Chris Banci
CS433
4/18/17

Written Homework #4

5.8) The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P0 and P1, share the following variables:

```
boolean flag[2]; /* initially false */
int turn;
```

The structure of process Pi (i == 0 or 1) is shown in Figure 5.21. The other process is Pj (j == 1 or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

- To insure that only one process is executing in a critical section, three requirements must be met:
  1) Mutual exclusion = the algorithm passes the first requirement through the flag and turn variables. If both are set true, only 1 will be able to enter.
  2) Process = the algorithm passes the second requirement through its flag and turn variable. If a process wants to re-enter the critical section before the other process, it will have to set its flag variable to true and change its turn variable again. Alternations cannot be made.
  3) Bounded waiting = the algorithm passes the third algorithm through its turn variable. Inside the critical section, the algorithm sets the next turn variable to the other process, making sure its the next to enter the critical section.

5.10) 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.

```
do {
   flag[i] = true;

   while (flag[j]) {
      if (turn == j) {
         flag[i] = false;
         while (turn == j)
            ; /* do nothing */
         flag[i] = true;
      }
   }

      /* critical section */

   turn = j;
   flag[i] = false;

      /* remainder section */
} while (true);
```

Figure 5.21 The structure of process P_i in Dekker's algorithm.

- User-level programs can prevent timer interrupts and context switching if given the ability to disable interrupts. Due to this reason alone, other processes aren't allowed to execute.

5.14) Describe how the compare and swap() instruction can be used to provide mutual exclusion that satisfies the bounded-waiting requirement.

- Compare and swap() instruction can be implemented as the test_and_swap() algorithm to provide mutual exclusion and bounded-waiting requirement.

```
do {
        waiting[i] = true;
        key = true;
        while (waiting[i] && key)
                key = test and set(&lock);
        waiting[i] = false;

        /* critical section */
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
                j = (j + 1) % n;
        if (j == i)
                lock = false;
        Else
                waiting[j] = false;

        /* remainder section */
} while (true);
```

- A wait and lock can be used to satisfy the mutual exclusion requirement where if waiting for both processes are set to true, only one is able to enter the critical section. Bound waiting can be proved by noting that there is a limited number of times a process is allowed to enter the critical section. Waiting processes wait n-1 turns.

5.17) 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 is the best mechanism for this scenario because it locks threads for the shortest time.

• The lock is to be held for a long duration.
  - Mutex lock is the best mechanism for this scenario because it can lock threads for long periods of time.

• A thread may be put to sleep while holding the lock.
  - Mutex lock is the best mechanism for this scenario as it allows for threads go into sleep mode while holding the lock.

5.20) Consider the code example for allocating and releasing processes shown in Figure 5.23.

a. Identify the race condition(s)

- The variable 'number_of_processes' is a race condition.

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).

- Acquire() should be placed at the beginning and release() should be placed at the end.

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)?

- No, replacing the integer variable with an atomic integer to prevent the race condition would not work because the race condition occurs in allocate_process().

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

- A counting semaphore can be used to limit the number of concurrent connections. The counting semaphore can be initialized to size N indicating that there is *N* available connections. Each time a new connection is made, the count is decremented and Each time a connection is closed, the count is incremented. This is similar to the producer-consumer problem where acquire and release functions are required to synchronize inserting and removing from the buffer.

7.16) In a real computer system, neither the resources available nor the demands of processes for resources are consistent over long periods (months). Resources break or are replaced, new processes come and go, and new resources are bought and added to the system. If deadlock is controlled by the banker's algorithm, which of the following changes can be made safely (without introducing the possibility of deadlock), and under what circumstances?

a. Increase Available (new resources added).

- This can be made safely anytime

b. Decrease Available (resource permanently removed from system).

- This can be made safely only when (max <= available)

c. Increase Max for one process (the process needs or wants more resources than allowed).

- This can be made safely only when (max <= available)

d. Decrease Max for one process (the process decides it does not need that many resources).

- This can be made safely anytime

e. Increase the number of processes.

- This can be made safely anytime.

f. Decrease the number of processes.

- This can be made safely anytime

7.22) Consider the following snapshot of a system:

	Allocation	*Max*
	A B C D	*A B C D*
P_0	3 0 1 4	5 1 1 7
P_1	2 2 1 0	3 2 1 1
P_2	3 1 2 1	3 3 2 1
P_3	0 5 1 0	4 6 1 2
P_4	4 2 1 2	6 3 2 5

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 processes may complete. Otherwise illustrate why the state is unsafe.

a. Available = (0, 3, 0, 1)

	MAX				ALLOCATION				NEED			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	5	1	1	7	3	0	1	4	2	1	0	3
p1	3	2	1	1	2	2	1	0	1	0	0	1
p2	3	3	2	1	3	1	2	1	0	2	0	0
p3	4	6	1	2	0	5	1	0	4	1	0	2
p4	6	3	2	5	4	2	1	2	2	1	1	3

p2 need (0,2,0,0) <= avail (0,3,0,1) -> p2 alloc + avail = avail (3,4,2,2)
p1 need (1,0,0,1) <= avail (3,4,2,2) -> p1 alloc + avail = avail (5,6,3,2)
p3 need (4,1,0,2) <= avail (5,6,3,2) -> p3 alloc + avail = avail (5,11,4,2)
p0 need (2,1,0,3) > avail (5,11,4,2)
p4 need (2,1,1,3) > avail (5,11,4,2)

Since the P4 and P5 is greater than the current available resource (5,11,4,2), the state is UNSAFE.

b. Available = (1, 0, 0, 2)

	MAX				ALLOCATION				NEED			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	5	1	1	7	3	0	1	4	2	1	0	3
p1	3	2	1	1	2	2	1	0	1	0	0	1
p2	3	3	2	1	3	1	2	1	0	2	0	0
p3	4	6	1	2	0	5	1	0	4	1	0	2
p4	6	3	2	5	4	2	1	2	2	1	1	3

P1 need (1,0,0,1) <= avail (1,0,0,2) -> p1 alloc + avail = avail (3,2,1,2)

P2 need (0,2,0,0) <= avail (3,2,1,2) -> p2 alloc + avail = avail (6,3,3,3)
P3 need (4,1,0,2) <= avail (6,3,3,3) -> p3 alloc + avail = avail (6,8,4,3)
P4 need (2,1,1,3) <= avail (6,8,4,3) -> p4 alloc + avail = avail (10,10,5,5)
P0 need (2,1,0,3) <= avail (10,10,5,5) -> p5 alloc + avail = avail (13,10,6,9)

Since all processes complete without a deadlock, the state is SAFE

- Safe sequence = (p1,p2,p3,p4,p0)

7.23) Consider the following snapshot of a system:

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
P_0	2 0 0 1	4 2 1 2	3 3 2 1
P_1	3 1 2 1	5 2 5 2	
P_2	2 1 0 3	2 3 1 6	
P_3	1 3 1 2	1 4 2 4	
P_4	1 4 3 2	3 6 6 5	

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 processes may complete.

	MAX				ALLOCATION				NEED			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	4	2	1	2	2	0	0	1	2	2	1	1
p1	5	2	5	2	3	1	2	1	2	1	3	1
p2	2	3	1	6	2	1	0	3	0	2	1	3
p3	1	4	2	4	1	3	1	2	0	1	1	2
p4	3	6	6	5	1	4	3	2	2	2	3	3

P0 need(2,2,1,1) <= avail (3,3,2,1) -> p0 alloc + avail = avail (5,3,2,2)
P3 need(0,1,1,2) <= avail (5,3,2,2) -> p3 alloc + avail = avail (6,6,3,4)
P4 need(2,2,3,3) <= avail (6,6,3,4) -> p4 alloc + avail = avail (7,10,6,6)
P1 need(2,1,3,1) <= avail (7,10,6,6) -> p1 alloc + avail = avail (10,11,8,7)
P2 need(0,2,1,3) <= avail (10,11,8,7) -> p2 alloc + avail = avail (12,12,8,10)

Since all processes complete without a deadlock, the state is SAFE.

- Safe sequence = (P0, P3, P4, P1, P2)

b. If a request from process P1 arrives for (1, 1, 0, 0), can the request be granted immediately?
   - Yes, the request can be granted immediately because the system is in a safe state where all processes can complete without a deadlock.

c. If a request from process P4 arrives for (0, 0, 2, 0), can the request be granted immediately?
   - No, the request cannot be granted immediately because the system is in an unsafe state where the processes cannot be satisfied with the current available resources.