1  Homework 4 on Concurrency

Problems 1 and 2: Peterson’s algorithm and a false mutex solution for the producer/consumer problem.

It is critical for CSI400, with its emphasis on concurrency, that you are able to analyze concurrent programs by interleaving their basic statements in various ways. If you didn’t get full credit on problems 1 or 2, see me (or the TA) for patient tutoring on how to get this right. Get this straight now, before the final!

Each of the erroneous variants cannot deadlock, since the only case of a deadlocked state is that both processes are in their repeatedly run their while loop with the condition evaluating to true each time and the values of the variables are unchanging. But this is impossible because the value of (shared variable) turn cannot be 0 and 1 simultaneously.

Some people used the following neat notation for expressing the solution to problem 2. Each row represents on step of computation. The column position each unique non-empty row entry indicate which process executed that step.

<table>
<thead>
<tr>
<th>Step</th>
<th>Producer</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>produce_item()</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>down(&amp;mutex)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>down(&amp;empty)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>down(&amp;full)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>down(&amp;mutex)</td>
<td></td>
</tr>
</tbody>
</table>

3: Tanenbaum ch.2 prob. 21

Peterson’s solution requires preemptive scheduling. Why?

4: Tanenbaum ch.2 prob. 22

(Implementing enter_region with atomic swaps.) The hardware is built so TSL or atomic swap instructions are atomic even in multiprocessor systems. The reason for using them is to implement mutual exclusion between processes running on different processors in a multiprocessor system. On a uniprocessor system, mutual exclusion can be achieved by disabling interrupts (provided the OS doesn’t command I/O devices to access lock variables).

5: Tanenbaum ch.2 prob. 24

(Implementing counting semaphores with binary semaphores.) This is a very subtle problem. Nobody got it completely! It asks you to write subroutines (called up() and down() below to function as the given operations on a counting semaphore, where you are allowed only to use some number of binary semaphores plus an ordinary shared integer variable within the subroutine bodies.

The first step towards a solution is to make use of one binary semaphore to enforce mutual exclusion between critical sections inside each subroutine. Hence the binary semaphore mutex should be initialized to 1. We also use the shared integer variable c to hold the count value. Thus we begin: The easy part is make the counting semaphore do the right thing when it down() should not sleep nor up() cause wakeups.
down: down(&mutex);                    up: down(&mutex);
  if ( C > 0 )                        if ( C >= 0 )
    {
      C--;                            C++;
      up(&mutex);                     up(&mutex);
      return;                         return;
    }
  else                                else
  {
    ...                            ...
    up(&mutex);                     up(&mutex);
    return;                         return;
  }

One hard part is to program these subroutines so that if down() is called when C is zero, our down() blocks, waiting for another thread to call our up(). The solution also must provide that when multiple up() calls are executed during a period when several threads are blocked in our down() subroutine, the number of blocked threads that are awakened (which causes their call to down() to return) is equal to the number of times up() was called.

The best solution avoids busy waiting. To do that, we must take advantage of the capacity of binary semaphores to block.

Some insights are (1) to use negative values of C to represent the number of threads currently blocked in down(), waiting for an up; (2) use another binary semaphore (named “wakeup”) to synchronize the wakeup of exactly one thread blocked in down() with each event that up() is called; (3) make down() release the mutex before waiting, so other threads can enter the critical sections of our up() and down() functions. So, our next try is:

down: down(&mutex);                    up: down(&mutex);
  if ( C > 0 )                        if ( C >= 0 )
    {
      C--;                            C++;
      up(&mutex);                     up(&mutex);
      return;                         return;
    }
  else                                else
  {
    // C <= 0, we should block..         // C < 0, so somebody is sleeping
    C--;                                C++;
    up(&mutex);                         up(&mutex); //?????
    down(&wakeup); //Wait for a wakeup.
    up(&wakeup); //Allow other down or ups
    ....                                ....
    return;                             return;
  }

This “solution” has a bug. Suppose first a set A of several threads call down(), so C becomes \(-2\). Following that, suppose another set B of more than one thread all call up(). All but one thread in B blocks in the down(&mutex) call. The one that didn’t block (so it got the mutex) calls up(&wakeup) and then reaches the //????? statement which unlocks mutex. The (perversely) scheduler might then choose another thread from B to run next. This will cause up(&wakeup) to be called a second time without an intervening down(&wakeup)....

The problem is wakeup is a binary semaphore. It cannot count higher than 1! Therefore, only one of the threads in A which called down() will be awakened!

The solution is quite tricky. We move the mutex release operation from the up() function to the down() function. That way, our semaphore system remains in its critical section after calling up(&wakeup) until one thread waiting on down(&wakeup) is awakened; exactly one thread is so awakened and it releases mutex. Here’s the final code:
down: down(&mutex);
if( C > 0 )
{
    C--;
    up(&mutex);
    return;
}
else
{ // C <= 0, we should block..
    C--;  //Add us to count of waiters.
    up(&mutex);  //Allow other down or ups
    down(&wakeup);  //Wait for a wakeup.
    up(&mutex);  //Moved from up()
    return;
}

up: down(&mutex);
if( C >= 0 )
{
    C++;
    up(&mutex);
    return;
}
else
{ // C < 0, so somebody is sleeping
    C++;
    up(&wakeup);
    //DO NOT release mutex., let
    //the waiter do it.
    return;
}

6: Tanenbaum ch.2 prob. 48
It is easy to implement message passing functions that use a single buffer or the unbuffered copy strategy outlined in the 3rd paragraph on page 123. The semaphore would simply provide mutual exclusion for the buffer. That is good enough for a simple solution.
A better solution is to make use of the semaphore based buffered producer-consumer solution. send(item) uses the producer() code from page 112 except item is the message send is called with. In the same way, receive(&item) written like consumer() except it returns the item instead of “consuming” it.

Homework 5 on Concurrency

Tanenbaum ch. 2 prob. 38
We can solve this problem by analyzing what the CPU is used for between the successive schedule() operations. First, the CPU runs schedule() and functions it calls, including the context switch, for a total CPU time usage of S. Second, a process runs until its quantum Q expires (time used is Q) or it blocks for I/O; the latter happens in time T on the average. What causes that process to stop running and the next schedule() operation to be started depends on which of these events occurs first. So, between the successive schedule() operations,

\[ U = \text{CPU time for useful work} = \min(Q, T) \]

\[ \text{Total CPU time} = S + U \]
\[ \text{CPU utilization} = \frac{\text{CPU time for useful work}}{\text{Total CPU time}} \]

You could now write down formulas for this for each of the given cases.

Tanenbaum ch. 2 prob. 39
Most people realized the jobs should be run the order depending on their expected run times, so the job with expected time X should be inserted into the order according to this. But a few people got the order (shortest-job-first) backwards!

Tanenbaum ch. 2 prob. 40
Nobody realized that in case (a), round robin fair share scheduling in a multiprogramming system with all the jobs submitted at once, every job turnaround waits until all the jobs are completed! This common turnaround time is 30 minutes, so the average is 30 minutes, not 30/5! Imagine if an auto service center operated this way. The technicians will be kept busy all the time which is excellent for getting the most work done with the given payroll, but the customers will all be equally unsatisfied.
For all the other parts, there are two equally correct ways to calculate the answer: For part (c) for example,

\[
\frac{(10) + (10 + 6) + (10 + 6 + 2) + (10 + 6 + 2 + 4) + (10 + 6 + 2 + 4 + 8)}{5}
\]

\[
\frac{(5 \times 10) + (4 \times 6) + (3 \times 2) + (2 \times 4) + (1 \times 8)}{5}
\]

**Tanenbaum ch. 2 prob. 41-44**

Straightforward, most people got full credit. For 43, the given times are the actual measured times the process ran before it blocked or was preempted. The predicted time is calculated by the OS, can be calculated by you using the formula of page 146, and is used by the OS merely to help choose which ready process to schedule.