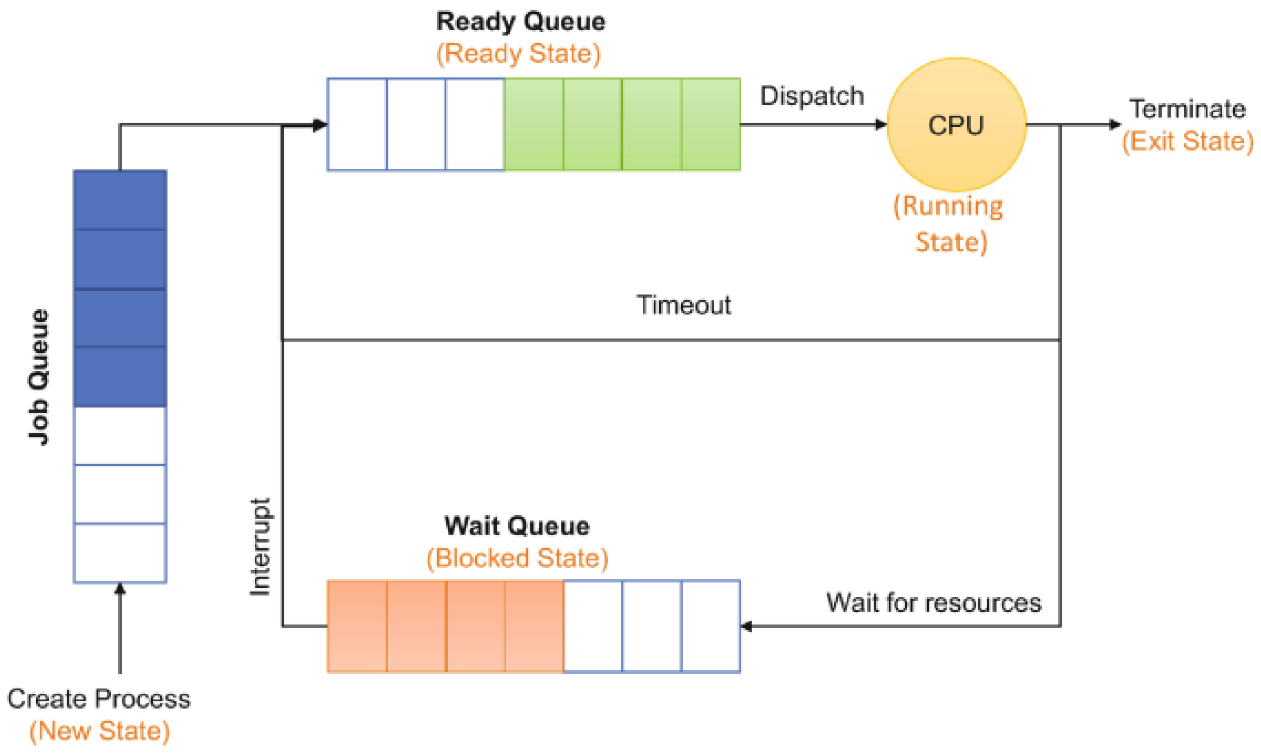
Implementing a Dispatcher in x86-64

By Michael Almandeel

We have learned a lot about operating systems in this course, but the one thing we didn't look very closely at is the dispatcher. For most students, this is a topic shrouded in mystery. We know what it does, but not how it actually works. I have been wondering the same thing; this paper outlines my ideas about how such a thing can work. I will introduce a simple dispatcher and gradually add functionality, until we have something that resembles a dispatcher that may be used in a multi programmed operating system.



So what is a dispatcher? At the simplest level, a dispatcher is responsible for saving the state of and suspending a currently running program, and restoring the state of and resuming some other program. This however, is only part of the picture. The dispatcher is also responsible for dispatching interrupts to the appropriate program they were intended for , and at times even calling a handler for such an interrupt which was defined by the target program(the program the interrupt was intended for). Further than that, correct function of the dispatcher depends on interrupt handlers performing additional work which enforces dispatcher policy(enforcing burst time for instance). All of this said, the most apt description of a dispatcher is an “interrupt manager”. In my implementation, the dispatcher is simply a timer interrupt handler, with some additional reentrant code used by all of the interrupt handlers. Before we delve into how all of this works, we have to develop a stronger understanding of interrupts, the local apic timer, and some details about memory.

Interrupts in x86-64

Interrupts are propagated to the cpu core by way of an *advanced programmable interrupt controller* , or apic for short. Each core has it’s own local apic, and there are one or more I/O apic’s connected to the system bus. While the I/O apic for the most part propagates I/O interrupts, the local apic does quite a bit. Most notably, the local apic comprises a timer, 255 interrupt channels(the first 32 reserved), and is responsible for sending and receiving IPI’s, or inter processor interrupts. IPI’s are notably used for cache invalidation and scheduling.

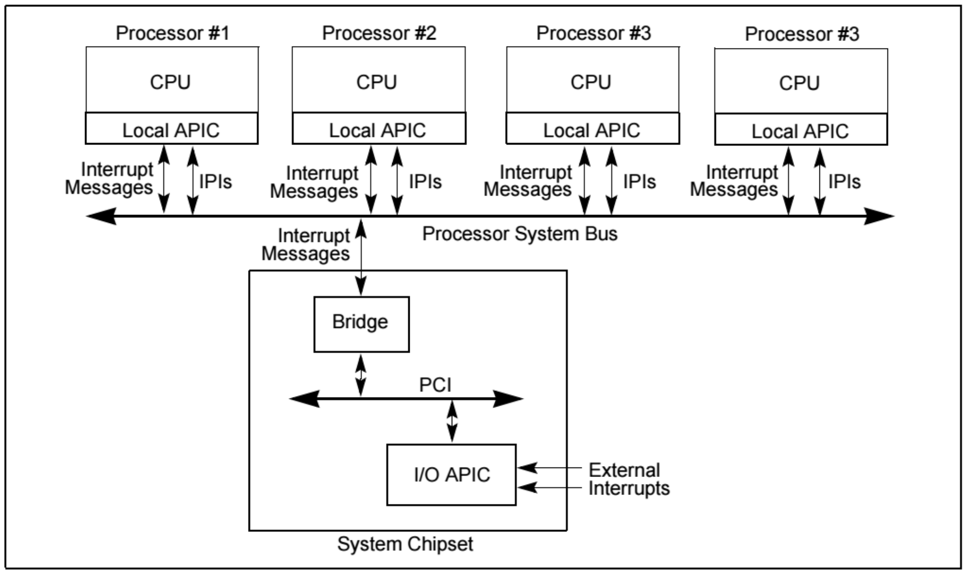


Figure 10-2. Local APICs and I/O APIC When Intel Xeon Processors Are Used in Multiple-Processor Systems

We restrict our discussion of the local apic to the timer, because that is the function we must utilize in order to implement our dispatcher. Each local apic has an LVT, or local vector table. This allows you to configure what interrupt will be propagated in response to certain events that the local apic monitors or simply manages itself. The LVT allows you to configure the mode of the timer and the channel it will fire on. The LVT also allows you to configure the vector and mode of response to bus pins, and the thermal management and performance apparatus's.

It is worth mentioning that there are two classes of interrupts: maskable and non-maskable(NMI) interrupts. A NMI cannot be interrupted until the IRETQ instruction is executed. NMI's are system critical interrupts, so for this reason they can interrupt a maskable interrupt, but cannot be interrupted themselves. Maskable interrupts can be blocked. They are blocked(and queued) while another interrupt is being serviced, and will be unmasked(allowed to fire) after all higher priority interrupts are serviced. This information will be useful later.

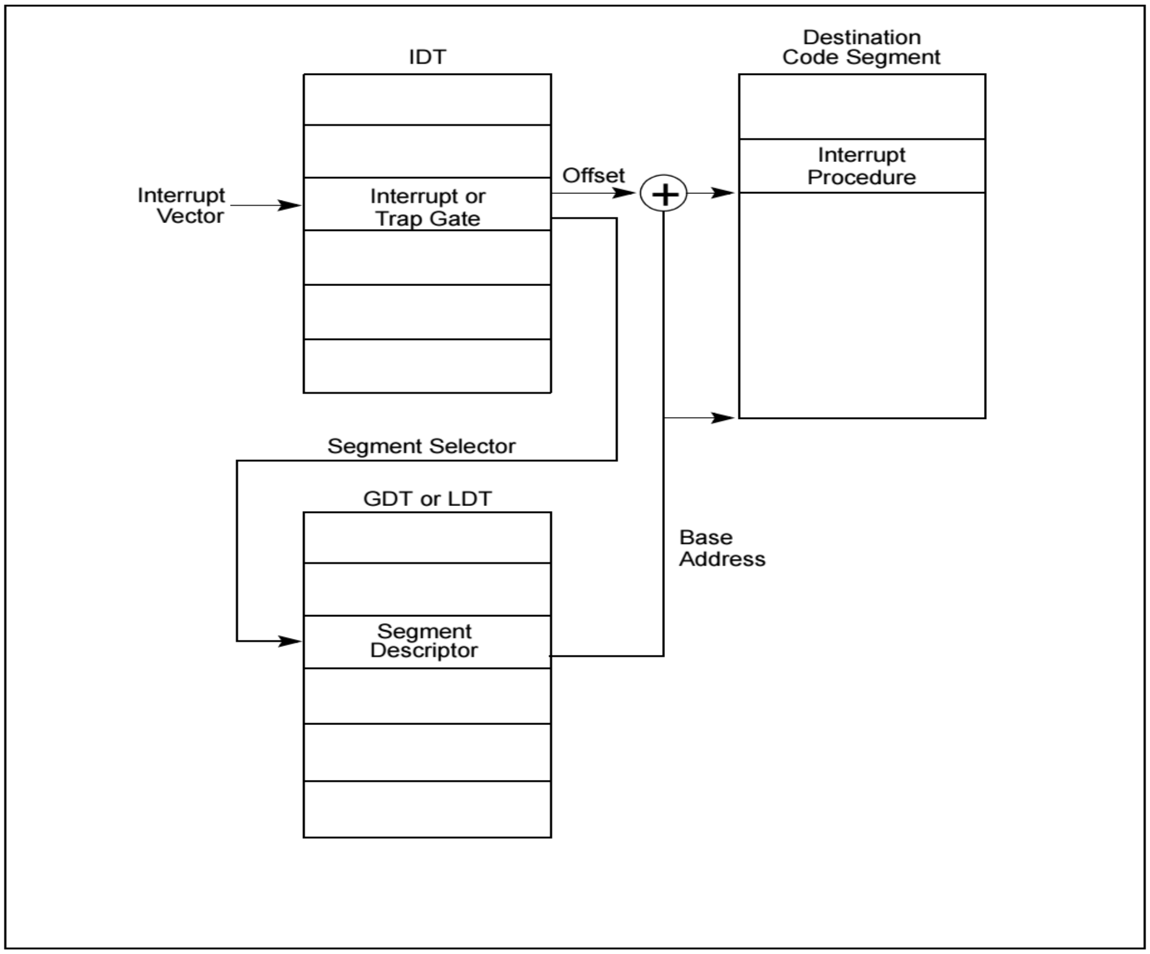
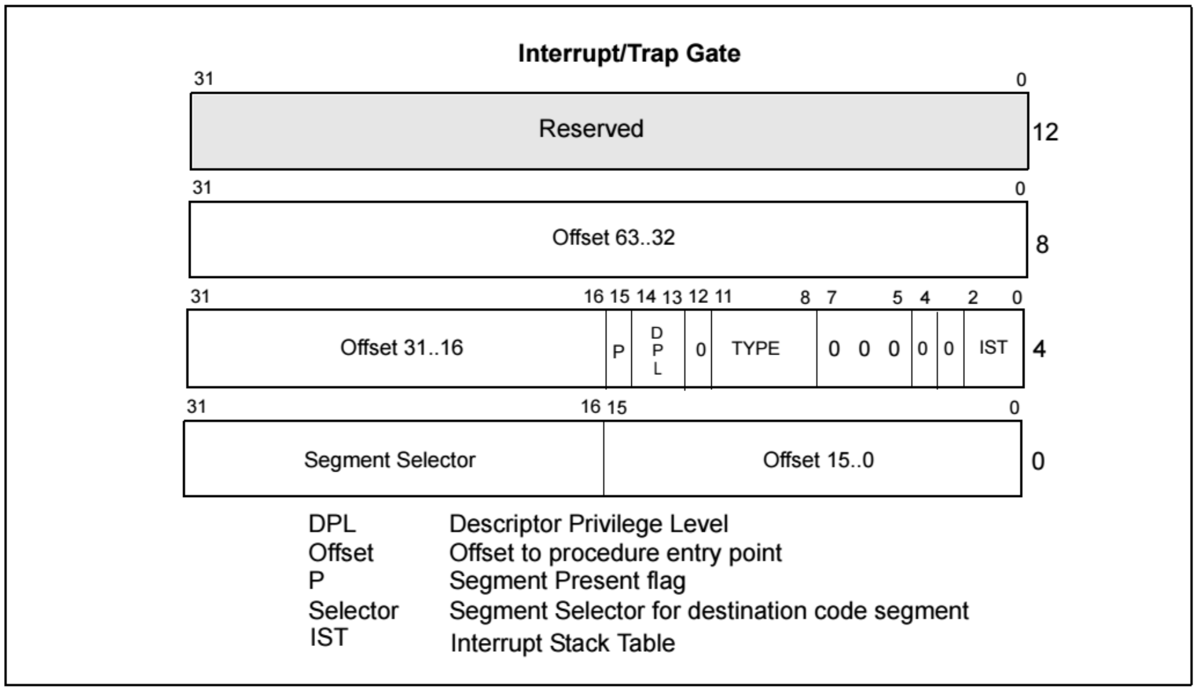


Figure 6-3. Interrupt Procedure Call

When an interrupt is received by the cpu core, the currently running program is suspended, and the cpu looks up the appropriate handler in the IDT(interrupt dispatch table). The corresponding IDT entry contains a segment selector, part of which is used as an offset into the GDT(global descriptor table). The GDT holds various types of descriptors that reference various things, but we are interested in the interrupt gate descriptor, which is the result of our lookup in the GDT.



DPL	Descriptor Privilege Level
Offset	Offset to procedure entry point
P	Segment Present flag
Selector	Segment Selector for destination code segment
IST	Interrupt Stack Table

Figure 6-8. 64-Bit IDT Gate Descriptors

The above figure describes the structure of an interrupt gate. The most important thing to notice is the offset, which we use as an entry point into the interrupt service routine which it corresponds to. The descriptor privilege level(DPL) indicates the highest privilege level that may access the isr this descriptor references. If the dpl is 0, it is only accessible in privileged mode. If the dpl is 3, anyone can initiate it. Dpl's of 3 are typically used for software interrupts. The IST is a new feature which allows the user to define their own set of interrupt stacks. This can be useful if you don't want interrupts to piggy back on the stack your currently using. You must initialize a tss(task state segment) to store it however. Another important flag here is the P flag. If you set P=0, it prevents interrupts that reference this descriptor from being serviced at all. This is useful if you want to temporarily block an interrupt(s).

After the descriptor is examined and found to be accessible, the cpu pushes the interrupted task's stack segment, stack pointer, flags, code segment, and instruction pointer(the error code in the figure represents what happens on an exception, which is beyond the scope of this paper).

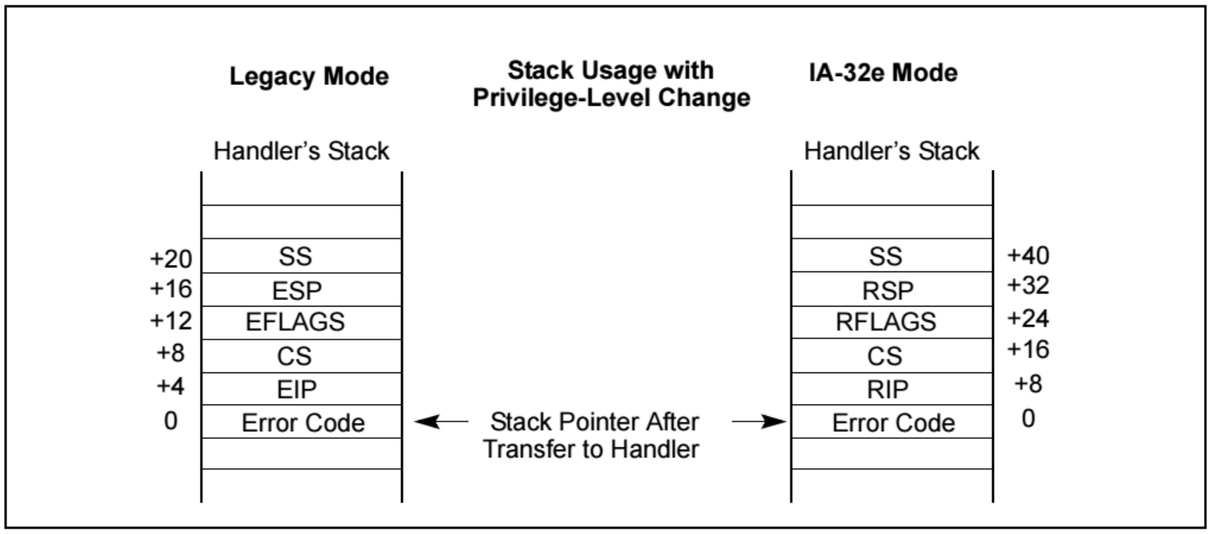
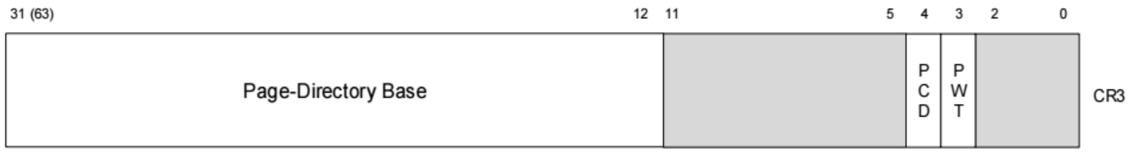


Figure 6-9. IA-32e Mode Stack Usage After Privilege Level Change

At this point, the cpu begins executing the code in the isr. When it is finished, it executes the IRETQ instruction, restoring the pushed stacks to their respective register and returning control to the user. It's important to note that when writing an isr, you should push any registers you intend to use to the stack beforehand, and pop them back into their respective registers before calling IRETQ. While executing, an isr can only be interrupted by an NMI , which cannot be interrupted at all. So in the worst case, we may be dealing with a singly nested interrupt. This is a concern which we will address later when we design our dispatcher.

Memory Management Concerns

We learned in class that each program gets its own PID. Intel has the same concept, but it's not-so-aptly named the 'CR3' register, the last 19 bits of which indicate the page table index.



The page table index is what allows each process to have the linear address space effectively to themselves. The master page table is partitioned into smaller page tables, each of which corresponds to its own page table index. Typical isr's are reentrant code that run in privileged mode within the interrupted process's address space. Generally, CR3 is not changed. However, when calling a dispatcher, it must access kernel data structures, which typically reside on page table index 0, which means we must hop laterally among page tables.

Although this may sound trivial, it's an enormous hurdle to overcome. Let's consider more carefully what happens when we change cr3 in the midst of executing code. After we update cr3, and we increment the instruction pointer, it will not reference the line of code that comes next, because this code resides in our previous page table. Instead, it tries to execute whatever is at that address within a now entirely new linear address space. Without very careful coordination, this is a recipe for disaster.

One solution is to place the kernel data structures in a privileged subset of your own page table by modifying a subset of the page tables *protection key*. However, keeping copies of kernel data structures in all page tables is a synchronization nightmare, and would require very significant overhead for the kernel to guarantee consistency. The solution then, is to place code segments strategically in kernel space so the entry point is the next linear address that the instruction pointer will access after setting the CR3 register to 0. In this case it will jump to the appropriate location in the appropriate address space.

Designing a Simple Dispatcher

Now that we are aware of the challenges, let’s start with a simple dispatcher that context switches between tasks after they have ran for the appropriate burst time. After that, we will augment this model with more features, until we have something that has the functionality of a dispatcher you might find in a modern operating system.

For the simple dispatcher, we are only concerned with the timer interrupt. We are going to pretend that other interrupts simply do not occur. The dispatcher selects the pcb that is scheduled by the scheduler, restores all registers that aren’t necessary to set the timer, sets the timer to ‘burst time’ and loads the remaining registers, with the exception of the instruction pointer and stack pointer (it needs these to execute). It sets CR3 and causes a jump to a stub with a linear address of rip+1 in the target address space. The stub contains code to restore these registers and push the appropriate values to the stack, and IRETQ to the location of the instruction pointer for that task, which is now in its own address space. One thing we need to make this possible is reserving some registers to store address values, so our stub can simply load these values from predisposed registers. There is likely a way to access other address spaces from the kernel without setting CR3, but I am not well versed enough to be aware of it yet.

So now that that's done, the program we dispatched will run for its burst time(or close to it) before being interrupted by the timer interrupt. When that happens, it passes control to the dispatcher, and it performs another context switch to a new task, setting its burst time accordingly. Of course if any interrupts occur while a task is running, the isr is going to hijack the task's burst time, which is something we don't want. Our next iteration deals with precisely that.

Maintaining Timer Consistency

We mentioned above that if a task is interrupted during its burst time that its cpu time would effectively be hijacked by the interrupt. To prevent this, we are going to have to patch the isr's code with instructions that cause them to jump to different code in the dispatcher, which handles precisely this. In order to do it, the dispatcher must have information about the interrupted task's burst time, when it began, and what time it is now. Immediately after its called, the isr pushes the timestamp. After the isr completes its execution, it loads the timestamp and return address information(as above) into the appropriate registers, and jumps into the dispatcher as described above, but into a different entry point. This entry point causes the dispatcher to cancel the timer by setting 'initial count' to 0. It then resets the timer as such:

set_timer(burst_time-(interrupted_time-start_time).

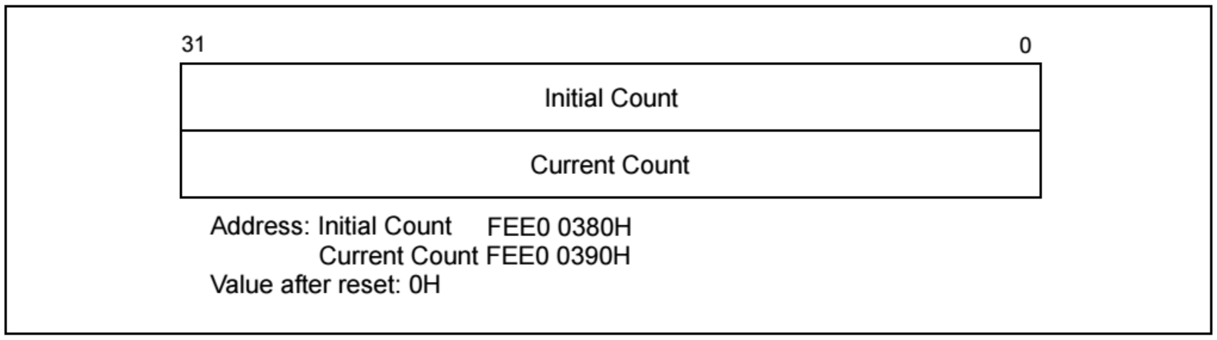


Figure 10-11. Initial Count and Current Count Registers

The dispatcher then uses the same stub technique to return control to the initial isr, who then restores the registers it used and calls IRETQ to return to the interrupted program.

Although this will work fine for one interrupt at a time, we also have to consider the scenario in which an interrupt is interrupted by an NMI. With the existing design, it would cause timer consistency issues and make the burst time inconsistent. Thankfully, the fix isn't too difficult. We make a minor change to our current design, by having the isr check its stack after pushing the time stamp. Note that the rflags register(eflags in the figure below), has the IF, or interrupt enabled flag. When running in an isr, we check rflags, which is in a predictable place, and determine whether or not we interrupted another interrupt. If we didn't, we proceed as described. If we did however, we simply pop the timestamp and IRETQ as usual to the initial interrupt after we are done , allowing it handle calling into the dispatcher.

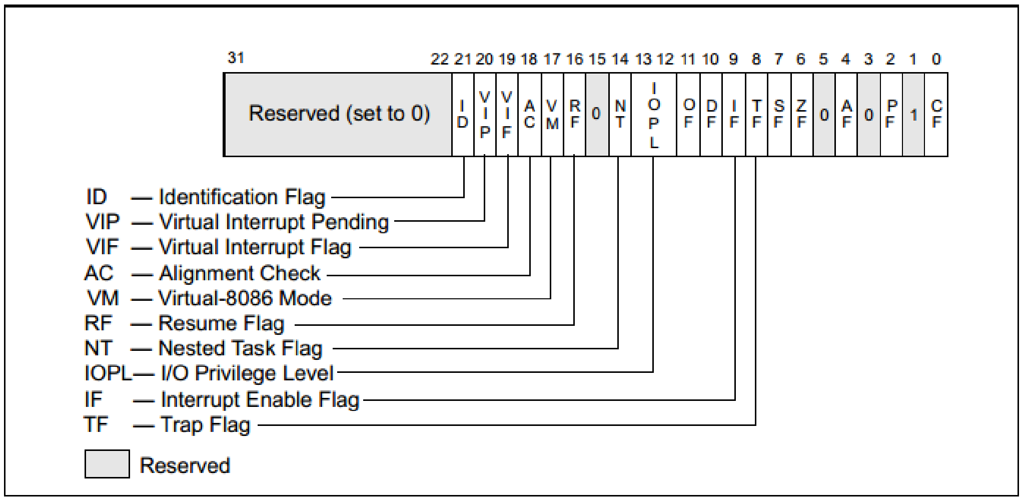


Figure 2-4. System Flags in the EFLAGS Register

Delivering User Targeted Interrupts

One thing that our model fails to address up to this point, is how to deliver an interrupt to a specific task. We have allowed handlers to run, maintaining integrity of the burst timer, but this functionality is sorely lacking. For instance, lets say a task makes a request to the disk, to load a file into ram. At some point, we have to notify that task that its request has been fulfilled and is ready for servicing. To enable this functionality, we have to establish some way for metadata surrounding an interrupt to be available: the target task of the interrupt is the bare minimum, but other information such as buffer address, size, time etc may be required as well.

We can do this by establishing a set of queues in the kernel, that contain metadata about the interrupt. These queues are written to by the interrupt source(such as disk) indicating the target task, and anything else that is required for effective servicing. In addition, we now have to augment the code at the entry point that non timer interrupts jump to, so that the dispatcher can inspect the queue for that interrupt, and attach any pending interrupt metadata into an array of 2 dimensional linked list, indexed by CR3(one for each pid). The nodes of the first list contain the interrupt vector and a task defined function that handles it. To each node is attached another list, the nodes of which contain the metadata about the received interrupts.

The next time the dispatcher selects a pcb switch to, it inspects the interrupt list, and notes any interrupts that have arrived while the task was waiting. It will then set the timer and jump as before, but the stub it jumps to is different than the first one we described. The new stub precedes the code in the existing stub by attaching call instructions to the user defined handlers. In this way, passing control back to the original program state will not occur until their handlers are serviced first, which is precisely what we are looking for.

Avoiding Synchronization Issues with the Scheduler

There is one thing that I didn't yet mention. The dispatcher and the scheduler are sharing common data, and although only an NMI can preempt the dispatcher, the scheduler could be preempted at any time, which may put our queues in jeopardy. In order to avoid this, we add a small amount of additional logic. Instead of assigning a burst time to the scheduler, we allow it to run until it fires the timer interrupt itself. This calls to the dispatcher as desired, but also ensures that it finishes its work before being preempted by a task with which it shares data structures. For other kernel tasks, this functionality may also be desirable.

User Level Thread Scheduling

Now that we have become aware of the true power of the timer interrupt, we can leverage it to also handle user level thread scheduling. Its very similar to what we did above, but far simpler. The user task simply maintains its own list of pcbs(minus CR3). It sets the timer and loads the registers in the pcb, finally jumping to the instruction pointer in that pcb. When the timer expires, the handler loads a new pcb and repeats. Thankfully, the C language enables inline assembly language to make this process possible for an application developer.