CSE511 Project 1

Rafi-ur-Rashid(mur5028)

Syed Md Mukit Rashid(szr5848)

Implementation of Synchronizer

- We first *tokenize* the path expression. We convert the path expression string into a list of tokens, e.g., braces, parenthesis, comma, semicolon, keywords (such as ‘path’, ‘end’) and procedure/operation names.

- The path expression and thus the list of tokens are by default in infix notation. We convert it into postfix with the use of a stack of operators. Here operators include all tokens except for the keywords and procedure names, i.e., braces, parenthesis, comma, and semicolon.

- As soon as we get the token list in postfix notation, we build a tree data structure out of it. Let’s call this a path expression tree.

- The next step is to recursively parse this tree starting from the root and add the appropriate semaphore methods, i.e., P, V, PP, VV at the prologue/ entry and epilogue/exit of each path expression following the implementation detail provided in the paper of Campbell et al.

- While doing the recursive parsing, we maintain two hash tables. One for storing the entry and exit semaphores (i.e. the semaphore type, and calling arguments) determined to be called for each procedure through the path expression. Let’s call this *entry-exit hash table*.
- Another hash table is also used , for storing the initialized values for each semaphore and counter. Let’s call it *sema-counter hash table*.

- The ENTER_OPERATION(op_name) and EXIT_OPERATION(op_name) functions respectively retrieve the entry and exit semaphores for the given *op_name* from the *entry-exit hash table*. Then the initialization values for the counters and semaphores are also retrieved from the *sema-counter hash table*. Finally the semaphore methods are executed using the retrieved arguments from the entry-exit hash table.

Monkey Crossing Problem

We implement the following 3 path expression to solve this problem:

path {EastCrossing}, {WestCrossing} end
path ReqEastCrossing end
path ReqWestCrossing end

- Here, while building the entry and exit semaphores from the path expressions, when we see the *ReqEastCrossing* or *ReqWestCrossing* operation, we initialize a semaphore with 5. This type of semaphore allows 5 processes to pass it without blocking , but blocks the 6th process so that not more than 5 processes cross the critical section at the same time.
- The first path Expression ensures that no monkey crosses from west-east when there are already monkeys in the critical section passing in the other direction, and vice versa. One of them is selected (COMMA operation), and the brace operator ensures multiple of the same direction monkeys can cross, and when all of them are done crossing then the next selection happens and monkeys wanting to go to the other direction may cross, if selected.
- The *east_crossing* function is guarded by both ENTER_OPERATION (EAST_CROSSING)/EXIT_OPERATION(EAST_CROSSING) and ENTER_OPERATION (REQ_EAST_CROSSING) / EXIT_OPERATION (REQ_EAST_CROSSING) , the first ensures a guard on direction, the second ensures a guard on number of monkeys crossing, Similarly *west_crossing* is guarded as well.

Child Care problem

In the child care problem, we used the following path expression:

```
"path {CaregiverArrive ; CaregiverLeave} end "
                       "path {ChildArrive ; ChildLeave} end "
                       "path Lock end "
                       "path {V_G ; P_G} end "
                       "path {V_C ; P_C} end "
```

Here, there are 4 path expressions:

- The first two constraints that the number of *CaregiverLeave* cannot be more than the number of *CareGiverArrive*, and also the number of *ChildLeave* cannot be more than the number of *ChildArrive*
- The third path helps us provide a Lock to synchronize the access to the counters we are going to use

- The last two paths are basically gives us two semaphore locks initialized to 0, thus ENTER_OPERATION(P_G or P_C) corresponds to a sem_wait(P_G or P_C) and EXIT_OPERATION(V_G or V_C) will correspond to a sem_post(V_G or V_C). We use the P_G/V_G semaphore to guard that the last *CareGiver* not leave when there is any Child in the center, and the P_C/V_C semaphore to guard that no child enters the center when there is no Caregiver.

We also have 4 counters, initialized to 0:

```
int NumActiveCareGivers = 0;
int NumPendingLeavingCareGivers = 0;

int NumPendingChild = 0;
int NumActiveChild = 0;
```

Which counts the
- Number of active caregivers (CareGivers started but CareLeave not entered)
- Number of pending leaving caregivers (CareLeave entered but not started)
- Number of Pending Childs (ChildArrive entered but not started)
- NumActiveChild (ChildArrive started and ChildLeave not ended)

We assume that there would be no ChildArrive entering at or after the last CaregiverLeave has entered.

The high level overview of our algorithm is:
- ENTER_OPERATION(LOCK)/ENTER_OPERATION(LOCK) is used wherever we are accessing / modifying the 4 counters.
- After CareGiverArrive has exited, we increase the number of active caregivers. If it is 1 , or there are no active or pending childs, we perform an EXIT_OPERATION(V_G) which increases its semaphore count by 1. If there are more than 1 caregiver / there is 1 caregiver but no child then the caregiver would be allowed to leave.
- If NumActiveCareGivers == 1 && NumPendingLeavingCareGivers != 0 , we perform an EXIT_OPERATION(V_G). If a new CareGiver arrives, then any CareGiver waiting to leave can leave.
- Also, if any child is waiting to start since there were no caregiver, they are allowed to enter through performing NumPendingChild of EXIT_OPERATION(V_C) , if this so is the case.

This portion has to be done right after the CareGiverArrive has exited, That is why it is done after noting the exit of CareGiverArrive and before the EXIT_OPERATION (CAREGIVER_ARRIVE).

- After CareGiverLeave has entered, We update the number of active caregivers and number of pending leaving caregivers, and wait on the semaphore P_G. This semaphore is designed such that its value would be 0 if the context is such that there is only 1 caregiver and there are at least 1 pending / active child in the center.
- After coming out of semaphore P_G, we decrease NumPendingLeavingCareGivers by 1.

This wait has to be done before the leave process has started, but strictly after the CareGiverArrive has ended. That is why it is placed after before the timestamp is set but after ENTER_OPERATION(CAREGIVER_LEAVE).

- In ChildArrive, before starting we check if there are no active or pending childs but at least 1 caregiver, we perform ENTER_OPERATION(P_G). This sets the semaphore value to be 1 less than the number of active caregivers, and the last caregiver will be forced to wait on P_G.
- We also increase NumPendingChild by 1. Also if there is at least 1 active caregiver , we perform an EXIT_OPERATION(V_C), so that the child doesn't have to wait on the subsequent V_C lock, which bars childArrive to proceed if there is no active CareGiver.
- We next have ENTER_OPERATION(P_C) which bars childArrive to proceed if there is no active CareGiver.
- After coming out of semaphore P_C/V_C we update the number of NumPendingChild and NumActiveChild.

We have to do them before the ChildArrive has started (e.g. the waiting in case there are no caregiver), that is why it is done before ENTER_OPERATION(CHILD_ARRIVE).

- In ChildLeave, after exiting, we decrease NumActiveChild by 1. We also check if NumPendingChild == 0 && NumActiveChild == 0, we increase the P_G/V_G semaphore by 1. Since there are no active/pending child , this increase of 1 will set the semaphore value equal to the number of CareGivers, so all caregivers can leave through not blocking in the P_G semaphore in CareGiverLeave.

Since this allowance of caregiver leaving has to be done after the last child has left, this part is done after EXIT_OPERATION(CHILD_LEAVE).