Pooja Baba
#002677117
09.26.2022

CSC 8980
Distributed Systems Fall 2022

Homework #2

1. Imagine a very long bridge across the Mississippi River. A car will take upwards of 15 minutes to cross this bridge. Due to construction, the bridge has been reduced to a single lane that has to be shared by traffic in both the west-east and the east-west direction. It is obviously not possible to allow traffic in both directions simultaneously, so a special traffic control mechanism is installed with the following rules:

- An arriving car will have to wait if the bridge is currently occupied by one or more cars moving in the opposite direction.
- Multiple cars are allowed to cross the bridge in the same direction (otherwise there would be little bridge utilization and arriving cars would have to wait a long time).
- In order to avoid starvation, the entry of cars onto the bridge in one direction must be stopped after a batch of k cars has entered the bridge to allow traffic in the opposite direction if there are any cars waiting.
- If there are no cars, the bridge is open in both directions and the first arriving car will determine the direction of traffic.

Viewing each car as a process that is traveling in West-East (WE) or East-West (EW) direction, develop a MONITOR that implements the rules listed above. You MONITOR must use the monitor procedures **Enter_WE(), Enter_EW(), Exit_WE(), Exit_EW(),** to be executed when a car enters and exits the bridge.

Your solution must show all the necessary MONITOR variables and the condition variables and must not unnecessarily restrict vehicles to cross the bridge, must be deadlock free, and must not starve traffic in any direction.

ANS: *(Assumption – time considered in minutes.)*

The monitor goes as follows –

```
i.   Initialize the following –
     • QueueEW: cars running from East to West
     • QueueWE: cars running from West to East
     • BridgeCars: an array containing #cars currently on the bridge
     • k: maximum #cars that can be on the bridge at a time
     • trafficDirection: traffic direction set by the first car arriving in either of the queues.
ii.  While (True):
     a. whichQueue = Listen(QueueEW, QueueWE)
        b. !isEmpty(QueueEW)?trafficDirection = "EW":trafficDirection = "WE"
     c. If !isEmpty(whichQueue):
        i. While(BridgeCars.length<=k):
           1. currentCar = whichQueue.pop()
           2. BridgeCars.add(currentCar)
        ii. For car in range(BridgeCars):
           1. travelStartTime.append(startTimer())
           2. trafficDirection=="EW" ? Enter_EW(car) : Enter_WE(car)
           3. If ((currentTime()–travelStartTime[BridgeCars.indexOf(car)]) == 15):
              a. trafficDirection=="EW" ? Exit_EW(car) : Exit_WE(car)
              b. BridgeCars.remove(BridgeCars.indexOf(car))
              c. travelStartTime[BridgeCars.indexOf(car)]
     d. During steps a.) and c.), if the other Queue not under process gets a car, the cars
        in the other queue are put on hold till the BridgeCars get empty. Means till the
        current traffic on the bridge doesn't clear, the cars wait in the queue.
```

Pooja Baba
#002677117
09.26.2022

2. For a semaphore *s*, we define the following:
init[s]::= initial value of *s*
start_P[s] ::= the number of times P(s) has been started
end_P[s] ::= the number of times P(s) has been completed
end_V[s] ::= the number of times V(s) has been completed
A useful invariant is: *end_P[s] = min(start_P[s], init[s] + end_V[s])*
Now consider the abstract version of the bounded buffer problem: empty, full, mutex:
semaphore = (N, 0, 1);

```
Producer:                            Consumer: while(1)
  while(1) {                           {
    P(empty);                            P(full);
    get buffer from freelist;            get buffer from fullist;
    …. produce                           …. produce
    put buffer on fullist;               put buffer on freelist;
    V(full)                              V(empty)
  }                                    }
```

Prove that 0 ≤ end_P[empty] – end_P[full] ≤ N

PROOF:

The maximum value of empty is N whereas the minimum value of empty is 0.
Lly, the maximum value of full is N, and the minimum value of full is 0.

For the number of times the **producer** is going to **produce** and decrease the empty value, the **consumer** is going to **consume** for **less than or equal** to the former.

Meaning, that the difference between the 2 can never be greater than the buffer size, i.e., **N**

If the producer's production is **X**, the consumer can consume a **maximum** of **X** and cannot be **X+1.**

Therefore, the difference cannot be less than 0.

Hence, the given statement: **0 ≤ end_P[empty] – end_P[full] ≤ N** is true.

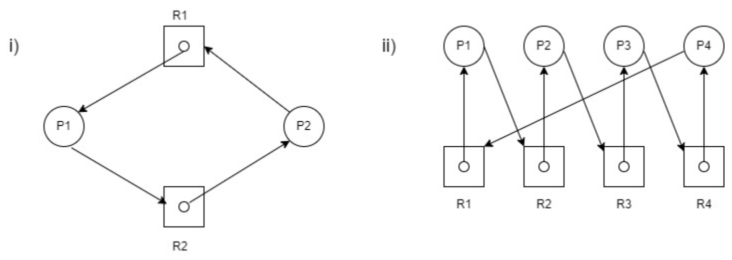
Pooja Baba
#002677117
09.26.2022

3. Proof the correctness or give a counter-example for each of the following statements. You must state whether the statement is true or false and then show your arguments.

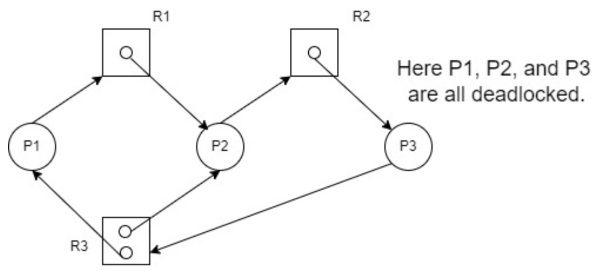
a. deadlock ➔ cycle
b. cycle ➔ deadlock
c. expedient & knot ➔ deadlock
d. deadlock ➔ knot

ANS:

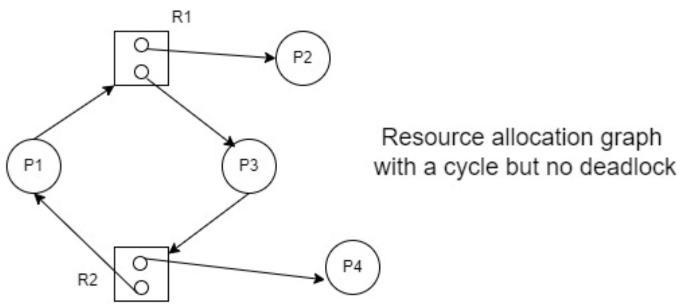
- *deadlock* -> *cycle*
  For a deadlock to occur following conditions **must** occur –
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait => this implies a cycle to be present in the graph
    Hence, every *deadlock* -> *cycle*
    Example-



- *cycle* -> *deadlock*
  If a graph contains cycle, a deadlock **may** exist.
  - If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred.

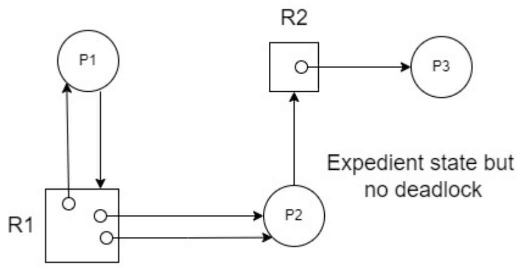


  - If each resource type has several instances, then a cycle does not necessarily imply a deadlock. The cycle is just a necessary but not a sufficient condition for a deadlock to occur.

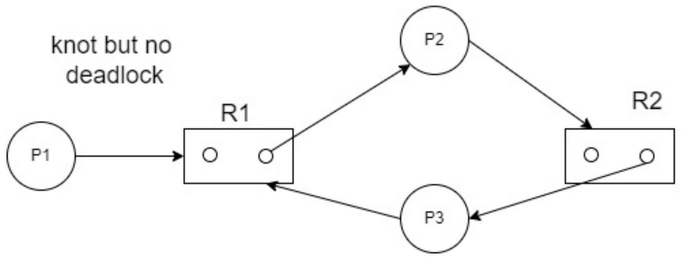


Pooja Baba
#002677117
09.26.2022

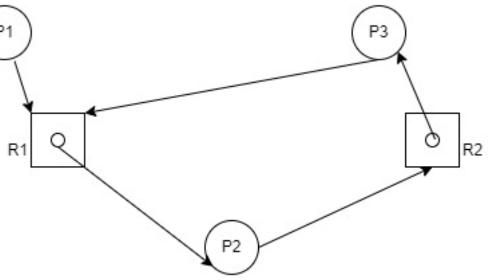
- *expedient & knot -> deadlock*
  Expedient state: All processes with outstanding requests that are blocked are said to be in the expedient state.



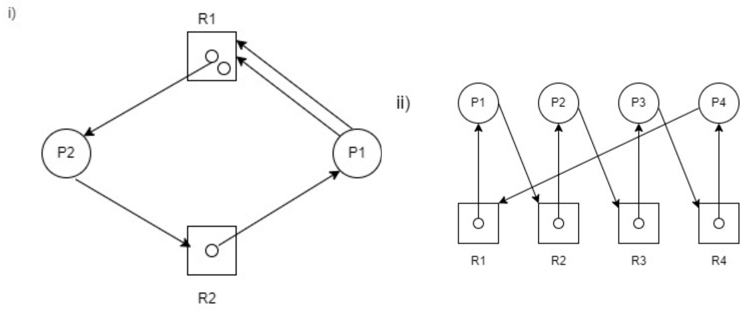
  Knot: k is a non-empty set of nodes with the property that for all nodes z in k is reachable(z) = k



  As we can see in the above examples, either of the conditions do not lead to deadlock. Both the conditions if they come together, the resultant is a deadlock. As shown in the below image.



- *deadlock -> knot*



In the above examples, the deadlock has been detected for the processes.
We know that the following 2 facts –
1. *knot -> cycle* 2. *deadlock -> cycle (proved above)*
Hence, we can say that *deadlock -> knot*

Pooja Baba
#002677117
09.26.2022

4. Consider a system with N blocks of storage, each of which holds one unit of information (e.g. an integer, character, or employee record). Initially, these blocks are empty and are linked onto a list called *freelist*. Three processes communicate using shared memory in the following manner:

Shared Variables: freelist, list-1, list-2: block (where block is some data type to hold items)

```
PROCESS #1
var b: pointer to type block;
while (1)
{
  b:= unlink(freelist);
  produce_information_in_block(b);
  link(b, list1);
}
```

```
PROCESS #2
var x,y: pointer to type block;
while (1)
{
  x:=unlink(list-1);
  y:=unlink(freelist);
  use_block_x_to_produce info_in_y(x, y);
  link(x, freelist);
  link(y, list-2);
}
```

```
PROCESS #3
var c: pointer to type block;
while(1)
{
  c:=unlink(list-2);
  consume_information_in_block(c);
  link(c, freelist);
}
```

Use simple semaphores to implement the necessary mutual exclusion and synchronization. **The solution must be deadlock-free and concurrency should not be unnecessarily restricted.**

ANS:

```
PROCESS #1
var b: pointer to type block;
while (1)
{
        P(freelist);
        P(list-1);
        b:= unlink(freelist);
        produce_information_in_block(b);
        link(b, list1);
        V(list-1);
        V(freelist);

 }
```

- We wait for the freelist using P(freelist). And for list-1 using P(list-1).
- Once the freelist is acquired by the process, required information is produced and using link it is added to the list-1.
- Once this is done, we list-1 and freelist are freed for other processes using V(list-1) and V(freelist) respectively.

Pooja Baba
#002677117
09.26.2022

```
PROCESS #2
var x,y: pointer to type block;
while (1)
{
        P(list-1);
        x:=unlink(list-1);
        V(list-1);
        P(freelist);
        P(list-2)
        y:=unlink(freelist);
        use_block_x_to_produce_info_in_y(x,y);
        link(x, freelist);
        link(y, list-2);
        V(list-2);
        V(freelist);
}
```

- We wait for the list-1 using P(list-1). And once acquiring it to fetch value we free it using V(list-1)
- Further we wait for freelist and list-2. Once they have been acquired by the process, further code executes.
- Post the execution, we free the list-2 and freelist using V(list-2) and V(freelist).

```
PROCESS #3
var c: pointer to type block;
while(1)
{
        P(freelist);
        P(list-2);
        c:=unlink(list-2);
        consume_information_in_b
        lock(c);
        link(c, freelist);
        V(list-2);
        V(freelist);
}
```

Process #3 works similar to Process #1 difference being, list-2.