Coby Schumitzky
Aaron Floreani
Ryan Toan Nguyen
Shaun Seah

Operating Systems Homework #2

1. The mutex-locking code in Figure 4.10 on page 111 contains two contiguous operations that remove the current thread from the runnable threads list and then release the associated spinlock. This ordering may seem suboptimal for performance because holding the spinlock for longer than necessary is not ideal. As such, a possible optimization might involve reversing these two operations. However, this optimization is not without potential issues. It is similar to the concept of two threads A and B, competing to acquire the same mutex. Thread A acquires the spinlock and discovers that the mutex is already locked. Consequently, Thread A adds itself to the waiters list and releases the spinlock. Prior to Thread A being preempted, Thread B acquires the spinlock, determines that the mutex is still locked, and also adds itself to the waiters list. Thread A gets preempted, and Thread B resumes execution. Now, consider the consequences of having reversed the two aforementioned operations. In this scenario, Thread B removes itself from the runnable threads list before checking if any waiters are present in the queue. Consequently, before Thread B yields to another runnable thread, Thread C becomes runnable and is selected for execution. Thread C attempts to acquire the mutex but fails because the mutex is still locked. Thread C then removes itself from the runnable threads list without yielding to another thread. Now, Thread A resumes execution. It acquires the spinlock, sets the mutex state to 0, and releases the spinlock. However, there are no waiters in the queue to notify because Thread B and Thread C both removed themselves from the list before yielding. Consequently, other threads trying to acquire the mutex will be unable to do so, leading to potential deadlock or undefined behavior. In conclusion, maintaining the original ordering of operations in Figure 4.10 is essential for proper functioning of the mutex-locking mechanism. While holding the spinlock for an extended period may seem suboptimal, it is necessary for correct and reliable operation.

2. The code in Figure 4.27 is a thread-safe method for selling tickets, where multiple threads may access the method concurrently. The code stores the current state of the ticket seller in a "State" object, which is accessed through an "AtomicReference" object named “state". The audit method is meant to let the ticket seller view the current state, such as the number of seats remaining and the cash on hand. However, if we replace the first three lines of the audit method with the two lines provided in the question, it could create a problem called a "race condition”. A race condition is when two different threads access the same piece of code at the same time, potentially leading to incorrect information being returned. In this case, two threads could access the "state" object at the same time, leading to the possibility of the audit thread getting inconsistent values for the number of seats remaining and cash on hand. For example, if one thread is in the middle of updating the state object while another thread is accessing it to perform an audit, the audit thread may get inconsistent values for the number of seats remaining and cash on hand. This is because the two separate "get()" calls in the new version of the audit method would be reading different snapshots of the state object. In contrast, the original audit method in Figure 4.27 obtains a consistent snapshot of the state object by using a single "get()" call at the beginning of the method, and then accessing the relevant fields of that snapshot. This ensures that the audit thread always sees a consistent view of the state object, regardless of any updates that may be happening concurrently. Thus, replacing the first three lines of the audit method with the two lines provided would be a bug that could lead to incorrect auditing information being reported due to race conditions.

3. See GitHub file “BoundedBufferTest.java” for implementation. Ensure JUnit is downloaded for execution of the program.

4. See GitHub file “BoundedBuffer2.java” for implementation.

5. For two-phase locking, it is not possible for T2 to see the old value of x but the new value of y. T1 will acquire a lock on both x and y before writing to them. T2 can read only the new values of x and y once the lock is released.

For read committed isolation level with short read locks, it is not possible either. Once T1 has written new values into both x and y, T2 will not be able to read these new values until T1 has committed its transaction. Then, T2 will only be able to read the new values of both x and y.

For snapshot isolation, it is possible for T2 to see the old value of x but the new value of y. Each write action stores the new value for an entity in a different location than the old value. Thus, a read action can read the most recent version or it can read an older version.

6. The virtual address of the first 4-byte word on page 6 is 24,576 and the physical address is 12,288. The virtual address of the last 4 byte word is 28,668, and the physical address is 16,380.

7. The IA-32 two-level page table has a page directory that can point to 1024 chunks of the page table, each of which can point to 1024 page frames. The IA-32 paged address mapping adheres to the following process. The virtual address is divided into a 20-bit page number and 12-bit offset within the page. The latter 12 bits are left unchanged by the translation process. The page number is subdivided into a 10-bit page directory index and a 10-bit page table index. Each index is multiplied by 4, the number of bytes in each entry, and then added to the base physical address of the corresponding data structure, producing a physical memory address from which the entry is loaded. The base address of the page directory comes from a register, whereas the base address of the page table comes from the page directory entry. This process is how page numbers 1047552 and 1047553 are calculated.

8. See GitHub file “vmArrayTimer.c” for implementation

1. We have a for loop that iterates through the array, adding 4096 to the index each time.
2. We then print the index and the value at that index. This will make the access to the array cache misses.
3. We then print out the time taken to complete the loop. This is the time to access the array.

In our code, changing the array size only sometimes has an effect on the average time. What we noticed is if the array size is small enough to fit entirely within the CPU cache, the average time does not change. However, if the array size is too large, then the program's performance may suffer (time wise).

This code was run on a MacBook Pro 2019 intel chip, compiled using and executed using gcc

9. See GitHub file “forkProgram.c” for implementation

This code was run on a MacBook Pro 2019 intel chip, compiled using and executed using gcc

After running the program, opening a second window, and running the ps command, we can see how the program works. The program creates two processes, a parent and a child process. Over all there are three processes we see: The shell process, aka the program, the parent process created by the program, and finally the child process created by the fork() system call which is replaced by the ps command eventually.