Project 3- Report – Ayushi Rajendra Kumar

Q4. Deadlock detection and avoidance.

Part1 and 2. Deadlock detection and avoidance- File- active_lock.c

1) al_initlock(al_lock_t *l)-This function is a system call that takes lock as input and initializes lock,guard,queue and priority of the lock to zero. It also checks if the number of locks is less than the max allowed locks

2) al_lock(al_lock_t* l) – spin lock on guard and in order to avoid indefinite due to different priorities, sleep for QUANTUM while spinning on guard. If the lock is not taken, the current process is given the lock and the proctable entry- pr_al_locks and pr_tot_locks of the process is updated. Also the owner of the lock is updated. If the lock is already taken by other process, before waiting on the lock, a check on deadlock needs to be made, therefore check_deadlock function is called. If it's a deadlock, it exits, else it waits in queue for the process to release the lock and parks/sleeps until then. Post parking/sleeping, it takes the ownership of the lock and updates the proctab entry for the lock.

3) al_unlock(al_lock_t* l) – If unlock is not called by the process owning the lock, syserror is thrown. Post spinning over guard, it removes the lock from the array in the proctable, and if queue is empty, it sets lock to zero else transfers the lock to next process in queue by calling unpark, which wakes up the process in the queue. Guard is finally set to zero.

4) bool8 al_trylock(al_lock_t* l) - it return to the calling function 1 if lock is already acquired, else it acquires the lock.

5) void al_park()- puts the process to sleep, and reschedules the process only if prpark flag of process is set.

6) void al_unpark(pid32 pid) -unparks the dequeued pids, i.e. awakens the sleeping process and inserts it to ready queue. If prpark of process is set, it resets it to 0.

7) void al_setpark()- sets the prp ark flag in process control block to 1.

8) pri16 al_restoreq(pid32 pid,qid16 q) – this function takes pid and queue as input and finds the max priority that is queued (this function is basically useful to update the priority field of the lock, which store the max of priorities from the processes in the waiting queue). This function is basically useful for next question and does not server relevant purpose here.

9) check_deadlock(pid32 pid, al_lock_t* l) –This function uses the processes waiting for the lock in the queue, iterates over the queue and for every process checks the lock that it currently holds, and recurses over the locks wait queue,until it reaches the empty wait queue or the current lock itself. Using recursion, it finds out if it ends up at the lock its started. If that's the case, it declares as deadlock and returns 1, else it returns 0. In lock function,when 1 is returned from check_deadlock, the process is returned with print message.

Test case- main_deadlock, to run deadlock detection , comment #define ACTIVE_TRY_LOCK, to run try_lock, case, comment #define ACTIVE_LOCK.

Test case1 for checking functionality of the code:

Test case1- to check single cycle deadlock with one process not being a part of the dead lock. The following cycle is tested. P1 → L1 implies Process 1 has acquired lock 1 and P1 -> L1 means process 1 is waiting for L1

```
P1→ L1- > P2 →L2- > P3 → L3 - > P4 → L4 -> P5 → L5 - > P6 → L6 - > P1
      |
    P7(waiting on L1)
```

On running the code for this test case, following result was expected,

Deadlock detected == P1-P2-P3-P4-P5-P6

Output – Deadlock detected = P6-P7-P8-P9-P10-P11 – as per OS process numbers, this is as per expected. A print of system error is printed as on deadlock detection, the process is not queued , therefore never acquires the lock. So when on deadlock, the control is returned and user tries to unlock the function, the code **prints system error.**

Test case 2: 2 cycles,

```
P1→L1- > P2 → L2 - > P3 → L3 - >P1     P4 →L4 - > P5 → L5 -> P6 →L6 - > P4
                  |
                P7 ( Waiting on L2)
```

Expected results – Deadlock detected=P1-P2-P3

Deadlock detected= P4-P5-P6

Output- Deadlock detected = P13-P14-P15

Deadlock detected = P17-P18-P19 , as per expectation

On try lock, deadlock should not occur. In main function, another method of increment that uses try lock is written. If a lock is already taken, it unlocks all the acquired lock, reverts the increments performed and tries again to acquire a lock after a certain delay.

Try lock is applied on test case1 and its noticed that deadlock doesn't occur. Therefore it is working as expected.

Question 5: Priority Inheritance

File- pi_lock.c

This lock has same functions as in deadlock with an additional function and slightly different implementation in lock and unlock.

2) pi_lock(pi_lock_t* l) – spin lock on guard and in order to avoid indefinite due to different priorities, sleep for QUANTUM while spinning on guard. If the lock is not taken, the current process is given the lock and the proctable entry- pr_pi_locks and pr_tot_locks of the process is updated. Also the owner of the lock is updated. If the lock is already taken by other process, then the process is enqueued, and the max priority amongst the processes waiting in queue is stored in l->prio by using restoreq function. If the highest priority in the waiting queue is greater than the current owner of the lock, the priority of the current process is changed by calling change_prio function call. Finally the process prepares for parks, sets guard to 0 and parks. Once back from park, it updates the current owner and updates the latests highest priority in l->prio, this flag always holds the highest priority that exists in the wait queue.

3) pi_unlock(pi_lock_t* l) – If unlock is not called by the process owning the lock, syserror is thrown. Post spinning over guard, it removes the lock from the array in the proctable. If the current priority is not equal to initial priority, then it checks the l->prio ,highest priority of other locks it holds if any and changes its priority to the highest priority in array of locks, if present, else, it restores to initial priority. For finding highest priority amongst the other lock it holds, it uses **restoreq_locks(pid) function**. If queue is empty, it sets lock to zero else transfers the lock to next process in queue by calling unpark, which wakes up the process in the queue. Guard is finally set to zero.

All other functions are same as described in deadlock – pi_park(),pi_unpark(pid), pi_setpark(),restoreq()

4) Change_prio(pid,newPrio)- This function changes the priority of the current process to the new priority, removes the process from the ready list and enqueues is back with new priority. To remove process from queue, it **uses removeq function**

5) restoreq_locks(pid,q)- this function is called in unlock, to find the highest priority l->prio amongst the other locks it holds. This is if the current process also holds other locks, we need to restore the priority to the highest priority of those locks, rather than the initial priority.

6) removeq(pid)- This function, removes the pid from the ready list.

--

Test case 1 : 3 threads are run, 1st thread being the lowest priority and 3rd thread being the highest priority. The threads are run such that, when the 2nd process enters, the 1st process has acquired lock L1 and L2, and when 3rd process entrers, P1 still has the lock L1 and L2.

1) pi_lock(L1)→ P1 PRIO1
2) pi_lock(L2)→ P1 PRIO1
3) pi_lock(L2) → P2 prio 2 (so that it waits for p1 to complete) , here it is expected that P1 inherits P2's prio, so prio of p1 should change from 1->2
4) pi_lock(L1) → P3 prio 3 here priority of P1 is expected to change from 2 to 3
5) pi_unlock(L2) → prio of P1 should go from 3 to 2 since P2 is also waiting for a lock held by p1
6) pi_unlock(L1) → prio of P1 should go from 2 to 1 since no more processes waiting for this lock.

   Expected- priority-changed= P1 :: 1-2
   priority-changed= P1 :: 2-3
   priority-changed= P1 :: 3-2
   priority-changed= P1 :: 2-1

The output was same as expected.

Test case 2: 4 threads with different priorities. P1- has L1,L2 P2 requests L2 , P3 requests L1,P4 requests L1,L2

1) pi_lock(L1) → P1
2) pi_lock(L2) → P1
3) pi_lock(L2) → P2 (expect p1 prio 1->2)
4) pi_lock(L1) → P3 (expect p1 prio 2->3)
5) pi_lock(L1) → p4 (expect p1 prio 3->4)
6) pi_unlock(L2) → p1 (expect p1 prio 4->2)
7) pi_lock(L2) → p4
8) pi_lock(L2) → p2 (expect p2 prio 2->4)
9) pi_unlock(L2)→ p2 (expect p2 prio 4->2)

Expected: priority-changed=P1 :: 1-2

Priority-changed=P1:: 2-3

Priority-changed=P1::3-4

Priority-changed=P1::4-2

Priority-changed=P1::2-1

Priority-changed=P2::2-4

Priority-changed=P1::4-2

The results were as expected on running.

All outputs are present in /temp/testcases folder for reference.