Thread Scheduling
Roadmap for This Lecture

- Overview
- Priorities
- Scheduling States
- Scheduling Data Structures
- Quantum
- Scheduling Scenarios
- Priority Adjustments (boosts and decays)
- Multiprocessor Scheduling
- Lab Demo
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)
Overview of Scheduling

- 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/Windows change priority)
- Processor affinity of a running thread changes

Selecting a thread causes a *context switch*
Priority Levels

32 priority levels: 0 thru 31

Threads within same priority are scheduled following the Round-Robin policy

Non-Real-time Priorities (1-15) are adjusted dynamically – hence called “dynamic” range
  - Priority elevation as response to certain I/O and dispatch events
  - Quantum boosting to optimize responsiveness

Real-time priorities (16-31) are assigned statically to threads
Thread Priority Levels

- **16 “real-time” levels**
- **15 variable levels**
  - Used by zero page thread
  - Used by idle thread(s)
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, Below Normal, Normal, Above Normal, High, Real-time
- Threads have a *relative priority* within the class
  - Idle, Lowest, Below_Normal, Normal, Above_Normal, Highest, and Time_Critical
- Base priority: a function of priority class and relative priority

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
Mapping Win API Priority Levels to Kernel Priority Levels

<table>
<thead>
<tr>
<th>Process Priority Classes</th>
<th>Realtime</th>
<th>High</th>
<th>Above Normal</th>
<th>Normal</th>
<th>Below Normal</th>
<th>Idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time-critical</td>
<td>31</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>Highest</td>
<td>26</td>
<td>15</td>
<td>12</td>
<td>10</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>Above-normal</td>
<td>25</td>
<td>14</td>
<td>11</td>
<td>9</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Normal</td>
<td>24</td>
<td>13</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>Below-normal</td>
<td>23</td>
<td>12</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Lowest</td>
<td>22</td>
<td>11</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Idle</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- **Table shows base priorities** ("current" thread priority may be higher if base is < 15, since it’s in the "dynamic range")
- **Many utilities** (such as Process Viewer) show the "current priority" of threads rather than the base (Performance Monitor can show both)
- **Drivers can set to any value with KeSetPriorityThread**
- **Process base priority default to middle of priority range**
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, i.e., priority i
  - 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 a thread if selected
- in certain cases, scans per-CPU ready queues for next thread

Zero page thread -- one per Windows 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)

IRQLs (x86)

<table>
<thead>
<tr>
<th>IRQL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>High</td>
</tr>
<tr>
<td>30</td>
<td>Power fail</td>
</tr>
<tr>
<td>29</td>
<td>Interprocessor Interrupt</td>
</tr>
<tr>
<td>28</td>
<td>Clock</td>
</tr>
<tr>
<td>27</td>
<td>Device n</td>
</tr>
<tr>
<td>26</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Device 1</td>
</tr>
<tr>
<td>22</td>
<td>Dispatch/DPC</td>
</tr>
<tr>
<td>21</td>
<td>APC</td>
</tr>
<tr>
<td>20</td>
<td>Passive_Level</td>
</tr>
</tbody>
</table>

Thread priorities 0-31

Hardware interrupts

Software interrupts
Thread States & Transitions

- **Ready** (1) = thread eligible to be scheduled to run
- **Deferred ready** (7) = thread selected to run but not scheduled
- **Standby** (3) = thread is selected to run on CPU (one per processor)
- **Running** (2) = voluntary switch
- **Waiting** (5) = gate waiting (8)
- **Transition** (6) = wait resolved but kernel stack paged out
- **Terminate** (4) = preemption, quantum end

Key:
- Preempt
- Wait resolved
- But kernel stack paged out

**Definitions:**
- *Ready* = thread eligible to be scheduled to run
- *Deferred ready* = thread selected to run but not scheduled
- *Standby* = thread is selected to run on CPU (one per 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 priority
Thread Scheduling

- No centralized 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

- Kernel maintains *dispatcher database*
  - Threads waiting to execute
  - Which processors executing which threads
Dispatcher Database

Process

Thread 1

Thread 2

CPU 0 Ready Queues

31

0

Ready Summary

Deferred Ready Queue

31

0

Bit mask for non-empty ready queues

Process

Thread 3

Thread 4

CPU 1 Ready Queues

31

0

Ready Summary

Deferred Ready Queue

31

0
Quantum Details

- Amount of time a thread gets to run before Windows checks for rescheduling
- Quantum internally stored as “3 * number of clock ticks”
  - Default quantum is 3 * 2 = 6 on Vista, 3 * 12 = 36 on Server
- Process and thread objects have a Quantum field
  - Process quantum is simply used to initialize thread quantum for all threads in the process
- Thread → Quantum field is decremented by 3 on every clock tick
- 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
Quantum Details

When Thread $\rightarrow$ Quantum reaches zero
(or less than zero):

- you’ve experienced quantum end
- waiting threads at same priority $\rightarrow$ context switch
- Thread $\rightarrow$ Quantum = Process $\rightarrow$ 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:

- are not charged to the current thread’s time
Quantum Boosting (favoring foreground applications)

- Window brought into foreground
- All threads in the same process: quantum x 3, e.g. 6 clock ticks
- More responsiveness to foreground applications
- Quantum boosting does not happen on Server
  - Quantum on Server is always 12 ticks
Quantum Selection

Windows can choose short or long quantums (e.g. for Terminal Servers)

Screen snapshot from:
System properties | Advanced | Performance settings | Advanced
Quantum Control

Finer grained quantum control can be achieved by modifying `HKLM\System\CurrentControlSet\Control\PriorityControl\Win32PrioritySeparation`

<table>
<thead>
<tr>
<th>6 bit value</th>
<th>4</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Short vs. Long</strong></td>
<td><strong>Variable vs. Fixed</strong></td>
<td><strong>Quantum Boost</strong></td>
<td></td>
</tr>
<tr>
<td>0,3</td>
<td>default (short for Vista, long for Server)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>long</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>short</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0,3</td>
<td>default (yes for Vista, no for Server)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>fixed (overrides above setting)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>double quantum of foreground threads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2,3</td>
<td>triple quantum of foreground threads</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 than Idle priority class
- Only applies if system running with fixed quantum (the default on Servers)

Values are 0-9
- 5 is default

<table>
<thead>
<tr>
<th>Scheduling class</th>
<th>Quantum units</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>36</td>
</tr>
<tr>
<td>6</td>
<td>42</td>
</tr>
<tr>
<td>7</td>
<td>48</td>
</tr>
<tr>
<td>8</td>
<td>54</td>
</tr>
<tr>
<td>9</td>
<td>60</td>
</tr>
</tbody>
</table>
Scheduling Scenarios (I)

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)
Scheduling Scenarios (II)

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

Termination
- Thread finishes running: returned from main() or killed
- Moves from running state to terminated state
- No more handles on the thread object:
  - action: thread and assoc. structures de-allocated
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 $\geq 2$)

- 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 $\geq 14$ 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 to Waiting state
Scheduling Scenarios

Quantum End

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 Windows is 2 quantum units, 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)
Dynamic priority adjustments (boost and decay) are applied to threads in “dynamic” classes

- Threads with base priorities 1-15
- Disable if desired with SetThreadPriorityBoost or SetProcessPriorityBoost

Seven cases:

- I/O completion
- Wait completion on events or semaphores
- When a thread has been waiting for an executive resource for too long
- When threads in the foreground process complete a wait
- When GUI threads wake up for windows input
- For CPU starvation avoidance
- Multimedia playback by Multimedia Class Scheduler Service (MMCSS)

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: After I/O Completion

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

- Applied to current priority (not base priority)
- After boost, run for one quantum
- Decays one priority level and continue until base priority level
Priority Boost and Decay

Behavior of these boosts:

- Applied to thread’s current priority
  - will not take you above priority 15
- After a boost, you get one quantum
  - Then decays 1 level, runs another quantum
Priority Boosting: Waiting on Executive Resources

- Five second wait (to avoid deadlock)

- At the end of wait:
  - Acquire dispatcher lock
  - Boost owning thread
  - Wait again

- Boosting operation:
  - Applied to base priority (not current priority)
  - Raise priority to 14
  - Only applied if pri < waiting thread and < 14
  - Quantum reset: can run a full quantum
Priority Boosting: Foreground Threads after Wait

- KiUnwaitThread boost current priority by PsPrioritySeparation
- Improve responsiveness of interactive apps
- Applies to all windows systems
- Can’t disable this boost
Priority Boosting: CPU Starvation Avoidance

- **Balance Set Manager** system thread looks for “starved” threads
  
  - Balance set manager 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” (see figure)
  - Priority is boosted to 15
  - Set quantum to 4
  - At quantum end, decays directly to base 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

- Dispatcher database:
  - Ready queues
  - Ready summary
  - Active processor mask: one bit for each usable processor
  - Idle summary: one bit for each idle processor
Multiprocessor 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) (PRCB = Processor Control Block).
  - 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 change system-wide state of a synchronization objects (mutexes, events and semaphores) and their waiting queues.

- Bottom line: dispatcher lock is now held for a MUCH shorter time.
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:
  
  \textit{SetThreadAffinityMask}, \textit{SetProcessAffinityMask}, \textit{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 during compilation
  - Search “Portable Executable and Common Object File Format Specification”
- Can also set “uniprocessor only” flag at compile time
  - sets affinity mask to one processor
  - System chooses 1 CPU for the process
    - Go 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:

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

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

- For Hyperthreaded systems: first logical processor on the next physical processor
- For NUMA systems: ideal node is chosen for a new process, ideal processor from ideal node assigned to the thread in this process
Choosing a CPU for a Ready Thread

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

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
Lab Demo

- Watching Foreground Priority Boosts and Decays
- “Listening” to MMCSS Priority Boosting
Lab

- Tchar.h
- Tlhelp32.h
- Traverse Processes (Simple & MSDN)
- How to Terminate a Process using PID
To simplify the transporting of code for international use, the Microsoft run-time library provides Microsoft-specific generic-text mappings for many data types, routines, and other objects. You can use these mappings, which are defined in Tchar.h, to write generic code that can be compiled for single-byte, multibyte, or Unicode character sets, depending on a manifest constant that you define by using a `#define` statement. Generic-text mappings are Microsoft extensions that are not ANSI compatible.
By using the Tchar.h, you can build single-byte, Multibyte Character Set (MBCS), and Unicode applications from the same sources. Tchar.h defines macros (which have the prefix _tcs) that, with the correct preprocessor definitions, map to str, _mbs, or wcs functions, as appropriate. To build MBCS, define the symbol _MBCS. To build Unicode, define the symbol _UNICODE. To build a single-byte application, define neither (the default). By default, _MBCS is defined for MFC applications.
**Tlhelp32.h**

- **Tool Help Functions**
- Used for creating tools for Windows

### ToolHelp Functions

**Windows Mobile 6.5**

A version of this page is also available for **Windows Embedded CE 6.0 R3**

4/8/2010

The following table shows the ToolHelp functions with a description of the purpose of each:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CloseToolhelp32Snapshot</td>
<td>Closes a handle to a snapshot.</td>
</tr>
<tr>
<td>CreateToolhelp32Snapshot</td>
<td>Takes a snapshot of the processes, heaps, modules, and threads used by the processes.</td>
</tr>
<tr>
<td>Heap32First</td>
<td>Retrieves information about the first block of a heap allocated by a process.</td>
</tr>
</tbody>
</table>
Further Reading


- Chapter 5 - Processes, Thread, and Jobs (from pp. 391)
- Thread Scheduling (from pp. 391)
Source Code References

Windows Research Kernel sources

\base\ntos\ke\i386, \base\ntos\ke\amd64:

- Ctxswap.asm – Context Swap
- Clockint.asm – Clock Interrupt Handler

\base\ntos\ke

- procobj.c - Process object
- thredobj.c, thredsup.c – Thread object
- Idsched.c – Idle scheduler
- Wait.c – quantum management, wait resolution
- Waitsup.c – dispatcher exit (deferred ready queue)

\base\ntos\inc\ke.h – structure/type definitions