…..	…..	…..
…..	…..	…..	…..	…..

Record

FILE

File

- File is a named collection of related information that is recorded on a secondary storage.
- It made of fixed length logical records that allow programs to read and write records rapidly in no particular order.

File Concept

- File system is the most visible aspect of an operating system.
- It consists of 2 parts: collection of files(each storing related data) and a directory structure which organizes and provides all the information about all the files in your system.
- File is the named collection of related information.
- File is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user.
- Commonly files represent programs(both source and object forms) and data.

File Attributes

- **Name** – only information kept in human-readable form. Name is usually a string of characters such as example.c. some systems differentiate between upper case and lower case and some don't.
- **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 Operations

- File is an **abstract data type. To define a file properly, we need to consider some operations which can be performed on files**
  - **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 Structure

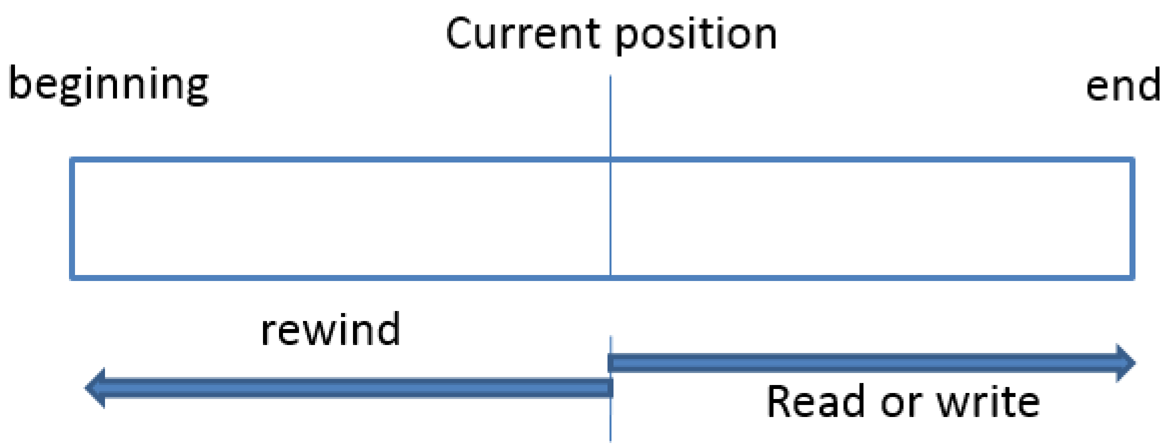
- Internal File Structure
- External File structure
  - Operating system provides various file structures for managing files.
  - Also provides set of special operations for manipulating files with these file structures.

Access Methods

- Files store information and this information must be accessed and read into computer memory.
- The information in the file can be accessed in several ways.

Sequential Access

- Information is processed on one order, one record after the other.
- A read operation—*read next*—reads the next portion of the file and automatically
  advances a file pointer, which tracks the I/O location. Similarly, the write operation—*write next*—appends to the end of the file and advances to the end of the newly written material (the new end of file).



Direct Access

- In direct access file is seen as a numbered sequence of blocks or records.
- Immediate access to large amount of information.
- The operations include:
  - *read n,* where *n* is the block number, rather than *read next.*
  - *write n,* where *n* is the block number, rather than *write next.*
  - *position to n,* where *n* is the block number.

- Sequential Access

  read next
  write next
  reset

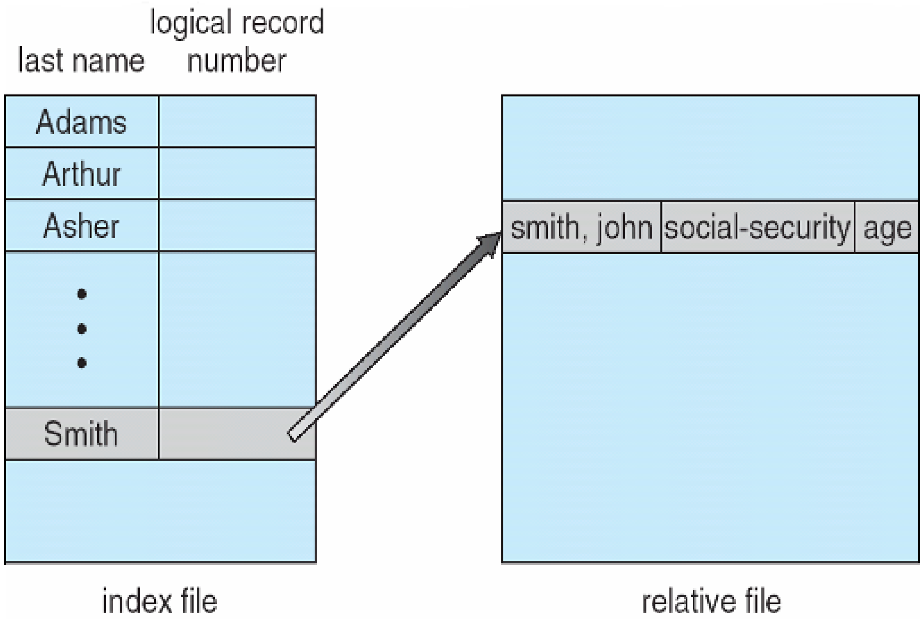
- Direct Access

  read *n*
  write *n*
  position to *n*
  read next
  write next

n = relative block number

Indexed Access Method

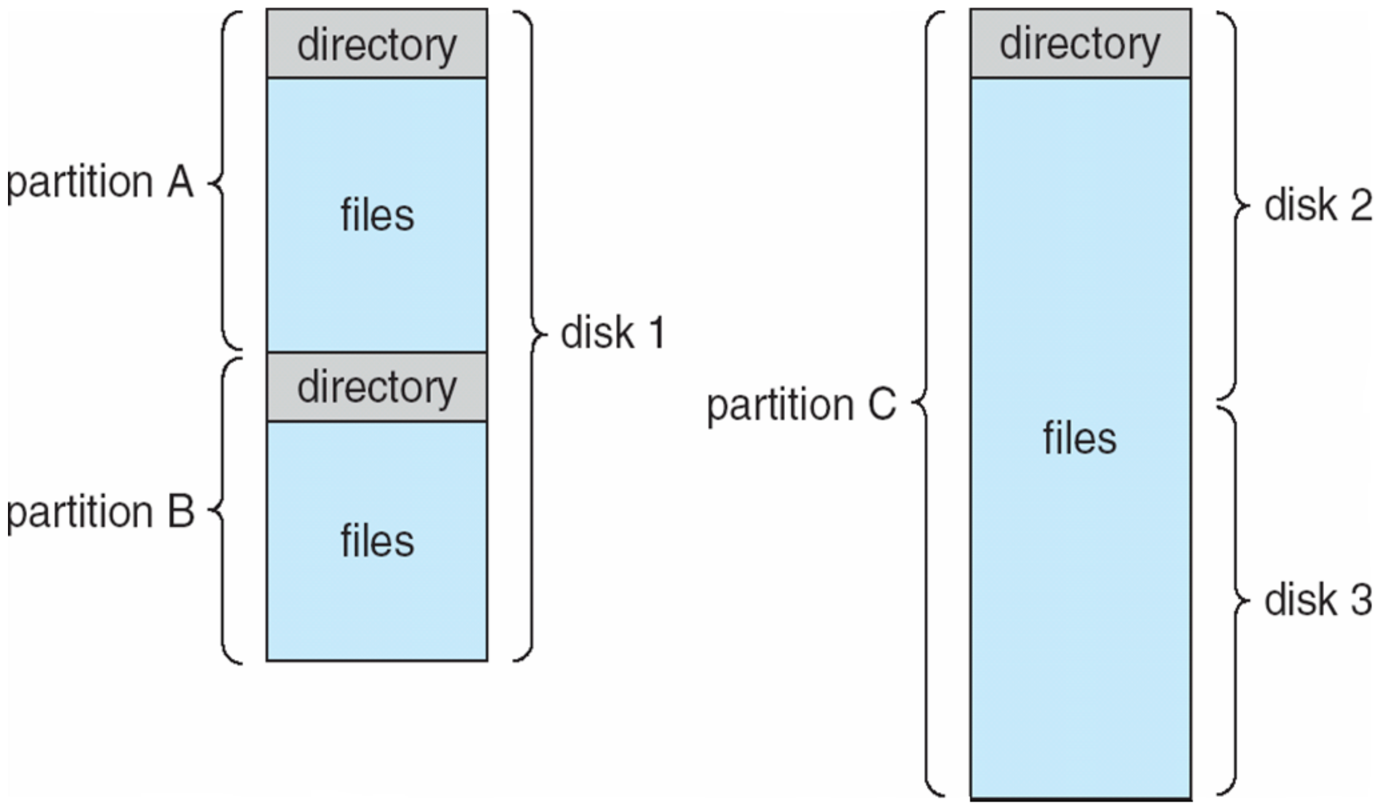
- An *index* is associated with the file containing pointers to the blocks.
- To find the record in the file, we search the index and then use the pointer to access the file directly and to find the desired record.



Disk Structure

- As the number of files increase the issue of managing the files becomes an issue.
- The organization is done in two parts, Firstly:
  - Disk can be subdivided into partitions
  - Disks or partitions can be RAID protected against failure
  - Disk or partition can be formatted with a file system
  - Partitions also known as minidisks, slices or volume
- Secondly, Each volume containing file system also tracks that file system's info in device directory or volume table of contents.

A Typical File-system Organization

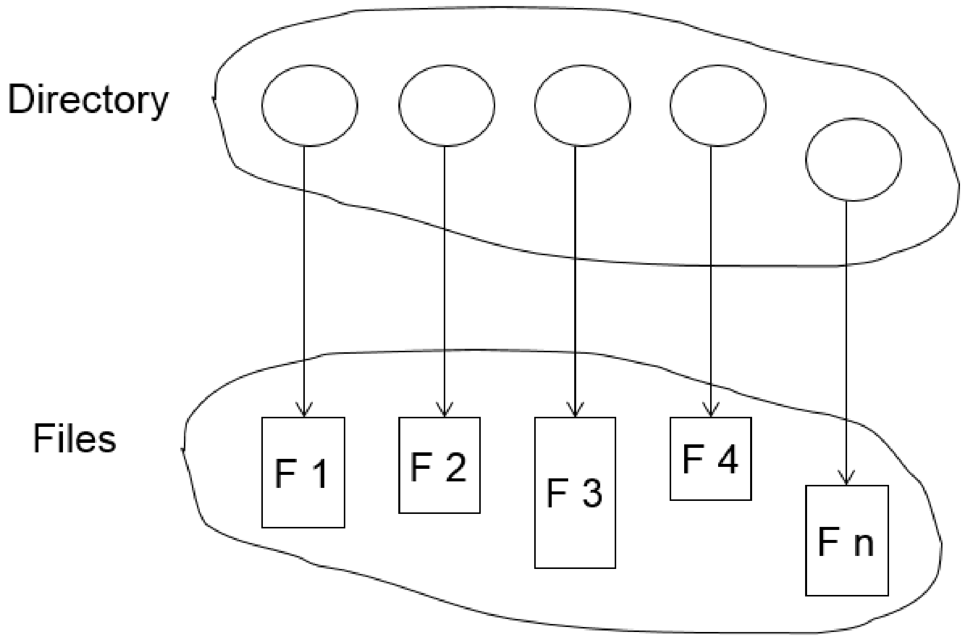


Operations Performed on Directory

- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system

Directory Structure

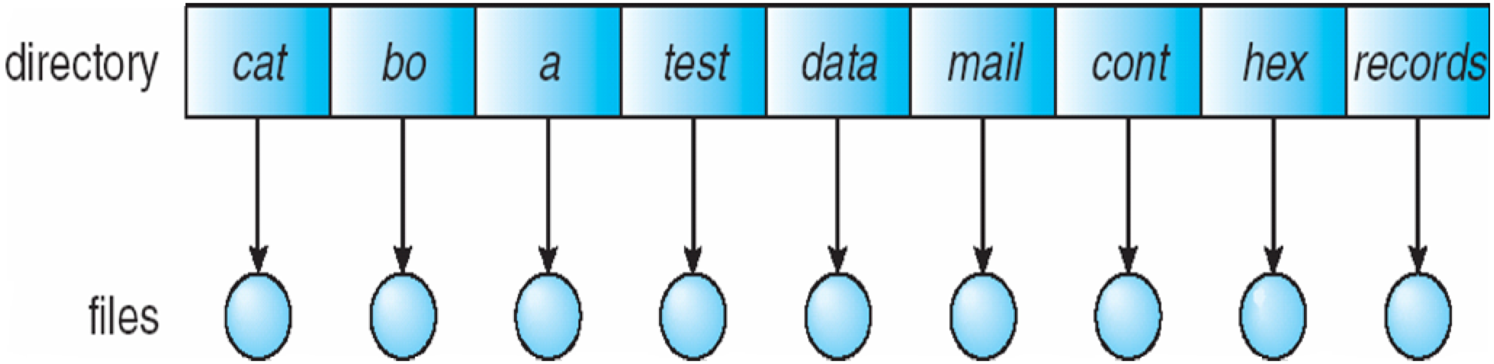
- The directory contains information for all files.



- Both the directory structure and the files reside on disk

Single-Level Directory

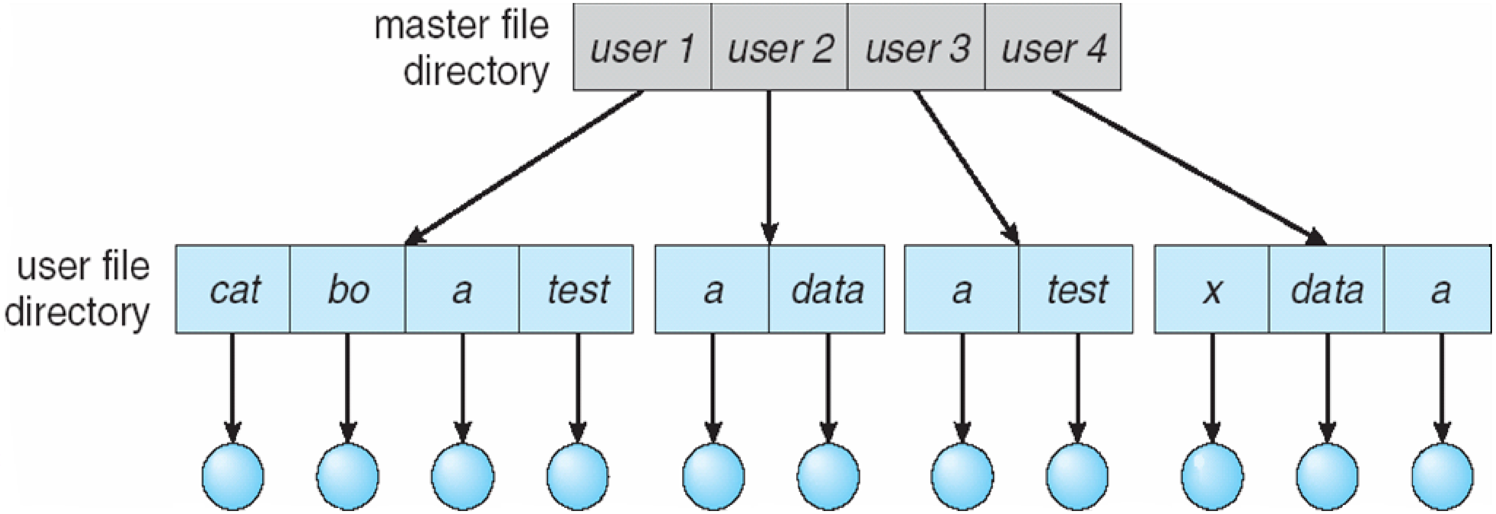
- A single directory for all users



- Easy to support and understand.
- Limitation:
- When number of files increases or when the system has more than one user, then Naming problem occurs. All files should have unique names.

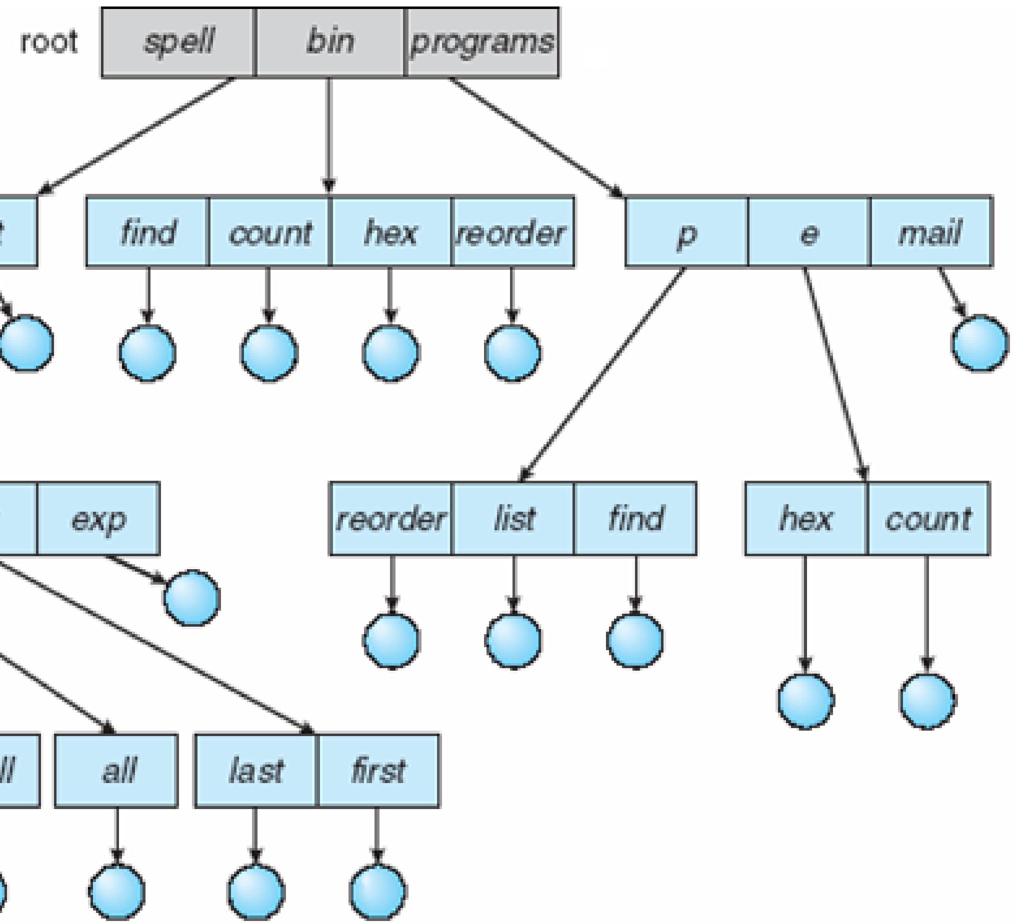
Two-Level Directory

- In two level directory, each user has his own user file directory(UFD). UFDs have the similar structure, but each lists only the files of a single user.



- Path name
- Can have the same file name for different user
- Efficient searching

Tree-Structured Directories



Tree-Structured Directories (Cont)

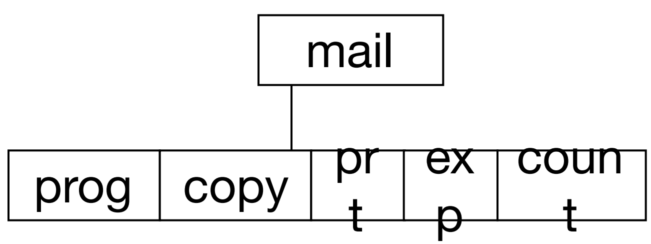
- **Absolute** or **relative** path name
- Creating a new file is done in current directory
- Delete a file

  rm <file-name>
- Creating a new subdirectory is done in current directory

  mkdir <dir-name>

  Example: if in current directory /mail

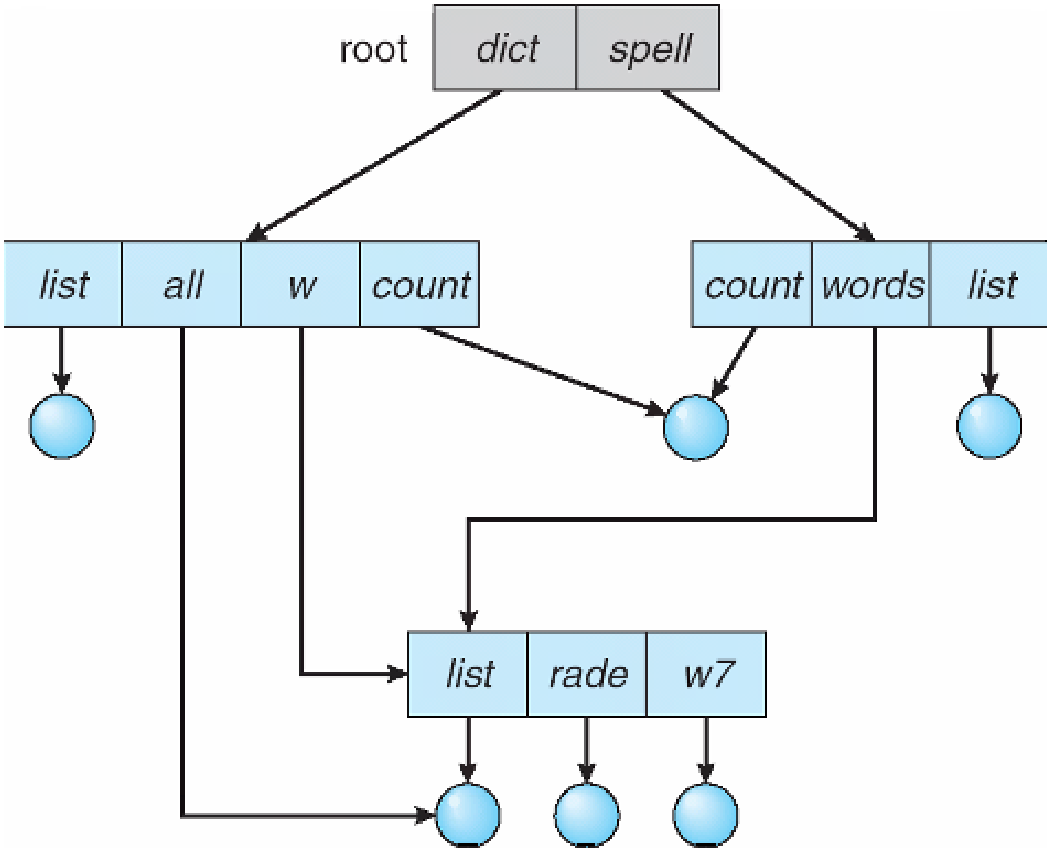
  mkdir count



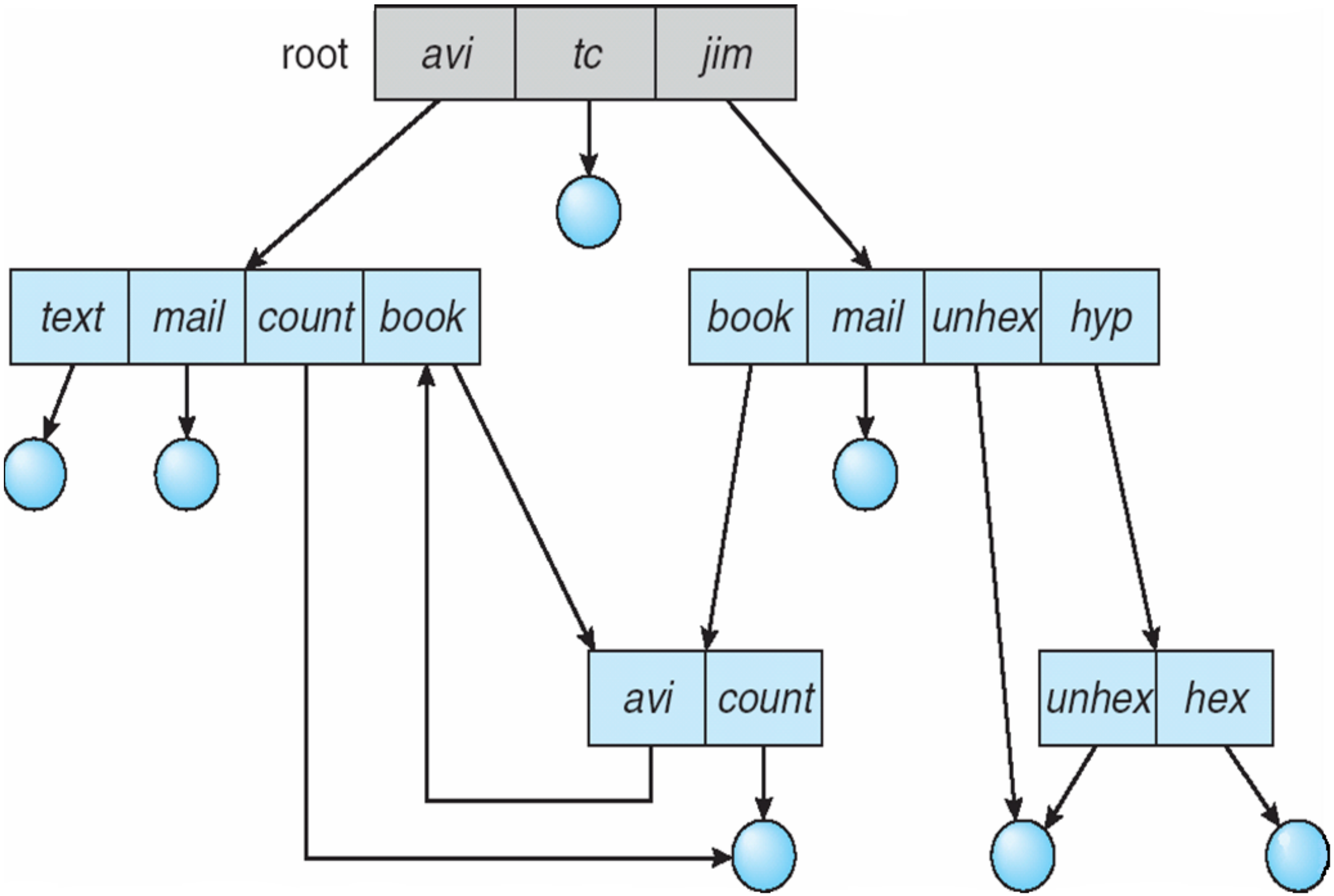
Deleting “mail” ⇒ deleting the entire subtree rooted by “mail”

Acyclic-Graph Directories

- Have shared subdirectories and files



General Graph Directory



Directory Implementation

1. **Linear List**
   - Simple to program
   - Time consuming to execute
   - Create new file
     - Make sure it's a unique file name
     - Add entry at end of directory
   - Delete a file
     - Search for the file
     - Release the space allocated to it
     - Mark the space as unused or attach it to list of free directory entries or copy the last directory entry here.

2. Hash Table

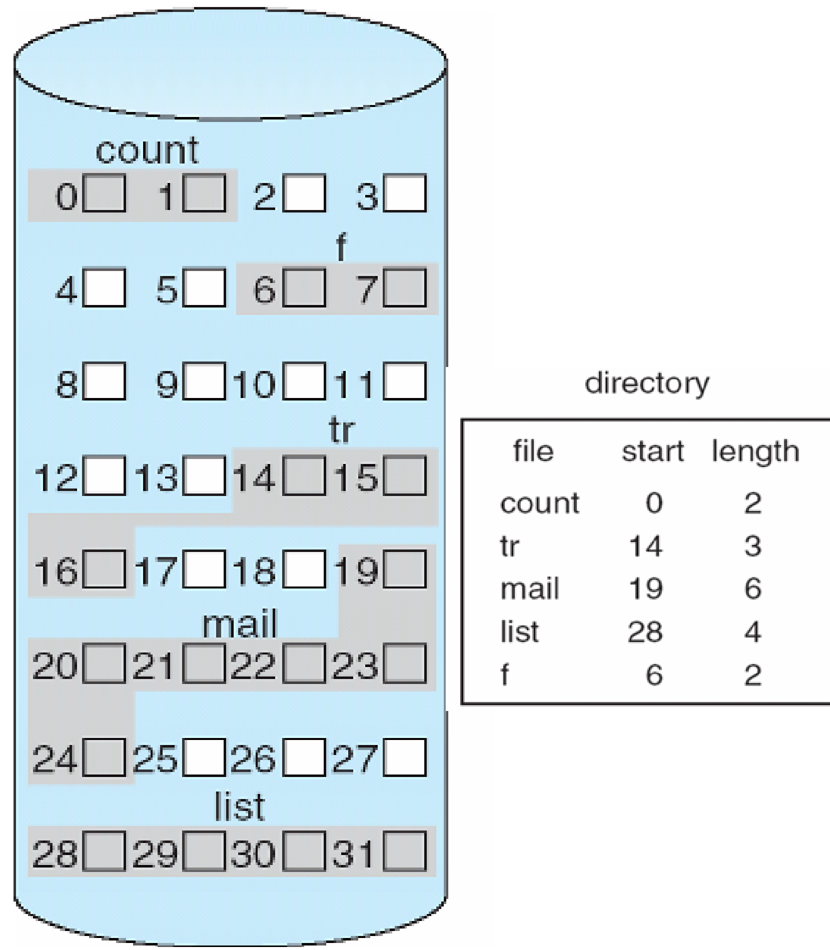
- Linear list + hash structure
- Compute hash value of file names and returns pointer to file in the linear list
- Decreases search time
- Collisions are to be avoided in case two file names hash to same location
- Disadvantage – hash functions are of fixed size (0-64 or 0-128 etc)

Allocation Methods

- An allocation method refers to how disk blocks are allocated for files so that disk space is utilized effectively and files can be accessed quickly.
- **Contiguous allocation**
- **Linked allocation**
- **Indexed allocation**

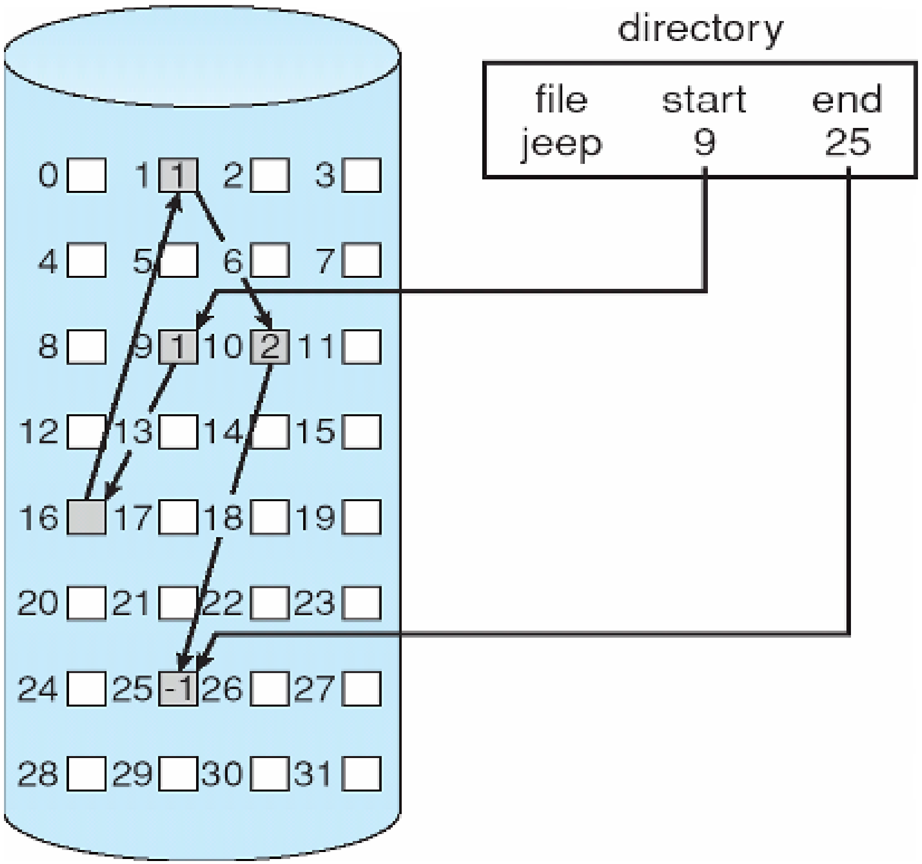
Allocation Methods

- Contiguous Allocation
  - File occupy contiguous blocks
  - Directory entry contains
    - File name
    - Starting block
    - Length (total blocks)
  - Access possible
    - Sequential
    - Direct
  - Problems
    - Finding space for new file
    - External fragmentation
    - Determining space requirement of a file

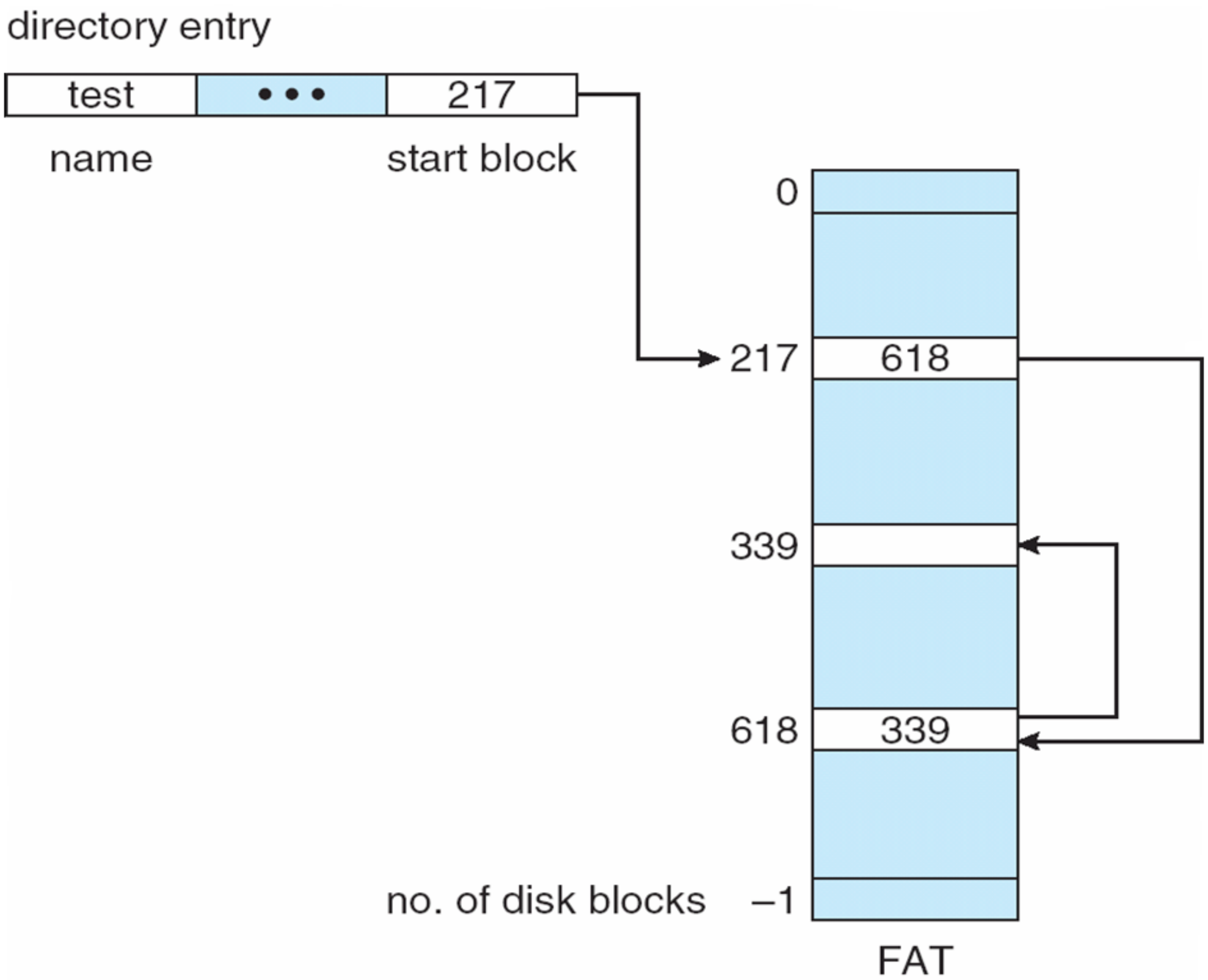


Linked Allocation

- Directory structure contains
  - Pointer to first and last block of file
- Advantages
  - No external fragmentation
  - No issue with increase in file size
- Disadvantages
  - Only sequential access
  - Reliability – loss of a pointer
  - Space required for pointers
  - Solution: make *cluster* of blocks
    - Problem: internal fragmentation

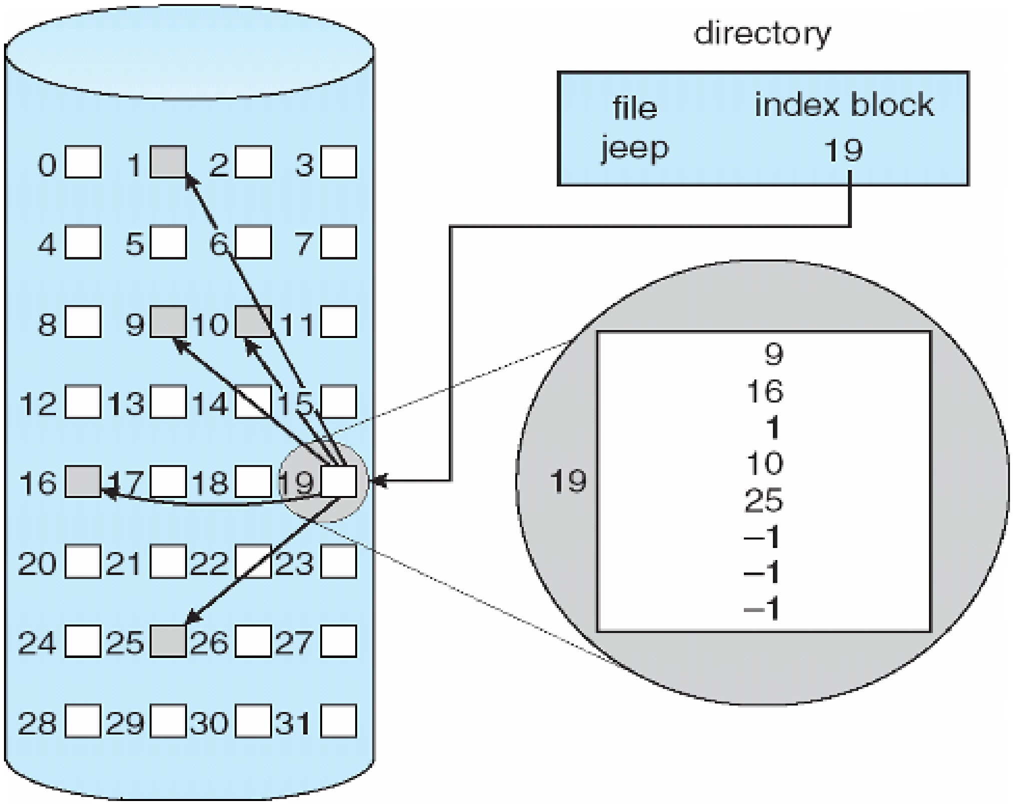


Example - FAT

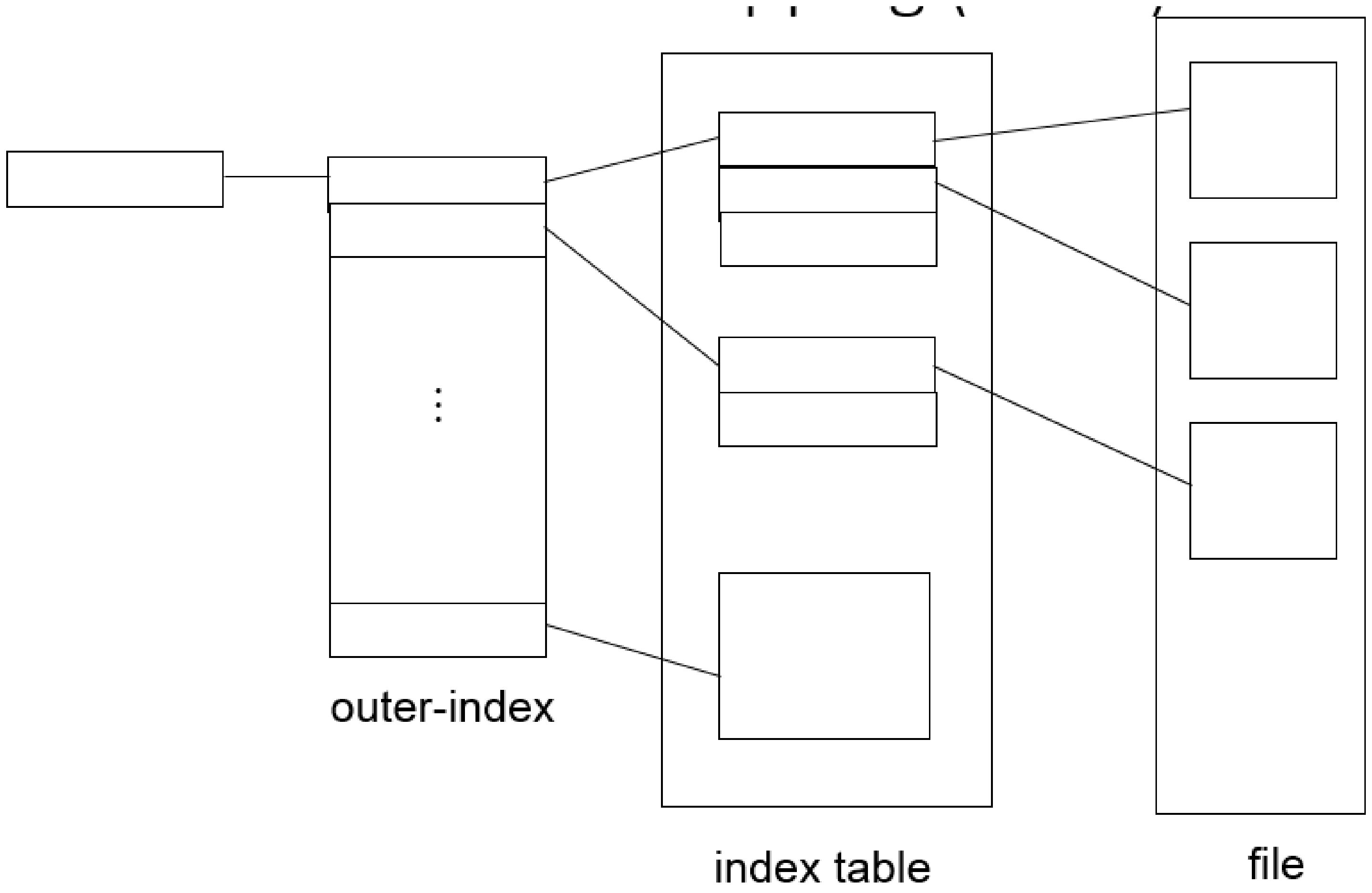

directory entry
test
• • •
217
name
start block
0
217
618
339
618
339
no. of disk blocks
–1
FAT

Indexed Allocation

- Clubs all the pointers into one block – index block
- Directory entry contains
  - File name
  - Index block number
- Access
  - Direct
- Issue
  - Size of index block
  - Sol: multilevel indexing



Indexed Allocation – Mapping (Cont.)



Performance

- Contiguous
  - Requires only 1 access to get a disk block
- Linked
  - Requires *i* disk reads to read i^{th} block
- Indexed
  - Depends on
    - level of indexing
    - Size of file
    - Position of desired block

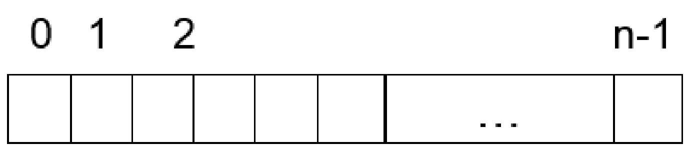
Free Space management

Disk space is limited, the space from deleted files is reused for new files, if possible. To keep track of free disk space, the system maintains a **free-space list.**

1. Bit vector
2. Linked list
3. Grouping
4. Counting

Bit Vector

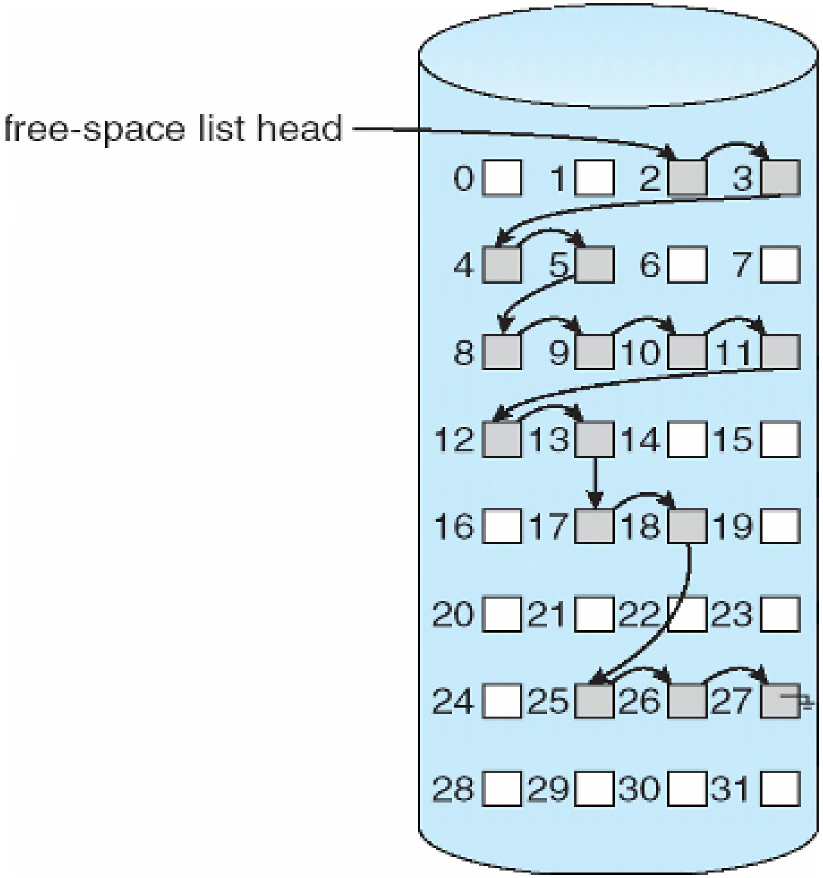
- The free-space list is implemented as a bit **map** or bit vector. Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0.



$$\text{bit}[i] = \begin{cases} 0 \Rightarrow \text{block}[i] \text{ free} \\ 1 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

Linked List

- Link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory.



Grouping

- Store the addresses of *n* free blocks in the first free block.
- The first n—1 of these blocks are actually free.
- The last block contains the addresses of another *n* free blocks, and so on.

Counting

- Keep the address of the first free block and the number *n* of free contiguous blocks that follow the first block.
- Each entry in the free-space list then consists of a disk address and a count.
- Each entry requires more space than would a simple disk address, overall list will be shorter, as long as the count is generally greater than 1.

Summary

- Files
- File Structure
- Access Methods
- Directory Structure
- Directory Implementation
- Allocation Methods
- Free Space Management