

**Seg register**

[0:1]RPL [1:3]: 0 for GDT, 1 for LDT [3:15]: index in table

**Cr3**:aka PDBR is used to point to the start address of the page directory **Cr4**:enable 4MB pages -> 0x0010 bit(bit 7 in PDE), mixed page-> bit 4 Cr0:enable paging on the processor ---> set bit 31

**Paging:** 3states: DNE| E and in physical mem| E,but on disk rather than mem **Each PD entry is a physical addr to PTE, Each PTE is an physical addr to page** 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 Segmentation Fault. Motivation for having PD and PT is to save space and have 4kB consistent size and alignment for PD, PT, and pages.

**Paging:** Given : **64 bi**t virtual byte address, **16 KB pages**, **32-bit** physical byte address. assuming that the valid, protection, dirty and use bits take a total of **4 bits** and all virtual pages are in use.

**Calculate page offset**: log2(page size in byte)

log2(16\*2^10) =14

**Calculate physical page number**: subtracting page offset from total number of bits allocated for physical address = 32 - 14, or 18 bits.

**Calculate Page Table Entry (PTE) size:** adding valid bit, protection bit, etc. to the calculated PPN = 14+8=22

**Calculate number of page entries:** subtracting page offset from the total number of bits of virtual byte address = 64-14 = 50. 2^50 entries.

Total page table size = 2^50 \* 22 bits

**Virtual Memory:** Indirection between memory addr seen by software and memory addr used by hardware, done with 4KB or 4MB in x86.

**The hardware (MMU, processor)** is responsible for translation from VM to physical mem. All programs can assume full access to entire virtual space 4GB

**Why use virtual memory ?:**

**Protection**: one program cannot accidentally or deliberately destroy another’s data.

**More effective sharing**: two (or more) programs that share library code can share a single copy

of the code in physical memory

**Limited Fragmentation:** Can piece together disconnected mem into a single "virtually" connected piece of memory

**Simplifies Program Loading:** Place a program wherever we like and simply redirect the ptrs the program is expecting to their actual locations

**Fragmentation**: Internal fragmentation is a user requests 3.3 MB, but you assign a 4MB block, other 1.7 MB is useless, happens when all blocks of memory are same. External fragmentation is when the user requests 3 MB, but the space between two already allocated memory blocks is too small and the 3 MB doesn’t fit between.

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

**Segmentation:** A Segment is a contiguous portion of a linear addr space (32bit-space of physical addr) Segmentation unit converts virtual or logical address to linear address. Segments are described by the GDT and GDT entries can also describe Local DT 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 registers select the segment being referenced - CS, SS, DS, ES, FS, GS

**Interrupt Descriptor Table (IDT)**

**1**. Associates the interrupt line with the int. handler routine.

**2**. 256 entries (each 8-bytes) or descriptors; each corresponds to an interrupt vector • hardware interrupts mapped into vectors 0x20 to 0x2F

3. All Linux interrupt handlers are activated by so called: interrupt gates (a descriptor type)

**IDT Initialization**

Kernel initialization(setup\_idt()) fillsall the 256 entries ofIDT with theprovisional (or null) handler

**DPL** set to 3 for user-level program,

**Interrupt Invocation steps**1**.** IDT : interrupt[4] //vector#0x24

2. PUSHL $0xFFFFFFFB

JMP common\_interrupt

// Saves the IRQ through negative number

// (-5 in the example)3. save all regs //SAVE\_ALL

linkage to C func //movl %esp, %eax

ESP->EAX (first arg.)

call do\_IRQ

JMP ret\_from\_intr

4. context switch?

restore all registers

IRET

**Interrupt Descriptor Table (IDT)**

**1**. Associates the interrupt line with the int. handler routine.

**2**. 256 entries (each 8-bytes) or descriptors; each corresponds to an interrupt vector • hardware interrupts mapped into vectors 0x20 to 0x2F

3. All Linux interrupt handlers are activated by so called: interrupt gates (a descriptor type)

**IDT Initialization**

Kernel initialization(setup\_idt()) fillsall the 256 entries ofIDT with theprovisional (or null) handler

**Fast call convention**

Used to pass arguments in registers;

reduces the number of memory accesses required for the call.

fastcall macro in Linux tells gcc to pass args in EAX, EDX, ECX

EAX points to saved registers (regs argument)

**IRQ**

irq # (0-15 for PIC)

Two approaches how to call interrupt handler:

**Chained interrupts:**

"chained" means that those interrupts are just chain of function calls (for example, SoC's GPIO module interrupt handler is being called from GIC interrupt handler, just as a function call)

**Interrupt chaining:** Service multiple handlers from single IRQ line, provides scalability while maintaining simplicity. Only allowed if all handlers agree to it, otherwise only first handler is successfully added. Drawback: long intr, needs compiler action

**Nested interrupts:**

"nested" means that those interrupts can be interrupted by another interrupt; but they are not really HW IRQs, but rather threaded IRQs

**Install intr handler:**

1.Request\_IRQ() (create new action list)is used to install thehandler 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. Otherwise, the handler is installed without a problem

**Uninstall intr handler:**

1. Linux calls function free\_IRQ(); 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 Control Flow:**

Running user code

Receive an interrupt on IRQ 2

Context switch to kernel space using TSS and push user context

Push bookkeeping information

Run kernel handler for IRQ2 based on bookkeeping

Teardown and return to user context

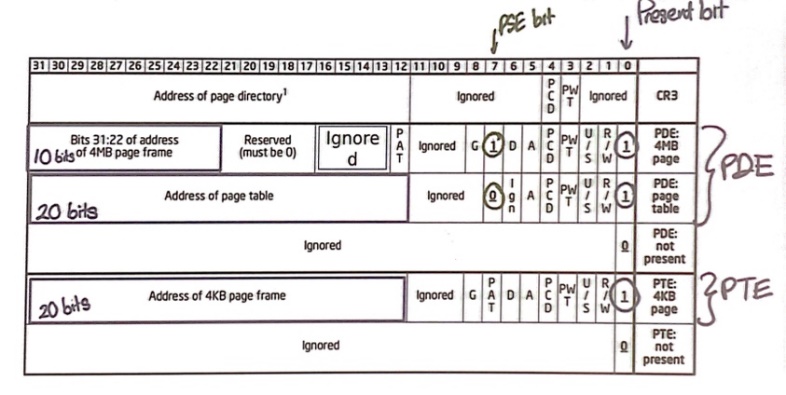
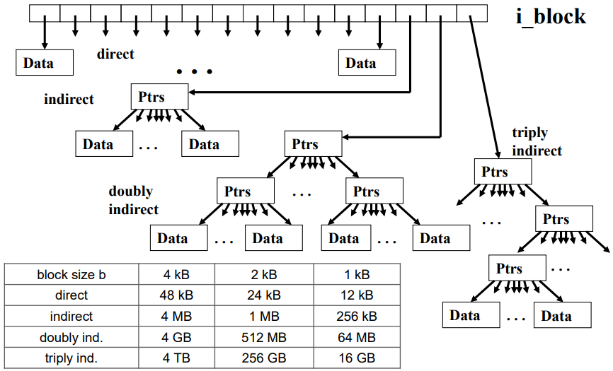
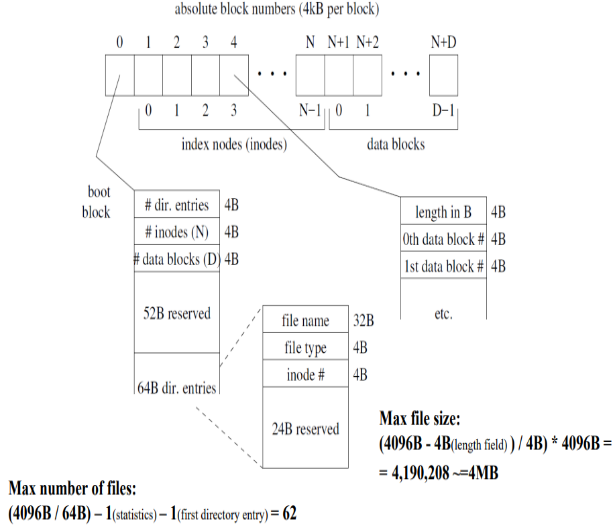
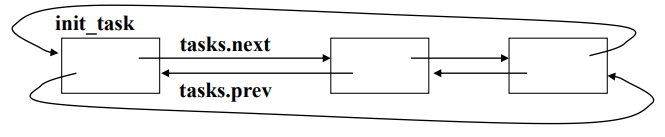
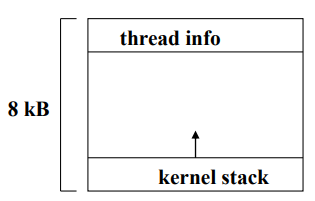
do\_irq()­ – INT\_FLOW\_Handler – handle\_level\_IRQ – handle\_IRQ\_event

**Soft-Interrupt:** (Tasklets) Allows user-level code the ability to have interrupt-like behavior, scheduled and serviced after hard intr at lower priority.

**Trap gate:** does not change int flag

**Int gate:** if disabled (set to 0) when executing. (processor will ignore other intr if it is currently servicing an intr) Service an intr block any other intr of **lower/ equal** priority level=> must send EOI to PIC after service intr

**Difference between sys calls control flow and intr control flow:** Also involves context switch, runs system call handler to grand users privilege to write to file/device.



**System call: (INT $0x80), EAX = system call#, EAX = return value(<0 for error)**

saves registers to stack->check for a valid system call# -> call specific syscall

handler routing using jump table (call \*sys\_call\_table (0, %eax, 4)) -> get return

code from eax ->restore registers ->IRET to resume the User Mode process

**System call wrapper: int open (const char\* name, int flags, int mode)**

open: pushl, movl 16(esp) edx | 12(esp), ecx | 8(esp), ebx | $(syscall #),eax |int

0x80 | cmpl 0xFFFFF001, eax(-1 to -4095 errors)| jb done (valid) | xorl edx, edx |

subl eax, edx | pushl edx | call \_errno\_location | popl ecx |movl ecx, (eax) (save

error number in errno) | or $0xFFFFFFFF, eax (return -1) Done: popl ebx | ret

**In Kernel:** all map to do\_fork which calls copy\_process() to set up the process descriptor and 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:

1B = 8bit

1kB= 2^10B

4kB=2^12B

1MB=2^20B

4MB=2^22B

1GB= 2^30B

**The EXT2 file system:** trades off a relatively inefficient disk usage to reduce the workload on the CPU. Not all blocks in the file system hold data (some used to describes the structure). EXT2 defines each file with an inode data structure. Every file is described by a single unique inode. (kept in inode tables). Directories contain pointers to the inodes of their d\_entries.

**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. **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

**Process/Task**: (a single running instance of a program)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 1 thread of execution. Each process is started with a single thread(primary thread)(can also create additional threads from any of its threads). **Thread:** A thread is the entity within a process. All threads of a process share its virtual addr space and system resources. Each thread maintains exception handlers, scheduling priority, thread local storage, unique thread identifier. 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 pid(traditional process id):a field in task structure/process descriptor), from 1 to 32,767 in Linux, used as task-unique identifier tgid(thread group id): pid for multithreaded applications (common id for all threads in process). Most processes belong to thread group consisting of single member

**Kernel view:** kernel handle many processes at a time; keeps 2 data structures in a single per-process area (8kB): kernel stack; thread\_info structure(keeps pointer to task structure or process descriptor). Both dynamically allocated architecture-dependent thread info shares space with kernel stack

**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/reboot init\_task(a kernel thread created during boot and persists until shutdown,pid = 1) Other kernel threads: ksoftirqd (softIRQ daemon) and idle process (pid = 0)

**Why use hierarchic strut**: 1 level P contain 2^22 entries of 4b each -> too big, 2-level save overhead memory from pages that do not exist/ are not used

**TSS(Task State Segment):** Descriptors in the GDT can describe aspects of program states **Important components: SS0 and ESP**0 -> 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.

**TLB**: 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 TLB is fairly small:32 to 64 are usually stored at a given time. Storing too many TLBs is bad because then that becomes slower. **Clear the TLB** when performing a **context switch** because program should not access each other’s physical memory by accident. **TLB hit** -> when we successfully find the translation result in the TLB **TLB miss** -> when have to perform the translation to find physical addr

**TLB**: 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 TLB is fairly small:32 to 64 are usually stored at a given time. Storing too many TLBs is bad because then that becomes slower. **Clear the TLB** when performing a **context switch** because program should not access each other’s physical memory by accident. **TLB hit** -> when we successfully find the translation result in the TLB **TLB miss** -> when have to perform the translation to find physical addr

**11 bits not used** in PTE,can be used to **implement permission/optimizations**

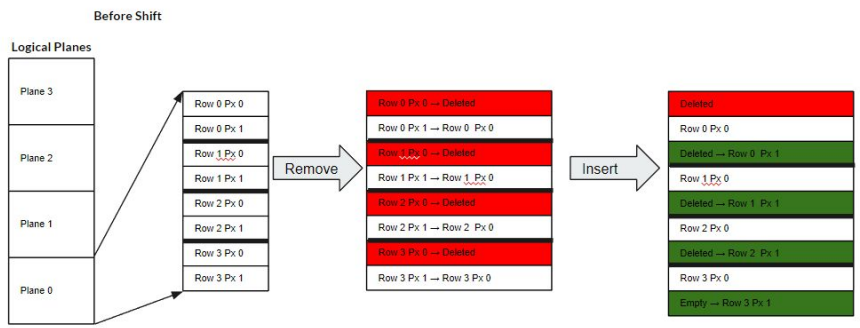
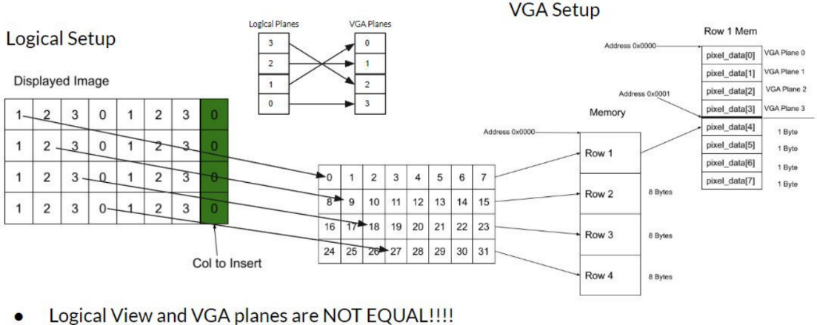
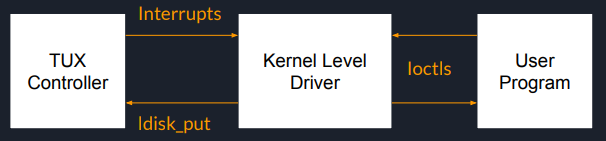
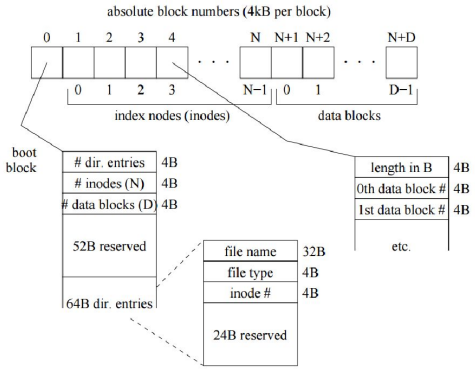
Protections: User(access to anyone)/Supervisor(PL<3) page or page table

**Optimizations:**TLB(4kB and 4MB pages have separate TLB)

**Global flag(G).** Used for kernel page. If set, the PT or page is present in all programs' VA spaces. TLB translations not flushed when PDBR changed.

**\*Translate virtual address to physical address:** The first 10 bits (offset ) indicate the page directory entry. Added to the PD stored in CR3 to give the specific entry. **Memory access 1:** access the data at that entry to get the correct page table entry**.** The next 10 bits are then used to indicate the index into the page table. **Memory access 2:** Access the page table to find the correct page frame Final 12 bits are used to offset into the page(byte offset so you need 12 bits). **Memory access 3**: data at the correct page frame. So there are 3 memory accesses involved \*(\*(CR3 + 4 \* (10 high bits)) + 4 \* (10 mid bits)) + page\_offset \***How memory fragmentation can negatively impact a system without virtual memory** If you have a 128MB system and you have three programs: A(16MB), B(32MB), and C(64MB). These programs are put into contiguous memory locations, leaving 112 MB till the end free (16MB). Let’s say B terminates, that means there are 48MB free. However, the memory is spread out, so even though there is, in total, 48MB free, only a program that’s 32MB or less can be put into memory **\*How does the Linux kernel track the number of uses of a file structure, memory map, or other structure shared between multiple tasks? Why does it do so?** The f\_count member of the file\_struct (and also the fs\_struct) tracks the number of file descriptors referring to the same file, ie it counts the numberof open instances of that file. The reason for tracking this is so that you don’t accidentally close / destroy a file while a process is still using it. \***Explain the advantages and disadvantages of using a 4-MB page size rather than a 4-kB page size to support paging?**They allow for faster allocation of larger memory sizes. However, 4 KB pages result inless internal memory fragmentation which results in more efficient memory usage. (In addition, with 4KB, you can toggle page tables on and off to save more space).**Paging = internal fragmentation transparent to programmer Segmentation = external fragmentation \*TLB workflow** On a virtual memory access, the CPU searches the TLB for the virtual page number of the page that is being accessed, an operation known as TLB lookup. If a TLB entry is found with a matching virtual page number, a TLB hit occurred and the CPU can go ahead and use the PTE stored in the TLB entry to calculate the target physical address. Now, the reason the TLB makes virtual memory practical is that because it is small—typically on the order of a few dozen entries—it can be built directly into the CPU and it runs at full CPU speed. This means that as long as a translation can be found in the TLB, a virtual access executes just as fast as a physical access. **TLB miss** is there is no TLB entry with a matching virtual page number **\*Explain the advantages and disadvantages of using a 4-MB page size rather than a 4-kB page size to support paging?** They allow for faster allocation of larger memory sizes. However, 4 KB pages result inless internal memory fragmentation which results in more efficient memory usage. (In addition, with 4KB, you can toggle page tables on and off to save more space). \***Explain the tradeoffs between handling translation lookaside buffer (TLB) misses in hardware and handling them in software (via an exception mechanism)**Handling them in hardware is much faster but disguises it from the os since there is no exception. The lack of an exception gives less options to the OS. \***Explain the benefits of using linked lists of handlers (actions) to support interrupt chaining. Comment on the drawbacks of chained handlers in general.** Benefits are that it lets you remove and add handlers relatively easily. Drawbacks are that if there’s >1 device, you must query devices to see if they raised an interrupt.. For 1 device you have to avoid stealing data/confusing the device. E.G sending 2 characters to serial port when there’s only 1 interrupt declaring the port is ready on and off to save more space). \***Describe the assembly linkage used to map hard interrupts into calls to do IRQ.** Pushal Call do\_irq Popal iret \* **In what cases is having more than two levels of protection rings supported by hardware useful?** Some operating systems give device drivers elevated but not kernel privilege \* **Explain the differences between disabling an interrupt in software and executing CLI** CLI masks any interrupt on the same process from occurring until the critical section is finished (STI). Disabling an interrupt in software means you mask a specific interrupt from executing until re-enabling it. It’s the difference between locking the bathroom door so no one can get in until you’re done (CLI/STI) and handcuffing a person to a chair so they can’t move at all until they are released (disable/enable) \***Two advantages of using an interrupt controller over an OR gate for bringing device interrupt lines into a processor’s interrupt input.** 1.You don’t have to query every single device to find out which device sent the interrupt. 2.Having a PIC allows for priority of certain devices. \***Kmalloc vs memcache?** For kmem\_cache\_create(), you're creating a cache for objects of a certain size. So, for example, if you needed to dynamically allocate a specific struct fairly often, it would be more efficient to create and use a slab cache instead of just using kmalloc(). If you're just allocating something once (or a few times) or have varying sizes, kmalloc() would be more useful \***Vector number to be used for an interrupt not clear to any given PIC in a cascaded set of PICs, and how is this issue addressed in the 8259A design?**The vector sent to the master PIC represents which interrupt on the master PIC to service, the master PIC then sends a signal to the correct slave PIC and the slave PIC sends its highest priority device’s data. \***As part of implementing their Tux controller code, a friend adds two new ioctl’s assigning them integer values -42 and 42 using an enumeration construct. Do these values reach your friends ioctl function?** Ioctls use an unsigned long as an id, so -42 would become 65494. The 42 would work, -42 wouldn’t. \***Why are I/O ports protected** using a processor’s hardware privilege level support? It’s a security feature to stop random userspace from writing directly to hardware devices. Also protects against collisions. \***What does “chaining” into an interrupt routine mean?** Calling the original ISR from your new ISR. \* **Why is assembly linkage necessary between the address specified for a system call in the IDT and the C function that implements the system call? Why dont operating systems simply use the C calling convention for system calls?** Because you need to use IRET to return from a system call, which is not supported in C. \***Copy on write** Linux uses copy on write rules and shares any non-writable memory maps. For instance if 2 programs use the libc shared library, there will only be one instance of it just mapped to 2 different virtual addresses. Also if the two programs are identical and do not modify memory (only read from it), then copy on write will keep only one copy with 2 different virtual addresses until the program writes. At that point, the program is copied to 2 different physical addresses. **What special data is stored in the first 1024 bytes of the PC’s memory?** Interrupt vector table \***why neither the master nor slave PIC alone has enough information to know what data to send to the CPU during an INTA** Need to know which one has priority over the other \***An End Of Interrupt (EOI)** is a signal sent to a Programmable Interrupt Controller (PIC) to indicate the completion of interrupt processing for a given interrupt. An EOI is used to cause a PIC to clear the corresponding bit in the In-Service Register (ISR), and thus allow more interrupt requests of equal or lower priority to be generated by the PIC **Every interrupt vector** has its own irq\_desc\_t descriptor **Descriptors are grouped** together in irq\_desc array, a data structure supported by Linux **When a device driver calls the request\_irq()** function a new structure to represent the handler is allocated and initialized **\*TLB:** Only operation: flush TLB entries include/asm-i386/tlbflush.h. **movl %0 cr3:** flush all TLB entries. invlpg addr: flush a single TLB entry • More efficient than flushing all TLB entries. **TLB flush when page table changes. The TLB is not transparently informed of changes made to paging structures. Therefore the TLB has to be flushed upon such a change. On x86 systems, this can be done by writing to the page directory base register (CR3) \*As part of debugging your device driver, you decide to count the number of command requests made to the device that have yet to generate an interrupt.** 1. use a shared variable incremented by the command request code and decremented by the interrupt handler. Suggest a better method that does not involve sharing data between these two pieces of code. 2. Make a static variable in the code that’s doing the command. Every time a command is sent, increment the counter. Have a function decrement\_counter() that the interrupthandler can call so that the handler never directly modifies the variable. \***Purpose of having more than 4 GB of physical memory of only 4 GB of virtual memory can be addressed?** Support more than one process since they each have different virtual memory mapping. \***Name one sadvantage AND one limitation of the EXT2 file system?** improved algorithms that greatly increase its speed. n average of half of the block size is wasted for each file. For example, with a a block size of 1024 bytes, each 1025 bytes file would require two blocks **TUX:** a state machine, send cmds to change states **Driver:** Middle man between User and Device **Polling approch:** inefficient, forcing more ints (DO NOT ASK TUX ABOUT THE STATE) **Tux interrupts** us when the state changes **LED spamming:** (Rapid LED change): ack flag, lower when set\_led, set when receive mtcp\_ack use MTCP\_BIOC\_ON and MTCP\_LED\_SET, handle MTCP\_ACK and MTCP\_BIOC\_EVENT

**tuxctl\_handle\_packet** handles all packets received by the computer from the Tux controller. **tuxctl\_ioctl** handles calls from user code (the game) to ioctl



\* show\_screen

\* DESCRIPTION: Show the logical view window on the video display.

\* SIDE EFFECTS: copies from the build buffer to video memory;

\* shifts the VGA display source to point to the new image

**void show\_screen () {**

**unsigned char\* addr;** /\* source address for copy

**int p\_off;** /\* plane offset of first display plane

**int i;** /\* loop index over video planes

/\* Calculate offset of build buffer plane to be mapped into plane 0 of display.

**p\_off = (3 - (show\_x & 3));**

**target\_img ^= 0x4000;** //Switch to the other target screen in video memory.

/\* Calculate the source address. \*/

**addr = img3 + (show\_x >> 2) + show\_y \* SCROLL\_X\_WIDTH;**

/\* Draw to each plane in the video memory. \*/

**for (i = 0; i < 4; i++) {**

**SET\_WRITE\_MASK (1 << (i + 8));**

**copy\_image (addr + ((p\_off - i + 4) & 3) \* SCROLL\_SIZE + (p\_off < i), target\_img); }**

/\* Change the VGA registers to point the top left of the screen to the video memory that we just filled.\*/

**OUTW (0x03D4, (target\_img & 0xFF00) | 0x0C);**

**OUTW (0x03D4, ((target\_img & 0x00FF) << 8) | 0x0D); }**

draw\_horiz\_line (int y) 0-based pixel row number of the line to be drawn

{ /\* buffer for graphical image of line \*/

**unsigned char buf[SCROLL\_X\_DIM];**

**unsigned char\* addr; /\* address of first pixel in build\*/**

**int p\_off; /\* offset of plane of first pixel\*/**

**int i; /\* loop index over pixels**\*/

/\* Check whether requested line falls in the logical view window. \*/

**if (y < 0 || y >= SCROLL\_Y\_DIM)**

**return -1;**

/\* Adjust y to the logical row value. \*/

**y += show\_y;** // logical view window coordinates

/\* Get the image of the line. \*/

**(\*horiz\_line\_fn) (show\_x, y, buf);**

/\* Calculate starting address in build buffer. \*/

//addr = upperleft + (old logical/4)+()

**addr = img3 + (show\_x >> 2) + y \* SCROLL\_X\_WIDTH;**

/\* Calculate plane offset of first pixel. \*/

// 3-(show\_x mod 4) = # of plane

**p\_off = (3 - (show\_x & 3));**

/\* Copy image data into appropriate planes in build buffer. \*/

**for (i = 0; i < SCROLL\_X\_DIM; i++) {**

**addr[p\_off \* SCROLL\_SIZE] = buf[i];**

**if (--p\_off < 0) {**

**p\_off = 3;**

**addr++;= } }**

**return 0; }**/\* Return success. \*/

void **tuxctl\_handle\_packet** (struct tty\_struct\* tty, unsigned char\* packet) {unsigned char buf[2]; unsigned a, b, c;

a = packet[0]; /\* Avoid printk() sign extending the 8-bit \*/

b = packet[1]; /\* values when printing them. \*/

c = packet[2];

switch(a){

**case MTCP\_ACK:**

// set ACK flag to 0

ACK = 0; break;

**case MTCP\_RESET:**

// initialize buffer

buf[0] = MTCP\_BIOC\_ON;

buf[1] = MTCP\_LED\_USR;

tuxctl\_ldisc\_put(tty, buf, 2);

tux\_LED (tty,LED\_save); //set LED

ACK = 0; break ;

**case MTCP\_BIOC\_EVENT:**

LEFT = (c & Left\_mask) >> Leftoffset;

DOWN = (c & Down\_mask) >> Downoffset;

button\_value =((b & Fourbit\_mask) | ((c & Right\_Up\_mask) << Coffset) | (LEFT << Six) | (DOWN << Five));

break; default: return;

} return;

Octree:

Bits store information: More bits = better description, more storage

Color (RRRRRGGGGGGBBBBB) -> 16 bits, 2^16 combinations

Levels (Tradeoff between descriptiveness and possible colors):

Lower level (2): less combination of colors, worse quality. Nodes =2^6 Addr(index) = 2:2:2

Higher level (4): more combinations, better quality Nodes =2^12 Addr(index) = 4:4:4

Hybrid of Levels: Some high quality, Default low quality colors

choose high quality -> Sort by frequency

Extra accuracy? -> average color counts

Screen: 8:8:8 Palette: 6:6:6 Photo:5:6:5

int tuxctl\_ioctl (struct tty\_struct\* tty, struct file\* file, unsigned cmd, unsigned long arg)

//call 自己写的usercode (LED, BUTTON)

**Scroll\_x\_width=rowsize/4**

**P\_off** is the number of plane we have to go down in memory. P\_offset = 2 means start at plane 1 in logical view

**“i” iterate over vga plane**