ECE 391 Exam 1 Review Session - Fall 2020

Brought to you by the 391 Course Staff and HKN

Slides will be posted on the HKN website

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

Keep in mind - TAs can help too
https://course.courses.illinois.edu/ece391/sp2020/ Office hours schedule and practice exams here
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
  - local LABEL, %edx — movl $LABEL, %edx
- Can’t have more than one memory access per instruction
  - e.g. movl (%eax), (%ebx)
- Memory is stored little-endian
- Comparisons
  - signed: jl (lower), jg (greater)
  - unsigned: jb (below), ja (above)

\[
\begin{align*}
&\text{cmp\ %esi, \%edi, jge LABEL} \\
&\text{movl \%eax, \%edx \# edx = eax} \\
&\text{movl (%eax), \%edx \# edx = M[%eax]} \\
\end{align*}
\]

x86/C Calling Conventions

- **Caller save registers** - EAX, ECX, EDX
- **Callee save registers** - EBX, ESI, EDI
- call vs jump:
  - jump → jmp LABEL
  - call → pushl %ebp; jmp LABEL
- enter - pushl %ebp; movl %esp, %ebp
  - “creates” the stack frame
- leave - movl %ebp, %esp; popdl %ebp
  - “tears down” the stack frame
- ret - popdl %ebp
- Push arguments from right to left

*We’ll go over a translation question later*
Synchronization Part 1

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

```c
/* 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

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

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:

```
spin_lock();      // locks the held but doesn't disable interrupts
```

- Introducing...the SPIN LOCK

```
spin_lock().try();     // 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?)

https://onedrive.live.com/redir?resid=2D6B8B9DE1793C16%21111&authkey=%21ALN1xVhHlfe3U&page=View&wd=target%28Review Sessions.one%7C82ee... 3/11
Synchronization Part IV

Semaphores
- 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
- Spinlocks are cheap to lock but still hog processor cycles

Interrupts, Exceptions, System Calls, and Tables!

<table>
<thead>
<tr>
<th>type</th>
<th>generated by</th>
<th>example</th>
<th>asynchronous</th>
<th>unexpected</th>
</tr>
</thead>
<tbody>
<tr>
<td>interrupt</td>
<td></td>
<td>packet arrived at network card</td>
<td></td>
<td></td>
</tr>
<tr>
<td>exception</td>
<td></td>
<td>divide by zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>system call/</td>
<td></td>
<td>print character to console</td>
<td></td>
<td></td>
</tr>
<tr>
<td>trap</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
- Tell us what to do when unexpected bad stuff happens (exception)
- 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)

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

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
- 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 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., $0x20$)
  - Data port (MUST be Command Port + 1)
- CPU - PIC Signals
  - INTR - Activated by the PIC upon interrupt
  - INTA - Pulsed by the CPU whenever (ack)通信：
  - 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]
- Priority: IR0, IR1, IR2, ..., IR7

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 iry line $P$
    - Width 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, S0, S1, S2, ..., S7, M4, ..., M7
    - replace M2, higher priority
    - then I$R$0s $\geq$ I$R$4 - 7

PIC Initialization (1/2)

5 Key steps

Lock and save flags (context) so you can initialize properly
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 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 (i.e., 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;  // 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->p_in_A == 0 && ps->p_in_B < 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 && ps->p_in_A < 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) {
                ps->waiting ++;
                in_wait_line = true;
            }
        }
        spin_unlock_irqrestore(ps->lock, flags);
    }
    return 0;
}

int professor_exit(ps_lock* ps) {
    unsigned int flags;
    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;
}

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 > 0) {
            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->p_in_B > 0) {
            ps->p_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) {
                ps->s_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.
If no room in lab A, attempt to enter lab B. If there is room, enter.
If the professor was not able to enter any of the labs, increment the wait counter.

Regardless of what happens, **UNLOCK THE LOCK AT THE BOTTOM OF THE LOOP**. Make sure no function forgot to release lock.

**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 a professor doesn't drop to -1, must consider the count bounds 0 and 6)

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
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;
}

Example Problem 2: C to x86

All Together Now:

int calculate(int32_t operation, int arg1, int arg2) {
    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;
}

Exam

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!

Setup stack frame
Save callee-save registers
Load parameters into registers
\( (ecx \leftarrow \text{operation}) \)
\( (ebx \leftarrow \text{arg1}) \)
\( (esi \leftarrow \text{arg2}) \)
// ...

Jump table
Index

The jump table represents a case and switch construct

https://onedrive.live.com/redir?resid=2D6B8B9DE1793C16%21111&authkey=%21ALNt1xVhHlfo3U&page=View&wd=target%28Review%20Sessions%29%28one%29%7C82ee...
All Together Now:

```c
int calculate(x86 int32_t operation, int arg1, int arg2) {
    int retVal;
    switch (Operation) { // jump table if else is
        case 0: // operation
            retVal = arg1 * arg2;
            break;
        case 1: // operation
            if (arg2 == 0) { retVal = -1; }
            else { retVal = arg1 / arg2; }
            break;
        case 2: // operation
            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 Questions!

- What questions do YOU have?