Suggested Solutions
(Midterm Exam October 27, 2005)

1 Short Questions (4 points)

Answer the following questions (True or False). Use exactly **one** sentence to describe why you choose your answer. Without the reasoning, you will not get any points.

(a) (1 point) Can Monitors deadlock?

True. A program inside a monitor can hold a resource and request for another while another program can do similarly to form a circular chain.

(b) (1 point) Intel Pentium processors have special instructions such as MMX and SSE for multimedia processing. Are the registers for such instructions part of the context of a process?

True. This is because any programs can use such instructions at any time.

(c) (1 point) Are the registers for I/O devices part of the context of a process?

False. I/O devices are accessed via device driver calls in the kernel.

(d) (1 point) As the ratio of time slice to job length decreases, round-robin CPU scheduling becomes equivalent to first-come-first-serve.

False. As the time-slice length goes to zero (the ratio decreases) round-robin looks increasingly different from FCFS.

2 Mutual Exclusion (6 points)

(a) (2 points) Many operating systems designed to run only on uniproccessors use disabling of interrupts to create critical sections in the code. Explain how this interrupt disabling prevents multiple threads from entering the critical section.

Disabling interrupts prevents interrupts (including the timer interrupt) from invoking the operating system scheduler that preempting the job during the critical section. So, the process will execute the critical section to completion before another job will be able to get the CPU and enter the critical section.

(b) (2 points) Explain why disabling of interrupts to implement critical sections will not work on a multiprocessor.

Disabling interrupts only prevent concurrent execution on the same processor; it does not prevent another processor on a multiprocessor from entering critical sections.

(c) (2 points) When running on a multiprocessor, would it ever make sense to have a spin lock also disable interrupts for the duration of a critical section? Explain your answer.

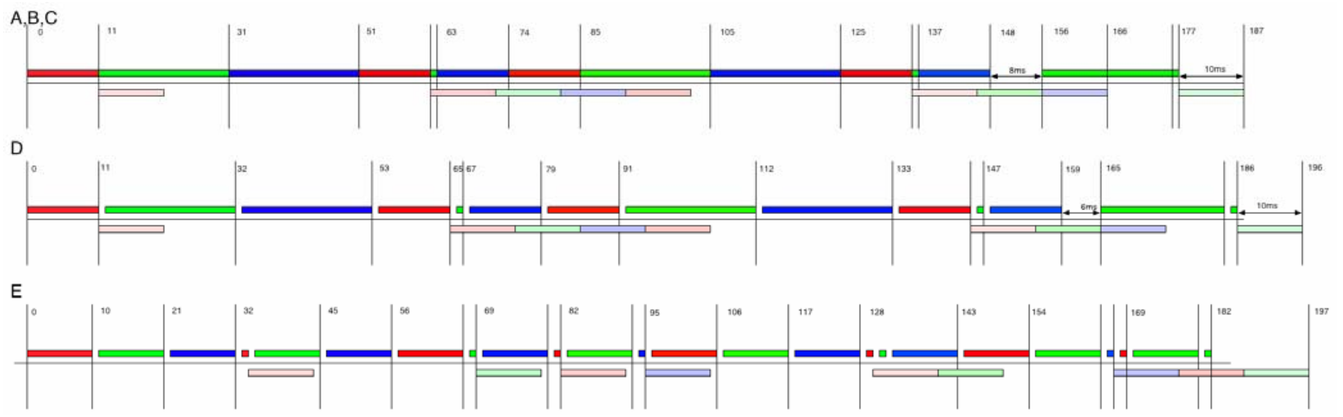
Yes. Disabling interrupts in conjunction with locks can be useful. If you don't disable interrupts and take an interrupt with a spin local held, you can have a deadlock if the interrupt tries to acquire the same spin lock.

3 CPU Scheduling (10 points)

Suppose you have the 3 processes running on one processor. Each process runs K iterations of computations and disk I/Os. In each iteration, a process runs its computation and a disk I/O in a serial fashion, but the disk I/O can overlap with the computation time of another process. The following table lists the computation period as CPU time and the disk I/O operation as I/O time per iteration for each process.

	CPU time	I/O time	Iterations (K)
Process 1	11ms	10ms	4
Process 2	21ms	10ms	3
Process 3	31ms	10ms	2

Suppose a system uses round-robin preemptive scheduling with a time slice quantum of 20ms and the order of the processes initially in the ready queue is process 1, process 2, and process 3. Assume that context-switch overhead is 0. Please answer the following questions:



A) Average turnaround time:
   Process 1 146ms
   Process 2 187 ms
   Process 3 166 ms
   Average 166.33 ms
B) CPU utilization = 169/187
C) Disk utilization = 90/187
D) With a 1ms context switch, Disk utilization = 90/196
E) With a 10ms quanta and 1ms context switch, Disk utilization = 90/197

4 Message Passing (10 points)

Suppose that **`mbox_t`** is an opaque data type for a mailbox and you have following mailbox primitives:

- **`mbox_t Open( int key );`**
  Open a mailbox with a key. The same key will return the same mailbox.
- **`close( mbox_t mbox );`**
  Close a mailbox `mbox`.
- **`send( mbox_t mbox, char buffer[], int size );`**
  Send a message to mailbox `mbox`. The message is in buffer in `size` bytes.
- **`int recv( mbox_t mbox, char * buffer);`**
  Receive a message from mailbox `mbox` and put the result in buffer and return the size of the message.

Use these primitives to solve the bounded-buffer problem for multiple producers and multiple consumers. Your solution should not use any other synchronization primitives and you don't need to worry about memory allocation issues. You can assume that the mailbox can buffer unlimited number of messages.

Use two mailboxes for this purpose: mboxCredits and mboxData

```
mboxCredits = open( 1 );
mboxData = open( 2 );

…
Producer:

  int size;

while (1) {
  size = recv( mboxCredits, credit );

  produce an item;

  send( mboxData, item, sizeof(item) );
}

Consumer:
  int I, size;

  for ( i=0; i<N; i++ )
    send( mboxCredits, credit, 1 );

  while ( 1 ) {
    size = recv( mboxData, &item );

    consume item;

    send( mboxCredits, credit, 1);
  }
```

5 Synchronization (10 points)

As mentioned in class, a *barrier* is a convenient primitive to synchronize multiple threads in a parallel program. When a thread reaches the barrier, it will check to see if other threads have arrived at the barrier. If one or more threads have not arrived, the thread will wait. When all threads reach the barrier, they can begin their execution on the next phase of the computation. An example of using a barrier is as follows:

```
while (true) {
  // Compute stuff;
  EnterBarrier();
  // Use other threads' results;
}
```

There are several issues that you need to consider. First, there is no master thread that controls the threads, waits for each of them to reach the barrier, and then tells them to restart. Instead, the threads must determine themselves when they should wait or proceed. Second, the barrier mechanism should work for many dynamic programs. The number of threads during the lifetime of the parallel program is unknown in advance, since a thread can spawn another thread, which will start in the same program stage as the thread that created it. Third, a thread may end before the barrier. In all cases, all threads must wait at the barrier for all other threads before anyone is allowed to proceed.

Your job is to design and implement the data structure and primitive operations for barrier. Your solution must support creation of a new thread (an additional thread that needs to synchronize), termination of a thread (one less thread that needs to synchronize), waiting when a thread reaches the barrier early, and releasing waiting threads when the last thread reaches the barrier.

You should first define the data structure of barrier and then show how to implement the following barrier primitives with Mesa-style monitor pseudo code:

- **`barrier = InitBarrier();`**
  Initialize a barrier.
- **`ThreadCreated( barrier );`**
  This primitive will be called when a thread is created.
- **`ThreadEnded( barrier );`**
  This primitive will be called when a thread terminates.
- **`EnterBarrier( barrier );`**
  Threads will call this primitive to enter a barrier.

Your solution should never busy-wait. You should use only Mesa-style monitor primitives and NOT use any other synchronization primitives in your implementation. Think carefully about efficiency and avoid unnecessary looping.

```
struct barrier_t {
  int n;            // number of threads to wait for
  int counter;      // number of threads at the barrier
  lock_t lock;
  lond_t cond;
}

barrier_t InitBarrier(void) {
  barrier_t * b;

  b = malloc(sizeof(barrier_t));
  b->n = 0;
  b->counter = 0;
  InitMutex(b->lock);
  InitCond(b->cond);
  return b;
}

ThreadCreated( barrier_t * b ) {
  Acquire( b->lock );
  b->n++;
  Release( b->lock );
}

ThreadEnded( barrier_t * b ) {
  Acquire( b->lock );
  b->n--;
  if ( b->n == b->counter ) {
    b->counter = 0;
    Broadcast( b->cond );
  }
  Release( b->lock );
}

EnterBarrier( barrier_t * b ) {
  Acquire( b->lock );
  b->counter++;
  if ( b->n > b->counter )
    Wait( b->lock, b->cond );
  else {
    b->counter = 0;
    Broadcast( b->cond );
  }
  Release( b->lock );
}
```