Zagazig University
Faculty of Engineering
Computer and Systems Dept.

Spring 2020
Third Year
CSE325: Operating Systems

Discussion #6

Covers: Chapter 6

Questions:

1. The following pseudocode illustrates the basic push() and pop() operations of an array-based stack. Assuming that this algorithm could be used in a concurrent environment, answer the following questions:
   a. What data have a race condition?
   b. How could the race condition be fixed?

```
push(item) {
      if (top < SIZE) {
            stack[top] = item;
            top++;
      }
      else
            ERROR
}

pop() {
      if (!is empty()) {
            top--;
            return stack[top];
      }
      else
            ERROR
}

is empty() {
      if (top == 0)
            return true;
      else
            return false;
}
```

2. Consider how to implement a mutex lock using the compare and swap() instruction. Assume that the following structure defining the mutex lock is available:

```
typedef struct {
      int available;
} lock;
```

The value (available == 0) indicates that the lock is available, and a value of 1 indicates that the lock is unavailable. Using this struct, illustrate how the following functions can be implemented using the compare and swap() instruction:

- `void acquire(lock *mutex)`
- `void release(lock *mutex)`

Be sure to include any initialization that may be necessary.

3. The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P0 and P1, share the following variables:

```
boolean flag[2];
int turn;
```

The structure of process Pi (i == 0 or 1) is shown below. The other process is Pj (j == 1 or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

```
do
{
      flag[i] = true;
      while (flag[j])
      {
            if (turn == j)
            {
                  flag[i] = false;
                  while (turn == j)
                        ; /* do nothing */
                  flag[i] = true;
            }
      }
      /* critical section */
      turn = j;
      flag[i] = false;
      /* remainder section */
} while (true);
```

4. The following algorithm shows how to use the test_and_set instruction to implement the critical section problem for *n* processes (consider that the code is for process Pi). The common data structures used by the algorithm are:

```
boolean waiting[n];
boolean lock;
```

The elements in the waiting array are initialized to false, and lock is initialized to false. Show that the solution satisfies the bounded waiting requirement.

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* remainder section */

} while (true);
```