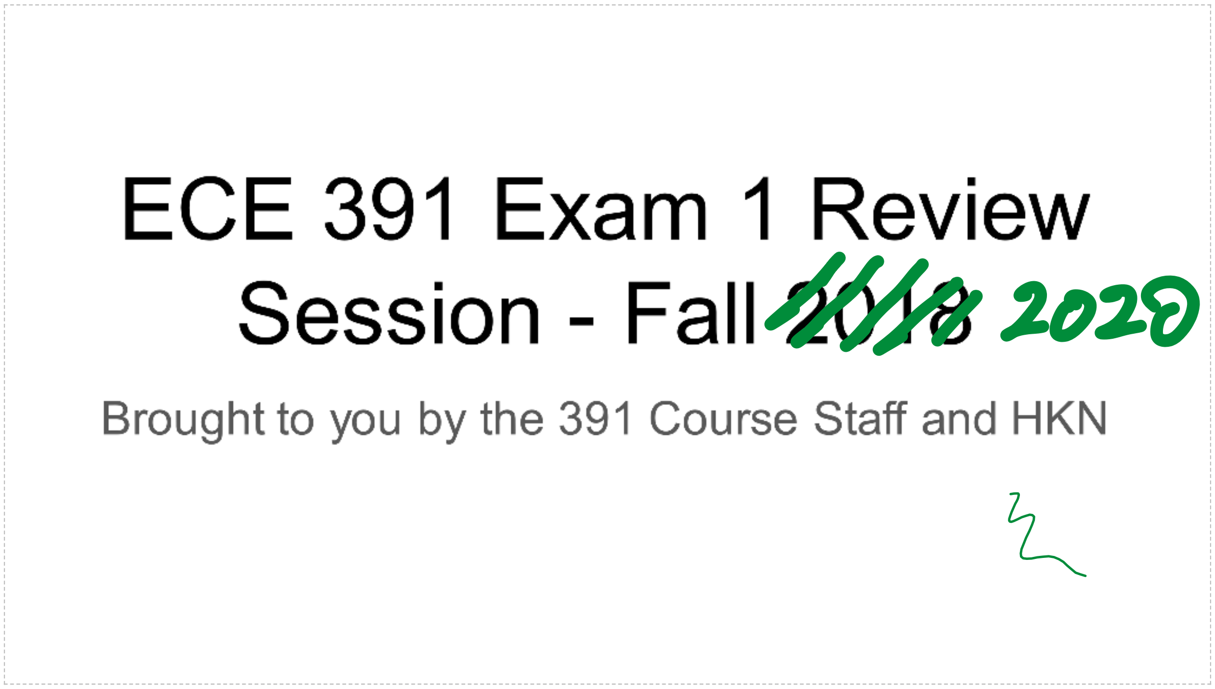
ECE 391 Midterm 1 Spring 2020

Saturday, February 22, 2020 1:31 PM

HKNECE...

HKNECE...



ECE 391 Exam 1 Review Session - Fall ~~2018~~ 2020

Brought to you by the 391 Course Staff and HKN

Slides will be posted on the HKN website

https://hkn.illinois.edu/service/ ← Find the slides here!

Keep in mind - TA's can help too
https://courses.grainger.illinois.edu/ece391/sp2020/ <- Office hours schedule and practice exams here

DISCLAIMER

There is A LOT (like a LOT) of information that can be tested for on the exam, and by the nature of the course you never really know what you'll be tested on. We're basing this review session to help you guys with the material that will most likely be on the exam, but there is a large possibility that there you will be tested on material we do not cover. Please be advised that you should still go over material on your own, and go to office hours to get TAs to help you.

x86 - Brief Overview (Reference sheet is included on the exam)

- `$LABEL` - literal value, `LABEL` - memory address
  - `leal LABEL, %edx` == `movl $LABEL, %edx`
- Can't have more than one memory access per instruction not allowed
  - e.g. `movl (%eax), (%ebx)`
- Memory is stored little-endian
- Comparisons end gets smallest address
  - signed: `jl` (lower), `jg` (greater)
  - unsigned: `jb` (below), `ja` (above)
    - `cmpl %esi, %edi; jge LABEL`
    - Performs the (signed) comparison `%edi` ≥ `%esi`

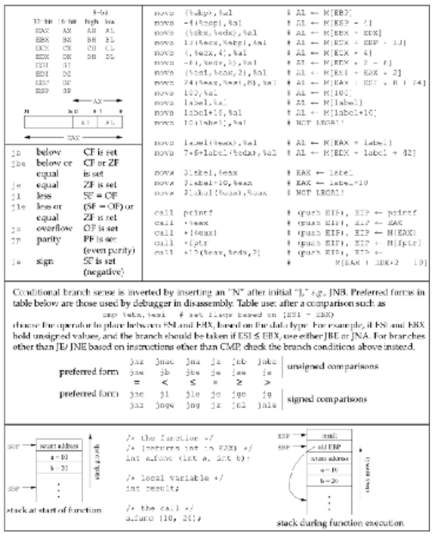
int a = x01234567

larger address

x 67
x 45
x 23
x 01

movl %eax, %edx // edx = eax
movl (%eax), %edx // edx = M[eax]

memory address dereferencing



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	BP		
ESP	SP		

AX
31
8 7
0
AH
AL
EAX

```
movb  (%ebp),%al               # AL ← M[EBP]
movb  -4(%esp),%al             # AL ← M[ESP - 4]
movb  (%ebx,%edx),%al          # AL ← M[EBX + EDX]
movb  13(%ecx,%ebp),%al        # AL ← M[ECX + EBP + 13]
movb  (,%ecx,4),%al            # AL ← M[ECX * 4]
movb  -6(,%edx,2),%al          # AL ← M[EDX * 2 - 6]
movb  (%esi,%eax,2),%al        # AL ← M[ESI + EAX * 2]
movb  24(%eax,%esi,8),%al      # AL ← M[EAX + ESI * 8 + 24]
movb  100,%al                  # AL ← M[100]
movb  label,%al                # AL ← M[label]
movb  label+10,%al             # AL ← M[label+10]
movb  10(label),%al            # NOT LEGAL!

movb  label(%eax),%al          # AL ← M[EAX + label]
movb  7*6+label(%edx),%al      # AL ← M[EDX + label + 42]

movw  $label,%eax              # EAX ← label
movw  $label+10,%eax           # EAX ← label+10
movw  $label(%eax),%eax        # NOT LEGAL!

call  printf                   # (push EIP), EIP ← printf
call  *%eax                    # (push EIP), EIP ← EAX
call  *(%eax)                  # (push EIP), EIP ← M[EAX]
call  *fptr                    # (push EIP), EIP ← M[fptr]
call  *10(%eax,%edx,2)         # (push EIP), EIP ←
                               #          M[EAX + EDX*2 + 10]
```

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

Conditional branch sense is inverted by inserting an "N" after initial "J," e.g., JNB. Preferred forms in table below are those used by debugger in disassembly. Table use: after a comparison such as

```
cmp %ebx,%esi   # set flags based on (ESI - EBX)
```

choose the operator to place between ESI and EBX, based on the data type. For example, if ESI and EBX hold unsigned values, and the branch should be taken if ESI ≤ EBX, use either JBE or JNA. For branches other than JE/JNE based on instructions other than CMP, check the branch conditions above instead.

	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	

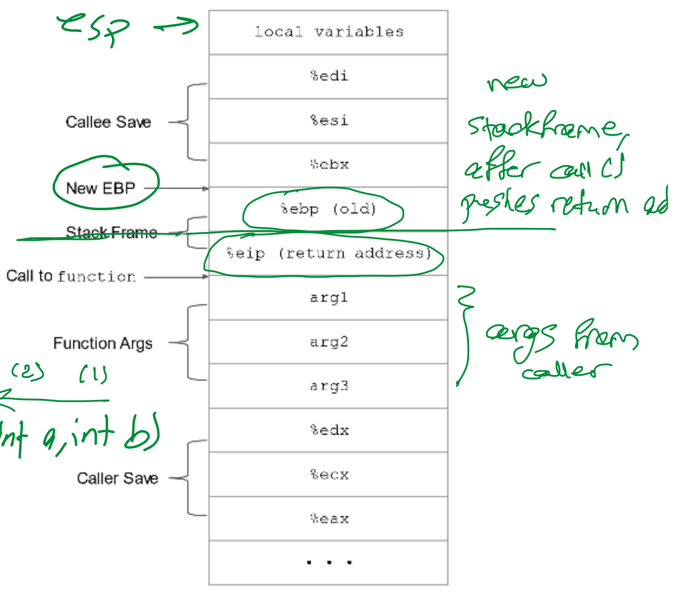
```
/* the function */
/* (returns int in EAX) */
int a_func (int a, int b);

/* local variable */
int result;

/* the call */
a_func (10, 20);
```

x86/C Calling Conventions

- Caller save registers - `EAX, ECX, EDX`
- Callee save registers - `EBX, ESI, EDI`
- 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`
- Push arguments from right to left

*We'll go over a translation question later

`function(arg1, arg2, arg3):`

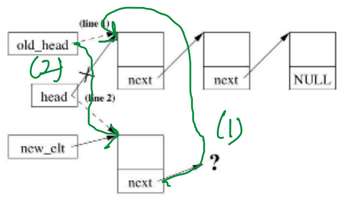


int func(int a, int b)
pushl b
push a

Synchronization Part 1

- Sharing data structures between program and interrupt handlers
  - Linked list example

```
/* line 1 */ old_head = head;
/* line 2 */ head = new_elt;
             /* INTERRUPT OCCURS HERE! */
/* line 3 */ new_elt->next = old_head;
```



don't break the linked list
if interrupts or other threads
might need it
(or protect using lock)

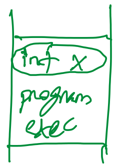
Synchronization Part Dos

```
volatile int ready = 0;
while(!ready);
```

makes processor
have to check
variable every time
(processor might not be
aware of interrupts using this bit,
so might get optimized by the compiler

Synchronization Part III

- Critical sections → code that runs without being interrupted
  - For single processor machines, use the interrupt flag to accomplish this (drawbacks to using this)
- For multiprocessors, we need more:



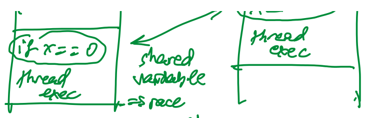
interrupt uses x
⇒ mask interrupts

spin_lock() : locks the lock but doesn't
disable interrupts

- Introducing...the SPIN LOCK
  - spin_lock_irq() ~~spin_lock~~ → doesn't push the flags to the stack before entering critical section
  - spin_lock_irqsave→ pushes the flags before entering critical section
  - Both however clear the IF flag (why?)

In case interrupt uses the protected

In case interrupt uses the protected variable, we never want race conditions or deadlocks



⇒ race condition

⇒ Use locks

Synchronization Part IV

- Semaphores expensive to lock, but will save processing cycles long-term
  - Up/Down operations
  - Process goes to sleep giving other processes access to the CPU
  - Can be used to protect longer critical sections
- Mutex standard semaphore implementation
  - Similar to a semaphore, except only the thread that locked it can unlock
- Reader/Writer Spinlocks
  - Can cause writer starvation always prioritize readers, too many readers = no time for writes
- Reader/Writer Semaphores
  - Helps prevent starvation has mechanisms to detect starvation

spinlocks ← cheap to lock but still hog processor cycles

Interrupts, Exceptions, System Calls, and Tables!

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

These guys interrupt the regular flow of a program, and are used to:

- Deal with something that requires urgent attention now (interrupt). This can usually be masked if we don't wanna (or can't) deal with that stuff. fired by device
- Tell us what to do when unexpected bad stuff happens (exception) last ditch efforts to save processor in case of failure
- Let the kernel, higher-privileged code, execute some instruction or carry out some task for us that we can't do ourselves (system call/trap) requested by user for privileged operations (eg. opening a device file descriptor)

Every time we get an interrupt, exception, or system call, we jump to a 256-entry vector table called the Interrupt Descriptor Table (IDT) to handle them. You can find the table in lecture slides/notes.

- Entries in the table from 0x00-0x1F are reserved and defined by Intel, more later in course
- Single entry 0x80 for all system calls. Privilege and stuff matters here, more later in course

specific syscall chosen differently (through register)

Programmable Interrupt Controller

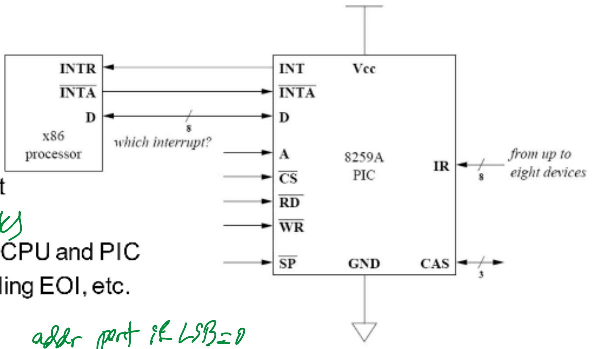
Useful for handling multiple interrupts from different devices, can't just stick all the interrupts onto a single bus and expect that to work. The PIC allows us to prioritize, mask, and individually address different interrupts that get raised.

- Each PIC handles 8 interrupts, but they can be configured in a master-slave configuration (with one master, and up to 8 other slaves) to handle up to 64 different interrupts. Next slides will go more in-depth 8 slave PICs (one on each line) = 8×8 = 64 lines
- After the processor (our point of view) receives an interrupt from the PIC, it calls ack(nowledge)

function to acknowledge receipt and masks all lower-priority interrupts, immediately sends an End-Of-Interrupt (EOI), then calls end function when done handling the interrupt to unmask lower-priority interrupts. must send EOI to avoid permanent masking of lower priority interrupts

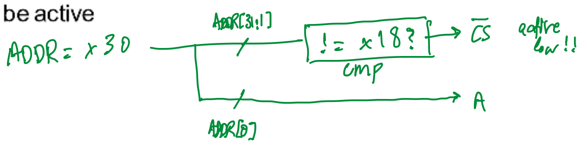
- This lets interrupts continue to build up in a queue so we can handle them later.

PIC Functionality

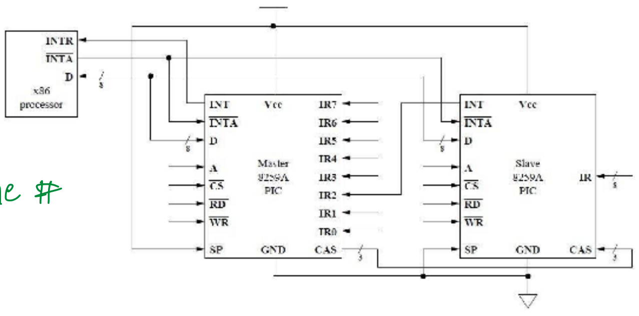


- Memory mapped to two ports
  - Command port (e.g. 0x20) x21
  - Data port (MUST be Command Port + 1)
- CPU - PIC Signals
  - INTR - Activated by the PIC upon interrupt
  - INTA - Pulsed by the CPU whenever (ack)
  - D - Bidirectional communication between CPU and PIC
    - Used in programming the PIC, sending EOI, etc.
- PIC Specific Signals
  - A - Distinguishes Command/Data port on PIC addr port if LSB=0 data port if LSB=1
    - 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]
- Priority: IR0, IR1, IR2, … , IR7

x30 = 0011 0000



PIC Cascading (i.e. Master/Slave configuration)



- Two Level Hierarchy
- SP - Decides whether a PIC is master or slave
  - 1 = Master, 0 = Slave
- CAS - How master PIC decides the slave CAS = slave PIC irq line #
  - Width is determined by number of slaves the PIC can support (basically the number of IR lines)
- Given this info, why must it be a two-level hierarchy?
- If a slave is attached, put all its priorities on the level where it's connected
  - e.g. slave on IR3, the total hierarchy is M0, M1, M2, $\underbrace{\textbf{S0, S1, S2, … , S7}}$, M4, … , M7

replace M2, higher priority than IRQs M4-7

PIC Initialization (1/2)

5 Key steps

Lock and save flags (context) so you can initialize properly

cannot receive interrupt before PIC is initialized

Mask interrupts to the PIC so you don't get disturbed while initializing

Initialize PIC

Unmask interrupts

Unlock and restore flags

PIC Initialization (2/2)

How to actually initialize PIC? All you need to do is send 4 control words!

But what do they mean?

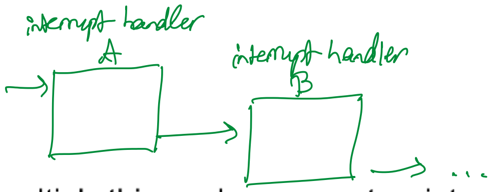
CONTROL WORD 1: Put PIC in initialization MODE (after this it expects the next 3 control words to come to it on a particular port)

CONTROL WORD 2: Tell PIC the start of IDT mapping

CONTROL WORD 3: Master: bitmap of slaves; Slave: input pin on master

CONTROL WORD 4: Some EOI stuff

More About Interrupts!



Interrupt Chaining:

- Don't you wish you could handle multiple things when you get an interrupt, well now you can!
    - With interrupt chaining, multiple handlers can be triggered by one interrupt
    - Several ways to do this, but generally doesn't happen to often in practice.

Soft Interrupts (tasklets):

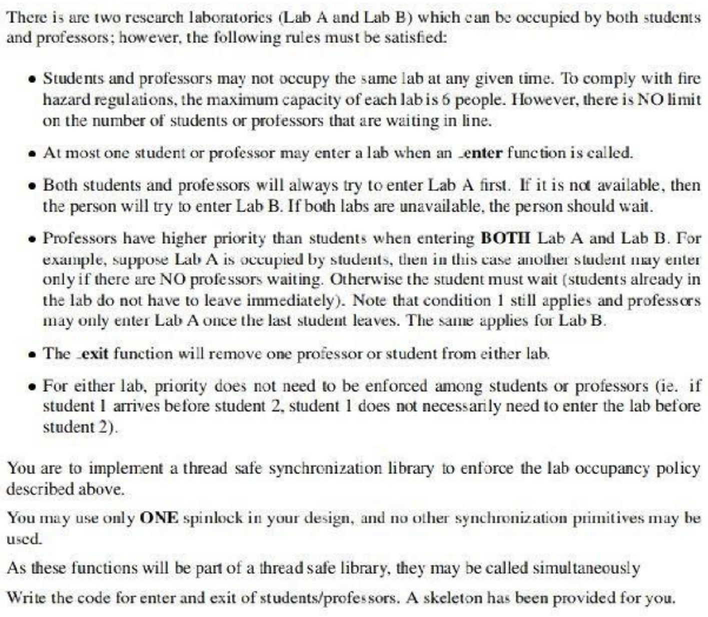
- It's not good to take too much time in a hardware interrupt handler- other interrupts may need to do things too! That's what software interrupts are for
    - Software interrupts operate at priority level between regular programs and hardware interrupts, so hardware interrupts can generate a software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

      required but low priority tasks can be scheduled for later

Anything else??

Example Problem 1

- This was a PS2 from a past exam



There is are two research laboratories (Lab A and Lab B) which can be occupied by both students and professors; however, the following rules must be satisfied:

- Students and professors may not occupy the same lab at any given time. To comply with fire hazard regulations, the maximum capacity of each lab is 6 people. However, there is NO limit on the number of students or professors that are waiting in line.
- At most one student or professor may enter a lab when an _**enter** function is called.
- Both students and professors will always try to enter Lab A first. If it is not available, then the person will try to enter Lab B. If both labs are unavailable, the person should wait.
- Professors have higher priority than students when entering **BOTH** Lab A and Lab B. For example, suppose Lab A is occupied by students, then in this case another student may enter only if there are NO professors waiting. Otherwise the student must wait (students already in the lab do not have to leave immediately). Note that condition 1 still applies and professors may only enter Lab A once the last student leaves. The same applies for Lab B.
- The _**exit** function will remove one professor or student from either lab.
- For either lab, priority does not need to be enforced among students or professors (ie. if student 1 arrives before student 2, student 1 does not necessarily need to enter the lab before student 2).

You are to implement a thread safe synchronization library to enforce the lab occupancy policy described above.

You may use only **ONE** spinlock in your design, and no other synchronization primitives may be used.

As these functions will be part of a thread safe library, they may be called simultaneously

Write the code for enter and exit of students/professors. A skeleton has been provided for you.

Struct Definition

```
typedef struct ps_enter_exit_lock {
spinlock_t lock;   or spinlock_t * lock
  volatile unsigned int p_in_A;
  volatile unsigned int p_in_B;
  volatile unsigned int s_in_A;
  volatile unsigned int s_in_B;
  volatile unsigned int p_waiting;
  volatile unsigned int s_waiting;
} ps_lock;
```

If spinlock is defined as pointer, can be passed as ps→lock
otherwise, must be referenced as &(ps→lock)

⇒ lock() and unlock() expect a pointer/ address

Description:

- `lock` - spinlock used to synchronize accesses
- `p_in_A` - count of professors in lab A
- `p_in_B` - count of professors in lab B
- `s_in_A` - count of students in lab A
- `s_in_B` - count of students in lab B
- `p_waiting` - count of professors waiting in line
- `s_waiting` - count of students waiting in line

Note: Has to be volatile because of multithreading.

```
int professor_enter(ps_lock* ps) {
  bool in_wait_line = false;
  unsigned int flags;
  if(ps == NULL) return -1;
  while(1) {
    spin_lock_irqsave(ps->lock, flags);
    if (ps->s_in_A == 0 && ps->p_in_A < 6) {
      ps->p_in_A++;
      ps->p_waiting = ps->p_waiting==0 ? 0 : ps->p_waiting-1;
      spin_unlock_irqrestore(ps->lock, flags);
      break;
    } else if(ps->s_in_B == 0 && p_in_B < 6) {
      ps->p_in_B++;
      ps->p_waiting = ps->p_waiting==0 ? 0:ps->p_waiting-1;
      spin_unlock_irqrestore(ps->lock, flags);
      break;
    } else {
      if(!in_wait_line) {
        p_waiting ++;
        in_wait_line=true;
      }
    }
    spin_unlock_irqrestore(ps->lock, flags);
  }
  return 0;
}
```

Null Check

Grab the lock. This ensures only we are modifying the variables and no one else.

First, attempt to enter lab A. Check if there is room. If so, enter. must check bounds and students in lab

If no room in lab A, attempt to enter lab B. If there is room, enter. repeat for lab B

If the professor was not able to enter any of the labs, increment the wait counter full / occupied by students ⇒ make them wait

<u>Regardless of what happens, **UNLOCK THE LOCK AT THE BOTTOM OF THE LOOP**</u> safe option, make sure no function forgets to release lock

```
int professor_exit(ps_lock* ps) {
    unsigned int flags;
    if(ps == NULL) return -1;
    spin_lock_irqsave(ps->lock,flags);
    if(ps->p_in_A > 0){
        ps->p_in_A --;
    } else if (ps->p_in_B > 0) {
        ps->p_in_B--;
    } else {
        spin_unlock_irqrestore(ps->lock, flags);
        return -1;
    }
    spin_unlock_irqrestore(ps->lock, flags);
    return 0;
}
```

Null Check.

Grab Lock.

Try exit a professor from lab A

If not possible, try exit a professor from lab B

If still not possible, release the lock and give up

Release the lock and exit

Make sure # professors doesn't drop to -1, must consider the count bounds (0 and 6)

```
int student_enter(ps_lock* ps) {
    unsigned int flags;
    if(ps == NULL) return -1;
    bool in_wait_line = false;
    while(1){
        spin_lock_irqsave(ps->lock, flags);
        if (ps->p_in_A+ps->p_waiting==0 && ps->s_in_A<6) {
            ps->s_in_A++;
            ps->s_waiting = ps->s_waiting==0 ?
                0 : ps->s_waiting-1;
            spin_unlock_irqrestore(ps->lock, flags);
            break;
        } else if (ps->p_in_B+ps->p_waiting==0 && s_in_B<6) {
            ps->s_in_B++;
            ps->s_waiting = ps->s_waiting==0 ?
                0:ps->s_waiting-1;
            spin_unlock_irqrestore(ps->lock, flags);
            break;
        } else {
            if(!in_wait_line) {
                s_waiting ++;
                in_wait_line=true;
```

Almost identical to `professor_enter`

Additional check: do not enter the lab when there are professors waiting

extra condition: must yield lab priority to professors

⇒ check if no professors waiting before entering as student

```
        }
        spin_unlock_irqrestore(ps->lock, flags);
    }
}
```

```
int student_exit(ps_lock * ps) {
    unsigned int flags;
    if(ps == NULL) return -1;
    spin_lock_irqsave(ps->lock, flags);
    if (ps->s_in_A > 0) {
        ps->s_in_A--;
    } else if (ps->s_in_B >0) {
        ps->s_in_B--;
    } else {
        spin_unlock_irqrestore(ps->lock, flags);
        return -1;
    }
    spin_unlock_irqrestore(ps->lock, flags);
    return 0;
}
```

Identical to `professor_exit`

Example Problem 2: C to x86

~~All Together Now:~~

```
int calculate(uint32_t operation, int arg1, int arg2) {
    int retVal;
    switch (operation) {
        case 0:
            retVal = arg1 * arg2;
            break;
        case 1:
            if(arg2 == 0) { retVal = -1; }
            else{ retVal = arg1 / arg2; }
            break;
        case 2:
            retVal = 0;
            while(arg1 > arg2) {
                retVal += arg1;
                arg1 -= 1;
            }
            break;
        default:
            retVal = -1;
            break;
    }
    return retVal;
}
```

Converting x86 to C can seem daunting given how verbose x86 is, but as long as you break down the code into smaller sections and convert the sections one at a time it's not that bad!

Exam

```
.GLOBAL calcul
calculate:
    pushl %ebp
    movl %esp,
    pushl %ebx
    pushl %esi
    pushl %edi
    movl 8(%eb
    movl 12(%e
    movl 16(%e
    cmpl $2, %
    ja default
    jmp *jumpt
op1:
    movl %ebx,
    imull %esi
    jmp done
op2:
    cmpl $0, %
    je default
    movl $0, %
    movl %ebx,
    idivl %esi
    jmp done
[...]
```

```
.GLOBAL calculate
calculate:
    pushl %ebp
    movl %esp, %ebp
    pushl %ebx
    pushl %esi
    pushl %edi
    movl 8(%ebp), %ecx
    movl 12(%ebp), %ebx
    movl 16(%ebp), %esi
    // [...]
    popl %edi
    popl %esi
    popl %ebx
    leave
    ret
```

```
int calculate(uint32_t operation, int arg1, int arg2) {
    int retVal;
    // [...]
    return retVal;
}
```

Setup stack frame
Save callee-save registers
Load parameters into registers
(ecx <-- operation)
(ebx <-- arg1)
(esi <-- arg2)
// [...]
Load callee-save registers
Teardown stack frame

jump table index

```
cmpl $2, %ecx
    ja default
    jmp *jumptable(,%ecx, 4)
op1:
    // [...]
```

```
switch (operation) {
    case 0:
        // [...]
        break;
```

smart way to implement cascaded

The jumptable represents a case and switch construct

```
    jmp done
op2:
    // [...]
    jmp done
op3:
    // [...]
    jmp done
default:
    movl $-1, %eax
done:
// [...]
jumptable: .long op1,op2,op3
```

```
    case 1:
        // [...]
        break;
    case 2:
        // [...]
        break;
    default:
        retVal = -1;
        break;
}
```

if-else statements

⇒ Instead, make an array of function pointers and jump to the appropriate function address

ebx ← arg1, esi ← arg2

```
op1:
    movl %ebx, %eax
    imull %esi, %eax
    jmp done
op2:
    cmpl $0, %esi
    je default
    movl $0, %edx
    movl %ebx, %eax
    idivl %esi
    jmp done
op3:
    movl $0, %eax
lp:
    cmpl %esi, %ebx
    jle done
    addl %ebx, %eax
    subl $-1, %ebx
    jmp lp
```

```
case 0:
    retVal = arg1 * arg2;
    break;
case 1:
    if(arg2 == 0) { retVal = -1; }
    else{ retVal = arg1 / arg2; }
    break;
case 2:
    retVal = 0;
    while(arg1 > arg2) {
        retVal += arg1;
        arg1 -= 1;
    }
    break;
```

eax ← -1
return

Fairly self explanatory, just determine the higher level operation being performed in the assembly code and find the C equivalent

All Together Now:

```
int calculate(uint32_t operation, int arg1, int arg2) {
    int retVal;
    switch (operation) {      // jump table / if else (s)
        case 0:
            retVal = arg1 * arg2;     // mul operation
            break;
        case 1:
            if(arg2 == 0) { retVal = -1; }     // cmpl and jmp if true
            else{ retVal = arg1 / arg2; }      // divide operation
            break;
        case 2:
            retVal = 0;
            while(arg1 > arg2) {
                retVal += arg1;        // loop with jmp and cmpl to check while condition
                arg1 -= 1;
            }
            break;
        default:
            retVal = -1;      return -1
            break;
    }
    return retVal;
}
```

Converting x86 to C can seem daunting given how verbose x86 is, but as long as you break down the code into smaller sections and convert the sections one at a time it's not that bad!

Exam Questions!

- What questions do YOU have?