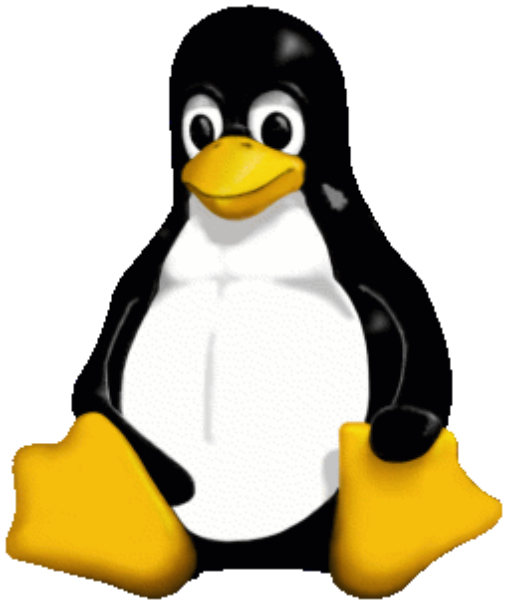
Exam 2 Review Spring 2022

Tejus Kurdukar, Alan Liu, Sahan Kumarasinghe, Noelle Crawford, Asher Mai

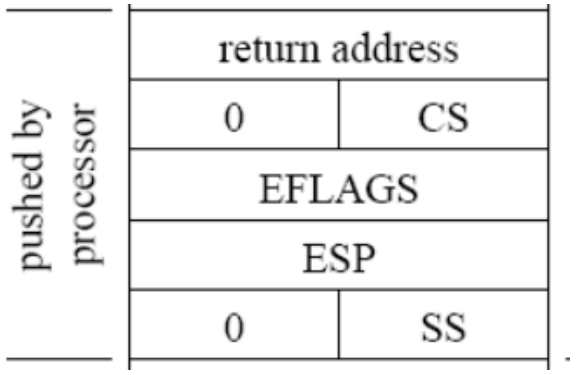


General Tips

- Meet with your group; Code review!
  - Check implementations that your teammates did, especially those that you are not familiar with
  - Check your bug log!
- Re-read MP2. If you did not implement some part of MP2, try to get help to understand the principles
  - Re-watch the discussions
- When going through past exams, generalize your solutions. Be prepared for problems with different parameters (e.g paging schema, filesys)
- On memorizing numbers: there's no shame in putting powers of 2/common hex values for 1KB/4KB/1MB on your cheatsheet ^_^

Interrupts

- Interrupt Descriptor Table
  - 256 entries; 0x80 is the syscall; first 32 are exceptions
  - Interrupt gates vs Trap gates – the IF flag
  - DPL in descriptors
  - Each entry points to assembly linkage – wrappers around interrupt handlers
- IRET vs RET?
  - IRET: Return from interrupt
  - IRET performs a Stack switch, privilege switch
    - Pops 5 things



Example Questions

- How do you install interrupt handlers? Who does it?
  - The device driver calls request_irq to install a handler
  - Some stuff to review (just the basics!): request_irq, setup_irq, do_irq
- How does the kernel invoke the interrupt handler?
  - ASSEMBLY LINKAGE in IDT entries, around do_irq
    - Save and restore registers and iret, in order to maintain C calling convention and return from interrupt
  - Remember, PIC sends vector in IDT upon receiving interrupt, masks equal and lower prio. IR lines until send_eoi

See lectures 12, 13 for completeness

Other stuff to review

- Interrupt Chaining
  - Use linked list of handlers, to be able to process multiple handlers for one IRQ vector
    - When installing a new device on an irq line, all devices on that irq line must agree to interrupt chaining! Not all devices can be queried.
    - For multiple handlers on one device, they should not confuse the device
- Software Interrupts (tasklets)
  - Runs at a priority between program and hardware interrupts; used for actions that don't involve the device
    - HW interrupts are supposed to be really fast! Wasteful to do these in hardware interrupts.
    - When HW interrupt ends, check for pending software interrupts

MP3.1 IDT – Example Questions

- Suppose you are implementing the rtc handler. Your keyboard handler works flawlessly! But for some reason, your rtc can't get more than one interrupt off... what happened?

A: Forgot to send one or both EOIs: one on the primary PIC for IR2, one on the secondary PIC for RTC!

- Why do we need assembly linkage?

A: Need to create a wrapper to save and restore registers and iret, in order to maintain C calling convention and return from interrupt properly

- Suppose a user program wants to open a file. This requires privilege, so the user program invokes the sys_open system call. The CPU raises a general protection fault, even though sys_open is implemented perfectly – what happened??

A: Syscall DPL level was not set to 3 in IDT! The user program does not have the privilege to use the system call.

- Ben Bitdiddle wants to use trap gates instead of interrupt gates for your team's IDT implementation. Explain what can go wrong.

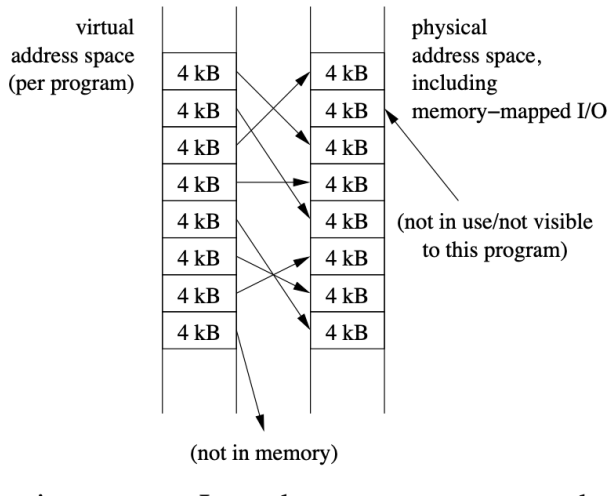
A:Trap gates do not clear IF. Interrupts may preempt each other, which can considerably delay an interrupt's execution and take more time to return to the user program

Virtual Memory

Insertion of level indirection between the memory address space seen by programs and the physical address space in the operating system

Perks/Trade-Offs:

- Protection
- Sharing
- Fragmentation
- Relocation

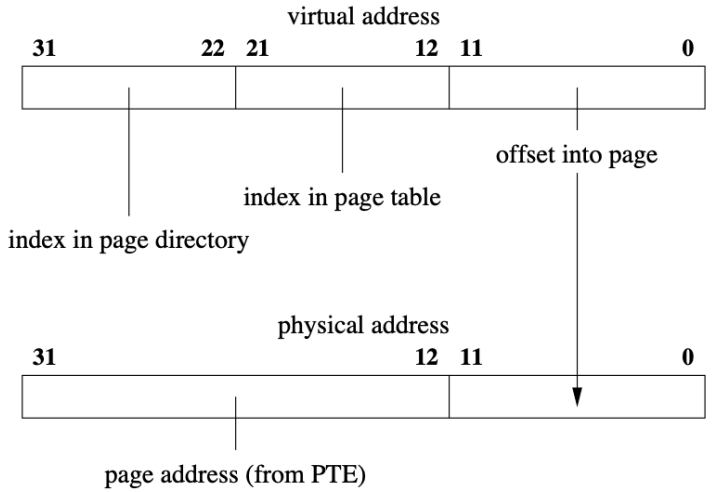


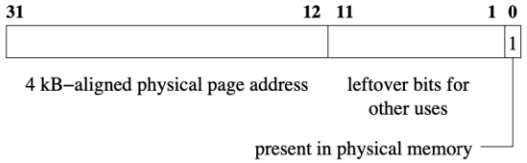
Segmentation

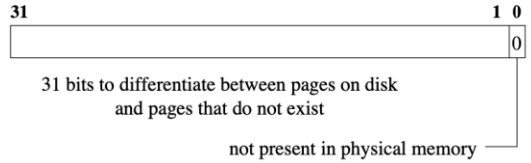
- No longer utilized by Linux
  - Bonus question: *why?*
- Cannot be turned off by x86 based processors, so segments are simplified to the point of being meaningless
- Segments described in one of two arrays called descriptor tables
  - Global (GDT)
  - Local (LDT)

Paging

- Virtualized memory
- Address split up into sections which each represent an offset
- Translation Lookaside Buffer (TLB)
  - Caching mechanism to store translation for pages
  - Located in Memory Management Unit
- Page Directory Base Register (CR3)
  - Pointer to page of base of page directory
  - Each write to it flushes TLB
- Page Fault Register (CR2)
  - Not only for debugging purposes – swap pages to disk
- Paging Functionality
  - Page Directory - Holds pointers to page tables or 4 MB pages
  - Page Table – Contains pointers to 4 kB pages
  - Each of table contains a 32 bit value where significant 20 are reserved for address and the remainder are leftovers used for protection and performances:
    - User/Supervisor
    - R/W
    - Global Flag
    - Page Size
- No, virtual addresses and physical addresses don't need to use the same # of bits







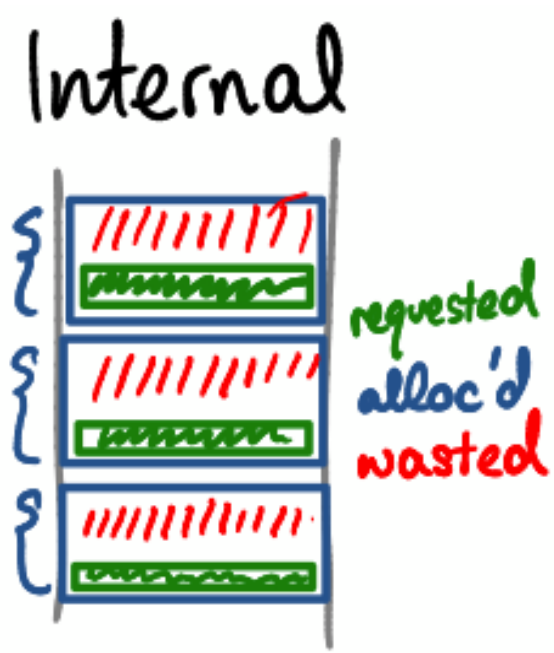
Paging Design Tradeoffs

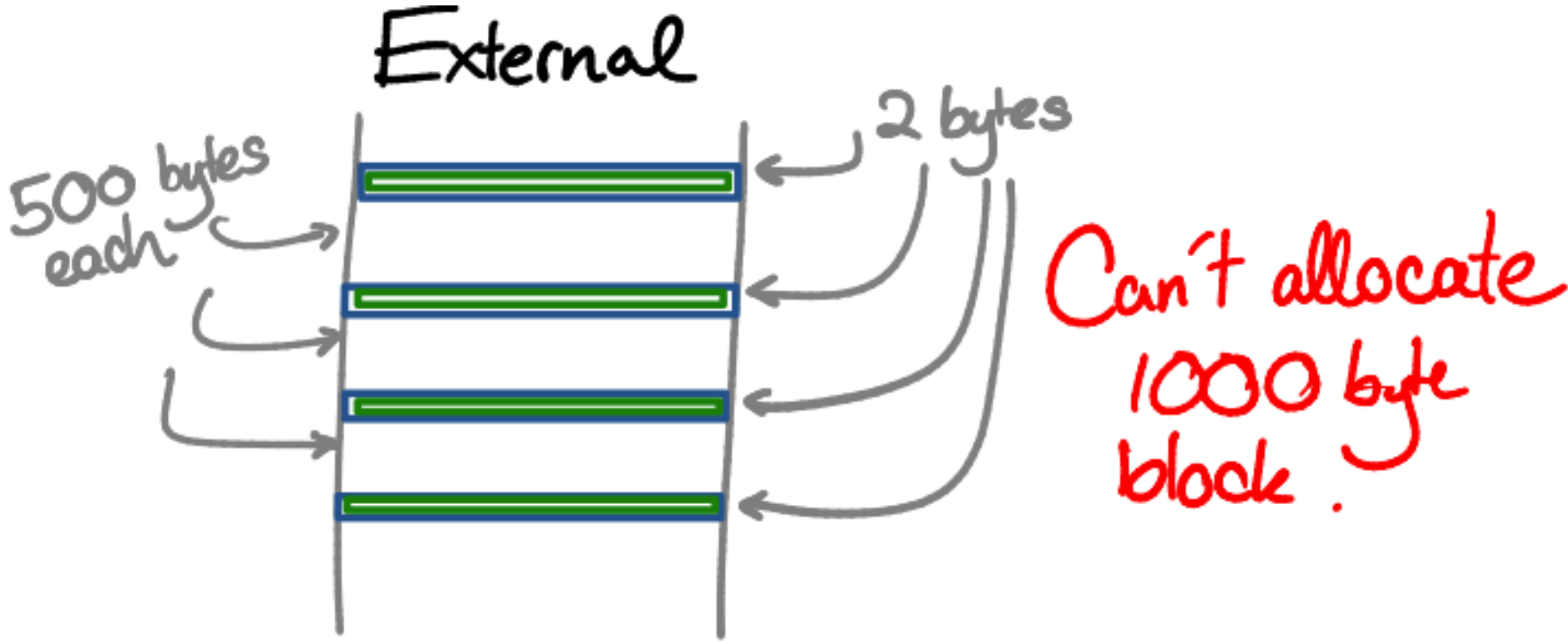
- Large Page Size
  - Lower Miss Rate on TLB
    - Fewer choices of things to cache
  - More risk for internal fragmentation
  - Smaller paging structures for same depth of hierarchy (fewer table entries)
- Small Page Size
  - Higher Miss Rate on TLB
  - Less Risk of internal Fragmentation
  - Larger paging structures for same depth of hierarchy (more table entries)

Internal Fragmentation: Waste of space within allocate page

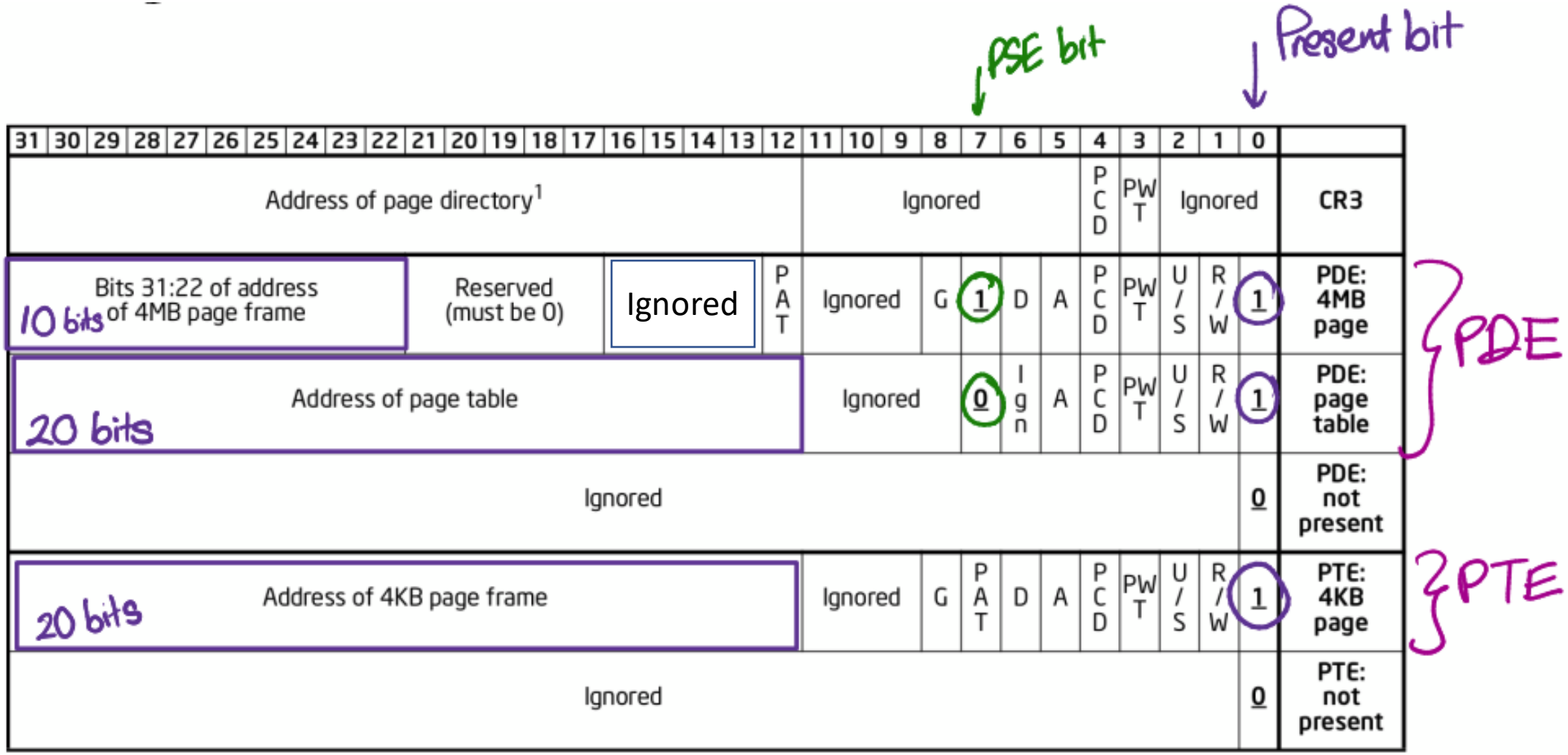
External Fragmentation: Uneven allocation of pages, specifically with allocating contiguous memory for a page.

Internal vs External Fragmentation Diagram





Descriptors used in x86's Paging



Paging in C terms, maybe

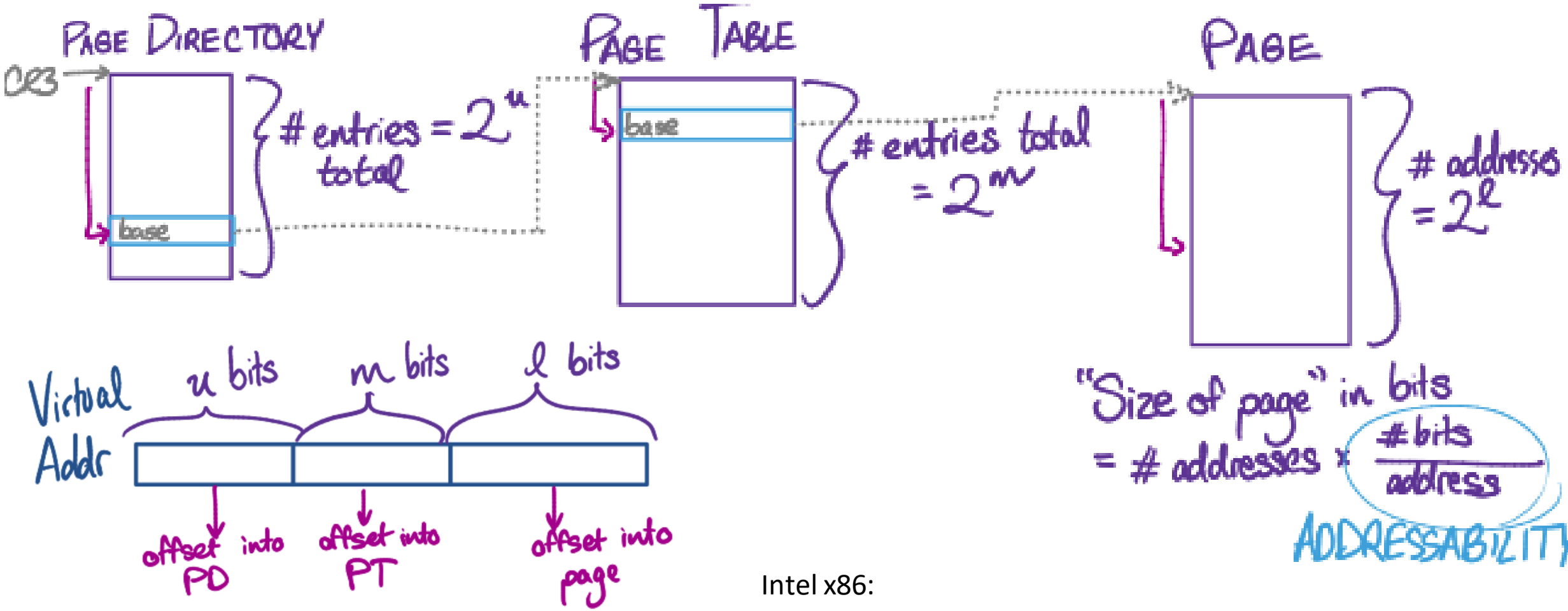
- Assume our x86 intel architecture, where the linear address is of form:
  - [off_PD = 10 bits, off_PT = 10 bits, off_PG = 12 bits]
  - Example, in the linear address [0x12300ABC]
    - off_PD = 0x048 (00/0100/1000)
    - off_PT = 0x300 (11/0000/0000)
    - off_PG = 0xABC
  - And the page directory is a global array of structs, `pd`
- Then this will obtains the physical address of a 4KB page:

pt = (pd[off_PD].base << 12)

real = (pt[off_PT].base << 12) + off_PG

* Insight: the "bit shifting" is contingent on the # of bits of "base" in the descriptor and the # of bits that make up a physical address

Paging Illustration: 3 Levels



Intel x86:

- PSE Not Set: U=10, M=10, L=12

Paging Sanity Check

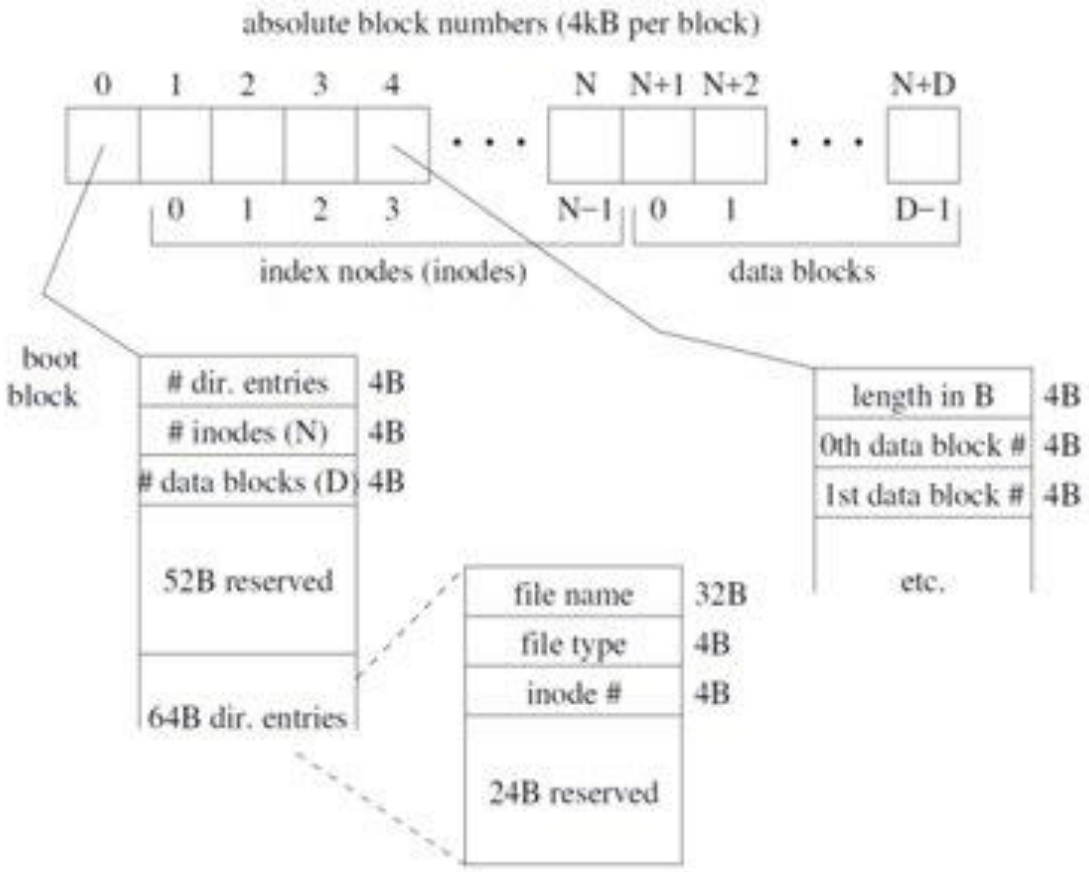
- In this question, assume the form of the virtual addresses is
  [<U bits>][<M bits>][<L bits>]
  , where U bits are used to index into the page directory, M bits are used to index into the page table, and L bits are used to index into the page.
  - If somebody tells you that in their architecture,
    - PD entries and PT entries have the same size each
    - Their page tables have *more* entries than the page directories
  - Which of the below relations is *necessarily* true? (Select all that apply)
    - U<M
    - U>M
    - M<L
    - M>L

  •If the PSE bit was set in such an architecture, how many bits would offset into the page, in terms of {U, M, L}?

  •In another scenario, if U=4 and M=7, and PDEs and PTEs have the same size, is the PD bigger, or is the PT bigger? How many times bigger?

File System

- Unix File System (ext2 filesystem)
  - A single file system is held within one file image
  - Consists of 3 components: superblock, inodes, data blocks
  - Linux considers everything a file
    - Devices (RTC, keyboard)
    - Directories

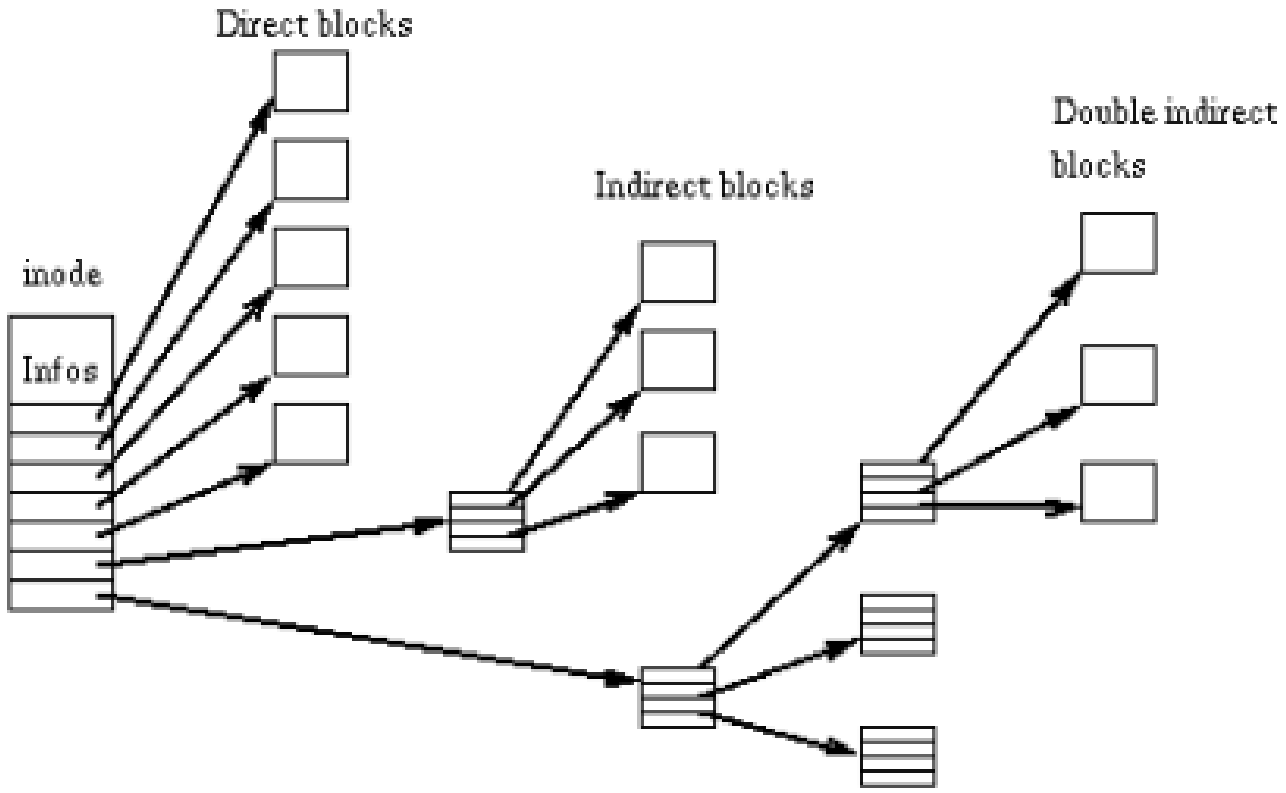


File System (cont.)

- Superblock
  - Block of data at the start of the file system that is given to the OS upon boot
  - Contains data entry blocks (dentries)
  - Contains information on free blocks and inodes
  - Many backups of superblock contained in filesystem
- Inode
  - Associates data blocks with a given file
    - For especially long files, indirection can be used to increase max file size (see next slide)
  - Contains important metadata about the file
    - Not filename
  - Indexed from 1 to n for n-1 total inodes

File System (cont.)

- Data Block
  - Contains the actual data
  - For MP3, only direct blocks are used



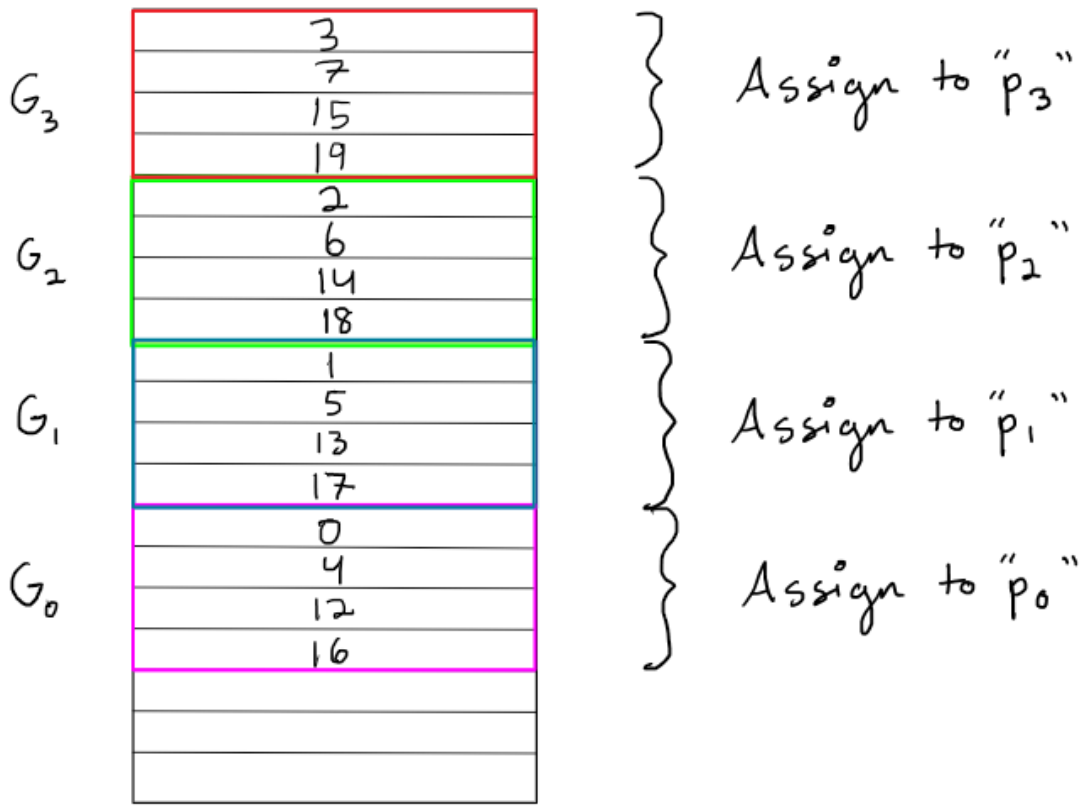
MP2.1

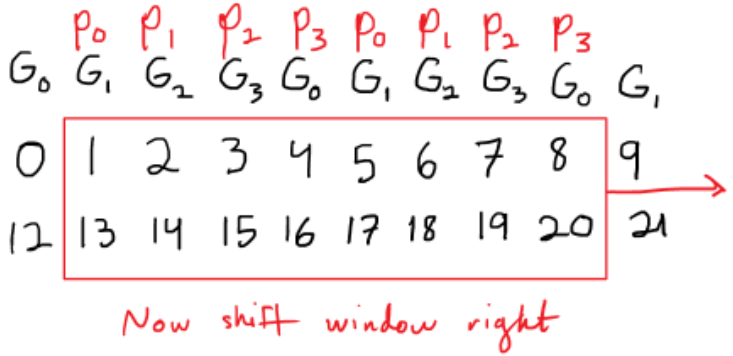
- Understand VGA modeX
  - Each video memory address refers to 4 pixels (in planes 0, 1, 2, and 3)
- Be able to do math to calculate the address (offset) + plane of a pixel coordinate
- Be able to explain the build buffer
  - What is the advantage of using one?
    - Less memory accesses/changes when scrolling
  - Why do we use 3, 2, 1, 0 plane order?
    - Removes need for space between each plane's section of the buffer
  - Why does the status bar not have a build buffer?
    - The status bar doesn't scroll so there is no need!
- Examples on next slides: shifting the logical window

P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3		
G_0	G_1	G_2	G_3	G_0	G_1	G_2	G_3	G_0	G_1
0	1	2	3	4	5	6	7	8	9
12	13	14	15	16	17	18	19	20	21

Call G_i "Group i"
↳ Constant, assigned to screen
Call P_i "Plane i"
↳ Relative to VGA

Build Buffer





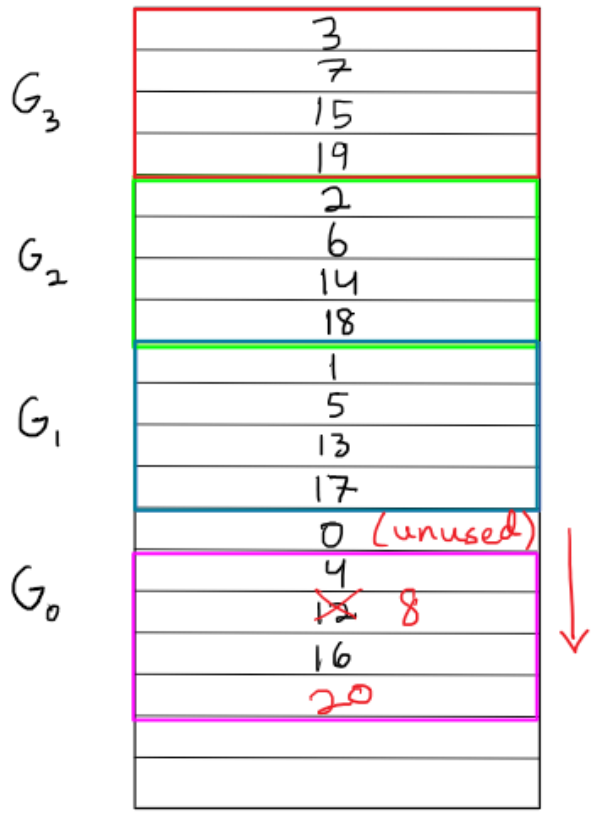
Call G_i "Group i"

↳ Constant, assigned to screen

Call p_i "Plane i"

↳ Relative to VGA

Build Buffer



Assign to "P_2"

Assign to "P_1"

Assign to "P_0"

Assign to "P_3"

p_i cyclically shifts

* Don't need to change every pixel!

Just slide window and change some pixels

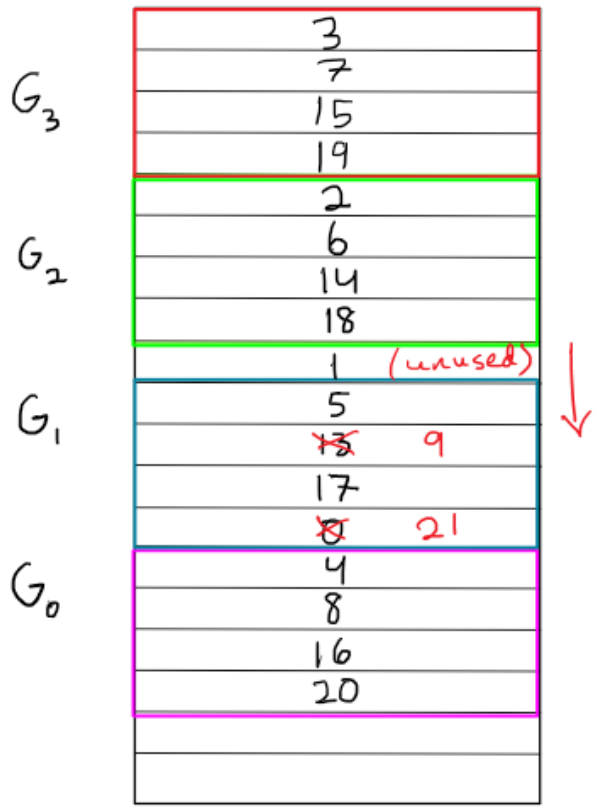
		P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
G_0	G_1	G_2	G_3	G_0	G_1	G_2	G_3	G_0	G_1
0	1	2	3	4	5	6	7	8	9
12	13	14	15	16	17	18	19	20	21

One more shift

Call G_i "Group i"
↳ Constant, assigned to screen

Call P_i "Plane i"
↳ Relative to VGA

Build Buffer



Assign to "P_1"

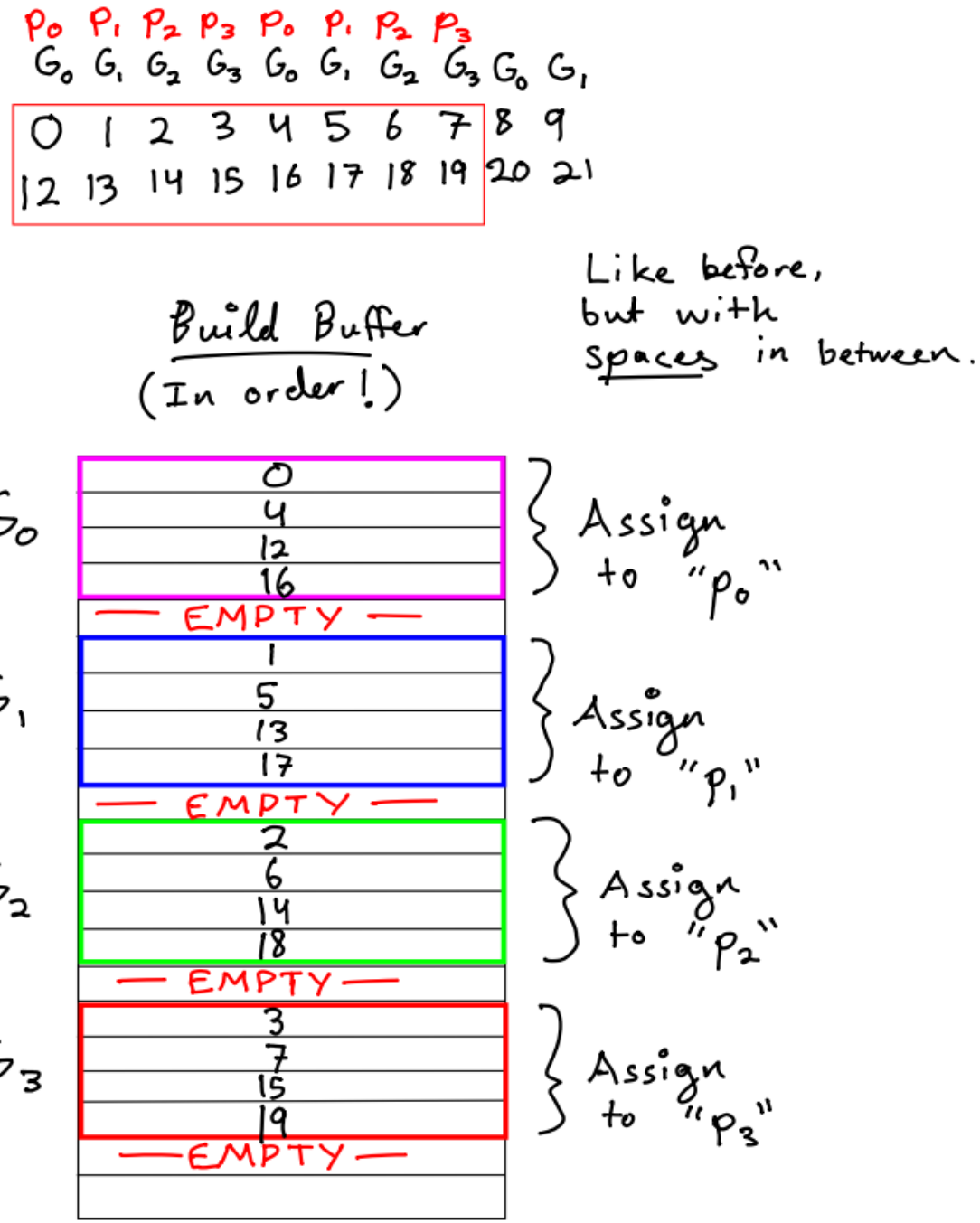
Assign to "P_0"

Assign to "P_3"

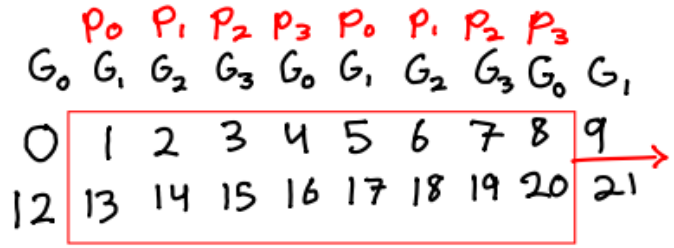
Assign to "P_2"

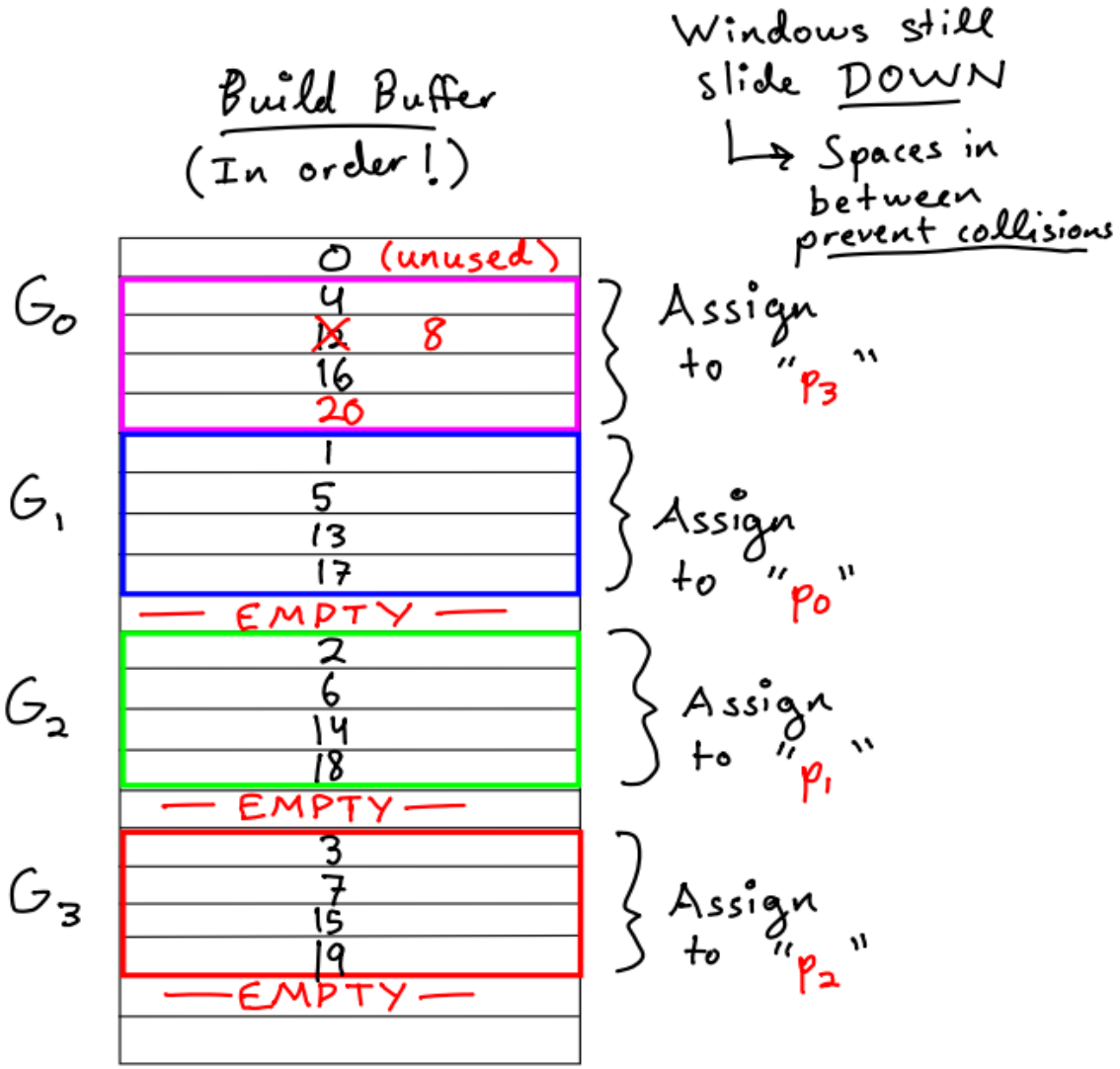
P_i cyclically shifts

* Don't need to change every pixel!
Just slide window and change some pixels



Q: What if you didn't want to reverse the order?
A: Fine, but make sure you have an empty space in between build buffer windows!





Q: What if you didn't want to reverse the order?
A: Fine, but make sure you have an empty space in between build buffer windows!

MP2.2 Tux

- Know the types of questions we asked in demos!
- How did you prevent LED spamming?

A: Using the ACK signals and a flag

- Why did we use an interrupt-driven approach between the driver and the device but a polling-based approach between the user program and the driver?

A: Communication with the device was very expensive – it takes too much time to poll the device, and the device may freeze. Spending too much time in an interrupt context is unacceptable.

- Why did we implement Tux controls in a separate thread?

A: Allows for simultaneous input with keyboard and tux controller without explicit preference

- Why did we sleep the tux thread?

A: We don't want it to waste resources when no buttons are pressed

Practice Problems

Q1: Paging ..... 18 points

The Physical Address Extension (PAE) mode of x86 uses 3-level page entries to be able to address more than 4 GB of physical memory, while keeping a 32-bit virtual address space. It is your job to create a page table walking functions for this page table system that translate a 32-bit virtual address into a 64-bit physical address.

(a) (4 points) From the documentation of PAE, 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 x86 2-level page table system).
- The total size of the Page Directory is 128 bytes.
- The sizes of Page Table 1 and Page Table 2 are the same.
- The physical pages are 4KB in size, just as in x86.
- Page Table and Directory entries are 64 bits long.

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

PD = _____ bits
PT1 = _____ bits
PT2 = _____ bits
Offset = _____ bits

(b) (9 points) Next you found the important information on how each of the entries are stored:

- The Present bit is the **most** significant bit of each of the entries. If the Present bit is 1, then the entry is valid. Otherwise the entry is invalid.
- The address of the next level of the page tables is stored in the **lower** bits of the entry.
- The other bits are reserved. You should not change or access them.
- Each Page Directory, Page Table, and Page are aligned to its own size.
- The `void *` pointers are 64-bit physical addresses. You may want to cast them to the `uint64_t` unsigned 64-bit integer data type; gcc allows the full range of arithmetic, logical, and shift operations to operate on this type.

Write the helper functions below to move from one level of the page tables to the next.

```
/* Takes in the base address of Page Directory and
 * virtual address as the argument.
 * Returns the base address of Page Table 1 if valid.
 * If invalid, returns NULL */
void* PD_to_PT1 (void * PD_addr, uint32_t v_addr) {

}
```

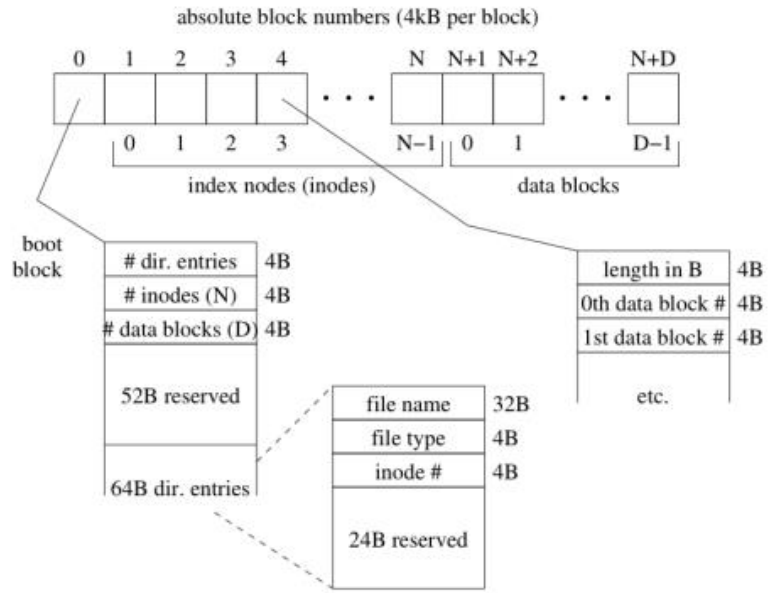

(c) (3 points) Write the full page table walking function to translate a virtual address into its corresponding physical address.

```
void* PD_to_PT1 (void* PD_addr, uint32_t v_addr);
void* PT1_to_PT2 (void* PT1_addr, uint32_t v_addr);
void* PT2_to_Page (void* PT2_addr, uint32_t v_addr);

void* get_Physical_Address (void* PDBR, uint32_t v_addr) {

}
```

(d) (2 points) What's the purpose of having more than 4 GB of physical memory if only 4 GB of virtual memory can be addressed?

Question 1: File System Fairy Tale . . . . . 16 points

In a land far away a file system stood. One from MP3, something to expect, you should. It was a magical place with wonder infused, yet there was one man who was a tad bit confused. He was taking a class, the workload a ton. This class, indeed, was 391. The file system had inodes, files, and even a byte, which the brilliant student just could not get right. He struggled for awhile, he was not a fan. The confusion continued until he developed a plan. He remembered his family, forever matriarchal, whose mother just happened to be the pony Twilight Sparkle. She had some connections to some students who knew, everything that the student would have to do. The student was pleased and thought it was great, anxiously hoping it wasn't too late. The information finally landed him here, where he requires you students to displace all his fear. If you finish his quest and finish it well, you will be rewarded with points and all will be swell. So grab your quill, pencil, or pen, and finish every task, requested by Ben.



Notes:

1KB = 1024 B

You may leave expressions in unsimplified form.

Assume only regular files and directories (e.g. no devices).

(a) (2 points) What is the maximum number of files that this file system can support?

(b) (2 points) What is the maximum size of any single file in this file system?

(c) (5 points) Assume that your file system contains the maximum number of files and that each of those files is 4B. What fraction of the total filesystem size is used for actual data?

Question 1 continues... ECE 391, Exam 2 Monday, November 10, 2014

(d) (2 points) Why do the reserved bytes exist in the boot block as well as in each of the directory entries?

(e) (5 points) Describe the process of reading a file from this file system step-by-step

(f) (5 points (bonus)) Implementing write on this filesystem would be inefficient. Explain why and suggest an additional file system data structure that would help.

Filesystem Question

Grace decided to use her filesystem to store her ECE391 notes so that she could brag about it to the recruiter. **Grace always begin her notes by noting down the lecture number in the format "Lecture #N", and they're always long enough to fill > 1 data block**. Her boyfriend David, in an attempt of April Fools prank, corrupted all the directory entries – **in both the super directory block and the subdirectory nodes -** with garbage values in Grace's file system. Desperate to study about file systems for midterm 2, Grace asks for your help to recover her notes for Lecture #16.

*Is it possible to **partially** recover Grace's notes for lecture 16?*

*Is it possible to **fully** recover Grace's notes for lecture 16?*

Problem 5 (10 points): MP2 - ModeX

In MP2, you drew an image into the build buffer, then copied the entire screen area from the build buffer to the io-mapped video memory. This only worked for the MP due to some subtleties with using a VM. In this question, you will be taking advantage of the plane system of ModeX in order to draw a solid shape directly to video memory. Implement the `draw_horizontal_line` function below using the following information:

- `(x, y)` is the leftmost coordinate of the line.
- `length` is the length of the line in pixels.
- Assume the line fits within the valid screen area.
- `color` is the palette index for the fill color of the line.
- `SET_MASK(x)` will set the VGA plane mask based on the low 4 bits of x.
  i.e. 0001 sets the VGA to write to plane 0, 1010 sets the VGA to write to plane 3 and 1

```
#define SCREEN_X_DIM        320
#define SCREEN_Y_DIM        200
#define SCROLL_X_WIDTH  (SCREEN_DIM_X / 4);
static char *const mem_image; //points to the start of video memory

void draw_horizontal_line(int x, int y, int length, char color)
{
    int i;
    int end_x;
    int start_plane, end_plane;
    int start_addr, end_addr;
    if(length <= 0) {
        __________;
    }
    end_x = x + length - 1; //x coordinate of the last pixel

    start_plane = __________; //plane for first pixel

    end_plane = _____________; //plane for last pixel

    start_addr = x / _____ + SCROLL_X_WIDTH * y; //address of first pixel

    end_addr = end_x / _____ + SCROLL_X_WIDTH * y; //address of last pixel

    if(start_addr == end_addr) { //draw line which doesn't cross addresses
        SET_MASK((0xf<<start_plane) & (0xf>>(3 - end_plane))); //set plane mask

        mem_image[start_addr] = ______;
    } else {
        SET_MASK(0xf<<start_plane); //set mask for first address

        mem_image[start_addr] = ______;

        SET_MASK(_____); //set mask for addresses between start and end
        for(i = start_addr + 1; i < end_addr; i++) {

            ________________________;
        }
        SET_MASK(0xf>>(3 - end_plane)); //set mask for last address

        mem_image[end_addr] = ______;
    }
}
```

```
#define SCREEN_X_DIM        320
#define SCREEN_Y_DIM        200
#define SCROLL_X_WIDTH  (SCREEN_DIM_X / 4);
static char *const mem_image; //points to the start of video memory

void draw_horizontal_line(int x, int y, int length, char color)
{
    int i;
    int end_x;
    int start_plane, end_plane;
    int start_addr, end_addr;
    if(length <= 0) {
1.      return ;
    }
    end_x = x + length - 1; //x coordinate of the last pixel

2.  start_plane = X % 4 ; //plane for first pixel

3.  end_plane = end_x % 4 ; //plane for last pixel

4.  start_addr = x / 4 + SCROLL_X_WIDTH * y; //address of first pixel
5.  end_addr = end_x / 4 + SCROLL_X_WIDTH * y; //address of last pixel

    if(start_addr == end_addr) { //draw line which doesn't cross addresses
        SET_MASK((0xf<<start_plane) & (0xf>>(3 - end_plane))); //set plane mask

6.      mem_image[start_addr] = color;
    } else {
        SET_MASK(0xf<<start_plane); //set mask for first address

7.      mem_image[start_addr] = color;
8.      SET_MASK(0xF); //set mask for addresses between start and end
        for(i = start_addr + 1; i < end_addr; i++) {
9.          mem_image[i] = color ;
        }
        SET_MASK(0xf>>(3 - end_plane)); //set mask for last address

10.     mem_image[end_addr] = color;
    }
}
```

1. Validation…

2-3. Planes in modex are always between 0 and 3

4-5. Remember this?
`img3 + show_x >> 2 + show_y * SCROLL_X_WIDTH.`
Same idea, but using mem_image instead of img3

Q: What does "draw line which doesn't cross addresses" mean?
A: Recall one address contains four pixels. "Not crossing addresses" means the entire horizontal line is contained in those 4 pixels, i.e. one virtual address.

6. Sets the mask to be one or more of the pixels in the single address that we care about, and draws color

7-10. First, set mask for starting address. Draw just the pixels we care about. Then, for each addresses between start and end, draw ALL 4 PIXELS. Finally, draw just the pixels we care about in the end address.

Problem 2 (10 points):

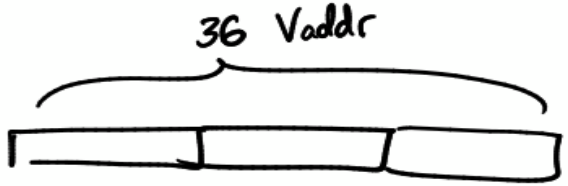
It is fairly common for FPGA's to have larger memory storage standards than an x86-style 32-bit addressable memory. One of your friends has designed a Memory-Management Unit for an x86-style architecture (byte-addressable) to deal with translating a 36bit virtual address space into a 32bit physical address space. There is an x86_32-style 2-level paging system (Page Directory → Page Table → Page) and each entry is 32 bits. The fields in the virtual address are split evenly for use in indexing into the page directory, the page table, and the page itself.

Part A (2 points): How many entries are in the PD? in the PT?

Part B (2 points): What is the biggest possible page size? (Assume the Page Size Extension bit has been set)

Part C (2 points): If only 10 bits are used for options/modifiers in the PTE, what is the smallest possible page granularity supported by a PTE?

Part A:



"Split evenly" → $\frac{36}{3} = 12$ bits for PD, PT, Page offsets each.

Max # of entries in PD = Max # of offsets = 2^{12}

Max # of entries in PT = " = 2^{12}

Part B:

Biggest page size: PSE Bit makes the "PT offset" part of page offset.

36 - 12 = 24 bits for offset.

Max page size ≡ max # of addresses in page = max # offsets into page = 2^{24}

$= 2^{20} 2^4 =$ 8MB

Part C: If 10 bits used for options in PTE, then # bits for address = # entry bits − # option bits = 32 − 10 = 22 bits of address.

Bit shift [# bits of physical − # bits of address field = 32 − 22] = 10 bits for valid address.

$2^{10} =$ 1KB Granularity

Part C (2 points): If only 10 bits are used for options/modifiers in the PTE, what is the smallest possible page granularity supported by a PTE?

Part D (2 points): Why is using the smallest granularity possible in the PTE a bad idea for this paging scheme?

Part E (2 points): Why would recycling this MMU scheme for a 32bit virtual address space (and zero-extending by 4 bits) be a bad idea?

Part D:

Ask: How many bits are used to offset into the page by VADDR definition?

↳ A: 12

↳ How many offsets? $2^{12} = 4K$

Ask: How many bits are used to offset into page with smallest gran?

↳ A: 10

↳ # offsets? $2^{10} = 1K$.

$\frac{1K \text{ used by PT}}{4K \text{ dedicated by VADDR}}$ = 25% of addresses are reachable.

What a waste.

Reason 1: Wasted addresses

2: Offset might go into another page.

(If we have time)

Q5: TLB(caching) ..... 18 points

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];
  }
}
```

(a) (5 points) Show the state of the TLB after each iteration of the inner loop, i.e., after the `result += ...` instruction has been executed. For each TLB translation entry, write down the virtual address of the corresponding page. Assume that the TLB starts out empty, and that the variables x, y, and result are stored in registers and thus do not require memory accesses; likewise, instruction fetches do not use the TLB. The table captures the first six iterations, with the first column already filled in for you.

	round 1 (x=0, y=0)	round 2 (x=0, y=1)	round 3 (x=0, y=2)	round 4 (x=0, y=3)	round 5 (x=1, y=0)	round 6 (x=1, y=1)
TLB entry 1	0x940000					
TLB entry 2	empty					
TLB entry 3	empty					

TLB entry	Round 1 (x=0,y=0)	Round 2 (x=0,y=1)	Round 3 (x=0,y=2)	Round 4 (x=0,y=3)	Round 5 (x=1,y=0)	Round 6 (x=1,y=1)
1	0x940000	0x940000	0x940000	0x940000	0x940000	0x940000
2	empty	0x942000	0x942000	0x946000	0x946000	0x946000
3	empty	empty	0x944000	0x944000	0x944000	0x942000

M m m m h m

(b) (3 points) What is the number of TLB misses that will occur during the execution of the entire program (i.e., 4×2048 rounds).?

(c) (5 points) Rewrite this code to perform the same calculation with fewer TLB misses

Pattern after the first four rounds is hit-miss-miss-miss. Therefore, number of misses in 4*2048 rounds is 4 + 3*2047

You get far less misses (only 8 total) if you simply switch the inner and outer loop, because changes in y are what lead to misses.

QUESTIONS?

Rate us on ~~Yelp~~ Google Forms!