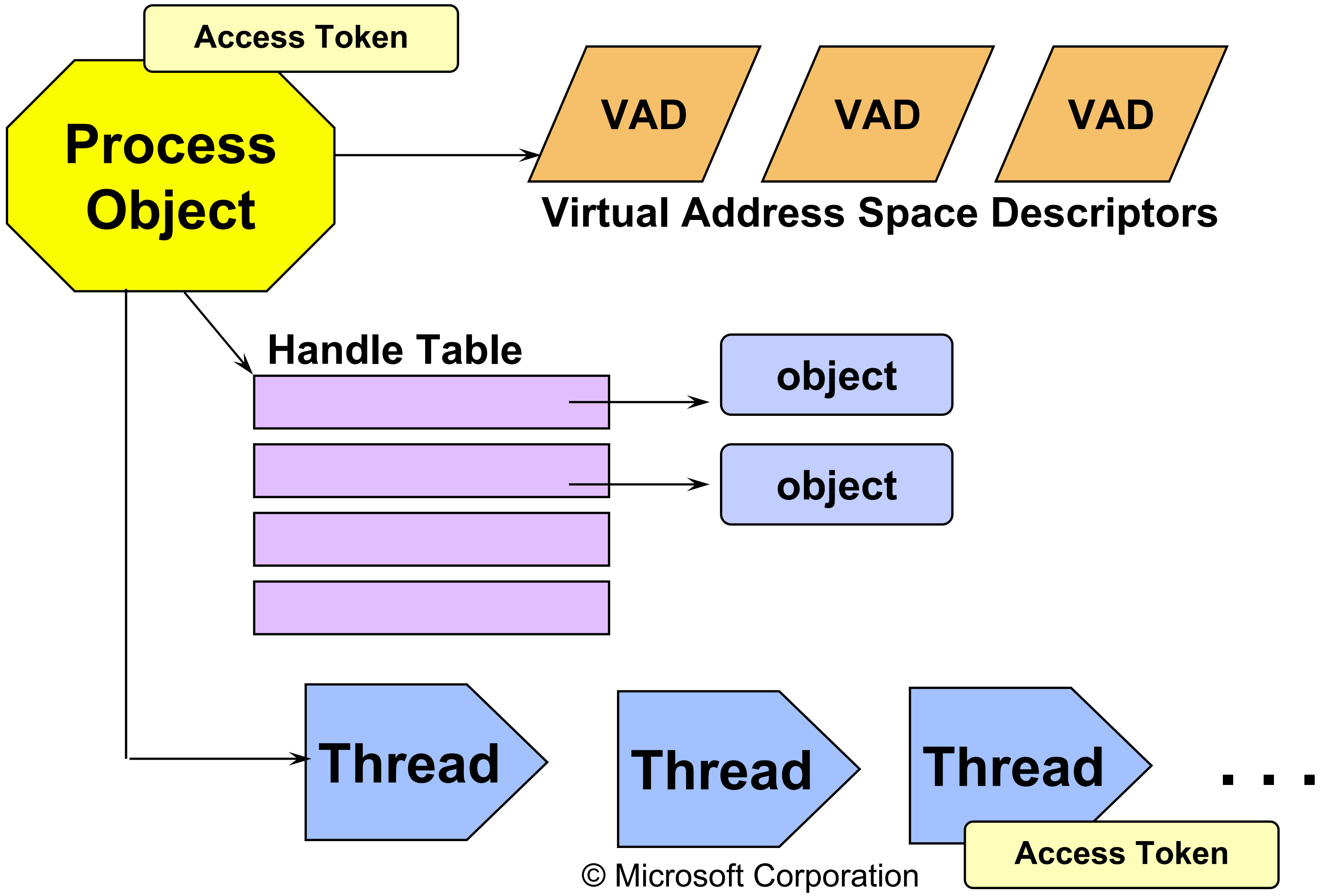
Windows Kernel Internals
Thread Scheduling

David B. Probert, Ph.D.

Windows Kernel Development

Microsoft Corporation

Processes & Threads



Each process has its own…

- Virtual address space (including program global storage, heap storage, threads' stacks)
  - processes cannot corrupt each other's address space by mistake
- Working set (physical memory "owned" by the process)
- Access token (includes security identifiers)
- Handle table for Win32 kernel objects
- These are common to all threads in the process, but separate and protected between processes

Each thread has its own…

- Stack (automatic storage, call frames, etc.)
- Instance of a top-level function
- Scheduling state (Wait, Ready, Running, etc.) and priority
- Current access mode (user mode or kernel mode)
- Saved CPU state if it isn't Running
- Access token (optional -- overrides process's if present)

Deferred Procedure Calls (DPCs)

- Used to defer processing from higher (device) interrupt level to a lower (dispatch) level
  - Driver queues request (one queue per CPU, but any CPU can drain each other’s DPC queue)
  - Executes specified procedure at dispatch IRQL (or “dispatch level”, also “DPC level”) when all higher-IRQL work (interrupts) completed
- Used heavily for driver “after interrupt” functions
  - Also used for quantum end and timer expiration

System Threads

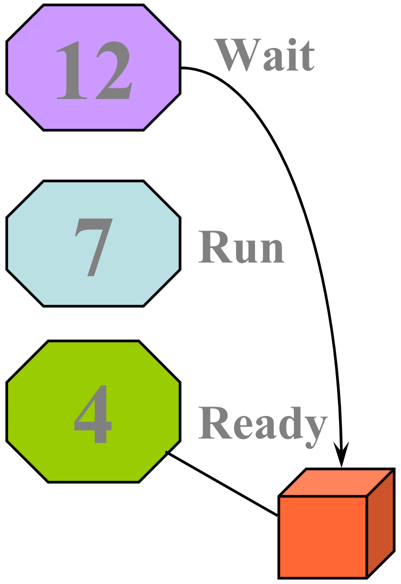
- Subroutines in OS and some drivers that need to run as real threads
  - E.g., need to run concurrently with other system activity, wait on timers, perform background “housekeeping” work
  - Always run in kernel mode
- Examples
  - Modified Page writer
  - Redirector and Server worker threads
  - General executive level worker threads

Balance Set Manager

- Balance Set Manager is a system thread
  - Wakes up every second. If paging activity high or memory needed:
    - Makes the kernel stack of threads that have been waiting for a certain amount of time, nonresident.
    - Removes processes from the balance set when memory gets tight and brings processes back into the balance set when there is more memory available.
    - Makes the kernel stack resident for threads whose wait has been completed, but whose stack is nonresident.
    - Adjusts depth of lookaside lists.
    - Invokes the MmWorkingSetManager() to trim workingsets
- Balance Set Manager also fights priority inversion deadlocks by boosting ready threads that haven't run in awhile

CPU Starvation Avoidance

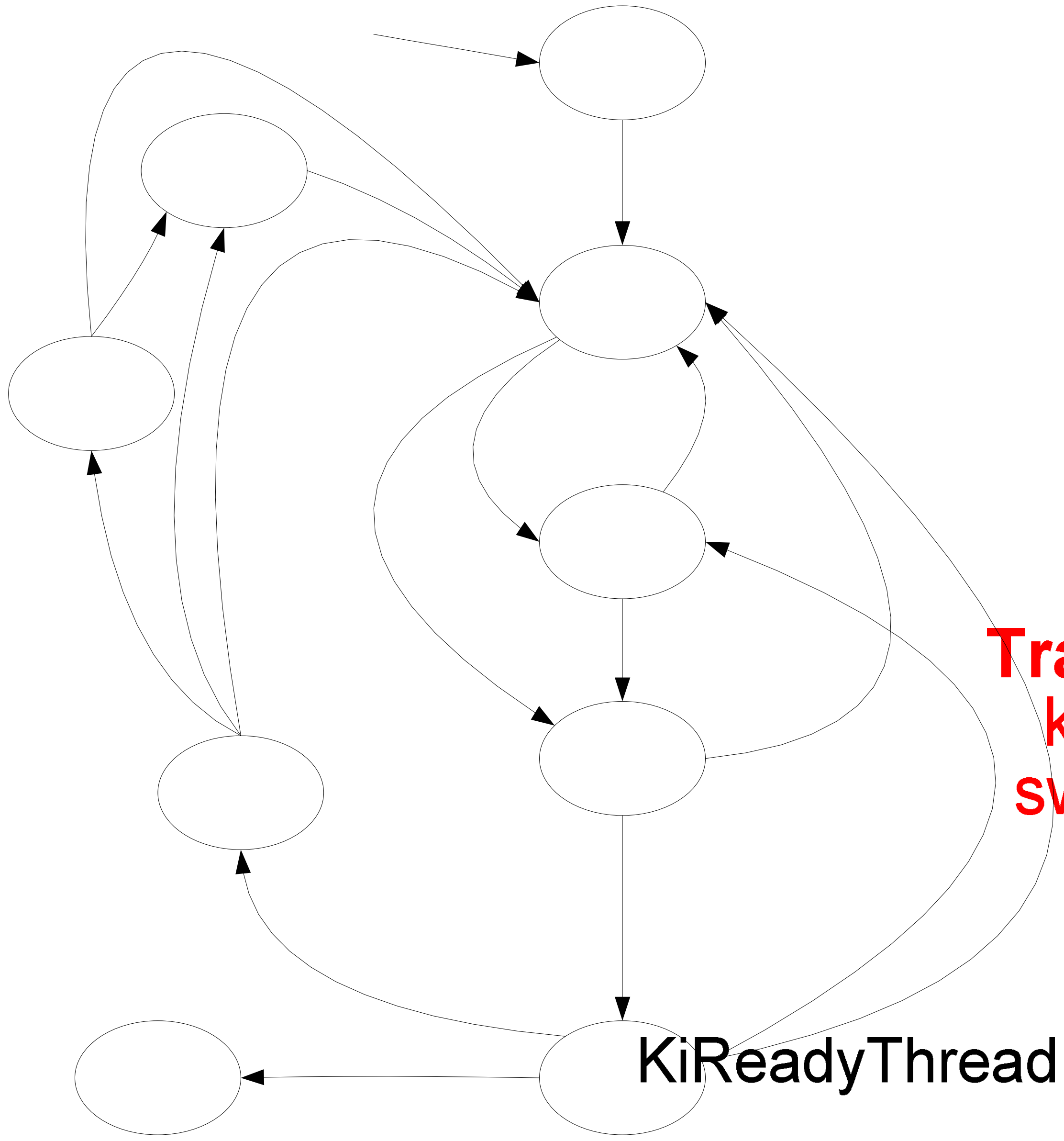
- Balance Set Manager looks for "starved" threads
  - Wakes up once per second and examines Ready queues
  - Looks for threads that have been Ready for n seconds or more
- Attempts to resolve "priority inversions"
  - E.g. a high priority thread (12 in diagram) waits on something locked by a lower thread (4), which can't run because of a middle priority CPU-bound thread (7)), but not deterministically (no priority inheritance)
- Special boost to priority 15, and quantum is doubled
  - 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)



Scheduling

- Threads are scheduled, not processes
- Strictly priority driven, preemptive
  - A FIFO queue of “ready” threads for each priority level
  - When a thread becomes Ready, it either runs immediately or is inserted at the tail of the Ready queue for its current (dynamic) priority
  - Highest priority Ready thread(s) always run
  - Event-driven; no guaranteed execution period before preemption
    - No formal scheduler loop
    - Quantum end event
    - Wait completion event
    - Manual/Voluntary wait
- Time-sliced, round-robin within a priority level
- Simultaneous thread execution on MP systems

Thread scheduling states



KeInitThre

Transition
k stack swapped

KiInsertDeferred

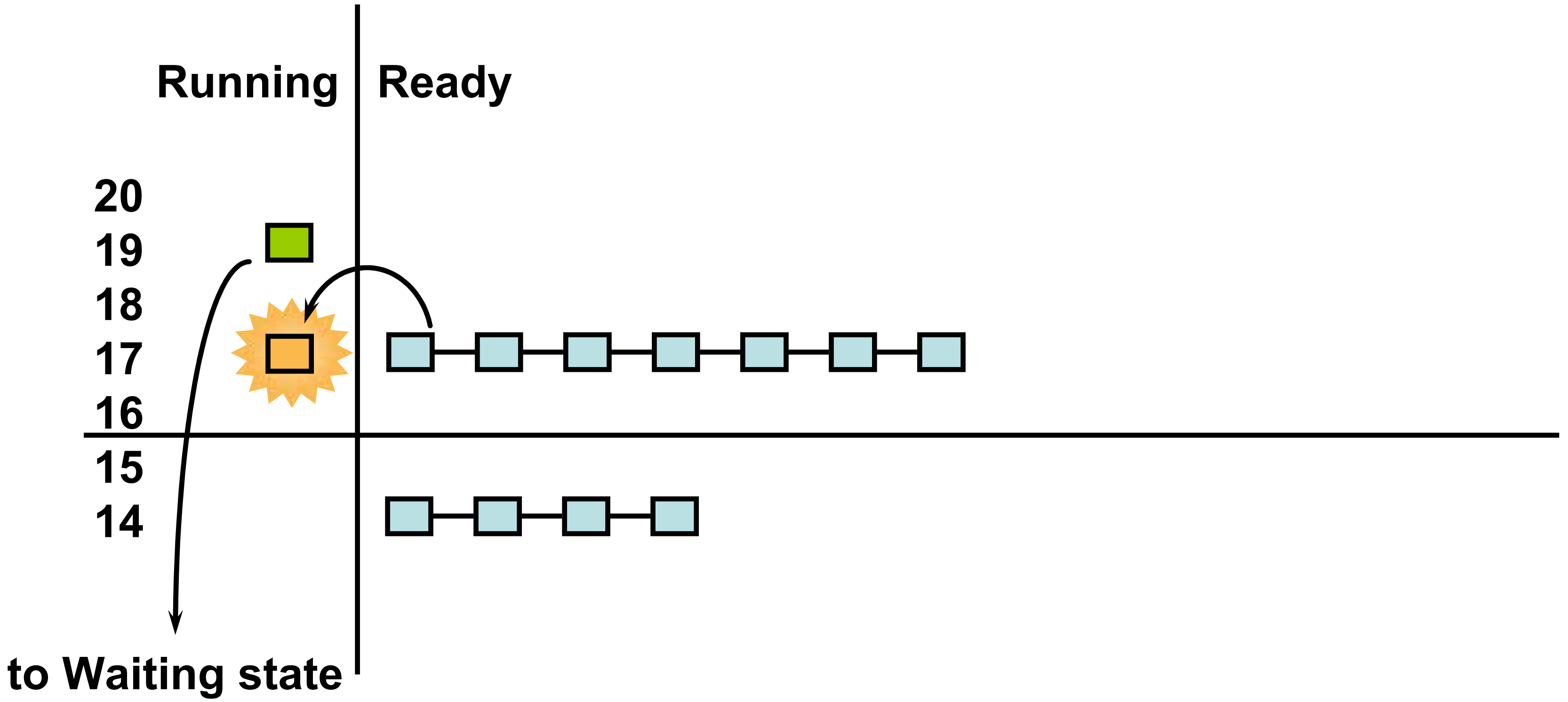
Ready

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

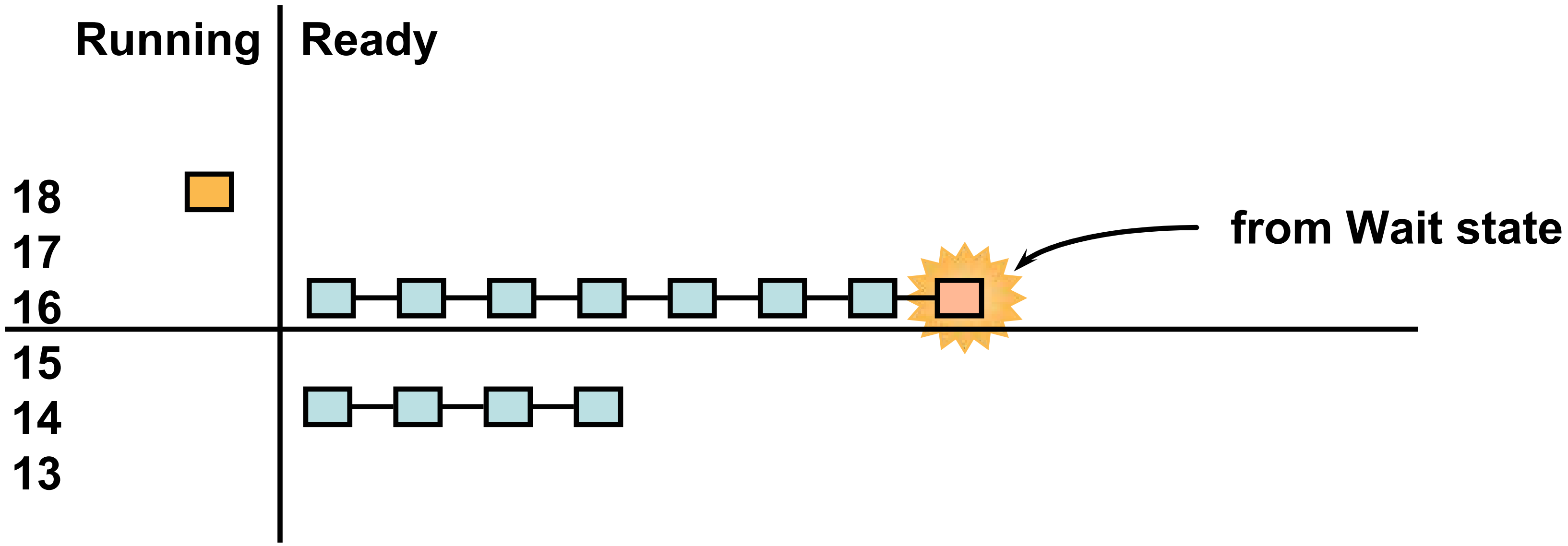
Scheduling Scenarios: Voluntary Switch

- The running thread gives up the CPU
- Schedule the thread at the head of the next non-empty "ready" queue



Scheduling Scenarios: Ready after Wait Resolution

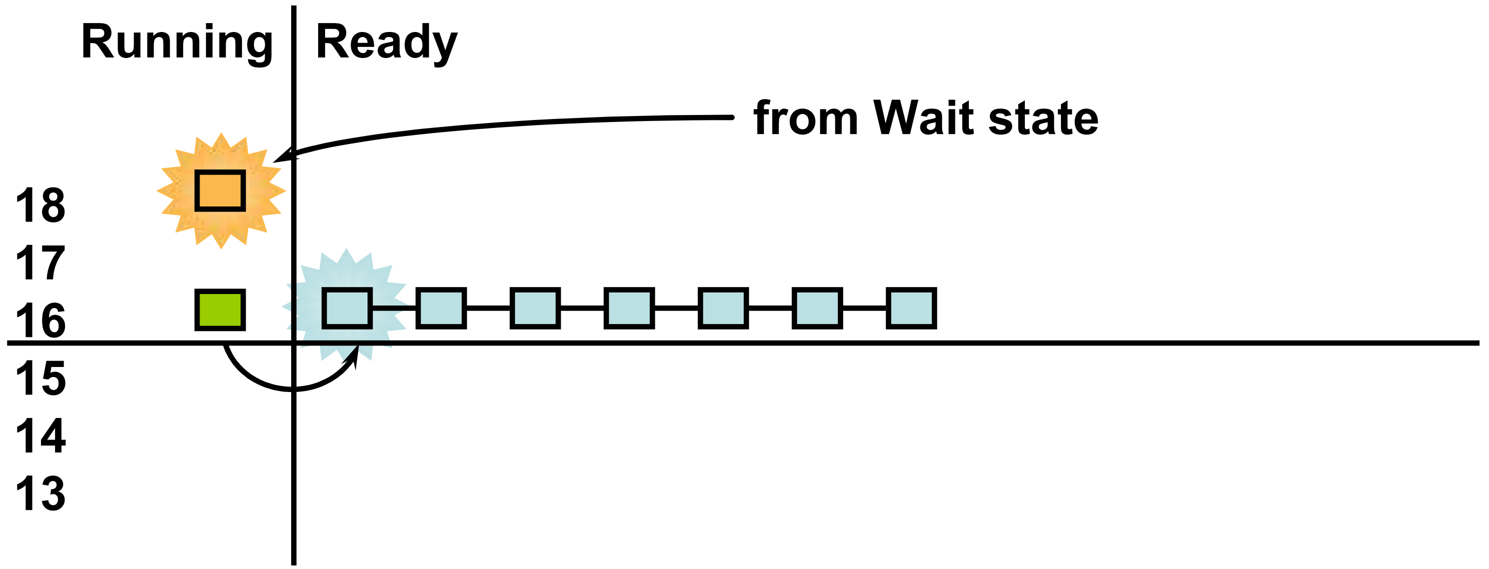
- If newly-ready thread is not of higher priority than the running thread…
- …it is put at the tail of the ready queue for its current priority



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 IRQL to >= 2)

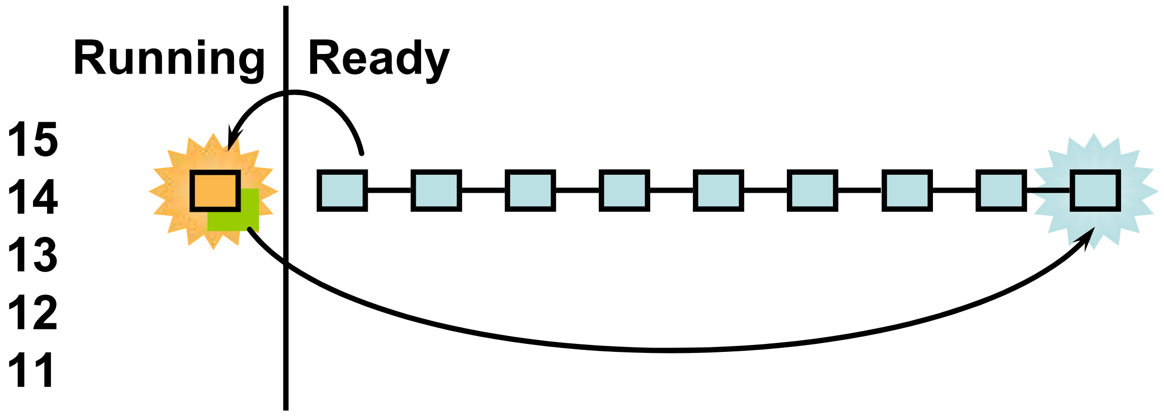


A preempted thread goes back to the head of its ready queue

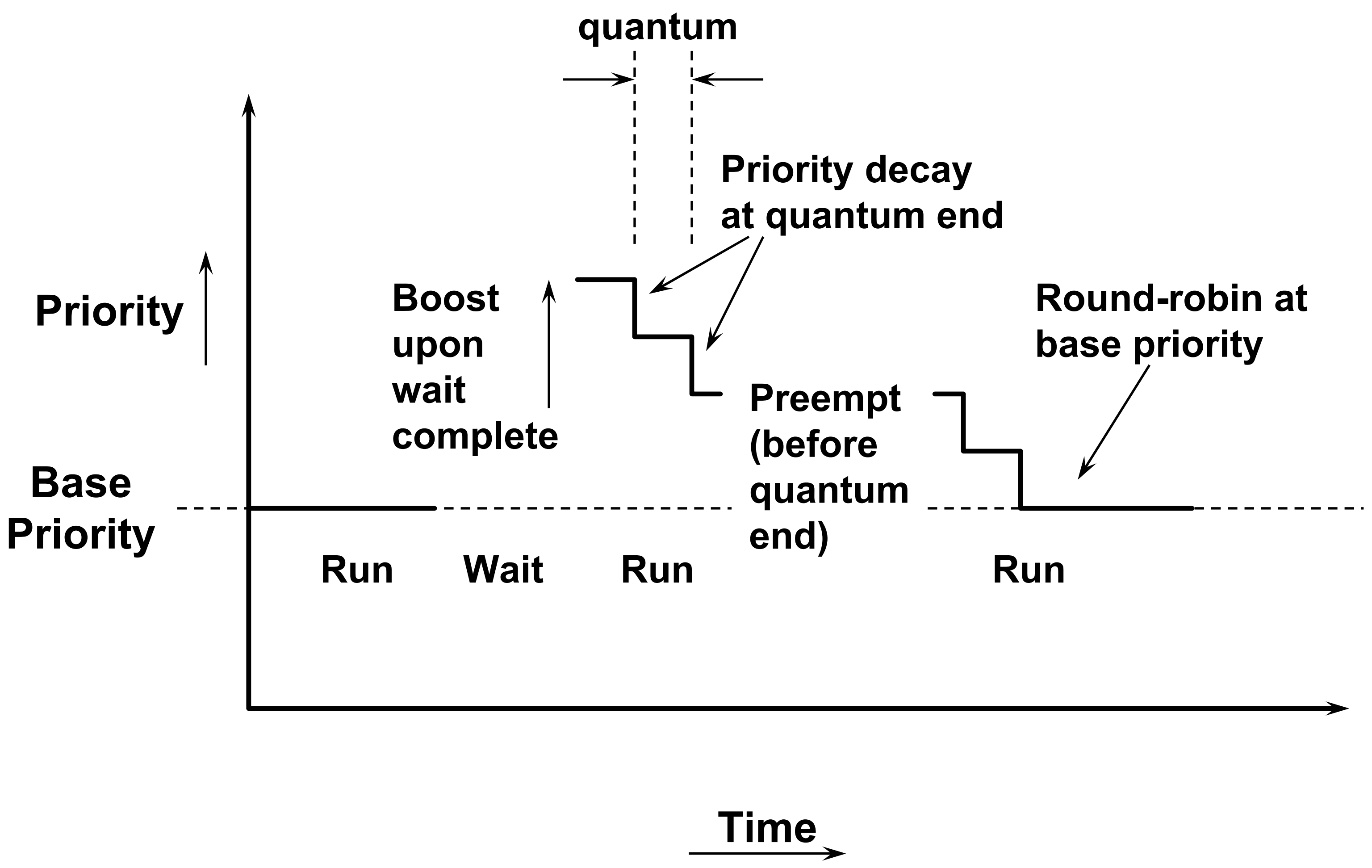
- also, if in real-time priority range, its quantum is reset (t.b.d.)

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
  - standard quantum is two clock ticks (12 on NT Server)
    - standard clock tick is 10 msec; might be 7.5 (Alpha) or 15 (MP Pentium) msec
  - 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)



Thread Priority Boost and Decay


quantum
Priority decay at quantum end
Priority
Boost upon wait complete
Round-robin at base priority
Preempt (before quantum end)
Base Priority
Run
Wait
Run
Run
Time

Processor Affinity

- Every thread has an “ideal processor”
  - Always TRY to run a thread on it’s home processor
  - default value set round-robin within each process
  - Settable via SetThreadIDealProcessor
  - Whistler extends this to a set of ideal processors which affects scheduling, and memory
- Hard Affinity can be used to restrict a thread/process to a set of processors
  - Only processors enabled for that thread are able to run the thread
  - Settable via SetThreadAffinityMask and SetProcessAffinityMask
  - TaskManager and JobControlApplet

KPRCB Fields

Per-processor ready summary and ready queues

- WaitListHead[F/B]
- ReadySummary
- SelectNextLast
- DispatcherReadyListHeads[F/B][MAXIMUM_PRIORITY]
- pDeferredReadyListHead

Processor information

- VendorString[], 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

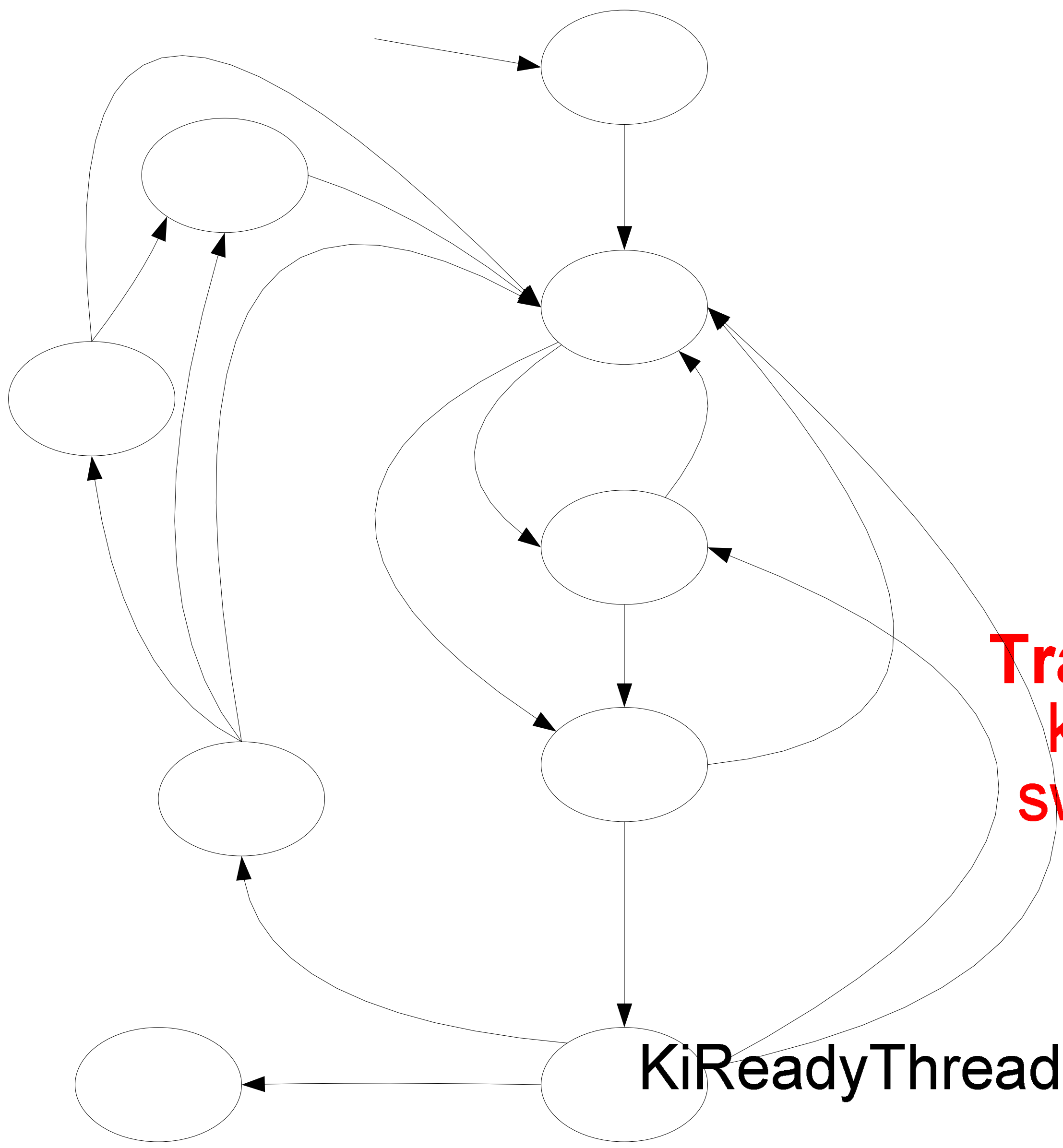
Scheduling-related fields

```
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;
```

Thread scheduling states (yet again)



enum _KTHREAD_STATE

Ready	Queued on Prcb->DispatcherReadyListHead
Running	Pointed at by Prcb->CurrentThread
Standby	Pointed at by Prcb->NextThread
Terminated	
Waiting	Queued on WaitList->WaitBlock
Transition	Queued on KiStackInSwapList
Deferred Ready	Pointed at by Prcb->DeferredReadyListHead
Initialized	

Where states are set

Ready	Thread wakes up
Running	KeInitThread, KiIdleSchedule, KiSwapThread, KiExitDispatcher, NtYieldExecution
Standby	The thread selected to run next
Terminated	Set by KeTerminateThread()
Waiting	
Transition	Awaiting inswap by KiReadyThread()
Deferred…	
Initialized	Set by KeInitThread()

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
```

KiQuantumEnd() – cont.

```
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)
    Prcb->ReadySummary |= PRIORITY_MASK(PRI)
    KiReleasePrcbLock(Prcb)
```

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

pThread			
ApcListEntry[2]			
pKernelRoutine			
pRundownRoutine			
pNormalRoutine			
pNormalContext			
SystemArguments[2]			
	Inserted	Mode	ApcStateIdx

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 workingset trims, and fixes deadlocks

Discussion