Windows Processes and Threads
Roadmap for This Lecture

- Windows Process Internals
  - Process data structures
  - Performance counters
  - Process APIs
  - Protected Processes
  - Process creation
- Windows Thread Internals
  - Thread data structures
  - Performance counters
  - Thread APIs
  - Thread creations
- Windows tools for Processes and Threads
- Windows Jobs
- Labs Demo
Data Structures for each process/thread:

- Executive process block (EPROCESS)
- Executive thread block (ETHREAD)
- Win32 process block
- Process environment block
- Thread environment block
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
- Memory management structures like the Working Set, VAD tree, AWE etc
Processes & Threads
Internal Data Structures

Process Object

Handle Table

Virtual Address Space Descriptors

VAD
VAD
VAD

object
object
object

Thread
Thread
Thread

Access Token

Access Token

See kernel debugger commands:
  dt
  !process
  !thread
  !token
  !handle
  !object
Per-Process Data

Each process has its own…

- Virtual address space (including program code, 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 Windows kernel objects
- Environment strings
- Command line

These are common to all threads in the process, but separate and protected between processes
Executive Process Block Layout

- **Kernel Process Block (or PCB)**
  - Process ID
  - Parent Process ID
  - Exit Status
  - Create and Exit Time
  - Next Process Block
  - Quota Block
  - Memory Management Info
  - Exception Port
  - Debugger Port
  - Process Environment Block
  - Image File Name
  - Image Base Address
  - Process Priority Class

- **EPROCESS Block**
  - Primary Access Token
  - Handle Table

- **Win32 Process Block**

- **KPROCESS Block (PCB)**
  - Contains info needed to schedule threads in the process

- **Dispatcher Header**
  - Kernel Time
  - User Time
  - Inswap/Outswap List Entry

- **Process Spin Lock**
- **Processor Affinity**
- **Resident Kernel Stack Count**
- **Process Base Priority**
- **Default Thread Quantum**
- **Process State**
- **Thread Seed**
- **Disable Boost Flag**

- **Process Page Directory**
Process Environment Block (PEB)

- Mapped in user space
- Image loader, heap manager, Windows system DLLs use this info
- View with !peb or dt nt!_peb

<table>
<thead>
<tr>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image base address</td>
</tr>
<tr>
<td>Module list</td>
</tr>
<tr>
<td>Thread-local storage data</td>
</tr>
<tr>
<td>Code page data</td>
</tr>
<tr>
<td>Critical section time-out</td>
</tr>
<tr>
<td>Number of heaps</td>
</tr>
<tr>
<td>Heap size info</td>
</tr>
<tr>
<td>GDI shared handle table</td>
</tr>
<tr>
<td>OS version no info</td>
</tr>
<tr>
<td>Image version info</td>
</tr>
<tr>
<td>Image process affinity mask</td>
</tr>
</tbody>
</table>

Process heap
# Process-Related Performance Counters

<table>
<thead>
<tr>
<th>Object: Counter</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process:%PrivilegedTime</td>
<td>Percentage of time that the threads in the process have run in kernel mode</td>
</tr>
<tr>
<td>Process:%ProcessorTime</td>
<td>Percentage of CPU time that threads have used during specified interval</td>
</tr>
<tr>
<td></td>
<td>%PrivilegedTime + %UserTime</td>
</tr>
<tr>
<td>Process:%UserTime</td>
<td>Percentage of time that the threads in the process have run in user mode</td>
</tr>
<tr>
<td>Process: ElapsedTime</td>
<td>Total lifetime of process in seconds</td>
</tr>
<tr>
<td>Process: ID Process</td>
<td>PID – process IDs are re-used</td>
</tr>
<tr>
<td>Process: ThreadCount</td>
<td>Number of threads in a process</td>
</tr>
</tbody>
</table>
Process Windows APIs

- CreateProcess
- OpenProcess
- GetCurrentProcessId - returns a global ID
- GetCurrentProcess - returns a pseudo-handle
- ExitProcess – notifies attached DLL
- TerminateProcess - no DLL notification
- Get/SetProcessShutdownParameters
- GetExitCodeProcess
- GetProcessTimes
- GetStartupInfo
Protected Processes

- Process with debug privilege:
  - Read/write any process memory
  - Inject code
  - Suspend and resume thread, etc
  - E.g. Process explorer and task manager

- Media industry requires protection when playing back advanced, high quality digital content
  - Blu-ray, HD-DVD

- Images file with Windows Media Certificate
  - Audiodg.exe and Windows Error Reporting (WER)

- Indicated by a flag in EPROCESS block

- Accessible to Windbg (kernel mode)
Flow of CreateProcess()

1. Validate parameters; convert subsystems flags and options to their native counterparts; parse, validate and convert attribute list to native counterparts

2. Open the image file (.EXE) to be executed inside the process

3. Create Windows NT executive process object

4. Create initial thread (stack, context, Win NT executive thread object)

5. Notify Windows subsystem of new process so that it can set up for new proc.& thread

6. Start execution of initial thread (unless CREATE_SUSPENDED was specified)

7. In context of new process/thread: complete initialization of address space (load DLLs) and begin execution of the program
Create a Windows Process

Convert/validate Params & flags

Open EXE and create section object

Create NT process object

Create NT thread object

Notify Windows subsystem

Start execution of the initial thread

Return to caller

Creating process

Windows subsystem

Set up for new process and thread

New process

Final process/image initialization

Start execution at entry point to image
Converting and validating params

CreationFlags: independent bits for priority class
→ NT assigns *lowest-priority* class set

Default priority class is normal unless creator has priority class idle

If real-time priority class is specified and creator has insufficient privileges:
  - The high priority class is used

Caller’s current desktop is used if no desktop is specified
Opening the image to be executed

What kind of application is it?

- Run CMD.EXE
  - MS-DOS .BAT or .CMD
- Run NTVDM.EXE
  - Win16 (not supported on 64-bit Windows)
- Use .EXE directly
  - Windows
  - Win32 (on 64-bit Windows)
  - MS-DOS .EXE, .COM, or .PIF
- Run OS2.EXE
- Run POSIX.EXE
- Run POSIX.EXE
- Run NTVDM.EXE

Win16 (not supported on 64-bit Windows)
If executable has no Windows format...

- CreateProcess uses Windows “support image”
- No way to create non-Windows processes directly
  - OS2.EXE runs only on Intel systems
  - Multiple MS-DOS apps may share virtual dos machine
  - .BAT of .CMD files are interpreted by CMD.EXE
  - Win16 apps may share virtual dos machine (VDM)
    Flags: CREATE_SEPARATE_WOW_VDM
    CREATE_SHARED_WOW_VDM
    Default: HKLM\System...\Control\WOW\DefaultSeparateVDM
- Sharing of VDM only if apps run on same desktop under same security
- Debugger may be specified under (run instead of app !!)
  \Software\Microsoft\WindowsNT\CurrentVersion\ImageFileExecutionOptions
Process Creation - next Steps...

CreateProcess has opened Windows executable and created a section object to map in process’ address space

Now: create executive process object via NtCreateProcess

- Set up EPROCESS block
- Create initial process address space (page directory, hyperspace page, working set list)
- Create kernel process block (set initial priority and quantum)
- Conclude setup of process address space (VM, map NTDLL.DLL, map language support tables, register process: PsActiveProcessHead)
- Set up Process Environment Block
- Complete setup of executive process object
Further Steps...(contd.)

Create Initial Thread and Its Stack and Context
- NtCreateThread; new thread is suspended until CreateProcess returns

Notify Windows Subsystem about new process
- KERNEL32.DLL sends message to Windows subsystem including:
  - Process and thread handles
  - Entries in creation flags
  - ID of process’s creator
  - Flag describing Windows app (CSRSS may show startup cursor)

Windows subsystem:
- duplicate handles (inc usage count), set priority class, bookkeeping
- allocate CSRSS proc/thread block, init exception port, init debug port
- Show cursor (arrow & hourglass), wait 2 sec for GUI call, then wait 5 sec for app to show window
CreateProcess: final steps

Process Initialization in context of new process:

- **KiThreadStartup** Lowers IRQL level (DPC/Dispatch → APC level)
- Enable working set expansion
- Queue APC to exec **LdrInitializeThunk** in NTDLL.DLL
- Lower IRQL level to 0 – APC fires,
  - Init loader, heap manager, NLS tables, TLS array, crit. sect. Structures
  - Load DLLs, call DLL_PROCESS_ATTACH function
- Debuggee: all threads are suspended
  - Send msg to proc’s debug port
    (Windows creates CREATE_PROCESS_DEBUG_INFO event)
- Image begins execution in user-mode (return from trap)
Process Shutdown Sequence

1. DLL notification
   - unless TerminateProcess used
2. All handles to executive and kernel objects are closed
3. Terminate any active threads
4. Process’s exit code changes from STILL_ACTIVE to the specified exit code

```c
BOOL GetExitCodeProcess(
    HANDLE hProcess,
    LPDWORD lpdwExitCode);
```
5. Process object & thread objects become signaled
6. When handle and reference counts to process object == 0, process object is deleted
Windows Thread Internals

Data Structures for each process/thread:
- Executive process block (EPROCESS)
- Executive thread block (ETHREAD)
- Win32 process block
- Process environment block
- Thread environment block
Thread

- Fundamental schedulable entity in the system
- Represented by ETHREAD that includes a KTHREAD
- Queued to the process (both E and K thread)
- I/O Request Packet list
- Impersonation Access Token
- Unique thread ID
- Associated User-mode Thread Environment Block (TEB)
- User-mode stack
- Kernel-mode stack
Per-Thread Data

Each thread has its own…

- User-mode stack (arguments passed to thread, automatic storage, call frames, etc.)
- Kernel-mode stack (for system calls)
- Thread Local Storage (TLS) – array of pointers to allocate unique data
- Scheduling state (Wait, Ready, Running, etc.) and priority
- Hardware context (saved in CONTEXT structure if not running)
  - Program counter, stack pointer, register values
  - Current access mode (user mode or kernel mode)
- Access token (optional -- overrides process’s if present)
Thread Block

**ETHREAD**

- **KTHREAD**
  - Create and Exit Time
  - Process ID
  - Thread Start Address
  - Impersonation Information
  - LPC Message Information
  - Timer Information

- **EPROCESS**
- **Access Token**

- **Pending I/O Requests**

**KTHREAD**

- Dispatcher Header
- Total User Time
- Total Kernel Time
- Thread Scheduling Information
- Trap Frame
- Synchronization Information
- List of Pending APCs
- Timer Block and Wait Blocks
- List of Objects Being Waiting On

- Kernel Stack Information
- System Service Table
- Thread Local Storage
- TEB
Thread Environment Block

- User mode data structure
- Context for image loader and various Windows DLLs
- View with !teb or dt nt!_teb

<table>
<thead>
<tr>
<th>Exception list</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack base</td>
</tr>
<tr>
<td>Stack limit</td>
</tr>
<tr>
<td>Thread ID</td>
</tr>
<tr>
<td>Active RPC handle</td>
</tr>
<tr>
<td>LastError value</td>
</tr>
<tr>
<td>Count of owned crit. sect.</td>
</tr>
<tr>
<td>Current locale</td>
</tr>
<tr>
<td>User32 client info</td>
</tr>
<tr>
<td>GDI32 info</td>
</tr>
<tr>
<td>OpenGL info</td>
</tr>
<tr>
<td>TLS array</td>
</tr>
</tbody>
</table>

- Subsyst. TIB
- Fiber info
- PEB
- Winsock data
# Thread-Related Performance Counters

<table>
<thead>
<tr>
<th>Object: Counter</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process: Priority Base</td>
<td>Base priority of process: starting priority for thread within process</td>
</tr>
<tr>
<td>Thread:%PrivilegedTime</td>
<td>Percentage of time that the thread was run in kernel mode</td>
</tr>
<tr>
<td>Thread:%ProcessorTime</td>
<td>Percentage of CPU time that the threads has used during specified interval</td>
</tr>
<tr>
<td></td>
<td>%PrivilegedTime + %UserTime</td>
</tr>
<tr>
<td>Thread:%UserTime</td>
<td>Percentage of time that the thread has run in user mode</td>
</tr>
<tr>
<td>Thread: ElapsedTime</td>
<td>Total lifetime of thread in seconds</td>
</tr>
<tr>
<td>Thread: ID Process</td>
<td>PID – process IDs are re-used</td>
</tr>
<tr>
<td>Thread: ID Thread</td>
<td>Thread ID – re-used</td>
</tr>
</tbody>
</table>
# Thread-Related Performance Counters (contd.)

<table>
<thead>
<tr>
<th>Object: Counter</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread: Priority Base</td>
<td>Base priority of thread: may differ from the thread’s starting priority</td>
</tr>
<tr>
<td>Thread: Priority Current</td>
<td>The thread’s current dynamic priority</td>
</tr>
<tr>
<td>Thread: Start Address</td>
<td>The thread’s starting virtual address (the same for most threads)</td>
</tr>
<tr>
<td>Thread: Thread State</td>
<td>Value from 0 through 7 – current state of thread</td>
</tr>
<tr>
<td>Thread: Thread Wait Reason</td>
<td>Value from 0 through 19 – reason why the thread is in wait state</td>
</tr>
</tbody>
</table>
Windows Thread APIs

- CreateThread
- CreateRemoteThread
- GetCurrentThreadId - returns global ID
- GetCurrentThread - returns handle
- SuspendThread/ResumeThread
- ExitThread – notifies DLLs
- TerminateThread - no DLL notification
- GetExitCodeThread
- GetThreadTimes

Windows 2000 adds:
- OpenThread
- new thread pooling APIs
Birth of a Thread

`CreateThread` Function in Kernel32.dll:

1. Converts API params to native flags and builds native
   `OBJECT_ATTRIBUTES`

2. Builds attribute lists of: client ID and TEB address (return after
   creation)

3. Call `NTCreateThreadEx` to create user-mode context, which calls
   `PspCreateThread` to create suspended ETHREAD object
   1. Create and initialize ETHREAD
   2. Set up the stack and context
   3. Allocate TEB for new thread
   4. Store start address in ETHREAD
   5. `KeInitThread` is called to set up the KTHREAD block
Birth of a Thread

6. `CreateThread` allocates activation stack and activates it
7. Notify Windows subsystems about the new thread
8. Thread handle and ID are returned
9. Thread is resumed and calls `KiThreadStartup` before calling the user specified start address
Thread Rundown Sequence

1. DLL notification
   - unless TerminateThread was used
2. All handles to Windows User and GDI objects are closed
3. Outstanding I/Os are cancelled
4. Thread stack is deallocated
5. Thread’s exit code changes from STILL_ACTIVE to the specified exit code

```c
BOOL GetExitCodeThread(
    HANDLE hThread,
    LPDWORD lpdwExitCode);
```

6. Thread kernel object becomes signaled
7. When handle and reference counts == 0, thread object deleted
8. If last thread in process, process exits
Start of Thread Wrapper

All threads in all Windows processes appear to have one of just two different start addresses, regardless of the .EXE running

- One for thread 0 (start of process wrapper), the other for all other threads (start of thread wrapper \textit{RtlUserThreadStart} in Ntdll.dll)

These “wrapper” functions are what Process Viewer shows as Thread Start Address for Windows apps

Start of process & start of thread wrappers have same behavior

- Provides default exception handling, access to debugger, etc.
- Forces thread exit when thread function returns

To find “real” Windows start address, use TLIST <processname> (or Kernel Debugger !thread command)
Tools for Obtaining Process & Thread Information

Many overlapping tools (most show one item the others do not)

Built-in tools in Windows XP +:
- Task Manager, Performance Tool
- Tasklist (new in XP)

Support Tools:
- pviewer - process and thread details (GUI)
- pmon - process list (character cell)
- tlist - shows process tree and thread details (character cell)

Resource Kit tools:
- apimon - system call and page fault monitoring (GUI)
- oh – display open handles (character cell)
- pviewer - processes and threads and security details (GUI)
- ptree – display process tree and kill remote processes (GUI)
- pulist - lists processes and usernames (character cell)
- pstat - process/threads and driver addresses (character cell)
- qslice - can show process-relative thread activity (GUI)

Tools from www.sysinternals.com
- Process Explorer – super Task Manager – shows open files, loaded DLLs, security info, etc.
- Pslist – list processes on local or remote systems
- Ntpmon - shows process/thread create/deletes (and context switches on MP systems only)
- Listdlls - displays full path of EXE & DLLs loaded in each process
Jobs are collections of processes

- Can be used to specify limits on CPU, memory, and security
- Enables control over some unique process & thread settings not available through any process or thread system call
  - E.g. length of thread time slice

Job object is a nameable, secure and shareable kernel object

- Allows a group of processes to be managed and manipulated as a unit
Creation of Jobs

How do processes become part of a job?

- Job object has to be created (CreateJobObject)
- Then processes are explicitly added (AssignProcessToJob)
  - Processes created by processes in a job automatically are part of the job
    - Unless restricted, processes can “break away” from a job
- Then quotas and limits are defined (SetInformationJobObject)
  - Examples on next slide…
Job Settings

Quotas and restrictions:

- Quotas: total CPU time, # active processes, per-process CPU time, memory usage
- Run-time restrictions: priority of all the processes in job; processors threads in job can run on
- Security restrictions: limits what processes can do
  - Not acquire administrative privileges
  - Not accessing windows outside the job, no reading/writing the clipboard
- Scheduling class: number from 0-9 (5 is default) - affects length of thread timeslice (or quantum)
  - E.g. can be used to achieve “class scheduling” (partition CPU)
Examples of Jobs

Examples where Windows OS uses jobs:

- Add/Remove Programs ("ARP Job")
- WMI provider
- RUNAS service (SecLogon) uses jobs to terminate processes at log out

Process Explorer highlights processes that are members of jobs

- Color can be configured with Options->Configure Highlighting
- For processes in a job, click on Job tab in process properties to see details
Further Reading


- Chapter 5 - Processes, Thread, and Jobs (from pp. 335)
- Process Internals (from pp. 335)
- Flow of Create Process (from pp. 348)
- Thread Internals (from pp. 370)
Lab: EPROCESS, KPROCESS and PEB blocks

lkd> !dt _eprocess
lkd> !dt _kprocess
lkd> !process
Lab: Show Windows API

2013-9-30
#include <Windows.h>


Hungarian notation (Wikipedia)
BOOL WINAPI CreateProcess(
    _In_opt_     LPCTSTR lpApplicationName,
    _Inout_opt_  LPTSTR lpCommandLine,
    _In_opt_     LPSECURITY_ATTRIBUTES lpProcessAttributes,
    _In_opt_     LPSECURITY_ATTRIBUTES lpThreadAttributes,
    _In_         BOOL bInheritHandles,
    _In_         DWORD dwCreationFlags,
    _In_opt_     LPVOID lpEnvironment,
    _In_opt_     LPCTSTR lpCurrentDirectory,
    _In_         LPSTARTUPINFO lpStartupInfo,
    _Out_        LPPROCESS_INFORMATION lpProcessInformation
);
TerminateProcess()

BOOL WINAPI TerminateProcess(
    _In_  HANDLE hProcess,
    _In_  UINT uExitCode
);

Lab: Start a process image

Install a debugger to run instead of notepad.exe. We chose Solitaire (sol.exe – a standard tool on every Windows system).

- start regedit.exe
- create (insert) key at HKLM\Software\Microsoft\WindowsNT\CurrentVersion\Image File Execution Options\notepad.exe
- insert value: Debugger (REG_SZ) C:\winnt\system32\sol.exe
- start notepad (!)
Lab: Trace Process Startup
Lab: ETHREAD, KTHREAD and TEB

lkd> dt nt!_ethread
lkd> dt nt!_kthread