**Spin lock implementation**

**Spin unlock implementation**

typedef uint32\_t spinlock\_t;

void **spin\_lock**(spinlock\_t \*lock);

movl 4(%esp), %eax

loop: movl $1, %ecx

xchgl %ecx, (%eax)

cmpl $1, %ecx

je loop

ret

void **spin\_unlock**(spinlock\_t \*lock);

movl 4(%esp), %eax

movl $0, (%eax)

ret

a: spinlock\_t the\_lock = SPIN\_LOCK\_UNLOCKED or void spin\_lock\_init (spinlock\_t \*lock);

void spin\_lock(spinlock\_t \*the\_lock); //obtain lock

// critical section

void spin\_unlock(spinlock\_t \*the\_lock); // release lock

b: spinlock\_t the\_lock = SPIN\_LOCK\_UNLOCKED //initilialization

unsigned long flags;

spin\_lock\_irqsave(&the\_Lock, flags); //save the state of interrupt enable in

//flags and then disable interrupts.

//critical section

spin\_unlock\_irqsave(&the\_Lock, flags); //return to the previous state saved

//in flags

c: spinlock\_t the\_Lock= SPIN\_LOCK\_UNLOCK;

unsigned long flags;

spin\_lock\_irq(&the\_Lock); // does not know if the interrupts are already disabled

// critical section

spin\_unlock\_irq(&the\_Lock); // could result in an unwanted interrupt re-enable

d: spinlock\_t the\_Lock= SPIN\_LOCK\_UNLOCK;

unsigned long flags;

void spin\_lock\_bh(spinlock\_t \*lock)

// critical section

void spin\_unlock\_bh(spinlock\_t \*lock)\*\*\* **IRQ** disable only for current CPU, function return non-zero on success (lock obtained), 0 otherwise.

int spin\_is\_locked (spinlock\_t\* lock); //returns 1 if held, 0 if not, but beware of races!  **|**  void spin\_unlock\_wait (spinlock\_t\* lock);)

int spin\_trylock (spinlock\_t\* lock); //make one attempt; returns 1 on success, 0 on failure  **|**  //wait until available (race condition again!

**4 functions that can lock a spinlock:**

**a. void spin\_lock (spinlock\_t \*lock);** obtain the lock

**b. void spin\_lock\_irqsave (spinlock\_t \*lock, unsigned long flags);** disables interrupts (on the local processor only) before taking the spinlock; the previous interrupt state is stored in flags.

**c. void spin\_lock\_irq (spinlock\_t \*lock);** unconditionally disable interrupt and take lock

**d. void spin\_lock\_bh(spinlock\_t \*lock);** disables software interrupts before taking the lock, but leaves hardware interrupts enabled.

**CLI** (Clear Interrupt) and **SLI** (Set Interrupt) only clear interrupt flag on current processor, other processor that did not received interrupt request may still access the resource.

**Interrupt flag (IF):** is a flag bit in the CPU's FLAGS register, which determines whether or not the (CPU) will respond immediately to maskable hardware interrupts. If the flag is set to 1, maskable interrupts are enabled. If set to 0 such interrupts will be disabled until interrupts are enabled. The Interrupt flag does not affect the handling of non-maskable interrupts (NMIs) or software interrupts generated by the INT instruction.

**CPU communicate with device:** 1. Independent I/O (use instruction and I/O port address space)

2. memory-mapped I/O (use loads/stores and assigned memory address space to I/O)

**I/O instruction (16-bit and little endian):** IN port, dest.reg | OUT src.reg, port

**Big endian:** most significant value in the sequence is stored first, at the lowest storage address

**Little endian:** least significant value in the sequence is stored first

**Big endian:** 4F52 would be stored as 4F52. **Little endian:** 4F52 would be stored as 524F

EAX, EBX, ECX. EDX AX

|  |  |  |
| --- | --- | --- |
|  | AH | AL |

ESP, EBP, EDI, ESI SP

|  |  |  |
| --- | --- | --- |
|  | AH | AL |

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Generated By** | **Asynchronous** | **Unexpected** |
| Interrupts | external device | YES | YES |
| Exceptions | invalid opcode or operand | NO | YES |
| System | deliberate, via INT instruction | NO | NO |

**Save Caller-save register after interrupt request:** Because the computer will start executing the interrupt handler. The value in the registers should not be changed. Interrupt handler will only save callee saved register

**Reason using tasklet rather than hardware interrupt handler:** Because tasklet can store long functions which can reduce the delay when executing it. Software interrupt will not change the state of CPU but hardware interrupt does. Software interrupt should be processed after the hardware interrupt is done.

**User-level adv and dis-adv:** Advantage: If the code has bugs it may crash the kernel and then the user has to reinstall the VM. Disadvantage: The user-level cannot test all bugs which may still result kernel panic error.

**mp1\_ioctl\_kill:**

pushl %ebp

movl %esp, %ebp

movl mp1\_list\_head, %ecx

**LOOP:**

cmpl $0, %ecx

jz NOT\_FOUND

movl ON\_CHAR(%ecx), %eax

cmpl $0x2E, %eax

je found

movl NEXT(%ecx), %ecx

jmp LOOP

**FOUND:**

movl $0x78, OFF\_CHAR(%ecx)

movl $0, %eax

leave ret

**NOT\_FOUND:**

movl $-1, %eax

leave

Dispatcher should NOT modify stack

No need to initialize stack frame

Other functions don’t need to return to the dispatcher

**Cheat sheet author**

**Name: Jiamao Xu**

**NetID: jiamaox2**

**Dispatcher:**

movl 12(%esp), %edx

cmpl $0, %edx

jl bad\_op

cmpl $2, %edx

jg bad\_op

jmp \*jumptable(,%edx,4)

bad\_op:

movl $-1, %eax

ret

jumptable: . long func1, func2, func3

**NO LEAVE:** when the operation input is invalid, the function will attempt to leave (move ebp) in this dispatcher function. Since this is a dispatcher function, the stack isn't changed to include the old ebp. Including leave would mess up the stack (leave will pop ebp) causing the rest of the program to fail if ecx = 3, the program will move ebp which is not we want.

**Caller-saved**

EAX, ECX, EDX

**Callee-Saved**

EBX, ESI, EDI

Word=2 bytes.

Long=4 bytes.

EAX=4bytes. Char=1 byte.

Int = 4 bytes.

**Tasklet**: (RTC) real-time clock can generate interrupts at given frequency.

It is interrupt handlers’ way of deferring work

movl 8(%ebp), %eax # EAX <- M[EBP+8]

leal 8(%ebp), %eax # EAX <- EBP+8 =4 bytes

movw $1,next(%ebx) # M[EBX + next] <- 1

**C calling convention**

int binary\_search (int key, int\* array, int size);

old EBP

return address //ebp+4

int key // ebp+8

int\* array // ebp+12

int size. // ebp+16

pushl %ebp

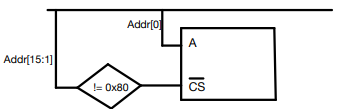
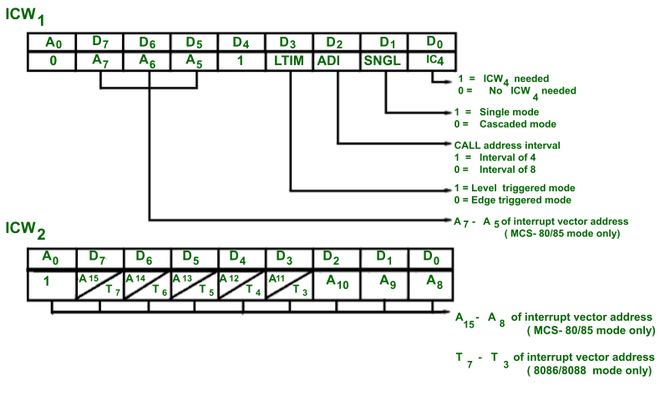
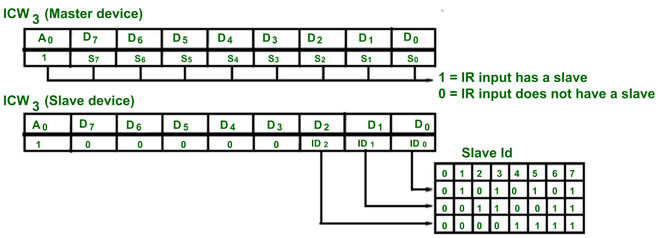
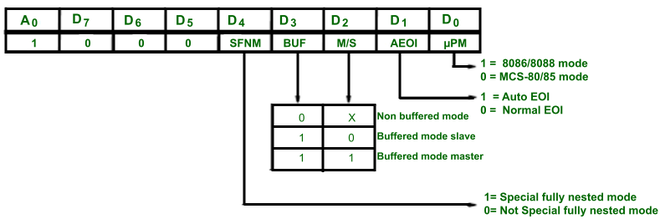
movl %ebp, %esp

pushl 3,2,1

call callee

addl $12, %esp //tear

popl 1,2,3 leave ret



YES

(IC4 =0)

Ready to accept IRQ

ICW4

NO (IC4 =0)

NO (single =1)

Is ICW4 needed

ICW3

YES

(Single =0)

In cascade mode

ICW2

ICW1

Four ICW word for primary PIC and Four ICW for secondary PIC

ICW1: A= 0 and D4 = 1, control bits for Edge and level triggering mode, single/cascaded mode, call address interval and whether ICW4 is required or not. Address lines A7 to A5 are used for interrupt vector addresses.

ICW2: A= 1, stores the information regarding the interrupt vector address.

ICW3: A=1, for primary PIC: bit vector of secondary PIC;

for secondary PIC: input pin on primary PIC

ICW4: A=1, ISA=x86, normal/auto EOI

primary PIC IR’s map #’s 0x20 – 0x27

secondary PIC IR’s map #’s 0x28 – 0x2F

**Let PIC occupy port 0x100 and 0x101**

0x100: 0000 0001 0000 0000 0x101: 0000 0001 0000 0001

First 15 bit = 000 0000 1000 0000 = 0x80

A(active high): port [0] CS (active low): port [15:1] != 0x80

**Reader/Writer Spinlock**

Multiple readers into a critical section simultaneously, only one writer

rwlock\_t my\_rwlock = RW\_LOCK\_UNLOCKED; /\* Static way \*/

rwlock\_t my\_rwlock;

rwlock\_init(&my\_rwlock); /\* Dynamic way \*/

lock method: lock, irqsave, irq, bh

**Barrier Synchronization**

extern static int NUM\_THREADS;

typedef struct { spinlock\_t lock;

thread\_count\_lock;

volatile int threads\_joined;

} barrier\_t;

void barrier\_init (barrier\_t \*b)

{ if (b == 0) // if pointer is zero

return; // return

b->the\_lock = SPIN\_LOCK\_UNLOCKED; // initialize the lock as unlocked

b->count= 0; //thread\_joined = 0

return; }

void barrier\_wait (barrier\_t \*b){ if (b==0) // if null pointer

return; // return

spin\_lock(&(b->the\_lock)); // lock the thread\_count\_lock

b->counter++; // increament thread\_joined

spin\_unlock(&(b->the\_lock)); // unlock the thread\_count\_lock

while(b->counter != NUM\_THREADS); // hold here

return;}

**CAS line:** The CAS line is used when implementing primary PIC and secondary PIC. Since only primary PIC can send interrupt requests to the processor, the secondary processor will never know when it should read/write data to the data bus. So primary PIC will tell secondary PIC to read/write data to data bus by sending signal on CAS bus. The signal is the IR port number secondary connected to which should be 0 to 7.

*divide\_by\_zero* exception handler:

The problem is the data vector transmitted through data bus was wrong. We should set up the correct data vector. ICW2 error OUTB(0x20, 0x21), so pic mapping to the wrong port.

CLI();

item\_to\_remove->prev->next = item\_to\_remove->next;

item\_to\_remove->next->prev = item\_to\_remove->prev;

STI();

