Unit OS4: Scheduling and Dispatch

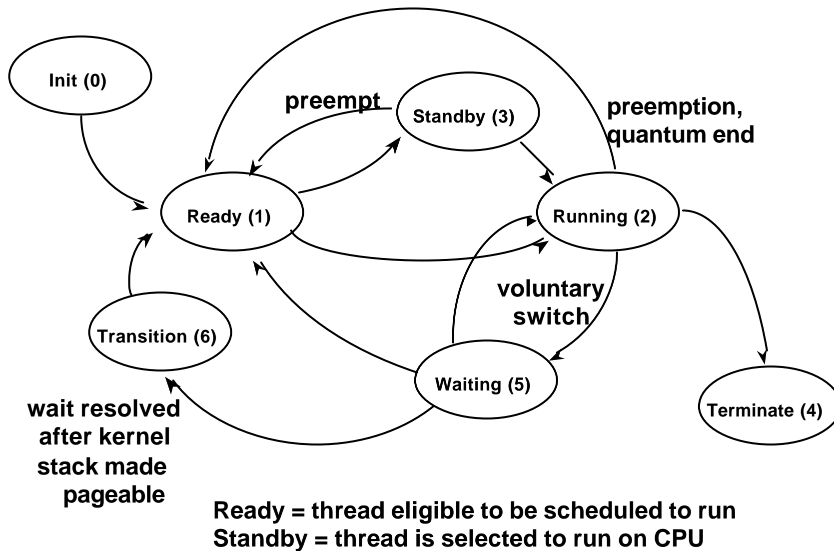
4.5. Advanced Windows Scheduling

Copyright Notice

© 2000-2005 David A. Solomon and Mark Russinovich

- These materials are part of the *Windows Operating System Internals Curriculum Development Kit,* developed by David A. Solomon and Mark E. Russinovich with Andreas Polze
- Microsoft has licensed these materials from David Solomon Expert Seminars, Inc. for distribution to academic organizations solely for use in academic environments (and not for commercial use)

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)

Waiting

- Flexible wait calls
  - Wait for one or multiple objects in one call
  - Wait for multiple can wait for "any" one or "all" at once
    - "All": all objects must be in the signalled state concurrently to resolve the wait
  - All wait calls include optional timeout argument
  - Waiting threads consume no CPU time
- Waitable objects include:
  - Events (may be auto-reset or manual reset; may be set or "pulsed")
  - Mutexes ("mutual exclusion", one-at-a-time)
  - Semaphores (n-at-a-time)
  - Timers
  - Processes and Threads (signalled upon exit or terminate)
  - Directories (change notification)
- No guaranteed ordering of wait resolution
  - If multiple threads are waiting for an object, and only one thread is released (e.g. it's a mutex or auto-reset event), which thread gets released is unpredictable

Looking at Waiting Threads

- For waiting threads, user-mode utilities only display the wait reason
- Example: pstat

Command Prompt

```
C:\WINDOWS\SYSTEM32>pstat
Pstat version 0.3:  memory: 130480 kb  uptime:  0 21:24:36.734
      :
pid:  0 pri: 0 Hnd:    0 Pf:      1 Ws:     16K Idle Process
 tid pri Ctx Swtch StrtAddr    User Time  Kernel Time  State
   0   0   2845450        0  0:00:00.000 20:55:56.375 Running
   0   0   3056193        0  0:00:00.000 21:09:33.234 Running

pid:  2 pri: 8 Hnd:  221 Pf:   1875 Ws:    200K System
 tid pri Ctx Swtch StrtAddr    User Time  Kernel Time  State
   1   0     21214 801c3f6c  0:00:00.000  0:00:39.687 Wait:FreePage
   3  16        51 8010ba7a  0:00:00.000  0:00:00.000 Wait:EventPairLow
   4  16     45518 8010ba7a  0:00:00.000  0:00:00.906 Wait:EventPairLow
      :
pid: 9e pri: 8 Hnd:   78 Pf:   8711 Ws:   1140K Explorer.exe
 tid pri Ctx Swtch StrtAddr    User Time  Kernel Time  State
  48  14    122844 77f052ec  0:00:04.703  0:00:26.312 Wait:UserRequest
  64   8       826 77f052e0  0:00:00.015  0:00:00.140 Wait:UserRequest
  a5  14     23048 77f052e0  0:00:04.140  0:00:11.562 Wait:UserRequest
  a6  14      4976 77f052e0  0:00:00.203  0:00:00.921 Wait:UserRequest
  a7  14      1378 77f052e0  0:00:00.000  0:00:00.000 Wait:LpcReceive
```

- To find out <u>what</u> a thread is waiting on, must use kernel debugger

Looking at Wait Queues

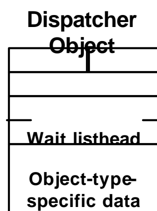
- !thread command to kernel debugger
  - Lists addresses of objects being waited on (if a mutex, shows owner)
  - !irpfind can search IRPs for an event object address

Command Prompt - i386kd -z d:\memory.dmp

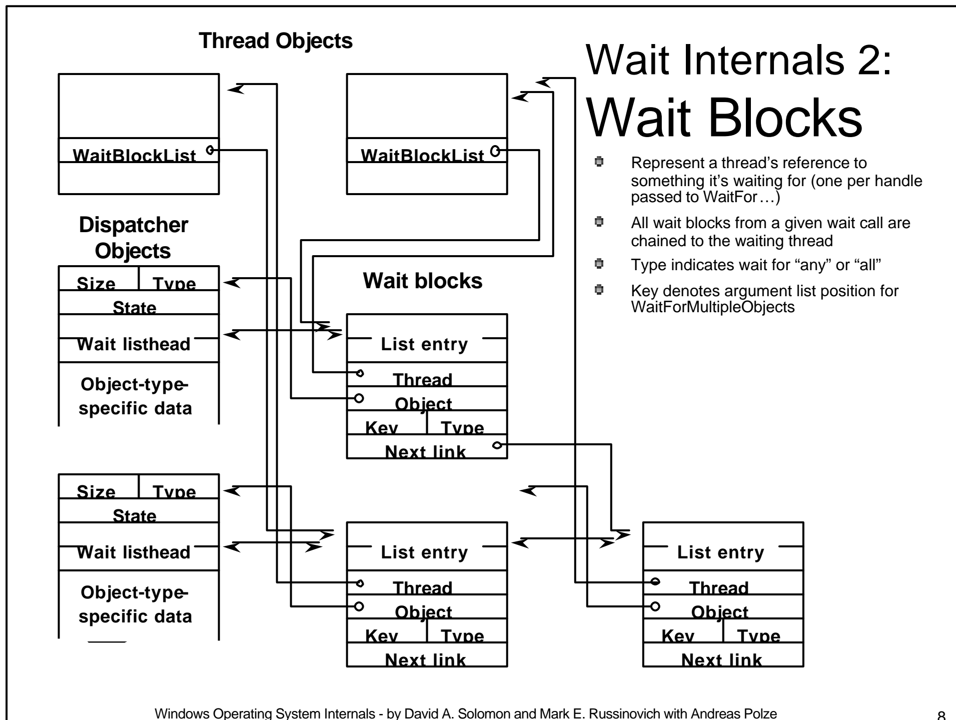
```
0: kd> !thread 80800960
!thread 80800960
THREAD 80800960  Cid 28.95  Teb: 7ffa9000  Win32Thread: 8014f330 WAIT: (UserRequ
est) UserMode Non-Alertable
    807ff300  SynchronizationEvent
    80800a48  NotificationTimer
Not impersonating
Owning Process 808a36a0
WaitTime (seconds)      3396
Context Switch Count    17
UserTime                  0:00:00.0000
KernelTime                0:00:00.0000
Start Address 0x77f052e0
Win32 Start Address 0x77e26473
Stack Init fc4a2000 Current fc4a1e64 Base fc4a2000 Limit fc49f000 Call 0
Priority 9 BasePriority 8 PriorityDecrement 0 DecrementCount 0
cannot get version packet on a crash dump
ChildEBP RetAddr  Args to Child
fc4a1e7c 80117020 00000000 fc4a1ec8 8018d601 ntkrnlmp!KiSwapThread+0x1b1
fc4a1ea0 8018d70d 807ff300 00000006 8018d601 ntkrnlmp!KeWaitForSingleObject+0x1b
8
fc4a1ef0 8013e31e 00000178 00000000 fc4a1ec8 ntkrnlmp!NtWaitForSingleObject+0xa9

fc4a1ef0 77f6819b 00000178 00000000 fc4a1ec8 ntkrnlmp!KiSystemService+0xbe
fc4a1e6c fc4a1ea0 807ff300 80800960 808009cc +0x77f6819b

0: kd> _
```


Wait Internals 1:
Dispatcher Objects

- Any kernel object you can wait for is a “dispatcher object”
  - some exclusively for synchronization
    - e.g. events, mutexes (“mutants”), semaphores, queues, timers
  - others can be waited for as a side effect of their prime function
    - e.g. processes, threads, file objects
  - non-waitable kernel objects are called “control objects”
- All dispatcher objects have a common header
- All dispatcher objects are in one of two states
  - “signalled” vs. “nonsignalled”
  - when signalled, a wait on the object is satisfied
  - different object types differ in terms of what changes their state
  - wait and unwait implementation is common to all types of dispatcher objects



(see \ntddk\inc\ddk\ntddk.h)



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

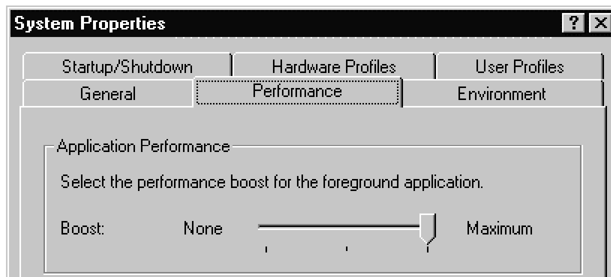
Scheduling Scenarios Quantum Details

- 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)

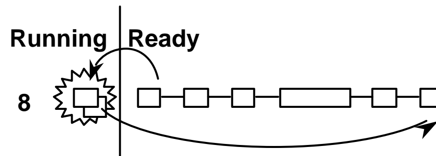
- If normal-priority process owns the foreground window, its threads may be given longer quantum
  - Set by Control Panel / System applet / Performance tab
  - Stored in …\System\CurrentControlSet\Control\PriorityControl Win32PrioritySeparation = 0, 1, or 2
  - New behavior with 4.0 (formerly implemented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

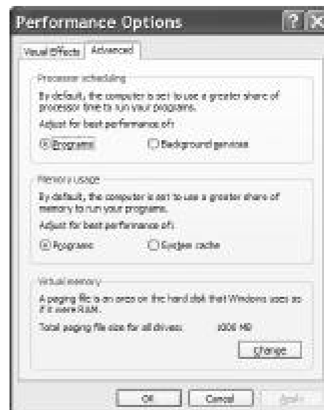
Quantum Selection

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

Windows 2000:

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

Windows XP:



Quantum Control

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

4	2	0
Short vs. Long	**Variable vs. Fixed**	**Quantum Boost**

- Short vs. Long
  - 0,3 default (short for Pro, long for Server)
  - 1 long
  - 2 short
- Variable vs. Fixed
  - 0,3 default (yes for Pro, no for Server)
  - 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 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

- 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

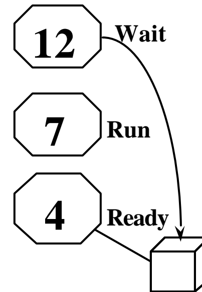
Common boost values (see NTDDK.H)
1: disk, CD-ROM, parallel, Video
2: serial, network, named pipe, mailslot
6: keyboard or mouse
8: sound

Lab: Foreground Priority Boosts

- **See Book "EXPERIMENT: Watching Foreground Priority Boosts and Decays", p.351**
- **See Book "EXPERIMENT: Watching Priority Boosts on GUI Threads", p.353**

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)

Lab: CPU Starvation Resolution

- See Book EXPERIMENT: Watching Priority Boosts for CPU Starvation, p.355
  - CpuStres with two compute-bound threads ("maximum" activity level)
  - One is at lower priority than the other
- See Book EXPERIMENT: "Listening to Priority Boosting", p.357

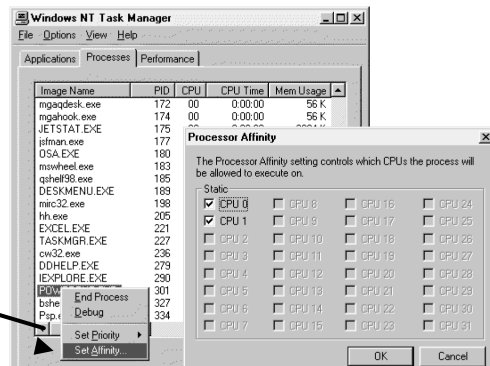
Multiprocessor Scheduling

- 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

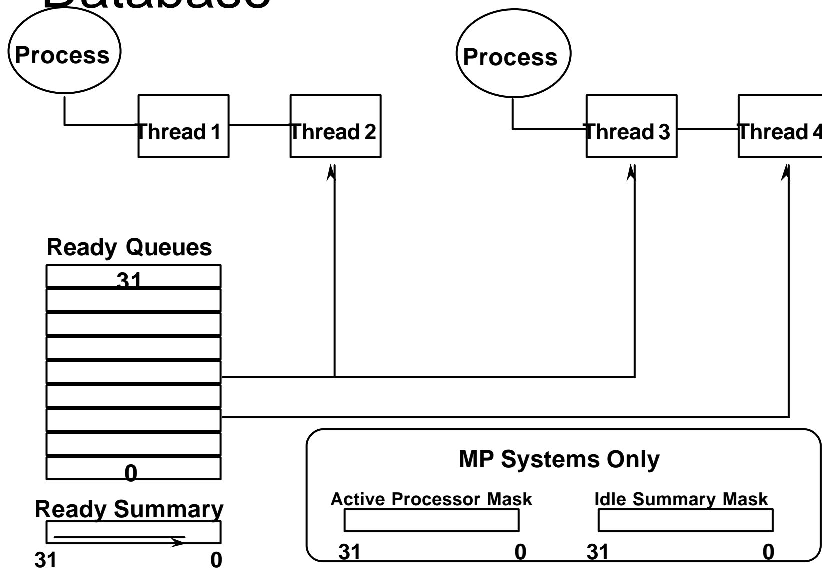
- Can also set an image affinity mask
  - See "Imagecfg" tool in Windows 2000 Server Resource Kit Supplement 1
    - E.g. `Imagecfg –a 2 xyz.exe` will run xyz on CPU 1
- Can also set "uniprocessor only": sets affinity mask to one processor
  - `Imagecfg –u xyz.exe`
  - System chooses 1 CPU for the process
    - 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



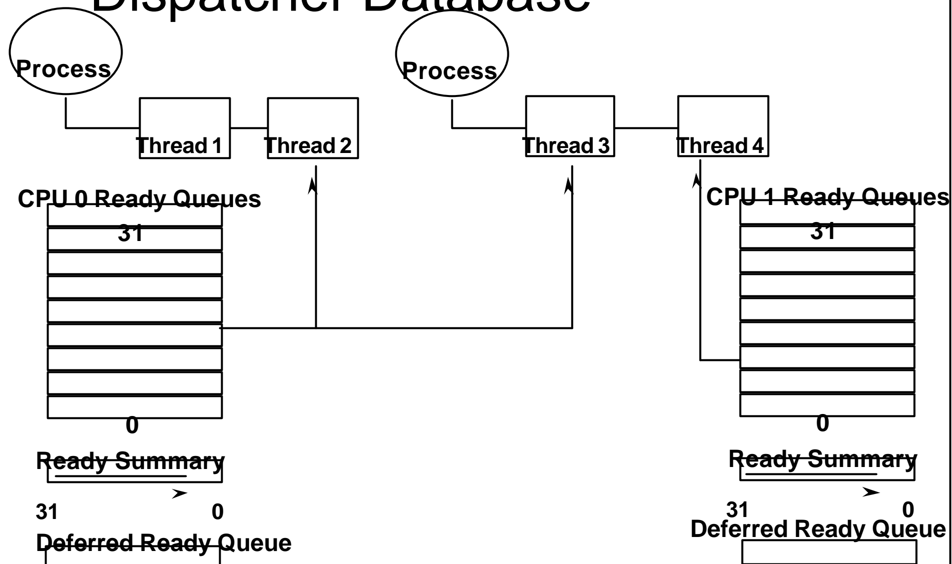
Choosing a CPU for a Ready Thread (Windows 2000)

- 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)

- 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

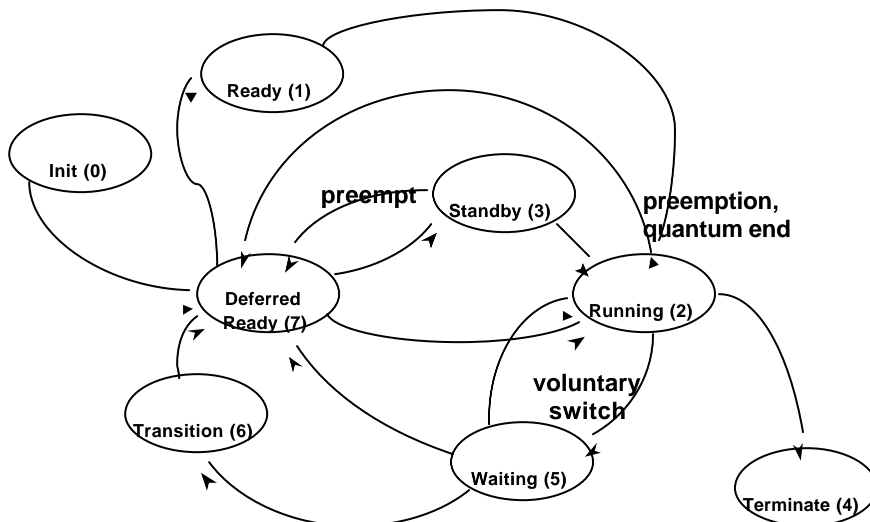
Windows Server 2003 Dispatcher Database



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 <u>the</u> 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