CS 219 Spring 2024: Quiz 2

(5 questions, 30 marks, 15% weightage)

Name:________________________ **Roll number:**________________________

1. **[6 marks]** Consider a toy synchronization problem, with N students and one instructor conducting an exam. The students start entering the exam hall at different times. The instructor waits for all N students to arrive (assume all N will arrive) and be seated before distributing the question paper. After all students receive the question paper, the instructor announces the start of the exam. The students do not begin the exam until this announcement. Assume the students and instructor are represented as threads in a program, and we will use locks and condition variables to achieve the desired synchronization between them.

   Given below is the code for the instructor thread, which uses one mutex, two counts (initialized to 0), and 4 condition variables (variable names beginning with `cv_`). Do not change this code.

```
//instructor thread
lock(mutex)
if(count_students_arrived < N) wait(cv_instructor1, mutex)
signal_broadcast(cv_students_arrived)
unlock(mutex)
//start distributing question paper
lock(mutex)
if(count_students_qp < N) wait(cv_instructor2, mutex)
signal_broadcast(cv_students_qp)
unlock(mutex)
//announce start of exam
```

   Complete the code shown below for the student threads. You must not use any new variables, beyond those given above, in your solution.

```
//arrive in exam hall
lock(mutex) //complete code below




unlock(mutex)
//start receiving question paper
lock(mutex) //complete code below




unlock(mutex)
//start exam
```

2. **[6 marks]** Consider a multi-threaded program with N threads. The threads in the program start at different times. Each thread calls the function `togetherWeStart()` when it starts. This function should block the first $N-1$ threads. When the N-th thread arrives and calls this function, it should return immediately, and it should also unblock the previous $N-1$ threads that are waiting. In other words, when all threads in the program call this function at the beginning of their execution, it will ensure that all the threads return from this function roughly at the same time. You must write code for the function `togetherWeStart` below. You may assume that N is defined as a global constant. List the global variables you will need to use in your code below, along with their initial values (if any).

```
//global variables shared across all threads
```

Use the variables you defined above, and write the code for `togetherWeStart` below.

```
void togetherWeStart(void) {



}
```

3. (a) **[3 marks]** Consider a simple implementation of a Last-In-First-Out stack data structure, consisting of struct nodes, each containing a pointer "next" to the next element. The variable "top" points to the top of the stack. The implementation of the push function is shown below. Note that this implementation is not thread-safe, and will be incorrect when invoked by multiple threads in a program concurrently.

```
push(struct node *n) {
  n->next = top;
  top = n;
}
```

Suppose two threads push into the stack concurrently. Thread T1 pushes node n1, and T2 pushes n2. Describe one example of a concurrent execution of the threads that results in an incorrect result at the end due to a race condition. You must list down the order in which the code statements of each thread are executed in this interleaved execution, and also show the incorrect state of the stack at the end of the execution.

(b) **[3 marks]** Now consider this updated implementation of the push function, where the top pointer is updated using the hardware atomic instruction Compare-And-Swap (CAS). Recall that the arguments to CAS are a pointer to the variable to be updated, the expected old value, and the new value. CAS updates the variable and returns true if the old value matches the expected old value provided. Else, it returns false without updating the variable.

```
push(struct node *n) {
  do
    n->next = top;
  while(!CAS(&top, n->next, n);
}
```

Does this new implementation of push work correctly when the threads execute concurrently? Assume no other form of mutual exclusion is used. Justify your answer by using the example of the previous part, where two threads T1 and T2 try to push nodes n1 and n2 on to the stack respectively. Use an example interleaving of the code statements from this new code, and explain whether a race condition occurs or not.

4. **[6 marks]** Consider a process P that has made a blocking action (e.g., reading from an empty pipe, or waiting for a running child to terminate) in xv6. As a result, P switches itself out, the scheduler thread runs, finds a ready process Q in the `ptable` array, and switches to it. Given below are some events that occur during the context switch, given in jumbled order.

(A) P acquires `ptable.lock`

(B) P acquires pipe lock

(C) P releases `ptable.lock`

(D) P releases pipe lock

(E) Q releases `ptable.lock`

(F) Q releases pipe lock

(G) P calls the function `sleep`

(H) P calls `sched`, and from that function, `swtch`

Given below are two examples of blocking actions performed by P. In each scenario, list the events that occur during the context switch in chronological order, from earliest to latest. List only those events that are relevant to the particular scenario, e.g., the events pertaining to pipe lock may not be relevant in a scenario without pipes. Write your answer in the format “A, B, C, D, ...”.

(a) P reads from an empty pipe

(b) P calls wait while its child process is still running

5. **[6 marks]** Answer True/False for each of the following questions. Please note that you will get 1 mark for each correct answer, and negative marks (-1 mark) for each wrong answer. However, the negative marks from this question will not spill over to other questions, i.e., you will not be awarded a total of less than 0 marks for this question. There is no need to provide any justification for your True/False answer.

   (a) When a thread in a userspace program acquires a userspace spinlock (e.g., a pthreads spinlock), interrupts on the core on which the thread is running are disabled.

   (b) When a process in kernel mode acquires a kernel spinlock in xv6, interrupts on the core on which the process is running are disabled.

   (c) When a thread T1 in a userspace program acquires a userspace spinlock (e.g., a pthreads spinlock) on one core, and another thread T2 starts spinning for the same spinlock on another core, then a deadlock will always occur.

   (d) When a process P1 in kernel mode acquires a kernel spinlock in xv6 on one core, and another process P2 in kernel mode starts spinning for the same spinlock on another core, then a deadlock will always occur.

   (e) Two threads in a userspace program (e.g., created using the pthreads API) must always use locks to avoid race conditions when accessing shared global variables, whether running concurrently on the same core, or in parallel on multiple cores.

   (f) Two threads in a userspace program (e.g., created using the pthreads API) must use locks to avoid race conditions when accessing shared global variables only when running on multiple cores, and not when running on a single core system.