|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| X86 Registers | x86 Feature   * Little endian      * Variable length instructions * Complex instruction set | | Imm. Val Vs Mem   * “$” before num will indicate an imm. val * If only num is there, then num is treated as address          * Immediate values can be 32-bits   + $0x\_\_ = hex   + $0\_\_\_ = octal   + $5 = decimal 🡪 first num has be non 0 | | | | Data Alignment   * Optional in x86 * makes access faster due to set up of memory * Example: ***.Align 4***; ***.long 0xFFFF*** * This places the at a mem address that is a multiple of 4 * Want to **align to a multiple of the largest primitive in a struct** | | | Device I/O   * Independent and mem map * Independent uses I/O ports * Examples      * When you get a word, the second byte is grabbed in sequences. |
| **Displacement (SR1, SR2, SCALE)**   * Scales SR2 by 1, 2, 4, 8 and then adds it to SR1 and the displacement value * Examples: | |
| Random instructions   |  |  |  | | --- | --- | --- | | Imull %ebx, %eax | Shr shift and pad 0s | Sar shift and SE | | Pushl (%ebp) | Shl/Sal are same | Movsx: move SE | | | | |
| Calling Convention   1. Caller save registers 2. Push arguments onto stack from left to right 3. Call function 4. Set up stack    1. Pushl %ebp    2. Movl %esp, %ebp 5. Callee save registers 6. Create space for local vars 7. Perform function 8. Place return val in %eax 9. Tear down local vars 10. Pop callee save registers 11. Leave; Ret 12. Pop arguments off of stack 13. Handle return val 14. Pop caller save registers | RTL for Important Instructions   * CALL foo   + Pushl %eip   + EIP <- foo * Leave   + ESP <- EBP + 4   + EBP <- M[EBP] * Ret   + Popl %eipssss | Interrupts and Synchronization   * System calls are made by the user to get access to system resources. The allow for execution of kernel code by user.   + Is accessed by calling **INT 0x80**. The arguments to the system call are placed in registers. **EAX contains the system call num** * Interrupts are generated by hardware devices when they need to be serviced by the system like disk drives, printers, and etc.   + **Vectors from 0x20 to 0x2F** ----> uses PIC to perform the interrupt * Exceptions occur when the user makes a mistake. For example, it executes divide by 0 or invalid opcode.   + **Intel reserves the** **first 32 values** in the IDTfor exception handlers | Also contains NMI handlers | | | | | | | | |
| Run Time Stack |
| X86 Different Call Types   * The star tells the compiler that this offset needs to be calculated at run time and cannot be assumed to be constant. * Random: **EACH LABEL DEFINITION MUST BE FOLLOWED BY A COLON** | | | | | | IDT   * X86 uses a **single vector table** to access handlers * Exceptions are first, then IRQs, and syscall at x80 * The IDT stores the pointers to the handlers for the exceptions, interrupts, and system calls. * Vectors are used to index into this jump table and access the correct handler. * To return from a handler, one needs to use **IRET**. | | |
| Shared Data and Resources   * Synchronization is needed to protect against data being corrupted between multiple threads and interrupt handlers. * Due to this a combination of spin locks and CLI/STI need to be used to protect against any data corruption.   + Combination is needed because **CLI/STI** work well on a uniprocessor system. However, they are of no use in a multiprocessor system. * We can’t just use a shared variable as a flag and then grab data. This is because an interaction with such a lock need to be fully **ATOMIC**. * Remember to add **VOLATILE** as part of any variable declaration that will be checked for changes repeatedly in a tight loop. * Due to this we have the **spin\_lock API and semaphores API** to help with the synchronization process. * **TRICK:** If you are reading or changing any shared variable, then you should own the lock. | | | | | | | | | SMP: Symmetric Multiprocessor | |
| Spin Lock   * Spins in a tight loop while **ATOMICALLY** checking to see if lock is unlocking. * If is unlocked, then it acquires the lock returns. * **The order in which locks are taken need to be the same.** | Assembly spin lock | Assembly spin unlock | | Semaphores   * Semaphores are more generally recommended to be used in user code. * The **key difference**: semaphores go to sleep and voluntarily give up the processor for other threads * They can be used as a **mutex** or they can be used to place a limit on the number of threads that can share a given resource. * If used with spin locks, then get the **semaphore first** and then try to get the spin lock. The reason for this is if a thread goes to sleep after getting a spin lock, this could cause a **dead lock**. | | | | | | |
| Spin Lock Example | | | | Problems with synchronization:   * **Dead Lock:** when a thread tries to get a lock and is stuck in the tight loop forever. * **Live Lock:** When a thread tries to get a lock and lets it go and then tries the same steps again. It looks like it’s doing work but it’s just stuck. * **Starvation:** when there are reader and writers spin locks, starvation becomes a problem because if more and more readers keep coming in then the writer can never get to write. * These problems can be avoided by adding a lock sometimes. | | | Reader/Writer\_Spin Lock   * Works on the same principles as the spin lock. * However, allows multiple readers at once and **at most** one writer at any given time. * This has a problem of starvation because as long as more readers are coming, **the writer is kept waiting**. Due to this the writer could theoretically wait forever. * To beat starvation, an extra lock can be added to the struct to allow access to reader/writer\_lock. | | | Reader/Writer\_Semaphores   * Works on the same principles as a semaphore. While put threads to sleep while they are not allowed to access the shared data. * There is no problem of starvation. This is handled internally through the scheduler. * **If there is a writer waiting, then any new readers that come are queued and have to wait until the writer gets to write and finishes writing.** * Due to this, reader/writer semaphores are less work. |
| When to use which synchronization method: | | | | | More specific when to use which: | | | | | General Thoughts:   * Keep critical sections short. * Long critical sections can cause programs to crash sometimes. * Spin locks on uniprocessors are considered no ops. |
| MP1 Dispatcher | | | | | | MP1 Facts:   * Tasklet is used to make sure that the interrupt handler is short so that other interrupts don’t have to be kept waiting. * Tasklet is like a software interrupt which has higher priority than user code but can still be interrupted. | | | | Good luck ☺ |
| 8259A Facts:   * This is used as an abstraction between the processor and the interrupts. * The PIC internally masks all interrupts that are below the priority of the one that is being serviced. * It knows which one is currently being serviced. * Steps:   + PIC raises the INT   + Processor strobes INTA’ (used as a synchronization since PIC can be slow)   + PIC writes IRQ to data bus   + Processor uses this to index into **IDT**   + After handling is finished, the processor sends and EOI(End of interrupt) telling the PIC that the interrupt has been handled**. If EIO is not received, the PIC will still think that the interrupt is being processed.** | | | | | | | Pins on PIC:   * INT is only connected to the master PIC * INTA’ is connected to all PICS * A and CS’ are shown in the **PORT ADDRESS BUS** below * The D is connected to all PICS and is internally connected to the **PORT DATA BUS** * RD’ and WR’ are used from the processors perspective to tell the PIC what it is doing. * **SP’ is pulled high on the master** * **SP’ is grounded for all slaves** * CAS bus is connected from the master too all of the slave PIC. And is used by the master to tell the PICS which slave has control of the data bus. | | | |
| 8259A Initialization:   * The initialization sequence requires that **four initialization control words (ICWs)** be sent to the 8259A. * 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 signaling, and a couple of other (unused) options. | | | | | | | | | | |
| Interrupt Controller Jump Table:   * **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. * **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 | | | | | | | | | | |
| Interrupt Chaining:   * Have a linked list of interrupt handlers associated with an interrupt line * 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. | | | | | | | | | | |