1. Scenarios in which Deadlock will NEVER occur.

Deadlock avoidance/prevention algorithms check resource requests and possible availability to prevent deadlock

– Guarantee that deadlock will never occur because of the way the system is structured.

– Breaks one of the four necessary condi3ons

Deadlock detection algorithms ﬁnd instances of deadlock and try to recover

– Admit the possibility of deadlock occurring and periodically check for it

*hold and wait* cannot occur.

Another prevention scheme is to impose an order on the resources and require processes to request resources in increasing order.  This prevents *cyclic wait*

a. What are the arguments for installing the deadlock-avoidance algorithm?

b. What are the arguments against installing the deadlock-avoidance algorithm?

Answer:

An argument for installing deadlock avoidance in the system is that we could ensure deadlock would never occur. In addition, despite the increase in turnaround time, all 5,000 jobs could still run.

An argument against installing deadlock avoidance software is that deadlocks occur infrequently and they cost little when they do occur.

A proactive approach would be *Deadlock Prevention* (to NEVER let this happen) or *Deadlock Avoidance* (TRY to avoid this situation).

If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback). This is known as *Deadlock Detection* (after the fact).

2. Differences between Long, Medium, and Short term Schedulers.

* Long Term Scheduler admits new jobs to keep load balanced between I/O and CPU bound processes
* Medium Term Scheduler - ensures the right mix by sometimes swapping out jobs to secondary storage (virtual memory) and resuming them later
* Short Term Scheduler - decides what process runs next on the CPU according to one or more a scheduling algorithms

1. Long Term Scheduler: It is also called job scheduler. Long term scheduler determines which programs are admitted to the system for processing. Job scheduler selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduler. The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. On some systems, the long term scheduler may be absent or minimal. Time-sharing operating systems have no long term scheduler. When process changes the state from new to ready, then there is a long term scheduler.
2. Short Term Scheduler: It is also called CPU scheduler. Main objective is increasing system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects from among the processes that are ready to execute and allocates the CPU to one of them. Short term scheduler also known as dispatcher, execute most frequently and makes the fine grained decision of which process to execute next. Short term scheduler is faster than long term scheduler.
3. Medium Term Scheduler: Medium term scheduling is part of the swapping function. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium term scheduler is in charge of handling the swapped out-processes.

3. Concept of multi-programming – and the OS’s that support vs. that do not.

* Concurrent applications share Processor Time
* Multiprogramming (1 CPU) or Multiprocessing (2 or more CPUs)

Only one access to a memory location can be made at a time

Read/Write different location at the same time is allowed.

Read/Write the same location at the same time is **NOT allowed**

Process scheduling is one of the most important functions of an operating system that supports multiprogramming. This function is heavily dependent on **queues**.

4. “Jacketing”

When a blocking system call has a user-level jacket, the jacket checks to see if the resource is available ( device is free) and if not another thread is started. When the calling thread is scheduled again (by the thread library) it once again checks the state of the device.

o The major problem with user-level threads is the blocking of all threads within a process when one blocks.

o A possible solution is known as jacketing.

o A blocking system call has a user-level jacket.

o The jacket checks to see if the resource is available.

Example: A device when it is free.

o If not another thread is started.

o When the calling thread is scheduled again (by the thread library) it once again checks the state of the device.

o So there has to be some way of determining if resources are available to accept

requests immediately.

5. Threads, part of the same process, share what?

It shares with other threads belonging to the same process its code section, data section, and other operating-system

resources, such as open ﬁles and signals.

6. Facts about the Round Robin Scheduling Algorithm

* 1. Priority or Non-Priority Based? Non-Priority Based
  2. Pre-emptive or Non Pre-emptive? Explain: Uses preemption based on a clock (timeslice)
  3. What are the Wait and Turn Around Times (Show Work)?

**Waiting:** (570-190) + (950-280-20) + (280-80-40) + (1200-400-60) + (1100-250-80) / 5 =  
540 ms

**Turnaround:** 570 + (950-20) + (280-40) + (1200-60) + (1100-80) / 5 = 780 ms

Uses preemption based on a clock (timeslice)

An amount of time is determined that allows each process to use the processor for that length of time

A timeslice is an allocated number of CPU cycles given to a process.

Choosing a time quantum

* + Too short - inordinate fraction of the time is spent in context switches.
  + Too long - reschedule latency is too great. If many processes want the CPU, then it's a long time before a particular process can get the CPU. This then acts like FCFS.
  + Adjust so most processes won't use their slice. As processors have become faster, this is less of an issue.

7. Facts about the Multi-Level Feedback Queue

* Each queue has its scheduling algorithm.
* Then some other algorithm (perhaps priority based) arbitrates between queues.
* Can use feedback to move between queues

Method is complex but flexible

* Penalizes jobs that have been running longer by lowering their priority
* Doesn’t matter remaining time process still needs to execute
* Even though priority is lowered, the next time around the remaining process gets a double timeslice.
* **Is like Round Robin, but is Priority Based**

Priority or Non-Priority Based? Priority Based

Pre-emptive or Non Pre-emptive? Explain: Pre-emptive because preemptive scheduling involves scheduling based on the highest priority

8. Process Image

Process tables: contains what the operating system must know to manage and control processes, including:

– Process location:

First a program statically consists of a set of instructions that manipulate data, thus the operating system needs to allocate space for its code and data. In addition, the dynamic execution of a program requires a stack that is used

to keep track of procedure calls and parameter passing between procedures. Finally, each process has associated with it a number of attributes that are used by the operating system for process control. Typically the operating system needs to maintain a structure called process control block (PCB) containing these attributes. The collection of the above code, data, stack, and attributes is referred to as process image.

The location of a process image depends on how the memory management is implemented.

When a programme is loaded as a process it is allocated a section of virtual memory which forms its useable address space. Within this process image there are typically at least four elements :

Program code (or text), Program data, Stack, Process Control Block

Facts about the Shortest Job First Algorithm

1. Priority or Non-Priority Based? Non-Priority Based
2. Pre-emptive or Non Pre-emptive? Preemptive policy –based on time slice or the need of the process being served

* Preemptive policy – if another process in the queue is shorter. Not priority Based.
* Process with shortest expected processing time is selected next, regardless of priority.
* Short process jumps ahead of longer processes

Optimal for minimizing queuing time, but difficult to implement. Tries to predict the process to schedule based on previous history.

Still possibility of starvation for longer processes.

Facts about the Consumer/Producer Algorithm

One or more producers are generating data and placing these in a buffer

A single consumer is taking items out of the buffer one at time

Only one producer or consumer may access the buffer at any one time

The problem is process synchronization.

Solution? The solution is the use of semaphores the protect the critical section.

11. Semaphores and their use for Synchronization

* A special integer value with three operations
  + initialize to a non-negative value
  + wait – decrements the value. If the value < 0, the thread executing wait is blocked
  + signal – increments the value. If the value <= 0, then a thread blocked by a wait operation is unblocked

Types:

* + *Binary* semaphore ([Example](http://www.doc.ic.ac.uk/~jnm/concurrency/classes/SemaDemo/SemaDemo.html)) – integer value can range only between 0 and 1; can be simpler to implement
  + Can implement a counting semaphore as a binary semaphore (simpler for some hardware)
* *Counting* ([Example](http://williamstallings.com/OS/Animation/Queensland/SYNC.SWF)) semaphore – integer value can range over an unrestricted domain
* A *spinlock* semaphore – no context switch is required. This is good for short interaction times between synchronized processes.
  + For longer times, the *wait* semaphore is modified to block the process, and transfer control back to the scheduler to select another process.
  + The wakeup operation resumes control.

12. Mutexes, Monitors

Mutexes

**Mutual Exclusion using Mutex Locks**

* Advantages of using Mutex Locks to protect critical section:
  + No busy waiting
  + Easy to apply in case of more than 2 threads

Monitors

* A high-level synchronization constructs (
* Ensures only one process at a time is active within the monitor
* Overcomes some of the “timing” issues when using semaphores

Mutexes are low level construct. They just provide mutual exclusion and memory visibility/ordering. Monitors, on the other hand, are higher level - they allow threads to wait for an application specific condition to hold.

So, in some cases monitors are just overkill over a simple lock/unlock, but in most cases mutexes alone are not nearly enough - so you see them used with one or more condition variables - conceptually using monitors equivalent

13. Ideal Solution to the Reader-Writer Problem

Any number of readers may simultaneously read the file.

Only one writer at a time may write to the file

If a writer is writing to the file, no reader may read it

Solution? Critical sections are used to provide data integrity, and a mutex lock is used for mutual exclusion of the write processes.

What is "Busy Waiting"?

* + Process is always checking to see if it can enter the critical section (using a “turn” variable).
  + Process can do nothing productive until it gets permission to enter its critical section

Busy waiting is where a process checks repeatedly for a condition- it is "waiting" for the condition, but it is "busy" checking for it. This will make the process eat CPU (usually).

Busy waiting wastes CPU cycles that some other process might be able to use productively.

While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire(). In fact, this type of mutex lock is also called a **spinlock** because the process “spins” while waiting for the lock to become available.

15. Solaris approach to threads

**Solaris**

-- supports multiple threads

* The key is versatility. We can use ULTs when logical parallelism does not need to be supported by hardware parallelism (we save mode switching). One of the examples is multiple windows but only one is active at any one time.
* If ULT threads can block then we can add two or more LWPs to avoid blocking the whole application
* Note versatility of SOLARIS that can operate like Windows or like conventional Unix
* Process includes the user’s address space, stack, and process control block
* User-level threads (threads library)
  + invisible to the OS
  + are the interface for application parallelism
* Kernel threads
  + the unit that can be dispatched on a processor
* Lightweight processes (LWP)
  + each LWP supports one or more ULTs and **maps to exactly one KLT**
  + LWPs are visible to applications
  + therefore LWP are the way the user sees the KLT
  + constitute a sort of ’virtual CPU’

16. Definition of Concurrency

Concurrency is a general term that deals with the communication and synchronization of processes (or threads) executing together on the same OS

17. Type of interrupt used to switch between jobs ????

* A process switch may occur whenever the OS has gained control of CPU. i.e. when:
  + **Interrupt** 
    - the cause is external to the execution of the current instruction. Control(Control Switch) is transferred to Interrupt Handler 🡸==

Examples

* Timers
  + process has used up its time - transfers to ready state
* I/O completion
  + transfer process to ready state
  + CPU scheduler will decide when to resume it
* Memory fault, page fault
  + In virtual memory systems, process requests a page that is not in memory
  + block process, read page in memory

Are threads resources? Why or why not?

Threads share most of its resources with other threads of the same process. Threads do own resources that define the thread's context. This includes the thread id, set of registers including the stack pointer and program counter, and stack. Threads must share other resources such as the processor, memory, and file descriptors required in order for it to perform its task. File descriptors are allocated to each process separately and threads of the same process compete for access to these descriptors. In memory, the processor, and other globally allocated resources, threads contend with other threads of its process as well as the threads of other processes for access to these resources.

A thread can allocate additional resources such as files or mutexes, but they are accessible to all the threads of the process. There are limits on the resources that can be consumed by a single process. Therefore, all the threads in combination must not exceed the resource limit of the process. If a thread attempts to consume more resources than the soft resource limit defines, it is sent a signal that the process's resource limit has been reached. Threads that allocate resources must be careful not to leave resources in an unstable state when they are canceled. A thread that has opened a file or created a mutex may be terminated, leaving the file open or the mutex locked. If the file has not been properly closed and the application is terminated, this may result in damage to the file or loss of data. A thread terminating after locking a mutex prevents access to whatever critical section that mutex is protecting. Before it terminates, a thread should perform some cleanup, preventing these unwanted situations from occurring

19. What happens between the Running and Waiting State?

CPU scheduling decisions occur when process:

1. Switches from running to waiting state.

* **RUNNING to WAITING** – It is handled by the Process Scheduler and is initiated by an instruction in the process such as a command to READ, WRITE or other I/O requests, or a page fetch is required.

The wait method defined in the Thread class,

can be used to convert a thread from Running state

to Waiting state.

The process scheduler chooses the processes or jobs to be executed only at the running state. Therefore, there is no transition happen between WAITING STATE to RUNNING STATE for the reason that there is no possibility to process a job by bypassing the running state. Also, in the waiting state, jobs on this state are the jobs waiting for an event happen from its required external devices or either waiting for its I/O completion.

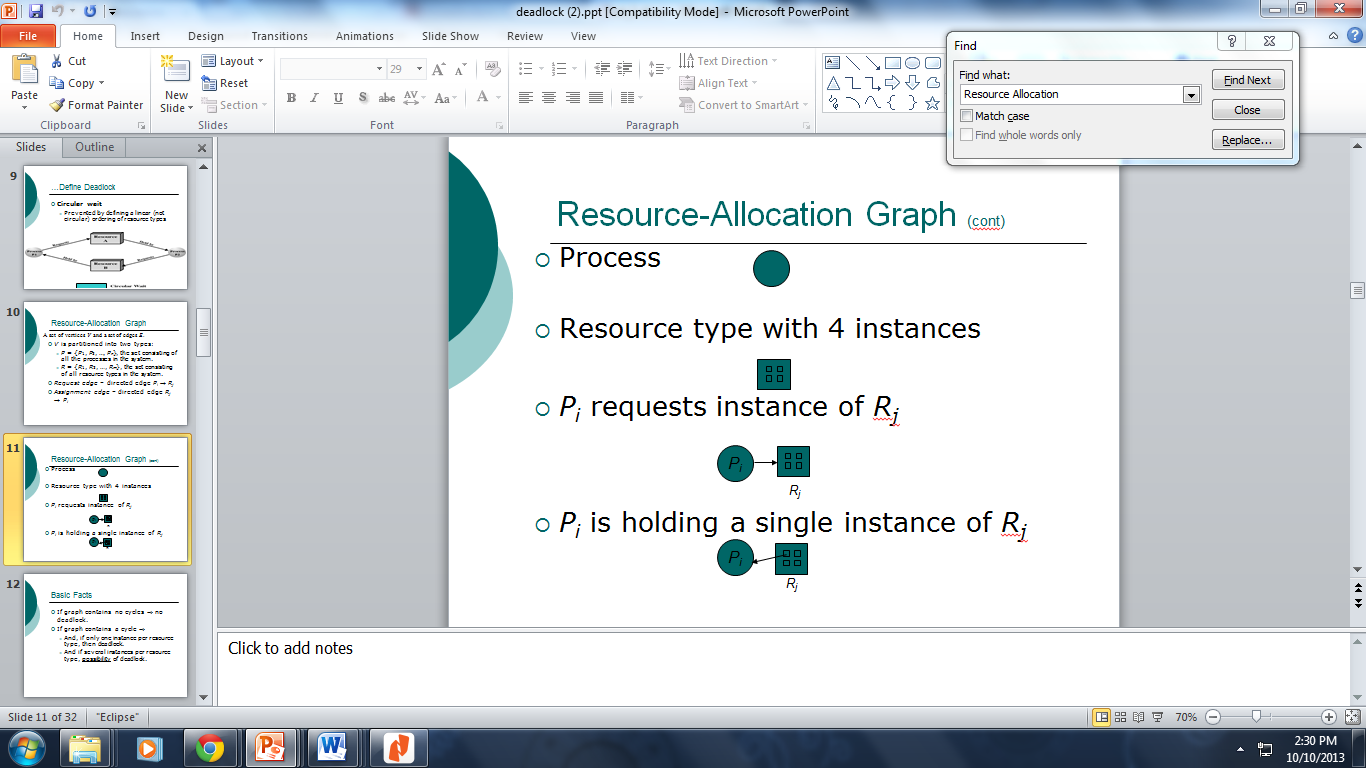
20. What is contained in the Process Control Block?

* Contains information associated with each process
  + **Process State** - e.g. new, ready, running etc.
  + **Process Number** – Process ID
  + **Program Counter** - address of next instruction to be executed
  + **CPU registers** - general purpose registers, stack pointer etc. (computer architecture specific)
  + **CPU scheduling information** - process priority, pointer
  + **Memory Management information** - base/limit information
    - **I/O Status information** - list of I/O devices allocated
    - **Thread information** (if applicable)

21. Resource Allocation to processes - when does/can it happen?

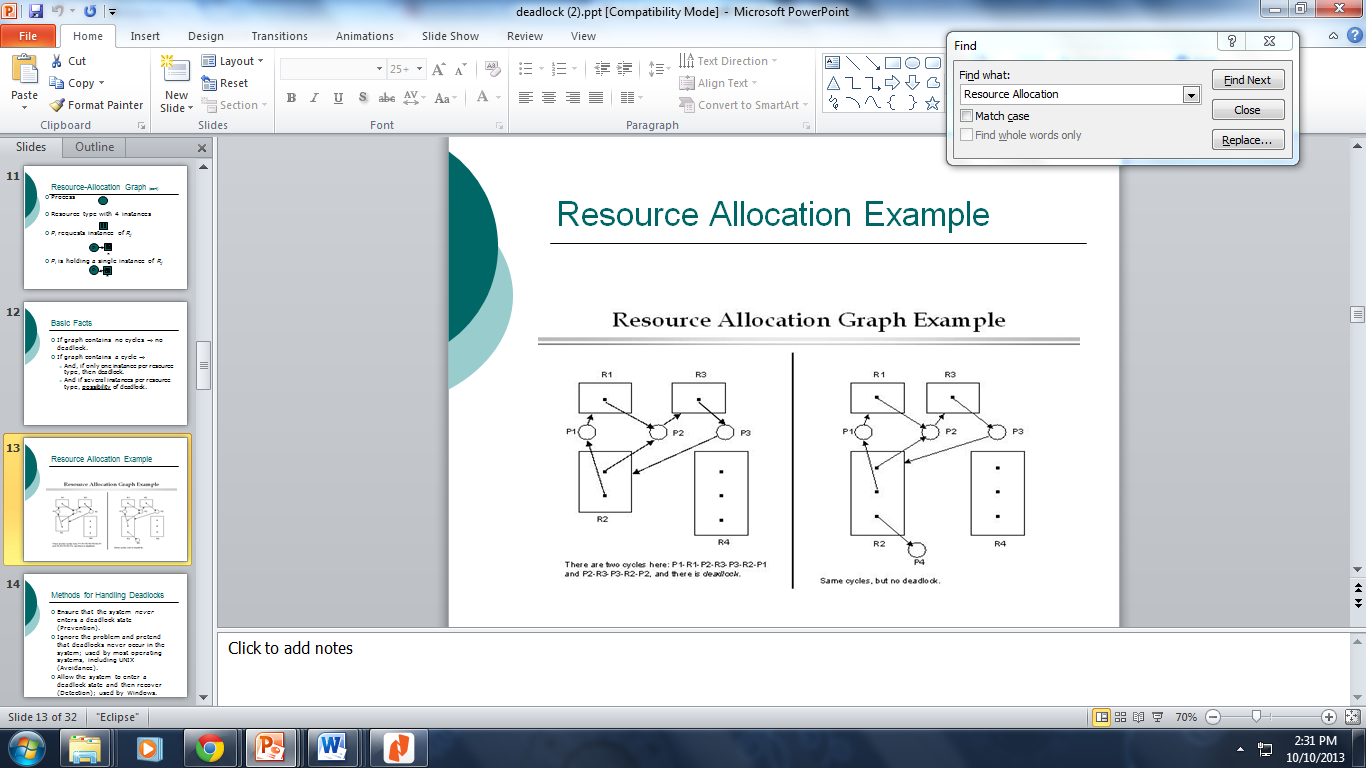
A set of vertices *V* and a set of edges *E*.

* *V* is partitioned into two types:
  + *P* = {*P*1, *P*2, …, *Pn*}, the set consisting of all the processes in the system.
  + *R* = {*R*1, *R*2, …, *Rm*}, the set consisting of all resource types in the system.
* *Request edge* – directed edge *P*i → *Rj*
* *Assignment edge* – directed edge *Rj* → *Pi*



* If graph contains no cycles ⇒ no deadlock.
* If graph contains a cycle ⇒
  + And, if only one instance per resource type, then deadlock.

And if several instances per resource type, possibility of deadlock



22. If process terminates, what happens to its children?

Reasons for Process Termination

* Parent terminates, so child processes terminate as well

23. The differences between Deadlock Detection, Avoidance, and Prevention

**Prevention**:

• The goal is to ensure that at least one of the necessary conditions for deadlock can never hold.

• Deadlock prevention is often impossible to implement.

• The system doesnot require additional apriori information regarding the overall potential use of each resource for each process.

• In order for the system to prevent the deadlock condition it does not need to know all the details of all resources in existence, available and requested.

• Deadlock prevention techniques include non-blocking synchronization algorithms, serializing tokens, Dijkstras algorithm etc.

• Resource allocation strategy for deadlock prevention is conservative, it under commits the resources.

• All resources are requested at once.

• In some cases preempts more than often necessary.

**Avoidance:**

• The goal for deadlock avoidance is to the system must not enter an unsafe state.

• Deadlock avoidance is often impossible to implement.

• The system requires additional apriori information regarding the overall potential use of each resource for each process.

• In order for the system to be able to figure out whether the next state will be safe or unsafe, it must know in advance at any time the number and type of all resources in existence, available, and requested.

• Deadlock avoidance techniques include Banker’s algorithm, Wait/Die, Wound/Wait etc.

• Resource allocation strategy for deadlock avoidance selects midway between that of detection and prevention.

• Needs to be manipulated until atleast one safe path is found.

• There is no preemption.

**Detection:**

• The goal is to detect the deadlock after it occurs or before it occurs.

• Detecting the possibility of a deadlock before it occurs is much more difficult and is, in fact, generally undecidable. However, in specific environments, using specific means of locking resources, deadlock detection may be decidable.

• The system doesnot requires additional apriori information regarding the overall potential use of each resource for each process in all cases.

• In order for the system to detect the deadlock condition it does not need to know all the details of all resources in existence, available and requested.

• A deadlock detection technique includes, but is not limited to, Model checking. This approach constructs a Finite State-model on which it performs a progress analysis and finds all possible terminal sets in the model.

• Resource allocation strategy for deadlock detection is very liberal. Resources are granted as requested.

• Needs to be invoked periodically to test for deadlock.

• Preemption is seen.

24. Differences between a mode switch, thread switch, and context switch

* Mode switching: process is suspended (e.g. a process interrupt) but immediately after this it resumes execution on the same CPU:
  + only information necessary to resume execution of the same process needs to be saved (e.g. program counter)
* Process switching: process is suspended but another process will take the CPU next:
  + more may need to be saved: context, data...
  + also process state must be updated.
* As a process executes, it changes *state* (context switch) – either *running* or *not running*.
* **context switch** is the process of storing and restoring the [state](http://en.wikipedia.org/wiki/State_(computer_science)) ([context](http://en.wikipedia.org/wiki/Context_(computing))) of a [process](http://en.wikipedia.org/wiki/Process_(computing)) so that execution can be resumed from the same point at a later time.

25. Types of Semaphores

* + *Binary* semaphore ([Example](http://www.doc.ic.ac.uk/~jnm/concurrency/classes/SemaDemo/SemaDemo.html)) – integer value can range only between 0 and 1; can be simpler to implement
  + Can implement a counting semaphore as a binary semaphore (simpler for some hardware)
* *Counting* ([Example](http://williamstallings.com/OS/Animation/Queensland/SYNC.SWF)) semaphore – integer value can range over an unrestricted domain
* A *spinlock* semaphore – no context switch is required. This is good for short interaction times between synchronized processes.
  + For longer times, the *wait* semaphore is modified to block the process, and transfer control back to the scheduler to select another process.
  + The wakeup operation resumes control.

26. Valid Semaphore operations

* A special integer value with three operations
  + initialize to a non-negative value
  + wait – decrements the value. If the value < 0, the thread executing wait is blocked
  + signal – increments the value. If the value <= 0, then a thread blocked by a wait operation is unblocked

27. Two major characteristics of a process.

1. **Unit of resource ownership** – a process is allocated.
   * The OS uses a virtual address space to hold process image containing the PCB.
   * It controls some resources (files, I/O devices...)
2. **Unit of execution** - process is an execution path (trace) through one or more programs
   * execution may be interleaved with other processes
   * the process has an execution state and a priority

These 2 characteristics are treated independently by some modern OS

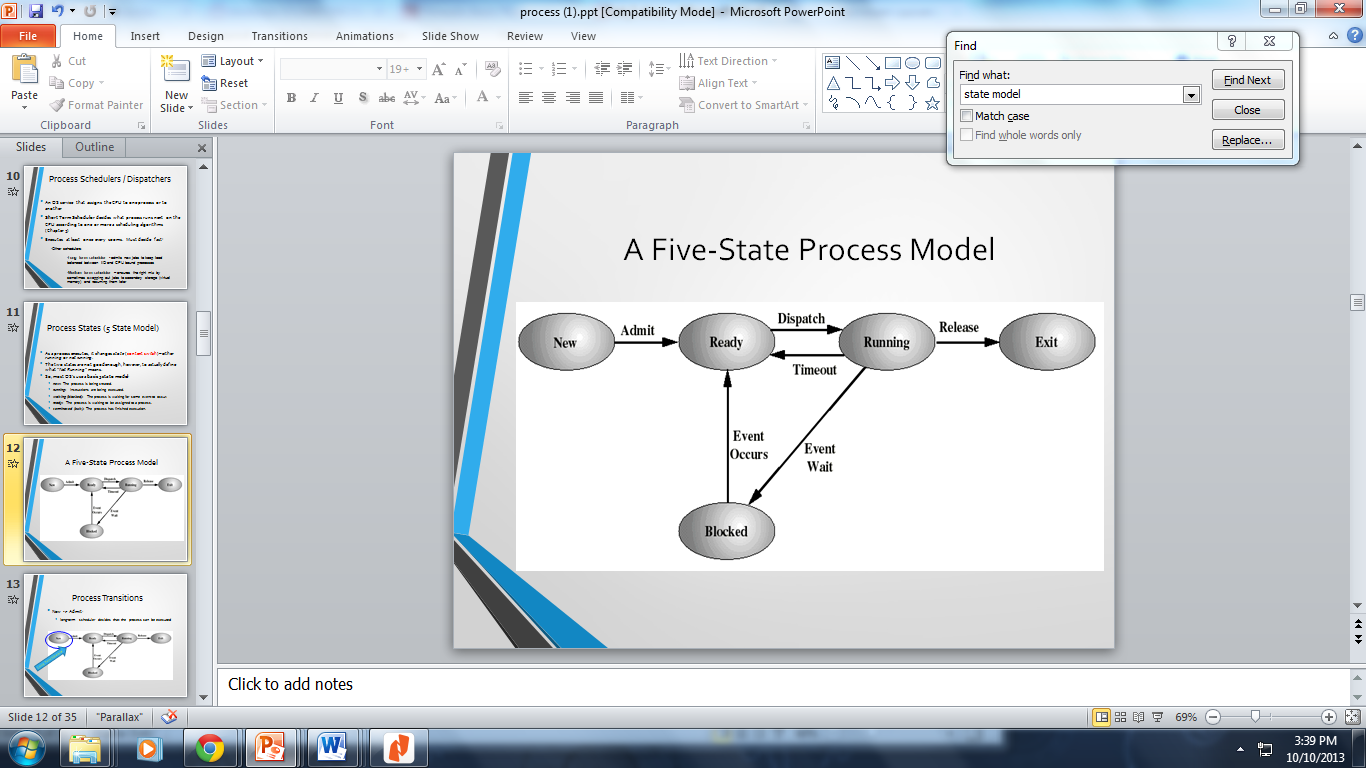
* The *unit of resource ownership* is usually referred to as the *process or task,* itself.
* The *unit of execution* is usually referred to as a thread.

28. 5 State vs. 7 State process model (differences)

Process States (5 State Model)

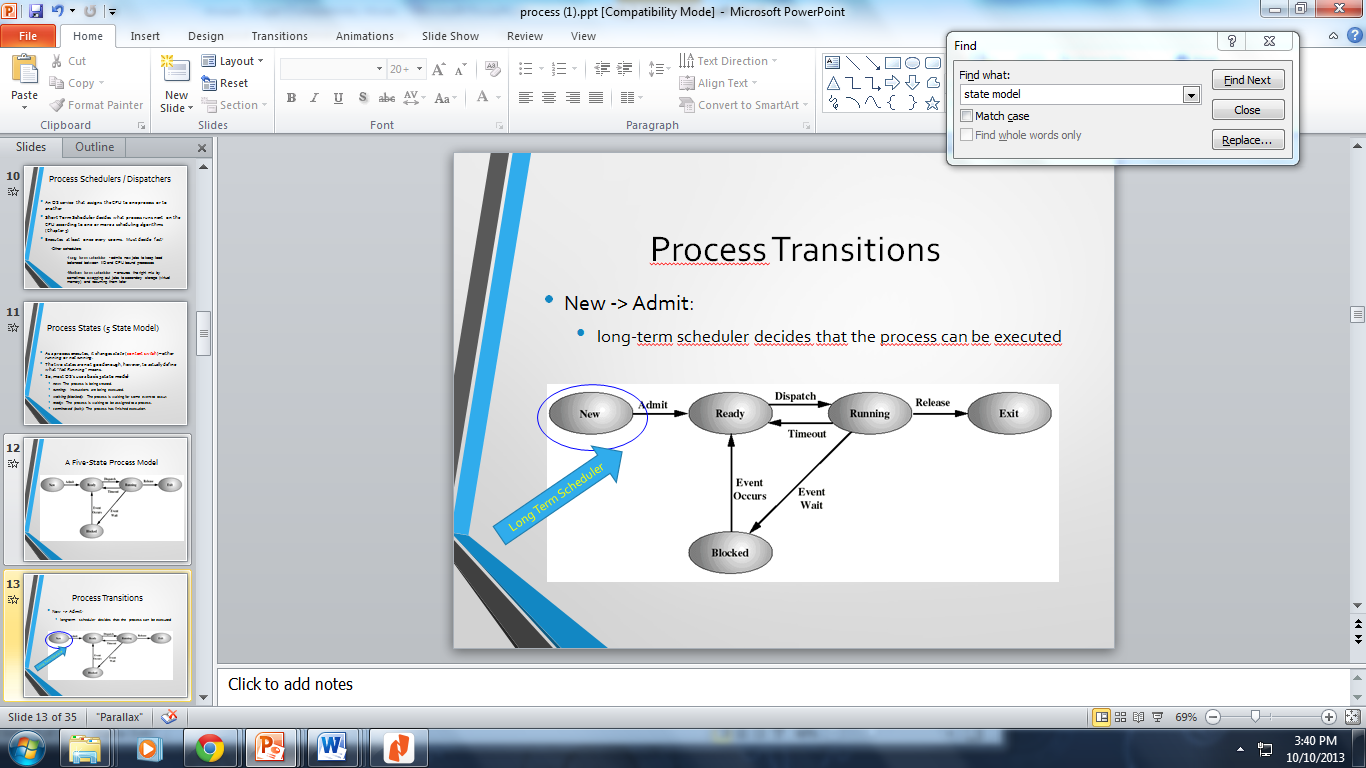
* As a process executes, it changes *state* (context switch) – either *running* or *not running*.
* The two states are not good enough, however, to actually define what “*Not Running*” means.
* So, most OS’s use a **basic 5 state model**:
  + ***new*:** The process is being created.
  + ***running*:** Instructions are being executed.
  + ***waiting (blocked)*:** The process is waiting for some event to occur.
  + ***ready*:** The process is waiting to be assigned to a process.

***terminated (exit)*:** The process has finished execution



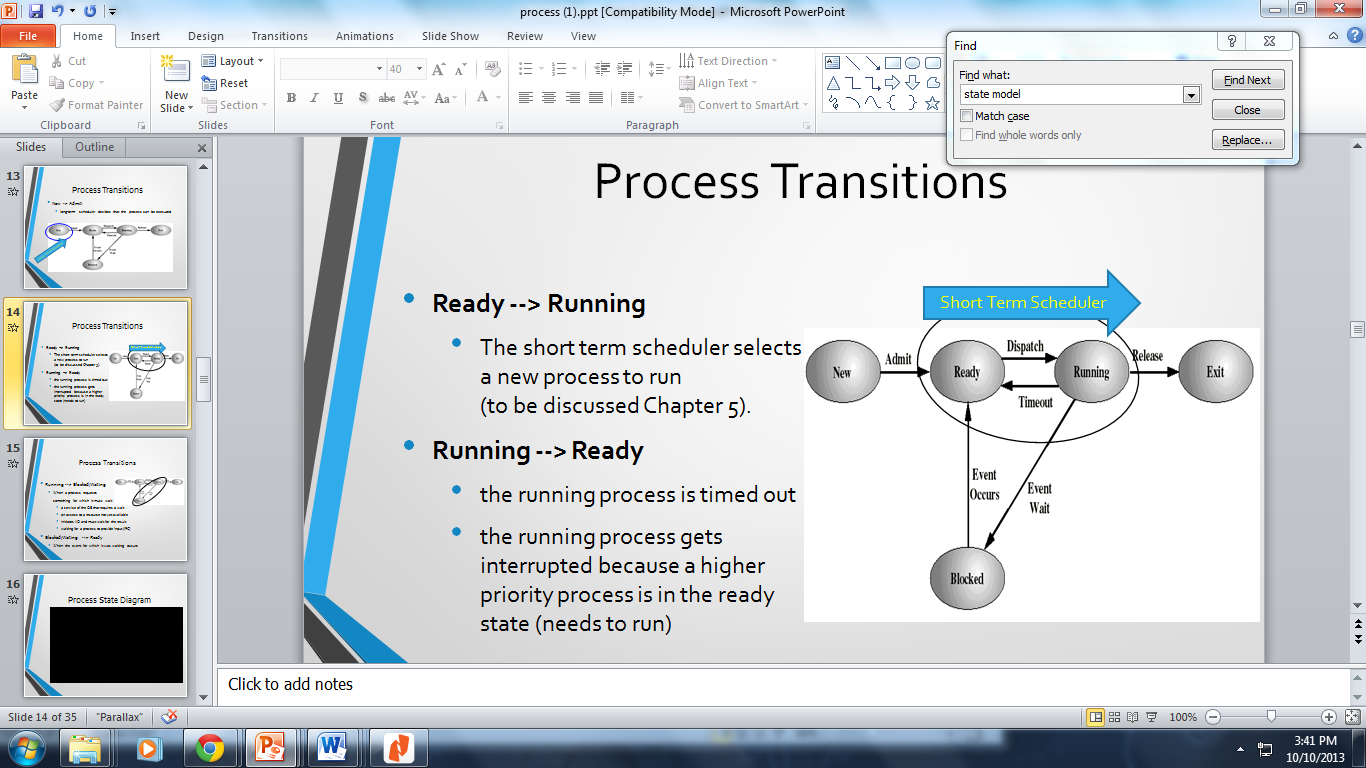
Process Transitions

* New -> Admit:
  + long-term scheduler decides that the process can be executed



Process Transitions

* **Ready --> Running**
  + The short term scheduler selects a new process to run   
    (to be discussed Chapter 5).
* **Running --> Ready**
  + the running process is timed out
  + the running process gets interrupted because a higher priority process is in the ready state (needs to run)

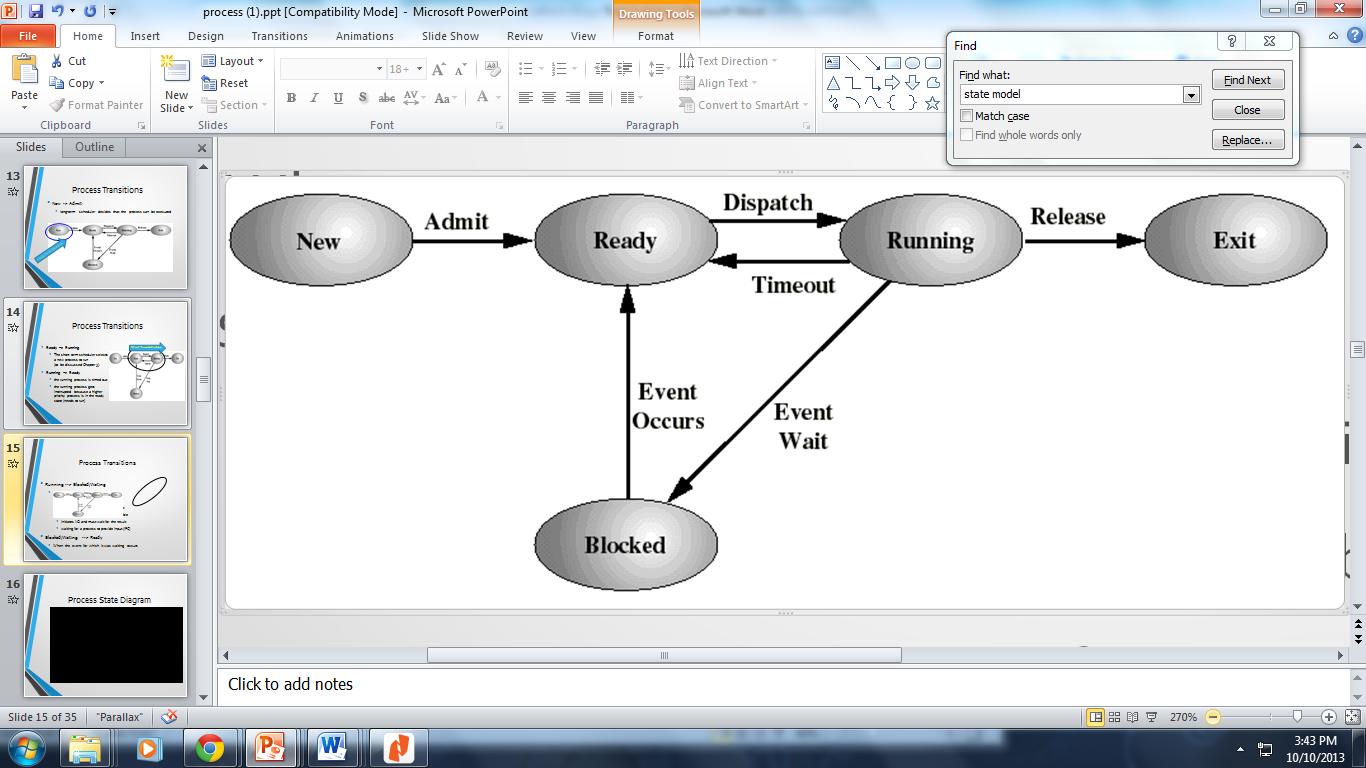


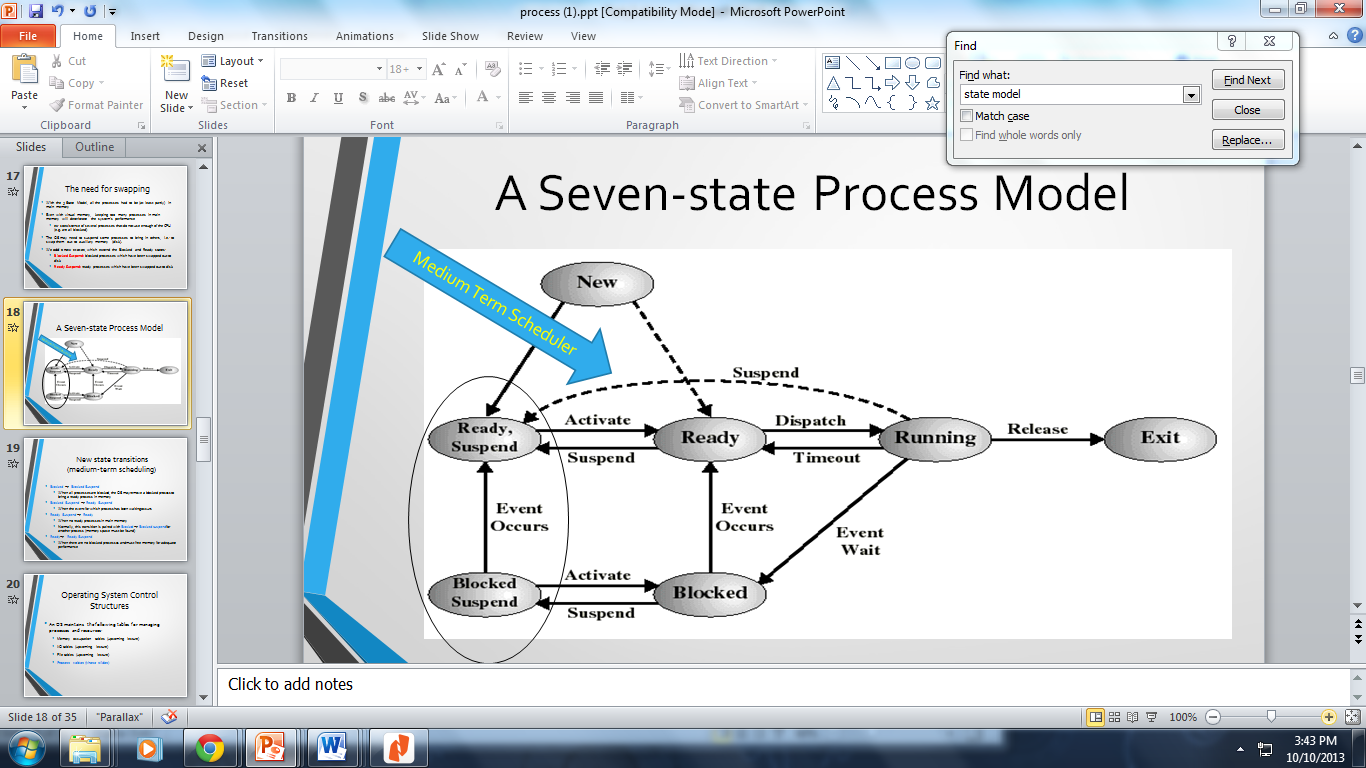
Process transition

* Running --> Blocked/Waiting
  + When a process requests

something for which it must wait

* + - a service of the OS that requires a wait
    - an access to a resource not yet available
    - initiates I/O and must wait for the result
    - waiting for a process to provide input (IPC)
* Blocked/Waiting --> Ready
  + When the event for which it was waiting occurs





The need for swapping

* With the 5 State Model, all the processes had to be (at least partly) in main memory
* Even with virtual memory, keeping too many processes in main memory will deteriorate the system’s performance
  + ex: coexistence of several processes that do not use enough of the CPU (e.g. are all blocked)
* The OS may need to suspend some processes to bring in others, i.e.: to swap them out to auxiliary memory (disk).
* We **add 2 new states**, which extend the Blocked and Ready states:
  + **Blocked Suspend:** blocked processes which have been swapped out to disk
  + **Ready Suspend:** ready processes which have been swapped out to disk
* Blocked --> Blocked Suspend
  + When all processes are blocked, the OS may remove a blocked process to bring a ready process in memory
* Blocked Suspend --> Ready Suspend
  + When the event for which process has been waiting occurs
* Ready Suspend --> Ready
  + When no ready processes in main memory
  + Normally, this transition is paired with Blocked --> Blocked suspend for another process (memory space must be found)
* Ready--> Ready Suspend
  + When there are no blocked processes and must free memory for adequate performance

29. Mutual Exclusion satisfied via Algorithm 1, 2, and Peterson’s Algorithm (differences)

**Satisfying Synchronization via Software Support, Algorithm 1**

* Employs “Busy Waiting”
  + Process is always checking to see if it can enter the critical section (using a “turn” variable).
  + Process can do nothing productive until it gets permission to enter its critical section
* This is an example of a co-routine
  + Software is designed to be able to pass execution control back and forth between itself
  + This Algorithm is inadequate to support concurrent processing because it satisfies mutual exclusion, **but not progress**
* The two threads take alternative turns to enter the critical sections. turn==0 means thread 0’s turn. If a thread wants to enter the CS but it’s not its turn yet, it will busy- wait at EnterCS().

**Problems of First Algorithm**

* Processes must strictly alternate in their use of their critical section
* If one process fails, the other process is permanently blocked…thus progress (one of the 3 conditions) is not satisfied.

**Algorithm 2**

* Uses a Flag for each process
  + Each process may examine, but not alter the other process’ flag.
  + A failure outside the critical section, allows others to keep running.
  + A failure inside the critical section, blocks all other processes.
* Does not guarantee mutual exclusion.

Each thread has its own key to enter the CS: flag[0]==true means “thread 0 is in CS now”. If one thread dies or blocks outside the CS, the other thread can still enter the CS.

Our second attempt does NOT guarantee mutual exclusion! Each thread can check the flags and then proceed to enter the critical section at the same time.

**Satisfying Synchronization via Software Support, (Peterson’s Solution)**

* Each process gets a turn at the critical section
* If a process wants the critical section, it sets its flag and also waits for its turn (mutual exclusion).
* Since *Pi* does not change the value of the variable turn while executing in the while statement, *Pi*, will enter the critical section (progress) after at most one entry by *Pj* (bounded waiting).
* All 3 rules have been met for cooperation and competition.
* In Peterson’s algorithm there are three global shared variables: flag[0], flag[1] and turn. flag[0] reflects whether thread 0 wants to enter CS. (similar for flag[1]) If both threads want to enter, the shared variable ‘turn’ determines who takes its turn first.

• Differences between Dekker and Peterson

– Dekker’s algorithm makes no assumptions about atomicity, while Peterson’s algorithm assumes assignment is an atomic operation.

– Dekker’s algorithm works on a machine where bits are scrambled during simultaneous assignment; Peterson’s algorithm does not.

Dekker's Algorithm is the first recognized algorithm that solves the mutual exclusion setback in concurrent programming and it's used in process queuing to allow two different threads to segment the same single-use source minus conflict.

The difference between Peterson's algorithm and Dekker's algorithm is Dekker algorithm has much more complex code with higher efficiency, while Peterson has simpler code. Dekker algorithm is not expendable as it's only for two maximum mutual exclusion processes, while Peterson can be extended for more than 2 processes

30. How thread can be “running” with its process is “blocked”.

* **Note: Thread states are maintained INDEPENDENT of the process state.** 
  + A thread may still be in a running state (well, sort of), while it’s process is blocked

----- > When a thread makes a system call, the whole process will be blocked but for the thread library that thread is still in the running state

**Benefits of Threads**

* Responsiveness
  + Allows a program to continue running, even if part of it is blocked

31. Using Banker's Algorithm, show system is in a safe or unsafe state (work on it)

Resource Allocation Denial

* Referred to as the **Banker’s Algorithm**
* State of the system is the current allocation of resources to processes
* Safe state is where there is at least one sequence that does not result in deadlock
* Unsafe state is a state that is not safe and may lead to deadlock.

32. Compute average Wait and Turn-Around Time for various processes.