Process/Thread structure

- Any Handle Table
  - Object Manager
    - Process Object
      - Thread
      - Thread
      - Thread
      - Thread
      - Thread

- Process' Handle Table
- Virtual Address Descriptors
- Files
- Events
- Devices
- Drivers
Process

Container for an address space and threads
Associated User-mode Process Environment Block (PEB)
Primary Access Token
Quota, Debug port, Handle Table etc
Unique process ID
Queued to the Job, global process list and Session list
MM structures like the WorkingSet, VAD tree, AWE etc
Thread

Fundamental schedulable entity in the system
Represented by ETHREAD that includes a KTHREAD
Queued to the process (both E and K thread)
IRP list
Impersonation Access Token
Unique thread ID
Associated User-mode Thread Environment Block (TEB)
User-mode stack
Kernel-mode stack
Processor Control Block (in KTHREAD) for cpu state when not running
CPU Control-flow

Thread scheduling occurs at PASSIVE or APC level (IRQL < 2)
APCs (Asynchronous Procedure Calls) deliver I/O completions, thread/process termination, etc (IRQL == 1)
   Not a general mechanism like unix signals (user-mode code must explicitly block pending APC delivery)
Interrupt Service Routines run at IRL > 2
ISRs defer most processing to run at IRQL==2 (DISPATCH level) by queuing a DPC to their current processor
A pool of worker threads available for kernel components to run in a normal thread context when user-mode thread is unavailable or inappropriate
Normal thread scheduling is round-robin among priority levels, with priority adjustments (except for fixed priority real-time threads)
Asynchronous Procedure Calls

APCs execute routine in thread context
not as general as UNIX signals
user-mode APCs run when blocked & alertable
kernel-mode APCs used extensively: timers,
notifications, swapping stacks, debugging, set
thread ctx, I/O completion, error reporting,
creating & destroying processes & threads, …

APCs generally blocked in critical sections
e.g. don’t want thread to exit holding resources
Deferred Procedure Calls

DPCs run a routine on a particular processor
DPCs are higher priority than threads
common usage is deferred interrupt processing
ISR queues DPC to do bulk of work

- *long DPCs harm perf, by blocking threads*
- *Drivers must be careful to flush DPCs before unloading*
also used by scheduler & timers (e.g. at quantum end)
kernel-mode APCs used extensively: timers, notifications, swapping stacks, debugging, set thread ctx, I/O completion, error reporting, creating & destroying processes & threads, ...

High-priority routines use IPI (inter-processor intr)
used by MM to flush TLB in other processors
System Threads

System threads have no user-mode context
  Run in ‘system’ context, use system handle table

System thread examples
  Dedicated threads
    Lazy writer, modified page writer, balance set manager,
    mapped pager writer, other housekeeping functions
  General worker threads
    Used to move work out of context of user thread
    Must be freed before drivers unload
    Sometimes used to avoid kernel stack overflows
  Driver worker threads
    Extends pool of worker threads for heavy hitters, like file server
Scheduling

Windows schedules threads, not processes
- Scheduling is preemptive, priority-based, and round-robin at the highest-priority
- 16 real-time priorities above 16 normal priorities
- Scheduler tries to keep a thread on its ideal processor/node to avoid perf degradation of cache/NUMA-memory
  - Threads can specify affinity mask to run only on certain processors

Each thread has a current & base priority
- Base priority initialized from process
- Non-realtime threads have priority boost/decay from base
  - Boosts for GUI foreground, waking for event
  - Priority decays, particularly if thread is CPU bound (running at quantum end)

Scheduler is state-driven by timer, setting thread priority, thread block/exit, etc

Priority inversions can lead to starvation
  - balance manager periodically boosts non-running runnable threads
Thread scheduling states

Transition
k stack
swapped

KiInsertDeferredRead

KiReadyThread

Ready process

KeInitThread
Thread scheduling states

• Main quasi-states:
  – Ready – able to run
  – Running – current thread on a processor
  – Waiting – waiting an event

• For scalability Ready is three real states:
  – DeferredReady – queued on any processor
  – Standby – will be imminently start Running
  – Ready – queue on target processor by priority

• Goal is granular locking of thread priority queues

• Red states related to swapped stacks and processes
KPRCB Fields

Per-processor ready summary and ready queues
• WaitListHead[F/B]
• ReadySummary
• SelectNextLast
• DispatcherReadyListHeads[F/B][MAXIMUM_PRIORITY]
• pDeferredReadyListHead

Processor information
•_VENDOR_STRING[], InitialApicId, Hyperthreading, MHz, FeatureBits, CpuType, CpuId, CpuStep
• ProcessorNumber, Affinity SetMember
• ProcessorState, PowerState
KPRCB Fields - cont.

Miscellaneous counters
- InterruptCount, KernelTime, UserTime, DpcTime, DebugDpcTime, InterruptTime, Cc*Read*, KeExceptionDispatchCount, KeFloatingEmulationCount, KeSecondLevelTbFills, KeSystemCalls, ...

Per-processor pool lists and QueueLocks
- PP*LookasideList[], LockQueue[]

IPI and DPC related fields
- CurrentPacket, TargetSet, IPIWorkerRoutine, RequestSummary, SignalDone, …
- DpcData[], pDpcStack, DpcRoutineActive, ProcsGenericDPC, …
KTHREAD

Scheduling-related fields
volatile UCHAR State;
volatile UCHAR DeferredProcessor;
SINGLE_LIST_ENTRY SwapListEntry;
LIST_ENTRY WaitListEntry;
SCHAR Priority;
BOOLEAN Preempted;
ULONG WaitTime;
volatile UCHAR SwapBusy;
KSPIN_LOCK ThreadLock;

APC-related fields
KAPC_STATE ApcState;
PKAPC_STATE ApcStatePointer[2];
KAPC_STATE SavedApcState;
KSPIN_LOCK ApcQueueLock;

© Microsoft Corporation
Thread scheduling states (yet again)

Transition

KilnitThread

KilInsertDeferredRead

KiReadyThread

Ready process

© Microsoft Corporation
```
enum _KTHREAD_STATE

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ready</td>
<td>Queued on Prcb-DispatcherReadyListHead</td>
</tr>
<tr>
<td>Running</td>
<td>Pointed at by Prcb-&gt;CurrentThread</td>
</tr>
<tr>
<td>Standby</td>
<td>Pointed at by Prcb-&gt;NextThread</td>
</tr>
<tr>
<td>Terminated</td>
<td></td>
</tr>
<tr>
<td>Waiting</td>
<td>Queued on WaitList-&gt;WaitBlock</td>
</tr>
<tr>
<td>Transition</td>
<td>Queued on KiStackInSwapList</td>
</tr>
<tr>
<td>Deferred</td>
<td>Pointed at by Prcb-&gt;DeferredReadyListHead</td>
</tr>
<tr>
<td>Initialized</td>
<td></td>
</tr>
</tbody>
</table>
```

© Microsoft Corporation
Where states are set

<table>
<thead>
<tr>
<th>State</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ready</td>
<td>Thread wakes up</td>
</tr>
<tr>
<td>Running</td>
<td>KeInitThread, KIdleSchedule, KiSwapThread, KiExitDispatcher, NtYieldExecution</td>
</tr>
<tr>
<td>Standby</td>
<td>The thread selected to run next</td>
</tr>
<tr>
<td>Terminated</td>
<td>Set by KeTerminateThread()</td>
</tr>
<tr>
<td>Waiting</td>
<td></td>
</tr>
<tr>
<td>Transition</td>
<td>Awaiting inswap by KiReadyThread()</td>
</tr>
<tr>
<td>Deferred…</td>
<td></td>
</tr>
<tr>
<td>Initialized</td>
<td>Set by KeInitThread()</td>
</tr>
</tbody>
</table>
Idle processor preferences

(a) Select the thread's ideal processor – if idle, otherwise consider the set of all processors in the thread’s hard affinity set
(b) If the thread has a preferred affinity set with an idle processor, consider only those processors
(c) If hyperthreaded and any physical processors in the set are completely idle, consider only those processors
(d) if this thread last ran on a member of this remaining set, select that processor, otherwise,
(e) if there are processors amongst the remainder which are not sleeping, reduce to that subset.
(f) select the leftmost processor from this set.
KiInsertDeferredReadyList ()

Prcb = KeGetCurrentPrcb();
Thread->State = DeferredReady;
Thread->DeferredProcessor = Prcb->Number;
PushEntryList(&Prcb->DeferredReadyListHead, &Thread->SwapListEntry);
KiDeferredReadyThread()

// assign to idle processor or preempt a lower-pri thread
if boost requested, adjust pri under threadlock
if there are idle processors, pick processor
    acquire PRCB locks for us and target processor
    set thread as Standby on target processor
    request dispatch interrupt of target processor
    release both PRCB locks
return
KiDeferredReadyThread() - cont

target is the ideal processor
acquire PRCB locks for us and target
if (victim = target->NextThread)
    if (thread->Priority <= victim->Priority)
        insert thread on Ready list of target processor
        release both PRCB locks and return
        victim->Preempted = TRUE
set thread as Standby on target processor
set victim as DeferredReady on our processor
release both PRCB locks
target will pickup thread instead of victim
return
KiDeferredReadyThread() – cont2

victim = target->CurrentThread
acquire PRCB locks for us and target
if (thread->Priority <= victim->Priority)
    insert thread on Ready list of target processor
    release both PRCB locks and return
victim->Preempted = TRUE
set thread as Standby on target processor
release both PRCB locks
request dispatch interrupt of target processor
return
KiInSwapProcesses()

// Called from only:
    KeSwapProcessOrStack  [System Thread]

For every process in swap-in list
    Sets ProcessInSwap
    Calls MmInSwapProcess
    Sets ProcessInMemory
KiQuantumEnd()

// Called at dispatch level
Raise to SYNCH level, acquire ThreadLock, PRCB Lock
if thread->Quantum <= 0
    thread->Quantum = Process->ThreadQuantum
pri = thread->Priority = KiComputeNewPriority(thread)
if (Prcb->NextThread == NULL)
    newThread = KiSelectReadyThread (pri, Prcb)
if (newThread)
    newThread->State = Standby
    Prcb->NextThread = newThread
else thread->Preempted = FALSE
release the ThreadLock
if (! Prcb->NextThread) release PrcbLock, return
thread->SwapBusy = TRUE
newThread = Prcb->NextThread
Prcb->NextThread = NULL
Prcb->CurrentThread = newThread
newThread->State = Running
thread->WaitReason = WrQuantumEnd
KxQueueReadyThread(thread, Prcb)
thread->WaitIrql = APC_LEVEL
KiSwapContext(thread, newThread)
KxQueueReadyThread(Thread, Prcb)

if ((Thread->Affinity & Prcb->SetMember) != 0)
    Thread->State = Ready
pri = Thread->Priority
Preempted = Thread->Preempted;
Thread->Preempted = 0
Thread->WaitTime = KiQueryLowTickCount()
insertfcn = Preempted? InsertHeadList : InsertTailList
Insertfcn(&Prcb->ReadyList [PRI],
        &Thread->WaitListEntry)
Prpcb->ReadySummary |= PRIORITY_MASK(PRI)
KiReleasePrcbLock(Prpcb)
KxQueueReadyThread … cont.

else
    Thread->State = DeferredReady
    Thread->DeferredProcessor = Prcb->Number
    KiReleasePrcbLock(Prcb)
    KiDeferredReadyThread(Thread)
KiExitDispatcher(oldIrql)

// Called at SYNCH_LEVEL
Prcb = KeGetCurrentPrcb()
if (Prcb->DeferredReadyListHead.Next)
    KiProcessDeferredReadyList(Prcb)
if (oldIrql >= DISPATCH_LEVEL)
    if (Prcb->NextThread && !Prcb->DpcRoutineActive)
        KiRequestSoftwareInterrupt(DISPATCH_LEVEL)
        KeLowerIrql(oldIrql)
    return
// oldIrql < DISPATCH_LEVEL
KiAcquirePrcbLock(Prcb)
KiExitDispatcher(oldIrql) – cont.

NewThread = Prcb->NextThread
CurrentThread = Prcb->CurrentThread
thread->SwapBusy = TRUE
Prcb->NextThread = NULL
Prcb->CurrentThread = NewThread
NewThread->State = Running
KxQueueReadyThread(CurrentThread, Prcb)
CurrentThread->WaitIrql = OldIrql
Pending = KiSwapContext(CurrentThread, NewThread)
if (Pending != FALSE)
    KeLowerIrql(APC_LEVEL);
    KiDeliverApc(KernelMode, NULL, NULL);
Kernel Thread Attach

Allows a thread in the kernel to temporarily move to a different process’ address space

• Used heavily in VM system
• Used by object manager for kernel handles
• PspProcessDelete attaches before calling ObKillProcess() so close/delete in process context
• Used to query a process’ VM counters
KiAttachProcess (Thread, Process, APCLock, SavedApcState)

Process->StackCount++
KiMoveApcState(&Thread->ApcState, SavedApcState)
Re-initialize Thread->ApcState
if (SavedApcState == &Thread->SavedApcState)
    Thread->ApcStatePointer[0] = &Thread->SavedApcState
    Thread->ApcStatePointer[1] = &Thread->ApcState
    Thread->ApcStateIndex = 1

// assume ProcessInMemory case and empty ReadyList
Thread->ApcState.Process = Process
KiUnlockDispatcherDatabaseFromSynchLevel()
KeReleaseInStackQueuedSpinLockFromDpcLevel(APCLock)
KiSwapProcess(Process, SavedApcState->Process)
KiExitDispatcher(LockHandle->OldIrql)
Asynchronous Procedure Calls

APCs execute code in context of a particular thread
APCs run only at PASSIVE or APC LEVEL (0 or 1)

Three kinds of APCs

**User-mode:** deliver notifications, such as I/O done
**Kernel-mode:** perform O/S work in context of a process/thread, such as completing IRPs
**Special kernel-mode:** used for process termination

Multiple ‘environments’:

**Original:** The normal process for the thread (ApcState[0])
**Attached:** The thread as attached (ApcState[1])
**Current:** The ApcState[ ] as specified by the thread
**Insert:** The ApcState[ ] as specified by the KAPC block
**KAPC**

<table>
<thead>
<tr>
<th>pThread</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApcListEntry[2]</td>
</tr>
<tr>
<td>pKernelRoutine</td>
</tr>
<tr>
<td>pRundownRoutine</td>
</tr>
<tr>
<td>pNormalRoutine</td>
</tr>
<tr>
<td>pNormalContext</td>
</tr>
<tr>
<td>SystemArguments[2]</td>
</tr>
<tr>
<td>Inserted</td>
</tr>
</tbody>
</table>
KeInitializeApc()

// assume CurrentApcEnvironment case
Apc->ApcStateIndex = Thread->ApcStateIndex
Apc->Thread = Thread;
Apc->KernelRoutine = KernelRoutine
Apc->RundownRoutine = RundownRoutine  // optional
Apc->NormalRoutine = NormalRoutine    // optional
if NormalRoutine
    Apc->ApcMode = ApcMode            // user or kernel
    Apc->NormalContext = NormalContext
else // Special kernel APC
    Apc->ApcMode = KernelMode
    Apc->NormalContext = NIL
Apc->Inserted = FALSE
KiInsertQueueApc()

Insert the APC object in the APC queue for specified mode

- **Special APCs (! Normal)** – insert after other specials
- **User APC && KernelRoutine is PsExitSpecialApc()** – set UserAPCPending and insert at front of queue
- **Other APCs** – insert at back of queue

For kernel-mode APC

- if thread is Running: KiRequestApcInterrupt(processor)
- if Waiting at PASSIVE &&
  (special APC && !Thread->SpecialAPCDisable ||
  kernel APC && !Thread->KernelAPCDisable) call KiUnwaitThread(thread)

If user-mode APC && threads in alertable user-mode wait
set UserAPCPending and call KiUnwaitThread(thread)
KiDeliverApc()

Called at APC level from the APC interrupt code and at system exit (when either APC pending flag is set)

All special kernel APC's are delivered first

Then normal kernel APC's (unless one in progress)

Finally

if the user APC queue is not empty

&& Thread->UserAPCPending is set

&& previous mode is user

Then a user APC is delivered
Scheduling Summary

Scheduler lock broken up per-processor
  Achieves high-scalability for otherwise hot lock
Scheduling is preemptive by higher priority threads, but otherwise round-robin
Boosting is used for non-realtime threads
Threads are swapped out by balance set manager to reclaim memory (stack)
Balance Set Manager manages residence, drives working set trims, and fixes deadlocks
Discussion