Unit OS4: Scheduling and Dispatch

4.3. Windows Process and Thread Internals

Copyright Notice

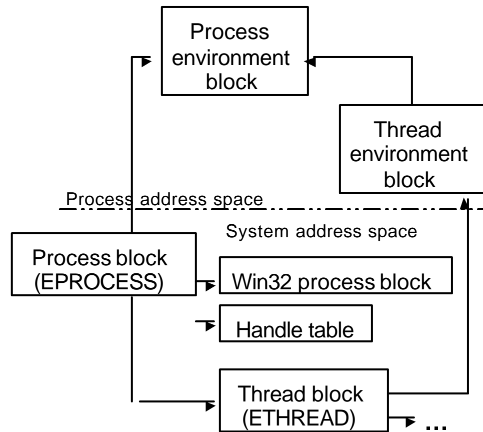
© 2000-2005 David A. Solomon and Mark Russinovich

- These materials are part of the *Windows Operating System Internals Curriculum Development Kit,* developed by David A. Solomon and Mark E. Russinovich with Andreas Polze
- Microsoft has licensed these materials from David Solomon Expert Seminars, Inc. for distribution to academic organizations solely for use in academic environments (and not for commercial use)

Windows Process and 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



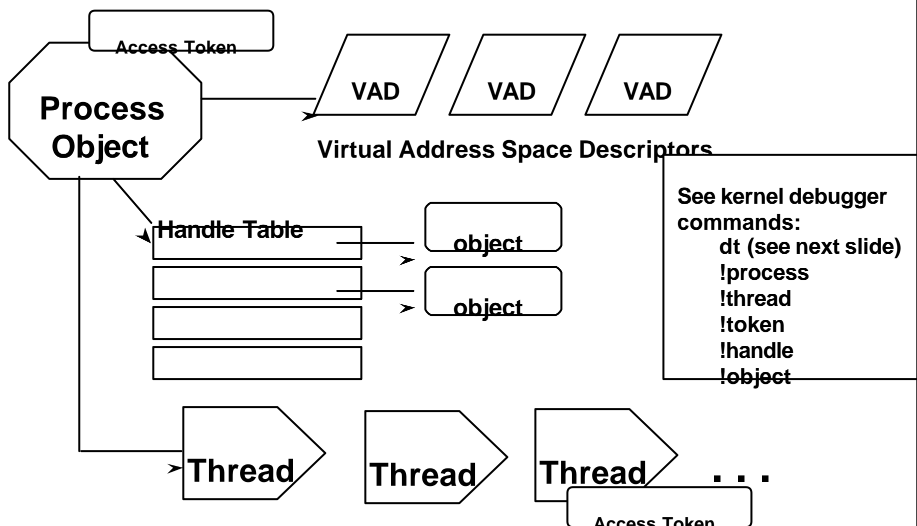
Process

- 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
- MM structures like the WorkingSet, VAD tree, AWE etc

Thread

- Fundamental schedulable entity in the system
- Represented by ETHREAD that includes a KTHREAD
- Queued to the process (both E and K thread)
- IRP list
- Impersonation Access Token
- Unique thread ID
- Associated User-mode Thread Environment Block (TEB)
- User-mode stack
- Kernel-mode stack
- Processor Control Block (in KTHREAD) for CPU state when not running

Processes & Threads Internal Data Structures



Process/Thread Kernel Debugger Commands

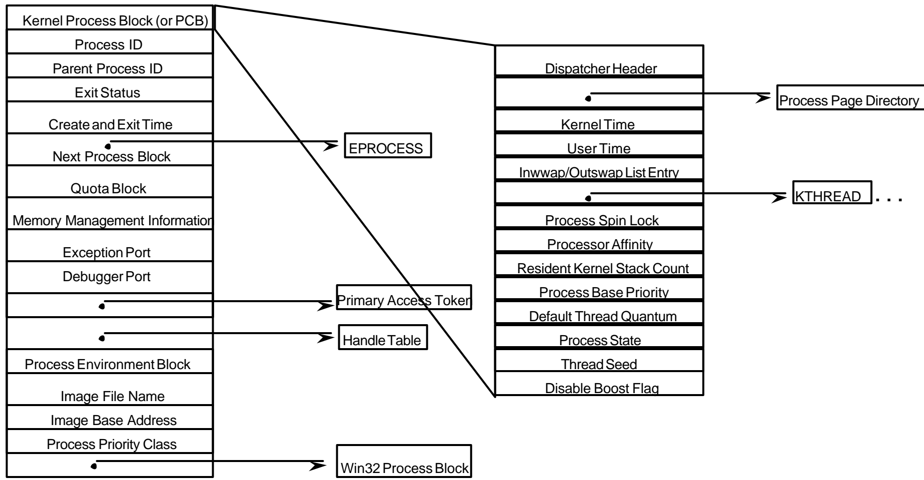
- !process [/s Session] [Address/Pid [Flags]]
  - !process – display current process (not full details)
  - !process 342 – display full details of process 342
  - !process 829fa030 – display process identified by EPROCESS address
  - !process 0 0 – summary display of all processes
  - !process 0 7 – full details of all processes
- !thread [Address [Flags]]
  - !thread – current thread
  - !thread 826e8898 – display thread identified by ETHREAD address
- To view user stack, must set process context:
  - .process <address of EPROCESS>
  - .context <address of page directory (Dirbase)>
- !peb [Address]
- !teb [Address]

Process Block (!process)

EPROCESS address · Process ID · Address of process environment block · Process ID of parent process · Physical address of Page Directory · root of the process's Virtual Address Descriptor tree · Time the process has been running, divided into User and Kernel time

```
PROCESS ff704020  Cid: 0075    Peb: 7ffdf000  ParentCid: 005d
    DirBase: 0063c000  ObjectTable: ff7063c8  TableSize:  70.
    Image: Explorer.exe
    VadRoot ff70d6e8 Clone 0 Private 229. Modified 236. Locked 0.
    FF7041DC MutantState Signalled OwningThread 0
    Token                             e1462030
    ElapsedTime                        0:01:19.0874
    UserTime                          0:00:00.0991
    KernelTime                        0:00:02.0613
    QuotaPoolUsage[PagedPool]         18317
    QuotaPoolUsage[NonPagedPool]      3824
    Working Set Sizes (now,min,max)  (727, 20, 45) (2908KB, 80KB, 180KB)
    PeakWorkingSetSize                757
    VirtualSize                       29 Mb
    PeakVirtualSize                   31 Mb
    PageFaultCount                    1396
    MemoryPriority                    FOREGROUND
    BasePriority                      8
    CommitCharge                      250
```

Process Block Layout



Process Block Layout

```
lkd> dt nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x06c ProcessLock      : _EX_PUSH_LOCK
   +0x070 CreateTime       : _LARGE_INTEGER
   +0x078 ExitTime         : _LARGE_INTEGER
   +0x080 RundownProtect   : _EX_RUNDOWN_REF
   +0x084 UniqueProcessId  : Ptr32 Void
   +0x088 ActiveProcessLinks : _LIST_ENTRY
   +0x090 QuotaUsage       : [3] Uint4B
   +0x09c QuotaPeak        : [3] Uint4B
   +0x0a8 CommitCharge     : Uint4B
   +0x0ac PeakVirtualSize  : Uint4B
   +0x0b0 VirtualSize      : Uint4B
                .
                .
```

- **NOTE: Add “-r” to recurse through substructures**

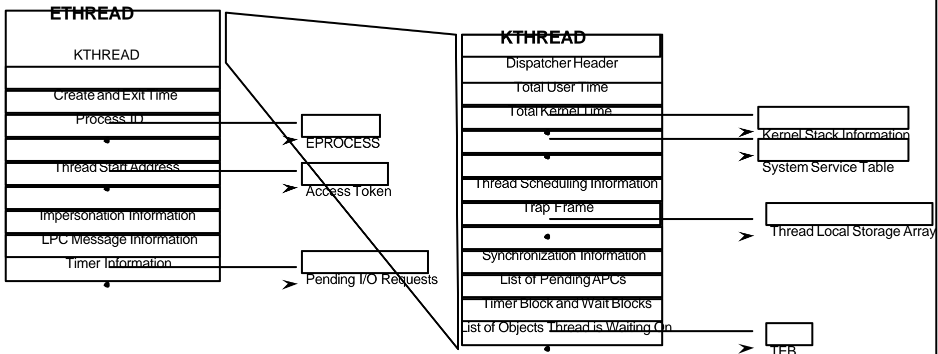
Thread Block (!thread)

Thread ID
Process ID
Address of thread environment block
Address of system service dispatch table
Address of EHTREAD
Thread state
Objects being waited on
Actual thread start address
Address of user thread function
Priority Information
Stack trace

```
 THREAD 83160f60  Cid 9f.3d  Teb: 7ffdc000  Win32Thread: e153d2c8
WAIT: (WrUserRequest) UserMode Non-Alertable
            808e9d60  SynchronizationEvent
        Not impersonating
        Owning Process 81b44880
        WaitTime (seconds)      953945
        Context Switch Count    2697                   LargeStack
        UserTime                  0:00:00.0289
        KernelTime                0:00:04.0664
        Start Address kernel32!BaseProcessStart (0x77e8f268)
        Win32 Start Address 0x020d9d98
        Stack Init f7818000 Current f7817bb0 Base f7818000 Limit f7812000 Call 0
        Priority 14 BasePriority 8 PriorityDecrement 6 DecrementCount 13
Kernel stack not resident.

        ChildEBP RetAddr  Args to Child
        f7817bb0 8008f430 00000001 00000000 00000000 ntoskrnl!KiSwapThreadExit
        f7817c50 de0119ec 00000001 00000000 00000000 ntoskrnl!KeWaitForSingleObject+0x2a0
        f7817cc0 de0123f4 00000001 00000000 00000000 win32k!xxxSleepThread+0x23c
        f7817d10 de01f2f0 00000001 00000000 00000000 win32k!xxxInternalGetMessage+0x504
        f7817d80 800bab58 00000001 00000000 00000000 win32k!NtUserGetMessage+0x58
        f7817df0 77d887d0 00000001 00000000 00000000 ntoskrnl!KiSystemServiceEndAddress+0x4
        0012fef0 00000000 00000001 00000000 00000000 user32!GetMessageW+0x30
```


Thread Block



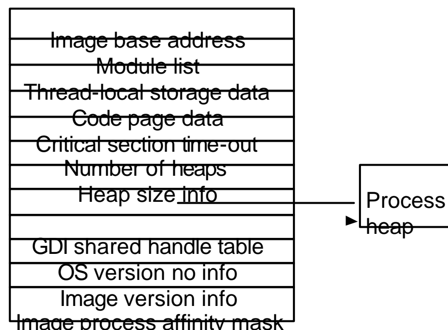
Thread Block (!strct ethread)

```
lkd> dt nt!_ETHREAD
   +0x000 Tcb              : _KTHREAD
   +0x1c0 CreateTime       : _LARGE_INTEGER
   +0x1c0 NestedFaultCount : Pos 0, 2 Bits
   +0x1c0 ApcNeeded        : Pos 2, 1 Bit
   +0x1c8 ExitTime         : _LARGE_INTEGER
   +0x1c8 LpcReplyChain    : _LIST_ENTRY
   +0x1c8 KeyedWaitChain   : _LIST_ENTRY
   +0x1d0 ExitStatus       : Int4B
   +0x1d0 OfsChain         : Ptr32 Void
   +0x1d4 PostBlockList    : _LIST_ENTRY
   +0x1dc TerminationPort  : Ptr32 _TERMINATION_PORT
   +0x1dc ReaperLink       : Ptr32 _ETHREAD
```

- **NOTE: Add "-r" to recurse through substructures**

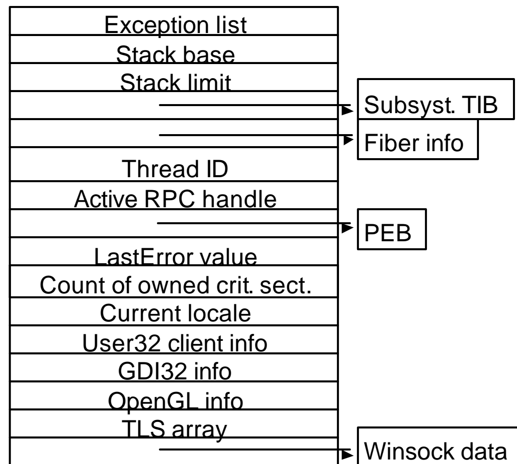
Process Environment Block

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



Thread Environment Block

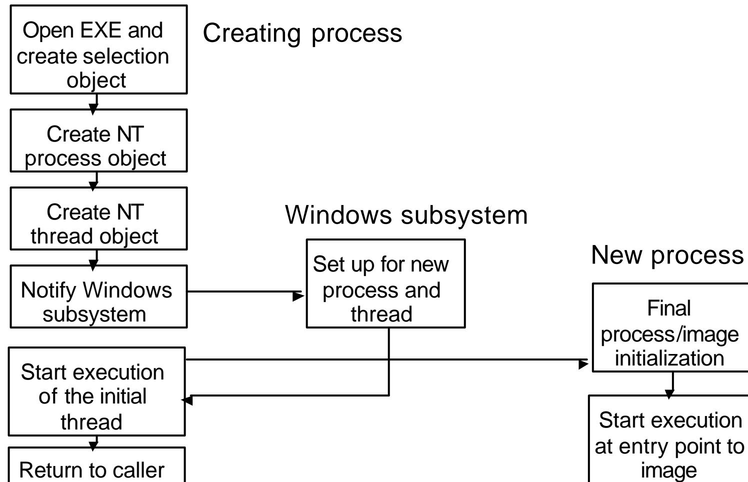
- User mode data structure
- Context for image loader and various Windows DLLs
- View with !teb or dt nt!_teb



Flow of CreateProcess()

1. Open the image file (.EXE) to be executed inside the process
2. Create Windows NT executive process object
3. Create initial thread (stack, context, Win NT executive thread object)
4. Notify Windows subsystem of new process so that it can set up for new proc.& thread
5. Start execution of initial thread (unless CREATE_SUSPENDED was specified)
6. In context of new process/thread: complete initialization of address space (load DLLs) and begin execution of the program

The main Stages Windows follows to create a process

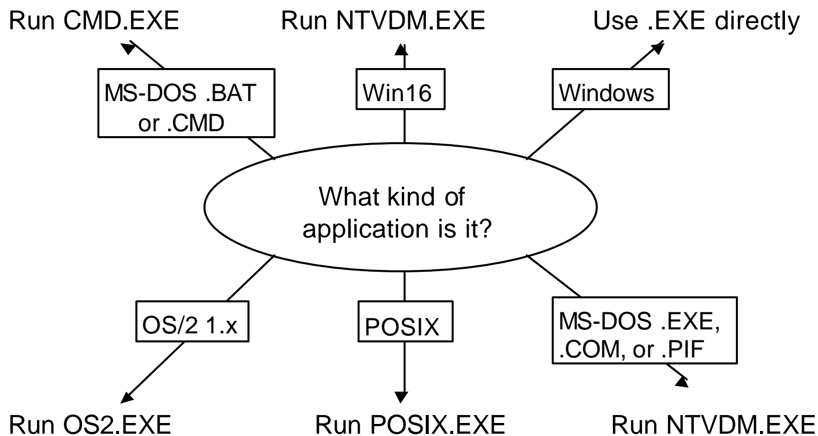


CreateProcess: some notes

- CreationFlags: independent bits for priority class -> NT assigns lowest-prio class set
- Default prio class is normal unless creator has prio class idle
- If real-time prio class is specified and creator has insufficient privileges: prio class high is used
- Caller's current desktop is used if no desktop is specified

Priority classes:
- Real-time
- High
- Normal
- idle

Opening the image to be executed



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 proc‘s addr 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 inital quantum)
- Conlude setup of process address space (VM, map NTDLL.DLL, map lang 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: duplicate handles (inc usage count), set prio 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 window

CreateProcess: final steps

Process Initialization in context of new process:

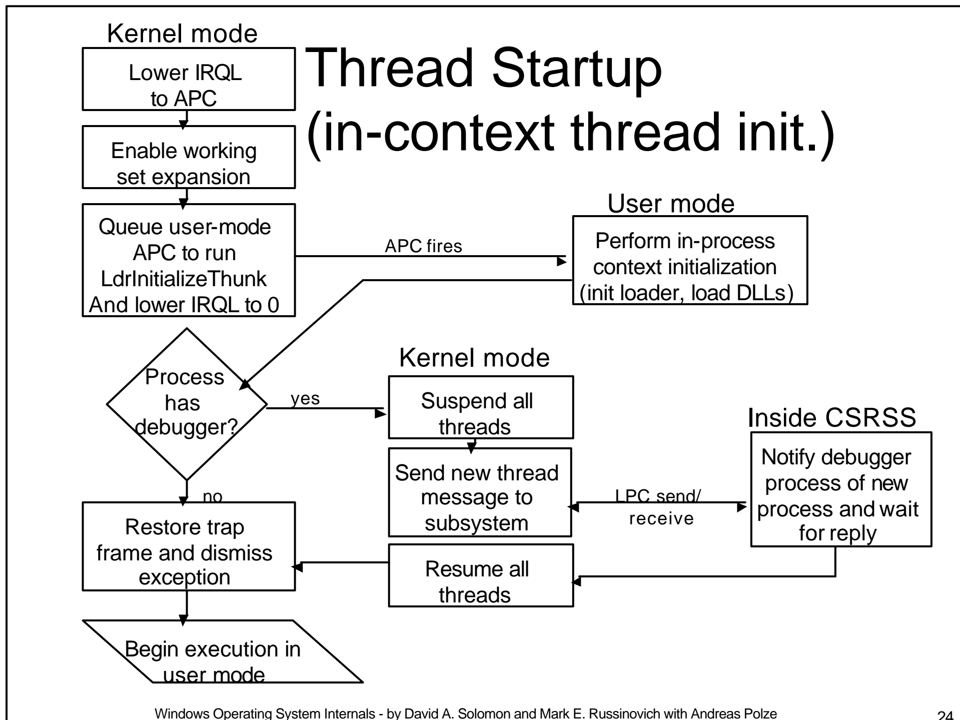
- Lower IRQL level (dispatch -> **A**sync.**P**roc.**C**all. level)
- Enable working set expansion
- Queue APC to exec *LdrInitializeThunk* in NTDLL.DLL
- Lower IRQL level to 0 – APC fires,
  - Init loader, heap manager, NLS tables, TLS array, crit. sect. Structures
  - Load DLLs, call DLL_PROCESS_ATTACH func
- 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 Rundown 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

```
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



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

```
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)
- 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)

Windows Start of Process/Thread Function(conceptual model)

```
void BaseProcessStart  [or BaseThreadStart - basically the same]  (
    LPTHREAD_START_ROUTINE lpStartAddr,
    LPVOID lpvThreadParm )
{
    __try {
        DWORD dwThreadExitCode = lpStartAddr(lpvThreadParm );
        ExitThread(dwThreadExitCode);
    }
    __except(UnhandledExceptionFilter(
            GetExceptionInformation())) {
        ExitProcess(GetExceptionCode());
    }
}
```

Windows Unhandled Exception Filter

```
if process has a debugger attached
   return EXCEPTION_CONTINUE_SEARCH
if AUTO=0 {                        // run debugger automatically?
   Display message box;            // no - ask user what to do
   if(clicked OK)
            ExitProcess();
}

// either AUTO=1, or (AUTO=0 and user clicked CANCEL),
// so run debugger
GetProfileString("AEdebug","debugger",...);
hEvent = CreateEvent( ... );
hProcess = CreateProcess(...);  // Create debugger
   - pass process id, event to signal
WaitForMultipleObjects( [hEvent, hProcess] );
return EXCEPTION_CONTINUE_SEARCH;
```

- **Implication: you can connect a debugger (VC++ or WinDbg) to a running process**

```
C:\> msdev -p pid
```

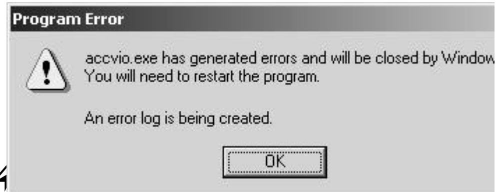

Process Crashes (Windows 2000)

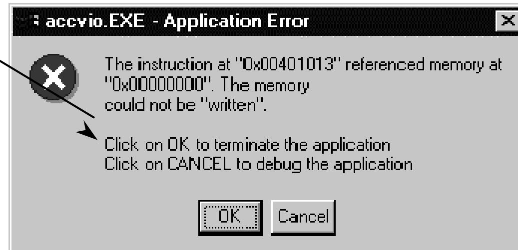
- Registry defines behavior for unhandled exceptions

  HKLM\Software\Microsoft\Windows NT\CurrentVersion\AeDebug

  Debugger=filespec of debugger to run on app crash

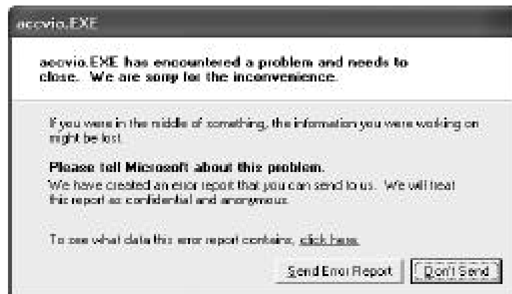
  Auto 1=run debugger immediately
  0=ask user first
- Default on retail system is
  Auto=1; Debugger=DRWTSN32.EXE
- Default with VC++ is
  Auto=0, Debugger=MSDEV.EXE





Process Crashes (Windows XP & Windows Server 2003)

- On XP & Server 2003, when an unhandled exception occurs:
  - System first runs DWWIN.EXE
    - DWWIN creates a process microdump and XML file and offers the option to send the error report
  - Then runs debugger (default is Drwtsn32.exe)



Windows Error Reporting

- Configurable with System Properties->Advanced->Error Reporting
  - HKLM\SOFTWARE\Microsoft\PCHealth\ErrorReporting
- Configurable with group policies
  - HKLM\SOFTWARE\Policies\Microsoft\PCHealth