University of California, Berkeley
College of Engineering
Computer Science Division — EECS

Fall 2006 John Kubiatowicz

Midterm I

October 11th, 2006

CS162: Operating Systems and Systems Programming

Your Name:	
SID Number:	
Discussion Section:	

General Information:
This is a **closed book** exam. You are allowed 1 page of **hand-written** notes (both sides). You have 3 hours to complete as much of the exam as possible. Make sure to read all of the questions first, as some of the questions are substantially more time consuming.

Write all of your answers directly on this paper. *Make your answers as concise as possible.* On programming questions, we will be looking for performance as well as correctness, so think through your answers carefully. If there is something about the questions that you believe is open to interpretation, please ask us about it!

Problem	**Possible**	**Score**
1	20	
2	24	
3	19	
4	20	
5	17	
Total	100	

[This page left for π]

3.141592653589793238462643383279502884197169399375105820974944

Problem 1: Short Answer [20pts]

Problem 1a[2pts]: What is a virtual machine?

Problem 1b[2pts]: Does a cyclic dependency always lead to deadlock? Why or why not?

Problem 1c[2pts]: What are exceptions? Name two different types of exceptions and give an example of each type:

Problem 1d[2pts]: List two reasons why overuse of threads is bad (i.e. using too many threads for different tasks). Be explicit in your answers.

Problem 1e[3pts]: For each of the following thread state transitions, say whether the transition is legal *and* how the transition occurs or why itcannot. Assume Mesa-style monitors.

1). Change from thread state BLOCKED to thread state RUNNING

2). Change from thread state RUNNING to thread state BLOCKED

3). Change from thread state RUNNABLE to thread state BLOCKED

Problem 1f[4pts]: Consider the Dining Lawyers problem, in which a set of lawyers sit around a table with one chopstick between each of them. Let the lawyers be numbered from 0 to n-1 and be represented by separate threads. Each lawyer executes `Dine(i)`, where "`i`" is the lawyer's number. Assume that there is an array of semaphores, `Chop[i]` that represents the chopstick to the left of lawyer `i.` These semaphores are initialized to 1.

```
void Dine(int i) {
   Chop[i].P();            /* Grab left chopstick */
   Chop[(i+1)%n].P();      /* Grab right chopstick */
   EatAsMuchAsYouCan();
   Chop[i].V();            /* Release left chopstick */
   Chop[(i+1)%n].V();      /* Release right chopstick */
}
```

This solution can deadlock. Assume that it does. List the four conditions of deadlock and explain why each of them is satisfied during the deadlock:

Problem 1g[3pt]: Pick one of the above four conditions and rewrite the code to eliminate it. Identify the condition you chose carefully and explain why your code doesn't deadlock:

Problem 1h[2pts]: The Banker's algorithm is said to keep the system in a "safe" state. Describe what a "safe" state is and explain how the Banker's algorithm keeps the system in a safe state. Keep your answer *short.*

EXTRA CREDIT

Problem 1i[2pts]: Describe what "core" memory is and how it looks.

Problem 2: Synchronization [24pts]

Assume that you are programming a multiprocessor system using threads. In class, we talked about two different synchronization primitives: Semaphores and Monitors.

The interface for a Semaphore is as follows:

```
   public class Semaphore {
      public Semaphore(int initialValue) {
         /* Create and return a semaphore with initial value: initialValue */
         …
      }
      public P() {
         /* Call P() on the semaphore */
         …
      }
      public V() {
         /* Call V() on the semaphore */
      }
   }
```

As we mentioned in class, a Monitor consists of a Lock and one or more Condition Variables. The interfaces for these two types of objects are as follows:

```
   public class Lock {              public class CondVar {
      public Lock() {                  public CondVar(Lock lock) {
         /* Create new Lock */            /* Creates a condition variable
         …                                   associated with Lock lock. */
      }                                   …
                                       }
      public void Acquire() {          public void Wait() {
         /* Acquire Lock */               /* Block on condition variable *’/
         …                                …
      }                                }
      public void Release() {          public void Signal() {
         /* Release Lock */               /* Wake one thread (if it exists) */
         …                                …
      }                                }
   }                                   public void Broadcast() {
                                          /* Wake up all threads waiting on cv*/
                                          …
                                       }
                                    }
```

Monitors and Semaphores can be used for a variety of things. In fact, each can be implemented with the other. In this problem, we will show their equivalence.

Problem 2a[2pts]: What is the difference between Mesa and Hoare scheduling for monitors?

Problem 2b[5pts]: Show how to implement the Semaphore class using Monitors (i.e. the Lock and CondVar classes). Make sure to implement all three methods, Semaphore(), P(), and V(). None of the methods should require more than five lines. Assume that Monitors are Mesa scheduled.

```
public class Semaphore {



   public Semaphore(int initialValue) {



   }
   public P() {



   }
   public V() {



   }
}
```

Problem 2c[3pts]: Show how to implement the Lock class using Semaphores. Make sure to implement the Lock(), Acquire(), and Release() methods. None of the methods should require more than five lines.

```
public class Lock {


   public Lock() {



   }
   public void Acquire() {



   }
   public void Release() {



   }
}
```

Problem 2d[2pts]: Explain the difference in behavior between Semaphore.V() and CondVar.Signal() when no threads are waiting in the corresponding semaphore or condition variable:

Problem 2e[12pts]: Show how to implement the Condition Variable class using Semaphores (and your Lock class from 2c). Assume that you are providing Mesa scheduling. Be very careful to consider the semantics of CondVar.Signal() as discussed in (2d). *Hint: the Semaphore interface does not allow querying of the size of its waiting queue; you may need to track this yourself.* None of the methods should require more than five lines.

```
public class CondVar {




   public CondVar(Lock lock) {





   }
   public void Wait() {





   }
   public void Signal() {





   }
   public void Broadcast() {





   }
}
```

Problem 3: Critical Sections [19 pts]

For each of the following techniques for synchronization, assume that there are two threads competing to execute a critical section. Further, assume that:

1. A critical section is "protected" if only one thread can enter the critical section at a time.
2. The synchronization is "fair" if, when each thread attempts to acquire the critical section repeadedly, then each thread will enter the critical section about the same number of times.

Note: Assume that all flags start out "false". Also assume that store is atomic.

Synchronization technique #1: Suppose each thread does the following:

```
1.  while (flag == true)
2.      do nothing;
3.  flag = true;
4.  Execute Critical Section;
5.  flag = false;
```

Problem 3a[2pts]: Will this protect the critical section? If "yes", explain why. If "no", give an example interleaving that will fail to protect the critical section.

Problem 3b[2pts]: Assume this code protects the critical section. Is this code "fair"? Explain.

Synchronization technique #2: Suppose we have different code for each thread:

```
    THREAD A                         THREAD B
A1. flag_A = true;               B1. flag_B = true;
A2. while (flag_B == true)       B2. if (flag_A == false)
A3.     do nothing;              B3.     Execute Critical Section;
A4. Execute Critical Section;    B4. flag_B = false;
A5. flag_A = false;
```

Problem 3c[2pts]: Will this protect the critical section? If "yes", explain why. If "no", give an example interleaving that will fail to protect the critical section.

Problem 3d[2pts]: Assume this code protects the critical section. Is this code "fair"? Explain.

Synchronization technique #3: Suppose each thread does the following:

```
1.  while (TestAndSet(flag) == false)
2.     do nothing;
3.  Execute Critical Section;
4.  flag = false;
```

Problem 3e[3pts]: Will this protect the critical section? If "yes", explain why. If "no", explain *and explain how to fix it.*

Problem 3f[2pts]: Assume the above code (or your fixed version). Will this code be "fair"? Explain.

Synchronization technique #4: Suppose we have different code for each thread:

```
    THREAD A                         THREAD B
A1. flag_A = true;               B1. flag_B = true;
A2. while (flag_B == true)       B2. while (flag_A == true)
A3.    do nothing;               B3.    do nothing;
A4. Execute Critical Section;    B4. Execute Critical Section;
A5. flag_A = false;              B5. flag_B = false;
```

Problem 3g[3pts]: Will this *protect* the critical section? If "yes", explain why. If "no", explain *and explain how to fix it.* Note that this question is only about protecting the critical section!

Problem 3h[3pts]: Explain why this code (or your fixed version) would not be a particularly good mechanism for synchronizing threads A and B. (hint: imagine that threads A and B repeatedly try to acquire the critical section). After describing the problem, explain how to fix the problem by replacing the "do nothing" with no more than three lines inside each while loop above.

Problem 4: Scheduling [20pts]

Problem 4a[2pts]:
Describe one way to predict the burst runtime (time between I/O operations) for a thread.

Problem 4b[3pts]:
What is priority inversion? Explain how a priority scheduler could be modified to avoid priority inversion.

Problem 4c[3pts]:
Explain what a multi-level feedback scheduler is and why it approximates SRTF.

Problem 4d[2pts]: Explain how to fool the multi-level feedback scheduler's heuristics into giving a long-running task more CPU cycles.

Problem 4e[5pts]:
Here is a table of processes and their associated arrival and running times.

Process ID	Arrival Time	CPU Running Time
Process 1	0	2
Process 2	1	6
Process 3	4	1
Process 4	7	4
Process 5	8	3

Show the scheduling order for these processes under 3 policies: First Come First Serve (FCFS), Shortest-Remaining-Time-First (SRTF), Round-Robin (RR) with timeslice quantum = 1. *Assume that context switch overhead is 0 and that new processes are added to the* ***head*** *of the queue except for FCFS, where they are added to the tail.*

Time Slot	FCFS	SRTF	RR
0			
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

Problem 4f[3pts]:
For each process in each schedule above, indicate the queue wait time and completion time (otherwise known as turnaround time, TRT). Note that wait time is the total time spend waiting in queue (all the time in which the task is not running), while TRT is the total time from when the process arrives in the queue until it is completed.

Scheduler	Process 1	Process 2	Process 3	Process 4	Process 5
FCFS wait					
FCFS TRT					
SRTF wait					
SRTF TRT					
RR wait					
RR TRT					

Problem 4g[2pts]:
Assume that we could have an oracle perform the best possible scheduling to reduce average wait time. What would be the optimal average wait time, and which of the above three schedulers would come closest to optimal? Explain.

[This page intentionally left blank]

Problem 5: Address Translation [17 pts]

Problem 5a[2 pts]:
Explain how Address Translation can protect processes from one another.

Problem 5b[3pts]:
Suppose we have a memory system with 32-bit virtual addresses and 4 KB pages. If the page table is full (with 2^{20} pages), show that a 20-level page table consumes approximately twice the space of a single level page table. *Hint: try drawing it out and summing a series.*

Problem 5c[2pts]:
Problem (5b) showed that, in a full page table, increasing the number of levels of indirection increases the page table size. Show that this is not necessarily true for a sparse page table (*i.e.* one in which not all entries are in use).

Consider a multi-level memory management scheme using the following format for virtual addresses:

Virtual seg # (4 bits)	Virtual Page # (8 bits)	Offset (8 bits)

Virtual addresses are translated into physical addresses of the following form:

Physical Page # (8 bits)	Offset (8 bits)

Problem 5d[4pts]: For the following Virtual Addresses, translate them into Physical Addresses. Use the Segment Table and Physical Memory table given on the next page. Segment entries point to page tables in memory. A page table consists of a series of *16 bit* page table entries (PTEs). The format of a PTE is given on the next page. Briefly, the first byte of the PTE is an 8-bit physical page #, and the second byte is an 8-bit flags field with one of the following values:

0x00 (Invalid), 0x06 (Valid, RO), 0x07 (Valid, R/W).

If there is an error during translation, make sure to say what the error is. Errors can be "**bad segment error**" (undefined or invalid segment), "**segment overflow error**" (address outside range of segment), or "**access violation error**" (page invalid, or attempt to write a read only (RO) page). Two answers are given:

Virtual Addr	Physical Addr
0x10123	**0x4123**
0x33423	**Segment overflow**
0x20456	

Virtual Addr	Physical Addr
0x31056	
0x10400	
0x00278	

Problem 5e[6pts]: Consider the same multi-level memory management scheme. Please return the results from the following load/store instructions. Addresses are virtual. The return value for load is an 8-bit data value or an error, while the return value for a store is either "**ok**" or an error. For errors, please specify which type of error (from the above set). Two answers are given:

Instruction	Result
Load [0x30115]	**0x57**
Store [0x30116]	**Access violation**
Load [0x51015]	
Load [0x00115]	

Instruction	Result
Store [0x00310]	
Load [0x31202]	
Store [0x10231]	
Load [0x12345]	

Virtual Address Format

Virtual seg # (4 bits)	Virtual Page # (8 bits)	Offset (8 bits)

Segment Table (Max Segment=3)

Seg #	Page Table Base	Max Page Entries	Segment State
0	0x2030	0x20	Valid
1	0x1020	0x10	Valid
2	0x3110	0x40	Invalid
3	0x4000	0x20	Valid

Page Table Entry

First Byte	Second Byte
Physical Page Number	0x00 = Invalid 0x06 = Valid, RO 0x07 = Valid, R/W

Physical Memory

Address	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
0x0000	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
0x0010	1E	1F	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D
….																
0x1010	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
0x1020	40	07	41	06	30	06	31	07	00	07	00	00	00	00	00	00
….																
0x2000	02	20	03	30	04	40	05	50	06	60	07	70	08	80	09	90
0x2010	0A	A0	0B	B0	0C	C0	0D	D0	0E	E0	0F	F0	10	01	11	11
0x2020	12	21	13	31	14	41	15	51	16	61	17	71	18	81	19	91
0x2030	10	06	11	00	12	07	40	07	41	07	00	00	00	00	00	00
….																
0x30F0	00	11	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF
0x3100	01	12	23	34	45	56	67	78	89	9A	AB	BC	CD	DE	EF	00
0x3110	02	13	24	35	46	57	68	79	8A	9B	AC	BD	CE	DF	F0	01
0x3120	03	06	25	36	47	58	69	7A	8B	9C	AD	BE	CF	E0	F1	02
0x3130	04	15	26	37	48	59	70	7B	8C	9D	AE	BF	D0	E1	F2	03
….																
0x4000	30	00	31	06	32	07	33	07	34	06	35	00	43	38	32	79
0x4010	50	28	84	19	71	69	39	93	75	10	58	20	97	49	44	59
0x4020	23	07	20	07	00	06	62	08	99	86	28	03	48	25	34	21

[This page intentionally left blank]

[Scratch Sheet (Feel free to remove)]