Name: Gustavo Hammerschmidt
CS433 Written Homework 4

(80 points) All question numbers refer to exercises in the textbook (10th edition). Make sure you use the right textbook and answer the right questions. Students must finish written questions individually. Type your answers and necessary steps clearly.

1. **(5 points) 6.8** Race conditions are possible in many computer systems. Consider an online auction system where the current highest bid for each item must be maintained. A person who wishes to bid on an item calls the bid(amount) function, which compares the amount being bid to the current highest bid. If the amount exceeds the current highest bid, the highest bid is set to the new amount. This is illustrated below:

```
       void bid(double amount)
{            if (amount >
highestBid)
        highestBid = amount;
   }
```

Describe how a race condition is possible in this situation and what might be done to prevent the race condition from occurring.

> Two or more bidders could make a bid at the exact same moment, that means that each one would make a bid on the latest highest bid for the same item and that they would alter the latest amount value, therefore leading to an inconsistence: the variable's new value could be either one of the bids made if either bidder had took the latest value of the variable into account; a way to circumvent this problem would be to set a incoming priority buffer to take the bid of every bidder, this the following bidder would always make a bid on the true latest bid or to define a mutex locker to give each incoming bidder -- first that comes get served first -- access to change the latest bid value; that would guarantee each one a chance to cahneg the resource, consistently, by bidding on the proper amount.

2. **(5 points) 6.10** The `compare_and_swap()` instruction can be used to design lock-free data structures such as stacks, queues, and lists. The program example shown in Figure E6.18 presents a possible solution to a lock-free stack using CAS instructions, where the stack is represented as a linked list of `Node` elements with `top` representing the top of the stack. Is this implementation free from race conditions?

```
typedef struct node {        value_t data;          struct node *next;
```

```
   } Node;

   Node *top; // top of stack

   void push(value_t item) {
     Node *old_node;
     Node *new_node;

             new_node = malloc(sizeof(Node));
new_node->data = item;
                 do
{
old_node = top;
      new_node->next = old_node;
    }
             while (compare_and_swap(top,old_node,new_node) !=
old_node);  }
       value_t pop() {
       Node *old_node;
       Node *new_node;
                 do {
       old_node = top;
       if (old_node == NULL)
                 return
NULL;
      new_node = old_node->next;
    }
    while (compare_and_swap(top,old_node,new_node) != old_node);
                 return
old_node->data;   }
```

The compare and swap will only provide progress to the structure, but it will not prevent race conditions.

3. (5 points) 6.12 Some semaphore implementations provide a function `getValue()` that returns the current value of a semaphore. This function may, for instance, be invoked prior to calling `wait()` so that a process will only call `wait()` if the value of the semaphore is > 0, thereby preventing blocking while waiting for the semaphore. For example: `if (getValue(&sem) gt; 0)`
`wait(&sem);`
Many developers argue against such a function and discourage its use. Describe a potential problem that could occur when using the function `getValue()` in this scenario.

The execution length of time of some threads could never really match or get synchronized efficiently in a manner that two threads would execute their critical sections while the another thread executes its remaining section; leading, in this scenario, to some resources or operations in

the critical section to never be used; but, usually, what this action would have a greater effect on would be the lost of performance that a program can have when executing itself concurrently; and the idea of doing such thing defies the main purpose of using parallel execution: because usually what you want is for either thread to execute it's critical section when an opportunity to do so is given.

4. **(5 points) 6.15** Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.

A user-level program could have a deadlock, if the interrupts were disabled, no preemption would occur and the processor would execute the program indefinetily, that would freeze the computer.

5. **(5 points) 6.19** Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism—a spinlock or a mutex lock where waiting processes sleep while waiting for the lock to become available:

- The lock is to be held for a short duration.
    - Spinlock.
- The lock is to be held for a long duration.
    - Mutex.
- The thread may be put to sleep while holding the lock.
    - Mutex.

6. **(10 points) 6.22** Consider the code example for allocating and releasing processes shown in Figure E6.21.

```
      #define MAX_PROCESSES 255
int number_of_processes = 0;
      /* the implementation of fork() calls this function
*/    int allocate_process() {     int new_pid;
```

```
    if (number_of_processes == MAX_PROCESSES)

return -1;        else
{
       /* allocate necessary process resources */
       ++number_of_processes;

  return new_pid;
     }
   }
      /* the implementation of exit() calls this function
*/    void release_process() {           /* release
process resources */
     --number_of_processes;
   }
```

---

a. Identify the race condition(s).

There is a race condition on the number_of_processes variable.

b. Assume you have a mutex lock named `mutex` with the operations `acquire()` and `release()`. Indicate where the locking needs to be placed to prevent the race condition(s).

The operations should be placed at the beginning and at the end of both functions: allocate_process and release_process.

c. Could we replace the integer variable `int number_of_processes = 0` with the atomic integer `atomic_t number_of_processes = 0}` to prevent the race condition(s)?

It wouldn't prevent the race condition on the if statement, two threads could still increment the number_of_processes variable at the same time, leading to more processes being created.

7. **(5 points) 6.23** Servers can be designed to limit the number of open connections. For example, a server may wish to have only N socket connections at any point in time. As soon as N connections are made, the server will not accept another incoming connection until an existing connection is released.

Illustrate how semaphores can be used by a server to limit the number of concurrent connections.

At every incoming request to establish a connection to the server, the server locks its mutex when replying the user with an synchronize-acknowledgement message and waits for the user to send an acknowledge message, thereby defining a 3-way handshake communication to establish connection, the server receives the message and decreases(acquires) its connections-counter semaphore, unlocks its mutex for further connections(a time to establish to connection would be defined so the mutex can be unlocked if the client waits too long). An there is another way to do it: when a client wishes to establish a connection, the server receives the synchronize request checks if there is any connection left in the semaphore, it acquires a connection from the semaphore temporarily, sends an synchronize-acknowledgment response and waits for the client's acknowledgement; if the client doesn't acknowledge it, then the server will return(release) the connection to the semaphore, hence ensuring that the limit of maximum connections allowed will not be trespassed.

8. **(5 points) 6.26** Describe how the `signal()` operation associated with monitors differs from the corresponding operation defined for semaphores.

   With semaphores, the signal() operation represents that a thread wishes to have access to the semaphore and that is waiting for the semaphore to be available; once it has it, then it can operate or access some resource. The semaphore's signal operation gives a thread the authority to have the processor for itself. The monitor's signal, basically, indicates that a thread or some resource is ready to be use by the monitor itself if it requires; here the monitor is the one with the authority over the processor and threads signal the monitor to give it authority to operate over them.

9. **(5 points) 7.8** The Linux kernel has a policy that a process cannot hold a spinlock while attempting to acquire a semaphore. Explain why this policy is in place.

   If a process is waiting to acquire a semaphore, an interrupt could occur and preempt the processor, the other processes, that the processor would execute, would not be able to acquire the spinlock and this would lead to a deadlock.

10. **(10 points) 8.18** Which of the six resource-allocation graphs shown in Figure E8.15 illustrate deadlock? For those situations that are deadlocked, provide the cycle of threads and resources. Where there is not a deadlock situation, illustrate the order in which the threads may complete execution.

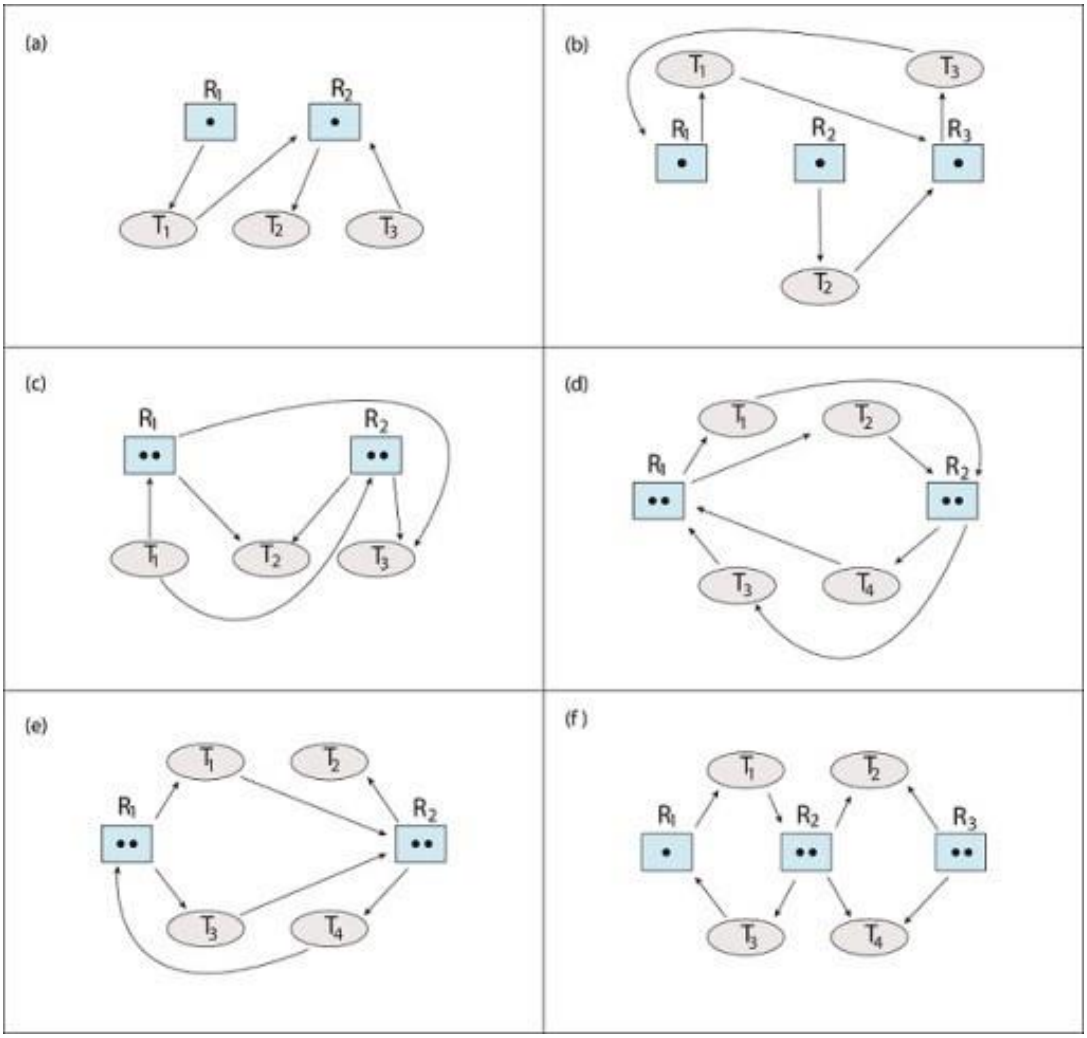

Figure E8.15 Resource-allocation graphs for Exercise 8.18.

a) No deadlock. The execution orders could be either T2->T3->T1 or T2->T1->T3.
b) There is a deadlock. The T3 requires R1, which is being held by T1, and T1 requires R1, which is being held by T3. The cycle is T1 -> R3 -> T3 -> R1.
c) No deadlock. The execution orders could be T3->T2->T1, T2->T3->T1, T2->T1->T3 or T3->T1->T2.
d) There is a deadlock. The cycle includes T1, T2, T3, T4, R1 and R2.
e) No deadlock. The execution orders could be T2->T1->T3->T4, T2->T1->T4->T3, T2->T3->T1->T4 or T2->T3->T4->T1.
f) No deadlock. The execution orders could be T2->T4->T1->T3 or T4->T2->T1->T3

11. (10 points) 8.27 Consider the following snapshot of a system:

	Allocation	*Max*	*Need*
	A B C D	*A B C D*	*A B C D*
T_0	1 2 0 2	4 3 1 6	3 1 1 4
T_1	0 1 1 2	2 4 2 4	2 3 1 2
T_2	1 2 4 0	3 6 5 1	2 4 1 1
T_3	1 2 0 1	2 6 2 3	1 4 2 2
T_4	1 0 0 1	3 1 1 2	2 1 1 1

a. Using the banker's algorithm, determine whether or not each of the following states is unsafe. If the state is safe, illustrate the order in which the threads may complete. Otherwise, illustrate why the state is unsafe.

b. a. ***Available*** = (2,2,2,3)

The system is in safe state. The order it may complete is T4 -> T0 -> T1 -> T2 -> T3.

Available	2 2 2 3	3 2 2 4	4 4 2 6	4 5 3 8	5 7 7 8
Thread	T4	T0	T1	T2	T3

c. b. ***Available*** = (4,4,1,1)

The system is in safe state. The order it may complete is T2 -> T4 -> T1 -> T0 -> T3.

Available	4 4 1 1	5 6 5 1	6 6 5 2	6 7 6 4	7 9 6 6
Thread	T2	T4	T1	T0	T3

d. c. ***Available*** = (3,0,1,4)

The system is not in safe state. There is no more resource B available to be allocated.

e. d. ***Available*** = (1,5,2,2)

The system is in safe state. The order it may complete is T3 -> T1 -> T2 -> T0 -> T4.

Available	**1 5 2 2**	2 7 2 3	2 8 3 5	3 10 7 5	4 12 7 7
Thread	**T3**	**T1**	**T2**	**T0**	**T4**

12. (10 points) 8.28 Consider the following snapshot of a system:

	Allocation	***Max***	***Need***	***Available***
	A B C D	*A B C D*	*A B C D*	*A B C D*
T_0	3 1 4 1	6 4 7 3	3 3 3 2	2 2 2 4
T_1	2 1 0 2	4 2 3 2	2 1 3 0	
T_2	2 4 1 3	2 5 3 3	0 1 2 0	
T_3	4 1 1 0	6 3 3 2	2 2 2 2	
T_4	2 2 2 1	5 6 7 5	3 4 5 4	

Answer the following questions using the banker's algorithm:

a. Illustrate that the system is in a safe state by demonstrating an order in which the threads may complete.

The order it may complete is T2 -> T1 -> T0 -> T3 -> T4.

Available	2 2 2 4	2 3 4 4	4 4 7 4	7 7 10 6	9 9 12 8
Thread	T2	T1	T0	T3	T4

b. If a request from thread T_4 arrives for (2,2,2,4), can the request be granted immediately?

No, because the resulting state is unsafe, even though the resources are available.

c. If a request from thread T_2 arrives for (0,1,1,0), can the request be granted immediately?

Yes, because (0, 1, 1, 0) ≤ (2, 2, 2, 4) and the resulting state is safe. The order it may complete is T2 -> T1 -> T0 -> T3 -> T4.

d. If a request from thread T_3 arrives for (2,2,1,2), can the request be granted immediately?

Yes, because (2, 2, 1, 2) ≤ (2, 2, 2, 4) and the resulting state is safe. The order it may complete is T3 -> T0 -> T1 -> T2 -> T4.