Chapter 6.

**Latency bound:** minimum value for the Cycle per element for any function that must perform the combining op. in a strict sequence. **Throughput bound:** minimum bound for the CPE based on the maximum rate at which the FU can produce results. **Firmware** are programs stored in ROM (BIOS, controllers for disk, network cards, graphics accelerators, security subsystems) **Volatile Memory** are DRAM and SRAM that lose information if powered off. Non-volatile memory are ROM, PROM,EPROM that can retain value even if powered off. **BUS** is a collection of parallel wires that carry address, data and control signals to transfer cpu-memory. Disk consists of **platters** each with **\*\*two** **surfaces**. each **surface** consists of concentric rings called **tracks**. Each tracks consists of **sectors** separated by **gaps**. Average time to access = **Tav.seek + Tsv.rotation+ Tav.transfer** where Tseek is time to position heads over cylinder containing target sector. Trotational =1/2 \* 1/f (f is in rev/ms)is time waiting for first bit of target sector to pass under r/w head. Ttransfer = 1/f \* 1/xsec/track time to read the bits in the target sector. The key to bridging CPU-Memory gap is a fundamental property of computer programs known as **locality. Principle of Locality** is “Programs tend to use data and instructions with addresses near or equal to those they have used recently. **Temporal locality**: Recently referenced items are likely to be referenced again in the future. eg:reference variable sum each iteration. cycle through loop repeatedly. **Spatial locality**: Items with nearby addresses tend to be referenced close together in time. eg: reference array elements in succession. reference instructions in sequence. **Cache**: A smaller faster storage device that acts as a staging area for a subset of the data in a larger, slower device. **Linking**: process of collecting and combining various pieces of code and data into a single file that can be loaded into memory and executed. **When Linking happen?:** i.Compile time: when program is compiled. ii. Load time: when a program is loaded into memory. iii. Run time: While a program is executing. \***Importance Linkers:** i. **Modularity**: Program can be written as a collection of smaller source files rather than one monolithic mass. ii. Efficiency: separate compilation**. 5 benefits** **of linkers helps**: i. build large programs(linker errors:missing modules or libraries) ii. avoid dangerous programming errors(eg. multiply defined global variable) iii.implementation of language scoping rules(eg. global vs local variables, static functions). iv. understand other systems like virtual memory **What Linkers do?** **stepi**. In Symbol resolution, linker associates each symbol reference with exactly one symbol definition. **step ii** In recolation:merges separate code and data sections into single sections. Relocates symbols from their relative in the .o files to their absolute memory locations in the executable. Updates all references to these symbols to reflect their new positions. **Kinds of Object Files**: i. **Relocatable object file**(.o file) contains code and data in a form that can be combined with other relocatable object files to form executable object file. ii.**Executable object file**(a.out file) contain code and data in a form that can be copied directly into memory and then executed. iii.**Shared object file**(.so file) special type of relocatable file that can be loaded into memory and linked dynamically at load time or run-time, called **Dynamic Link Libraries(DLLS). Executable and Linkable Format)ELF OBJECT FILE format:** i. elf header(word size, byte ordering, file type), ii.Segment header table(Page size, virtual address memory segments, segment sizes). iii. .text section: machine code of compiled program iv. .rodata section: read only data: jump tables, format strings, printf v. .data section: initialized global and static C variables vi. .bss section: uninitialized global and static C variables. occupies no space vii. .symtab section: Symbol table, procedure and static variable names, section names and locations. viii. .rel.text section: relocation info for .text section. Addresses of instructions that will need to be modified in executable. ix. .rel.data section: relocation info for .data section. Addresses of instructions that will need to be modified in the merged executable. x. .debug section: info for symbolic debugging(gcc -g). xi.Section header table: offsets and sizes of each section. **Linker Symbols**: i. **Global symbols**: symbols defined by module m that can be referenced by other modules. eg: non-static C functions and **non-static** global variables. ii**. External Symbols**: Global symbols that are referenced by module m but defined by some other module. iii.**Local Symbols**: Symbols that are defined and referenced exclusively by module m. C functions and global variables with the **static** attribute. Local Linker symbols are not local program variables( local nonstatic variables are managed on the stack, no interest to linker). **Local non-static C variable** is stored on the stack and **Local static C variables** is stored in either .bss or .data. **Program Symbols : Strong**->are procedures and initialized globals. **Weak** are uninitialized globals. **Linkers Symbol Rules**: **Rule 1**.Multiple strong symbols are not allowed: each item can be defined only once otherwise error. **Rule 2**: Given a string symbol and multiple weak symbols, choose the strong symbol. **Rule 3:** If there are multiple weak symbols, pick an arbitrary one. Global Variable AVOID otherwise i. use static or initialize if you define a global variable or use extern if you reference an external global variable. **using Static library :**Linkers **algorithm** for resolving external references: i. scan .o files and .a files in the command line order. ii. During the scan, keep a list of the current unresolved references. iii. As each new .o or .a file, obj, is encountered, try to resolve each unresolved reference in the list against the symbols defined in obj. iv. If any entries in the unresolved list at end of scan, then error. **Problem:** command line order matters. Solution: Put libraries at the end of the command line. **STATIC Libraries Disadvantage:** i.Duplication in the stored executables. ii. Duplication in the running executables. iii. Minor bug fixes of system libraries require each application to explicitly relink. **Modern Solution**: i. object files that contain code and data that are loaded and linked into an application dynamically, at either load-time or run-time. **Load-Time Linking:** Dynamic linking that can occur when executable is first loaded and run.eg: Standard C library **Run-Time Linking**: Dynamic linking that can occur after program has begun. Done by calls doplen() interface.eg: Distributing Software. High performance we servers.

Chapter 9

Virtual(Logical) address: Logical Address is generated by CPU while a program is running. The logical address is virtual address as it does not exist physically, therefore, it is also known as Virtual Address. This address is used as a reference to access the physical memory location by CPU. The term Logical Address Space is used for the set of all logical addresses generated by a program’s perspective.

The hardware device on CPU chip called **Memory-Management Unit** is used for mapping logical address to its corresponding physical address. It translates virtual addresses on the fly, using a lookup table stored in main memory whose contents are managed by the operating system.

Physical Address: Physical Address identifies a physical location of required data in a memory. The user never directly deals with the physical address but can access by its corresponding logical address. The user program generates the logical address and thinks that the program is running in this logical address, but the program needs physical memory for its execution, therefore, the logical address must be mapped to the physical address by MMU before they are used. The term Physical Address Space is used for all physical addresses corresponding to the logical addresses in a Logical address space.

The task of converting a virtual address to a physical one is known as **address translation**.

Address Spaces: is an ordered set of non-negative integer addresses {0,1,2,…} If the integer in the address space are consecutive, then it is a linear address space. For 64-bit virtual address spaces, N = 2^n = 2^64. CPU generates virtual addresses from an address space of N = 2^n addresses call virtual address space.

DRAM is at least 10 times slower than an SRAM and the disk is about 100,000 times slower than a DRAM. Thus, misses in DRAM caches are very expensive compared to misses in SRAM caches because DRAM cache misses are served from disk, while SRAM cache miss are usually served from DRAM-based main memory. Due to large miss penalty DRAM caches are fully associative, that is **any virtual page can be placed in any physical page.**

**virtual memory is an array of N contiguous bytes stored on disk.**

**A page table is a data structure stored in physical memory that maps virtual pages to physical pages. If valid bit is not set, null address indicates that the virtual page has not yet been allocated.**

**Page hit: reference to VM word that is in physical memory (DRAM cache hit). Page fault: reference to VM page that is not in physical memory (DRAM cache miss).**

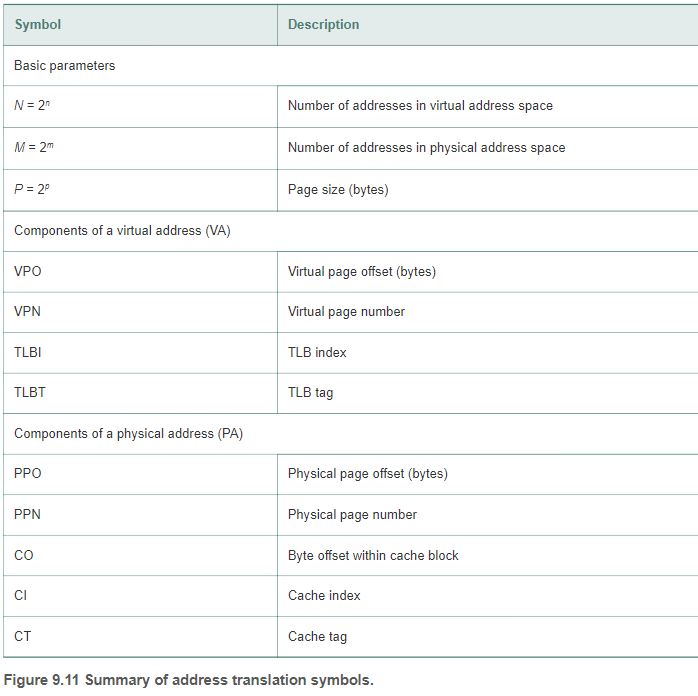
**Waiting until the miss to copy the page to DRAM is known as demand paging**

**The activity of transferring a page between disk and memory is known as swapping or paging.**

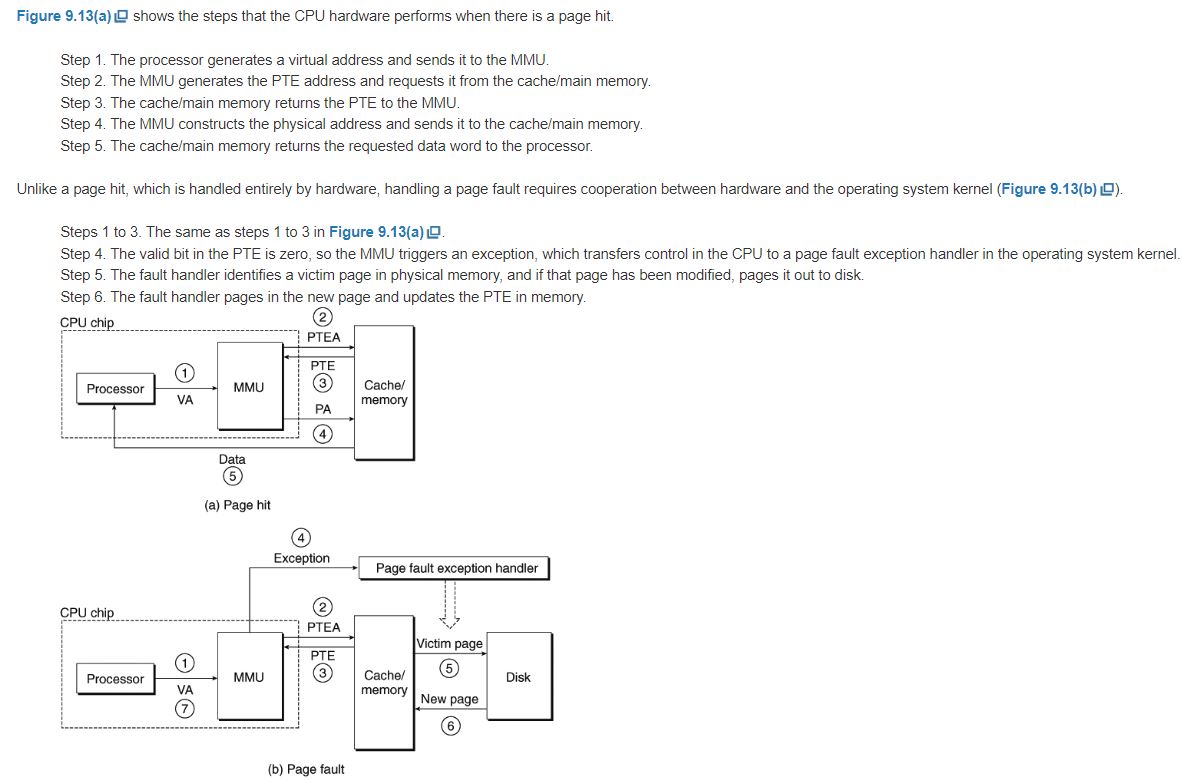
**At any point in time, programs tend to access a set of active virtual pages called the working set.**

**If (working set size < main memory size), Good performance for one process after compulsory misses**

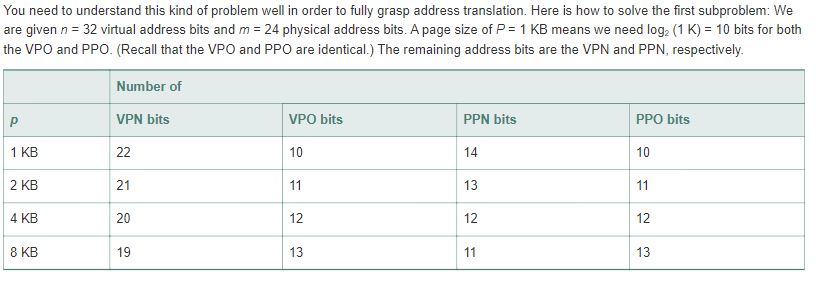
**If ( SUM(working set sizes) > main memory size ), Thrashing: Performance meltdown where pages are swapped (copied) in and out continuously**

****

**PTEA: Page table entry address. Page hit and misses.**

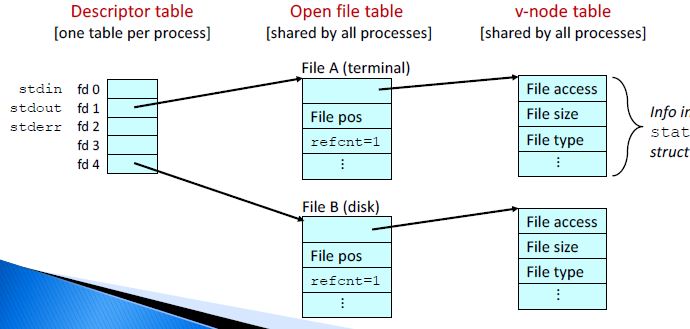
****

**Solution 9.3**

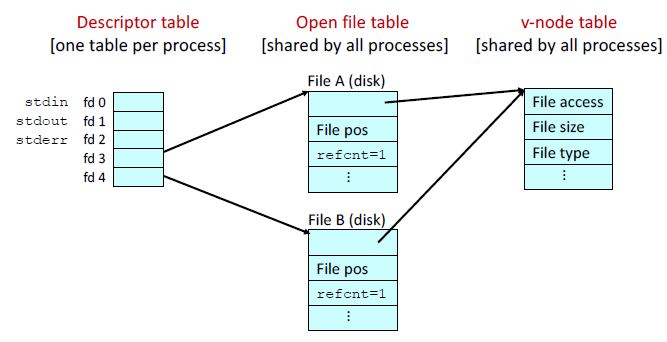
****

**Translation lookaside Buffer** is a small virtually addressed cache where each line holds a block consisting a single PTE.

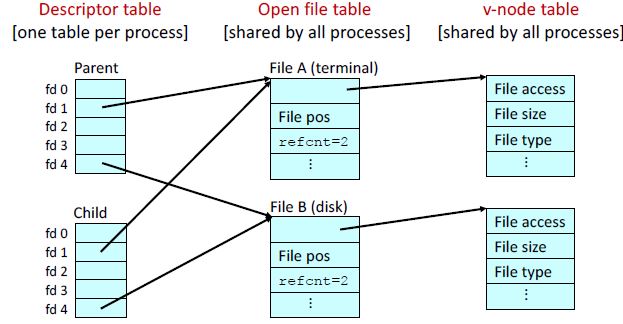
**A linux File** is a sequence of m bytes: B0, B1, B2, … B(K-1) eg1: I/O devices: /dev/sda2 & /dev/tty2 eg2:kernel: /boot/vmlinuz-3.13.0-55-generic. **File Types:** regular file(contains arbitrary data), directory(index for a related group of files) and Socket(communicating with a process on another machine). Textfiles are regular files with only ASCII or Unicode characters. Binary files are everything else eg:objectfiles, jpeg\_images Text file is sequence of text lines and text line is sequence of chars terminated by the new line char. Newline is 0xa same as ASCII line feed character(LF). Directories consists of an array of links: each link maps a filename to a file. Each directory contains at least two entries: .(dot):link to itself, ..(dotdot): link to the parent directory in the directory hierarchy. Absolute pathname(path from th root node)‘/home’ and relative pathname (path from the current working directory)‘../home’.**Opening fileint fd; /\* file descriptor \*/if ((fd = open("/etc/hosts", O\_RDONLY)) < 0) {perror("open");exit(1);}** informs the kernel that you are getting ready to access that file. returns a small identifying integer file descriptor:fd==-1 indicates that an error occurred. Each process created by a Linus shell begins life with three open files associated with a terminal, 0:standard input(stdin),1:standard output(stdout),2:standard error(stderr). **Closing a fileint fd; /\* file descriptor \*/int retval; /\* return value \*/if ((retval = close(fd)) < 0) {perror("close");exit(1);}** informs the kernel that you are finished accessing that file. Closing an already closed file is a recipe for disaster in threaded programs.Moral:Always check return codes for close(), open().**ReadingFile char buf[512];int fd; /\* file descriptor \*/int nbytes; /\* number of bytes read \*//\* Open file fd ... \*//\* Then read up to 512 bytes from file fd \*/if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {perror("read");exit(1);}**copies bytes from the current file position to memory and updates the position. returns number of bytes read from file fd into buf:return types ssize\_t is signed integer. nbytes<0 indicates that an error occurred. short counts(nbytes<sizeof(buf) possible and not errors. 0 indicates EOF(end of file). **WritingFiles char buf[512];int fd; /\* file descriptor \*/int nbytes; /\* number of bytes read \*//\* Open the file fd ... \*//\* Then write up to 512 bytes from buf to file fd \*/if ((nbytes = write(fd, buf, sizeof(buf)) < 0) {perror("write");exit(1);}**copies bytes from memory to the current file position, and updates current file position., returns number of bytes written from buf to file fd:nbytes<0 indictaes that an error occurred. As with reads, short counts are possible and are not errors.**Example:(SimpleUnixI/O eg)Copying stdin to stdout, one byte at a time)::#include "csapp.h" int main(void){char c;while(Read(STDIN\_FILENO, &c, 1) != 0)Write(STDOUT\_FILENO, &c, 1);exit(0);} Metadata:** is data about data that an application can **retrieve/access** by calling the stat and fstat functions.

The KERNEL represents open files using three related **Data Structures:(NO Sharing below)(BeforeFork)** **Two descriptors referencing two distinct open files.Descriptor 1 (stdout) points to terminal, and descriptor4 points to open disk file**

The KERNEL represents open files using three related **Data Structures(Sharing)** **Two distinct descriptors sharing the same disk file through two distinct open file table entries.** **E.g. Calling open twice with the same filename argument**

****

Process ShareFile: **FORK** :A Child process inherits its parent’s open files.**(AfterFORK)**

****

How does a shell implement I/O redirection? linux> ls > foo.txt Answer: By calling the dup2(oldfd, newfd) function:Copies (per-process) descriptor table entry oldfd to entry newfd

I/**O Redirection Example**: Stepi. open file to which stdout should be redirected which happens in child executing shell code, before exec. Step2. call dup2(4,1), which cause fd=1(stdout)to refer to disk file pointed at by fd=4.

eg standard I/O function: fopen,fclose,fread,fwrite,fgets,fputs,fscanf,fprintf: opening,closingfiles, reading,writingbytes ,reading,writinglines, formattedreadingwriting.

Pros of Unix I/O: Most general and lowest overhead for of I/O, all other I/O packages are implemented using UNIX I/O functions. Provides functions for accessing file metadata. Async-signal-safe and can be used safely in signal handlers. CONS: dealing with short counts is tricky and error prone. Efficient reading of text lines requires some form of buffering, also tricky and error prone. Both of these issues are addressed by the standard I/O packages.

Pros of Standard I/O: Pros: Buffering increases efficiency by decreasing the number of read and write system calls. Short counts are handled automatically. Cons:Provides no function for accessing file metadata. Standard I/O function for accessing file metadata. Standard I/O functions are not async-signal-safe, and not appropriate for signal handlers. Standards I/O is not appropriate for input and output on network sockets. There are poorly documented restrictions on streams that interact badly with restrictions on sockets.

**USE standard I/O when** working with disk or terminal files**. Use raw Unix I/O inside** signal handler, because Unix I/O is async-signal-safe and in rare case when you need absolute highest performance.

When working with binary files, i. text oriented I/O such as fgets, scanf ii. String functions such as strlen, strcpy, strcat.

Concurrent Challenges: Races:outcome depends on arbitrary scheduling decisions. Deadlock:improper resource allocation prevents forward progress. LiveLock/Starvation/Fairness: external events and/or system scheduling decisions can prevent subtask progress. **ProcessBased**:kernel automatically interleaves multiple logical flows. Each flow has its own private virtual address space. needs explicit inter-process communication(IPC) mechanismeg. socket interface. **EventBased:**Programmer manually interleaves multiple logical flows in context of signle process. Flows modeled with state machine, state transitions based on data arrival. All flows share the same address space. Use Technique called I/Omultiplexing. **ThreadBased:** kernel automatically schedules multiple logical flows. Process flows share the same address space. Hybrid of process-based and event based. **PropertiesofMultipleThreadsinProcess:** each thread has its own logical control flow, each thread shares the same code, data, and kernel context. Each thread has its own stack for local variables, but not protected from other threads. Each thread has its own thread if(TID).. Two threads are concurrent if their flows overlap in time, otherwise sequential. Single core Processor simulates parallelism by time slicing. Multi-Core processor can have true parallelism. **Threads & Process Similar:** each has its own logical control flow. each can run concurrently with others(possibly on different cores). each is context switched. **Difference**: Threads share all code and data(except local stacks) but processes(typically) do not. Threads are somewhat less expensive than processes in smaller context. Process control(creating and reaping) twice as expensive as thread control. Linux numbers:~20k and ~10k cycles to create and reap a process and thread. **P**osix Thread: standard interface, 1995, ~60 functions to manipulate threads from c programs.: creating and raping threads: pthread\_create() pthread\_join. Determining thread: pthread\_self(), Terminating: pthread\_cancel() pthread\_exit() exit(). Synchronizing: pthread\_mutex\_init & pthread\_mutex\_[un]lock. **PROS: i.** easy to share data structures between threads: eg: logging information, file cache **ii.** Threads are more efficient than process. **CONS i.**ease with which data can be shared is both the greatest strength and the greatest weakness of threads. **ii**. Hard to know which data shared & which private. **iii**. Hard to detect by testing since probability of bad race outcome very low. **Mapping Variable: Global:** variable declared outside of a function->Virtual Memory contains exactly one instance of any global variable. **Local**: Variable declared inside function with the static attribute. Each thread stack contains one instance of each local variable. **Local Static**: Variable declared inside function with the static attribute. Virtual Memory contains exactly one instance of any local static variable. **Variable v is shared** **if and only if** one of its instances is referenced by more than one thread. Progress graph depicts the discrete execution state space of concurrent threads. Each axis corresponds of the sequential order of instructions in a thread. Each point corresponds to a possible execution state (L1, S2) denotes state where thread 1 has completed L1 and thread 2 has completed S2. A trajectory is a sequence of legal state transitions that describes one possible concurrent execution of the threads. **A Trajectroy** is safe iff it does not enter any unsafe region. A trajectory is correct iff it is safe. **TO Guarantee a Safe Trajectory** : We must synchronize the execution of the threads so that they can never have an unsafe trajectory i.e. need to guarantee mutually exclusive access for each critical section. solution: semaphores, monitors, mutex and condition variables. **Semaphores:** non-negative global integer synchronization variable. Manipulated by **P and V operations**. **P(s):** i. If s is non zero, then decrement s by 1 and return immediately.->Test and decrement operations occur atomically(indivisibly) ii.If s is zero, then suspend thread until s becomes nonzero and the thread is restarted by a V operation. After restarting, the P operations decrement s and returns control to the caller. **V(s)** i.increments s by 1. If there are any threads blocked at a P operation waiting for s to become nonzero, then the V operation restarts exactly one of these threads, which then completes its P operation by decrementing s. Semaphore invariants (s>=0). **Using Semaphores for mutual exclusion:** associate a unique semaphore mutex, initially 1, with each shared variable (or related set of shared variables) and surround corresponding critical sections with P(mutex) and V(mutex operations. **Binary semaphore:** semaphore whose value is always 0 or 1. **Mutex** is **binary semaphore** used for mutual exclusion i. P operation: “locking” the mutex ii. V operation: “unlocking” or “releasing” the mutex iii.”Holding” a mutex: locked and not yet unlocked.. **Counting semaphore**: used as a counter for set of available resources. **Proper Synchronization** i.Define and initialize a mutex for the shared variable. ii. Surround critical section with P and V. **WHY MUTEX WORK:** It provides mutually exclusive access to shared variable by surrounding critical section with P and V operations on semaphores s (initially set to 1). And, since, semaphore invariant creates a forbidden region that encloses unsafe region and that cannot be entered by any trajectory. **Accessing shared resources by threads->**Thread uses a semaphore operation to notify another thread that some condition has become true: i. use counting semaphores to keep track of resource state and to notify other threads ii.use mutex to protect access to resource. **Starvation:** is state where a thread waits indefinitely. A process is **deadlocked iff** it is waiting for a condition that will never be true. Typical scenario: i. process 1 and 2 needs tworesources(A andB) to proceed ii.Process 1 acquires A, waits for B iii. Process2 acquires B, waits for A. Both will wait forever. **Race: event** occurs when the correctness of the program depends on one thread reaching point x before another thread reaches point y.