# ECE 391 Exam 2, Fall 2015

Q1: Paging . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 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 function for this page table system that translates 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= 4 dir size/size of entry= 16 -> 2^4=16

PT1=8 Pd+PT1+PT2+offset = 32 & PT1 and PT2 are the same

PT2 = 8 Pd+PT1+PT2+offset = 32 & PT1 and PT2 are the same

Offset = 12 physical page size-> 2^12> 4096B

b)

/\* 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) {

uint32\_t dir\_index, dentry,valid,address;

if (PD\_addr == NULL || v\_addr == 0)

return NULL;

dir\_index = v\_addr >> 28; //Want last 4-bits

dentry = \*(PD\_addr + dir\_index);

// 15 0’s (64-bit)

valid=dentry&0x8000000000000000;

if (valid==0)

return NULL;

address = (dentry & 0x00FF FFFF FFFF FFFF) << 8;

return address;

}

/\* Takes in the base address of Page Table 1 and

\* virtual address as the argument.

\* Returns the base address of Page Table 2 if valid.

\* If invalid, returns NULL \*/

void\* PT1\_to\_PT2 (void\* PT1\_addr, uint32\_t v\_addr) {

uint32\_t pte1, pte\_index;

if (PT1\_addr == NULL || v\_addr == 0)

return NULL;

pte\_index = (v\_addr >> 20) & 0xFF; // Want bits [27:20]

pte1 = \*(PT1\_addr + pte\_index);

// piazza note 1010

return (pte1 >> 27) & 1 ? (pte1 & 0x1FFFFFFFFFFFFF) << 11 : NULL;

}

/\* Takes in the base address of Page Table 2 and

\* virtual address as the argument.

\* Returns the base address of the Page if valid.

\* If invalid, returns NULL \*/

void\* PT2\_to\_Page (void\* PT2\_addr, uint32\_t v\_addr) {

uint32\_t pte2\_index, pte2;

if (PT2\_addr == NULL || v\_addr == 0)

return NULL;

pte2\_index = (v\_addr >> 12) & 0xFF; // Want bits [19:12]

pte2 = \*(PT2\_addr + pte2\_index);

return pte2 !=0 ? (void \*) pte2 : NULL;

}

c) 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) {

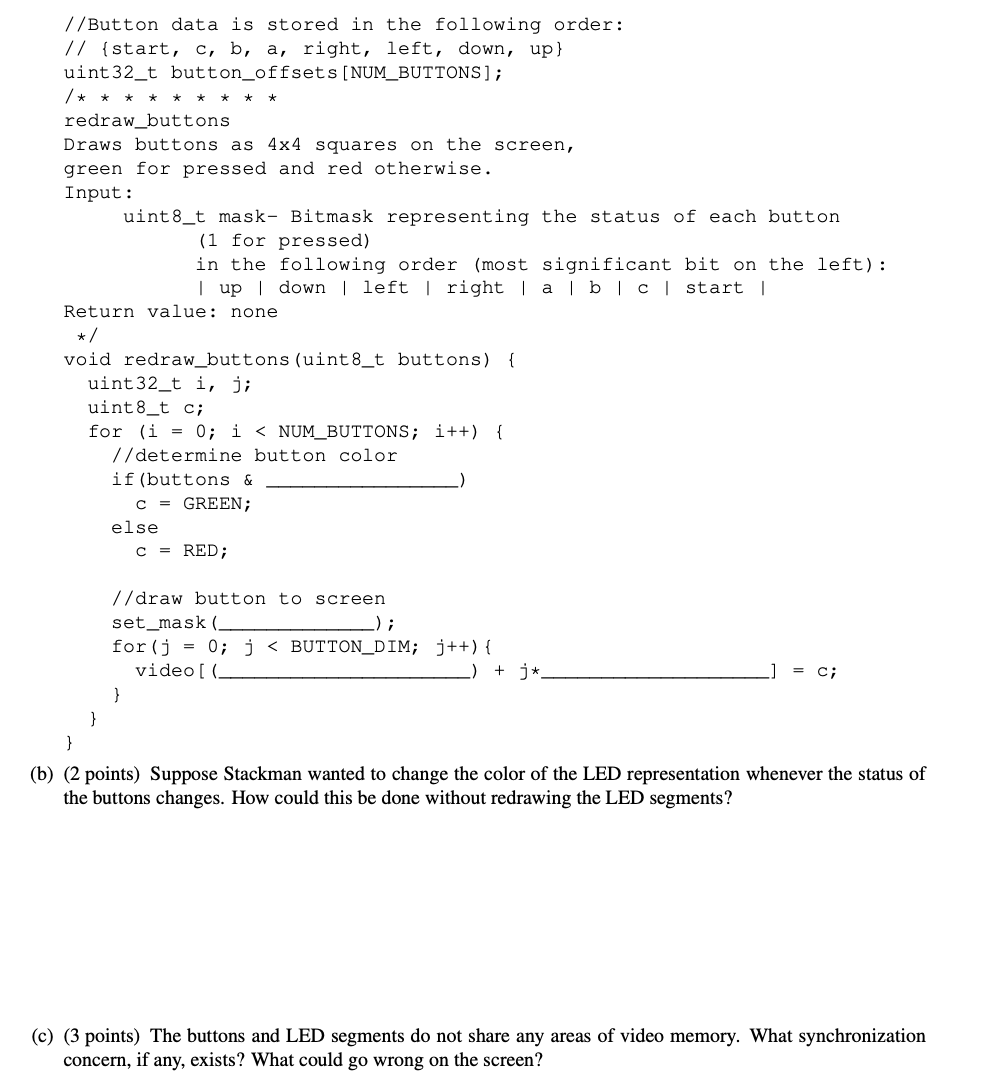
}

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

So with a bigger physical address lets say from 4kb to 4MB you can skip using page tables because there is no need to push unused program data to disk because physical memory is large enough to store it.

Lets say PD size is 4kb then it needs 10 bits for PD and lets say physical pages are 4MB that requires an offset of 22 bits. That is exactly the 32 bits we have for virtual memory so we skip level of indirection. Overall this saves time.

This enables multiple virtual machines, each with 4 GB of virtual memory to run on the same system

Q2: Attack of the Stackman:

(a)

#define SCREEN\_HEIGHT 200

#define SCREEN\_WIDTH 320

#define SCREEN\_AREA (SCREEN\_WIDTH \* SCREEN\_HEIGHT)

#define SCREEN\_PLANE (SCREEN\_AREA >> 2)

#define SCREEN\_PLANE\_WIDTH (SCREEN\_WIDTH >> 2)

//button palette indices

#define RED 2

#define GREEN 3

#define BUTTON\_DIM 4

#define NUM\_BUTTONS 8

uint32\_t button\_offsets[NUM\_BUTTONS];

void set\_mask(uint8\_t mask);

//Pointer to the start of video memory.

uint8\_t \* video;

void redraw\_buttons(uint8\_t buttons)

{

uint32\_t i,j;

uint8\_t c;

for(i=0;i<NUM\_BUTTONS;i++)

{

if(buttons & (1<<(7-i))) // pay attention to this, the sequence is important

{

c = GREEN;

}else{

c= RED;

}

set\_mask(0x0F);

for(j=0;j<BUTTON\_DIM;j++)

{

video[(button\_offsets[i]>>2)+j\*SCREEN\_PLANE\_WIDTH] = c;

}

}

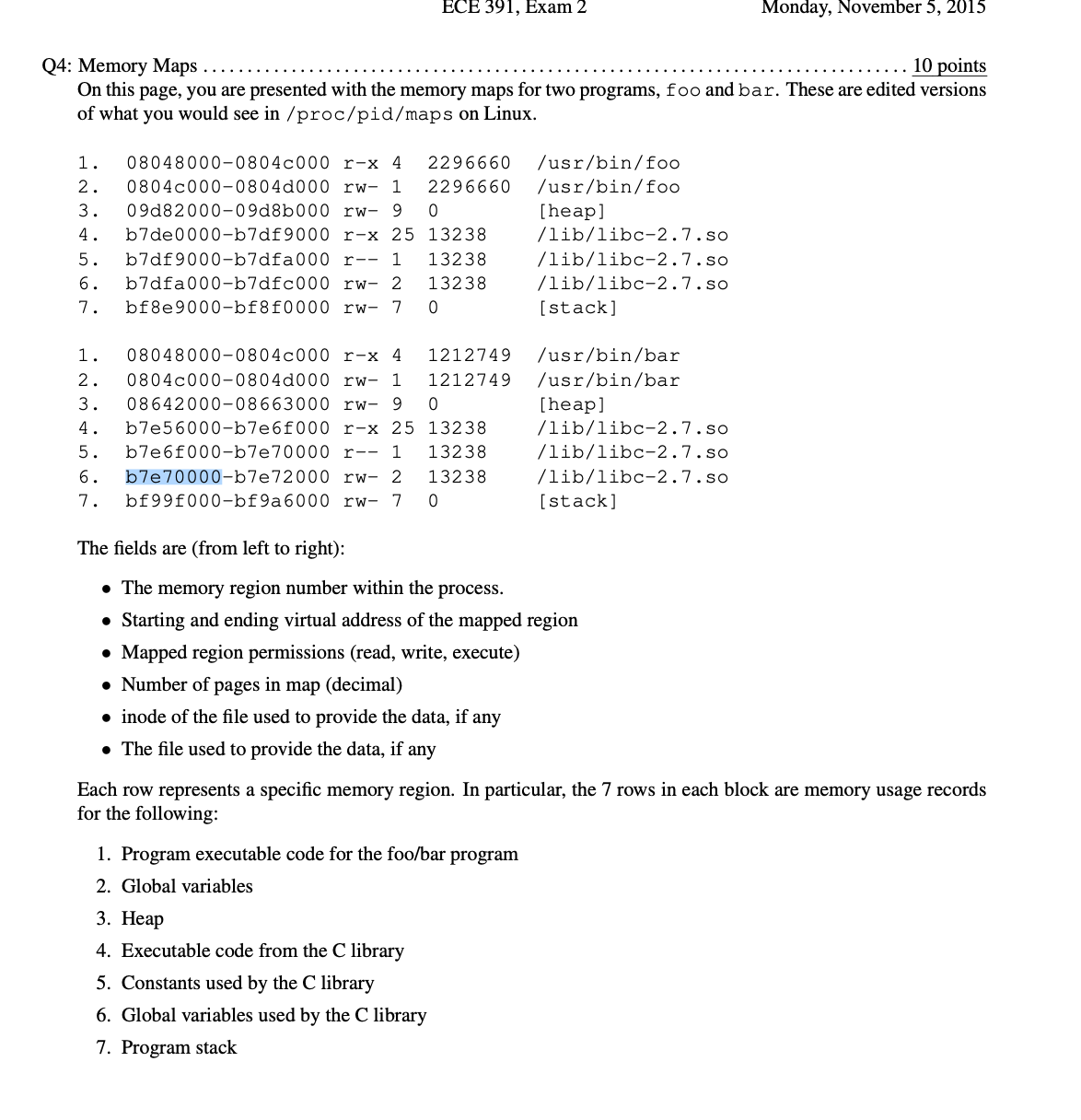
}

(b) I think we should just change the palette.

(c) Errors are caused by palette or set\_mask function. Maybe there the led/buttons will be filled in wrong color. For eg. LED thread may change the mask while the buttons thread is writing to video memory.

Q3: Scheduling:

Q4: Memory Mapping



1. 4,5,6
2. 1,2 4,5,6 ? I think it should be 1,4,5,6 since global variables should not be shared across processes?
3. 10 pages (process 2,6,& 7 are write accessible and 3 wouldn’t be considered since it is the heap).
4. 16kB

Q5: TLB (Similar to Q3 from Fall 2014)

(a)

|  | Round 1 | Round 2 | Round 3 | Round 4 | Round 5 | Round 6 |
| --- | --- | --- | --- | --- | --- | --- |
| TLB entry1 | 0x940000 | 0x940000 | 0x940000 | 0x940000 | 0x940000 | 0x940000 |
| TLB entry2 | Empty | 0x942000 | 0x942000 | 0x946000 | 0x946000 | 0x946000 |
| TLB entry3 | Empty | Empty | 0x944000 | 0x944000 | 0x944000 | 0x942000 |

(b) 2\*(4 + 1023\*3)

(c) int x,y;

Result = 0;

for (y = 0; y < 4; y++){

for (x = 0; x < 2048; x++){

/\* I am too lazy to copy everything over, you get the idea…\*/

result += matrix[0][x] \* matrix[y][x]; // I think that is enough.

}

}

(d) TLB misses:

(0, 0) 1 miss

(0, 1023 - 0, 1024) 1 miss (0, 2047 - 1, 0) 1 miss

(1, 1023 - 1, 1024) 1 miss (1, 2047 - 2, 0) 1 miss

(2, 1023 - 2, 1024) 1 miss (2, 2047 - 3, 0) 1 miss

(3, 1023 - 3, 1024) 1 miss

8 misses in total

Alternate way to calculate: We need to have a miss every 1024 table entries, since at that point we have read the entire page. There are 4\*2048 table entries, (4\*2048)/1024 = 8 total misses

Just a table version of the above math, showing when the TLb misses occur and what they are.

| (x,y) | 0,0 | 0,1024 | 1,0 | 1,1024 | 2,0 | 2,1024 | 3,0 | 3,1024 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| TLB 1 | 0x940 | 0x940 | 0x940 | 0x940 | 0x940 | 0x940 | 0x940 | 0x940 |
| TLB 2 |  | 0x941 | 0x941 | 0x941 | 0x941 | 0x941 | 0x941 | 0x941 |
| TLB 3 |  |  | 0x942 | 0x943 | 0x944 | 0x95 | 9x960 | 9x967 |

(e) We flush the TLB mappings whenever we do context switch, therefore, after every inner loop iteration, we need to calculate a new mapping between the virtual and the physical address.

# ECE 391 Exam 2, Fall 2014

Q1: File System Fariy tale

1. 63 files
2. 1023\*4 kB = 4,190208 Bytes
3. Total = (N+D) \* 4kB = (1+ 63+63)\* 4 kB = 520, 192 bytes’

(63\*4 Bytes)/ 520,129 bytes = 63/130,048

1. The extra bits are added in so that each entry is a size that is a power of 2. (ie round up to 2^6 = 64 bits
2. First pass in the filename. Search the boot block for the entry for that file. Find the index for the inode associated with that file. Go to that inode. Find the indexes for the data blocks for that inode. Go to those data blocks to actually read out the relevant data
3. I assume its because locating the spot to write in, every single time is very inconvenient, and slow because it involves so many memory accesses. Using the PCB to keep the file close at hand should help.

2. a.

| # Level | Size of page | # page offset bits | # index bits / level |
| --- | --- | --- | --- |
| 2 | 4kB | 12 | 10 |
| 20 | 4kB | 12 | 1 |
| 1 | 1GB | 30 | 2 |
| 2 | 4B | 2 | 15 |
| n | s | log2(s) | (32-log2(s))/n |

| # Levels | Size of Page | Size of 1 entry | # of entries | Size of paging data / level |
| --- | --- | --- | --- | --- |
| 2 | 4kB | 4B | 1024 | 4kB |
| 20 | 4kB | 4B | 2 | 8B |
| 1 | 1GB | 4B | 4 | 16B |
| 2 | 4B | 4B | 2^15 | 2^17 |
| n | s | 4B | 2^(# index bits in last row) | 4\*2^(# index bits in last row) |

b. Why would we choose the paging scheme with one level of 1GB pages over the scheme with 20 levels of 4kB pages?

There are 2 reasons to choose the paging scheme with one level of 1 GB pages over one with 20 levels with 4kB pages. Firstly, with 20 levels of paging, in order to fetch the data at a specific page, it has to go through so many levels of indexing which is time consuming. The other reason is that with 1 GB pages, we are more likely to access data from the same page, meaning we will get a really low miss rate in our TLB.

c. Why would we choose the paging scheme with two levels of 4B pages over the scheme with one level of 1GB pages?

The only reason that would make sense for using this over the 1 level of 1GB pages is to avoid having internal fragmentation. Paging already alleviates the risk of having external fragmentation, but internal fragmentation could still persist, and with 1 GB pages, it is way more likely to risk it than with 4B pages.

# ECE 391 Exam 2, Spring 2013

* 1. Suppose a request is made for a large chunk of memory that exceeds the smallest page size supported by the kernel. Give three reasons why it is beneficial for the memory allocator to provide either contiguous page frames or a single large enough page, instead of randomly picking multiple (smaller) free page frames
     1. The way that memory chips work, it is faster to write and read from contiguous physical memory locations
     2. Assigning contiguous pages means its easier for other programs to do the same.
     3. If only a single large page is used, relocating is easier when, say we do a fork() call.
     4. Contiguous memory addresses are better for cache hits. ***For contiguous pages, WHY??***
     5. Maybe contiguous page could further reduce the risk of fragmentation (since paging could not eliminate fragmentation)
  2. Any memory allocator for dynamic requests of various sizes from a common memory pool (such as a buddy allocator) has to implement two core operations on that memory pool, one on allocation and one on deallocation. What are they?

I think this has something to do with Memory maps (not tested)

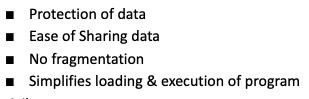
* 1. Explain why disabling interrupts is the only precaution necessary for protecting a critical section that is shared between an application and an interrupt handler in the development environment for this class.

The development environment is uniprocessor thus IF is enough.

* + 1. handler only receive control if interrupt received (due to uniprocessor system), turning off IF prevents handler from taking control, keeping section safe.
  1. Give one reason interrupt chaining is useful.
     1. Can add multiple devices on one interrupt line.
     2. Service multiple handlers from a single IRQ line
  2. Name one disadvantage AND one limitation of the EXT2 file system.
  3. Please fill-in the following sentence with the following terms: User Program, Interrupt Handler, System Call Operating systems leverage special assembly linkage within the (A) and a particular calling convention to allow the (B) to request the service of the desired (C).
     1. A - Interrupt handler, B- User program, C - system call.

// As bob mentioned, I think this is the correct version.

* 1. Linux uses x86 segmentation in a very limited way. All processes in user mode use the same pair of segments, user code segment and user data segment. Similarly, all processes in kernel mode use a pair of segments, kernel code segment and kernel data segment. The Linux operating system effectively makes segmentation a pass-through operation by the way it establishes those segments. Please write the base address (starting address) of these four segments in Linux.
     1. 0x00000000, 0x00000000, 0x00000000, 0x00000000
  2. Virtual memory acts as a logical layer between the application memory requests and the hardware Memory Management Unit (MMU). Please list three advantages of using virtual memory instead of sending application memory addresses directly to the MMU.



* 1. In lecture, one of the solutions we came up with for connecting multiple devices to the single interrupt pin was to connect all devices to an OR gate. Give two reasons why this solution is not used.
     1. There is no way to know which device raised the interrupt
  2. Explain why neither the master nor slave PIC alone has enough information to know what data to send to the CPU during an INTA. For full credit, you must explain why that is the case for both PICs.
     1. The Slave pic does not know if its interrupt is being serviced, just because it raised an interrupt,
     2. Master Pic does not have the data necessary to be sent, must convey signal to slave so it can generate information that actually goes to interrupt handler.

Problem 2

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.

* 1. How many entries are in the PD? in the PT?
     1. There are 4096 entries in both
  2. What is the biggest possible page size? (Assume the Page Size Extension bit has been set)
     1. I think it is 16Mb (2^24)
  3. If only 10 bits are used for options/modifiers in the PTE, what is the smallest possible page granularity supported by a PTE?
     1. Smallest possible is 1 kb
  4. Why is using the smallest granularity possible in the PTE a bad idea for this paging scheme?
     1. Bad idea bc if using granularity of 1 kb with 4kb page size (because 12 bits of offset = 2^12 = 4kb), could Have overlap bw pages

* 1. Why would recycling this MMU scheme for a 32-bit virtual address space (and zero-extending by 4 bits) be a bad idea?
     1. If we zero extend, then the offset field's last 4 bits are zero, and so it can only access locations within a page that are multiple of 2^4, so we waste addresses. This method may run out of memory fast, because the virtual memory is essentially 64 gb, and is 16B-addressable.

**I think the answer is:**

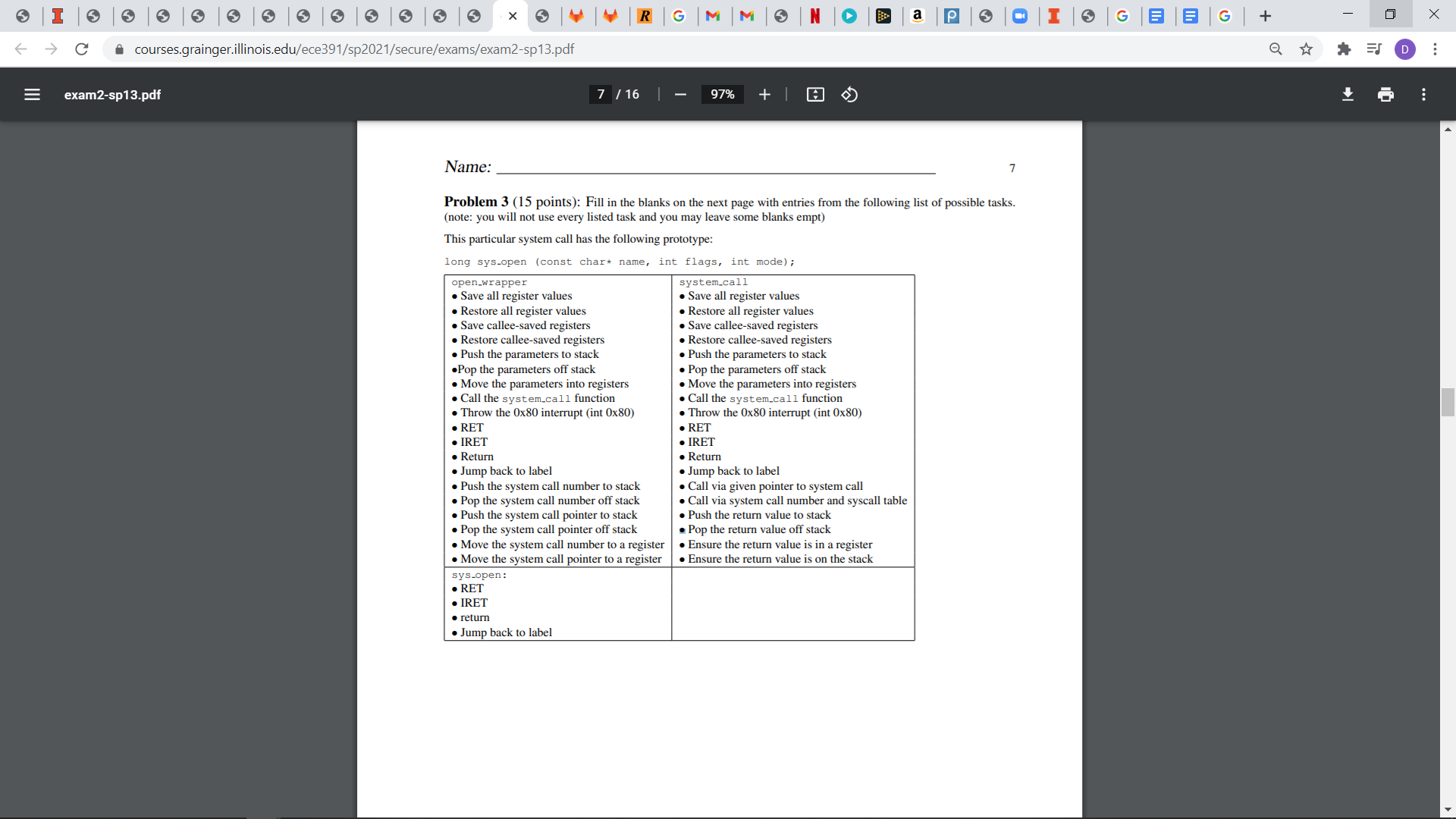
The storage of 36 bits content will conflict with the 32 bits content, that is, we assume that the size of virtual memory cannot be larger than the physical memory in our lecture, because we need to set a map relation between virtual memory and physical memory. At this time, if we recycle the MMU scheme, filling the 36 bits space cannot align to the 32 bits space, that is the problem.

Probably bc 8 to 12 bits did not really use all space in PD, i.e. only 1/16 of the desired case, thus a waste?

A potential answer for this question (I am confused with the above answer)

Maybe the answer is if we add 4 zeros at the start of an original 32 bit virtual memory and then we take 12 12 12 as our level index, it may change the original mapping? I mean, for the original 32 bit va, maybe it is 10 10 12, then we add 4 zeros at the start, then some part of first 10 combine with next 10 (both 10 are in original scheme)

Problem 3 (15 points): Fill in the blanks on the next page with entries from the following list of possible tasks. (note: you will not use every listed task and you may leave some blanks empt) This particular system call has the following prototype: long sys open (const char\* name, int flags, int mode);



| open\_wrapper (x86) | system call\_(x86) | sys\_open (C) |
| --- | --- | --- |
| ... // note: assume all registers are in use  // Open the desired file: → | →  ←  put return value in proper place  //Done: ready to return | //Execute the desired action  //Done: ready to return |

Problem 5:

#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) {

/\* return \*/;

}

end\_x = x + length - 1; //x coordinate of the last pixel

start\_plane = /\* 3 - (x&3) \*/; //plane for first pixel

end\_plane = /\* 3 - (end\_x & 3) \*/; //plane for last pixel

start\_addr = x / /\* 4 \*/ + SCROLL\_X\_WIDTH \* y; //address of first pixel

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

mem\_image[start\_addr] = /\*color\*/;

} else {

SET\_MASK(0xf<<start\_plane); //set mask for first address

mem\_image[start\_addr] = /\*color\*/;

SET\_MASK(/\*0xf\*/); //set mask for addresses between start and end

for(i = start\_addr + 1; i < end\_addr; i++) {

/\* mem\_image[i] = color\*/;

}

SET\_MASK(0xf>>(3 - end\_plane)); //set mask for last address

mem\_image[end\_addr] = /\*color\*/;

}

}

Problem 6: PIC design and interface

1. Missing line outb((EOI+2), Master\_port), this line will be after first slave line code is called. If condition is incorrect, must replace ‘&’ with ‘>=’
2. Only the interrupts on the slave will be affected
3. 8-15

# 

# ECE 391 Exam 2, Spring 2010 (P3 check)

1. Why should you not allocate large objects on the stack in the kernel?
   1. The kernel stack is small in size so if you allocate a large object on the kernel you could run out of memory and overwrite memory.
2. Explain the difference between kmalloc and kmem\_cache\_alloc and the rationale for having the two mechanisms.
   1. Kmalloc: Obtains kernel memory in byte-sized chunks in a contiguous region of memory.
   2. Kmem\_cache\_alloc: Keeps copies of predefined size objects pre-allocated in the cache. So if you have a struct you use a lot it will allocate copies of the struct in memory and will return an address of the struct when a copy is needed. The function is a lot quicker than kmalloc.
3. What mechanism would be used for allocating a vmarea\_struct?
   1. Kmalloc as it needs contiguous memory
4. Explain the purpose of the Translation Lookaside Buffer (TLB)
   1. The TLB is a memory cache used to reduce the time taken to access a memory location. The TLB stores recent memory access information so if we need we don’t need to call directly from memory again which is slow.
   2. Shortened version - TLB used to cache VM to Physical memory lookups for quicker translation thru less memory reads.
5. Why must the TLB be flushed when switching between processes?
   1. When the processes switch there is now a new active page table so all the entries in the TLB are no longer valid.
6. What is the difference between a process and a thread?
   1. Process: Program in execution
   2. Thread: A segment of a process

* Alternate Difference

1. Process : Does not have access to data from other processes
2. Thread : Does have access to data from other threads (as long as they’re within the same process)
3. Describe two advantages and one disadvantage of using paging instead of segmentation?
   1. Pros:
      1. Swapping is easy between equal sized pages
      2. Memory does not have to be contiguous
      3. (additional advantage) Paging requires no intervention by the user
   2. Cons:
      1. Can cause internal fragmentation
4. IRQ list
   1. IR0, IR1, SIR1 - SIR7, IR3 - IR7
   2. I think this answer would be more accurate if we said: IR0, 1, 8, 9, 10 ..15, 3, 4, 5, 6, 7, because they asked to arrange IRQ 0- 15 in order.

4. Virtual Memory

1. PDE = 10 bits
2. PTE1 = 6 bits
3. PTE2 = 6 bits
4. Offset = 10 bits

//YOU CAN NOT RETURN TEMP AS THE FUNCTIONS ARE A POINTER TO AN INT NOT AN INT. IF YOU RETURN TEMP AS AN INT YOU GET A WARNING WHEN YOU COMPILE. IN THE LAST FUNCTION YOU HAVE TO RETURN A DEREFERENCED POINTER TO AN INT. STOP CHANGING THE POINTERS IN THE CODE. IF YOU ARE CONFUSED ON SYNTAX THEN I SUGGEST YOU GO EXPLORE COMPILING THE CODE IN THE ONLINE C COMPILER.

//Problem 4 code

//PD\_addr is the base address in memory of the first page directory

//v\_addr is the offsets in memory of each entry

uint32\_t\* PD\_to\_PT1(uint32\_t\* PD\_addr, uint32\_t v\_addr){

uint32\_t\* PT1\_base\_address;

uint32\_t temp;

//check arguments

if(PD\_addr == NULL)

return NULL;

PT1\_offset = v\_addr >> 22; //gets offset of the page directory entry

PT1\_base\_address = PD\_addr + PT1\_offset; //gets the PA

//check present bit

if((\*PT1\_base\_address & 0x01) != 1)

return NULL;

//bitmask the bottom 8 bits as the PTE1 is size aligned and the size is 256

temp = \*PT1\_base\_address & 0xFFFFFF00;

return temp;

}

uint32\_t\* PT1\_PT2(uint32\_t\* PT1\_addr, uint32\_t v\_addr){

uint32\_t\* PT2\_base\_address;

uint32\_t PT2\_offset;

uint32\_t temp;

//check arguments

if(PT1\_addr == NULL)

return NULL;

PT2\_offset = (v\_addr >> 16) & 0x3F; //gets offset of the page table 1

PT2\_base\_address = PT1\_addr + PT2\_offset; //gets the PA

//check present bit

if((\*PT2\_base\_address & 0x01) != 1)

return NULL;

//bitmask the bottom 8 bits as the PD is size aligned and the size is 256

temp = \*PT2\_base\_address & 0xFFFFFF00;

return temp;

}

uint32\_t\* PT2\_to\_Page(uint32\_t\* PT2\_addr, uint32\_t v\_addr){

uint32\_t\* Page\_base\_address;

uint32\_t Page\_offset;

uint32\_t temp;

//check arguments

if(PT2\_addr == NULL)

return NULL;

Page\_offset = (v\_addr >> 10) & 0x3F; //gets offset of the page table 1

Page\_base\_address = PT2\_addr + Page\_offset; //gets the PA

//check present bit

if((\*Page\_base\_address & 0x01) != 1)

return NULL;

//bitmask the bottom 10 bits as the Page is size aligned and the size is 1024

temp = \*page\_base\_address & 0xFFFFFC00;

return temp;

}

uint32\_t get\_Physical\_address(uint32\_t\* PDBR, uint32\_t v\_addr){

uint32\_t temp\_ptr;

uint32\_t \* page\_dir\_addr;

uint32\_t \* temp\_ptr;

temp\_ptr = \*PDBR >> 12; //get page directory address starting at bit zero

page\_dir\_addr = &temp\_ptr;

//call the functions

temp\_ptr = PD\_to\_PT1(page\_dir\_addr, v\_addr);

if(temp\_ptr == NULL) return NULL;

temp\_ptr = PT1\_PT2(temp\_ptr, v\_addr);

if(temp\_ptr == NULL) return NULL;

temp\_ptr = PT2\_to\_Page(temp\_ptr, v\_addr);

return \*temp\_ptr;

}