WRITING TOPIC FOUR

INTERRUPTS AND SYNCHRONIZATION

December 4, 2016

Steven Silvers
Oregon State University
CS 444 Operating Systems II

I. Interrupts and Synchronization in Linux

Interrupts in the Linux operating system are split into two main categories, top halves and bottom halves. Top halves deal with hardware events that are timing critical, meaning that a top half interrupt cannot sleep or block. The main function of the top half is to make sure that the hardware is ready to handle the incoming interrupt request.

Anything that does not fall into the top half category is called a bottom half. Bottom halves are not as time sensitive as top halves, so some bottom halves can sleep, block and can also be interrupted. There are three main types of bottom halves currently used in the Linux kernel, and those are softirqs, tasklets and work queues[1]. Softirqs are rarely used, as they cannot be dynamically allocated or destroyed and there is a maximum of 32 softirqs in existence within the kernel. Because of these limitations, most interrupt drivers are written with tasklets. Tasklets are built on and processed by softirqs based on their priority level. The key advantage of using tasklets is that they can be dynamically allocated and destroyed[1]. Work queues are completely different from softirqs and tasklets in that while the first two run in interrupt context, work queues run in process context. This means that they run like a normal process, with the ability to sleep and block. This allows work queues to perform tasks such as memory allocation and I/O.

Synchronization is necessary in every modern operating system, which includes Linux. Synchronization tools are needed whenever processes might execute concurrently to prevent the creation of race conditions. To attempt to prevent race conditions in Linux, a concept called critical sections was implemented. A critical section is a segment of code that is protected from concurrent multiple access[2]. Whenever a thread is operating within a critical section, all other threads are excluded from that critical section. The problem this can raise is that of deadlock. If their are two protected resources, and two different threads each are currently accessing one of resources but need to use both resources to finish executing their process, neither thread will be able to complete their process as they will permanently lock each other out of their respective resources.

One way that synchronization is implemented in the Linux kernel is called a spinlock. A spinlock is a busy-wait method of mutual exclusion. If the lock is available, the thread takes the lock and runs what it needs to run and then releases the lock. If the lock is unavailable the thread busy-waits on the lock until it becomes available again[2]. Other synchronization methods in the Linux kernel include atomic functions, reader/writer locks, mutexes and the Big kernel Lock.

II. Interrupts and Synchronization in Windows

At a high level, the Windows operating system handles interrupts in a fairly similar way to Linux. Only when you get into the finer details do you start to notice a difference. Instead of top halves and bottom halves, Windows uses what it calls ”traps” which are mechanisms in the kernel that capture a thread that is currently executing when an interrupt occurs. Control is then transferred to a trap handler, which is responsible for running a function that is specific to whatever interrupt or exception that occurred[3]. The kernel distinguishes interrupts from exceptions in that interrupts are asynchronous events that have nothing to do with what the CPU was currently running, such as I/O. Exceptions are synchronous and are usually caused by whatever the CPU was running at the time.

The Windows operating system sets interrupt priority with a system called Interrupt Request Level (IRQL). This is represented in the kernel on x86 systems as number 0 through 31. The higher the number, the higher the priority of the interrupt. A level 0 means to run it at regular thread execution, 1 and 2 are software level interrupts and 3 through 31 are all various types of hardware interrupts[3]. A higher level interrupt can always interrupt a lower level interrupt, causing the processor to save what was already done and then call the trap dispatcher for the higher level interrupt. This is different from Linux, where there are many interrupts that cannot be interrupted and disable other interrupts from happening.

Synchronization within the Windows operating system follows many of the same principles that Linux does, just with slightly different implementation. Windows also makes use of the idea of mutual exclusion to prevent race conditions from happening. Windows uses different methods of mutual exclusion based on the priority level of the IRQL. When dealing with high-level IRQLs Windows makes use of Spinlocks, as described in the Linux section. Windows mostly uses what it calls a Queued Spinlock, where processors that want the lock are held in a queue and instead of waiting on the lock itself they wait on a pre-processor flag[3]. This reduces that amount of traffic on the

bus, as well as creates a FIFO order of which processor gets the lock, instead of giving it to a random processor. For low level IRQLs, Windows has what it calls kernel dispatcher objects that can be used in the Windows API. Examples of these objects are semaphores, mutexes events, and waitable timer[3].

III. Conclusion

The implementation of interrupts in Linux and Windows follow the same high level algorithm of interrupt happened, stop what we are doing, handle interrupt, resume what we are doing. Only when you begin to look at the specific implementation of how interrupts are handled do you start to see differences in the two operating systems. The same can be said for synchronization, both operating systems are trying to prevent race conditions from occurring. There are slight differences in implementation, but ultimately the concepts are the same.

REFERENCES

[1] K. McGrath, *Operating Systems*. Top Hat, 2016.
[2] T. Jones, "Anatomy of linux synchronization methods." http://www.ibm.com/developerworks/library/l-linux-synchronization/. Accessed: 12-3-2016.
[3] D. S. M. Russinovich, *Windows Internals*. Microsoft Press, 6th ed., 2012.