ECE391: Computer Systems Engineering **Spring 2007**
Lecture Notes Set 2

Interrupts and Synchronization

These notes cover interrupts and topics relevant to their generation, control, and handling within the Linux kernel on an x86-based computer. The notes begin with a discussion of interrupts and their relationship to system calls and exceptions. The next few sections describe how interrupts are supported within a processor, including necessary aspects of the processor's state machine, external inputs, and architecturally visible (*i.e.*, ISA-level) data structures and registers. We recall the LC-3 implementation as a familiar example before introducing the analogous x86 features. We then consider the problem of sharing data between interrupt handlers and programs, touching briefly on how these issues are exacerbated in the presence of multiple processors. Several synchronization constructs are presented. The notes then continue with discussion of the logic necessary to connect many devices to a processor's single interrupt input, focusing on the Intel 8259A chip used by or emulated in most x86-based systems. After describing how both the interrupt controller and interrupts are abstracted within the Linux kernel, the notes conclude with an example and a few references and pointers for further study.

A Note on the Notes

As mentioned in the last set of notes, a bold font highlights definitions, and italicization emphasizes pitfalls. Linux sources are given with path names relative to the Linux source directory when first referenced. Header files are given with path names relative to the `include` subdirectory. References to Patt and Patel are to the second edition.

System Calls, Interrupts, and Exceptions

As you may recall from ECE190, system calls (also called traps), interrupts, and exceptions are all quite similar to procedure calls. **System calls** are almost identical to procedure calls. As with procedure calls, a calling convention is used: before invoking a system call, arguments are marshaled into the appropriate registers or locations in the stack; after a system call returns, any result appears in a pre-specified register. The calling convention used for system calls need not be the same as that used for procedure calls, and is typically defined by the operating system rather than the ISA. The details of this convention are generally handled by library code, however, which maps the ISA's calling convention into that of the OS' system calls. Rather than a CALL or JSR instruction, system calls are usually initiated with an INT or TRAP instruction. With many architectures, including the x86, a system call places the processor in privileged or kernel mode, and the instructions that implement the call are considered to be part of the operating system. The term system call arises from this fact.

Unexpected processor interruptions can occur due to interactions between a processor and external devices or to erroneous or unexpected behavior in the program being executed. The term **interrupt** is reserved for asynchronous interruptions generated by other devices, including disk drives, printers, network cards, video cards, keyboards, mice, and any number of other possibilities. **Exceptions** occur when a processor encounters an unexpected opcode or operand. An undefined instruction, for example, gives rise to an exception, as does an attempt to divide by zero. Exceptions usually cause the current program to terminate, although many operating systems allow the program to catch the exception and to handle it more intelligently. The table below summarizes the characteristics of the two types and compares them to system calls.

type	generated by	example	asynchronous	unexpected
interrupt	external device	packet arrived at network card	yes	yes
exception	invalid opcode or operand	divide by zero	no	yes
system call/trap	deliberate, via INT instruction	print character to console	no	no

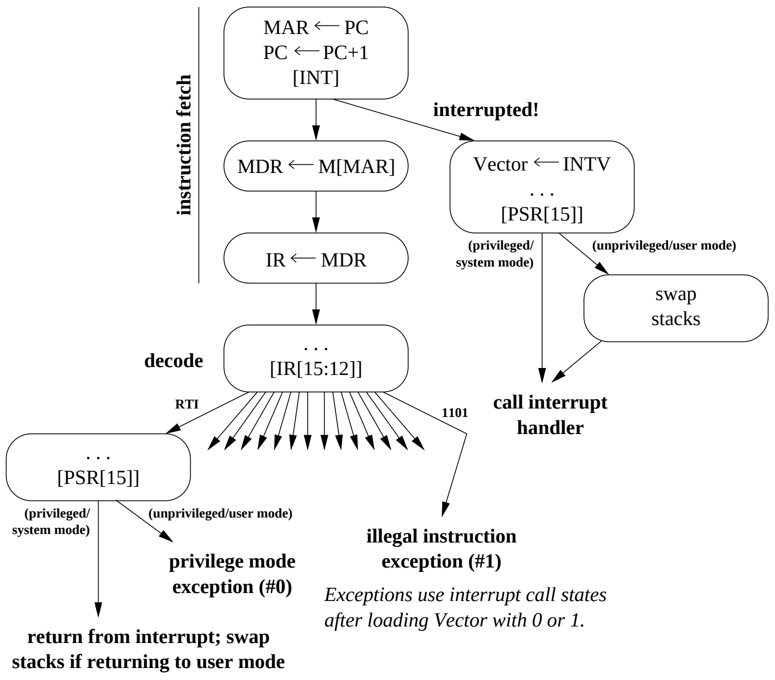
Interrupts occur asynchronously with respect to the program. Most designs only recognize interrupts between instructions, *i.e.*, the presence of interrupts is checked only after completing an instruction rather than in every cycle. In pipelined designs, however, several instructions execute simultaneously, and the decision as to which instructions occur "before" an interrupt and which occur "after" must be made by the processor. Exceptions are not asynchronous in the sense that they occur for a particular instruction, thus no decision need be made as to instruction ordering. After determining which instructions were before an interrupt, a pipelined processor discards the state of any partially executed instructions that occur “after” the interrupt and completes all instructions that occur “before.” The terminated instructions are simply restarted after the interrupt completes. Handling the decision, the termination, and the completion, however, increases the design complexity of the system.

The code associated with an interrupt, an exception, or a system call is a form of procedure called a **handler**, and is found by looking up the interrupt number, exception number, or trap number in a table of function pointers called a **vector table** (or jump table). Separate vector tables can exist for each type (interrupts, exceptions, and system calls), or can be shared. The x86 ISA, unlike the LC-3, uses a single common table.

Processor Support for Interrupts

We now discuss the interrupt support provided by the processor in more detail, first by recalling the extensions made to the basic LC-3 design, then by discussing their analogues in the x86 ISA.

Interrupts in the LC-3: As you may recall, the LC-3’s interrupt support is covered only briefly in ECE190. The portion of the LC-3 state machine shown to the right is based on Figure C.7 of Patt and Patel, but has been simplified and annotated to highlight the elements most relevant to our course.



instruction fetch
MAR ← PC
PC ← PC+1
[INT]
interrupted!
MDR ← M[MAR]
IR ← MDR
Vector ← INTV
. . .
[PSR[15]]
(privileged/ system mode)
(unprivileged/user mode)
swap stacks
call interrupt handler
decode
. . .
[IR[15:12]]
RTI
1101
. . .
[PSR[15]]
(privileged/ system mode)
(unprivileged/user mode)
privilege mode exception (#0)
illegal instruction exception (#1)
Exceptions use interrupt call states after loading Vector with 0 or 1.
return from interrupt; swap stacks if returning to user mode

The LC-3 uses a priority encoder to map up to eight device interrupt requests into a three-bit priority level, then raises an interrupt generation signal (INT) internally when an interrupt should occur. The prioritization allows the LC-3 to prevent interrupts with equal or lower priority from interrupting the execution of more important interrupts.

The first state in the instruction fetch sequence checks INT, and, if it is high (is a 1, meaning that it has been asserted by some device), the LC-3 initiates an interrupt handler call. If not, it continues with instruction fetch. Once instruction fetch has progressed to the second cycle, interrupts are ignored until the instruction has been fully processed. The LC-3, like almost all processors, does not allow interrupts to split individual instructions.

Calling an interrupt handler involves reading an eight-bit interrupt vector (INTV in the diagram) supplied as input to the processor when an interrupt is requested, and using this value as an index in the interrupt vector table, an array of interrupt handler addresses. The interrupt vector table is identical in form to the trap vector table (with which you are more familiar), and resides at memory locations 0x100 through 0x1FF. A few other operations are also necessary, such as saving the return address, processor status, and flags on the stack; possibly swapping stacks (discussed later, in the resource-sharing section); and adjusting the PC.

The interrupt handler ends with the RTI instruction, which appears in the lower left of the diagram. The processing for this instruction restores the processor status and flags at the time of the interrupt, swaps the stack if necessary, and resets the PC to point to the instruction that was about to execute when the interrupt occurred.

The LC-3 handles exceptions in the same way that it handles interrupts. Exceptions are generated when an instruction with an illegal opcode is executed, or when the RTI instruction is used outside of an interrupt handler (detected using the privilege bit in the status register). In both cases, after selecting the appropriate vector number, the state machine transitions into the interrupt handler sequence to the right.

Interrupts in the x86: By design, the LC-3 uses a simplified version of the x86 interrupt support. The basic elements are the same. The x86's INTR input indicates that some device has requested an interrupt. In most x86 implementations, these interrupts are prioritized and masked according to priority by an external interrupt controller, which allows for a more flexible prioritization scheme. In some recent implementations, interrupt controller functionality is integrated into the processor.

The x86 allows software to block all interrupts requested by the INTR pin using a processor status flag. If the **interrupt enable flag** (**IF**) is set in the flags register, interrupts are allowed to occur. If it is clear, interrupts are masked regardless of their priority. The STI and CLI instructions change the value of IF. The external interrupt controller (discussed later) also masks equal and lower priority interrupts until the processor indicates that the interrupt handler has completed, so these interrupts are masked whether or not the IF flag is clear.

Unlike the LC-3, the x86 also has a second interrupt input, NMI, that cannot be masked by setting a status flag. These **non-maskable interrupts** are used to indicate serious conditions such as parity failure in memory, critically low energy levels in batteries, *etc.*, and will not be addressed in great detail in our course, as most of the hardware and software mechanisms involved are nearly identical to those used for normal interrupts. For more information, read the file `arch/i386/kernel/nmi.c`.

The x86 uses a single vector table called the **Interrupt Descriptor Table**, or **IDT**, for interrupts, exceptions, and system calls. Although this table is a 256-entry array indexed by vector number, the elements are not simply code addresses, but also include some information about privilege level and other topics to be covered later in this course. For now, however, you can view the IDT as a simple table of code addresses pointing to interrupt, exception, and system call handlers. Each of these ends with an IRET instruction.

As with the LC-3, the exception vectors are specified as part of the ISA; Intel reserves the first 32 values for this purpose. The rest of the table is left to the operating system, although the interrupt controller hardware (discussed later) does restrict interrupt handlers to two contiguous blocks of eight values. Under Linux, the IDT is structured as shown to the right. The abbreviation IRQ stands for interrupt request.

	0x00	division error	
	⋮		
	0x02	NMI (non-maskable interrupt)	
	0x03	breakpoint (used by KGDB)	
0x00–0x1F	0x04	overflow	
	⋮		
defined	0x0B	segment not present	
by Intel	0x0C	stack segment fault	
	0x0D	general protection fault	
	0x0E	page fault	
	⋮		
	0x20	IRQ0 — timer chip	
	0x21	IRQ1 — keyboard	
0x20–0x27	0x22	IRQ2 — (cascade to slave)	
	0x23	IRQ3	
master	0x24	IRQ4 — serial port (KGDB)	
8259 PIC	0x25	IRQ5	
	0x26	IRQ6	example of possible settings
	0x27	IRQ7	
	0x28	IRQ8 — real time clock	
	0x29	IRQ9	
0x28–0x2F	0x2A	IRQ10	
	0x2B	IRQ11 — eth0 (network)	
slave	0x2C	IRQ12 — PS/2 mouse	
8259 PIC	0x2D	IRQ13	
	0x2E	IRQ14 — ide0 (hard drive)	
	0x2F	IRQ15	
0x30–0x7F	⋮	APIC vectors available to device drivers	
0x80	0x80	system call vector (INT 0x80)	
0x81–0xEE	⋮	more APIC vectors available to device drivers	
0xEF	0xEF	local APIC timer	
0xF0–0xFF	⋮	symmetric multiprocessor (SMP) communication vectors	

Input and Output

An interrupt usually occurs when a device needs attention. For example, interrupts occur when a key is pressed, when the mouse is moved, and when a block of data is available from the disk. The handler code that executes on behalf of a device interacts with the device using through the processor's input/output, or I/O interface. The interrupts themselves are in fact a special form of I/O in which only the signal requesting attention is conveyed to the processor, but let's leave the details of that signaling for now and recall how a processor sends and receives data from devices.

Communication of data occurs through instructions similar to loads and stores. A processor is designed with an **I/O port space** similar to a memory address space. Devices are connected to the processor through a bus (or several buses), and each device is associated with some port or set of ports. Reads and writes to device registers are then transmitted by the processor on the bus using the port numbers as addresses. When a device recognizes a bus transaction targeting one of its ports, it must respond appropriately. For example, a device may deliver data onto the bus in response to a read and record the data written in a register in response to a write.

The question remains as to exactly how I/O ports are accessed in software. One option is to create special instructions, such as the IN and OUT instructions of the x86 architecture. Port addresses can then be specified in the same way that memory addresses are specified, but use a distinct address space. Just as two sets of special-purpose registers can be separated by the instructions of an ISA, such an **independent I/O** system separates I/O ports from memory addresses by using distinct instructions for each class of operation.

Alternatively, device registers can be accessed using the same load and store instructions as are used to access memory. This approach, known as **memory-mapped I/O**, requires no new instructions for I/O, but demands that a region of the memory address space be set aside for I/O. The memory words with those addresses, if they exist, can not be accessed during normal processor operations.

As mentioned in the last set of notes, most x86-based computers use both models, with some devices fully mapped into the port space, others fully mapped into memory, and still others with registers in both spaces.

Shared Data and Resources

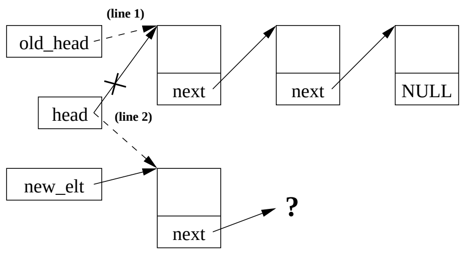
The asynchronous and unexpected nature of interrupts with respect to normal program execution can lead to problems if data and resources shared by the two pieces of code are not handled carefully. This section discusses a wide range of possible problems, ranging from the obvious to the subtle. Solutions to most issues are also given, while subsequent sections describe techniques for correctly managing some of the more complex issues.

Like any other type of procedure, interrupt handlers must preserve the contents of registers and must avoid overwriting memory locations used by the interrupted program. Unlike procedures or system calls, however, interrupts must preserve all registers used. The program does not expect a register to be overwritten between instructions just because the procedure calling convention specifies it as being caller-saved; no call was made! Registers are typically preserved by writing them onto a stack and restoring their values before returning from the interrupt. If an interrupt handler needs additional memory for storage, memory locations can be dedicated to the handler's private use.

Some less obvious shared resources include flag and status registers as well as any data structures (in memory) used to convey information between the interrupt handler and the program. Flag and status registers can be saved to the stack along with all other registers, either by the handler or by the processor itself. As the LC-3 has no instruction capable of loading or storing the Process Status Register (PSR), the processor itself writes the PSR to the stack on an interrupt and restores it when executing an RTI instruction. In the x86 ISA, the EFLAGS register can be pushed and popped, but is also usually saved and restored automatically.

Sharing data structures between a program and an interrupt handler can be complicated, even if the interrupt handler is not changing the data structure. Consider a handler that walks down a linked list. The handler interrupts the code at the top of the next page, which places `new_elt` at the head of the list, between lines 2 and 3. The figure to the right of the code illustrates the state of the list before the code executes (solid lines) and after the first two lines have executed (dashed lines are new). The question mark represents the uninitialized `next` field of `new_elt`. The interrupt handler starts from `head` and follows the dashed line to oblivion.

```
/* line 1 */ old_head = head;
/* line 2 */ head = new_elt;
             /* INTERRUPT OCCURS HERE! */
/* line 3 */ new_elt->next = old_head;
```



The example above illustrates a problem at the level of C source statements, but interrupts occur between instructions, which may or may not be part of the same statement in the source language. You may have noticed that this example can be rewritten to make the change to the list with a single instruction (a store to `head`). In general, all data structure operations can be written in such a manner, but not always without making the operations slow. One must also work around the compiler, which is allowed to reorder and even to intermingle instructions corresponding to statements in high-level languages in the interest of improving performance. Clearly, we need some mechanism to keep interrupts from happening in certain parts of the program.

The most subtle interactions between programs and interrupts involve multi-cycle operations and security. For the first case, consider an LC-3 program that prints a string to the display, and a keyboard interrupt handler that echoes keystrokes back to the display. The program has read the display status and found that it is ready to accept another character, but before writing that character to the display's data register, it is interrupted by a keystroke. The interrupt handler reads in the key and echoes it back to the display, then returns control to the program. The display takes some time to deal with the echoed keystroke, and thus ignores the character sent by the program. From the program's point of view, the display status register magically changed from ready to not-ready between instructions.

A similar type of error can be inserted into your program through compiler optimizations. In order to make the common case fast, compilers ignore the possibility of interrupts affecting memory under the control of the program. Typically, such assumptions do not lead to problems. However, if you deliberately share data in this manner, you may need to tell the compiler. As a simple example, you could create a variable and initialize it to zero, then wait for an interrupt to change the variable to one. The naive implementation is as follows:

```
int sync = 0;
while (!sync);
```

The problem lies in optimization. The compiler analyzes the loop and decides that nothing can change the value of `sync`, and thus that the memory read can be moved out of the loop. Once the load is out of the loop, the value is obviously constant, so the test is always the same, and only need be done once. The result? A load, a single test, and an infinite loop in the form of a single branch instruction. Very fast! Unfortunately, even after the interrupt handler updates `sync`, the optimized program continues to loop.

To prevent such problems, add the qualifier `volatile` to the variable declaration (*i.e.*, `volatile int sync=0;`). This qualifier tells the compiler that the value stored in memory may change at any time, and that it may not assume that two loads of the variable return the same value.

The last issue to be considered is security. One role of the operating system is to provide isolation and protection between programs run by various users. Keeping users from crashing the machine is one implication of this role, and preventing information leaks from the operating system to programs is another. However, using a program's stack to store information during execution of interrupt handlers can be hazardous. If the stack has insufficient space, or if the stack pointer has been set to point to important data rather than a stack, blindly dumping registers onto the stack may crash the operating system. Similarly, when an interrupt handler completes, the data from the handler as well as any calls made by the handler are still on the stack, and are now visible to the program. To avoid these issues, many ISA's use a separate stack register for the operating system (*i.e.*., when operating in privileged mode, protected mode, system mode, kernel mode, *etc.*). This stack-swapping is illustrated by the LC-3 state diagram earlier in these notes; when the processor switches from unprivileged to privileged mode or vice-versa, the stacks are swapped.

Critical Sections

The last section raised the question of how to prevent interrupts from interfering with data structure operations, and in particular how to ensure that interrupts can be prevented from accessing data structures in invalid states (recall the broken linked list).

Conceptually, we want to be able to mark critical sections of the program. A **critical section** is a block of code that executes a set of operations that should be executed without stopping, *i.e.*, without interruption. As a real-world example, if I am changing the battery in my pacemaker, you should not tap me on the shoulder after I pull out the old battery and ask me to make you a pot of coffee. Or, rather, if you do so, I should ignore you until the new battery is safely in place. Pulling out the old battery and putting in the new one is a critical section. Similarly, the code in the linked list example forms a critical section, as shown below. Technically, only lines 2 and 3 need be included for the interrupt handler described in the example, but marking the entire operation as a critical section makes it compatible with interrupt handlers that change the list as well as those that only read it.

```
             /* start of critical section */
/* line 1 */ old_head = head;
/* line 2 */ head = new_elt;
/* line 3 */ new_elt->next = old_head;
             /* end of critical section */
```

Comments are not sufficient, of course; they must be replaced with operations that prevent conflicting operations from interfering and tell the compiler to avoid moving instructions out of the critical section when optimizing the code.

Once the code has been marked appropriately, the critical section occurs **atomically** with respect to the interrupt. The term atomic here implies indivisibility[1]: the entire critical section has either been executed when an interrupt occurs, or none of it has been executed. The interrupt never finds it half done.

What happens if the interrupt must find `new_elt` in the list? For example, consider another interrupt handler that removes a specified element from the list and frees the element's structure. If the element is not found, the kernel leaks dynamically allocated memory until it runs out of space (and probably crashes). If the critical section above is not guaranteed to execute before an interrupt handler tries to remove `new_elt`, the code suffers from a **race condition**. If the critical section wins the race, nothing goes wrong. If the interrupt handler wins the race, garbage piles up in the kernel. Such race conditions are bugs! They can also be quite subtle and difficult to locate, since they may only occur once in a while, and may disappear when you add a debugging print statement.

When such a race condition exists, the only solution is to rewrite the code or extend the critical section to guarantee that the interrupt handler cannot try to remove `new_elt` before the critical section inserts it into the list. In general, removing race conditions may require substantial design changes.

Let's return to the problem of creating critical section boundaries. On computers with only a single processor, the approach is simple enough: use interrupt masking at the boundaries of critical sections.[2] Also, when writing in a high-level language, tell the compiler that all of memory is volatile at these boundaries. Unlike use of the `volatile` keyword, the compiler can then optimize variable accesses inside and outside of critical sections, but cannot optimize accesses across the boundaries.

Masking interrupts does have some drawbacks, of course. One advantage of using interrupts is to reduce the average time that a device must wait for service. Delays can reduce throughput for data transfers to disks and the network, and in extreme cases can cause loss of information. For example, a serial line only buffers one character, allowing the processor about 100 microseconds to read the data before the next character overwrites it. If a program executes for long periods with interrupts masked, problems may arise. Critical sections should thus be as short as possible, and should not include operations that can be performed outside of the critical section without significant drawbacks.

A second drawback to masking arises from the fact that non-maskable interrupts are not masked. The NMI line is typically only used for unusual events such as memory or bus failure, but any data touched by the NMI handler must be protected through means other than mere masking.

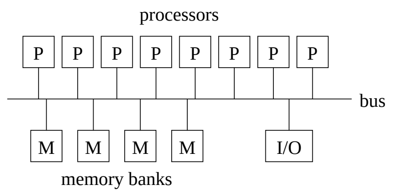
[1]Please don't ask any quarky questions about the atomicity property.

[2]We assume that programs do not share data.

Multiprocessors and Locks

On a uniprocessor, simultaneous execution of several pieces of code can occur only logically, with all but one piece of code suspended while the remaining piece executes. In contrast, multiple processors allow multiple pieces of code to execute at the same time. For example, a program and an interrupt handler can execute concurrently on a multiprocessor. As a result, interrupt masking does not suffice to protect critical sections on a multiprocessor. The interrupt mask flag operates on a per-processor basis, thus only the processor executing the code is prevented from taking an interrupt. Interprocessor communication—for example, to request that all processors mask interrupts—is far too slow and cumbersome to use at every critical section boundary.

Most commercial multiprocessors are of the SMP, or symmetric multiprocessor, variety. The symmetry lies in the relationship between processors and memory banks; while memory data can still be cached close to a processor for performance reasons, the access time from any processor to any uncached memory location is identical. An idealized diagram of an SMP is shown to the right.



Multiprocessor synchronization is a vast topic; in these notes, we focus only on the practical constructs used in Linux: spin locks, semaphores, and reader/writer locks. The discussion here is limited to lock operation semantics; a more advanced discussion is provided at the end of these notes, but in general is beyond the scope of the course.

The term **spin lock** refers to the fact that a program waiting for a lock "spins" idly in a small loop while waiting rather than going off to do other useful work or allowing other programs to use the processor. A spin lock is initially said to be available or unlocked. A program attempts to acquire a lock by atomically changing the lock from the available to the held or locked state. If successful, the program is said to hold or to own the lock. Only one program can own a lock at any time. The basic spin lock call keeps trying until it succeeds, but other lock functions allow a program to try once and to perform other work before trying again. When finished with a critical section, a program releases any locks that it owns by changing the locks' state back to available. *Only the owner of a lock should unlock it.*

The `spinlock_t` structure represents a spin lock in Linux, and a fairly rich set of functions is provided for manipulating them, as shown in the table below. The implementations of these functions depends on the kernel configuration. When configured for a uniprocessor, the macros in `linux/spinlock.h` expand to empty operations and automatic successes. When the kernel is configured for use with an SMP, the actual lock implementations defined in `asm/spinlock.h` are used (by inclusion from `linux/spinlock.h`).

Spin locks must be initialized to the available state before they are used. A statically allocated spin lock can be initialized statically, as shown in the example on the next page. Dynamically allocated spin locks require a call to `spin_lock_init` after allocation; *be sure that no race conditions allow a dynamically allocated spin lock to be used before it is initialized!*

initialization

`void spin_lock_init (spinlock_t* lock);`	Initialize a dynamically-allocated spin lock.

basic lock and unlock functions

`void spin_lock (spinlock_t* lock);`	Obtain a spin lock; call returns only when lock is obtained.
`void spin_unlock (spinlock_t* lock);`	Release a spin lock; must only be called on locks owned by caller.

miscellaneous testing functions

`int spin_is_locked (spinlock_t* lock);`	Check if a spin lock is held. Returns 1 if held, 0 if available. Note that the lock may be claimed again before the caller can do anything!
`int spin_trylock (spinlock_t* lock);`	Make one attempt to obtain a lock. Returns 1 on success, 0 on failure.
`void spin_unlock_wait (spinlock_t* lock);`	Wait until a spin lock is available. Note that the lock may be claimed again before the caller can do anything!

lock and unlock with interrupt masking

`void spin_lock_irqsave (spinlock_t* lock, unsigned long& flags);`	Save processor status in `flags`, mask interrupts, and obtain a spin lock; call returns only when lock is obtained.
`void spin_unlock_irqrestore (spinlock_t* lock, unsigned long flags);`	Release a spin lock, then set processor status to `flags`; must only be called on locks owned by caller.
`void spin_lock_irq (spinlock_t* lock);`	Mask interrupts and obtain a spin lock; call returns only when lock is obtained. Note that this version *does not preserve* the current value of the interrupt masking flag.
`void spin_unlock_irq (spinlock_t* lock);`	Release a spin lock, then enable interrupts; must only be called on locks owned by caller. Note that this version *does not restore* the previous value of the interrupt masking flag.

Normal locking and unlocking, as well as the miscellaneous lock testing functions, are mostly useful on SMPs. You should use the spin lock calls with interrupt masking—most often in the irqsave and irqrestore forms—to protect your critical sections. The `spin_lock_irqsave` prototype in the table uses C++-style reference notation for the `flags` argument to indicate that the flags are passed using only the variable name, but are changed by the function. As you know, such behavior is only possible in C if the “function” is implemented as a preprocessor macro, as is the case here. Let’s rewrite our linked list example using the spin lock **application programming interface** (**API**):

```
             static spinlock_t the_lock = SPIN_LOCK_UNLOCKED;
             unsigned long flags;

             /* start of critical section */
/* line 0 */ spin_lock_irqsave (&the_lock, flags);
/* line 1 */ old_head = head;
/* line 2 */ head = new_elt;
/* line 3 */ new_elt->next = old_head;
/* line 4 */ spin_unlock_irqrestore (&the_lock, flags);
             /* end of critical section */
```

The two lock functions called to demarcate the critical section are actually C preprocessor macros defined in `linux/spinlock.h`. The interrupt masking functions called by these macros, `local_irq_save` and `local_irq_restore`, are themselves macros defined in `asm/system.h`. By unrolling those macros, we can rewrite the code as shown on the next page.

The program enters the critical section with line 0, which consists of saving the flags and acquiring a lock. The program saves the current EFLAGS register into a local variable by pushing the flags and then popping it back into some other register chosen by the compiler, after which the compiler writes the register that it chose into the variable `flags`. The “memory” argument tells `gcc` that all of memory should be considered volatile across this piece of assembly code, and that no loads or stores should be moved across it, as is necessary at a critical section boundary. After saving the old value of the flags for later restoration, the program clears IF, blocking all interrupts. Only at that point does it attempt to acquire the lock. For a detailed explanation of the syntax and properties of the `asm` directive in GCC, read the “Extended Asm” node of the info page on GCC.

The program leaves the critical section with line 4, which consists of releasing the lock and restoring the original value of EFLAGS. Interrupts may or may not have been masked before entering the critical section—for example, by this function’s caller; with the functions used here, the programmer does not need to know. If the `spin_lock_irq` and `spin_unlock_irq` functions are used instead, interrupts are enabled at the end of the critical section, and no caller to the function containing the critical section can expect to keep interrupts masked across the call. Restoring EFLAGS again involves a PUSH and a POP, this time moving the value of `flags` into EFLAGS. The “cc” argument tells GCC that the condition codes register—the EFLAGS register on the x86—is modified by the assembly code.

The ordering of operations is important. If a program acquires a lock and is then interrupted by a handler that tries to acquire the same lock, the processor **deadlocks**: the handler must wait until the program releases the lock, but the program must wait until the handler finishes, so neither can make progress, and the machine freezes up.

```
                static spinlock_t the_lock = SPIN_LOCK_UNLOCKED;
                unsigned long flags;

                /* start of critical section */
/* line 0 */ asm volatile ("      # local_irq_save macro implementation
                    pushfl          # save EFLAGS to stack
                    popl %0         # pop EFLAGS into output 0
                    cli             # mask interrupts
                " :  "=g" (flags)    /* output 0 is a general-purpose register */
                                    /*   which should then be stored in flags */
                  :                 /* no inputs                              */
                  :  "memory"       /* see text                               */
                );
                spin_lock (&the_lock);
/* line 1 */ old_head = head;
/* line 2 */ head = new_elt;
/* line 3 */ new_elt->next = old_head;
/* line 4 */ spin_unlock (&the_lock);
                asm volatile ("      # local_irq_restore macro implementation
                    pushl %0        # save input 0 to stack
                    popfl           # pop input 0 into EFLAGS
                " :                 /* no outputs                             */
                  :  "g" (flags)    /* input 0 is a general-purpose register  */
                                    /*   which should hold the value in flags */
                  :  "memory", "cc" /* see text                               */
                );
                /* end of critical section */
```

With the ordering shown in the code, interrupts are masked before the lock is acquired and only enabled after the lock is released. The processor on which the program executes can never be interrupted while it holds the lock, thus the deadlock scenario just discussed cannot occur. However, another processor in an SMP can execute the interrupt handler, which may block waiting for the lock to be released by the program. In this case, the low-priority program running on one processor blocks execution of the high-priority interrupt handler running on another processor. This type of behavior makes it even more important that programmers make critical sections as short as possible.

As a final note, the calls to `spin_lock` and `spin_unlock` implement real locks on an SMP, and are empty macros (NOPs) on a uniprocessor. Thus, if you use the style above, your code will work on any platform.[3]

Semaphores

The term semaphore refers to visual systems of signaling, typically with flags. Railroads use semaphores to ensure that two trains do not enter a single stretch of track while heading in opposite directions. Traffic lights are sometimes called semaphores (always in some languages).

In computer systems, a **semaphore** generalizes the concept of a lock to allow some fixed number of programs to enter some set of critical sections simultaneously. For example, if we have three keyboards attached to a computer, we can use a semaphore to arbitrate access to the keyboards. The semaphore's value is initially set to three, meaning that all keyboards are free. When a program wants a keyboard, it executes a **down** operation to claim one, atomically reducing the semaphore's value to two. Down is also called wait, test, or P for the Dutch word proberen (to test). Once all three keyboards have been claimed, the semaphore's value is zero, and programs attempting to down the semaphore block until a keyboard becomes available. When a program is ready to relinquish a keyboard that it owns, it executes an **up** operation on the semaphore, which increments the semaphore's value by one. Up is also called signal, post, or V for the Dutch word verhogen (to increment).[4]

[3]SMP code tested on a uniprocessor "works" by definition, but at least you have some hope of it working on an SMP, too.

[4]Dutch is relevant because E. Dijkstra, a well-known pioneer in algorithms and computer systems, was Dutch. See `http://www.cs.utexas.edu/users/UTCS/notices/dijkstra/ewdobit.html`.

In Linux, semaphores differ from spin locks in that a program waiting on a semaphore allows other programs to execute while it waits (recall that a spin lock spins on the processor, repeatedly trying the lock). You should use semaphores rather than spin locks with any code that manages data shared between user programs executing in the kernel (through a system call). On the other hand, *semaphores should not be used in code that shares data with interrupt handlers, nor should code ever wait on a semaphore while a spin lock is held.*

Allowing other programs to run when waiting on a semaphore means that semaphores can be used to protect longer critical sections. Consider, for example, a device that accepts commands and responds only once a command has been fully executed. Each command/response pair forms a critical section, as intervening commands may be ignored or may cancel the first command. However, command execution may be slow, and maintaining control of the processor while a slow device operates is an unattractive solution. Instead, a programmer can use a semaphore to ensure that the device is not given a new command while it is busy processing, and can yield the processor to other programs even while it holds the semaphore. Allowing this logical concurrency makes synchronization of shared resources with semaphores important even in uniprocessors, and the calls do not turn into empty macros, as do the spin lock calls.

Linux supports a semaphore abstraction optimized for uncontended access. A semaphore is represented by a `struct semaphore`. The full versions of the API functions are in `arch/i386/kernel/semaphore.c`. Fast versions that handle success on the first try and fall back on the full versions when not successful are written as preprocessor macros in the header file, `asm/semaphore.h`. Linux' semaphore API appears below.

initialization

`void sema_init (struct semaphore* sem, int val);`	Initialize a dynamically allocated semaphore to a value.
`void init_MUTEX (struct semaphore* sem);`	Initialize a dynamically allocated semaphore to the value one.
`void init_MUTEX_LOCKED (struct semaphore* sem);`	Initialize a dynamically allocated semaphore to the value zero.

down and up

`void down (struct semaphore* sem);`	Wait on a semaphore; call returns only after success.
`void up (struct semaphore* sem);`	Signal a semaphore; must only be called by programs that have previously waited on the semaphore.

miscellaneous functions

`int down_interruptible (struct semaphore* sem);`	Wait on a semaphore, but allow other programs to execute while waiting; call returns 0 on success or -EINTR on interruption.
`int down_trylock (struct semaphore* sem);`	Make one attempt to wait on a semaphore. Returns 0 on success, 1 on failure (*the opposite of the spin_trylock function!*).

When only one program can enter a critical section at a time, the presence of programs in the critical section is mutually exclusive, and the term **mutex**, an abbreviation of this phrase, is used to describe synchronization of this type. The dynamic initialization routines allow a semaphore to be initialized with a specific initial value (`sema_init`) or as a mutex (a semaphore allowing one program at a time). Semaphores can also be statically allocated and initialized using the macros below, which expand into variable declarations *without a* `static` *qualifier in front of them*, and thus must appear either outside of any function or at the start of a function or compound statement (usually with `static` in front). Each dynamic initialization function has a corresponding macro for static/runtime declaration and initialization:

```
/* Allocate statically and initialize to val.  */
static __DECLARE_SEMAPHORE_GENERIC (name, val);
/* Allocate on stack and initialize to one.  */
DECLARE_MUTEX (name);
/* Allocate on stack and initialize to zero.  */
DECLARE_MUTEX_LOCKED (name);
```

The behavior of the `down` and `up` functions have already been discussed. The `down_interruptible` form is useful when control should be returned to a user-level process on receiving a signal (the user-level form of an interrupt, to be discussed later in the course; as an example, pressing CTRL-C generates a signal).

Reader/Writer Spin Locks

Certain types of synchronization benefit from separating pieces of code that share a particular datum into those pieces that write the datum and those that only read it. Readers in general do not interfere with one another, and thus any number of readers can be allowed to enter their critical sections simultaneously. However, writers may interfere with one another, and writers may also interfere with readers, thus a writer should only be allowed to access the datum when no other programs—readers or writers—are accessing it. These properties are provided by a **reader/writer lock**. Linux supports both reader/writer spin locks and reader/writer semaphores. The former are discussed in this section, and the latter in the next section.

The Linux API for reader/writer spin locks is shown below. As with normal spin locks, these locks should be used whenever data are shared with an interrupt handler, and should most often be used with interrupt masking. A reader/writer spin lock can be statically allocated and initialized as shown here:

```
rwlock_t an_rw_lock = RW_LOCK_UNLOCKED;
```

Dynamically allocated locks must be initialized with `rwlock_init`. There is no non-blocking version of the `read_lock` function, *i.e.*, `read_trylock` is not defined.

initialization

`void rwlock_init (rwlock_t* rw);`	Initialize a dynamically-allocated reader/writer lock.

basic lock and unlock functions

`void read_lock (rwlock_t* rw);`	Lock for reading; blocks until lock is obtained.
`void write_lock (rwlock_t* rw);`	Lock for writing; blocks until lock is obtained.
`void read_unlock (rwlock_t* rw);`	Release a lock previously locked for reading.
`void write_unlock (rwlock_t* rw);`	Release a lock previously locked for writing.

miscellaneous testing functions

`int write_trylock (rwlock_t* rw);`	Make one attempt to lock for writing. Returns 1 on success, 0 on failure.

lock and unlock with interrupt masking

`void read_lock_irqsave (rwlock_t* rw, unsigned long& flags);`	Save processor status in `flags`, mask interrupts, and lock for reading; blocks until lock is obtained.
`void write_lock_irqsave (rwlock_t* rw, unsigned long& flags);`	Save processor status in `flags`, mask interrupts, and lock for writing; blocks until lock is obtained.
`void read_unlock_irqrestore (rwlock_t* rw, unsigned long flags);`	Release a lock previously locked for reading, then set processor status to `flags`.
`void write_unlock_irqrestore (rwlock_t* rw, unsigned long flags);`	Release a lock previously locked for writing, then set processor status to `flags`.
`void read_lock_irq (rwlock_t* rw);`	Mask interrupts and lock for reading; blocks until lock is obtained. Note that this version *does not preserve* the current value of the interrupt masking flag.
`void write_lock_irq (rwlock_t* rw);`	Mask interrupts and lock for writing; blocks until lock is obtained. Note that this version *does not preserve* the current value of the interrupt masking flag.
`void read_unlock_irq (rwlock_t* rw);`	Release a lock previously locked for reading, then enable interrupts. Note that this version *does not restore* the previous value of the interrupt masking flag.
`void write_unlock_irq (rwlock_t* rw);`	Release a lock previously locked for writing, then enable interrupts. Note that this version *does not restore* the previous value of the interrupt masking flag.

The Linux implementation of reader/writer spin locks is fast, but does not prevent writer **starvation**. In particular, readers can enter a critical section even if a writer is waiting. As a result, readers may continuously enter and leave a critical section without the number of active readers ever reaching zero. As writers can only enter their critical sections when the number of readers (and writers) is zero, they may wait forever, a behavior known as starvation.

Reader/Writer Semaphores

The reader/writer semaphore abstraction in Linux has the exclusion properties of reader/writer locks and the scheduling properties of semaphores. Specifically, any number of readers can enter critical sections protected by a reader/writer semaphore simultaneously, but only one writer can enter a critical section at any time, and only when no other readers or writers are in critical sections protected by the same reader/writer semaphore. As with semaphores, a program attempting to acquire a reader/writer semaphore may yield the processor to another program.

Unlike semaphores, reader/writer semaphores are not parametrized by the number of programs allowed to enter critical sections. And unlike Linux reader/writer spin locks, reader/writer semaphores do not admit starvation. A waiting writer blocks any new readers from entering, for example, thus ensuring that the writer gets a turn once the current readers have finished their critical sections.

A `struct rw_semaphore` represents a reader/writer semaphore in Linux. The API for reader/writer semaphores appears below. They can be declared and initialized as shown here:

```
DECLARE_RWSEM (name);
```

or can be allocated dynamically and initialized with `init_rwsem`.

initialization

`void init_rwsem (struct rw_semaphore* sem);`	Initialize a dynamically allocated semaphore.

down and up

`void down_read (struct rw_semaphore* sem);`	Wait for reading; blocks until successful.
`void down_write (struct rw_semaphore* sem);`	Wait for writing; blocks until successful.
`void up_read (struct rw_semaphore* sem);`	Signal a semaphore previously waited on for reading.
`void up_write (struct rw_semaphore* sem);`	Signal a semaphore previously waited on for writing.

Selecting a Synchronization Mechanism

As mentioned earlier in these notes, synchronization is a vast topic, and choosing between the various forms of synchronization available in the Linux kernel based on their properties may seem quite difficult at first. This problem is exacerbated by the fact that the synchronization constructs in Linux have properties not traditionally implied by the names chosen for them. In this section, we present two approaches for selecting synchronization mechanisms.

Both approaches simplify the set of possibilities by using semaphores only as mutexes (allow only one program at a time in critical sections). The first approach further simplifies the decision process by always using the interrupt save and restore form of spin locks. With these restrictions, one can view the four synchronization constructs as a combination of two choices between two sets of properties, as shown in the table below.

The choice for the vertical axis depends on whether any interrupt handler must enter a critical section protected by the mechanism to be chosen. Kernel code is otherwise only executed by programs making system calls, or sometimes by programs running in the kernel, which can be handled similarly. If an operation on a data structure must be atomic with respect to other operations on the data structure, but the data are never accessed by an interrupt handler, semaphores should be used, as shown in the bottom row of the table. The choice for the horizontal axis is between mutual exclusion and reader/writer properties.

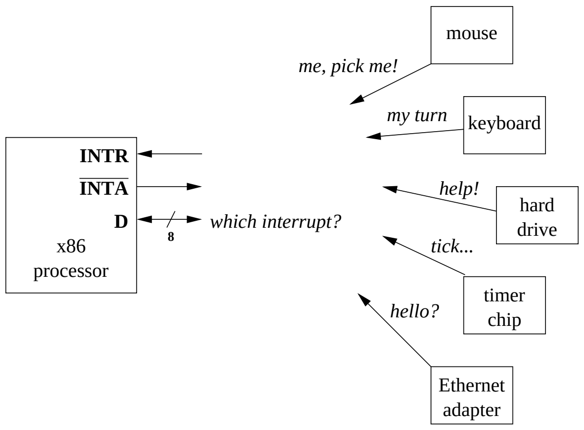
	mutual exclusion	reader/writer	
data shared by interrupt handlers	`spin_lock_irqsave` `spin_unlock_irqrestore`	`read_lock_irqsave` `read_unlock_irqrestore`	`write_lock_irqsave` `write_unlock_irqrestore`
data shared only by system calls	`down` `up`	`down_read` `up_read`	`down_write` `up_write`

Alternatively, one can use the more complex decision tree shown below to obtain slightly faster synchronization. Only the mutual exclusion option is shown here. For reader/writer properties, use reader/writer semaphores in place of semaphores, and use reader/writer spin locks in place of spin locks with the same interrupt masking properties. Finally, when more than one lock must be acquired, only the first lock needs to mask interrupts.

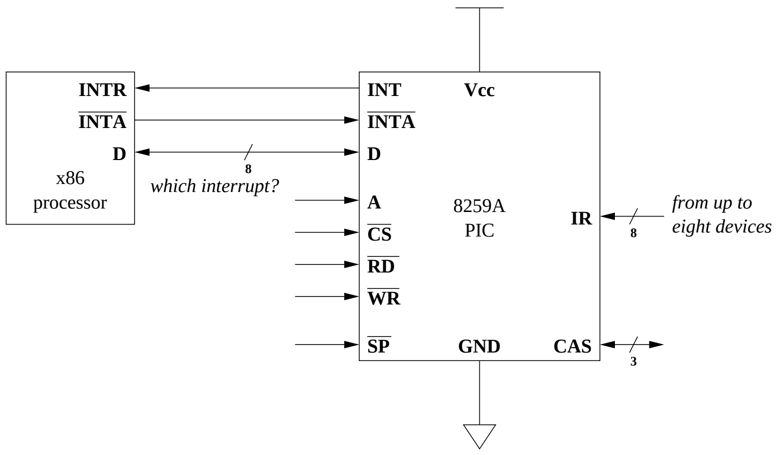
type of code entering critical section	critical section shares data with	for mutual exclusion, use
system calls only	other system calls	`up` `down`
	interrupt handlers	`spin_lock_irq` `spin_unlock_irq`
both system calls and interrupt handlers	both system calls and interrupt handlers	`spin_lock_irqsave` `spin_unlock_irqrestore`
interrupt handlers only	system calls	`spin_lock` `spin_unlock`
	higher priority interrupt handlers	`spin_lock_irqsave` `spin_unlock_irqrestore`

Interrupt Control

At this point, you should feel fairly comfortable with the interrupt support provided by a typical processor. However, connecting a large number of devices to a processor is not completely straightforward. A diagram of an x86 processor with five devices appears to the right. Each device generates a single output signal requesting attention, while the processor has a single interrupt input pin, INTR, an interrupt acknowledgement output, $\overline{\text{INTA}}$ (the bar means that the signal is active low rather than high), and a data bus over which the interrupt vector must be delivered. Device register inputs and outputs are not shown.



One might imagine simply joining the device interrupt lines with an OR gate and feeding it into the processor's INTR input. However, something must be done to choose amongst the devices when more than one needs attention. Clearly, to avoid shorts, only one can write an interrupt vector to the bus. The interrupt acknowledge signal ($\overline{\text{INTA}}$) can be used to pick, but in that case can only be delivered to one device at a time, and thus doesn't resolve the problem. Furthermore, some priority scheme for the devices can be useful, allowing a high priority interrupt to preempt (interrupt) the execution of a low priority interrupt handler.



The solution to this problem requires an **interrupt controller**, an additional piece of hardware to manage the interrupt signals and priorities. The x86 has traditionally used Intel's 8259A **Programmable Interrupt Controller**, or **PIC**, chip, for this purpose. The diagram to the right shows all 28 pins, which we describe piece by piece in the following text. First, however, we review the implementation of the LC-3's interrupt controller.

Although Patt and Patel describes a part of the interrupt controller, and the microarchitecture requires a three-bit priority signal rather than a single interrupt line, most of the controller implementation is external to the LC-3. Using an external implementation allows the processor to work with different prioritization schemes, different numbers of devices, *etc.*, and only the most recent x86 implementations have integrated the interrupt controller onto the processor. The simplest external implementation is to use a priority encoder to select from up to eight devices with fixed prioritization. The output of the priority encoder is fed into the LC-3's interrupt priority inputs, and can also be used to select the interrupt vector number or to gate the devices' writing a vector.

The 8259A is both more complicated and more flexible than the basic design just described. It operates asynchronously, with data bus transactions driven by control signals from the processor. The most common operation is reporting an interrupt to the processor, and is handled as follows. The PIC uses internal state to track which of the eight interrupts are currently being serviced by the processor. When one of the IR inputs goes high, the PIC decides whether or not it should report the new interrupt immediately based on the prioritization scheme and the set of interrupts currently in service. Lower-numbered IR lines have higher priority, and an interrupt of priority equal or lower to any interrupt already in service is masked. To report an interrupt, the PIC raises the INT output and waits for the processor. The processor (or some glue logic) strobes the $\overline{\text{INTA}}$ input to request that the PIC write the interrupt vector to the data bus D. Once the vector has been written, the PIC marks the reported interrupt as being in service and returns to waiting. At some point, the interrupt handler tells the PIC that the interrupt has been serviced by writing certain bits to the address A and data D inputs with an OUT instruction. On receiving this **end-of-interrupt** (**EOI**) signal, the PIC removes the interrupt from its set of in-service interrupts. *If an interrupt handler fails to send an EOI, the PIC continues to mask all interrupts of equal and lower priority indefinitely!* If the interrupt handler has not interacted with the device that raised the interrupt before sending the EOI, the interrupt is likely to be generated again immediately, thus it is important to perform these actions in the correct order. As we discuss later, the proper ordering is embedded into the software infrastructure in the Linux kernel, making it substantially harder for a programmer to screw up.

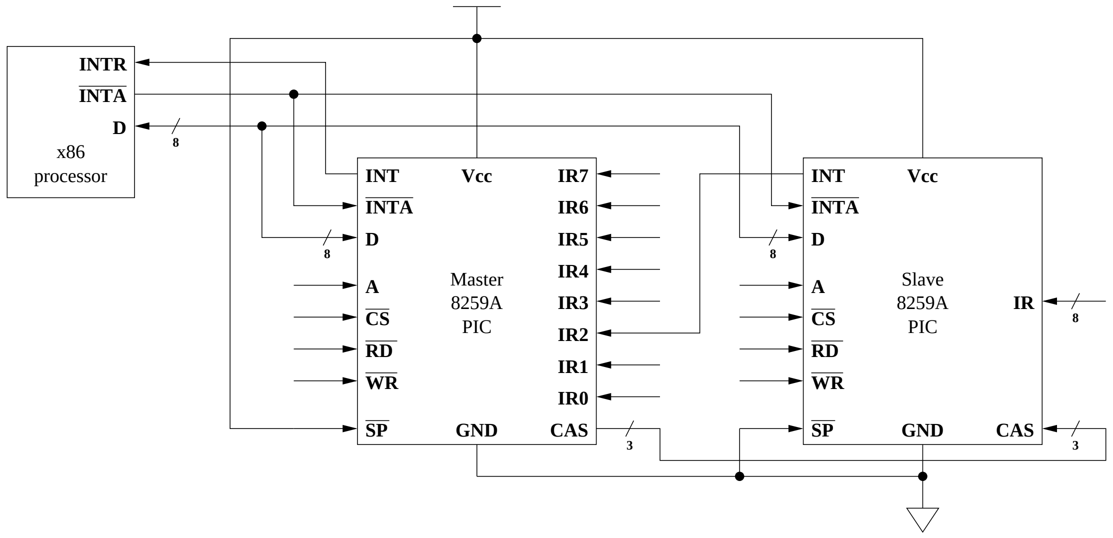
The simple overview just provided leaves a number of inputs and outputs in the diagram unused. The next block to be considered are the address A, chip select $\overline{\text{CS}}$, read $\overline{\text{RD}}$, and write $\overline{\text{WR}}$ signals. The read and write signals are named from the processor's point of view and tell the PIC that the CPU has written data to the data bus D or expects to read data from D. However, the PIC is not the only device on the data bus! An 8259A is mapped into two adjacent locations in the processor's port space, and must only respond when the processor communicates with one of those two ports. The chip select input must be active (low) only when the processor specifies one of these two ports on its address bus. If you assume that the processor uses a 16-bit address bus and wants to map the 8259A at ports 0x20 and 0x21, you should be able to generate the PIC's $\overline{\text{CS}}$ and A inputs using a handful of gates.

The 8259A uses two ports because the full functionality extends well beyond that described so far, and, while we shall not make use of much of it in the class, you should recognize that it is a programmable chip. The options supported by the 8259A include support for more than one ISA (the other ISA uses a slightly different handshaking protocol on the bus), options for various priority schemes such as rotating priorities for fairness, masking certain interrupts out indefinitely (used by Linux and most operating systems to prevent spurious interrupts), and setting the five high bits of the interrupt vector number; the three low bits are used to identify which IR input is being reported. You will probably be asked to implement a simpler, synchronous version as part of a homework. If you are interested in the asynchronous implementation, you should take ECE462.

One problem with the 8259A as presented so far is that eight interrupt lines may not be enough. So what should be done? One can certainly hook two, three, or four 8259A's to a processor using different port addresses, but doing so only reintroduces the original problem of how to merge the INT signals from the PICs into a single INTR input for the processor. One could design a new PIC with 16 inputs, or 64 inputs, or 256 inputs, but that may be a bit wasteful if many applications use only a few interrupt lines. The solution? Allow multiple 8259As to operate as a hierarchy (a cascade) and let the system designer pick.[5]

Most x86-based computers use two 8259A's connected as shown in the diagram on the next page, supporting seven IR lines on the master PIC and eight more on the slave. Interrupts generated by devices hooked to the slave PIC are delivered to IR_2 on the master PIC, and only delivered to the processor when no other higher-priority interrupts (IR_0 and IR_1 on the master PIC) are in service.

[5]Sorry.



If you think carefully about the protocol for interactions between the processor and the PIC, you should realize that the functionality discussed so far is not sufficient to support the cascaded design shown above. In particular, which PIC should write an interrupt vector to the data bus when the processor strobes $\overline{\text{INTA}}$? The slave may have raised an interrupt, but it doesn't know whether the master is reporting its interrupt or one with higher priority, so it cannot decide which chip should provide the vector. Similarly, while the master knows which chip should provide the vector, it knows neither the high bits of vector numbers used by the slave nor which of the slave's interrupt lines is being reported, and thus cannot report on behalf of the slave. Instead, it must tell the slave to write a number to the data bus. The CAS bus supports this communication: effectively, the master transmits the identification number for one of up to eight possible slaves on the bus (8259A's can be cascaded to allow up to 64 interrupt inputs). The slave program input $\overline{\text{SP}}$ is used to differentiate the master from the slave PICs and to configure the CAS bus as either input or output.

The PIC in Hardware and Software

Now that you are familiar with the basic purpose of the 8259A and some of its operations, we are ready to discuss the configuration used with most x86-based processors and the software and protocols used to drive the PICs in more detail. This section also discusses the abstractions and functions used to represent interrupt controllers within the Linux kernel.

Two 8259A PICs are used in cascade, as shown in the figure in the previous section. The master PIC is mapped at ports 0x20 and 0x21, and the slave PIC at ports 0xA0 and 0xA1. The INT output on the slave feeds into IR_2 on the master. The file `arch/i386/kernel/i8259.c` holds the Linux source code for configuring and interacting with the 8259A. The first function of interest is the initialization routine:

```
void init_8259A (int auto_eoi);
```

The single argument allows for use of the 8259A's optional automatic generation of EOI signals (as opposed to requiring the interrupt handler to send them to the PIC), but is not used in Linux.

Only one processor should initialize the PICs, and it should not be interrupted while changing the PICs' internal state. The entire initialization function is thus one critical section. After masking interrupts on the processor and acquiring a lock, the code masks all interrupts (on both PICs), executes the initialization sequence, restores the mask settings, releases the lock, and restores the IF flag. All interrupts are initially masked during boot, but in theory one could reinitialize the PICs by calling this function, in which case any active interrupts must be allowed.

Masking and unmasking of interrupts on an 8259A outside of the interrupt sequence requires only a single write to the second port (0x21 or 0xA1), and the byte written to this port specifies which interrupts should be masked.

The initialization sequence requires that four initialization control words (ICWs) be sent to the 8259A. You may want to look through the 8259A specification to understand the detailed meaning of the control words used in Linux. The first word, ICW1, is delivered to the first PIC port—either 0x20 or 0xA0—and tells the PIC that it is being initialized, that it should use edge-triggered input signals, that it is operating in cascade mode (*i.e.*, using more than one 8259A), and that four control words will be sent in all. The remaining ICWs are written to the second port. The high bits of the interrupt vector numbers are provided in ICW2: the master 8259A is mapped to interrupt vectors 0x20 through 0x27, and the slave 8259A is mapped to interrupt vectors 0x28 through 0x2F. The specific IR pin used in the master/slave relationship is specified by ICW3. Finally, ICW4 specifies the 8086 protocol, normal EOI signalling, and a couple of other (unused) options that we have ignored here.

Linux abstracts interrupt controllers using a **jump table** structure, thus allowing other code to perform generic operations on the relevant interrupt controller without knowing the details of the controller itself. Although the machines that you use in this class have only the two 8259As described here, many modern processors have embedded "advanced" PICs, or APICs, and most SMPs require I/O APICs for synchronization of interrupt delivery to a single processor and to support interprocessor communication. The protocol used to control an 8259A has essentially nothing in common with the protocols used for these APICs other than at the level of abstract operations listed in the jump table. The jump table, called a `hw_irq_controller`, is defined in `linux/irq.h` as shown below:

```
struct hw_interrupt_type {
    const char* typename;
    unsigned int (*startup) (unsigned int irq);
    void (*shutdown) (unsigned int irq);
    void (*enable) (unsigned int irq);
    void (*disable) (unsigned int irq);
    void (*ack) (unsigned int irq);
    void (*end) (unsigned int irq);
    void (*set_affinity) (unsigned int irq, unsigned long mask);
};
typedef struct hw_interrupt_type hw_irq_controller;
```

The jump table for the 8259, `i8259A_irq_type`, is defined in the `i8259.c` file mentioned earlier, and is filled with the names of the appropriate functions, which are also defined in the file. The `typename` field is a human-readable name used when reading the contents of the `/proc/interrupts` file, and is set to "XT-PIC" for the 8259A.

The other fields in the jump table are function pointers to controller-specific code. The `startup` function is called when the first request is made to attach an interrupt handler to a particular interrupt (more than one can be associated with a given interrupt, as discussed later in these notes). Both PICs are initially configured to mask all interrupts; the `startup` function for the 8259A tells the appropriate PIC to allow the interrupt line to generate interrupts, *i.e.*, it unmasks the interrupt on the PIC.[6] The `shutdown` function is called when the last handler is removed from an interrupt, and, in the case of the 8259A's function, tells the appropriate PIC to mask the interrupt.

The `disable` and `enable` functions allow nested disabling and re-enabling of active interrupts. The generic code only calls these controller-specific functions on the first call to disable and the last call to enable. The 8259A functions are identical to those used for `startup` and `shutdown`: they unmask and mask the specified interrupt on the appropriate PIC, respectively.

The `ack` and `end` functions are used to wrap the interrupt handler associated with a given interrupt. When an interrupt occurs, the controller-specific `ack` function is called to acknowledge receipt of the interrupt. The interrupt handler is then executed. Finally, the controller-specific `end` function is called to end the interrupt. The `ack` function for the 8259A masks the interrupt on the PIC, then sends the EOI signal. Although this ordering differs from the one described earlier, the mask bit on the PIC prevents further interrupts from occurring even though the device has yet to be serviced. *However, if your handler disables and re-enables the interrupt before servicing the device, a new interrupt will be generated, and will make your interrupts slower.* The generic interrupt handling code in Linux keeps such interactions from creating infinite loops by squashing the second interrupt and only leaving a "pending" marker behind, which causes the first interrupt to re-execute the handler after it finishes the first time. The 8259A's `end` function unmasks the interrupt on the appropriate PIC.

Finally, the `set_affinity` function is used to specify which CPUs within an SMP are allowed to execute a given interrupt. The most common use of this function is to restrict an interrupt to execute only on one processor, which simplifies data sharing at the expense of performance.

[6]The return value from `startup` is always 0 for the 8259A and is only used by the auto-probe code, which is outside of the scope of this course; see the section on advanced topics at the end of these notes for pointers.

Common Interrupt Abstractions

We are almost ready to explore the infrastructure provided by Linux to abstract and manage interrupts within the kernel. Before doing so, however, we discuss two common abstractions used in many systems to extend the utility of interrupts as supported by most processors: interrupt chaining, and soft(ware) interrupts.

Interrupt chaining: We have thus far assumed that each interrupt vector is associated with a single handler, and such is certainly the case as far as the hardware is concerned: an x86 transfers control to a handler specified by a single entry in the IDT, and control returns when a IRET instruction is executed. However, several software systems may wish to act when an interrupt occurs, and providing the handler is the only method through which they can take action. Early DOS systems supported terminate-and-stay-resident (TSR) programs, which often installed interrupt handlers to respond to keystrokes or mouse motion (the handlers stayed resident), then returned control to DOS (the program terminated). These programs **chained** their interrupt handlers to those already present in the interrupt vector table, usually by rewriting a JMP instruction at the end of their handler to jump to the old handler. As a result, the resulting chain (linked list) of handlers was somewhat brittle: none of the programs had any way to walk over the chain, so removing a single program's handler cleanly was effectively impossible.

Similarly, although only a single interrupt is generated when an interrupt controller's input line goes high, it is possible to connect more than one device to that input, in which case any of the devices so connected may have been the source of the interrupt. Hooking multiple devices to an interrupt line typically also requires that the software allow chaining of interrupt handlers, and furthermore that the the devices associated with the chain can be queried for their interrupt status. Clearly, only the device that generated the interrupt should receive service; others should be ignored. When an interrupt occurs, control is passed to the handler for the first device, which accesses device registers to determine whether or not that device generated an interrupt. If it did, the appropriate service is provided. If not, or after the service is complete, control is passed to the next handler in the chain, which handles interrupts from the second device, and so forth until the last handler in the chain completes. At this point, registers and processor state are restored and control is returned to the point at which the interrupt occurred. Such device interrogation is often slow, thus chaining of this type occurs fairly rarely in practice.

Soft interrupts: The purpose and importance of interrupts generated by hardware is probably clear to you. They allow relatively slow devices to get attention from a processor in a timely manner without requiring the processor to poll the devices periodically. An interrupt handler takes control of the processor as soon as it needs to run, thus interrupts can be thought of as having higher priority than most programs (except for critical sections).

These two features are also attractive for systems based purely on software. For example, I may have one program that controls a database, and a second that manages a network connection with a remote user. A single database query may require several packets of data to be sent over the network, thus not every hardware interrupt generated by the network card corresponds to a new command from the remote user. However, once a complete command has been received from the user, as recognized by the second program, allowing this program to interrupt the database control program is a useful abstraction. Unfortunately, the network program is not a device, and has no access to the INTR input on the processor.

Similarly, hardware interrupts generally require some small amount of work to service a device, but may require much more work to handle any data delivered by the device or modified as a result of the interrupt. Network packets arrive at a machine, for example, and must be examined to identify the program to which they are being sent. Making this decision and giving the data to the program takes time, much more time than should be spent in a hardware interrupt handler, especially since the network card neither needs any more service nor can help with the remaining work.

Most operating systems support the notion of software-generated, or **soft interrupts**, to address these needs. Soft interrupts generally operate at a priority level between hard interrupts and programs. They can interrupt a program, but can in turn be interrupted by an interrupt generated by a device. Also, many hard interrupts have associated soft interrupts to handle any work that does not require interaction with the device. The hard interrupt handler in this case generates the soft interrupt, which typically takes control of the processor after the hard interrupt has finished its work, but defers to other hard interrupts handlers.

Basics of Linux Interrupts

This section discusses the basics of setting up, executing, and removing interrupt handlers within Linux. Many of Linux' interrupt management routines are defined in `arch/i386/kernel/irq.c` and declared in `linux/interrupt.h`. For now, we limit the discussion to hard interrupts; in the next section, we discuss the tasklet abstraction provided to support soft interrupts in Linux.

Installing an interrupt handler: In order to install an interrupt handler in Linux, a call is made—usually by a device driver—to the function `request_irq` shown here:

```
int request_irq (unsigned int irq,
                 void (*handler) (int, void*, struct pt_regs*),
                 unsigned long irqflags,
                 const char* devname,
                 void* dev_id)
```

The function takes five arguments. The `irq` value specifies the interrupt vector. A pointer to the interrupt handler is passed in `handler`. The `irqflags` argument provides a bit vector specifying options (discussed below). Human-readable interfaces, such as the file `/proc/interrupts`, make use of the `devname` field. Finally, the `dev_id` is an arbitrary pointer that is returned to the interrupt handler when the interrupt occurs. Usually, it is a pointer to a structure holding information about the device's status within the kernel; the contents of this block are defined by the device driver, and the kernel views the block as opaque, as is the case here. The function returns 0 on success, and a negative error code on failure. The prototype for `request_irq` resides in `linux/sched.h`, and the implementation in `irq.c`.

Two important flags are defined for the `request_irq` call: SA_SHIRQ and SA_INTERRUPT. The shared IRQ (SA_SHIRQ) flag allows the interrupt vector to be shared by other handlers, but only if *all* of the handlers agree to share. If disagreement occurs, the later request to add a handler is denied through the function's return value. The SA_INTERRUPT flag keeps all interrupts masked on the processor throughout the handler's execution, and should only be used for short handlers. The PIC already masks all interrupts of equal and lower priority, so using SA_INTERRUPT can invert the normal priority.[7]

Interrupt handlers are typically written in C and use standard C linkage, *i.e.*, the usual calling convention. The three arguments to the handler are the interrupt vector (in case one handler is used for multiple vectors), the `dev_id` pointer provided in the call to `request_irq`, and a pointer to a structure holding the values of the registers at the time that the interrupt occurred. The structure actually resides on the stack, and is simply the memory locations to which the registers were pushed at the start of the interrupt.

The Linux kernel keeps track of information pertaining to interrupt vectors in an array, `irq_desc`, of descriptors (of type `irq_desc_t`, defined in `linux/irq.h`). Each descriptor holds a bit vector tracking the status of the interrupt, a pointer to a jump table for the associated interrupt controller, a linked list of interrupt handlers for the vector, a count of calls to disable the vector (these are nested), and a spin lock to manage access to the descriptor.

When a device driver calls the `request_irq` function to install a handler, a new structure to represent the handler is allocated and filled in. The kernel uses the `irqaction` structure, defined in `linux/interrupt.h`, to represent the handler. This structure holds the information provided to `request_irq`, including the handler pointer, the flags, the device name, and the `dev_id` pointer.

When no handlers are currently associated with the requested interrupt vector, the new `irqaction` structure becomes the action list (of one element) for that vector in the descriptor array, and the interrupt controller startup function is called. When other handlers are already present, the new handler and flags are first checked for compatibility with existing handlers. For example, do all agree to share the vector? If the new handler is incompatible, the call fails. Otherwise, the new `irqaction` is linked at the end of the handler list in the requested interrupt vector's descriptor. In effect, the interrupt handler is chained to the end of the existing chain of handlers for that vector.

Linux makes many kernel internals available to the superuser—*i.e.*, the `root` login—through the `/proc` directory. Reading and writing files in this directory translates to executing associated functions within the kernel. As an example, after installing a new handler, `request_irq` creates a new directory in `/proc/irq`; on an SMP, this directory holds

[7] While not mixing these two flags is not explicitly forbidden, only the *first* handler's SA_INTERRUPT flag is checked when deciding whether to unmask interrupts or not before calling the handlers in a list.

a human-readable hexadecimal value representing the bit vector of processors allowed to execute the interrupt (recall the `smp_affinity` function in the controller jump table). The superuser can write a new hex value into this file to change the allowed set of processors for an interrupt. Most operating systems do not expose their internal data in this manner, but it can be convenient at times.

Uninstalling an interrupt handler: Removing an interrupt handler in Linux uses the function `free_irq` shown below, which declared and defined in the same files as `request_irq`.

```
void free_irq (unsigned int irq, void* dev_id);
```

The function takes two arguments: the interrupt vector, and the pointer to the device information structure passed earlier to `request_irq` when installing the handler to be removed.

The function checks the linked list of actions associated with the specified interrupt vector for one with a matching `dev_id` pointer, and, if one is found, removes it from the linked list. When a vector has several handlers chained together, the use of the `irqaction` structures allows Linux to remove the specified handler cleanly, thereby extracting a single link from the chain. Recall that the use of data embedded in the interrupt handlers themselves in DOS made this operation effectively impossible.

If the action being removed is the only one in the vector's list, the interrupt controller's shutdown function is called on the interrupt vector. As before, both the linked list of actions and the associated interrupt controller's jump table are obtained from the interrupt vector's descriptor in the array `irq_desc`.

Interrupt invocation: The main interrupt handling function in Linux is the C function `do_IRQ` in `irq.c`. However, the interface used by the hardware when generating an interrupt is essentially unrelated to the C calling convention. Interrupt invocation for 8259A interrupts thus requires some assembly code to wrap the C function. Such code is called **linkage** because it links the interface used by the hardware to the calling convention used when compiling `do_IRQ`.

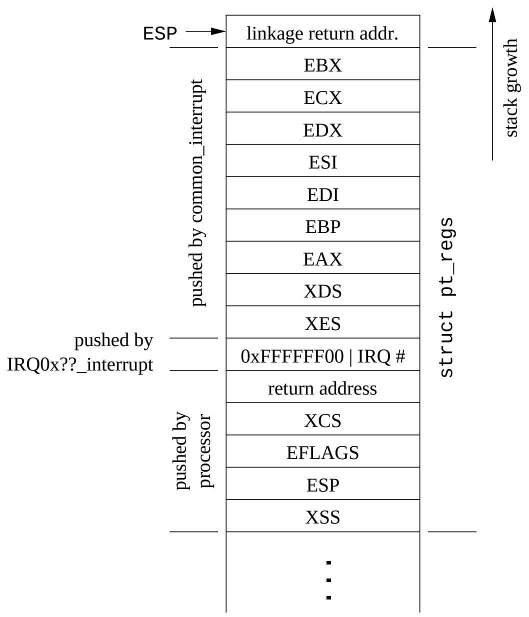
When starting an interrupt, the x86 switches the stack if necessary and records the (unprivileged) stack pointer the flags, and the return address. The first bit of linkage (*e.g.*, `IRQ0x06_interrupt`) pushes the interrupt number onto the stack (*e.g.*, 6) and jumps to a second bit of linkage called `common_interrupt`. This bit of handoff code is generated in `i8259.c` for each 8259A interrupt through use of the `BUILD_IRQ` macro in `asm/hw_irq.h`. The IDT entry is then set up by the `init_IRQ` function by calling `set_intr_gate` with a pointer to the handoff code.

The `common_interrupt` linkage, as defined in `hw_irq.h`, saves all registers not already saved by the processor, calls the `do_IRQ` function, and jumps to `ret_from_intr`, the last bit of linkage, from `arch/i386/kernel/entry.S`. This final part can involve some scheduler activity to give control of the processor to a new program from time to time, but eventually restores all of the registers and executes an IRET instruction (possibly returning control to a different program, but we leave that issue for later in the course).

Interrupt execution: The `do_IRQ` function in `irq.c` serves as the interrupt handler for all 8259A interrupts:

```
unsigned int do_IRQ (struct pt_regs regs);
```

Notice that the argument is a structure, requiring that the whole structure be copied onto the stack before calling the function. In the case of `do_IRQ`, this "structure" is simply the copy of the registers that had to be pushed onto the stack to save them anyway, and no extra work is required to create it. The top of the stack on entry to `do_IRQ` appears in the diagram to the right. The `struct pt_regs` is defined in `asm/ptrace.h`, and the `SAVE_ALL` and `RESTORE_ALL` macros in `entry.S` push and pop the registers from the stack, respectively. The segment registers—XDS, XES, XCS, and XSS—were deliberately ignored in our discussion of x86 assembly code. They are used to partition a machine's physical address space amongst several programs, and will be discussed in more detail later in the course. For now, consider XCS (extended code segment) to be part of the return address, and XSS (extended stack segment) to be part of the stack pointer. The XDS and XES registers are effectively constants.



You might wonder why the linkage is used to record the interrupt vector and to pass control to a general function rather than saving some overhead by instantiating separate copies the code for each interrupt vector. Two factors contribute to this choice: first, the function is not entirely trivial, and making a large number of copies can bloat the kernel; second, while we are considering only 16 interrupts, most servers support up to 224 interrupts through the I/O APIC, making the kernel bloat more substantial.

Put briefly, the `do_IRQ` function uses the interrupt vector passed by the assembly linkage discussed in the last section to find the correct element of the interrupt descriptor array `irq_desc`, interacts with the corresponding interrupt controller using the jump table pointer in the interrupt's descriptor, and calls any handlers installed by `request_irq`.

The detailed behavior of `do_IRQ` depends on the status flags in the interrupt descriptor. As mentioned earlier, each interrupt descriptor contains a bit vector of status flags, which are defined in `linux/irq.h`. The four flags relevant to our discussion are IRQ_PENDING, which indicates that an interrupt has occurred; IRQ_INPROGRESS, which indicates that handlers are currently being executed; IRQ_DISABLED, which indicates that the interrupt vector has been disabled temporarily; and IRQ_REPLAY, which indicates that the source of an interrupt is software rather than hardware, and in particular that the interrupt is being replayed because it was disabled by software when the hardware generated it, as described below.

The first step taken by `do_IRQ` is to acknowledge the interrupt to the interrupt controller. For the 8259A, this call masks the specific interrupt on the PIC and sends an EOI for that interrupt. If this interrupt is disabled and then re-enabled by an interrupt handler, the PIC may raise the interrupt again before `do_IRQ` finishes. After the acknowledgement, the IRQ_REPLAY flag is cleared, as the source of the interrupt is no longer relevant.

Interrupts in Linux may occur while disabled by software or while the same interrupt is already being serviced. The IRQ_DISABLED status flag in the descriptor indicates that the interrupt should not be executed immediately.[8] Instead, the IRQ_PENDING flag is set, and the interrupt is executed (synchronously) when the call is made to re-enable that interrupt. When calling `do_IRQ` from software for this purpose, the IRQ_REPLAY flag is set. An interrupt that occurs while the same interrupt is already being serviced simply leaves the IRQ_PENDING status flag set in the interrupt descriptor; this flag is used later to repeat the handlers for the second (third, *etc.*) interrupt. Both cases skip the next portion of `do_IRQ`, in which the handlers provided by calls to `request_irq` are executed.

Handler execution begins by changing the status from IRQ_PENDING to IRQ_INPROGRESS. A call to `handle_IRQ_event` executes the chain of handlers, and is repeated whenever another interrupt has occurred during execution, as indicated by the presence of the IRQ_PENDING flag. The `handle_IRQ_event` subroutine records the fact that the processor is executing a hardware interrupt by calling `irq_enter`, sets the IF flag to enable interrupts on the processor unless the first handler is marked with the SA_INTERRUPT flag, executes all handlers in the chain for the interrupt, disables interrupts again, and records the fact that the processor is no longer executing a hardware interrupt by calling `irq_exit`. *Note that the function* `do_IRQ` *executes in its entirety with the IF flag clear,* i.e., *with all interrupts masked. Only when executing a handler without the SA_INTERRUPT flag set are interrupts allowed, and the STI and CLI instructions appear in* `handler_IRQ_event`. Once the handlers have been executed without an intervening interrupt of the same type, the IN_PROGRESS flag is removed, and `do_IRQ` proceeds to the cleanup phase.

The last step for handling the hardware interrupt in `do_IRQ` is to tell the interrupt controller that the interrupt has ended. For the 8259A, this call normally unmasks the interrupt on the PIC. However, in the case of an interrupt taken while disabled in software or while the same interrupt is already in progress, it does nothing.

Finally, `do_IRQ` checks for the need to execute soft interrupt handlers by calling `do_softirq`. The return value from `do_IRQ` is always 1.

Additional Interrupt Abstractions

The main soft interrupt support in Linux takes the form of tasklets. A **tasklet** is a data structure used to wrap a single handler function used as a soft interrupt handler. A soft interrupt corresponding to a tasklet is generated when some piece of code—usually a hard interrupt handler—requests that the handler function associated with the tasklet be scheduled for execution. When scheduled, a tasklet is linked into a list to be executed along with any other tasklets

[8] Taking an 8259A interrupt with IRQ_DISABLED set for that interrupt can only occur on an SMP, but can be common with interrupt controllers that do not support masking of specific interrupts on the controller.

of the same priority level. A separate list is maintained for each processor, thus tasklets in an SMP execute on the processor on which they were scheduled. Tasklets typically execute whenever a hard interrupt handler has completed, but may be deferred (due to software having disabled the tasklet, for example), in which case another program running in the Linux kernel periodically tries to execute them.

The tasklet data structure, `struct tasklet_struct`, is defined in `linux/interrupt.h` along with declarations for the tasklet API. The structure has five fields: a pointer to the associated handler function, a 32-bit value to be given to the handler when called, a count of times that the tasklet has been disabled (as with hard interrupts, calls to disable and to re-enable a tasklet can be nested), a bit vector of status flags, and a pointer to form a linked list.

The tasklet API appears in the table below. Dynamically allocated tasklets can be initialized using the `tasklet_init` call. Statically allocated and initialized tasklets can be declared using either of the forms here:

```
DECLARE_TASKLET (name, func, data);
DECLARE_TASKLET_DISABLED (name, func, data);
```

As with the other DECLARE_ macros discussed in these notes, the qualifier `static` should prefix the declaration of any tasklet declared within a function to ensure that the tasklet is not allocated on the stack. In the second form shown above, the tasklet's disable count is initialized to one rather than zero, and an unpaired call to `tasklet_enable` or `tasklet_hi_enable` should be used to enable the tasklet initially. Otherwise, calls to disable and to re-enable a tasklet, like those for hard interrupts, are intended to be nested and to appear in pairs in the code.

initialization

`void tasklet_init (struct tasklet_struct* t, void (*func) (unsigned long), unsigned long data);`	Initialize a dynamically allocated tasklet.

scheduling

`void tasklet_schedule (struct tasklet_struct* t);`	Schedule a tasklet for execution with low priority.
`void tasklet_hi_schedule (struct tasklet_struct* t);`	Schedule a tasklet for execution with high priority.

disabling and re-enabling

`void tasklet_disable_nosync (struct tasklet_struct* t);`	Disable a tasklet, *i.e.*, mask the soft interrupt associated with the tasklet. Pair with calls to enable.
`void tasklet_disable (struct tasklet_struct* t);`	Disable a tasklet and, if it is currently executing on some processor, wait for it to finish. *Can deadlock if called from within the tasklet's handler on an SMP.* Pair with calls to enable.
`void tasklet_enable (struct tasklet_struct* t);`	Re-enable a tasklet, *i.e.*, unmask the soft interrupt associated with the tasklet. Pair with calls to disable.
`void tasklet_hi_enable (struct tasklet_struct* t);`	Re-enable a high-priority tasklet, *i.e.*, unmask the soft interrupt associated with the tasklet. Pair with calls to disable.

Four soft interrupt types are defined in Linux, two of which serve as tasklet priority levels. The handler pointers for each type are kept in the `softirq_vec` array. The array also has a second field called `data` for each soft interrupt, which is apparently unused. The soft interrupt types are:

- HI_SOFTIRQ — high-priority tasklets (function `tasklet_hi_action`)
- NET_TX_SOFTIRQ — network transmission
- NET_RX_SOFTIRQ — network reception
- TASKLET_SOFTIRQ — low-priority tasklets (function `tasklet_action`)

Tasklet priority has little significance, but does affect the order in which tasklets are executed. As mentioned in the previous section, soft interrupts are executed by the `do_softirq` function in `kernel/softirq.c`. Most of the tasklet support code is also in this file.

The do_softirq function first checks whether or not an interrupt of any type, either hard or soft, is already being executed by the processor. If so, the function terminates.

The next few steps are performed with interrupts masked on the processor. After masking interrupts, do_softirq checks for pending soft interrupt types on the processor, and terminates if none are pending (after restoring the previous IF value). Otherwise, it records any pending soft interrupt types and resets the bit vector of pending types. Finally, do_softirq records the fact that the processor is executing a soft interrupt using the __local_bh_disable macro from asm/softirq.h. The IF flag is then set, re-enabling hard interrupts.

With hard interrupts enabled, do_softirq then executes the handler function for each soft interrupt type that was either pending at the start of the function or was raised during the execution of other handlers. The handler for each type is executed at most once. The two tasklet handlers, which we describe in more detail later in the notes, walk a per-CPU linked list of scheduled tasklets and execute each tasklet in the list.

Hard interrupts are then disabled again for the final few steps. The __local_bh_enable macro is then used to record the fact that the processor is no longer executing a soft interrupt. If a soft interrupt type that was executed was raised (scheduled) again during the function's execution, do_softirq wakes a kernel daemon associated with the processor. These daemons are created by spawn_ksoftirqd, which starts a kernel thread for each processor to serve as the soft interrupt daemon, using ksoftirqd as the main function for each thread. The threads check for pending soft interrupts, execute them if necessary by calling do_softirq, and put themselves to sleep. When a soft interrupt must be postponed on some processor, that processor's daemon is reawakened and, once it is given a turn on the processor, the daemon tries to execute the scheduled soft interrupts. After restoring the original IF flag, the do_softirq function terminates.

Tasklet scheduling and execution makes use of the status bit vector in the tasklet structure. Two flags are defined: TASKLET_STATE_SCHED, and TASKLET_STATE_RUN. The TASKLET_STATE_SCHED flag is used both to indicate that a tasklet has been scheduled but has yet to execute, and as a lock bit to ensure that a tasklet is only scheduled on one CPU at a time. On an SMP, a tasklet may be scheduled again while it executes. In this case, the TASKLET_STATE_RUN bit is used (as a lock bit) to serialize execution of the tasklet on the processors on which it is scheduled.

When a tasklet is changed from unscheduled to scheduled, it is linked into either the tasklet_hi_vec or tasklet_vec list for the scheduling processor, depending on the priority requested. A tasklet executes at least once after being scheduled, and usually executes exactly once.

The two tasklet execution functions, tasklet_hi_action and tasklet_action, are almost exactly the same, and merely operate on different linked lists (the two just mentioned). In each case, the current list is extracted with interrupts masked (a critical section, given that the separate lists are maintained for each processor). For each tasklet in this initial list, the tasklet is marked as running (only on an SMP), the disabled count is checked, the scheduled flag is cleared, and the tasklet is executed. If any of the checks fail, *e.g.*, the tasklet is already running elsewhere, or is currently disabled, execution is postponed by linking the tasklet back into the original list and marking the corresponding soft interrupt type as pending for that processor (which causes do_softirq to wake the daemon for that processor).

Interrupt Control and Status Functions

Specific hard interrupts can be disabled and re-enabled using functions declared in asm/irq.h. These functions are intended to be used in disable/enable pairs, and can be used without concern about previous calls, as the total number of calls is tracked in the interrupt descriptor array. For example, rather than masking all interrupts to keep the keyboard interrupt (typically IRQ1) from executing, one might write:

```
disable_irq (1);
/* protected keyboard access code goes here */
enable_irq (1);
```

Enabling and disabling individual interrupts is more effective than masking interrupts through IF, since a disabled interrupt cannot run on any processor in an SMP. However, the call to disable can require interaction with the 8259A, which is substantially slower than flipping the IF flag. Note that the disable_irq call waits for the interrupt to finish executing on any processor on which it is currently executing. As a result, *calling this function from an interrupt* *handler may lead to deadlock*. Instead, call the `disable_irq_nosync` function, which disables the interrupt but does not wait for a currently executing handler to finish.

As mentioned in the previous section, similar nesting of disables and enables is allowed with the tasklet API. In the case of tasklets, disabling a single tasklet with `tasklet_disable` and disabling all tasklets with `local_bh_disable` (discussed below) have roughly the same cost, but analogous scope, *i.e.*, `tasklet_disable` disables one tasklet on all processors, and `local_bh_disable` disables all tasklets on one processor.

Information about the interrupts currently executing on each processor is recorded in the `irq_stat` array. Each array element is an `irq_cpustat_t` structure, as defined in `asm/hardirq.h`, and is accessed using the standard macros defined in `linux/irq_cpustat.h`:

`softirq_pending (cpu)`	bit vector of pending soft interrupt types
`local_irq_count(cpu)`	number of hard interrupts in progress
`local_bh_count(cpu)`	number of soft interrupts in progress
`syscall_count(cpu)`	number of system calls in progress
`ksoftirqd_task(cpu)`	soft interrupt daemon identifier
`nmi_count(cpu)`	number of non-maskable interrupts in progress

These macros are **lvalues**—*i.e.*, you can assign values to them, increment them, *etc.*—and should be used with another specific macro in places of the `cpu` argument. For example,

```
softirq_pending (smp_processor_id ())
```

returns the bit vector of pending soft interrupt types on the current processor. In GDB, a fixed processor number can be used, and 0 should be used on a uniprocessor. Try `print irq_stat[0]`.

A number of status and interrupt control functions and macros make use of the `irq_stat` array. These functions are defined in either `asm/hardirq.h` or `asm/softirq.h`, and are described in the table below.

status functions

`int in_interrupt ();`	Returns 1 if processor is currently executing an interrupt handler (either soft or hard), 0 otherwise.
`int in_irq ();`	Returns 1 if processor is currently executing a hard interrupt handler, 0 otherwise.
`int in_softirq ();`	Returns 1 if processor is currently executing a soft interrupt handler, 0 otherwise.

control functions

`void irq_enter (int cpu, int irq);`	Increment the count of hard interrupts being executed on processor `cpu`. Second argument is ignored.
`void irq_exit (int cpu, int irq);`	Decrement the count of hard interrupts being executed on processor `cpu`. Second argument is ignored.
`void local_bh_disable ();`	Mask/disable soft interrupt execution on this processor; works by incrementing the count of soft interrupts being executed on current processor.
`void local_bh_enable ();`	Unmask/enable soft interrupt execution on this processor; works by decrementing the count of soft interrupts being executed on current processor.

Example: The Real-Time Clock Driver

Many x86-based computers have real-time clocks that are compatible with Motorola's MC146818 chip. In this section, we briefly discuss some of the capabilities of this chip and examine the interrupt-related functionality in the driver. The first programming assignment for our course, MP1, involves using the real-time clock driver to animate text-based graphics on the screen using system calls and a soft interrupt handler. You will only write the x86 assembly code for handling the system calls and soft interrupts; we will provide the linkage between the driver and your code as well as a user-level test harness to enable you to perform most debugging without recompiling the kernel.

All device driver code for the real-time clock is in `drivers/char/rtc.c`. The interrupt handler, `rtc_interrupt`, is installed in by the `rtc_init` function, as shown here:

```
if (request_irq (RTC_IRQ, rtc_interrupt, SA_INTERRUPT, "rtc", NULL)) {
    printk (KERN_ERR "rtc:  IRQ %d is not free.\n", RTC_IRQ);
    return -EIO;
}
```

RTC_IRQ is defined as 8 in `asm/mc146818rtc.h`. The `rtc_init` function is called by the kernel initialization code, or, if the the real-time clock driver is compiled as a kernel module, when the module is loaded into the kernel.[9] The following declarations generate the appropriate calls, including a call to `rtc_exit` if the driver is compiled as a module and removed from the kernel.

```
module_init (rtc_init);
module_exit (rtc_exit);
```

The `rtc_exit` function uninstalls the interrupt handler using this call:

```
free_irq (RTC_IRQ, NULL);
```

The MC146818 can generate interrupts for three types of events; each type can be configured and handled separately. The first is an alarm clock, which generates an interrupt at a specified time. The second is an update notification, which generates an interrupt each second, *i.e.*, whenever the time changes. The third is an interrupt generated periodically, with a period ranging from 2 to 8192 Hz (limited by Linux), programmable in powers of two. The function below handles the hard interrupts generated by the clock; the modifications in boldface were made to create and schedule an associated soft interrupt after each hard interrupt. These soft interrupts are then used to drive the animation in MP1, and the function `mp1_rtc_tasklet` referenced in the tasklet declaration must be written by each student.

```
/* Student's tasklet */
static DECLARE_TASKLET (mp1_rtc_tasklet_struct, mp1_rtc_tasklet, 0);

static void rtc_interrupt (int irq, void* dev_id, struct pt_regs* regs)
{
    /*
     * Can be an alarm interrupt, update complete interrupt,
     * or a periodic interrupt.  We store the status in the
     * low byte and the number of interrupts received since
     * the last read in the remainder of rtc_irq_data.
     */

    spin_lock (&rtc_lock);
    rtc_irq_data += 0x100;
    rtc_irq_data &= ~0xff;
    rtc_irq_data |= (CMOS_READ(RTC_INTR_FLAGS) & 0xF0);

    if (rtc_status & RTC_TIMER_ON)
        mod_timer (&rtc_irq_timer, jiffies + HZ/rtc_freq + 2*HZ/100);

    spin_unlock (&rtc_lock);

    /* Now do the rest of the actions */
    wake_up_interruptible (&rtc_wait);

    /* Schedule the MP1 tasklet to run later */
    tasklet_schedule (&mp1_rtc_tasklet_struct);

    kill_fasync (&rtc_async_queue, SIGIO, POLL_IN);
}
```

[9] Linux supports loading sections of code called modules into the kernel on demand, *i.e.*, only when they are needed, and being able to remove them when they are no longer necessary. Modules can keep the kernel small while supporting plug-and-play style functionality, but they make debugging with KGDB more complex. We thus chose to turn module support off in our kernel configuration files, and will not discuss it in any detail in the class.

The real-time clock device provides a file interface through which four-byte updates arrive periodically. The `rtc_irq_data` variable tracks both the number of interrupts that have occurred since the last update through the file interface as well as the type of interrupt last seen. The interrupt type occupies the low byte of the variable, and the remaining (high) 24 bits are used to count the interrupts between updates. Symbolically, as given in `linux/mc146818rtc.h` the flag bits are RTC_IRQF to indicate that some interrupt has occurred, RTC_PF for a periodic interrupt, RTC_AF for an alarm clock interrupt, and RTC_UF for an update interrupt. The contents of the `rtc_irq_data` variable are delivered as the update when a program reads the clock via the file interface.

The file interface may be running on another processor when the interrupt occurs, so the handler first acquires a spin lock to protect the accesses as well as the device interaction (using `asm/mc146818rtc.h`'s CMOS_READ macro). The conditional and timer operation within the critical section handle cases in which the period of the periodic interrupt is set too high and an interrupt is missed. Certain devices have higher priority than the real-time clock, and their interrupt handlers may prevent an overly frequent clock interrupt from being observed before the next one should occur. In this case, the clock may stop generating interrupts, and the device driver must notice, reset it, and estimate the number of interrupts lost. Use of Linux' internal timers are outside of the scope of our interest for now, but feel free to read the rest of the clock driver code as an example of use.

After releasing the spin lock, the original interrupt handler wakes up any program blocked waiting to read an update from the clock's device file (`wake_up_interruptible`) and sends a signal to any program that has asked to receive signals when the clock's device file has new data to be read (`kill_fasync`). For MP1, we simply inserted scheduling the tasklet into this list of things that need to be done by the handler. Along with the declaration and writing the soft interrupt handler (left to you in MP1), this simple addition to the code is enough to support our needs. For proper module support, we must also add a call to `tasklet_kill` after the call to `free_irq` to wait until any scheduled soft interrupts have been executed.

Terminology

You should be familiar with the following terms after reading this set of notes.

- procedural abstractions
  - system call
  - interrupt
  - exception
  - handler (function)
- processor support
  - vector table
  - Interrupt Descriptor Table (IDT)
  - interrupt enable flag (IF)
  - non-maskable interrupt (NMI)
- input/output concepts
  - I/O port space
  - independent vs. memory-mapped I/O
- software and programming abstractions
  - application programming interface (API)
  - linkage
  - lvalues
  - jump table (for function indirection)
- synchronization concepts
  - critical section
  - atomicity
- synchronization mechanisms
  - mutex
  - spin lock: lock/obtain and unlock/release operations
  - semaphore: down/wait and up/signal operations
  - reader/writer lock
- synchronization problems
  - race condition
  - deadlock
  - starvation
- interface between processor and devices
  - interrupt controller
  - Programmable Interrupt Controller (PIC), such as Intel's 8259A
  - end-of-interrupt (EOI) signal
- interrupt abstractions in software
  - interrupt chaining
  - hard vs. soft interrupts
  - Linux' tasklet abstraction

Advanced Topics

This section mentions a few topics for those interested in further study. None of these are included in the required course material.

Auto-probing for interrupts: Linux has support for automatically finding the interrupt vector generated by a device rather than hardcoding it into a device driver. The protocol for using this support is described at end of `linux/interrupt.h`. Essentially, the device driver puts the device into a state in which it will generate an interrupt shortly. The driver then waits for a bit, during which time the vector numbers of all interrupts received but not already associated with other devices are recorded. At the end of the waiting period, the correct vector is identified if exactly one unknown interrupt arrived during the period.

Spin locks: The caching and bus transaction aspects of the algorithms may be beyond you at this point, but are mentioned here for interest.

The x86 ISA uses the notion of instruction prefixes to change the properties of certain instructions. Instruction prefixes are usually one-byte sequences placed in the code before another instruction. One such prefix is called "LOCK," and has the effect of claiming the memory bus on behalf of the processor for the duration of the following instruction. On a uniprocessor, or for an instruction that performs zero or one memory bus transactions, this prefix has no effect. However, for an instruction that reads a value, modifies it, and writes it back to memory—sometimes called a read-modify-write operation—locking the bus on a multiprocessor guarantees that no other processor's bus transactions can be interleaved between the two transactions associated with our processor's instruction. Read-modify-write operations are the instruction-level equivalent of critical sections. If two processors try to add one to the value at a memory location in this way, most programmers would expect a total of two to be added to the value in memory. However, without lock prefixes on the instructions, both processors may read the same value from memory and thus write back the same value, *i.e.*, the original value plus one. In most modern systems, atomic operations are performed in the lowest-level cache, which is much faster and less intrusive than locking the memory bus, but is still fairly slow relative to the processor's clock (tens of cycles per operation).

The simplest spin lock algorithm, known as **test-and-set** (T&S), repeatedly attempts to obtain a lock by atomically reading a bit from memory and replacing that bit with a 0 (using either an instruction with a LOCK prefix or a special atomic instruction). If the bit was previously a 1, the lock has been obtained. If it was previously a 0, some other code is holding the lock, and the algorithm tries again. Most modern ISAs, including newer flavors of x86, have instructions designed specifically for this purpose (see BTR and BTS).

While the T&S algorithm can be best when lock contention is low, it generates far too much bus traffic when more than one processor is waiting on a lock. Each processor continuously demands ownership of the lock, which thrashes back and forth across the bus, slowing traffic from other processors, potentially including the one that owns the lock.

Linux uses a variant of a slightly better algorithm known as **test-and-test-and-set** (T&T&S). This algorithm takes advantage of the fact that the cache line with the lock can be held in many processors' caches so long as none of them tries to write to the lock. In particular, after failing to obtain the lock once, a processor backs off and reads the lock until it sees the lock present. In effect, bus traffic is generated only when a new processor tries to obtain the lock or the processor with the lock gives it up. The specific variant used in Linux decrements a lock byte atomically and decides whether or not the lock was obtained by checking the sign flag.[10]

Memory consistency: The level at which we have discussed synchronization is necessarily somewhat simplistic, and ignores the fact that some ISAs allow a processor to reorder memory transactions in the interest of performance. On an ISA that allows such reordering, telling the compiler not to move memory operations past critical section boundaries is not sufficient to ensure atomicity. Instead, additional instructions must be used to tell the processor itself to avoid such behavior. These issues are covered to some degree in ECE411, and in more depth in ECE511.

Lock-free synchronization: For general and specific methods of constructing synchronization methods without locks, the non-blocking and wait-free work by Herlihy and Wing, as well as the notion of universal synchronization primitives, is the natural starting point. Rather than provide citations here, I refer you to the bibliography of my dissertation, which you can find online through my home page. To my knowledge, almost none of this material is covered by classes here, although you may see some of it in ECE/CS428.

Connecting devices to the processor: If you are interested in the details of the hardware used to connect a processor to devices, busses, and other hardware elements, you should take ECE412, which focuses on this type of problem in the context of an IPAQ PDA (personal data assistant, *i.e.*, handheld computer) running an embedded version of Linux.

[10]This approach relies on the number of contenders being no more than 129, at which point wraparound magically generates 128 new locks.

Synchronization implementations: Some documentation on the algorithms used for the synchronization constructs in Linux is available in the header files.

Header and Source Files

The table below serves as a brief reference on the contents of those header and source files within the Linux-2.4.18-3 kernel that are relevant to the topics covered in these notes.

header files	
include/asm	
atomic.h	a few atomic primitives for x86
bitops.h	bit operations (*e.g.*, test bit, find first bit set, *etc.*)
hardirq.h	irq_cpustat_t structure, which tracks hard and soft IRQ information for each processor; a few interrupt status-checking functions; irq_enter and irq_exit macros
hw_irq.h	SYSCALL_VECTOR (0x80); BUILD_IRQ macro used in `i8259.c` to generate handler code for IDT
irq.h	controller-independent interrupt management interface
locks.h	antiquated — DO NOT USE
mc146818rtc.h	x86-dependent parameters for real-time clock
rwlock.h	helper code for r/w locks and semaphores
rwsem.h	x86 implementation of reader/writer semaphore macros
semaphore.h	defines semaphores in interruptible and non-interruptible forms
softirq.h	soft interrupt management and status-checking functions
spinlock.h	spin lock and rw_lock functionality for x86 SMPs
system.h	x86-specific hardware interrupt primitives (*e.g.*, masking, flag save/restore)
include/linux	
interrupt.h	architecture-independent hard and soft interrupt structures and functions
irq.h	interrupt controller abstraction (PIC jump table), hard interrupt description structure, status flags
irq_cpustat.h	standard access macros for CPU stat structure defined in `hardirq.h`
mc146818rtc.h	architecture-independent parameters for real-time clock
rwsem.h	public interface to reader/writer semaphores
rwsem-spinlock.h	architecture-independent implementation of reader/writer semaphore macros
tqueue.h	task queues; antiquated in Linux-2.6 — DO NOT USE
sched.h	request and free IRQ function prototypes

source files	
arch/i386/kernel	
entry.S	interrupt, exception, and system call linkage
i8259.c	8259A PIC code, including jump table (XT-PIC in `/proc/interrupts`)
irq.c	irq_desc declaration and initialization; most of the hard interrupt handling code
nmi.c	NMI handling
semaphore.c	functions used to support semaphores and reader/writer spin locks when contention seen on lock (fast path for no contention given inline via header files)
drivers/char	
rtc.c	real-time clock device driver (mc146818)
kernel	
softirq.c	soft interrupt code, including kernel daemon for wakeup, *etc.*
lib	
rwsem.c	functions used to support reader/writer semaphore when contention seen
rwsem-spinlock.c	alternate implementation — NOT COMPILED INTO OUR KERNEL