Operating Systems 2 (CS3523)

Theory Assignment

SURAJ TELUGU

CS20BTECH11050

- 1.sol

In the increment function discussed in book, let there be process P assume it had a context switch or interrupt after storing temp as v and v may have been incremented in another process making the process P unable to terminate. Therefore, many processes such as P will **remain waiting and never terminate**.

To encounter the above problem, we need to ensure that once a certain process enters the do – while loop no other process can enter the loop. Do – while loop section of this function is the **critical section** of the problem. We can use lock **with compare_and_swap, test_and_set functions, mutex locks, binary sophomores** and make the all the other processes wait while one process running the critical section.

Binary semaphore is used below which serves as lock and ensures mutual execution, as wait and signal maintain a queue for processes there would not be any starvation. All the processes **will eventually terminate if they are mutually exclusive**.

```
void Modified1_increment (atomic_int *v)
{
    semaphore *lock_sem;
     lock_sem ->value = 1;
    int temp;
    wait(&lock_sem);
   do {temp = *v;}
   while (temp != (compare_and_swap(v,temp,temp+1));
   signal(&lock_sem);
}
```

➢ 2.sol

The Following is an alternative solution in which neither the reader nor the writers will starve. In this solution we remove the starvation problem by introducing another mutex lock implemented using a semaphore in_mutex, process having access to this mutex lock can enter the workflow and therefore, have access to the resource. This implements a check to readers that come after writers as all the processes are pushed into queue of the semaphore in_mutex. Thus, this algorithm is starve-free.

Initialisation:

```
Semaphore *in_mutex, *out_mutex,*write_sem;

in_mutex->value = 1; out_mutex->value = 1; write_sem->value = 0;

int readers_started = 0, readers_completed = 0; // Changed by different semaphores

bool writer_waiting = false; // Indicates if a writer is waiting
```

Reader

```
do{
      // Entry Section
      wait(in_mutex);
      readers_started++;
      signal(in_mutex);
      / ***** Critical Section (Reading is done) *****/
      // Exit Section
       wait(out_mutex);
       readers_completed++;
   if(writer_waiting && readers_started == readers_completed) { signal(write_sem); }
   signal(out_mutex);


   // Remainder section


}while(true);
```

```
Writer
do{
   // Entry Section
   wait( in_mutex);
   wait(out_mutex);
   if(readers_started == readers_completed) { signal(out_mutex); }
  else{
      writer_waiting = true;
      signal(out_mutex);
      wait(writer_sem);
      writer_waiting = false;
   }
   / ***** Critical Section (Writing is in progress) *****/
  // Exit Section
  signal(in_mutex);
  // Remainder Section
}while(true)
```

Mutual Exclusion

In the above solution, in_mutex ensures mutual exclusion among all the processes. The out_mutex implements mutual exclusion for the variables readers_completed and writer_waiting. The in_mutex serves another role of ensuring the mutual exclusion for the variable readers_started. Furthermore, the semaphore write_sem ensures mutual exclusion among the readers and the waiting writer.

Bounded Waiting

In both the methods, before entering the critical section, all the processes must pass through in_mutex which stores all the waiting processes in a FIFO data structure. So, for a finite number of processes, the waiting time for any process in the queue is finite or bounded.

Progress Requirement

Both the algorithms have such a structure that the systems cannot enter a state of deadlock. As long as the time of execution is finite for all the processes in their critical sections, there will be progress as the processes will keep on executing after waiting in the queue of in_mutex.

References:

https://arxiv.org/ftp/arxiv/papers/1309/1309.4507.pdf

https://en.wikipedia.org/wiki/Readers%E2%80%93writers_problem

- 3.sol

```
void lock_spinlock (int *lock) {
{
   while (true) {
      if (*lock == 0) {
      /* lock available */
       if (! compare_and_swap(lock, 0, 1))
        break;
      }
   }
}
```

Yes, The Above idiom lock_spinlock works appropriately for implementing spinlocks.

If lock equals 0, process is in unlocked state the compare_and_swap function returns 0 which breaks the infinite while loop allowing the process to continue.

If lock equals 1, process is in locked state the loop acts as infinite loop and the process waits in the loop while the next processes with lock 0 can continue.

If any context switch or interrupt has occurred which made a process lock equals to 1 just after entering the if statement the compare_and_swap will take care of it since its an atomic function by returning 1 and locking the process in the infinite loop.

Therefore, in any case the lock_spinlock function works same as original spinlock, the context switches and interrupts does not affect its purpose.

- 4.sol

Let the above getValue function has been used in Bounded Buffer Problem's solution using semaphores let the producer function be as below:

```
int n; (n = Maximum number of processes in buffer)
semaphore mutex = 1, empty = n, full = 0;
while (true) {
  /* produce an item*/
  if (getValue(&empty) > 0) wait(empty);
  wait(mutex);
  /* add produced item to the buffer */
  signal(mutex);
  signal(full);
}
```

The getValue function is added into bounded buffer problem's solution creates unbounded waiting making some processes to wait continuously and never terminated. For example, let there be 3 processes P1, P2, P3 and the buffer size be 1.

Whenever process P1 checks the value of semaphore some process P2 might be adding produced item into buffer having empty equals 0 the process P1 does not execute if it produces item slowly then another process P3 which produces item faster than process P1 can execute making P1 to wait.

Therefore, processes which produces items slowly tend to wait for a very long time to get into buffer while all the faster processes execute continuously and may start executing gain if they are periodic **making the producer not bounded waiting**.

So, using getValue function is discouraged because it makes process to wait unboundedly.