Unit OS2:
Operating System Principles

2.2. Windows Core System Mechanisms

Windows Operating System Internals - by David A. Solomon and Mark E. Russinovich with Andreas Polze

Copyright Notice

Roadmap for Section 2.2.

- Object Manager & Handles
- Local Procedure Calls
- Exception Handling
- Memory Pools

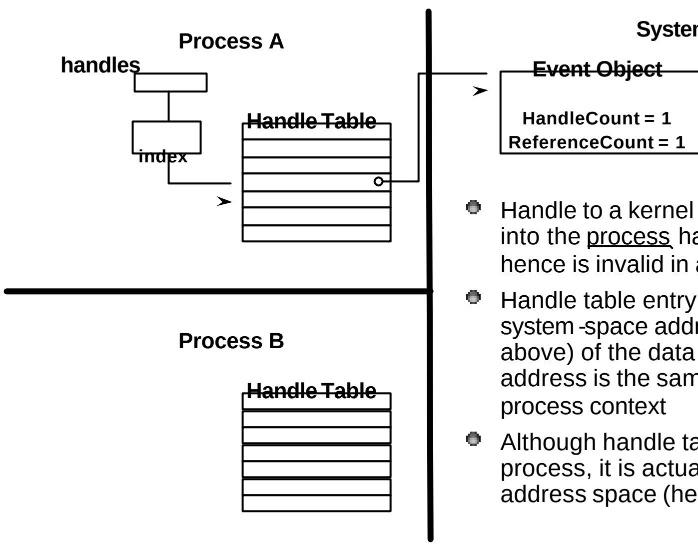
Objects and Handles

- Many Windows APIs take arguments that are handles to system-defined data structures, or "objects"
  - App calls CreateXxx, which creates an object and returns a handle to it
  - App then uses the handle value in API calls that operate on that object
- Three types of Windows objects (and therefore handles):
  - Windows "kernel objects" (events, mutexes, files, processes, threads, etc.)
    - Objects are managed by the NT "Object Manager", and represent data structures in system address space
    - Handle values are private to each process
  - Windows "GDI objects" (pens, brushes, fonts, etc.)
    - Objects are managed by the Windows subsystem
    - Handle values are valid system-wide / session-wide
  - Windows "User objects" (windows, menus, etc.)
    - Objects are managed by the Windows subsystem
    - Handle values are valid system-wide / session-wide

Handles and Security

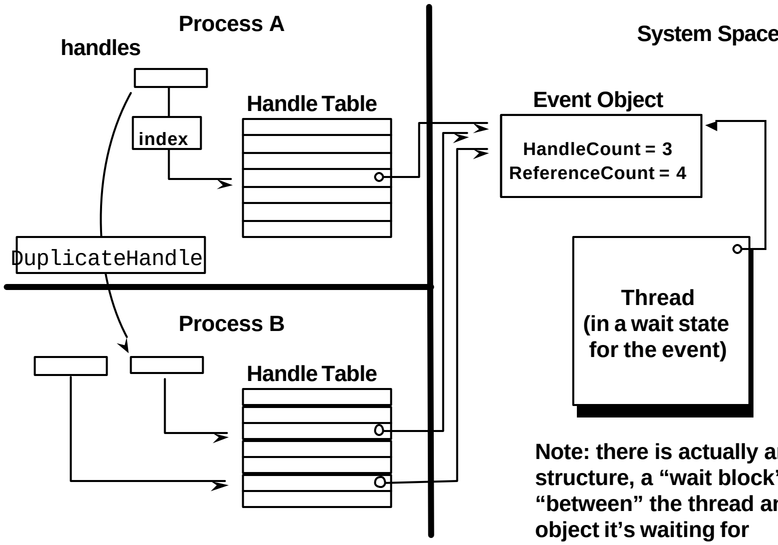
- Process handle table
  - Is unique for each process
  - But is in system address space, hence cannot be modified from user mode
  - Hence, is trusted
- Security checks are made when handle table entry is created
  - i.e. at CreateXxx time
  - Handle table entry indicates the "validated" access rights to the object
    - Read, Write, Delete, Terminate, etc.
- APIs that take an "already-opened" handle look in the handle table entry before performing the function
  - For example: TerminateProcess checks to see if the handle was opened for Terminate access
  - No need to check file ACL, process or thread access token, etc., on every write request---checking is done at file handle creation, i.e. "file open", time

Handles, Pointers, and Objects



- Handle to a kernel object is an index into the <u>process</u> handle table, and hence is invalid in any other process
- Handle table entry contains the system-space address (8xxxxxxx or above) of the data structure; this address is the same regardless of process context
- Although handle table is per-process, it is actually in system address space (hence protected)

Handles, Pointers, and Reference Count



Note: there is actually another data structure, a "wait block", "between" the thread and the object it's waiting for

Object Manager

- Executive component for managing system-defined "objects"
  - Objects are data structures with optional names
  - "Objects" managed here include Windows Kernel objects, but not Windows User or GDI objects
  - Object manager implements user-mode handles and the process handle table
- Object manager is not used for all NT data structures
  - Generally, only those types that need to be shared, named, or exported to user mode
  - Some data structures are called "objects" but are not managed by the object manager (e.g. "DPC objects")

Object Manager

- In part, a heap manager…
  - Allocates memory for data structure from system -wide, kernel space heaps
    (pageable or nonpageable)
- … with a few extra functions:
  - Assigns name to data structure (optional)
  - Allows lookup by name
  - Objects can be protected by ACL-based security
  - Provides uniform naming, sharing, and protection scheme
    - Simplifies C2 security certification by centralizing all object protection in one place
  - Maintains counts of handles and references (stored pointers in kernel space) to each object
    - Object cannot be freed back to the heap until all handles and references are gone

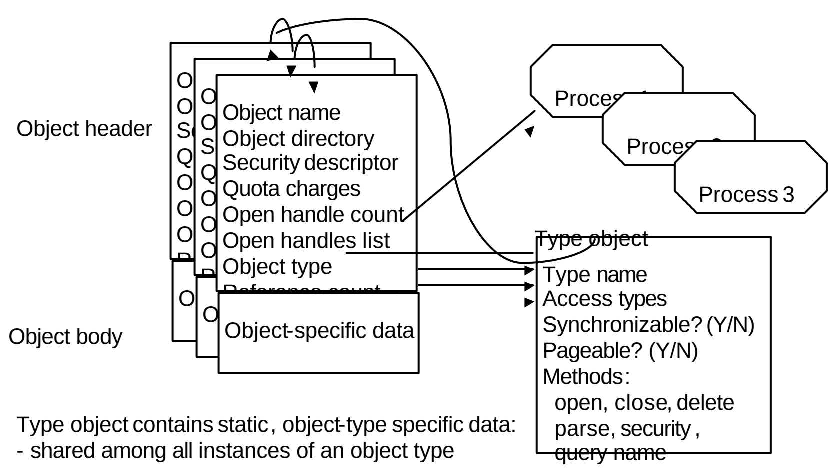
Executive Objects

Object type	Represents
Object directory	Container object for other objects: implement hierarchical namespace to store other object types
Symbolic link	Mechanism for referring to an object name indirectly
Process	Virtual address space and control information necessary for execution of thread objects
Thread	Executable entity within a process
Section	Region of shared memory (file mapping object in Windows API)
File	Instance of an opened file or I/O device
Port	Mechanism to pass messages between processes
Access token	Security profile (security ID, user rights) of a process or thread

Executive Objects (contd.)

Object type	Represents
Event	Object with persistent state (signaled or not) usable for synchronization or notification
Semaphore	Counter and resource gate for critical section
Mutant	Synchronization construct to serialize resource access
Timer	Mechanism to notify a thread when a fixed period of time elapses
Queue	Method for threads to enqueue/dequeue notifications of I/O completions (Windows I/O completion port)
Key	Reference to registry data – visible in object manager namespace
Profile	Mechanism for measuring execution time for a process within an address range

Object Structure



Type object contains static, object-type specific data:
- shared among all instances of an object type
- link all instances together (enumeration)

Object Methods

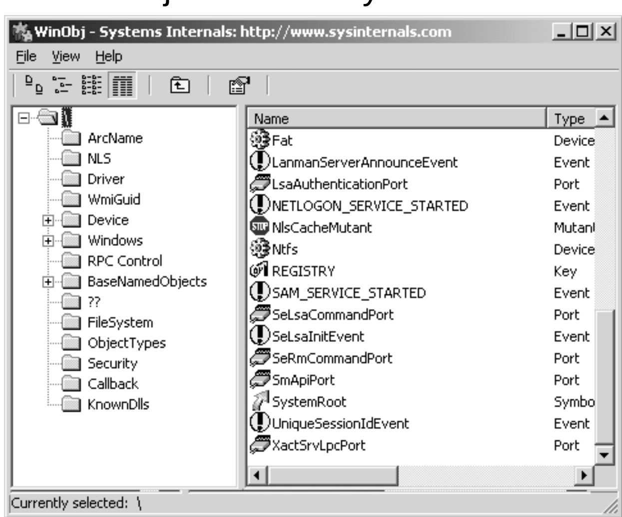
Method	When method is called
Open	When an object handle is opened
Close	When an object handle is closed
Delete	Before the object manager deletes an object
Query name	When a thread requests the name of an object, such as a file, that exists in a secondary object domain
Parse	When the object manager is searching for an object name that exists in a secondary object domain
Security	When a process reads/changes protection of an objects, such as a file, that exists in a secondary object domain

Example:

- Process opens handle to object \Device\Floppy0\docs\resume.doc
- Object manager traverses name tree until it reaches Floppy0
- Calls parse method for object Floppy0 with arg \docs\resume.doc

Object Manager Namespace

- System and session-wide internal namespace
- View with Winobj from www.sysinternals.com



Interesting Object Directories

- in \ObjectTypes
  - objects that define types of objects
- in \BaseNamedObjects

  these will appear when Windows programs use CreateEvent, etc.
  - mutant (Windows mutex)
  - queue (Windows I/O completion port)
  - section (Windows file mapping object)
  - event
  - Semaphore
- In \GLOBAL??
  - DOS device name mappings for console session

Object Manager Namespace

- Namespace:
  - Hierarchical directory structure (based on file system model)
  - System-wide (not per-process)
    - With Terminal Services, Windows objects are per-session by default
    - Can override this with "global\" prefix on object names
  - Volatile (not preserved across boots)
    - As of Server 2003, requires SeCreateGlobalPrivilege
  - Namespace can be extended by secondary object managers (e.g. file system)
    - Hook mechanism to call external parse routine (method)
  - Supports case sensitive or case blind
  - Supports symbolic links (used to implement drive letters, etc.)
- Lookup done on object creation or access by name
  - Not on access by handle
- Not all objects managed by the object manager are named
  - e.g. file objects are not named
  - un-named objects are not visible in WinObj

Kernel Memory Pools
(System-Space Heaps)

- Nonpaged pool
  - Has initial size and upper limit (can be grown dynamically, up to the max)
  - 32-bit upper limit: 256 MB on x86 (NT4: 128MB)
    - 64-bit: 128 GB
  - Performance counter displays current total size (allocated + free)
  - Max size stored in kernel variable MmMaximumNonPagedPoolInBytes
- Paged pool
  - 32-bit upper limit: 650MB (Windows Server 2003), 470MB (Windows 2000), 192MB (Windows NT 4.0)
    - 64-bit: 128 GB
  - Max size stored in MmSizeOfPagedPoolInBytes
- Pool size performance counters display current size, not max
  - To display maximums, use “!vm” kernel debugger command

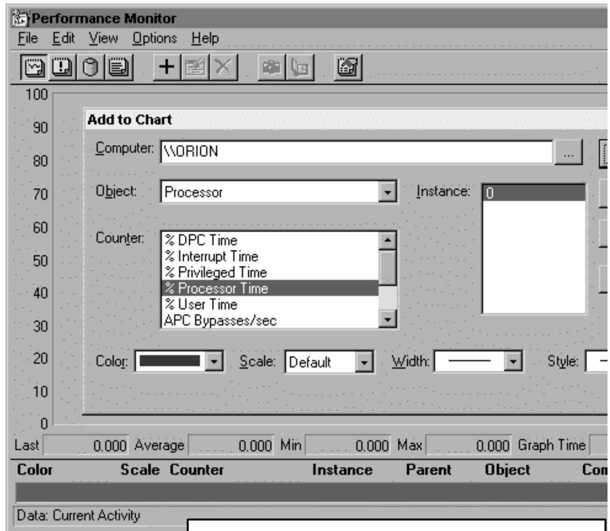
Invoking Kernel-Mode Routines

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

- Requests from user mode
  - Via the system service dispatch mechanism
  - Kernel-mode code runs in the context of the requesting thread
- Interrupts from external devices
  - Interrupts (like all traps) are handled in kernel mode
  - NT-supplied 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
- Dedicated kernel-mode 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

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
  - other kernel-mode time (no separate counter for this)



Screen snapshot from: Programs | Administrative Tools | Performance Monitor click on "+" button, or select Edit | Add to chart...

Kernel Memory Pools (System-Space Heaps)

- NT provides two system memory pools:
  - "Nonpaged Pool" and "Paged Pool"
  - Used for systemwide persistent data (visible from any process context)
  - Nonpaged pool required for memory accessed from DPC/dispatch IRQL or above
    - Page faults at DPC/dispatch IRQL or above cause a system crash
- Pool sizes are a function of memory size & Server vs. Workstation
  - Can be overidden in Registry:
    - HKLM\System\CurrentControlSet\Control\Session Manager \Memory Management
  - But are limited by implementation limits (next slide)

Lookaside Lists

- Instead of frequently allocating and freeing pool…
- …Use lookaside lists instead
- Implements a driver private list of preallocated blocks of pool
- Routines automatically extend the lookaside list from the systemwide pool
- Avoids hitting systemwide pool spinlock for every allocation/release
- Optimizes CPU/memory cache behavior (the same driver will use the same list items over and over)
- See:
  - ExInitialize[N]PagedLookasideList
  - ExAllocateFrom[N]PagedLookasideList
  - ExFreeTo[N]PagedLookasideList
  - ExDelete[N]PagedLookasideList

Increased System Memory Limits

- Key system memory limits raised in XP & Server 2003
- Windows 2000 limit of 200 GB of mapped file data eliminated
  - Previously limited size of files that could be backed up
- Maximum System Page Table Entries (PTEs) increased
  - Can now describe 1.3 GB of system space (960 MB contiguous)
  - Windows 2000 limit was 660 MB (220 MB contiguous)
  - Increases number of users on Terminal Servers
  - Also means maximum device driver size is now 960 MB (was 220 MB)

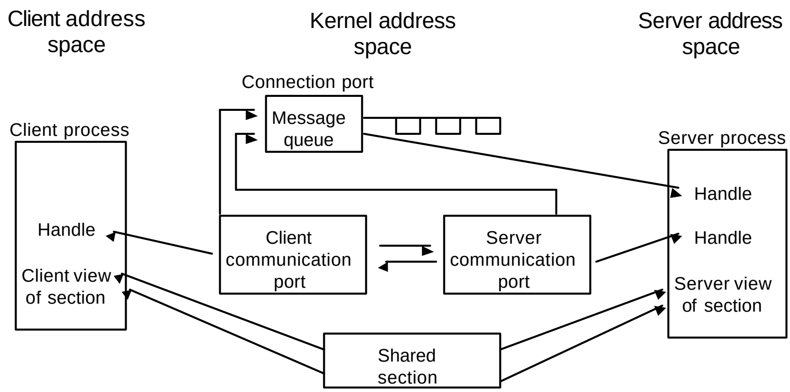
Local Procedure Calls (LPCs)

- IPC – high-speed message passing
- Not available through Windows API – Windows OS internal
- Application scenarios:
  - RPCs on the same machine are implemented as LPCs
  - Some Windows APIs result in sending messages to Windows subsyst. proc.
  - WinLogon uses LPC to communicate with local security authentication server process (LSASS)
  - Security reference monitor uses LPC to communicate with LSASS
- LPC communication:
  - Short messages < 256 bytes are copied from sender to receiver
  - Larger messages are exchanged via shared memory segment
  - Server (kernel) may write directly in client's address space

Port Objects

- LPC exports port objects to maintain state of communication:
  - **Server connection port**: named port, server connection request point
  - **Server communication port**: unnamed port, one per active client, used for communication
  - **Client communication port**: unnamed port a particular client thread uses to communicate with a particular server
  - **Unnamed communication port**: unnamed port created for use by two threads in the same process
- Typical scenario:
  - Server creates named connection port
  - Client makes connection request
  - Two unnamed ports are created, client gets handle to server port, server gets handle to client port

Use of LPC ports

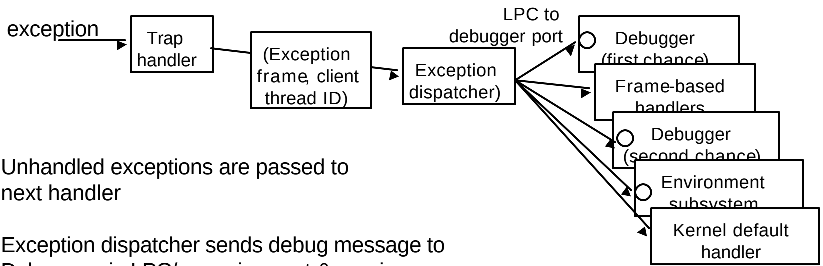


Exception Dispatching

- Exceptions are conditions that result directly from the execution of the program that is running
- Windows introduced a facility known as structured exception handling, which allows applications to gain control when exceptions occur
- The application can then fix the condition and return to the place the exception occurred,
  - unwind the stack (thus terminating execution of the subroutine that raised the exception), or
  - declare back to the system that the exception isn't recognized and the system should continue searching for an exception handler that might process the exception.

Exception Dispatching (contd.)

- Structured exception handling;
  - Accessible from MS VC++ language: __try, __except, __finally
  - See Jeffrey Richter, „Advanced Windows“, MS Press
  - See Johnson M.Hart, „Win32 System Programming“, Addison-Wesley



Unhandled exceptions are passed to next handler

Exception dispatcher sends debug message to Debugger via LPC/excepion port & session manager process

Internal Windows API exception handler

- Processes unhandled exceptions
  - At top of stack, declared in StartOfProcess()/StartOfThread()

```
void Win32StartOfProcess(LPTHREAD_START_ROUTINE lpStartAddr,
                         LPVOID lpvThreadParm) {
    __try {
            DWORD dwThreadExitCode = lpStartAddr(lpvThreadParm);
            ExitThread(dwThreadExitCode);
    }       __except(UnhandledExceptionFilter(
            GetExceptionInformation())) {
            ExitProcess(GetExceptionCode());
    }
}
```

Further Reading

- Mark E. Russinovich and David A. Solomon, Microsoft Windows Internals, 4th Edition, Microsoft Press, 2004.
- Chapter 3 - System Mechanisms
  - Object Manager (pp. 124 ff.)
  - System Worker Threads (pp. 166 ff.)
  - Local Procedure Calls (LPCs) (pp. 171 ff.)
  - Exception Dispatching (pp. 109 ff.)