Unit 5: Memory Management

5.4. Physical Memory Management

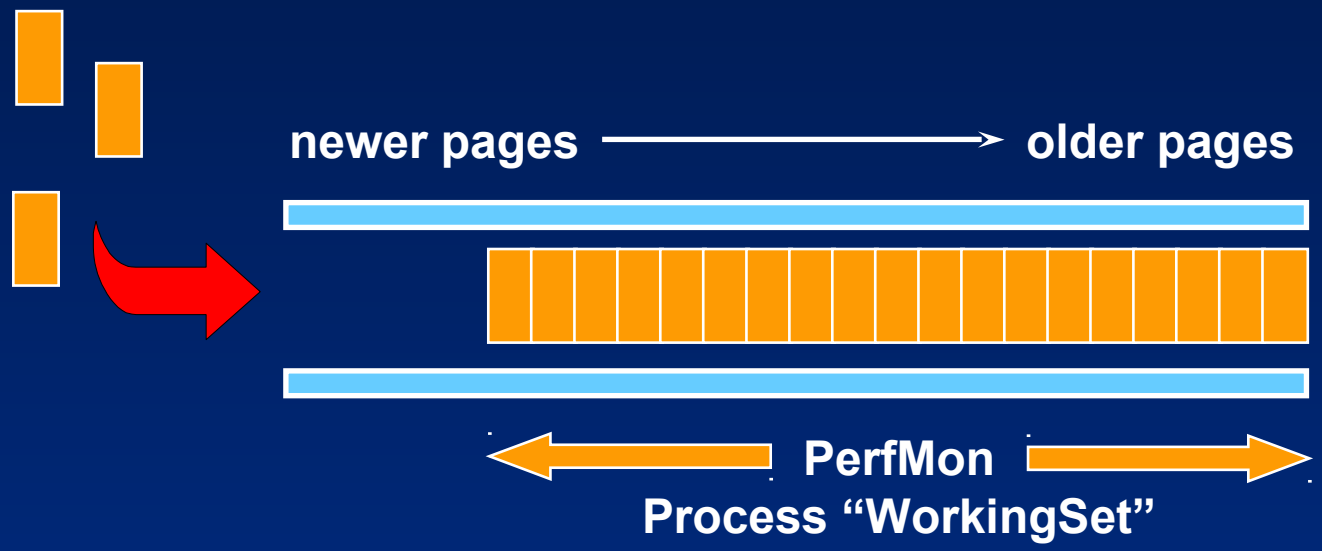
Roadmap for Section 5.4.

- From working sets to paging dynamics
- Birth of a process working set
- Working set trimming, heuristics
- Paging, paging dynamics
- Hard vs. soft page faults
- Page files

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”

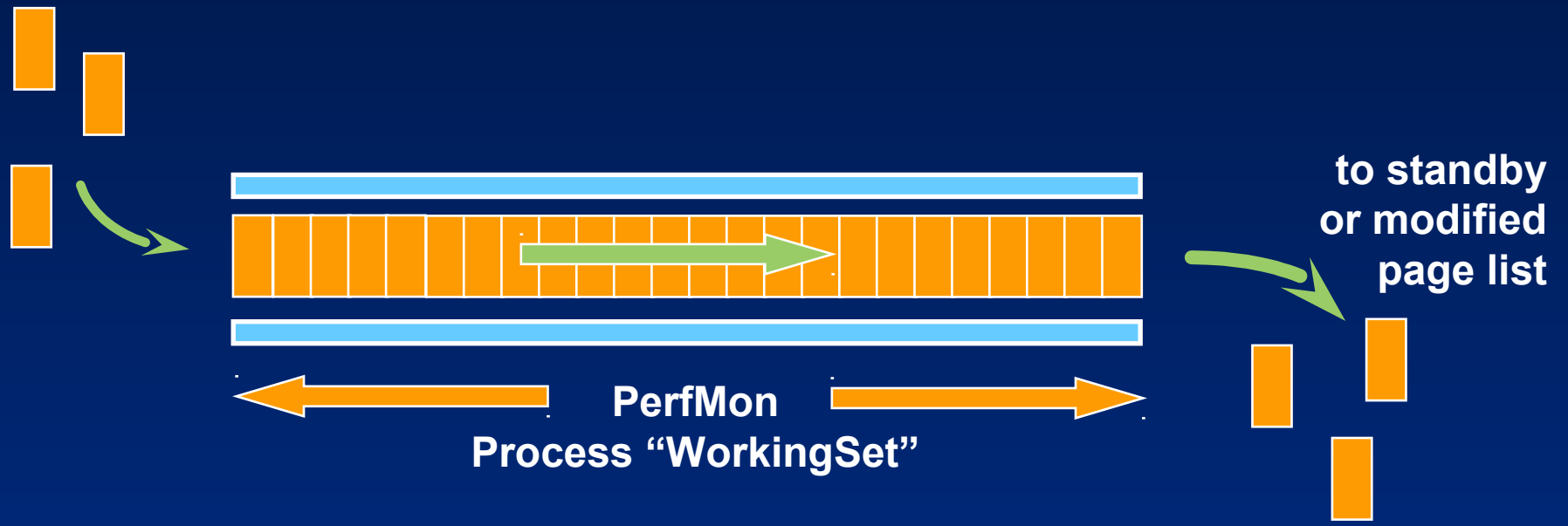
Prefetch Mechanism

- 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

Prefetch Mechanism

- In addition, every 3 days, system automatically defrags files involved in each application startup
- Bottom line: Reduces disk head seeks
  - This was seen to be the major factor in slow application/system 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

Soft vs. Hard Page Faults

- Types of “soft” page faults:
  - Pages can be faulted back into a process from the standby and modified page lists
  - A shared page that’s valid for one process can be faulted into other processes
- Some hard page faults unavoidable
  - Process startup (loading of EXE and DLLs)
  - Normal file I/O done via paging
    - Cached files are faulted into system working set
- To determine paging vs. normal file I/Os:
  - Monitor Memory->Page Reads/sec
    - Not Memory->Page Faults/sec, as that includes soft page faults
  - Subtract System->File Read Operations/sec from Page Reads/sec
  - Or, use Filemon to determine what file(s) are having paging I/O (asterisk next to I/O function)
  - Should not stay high for sustained period

Working Set System Services

- Min/Max set on a per-process basis
  - Can view with !process in Kernel Debugger
- System call below can adjust min/max, but has minimal effect prior to Server 2003
  - Limits are “soft” (many processes larger than max)
  - Memory Manager decides when to grow/shink working sets
- New system call in Server 2003 (SetProcessWorkingSetSizeEx) allows setting hard min/max
- Can also self-initiate working set trimming
  - Pass -1, -1 as min/max working set size (minimizing a window does this for you)

Windows API:

```
SetProcessWorkingSetSize( HANDLE hProcess,
                          DWORD dwMinimumWorkingSetSize,
                          DWORD dwMaximumWorkingSetSize )
```

Locking Pages

- Pages may be locked into the process working set
  - Pages are guaranteed in physical memory (“resident”) when any thread in process is executing

```
Windows API:
status = VirtualLock(baseAddress, size);
status = VirtualUnlock(baseAddress, size);
```

- Number of lockable pages is a fraction of the maximum working set size
  - Changed by SetProcessWorkingSetSize
- Pages can be locked into physical memory (by kernel mode code only)
  - Pages are then immune from “outswapping” as well as paging

```
MmProbeAndLockPages
```

Balance Set Manager

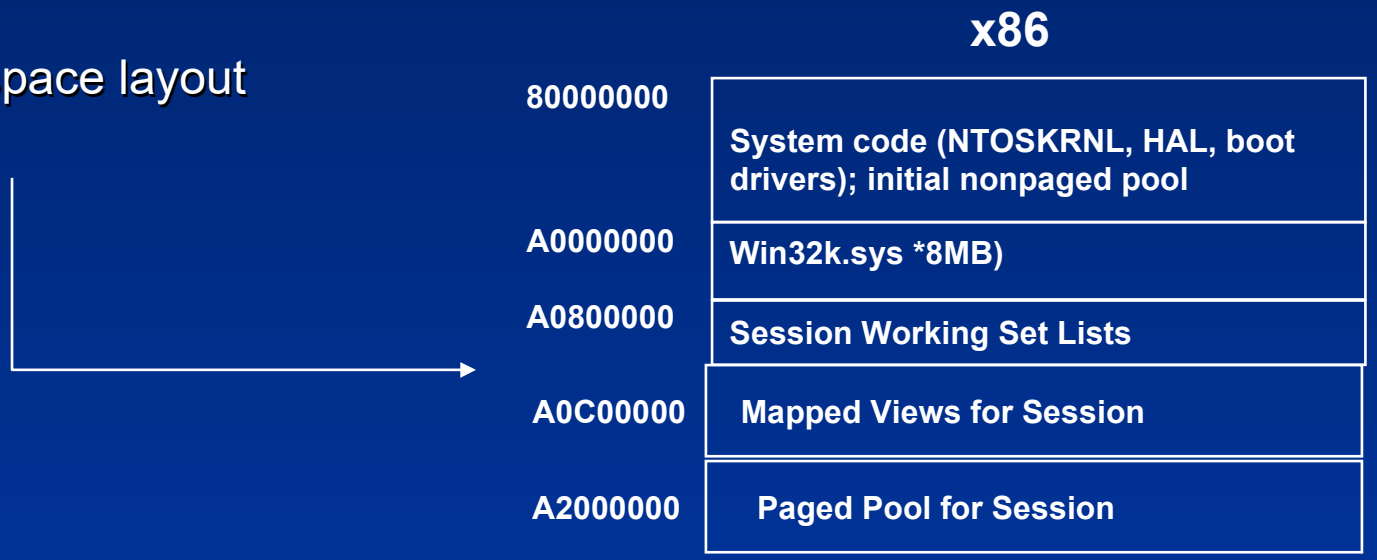
- Nearest thing Windows has to a “swapper”
  - Balance set = sum of all inswapped working sets
- Balance Set Manager is a system thread
  - Wakes up every second. If paging activity high or memory needed:
    - trims working sets of processes
    - if thread in a long user-mode wait, marks kernel stack pages as pageable
    - if process has no nonpageable kernel stacks, “outswaps” process
    - triggers a separate thread to do the “outswap” by gradually reducing target process’s working set limit to zero
- Evidence: Look for threads in “Transition” state in PerfMon
  - Means that kernel stack has been paged out, and thread is waiting for memory to be allocated so it can be paged back in
- This thread also performs a scheduling-related function
  - CPU starvation avoidance - already covered (*see Unit 4*)

System Working Set

- Just as processes have working sets, Windows' pageable system-space code and data lives in the "system working set"
- Made up of 4 components:
  - Paged pool
  - Pageable code and data in the executive
  - Pageable code and data in kernel-mode drivers, Win32K.Sys, graphics drivers, etc.
  - Global file system data cache
- To get physical (resident) size of these with PerfMon, look at:
  - Memory | Pool Paged Resident Bytes
  - Memory | System Code Resident Bytes
  - Memory | System Driver Resident Bytes
  - Memory | System Cache Resident Bytes
  - Memory | Cache bytes counter – is total of these four "resident" (physical) counters (not just the cache; in NT4, same as "File Cache" on Task Manager / Performance tab)

Session Working Set

- New memory management object to support Terminal Services in Windows 2000
- Session = an interactive user
- Session working set = the memory used by a session
  - Instance of WinLogon and Win32 subsystem process
  - WIN32K.SYS remapped for each unique session
    - Win32 subsystem objects
    - Win32 subsystem paged pool
  - Process working sets page within session working set
- Revised system space layout



Managing Physical Memory

- System keeps unassigned physical pages on one of several lists
  - Free page list
  - Modified page list
  - Standby page list
  - Zero page list
  - Bad page list - pages that failed memory test at system startup
- Lists are implemented by entries in the “PFN database”
  - Maintained as FIFO lists or queues

Paging Dynamics

- New pages are allocated to working sets from the top of the free or zero page list
- Pages released from the working set due to working set replacement go to the bottom of:
  - The modified page list (if they were modified while in the working set)
  - The standby page list (if not modified)
    - Decision made based on “D” (dirty = modified) bit in page table entry
  - Association between the process and the physical page is still maintained while the page is on either of these lists

Standby and Modified Page Lists

- Modified pages go to modified (dirty) list
  - Avoids writing pages back to disk too soon
- Unmodified pages go to standby (clean) list
- They form a system-wide cache of “pages likely to be needed again”
  - Pages can be faulted back into a process from the standby and modified page list
  - These are counted as page faults, but not page reads (from disk)

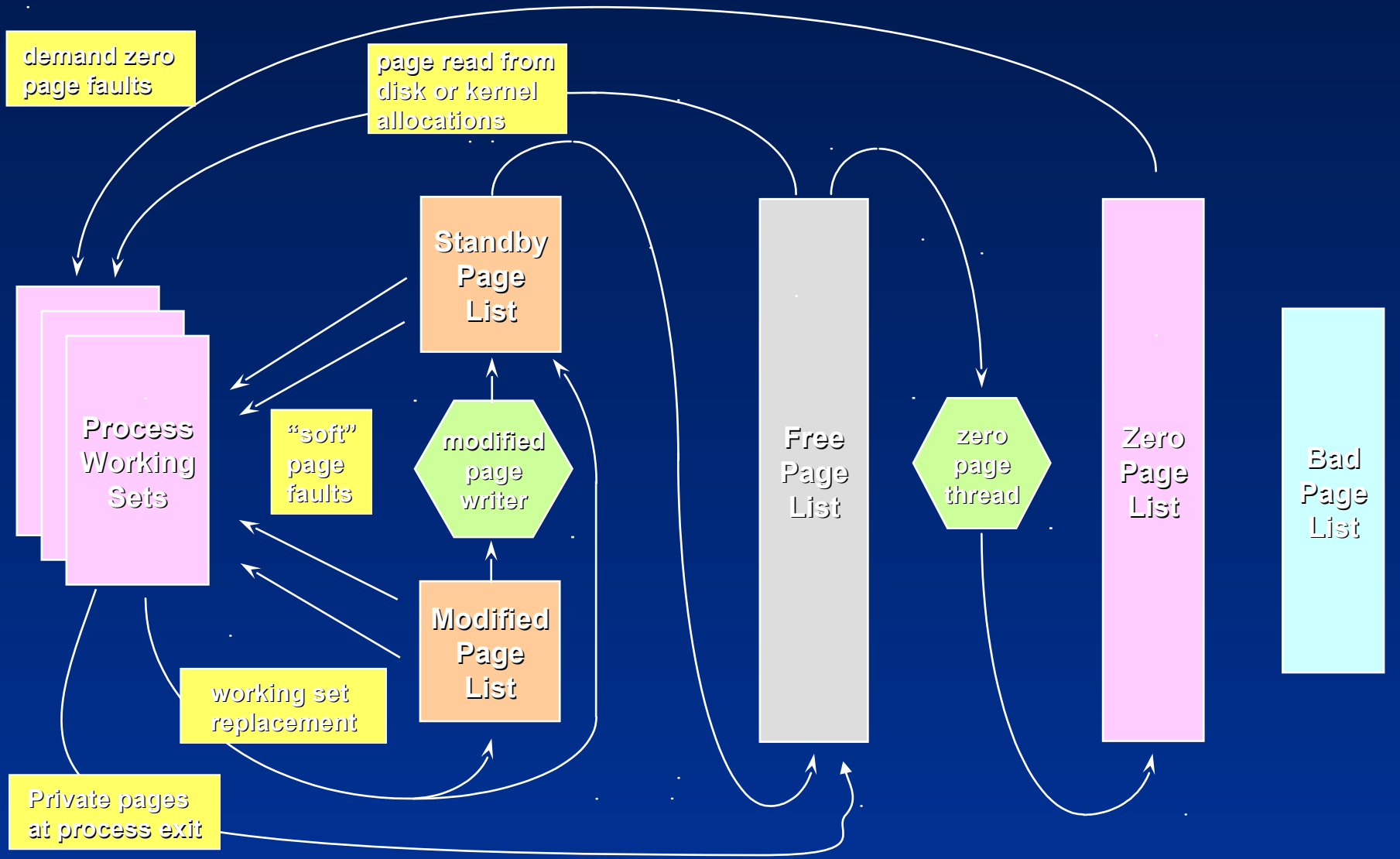
Modified Page Writer

- When modified list reaches certain size, modified page writer system thread is awoken to write pages out
  - See MmModifiedPageMaximum
  - Also triggered when memory is overcomitted (too few free pages)
  - Does not flush entire modified page list
- Two system threads
  - One for mapped files, one for the paging file
- After been written to disk, these pages move from the modified list to the standby list
  - E.g. they can still be soft faulted into a working set

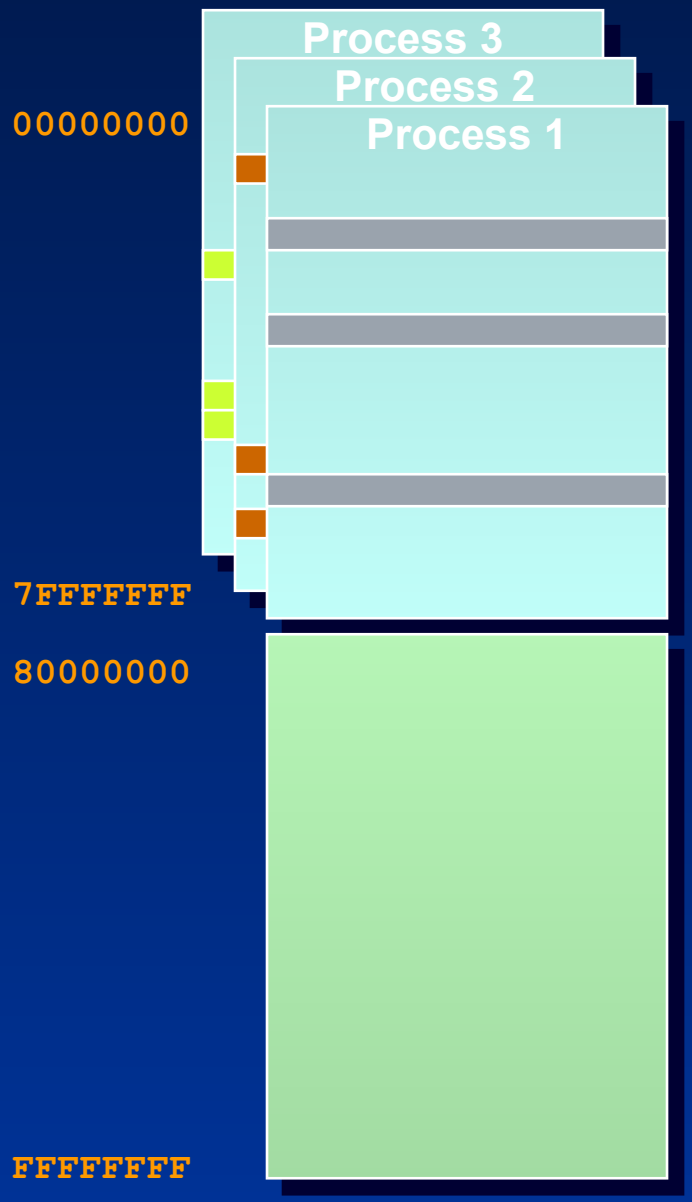
Free and Zero Page Lists

- Free Page List
  - Used for page reads (from disk)
  - 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



Working Sets in Memory



Pages in Physical Memory

- As processes incur page faults, pages are removed from the free, modified, or standby lists and made part of the process working set
- A shared page may be resident in several processes' working sets at one time (this case is not illustrated here)

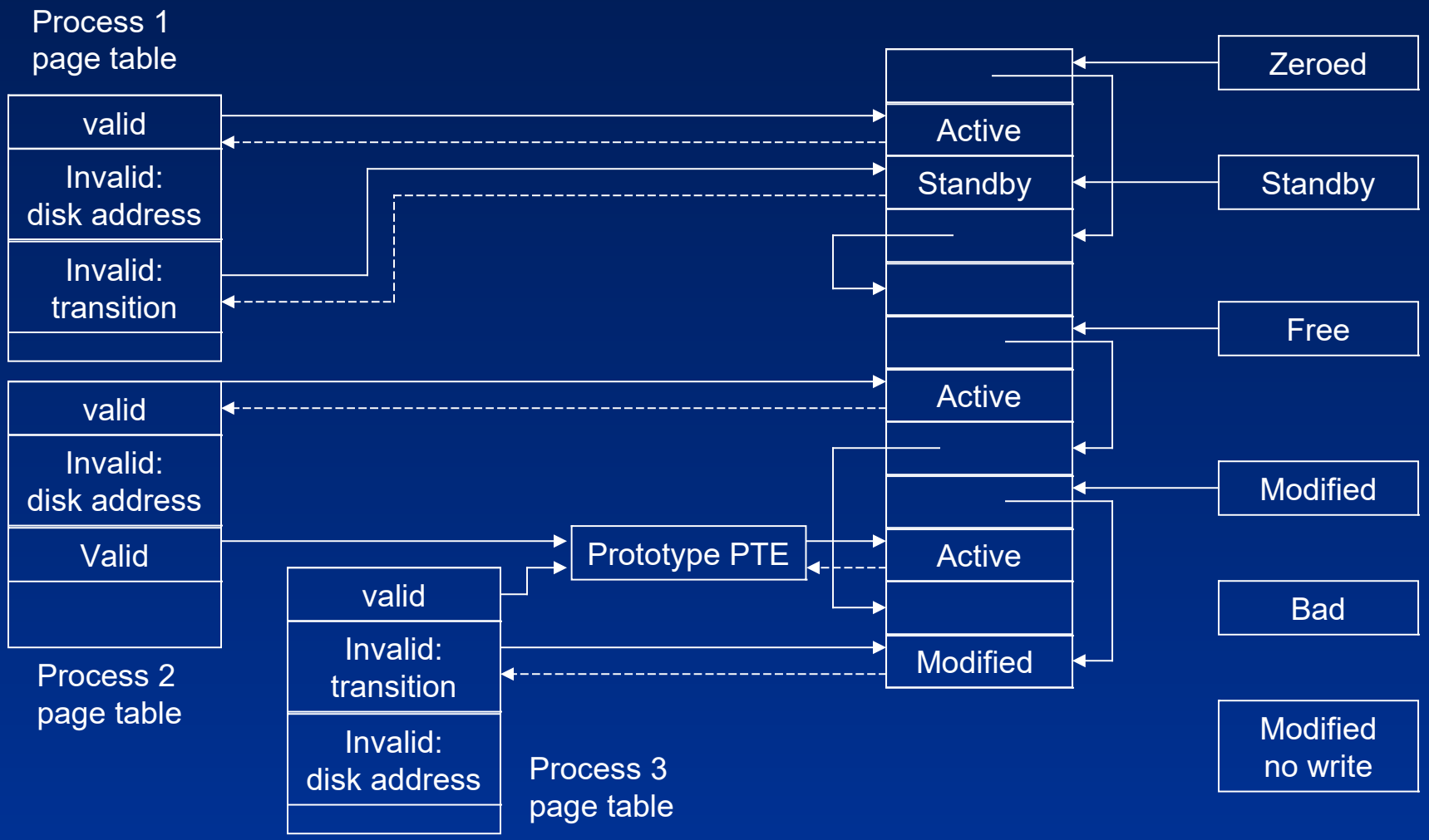
Page Frame Number Database

- PFN database contains one entry (24 bytes) for each physical page
  - Describes state of each page in physical memory
- Entries for active/valid and transition pages contain:
  - Original PTE value (to restore when paged out)
  - Original PTE virtual address and container PFN
  - Working set index hint (for the first process...)
- Entries for other pages are linked in:
  - Free, standby, modified, zeroed, bad lists (parity error will kill kernel)
- Share count (for active/valid pages):
  - Number of PTEs which refer to that page; 1->0: candidate for free list
- Reference count:
  - Locking for I/O: INC when share count 1->0; DEC when unlocked
  - Share count = 0 & reference count = 1 is possible
  - Reference count 1->0: page is inserted in free, standby or modified lists

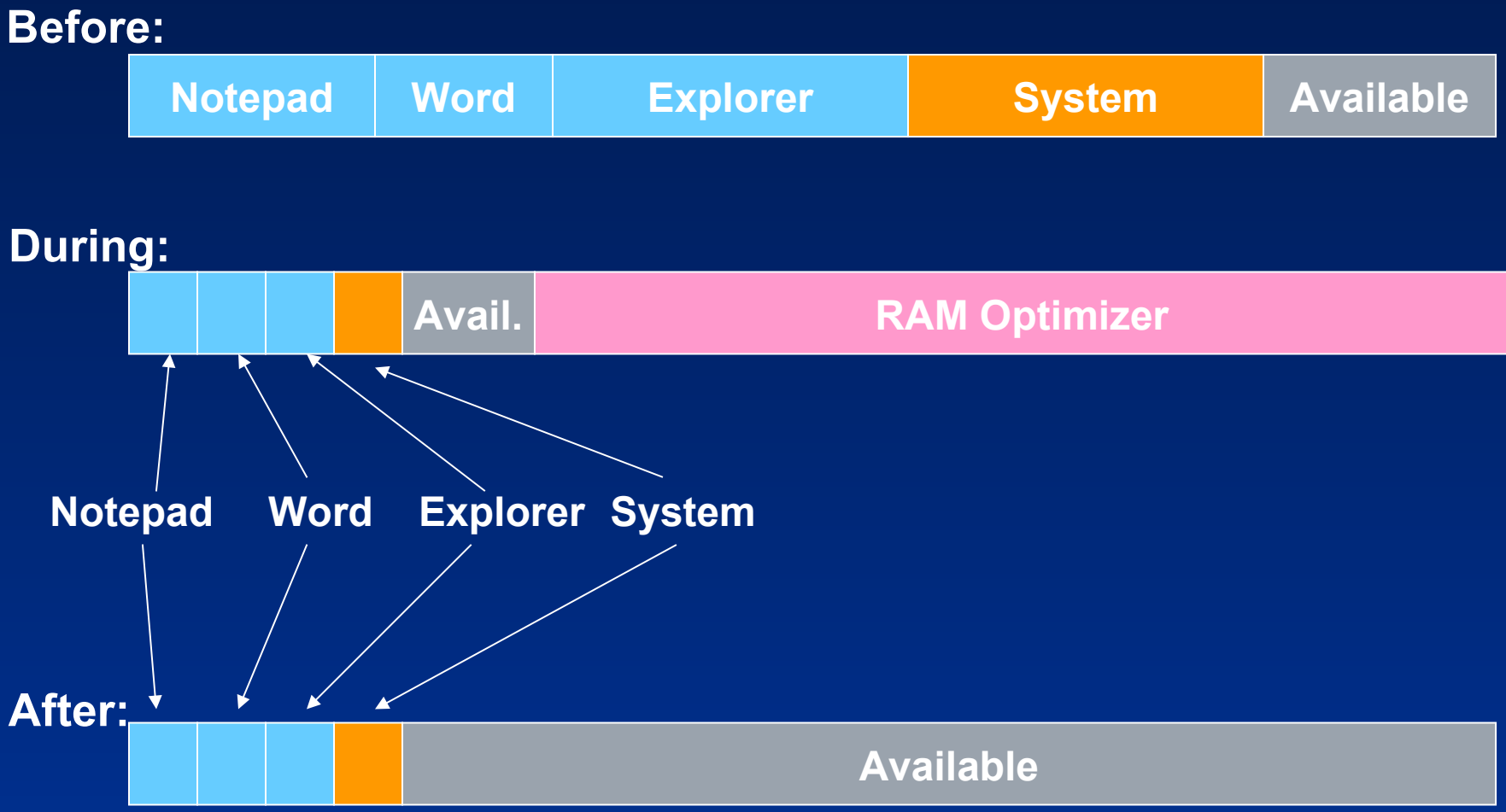
Page Frame Number Database – states of pages in physical memory

Status	Description
Active/valid	Page is part of a working set (system/process), valid PTE points to it
Transition	Page not owned by a working set, not on any paging list I/O is in progress on this page
Standby	Page belonged to a working set but was removed; not modified
Modified	Removed from working set, modified, not yet written to disk
Modified no write	Modified page, will not be touched by modified page writer, used by NTFS for pages containing log entries (explicit flushing)
Free	Page is free but has dirty data in it – cannot be given to user process – C2 security requirement
Zeroed	Page is free and has been initialized by zero page thread
Bad	Page has generated parity or other hardware errors

Page tables and PFN database



Why “Memory Optimizers” are Fraudware



See Mark’s article on this topic at
http://www.winnetmag.com/Windows/Article/ArticleID/41095/41095.html

Page Files

- What gets sent to the paging file?
  - Not code – only modified data (code can be re-read from image file anytime)
- When do pages get paged out?
  - Only when necessary
  - Page file space is only reserved at the time pages are written out
  - Once a page is written to the paging file, the space is occupied until the memory is deleted (e.g. at process exit), even if the page is read back from disk
- Windows XP (& Embedded NT4) can run with no paging file
  - NT4/Win2000: zero pagefile size actually creates a 20MB temporary page file (\temppf.sys)
  - WinPE never has a pagefile
- Page file maximum limits:
  - 16 page files per system
  - 32-bit x86: 4095MB
  - 32-bit PAE mode, 64-bit systems: 16 TB

Why Page File Usage on Systems with Ample Free Memory?

- Because memory manager doesn't let process working sets grow arbitrarily
  - Processes are not allowed to expand to fill available memory (previously described)
    - Bias is to keep free pages for new or expanding processes
  - This will cause page file usage early in the system life even with ample memory free
- We talked about the standby list, but there is another list of modified pages recently removed from working sets
  - Modified private pages are held in memory in case the process asks for it back
  - When the list of modified pages reaches a certain threshold, the memory manager writes them to the paging file (or mapped file)
  - Pages are moved to the standby list, since they are still "valid" and could be requested again

Sizing the Page Files

- Given understanding of page file usage, how big should the total paging file space be?
  - (Windows supports multiple paging files)
- Size should depend on total private virtual memory used by applications and drivers
  - Therefore, not related to RAM size (except for taking a full memory dump)
- Worst case: system has to page out all private data to make room for code pages
  - To handle this worst case, minimum size of total paging file space should be the maximum of VM usage (see the performance counter “Commit Charge Peak”)
    - Hard disk space is cheap, so why not double this
- Normally, make maximum size same as minimum
  - But, max size could be much larger if there will be infrequent demands for large amounts of page file space
    - Performance problem: page file extension will likely be very fragmented
    - Extension is deleted on reboot, thus returning to a contiguous page file

Contiguous Page Files

- A few fragments won't hurt, but hundreds of fragments will
- Will be contiguous when created if contiguous space available at that time
- Can defrag with Pagedefrag tool (freeware - www.sysinternals.com)
  - Or buy a paid defrag product…

When Page Files are Full

- When page file space runs low
  - 1. “System running low on virtual memory”
    - First time: Before pagefile expansion
    - Second time: When committed bytes reaching commit limit
  - 2. “System out of virtual memory”
    - Page files are full
- Look for who is consuming pagefile space:
  - Process memory leak: Check Task Manager, Processes tab, VM Size column
    - or Perfmon “private bytes”, same counter
  - Paged pool leak: Check paged pool size
    - Run poolmon to see what object(s) are filling pool
    - Could be a result of processes not closing handles - check process “handle count” in Task Manager

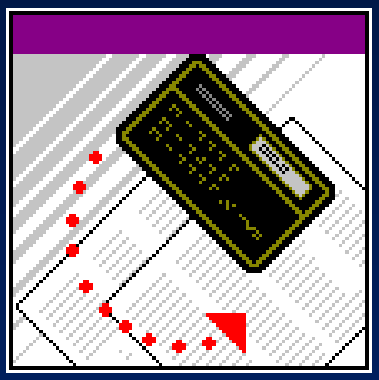
Optimizing Applications
Minimizing Page Faults

- **Some page faults are unavoidable**
  - code is brought into physical memory (from .EXEs and .DLLs) via page faults
  - the file system cache reads data from cached files in response to page faults
- **First concern is to minimize number of "hard" page faults**
  - i.e. page faults to disk
  - see Performance Monitor, "Memory" object, "page faults" vs. "page reads" (this is system-wide, not per process)
  - for an individual app, see Page Fault Monitor:

```
c:\> pfmon /?
c:\> pfmon program-to-be-monitored
```

  - note that these results are highly dependent on system load (physical memory usage by other apps)

Accounting for Physical Memory Usage



- Process working sets
  - Perfmon: Process / Working set
  - Note, shared resident pages are counted in the process working set of every process that's faulted them in
  - Hence, the total of all of these may be greater than physical memory
- Resident system code (NTOSKRNL + drivers, including win32k.sys & graphics drivers)
  - see total displayed by !drivers 1 command in kernel debugger
- Nonpageable pool
  - Perfmon: Memory / Pool nonpaged bytes
- Free, zero, and standby page lists
  - Perfmon: Memory / Available bytes
- Pageable, but currently-resident, system-space memory
  - Perfmon: Memory / Pool paged resident bytes
  - Perfmon: Memory / System cache resident bytes

  Memory | Cache bytes counter is really total of these four "resident" (physical) counters
- Modified, Bad page lists
  - can only see the size of these with !memusage command in Kernel Debugger

Further Reading

- Pavel Yosifovich, Alex Ionescu, et al., “Windows Internals”, 7th Edition, Microsoft Press, 2017.
  - Chapter 5 – Memory management (from pp. 421)
    - System memory pools (from pp. 454)
    - Page fault handling (from pp. 531)
    - Page frame number database (from pp. 589)
    - Working sets (from pp. 572)