# NEW YORK UNIVERSITY ABU DHABI
## Review Session (Spring Semester) 2024

**Date of Examination:** 3-4-24 **Session(FN/AN)** FN/AN **Duration** $3-\epsilon$ **hr, Marks = 20**

**Subject Name :** Operating Systems **Subject No :** CS UH Operating Systems 3010

**Department/Centre/School :** Computer Science

1. In 1960's Dijkstra posed the concurrency problem to his friends, and one of them, a mathematician named Theodorus Jozef Dekker, came up with a solution. Unlike the solutions we discussed in the class, which used specialized hardware and also OS support, Dekker's algorithm uses just loads and stores. This was later modified by Peterson, and is also called as Peterson's lock.

   The description is as follows:

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

void init(){
//indicate you intend to hold the lock with 'flag'
flag[0]=flag[1]=0;
//whose turn is it? (thread 0 or 1)
turn=0;
}

void lock(){
  //'self' is thr threadid of the caller
  flag[self]=1;
  //make it other thread's turn
  turn=1-self;
  while((flag[1-self]==1)&&(turn==1-self))
     ; //spin-wait while it is not your turn
}

void unlock(){
   //simply undo your intent
   flag[self]=0;
}
```

   Answer the following questions:

   (a) Explain how Peterson's lock preserves Mutual Exclusion of the critical section?

   (b) Prove bounded waiting, that is there exists a bound or limit on the number of times that other threads are allowed to enter the critical section after a thread has requested to enter the critical section and before that request is granted.

   (c) However, Peterson's solution does not work on modern computers which allow reordering of loads and strore instructions which does not have dependencies. Explain the same.

2. What is the meaning of the term busy waiting? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.

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

4. In a system with threads, is there one stack per thread or one stack per process when user-level threads are used? What about when kernel-level threads are used? Explain.

5. Consider a computer with two processes, H, with high priority, and L, with low priority. The scheduling rules are such that H is run whenever it is in ready state. At a certain moment, with L in its critical region, H becomes ready to run (e.g., an I/O operation completes). H now begins busy waiting, but since L is never scheduled while H is running, L never gets the chance to leave its critical region, so H loops forever. This situation is sometimes referred to as the priority inversion problem. Does the same problem occur if round robin scheduling is used instead of priority scheduling? Discuss. Can you suggest any other alternative solution?

6. Study the following assembly code:

```
.var flag
.var count

.main
.top

.acquire
mov  flag, %ax      # get flag
test $0, %ax        # if we get 0 back: lock is free!
jne  .acquire       # if not, try again
mov  $1, flag       # store 1 into flag

# critical section
mov  count, %ax     # get the value at the address
add  $1, %ax        # increment it
mov  %ax, count     # store it back

# release lock
mov  $0, flag       # clear the flag now

# see if we're still looping
sub  $1, %bx
test $0, %bx
jgt .top

halt
```

   The code implements locking with a single memory flag. Can you explain the assembly? When can it go wrong?

7. Download from brightspace the files x86.py, a simulator which allows you to see how different thread interleavings either cause or avoid race conditions. See also the README for details on how the program works. Run the above code on the simulator for the following cases:

   (a) Can you predict the value of count at the end with a single thread? Use the -M and -R flags to trace variables and registers, and turn on -c flag to see their values.

   (b) Change the %bx register with -a flag (like `-a bx=2,bx=2` if you are running two threads). Can you predict the response?

   (c) Now set bx to a high value for each thread, and use the -i flag to generate different interrupt frequencies. What values lead to bad outcomes?

8. Now consider the following code:

```
.var mutex
.var count

.main
.top

.acquire
mov  $1, %ax
xchg %ax, mutex     # atomic swap of 1 and mutex
test $0, %ax        # if we get 0 back: lock is free!
jne  .acquire       # if not, try again

# critical section
mov  count, %ax     # get the value at the address
add  $1, %ax        # increment it
mov  %ax, count     # store it back

# release lock
mov  $0, mutex

# see if we're still looping
sub  $1, %bx
test $0, %bx
jgt .top

halt
```

Try to understand the code, which uses the `xchg` instruction to build a simple locking primitive. Now run the code, changing the value of the interrupt interval again. Make sure the loop runs for a good number of times. Does the code always work as expected? Does it sometimes lead to an inefficient use of the CPU?