

**University of Waterloo**  
**Midterm Examination**  
**Term: Fall Year: 2013**

Student Family Name \_\_\_\_\_

Student Given Name \_\_\_\_\_

Student ID Number \_\_\_\_\_

Section : Circle one    (Brecht 8:30)    (Brecht 2:30)    (Salem 4:00)

Course Abbreviation and Number: CS 350  
Course Title: Operating Systems  
Section(s): 2  
Instructors: Tim Brecht and Ken Salem

Date of Exam: October 30, 2013  
Time Period Start time: 7:00 pm End time: 9:00 pm  
Duration of Exam: 120 minutes  
Number of Exam Pages: 9 (including cover sheet)

**NO CALCULATORS, NO ADDITIONAL MATERIAL**

Problem	Topic	Marks	Score	Marker's Initials
1	Miscellaneous	10		
2	Threads / Thread Fork	8		
3	Forking Processes	8		
4	System calls, Interrupts and Stacks	10		
5	Synchronization (Semaphores)	13		
6	Synchronization (Spinlocks)	9		
7	Synchronization (Reader/Writer Locks)	12		
8	Virtual Memory (Address Translation)	10		
Total		80		



**Problem 2 (8 marks)**

Consider the program below, and suppose that a single initial thread starts executing the `main()` function. As the initial thread runs, additional threads are created as a result of calls to `thread_fork()`. Answer the questions below about the output of this concurrent program.

```
main() {
    helper(NULL,0);
}

void
helper(void *p, unsigned long i) { /* parameter p is not used */
    if (i < 3) {
        kprintf("%ld",i); /* print the value of i */
        thread_fork("helper1",NULL,helper,NULL,i+1); /* fork thread to run helper(NULL,i+1) */
        thread_fork("helper2",NULL,helper,NULL,i+1); /* fork thread to run helper(NULL,i+1) */
    }
    /* was i=0, as announced this should be i==0 */
    if (i==0) {
        kprintf("%ld",i);
    }
    thread_exit();
}
```

Which of the following outputs could possibly be generated by the concurrent program shown above? Write **YES** after each output that could possibly be generated, and write **NO** after each output that could not possibly be generated.

Note: to avoid rewarding random guessing, the marking scheme for this question awards points only for 5 or more correct answers.

- a. 01221220
- b. 01120222
- c. 01342560
- d. 01222120
- e. 01122220
- f. 01234560
- g. 01212220
- h. 00112222

**Problem 3 (8 marks)**

---

(8 mark(s)) For the program shown below, fill in the table at the bottom of the page to show the output that would be printed by both the parent and child processes. Briefly explain how you arrived at your solution. Assume that all function, library and system calls are successful. If you need to make additional assumptions be sure to clearly state them.

```
int x;

main()
{
    int rc;
    x = 0;

    rc = fork();

    if (rc == 0) {
        x = 10;
        printf("A: %d\n", x);
    } else {
        printf("B: %d\n", x);
        x = 100;
    }

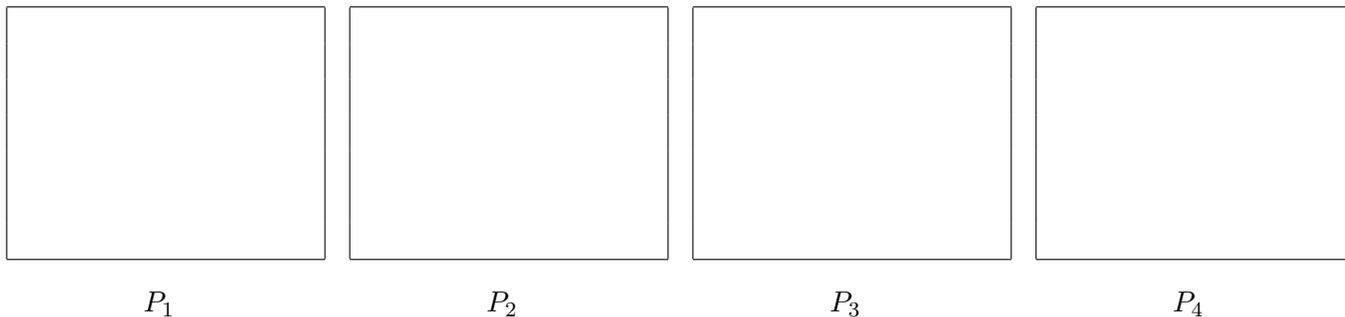
    printf("C: %d\n", x);
}
```

Parent	Child

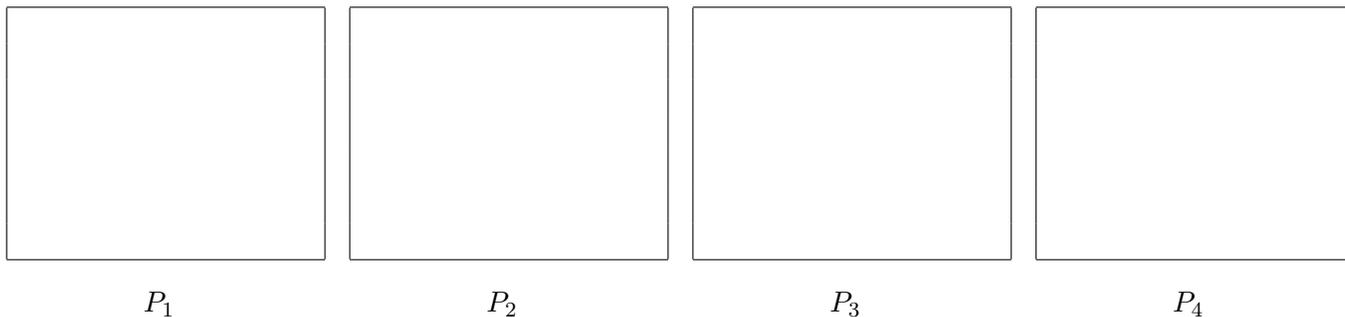
**Problem 4 (10 marks)**

Consider an OS/161 system with 2 CPUs ( $C_1$  and  $C_2$ ) and 4 processes ( $P_1$ ,  $P_2$ ,  $P_3$ , and  $P_4$ ). Processes  $P_1$  and  $P_2$  only ever run on CPU  $C_1$ . Currently  $P_1$  is running and  $P_2$  is on the ready-to-run queue for  $C_1$ . Processes  $P_3$  and  $P_4$  only ever run on CPU  $C_2$ . Currently  $P_3$  is running and  $P_4$  is on the ready-to-run queue for  $C_2$ . When answering the questions below recall that stacks will grow from high addresses to low addresses. Your diagrams MUST place high addresses at the top of the stack picture.

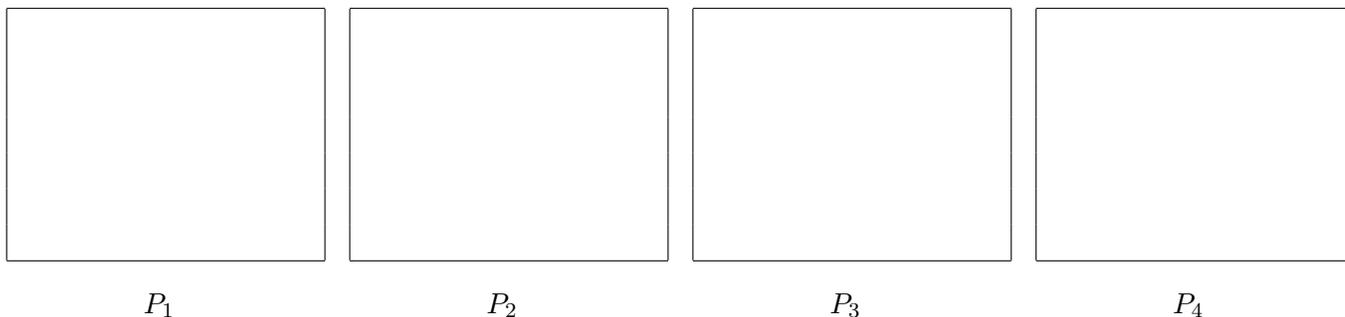
- a. (4 mark(s)) The boxes in the diagram below represent the kernel stacks of the four processes. Complete the diagram to show what the kernel stacks currently contain. Include only trap frames and switch frames and do not include frames that have been popped. Mark any empty stacks with a large X.



- b. (3 mark(s)) Next, assume that a **blocking** system call is executed by  $P_1$ , and  $P_2$  runs on CPU  $C_1$  after the system call. Fill in the diagrams below to show the contents of the kernel stacks for each of the 4 processes after execution of  $P_2$  resumes. Include only trap frames and kernel frames and do not include any that have been popped. Mark any empty stacks with a large X.



- c. (3 mark(s)) Next, assume that a timer interrupt occurs on CPU  $C_2$ . While handling the interrupt, OS/161 decides that  $P_3$ 's scheduling quantum has expired, and that  $P_4$  should start running next on  $C_2$ . Fill in the diagrams below to show the contents of the kernel stacks for each of the 4 processes at the point in the OS/161 kernel just before `thread_switch` is called to switch from  $P_3$  to  $P_4$ . Include only trap frames and switch frames and do not include any that have been popped. Mark any empty stacks with a large X.



**Problem 5 (13 marks)**

Suppose that a system initially has only a single thread (T1), which is running the function `f1()`. `f1()` calls `thread_fork()` to create a new thread, T2, to run function `f2()` concurrently with T1. These two threads and the functions they execute are illustrated in the two columns of the figure below.

As illustrated in the figure, thread T1 calls function `fb()`, and thread T2 calls functions `fa()` and `fc()`. Your task is to synchronize the calls to `fa()`, `fb()`, and `fc()` so that the following synchronization rules are enforced:

- function `fa()` finishes before function `fb()` is called, and
- function `fb()` finishes before function `fc()` is called.

Show how to use semaphore(s) to enforce these synchronization rules by adding `P()` and `V()` calls in suitable places in functions `f1()` and `f2()` in the diagram below. You must also declare (at the top of the diagram) global variables to point to any semaphores you use in your solution. Finally, you must include `sem_create` calls in suitable places in `f1()` and/or `f2()` to create and initialize the semaphores you need. Be sure that your `sem_create()` calls show the initial semaphore value for each newly created semaphore.

Do not use any synchronization primitives or techniques other than semaphores. Keep your solution as simple as possible - unnecessarily complex solutions may be penalized.

<pre>/* declare any global semaphore variables you need here, like this: */ /* struct semaphore *s1 */</pre>	
Thread T1	Thread T2
<pre>void f1() {      /* create thread T2 to run f2() */     thread_fork(...)      /* call function fb() */     fb();      thread_exit(); }</pre>	<pre>void f2() {      /* call function fa() */     fa();      /* call function fc() */     fc();      thread_exit(); }</pre>



**Problem 7 (12 marks)**

---

- a. (8 mark(s)) Assume that each of the functions below (FuncA, FuncB, and FuncC) are being executed by different threads using the OS/161 kernel thread library. Assume that reader/writer locks have been implemented as described in the course notes and in class. Add calls to `rwlock_acquire(rwlock *lk, READ_MODE)` or `rwlock_acquire(rwlock *lk, WRITE_MODE)`, and `rwlock_release(rwlock *lk)` and only those calls to ensure that FuncA, FuncB, and FuncC are atomic. The locks have already been declared and initialized and `xlock`, `ylock`, and `zlock` must be used to protect the variables `x`, `y`, and `z`, respectively. Additionally, your locks **must not use WRITE\_MODE unnecessarily, AND you must ensure that that deadlock can not occur.**

```
volatile int x = 10;
volatile int y = 20;
volatile int z = 0;
struct rwlock *xlock; /* protect x */
struct rwlock *ylock; /* protect y */
struct rwlock *zlock; /* protect z */

void init() {
    xlock = rwlock_create("xlock");
    ylock = rwlock_create("ylock");
    zlock = rwlock_create("zlock");
}

void FuncB(int i) {

    if ((i == z) || (i == x)) {
        x = z + x;
    }

}

void FuncA() {

    if (y == 10) {
        y = y + z;
    }

}

void FuncC() {

    x = y + x + z;
    y = z + x;

}
```

- 
- b. (2 mark(s)) Briefly explain why deadlock can not occur with your solution.

- c. (2 mark(s)) In your solution, which if any of the functions can be executing in their critical sections concurrently. Briefly explain your answer.

**Problem 8 (10 marks)**

---

Suppose that a system uses a single CPU with an MMU that uses a relocation register and a max address (or address limit) register to perform dynamic relocation. Several processes are running in this system. Each process has a simple virtual address space with virtual address ranging from 0 to the maximum address (or limit address) for that process. The kernel maintains a list of processes, their maximum virtual address (limit address) and where in physical memory each process is located (the relocation address), as shown below. All of the addresses used in this problem are decimal (base 10) numbers.

PID	Max Addr	Relocation Addr
100	1000	8000
123	2000	12
159	3000	2200
230	3000	10000
393	1000	6000
516	2000	20000

Where possible perform each of the address translations described below. Assume the kernel takes the necessary steps to set up the MMU before running each process. When the address translation is not possible explain why. In all cases be sure to describe how you arrived at your answer. Assume all addresses are in decimal.

- (2 mark(s))** When PID 516 is running, what is the resulting physical address for virtual address 1999?
  
- (2 mark(s))** When PID 516 is running, what is the resulting physical address for virtual address 2000?
  
- (2 mark(s))** Which virtual address, in which process, corresponds to physical address 2800? Indicate both a virtual address and a PID, or answer “NONE” if no virtual address maps to the given physical address.
  
- (2 mark(s))** Which virtual address, in which process, corresponds to physical address 7100. Indicate both a virtual address and a PID, or answer “NONE” if no virtual address maps to the given physical address.
  
- (2 mark(s))** If process 100 is running and a context switch occurs to process 230, briefly describe the steps the kernel takes to set up the MMU.