Theory Assignment 3

Aaryan, CO21BTECH11001

Q.1 A possible method for preventing deadlocks is to have a single, higher-order resource that must be requested before any other resource. For example, if multiple threads attempt to access the synchronization objects A ··· E, deadlock is possible. (Such synchronization objects may include mutexes, semaphores, condition variables, and the like.) We can prevent deadlock by adding a sixth object, F. Whenever a thread wants to acquire the synchronization lock for any object A ··· E, it must first acquire the lock for object F. This solution is known as containment: the locks for objects A ··· E are contained within the lock for object F. Is there any drawback of this scheme?

Solution:

Yes, there are following drawbacks of this scheme:

- One of the issue is that the threads will spend more time waiting for lock F to be available, even if they need access to only one of the resource A ··· E.
- At any point of time, only one thread will be executing which makes the point of having multiple threads useless.

In conclusion, this scheme leads to decreased concurrency, which can lead to decreased performance.

Q.2 Consider the following resource-allocation policy. Requests for and releases of resources are allowed at any time. If a request for resources cannot be satisfied because the resources are not available, then we check any threads that are blocked waiting for resources. If a blocked thread has the desired resources, then these resources are taken away from it and are given to the requesting thread. The vector of resources for which the blocked thread is waiting is increased to include the resources that were taken away.
For example, a system has three resource types, and the vector Available is initialized to (4,2,2). If thread T0 asks for (2,2,1), it gets them. If T1 asks for (1,0,1), it gets them. Then, if T0 asks for (0,0,1), it is blocked (resource not available). If T2 now asks for (2,0,0), it gets the available one (1,0,0), as well as one that was allocated to T0 (since T0 is blocked). T0 's Allocation vector goes down to (1,2,1), and its Need vector goes up to (1,0,1).

a. **Can deadlock occur? If you answer "yes," give an example. If you answer "no," specify which necessary condition cannot occur.**
b. **Can indefinite blocking occur? Explain your answer.**

Solution:

a. No, deadlock will not occur because the necessary condition of "No preemption" cannot occur.
Since resources are taken away from a blocked thread before it completes, preemption is taking place.
To prove it by contradiction, let's assume that a deadlock occurs. It means that a thread (say T_i) is waiting for resources held by other threads, some of which are waiting for resources held by T_i. But before deadlock was formed, if T_i was already blocked and some other thread would have requested for resources held by T_i, then those resources should have been allocated to that thread.
Alternatively, if T_i would have made a request for resources held by some other thread, then those resources should have been allocated to T_i and T_i should not be waiting for those resources now. This contradicts our assumption.
Therefore, a deadlock can never occur with this scheme.

b. Yes, indefinite blocking can occur with this scheme.
Let's suppose a thread T_1 is waiting for a set of resources $R_w = \{R_{w1}, R_{w2}, ..., R_{wn}\}$ and is holding a set of resources $R_h = \{R_{h1}, R_{h2}, ..., R_{hm}\}$.
Now, let's suppose another thread T_2 comes and request for resources of R_h .Now, R_h will be taken away from T_1 and will be given to T_2 and T_2 will execute.
After execution of T_2 resources R_h are released, but before T_1 would have acquired them, another thread T_3 come and get those resources.
After T_3 completes, R_h is released and a subset of R_w is released, therefore T_1 has to wait again. Other threads will continuously get created and executed and T_1 will never get the complete set of resources ($R = R_w \, U \, R_h$) and will wait indefinitely.

Q.3 Consider a system consisting of four resources of the same type that are shared by three threads, each of which needs at most two resources. Show that the system is deadlock free.

Solution:

If all threads are holding one instance of each resource, then one resource will still be available. Therefore, the available resource will be given to a thread which requested it first. Let's say the available resource is given to the thread T_i.
After T_i has finished execution, it will release two resources which can be acquired by other threads.
In conclusion, it will never happen that all the three threads are waiting for resources held by another thread because in that case, request of one thread can always be satisfied by the available resource. Therefore, the system is deadlock free.

Q.4 Consider the version of the dining-philosophers problem in which the chopsticks are placed at the center of the table and any two of them can be used by a philosopher. Assume that requests for chopsticks are made one at a time. (a) Describe a simple rule for determining whether a particular request can be satisfied without causing deadlock given the current allocation of chopsticks to philosophers. (b) Explain the practical difficulty in implementing this rule.

Solution:

For the given scenario, a deadlock can occur when every philosopher picks one chopstick and is waiting for another chopstick.

a. A simple rule for avoiding deadlock is to reject the request of a philosopher when:
    i. There is only one chopstick available and
    ii. No philosopher has two chopsticks in hand and
    iii. The philosopher who is requesting doesn't have any chopstick in hand.

b. To implement this rule, we need to keep track of the number of chopsticks available and the number of philosophers with two chopsticks. To determine the latter, an array needs to be maintained which will store the number of chopsticks that are taken by the philosophers.
    Let's denote these quantities with the following variables:
    `numChopStickAvail` : Number of chopsticks available
    `numPhiloWith2Chop` : Number of philosophers with two chopsticks.
    `numChopSticks` : $numChopSticks_i$ denote the number of chopsticks taken by i^{th}

philosopher.
All of these variables need to be updated whenever a philosopher `pickUp` or `putDown` chopstick(s).
To avoid multiple threads to update the variables `numChopStickAvail` or `numPhiloWith2Chop` at the same time, we have to make mutex variables, for example locks or semaphores, for both of these quantities.

Q.5 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**	

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.
b. If a request from thread T4 arrives for (2, 2, 2, 4), can the request be granted immediately?
c. If a request from thread T2 arrives for (0, 1, 1, 0), can the request be granted immediately?
d. If a request from thread T3 arrives for (2, 2, 1, 2), can the request be granted immediately?

Solution:

a. In the given state of the system `Need` matrix can be calculated by

```
Need = Max - Allocation
```

	Need	Available
	A B C D	A B C D
T_0	3 3 3 2	2 2 2 4
T_1	2 1 3 0	
T_2	0 1 2 0	
T_3	2 2 2 2	
T_4	3 4 5 4	

Threads can execute in the following sequence:

i. Need(T_2) <= Available, therefore T_2 will execute.
   After execution, Available = (4, 6, 3, 7)
ii. Need(T_3) <= Available, therefore T_3 will execute.
   After execution, Available = (8, 7, 4, 7)
iii. Need(T_0) <= Available, therefore T_0 will execute.
   After execution, Available = (11, 8, 8, 8)
iv. Need(T_1) <= Available, therefore T_1 will execute.
   After execution, Available = (13, 9, 8, 10)
v. Need(T_4) <= Available, therefore T_4 will execute.
   After execution, Available = (15, 11, 10, 11)

Therefore, the order is <T_0,T_1,T_2,T_3,T_4> and the system is in safe state.

b. $Request_4 = (2, 2, 2, 4)$.

$Request_4 <= Need_4$

$Request_4 <= Available$

$Available = (2, 2, 2, 4) - (2, 2, 2, 4) = (0, 0, 0, 0)$

Now, no thread can execute since $Need_i > Available \; \forall \; i$.

Therefore, the system goes into an unsafe state. So, request cannot be granted immediately T_4 must wait for $Request_4$.

c. $Request_2 = (0, 1, 1, 0)$

$Request_2 <= Need_2$

$Request_2 <= Available$

$Available = (2, 2, 2, 4) - (0, 1, 1, 0) = (2, 1, 1, 4)$

$Allocation_2 = (2, 4, 1, 3) + (0, 1, 1, 0) = (3, 5, 1, 3)$

$Need_2 = (0, 1, 2, 0) - (0, 1, 1, 0) = (0, 0, 1, 0)$

Threads can execute in the sequence $< T_2, T_0, T_1, T_3, T_4 >$.

d. $Request_3 = (2, 2, 1, 2)$

$Request_3 <= Need_3$

$Request_3 <= Available$

$Available = (2, 2, 2, 4) - (2, 2, 1, 2) = (0, 0, 1, 2)$

$Allocation_3 = (4, 1, 1, 0) + (2, 2, 1, 2) = (6, 3, 2, 2)$

$Need_3 = (2, 2, 2, 2) - (2, 2, 1, 2) = (0, 0, 1, 0)$

Threads can execute in the sequence $< T_3, T_1, T_0, T_2, T_4 >$.