University of Illinois at Urbana-Champaign
Department of Computer Science

First Examination

CS 323 Operating System Design
Spring, 1996

9–9:50 am Monday February 19
1310 Digital Computer Laboratory

Print your name and ID number neatly in the space provided below; print your name at the upper right corner of every page.

Name:	
ID Number:	

This is an **closed book, closed notes** exam. You may not use calculators or similar electronic devices.

Do all parts of all four problems in this booklet. This booklet should include this title page, plus pages 1-6. Do your work inside this booklet, using the backs of pages if needed. All problems are equally weighted, so do not spend too much time on any one question.

Page	Possible	Score	Grader
1	20		
2	10		
3	10		
4	10		
5	10		
6	10		
7	10		
8	20		
Total	100		

1. **Short Answers.**

   (a) Why would a general-purpose operating system (*e.g.* Solaris) use a relatively complex process scheduling algorithm like multilevel feedback queues (MLFQ)? (*7 points*)

   A general-purpose operating system needs to provide good services to both IO bound processes and cpu bound processes. MLFQ can provide quick response to interactive processes because those processes have higher priorities (though shorter cpu slice) and give long cpu slice to cpu intensive process.

   (b) What is the primary disadvantage of test-and-set spin locks (`while(test_and_set()) { no_op();}`) in uniprocessor environments? (*6 points*)

   busy wait, it wastes the cpu time doing nothing. It is possible to give the cpu to other process while it is waiting.

   (c) Why do we use volatile and non-volatile storage devices when the technology to do stable storage is available? (*7 points*)

   Compared with stable storage, volatile is faster, non-volatile is cheaper (and a little faster). Stable storage is usually implemented by replicating non-volatile storage, so stable storage has large overhead.

   Stable storage can't replace them because of the speed and cost, unless reliability is the primary requirement.

2. **Five Philosophers.**

In the Five Philosophers problem, a philosopher alternates between eating and thinking. (S)he first grabs the left chopstick. After (s)he gets the left chopstick, (s)he then grabs the right chopstick. If a chopstick is not available, the philosopher will be blocked. Followed is the pseudo-code. The code could result in deadlock.

```
/* The structure of philosopher i */
repeat
    wait(chopstick[i]);
    wait(chopstick[(i+1) mod 5]);
        ...
    eat
        ...
    signal(chopstick[i]);
    signal(chopstick[(i+1) mod 5]);
        ...
    think
        ...
until false;
```

(a) Modifies the Five Philosophers problem so that it is free of deadlock. In your solution, 1) a philosopher can only pick up the nearby chopsticks, 2) s(he) can eat after two chopsticks have been grabbed and 3) deadlock should not happen at all. Write pseudo-code for the solution. (*10 points*)

```
        /* The structure of philosopher i */
        repeat
    if (i mod 2 == 0)
            {
              wait(chopstick[i]);
              wait(chopstick[(i+1) mod 5]);
            }
            else
            {
              wait(chopstick[(i+1) mod 5]);
              wait(chopstick[i]);
            }


                ...
            eat
                ...
            signal(chopstick[i]);
            signal(chopstick[(i+1) mod 5]);
                ...
            think
                ...
        until false;
```

2. **Five Philosophers, continued.**

(b) Whether or not your solution is fair to each philosopher. Explain it. (*5 points*)

It is not fair to philosopher[4]. Because both philosophers 1 3 will try to grab a chopstick nearby philosopher[4] at the first attempt. I.e., if either philosopher[3] or philosopher[0] is holding a chopstick, philosopher[4] can't get two chopsticks.

(c) Argue that whether or not your solution provide the maximum parallelism. (*5 points*)

Yes. The maximum number of philosophers that can eat at the same time is 2. In the solution above, we can achieve this.

3. **Producer/Consumer.**

   (a) The following producer/consumer sometimes stops working. (Assume that the program is syntactically correct.) (*10 points*)

```
semaphore mutex(1), empty(1), full(0);
int Result_buffer[1]; //the critical region
producer()
{
    while (1)
    {
        P(mutex);
        P(empty);
        Result_buffer[0] = whatever;
        V(mutex);
        V(full);
    }
}
consumer()
{
    while (1)
    {
        P(mutex);
        P(full);
        whatever = Result_buffer[0];
        V(mutex);
        V(empty);
    }
}
```

   i. How to fix it?

   Let's first analyse the problem. If producer runs two consecutive iterations in a roll, it will be stopped at the `P(empty)` *of the second iteration. But the* `P(mutex)` *was executed by producer so that the consumer can not pass the* `P(mutex)`*. The system is deadlocked. On the other hand, if consumer runs first, it locks the mutex and is stopped at* `P(full)`*. Then the producer can't proceed since it will be stopped at* `P(mutex)`*.*

   Switch the `P(mutex) P(empty)` *in producer and switch* `P(mutex) P(full)` *in consumer.*

   ii. Why your fix works?

   1) A producer will be stopped at the `P(empty)` *if it runs two consecutive iterations. A consumer can then pass the* `P(mutex)` *and proceed. 2) If consumer runs first, it is blocked at* `P(full)` *without locking the mutex. The producer can then proceed.*

3. **Producer/Consumer, continued.**

   (b) Write the solution for buffer size = n; (*10 points*)

```
semaphore mutex(1), empty(n), full(0);
int Result_buffer[n];           //the critical region
int write_to = 0, read_from = 0;
producer()
{
    while (1)
    {
        P(empty);
        P(mutex);
        Result_buffer[write_to] = whatever;
        write_to = (write_to + 1) % n;
        V(mutex);
        V(full);
    }
}
consumer()
{
    while (1)
    {
        P(full);
        P(mutex);
        whatever = Result_buffer[read_from];
        read_from = (read_from + 1) % n;
        V(mutex);
        V(empty);
    }
}
```

4. **Critical Region.**

   Two process share one critical region.

   (a) What is the problem with the following code? (Assume the syntex is correct and don't care busy wait.) (*5 points*)

```
repeat
    flag[i]:=true;
    while flag[j] do no-op;
    ...critical section...
    flag[i]:=false;
    ...remainder section...
until false;
```

   If two process get to flag[i]:=true at the same time, no progress.

   (b) An improved version is shown below. Again what is the problem? (*5 points*)

```
repeat
    flag[i]:=true;
    while flag[j]
        do begin
            flag[i]:=false;
            delay random time;
            flag[i]:=true;
        end
    ...critical section...
    flag[i]:=false;
    ...remainder section...
until false;
```

   Indefinite delay could hapen if two process delay the same random time.

4. **Critical Region, continued.**

   (c) based on 1) or 2), give a solution that eliminates the problems in 1) and 2). (Hint, you can add another shared variable) (*10 points*)

```
repeat
    flag[i]:=true;
    turn = j;
    while (flag[j] and turn == j) do no-op;
    ...critical section...
    flag[i]:=false;
    ...remainder section...
until false;
```

5. **Scheduling.**

Three jobs J1 J2 J3 arrived at the same time but are queued in the order: J1, J2, J3 with J1 on top of queue then J2 J3. The job lengths are J1=5, J2=3, J3=4. Calculate the average Turnaround Time, Waiting Time and Response Time for the scheduling algorithms. 1) First Come First Serve 2) Shortest Job First 3) Round-Robin with CPU slice = 1;

Give numerical values.

```
time: 1 2 3 4 5 6 7 8 9 0 1 2

FCFS: 1 1 1 1 1 2 2 2 3 3 3 3

SJF:  2 2 2 3 3 3 3 1 1 1 1 1

RR:   1 2 3 1 2 3 1 2 3 1 3 1
```

(a) average Turnaround Times

i. (*2 points*) FCFS: *(5+8+12)/3 = 8.3*

ii. (*2 points*) SJF: *(3+7+12)/3 = 7.3*

iii. (*3 points*) RR: *(12+8+11)/3 = 10.3*

(b) average Waiting Time

i. (*2 points*) FCFS: *(0+5+8)/3 = 4.3*

ii. (*2 points*) SJF: *(7+0+3)/3 = 3.3*

iii. (*3 points*) RR: *(7+5+7)/3 = 6.3*

(c) average Response Time

i. (*2 points*) FCFS: *(0+5+8)/3 = 4.3*

ii. (*2 points*) SJF: *(7+0+3)/3 = 3.3*

iii. (*2 points*) RR: *(0+1+2)/3 = 1*