ECE391: Operating Systems

Pradyun Narkadamilli

Last Updated: May 24th 2023

Contents

1 Disclaimer

Keep the following in mind when using this document:

> This is not an official course resource, but my own notes from when I took this class. As a result, it may *not be* comprehensive of the content that you may be tested on, and its MP sections will not perfectly match you.
>
> Furthermore, in the event that you find an inacurracy in this document, please notify me *directly*, and I will update it ASAP. Along those lines, I'd also like to recognize that this document may have mistakes - please make sure that you do not treat this as your only resource.

Also keep in mind that this is written such that it *directs* you to think about what you need to know for the exam. As such, its depth is not as comprehensive as other materials.

2 Review Material (220)

- **Addressability** - granularity of data that is associated with an address in the architecture. e.g - 16 bit addressability imputes that there are 16 bits stored at each address in memory. You can do a 32-bit read from an address, but it'll really be pulling from multiple addresses.
- **representation** - a way of encoding things to be represented as bits. ex: representing an integer as two's complement
- There is a duality between arrays and pointers - you can treat one as the other.
- **Primitives** - signed/unsigned integers, single/double precision `float`, pointers.
  - 8 bit `char`, 16 bit `short [int]`, 32 bit `int`, 32/64 bit `long [int]`, 64 bit `long long [int]`
- **scope** - what areas of program can access program, whereas **storage class** is where to store it. Both depend on where var is defined and if `static` qualifier is used.
  - `static` stores variable in static data region
  - `automatic` - variable is stored on the stack
- if defined outside of a function, you get a **global variable**
- Variable **Definition** creates a new variable. Variable **Declaration** makes a variable that is defined elsewhere accessible.
  - Variable declaration is preceded by `extern`
- Most ISAs use 8-bit addressable memory but require 32 bit load/stores. Not possible to create an unaligned load - must add padding bytes. Don't make assumptions about field offsets and size for a structure in your code - you don't know what's going on under the hood!
- Enum Syntax - good to know for personal reference

```
typedef enum {
    CONST_A, // 0
    CONST_B, // 1
    CONST_C, // 2
    NUM_CONST// 3
} constant_t;
```

- C arguments are all passed by value
- `#define` is just text replacement
- **Strongly Typed Languages** : won't let you treat a variable as some other type - i.e Java. You cannot change the type of a datum.

- **Shadowing**: having two variables with the same name but different scopes can make data inaccessible in internal scopes, and in turn obfuscates your code. It can be useful for doing some monkey style stuff though.
- `static` - dumps var into static data region.
- packed `structs` - does not honor 32-bit alignment for individual attributes. Attributes can cross address boundaries (no padding bytes)

3 x86 Architecture

- **CISC** - Complex Instruction Set Computing
- **RISC** - Reduced ISC
- x86 is by modern standards a badly designed ISA
  - Why do we still use it?
- Modern flavors of x86 (Intel Architecture 32 [IA32]) have 8 32 bit integer registers. The "E" prefix stands for "extended".
  - These can be accessed in subregs too - the 16 bit register will access the LSBs, and the 8-bit registers subdivide that 16-bit space further.

Register	Purpose	16bit	8-bit (high)	8-bit (low)
EAX	Accumulator/return value (arithmetic)	AX	AH	AL
EBX	base address of array in memory	BX	BH	BL
ECX	count (loop iterator)	CX	CH	CL
EDX	data (ex: second operand for binary op)	DX	DH	DL
ESI	Source index (array access or string copy)	SI		
EDI	destination index (string copy or array access)	DI		
EBP	base pointer (stack frame)	BP		
ESP	stack pointer (top)	SP		

- Special Registers: `EIP` stores the instruction pointer, `EFLAGS` stores the flags (condition codes et al)
- Remember - register names need to be prefixed with `%` in assembly.
- x86 ISA supports architecture-level operation for 2's complement and unsigned integers (32b, 16b, 8b), single/double precision floating-point, 80-bit Intel floating-point, strings, BCD.
- x86 memory is byte-addressable, uses 32-bit addresses
  - Few machines fully use this address space (equates to about 4GB)

- x86 is **little-endian**. This means that each individual memory address retains the order of its own bits, and the least significant segment is stored in the smallest memory address. View the below example for how it would work in a 4-bit addressable system.
  - `Store the 32 bit value x0A0B0C0D in address 0x00`

    ```
    0x00 | 0x0D
    0x01 | 0x0C
    0x02 | 0x0B
    0x03 | 0x0A
    ```

- Some x86 I/O is memory-mapped, and some is done with a separate port space and the `IN OUT` instructions.

3.1 x86 ISA

- **Arithmetic**: `add`, `sub`, `neg` (negate), `not`, `inc` (increment), `dec` (decrement)
  - Takes two arguments, where the second one is treated as the destination
  - EX: `xorl (%eax, %edx, 4), %edx` will perform xor w/ two 32 bit values such that `edx <- edx ^ M[eax + 4*edx]`

- **Logical**: `and`, `or`, `xor`
  - same args as above

- **Funky**: `shl` (shift left), `sar` (arithmetic right shift), `shr` (logical right shift), `rol` (left-shift w/ wraparound), `ror` (cycle bits to the right)
  - same arg templating as above

- **Data Movement**: all load/stores are unified into `mov`, the turing-complete disaster of an instruction
  - same arg templating as above.

- **Multiplication/Division**: requires that one operand be in `EAX` or its subregisters (`AX`, `AL`)
  - General form: one of the arguments is implicit, so the command itself has one argument in assembly. This can either be a register or memory. e.g `idiv %ecx` or `mul 3(%ebx)`
  - `MUL` (unsigned mult) and `IMUL` (signed mult) . High bits stored in `EDX`, bottom stored in `EAX` (or `DX:AX` or `AX`).
    - `IMUL` also allows for two and three operand formats. In these forms, high bits are discarded
      - `imull %ebx, %eax` where `EAX <- EAX * EBX`
      - `imull $1000, %ebx, %eax` where `eax <- ebx*1000`

        - three-operand form requires first arg to be an immediate/const
    - `DIV` and `IDIV`. Dividend placed in `EDX:EAX` (or `DX:AX` or `AX`). After `IDIV`, `EAX` (or `AX` or `AL`) has quotient and `EDX` (or `DX` or `AH`) holds remainder.
        * Exception generated if remainder overflows the destination reg.

- Instructions are suffixed with a letter indicating operand lenguth (`bwl`) or two letters if you have different data types for the source and destination. (ex: `movwl` or `movzwl` for zero extension, or `movswl` for sign extension)
- Immediates and labels must be prefixed with `$` to get their numerical value, otherwise `M[label]` is used. This rule is broken in the "immediate" addressing mode - the number or label should be written on its own in front of the first parenthesis.
- Special Conversions exist for EAX reg:
    - `CBTW` converts signed byte `AL` to word `AX`.
    - `CLTD` converts signed long EAX to double word `EDX:EAX`.
    - `CWTD` converts `AX` to `DX:AX`. Be careful with that one
- **I/O instructions** - used to communicate with the IO port space. Require specific registers - data must be in `EAX`, port number can either be an immediate or have to be in `DX`.
    - `inb $0x40, %al` imputes `AL <- P[0x40]`
    - `inw (%dx), %ax` imputes `AL <- P[DX], AH <- P[DX + 1]`
    - `out[bwl]` follows the same convention.
    - Ports address space, like memory, is byte-addressable and little endian.
- **Stack** : `push` and `pop`
- `CLI=/=STI` - `CLI` masks the `IF`, `STI` unmasks interrupts.
- `LEA` - given a memory reference, instead of accessing memory it copies the memory address into destination register
- `CMPL A, B` will do `B sub A`. The `jmp` variant after this instruction will follow the same ordering in the comparator
    - Despite our usage, `CMP` has the same 2-operand compat as everything else.
    - EX: `CMPL A,B` followed by `ja ADDR` is checking for an unsigned `B > A`

- **Sign Flags** - unlike LC-3, multiple may be high at the same time
  - sign flag (SF) is high if last result had a negative integer.
  - zero flag (ZF)
  - Carry Flag (CF) - did last result require a carry or borrow? also used to hold bits shifted out. Lots of instruction-specific effects
  - Overflow Flag (OF) - did last op overflow when interpreted as a 2's complement operation
  - Parity Flag (PF) - even or odd number of 1's in last result
  - Basically all the instructions in the first 4 categories above affect all conditions, with below exceptions
    - No Changed Flags: `mov`, `lea`, `not`, `in`, `out`
    - Only OF/CF: `ror`, `rol`
    - All Flags but CF: `inc`, `dec`
- **Branch Instructions**
  - `jo` : **jump overflow**: check if OF is set
  - `jp`: **jump parity**: check if PF set
  - `js`: **jump sign**: check if SF set
  - `je`: **jump equal**: check if ZF is set
  - `jb`: jump below: unsigned comparison `<`. checks `CF`
  - `jbe` : jump below equal: `CF` or `ZF`
  - `jl` : jump less: signed comparison `<`. `SF` != `OF`
  - `jle`: jump less equal : `SF ! OF`= or `ZF`
  - All jumps above can be inverted by dumping `n` after `j`.
  - Common inverted aliases:
    - Unsigned `>=` : `jnb -> jae`
    - Unsigned `>` : `jnbe -> ja`
    - Signed `>=` : `jnl -> jge`
    - Signed `>` : `jnle -> jg`
  - **NOTE:** To set the flags based on a `MOV` or `LEA` or something else that does not set flags, use `CMP` to set all flags (subtracts first argument from second, does not store result) or `TEST` (performs an `AND`, clears OF/CF, `SF, ZF, PF` are set as needed).

- **Other Control Instructions**
  - `CALL` is used to call a subroutine, pushes return address to stack before changing instruction pointer. Use `RET` to exit the subroutine.
    - Indirection can be used with Call by using deref operator
    - EX: `call *10(%eax, %edx, 2)` represents `(push EIP), EIP <- M[EAX + EDX*2 +10]`. Removing the `*` just jumps to the computed mem address.
  - `JMP` is an unconditional jump. Can use same memory addressing shenanigans as `CALL`.

- Use `#` for comments
- Assembler supports `.GLOBAL` and `.EXTERN` to declare symbols visible externally or to be defined externally, like C's keywords.
- `.SPACE n` will allocate `n` bits of empty space. `.STRING "cool beans"` puts the string in memory. You can use similar directives to populate a memory address with a value at assembly time.
  - another example: `.byte 12, -15` will populate two bytes in memory with `12` and=-15=.
- x86 originally only used a separate serial port for I/O, but when high-speed communication was needed for graphics card, memory-mapped I/O was introduced. x86 now uses a mix of the two.

3.2 Calling Convention

- Function parameters are pushed onto stack in x86. Arguments are pushed from **right to left** so that the first argument is placed at the top of the stack. This way, the /n/th argument can be accessed relative to the stack pointer without needing to track where arguments start or how many total arguments there are.
- For pointers and integers no more than 32b, return placed in EAX. Values of length up to 64 bit can be returned in the `EDX:EAX` form.
- Floating point values are returned on top of floating point stack.

3.2.1 Caller vs Callee

- Most registers are caller-owned, and thus must be callee-saved. `ESP` and `EBP` (stack/base pointers), along with `EBX`, `ESI`, and `EDI` must be callee saved.
- `EAX`, `EDX`, `ECX`, and `EFLAGS` must be caller-saved, as system operations/general runtime are expected to modify these registers in a subroutine/function
- **Caller**

  - Pushes function params (called **formals** in function) onto stack,

then executes the 'call' instruction to backup `EIP` and jump to function.

  - After `RET`, the Caller must remove the function params from the

stack - this can be done with an `ADD` or a bunch of `POPL` instructions.

  - (potentially) stores EAX into a local variable

- **Callee**
  - push `EBP`, `ESP` gets copied into `EBP`
  - Save any necessary callee-saved regs
  - once func ends, teardown starts. pop off anything on stack, then use `LEAVE` to restore `EBP` and pop the record. use `RET` to restore `EIP` and pop that record.

4 SysCalls, Interrupts, Exceptions

- **System Calls**: nearly identical to procedure/subroutine calls. Calling convention is still used. SysCall will place the processor in privileged/kernel mode for the execution, and instructions implementing the call are considered part of the OS.
  - Generated by `INT` instruction
  - EX: print char to console
  - synchronous, expected
- **Interrupt**: asynchronous interruptions generated by other devices. Triggers an **ISR** (interrupt service routine)
  - ex: packet arrived from network card
  - asynchronous, unexpected
  - Usually processed between instructions, not immediately when encountered. Processor can choose to respect atomicity.
  - Harder to determine when to service an interrupt in a pipelined design, when many instructions occur simultaneously
- **Exception**: Processor encounters unexpected opcode or operand
  - ex: undefined instruction, divide by zero instruction
  - Usually causes program termination

  - synchronous, but unexpected (happens for a particular instruction rather than just out of nowhere in terms of program execution)

- **Handler**: subroutine associated with the interrupt/exception/syscall number in **vector** (or jump) **table**.
  - x86 uses a single unified table for all 3 types of OS calls, called the **Interrupt Descriptor Table**
- In LC-3, because there is no pipelining interrupts are processed during the first load state in the FSM. Checks `INT`, if high it initiates an interrupt handler call.
- When interrupt is requested, an 8b interrupt vector is supplied to index IDT.
- x86 can block all interrupts from `INTR` input if **the IF (interrupt enable) flag in EFLAGS** is low.
  - `STI=/=CLI` instructions change `IF` value.
  - Low `IF` masks all interrupts
- x86 has *another* input called `NMI`, for **non-maskable interrupts**
  - You used these in the NES!
  - These tend to indicate more serious conditions, like memory parity failure, low battery, etc.
- **IDT** has 256 entries indexed with vector number
  - contains pointer to handler, privilege level, and some other elements
  - `0x00-0x1F` vectors defined by Intel
  - `0x20-0x27` vectors defined by primary PIC
  - `0x28-0x2F` vectors defined by secondary PIC
  - `0x30-0x7F` APIC vectors for device drivers
  - `0x80` system call vector
  - `0xEF` local APIC tiemr
  - `0xF0-0xFF` symetric multiprocessor (SMP) communication vectors
- **IRQ** - short for "Interrupt Request"
- Handler code for a given device's interrupt interacts using processor's I/O ports

  - This interface can either be organized through special registers/ports with **independent I/O**
  - Alternatively, simply use the same `load=/=store` instructions as normal but with special addresses mapped as necessary. **These addresses cannot be accessed during normal processor operation.**

- Interrupts are asynchronous and unexpected - this can lead to problems with shared data/resources (memory, registers, etc.)
  - *All* registers should be saved to stack and restored before returning from interrupt. Callee-save everything.
  - Should avoid overwriting memory locations in-use by interrupted process
  - Less obvious shared resources: data structures in memory used to communicate between interrupt handler and processor, flag/status registers.

- If the interrupt handler triggers when the shared data structure is incomplete or unusable, it may not be able to execute correctly
- `volatile` - assume that data has changed on every access, avoid wonky optimizations when trying to monitor shared data that you know might change.
- security is also an issue when dealing with both user/kernel programs. using a program's stack during interrupt routing can be bad
  - there may not be enough space, stack pointer may not be pointing to the stack at that point of time
  - data from handler and any calls made are still there, now visible to lower-privileged program.
  - To avoid issue, **many ISAs use a separate stack for privileged mode/operating system**

5 Critical Sections

- **Atomicity**: set of operations is executed as if it were a single operation. Interrupts are either executed before or after.
- **Race Condition**: when multiple processes attempt to access data at the same time and you cannot guarantee which one runs first.
  - some race conditions can produce bugs if the relevant processes do not execute in an order that they are expected to run in. Race conditions are hard to track down because they may occur infrequently.

- **Critical Section** :set of operations need to be executed without stopping.

  - in a uniprocessor, boundaries can be created by masking interrupts
  - in high level langs, mark all memory at critical section boundary as volatile
  - *Minimize Length*: masking interrupts for too long will cause slowdowns in the rest of the processor, reduce throughput, and can cause loss of information (in extreme cases)

  Using the above approach *could* work for normal interrupts in a uniprocessor, but does not immunize the critical section to **NMI** handlers.

- `IF` flag operates on one processor at a time - an interrupt handler in another processor could still trigger.
  - Too slow to just shut down interrupts in every processor, do the critical section, then revert. We need a better way.

- **SMP (Symmetric Multiprocessor)**: Multiprocessor where all processor cores have identical (symmetric) access to memory banks and I/O.
  - Uncached access time from any processor to a memory location is identical.
  - Data can be cached near a processor for performance

- **Spin Lock** - program will idle, or 'spin', until it can consume the lock.
  - Program will atomically attempt to acquire the lock by atomically changing lock to held state. Only one program can hold lock at a time.
  - Only owner of lock should unlock it
  - these should be used for shared resources/data **that an interrupt handler may try to access**
  - Linux spinlocks represented by `spinlock_t` structure
    - Statically allocated spinlocks are good to go after var init.
    - `malloc`'d spinlocks require a call to `spin_lock_init` after allocation

  ***** Basic API :CUSTOM$_{\text{ID}}$: basic-api

Function	Purpose
`void spin_lock_init(spinlock_t* lock)`	initialize a malloc'd spinlock
`void spin_lock(spinlock_t* lock)`	obtain spinlock, return when got g
`void spin_unlock(...)`	release spinlock, only call if lock al
`int spin_is_locked(...)`	check if held
`int spin_trylock(...)`	attempt a lock, but don't spin
`void spin_unlock_wait(...)`	wait until spinlock is available - do
`void spin_lock_irqsave(..., unsigned long& flags)`	save processor status in `flags`, ma
`void spin_unlock_irqrestore(..., unsigned long flags)`	release spinlock normally, set proc
`void spin_lock_irq(...)`	unconditionally mask interrupts, t
`void spin_unlock_irq(...)`	release spinlock normally, uncondi

- Normal lock/unlock + misc. testing functions are mostly useful on SMPs.
- Use interrupt masked calls - usually `irqsave/irqrestore` - to protect crit. sections.
- For `irqsave`, processor status is saved by pushing `EFLAGS` to stack, then popping it into another register and saving that to the `flags` variable
- **deadlock**: when multiple processes hold information that another requires, but also require information that another holds, no process will relinquish control over their resources. This causes the machine to freeze up.
  - EX: if an interrupt handler interrupts a process that holds a lock that the handler wants, then the handler will stall out.
- `spin_lock` and `spin_unlock` are =NOP=s on a uniprocessor. For uniprocessors, masking interrupts is basically all you can do.
- **Sempahore**: generalizes concept of a lock to allow fixed number of programs to enter set of critical sections at the same time (basically represents a finite amount of resources)
  - Proberen (P) is the **down** operation, which tries to decrement semaphore atomically
    - When semaphore is 0, programs trying to claim semaphore will block
  - Verhogen (V) is the **up** operation, which will increment semaphore to free up an abstracted resource
  - Program waiting on a semaphore allows other programs to execute while waiting - the thread waiting for the semaphore is slept, then woken when a semaphore is up'd

    SEMAPHORES SHOULD NOT BE USED IN CODE THAT SHARES DATA W/ INTERRUPT HANDLERS

    - don't use semaphores in sections holding a spinlock - you essentially sleep the thread while trying to claim a semaphore, so every other process waiting on the spinlock just spins indefinitely because you don't free the spinlock
  - Since semaphores allow other code to run when waiting, semaphores can protect longer critical section without the same slowdown concerns.
- Code that manages data shared within the user space should use semaphores
- **Linux semaphores** are optimized for uncontended access
  - For this class: ***mutexes*** *are simply binary semaphores.* They allow critical section access to one program at a time. (short for **mutual exclusion**)

5.1 Basic Semaphore API

Function	Purpose
`void sema_init(struct semaphore* sem, val)`	inits a dynamically allocated semaphore
`void init_MUTEX(struct semaphore* sem)`	initializes a dynamic semaphore to 1
`void init_MUTEX_LOCKED(...)`	initialize dynamic semaphore to 0
`void down(...)`	wait on a semaphore, return after success (sleeps thread)
`void up(...)`	signal a semaphore, wake any "down" threads if semaphore

- Static initialization/allocation of a semaphore:

```
static _DECLARE_SEMAPHORE_GENERIC (name, val);
DECLARE_MUTEX(name);
DECLARE_MUTEX_LOCKED(name);
```

- Semaphore calls do *not* become `NOP` on a uniprocessor.
- **Reader/Writer Locks**: The main conflicts you run into are read-write and write-write. Read-write locks add a level of complexity, but make read-read accesses allowed
  - Can be implemented with both a semaphore and spinlock
  - The base spinlock API is `read_lock(rwlock_t* rw)` and `void write_lock(...)`. Usual `unlock`, `irq`, `irqsave`, and `irqrestore` variants are offered, just like spinlock. Only `int write_trylock(...)` testing function is offered though.
    - R/W spinlock implementation is fast, but does not prevent **starvation**. If you have a constant supply of readers, a writer may not ever be able to get write access (to avoid read-write conflict)
  - Semaphore implementation has R/W exclusion properties, but also scheduling properties of semaphores. Program trying to get reader/writer access can yield processor to another program (normal semaphore stuff)
    - `void init_rwsem(struct rw_semaphore* sem)` to initialize dynamic sem. `up/down` variants offered for both `read/write`. EX: `void down_read(struct rw_semaphore* sem)`
    - Sempahore implementation does not run into starvation problem - a waiting writer can block new readers.
- Use spinlock implementations of *mutex* and *reader/writer* if sharing data with interrupt handlers. If only sharing data with system calls, use semaphore implementations.
  - If the process can be interrupted, you generally want to use either `irq` or `irqsave` variants. If it's the highest priority with data access, a normal `spin_lock` is fine

5.1.1 Decision Tree for what Mutex to Use

type of code entering crit section	data shared with…	for mutex, use…
system calls only	other system calls	`up=/=down`
system calls only	interrupt handlers	`spin_lock_irq`
both system calls and interrupt handlers	both system calls and interrupts	`spin_lock_irqsave`
interrupt handlers	system calls	`spin_lock`
interrupt handlers	higher priority interrupt handlers	`spin_lock_irqsave`

5.2 Interrupt Control

- You can't just directly bridge all the devices outputting interrupts to `INTR` and `INTA'` (active low ack) pins.
- **Programmable Interrupt Controller (PIC)**: x86 uses the 8259A PIC to handle prioritization and arbitration between different devices' interrupt signals.
  - LC-3 uses a simple priority encoder to decide between different interrupts
  - 8259A is asynchronous of the processor clock - its data bus transactions are driven by processor's control signals
  - 8259A has 8 interrupt (IR) lines, lower numbers have higher priority.
  - **Control Flow for a PIC Interrupt**
    - If a new interrupt (with higher priority than any in-service interrupts) is signalled, PIC will raise `INT` pin
    - processor strobes (lowers) `INTA'` pinto acknowledge the `INT`, strobes repeatedly requesting the interrupt vector over `D` line. `INTA'` essentially creates cycles for the PIC
    - PIC marks this interrupt as in-service internally
    - Eventually, interrupt handler will signal **EOI (end of int.)** by writing to the address `A` and data (`D`) inputs of PIC with an `OUT`. (PIC is communicated over port space, not memory map). At this point, PIC removes interrupt from in-service mask.
      - If EOI never signaled, PIC will indefinitely mask same/lower-priority interrupts
  - Other pins on PIC: `CS'` is low when processor writes to address `0x20` or `0x21` (PIC is mapped to those ports). The LSB of that address is passed into the A pin. `RD'` and `WR'` are from processor's point of view (if `RD'` is low, processor is expecting a write from the PIC). `A` bit is used to differentiate command vs data.
  - Multiple 8259As can operate in a hierarchy by cascading them
    - Set `SP'` (slave program) low to enable slave mode

    * 3-bit `CAS` bus allows primary PIC to send identifier number of which PIC should be active to respond to the processor when `INTA'` is strobed (tells which secondary to write to bus)
        - Generalization (short answer): ***CAS bus is used whenever we want a secondary PIC to respond to `INTA'`***.

- Secondary PIC will be maped to `0xA0/0xA1`
- `void init_8259(int auto_eoi)` initializes the PIC, the one input allows for use of 8259A generating Auto-EOI (rather than interrupt handler sending to PIC). Auto-EOI not used in Linux
- Only one processor should initialize PIC atomically (use a critical section for this)
    * Mask all interrupts (on both PICs), initialize, restore mask settings, release lock and restore IF flag
- All interrupts are masked on the PIC by default. By writing a byte to the `0x21` or `0xA1` port, PIC's interrupts can be masked or unmasked.
- Initialization sequence of PIC requires 4 initialization control words (ICWs) to be sent. These are sent to the first PIC port (`0x20` or `0xA0`)
    * First word tells PIC to initialize, tells it to use edge-triggered signals (or level-triggered), and whether to use cascade mode, indicates 4 control words total
    * Remaining ICWs are written to second port (`0x21` or `0xA1`).
    * High bits of interrupt vector region sent in ICW2
        - Vectors `0x20 - 0x27` used by first PIC
        - Vectors `0x28 - 0x2F` used by second PC
        - ICW3 specifies which pin is used to connect the primary PIC to a secondary
        - ICW4 specifies 8086 protocol, normal EOI signaling, etc.
- Linux abstracts interrupt controllers with **jump table** - other code can perform generic operations (startup, shutdown, enable, disable, etc.) without knowing how exactly to perform that operation on the particular PIC used. This allows code to be more general
    * `startup` unmasks all the interrupts on 8259, `shutdown` is called when last handler removed from interrupts (masks interrupts)
    * `disable/enable` allow nested disabing and re-enabling of active interrupts
        - for 8259, identical to startup/shutdown function
    * `ack` called to acknowledge receipt of interrupt, then interrupt handler is executed. `ack` for 8259A masks interrupt on PIC then sends EOI
        - while interrupt is not marked in-service, the mask will ensure that further interrupts do not trigger

    * `end` is called to end interrupt - on 8259 `end` will unmask the interrupt on appropriate PIC
    * `set-affinity` will specify which CPUs in SMP can execute an interrupt
        · This can help restrict an interrupt to execute on only one processor - makes data sharing easier, but can give a performance hit

6 MT1 Lecture Review

- x86 can take up anywhere from 1-16 bytes per instruction (variable length)
- 32-bit, byte addressable
- Big endian - MSB segment is stored in lower memory address
    - Ex: for 0x1234, you would have:

      ```
      Address 0x0: 0x12
      Address 0x1: 0x34
      ```

      So you encounter values in memory in the order you read them.
    - Little endian is the opposite.
- data type at the end of the instruction (like `ORL`) is usually optional - can generally be inferred by the assembler
- Immediates marked by a dollar sign
    - usually up to 32 bits (this is why labels can be used as an immediate)
- Displacement formatted as `D(A, B, scale)` where `A/B` are registers. Memory address is computed as `D + R[A] + R[B]`. `scale` can be 1,2,4,8 (defaults 1). Reg B can be anything except for ESP. D is just a normal immediate
    - Wrap register in parenthesis to use its contents as a memory address
    - Prepend with asterisk to dereference
    - `CF` is used for unsigned <, signed `<` uses `OF ^ SF`
- `CALL` pushes current `EIP` value to stack, loads arg into `EIP`. `RET` undoes this op, assuming that `ESP` points to the top of stack
- `XORL REGA, REGA` can be used to clear `REGA`
- C pushes its arguments from right to left

  - order can be language-dependent

- More calling convention
  - EAX/EDX can be clobbered by subroutine - along with `ECX` and `EFLAGS`, these are caller-saved
  - stack structure (`ESP` and `EBP`) along with other regs (`EBX`, `ESI`, `EDI`) are all callee-saved
  - `EAX` used for all return values up to 32-bit, `EDX:EAX` will be used for 64 bit return values.
  - **Callee Sequence**: save old base pointer, update `EBP` to `ESP`, save callee registers that you are using, make space for local vars, do function body, do stack teardown, restore registers, reload `EBP` with `LEAVE`, return.
  - Benefits of unconditionally saving all callee-saved registers - if you ever start using another register or change how your code works, you can basically do whatever the heck you want. You aren't gonna break anything.

- **Roles of System Software**
  - TL;DR (Exam answer): **Abstraction, particularly hiding asynchrony**
    * Hides away hardware complexity and asynchronous nature of CPU/device interaction. Provides abstractions that are more convenient to work with in software. Simpler, more powerful interfaces.
    * Abstraction can help us make more generic code that we can use between systems, as those abstractions can be redefined in different environments
    * EX: As a developer, the `gets` function in C just takes input. But at the hardware/CPU level, there are many asynchronous keystrokes and system-level instructions that need to be run to actually populate the buffer with different keystrokes. The program is put to sleep as well.
  - **Protection**
    * reduces/eliminates chance that program will accidentally(or maliciously) destroy results/data of another program
    * *My paraphrasing* :Ensures that multiple programs to not interfere with each other's data, causing adverse effects. If a program mucks up, it mucks up *itself*, not your entire system.
  - **Virtualization**
    * *Big thing*: illusion of multiple/practically unlimited resources.
    * why should you as a developer worry about how much memory your system has left or how to work with the hardware? there are system calls that can handle that for you, and all you are left with is nice shiny boxes to put data into. Maybe it's a string. Maybe it's a picture. Maybe it's your credit card number so you can buy another keyboard!

- Subroutines allow programmer to encapsulate common operations
- Virtual machine allows you to test code that may have bugs without crashing your own system, without risking harm to the source code, etc.
- In the operating system you want to provide an interface including common operations, but you want to avoid re-linking programs or relying on everyone having same OS version
  - You can add indirection for this by maintaining a vector table - the **Interrupt Descriptor Table**
  - Instead of calling functions, you call *handler numbers* which will use the vector table to jump. Call handler number with the `INT` instruction.
  - `INT` is called a "trap" - lot of its complexity comes from crossing the protection boundary and calling "privileged" code. Also called a **system call**
- If software does a dumd, exception triggers
- **Port I/O** is technically called **Independent I/O**
  - has gotten phased out for **Memory-Mapped I/O** - memory-mapped is much faster as a bus. use it with better/faster devices.
  - Newer, faster devices tend to use Memory I/O
  - Port space is 16-bit, byte-addressable and little endian (just like the rest of x86)
- Two execution contexts - user space and kernel space.
  - There is a protection boundary in between - that is crossed by using system calls (`TRAP`, handlers) to execute *actions that the user wants* with *the privilege of the kernel space.*
  - Split Kernel space into "top" and "bottom" halves.
    - Top half interacts with user space
    - Bottom half interacts with interrupts, etc.
- `INT` calls are synchronous, interrupts are inherently asynchronous. Top half deals with synchrony, bottom half deals with asynchrony
- Critical sections should be short so that you can avoid delaying device service by interrupt handler, don't "lose" IRQs, etc. long delays can crash system (swapdisk driver timeout)
- **Conservative Metric**: two functions conflict only if read-write or write-write conflict to piece of shared data.
  - If neither of these is true, then interleaving same as serial execution

  - If any two function conflict, they must be protected by same lock

- **Deadlock** - a process is holding onto a resource and will not let go of it, so the current process cannot proceed, nor can another process that requires the same resource. Process has halted, machine has **frozen**
- **Livelock** - a process holds one resource, attempts to hold another, but releases its first resource since another process is holding it. If multiple processes do this repeatedly in tandem, then neither process ever moves forward. So while the machine is executing code, functionally it does not proceed. **Process does not make progress, but it is still running.**
- For multiprocess deadlock/livelock issues, partial lock ordering can arbitrate this issue.
- Blocking vs Non-Blocking refers to the activity of a thread in scheduler
- **Variable Lock**: Associating spinlock with a single variable rather than a full struct
- GNU and AT&T syntax **are the same**

7 MP Review (MT1)

- Virtual memory - lets each user program to think it has separate memory space
  - This is why we need `copy_to/copy_from_user` functions when accessing user memory - to translate the virtual user memory address to an absolute memory address that the kernel space can copy to/from.
- Use `malloc=/=free` just like in C to allocate/free memory in MP assembly
- Text-Mode Video: each char on text display takes 2 bytes in mem. High byte attribute (color) and low byte for actual char itself.
- Linux drivers let it treat all devices as a regular file. Files stored in `/dev/` are a bunch of devices linux is dealing with.
  - ex: first serial port is `/dev/ttyS0`
  - since abstraction is a file, linux drivers must support usual file ops
- RTC - real time clock. Can generate interrupts at a configured frequency.
  - Linux programs use this to perform timing-critical functions (ex: flashing our fish!!)
  - RTC driver uses `open` and `close` ops as initalization/cleanup
  - use `read` or `poll` file ops to wait on four bytes of data from `/dev/rtc` that are released every RTC interrupt. This way, you determine *when* the interrupt has been generated

- `ioctl(int file_descriptor, int IOCTL_COMMAND, unsigned long data_argument)` - from user space, `ioctl` will perform the `IOCTL_COMMAND` on a device. This device is passed in as `file_descriptor`, the return value of an `open` call to access a device in `/dev/`. `data_argument` is simply an argument that can be used within the `ioctl` itself. In MP1, this is a pointer to a struct used for storing data we needed for our different `ioctls`.
- **Tasklet** - way for the interrupt handler to defer work until after kernel is done with time-critical tasks, about to return to user program
  - interrupt handler will do all the time-critical things in the main handler, then schedule a tasklet to run before returning to user program -this tasklet will do all the heavy I/O and computation
    - Helps keep interrupt handlers short to prevent backup/delay of service for other IRQs
  - Operating system can keep all interrupts unmasked during tasklet execution
  - *In our MP*: RTC interrupt handler will trigger at configured time interval, signal `EOI` to the PIC so it can continue servicing interrupts, then schedule our tasklet to run so that our fish will update. During this (relatively) I/O-intensive process, interrupts are allowed.

8 Interrupts

- **Interrupt Chaining**: a handler invoked by the hardware interrupt may invoke another interrupt handler.
  - Older systems did this with a `JMP` at the end of the handler to jump to the old handler - this made interrupt chains very brittle. Made cleanly removing a single handler impossible.
  - If multiple systems are connected to a single interrupt line, then each device has to be queried to see if it should be serviced. You then need to invoke the relevant devices' handlers.
    - Device interrogation slow, this kind of chaining rare
- **Soft Interrupts**: software-generated interrupts. Operate at a priority in between HW interrupts and programs.
  - Hardware interrupts' behavior is not *fully* dependent on the hardware. Some of its functionality can be deferred into soft interrupts to allow more hardware interrupts to be serviced.
    - Ex: after network packet extracted from device, it needs to be examined to see which program its data should be forwarded to. This work can be deferred, and shouldn't be prioritized over servicing other hardware.

8.1 Quick IDT Refresher

Vectors	Function
0x00 - 0x1F	defined by Intel (Exceptions)
0x20 - 0x27	primary PIC
0x28 - 0x2F	secondary PIC vectors
0x80	System Call vector

8.2 Linux Interrupts

NOTE: This section is a bit over-detailed. FOr the most part, you DO NOT need to know the linux-specific implementation details. Ignore any overly specific flag-based information

- IDT has 256 entries, invoked when you receive an interrupt vector corresponding to entry, etc. etc. MT1 Content
- Linux abstracts PICs as a jump table, each entry is a separate functionality
  - Table is `hw_irq_controller` structure - each interrupt vector has its own table
  - IRQs are #'d from 0-15 (since IDT is `0x20` to `0x2F`)
- Initially, all 8259A interrupts are masked
- startup called when first handler is installed for an IRQ, shutdown called after last handler is removed for an interrupt (they edit mask bit in 8259A implementation)
- ack called at beginning of interrupt to ack receipt (on 8259A, masks interrupt then sends EOI), end called at end of interrupt handling (on 8259 it unmasks IRQ on PIC)
- `request_irq(uint irq, void (handler)(int int_vect, void* dev_id, struct pt_regs*), ulong irqflags, const char* devname, void* dev_id)`
  - Used to install handlers, where `irq` is interrupt number and `irqflags` specifies options
  - `devname` is human-readable name (seen in `/proc/interrupts`), `dev_id` is pointer to device-specific data (returned to handler when called)
- Two important flags for `request_irq`
  - `IRQF_SHARED` allows multiple handlers to share a single int vector (chaining)
    - Requires all other handlers on action list to also have this flag
  - `IRQF_DISABLED` masks the `IF` while handler executes
- Interrupt handlers usually written in C, use standard C linkage/calling convention.

  - `pt_regs` is the struct on the kernel stack storing initial state of registers at start of interrupt

- Kernel keeps track of int vector-related information in `irq_desc` array of descriptors `irq_desc_t`
- `irq_desc_t` stores the following:
  - `status` - a bit vector tracking interrupt status
  - `chip` - pointer to jump table
  - `action_list` - linked list of `irqactions` for the current irq
  - `disable_depth` - count of calls to disable vector
  - `descriptor_lock` - spinlock to manage descriptor access
- When `request_irq` called, a new `irqaction` structure is allocated to represent handler.
  - `irqaction` stores `request_irq` arguments and pointer to next node in LL
  - `irqaction` is added to list using `setup_irq`
  - If there are not `irqaction` structs defined for the irq yet, it is directly assigned to the action list in `irq_desc_t` as the head. Make sure PIC table has proper default functions, clear some status flags, clear previous software disables, interrupt controller startup function is called.
  - If another `irqaction` exists, new handler and flags are checked for compatibility
    - ex: is `IRQF_SHARED` acceptable
- Handlers are uninstalled with `free_irq(uint irq, void* dev_id)`.
  - If a handler for the specified irq is found with same `dev_id`, it is removed.
  - If no handlers left after link removal, interrupt controller shutdown is called, software disablement is turned on so that any interrupts waiting for descriptor lock will abort.
  - Remove `/proc/irq/<irq #>/<action name>`
- Interrupts are invoked with `do_irq`, but since hardware will not "call interrupt" with valid convention, assembly **linkage** used to wrap C function call
  - when interrupt starts, x86 will switch to kernel stack if needed. It records user stack pointer, `EFLAGS`, return adddress
  - `common_interrupt` section of linkage will push regs to stack, then invoke `do_IRQ`
    - `EAX` will point to the part of the stack that regs are saved to

- `do_IRQ` acts as handler for all 8259A interrupts
  - Uses interrupt vector to index `irq_desc`, interacts with interrupt controller using `irq_desc_t` jump table, calls all handlers in action list
- `do_IRQ` will do different things based on status flags in `irq_desc_t`, 4 values are relevant for us
  - `IRQ_PENDING` (interrupt has occurred, waiting to be executed)
  - `IRQ_INPROGRESS` (handlers being executed)
  - `IRQ_DISABLED` (interrupt vector temporarily disabled, postpone execution)
  - `IRQ_REPLAY` (replaying previously postponed execution)
- When called, `do_IRQ` will immediately ack the interrupt. For our PIC, the interrupt is masked then EOI'd
  - If interrupt disabled then re-enabled by handler, PIC may raise interrupt again before `do_IRQ` ends
  - `IRQ_REPLAY` cleared after acking
  - When interrupt completes, in-progress flag is removed and PIC unmasked
- Handler is executed via `handler_IRQ_event` from within `handle_level_irq`
  - usually done with `IF=1`, so we avoid using descriptor lock to prevent deadlock
- **Tasklet**: Data structure (`struct tasklet_struct`) used to wrap a singler handler function used as a soft interrupt handler
  - soft interrupt for tasklet generated when some piece of code requests that the handler associated with tasklet be scheduled
  - when scheduled, tasklet is linked into list to be executed with any other tasklets of same priority - this list is maintained on a per-processor basis
- Four soft interrupt types
  - `HI_SOFTIRQ` - high priority tasklet
  - `TASKLET_SOFTIRQ` - low priority tasklet
  - `NET_TX_SOFTIRQ` and `NET_RQ_SOFTIRQ` - network transmission/reception
- **Gate Descriptor** - a single descriptor in the IDT. The gate type indicates what kind of code is associated with the handler

- Each IDT entry is 8 bytes (64-bit)
- When descriptor of type **interrupt gate** is invoked, processor disables interrupts
- Exceptions and System Calls are **trap gates**.
- Before enabling interrupts, kernel must initialize `idtr` register to point to IDT table

- Kernel maintains global array of function pointers called `interrupt[NR_IRQS]`
  - for PIC, `NR_IRQS` is 16
  - Stores pointers to interrupt stubs (interrupt handlers that all call `do_IRQ` for different IRQ numbers)
    * Kernel code/size vs speed tradeoff - making separate full handlers for each one when you have 256 IRQs is a lot of space for minor functional differences (why we use functions)/
  - At init, kernel will set all the interrupt gates to addresses found in `interrupt[NR_IRQS]` array

8.3 Interrupt Control and Status Functions

- `disable_irq(irq)` and `enable_irq(irq)` used to mask specific IRQs without masking all interrupts
  - waits for interrupt to stop executing on any processor - can cause deadlock if you call it from an interrupt handle. use `disable_irq_nosync` to avoid deadlock - it does not wait for executing interrupt handlers to finish.

9 Virtual Memory

This constitutes a large part of the exam content… know it all.

- Lecture Definition of Virtual Memory: indirection between memory addresses seen by software and those used by hardware
- Central concept: add a level of indirection between a program's memory address space and system's actual memory address system - we separate this with **virtual/logical addresses** vs **physical addresses**
  - At a hardware level, the CPU uses a **memory management unit** (MMU) to convert from virtual to physical address.
  - x86 uses virtual addresses at every privilege level, while other ISAs may not
- **page** - single chunk of virtual memory

  - Program may not map every page into physical memory - some pages may not be mapped at all, while other are mapped into device memory
  - In physical memory, page is generally mapped to a multiple of page size

- Four main advantages of virtual memory $***$ **IMPORTANT** $***$
  - *Protection* - prevents program from destroying other programs' data
  - *Sharing* - since programs don't modify code in libraries, a single memory location can be mapped into multiple programs' virtual address space for reuse. Code and data not being actively used by a program can be pushed out to disk to make space for active data
    - Provides illusion of a much larger physical memory
  - *Memory Fragmentation* - memory allocated to each program (usually) needs to be a contiguous region of memory. When a program frees a chunk of memory, it can leave a "hole" in the contiguous region. Virtual memory can allow a program to use regions of memory that are non-contiguous in physical memory, but appear contiguous in execution.
  - *Relocation* - absolute addresses used by each program need to be adjusted to avoid using memory not owned. If non-contiguous blocks of memory are used, then this becomes a tedious (and difficult) process. W/ virtual memory, using a single offset into the virtual memory address space is enough to do this relocation.
  - ***Tradeoff*** - More complexity during memory accesses, storage requirements to store the paging data, takes longer to perform each individual memory translation.
  - **Internal Fragmentation**: Too much memory is allocated and it goes unused (more likely with large page sizees).
  - **External Fragmentation**: When a program tries to allocate memory, it cannot find a large enough *contiguous* memory region (more likely with small page sizes).

9.1 x86 Protection Model

- protection organized into "rings", with innermost as level 0 and outermost as level 3
  - level 0 is kernel, level 3 is user, level 1/2 not used by Linux
- To make this work, code operating at a privilege level should never call into code operating at a (numerically) larger level. System calls should be used to request services from (numerically) lower levels, and those services should validate all input data.
- **Current Privilege Level (CPL)** is stored in register as part of processor state

- Accesses to memory are associated with a **request privilege level (RPL)**. Allows high-privilege code executing on behalf of less privileged code to make hardware enforce necessary privileges as if it were the less-privileged code making the access
- **descriptor privilege level (DPL)** associated with each memory location. On an access, the maximum of the CPL and DPL must be less than the DPL to have a successful access.
    - If this is not the case, the access is illegal and a **General Protection Fault** is triggered

9.2 Segmentation

- **segment** is a contiguous portion of an address space.
    - x86 processors always use segmentation in protected mode
    - segmentation is no longer used by most operating systems, still there for backwards compatability
- **segmentation** is used to convert each virtual address into a **linear** address
    - if paging is off,this linear address *is* the physical address
- Problems with **segmentation**
    - Address space needs to be continuous - lot sof potential for external fragmenttion
    - Low granularity of access control (hard to do protection)
- Segments are described in one of two arrays called *descriptor tables*
    - **Global Descriptor Table (GDT)** - used by any program
        - One GDT per CPU
        - pointer to GDT and 16-bit limit (size-1 in bytes) are stored in 48-bit `GDTR`
        - GDT can describe up to 8192 segments, where each segment descriptor is 8B
        - Each GDT entry is 8 bytes
        - Segment 0 is never used
    - **Local Descriptor Table (LDT)** - per-task segment table
        - Pointer to current LDT stored in 64-bit `LDTR`. Same 48-bit scheme as `GDTR`, but another 16-bit suffix describing index of LDT in GDT
- Each segment descriptor contains base address, limit on offset, DPL, and some other bits.

  - Given segment reference, processor checks that address does not exceed segment limit, then adds base address to calculate linear addr
  - GDT segment descriptors also contain code vs data descriptor (readable vs read/write)

- Six segment registers used to select referenced segment - code segment (CS), data segment (DS), extra segment (ES), stack segment (SS), FS/GS.
  - Subroutine calls uses code segment, `mov` uses data segment, string copying uses `DS/ES`, stack ops use `SS`. Any segment can be used explicitly by prefixing instruction as needed
  - Architecturally, 16 bits of segment register are visible - `[15:3]` are index, `[2]` is GDT/LDT, `[1:0]` used for RPL.
  - Remaining shadow bits used to cache values in GDT/LDT

- GDT entries can describe LDTs or **task state segments (TSS)**, which contain… task state for an individual program
- Linux essentially bypasses segmentation - linear address is same as virtual address
  - segments 0/1 unused, segments `2/3/4/5` start at address 0 and encompass full 4GB address space - only differ in DPL/type (code/data).
  - two GDT entries p/processor used to store LDT descriptor and TSS
    * Linux rewrites TSS (processor-independent) for a CPU before running a new program
    * only one LDT exists, all LDT entries point to same LDT in physical memory
  - In linux, segment selectors accessed with macros
    * `KERNEL_CS`, `USER_CS`, `KERNEL_DS`, `USER_DS` (indices into GDT from GDTR)

9.2.1 Format of an Entry in GDT

63:56	55:52	51:48	47:40	39:16	15:0
Base[31:24]	Flags	Limit[19:16]	Access Byte	Base[23:0]	Limit

- Base - 32-bit value containing linear address where segment begins
- G (granularity flag) - is segment sized in bytes or multiples of 4kB
- DPL - descriptor privilege level

9.3 Paging

- Second level of indirection, linear address passed into paging unit
- Any given page can be present, swapped out, or nonexistent
  - Mapping for some pages can be left undefined altogether, since they are never actually used
  - If a page exists, but it is not in the physical memory we call it **swapped out**
    - If data was moved out of memory to free up room, the page's data is kept on a **swapdisk**.
    - Swapdisk is inaccessible to normal users
  - If program tries to access nonexistent or swapped out, processor throws an exception
    - Gives OS time to either sleep program and move data back from swapdisk to memory, create new page, or send signal to program (Segfault is default result of this)
- **Page Table**: stores mapping from virtual to physical memory (or disk)
- We use hierarchical paging structures so that we don't need to allocate memory to map the entire virtual address space at once - especially across many processes, trying to allocate memory to page the full 4GB space right off the bat will burn our memory up in no time.
- $\hat{}$ is an array of **page table entries**, where each is 4B (32-bit)
  - Bit 0 used to indicate present vs. not-present in physical memory. (In x86) if `[0]` is 0, rest of entry ignored
- Page table is also paged to save space when memory is unmapped - this is organized in **page directory**
  - Page table/directory each has 1024 entries (10-bit addressability)
- **Page Directory Base Register (PDBR)** stores physical address of page directory
  - Referred to as `CR3` in assembly
- x86 Paging setup
  - High 10 bits of virtual addres used to index Page Directory
  - If 4MB entry, bottom 22 bits of virtual used as index into 4MB page, top 10 of PDBR used to find 4MB page in physical memory

  - If PDE points to another PT, then the next 10 bits of virtual address index index into the page table. Top 20 bits of PTE are used as part of the physical address and the bottom 12 bits of the virtual address are used as offset into that page.

- **Translation Lookaside Buffer (TLB)** used to cache translations to minimize the number of page-walks needed.
  - If virtual address not found in the TLBs, it is called a **TLB miss**
  - Translations are flushed when PDBR (`cr3`) is changed
- **G flag** - TLB is not flushed when CR3 changes, used for kernel pages (in Linux)
- Paging enabled with bit 31 in `CR0`, 4MB pages enabled with PSE bit (bit 4) in `CR4`
- **Physical Address Extension(PAE)** - used to allow 32-bit system to address more than 4GB of memory
- Separate **TLB** for 4MB pages and 4kB
- User/Supervisor mismatch (Supervisor mode but CPL/RPL is 3) causes General Protection Fault

9.3.1 Page Table/Directory Entries

Virtual Addresses

31:22	21:12	11:0
directory #	page #	offset

or

31:22	21:0
directory #	offset (into 4MB block)

PDE Entry (4MB)

31:22	21	20:13
ADDRESS[31:22]	RSVD(0)	ADDRESS[39:32]

12	11:9	8	7	6	5	4	3	2	1	0
PAT	AVL	G	PS(1)	D	A	PCD	PWT	U/S	RW	P

- PAT - page atribute table
- AVL - available (for kernel use)
- G - global (do not flush TLB when CR3 reloaded)
- A - accessed
- PCD - disable cache

- PWT - write-through
- U/S - user vs supervisor (Supervisor requires `PL<3`, User is for everyone)
- RW - read/write
- P - present
- D - dirty

PDE Entry (Table)

31:12	11:8	7	6
PT Addr	AVL	PS(0)	AVL

Bottom 6 bits are same as the 4MB entry.

A PTE has the same upper 20 bits as a PDE Table entry (but for a 4kB block of memory), and uses the same bottom 12 bits as the 4MB PDE entry. The upper 20 bits are prepended to the VA offset to form a physical address.

9.4 Filesystem

- VFS (virtual filesystem) provides common interfaces - create, open, read, write, etc.
  - user-space write() maps to VFS $\text{sys}_{\text{write}}()$ which maps to *filesystem* write method.
  - Application communicates with VFS, which communicates with underlying filesystem (ext4, FAT32, BTRFS, etc.)
- Unix-like filesystem is an information container structured as sequence of bytes
  - files organized into a tree. internal nodes denote directories, leaves denote files
- Common filetypes:
  - Files, Directory, SymLink, Block-oriented evice file, Char-oriented device file, [named] pipes, socket
- Clear distinction between file contents and file information
  - Inode contains information needed by filesystem to handle file (access perms, size, owner, etc.)
  - Each file has an inode
- File descriptor indexes a kernel-level data structure containing details of all open files
  - created by process when file opened

- Filesystem Objects
  - Superblock - specific mounted filesystem
  - Inode - specific file
  - Dentry - represents directory entry, single component of a path
  - File object - represents open file as associated with a process (only in kernel memory)
- File operations structure is a jump table of file ops/character driver ops
  - There is a generic instance for files on disk, and distinct instances for sockets, etc.
  - General rule - *one instance per device type*
- ext2 implementation (not explicitly tested, but good to know)
  - ext2 has a boot block and then a bunch of block groups
  - each block group has:
    - superblock copy
    - group descriptors copy
    - data block bitmap - represents free blocks for data
    - inode bitmap - represents free blocks for index nodes
    - inode table
    - data blocks
- Superblock image (same as boot block)
  - Contains filesystem check information (# of mounts between checks, time between checks, error count, etc.)
  - contains filesystem state (mounted/uncleanly dismounted, cleanly dismounted, errors found)
  - Reserved blocks/auth data, Volume name, performance spec
  - Group Descriptor Image
    - contains shortcuts to block/inode bitmaps and inode table/data blocks
- Index Node (inode) contains a field called `i_block` - this contains data block indices for the file.
  - There are 15 entries in `i_block` - 13th block number points to another array of block indices
  - 14th block points to an array of 13th entry configurations. 15th block points to an array of 14th entry configurations.

9.5 System Call Linkage

- System Calls identified by number, registers used to pass operands rather than pushing to stack
  - Invoked with `int 0x80`
  - `EAX` used to select specific system call - if valid, it is used to index the `syscall_table` jumptable (`entry.S`)
  - Return value is still put into `EAX`
  - Errors are indicated when the return value is [-1, -4095]
- System Calls in Linux are written as C functions
- System Call vector table requires linkage from/to C functions
- IDT entry `0x80` contains the `system_call` function
- `system_call` function starts by saving all registers to stack - if program is under a debugger, debugger can intercept system call.
  - SystemCall saves its registers *above* the data that IRET uses for restoration
    - Order of register access from top of stack - `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `EBP`
  - `call` made to the appropriate routine in system call jumptable
- C library error code stored in `errno`, on error C library returns -1
- Since **errno** is part of a relocatable library, we use a "fake call" to get the EIP, then use an offset from that instruction to find `errno` location. This is done in the `_errno_location` function
- User linkage takes function arguments and moves to registers (`ebx`, `ecx` , `edx`) and moves syscall number into `eax`, runs `int 0x80`, registers are pushed to stack for the jumptable function.
  - `EAX` interrupt number is boundchecked when `int 0x80` is run

10 Processes and Tasks

- OS creates virtual environment for a program to abstract protect and share resources
  - Virtual memory (segmentation, paging)
  - Virtual devices (filesystem, ioctl)
  - Virtual CPU (scheduling, processes)

- Execution environment of a program is called process
- Multiple prorams are run on a CPU by **time slicing**
  - One program run for a bit, then it is suspended and another one starts running
- **Process** - virtual execution environment that includes virtual memory, I/O, FS state, etc.
- **Task** - unit of scheduling
  - encompasses thread of execution, continuation, kernel stack
  - A single process can have multiple tasks by multithreading
- A task belongs to one or zero processes - kernel task belongs to no process
- **Continuation** - captures executing state of process.
  - Stored in each process's own kernel stack.
  - Contains flags, stack position, registers, return address, linkage return address, etc.
  - Store user stack information (SS and ESP). Kernel stack informatin will be in TSS
  - During context switch, current process continuation saved, next process continuation restored
  - A context switch *always* switches tasks, but multiple tasks can be related to a single process
  - If a context switch is also between processes, VirtMem and kernel state must be updated
- **Task State Segment (TSS)**
  - Contains I/O Map Base Address, LDT segment select, segment registers, register state (`pushal` configuration), `EFLAGS`, PDBR, stack setup information, Previous Task Link, etc.
  - When switching privilege levels between tasks, **Task State Segment** used to switch stacks. Also used to switch between paging configurations
    - (SS2/ESP2), (SS1/ESP1), (SS0/ESP0) are saved values for privilege level (stack setup information)
  - I/O Map Base address points to start of I/O permission bitmap (IO Map)
    - I/O bitmap length determined by TSS descriptor (in GDT) segment limit
    - `IOPL` in `EFLAGS` defines privilege level needed for arbitrary IO access
    - if IOPL not met by `CPL`, then processor checks I/O bitmap in TSS

  - TSS is setup by initializing TSS data struct in memory. TSS descriptor entry is set up in GDT, and `LTR` instruction loads task register (`TR`) with segment selector
    * SS0 and ESP0 point to kernel stack

- Hardware context switch - registers saved in current TSS, reloaded from new TSS, and `TR` is updated via `LTR`.
  - hardware CS is limited - each task needs a GDT entry, and there are performance issues with a far jump

- TSS is only really used for the kernel stack pointers - we do all the other work ourselves.
  - one TSS used per CPU
  - On context switch, `esp0` and `ss0` are updated in TSS (context switch is always task switch)
  - Manually save/restore registers, EIP, CR3, LDT, ESP from kernel data structures

11 MP Review (MT2)

- `IRET` returns from interrupt, pops off 5 arguments
  - (from top of stack) return address, code segment (`CS`), `EFLAGS`, `ESP`, stack segment (`SS`)
    * These are pushed by the `int` command
  - Unlike `RET`, `IRET` is capable of switching from kernel space to user space

11.1 Filesystem (MP3)

- Boot Block contents - 4kB (1 block)
  - number of directories
  - number of inodes
  - number of data blocks
  - reserved 52B
  - 63-long list of dentry objects

- Dentry (part of boot block)
  - contains filename, filetype, $\text{inode}_{\text{num}}$

- inode (index node)

  - consumes 1 block
  - contains length, and `int32_t data_block_num[1023]` to store block no.'s
    * can be optimized - use indirection like ext2
    * indirection points to a block that is FULLY block numbers, no length in the first 4B
- Overall FS structure
  - Each block is 4kB
  - Boot Block at index 0, followed by inodes, followed by data blocks
  - File array tracks open files - each index in file array has file operations table pionter, inode, file position, and flags
    * File array stored in PCB (process control block)
    * indexed using file descriptor (`fd`)
- Each task has up to 8 open files
- Process Control Block (PCB) analogous to task struct in linux
  - stored above kernel stack for corresponding process
- Inode block numbers are not indirected in the filesystem
  - This may be something you are requested to add
- Reserved bits are used for padding
- **Advantage of Storing File Name in Dentry (separate from inode)**: Isolates file data from file hierarchy, makes it easier to do things like reorganize file structure and rename files without editing inode (only one write instead of two writes, no need to maintain parity between dentry and inode). Furthermore, you can have multiple directory entries associated with different filenames pointing to same filedata.
- Maximum filename in dentry is 32 characters, smaller filenames require a nullchar at the end of the line

11.2 Tux Driver Questions

- Interrupt based approach
  - We turned on `BIOC` interrupts, and waited for `BIOC_EVENT` packets in our packet handler. This way we don't need to poll!
- Synchronization
  - A mutex was used to lock the state of player movement
- How to handle spammming inputs
  - We used an ack signal

11.3 ModeX

- Mode X pixels are separated into 4 planes - each memory location stores 4 pixels, each taking up a byte in memory (one memory location is 32-bit)
  - Each byte contains 8-bit index into VGA palette
- VGA Palette contains palette with 18-bit color (6 bits per RGB)
- Write Mask is a register - anything written to video memory will be written to bytes in each 32-bit entry corresponding to write mask.
- VGA Mode X is double buffered for game frames, but statusbar isn't (statusbar always starts rendering at address 0)
- Build buffer is organized such that logical plane 3 is at the beginning
  - As you move in a direction, the build buffer up and down within a memory fence to account for it. This causes a "bubble" (empty index) to form between planes within the buffer.

12 Scheduling

Some of this is a review from Lecture 17 (programs to processes), I thought it'd be good context

- Operating System creates virtual environment to abstract and distribute resources - virtul memory, virtual devices (filesystem, ioctl), and virtual CPU (scheduling)
  - A program's environment is called **process** (virtual execution environment)
- Scheduling provides illusion of multiple processes running simultaneously
- Linux 2.6 adds pre-emption of kernel code to support real-time tasks
- *Time Slicing* - run one program for a bit, suspend it and start running a second one. In this paradigm, **time** is the resource being distributed
- **Continuation** - saved execution state (registers, flags, segments, etc.). Processor saves one and then performs a *context switch* (restore continuation of next process)
  - Continuation, like with an interrupt, is saved on kernel stack. Each process has a separate kernel stack
- **task** is a single unit of scheduling
  - Each *task* will have its own kernel stack and execution state, but a single *process* can have multiple *tasks* (multithreading)
  - *task* belongs to either one process or zero processes (if it's a kernel task)

- Context switch - *always* a task switch, *usually* a process switch (in which case virtual memory and kernel state is updated)
- x86 wants you to store continuation in the *TSS* (task state segment)
  - Stores segment values (`GS`, `FS`, etc.), LDT segment selector, IO Map Base address, register values, flags, `EIP`, `CR3` (PDBR), kernel ESP (`EP0`).
    - One `SS` / `ESP` set per privilege level.
  - Ex: during an interrupt, the CPU will pull the kernel stack ESP from `tss.esp0`
  - Second two parts of TSS: interrupt redirection (emulating earlier ISAs) and I/O permission bitmap (for individual ports). Address given by I/O map base address.
    - *I/O Privilege Level* (IOPL) in `EFLAGS`, used to denote maximum privilege level awarded arbitrary I/O access
    - Used when CPL>IOPL - verifies that request is with necessary permissions by crosschecking with IO bitmap in TSS. Exception if either not enough bits in bitmap to denote specified port or bit representing port is set to 0
- *TSS setup*
  - Create TSS in memory. Minimum parameters: `SS0` / `ESP0` init
  - TSS descriptor made in GDT, `LTR` used to load task register `TR` with TSS GDT entry
- Hardware context switch (far jump) - registers saved in current TSS, reloaded from new TSS, and `TR` is updated via `LTR`.
  - hardware CS is limited - each task needs a GDT entry, and there are performance issues with a far jump
- TSS is only really used for the kernel stack pointers - we do all the other work ourselves.
  - one TSS used per CPU
  - On context switch, `esp0` and `ss0` are updated in TSS (context switch is always task switch)
    - For MP3, `SS0` is always kernel DS. But I mention having to update it as a theoretical thing.
  - Manually save/restore `registers`, `EIP`, `CR3`, `LDT`, `ESP` from kernel data structures
  - MP3 - we use `IRET` to set up a dummy stack and return to user mode in a new context
  - Prevents "cloning of resources" - all the functionality and information is all in software. Also some benefits of performance.

- Task Structure (or PCB in MP3)
  - Maintains fields used to interact with and describe the current process
  - MP3: parent process pid, `fd` array, active or not (flags), `esp/ebp` backups to return to parent
  - Linux: pid, state/prio (for scheduling), prev/next (linked list), `*mm` (memory map) `*files *fs *signals` (process components), `esp/esp0` backups
  - Task State
    - `TASK_RUNNING` - on the run queue, currently executing or waiting to execute
    - `TASK_INTERRUPTIBLE` - on the wait queue, sleeping for a semaphore/condition/signal. can be woken by a signal
    - `TASK_UNINTERRUPTABLE` - on the wait queue, sleeping for a semaphore/condition/signal. Can only be woken by NMI or completion of the intended task.
      - Implication: task is busy with something that cannot be stopped. Device will go into unrecoverable state w/o further task interaction.

The following is all linux-specific implementation. It's probably not very important for the midterm.

- Task structures arranged in doubly linked list - head is the `init_task` sentinel
- To find current node, a hash table is used to map from PID to an array of struct pid* pointers - these structs implement a doubly linked list for each hashtable bin "hit" and reference task structures
- User creates processes/tasks with `fork`, `vfork`, `clone` system calls
  - In the kernel, these all map into different calls of `do_fork` - returns new pid or negative on error
  - `do_fork` params: `clone_flags` for sibling instead of child task, `stack_start` in userspace, `regs` for new task, `stack_size`, `parent` and `child` tidptr
  - `copy_process` called to set up process descriptor/kernel DS
    - x86 linux only uses `stack_start` and `regs`
  - Program usually started by a shell or other interface program
    - `fork` to create new process, `exec` to load a new program if necessary - this is the common case

- `fork` duplicates process address space (deep copy of all corresponding pages), which was slow. `vfork` blocks the parent while child uses address space, and returns address space control to parent when child exits
  - `fork` uses copy-on-write

- **Copy-On-Write** - data is not duplicated, the page tables are with *no write permission*. If either process attempts a write to a shared page, a private copy is made (no data corruption for second process)
  - Example of a lazy approach
- **Kernel Thread** - task without associated address space. Address space inherited from last user task.
  - `init_task` is a kernel thread - persists until shutdown.
- **Scheduling Design Goals**: efficient, fair, responsive
- Scheduling Job Types
  - *Interactive* - driven by human interaction, not much processing after each event. Want to minimize response time, but time slice does not need to be very long afterwards.
  - *Batch* - only completion time matters, want a fair chunk of CPU's time/throughput
  - in essence: IO-bound vs Compute-bound $*$ **IMPORTANT BUZZWORDS** $*$
- Real-Time: there is a deadline for by when the compute needs to be done before it becomes useless or something has gone wrong
- **Priority** - used to determine which processes are more urgent/time-sensitive or important, and thus should be scheduled earlier
  - Compute-bound applications generally have lowest priority
  - LeBron, Linus Torvalds, and Mr. Bean walk into a hospital. LeBron thinks he has bronchitis, Linus is shot, and Mr. Bean has a rash on his neck. What order do you tend to them in?
- **Turnaround Time** - A process is given to the scheduler at time T and finishes at time E. The *turnaround time*, $E - T$, is the amount of time it took to complete.

12.1 Linux Scheduling Strategy and Implementation

Not explicitly on exam, good to understand a general scheduling algorithm

- Time is broken into *epochs*. Each task is given a 'quantum' of time, tasks are run until no task has time left. New epoch starts after.
- Real-time jobs given priority over non-RT. Prioritized against other RT jobs w/ priority levels.
  - Static vs dynamic priority used for non-real-time jobs
- Interactive jobs handled w/ heuristics - estimate job interactiveness
  - Can preempt the current job based on interrupts
  - Can continue running after "out of time" - we assume that interactive jobs don't actually use quantum
- Each processor has a run queue - each queue has two priority arrays (lists of tasks of each priority).
  - Double buffered to implement epochs (separated into active/runnable proceses vs expired (out-of-time) processes)
  - 100 RT priorities (0-99), 40 regular priorities (100-139). One list per priority (in each priority array), each index is a doubly linked list of tasks at that priority.
    - Bitmap used to speedup non-empty list lookup
- Each processor will also have a wait queue, used to track processes that are asleep

12.2 Scheduler Policies

The following policies are discussed in the context of real-time tasks. While scheduling is still occurring, these policies determine how real-time processes are prioritized and run (in linux). Know what these are and how they work at a high level - *they are algorithms for dividing compute time between multiple same-priority processes.*

All of the below algorithms have their own pre-emption schemes. Note that this refers to ordering multiple same-priority tasks - if a higher priority task arrives, *all of the algorithms below will pre-empt the current one.* Priority is always mentority.

- **FIFO Scheduler Policy** - also called *FCFS*
  - When scheduler assigns CPU to an RT process, it continues running that RT process until completion *unless* a higher-priority RT process is runnable (process is put at front of runqueue)
  - Priority levels are maintained, and this RT process cannot be pre-empted

  - Simple to implement/maintain, but can cause long wait times and increase average waiting time if the first process is very long
    * If the current process is especially long, it can cause starvation

- **Round Robin Scheduler Policy** - we implement this in MP3, or at least a BangGood version of it
  - When CPU assigned to process, it is moved to back of runqueue
  - Essentially, each process is assigned a time slice, run for its duration. At end of timeslice, rescheduled
  - When a new process is launched at the end of a timeslice, it is added to the waitqueue before the currently running process
  - Every process gets an even share of the CPU and no starvation because RR is cyclic, but setting a short quantum increases process switching overhead while a long quantum causes poor response time to short processes. Under RR, average waiting time is long and a high-quantum RR can degrade to FIFO

- **Shortest Job First** - pulled from review session, not formally discussed in class
  - Do whichever of the jobs takes the least amount of time - if a shorter job shows up, you can preempt and do that one
    * SJF with pre-emption is the same as *Shortest Remaining Time* algorithm
    * Essentially me with my work. Would you rather finish a 311 Lab or start MP3?
  - Jobs are finished to completion - same execution policy as FIFO, different prioritization scheme
  - Allows shorter jobs to be completed quickly (more responsive) and minimizes average waiting time, but can cause starvation if shorter and shorter processes keep coming

12.3 Rescheduling

- Task can change (context switch) if current task yields or current task runs out of time
  - Task may implicitly yield by attempting to acquire a semaphore or when waking up a process

- Every tick, reduce current task's time (ticks indicated with PIT interrupt on IRQ 0)

12.4 Walking through HKN CC question

Consider the following processes - how would they be scheduled per the round robin algorithm? Assume that new tasks are added to the queue before the current task.

pid	time start	len
A	0	5
B	3	2
C	5	1
D	7	6
E	8	4

12.4.1 Solution

Queue State - queue and new appended act as the "queue" for the next cycle

curr	queue	new (1 timeslot early for visualization)
A		
A		
A		B
B	A	
A	B	C
B	CA	
C	A	
A		D
D		E
E	D	
D	E	
E	D	
...		

13 System Call Linkage

Blatantly reused from my Exam 2 Notes. Should be sufficient though.

- System Calls identified by number, registers used to pass operands rather than pushing to stack
  - Invoked with `int 0x80`
  - `EAX` used to select specific system call - if valid, it is used to index the `syscall_table` jumptable (`entry.S`)
  - Return value is still put into `EAX`
  - Errors are indicated when the return value is [-1, -4095]
- System Calls in Linux are written as C functions

- System Call vector table requires linkage from/to C functions
- IDT entry `0x80` contains the `system_call` function
- `system_call` function starts by saving all registers to stack - if program is under a debugger, debugger can intercept system call.
  - SystemCall saves its registers *above* the data that IRET uses for restoration
    - Order of register access from top of stack - `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `EBP`
  - `call` made to the appropriate routine in system call jumptable
- C library error code stored in `errno`, on error C library returns -1
- Since **errno** is part of a relocatable library, we use a "fake call" to get the EIP, then use an offset from that instruction to find `errno` location. This is done in the `_errno_location` function
- User linkage takes function arguments and moves to registers (`ebx`, `ecx` , `edx`) and moves syscall number into `eax`, runs `int 0x80`, registers are pushed to stack for the jumptable function.
  - `EAX` interrupt number is boundchecked when `int 0x80` is run

14 Memory Allocation

- Some portion of physical memory assigned to kernel for code/data, rest dynamically assigned at runtime
  - Kernel must track whether each page in physical memory is used

---

Sidebar for more linux stuff.

- *x86* Memory Allocation Constraints
  - Direct Memory Access (DMA) - older hardware can only address first 16MB
  - 32-bit computers might not be able to address all of the physical memory available
    - PAE (physical address extension) bit used
- Linux Memory Zones
  - `ZONE_DMA`: Pages below 16MB
  - `ZONE_NORMAL`: Between 16MB and 896MB

- `ZONE_HIGHMEM`: Above 896MB

- `kmalloc` uses exponentially sized slab caches (ranging from 8B to 4MB in our kernel)
  - used to allocate a few small items
  - Manages a private [slab] cache of objects - frequent allocations/deallocations
  - leaning on slab cache helps minimize internal fragmentation and dynamically allocate small buffers
- **Slab Cache** - `kmem_cache_alloc` and `kmem_cache_free` to allocate and free items. Underlying interface of kmalloc.
  - Contiguous physical memory
  - Used to allocate a lot of items (mainly structs) repeatedly - one slab cache created per item type
  - Breaks up a page into tightly packed versions of a single struct - kmalloc returns a pointer within this page
  - Will grow/shrink as needed automatically - passes down to `getfreepages` if a new slab is allocated
  - `kmem_cache_create` and `kmem_cache_create` to create and destroy items
- Get big chunks of memory by consuming free pages
  - `get_free_page` and `free_page` used to obtain/free pages. *Important for any memory allocator to have these implemented* so that they can grow and shrink as necessary.
- Obtain large chunk of memory in contiguous *virtual* address space with `vmalloc`. No guarantee on physical contiguity - good for if we need larger pieces of memory, or don't care about physical contiguity.
- Benefits of contiguous physical memory over a bunch of smaller pages
  - Less external fragmentation
  - Less page table traversion needed (one large page vs a bunch of smaller ones)
  - More likely to get TLB hits because of fewer virtual address ranges
- **Fragmentation**
  - *External* - Not enough contiguous physical memory to allocate large block of contiguous memory

    * Lots of small 'gaps' in memory with enough free space for a large block, but no gap large enough to shove in a *contiguous* large block
    * Allocating contiguous physical pages decreases external fragmentation, can potentially increase internal.
  - *Internal* - mismatch between size of memory request and size of memory area allocated to satisfy request
    * Memory going unused after allocation, should have set aside for another process
    * On-Demand paging improves internal fragmentation rate while potentially increasing external fragmentation

14.1 Buddy Allocator

- Problems to Solve: We want page alignment for allocations, may need contiguous regions of physical memory, need flexible allocatoin granularity, etc.
- Separate a large chunk of memory into power-2 bins (1 page, 2 page, 4 page, 8 page, etc.)
  - Allocate bins, separate bins into smaller bins, recombine into larger bins as necessary
- Each "tier" has a bitmap denoting its pairs as *partially* busy. Note that this refers to the *pair*... Thus if *1* buddy is in use, we mark the pair as busy.
  - Bitmap used to "recombine" entries during deallocation (if bit is inverted to 0, both are free)
  - There is a separate free list at each tier, where each entry corresponds to a single block. Use this to find free blocks
  - Ensure that lower index of the pair is divisible by the new 2^n order. Otherwise, pair is not aligned and cannot merge
- If a block is not currently associated with a certain order n block size, it will not be marked as free in that blocksize's free list even it's free. Prevents false hits at smaller blocksizes when the block is still associated with a larger size.
- **Drawbacks**
  - Requires blocks to be aligned with $m2^n$ for the mth order n block. Even if the space is available, the buddy allocator might not be able to allocate it per its allocations scheme.
  - The 2^n allocation size could cause internal fragmentation at larger block sizes - if you want enough space for a 24B structure, suck it up you're still getting 4kB!
- TL;DR: prevents external fragmentation but not internal fragmentation

  - Linux solves the internal fragmentation issue by layering a slab allocator/cache on top of the buddy allocator to distribute small pieces of memory from the buddy allocator's pages

- Interface: `__get_free_page` and `__get_free_pages`

  The high level overview - we layer `kmalloc` on a slab cache on an **OK BUDDY** allocator.

14.2 Memory Maps

- Virtual address space has different memory regions
  - Code, Data, Heap, Stack, Shared libraries
  - Tracked via a collection of memory maps - each process has one.
    * Linux arranges these as a doubly linked list
- *Memory Map Structure*
  - Keeps track of the different regions of memory
- Modern OS, like linux, can dynamically grow memory regions like the stack by allocating more memory… how?
  - If value pushed that would cause stack overflow, the page fault handler is triggered. If this happens, and we determine it's because the memory region is too small, OS will increase stack size
- **On-Demand Paging** - during dynamic allocation, update kernel bookkeeping for how big heap *should* be, but set pages to not present. Don't actually map into physical memory until the program tries to access that memory.
  - Prevents internal fragmentation (operating system won't provide the memory until the program does something with the provided virtual memory *address*). Lazy approach to allocation!
  - Pagefault handler used as 'trigger' for allocation - if the address causing a pagefault is in the memory map, then allocate. If not in a region in the kernel's memory map struct, send back a segmentation violation
  - Other uses: defer reading code from disk, zeroing out physical memory, or even copy-on-write (that copying is done in pagefault handler).
- C library is shared (even if write access), everything else is not (r/w executable files, heap, stack, whatever the heck else). This applies to even multiple instances of the same program
- Number of page tables for a single process isn't just based on the *number* of pages, but also the contiguity of the virtual address space of the memory map.
  - Ignore the bottom 12 bits (3 hexadecimals). There should only be 10 bits of variance above that for the most part (each page table has 10 bit indexing).

15 Signals

- Operating system serves to abstract away hardware asynchronicity from software - signals add user-level interrupts, meaning that software can implement its own asynchronous behaviors.
- Signal notifies process of events, process will execute an appropriate handler
- Common Signals

Name	When	What
SIGINT	Ctrl-C	Terminate
SIGSEGV	Illegal memory access	Dump core
SIGKILL	Explicit signal (kill -9)	Terminate w/ prejudice
SIGALARM	Timer signal from `alarm()`	terminate
SIGABRT	default exception signal (`abort()`)	dump core
SIGSTOP	want to stop process (ctrl+z)	stop (always)

- Signals vs Interrupts
  - *Similarities*
    - User program sees them as asynchronous (but from a hardware perspective, signals are synchronous)
    - can be ignored or blocked (masked)
    - Some signals are NMI (`SIGKILL`, `SIGSTOP`) - cannot be ignored, blocked, or caught
      - handler cannot be defined for these, OS will handle them (default handler always used). `SIGKILL` always immediately terminates a process, `SIGSTOP` will always stop a process.
    - User defines handler for each signal, user defines handler for each interrupt
    - Traditionally, signals not queued - 2 signals will result in one handler call, and signal blocked during handler
    - Only information sent regarding signal is its number
  - *Differences*
    - Signals software-generated (either kernel or program w/ syscall), interrupts are hardware-driven
      - Ignore the soft interrupt edge-case - those are technically serviced by `ksoftirqd`, which is a kernel daemon operating at a slightly higher priority than the user
    - Signals do not have a "device" - only software with permissions can send signal
    - POSIX queues real-time signals, can send signals to both threads *or* processes, and the `siginfo` struct contains additional information about signal
- **blocked** - signal is *temporarily* masked, handled later
- **caught** - used to execute a *program-defined* handler

- `kill` system call from user program sends signal to a PID - verifies that signal and PID are valid, and that current process has permissions to send the signal
  - Anyone can *attempt* a `kill` call but it can potentially fail if privilege is not met or garbage is passed
  - Permission checks: sysadmin/kernel mode/privileged call can always send signal, process with same user ID can always send signal, process with same login session can send `SIGCONT` (make the process continue)
  - If checks pass and signal is not ignored, signal is added to pending signals - sleeping recipient is woken up and kicked out of wait queues (unless it is a `TASK_UNINTERRUPTIBLE`, those are only woken by NMIs)
  - Kernel will use `send_sig_info` mainly, supplies signal number, information about *why* signal is sent, and the target task.
    - `force` variant ignores signal mask/signal ignorance, resets signal handler, and sends it
- `force` variants are nice for exceptions - avoid otherwise since it can introduce bugs into the program
- All exceptions trigger a signal - it depends on which exception though
  - Ex: `SIGSEGV` is specialized, but a div-0 exception will cause a `SIGABRT`
- Check `sigpending` in task structure when returning from any interrupt, exception, or system call - these signals are then **delivered** and corresponding handlers are executed
  - Implicit understanding - signals can only be delivered for the currently running process
- Kernel will store signal behavior in a process's `sigaction` structs
  - Stores pointer to handler, whether to mask
  - Contains other flags:
    - Whether to send more information than just signal number
    - Reset signal handler to default after executing current handler
    - Don't mask signal while handler executing (second signal of same type can interrupt the first's handler execution)
    - Don't send signal if a child program stops or continues (only for `SIGCHILD`)
- `sigaction` struct initialized with `sigaction` function - very funny
  - Supply pointer to current sigaction struct and the new one that you want to populate with

- `sigprocmask` lets program read/change its signal masks - any masked interrupts are 'held' by the OS until the mask is removed, and the signal is immediately sent
  - These signals are called **pending** while they wait for delivery - differ from **ignored** signals since those never make handler calls
  - Currently pending (masked) signals can be checked with `sigpending`
- `sigsuspend` - suspend program until a certain (of potentially multiple) signal[s] is received. Program is slept until then.
- Default handlers are executed by the kernel directly - program-defined handlers are executed by adding a new execution context to the top of user stack and then running the signal handler in user mode.
  - Machine state context is added to user stack (view table below to see user stack state)

(`ESP`) $\rightarrow$ Return Address
Signal Number
Siginfo Pointer ($\&\beta$)
HW Context Ptr ($\&\alpha$)
β) Siginfo Struct
α) HW context
exec sigreturn() syscall
Old stack

    * If no `siginfo` struct passed into `sigaction` function, then `siginfo` pointer and struct will not be pushed to user stack
    * Return address points to the 'exec sigreturn() syscall' snippet of code, which is on the user stack. `sigreturn()` goes back into kernel space after the "custom" signal handler, restores hardware context from the user mode signal handler context, then goes back into user mode to resume process execution
      · Stack snippet just invokes `sigreturn` syscall (`int 0x80`)
    * Hardware context includes: registers, flags, interrupt context. Whatever context we would normally use to restore our user mode execution state needs to be synced with the user context.
      · We offload execution context to hardware stack to prevent inundating kernel stack (from either recursion or malicious program) and to let user handler modify machine context (since it can access the continuation).
  - `sigreturn` will copy hardware context back into kernel stack, teardown user stack
    * check if the signal was delivered during a system call (syscall might have been on a wait queue, woken by signal), and whether system call needs to be restarted as a result

    - If system call shouldn't be restarted, return `-EINTR` to indicate interruption by signal. Else, restore `EAX` used for signal call (can be found in hardware context) and move program PC back to the `INT 0x80` instruction (decrement by two bytes).
    - if `sigreturn` isn't called in the signal handler (code never returns), then user stack could be borked/overflowed but kernel is completely fine (all of the state was taken off kernel stack)

- *Fun Tidbit* - technically, only *real-time* signals are queued by the POSIX standard. But this class is weird, so just retain the normal understanding of signals being queued into `sigpending` as a bunch of `sigaction` structs `:)`

16 Driver Design and the I-Mail Case Study

16.1 Background

- **Kernel's Role in System**
    - Process management (scheduling)
    - Memory management (virtual memory, memory allocation, etc.)
    - Filesystem (everything in linux is a file)
    - Device Control (system interactions with hadware)
    - Networking (not covered because the internet doesn't exist)
- Kernel uses device drivers to interact with IO - allows encapsulation of device code, a well-defined programming interface, abstraction of device details, and dynamic load/unload of drivers
    - For the last point, think about what we did with the Tux driver
- I/O devices treated as *device files* - same file syscalls used (`write`, `read`, whatever else)
    - Device can either be block or character
- Block device - data only accessible as fixed-size blocks, device determines blocksize
    - Allows random access addressing, but transfers to/from device usually buffered (to) and cached (from)
- Character device - device is a contiguous space of bytes, some allow random access while others require sequential reads (sound card)
    - Device file usually stored as a real file, where `inode` stores identifier of hardware device corresponding to device file

- Devices identified by major/minor numbers (major is device type, minor is instance number)
  - Each major number has a `fops` structure associated with it
  - Device driver registers with `insmod`, removed with `rmmod`
  - Driver ops use three kernel data structures
- File operations jumptable - one instance per device type, file structure will have a pointer to relevant jumptable
- File structure/object (in fd arrray) represents open file. created by open system call
  - `inode` - used by kernel to reperesent file metadata, stored in permanent storage
- System calls to device file will use fops jumptable to issue driver function rather than underlying filesystem function
- Kernel must define `module_init` and `module_exit` for device driver to support `insmod` and `rmmod` commands
- **Blocking** - waits until information is received to return
  - `Read` and `Poll` are blocking, `write` is non-blocking (this is I-Mail specific)
- Direct device access reduces overhead for individual interactions, but driver interfaces that run in kernel space allow for more standardized interfaces that abstract away inconveniences with device communication.

16.2 Driver Design Process

16.2.1 Security

- *Separation of Privileges* - use fine-grained privileges rather than a binary scheme (god mode vs pleb mode)
- *Role-based Access Control* - separating roles from individuals and groups. We make a role that gets a group of privileges, then can assign that role to groups or individuals, who will inherit them
- Instead of using sysadmin for admin functions, have an I-mail admin as a user with privileges
  - Sysadmin can hand off I-Mail rights to a normal user without giving other system permissions (I-Mail admin does not require root admin)
  - Imail user can send/receive messages, cannot see other user's messages or do anything with their messages.
  - I-mail admin can create and remove users

16.2.2 Operations

- User has authentication data, an association with a program (only one file at a time), a list of messages, and (potentially) a message being written
- When a driver operation does not have a direct analog to a file system call, then it is mapped as an `ioctl` operation, as a part of its jumptable
- Any user can access any I-mail account, any user can write/execute their own programs to make use of I-mail without admin approval
- `read write` are obvious, and they're mapped onto individual messages. `release` used to "clean up" after user ends session.
- `fsync` used to indicate a completed message (all `writes` are done), messages should be sent
  - Creating a new message is another `ioctl`
- User can only read first message - this is mapped to another `ioctl`
  - user can seek within first message using `llseek`'s default implementation

16.2.3 Data Structures

- We want user-level admin that requiring machine's sysadmin privileges
  - Admin can't be deleted, so make it the head of user list (statically allocated. other users are dynamically allocated)
- I-Mail has a singly linked list of users
  - User list not maintained betwee driver installations - reboot will clear users, except for the I-mail administrator. This simplifies our driver to not require disk writeback
  - Each user is attached to a singly linked list of incoming messages called the mailbox, and contains a pointer to a message being written
  - User is also attached to a file structure once opened
- Authenticating a user will add it to the back of the user linked list
- I-Mail owns its own data structures, while the kernel owns file structures (`fd` array)
  - I-Mail provides `release` method to clean up I-Mail data structures after last file strutcture closed by kernel

16.2.4 Locking

- I-Mail operations all traverse user list to find data for specific users - semaphores used because I-Mail is not interacted with from any interrupts
  - We want to allow concurrency as much as possible - use an *RW semaphore* to protect user list (as well as auth information/next user pointer in user struct)
- Individual user is more likely to be read/written to as messages are created, written, and read. We only expect a single process to interact with each user at a time, so use a *semaphore*
- Lock ordering - always consume R/W semaphore before user data semaphore if both are needed. Failure to do can deadlock.
  - I-Mail does *not* order use semaphores - you can only have one at a time.
- **Which locks do I need to acquire?** If you're trying to traverse the linked list or do anything with authentication, you need to obtain the RW semaphore to perform those tasks. For pretty much every operation with user data interaction, you will need a user data semaphore.
  - *Ex:* Message delivery. We find the other user, add our message to their mailbox, and remove the message from our own mailbox. Read access needed to traverse the list, sender's semaphore needed to remove outgoing message, recipient semaphore needed to add to mailbox.
  - Special case: if we delete a user, we obtain user semaphore so that we can ensure deletion synchronized with other user ops.

16.2.5 Blocking (Wait Queues)

- Uninterruptible task state needed to ensure devices requiring attention from sleeping task (eventually) cannot be prevented from receiving it by a user-level interrupt
  - *Ex:* Device asked to perform command and requires an ack - the ack has to be provided by *some* task.
- **Wait Queue**: Doubly linked list of tasks waiting for an event.
  - Tasks are put to sleep by using `wait_event_interruptible` with an interruptible condition to wake it back up
  - The function above is evaluated repeatedly, so never use something that has side effects as a condition when using the macro
  - `wake_up` wakes up all tasks in the wait queue
  - **Blocking** - wait until information returns

  - `read` and `poll` are blocking calls, because you're waiting for the return value. Sleep acts lke blocking, since the process won't do anything for a while. `write` is non-blocking because once information is snet you don't really care (at least in I-Mail).
    * `read` is slept when mailbox is empty, same as `poll`
    * Wake up tasks if the user is deleted

- Wait queue uses a spinlock (has its own synchronization), can be used without optaining semaphore.
  - Task cannot have a user data semaphore while it sleeps, since it can cause deadlock

- **Wait Queue Sleeping Protocol**
  - Release all locks
  - Check sleeping conditions without locking
    * Must be atomically safe to check so that condition evaluation can never inherently cause a crash (very strict requirement)
    * Condition does not lock, so does not wait for critical sections to finish - make sure that any interleaving of the code with the relative data structure
    * *Ex*: comparing a pointer with `NULL` and then dereferencing it is *not* safe because the pointer may have become `NULL` or invalid in between the instructions. Interacting with pointers in general tends to be unsafe. If you really must do pointer logic, try to update a conditional flag every critical section interacting with those pointers.
  - Go to sleep by calling `wait_event_interruptible`
  - When avoken, reacquire locks
  - Recheck all validity requirements
  - If sleeping condition is still valid, then spurious wakeup - restart the sleeping process

16.2.6 Dynamic Allocation

- Need to decide what interface we want to use for allocating memory for data structures dynamically - in I-Mail we don't have anything especially big so we choose between `kmalloc` and slab caches.
  - Walking user list is common, so we can use a slab cache there - if it's all in one page then we are more likely to have TLB hits while doing the traversion.
  - Mailbox traversion (fully) is not very common, so a simple `kmalloc` is sufficient.
  - Structures allocated when I-mail admin executes add user, messages allocated when user starts writing a new one

- Deallocate messages when a user deletes it from mailbox - when user is deleted, delete all messages in mailbox then delete the user
  - When I-Mail is shutdown, it cascades down - clean all messages, then all user structs
  - When message is sent but can't be delivered, just delete the message instead of returning the undeliverable
    - Message delivery deletes the message atomically under protection of sender semaphore
  - User deletion requires consuming user data semaphore, ensuring that no files are uing driver when driver's exit routine is invoked so shutdown is safe
- User is using I-Mail when user data deleted… what do we do?
  - Can't just change file structure pointer to the user data structure, since the file structure's pointer may have been dumped into a register, or waiting for user data semaphore
  - We have to wait until no other task is using structure - start by removing user from the list so that no new traversions will be able to authenticate
  - `private_data` field of file structure is unprotected, use the field carefully. Be careful about reading from this field, ensure `NULL` checks every time we load from the field. But this creates significantly more complexity within our driver to make all these checks
    - Defer the deallocation to `release`, which is only called after all files are closed, so we can safely get rid of stuff then. In short - take the user "out of the game" as soon as it's deleted, defer memory deallocation until you can confirm all files referring to this data are closed (file structure is owned by kernel, technically).
    - Tasks will know that user has been deleted, I-Mail operations can compare user data structure's file structure pointer with `NULL`

17 MP3 Execute

- SysCall Linkage Context

linkage return address
EBX
ECX
EDX
ESI
EBP
EAX
DS
ES
FS
IRQ #
CS
EFLAGS
ESP
SS

– This is the stack context after the syscall makes the call to the execute function. Execute immediately saves the current `ESP` and `EBP`, so that it can later restore them in `HALT` and use `RET` to jump back to the linkage.