Windows Internals

Module 7: Processes & Threads (Part 3)

Pavel Yosifovich
CTO, CodeValue
pavely@codevalue.net
http://blogs.Microsoft.co.il/blogs/pavely

pluralsight
hardcore developer training

Priority Boosts

- **Windows boosts the priority of threads in a number of scenarios**
  - Completion of I/O operations
  - After waiting for events or semaphores
  - During waiting for an executive resource
  - After threads in the foreground process complete a wait operation
  - When GUI threads wake up because of windowing activity
  - When a thread is starved
- **Thread priorities in the realtime range don't receive any boost**

Completion of I/O Request or Wait

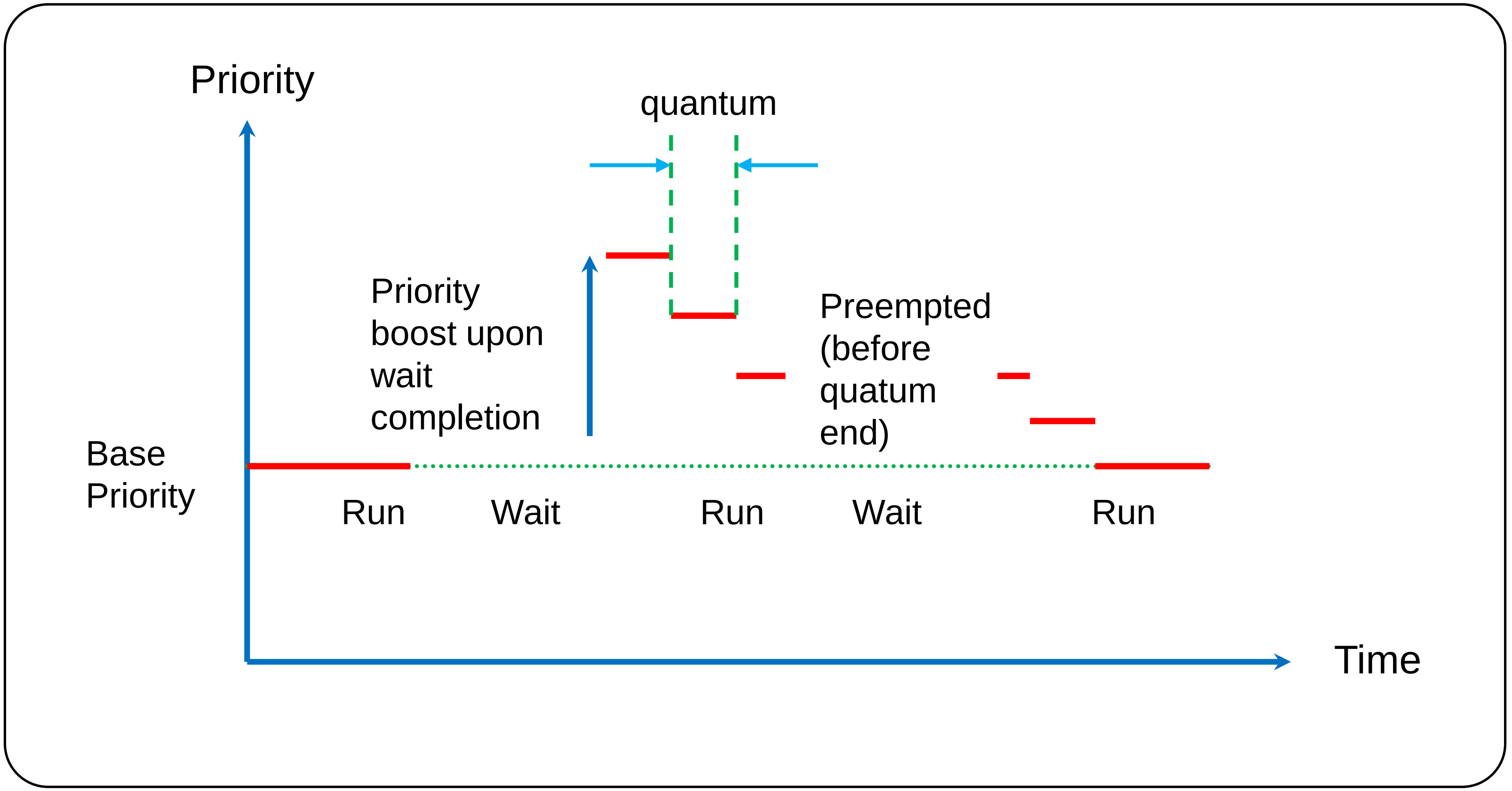
- **Occurs when an I/O or wait completes**
  - Can be specified by a driver or the Executive
    - **KeSetEvent** (Event, Increment)
    - **IoCompleteRequest** (Irp, PriorityBoost)
- **After a boost, thread runs for one quantum at that priority**
  - Then drops one level, runs another quantum
  - Then drops another level, etc., until back to base priority
- **Recommended boost values defined in <ntddk.h>**

#define IO_SERIAL_INCREMENT 2

#define EVENT_INCREMENT 1

#define IO_KEYBOARD_INCREMENT 6

Thread Priority Boost and Decay



Foreground Process Wait Boost

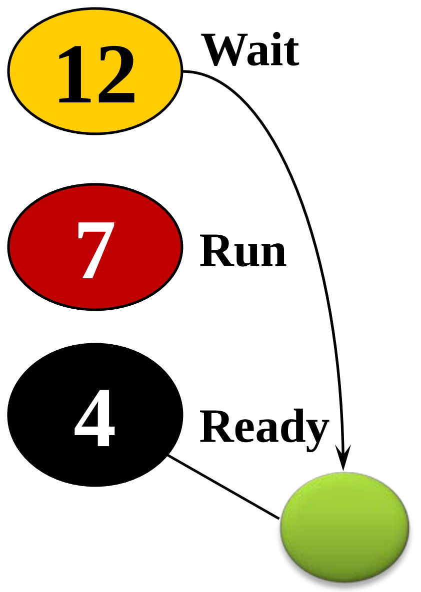
- **Foreground process**
  - The process which contains the thread who is the owner (and creator) of the foreground window
- **After a thread running in the foreground process completes a wait on a kernel object**
- **Receives a boost in the amount of the value set in the registry for foreground priority boost**
  - +2 by default

GUI Thread Wakeup

- GUI threads receive a priority boost of 2 when they wake up due to a Window message arriving
- Provided by Win32k.sys
- Improves their chance of running sooner, giving a better responsiveness to the user

Priority Inversion / Starvation

- **Priority Inversion**
  - High-priority thread waits on something locked by a lower priority thread which can't run because of a middle priority thread running
- **Boosts thread to avoid priority inversion**
  - Threads staying in ready state a long time (four seconds) get a big boost to priority 15
    - Get to run for 3 quantums at this special boost
    - Then priority drops to base
- **Technically, starvation avoidance**
- **Implemented by the balance set manager**
  - Scans at most 16 threads per pass
  - Boosts at most 10 threads per pass



Demo

Priority boosts

Multiprocessing - Soft affinity

- **Ideal Processor**
  - Every thread has an ideal processor
  - Default value set in round-robin within each process
    - A random starting value
  - Can Override with **SetThreadIdealProcessor**
  - On hyper-threaded systems, the next ideal processor selected is from the next physical core (not logical)

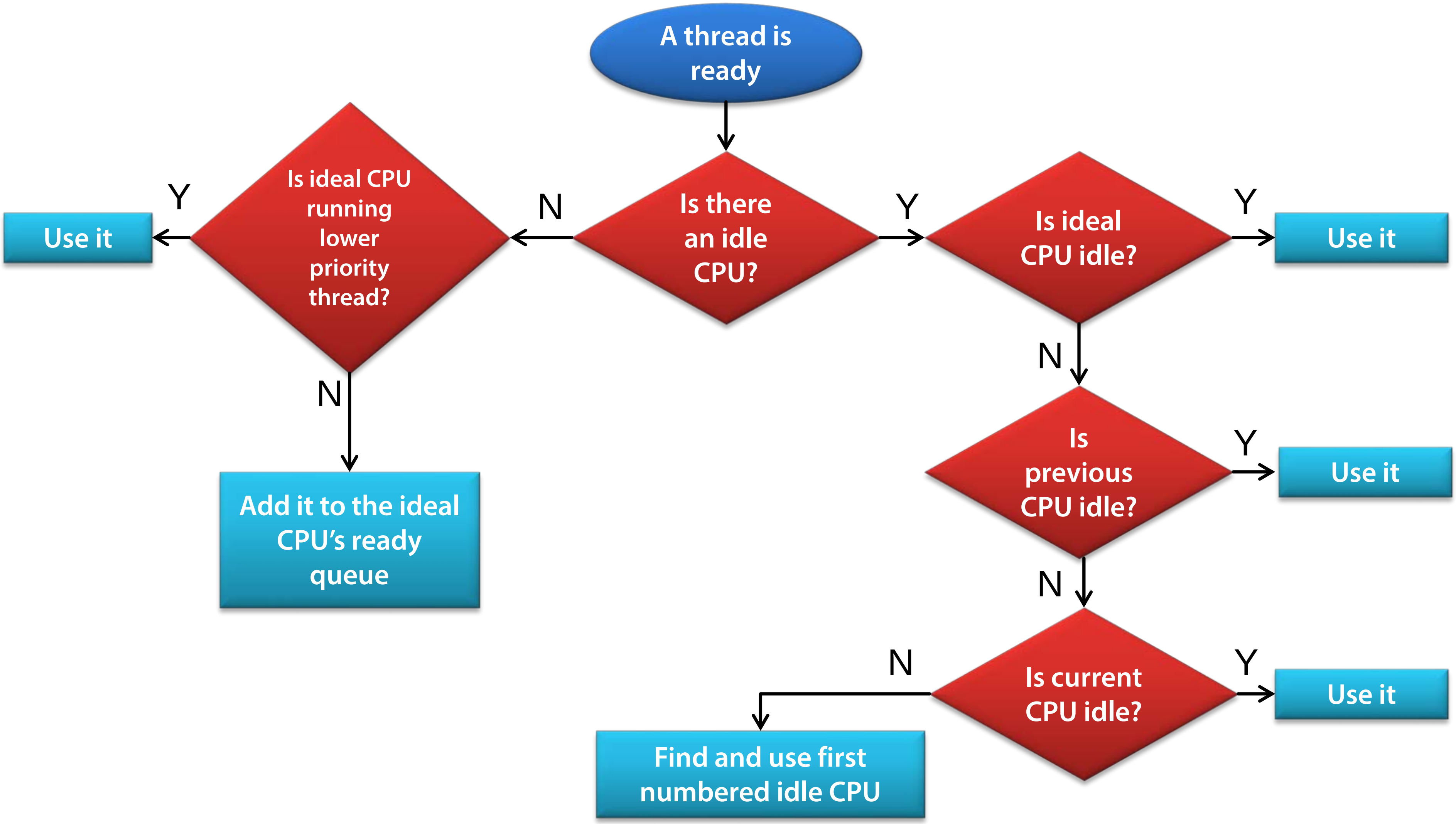
Multiprocessing - Hard Affinity

- **Threads can run on any CPU unless hard affinity is set for that thread**
  - **SetThreadAffinityMask**
  - The mask is a bit mask of allowed CPUs to run that thread
    - Default is process affinity mask, which defaults to all processors
  - Calling **SetProcessAffinityMask** changes priority mask for all threads running under that process
    - And future created threads in that process
- **Using hard affinity may result in threads getting less CPU time**

Multiprocessor Scheduling

- **Single CPU scheduling is relatively simple**
  - Use the highest priority thread
- **Multi CPU systems complicate things**
  - Windows attempt to balance priority needs with thread's preferred and previous CPUs
  - The only guarantee is that one of the highest priority threads is running on some CPU
- **NUMA (Non uniform memory architecture) complicate things further**

Scheduling on Multi-CPU System (Simplified)



Thread Synchronization

- **Threads sometimes need to coordinate work**
- **Canonical example**
  - Accessing a linked list concurrently from multiple threads
- **Synchronization is based upon waiting for some condition to occur**
- **The kernel provides a set of synchronization (dispatcher) primitives on which threads can wait efficiently**

Kernel Dispatcher Objects

- **Maintain a state (signaled or non-signaled)**
  - The meaning of “signaled” depends on the object type
- **Can be waited to change to the signaled state**
  - Windows API: **WaitForSingleObject**, **WaitForMultipleObjects** and their variants
  - Kernel mode: **KeWaitForSingleObject**, **KeWaitForMultipleObjects**
- **Dispatcher object types**
  - Process, thread, event, mutex, semaphore, timer, file, I/O completion port
- **Higher level wrappers exist**
  - MFC: **CSyncObject** (abstract base of **CMutex**, **CSemaphore** and others)
  - .NET: **WaitHandle** (abstract base of **Mutex**, **Semaphore** and others)

“Signaled” Meaning

- **Process**
  - The process has terminated
- **Thread**
  - The thread has terminated
- **Mutex**
  - The mutex is free
- **Event**
  - The event flag is raised
- **Semaphore**
  - The semaphore count is greater than zero
- **File, I/O completion port**
  - I/O operation completed
- **Timer**
  - Interval time expires

Mutex

- **Mutual exclusion**
- **Called Mutant in kernel terminology**
- **Allows a single thread to enter a critical region**
- **The thread that enters the critical region (its wait has succeeded) is the owner of the mutex**
- **Releasing the mutex allows one (single) thread to acquire it and enter the critical section**
- **Recursive acquisition is ok (increments a counter)**
  - If the owning thread does not release the mutex before it terminates, the kernel releases it and the next wait succeeds with a special code (abandoned mutex)

Semaphore

- **Maintains a counter (set at creation time)**
- **Allows x callers to "go through" a gate**
- **When a thread succeeds a wait, the semaphore counter decreases**
  - When the counter reaches zero, subsequent waits do not succeed (state is non-signaled)
  - Releasing the semaphore increments its counter, releasing a thread that is waiting
- **Is a Semaphore with a maximum count of one equivalent to a Mutex?**
- **Does not maintain any ownership**

Event

- **Maintains a Boolean flag**
- **Event types**
  - Manual reset (Notification in kernel terminology)
  - Auto reset (Synchronization)
- **When set (signaled) threads waiting for it succeed the wait**
  - Manual reset event releases any number of threads
  - Auto reset event releases just one thread
    - And the event goes automatically to the non-signaled state
- **Useful when no other object fits the bill**
  - Provides flow synchronization as opposed to data synchronization

Critical Section

- **User mode replacement for a mutex**
- **Can be used to synchronize threads within a single process**
  - Operates on a structure of type **CRITICAL_SECTION**
- **Cheaper than a mutex when no contention exists**
  - No transition to kernel mode in this case
- **Uses EnterCriticalSection and LeaveCriticalSection API functions**
  - No way to specify a timeout other than infinite and zero
    - Zero is accomplished with **TryEnterCriticalSection**
- **.NET**
  - A similar effect is achieved with the **lock** C# keyword
  - Calls the framework's **Monitor.Enter/Exit** in a **try/finally** block

Demo

Thread Synchronization

More threading

- **Thread pools**
  - Simplifies thread management
  - Potentially boosts performance as threads don't need to be created/destroyed explicitly
- **C++11 and .NET 4+ provide helpers for fork/join scenarios**
  - **parallel_for** (C++), **Parallel.For** (.NET)
  - Simplify operations where order is unimportant
- **Other higher level threading helpers exist in C++ 11 and .NET 4+**
  - Manual thread management considered "low level"
  - Understanding threads can help make the right choices and solve problems

Demo

Automatic parallelization

Jobs

- **Kernel object that allows managing one (or more) processes as a unit**
- **System enforces Job quotas and security**
  - Total and per process CPU time, working sets, CPU affinity and priority class, quantum length (for long, fixed quantums only)
  - Security limits
  - UI limits
- **API**
  - **CreateJobObject** / **OpenJobObject**
  - **AssignProcessToJobObject**
  - **TerminateJobObject**
  - **SetInformationJobObject**

Demo

Jobs

Summary

- Process is a management object
- Threads are the real workers
- Windows schedules threads to run on processors
- Understanding the way threads and processes work helps in design, debugging and troubleshooting