Processes and Threads:

Process: A running program. Can run (pseudo) concurrently with other processes.
Program: Set of instructions for the process.
Processor: The object running the process.
Input: A set of arguments required for a process to run properly.
Output: The result of the process.
Program counter: Pointer to the currently executing statement of the program.

Multiprogramming: The act of letting multiple processes run (pseudo) concurrently, taking turns where necessary.
Uniprogramming: The act of running one and only one program at a time.

Methods to create a process:

- Initializing the system to allow the process to run (Booting a computer);
- Properly executing a process creating system call (Forking);
- User request for process creation (Opening an internet browser);
- Initiating a batch job (Any batch system input).

Process Control Block: How a process is represented:

- Identification number;
- State vector {CPU, Process ID, Memory used, required open files, other required resources};
- Status {Type of process, subroutine list};
- Creation Tree {What is the parent, which are the children}
- Priority

Fork: System call in UNIX which creates a new process. It creates an identical copy of the calling process, both processes then run concurrently, competing for resources. The fork() operation is where the process splits, but the parent process will return some non-zero integer while the child returns zero, allowing for the programs to differ where necessary.

A process can be in one of many **states:**

- New: A newly created process is still acquiring resources, it is not yet ready to run;
- Ready: Default state after creation. The states all other states return to in order to indicate that the process may be run again;
- Active: The currently running process. Only one process may be in this state at any given time, once a ready process is *dispatched*. The active process is the one being executed by the CPU;
- Blocked: An active process requires some event or resource which is not available. Once that event or resource becomes available, a blocked process switches to a ready state;
- Stopped: A process has run sufficiently long enough and must now allow another process to be active, or it is suspended by the operator. Once it is ready to resume, a stopped process switches to a ready state;
- Exiting: An active process terminates or is killed, a stopped process is killed, or a blocked process encounters an error. The process is removed from the pool of executable processes.

Dispatcher: Responsible for stopping running processes, saving their states, and loading the states of other processes in order to allow them to run. During the context switch, the dispatcher saves all items that could be damaged by next process (Program counter, Registers, File Access Pointers, Memory [but only when required, as this could be time-consuming]).

File Descriptor: External devices pointed to by a process. By default, handle #0 is the keyboard (standard in), handle #1 is the screen (standard out) and handle #2 is the error handler (standard error).

Address Space: All addresses that could be generated by the process. Part of this address space is taken by the kernel in order to allow inter-process communication.

Command Piping: Connecting the output of one process to the input of another. This transfer is done through kernel space by overwriting file descriptors.

Parallelism: A system can perform more than one task simultaneously.

Concurrency: A system can support more than one task progressing.

Amdhal's Law: Performance improvement obtained by applying an enhancement on an application is limited by the fraction of the time said enhancement can be applied.

Speedup: The obtainable acceleration of a program by adding N cores to the application $SPEEDUP \leq \frac{1}{S+\frac{1-S}{N}}$.

Threads: Lightweight processes which allow not-so-tightly coupled activities to execute concurrently

- Processes
    - Heavy
    - Every process needs distinct address space
    - Unique data structures
- Threads
    - Live within processes
    - All threads of 1 process share memory and resources
    - Less fault tolerant
    - Can introduce synchronization issues
- User Threads: Managed by user-level libraries. Many of these are mapped to a single kernel thread, one block causes them all to block, may not run in parallel.
- Kernel Threads: Managed by kernel, as seen in all general OS's.

Pthreads: An API for a thread specification. Each one has it sown identity, returned to parent with pthread_create() and returned to self using pthread_self(). Use pthread_join() to merge a terminated thread with another in order to avoid "zombie" threads. Use pthread_detach() to remove a thread from a set of merged threads. Cancelling a thread can be done asynchronously (immediate) or deferred (periodic check until termination is appropriate), may only occur upon reaching a cancellation point, where a cleanup handler will be invoked.

<u>Synchronization:</u>

Competition between processes: Processes cannot affect another's execution, but they can compete for resources. Must be able to stop and re-start without side effects.

Cooperation between processes: Processes that work together, like by exchanging messages or sharing objects, may affect the other's execution. Specific execution may be irreproducible. Must be carful about execution order.

Race Conditions: Two or more processes run and the final result is dependent on execution order. Avoid these with mutual exclusion, allowing only one process to read/write to shared data at any given time.

Critical Section: The part of the program where shared data is accessed. For effective cooperation, no two programs can simultaneously be in their critical sections, no assumptions must be made about their execution speeds, no processes outside their critical sections may block other processes, and none should have to wait indefinitely to enter a critical section. When a critical section begins execution, it must run to completion without being interrupted.

Spin lock: An inefficient method for solving critical sections. For 2 programs, assign a global variable which decides when which program can enter the section. Problem: continuous checking causes busy waiting, which wastes CPU time.

Dekker's Algorithm: A method to provide mutual exclusion. Turn provides arbitration and array of flags expresses interest to enter critical section

Process 0	Process 1

…
flag[0] = true
while(flag[1]){
  if(turn==1){
  flag[0] = false
  while(turn==1){/*wait*/}
  flag[0] = true
  }
}
/*Critical section*/
turn = 1
flag[0] = false
…
	…
	flag[1] = true
	while(flag[0]){
	  if(turn==0){
	  flag[1] = false
	  while(turn==0){/*wait*/}
	  flag[1] = true
	  }
	}
	/*Critical section*/
	turn = 0
	flag[1] = false
	…

Peterson's Algorithm: Same idea as Dekker's Algorithm

Process 0	Process 1
… flag[0] = true turn = 1 while(flag[1]&&turn==1){/*wait*/} /*Critical Section*/ flag[0] = false …	… flag[1] = true turn = 0 while(flag[0]&&turn==0){/*wait*/} /*Critical Section*/ flag[1] = false …

Semaphore: Allows cooperation between processes by sending messages. Use signal(s) to send a message and wait(s) to receive a message.

- Initialized to nonnegative integer;
- Wait(s) decrements value, if it becomes negative process executing wait(s) is blocked. Blocked processes are kept in a queue;
- Signal(s) increments value, if not positive, a blocked process is unblocked;
- These two operations are atomic, cannot be interrupted;

Monitor: An abstraction that combines and hides shared data, data operations, and synchronization. Only on process at a time may be active within a monitor. They use **condition variables** to provide synchronization use its own wait (suspends executing process until another process does a signal) and signal (resumes one [random] waiting process, which will be the next to begin executing after the current executing process either exits or waits). So many processes can be in the monitor, but only one may run at a time, all others will be waiting. A good strategy is to have signal be the last operation in a process.

<u>Deadlock:</u>

Deadlock: Permanent blocking of a set of processes while competing for resources.

Resource Classification:

- *Reusable:* Is not depleted after being used. Are released for re-use. CPU, files, memory…
- *Consumable:* Are created and destroyed, not re-used. Interrupts, messages, signals…
- *Preemptable:* Can be taken away from the process using it without ill effects. Requires a save/restore function. Memory, CPU…
- *Non-preemptable:* Cannot be taken away without causing failure. Printer, floppy disk…

Conditions for a deadlock to occur:

- *Mutual Exclusion:* A process requires exclusive control of its resources.
- *Hold and Wait:* A process will hold on to its currently acquired resources while waiting for others.
- *No pre-emption:* Until a process is finished using a resource, it will not release it.
- *Circular wait:* Each process holds a resource requested by another.
- All of these must be satisfied.

Avoiding Deadlocks: Prevent a condition from occurring, anticipate resource allocation, provide checkpoints so a process may back out if needed. The most common strategies are:

- *Ignorance:* Pretend there is no problem;
- *Prevention:* Exclude deadlock in system design, but imposes restrictions on processes. Can impose restrictions on how a process acquires resources, and provide methods for releasing if needed;
- *Avoidance:* Make a dynamic decision, seeing if a request will lead to a deadlock before making said decision. Requires knowledge of future requests;

- *Detection:* Take some action to recover once the deadlock has occurred. Can check for occurrence of deadlock after every resource request. Leads to early detection, but frequent checks consume a lot of processor time;

Safe State: When all processes in the system can have their resources satisfied by current available resources. Ensuring that an unsafe state is never encountered is a deadlock avoidance method.

Banker's Algorithm: Deadlock avoidance method. Calculation to determine if a request can be completed without leading to an unsafe state.

Deadlock Recovery:

- *Pre-emption:* Remove a resource from its owner and give it to another process;
- *Rollback:* Arrange for periodic checkpoints and undo some steps;
- *Termination:* Kill a process in the cycle
- These are not always possible!

<u>Secondary Storage:</u>

Secondary Storage: Non-volatile storage for data, programs, memory. Managed by OS, which maintains it persistently. Handles concurrent access.

File: A named collection of information, typically a sequence of bytes. Seen with either the *logical view* by the user or the *physical view* as the machine reads on secondary storage. Non-volatile, long lasting, can be moved around and easily copied. Files have attributes, like Name, Type, Location, Organization, Access Permissions, Time & Date, Size, …

Directory: A symbol table that can be searched for information, which gets implemented as a file. Provides a space for files to aggregate. A *directory entry* contains non-volatile information about a file, which can easily be added or removed.

Name Space: Set of symbols which organizes objects, so they may be referred to by name.

File System: Keeps track of files, provides I/O support so the user may easily manipulate the system, management of secondary storage, provides protection of information.

- *Master Boot Record:* Sector 0 of the file system, which boots the computer;
- *Partition Table:* Starting and ending address of each partition;
- *Superblock:* Key parameters (number of blocks in file system, file size, length of inode table, etc.) read into memory;
- *Inodes:* Array of data structures, one per file, listing all information about the file (except name).

File Operations:

- *Create:* Basic file attributes are set, file initialized with no data;
- *Delete:* File storage is released to free pool, as file and data is no longer needed;
- *Open:* Fetch file attributes and disk addresses, place into main memory;
- *Close:* Free internal table space, forces a writing of files last block;

- *Read:* Reads data at current position;
- *Seek:* Repositions file pointer, after seeking, data can be read or written where specified;
- *Write:* Writes data at current position, may increase file size;

File Access Methods:

- *Sequential:* File stored in order, one data block after another. Allows simple and easy access with few seeks;
- *Direct:* File stored in any order;
- *Indexed:* File stored in any order, but accessed using particular keys like a hash table. Allows dynamic growth of files, but requires lots of seeking Can be enhanced with a *File Allocation Table*, which keeps track of the blocks where the file resides;

File Allocation Problem: Allocate files to minimize trade-off, large blocks mean wasted space, small blocks mean many seeks.

Free Space Management: Need to re-use free space, and keep track of this free memory

- *Bitmap:* Store a map of bits at a well-known address, with each bit mapped to a block to show if it is used;
- *Linked list:* List of a set of free block lists. First pointer at well-known location, have a pointer to another free block, repeat;
- *Indexed:* Maintain index pointing to all free blocks;

Disk Caching: Read/Write an entire track with each disk access.

Disk Scheduling: Maximize throughput, minimize time spent seeking data.

- *Random:* Select from available pool, worst performer;
- *First Come First Serve:* Honour requests in order they are requested. Works well for few processes, approaches random as competing resources increase;
- *Priority:* Access order based on execution priority, designed to meet job throughput criteria;
- *Last in First Out:* Service most recent request. Useful for processing sequential files, but provides danger of starvation;
- *Shortest Service Time First:* Select item with shortest seeking time. Random tie-breaker if needed. Generally better than FIFO;
- *SCAN back and forth:* Head moves in one direction until the last track is reached, then reserves direction. Method is called LOOP if no more requests are present. Biased against most recent used area;
- *C-SCAN:* Scans in one direction, quickly returns to beginning when end is reached. Maximum delay for new arrivals is reduced;
- *LOOK:* Look for a request before moving in a direction. When no more requests are in that direction, reverse;
- *C-Look:* Scan in one direction until no more requests are in that direction, then quickly return to beginning;

CPU Scheduling

Scheduling: A set of policies that control the sequence of jobs executed by a computer. Used to maximize CPU utilization, minimize response time, and maximize throughput.

Allocation: A set of policies which determine which jobs get to utilize which resources.

Scheduling Approaches:

- *Non-preemptive:* Process will run until it blocks for some event, or demands a resource. The scheduler will not interrupt it (Good for background jobs);
- *Preemptive:* Scheduler may interrupt process, force it to release CPU. Occurs at clock interrupt (Good for foreground jobs);

Short-Term Scheduler: Selects jobs among the ready queue for execution, CPU will be allocated to one of them.

Evaluation of a scheduler:

- *CPU Utilization:* $\frac{capacity\ used}{total\ capactiy}$, on a scale from 100% to 0%;
- *Efficiency:* $\frac{Useful\ work}{Total\ work}$;
- *Throughput:* The number of jobs completed per unit time;
- *Turnaround Time:* Difference in time between job arrival and completion;
- *Waiting Time:* Total time a job spent in ready queue;
- *Service Time:* Total time a job spent in the active queue;
- *Response Time:* Time between first response and arrival of a job;
- Of the above criteria, order them in desired performance importance, and design with careful balance, as one criterion is not necessarily independent of another;

Scheduling Techniques:

- *First Come First Serve:* Non-preemptive, jobs are executed in the order they arrive, performs well for long jobs. However, those, along with interactive jobs, hog the CPU, and can lead to a lengthy queue;
- *Shortest Job First:* Classically non-preemptive, but can be made preemptive, select the shortest expected time and run to completion. Need to know processing time, may starve long jobs if short jobs are plentiful;
    - Calculation for determining length of next CPU burst: $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$, where $0 \leq \alpha \leq 1$, t_n is the true length of nth CPU burst, τ_n is the predicted length of nth CPU burst;
- *Round Robin:* Preemptive, suspend current job when its **time slice** is over, will be rescheduled once all other jobs have run. Efficiency depends on time slice length, if short, too much time spent switching jobs, if long, interactive jobs suffer due to many interrupts and lots of waiting;
- *Priority:* Preemptive, each process is assigned a priority, processes run highest priority first. Aging jobs boost in priority to prevent starvation;

Scheduler Classification:

- *Long-Term:* Scheduling done when a process is created. Controls degree of multiprogramming;
- *Medium-Term:* Involves suspend/resume by swapping processes in or out of memory;
- *Short-Term:* Scheduling done frequently, only decides which process to run next;

Multi-Level Feedback: A scheduler design example which works well with interactive workstations:

- Implement several queues, entering jobs enter the queue that matches their assigned priority;
- If a job of higher priority than the one currently running arrives, preempt currently running job;
- Every queue works like a Round Robin: give a time slice for each process, return it to the end of its priority queue once enough time has passed;
- Jobs from queue k will run for time maximal $2^{k-1}T$, where T is the time jobs in the first queue run. Once a job exceeds its allotted time, enqueue it in the next priority queue and execute the next logical process;
- If a job goes to wait in less than T time, place it in queue 1, if a job goes to wait in less than $2T + T$ time, place it in queue 2, etc.;

Memory Management:

Memory Management: Ensure safe execution of programs by sharing memory (where programs and data are stored) and protecting memory. Sharing leads to the following issues:

- *Transparency:* Processes may co-exist in memory without being aware of each other;
- *Safety:* One process may not corrupt another, nor the OS;
- *Efficiency:* Preserve CPU utilization, fairly allocate memory;
- *Relocation:* Ability to run in different memory location;

Storage Allocation: Classification of stored information: Program and data, read-only and read-write, addresses, data, and bindings. This is managed by the compiler, linker, loader, and various run-time libraries.

Creating an Executable:

- *Compiler:* Generates object code;
- *Linker:* Combines all object code into one self-sufficient executable code. Can provide **dynamic linking**, which allows for loading and unloading of routines at runtime;
- *Loader:* Copies executable code into memory. Can load **absolutely**, where programs are always loaded to designated locations, **relocatable**, where they can be in many different locations, or **dynamic**, where functions are loaded only when first called;
- *Execution:* Dynamic memory allocation as code runs.

Address Binding: Assigning instructions and data to memory addresses. Can be:

- *Static:* Where these locations are decided before execution. Can be assigned by the compiler, which translates symbolic addresses (like variables) to absolute addresses, or the loader, which addresses the compiler's relocatable addresses to absolute addresses;

- *Dynamic:* Where these locations are determined during execution, where hardware generates absolute addresses.

Contiguous Memory Allocation Techniques:

- *Direct Placement:* No relocation. User programs are loaded one at a time. Same loading address for every program. As seen in early batch monitors;
- *Overlays:* Large programs organized into tree structure to fit it in smaller memory. Root always loaded, subtrees come and go as needed;
- *Partitioning:* Memory is divided into several contiguous regions, so many can be loaded at once. Static partitioning divides in fixed regions, and programs are loaded into the smallest region which accommodates them. Dynamic partitioning loads programs in as long as there is room for them, addresses are fixed once loaded, so they cannot be relocated;

Dynamic Memory Allocation:

- *First Fit:* Allocate first hole that's big enough for the program;
- *Next Fit:* First Fit using a circular list;
- *Best Fit:* Scan all, allocate smallest hole;
- *Worst Fit:* Scan all, allocate largest hole;

Fragmentation: Unused memory that cannot be allocated. Can be *internal*, where the cause is the difference between the size of the partition and the loaded process, or *external*, where wasted memory is scattered non-contiguously.

Swapping: Treat memory as a preemptable resource. Brings flexibility to programs with fixed partitions, as it can make room for high priority jobs.

Memory Protection: Protect the programs which share memory from each other. Can be seen in hardware with address translation, or software with strong typing and fault tolerance.

Segmentation: Allow each process to be split over several segments, which are regions of contiguous memory. A segment table keeps track of the start and end of every segment. When a process is created, an empty segment table is inserted into the process control block, entries are filled as new segments are allocated. Segments are all released upon termination. This causes external fragmentation. Because of this, only a segment of a process may have to be swapped out to accommodate a new one.

Paging: Physical memory is divided into fixed size blocks (*frames*) which are in turn divided into same size blocks (*pages*). A *page table* maps base addresses of each page for each frame in main memory, a *frame table* keeps track of unused frames. This makes memory swapping easier and reduces fragmentation.

Associative Memory: Use *Translation Buffers* to counteract extra memory references. A very fast but small memory is used to store translation table entries. This is known as the **Translation Lookaside Buffer**.

Use the free() and malloc() operations to allocate and free memory as needed, either in the stack (hierarchical organization, therefore predictable operations) or the heap (loses 50% of blocks, but anything can be accesses).

Free Memory Management:

- *Bit Maps:* Divide memory into fixed-size blocks, keep a bit array which shows which blocks are in use;
- *Linked Lists:* Keep tracks of unused memory in a free list;
- *Buddy System:* As a process requests memory, it is a given a block of memory with a size of power two in which it fits best'

Reclaiming Memory: 2 problems are presented in memory reclaiming:

- *Dangling Pointers:* When the original allocator frees a shared pointer;
- *Memory Leaks:* Occur when we forget to free storage;

Reference Counts: Keep track of the number of pointers to the memory, free the memory when that count reaches zero.

Garbage Collection: Automatic sweep of memory, which deletes dangling pointers and frees memory. Could use more than 20% of CPU time.

Principle of Locality:

- *Locality of Reference:* Processes favor subsets of their address space during execution;
- *Locality:* Small cluster of pages to which most references are made during some time period;
- *Spatial Locality:* Tendency for a process to involve clustered memory spaces;
- *Temporal Locality:* Tendency for a process to reference recently used memory;

<u>Virtual Memory:</u>

Virtual Memory: Allows a program to run without being entirely in main memory. Create an illusion of memory as large as the disk, and as fast as main memory. A program only needs access to a portion of its virtual address at any time (locality of reference). Implemented as an extension of paged or segmented memory management.

Page Fault Handling: Block the current process, find or force and empty frame, determine location of requested page, fetch page to main memory, wake up the process.

Basic Policies: Decisions about Virtual Memory that must be made by the OS

- *Allocation:* how much physical memory is to be allocated to each ready program, lots of frames per program reduces fault rate, little memory reduces need for swapping;
- *Fetching:* When pages have to be brought into main memory;
- *Placement:* Where in the memory a fetched page should be loaded;
- *Replacement:* Which page should be removed when form main memory, can be First-In-First-Out, Least-Recently-Used, Optimality (requires knowing the future), Random;

Clock Algorithm: All pointers, and one used bit per pointer, are kept in a circular queue. A pointer indicates the most recently replaced page, and when a frame is needed, advance the pointer until a 0 bit is found, changing all encountered bits in the process.

Thrashing: When there are too many processes in memory, leads to high page fault frequency, so the system spends too much time swapping processes in and out of memory and not doing useful things.

Working Set: The working set of a program is the set of pages accessed by the last Δ references. A program should run only if its working set is in memory, and a page may not be victimized if it is in the working set of a runnable program.

Page Fault Frequency: $P = \frac{\#\ of\ page\ faults}{specified\ time\ period}$

Virtual Machines:

Virtualization:

- *Multi-Threading:* Virtual CPUs which give the illusion that enables threads to persist their computation as switching happens;
- *Multi-Processing:* Each process has its own memory view; file system and kernel are shared;

Virtual Machine: Abstract hardware of one computer into several execution environments. **Host** is the underlying hardware system, **Virtual Machine Monitor (VMM)** creates and runs virtual machines, providing interface identical to the host's, **Guest** is the process provided with a virtual copy of the host.

Cloud Computing: Multiple clients share the same physical machine (software service, data storage, computing services, …)

Implementing VMMs:

- *Type 0 Hypervisor:* Hardware solution that provide support for virtual machines and their creation;
- *Type 1 Hypervisor:* OS-like software which provides virtualization, general purpose OSs that provide standard and VMM functions;
- *Type 2 Hypervisor:* Run on OSs but provide VMM features to guest OSs;
- *Paravirtualization:* Guest OS is modified to cooperate with VMM for optimization;
- *Programming Environment Virtualization:* VMM creates an optimized virtual system;
- *Emulators:* Allow applications written for one environment to run on a different environment;
- *Application Containment:* Segregates applications from the OS, making them more secure. This is not a real VM;

Sensitive Instructions:

- *Control Sensitive Instructions:* Affect allocation of resources available to VM, change processor mode;
- *Behaviour Sensitive Instructions:* Effect of execution depends on location in real memory;

Privileged Instructions:

- Cause a fault in user mode;
- Work only in privileged mode;

A VMM may be constructed if the set of sensitive instructions for that computer is a subset of the privileged instructions.

Real 360 Architecture: Memory is protectable, all interrupt vectors and clock are in first 512 bytes, I/O done via channeled programs which are initiated with privileged instructions, dynamic address translation.

Trap and Emulate:

- Guest executes in kernel mode;
- Kernel runs in kernel mode;
- Unsafe to let guest kernel run in kernel mode;
- VM needs 2 modes: Virtual User Mode, Virtual Kernel Mode;
- User mode code in guest runs at same as if not a guest;
- Kernel privilege mode runs slower due to trap-and-emulate;

Sensitive x86 Instruction: popf: pops word off stack, set processor flag according to content (all flags in kernel mode, none in user mode).

Binary Re-Writing: Rewrite kernel binaries of guest OSes, replace sensitive instructions with hypercalls, translated code is cached.

Binary Translation: If guest VCPU is in user mode, can run instructions natively. If in kernel mode, examine every instruction, run non-special instructions natively, translate special instructions into new set of instructions that performs equivalent task. Generates native binary code that executes in place of original code.

Paravirtualization: Sensitive instructions replaced with hypervisor calls, VM provides higher-level device interface.

Protection:

Protection: Controlling access of programs, processes, or users. Initially conceived as a part of multi-programming.

Protection Domain: A specification of the resources that a process may access (a set of ordered pairs of the form <object-name, rights-set>). Association between a process and a domain may be:

- *Static:* Always have read and write access in the domain. Don't use this, as it violates the need-to-know principle (idea that a process should have access only to the resources and rights it needs);
- *Dynamic:* Control domain switches as processes require new permissions.

Sandboxing: Restricting access of untrusted code to a subset of resources.

Access Matrix: Specifies what a process executing in domain D_i can access and what operations are allowed on those objects. It may terminate the function if those accesses are violated. This matrix is shaped as a $(Number\ of\ files + Number\ of\ devices + Number\ of\ domains) *$

$(Number\ of\ domains)$ matrix. We often need to change the matrix itself. Implement it either by column (display access control lists) or by row (display capability lists for domains).

Access Control List: An ordered list associated with each object which includes all the domains that can access the object and how they may do that.

Capability List: An ordered list associated with each user which includes all the objects that user may interact with and how they may do that.

Trusted System: A system with no security faults. Not so widespread, as support for legacy systems is not always well implemented, trusted systems need be simple, which is not a staple in today's technological environment.

Trusted Computing Base (TCB): The center of a trusted system. Consists of hardware and software necessary for enforcing security rules, system cannot be compromised if it works well.

Multi-Level Security:

- *Discretionary Access Control:* Individual users determine who may access their files and with which rights. Some access policies are written by the organization and cannot be overridden, without exceptions;
- *Mandatory Access Control:* As well as DAC, regulate flow of information.

Bell-La Padula Model: A multi-level security model designed for the military:

- *Simple Security Property:* A process running at some security level can only read objects at its level or lower;
- ** Property:* A process running at some security level can only write objects at its level or higher

Trojan Horse: Malicious program which misleads users of its true intent (send recorded keystrokes to attacker, look for log files, etc.).

Intruder: Unauthorized individual exploiting a user account or authorized individual misusing privilege.

Detecting Intrusions:

- *Signature Based Detection:* Characterize attack conditions, determine when they occur;
- *Anomaly Detection:* Characterize normal conditions, detect when they are violated.