Windows Internals II

Object Management

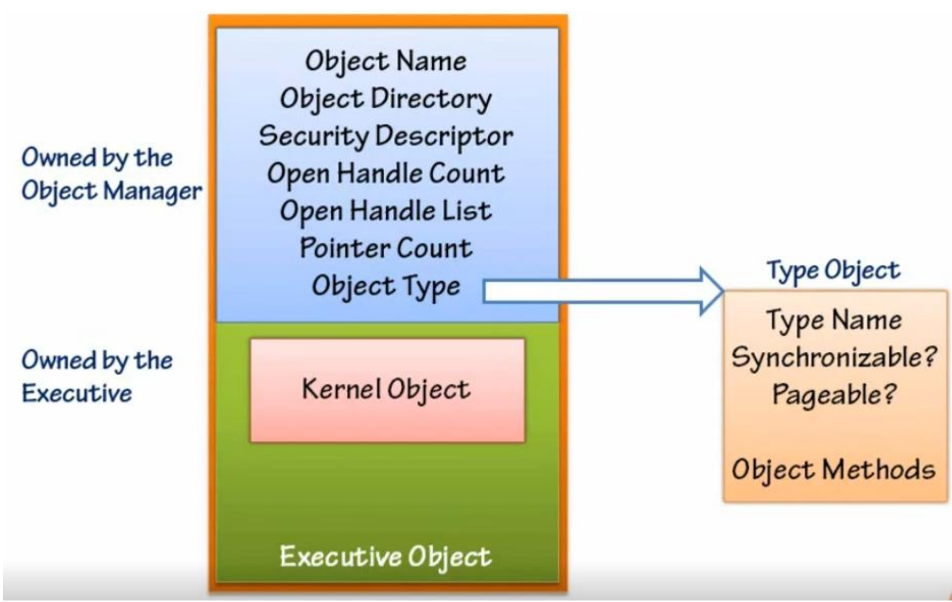
Object Manager

- It's a part of the executive.
- It manages creating, deleting and tracking objects.
- It maintains objects in a tree-like structure (can be partially viewed with WinObj tool).
- User mode clients can obtain handles to objects (but can't touch actual memory structure).
- Kernel mode clients can use both handles and objects themselves.

Object Types Exposed by the Windows API

- Process (CreateProcess, OpenProcess).
- Thread (CreateThread, OpenThread).
- Job (CreateJobObject, OpenJobObject).
- File Mapping (CreateFileMapping, OpenFileMapping).
- File (Create File).
- Token (LogonUser, GetProcessToken).
- Mutex or Mutant (CreateMutex, OpenMutex).
- Event (CreateEvent, OpenEvent).
- Semaphore (CreateSemaphore, OpenSemaphore).
- Timer (CreateWaitableTimer, OpenWaitableTimer).
- I/O Completion Port (CreateIoCompletionPort).
- Window Station (CreateWindowStation, OpenWindowStation).
- Desktop (CreateDesktop, OpenDesktop).

Object Structure



Object and Handles

- When a process creates or opens an object, it receives a handle to that object.
  - A handle is just a number and it serves as an index to a table maintained by EPROCESS.
  - Used as an opaque, indirect pointer to the underlying object.
  - It allows sharing objects across processes.
- In .NET, handles are used internally by types such as FileStream, Mutex, Semaphore, etc.
- Each Process has a private handle table and it can't be shared with other processes.
- Viewing process handles:
  - Process Explorer (GUI) or handle.exe (Console).
  - Resource Monitor.
  - !handle debugger command.

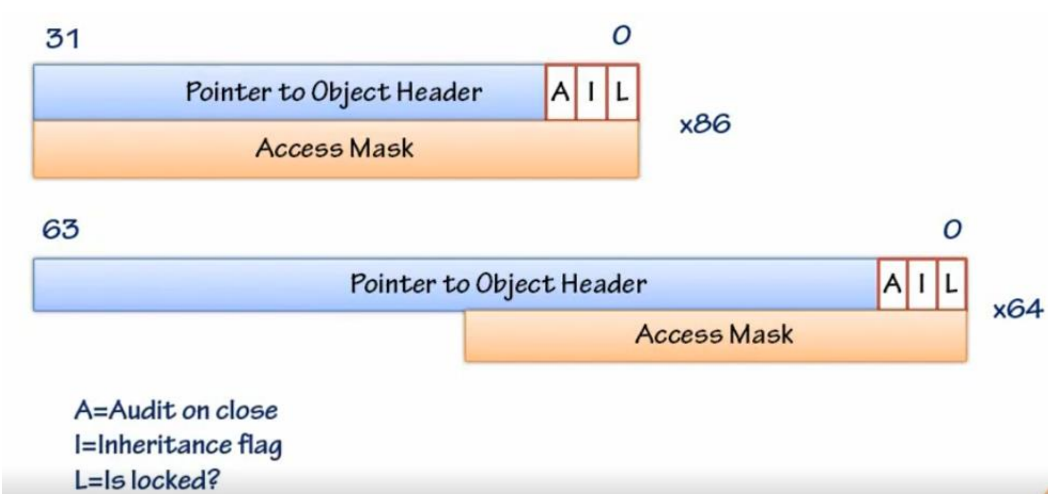
Handle Usage

- User mode processes retrieve a handle by calling CreateFunction or OpenFunction.
- Each object must have a unique name or GetLastError() returns ERROR_ALREADY_EXISTS.
- Kernel code can obtain handles that reside in system space. It can also obtain a direct pointer to underlying object given a handle by calling ObReferenceObjectbyHandle.
- When a process terminates for whatever reason all handles will be closed by the terminal.

Sharing Objects

- A handle is private to its containing process.
- Sharing is possible through:
  - Process handle inheritance (some handles are copied into the newly created process).
  - Opening an object by name (dangerous because names are global and can be changed).
  - Duplicating a handle.

Handle Entry Layout



Object Names and Sessions

- Each session should have its own objects.
- The object manager creates a Sessions directory with a session ID subdirectory.
- Processes can access the global session objects by prefixing object names with "Global\".
- CreatePrivateNamespace function is used for tightened security.

User and GDI Objects

- The Object Manager is responsible for kernel objects only.
- User and GDI objects are managed by Win32k.sys.
- The API functions in user32.dll and gdi32.dll don't go through NtDll.dll (because Ntdll.dll is related directly to the kernel such as handling memory, processes, synchronization, threads, security, etc).
- user32.dll and gdi32.dll invoke the sysenter/syscall instructions directly.
- User objects: Windows (HWND), menus (HMENU) and hooks (HHOOK).
- User objects handles: No reference/handle counting and private to a Window Station.
- GDI objects: Device context (HDC), pen (HPEN), brush (HBRUSH), bitmap (HBITMAP), etc.
- GDI object handles: No reference/handle counting and private to a process.

Object Management Summary

- Objects are structured entities managed by the Object Manager.
- Objects are reference counted.
- Objects can be shared between processes.
- User mode clients work with handles.

Memory Management

Memory Manager Fundamentals

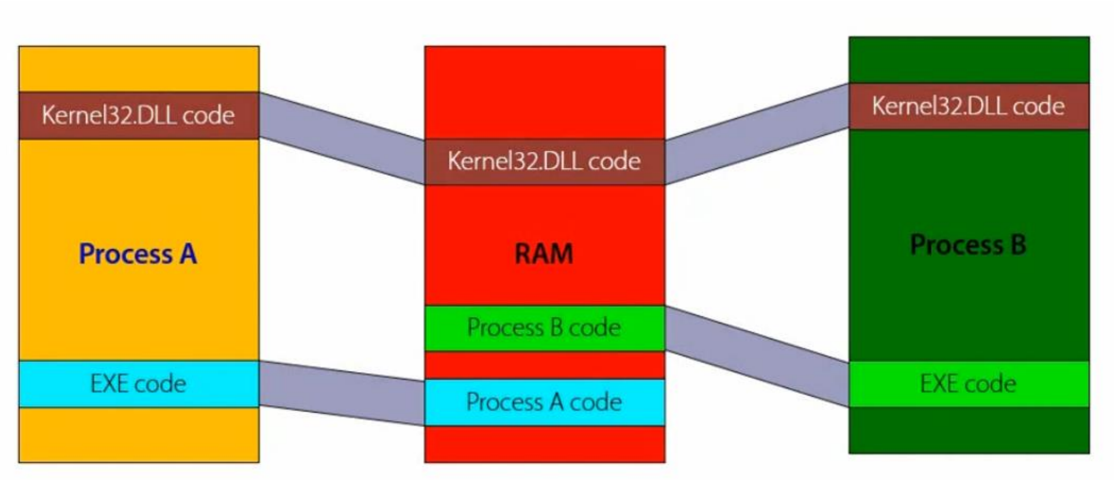
- Each process sees a virtual address space (2GB for 32-bit and 8TB for 64-bit).
- Memory Manager tasks:
  - Mapping virtual addresses to physical addresses.
  - Using page files to back up pages that cannot fit in the physical memory.
  - Provide memory management services to other system components.
- Memory is managed by Pages.
- Page size is determined by CPU type.
- Allocations, de-allocations and other memory block attributes are always per page.

Architecture	Small (normal) page size	Large page size
x86	4 KB	2 MB (PAE), 4MB (Non PAE)
x64	4 KB	2 MB
IA-64	8 KB	16 MB

Virtual Page States

- Free: Unallocated page. Any access causes violation exception.
- Committed: Allocated page that can be accessed. It may have a backup on disk.
- Reserved: Unallocated pages causing violation on access. Address range won't be for future allocations unless specifically requested.
- They can be viewed using VMMap tool from Sysinternals.

Sharing Pages



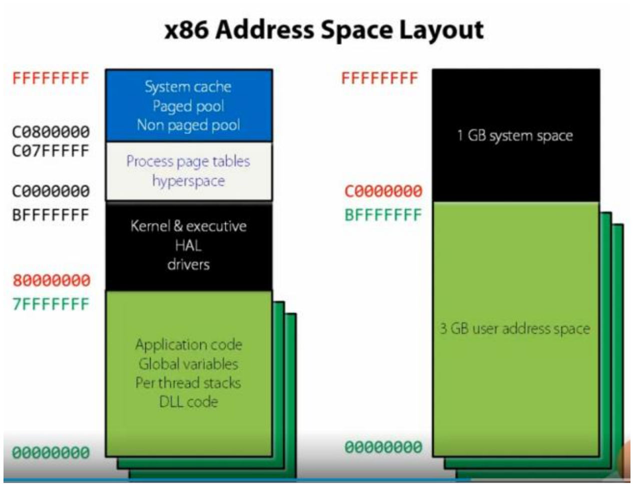
- Code pages are shared between processes.
  - Two or more processes based on the same image.
  - DLL code (however, DLLs must be loaded in the same address).
- Data pages (read/write) are shared at first.
  - They are shared with special protection called Copy-On-Write.
  - If one process changes the data, an exception is caught by the Memory Manager, which creates a private copy of the accessed page for that process (removing the protection).
  - No other process would see this change.
- Data pages can be created without Copy-On-Write.

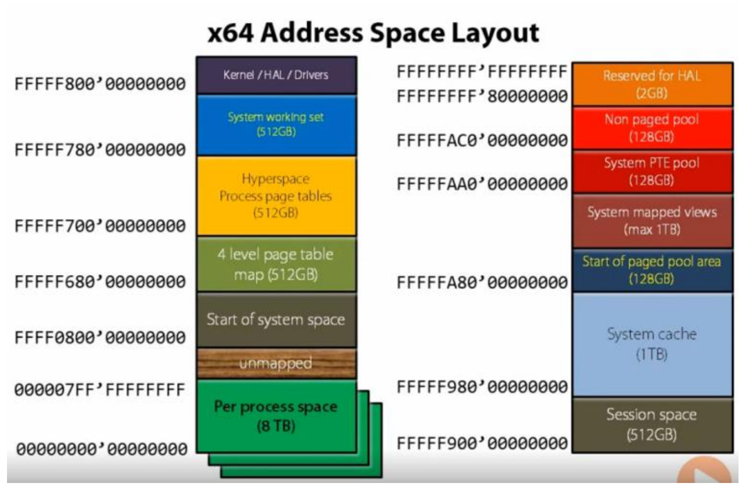
Page Directory

- It's one per process and physical address of page directory stored in KPROCESS structure.
- While a thread is executing, the CR3 register stores its address.
- When a thread context switch occurs between threads of different processes, CR3 is reloaded from the appropriate KPROCESS instance.

Virtual Address Space Layout

- Each process sees its own private address space.
- System Space is part of the entire address space visible but not accessible by user-mode.
- The layout depends on the "bitness" of the OS and the specific process.

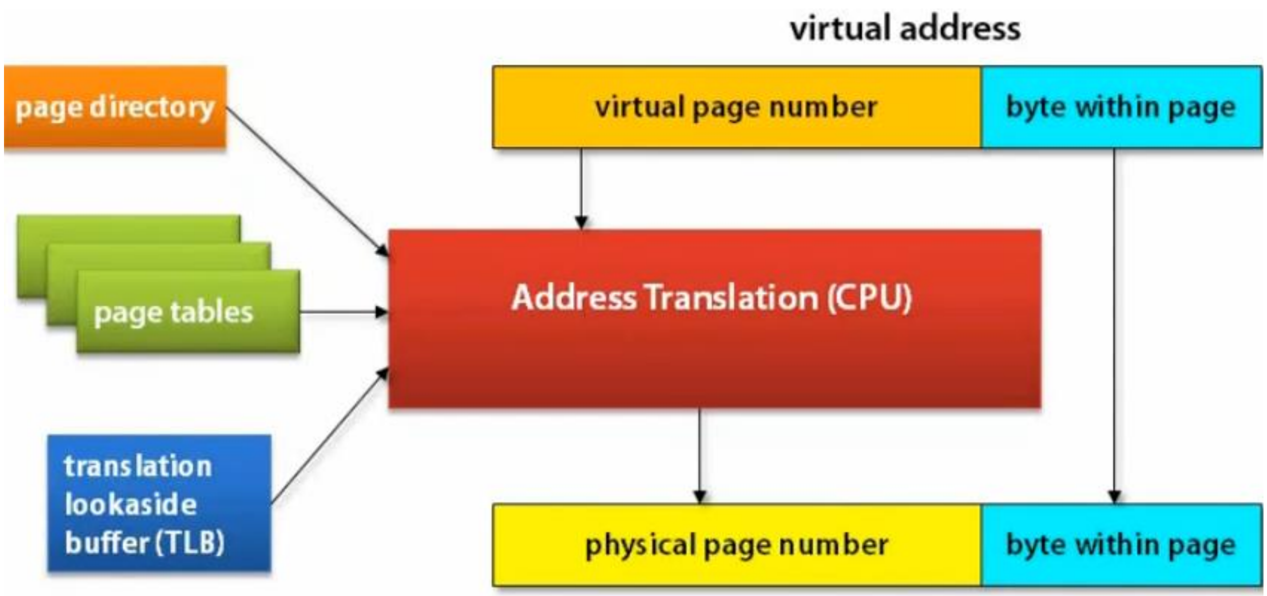


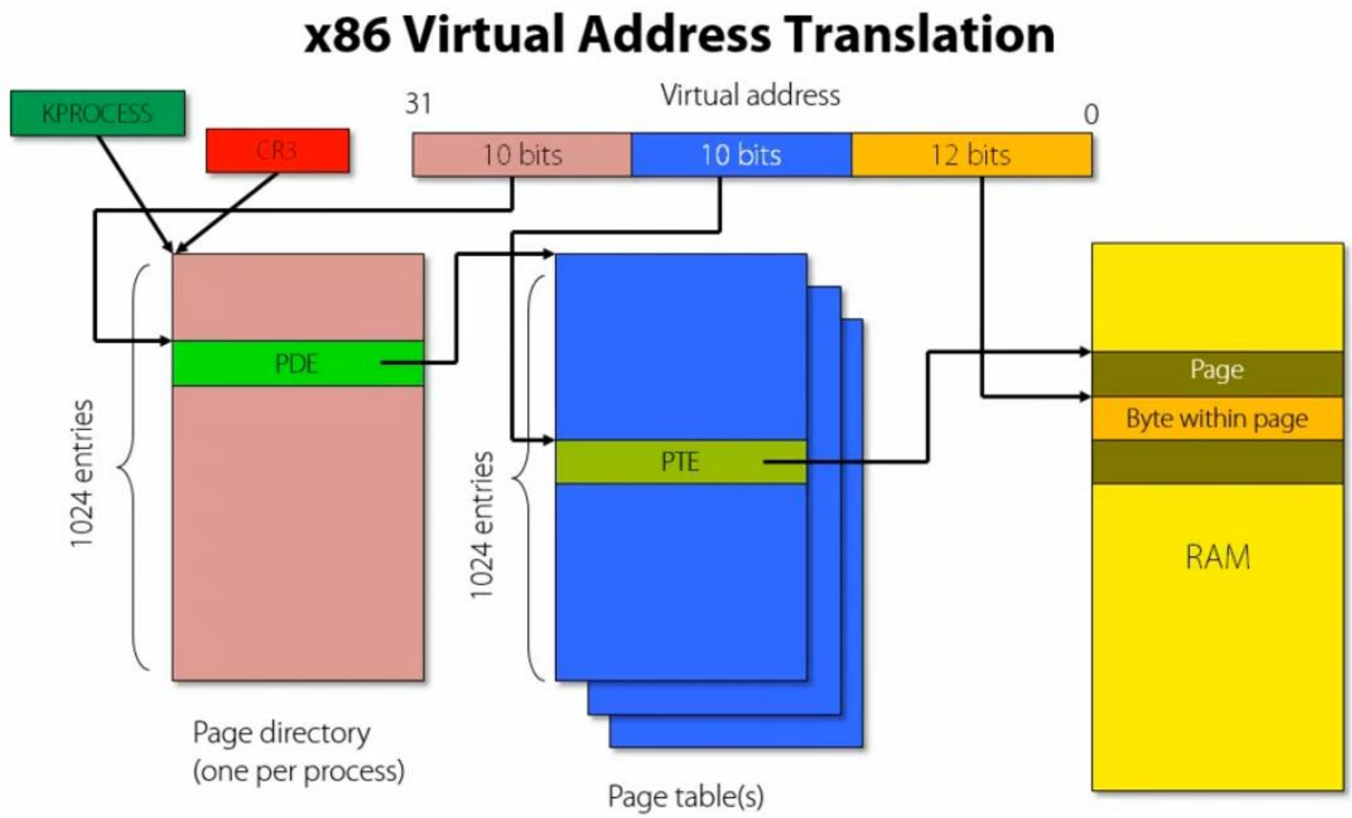


x64 Address Limitations

- 64 bits of addresses can reach to 2^64 = 16EB which is unimaginable amount of memory.
- Current CPU architectures only support 48 bits addressing.
- Current kernel implementations can work with 16TB at most.

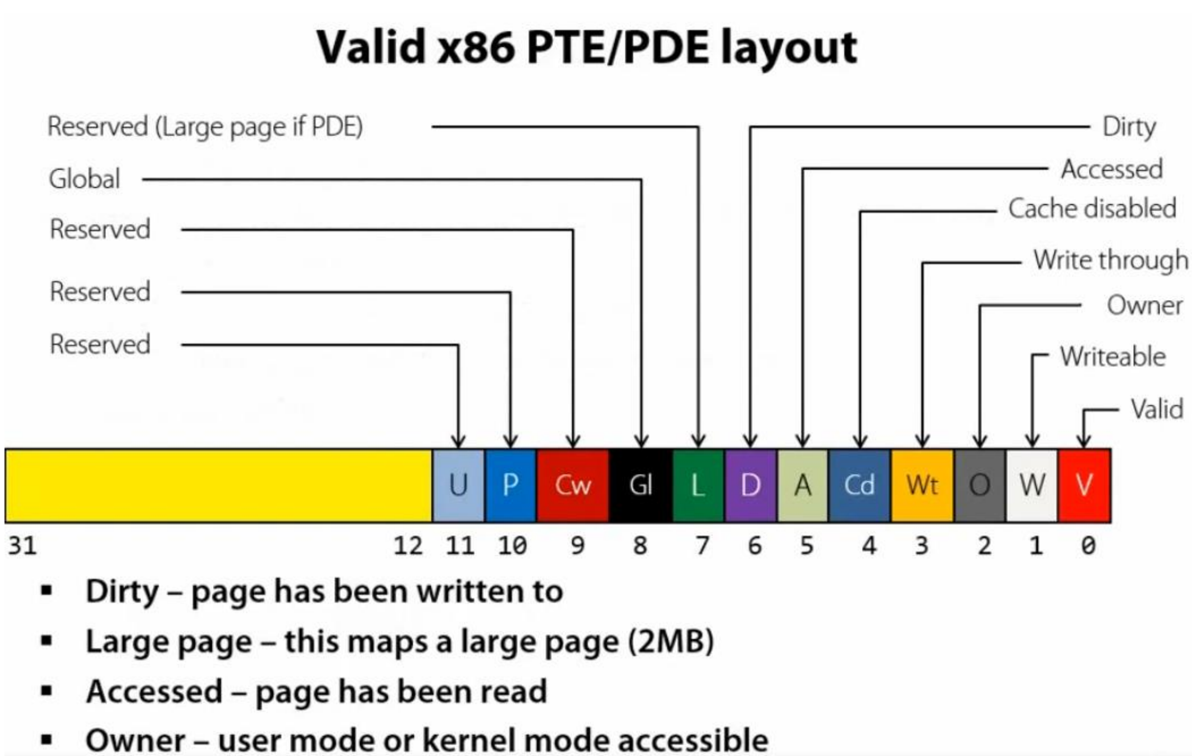
Virtual Address Translation





x86 PDE and PTE

- Each entry is 32 bits (64 bits on PAE).
- Upper 20 bits is the Page Frame Number (PFN).
- Bit 0 is the Valid bit (If set, it indicates that page is in RAM. Otherwise, accessing the page causes a page fault).

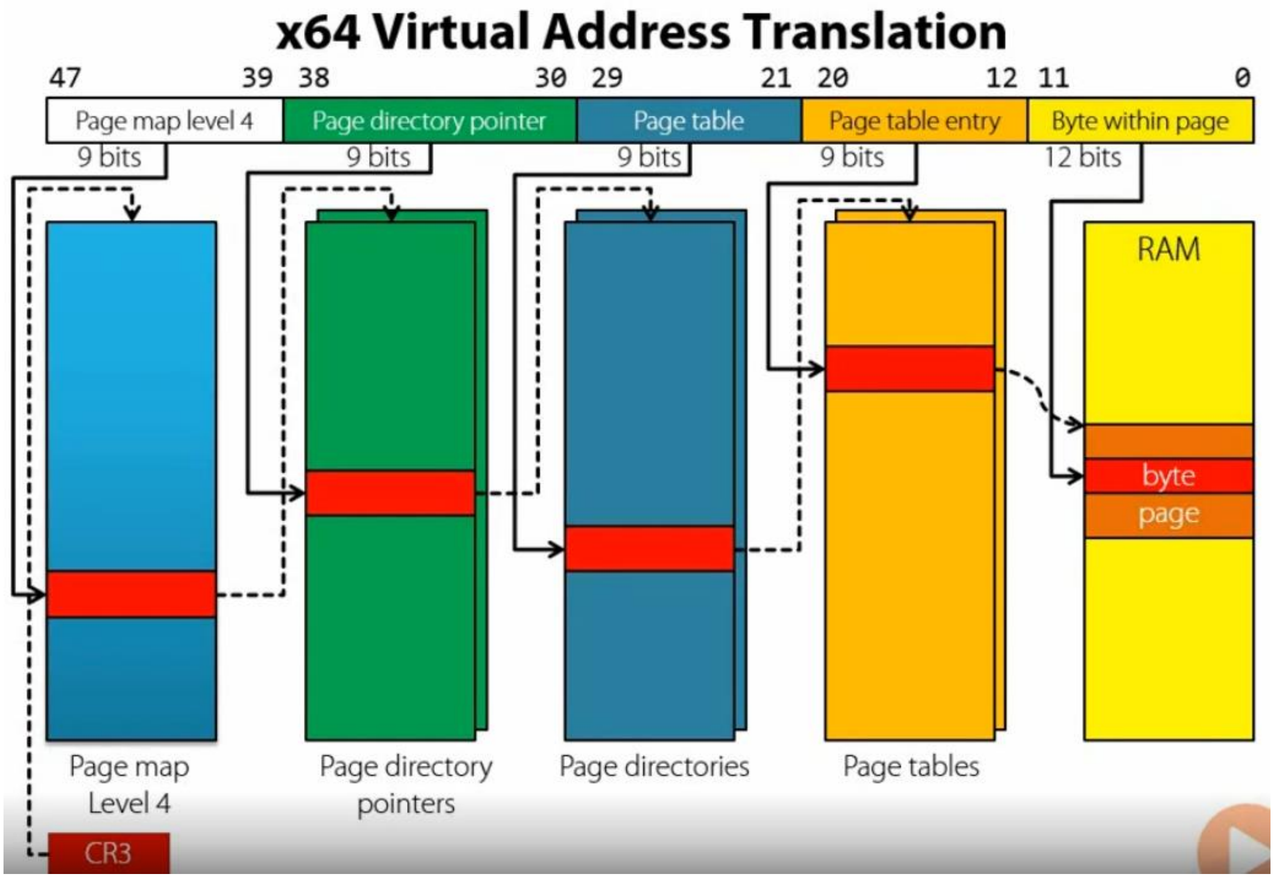


Physical Address Extensions, PAE

- Intel Pentium Pro and later processors support a new PAE mode.
- Virtual address translation contains an extra level of indirection.
- Each PTE/PDE is 64 bits, of which 24 are the PFN.
- Default 32-bit kernel is the PAE kernel.

x64 Virtual Address Translation

- The idea is the same but with more tables and indirections.



Page Faults

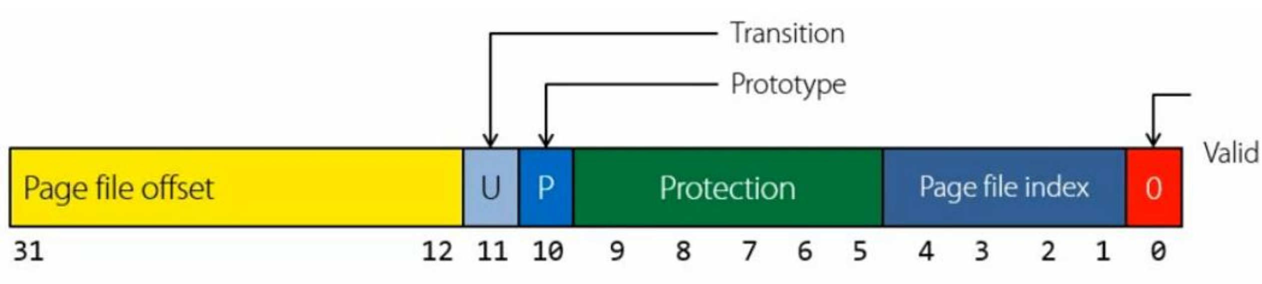
- Valid PTE/PDE results in the CPU accessing data in physical memory.
- If PTE/PDE is invalid, the CPU throws a page fault exception.
- Windows has to get the data from disk, fix the required PTE/PDE and let the CPU try again.
- Page fault types:
  - Hard page fault – requires disk access.
  - Soft page fault – does not require disk access.
    - Example: a needed shared DLL is simply directed to the process by pointing PTE to it.

Some Reasons For Faults

Reason for fault	Result
Accessing a page that is not in RAM but in a page file or mapped file	Allocate a physical page, read the data from disk and add page to the working set
Accessing a page that is in the modified or standby page list	Move the page to the working set
Accessing a page that is not committed	Access violation
Accessing a page from user mode that can only be accessed from kernel mode	Access violation
Writing to a page that is read only	Access violation
Accessing a demand zero page	Add a zeroed page to the working set
Writing to a guard page	Guard page violation (if part of a thread's stack, commit more memory and add to working set)
Writing to a copy-on-write page	Make a process private page copy and replace in working set
Executing code in page marked no-execute (if hardware supports it)	Access violation

Invalid PTEs

- The CPU throws a page fault exception when the Valid bit in a PTE is clear.
- Windows uses the other PTE bits to indicate where the required page can be found.
- Example: a page that resides in a page file (x86 w/o PAE).



Page Files

- Backup Storage for writeable, non-shareable committed memory:
  - Up to 16 page files are supported.
  - Initial size and maximum size can be set.
  - Named PageFile.Sys on disk (root partition).
  - Created contagious on boot.
  - Initial value should be maximum of normal usage.
- Page files information in the registry.
  - HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\ Memory Management\ PagingFiles
- Page file maximum size is 4GB for x86 original kernel and 16TB for x64.

Commit Charge

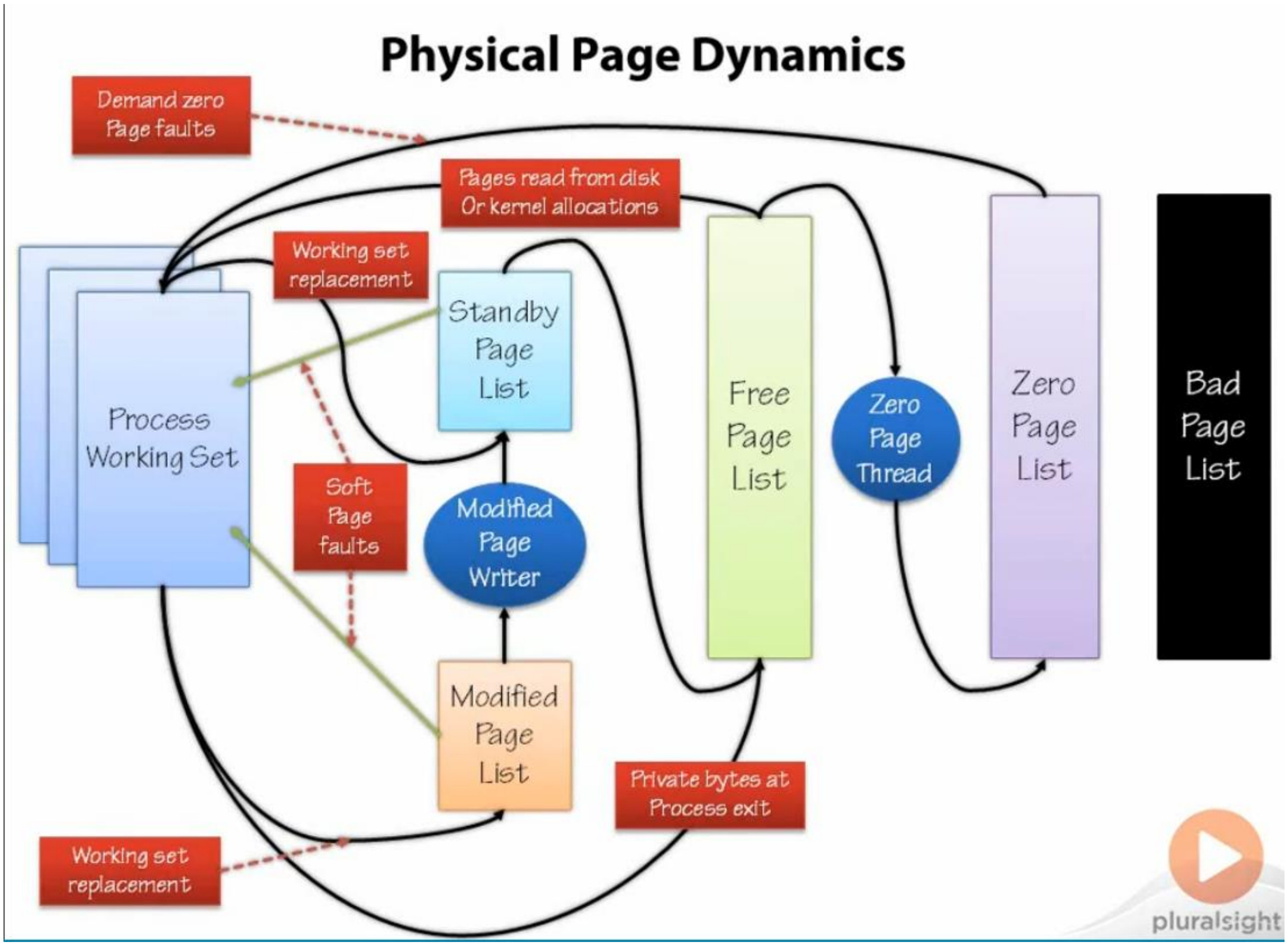
- It represents the memory that can be committed in RAM and existing page files.
- Contributors to the commit charge:
  - Private committed memory (VirtualAlloc with MEM_COMMIT flag).
    - No RAM or page file is used until memory is actually touched until then considered demand zero pages.
  - Page file backed memory mapped file allocated with MapViewOfFile.
  - Copy on write mapped memory.
  - Kernel non-paged and paged pools.
  - Kernel mode stacks.
  - Page tables and yet-to-be-created page tables.
  - Allocations with Address Windowing Extensions (AWE) functions.
- The Commit limit is basically the amount of RAM + maximum size of all page files.
- If a page file grows, the committed limit can grow with it.

Working Sets: the amount of physical memory used by some entity.

- Process Working Set: The subset of the process committed memory residing in physical memory.
- System Working Set: The subset of system memory residing in physical memory.
- Systems with terminal services.
- Demand paging (when a page is needed from disk, more than one is read at a time to reduce I/O.

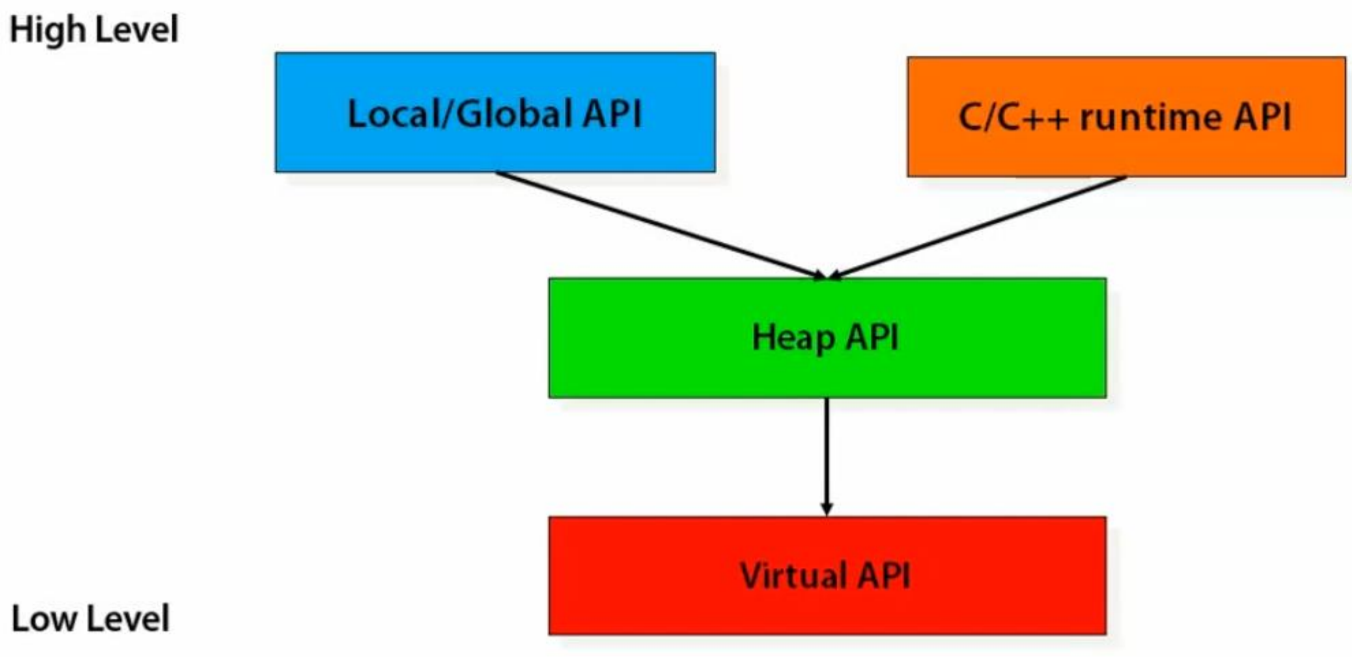
Page Frame Number Database

- It describes the state of all physical pages.
- Valid PTEs point to entries in the PFN database.
- A PFN entry points back to the PTE.
- The structure layout of a PFN entry depends on the state of the page.
- Kernel debugger: !memusage, !pfn.



Memory APIs in User Mode

- Virtual API:
  - VirtualAlloc, VirtualFree, VirtualProtect, etc.
  - Lowest level API
    - Works on page granularity only.
    - Allows reserving and/or committing of memory.
    - Good for large allocations.
- Heap API:
  - HeapCreate, HeapAlloc, HeapFree, etc.
  - Uses the Virtual API internally.
  - Manages small allocations without wasting pages.
- C/C++ runtime:
  - malloc, realloc, free, operator new, etc.
  - Uses the heap API (usually – compiler independent).
- Local/Global API:
  - LocalAlloc, GlobalAlloc, GlobalLock, LocalFree, etc.
  - Mostly for compatibility with Win16.



The Heap Manager

- Allocating in page granularity is sometimes too much (It needs fine grained control).
- It manages smaller allocations (8 bytes minimum).
- The HeapXxx Windows API functions are a thin wrapper over the native NtDll.Dll functions.

Heap Types

- One heap is always created with a process, called the Default Process Heap.
  - It can be accessed using GetProcessHeap.
- Additional Heaps can be created using the HeapCreate function.
- A heap can be fixed in size or grownabe (the default is grownable).
- Low Fragmentation Heap, LFH.

System Memory Pools

- Kernel provides two general memory pools for using by the kernel itself and device drivers.
  - Non-paged pool
    - Memory always resides in RAM and never paged out.
    - Can be access at any IRQL.
- Paged pool
  - Memory can be swapped to disk.
  - Should be accessed at IRQL < DPC_LEVEL (2) only.

- Pool sizes depend on the amount of RAM and the OS type.
  - Can be altered up to some maxima in the registry.
    - HKLM\System\CurrentControlSet\Control\Session Manager\Executive.
- Task Manager displays current sizes.

- APIs
  - ExAllocatePool: Allocate memory from the paged or non-paged pool.
  - ExAlocatePoolWithTag: allocate memory at tag with it a 4-byte value and can be used to track memory leaks.
  - ExFreePool: Frees memory previously allocated on whatever pool.
- Pool usage and tags can be viewed with PoolMon.Exe (It's a part of Windows Driver Kit).
- Known pool tags can be found with the debugging tools for Windows (triage subfolder).

Memory Mapped Files

- Internally called Section objects.
- Allow the creation of views into a file (return a memory pointer for data manipulation).
- Imply shared memory capabilities (usual case with mapping EXEs and DLLs).
- Can create pure shared memory.
  - Backed up by page files.
  - When memory mapped file object destroyed, memory is recycled.
- APIs
  - Win32
    - CreateFileMapping: Creates a file mapping object based on a specific file which was previously created with CreateFile or based on system paging file.
    - OpenFileMapping: Opens an existing MMF based on its name NOT the filename.
    - MapViewOfFileEx: Creates a view into the MMF.
  - .NET
    - System.IO.MemoryMappedFiles
    - Classes: MemoryMappedFile, MemoryMappedViewStream and MemoryMappedViewAccessor.
  - Kernel
    - ZwCreateSection: Creates a section object.
    - ZwCreateFile.
    - ZwMapViewOfSection: Maps a view into the system space.

Large Pages

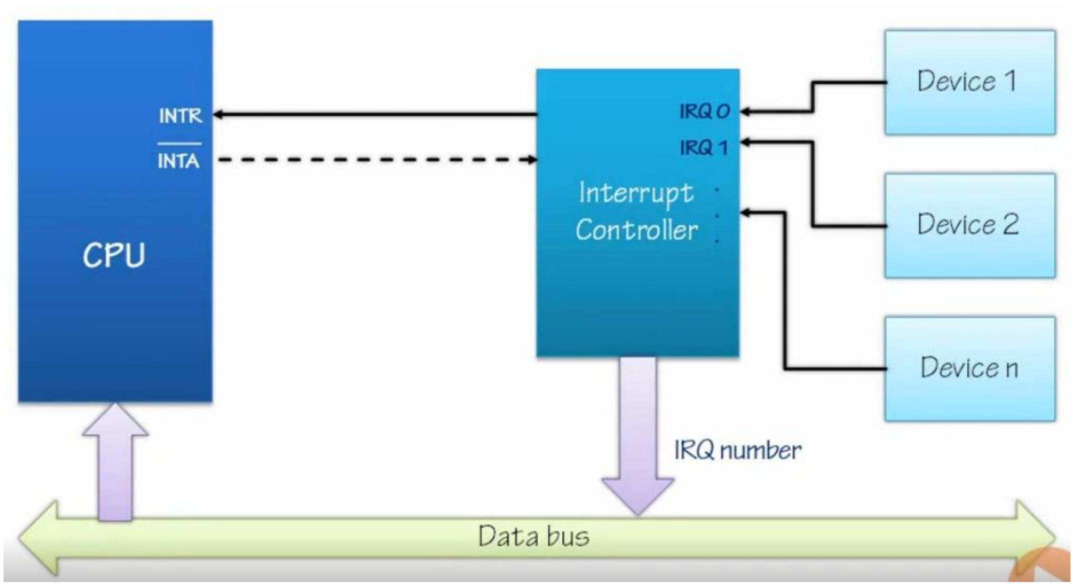
- Large pages allow mapping using a PDE only (no need for PTEs).
  - Advantage: Better use of the translation look aside buffers.
- Windows maps by default large pages for NtOsKrnl.Exe and Hal.Dll as well as core system data (initial non-paged pool and PFN database).
- Potential disadvantages
  - Single protection to entire page.
  - May be more difficult to find a large page size contagious physical memory for mapping.

- Programmatically using large pages.
  - Specifying the MEM_LARGE_PAGE in calls to VirtualAlloc function.
  - Size and alignment must be of large page size.
    - Can be determined by calling the GetLargePageMinimum function.

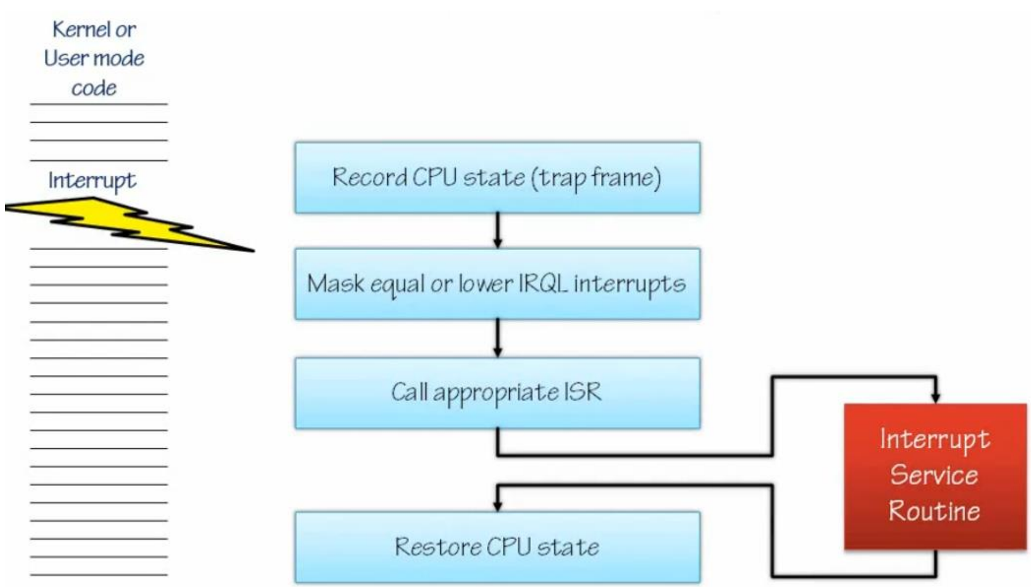
Trap Dispatching

- Kernel mechanisms for capturing an execution thread's state when an interrupt or exception occurs and transferring control to a handling routine.
- Traps: Interrupts or exceptions. Divert code execution outside the normal flow.
- Interrupt: Asynchronous event, unrelated to the current executing code.
- Exception: Synchronous call to certain instructions. Reproducible under same conditions.

Hardware Interrupts

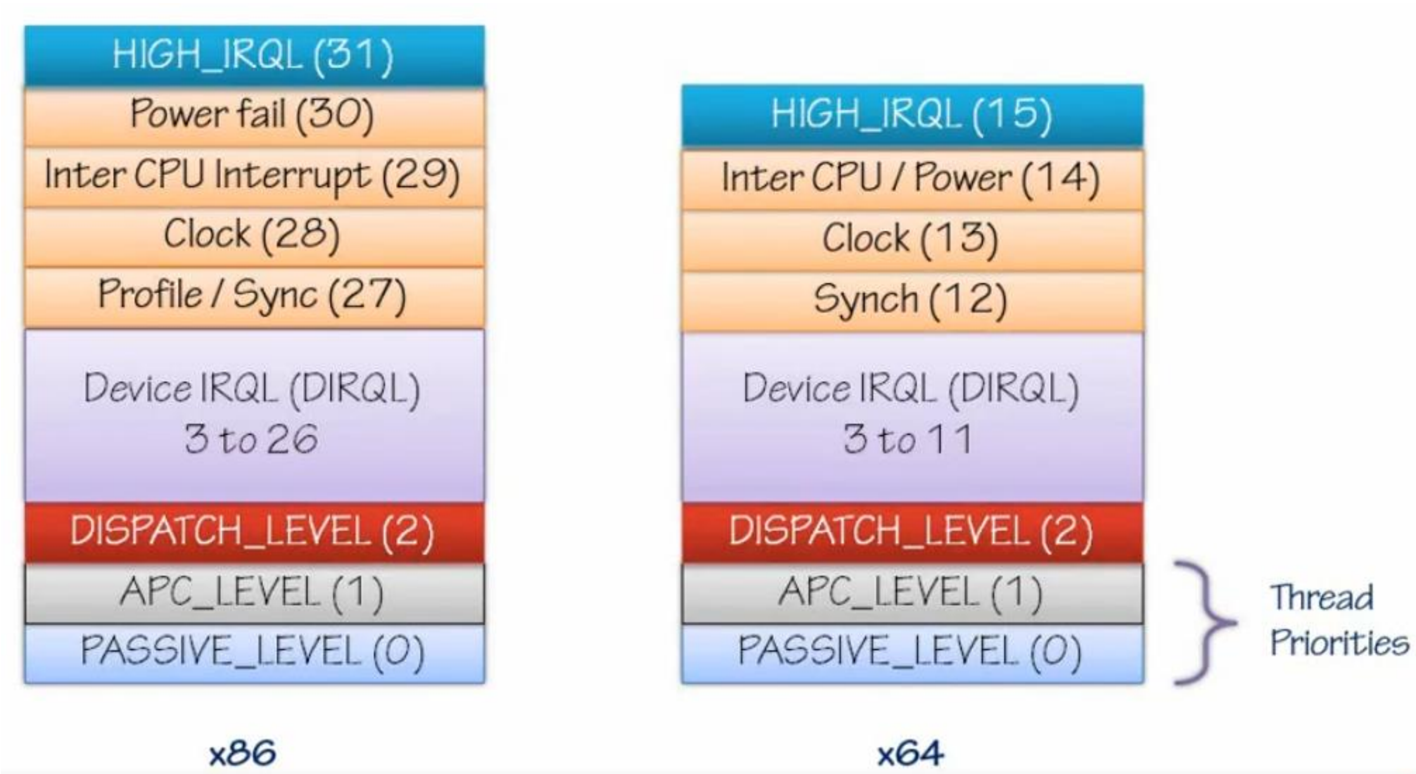


Interrupt Dispatching



Interrupt Request Level, IRQL

- Each interrupt has an associated IRQL.
  - It can be considered its priority.
  - Hardware interrupts are mapped by the HAL.
- Each processor's context includes its current IRQL (CPUs always run the highest IRQL code).
- Servicing an interrupt raises the processor IRQL to the level of the interrupt's IRQL.
- Dismissing an interrupt restores the processor's IRQL to that prior to the interrupt.



- PASSIVE_LEVEL (0)
  - The normal IRQL level.
  - User mode code always runs at this level as well as most kernel code.
- APC_LEVEL (1)
  - Used for special kernel APCs.
- DISPATCH_LEVEL or DPC_LEVEL (2)
  - The kernel scheduler runs at this level.
    - If the CPU runs code at this or higher level, no context switching will occur on that CPU until IRQL drops below this level.
    - No waiting on kernel objects as it requires scheduler.
- Page faults can only be handled in IRQL < DISPATCH_LEVEL
  - Code running at this or higher IRQL must always access non-paged memory.

- Device IRQL, DIRQL
  - Used for hardware devices.
  - The level that an ISR runs at.
  - Always greater than DISPATCH_LEVEL (2).
- HIGH_LEVEL (x86 = 31, x64 = 15)
  - The highest level possible.
  - If code runs at this level, nothing can interfere on that CPU.
  - However, other CPUs aren't affected.
- Other levels exist for kernel internal usage.

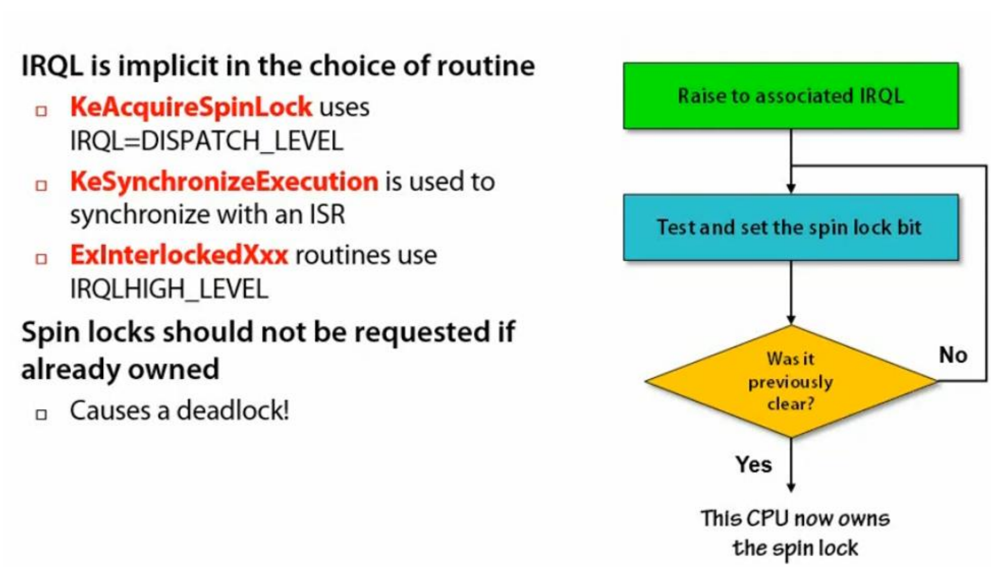
IRQL vs. Thread Priorities

- IRQL is an attribute of a CPU.
- Thread priority is an attribute of a thread.
- With IRQL >= DISPATCH_LEVEL (2)
  - Thread priorities has no meaning.
  - The currently executing thread will execute forever until IRQL drops below 2.
- IRQLs can be changed in kernel mode only using KeRaiseIrql and KeLowerIrql.

The Spin Lock

- Synchronization on MP systems uses IRQLs within each CPU and spin locks to coordinate among the CPUs.
- A spin lock is just a data cell in memory.
  - It is accessed with a test and modifies operation, atomic across all processors.
  - Similar in concept to a mutex.
- Mutex synchronizes threads, Spin Locks synchronize processors.
- Not exposed and not need to user mode applications.

Acquiring a Spin Lock



Exceptions

- Synchronous event resulting from certain code (dividing by zero, access violation, stack overflow, invalid instructions, others).
- Structured Exception Handling: Mechanism used to handle and possibly resolve exceptions.
- Exceptions are connected to entries in the IDT.

Exceptions Handling

- Some exceptions are handled transparently and some are filtered back to user for handling.
- Frame based exception handlers are searched (32-bit systems).
  - If the current frame has no handlers, the previous frame is searched and so on.
  - 64 bit systems don't use frames but search mechanism is the same.
- Unhandled exceptions from kernel mode generate a bug check (Blue Screen of Death).

Structured Exception Handling, SEH

- Exposed for developers by extended C keywords
  - __try: Wraps a block of code that may throw exceptions.
  - __except: Possible block for handling exceptions in the preceding __try block.
  - __finally: Executes code whether an exception occurred or not.
  - __leave: Jumps to the __finally clause.
- Allowed blocks are __try/__finally and __try/__except. However, can be nested to any level
- Works in kernel mode and user mode.
- Custom exceptions can be raised with RaiseException (Win32).

System Crash

- Blue Screen of Death.
- Occurs when an exception in kernel mode was unhandleded.
- Stops everything (code executing in kernel mode is supposed to be trusted).
- If the system crashes it can write crash dump information to a file and be analyzed with dbg.

ادعولي :)

Resolving Exceptions: