ECE 391 Lecture Notes

ECE 220 Review

Topics to ignore (or not worry about too much):

- **Heap**
    - Not really talked about much, even when talking about dynamic memory allocation.
- **LC-3**
    - In LC-3, you only had a few things to worry about in terms of instructions and registers. Since they're going to cover x86, LC-3 knowledge helps but don't be stressed if you don't remember much.
- **Recursion**
    - You have a limited amount of stack space to work with, do not make unnecessary function calls with recursion here.
- **Trees**
    - Implementation knowledge isn't necessary, knowing that the Linux filesystem is based on a red-black tree (google it if you don't know) is good enough.

Topics to know:

- **Stack (main thing)**
    - Since this is very important to fully understand, will introduce this now and early in x86 terms (cause I forgot LC-3). // **TODO**
- **Static Keyword**
    - You'll encounter this a lot throughout, so let's look at some examples for it.

      ```
      static void helper(int arg1);
      ```

    - `helper` will only be defined in the file that you put it in. You should not need to include any static functions in your header file, just declare them at the top of the C file if you use it in a function defined before it.
    - static variables are similar enough:

```
static int global_in_file;
int func() {
      static int counter = 0;
      return counter++;
}
```

- `global_in_file` will be available for all functions in its specific C file to be used. `counter` though is only available to `func( )`. One thing to remember is that these variables are initialized once when the program starts, and keep their values throughout the lifespan of the program.
    - This means every time func( ) is called, counter increments by one.

- **Structs**
    - Example struct:

```
// Some people may complain about dummy_t not being dummy_s
typedef struct dummy {
      uint32_t addr;
      int* ptr;
} dummy_t;
```

    - In our simple struct, it only holds two elements. Struct members are **public** by default, meaning anyone with access to a certain struct can modify those members at will. There is no actual definition of **private** (in which only relevant class functions can modify them) as perfect encapsulation can be done by forward declaring structs and implementing them in the .c file (**Note: not relevant for the class but interesting either way)**.

```
*****example.h*****
// Forward declare struct example, then you can typedef
struct example;
typedef struct example example_t;
example_t* new_example();
void free_example(example_t* dummy);
```

```
*****example.c*****
// Simple implementation, no error handling done
#include <stdlib.h>
#include “example.h”

struct example {
      int a;
};

// You can use the typedef in the header here
example_t* new_example() {
      return malloc(sizeof(example_t));
}

void free_example(example_t* dummy) {
      free(dummy);
}


*****main.c*****
#include “example.h”
int main() {
      example_t* dummy = new_example();
      free_example(dummy);
      // int temp = dummy->a would cause a compiler error.
      // int temp = get_a(dummy) would work if implemented
      return 0;
}
```

The above should compile fine (except google docs changes quotation marks so 2 changes to the includes involving example.h have to be made if copy pasted) :(

```
gcc main.c example.c
```

You can’t access the struct’s properties unless you’re in example.c. No need for public or private if the members are all private, and now you can emulate C++ style objects using getters, setters,

constructors/destructors, etc.

- **Singly Linked Lists (Can skip if comfortable with them)**
  - Pretty important for MP1 (even if though it's in x86), and certain structs talked about later in the course (doubly-linked, circular doubly-linked) for other kernel related implementations.
  - Let's look at a simple node implementation and cover different cases:

```
/* Node struct just holding integers */
typedef struct node {
      int data;
      struct node *next;
} node_t;
```

  - Now what do you need to worry about?
    - The head of the list being null.
    - next being null.
    - How to traverse and update properly.
  - If the head is null, a simple check at the beginning is enough.
  - When traversing, if not found, return null and let error checking happen afterwards.
  - With that in mind, a simple traversing function can be made:

```
1     // Function just finds a specific node and inserts data
2     // right before it in the list
3     int insert(int node_data, int insert_data, node_t *head) {
4           if (!head) { return -1; }
5           // Assume the function finds prev node knowing data
6           node_t *prev = prev_node(node_data);
7           if (!prev) { return -1; }
8           node_t *new_node = (node_t*) malloc(sizeof(node_t));
9
10          // Update node data and insert into list
11          new_node->data = insert_data;
12          new_node->next = prev->next;
13          prev->next = new_node;
```

```
14          return 0;
15      }
```

- In a singly linked list, we don’t have access to the previous pointer, meaning we have to manually search through it.
  - In example function, we’re not given a pointer to the actual node we want to insert it before, instead passed in the data it holds.
  - The prev_node function then has to check whether it actually exists or not, keeping track of both the current node and previous one until found or at the end of the list.

1 - Class overview and big picture

Logistical lecture, quotes from past students, no material on class related topics. I thought class started at 2:30 and was 30 minutes late, good way to start the semester.

Discussion Week 1 - Overview of MPs and Environment

- Look at slides related to GDB, Git, Make, and other stuff. Here will talk about how to actually use QEMU (virtual machines you'll be using to test and debug MPs)
- After you have it all set up, either following MP0 directions or setting up on your own environment, focus on 3 shortcuts:
    - **test_nodebug** - Run your programs from here if you don't want to use GDB (useless when MP3 comes around, useful for MP2)
    - **test_debug** - When you're ready to debug your programs, use this VM in conjunction with **devel** to test.
    - **devel** - Launch your programs from here to debug. Start it up, go to your work directory with program you want to debug, and use gdb. For the simplest example, say you want to run a file called mp1 to debug. Following along:

      ```
      gdb mp1
      ...gdb stuff...
      (gdb) tar re 10.0.2.2:1234
      ...something pops up if test_debug is open…
      (gdb) start
      ```

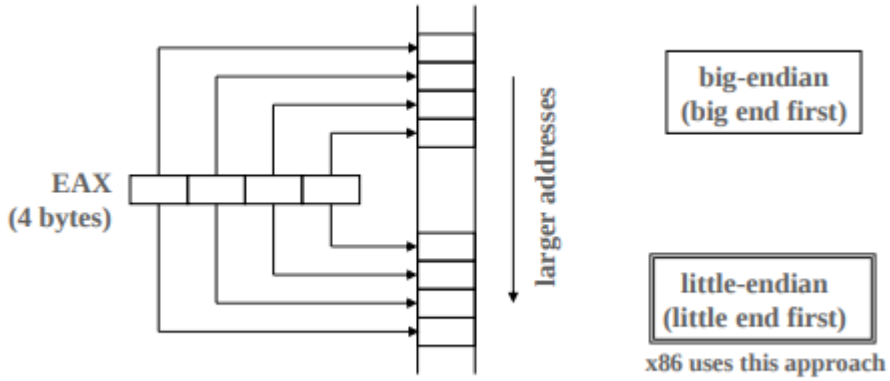
    - tar re is shortened for target remote, in which you use it to access your test_debug VM (saves a lot of time in the long run, especially for MP3).
    - When you type tar re 10.0.2.2:1234, you can also click on VM, click and hold your cursor until it's highlighted like this: tar re 10.0.2.2:1234, right click, and it should copy to clipboard and paste a copy onto the screen (should be fine to press enter here). Can right click to paste command now to save time until **devel** closed/bug happens.

2 - x86 instruction set architecture: introduction and instructions

- The six main registers you'll be using to manipulate data are **EAX, EBX, ECX, EDX, EDI, ESI,** which you can use just like LC-3 registers**.**

```
/* General syntax example */
movl    $4, %eax            # moves 4 into eax
addl    $2, %eax            # adds 2 to eax
movl    %eax, %esi          # moves eax's value into esi
movl    (, %ebx, 1), %edi   # ebx dereferenced, store in edi
```

- Other registers are **ESP, EBP, EIP, EFLAGS, CS, DS, ES, SS, FS, GS**
  - **ESP, EBP** are the stack pointer and base pointer respectively.
  - **EIP** is the instruction pointer, holding the current instruction address.
  - **EFLAGS** holds all of the flag stuff (eg comparing values).
  - **CS (Code Segment), DS (Data Segment), ES (Extra Segment), SS (Stack Segment), FS, GS**. Will be important in MP3 but not now.
- The 'X' registers (like **EAX**) can be specified to access certain bits (**AX** is lower 16 bits, **AH** and **AL** hold upper and lower 8 bits of **AX** respecitively)
  - For the other 'X' registers, similar style (**BX, BH, BL** for example)
- Memory in x86 uses little-endian style (look at picture to make sense of it)



- Example x86 code

```
movl    $321, %eax
```

```
pushw   %ax
movb    %ah, 0xECE391(,1)
```

- `$321` - Signifies a constant (321) that can be up to 32 bits
- `%eax` - Use % to access a register
- `movl` - Moves the first argument into the second argument. The **l** at the end signifies the type of data you're working with. Words from the professor: **Always include the data type**. The possible data types are **b (8 bits), w (16 bits), l (32 bits)**.
- `pushw` - Pushes onto the stack 16 bits/2 bytes (in this case, **AX**). It's the equivalent of doing

```
subl    $2, %esp
movw    %ax, %esp
```

- `movb %ah, 0xECE391(,1)` - Moving the 8 bits in **AH** into memory address **0xECE391** (will most likely crash or give an error without knowing specific details).

- **Important instructions to become acquainted with**
  - **CALL** - calls a function. **EIP** is pushed onto the stack to return to original function when the called function ends.

```
call    some_function
```

  - **RET** - Returns from a function, popping return address from the stack.

```
...end of function reached...
ret
```

  - **LEAVE** - Don't use all the time, an initial stack frame using **ESP** and **EBP** needs to be done before this.

```
...end of function reached...
leave
ret
```

- **MOV** - Main instruction when working with data, as you can do direct and indirect stuff if you know memory addresses.

```
movw    %ax, %bx      # move 2 bytes into bx
```

- **PUSH, POP** - Push and pop items onto/from the stack.

```
pushb   %ah           # push lower byte of ax
popl    %ebx          # pop 4 bytes from stack onto ebx
```

- **JMP** - Jump to a different part of the program, either by using a label or having a jump table (holds function pointers to go to a function).

```
    jmp     some_label
    ...
some_label:
    ...
```

- **IRET** - Similar to ret, but it does a lot of background stuff that will be useful later. Don't use for now.

```
...end of function...
iret
```

- **INT** - Causes an interrupt based on **interrupt descriptor table** (**IDT**).

```
int     $0x80   # goes to system call interrupt handler
```

- Use **CMP, TEST** to set flags such as **Sign Flag, Zero Flag, Carry Flag, Overflow Flag, and Parity Flag**.
  - One thing to note is that unsigned and signed comparisons are different. The following instructions jump to specified label if condition is met (eg `ja label_name` causes a program to move to `label_name` if a comparison happened, and val1 > val2)

unsigned	jne	jb	jbe	je	jae	ja
relationship	≠	<	≤	=	≥	>
signed	jne	jl	jle	je	jge	jg

- For memory accesses (dereferencing), the general syntax is something like this:

```
movl    (%eax, %ebx, 4), %ecx
```

  - Translation (with variables, assume eax is unsigned long type): `int32_t ecx = *(int32_t*)(eax + ebx*4);`
    - Note, eax is unsigned long in this example instead of int* to deal with the fact that pointer arithmetic will change what happens based on the type it points to, so typecast and deref.
  - The value in **EAX** is offset by the value in **EBX*4**, which will then be dereferenced to store in **ECX**.
- For reference, use these two links when programming in x86:
  - http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html
    - Definitely keep this open if you're forgetful of syntax, as there are multiple ways some instructions can be done.
  - https://drive.google.com/file/d/1Gvhio3cgXXQ6u8FrudeJ7c-ykXKkdkZW/view?usp=sharing (the x86 reference sheet from course site)

3 - x86 isa: assembler conventions, calling convention, examples

4 - C to x86 linkage, device I/O; role of system software, system calls

- **Calling convention (IMPORTANT TO KNOW)**
  - Say you call a function in x86 that's defined like this

    ```
    int c_func(int a, int b, int c);
    ```

  - Parameters are pushed from right to left, meaning you would do something like this (**EAX** = **a, EBX** = **b, ECX** = **c**, random regs picked):

    ```
    .globl   c_func

    random_label:
    ....some lines of code later....
         pushl    %ecx
         pushl    %ebx
         pushl    %eax
         call     c_func
         addl     $12, %esp          # args no longer needed
    ....rest of code....
    ```

  - Why push like this? Convention people agreed to. It allows functions to access variables in an easy way, knowing that the first variable is (ignoring the return address) the first on the stack, second variable is second, and so on.
  - `addl $12, %esp` restores stack back to state it was before **c_func** was called.
  - Registers **c_func** destroys if not saved properly:
    EAX, EDX (return value stored in EAX; EDX too if 64 bits), **ECX, EFLAGS** (shouldn't need to preserve this for the most part).
  - Registers **c_func** must preserve: **ESP, EBP, EBX, ESI, EDI**
  - Same goes for x86 functions, say you have a simple function defined like this

```
int called_func(int argument);
```

that's called from a C function (the function itself is useless):

```
1 called_func:
2     pushl    %ebp              # setup new stack frame
3     movl     %esp, %ebp        # by giving ebp esp val
4     pushl    %ebx              # using ebx, save prev val
5     movl     8(%ebp), %ebx     # get first argument
6     popl     %ebx              # restore ebx
7     leave                      # tear down stack frame
8     ret                        # return to prev function
```

- Go through it step by step
  - Lines 2-3 are setting up the stack frame for the function, you don't always need this (as seen in MP1 by ioctl dispatcher)
  - Since this function is called, it should preserve any registers that it uses that need to be preserved (in this case line 4, **EBX**)
  - Line 5 gets the first argument off the stack, puts it into **EBX**
    - **ESP** points to return address at the start of the function
    - Since we pushed **EBP**, there's an initial off by 4 to take into account
    - Offset needed to get the argument (not the return address) is then 8, meaning movl 8(%ebp) just gets the 4 bytes starting at memory (%ebp + 8) and puts into **EBX.**
    - If **EBX** wasn't pushed and used a different register, first argument would be gotten like this.

      ```
      movl    4(%ebp), %eax
      ```

  - Once you're done using the register, has to be popped in **reverse order** (if you pushed say **EBX, ESI,** you should pop **ESI, EBX,** in that order).

        - Line 7, leave, is the equivalent of doing:

```
addl    $4, %esp
movl    %esp, %ebp
```

        - Restores ebp if stack is back to how it was from pushing **EBP**, otherwise strange errors (and kernel panics) can occur. **Don't use leave if you don't set up a stack frame.**
        - ret goes back to function that called called_func, popping the return address off the stack.

- **Device I/O (Important for MP2, MP3)**
    - Ports are 16 bits, byte addressable and little-endian
    - **IN -** (port, dest_register) as arguments (reads data from port)
    - **OUT** - (source_register, port) as arguments (writes data to port)
        - When dealing with register/port stuff in MP2, refer back to here and the macros they have defined (I think arguments are reversed in the C macros). I was stuck for a while until I realized it was simple enough to use.
    - One thing to note is that the ports are in memory, so you have to know the memory address of whichever port you want to use to read/write data to it.

- **Data Type Alignment**
    - In C, structs may be padded to deal with how the processor accesses data. Say you have a struct like this:

```
1    typedef struct dummy {
2         int a;
3         char b;
4         int c;
5    } dummy_t;
```

    - If you were to use the sizeof( ) function and see that the size is 12 and not 9, padding was at play here to make it easier on the processor to access data faster.
        - Adding `__attribute__((__packed__))` between 'struct' and

'dummy' will give you 9 (not recommended for x86).

- No padding to worry about in MP1 though, use defined constants they give you.

- Let's compare two structs:

```
typedef struct dummy1 {
      int a;
      short b;        // short is 2 bytes, int is 4
} dummy1_t;

typedef struct dummy2 {
      short a;
      short b;
      short c;
      short d;
} dummy2_t;
```

- Both structs will be 8 bytes in size, but `dummy2_t` seems to be able to hold more data if we distribute it as 4 shorts (or an int and 2 shorts) instead of an int and a short.
- In x86, use .align #, where # is the argument in however much you want to align certain things. For example:

```
random_label:
      .word    0
      .long    0
      .align   4
```

- .align here will make sure you can access the .long data by offsetting `random_label`'s memory address by 4.

- **System Calls**
  - I don't see this anywhere on the slides but can give info (mostly midterm 2 material though).
  - **System calls** are the way to access kernel level functions through userspace.

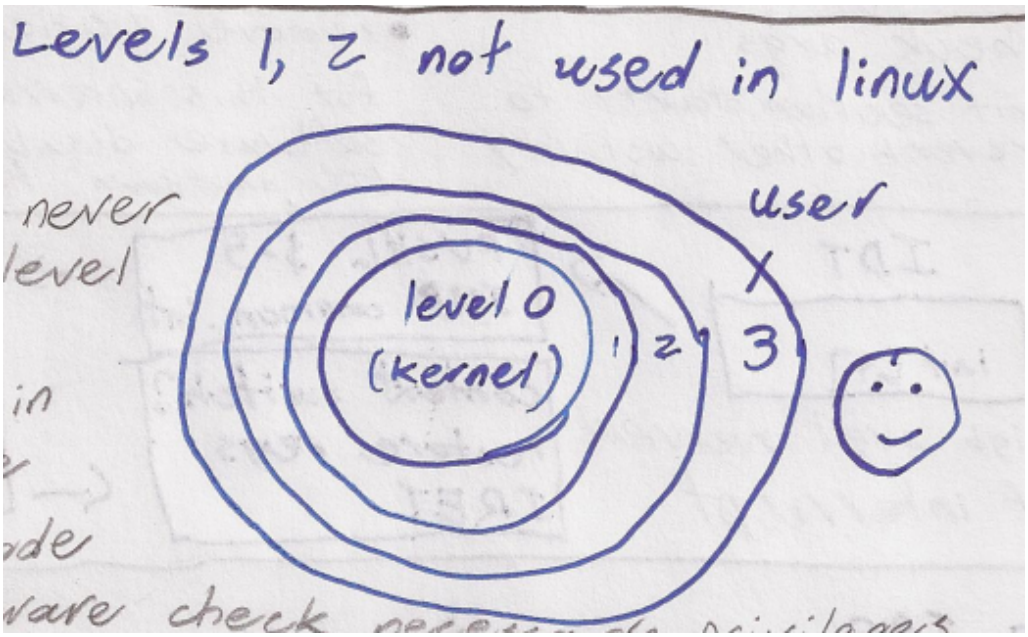
- Let's look at `malloc()` for a bit:

  ```
  int* temp = (int*) malloc(sizeof(int));
  ```

  - The user interface is the function we all know; underneath it though is a bunch of kernel functions doing the actual heavy work for it. How are parameters passed in to kernel functions? Registers.
  - In the IDT, each system call would have a corresponding number, with **EAX** holding that value. So when

    ```
    int $0x80  # goes to system call interrupt handler
    ```

    happens, the system call handler would use **EAX's** value to call the correct function.
  - The registers, in order of parameters, are **EBX, ECX, EDX**. This means `sizeof(int)` would be put into **EBX**.
- Two privilege levels/rings to worry about: **ring 3** (user space) and **ring 0** (kernel space).



  - Trying to do anything that requires **ring 0** privilege level in user space results in an exception occurring (for example, trying to access kernel memory). Getting back to **ring 3** from **ring 0** is a bit of work that isn't covered yet.

Discussion Week 2 - PS1, x86

- **Function Pointers and Jump Tables**
    - At first, it may seem weird to use, but there are some cases where function pointers are needed (x86 in MP1, C in MP3).
    - **Jump tables** are used to call a function knowing offset and an actual table.

```
/* void* = can be casted to different stuff */
void *jmp_table[3] = {func1, func2, func3}
```

    - A simple table like above could be used to lower amount of code used (for example, instead of if statements/a switch, can call a function knowing index you want based on conditions).
    - In x86, a jump table can be declared like this:

```
func1:....
func2:....
jump_table:
     .long func1, func2
```

    - In this table, func1 and func2 would be the x86 labels (or C function addresses in C). To actually jump to a function, the general syntax would be like this:

```
/* Calling func2 here */
movl    $1, %ecx
jmp     *jump_table(, %ecx, 4)
```

    - The * before jump_table means it will be based on `func2`'s value. You may think it's dereferencing the jump table, but technically it's not doing so, it's slightly weird.
- **Typecasting (Very important later, PS1)**
    - C is a language that, if you mess up and it compiles, it's going to be a bad time. But because it's like this, you can do a lot of weird stuff. For example, say you have something like this:

```
/*
 *  uint32_t = unsigned int (4 bytes).
 *  register lets C know to keep variable in register (in
 *  this case, ebp since it holds ebp already).
 *  The asm part just gets the value in ebp for the C func.
 *  Note that this doesn't work without register keyword.
 */
register uint32_t ebp asm("%ebp");
uint32_t addr = *((uint32_t*) (ebp+4));
```

- What this effectively does is get you the return address of the function that called the current function, and putting it into `addr`. For the problem set, if it doesn't make sense, typecasting is what they want you to do probably.
- Another silly example:

```
/* 0xECE391 holds a pointer to a string */
uint32_t x = 0xECE391;
char* string_391 = (char*) *((uint32_t*)x);
```

- **x86 to C**
  - Go to discussion and listen to TA

5 - Interrupts and exceptions, processor and ISA support (Note: A lot of this you won't see/apply until MP3, but will be on the midterm)

- System software serves 3 main purposes: **virtualization, protection, abstraction.**
    - **Virtualization** - Makes it look like there's unlimited resources available to use.
    - **Protection** - Stops the user from doing silly stuff or someone else from being hacker man.
    - **Abstraction** - Hides asynchronous nature by appearing to be running in parallel (in a single core processor).

- **System Calls and Exceptions**
    - Uses a jump table to go to correct function.
        - To be able to use a **system call**, the following needs to happen:

          ```
          int $0x80
          ```

        - (FYI: This version of working with system calls is deprecated due to speed in newer versions of linux, but used in this class).
        - Similar to LC-3 **TRAP** instruction, **INT** goes to specific IDT table entry (in this case, IDT entry 0x80).
    - **Exceptions** happen when something goes wrong in a program/process.
        - Like system calls, map a table of exception handlers to deal with different exceptions.
        - Around 20 intel mapped exceptions.

- **Interrupts**
    - An **interrupt** interrupts the OS (wow). It can happen unexpectedly whenever any program is running.
        - **Maskable interrupt** can be stopped for a certain amount of time, either by clearing the interrupt enable flag (**IF**) (using **CLI** in x86) or not actually being set up properly in MP3 :(

- **STI** restores **IF**, use after you use **CLI.**

```
cli
...some x86 stuff…
sti
```

        - **Non-maskable interrupt** can't be stopped.
    - Two interrupt types exist to deal with the fact that some are too important to stop.

- **Interrupt Descriptor Table (IDT)**
    - First, go through what each group entry represents.
        - **0x00 - 0x1F** - Defined by intel.
            - Entry groups for all exceptions that happen, there may be blank entries in here if not defined yet.
        - **0x20 - 0x2**7 - Master 8259 PIC (PIC introduced in lecture 9).
            - 8 interrupt request (IRQ) lines. Important ones are:
                - **IRQ0** = Timer chip, use this to schedule in MP3.
                - **IRQ1** = Keyboard handler (I hated this).
                - **IRQ2** = Cascade to slave PIC (more IRQ lines).
                - **IRQ4** = Serial port (KGDB).
        - **0x28 - 0x2F** - Slave 8259 PIC.
            - Another 8 IRQ lines. The important ones are:
                - **IRQ8** = Real time clock.
                - **/* Only IRQ8 from here worried about in MP3 */**
                - **IRQ11** = network stuff.
                - **IRQ12** = PS/2 mouse.
                - **IRQ14** = hard drive.
        - **0x30 - 0x7F** - Vectors available to device drivers.
        - **0x80** - System call vector.
        - **0x81 - 0xFF** - Idk other stuff.
    - The IDT has to be set up properly for all of the different interrupt

types, otherwise nothing will go through

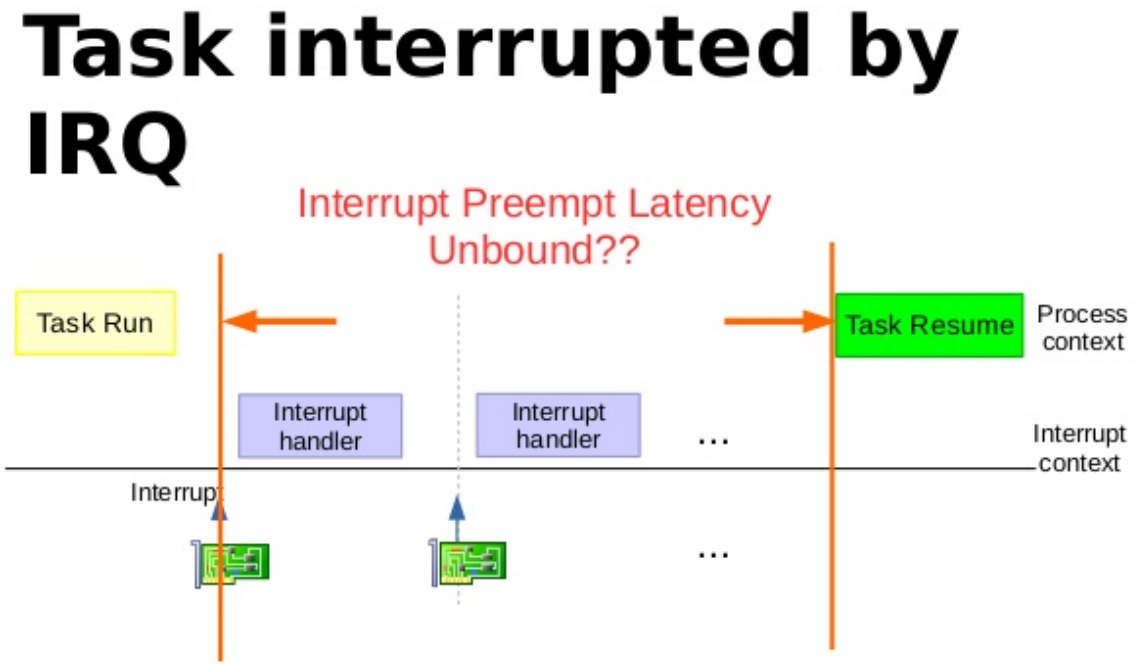
        - Going along with this, the PIC has to also be setup for the 8 (16 w/slave) IRQ lines to handle those specific hardware interrupts.

    - Why is this important? It is the main entryway for the operating system to actually do important things (keyboard input, launching programs, scheduling, etc.).
    - An efficient way to deal with interrupts and exceptions, in that you can change them without messing up the operating system if you just have function pointers in your IDT.

Extra Notes on Lecture

- The point of this section is to understand certain concepts that may not be fully understood at the moment, so it touches a bit on later lecture material to try to help you understand.
- Interrupts are going to be a very important part of the course. They touch on it a bit in MP2, but MP3 deals with many interrupts.
- **PIC**
    - The PIC is a hardware device that lets your OS actually do stuff.
    - The IR lines on it are the IRQ lines talked about.
    - Certain data has to be sent to the PIC to actually initialize it using physical memory I/O.

- **Interrupts and Exceptions**



  - **Interrupt Handlers** are functions that deal with the interrupt that occurred.
    - For example, keyboard interrupts caused by clicking on a key needs a keyboard handler to deal with it.
    - After an interrupt has finished, at the end you need to send an end of interrupt (**EOI**) signal to the PIC to the correct IRQ line, otherwise the interrupt will most likely not happen again.
      - If your interrupt handler is on IRQ 2 for example, send **EOI** to IRQ 2.
      - For how to actually send **EOI**, refer to linux code handouts they give later on to get important information for MP3.
  - **Exception Handlers** are functions that deal with any exception that occurs in the system.

        - In MP3, your exception handlers have to halt the current process and return to the previous process.

- **High Level Overview of an Interrupt**
    - It can happen unexpectedly, or scheduled at a specific rate.
        - **Real Time Clock (RTC)** and **Programmable Interval Timer (PIT) / Timer Chip** are examples of scheduled interrupts.
        - Each one sends an interrupt after certain time has passed, based on initialization settings.
    - When a process is running, and the specific IRQ line isn't masked, an interrupt can occur at any given time.
        - Masking stops it from occurring until unmasked again.
    - Since it can happen unexpectedly, a few things have to happen
        - Save registers.
        - Context switch from user level to kernel level by doing the appropriate work needed for it (MP3)
        - Be able to return to previous process/function without having modified their stack/registers.
    - Synchronization constraints and other issues are worried about when dealing with interrupts, and designing how they will work in the system.

Discussion Week 3 - MP1, x86, calling convention

- **Demo Questions**
    - The questions they give aren't too bad, if you can give a semi decent answer your TA will probably guide you to what they wanted to get you those nice 10 points on the MP.

- **MP1 Overview**
    - The infamous 391 workload begins with MP1.
    - All in x86.
    - Fall MP (Missile Command) is harder than Spring one (Text-Mode Fish Animation).
    - Lots of lines of code incoming.

- **Tasklet**
    - Tasklets are a part of interrupt handlers, in that they are used to defer work from actual handler to this type of function.
        - Interrupt handlers need to be as short as possible due to them stopping the process they interrupted.
        - If work is deferred, can schedule another process while previous process is happening.
    - You're creating a tasklet in MP1 for your animation to happen, working with the **RTC** (only need to write tasklet, makefile/instructions help setup your tasklet).

- **Magic Numbers**
    - This will be pretty hard to keep track of; you're gonna have to comment a lot, define a lot of constants, and just try very hard to not have spaghetti code in your MPs.

```
1    movl    %eax, %ebx
2    addl    $23, %eax
3    pushl   %eax
4    call    some_func
```

- Why does the above code move a value into ebx, then push an updated eax value? You can infer that you're saving the previous eax value, but why add by 23?.
- Comment your code, and define as many constants as you can to lower the amount of magic number usage.

```
addl    $VIDEO_OFFSET, %eax
```

- With constants, you can make sense of stuff easier as you go through the MP and debug when bugs arise.
- You also don't need to go overboard with comments.

```
1    .data
2         NULL    = 0
3         SUCCESS = 0
4         ERROR   = -1
5
6    .text
7
8    /* int some_func()
9     *   Description: Sample function to show commenting, check
10    *                if the memory addr passed in is null
11    *   Inputs: %eax - Holds the previous func's memory addr
12    *   Outputs: 0 on success, -1 on error
13    *   Registers: %eax holds output, %ecx will then hold
14    *              memory addr passed in (if not null)
15    */
16   some_func:                  # no stack frame needed here
17        cmpl    $NULL, %eax   # check if addr null, error if so
18        je      null_addr
19   func_success:
20        movl    %eax, %ecx    # saving output in eax, addr in ecx
21        movl    $SUCCESS, %eax
22        ret
23   null_addr:
24        movl    $ERROR, %eax
25        ret
```

    - One thing to note is that they really want you to write interfaces like the one above the actual function, so get in the habit of doing that.
    - You can lose a lot of points in your MPs over time because of magic numbers.

- **Virtual Memory**
    - One thing to note is that user-space addresses are different from kernel space ones.
        - Use provided functions like **mp1_copy_to_user** and **mp1_copy_from_user** when dealing with any user space address (or kernel to user space).
        - Also use to check for bad addresses instead of null checking since it does more checks.
    - Don't worry too much for now about it.

- **Debugging**
    - Use gdb as much as you can when your program is messing up.
    - Working in user != working in kernel.
    - tar re 10.0.2.2:1234 is your friend.

6 - Shared resources, critical sections, examples

- **Shared Resources**
  - Due to the nature of operating systems, lots of resources may be shared by multiple parts.
    - This in turn leads to problems in dealing with how to share the resources.
  - What if something like a linked list is shared between two very different things?

```
list_head:
     .long 0

some_interrupt:
     ...uses list_head here...

1 update_head:
2    movl    list_head, %eax    # store previous head
3    movl    %ecx, list_head    # update list head
4    movl    %eax, 4(%ecx)      # update next ptr of head
5    ret
```

  - The problem with the above `update_head` function is that it doesn't properly update the head.
    - `list_head` is updated in line 3 without taking into account that the function can be interrupted at any given moment.
    - Since `some_interrupt` uses the list data, if it interrupts `update_head` at line 3, it will only have the new head's data, which could have garbage or NULL for the next ptr, making it the only item in the list or worse.
  - By taking the interrupt into account, can swap instructions:

```
1 update_head:
2    movl    list_head, %eax    # store previous head
3    movl    %eax, 4(%ecx)      # update next ptr of head
4    movl    %ecx, list_head    # update list head
5    ret
```

- A simple switch in which instruction happens first can fix this problem.
- There are more complex problems to worry about though, and so different measures will have to be taken into account.

- **Compiler Optimization**
  - When you're first introduced to this, you might think problems that may arise in MPs are due to this and start turning stuff into volatile.
    - Most likely not the problem but still a possibility.
  - **Volatile**
    - Any variables marked **volatile** will be constantly re-initialized/reloaded. (For this example, just know there can be multiple threads in a program, each one able to do its own thing and focus on a specific part/multiple parts).

```
1     volatile static int busy_variable = 0;
2
3     /* Thread 1 */
4     int func1() {
5           while (busy_variable == 0) {
6           ...do stuff...
7           }
8     }
9
10    /* Thread 2 */
11    int func() {
12          ...do stuff...
13          busy_variable = 1;
14          ...do stuff...
15    }
```

    - In the example above if you didn't have the keyword, there's the off chance that the compiler would optimize your program and would make the while loop like this:

```
while (1)
```

- To the compiler, it may look like `busy_variable` is never modified and may change your while loop to always occur since it's never modified in `func1()`.
- Volatile variables come at a cost. Since you're always reloading the variable, a performance hit is taken every time you use it.

```
busy_variable = 1;
// Simplified version of what happens
// busy_variable = old value in memory
// busy_variable = 1
```

- **Code reordering**
  - Another thing compilers may do is change how your code actually executes by switching which instructions happen first. It's not predictable, and not much you can do about that when it happens (unless you know the inner workings of gcc).
  - If you have the unlucky fate of a bug happening because of this, I'm sorry.
- **Atomic Operations**
  - When code executes **atomically** it is happening as if it was one operation from a different perspective.

```
movl    %eax, %ebx
```

  - The above is atomic since it's only one instruction.
  - What else can be considered atomic?

```
cli
...multiple instructions...
sti
```

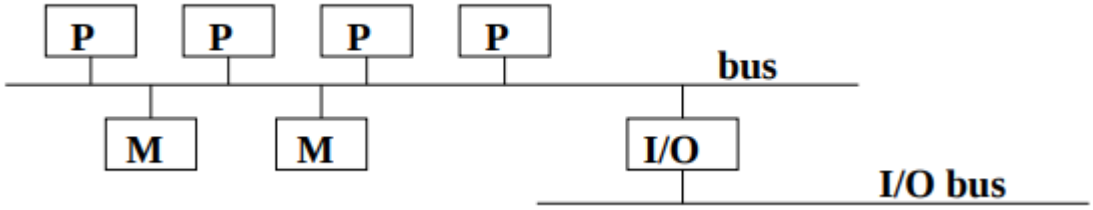
  - Since **CLI** clears **IF**, the only way this can be interrupted is through an exception or non-maskable interrupt, none of which should change anything that happens to data it uses.
    - **CLI** and **STI** can be made C macros to use these in C files, or even this probably works:

```
asm volatile (“cli”);
...C stuff...
asm volatile (“sti”);
```

- **Critical Sections**
    - Used when you need to make sure code executes in a specific order without being interrupted (and, even with code reordering, will work as intended).
    - Using **CLI** and **STI** is one way of creating critical sections, as interrupts that would affect/use certain data can’t occur.
    - Drawbacks: Since it clears **IF**, no maskable interrupt handlers are going through, meaning that any interrupts that are happening at the time are delayed or not happening at all.
        - Can crash the system due to timeouts if certain handlers aren’t handling important maskable interrupts.
        - Keep critical sections as short as possible, otherwise risk both performance losses and bad things happening with the system itself.
    - Critical sections by themselves are no longer enough due to how complex modern operating systems have become. Having multiple processors/cores makes things pretty complicated.
        - Solution(s)? Locks (next lecture).

7 - Multiprocessors and locks, conservative synchronization design

- **Multiprocessor**
  - **Symmetric Multiprocessor (SMP)**



    - All processors have equal latency to memory banks.
    - Different than multi-core processors but similar from our perspective.

  - **Non-uniform memory access (NUMA)**
    - Memory access depends on location relative to the processor.
    - Processor can access local memory faster than non-local memory.
    - Used for cluster multiprocessing systems, follow from SMP.
- **Locks**
  - If you clear **IF** on one processor, other processors still have their **IF**.
  - In terms of speed, clearing the interrupt flag for thousands of processors in a huge system is a heavy cost in performance.
  - **Locks** use shared memory between multiple processors to be able to synchronize properly. A simple explanation of how they work is down below:

    1. Wait until it's unlocked.
    2. Once unlocked, one processor/thread locks it.
    3. Do stuff with the data.
    4. Unlock it for others to use.

  - How are locks made, such that they happen atomically? That topic is beyond the scope of this class (highly optimized implementations that is, simple example next part).

- **Spinlock**
  - From the wikipedia page, a simple implementation could be something like this (converted to our syntax from intel):

```
1    locked:
2         .long   0             # 1 = locked, 0 unlocked
3
4    spin_lock:
5         movl    $1, %eax
6         xchg    locked, %eax  # atomically swaps values
7         cmpl    $1, %eax      # stop when locked changes value
8         je      spin_lock
9         ret
10
11   spin_unlock:
12        xorl    %eax, %eax    # clear eax, and unlock locked
13        xchg    %eax, locked
14        ret
```

  - For a uniprocessor/single core processor, spinlocks are replaceable.
    - Since you can clear the **IF,** that's all you need.
  - Used a lot in modern operating systems due to reliability (but only when it's for a short period).
  - Why not long periods? It's essentially doing this:

```
while (locked) {
     // spin, in other words do nothing
}
```

  - The thread trying to access the lock essentially becomes useless until it acquires it, ignoring doing anything else (besides handling interrupts if IF not cleared) while waiting for the lock to be unlocked.
    - You can also clear interrupts along with the spinlock if you need to.
  - Simple example of how to utilize spinlock API:

```c
#include <linux/spinlock.h>
// Usually want lock initialized at start, not dynamically
// but there are cases when it needs to be dynamic.
// SPIN_LOCK_UNLOCKED is from included file.
static spinlock_t lock = SPIN_LOCK_UNLOCKED;

// spin_lock and spin_unlock both take in pointers to the
// lock to change it.
int thread_1() {
      // Activates in middle of program once only
      spin_lock(&lock);
      ...thread 1 stuff...
      spin_unlock(&lock);
      return 0;
}

int thread_2() {
      // Active thread, do stuff until breaks
      while (some_condition) {
            spin_lock(&lock);
            ...thread 2 stuff...
            // Unlock after every loop cycle in case thread 1
            // is started and uses it
            spin_unlock(&lock);
      }
      return 0;
}
```

    - **Remember: Keep lock as short as possible.**
    - With respect to operating systems, there are different spinlock functions you should use based on the situation, such as if you have an interrupt happening that shares data with a system call (this is where it gets interesting).

- **Race Conditions**
    - Say you have two race cars trying as fast as they can to reach a pit stop that only takes in one car at a time.
        - Race car 1 reaches it before race car 2.

        - Race car 1 is in such bad condition that race car 2 has to wait for a very long time.
    - **Race conditions** are programming conditions that can lead to programs acting differently based on which process/thread reached a certain point first.
        - One of the outcomes is a **deadlock** (similar to race car example, except waiting forever instead).
    - The thing about race conditions is the fact that many things could end up causing it/many things can go wrong.
        - Not accessing lock appropriately.
        - Critical sections not done right.
        - Accessing data that may have been modified in bad way.
        - Interrupts happening at unexpected times.
    - In next lecture, will go more in-depth into synchronization hazards.

Discussion Week 4 - PS2, Synchronization

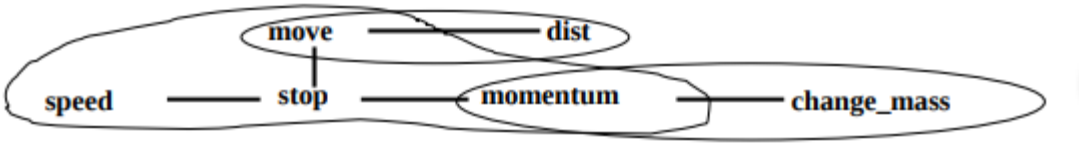
- Don't overthink the problem set. If you think there are certain conditions happening that can mess things up, don't overthink it.
- Most of discussion ended up talking about earlier oops, will just expand on a few things from the ***When Using Locks*** slide.
- **Do not protect regions of memory from modification.**
    - Protecting regions from modification = no.
        - Never actually mark something as unmodifiable.
    - Good locking strategy is to protect only when need to, use for as short of a period as possible, and make sure performance isn't taking a hit.
- **Do not mark certain data structures as locked.**
    - Inefficient, maybe you want to access certain parts of the structure but not something that can be modified by another thread?
    - Use locks when you have to read/write to a struct part that can be modified somewhere else.
- **Adding a lock to a struct does not magically protect that struct. Does not matter if lock is in the struct or not. It is up to the programmer to protect against race conditions.**
    - Your design has to make sense with how you're using locks.
        - Too many locks = bad performance.
        - Too little locks = race conditions and other bad stuff.
- **Reference (some locking examples w/ other stuff):** https://web.stanford.edu/~ouster/cgi-bin/cs140-spring14/lecture.php?topic=locks

8 - Linux synchronization interface, synchronization hazards

- **Lock Ordering**
    - Sometimes, you need to make sure that you lock and unlock in a specific order (next part similar to lecture example, but slightly different).

Person 1	Person 2	Person 3
LOCKA	...	LOCKA
...	LOCKB	...
LOCKB	LOCKA	...
UNLOCKA	UNLOCKB	LOCKB
UNLOCKB	UNLOCKA	UNLOCKA
		UNLOCKB

        - Person 1 with Person 2 leads to a very high chance of a **deadlock** happening.
        - Person 1 with Person 3 leads to a safe (but very inefficient example), as they both go for **LOCKA** but only one person can get it at a time.
        - Person 2 with Person 3 also high chance of **deadlock**.
            - Why do I say high chance? Unless you're absolutely sure of when certain operations are going to happen, you can just say high chance.
            - If you say each line is a specific time slot and an operation is guaranteed to occur at that slot, then a **deadlock** does happen.



    - Pic above from lecture, but each one is a different function affecting an object/data (lecture slide has related code).
        - The above graph shows the need to lock certain stuff based on the type of data being read/written.
        - You can just make one giant lock for everything, but it becomes slow.

        - **move, speed, and momentum** are all connected by **stop**, with lines signifying which function needs to be atomic w/ respect to another.
        - Since the other functions aren't concerned, use one lock for them.
        - Some functions would need two locks (**move, momentum**) since they would modify/read data used by other functions not related to stop.
    - The example signifies the need to look at data to make sure you can make performance as good as possible, while also keeping race conditions in check.
        - Having just one thread/process = don't worry about this stuff though.

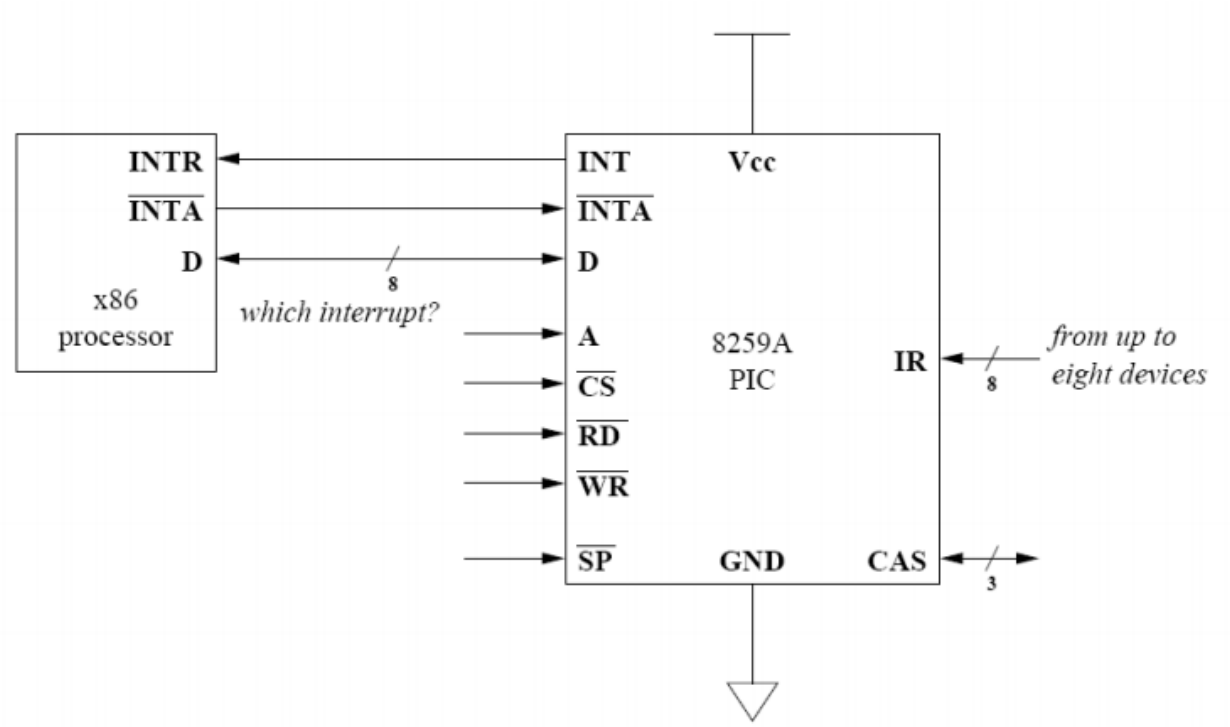
- **Semaphore** (I was confused about this for a while)
    - A **semaphore** is another type of lock that puts other threads to sleep when they try to access it but it's locked.
        - Can increase performance when a thread that doesn't need to be active is put to sleep.
    - It allows a certain number of threads to access certain resources/be able to do stuff (number has to be specified), basically putting other threads into a queue to wait, waking up every now and then to check.
    - A **mutex** is commonly thought of as a semaphore with only 1 thread allowed since it also puts other threads to sleep when being accessed while locked.
        - Think about this though, say you have a bathroom and you use a mutex to protect the key.
        - Now say you have 2 bathrooms (men, women) and use a semaphore. If you're using a semaphore for the two bathrooms, wrong way to use it.
        - You would instead use 2 distinct mutexes for each bathroom.
        - Why? Using a semaphore can be analogous to putting keys to the two bathrooms in a basket, but you need to keep track of the bathrooms separately since they're not identical.

        - If the keys unlocked both bathrooms and anyone can enter either bathroom, then a semaphore would be fine since you don't need to worry about the type of key being used.
        - Summary - Use mutexes for when you need to protect distinct resources, and a semaphore when it doesn't matter.
    - **Don't use for interrupt handlers**.
        - Since it puts other threads/processes to sleep, negative effects can happen in the kernel since interrupt handlers are supposed to be fast.
        - Sleep = kernel gonna die.

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`

    - Make sure to understand above picture, can pop up on tests.
    - up, down = semaphore/mutex locking/unlocking related stuff.
    - The main difference to note between `spin_lock_irq` and `spin_lock_irqsave` is the fact that `spin_lock_irq` does not save the previous interrupt state.
        - If the previous locking state mattered, use `spin_lock_irqsave`.
        - `spin_lock_irq` is 'only safe when you know that interrupts were not already disabled before the acquisition of the lock' (Robert Love, 2002)
        - Easier to just use `spin_lock_irqsave`, but know differences of when to use either one.

9 - Programmable interrupt controller motivation and design



Covering more in next lecture, this one mainly talks about how PIC came about and its design.

- **Hardware**
    - **INT** (Interrupt) - connected to CPU's interrupt pin, high when valid interrupt request made.
    - **INTA** (Interrupt Acknowledge) - Enables PIC interrupt vector data (think of the IDT here).
    - **D** (Bidirectional Data Bus) - Data flowing from CPU to PIC and vice versa.
    - **A** (Address Bus) - Specifies physical address for data.
    - **CS** (Chip Select) - enables read (**RD**) and write (**WR**) when low (0).
    - **SP** (Slave Program) - specifies if it's master (1) or slave (0) PIC.

- **Programmable Interrupt Controller (and Interrupts)**
    - The **PIC** is the main device that is used when dealing with interrupts.
    - An interrupt handler that deals with certain hardware interrupts can correspond to 1 of the 8 **IR** lines on it.

        - For example, the keyboard handler would be on **IR** line 1 (lines are 0-7) in x86
    - **Interrupts** are a way for hardware to send signals to the kernel that it finished whatever it was doing, and allows software (interrupt handlers) to then deal with it.
    - Since interrupts are asynchronous, they're able to interrupt a task without worrying about that task needing to have finished yet.
        - One thing to note is that interrupt handlers should be dealt with **fast**, as slow handlers could cause problems and crashes
    - One of the benefits of using interrupts is that you don't have to poll (or continuously check) whether something is available, instead allowing the PIC to send the signal whenever it's ready for the interrupt.
    - The PIC allows the processor to do its own thing, and only deal with the interrupts as they happen.

- **Cascading**
    - The PIC allows you to connect multiple PICs (slaves) to the PIC connected to the CPU (master).
    - Each external PIC would be connected to one of the IR lines on the master, with the same interrupt rules applying based on the line a slave is connected to.
    - For MP3, you'll have one slave PIC to worry about, make sure you properly send eoi.

❤ *Discussion Week 5 - MP2.1, Synchronization* ❤

(Valentine's day discussion slides)

- **MP2.1**
    - You may feel overwhelmed at first, and I'm just gonna say it'll be ok. Do the best that you can, and don't let it get to you.
    - Try to finish early because of the test around this time, and MP2.2 is only a week for a lot more points.
    - Status bar is a pain no matter if it's spring or fall, get it done as soon as possible (though harder in fall as it needs to be synchronized there I think).
    - Setup a debugging environment that works for you fast, as this one is tough to debug. Will have info in a bit, just need to search piazza from TA about his setup.
    -
- **Mode X**
    - There's one small paragraph under **Mode X and Graphic Images** (pg4 of spring MP2) that states the double buffering method is not the same in QEMU due to emulation, take this into account and not get confused if some things seem different from what you've read.
    - Looking at the **modex.c** file, it's around 1000 lines by itself. I'd say start by looking at main (at the very bottom) and start working your way through each function that's made if you can't continue until you understand what's happening (note that doing this can take a while). Figure out what's important and what isn't for you to add the needed functionality.
        - As an example, **open_memory_and_ports** is the first function called. Lets determine if it's needed for us or not.
        - First, it's trying to see if we can actually use the ports with a system call **ioperm**, in which the ports are just VGA registers (general, graphics, attribute, etc, digging through problem set 2 link gives us this info).
        - Using another system call **open** to use physical memory for the next step.
        - The next one, **mmap**, will allow you to write to physical memory using a virtual address space from **mmap**. This is

noted in the documentation that the addresses may not be the same which is fine, since it's just mapping a different address space to another one. So if you write to something like 0xff0241, it'll write to physical memory 0xA0000 if the output of **mmap** was 0xff0241. This ensures that you actually modify the addresses you want instead of some weird bugs happening because of virtualized addresses.

- Close is called for cleanup, ensuring no open processes/memory is being used after the program ends.
- So is it important for us? Not quite. It helps to clear up a bit of the documentation and helps us understand why certain things are working the way they are, and shows us that **mem_image** will be what we're going to modify to change the game (though the comment on **mem_image** already lets us know); other than that, it's mainly info on what's happening to initialize registers/memory stuff for Mode X.

- This link could also be useful, more VGA stuff (quite a bit actually).

10 - Linux abstraction of PIC; Interrupt support in Linux: data structures, installation and removal

12 - Interrupt support in Linux: initialization and assembly linkage

Discussion Week 7 - MP2.2, Tux Synchronization

13 - Interrupt support in Linux: invocation; summary of the interrupt support

14 - Soft interrupts/tasklets; Virtual memory: rationale, segmentation

Discussion Week 8 - MP3 overview, MP3.1

15 - Virtual memory: paging

16 - Filesystem: philosophy, ext2 as example (file system in MP3)

Discussion Week 9 - MP3.2

17 - Programs to processes: rationale, terminology, and structures (task structure, kernel stack, TSS)

18 - Programs to processes: creating processes; job types and basics of scheduling; scheduler design and implementation

- **Types of Jobs**
    - Depending on the job time, scheduler can focus on other processes/give them more priority based on what is needed from the different types of jobs.
    - **Interactive**
        - Although there's not much work for this type of job, since the user is interacting with it, a quick response time is needed.
        - Wake up quick when inactive, otherwise can appear funky.
    - **Batch**
        - Data processed at regular intervals.
        - Run in background mainly.
        - Since it uses a fair amount of CPU, best to schedule when there are not other main processes that need good speed/not much activity.
    - **Real-time**
        - Work should be done quick based on requirements needed for the specific job.
        - Shouldn't be blocked by processes with lower priority, as they should have a short guaranteed response time.
            - Given highest priority, but priority for separate ones decided among themselves.
    - There are 2 other ways to classify, namely **I/O-bound** and **CPU-bound**.
        - **I/O-bound -** Uses a lot of I/O devices and operations (database server is one).
        - **CPU-bound -** Requires heavy CPU usage to complete (image-rendering program is one).
- The linux scheduler has certain algorithms in play to be able to dynamically change priority of a given process based on many different factors.

- **Scheduling**
    - Rule of thumb is to keep quantum duration as long as possible, while also as responsive as possible (interrupts, interactions, etc).
    - **Preemptive**
        - Involuntarily suspend running process to work on another (used in Linux).
        - **Timeslice** / **Quantum** - time a process has before another one is then worked on (aim for milliseconds).
            - If it's too short, overhead from constantly switching processes becomes high, slowing down applications.
            - If it's too long, concurrent illusion is no longer true (say it's 5 seconds, your program will stop for 5 * (number of current tasks - 1)).
            - If you have preemptive scheduling and priority in place, having long quantums shouldn't have too much of a bad effect since you can change the task whenever (along with decent enough CPU).
        - Interactive jobs can preempt when something like a keyboard interrupt occurs.
    - **Cooperative**
        - Process does not stop running until voluntarily does so.
        - Not used much as processes that hogs CPU / becomes stuck never finishes, effectively killing OS.
    - There are five states any task can be in at a given moment.
        - **TASK_RUNNING**
            - Process is executing / in run-queue.
        - **TASK_INTERRUPTIBLE**
            - Suspended until a condition becomes true.
        - **TASK_UNINTERRUPTIBLE**
            - Same as above, except signals can't wake up this process.
        - **TASK_STOPPED**

            - Execution has stopped.
                - SIGSTOP, SIGSTP, SIGTTIN, SIGTTOU signals can cause process to be in this state.
                - SIGSTOP cannot be ignored or caught.
                - SIGCONT lets process continue.
        - **TASK_ZOMBIE**
            - Execution is terminated, but has to wait until parent receives info about child process.
    - **Run-queue**
        - Array like data structure to place active processes in.
        - Quantum stops, put back into run-queue, and start next process with highest priority.
        - Removed from run-queue when sleep, waiting for a resource, or are terminated.
    - **Priority Array**
        - Two in each run-queue (and 1 run-queue for each CPU), active and expired.
        - Provide O(1) scheduling using bitmaps to find highest priority task in system.
    - **Rescheduling**
        - Task can be rescheduled due to a signal, interrupt, or user calls sched_yield to go to next task.
        - Use timer ticks (each one = 10ms) to be able to change task when given time for it is up.
- **First in first out (FIFO)**
    - Process starts. ends when finished.
- **Round robin scheduling**
    - Each task gets equal quantum, meaning inactive tasks are also given same time as others.
    - Implemented in MP3, simple enough, can optimize it a bit though to ignore inactive tasks.

Discussion Week 10 - MP3.3

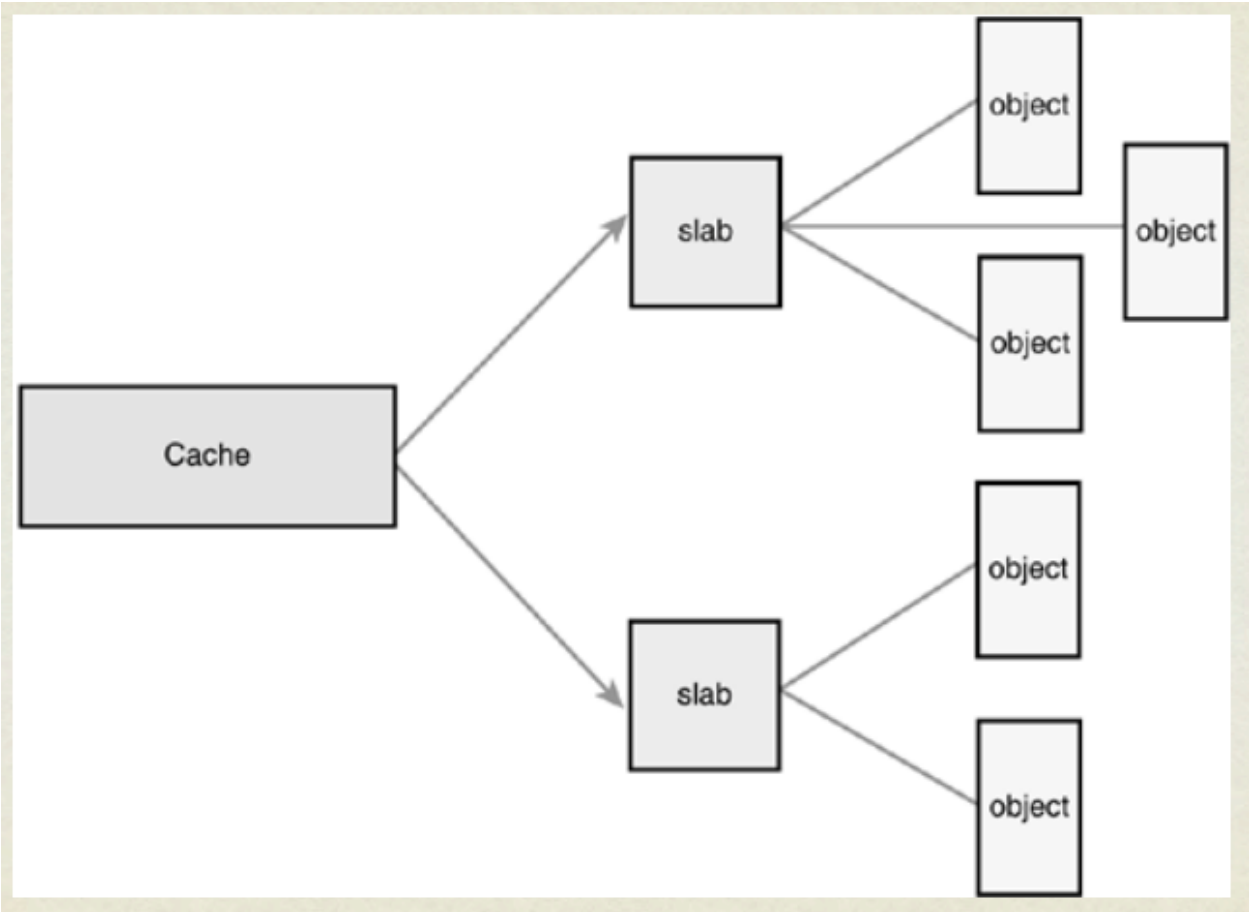
19 - System call linkage

21 - Memory allocation

- **Overview**
    - This lecture focused a lot on how linux handles memory allocation, and introduces the buddy system that can bring your exam grade down (this one's gonna be big).

- **kmalloc** - Used for a few small items (less than a page size type of stuff).
    - `void* kmalloc(size_t size, gfp_t flags);`
        - There are too many flags to list/remember, just know that the flags are to specify types of allocations/memory to allocate, conditions, etc. Can have multiple at same time by ORing.
        - Normal method of allocating memory in kernel
    - **gfp_t** is just unsigned int/long (depending on system).
    - Exponentially-sized slab caches used, contiguous in physical memory

- **slab cache -** Lots of items, and for reusing.
    - Frequent allocations/deallocations.
    - One cache per item type, w/ physical contiguous memory.
    - Constant initialization can slow down performance, instead of re-initializing constantly used objects, slab caches were made.
    - Helps alleviate memory fragmentation by being able to re-allocate certain caches when an object no longer in use.
    - **Constructor function**
        - Only called when new slab needed for an object type
        - `void* kmem_cache_alloc(kmem_cache*, gfp_t flags)`
        - `void* kmem_cache_zalloc(kmem_cache*, gfp_t flags)` (zeroes memory in object).
        - Flags passed to kmalloc iff (if and only if) new slab is allocated.

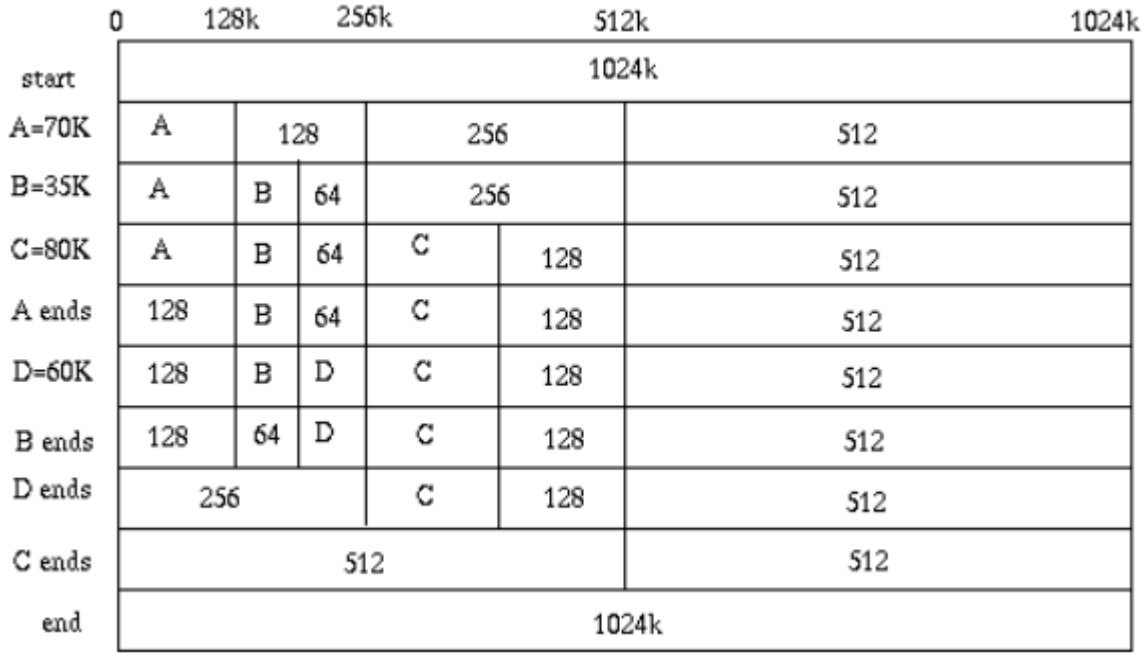


- **free pages** - Big contiguous region of memory.
    - `unsigned long ...page(gfp_t flags) (3 versions).`
    - Multiple of page size (4kB x86), contiguous, same flags as kmalloc.
    - Pages requested are log (base 2) of number of pages requested (3rd version, `__get_free_pages(gfp_t flags, unsigned int order).`
    - `void free_page(unsigned long)`
    - `void free_pages(unsigned long, int order)`
        - Only free pages you allocate, otherwise corruption can occur since the function doesn't check for you.
- **vmalloc** - Lots of virtual memory (doesn't have to be contiguous though).
    - `void* vmalloc(unsigned long size) (size is in bytes).`
    - If you don't need contiguous memory and need a lot of memory, use vmalloc.
        - Drawback to using this is performance, kmalloc is faster but

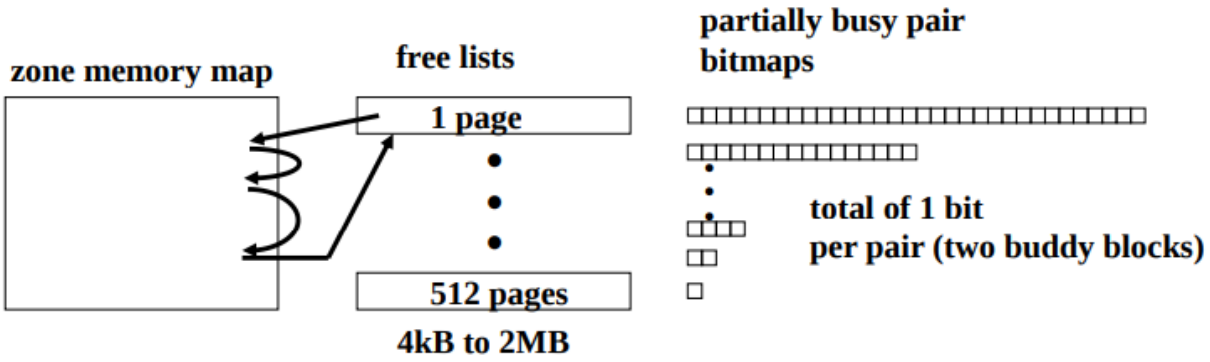
can’t be used for everything.

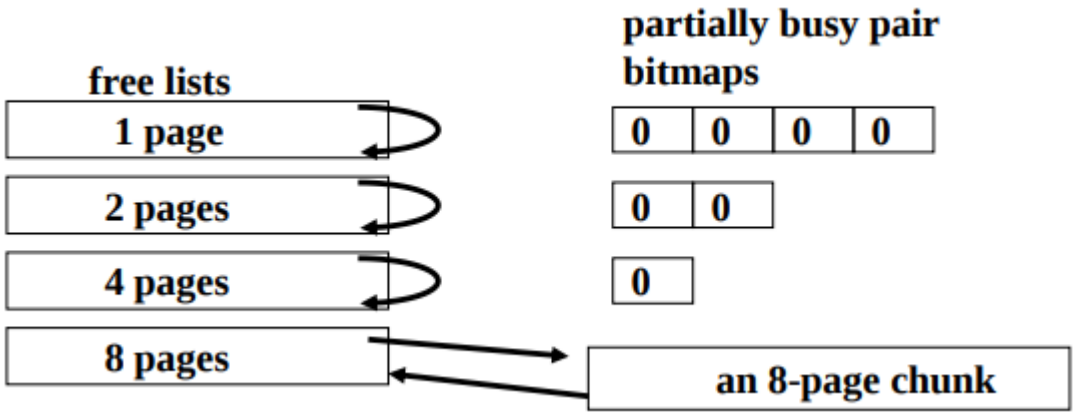
- If you don’t know whether it will go through something that needs physical contiguous memory, don’t use this.

- **Buddy System**
  - **granularity** - Smallest possible size something can be.
  - The many design problems that come with memory allocation (page alignment, contiguous or not, allocation granularity size, rewriting page tables, how to add information to allocated memory) bring the **Buddy System** into play.
  - **High Level Overview**
    - Entire memory space available for allocation treated as a single block, size = power of 2.
      - If requested size > half of initial block, entire block allocated.
      - Else, block is split into two equal ‘buddies’.
        - If requested size > half of new block, new block is allocated.
        - Else, repeat until smallest block >= request size is found and allocated



- Let's look at the slide pictures





- Each buddy system has own set of bitmaps to keep track of blocks that have been allocated.
  - If bit of bitmap is 0, both buddy blocks of pair are free or busy.
  - If bit is 1, one of the blocks is busy.
  - If both blocks are free, just a giant block of combined size.
- Two blocks are considered buddies if:
  - Located in contiguous physical memory.
  - Physical address of first page frame
- Reference: https://www.halolinux.us/kernel-reference/the-buddy-system-algorithm.html

- **Memory Zones**
    - Location of zone memory maps, having information about each type of memory (Note that zones are different depending on architecture, the following is based on x86 32 bit machines).

Zone	Description	Physical Memory
`ZONE_DMA`	DMA-able pages	< 16MB
`ZONE_NORMAL`	Normally addressable pages	16–896MB
`ZONE_HIGHMEM`	Dynamically mapped pages	> 896MB

    - Each zone struct has an array of free area structs.
        - Indexed by page order (0 to 10; 4kB to 4MB)
        - Keeps track of page usage statistics, free area, locks, etc.
    - Each free area struct includes a doubly-linked list of chunks of memory, and a count of chunks in it.

Discussion Week 12 - MP3.4, system calls

- Not much to this discussion actually, just read the slides.

22 - Memory management data structures - process address space

- **Overview**
    - Just going more in depth into material from lecture 21
- **Sharing (is caring)**
    - Task structs may share data with other tasks, including:
        - File pointer array
        - Filesystem info
        - Signal Delivery and handlers
        - Memory map
    - Having shared memory makes passing data from one program to another more efficient.
        - Memory is also shared between multiple threads for programs with multi-threading.
    - `do_fork` has clone flags for different tasks to share certain things like the ones mentioned in the first bullet points.
        - Threading is different from fork since fork creates a copy of the process, while threads in 1 program are all in 1 process.
        - Switching between threads > Switching between tasks made from fork/original due to having to do a full context reload (like TLB flushing) with fork.
- **Memory Maps**
    -

23 - Abstracting devices: block and character devices; device drivers

- **Overview**
    - Focuses on Linux abstraction on device drivers, along with an example driver made be Lumetta (crazy guy)

-

Discussion Week 13 - MP3.5, Scheduling

24 - Driver development process and detailed example

25 - Detailed example of driver development, continued

Discussion Week 14 - MP3.5

26 - Signals: user-level analogue of interrupts, controlling behavior

Glossary - Terms and Definitions