**Lec. 1**

Computer Do

-Runs Programs

What is a program?

-A set of instructions

How does computer run program?

-Processer fetches an instruction from memory

-Decode the instruction

-Executes the instruction

-Move to the next Instruction

How do you share processors, memory, and hardware devices

across **multiple programs**?

-Need to protect applications from each other

-Provide fair, efficient access to resources

What is an Operating System?

-Software that converts hardware into a useful form for applications

-Software that acts as a resource manager

OS as resource manager

-Manages resources such as CPU, memory, and disk

-Allow many programs to run (Sharing the CPU) why?

-Many programs to concurrently access their instructions and data.

-Many programs to access devices (Sharing disks/storage)

**OS in three easy pieces**

-**virtualization**: make instance of a running program (i.e., a process) believe it has each resource to itself.

-**concurrency**: Provide processes with abstractions for managing concurrency Threads with

shared memory using locks, semaphores, condition variables to control access

-**persistence**: Ensure lifetime of information extends beyond lifetime of process, even

in the presence of failures (Make easy to write appli. , so that can use the same program for many types of storage

**Virtualizing Memory**

-The physical memory is an array of bytes

-A program keeps all of its data structures in memory.

\*Read memory (load): Specify an address to be able to access the data

\*Write memory (store): Specify the data to be written to the given address

-Each process accesses its own private virtual address space

-The OS maps virtual address space onto the physical memory.

-A memory reference within one running program does not affect the

address space of other processes.

-Physical memory is a shared resource, managed by the OS

**Persistence**

-Devices such as DRAM store values in a volatile(liable to change rapidly)

-Hardware and software are needed to store data persistently

\*Hardware: I/O device such as a hard drive, solid-state drives(SSDs)

\*Software: File system manages the disk. Responsible for storing any files the user creates

**What does OS do in order to write to disk?**

-Figure out where on disk this new data will reside

-Issue I/O requests to the underlying storage device

File system handles system crashes during write

\*Journaling or copy-on-write

\*Carefully ordering writes to disk

**Why Use Processes ?**

-Support principle of divide and conquer (Decompose large problem into smaller ones)

-Express concurrency (Systems have many concurrent jobs)

-Support isolated execution (Processes operate in isolation)

**Process Address Space**

Address Space (AS): all the memory that a process can address( Really large memory to use)

-Each time process created, OS creates an independent AS

-Process isolation: one process can’t access another’s AS

**Thread:**

-separate streams of executions that share an address space

\*Allow one process to have multiple points of execution, can use

multiple CPUs

\*Duplicates only minimum information needed for independent

execution of multiple concurrent threads

**OS Design Goals**

**-** Build appropriate abstractions (Make the system convenient and easy to use)

**-** Provide high performance (Minimize the overhead of the OS)

-Protection between applications (Isolation: Bad behavior of one does not harm other or OS Itself!

-Reliability

-Security

**Lec. 2**

**CPU Virtulization**

What are some key things that an OS provides ?

-Manages resources such as CPU, memory, and disk

-Operating system can create the illusion that many CPUs exist

-Time sharing ( run one, stop it, run another process)

-Potential performance cost

**A process**: is running program, set of program instructions & execution state

What state is important?

-register (program counter/instruction pointer) , memory (address space) isolated

**Process vs. Program**

Process: Dynamic instance of code and data

Program: Static code and static data

**Process creation**

-Load a program code into memory into the address space of the process

-The program’s run-time **stack** (local var., func. para. , return add.) is allocated

-The program’s heap is created

-The OS does some other initialization tasks (Input/Output(I/O) setup

-Start the program running at the entry point -namely, main()

**Process States**

-Running, ready, blocked.

**Lec. 3**

**How does the operating system create the illusion that many CPUs exist that together run many processes?**

-Create many virtual CPUs(process)

- Time share the physical CPU

\*Switch a virtual CPU (process) onto the physical CPU to execute its code

\*When a process executes, its (virtual) addresses must be mapped or

associated with, real/physical addresses in memory

**how would the OS protect access to restricted resources?**

- Protected Control Transfer : Operating system carefully exposes protected functionality using

privilege modes

**OS allows user process to temporarily enter kernel mode. When/ how ?**

- program can include System Calls

**System Calls:** planned program to kernel transfer of control

-implemented as a **trap instruction (**Mechanism to realize system call) that transfers control to the operating system

-Execution of a system call instruction changes mode from user to kernel

-Completion of system call execution changes mode back to user

**How to use SYSCALL system services**

- Load service number in register $v0.

-Load argument values, if any, in $a0, $a1, $a2, or $f12 as specified.

-Issue the SYSCALL instruction.

-Retrieve return values, if any, from result registers as specified.

**Where does OS live?**

- lives in the same address space as process

**How does the OS regain control of the CPU?**

-Cooperative Approach: wait for system calls

-Non-Cooperative Approach: OS takes control

\***Interrupts**: Asynchronous events usually produced by I/O devices

\***Exceptions**: Synchronous events usually associated with software requesting

\* Traps: Synchronous events produced by special instructions typically system call

**How to Handle Multiple Interrupts ?**

-Two ways OS can be designed to handle:

\*Disable interrupts during interrupt processing( Sequential processing)

\*Use nested interrupts

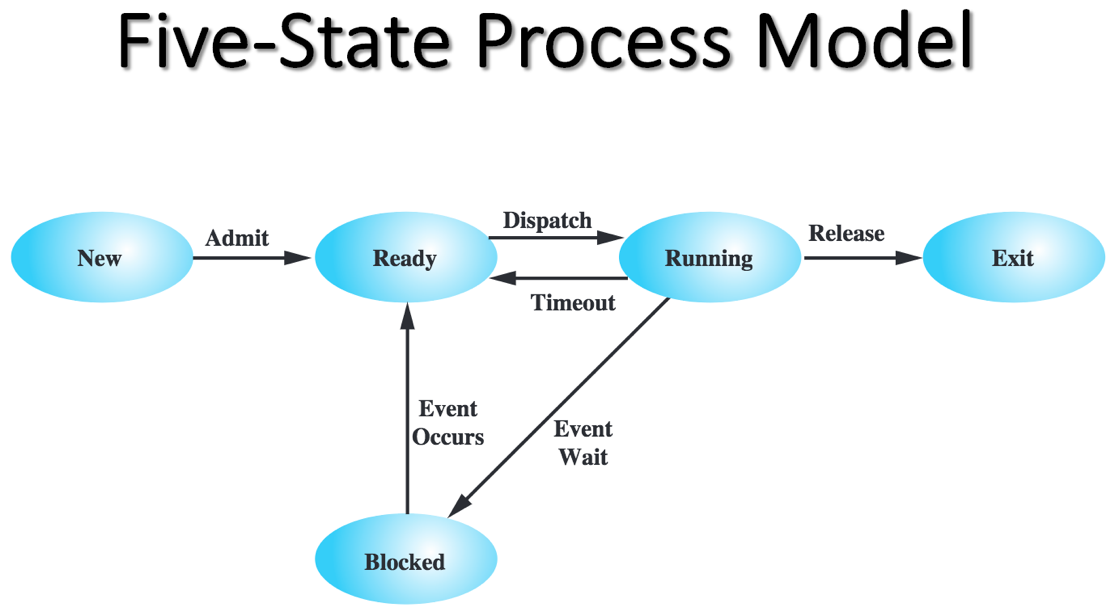
**lec. 4**

**Basic Instruction Cycle**

-Start, fetches instruction, execute instruction, halt

**Suspended Processes**

-swapping



**Lec. 5**

Virtualizing the CPU

**Process Creation, Management,& Scheduling**

**Registers**

Program Counter (PC): address of instruction

Instruction Register (IR): instruction being executed

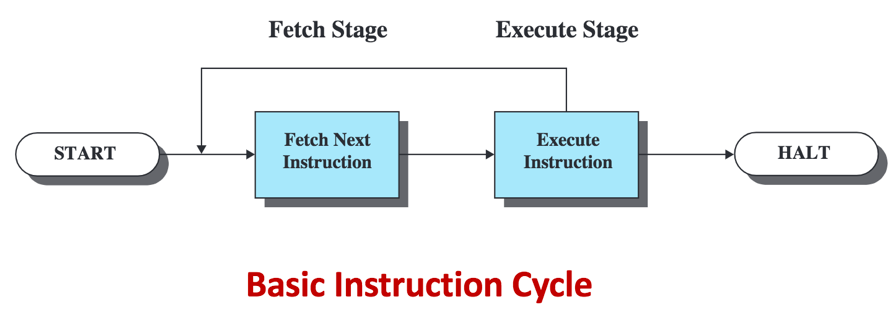
Memory address register (MAR): specified address for next read/write

Memory buffer register (MBR): data to be written to/receives data from mem

I/O address register (I/O AR): specifies an I/O device

I/O buffer register (I/O BR): used to exchange data between I/O and CPU

Accumulator (AC): Temporary Storage



**Instruction Fetch and Execute**

-The processor fetches an instruction from memory

\*Typically the program counter (PC) holds the address of the next instruction to be fetched

\*PC is incremented after each fetch

**Instruction Register (IR)**

-Fetched instruction loaded into Instruction Register (IR)

-Processor interprets the instruction, performs required action:

\*Processor-memory

\*Processor-I/O

\*Data processing

\*Control

**How does the OS control the physical CPU?**

-Cooperative Approach: wait for system calls

-Non-Cooperative Approach: OS takes control

•Interrupts

•Exceptions

**OS Control Mechanism Terminology**

**Traps**: Synchronous events produced by special instructions typically used to allow

secure entry into operating system code

-System calls

**Interrupts:** Asynchronous events usually produced by I/O devices which must be

handled by the processor by interrupting execution of the currently running process

-Disk

**Exceptions**: Synchronous events usually associated with software requesting

something the hardware can’t perform

-illegal addressing, illegal op code, etc

**How to Handle Multiple Interrupts**

Two ways OS can be designed to handle:

-**Disable interrupts** during interrupt processing (Sequential processing)

-Use nested interrupts

**\***Interrupt the interrupt..but only if the interrupter has higher priority!

**Thread vs. process**

Why threads?

-Thread allows running code concurrently within a single process

-Switching among threads is light-weight

-Sharing data among threads requires no inter process communication

Why processes?

Fault isolation: One buggy process cannot crash others

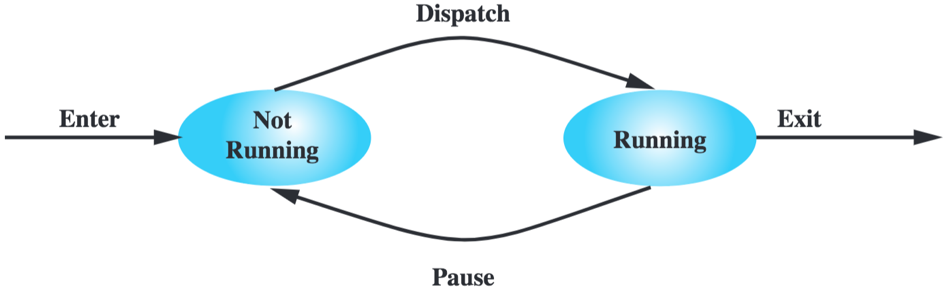
**Threads and Shared Resources**

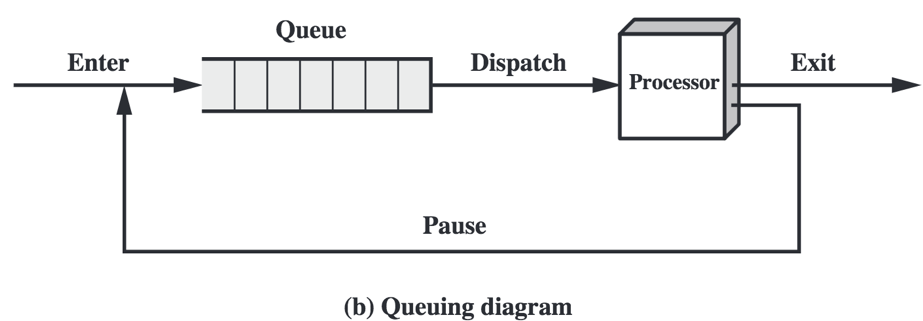
-Changes made by one thread to shared system resources will be seen by all other threads

-Two pointers having the same value point to the same data

-Reading and writing to the same memory locations is possible

**Two-State Process Model**





**Process Creation**

**Process spawning**: When the OS creates a process at the explicit request of another process

**Parent process**: is the original, creating, process

**Child process**: is the new process

**Process Termination**

-There must be a means for a process to indicate its completion

-A batch job should include a HALT instruction or an explicit OS service call for termination

-For an interactive application, the action of the user will indicate when the process is completed (e.g. log off, quitting an application)

**Suspended Processes**

-**Swapping:**

-Involves moving part of all of a process from main memory to disk

-When none of the processes in main memory is in the Ready state, the OS swaps one of the blocked processes out on to disk into a suspend queue

-Swapping, however, is an I/O operation and therefore there is the potential for making the problem worse, not better. Because disk I/O is generally the fastest I/O on a system, swapping will usually enhance performance

**Characteristics of a Suspended Process**

- The process is not immediately available for execution

- The process may or may not be waiting on an event

- The process was placed in a suspended state by an agent: either itself, a parent process, or the OS, for the purpose of preventing its execution

- The process may not be removed from this state until the agent explicitly orders the removal

**Reason for Process Suspension**

-swapping, other OS reason, interactive user request, timing, parent process request.

**Reason for Process Termination**

- Normal completion, Time limit exceeded, Memory unavailable, Bounds violation, Protection error

**Lec. 6**

Processes and thread

**Process Scheduling**

-Process has CPU burst and I/O burst cycle

•When I/O burst, CPU is idle( lazy, avoiding work)

•Want to exploit idle CPU to achieve parallel tasking

**Process States & Transitions**

New: process is being created

Running: instructions are being executed on the processor

Waiting/blocked: process is waiting for some event to occur

Ready: process is waiting to use the processor

Terminated: process has finished execution

**Process Control Block**

-OS maintains a data structure for each process that contains information needed to

pause and restart process execution

**Process Creation example**

- A process can create several other processes

- Each process could get its resources directly from the OS

- New process gets its own private virtual address space in memory

**Process creation step**

-assign unique process identifier to new process

-allocate space for process

-initialize process control block

-set appropriate linkages

-create or expand other data structures

**Using process**

-Create new process using **fork()** system call

fork(): return process ID from parent to child

<0 : fork failed

== 0 : child (new process)

>0 : parent

**How to tell the difference between the original and new process**

-check the return value of fork

**lec. 7**

Thread

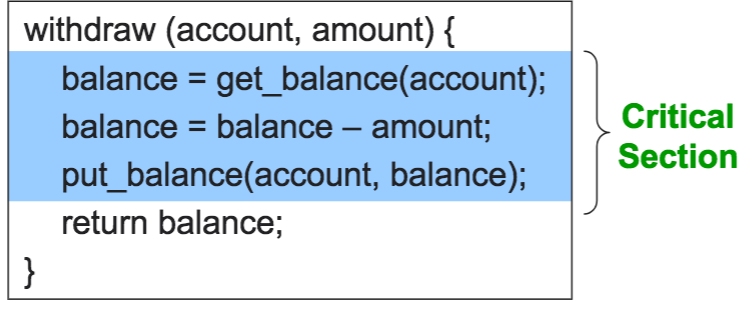
**When one process needs to wait, processor can switch to another,**

-required context switch

**Threads are made to share**

**-**shared: global variable and static object, dynamic object and other heap object

-not shared: local variable



**Mutual exclusion**

-One way to ensure who wins the race is to only let one

Thread “compete” ; this is called **mutual exclusion**

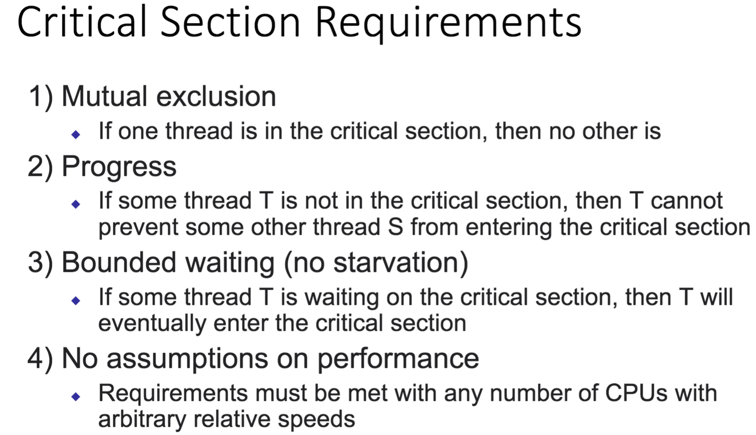
-Code that uses mutual exclusion to synchronize its

execution is called a **critical section**

\*only one thread at a time can execute critical section

\*all other threads are forced to wait on entry

\*when thread leaves critical section, another can enter

****

**Locks**

-one way to implement critical section is to “lock the door”, on the way in, and unlock it again on the way out.

A lock is an object in memory providing two operation

**\*acquire():** before entering critical section

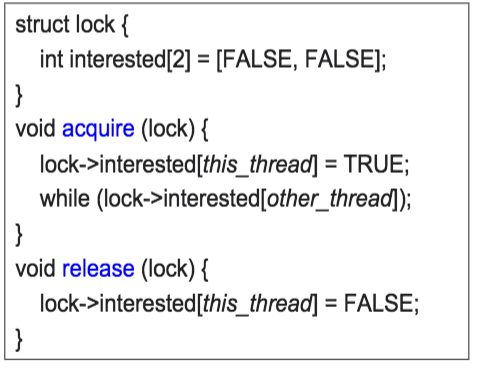
\***release():** after leaving critical section

Thread **pair calls** to acquire() and release()

-between acquire() / release() , the thread hold the lock

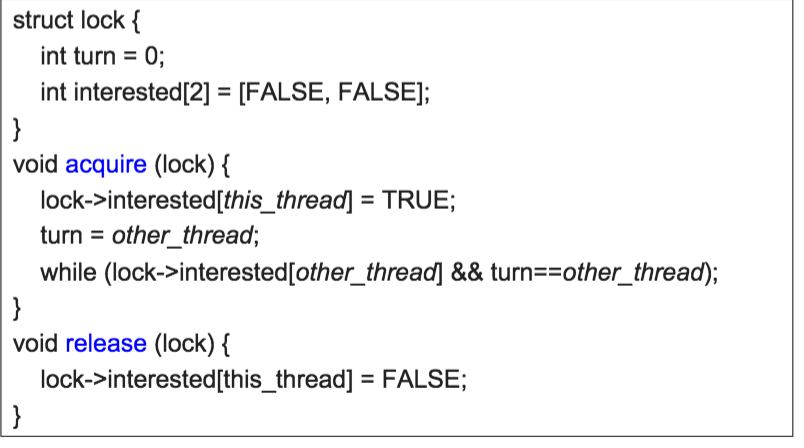
-acquire() does not return until any previous holder releases

**Declaring intent**

****

**-**problem was we didn’t know if other thread was ready(let wait until other thread isn’t interest

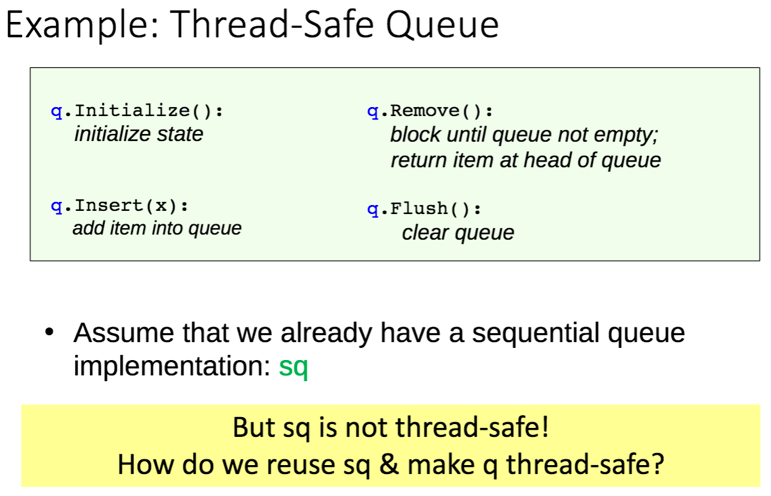
Peterson algorithm

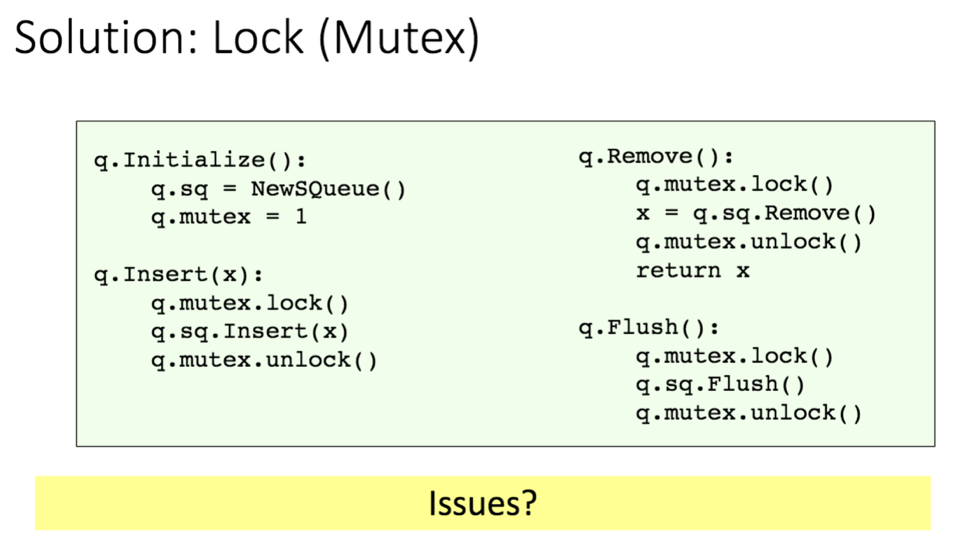


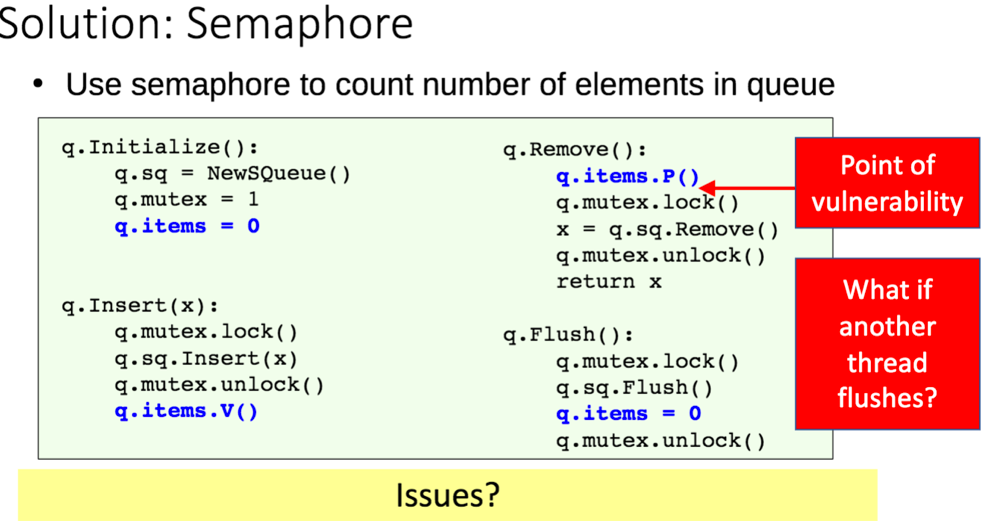
**-**take turns only if somebody else is interested; otherwise begin

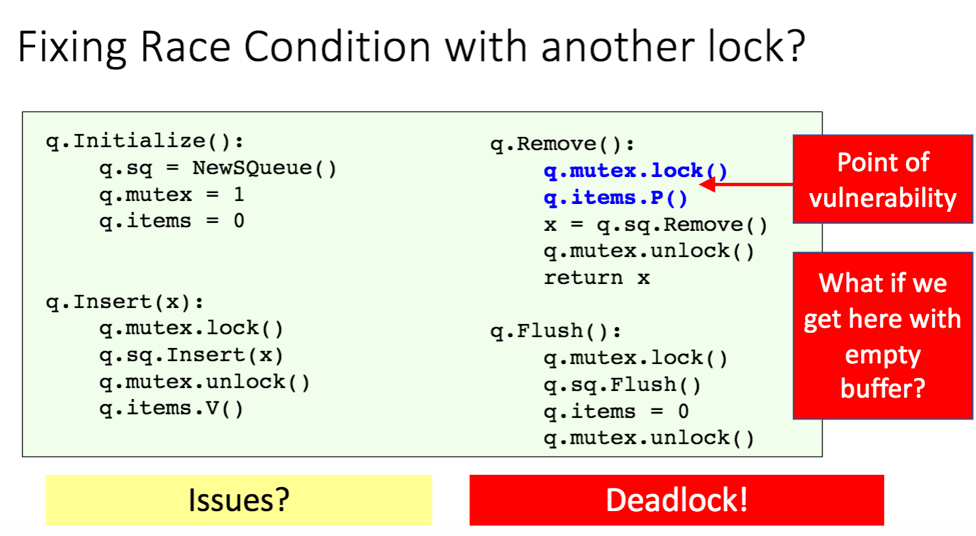
**lec. 8**

Thread Synchronization



****

****



Thread

-new abstraction for single running process

Multi-threaded program

-each thread has its own program counter and set of register

**Midterm exam review**

Class{

Private Boolean lock = true; **//if it’s false, it would guarantee process, guarantee mutual exclusion**

Public void acquire(){

While(testAndSet,true) **// not passed by reference**

}

public void release{

lock = false; //only one with the lock, so no need (testAndSet,false)

**Guarantee process?**

No -first, it’s initialize to true, so no one can get the lock

**Guarantee mutual exclusion?**

No -only one thread can be in lock, so no race condition

**class lock{**

**private int turn =0;**

**private boolen lock[2] = {false,false}**

**public void acquire(int id){**

**lock[id] = true;**

**turn = 1- tid;**

**while( lock[1-id] && turn == (1-id));**

**}**

**public void release(int tid){**

**lock[id] = false;**

**}**

**Guarantee process?**

**-**

**Guarantee mutual exclusion?**

-yes, if it’s turn, and interested, so it get the lock, only one thread can enter the lock.

//waste time

//signal that one lock want its turn

class lock{

public void acquire (){

disableInterrupt();

}

public void release(){

enableInterrupt();  
}

**Guarantee process?**

-yes, because the interrupt affect the context switch

-as long you hold the lock (disable process), no one can access the lock

**Guarantee mutual exclusion?**

-No, because interrupt is disable, the thread can’t enter the lock

for(int i = 0; i < 3; i++){

fork()

print(“hello”)

}

**how many times is hello printed**

-14

i= 0

fork

i= 1 i =1

fork fork

i= 2 i=2

fork fork fork fork

**Semaphore**

-call UP = increment lock

**-lec 11 – slice 21 farthest**

-use of semaphore:

Producer & consumer

p(empty) > 0 , process

p(empty) < / == =, block

-exact call

-different context switch of process/ threat

-state

Midterm

What is an Operating System?

-Software that converts hardware into a useful form for applications

-Software that acts as a resource manager

OS as resource manager

-Manages resources such as CPU, memory, and disk

-Allow many programs to run (Sharing the CPU) why?

-Many programs to concurrently access their instructions and data.

-Many programs to access devices (Sharing disks/storage)

**Process vs. Program**

Process: Dynamic instance of code and data

Program: Static code and static data

**OS in three easy pieces**

-**virtualization**: make instance of a running program (i.e., a process) believe it has each resource to itself.

-**concurrency**: Provide processes with abstractions for managing concurrency Threads with

shared memory using locks, semaphores, condition variables to control access

-**persistence**: Ensure lifetime of information extends beyond lifetime of process, even

in the presence of failures (Make easy to write appli. , so that can use the same program for many types of storage

**Why Use Processes ?**

-Support principle of divide and conquer (Decompose large problem into smaller ones)

-Express concurrency (Systems have many concurrent jobs)

-Support isolated execution (Processes operate in isolation)

**Thread:**

-separate streams of executions that share an address space

\*Allow one process to have multiple points of execution, can use

multiple CPUs

\*Duplicates only minimum information needed for independent

execution of multiple concurrent threads

**Thread vs. process**

Why threads?

-Thread allows running code concurrently within a single process

-Switching among threads is light-weight

-Sharing data among threads requires no inter process communication

Why processes?

Fault isolation: One buggy process cannot crash others

**Process creation**

-Load a program code into memory into the address space of the process

-The program’s run-time **stack** (local var., func. para. , return add.) is allocated

-The program’s heap is created

-The OS does some other initialization tasks (Input/Output(I/O) setup

-Start the program running at the entry point -namely, main()

**Process States**

-Running, ready, blocked.

**How does the operating system create the illusion that many CPUs exist that together run many processes?**

-Create many virtual CPUs(process)

- Time share the physical CPU

\*Switch a virtual CPU (process) onto the physical CPU to execute its code

\*When a process executes, its (virtual) addresses must be mapped or

associated with, real/physical addresses in memory

**how would the OS protect access to restricted resources?**

- Protected Control Transfer : Operating system carefully exposes protected functionality using

privilege modes

**OS allows user process to temporarily enter kernel mode. When/ how ?**

- program can include System Calls

**System Calls:** planned program to kernel transfer of control

-implemented as a **trap instruction (**Mechanism to realize system call) that transfers control to the operating system

-Execution of a system call instruction changes mode from user to kernel

-Completion of system call execution changes mode back to user

**Where does OS live?**

- lives in the same address space as process

**How does the OS regain control of the CPU?**

-Cooperative Approach: wait for system calls

-Non-Cooperative Approach: OS takes control

\***Interrupts**: Asynchronous events usually produced by I/O devices

\***Exceptions**: Synchronous events usually associated with software requesting

\* Traps: Synchronous events produced by special instructions typically system call

**How to Handle Multiple Interrupts ?**

-Two ways OS can be designed to handle:

\*Disable interrupts during interrupt processing( Sequential processing)

\*Use nested interrupts

**Process Creation**

**Process spawning**: When the OS creates a process at the explicit request of another process

**Parent process**: is the original, creating, process

**Child process**: is the new process

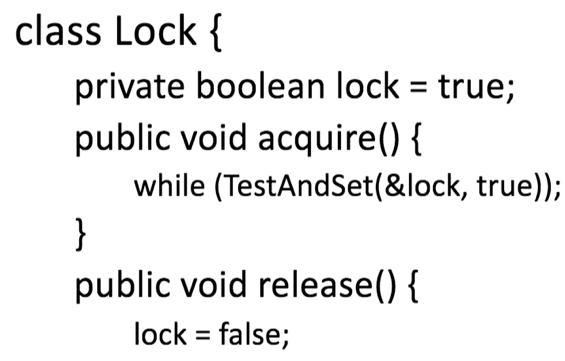
**Suspended Processes**

-**Swapping:**

-Involves moving part of all of a process from main memory to disk

-When none of the processes in main memory is in the Ready state, the OS swaps one of the blocked processes out on to disk into a suspend queue

-Swapping, however, is an I/O operation and therefore there is the potential for making the problem worse, not better. Because disk I/O is generally the fastest I/O on a system, swapping will usually enhance performance



Guarantee mutual exclusion

-Yes, it would not be possible for more than one

process to acquire the lock

guarantee progress

-No, lock initialized to true; will not ever be able to

acquire the lock

**Mutual exclusion**

-One way to ensure who wins the race is to only let one

Thread “compete” ; this is called **mutual exclusion**

-Code that uses mutual exclusion to synchronize its

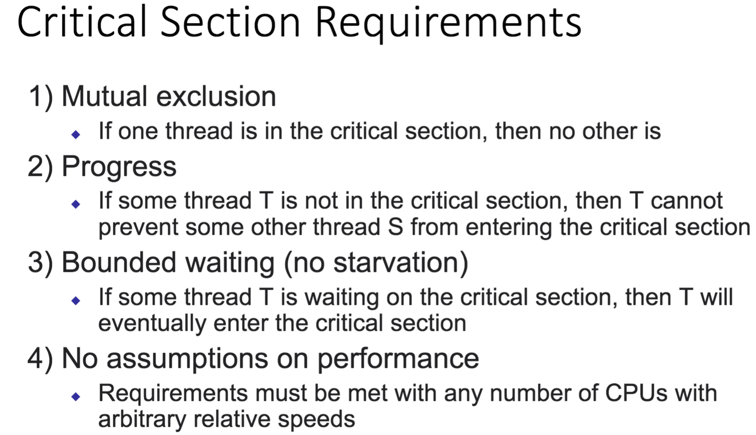
execution is called a **critical section**

-code segment that accesses shared variables and has to be executed as an atomic action. only one process must be executing its **critical section**.

\*only one thread at a time can execute critical section

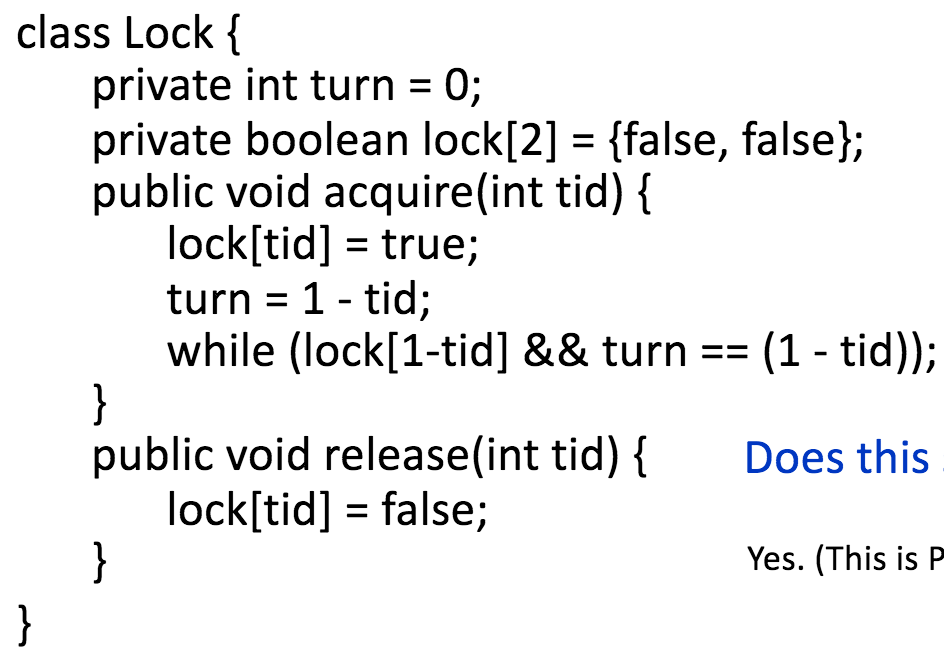
\*all other threads are forced to wait on entry

\*when thread leaves critical section, another can enter

****

**Peterson's algorithm**

-for mutual exclusion that allows two or more processes to share a single-use resource without conflict, using only shared memory for communication.

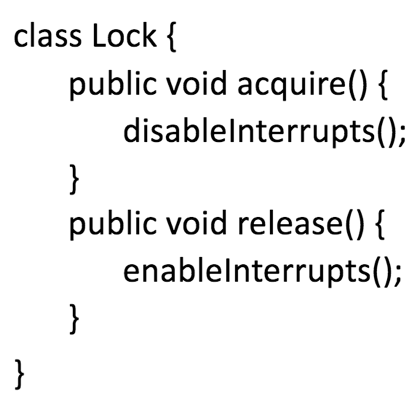


Guarantee mutual exclusion

-yes, Peterson algorithm

guarantee progress

-yes, Peterson algorithm



Guarantee mutual exclusion

- Yes. (On a single processor only, though. On a multiprocessor,

it’s possible that a process that also requires the lock could be

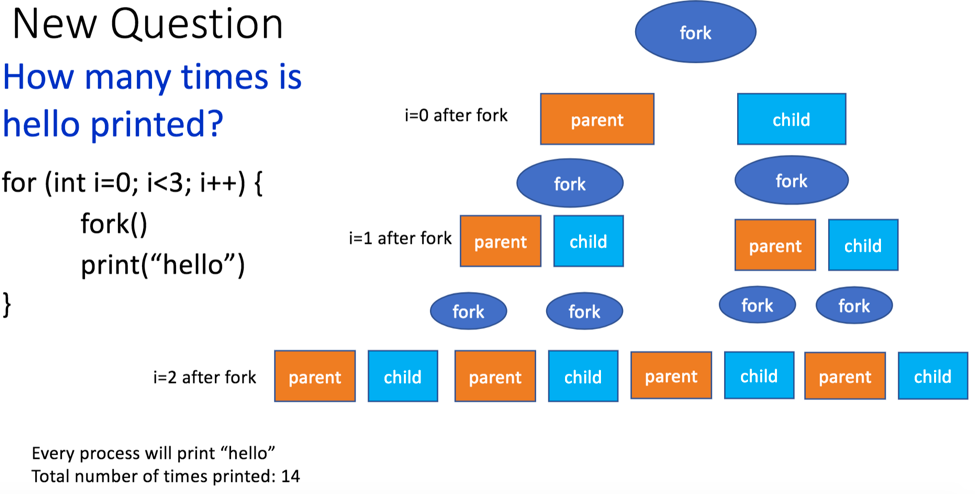
running on another processor.)

guarantee progress

- Yes ,(Assuming that if the process requires I/O, the process just runs

until I/O completes; there is no interrupt signal generated since

the process is not switched off the CPU to wait for I/O)



**Key Abstraction for CPU Virtualization: Process**

**A process** is a running program.

-a process is the set of program instructions & execution state

**What state information is important**?

Registers:

-program counter: Mem addr of next instruction to execute

-instruction register: instruction to execute

-stack pointer: Mem addr of top of call stack

Memory (address space): instructions and data

-Isolated!

**Switching Between Processes**

•Taking one process off the physical CPU and putting another on is called a **context switch**

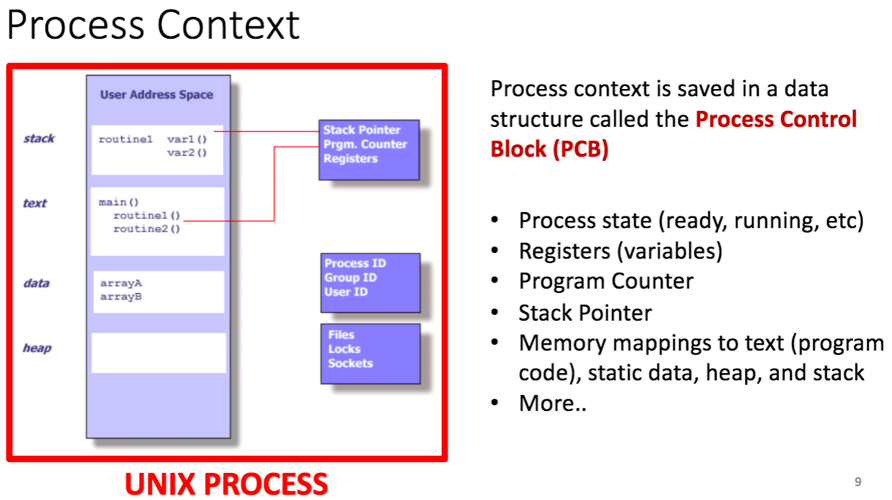
•Must save the context (i.e., all state information) about the process that is being” stopped”

•Must load the context about the process that will run

Process Context

Process context is saved in a data structure called the

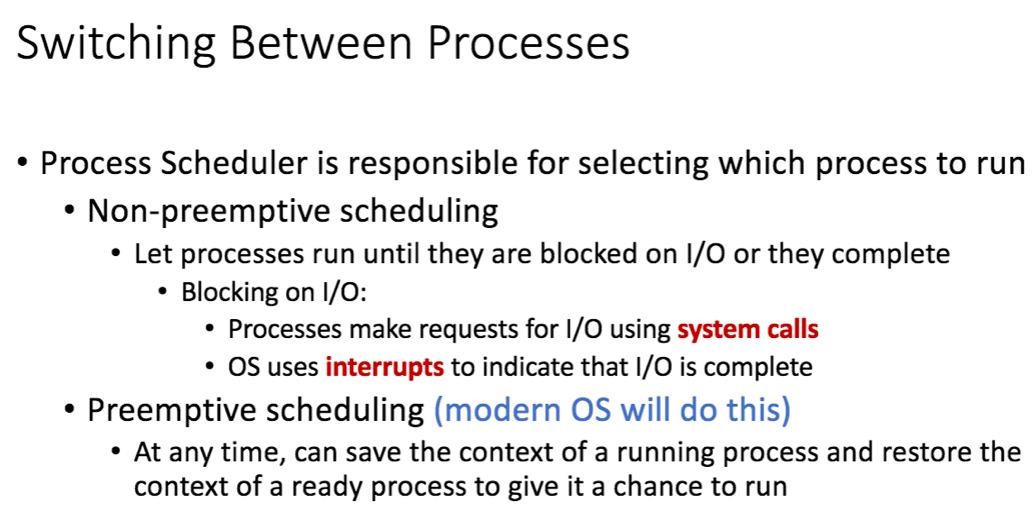
Process Control Block (PCB)



**Threads are made to share**

**-**shared: global variable and static object, dynamic object and other heap object

-not shared: local variable



chapter 3

A computer platform consists of a collection of hardware resources, such as the processor, main memory, I/O modules, timers, and disk drives.

TRUE

For efficiency, applications should be written directly for a given hardware platform.

FALSE

A design change in the structure or semantics of the process control block could affect a number of modules in the OS.

TRUE

The **process control block** is the key tool that enables the OS to support multiple processes and to provide for multiprocessing.

True

It is not the responsibility of the operating system to control the execution of processes.

FALSE

The first step in designing an OS to control processes is to describe the behavior that we would like the processes to exhibit.

TRUE

The OS may create a process on behalf of an application.

TRUE

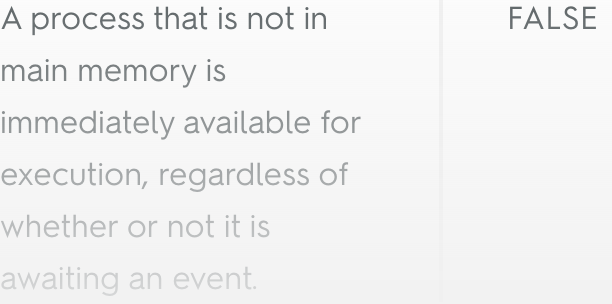
Swapping is not an I/O operation so it will not enhance performance.

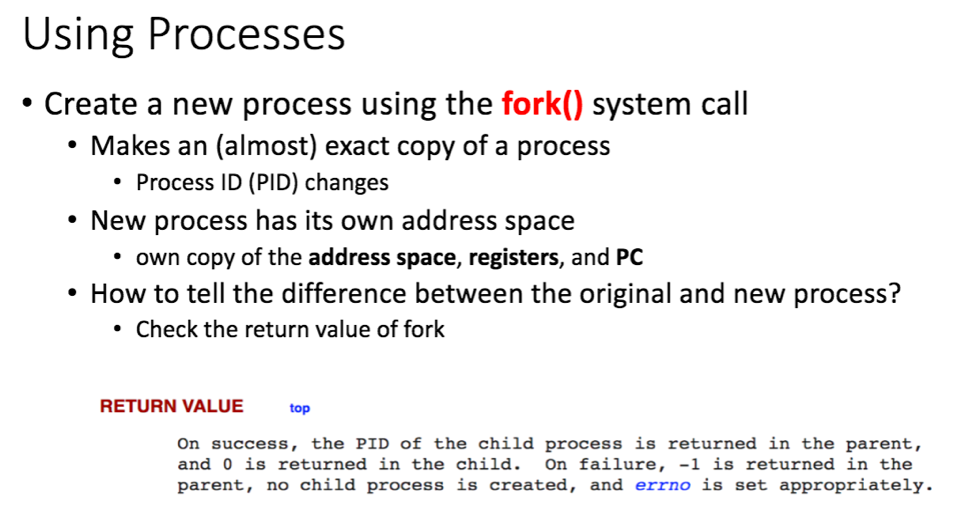
FALSE

**Process spawning**: When the OS creates a process at the explicit request of another process

If a system does not employ virtual memory each process to be executed must be fully loaded into main memory.

TRUE



****

The processor itself provides only limited support for multiprogramming, and \_\_\_\_\_\_\_\_\_\_ is needed to manage the sharing of the processor and other resources by multiple applications at the same time.  
  
A) memory  
B) data  
**C) software**D) hardware

"The process was placed in a suspended state by an agent; either itself, a parent process, or the OS, for the purpose of preventing its execution," is a characteristic of a \_\_\_\_\_\_\_\_\_ process.  
  
A) blocked  
**B) suspended**  
C) ready  
D) swapped

A \_\_\_\_\_\_\_\_\_\_ is a unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system resources.  
  
A) identifier  
**B) process**C) state  
D) kernel

We can characterize the behavior of an individual process by listing the sequence of instructions, referred to as a \_\_\_\_\_\_\_\_\_\_, that executes for that process.  
  
A) state  
**B) trace**C) process block  
D) priority

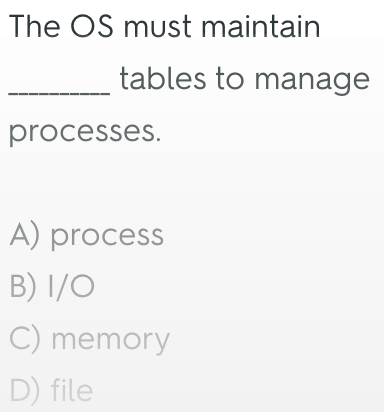
It is the principal responsibility of the \_\_\_\_\_\_\_\_\_\_ to control the execution of processes.  
  
**A) OS**B) process control block  
C) memory  
D) dispatcher

When one process spawns another, the spawned process is referred to as the \_\_\_\_\_\_\_\_\_\_ .  
  
A) trap process  
**B) child process**C) stack process  
D) parent process

\_\_\_\_\_\_\_\_\_\_ involves moving part or all of a process from main memory to disk.  
  
**A) Swapping**B) Relocating  
C) Suspending  
D) Blocking

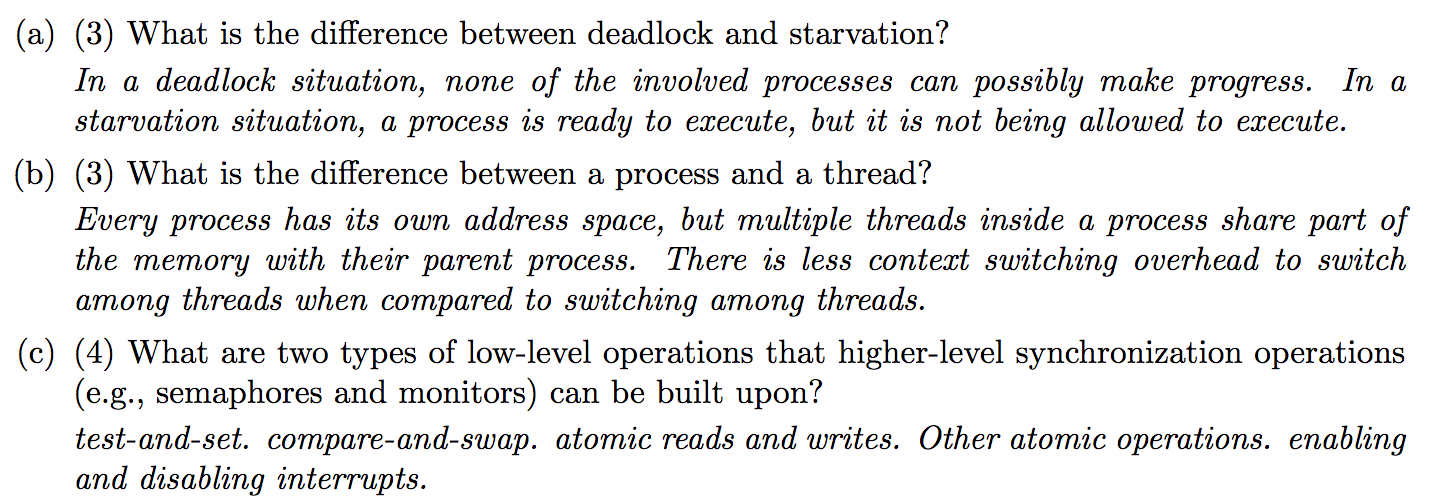
When a process is in the \_\_\_\_\_\_\_\_\_ state it is in secondary memory but is available for execution as soon as it is loaded into main memory.  
  
A) Blocked  
B) Blocked/Suspend  
C) Ready  
**D) Ready/Suspend**

A process is in the \_\_\_\_\_\_\_\_\_ state when it is in main memory and awaiting an event.  
  
**A) Blocked**B) Blocked/Suspend  
C) Ready/Suspend  
D) Ready



What does a test-and-set instruction do?

-performs a memory read and write in a single atomic step



**A TRAP instruction** is a software interrupt. TRAP instructions are used to switch from user mode to Kernel mode and start execution at a fixed address within the kernel.

