Unit 4: Scheduling and Dispatch

4.3. Windows Process and Thread Internals

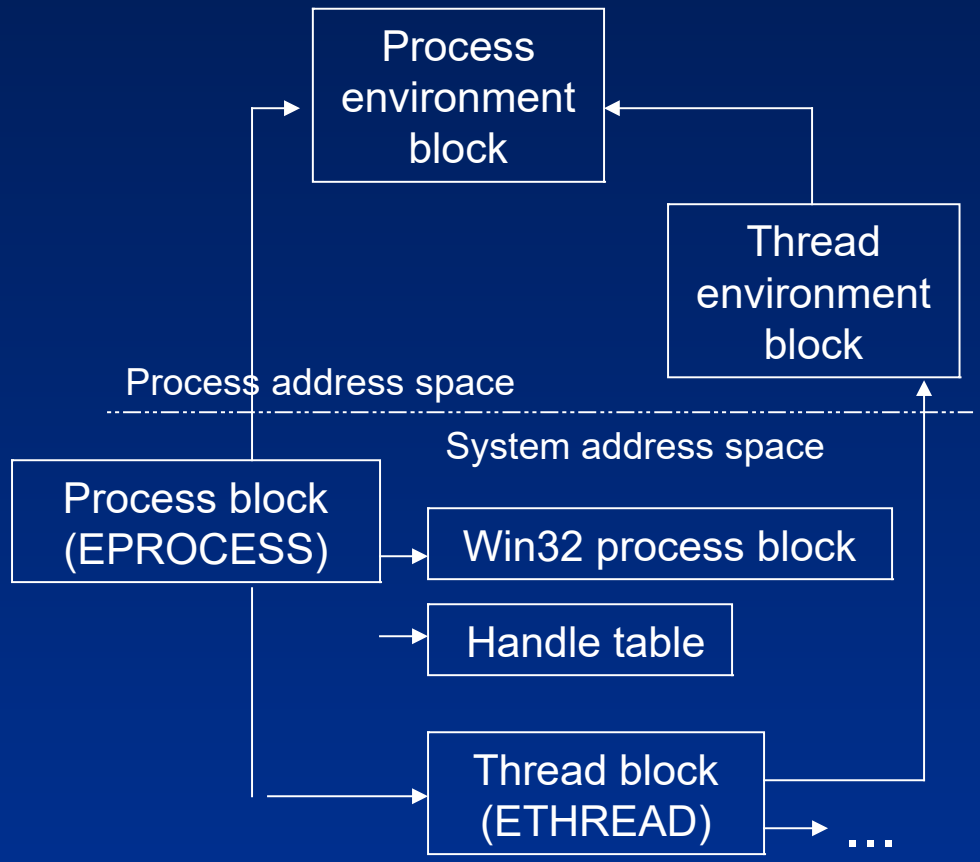
Roadmap for Section 4.3.

- Windows Process and Thread Internals
- Thread Block, Process Block
- Flow of Process Creation
- Thread Creation and Deletion
- Process Crashes
- Windows Error Reporting

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



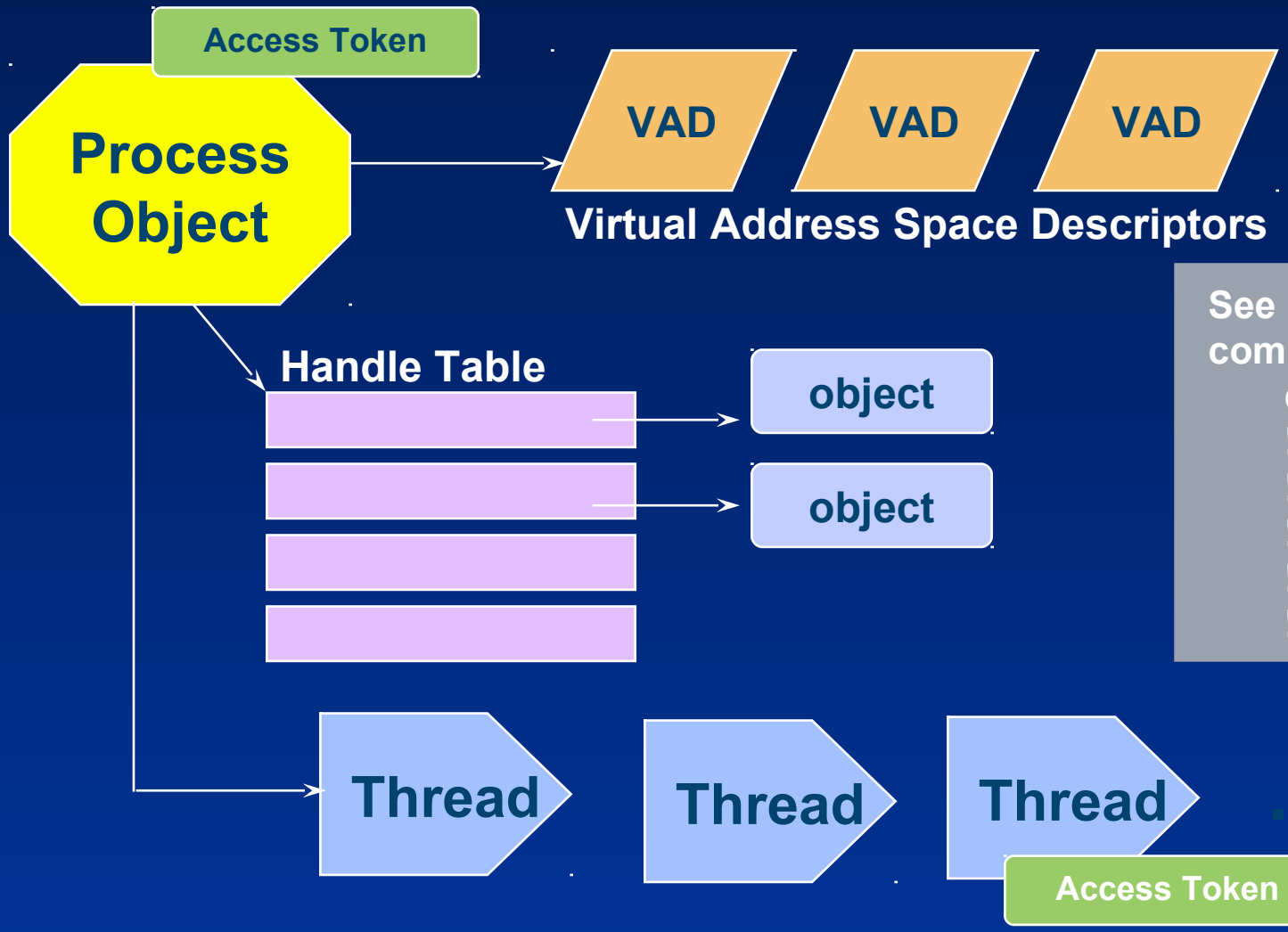
Process

- 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 list, Global process list and Session list
- Memory Management structures like the Working Set, 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

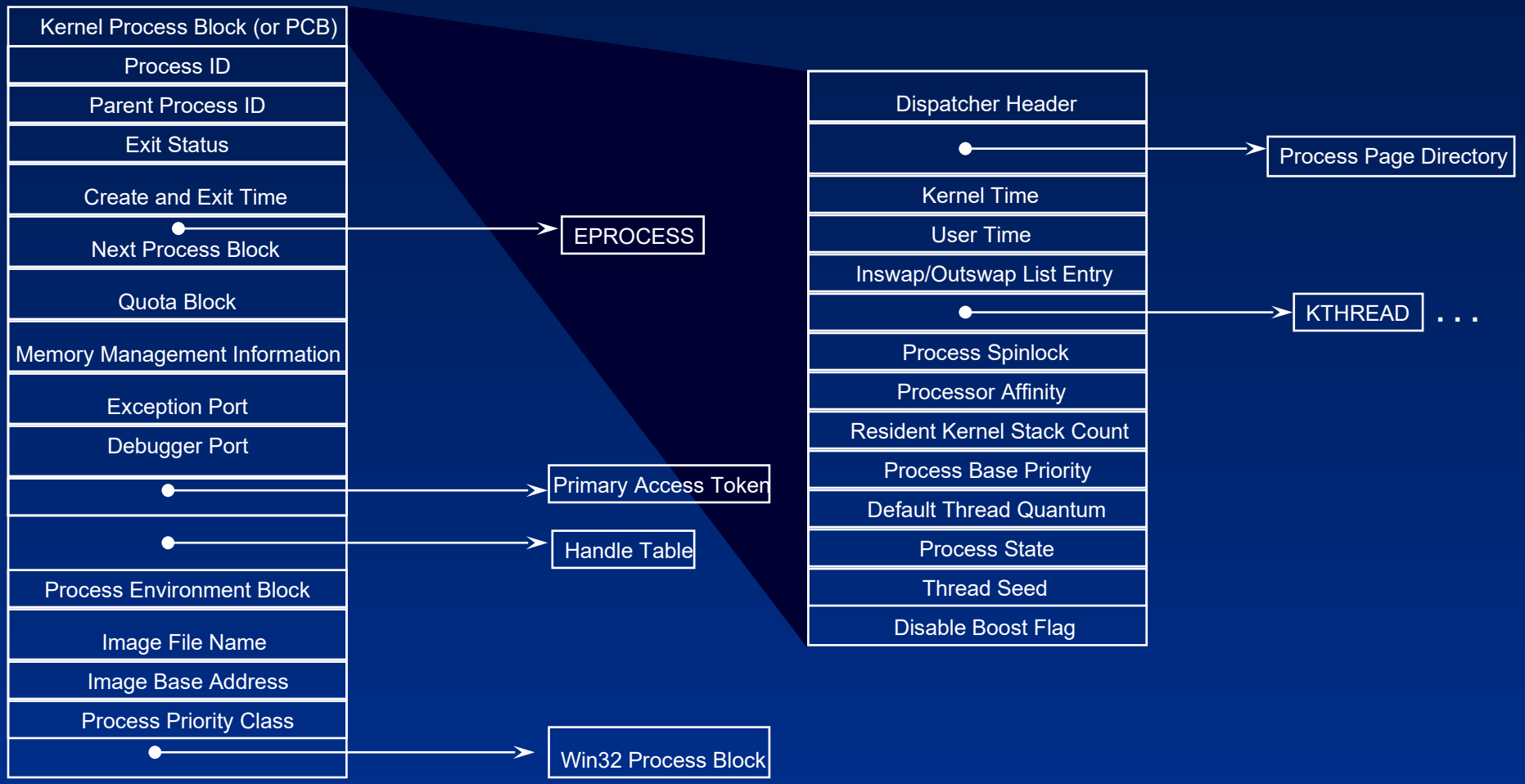
Processes & Threads
Internal Data Structures



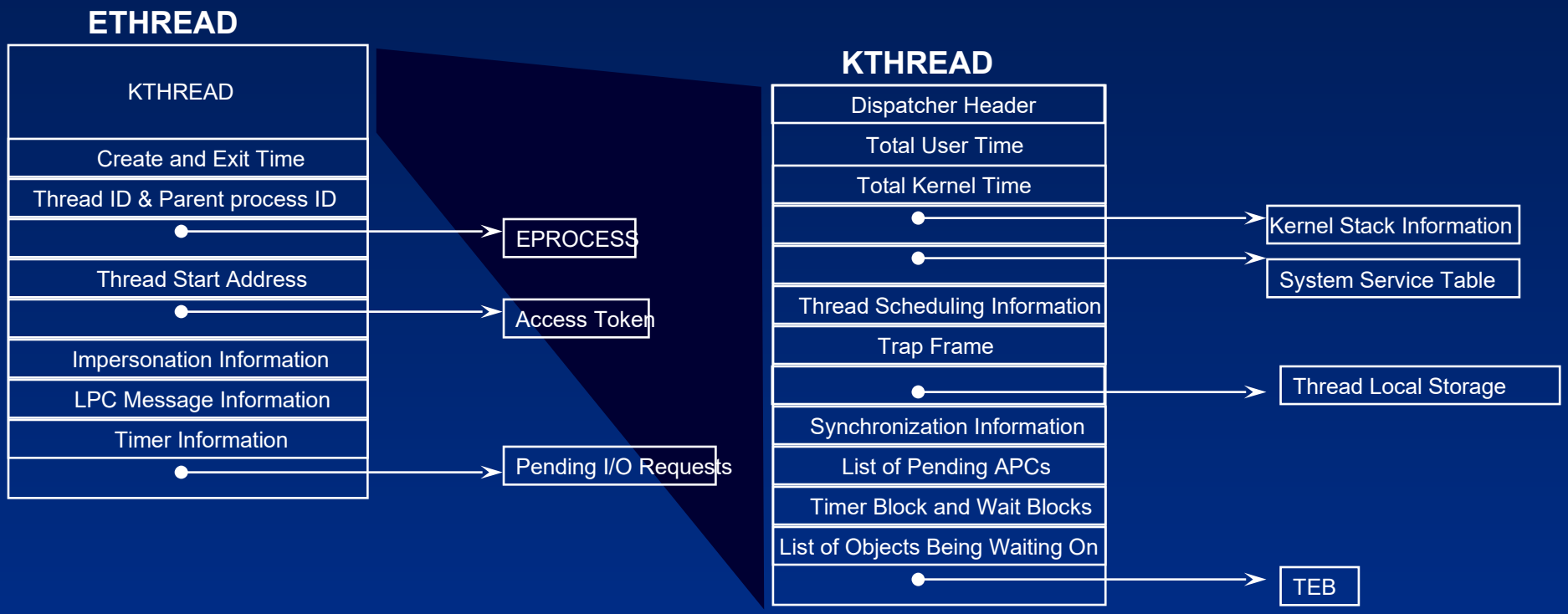
See kernel debugger commands:

- dt (see next slide)
- !process
- !thread
- !token
- !handle
- !object

Process Block Layout

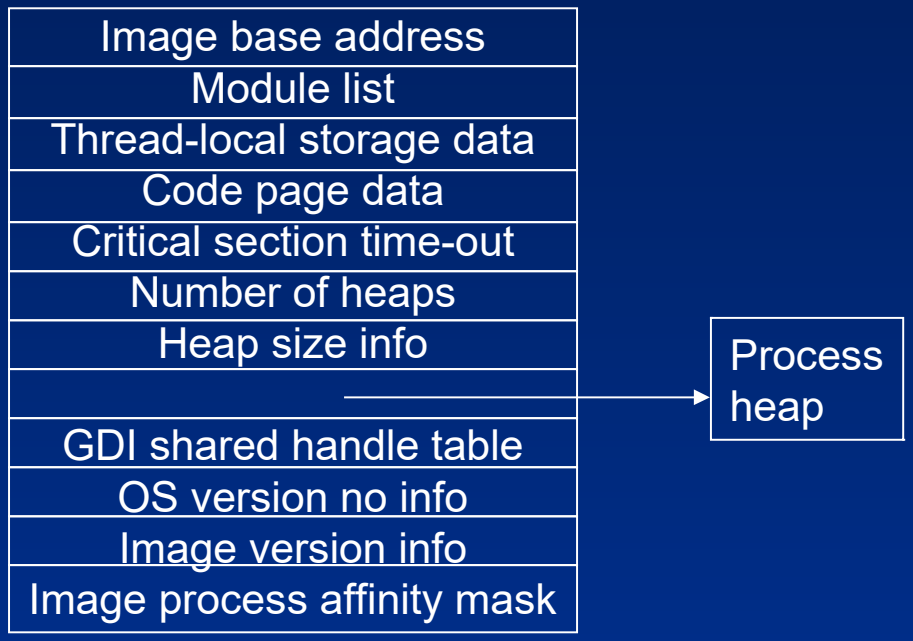


Thread Block Layout



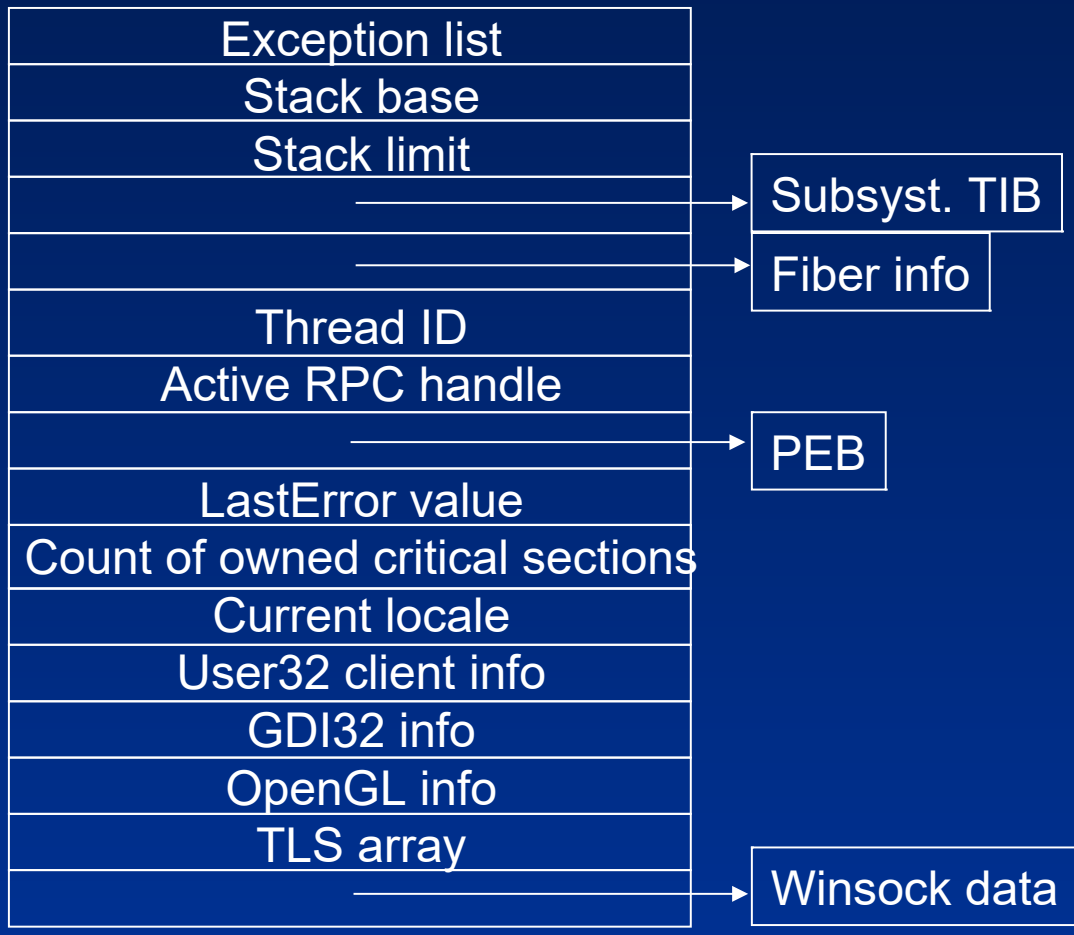
Process Environment Block

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



Thread Environment Block

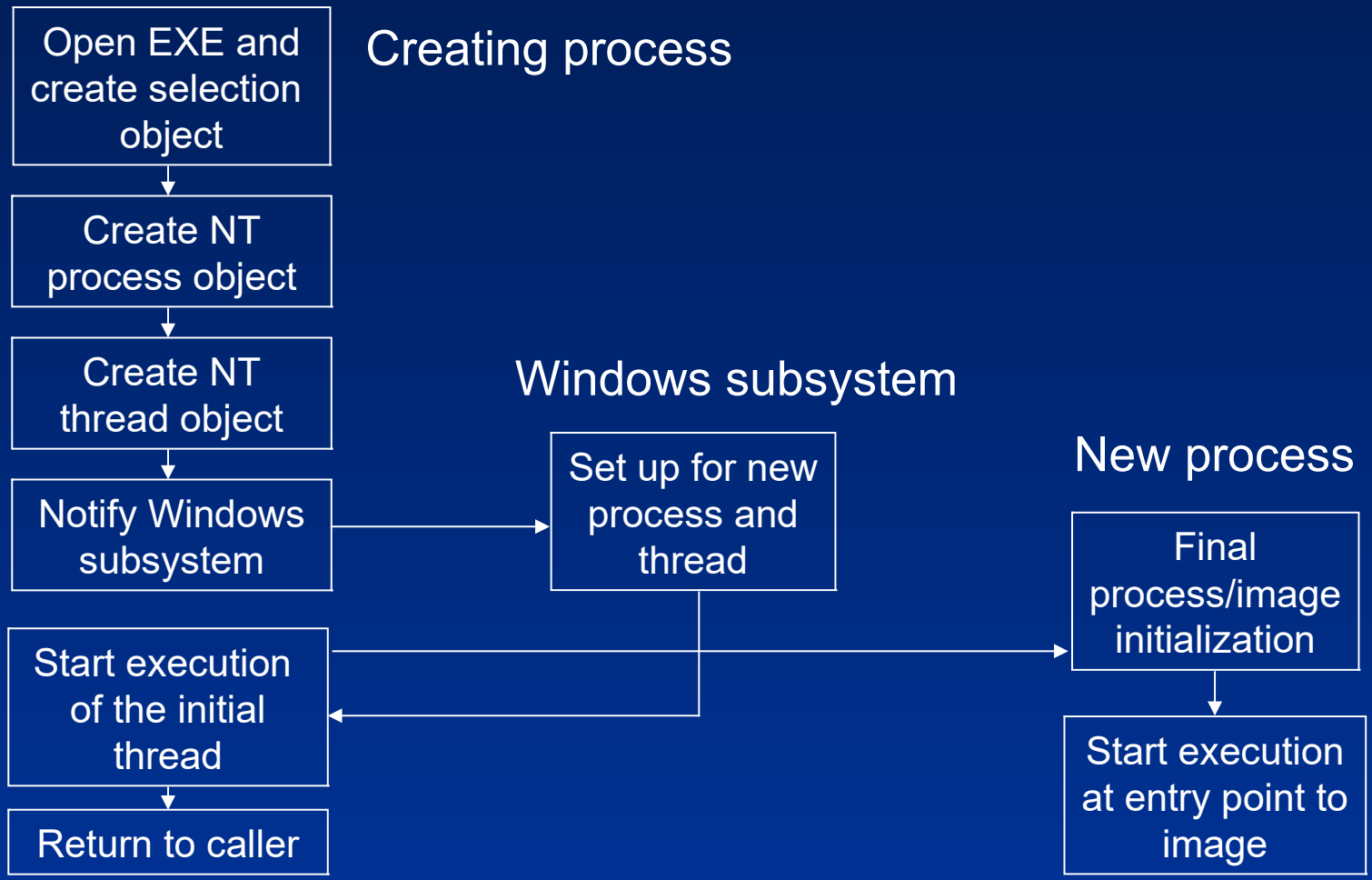
- User mode data structure
- Context for image loader and various Windows DLLs
- View with !teb or dt nt!_teb



Flow of CreateProcess()

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

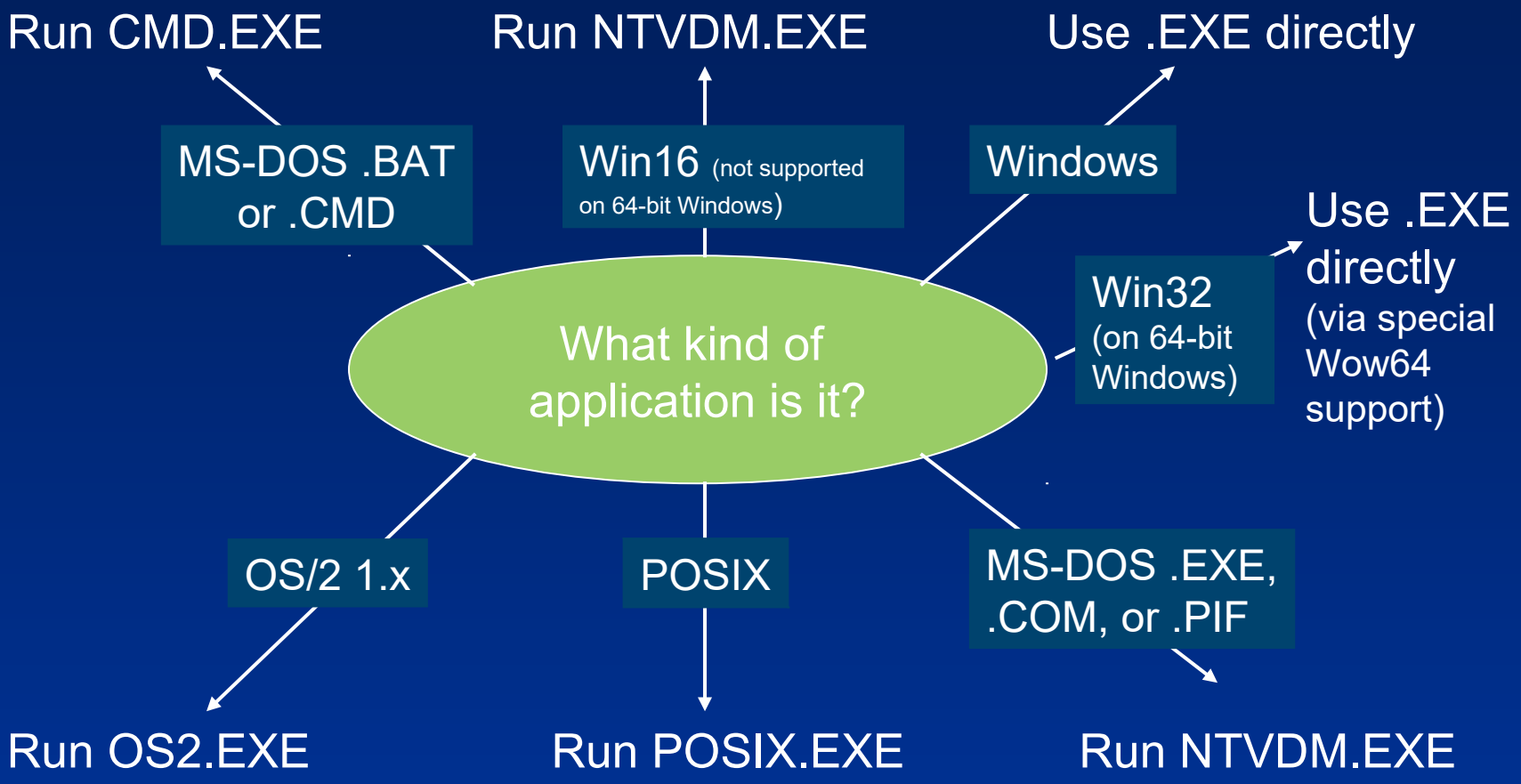
The main Stages Windows follows to create a process



CreateProcess: some notes

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

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 (before introduction of pico processes and WSL in Windows 10)
  - OS2.EXE runs only on Intel systems
  - Multiple MS-DOS apps may share virtual dos machine (VDM)
  - .BAT of .CMD files are interpreted by CMD.EXE
  - Win16 apps may share virtual dos machine (VDM)
    Flags: CREATE_SEPARATE_WOW_VDM
    CREATE_SHARED_WOW_VDM
    Default: HKLM\System...\Control\WOW\DefaultSeparateVDM
  - Sharing of VDM only if apps run on same desktop under same security
- Debugger may be specified under (run instead of app !!)

  \Software\Microsoft\WindowsNT\CurrentVersion\ImageFileExecutionOptions

Process Creation - next Steps...

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

Now: create executive process object via NtCreateProcess

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

Further Steps… (contd.)

- Create Initial Thread and Its Stack and Context
  - NtCreateThread; new thread is suspended until CreateProcess returns
- Notify Windows Subsystem about new process

  KERNEL32.DLL sends message to Windows subsystem (CSRSS) 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 priority class, bookkeeping
  - allocate CSRSS process/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, critical sections Structures
  - Load DLLs, call DLL_PROCESS_ATTACH function
- Debuggee: all threads are suspended
  - Send messsage to process'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

Creation of a Thread

1. The thread count in the process object is incremented.
2. An executive thread block (ETHREAD) is created and initialized.
3. A thread ID is generated for the new thread.
4. The TEB is set up in the user-mode address space of the process.
5. The user-mode thread start address is stored in the ETHREAD.

Creation of a Thread

6. KeInitThread is called to set up the KTHREAD block.
   - The thread's initial and current base priorities are set to the process's base priority, and its affinity and quantum are set to that of the process.
   - KeInitThread allocates a kernel stack for the thread and initializes the machine-dependent hardware context for the thread, including the context, trap, and exception frames.
   - The thread's context is set up so that the thread will start in kernel mode in KiThreadStartup.
   - Finally, KeInitThread sets the thread's state to Initialized and returns to PspCreateThread.
7. Any registered systemwide thread creation notification routines are called.
8. The thread's access token is set to point to the process access token,
   - an access check is made to determine whether the caller has the right to create the thread.
9. Finally, the thread is readied for execution.

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)

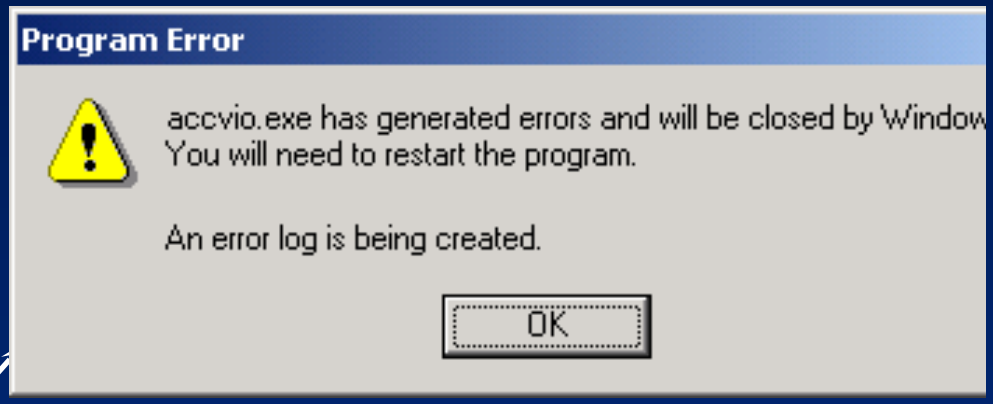
- Registry defines behavior for unhandled exceptions

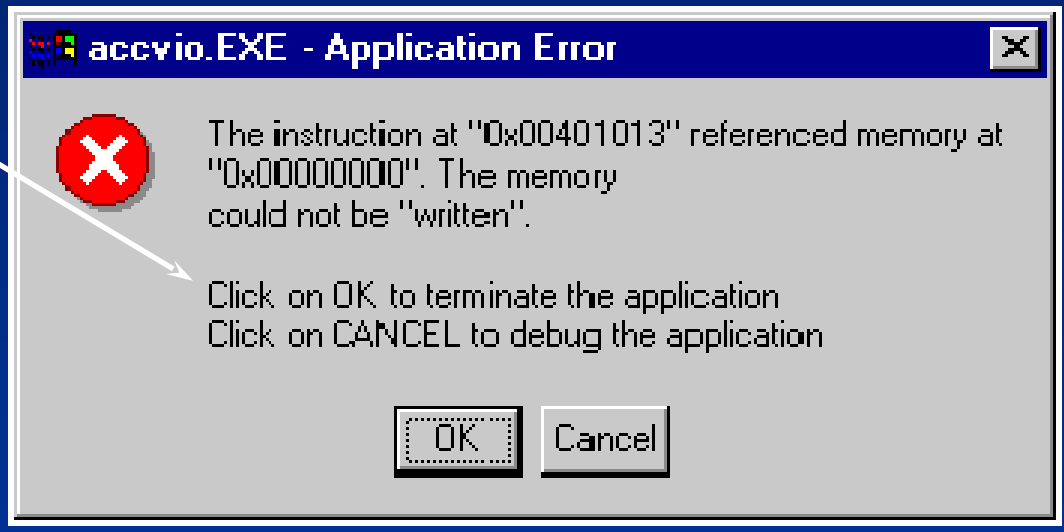
  HKLM\Software\Microsoft
  \Windows NT\CurrentVersion
  \AeDebug

  Debugger=filespec of debugger to run on app crash

  Auto 1=run debugger immediately
  0=ask user first

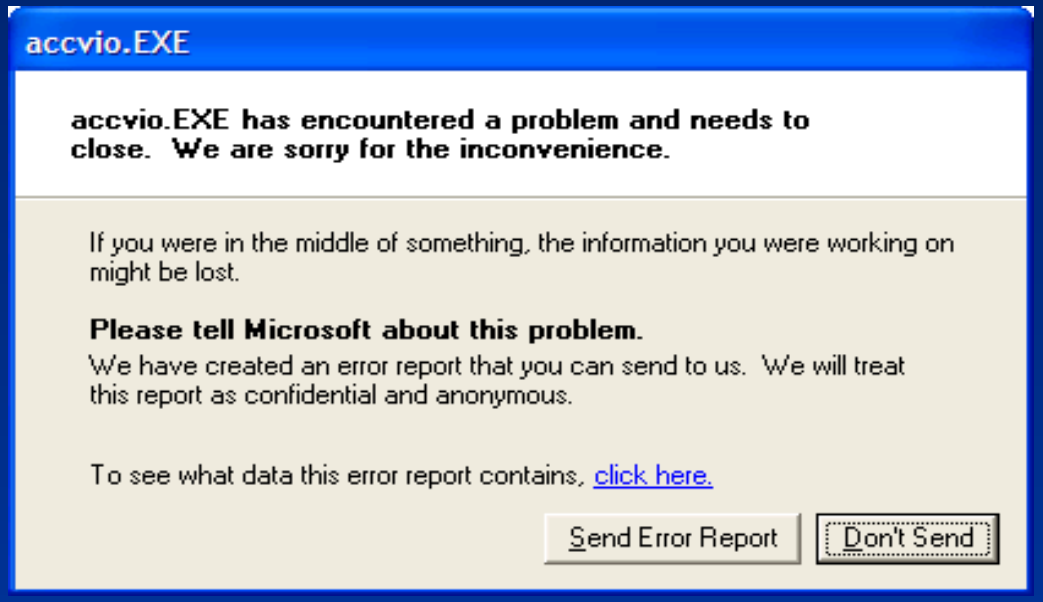
- Default on retail system is
  Auto=1; Debugger=DRWTSN32.EXE
- Default with VC++ is
  Auto=0, Debugger=MSDEV.EXE





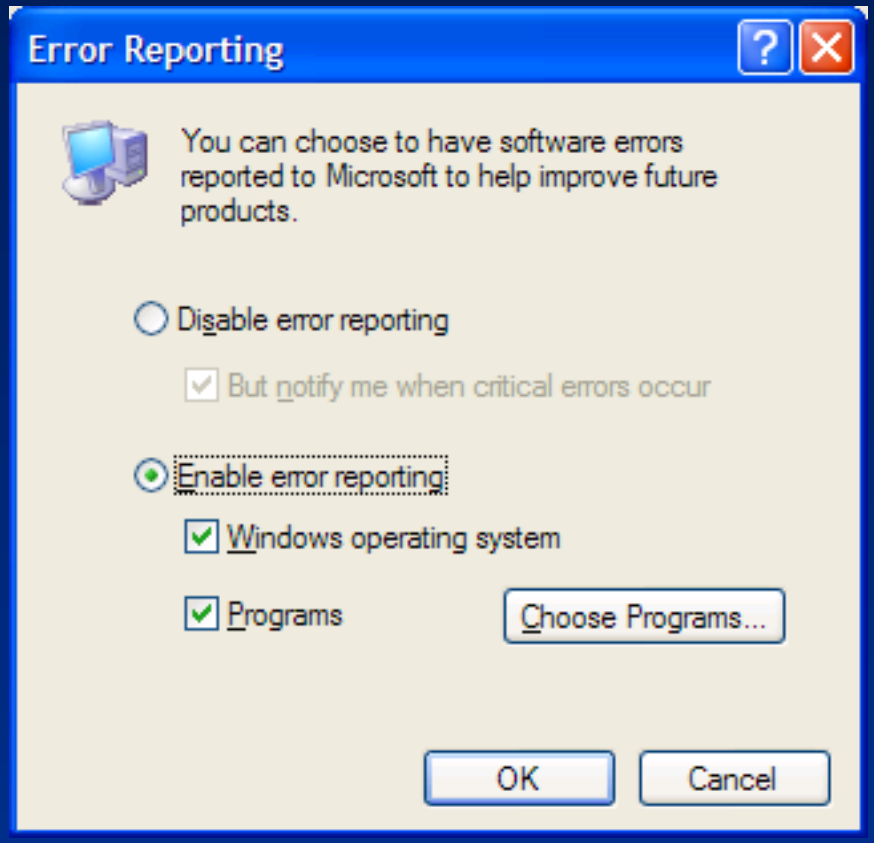
Process Crashes (Windows XP & Windows Server 2003)

- 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



Further Reading

- Pavel Yosifovich, Alex Ionescu, et al., “Windows Internals”, 7th Edition, Microsoft Press, 2017.
  - Chapter 3 – Processes and jobs (from pp. 156)
    - Process internals (from pp. 161)
    - Flow of *CreateProcess* (from pp. 192)
  - Chapter 4 – Threads (from pp. 275)
    - Thread internals (from pp. 276)