University of Newcastle
School of Electrical Engineering and Computer Science

COMP2240 - Operating Systems
Workshop 5
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.).

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.

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

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

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]

Supplementary problems:

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

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.

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?

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.

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.

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