Academic Club

Windows Internals Tour

Windows Processes, Threads and Memory

Andrea Dell'Amico – Microsoft Student Partner

andrea.dellamico@msptechrep.com

Academic Tour 2011 imagine cup

Microsoft

Roadmap

Processes and Threads

- Processes, Threads, Jobs and Fibers
- Processes and Threads Data Structures
- Create Processes
- Scheduling

Memory

- Memory Manager Features and Components
- Virtual Address Space Allocation
- Shared Memory and Memory-Mapped Files
- Physical Memory Limits

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

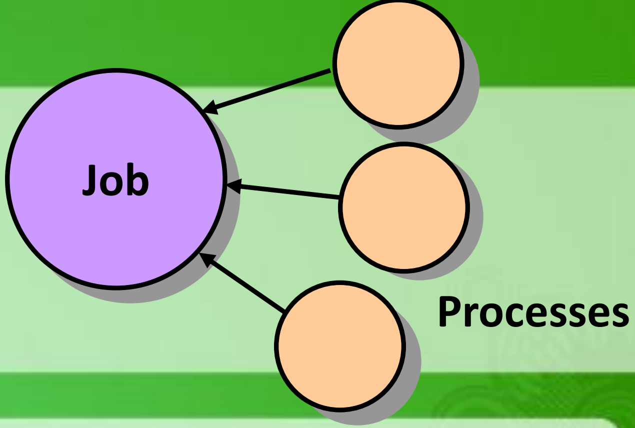
Windows Threads

- 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

Processes & Threads

- Why divide an application into multiple threads?
  - Perceived user responsiveness, parallel/background execution
  - Take advantage of multiple processors
    - On an MP system with *n* CPUs, *n* threads can literally run at the same time
  - Does add complexity
    - Synchronization
    - Scalability

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 time slice
- 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 threads in job can run on
  - Security restrictions: limits what processes can do

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

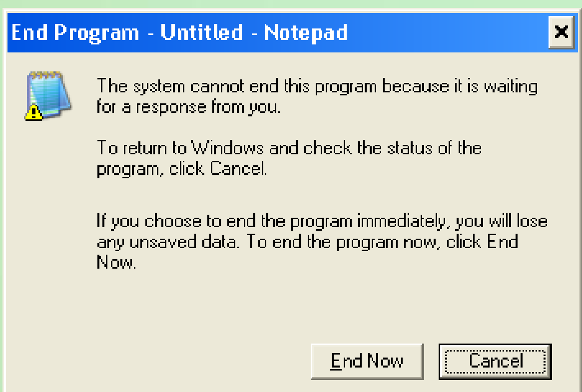
- Created within a process with a CONTEXT record
  - Starts running in the kernel but has a trap frame to return to user mode
- Threads run until they:
  - The thread returns to the OS
  - ExitThread is called by the thread
  - TerminateThread is called on the thread
  - ExitProcess is called on the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

Academic Tour 2011 imagine cup

Microsoft

Why 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



- 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

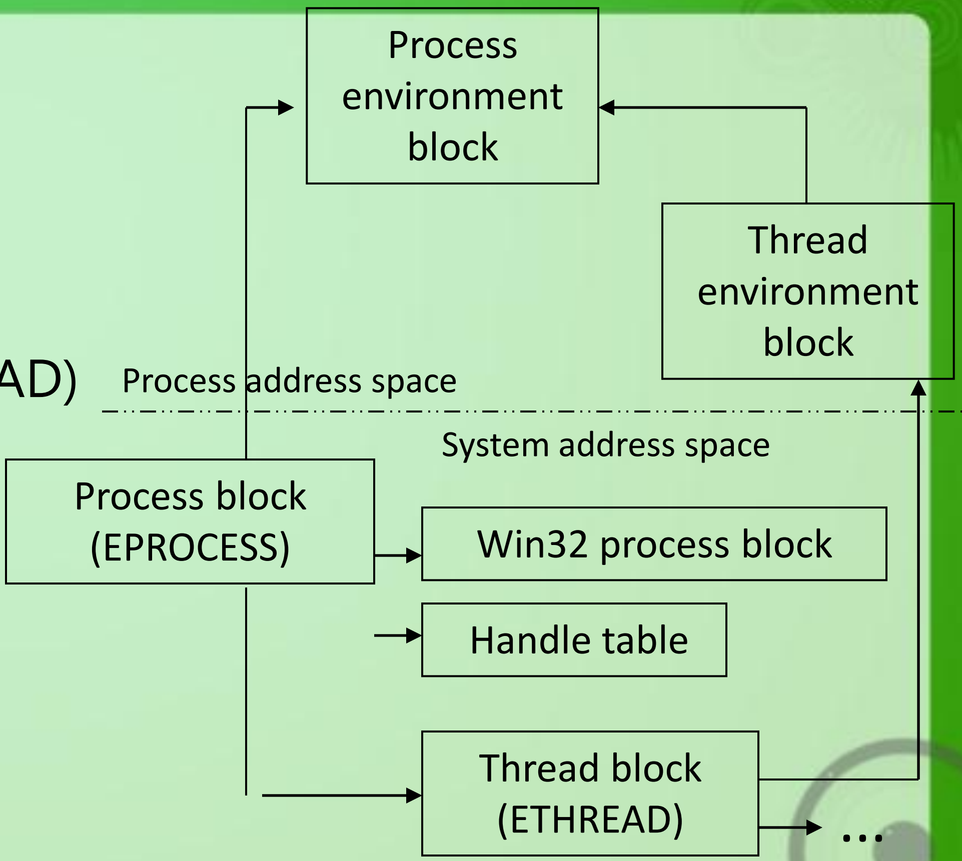
Fibers

- Implemented completely in user mode
  - no “internals” ramifications
  - Fibers are still scheduled as threads
  - Fiber APIs allow different execution contexts within a 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

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



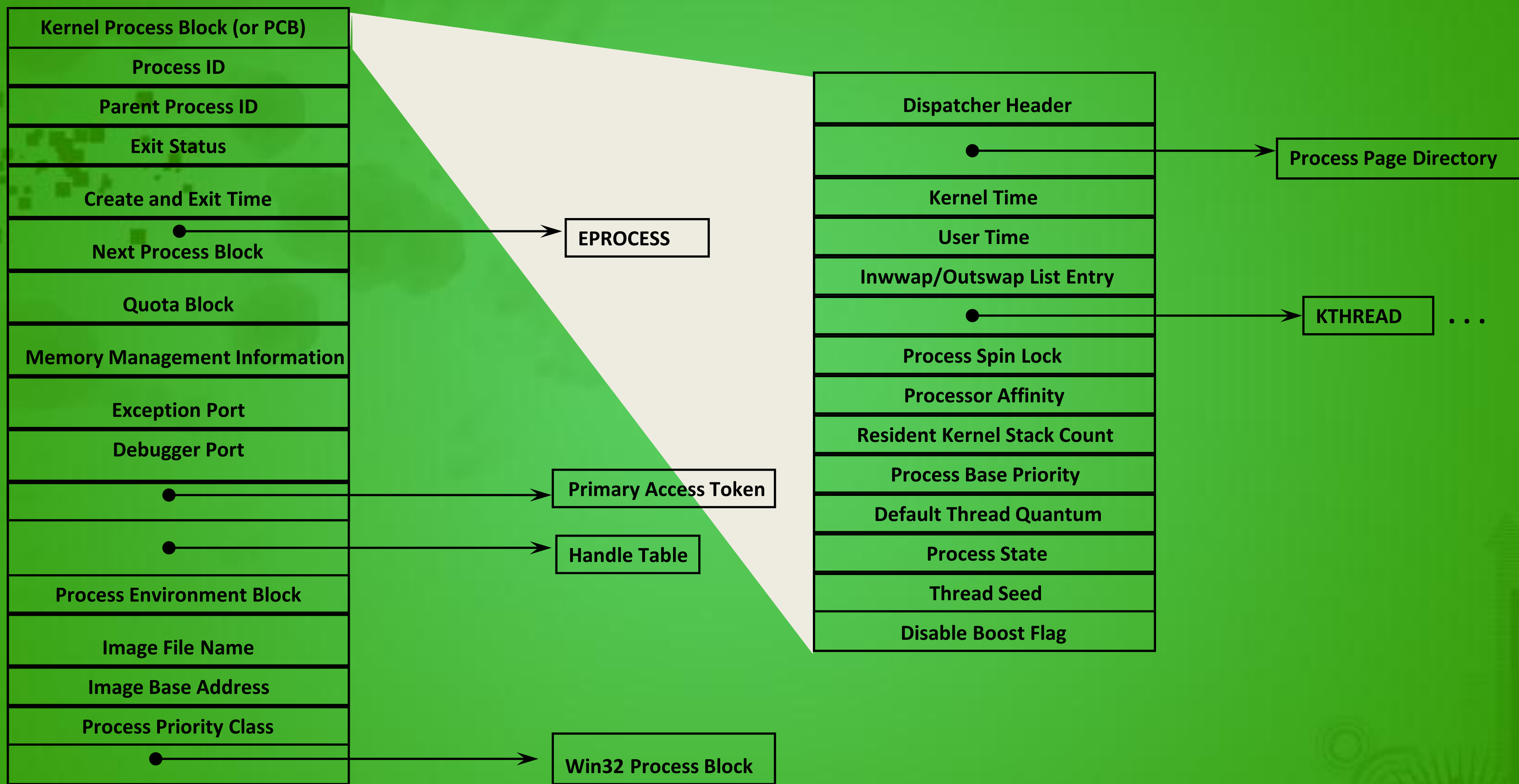
Process

- 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

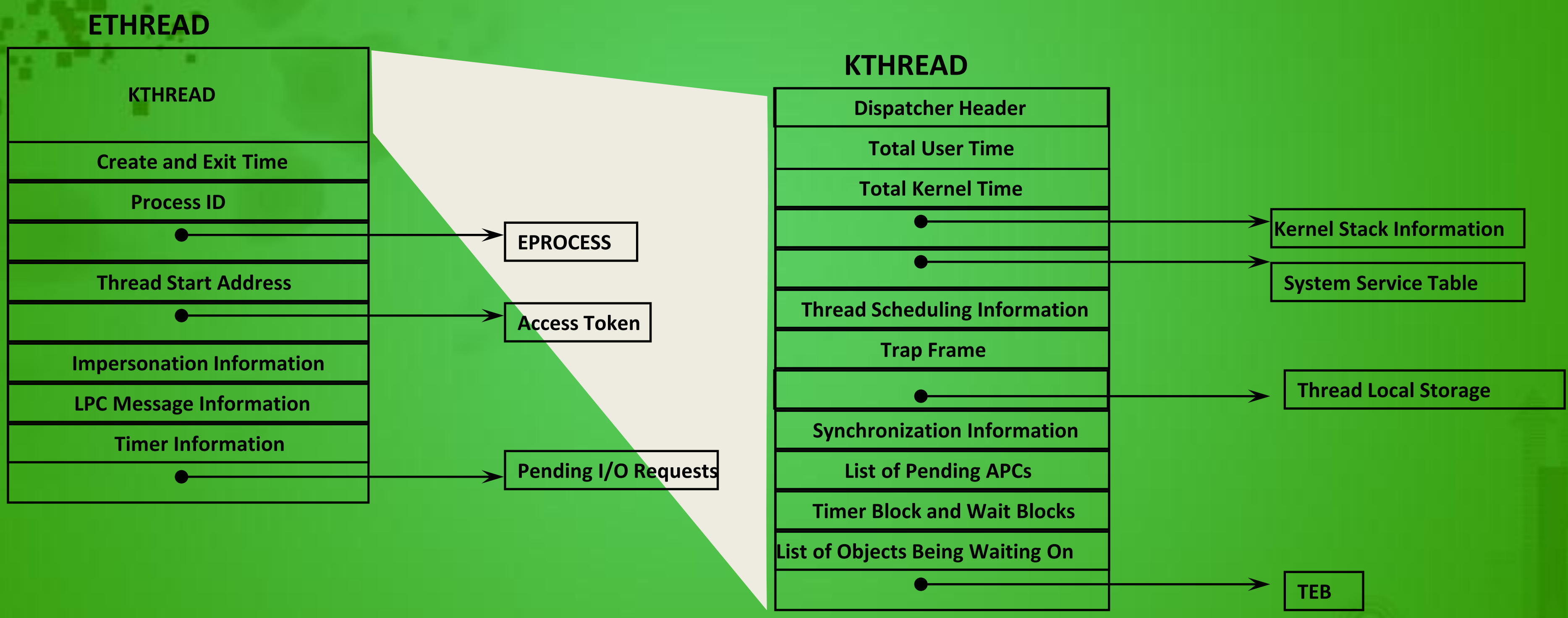
Thread

- 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

Process Block Layout

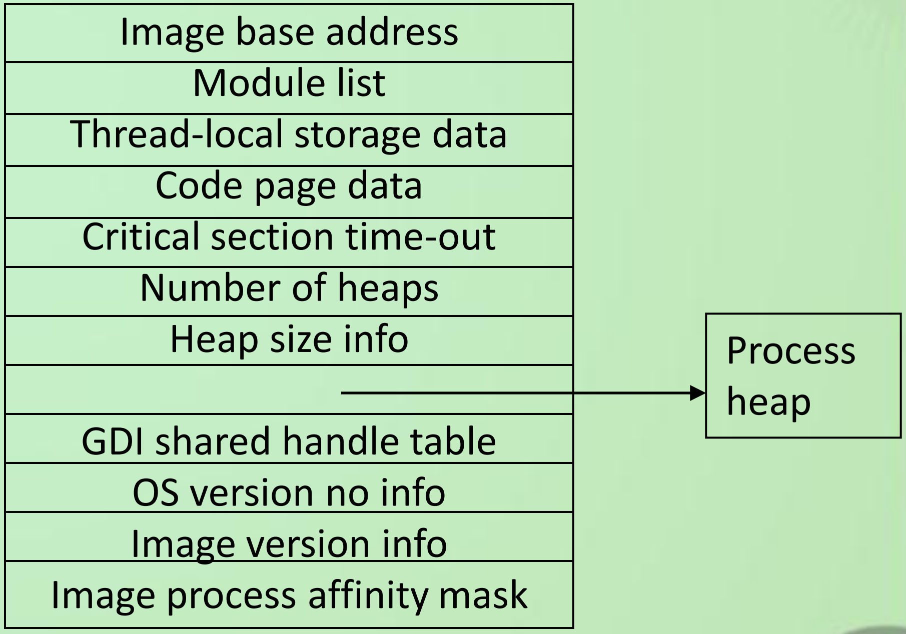


Thread Block



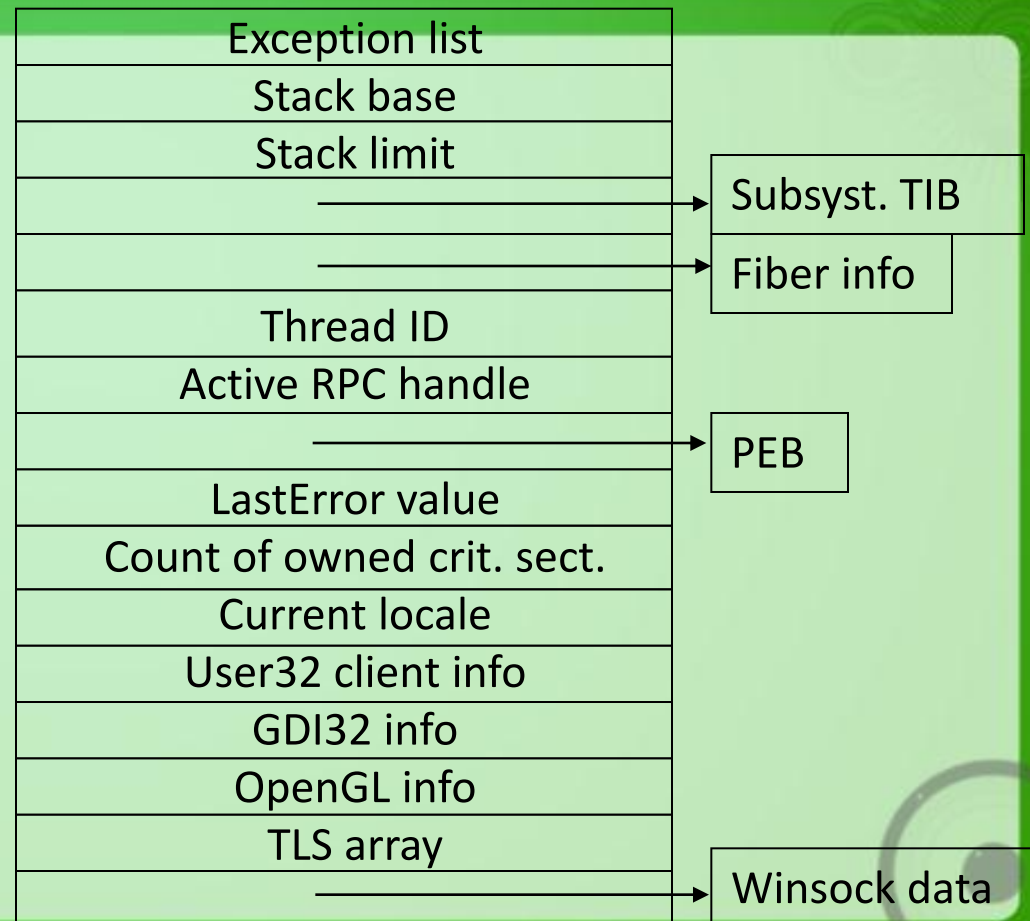
Process Environment Block

- 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



Process Creation

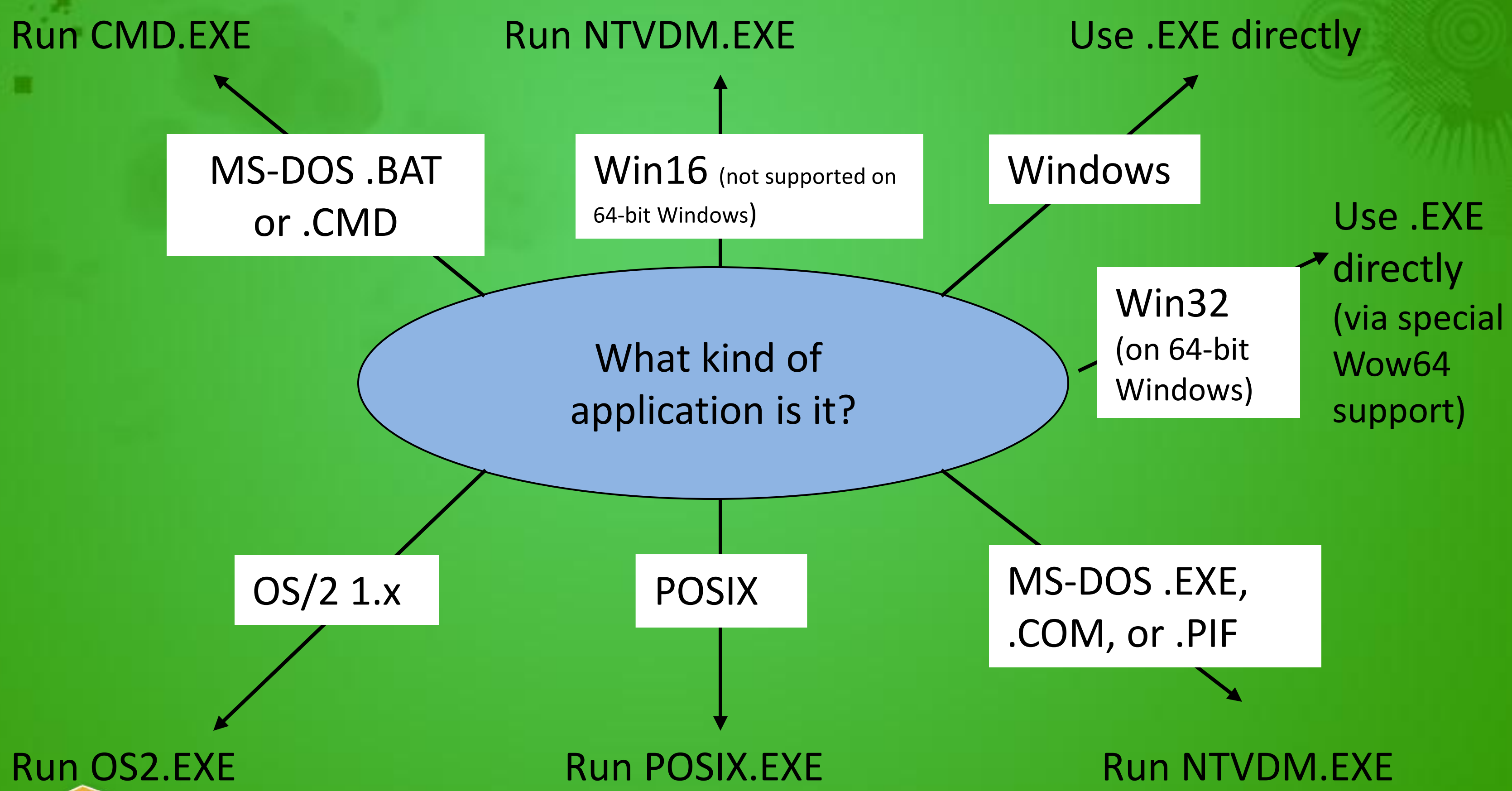
- No parent/child relation in Win32
- *CreateProcess()* – new process with primary thread

```
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)
```

UNIX & Win32 comparison

- Windows API has no equivalent to fork()
- CreateProcess() similar to fork()/exec()
- UNIX $PATH vs. lpCommandLine argument
  - Win32 searches in dir of curr. Proc. Image; in curr. Dir.; in Windows system dir. (GetSystemDirectory); in Windows dir. (GetWindowsDirectory); in dir. 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

Opening the image to be executed



If executable has no Windows format…

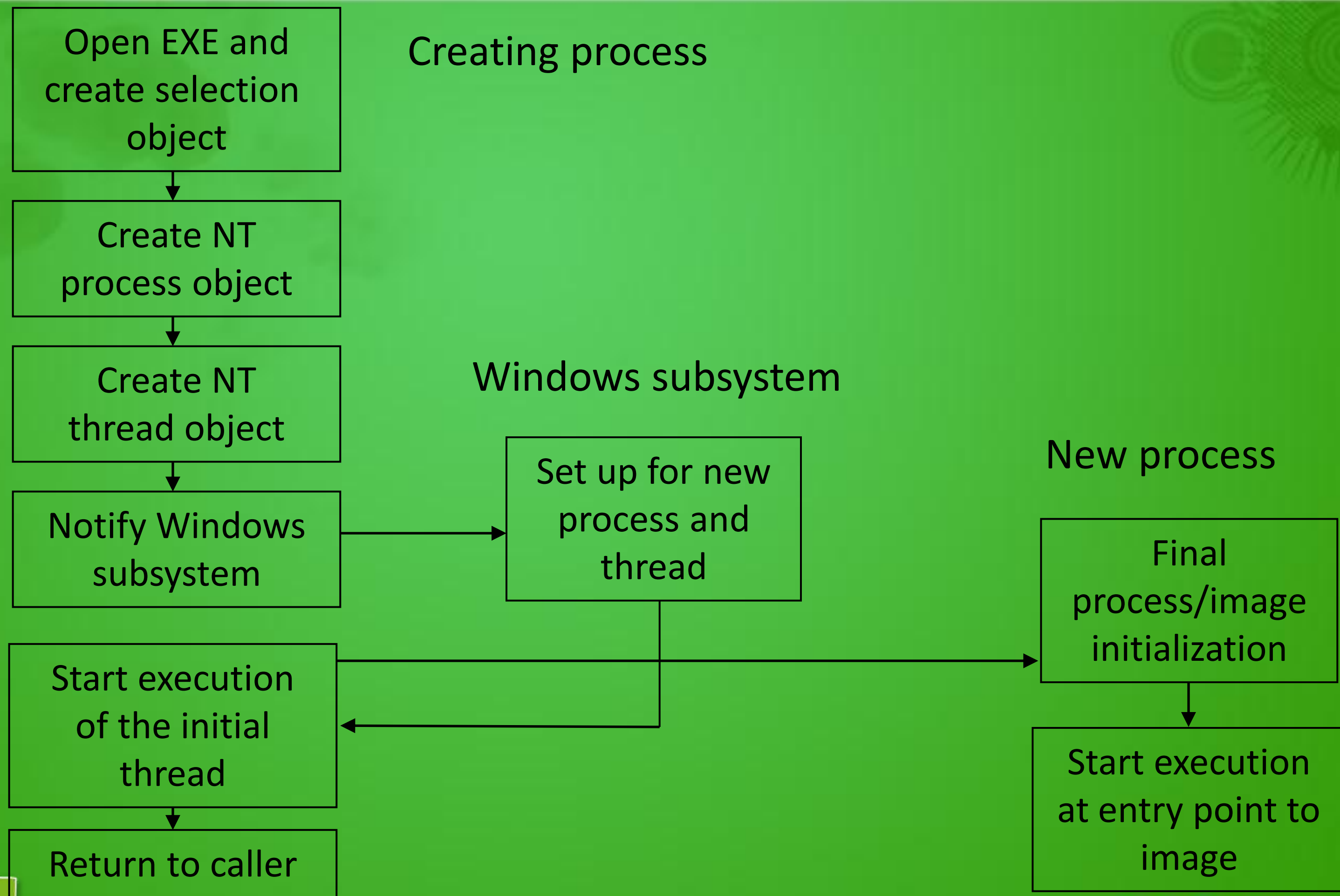
- CreateProcess uses Windows „support image“
- No way to create non-Windows processes directly
  - OS2.EXE runs only on Intel systems
  - Multiple MS-DOS apps may share virtual dos machine
  - .BAT of .CMD files are interpreted by CMD.EXE
  - Win16 apps may share virtual dos machine (VDM)
    Flags: 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

Flow of CreateProcess()

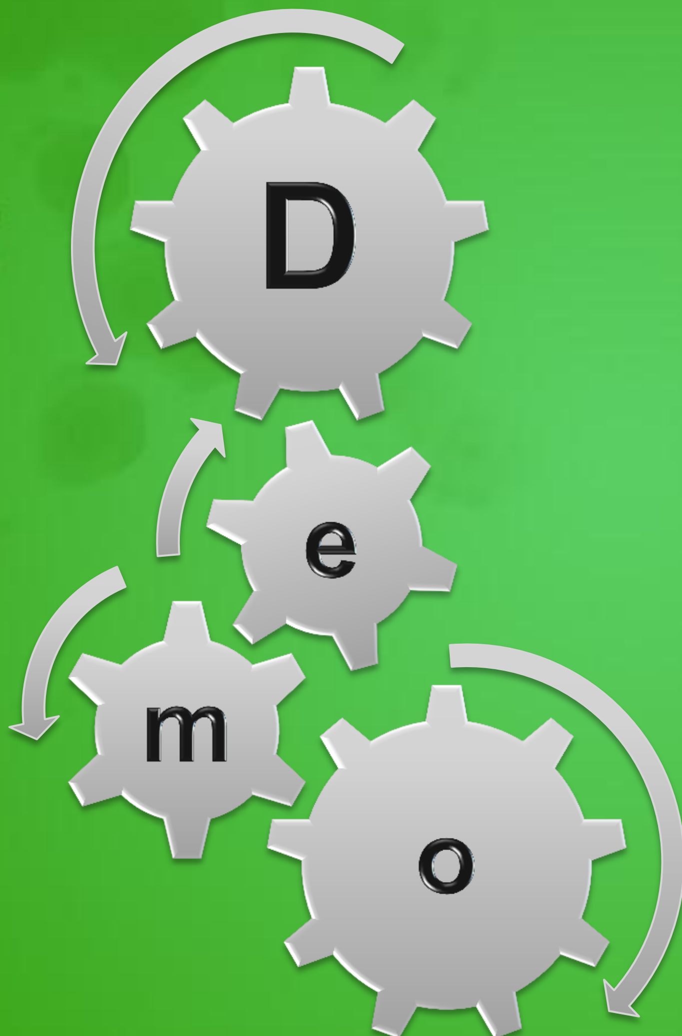
1. Open the image file (.EXE) to be executed inside the process
2. Create Windows NT executive process object
3. Create initial thread (stack, context, Win NT executive thread object)
4. Notify Windows subsystem of new process so that it can set up for new proc.& thread
5. Start execution of initial 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-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



Process Explorer

Image File Execution Options

Task Manager

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.

Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes/threads that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process/thread
- Waiting time – amount of time a process/thread has been waiting in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced, **not** output (i.e.; the hourglass)

How does the Windows scheduler relate to the issues discussed:

- Priority-driven, preemptive scheduling system
- Highest-priority runnable thread always runs
- 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 Scheduling Principles

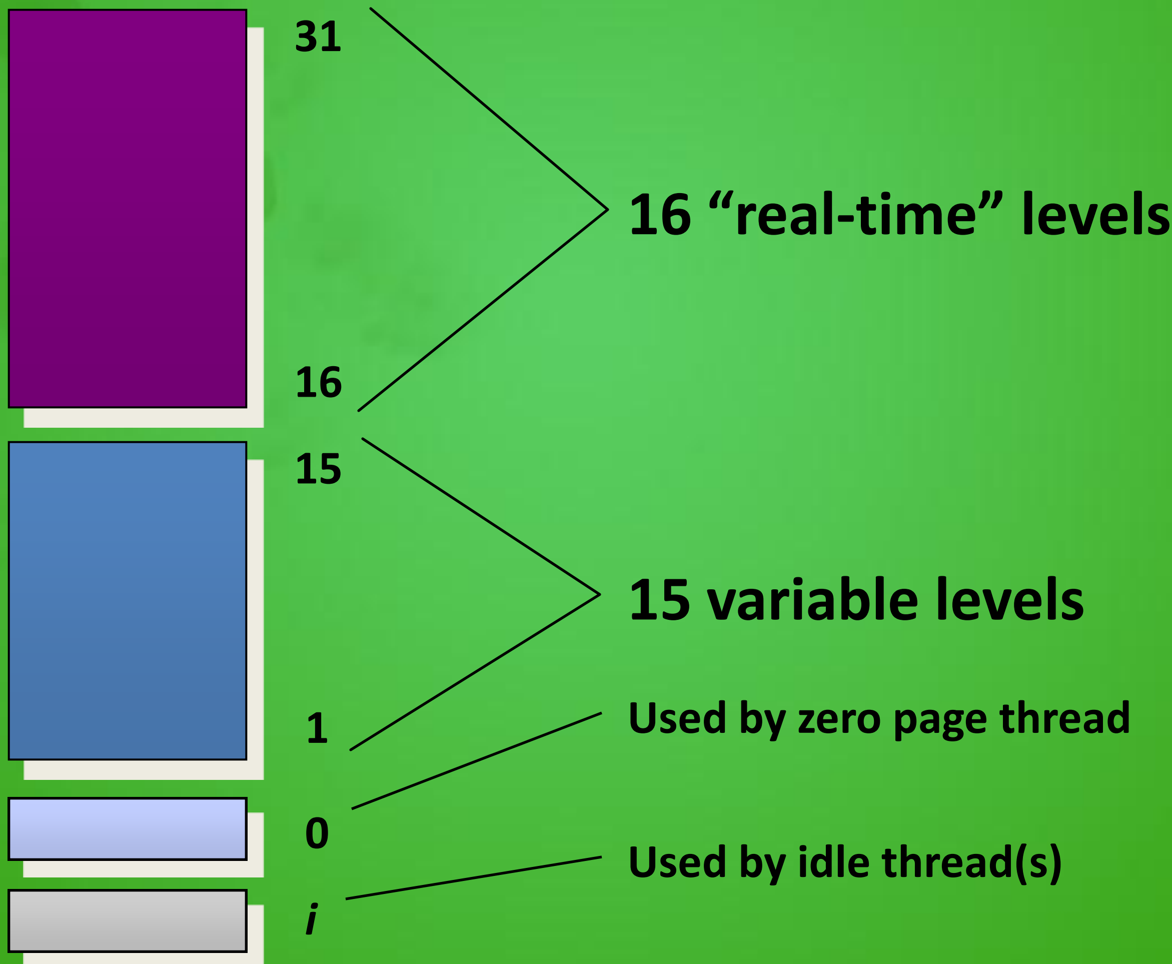
- 32 priority levels
- Threads within same priority are scheduled following the Round-Robin policy
- Non-Realtime Priorities are adjusted dynamically
  - Priority elevation as response to certain I/O and dispatch events
  - Quantum stretching to optimize responsiveness
- Realtime priorities (i.e.; > 15) are assigned statically to threads

Windows vs. NT Kernel Priorities

		Win32 Process Classes					
		Realtime	High	Above Normal	Normal	Below Normal	Idle
Win32 Thread Priorities	Time-critical	31	15	15	15	15	15
	Highest	26	15	12	10	8	6
	Above-normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Below-normal	23	12	9	7	5	3
	Lowest	22	11	8	6	4	2
	Idle	16	1	1	1	1	1

- Table shows base priorities ("current" or "dynamic" thread priority may be higher if base is < 15)
- Many utilities (such as Process Viewer) show the "dynamic priority" of threads rather than the base (Performance Monitor can show both)
- Drivers can set to any value with KeSetPriorityThread

Kernel: Thread Priority Levels



Special Thread Priorities

- Idle threads -- one per CPU
  - When no threads want to run, Idle thread “runs”
    - Not a real priority level - appears to have priority zero, but actually runs “below” priority 0
    - Provides CPU idle time accounting (unused clock ticks are charged to the idle thread)
  - Loop:
    - Calls HAL to allow for power management
    - Processes DPC list
    - Dispatches to a thread if selected
- Zero page thread -- one per NT system
  - Zeroes pages of memory in anticipation of “demand zero” page faults
  - Runs at priority zero (lower than any reachable from Windows)
  - Part of the “System” process (not a complete process)

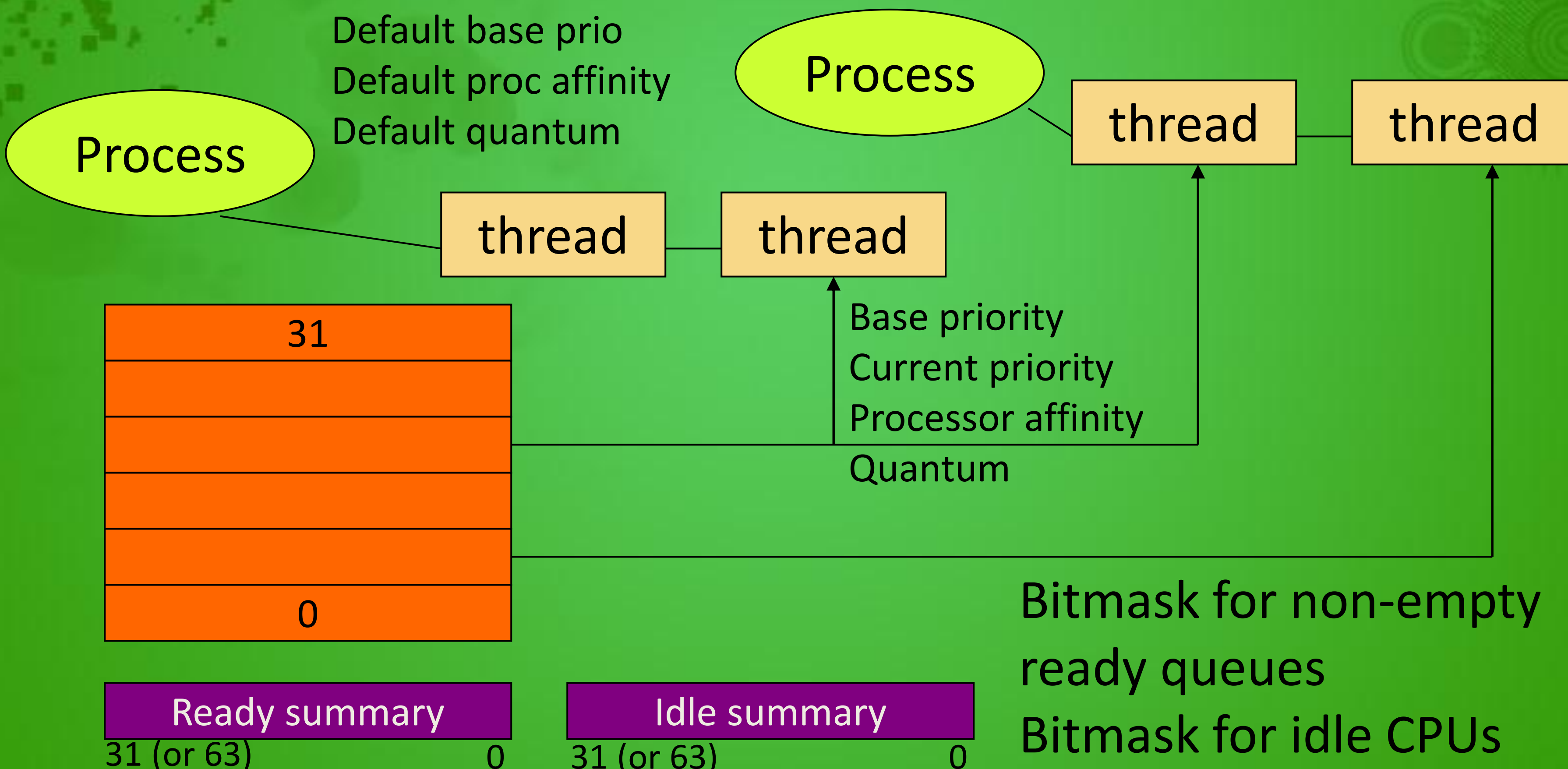
Single Processor Thread Scheduling

- Priority driven, preemptive
  - 32 queues (FIFO lists) of "ready" threads
  - UP: highest priority thread always runs
  - MP: One of the highest priority runnable thread will be running somewhere
  - No attempt to share processor(s) "fairly" among processes, only among threads
    - Time-sliced, round-robin within a priority level
- Event-driven; no guaranteed execution period before preemption
  - When a thread becomes Ready, it either runs immediately or is inserted at the tail of the Ready queue for its current (dynamic) priority

Thread Scheduling

- No central scheduler!
  - i.e. there is no always-instantiated routine called “the scheduler”
  - The “code that does scheduling” is not a thread
  - Scheduling routines are simply called whenever events occur that change the Ready state of a thread
  - Things that cause scheduling events include:
    - interval timer interrupts (for quantum end)
    - interval timer interrupts (for timed wait completion)
    - other hardware interrupts (for I/O wait completion)
    - one thread changes the state of a waitable object upon which other thread(s) are waiting
    - a thread waits on one or more dispatcher objects
    - a thread priority is changed
- Based on doubly-linked lists (queues) of Ready threads
  - Nothing that takes “order-*n* time” for *n* threads

Scheduling Data Structures
Dispatcher Database

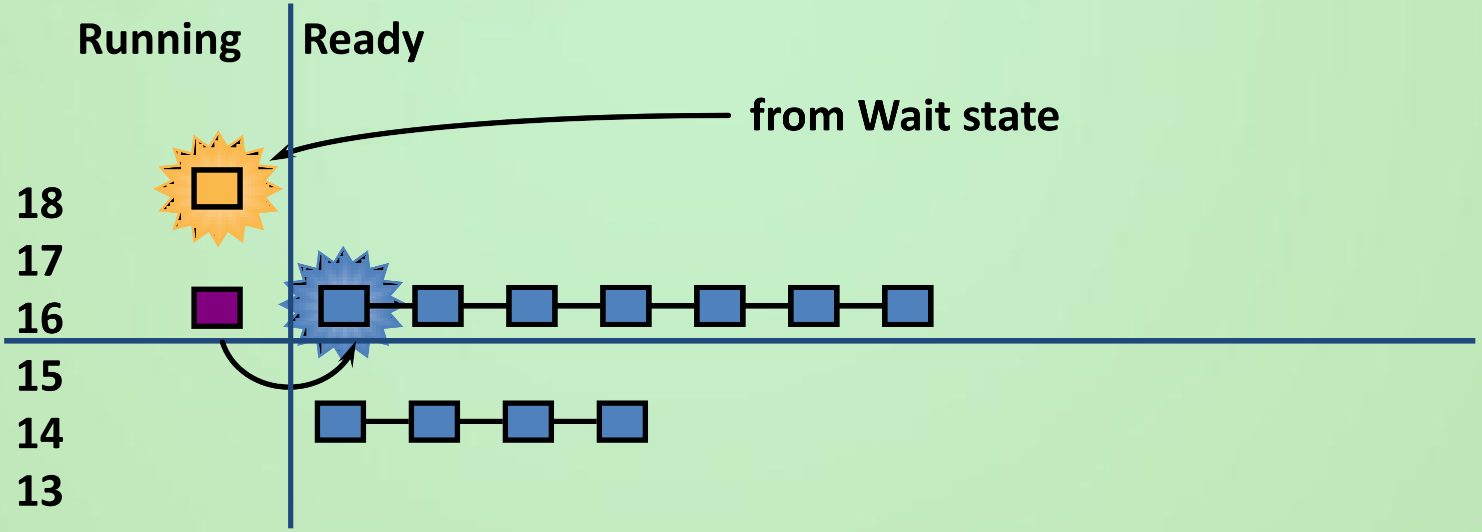


Scheduling Scenarios

- Preemption
  - A thread becomes Ready at a higher priority than the running thread
  - Lower-priority Running thread is preempted
  - Preempted thread goes back to <u>head</u> of its Ready queue
    - <u>action:</u> pick lowest priority thread to preempt
- Voluntary switch
  - Waiting on a dispatcher object
  - Termination
  - Explicit lowering of priority
    - <u>action:</u> scan for next Ready thread (starting at your priority & down)
- Running thread experiences quantum end
  - Priority is decremented unless already at thread base priority
  - Thread goes to <u>tail</u> of ready queue for its new priority
  - May continue running if no equal or higher-priority threads are Ready
    - <u>action:</u> pick next thread at same priority level

Scheduling Scenarios
Preemption

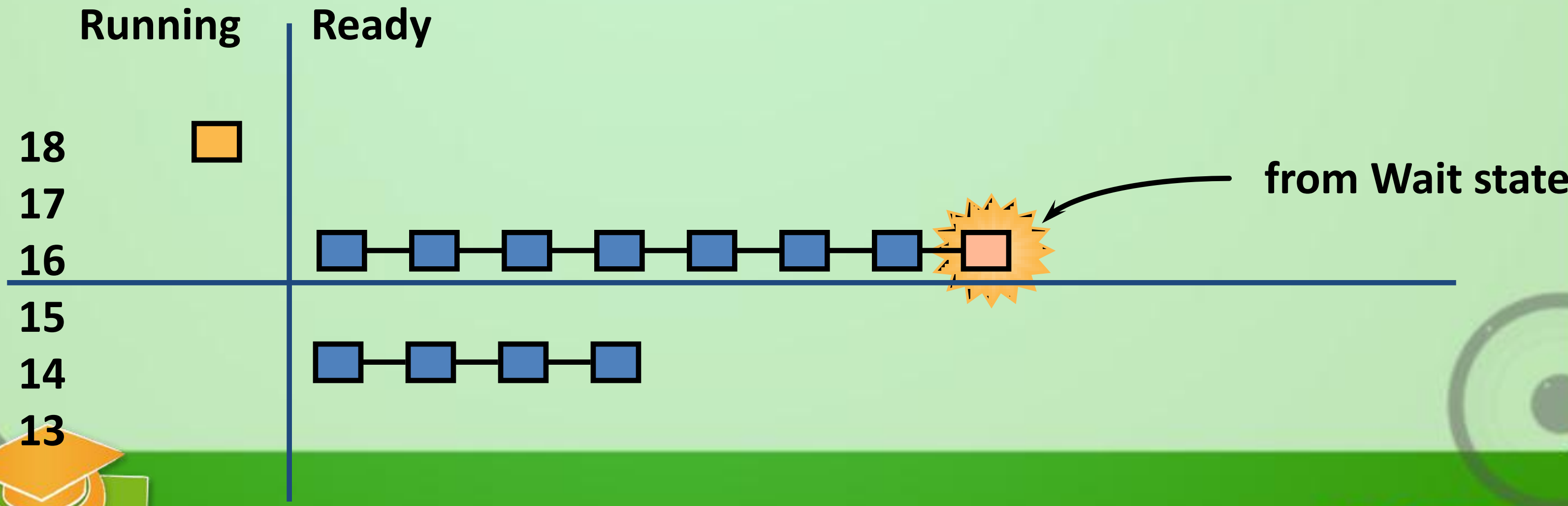
- Preemption is strictly event-driven
  - does not wait for the next clock tick
  - no guaranteed execution period before preemption
  - threads in kernel mode may be preempted (unless they raise IRQL to >= 2)



- A preempted thread goes back to the head of its ready queue

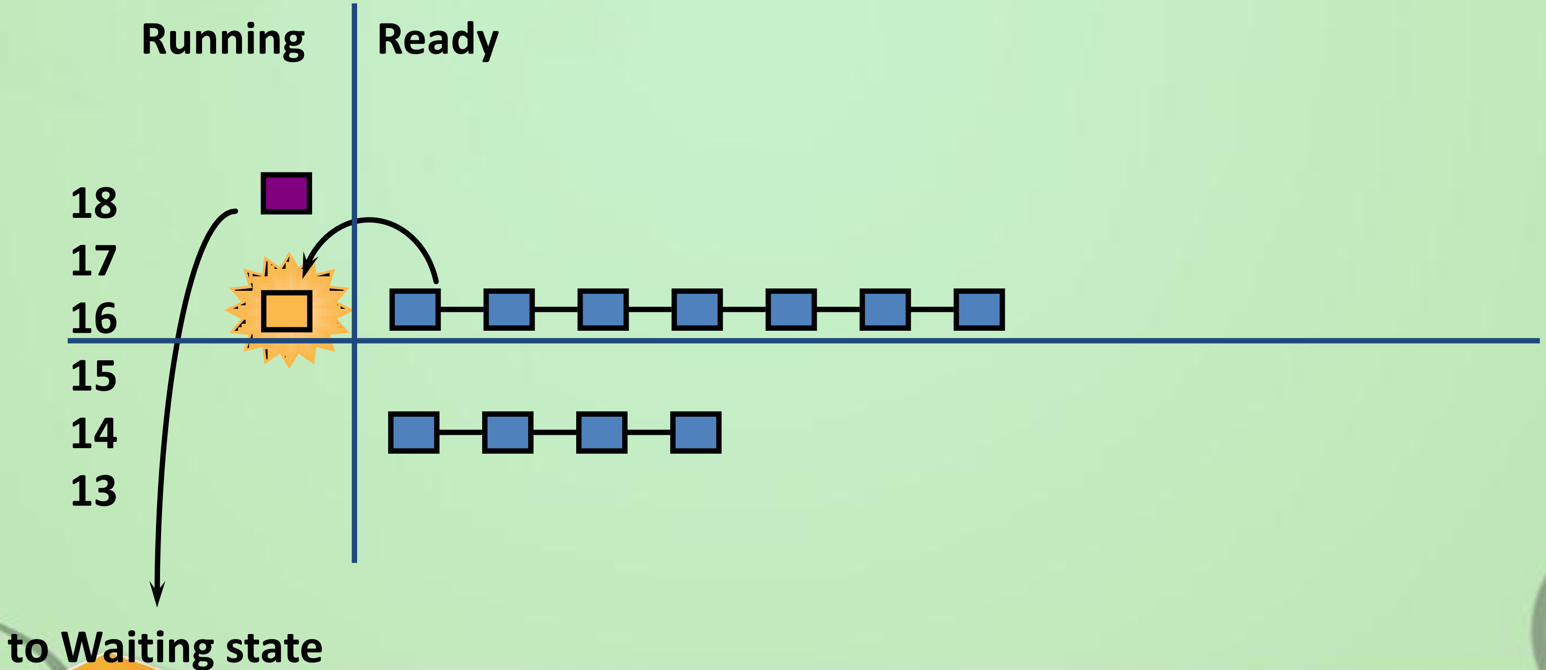
Scheduling Scenarios
Ready after Wait Resolution

- If newly-ready thread is not of higher priority than the running thread…
- …it is put at the tail of the ready queue for its current priority
  - If priority >=14 quantum is reset (t.b.d.)
  - If priority <14 and you're about to be boosted and didn't already have a boost, quantum is set to process quantum - 1



Scheduling Scenarios
Voluntary Switch

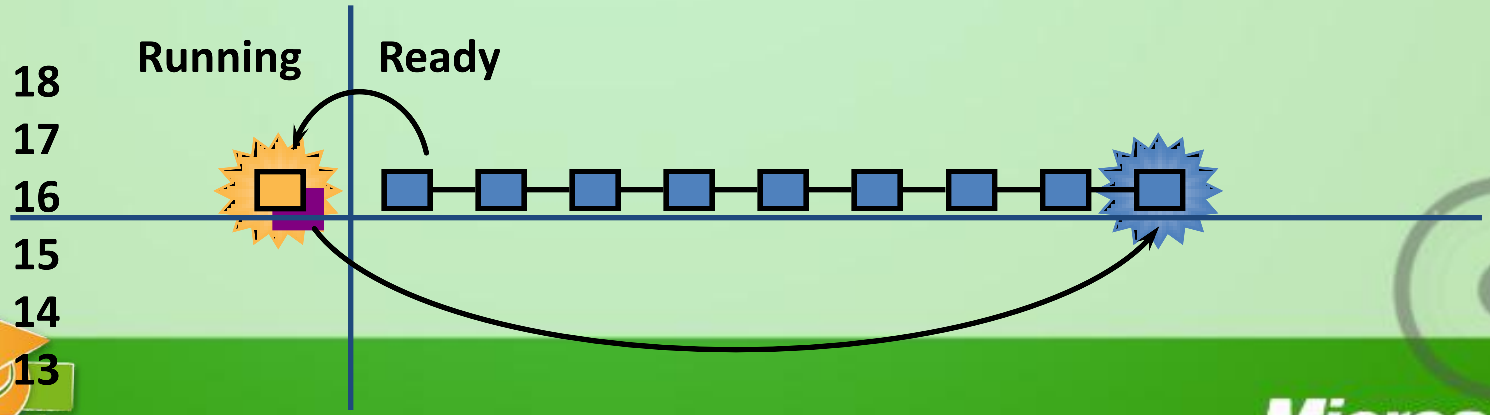
- When the running thread gives up the CPU…
- …Schedule the thread at the head of the next non-empty “ready” queue



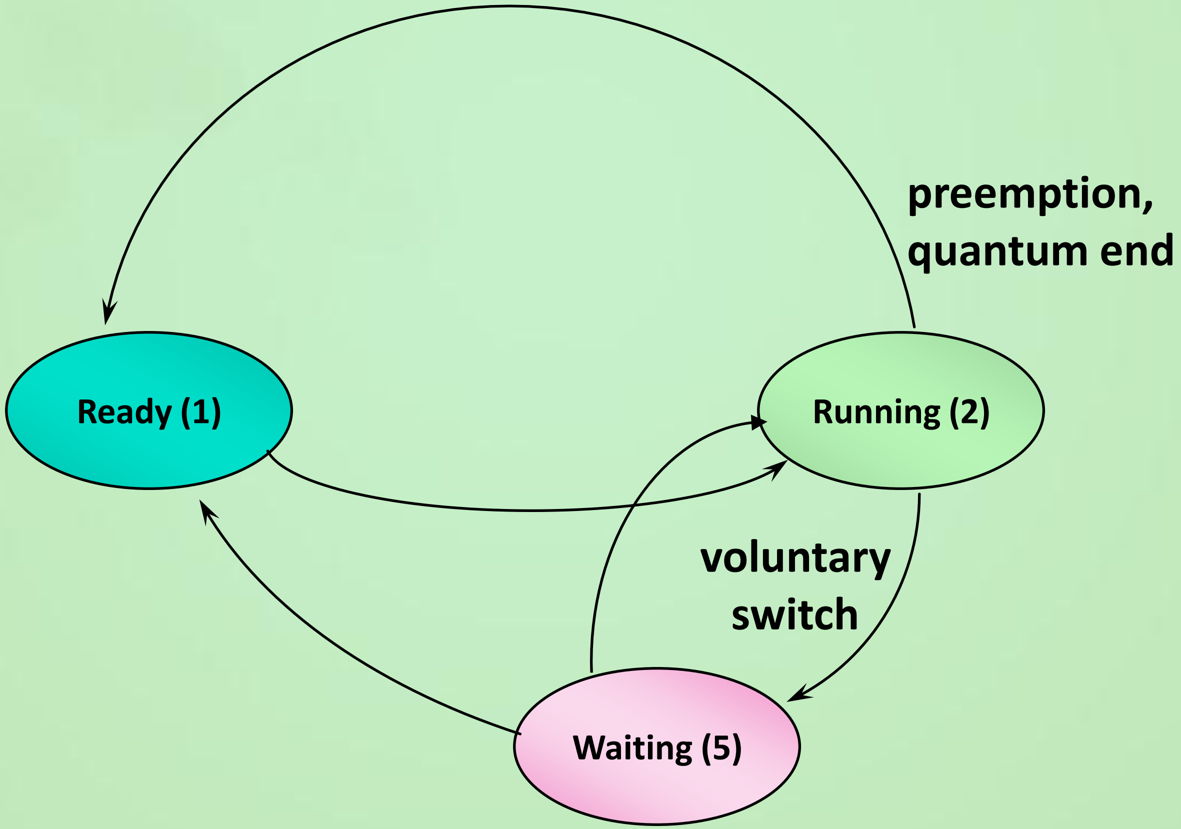
Scheduling Scenarios

Quantum End ("time-slicing")

- When the running thread exhausts its CPU quantum, it goes to the end of its ready queue
  - Applies to both real-time and dynamic priority threads, user and kernel mode
    - Quantums can be disabled for a thread by a kernel function
  - Default quantum on Professional is 2 clock ticks, 12 on Server
    - standard clock tick is 10 msec; might be 15 msec on some MP Pentium systems
  - if no other ready threads at that priority, same thread continues running (just gets new quantum)
  - if running at boosted priority, priority decays by one at quantum end (described later)



Basic Thread Scheduling States



Priority Adjustments

- Dynamic priority adjustments (boost and decay) are applied to threads in "dynamic" classes
  - Threads with base priorities 1-15 (technically, 1 through 14)
  - Disable if desired with SetThreadPriorityBoost or SetProcessPriorityBoost
- Five types:
  - I/O completion
  - Wait completion on events or semaphores
  - When threads in the foreground process complete a wait
  - When GUI threads wake up for windows input
  - For CPU starvation avoidance
- No automatic adjustments in "real-time" class (16 or above)
  - "Real time" here really means "system won't change the relative priorities of your real-time threads"
  - Hence, scheduling is predictable with respect to other "real-time" threads (but not for absolute latency)

Priority Boosting

To favor I/O intense threads:

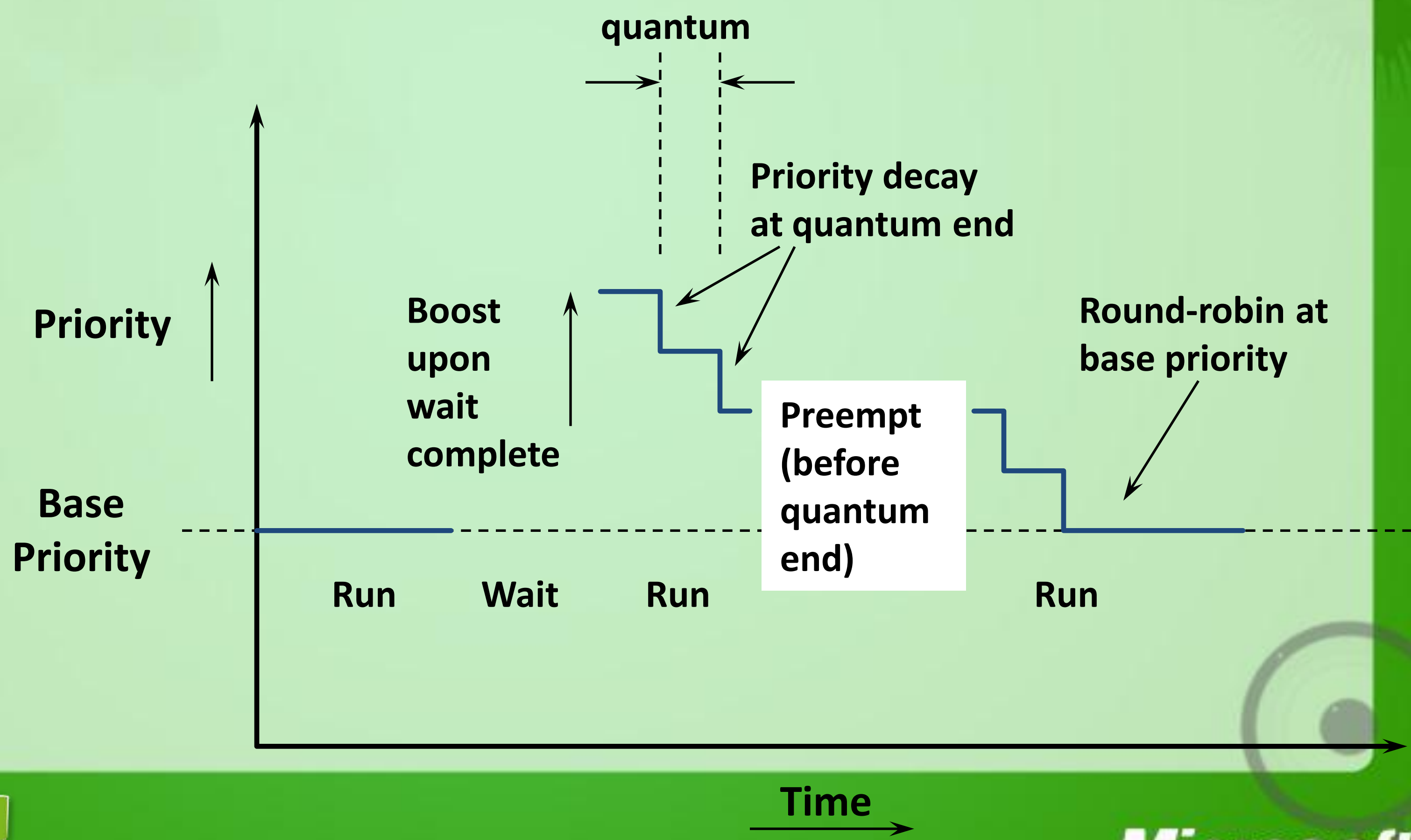
- After an I/O: specified by device driver
  - IoCompleteRequest(Irp, PriorityBoost)

Common boost values (see NTDDK.H)
1: disk, CD-ROM, parallel, Video
2: serial, network, named pipe, mailslot
6: keyboard or mouse
8: sound

Other cases:

- After a wait on executive event or semaphore
- After any wait on a dispatcher object by a thread in the foreground process
- GUI threads that wake up to process windowing input (e.g. windows messages) get a boost of 2

Thread Priority Boost and Decay



Five minutes break

Academic Tour 2011 imagine cup

Microsoft

Windows Memory Management Fundamentals

- Classical virtual memory management
  - Flat virtual address space per process
  - Private process address space
  - Global system address space
  - Per session address space
- Object based
  - Section object and object-based security (ACLs...)
- Demand paged virtual memory
  - Pages are read in on demand & written out when necessary (to make room for other memory needs)

Windows Memory Management Fundamentals

- **Lazy evaluation**
  - Sharing – usage of prototype PTEs (page table entries)
  - Extensive usage of copy_on_write
  - ...whenever possible
- **Shared memory with copy on write**
- **Mapped files** (fundamental primitive)
  - Provides basic support for file system cache manager

Memory Manager Components

- Six system threads
  - ***Working set manager* (priority 16)** – drives overall memory management policies, such as working set trimming, aging, and modified page writing
  - ***Process/stack swapper* (priority 23)** – performs both process and kernel thread stack inswapping and outswapping
  - ***Modified page writer* (priority 17)** – writes dirty pages on the modified list back to the appropriate paging files
  - ***Mapped page writer* (priority 17)** – writes dirty pages from mapped files to disk
  - ***Dereference segment thread* (priority 18) –** is responsible for cache and page file growth and shrinkage
  - ***Zero page thread* (priority 0)** – zeros out pages on the free list

MM: Working Sets

- Working Set:
  - The set of pages in memory at any time for a given process, or
  - All the pages the process can reference without incurring a page fault
  - Per process, private address space
  - WS limit: maximum amount of pages a process can own
  - Implemented as array of working set list entries (WSLE)
- Soft vs. Hard Page Faults:
  - Soft page faults resolved from memory (standby/modified page lists)
  - Hard page faults require disk access
- Working Set Dynamics:
  - Page replacement when WS limit is reached
  - NT 4.0: page replacement based on modified FIFO
  - From Windows 2000: Least Recently Used algorithm (uniproc.)

MM: Working Set Management

- Modified Page Writer thread
  - Created at system initialization
  - Writing modified pages to backing file
  - Optimization: min. I/Os, contigous pages on disk
  - Generally MPW is invoked before trimming
- Balance Set Manager thread
  - Created at system initialization
  - Wakes up every second
  - Executes MmWorkingSetManager
  - Trimming process WS when required: from current down to minimal WS for processes with lowest page fault rate
  - Aware of the system cache working set
  - Process can be out-swapped if all threads have pageable kernel stack

MM: I/O Support

- I/O Support operations:
  - Locking/Unlocking pages in memory
  - Mapping/Unmapping Locked Pages into current address space
  - Mapping/Unmapping I/O space
  - Get physical address of a locked page
  - Probe page for access
- Memory Descriptor List
  - Starting VAD
  - Size in Bytes
  - Array of elements to be filled with physical page numbers
- Physically contiguous vs. Virtually contiguous

Memory Manager: Services

- Caller can manipulate own/remote memory
  - Parent process can allocate/deallocate, read/write memory of child process
  - Subsystems manage memory of their client processes this way
- Most services are exposed through Windows API
- Services for device drivers/kernel code (Mm…)

Protecting Memory

Attribute	Description
PAGE_NOACCESS	Read/write/execute causes access violation
PAGE_READONLY	Write/execute causes access violation; read permitted
PAGE_READWRITE	Read/write accesses permitted
PAGE_EXECUTE	Any read/write causes access violation; execution of code is permitted (relies on special processor support)
PAGE_EXECUTE_ READ	Read/execute access permitted (relies on special processor support)
PAGE_EXECUTE_ READWRITE	All accesses permitted (relies on special processor support)
PAGE_WRITECOPY	Write access causes the system to give process a private copy of this page; attempts to execute code cause access violation
PAGE_EXECUTE_ WRITECOPY	Write access causes creation of private copy of pg.
PAGE_GUARD	Any read/write attempt raises EXCEPTION_GUARD_PAGE and turns off guard page status

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 (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

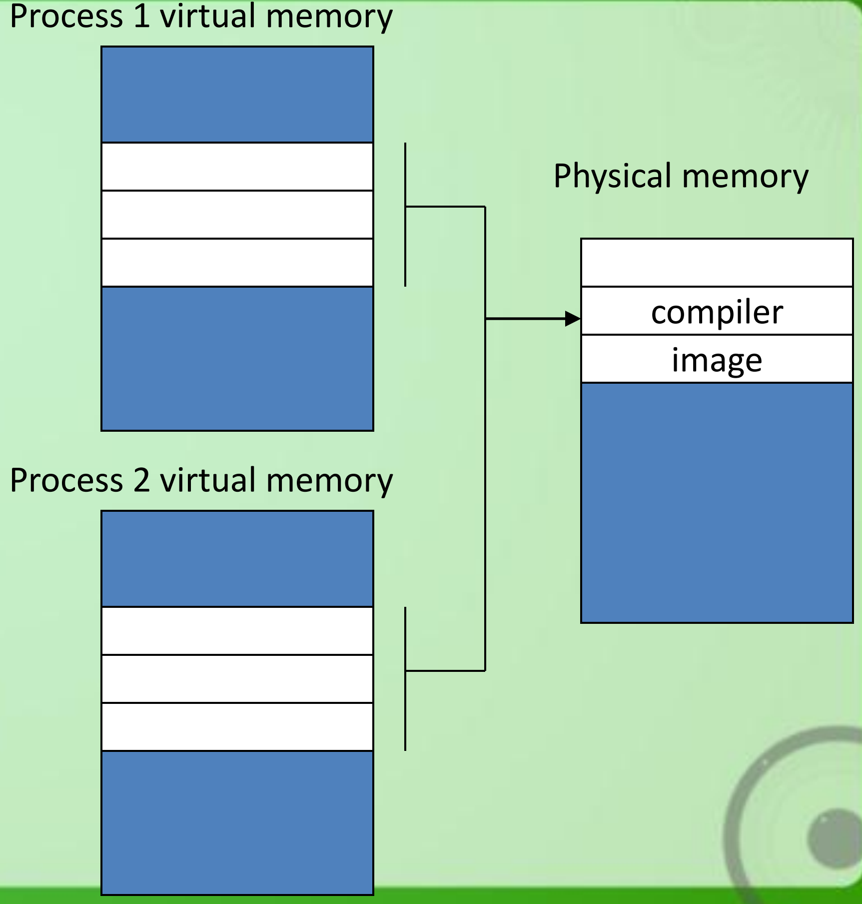
A thread‘s user-mode stack is constructed using this 2-phase approach: initial reserved size is 1MB, only 2 pages are committed: stack & guard page

Features new to Windows XP/2003 and newer OS in Memory Management

- 64-bit support
- Up to 1024 GB physical memory supported (2048 on 2008 R2)
- Support for Data Execution Prevention (DEP)
  - Memory manager supports HW no-execute protection
- Performance & Scalability enhancements

Shared Memory & Mapped Files

- Shared memory + copy-on-write per default
- Executables are mapped as read-only
- Memory manager uses section objects to implement shared memory
  (file mapping objects in Windows API)



Virtual Address Space Allocation

- Virtual address space is sparse
  - Address spaces contain reserved, committed, and unused regions
- Unit of protection and usage is one page
  - On x86, default page size is 4 KB (x86 supports 4KB or 4MB)
    - In PAE mode, large pages are 2 MB
  - On x64, default page size is 4 KB (large pages are 4 MB)
  - On Itanium, default page size is 8 KB
    (Itanium supports 4k, 8k, 16k, 64k, 256k, 1mb, 4mb, 16mb, 64mb, or 256mb) – large is 16MB

Large Pages

- Large pages allow a single page directory entry to map a larger region
  - x86, x64: 4 MB, IA64: 16 MB
  - Advantage: improves performance
    - Single TLB entry used to map larger area
- Disadvantage: disables kernel write protection
  - With small pages, OS/driver code pages are mapped as read only; with large pages, entire area must be mapped read/write
    - Drivers can then modify/corrupt system & driver code without immediately crashing system
  - Driver Verifier turns large pages off
  - Can also override by changing a registry key

Data Execution Prevention

- Windows XP SP2 and newer OS support Data Execution Prevention (DEP)
  - **Prevents code from executing in a memory page not specifically marked as executable**
  - **Stops exploits that rely on getting code executed in data areas**

- Relies on hardware ability to mark pages as non executable, AMD NX or Intel XD

- Processor support:
  - About all CPU from Intel, AMD and VIA shipped in last 4 years.

Data Execution Prevention

- Attempts to execute code in a page marked no execute result in:
  - User mode: access violation exception
  - Kernel mode: ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY bugcheck (blue screen)

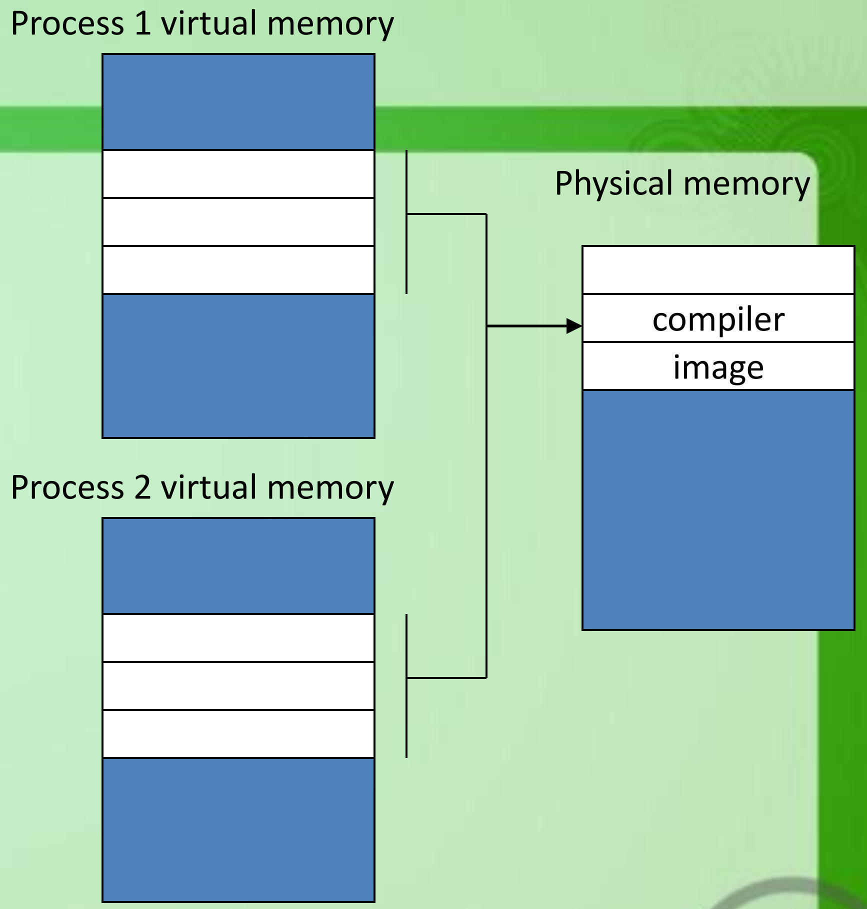
- Memory that needs to be executable must be marked as such using page protection bits on VirtualAlloc and VirtualProtect APIs:
  - PAGE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE, PAGE_EXECUTE_WRITECOPY

Mapped Files

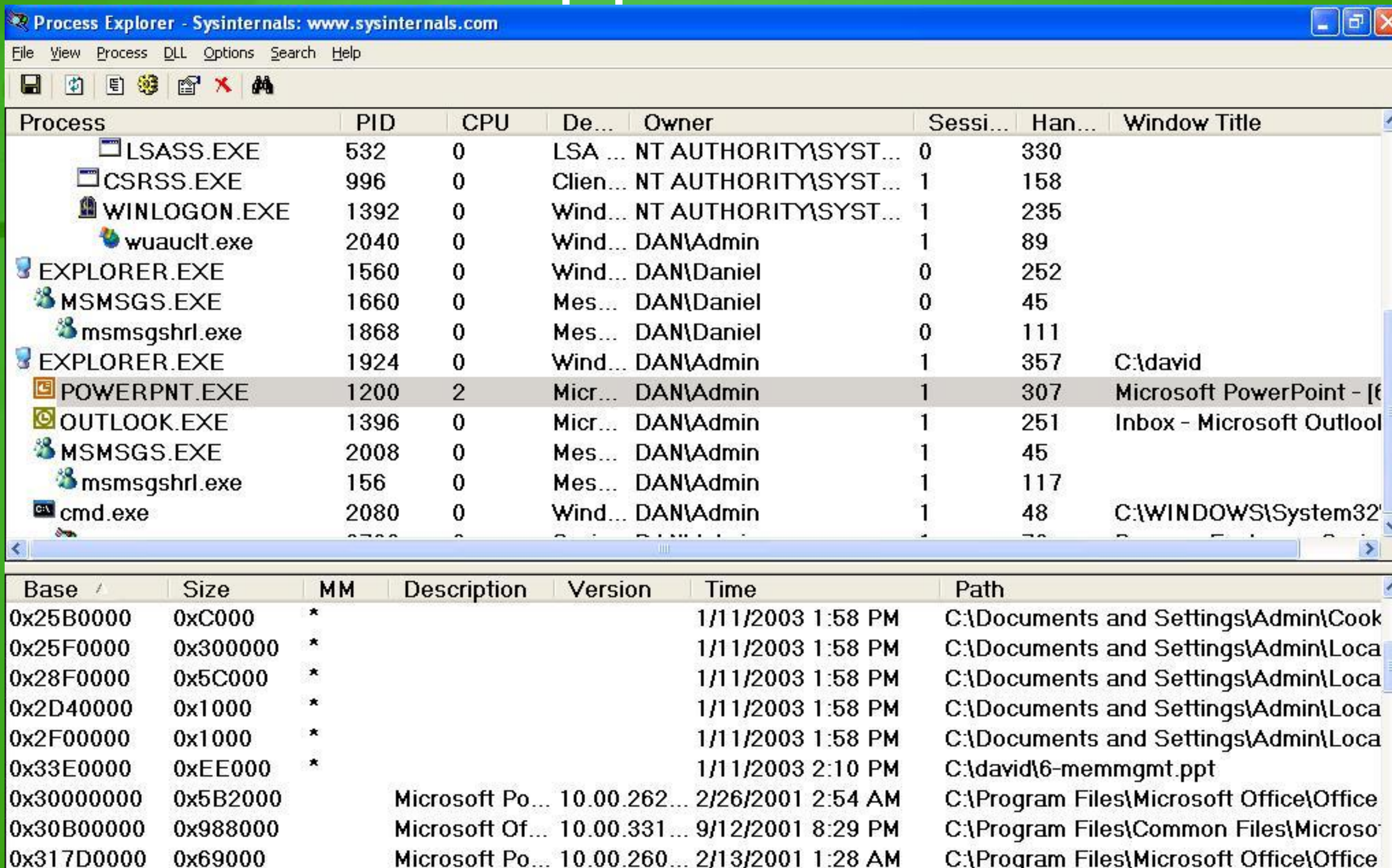
- A way to take part of a file and map it to a range of virtual addresses
  (address space is 2 GB, but files can be much larger)
- Called “file mapping objects” in Windows API
- Bytes in the file then correspond one-for-one with bytes in the region of virtual address space
  - Read from the “memory” fetches data from the file
  - Pages are kept in physical memory as needed
  - Changes to the memory are eventually written back to the file (can request explicit flush)
- Initial mapped files in a process include:
  - The executable image (EXE)
  - One or more Dynamically Linked Libraries (DLLs)

Shared Memory

- Like most modern OS's, Windows provides a way for processes to share memory
  - High speed IPC (used by LPC, which is used by RPC)
  - Threads share address space, but applications may be divided into multiple processes for stability reasons
- It does this automatically for shareable pages
  - E.g. code pages in an EXE or DLL
- Processes can also create shared memory sections
  - Called page file backed file mapping objects
  - Full Windows security



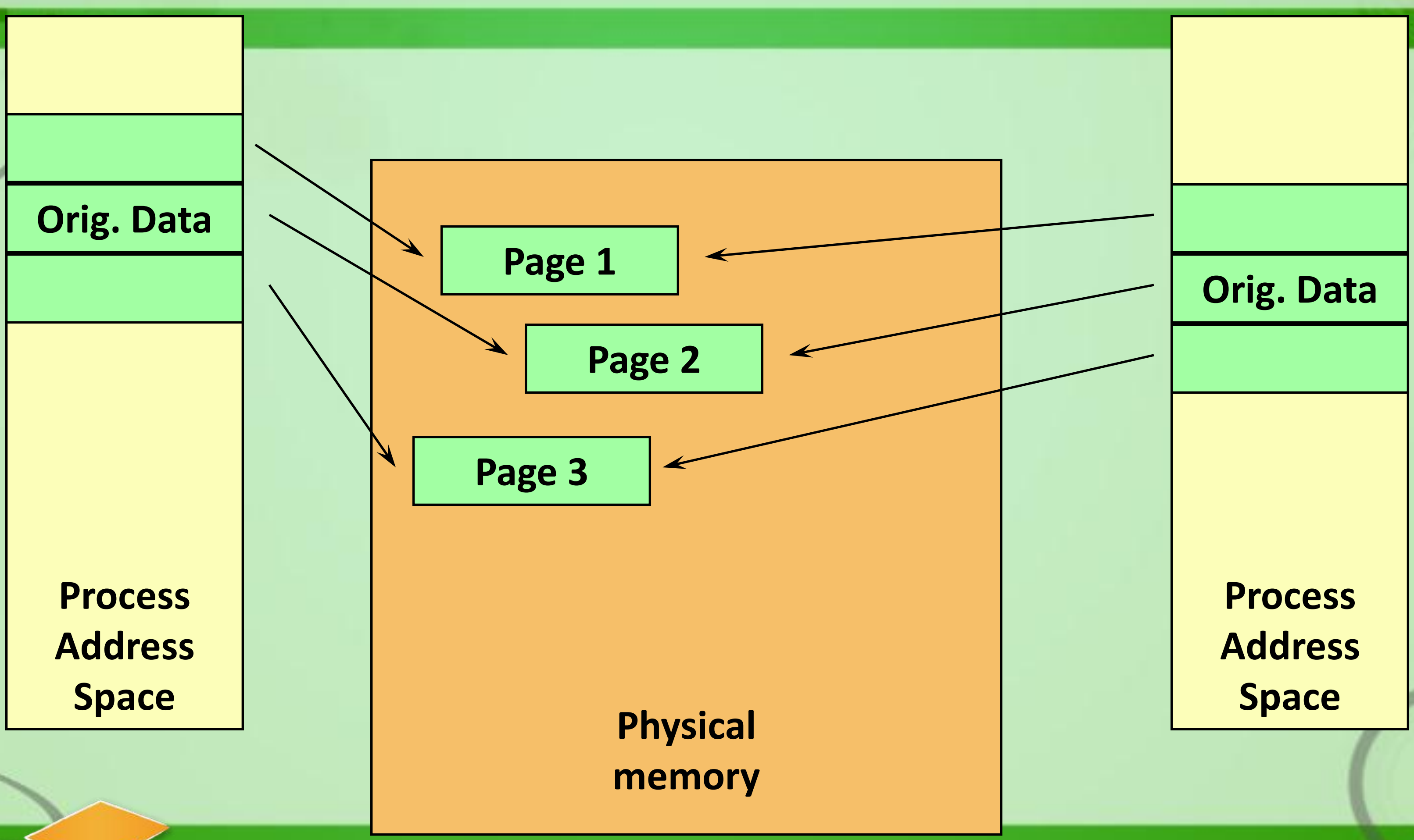
Viewing DLLs & Memory Mapped Files



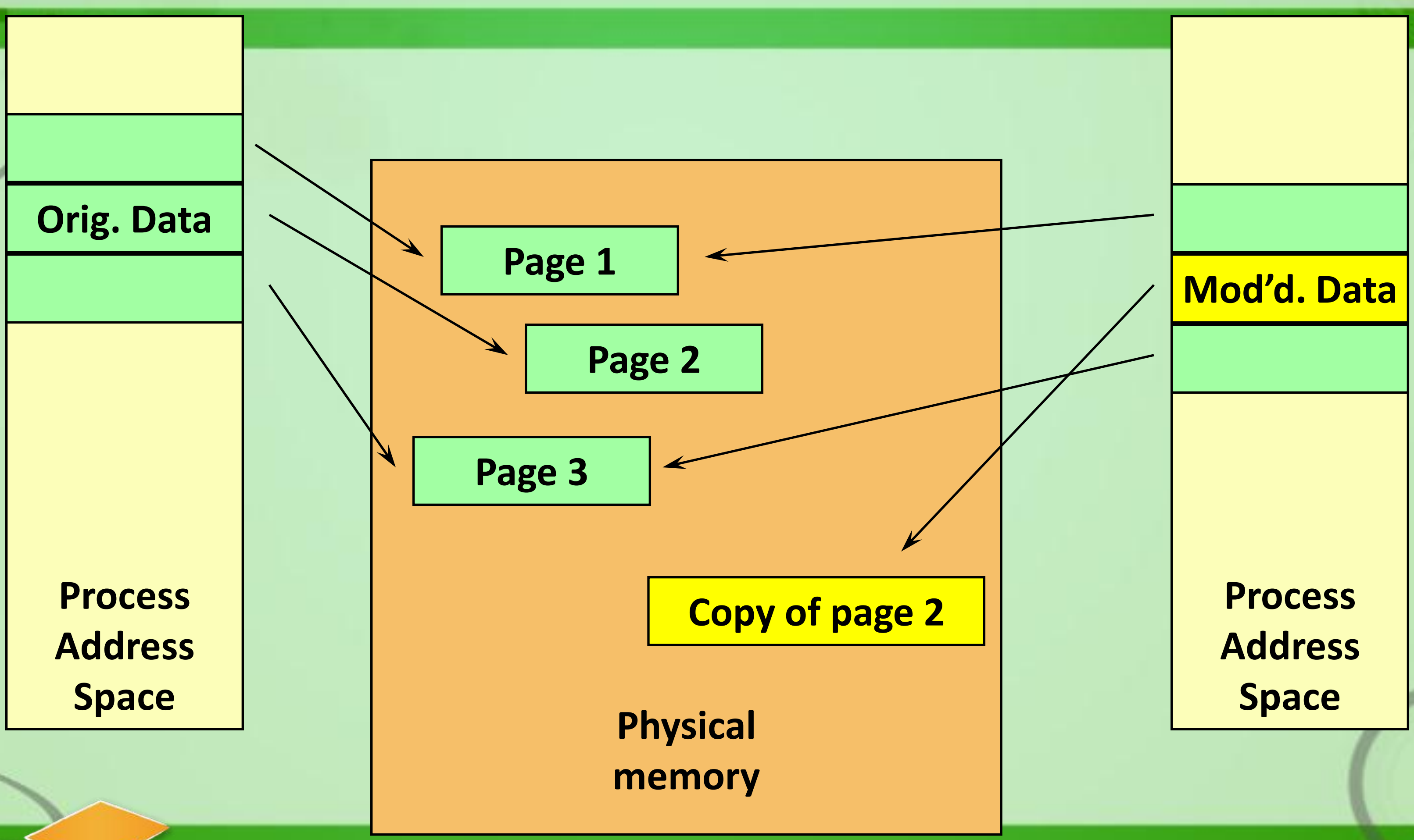
Copy-On-Write Pages

- Used for sharing between process address spaces
- Pages are originally set up as shared, read-only, faulted from the common file
  - Access violation on write attempt alerts pager
    - pager makes a copy of the page and allocates it privately to the process doing the write, backed to the paging file
  - So, only need unique copies for the pages in the shared region that are actually written (example of "lazy evaluation")
  - Original values of data are still shared
    - e.g. writeable data initialized with C initializers

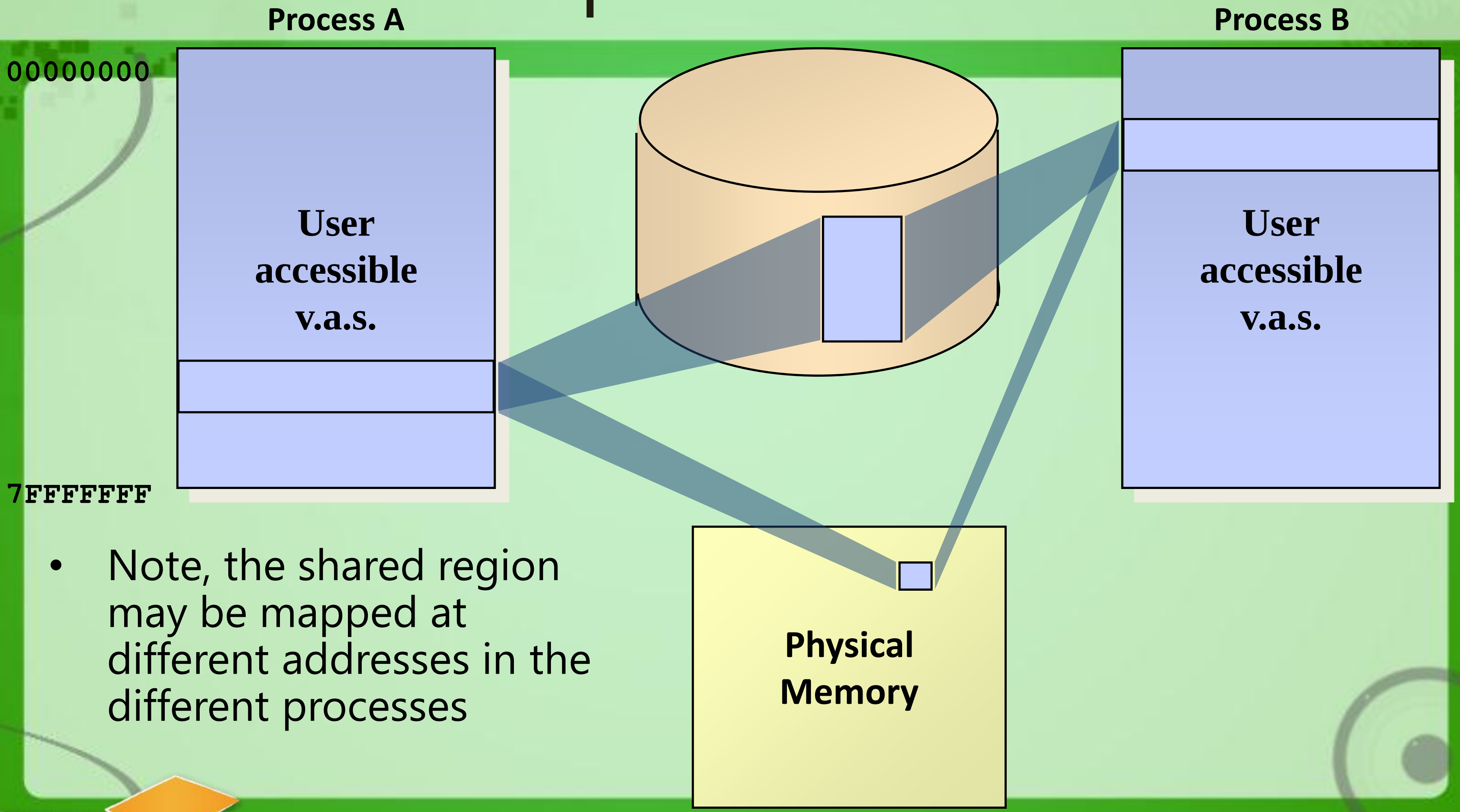
How Copy-On-Write Works
Before



How Copy-On-Write Works
After



Shared Memory = File Mapped by Multiple Processes

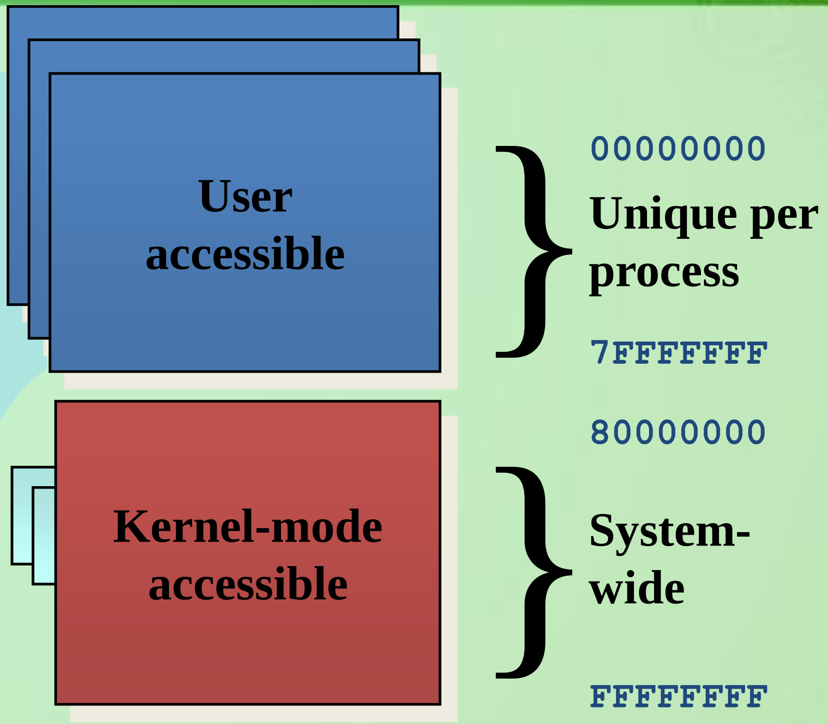


- Note, the shared region may be mapped at different addresses in the different processes

Virtual Address Space (V.A.S.)

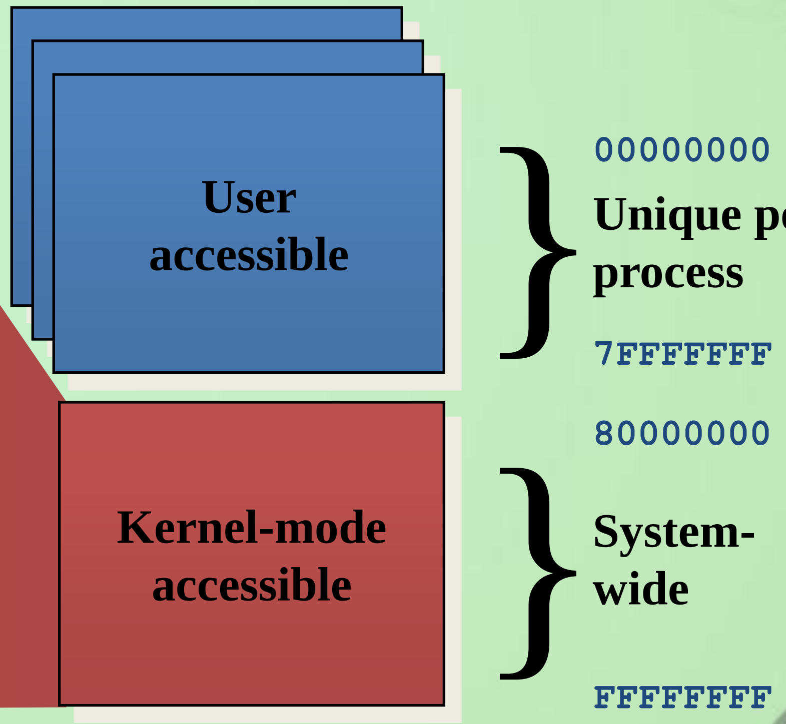
Process space contains:

- The application you're running (.EXE and .DLLs)
- A user-mode stack for each thread (automatic storage)
- All static storage defined by the application

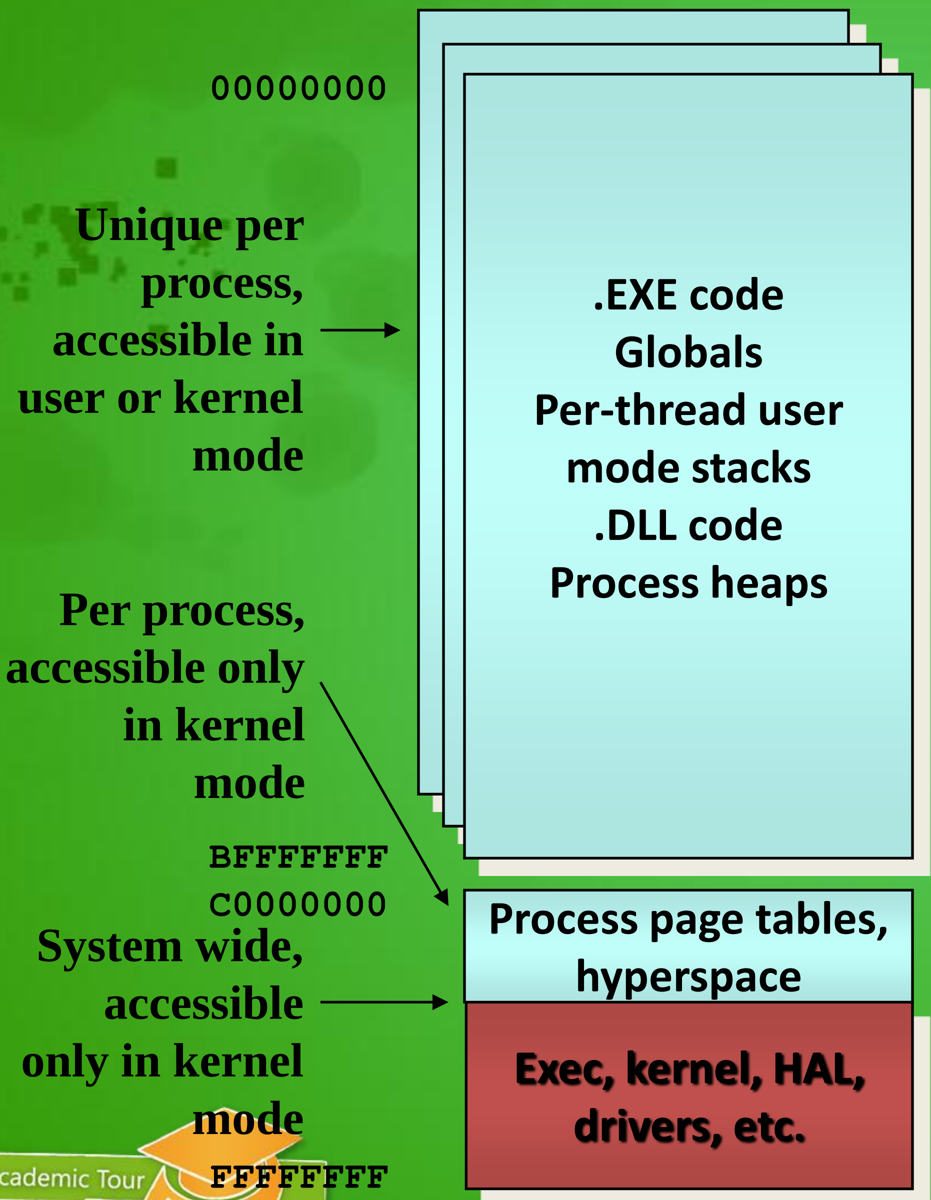


Virtual Address Space (V.A.S.)

- System space contains:
  - Executive, kernel, and HAL
  - Statically-allocated system-wide data cells
  - Page tables (remapped for each process)
  - Executive heaps (pools)
  - Kernel-mode device drivers (in nonpaged pool)
  - File system cache
  - A kernel-mode stack for every thread in every process



3GB Process Space Option



- Only available on operating system newer than Windows 2000 Server.
  - Can be activated from Boot.ini (Win 2k3, XP) or BCD (Vista, 7, 2008)
- Provides 3 GB per-process address space
  - Commonly used by database servers (for file mapping)
  - .EXE must have “large address space aware” flag in image header, or they’re limited to 2 GB (specify at link time or with imagecfg.exe from ResKit)
  - Chief “loser” in system space is file system cache
  - Better solution: address windowing extensions
  - **Even better: 64-bit Windows**

Physical Memory

- Maximum on Windows NT 4.0 was 4 GB for x86 (8 GB for Alpha AXP)
  - This is fixed by page table entry (PTE) format
- What about x86 systems with > 4 GB?
  - If CPU has PAE support can manage more than 64 GB (36 bits addressing)

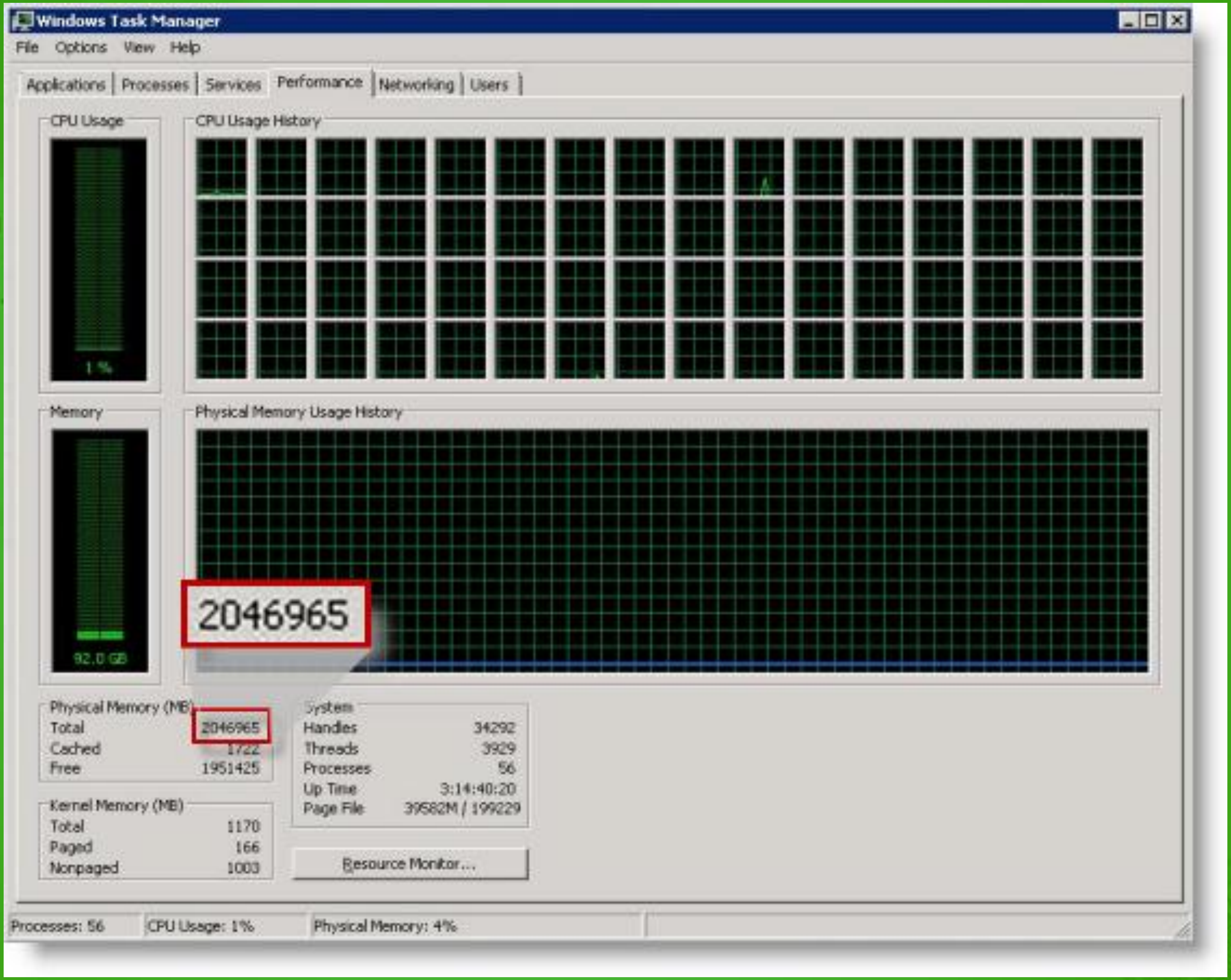
- Windows 2000 added proper support for PAE
  - Requires booting /PAE to select the PAE kernel

- Actual physical memory usable varies by Windows SKU.

Physical Memory Limits

	x86	x64 32-bit	x64 64-bit
XP Home	4	4	n/a
XP Professional	4	4	16 GB
Vista Home Premium	4	4	16 GB
Vista Bus / Ent / Ultimate	4	4	128 GB
Seven Home Premium	4	4	16 GB
Seven Pro / Ent / Ultimate	4	4	196 GB
2008 R2 Standard	n/a	n/a	32 GB
2008 R2 Ent / Datacenter	n/a	n/a	2 TB

http://msdn.microsoft.com/en-us/library/aa366778(VS.85).aspx



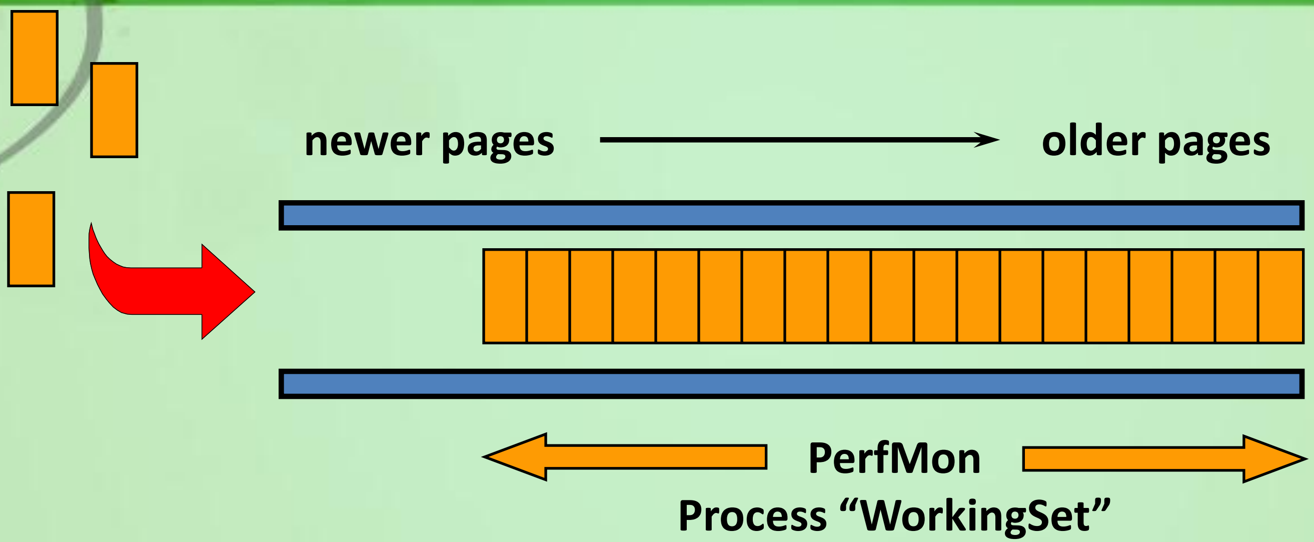
Academic Tour 2011 imagine cup

Microsoft

Working Set

- Working set: All the physical pages "owned" by a process
  - Essentially, all the pages the process can reference without incurring a page fault
- Working set limit: The maximum pages the process *can* own
  - When limit is reached, a page must be released for every page that's brought in ("working set replacement")
  - Default upper limit on size for each process
  - System-wide maximum calculated & stored in MmMaximumWorkingSetSize
    - approximately RAM minus 512 pages (2 MB on x86) minus min size of system working set (1.5 MB on x86)
    - Interesting to view (gives you an idea how much memory you've "lost" to the OS)
  - True upper limit: 2 GB minus 64 MB for 32-bit Windows

Working Set List



- A process always starts with an empty working set
  - It then incurs *page faults* when referencing a page that isn't in its working set
  - Many page faults may be resolved from memory (to be described later)

Birth of a Working Set

- Pages are brought into memory as a result of page faults
  - Prior to XP, no pre-fetching at image startup
  - But readahead is performed after a fault
    - See MmCodeClusterSize, MmDataClusterSize, MmReadClusterSize
- If the page is not in memory, the appropriate block in the associated file is read in
  - Physical page is allocated
  - Block is read into the physical page
  - Page table entry is filled in
  - Exception is dismissed
  - Processor re-executes the instruction that caused the page fault (and this time, it succeeds)
- The page has now been “faulted into” the process “working set”

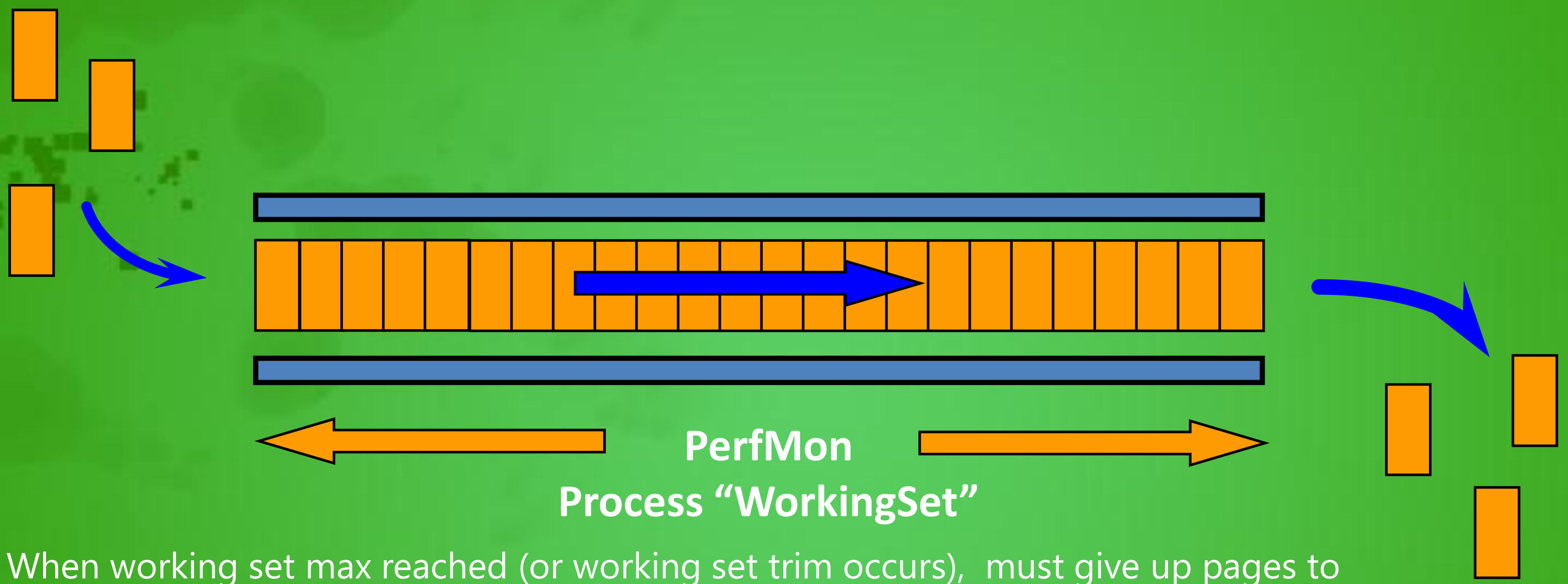
Prefetch Mechanism

- First 10 seconds of file activity is traced and used to prefetch data the next time
  - Also done at boot time (described in Startup/Shutdown section)
- Prefetch “trace file” stored in \Windows\Prefetch
  - Name of .EXE-<hash of full path>.pf

- When application run again, system automatically
  - Reads in directories referenced
  - Reads in code and file data
    - Reads are asynchronous, but waits for all prefetch to complete

- In addition, every 3 days, system automatically defrags files involved in each application startup

Working Set Replacement

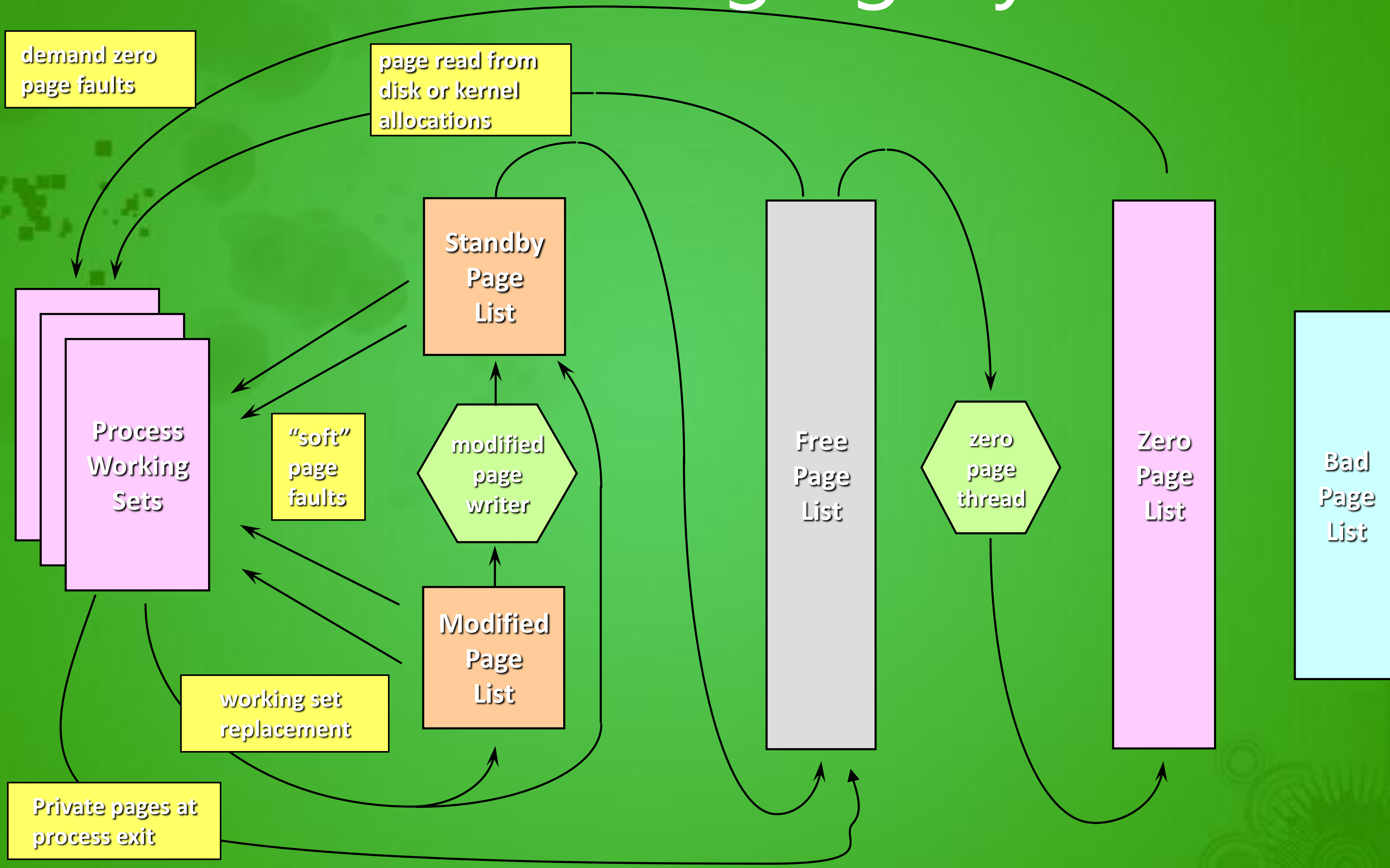


- When working set max reached (or working set trim occurs), must give up pages to make room for new pages
- Local page replacement policy (most Unix systems implement global replacement)
  - Means that a single process cannot take over all of physical memory unless other processes aren't using it
- Page replacement algorithm is least recently accessed (pages are aged)
  - On UP systems only in Windows 2000 – done on all systems in Windows XP/Server 2003
- New VirtualAlloc flag in XP/Server 2003: MEM_WRITE_WATCH

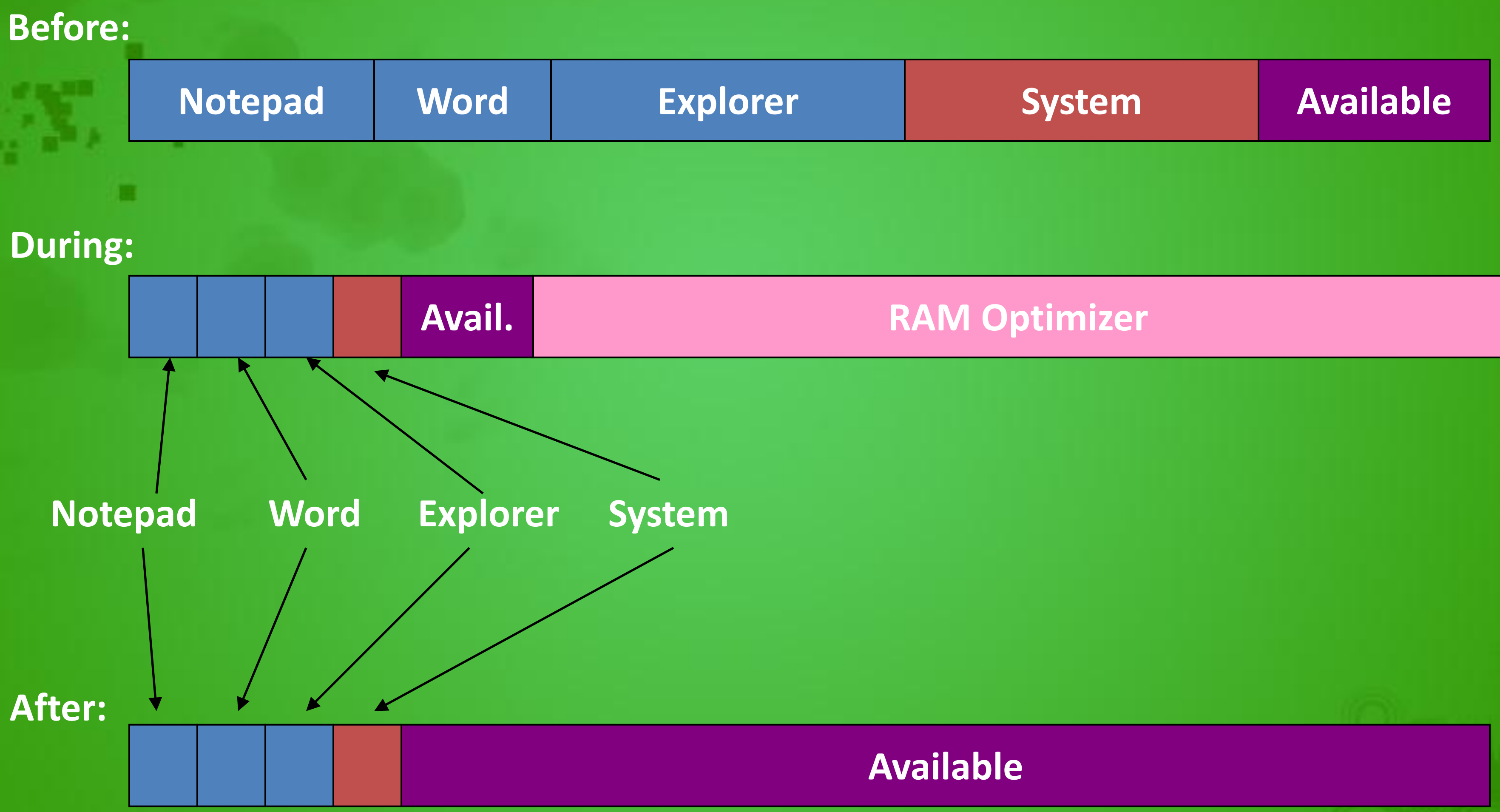
Free and Zero Page Lists

- Free Page List
  - Used for page reads
  - Private modified pages go here on process exit
  - Pages contain junk in them (e.g. not zeroed)
  - On most busy systems, this is empty
- Zero Page List
  - Used to satisfy demand zero page faults
    - References to private pages that have not been created yet
  - When free page list has 8 or more pages, a priority zero thread is awoken to zero them
  - On most busy systems, this is empty too

Paging Dynamics



Why “Memory Optimizers” are Fraudware



DOMANDE, RICHIESTE, SUGGERIMENTI?

GRAZIE A TUTTI PER L'ATTENZIONE!

Copyright Notice

Academic Tour 2011 imagine cup

Microsoft

Microsoft®

Your potential. Our passion.™

Academic Tour 2011 imagine cup

Microsoft