CSMC 412

Operating Systems

Prof. Ashok K Agrawala

Set 5

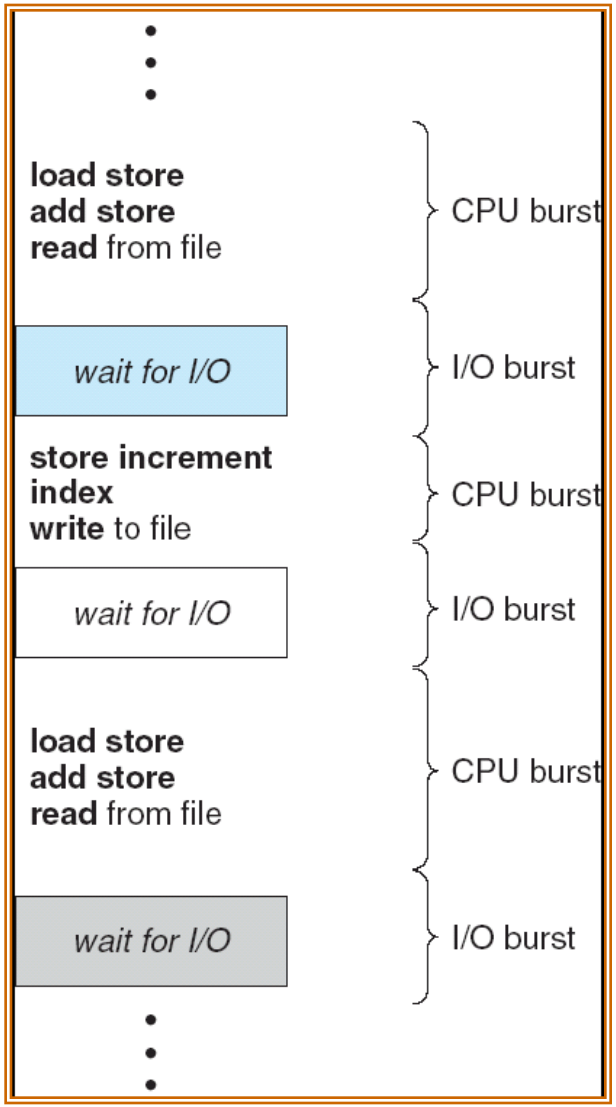
CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Real-Time Scheduling
- Thread Scheduling
- Operating Systems Examples
- Java Thread Scheduling
- Algorithm Evaluation

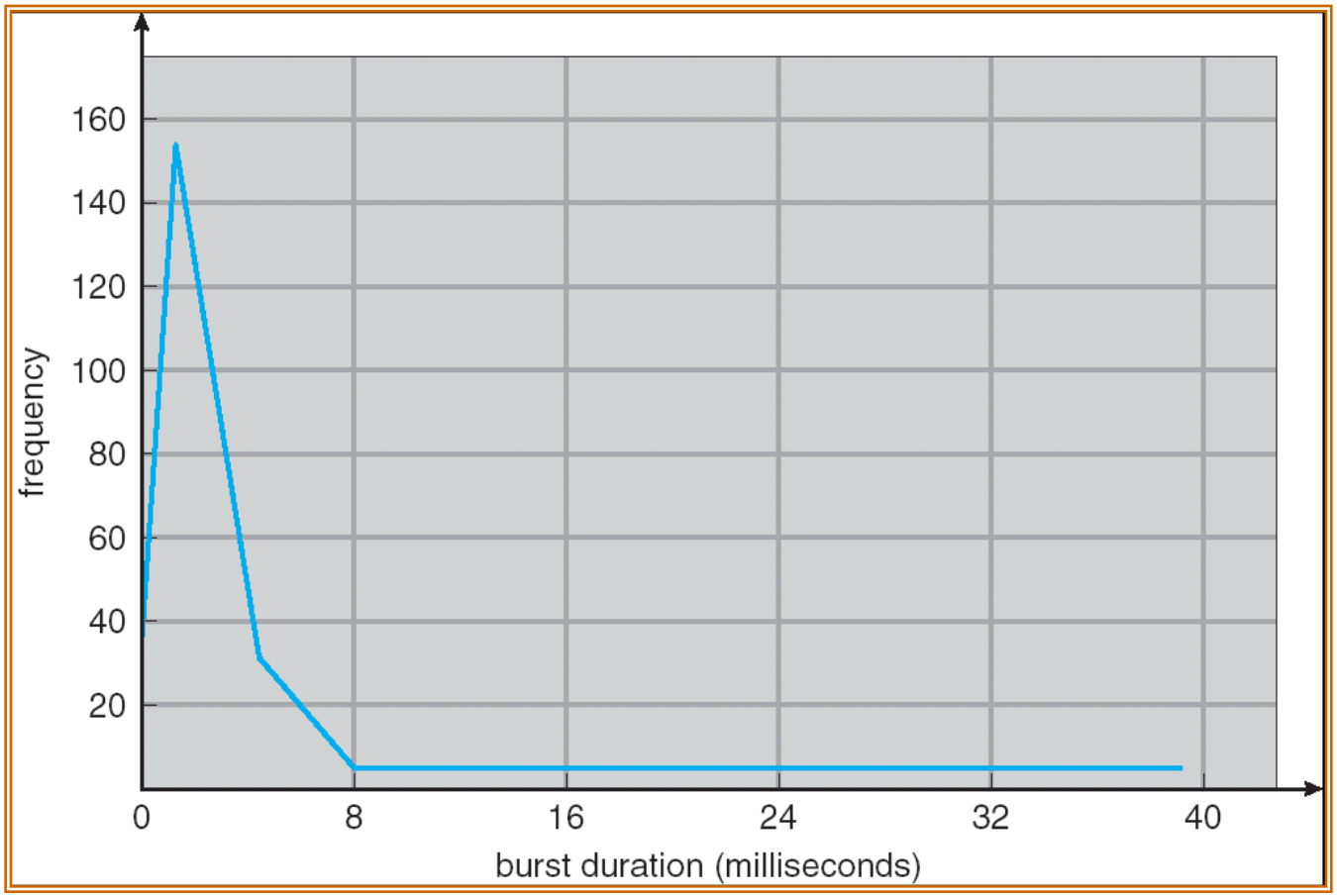
Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait
- CPU burst distribution

Alternating Sequence of CPU And I/O Bursts



Histogram of CPU-burst Times



CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is *nonpreemptive*
- All other scheduling is *preemptive*

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- *Dispatch latency* – time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- Waiting time – amount of time a process has been waiting in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)

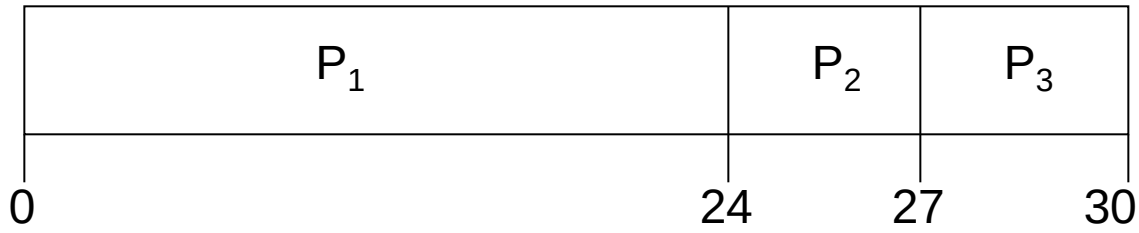
Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:

- Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- *Convoy effect* short process behind long process

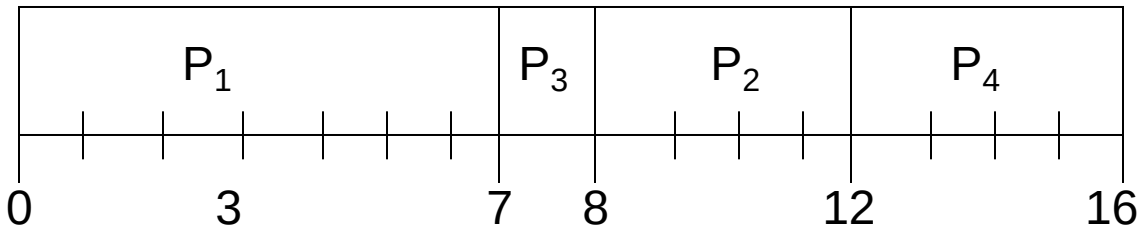
Shortest-Job-First (SJR) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the
    Shortest-Remaining-Time-First (SRTF)
- SJF is optimal – gives minimum average waiting time for a given set of processes

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

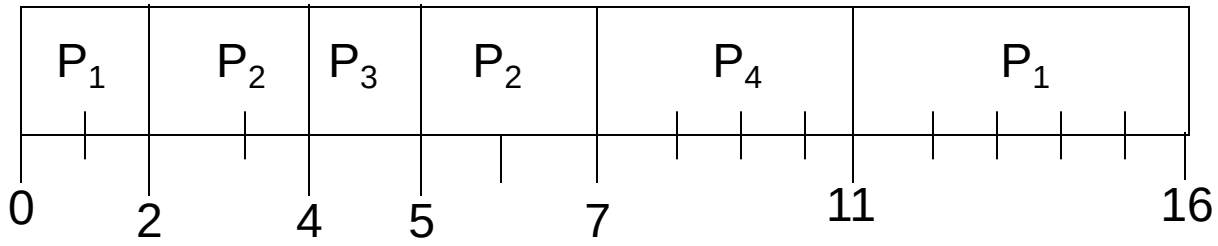


- Average waiting time = (0 + 6 + 3 + 7)/4 - 4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

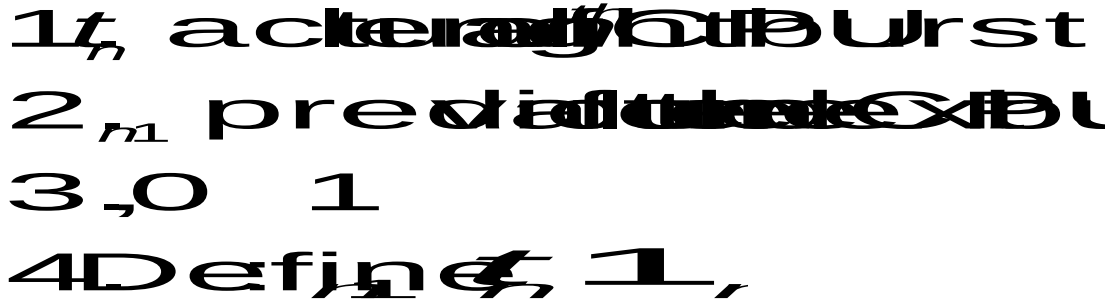
- SJF (preemptive)



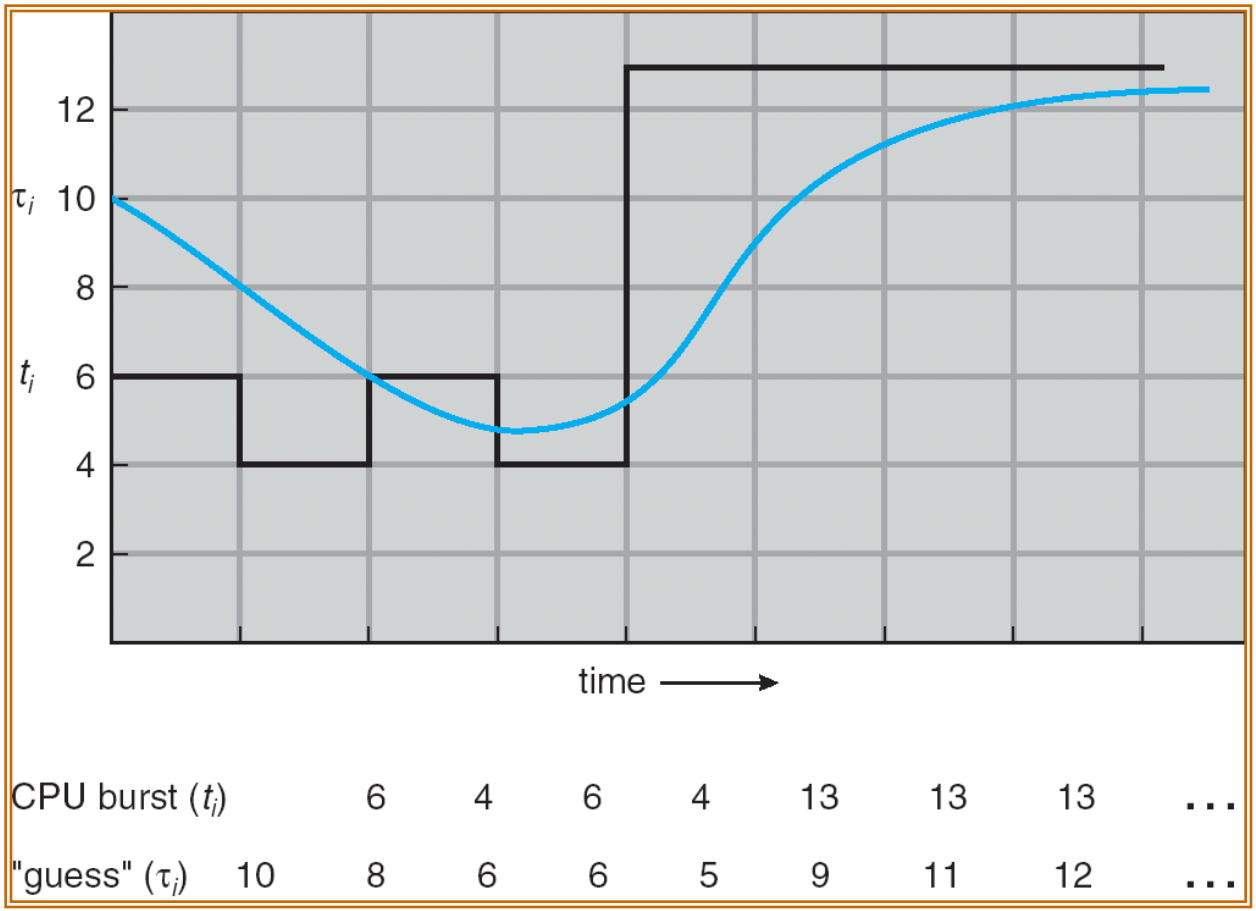
- Average waiting time = (9 + 1 + 0 +2)/4 - 3

Determining Length of Next CPU Burst

- Can only estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging



Prediction of the Length of the Next CPU Burst



Examples of Exponential Averaging

- =0
  - $_{n+1} = _{n}$
  - Recent history does not count
- =1
  - $_{n+1} = t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$_{n+1} = t_n + (1 -) t_n - 1 + \ldots$$
$$+ (1 -)^j t_n - 1 + \ldots$$
$$+ (1 -)^{n=1} t_n \; _0$$

- Since both and (1 -) are less than or equal to 1, each successive term has less weight than its predecessor

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer highest priority)
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem Starvation – low priority processes may never execute
- Solution Aging – as time progresses increase the priority of the process

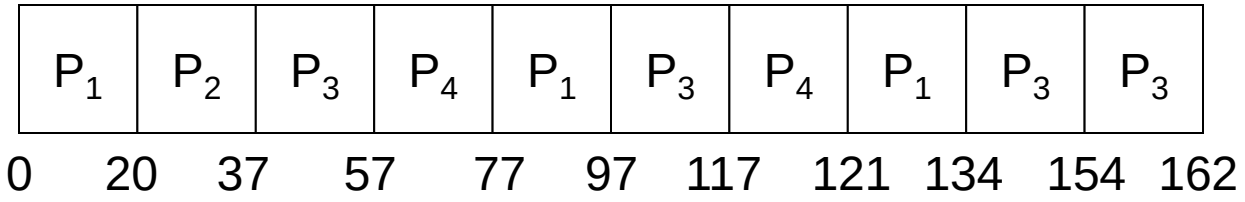
Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- Performance
  - *q* large FIFO
  - *q* small *q* must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 20

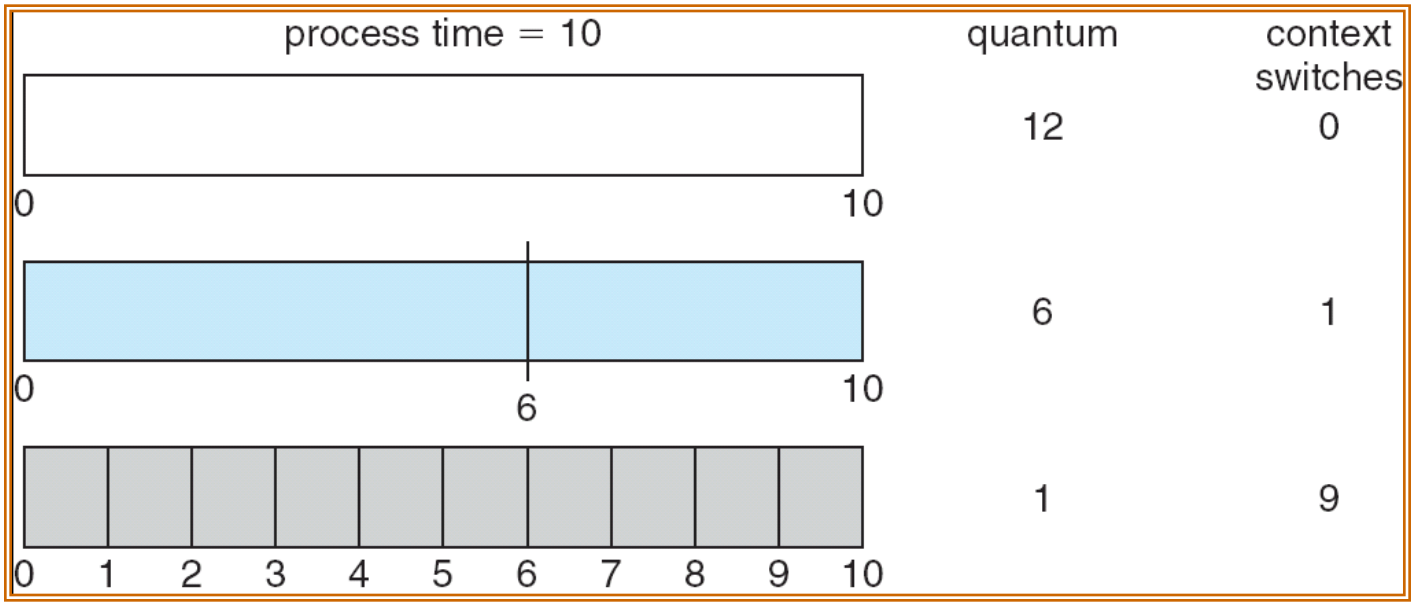
Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:

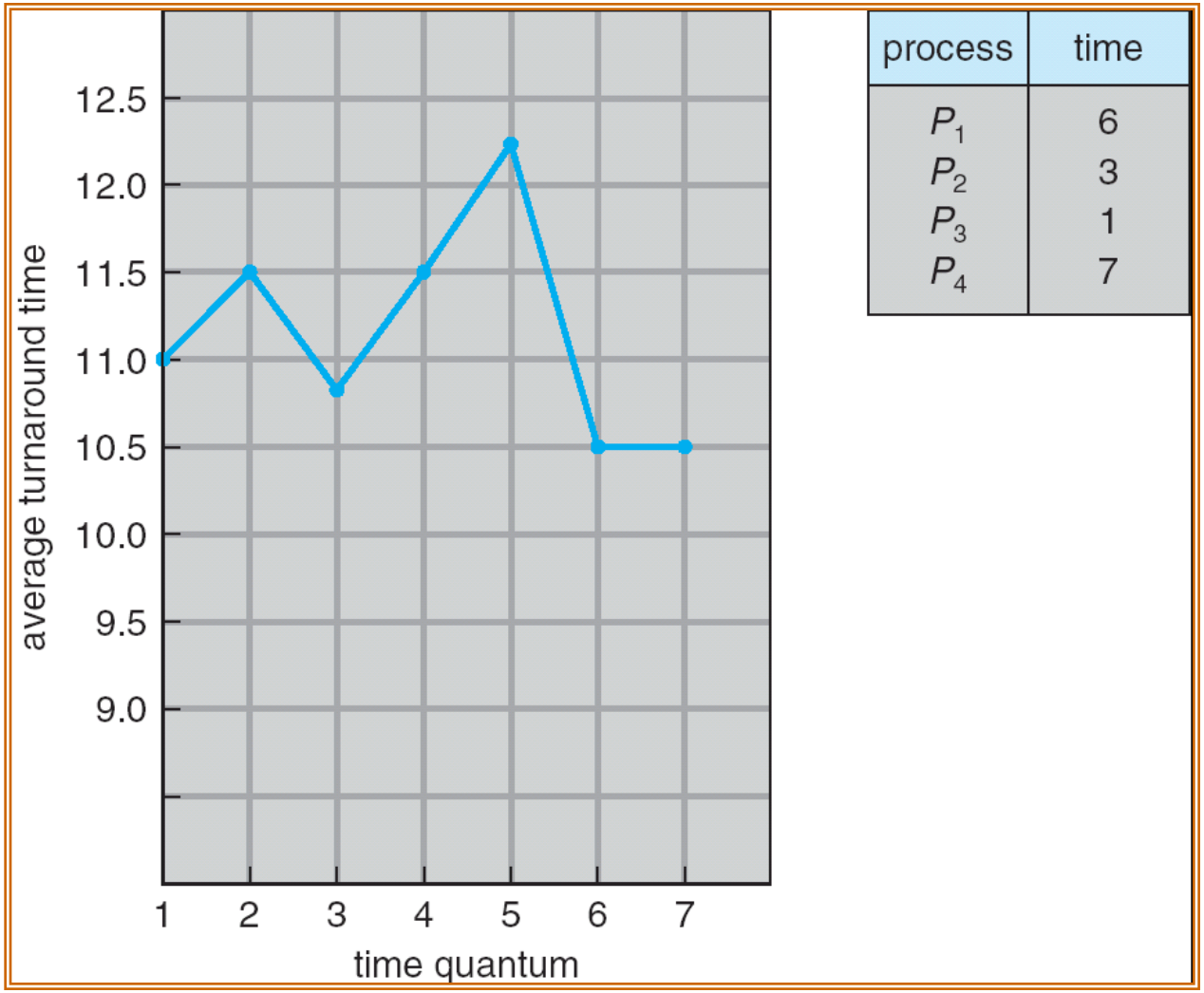


- Typically, higher average turnaround than SJF, but better *response*

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum

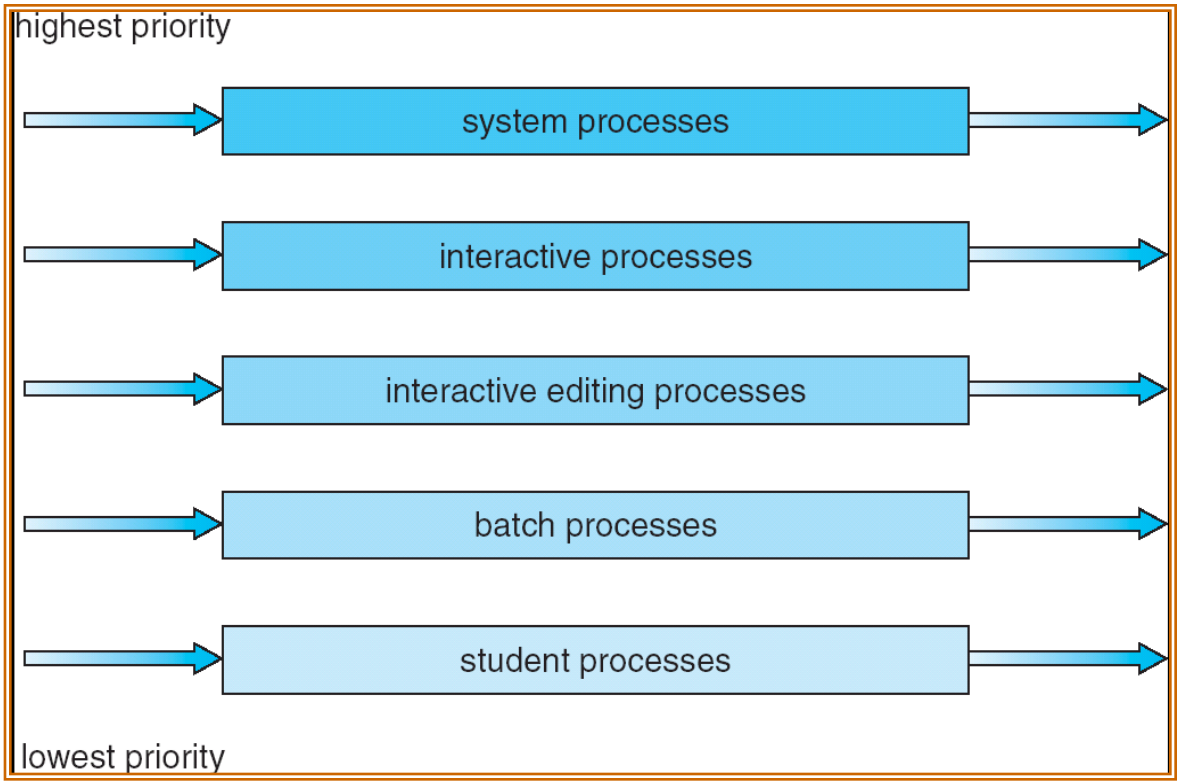


process	time
P_1	6
P_2	3
P_3	1
P_4	7

Multilevel Queue

- Ready queue is partitioned into separate queues:
  foreground (interactive)
  background (batch)
- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS

Multilevel Queue Scheduling



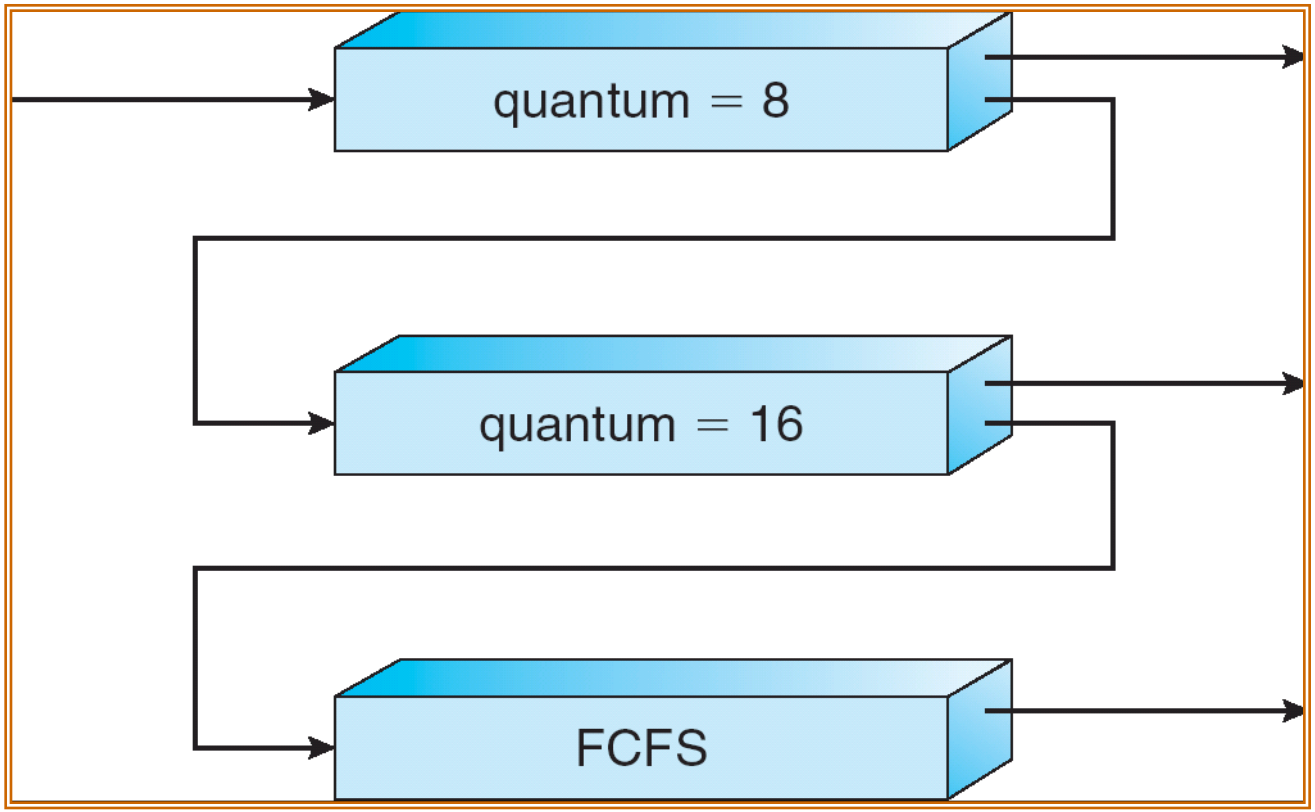
Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – time quantum 8 milliseconds
  - Q_1 – time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1.
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

Multilevel Feedback Queues



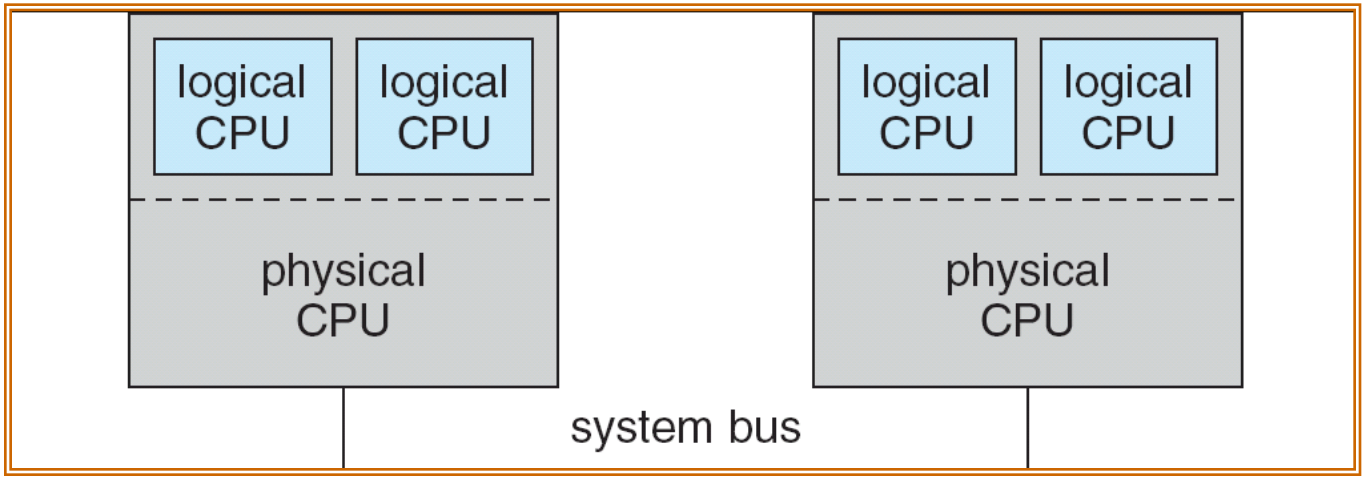
Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- *Homogeneous processors* within a multiprocessor
- *Load sharing*
- *Asymmetric multiprocessing* – only one processor accesses the system data structures, alleviating the need for data sharing

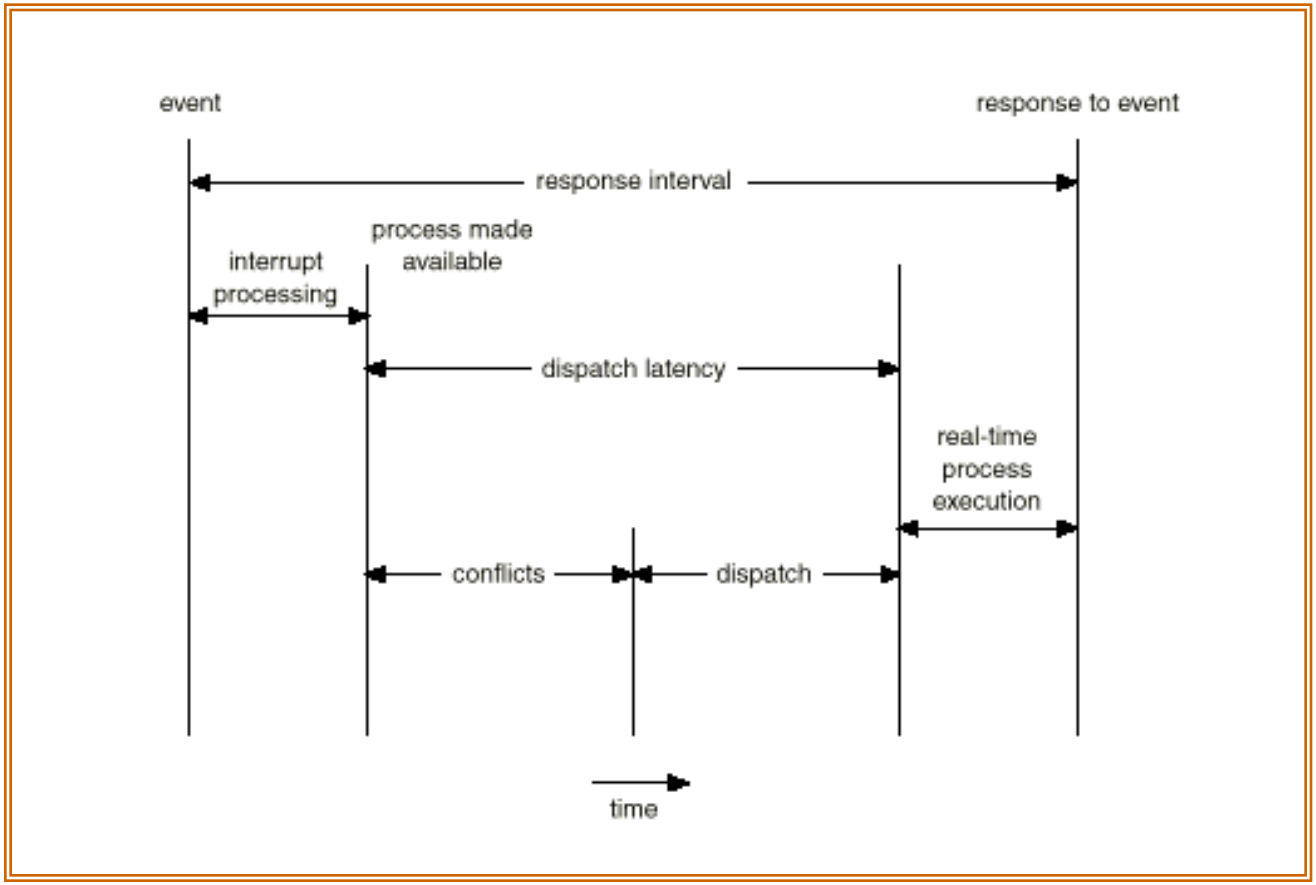
Real-Time Scheduling

- *Hard real-time* systems – required to complete a critical task within a guaranteed amount of time
- *Soft real-time* computing – requires that critical processes receive priority over less fortunate ones

A Typical SMT Architecture



Dispatch Latency



Thread Scheduling

- Local Scheduling – How the threads library decides which thread to put onto an available LWP
- Global Scheduling – How the kernel decides which kernel thread to run next

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5
int main(int argc, char *argv[])
{
      int i;
     pthread t tid[NUM THREADS];
     pthread attr t attr;
     /* get the default attributes */
     pthread attr init(&attr);
     /* set the scheduling algorithm to PROCESS or SYSTEM */
     pthread attr setscope(&attr, PTHREAD SCOPE SYSTEM);
     /* set the scheduling policy - FIFO, RT, or OTHER */
     pthread attr setschedpolicy(&attr, SCHED OTHER);
     /* create the threads */
     for (i = 0; i < NUM THREADS; i++)
     pthread create(&tid[i],&attr,runner,NULL);
```

Pthread Scheduling API

```
    /* now join on each thread */
    for (i = 0; i < NUM THREADS; i++)
    pthread join(tid[i], NULL);
}
 /* Each thread will begin control in this function */
void *runner(void *param)
{
    printf("I am a thread\n");
    pthread exit(0);
}
```

Thread Priorities

Priority	Comment
Thread.MIN_PRIORITY	Minimum Thread Priority
Thread.MAX_PRIORITY	Maximum Thread Priority
Thread.NORM_PRIORITY	Default Thread Priority

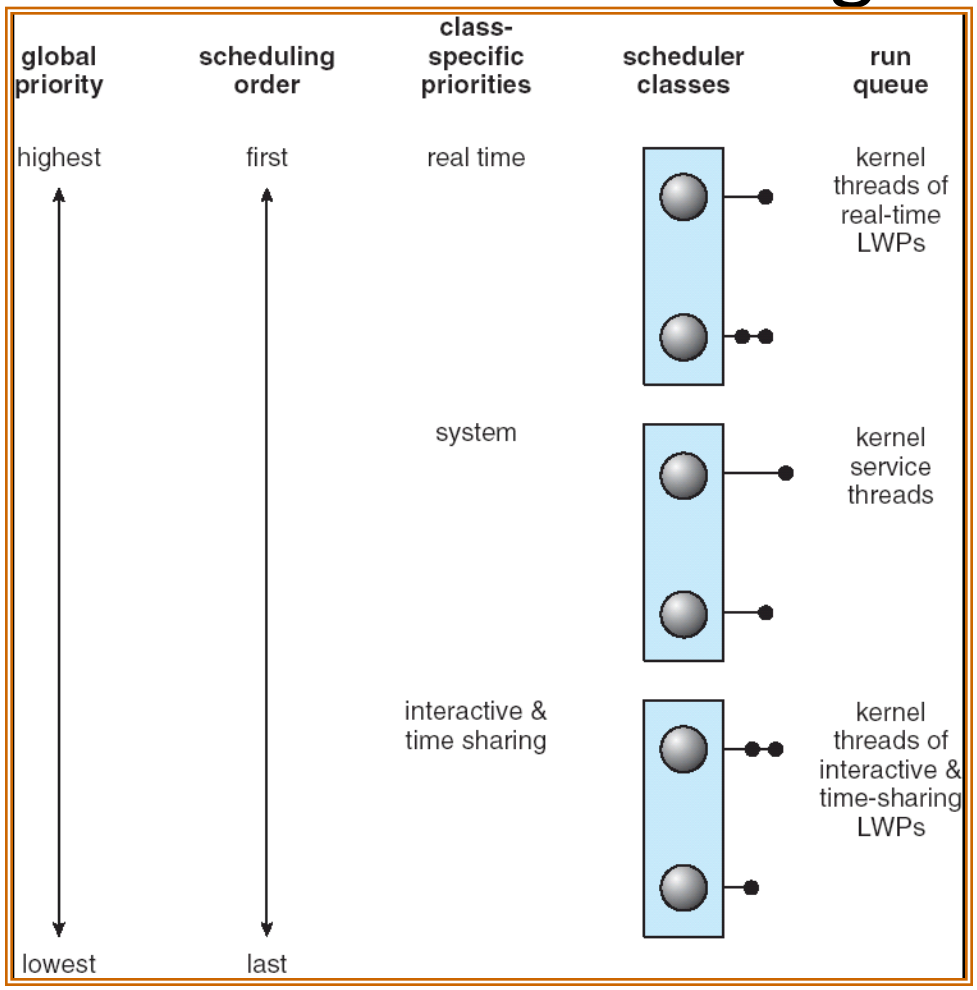
Priorities May Be Set Using setPriority() method:

```
setPriority(Thread.NORM_PRIORITY + 2);
```

Solaris Scheduling

- Priority Based Thread Scheduling
- Priority Classes
  - Real Time
  - System
  - Time Sharing – Default
    - Uses multiple level feedback queuing
  - Interactive
- Within a class
  - Different Priorities
  - Different Scheduling Algorithm
- Scheduler converts class specific priorities to global priorities and selects a thread with highest global priority

Solaris Scheduling



Solaris Dispatch Table

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

- Priority – Class dependent for time sharing and interactive – higher number means higher priority
- Time Quantum
- Time Quantum Expired- New priority of thread
- Return From Sleep - Priority

Solaris 9

- Introduced two additional classes
  - Fixed priority –
    - Have same priorities as in Time sharing class but do not change
  - Fair Share
    - Use CPU shares rather than priorities
      - CPU Shares allocated to a group of processes - PROJECT

Windows XP Scheduling

- Threads are scheduled using a priority-based preemptive scheduling
- A thread runs until:
  - It is preempted
  - Terminates
  - Time quantum ends
  - It calls a blocking system call
- Dispatcher uses 32 levels of Priority
  - Two clases :
    - Variable Class : Priorities 1-15
    - Real-time Class : Priorities 16 – 31
  - Thread running at Priority 0 is used for Memory Management
  - Lowest Priority thread is *Idle Thread*

Windows XP Priorities

Priority Class

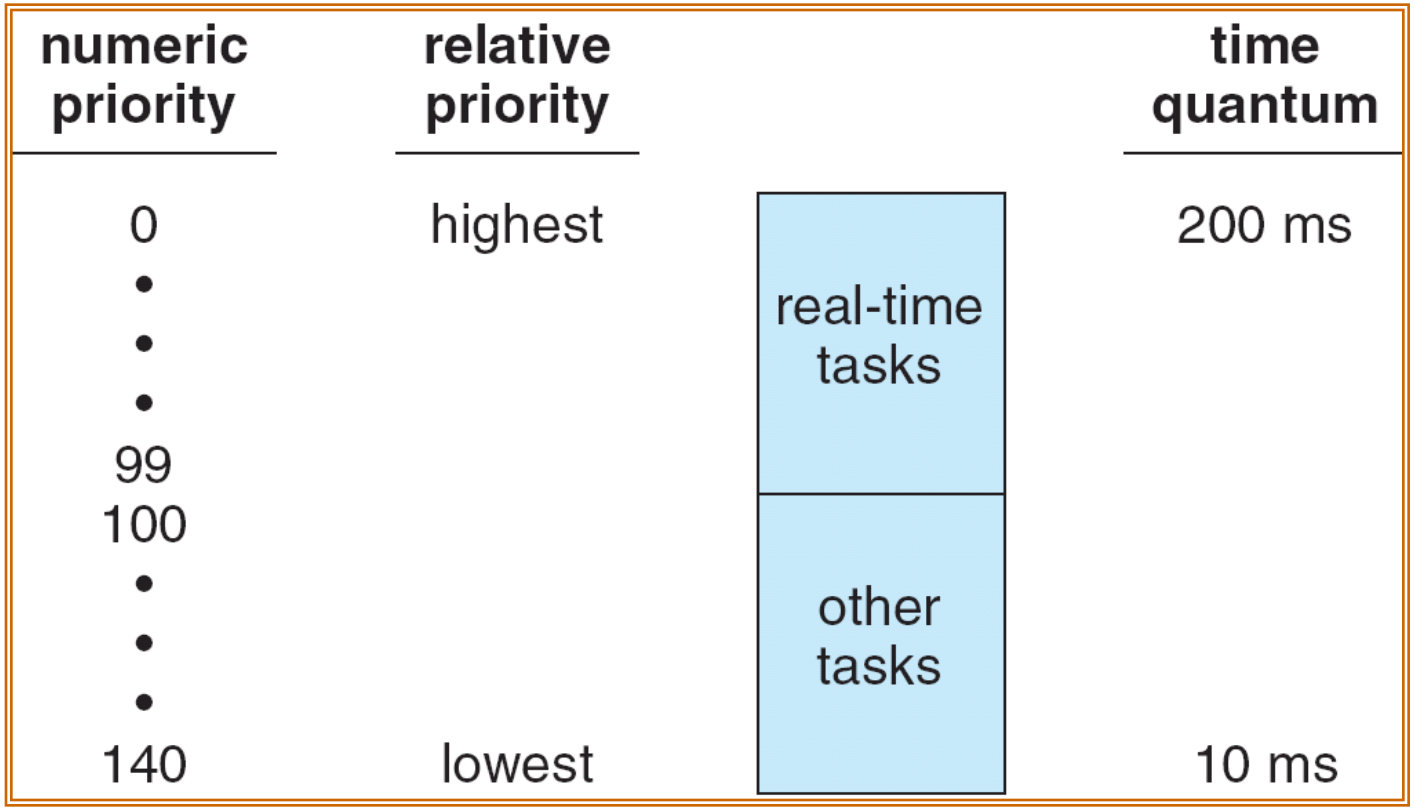
Relative Priority

	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

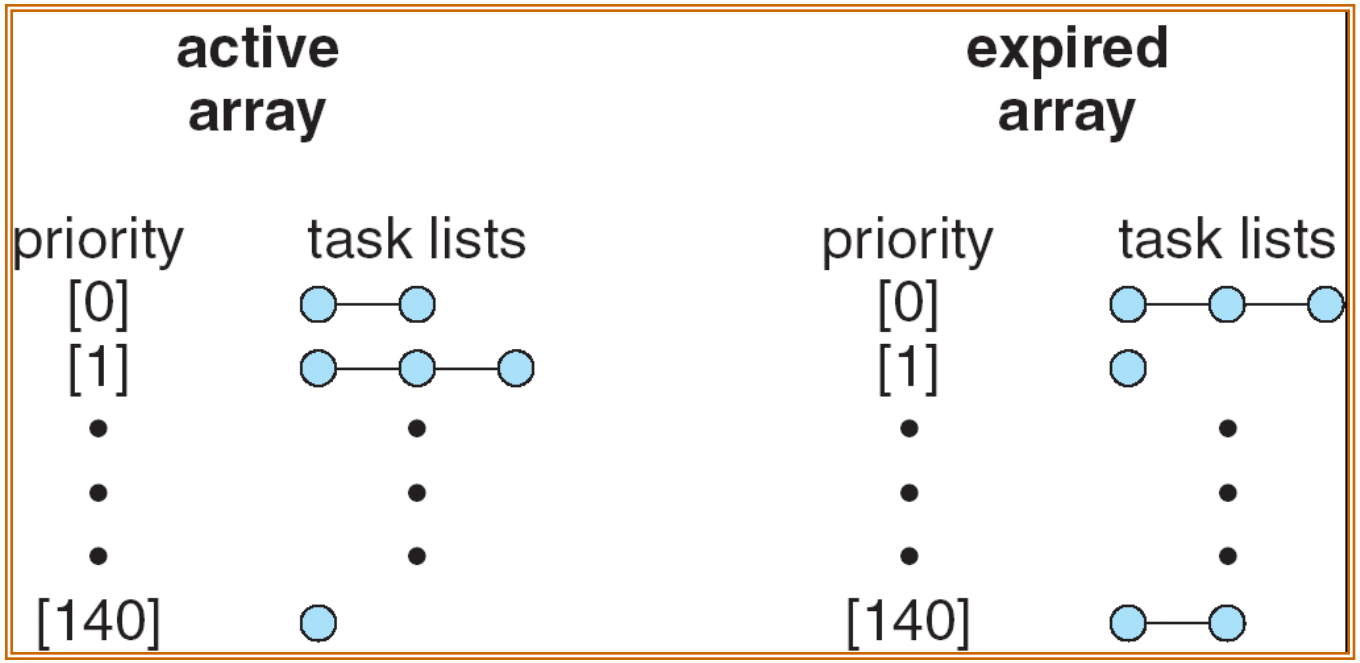
Linux Scheduling

- Two algorithms: time-sharing and real-time
- Time-sharing
  - Prioritized credit-based – process with most credits is scheduled next
  - Credit subtracted when timer interrupt occurs
  - When credit = 0, another process chosen
  - When all processes have credit = 0, recrediting occurs
    - Based on factors including priority and history
- Real-time
  - Soft real-time
  - Posix.1b compliant – two classes
    - FCFS and RR
    - Highest priority process always runs first

Priorities and Time Slice length



Tasks indexed according to Priority



Java Thread Scheduling

- JVM Uses a Preemptive, Priority-Based Scheduling Algorithm

- FIFO Queue is Used if There Are Multiple Threads With the Same Priority

Java Thread Scheduling (cont)

JVM Schedules a Thread to Run When:

1. The Currently Running Thread Exits the Runnable State
2. A Higher Priority Thread Enters the Runnable State

* Note – the JVM Does Not Specify Whether Threads are Time-Sliced or Not

Time-Slicing

Since the JVM Doesn't Ensure Time-Slicing, the yield() Method

May Be Used:

```
while (true) {
// perform CPU-intensive task
. . .
Thread.yield();
}
```

This Yields Control to Another Thread of Equal Priority

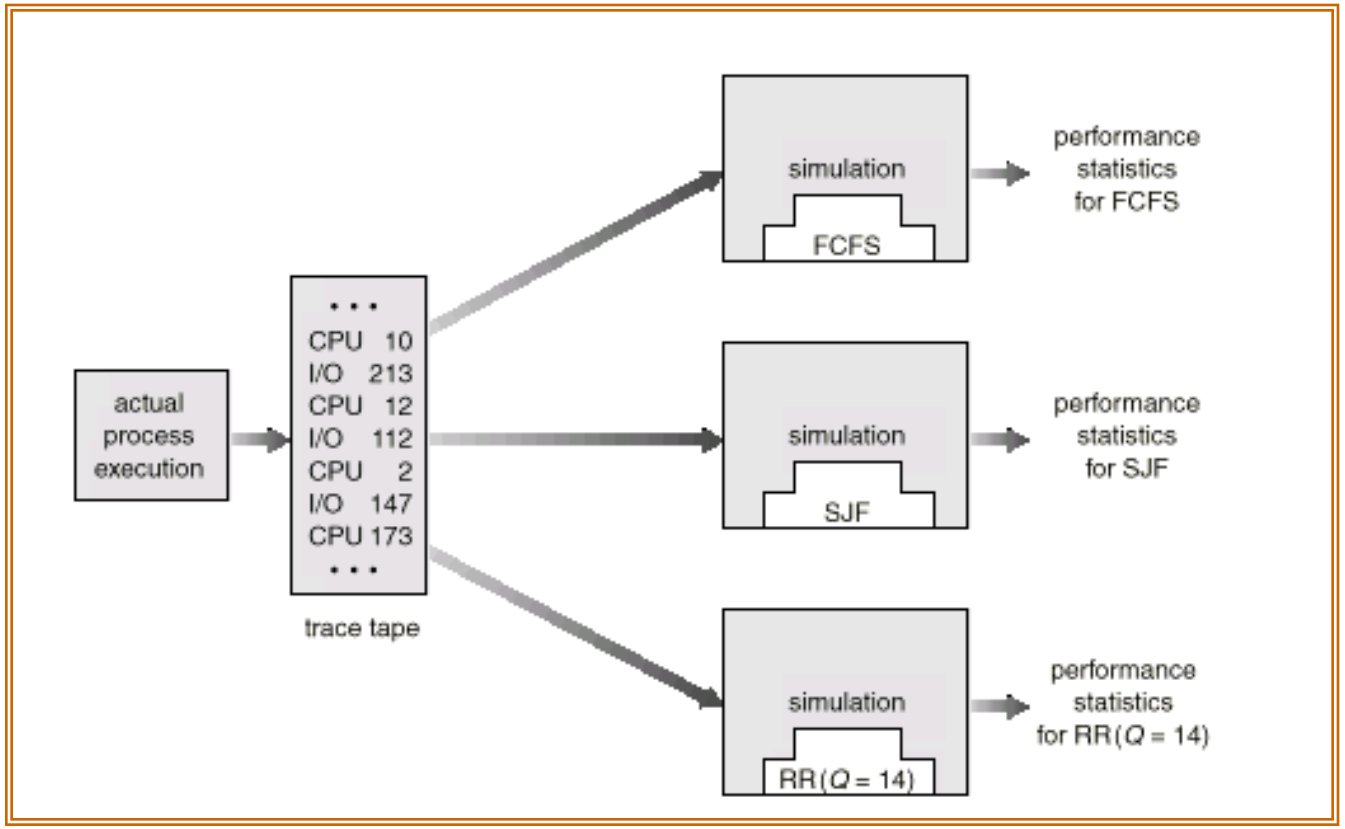
Algorithm Evaluation

- Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Queueing models
- Implementation

Deterministic Modeling

- Uses a predetermined workload
- Determines the performance of each algorithm for that worklaod

Evaluation of CPU Schedulers by Simulation



Scheduling and Dispatch

Windows Thread Scheduling

Roadmap

- Windows Scheduling Principles
- Windows API vs. NT Kernel Priorities
- Scheduling Data Structures
- Scheduling Scenarios
- Priority Boosts and Priority Adjustments

Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes/threads that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process/thread
- Waiting time – amount of time a process/thread has been waiting in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced, **not** output (i.e.; the hourglass)

How does the Windows scheduler relate to the issues discussed:

- Priority-driven, preemptive scheduling system
- Highest-priority runnable thread always runs
- Thread runs for time amount of *quantum*
- No single scheduler – event-based scheduling code spread across the kernel
- Dispatcher routines triggered by the following events:
  - Thread becomes ready for execution
  - Thread leaves running state (quantum expires, wait state)
  - Thread‘s priority changes (system call/NT activity)
  - Processor affinity of a running thread changes

Windows Scheduling Principles

- 32 priority levels
- Threads within same priority are scheduled following the Round-Robin policy
- Non-Realtime Priorities are adjusted dynamically
  - Priority elevation as response to certain I/O and dispatch events
  - Quantum stretching to optimize responsiveness
- Realtime priorities (i.e.; > 15) are assigned statically to threads

Scheduling

- Multiple threads may be ready to run
- "Who gets to use the CPU?"
- From Windows API point of view:
  - Processes are given a priority class upon creation
    - Idle, Normal, High, Realtime
    - Windows 2000 added "Above normal" and "Below normal"
  - Threads have a relative priority within the class
    - Idle, Lowest, Below_Normal, Normal, Above_Normal, Highest, and Time_Critical
- From the kernel's view:
  - Threads have priorities 0 through 31
  - Threads are scheduled, not processes
  - Process priority class is not used to make scheduling decisions

Windows Scheduling-related APIs:

Get/SetPriorityClass

Get/SetThreadPriority

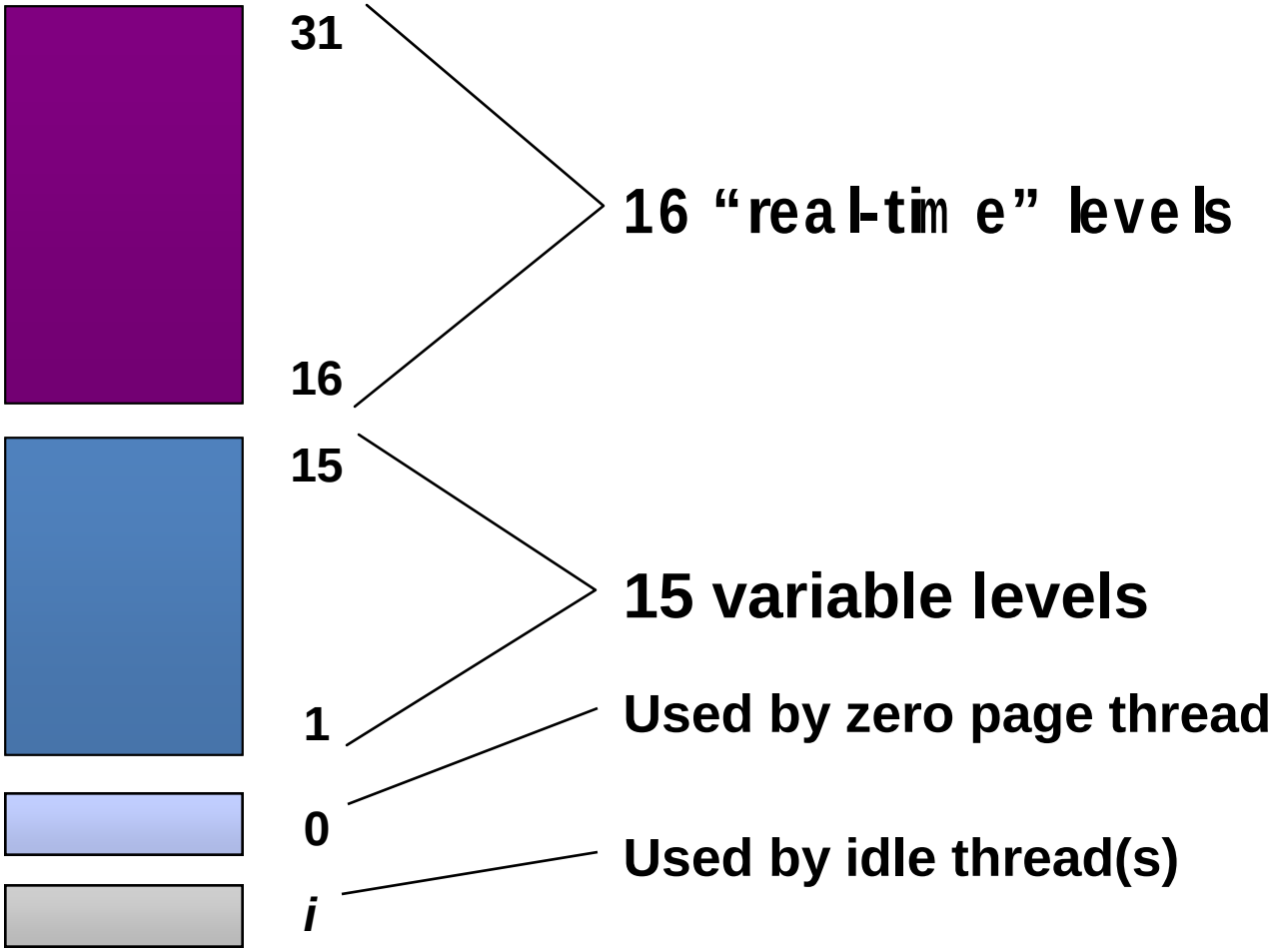
Get/SetProcessAffinityMask

SetThreadAffinityMask

SetThreadIdealProcessor

Suspend/ResumeThread

Kernel: Thread Priority Levels



Windows vs. NT Kernel Priorities

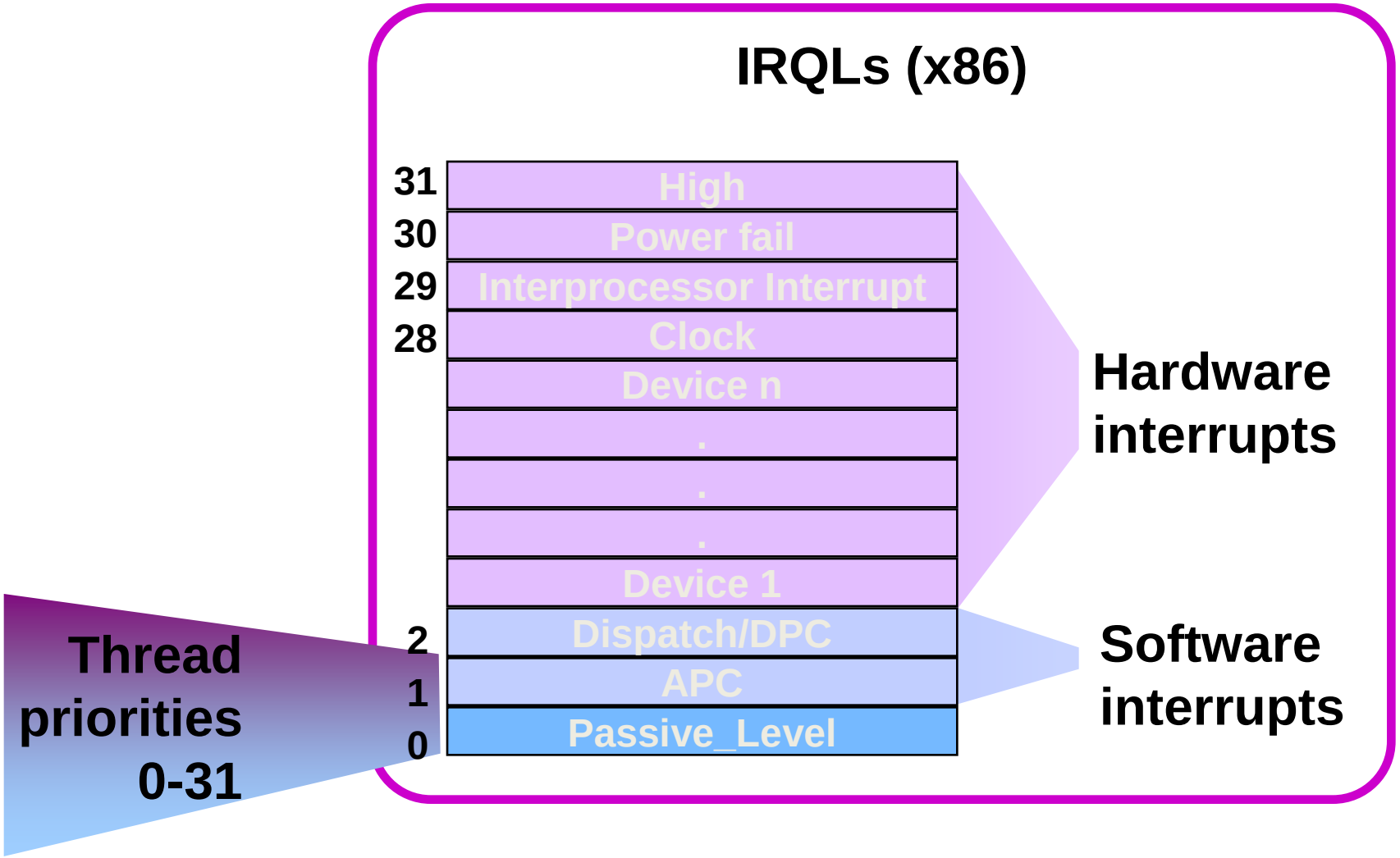
		Win32 Process Classes					
		Realt	Hig	Abo Norr	Norr	Belc Norr	Idle
Win	Time-c	31	15	15	15	15	15
Thre	High	26	15	12	10	8	6
Prior	Above-r	25	14	11	9	7	5
	Nori	24	13	10	8	6	4
	Below-r	23	12	9	7	5	3
	Low	22	11	8	6	4	2
	Idl	16	1	1	1	1	1

- Table shows base priorities ("current" or "dynamic" thread priority may be higher if base is < 15)
- Many utilities (such as Process Viewer) show the "dynamic priority" of threads rather than the base (Performance Monitor can show both)
- Drivers can set to any value with KeSetPriorityThread

Special Thread Priorities

- Idle threads -- one per CPU
  - When no threads want to run, Idle thread “runs”
    - Not a real priority level - appears to have priority zero, but actually runs “below” priority 0
    - Provides CPU idle time accounting (unused clock ticks are charged to the idle thread)
  - Loop:
    - Calls HAL to allow for power management
    - Processes DPC list
    - Dispatches to a thread if selected
  - Server 2003: in certain cases, scans per-CPU ready queues for next thread
- Zero page thread -- one per NT system
  - Zeroes pages of memory in anticipation of “demand zero” page faults
  - Runs at priority zero (lower than any reachable from Windows)
  - Part of the “System” process (not a complete process)

Thread Scheduling Priorities vs. Interrupt Request Levels (IRQLs)



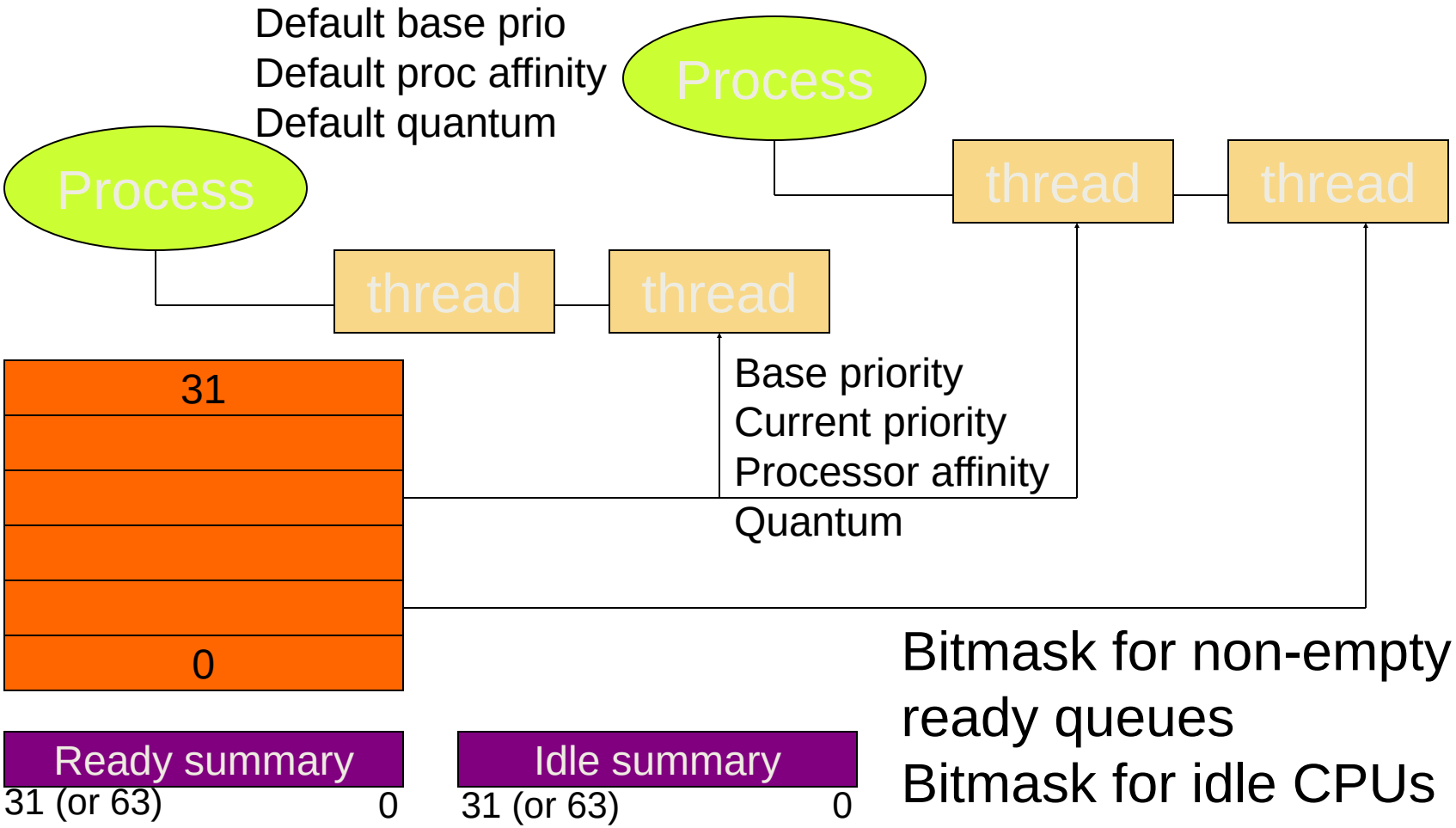
Single Processor Thread Scheduling

- Priority driven, preemptive
  - 32 queues (FIFO lists) of “ready” threads
  - UP: highest priority thread always runs
  - MP: One of the highest priority runnable thread will be running somewhere
  - No attempt to share processor(s) “fairly” among processes, only among threads
    - Time-sliced, round-robin within a priority level
- Event-driven; no guaranteed execution period before preemption
  - When a thread becomes Ready, it either runs immediately or is inserted at the tail of the Ready queue for its current (dynamic) priority

Thread Scheduling

- No central scheduler!
  - i.e. there is no always-instantiated routine called “the scheduler”
  - The “code that does scheduling” is not a thread
  - Scheduling routines are simply called whenever events occur that change the Ready state of a thread
  - Things that cause scheduling events include:
    - interval timer interrupts (for quantum end)
    - interval timer interrupts (for timed wait completion)
    - other hardware interrupts (for I/O wait completion)
    - one thread changes the state of a waitable object upon which other thread(s) are waiting
    - a thread waits on one or more dispatcher objects
    - a thread priority is changed
- Based on doubly-linked lists (queues) of Ready threads
  - Nothing that takes “order-*n* time” for *n* threads

Scheduling Data Structures

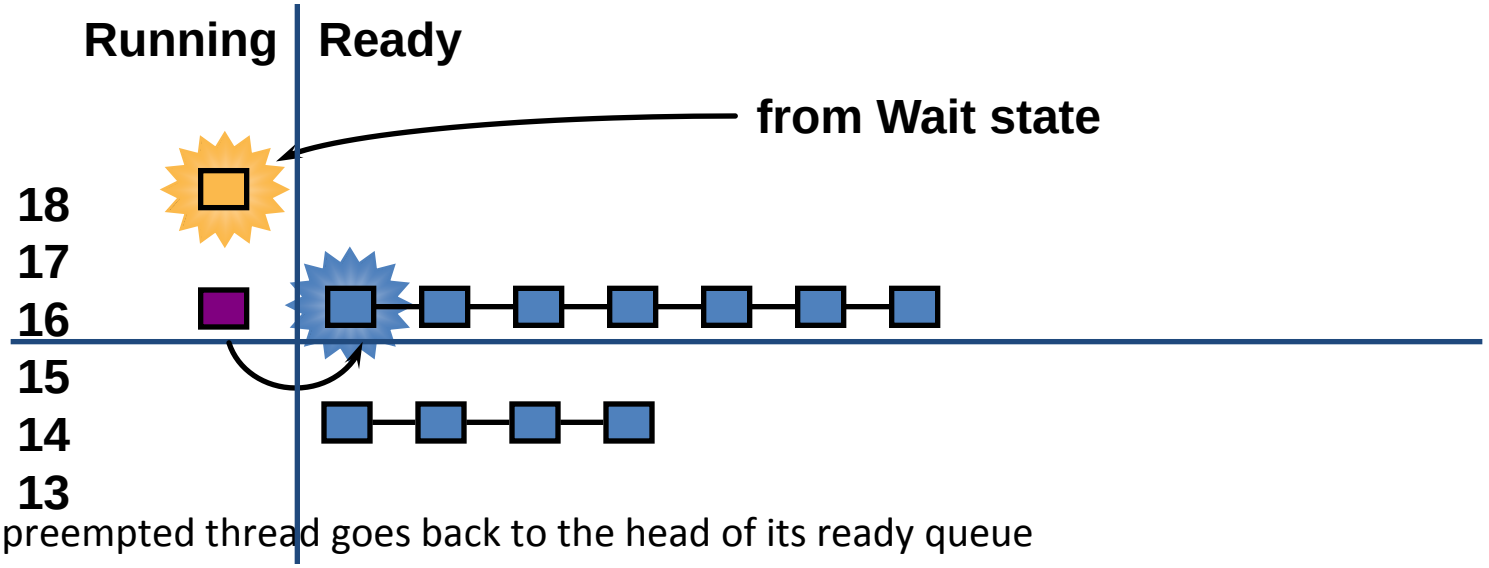


Scheduling Scenarios

- Preemption
  - A thread becomes Ready at a higher priority than the running thread
  - Lower-priority Running thread is preempted
  - Preempted thread goes back to head of its Ready queue
    - action: pick lowest priority thread to preempt
- Voluntary switch
  - Waiting on a dispatcher object
  - Termination
  - Explicit lowering of priority
    - action: scan for next Ready thread (starting at your priority & down)
- Running thread experiences quantum end
  - Priority is decremented unless already at thread base priority
  - Thread goes to tail of ready queue for its new priority
  - May continue running if no equal or higher-priority threads are Ready
    - action: pick next thread at same priority level

Scheduling Scenarios
Preemption

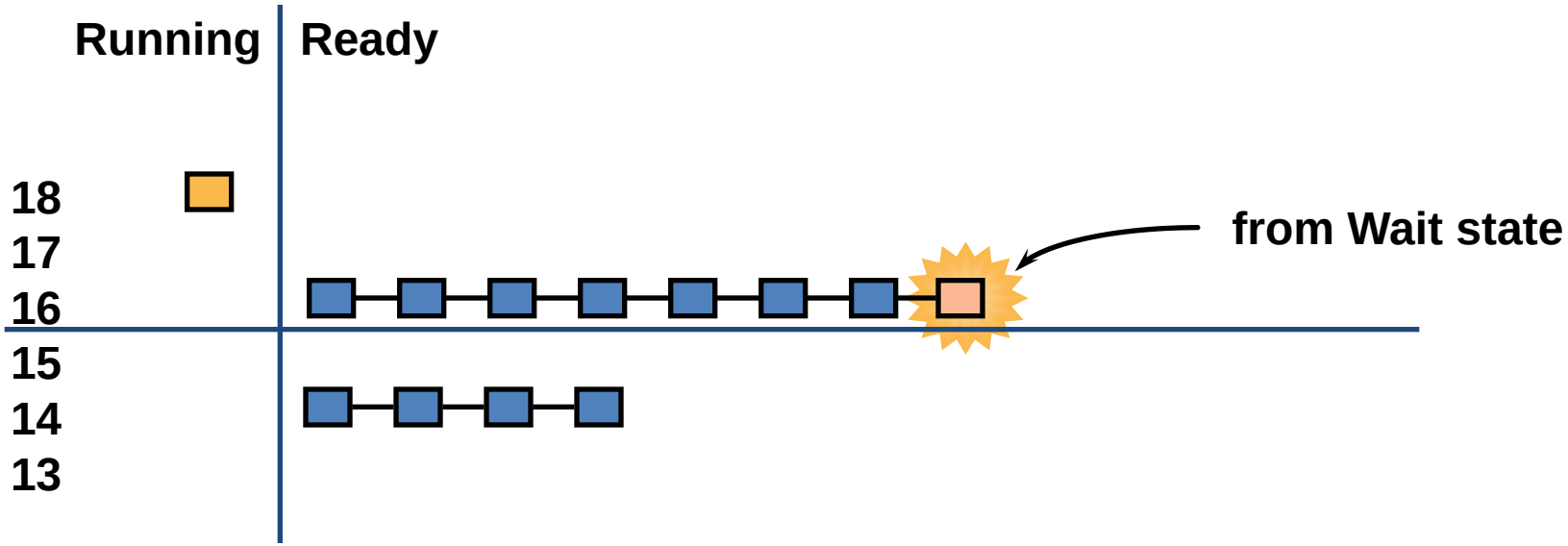
- Preemption is strictly event-driven
  - does not wait for the next clock tick
  - no guaranteed execution period before preemption
  - threads in kernel mode may be preempted (unless they raise IRQL to >= 2)



- A preempted thread goes back to the head of its ready queue

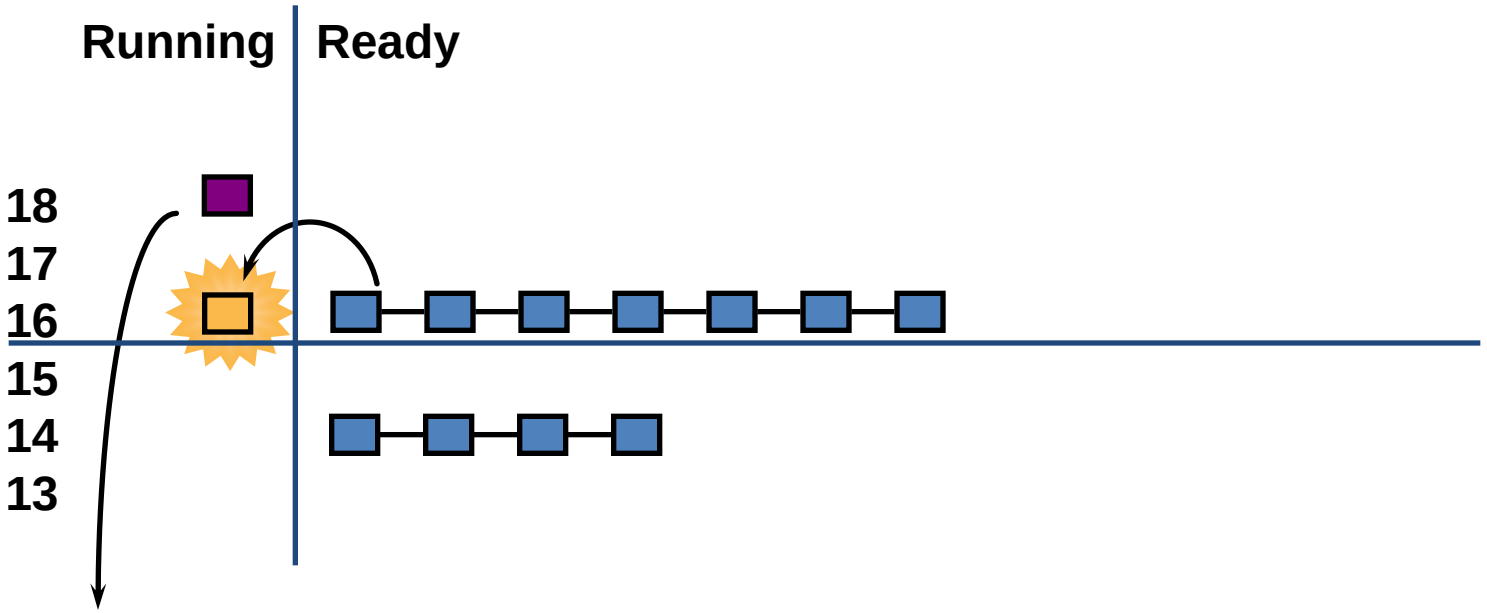
Scheduling Scenarios
Ready after Wait Resolution

- If newly-ready thread is not of higher priority than the running thread…
- …it is put at the tail of the ready queue for its current priority
  - If priority >=14 quantum is reset (t.b.d.)
  - If priority <14 and you're about to be boosted and didn't already have a boost, quantum is set to process quantum - 1



Scheduling Scenarios
Voluntary Switch

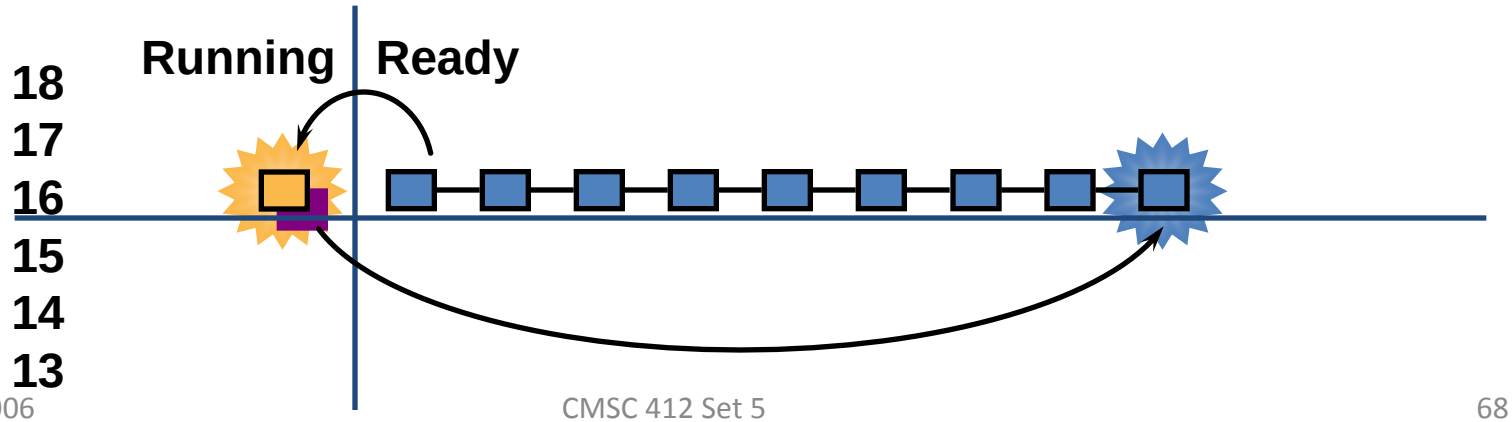
- When the running thread gives up the CPU…
- …Schedule the thread at the head of the next non-empty “ready” queue



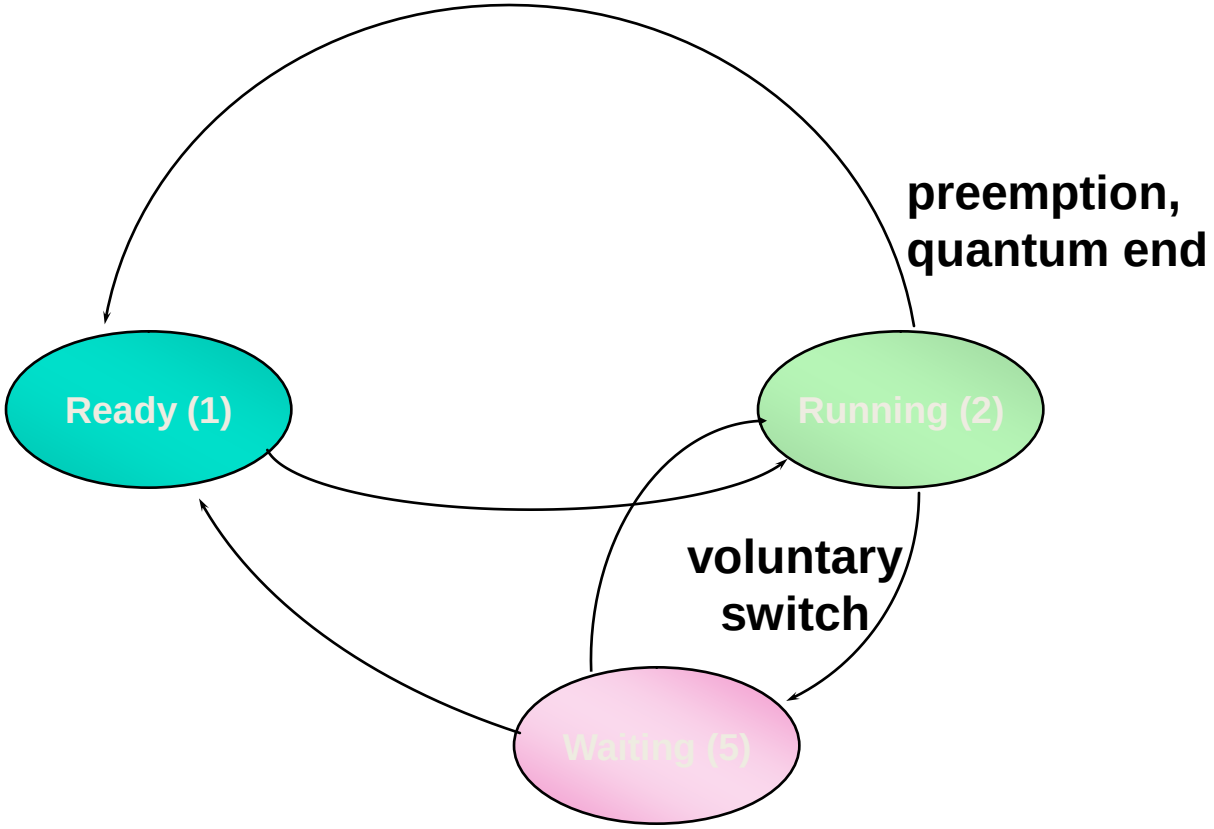
Scheduling Scenarios

Quantum End ("time-slicing")

- When the running thread exhausts its CPU quantum, it goes to the end of its ready queue
  - Applies to both real-time and dynamic priority threads, user and kernel mode
    - Quantums can be disabled for a thread by a kernel function
  - Default quantum on Professional is 2 clock ticks, 12 on Server
    - standard clock tick is 10 msec; might be 15 msec on some MP Pentium systems
  - if no other ready threads at that priority, same thread continues running (just gets new quantum)
  - if running at boosted priority, priority decays by one at quantum end (described later)



Basic Thread Scheduling States



Priority Adjustments

- Dynamic priority adjustments (boost and decay) are applied to threads in "dynamic" classes
  - Threads with base priorities 1-15 (technically, 1 through 14)
  - Disable if desired with SetThreadPriorityBoost or SetProcessPriorityBoost
- Five types:
  - I/O completion
  - Wait completion on events or semaphores
  - When threads in the foreground process complete a wait
  - When GUI threads wake up for windows input
  - For CPU starvation avoidance
- No automatic adjustments in "real-time" class (16 or above)
  - "Real time" here really means "system won't change the relative priorities of your real-time threads"
  - Hence, scheduling is predictable with respect to other "real-time" threads (but not for absolute latency)

Priority Boosting

To favor I/O intense threads:

- After an I/O: specified by device driver
  - IoCompleteRequest(Irp, PriorityBoost)

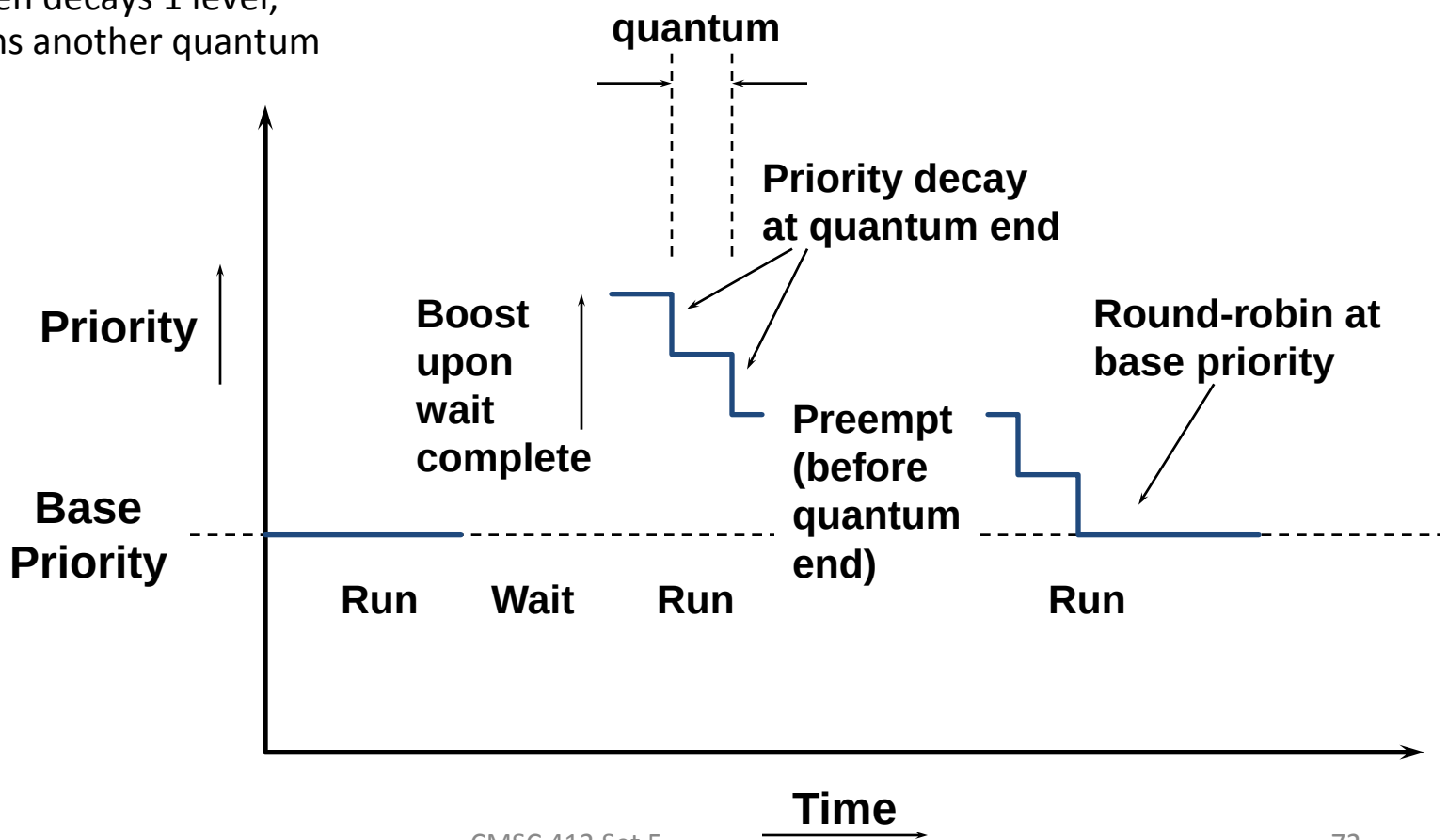
Common boost values (see NTDDK.H)
1: disk, CD-ROM, parallel, Video
2: serial, network, named
pipe, mailslot
6: keyboard or mouse
8: sound

Other cases discussed in the Windows Scheduling Internals Section

- After a wait on executive event or semaphore
- After any wait on a dispatcher object by a thread in the foreground process
- GUI threads that wake up to process windowing input (e.g. windows messages) get a boost of 2

Thread Priority Boost and Decay

- Behavior of these boosts:
  - Applied to thread's base priority
    - will not take you above priority 15
  - After a boost, you get one quantum
    - Then decays 1 level, runs another quantum



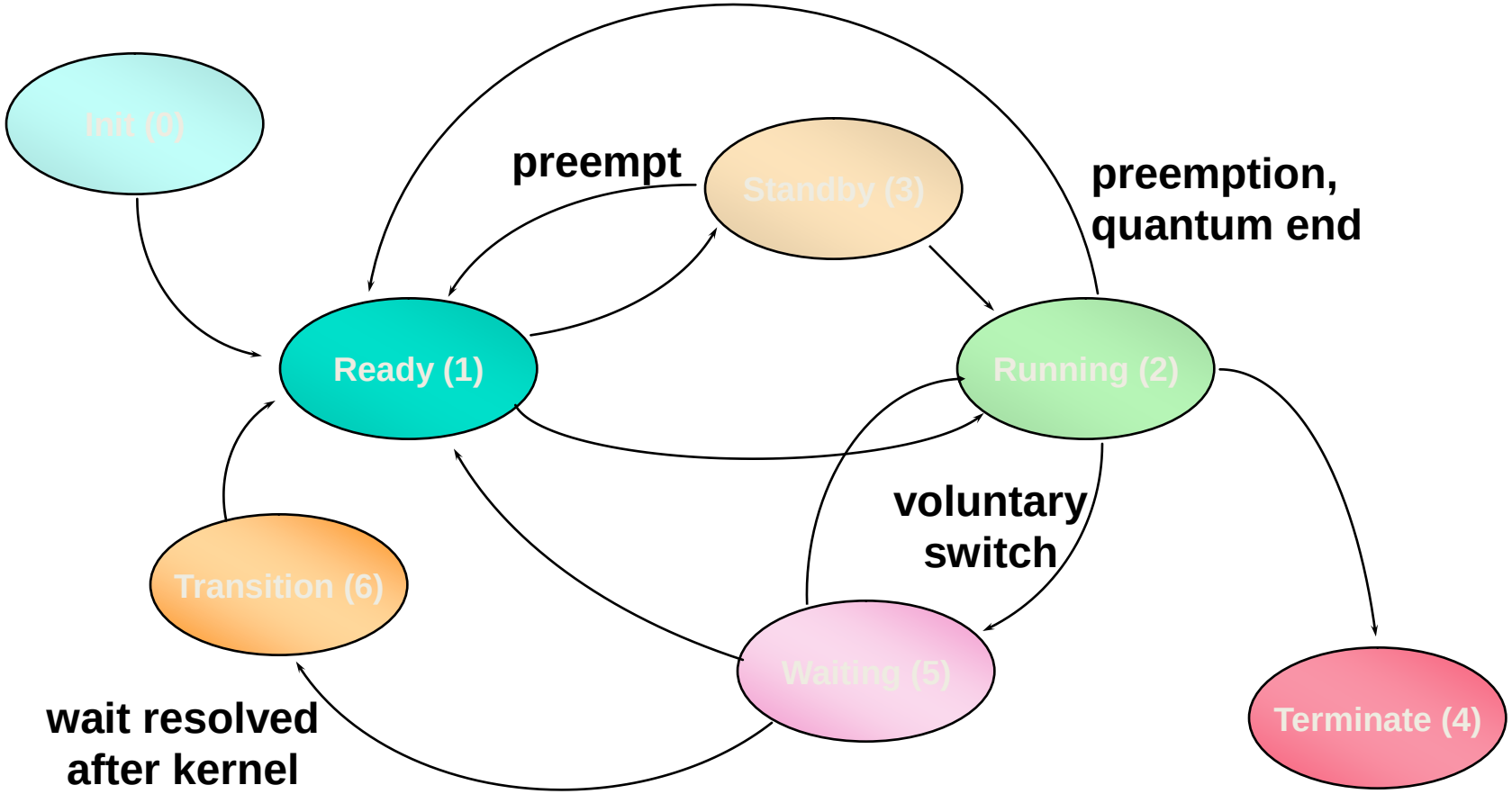
Scheduling and Dispatch

Advanced Windows Thread Scheduling

Roadmap

- Thread Scheduling Details
- Quantum Stretching
- CPU Starvation Avoidance
- Multiprocessor Scheduling
- Windows Server 2003 Enhancements

Thread Scheduling States (2000, XP)



Ready = thread eligible to be scheduled to run
Standby = thread is selected to run on CPU

Other Thread States

- Transition
  - Thread was in a wait entered from user mode for 12 seconds or more
  - System was short on physical memory
  - Balance set manager (t.b.d.) marked the thread's kernel stack as pageable (preparatory to "outswapping" the thread's process)
  - Later, the thread's wait was satisfied, but...
  - ...Thread can't become Ready until the system allocates a nonpageable kernel stack; it is in the "transition" state until then
- Initiate
  - Thread is "under construction" and can't run yet
- Standby
  - One processor has selected a thread for execution on another processor
- Terminate
  - Thread has executed its last code, but can't be deleted until all handles and references to it are closed (object manager)

Scheduling Scenarios
Quantum Details

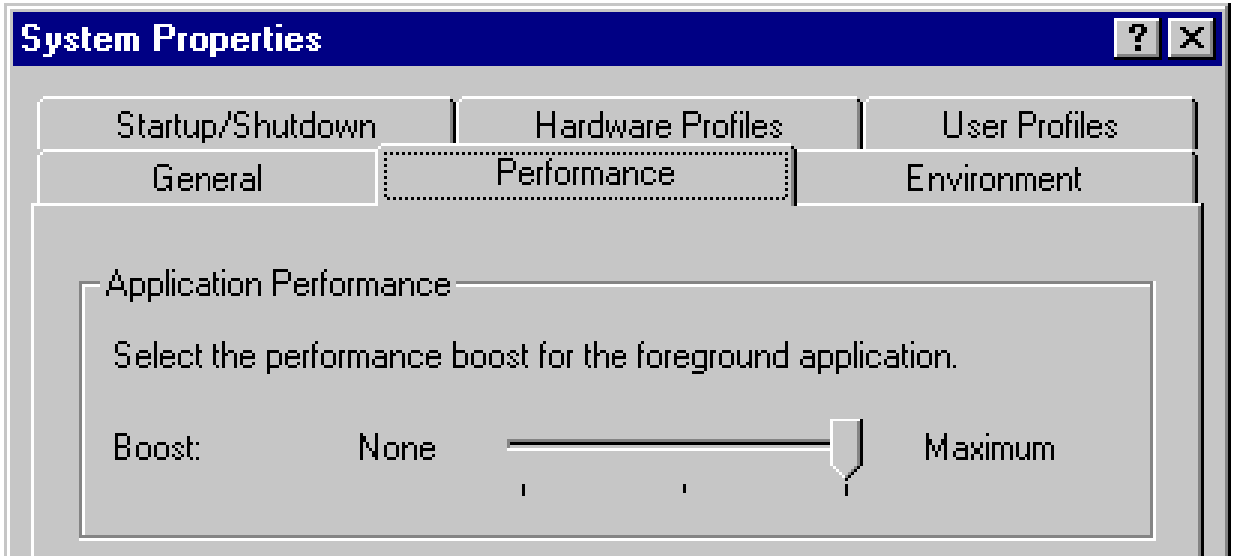
- Quantum internally stored as “3 * number of clock ticks”
  - Default quantum is 6 on Professional, 36 on Server
- Thread->Quantum field is decremented by 3 on every clock tick
- Process and thread objects have a Quantum field
  - Process quantum is simply used to initialize thread quantum for all threads in the process
- Quantum decremented by 1 when you come out of a wait
  - So that threads that get boosted after I/O completion won't keep running and never experiencing quantum end
  - Prevents I/O bound threads from getting unfair preference over CPU bound threads

Scheduling Scenarios
Quantum Details

- When Thread->Quantum reaches zero (or less than zero):
  - you've experienced quantum end
  - Thread->Quantum = Process->Quantum;// restore quantum
  - for dynamic-priority threads, this is the only thing that restores the quantum
  - for real-time threads, quantum is also restored upon preemption
- Interval timer interrupts when previous IRQL >= 2:
  - are not charged to the current thread's "privileged" time
  - but do cause the thread "remaining quantum" counter to be decremented

Quantum Stretching
(favoring foreground applications)

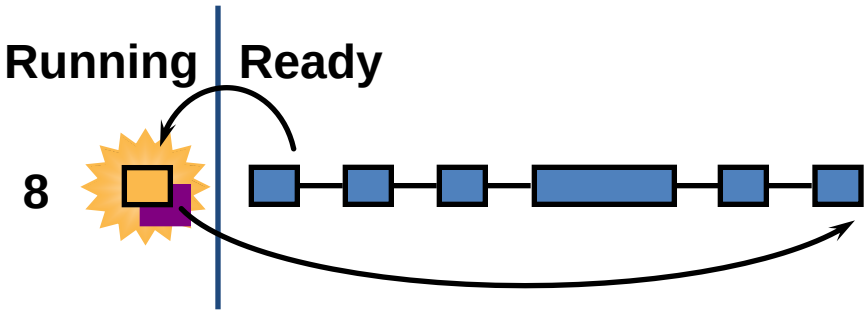
- If normal-priority process owns the foreground window, its threads may be given longer quantum
  - Set by Control Panel / System applet / Performance tab
  - Stored in …\System\CurrentControlSet\Control\PriorityControl Win32PrioritySeparation = 0, 1, or 2
  - New behavior with 4.0 (formerly implemented via priority shift)



Screen snapshot from:
Control Panel | System | Performance tab

Quantum Stretching

- Resulting quantum:
  - "Maximum" = 6 ticks
  - (middle) = 4 ticks
  - "None" = 2 ticks

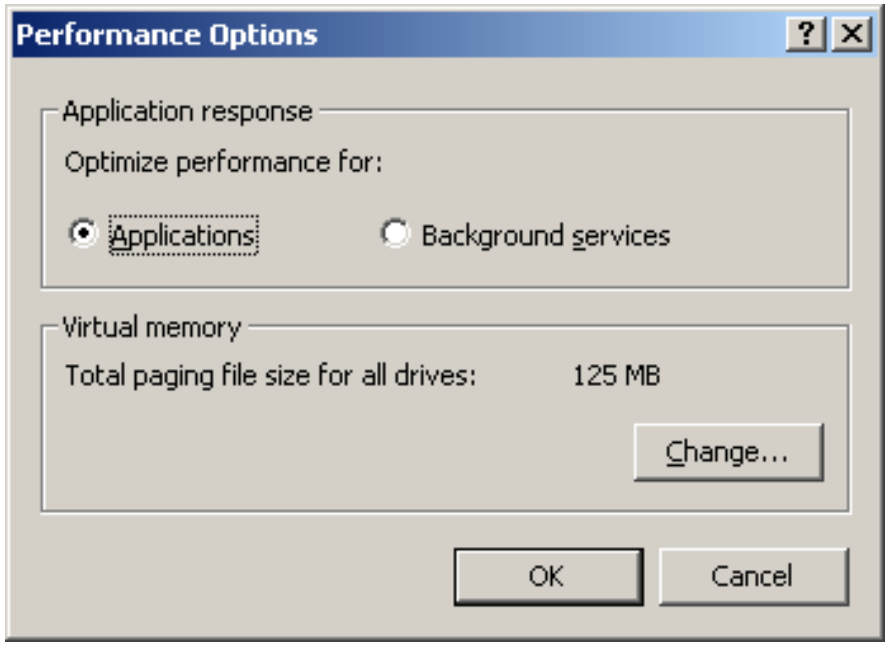


- Quantum stretching does not happen on Server
  - Quantum on Server is always 12 ticks

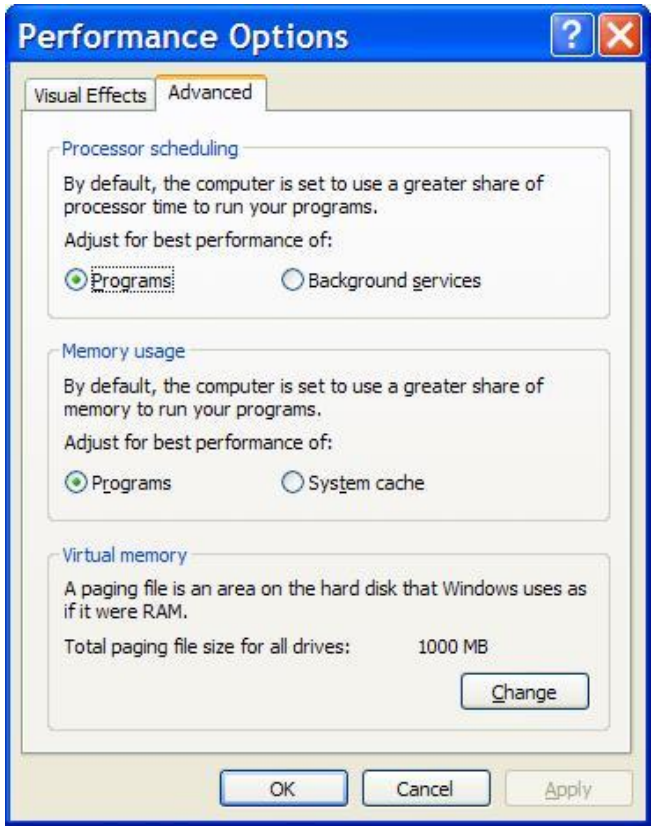
Quantum Selection

- As of Windows 2000, can choose short or long quantums (e.g. for Terminal Servers)
  - NT Server 4.0 was always the same, regardless of slider bar

Windows 2000:



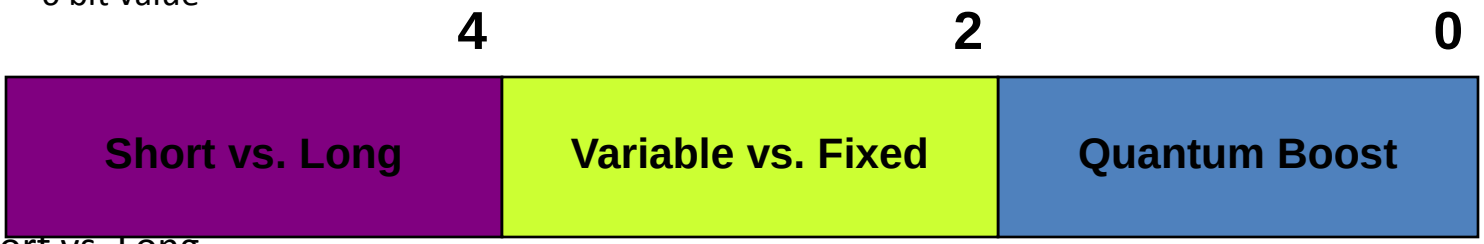
Windows XP:



Screen snapshot from:
Control Panel | System | Advanced tab | Performance

Quantum Control

- Finer grained quantum control can be achieved by modifying HKLM\System\CurrentControlSet\Control \PriorityControl\Win32PrioritySeparation
  - 6 bit value



- Short vs. Long
  0,3default (short for Pro, long for Server)
  1long
  2short
- Variable vs. Fixed
  0,3default (yes for Pro, no for Server)
  1yes
  2no
- Quantum Boost
  0fixed (overrides above setting)
  1double quantum of foreground threads
  2,3triple quantum of foreground threads

Controlling Quantum with Jobs

- If a process is a member of a job, quantum can be adjusted by setting the "Scheduling Class"
  - Only applies if process is higher then Idle priority class
  - Only applies if system running with fixed quantums (the default on Servers)
- Values are 0-9
  - 5 is default

Scheduling class	Quantum units
0	6
1	12
2	18
3	24
4	30
5	36
6	42
7	48
8	54
9	60

Priority Boosting

Common boost values (see NTDDK.H)
1: disk, CD-ROM, parallel, Video
2: serial, network, named pipe, mailslot
6: keyboard or mouse
8: sound

- After an I/O: specified by device driver
  - IoCompleteRequest(Irp, PriorityBoost)
- After a **wait on executive event** or semaphore
  - Boost value of 1 is used for these objects
  - Server 2003: for critical sections and pushlocks:
    - Waiting thread is boosted to 1 more than setting thread's priority (max boost is to 13)
    - Setting thread loses boost (lock convoy issue)
- After any **wait on a dispatcher object** by a thread in the foreground process:
  - Boost value of 2
    - XP/2003: boost is lost after one full quantum
  - Goal: improve responsiveness of interactive apps
- **GUI threads that wake up** to process windowing input (e.g. windows messages) get a boost of 2
  - This is added to the current, not base priority
  - Goal: improve responsiveness of interactive apps

CPU Starvation Avoidance

- Balance Set Manager system thread looks for "starved" threads
  - This is a thread, running at priority 16
  - Wakes up once per second and examines Ready queues
  - Looks for threads that have been Ready for 300 clock ticks (approximate 4 seconds on a 10ms clock)
  - Attempts to resolve "priority inversions" (high priority thread (12 in diagram) waits on something locked by a lower thread (4), which can't run because of a middle priority CPU-bound thread (7)),
    but not deterministically (no priority inheritance)

- Priority is boosted to 15 (14 prior to NT 4 SP3)
  - Quantum is doubled on Win2000/XP and set to 4 on 2003
  - At quantum end, returns to previous priority (no gradual decay) and normal quantum
- Scans up to 16 Ready threads per priority level each pass
- Boosts up to 10 Ready threads per pass
- Like all priority boosts, does not apply in the real-time range (priority 16 and above)

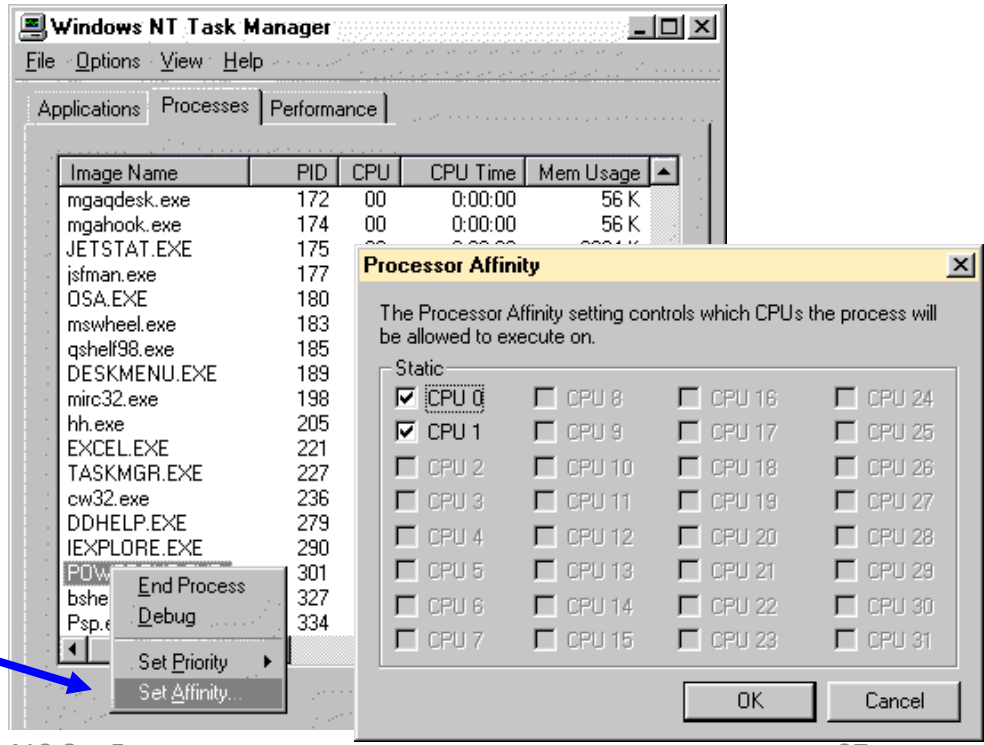
Multiprocessor Scheduling

- Threads can run on any CPU, unless specified otherwise
  - Tries to keep threads on same CPU (“soft affinity”)
  - Setting of which CPUs a thread will run on is called “hard affinity”
- Fully distributed (no “master processor”)
  - Any processor can interrupt another processor to schedule a thread
- Scheduling database:
  - Pre-Windows Server 2003: single system-wide list of ready queues
  - Windows Server 2003: per-CPU ready queues

Hard Affinity

- Affinity is a bit mask where each bit corresponds to a CPU number
  - Hard Affinity specifies where a thread is permitted to run
    - Defaults to all CPUs
  - Thread affinity mask must be subset of process affinity mask, which in turn must be a subset of the active processor mask

- Functions to change:
  - *SetThreadAffinityMask*, *SetProcessAffinityMask*, *SetInformationJobObject*
- Tools to change:
  - Task Manager or Process Explorer
    - Right click on process and choose "Set Affinity"
  - Psexec -a



Hard Affinity

- Can also set an image affinity mask
  - See "Imagecfg" tool in Windows 2000 Server Resource Kit Supplement 1
    - E.g. `Imagecfg –a 2 xyz.exe` will run xyz on CPU 1
- Can also set "uniprocessor only": sets affinity mask to one processor
  - `Imagecfg –u xyz.exe`
  - System chooses 1 CPU for the process
    - Rotates round robin at each process creation
  - Useful as temporary workaround for multithreaded synchronization bugs that appear on MP systems
- NOTE: Setting hard affinity can lead to threads' getting less CPU time than they normally would
  - More applicable to large MP systems running dedicated server apps
  - Also, OS may in some cases run your thread on CPUs other than your hard affinity setting (flushing DPCs, setting system time)
    - Thread "system affinity" vs "user affinity"

Soft Processor Affinity

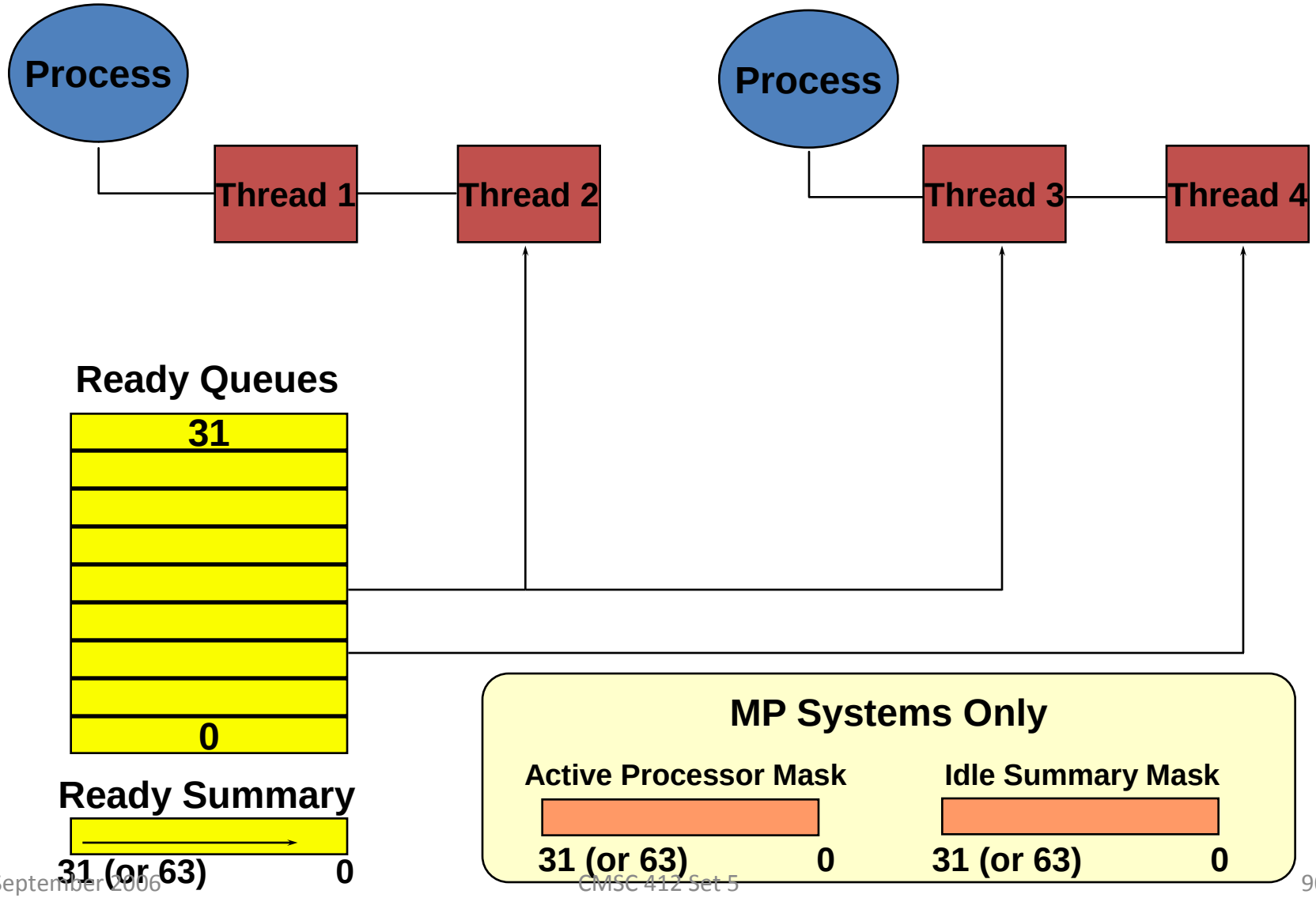
- Every thread has an “ideal processor”
  - System selects ideal processor for first thread in process (round robin across CPUs)
  - Next thread gets next CPU relative to the process seed
  - Can override with:

```
SetThreadIdealProcessor (
HANDLE hThread,// handle to the thread to be changed
DWORD dwIdealProcessor);// processor number
```

  - Hard affinity changes update ideal processor settings
  - Used in selecting where a thread runs next (see next slides)

  - For Hyperthreaded systems, new Windows API in Server 2003 to allow apps to optimize
    - GetLogicalProcessorInformation
  - For NUMA systems, new Windows APIs to allow applications to optimize:
    - Use GetProcessAffinityMask to get list of processors
      - Then GetNumaProcessorNode to get node # for each CPU
    - Or call GetNumaHighestNodeNumber and then GetNumaNodeProcessorMask to get the processor #s for each node

Windows 2000/XP Dispatcher Database



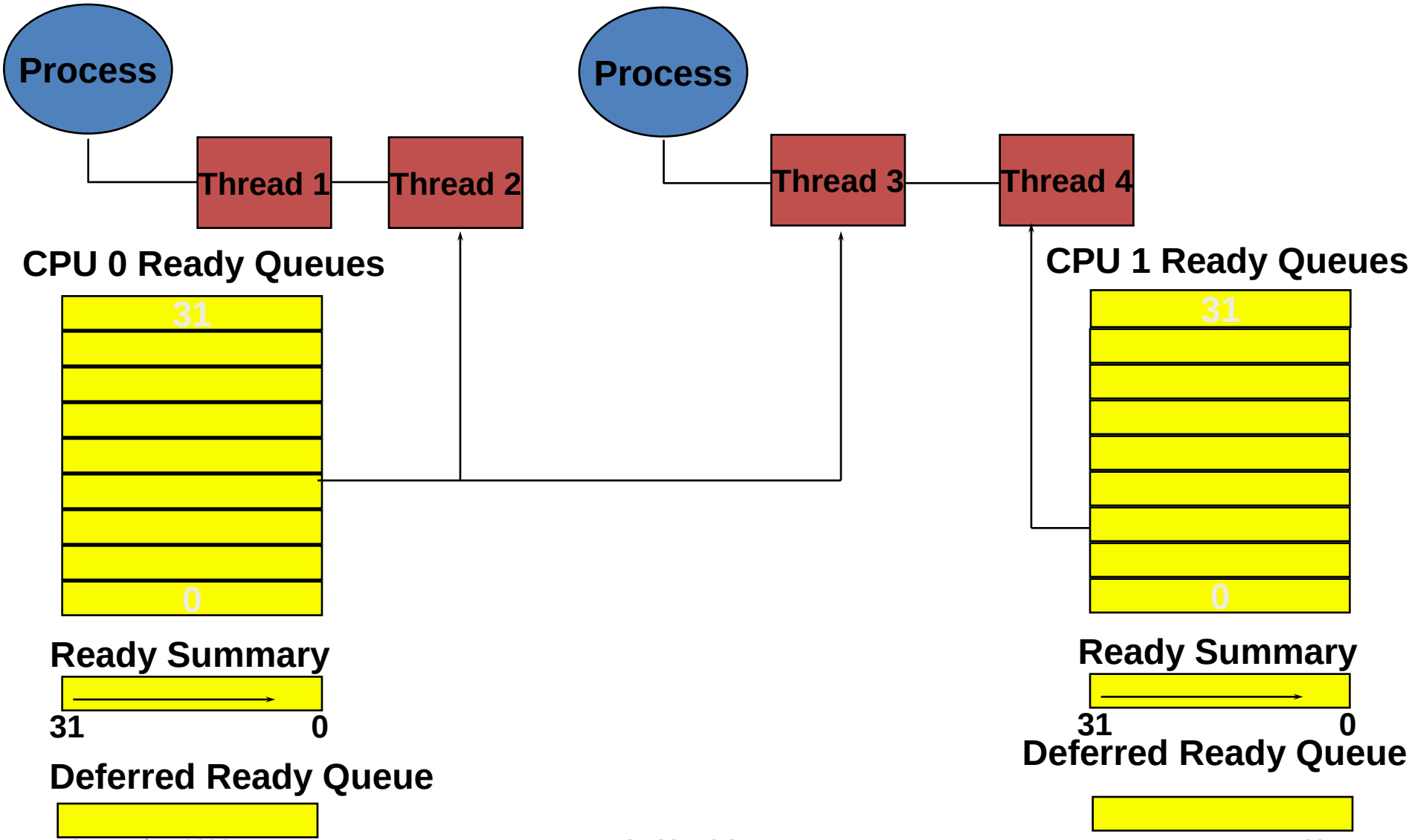
Choosing a CPU for a Ready Thread (Windows 2000 & XP)

- When a thread becomes ready to run (e.g. its wait completes, or it is just beginning execution), need to choose a processor for it to run on
- First, it sees if any processors are idle that are in the thread's hard affinity mask:
  - If its "ideal processor" is idle, it runs there
  - Else, if the previous processor it ran on is idle, it runs there
  - Else if the current processor is idle, it runs there
  - Else it picks the highest numbered idle processor in the thread's affinity mask
- If no processors are idle:
  - If the ideal processor is in the thread's affinity mask, it selects that
  - Else if the the last processor is in the thread's affinity mask, it selects that
  - Else it picks the highest numbered processor in the thread's affinity mask
- Finally, it compares the priority of the new thread with the priority of the thread running on the processor it selected (if any) to determine whether or not to perform a preemption

Selecting a Thread to Run on a CPU (Windows 2000 & XP)

- System needs to choose a thread to run on a specific CPU at:
  - At quantum end
  - When a thread enters a wait state
  - When a thread removes its current processor from its hard affinity mask
  - When a thread exits
- Starting with the first thread in the highest priority non-empty ready queue, it scans the queue for the first thread that has the current processor in its hard affinity mask and:
  - Ran last on the current processor, or
  - Has its ideal processor equal to the current processor, or
  - Has been in its Ready queue for 3 or more clock ticks, or
  - Has a priority >=24
- If it cannot find such a candidate, it selects the highest priority thread that can run on the current CPU (whose hard affinity includes the current CPU)
  - Note: this may mean going to a lower priority ready queue to find a candidate

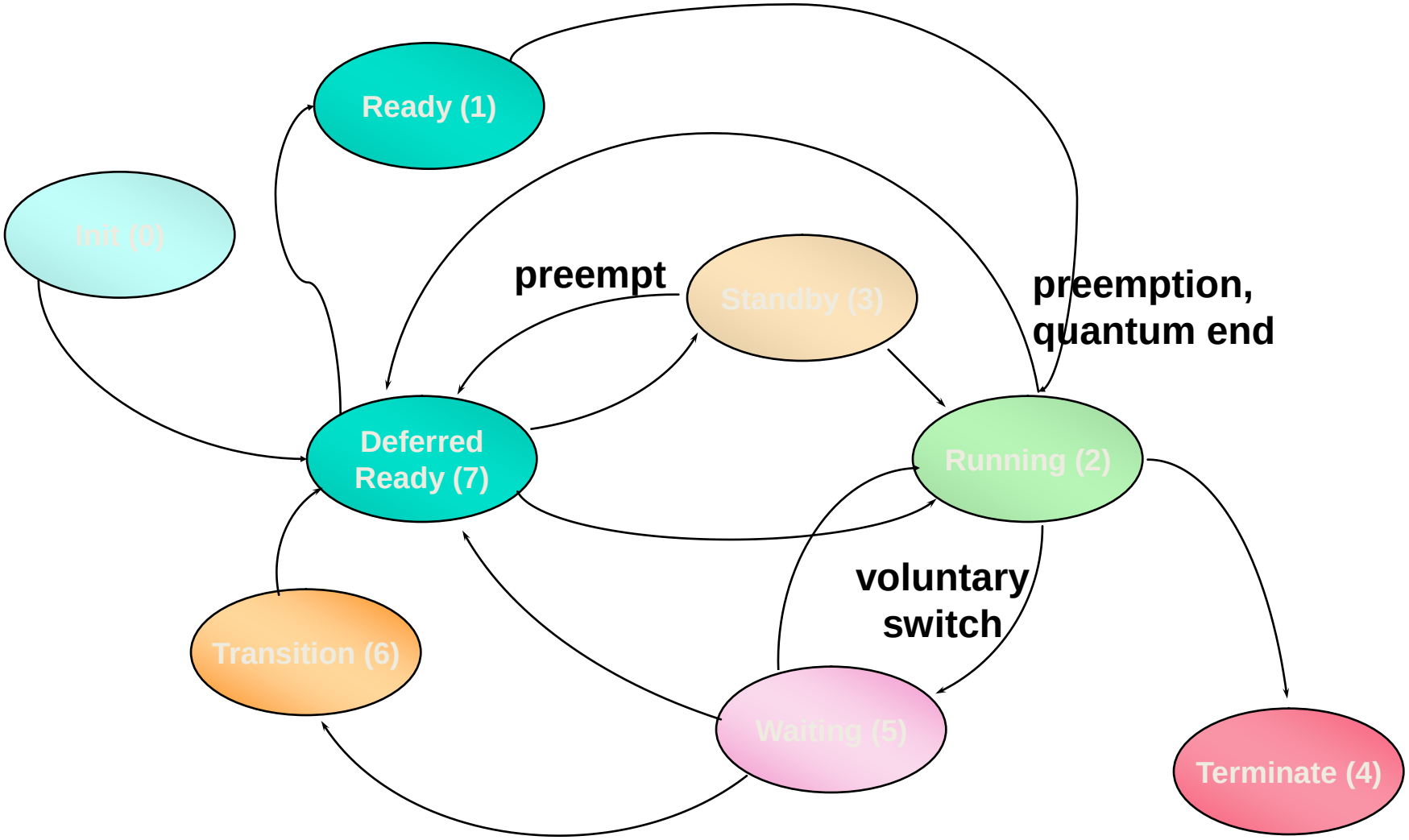
Windows Server 2003 Dispatcher Database



Server 2003 Enhancements

- Threads always go into the ready queue of their ideal processor
- Instead of locking the dispatcher database to look for a candidate to run, per-CPU ready queue is checked first (first grabs PRCB spinlock)
  - If a thread has been selected to run on the CPU, does the context swap
  - Else begins scan of other CPU's ready queues looking for a thread to run
    - This scan is done OUTSIDE the dispatcher lock
    - Just acquires CPU PRCB lock
- Dispatcher lock still acquired to wait or unwait a thread and/or change state of a dispatcher object
- Bottom line: dispatcher lock is now held for a MUCH shorter time

Thread Scheduling States (Server 2003)



Server 2003 Enhancements

- Idle processor selection further refined to:
  - If a NUMA system: if there are idle CPUs in the node containing the thread's ideal processor, reduce to that set
  - If a hyperthreaded system: if one of the idle processors is a physical processor with all logical processors idle, reduce to that set
  - Then try to eliminate idle CPUs that are sleeping
  - If thread ran last on a member of the set, pick that CPU
    - Else pick lowest numbered CPU in remaining set

“Affinity Collisions”

- Highest-priority *n* threads may not be Running if thread affinity interferes
- NT guarantees the highest-priority thread will be Running
  - But lower-priority *n*-1 Ready threads may not be…
  - …because scheduler will not “move” running threads among CPUs
- Example: Threads became Ready in order A, B, C