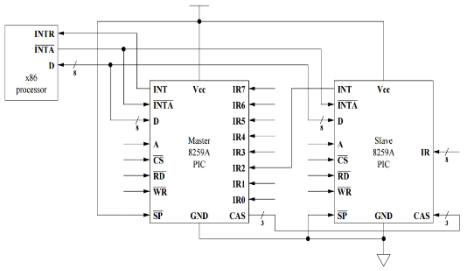
8259A Facts:

- The PIC internally masks all interrupts that are below the priority of the one that is being serviced.
- Steps:
  1. PIC raises the INT
  2. Processor strobes INTA' (used as a synchronization since PIC can be slow)
  3. PIC writes IRQ to data bus
  4. Processor uses this to index into **IDT**
  5. Processor sends and EOI(End of interrupt) telling the PIC that the interrupt has been handled**. If EIO is not received, the PIC will still think that the interrupt is being processed.**
  6. Interrupt handler is executed and then interrupt is unmasked



Pins on PIC:

- INT is only connected to the master PIC
- INTA' is connected to all PICS
- A and CS' are shown in the **PORT ADDRESS BUS** below
- The D is connected to all PICS and is internally connected to the **PORT DATA BUS**
- RD' and WR' are used from the processors perspective to tell the PIC what it is doing.
- **SP' is pulled high on the master**
- **SP' is grounded for all slaves**
- CAS bus is connected from the master too all of the slave PIC. And is used by the master to tell the PICS which slave has control of the data bus.

8259A Initialization:

- **four initialization control words (ICWs)**
- The first word, **ICW1**, **on either 0x20 or 0xA0** and tells the PIC that it is being initialized, that it should use edge-triggered input signals, that it is operating in cascade mode (i.e., using more than one 8259A), and that four control words will be sent in all. (M:0x11, S:0x11)
- Remaining words go to second port. **ICW2** tells the master 8259A is mapped to interrupt vectors 0x20 through 0x27, and the slave 8259A is mapped to interrupt vectors 0x28 through 0x2F. (M:0x20, S:0x28)
- The specific IR pin used in the master/slave relationship is specified by **ICW3**. (M:0x04, S:0x02)
- Finally, **ICW4** specifies the 8086 protocol, normal EOI signaling, and a couple of other (unused) options. (M:0x01, S:0x01)

Interrupt Chaining:

- Have a linked list of interrupt handlers associated with an interrupt line
- Similarly, although only a single interrupt is generated when an interrupt controller's input line goes high, it is possible to connect more than one device to that input, in which case any of the devices so connected may have been the source of the interrupt. Hooking multiple devices to an interrupt line typically also requires that the software allow chaining of interrupt handlers, and furthermore that the the devices associated with the chain can be queried for their interrupt status. Clearly, only the device that generated the interrupt should receive service; others should be ignored. When an interrupt occurs, control is passed to the handler for the first device, which accesses device registers to determine whether or not that device generated an interrupt. If it did, the appropriate service is provided. If not, or after the service is complete, control is passed to the next handler in the chain, which handles interrupts from the second device, and so forth until the last handler in the chain completes. At this point, registers and processor state are restored and control is returned to the point at which the interrupt occurred.

Installing Interrupt Handler:

1. **Request_IRQ()** is used to install the handler for an interrupt within the action list (called by device driver)
2. If there are already interrupt handlers associated with the line, then the shared flag (**SA_SHIRQ**) must be set and match for all
   a. Otherwise, the handler is installed without a problem

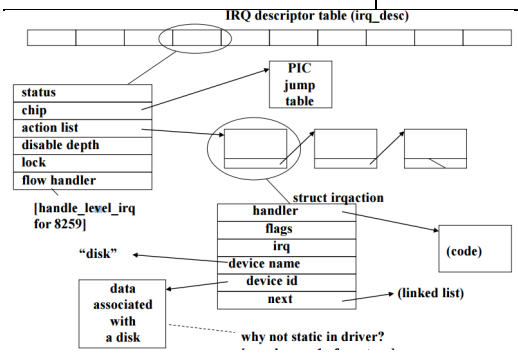
Note: In DOS, interrupts were linked using jmps and this caused inability to install an uninstall any IRQ handler

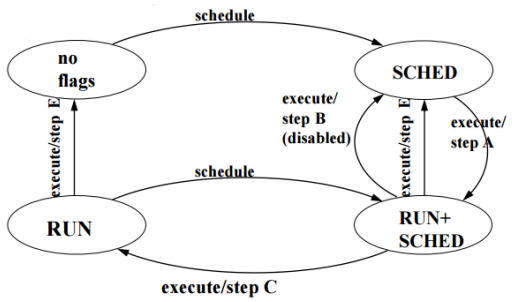
Uninstalling Interrupt Handler:

1. Linux calls function **free_IRQ(uint IRQ, void * dev_id);**
   a. This goes into IRQ desc table and steps through action list to find the match for the specified dev_id.
   b. If the device is found, then the irqaction structure is removed from the list
2. If this is the last handler in the action list, then this IRQs **shutdown()** function is called from the PIC jump table.

Interrupt Invocation and execution:

1. When INT occurs, x86 records IRET context and switches stack using **TSS**.
2. Pointer in the **IDT** pointes to a asm linkage function that pushes the IRQ number onto stack
3. Then **do_IRQ()** is called which uses the interrupt vector index into the **IRQ Desc Table**
4. do_IRQ() clears interrupts and works within a critical section
5. First, **ack()** from the PIC jump table is called
   a. Masks interrupt on PIC and sends **EOI**
6. Executes all handlers in the action lists unless it has been disabled by software via **IRQ_DISABLED** status flag and set **IRQ_PENDING** flag and will be executed from software when last enable is called
7. If allowed to execute, then **handle_IRQ_event()** is called**;** if **SA_INTERRUPT** is not set, then this sets the interrupt flag and starts to step through each struct irqaction and execute the handler. Handlers query the devices.
8. Lastly, handle_IRQ_event() clears interrupts and return control to do_IRQ()
9. do_IRQ() unmasks interrupt on the PIC and calls **do_softIRQ()**;





General Tasklet Information:

- A Tasklet is a data structure used to wrap the handler for a soft-interrupt.
- Generally scheduled by a hard interrupt handler and is placed into a list of other tasklets of the same priority
- Are used to execute tasks that are not time sensitive – e.g. processing of data provided by a device.
- Executed by do_softIRQ. This is called when do_IRQ() finishes or when a periodic **daemon** wakes up and checks to see if there are any tasklets that need to be executed.
- Tasklet structures are generally declared statically.

next	→ linked list
state	TASKLET_STATE_SCHED, TASKLET_STATE_RUN
count	# of disables
func	pointer to the tasklet function
data	integer which can be used by the tasklet function

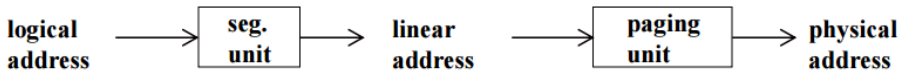
do_softIRQ():

1. First, checks if any interrupt is currently being executed and if so, then it terminates
2. Masks interrupts for critical section and checks for pending soft interrupt
3. If there are any, then it uses local_bh_disable() to record the fact that it is exec a tasklet.
4. Then it enables interrupts and executes the handler functions for each tasklet that was pending at the start. i.e. it walks the regular and high priority list
5. set TASKLET_STATE_RUN atomically (if already set, stop)
6. check if tasklet is software disabled (count field)
   a. if so, clear TASKLET_STATE_RUN
   b. leave the tasklet in the linked list for this priority
   c. set the pending bit for this priority and daemon will try again later
7. clear TASKLET_STATE_SCHED, then execute handler, then, clear TASKLET_STATE_RUN
8. Do steps 5 – 7 for all pending tasklets in list.
9. It disables interrupts again and does local_bh_enable().
10. If tasklet already executed was scheduled again, then wake kernel daemon, enable INTS

Larger Paging = Faster Table Walk, More TLB hits (less misses), More fragmentation over smaller pages // Smaller Paging = Slower table walk, Less TLB hits (more misses), Less fragmentation than larger pages // Internal Fragmentation: wasted space from rounding up (ex. allocating a 4KB page when only 3KB needed) // Internal Fragmentation: wasted space from rounding up (ex. allocating a 4KB page when only 3KB needed) // Lesss levels => faster (on TLB miss)

Virtual Memory:

- Virtual memory is the insertion of a level of indirection between the memory address space seen by a program and the physical memory space of a system.
- The hardware **(MMU or the processor)** is responsible for the translation from VM to physical memory.
- X86 uses virtual memory regardless of the privilege level.
- Some pages are mapped into a device's memory instead of physical memory → **Memory Mapped I/O**
- Translation occurs as shown in image below:



VM Advantages (In order of importance):

1. **Protection** - VM prevents programs from accessing each other's data
2. **Sharing** - sharing occurs through use of libraries. Library code is placed in physical memory and mapped into both program's VM
3. **Prevents Memory Fragmentation** - By dividing memory of a program into 4kB pages, the memory of a program does not need to be contiguous in physical memory.
4. **Relocation** - Avoids having to change absolute addresses since the hardware will take care of that for us.

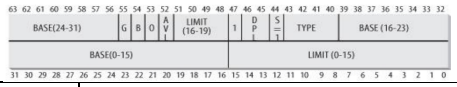
VM Disadvantage**:** Storage requirement for the paging structure and the time overhead to perform translations of VA to physical address.

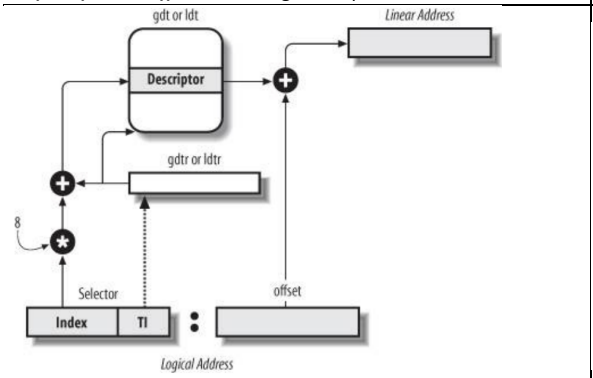
Protection Model:

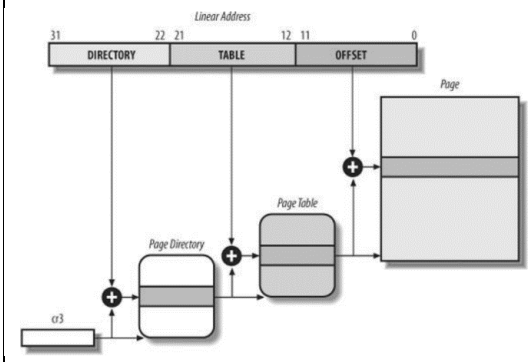
- **Kernel** – level 0 | **User** – level 3
- Idea: higher privileged code will never call less privileged code. And lower privileged code must pass through system calls to request services from kernel.
- **RPL** is in the segment selector – part of logical address (GDT entry)
- **CPL** is in the current code/data segment registers
- **DPL** is stored in the quad word GDT entry (segment descriptor)
- If **(max(CPL, RPL)) > DPL**, then general protection fault.

Segmentation:

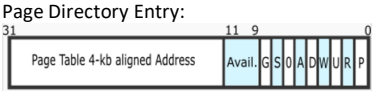
- Segmentation unit converts virtual or logical address to linear address.
- Segments are described by the **Global Descriptor Table** and possibly **Local Descriptor Table**
- **GDTR** holds the physical address to the GDT and a 16-bit limit (size -1 in bytes)
- GDT stores segment descriptors which contain the base address, max offset of segment, and the DPL.
- To convert to linear address, you take your VA and add the offset stored in the GDT segment descriptor.
- Segment registers select the segment being referenced - CS, SS, DS, ES, FS, GS # Only one LDT used.



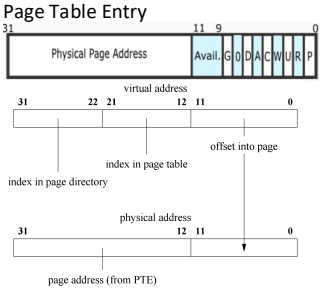




Page Directory Entry:



Page Table Entry



Task State Segment:

- Entries in the GDT can describe **TSSs** which hold information pertaining for a particular program.
- Important components: **SS0 and ESP0** ---> used when we switch from user mode to kernel mode.
- SS0 is the stack segment selector for the kernel
- ESP0 is the offset within the segment to get to the start of the kernel stack. These combine to form the virtual address or the logical address.
- Other components: ESP0 SS0 … CR3 EIP EFLAGS EAX ECX EDX EBX ESP EBP ESI EDI ES CS SS DS FS GS LDTR IOPB

Paging:

- Page States: **Present**, **Swapped Out**, **Non-existent**
- Present means that a program has access to allocated page
- Swapped out means that a program has access to the page but the page has been moved to a **swap disk**
- If a program tries to access a page that is non-existent, this will cause a **page fault**.
- If a program tries to access a page that has been swapped out, the OS has the option to swap the page back in or send a signal to the program aka Segmentation Fault.
- Motivation for having PD and PT is to save space and have **4kB consistent size and alignment** for PD, PT, and pages.
- Registers associated with paging:
  - **Cr3** – aka PDBR is used to point to the **physical address** of the page directory
  - **Cr4** – used to enable 4MB pages ---> set 0x0010 bit
  - **Cr0** – used to enable paging on the processor ---> set MSB

Transition Lookaside Buffers (TLBs):

- Caches the result of paging translation
- This is done by mapping 20 MSB bits Linear Address to 20 MSB bits of the physical address corresponding to the start of the page.
- Number of TLBs is fairly small ---> around 32 to 64 are usually stored at a given time.
- Storing too many TLBs can be bad because then that becomes the bottle neck.
- Must clear the TLB when performing a context switch because program should not access each other's physical memory by accident.
- **TLB hit** occurs when we successfully find the translation result in the TLB
- **TLB miss** occurs when we have to perform the translation to find the physical address.

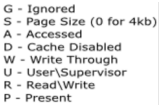
Paging

```
unsigned int i, j;
for (i = 0; i < PAGE_DIRECTORY_SIZE; i++)
page_directory[i] = 0x00000000;
for (j = 0; j < PAGE_TABLE_SIZE; j++)
page_table_0[j] = 0x00000000;
page_directory[0] = ((unsigned
long)(page_table_0) & PTBA_MASK) |
PRESENT_MASK | R_W_MASK;
page_table_0[VIDEO_VIRTUAL] = VIDEO_START
| R_W_MASK | PRESENT_MASK;
page_directory[1] = KERNEL_START |
PAGE_SIZE_MASK | R_W_MASK | PRESENT_MASK;
clear_in_cr4(CR4_PAE_FLAG);
set_in_cr4(CR4_PSE_FLAG);
write_cr3((unsigned long)page_directory);
set_in_cr0(CR0_PG_FLAG);
```

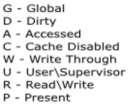
Page Directory/Table Entries:

- Protection: each PTE has a privilege level bit (U) which is required for use in address translation. When U is set, everyone has access. When U is 0, **privilege level** check is made ---> max(CPL, RPL) < 3 to use translation ; Read only / read+write.
- Performance: global flag (G). If set, the PT or page is present in all programs VA spaces in the space place. This allows retention of TLB translations when PDBR is changed.
- 4kB and 4MB pages have separate TLBs.
- Using larger pages reduces the number of TLBs required, but risks fragmentation due to larger amount of contiguous memory being used.
- To use 4BM pages, the page size bit (S) needs to be set in the PDE.

PDE:



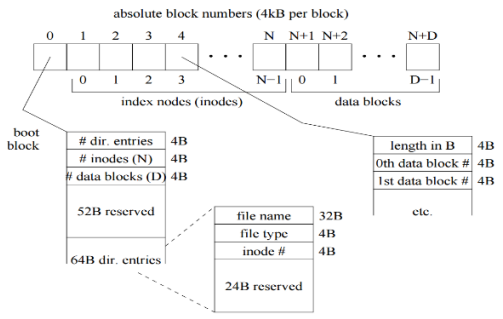
PTE:



File System:

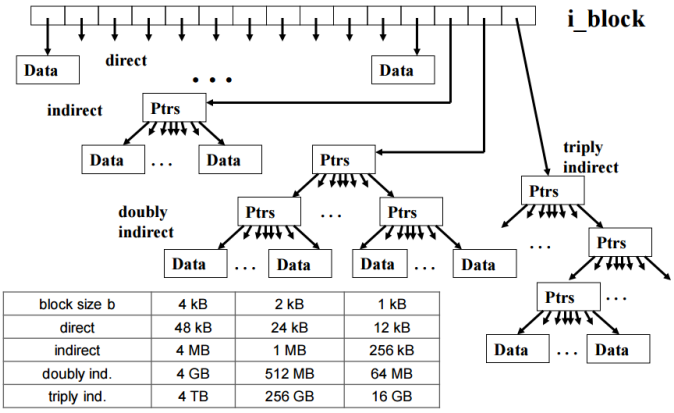
- Boot block: block of data at the start of the file system that is given to the OS upon boot.
- Dentry: represents a directory entry, a single component of a path.
- Inode: contains information about the data blocks associated with a file and contains the length of the file in bytes. Each file has its own inode.
- Data blocks: 4kB blocks of data.
- 1KB Superblock: A specific mounted file system. Volume name/FS check/Size selection, redundant.
- 32B*32 GD: Shortcut to B/IN bitM and iNode T/D

```
typedef struct {
    unsigned long direntries;
    unsigned long NumberOfinodes;
    unsigned long datablocks;
    unsigned char reserved[BOOT_RES_SIZE];
    dentry_t      files[NUM_FILES_CAP];
} boot_block_t;
typedef struct {
    unsigned char filename[FILE_NAME_SIZE];
    unsigned long filetype;
    unsigned long inodeNumber;
    unsigned char reserved[FILE_RES_SIZE];
} dentry_t;
typedef struct {
    uint32_t length;
    uint32_t block_numbers[MAX_BLOCKS];
} inode_t;
```

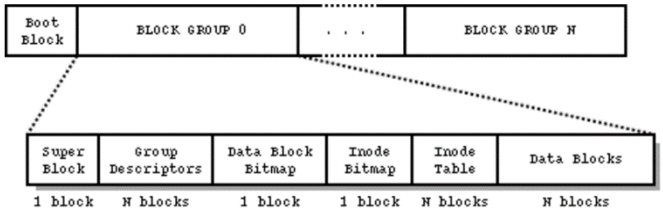


Ext2:

The EXT2 file system, like a lot of the file systems, is built on the premise that the data held in files is kept in data blocks. These data blocks are all of the same length. Every file's size is rounded up to an integral number of blocks. If the block size is 1024 bytes, then a file of 1025 bytes will occupy two 1024 byte blocks. Unfortunately, this means that on average you waste half a block per file. Linux, along with most operating systems, trades off a relatively inefficient disk usage in order to reduce the workload on the CPU. Not all of the blocks in the file system hold data, some must be used to contain the information that describes the structure of the file system. EXT2 defines the file system topology by describing each file in the system with an inode data structure. An inode describes which blocks the data within a file occupies as well as the access rights of the file, the file's modification times and the type of the file. Every file in the EXT2 file system is described by a single inode and each inode has a single unique number identifying it. The inodes for the file system are all kept together in inode tables. EXT2 directories are simply special files (themselves described by inodes) which contain pointers to the inodes of their directory entries.



block size b	4 kB	2 kB	1 kB
direct	48 kB	24 kB	12 kB
indirect	4 MB	1 MB	256 kB
doubly ind.	4 GB	512 MB	64 MB
triply ind.	4 TB	256 GB	16 GB

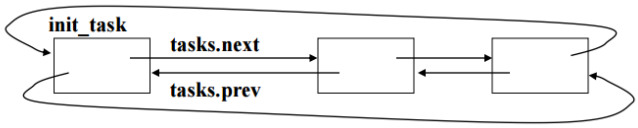


Processes/Tasks:

- Process/Task is a single running instance of a program and linux treats it as a unit of scheduling
- **Process:** Each process provides the resources needed to execute a program. A process has a virtual address space, executable code, a unique process identifier, and at least one thread of execution. Each process is started with a single thread, often called the primary thread, but can create additional threads from any of its threads.
- **Thread:** A thread is the entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled. The thread context includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process.
- **User-level view**
  - each execution context that can be independently scheduled has its own process descriptor
  - traditional process id (pid, a field in task structure/process descriptor)
  - from 1 to 32,767 in Linux, used as task-unique identifier
  - tgid (thread group id) plays process id role for multithreaded applications (common id for all threads in process)
  - most processes belong to a thread group consisting of a single member
- **Kernel view - DO NOT USE RECURSION IN KERNEL - EASILY OUT OF 8K BOUND**
  - kernel must handle many processes at the same time
  - keeps two data structures in a single per-process area (8kB)
    - thread_info structure (keeps pointer to task structure or process descriptor)
    - kernel stack
    - both dynamically allocated
  - architecture-dependent thread info shares space with kernel stack

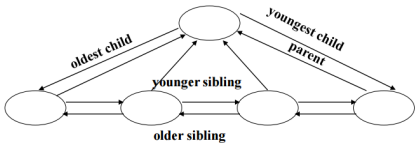
Task Identification and Linkage:

- Task structures
  - placed in a cyclic, doubly-linked list
  - list starts with sentinel: init_task
    - first task created by kernel at boot time
    - persists until machine shut down/rebooted



For system calls: find_pid use a hash table
back links only for deletion
hash function = ((pid * big prime #) >> shift)
Threads run in shared memory spaces, while processes run in separate memory spaces.
swapping between physical memory and disk allows us to use more memory than what's available

parent	parent pointer
children.next	oldest child pointer
children.prev	youngest child pointer
sibling.prev	older sibling pointer
sibling.next	younger sibling pointer
real_parent	original parent (for debugging)

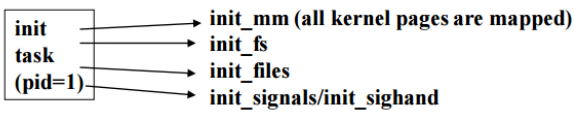


Creating User Processes/Tasks:

- User-level: fork, vfork, and clone system calls
- In Kernel: all map to **do_fork** which calls copy_process() to set up the process descriptor and any other kernel data structures needed for child execution.
- Creating processes at User-level: other programs have to start it. i.e. shell
- First, **fork** is called to create a copy of current program and then **exec** is called which loads the new program and starts it.
- Implementation Strategies of fork: **copy-on-fork or copy-on-write**
  - Copy-on-fork: instantly copies the writable portion of the original program's address space. Address space is instantly disjoint. "Eager" approach.
  - Copy-on-write: Instead of copying data, fork creates a new page directory and creates copies of the page tables which point to the same pages. It also turns off write permission. At first, processes share the same physical address space. When one of the processes tries to write to a shared page, a private copy of the page is made for that process. "Lazy" approach. Avoid useless works.
- vfork: parent blocks while child uses the same address space. After child execs, control of address space returns to parent.
- Clone: clone is used to implement threads. This allows multiple threads in a program to run concurrently in a shared memory space. Unlike fork, children created using clone share parts of the execution context with the calling process -----> such as memory and tgid

Creating a Kernel Thread:

- A task without an associated address space.
- Inherits address space from last user task to execute. All addr spaces map the kernel pages and data structures.
- Init_task is a kernel thread that is created during boot and it persists until shutdown ---> pid = 1
- Other kernel threads: ksoftirqd (softIRQ daemon) and idle process (pid = 0)

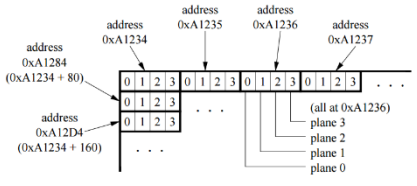


Random Coding Syntax Stuff:

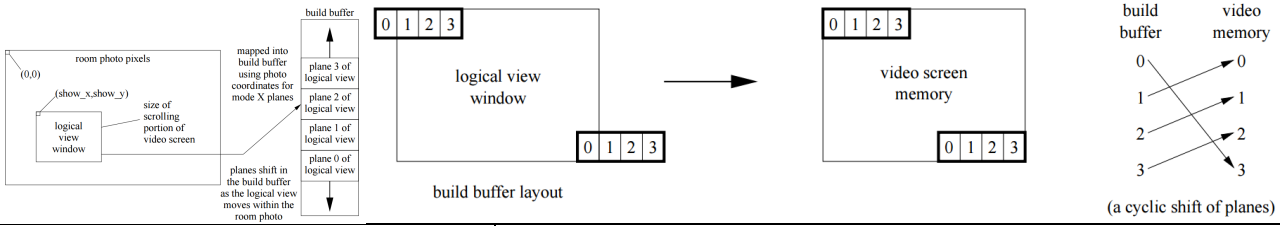
```
pde_t page_directory[PD_ENTRIES] __attribute__((aligned(BYTES_TO_ALIGN_TO)));
pte_t page_table[PT_ENTRIES] __attribute__((aligned(BYTES_TO_ALIGN_TO)));
typedef union random {
    uint8_t val[1];
    struct {
        uint8_t dpl : 2;
    } __attribute__((packed));
} random_t;
```

MP2:

- Video Memory is 32-bit addressable and so we need to write to it by planes.
- We can write to multiple planes at once if we want to through the use of set_write_mask()
- To split the screen, modify the line compare register.
- set colors by modifying VGA palette.



MP2 Build Buffer:



Octrees:

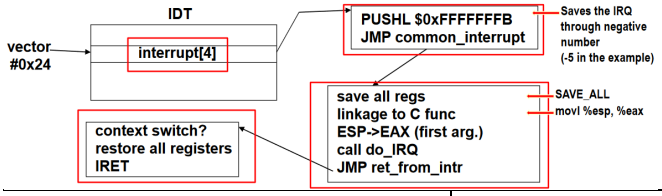
The basic idea behind the algorithm is to set aside 64 colors for the 64 nodes of the second level of an octree, and then to use the remaining colors (in this case, 128 of them) to represent those nodes in the fourth level of an octree that contain the most pixels from the image. In other words, you choose the 128 colors that comprise as much of the image as possible and assign color values for them, then fill in the rest of the image with what's left.

Tux Driver:

- **TUX_INIT**: Takes no arguments. Initializes any variables associated with the driver (see TUX READ LED) and returns 0. Assume that any user-level code that interacts with your device will call this ioctl before any others.
- **TUX_SET_LED:** The argument is a 32-bit integer of the following form: The low 16-bits specify a number whose hexadecimal value is to be displayed on the 7-segment displays. The low 4 bits of the third byte specifies which LED's should be turned on. The low 4 bits of the highest byte (bits 27:24) specify whether the corresponding decimal points should be turned on. This ioctl should return 0.
- **TUX_BUTTONS:** Takes a pointer to a 32-bit integer. Returns -EINVAL error if this pointer is not valid. Otherwise, sets the bits of the low byte corresponding to the currently pressed buttons. Uses copy-to-user();

RTC

```
int32_t rtc_read(int32_t fd, unsigned char *buf, int32_t nbytes) {
 /*if file not opened, return -1*/
   if (fileStatusArray.RTC_STATUS == STATUS_CLOSED) {
    printf("Error: File not opened yet.\n");
    return -1;
   }
   /*check valid fd*/
   if (fd < 0 || fd > MAX_FILE_OPEN) {
    printf("ece391_WARNING::fd out of range.\n");
   } else if(fileStatusArray.FILE_TO_OPEN[fd].filetype != RTC_FILE_TYPE) {
    printf("ece391_WARNING::fd provided does not match an rtc file.\n");
   }
   sti();
   rtcFlag = 0;   /*set the rtc flag to 1*/
   while (rtcFlag<rtcRelativeFreq);   /*check whether a rtc interrupt completed*/
   return 0;
```

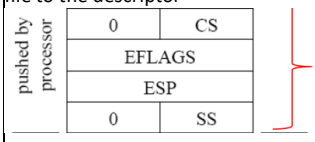



```
int32_t rtc_write(int32_t fd, const unsigned char *buf, int32_t nbytes) {
   int32_t freqency;
   /*if file not opened, return -1*/
   if (fileStatusArray.RTC_STATUS == STATUS_CLOSED) {
    printf("Error: File not opened yet.\n");
    return -1;
   }
   /*check valid fd*/
   if (fd < 0 || fd > MAX_FILE_OPEN) {
    printf("ece391_WARNING::fd out of range.\n");
   } else if(fileStatusArray.FILE_TO_OPEN[fd].filetype != RTC_FILE_TYPE)
    printf("ece391_WARNING::fd provided does not match an rtc file.\n");
   if (buf == NULL) return -1;     /*check whether the buffer is valid*/
   if (nbytes != 4) return -1;     /*the freqency is a 4 bytes int*/
   freqency = *buf;
   if (freqency > HIGHEST || freqency <= 1) return -1;    /*freqency is out of range*/
   if (freqency & (freqency - 1)) {       /*input is not power of 2*/
     printf("input is not power of 2");
     return -1;
   }
   rtcRelativeFreq = HIGHEST / freqency;
   return nbytes;
```

System Calls

Exception & HW interrupt: descriptor type is Interrupt gate, IF = 0, DPL = 0.
System Calls: descriptor type is trap gate, IF don't change, DPL = 3 (Reason for GPError)
# EAX = system call # (asm/unistd.h)
EAX = return value (negative for errors)
# Vector in IDT is system_call, saves registers to stack, check for a valid system call #, call specific routine using a jump table: sys_call_table. Order: ret, B C DX SI DI BP

open -> sys_open -> do_sys_open -> get a free file descriptor, open the file(do_filp_open), attach the file to the descriptor



In dynamically-linked code, code addresses generally linked through jump tables. data addresses often
- make a "fake" call
- call pushes return address onto stack
- return address is located at fixed offset from static variable load from stack and add offset to obtain pointer to variable

```
_errno location:     call getIP          # EAX ← raddr
addr:       addl $errno-raddr,%eax    # adjust return value
            ret
getIP:      movl (%esp),%eax      # read return address into EAX
            ret
```

```
 if(irq_num >= 8) {
      irq_num -= IRQ_NUM_;
 value = EOI | irq_num;
      outb(value, SLAVE_8259_PORT);
      outb((EOI | 0x02), MASTER_8259_PORT);
 }
```

setup_irq(in request_irq)
If this handler is the first
make sure that PIC jump table has proper default functions
clear some status flags
clear any previous software disablement (depth)
call the PIC startup function

System call wrapper:
Push callee saved register
Move argument to register(ebx - edx)
Move system call number to eax
Int $0x80
Pop callee saved register
Ret
System call:
Push all register
Push ebx - edx
Call system call vector table
Pop ebx - edx
Pop all regsiter
iret