CIS 415: Operating Systems

Prof. Allen D. Malony

Midterm – November 5, 2019

NAME: ______________________________

Section	**Concepts**	**Problems**	**Total**	**Score**
1. Processes and Threads	18	15	33	
2. CPU Scheduling	12	15	27	
3. Synchronization / Deadlocks	18	15	33	
EXTRA CREDIT	–	10	10	
Total	48	45 + 10	93 + 10	

Comments:

1. Take a deep breath. You did your best to study and prepare. The hard part is over. Congratulations!
2. Write your name on the front page now and initial all other pages.
3. You have 80 minutes to do the midterm. There are 3 sections, plus EXTRA CREDIT. Do all 3 sections. Each section is designed to take 25 minutes, more or less. EXTRA CREDIT is there to help you score some more points only if you get really stuck on the section problems.
4. Concept questions are approximately 50% of the total points. They are intended to be short answer. If you are spending more than 5 minutes on ANY concept questions, you are writing too much! Move on!
5. If you have a question, raise your hand and we will try come to you, if we can. Otherwise, we will acknowledge you and you can come to us.

Exams should be challenging learning experiences! Enjoy it!!!

1 Processes and Threads (33)

1.1 Concepts (18)

!!! CHOOSE ONLY 3 OF THE FOLLOWING CONCEPT QUESTIONS TO ANSWER !!!
NOTE: Each question is worth the same. You can choose to answer more than 3 questions, but only 3 will be counted towards your score. Please make it clear which questions should be graded.

What are 2 forms of interprocess communication (IPC)? Give 2 advantages for each.

Shared memory. Through the use of shared segments, processes that are running on the same machine are able to read and write to the same memory. This mechanism is fast and it does not involve the OS except to set up the shared segments. That is, read/write operations happen directly to memory.

Message passing. IPC based on message passing utilizes network communication mechanisms in the OS to allow processes to send and receive messages. One advantage is that the processes do not have to be located on the same machine. A second advantage is that the message communication protocol avoids issues of memory synchronization that are present in shared memory IPC.

Where do the addresses used to reference a process address space come from? Why are these addresses regarded as logical addresses?

The addresses come from executable code that the CPU executes. The compiler, linker, and loader all contribute to creating the addresses produced by the instructions. The addresses are logical because they have not yet been translated to actual physical addresses that will reference the physical memory system.

Why are processes regarded as "heavyweight" and threads as "lightweight" from the perspective of being managed by the OS?

The notion of "weight" here has to do with 2 things: how much information the OS has to manage and the overhead of that management to do certain things (e.g., context switching). A process requires all information necessary to be able to execute it be stored in the process control block (PCB). However, a thread of a process only needs incremental information about its state to be stored, specifically its CPU state. It shares the rest of the information in the PCB. Thus, threads are relatively lighter weight in terms of the size of information. Because of this, the context switching of a process will have higher overhead than the context switching of a thread because the OS has more work to do.

Why are processes thought of a being "real", while user-level threads are considered to be "virtual" (as Prof. Malony tried to emphasize in class)?

Picking up on the answer above, processes are "real" because they have to be represented in the OS by the PCB to be able to execute. That is, they will take up resources in the OS for this purpose. In contrast, user-level threads (we are NOT talking about kernel threads here) are contained within user programs, not the OS. Of course, user-level threads will need to be mapped to kernel-level threads in order to execute.

What do the terms *user mode* and *system mode* mean? What are the effects on what can be done (software and hardware) when in one mode versus the other?

In general, the terms have to do with a mode of the CPU that allows certain instructions to be executed depending on which mode you are in. In particular, system mode lets the currently running program execute instructions that will let it do more "privilege" things. Typically, it is only the OS that is allowed to run in system mode.

Some of the privileged things have to do with accessing kernel data structures, controlling devices, and context switching processes. In user mode it is important to disallow certain operations that can violate process protection, manipulate shared data outside of the user process, and so on. In this way, the notion of "mode" extends beyond just allowing certain CPU instructions to be executed to include enabling certain (more privileged) software to be run.

What is the purpose of interrupts? Is it possible for a user program to disable interrupts? What is the common mechanism in a machine to get control back to the OS?

The main purpose of interrupts is so that the OS can be made aware of and respond to external machine events. Interrups are hardware actions whereby the CPU will start executing OS code (i.e., the interrupt handler) when an interrupt occurs. In this way, the OS is able to take back control. In particular, a timer interrupt is a key mechanism to prevent a situation whereby no other interrupts occur and an errant process can take over control of the machine (e.g., entering an infinite loop). Interrupts can be disabled, but only by the OS. Otherwise, a user process could prevent the OS from every gaining back control, if all interrupts, especially the timer interrupt is disabled.

1.2 Problems (15)

1.2.1 "When you come to a fork in the road, take it!" – Yogi Berra (15)

Consider the following program. How many processes are created altogether, including the original main process? The `sleep(10)` statement is used to ensure that all processes will be sleeping before ANY process wakes up. Draw the process hierarchy at that time (i.e., at the time when all the processes that could be created are sleeping). What are the values of the variables at that time for each process: `pid1`, `pid2`, `pid3`, `pid4`? If you need to know a Unix process identifier (pid), just use a unique 5-digit number.

```
#include <stdio.h>
#include <unistd.h>

int pid1, pid2, pid3, pid4;

int main ()
{
  pid1 = -1; pid2 = -2; pid3 = -3; pid4 = -4;
  pid2 = fork();
  pid3 = fork();
  pid4 = fork();
  sleep(10);
  return 0;
}
```

We are going to assume that all `fork()` calls are successful. The trick to this problem is to follow carefully the child processes created and where they start executing. The child processes themselves (except for the last ones) will fork their own children. The following shows what occurs with respect to the creation of child processes. I chose process identifiers to help keep track. The values of the variables for each process are shown in parentheses.

```
P1 (starting process)             % P1 creates 3 child processes
  +-> P11   +-> P12   +-> P13     % (pid1 = -1; pid2 = 11000; pid3 = 12000; pid4 = 13000;
P11                                 % P11 creates 2 child processes
  +-> P111  +-> P112                % (pid1 = -1; pid2 = 0; pid3 = 11100; pid4 = 11200;

P111                                % P111 creates 1 child process
  +-> P1111                         % (pid1 = -1; pid2 = 0; pid3 = 0; pid4 = 11110;

P1111                               % P112 creates 0 processes
                                    % (pid1 = -1; pid2 = 0; pid3 = 0; pid4 = 0;

P112                                % P112 creates 0 processes
                                    % (pid1 = -1; pid2 = 0; pid3 = 11100; pid4 = 0;

P12                                 % P12 creates 1 child process
  +-> P121                          % (pid1 = -1; pid2 = 11000; pid3 = 0; pid4 = 12100;

P121                                % P121 creates 0 processes
                                    % (pid1 = -1; pid2 = 11000; pid3 = 0; pid4 = 0;

P13                                 % P13 creates 0 processes
                                    % (pid1 = -1; pid2 = 11000; pid3 = 12000; pid4 = 0;
```

A total of 8 processes execute.

2 Scheduling (27)

2.1 Concepts (12)

!!! CHOOSE ONLY 2 OF THE FOLLOWING CONCEPT QUESTIONS TO ANSWER !!!
NOTE: Each question is worth the same. You can choose to answer more than 2 questions, but only 2 will be counted towards your score. Please make it clear which questions should be graded.

What is the *CPU-I/O burst cycle* and why is it relevant to processor scheduling?

The execution of a process can be described as a sequence of needing to run on the CPU and then needing to do I/O. (The notion of "I/O" is better interpreted as "not needing the use of the CPU for some reason" in practice.) The term "burst" is just used to indicate how long a process is using the CPU or doing I/O. Knowing something about the CPU-I/O burst profile of a process during its execution would help the process scheduler make more effective decisions. It could, for instance, identify compute-intensive processes from interactive one.

List 3 actions that might cause the OS to make a scheduling decision.

1. Round-robin, quanta-based scheduling depends on timer interrupts.
2. A process makes a system call to perform file I/O which might result in that process blocking until the I/O is completed. In that case, the OS will make a decision to schedule another process.
3. Various actions during a process execution can cause the OS to regain control and decide to schedule another process. A segmentation fault is a good example. The process in this case will halt and the OS must make a scheduling decision.

If a machine had more than 1 CPU, would it make sense to run a different scheduling policy on different CPUs? Explain.

The availability of multiple CPU resources would allow processes of different execution types (e.g., interactive, computation) to be isolated to different CPUs and scheduled with algorithms that are more effective for processes of those types. For instance, interactive processes might be given shorter time quanta versus longer ones for computation-intensive processes. Mixing the different types of processes together and scheduling them on all the CPUs might lead to less performance overall.

What is the quantum in Round-Robin scheduling? How does it get implemented? What happens in Round-Robin scheduling as the quantum increases? What happens in Round-Robin scheduling as the quantum decreases?

The quantum represents that amount of time the RR scheduler will allot to a process to run on the CPU before it will preempt it. The preemption mechanism is implemented using a clock interrupt to enable the OS to regain control. Because there is overhead in responding to the clock interrupt, as well as process context switching, the quantum size matters. Increasing the quantum can reduce responsiveness and increase average waiting time. Too big of a quantum results in the RR scheduler approximating FCFS. Decreasing the quantum can make scheduling less efficient because relatively more time will be spent in overhead versus executing the processes.

What is the idea behind using a multilevel queueing system for scheduling? What advantage is there in having different quantum sizes at different levels of a multilevel queueing system?

The general idea behind a multilevel queueu system is to try to best place a process in a queue that is scheduled in a way that best matches certain characteristics of the processes in that queue. This can be done through a combination of scheduling algorithms, priorities, quantum, and so on.
Adjusting the quantum sizes at different levels can be used, for instance, to distinguish between interactive processes with short CPU burst and more computationally intensive processes with longer CPU bursts.

2.2 Problems (15)

!!! CHOOSE ONLY 1 PROBLEM TO ANSWER !!!
NOTE: Each problem is worth the same. Each problem may have multiple questions. Please make it clear which problem should be graded.

2.2.1 Changing Priorities (15)

Priority-based process scheduling selects the next process to be allocated to the CPU based on the process's priority, computed as a function of process and system parameters. The behavior of most scheduling algorithms can be replicated by a priority-based scheduler with the proper choice of priority function. For each of the scheduling algorithms below:

i. Give a brief definition of what the scheduling algorithm does.

ii. Specify the priority function as a simple function (arithmetic operators) of one or more of the following parameters:

a = attained service time

r = real time the process has spent in the system

t = total service time required by the process

The process with the HIGHEST priority is scheduled next.

FIFO (First In / First Out)

FIFO schedules jobs based on their arrival time, earliest first.
$priority = r$
When a job arrives to the system, its r time value starts at 0 and begins ticking. Any job that arrived earlier and is still in the system will have a larger r value. Any job that arrived later will have a smaller r value. FIFO is nonpreemptive by definition.

LIFO (Last In / First Out)

LIFO schedules jobs based on arrival time, latest first.
$priority = 1/r$ or $priority = -r$
This is exactly the opposite of FIFO. When a job arrives to the system, its priority is nonpreemptive. In this case, scheduling decisions are made after the currently running process completes. If the scheduler is preemptive, the currently running process could be interrupted and context switched to the new job.

SJF (Shortest Job First)

SJF schedules jobs based on required service times, shortest first.
$priority = 1/t$ or $priority = -t$
The total service time must be known at the time of job arrival. The scheduler is assumed to be nonpreemptive, meaning scheduling decisions are made when the current job completes. If the scheduler is preemptive, the problem turns into SRT. See below.

RR (Round Robin)

RR gives each process an equal quantum of time.
$priority = c$ where c is some constant
This is a bit of a trick question, since it actually does not involve any of the parameters. It differs from the others in that it is necessarily preemptive.

SRT (Shortest Remaining Time)

SRT schedules jobs based on amount of service time remaining, shortest first.
$priority = 1/(t - a)$ or $priority = a - t$
This is SJF with preemption. If when a job arrives, it has smaller service time than the remaining service time of the currently running job, a context switch occurs with the new job. The previously running job goes back into the ready queue in SRT order.

2.3 Bursting the CPU Bubble! (15)

Consider the following set of processes, their CPU burst times, their arrival times, and their priorities (where a lower number represents a higher priority).

Process ID	CPU Burst Time	Arrival Time	Priority
P_1	6	0	2
P_2	3	4	3
P_3	4	5	1
P_4	1	7	6

(a) Draw a Gantt Chart (timeline) for the task schedule that would be generated using a First-Come, First-Serve scheduler. (Each cell represents 1 time unit, starting at time 0.)

P_1	X	X	X	X	X	X													
P_2							X	X	X										
P_3										X	X	X	X						
P_4														X					

(b) Draw a Gantt Chart (timeline) for the task schedule that would be generated using a preemptive Shortest Job

First scheduler. (Each cell represents 1 time unit, starting at time 0.)

P_1	X	X	X	X	X	X													
P_2							X		X	X									
P_3											X	X	X	X					
P_4								X											

(c) Draw a Gantt Chart (timeline) for the task schedule that would be generated using a preemptive priority scheduler. (Each cell represents 1 time unit, starting at time 0.)

P_1	X	X	X	X	X					X									
P_2											X	X	X						
P_3						X	X	X	X										
P_4														X					

3 Synchronization / Deadlocks (33)

3.1 Concepts (18)

!!! CHOOSE ONLY 3 OF THE FOLLOWING CONCEPT QUESTIONS TO ANSWER !!!
NOTE: Each question is worth the same. You can choose to answer more than 3 questions, but only 3 will be counted towards your score. Please make it clear which questions should be graded.

Why do we care about concurrency in operating systems? Is it advantageous to have logical concurrency even without physical concurrency (e.g., multiple CPUs)? Give one reason why.

The concept of concurrency in computer science is simply when more than one action can take place at the same (logical) time. We think of concurrency in the context multiple processes executing during the same (logical) time. Because an operating system is managing a complex machine environment, it is advantageous for it to be able to logically separate actions taking place in the machine and allow those actions to be concurrent. By doing so, it can manage resources more efficiently and support logical operating abstractions that hide lower-level complexity. Logical concurrency does not require physical concurrency to have positive advantages. For instance, allowing logical concurrency is necessary for the OS to effectively orchestrate the sharing of machine resources (e.g., CPU, memory) across multiple processes.

What does it mean for a set of instructions to be atomic?

A set of instructions is atomic if the execution of all of those instructions can not be interrupted. It is as if the set of instructions executed like a single instruction.

What is the difference between busy waiting and blocking (e.g., in a semaphore's implementation) with respect to responsiveness and efficiency? Do you see any advantages to using the `sched_yield()` Linux system call to improving the performance of busy waiting?

```
int sched_yield(void);
sched_yield() causes the calling thread to relinquish the CPU.  The
thread is moved to the end of the queue for its static priority and a
new thread gets to run.
```

A process waiting on a semaphore using busy waiting will continue to test the wait condition. A blocking semphore will stop the process when it would otherwise busy wait and put it on a waiting queue. Busy waiting can be responsive to a signal that might occurs quickly, but inefficient because it will keep the CPU busy if the signal takes a long time to arrive. In contrast, blocking is efficient in CPU usage, but requires context switching to stop and wakeup a waiting process. The advantage of using `sched_yield()` Linux system call is to give another process a chance to do something, while keeping the calling process alive. For instance, the process calling `sched_yield()` might be in a wait condition and instead of blocking decides to let another process run, one that just might be removing the wait condition. In this way, it is possible for the process encountering waiting to more quickly keep execution once the wait condition is satisfied.

In deadlock avoidance, what is meant for the system to be in a safe state?

A system is in a safe state if there exists a sequence of resource assignments to processes such that all processes will terminate.

What is starvation?

Starvation is a condition of a process when it is not allowed access to a resource required for its continued execution for an indefinite period of time. This could happen, for instance, because the process is not assigned the CPU or is unable to gain access to a data resource that it needs to execute.

3.2 Problems (15)

!!! CHOOSE ONLY 1 OF THE 2 PROBLEMS FOLLOWING PARTS TO ANSWER !!!
NOTE: Each problem is worth the same. Each problem may have multiple questions. Please make it clear which problem you are doing should be graded.

3.2.1 Critical Sections – Believe It or Not! (15)

Consider the following proposed solution to the critical section problem for two processes as published by Hyman in the Communications of the ACM in 1966:

```
int blocked[2]={0,0}; /* initially blocked[0] = 0, blocked[1] = 0 */
int turn=0;           /* initially turn=0 */

// process 0                              // process 1
while (1) {                               while (1) {
  <code outside critical section>           <code outside critical section>
  blocked[0] = 1;                           blocked[1] = 1;
  while (turn != 0) {                       while (turn != 1) {
    while (blocked[1] == 1);                  while (blocked[0] == 1);
      ;                                         ;
    turn = 0;                                 turn = 1;
  }                                         }
  <CRITICAL SECTION>                        <CRITICAL SECTION>
  blocked[0] = 0;                           blocked[1] = 0;
  <code outside critical section>           <code outside critical section>
}                                         }
```

Was the CACM fooled into publishing an incorrect solution? If yes, find a counter example that demonstrates this.

Yes. Suppose that both processes have set their `blocked` flag and are executing the next `while` statement. If `turn` is 0, process 0 will be allowed to execute the critical section, but process 1 will execute the next `while` statement where it waits for `blocked[0]` to be set to 0. It is the case that process 0 does this, but suppose it then resets `blocked[0]` to 1, before process 1 sees that it was set to 0. If this continues, process 1 will be locked out and bounded waiting is not satisfied.

3.2.2 Monkey Business (15)

Lost in the jungle there is a deep ravine with a roaring river where thrill-seeking monkeys go to play. The only way to cross the ravine is a vine hanging from 2 trees on either side. Monkeys use the vine to climb from one side to the other. However, it is only possible to do so in 1 direction at a time. Furthermore, there can only be up to 6 monkeys on the vine at any time, otherwise the vine will break and the monkeys will fall into the water. See figure below.

Each monkey operates in a separate thread, which executes the code below:

```
typdef enum {EAST, WEST} Destination; /* EAST = 0, WEST = 1 */
void monkey(int id, Destination d) {
  WaitUntilSafeToCross(d);
  CrossRavine(id, d);
  DoneWithCrossing(d);
}
```

Parameter `id` holds a unique number identifying that monkey. `CrossRavine(int id, Destination d)` is provided to you and returns when the calling monkey has safely reached its destination. Semaphores are used to implement `WaitUntilSafeToCross()` and `DoneWithCrossing()`.

Part A. Consider the following partial solution showing only `WaitUntilSafeToCross()` code in full. Fill in the `DoneWithCrossing()` code where "..." appears. (Hint: only 1 statement is needed for each "..." line.)

```
int monkeycount[2] = {0,0}; /* monkey counter for each direction */
semaphore mutex[2] = {1,1}; /* mutual exclusion for each direction*/
semaphore maxonrope = 6;  /* ensure maximum 6 monkeys on the rope*/
semaphore rope = 1;         /* ensure monkey on rope is heading in same direction */

void WaitUntilSafeToCross(Destination d) {
  wait(mutex[d]);
  monkeycount[d]++;
  if (monkeycount[d] == 1) { /* first monkey in line, acquire rope */
    wait(rope);
  }
  signal(mutex[d]);
  wait(maxonrope);
}

void DoneWithCrossing(Destination d) {
  ...   /* 1 */
  signal(maxonrope);
  ...   /* 2 */
  if (monkeycount[d] == 0) { /* last monkey, release the rope */
    ... /* 3 */
  }
  signal(mutex[d]);
}
```

1: wait(mutex[dest]);
2: monkeycount[dest]–;
3: signal(rope);

Part B. Your solution above essentially maximizes throughput. However, it is likely there is a problem. What is it? Consider the following addition of a semaphore and change to the `WaitUntilSafeToCross()` code. Does it fix the problem? (Hint: How do you change direction? Interestingly, the `DoneWithCrossing()` code does not need to be changed (that is, if it is implemented properly) to get correct solution.)

```
semaphore order = 1;
void WaitUntilSafeToCross(Destination d)
{
  wait(order);
  wait(mutex[d]);
  monkey_count[d]++;
    if (monkeycount[d] == 1) { /* first monkey in line, acquire rope */
      wait(rope);
    }
    signal(mutex[d]);
    signal(order);
    wait(maxonrope);
  }
```

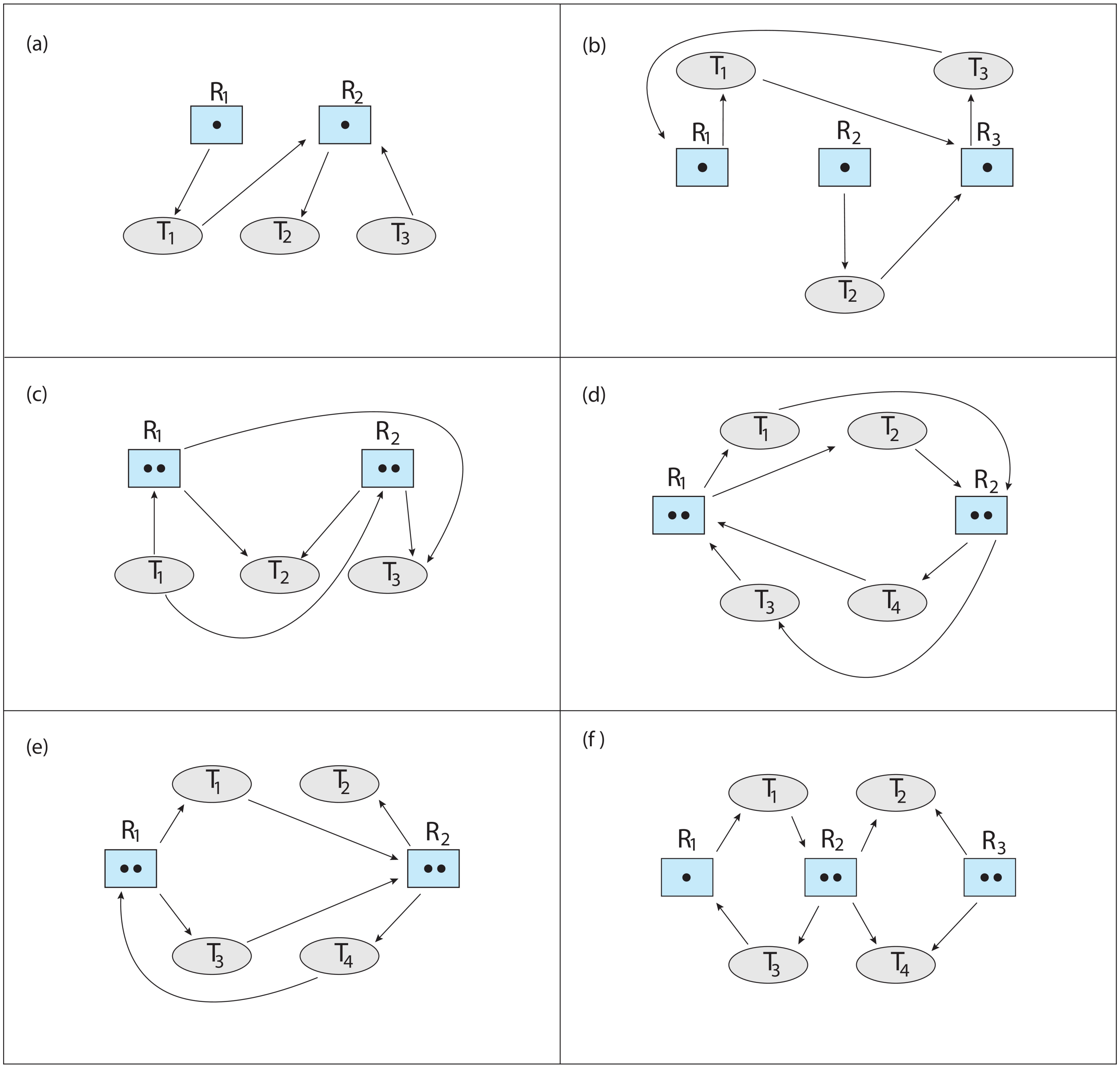
The problem is that a continual stream of monkeys from one side could effectively lock out the monkeys from the other from ever getting access to the rope and crossing. The code change essentially forces a change in the order of rope access when both sides have monkeys waiting to cross. It is possible that there are several monkeys "on the rope" when this happens, but they will complete their crossing before the order is switched.

4 EXTRA CREDIT (10)

!!! This is EXTRA CEDIT. Do only if you get stuck on other things. !!!

4.1 To be deadlocked or Not to be deadlocked, That is the question! (10)

Which of the six resource allocation graphs shown below illustrate deadlock? For those situations that are deadlocked, provide the cycle of threads and resources. Where this is not a deadlock situation, illustrate the order in which the threads may complete execution.



```
(a) No deadlock: T2 -> T3 -> T1
(b) Deadlock: R1 -> T1 -> R3 -> T3 -> R1
(c) No deadlock: T2 -> T3 -> T1
(d) Deadlock: R1 -> T1 -> R2 -> T4 -> R1
(e) No deadlock: T2 -> T3 -> T1 -> T4
(f) No deadlock: T2 -> T4 -> T1 -> T3
```