ECE 391 MT 2 Review

Patrick Marschoun, Noelle Crawfish, Pradyun Narkadamilli

Reminders

- Exam is **Monday, November 6th (7:00-9:00pm)**
  - 2 pages of notes allowed
- Keep the lab clean

Exam Content

- Virtual Memory
- Filesystems
- Interrupts
- MP2 - Modex, Tux, Octrees
- MP3 - Checkpoints 1 & 2
    - Paging, Keyboard/Terminal Driver, Filesystem, etc.

Virtual Memory

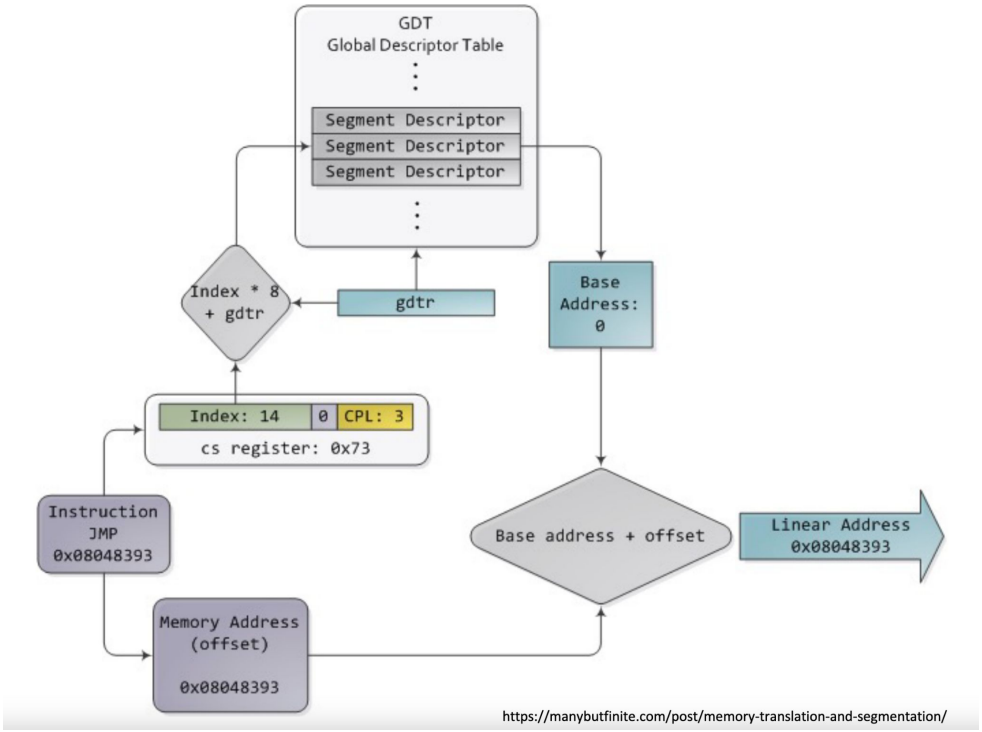
Why do we need virtual memory?

- On a 32-bit system, we can only index 2^{32} = 4GB of memory directly
  - This is not a lot
- We don't necessarily know how much memory a process will want / actually use when it spawns
  - Would like the ability to allocate memory lazily
- Want to provide programs with the illusion of always running in the same place
- Want to provide security between program spaces

Segmentation

- Segment: contiguous portion of linear address space
- Uses GDT to define segments
- Segment descriptors
  - differentiate code (executable vs. possibly readable) from data (readable vs. possibly writable), etc.
  - Describe some parts of program state (ie. TSS)

Segmentation

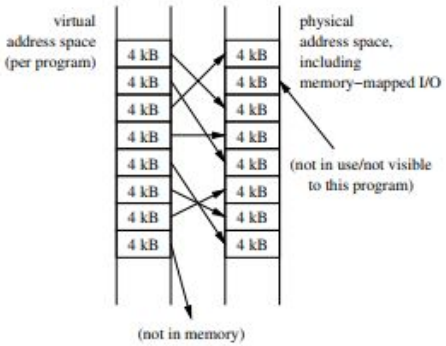


https://manybutfinite.com/post/memory-translation-and-segmentation/

Problems with Segmentation

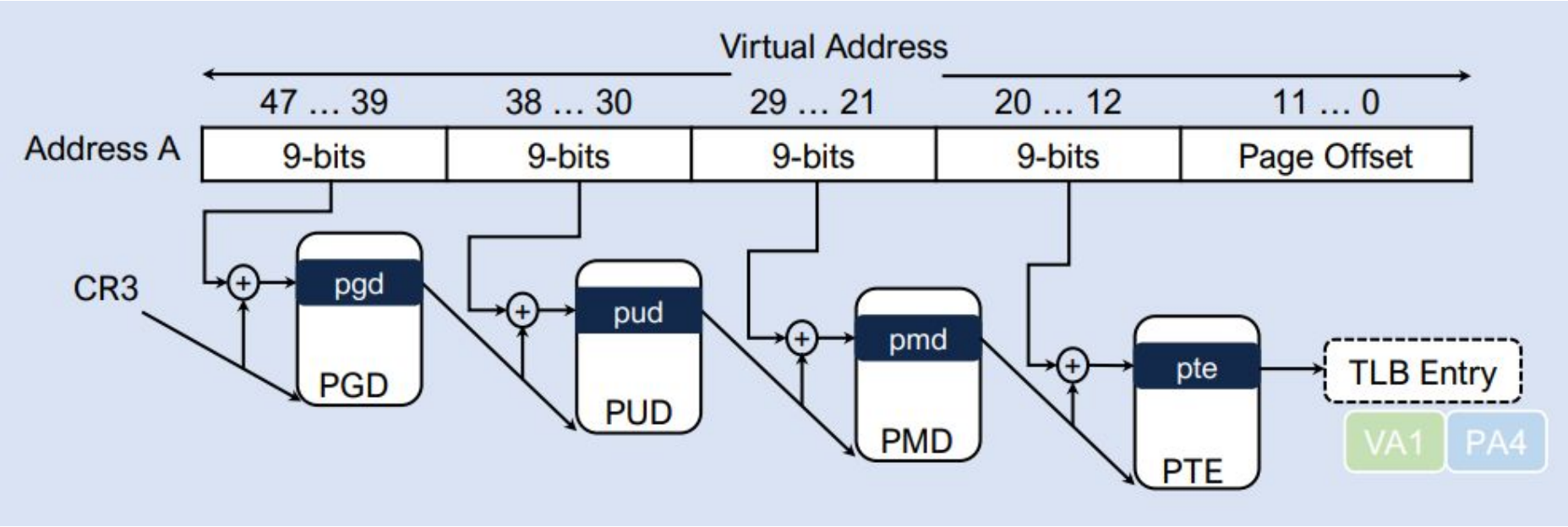
- Address space must be continuous
  - Lots of opportunities for external fragmentation
  - Harder to change size of allocation
- Low granularity of access control
  - Harder to implement protection mechanisms

Paging



- Map fixed size chunks of virtual address space to chunks of physical memory
  - 4KB, 4MB for us, 4KB, 2MB, 1GB for modern CPU
- Reduces external fragmentation
  - Continuous virtual addresses don’t have to be physically continuous
- Translations are per-process
  - PD, PT, cr3
- Page faults

—



Traditional (Radix Style) Paging

Why do we need multiple levels of paging structures?

It is infeasible to store a complete table of page mappings for all running processes.

Here's the Math

Single Level Paging Scheme

# of entries = (4 GB / 4KB) = 2^{20}

table size = (2^{20} entries * 32 b / entry) = 2^{22} B = 4 MB

We need 4MB of memory for every process's minimal paging structure -> that's a lot :(

2 Level Paging Scheme

PD entries = (4KB / 32 B) = 1024

PT entries = (4KB / 32 B) = 1024

Total indexable memory = 1024 * 1024 * 4KB = 2^32 B = 4GB

We need 8KB of memory for every process's minimal paging structure

Traditional (Radix Tree) Paging

- Use of radix trees reduces the overhead associated with indexing smaller amounts of memory
  - This is essential if we would like to run > 1 process
- Address translation becomes a large overhead
  - Multiple memory accesses required to translate one address
  - This is why we cache them!
- Other page table schemes do exist

Example Time!

34-bit system

32-bit physical address space

8 KB, 2 MB, 512 MB pages

8 B addressability

Design a 3 level radix-tree type paging scheme…

** Assume PD, PT1, PT2 have the same number of entries*

1. find MIN required offset bits

   page size / addressability = 8KB / 8B = 2^{10} → 10 bits

2. decide # of entries in paging structs

   34 bits - 10 bits = 24 bits → $\frac{24}{3}$ = 8 bits to index

   ↳ we can have 2^8 entries

   page size / num entries = max entry size

   ↳ 8KB / 2^8 = 32B → this is v. big, we will be fine w/ putting paging structs in the smallest page

NOTE: numbers represent the index of the bit to the left of the line

3.

8KB indexing

offset bits = 2 MB / 8B = 18 bits

2 MB indexing

offset bits = 512 MB / 8B = 26 bits

512 MB indexing

Page Translation Caching

- **TLB** = **T**ranslation **L**ookahead **B**uffer
  - Caches virtual to physical address translations
  - Assuming good spatial locality of data / temporal locality of accesses, significantly reduces overhead
- **MMU** = **M**emory **M**anagement **U**nit
  - Caches partial page translations
  - Probably don't worry about this :)
  - Take 411 if you're interested in this

Consider a virtual memory system with 4 KB pages, and a TLB that has 3 translation entries. The TLB uses a "least recently used" (LRU) replacement policy. A program operates on a 2-dimensional array of 4×2048 32-bit integers, `int matrix[4][2048]`. The array is stored consecutively in memory, starting at virtual address 0x940000. Table 1 shows the layout of the array.

Virtual address	Data
0x940000	matrix[0][0]
0x940004	matrix[0][1]
0x940008	matrix[0][2]
...	...
0x942000	matrix[1][0]
0x942004	matrix[1][1]
...	...

Table 1: matrix array storage in memory

Consider the following code:

```
int x, y;
result = 0;
for (x = 0; x < 2048; x++) {
  for (y = 0; y < 4; y++) {
    result += matrix[0][x]*matrix[y][x];
  }
}
```

int matrix[4][2048] → 4B · 4 · 2048 = 32KB
= 8 pages

↳ matrix[i][j] → 2 pages per i
j ∈ [0, 1024) and j ∈ [1024, 2048)

how many Hb misses?

2 for matrix[0][x]

will miss every time for matrix[y][x] (unless y=0)

↳ 3 · 2048 + 2 = 6146 misses

how do we miss less? flip the loops

Filesystem

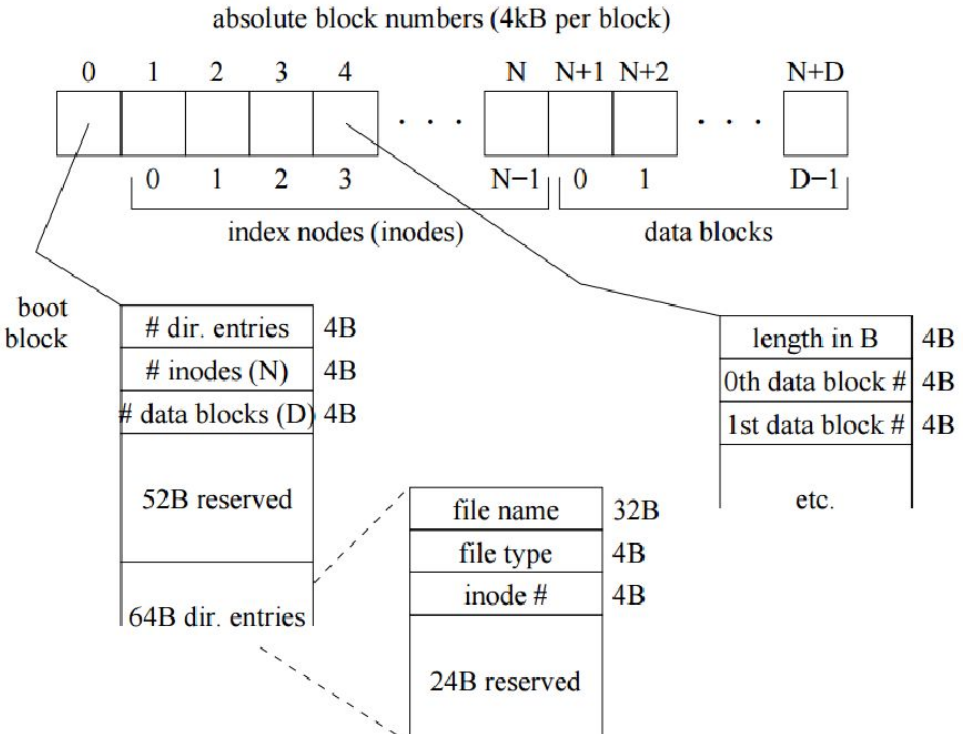
Why we need a filesystem?

- *Fundamentally - we want a way to semantically organize data*
- Ability to store both many small files and fewer large files efficiently
  - What's wrong with storing every file consecutively in memory?
- API between user functions (open, close, read, write) and the device driver functions (limited functionality)

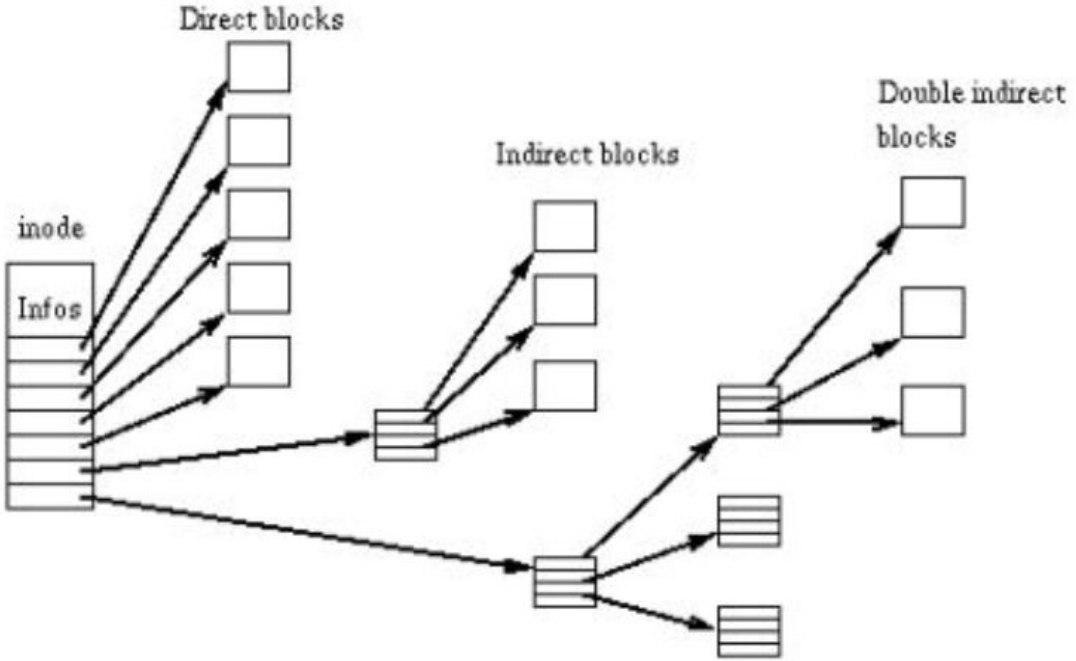
Filesystem Components

- Superblock – represents a specific mounted file system
  - Sometimes called the “boot block” as well
  - Stores a set of *directory entries* (or dentries)
- Inode object – represents a specific file/directory
  - Contains specific information to handle file
- File descriptor - Index to a data structure (at the kernel level) containing the details of all open files
  - You just implemented this in MP3.2 and 3.3!
  - In our case, we use the *PCB* to store these details

MP3 Filesystem



Indirect Blocks



Issues with the MP3 Filesystem

- Fixed inode size
  - Trivial maximum filesize, fixed overhead for each file
- No write access
- No such thing as file hierarchy

There are (many) more limitations to the MP3 filesystem - some obvious, some not.

Example Time!

Modify the MP3 file system so the last data block number in an inode represents an indirect data block. What is the maximum size of a file?

Answer

Modify the MP3 file system so the last data block number in an inode represents an indirect data block. What is the maximum size of a file?

A. 1022 direct data blocks

1024 indirect data blocks

4kB of data per data block

8184 kB of total data

Another Example

Using the MP3 file system structure, how much space would be used to store 10 5 kB files? Include the boot block, inodes, and data blocks.

Answer

Using the MP3 file system structure, how much space would be used to store 10 5 kB files? Include the boot block, inodes, and data blocks.

4 kB for boot block

10 inodes of 4 kB each

20 data blocks of 4 kB each

124 kB total space

Interrupts

What is an interrupt?

Interrupts

- Piece of code that preempts the currently running process for a time sensitive goal
- Not managed by the scheduler
  - Executes w/ priority until finished, Cannot sleep
  - Longer interrupts can cause device stalls
- Many types of interrupts
  - Exceptions
  - Hardware/Device interrupts (PIC)
  - Syscall

How can we avoid stalls during more complex device interrupts?

Device Interrupts (The PIC)

- The PIC is a *physical* thing, devices using it must be *wired* to the IRQs on which they will be serviced
  - For us IRQ 8 = RTC, IRQ 1 = Keyboard
- Interrupt handlers for devices are registered in the IDT
  - Linux allows for the chaining of interrupt handlers
  - Why is this useful? Why is it necessary?

IDT

The **IDT** contains interrupt descriptors, which tell the processor where to go in the event of an interrupt

- Address of function, privilege level, etc.
- There are reserved entries for exceptions and system calls

Trap Gate vs **Interrupt Gate**

- Interrupt gate will *clear the IF*
- Additionally, changes which instruction address is saved

IDT

- PIC interrupts are interrupt gates
- System Calls are trap gates

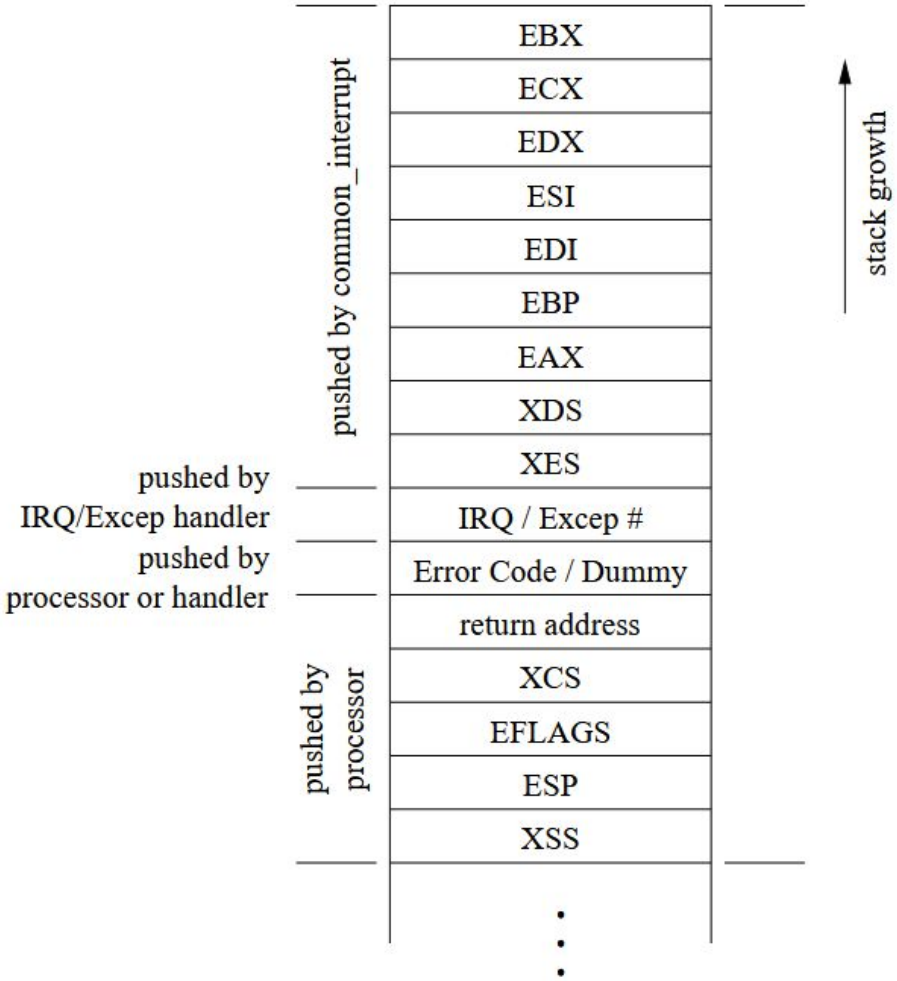
What about exceptions?

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 primary 8259 PIC	0x20	IRQ0 — timer chip	example of possible settings
	0x21	IRQ1 — keyboard	
	0x22	IRQ2 — (cascade to secondary)	
	0x23	IRQ3	
	0x24	IRQ4 — serial port (KGDB)	
	0x25	IRQ5	
	0x26	IRQ6	
	0x27	IRQ7	
0x28–0x2F secondary 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	

Interrupt Control Flow

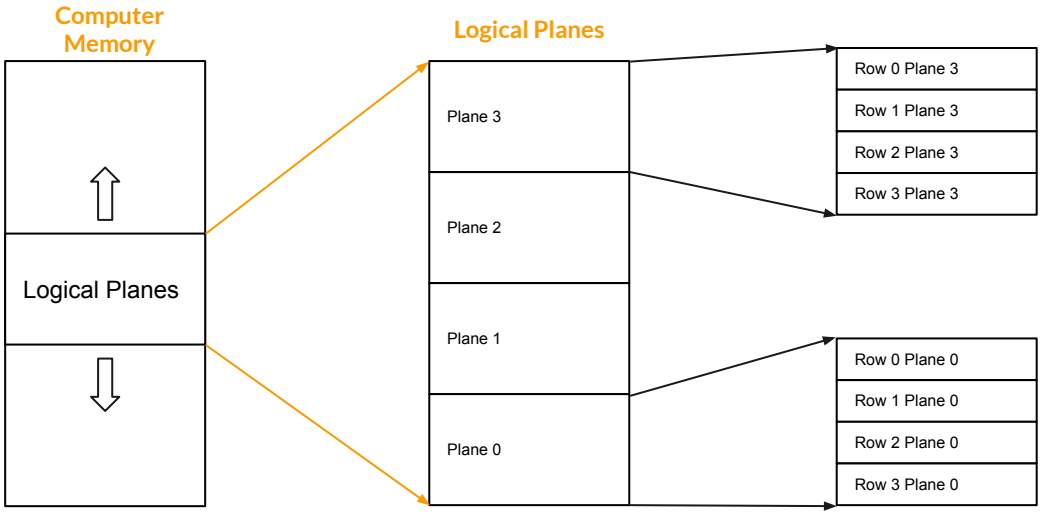
- Running user code . . .
- Receive an interrupt
- Context switch to kernel space using TSS and push user context
- Push bookkeeping information
- Run kernel handler for IRQ 2 based on bookkeeping
- Teardown and return to user context

How does this change if already in kernel mode?



MP2 - Modex and Tux

Modex



- Plane Setup is Still Row Ordered
- Logical Planes useful for managing context of full image
- Build buffer requires excess memory
- LOGICAL PLANES != VGA PLANES
  - Order matters when loading into vga
- Top plane must be 3, and bottom 0
  - Why?

16
20
28
32
15
19
27
31
14
18
26
30
13
17
25
29

Modex

0	1	2	3	0	1	2	3	0	1	2	3
1	2	3	4	5	6	7	8	9	10	11	12
13	14	15	16	17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32	33	34	35	36

16
20
28
32
15
19
27
31
14
18
26
30
~~13~~
17
~~25~~ ➡ 21
29
➡33

Modex

0	1	2	3	0	1	2	3	0	1	2	3
1	2	3	4	5	6	7	8	9	10	11	12
13	14	15	16	17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32	33	34	35	36

- Logical view shifts right
- Plane 0 shifts down one address

16
20
28
32
15
19
27
31
~~14~~
18
~~26~~ ➡ 22
30
~~13~~ ➡ 34
17
21
29
33

Modex

0	1	2	3	0	1	2	3	0	1	2	3
1	2	3	4	5	6	7	8	9	10	11	12
13	14	15	16	17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32	33	34	35	36

- Logical view shifts right
- Plane 1 shifts down one address

➡ 4
➡ 8
16
20
~~28~~ ➡ 3
~~32~~ ➡ 7
15
19
~~27~~
~~31~~ ➡ 6
~~14~~ ➡ 10
18
22
~~30~~ ➡ 5
~~34~~ ➡ 9
17
21
~~29~~
~~33~~

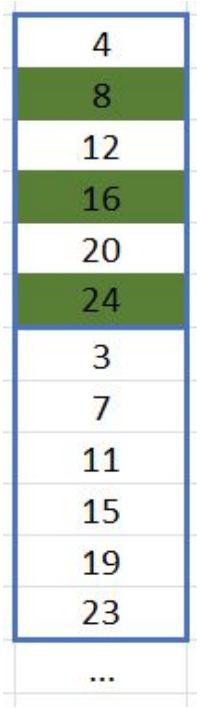
Modex

0	1	2	3	0	1	2	3	0	1	2	3
1	2	3	4	5	6	7	8	9	10	11	12
13	14	15	16	17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32	33	34	35	36

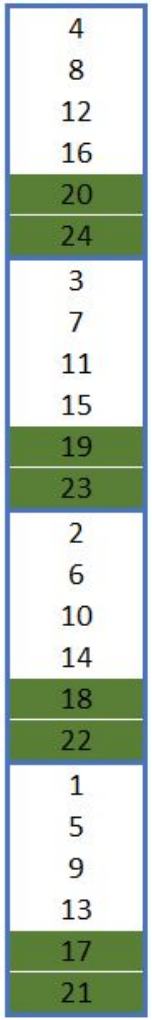
- Logical view shifts up
- All planes shift up 2 addresses

Modex (Draw_vert_line)

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24



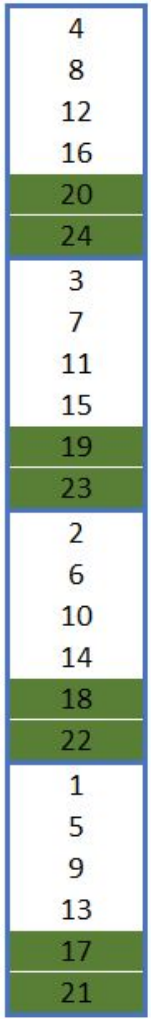
- All new pixels in same plane
- Each pixel isn't sequential in memory



Modex (Draw_horiz_line)

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24

- Each pixel is placed into new plane
- Each subsequent pixel is located in the next memory address



Modex (Draw_horiz_line)

```
for (i = 0; i < SCROLL_Y_DIM; i++) {
        addr[p_off * SCROLL_SIZE] = buf[i];
        if (--p_off < 0) {
                p_off = 3;
                addr++;
        }
}

SCROLL_Y_DIM = 3
SCROLL_X_WIDTH = 8 / 4
SCROLL_SIZE = SCROLL_X_WIDTH * SCROLL_Y_DIM
```

Structure of Tux

- Driver: Middle man between User and Device
- Tux as a State Machine
  - Send commands to change the state
  - Tux interrupts us when the state changes
    - WE DO NOT ASK TUX ABOUT THE STATE (This is polling, it is bad)

Octrees

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 (ex 2): less combination of colors (2^6), but worse quality
- Higher level (ex 4): more combinations, better quality

Hybrid of Levels:

- Some high quality colors
- Default low quality colors
- How do we choose high quality? -> Sort by frequency
- Extra accuracy? -> average color counts

Ex of avg: R(11000), R(11010), R(11001) all in level 2 R(11000), but the average is 11001

TA Exam Hints

- Powers of 2
  - 1 KB = 2^{10} B, 1 MB = 2^{20} B, 1 GB = 2^{30} B
- Internal vs external fragmentation
- Modex logical view to plane address translation
  - Can you write a for loop to do this
  - Understand build buffer, why it bubbles, etc.
- Know MP3
  - Can you explain *all* code your group has written so far
  - Understand *why* you made the decisions you did in MP3
- At a high level, how do “traditional” implementations work?
  - Acts as a comparative tool, can provide examples for SAQs
- Filesystem inodes contain indices into data block array

Questions?