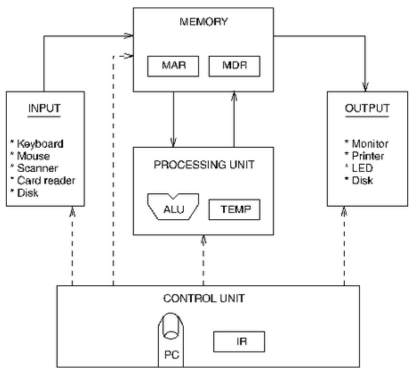
A **process** (task) is a program in execution, consisting of data, stack, registers needed
- multitasking -- allow several processes co exist ( waiting )
- process table -- stores information about each process each occurrence of a process is called an **instance** think about a timesharing system
- a computer CANNOT execute 2 processes at the same time

**Operating System**
- resource allocation
- provides runtime environment
- primary goal is convenience of operation
- secondary goal is efficiency of operation
- a **virtual machine** — it hides complicated and often heterogeneous hardware from the application and provide higher level abstraction to them
- By using CPU scheduling, and virtual memory techniques, an OS can create an illusion of multiple processes, each executing on its own memory and CPU ( virtual machine )

**The Von Neumann Architecture of Computer Systems**




- MAR -- Memory Address Register
- MDR -- Memory Data Register
- ALU -- Arithmetic Logic Unit
- CU -- Control Unit
- CPU -- Central Processing Unit = ALU + CU + Registers + some cache
- I/O -- Input/Output Unit
- MU -- Memory Unit
- Von Neumann Computer = CU + ALU + MU + I/O unit

**process states**
- **new**: the process is being created
- **ready**: process waiting to be assigned to a processor
- **running**: instructions being executed
- **blocked**: waiting for some events to happen ( e.g. I/O or reception of signal )
- **terminate**: process has finished execution

**Fork( )**
- used to duplicate a process ( 2^n )
- The call to fork in the parent return the PID ( process ID ) of the new child process. The new child process continues to execute just like the original, with the exception that in the child process the call to the fork returns 0

**Layered Systems:**
- OS → hierarchy layers
- flexible, has modularity, easier to debug, first layer can be debugged without any concern with the rest of the system

5 — operator
4 — user program
3 — i/o management
2 — operator-process communication
1 — memory management
0 — allocation of CPU for processes and multiprogramming

**process table ( PT )** saves Process Control Blocks ( PCB )
each process has a PCB with the information:
- process state
- program counter ( PC )
- CPU registers
- memory-management information
- accounting information ( e.g. time limits, process #, ... )
- I/O status information ( e.g. list of open files by the process )

**Goals for Scheduling Disciplines**
- CPU utilization -- keep CPU as busy as possible
- Fairness -- each process has fair share of CPU
- Minimize response time -- time from the submission of a request until the first response is produced
- Minimize overhead (context swaps)
- Minimize turnaround time -- the interval from the time of submission to the time of completion: waiting time to get into memory + waiting in ready queue + execution on CPU + I/O time
- Maximize throughput -- the number of processes that are completed per unit time

**Java**
- Objected-oriented, architectural, distributed, multithreaded programming language
- A Java program consists of one or more classes
- Each class ~ architectural byte code
- Runs on Java Virtual Machine ( JVM )
- JVM consists of a class loader

**Round Robin**
if quantum = 20 ms and switching time = 5 ms
% of CPU time 'wasted' = 5 / ( 20 + 5 ) = 20%

**Turnaround time** = wait + execution time
Wait time = wait

**Threads**
- process ~ program in execution + resources needed
- A thread ( light weight process ( LWP ) ) is a basic unit of CPU utilization ~ PC + registers + stack space
- a thread shares with peer threads its coded section, data section, and OS resources ( such as open files and signals ) ~ task
- traditionally heavy weight process = thread + task
- CPU switching much easier
- no memory-management related work need to be done

**Monitors**
- high-level synchronization,
- consists of procedures, shared resources, and administrative data
- only one process can be active inside a monitor, i.e. only one process can execute any of the methods at any time (mutual exclusion)
- procedures of monitor can only access data inside monitor
- local data of monitor cannot be accessed from outside
- When a task attempts to access a monitor method, it is put in the monitor's entry queue. Each monitor has one waiting queue.
- A **condition variable** is part of the monitor. Sometimes these are called *event queues* or *variable queues*. A condition variable queue can only be accessed with two monitor methods associated with this queue. These methods are typically called **wait** and **signal**. The signal method is called **notify** in Java. Condition variable queues contain processes that have executed a condition variable wait operation. There is one such queue for each condition variable.
- A task that holds the monitor lock may give it up and enter a condition variable queue by executing the corresponding wait method.
- A task that holds the monitor lock may revive a task waiting in a condition variable queue with the notify method of that queue.
- The notify method removes one task from the condition variable queue if the queue is not empty.
- Processes in the monitor queues are waiting to acquire the monitor lock.
- When a task is removed from one of the condition variable queues, it is put in the **waiting queue**.

**Weakness:**
1. one process active inside monitor => defeats concurrency purpose
2. nested monitor calls --> deadlock

Java Synchronization ~ monitor

**System Model**
- A system consists of a finite number of resources to be distributed among a number of competing processes.
- Resources can only be used by a single process at any single instance of time
- Resources can be of different types, e.g. memory space, CPU cycles, files, I/O devices
- A process must request a resource before using it, and must release the resource after using it
- A normal operation consists of a sequence of events:
  - Request: if request cannot be granted immediately, process must wait until it can acquire the resource
  - Use: operate on the resource ( e.g. printing )
  - Release: release the resource

Examples:
open file, read file, close file
allocate memory, use memory, free memory
- A set of processes is in a **deadlock** state when every process in the set is waiting for an event that can be caused by another process in the set.

**Deadlock modeling**
**Four necessary conditions for deadlock:**
- mutual exclusion -- only one process at a time can use the resource
- hold and wait -- there must exist a process that is holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes.
- no preemption -- resources cannot be preempted; a resource can be released only voluntarily by the process holding it.
- circular wait --
- if graph contains no cycles => no process is in deadlock
- if graph contains cycles => deadlock **may** exist

**Deadlock handling strategies**
- Ignore the problem altogether
- Prevention -- use a protocol to ensure that the system will never enter a deadlock state
- Detection and recovery -- allow the system to enter a deadlock state and then recover
- Dynamic avoidance -- by careful resource allocation

a) The Ostrich Algorithm
- - pretend there's no problem ( Windows, UNIX )
  undetected deadlock may result in deterioration of the system performance; eventually, the system will stop functioning and will need to be restarted manually

b) Detection and Recovery
- system monitors requests and releases resources
- check resource graph to see if cycle exists; kill a process if necessary
- or if a process has been blocked for, say one hour, kill it

c) Deadlock Prevention
- **mutual exclusion**: printer cannot be shared by 2 processes

read-only file -- sharable
A process never has to wait for a sharable resource
In general, it is not possible to prevent deadlocks by denying mutual-exclusion condition
- **Hold and wait**
  have all processes request all their resources before start execution -- inefficient, also, process don't know in advance resources needed before a process can request any additional resources, it must release temporarily all the resources that it currently holds, if successful, it can get the original resources back
  may have starvation
- **No preemption**: allows preemption
- **Circular wait**

order resource types
process requests resources in an increasing order

d) Deadlock Avoidance
In prevention strategy, we restrain how request can be made. Here, we want to develop an algorithm to avoid deadlock by making the right choice all the time
- **Banker's Algorithm for Single Resource Type**

Dijkstra's Banker's Algorithm is an approach to trying to give processes as much as is possible, while guaranteeing no deadlock.
**safe state** -- a state is safe if the system can allocate resources to each process in some order and still avoid a deadlock.
The banker's algorithm is to consider each request as it occurs, if granting the request would result in a safe state, request is granted
otherwise, it is postponed

**The benefits of using Threads:**
- Performance gains from multiprocessing hardware ( parallelism )
- Increased application throughput
- Increased application responsiveness
- Enhanced process-to-process communications
- Efficient use of system resources
- Inherent effectiveness for distributed objects
- Only one binary needed for uniprocessor and multiprocessors
- The ability to create well-structured programs
- There can be a single source for multiple platforms

**SJF can be preemptive or nonpreemptive**
- nonpreemptive -- when shortest job comes in, it has to wait until the currently-executed job has finished
- preemptive -- when shortest job comes in, currently-executed job will be suspended
- then, shortest-remaining-time-first

**Semaphores are not provided by hardware. But they have several attractive properties:**
- Machine independent.
- Simple.
- Powerful. Embody both exclusion and waiting but do NOT need to be busy waiting
- Correctness is easy to determine.
- Work with many processes.
- Can have many different critical sections with different semaphores.
- Can acquire many resources simultaneously (multiple P's).
- Can permit multiple processes into the critical section at once, if that is desirable.

**Semaphores**
- Generalization: some code sections may be accessed by a limited number of threads
- A semaphore S is an integer variable, apart from initialization is accessed through standard atomic operations:

**Named Semaphores**
The advantage of named semaphores is that they provide synchronization between unrelated process and related process as well as between threads. ( Related processes refer to parent-child processes. ) A named semaphore is created by calling following function:

**Unnamed Semaphores**
Again, according to the man pages, an unnamed semaphore is placed in a region of memory that is shared between multiple threads (a thread-shared semaphore) or processes (a process-shared semaphore). A thread-shared semaphore is placed in a region where only threads of a process share them, for example a global variable. A process-shared semaphore is placed in a region where different processes can share them, for example something like a shared memory region. An unnamed semaphore provides synchronization between threads and between related processes and are process-based semaphores.

**multiple processes**
- has states new, ready, running, blocked, terminated
- processes can create child processes
- each process operates independently of the others, has its own PC, stack pointer and address space

**multiple threads**

- like a process, a thread has states new, ready, running, blocked, terminated
- threads can create child threads
- threads are not independent of each other; threads can read or write over any other's stack

**Process Creation**

- When a process creates a new process, two possibilities exist in terms of execution
- The parent continues to execute concurrently with its children
- The parent waits until some or all of its children have terminated
- There are two possibilities in terms of the address space of the new process
- the child process is a duplicate of the parent
- the child process has a program loaded into it

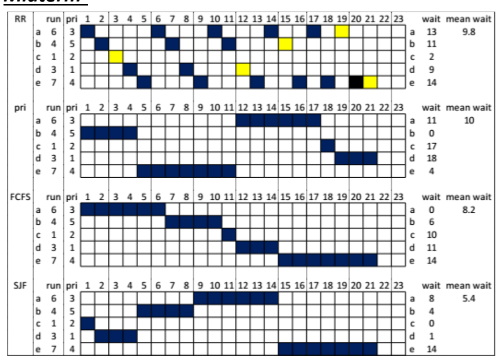
**Condition Variables**

For some problems, using semaphores could be complex.
A condition variable is a queue of threads (or processes) waiting for some sort of notifications.
Supported by POSIX and SDL; Win-32 events
A condition variable queue can only be accessed with two methods associated with its queue. These methods are typically called wait and signal. The signal method is called notify in Java.
Threads waiting for a guard to become true enter the queue.
Threads that change the guard from false to true could wake up the waiting threads.

**Multiple-process solutions**
**Lamport's Bakery Algorithm**

get a ticket number to purchase baked goods
if two processes pick the number at the same time, it does not gaurantee two processes have different #
if number( Pi ) < number ( Pj ) serve Pi first
if number ( Pi ) = number ( Pj ), then check i, j ( unique process ids ); if i < j, then serve Pi first
The notation (a,b) < (c,d) is defined as (a < c) or (a = c and b < d)

***Midterm-***




**2)Consider a readers-writers problem with readers-priority, where nr denotes the number of reader threads that have arrived to access the file and nw denotes the number of writer threads that have arrived. Use guarded commands to:**

- **present the solution of the readers operations,**
- **present the solution of the writers operations.**

a)

```
void reader()
{
  when ( nw == 0 ) [
   nr++;
  ]
  //read
  [nr--;]
}
```

b)

```
void writer()
{
  when ( (nr == 0) && (nw == 0) )[
   nw++;
  ]
  //write
  [nw--;]
}
```

**3) a) In general, a semaphore has two functions, down() which decreases the semaphore value and up(), which increases the semaphore value. What is special about these functions? Give an alternate name that people commonly used for each of the two functions.**

**b) and c) Two threads, T1() and T2() are accessing the same critical section ( CS ). They have to wait on two semaphores S1, and S2 before they can enter the CS. Consider the following piece of code for T1 and T2:**

```
T1:
        down ( S1 );
        down ( S2 );
        CS();
        up ( S2 );
        up ( S1 );
T2:
        down ( S2 );
        down ( S1 );
        CS();
        up ( S1 );
        up ( S2 );
```

a) A counting semaphore is a semaphore with an integer value that can be larger than 1. A semaphore is an integer variable that, apart from initialization, is accessed only through two standard operations: down() and up(), which differ from usual functions in the way that they are atomic. That is, they must be executed indivisibly.
A semaphore is a tool used for synchronization or to achieve mutual exclusion between processes or threads when accessing a critical section.
Alternate names:

down() ~ wait(), P()
up() ~ signal, V(), notify()

**b) Does the code achieve the purpose of mutual exclusion? Why?**
Yes, it is because a thread can enter the critical section only if it has acquired the two semaphores ( "keys" ). If one thread holds a semaphore, the other cannot have it.

**c) Is deadlock possible for T1 and T2? Why? If your answer is yes, modify the code so that the two threads are deadlock free. If your answer is no, prove your claim.**
Yes. Because threads run concurrently, it could happen that T1 has executed down ( S1 ) and T2 has executed down ( S2 ) . So now T1 is waiting for T2 to release ( UP ) S2 and T2 is waiting for T1 to release S1 and thus they are in deadlock.
To prevent deadlock, we can simply require both T1 and T2 to 'lock' ( down ) semphores in the same order S1, S2 so that whoever has first locked S1 can proceed to lock S2 and the other thread has to wait on S1. The following is the modified code that is deadlock free:

```
T1:
        down ( S1 );
        down ( S2 );
        CS();
        up ( S2 );



        up ( S1 );
T2:
        down ( S1 );
        down ( S2 );
        CS();
        up ( S2 );
        up ( S1 );
```

- Priority inversion may occur in real-time scheduling — **true**
- a thread shares with its peers the data section — **true**
- a process shares with its peers the data section — **false**
- each processor in symmetric multiprocessing performs self-scheduling — **true**
- shortest job first scheduling is optimal — **true**
- A thread can be in only one semaphore's waiting queue at a time — **true**
- Nonpreemptible resources must be hardware — **false**
- Processes may deadlock as a result of contending for the processor — **false**
- An unsafe state is a deadlocked state — **false**
- Two processes, A and B may deadlock if both of them request three records, 1, 2, and 3 in a database in the order 2, 1, 3 — **false**

## Class Exercises

1. Operating systems manage only hardware. True or false?
   - False
2. What is the primary goal of an OS?

- resource allocation
- provides run-time environment
- primary goal is convenience of operation
- secondary goal is efficiency of operation
- a virtual machine -- it hides complicated and often heterogeneous hardware from the application and provide higher level abstraction to them

3. What limited the size and capabilities of programs in the 1950s?
   - Memory
4. What is the difference between a purely layered architecture and a microkernel architecture?
   - Not limited to adjacent layer ..........!!!
5. How do microkernels promote portability?
   - Different models OS is divided on models
6. What is the output of the following?

```
int main()
{
   int i;

   for ( i = 0; i < 3; ++i ) {
     fork();
     cout << i << endl;
   }
   return 0;
}
```

## Class Exercises

1. What is a semaphore?
   - A semaphore S is an integer variable, apart from initialization is accessed through standard atomic operations
2. A thread can be in only one semaphore's waiting queue at a time. True or false?
   - True. When the semaphore in the queue it goes to sleep and need to be waking up
3. What is a major benefit of implementing semaphores in kernel?
   - It can put it in blocked and do other instead of waiting and sleeping
4. Consider a semaphore that allows the thread with the highest priority to proceed when UP() is called. What potential problem can this cause?
   - If there is someone with low priority and one with high priority. Then the low don't have a chance. "Starvation"
5. What could potentially happen if a thread called UP() without having called the down() operation?
   - It will never get to 0.

## Class Exercises

1. Processes do not deadlock as a result of contending for the processor. True or false?
   - True; Processes are preemptive

2. Nonpreemptible resources must be hardware. True or false?
   - False; it is not always hardware

3. Compare and contrast deadlock prevention and deadlock avoidance.
**Deadlock Prevention:** a set of methods for ensuring that at least one of the necessary conditions for deadlock cannot hold; Prevention by constraining how requests for resources can be made in the system and how they are handled (system design).
**Deadlock Avoidance:** The system dynamically considers every request and decides whether it is safe to grant it at this point to use during its lifetime. The system requires additional apriori information regarding the overall potential use of each resource for each process.
Allows more concurrency.
Similar to the difference between a traffic light and a police officer directing traffic; Preventing deadlocks by constraining how requests for resources can be made in the system and how they are handled (system design).

4. An unsafe state is a deadlocked state. True or false?
   - False

5. Two processes, A and B, each needs three records, 1, 2, and 3 in a database. Is deadlock possible, if both processes request the records in the order 1, 2, 3. How about if they both request the records in the order 2, 1, 3? How about A requests in order 1, 2, 3 and B requests in order 3, 2, 1?

A : 1, 2, 3
B : 1, 2, 3
No possible deadlock because the semaphores in the same order.

A : 2, 1, 3
B : 2, 1, 3
No possible deadlock because the semaphores in the same order.

A : 1, 2, 3
B : 3, 2, 1
Possible deadlock may occur. Threads run concurrently, it could happen that A executed 1 and 2 and B executed 3 then both will be locked waiting for the other thread to release the key. To prevent this both have to be in the same order.

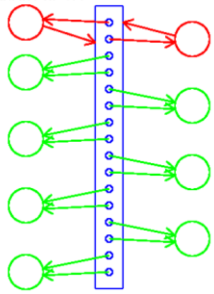
Ex. 1,2,3,1,4,2,1,5,6,2,1,3,7,6,3,2,1,2,3,7
Page fault: starts blank and everytime you load a page into a frame it's considered a page fault. Only exclusion is doubles ex. 7,7 [1 frame], 12,13 [2 frame]
LRU [Least Recently Used]: look back untill you know which 1 frame you are replacing
First In 1st out: Start replacing @ the top of your stack and keep moving down until you reach the end and start over.
Optimal Replacement: look ahead to see which n-1 frames you are going to keep.

2. ( 50 points )
a) A computer system has sixteen drives, with 9 processes competing for t[illegible] need a maximum of two drives. Is the system deadlock free? Give your rea[illegible] with the help of **resource graph diagrams**.

**Solutions**:

Yes, the system is deadlock **free**. This is because in the worst case, all 9 processes request two drives. As we have 16 drives, some of the processes can get two drives to finish their tasks and return the the drives. Then some of the remaining processes can get two drives, finish and return and so on. So eventually, all processes can finish and return all drives.

The figure on the right illustrates the worst case situation. The processes colored in red are competing for a drive but when any green process has finished using and returned the two drives, they can be allocated to the red processes.



b) In general, a semaphore has two functions, **down()** which decreases the semaphore value and **up()**, which increases the semaphore value.

1. There is a special requirement for these functions in order that a semaphore works properly. What is it?

   **Solutions**:

   The functions must be **atomic** (indivisible).

2. Two threads, **T1()** and **T2()** are accessing the same critical sections ( CS ). They share three semaphores S1, S2, and S3. Consider the following piece of code for T1 and T2:

| | | |
|---|---|---|
| T1:<br>down ( S1 );<br>down ( S3 );<br>down ( S2 )<br>CS();<br>up ( S2 );<br>up ( S3 );<br>up ( S1 ); | | T2:<br>down ( S3 );<br>down ( S2 );<br>CS();<br>up ( S2 );<br>up ( S3 ); |

(i) Does the code achieve the purpose of mutual exclusion? Why?
(ii) Is deadlock possible for T1 and T2? Why? If your answer is yes, modify the code so





that the two threads are deadlock free. If your answer is no, explain or prove your claim.

**Solutions**:

- (i) Yes, it is because a thread can enter the critical section only if it has acquired the two or three semaphores ( "keys" ). If one thread holds a semaphore, the other cannot have it. So only one thread can enter the critical section.
- (ii) No, deadlock is not possible. This is because both threads request the semaphores in the same order (S3 then S2). Thus there can never be a circular wait and consequently no deadlock is possible.

**The low cost of main memory coupled with the increase in memory capacity in most systems has obviated the need for memory management strategies. True or false?**

- False. Despite the low cost and high capacity of main memory, there continue to be environments that consume all available memory. Also, memory management strategies should be applied to cache, which consists of more expensive, low-capacity memory. In either case, when memory becomes full, as system must implement memory management strategies to obtain the best possible use of memory.

**Why is first-fit an appealing strategy?**

- Because it does not require that the free memory list be sorted, so it incurs little overhead. However, it may operate slowly if the holes that are too small to hold the incoming job are at the front of the free memory.

**The number of faults for a particular process always decreases as the number of page frames allocated to a process increases. True or false?**

- False. This is indeed the normal behavior, but if the algorithm is subject ot Belady's Anomaly, the number of faults might decrease.

**Does looping through an array exhibit both spatial and temporal locality? Why?**

- Yes. It exhibits spatial locality because the elements of an array are contiguous in virtual memory. It exhibits temporal locality because the elements are generally much smaller than a page. Therefore references to two consecutive elements usually result in the same page being referenced twice within a short period of time.

**LRU is designed to benefit processes that exhibit spatial locality. True or false.**

- False. LRU benefits processes that exhibit temporal locality.

**Suppose a block mapping system represents a virtual address v = ( b, d ) using 32 bits.**
**If d is n bits, how many blocks does the virtual address space contain?**
**Discuss how setting n = 6, n = 12, and n = 24 affects memory fragmentation and the overhead incurred by mapping information.**

- 2^(32-n) blocks. If n = 6, block size would be small and there is not much internal fragmentation,
- but the number of blocks would be so large as to make implementation infeasible.
- If n = 24, quite a lot of internal fragmentation. But block mapping table would not consume too much memory
- n = 12, balance, appropriate size. page size = 2^12 = 4096 has been a popular choice.

## Class Exercises: Storage Systems

**Explain the benefits of using a contiguous file allocation scheme.**

- Locating file data is straightforward. Also, files can be accessed quickly, because the storage does not need to perform lengthy seeks after it locates the first block.

**Explain the disadvantages of using a contiguous file allocation scheme.**

- They can lead to a significant external fragmentation and poor performance when a file grows tool large to be stored contiguously at the current location and must be moved.

**How do user classes reduce the storage overhead by access control information?**

- User classes enable the owner to grant permissions to a group of users using one entity.

**Discuss the advantages and disadvantages of storing access control data as part of the file control block.**

- Advantage: There is little overhead if the user is granted access to a file, because the file system needs the file control block before opening the file anyway. Disadvantage: If access is denied, the system will have wastefully performed several lengthy seeks to access the file control block for a file that the user cannot open.

**Why is a single file server impractical for environment such as a large organizations?**

- Ans. If the server fails, the whole file system becomes unavailable. A central server can also become a bottleneck.

**Why are physical safeguards insufficient for preventing loss of data in the event of disaster?**

- Physical safeguards prevent access to data but do not prevent loss of data due to natural disasters, such as fires and earthquakes, or to hardware and power failure.

**Paging**

- old days, program split into pieces -- overlay, overlay0, overlay1 swapped by OS but program split was done by programmers
- modern computers have special hardware called a memory management unit (MMU). Whenever the CPU wants to access memory (whether it is to load an instruction or load or store data), it sends the desired memory address to the MMU, which translates it to another address before passing it on the the memory unit.
- address generated by the CPU -- virtual address,
- address translated to by the MMU -- physical address

**Contiguous Allocation**

- reduce seek time for contiguous access
- access is easy
- supports both sequential and direct access
- dynamic allocation problem ( may use first fit, best fit, worst fit .. )
- External fragmentation — need compaction : expensive
- Widely used in CD-ROMs, DVDs where final file sizes are known in advance and won't change

**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 space
- no external fragmentation
- easy to create a file ( no allocation problem )
- only better for sequential access
- overhead — pointers require space
- reliability — damaged pointers

## Indexed Allocation

- direct access
- no external fragmentation
- easy to create a file ( no allocation )
- support both sequential and direct access
- overhead: space for index blocks

**Authentication vs Authorization**

Protection Mechanisms:

- User identification (**authentication**): make sure we know who is doing what.
- **Authorization** determination: must figure out what the user is and is not allowed to do. Need a simple database for this.
- Access enforcement: must make sure there are no loopholes in the system.
- Even the slightest flaw in any of these areas may ruin the whole protection mechanism.

**Authentication**

- User identification is most often done with passwords, a relatively weak form of protection.
  - A password is a secret piece of information used to establish the identity of a user.
  - Passwords should not be stored in a readable form. One-way transformations should be used.
  - Passwords should be relatively long and obscure.
- Another form of identification: badge or key.
  - Does not have to be kept secret.
  - Should not be able to be forged or copied.
  - Can be stolen, but the owner should know if it is.
- Biometrics
- Key paradox: key must be cheap to make, hard to duplicate. This means there must be some trick (i.e. secret) that has to be protected.
- Once identification is complete, the system must be sure to protect the identity since other parts of the system will rely on it.

**Authorization**

- Must indicate who is allowed to do what with what. Draw the general form as an access matrix with one row per user, one column per file. Each entry indicates the privileges of that user on that object. There are two general ways of storing this information: access lists and capabilities.
- **Access Lists**: with each file, indicate which users are allowed to perform which operations.
  - In the most general form, each file has a list of pairs.
  - It would be tedious to have a separate listing for every user, so they are usually grouped into classes. For example, in Unix there are three classes: self, group, anybody else (nine bits per file).
  - Access lists are simple, and are used in almost all file systems.
  - *Example:*
  - File A:
    - { Alice, (read, write) }
    - { Bob, (read, write) }
    - { Chris, (read) }
  - File B:
    - { Alice, (read, write) }
    - { Bob, (read, write) }
    - { David, (read) }
  - Printer:
    - { Alice, (print) }
    - { David, (print) }
    - { Edward, (print, delete) }
- **Capabilities**: with each user, indicate which files may be accessed, and in what ways.
  - A capability is a pointer or token that grants privileges to a subject that possesses it.
  - Store a list of pairs with each user. This is called a capability list. ( ~ tickets in a fair )
  - Typically, capability systems use a different naming arrangement, where the capabilities are the only names of objects. You cannot even name objects not referred to in your capability list.
  - In access-list systems, the default is usually for everyone to be able to access a file. In capability-based systems, the default is for no-one to be able to access a file unless they have been given a capability. There is no way of even naming an object without a capability.
  - Capabilities are usually used in systems that need to be very secure. However, capabilities can make it difficult to share information: nobody can get access to your stuff unless you explicitly give it to them.
- **Lock-key Mechanism**
  - compromise between access lists and capability lists
  - each object has a list of unique bit patterns, called locks
  - each domain ( user ) has a list of unique bit patterns, called keys
  - a process executing in a domain can access an object only if that domain has a key that matches one of the locks of the object
  - As was the case with capability lists, the list of keys for a domain must be managed by the OS on behalf of the domain. Users are not allowed to examine or modify the list of keys ( or locks ) directly

Consider a file currently consisting of 200 blocks. Assume that the FCB ( and the index block, in the case of indexed allocation ) is already in memory. Calculate how many disk I/O operations are required for contiguous, linked, and indexed ( single-level ) allocation strategies if, for one block, the following conditions hold. In the contiguous-allocation case, assume that there is no room to grow in the beginning, but room to grow in the end. Assume that the block information to be added is stored in memory.

a. The block is added at the the beginning.
b. The block is added in the middle.
c. The block is added at the end.
d. The block is removed from the beginning.
e. The block is removed from the middle.
f. The block is removed from the end.
Answer:

a. **Contiguous: 401**
Because each read and write block has to moved over once, so 200 x 2 + the one extra block.

**Linked: 1**
Just write the new block making it point to the next block. Update the first block pointer in memory.

**Indexed: 1**
Just write the new block and update the index in memory.

b. **Contiguous: 201**
100 blocks (the second half) must be shifted over one block and the new block must be written. As before, the shift takes one read operation and one write operation.

**Linked: 102**
I must read 100 blocks to find the middle. Then I must write my new block somewhere with the next block pointing to the block after the 100th block. Then I must write the 100th block to point to this new block.

**Indexed: 1**
Just write the new block and update the index in memory.

**c. Contiguous: 1**
Just write the new block.

**Linked: 3**
(assuming that in order to modify the a block's next block pointer, I must first read the whole block in, then write the whole block out just with that pointer changed). First I read the last block as given by the last block pointer, then I write my new block, then I write the last block back modifying its next block pointer to point to my new block, then I update my last block pointer in memory.

**Indexed: 1**
Just write the new block and update the index in memory.

**d. Contiguous: 0**
Just point the first block pointer to the second block.

**Linked: 1**
Read in the first block to get the second block's pointer and then set the first block pointer to point to the second block.

**Indexed: 0**
Just remove the block from the index in memory.

**e. Contiguous: 198**
Assuming that we are removing the 101st block, we will have to move 99 blocks one block over this takes a read and a write operation each.

**Linked: 102**
It takes 101 reads to find the pointer to the 102nd block, then we need to update the 100th block's next block pointer with this value.

**Indexed: 0**
Just remove the block from the index in memory.

**f. Contiguous: 0**
Just update the length information stored in memory.

**Linked: 199 or 200**
We need to update the last block pointer with the second to last block and the only way to get the second to last block is to read the preceding 199 blocks. Then we probably want to mark the 199th block (the new last block) as having a null pointer and this would give the 200th operation. We save the new last block in our last block pointer in memory.

**Indexed: 0**
Just remove the block from the index in memory.

**I/O Systems:**
Interrupts
- non-maskable interrupts ( NMI ) — cannot be masked off, reserved for serious errors
- maskable interrupt — can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted

Direct Memory Access (DMA)
- Overview: DMA is an operational transfer mode which allows data transfer within memory or between memory and I/O device without processor's intervention. A special DMA controller manages that data transfer.

Disk Scheduling:
- requests are queued
- FCFS ( first-come-first-served ) scheduling
- SSTF ( Shortest-seek-time-first ) scheduling
- SCAN scheduling
- the disk arm starts at one end of the disk, and moves toward the other end, servicing requests as it reaches each cylinder; when it reaches the other end, the direction of head movement is reversed,
- e.g. queue = 98, 183, 37, 122, 14, 124, 65, 67
- current head position at 53, moving towards track 0
- service sequence: 37, 14, 65, 67, 98, 122, 124, 183
- C-SCAN ( circular scan ) scheduling
- when it reaches the other end, it immediately return to the beginning
- service sequence of above example: 65, 67, 98, 122, 124, 183, 14, 37

RAIDS:
- RAID 0: non-redundant block-level striping ( improves speed but no fault tolerance )
- RAID 1: mirrored disks ( fault tolerance but no improvement in speed )
- RAID 2: memory-style error correcting ( ECC ), not used in practice because level 3 is better
- RAID 3: bit-level stripping with parity ( both speed and fault tolerance )
- RAID 4: block-level striping with parity ( both speed and fault tolerance )