Part 1: Concurrency and Synchronization

Harry Hacker is a new programmer at Botch Technologies. His first assignment is to implement a standard reader/writer lock for multi-threaded programs on shared-memory multiprocessors.

Botch already has an implementation of the reader/writer lock for use within its Unix kernel for uniprocessors. This implementation is built using the following synchronization primitives:

```c
sleep(); /* block the calling thread */
wakeup(); /* wake up all blocked (sleeping) threads */
```

The kernel reader/writer lock is implemented as follows:

```c
integer readers = 0; /* count of active readers */
integer writers = 0; /* count of active writers */

/* Acquire the lock for reading. Multiple threads may hold the lock for reading, but only if no thread holds the lock for writing. */
procedure LockForRead ()
{
    while (writers > 0)
    {
        sleep();
        readers = readers + 1;
    }
}

/* Acquire the lock for writing. Only one writer may hold the lock at a time, and only when no readers hold the lock. */
procedure LockForWrite ()
{
    while ((readers > 0) or (writers > 0))
    {
        sleep();
        writers = writers + 1;
    }
}
```
/* Release a lock held for reading. */
procedure UnlockRead ()
{
    readers = readers - 1;
    wakeup();
}

/* Release a lock held for writing. */
procedure UnlockWrite ()
{
    writers = writers - 1;
    wakeup();
}

1. Botch’s existing implementation is acceptable because a thread is never preempted when it is executing kernel code. However, Botch’s code will behave incorrectly on a shared memory multiprocessor, even if the primitives are used as intended. Show this by outlining a sequence of events that will cause this implementation to fail.
2. Mr. Hacker’s first job with Botch will be to produce a better implementation of the reader/writer lock, based on mutexes (binary semaphores) and condition variables instead of \textit{sleep} and \textit{wakeup}. You will be asked to provide your own implementation in the next question. But first, list the operations for mutexes and condition variables, and explain their semantics. (If you are unfamiliar with mutexes and condition variables you may describe another general-purpose synchronization facility, such as counting semaphores.)
3. Give an implementation for the reader/writer lock using the synchronization facilities you defined for question #2. Your implementation should function correctly on multiprocessors. It is acceptable for your code to have usage limitations (e.g., it may be vulnerable to starvation under some conditions), but please explain any such limitations.
4. Briefly discuss the performance of your implementation of reader/writer locks. Could the implementation be made more efficient?

5. Mr. Hacker’s boss believes that the reader/writer lock can be implemented more efficiently using only an atomic test-and-set instruction for synchronization. Is she correct? Explain your answer.
Part 2: System Structure and the Hardware/Software Interface

Some modern programming languages provide an exception handling facility that allows a programmer to supply a block of code (an exception handler) to be invoked if some exceptional condition (e.g., divide-by-zero or a reference to an illegal address) occurs while the program is running. Exception handling implementations for compiled languages such as C and Modula generally rely on the hardware to detect exceptional conditions and notify operating system software, which initiates exception handling in the runtime system.

6. Outline the role of the hardware, operating system kernel, and runtime system in handling exceptions. How does knowledge of the exception propagate to each level of the system?
Operating systems that support virtual memory rely on memory management facilities provided by the hardware. One facility common on RISC processors is a *software-controlled* translation buffer (TLB). The TLB is a small hardware cache of virtual-to-physical address translations. The hardware directly handles references to addresses whose translations are present in the TLB; any other memory reference requires intervention by system software to load the needed translation into the TLB.

Processors with software-controlled TLBs provide special instructions to control the TLB. These instructions might include:

- **tlb_load_entry**: load an entry (i.e., a virtual page address and its corresponding physical page address) into the TLB.
- **write_enable** and **write_disable**: enable/disable write privilege for a virtual page. Any attempt to write to a page with a write-disabled translation will generate a trap to system software.
- **tlb_invalidate_entry**: remove a translation from the TLB.

7. Explain how and when the TLB control instructions might be used by the operating system software to implement a virtual memory operating system such as Unix.

a) **tlb_load_entry**
b) `write_enable` and `write_disable`

c) `tlb_invalidate_entry`

8. The TLB control instructions are generally not available to user programs. Why not? How are user programs prevented from using these instructions?
Part 3: Distributed Systems

9. Most distributed file systems cache recently used file data in client memory. What are the performance benefits of file caching? What are the performance costs?

10. File caching introduces the problem of cache consistency when files are shared across the network. Explain the problem and demonstrate it with an example.
11. Outline a plausible scheme for dealing with the problem of cache consistency in a distributed file system. You may ignore the problem of failures in your answer (but see question #12). Your scheme need not be identical to that used in any specific distributed file system.
12. What are the limitations of the file caching scheme you described for question #11? Explain how your scheme can handle failures (crashes), assuming that systems fail only by stopping, discarding the contents of their memory, and restarting.