Lab 9 Questions

Question 1

(a) How Peterson’s Lock Preserves Mutual Exclusion

Explanation: Peterson’s lock mechanism ensures mutual exclusion by using two primary variables: an array `flag[]` and a variable `turn`. Here's how it works:

1. Each thread sets its respective `flag[self]` to 1 to indicate its intention to enter the critical section.
2. It then assigns `turn` to the other thread (i.e., `turn = 1 - self`), indicating that it's the other thread's turn.
3. The thread enters a spin-wait loop, waiting as long as both conditions are true: the other thread also wants to enter the critical section (`flag[1-self] == 1`) and it's the other thread's turn (`turn == 1-self`).

Mutual exclusion is preserved because a thread can only enter the critical section if it's its turn and the other thread doesn't intend to enter the critical section. This ensures that at most one thread can enter the critical section at any given time.

(b) Proof of Bounded Waiting in Peterson's Algorithm

Explanation: Bounded waiting is ensured in Peterson’s algorithm through the use of the `turn` variable. Here's the proof:

1. When a thread sets `flag[self] = 1`, it indicates its desire to enter the critical section and hands over the turn to the other thread (`turn = 1 - self`).
2. If the other thread is not interested in entering the critical section (`flag[1-self] = 0`), the waiting thread can immediately enter the critical section.
3. If the other thread is also interested (`flag[1-self] = 1`), the waiting thread can enter the critical section only when `turn` comes back to `self`, ensuring that the other thread cannot continuously monopolize the critical section without giving turn back to the waiting thread.

This mechanism ensures that there is a limit to the number of times the other thread can enter the critical section before the waiting thread gets its turn, thus proving bounded waiting.

(c) Why Peterson’s Solution Does Not Work on Modern Computers

Explanation: Peterson's solution relies on the sequential execution of load and store operations as per program order. However, modern processors and compilers may reorder instructions for optimization and efficiency (out-of-order execution), especially when there are no explicit dependencies between them. Such reordering can break the assumptions made in Peterson’s algorithm about the order of execution of the `flag[]` and `turn` assignments and checks. For instance, the assignment to `turn` might be reordered after the beginning of the spin-wait loop, leading to both threads entering the critical section. Memory barrier instructions are necessary to prevent such reordering, but Peterson's algorithm does not include them.

Question 2

Explanation: Busy Waiting: Busy waiting occurs when a process repeatedly checks for a condition (such as a lock release) without relinquishing CPU control, thereby consuming CPU cycles without doing any productive work.

Other Kinds of Waiting:

- Blocking/Sleeping Wait: Instead of consuming CPU resources, the process is suspended and placed in a waiting queue until the condition is met.
- I/O Waiting: A process waits for an I/O operation to complete.

Busy waiting can sometimes be avoided by using blocking or sleeping mechanisms, where the thread or process is put to sleep and only wakes up when it can proceed, thereby saving CPU cycles. However, for very short wait times, the overhead of context switching associated with blocking might make busy waiting more efficient.

Question 3

Single-Processor Systems: In single-processor systems, spinlocks are generally inefficient because if a thread holds a spinlock and is pre-empted, no other thread can run and release the lock until the first thread is scheduled again.

Multiprocessor Systems: In multiprocessor systems, spinlocks are more suitable because while one thread is busy waiting, other processors can continue running other threads, potentially including the one that will release the lock. This makes spinlocks efficient for short durations where the overhead of locking primitives is higher than the wait time.

Question 4

User-Level Threads: All threads share the same stack because they are not recognized by the kernel.

Kernel-Level Threads: Each thread has its own stack since they are managed by the kernel.

Question 5

In a priority scheduling system, a high-priority process can end up busy waiting indefinitely if a lower-priority process holds a needed resource but cannot run due to the scheduler's rules. This is known as priority inversion.

Round Robin Scheduling: In round-robin scheduling, each process gets a fair amount of CPU time in a cyclic manner, which can prevent a high-priority process from starving lower-priority processes. However, issues might still arise if time slices are very short and context switching is frequent.

Solution: One common solution to priority inversion is the use of priority inheritance, where a lower-priority thread inherits the priority of a higher-priority thread if it holds a lock needed by the higher-priority thread. This temporarily boosts the lower-priority thread’s priority, allowing it to complete its work and release the resource.

Question 6

Explanation of Operation

Lock Acquisition: The program starts by attempting to acquire a lock. It repeatedly checks if the flag is 0 (lock is free). If flag is not 0, it continues to spin in the .acquire loop (busy waiting). Once the flag is 0, it sets flag to 1 to claim the lock. Critical Section: After acquiring the lock, it reads the current value of count, increments it by 1, and then writes it back. Lock Release: It sets flag back to 0, effectively releasing the lock.

Loop Control: The program decrements the BX register (which presumably controls the number of iterations) and continues if BX is still greater than zero. When BX reaches zero, the program halts.

Potential Issues

Busy Waiting: The .acquire section employs busy waiting, which can be highly inefficient on CPU resources as it keeps the CPU busy in a loop checking the lock status.

Lack of Protection Against Interruptions: If an interrupt occurs right after the lock is acquired but before it is released, it could lead to deadlock situations where the lock is never released.

Atomicity: The check and set of the flag variable are not atomic, meaning an interrupt or another thread could potentially modify the flag between the test and mov instructions.

Question 7

(a) Predicting count with a Single Thread

When the program is executed with a single thread, the behavior of the variable `count` can be accurately predicted since there is no concurrent modification. By assuming an initial value of 0 for `count` and a positive integer `N` for the register `bx`, the loop will decrement `bx` once per iteration until it reaches zero. As a result, the `count` variable will be incremented exactly `N` times.

For example, if `bx` is initialized to 10, the `count` variable will increment from 0 to 10 across 10 iterations of the loop.

(b) Modifying %bx for Two Threads

When modifying `%bx` using the `-a` flag to set it for two threads (e.g., `-a bx=2,bx=2`), the situation becomes more complex due to potential concurrency issues. However, if the setup does not actually support real concurrency (i.e., the threads are not truly running in parallel but are perhaps simulated sequentially in a single-threaded process), the outcome remains predictable.

Under this assumption, each "thread" would iterate two times. If executed sequentially, `Thread 1` would increment `count` twice, and then `Thread 2` would start and increment `count` two more times. Thus, `count` would increment a total of four times, ending up with the value of 4 if starting from 0.

(c) Using High bx Values and Varying Interrupt Frequencies (-i Flag)

With the following command interrupting at the critical section: `-p flag.s -t 2 -a bx=20,bx=20 -R bx -M 2000 -c -i 6`

Predictions:

- Concurrency Effects: With two threads both trying to increment `count` 20 times and interrupts occurring every 6 cycles, race conditions are expected due to the non-atomic management of the

`flag` variable used for locking.
- Final Value of `count`: Ideally, if each thread successfully increments `count` 20 times without any race conditions, the final value of `count` should be 40. However, due to possible race conditions and interrupts affecting the sequence of operations, the actual count could be less.

Testing Method:

- Simulate the Execution: Use an x86 emulator or a similar tool configured with the specified flags to track the execution flow, register values (`bx`), and memory modifications (`count`).
- Monitoring: Pay close attention to the register and memory traces for both threads, especially around lock acquisition and release points.

Observations:

- **Interrupt Handling:** Every 6 cycles, an interrupt might cause a thread to be preempted, potentially in the middle of the critical section. This could lead to both threads entering the critical section simultaneously, resulting in a race condition.
- **Lock Mechanism:** The simple binary flag used as a lock (`flag`) may not accurately reflect the lock state under high-frequency interrupts, raising doubts about its efficacy in handling access to the critical section.
- **Value of `count`:** Due to potential overlaps in critical section access, the actual value of `count` may not reach the expected value of 40. The precise timing of interrupts and thread execution resumption will heavily influence the result.
- **Potential Deadlocks or Starvation:** Poor alignment of timing and interrupts may lead to scenarios where one thread continually gets preempted when attempting to set the flag, potentially causing starvation or deadlocks if both threads end up in a waiting loop.

Question 8

Lock Acquisition

The lock acquisition process involves the following steps:

1. Load the value 1 into the `ax` register.
2. Perform an atomic exchange between `ax` and the mutex variable. This sets the `mutex` to 1 and moves the previous value of the mutex into `ax`.
3. Check if the previous value of `mutex` was 0 (indicating the lock was free).
4. If the `mutex` was not 0, indicating that the lock was held, jump back to the acquire step and spin-wait until the lock is free.

Critical Section

The critical section process involves the following steps:

1. Load the current value of count into `ax`.
2. Increment `ax` by 1.
3. Store the incremented value back in count.

Lock Release

The lock release process involves setting the `mutex` to 0, indicating that the lock is now free.

Loop Control

The loop control process involves the following steps:

1. Decrement `bx` by 1 (`bx` controls the loop iteration).
2. Test if `bx` is 0.
3. If `bx` is greater than zero, jump back to the top to repeat the loop.

Execution with Two Threads and Frequent Interrupts

With the following command: `-p test-and-set.s -t 2 -a bx=20,bx=20 -R bx -M 2000 -c -i 6` In this scenario, interrupts occur every 6 cycles. The following considerations apply:

- **Atomic Lock Acquisition:** The use of the `xchg` instruction ensures atomic lock acquisition and release, preventing simultaneous entry into the critical section by multiple threads.
- **Impact of Interrupts:** While interrupts during lock acquisition can lead to inefficiencies and increased CPU usage due to frequent context switches and spin-waiting, the atomic nature of the `xchg` instruction ensures the lock's consistency even if an interrupt occurs immediately after acquisition.
- **Inefficiency Concerns:** The busy-waiting loop (`jne .acquire`) can lead to inefficiencies, especially if one thread consistently acquires the lock faster than the other, potentially causing starvation. High interrupt rates can exacerbate this inefficiency.
- **Correctness and Mutual Exclusion:** Despite potential inefficiencies, the mutual exclusion property is maintained, ensuring correct incrementation of the shared variable count up to 40 times (20 times per thread) as long as the loop runs the required number of iterations without external interruptions or errors.