Process

- Container for an address space and threads
- Primary Token
- Quota, Debug port, Handle Table etc
- Unique process ID
- Queued to the Job, global process list and Session list
- MM structures like the VAD tree, AWE etc
Thread

- Fundamental schedulable entity in the system
- Structure is the ETHREAD that holds a KTHREAD
- Queued to the process (both E and K thread)
- IRP list
- Impersonation information
- Unique thread ID
- Flags or various sorts and TEB pointer
Job

- Container for multiple processes
- Queued to global job list, processes and jobs in the job set
- Security token filters and job token
- Completion ports
- Counters, limits etc
Processes & Threads

Process Object

Handle Table

Thread

Virtual Address Space Descriptors

VAD

object

Access Token

Thread

Thread

Thread

© Microsoft Corporation
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)
KPROCESS fields

DISPATCHER_HEADER Header
ULPTR DirectoryTableBase[2]
KGDTENTRY LdtDescriptor
KIDTENTRY Int21Descriptor
USHORT lopmOffset
UCHAR iopl
volatile KAFFINITY ActiveProcessors
ULONG KernelTime
ULONG UserTime
LIST_ENTRY ReadyListHead
SINGLE_LIST_ENTRY SwapListEntry
LIST_ENTRY ThreadListHead
KSPIN_LOCK ProcessLock

KAFFINITY Affinity
USHORT StackCount
SCHAR BasePriority
SCHAR ThreadQuantum
BOOLEAN AutoAlignment
UCHAR State
BOOLEAN DisableBoost
UCHAR PowerState
BOOLEAN DisableQuantum
UCHAR IdealNode
EPROCESS fields

KPROCESS Pcb
EX_PUSH_LOCK ProcessLock
LARGE_INTEGER CreateTime
LARGE_INTEGER ExitTime
EX_RUNDOWN_REF RundownProtect
HANDLE UniqueProcessId
LIST_ENTRY ActiveProcessLinks
Quota Fields
SIZE_T PeakVirtualSize
SIZE_T VirtualSize
LIST_ENTRY SessionProcessLinks
PVOID DebugPort
PVOID ExceptionPort
PHANDLE_TABLE ObjectTable
EX_FAST_REF Token
PFN_NUMBER WorkingSetPage

KGUARDED_MUTEX AddressCreationLock
KSPIN_LOCK HyperSpaceLock
struct _ETHREAD *ForkInProgress
ULONGLONG_PTR HardwareTrigger;
PMM_AVL_TABLE PhysicalVadRoot
PVOID CloneRoot
PFN_NUMBER NumberOfPrivatePages
PFN_NUMBER NumberOfLockedPages
PVOID Win32Process
struct _EJOB *Job
PVOID SectionObject
PVOID SectionBaseAddress
PEPROCESS_QUOTA_BLOCK QuotaBlock
EPROCESS fields

PPAGEFAULT_HISTORY
  WorkingSetWatch
HANDLE Win32WindowStation
HANDLE InheritedFromUniqueProcessId
PVOID LdtInformation
PVOID VadFreeHint
PVOID VdmObjects
PVOID DeviceMap
PVOID Session
UCHAR ImageFileName[16]
LIST_ENTRY JobLinks
PVOID LockedPagesList
LIST_ENTRY ThreadListHead
ULONG ActiveThreads
PPEB Peb
IO Counters

PVOID AweInfo
MMSUPPORT Vm
Process Flags
NTSTATUS ExitStatus
UCHAR PriorityClass
MM_AVL_TABLE VadRoot
KTHREAD fields

- DISPATCHER_HEADER Header
- LIST_ENTRY MutantListHead
- PVOID InitialStack, StackLimit
- PVOID KernelStack
- KSPIN_LOCK ThreadLock
- ULONG ContextSwitches
- volatile UCHAR State
- KIRQL WaitIrql
- KPROC_MODE WaitMode
- PVOID Teb
- KAPC_STATE ApcState
- KSPIN_LOCK ApcQueueLock
- LONG_PTR WaitStatus
- PRKWAIT_BLOCK WaitBlockList
- BOOLEAN Alertable, WaitNext
- UCHAR WaitReason
- SCHAR Priority
- UCHAR EnableStackSwap
- volatile UCHAR SwapBusy
- LIST_ENTRY WaitListEntry
- NEXT SwapListEntry
- PRKQUEUE Queue
- ULONG WaitTime
- SHORT KernelApcDisable
- SHORT SpecialApcDisable
- KTIMER Timer
- KWAIT_BLOCK WaitBlock[N+1]
- LIST_ENTRY QueueListEntry
- UCHAR ApcStateIndex
- BOOLEAN ApcQueueable
- BOOLEAN Preempted
- BOOLEAN ProcessReadyQueue
- BOOLEAN KernelStackResident

© Microsoft Corporation
KTHREAD fields cont.

UCHAR IdealProcessor
volatile UCHAR NextProcessor
SCHAR BasePriority
SCHAR PriorityDecrement
SCHAR Quantum
BOOLEAN SystemAffinityActive
CCHAR PreviousMode
UCHAR ResourceIndex
UCHAR DisableBoost
KAFFINITY UserAffinity
PKPROCESS Process
KAFFINITY Affinity
PVOID ServiceTable
PKAPC_STATE ApcStatePtr[2]
KAPC_STATE SavedApcState
PVOID CallbackStack
PVOID Win32Thread

PKTRAP_FRAME TrapFrame
ULONG KernelTime, UserTime
PVOID StackBase
KAPC SuspendApc
KSEMAPHORE SuspendSema
PVOID TlsArray
LIST_ENTRY ThreadListEntry
UCHAR LargeStack
UCHAR PowerState
UCHAR Iopl
CCHAR FreezeCnt, SuspendCnt
UCHAR UserIdealProc
volatile UCHAR DeferredProc
UCHAR AdjustReason
SCHAR AdjustIncrement
ETHREAD fields

KTHREAD tcb
Timestamps
LPC locks and links
CLIENT_ID Cid
ImpersonationInfo
IrpList
pProcess
StartAddress
Win32StartAddress
ThreadListEntry
RundownProtect
ThreadPushLock
Thread and Process Enumeration

• Threads and processes all enumerable until their last reference is released
• No need to hold locks while processing each process/thread
• Code uses safe references to prevent the double return to zero problem
Thread Enumeration Example

```c
for (Thread = PsGetNextProcessThread (Process, NULL);
    Thread != NULL;
    Thread = PsGetNextProcessThread (Process, Thread)) {

    st = STATUS_SUCCESS;
    if (Thread != Self) {
        PspTerminateThreadByPointer (Thread, ExitStatus);
    }
}
```
Process Enumeration Internals

PEPROCESS PsGetNextProcess (IN PEPROCESS Process)
{
    for (ListEntry = Process->ActiveProcessLinks.Flink;
        ListEntry != &PsActiveProcessHead;
        ListEntry = ListEntry->Flink) {

        NewProcess = CONTAINING_RECORD (ListEntry,
                                        EPROCESS,
                                        ActiveProcessLinks);

        if (ObReferenceObjectSafe (NewProcess)) {
            break;
        }
    }
    NewProcess = NULL;
}
Process Creation

BOOL WINAPI CreateProcessW(
    LPCWSTR lpApplicationName,
    LPWSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCWSTR lpCurrentDirectory,
    LPSTARTUPINFOW lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
)
Process Creation – CreateProcess

CreateProcess()
Locate imagefile (path search)
Convert DOS name to NT name
Call NtOpenFile()
Call NtCreateSection(SEC_IMAGE)
Check for special handling: VDM, WoW64, restrictions, CMD files
Call NtQuerySection() to get ImageInformation
Use LdrQueryImageFileExecutionOptions() to see if debugging
Special handling for POSIX executable
Create the new process in the kernel via NtCreateProcessEx()
If requested, call NtSetInformationProcess(ProcessPriorityClass)
If (dwCreationFlags & CREATE_DEFAULT_ERROR_MODE)
    call NtSetInformationProcess(ProcessDefaultHardErrorMode)
CreateProcess() – cont.

Call **BasePushProcessParameters()** to push params into new process
Stuff in the standard handles if needed

Call **BaseCreateStack()** to create a user-mode stack in process

Call **BaseInitializeContext()** to create an initial thread context

Call **NtCreateThread()** to create the first thread

// thread may run, so no more modification to new process virtual space

Use **CsrClientCallServer(BasepCreateProcess)** to register new process and thread with CSRSS

If app is restricted
    Set a restricted token on the process
    assign it to a job object so that it can't escape the token.

Unless the initial thread was created suspended, start it with **NtResumeThread()**
NtResumeThread()

Acquire the thread's ApcQueueLock and raise to Synch Level
Decrement the SuspendCount
If SuspendCount and FreezeCount both 0
   Lock the dispatcher database
   Increment the thread's SuspendSemaphore and call KiWaitTest() to resume the thread
   Unlock the dispatcher database
Release the thread's ApcQueueLock
Call KiExitDispatcher(), which may schedule a new thread
**BaseCreateStack** (Process, [StackSize], [MaxStackSize], pInitialTeb)

If not specified, fill StackSize and MaxStackSize out of image header, check PEB for minimum StackSize

Use `NtAllocateVirtualMemory (&Stack, MaxStackSize, MEM_RESERVE)` to reserve the usermode stack

Remember Base/Limit of stack in the TEB

StackTop = Stack + MaxStackSize - StackSize

Commit stack: `NtAllocateVirtualMemory(StackTop, StackSize, MEM_COMMIT)`

If there is room (StackTop > Stack), create a guard page:

`NtProtectVirtualMemory(StackTop - PAGE_SIZE, PAGE_GUARD)`
BasePushProcessParameters()

BasePushProcessParameters(
    dwBasePushProcessParametersFlags, ProcessHandle, Peb, lpApplicationName, CurdirBuffer, QuoteInsert || QuoteCmdLine ? QuotedBuffer : lpCommandLine, lpEnvironment, &StartupInfo, dwCreationFlags | dwNoWindow, bInheritHandles, IsWowBinary ? IMAGE_SUBSYSTEM_WINDOWS_GUI : 0, pAppCompatData, cbAppCompatData
)

© Microsoft Corporation 22
BasePushProcessParameters

BasePushProcessParameters(newproc)

Build up the DLL and EXE search paths, the CommandLineString, CurrentDirString, DesktopInfo, and WindowTitle

Call RtlCreateProcessParameters() to put them into a RTL_USER_PROCESS_PARAMETERS buffer

Call NtAllocateVirtualMemory(newproc) for the environment block
Call NtWriteVirtualMemory(newproc) to copy the environment block

Finish filling in the ProcessParameterBlock
- Copy in more of the main window settings
- Set the console handles for stdin/stdout/stderr
- Set PROFILE flags

Call NtAllocateVirtualMemory(newproc) for ProcessParameterBlock
Copy in with NtWriteVirtualMemory(newproc)

Modify the PEB in newproc so that it points to the parameter block
Allocate and write AppCompat data to the new process
Set pointer in new process’ PEB
RtlCreateProcessParameters()

Formats NT style RTL_USER_PROCESS_PARAMETERS record

Record self-contained in block of memory allocated by this function
Allocation method is opaque so free with RtlDestroyProcessParameters
The process parameters record is created in a de-normalized form
Caller will fill in additional fields before calling RtlCreateUserProcess()
Kernel: \texttt{NtCreateProcessEx()}

Take reference on parent process, if specified
Create an object of PsProcessType for KPROCESS/EPROCESS object
Initialize rundown protection in the thread
Call \texttt{PspInheritQuota()} to set the quota block
Call \texttt{ObInheritDeviceMap()} to setup DosDevices to right device map
If passed section handle, take reference -- otherwise clone parent VA
If cloning parent, acquire rundown protection to avoid parent exit
If passed debug and/or exception ports, point newproc at them
Call \texttt{MmCreateProcessAddressSpace()}
If not cloning a parent
  Process->ObjectTable = CurrentProcess->ObjectTable
Call \texttt{KeInitializeProcess()} to init newproc with default scheduling information and mark newproc as InMemory
Call \texttt{PspInitializeProcessSecurity()} to duplicate the parents token as the primary token for the process
Initialize the fast references for newproc’s token
Set newproc’s scheduling parameters
If cloning a parent Call \texttt{ObInitProcess()}
NtCreateProcessEx() – cont.

// Initialize newproc’s address space. Four possibilities

Boot Process: Address space already initialized by MmInit()

System Process: Address space only maps system space
(process is same as PspInitialSystemProcess)

Cloned User Process: Address space cloned from specified parent

New User Process: Address is initialized to map specified section

If cloning parent

Call MmInitializeProcessAddressSpace(Process, Parent)

else

Call MmInitializeProcessAddressSpace(Process, SectionObject)

Call ExCreateHandle(PspCidTable) to allocate a CID for the process

Set the process CID in the handle table (for checks and debugging)

If parent in a job add in this process to the job

If cloning parent

Call MmCreatePeb()

else

Copy the parents PEB via MmCopyVirtualMemory()
NtCreateProcessEx() – cont. 2

Insert new process into the global process list (PsActiveProcessHead)
Call SeCreateAccessStateEx() to create an AccessState structure
Call ObInsertObject(Process, AccessState, DesiredAccess, &handle) into the handle table
Write the handle back into the user-mode handle buffer
Call ObGetObjectSecurity (Process, &SecurityDescriptor) and pass to SeAccessCheck()
If the access check fails, take away all process access rights
Call KeQuerySystemTime (&Process->CreateTime)
Give back the extra reference we used to keep the process from being prematurely deleted
NtCreateSection(SEC_IMAGE)

Validate/capture parameters and call MmCreateSection()
Call `CcWaitForUninitializeCacheMap()` to synch with teardown of residual data section.refs in cache manager
Allocate a temporary ControlArea
Acquire the ERESOURCE lock to synchronize with the file system
Call `MiFindImageSectionObject()` to find an existing image ControlArea for this file
Call `MiLockPfnDatabase()` to take PFN lock
Deal with race conditions, like existing ControlArea being deleted
Call `MiUnlockPfnDatabase()` to release PFN lock
If existing ControlArea
   New SectionObject will share the segment in the existing ControlArea, so
   NumberOfSectionReferences++
   Call MiFlushDataSection() to flush any data section for the file
   Discard the temporary ControlArea
   Release the ERESOURCE file system lock
else
   Use the temporary ControlArea we allocated
   Call MilnsertImageSectionObject(File, ControlArea) to insert the
   new ControlArea into the FileObject
   Call MiCreateImageFileMap(File, &Segment) to do the actual
   mapping and create real ControlArea
   Call KeAcquireQueuedSpinLock(LockQueuePfnLock)
   Call MiRemoveImageSectionObject (File, NewControlArea)
   Call MilnsertImageSectionObject (File, real ControlArea)
   Delete the temporary ControlArea
   Deal with race conditions, like another thread creating the same
   section
   Call KeReleaseQueuedSpinLock(LockQueuePfnLock)
NtCreateSection (SEC_IMAGE) – 3

Call `ObCreateObject(MmSectionObjectType, &NewSectionObject)` to create the real section object.
Fill in `NewSectionObject` with the values we have accumulated on our stack.
Pass out the `NewSectionObject`.

`ObInsertObject(Section, ..., &handle)`
MiFindImageSectionObject()

Searches the control area chains (if any) for an existing cache of the specified image file
For non-global control areas, there is no chain and control area is shared for all callers and sessions
Likewise for systemwide global control areas
For global PER-SESSION control areas, we must walk the list

MiInsertImageSectionObject()

Inserts the control area into the file's section object pointers
For non-global control areas and systemwide, there is no chain …
For global PER-SESSION control areas, we must do a list insertion
MiCreateImageFileMap()

Call **FsRtlGetFileSize(File,&EndOfFile)**
Read in the image header and validate it:
  - Initialize an Event and an Mdl on the stack
  - Call **MiGetPageForHeader()** to allocate pageframe for image header
  - Call **MiFlushDataSection()**
  - Call **IoPageRead(File, Mdl, 0, Event)** to do the read
  - Wait on the Event
  - Call **MiMapImageHeaderInHyperSpace()** to map the image header into per-process KVA
  - Validate image header
    - If header more than one page, read another 8KB

Compute the number of PTEs needed to map the image
Allocate a control area and a subsection for each section header plus one for the image header which has no section
Establish the prototype PTEs for each subsection, and point them all at their subsection
Return the Segment
MmCreateProcessAddressSpace
(x86)

Take the WorkingSet lock
Take the PFN lock so we can get physical pages
Allocate the page directory and set into DirectoryTableBase[0]
Allocate the page directory for hyperspace and set into DirectoryTableBase[1]
Allocate pages for the VAD allocation bitmap and the working set list
Release the PFN lock
Initialize the hyperspace map
Under the expansion lock insert the new process onto MM's internal ProcessList
Map the page directory page into hyperspace
Setup the self-map
Fill in the system page directories
Release the WorkingSet lock
Increment the session reference count
MmCreatePeb()

Attach to the target process
Map in the NLS tables
Call MiCreatePebOrTeb() to allocate a PEB in the user address space
Initialize the PEB, including values from the InitialPeb, the NLS tables, the system defaults, and the image header
Detach from the process
Return the allocated PEB address

MiCreatePebOrTeb()

Allocate VAD and mark non-deletable and with unchangeable protection
Lock the address space
Find a VA for the block
Finish initializing the VAD
Unlock the address space
NtCreateThread()

Take a reference on the process that will contain the thread
Create an object of PsThreadType (this will contain the
  KTHREAD/ETHREAD data structure)
Initialize the rundown protection in the thread
Point the thread at its process
Initialize the various fields used by MM, LPC, IO, Registry, thread lock,
  timers, queues, etc.
Call `ExAcquireRundownProtection()` to keep the process from
  terminating (bail if it is already doing so)
Call `MmCreateTeb()` to create the user-mode TEB
Set the StartAddress and Win32StartAddress in the kernel thread object
Call `KeInitThread()` to finish setting up the thread object
**N.B.** kernel-mode execution will begin at PspUserThreadStartup
NtCreateThread() – cont.

Take the process lock: **PspLockProcessExclusive()**
Process->ActiveThreads++
Insert thread at tail of Process->ThreadList
Call **KeStartThread()** to set up thread
Call **PspUnlockProcessExclusive()**
Call **ExReleaseRundownProtection()**
If this is the first thread in the process invoke callbacks registered for notification of process creation
If process is in a job and this is our first chance to report in, send the notification to the job's CompletionPort
Invoke callbacks for notification of thread creation
If thread was to be created suspended, call **KeSuspendThread()** on it
Call **SeCreateAccessStateEx()** to create an AccessState structure
Call **ObInsertObject(Thread, AccessState, DesiredAccess, &handle)** into the handle table
Write the handle back into the user-mode handle buffer
NtCreateThread() – cont. 2

Set the thread CreateTime
Call ObGetObjectSecurity (Thread, &SecurityDescriptor) and pass to SeAccessCheck()
If the access check fails, take away all access to the thread except terminate, set/query information
Call KeReadyThread()
Give back the extra reference we used to keep the thread from being prematurely deleted
CsrClientCallServer (BasepCreateProcess)

AcquireProcessStructureLock()
Duplicate handles to the process and thread into CSRSS
Allocate a process structure within CSRSS
Copy any per-process data from parent structure to child structure
Set CSRSS' CsrApiPort to be the child's exception port
If the process being debugged, setup debug port and the process group, if we are the leader.
Capture thread creation time as a sequence number for the tid
Allocate a thread structure within CSRSS
Increment process ThreadCount, insert thread into process ThreadList
Insert thread into CsrThreadHashTable[]
Bump reference count on current session
Write the pid/tid into process and thread structures
Save the duplicated process/thread handles in their respective structures
Add the process to the tail of the global list
For each DLL loaded in CSRSS notify it about the new process
Tell the kernel that the new process is a background process
ReleaseProcessStructureLock()
KeInitThread()

The priority, affinity, and initial quantum are taken from the parent process object
Initialize most the other fields including the thread context
Thread->State = Initialized
Set intial code to run: PspUserThreadStartup()

PspUserThreadStartup()

Call KilnitializeUserApc() to set an initial user-mode APC to the thread
Initial APC will execute LdrInitializeThunk()
KeStartThread()

Initialize some more fields (DisableBoost, Iopl, Quantum, ...)
Raise to SYNC_LEVEL and acquire ProcessLock
Copy the BasePriority and Affinity from the process
Set the IdealProcessor
Lock the dispatcher database
Insert thread into process list and increment process StackCount
Unlock the dispatcher database
Lower the IRQL and release ProcessLock
LdrInitialize()/LdrpInitialize()

// LdrpProcessInitialized
// 0 means no thread has been tasked to initialize the process
// 1 means a thread has been tasked but has not yet finished
// 2 means a thread has been tasked and initialization is complete

while (1 == InterlockCompExch (&LdrpProcessInitialized, 1, 0))
    while (LdrpProcessInitialized == 1) NtDelayExecution(30mS)
If LdrpProcessInitialized == 0
    Initialize the LoaderLock
    Call LdrpInitializeProcess()
    LdrpTouchThreadStack (Peb->MinimumStackCommit)
    InterlockedIncrement (&LdrpProcessInitialized) // 1 -> 2
else
    if (Peb->InheritedAddressSpace)
        Initialize critical section list // otherwise don’t clobber the clone
    else
        Call LdrpInitializeThread()
LdrpInitializeProcess()

Figure out the image name from the ProcessParameters
NtHeader = RtlImageNtHeader(Peb->ImageBaseAddress)
Check ImageFileExecutionOptions for this image in the registry
ProcessParameters = RtlNormalizeProcessParams
                      (Peb->ProcessParameters)
RtlInitNlsTables (Peb->AnsiCodePageData, Peb->OemCodePageData,
                 Peb->UnicodeCaseTableData, &xInitTableInfo)
Setup process parameters based on the image file
Initialize process data structures for allocation TLS and FLS
Initialize the LoaderLock
Initialize various critical sections
Call RtlInitializeHeapManager()
ProcessHeap = RtlCreateHeap()
LdrpHeap = RtlCreateHeap()
Call RtlInitializeAtomPackage()
Setup DLL search path and current directory from ProcessParameters
LdrpInitializeProcess() – cont.

Initialize the loaded module list and insert the image into the list
If this is a Windows GUI app, load Call LdrLoadDll(kernel32.dll)
Call LdrpWalkImportDescriptor() to recursively walk the Import
 Descriptor Table (IDT) and load each referenced DLL
If the image was not loaded at the base address in the binary, toggle
 page protections and call LdrRelocateImage()
Call LdrpInitializeTls()
Now that all DLLs are loaded, if (Peb->BeingDebugged)
    Call DbgBreakPoint() to notify the debugger
Load AppCompat shim engine and shims
Call LdrpRunInitializeRoutines() to run all the DLL initialization routines
LdrpInitializeThread()

Take the LoaderLock
Walk the loaded module list calling the DLL init routines:
  LdrpCallInitRoutine(DLL_THREAD_ATTACH)
If the image has TLS, call its initializers:
  LdrpCallTlsInitializers(DLL_THREAD_ATTACH)
Release the LoaderLock
Synchronization Classes

- Write once fields like process job and thread impersonation info
- Torn down (rundown) structures like handle table, thread TEB etc
- Infrequently changing fields like the process token
- Frequently changing stuff like thread list of a process or impersonation token
Process Synchronization

- ProcessLock – Protects thread list, token
- RundownProtect – Cross process address space, image section and handle table references
- Token, Prefetch – Uses fast referencing
- AWE – Uses cache aware pushlocks
- Token, Job – Torn down at last process dereference without synchronization
Thread scheduling states

Transition

KiReadyThread

KiInsertDeferredRead

Ready process
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
Process Lifetime

• Created as an empty shell
• Address space created with only ntdll and the main image unless forked
• Handle table created empty or populated via duplication from parent
• Process is partially destroyed on last thread exit
• Process totally destroyed on last dereference
Thread Lifetime

• Created within a process with a CONTEXT record
• Starts running in the kernel but has a trap frame to return to use mode
• Kernel queues user APC to do ntdll initialization
• Terminated by a thread calling NtTerminateThread/Process
NtTerminateThread(thandle, status)

PspTerminateThreadByPointer(pThread, status, bSelf)
if (bSelf)
    PspExitThread(status)  // never returns
if Thread->CrossThreadFlags & _TERMINATED
    return

exitApc = ExAllocatePool(sizeof(KAPC))
KeInitializeApc (ExitApc,
    Thread,
    OriginalApcEnvironment,  // thread has to detach before exiting
    PsExitSpecialApc,
    PspExitApcRundown,  // runs at end to free exitApc
    PspExitNormalApc,
    KernelMode,
    status)
KeInsertQueueApc (ExitApc, ExitApc, NULL, 2)
KeForceResumeThread (&Thread->Tcb)
PspExitThread(status)

ExWaitForRundownProtectionRelease (&Thread->RundownProtect)
<Notify registered callout routines of thread exit>
PspLockProcessExclusive (Process, Thread)
Process->ActiveThreads--
if (Process->ActiveThreads == 0)
  LastThread = TRUE
  Process->Flags |= PROCESS_DELETE
  Wait until all other threads have exited
PspUnlockProcessExclusive (Process, Thread)
if (Process->DebugPort)
  LastThread? DbgkExitProcess (status) : DbgkExitThread (status)
// rundown Win32
(PspW32ThreadCallout) (Thread, PsW32ThreadCalloutExit)
if (LastThread)
  (PspW32ProcessCallout) (Process)
PspExitThread(status) cont. 1

IoCancelThreadIo (Thread)
ExTimerRundown ()
CmNotifyRunDown (Thread)
KeRundownThread ()
<Delete the thread's TEB>
LpcExitThread (Thread)
Thread->ExitStatus = ExitStatus;
KeQuerySystemTime (&Thread->ExitTime)
if (! LastThread)
  <Rundown pending APCs>
  KeTerminateThread ()
PspExitThread(status) cont. 2

Process->ExitTime = Thread->ExitTime
PspExitProcess (TRUE, Process)
ProcessToken = PsReferencePrimaryToken (Process)
SeAuditProcessExit (Process);
PsDereferencePrimaryTokenEx (Process, ProcessToken)
ObKillProcess (Process) // Rundown the handle table
ObDereferenceObject (Process->SectionObject)
PspExitProcessFromJob (Process->Job, Process)
<Rundown pending APCs>
MmCleanProcessAddressSpace (Process)
KeSetProcess (&Process->Pcb, 0) // signal the process
KeTerminateThread ()
PspExitProcess(LastThread, Process)

ObDereferenceObject (Process->SecurityPort)
if (LastThread)
    <Notify registered callout routines of process exit>
    <Finish cleaning up Job Object>
    return
// we were called from PspDeleteProcess()
MmCleanProcessAddressSpace (Process)
KeTerminateThread()

<Raise to SYNCH_LEVEL, acquire process lock, set swap busy>
<Insert the thread in the reaper list>
<Acquire dispatcher lock>
<Queue reaper work item if needed>
if (Thread->Queue)
    RemoveEntryList(&Thread->QueueListEntry)
    KiActivateWaiterQueue (Queue)
RemoveEntryList(&Thread->ThreadListEntry)       // from parent’s list
<Release process lock without dropping IRQL>
Thread->State = Terminated
Process->StackCount -= 1
<Initiate an outswap of the process if StackCount now 0>
KiRundownThread (Thread)               // rundown arch-specific data
<Acquire dispatcher lock>
KiSwapThread (Thread, CurrentPrcb)     // yield processor final time
PspProcessDelete ()

- Remove the process from the global list
- PspRemoveProcessFromJob (Process->Job, Process)
- ObDereferenceObjectDeferDelete (Process->Job)
- ObDereferenceObject (Process->DebugPort)
- ObDereferenceObject (Process->ExceptionPort)
- ObDereferenceObject (Process->SectionObject)
- PspDeleteLdt (Process)
- KeStackAttachProcess (&Process->Pcb, &ApcState)
  - ObKillProcess (Process)
  - PspExitProcess (FALSE, Process)
- KeUnstackDetachProcess (&ApcState)
- MmDeleteProcessAddressSpace (Process)
- ExDestroyHandle (PspCidTable, Process->UniqueProcessId)
- PspDeleteProcessSecurity (Process)
- ObDereferenceDeviceMap (Process)
- PspDereferenceQuota (Process)
PspThreadDelete()

MmDeleteKernelStack()
ExDestroyHandle (PspCidTable, Thread->Cid.UniqueThread)
PspDeleteThreadSecurity (Thread)
if (! Thread->Process) return       // never inserted in process

PspLockProcessExclusive (Process, CurrentThread)
RemoveEntryList (&Thread->ThreadListEntry)
PspUnlockProcessExclusive (Process, CurrentThread)
ObDereferenceObject (Process)
**Summary: Native NT Process APIs**

<table>
<thead>
<tr>
<th>Process API</th>
<th>Thread API</th>
</tr>
</thead>
<tbody>
<tr>
<td>NtCreateProcess()</td>
<td>NtCreateThread()</td>
</tr>
<tr>
<td>NtTerminateProcess()</td>
<td>NtTerminateThread()</td>
</tr>
<tr>
<td>NtQueryInformationProcess()</td>
<td>NtSuspendThread()</td>
</tr>
<tr>
<td>NtSetInformationProcess()</td>
<td>NtResumeThread()</td>
</tr>
<tr>
<td>NtGetNextProcess()</td>
<td>NtGetContextThread()</td>
</tr>
<tr>
<td>NtGetNextThread()</td>
<td>NtSetContextThread()</td>
</tr>
<tr>
<td>NtSuspendProcess()</td>
<td>NtQueryInformationThread()</td>
</tr>
<tr>
<td>NtResumeProcess()</td>
<td>NtSetInformationThread()</td>
</tr>
<tr>
<td>NtAlertThread()</td>
<td></td>
</tr>
<tr>
<td>NtQueueApcThread()</td>
<td></td>
</tr>
</tbody>
</table>
Discussion