ECE391
Computer System Engineering

Lecture 20

Dr. Zbigniew Kalbarczyk

University of Illinois at Urbana- Champaign

Fall 2022

Lecture Topics

- **Exam2 Review**

ECE391 EXAM 2

- **EXAM 2 – November 1 (Tuesday)**
  - **Time: 7:00pm to 9:00pm**

- **Conflict Exam**
  - Deadline to request conflict exam:

    Thursday, October 27, by 5:00pm

    (by email to: *kalbarcz@Illinois.edu*)

ECE391 EXAM 2

- **Exam 2 Review Session in collaboration with HKN**
  - Date: Saturday, October 29;
  - Time: from 2pm to 4pm
  - Location: ECEB 1002 (tentative)

ECE391 EXAM 2: Topics

- **Topics covered by the EXAM 2**
  - Material covered in lectures
    (Lecture12, Lecture13, Lecture 14, Lecture 15, Lecture 16, Lecture 17, Lecture 19)
  - MP2 (ModeX, Tux controller)
  - MP3.1 and MP3.2 (file system)
  - Material covered in discussions

NOTE: **Scheduling NOT** included in Exam 2

- **NO Lecture on Tuesday, November 1**

General Study Tips

- Active Recall
  - Design your studying on questions, rather than facts, and attempt to recall the answer
  - Do not passively study notes, this is not very helpful
- Practice Exams
  - Attempt exams without a cheat sheet
  - For all problems you cannot solve look up the answer and add necessary information to your cheat sheet
- Confirm you are correct!
  - Office hours, Piazza, Study groups
- Meet with your group-mates and make sure everyone understands everything from MP3.1 and MP3.2 (the file system)

Lecture Materials - Interrupt Control Flow

- The lecture slides have a lot of very specific information,
- You do not need to study the details !
- Understand the fundamental concepts !

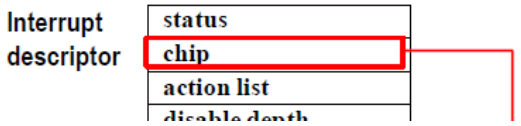
Linux setup_i

(kernel/irq/manag

```
int
setup_irq (unsigned int irq, struct irqaction* new)
{
    struct irq_desc* desc = irq_desc + irq;
    struct irqaction* old;
    struct irqaction** p;
    unsigned long flags;
    int shared = 0;

    if (irq >= NR_IRQS)
        return -EINVAL;
    if (desc->chip == &no_irq_chip)
        return -ENOSYS;

    if (new->flags & IRQF_SAMPLE_RANDOM)
        rand_initialize_irq (irq);

    spin_lock_irqsave (&desc->lock, flags);
    p = &desc->action;
    if ((old = *p) != NULL) {;
        /* Can't share interrupts unless both agree to and are same type. */
        if (!((old->flags & new->flags) & IRQF_SHARED) ||
            ((old->flags ^ new->flags) & IRQF_TRIGGER_MASK)) {
            spin_unlock_irqrestore (&desc->lock, flags);
            return -EBUSY;
        }

        /* add new interrupt at end of irq queue */
        do {
            p = &old->next;
            old = *p;
        } while (old);
        shared = 1;
    }

    *p = new;
```

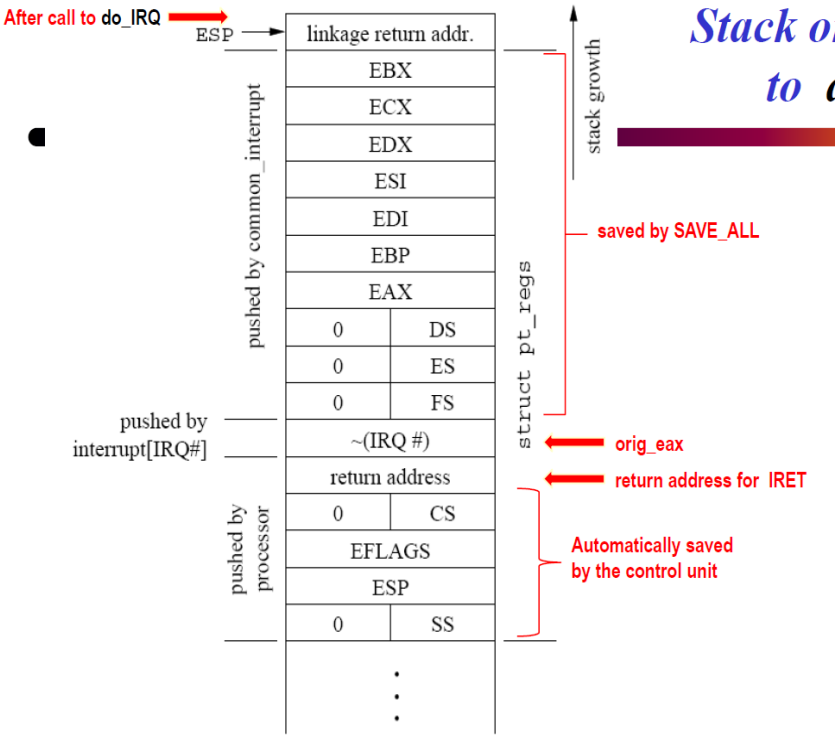
Critical section begins

New action is added at the end of the link list



Lecture Materials - 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 IRQ 2 based on bookkeeping
- Teardown and return to user context

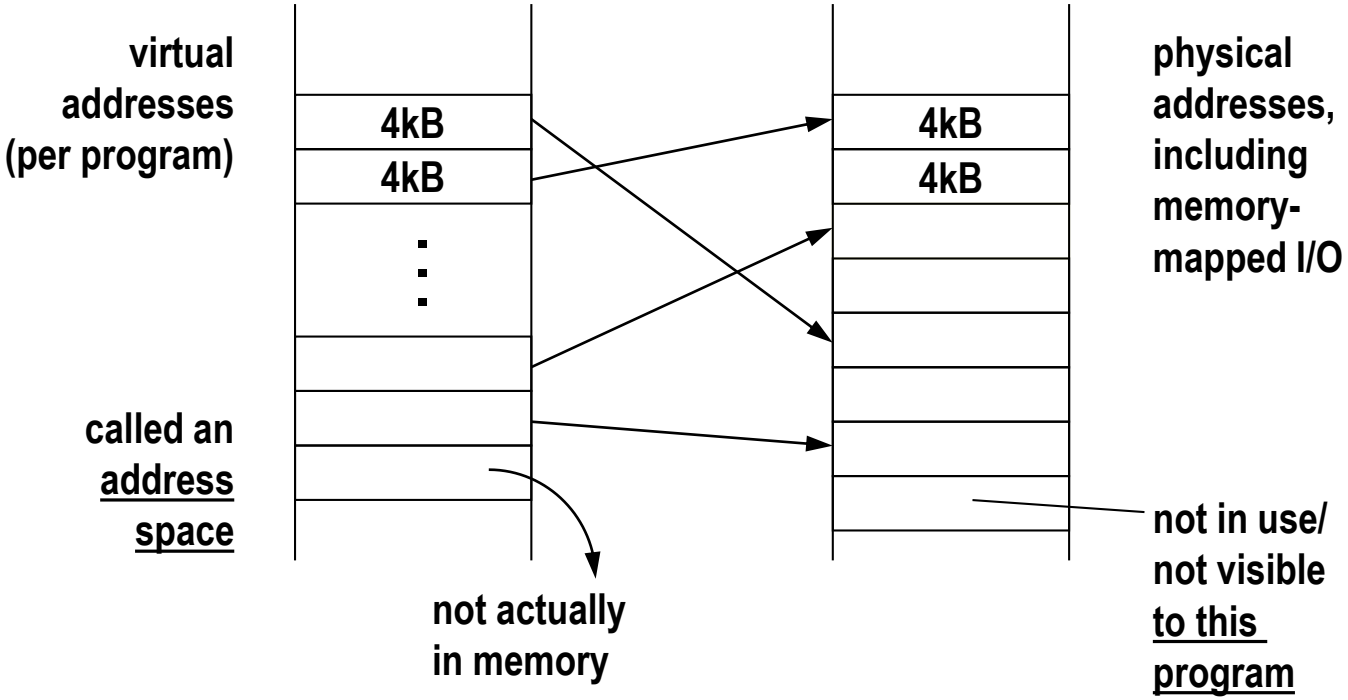


Interrupt Descriptor Table

0x00–0x1F defined by Intel	0x00	division error	
	⋮		
	0x02	NMI (non-maskable interrupt)	
	0x03	breakpoint (used by KGDB)	
	0x04	overflow	
	⋮		
	0x0B	segment not present	
	0x0C	stack segment fault	
	0x0D	general protection fault	
	0x0E	page fault	
	⋮		
0x20–0x27 master 8259 PIC	0x20	IRQ0 — timer chip	example of possible settings
	0x21	IRQ1 — keyboard	
	0x22	IRQ2 — (cascade to slave)	
	0x23	IRQ3	
	0x24	IRQ4 — serial port (KGDB)	
	0x25	IRQ5	
	0x26	IRQ6	
	0x27	IRQ7	
0x28–0x2F slave 8259 PIC	0x28	IRQ8 — real time clock	
	0x29	IRQ9	
	0x2A	IRQ10	
	0x2B	IRQ11 — eth0 (network)	
	0x2C	IRQ12 — PS/2 mouse	
	0x2D	IRQ13	
	0x2E	IRQ14 — ide0 (hard drive)	
	0x2F	IRQ15	
0x30–0x7F	⋮	APIC vectors available to device drivers	
0x80	0x80	system call vector (INT 0x80)	
0x81–0xEE	⋮	more APIC vectors available to device drivers	
0xEF	0xEF	local APIC timer	
0xF0–0xFF	⋮	symmetric multiprocessor (SMP) communication vectors	

Virtual Memory Definition

- What is virtual memory?
  - indirection between memory addresses seen by software and those used by hardware
- What are some advantages/disadvantages?



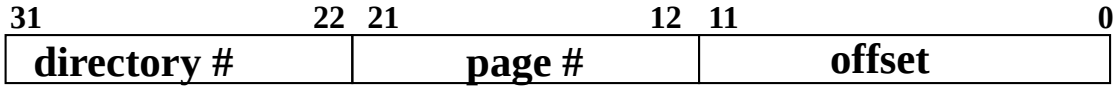
Lecture Materials - Virtual Memory

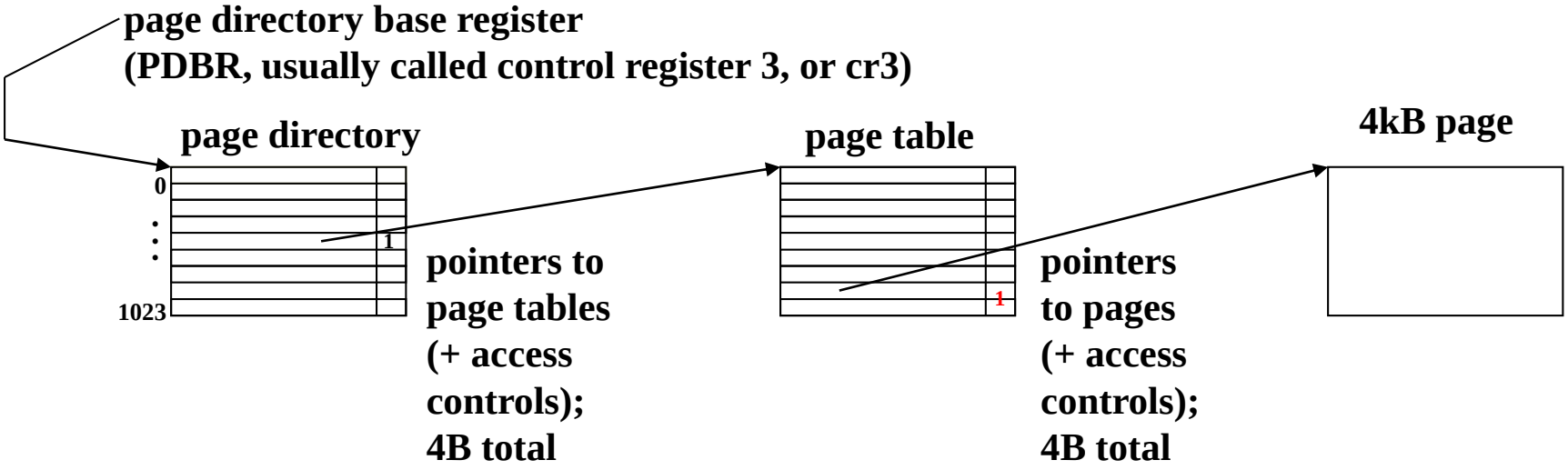
- Protection
  - Programs cannot access memory outside of their space
- Effective Sharing
  - Can share memory between two programs quite easily
  - Libraries, global data
- Limited Fragmentation
  - Can piece together disconnected memory into a single "virtually" connected piece of memory
- Simplifies Program Loading
  - We can place a program wherever we like and simply redirect the pointers the program is expecting to their actual locations

Lecture Materials - x86 Support for VM

- segmentation
- paging

Lecture Materials - x86 Paging





x86 Paging

- Way too slow to do on every memory access!

- Hence the translation lookaside buffers (TLBs)
  - keep translations of first 20 bits around and reuse them
  - only walk tables when necessary (in x86, OS manages tables, but hardware walks them)
  - TLBs flushed when cr3 is reloaded

x86 Paging

- Remember the 11 free bits in the PTEs?
- What should we use them to do?
  - protect
  - optimize to improve performance

- Protect
  - User/Supervisor (U/S) page or page table
    - User means accessible to anyone (any privilege level)
    - Supervisor requires PL < 3 (i.e., MAX (CPL,RPL) < 3)
  - Read-Only or Read/Write

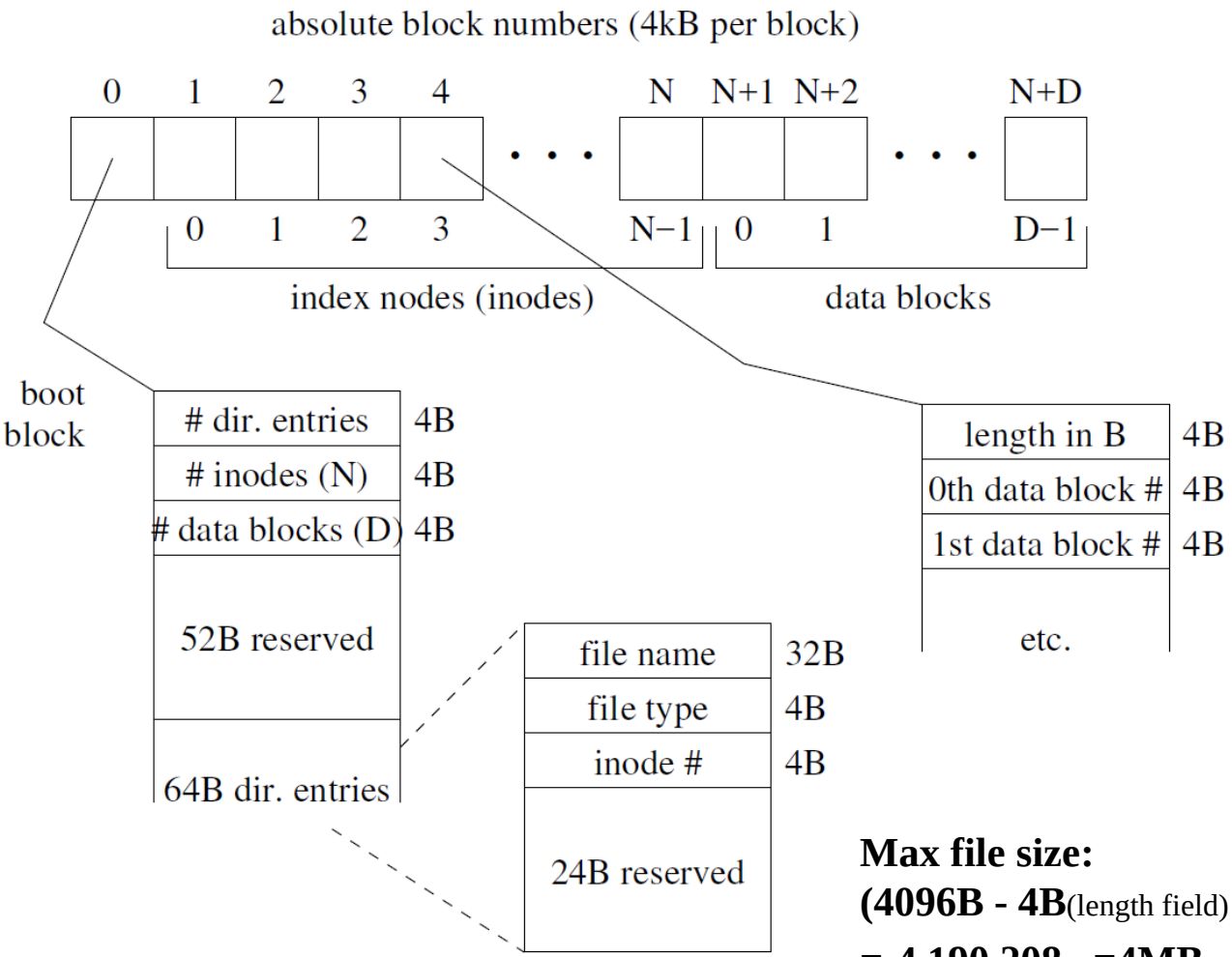
x86 Paging

- Optimize
  - TLBs must be fast, so you can't use many (~32 or 64)
  - nice if
    - some translations are the same for all programs
    - bigger translations could be used when possible
      (e.g., use one translation for 4MB rather than 1024 translations)

x86 Paging

- x86 supports both
  - **G flag—global**
    - TLB not flushed when changing to new program or address space (i.e., when cr3 changes)
    - used for kernel pages (in Linux)
  - **4MB pages**
    - skip the second level of translation
    - indicated by PS (page size) bit in PDE
    - PS=1 means that the PDE points directly to a 4MB page
    - remaining 22 bits of virtual address used as offset
    - x86 provides separate TLBs for 4kB & 4MB translations

MP3 File System

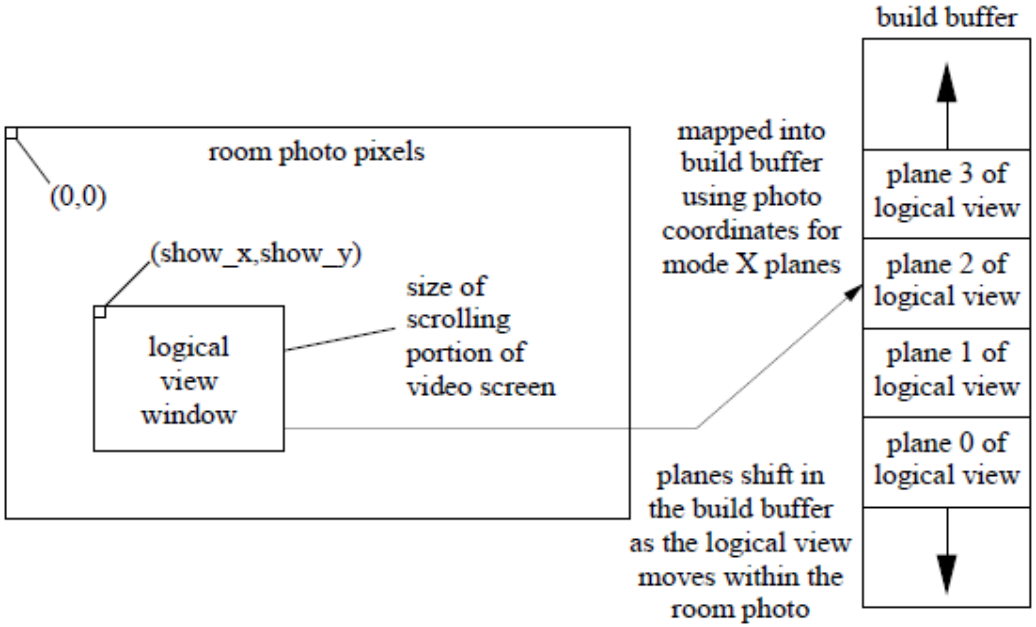


Max file size:
(4096B - 4B(length field)**) / 4B) * 4096B =**
= 4,190,208 ~=4MB

Max number of files:
(4096B / 64B) – 1(statistics)**– 1**(first directory entry)**= 62**

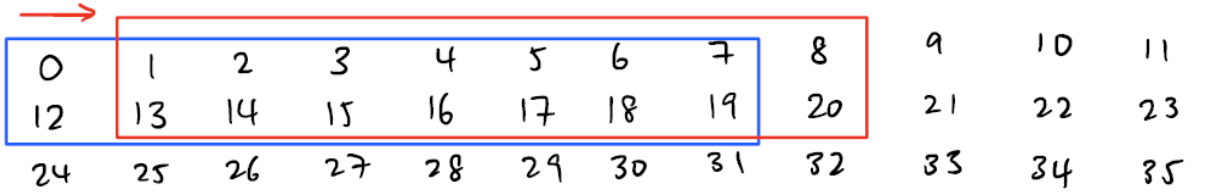
MP2 - Mode X/VGA

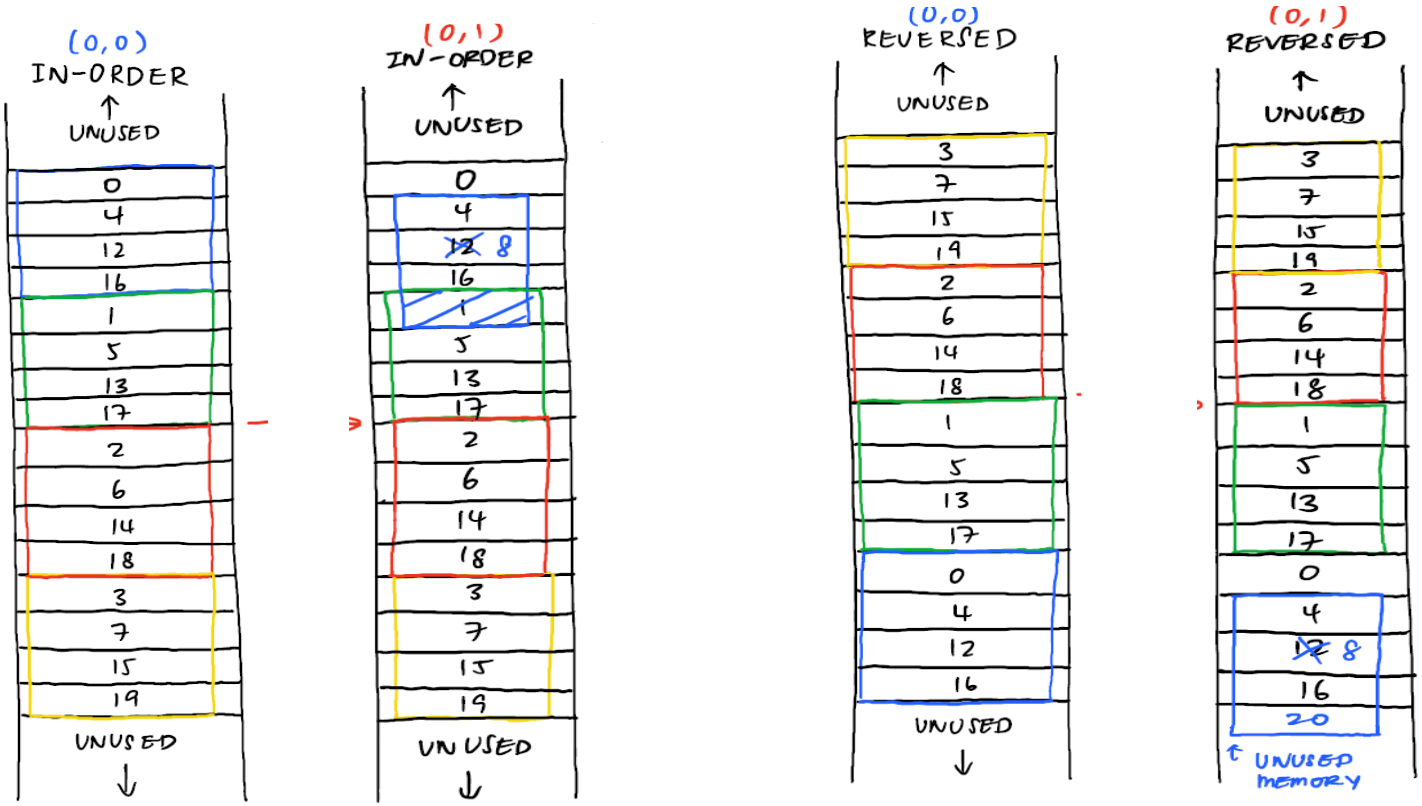
- Planar Format, 4-Byte Address-ability
- Separate Video Buffer and Build Buffer
- Reverse Ordering used for the four planes: Why?
- Palette management



Build Buffer in Mode X

- Consider *a room photo* of 12px and height 3px
- Let the logical view window be 8px and 2px tall and position at (0,0) in the maze world. Current logical view window is represented in blue,
- Let us set up the build buffer





MP2 - Tux Driver

- Synchronization
- Interrupt based approach. Advantages of interrupts over polling?
- How to handle spamming inputs?
- A Tux Driver question on the exam could be of the form
  - We want to implement a new feature, write code to do so.
  - Here is some driver code that is broken. What is wrong and how to fix it?

Example (1)

Problem 4 **Virtual Memory** (**18** points)

A 4-level page table system (Page Directory → Page Table 1 → Page Table 2 → Page) has been created for a computer with a 32-bit address. It is your job to create a page table walking function for this new page table system.

a **Parameters** (**4** points) From the documentation of this 4-level page table system, you recognize the following properties about the size of the tables:

- The sizes of Page Directories, Tables, and Pages could be different (i.e., they are not all the size of a page like in Linux's 3-level page table system).
- The total size of the Page Directory is 4KB.
- The sizes of Page Table 1 and Page Table 2 are the same.
- The offset field in the virtual address is 10 bits wide.
- Page Table and Directory entries are 32 bits long.

Given this, list out the lengths of each of the fields in the 32-bit virtual address:

PDE = _____ bits

PTE1 = _____ bits

PTE2 = _____ bits

Offset = _____ bits

Example 2

- Ben is building an embedded system which has a small physical RAM. He decided that in order to reduce the memory fragmentation he should use 256-byte memory pages.
- He also wants Page Tables to be of size 256 Bytes each and each entry in a table to be of size 4B.
- Your task is to determine:
  - the number of index bits for indexing the Page Directory and Page Table(s) entries and the number of offset bits into the Memory Page.

  - **Page directory:**
  - **Page table:**
  - **Offset:**

Example (3)

- Explain what is TLB (Translation Lookaside Buffer) and why is it used?

- Assume that you allow both 4kB and 4MB memory pages. Would you use same TLB to cache the results of virtual-to-physical address translation for 4kB and 4MB pages?

Example 4

- After initializing your IDT in the kernel, you execute int $0x80 to test your system call vector and it works fine. However, when a user level application tries to execute a system call, a General Protection Fault occurs.
- Why does the fault occur and how do you remedy the issue?
  - DPL in IDT entry corresponding to the system call is wrong.
  - Set DPL to 3

Example (5)

You are provided with the code for initializing IDT entries as well as the assembly linkage for interrupt handlers.

```
void initialize_idt_entries(){
    int i;

    /* NUMBER_OF_VECTS = 256 */
    for(i = 0; i < NUMBER_OF_VECTS; i++){
        idt[i].seg_selector = KERNEL_CS;
        idt[i].reserved4 = 0x00;

        /* EXCEPTIONS = 0x20 */
        if(i < EXCEPTIONS)
            idt[i].reserved3 = 0x1;
        else
            idt[i].reserved3 = 0x0;

        idt[i].reserved2 = 0x1;
        idt[i].reserved1 = 0x1;
        idt[i].size = 0x1;
        idt[i].reserved0 = 0x0;

        if(i == SYSTEM_CALL)
            idt[i].dpl = 0x3;
        else
            idt[i].dpl = 0x0;

        idt[i].present = 0x1;
    }

    /* Set exception handler addresses in IDT */
    SET_IDT_ENTRY(idt[0], divide_error);
    ...
    SET_IDT_ENTRY(idt[31], reserved);

    /* Set linkage addresses in IDT */
    SET_IDT_ENTRY(idt[KEYBOARD_VECT], keyboard_linkage);
    SET_IDT_ENTRY(idt[RTC_VECT], rtc_linkage);
    SET_IDT_ENTRY(idt[TIMER_DEVICE_VECT], timer_device_linkage);
    SET_IDT_ENTRY(idt[SYSTEM_CALL], system_call_linkage);
}
```

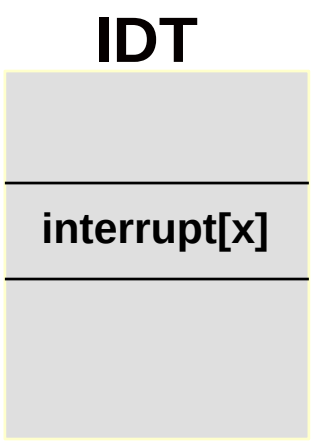
```
.global keyboard_linkage
.global rtc_linkage
.global timer_device_linkage

keyboard_linkage:
    pushal
    call keyboard_int_handler
    popal
    iret

rtc_linkage:
    pushal
    call rtc_int_handler
    popal
    iret

timer_device_linkage:
    pushal
    call timer_device_int_handler
    popal
    iret
```

Example 5(cont.)

- Assume that:
  - a timer device (separate from the RTC) is added to the kernel. This device is connected to IRQ0 of the PIC and corresponds to entry 0x20 in IDT.
  - whenever the timer device generates an interrupt, the kernel should service it immediately.
- Will the new timer device work as expect?
  - NO
  - IDT entries for RTC/keyboard are using Interrupt Gates
  - **“IF”** flag is cleared while executing RTC/keyboard handlers.
  - Processor will ignore timer device if it is currently servicing an RTC/keyboard interrupt.
  - Fix by either using trap gates for keyboard/rtc IDT entries, or by doing STI in keyboard/rtc assembly linkage.

Interrupt Descriptor Table (IDT)

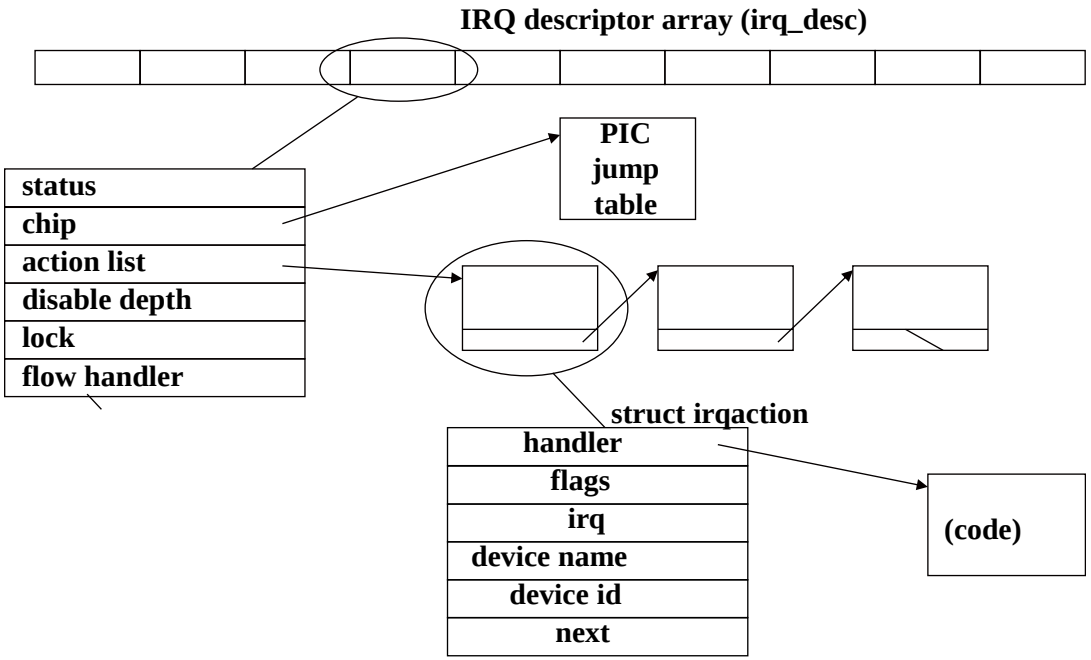
- Associates the interrupt line with the int. handler routine.
- 256 entries (each 8-bytes) or descriptors; each corresponds to an interrupt vector
  - hardware interrupts mapped into vectors 0x20 to 0x2F
- IDT descriptors corresponding to hardware interrupt handlers are activated as so called: *interrupt gates (a descriptor type)*
- Whenever an interrupt gate transfer control to the interrupt handler, the processor clears the *IF flag, i.e.,* interrupts are disabled automatically by the processor



- Before the kernel enables interrupts, it must initialize the *idtr* register to point to the IDT table *(set by the kernel using* `lidt` *instruction)*

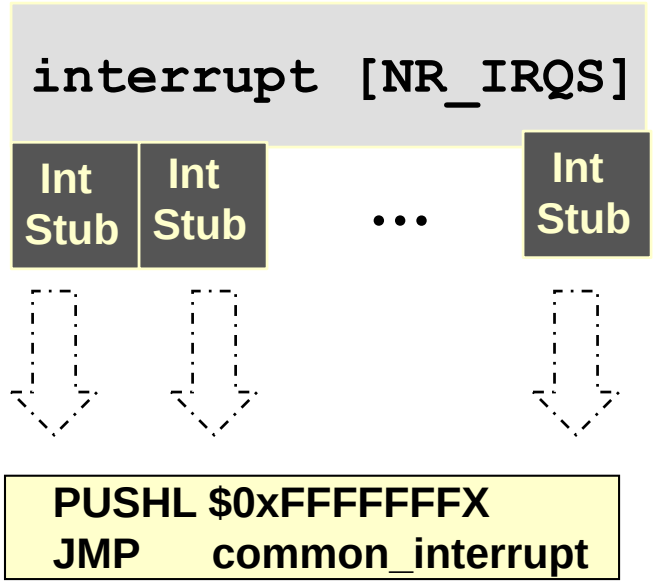
- **IDT Initialization**
- Kernel initialization (`setup_idt()`) fills all the 256 entries of IDT with the provisional (or null) handler

`irq_desc` Array



- 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

interrupt[NR_IRQS] Array



- Kernel maintains one global array of function pointers (`interrupt[NR_IRQS]`) in which it stores pointers to interrupt stubs (`NR_IRQS` is 16 if we use the PIC)

Initialization of Interrupt Gates

- During initialization `init_IRQ()` sets the status field of each IRQ descriptor to IRQ_DISABLED
- `init_IRQ()` updates the IDT by replacing the provisional interrupt gates with new ones

```
for (i = 0; i < NR_IRQS; i++)
     if (i+32 != 128)
          set_intr_gate(i+32, interrupt[i]);
```

- Interrupt gates are set to the addresses found in the `interrupt[NR_IRQS]` array