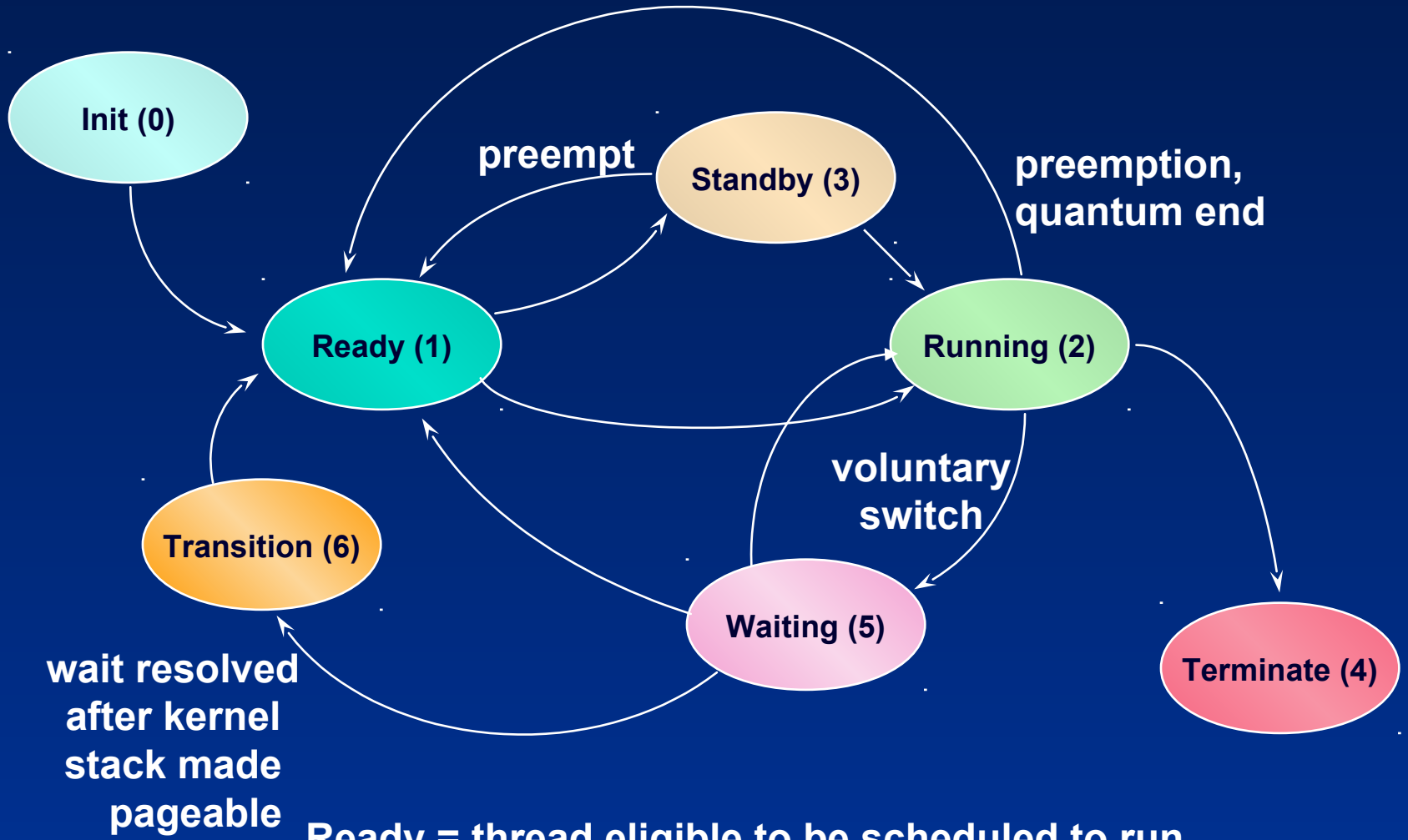
Unit 4: Scheduling and Dispatch

4.5. Advanced Windows Thread Scheduling

Roadmap for Section 4.5.

- 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 another 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

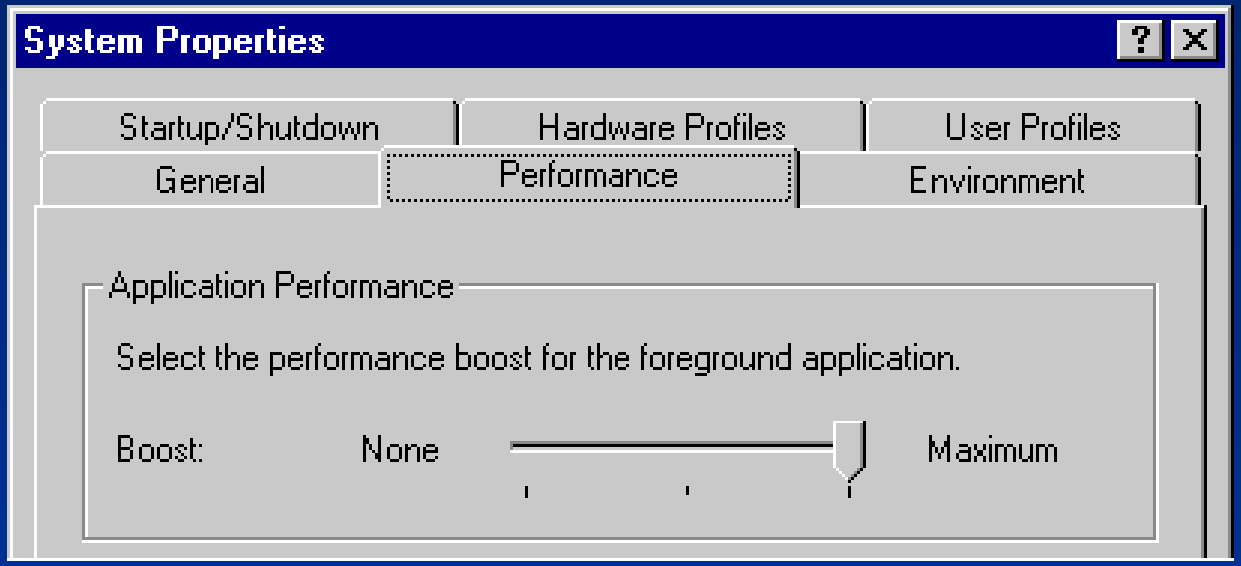
- Quantum internally stored as “3 * number of clock ticks”
  - Default quantum is 6 on NT Client versions, 36 on Server versions
- Thread->Quantum field is decremented by 3 on every clock tick
- Process and thread objects have a Quantum field
  - Process quantum is simply used to initialize thread quantum 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)

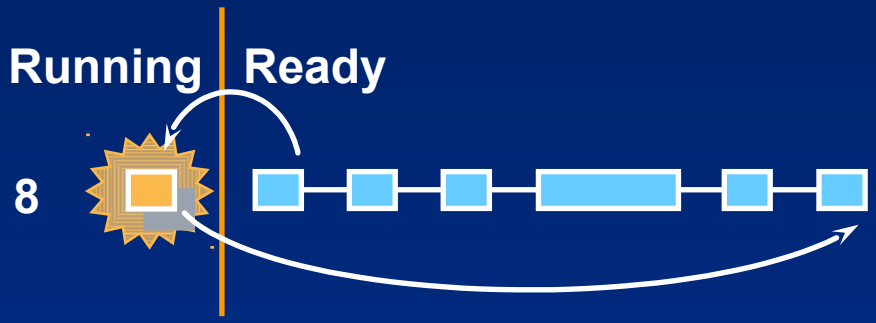
- If normal-priority process owns the foreground window, its threads may be given longer quantum
  - Set by Control Panel / System applet / Performance tab
  - Stored in …\System\CurrentControlSet\Control\PriorityControl Win32PrioritySeparation = 0, 1, or 2
  - New behavior with NT 4.0 (formerly implemented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



Screen snapshot from:
Control Panel | System | Performance tab

Quantum Stretching

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

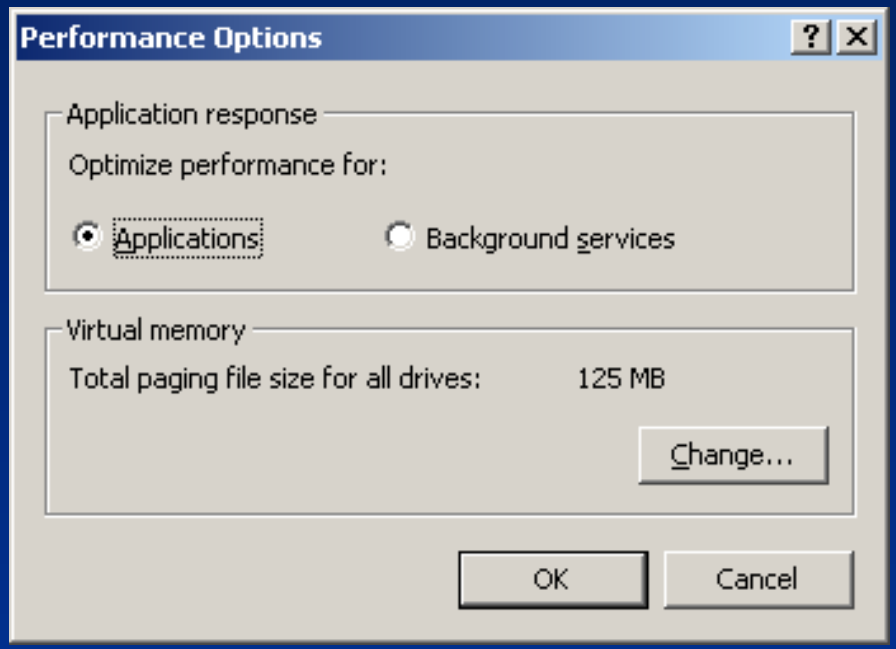


- Quantum stretching does not happen on NT Server versions
  - Quantum on Server is always 12 ticks

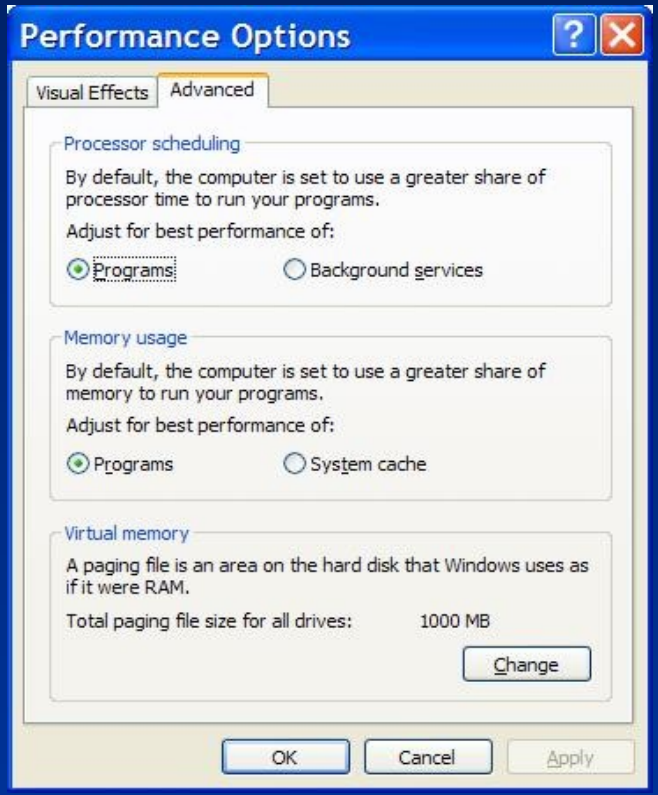
Quantum Selection

- From Windows 2000, you 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:



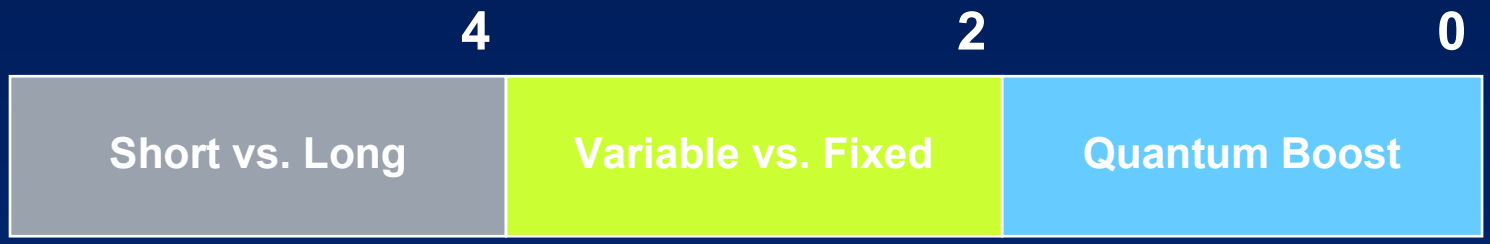
Windows XP:



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

Quantum Control

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



- Short vs. Long

0,3	default (short for NT Client versions, long for NT Server versions)
1	long
2	short

- Variable vs. Fixed

0,3	default (yes for NT Client versions, no for NT Server versions)
1	yes
2	no

- Quantum Boost

0	fixed (overrides above setting)
1	double quantum of foreground threads
2,3	triple 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 NT 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” (example: high priority thread (12 in diagram) waits on something locked by a lower thread (4 in diagram), which can’t run because of a middle priority CPU-bound thread (7 in diagram)), but not deterministically (no priority inheritance)

  - Priority is boosted to 15 (14 prior to NT 4 SP3)
    - Quantum is doubled on Windows 2000/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)

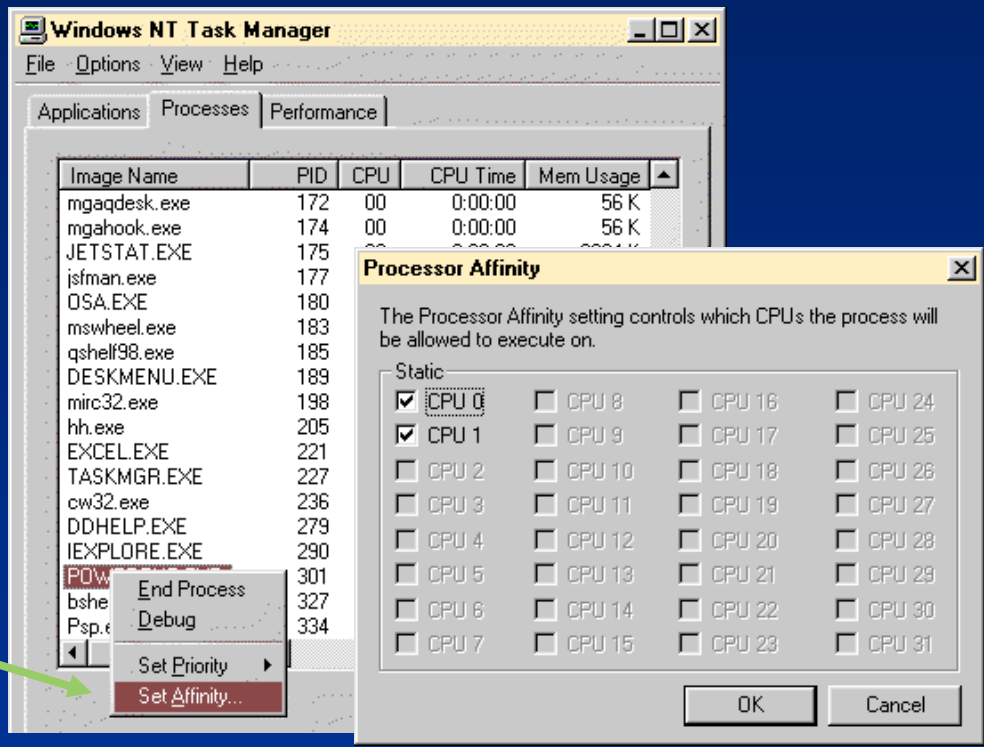
Multiprocessor Scheduling

- 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”

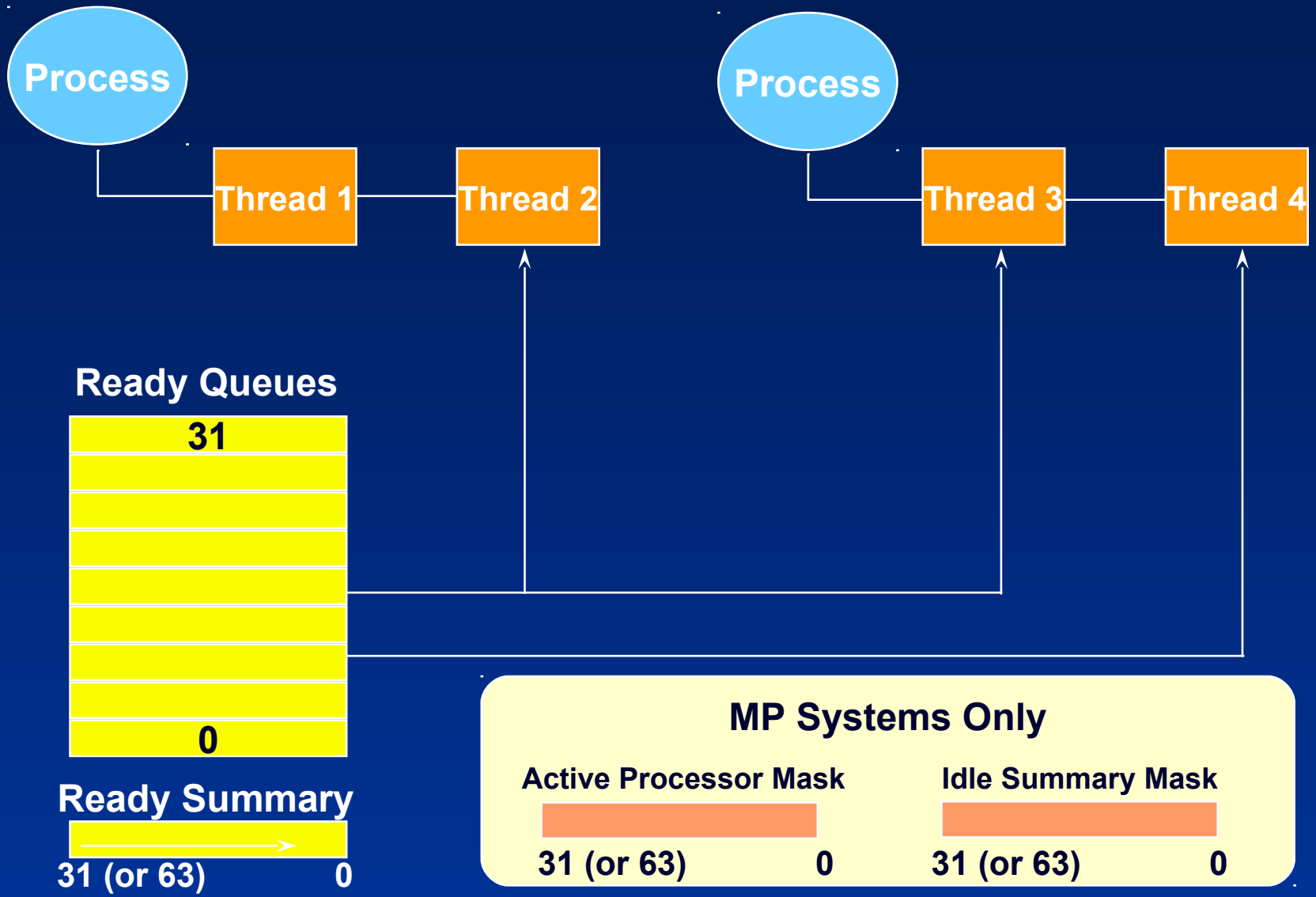
Soft Processor Affinity

- 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



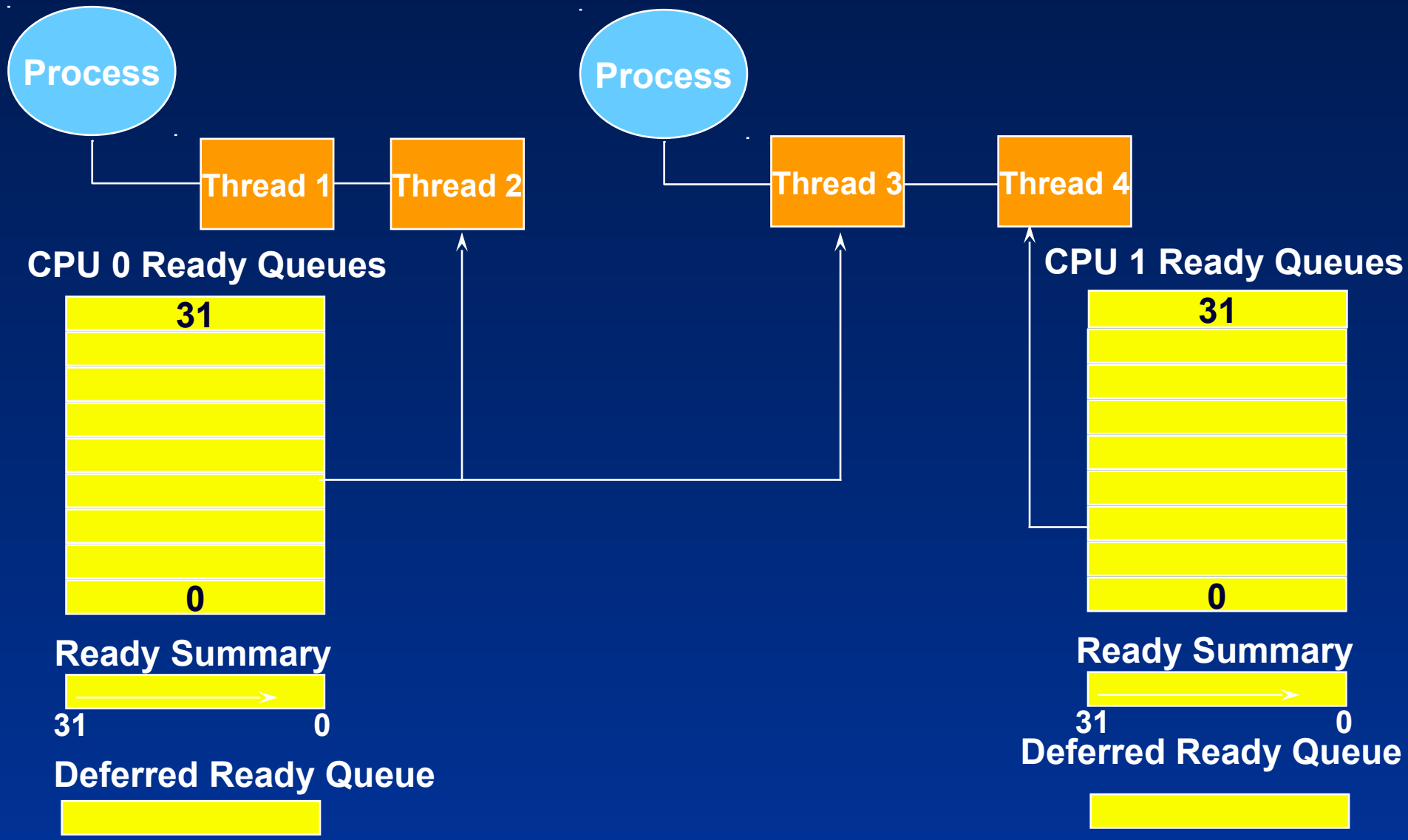
Choosing a CPU for a Ready Thread (Windows 2000 & XP)

- 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 it ran on 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 in the following cases:
  - 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

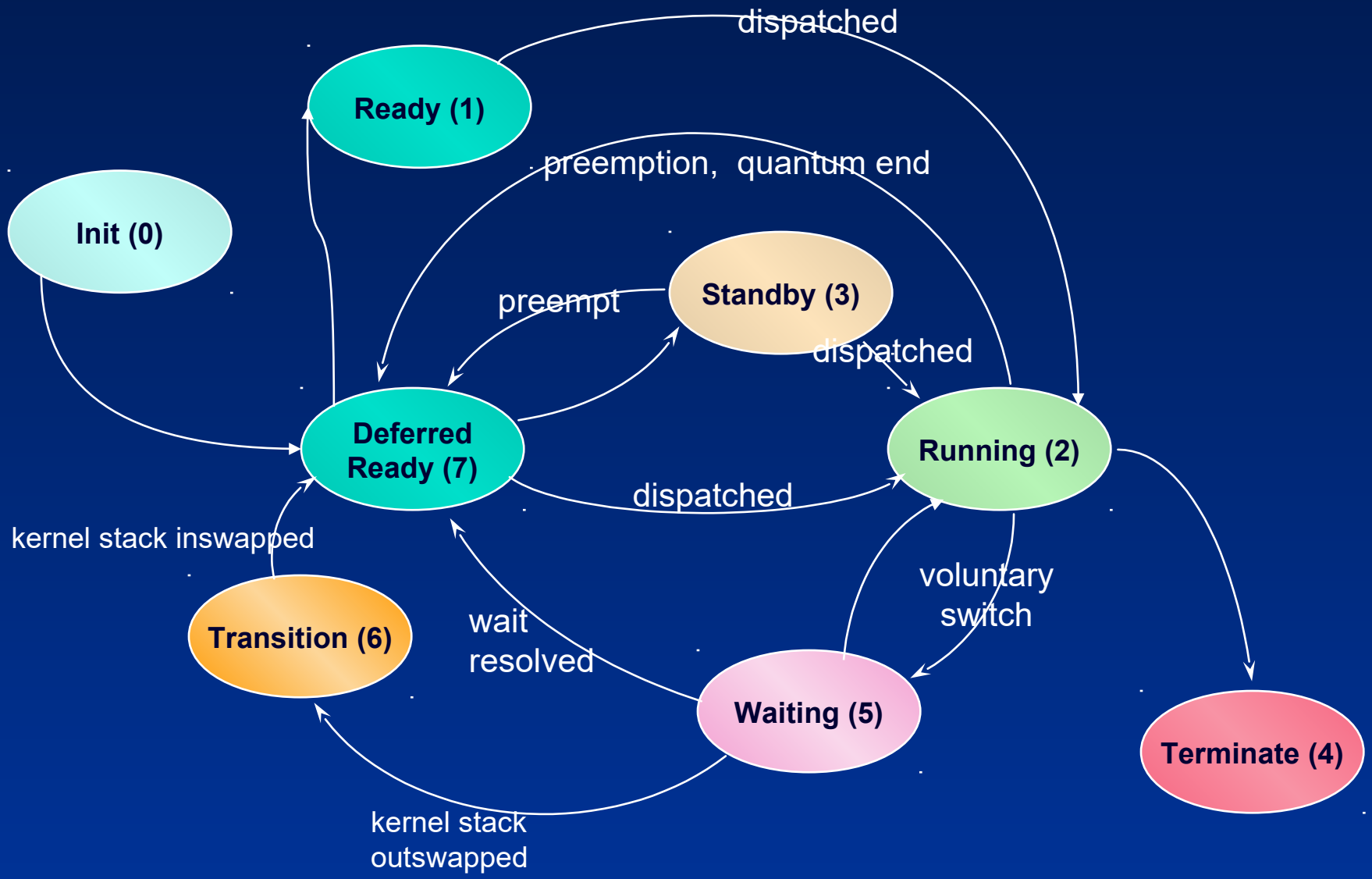
Windows Server2003 Dispatcher Database



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 switch
  - 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)

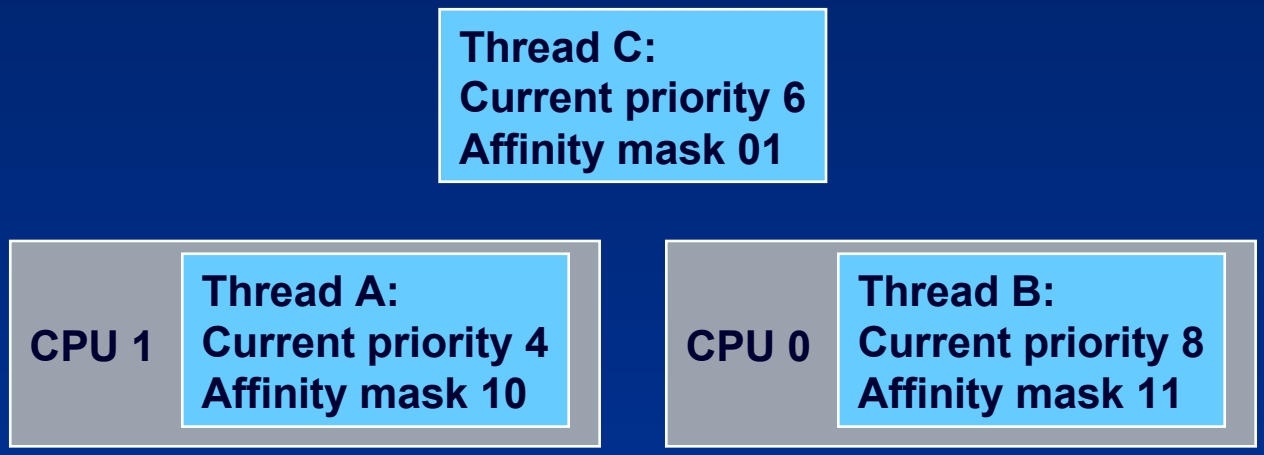


Server 2003 Enhancements

- 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



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)
    - Scheduling scenarios (from pp. 361)
    - Multiprocessor systems (from pp. 376)