University of Newcastle
School of Electrical Engineering and Computer Science

COMP2240 - Operating Systems
Workshop 5 - Solution
Topics: Concurrency: Mutual Exclusion and Synchronization

1. The COMP2240 Teaching Assistant holds office hours twice a week in his office. His office can hold 2 persons: *1 TA* and *1 student*. Outside his office are 4 chairs for waiting students. If there are no students waiting to see the TA, the TA plays Minesweeper. If a student arrives at the TA's office and the TA is playing Minesweeper, the student knocks at his door and the TA invites the student in and begins helping him. If a student arrives at the TA's office and the TA is busy with another student, the student waits in a chair outside the TA's office until the TA is free. If the arriving student finds all the chairs are occupied, then he leaves.
Using semaphores, write two process, `student_i` and `TA`, that synchronize access to the TA's office during his office hours. These processes will have approximately the following structure:

```
Process TA
  Loop
   <Entry protocol to
synchronize with a student>
    <Advise a student>
  <Exit protocol>
  end loop
end TA
```

```
process student_i
  <Entry protocol to
synchronize with the TA>
  <Get advice or leave>
  <Exit protocol>
end studenti
```

(Note that although you are writing one student process; assume multiple instances of the process are active simultaneously.).

Answer:
This solution uses binary semaphores for students to wait on and for the TA to play minesweeper on.

```
var newStudent : binary_semaphore := 0
TAisFree : binary_semaphore := 0
mutex : binary_semaphore := 0
numWaiting : integer := 0
```

```
process TA begin
   loop
      down(newStudent)
       /* Play Minesweeper */
      down(mutex)
      numWaiting -= 1
      up(TAisFree)
      up(mutex)
      <advise student >
   end loop
end TA
```

```
process student_i begin
   down(mutex)
   if (numWaiting < numChairs) then
      numWaiting += 1
      up(newStudent)
         /* knock door */
      up(mutex)
      down(TAisFree)
         /* Sync with TA */
      <get advice >
   else
      up(mutex)
        /* Leave without advice */
   endif
end student_i
```

2. When a special machine instruction such as *compare_and_swap* or *exchange* is used to provide mutual exclusion, there is no control over how long a process must wait before being granted access to its critical section. Devise an algorithm that uses the *compare_and_swap* instruction but guarantees that any process waiting to enter its critical region will do so within *n* - 1 *turns*, where *n* is the number of processes that may potentially require access to the critical section and a *turn* is an event consisting of one process leaving the critical section and another being granted access.

Answer:

```
{Common Data Structures}
     var  waiting: array[0..n-1] of boolean;
          lock: boolean;  /* 0: false, 1: true */
{per process Data Structures}
     var  j: 0..n-1;
          key: boolean;
     while (true){
       waiting[ i ] := true;
       key := true;

       while (waiting[ i ] && key){
           key := (compare_and_swap(lock,0,1) == 0);
       }
       waiting[ i ] := false;

       <critical section>

       j := i + 1 mod n;
       while (j != i  && !waiting[j] )
           j := j + 1 mod n;

       if (j == i)   lock := false;
       else  waiting[ j ] = false;
                   /* will make the busy waiting while
                              loop false for the process j */
       <remainder section>
     }
```

The common data structures are initialised to false. When a process *i* leaves its critical section, it scans the array `waiting` in cyclic ordering (i+1, i+2, ..., n-1, 0, 1, ..., i-1). It designates the first process *j* in this ordering that is in the entry section (`waiting[j] = true`) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within *n* - 1 turns.

3. Show counting semaphore and binary semaphores have equivalent functionality.

Answer:

A procedure to implement counting semaphores by using binary semaphores is as follows:

```
S : Counting semaphore
C : Integer counter
S1 : Binary semaphore for updating C
```

```
S2 : Binary semaphore waiting for C to become positive
S3 : Binary semaphore for serialization
wait : wait procedure for binary semaphore
signal : signal procedure for binary semaphore
```

```
// wait procedure for counting semaphore
WAIT(S):
{
   wait(S3);
   wait(S1);
   C=C-1;
   if(C<0)
   {
      signal(S1);
      wait(S2);
   }
   else
      signal(S1);
   signal(S3);
}
```

```
// signal procedure for counting semaphore
SIGNAL(S) :
{
   wait(S1);
   C=C+1;
   if(C<=0)
      signal(S2);
   signal(S1);
}
```

4. Demonstrate that monitors and semaphores are equivalent insofar as they can be used to implement the same types of synchronization problems.

Answer:

A semaphore can be implemented using the following monitor code:

```
monitor semaphore {
   int value = 0;
   condition c;
   semaphore increment() {
      value++;
      c.signal();
   }

   semaphore decrement() {
      while (value == 0)
         c.wait();
      value--;
   }
}
```

A monitor could be implemented using semaphores in the following manner. Each condition variable is represented by a queue of threads waiting for the condition. Each thread has a semaphore associated with its queue entry. When a thread performs a wait operation, it creates a new semaphore (initialized to zero), appends the semaphore to the queue associated with the condition variable, and performs a blocking semaphore decrement operation on the newly created semaphore. When a thread performs a signal on a condition variable, the first process in the queue is awakened by performing an increment on the corresponding semaphore.

5. A bakery has decided to automate the task of production of cakes one department at a time. The first department that is intended for automation is the packing department. The packing department comprises the three main units, namely, the testing unit, the packing queue and the packing

machine. Once the cakes are baked, they are sent to the packing department. In the packing department, the packing operation is to be carried out by a fully automated system. The details of the proposed packing procedure are as follows:

a. On arriving at the packing department, each cake is inspected by the cake-testing machine.
b. If the tester identifies a cake as damaged, it discards the cake.
c. If the cake is fine, it is then placed in the packing queue.
d. The packing queue maintains a first-in-first-out order.
e. The maximum size of the packing queue is set to 20.
f. The packing machine picks the first cake from the packing queue, puts it in a packet, and seals it.

A problem of synchronization has been identified in the implementation of this procedure: there will be a possible disparity between the speed at which the tester places a cake on the packing queue; and the speed at which the packing machine picks a cake up from the packing queue; and the status of one machine cannot always be communicated with the other. The main mode of communication between the tester and the packer is the packing queue. Both the tester and the packer are aware of the number of cakes that are in the packing queue at any given time.
Devise a synchronization protocol that allows both the tester and the packer to operate concurrently at their respective speeds in such a way that cakes are packed in the exact order in which they are placed in the packing queue (in a FIFO manner).
[*Hint:* You may think of it as a producer/consumer problem]

Answer:

The following synchronization protocol can be used for the required purpose:

```
constant int  maxQ=20;
int           in = 0, out = 0;
cake          c;/* declare a temporary variable that denotes a cake */
cake          pqueue[maxQ];
                        /*declare packing queue that can store cakes*/
```

```
void tester()
{
   while(true)
   {
      c = get_cake();
      t = test_cake(c);

/* checks if cake is all right */
      if(t=="No")
      {
         discard(c);
         continue();
      }


     while((in + 1) % maxQ ==
     out){
      /* do nothing if the packing
queue is full */ ;
      }

      pqueue[in]= c;
   /* keep the take in packing Q*/
```

```
void packer()
{
   while(true)
   {
      while(in == out){
      /* do nothing if queue is
empty */ ;
      }

      c = pqueue[out];
      /* take a cake out of the
      queue */
      out = (out + 1 ) % maxQ;

      pack_seal(c); /* packs and
      seals the cake */
   }
}


void main()
{
   parbegin (tester, packer);
```

```
        in = (in + 1) % maxQ;
    }
}
```

```
}
```

Supplementary problems:

S1. Is busy waiting always less efficient (in terms of using processor time) than a blocking wait? Explain?

Answer:

In general, yes.

But consider an event that always occurs with a few cycles of the action that triggered it (such as an I/O operation that simply reads a control register, but performs no actual I/O).

In the time to perform a semaphore wait, block, and then subsequently release the blocked process, the event will have happened. More cycles are wasted in using the semaphore than if we had simply performed a busy wait, so in this case, busy waiting is more efficient.

S2. The Linux kernel has a policy that a process cannot hold a *spinlock* while attempting to acquire a semaphore. Explain why this policy is in place.

Answer:

You cannot hold a spin lock while you acquire a semaphore, because you might have to sleep while waiting for the semaphore, and you cannot sleep while holding a spin lock.

S3. A multithreaded web server wishes to keep track of the number of requests it services (known as **`hits`**). Consider the two following strategies to prevent a race condition on the variable hits. The first strategy is to use a basic mutex lock when updating hits:

```
int hits;
mutex_lock hit lock;

hit_lock.acquire();
hits++;
hit_lock.release();
```

A second strategy is to use an atomic integer:

```
Atomic_t hits;
Atomic_inc(&hits);
```

Explain which of these two strategies is more efficient?

Answer:

The use of locks is overkill in this situation. Locking generally requires a system call and possibly putting a process to sleep (and thus requiring a context switch) if the lock is unavailable. (Awakening the process will similarly require another subsequent context switch.) On the other hand, the atomic integer provides an atomic update of the hits variable and ensures no race condition on hits. This can be accomplished with no kernel intervention and therefore the second approach is more efficient.

S4. Windows Vista provides a lightweight synchronization tool called **slim reader–writer** locks. Whereas most implementations of reader–writer locks favor either readers or writers, or perhaps order waiting threads using a FIFO policy, *slim reader–writer* locks favor neither readers nor writers, nor are waiting threads ordered in a FIFO queue. Explain the benefits of providing such a synchronization tool.

Answer:
Simplicity. If RWlocks (reader-writer locks) provide fairness or favor readers or writers, there is more overhead to the lock. By providing such a simple synchronization mechanism, access to the lock is fast. Usage of this lock may be most appropriate for situations where reader–locks are needed, but quickly acquiring and releasing the lock is similarly important.

S5. Explain why Windows, Linux, and Solaris implement multiple locking mechanisms. Describe the circumstances under which they use *spin locks*, *mutex locks*, *semaphores*, *adaptive mutex locks*, and *condition variables*. In each case, explain why the mechanism is needed.

Answer:
These operating systems provide different locking mechanisms depending on the application developers' needs.
Spinlocks are useful for multiprocessor systems where a thread can run in a busy-loop (for a short period of time) rather than incurring the overhead of being put in a sleep queue.
Mutexes are useful for locking resources.
Solaris 2 uses *adaptive mutexes*, meaning that the mutex is implemented with a spin lock on multiprocessor machines.
Semaphores and *condition variables* are more appropriate tools for synchronization when a resource must be held for a long period of time, since spinning is inefficient for a long duration.

S6. Explain why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems.

Answer:
Spinlocks are not appropriate for single-processor systems because the condition that would break a process out of the spinlock can be obtained only by executing a different process. If the process is not relinquishing the processor, other processes do not get the opportunity to set the program condition required for the first process to make progress. In a multiprocessor system, other processes execute on other processors and thereby modify the program state in order to release the first process from the spinlock.