Windows IRQL Internals

Authors: astr0 and Ahmed Bahaa

Introduction:

In this paper, we will explain how “Task Priority” and IRQLs work internally in Windows, plus I am providing a poster explaining the “Interrupt Handling” flow with Pentium 4 and Intel xeon processors. At the end of the paper, before we start I would like to give a special thanks to Ahmed Bahaa for helping me alot in formatting the paper. Thank you.

Objectives:

- What are IRQLs?
- Introduction to APIC and x64 Interrupt Handling
- Interrupt vector numbers and Task Priority

Terms:

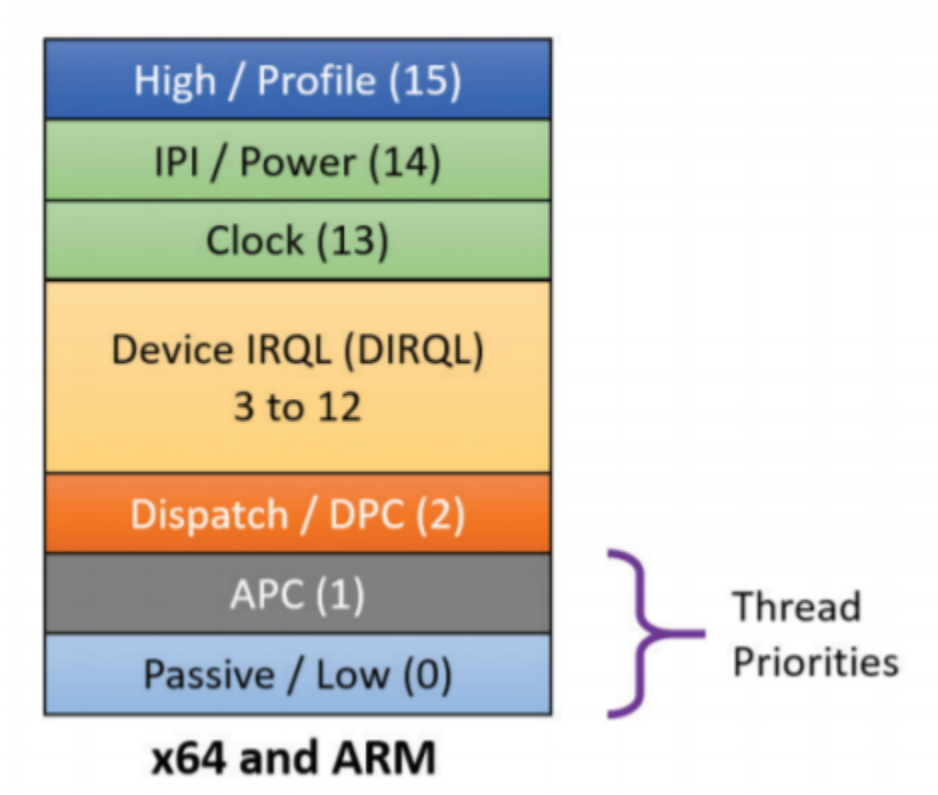
Here are some terms I will be using during the paper, if you feel you don't understand a specific term just refer to it.

- **IRQL**: an IRQL defines the hardware priority at which the CPU operates at any given time.
- **Interrupts**: messages from software/hardware that interrupt the CPU to perform a specific operation causing the cpu to change its instruction pointer to point to the ISR which is the function that will handle interrupt. (eg: keystroke or a syscall operation).
- **Interrupt handlers**: also called ISRs (Interrupt Service Routines). An ISR is a function that gets called when an interrupt is raised, in order to handle it (eg: handling divide errors).
- **TPR**: task priority register is a register in the local APIC that stores the current task priority.
- **MSR**: model specific registers used to store CPU information and control certain features.
- **CR8**: a control register that provides an interface with TPR
- **Page fault**: memory access error caused when attempting to access an address that's not in physical memory or trying to access a page whose page table entry has its NX bit (i.e: memory content cannot be executed) not cleared.
- **APIC**: Advanced Programmable Interrupt Controller.
- **Quantum time**: time given to a thread to run before running any thread after it.
- **External I/O APIC**: receives interrupts from external I/O devices and passes them to local APIC.
- **Local APIC**: receives interrupts from external I/O APIC and passes them to the CPU cores.
- **XAPIC/x2APIC**: extensions to APIC architecture
- **IDT (Interrupt Descriptor Table)**: a table of function pointers to interrupt handlers.
- **Interrupt vector number**: an id given to each interrupt used to index in the IDT.
- **IA32_x2APIC_TPR**: MSR in **x2APIC** that can be used to read/write TPR.
- **EOI (End of Interrupt)**: used by a software to signal / indicate that an interrupt has completed its work or end.
- **IRR (Interrupt Request Register)**: contains accepted interrupts by the local APIC but not yet dispatched to the CPU cores for processing
- **ISR (Interrupt Status Register)**: a local APIC register which holds the interrupts that have been accepted but are still waiting for EOI. The local APIC either queues the interrupt in IRR or ISR.
- **ISRV**: an interrupt vector number of the highest priority bit that is set in the ISR or 0x0 if no bit is set in the ISR.
- **Paged pools**: memory that can be transferred / paged to disk
- **Non paged pool**: memory that can't be transferred / paged to disk.
- **NMI** (non-maskable interrupts): interrupts that can’t be discarded and local APIC passes them directly to the CPU cores
- **SMI** (system management interrupt): like NMI cannot be discarded but they have higher priority and are handled in SMM (system management mode).
- **DPCs** (deferred procedure calls): routines that have a dispatch IRQL used by hardware device drivers to service interrupts.

What Are IRQLs ?:

IRQLs are short for interrupt request levels. They define the priority the CPU operates in, what interrupt should take attention before others, what should be processed first and what should be pended.

The main idea behind IRQLs and task priority is that an interrupt with a lower IRQL value cannot take over an interrupt with a higher IRQL value. Example: if the CPU got two interrupts to process the first with a **dispatch** IRQL which has value of 2 and the second has a **passive** IRQL which has a value of zero, the CPU will process the one with Dispatch first because it has a higher IRQL value numerically.
In x64 windows there are 16 different IRQLs levels:



Passive, Dispatch and APC are IRQLs for software. The rest are for hardware to use.
We will focus on software IRQLs for now.

Passive: normal IRQL that the user / kernel mode code, it has a value of zero. User mode code is always at passive but kernel mode code can be raised to apc or dispatch.

APC: IRQL with value 1 used for apc's and page fault handlers.

Dispatch: IRQL level that the kernel thread scheduler works in. It has a value of 2.
Threads that have dispatch IRQL have an infinite quantum time and so they can't be interrupted by other threads or be suspended.

Routines that have a dispatch IRQL such as called DPCs (deferred procedure calls), shouldn't access or call routines with IRQL < Dispatch and it should only access nonpaged pools, attempting to access page pools causes a bsod because there is page fault handling in dispatch so suppose a memory got paged so it's not in RAM, but on the disk, and a driver tried to access it. This causes a page fault and there is nothing to handle page fault so windows just tends to BSOD to fix what happened.

This explains when you bsod the stop code **IRQL_NOT_LESS_OR_EQUAL**. This stop code happens when a page fault happens at Dispatch level 2 but this page fault can only be handled at Passive (0) or APC level (1), so Dispatch (2) is not equal or less than passive (1) or APC (0).

If you are writing a windows driver to get the current IRQL you can use function like **KeGetCurrentIrql** in assembly its very simple:

```
   public KeGetCurrentIrql
   KeGetCurrentIrql proc near
   mov     rax, cr8
   retn
   KeGetCurrentIrql endp
```

We can see it just moving the CR8 control register value to RAX as the function return value.
So, calling this function is just the same as reading CR8 like this:

```
UINT64 NTAPI KeGetCurrentIrql()
{
   return __readcr8();
}
```

It means that the current IRQL / priority is stored in CR8, but have you wondered why CR8 is used ? and how it relates to IRQLs and CPU interrupt priority. That's what we will discover by digging into APIC.

Introduction To APIC And x64 Interrupt Handling:

Disclaimer: APIC is very well explained in the intel manual vol3. Ch10 I don't want to rewrite stuff and I won't go through all the details. So, I will just summarize what's important and explain how this relates to IRQLs to gain a deeper understanding.

APIC (advanced programmable interrupt controller) has the job of receiving interrupts from internal and external I/O devices, it then transfers them to the CPU cores as interrupt messages to process and handle them.

In SMP it sends and receives IPI (inter-processor interrupt) messages to and from logical processors on the system bus. IPI messages are used to distribute interrupts among system's processors if a processor needs another processor's attention (such as booting up processors, distributing work among a group of processors, shutting down the system, TLBs, etc…

Since in SMP systems all processors are equal and use a shared memory, a processor may change the memory mapping (i.e. a processor can change the memory base address it operates on), this will result in an interrupt from the CPU that changed the memory mapping to other CPUs.

Intel processors has two types of APIC's:

- Local APIC
- External I/O APIC

These two work together, The external I/O APIC is implemented in the intel motherboard chipset, it receives interrupts from external I/O devices and transfers them to the local APIC which resides in the CPU.
The local APIC then transfers the interrupt messages to the CPU cores to start processing them.

Local APIC has its own registers, they are used to control messages, define the interrupt priority, etc.... They are mapped to 4kb. And software can interact with the local APIC through its registers.
For example: we can change the registers memory mapping by modifying the **IA32_APIC_BASE** MSR to another base address instead of the default address (**FEE00000h**), however, system software shouldn't map in regular system memory. Attempting to do this will trigger a **#UD** exception (invalid opcode exception).

Interrupt Vector Numbers & Task Priority:

Each interrupt has a special id called the “interrupt vector number”. They are a set of indexes ranging from 0-255, used as indexes into the IDT in order to see which interrupt handler to call.
Example: if an interrupt has a vector number of 0x1, when indexing IDT, this refers to **nt!KiDivideErrorFaultShadow.** This function is just a shadow or a wrapper for **nt!KiDivideError** which is used for handling divide errors. It has the Id “0x0”. If you tried to disassemble **nt!KiDivideErrorFaultShadow** u will see that it just jumps to **nt!KiDivideError** all shadows works like that just remove the word “Shadow” and you will get the real interrupt handler.

To list these shadows you can execute “!idt” in windbg:

```
Dumping IDT: fffff80666e6c000

00:    fffff80669021100 nt!KiDivideErrorFaultShadow
01:    fffff80669021180 nt!KiDebugTrapOrFaultShadow
02:    fffff80669021240 nt!KiNMIInterruptShadow
03:    fffff806690212c0 nt!KiBreakpointTrapShadow
04:    fffff80669021340 nt!KiOverflowTrapShadow
05:    fffff806690213c0 nt!KiBoundFaultShadow
06:    fffff80669021440 nt!KiInvalidOpcodeFaultShadow
07:    fffff806690214c0 nt!KiNpxNotAvailableFaultShadow
08:    fffff80669021540 nt!KiDoubleFaultAbortShadow
09:    fffff806690215c0 nt!KiNpxSegmentOverrunAbortShadow
0a:    fffff80669021640 nt!KiInvalidTssFaultShadow

...
```

We should also differentiate between interrupts and exceptions, since the first 32 IDT entries not all of them are interrupts, interrupts and exceptions share the same concept that it's an event that needs the CPU's attention. Interrupts may occur at random time during runtime and they happen due to say a keystroke, an i/o request or by executing “int n” instruction, on the other side exceptions happen due to errors like dividing by zero or a page fault. When an interrupt is raised the cpu saves its state and starts a context switch, and then starts to execute interrupt handler in case of an interrupt or an exception handler if it's an exception.

Intel reserves the first 32 [0:31] Interrupt vector numbers such as #DE (divide error), while ones from 32-255 are user defined (i.e: custom made by the OS) and not reserved.

Since we are talking mainly about interrupt priority, each interrupt comes with an interrupt priority class based on its interrupt vector number which is just the bits[7:4] from its interrupt vector number. The lowest interrupt-priority class is 1 and the highest is 15. The higher the interrupt priority class, the higher the priority is.

The CPU uses a special register called TPR (task priority register) in Local APIC to store current priority.
It has this structure:

```
union TPR
{
    UINT64 Flags;

    struct{
       UINT64  TaskPrioritySubClass : 4;
       UINT64  TaskPriorityClass : 4;
       UINT64  Reserved : 24;
    }Bits;
};
```

The **TaskPriorityClass** bits [7:4] are used to store the current task priority (**IRQL** in windows).
When the local APIC receives an interrupt, it compares its **TaskPriorityClass** value against the received interrupt priority class. If **TaskPriorityClass** is less than the received interrupt priority class, it passes it to the CPU cores else, it pends it.

Also, in local APIC there is a PPR register which is used to define the current processor priority, its value is based on the TPR value, It has this structure:

```
union PPR
{
  UINT64 Flags;

  struct
  {
     UINT64  ProcessorPrioritySubClass : 4;
     UINT64  ProcessorPriorityClass : 4;
     UINT64  Reserved : 24;
  }Bits;
};
```

As we said, the PPR value is based on TPR register value and on the ISRV (which is the vector number of the highest priority bit that is set in the ISR or 0x0 if no bit is set in the ISR).
Thus, the value of PPR falls under 3 conditions:

```
If TPR[7:4] > ISRV[7:4], PPR[3:0] is TPR[3:0]

If TPR[7:4] < ISRV[7:4], PPR[3:0] is 0

If TPR[7:4] = ISRV[7:4], PPR[3:0] may be either TPR[3:0] or 0
```

After gaining some information on how task priority works in general, we need to discuss how a software can interact with TPR and modify the task priority**.**
Intel provides two ways:

- To interact with it directly through the local APIC
- Use the CR8 register as an interface, and any change in it will be reflected in the **TPR.** (Windows uses this technique as we saw in KeGetCurrentIrql).

The CR8 structure looks like this:

```
union CR8
{
    UINT64  Flags;

    struct
    {
       UINT64  TaskPriority : 4;
       UINT64  Reserved : 60;
    }Bits;
};
```

We can see that only four bits (TaskPriority) can be used by software. They are used to represent the IRQL / task priority value.

Viewing CR8 using windbg:

```
0: kd> .formats cr8
Evaluate expression:
  Hex:     00000000`0000000f
  Decimal: 15
  Octal:   0000000000000000000017
  Binary:  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00001111
  Chars:   ........
  Time:    Thu Jan  1 02:00:15 1970
  Float:   low 2.10195e-044 high 0
  Double:  7.41098e-323
```

We can see that the lower 4 bits are set to 1 and in decimal its 15, which is the highest the IRQL / task priority class in x64 architecture.

When using the CR8 model the local APIC when receiving an interrupt, it compares the interrupt priority class value bits[7:4] value against CR8 value.

- If “CR8” < interrupt priority class value “bits[7:4]” it sends the interrupt to the CPU cores for handling
- Else, it pends it in ISR or IRR until its priority becomes “> CR8”.

To see how CR8 value reflects to TPR, we can view the TPR by reading **IA32_x2APIC_TPR [0x00000808]** (this MSR is specific to x2APIC and is used to r/w TPR).

```
0: kd> rdmsr 0x00000808
msr[808] = 00000000`000000f0
0: kd> .formats 00000000`000000f0
Evaluate expression:
  Hex:     00000000`000000f0
  Decimal: 240
  Octal:   0000000000000000000360
  Binary:  00000000 00000000 00000000 00000000 00000000 00000000 00000000 11110000
  Chars:   ........
  Time:    Thu Jan  1 02:04:00 1970
  Float:   low 3.36312e-043 high 0
  Double:  1.18576e-321
```

We can see that bits [7:4] are set to 1. These are **TaskPriorityClass** values.
So, we deduce that:

- **CR8[3:0] == TPR[7:4].**
- **CR8** provides an interface to the TPR and its bits [3:0] represents the IRQL / task priority.

Conclusion:

That's all folks, I hope you enjoyed this article. I will try to turn this into a series if i had time and discuss other interrupt handling topics like xAPIC / x2APIC and windows IDT internals, the paper is just an intro and i tried to make it as simple as possible also I am not so much familiar with all windows internals and Intel architecture details and I have knowledge gaps, so if you think I missed something just ping me at **astr0#8214**.

Poster:

References:

Intel Manual Vol3. Ch10

Developing Drivers With Windows Driver Foundation: Interrupt Request Levels Chapter

Windows Internals: Ch6. Interrupt Request Levels And Deferred Procedure Calls

Managing Hardware Priorities

OSDev Wiki: APIC

Not everything is executable