

Please staple pages together. CS241 Homework 2. Fall 2008. **netid:** _ _ _ _ _

*Learning Objectives: Review and application of all CS241 material to date.
Preparation for CS241 Midterm exam.*

DUE: MONDAY LECTURE 10/6/08 DCL 1310 @ 10AM (NO LATE HOMEWORKS!)

INSTRUCTIONS:

- 1) Please **TYPE** your homework solutions (unless otherwise mentioned).
Handwritten solutions will **NOT** be accepted.
- 2) You can use separate sheets to type all solutions – for some questions (e.g.,
Question 1), you can print out the question and mark the answers in pen/pencil.
- 3) Please submit printed sheets (double sided preferred).
- 4) Type your netid on **EVERY stapled** sheet.

	QUESTION (each 10pts)						
	1	2	3	4	5	6	TOTAL
SCORE							
GRADER							

NAME LAST, FIRST: _____ SECTION (DATE, TIME) :

Please staple pages together. CS241 Homework 2. Fall 2008. **netid:** _ _ _ _ _

QUESTION 1: (GAME, SET AND) MATCH IT! (This question has two parts)

A famous tennis-playing sister team is trying to learn Operating Systems, and they need your help. Help them out with each of the two parts below. To submit a solution on printed paper, you may either type it out or draw lines with a pen/pencil for this question.

PART I. [7 POINTS]

Connect each of behaviors on the left with *all* matching scheduling algorithms from the right.

- | | |
|--|------------------------------|
| 1. Minimizes Average Waiting Time | a. FIFO |
| 2. May cause Starvation | b. Non-preemptive SJF |
| 3. Suffers from Convoy Effect | c. Round-Robin |
| 4. May cause Deadlocks | d. Preemptive Priority-Based |
| 5. If priority=length of job left, these two algorithms are the same | e. Preemptive SJF |
| 6. For N jobs, total number of context switches may be more than N | |

PART II. [3 POINTS]

Connect each of command names or calls on the left with the *most* appropriate definition on the right. If more than one definition applies, choose the best and most accurate one.

- | | |
|------------|--|
| 1. wait | a. Terminates a UNIX process |
| 2. fork | b. Creates a new POSIX thread |
| 3. exit | c. creates an orphan process |
| 4. getppid | d. Returns own process identifier |
| 5. nice | e. Makes a copy of an existing process |
| | f. Waits for a specific child to return. |
| | g. Returns parent's process identifier |
| | h. Changes process priority |
| | i. Waits for any child process to return |

Please staple pages together. CS241 Homework 2. Fall 2008. **netid:** _ _ _ _ _

QUESTION 2: NOBEL PROCESS HIERARCHY [10 POINTS]

In order to be environment-friendly, an ex-vice-president would like you to build a process tree. In class, you have seen process hierarchies, and examples of a process fan, process chain, etc.

In this problem, instead of a fan or chain, you will write code to create a process hierarchy that looks like a *balanced binary tree* of depth N . That is, each process (except the leaf processes) has two children, and there are a total of $2^N - 1$ processes in the hierarchy. So, for instance with $N=3$, you would have 7 processes over 3 levels. For this solution, the only system call you are allowed to use is `fork()`. Assume `fork` follows the same syntax as described in class. For convenience, you may assume that `fork` always creates a new process (i.e. ignore errors). For every correct tree built in the class, one real tree may be saved.

QUESTION 3: DOES IT SYNC? [10 POINTS] (This question has five parts.)

Your little sibling has started walking and has also scribbled some multi-threaded code. While your parents ensure he doesn't fall down, you are given responsibility to check the code to see if it follows synchronization principles.

Each of the following cases show pseudo-code executed by two different threads, thread 1 and thread 2 around a critical section. (If code of thread 2 is not mentioned, it is the same as thread 1.) By filling in the table below, for each case, specify:

- i. whether mutual exclusion is guaranteed and
- ii. whether progress is guaranteed.

In all cases, assume that all variables are initialized to zero in the beginning and the given code is executed multiple times.

	Case				
Guarantees...	a	b	c	d	e
Mutual Exclusion(Yes/No)					
Progress (Yes/No)					

Thread 1

```
a) if (x is odd) {
    x++;
    execute-critical-section;
}
```

```
b) while (x == 0) {};
   x = 1;
   execute critical section;
   x = 0;
```

```
c) y1 = 1;
   while (y2 == 1) {};
   execute critical section;
   y1 = 0;
```

```
d) while (y2 == 1) {};
   y1 = 1;
   execute critical section;
   y1 = 0;
```

```
e) while (testandset (address-of x)) {};
   execute critical section;
   x = 0;
   (note: use the slides version of the testandset instruction)
```

Thread 2

```
if (x is even) {
    x++;
    execute critical section;
}
```

same as Thread 1

```
y2 = 1;
while (y1 == 1) {};
execute critical section;
y2 = 0;
```

```
while (y1 == 1) {};
y2 = 1;
execute critical section;
y2 = 0;
```

same as Thread 1

QUESTION 4: PING-PONG 1 [10 POINTS] (This question has two parts.)

The processes of the world have assembled in Urbana for Olympics 2008. You are the referee for the ping-pong championship, and you have to iron out one tiny problem before the competition starts.

In the code below, pingthread and pongthread are two separate threads executing their respective procedures ping() and pong(). The code below is intended to cause them to forever take turns, alternately printing "pong" and "ping" to the screen. mythread_stop() and mythread_start () routines are system calls. mythread_stop() blocks the calling thread. mythread_start () makes a specific thread runnable if that thread has previously been stopped, otherwise its behavior is unpredictable, i.e., calling mythread_stop() on a thread that is not stopped will have no future effect. In the code below, 'pingthread' and 'pongthread' are thread ids.

```
/* pingthread runs the function ping() */
...
void ping()
{
    while(true) {
        mythread_stop();
        printf("ping is here\n");
        mythread_start(pongthread);
    }
}

/* pongthread runs the function pong() */
...
void pong()
{
    while(true) {
        printf("pong is here\n");
        mythread_start(pingthread);
        mythread_stop();
    }
}
```

i. **[2 points]** The code shown above exhibits a well-known synchronization flaw. Briefly outline a scenario in which this code would fail, and the outcome of that scenario.

Please staple pages together. CS241 Homework 2. Fall 2008. **netid:** _ _ _ _ _

ii. **[8 points]** Show how to fix the problem by replacing the `mythread_stop` and `start` calls with semaphore `wait()` and `signal()` operations (or `down` and `up`, or `P` and `V`: whichever syntax you prefer).

QUESTION 5: GOOHOO!! INC. 2 [10 POINTS] (This question has two parts.)

You have just graduated and been hired by the hot new company Goohoo!! Inc. Your first job is to review some of their code. Below is their `atomic_swap` procedure. It is intended to work as follows: `atomic_swap` should take two Stack data structures as arguments (warning: this Stack is different from the process stack), pop an item from each, and push each item onto the opposite Stack. If either Stack is empty, the swap should fail and the Stacks should then be left as they were before the swap was attempted. The swap must appear to occur *atomically* – an external thread should not be able to observe that an item has been removed from one Stack but not pushed onto the other one. In addition, the implementation must be concurrent – it must allow multiple swaps between unrelated Stacks to happen in parallel. Finally, the system should never deadlock. Notice that each Stack `*s` has a mutex defined within it (`s->mutex`) and this mutex is initialized to 1.

```
extern Item *pop(Stack *); // pops an item from the Stack
extern void push(Stack *, Item *); // pushes an item onto the Stack
void atomic_swap(Stack *s1, Stack *s2) {
    Item *item1;
    Item *item2; // items being transferred
    down(s1->mutex);
    item1 = pop(s1);
    if(item1 != NULL) {
        down(s2->mutex);
        item2 = pop(s2);
        if(item2 != NULL) {
            push(s2, item1);
            push(s1, item2);
            up(s2->mutex);
            up(s1->mutex);
        }
    }
}
```

i. [3 points] The above implementation has *three* major types of flaws in this code by Goohoo!! Name *all three types* of flaws (there may be more than one flaw of each type). Assume that you have access to pop and push operations on Stacks with the signatures given in the code. You may assume that `s1` and `s2` never refer to the same Stack.

Please staple pages together. CS241 Homework 2. Fall 2008. **netid:** _ _ _ _ _

ii. **[7 points]** To earn your salary (and a promotion!) at Goohoo!!, rewrite the above code so that all these flaws are eliminated (and the original goals are satisfied). Assume that you have access to pop and push operations on Stacks with the signatures given in the code. You may assume that s1 and s2 never refer to the same Stack. You may add additional fields to the Stack data structure, as long as you document clearly what they are.

Please staple pages together. CS241 Homework 2. Fall 2008. **netid:** _ _ _ _ _

QUESTION 6: CHECK YOUR CONCEPTS [10 POINTS]

At the presidential/vice-presidential debate for 2008, your favorite presidential/vice-presidential candidate is given a questionnaire on Operating Systems. Help him/her out by marking either True or False for each of the ten parts below. Each part carries 1 point. (Hint: If you can find a counter example to the statement, mark the answer as false and write the counter example you found.)

- i. When a thread calls *return*, the entire process will terminate _____
- ii. When a thread calls *exit*, the entire process will terminate _____
- iii. When a thread calls *pthread_exit*, the entire process will terminate _____
- iv. A command shell is a descendant of the *init* process in the process hierarchy _____
- v. If two threads call *sem_wait* on the same semaphore before entering a critical section (and call *sem_post* after exiting the critical section), mutual exclusion is guaranteed. _____
- vi. When a process is preempted it goes from running to blocked state. _____
- vii. Round robin scheduling is suitable for interactive systems. _____
- viii. Killing a process also kills all its child processes. _____
- ix. If a process has only one child, then calling *waitpid*(child's pid, ...) is the same as calling *wait*(). _____
- x. *getppid*() returns -1 for an orphaned process. _____