Unit 4: Scheduling and Dispatch

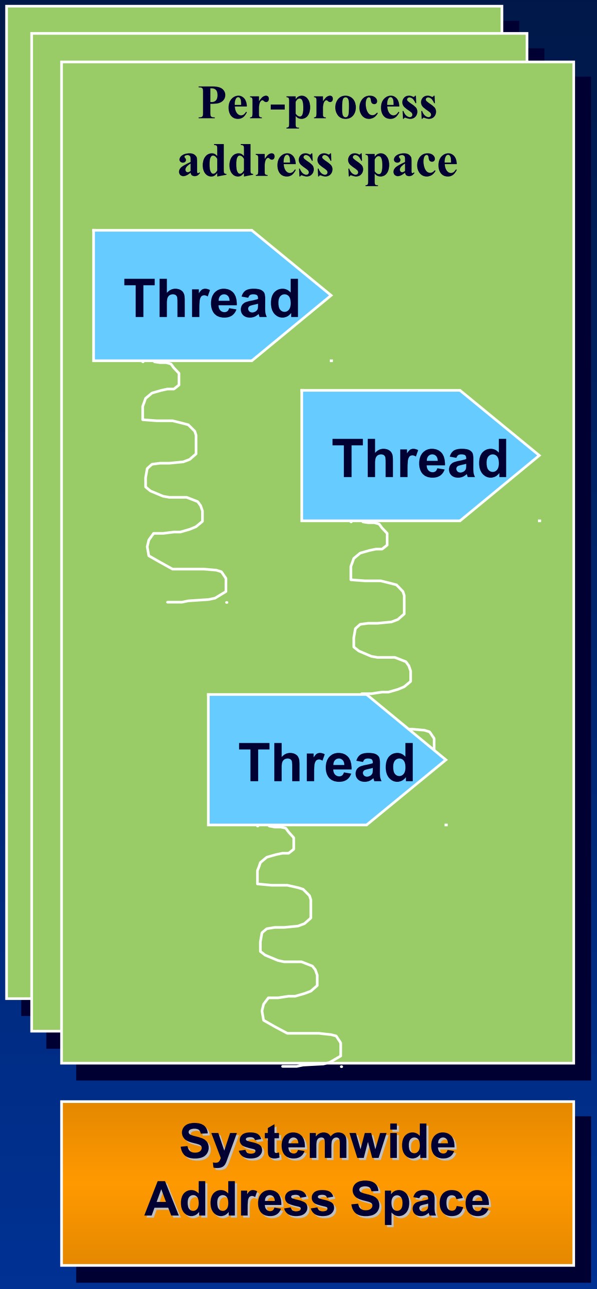
4.2. Windows Processes and Threads

Roadmap for Section 4.2.

- Windows Processes and Threads
- Performance Counters
- Process and Thread Lifetime
- Jobs
- Windows APIs for Process and Thread creation

Windows Processes

- What is a process?
  - Represents an instance of a running program
    - you create a process to run a program
    - starting an application creates a process
  - Process defined by:
    - Address space
    - Resources (e.g. open handles)
    - Security profile (token)
- Every process starts with one thread
  - First thread executes the program's "main" function
    - Can create other threads in the same process
    - Can create additional processes



Windows Threads

- What is a thread?
  - An execution context within a process
  - Unit of scheduling (threads run, processes don’t run)
  - All threads in a process share the same per-process address space
    - Services provided so that threads can synchronize access to shared resources (critical sections, mutexes, events, semaphores)
  - All threads in the system are scheduled as peers to all others, without regard to their “parent” process
- System calls
  - Primary argument to CreateProcess() is image file name (or command line)
  - Primary argument to CreateThread() is a function entry point address

Processes & Threads

- Why divide an application into multiple threads?
  - Perceived user responsiveness, parallel/background execution
    - Examples: Word background print – can continue to edit during print
  - Take advantage of multiple processors
    - On an MP system with *n* CPUs, *n* threads can literally run at the same time
    - Question: given a single threaded application, will adding a 2nd processor make it run faster?
  - Does add complexity
    - Synchronization
    - Scalability well is a different question…
      - # of multiple runnable threads vs # CPUs
      - Having too many runnable threads causes excess context switching

Per-Process Data

- Each process has its own…
  - Virtual address space (including program code, global storage, heap storage, threads’ stacks)
    - processes cannot corrupt each other’s address space by mistake
  - Working set (physical memory “owned” by the process)
  - Access token (includes security identifiers)
  - Handle table for Windows kernel objects
  - Environment strings
  - Command line
  - These are common to all threads in the process, but separate and protected between processes

Per-Thread Data

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

Process and Thread Identifiers

- Every process and every thread has an identifier
- Generically: “client ID” (debugger shows as “CID”)
  - A.K.A. “process ID” and “thread ID”, respectively
  - Process IDs and thread IDs are in the same “number space”
  - These identify the requesting process or thread to its subsystem “server” process (i.e. CSRSS.EXE), in API calls that need the server’s help
- Visible in PerfMon, Task Manager (for processes), Process Viewer (for processes), kernel debugger, etc.
- IDs are unique among all existing processes and threads
  - But might be reused as soon as a process or thread is deleted

Process-Related Performance Counters

Object: Counter	Function
Process:%PrivilegedTime	Percentage of CPU time that the threads in the process have run in kernel mode
Process:%ProcessorTime	Percentage of CPU time that threads have used during specified interval, i.e. %PrivilegedTime + %UserTime
Process:%UserTime	Percentage of CPU time that the threads in the process have run in user mode
Process: ElapsedTime	Total lifetime of process in seconds
Process: ID Process	PID – process ID
Process: ThreadCount	Number of threads in a process

Thread-Related Performance Counters

Object: Counter	Function
Thread:%PrivilegedTime	Percentage of CPU time that the thread was run in kernel mode
Thread:%ProcessorTime	Percentage of CPU time that the thread has used during specified interval, i.e. %PrivilegedTime + %UserTime
Thread:%UserTime	Percentage of CPU time that the thread has run in user mode
Thread: ElapsedTime	Total lifetime of process in seconds
Thread: ID Process	PID – thread's process ID
Thread: ID Thread	TID – thread ID

Thread-Related Performance Counters (contd.)

Object: Counter	Function
Process: Priority Base	Base priority of process: starting priority for all threads within that process
Thread: Priority Base	Base priority of thread: may differ from the thread‘s starting priority
Thread: Priority Current	The thread‘s current dynamic priority
Thread: Start Address	The thread‘s starting virtual address (the same for most threads)
Thread: Thread State	Value from 0 through 7 – current state of thread
Thread: Thread Wait Reason	Value from 0 through 19 – reason why the thread is in wait state

Tools for Obtaining Process & Thread Information

- Many overlapping tools (most show one item the others do not)
- Built-in tools introduced in Windows 2000/XP:
  - Task Manager, Performance Tool
  - Tasklist (new in XP)
- Support Tools
  - pviewer - process and thread details (GUI)
  - pmon - process list (character cell)
  - tlist - shows process tree and thread details (character cell)
- Resource Kit tools:
  - apimon - system call and page fault monitoring (GUI)
  - oh - display open handles (character cell)
  - pviewer - processes and threads and security details (GUI)
  - ptree - display process tree and kill remote processes (GUI)
  - pulist - lists processes and usernames (character cell)
  - pstat - process/threads and driver addresses (character cell)
  - qslice - can show process-relative thread activity (GUI)
- Tools from www.sysinternals.com
  - Process Explorer – super Task Manager – shows open files, loaded DLLs, security info, etc.
  - Pslist – list processes on local or remote systems
  - Ntpmon – shows process/thread create/deletes (and context switches on MP systems only)
  - Listdlls – displays full path of EXE & DLLs loaded in each process

Process Lifetime

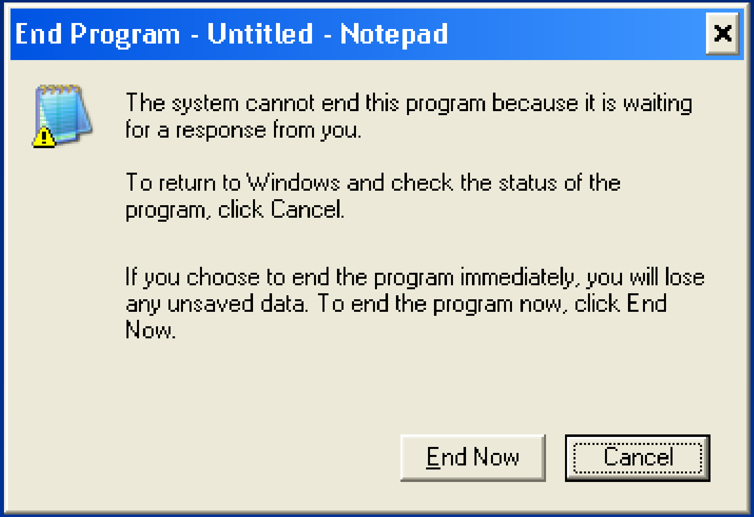
- Created as an empty shell
- Address space created with only ntdll and the main image unless created by POSIX fork()
- Handle table created empty or populated via duplication from parent
- Process is partially destroyed on last thread exit
- Process totally destroyed on last dereference

Thread Lifetime

- Created within a process with a CONTEXT record
  - Starts running in the kernel but has a trap frame to return to user mode
- Threads run until they:
  - The thread returns to the OS
  - ExitThread is called by the thread
  - TerminateThread is called on the thread
  - ExitProcess is called on the process

Why/When Do Processes Exit? (or Terminate?)

- Normal: Application decides to exit (ExitProcess)
  - Usually due to a request from the UI
  - or: C RTL does ExitProcess when primary thread function (main, WinMain, etc.) returns to caller
    - this forces TerminateThread on the process's remaining threads
    - or, any thread in the process can do an explicit ExitProcess



End Program - Untitled - Notepad

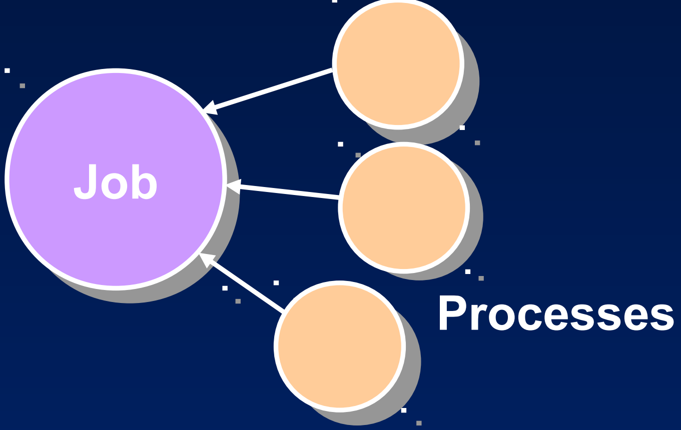
The system cannot end this program because it is waiting for a response from you.

To return to Windows and check the status of the program, click Cancel.

If you choose to end the program immediately, you will lose any unsaved data. To end the program now, click End Now.

- Orderly exit requested from the desktop (ExitProcess)
  - e.g. "End Task" from Task Manager "Tasks" tab
  - Task Manager sends a WM_CLOSE message to the window's message loop…
  - …which should do an ExitProcess (or equivalent) on itself
- Forced termination (TerminateProcess)
  - if no response to "End Task" in five seconds, Task Manager presents End Program dialog (which does a TerminateProcess)
  - or: "End Process" from Task Manager Processes tab
- Unhandled exception
  - Covered in Unit 4.3 (Process and Thread Internals)

Jobs



- Jobs are collections of processes
  - Can be used to specify limits on CPU, memory, and security
  - Enables control over some unique process & thread settings not available through any process or thread system call
    - E.g. length of thread's timeslice (i.e. quantum length)
- How do processes become part of a job?
  - Job object has to be created (CreateJobObject)
  - Then processes are explicitly added (AssignProcessToJob)
    - Processes created by processes in a job automatically are part of the job
      - Unless restricted, processes can "break away" from a job
  - Then quotas and limits are defined (SetInformationJobObject)
    - Examples on next slide…

Job Settings

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

Jobs

- Examples where Windows OS uses jobs:
  - Add/Remove Programs (“ARP Job”)
  - WMI provider
  - RUNAS service (SecLogon) uses jobs to terminate processes at log out
- Process Explorer highlights processes that are members of jobs
  - Color can be configured with Options->Configure Highlighting
  - For processes in a job, click on Job tab in process properties to see details

Process Windows APIs

- CreateProcess
- OpenProcess
- GetCurrentProcessId – returns a global ID (“client” ID)
- GetCurrentProcess – returns a handle
- ExitProcess
- TerminateProcess – no DLL notification
- Get / SetProcessShutdownParameters
- GetExitCodeProcess
- GetProcessTimes
- GetStartupInfo

Windows Thread APIs

- CreateThread
- CreateRemoteThread
- GetCurrentThreadId – returns global ID (“client” ID)
- GetCurrentThread – returns handle
- SuspendThread / ResumeThread
- ExitThread
- TerminateThread – no DLL notification
- GetExitCodeThread
- GetThreadTimes
- Windows 2000 added:
    - OpenThread
    - new thread pooling APIs

Fibers

- Implemented completely in user mode
    - no “internals” ramifications
    - Fibers are still scheduled as threads
    - Fiber APIs allow different execution contexts within a (kernel) thread
        - stack
        - fiber-local storage
        - some registers (essentially those saved and restored for a procedure call)
        - cooperatively “scheduled” within the thread
    - Analogous to threading libraries under many Unix systems
    - Analogous to co-routines in assembly language
    - Allow easy porting of apps that “did their own threads” under other systems

Process Creation

- No parent/child relation in Win32 environment subsystem
- *CreateProcess()* – new process with primary thread

```
BOOL CreateProcess(
    LPCSTR lpApplicationName,
    LPSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation)
```

Parameters

- dwCreationFlags:
  - CREATE_SUSPENDED, DETACHED_PROCESS, CREATE_NEW_CONSOLE, CREATE_NEW_PROCESS_GROUP
- lpStartupInfo:
  - Main window appearance
  - Parent's info: GetStartupInfo
  - hStdIn, hStdOut, hStdErr fields for I/O redirection
- lpProcessInformation:
  - Pointer to handle & ID of new process/thread

```
typedef struct _PROCESS_INFORMATION {
            HANDLE hProcess;
            HANDLE hThread;
            DWORD dwProcessId;
            DWORD dwThreadId;
} PROCESS_INFORMATION;
```

UNIX & Win32 comparison

- Windows API has no equivalent to fork()
- CreateProcess() similar to fork() + exec() in child
- UNIX $PATH vs. lpCommandLine argument
  - Win32 searches in directory of current process image; in current directory; in Windows system directory (GetSystemDirectory); in Windows directory (GetWindowsDirectory); in directories given in PATH
- Windows API has no parent/child relations for processes
- No UNIX process groups in Windows API
  - Limited form: group = processes to receive a console event

Windows API Thread Creation

```
HANDLE CreateThread (
     LPSECURITY_ATTRIBUTES lpsa,
     DWORD cbStack,
     LPTHREAD_START_ROUTINE lpStartAddr,
     LPVOID lpvThreadParm,
     DWORD fdwCreate,
     LPDWORD lpIDThread)
```

cbStack == 0: thread‘s stack size defaults to primary thread‘s size

- lpstartAddr points to function declared as

```
DWORD WINAPI ThreadFunc(LPVOID)
```

- lpvThreadParm is 32-bit argument
- LPIDThread points to DWORD that receives thread ID non-NULL pointer !

Exiting and Terminating a Process

- Shared resources must be freed before exiting
  - Mutexes, semaphores, events
  - Use structured exception handling
- But:
  _finally, _except handlers are not executed on ExitProcess;
- no SEH on TerminateProcess

```
VOID ExitProcess(
        UINT uExitCode);
BOOL TerminateProcess(
        HANDLE hProcess,
        UINT uExitCode);
BOOL GetExitCodeProcess(
        HANDLE hProcess,
        LPDWORD lpExitCode);
```

Windows API Thread Termination

```
VOID ExitThread( DWORD devExitCode )
```

- When the last thread in a process terminates, the process itself terminates
(TerminateThread() does not execute final SEH)
- Thread continues to exist until last handle is closed (CloseHandle())

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

- Returns exit code or STILL_ACTIVE

Suspending and Resuming Threads

- Each thread has suspend count
- Can only execute if suspend count == 0
- Thread can be created in suspended state

```
DWORD ResumeThread (HANDLE hThread)
DWORD SuspendThread(HANDLE hThread)
```

- Both functions return suspend count or 0xFFFFFFFF on failure

Synchronization & Remote Threads

- WaitForSingleObject() and WaitForMultipleObjects() with thread handles as arguments perform thread synchronization
  - Waits for thread to become signaled
  - ExitThread(), TerminateThread(), ExitProcess() set thread objects to signaled state
- CreateRemoteThread() allows creation of thread in another process
  - Not implemented in Windows 9x
- C library is not thread-safe; use libcmt.lib instead
  - #define _MT before any include
  - Use _beginthreadex/_endthreadex instead of Create/ExitThread

Further Reading

- Pavel Yosifovich, Alex Ionescu, et al., “Windows Internals”, 7th Edition, Microsoft Press, 2017.
  - Chapter 3 – Processes and jobs (from pp. 156)
  - Chapter 4 – Threads (from pp. 275)
- Jeffrey Richter, Programming Applications for Microsoft Windows, 4th Edition, Microsoft Press, September 1999.
  - Chapter 4 - Processes
  - Chapter 5 - Jobs
  - Chapter 6 - Thread Basics