Operating Systems Case Study
Windows

Universität Innsbruck

Andreas Schabus
Microsoft Österreich GmbH
aschabus@microsoft.com
Contents

1 Background and introduction
2 System Architecture
3 System Mechanisms
4 Processes and Threads
5 Memory Management
1 Background and Introduction
History

• 1988: Gates approaches David Cutler (DEC)
• 1993: WinNT Launched
• 2000: Windows 2000 (previously “NT5.0”)
• 2001 Windows XP (build 2600)
• 2003 Windows Server 2003
Requirements and Design Goals

• Provide a true 32-bit, preemptive, reentrant, virtual memory operating system
• Run on multiple hardware architectures and platforms
• Run and scale well on symmetric multiprocessing systems
• Be a great distributed computing platform (Client & Server)
• Run most existing 16-bit MS-DOS and Microsoft Windows 3.1 applications
• Meet government requirements for POSIX 1003.1 compliance
• Meet government and industry requirements for operating system security
• Be easily adaptable to the global market by supporting Unicode
Goals (contd.)

• **Extensibility**
  – Code must be able to grow and change as market requirements change.

• **Portability**
  – The system must be able to run on multiple hardware architectures and must be able to move with relative ease to new ones as market demands dictate.

• **Reliability and Robustness**
  – Protection against internal malfunction and external tampering.
  – Applications should not be able to harm the OS or other running applications.

• **Compatibility**
  – User interface and APIs should be compatible with older versions of Windows as well as older operating systems such as MS-DOS.
  – It should also interoperate well with UNIX, OS/2, and NetWare.

• **Performance**
  – Within the constraints of the other design goals, the system should be as fast and responsive as possible on each hardware platform.
Services, Functions, and Routines

• Win32 API functions:
  – Documented, callable subroutines
  – `CreateProcess`, `CreateFile`, `GetMessage`

• Windows system services (executive system services):
  – Undocumented functions, callable from user space
  – `NtCreateProcess` is used by `CreateProcess` as an internal service

• Windows internal routines:
  – Subroutines inside the Windows executive, kernel, or HAL
  – Callable from kernel mode only (device driver, NT OS components)
  – `ExAllocatePool` allocates memory on Windows system heap
Services, Functions, and Routines (contd.)

• Windows services:
  – Processes which are started by the Service Control Manager
  – Example: The Schedule service supports the at-command

• DLL (dynamic link library)
  – Subroutines in binary format contained in dynamically loadable files
  – Examples:
    MSVCRT.DLL – MS Visual C++ run-time library
    KERNEL32.DLL – one of the Win32 API libraries
Processes and Threads

- Program – sequence of instructions
- Process – container for threads executing a program
- A Windows process is described by:
  - Executable program (code + data)
  - Private virtual address space
  - System resources (semaphores, communication ports, files)
  - Unique identifier – process ID (intern: client ID)
  - At least one thread

- Job (introduced with Windows 2000)
  - Collection of processes that share a set of quotas, limits, and security settings
Processes and Threads (contd.)

- Thread is the unit of *scheduling* in Windows
  - Multiple threads may share the address space of a container process.

- A *thread* is described by:
  - Register content (processor state)
  - Two stacks (user mode/kernel mode)
  - Private memory address space used by
    - Subsystems,
    - Runtime library,
    - DLLs
  - Unique identifier – thread ID (internally: client ID)
    - Process IDs and thread IDs don’t overlap

- Thread context is architecture-specific
  - See GetThreadContext() from Win32 API
Virtual Memory

- 32-bit address space (4 GB)
- 2 GB user space (per process)
- 2 GB operating system
- Memory manager maps virtual onto physical memory
Kernel Mode vs. User Mode

• No protection for components running in kernel mode
• Transition from user mode to kernel mode through special instruction (processor changes privilege level)
  – OS traps this instruction and validates arguments to syscalls
  – Transition from user to kernel mode does not affect thread scheduling
• Performance Counters: System/Processor/Process/Thread – Privileged Time/User time
• Performance Monitor – perfmon.exe
Objects and Handles

• Process, thread, file, event objects in Win32 - are mapped on NT executive objects
• Object services read/write object attributes
• Objects:
  – Human-readable names for system resources
  – Resource sharing among processes
  – Resource protection against unauthorized access
• Security/Protection based on NT executive objects
• 2 forms of access control:
  – Discretionary control: read/write/access rights
  – Privileged access: administrator may take ownership of files
Security

- Windows 2000 supports C2-level security (DoD 5200.23-STD, December 1985)
  - Discretionary protection (need-to-know) for shareable system objects (files, directories, processes, threads)
  - Security auditing (accountability of subjects and their actions)
  - Password authentication at logon
  - Prevention of access to un-initialized resources (memory, disk space)
- Windows NT 3.51 was formally evaluated for C2
- Windows NT 4.0 SP 6a passed C2 in December 1999
  - Networked workstation configuration
- European IT Security Criteria FC2/E3 security level
Registry

- System wide software settings: boot & configuration info
- Security database
- Per-user profile settings
- In-memory volatile data (current hardware state)
  - What devices are loaded?
  - Resources used by devices
  - Performance counters are accessed through registry functions
    - HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control
    - HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services
    - HKEY_LOCAL_MACHINE\Software
- Regedt32.exe is the tool to view/modify registry settings
Unicode

• Most internal text strings are stored/processed as 16-bit wide Unicode strings

• Win32 string functions have 2 versions
  – Unicode (wide) version
    • L“This string uses 16-bit characters“
  – ANSI(narrow) version
    • “This string uses 8-bit characters“
  – Generic character representation in Win32
    • _T (“This string uses generic characters“)

(Windows 95/98/ME have Win32 but no Unicode characters, Windows CE has Win32 but only Unicode characters)
File System

• Multiple file system formats
  – CDFS, UDF, FATx, NTFS

• Local/Network FSD

• Installable File System (IFS) Kit

• NTFS Advanced Features
  – Multiple Data Streams, Unicode-based names, General indexing facility,
  Dynamic bad-cluster remapping, Hardlinks & Junctions, Compression and spare Files, Change Logging, Per-user volume quota

• WinFS (future)
Symmetric Multiprocessing (SMP)
Sources of Information

- Windows NT Resource Kits (www.reskit.com)
- Platform SDK and Windows NT DDK
  - MSDN Development Platform
- Knowledge Base at www.microsoft.com
- TechNet CD-ROM edition
- Free Builds and Checked Builds
  - Kernel Debuggers
  - I386KD.EXE (command line)
  - WINDBG.EXE (GUI) with platform SDK
## Tools for Viewing Windows Internals

<table>
<thead>
<tr>
<th>Tool</th>
<th>Executable</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance Monitor</td>
<td>PerfMon</td>
<td>Windows 2000</td>
</tr>
<tr>
<td>Registry Editor</td>
<td>RegEdt32</td>
<td>Windows 2000</td>
</tr>
<tr>
<td>Windows 2000 Diagnostics</td>
<td>WinMSD</td>
<td>Windows 2000</td>
</tr>
<tr>
<td>Kernel Debugger</td>
<td>i386kd, KD, WINDBG</td>
<td>Platform SDK, Windows 2000 DDK</td>
</tr>
<tr>
<td>Pool Monitor</td>
<td>poolmon</td>
<td>Windows 2000 CD \Support\Tools</td>
</tr>
<tr>
<td>Global Flags</td>
<td>gflags</td>
<td>Windows 2000 CD \Support\Tools</td>
</tr>
<tr>
<td>Open Handles</td>
<td>oh</td>
<td>Windows 2000 Resource Kits</td>
</tr>
<tr>
<td>QuickSlice</td>
<td>qslice</td>
<td>Windows 2000 Resource Kits</td>
</tr>
<tr>
<td>Process Viewer</td>
<td>pviewer,</td>
<td>Windows 2000 CD \Support\Tools</td>
</tr>
<tr>
<td>Process Explode</td>
<td>pview</td>
<td>Platform SDK</td>
</tr>
<tr>
<td>Pool Monitor</td>
<td>poolmon</td>
<td>Windows 2000 CD \Support\Tools, DDK</td>
</tr>
<tr>
<td>Object Viewer</td>
<td>WinObj</td>
<td>Platform SD, <a href="http://www.sysinternals.com">www.sysinternals.com</a></td>
</tr>
<tr>
<td>Page Fault Monitor</td>
<td>PFMon</td>
<td>Windows 2000 Resource Kits, Platform SDK</td>
</tr>
<tr>
<td>Service Control Tool</td>
<td>sc</td>
<td>Windows 2000 Resource Kits</td>
</tr>
<tr>
<td>Task (Process) List</td>
<td>tlist</td>
<td>Windows 2000 CD \Support\Tools</td>
</tr>
</tbody>
</table>
www.sysinternals.com

- Windows NT internals articles and tools
  - Many generated using reverse engineering; e.g., no source access

- Some examples:
  - Handlex - show open handles and DLLs by process
  - Lstdlls - show DLLs loaded in each process
  - Diskmon/Filemon - log all file I/O operations
  - Regmon - log all registry accesses
  - Winobj - view object manager namespace and objects

- Caveat: Most include a device driver, hence you’re added “trusted code”
2 System Architecture
Windows Architecture (simplified)
Hardware Abstraction Layer

Loadable kernel module (HAL.DLL)

- Low-level interface to NT hardware platform
- Hides I/O interface, interrupt controllers, MP comm.
  - Architecture-specific, machine-dependent details
- Device driver call HAL routines for platform-dep. Info
- Only one HAL.DLL is installed
  - Many HAL*.DLL on distribution media
Kernel

Most fundamental operations in NT
• Thread scheduling and dispatching
• Trap handling and exception dispatching
• Interrupt handling and dispatching
• Multiprocessor synchronization
• Base kernel objects for executive
• Never paged out of memory
• Never preempted
• Small, compact, portable, efficient: C, assembly lang.
  – no probes for parameter accessibility
  – Some functions documented in DDK (Ke...)
Kernel objects

• Little overhead, small, efficient
• Control objects:
  – Kernel process object
  – Asynchronous procedure call object
  – Deferred procedure call object
  – Interrupt object
• Dispatcher objects
  – Synchronization objects
  – Kernel thread, mutex (mutant), kernel event pair, semaphore, timer, wait able timer
• Kernel supports set of interfaces that are portable and semantically identical across architectures
Executive

Upper layer of NTOSKRNL.EXE
(kernel: lower layer)

Contains:

• Exported func., callable through NTDLL.DLL, Win32...
• Exported func., not currently available through subsystem
  – LPCs, query functions: NtQueryInformationxxx
  – Specialized functions: NtCreatePagingFile
• Doc. functions callable from kernel mode, NT DDK
• Internal support routines
Executive components

- Process and thread manager
- Virtual memory manager
- Security reference monitor: protection/auditing
- I/O system: device independent I/O
- Configuration manager
- Plug and Play manager
- Power manager
- Cache manager: uses mem.manag. - mapped files
- Object manager: processes, threads, synch. objects
- LPC facility: flexible, optimized version of DCE RPC
- Run-time library: math, string, data types
- WMI routines
Device Drivers

• Loadable kernel modules
• Don‘t manipulate hardware, but call parts of HAL
  – Written in C/C++ typically
  – Source code portable across CPU architectures

Types:
• Hardware device drivers: implement device/network I/O
• File system drivers: file I/O <-> device I/O
• Filter drivers: disk mirroring, encryption
• Network redirectors and servers: send/receive remote I/O requests
List Drivers

- Control Panel -> Devices: installed drivers
- DRIVERS.EXE / pstat: loaded drivers

D:\home> drivers

<table>
<thead>
<tr>
<th>ModuleName</th>
<th>Code</th>
<th>Data</th>
<th>Bss</th>
<th>Paged</th>
<th>Init</th>
<th>LinkDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>ntoskrnl.exe</td>
<td>270272</td>
<td>40064</td>
<td>0</td>
<td>434816</td>
<td>82880</td>
<td>Sun May 11 ...</td>
</tr>
<tr>
<td>hal.dll</td>
<td>20384</td>
<td>2720</td>
<td>0</td>
<td>9344</td>
<td>11936</td>
<td>Mon Mar 10 ...</td>
</tr>
<tr>
<td>atapi.sys</td>
<td>22368</td>
<td>1088</td>
<td>0</td>
<td></td>
<td>768</td>
<td>Sat Apr 04 ...</td>
</tr>
<tr>
<td>SCSIPORT.SYS</td>
<td>9792</td>
<td>32</td>
<td>0</td>
<td>15840</td>
<td>2208</td>
<td>Sat Apr 04 ...</td>
</tr>
<tr>
<td>CPQSPM.sys</td>
<td>4896</td>
<td>64</td>
<td>0</td>
<td>544</td>
<td>544</td>
<td>Thu Feb 05 ...</td>
</tr>
<tr>
<td>Disk.sys</td>
<td>3328</td>
<td>0</td>
<td>0</td>
<td>7072</td>
<td>1600</td>
<td>Fri Apr 25 ...</td>
</tr>
<tr>
<td>CLASS2.SYS</td>
<td>7040</td>
<td>0</td>
<td>0</td>
<td>1632</td>
<td>1152</td>
<td>Fri Apr 25 ...</td>
</tr>
<tr>
<td>ScsiPwr.sys</td>
<td>8576</td>
<td>1248</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Mon Sep 09 ...</td>
</tr>
<tr>
<td>Ntfs.sys</td>
<td>68160</td>
<td>5408</td>
<td>0</td>
<td>269632</td>
<td>8704</td>
<td>Fri Apr 18 ...</td>
</tr>
<tr>
<td>Floppy.SYS</td>
<td>1088</td>
<td>672</td>
<td>0</td>
<td>7968</td>
<td>6112</td>
<td>Wed Jul 17 ...</td>
</tr>
<tr>
<td>Cdrom.SYS</td>
<td>12608</td>
<td>32</td>
<td>0</td>
<td>3072</td>
<td>3104</td>
<td>Wed Jul 17 ...</td>
</tr>
<tr>
<td>Fs_Rec.SYS</td>
<td>64</td>
<td>0</td>
<td>0</td>
<td>2912</td>
<td>1152</td>
<td>Mon Mar 10 ...</td>
</tr>
<tr>
<td>Null.SYS</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>288</td>
<td>416</td>
<td>Wed Jul 17 ...</td>
</tr>
<tr>
<td>KSecDD.SYS</td>
<td>1280</td>
<td>224</td>
<td>0</td>
<td>3456</td>
<td>1024</td>
<td>Thu Jul 18 ...</td>
</tr>
<tr>
<td>Beep.SYS</td>
<td>1184</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>704</td>
<td>Wed Apr 23 ...</td>
</tr>
</tbody>
</table>
Key System Components

System service dispatcher

Kernel

Hardware abstraction layer (HAL)

Hardware interfaces
(Buses, I/O devices, interrupts, interval timers, DMA, memory cache control, and so on)
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
  – For details, see DDK documentation on PsCreateSystemThread()

• What process do they appear in?
  – windowing system threads appear in “csrss.exe” (Win32 subsystem process) - rest in “System” (PID 4)
Examples Of System Threads

• Core operating system (NTOSKRNL.EXE)
  – Modified Page Writer
  – Balance Set Manager
  – Swapper (kernel stack, working sets)
  – Cache manager lazy writer
  – Zero page thread (thread 0, priority 0)
  – General pool of worker threads
    (ExQueueWorkItem())

• File server (SRV.SYS)

• Floppy driver (FLOPPY.SYS)
NTDLL.DLL

Support library for use of subsystem DLLs:

- System service dispatch stubs to NT executive system services
  - NtCreateFile, NtSetEvent
  - More than 200
  - Most of them are accessible through Win32 Stubs call service-dispatcher/kernel-mode service in NTOSKRNL.EXE

- Support functions used by subsystems
  - Image loader (Ldr...)
  - Heap manager
  - Win32 subsyst. Comm. func. (Csr...)
  - Runtime library func. (Rtl...)
  - User-mode asynch. procedure call (APC) dispatcher, exception disp.
System Processes

- Idle Process
- System process (knl mode system threads)
- Session Manager (SMSS.EXE)
  - First user-mode proc, completes system initialisation
- Win32 subsystem (CSRSS.EXE)
- Logon (WINLOGIN.EXE)
  - Handles logon, calls UNSERINIT.EXE to create user proc
- LSA (LSASS.EXE)
  - Validates authentication data and creates access token
- Service controller (SERVICES.EXE)
  - Starts and stops NT services (e.g. event log)
**System Process Tree**

- **smss.exe**  
  Session Manager  
  The first “created” process; takes parameters from  
  \Registry\Machine\System\CurrentControlSet\Control\  
  Session Manager  
  Launches required subsystems (csrss) and then winlogon

- **csrss.exe**  
  Win32 subsystem

- **winlogon.exe**  
  Logon process: Launches services.exe, lsass.exe, and nddeagnt.exe;  
  presents first login prompt;  
  presents “enter username and password” dialog  
  When someone logs in, launches userinit.exe

- **services.exe**  
  Service Controller; also, home for many Windows NT-supplied services  
  Starts processes for services not part of services.exe (driven by  
  \Registry\Machine\System\CurrentControlSet\Services )

- **lsass.exe**  
  Local Security Authentication Server

- **userinit.exe**  
  Started after logon; starts desktop (Explorer.exe) and exits (hence does not show up in tlist output; Explorer appears to be an orphan)

- **explorer.exe**  
  and its children are the creators of all interactive apps
Win32 Subsystem

• Environment subsystem process (CSRSS.EXE):
  – Console (text) windows
  – Creating and deleting processes and threads
  – Portions of the support for 16-bit virtual DOS machine (VDM)
  – Other func: GetTempFile, DefineDosDevice, ExitWindowsEx

• kernel-mode device driver (WIN32K.SYS):
  – Window manager: manages screen output;
  – input from keyboard, mouse, and other devices
  – user messages to applications.
  – Graphical Device Interface (GDI)
Win32 Subsystem (contd.)

• Subsystem DLLs (such as USER32.DLL, ADVAPI32.DLL, GDI32.DLL, and KERNEL32.DLL)
  – Translate Win32 API functions into calls to NTOSKRNL.EXE and WIN32K.SYS.

• Graphics device drivers
  – graphics display drivers, printer drivers, video miniport drivers
Key Windows System Files

HAL.DLL Hardware abstraction layer
NTOSKRNL.EXE** Executive and kernel
NTOSKRNL.EXE Executive and kernel with Physical Address Extension (PAE) support; up to 64 GByte Memory
WIN32K.SYS Win32 USER and GDI kernel-mode components
KERNEL32.DLL, ADVAPI32.dll, USER32.DLL, GDI32.DLL. Win32 subsystem DLLs
SERVICES.EXE Service controller process
WINLOGON.EXE Logon process
SMSS.EXE Session manager process
CSRSS.EXE* Win32 subsystem process
NTDLL.DLL Internal support functions and system service dispatch stubs to executive functions
Accounting for Kernel-Mode Time

“Processor Time” =
  total busy time of processor
  (equal to elapsed real time - idle time)

“Processor Time” =
  “User Time” + “Privileged Time”

“Privileged Time” =
  time spent in kernel mode

“Privileged Time” includes:
  – Interrupt Time
  – DPC Time

Note:
  Interrupts and DPCs are not charged to any process or thread
3 System Mechanisms
Interrupt Dispatching

user or kernel mode code

interrupt!

kernel mode

Interrupt dispatch routine

Disable interrupts

Record machine state (trap frame) to allow resume

Mask equal- and lower-IRQL interrupts

Find and call appropriate ISR

Dismiss interrupt

Restore machine state (including mode and enabled interrupts)

Note, no thread or process context switch!

Interrupt service routine

Tell the device to stop interrupting

Interrogate device state, start next operation on device, etc.

Request a DPC

Return to caller
Interrupt Precedence via IRQLs

- **IRQL =** Interrupt Request Level
  - The “precedence” of the interrupt with respect to other interrupts
  - Different interrupt sources have different IRQLs
  - Not the same as IRQ

- IRQL is also a state of the processor
- Servicing an interrupt raises processor IRQL to that interrupt’s IRQL
  - This masks subsequent interrupts at equal and lower IRQLs
- User mode is limited to IRQL 0

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    | High |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    | Power fail |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    | Interprocessor Interrupt |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    | Clock |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    | Device n |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    | . |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    | . |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    | . |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    | Device 1 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    | Dispatch/DPC |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    | APC |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    | Low |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
Getting Into Kernel Mode

Code is run in kernel mode for one of three reasons:

1. Requests from user mode
   - Via the system service dispatch mechanism
   - Kernel-mode code runs in the context of the requesting thread

2. Interrupts from external devices
   - Windows interrupt dispatcher invokes the interrupt service routine
   - ISR runs in the context of the interrupted thread
     (so-called “arbitrary thread context”)
   - ISR often requests the execution of a “DPC routine,”
     which also runs in kernel mode
   - Time not charged to interrupted thread

3. Dedicated kernel-mode system threads
   - Some threads in the system stay in kernel mode at all times
     (mostly in the “System” process)
   - Scheduled, preempted, etc., like any other threads
Windows - SysCall

- System Call Dispatching:
  - the user program invokes a Win32 API function, with arguments
  - the request is dispatched to the corresponding .dll (kernel32.dll, …)
  - kernel32.dll does argument checking/adaptation and invokes the NT native API function (e.g. NtCreateFile,...) located in ntdll.dll
  - ntdll.dll will issue the syscall (int 2e) passing
    - the syscall number in a register (eax)
    - the address of the arguments on the current stack in a register (edx)
- The int 2e is dispatched via the Interrupt Dispatch Table
- KiSystemService
  - ENTER_SYSCALL macro completes the trap_frame on the stack
  - the syscall number is validated against KiServiceLimit (258 in Windows 2000)
  - the required arguments are probed for access (Ke) and copied to kernel stack
  - the syscall # is used as an index in the KiServiceTable
  - the NtService is called (call ebx)
  - last parameter checking/probe is done here (service specific)
  - … the NtService is performed …
  - return path is taken (synchronous / asynchronous)
IPC abstractions

• Deferred Procedure Call (DPC)
  – Used within kernel
  – Execute in context of arbitrary (current) thread
  – Schedule a knl routine to run soon/when convenient

• Asynchronous Procedure Call (APC)
  – Runs within user space
  – Executes in context of a specific thread (& process)
  – One APC queue per thread

• Exception Handlers
  – Stack of, declared to handle e.g. address error, debug breakpoint, divide by zero
  – Support user-level exception ports and debugger ports
Object Manager

- Uniform approach towards:
  - Object protection (C2 security)
  - Charging
  - Object naming
  - Object retention and garbage collection
  - Object access (via handles)
  - Standard object attributes
  - Standard object methods
Object types - examples

- Process
- Memory Section
- Event
- Timer
- Object directory
- Queues

- Thread
- File
- Semaphore
- Symbolic link
- Port
- (registry) key
Kernel concurrency

• Disabling interrupts insufficient on multiprocessor
• Another CPU may have interrupts enabled. So:
  – Raise IRQL to that of highest IR handler using x
  – Spinlock to prevent interrupts from other CPUs
  – Contrast to conventional UNIX critical section
• Only possible for code that is:
  – Non-paged
  – Doesn’t ref paged data
  – Doesn’t generate interrupts of exceptions
Local Procedure Call (LPC)

• “Internal” IPC between address spaces, e.g.
  – Transport for local RPC
  – Calls to Win32 subsystem

• Variants:
  – \( n \leq 256 \) bytes done inband
  – \( n > 256 \) bytes via shmem section
  – \( n > \) shmem section – addr-to-addr copy

• Typically client-server
  – Client connects to servers well-known LPC port
  – Server opens new port and tells client its address
4 Processes and threads
A Process and its Resources

- Process object
  - Access token
  - Handle table
  - Virtual address space descriptors (VADs)
    - VAD
    - VAD
    - VAD
  - Object
  - Object
  - Object
  - thread
  - thread
  - thread
    - Access token
    - ...
Data Structures for each process/thread

- Executive process block (EPROCESS)
- Executive thread block (ETHREAD)
- Win32 process block
- Process environment block
- Thread environment block
Structure of Executive Process Block

- Kernel process block (or PCB)
  - Process ID
  - Parent process ID
  - Exit Status
  - Create and exit times
  - Next process block
  - Quota block
  - Memory management information
    - Exception port
    - Debugger port
  - Process environment block
    - Image filename
    - Image base address
    - Process priority class

PsActiveProcessHead → EPROCESS → Primary access token → Handle table → Win32 process block
# Key substructure: Kernel process block

Sometimes called PCB – process control block

<table>
<thead>
<tr>
<th>Dispatcher header</th>
<th>Process page directory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel time</td>
<td></td>
</tr>
<tr>
<td>User time</td>
<td></td>
</tr>
<tr>
<td>Inswap/Outswap list entry</td>
<td></td>
</tr>
<tr>
<td></td>
<td>KTHREAD</td>
</tr>
<tr>
<td>Process spinlock</td>
<td></td>
</tr>
<tr>
<td>Processor affinity</td>
<td></td>
</tr>
<tr>
<td>Resident kernel stack count</td>
<td></td>
</tr>
<tr>
<td>Process base priority</td>
<td></td>
</tr>
<tr>
<td>Default thread quantum</td>
<td></td>
</tr>
<tr>
<td>Process state</td>
<td></td>
</tr>
<tr>
<td>Thread seed</td>
<td></td>
</tr>
<tr>
<td>Disable boost flag</td>
<td></td>
</tr>
</tbody>
</table>
# 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>
The main Stages NT follows to create a process

Creating process

- Open EXE and create selection object
  - Create NT process object
    - Create NT thread object
      - Notify Win32 subsystem
        - Set up for new process and thread
          - Start execution of the initial thread
            - Return to caller

Win32 subsystem

New process

- Final process/image initialization
  - Start execution at entry point to image
Opening the image to be executed

What kind of application is it?

- Run CMD.EXE
- Run NTVDM.EXE
- Use .EXE directly

MS-DOS .BAT or .CMD
Win16
OS/2 1.x
POSIX
MS-DOS .EXE, .COM, or .PIF
Run OS2.EXE
Run POSIX.EXE
Run NTVDM.EXE
Win32
Looking at the Process Hierarchy with TLIST -T

- **Understanding the parent of a process helps identify what it is and where it came from**
- **tlist -t shows the tree**
  - If parent not alive, left justifies process
    - I.e., cannot see creator if it is gone
  - For example, explorer.exe’s parent is dead (it is actually started by userinit.exe, which then exits)
- **Windows**
  - Perfmon can show parent process id
  - Task Manager has a “kill process tree”
Idle Thread

- NT dispatches idle thread when no runnable thread exists on a CPU
- Idle thread has no priority (reported as 0); runs at IRQL 2
- Control flow of idle thread:
  - Enable/disable interrupts (allow pending ints to be delivered)
  - Checks, whether DPCs are pending
  - Checks, whether thread has been selected to run next on CPU; if so: dispatches that thread
  - Calls HAL idle routine (to perform power management)
- Varying names: System Idle Process (TaskManager), Idle (Pview/Pviewer), Idle Process (Pstat), System Process (Tlist/Qslice)
Task Manager

- Processes tab: List of processes
- Can configure with View-> Select columns
- Click on column heading to sort by that column
- Right-click on a process name to change priority, end process tree (new in Windows 2000), or (on MP) CPU assignments
- Performance tab: Subset of Windows NT performance counters

- To start: Ctrl+Shift+Esc; or Ctrl+Alt+Del; or right click on empty area of task bar
- Overlaps with other process display utilities
  - Except Win16 process info, only visible here (On Processes tab, click on Options->Show 16-bit tasks)
- Applications tab: List of top level visible windows
  - Windows are owned by threads (right-click on a window and select “go to process”)
Process Viewer

- Pviewer.exe in Resource Kit, pview.exe in Platform SDK
  - Shows start address of each thread
    - Needed to analyze system threads
- Can display remote process list
  - But cannot kill remote processes
    - Use rkill in ResKit!
Looking at open Handles

- Handle leaks can show up as system memory leaks!
- Task Manager can show total # handles by process
- Resource Kit “oh” tool (first time run will set an Windows Global Flag - see gflags.exe in ResKit; reboot required)

```
C:\>oh /?
Usage: oh [-p n] [-t typeName] [-a] [name]
where: -p n - displays only open handles for process with ClientId of n
      -t typeName - displays only open object names of specified type.
      -a - includes objects with no name.
      name - displays only handles that contain the specified name.
```

```
C:\>oh -p 274 -t file
274 POWERPNT.EXE File  0014 \Slides\ntint
274 POWERPNT.EXE File  000c \Slides\ntint\int15e.ppt
274 POWERPNT.EXE File  010c \Program Files\Common Files\Microsoft Sha
```

- handleex (GUI) or nthandle (console) from www.sysinternals.com
DLL Usage - Static references

- Depends.exe in Resource Kit
- Displays static linkage from EXE to DLLs
DLL Usage - Actual files

- To diagnose DLL conflicts, you need to know which DLLs were loaded and from where.
- `tlist <processname>` or `tlist <processid>` lists the DLLs, but not the path.
- `listdlls` from www.sysinternals.com lists full path.

```
> listdlls -p outlook

ListDLLE U2.0
Copyright (C) 1997 Mark Russinovich
http://www.sysinternals.com

---------------------------------------------
<table>
<thead>
<tr>
<th>Base</th>
<th>Size</th>
<th>Version</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x38000000</td>
<td>0x8000</td>
<td>8.05.5104.0000</td>
<td>C:\Program Files\Microsoft Office\Office</td>
</tr>
<tr>
<td>0x77f80000</td>
<td>0x5e00</td>
<td>4.00.1381.0000</td>
<td>C:\WINNT\System32\ntdll.dll</td>
</tr>
<tr>
<td>0x6c3f0000</td>
<td>0x6500</td>
<td>8.05.5104.0007</td>
<td>C:\Program Files\Microsoft Office\Office</td>
</tr>
<tr>
<td>0x78000000</td>
<td>0x3d00</td>
<td>6.00.8267.0000</td>
<td>C:\WINNT\system32\MSVCR71.dll</td>
</tr>
<tr>
<td>0x77f80000</td>
<td>0x5e00</td>
<td>4.00.1381.0000</td>
<td>C:\WINNT\system32\KERNEL32.dll</td>
</tr>
<tr>
<td>0x77b20000</td>
<td>0x2800</td>
<td>4.00.1381.0004</td>
<td>C:\WINNT\system32\ole32.dll</td>
</tr>
<tr>
<td>0x7e1d0000</td>
<td>0x5200</td>
<td>4.00.1381.0004</td>
<td>C:\WINNT\system32\RPCRT4.dll</td>
</tr>
<tr>
<td>0x77d00000</td>
<td>0x3a00</td>
<td>4.00.1381.0004</td>
<td>C:\WINNT\system32\ADVAPI32.dll</td>
</tr>
<tr>
<td>0x77f80000</td>
<td>0x5e00</td>
<td>4.00.1381.0000</td>
<td>C:\WINNT\system32\BASEEFS.dll</td>
</tr>
<tr>
<td>0x77e40000</td>
<td>0x2c00</td>
<td>4.00.1381.0000</td>
<td>C:\WINNT\system32\GDI32.dll</td>
</tr>
<tr>
<td>0x70300000</td>
<td>0x7300</td>
<td>4.72.3110.0001</td>
<td>C:\WINNT\system32\COMCTL32.dll</td>
</tr>
<tr>
<td>0x30c00000</td>
<td>0x5200</td>
<td>8.00.0000.4228</td>
<td>C:\Program Files\Microsoft Office\Office</td>
</tr>
<tr>
<td>0x6fa90000</td>
<td>0xda00</td>
<td>5.05.2174.0000</td>
<td>C:\WINNT\System32\MAPI32.DLL</td>
</tr>
</tbody>
</table>
```
Windows Thread Scheduling

- Priority-driven, preemptive scheduling system
- Highest-priority runnable thread always runs
- Restricted by processor affinity
- Thread runs for time amount of quantum
- No single scheduler – event-based scheduling code spread across the kernel

Dispatcher routines triggered by the following events:
- Thread becomes ready for execution
- Thread leaves running state (quantum expires, wait state)
- Thread’s priority changes (system call/NT activity)
- Processor affinity of a running thread changes
Windows priority levels

- 16-32: static priorities (real-time)
- 1-15: variable priorities
- 0: MemManager: zero-page thread
- 0: Idle: „lower than 0“, no thread
- Idle Threads: one per CPU (in proc with PID 0)
Win32 priorities

- Thread priority is based on combination of process priority class and relative thread priority
- Row „normal“ is base priority for the priority classes
- Base priority can be changed (SetProcessPriority())
  - Default base priority is „normal“ (24, 13, 8, 4)
  - NT system proc. have higher base prio. than default for normal class (session manager, service controller, local authentication server)
- Thread priority is adjusted by Windows (variable levels)
- Real-time priorities are never adjusted
Interrupt levels vs. Priority levels

- All threads run at IRQL 0 or 1
- Threads normally run at IRQL 0
- Only kernel mode APCs (asynch. proc. calls) run at IRQL 1
- No thread ever blocks hardware interrupts
- Thread scheduling at IRQL 2 (dispatching)
- Spinlock synchronizes access to scheduling data on MP system (*KiDispatcherLock*)

<table>
<thead>
<tr>
<th>IRQLs</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31</td>
</tr>
<tr>
<td>High</td>
<td></td>
</tr>
<tr>
<td>Power fail</td>
<td></td>
</tr>
<tr>
<td>Interprocessor interrupt</td>
<td></td>
</tr>
<tr>
<td>Clock</td>
<td></td>
</tr>
<tr>
<td>Device n</td>
<td></td>
</tr>
<tr>
<td>Device 1</td>
<td></td>
</tr>
<tr>
<td>Dispatch/DPC</td>
<td></td>
</tr>
<tr>
<td>APC</td>
<td></td>
</tr>
<tr>
<td>Passive_Level</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HW int</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SW int</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Scheduling rules (1)

• Preemptive, priority driven
  – FIFO ready queues per priority level
  – A ready thread either runs or is inserted at end of queue
  – ...a preempted thread is inserted at head of queue

• Time-sliced round-robin per priority level
  – Threads have base priority & varying current priority
  – Process has only base priority (starting prio for threads in this process)

• On multiprocessor systems:
  – Tries to keep thread on same CPU
  – Lowest priority thread is preempted
  – Any processor can interrupt another processor to schedule a thread
Scheduling rules (2)

• Voluntary switch:
  – Enters wait state
  – SwitchToThread (NtYield)
  – Explicit priority decrease
    (SetThreadPriority – NtSetInformationThread)
  – Action: the next_ready thread is run, previous goes to right/state list

• Preemption: (fixed prio vs. variable prio)
  – A higher priority thread becomes ready
  – Action: lowest priority thread is preempted (goes to head of queue)
  – Preemption is immediate (fixed prio) or at quantum end (variable prio)

• Quantum end:
  – Next_ready thread is run, previous has prio decrease by 1
    (no lower than base priority)
Scheduling rules (3)

• Quantum duration:
  – Workstation (Professional): 2 clock ticks
  – Server: 12 clock ticks
  – Clock tick: 10 ms (7.5-15 ms)

• Quantum stretching (favouring foreground applications)
  – Longer quantum: 2, 4, 6 clock ticks
  – HKLM\CCS\Control\PriorityControl\W32PrioritySeparation=0,1,2

• Priority boosting (keep the I/O system busy)
  – After a wait (e.g. I/O) is satisfied a priority boost is given (not over 15)
  – Default boost values:
    • 1 for disk, CD-ROM, parallel, video, semaphore
    • 2 for serial, network, named pipe, mailslot
    • 6 for keyboard or mouse
  – Amount can be specified by driver or executive
Scheduling Data Structures

Default base prio
Default proc affinity
Default quantum

Process

Thread

Base priority
Current priority
Processor affinity
Quantum

Ready summary

Idle summary

Bitmask for non-empty ready queues
Bitmask for idle CPUs
Scheduling Scenarios – voluntary switch

Thread gives up CPU (wait for event, mutex, semaphore, I/O completion port, process, thread, window message)

- Priority of relinquishing thread is not reduced
- Quantum value is decremented by 1 (when wait satisfied)
Scheduling Scenarios – preemption

- A higher-priority thread’s wait completes
- A thread priority is increased or decreased
- Preempted thread is put at head of ready queue

No priority boost for threads in real-time range
Scheduling Scenarios – Quantum end

Running thread exhausts CPU quantum:
• Should the thread’s priority be decremented?
• Should another thread be run on the processor?
Thread scheduling events

- On clock tick
- On Thread leaves wait state
- On Pre-emption (higher priority thread enters ready state)
- On quantum exhausted
- On new thread
- On Wait() called
- On new foreground window selected
- On I/O op completes
- On one second timer
On Clock Interrupt

/* decrement quantum of current thread */
t.quantum’ = t.quantum – 3;
if (t.quantum == 0)
    OnQuantumExhausted(t);

Events
On thread leaves wait state
if (t.priority > 15) {  // real-time priority
    t.quantum = p.default_quantum;  // reset
} else {
    t.quantum = t.quantum – 1;
    if (t.quantum == 0)
        t.quantum = p.quantum;
}
Put_at_end_of_ready_queue(t);
RunHighestPriorityThread();
Events
On Thread Pre-emption

/* retain advantage over threads at */
/* same priority level */
Put_at_front_of_ready_queue(t);
RunHighestPriorityThread();
On quantum exhausted

/* depends on how why thread priority was */
/* increased in the first place.           */
If (supposed_to_do_so(t)) &
    t.priority > t.base_priority {
    t.priority = t.priority – t.priority_dec;
}
RunHigestPriorityThread();

Events
On New Thread

/* default quantum from parent process */
t.quantum = p.default_quantum;
RunHighestPriorityThread();
On Wait()

/* ensure fast response when wait exits */
if (interactive_session & WaitingOnWindowsMessage(t)) {
    t.priority = 14;
}
RunHighestPriorityThread();

Events
On select New Foreground Window

/* NTW favours the foreground window */
if NTWKS &
    isNewForegroundWindows(p) &
    Win32Priority(p) >= "Normal"
{
    p.default_quantum = p.default_quantum * 3
}
RunHighestPriorityThread();

Events
On completion of an I/O op

/* and some other types of wait() event */
/* long wait on slow device -> high boost 8*/
/* short wait on fast device -> low boost */
i=ChosenByDeviceDriver() // 0<i<9
if (t.priority < 16) // leave real-time as is
t.priority = max(t.priority+i, 15);
/* see OnQuantumExhausted() */
t.priority_dec = 1;
/* i.e. decrements towards p.base_priority */
RunHighestPriorityThread();
Events
On OneSecondTimer
/* how NT avoids priority inversion */
/* i.e. hi-priority waiting on lo-priority */
/* where mid-priority consuming all CPU */
For all ready threads, in round-robin order:
  if SatPatientlyFor(t, 300msecs) {
    t.priority = 15; // pretty high
    t.quantum = t.quantum * 2; // just this once
  }
RunHighestPriorityThread();
Events
Context Switching

Thread‘s context and context switching are arch-specific

- Context switch requires saving/loading of these data:
  - Program counter
  - Processor status register
  - Other register contents
  - User and kernel stack pointers
  - Pointer to address space in which thread runs (page table directory)

- Kernel saves kernels stack pointer in KTHREAD block and loads new thread‘s kernel stack address
- Loads new thread‘s context & page tab.dir.; flushes TLB
- Pending kernel APCs are delivered (IRQL 1)
- Control passes to new thread‘s PC; execution resumes

Kernel saves this info by pushing it on kernel stack
Multiprocessor considerations

- Try to preserve on-chip cache on multiprocessors
  - PIII runs at, say, 500MHz
  - Memory runs at, say, 50MHz
  - So try to avoid memory hits and cache flush by keeping threads on the same processor
- Each thread has CPU affinity mask (can run on..)
- Each thread has “preferred” and “next” CPU
- NT looks for a ready thread to run on CPU $n$ that:
  - Last ran on CPU $n$, or
  - Specifies CPU $n$ as its preferred CPU, or
  - Has a priority > 24
- Note: Windows will not shuffle threads around CPUs in order to free up a CPU $m$ for any specific thread
Adjusting Thread Scheduling

• Quantum stretching for threads in foreground process
  – New to Windows NT 4.0
  – Before: increase base priority for foreground threads;
    problem: starvation of background processes

• Boosting priority upon wait completion
  – Suggested values in \ddk\include\ntddk.h
  – Event: 1, Disk/CD: 1, Network: 2, Keyboard/Mouse: 6, Sound: 8
  – Priority drops slowly, one level per quantum

• Boosting priority for threads entering a wait state
  – CSRSS boosts GUI thread waiting for windows message to 14
  – Priority drops immediately; thread runs for double quantum at
    high prio

• Boosting priority for threads not getting CPU time
Priority boosting and decay

Lower priority thread blocks high priority thread
- Balance set manager scans ready queue for threads that have not run for longer than 300 clock ticks (1/sec)
- Boosts prio to 15; double quantum; no more than 16 threads checked; no more than 10 threads boosted
Watching Priority boosts for CPU Starvation
Boosted thread can still be preempted
Relevant Tools

- View (and change) process base priority with:
  - TaskManager, Pview, Pviewer
- View numeric process base priority with:
  - PerfMon, pstat
- View thread priorities with
  - PerfMon, Pview, Pviewer, Pstat
- No general utility to change relative thread priority levels
- Need *increase scheduling priority* privilege
  - Important NT kernel threads run in real-time priority class
  - Be careful with threads spending excessive time in RT prio range
View thread state changes with perf mon
5 Memory Management
Windows Memory Manager

• Provides 4 GB flat virtual address space (32-bit addresses)
• Exports memory-mapped files
• Allows pages shared between processes
• Provides support for file system cache manager
• Windows 2000 enhancements:
  – Integrated support for Terminal Server
  – Ability to use up to 64 GB physical memory
  – Performance and scalability improvements
  – Driver verifier
4GB Virtual Address Space

- **2 GB per-process**
  - Address space of one process is not directly reachable from other processes

- **2 GB systemwide**
  - The operating system is loaded here, and appears in every process’s address space
  - There is no process for “the operating system” (though there are processes that do things for the OS, more or less in “background”)

**Top 4GB:**
- .EXE code
- .DLL code

**Bottom 4GB:**
- Exec, Kernel, HAL, drivers, per-thread kernel mode stacks
- Win32K.Sys

**Process piles:**
- Process heaps
- Per-thread user mode stacks
- File system cache
- Hyperspace
- Non-paged pool
- Paged pool

**Address spaces:**
- Unique per process, accessible in user or kernel mode
- 2 GB per-process
- System wide, accessible only in kernel mode
- 2 GB systemwide
- System wide, accessible only in kernel mode
- Unique per process, accessible in user or kernel mode
3GB Process Space Option

- Only available on x86 Windows 2000 Advanced Server
  - Boot with /3GB option in BOOT.INI
  - Chief “loser” in system space is file system cache
- Expands per-process address space
  - But image must be marked as “large address space aware”
- 16GB maximum physical memory
- A stopgap while we wait for 64-bit Windows 2000 (Itanium)
Address Translation - Mapping virtual addresses to physical memory

- Mapping via *page table entries*
- Indirect relationship between virtual pages and physical memory
Translating a virtual address:

1. Memory management HW locates page directory for current process (cr3 register on Intel)
2. Page directory index directs to requested page table
3. Page table index directs to requested virtual page
4. If page is valid, PTE contains physical page number (PFN – page frame number) of the virtual page
   • Memory manager fault handler locates invalid pages and tries to make them valid
   • Access violation/bug check if page cannot be brought in (prot. fault)
5. When PTE points to valid page, byte index is used to locate address of desired data
## Protecting Memory

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAGE_NOACCESS</td>
<td>Read/write/execute causes access violation</td>
</tr>
<tr>
<td>PAGE_READONLY</td>
<td>Write/execute causes access violation; read permitted</td>
</tr>
<tr>
<td>PAGE_READWRITE</td>
<td>Read/write accesses permitted</td>
</tr>
<tr>
<td>PAGE_EXECUTE</td>
<td>Any read/write causes access violation; execution of code is permitted (not implemented by x86 or Alpha)</td>
</tr>
<tr>
<td>PAGE_EXECUTE_READ</td>
<td>Read/execute access permitted (not implemented by x86 or Alpha)</td>
</tr>
<tr>
<td>PAGE_EXECUTE_READWRITE</td>
<td>All accesses permitted (not impl. by x86 or Alpha)</td>
</tr>
<tr>
<td>PAGE_WRITECOPY</td>
<td>Write access causes the system to give process a private copy of this page; attempts to execute code cause access violation</td>
</tr>
<tr>
<td>PAGE_EXECUTE_WRITECOPY</td>
<td>Write access causes creation of private copy of pg.</td>
</tr>
<tr>
<td>PAGE_GUARD</td>
<td>Any read/write attempt raises EXCEPTION_GUARD_PAGE and turns off guard page status</td>
</tr>
</tbody>
</table>
Page directories & Page tables

• Each process has a single page directory (phys. addr. in KPROCESS block, at 0xC0300000, in cr3 (x86))
  – cr3 is re-loaded on inter-process context switches
  – Page directory is composed of page directory entries (PDEs) which describe state/location of page tables for this process
    • Page tables are created on demand
  – x86: 1024 page tables describe 4GB

• Each process has a private set of page tables

• System has one set of page tables
  – System PTEs are a finite resource: computed at boot time
  – HKLM\System...\Control\SessionManager\SystemPages
System and process-private page tables

- On process creation, system space page directory entries point to existing system page tables.
- Not all processes have the same view of system space (after allocation of new page tables).
Page Table Entries

- Page tables are array of Page Table Entries (PTEs)
- Valid PTEs have two fields:
  - Page Frame Number (PFN)
  - Flags describing state and protection of the page

Reserved bits are used only when PTE is not valid

Page frame number | U | P | Cw | Gi | L | D | A | Cd | Wt | O | W | V
---|---|---|----|---|---|---|---|----|----|---|----|---
31 |   |   |    |   |   |   |   |    |    |   |    |   |
12 |   |   |    |   |   |   |   |    |    |   |    |   |
0  |   |   |    |   |   |   |   |    |    |   |    |   |
valid
## PTE Status and Protection Bits

*(Intel x86 only)*

<table>
<thead>
<tr>
<th>Name of Bit</th>
<th>Meaning on x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessed</td>
<td>Page has been read</td>
</tr>
<tr>
<td>Cache disabled</td>
<td>Disables caching for that page</td>
</tr>
<tr>
<td>Dirty</td>
<td>Page has been written to</td>
</tr>
</tbody>
</table>
| Global          | Translation applies to all processes  
|                 | (a translation buffer flush won’t affect this PTE)                           |
| Large page      | Indicates that PDE maps a 4MB page (used to map kernel)                      |
| Owner           | Indicates whether user-mode code can access the page of whether the page is limited to kernel mode access |
| Valid           | Indicates whether translation maps to page in phys. Mem.                    |
| Write through   | Disables caching of writes; immediate flush to disk                          |
| Write           | Uniproc: Indicates whether page is read/write or read-only;  
|                 | Multiproc: ind. whether page is writeable/write bit in res. bit              |
Page Fault Handling

• Reference to invalid page is called a page fault
• Kernel trap handler dispatches:
  – Memory manager fault handler (MmAccessFault) called
  – Runs in context of thread that incurred the fault
  – Attempts to resolve the fault or raises exception
• Page faults can be caused by variety of conditions
• Four basic kinds of invalid Page Table Entries (PTEs)
Reasons for access faults

• Accessing a page that is not resident in memory but on disk in page file/mapped file
  – Allocate memory and read page from disk into working set

• Accessing page that is on standby or modified list
  – Transition the page to process or system working set

• Accessing page that has no committed storage
  – Access violation

• Accessing kernel page from user-mode
  – Access violation

• Writing to a read-only page
  – Access violation
Reasons for access faults (contd.)

• Writing to a guard page
  – Guard page violation (if a reference to a user-mode stack, perform automatic stack expansion)

• Writing to a copy-on-write page
  – Make process-private copy of page and replace original in process or system working set

• Referencing a page in system space that is valid but not in the process page directory
  (if paged pool expanded after process directory was created)
  – Copy page directory entry from master system page directory structure and dismiss exception

• On a multiprocessor system: writing to valid page that has not yet been written to
  – Set dirty bit in PTE
Invalid PTEs and their structure

- **Page file**: desired page resides in paging file
  
in-page operation is initiated

- **Demand Zero**: pager looks at zero page list; if list is empty, pager takes list from standby list and zeros it;
  
PTE format as shown above, but page file number and offset are zeros
Invalid PTEs and their structure (contd.)

• **Transition**: the desired page is in memory on either the standby, modified, or modified-no-write list
  – Page is removed from the list and added to working set

• **Unknown**: the PTE is zero, or the page table does not yet exist
  - examine virtual address space descriptors (VADs) to see whether this virtual address has been reserved
  - Build page tables to represent newly committed space
Prototype PTEs

• Software structure to manage potentially shared pages
  – Array of prototype PTEs is created as part of section object (part of segment structure)
  – First access of a page mapped to a view of a section object: memory manager uses prototype PTE to fill in real PTE used for address translation;
  – Reference count for shared pages in PFN database

• Shared page valid:
  – process & prototype PTE point to physical page

• Page invalidated:
  – process PTE points to prototype PTE

• Prototype PTE describes 5 states for shared page:
  – Active/valid, Transition, Demand zero, Page file, Mapped file

• Layer between page table and page frame database
Prototype PTEs for shared pages – the bigger picture

- Two virtual pages in a mapped view
- First page is valid; 2nd page is invalid and in page file
  - Prototype PTE contains exact location
  - Process PTE points to prototype PTE
Translation Look-Aside Buffer (TLB)

- Address translation requires two lookups:
  - Find right table in page directory
  - Find right entry in page table

- Most CPU cache address translations
  - Array of associative memory: translation look-aside buffer (TLB)
  - TLB: virtual-to-physical page mappings of most recently used pages

<table>
<thead>
<tr>
<th>Virtual page #</th>
<th>Page frame</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>290</td>
<td>Valid</td>
</tr>
<tr>
<td>64</td>
<td>Invalid</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>1004</td>
<td>Valid</td>
</tr>
<tr>
<td>7</td>
<td>801</td>
<td>Invalid</td>
</tr>
<tr>
<td>65</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Simultaneous read and compare
Soft Versus Hard Page Faults

- Hard page faults involve a disk read
  - Some hard page faults are unavoidable
    - Code is brought into physical memory from .EXEs and .DLLs) via page faults
    - The file system cache reads data from cached files in response to page faults
- Soft page faults are satisfied in memory
  - A shared page that’s valid for one process can be faulted into other processes
  - Pages can be faulted back into a process from the standby and modified page list (described later)

- Performance counters:
  - “Page faults/sec” versus “page reads/sec”
  - “Demand zero” faults/second
Reserving & Committing Memory

• Optional 2-phase approach to memory allocation:
  1. Reserve address space (in multiples of page size)
  2. Commit storage in that address space
     – Can be combined in one call (Win32 VirtualAlloc, VirtualAllocEx)

• Reserved memory:
  – Range of virtual addresses reserved for future use (contiguous buffer)
  – Accessing reserved memory results in access violation
  – Fast, inexpensive

• Committed memory:
  – Has backing store (pagefile.sys, memory-mapped file)
  – Either private or mapped into a view of a section
  – Decommit via VirtualFree, VirtualFreeEx

A thread's user-mode stack is constructed using this 2-phase approach: initial reserved size is 1MB, only 2 pages are committed: stack & guard page
Working Set

- Working set: The subset of the virtual address space in physical memory
  - Essentially, all the pages the process can reference without incurring a page fault
  - Upper limit on size for each process
  - When limit is reached, a page must be released for every page that’s brought in (“working set replacement”)

- Working set limit: The maximum pages the process can own
  - Default value for new processes
  - System-wide maximum computed at boot time (see MmMaximumWorkingSetSize)
A process always starts with an empty working set
- Pages itself into existence
- Many page faults may be resolved from memory (to be described later)
• When working set “count” = working set size, must give up pages to make room for new pages
• Page replacement is ”modified FIFO”
  – Windows on uniprocessor x86 implements “least recently accessed”
Memory Management Components

- Working Set Manager (priority 16)
- Process/stack swapper (priority 23)
- Modified page writer (priority 17)
- Mapped page writer (priority 17)
- Dereference segment thread (priority 18)
- Zero paged thread (priority 0)
Unassigned Physical Memory

• System keeps unassigned (available) physical pages on one of several lists:
  – Free page list
  – Modified page list
  – Standby page list
  – Zero page list
  – Bad page list - pages that failed memory test at system startup

• Lists are implemented by entries in the “PFN database”
  – Maintained as FIFO lists or queues
Paging Dynamics

demand zero page faults
page read from disk or kernel allocations

Standby Page List
modified page writer

Free Page List
zero page thread

Modified Page List
working set replacement

Private pages at process exit

Process Working Sets
“soft” page faults

Bad Page List
Balance Set Manager

• Nearest thing Windows has to a “swapper”
  – Balance set = sum of all inswapped working sets

• Balance Set Manager is a system thread
  – Wakes up every second. If paging activity high or memory needed:
    • Trims working sets of processes
    • If thread in a long user-mode wait, marks kernel stack pages as pageable
    • If process has no nonpageable kernel stacks, “outswaps” process
    • Triggers a separate thread to do the “outswap” by gradually reducing target process’s working set limit to zero

• Evidence: Look for threads in “Transition” state in PerfMon
  – Means that kernel stack has been paged out, and thread is waiting for memory to be allocated so it can be paged back in

• This thread also performs a scheduling-related function
  – Priority inversion avoidance
Standby And Modified Page Lists

• Used to:
  – Avoid writing pages back to disk too soon
  – Avoid releasing pages to the free list too soon

• The system can replenish the free page list by taking pages from the top of the standby page list
  – This breaks the association between the process and the physical page
  – I.e., the system no longer knows if the page still contains the process’s info

• Pages move from the modified list to the standby list
  – Modified pages’ contents are copied to the pages’ backing stores (usually the paging file) by the modified page writer (see next slide)
  – The pages are then placed at the bottom of the standby page list

• Pages can be faulted back into a process from the standby and modified page list
  – The SPL and MPL form a system-wide cache of “pages likely to be needed again”
Modified Page Writer

- Moves pages from modified to standby list, and copies their contents to disk
  - I.e., this is what writes the paging file and updates mapped files (including the file system cache)

- Two system threads
  - One for mapped files, one for the paging file

- Triggered when
  - Memory is overcommitted (too few free pages)
  - Or modified page threshold is reached
  - Does not flush entire modified page list
Zero Page List

- Large uninitialized data regions are mapped to demand zero pages
- On first reference to such a page, a page is allocated from the zero page list
  - No need to read zeroes from disk to provide the “data”
  - After modification, these pages are mapped to the paging file
- Zero page list is replenished by the “zero page thread”
  - Thread 0 in “System” process (routine name is Phase1Initialization)
  - Runs at priority 0
    (lower than can be reached by Win32 applications, but above idle threads)
  - One per system (even on SMP)
  - Takes pages from the free page list, fills them with zeroes, and puts them on the zero page list while the CPU is otherwise idle
  - Usually is waiting on an event - which is set when, after resolving a fault, system notices that zero page list is too small
Examining Sizes of Page Lists

- Must use Kernel Debugger

kd> !memusage
!memusage
loading PFN database..........................
  Zeroed: 0 ( 0 kb)
  Free: 322 ( 1288 kb)
  Standby: 1032 ( 4128 kb)
  Modified: 119 ( 476 kb)
  ModifiedNoWrite: 0 ( 0 kb)
  Active/Valid: 2623 ( 10492 kb)
  Transition: 0 ( 0 kb)
  Unknown: 0 ( 0 kb)
  TOTAL: 4096 ( 16384 kb)

Screen snapshot from: Kernel debugger !memusage command
Memory Management Information
Task manager performance tab

“Available” memory = total of free, zero, and standby lists
(majority usually are standby pages)

Windows 2000: System cache = total of cache, paged pool,
system code + size of standby list
(displayed instead of file cache which did not include size of standby list)
Memory Management Information
Task manager performance tab

• “Commit charge total” = total of private (not shared) committed virtual space in all processes; i.e., total of “VM Size” from processes display, + Kernel Memory paged

• “Commit charge limit” = sum of available physical memory for processes + free space in paging file
Memory Management Information
Task manager processes tab

- “Mem Usage” = physical memory used by process (working set size, not working set limit)
- “VM Size” = private (not shared) committed virtual space in processes
- “Mem Usage” in status bar is same as “commit charge/commit limit” in “Performance” tab (see next slide) - not same as “Mem Usage” column here!
Memory Management Information

PerfMon - process object

- “Working Set” = working set size (not limit)
- “Private Bytes” = same as “VM Size” from Task Manager Processes list
- “Virtual Bytes” = committed virtual space, including shared pages
- Also: In Threads object, look for threads in Transition state - evidence of swapping (usually caused by severe memory pressure)

Screen snapshot from: Performance Monitor counters from Process object
Memory information for a process
Resource Kit pview.exe

Virtual sizes of committed sections of image and DLLs or total of all (total selected)

Virtual sizes of sections mapped after image startup (including DLLs loaded with LoadLibrary)

Process-private committed virtual address space (i.e. paging file allocation)

note, “writecopy” = “writeable, but not written to yet”. Windows NT has yet to create process-private pages for these; they are still shared; they become “private commit” when written to

Some, but not all, of this info is also shown by Process Viewer’s “memory detail” button
Memory information for a process
Resource Kit pview.exe

Total virtual address space (committed PLUS reserved, private and shared)

WS = working set (physical)

PF = paging file space allocated (not necessarily written to!)

Same as PerfMon “private bytes”, TaskMan “VM size”

Systemwide paged pool (virtual) and nonpaged pool used by this process

Systemwide paged pool

Systemwide nonpaged pool

Paging file space allocated by all processes + OS

Note, “limits” in the last three groups are per-process limits; i.e., how much each process can use of these