week 1: creation of process

-When the process is not executing on a CPU, the virtual CPU state must be saved so that it can be restored on the real CPU, once the process is restarted.

-**Multitasking** is the ability of the OS to share physical memory and the CPU(s) among many processes. The result is one or more processes in some state of execution at the same time.

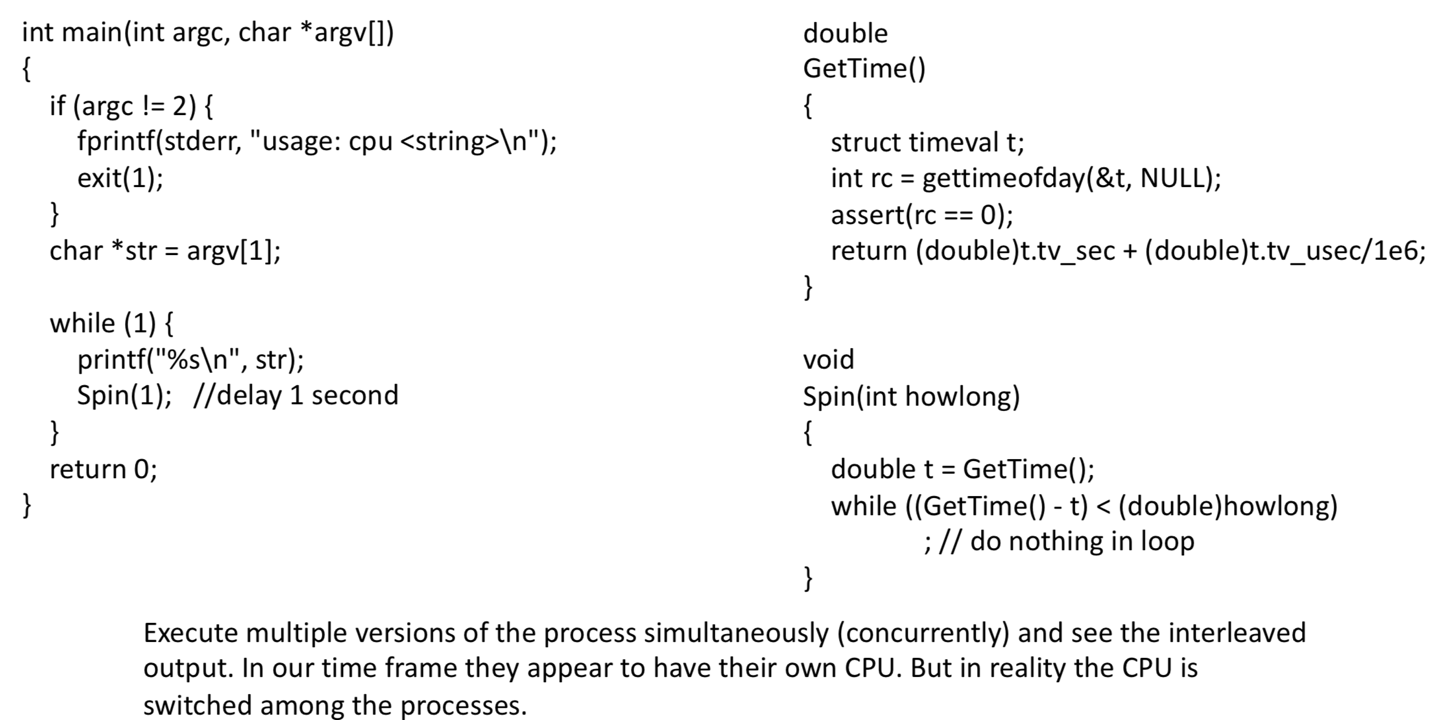
-The operating system provides the control or allocation of the CPU to the program and access to the other components of the computer system

-It cyclically executes fetch instruction, decode instruction, execute instruction

-**Operating System** (OS): **Software** that converts hardware into a useful form for applications

-**Program** – static instructions that we develop to control the CPU

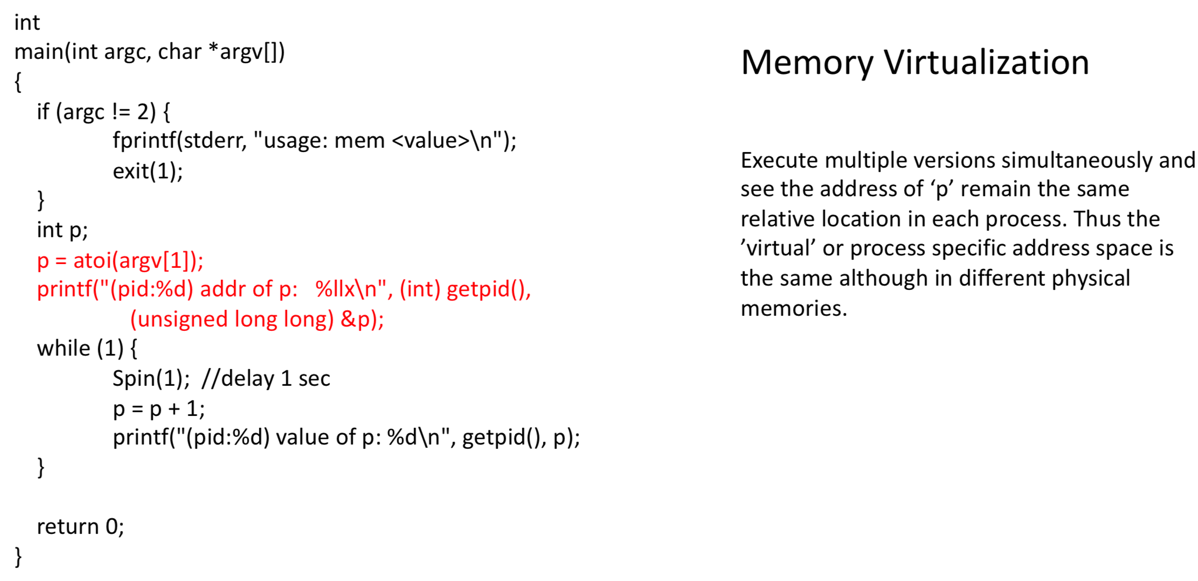
-**Computer system** has devices other than the CPU: Memory, Disks, other I/O, network, etc



That is, each process has a ‘virtual CPU

- we only have one processor , somehow four of these program seem to run at same time

- illusion many virtual CPU exist called virtualizing the CPU



-In a second example, a simple program declared a variable and stored a value in the variable. When executing multiple instances of the program, we found that the memory address for the variable was the same for each process instance.

- we learned that addressing within a process is a relative or virtual address

- Each process instance occupied its own physical memory space, yet the address of the variable is the same!

- As we study memory management: we will find that the CPU executes program code using virtual addresses (that is, program relative addresses) and converts the virtual address to a physical address to get to the real memory cell

**PCB**: holds the ‘process state’ when the process is not executing(not using the CPU).

-There are two components of the PCB: the state of the CPU and the state of resources associated with the process

-When the program starts, the program counter (PC) and stack pointer (SP) in the CPU are set to point to the locations in the process program code and stack. The fetches and executes of the CPU are via these CPU register

-Each program has an address space, created by the linker when the image of the program is created. This address space is loaded into memory. A program executes with its own ‘address space’ yet resides in a memory space. This means that the program addresses must be converted to real addresses as the program executes. This is memory virtualization and we will study this in detail.

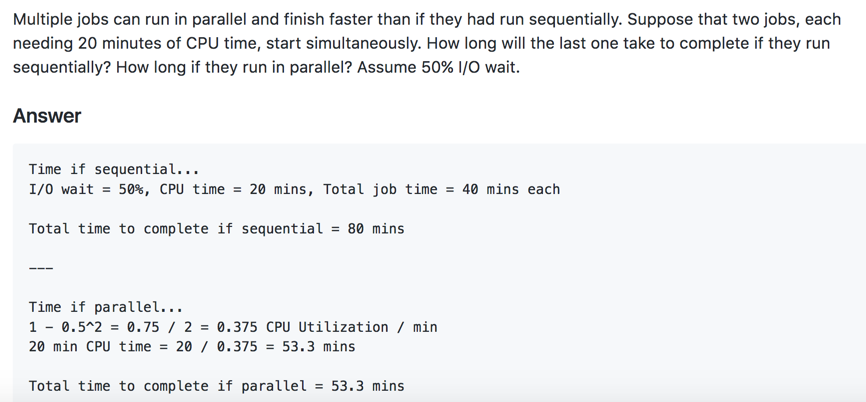
What is multiprogramming

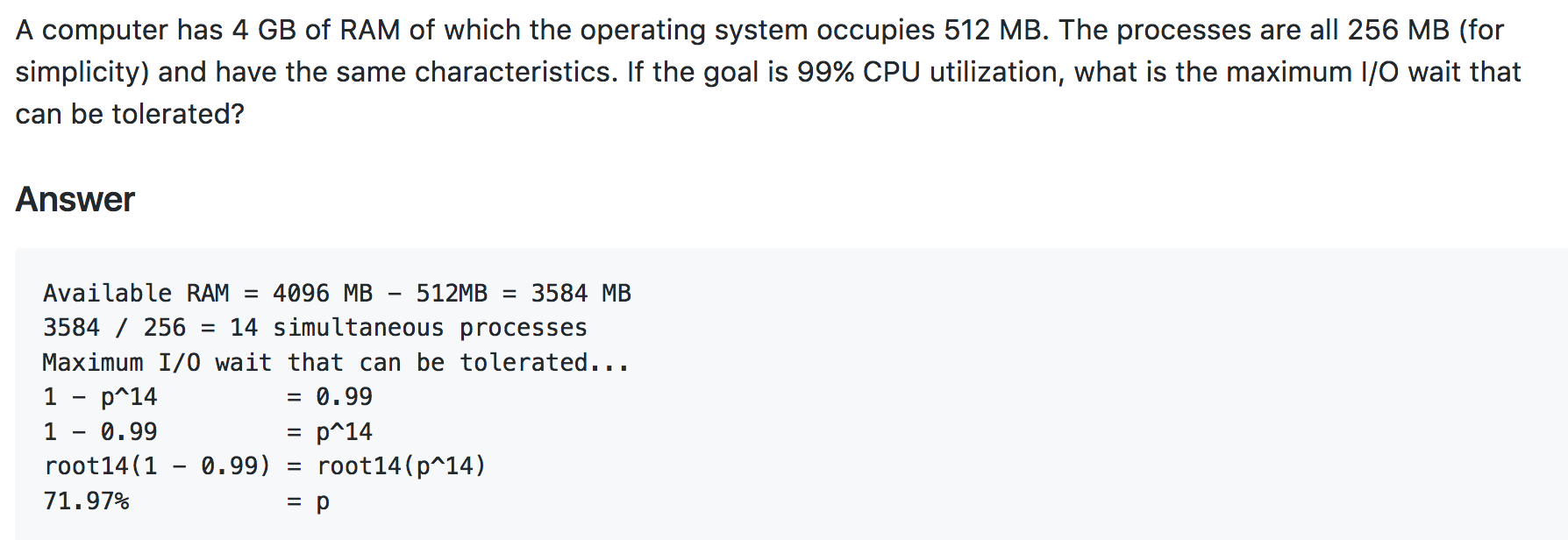
- The CPU allowed multiple programs to run. This increases CPU utilization, decreases response and turnaround time While one program is waiting for I/O, another can be using the CPU.

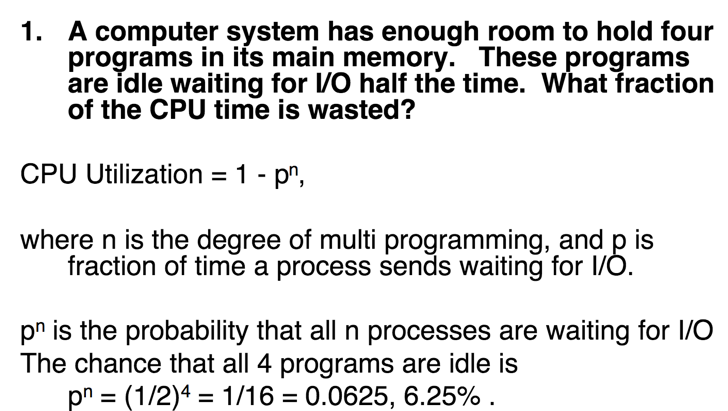
Which of the following instructions should be allowed only in kernel mode?

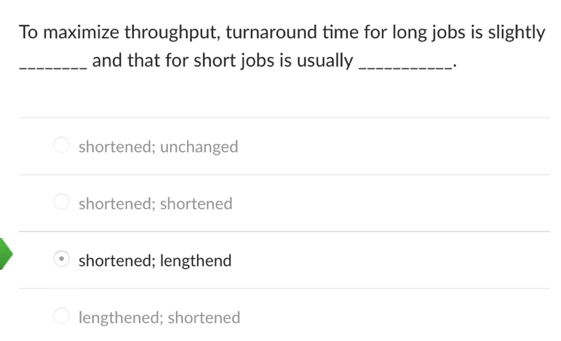
1. disable all interrupts
2. read the time of day clock
3. set the time of day clock
4. change the memory map

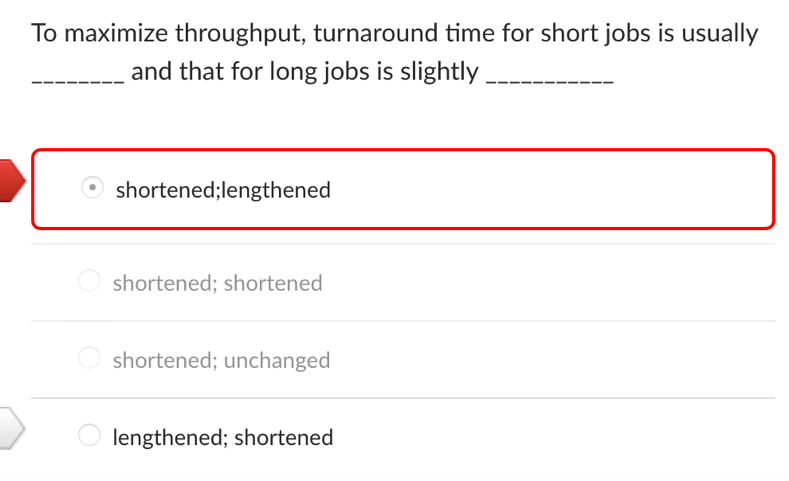
-a Kernel mode only. Obvious, c: Needs to be done only in kernel mode otherwise, a job could set the clock back to increase its processor time slice (among other things) , d: Kernel mode only. b: but the reading of the time-of-day clock should be allowed in both the user and kernel mode.











week 2: Process scheduling

- we know that the ‘virtual CPU’ must include the ‘state’ of the process

- We looked at the process scheduler and its function of switching a virtual CPU onto the real CPU

- A primary job of the process scheduler: (a component called the dispatcher) is to save the state of the executing process so that it can be restored when the process continues to run in the future, and restore the state of the process that has been selected to run next.

- The state of the process will include all of the information that describes the contents of the CPU at the time process stops running, its stack and some other private data

- Multitasking means that multiple processes (i.e., running programs, or executing elements) can simultaneously reside in memory and each uses the CPU(s

- Preemptive means that the rules governing which processes receive use of the CPU and for how long are determined by the kernel process scheduler (rather than by the processes themselves

- **Preemptive** indicates that that the scheduled takes the CPU away from an executing process when there is a higher priority activity that needs the CPU.

-Creation and termination of processes: The kernel can load a new program into memory, give it resources (e.g., CPU, memory, and access to files) in order to run. Such an **instance of a running program** is termed a **process**. The kernel ensures that the resources it uses are freed for subsequent reuse once the process terminates

-**The kernel through** the memory management system must convert the virtual addresses provided by the CPU to the physical memory addresses as the program executes.

- **Process virtual addresses begin at 0 and continue to the size of the process**.

- The physical address space has cells numbered from 0 to 2k-1, for the 2k storage cells

- Processes are stored together in main or real memory, so the **virtual addresses must be converted to the actual addresses as the process executes**.

- Physical – the actual address space or set of real addresses in the main memory of the CPU

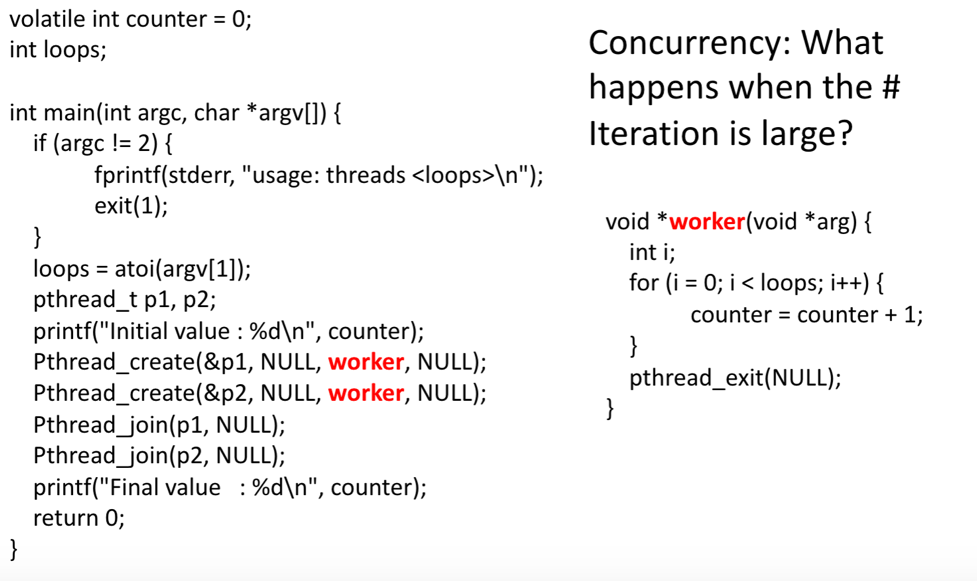
- Virtual – the address space of a process

- As a process executes, it’s (virtual) addresses must be mapped or associated with, real/physical addresses

- Values stored in memory are binary and there use depends on the context of access

-**a thread** is an independent execution unit within a program.

-(Every program contains at least one thread



A single process was executed and we expected that for a specified number of loop iterations, N, a result of 2N would be printed. This was true when N was relatively small. When we made N large, we got results that were less than 2N or greater than 2N – all incorrect.

-The CPU was switched among the threads in the same way it would be switched among multiple processes. When the loop count was small, the thread was able to complete its iterations before the other thread was executed. This a result of 2N. When the loop count was large, the CPU was switched away from the thread before it finished and the other thread operated on the counter variable for a while.

-In addition, incrementing a variable is actually multiple machine operations ( load variable to register, add one to register, store result in register). (p++)

-The kernel 'owns' all devices in the computer system and shares these resources and access to them to processes

**Persistence**: Provide permanent access to information

-Lifetime of information is longer than lifetime of any one process

**RAM”** is volatile

Is RAM Persistence storage:

-not, data is erased when power is turned off

what is persistence storage

-any data **storage** device that retains data after power to that device is shut off.

Is ROM Persistence storage

-yes, typically used for the task of secondary **storage**, or long-term **persistent storage**. non-volatile

-**As the CPU is switched** between these executing units, the state of the execution must be saved before the switch and then the state of the 'switched to' unit restored.

-instruction pointer (program counter). stack pointer, and processor status word (PSW). These are a significant part of the dynamics of the execution of a process and are volatile (that is, are lost, if not saved, when the CPU is reallocated to another process).

- The general and special **purpose registers** of the CPU provide the state of the CPU that will need to be saved and then restored if we want a preempted program to continue in the future without loss of information.

**- Interrupts** result in the suspension of execution of the current process and start the execution of an interrupt service routine (ISR)

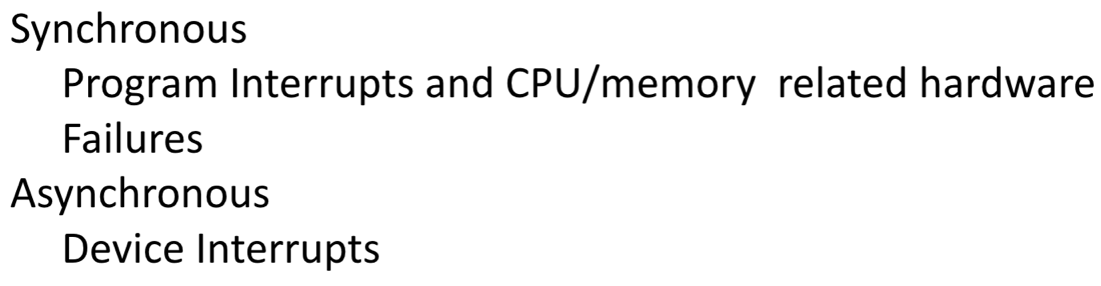
State of Process

-PC, SP,

-Processor status word: Internal state of the CPU while executing • Sets of bits allocated to certain functions or status

- Each instance or type of **interrupt** has an associated ISR and that ISR is executed in response to the interrupt. This is done by hardware because the instruction pointer (program counter) must be pointing to the first instruction in the ISR before the next CPU fetch. PSW to be used while executing within the ISR.

-Regardless of interrupt type, each interrupt requires servicing. Servicing is done within some code written specifically for that interrupt type



-The same interrupt mechanism is used for servicing interrupts from external devices, the CPU (for example CPU faults), special program instructions, or timer type devices. The mechanism involves saving the critical components of the current state of the CPU (the program counter and processor status word, at a minimum) **on the kernel stack**,

When a process is executing, it is assigned the CPU and instructions are fetched and executed. • For what reasons may it give up the CPU?

Time Savings Due to the Use of Interrupts,

rather than I/O Wait

Interrupts; short I/O wait

A **trap** is a software-generated interrupt caused either by an error (an exception) or a **user request** (that is an instruction

Week 3: Interrupt , hardware support, processes

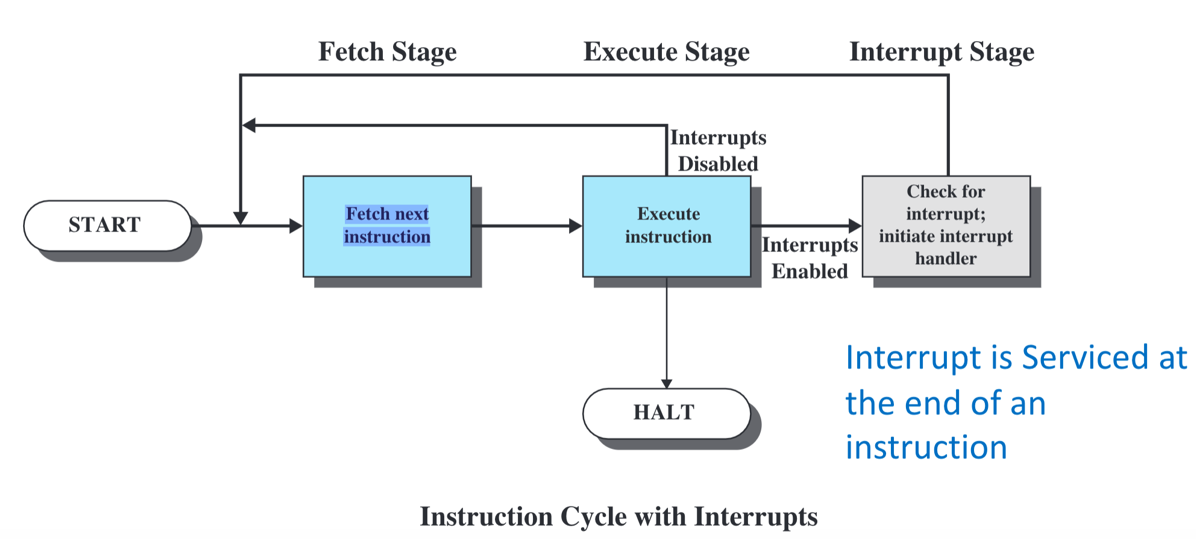
-Interrupt is Serviced at the end of an instruction

- If an interrupt occurs when a process is executing, the interrupt is serviced at the end of the instruction during which the interrupt arrived

- Since the ISR will be executing on the same CPU as the interrupted code, anything in the CPU belonging to the executing process is volatile and must be saved.

-The immediate critical items are the instruction pointer (IP or PC) and the PSW. These will have to be restored if the process is to restart where it left off.

- Hardware pushes the IP and PSW onto the kernel stack and loads a new IP and PSW into the CPU from the IV of the interrupting device. This happens between the last instruction executing for the process and the next fetch by the CPU, guaranteeing that the next fetch is from the ISR



-Interrupt servicing can be disabled. This can be done by the kernel. Interrupt requests that occur when interrupts are disabled are not necessarily lost.

-The device raising the request while interrupts are disabled has its interrupt request left in a 'pending' state. Once interrupts are reenabled, the request will be honored, and the interrupt serviced

-The PSW is a very important register as we have discussed. It is a privileged CPU register, accessible only to the kernel. So far, we have discussed some of the bits allocated in the PSW: mode bit, CPU priority bits.

-The switch from kernel mode to user mode happens when a new process or thread is created and started [fork(),

-Return from an ISR to an interrupted process involves a return from kernel mode to user mode.

Transparent restartable execution :User program does not know interrupt occurred

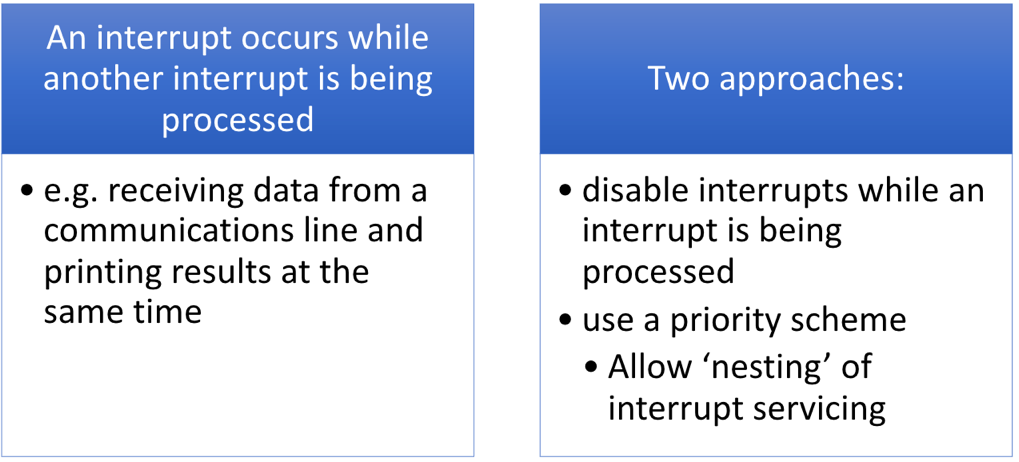
-Entry to kernel mode from user mode occurs automatically as a result of interrupt servicing.

-Interrupt servicing begins in kernel mode since the ISR is code in the kernel. Recall that there are several types of interrupts, including traps, which are instructions to execute kernel code.

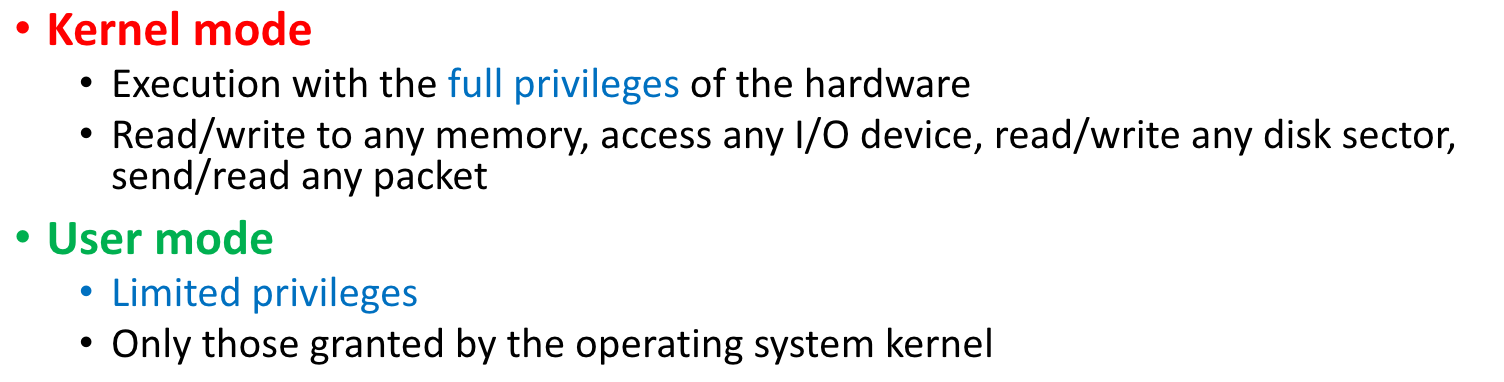
-The memory hierarchy illustrates the properties of the speed of access, cost and size of storage.

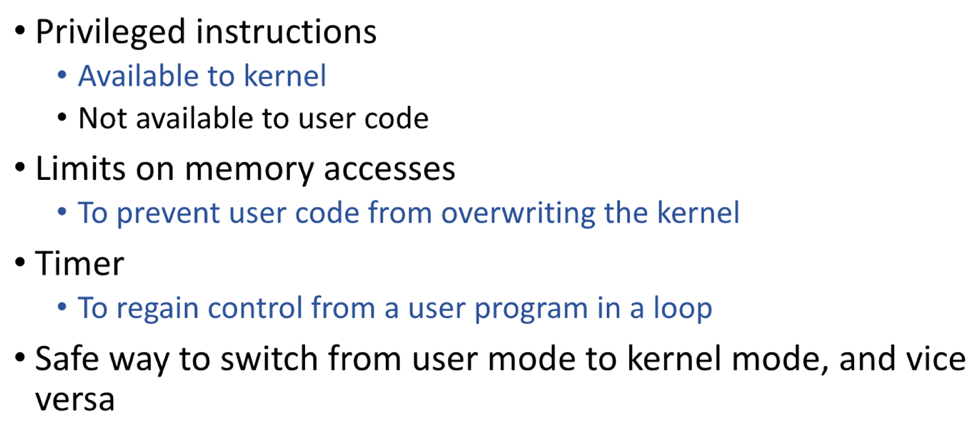
-The key to the success of this organization is the decreasing frequency of access at lower (slower) levels.

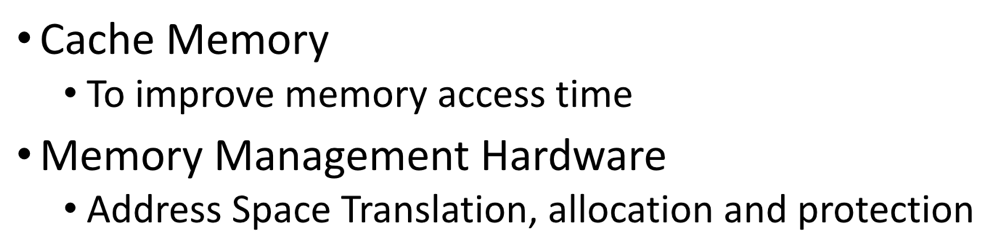
- Multiple interrupt



**Kernel**: core code of the operating system that manages resources and provides services to other parts of the OS







* Identifier: A unique identifier associated with this process, to distinguish it from all other processes.
* State: If the process is currently executing, it is in the running state.
* Priority: Priority level relative to other processes.
* Program counter: The address of the next instruction in the program to be executed.
* Memory pointers: Includes pointers to the program code and data associated with this process, plus any memory blocks shared with other processes.
* Context data: These are data that are present in registers in the processor while the process is executing.
* I/O status information: Includes outstanding I/O requests, I/O devices (e.g., disk drives) assigned to this process, a list of files in use by the process, and so on.
* Accounting information: May include the amount of processor time and clock time used, time limits, account numbers, and so on.

**Spatial** –adjacent cells, or cells in relatively close memory regions

**Temporal** –reuse of cells or resources within a relatively short time duration

Week 4: cpu scheduling

-We can characterize the behavior of an individual process by listing the sequence of instructions that execute for that process. Such a listing is referred to as a ‘trace’ of the process.

-In virtually all operating systems, each process is assigned a unique numeric identifier, which may simply be an index into the primary process table;

-When processes can create other processes, identifiers indicate the parent and descendants of each process.

**fork**

A parent process uses fork() to create a new child process. The child process is a copy of the parent. After the fork, both parent and child executes the same program but in separate processes.

**exec**

Replaces the program executed by a process. The child may use exec after a fork to replace the process’ memory space with a new program executable making the child execute a different program than the parent.

**exit**Terminates the process with an exit status.

**wait**

The parent may use wait to suspend execution until a child terminates. Using wait the parent can obtain the exit status of a terminated child.

-On success fork returns twice: once in the parent and once in the child. After calling fork, the program can use the fork return value to tell whether executing in the parent or child.

-If the return value is 0 the program executes in the new child process.

-If the return value is greater than zero, the program executes in the parent process and the return value is the process ID (PID) of the created child process.

-On failure fork returns -1.

-The wait system call blocks the caller until one of its child processes terminates. If the caller doesn’t have any child processes, wait returns immediately without blocking the caller. Using wait the parent can obtain the exit status of the terminated child.

-If you don’t want to execute the same program in both the parent and the child, you will need to use a system call of the exec family. The **exec system calls** will replace the currently executing program with a new executable.

When a process calls exec, all code (text) and data in the process is lost and replaced with the executable of the new program.

-An exec call is often used following the creation of a child process (i.e., following a fork() ). But, an exec can be used to overlay any executing program with another. **This is done without have to create a new process**. The ‘overlaid code/data’ begin executing within the “context of the existing process”.

Direct Memory Access (**DMA**) is a mechanism that enables a device to transfer information to or from addresses in the address space

Memory-mapped I/O extends the use of a machine's address space to provide access to devices beyond direct memory storage (RAM and ROM).

Week 5: Thread

-**Threads** are like processes, except: multiple threads of same process share an address space

-Takes less time to create a new thread than a process

- Less time to terminate a thread than a process

- Switching between two threads takes less time than switching between processes

- Threads enhance efficiency in communication between programs

**Each Thread have:**

-an execution state (Running, Ready, etc.

- saved thread context when not running

-an execution stack

-access to the memory and resources of its process (all threads of a process share this)

-A dispatchable unit of work.

-Resource ownership belongs to processes

- Scheduling (or dispatching) and Execution context belong to threads.

- **the unit of dispatching** is usually referred to as a thread or lightweight process

- t**he unit of resource ownership** is usually referred to as a process or task.

- **Multithreading** refers to the ability of an OS to support multiple, concurrent paths of execution within a single process.

- all of the threads of a process share the state and resources of that process

- When one thread alters an item of data in memory, other threads see the results if and when they access that item

-They reside in the same address space and have access to the same data.

- If one thread opens a file with read privileges, other threads in the same process can also read from that file.

-**threads share** : Memory Regions, address space, resource of process, global variable, heap, static data

-**Thread don’t share**: instruction pointer, stack pointer, register, program counter.

- **What are threads and what do they consist of**: A thread is a basic unit of CPU utilization; it consists of:  
- its own copy of registers and stack space  
- its own program counter  
- a thread ID

-**When can threads be implemented**: Threads can be implemented at user or kernel level.

**-How are user level threads implemented:** User level threads are implemented in user-level libraries, rather than via system calls, so thread switching does not  
need to call the operating system and generate an interrupt; kernel has no knowledge of user-level threads.

- **What are the advantages and disadvantages of user level threads**: Advantage: no kernel involvement in switching; fast  
Disadvantage: since the kernel is not aware of user-level threads, if one user level thread blocks when making a system call, all user level threads in the same task may block too.

**Thread Control Block** (**TCB**) is a data structure in the operating system kernel which contains [thread-specific (Links to an external site.)Links to an external site.](https://en.wikipedia.org/wiki/Thread_(computing)) information needed to manage it.

-Thread Identifier: Unique id (tid) is assigned to every new thread

-Stack pointer: Points to thread's stack in the process

-Program counter

-State of the thread (running, ready, waiting, start, done)

-Thread's register values

-Pointer to the Process control block (PCB) of the process that the thread lives on

The Thread Control Block acts as a library of information about the threads in a system. Specific information is stored in the thread control block highlighting important information about each process

**There are four basic thread operations associated with a change in thread state:**

**-Spawn:** Typically, when a new process is spawned, a thread for that process is also spawned. Subsequently, a thread within a process may spawn another thread within the same process, providing an instruction pointer and arguments for the new thread. The new thread is provided with its own register context and stack space and placed on the ready queue.

**-Block**: When a thread needs to wait for an event, it will block (saving its user registers, program counter, and stack pointers). The processor may now turn to the execution of another ready thread in the same or a different process.

**-Unblock**: When the event for which a thread is blocked occurs, the thread is moved to the Ready queue.

**-Finish**: When a thread completes, its register context and stacks are deallocated.

**-**If CPU Scheduling by the OS is at the thread level, there is no blocking

**-** All of the threads of a process share the same address space and other resources, such as open files

**-** It is, therefore, necessary to synchronize the activities of the various threads so that they do not interfere with each other or corrupt data structures.

- For example, if two threads each try to add an element to a doubly linked list at the same time, one element may be lost, or the list may end up malformed. The issues raised, and the techniques used in the synchronization of threads are, in general, the same as for the synchronization of processes.

**Thread synchronization:**

-It is necessary to synchronize the activities of the various threads

-all threads of a process share the same address space and other resources

-any alteration of a resource by one thread affects the other threads in the same process

Thread-join is issued by one thread when it

wants to stop and wait for the termination of

another thread.

**Week** 5: Piping

**-file descriptor** (termed FD, less frequently fildes) is an abstract indicator (handle) used to access a file or other input/output resource, such as a pipe or network socket

-A file descriptor is a non-negative integer, generally represented in the C programming language as the type int (negative values being reserved to indicate "no value" or an error condition

-A file descriptor refers not just to a file, but the process's current "context" for the file.

A user can open a file for access using a function such as open().

-When a user opens a file, the kernel creates the data structure and entry in the file table and makes an entry in a file descriptor table associated with the process. The files descriptor is a 'pointer' into the file table managed by the kernel.

-**A Unix pipe** is a kernel buffer with 2 file descriptors • One for writing; one for reading • Data is read in exactly the same sequence as was written

-The first integer in the array (fd[0] element 0) is set up and opened for reading, while the second integer (fd[1] element 1) is set up and opened for writing. All data traveling through the pipe moves through the kernel.

-If the parent wants to (just) receive data from the child, it should close fd1 (it’s output), and the child should close fd0 (it’s input).

-If the parent wants to (just) send data to the child, it should close fd0 (it’s input), and the child should close fd1 (it’s output).

**the dup() system call**: duplicates an original open file descriptor.

PROTOTYPE: int dup( int oldfd ); RETURNS: new descriptor on success NOTES: the old descriptor is not closed! Both may be used interchangeably. We will typically close one of the standard streams first. The dup() system call uses the **lowest- numbered, unused descriptor** when creating a new one

**dup2(),** is **now required** by the POSIX standard

int dup2( int oldfd, int newfd );

NOTES: the old descriptor is closed with dup2()!

In addition, it is guaranteed to be **atomic**, which essentially means that it will never be interrupted by an arriving signal.

week 6: networking programming

-A server manages some resource, and it provides some service for its clients by manipulating that resource.

A client-server transaction consists of four steps:

-When a client needs service, it initiates a transaction by sending a request to the server. For example, when a Web browser needs a file, it sends a request to a Web server.

-The server receives the request, interprets it, and manipulates its resources in the appropriate way. For example, when a Web server receives a request from a browser, it reads a disk file.

-The server sends a response to the client and then waits for the next request. For example, a Web server sends the file back to a client.

-The client receives the response and manipulates it. For example, after a Web browser receives a page from the server, it displays it on the screen

What Does an internet Protocol Do

Provides a *naming scheme* ,**Delivery mechanism**

The Hosts are mapped to a set of 32-bit *IP addresses*

set of IP addresses is mapped to a set of identifiers called Internet *domain names*

week 6:

type of thread: ULT & KLT

**ULT**:

-All thread management is done by the application

-The kernel is not aware of the existence of threads

**Advantage**: Thread switching does not require kernel mode privileges

**Disadvantage**:

-In a typical OS many system calls are blocking .as a result, when a ULT executes a system call, not only is that thread blocked, but all of the threads within the process are blocked

-In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing

KLT:

- Thread management is done by the kernel, no thread management is done by application

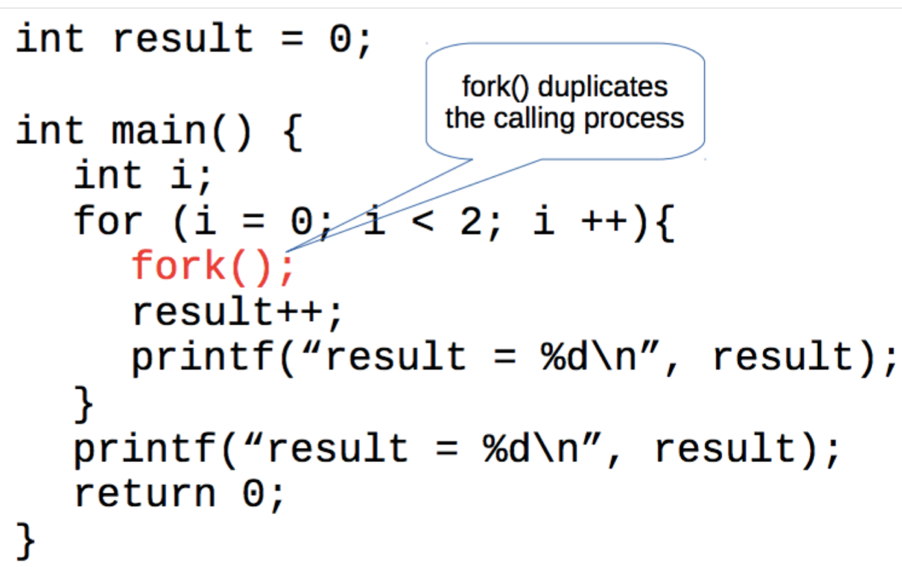
**Advantage:**

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors

-If one thread in a process is blocked, the kernel can schedule another thread of the same process

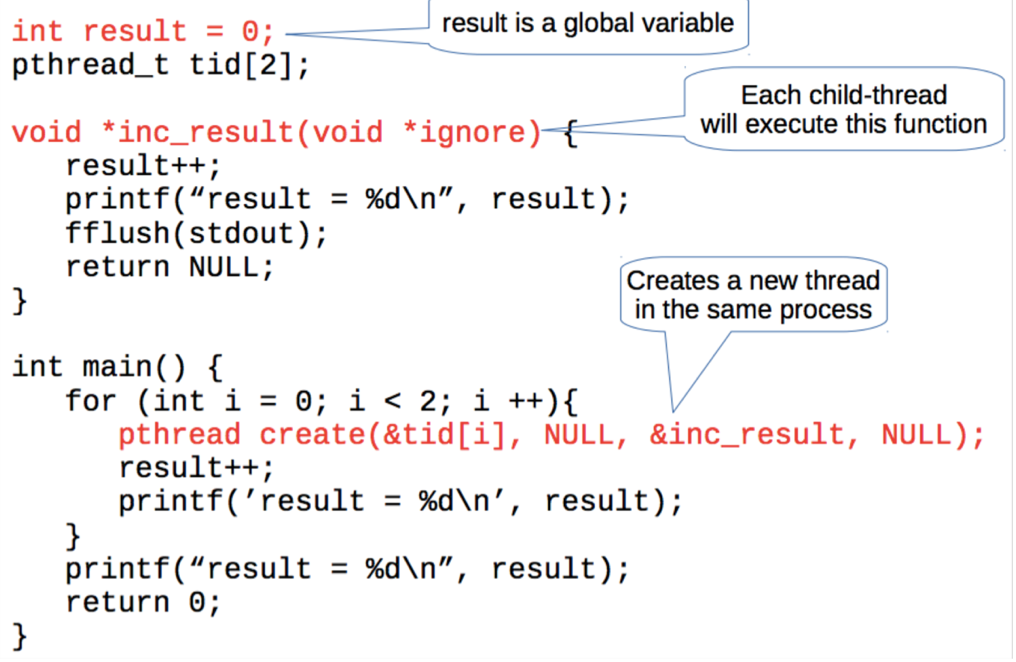
**Disadvantage:**

-The transfer of control from one thread to another within the same process requires a mode switch to the kernel

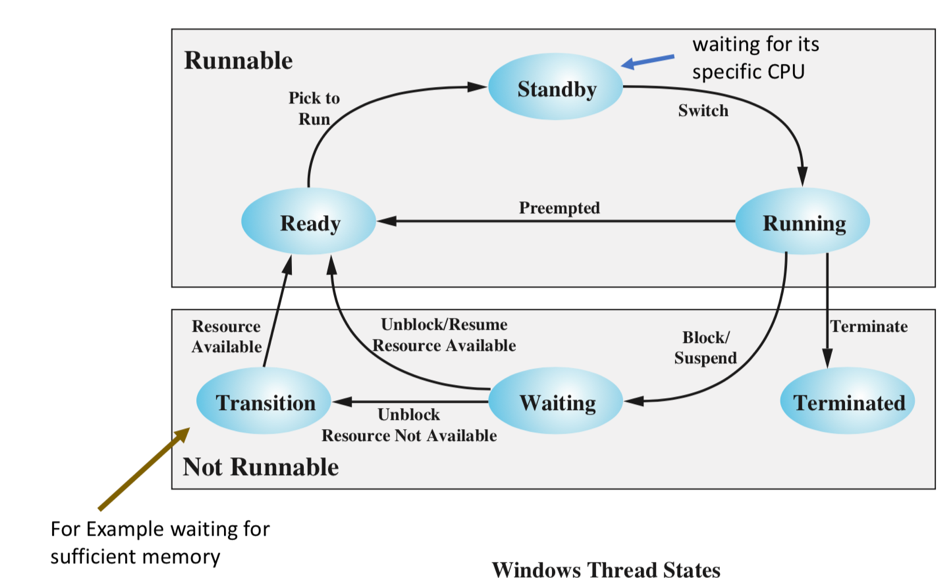


-4 process

Threads are created with respect to the resources of the parent process. Threads share the parent’s variables and have their own execution context.



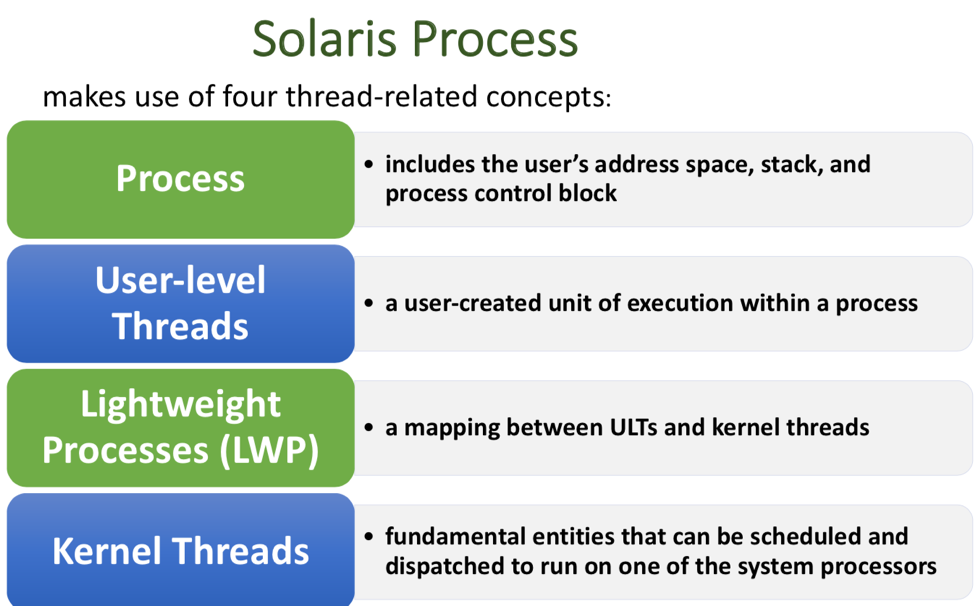
-3 thread



**Standby**: A standby thread has been selected to run next on a particular processor. The thread waits in this state until that processor is made available.

-If the standby thread’s priority is high enough, the running thread on that processor may be preempted in favor of the standby thread. Otherwise, the standby thread waits until the running thread blocks or exhausts its time slice.

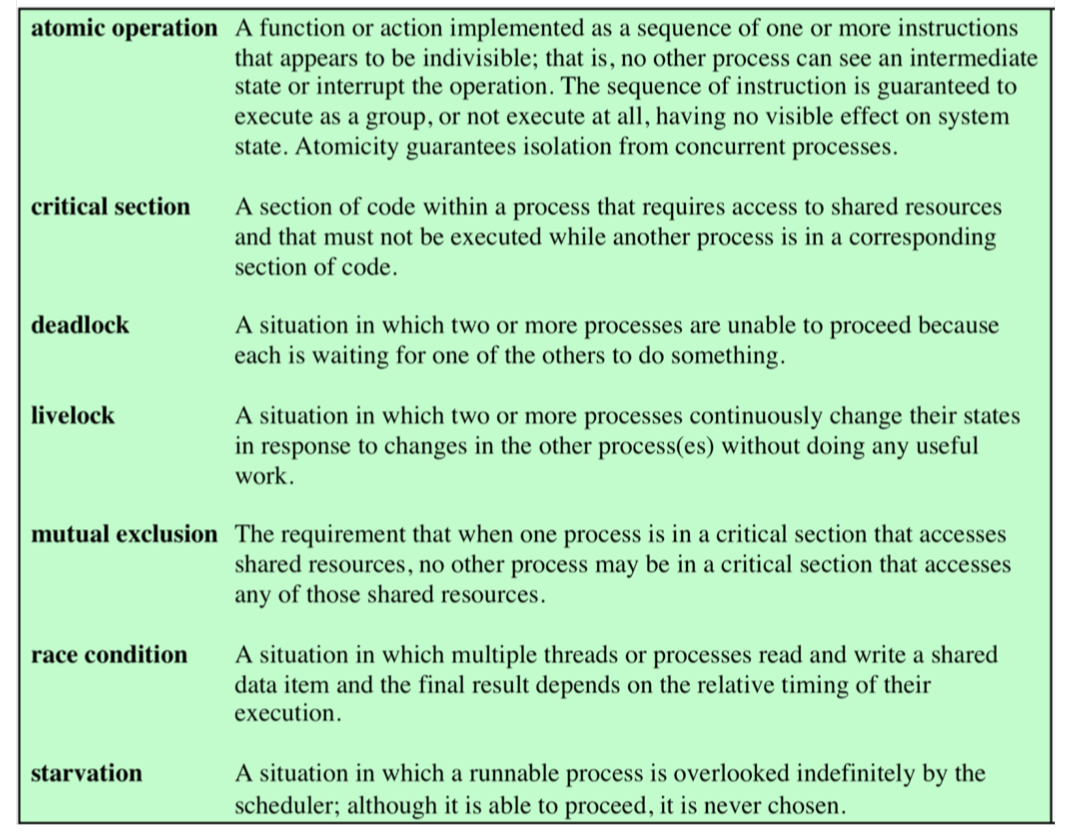
**Transition**: like Ready/Suspended



**Race condition:**

-Can occur when multiple processes or threads read and write shared data items

-the “loser” of the race is the process that updates last and will determine the final value of the variable



week 7: Synchronnization

-**Fine-grained atomic** actions are defined as machine instructions. We know that they go to completion once started because interrupts are serviced at the end of an instruction. non- interruptable.

-Coarse-grained Atomic Actions :

-A sequence of fine-grained atomic actions that appear to execute indivisibly. Need a synchronization mechanism to construct a “coarse- grained” atomic action

Achieve mutual exclusion:

-disable interrupt, lock variable, strict alternation, TSL, Peterson algor.

-The tie-breaker algorithm(Peterson algor.) is a busy waiting algorithm that uses a variable (the tie-breaker) to determine which of 2 processes attempting to enter a shared region, will enter, locking the other from entering, thus creating the mutual exclusion

-Busy waiting:

Is a form of Synchronization in which a process repeatedly checks a condition until it becomes TRUE

-The Spin-Lock is a busy waiting loop that includes a TSL or swap instruction

-We will use the spinlock to help implement the semaphore operations in user mode.

week 9: semaphore

-As discussed, P() and V() are functions that can be 'called' or invoked by programs

-How are these functions implemented? The issue is that the functions each need to be atomic. Once P() or V() is entered, all of the logic and potential modification of the semaphore data structure must be completed without "functional interruption"

-With only one producer there is no need to worry about mutual exclusion

-Note that the semaphores are used not only for signaling, but also for resource counting

Role of Synchronization

Mutual Exclusion:

Condition Synchronization: concerned with delaying a process until the state is conducive to further execution (change) - safe

Uniprocessors – could disable interrupts

Atomic = non-interruptable

week 10 Condition variable

-This problem where we have a critical section created with mutual exclusion, but also need synchronization, occurs often. A solution is a condition variable.

A Condition Variable (CV) is

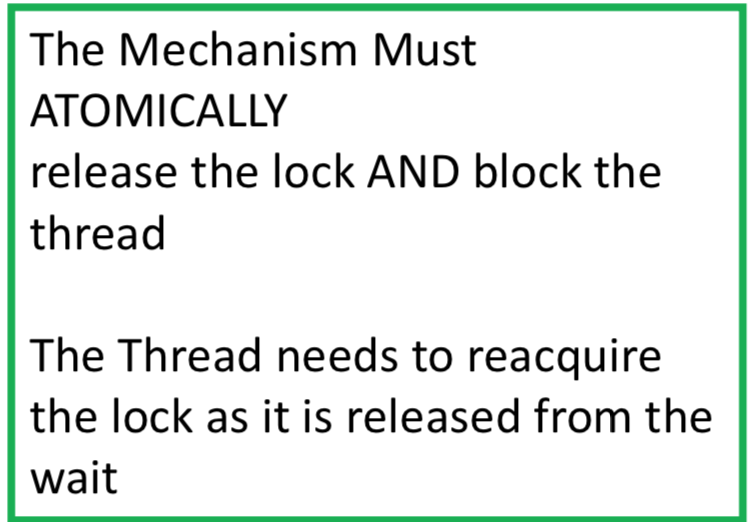
- a synchronization object that lets a thread efficiently wait for a change to shared state protected by a lock.

-CV has block/wakeup features

-Since the shared state is protected by a lock, the CV is used while 'holding' the lock

-called only when holding the lock

-Condition variable is synchronization FOR shared state



There are three functions that can operate on a condition variable:

-wait(): atomically release lock and relinquish processor until signaled

-signal(): wake up a waiter, if any

-broadcast(): wake up all waiters, if any

The CV is memoryless; i.e., a signal with no waiters, has no effect.

A wait() always blocks. The release of the block can only result from a subsequent (future) signal().

-For example, when a thread is holding a lock and issues a wait(), the lock must be freed and the thread placed on a wait queue. These operations must be atomic - atomically (release the lock and block the thread).

A signal operation must release a thread that previously issued a wait on that CV, and the thread that is released must reacquire the lock before it proceeds.

The signal operation releases the waiting thread from the blocked queue and then the released thread must reacquire the lock before completing the return from the wait

-The broadcast function releases all waiting threads from the blocked queue and each must reacquire the lock before they can return from their wait.

-A thread that is currently holding the lock, issues a signal on the condition variable. The thread is released from the wait queue and must reacquire the lock to complete the return from the wait().If the scheduler chooses the thread in the lock::acquire to run before the thread released from the wait queue, then the lock::acquire thread will hold the lock and be able to change the shared state.

-If this thread expects the shared state to be the same as when it was signaled, it may be wrong! The state could have been changed by the thread that previously accessed the shared state.

We address this problem by placing the CV wait in a loop checking for the change in shared state:

while (state\_condition\_to\_wait for)  
             condition.Wait(lock);

Thus, when the thread returns from the condition.Wait, it checks the shared state to make sure the state is as expected. If not, go back and wait on the condition.

SharedObject:: someMethodThatWaits() {  
                         Lock.acquire();  
                               //shared state access HERE  
  
                         While (!testOnSharedState()) {  
                                   cv.wait(&lock);  
                          }

                              //shared state access HERE  
                         Lock.release();  
}

**The waiting thread would be released resulting from a thread executing:**

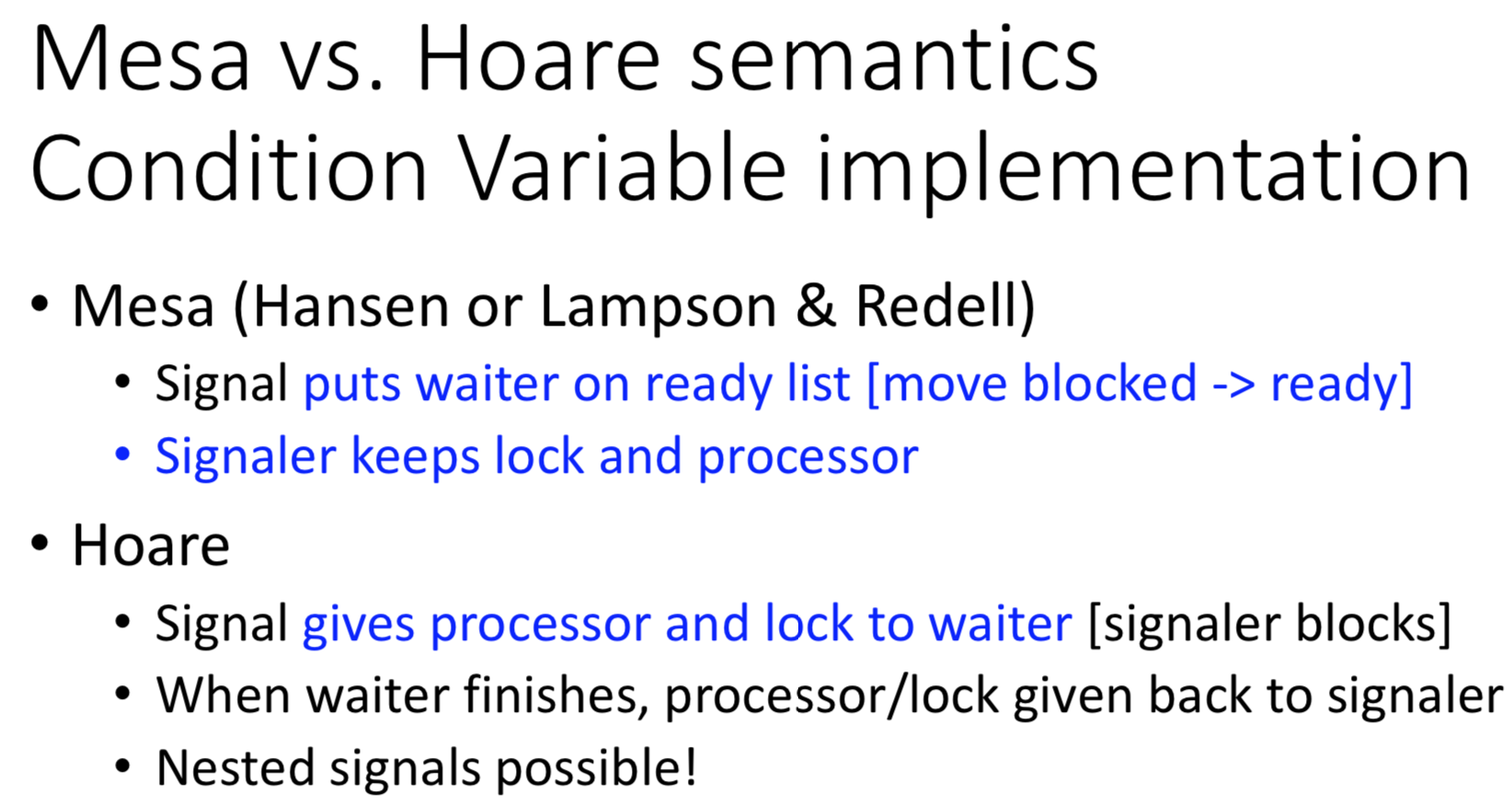
**SharedObject:: someMethodThatSignals() {  
                         Lock.acquire();  
                                 //shared state access HERE  
                                 //Check if state change causes notify another task**

**cv.signal()  
                         Lock.release();  
}**

-One of the implied actions is that the release of a thread from a CV wait and the release of the lock are separate events and that a released thread must reacquire the lock to continue executing. Thus, we do not 'throw the lock' to another thread. This means that another thread can just assume the mutual exclusion that was being held by another thread

-A thread must explicitly release mutual exclusion and another thread must explicitly acquire mutual exclusion. In programming practice, we should always follow this rule.

The implementation of the mutual exclusion of the monitor functions and the implementation of the condition variables follows the Hoare semantics



Mesa semantics encourage the use of condition variables within a loop that tests the safety of the state that the application waits on. Good programming practice with condition variables includes this testing.

Week 11: condition variable with barrier

-A Barrier is a synchronization mechanism that is very useful when we have many threads working to solve a problem in stages.

A barrier synchronizes the arrival of all threads at a particular state, blocking threads as they arrive and then releasing all once the last thread arrives.

-Synchronization mechanism where groups of processes work together in phases. Processes go to the next phase only when all are at the end of particular phase.

When the last arrives, a thread is awakened to move through the barrier. The awakened thread awakens another waiter. This is a 'turnstile' effect, finally releasing all threads.

Week 11: Memory Management

-Memory management is one of the fundamental features and responsibilities of an OS

-Each program must reside in memory in order to execute.

-The CPU fetches and executes instructions from main memory and CPU registers (data elements).

-The way memory is managed is somewhat dependent on the way programs are built (compiled and linked) and loaded into memory to be executed.

-this means that we have more active and running programs than will fit in memory. So, some programs at times will be swapped to disk and then later brought back to memory

-a program that leaves memory may be brought back to a different memory region.

-Also, a program may be brought into memory to a different area each time it is executed.

-When an object module is created by the compiler, the addresses in the module are determined s space is allocated to the code by the compiler.

-The linker combines object modules producing a memory image. In doing so, the object modules are allocated space in the resulting image. Addresses in the modules must again be relocated to their new location in the image. Without some additional mechanisms for changing addresses, it would not be possible to have programs loaded and executed in different memory regions. This is because the address references are fixed and if the program is moved, the CPU will still reference the same addresses specified in the original program image.

--Program must be brought (from disk) into memory and placed within a process for it to be run

-Main memory and registers are the only storage the CPU can access directly

-**Cache** sits between main memory and CPU registers

Simple multiple organization base and limit

-A **base** and **limit** register pair define the logical address space of a (**contiguous**) process

-If multiple processes occupy physical memory and are contiguous, we need to define their address space

-**RECALL: A process Executes using Virtual Addresses**

-When a program is loaded into memory, this register is loaded with the lowest memory address the program occupies. This value is stored in the PCB for the process. When the program is selected to run, the PC is loaded from PCB and the relocation register is loaded from the PCB.

Binding of Instructions and Data to Memory

-Address binding of instructions and data to memory addresses can happen at three different stages

**Compile time**: static binding &allocation

**Load time :** static binding &allocation

**Execution time :** dynamic binding &allocation

Base and limit address

-Base and limit registers can be used to give each process a separate address space

- Addresses referenced in the program (virtual addresses) are offset by the contents of the base register.

Logical vs. Physical Address Space

**Logical address** – generated by the CPU; also referred to as **virtual address (or, our view of our program)**

**Physical address** – address produced by the memory unit

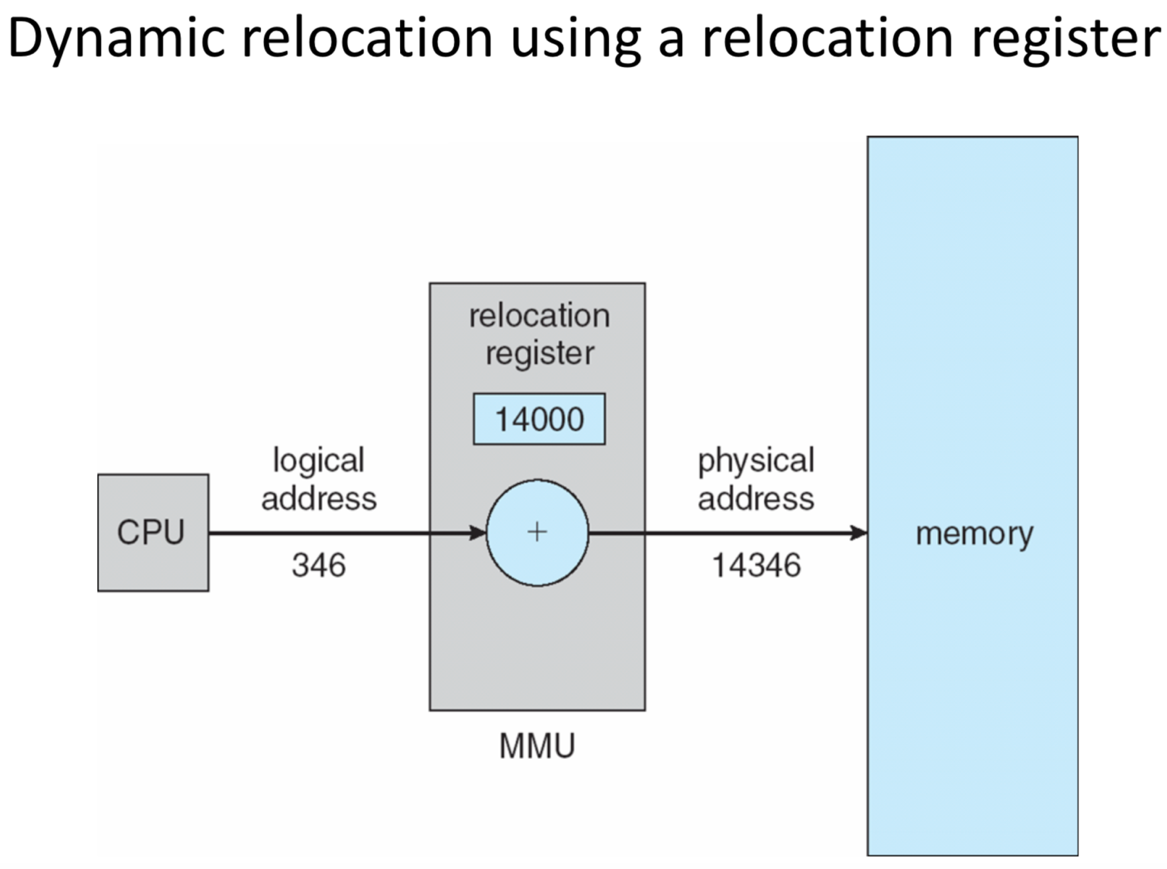
-Logical and physical addresses are the same in compile-time and load-time address-binding schemes

-logical (virtual) and physical addresses differ in execution-time address-binding scheme

*Logical address:* address of an instruction or data byte as used in a process by the CPU

Can Be Viewed as a pair (*compi*, *bytei*)

*Physical address:* address in memory where an instruction or data byte exists



The maximum size of the program when it is allocated space in memory is the value used in the limit register.

**Hole** – block of available memory; holes of various size are scattered throughout memory

Week 12: Non-Contiguous Memory and Paging

- reduce the OS overhead in memory management, we consider memory allocation for programs to be non-contiguous.

logical address:

-traditionally, this is viewed as an integer in the range of 0 to 2 k -1,where the size of the logical or program space is 2k

- Another view is to break the address space into blocks. Then an address of a byte is specified by (blockLi, bytej). Each block is composed of contiguous bytes.

- **MMU = memory-management unit** - a hardware device that maps virtual addresses to physical addresses

- **Relocation register** = base register for MMU.

-If a block of physical memory is the same size as a block of logical memory storing the program block (logical) in the physical memory means that addresses within the block are the same, whether physical or logical.

- Placing logical blocks in physical memory blocks means that we need to remember the mapping of the logical block number to the physical block number.

**Paging**

-involves breaking the physical memory space into equal size blocks of contiguous bytes.

- The block size is a power of 2, generally 512, 1024, 2048 or 4196.

- These physical blocks are called frames

- Programs are broken into blocks of equal size called pages.

- In paging, the size of the pages and the size of the frames are equal. This means that any page can be stored in any frame.

- Since the within page byte address is the same as the within frame byte address, address translation of a virtual address to a physical address only requires mapping the page to the frame in which it is store.

-To run a program of size *n* pages, need to find *n* free frames and load program

-Set up a *page table* to translate logical to physical addresses

week 13: memory management

-**Hashed Page Tables** are another way to organize paging while trying to reduce the memory required for a page table.

-In this implementation, the page number is removed from the virtual address and used as the input to a hashing function.

-The output would be the hash table entry containing the corresponding frame.

-Hash tables are used in some systems where the address space is greater than 32 bits.

**An inverted page table**

-means that we have a table where the entries are page numbers rather than frame numbers.

-The index used to address the table is a frame number rather than a page number.

-There is only one frame table and entries identify the contents of each frame (PID + Page number)

-This approach can eliminate the need for a page table for each process, thus considerably reducing the memory overhead for paging.

-The downside is that in order to translate from page number to frame number, the frame table must be searched for a match of the PID and page number

**Page table entries**

-contain more than just bits encoding the frame number.

**Segmentation**

-is another implementation of memory management where a program is separated in to blocks of contiguous addresses.