Unit 4: Scheduling and Dispatch

4.4. Windows Thread Scheduling

Roadmap for Section 4.4.

- 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

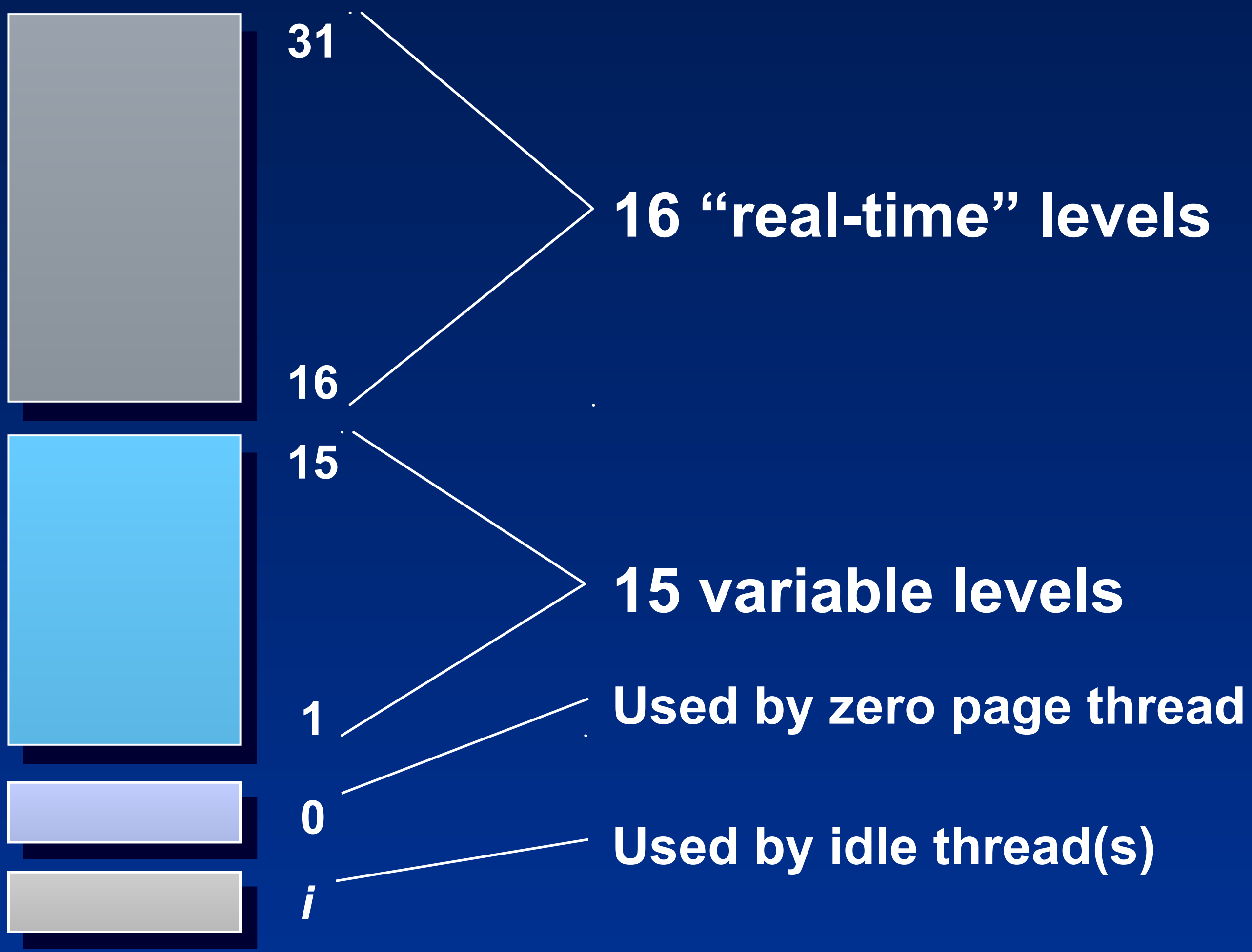
Scheduling

- 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

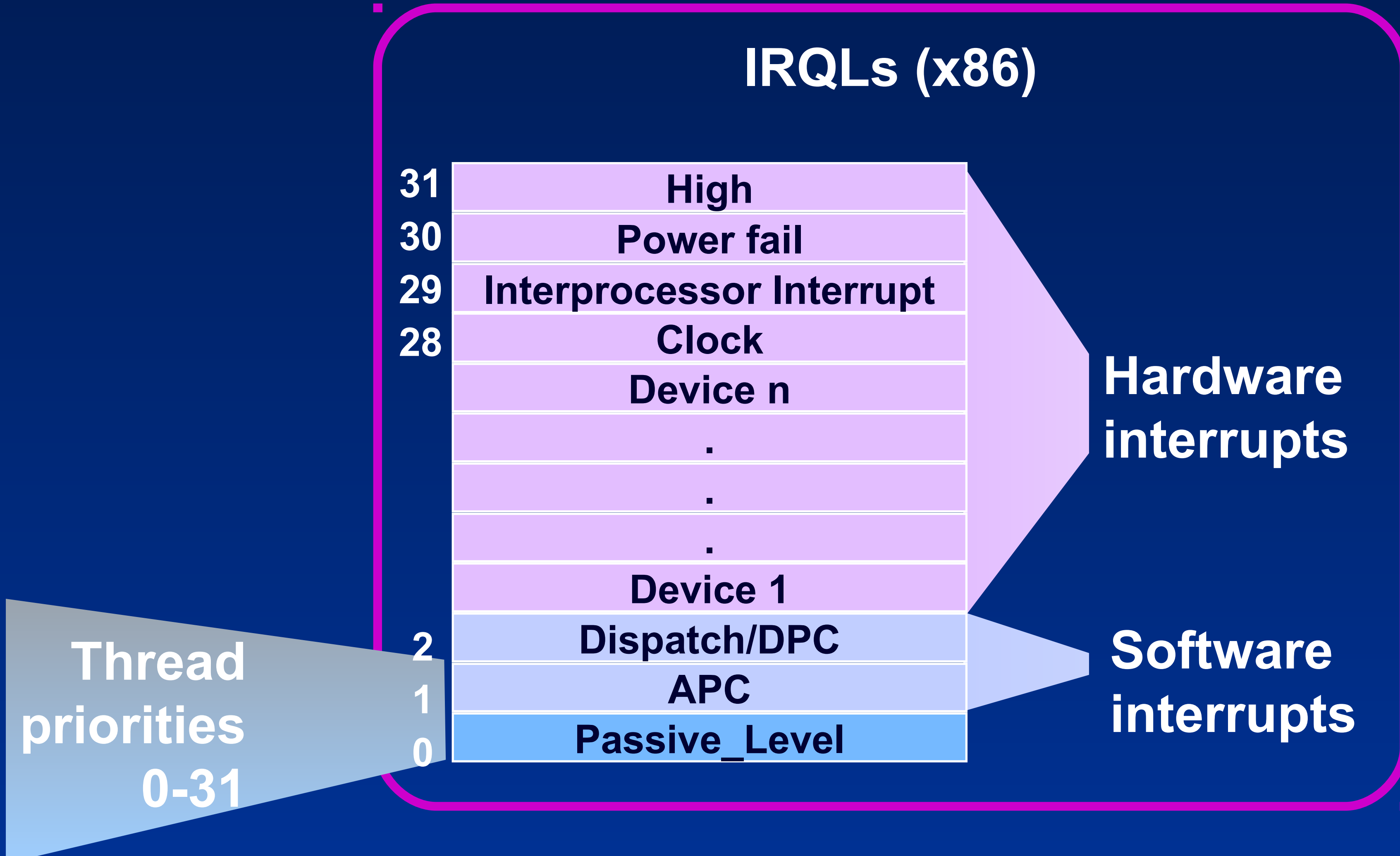
		Win32 Process Classes					
		Realtime	High	Above Normal	Normal	Below Normal	Idle
Win32 Thread Priorities	Time-critical	31	15	15	15	15	15
	Highest	26	15	12	10	8	6
	Above-normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Below-normal	23	12	9	7	5	3
	Lowest	22	11	8	6	4	2
	Idle	16	1	1	1	1	1

- 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)



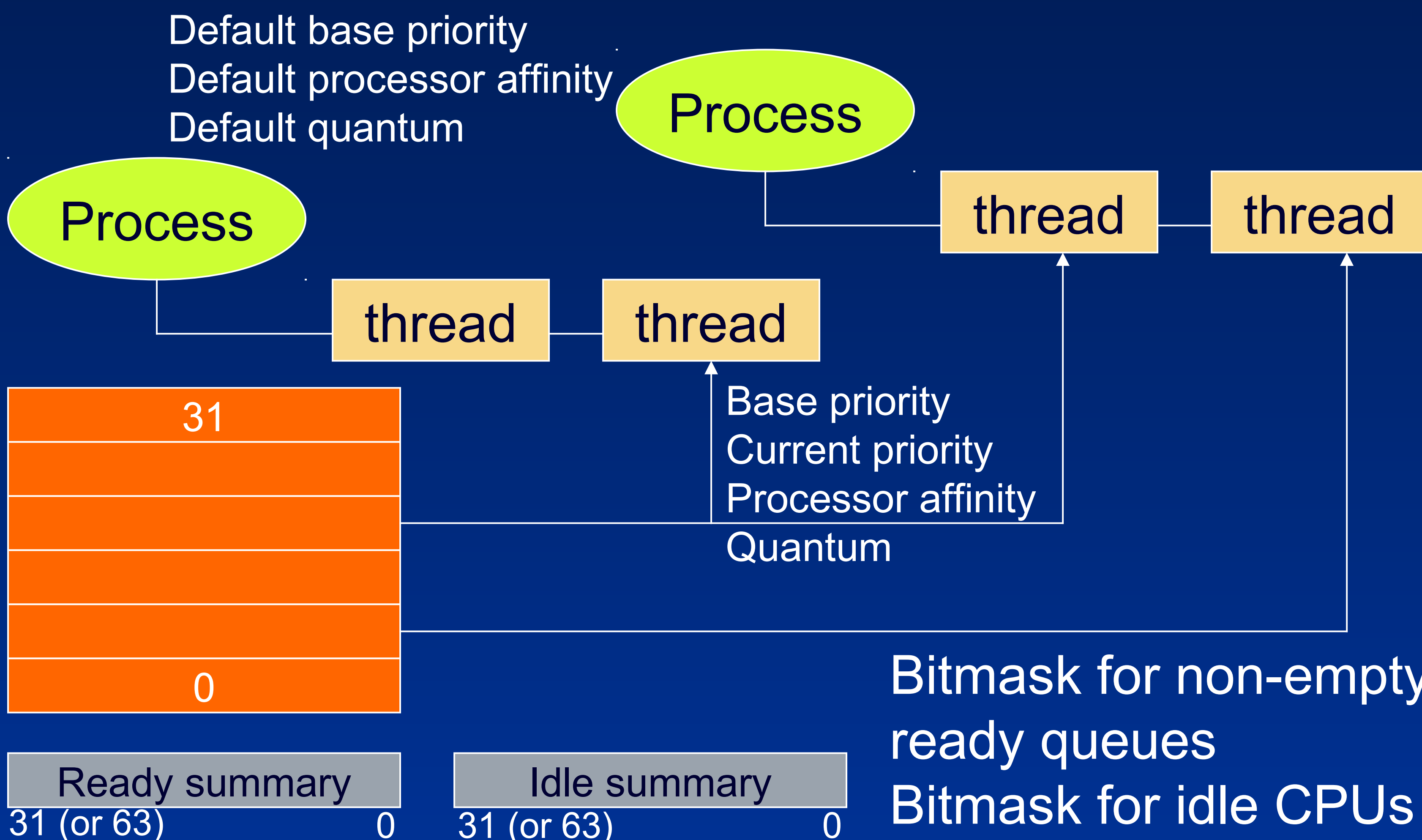
Single Processor Thread Scheduling

- Priority driven, preemptive
  - 32 queues (FIFO lists) of “ready” threads
  - UP: highest priority thread always runs
  - MP: <u>One</u> 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

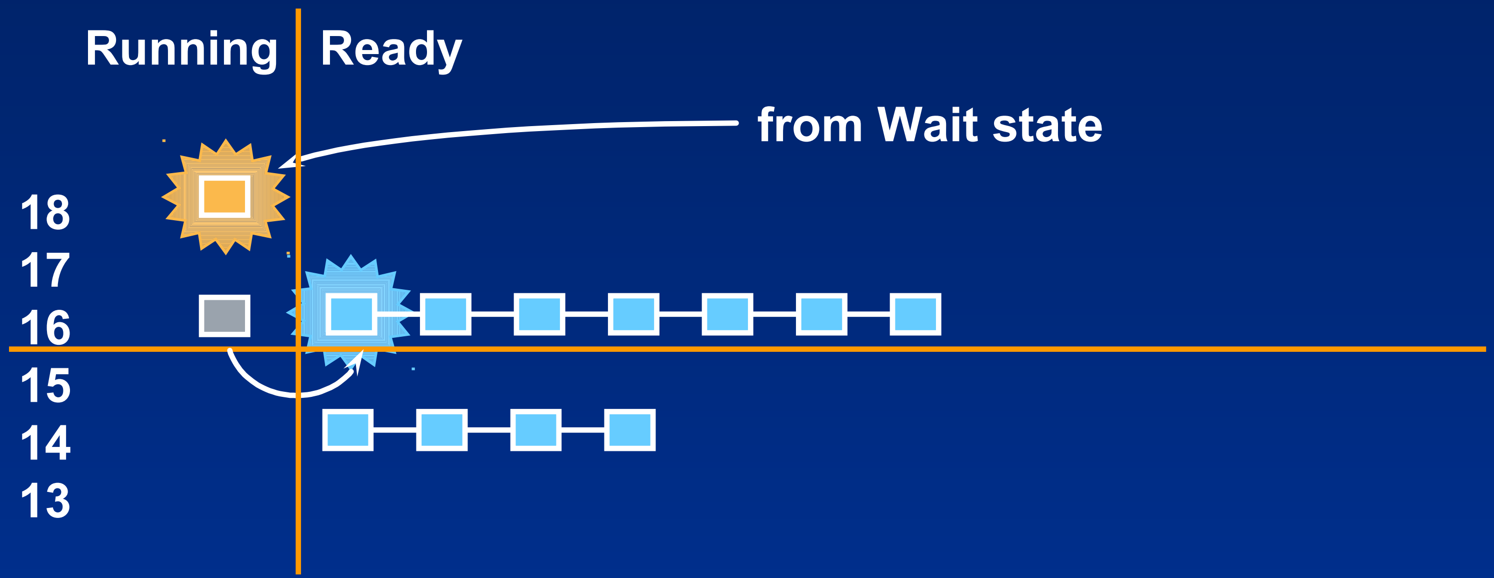


Scheduling Scenarios

- Preemption
  - A thread becomes Ready at a higher priority than the running thread
  - Lower-priority Running thread is preempted
  - Preempted thread goes back to <u>head</u> of its Ready queue
    - <u>action:</u> pick lowest priority thread to preempt
- Voluntary switch
  - Waiting on a dispatcher object
  - Termination
  - Explicit lowering of priority
    - <u>action:</u> 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 <u>tail</u> of ready queue for its new priority
  - May continue running if no equal or higher-priority threads are Ready
    - <u>action:</u> pick next thread at same priority level

Scheduling Scenarios
Preemption

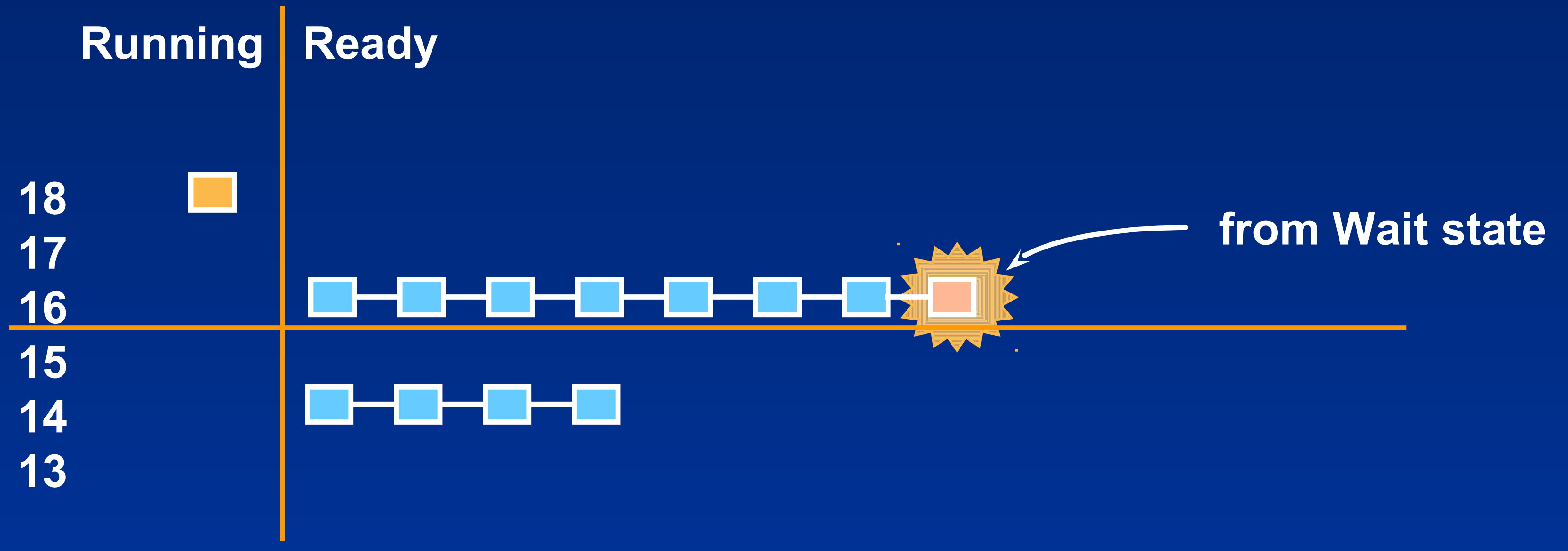
- Preemption is strictly event-driven
  - does not wait for the next clock tick
  - no guaranteed execution period before preemption
  - threads in kernel mode may be preempted (unless they raise IRQL to >= 2)



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

Scheduling Scenarios
Ready after Wait Resolution

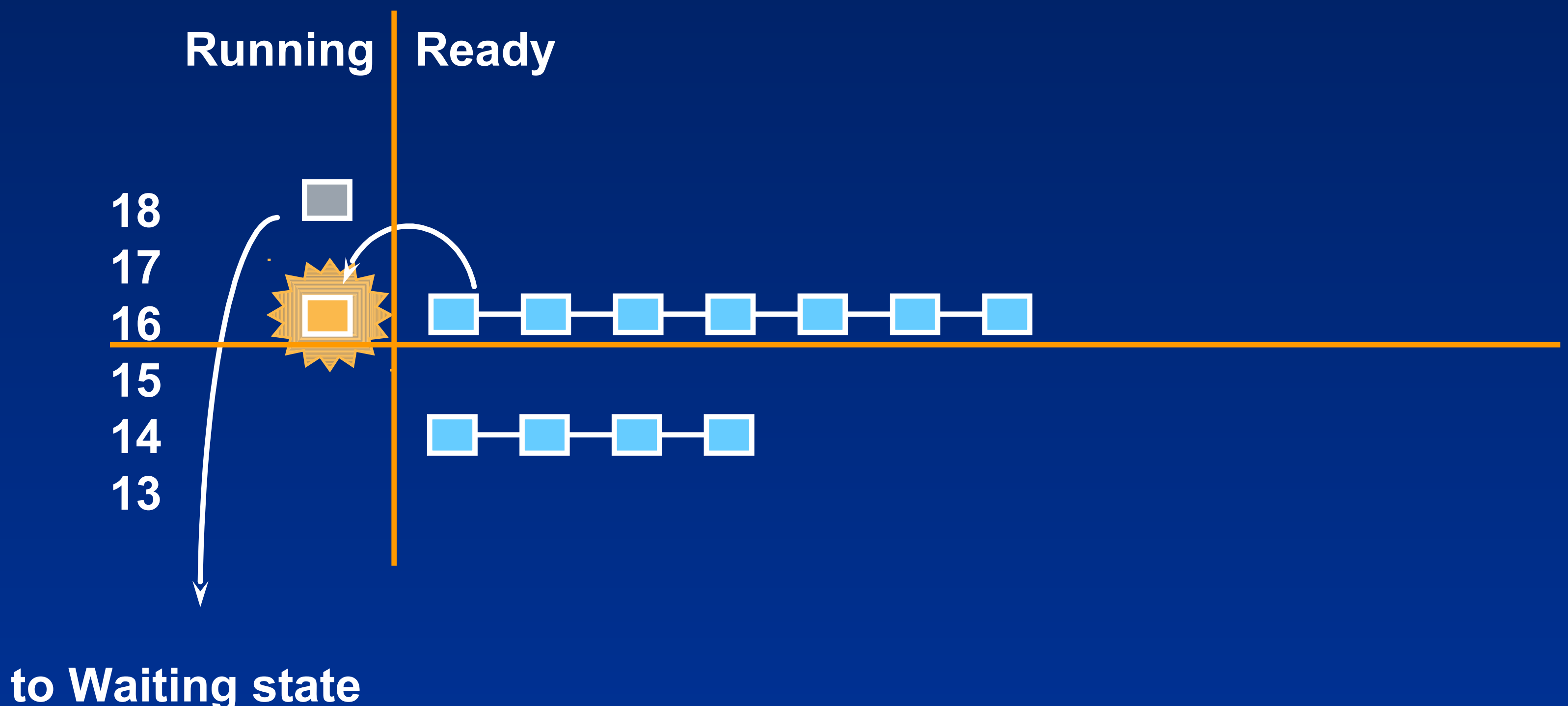
- If newly-ready thread is not of higher priority than the running thread…
- …it is put at the tail of the ready queue for its current priority
  - If priority >=14 quantum is reset (t.b.d.)
  - If priority <14 and you're about to be boosted and didn't already have a boost, quantum is set to process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



Scheduling Scenarios

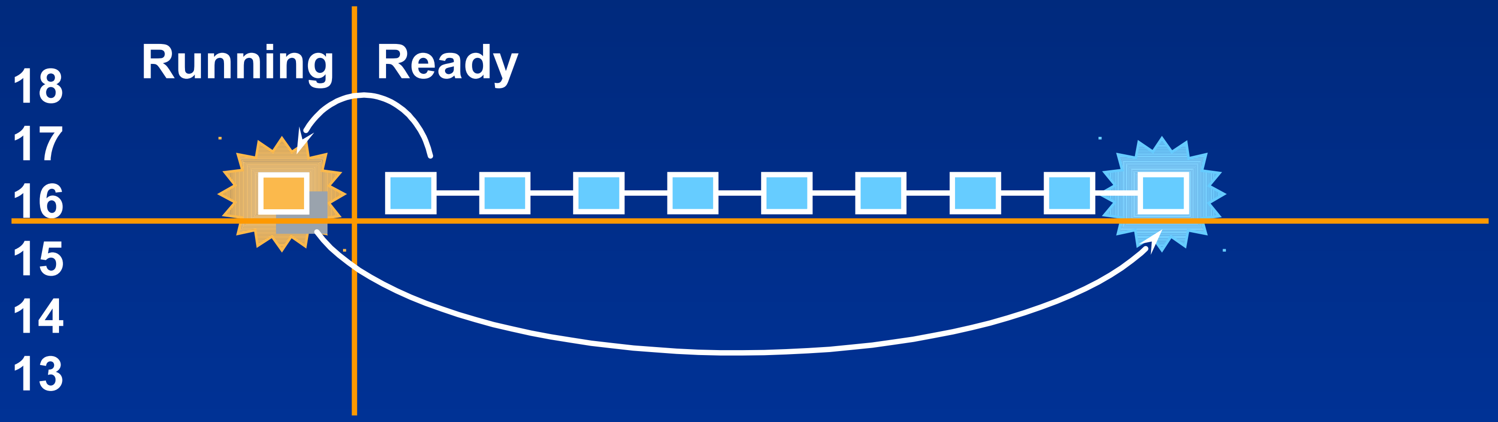
Voluntary Switch

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

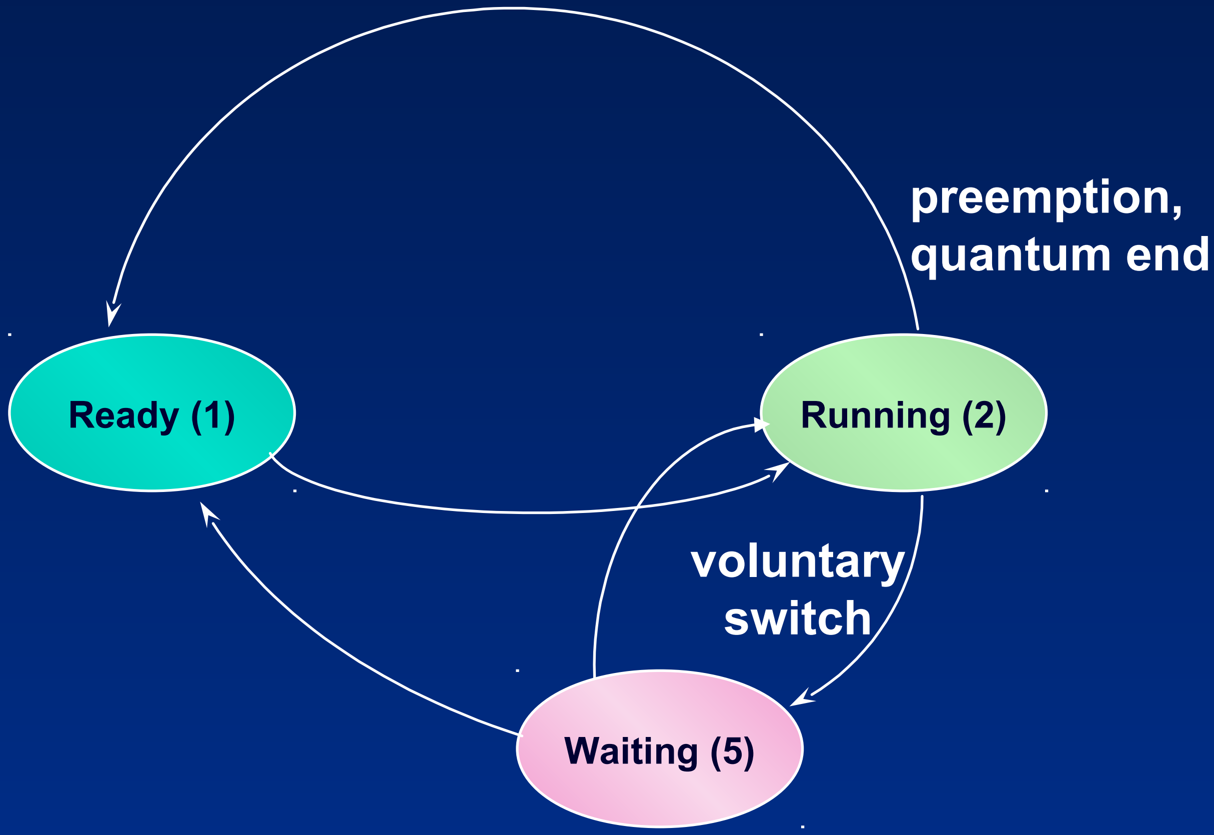


Scheduling Scenarios
Quantum End ("time-slicing")

- 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 NT Client versions is 2 clock ticks, 12 on Server versions
    - 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)

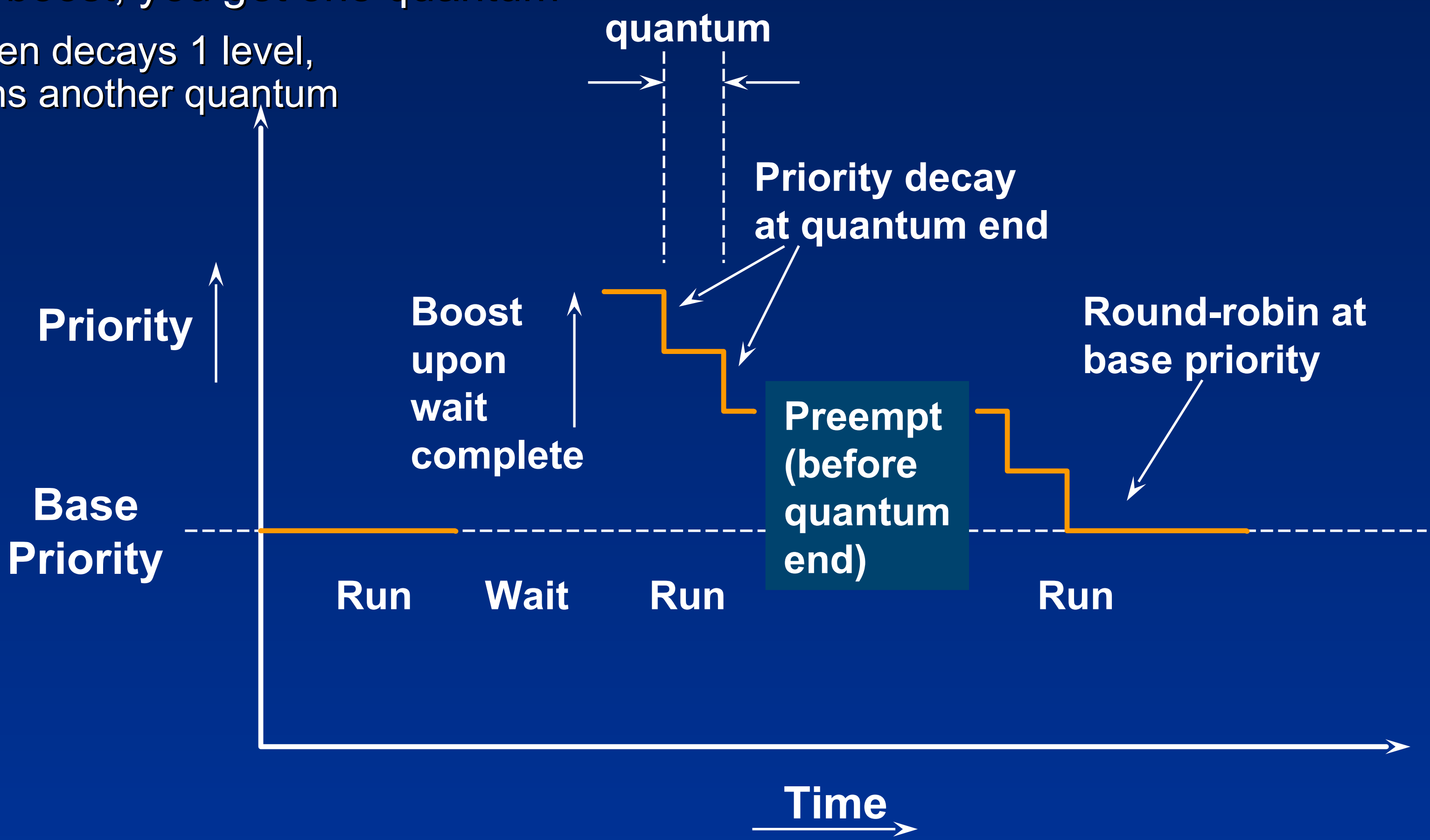
Common boost values (see NTDDK.H)
1: disk, CD-ROM, parallel, video
2: serial, network, named pipe, mailslot
6: keyboard or mouse
8: sound

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



Further Reading

- Pavel Yosifovich, Alex Ionescu, et al., “Windows Internals”, 7th Edition, Microsoft Press, 2017.
  - Chapter 4 – Threads (from pp. 275)
    - Thread scheduling (from pp. 303)
    - Thread states (from pp. 315)
    - Scheduling scenarios (from pp. 361)