



# Operating System Design

Dr. Jerry Shiao, Silicon Valley University

# CPU Scheduling

## ■ Overview

### □ CPU Scheduling

#### ■ Scheduling Algorithms

- First-Come-First-Serve
- Shortest-Job-First
- Priority
- Round Robin
- Multilevel Queue Scheduling

#### ■ Multiple-Processor Scheduling

- Processor Affinity
- Load Balancing
- Multicore Processors
- Virtualization

#### ■ Computer System Schedulers

- Windows XP
- UNIX / Linux

### □ Process Synchronization

### □ Race Condition: Concurrent Execution

### □ Synchronization Hardware

# CPU Scheduling

## ■ Overview

### □ Semaphores

- Counting Semaphores
- Binary Semaphores ( Mutexes )
- Spinlocks
- Priority Inversion / Priority Inheritance

### □ Deadlock Characteristic

- Necessary Conditions for Deadlock.

### □ Methods for Handling Deadlocks.

- Deadlock Prevention.
- Deadlock Avoidance.
- Deadlock Detection.

### □ Recovery from Deadlock.

- Manual Recovery.
- Automatic Recovery.
  - Process Termination
  - Resource Preemption

# CPU Scheduling

## ■ CPU Scheduler

- Scheduler manages queues to minimize queuing delay and to optimize performance in a queuing environment.

## ■ Types of Scheduling: *4 types: long term, medium, short term, I/O*

### □ Long-Term Scheduling

- Decision on which programs are admitted to the system for processing, controlling the degree of Multiprogramming.
- Long-Term Scheduler may **limit the degree of Multiprogramming** to provide satisfactory service to current set of processes.

### □ Medium-Term Scheduling *(to find out which process need to be moved out of RAM)*

- Decision **to Swap in a process**, manage degree of Multiprogramming.

### □ Short-Term Scheduling *(mostly for Ready Q)*

- Dispatcher, makes decision on which process to execute next.
- Invoked whenever event can lead to blocking of current process or preempt the current running process: **Clock Interrupts, I/O Interrupts, System Calls, Signals (Semaphores).**

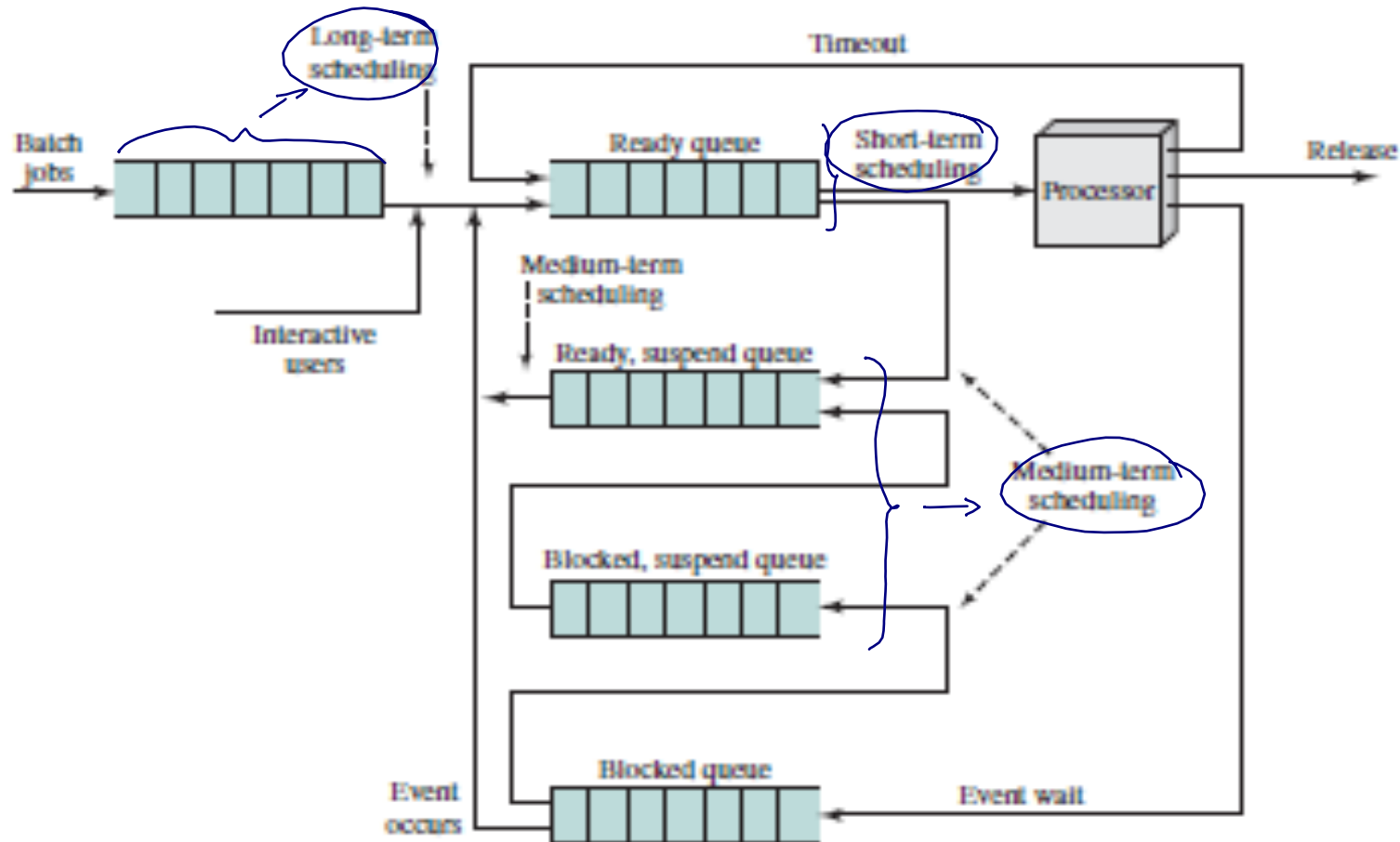
### □ I/O Scheduling

- Decision as to which process pending I/O request shall be handled by the available I/O Device.

*I/O queue handling.*

# CPU Scheduling

- CPU Scheduler
- Queuing Diagram for Scheduling:



# CPU Scheduling

## ■ Scheduling Criteria

## ■ Criteria for comparing CPU-Scheduling Algorithm:

### □ CPU Utilization

- Maximize.
- CPU 40% (lightly loaded) to 90% (heavily loaded).

### □ Throughput

- Maximize.
- Schedule 10 processes per second ( Interactive Load ) to 1 process per second ( Batch Load ).

### □ Turnaround Time

- Minimize.
- Process execution time: Sum of periods waiting to swap into memory, waiting in the Ready Queue, waiting for I/O, and executing on CPU.

### □ Waiting Time

- Minimize.
- Sum of the periods the process spends waiting in the Ready Queue.

### □ Response Time

- Minimize.
- Measure of time from the submission of a request until the time it takes to start responding.

# CPU Scheduling

FCFS: non preemptive, simple, long waiting time.

## Scheduling Algorithms

### First-Come, First-Served Scheduling

- FCFS Scheduling is nonpreemptive.
- Problem for Time-Sharing Systems (needs CPU at regular intervals).
- Simple algorithm, but long wait time.

Burst Time:  
P1 = 24 MSec  
P2 = 3 MSec  
P3 = 3 MSec



P1  
0  
Average Wait Time =  $(0 + P2 + P3) / 3$

P2  
24  
P3  
27

Average wait time.

P1 P2 P3  
 $(0 + 24 + 27) / 3 = 17$



P2  
0  
P3  
3  
P1  
3  
Average Wait Time =  $(0 + P3 + P1) / 3$

Average wait time.

P1 P2 P3  
 ~~$(6 + 0 + 3) / 3 = 9$~~

- Average Wait Time varies on order and CPU bursts.  $(0 + 3 + 3) / 3 = 2$

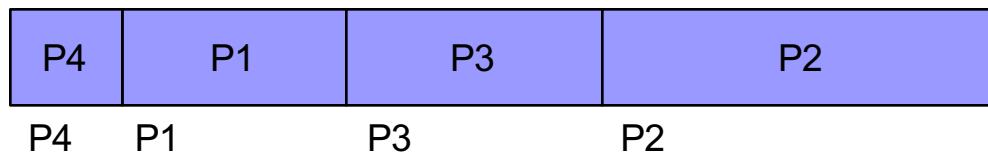
# CPU Scheduling

*SJFS: minimal avg waiting time  
preemptive / non preemptive  
base on CPU burst prediction*

## ■ Scheduling Algorithms

## ■ Shortest-Job-First Scheduling

- CPU is assigned to the process that has the smallest next CPU burst.
- Minimum average waiting time for a set of processes.
- In Short term scheduling, predict the next CPU burst value.
- Preemptive SJF takes CPU when arrived process burst time is less than the remaining time of current process.
- NonPreemptive SJF allows current process to complete burst time before scheduling next process based on burst time.
- Length of CPU burst? Predict value of next CPU burst.



$$\text{Average Wait Time} = 0 + P2 + P3 + P1 / 4$$

*Base on  
the predicted burst  
time, the priority  
of the process in  
the Q is arranged  
accordingly.*

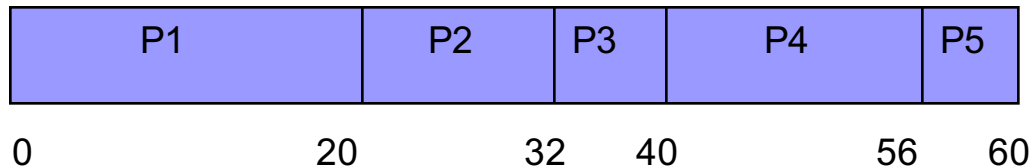
Burst Time:  
P1 = 6 MSec  
P2 = 8 MSec  
P3 = 7 MSec  
P4 = 3 MSec



# CPU Scheduling

## Scheduling Algorithms

### FirstCome, First-Served Scheduling *long waiting time*



#### □ Burst Time:

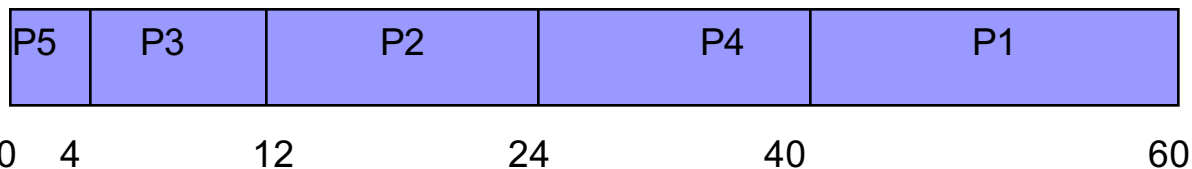
- P1: 20, P2: 12, P3: 8, P4: 16, P5: 4

#### □ Wait Time ( Order of Arrival ).

- P1: 0, P2: 20, P3: 32, P4: 40, P5: 56

#### □ Average Wait Time: $148 / 5 = 29.6$

### Shortest-Job-First Scheduling *not very suitable for realtime system*



#### □ Wait Time ( In Ready Queue at Same Time ).

- P1: 40, P2: 12, P3: 4, P4: 24, P5: 0

#### □ Average Wait Time: $80 / 5 = 16$

# CPU Scheduling

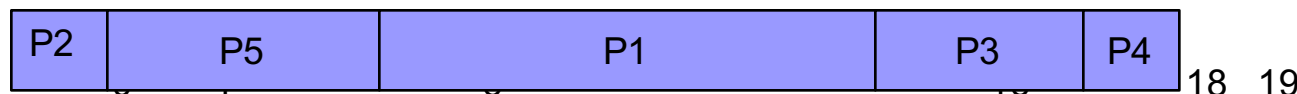
Priority scheduling: - no prediction of CPU burst  
- preemptive / non preemptive  
- introduce Aging concept to help low priority processes executed.

## ■ Scheduling Algorithms

## ■ Priority Scheduling ← more suitable for realtime system

- Inverse of predicted next CPU burst ( Lower burst, higher priority ).  
if a process is running & there is another higher priority process to be served, this process will be taken out of the Q.
- Preemptive: Compare arriving process priority with current running process priority. opposite of preemptive.
- Nonpreemptive: Arriving process priority higher than current running process priority placed in front of the Ready Queue.
- Problem:
  - Indefinite blocking of low priority processes until system lightly loaded.
  - Solution: Increase priority of processes in the Ready Queue ( Aging ).

Process/Burst Time (msec)/Priority: P1/10/3, P2/1/1, P3/2/4, P4/1/5, P5/5/2 = 8.2 msec Average



# CPU Scheduling

## ■ Scheduling Algorithms

## ■ Compare FCFS, SJF, Priority Scheduling

### □ FCFS (First-come, First-Served)

- Non-preemptive. *long waiting time*

### □ SJF (Shortest Job First)

- Non-preemptive or Preemptive.
- When a new (shorter burst) job arrives, can preempt current job or not.
- Short Wait Time.
- Possible process starvation.

### □ Priority

- Non-preemptive or Preemptive.
- When a new (higher priority job arrives).
- Internal / External/ Static / Dynamic
- Possible process starvation.
- Linux 2.4 Kernel nonpreemptive/Linux 2.6 Kernel Preemptive

# CPU Scheduling

Round Robin: FCFS + Preemption  
- time sharing or multitasking

## ■ Scheduling Algorithms

## ■ Round-Robin Scheduling

- Designed for Time-Sharing or Multi-Tasking Operating Systems.
- Based on First-Come-First-Serve Algorithm, but use Preemption to switch between processes. *allowed CPU burst*  
↓
- Preemption: Process will be preempted when quantum expires.
- Quantum: Time slice (unit of time) in which a process executes.
  - Time Quantum too large, RR Policy becomes FCFS Policy. *← no more switching.*
  - Time Quantum too small, RR Policy process time affected by Context Switch time ( < 10 microseconds ). *← Waste time on too many switches*
- Timer Interrupt when Quantum Expires.
  - Current process placed at end of Circular Queue.
  - Current process completes before Quantum Expires, next process selected, no Interrupt.

# CPU Scheduling

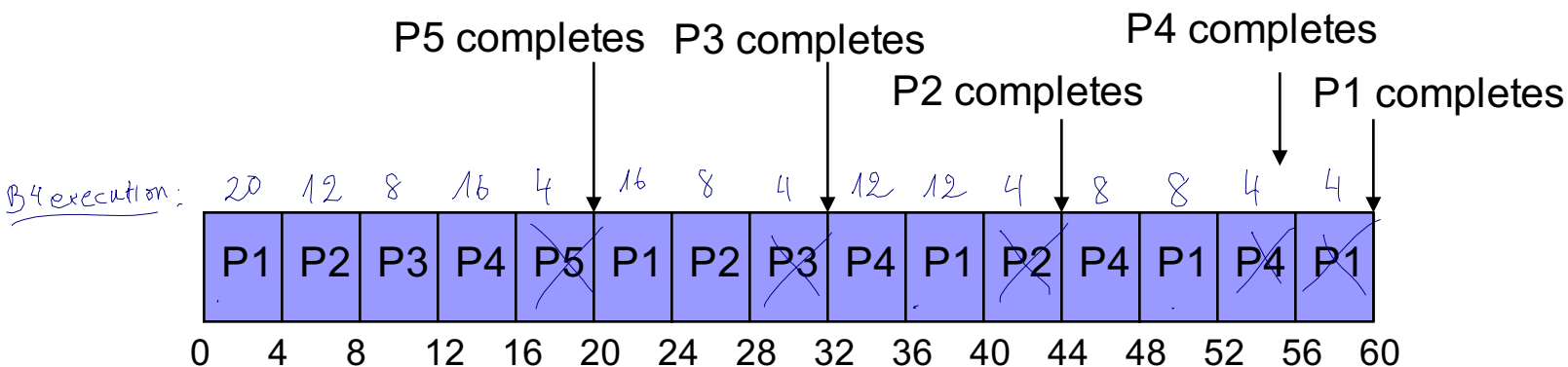
## - Scheduling Algorithms - Round-Robin Scheduling

Order of Queue and CPU Burst

P1: 20 P2: 12 P3: 8 P4: 16 P5: 4

Quantum = 4 Process runs until:

- a) Remaining CPU Burst < Quantum
- b) Remaining CPU Burst > Quantum  
Interrupt Context Switch to Next Process in Ready Queue.



### - Waiting times:

Average wait time: 30.4

P1:  $60 - 20 = 40$  (time when process complete minus all of its processing time).

P2:  $44 - 12 = 32$

P3:  $32 - 8 = 24$

P4:  $56 - 16 = 40$

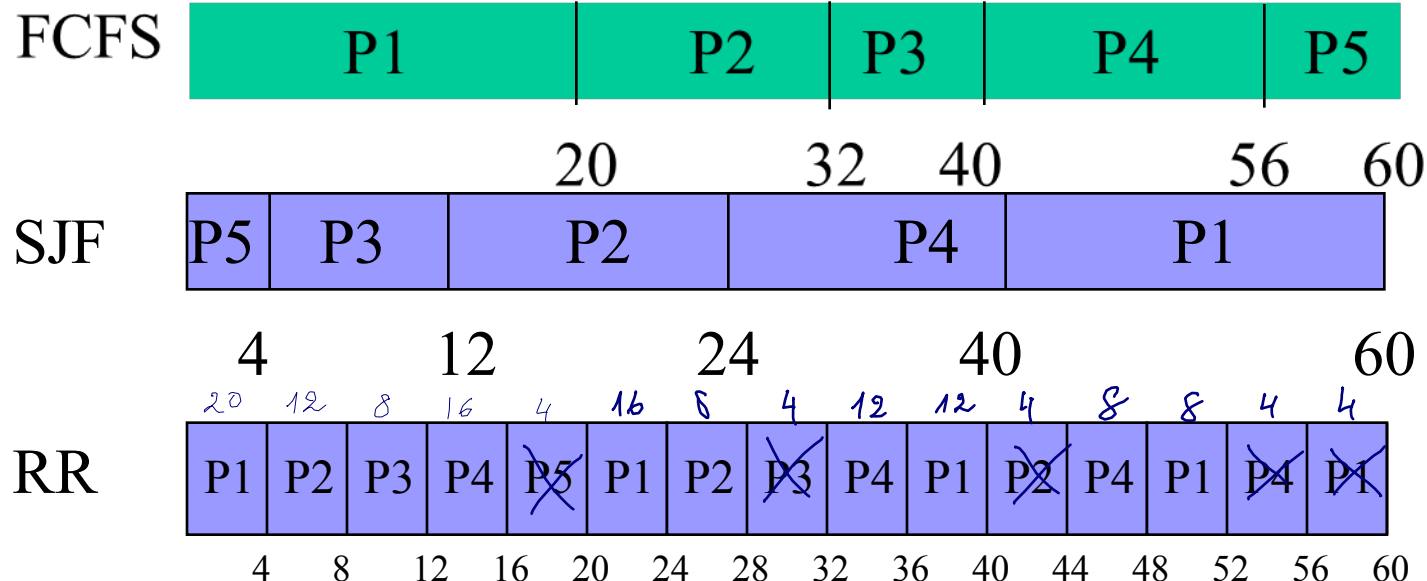
P5:  $20 - 4 = 16$

# CPU Scheduling

## Scheduling Algorithms

- Sample distribution based on **Nonpreemptive FCS and SJF**. **Preemptive RR**.
- Response time
  - Measure of time from the submission of a request until the time it takes to start responding.
  - Assume all jobs submitted at same time, in order given.
- Turnaround time
  - Process execution time. Time interval from submission of process until completion of process.
  - Assume all jobs submitted at same time

Order of Queue and CPU Burst  
 P1: 20 P2: 12 P3: 8 P4: 16 P5: 4



# CPU Scheduling

## ■ Scheduling Algorithms

## ■ Performance Characteristics of Scheduling Algorithms

- Round Robin ( RR )
  - good response time.
  - average waiting time
  - above average turn around time.
  - Average waiting time, often high, causing above average Turnaround Time.
    - Turnaround Time <sup>is</sup> good when CPU bursts <sup>is</sup> shorter than Quantum.
  - Quantum provides good response time.
    - Quantum too long, approach FCFS Policy behavior.
    - Quantum too short, causes additional Context Switch overhead,
    - Quantum 10 to 100 milliseconds.
  - Important for Interactive or Timesharing.
- Shortest-Job-First ( SJF ).
  - Best average waiting time.
  - CPU burst length or estimates difficult to predict.

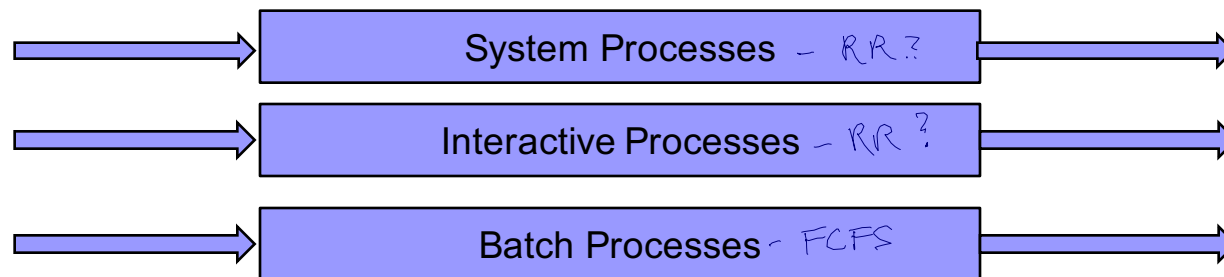
# CPU Scheduling

## ■ Scheduling Algorithms

### □ Multilevel Queue Scheduling

- Partition Ready Queue into several separate queues.
- Queues are defined based on process property: interactive process type, background process type, process priority.
- Each queue has its own Scheduling Algorithm.
  - Time-Sharing Queue scheduled by RR Algorithm.
  - Batch Queue scheduled by FCFS Algorithm.
- Each queue has priority:
  - No process in the lower priority queues will run until the higher priority queues has completed.
  - When process in higher priority queue is received, the lower priority queue processing is preempted.
  - Time-Slice between queues (prevents starvation).

- partition the Ready Q into several Qs  
- each Q has different function (interactive, batch)  
- each Q has its own scheduling algo, some uses RR (interactive), some uses FCFS (batch)  
- each Q has its own priority, preemption between Qs is allowed.



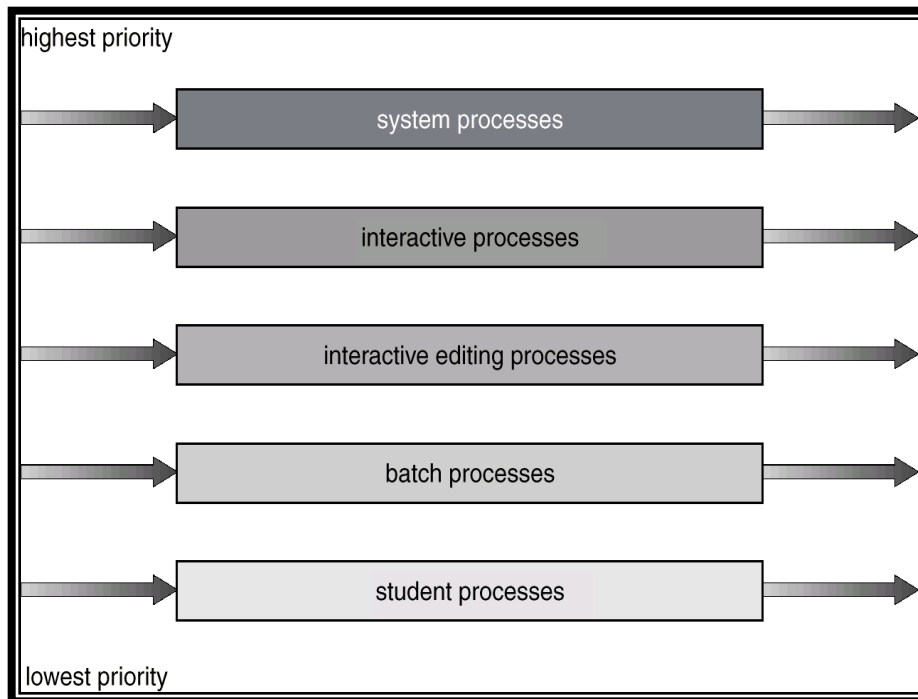


# CPU Scheduling

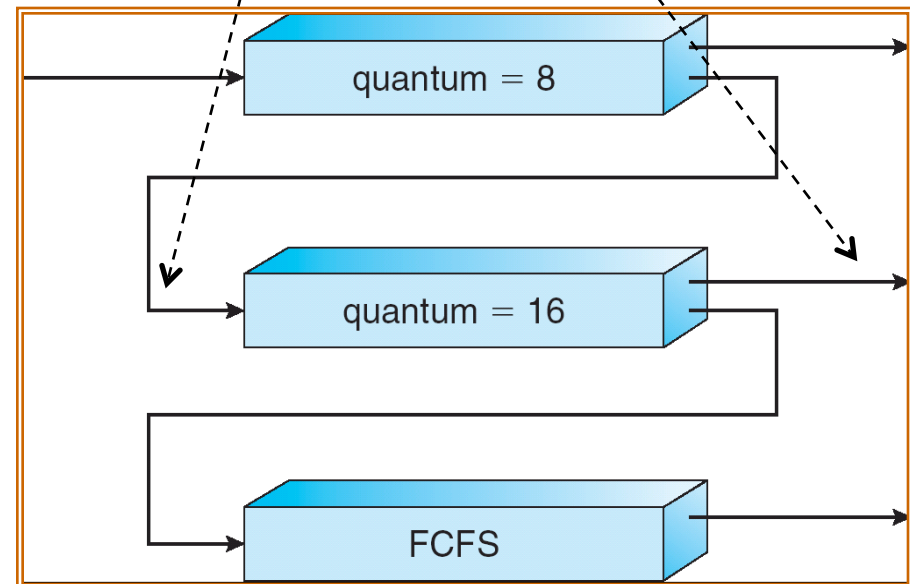
- Scheduling Algorithms
- Multilevel Feedback Queue Scheduling

Process exceeds quantum,  
placed in lower priority  
queue.  
Process not getting CPU,  
placed in higher priority  
queue.  
Scheduling Policy for each  
queue.

**Multilevel Queue (Fixed)**



**Multilevel Feedback Queue (Movement)**



# CPU Scheduling

## ■ Multiple-Processor Scheduling

## ■ Processor Affinity

*Stick a process to a single processor to prevent memory copy between different processors, reduce processing time.*

□ Cache Memory: Process prefers running on a specific CPU.

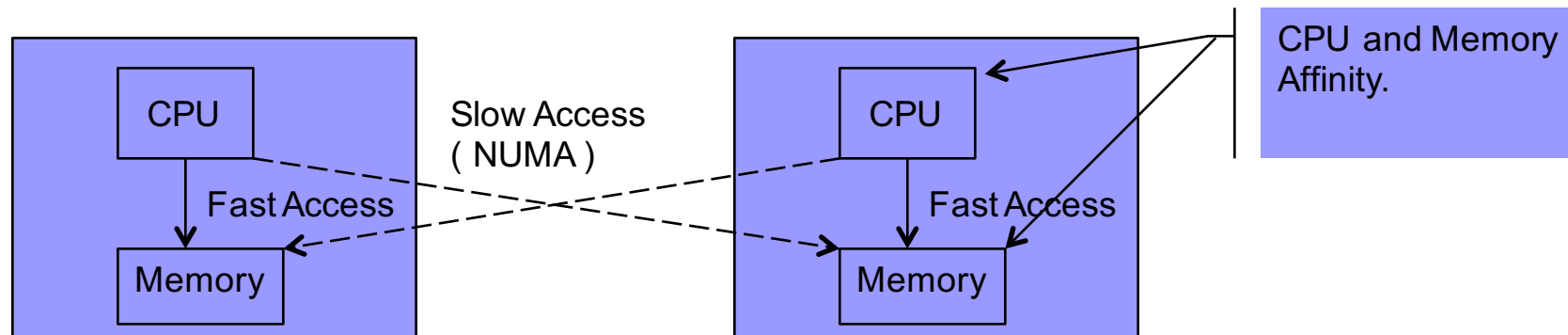
- Keep Cache Memory Validated.

- Soft Affinity: Operating System has policy to keep process on same CPU, but not guaranteed.

- Hard Affinity: Process cannot migrate to other CPU.

  - Linux, Solaris.

□ Main Memory Architecture: Non-Uniform Memory Access (NUMA) when memory access speed non-uniform.



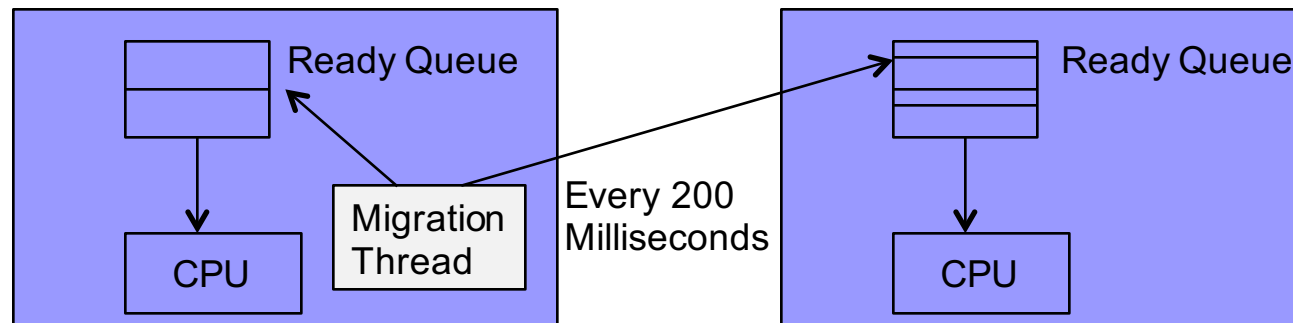
# CPU Scheduling

## ■ Multiple-Processor Scheduling

## ■ Load Balancing

- Keep CPU processing balanced.
  - Only when processors have separate CPU Ready Queues.
- Distributing processes between CPUs can counteracts Processor Affinity ( Imbalance determined by queue threshold ).
- Push Migration: Linux/FreeBSD → *by special processes*
  - Special process checks CPU Ready Queues and redistributes processes by pushing process from one CPU Ready Queue to another.
- Pull Migration: Linux/FreeBSD → *by (idle) CPU*
  - Idle processor pulls a waiting process from a CPU's Ready Queue.

*pull or push:  
depend on who  
does the work*



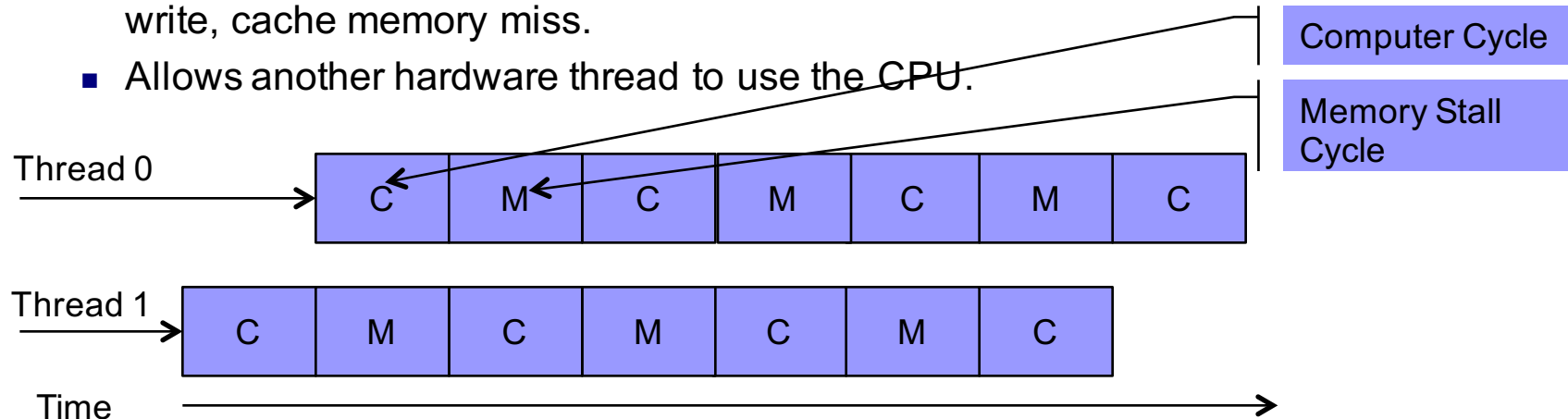
# CPU Scheduling

Multicore SMP > single core multiprocessor  
Energy lesser  
Speed. faster

## ■ Multicore Processors Scheduling

## ■ Multiple Processor Cores on Same Physical Chip

- Each core separate set of registers.
- Multicore SMPs are faster and consume less power than single core multiprocessor SMPs.
- **Multithreading processor cores**: Two or more hardware threads assigned to each core.
  - Memory Stall: A hardware thread will block when waiting for I/O to arrive or write, cache memory miss.
  - Allows another hardware thread to use the CPU.



# CPU Scheduling

## ■ Multicore Processors Scheduling

## ■ Each hardware thread is a logical processor.

- Dual-core dual-threaded CPU has four logical processors.

## ■ Multithreading Processor Implementation

+ coarse-grained:  
larger components. Simply  
wraps one or more fine-grained  
services together into a more  
coarse-grained operation

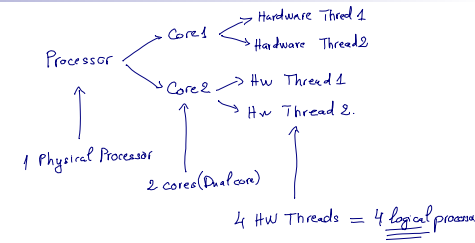
+ fine-grained:  
smaller components of which  
the larger are composed  
lower-level services

- Coarse-grained: Hardware thread executes until memory stall ( or long-latency event ) occurs. Processor now switch to another hardware thread. Cost of thread switching is high ( instruction pipeline has to be flushed ).

Fine-grained ( Interleaved ): Hardware thread switches at boundary of instruction cycle. All interleaved multithreading processor has hardware support for thread switching and is fast.

## ■ Two Levels of Scheduling:

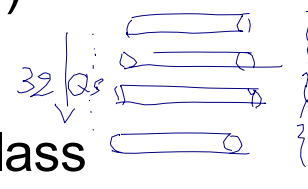
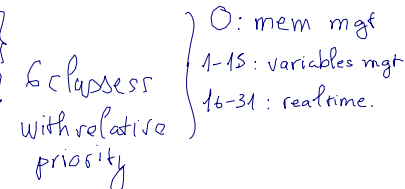
- Scheduling Algorithm by Operating System to select software thread to run.
- Each core decides which hardware thread to run.
- Round Robin (UltraSPARC T1).
- Events to trigger thread switch given priority (Intel Itanium).



# CPU Scheduling

- Virtualization and Scheduling
- Virtualization Software Schedules Physical CPU (Host OS) for Each Virtual Machine (Guest OS).
- Multiple Scheduling Layers All Sharing CPU Cycle
  - Host Operating System Scheduling.
  - Virtualization Software Scheduling Virtual Machine.
  - Virtual Machine Operating System Running Own Scheduler.
    - 100 Millisecond Virtual CPU time might take 1 second.
    - Poor response times to uses logged into Virtual Machine.
    - Time-of-day clocks are incorrect, timers take longer to trigger.
    - Make a Virtual Machine scheduling algorithm appear poor.

# CPU Scheduling

- Windows XP Operating System
- Priority-Based, Preemptive Scheduling
- Scheduling on the basis of threads
- Highest priority thread always be dispatched, and runs until:
  - Preempted by a higher priority thread
  - Terminates
  - Time quantum ends *← time sharing as well.*
  - Makes a blocking system call (e.g., I/O)
- 32 priorities, each with own queue:
  - 6 Classes with 6 Relative Priority per Class 
  - Mem. Mgmt.: 0; Variable: 1 to 15; Real-Time: 16 to 31 

# CPU Scheduling

- Windows XP Operating System
- Within Priority Class contains Relative Priority
- All Classes Except Real-Time are Variable
  - Thread Quantum Runs Out: Lower Priority.
  - Thread Released From Wait State (Keyboard or Disk I/O): Raise Priority.

**Priority Classes**

Relative Priority		Real-Time	High	Above Normal	Normal	Below Normal	Idle Priority
	Time-Critical	31	15	15	15	15	15
	Highest	26	15	12	10	8	6
	Above Normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Below Normal	23	12	9	7	5	3
	Lowest	22	11	8	6	4	2
	Idle	16	1	1	1	1	1



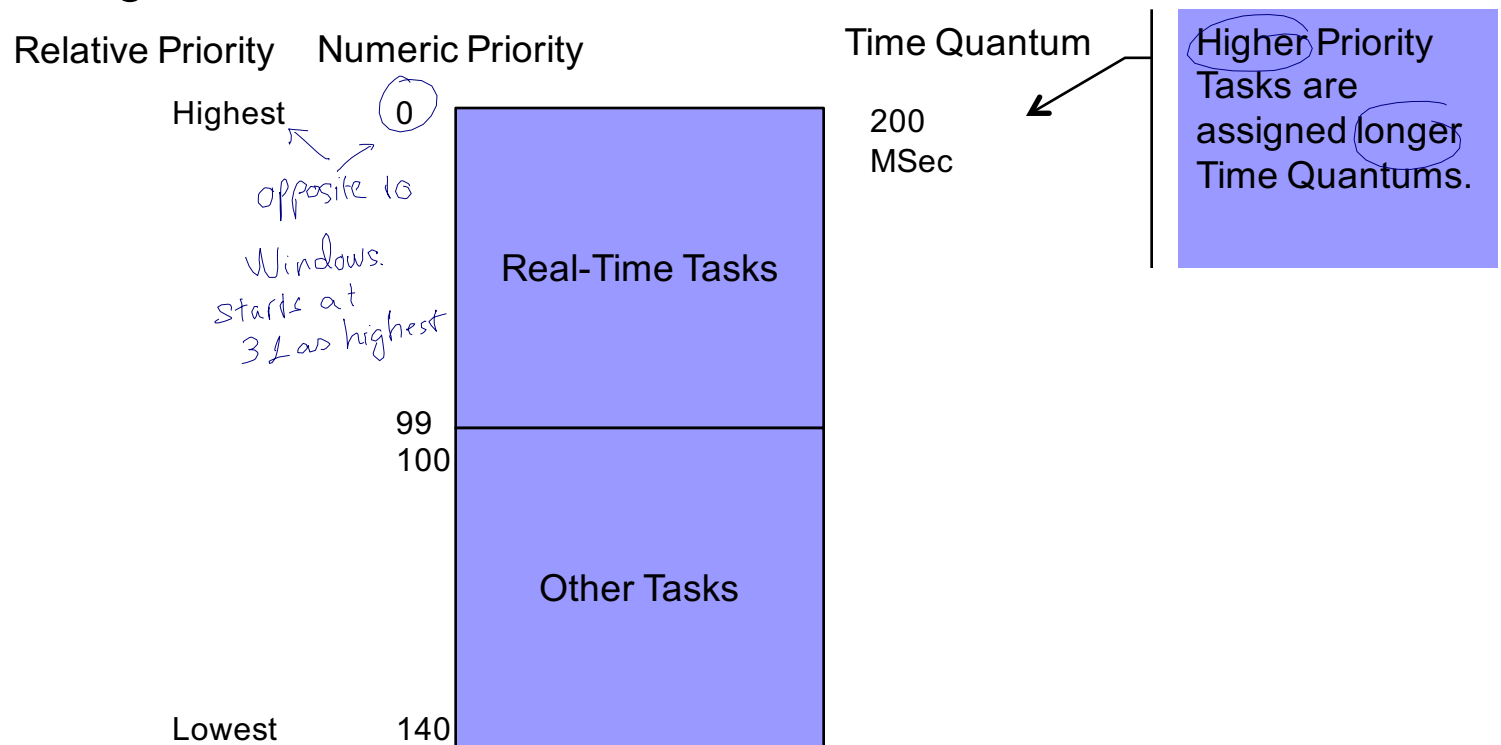
# CPU Scheduling

- Linux Operating System

- Preemptive, Priority-Based Scheduler (similar to WinXP)

- Real-Time Priority Range: 0 – 99 (only 16-31 in WinXP)

- Nice Range: 100 - 140

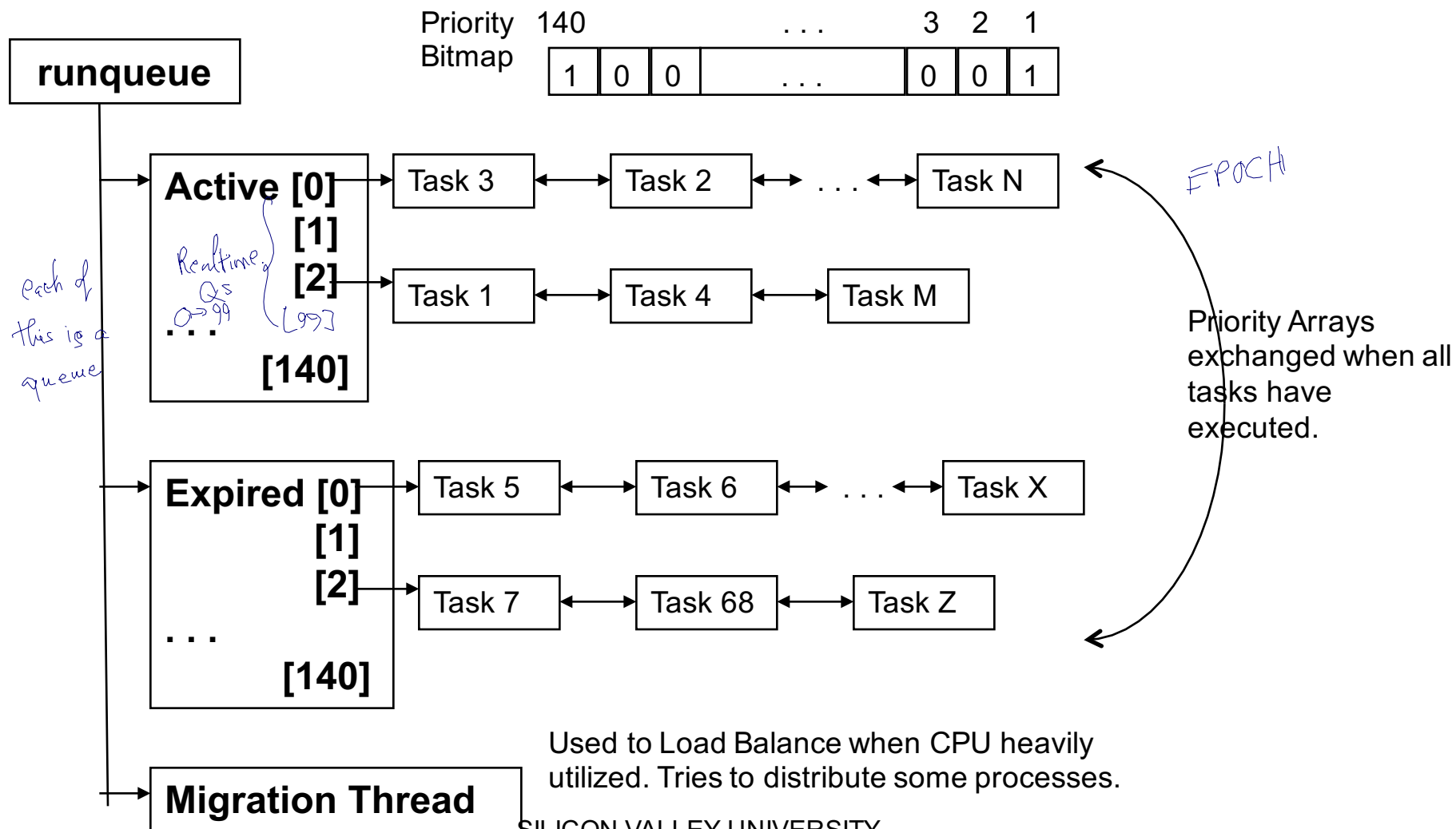


# CPU Scheduling

- Linux Operating System
- Real-Time Tasks Assigned Static Priorities
- Other Tasks Has Dynamic Priorities
  - Nice Value and how long task is sleeping.
  - Longer sleep time has higher priority adjustments ( typically I/O bound tasks ).
  - Shorter sleep time has lower priority adjustments ( typically CPU-bound tasks ).
- Runqueue Data Structure per CPU
  - Active Priority Array: All tasks with time remaining in Quantum.
  - Expired Priority Array: All tasks with expired Quantum.
  - Scheduler selects highest priority task from the Active Priority Array.
  - When Active Priority Array is empty (transitioned to Expired Priority Array or in Wait Queues), arrays are swapped.

# CPU Scheduling

## Linux 2.6 Scheduler Algorithm



# Process Synchronization

- Operating System Design central theme is the management of processes and threads in: **Multiprogramming**, **Multiprocessing**, **Distributed Processing**.

Distributed Processing is any computing that → involves multiple computers remote from each other have a role in a computational problem or info processing.

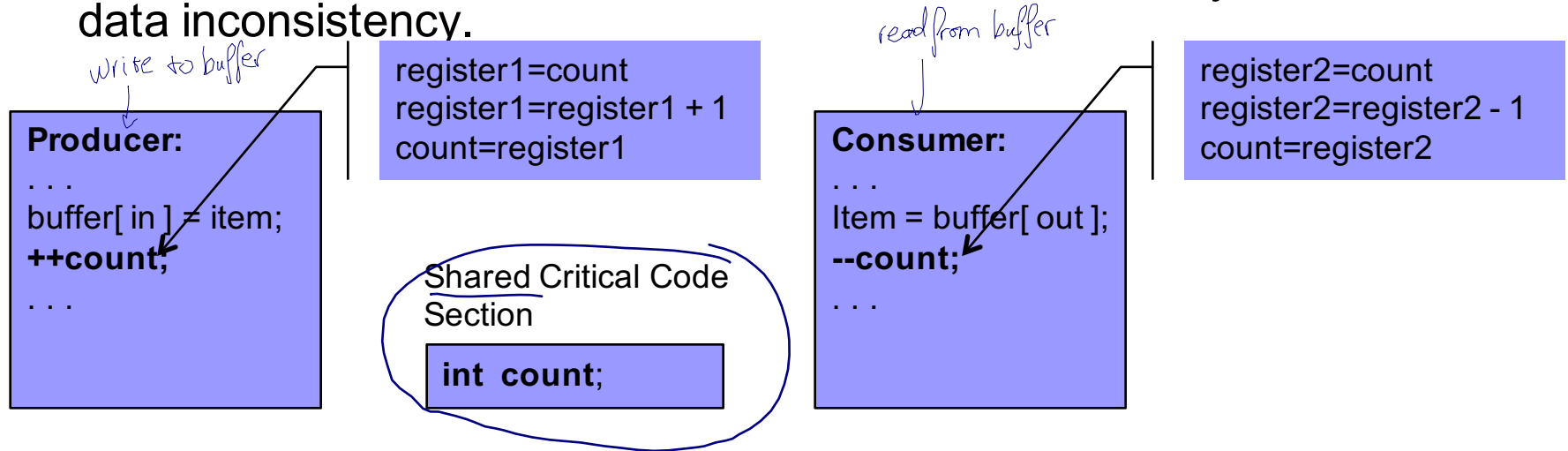
process one small part of each process to make sure time sharing, give an indication of multi programs running at the same time. The main idea is to maximize C.P.U. usage & keeps the CPU busy all the time.

refers to HW (CPU unit) more than SW (processes). If the HW has more than 1 processor then that is multiprocessing.

- Common to these areas is concurrency:
  - Process communication, resource competition, sharing of resources, synchronization of processes, and allocation of processor time to processes.
- **Process Synchronization** Issue is driven by cooperating processes utilizing shared address space and shared data through files or messages.
  - Processes or Threads running asynchronously and possibly sharing data.
    - **Problem Bounded Buffer**: Used by processes to share memory.
    - **Producer** and **Consumer** concurrent race condition accessing shared memory.
    - **Race Condition**: Several processes access and manipulates same data concurrently and outcome depends on the order in which access takes place.

# Process Synchronization

- **Race Condition:** Concurrent access to shared data may result in data inconsistency.



- Consumer and Producer processes do NOT block shared critical section ( memory location “count” ).

- |                |         |                           |                 |
|----------------|---------|---------------------------|-----------------|
| □ T0: producer | execute | register1 = count         | {register1 = 5} |
| □ T1: producer | execute | register1 = register1 + 1 | {register1 = 6} |
| □ T2: consumer | execute | register2 = count         | {register2 = 5} |
| □ T3: consumer | execute | register2 = register2 - 1 | {register2 = 4} |
| □ T4: producer | execute | count = register1         | {count = 6}     |
| □ T5: consumer | execute | count = register2         | {count = 4}     |

*in sequence*

*in parallel*

- Execute Separately, “count” = 5.
- Execute Concurrently, “count” = 4

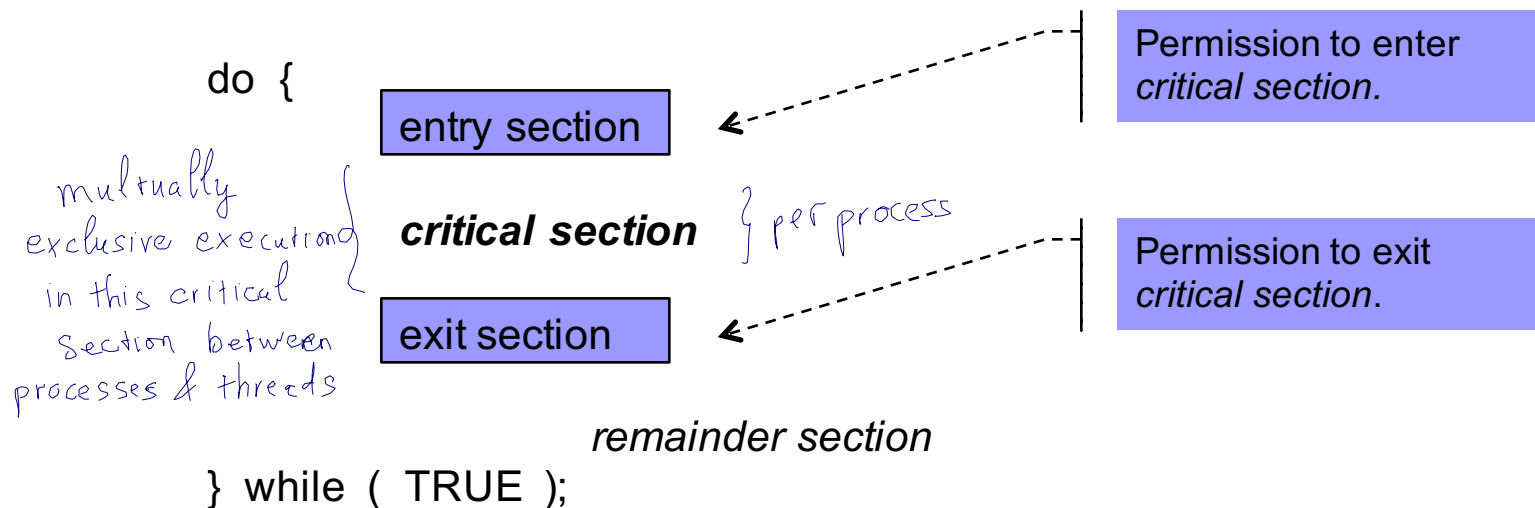
Shared critical section, “count”, changed at different times because interleaved execution with register contents save wrong values to “count”. “count” correct value is “5”.

# Process Synchronization

## ■ The Critical-Section Problem

### ■ Multiple processes sharing common resource (i.e. memory or file).

- Each process has Critical Section of code that will; change common variables, update shared table, write shared file.
- No processes are executing in their critical sections at the same time.



# Process Synchronization

- The Critical-Section Problem
- Solution Requirements: Cooperation Protocol Between Processes To Enter/Execute/Exit Critical Section
- Satisfy three requirements:
  - **Mutual Exclusion**: When process executes in its Critical Section, no other process must execute in their Critical Section.
  - **Progress**: Process requesting to enter their Critical Section can enter if no processes currently in their Critical Section.
  - **Bounded Waiting**: Bound or limit on the number of times other processes are allowed to enter their Critical Section, after a process has requested to enter its critical section.

# Process Synchronization

## ■ Semaphores

A semaphore is a variable or abstract data type that is used to control access, by multiple processes, to a common resource in parallel programming or a multiuser environment

## ■ Synchronization Tool Favored by Application Programs

## ■ Wait ( ) Atomic Operation

□ Originally "P": Mean "to test".

```
wait ( S ) {  
    while S <= 0; // no-op  
    S --;  
}
```

blocking

simplest wait to implement semaphore.

systems call in kernel (no userspace)

## ■ Signal ( ) Atomic Operation

□ Originally "V": Mean "to increment".

```
signal ( S ) {  
    S ++;  
}
```

start executing



# Process Synchronization

## ■ Semaphores

4 types of Semaphore :

- Counting : base on no. of available resources.
- Binary : ON/OFF
- Busy Waiting : loop endlessly .
- Spinlock : semaphore with Busy Waiting

### □ Counting Semaphores

- Control access to resource to finite instances of the resource.

- Wait ( S ) and Signal ( S ):

- "S" emaphore initialized to number of resources available.
- When Semaphore is zero, next process will Block.

### □ Binary Semaphores

- Semaphore is either "0" or "1".

- Provide Mutual Exclusion: Wait ( S ) and Signal ( S ).

- When Semaphore is zero, next process will Block.

### □ Busy Waiting ( Loop Continuously in Entry Code )

### □ Spinlock: Semaphore with Busy Waiting.

prev  
slide

- Thread "spins" on a processor.
- Used to wait short time for lock.

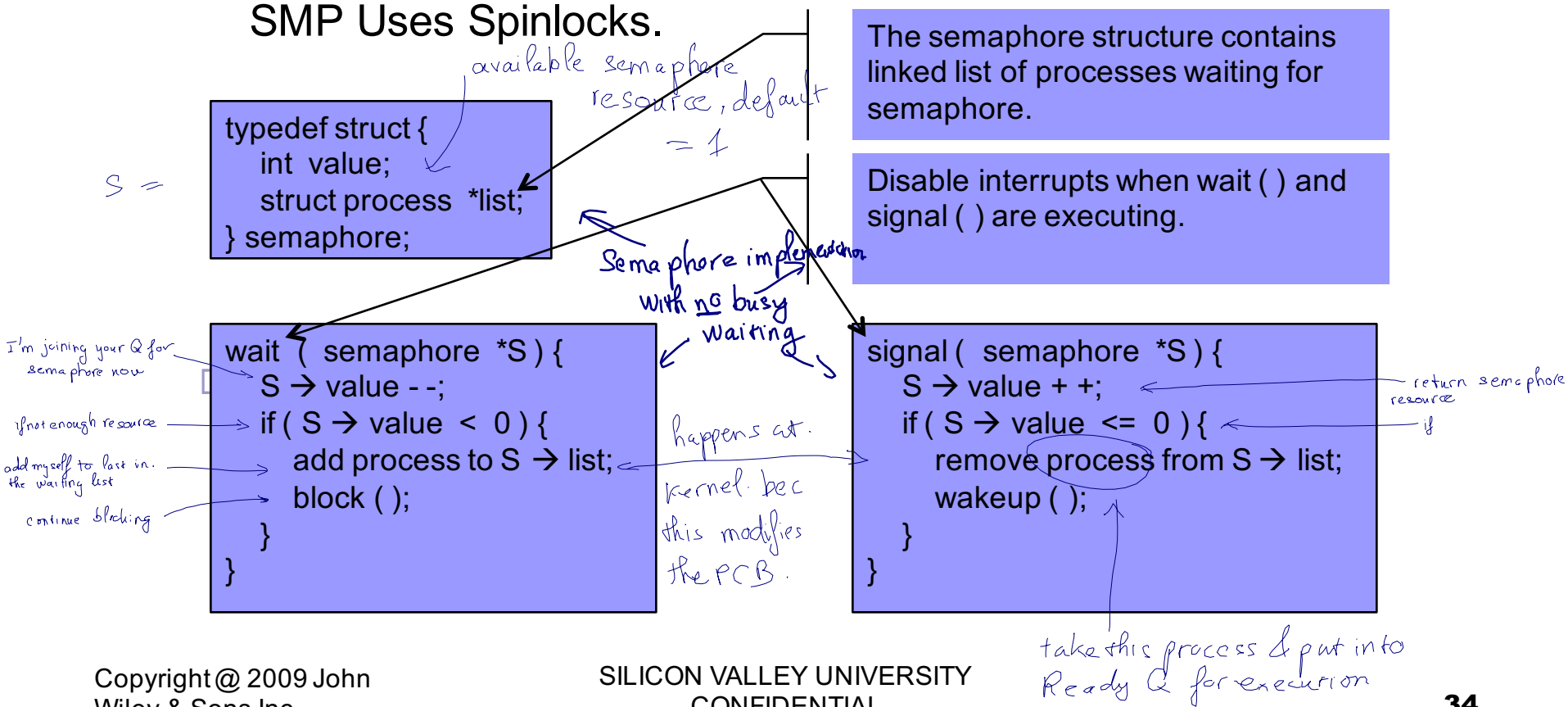
- Multiprocessor System: One processor has thread executing spinlock while thread on another processor performs Critical Section.

→ only work for multiprocessor  
system, not for single processor  
System

# Process Synchronization

## ■ Semaphores

- **block ( ) and wakeup ( ) instead of Busy Waiting**
- wait ( ) and signal ( ) must be Atomic.
- Multiprocessor environment: Overhead Disabling Interrupts, SMP Uses Spinlocks.



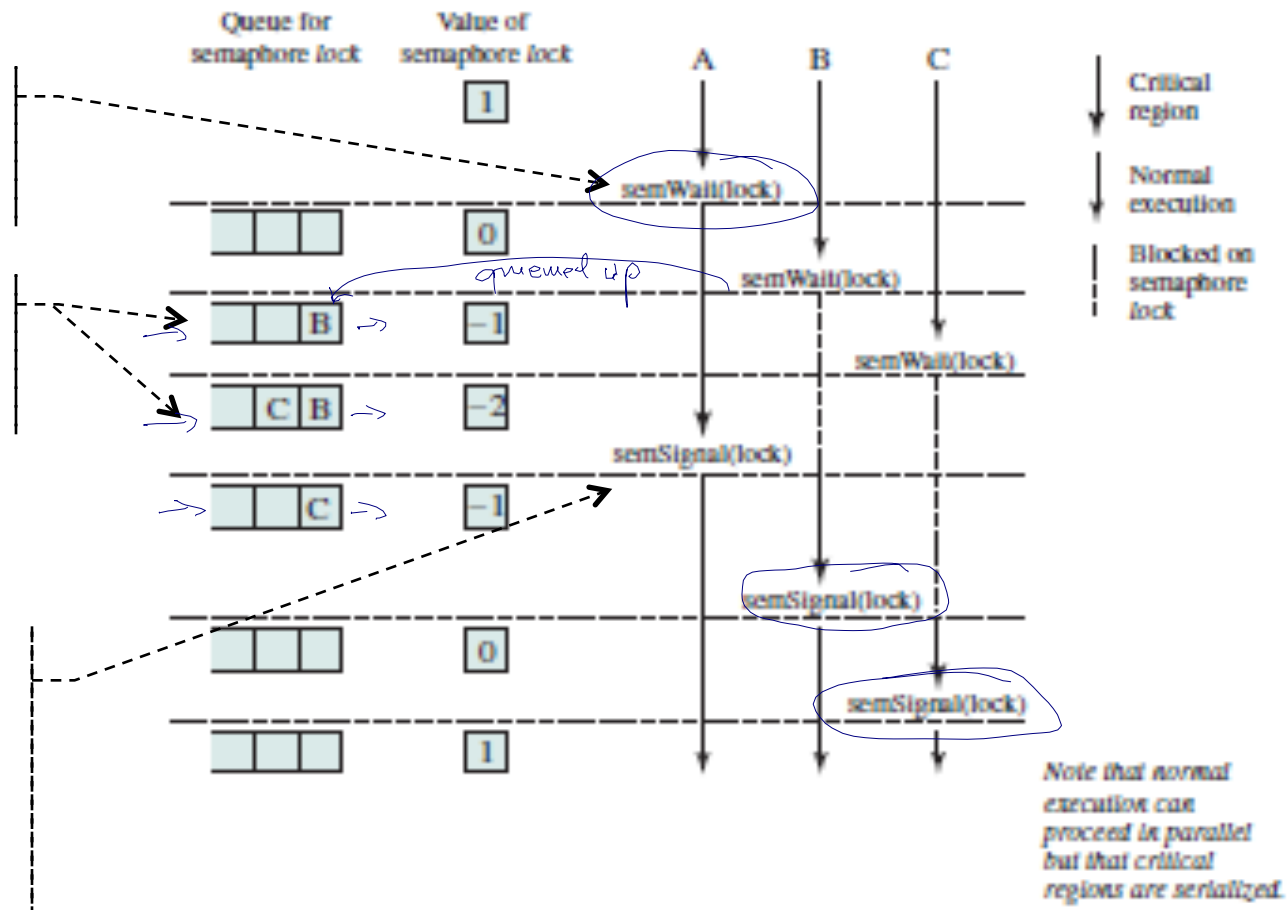
# Process Synchronization

- Semaphores
- Semaphore Queue Protecting Shared Data

Process A executes `semWait(lock)` and enters critical section.

Process B and C execute `semWait(lock)` and are blocked.

Process A exits critical section and executes `semSignal(lock)`. Process B placed in Ready Queue and acquires semaphore.



# Process Synchronization

## ■ Semaphores

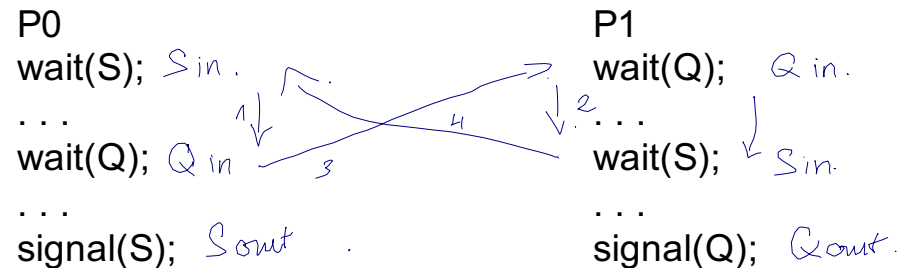
there are 3 potential problems when using semaphore:

## ■ Deadlocks

- Deadlock
- Starvation
- Priority Inversion

(semaphore)

- Deadlock when two or more processes waiting on event that is caused by one of the waiting processes.



## ■ Starvation

- Starvation or Infinite Blocking: Process wait indefinitely within Semaphore.

If

- List in Semaphore is processed LIFO ( Last-In, First-Out ) order.

## ■ Priority Inversion

- Higher priority task waiting on a semaphore held by a lower priority task. Before lower priority task release the semaphore, it is interrupted by another task (medium priority) and the higher priority task has to wait.

# Process Synchronization

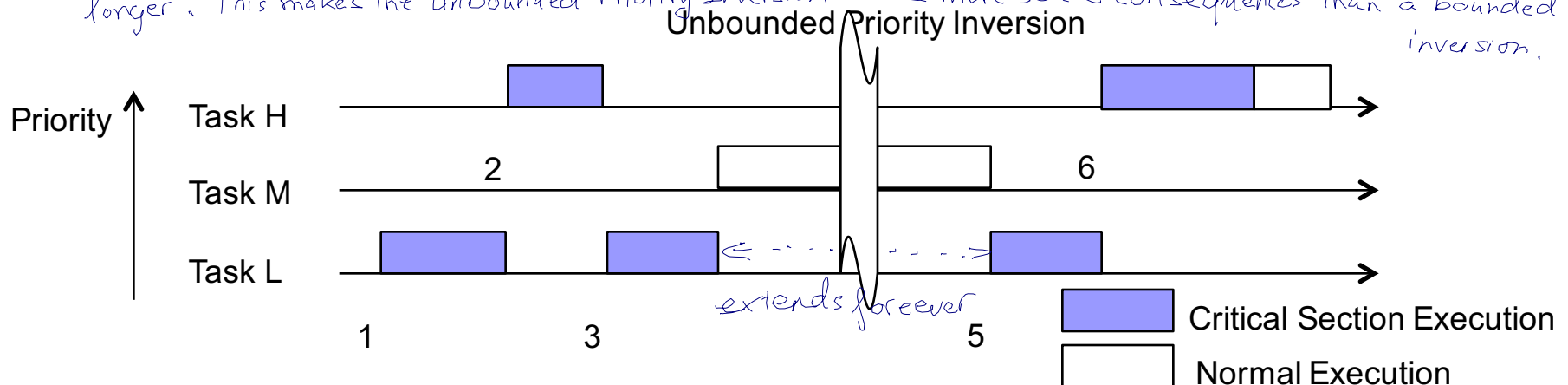
## ■ Priority Inheritance Protocol

*Solution of inverse priority protocol*

- All processes holding lock on resource needed by a higher priority process has the process's priority raised to the higher priority process.
- After process releases the Critical Section, it returns the process to its original priority.

## ■ Unbounded Priority Inversion.

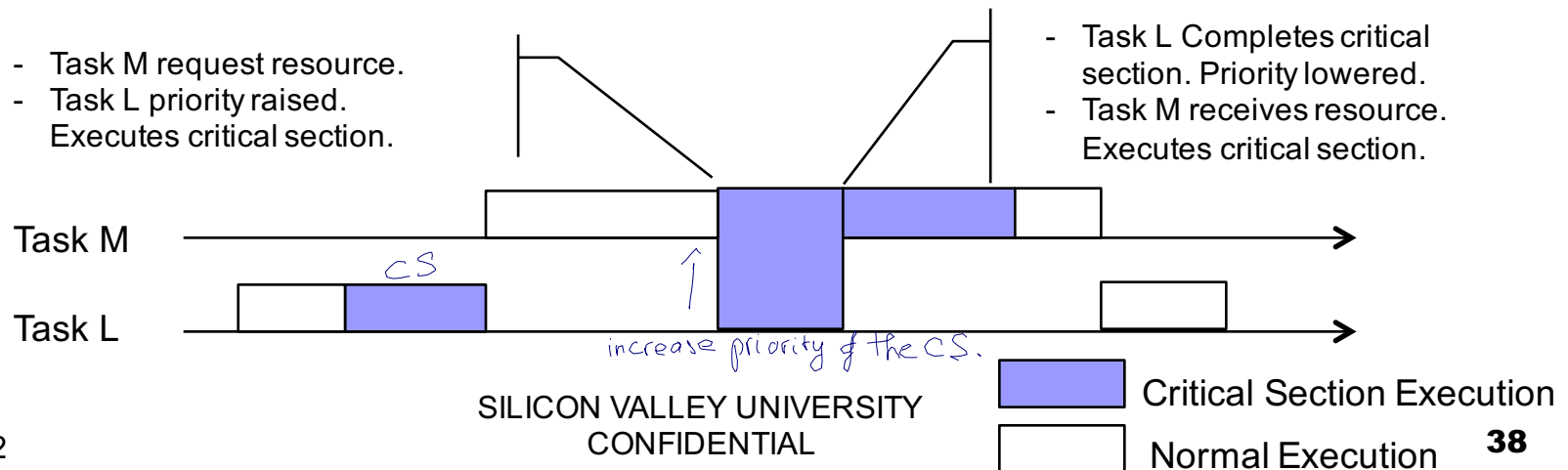
*This is similar to bounded priority inversion BUT the difference is task M takes a long time to finish its task, which extends the critical region task L, and therefore task H has to wait longer. This makes the Unbounded Priority Inversion cause more severe consequences than a bounded inversion.*



# Process Synchronization

## ■ Priority Inheritance:

- Low-priority task has priority raised until completes critical section (releases shared resource).
- Nested resource locks can lead to deadlock.
  - Avoid deadlock by allowing each task to own one shared resource.
  - Do not allow nested locks.
  - Overhead raising priority and then lowering priority. ?



# Process Synchronization

Mutex = Reader Q.  
WRT = Writer Q. } using non-busy waiting semaphore implementation in prev slide. (wait/signal)

## ■ The Reader-Writer Problem: Concurrency/Shared Data

□ Writer Exclusive Access, Reader Shared Access.

1 reader → Semaphore mutex ( 1 ) mutual exclusion for "readcount" ( 0 ).  
share with 1 writer Semaphore wrt ( 1 ) mutual exclusion for writers.

### Reader Process

```
do {
    wait ( mutex );
    readcount++;
    if ( readcount == 1 )
        wait ( wrt );
    signal ( mutex );
    ...
    // Reading is Performed
    ...
    wait ( mutex );
    readcount--;
    if ( readcount == 0 )
        signal ( wrt );
    signal ( mutex );
} while ( TRUE );
```

check the reader Q if no other reader is in the Q, proceed. If not, Q up & halt, wait for writer signal, update "readcount".  
If I'm the first reader  
If none is in writer Q, proceed. Otherwise, get in line & wait for the writer to finish & then wake it up.  
Release readers lock & launch next reader already in Q.  
Proceed or get in line until its turn to wake up to update "read count".  
Decrease "read count".  
If no more reader  
then wake up the next writer (if any).  
release reader lock & let the next reader decrease the "readcount".

Readers reading database locks out Writer.

all readers complete reading.

clear locking

- 1) First Reader wait(mutex) to update "readcount". Calls wait(wrt) to lock out writer.
- 2) Readers can share database access, wait(mutex) to update "readcount".
- 3) Last Reader calls signal(wrt) to allow writer to write database.

### Writer Process

```
do {
    wait ( wrt );
    ...
    // Writing is Performed
    ...
    signal ( wrt );
} while ( TRUE );
```

Writer writes to database, Readers are locked out.

# Process Synchronization

- Synchronization in Solaris
- Adaptive Mutex (Reader)

Spinlock: a semaphore type that causes the processes to 'spin' (looping) while waiting for resource to execute

- On Multiprocessor system Adaptive Mutex initially <sup>is</sup> a Spinlock.

- If thread holding the lock not running on another processor, the mutex blocks.

When the Adaptive Mutex requires a lock held by other thread:  
- if other thread is running on another processor, the mutex will spin & wait for that thread to finish (as it will finish soon)  
- if other thread is sleeping, the mutex will block (sleep) since that thread will not finish any sooner.  
i.e. never uses a spin lock when waiting

- On single processor system, Adaptive Mutex always blocks.

to protect a critical section.

- Critical code more than a few hundred instructions uses semaphores (block).

(non busy waiting)

- In short code segments, spinlock cost (putting thread to sleep, waking it, and context switch) is less than blocking.

\* Short code segment:  
Spin lock cost < Blocking (context switch, sleep, wake up)  
\* Long code segment:  
Spin lock cost > Blocking cost

- Reader-Writer Locks: Access protected data frequently, but in read-only manner. Concurrent access, semaphores are serial access. Expensive to implement → only used in long code segment.

- TurnStile (Queue) maintained by first blocked process for Adaptive Mutex and Reader-Writer Locks.

Turnstile is a Q of all processes waiting for access to a lock obj (Adaptive Mutex / Reader-Writer locks). It also implements the priority inheritance model to avoid Priority Inversion problem.

has different strategies on different CPU types.

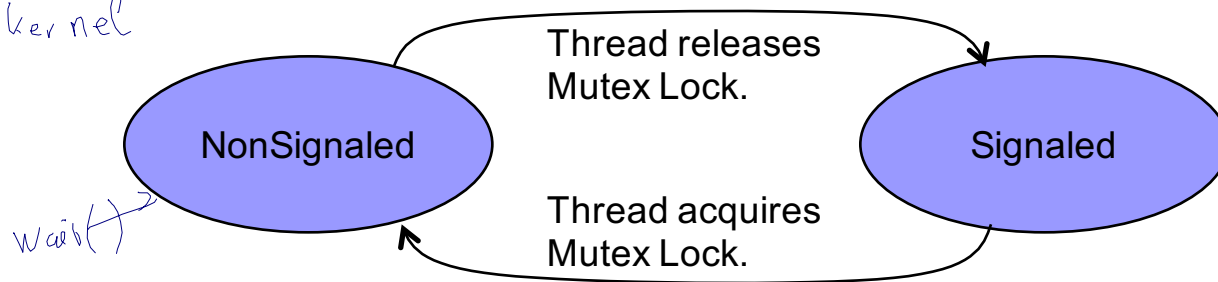
Spinlock is only efficient with short code segment (less than a few hundreds of instructions). More than that, condition variables or semaphores (block) should be used.



# Process Synchronization

- Synchronization in Windows XP
- Uniprocessor System, Windows XP kernel accesses a global resource, will temporarily <sup>disable</sup> mask interrupts.
- Multiprocessor System, Windows XP uses spinlocks for short code segments ( Solaris ).
- Thread Synchronization, Windows XP uses Dispatcher Objects for mutexes, semaphores, events, and timers.

- Signaled State: Dispatch object is available.
- Nonsignaled State: Dispatch object is not available and access will block.



Transition to Signaled, a mutex will select one thread from waiting queue. When event object, all threads selected from waiting queue.

*because of 'mutex'!*

# Process Synchronization

## ■ Synchronization in Linux

## ■ Uniprocessor Systems, Linux Enables and Disables Kernel preemption.

- `preempt_disable ( )` : *disable interrupt → when acquire the lock*
- `preempt_enable ( )` : Kernel is **not preemptable** when kernel-mode process holding a lock. *→ when release the lock*

- `preempt_count` in thread-info structure to count number of locks.

## ■ Multiprocessor Systems, Spinlocks for short durations.

Single Processor	Multiple Processor
Disable Kernel Preemption	Acquire Spinlock
Enable Kernel Preemption	Release Spinlock

## ■ Synchronization in Pthreads API

- **Mutex Locks** (fundamental synchronization technique), **Condition Variables**, **Read-Write Locks** for Thread Synchronization.

# Deadlocks

## ■ Deadlock Problems Becoming More Common:

- Number of processes contending.
- Multithreaded programs.
- Many resources types within system with multiple instances.
  - Memory, CPU cycles, Files, I/O Devices ( printers, DVD drives ).

## ■ System Model *processes must use acquire() & release() to obtain & release SHARED resource*

- Resources shared between Processes must use acquire() and release() System Calls to request resource / release resource.
- Process utilize resource using Normal Mode of Operation:
  - Request ( open ( ), allocate ( ) ): Granted or wait for resource.
  - Use: Operate on the resource.
  - Release ( close ( ), free ( ) ): Release the resource.
- System Table used for recording / queuing:
  - Records resource free or allocated (process allocated).
  - Queue of processes waiting for resource.

## ■ Deadlock State: Every process in the set is waiting for an event that can be caused only by another process in the set.

# Deadlocks

## ■ Deadlock Characterization

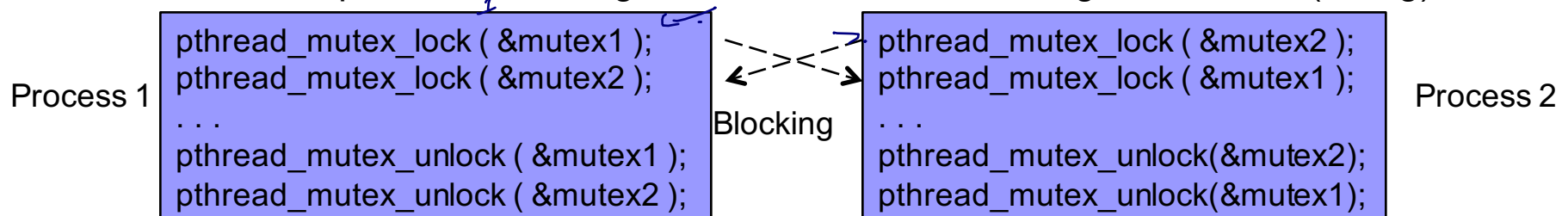
## ■ In Deadlock:

- Processes Never Finish Executing, and System Resources are Tied Up, Preventing Other Jobs from Starting.

## ■ Deadlock Caused by Four Conditions Held Simultaneously:

### □ Mutual Exclusion:

- One process is holding resource and others waiting for resource (timing).



### □ Hold and Wait

- Hold one resource and acquiring another held by other processes.

### □ No Preemption

- A resource released voluntarily by process holding it.

### □ Circular Wait

- A set of waiting processes, where each process is waiting on process before it.

# Deadlocks

## ■ Deadlock Characterization

## ■ Resource-Allocation Graph

*to detect if deadlocks base on current state of the processes / semaphores*

### □ Directed Graph:

- Request Edge: Process  $P_i$  request an instance of Resource  $R_j$ .
- Assignment Edge: Resource  $R_j$  allocated to Process  $P_i$ .

Processes = {  $P_1, P_2, P_3$  }

Resources = {  $R_1, R_2, R_3, R_4$  }

Edges = {  $P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3$  }

P1:

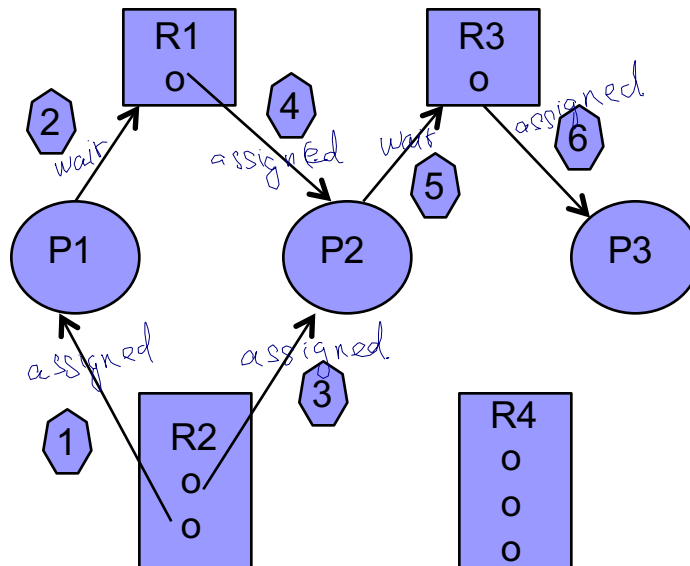
- 1) Allocated instance of R2.
- 2) Request instance of R1.

P2:

- 3) Allocated instance of R2.
- 4) Allocated instance of R1.
- 5) Request instance of R3.

P3:

- 6) Allocated instance of R3.



- 6) P3 release R3.
- 5) R3 allocated to P2.
- 4) P2 release R1.
- 2) R1 allocated to P1

*no deadlock yet*

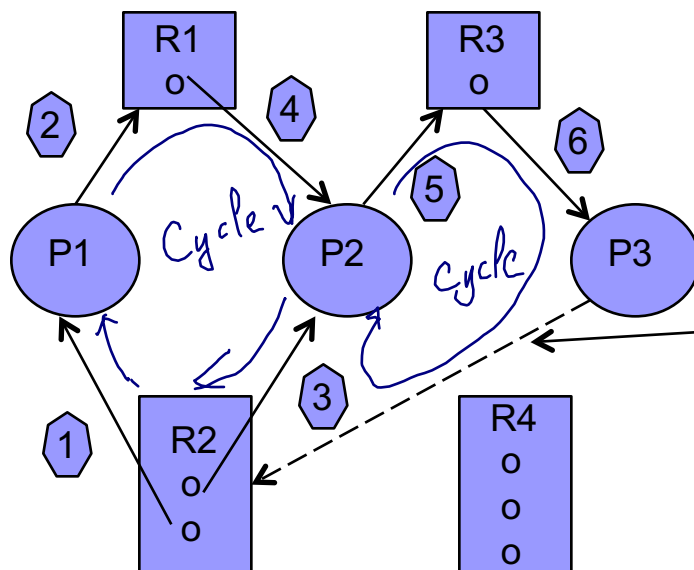
# Deadlocks

HOW TO ANALYSE  
RESOURCE ALLOC GRAPH

\* Cycle + One Instance of Resource  
⇒ Deadlock.  
\* Cycle + 2+ Instances of Resource  
⇒ No deadlock yet

## ■ Resource-Allocation Graph

- Cycle: For Resource Type with **One Instance**
  - Deadlock.
- Cycle: For Resource Type with **2+ Instances**
  - Does not imply Deadlock.



deadlock occurs when P3 tries to request for R2

P3 requests instance of R2. R2 has two instances, but both allocated to Process P1 and P2. P3 will block, but will deadlock ( two cycles ) because P1 is waiting for R1 and P2 is waiting R3.

# Deadlocks

## ■ Methods for Handling Deadlock

## ■ To ensure Deadlock cannot occur, Operating System can use:

- Deadlock Prevention – Set of methods for ensuring at least one of the necessary deadlock conditions cannot hold.

putting constraints  
on resources  
usage so at  
least 1 cond of  
deadlock can't  
hold, therefore  
low device usage  
and low  
thruput

- **Constraining** how requests for resources can be made.

- Deadlock condition when four simultaneous conditions occur:

- 1) Mutual Exclusion
- 2) Hold and Wait
- 3) No Preemption
- 4) Circular Wait

- Low device utilization and reduced system throughput.

- Deadlock Avoidance – Operating System has advance information on resources a process will request and use.

The OS reviews current resource  
usage & forecast future resource  
request, future release, using  
some algorithm to construct the  
resource alloc graph to avoid  
dead lock in advance.

- Operating System can decide for each request whether or not the process should block.

- Consider resources available, allocated, future requests, and future releases (each process declares max number of resources of each type it will use).

- Algorithm can be constructed with a priori information to form resource-allocation graph.

# Deadlocks

- Methods for Handling Deadlock
- Deadlock Prevention of Four Necessary Conditions
  - Mutual Exclusion
    - Occurs when using Mutual Exclusion for nonsharable resources.
    - Cannot deny Mutual-Exclusion because some resources are nonsharable.
    - Sharable resources do not require Mutual Exclusion.
  - Hold and Wait
    - Prevent Deadlock, whenever a process requests a resource, it does not hold any other resources.
      - Require process to request and be allocated all resources before exe.
        - Resources are allocated and unused for long periods of time.
        - Starvation caused by resources allocated to other processes.
  - No Preemption
    - Deadlocks occur when there is no preemption of resources that have already been allocated.
      - Possible prevention require releasing all resources when another resource cannot be allocated to it (process must wait). Try to allocate all resources again.
      - Technique used for CPU registers and memory allocation.

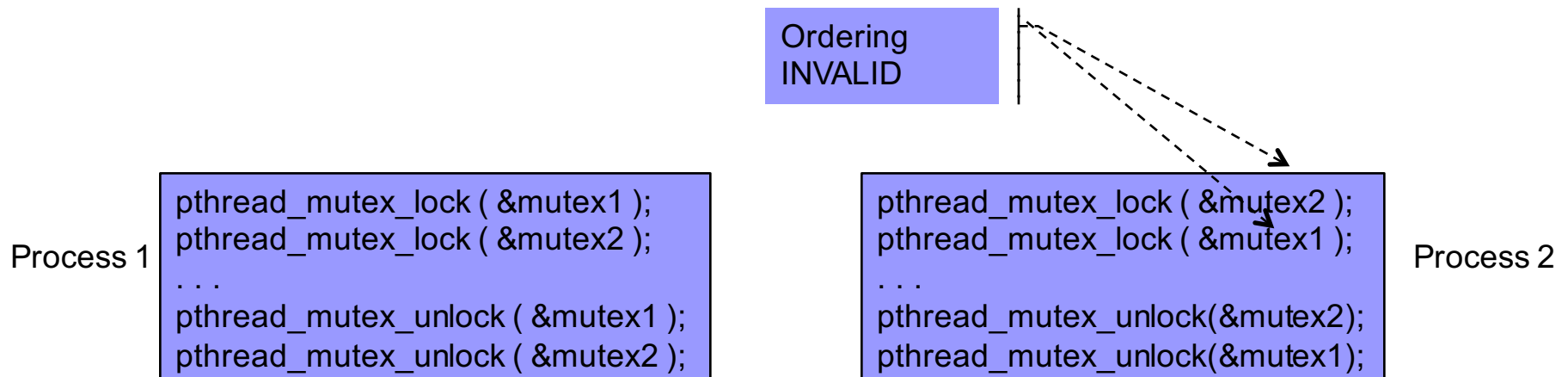


# Deadlocks

- Methods for Handling Deadlock
- Deadlock Prevention of Four Necessary Condition

- Circular Wait

- Prevent Deadlock by imposing ordering of all resources and each process must request resources in an increasing order of enumeration.
- Resources acquired in proper order responsibility of software programs.
- Mutual Exclusion locks to protect order in the system.



# Deadlocks

## ■ Methods for Handling Deadlock

## ■ Using Deadlock Avoidance

- Algorithm uses information on how resources will be requested.
  - Process1 requests in order R1 and R2. Process2 requests in order R2 and R1. Process1 or Process2 should wait, based on:
    - The resources currently available.
    - The resources currently allocated to each process.
    - Future requests and releases of each process.
  - Algorithm: Each process declare the maximum number of resources.
    - Given a priori information, algorithm to avoid Deadlock.
- Algorithm dynamically examines resource-allocation state.
  - Resource-allocation state is the number of available and allocated resources and the maximum demands of the processes.

PC system : unlikely

Production / High end : may implement this technique

# Deadlocks

## ■ Methods for Handling Deadlock

## ■ To Detect Deadlock, Operating System can use:

### □ Deadlock Detection and Recovery

- Algorithm examine state of the Operating System to determine whether a deadlock has occurred.
- An algorithm to recover from the deadlock.

### □ Manual Recovery Methods

- System performance deteriorates.
- Unable to identify cause of deadlock.
- Possible used instead of Deadlock Prevention and Deadlock Avoidance techniques.

## ■ Detection-and-Recovery Scheme Overhead:

- Run-time costs of maintaining the necessary information.
- Executing the detection algorithm.
- Losses (i.e. process execution, blocked resources) inherent in recovering from Deadlock.

# Deadlocks

## ■ Methods for Handling Deadlock

- Which Method Used: Manual Recovery *used.*  
- most of the time  
- least expensive  
- fast
- Deadlocks occur infrequently.
- **Less expensive** (implementation, extra resources) than Prevention, Avoidance, or Detection and Recovery Methods.
  - Overhead in run-time cost of maintaining the necessary information.
  - Detection algorithm must run continuously.
- Other system problems can resemble deadlock condition, and Manual Recovery need to be used anyway.

# CPU Scheduling

end on June 2nd.

## ■ Summary

- CPU Scheduling
  - Scheduling Algorithms: FCFS, SJF, Priority, RR, Multilevel Queue
  - Multiple-Processor Scheduling: Load Balancing, Virtualization
  - Computer System Schedulers: Windows XP, UNIX/Linux
- Process Synchronization
- Race Condition: Concurrent Execution
- Synchronization Hardware
- Semaphores: Counting, Mutexes, Spinlocks
- Deadlock Characteristic
- Methods for Handling Deadlocks.
  - Deadlock Prevention, Avoidance, Detection.
- Recovery from Deadlock.
  - Manual Recovery.
  - Automatic Recovery.
    - Process Termination
    - Resource Preemption