Computer Science 15-412: Operating Systems
Midterm Exam, Spring 2003

1. This is a closed-book, closed-notes, in-class exam. You may not use any reference materials during the exam.
2. You must complete the exam by the end of the class period.
3. Answer all questions. The weight of each question is indicated on the exam.
4. Please be concise in your answers. You will receive partial credit for partially correct answers, but truly extraneous remarks may count against your grade.
5. Be sure to put your name and Andrew ID below *and* your Andrew ID at the top of *each* following page.

Andrew Username	
Full Name	

Question	Max	Points	Grader
1.	20		
2.	20		
3.	20		
4.	20		
5.	20		

1. Are keyboard interrupts really necessary? Outline an approach, suitable for a preemptively scheduled multi-user, multi-process operating system, for ensuring timely delivery of keypress events from the keyboard controller's I/O port to waiting processes without the use of keyboard interrupts. Assume that a person pressing a key needs to see the response on the screen in 50-60 milliseconds, that reasonable scheduling quantum values range from 5 milliseconds to 20 milliseconds, and that your solution must work even if there are "many" (e.g., 100) processes running and all of them choose to call a blocking getchar() system call inside a tight loop.

2. (a) Imagine an OS kernel similar to the one you are implementing for P3. In particular, assume that the OS does not page or swap memory pages to disk. The kernel supports a system call getfreeframecount() which returns the number of memory frames which are currently free (i.e., frames which are not reserved for kernel memory and could be assigned to user processes).

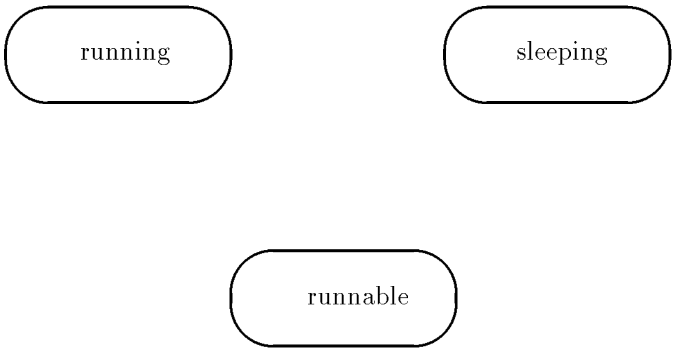
   Imagine a situation where exactly two processes are running, parent and child. The parent calls getfreeframecount(), records the value in a variable called fc1, sends a "please exit now" message to the child process, wait()s for the child process to exit(), and then calls getfreeframecount() again, recording the value in a variable called fc2.

   Immediately before the child process exited, its memory regions had the following sizes:

128K	code
64K	data
16K	bss
32K	heap
16K	stack

   The parent process observes that $(fc2 - fc1)$, when converted from frames to bytes, works out to be 50K. Explain how this could be a correct outcome (i.e., not due to a bug).

(b) Here are graph nodes for three states a process can be in. Draw directed arcs (i.e., arrows) between the nodes to indicate all legal transitions. Label each arc with a 1-word summary term and then, below the figure, provide a brief explanation (1 or 2 sentences should suffice) of what you meant by each summary term. For any node pair (i,j) you do not need to draw more than a single arc from i to j, i.e., you do not need to draw more than six arcs (and you may need to draw fewer). Another way to say this is that, for each transition, you need provide us with only one explanation of why such a transition happens.



(c) Imagine you and your programming partner are both logged in to UNIX3.ANDREW.CMU.EDU when you are seized by an impulse to disrupt your partner's calm, cool demeanor. Your first idea is to write a C program which will write zeroes throughout the address space of your partner's shell (command interpreter, i.e., bash, tcsh, etc.). To your sadness you discover that the operating system has placed an insurmountable hardware barrier in your path. Explain why you have no hope of accessing memory belonging to your partner's processes.

3. On shared-memory multi-processor machines, mutexes are frequently implemented in terms of two special instructions, called load-linked and store-conditional. The reason for this is that the instructions commonly used on single-processor systems, such as exchange(), compare-and-swap(), or test-and-set(), require bus locking, which can be very expensive on multiprocessor systems. Instead of locking the system memory bus, load-linked and store-conditional indicate that the processor should use a special signalling protocol on the memory bus and also closely monitor how other processors use the memory bus.

   Load-linked(address1) returns the contents of the designated memory address. Store-conditional(address2, value) attempts to perform the designated memory-store operation (i.e., "*address2 = value;") and returns a boolean value indicating whether or not the memory-store operation succeeded. The operation will typically succeed, but may fail for one of the following reasons.

   (a) Within a process, each load-linked() operation cancels any previous outstanding load-linked() operation. That is, the following sequence will always fail:
       v1 = load-linked(a1);
       v2 = load-linked(a2);
       ok = store-conditional(a1, 99); /* a2 has cancelled a1, ok == false */

   (b) Any trap, exception, or interrupt which "distracts" the processor which was running our process from the time it began executing the load-linked() instruction until the time the matching store-conditional() completes causes the store-conditional() to fail.

   (c) If, from the time when our processor begins to execute load-linked(a1) on our behalf until the time when our processor begins to execute store-conditional(a1, v), some other process or thread executing on another processor has executed load-linked(a3) and the virtual memory system maps our a1 and that process's/thread's a3 to the same physical memory address, our store-conditional() will fail.

   Please write pseudo-code for the following thread library functions, in terms of load-linked and store-conditional. You may assume that the result of calling mutex_lock() or mutex_unlock() on a mutex before mutex_init() is undefined.

   In addition to load-linked() and store-conditional(), you may use standard C variables and short scraps of code (but you may not call arbitrary library routines). You may also use the following version of the yield() system call: "void yield(void);". Observe that to get full credit for this question your design must take into account that it is targeted at multiprocessor machines.

```
struct mutex {




}

void mutex_init(struct mutex *mp)
{




}

void mutex_lock(struct mutex *mp)
{










}

void mutex_unlock(struct mutex *mp)
{







}
```

4. Imagine a multi-threaded version of the children's card game "Concentration". The game is played with a two-dimensional array of cards which are face-down. Players take turns selecting two cards and turning them face-up. If the cards match, the player captures the pair of cards; otherwise, the player must turn both cards face-down and another player takes a turn. The code below plays the game according to a very naive strategy, but that is not as significant as the fact that it will occasionally deadlock if multiple threads run play() simultaneously.

```
int cards[6][6];
mutex lock[6][6];
mutex pairs_lock;
int pairs_left;
int everybody_done;
```

```
void init(void) {
  ...shuffle 36-card deck into cards[ ][ ]
  ...mutex_init() all of lock[ ][ ]...
  mutex_init(&pairs_lock);
  pairs_left = 18; everybody_done = 0;
}
```

```
int play()
{
  int score = 0;
  while (!everybody_done) {
    int i1, j1, i2, j2, tmp1, tmp2, tmp3, tmp4;

    i1 = generate_random(0, 5); j1 = generate_random(0, 5);
    i2 = generate_random(0, 5); j2 = generate_random(0, 5);

    if ((i1 == i2) && (j1 == j2))
      continue;

    mutex_lock(lock[i1][j1]); mutex_lock(lock[i2][j2]);

    if ((cards[i1][j1] == cards[i2][j2]) && (cards[i1][j1] != GONE)) {
      ++score; cards[i1][j1] = cards[i2][j2] = GONE;
      mutex_lock(pairs_lock);
      if (--pairs_left == 0)
        everybody_done = 1;
      mutex_unlock(pairs_lock);
    }
    mutex_unlock(lock[i1][j1]); mutex_unlock(lock[i2][j2]);
  }
  return (score);
}
```

(a) Draw a "system resource-allocation graph" (as presented in the text and in class) depicting a deadlock situation that can arise from running this code as described. Your graph must depict both threads and resources.

(b) Present a *small* modification to the code above which will ensure it will not deadlock. You should not modify the thread library or call library routines which are not already being called. There are multiple correct answers, but some are better than others. It is acceptable for you to present one or more sequences of code to be substituted and/or added, instead of presenting the entire new version of play(), as long as your "edit" specification(s) is/are clear.

5. Consider the following critical-section protocol:

```
boolean waiting[2] = { false, false };
int turn = 0;

1.        waiting[i] = true;
2.        while (turn != i) {
3.             while (waiting[j])
4.               /* do nothing */ ;
5.             turn = i;
          }
6.        ...critical section...
7.        waiting[i] = false;
8.        ...remainder section...
```

(This protocol is presented in the standard form, i.e., if process 0 is running this code, $i == 0$ and $j == 1$; if process 1 is running this code, $i == 1$ and $j == 0$.)

There is a problem with this protocol. That is, it does not ensure that all three requirements (mutual exclusion, progress, and bounded waiting) are always met. Identify a requirement which is not met and lay out a scenario which demonstrates your claim. Use the format presented in class, i.e.,

P0	P1
waiting[0] = false;	
	turn = 0;