Operating Systems: Solutions to exercises on deadlocks

1. Consider the following snapshot of a system:

   Solutions:

   (a) Safe, T_4, T_0, T_1, T_2, T_3

   (b) Safe, T_2, T_4, T_1, T_0, T_3

   (c) Unsafe. All the threads have need for $B > 0$, thus no thread can complete its execution

   (d) Safe, T_3, T_2, T_0, T_1, T_4

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

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.

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

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

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

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

2. Consider the following snapshot of a system:

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

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

   Solution: T_2, T_0, T_1, T_3, T_4

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

   Solution: No, available become equal to 0 for all resources, but no thread can complete execution with its current resources allocation, so unsafe state if this request is accepted

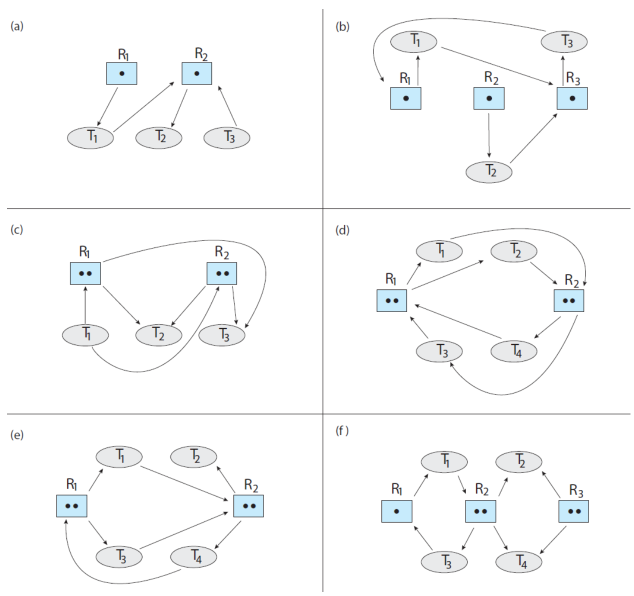
(c) If a request from thread T_2 arrives for (0, 1, 1, 0), can the request be granted immediately?

Solution: Yes, there is feasible execution sequence after this request is granted T_2, T_0, T_1, T_3, T_4

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

Solution: Yes, there is feasible execution sequence after this request is granted T_3, T_0, T_1, T_2, T_4

3. 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 there is not a deadlock situation, illustrate the order in which the threads may complete execution



Solutions:

(a) Wait-for-graph $T_1 \rightarrow T_2 \leftarrow T_3$ has no circle. Thus T_2 will complete first followed by T_1 or T_3 depending on which one capture the resource R_2 first

(b) In the wait-for-graph, $T_1 \rightarrow T_3 \rightarrow T_1$, this form a circle, the system is deadlock

(c) No deadlock. T_2 or T_3 can terminate, releasing resources for T_1. A possible execution sequence is T_2, T_3, T_1

(d) The system is deadlocked, not a single thread can finish as there is no resource available, but each thread needs one resource to complete execution

(e) No deadlock. T_2 don't need extra resource to complete, so can finish first. It will release one unit of resource 2, which will allow either T_1 or T_3 to complete execution. Thus a possible execution sequence is T_2, T_1, T_3, T_4.

(f) Figure (f) has a mistake, 3 instances of resources R_2 are allocated, but this resource has only 2 instances.

4. The program example shown below doesn't always lead to deadlock. Describe what role the CPU scheduler plays and how it can contribute to deadlock in this program.

   Solution: For a guarantee deadlock to occur we must have the scheduler to de-schedule the thread do_work_one just after this thread has captured first_mutex, and then the scheduler schedules the thread do_work_two immediately after do_work_one. Then at this point there is a deadlock as threads do_work_one and do_work_two wait for each other indefinitely.

   If the scheduler schedules a third thread after do_work_one, there might be or might not be a deadlock. For example, if the third thread captures second_mutex without referencing to first_mutex, there will be no deadlock.

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```