- reader/writer spin locks (rwlock_t)
  - extension of spin locks
  - use for short critical sections
  - ok to use in interrupt handlers
  - admit writer starvation (you must ensure that this not happen)
- reader/writer semaphores (struct rw_semaphore)
  - extension of semaphores
  - use only in system calls
  - do not admit writer starvation (new readers wait if writer is waiting)

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

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

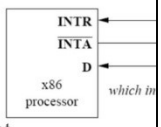
***Hardware interrupts*** require some small amount of work to service a device, but may require much more work to handle any data delivered by the device or modified as a result of the interrupt

***Software interrupts*** can interrupt programs, but can be interrupted by device // allow slower devices to get attention from processor in a timely manner w/o requiring processor to poll devices periodically

Spinlocks – ensures no interrupt & suitable for quick processes like timing // poll in a tight loop, doing nothing else until the resource is acquired

Semaphore – long critical section = longer waiting time; can put waiting processes to sleep & free processor to do other processes first; sys call; Up/Down

Memory mapped to two ports
- Command port (e.g. 0x20)
- Data port (MUST be Command Port + 1)



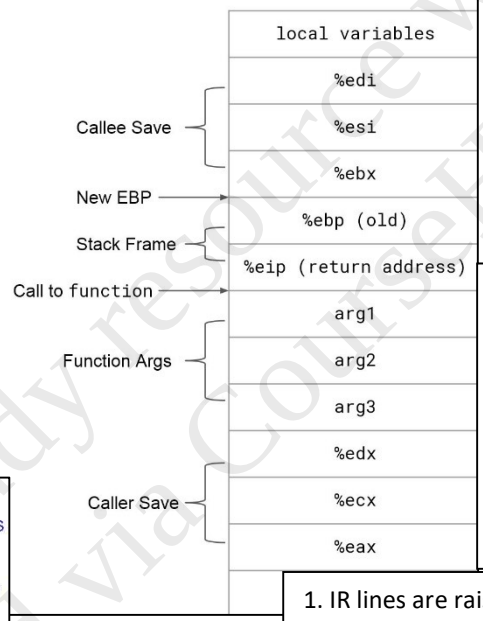

CPU - PIC Signals
- INTR - Activated by the PIC upon interrupt
- INTA - Pulsed by the CPU whenever
- D - Bidirectional communication between CPU and PIC
  - Used in programming the PIC, sending EOI, etc.

PIC Specific Signals
- A - Distinguishes Command/Data port on PIC
  - Can be directly mapped to ADDR[0] (why?)
- CS - Determines whether the given PIC should be active
  - Checks if ADDR[31:1] == PORT[31:1]

It's not good to take too much time in a hardware interrupt handler- other interrupts may need to do things too! That's what software interrupts are for
- Software interrupts operate at priority level between regular programs and hardware interrupts, so hardware interrupts can generate a software interrupt to handle more time-intensive tasks, allowing other hardware interrupts to interrupt the software interrupts

1. Lock & save flags
2. Mask interrupts to PIC so you don't get disturbed while initializing
3. Initialize PIC
4. Unmask interrupts
5. Unlock & restore flags

```
function(arg1, arg2, arg3):
```




Deadlock – waiting thread is still holding onto another resource that the first needs before it can finish // locks & unlocks are in reverse order

Livelock – thread tries to get lock, lets it go, then tries again//seems like it's working but it's stuck

Starvation – if many readers try to come in and writers can't write

| | port(A=?) | info contained in Initialization Control Word |
|---|---|---|
| ICW1 | 0 | start init, edge-triggered inputs, cascade mode, 4 ICWs |
| ICW2 | 1 | high bits of vector # |
| ICW3 | 1 | master: bit vector of slaves; slave: input pin on master |
| ICW4 | 1 | ISA=x86, normal/auto EOI |

## 8259A PIC

1. IR lines are raised that set corresponding IRR bits
2. PIC resolves priority and sends INT signal to CPU
3. The CPU acknowledges with INTA pulse.
4. When INTA signal received from CPU, highest priority ISR bit is set & corresponding IRR bit is reset. PIC does not drive data bus during this period.
5. The CPU will send second INTA pulse. PIC releases pointer to data bus from where it is read by the CPU.
6. ISR bit remains set until EOI command is issued at the end of interrupt subroutine.

*INTA* – strobes pin after receiving INT signal // PIC sees & writes interrupt vector to D

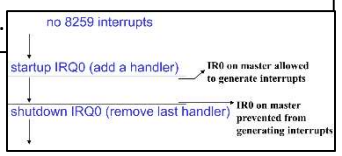
*RD* – high, processor will read data (vector #) from PIC

*WR* – After the interrupt handler finishes, tells the PIC it is done by writing to A & D // processor will write data (comm and, EOI) to PIC // done via OUT

- PIC removes interrupt from the list of in-service interrupt and returns to waiting

*SP* – master(1) or slave(0) || *CS* – ADDR[1:15] // remove LSB

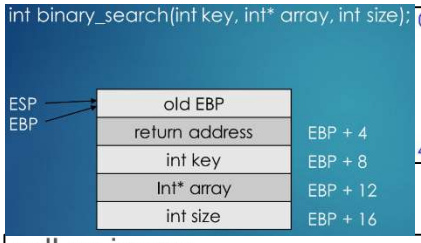
*CAS* -- identify a particular slave PIC to write to the data bus // output pins for master & input pins for slave




*EOI* – If interrupt handler fails to send EOI, PIC keeps masking all interrupts of equal/lower priority indefinitely

*Ack* – start of interrupt handling acknowledge receipt of interrupt // masks the interrupt on the PIC, then sends the EOI signal

*End* – called to end interrupt//enables interrupt (unmasks it) on PIC || *disable* and *enable* allow nested disabling and re-enabling of active interrupts




0. save caller-saved registers (if desired)
1. **push** arguments onto stack
2. make the call
3. **pop** arguments off the stack
4. restore caller-saved registers

Volatile - value of the variable may change at any time w/o any action being taken by the code the compiler finds nearby -- ESP instead of EBP: won't mess up base ptr, don't need to save old ebp; but no guarantee ESP is in right spot b/c it changes over time with stack

-- After call, must restore stack ptr (ie pushl $16, call , addl $4, %esp)||store args from RTL

call vs jump:

- jump → `jmp LABEL`
- call → `pushl %eip; jmp LABEL`

enter - `pushl %ebp; movl %esp, %ebp`

- "creates" the stack frame

leave - `movl %ebp, %esp; popl %ebp`

- "tears down" the stack frame

ret - `popl %eip`

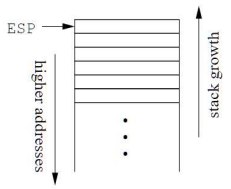
**Stack operations:** The x86 ISA supports a stack abstraction directly rather than as a software convention. The PUSH and POP instructions provide the necessary functionality. As shown to the right, the stack convention used is that ESP contains the address of the element on top of the stack. The stack grows downward in addresses, like that of the LC-3, so

```
pushl %eax   # M[ESP - 4] ← EAX, ESP ← ESP - 4
```

is equivalent to

```
movl  %eax,-4(%esp)  # M[ESP - 4] ← EAX
subl  $4,%esp        # ESP ← ESP - 4
```

Other than a POP into the EFLAGS register, PUSH and POP do not affect the flags.




```
/* call the function */    # the call site...
value = a_func (10, 20);       pushl  $20            # push second argument
                               pushl  $10            # push first argument
                               call   a_func         # call the function
                               addl   $8,%esp        # pop the arguments
                               movl   %eax,-4(%ebp)  # store result in 'value'

int                        a_func:
a_func (int a, int b)          pushl  %ebp           # save old frame pointer
{                              movl   %esp,%ebp      # point to new frame
   int result;                 subl   $4,%esp        # make room for 'result'
                               movl   8(%ebp),%eax   # put 'a' into EAX
   result = a * b + 1;         imull  12(%ebp),%eax  # multiply EAX by 'b'
                               incl   %eax           # add one
   return result;              movl   %eax,-4(%ebp)  # store into 'result'
}                              movl   -4(%ebp),%eax  # return 'result' in EAX
                               leave                 # restore frame pointer
                               ret                   # return
```

```
spin_lock:
    movl   4(%esp), %eax
loop:
    movl   $1, %ecx
    xchgl  %ecx, (%eax)
    cmpl   $1, %ecx
    je     loop
    ret

spin_unlock:
    movl   4(%esp), %eax
    movl   $0, (%eax)
    ret
```

Initially, PIC interrupts masked out using mask
*Startup* -- 8259A tells the appropriate PIC to allow the interrupt line to generate interrupts; unmasks the interrupt on PIC
*Shutdown* -- called when the last handler is removed from an interrupt & for 8259A, tells the appropriate PIC to mask interrupt

| 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 |

What does the "_p" mean on the "outb" macros?
- add PAUSE instruction after OUTB; "REP NOP" prior to P4
- delay needed for old devices that cannot handle processor's output rate

Critical section spans the whole function; why?
- avoid other 8259A interactions during initialization sequence
- (device protocol requires that four words be sent in order)

Why use _irqsave for critical section?
- this code called from other interrupt initialization routines
- which may or may not have cleared IF on processor

^^^^^^^^^^^^^^

hw_irq_controller structure/struct irq_chip --each vector # associated with table; used to interact with an appropriate PIC

Interrupt Chaining – linked list of interrupt handlers associated w/ interrupt line

--for >1 device • must query devices to see if they raised interrupt • not always possible

--for 1 device • must avoid stealing data/confusing device (ie by sending two characters to serial port; in response to interrupt declaring port ready for one char; reading mouse location twice; if device protocol specifies reading once per interrupt)

- software-generated (soft) interrupt
- (similarly, but later, signals—user-level soft interrupts)
- runs at priority between program and hard interrupts
- usually generated
  - by hard interrupt handlers
  - to do work not involving device

Linux version is called tasklets

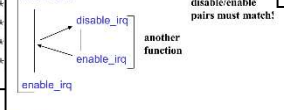
| human-readable name |
|---|
| startup function |
| shutdown function |
| enable function |
| disable function |
| mask function |
| mask_ack function |
| unmask function |

Kernel test – safer & won't corrupt kernel-not using actual mem space that program will be running in -invalid RW silenced in user level/will crash in kernel

Struct padding – combine types into sizes of 4 bytes and add buffer to make sure final struct size is multiple of 4

```
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                               */
);
```

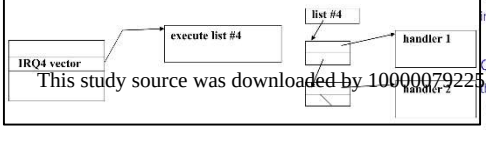
```
asm volatile ("      # local_irq_restore macro implementation
    pushl %0         # save input 0 to stack
    popfl            # pop input 0 into EFLAGS
" :                  /* no outputs                             */
  : "g" (flags)      /* input 0 is a general-purpose register */
                     /*   which should hold the value in flags */
  : "memory", "cc"   /* see text                               */
);
```




on 8259
- first *disable_irq* calls jump table disable, which masks interrupt on PIC
- last *enable_irq* calls jump table enable, which unmasks interrupt on PIC

| | | |
|---|---|---|
| 0x00–0x1F | 0x00 division error<br>⋮<br>0x02 NMI (non-maskable interrupt)<br>0x03 breakpoint (used by KGDB)<br>0x04 overflow<br>⋮<br>defined by Intel: 0x0B segment not present<br>0x0C stack segment fault<br>0x0D general protection fault<br>0x0E page fault<br>⋮ | |
| 0x20–0x27 | 0x20 IRQ0 — timer chip<br>0x21 IRQ1 — keyboard<br>0x22 IRQ2 — (cascade to slave)<br>0x23 IRQ3<br>master 8259 PIC: 0x24 IRQ4 — serial port (KGDB)<br>0x25 IRQ5<br>0x26 IRQ6<br>0x27 IRQ7 | example of possible settings |
| 0x28–0x2F | 0x28 IRQ8 — real time clock<br>0x29 IRQ9<br>0x2A IRQ10<br>0x2B IRQ11 — eth0 (network)<br>slave 8259 PIC: 0x2C IRQ12 — PS/2 mouse<br>0x2D IRQ13<br>0x2E IRQ14 — ide0 (hard drive)<br>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 | |

- interrupt chaining with linked list data structure
- (not list embedded into code!)




16-bit port space
- byte addressable
- little-endian (looks like memory)

Instructions

IN port, dest.reg

OUT src.reg, port

The register operands are NOT general-purpose registers
- all data to/from AL, AX, or EAX
- port is either an 8-bit immediate (not 16!) or DX

independent I/O — use special instructions and a separate I/O port address space

memory-mapped I/O — use loads/stores and dedicate part of the memory address space to I/O

On full SMP systems, where multiple threads could be active in the kernel at the same time and interrupts could be delivered to pretty much any CPU, it's no longer enough to only disable interrupts on single processor, or only grab a single lock. Both are required: disabling interrupts protects from IRQ handler *on the same CPU*, holding a lock protects from other threads entering the same critical sections *on different CPU*. This is exactly why `spin_lock_irqsave()` and `spin_unlock_irqrestore()` were invented.