Threads

Threads

Despite of the fact that a thread must execute in process, the process and its associated threads are different concept. Processes are used to group resources together and threads are the entities scheduled for execution on the CPU.
A thread is a single sequence stream within in a process*. Because threads have some of the properties of processes, they are sometimes called *lightweight processes. In a process, threads allow multiple executions of streams. In many respect, threads are popular way to improve application through parallelism. The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel. Like a traditional process i.e., process with one thread, a thread can be in any of several states (Running, Blocked, Ready or Terminated). Each thread has its own stack. Since thread will generally call different procedures and thus a different execution history. This is why thread needs its own stack. An operating system that has thread facility, the basic unit of CPU utilization is a thread. A thread has or consists of a program counter (PC), a register set, and a stack space. Threads are not independent of one other like processes as a result threads shares with other threads their code section, data section, OS resources also known as task, such as open files and signals.

Processes Vs Threads

As we mentioned earlier that in many respect threads operate in the same way as that of processes. Some of the similarities and differences are:

Similarities

- Like processes threads share CPU and only one thread active (running) at a time.
- Like processes, threads within a processes execute sequentially.
- Like processes, thread can create children.
- And like process, if one thread is blocked, another thread can run.

Differences

- Unlike processes, threads are not independent of one another.
- Unlike processes, all threads can access every address in the task .
- Unlike processes, thread are design to assist one other. Note that processes might or might not assist one another because processes may originate from different users.

Why Threads?

Following are some reasons why we use threads in designing operating systems.

1. A process with multiple threads make a great server for example printer server.
2. Because threads can share common data, they do not need to use interprocess communication.
3. Because of the very nature, threads can take advantage of multiprocessors.

Threads are cheap in the sense that

1. They only need a stack and storage for registers therefore, threads are cheap to create.
2. Threads use very little resources of an operating system in which they are working. That is, threads do not need new address space, global data, program code or operating system resources.

3. Context switching are fast when working with threads. The reason is that we only have to save and/or restore PC, SP and registers.

But this cheapness does not come free - the biggest drawback is that there is no protection between threads.

User-Level Threads

User-level threads implement in user-level libraries, rather than via systems calls, so thread switching does not need to call operating system and to cause interrupt to the kernel. In fact, the kernel knows nothing about user-level threads and manages them as if they were single-threaded processes.

Advantages:

The most obvious advantage of this technique is that a user-level threads package can be implemented on an Operating System that does not support threads. Some other advantages are

- User-level threads does not require modification to operating systems.
- Simple Representation:
  Each thread is represented simply by a PC, registers, stack and a small control block, all stored in the user process address space.
- Simple Management:
  This simply means that creating a thread, switching between threads and synchronization between threads can all be done without intervention of the kernel.
- Fast and Efficient:
  Thread switching is not much more expensive than a procedure call.

Disadvantages:

- There is a lack of coordination between threads and operating system kernel. Therefore, process as whole gets one time slice irrespect of whether process has one thread or 1000 threads within. It is up to each thread to relinquish control to other threads.
- User-level threads requires non-blocking systems call i.e., a multithreaded kernel. Otherwise, entire process will blocked in the kernel, even if there are runable threads left in the processes. For example, if one thread causes a page fault, the process blocks.

Kernel-Level Threads

In this method, the kernel knows about and manages the threads. No runtime system is needed in this case. Instead of thread table in each process, the kernel has a thread table that keeps track of all threads in the system. In addition, the kernel also maintains the traditional process table to keep track of processes. Operating Systems kernel provides system call to create and manage threads.

Advantages:

- Because kernel has full knowledge of all threads, Scheduler may decide to give more time to a process having large number of threads than process having small number of threads.
- Kernel-level threads are especially good for applications that frequently block.

Disadvantages:

- The kernel-level threads are slow and inefficient. For instance, threads operations are hundreds of times slower than that of user-level threads.

- Since kernel must manage and schedule threads as well as processes. It require a full thread control block (TCB) for each thread to maintain information about threads. As a result there is significant overhead and increased in kernel complexity.

Advantages of Threads over Multiple Processes

- **Context Switching** Threads are very inexpensive to create and destroy, and they are inexpensive to represent. For example, they require space to store, the PC, the SP, and the general-purpose registers, but they do not require space to share memory information, Information about open files of I/O devices in use, etc. With so little context, it is much faster to switch between threads. In other words, it is relatively easier for a context switch using threads.
- **Sharing** Treads allow the sharing of a lot resources that cannot be shared in process, for example, sharing code section, data section, Operating System resources like open file etc.

Disadvantages of Threads over Multiprocesses

- **Blocking** The major disadvantage if that if the kernel is single threaded, a system call of one thread will block the whole process and CPU may be idle during the blocking period.
- **Security** Since there is, an extensive sharing among threads there is a potential problem of security. It is quite possible that one thread over writes the stack of another thread (or damaged shared data) although it is very unlikely since threads are meant to cooperate on a single task.

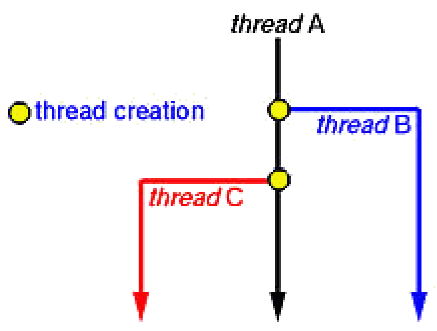
Application that Benefits from Threads

A proxy server satisfying the requests for a number of computers on a LAN would be benefited by a multi-threaded process. In general, any program that has to do more than one task at a time could benefit from multitasking. For example, a program that reads input, process it, and outputs could have three threads, one for each task.

Application that cannot Benefit from Threads

Any sequential process that cannot be divided into parallel task will not benefit from thread, as they would block until the previous one completes. For example, a program that displays the time of the day would not benefit from multiple threads.

Resources used in Thread Creation and Process Creation



When a new thread is created it shares its code section, data section and operating system resources like open files with other threads. But it is allocated its own stack, register set and a program counter.

The creation of a new process differs from that of a thread mainly in the fact that all the shared resources of a thread are needed explicitly for each process. So though two processes may be running the same piece of code they need to have their own copy of the code in the main memory to be able to run. Two processes also do not share other resources with each other. This makes the creation of a new process very costly compared to that of a new thread.

Context Switch

To give each process on a multiprogrammed machine a fair share of the CPU, a hardware clock generates interrupts periodically. This allows the operating system to schedule all processes in main memory (using scheduling algorithm) to run on the CPU at equal intervals. Each time a clock interrupt occurs, the interrupt handler checks how much time the current running process has used. If it has used up its entire time slice, then the CPU scheduling algorithm (in kernel) picks a different process to run. Each switch of the CPU from one process to another is called a context switch.

Major Steps of Context Switching

- The values of the CPU registers are saved in the process table of the process that was running just before the clock interrupt occurred.
- The registers are loaded from the process picked by the CPU scheduler to run next.

In a multiprogrammed uniprocessor computing system, context switches occur frequently enough that all processes appear to be running concurrently. If a process has more than one thread, the Operating System can use the context switching technique to schedule the threads so they appear to execute in parallel. This is the case if threads are implemented at the kernel level. Threads can also be implemented entirely at the user level in run-time libraries. Since in this case no thread scheduling is provided by the Operating System, it is the responsibility of the programmer to yield the CPU frequently enough in each thread so all threads in the process can make progress.

Action of Kernel to Context Switch Among Threads

The threads share a lot of resources with other peer threads belonging to the same process. So a context switch among threads for the same process is easy. It involves switch of register set, the program counter and the stack. It is relatively easy for the kernel to accomplished this task.

Action of kernel to Context Switch Among Processes

Context switches among processes are expensive. Before a process can be switched its process control block (PCB) must be saved by the operating system. The PCB consists of the following information:

- The process state.
- The program counter, PC.
- The values of the different registers.
- The CPU scheduling information for the process.
- Memory management information regarding the process.
- Possible accounting information for this process.
- I/O status information of the process.

When the PCB of the currently executing process is saved the operating system loads the PCB of the next process that has to be run on CPU. This is a heavy task and it takes a lot of time.

File Management

File Concept

File Concept

- File is an abstract data type
- named collection of related information
- recorded on secondary storage
- file is the smallest allotment of logical secondary storage
- 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
  - indexed files
- Who decides:
  - Operating system
  - Program

File Attributes

- Name – only information kept in human-readable form.
- 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

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

File Types: Name Extension

- Unix uses "magic number" that indicates the file
- The Macintosh has a type and a creator attribute (program that created it)
- Note: On Unix the extension is generally advisory. In Windows it is not.

File Type	**Usual Extension**	**function**
executable	exe, com, bin or	ready to run machine language programs

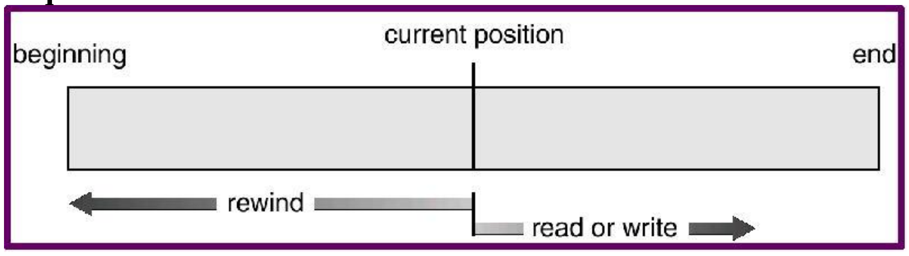
	none	
object	obj, o	compiled, machine language. Not linked
source code	c, cc, java, .pas, asm, etc	source code in various languages. Actually text files
batch/script	bat, sh, vbs, cmd, etc	commands for command interpreter(s)
text	txt, doc, none	text files
word processor	wp, tex, rtf, doc, sxw, etc	various word processor formats
library	lib, a, so, dll, class	libraries of routines for programming
print or view	ps, pdf, jpg, gif	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar, tgz	related files grouped into one. Sometimes compressed for archiving or storage
multimedia	mpeg, mpg, mov, rm, etc	binary file containing audio or A/V information

File Access Methods

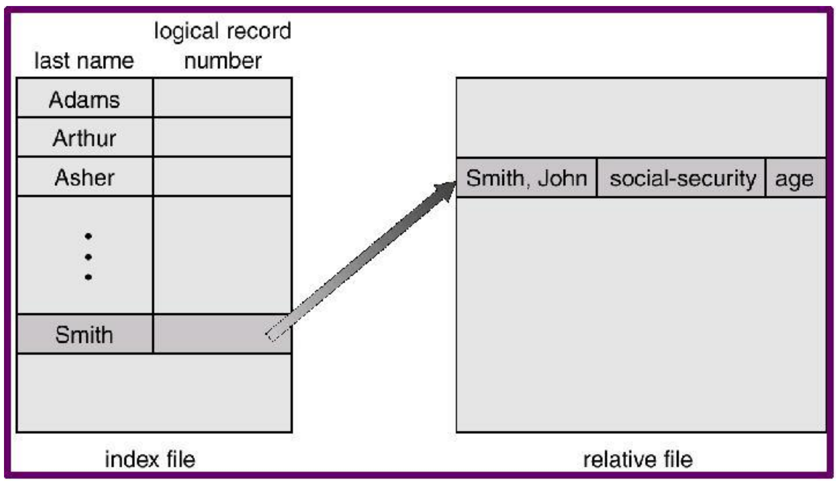
- Sequential Access
  - read next
  - write next
  - reset
  - no read after last write (rewrite)
- Direct (random) Access
  - read n
  - write n
  - position to n
    - read next
    - write next
  - rewrite n

n = relative block number

Sequential Access



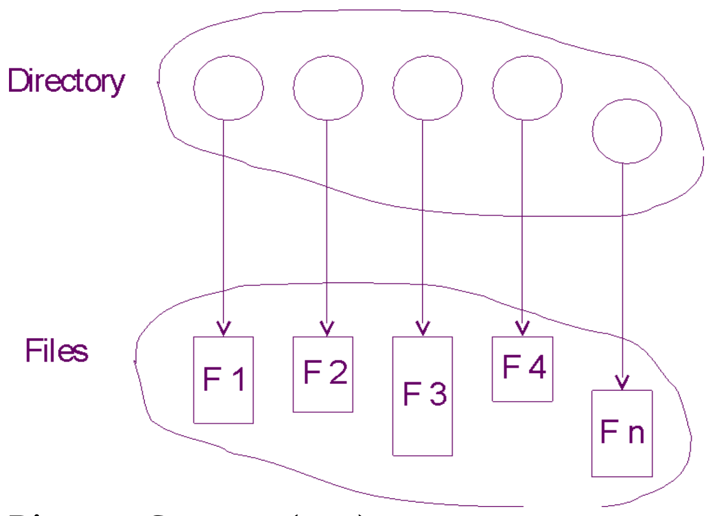
Indexed and Relative files



Directories

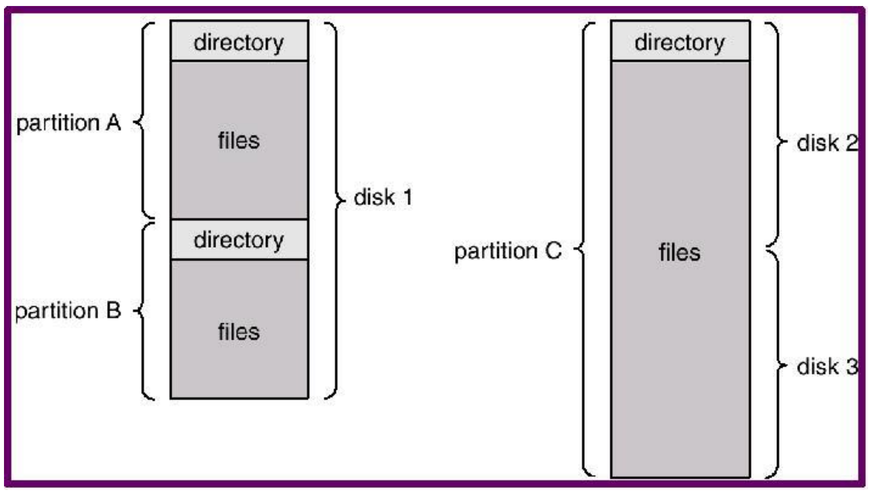
Directory Structures

- A collection of nodes containing information about all files.



Directory Structure (cont)

- Each disk is split into one or more *partitions* (*minidisks*, *volumes*)
- Multiple disks may actually be grouped in some systems
- Each partition contains information about files within it: device directory or volume table of contents



Information in a Device Directory

- Name
- Type
- Address
- Current length
- Maximum length
- Date last accessed (for archival)
- Date last updated (for dump)
- Owner ID (who pays)
- Protection information (discuss later)

Operations Performed on Directory

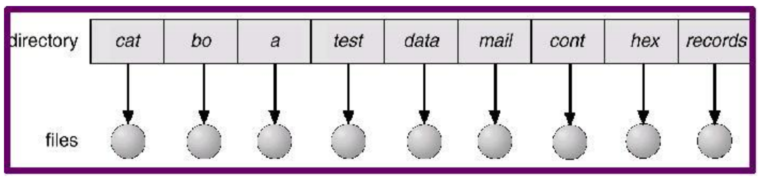
- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system

Directory Organization

- Efficiency – locating a file quickly.
- Naming – convenient to users.
- Two users can have same name for different files.
- The same file can have several different names.
- Grouping – logical grouping of files by properties, (e.g., all Java programs, all games, …)

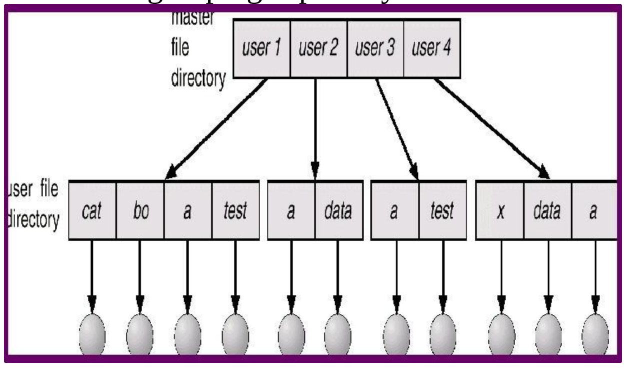
Single-Level Directory

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

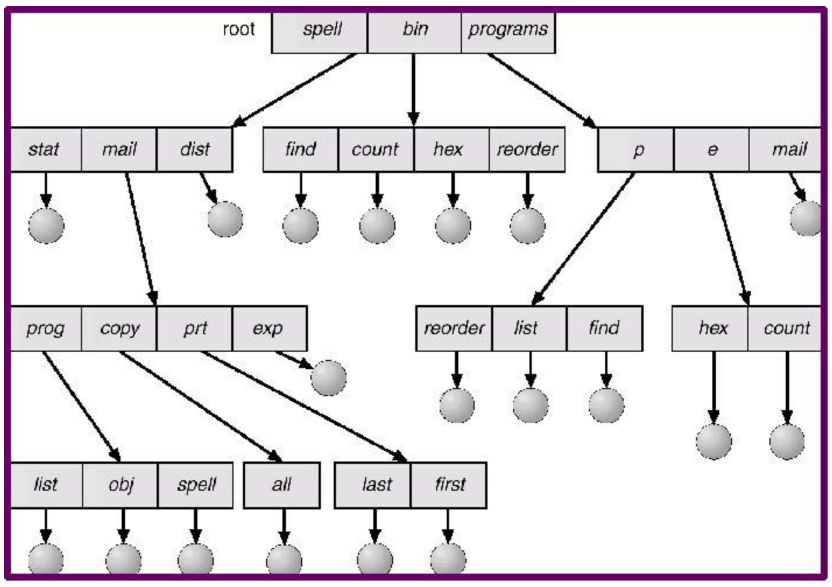


Two-level Directory

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

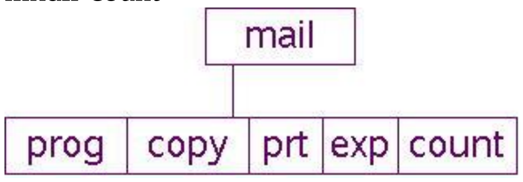


Tree Structured Directories



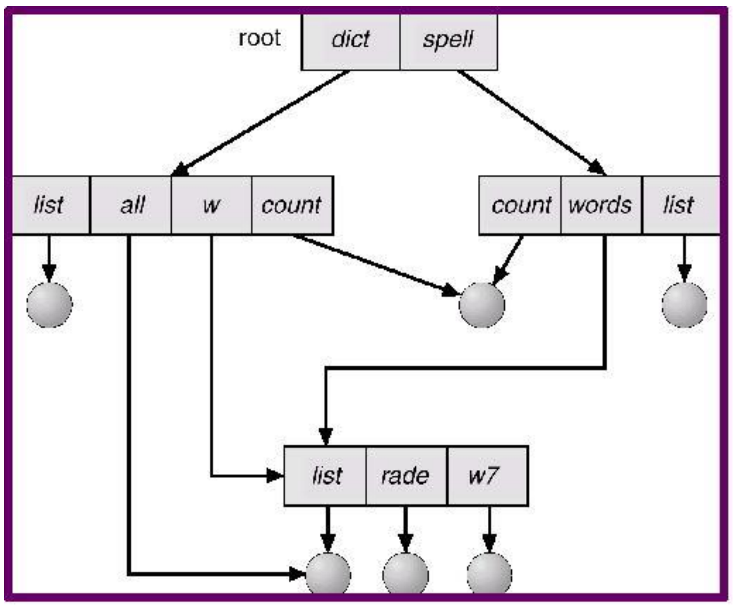
Tree Structured Directories (cont)

- Efficient searching
- Grouping Capability
- Current directory (working directory)
  - cd /spell/mail/prog
  - type list
- 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.
  - Example: if in current directory /mail
    - mkdir count



Acyclic Graph Directories

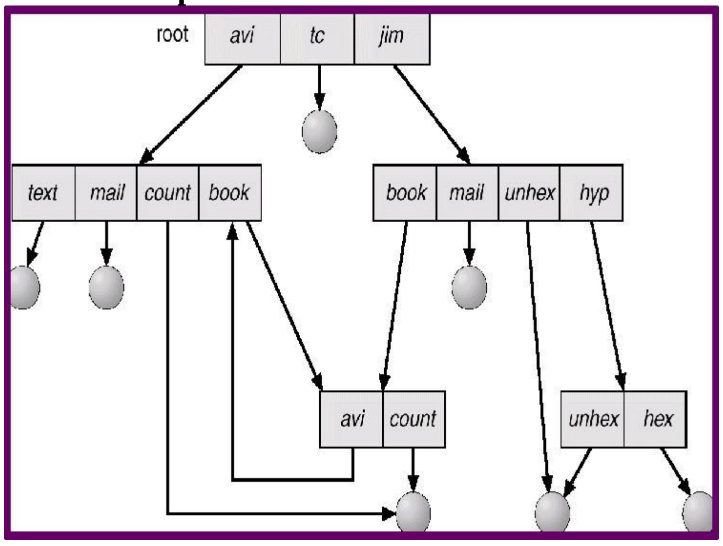
Have shared subdirectories and files.



Acyclic Graphs (cont)

- Two different names (aliasing)
- If dict deletes list dangling pointer.
- Solutions:
  - Backpointers, so we can delete all pointers.
  - Variable size records a problem.
  - Backpointers using a daisy chain organization.
  - Entry-hold-count solution.

General Graphs



General Graphs (cont)

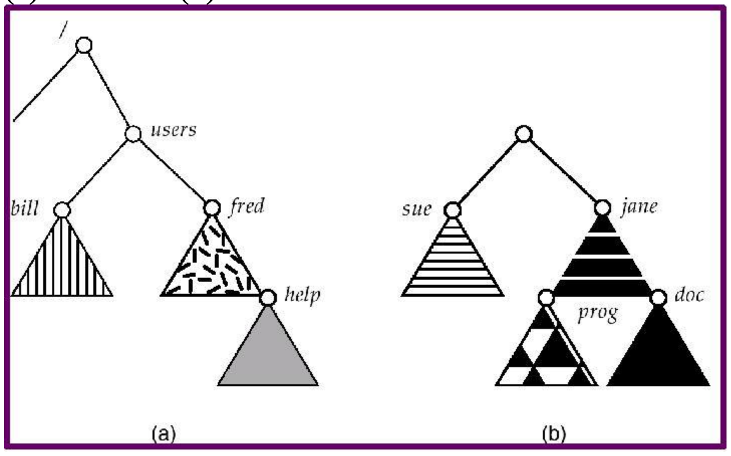
How do we guarantee no cycles?

- Allow only links to file not subdirectories.
- Garbage collection.
- Every time a new link is added use a cycle detection algorithm to determine whether it is OK.

File System Mounting

- A file system must be mounted before it can be accessed.
- A unmounted file system (I.e. Fig. 11-11(b)) is mounted at a mount point.

(a) Mounted. (b) Unmounted Partition



File Sharing and Protection
File Sharing and Protection

- 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.

Protection

- File owner/creator should be able to control:
  - what can be done
  - by whom
- Types of access
  - Read
  - Write
  - Execute
  - Append
  - Delete
  - List

Access Lists and Groups

- Found in Unix-like systems
- Mode of access: read, write, execute
- Three classes of users

```
•                                   R W X
•          a) owner access    7    ==>    1 1 1
•          b) group access    6    ==>    1 1 0                    RWX
       c) public access    1    ==>    0 0 1
```

- To share a file
  - Ask manager to create a group (unique name), say G, and add some users to the group.
  - Attach a group to a file or directory
    chgrp G game
  - Set protection for the file/directory
    chmod 761 game

ACL's

- More general file protection scheme
- For each file we need to associate a list of users and their permitted access
- this may be a tedious task
- this is a variable length list

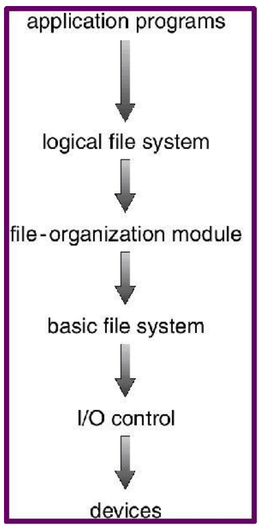
File System Implementation

- File System Structure
- File System Implementation
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- Recovery
- Log-Structured File Systems
- Network File Systems

File System Structure
File System Structure

- File structure
  - Logical storage unit
  - Collection of related information
- File system resides on secondary storage (disks).
- File system organized into layers.
- File control block – storage structure consisting of information about a file.

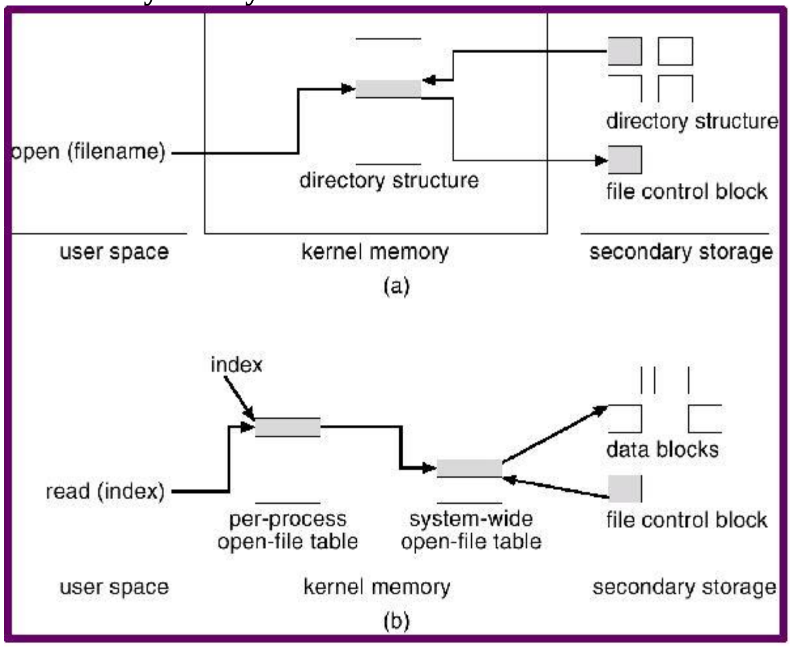
Layered File System



File Control Block

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks

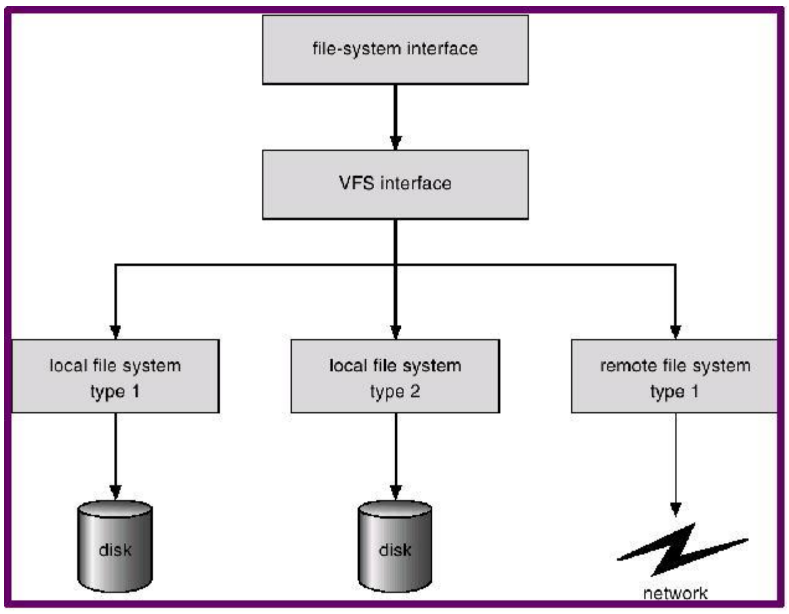
In Memory File System Structures



Virtual File Systems

- Virtual File Systems (VFS) provide an object-oriented way of implementing file systems.
- VFS allows the same system call interface (the API) to be used for different types of file systems.
- The API is to the VFS interface, rather than any specific type of file system.

Virtual File Systems: Schematic View



Directory Implementation

Linear list of file names with pointer to the data blocks

- simple to program
- time-consuming to execute

Hash Table – linear list with hash data structure.

- decreases directory search time
- collisions – situations where two file names hash to the same location
- fixed size

Allocation Methods

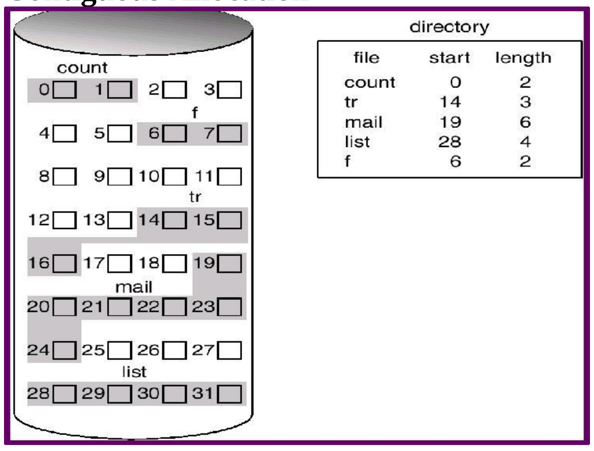
An allocation method refers to how disk blocks are allocated for files:

- Contiguous allocation
- Linked allocation
- Indexed allocation

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.

Contiguous Allocation



directory

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

Extent Based Sytems

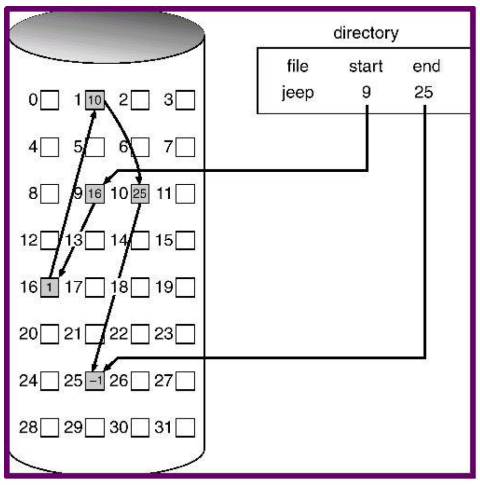
- Many newer file systems (I.e. Veritas File System) use a modified contiguous allocation scheme.
- Extent-based file systems allocate disk blocks in extents.
- An extent is a contiguous block of disks. Extents are allocated for file allocation. A file consists of one or more extents.

Linked Allocation

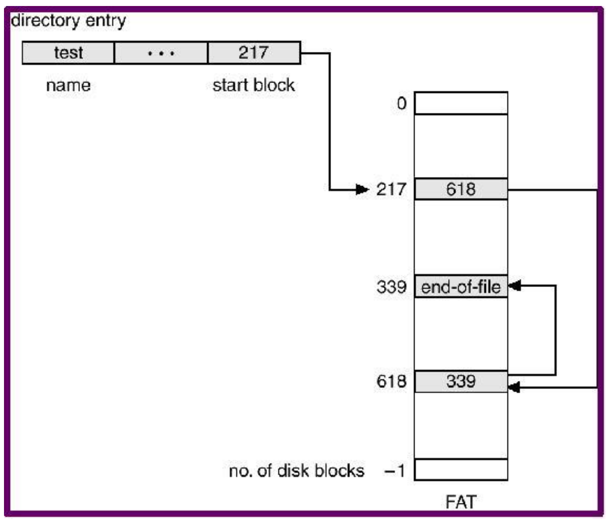
- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.
- Simple – need only starting address
- Free-space management system – no waste of space
- No random access

- File-allocation table (FAT) – disk-space allocation used by MS-DOS, Windows 9X.

Linked Allocation

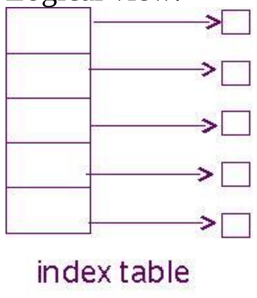


FAT



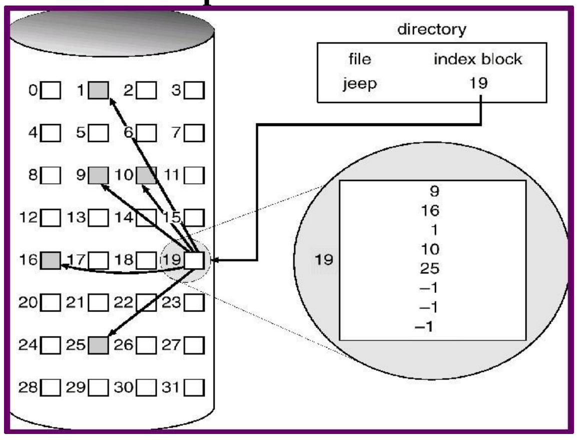
Indexed Allocation

- Brings all pointers together into the index block.
- Logical view.

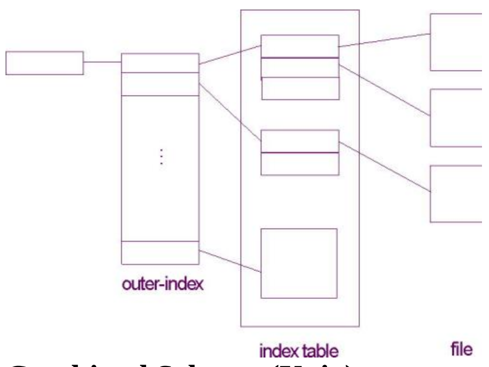


- Need index table (possibly linked tables for large files)
- Random access
- Dynamic access without external fragmentation, but have overhead of index block.

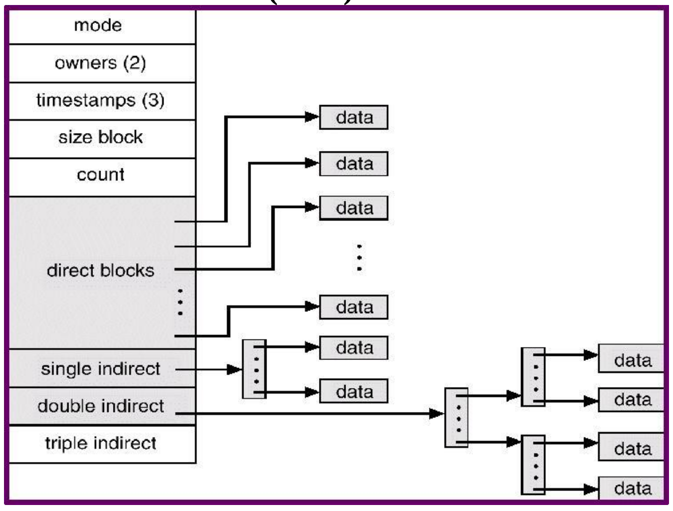
Indexed Example



Multiple Indexes

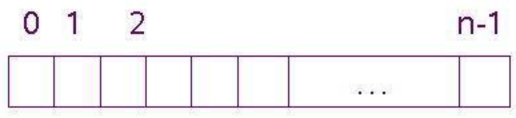


Combined Scheme (Unix)



Free Space Management

Bit vector (n blocks)

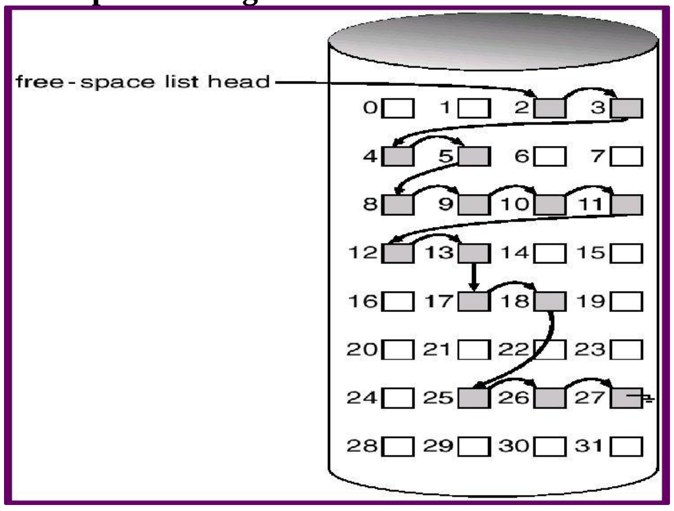


$$\text{bit}[i] = \begin{cases} 0 \Rightarrow \text{block}[i] \text{ free} \\ 1 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

Block number calculation

- (number of bits per word) *
  (number of 0-value words) +
  offset of first 1 bit

Free Space Management: Linked List



Free Space Issues

- Bit map requires extra space. Example:
  - block size = 212 bytes
  - disk size = 230 bytes (1 gigabyte)
  - n = 230/212 = 218 bits (or 32K bytes)
  - Easy to get contiguous files
- Linked list (free list)
  - Cannot get contiguous space easily
  - No waste of space
- Grouping

- Counting

Free Space Issues (cont)

Need to protect

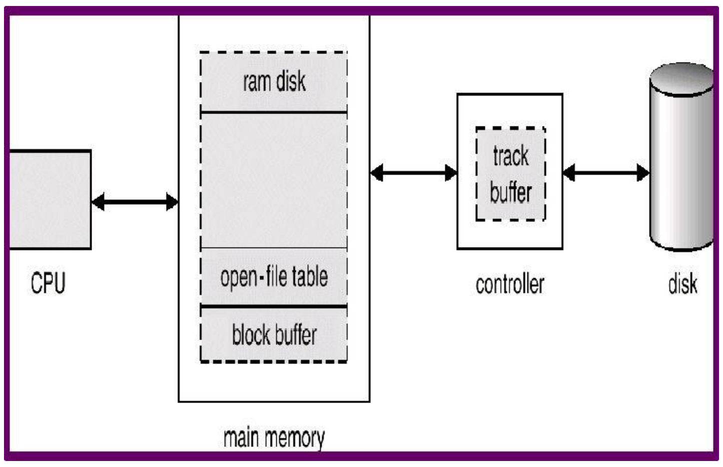
- Pointer to free list
- Bit map
    - Must be kept on disk
    - Copy in memory and disk may differ.
    - Cannot allow for block[i] to have a situation where bit[i] = 1 in memory and bit[i] = 0 on disk.
- Solution:
    - Set bit[i] = 1 in disk.
    - Allocate block[i]
    - Set bit[i] = 1 in memory

Efficiency and Performance

Efficiency and Performance Issues

- Efficiency dependent on:
    - disk allocation and directory algorithms
    - types of data kept in file's directory entry
- Performance
    - disk cache – separate section of main memory for frequently used blocks
    - free-behind and read-ahead – techniques to optimize sequential access
    - improve performance by dedicating section of memory as virtual disk, or RAM disk.

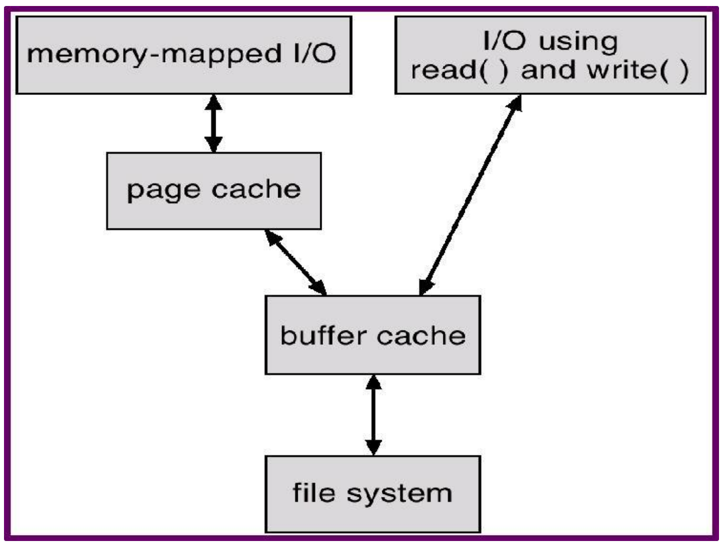
Disk Cache



Page Cache

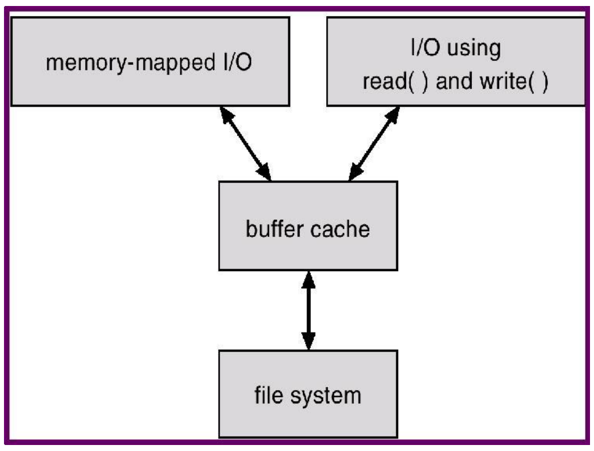
- A page cache caches pages rather than disk blocks using virtual memory techniques.
- Memory-mapped I/O uses a page cache.
- Routine I/O through the file system uses the buffer (disk) cache.
- This leads to the following figure.

I/O without unified Buffer Cache



Unified Buffer Cache

A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O.



Recovery

- Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies.
- Use system programs to back up data from disk to another storage device (floppy disk, magnetic tape).
- Recover lost file or disk by restoring data from backup.

Log Structured File Systems

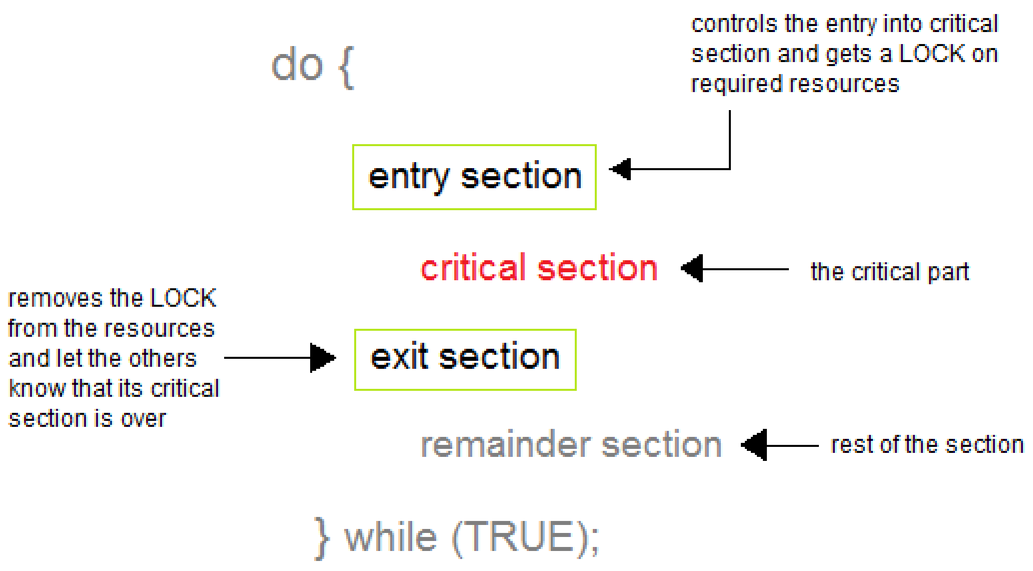
- Log structured (or journaling) file systems record each update to the file system as a transaction.
- All transactions are written to a log. A transaction is considered committed once it is written to the log. However, the file system may not yet be updated.
- The transactions in the log are asynchronously written to the file system. When the file system is modified, the transaction is removed from the log.
- If the file system crashes, all remaining transactions in the log must still be performed.

Process Synchronization

Process Synchronization means sharing system resources by processes in a such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. Maintaining data consistency demands mechanisms to ensure synchronized execution of cooperating processes.
Process Synchronization was introduced to handle problems that arose while multiple process executions. Some of the problems are discussed below.

Critical Section Problem

A Critical Section is a code segment that accesses shared variables and has to be executed as an atomic action. It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section. If any other process also wants to execute its critical section, it must wait until the first one finishes.


do {
controls the entry into critical section and gets a LOCK on required resources
entry section
critical section
the critical part
removes the LOCK from the resources and let the others know that its critical section is over
exit section
remainder section
rest of the section
} while (TRUE);

Solution to Critical Section Problem

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

1. Mutual Exclusion
Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

2. Progress
If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

3. Bounded Waiting
After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, system must grant the process permission to get into its critical section.

Synchronization Hardware
Many systems provide hardware support for critical section code. The critical section problem could be solved easily in a single-processor environment if we could disallow interrupts to occur while a shared variable or resource is being modified.
In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without pre-emption. Unfortunately, this solution is not feasible in a multiprocessor environment.
Disabling interrupt on a multiprocessor environment can be time consuming as the message is passed to all the processors.
This message transmission lag, delays entry of threads into critical section and the system efficiency decreases.

Mutex Locks
As the synchronization hardware solution is not easy to implement for everyone, a strict software approach called Mutex Locks was introduced. In this approach, in the entry section of code, a LOCK is acquired over the critical resources modified and used inside critical section, and in the exit section that LOCK is released.
As the resource is locked while a process executes its critical section hence no other process can access it.
Introduction to Semaphores
In 1965, Dijkstra proposed a new and very significant technique for managing concurrent processes by using the value of a simple integer variable to synchronize the progress of interacting processes. This integer variable is called **semaphore**. So it is basically a synchronizing tool and is accessed only through two low standard atomic operations, **wait** and **signal** designated by P(S) and V(S) respectively.
In very simple words, **semaphore** is a variable which can hold only a non-negative Integer value, shared between all the threads, with operations **wait** and **signal**, which work as follow:

```
P(S): if S ≥ 1 then S := S - 1
   else <block and enqueue the process>;

V(S): if <some process is blocked on the queue>
    then <unblock a process>
   else S := S + 1;
```

The classical definitions of **wait** and **signal** are:

- **Wait**: Decrements the value of its argument S, as soon as it would become non-negative(greater than or equal to 1).
- **Signal**: Increments the value of its argument S, as there is no more process blocked on the queue.

Properties of Semaphores

1. It's simple and always have a non-negative Integer value.
2. Works with many processes.
3. Can have many different critical sections with different semaphores.
4. Each critical section has unique access semaphores.
5. Can permit multiple processes into the critical section at once, if desirable.

Types of Semaphores
Semaphores are mainly of two types:

1. **Binary Semaphore:**

It is a special form of semaphore used for implementing mutual exclusion, hence it is often called a **Mutex**. A binary semaphore is initialized to 1 and only takes the values 0 and 1 during execution of a program.

2. **Counting Semaphores:**

These are used to implement bounded concurrency.

Example of Use
Here is a simple step wise implementation involving declaration and usage of semaphore.

```
Shared var mutex: semaphore = 1;
Process i
    begin
    .
    .
    P(mutex);
    execute CS;
    V(mutex);
    .
    .
    End;
```

Limitations of Semaphores

1. **Priority Inversion** is a big limitation of semaphores.
2. Their use is not enforced, but is by convention only.
3. With improper use, a process may block indefinitely. Such a situation is called **Deadlock**. We will be studying deadlocks in details in coming lessons.

Classical Problems of Synchronization
In this tutorial we will discuss about various classic problem of synchronization.
Semaphore can be used in other synchronization problems besides Mutual Exclusion.
Below are some of the classical problem depicting flaws of process synchronaization in systems where cooperating processes are present.
We will discuss the following three problems:

1. Bounded Buffer (Producer-Consumer) Problem
2. The Readers Writers Problem
3. Dining Philosophers Problem

Bounded Buffer Problem

- This problem is generalised in terms of the **Producer Consumer problem**, where a **finite** buffer pool is used to exchange messages between producer and consumer processes.

Because the buffer pool has a maximum size, this problem is often called the **Bounded buffer problem**.

- Solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively.

The Readers Writers Problem

- In this problem there are some processes(called **readers**) that only read the shared data, and never change it, and there are other processes(called **writers**) who may change the data in addition to reading, or instead of reading it.
- There are various type of readers-writers problem, most centred on relative priorities of readers and writers.

Dining Philosophers Problem

- The dining philosopher's problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.
- There are five philosophers sitting around a table, in which there are five chopsticks/forks kept beside them and a bowl of rice in the centre, When a philosopher wants to eat, he uses two chopsticks - one from their left and one

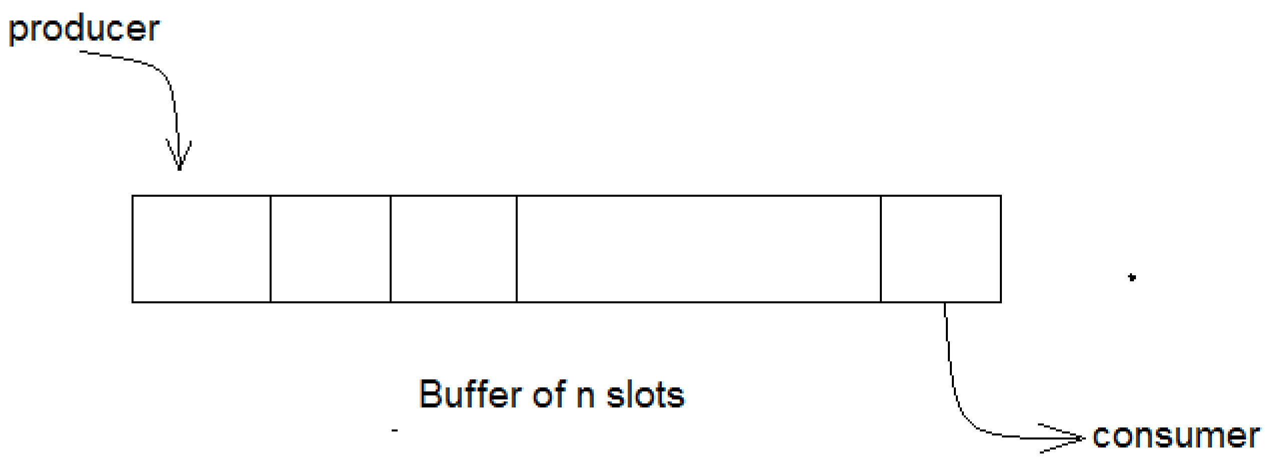
from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

Bounded Buffer Problem

Bounded buffer problem, which is also called **producer consumer problem**, is one of the classic problems of synchronization. Let's start by understanding the problem here, before moving on to the solution and program code.

What is the Problem Statement?

There is a buffer of n slots and each slot is capable of storing one unit of data. There are two processes running, namely, **producer** and **consumer**, which are operating on the buffer.



Bounded Buffer Problem

A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently. There needs to be a way to make the producer and consumer work in an independent manner.

Here's a Solution

One solution of this problem is to use semaphores. The semaphores which will be used here are:

- m, a **binary semaphore** which is used to acquire and release the lock.
- empty, a **counting semaphore** whose initial value is the number of slots in the buffer, since, initially all slots are empty.
- full, a **counting semaphore** whose initial value is 0.

At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

The Producer Operation

The pseudocode of the producer function looks like this:

```
do
{
   // wait until empty > 0 and then decrement 'empty'
   wait(empty);
   // acquire lock
   wait(mutex);

   /* perform the insert operation in a slot */

   // release lock
   signal(mutex);
   // increment 'full'
   signal(full);
}
while(TRUE)
```

- Looking at the above code for a producer, we can see that a producer first waits until there is atleast one empty slot.
- Then it decrements the **empty** semaphore because, there will now be one less empty slot, since the producer is going to insert data in one of those slots.
- Then, it acquires lock on the buffer, so that the consumer cannot access the buffer until producer completes its operation.
- After performing the insert operation, the lock is released and the value of **full** is incremented because the producer has just filled a slot in the buffer.

The Consumer Operation

The pseudocode for the consumer function looks like this:

```
do
{
   // wait until full > 0 and then decrement 'full'
   wait(full);
   // acquire the lock
   wait(mutex);

   /* perform the remove operation in a slot */

   // release the lock
   signal(mutex);
   // increment 'empty'
   signal(empty);
}
while(TRUE);
```

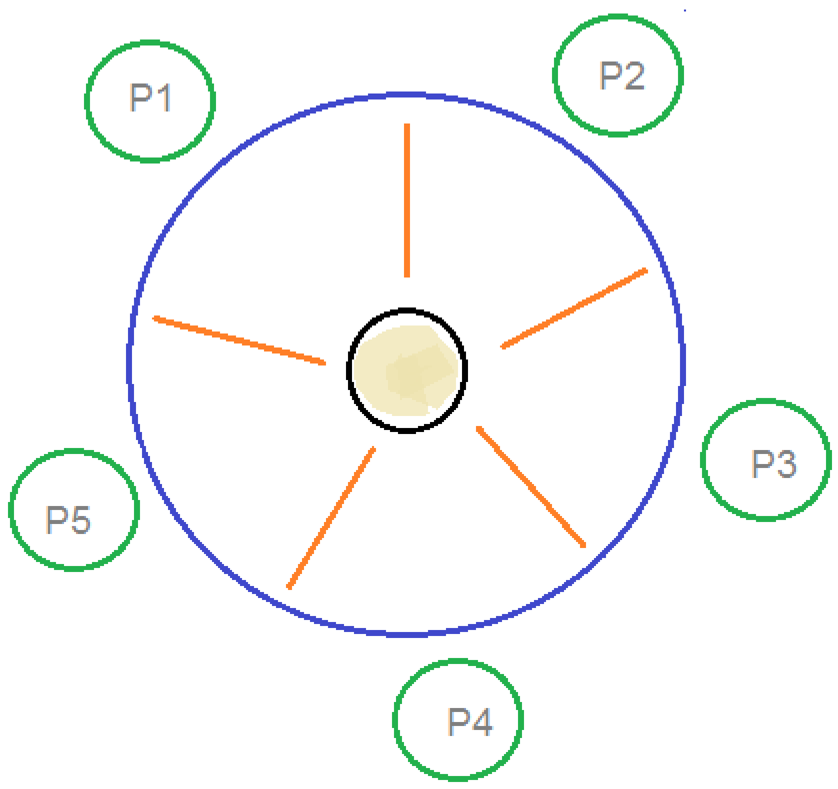
- The consumer waits until there is atleast one full slot in the buffer.
- Then it decrements the **full** semaphore because the number of occupied slots will be decreased by one, after the consumer completes its operation.
- After that, the consumer acquires lock on the buffer.
- Following that, the consumer completes the removal operation so that the data from one of the full slots is removed.
- Then, the consumer releases the lock.
- Finally, the **empty** semaphore is incremented by 1, because the consumer has just removed data from an occupied slot, thus making it empty.

Dining Philosophers Problem

The dining philosophers problem is another classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes.

What is the Problem Statement?

Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice in the middle as shown in the below figure.



Dining Philosophers Problem

At any instant, a philosopher is either eating or thinking. When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

Here's the Solution

From the problem statement, it is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time. The philosopher is in an endless cycle of thinking and eating.

An array of five semaphores, stick[5], for each of the five chopsticks.

The code for each philosopher looks like:

```
while(TRUE)
{
   wait(stick[i]);
   /*
      mod is used because if i=5, next
      chopstick is 1 (dining table is circular)
   */
   wait(stick[(i+1) % 5]);

   /* eat */
   signal(stick[i]);

   signal(stick[(i+1) % 5]);
   /* think */
}
```

When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.

But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever. The possible solutions for this are:

- A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
- Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

What is Readers Writer Problem?
Readers writer problem is another example of a classic synchronization problem. There are many variants of this problem, one of which is examined below.
The Problem Statement
There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer** is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non zero number of readers accessing the resource at that time.

The Solution
From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.
Here, we use one **mutex** m and a **semaphore** w. An integer variable read_count is used to maintain the number of readers currently accessing the resource. The variable read_count is initialized to 0. A value of 1 is given initially to m and w.
Instead of having the process to acquire lock on the shared resource, we use the mutex m to make the process to acquire and release lock whenever it is updating the read_count variable.
The code for the **writer** process looks like this:

```
while(TRUE)
{
   wait(w);

  /* perform the write operation */

  signal(w);
}
```

And, the code for the **reader** process looks like this:

```
while(TRUE)
{
   //acquire lock
   wait(m);
   read_count++;
   if(read_count == 1)
       wait(w);

   //release lock
   signal(m);

   /* perform the reading operation */

   // acquire lock
   wait(m);
   read_count--;
   if(read_count == 0)
       signal(w);

   // release lock
   signal(m);
}
```

Here is the Code uncoded(explained)

- As seen above in the code for the writer, the writer just waits on the **w** semaphore until it gets a chance to write to the resource.
- After performing the write operation, it increments **w** so that the next writer can access the resource.
- On the other hand, in the code for the reader, the lock is acquired whenever the **read_count** is updated by a process.

- When a reader wants to access the resource, first it increments the **read_count** value, then accesses the resource and then decrements the **read_count** value.
- The semaphore **w** is used by the first reader which enters the critical section and the last reader which exits the critical section.
- The reason for this is, when the first readers enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.
- Similarly, when the last reader exits the critical section, it signals the writer using the **w** semaphore because there are zero readers now and a writer can have the chance to access the resource.