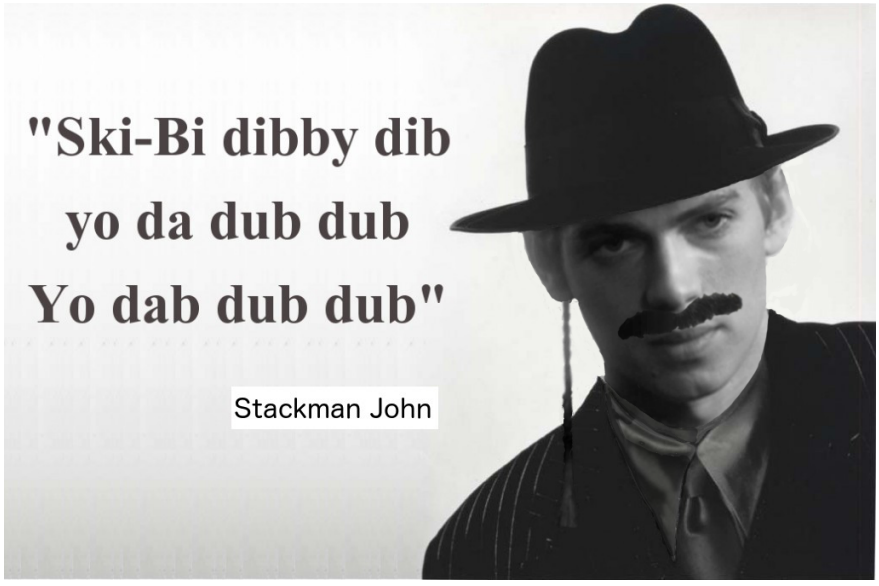
ECE 391 Exam 2, Fall 2015

Thursday, November 5, 2015

Name:_____

NetID:_____

- Be sure that your exam booklet has 13 pages (not including this one).
- Please write your NetID at the bottom of each page in the space provided.
- This is a closed book exam.
- You are allowed two $8\,^1/_2 \times 11$" sheet of notes.
- Absolutely no interaction between students is allowed.
- Show all of your work.
- Don't panic, and good luck!



Page	Points	Score
2	13	
4	5	
5	6	
6	5	
7	11	
8	8	
10	10	
11	5	
12	8	
13	5	
Total:	76	

Q1: Paging ..... 18 points

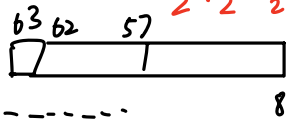
The Physical Address Extension (PAE) mode of x86 uses 3-level page entries to be able to address more than 4 GB of physical memory, while keeping a 32-bit virtual address space. It is your job to create a page table walking functions for this page table system that translate a 32-bit virtual address 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 = 4 bits
PT1 = 8 bits
PT2 = 8 bits
Offset = 12 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) {
    if (PD_addr == NULL) return NULL;
    uint32_t offset = (v_addr & 0xF0000000) >> 28;
    if (offset > 15) return NULL;
    uint64_t* PD_entry = (uint64_t*)(PD_addr) + offset;
    uint64_t valid = (*PD_entry & 0x8000000000000000) >> 63;
    if (valid != 1) return NULL;
    uint64_t PT1_addr = (*PD_entry) << 11;
    return (uint64_t*) PT1_addr;
}
```

```
/* 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) {
    if (PT1_addr ==NULL) return NULL;
    uint32_t offset = (v_addr & 0x0FF00000)>>20 ;
    if (offset >255) return NULL;
    uint64_t* PT_entry = (uint64_t *)(PT1_addr)+ offset;
    uint64_t valid=(*PT_entry & 0x8000000000000000) >>63;
    if (valid != 1) return NULL;
    uint64_t PT2_addr=(*PD_entry &[illegible])<< 11;
    return (uint64_t*) PT2_addr;




}

/* 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) {
    if (PT2_addr ==NULL) return NULL;
    uint32_t offset = (v_addr & 0x000FF000)>>12 ;
    if (offset >255) return NULL;
    uint64_t* PT_entry = (uint64_t *)(PT2_addr)+ offset;
    uint64_t valid=(*PT_entry & 0x8000000000000000) >>63;
    if (valid != 1) return NULL;
    uint64_t p_addr=(*PD_entry &[illegible]) <<12;
    return (uint64_t*) p_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) {
    /* input checked in helper function */
    uint64_t* PT1_addr = PD_to_PT1(PDBR, v_addr);
    if (PT1_addr == NULL) return NULL;
    uint64_t *PT2_addr = PT1_to_PT2(PT1_addr, v_addr);
    if (PT2_addr == NULL) return NULL;
    uint64_t* P_addr = PT2_to_Page(PT2_addr, v_addr);
    if( P_addr == NULL) return NULL;
    uint8_t* ret = (uint8_t*)(P_addr) + (v_addr & 0x00000FFF)
    return (uint64_t*) ret;
}
```

use more programs

(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?

than 4GB

We can have more physical memory area to store code and data as we don't need the one-to-one mapping from virtual memory to every physical memory, only to important physical memory such as head of a linked list.

Q2: Attack of the Stackman ..... 11 points

All italicized text is completely optional and unnecessary

There was unrest in the college of engineering. Several incidents of network failure and data loss had frustrated students enough to draw the attention of Obi-Wan Kebailey and his young apprentice, Stackman John. While Kebailey visited various buildings to investigate, Stackman stayed in a computer lab with LC-3PO and R2-EK to work on a test program for the Tux controller, which he had heard was used by ECE 391 students; he wanted to be just like them.

After several hours of development, EWS went down yet again, while Stackman was editing the graphical component of his program. Not knowing when it would be possible to continue his work, Stackman set out to investigate on his own, starting with 238 Everitt (the 391 lab at the time). He spotted a shadowy figure in the doorway and ducked behind a corner to listen. The figure was talking on a phone, and from what Stackman could tell, he was a TA of ECE 391, quite possibly the best. The students had procrastinated on a machine problem, and all of the last minute debugging was overloading the network. Somewhat bitterly, he added that due to his relative fame, the students blamed all problems on him, regardless of the situation.

Suddenly, LC-3PO and R2-EK approached in a less than stealthy manner (loudly debating the proper way to indent code). Stackman was not sure if the imposing figure would be as wrathful as the students seemed to think, but he did not want to be caught lurking suspiciously in a dimly lit hallway, so he flew down the stairs and ran across the street to hide in a crowded restaurant. One plate of hot and spicy chicken later, Stackman headed to the depths of Siebel, where he found Kebailey and relayed what he had discovered so they could discuss potential solutions.

A week later, Stackman and Kebailey rode into 238 Everitt on a festive pink pony, bearing exciting news for ECE 391: they would be receiving a server for their class alone, so they would hopefully avoid similar problems in the future. The students and several teaching assistants cheered, but among the balloons and confetti, Stackman could not help but detect one notable absence...

Stackman John wrote a Tux driver test program which can display the button status and expected LED status on the screen in Mode X. The LED segments are drawn to the screen when the program attempts to set the LEDs on the device. Buttons are redrawn when a change in status is detected; button and LED drawing are handled in separate threads. For simplicity, all details are drawn directly to video memory (no build buffer).

(a) (6 points) Stackman lost several unsaved changes to his code. Complete the `redraw_buttons` function by filling in the blanks. You may assume that the rest of the code is correct and that button images always start at logical addresses that are multiples of 4.

```
#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
/* Provided function defined elsewhere.
 * Sets draw mask using lowest four bits, so 0x01 corresponds to plane 0,
 * 0x02 corresponds to plane 1, etc.
 */
void set_mask(uint8_t mask);
//Pointer to the start of video memory.
uint8_t * video;
//Array of logical coordinates to upper left corner of each button, in row
major.
```

0001 0
0010 1
0100 2
1000 3

```
//Button data is stored in the following order:
// {start, c, b, a, right, left, down, up}
uint32_t button_offsets[NUM_BUTTONS];
/* * * * * * * * *
redraw_buttons
Draws buttons as 4x4 squares on the screen,
green for pressed and red otherwise.
Input:
     uint8_t mask- Bitmask representing the status of each button
           (1 for pressed)
           in the following order (most significant bit on the left):
           | up | down | left | right | a | b | c | start |
Return value: none
 */
void redraw_buttons(uint8_t buttons) {
  uint32_t i, j;
  uint8_t c;
  for (i = 0; i < NUM_BUTTONS; i++) {
    //determine button color
    if(buttons & (0x1 << i))
      c = GREEN;
    else
      c = RED;

    //draw button to screen
    set_mask(0x0F);
    for(j = 0; j < BUTTON_DIM; j++){
      video[(button_offsets[i]/4) + j*SCREENPLANE] = c;
    }
  }
}
```

(b) (2 points) Suppose Stackman wanted to change the color of the LED representation whenever the status of the buttons changes. How could this be done without redrawing the LED segments?

Let the color of the LED be a global variable C, change value of C whenever the status of the buttons changes

change the RGB color of that index in the color palette

(c) (3 points) The buttons and LED segments do not share any areas of video memory. What synchronization concern, if any, exists? What could go wrong on the screen?

There may exist race condition between buttons and LED that leads to video updated after buttons been pressed but the status of LED has not been updated

Set_mask is the main synchronization concern if one thread is writing to the video memory while the other wants to start it may overwrite the mask desired by the other thread and cause pixels to be in the incorrect page

Q3 Scheduling . . . . . 19 points

(a) (8 points) Listed below is a table of processes and associated arrival and run times.

Process ID	Arrival Time	Run Time
A	0	4
B	2	3
C	4	5
D	6	1

Show the scheduling order for these processes under the following schedulers:

- First-In-First-Out (FIFO)
- Shortest-Job First (SJF) with run to completion
- Round-Robin (RR) with a quantum = 2 time units

Assume that the context switch overhead is 0 and the new processes should be scheduled first if all else is equal. A process arriving at time t can first be scheduled for time slot t to $t + 1$. If a process ends part way through a quantum, the next process is scheduled immediately.

Time Slot	FIFO	SJF	RR
0-1	A	A	A
1-2			
2-3			
3-4			
4-5			
5-6			
6-7			
7-8			
8-9			
9-10			
10-11			
11-12			
12-13			
13-14			
14-15			
15-16			
16-17			
17-18			

(b) (1 point) How long is a typical time slice (quantum)? Circle the correct choice.

A. 10–100 ns
B. 10–100 ms
C. 10–100 s

(c) (2 points) What measurement of scheduler performance gets better as the quantum gets bigger? What measurement of scheduler performance gets worse as quantum gets bigger?

Better:

Worse:

(d) (2 points) In practice, why can't we have an arbitrarily small quantum?

(e) (2 points) What function call in the kernel, previously discussed in class, might cause the process to go to sleep before its allotted time and cause another process to be scheduled?

(f) (4 points) In what type of computing environment would a FIFO make the most sense and why? In what type of computing environment would a RR make the more sense and why? (Environments include but are not limited to: home desktop, server, supercomputer, etc. Choose one environment per question.)

Q4: Memory Maps . . . . . 10 points

On this page, you are presented with the memory maps for two programs, `foo` and `bar`. These are edited versions of what you would see in `/proc/pid/maps` on Linux.

```
1.  08048000-0804c000 r-x 4  2296660  /usr/bin/foo
2.  0804c000-0804d000 rw- 1  2296660  /usr/bin/foo
3.  09d82000-09d8b000 rw- 9  0        [heap]
4.  b7de0000-b7df9000 r-x 25 13238    /lib/libc-2.7.so
5.  b7df9000-b7dfa000 r-- 1  13238    /lib/libc-2.7.so
6.  b7dfa000-b7dfc000 rw- 2  13238    /lib/libc-2.7.so
7.  bf8e9000-bf8f0000 rw- 7  0        [stack]

1.  08048000-0804c000 r-x 4  1212749  /usr/bin/bar
2.  0804c000-0804d000 rw- 1  1212749  /usr/bin/bar
3.  08642000-08663000 rw- 9  0        [heap]
4.  b7e56000-b7e6f000 r-x 25 13238    /lib/libc-2.7.so
5.  b7e6f000-b7e70000 r-- 1  13238    /lib/libc-2.7.so
6.  b7e70000-b7e72000 rw- 2  13238    /lib/libc-2.7.so
7.  bf99f000-bf9a6000 rw- 7  0        [stack]
```

The fields are (from left to right):

- The memory region number within the process.
- Starting and ending virtual address of the mapped region
- Mapped region permissions (read, write, execute)
- Number of pages in map (decimal)
- inode of the file used to provide the data, if any
- The file used to provide the data, if any

Each row represents a specific memory region. In particular, the 7 rows in each block are memory usage records for the following:

1. Program executable code for the foo/bar program
2. Global variables
3. Heap
4. Executable code from the C library
5. Constants used by the C library
6. Global variables used by the C library
7. Program stack

(a) (4 points) Which of the areas are shared between the `foo` and `bar` processes? (I.e., use the same physical memory.) List the area numbers (1–7)

(b) (4 points) The user starts a new `foo` process (from another shell). Which areas are shared between the two `foo` processes?

(c) (2 points) The `bar` process executes `fork()` to create a copy of itself. How much extra physical memory is used right after the `fork()` call? Only count physical pages, and not the space used by page tables or kernel data structures. Explain your answer.

(d) (2 points (bonus)) How much space is used by the page tables of the `bar` process? Assume that all pages of each memory region are currently present in physical memory. Do not include the space used by the page directory or any kernel data structures.

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];
  }
}
```

2 page per y

(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	0x940000	0x940000	0x940000	0x944000	0x944000
TLB entry 2	empty	0x942000	0x942000	0x946000	0x946000	0x946000
TLB entry 3	empty	empty	0x944000	0x944000	0x944000	0x942000

0 1
1 1
2 1
3 1
0 0
1 1
2 1
3 1

0x944000 0x946000 0x944000
0x941000

(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).?

misses = (4 + 1023 - 3) · 2 = 6146

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

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

0	1
1	0
2	0
3	0

0x940000
0x941000
0x942000
0x943000
5
6
7

(d) (2 points) How many TLB misses will occur during the execution of your revised code?

miss = 1+1+1+1+1+1+1+1 = 8 ✓

(e) (3 points) Your partner, Ben Bitdiddle, has configured the scheduler so that there is a context switch after every inner loop iteration. He notes, with pride, that your optimized code and the original now both incur the same number of TLB misses. Explain why.

may change page structure

every content switch will flushes the TLB, that results into a TLB miss

page Hit