CS 350: Homework #9

Due 3PM, Thursday, April 14

Kristel Tan (ktan@bu.edu)

April 6, 2016

Question 1

Part a

Given the allocation matrix, we know that 7 units of R1, 2 units of R2, and 5 units of R3 have been allocated. The maximum number of units for each resource as seen in the claim matrix for 9 for R1, 5 resources for R2, and 3 for R3. Because 5 units of R3 are already being used, we know that its maximum usage cannot actually be 3, but instead at least one more than is currently allocated (5+1 = 6). This is because the system would never reach a safe state otherwise. Therefore, we can hypothesize that the minimum total amounts for resources R1, R2, and R3 are 9, 5, and 6 respectively.

To test this hypothesis, we can distribute the resources to the processes to fulfill them as seen below:

Claim Matrix

	R1	R2	R3
P1	7	5	3
P2	3	2	2
P3	9	0	2
P4	2	2	2
P5	4	3	3

Allocation Matrix-1

	R1	R2	R3
P1	0	1	0
P2	2	0	0
P3	3	0	2
P4	2	1	1
P5	0	0	2

Resource Units Left: (2, 3, 1)

Allocation Matrix-2

	R1	R2	R3
P1	0	1	0
P2	2	0	0
P3	3	0	2
P4	2	2	2
P5	0	0	2

Resource Units Left: (2, 2, 0) —> Finish P4 (add back resources) —> (4, 4, 2)

Allocation Matrix-3

	R1	R2	R3
P1	0	1	0
P2	3	2	2
P3	3	0	2
P5	0	0	2

Resource Units Left: (3, 2, 0) —> Finish P2 (add back resources) —> (6, 4, 2)

Allocation Matrix-4

	R1	R2	R3
P1	0	1	0
P3	9	0	2
P5	0	0	2

Resource Units Left: (0, 4, 2) —> Finish P3 (add back resources) —> (9, 4, 4)

Allocation Matrix-5

	R1	R2	R3
P1	7	5	3
P5	0	0	2

Resource Units Left: (2, 0, 1) —> Finish P1 (add back resources) —> (9, 5, 4)

Allocation Matrix-6

	R1	R2	R3
P5	4	3	3

Resource Units Left: (5, 2, 3) —> Finish P5 (add back resources) —>
✓ **(9, 5, 6) The system's resources are at a safe state and the hypothesis is correct.**

Part b

We can run through the same test as in part(a) to see if we should honor P2's initial resource request by checking if the system will be at a safe state after attempting to serve all processes.

Allocation Matrix-1-1

	R1	R2	R3
P1	0	1	0
P2	2	0	0
P3	3	0	2
P4	2	1	1
P5	0	0	2

Resource Units Left: (3, 3, 2)

Allocation Matrix-1-1-2

	R1	R2	R3
P1	0	1	0
P2	3	0	2
P3	3	0	2
P4	2	1	1
P5	0	0	2

P2 requests (1, 0, 2) —> Resource Units Left: (2, 3, 0)

Allocation Matrix-2-1

	R1	R2	R3
P1	0	1	0
P2	3	2	2
P3	3	0	2
P4	2	1	1
P5	0	0	2

Resource Units Left: (2, 1, 0) —> Finish P2 (add back resources) —> (5, 3, 2)

Allocation Matrix-3-1

	R1	R2	R3
P1	0	1	0
P3	3	0	2
P4	2	2	2
P5	0	0	2

Resource Units Left: (5, 2, 1) —> Finish P4 (add back resources) —> (7, 4, 3)

Allocation Matrix-4-1

	R1	R2	R3
P1	0	1	0
P3	3	0	2
P5	4	3	3

Resource Units Left: (3, 1, 2) —> Finish P5 (add back resources) —> (7, 4, 5)

Allocation Matrix-5-1

	R1	R2	R3
P1	7	5	3
P3	3	0	2

Resource Units Left: (0, 0, 2) —> Finish P1 (add back resources) —> (7, 5, 5)

Allocation Matrix-6-1

	R1	R2	R3
P3	9	0	2

Resource Units Left: (1, 5, 5) —> Finish P3 (add back resources) —>
✓ **(10, 5, 7) The system will be at a safe state after serving P2's request for (1, 0, 2) units first. Hence, this request should be granted.**

Part c

For this problem, we can pickup where P2 requests (1, 0, 2) from part(b) and then test to see if we should honor P1's following resource request by checking if the system will be at a safe state after attempting to serve all processes.

Allocation Matrix-2-2

	R1	R2	R3
P1	0	1	0
P2	3	2	2
P3	3	0	2
P4	2	1	1
P5	0	0	2

Resource Units Left: (2, 1, 0) —> Finish P2 (add back resources) —> (5, 3, 2)

Allocation Matrix-3-2

	R1	R2	R3
P1	0	3	0
P3	3	0	2
P4	2	1	1
P5	0	0	2

P1 requests (0, 2, 0) —> Resources Units Left: (5, 1, 2)

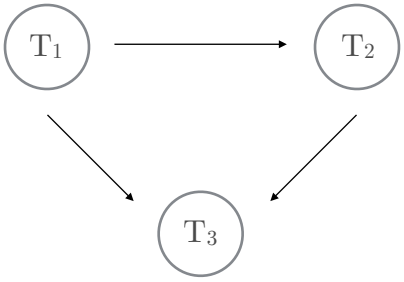
We know that P1 claims (7, 5, 3) resources. However, after granting P2's request and then giving P1 (0, 2, 0) resource units, we are left with (5, 1, 2) resource units. Because this is not enough to fulfill P1's claim, granting P1's request would not leave the system at a safe state. Therefore, we should not grant its request.

Question 2

We can draw a graph to represent the dependencies transactions have with each other for the validation attempts in this problem. If a transaction writes or reads a set that a transaction uses later, then the later transaction is dependent upon the previous one. To depict this relationship, we can draw an arrow from the supporting transaction to the transaction(s) that depend upon it.

Validation Attempt 1

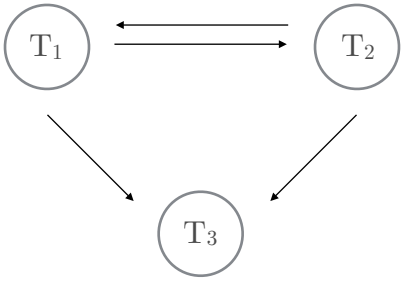
R1; **V1({A,B},{C})**; W1; R2;**V2({B,C},{D})**; W2; R3; **V3({C,D},{A})**; W3
For clarity, the validation attempt above can be rewritten as the following schedule:
R1(A); R1(B); W1(C);
R2(B); R2(C); W2(D);
R3(C); R3(D); W3(A);



Because there are no cycles in this graph, this validation should succeed and serializability can be guaranteed.

Validation Attempt 2

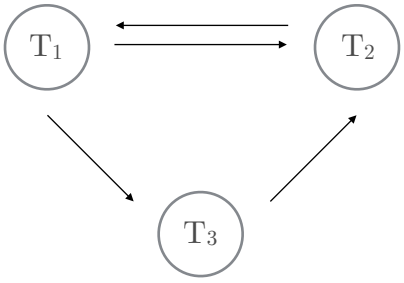
R1; R2; **V1({A,B},{C})**; W1; **V2({B,C},{D})**; W2; R3; **V3({C,D},{A})**; W3
For clarity, the validation attempt above can be rewritten as the following schedule:
R1(A); R1(B); R2(B);
R2(C); W1(C); W2(D);
R3(C); R3(D); W3(A)



Because there are no cycles in this graph, this validation should succeed and serializability can be guaranteed.

Validation Attempt 3

R1; R2; **V1({A,B},{C})**; W1; R3; **V2({B,C},{D})**; W2; **V3({C,D},{A})**; W3
For clarity, the validation attempt above can be rewritten as the following schedule:
R1(A); R1(B); R2(B);
R2(C); W1(C); R3(C);
R3(D); W2(D); W3(A)



Because there is a cycle in this graph, this validation should fail and serializability cannot be guaranteed.

Question 3

Part a

A deadlock can occur in the following scenario where we assume x is initially < 0 and transactions A and B proceed concurrently.

Transaction A	Transaction B
lock(x) requested —> granted	lock(z) requested —> granted
read(x)	read(z)
lock(z) requested —> denied	lock(y) requested —> granted
- -	read(y)
- -	lock(x) requested —> granted
- -	write(x)
deadlock	**deadlock**

Part b

```
Transaction A {
  read x;
  if x > 0 {
      write  y ;
  else {
      write  z ;
  }
}

Transaction B {
  write  x ;
  read z;
  read y;
}
```

Part c

The disadvantage of using this deadlock prevention approach occurs if transaction A and B start concurrently. This is because both transactions initially request a lock on x. However, only one will be granted the lock on x and the other must wait before it can proceed.

Question 4

- It is necessary that "when data on a customer's volume is being re-mirrored, access to that data is blocked until the system has identified a new primary (or writable replica)" for protection against customer data loss and "consistency of EBS volume data under all potential failure modes". In other words, while a customer's data is being accessed, we want to ensure that it does get overwritten, corrupted, or deleted.

- The EBS control pane became a victim of starvation when create volume API requests started to get backed up. At some point, the EBS control pane's threads became completely used up because of a re-mirroring storm that exhausted its available capacity. The cluster was then unable to service further requests once its threads were fully depreciated and the API calls were therefore starved.

- The leader election problem is reflected in Amazon's system when a node loses connectivity to another node which its data is being duplicated to because it has run out of space. In this case, the node must look for another node — or "leader" if you will — within the cluster to carry out its data replication. This process is referred to as "re-mirroring" in the article. "Re-mirroring storms" occurred when a large quantity of nodes had to look for a new leader under these conditions but became stuck because the cluster ran out of space. In order to prevent this from happening in the future, Amazon planned to implement a solution where the nodes that lost connectivity focused on re-establishing that connection with previous replicas rather than looking for new nodes to replicate its data to.

- The Amazon system designers intentionally enforced an upper bound on the level of concurrency when data for a volume needed to be re-mirrored. As a result, a race condition takes place in which negotiations between the EC2 instance, the EBS nodes with volume data, and the EBS control plane decide which copy of the data is designated as the primary copy. The benefit of this is that it provides strong consistency of EBS volumes. However, on the other hand, the race condition also caused EBS nodes to fail because the volume of negotiations was too high.

- A service that could have benefitted from a cap on the level of concurrency was the EBS control plane. The EBS cluster began failing to service requests because of its long time-out period. If there was a cap on its queue size, however, the queue would not have been as backed up with a large number of requests and the cluster could in turn prevent thread starvation.