1. Consider a kernel design for an SMP (symmetric multiprocessor) system. There are two design choices for the queue of ready processes in the kernel. The kernel could maintain separate ready queues for each processing core, and schedule processes on a core from its local ready queue. Or, the kernel could have a common ready queue across all cores and run a process that is in the common queue on whichever core is free.

(a) Provide one advantage of separate ready queues over a common queue.
(b) Provide one advantage of a common ready queue over separate queues.

Ans:

(a) Cache locality. No contention for the common queue.
(b) Better load balancing across cores.

2. Consider the pseudocode to acquire and release a lock, based on the test-and-set atomic instruction.

```c
acquire():
    while(1) {
        while(lock == 1);
        if(test_and_set(&lock) == 0) return;
    }

release():
    lock = 0;
```

Consider the scenario where multiple threads of an application contend to use this lock across multiple CPU cores.

(a) Is this lock a spinlock? That is, do contending threads consume CPU cycles while waiting to acquire the lock?
(b) Explain what cache coherence traffic is generated when a new thread calls the acquire function when a thread on another core holds the lock.
(c) Explain what cache coherence traffic is generated when a thread holding the lock releases the lock, when there are threads on other cores waiting for the lock.
(d) What is the cache coherence traffic on the system bus when multiple threads are contending for a lock that is held on another core, in period when there are no new acquires or releases?

**Ans:**

(a) Yes, this is a spin lock (test and test and set lock).

(b) When a node tries to acquire a lock already held, it gets a copy of the lock variable in shared state in its cache.

(c) When a thread releases the lock and changes its value to 0, the cached copies in all other cores are invalidated and all cores read the new value. Once other threads on other cores realize that the lock is free, they all perform test and set, causing the cache line to bounce across cores. Eventually, once some thread acquires the lock, all the other contending threads realize that their test and set has failed, and go back to waiting on the read-only copy of the lock variable.

(d) No cache coherence traffic is generated during the wait period.

3. Consider a lockfree stack discussed in class, with the following code for push and pop.

```c
push(node *n) {
    do
        n->next = top;
    while(!CAS(&top, n->next, n);
}
	node *pop {
    node *result;
    while((result=top) != NULL) //line X
        if(CAS(&top, result, result->next)) //line Y
            break;
    return result;
}
```

Consider the following series of operations made by 3 threads of an application on the stack.

- **Thread 1 performs** push (aaa); push (aa); push(a); push (b); push (c);
- **Thread 2 performs** x = pop();
- **Thread 3 performs** y = pop(); z = pop(); push(y);

Show the state of the stack (the nodes and all the pointers) after the following scenarios occur.
(a) The three threads run sequentially one after the other.
(b) Thread 1 runs first, and after it completes, threads 2 and 3 execute concurrently. More specifically, all operations of thread 3 run after thread 2 has executed line X of its pop function, has computed the arguments to the CAS function in line Y, but before the call to CAS has happened in thread 2.
(c) Thread 1 runs first, and after it completes, threads 2 and 3 execute concurrently like part (b). However, now, thread 2 runs in between the execution the first pop function invocation of thread 3.

Ans:
(a) The stack has c, b, a, aa, aaa.
(b) After thread 3 runs, the stack has c, a, aa, aaa. After thread 2 runs, the stack (incorrectly) has b, a, aa, aaa. This is an example of the ABA problem, where the CAS statement in thread 2 gets stale pointers but does not detect the problem, because the top variable is the same.
(c) This scenario does not result in the ABA problem because top has changed. The stack correctly has b, aa, aaa.

4. Consider multiple contending threads trying to acquire a spinlock based on the ticket lock implementation studied in class. All contending threads share two integers, next_ticket and now_serving, both initialized to 0. Further, my_ticket is a variable that is local to every contending thread. Below is the code invoked by a thread to acquire a spinlock.

```c
my_ticket = fetch_and_increment(&next_ticket); //incr and return old val
while(now_serving != my_ticket); //busy wait
```

(a) Write the corresponding code to release the spinlock.
(b) Consider an alternative implementation of the acquire function shown below.

```c
my_ticket = next_ticket;
fetch_and_increment(&next_ticket);
while(now_serving != my_ticket); //busy wait
```

Is this implementation of acquire correct? Answer yes/no, and justify your answer.

Ans:
(a) now_serving++; or fetch_and_increment(&now_serving); Note that an atomic increment is not strictly necessary since only one thread (the one holding the lock) will attempt to release the lock at a time.
(b) No, this is not a correct implementation, because a thread could be interrupted between its first two lines, and another acquire call can proceed concurrently, leading to multiple threads getting the same ticket number before it is incremented.
5. Consider a simple lock-free stack studied in class, where nodes are pushed and popped from the top of the stack. The code for the push and pop functions is shown below. The variable top is a global variable that points to the node at the top/head of the stack.

```c
push(node *n) {
    do {n->next = top;}  
    while(!compare_and_swap(&top, n->next, n);  
}
node *pop {
    node *result;
    while( (result=top) != NULL)
        if(compare_and_swap(&top, result, result->next))
            break;
    return result;
}
```

Now, consider an alternative implementation of push, called push1, shown below. This alternative implementation is based on the `swap` atomic instruction. The implementation of pop remains unchanged.

```c
push1(node *n) {
    node *old_top = swap(&top, n);  
    n->next = old_top;
}
```

You are told that the workload consists of multiple threads issuing push and pop calls on the stack concurrently. However, no thread frees the memory of any element popped. Further, no thread pushes back an element that it has previously popped. Given this workload, is the above implementation of push1 correct? If yes, explain why. If not, come up with one example scenario where this implementation yields incorrect results.

**Ans:** No, this implementation is not correct. Suppose a thread calls push1(node N1), and it has swapped top to point to the new node N1. However, it has not updated the next pointer of N1 to point to the old top yet. Now, if the push1 code is interrupted by the pop, the pop function will find the new node N1 at the top but will find its next pointer set to NULL, and will incorrectly infer that the stack has only one node N1. Upon popping N1, it will set top to NULL, which is clearly wrong.

6. Consider an application with two threads, a producer and a consumer. The threads are pinned to different CPU cores if the application is run on a multicore system. The producer thread dynamically allocates memory from the heap for items it produces, and passes the pointers to the consumer thread. The consumer processes the data and frees the memory pointer. Recall that the blowup of a dynamic memory allocation scheme is defined as the ratio of the amount of memory consumed by the allocation scheme in the multicore scenario to the memory consumed
when running on a single core. Compute the blowup of various memory allocation schemes described below, when the system is running this particular application workload only.

(a) The memory allocation scheme uses a single heap per process, shared across all cores. Threads on different cores use locks to ensure mutual exclusion on the heap.

(b) The memory allocation scheme uses per-core heaps. A thread allocates memory from the heap of the core on which it is executing. Memory freed by a thread is returned to the heap of the core running the thread. Free memory is not exchanged between the per-core heaps.

(c) Same as part (b) above, except that memory deallocated by a thread is returned to the heap from which it was allocated.

Ans:

(a) Blowup is 1. The memory consumed is the same in single core and multicore cases because a common heap is used.

(b) Blowup is infinity, as the producer heap allocates memory but never frees it.

(c) Blowup is 1 again, as memory is allocated and returned to the producer’s heap.

7. Write down the code to acquire a simple spinlock using the following atomic instructions. You may assume that an integer variable lock represents the spinlock being contended for. You must atomically set this variable to 1 to acquire it, and the lock is set to 0 when released.

(a) compare and swap
(b) swap or its x86 equivalent xchg

Ans:

(a) while(!compare_and_swap(&lock, 0, 1);
(b) while(swap(&lock, 1) != 0);