Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.894 Distributed Operating Systems Engineering: Fall 2000

Midterm Solutions

This midterm has 7 questions. In order to receive credit you must fill in the blank(s) or mark the correct answer or answers for each question. You have 80 minutes to answer this exam.

Write your name on this cover sheet AND at the bottom of each page of this booklet.

Some questions may be much harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

THIS IS AN OPEN BOOK, OPEN NOTES MIDTERM.

Do not write in the boxes below

1 (xx/20)	2 (xx/20)	3-4 (xx/20)	5-6 (xx/20)	7 (xx/20)	Total (xx/100)

Name: Solutions (Mean: 88.28, Median: 89)

I Virtual memory primitives for user programs

1. [10 points]: The paper "Virtual memory primitives for user programs" by Appel and Li describes a concurrent garbage collector (Section 3, page 97). Explain why and how MAP2 is used in this application. Be concise and brief.

Answer:
In the concurrent garbage collector, MAP2 is used to assign different levels of protection to the same page in a address space. In particular, the from-space is accessible to the collector but not the mutator that is running at the same time.

II Scheduler activations

A multi-threaded process called P1 is running on a multi-CPU machine whose kernel uses scheduler activations. At the moment P1 is using two CPUs, running one thread on each. Call the two CPUs CPU1 and CPU2, and the two threads T1 and T2. In addition, process P2 is running thread T3 on CPU3. Both P1 and P2 have other threads ready to run. Figure 1 summarizes the scenario.

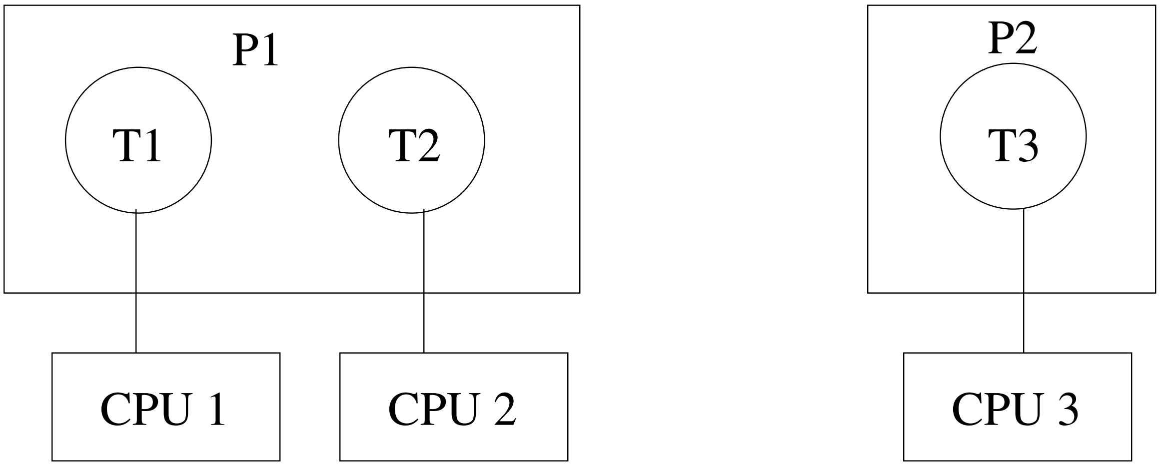

Figure 1: The scenario

Suppose that the kernel scheduler would like to take CPU2 away from P1 and give it to P2. What sequence of upcalls does the kernel use to do this? List the upcalls in the order that you think they'll occur, and for each one, specify:

- The CPU that the upcall executes on.
- The process that receives the upcall.
- The events and event arguments delivered by the upcall.
- The actions the user thread scheduler takes during the upcall, in a nutshell.

You'll find a list of upcall events in Table 2 of the paper ("Scheduler activations: Effective kernel support for the user-level management of parallelism" by Anderson et al.).

2. **[20 points]:** Fill in this list of upcalls; use as many as you need.

(List for each upcall the information requested above.)

- 1.
- Delivered on CPU: **CPU2**
- Process: **P2**
- Events plus args: **Add this processor (CPU2)**
- Actions by user-level scheduler: **Schedule a runnable thread from its ready-list to run on CPU2**

- 2.
- Delivered on CPU: **CPU1**
- Process: **P1**
- Events plus args:
  - **A. Preempted activation associated with T1 and its saved state**
  - **B. Preempted activation associated with T2 and its saved state**
- Actions by user-level scheduler: **Put T1 and T2 to its ready-list, schedule some runnable thread to run on the remaining processor (CPU1).**

- 3.
- Delivered on CPU:
- Process:
- Events plus args:
- Actions by user-level scheduler:

- 4.
- etc.

III Eliminating receive livelock

Mogul and Ramakrishnan in “Eliminating Receive Lock in an an interrupt-driven kernel” describe how to eliminate livelocks. The following pseudo code corresponds to the polling thread they describe:

```
poll ()
{
  int io;

  while (1) {
   foreach (device) {
     if (clock_interrupt) {
        clock++;
        io = 1;
        for (i = 0; i < nthreads; i++) {
          if (thread[i].queue is full) {  // should thread i run?
            io = 0;
            break;  // let the scheduler schedule a user-level process
          }
        }
     } else {  // network device
       for (maximum of n pkts on device) {
         switch (type) {
         case RECEIVE:
           do device-specific stuff (e.g., copy packet from card);
           process_receiveintr(pkt);
           break;
         case SENDCOMPLETE:
           do device-specific stuff;
           process_sendcomplete(pkt);
           break;
         }
       }
     }
   }
   if (io) enable_interrupts(); // enable network interrupts
   wait (polling);
  }
}
```

This fragment of code is identical in function—the code is slightly improved for clarity—to the pseudocode that was handed out in lecture, which described a small interrupt-driven kernel. The complete unmodified handout is appended to the end of this midterm.

Name: Solutions

3. [10 points]:. What problem does the statement

```
for (maximum of n pkts on device)
```

in the function poll() address? Be concise and brief.

Answer: Firstly, this restricts the maximum number of packets to be processed from a single network interface during each round-robin polling. This is used to achieve fairly allocation of resources among devices. Secondly, this enables batching of multiple packets (n¿1) from a single device to improve performance.

4. [10 points]:. What problem does the following statements

```
if (thread[i].queue is full)  // should thread i run?
  io = 0;
  break;   // let the scheduler schedule a user-level process
}
```

in the function poll() address? (If you omit the statement, what problems can occur?) Be concise and brief.

Answer: This code ensures that other threads (e.g., user programs) get time to run so that they can process the packets that are queued for them. (Notice that the above statement disabled further interrupt.)

IV Fast Mutual Exclusion for Uniprocessors

The Intel x86 CPU hardware provides an atomic exchange instruction called xchg that acts like this:

```
void
xchg(int *p1, int *p2) {
  // Lock out other memory accesses.
  int tmp = *p1;
  *p1 = *p2;
  *p2 = tmp;
  // Unlock memory system.
}
```

The hardware arranges for no other memory accesses to occur between the Lock and Unlock comments. This means that interrupts are deferred and memory references from other CPUs are forced to wait.

Suppose someone tried to implement xchg as a Restartable Atomic Sequence by turning the Lock/Unlock comments into atomic sequence delimiters, in the style of Figure 3 in the Fast Mutual Exclusion paper:

```
void
xchg_RAS(int *p1, int *p2) {
  BEGIN RESTARTABLE ATOMIC SEQUENCE
  int tmp = *p1;  // 1.
  *p1 = *p2;      // 2.
  *p2 = tmp;      // 3.
  END RESTARTABLE ATOMIC SEQUENCE
}
```

This code turns out not to work. For example, if there are two variables

```
int a = 1;
int b = 2;
```

and a process executes

```
xchg_RAS(&a, &b);
```

then we expect that `a` and `b` end up with values 2 and 1, respectively, as long as no other process modifies `a` or `b`. This won't always happen with xchg_RAS.

5. [10 points]: List a sequence of events in which xchg_RAS fails to work correctly. You can refer to the instructions by their number (as listed in the comments).

Answer:

```
a =1;
b =2;
xchg_RAS called
#1 executed
#2 executed (a=2)
******interrupted!*********
RAS restarted
#1 executed (tmp = 2)
#2 executed (a = 2)
#3 executed (b = 2)
```

At the end of RAS, we see that a and b are both assigned 2, which is not our desired output.

Suppose you've implemented an atomic linked list insert using the x86 xchg instruction:

```
struct List {
  int value;
  struct List *next;
};

insert(int value, struct List **head_addr) {
  struct List *l = new List;
  l->value = value;
  l->next = l;
  xchg((int *) head_addr, (int *) &(l->next));
}
```

Here's an example of code that uses insert() to produce the list 0, 1, 2:

```
  struct List *head = 0;

  insert(2, &head);
  insert(1, &head);
  insert(0, &head);
```

Of course, this insert() won't work on a CPU that doesn't support the atomic xchg instruction.

(Turn the page...)

6. [10 points]: Rewrite insert() to use a Restartable Atomic Sequence so that it can be used on any uni-processor.

Answer:

```
insert(int value, struct List **head_addr) {
  struct List *l = new List;
  l->value = value;

  BEGIN RESTARTABLE ATOMIC SEQUENCE
  l->next = *head_addr;
  *head_addr = l;
  END RESTARTABLE ATOMIC SEQUENCE

}
```

V Remote Procedure Call

Here's an implementation of a bank account server written with Java Remote Method Invocation (RMI), a system very similar to Network Objects.

First, the abstract interface to the server's methods:

```
public interface Account extends Remote {
  public void deposit(float amount) throws RemoteException;
}
```

Second, an implementation of the server methods:

```
public class AccountImplementation implements Account {
  private float theBalance = 0;

  public void deposit(float amount) throws RemoteException {
    theBalance += amount;
  }
}
```

Finally, a sample client program:

```
public class AccountClient {
  public static void main(String argv[]) {
    Account a = look up an Account server;

    try {
      a.deposit(100);
      System.out.println("OK");
    } catch (RemoteException e) {
      System.out.println("Oops");
    }
  }
}
```

Just as in the Network Objects paper, when the client program runs, the Account object it uses is actually a stub object. When the client makes a call to an Account method, the stub object marshalls the arguments, sends them over the network to the server, waits for and unmarshalls the reply (if there are return values in the reply), and returns the values (if any) to the client code. If the stub sees no reply after waiting for 10 seconds, it raises a RemoteException.

You can assume that the network never duplicates packets, that the client and server don't crash, and that the client stub software sends the marshalled call to the server exactly once. However, sometimes the network drops packets, and sometimes it delays them for more than 10 seconds.

Name: Solutions

7. [20 points]: Circle the outcomes that are possible after one execution of the client:

A. The client prints OK and the balance on the server is 0.

B. The client prints OK and the balance on the server is 100.

C. The client prints OK and the balance on the server is 200.

D. The client prints Oops and the balance on the server is 0.

E. The client prints Oops and the balance on the server is 100.

F. The client prints Oops and the balance on the server is 200.

End of Midterm