CS 3224 Midterm

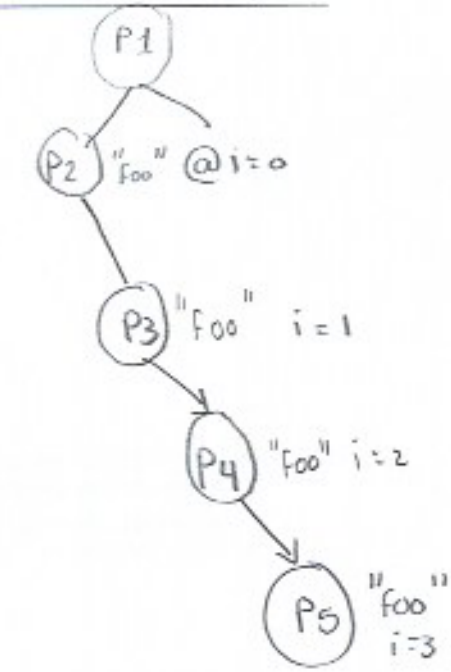
March 24, 2016

Answer the questions in the spaces provided on the question sheets. If you run out of room for an answer, you can continue on the back of the page.

Question	Points	Score
1	10	
2	10	
3	10	
4	15	
5	10	
6	10	
7	10	
8	10	
9	10	
10	5	
Total:	100	

Name: ______________________________

1. Consider the following C program:




```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
    int i = 0;
    for (i = 0; i < 4; i++) {
        if (fork() == 0) {
            printf("foo\n");
        }
        else {
            wait(NULL);
            exit(1);
        }
    }

    return 0;
}
```

(a) (4 points) How many times will "foo" be printed? ~~3~~ 4 times

(b) (4 points) How many processes will be created (including the initial process)?

5 Procs ~ Processes.

(c) (2 points) Will any processes become a zombie? Why or why not?

when the child exits and parent hasn't called wait on it.

0 zombies.

Ex: fork 3 times and wait only twice then you end up with 1 zombie

2. Below is some 32-bit x86 assembly code that implements a mystery function with the signature `int mystery(int n)`. Read it and then answer the following questions.

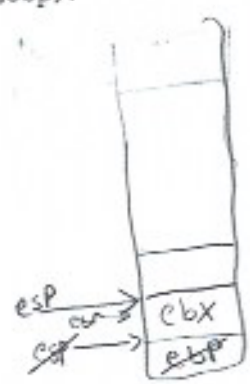
Args: pushed to stack
Return value eax

```
mystery:
        push    %ebp
        mov     %esp,%ebp
        push    %ebx
        sub     $0x4,%esp
        cmpl    $0x0,0x8(%ebp)
        jne     mystery_label1
        mov     $0x1,%eax
        jmp     mystery_done
mystery_label1:
        cmpl    $0x1,0x8(%ebp)
        jne     mystery_label2
        mov     $0x1,%eax
        jmp     mystery_done
mystery_label2:
        mov     0x8(%ebp),%eax
        dec     %eax
        push    %eax
        call    mystery
        add     $0x4,%esp
        mov     %eax,%ebx
        mov     0x8(%ebp),%eax
        sub     $0x2,%eax
        push    %eax
        call    mystery
        add     $0x4,%esp
        add     %ebx,%eax
mystery_done:
        mov     -0x4(%ebp),%ebx
        mov     %ebp,%esp
        pop     %ebp
        ret
```

setting up the stack

Reminders:

- Values starting with `$` are constants, those starting with `%` are registers.
- The syntax `0x8(%ebp)` is a reference to the data in memory at address `%ebp + 0x8`.
- The GCC calling convention (used here) specifies that arguments are pushed on the stack, return values are placed in `%eax`, and the caller is responsible for clearing any arguments pushed once the called function returns.
- *Hint*: The argument to the mystery function is at `0x8(%ebp)`.



Answer(a)
int mystery(2){
if (n==0){
return 1;
else if (n==1)
return 1

(a) (5 points) What is the value returned by `mystery(2)`?

0→1 2→2 4→5
1→1 3→3 5→8

fibonacci

(b) (5 points) What function does this compute? (If you don't know its name, you can just give the values of `mystery(n)` for $n = 0, 1, 2, 3, 4, 5$)

b) fibonacci

b) 0→1 2→2 4→5
1→1 3→3 5→8

eax = n;
eax --
ebx = mystery(n-1)
eax = n;
eax = eax - 2;
eax = mystery(n-2)

return(mystery(n-1) + mystery(n-2));

3. Three batch jobs, A through C, arrive at a computer center at the same time. They have estimated running times of 10, 6, 2 minutes respectively. At time 3, jobs D and E arrive, which take 4 and 8 minutes respectively. For each of the following scheduling algorithms, determine the mean process turnaround time. Ignore process switching overhead.

   (a) (5 points) First-come, first-served (run in order A, B, C, D, E).

   (b) (5 points) Shortest job first.

4. The following code from xv6 (slightly simplified here) gets a single integer argument from the user program during a system call.

```
1 // Fetch the nth 32-bit system call argument.
2 int fetchint(int n, int *ip)
3 {
4   uint addr = proc->tf->esp + 4 + 4*n;
5   if(addr >= proc->sz || addr+4 > proc->sz)
6     return -1;
7   *ip = *(int*)(addr);
8   return 0;
9 }
```

Alyssa P. Hacker decides to change the system call convention. She decides that the first and second arguments to the system call will be passed in the `%eax` and `%ebx` registers, and any remaining arguments will be passed on the stack.

(a) (10 points) Rewrite the `fetchint` function to match this new convention by filling in the code below:

```
// Fetch the nth 32-bit system call argument.
int fetchint(int n, int *ip)
{
  uint val;
  if (n == 0) {
   val = Proc -> tf -> eax;

  }
  else if (n == 1) {
   val = Proc -> tf -> ebx;

  }
  else { unit addr = Proc -> tf -> esp + 4 + 4 * (n-2);
   if (addr ≥ Proc -> sz || addr+4 > Proc -> sz)
        return -1;
        val = * (int*) addr;
  }
  *ip = val;
  return 0;
}
```

(b) (5 points) Is the check at line 5 of the original `fetchint` still needed for the first three arguments? Why or why not?

5. A computer has four page frames. The time of loading, time of last access, and the R (read) and M (modified) bits for each page are as shown below (the times are in clock ticks):

Page	Loaded	Last ref.	R	M
0	26	280	1	0
1	230	265	0	1
2	140	270	0	0
3	110	285	1	1

(a) (2 points) Which page will NRU replace? 2

(b) (2 points) Which page will FIFO replace? 0

(c) (2 points) Which page will LRU replace? 1

(d) (2 points) Which page will second chance replace? 2

(e) (2 points) Why is exact LRU not usually used for memory page replacement in real systems? Write ts every access

6. Suppose that a (very slow) computer can read or write a memory word in 100 microseconds (1/10 of a millisecond). Also suppose that when the timer interrupt occurs, 32 CPU registers (each one word), plus the program counter (also one word) are saved by pushing them onto the stack.

   (a) (5 points) If the interrupt handler just acknowledges the interrupt (which takes 3.4 milliseconds) and then returns (restoring the saved CPU registers and program counter), how long does it take to handle one timer interrupt? $1/10$ ms

   $33 \times \frac{1}{10}$ ms

   $+$ 34 ms

   $+ 33 \times 1/10$ ms

   3.3

   $+$ 3.4

   $+$ 3.3

   10 ms

   (b) (5 points) Assuming the system timer ticks once every 100 milliseconds, what percentage of the system's time is spent handling timer interrupts?

   $\frac{10}{100} = 10\%$

7. (10 points) A fictional x86 PC has a mouse controller that works as follows. Whenever the mouse is moved, it generates interrupt number IRQ_MOUSE. The amount the mouse has moved can then be read out using port I/O. The mouse controller works by writing the axis you want to read (0 for the x axis, 1 for the y axis) out to port 0xBEEF; the amount the mouse has moved along that axis (a single byte, 0-255) can then be read out on port 0xF00D.

Use this information to fill out the code below. Your interrupt handler should get the amount of movement on the x and y axes, and then use cprintf to print out a message like "Mouse moved by x=123 y=22". You can use the inb and outb functions for port I/O from C; their prototypes are given below.

```
// Reads in one byte from the given port
uchar inb(ushort port);

// Writes a byte out to the specified port
void outb(ushort port, uchar data);

void trap(struct trapframe *tf)
{
  // [...]
  switch(tf->trapno){
  // [...]
  case T_IRQ0 + IRQ_KBD:
    kbdintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_MOUSE:
  // Your code here


    break;
  }
  // [...]
}

void mouseintr(void) {
  // Your code here










}
```

outb(0xBEEF,0);
int xdiff = inb(0XF00D);
outb(0xBEEF,1);
int ydiff = inb(0XF00D);
cPrintf("Mouse moved by x= %u
y = %u\n", xdiff,
ydiff);

8. On the next page is the code for the `scheduler` function in xv6, which picks the next process to run. Refer to it as you answer the following questions:

   (a) (4 points) What scheduling algorithm is implemented by this code? Is it fair (i.e., does each process get an equal share of the CPU)?

   Round Robin.
   yes it's fair, because each process get an opportunity to work on a compute

   NO: I/O bound run less often
   No priorities

   (b) (4 points) On lines 20–21, we skip the rest of the loop if the process state is not `RUNNABLE`. What is an example of another state that a process could be in? What would go wrong if we tried to run a process in that state?

   P→state = ZOMBIE

   (c) (2 points) The `swtch` function, called on line 29, is written in assembly. What is its purpose, and why does it need to be written in assembly rather than C?

```
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run
//  - swtch to start running that process
//  - eventually that process transfers control
//      via swtch back to the scheduler.
void
scheduler(void)
{
  struct proc *p;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      proc = 0;
    }
    release(&ptable.lock);

  }
}
```

9. Recall that in 32-bit x86, page directories and page tables are each made up of 1024 32-bit entries. Suppose we have 4 processes on a system, each of which has every possible virtual address mapped.

   (a) (5 points) How much memory is used to store the page directories and page tables if 4KB pages are used?

4procs

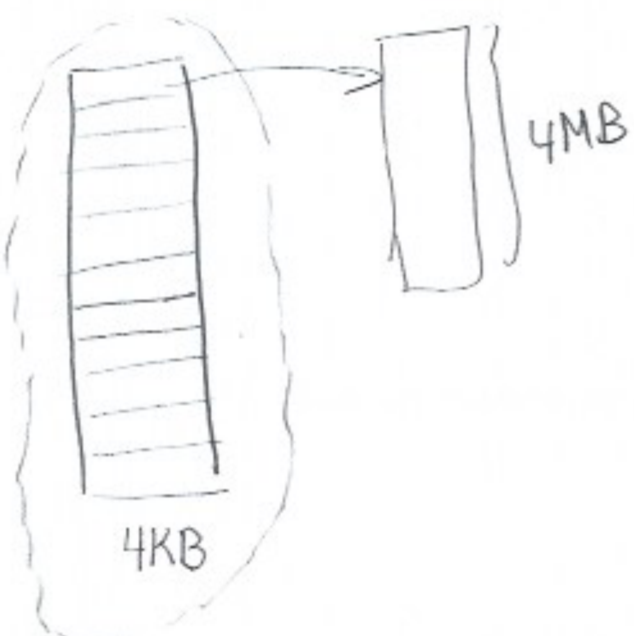
$\hookrightarrow 4 \times (1PD + 1024 PT) \times 4KB$

$4 \times 1024 = 4MB$

$4 \times (4MB + 4KB) = \sim 16MB$

   (b) (5 points) If 4MB pages (*super pages*) are used, then the entries in the page directory point directly to the page frame (i.e., no second-level page tables are used). How much memory would be taken up by page directories in this case?

~~$4 \times (1PD) \times 4KB \rightarrow 16KB$~~



10. (5 points) (a) What is the most interesting bug you've written in this course so far, and what was the solution?

(b) What is the best part of this course so far?

(c) What is the worst part of this course so far (*aside* from this exam)?