3. Shared data and resources

① Interrupt handler: preserve the contents of regs, avoid overwriting memory used by interrupted program. Interrupt should preserve all regs used!!!

② 问题点:

I. Shared linked list:

```
old_head = head;
head = new_elt;
interrupt!
new_elt->next = old_head
```

此时 head->next 为随机数

II. compiler optimizations

```
busy = 1;
while(busy);
infinite loop
```

用 volatile: volatile int busy = 1; this value may change at any time, so compiler reload it each loop and won't optimize it

不能全volatile: ① always reload var ② more memory ops ⇒ slower program

III. security(OS): ① keep user from crashing machine

② prevent information leaks from OS ⇒ 用 separate stack for OS(kernel)

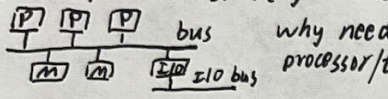
③ Critical section: 用改IF实现 should be short: ① avoid delaying service by interrupt handler ② long delays crash system

CLI: clear IF

code 🡒 atomically: the entire critical section has either been executed or none executed (no moving ops in/out critical section)

STI: set IF

4. Multiprocessor: SMP (symmetric multiprocessor) ⇒ all processors has same latency to all memory banks



why need lock: IF not cleared on other processor/take too much time clearing all processor

② spin_lock: lock "1", unlock "0", program 等锁释放不执行其他 busy waiting

locking must be atomic respect to other processors 用 interrupt handler ① mask intr ② restore IF's

```
pushl %ebx
movl 8(%ebp),%ebx #ebx<-&lock
movl $1, %eax #eax=1
xchg %eax,(%ebx)
popl %ebx
ret
```

```
spin_lock_irqsave(&lock,flags)
asm volatile("
    pushf
    popl %0
    CLI
    "=g"(flags)
    :"memory"
);
spin_lock(&lock)
```

先CLI再lock 防止deadlock

```
spin_unlock_irqrestore(&lock,flags)
```

先unlock再STI; "cc": condition codes change.

deadlock: a thread always hold the lock is waiting for an interrupt to finish. But the interrupt is trying to obtain the lock and fail. thus the thread block and lock always belong to this thread.

livelock: multiple threads all need to acquire some same locks but none of them get all locks. They will release the locks they acquire and try for new, but never get all locks never belong to one thread.

解决方法: 按同一顺序 acquire locks, 同一相反顺序 release locks.

③ Semaphore: 允许 fixed number 的 processor 访问同一个 critical section

P(down): test/decrement 尝试获取 V(up): increment 用完释放

当一个程序等待 semaphore 时, 即尝试时值为0, 休眠 sleep, 允许其他程序在 processor 上运行

thus must not be used by interrupt handlers

used for system calls, 尤其 long critical sections

同时用 semaphor 和 lock 时, 先 acquire semaphore 再 acquire spin lock

若先等待 spin lock, semaphor sleep 时其他程序也要此 lock ⇒ dead lock

5. reader/writer problem: 允许无数 reader, 但仅1 writer, 且在无 reader 时工作

① reader/writter ~~semaphores~~ spinlocks: 用于 short critical sections/intrupt handlers (admit writer starvation (一直有reader请求, 无法写)

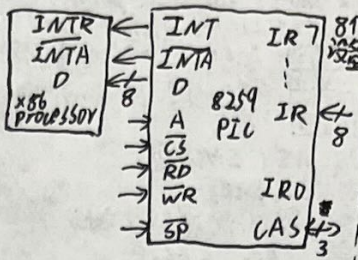
② reader/writter semaphores: 用于 system calls, 不 admit writer starvation, (writer请求时, new reader 排在 writer 后面)

why we need synchronization: ① protect shared resources ② prevent interrupts from corrupting data ③ race condition ④ critical section (run completion)

三、Programmable interrupt controller (PIC)

1. why not use OR gate: ① don't know which device write vector # ② processor need to query all devices, not all devices support query ③ priority,

2.



PIC作器件: interrupt 产生到 PIC

中看 PIC 是否在处理 interrupt, 比较 priority { <=: do nothing; >: 打断, mark 高优先级 as in service

执行完后: report EOI: end of interrupt, PIC把该 in service

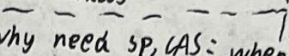
INT: 告诉CPU有 interruption 产生

INTA': CPU 告诉 PIC 已收到 int 并准备处理

D: data bus sent to CPU IOT

map to port 0x20, 0x21: 0x20 0010 000 0x21 0010 001

A: to differentiate between command and data port



CS': to tell PIC which CPU is RD/WR to it

RD': processor will read from PIC (use IN)

WR': processor will write to PIC (use OUT)

why need SP, CAS: when 支持多设备, cascaded PIC

① 对于 secondary PIC: don't know the priority of the interruption in primary PIC (higher?) ② 对于 primary PIC: don't know the high bits of vector numbers used by the secondary PIC nor which interrupt line of secondary PIC is being reported.

SP': Used to identify whether PIC is the primary or a secondary PIC

CAS': Used by the primary PIC to identify which secondary PIC should write to the data line

优先级: second PIC 连到 primary PIC 的 IR上, IR0 高优先

Priority: P0 > P1 > S0 > S1 > ... > S7 > P2 > P3 > ... > P7

四、MPI:

1. user-level test harness

① pros: I. faster development cycle II. protect your test machine from crashing ② cons: may not expose all bugs and may not work in kernel.

一、x86寄存器 32 bit, 4 byte, byte addressable, 1 byte = 8 bit

1. 寄存器：EAX, EBX, ECX, EDX, "E" extend ⇒ AX: 16 bit
AH: bit [15:8]?[7:4], AL: bit [3:0], 重写低位高位不变
ESI: source index (string copy), EDI: destination index
EBP: base pointer (base of stack frame), ESP: stack pointer
EIP: instruction pointer   EFLAGS: flags/condition codes
存储方式：little endian   加%表示寄存器
0x12345678 → 0x78, 0x56 … 0x12   便于直接访问low bits
位数小的在小内存

0x78	小地址
0x56	
0x34	↓
0x12	大地址

2. 操作：① arthmetic: ADD, SUB, NEG, INC, DEC
② logical: AND, OR, NOT, XOR 按位执行
③ shift: SHL 左移, SAR 算右移, 补符, SHR 逻右移, 补0
ROL 循环左移, ROR 循环右移
ORL %ECX, %EBX   # EBX ← EBX OR ECX
operation data type → second source → destination and first source
L: long 32 bit, W: word 16 bit, B: byte 8 bit
④ immediate number, 用$表示, 最大支持32 bits
hex: $0x__, Octal: $0__, decimal: $__
⑤ memory operand: displacement (SR1, SR2, scale)   default ↓ can be 1, 2, 4, 8
⇒ displacement + SR1 + SR2 * scale, SR2 不能是 ESP, 可以有1 reg (, %ecx, 4)
MOVL src, dst   # dst ← M[src], 最多支持1个 memory reference
LEAL src, dst   # dst ← src, src为 memory reference
LEAL LABEL+4, %ESI   # ESI ← LABEL + 4
⑥ conditional code: SF: sign flag 负设1, ZF: zero flag 零设1
CF: carry flag 首位carry/borrow设1, OF: overflow flag 看结果
PF: parity flag 偶数个设1
CMPL %EAX, %EBX   # flags ← EBX − EAX   } 仅改flags, 不改EAX和EBX
TESTL %EAX, %EBX   # flags ← EBX AND EAX
MOV, LEA, NOT 不改flags; ROL, ROR 改OF, CF; INC, DNC 不改CF
unsigned 比用 a, b, signed 比用 g, l   与符相反
cmpl %EAX, %EBX ; jb DONE # DONE if %EAX > %EBX
⑦ stack operation:
PUSHL %EAX # M[ESP-4] ← EAX, ESP ← ESP-4
POPL %EAX # EAX ← M[ESP], ESP ← ESP+4
LEAVE # movl %ebp, %esp ; popl %ebp
RET # EIP ← M[ESP], ESP ← ESP+4
⑧ data size convension: mov size from to
movsBL %AH, %ECX # ECX ← sign extend to 32 bit (AH)
movzBL %AL, %EAX # EAX ← zero extend to 32 bit (AL)
⑨ assemble conventions: label: 声明时加, 使用不加
.string NULL结尾 .byte .word .long .quad ← 64 bit float
⑩ multiplication and division (64 bit: EDX:EAX; 32 bit DX:AX; 16 bit AX)
MULL %EBX # unsigned EDX:EAX ← EAX * EDX 默认EAX
DIV %EBX # unsigned EAX ← EDX:EAX / EBX 商, EDX ← remainder
IMULL %ECX, %EBX # signed EBX ← EBX * ECX   ↳ quotient
IMULL $20, %EDX, %ECX # signed ECX ← 20 * EDX ↳ 高位舍去   IDIV signed
⑪ input, output: independent I/O   {all data to/from AL AX EAX; port是8 bit $ 或DX}
IN port, dest reg   OUT src reg, port
INB $0x40, %AL # AL ← P[0x40]   INW (%DX), %AX # {AL ← P[DX]; AH ← P[DX+1]}
OUTB %AL, (%DX) # P[DX] ← AL   OUTW %AX, 68 # {P[68] ← AL; P[69] ← AH}

3. Calling convention
① C函数参数从右往左压入栈
int fun(int key, int* array, int size)
Reason: ① allow for a variable number of parameter without requiring additional space for para counts or sentinels
② the argument's memory address is fixed for kernel, like 1st argument is at 8(%ebp)

ESP →	local vari
EBP →	old EBP
	return addr
	key
	array
	size

system call use reg instead of stack:
Pros: reg faster, easier for R/W, reduce memory space
cons: has number limit

② subroutine return value: 32 bit: EAX; 64 bit: EDX:EAX (高 低)
Caller saved reg: EAX, ECX, EDX, EFLAGS
Callee saved reg: ESP, EBP, EBX, ESI, EDI

[Caller sequence]
I. save caller-saved reg (push)
II. argument push入栈
III. call 子程序
IV. argument pop 出栈
V. restore caller-saved reg (pop)
如 push arg1
call func1
addl $4, %esp

[Callee sequence]
I. pushl %ebp; movl %esp, %ebp
II. save callee-saved reg. (push)
III. make space for local vari
subl $4, %esp
IV. 正常执行
V. tear down local vari
VI. restore callee-saved reg. (pop)
VII. LEAVE; RET
如 pushl %ebp
movl %esp, %ebp
subl $4, %esp
⋮
movl -4(%ebp), %eax
leave ← addl $4, %esp
ret

```
int dispatcher(unsigned int arg1, ... arg2, ... opera)
.data
jumptable:
.long func1, func2, func3
.text
dispatcher:
    movl 12(%esp), %eax
    cmpl $0, %eax
    jl   bad_op
    cmpl $2, %eax
    jg   bad_op
    jmp  *jumptable(, %eax, 4)
bad_op: movl $-1, %eax
    ret   # 不能LEAVE
```

4. 其他
① 指针, 结构体 [long min | long max] → mm, EBX 存&mm
movl %EDX, 0(%EBX) # mm.min ← EDX   注意每个field的数据长度
movl %EAX, 4(%EBX) # mm.max ← EAX
data type alignment:
struct pixel {
char red, green;
int alpha;
char blue; };
sizeof(Pixel) = 12 bytes
Char: 1 byte, int 4 byte
自动的最长对齐
memory addr 整除要求
int → multiple of 4

red	green		
alpha			
blue			

② 为什么移除arguments而非reuse: argument 可能会变

二、Interrupt和Synchronization
1. role of system software: ① virtualization the illusion of multiple / practically unlimited resources, 多个虚拟机在同一电脑执行
② protection: 减少accidential / malicious destruction of data by other program. ③ abstraction: hide fundamentally asynchronous nature of processor/device interaction; provide simpler, more powerful interfaces.
2. ① system call: INT <8-bit imm> # push EIP, EIP ← table[imm8]
call #1 → [0, 1, 2, 3] → 子目 handlers
→ jump table (vector table)
indirection:
① rewrite OS only need change table
② application code doesn't change
② exception: processor maps each problem to a vector # 在jumptable调用 (出错情况: 除0, 访问不存在内存/文件, 或权限不足)
③ interrupt: 打断正常程序执行, 处理特殊事件/硬件响应

type	generate by	asynchronous	unexpected
interrupt	external state	yes	yes
exception	invalid opcode/operand	No	yes
system call	deliberate, via INT	No	No

x86中, 均通过 Interrupt Descriptor Table 实现

IDT:

0x00 ~ 0x1F (32个) defined by Intel	exception
0x20 ~ 0x27 Primary 8259 PIC	IRQ0 ~ IRQ7
0x28 ~ 0x2F secondary 8259 PIC	IRQ8 ~ IRQ15
⋯	
0x80	system call

x86 ← INTR
← NMI ⇔ non maskable interrupt
↔ data bus 8

EFLAGS中 IF: interrupt enable flag   CLI STI
{set: 有 interrupt; clear: mask interrupt}

VGA reg
VGA video memory
Independent I/O: use distinct instructions
separate I/O ports from memory addr
memory-mapped I/O: No new Instructions
has addr set aside for I/O