Lab 2 report - Group 53 - Venkata Sai Dinesh Uddagiri, Souptik Paul

Data Structures

We have added a few data structures and variables in batch-scheduler.c, for the implementation of narrow bridge synchronisation problem, namely:-
int current_direction_thread - This variable is used for storing the direction of the data transfer.
struct semaphore bus_size - This semaphore is used for maintaining the number of slots in the bus.
struct semaphore send_priority_task_sema - This semaphore is used for high priority tasks which are waiting to be sent.
struct semaphore receive_priority_task_sema - This semaphore is used for high priority tasks which are waiting to received.
struct semaphore send_lowPriority_task_sema - This semaphore is used for low priority tasks which are waiting to be sent.
struct semaphore receive_lowPriority_task_sema- This semaphore is used for low priority tasks which are waiting to receive.
struct lock mutex_bus - This structure variable of lock type, is used to ensure mutual exclusion of the resource, pertaining to each thread.

Algorithms

In this implementation we are striving to solve the problem of narrow bridge, which is a problem of handling synchronisation issues that arises when scheduling different batches of jobs. To handle this problem we have made the implementation in the function, that were already defined for us, as mentioned below:-

Implementation in getSlot() function defined in 'batch-scheduler.c':

The getSlot() function takes a task as input. In this function first we check the priority of the task, followed by the direction of the task. Then we place the task in the respective semapore depending on the priority and direction. We achieve this by using the priority and direction attributes on the task structure. We firstly check for high priority tasks, using the condition, if (task.priority == HIGH), if the condition is true we check for the direction of the tasks(whether it is sending or receiving), using the condition if(task.direction == SENDER). Then based on the output of the condition we place the high priority tasks in their respective semaphores using the sema_up() function. If the condition of the statement if(task.priority==HIGH), was false, it is a low priority task hence, using the same set of conditions to check the direction and place the low priority tasks in their respective semaphores.

Subsequently, we start a while forever loop, to provide the resource to the tasks. Hence first we acquire the access rights for the lock, to the task using the lock_acquire() function. In this loop, first we check if the bus is free or if the bus is the direction of the respective task. If the condition is true, then we first check if there are any high priority tasks waiting for the resource. If this condition is also true, the task enter into the bus, in this case we also the direction of the bus, to the direction of the current task, so that other tasks in the same direction can use the bus(if space is available on the bus). Subsequently, we also reduced the space available on the bus, using the sema_down() function on the bus_size. Then we check if the process is a part of high priority processes or low priority processes and accordingly, decrement the respective semaphores using the sema_down() function. Then we release the lock of the task, using the lock_release() function. Finally we put the task(or thread) to sleep using the timer_sleep(), implemented in the previous lab, for allowing other tasks(or threads) to access the lock.

Implementation in transferData() function defined in 'batch-scheduler.c':

This function, takes a task as input. In this function, we indicating the process when a tasks has already requested for a resource. As the task is waiting for the transfer process, we are putting the tasks to sleep for a random time using the timer_sleep function, which we have updated in the previous lab.

Implementation in leaveSlot() function defined in 'batch-scheduler.c':

This function takes a task input and the slots in the bus after a task has left the bus. To implement this we are using the sema_up() function to increase the bus semaphore.

Synchronization

Using this implementation we were striving to solve the problem of handling synchronization issues that arises, when scheduling this batches of jobs. As per the requirement, the implementation ensures that no more than 3 tasks are using the bus in the same direction. To ensure this we are initialising the bus_slots semaphore to 3 in the statement sema_init(&bus_size,BUS_CAPACITY). This ensure no more than 3 tasks of the same direction can be present on the bus.

We also need to ensure that tasks from opposite directions are not using the bus simultaneously. To ensure this, when a thread enters in the critical section, we first check if the bus is free or if the task in bus is in the same direction. This ensure that only thread of the same directions can enter into the bus, if the bus is free. When a task in the opposite direction is available, the condition if current_direction_thread == task direction, will be false. We also check if there is tasks present on the bus and the number of tasks on the bus using the condition bus_size.value==BUS_CAPACITY. Hence in the event the conditions returns false, the task will not access the bus and eventually release the lock.

We also need to ensure that high priority tasks have to given access to bus, before the low priority tasks. To implement this we are using the condition task.priority ==HIGH or send_priority_task_sema ==0 and receive_priority__task_sema.value==0. This condition ensures that when accessing bus slots, we check that if there is high priority task waiting in the same direction(or any direction if the bus is empty), then we should give a slot to these tasks. Only if there are no high priority tasks, then we will give a slot to the low priority tasks. In this case, we also have to ensure that high priority tasks do not use the bus, while there are tasks on the bus which are in the opposite direction to that of the high priority task. To ensure this we have the condition bus_size.value==BUS_CAPACITY or current_direction_thread == task.direction, in place. This ensure that if a high priority tasks is available but in the opposite direction, to the tasks currently in the bus, it will not be given access to bus and releases the lock.

Rationale

In this implementation, the high and low priority tasks and handled very efficiently. In our implementation, we ensure that no more than 3 tasks will access the bus at any given time and also tasks from opposite direction will not enter the bus simultaneously. Our implementation also ensures that, we give high priority tasks privilege over low priority tasks such that if there is a high priority tasks which can enter the bus(in the same direction), then it will be allowed to enter the bus, however if the direction of the bus is different then even high priority tasks will not be allowed to enter the bus. Owing to these elements the solution adequately solves the problem of synchronization of batch processes very efficiently.