Implementing a Dispatcher in x86-64

By Michael Almandeel

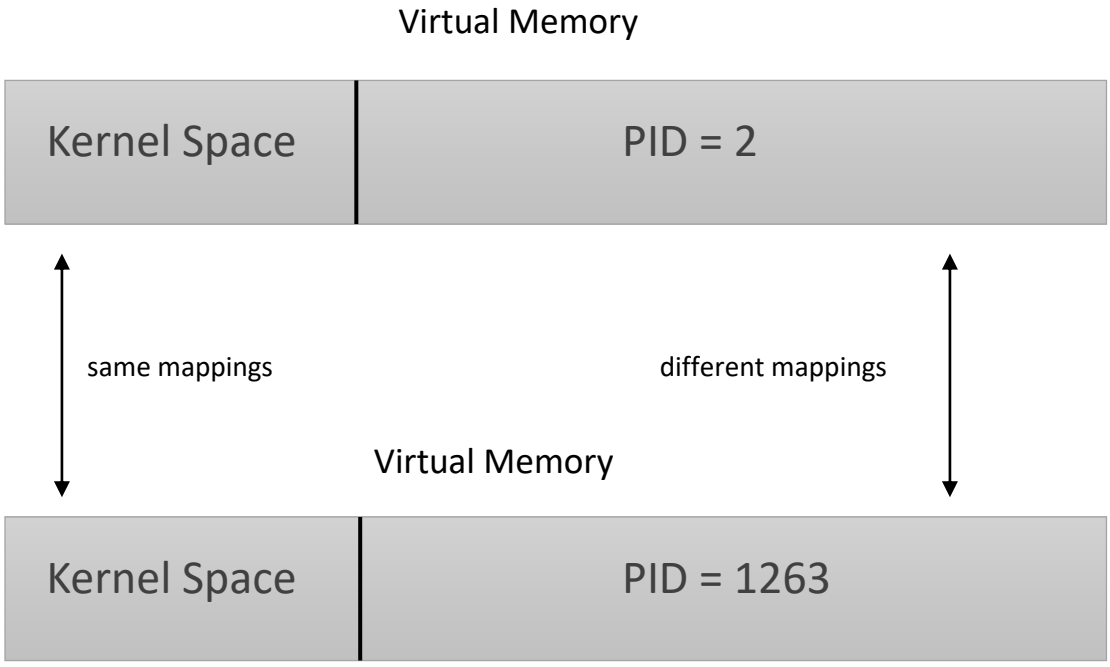
We have learned a lot about operating systems in this course, but the one thing we didn't look very closely at is the dispatcher. For most students, this is a topic shrouded in mystery. We know what it does, but not how it actually works. I have been wondering the same thing; this paper outlines my ideas 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 invoking an isr(in userspace) for such an interrupt which was defined by the target program. 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 effectively 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.

Memory Structure Requirements

In order to guarantee that kernel execution proceeds appropriately when transitioning between different page tables, there are constraints that we must place on how we map virtual to physical addresses for the kernel. The following rules must be followed:

1- At boot time, the kernel shall be loaded into the beginning of the physical address space(depending on the boot routine there may be boot data/code here so the kernel may load directly behind this data)
2- The kernel establishes a set of mappings from the beginning of the virtual address space to correspond to where its loaded in physical memory
3- The kernel mappings remain fixed until the cpu is powered off
4- The mappings we established are loaded into each processes page table when the process is created

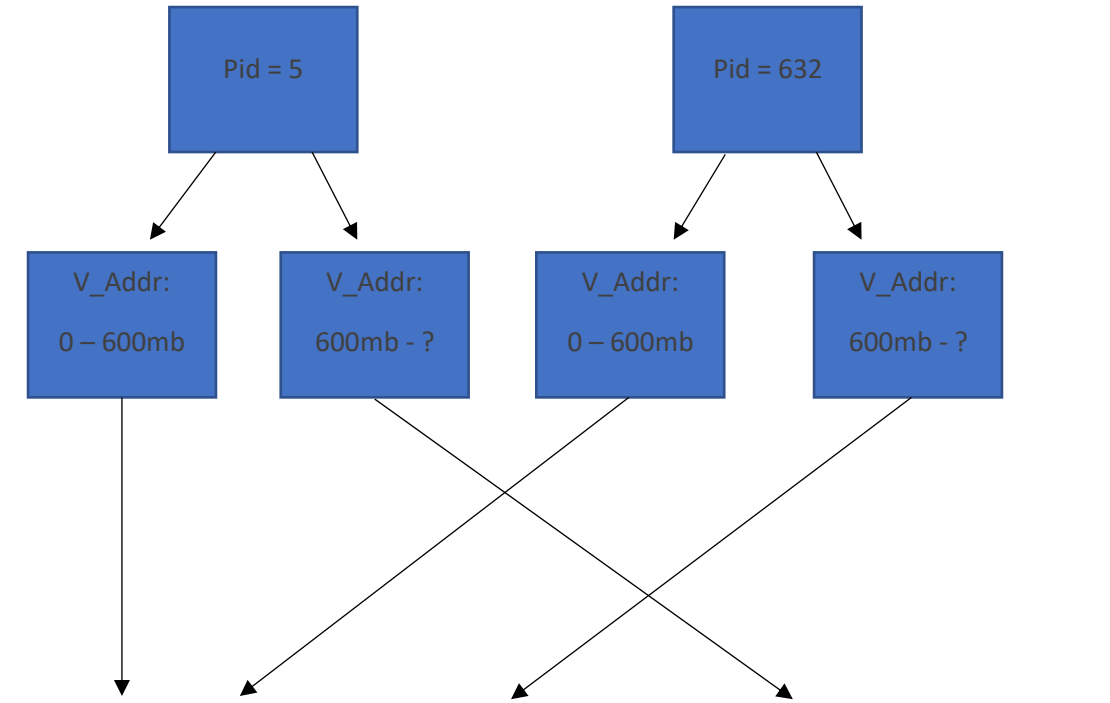


The reason this is important is we may be executing kernel code within a processes page table(when in privileged mode), and wish to switch to a new processes page table at some point during this execution. Guaranteeing the same mappings ensures that the kernel code executes the next instruction which was intended. Note that without the global mapping guarantee we may transition into a different execution path, or something else entirely.

As indicated earlier, each process gets its own page table. The entries in the page table(both sub-tables and references to the pages themselves) contain flags that indicate what privilege level can access them(the **protection key**) and what pages can be swapped to disk. We must ensure that kernel pages cannot be swapped to disk, and that their protection key is 0, indicating that they can only be accessed in **privileged mode**. These controls ensure that kernel mappings are consistent for all page tables, and that the corresponding processes can't modify or access them.

There are cases where a process gets its own user specific kernel pages, and these pages can be swapped to disk. One example may be some buffers that the kernel uses to queue i/o transfers for a user. User specific kernel pages are also extensively used in security. A **guard page** is placed in between the stack and the heap in programs, to prevent overrunning the stack by throwing an exception when the user tries to write to the guard page. The heap allocator can use the location of the guard page to determine whether or not an allocation may overlap with the stack(in which case it throws an exception instead of allocating). Another example is a **shadow stack,** which contains copies of a subset of each stack frame pushed onto the stack at call time. When returning, the shadow stack checks/overwrites the return address on the stack frame to ensure it returns where the developer intended, rather than to a malicious address.

As indicated earlier, there is a different page table for each process. The **memory management unit** selects which page table perform its lookups in based on the **page table index**, which is located in the **%cr3** register. Coincidentally, the page table index is identical to the **pid** for a process. The kernel changes the value in the %cr3 register to perform a **virtual address space switch**. These occur during dispatching(as part of the **context switch**), or during interrupts, to write process specific data to buffers to a process other than the one which is currently running.



Physical Memory

Interrupts

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.

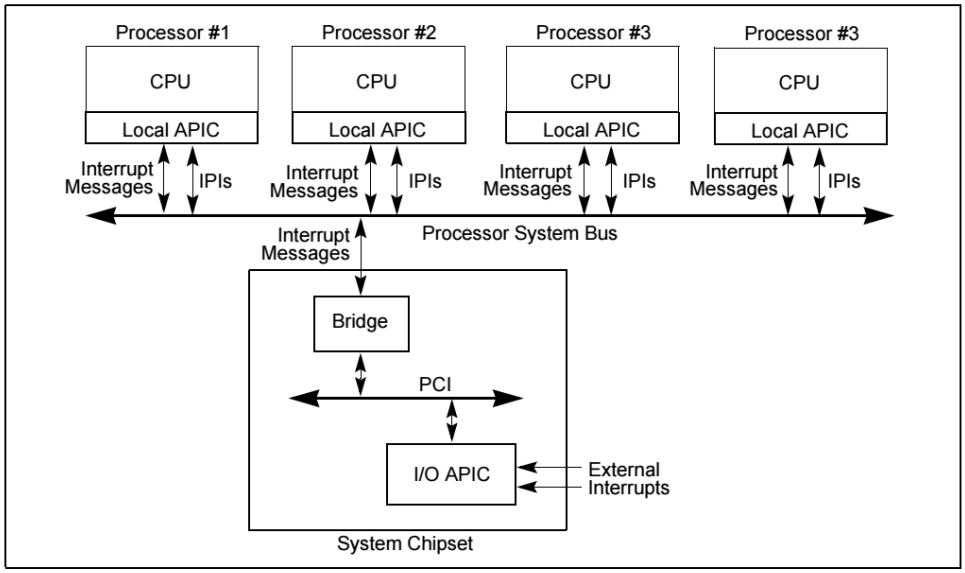


Figure 10-2. Local APICs and I/O APIC When Intel Xeon Processors Are Used in Multiple-Processor Systems

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.

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 an interrupt gate, which contains among other things the **entry point** of the **isr** to which it corresponds.

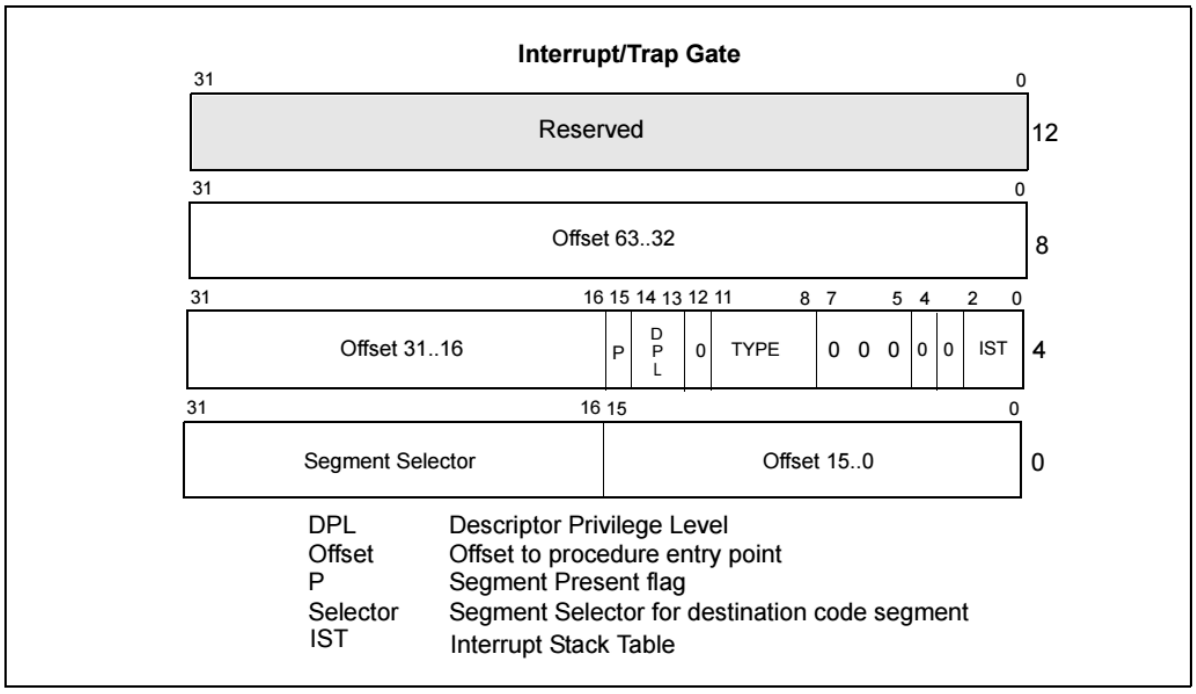


Figure 6-8. 64-Bit IDT Gate Descriptors

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 **kernel stack**. 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 **%rsp**, **%rflags**, and **%rip**(the error code in the figure represents what happens on an exception, which is beyond the scope of this paper). Note that in older series intel 64 bit chips, the stack segment and code segment are pushed as well. However, in newer series, segment registers all default to a 0 base, making them effectively obsolete, and are not pushed.

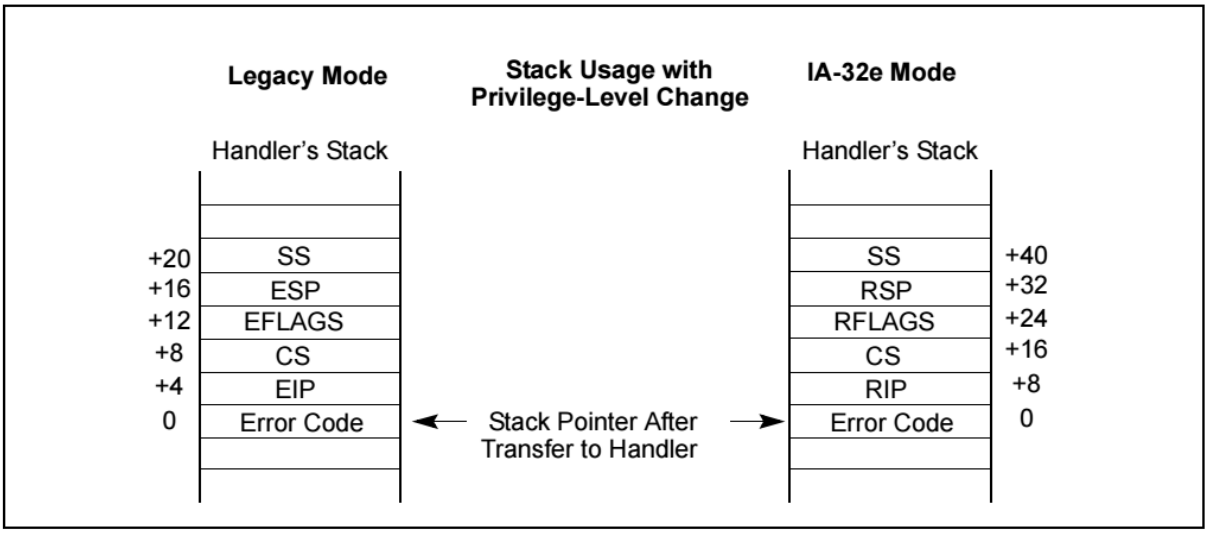


Figure 6-9. IA-32e Mode Stack Usage After Privilege Level Change

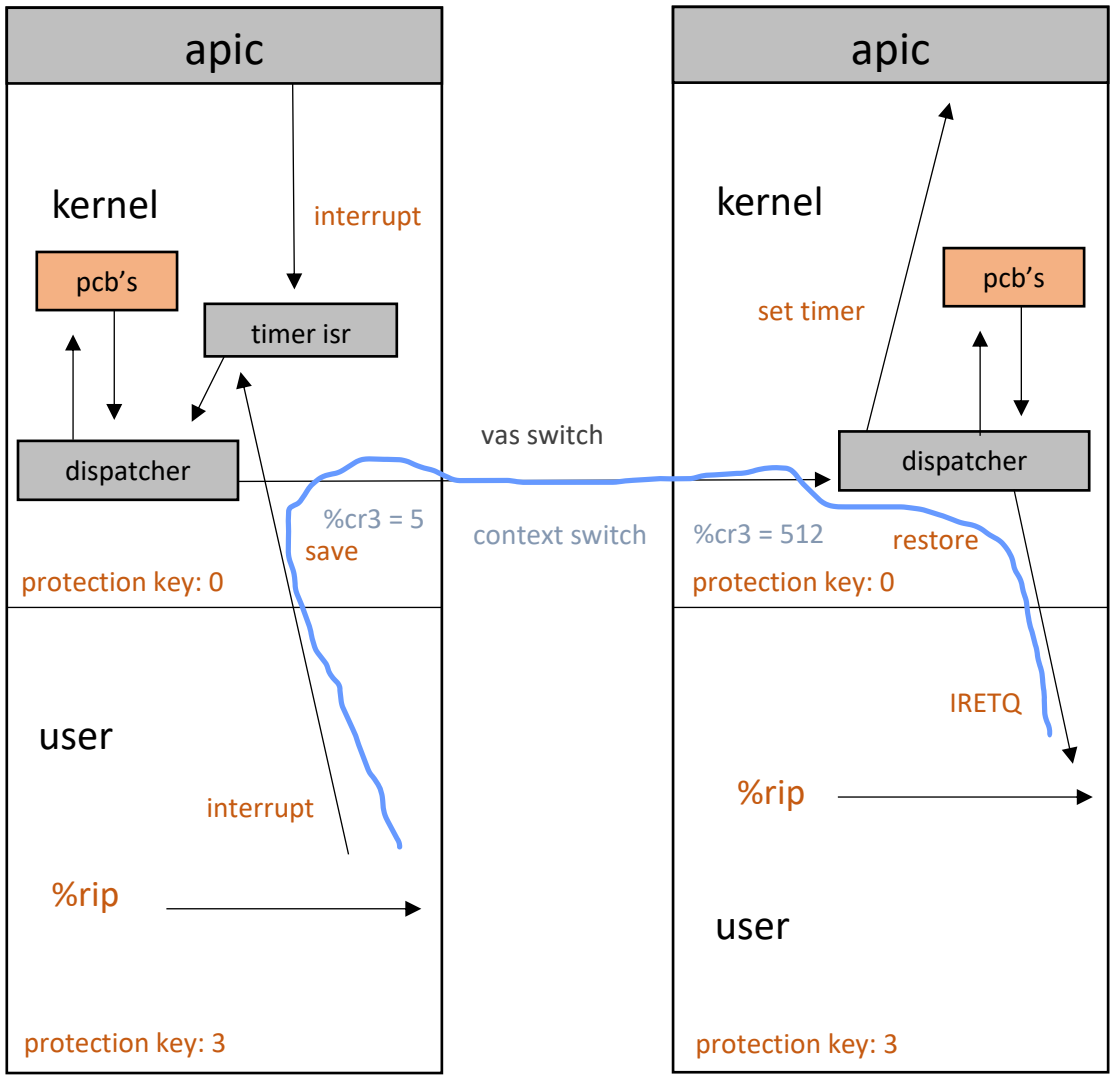
At this point, the cpu begins executing the code in the **isr**. When it is finished, it executes the **IRETQ** instruction, restoring the previous values pushed on the stack to their respective register and returning control to the user. It's important to note that when writing an isr, you should push 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. Another interesting aspect of interrupts is the concept of a **priority level**, which is used by the apic to order the delivery of interrupts to the core. Regardless of priority however, once an isr begins running, it can only be interrupted by an NMI until it IRETQ's, which sets the **interrupt flag** to true, allowing the cpu to be interrupted again.

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. The heart of the dispatcher lies in the **burst timer isr**. We set a timer to burst time, which fires an interrupt, eventually calling its associated isr(we can specify the interrupt vector for the timer on the apic LVT). This isr jumps to the dispatcher, which then performs a **context switch**. Note that you may simply make the dispatcher the timer isr itself, but the author prefers a bit more modularity.

As alluded to, the dispatcher performs a procedure known as a context switch. A **context** is a set containing each registers state for an executing program at a specific point in its execution. In order to context switch, all registers of the task we are switching from must be saved to their **pcb**, and replaced with their corresponding registers in the target pcb. At the point **%cr3** is changed to the page table index of the target task, we now reference the new page table from herein, and this transition is called a **virtual address space switch**. Note that the pcb values for %rsp, %rip and %rflags in the pcb are taken from the **kernel stack**, where they were initially pushed by the burst timer interrupt. Similarly, when restoring these registers from the new pcb, we write them to their corresponding locations in the kernel stack(instead of

their registers) and call **IRETQ** to return control to the new context(task). Immediately before passing back control, we set the timer to burst time.



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 save the timestamp , complete its execution, and jump to code in the dispatcher(the **burst time manager**), which **resets** the timer appropriately and passes control back to the interrupted task. In order to maintain integrity of the burst timer, the burst time manager must have information about the interrupted task's burst time, when it began, and what time it is now, in order to calculate the appropriate value for the new timer.

Immediately after its called, the isr pushes the timestamp. After the isr completes its execution, it jumps into the burst time manager. The burst time manager cancels the timer by setting 'initial count' to 0. It then resets the timer as such:

set_timer(burst_time-(interrupted_time-start_time).

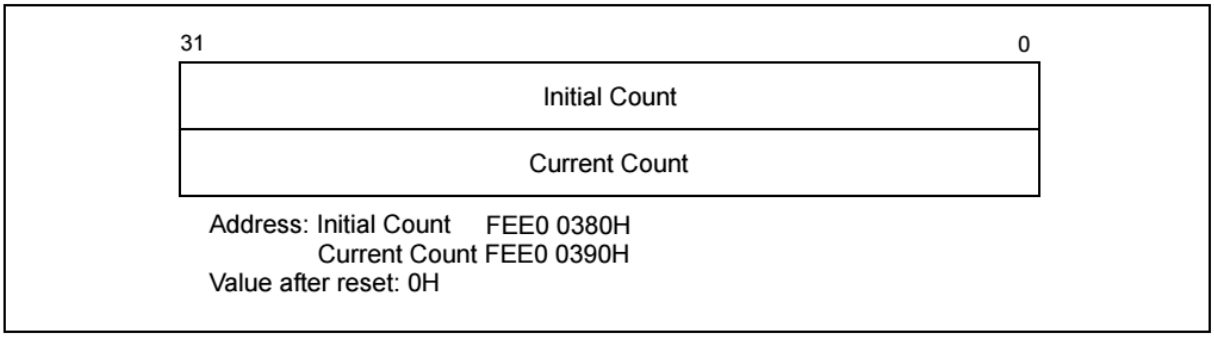
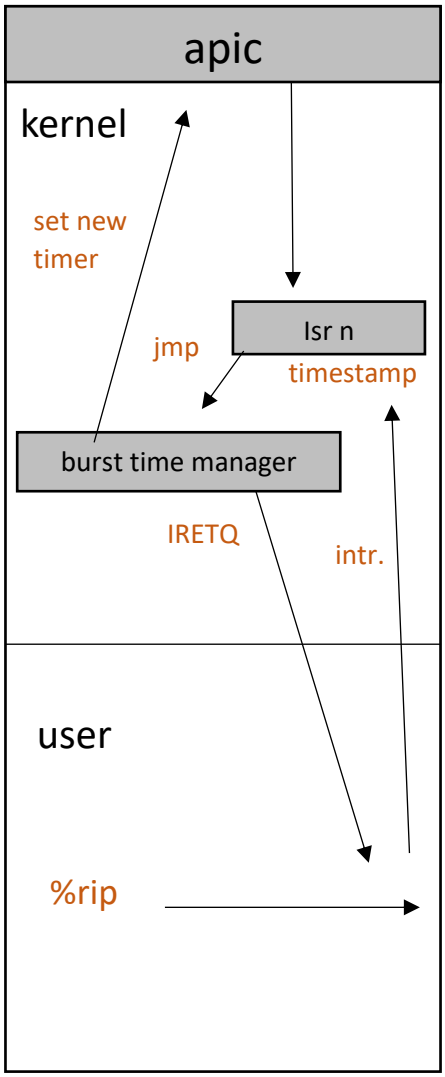


Figure 10-11. Initial Count and Current Count Registers



The burst time manager then restores the registers used during this process(which were pushed by the isr on top of the timestamp) 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 jump into the burst timer prematurely, and cause the previously interrupted isrs work to effectively be lost forever. 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 to jump into the burst time manager.

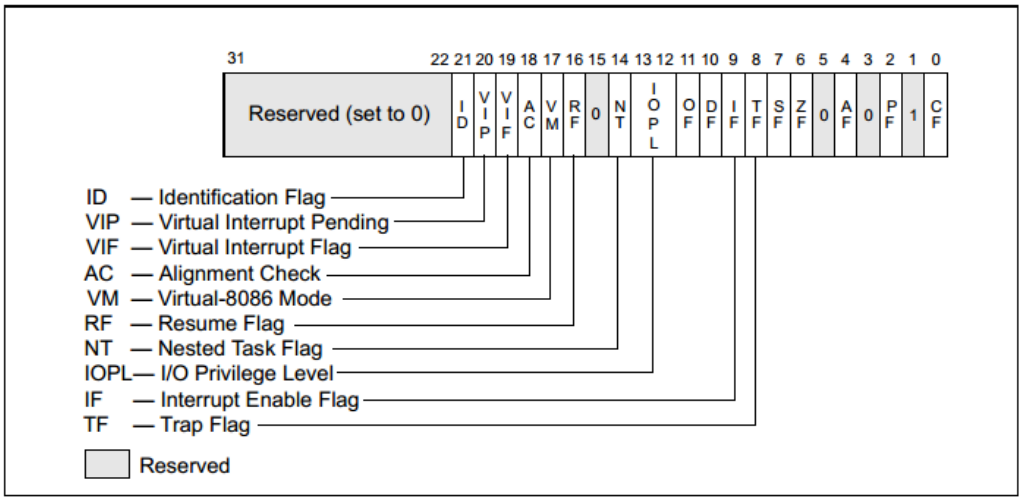


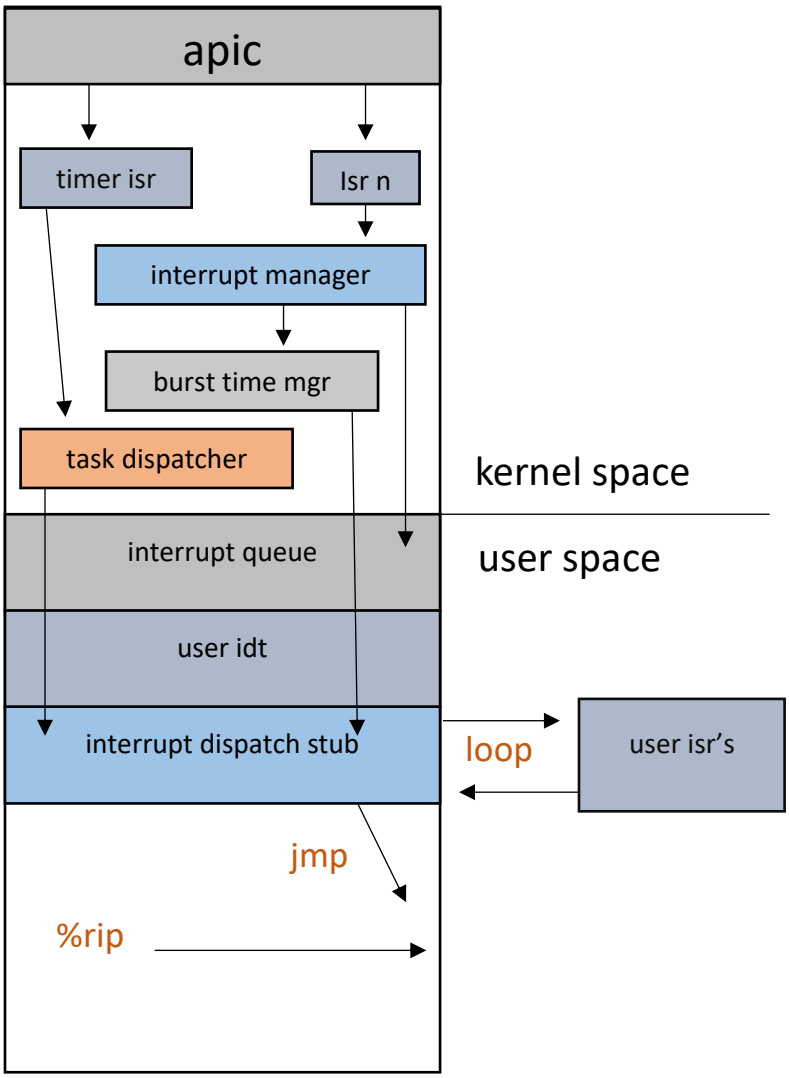
Figure 2-4. System Flags in the EFLAGS Register

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 interrupts. These queues are written to by the interrupt source(such as disk) indicating the target task, and anything else that is required for servicing. In addition, we now have to change the code at the entry point that non timer interrupts jump to replacing the entry point of the burst time manager with that of the **interrupt manger**. Additionally, the isr must set the variable **current_int** to the vector that its currently servicing. Once entered, the interrupt manager inspects the queue for the interrupt indicated by current_int, and attaches any pending interrupt metadata into the target process's interrupt queue, which may require an intermediate virtual address space switch by temporarily accessing the corresponding processes page table(using %cr3) to access the process's interrupt queue. In the case that the

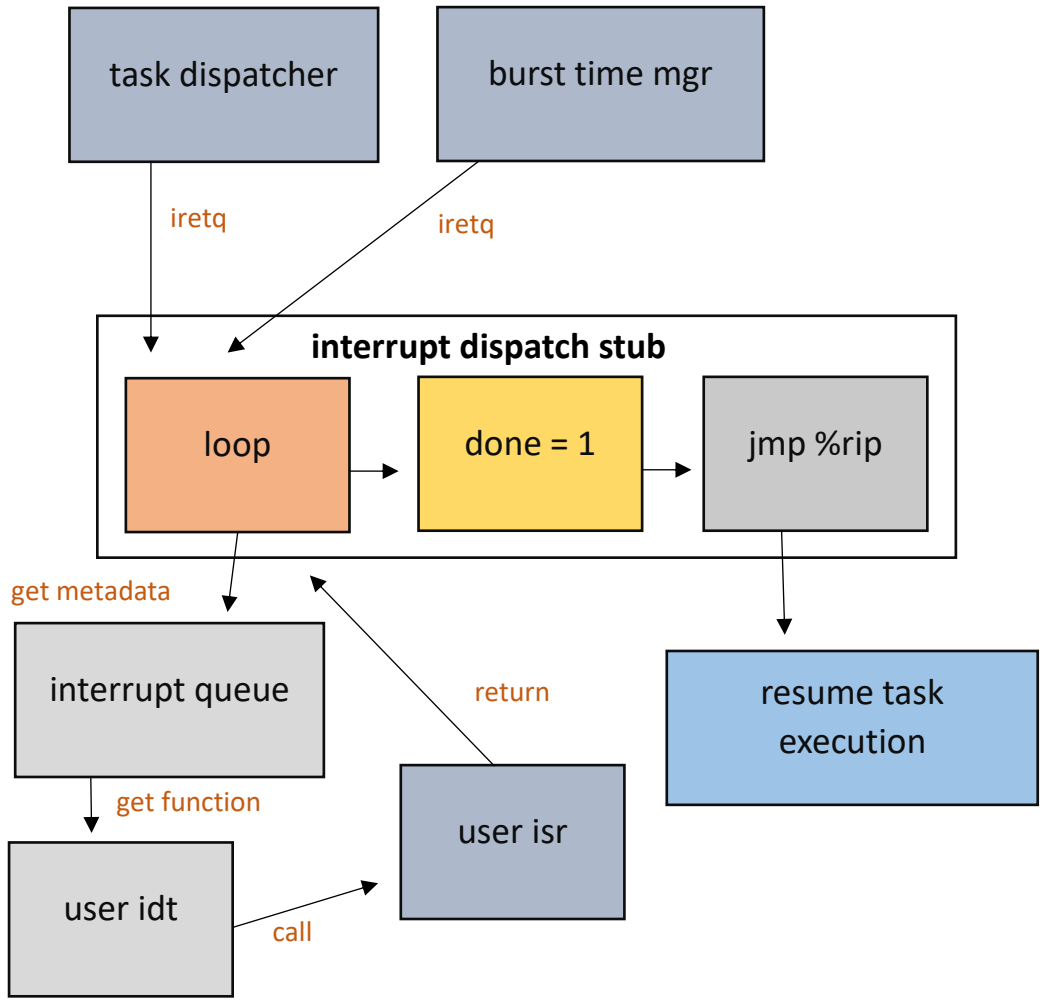
interrupt is intended for the interrupted process, it will be delivered immediately(described below).



Note that after doing its work, interrupt manager jumps to the burst time manager, so it can reset the timer and restore the interrupted programs state.

The Interrupt Dispatch Stub

Before we get describe the stubs structure, we must establish rules about restoring program state. Note that both the task dispatcher and the burst time manager restore the execution of a program within a certain context. With the addition of the interrupt dispatch stub, we need to introduce more subtlety to this process. If the task to be restored is currently executing within the stub, we must restore control as usual. If the task is executing outside of the stub, we always return control to the stub first, which after completing its work, will restore the execution for us.



We can do this by establishing a variable, ‘done’. When the task dispatcher/burst time manager pass control to the stub, they set ‘done’ to 0. When the stub completes its work, it sets ‘done’ to 1. When the task dispatcher/burst time manager goes to restore control, it checks whether or not ‘done’ == 0. If it is, they restore control as usual. If it isn’t, they set ‘done’ to 0 and pass control to the interrupt dispatch stub. Every time the task dispatcher/burst time manager iretqs to the entry point of the stub, they first write the operand of a jmp instruction to the %rip of the context it intends to restore. The jmp instruction is at the end of the stub, and facilitates restoring execution context after restoring all of the other registers.

The stub utilizes two data structures to complete its work. The interrupt queue contains a linked list , where each node contains a vector number, and a pointer to metadata. The user idt contains an array of function pointers, indexed by interrupt vector. These function pointers are the user defined isr’s.

At the stubs entry point, it pushes all the registers, except %cr3, %rsp and %rip. It then enters a loop, which reads each entry of the interrupt queue, calling its respective isr and repeating this until the queue is empty. It then restores all registers(except %rip, which is the operand of the next jmp instruction), and sets ‘done’ to 1, finally jumping to the where the task left off, restoring its context.

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.