	2's complement	unsigned
8 bits	`char`	`unsigned char`
16 bits	`short` `short int`	`unsigned short` `unsigned short int`
32 bits	`int`	`unsigned` `unsigned int`
32 or 64 bits	`long` `long int`	`unsigned long` `unsigned long int`
64 bits	`long long` `long long int`	`unsigned long long` `unsigned long long int`

• If a static variable is declared inside a compound statement, only one copy exists.
• The backslashes at the end of each line are necessary to tell the preprocessor that the macro definition continues on the next line.
volatile tells the compiler that the value stored in memory may change at any time

C Programming

x86 Assembly

`jo`	overflow	OF is set
`jp`	parity	PF is set (even parity)
`js`	sign	SF is set (negative)
`je`	equal	ZF is set
`jb`	below	CF is set
`jbe`	below or equal	CF or ZF is set
`jl`	less	SF ≠ OF
`jle`	less or equal	(SF ≠ OF) or ZF is set

BASIC

• x86 is little-endian
• NEG (negate) INC (increment) DEC (decrement) SHL (left) SAR (arithmetic right) SHR (logical right) shift with wraparound, to the left (ROL) or to the right (ROR)
• Long 32-bit Word 16-bit Byte 8-bit
• The x86 destination register appears as the last operand rather than the first
• Immediate values of up to 32-bits
• Values fit into fewer bits as shorter instructions
• Numbers starting with 1~9 are decimal
• Numbers starting with 0x are hexadecimal
• Numbers starting with 0 are octal values
• Be careful to include a "$" when you want a number interpreted as an immediate value!

CONDITION

• Most instructions affect all flags.
• MOV, LEA, NOT, affect no flags
• ROL, ROR, affect only OF and CF
• INC and DEC, which affect all but CF.
• The CMP instruction performs a subtraction
• The TEST instruction performs an AND operation

CODING

• Each label definition must be followed by a colon
• Labels can begin with letter, a period, an underscore
• CALL-RET
• Labels are case sensitive.
• The x86 assembler supports .GLOBAL and .EXTERN to declare symbols to be visible externally and to be defined externally
• .INCLUDE equivalent to #include
• MOVZBL zero extends a byte into a long
• Special forms are available for EAX: CBTW converts signed byte AL to word AX, CWTL converts signed word AX to long word EAX, and CLTD converts signed long word EAX to double word EDX:EAX

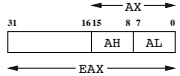
CALLING CONVENTION

• Parameters are pushed from right to left
• For pointers and integers no more than 32 bits, the return value is placed in EAX.
• Integers and other non-floating-point types of more than 32 but no more than 64 bits are split between EDX (high bits) and EAX (low bits)
• EBX, ESI, and EDI are callee-saved

	`jnz`	`jnae`	`jna`	`jz`	`jnb`	`jnbe`	unsigned comparisons
preferred form	`jne`	`jb`	`jbe`	`je`	`jae`	`ja`	
	≠	<	≤	=	≥	>	
preferred form	`jne`	`jl`	`jle`	`je`	`jge`	`jg`	signed comparisons
	`jnz`	`jnge`	`jng`	`jz`	`jnl`	`jnle`	

`%eax`	accumulator (for adding, multiplying, *etc.*)
`%ebx`	base (address of array in memory)
`%ecx`	count (of loop iterations)
`%edx`	data (*e.g.*, second operand for binary operations)
`%esi`	source index (for string copy or array access)
`%edi`	destination index (for string copy or array access)
`%ebp`	base pointer (base of current stack frame)
`%esp`	stack pointer (top of stack)
`%eip`	instruction pointer (program counter)
`%eflags`	flags (condition codes and other things)

32–bit	16–bit	8–bit high	8–bit low
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL
ESI	SI		
EDI	DI		
EBP	SP		
ESP	BP		




```
displacement(SR1,SR2,scale)
leal      (%eax,%ebx),%ecx     # ECX   EAX + EBX
movl      $10,%esi  # ESI   10
movl      %eax,%ecx            # ECX   EAX
xorl      %edx,%edx            # EDX   0
movb      -6(,%edx,2),%al      # AL   M[EDX * 2 - 6]
movb      label+10,%al         # AL   M[label+10]
movb      13+8 * 8-35+label(%edx),%al
          # AL   M[EDX + label + 42]
movl      $label+10,%eax       # EAX   label+10
cmp       %ebx,%esi
          # set flags based on (ESI - EBX)
call      printf
          # (push EIP), EIP   printf
call      *%eax
          # (push EIP), EIP   EAX
inw       (%dx),%ax
          # AL   P[DX], AH   P[DX + 1]
outw      %ax,68
          # P[68]   AL, P[69]   AH
jmp       *operations-4(, %ecx, 4)
------------------------------
pushl     %ebp                 # save old frame pointer
movl      %esp,%ebp            # point to new frame
...
leave     # ESP   EBP + 4, EBP   M[EBP]
ret       # return
------------------------------
.align    4
          # the label my array is a multiple of 4
my array:
          .long 100000,4000000000,24,0

leal LABEL, %edx == movl $LABEL, %edx
```

ABBREVIATION

RTL,register transfer language
SMP, symmetric multiprocessor
APICs, advanced PICs
IDT, Interrupt Descriptor Table
EOI, end-of-interrupt

	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`

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`

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 and Synchronization

SYSTEM CALLS, INTERRUPTS, EXCEPTIONS

• **System calls** are initiated with an INT or TRAP
• A system call places the processor in privileged/kernel
• **Interrupt** is reserved for asynchronous interruptions generated by other devices
• **Exceptions** occur when a processor encounters an unexpected opcode or operand
• 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)
• The x86 ISA uses a single common table

INTERRUPT

• The x86's INTR input indicates that device has requested an interrupt.
• Interrupts are prioritized and masked according to priority by an external interrupt controller
• If the interrupt enable flag (IF) is set, interrupts are allowed. If it is clear, interrupts are masked regardless of their priority.
• The STI and CLI instructions change the value of IF.
• The x86 uses a single vector table IDT for interrupts, exceptions, and system calls.
• Each of item in IDT with an IRET instruction.
• Interrupt handlers must preserve the contents of registers and must avoid overwriting memory locations used by the interrupted program.
• Interrupts must preserve all registers used.
• EFLAGS register can be pushed and popped, but is also usually saved and restored automatically.
• A separate stack register is used for the operating system when the processor switches from unprivileged to privileged mode, the stacks are swapped.

I/O

• The handler code that executes on behalf of a device interacts with the device using through the processor's input/output
• A processor is designed with an I/O port space similar to a memory address space.
• Each device is associated with some or set of ports.
• **Independent I/O system** separates I/O ports from memory addresses by using distinct instructions for each class of operation.
• **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 can not be accessed during normal processor operations.

MULTIPROCESSORS AND LOCKS

• SMP: the access time from any processor to any uncached memory location is identical.
• 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 statically allocated spin lock can be initialized statically. 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!
• 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.
• When more than one lock must be acquired, only the first lock needs to mask interrupts.

CRITICAL SECTION

• A critical section is a block of code that executes a set of operations that should be executed without interruption. The critical section occurs atomically with respect to the interrupt.
• On computers with only a single processor, the approach is simple enough: use interrupt masking at the boundaries of critical sections
• Make critical sections as short as possible.

SEMAPHORES

• A semaphore generalizes the concept of a lock to allow some fixed number of programs enter some set of critical sections simultaneously.
• When a program wants a keyboard, it executes a down operation
• Programs attempting to down the semaphore block until it available.
• When a program is ready to relinquish a keyboard, it executes an up.
• Semaphores differ from spin locks in that a program waiting on a semaphore allows other programs to execute while it waits
• Semaphores should not be used in code that shares data with interrupt handlers, nor should code wait on a semaphore while a spin lock is held.
• Semaphores can be used to protect longer critical sections.
• 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** is used to describe synchronization of this type.

READER/WRITER SPIN LOCKS

• Any number of readers can enter their critical sections simultaneously.
• Writer should only be allowed to access when no other are accessing.
• Readers can enter a critical section even if a writer is waiting.
• 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

• 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.
• Reader/writer semaphores do not admit starvation

INTERRUPT CONTROL (PIC)

• Processor has a single interrupt input pin, INTR; an interrupt acknowledgement output, INTA'; a data bus over which the interrupt vector must be delivered.
• 8259A operates asynchronously
• 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
• To report an interrupt, PIC raises INT and waits for processor.
• Processor strobes INTA to request PIC write interrupt vector to D
• The interrupt handler tells PIC that interrupt has been serviced by writing certain bits to the address A and data D inputs with an OUT instruction.
• On receiving this EOI, PIC removes it from in-service interrupts.
• 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 D or expects to read data from D.
• CAS bus is for the master transmits the identification number for one of up to eight possible slaves on the bus
• The master PIC is mapped at ports 0x20 and 0x21, and the slave PIC at ports 0xA0 and 0xA1
• The initialization sequence requires that four initialization control words (ICWs) be sent to the 8259A
• 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.

SOFT INTERRUPT: time tensive tasks, (tasklet), between program and hard interrupt

INTERRUPT CHAINNING: multiple handler can be triggered by one interrupt

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

```
            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 */
```

```
            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 */
```

```
/* 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);
```

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.

initialization

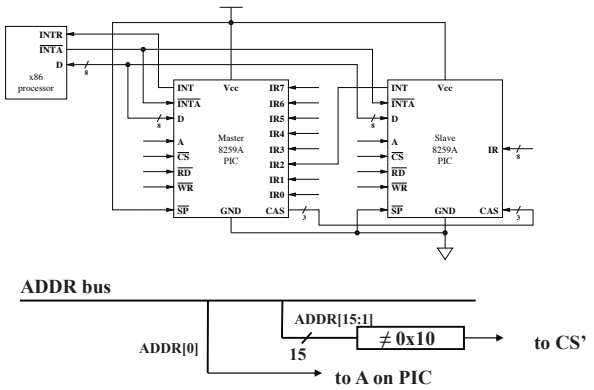
`void sema_init (struct semaphore* sem, int val);`	Initialize a dynamically allocated semaphore to a value.
`void init_MUTEX (struct semaphore* sem);`	Initialize a dynamically allocated semaphore to the value one.
`void init_MUTEX_LOCKED (struct semaphore* sem);`	Initialize a dynamically allocated semaphore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




```
void init_8259A(int auto_eoi)
{
        unsigned long flags;

        i8259A_auto_eoi = auto_eoi;

        spin_lock_irqsave(&i8259A_lock, flags);

        outb(0xff, 0x21);       /* mask all of 8259A-1 */
        outb(0xff, 0xA1);       /* mask all of 8259A-2 */

        /*
         * outb_p - this has to work on a wide range of PC hardware.
         */
        outb_p(0x11, 0x20);     /* ICW1: select 8259A-1 init */
        outb_p(0x20 + 0, 0x21); /* ICW2: 8259A-1 IR0-7 mapped to 0x20-0x27 */
        outb_p(0x04, 0x21);     /* 8259A-1 (the master) has a slave on IR2 */
        if (auto_eoi)
                outb_p(0x03, 0x21);     /* master does Auto EOI */
        else
                outb_p(0x01, 0x21);     /* master expects normal EOI */

        outb_p(0x11, 0xA0);     /* ICW1: select 8259A-2 init */
        outb_p(0x20 + 8, 0xA1); /* ICW2: 8259A-2 IR0-7 mapped to 0x28-0x2f */
        outb_p(0x02, 0xA1);     /* 8259A-2 is a slave on master's IR2 */
        outb_p(0x01, 0xA1);     /* (slave's support for AEOI in flat mode
                                    is to be investigated) */

        if (auto_eoi)
                /*
                 * in AEOI mode we just have to mask the interrupt
                 * when acking.
                 */
                i8259A_irq_type.ack = disable_8259A_irq;
        else
                i8259A_irq_type.ack = mask_and_ack_8259A;

        udelay(100);            /* wait for 8259A to initialize */

        outb(cached_21, 0x21);  /* restore master IRQ mask */
        outb(cached_A1, 0xA1);  /* restore slave IRQ mask */

        spin_unlock_irqrestore(&i8259A_lock, flags);
}
```

```
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
```