Name: Duc Anh Nguyen

Course: Operating System

Date: 5/1/2023

Learning Reflection 2

Reflection Prompt 1

Summarize the key learning takeaways you got about using **mutual exclusion** locks to synchronize concurrent code, including moderately complex programs involving multiple mutexes. (Give 2-4 substantial bullet points / very short paragraphs.)

- Mutual Exclusion locks are synchronization mechanisms used to prevent race conditions among threads. A race condition occurs when two or more threads access a shared resource at the same time, resulting in unpredictable behavior and incorrect input.
- To ensure this mechanism, threads use a locking mechanism such as a mutex, which provides exclusive access to the shared resource. When a thread wants to access a critical section, it must first acquire the mutex, execute the critical section, and then release the mutex so that other threads can access it.
- To use mutual exclusion locks on a complicated data structure that involves multiple mutexes, there are different ways to implement this, but here are some of the methods being discussed in class:
    - Approximate Counters: In situations where a counter needs to be updated frequently and doesn't require exact accuracy, approximate counters can be used to reduce contention for the same lock. With this method, multiple counters are

used as representatives of a single logical counter, and each counter is updated independently by different threads. The final value of the logical counter is the sum of all individual counters. This can significantly improve performance by reducing the frequency of locks.

- Semaphore: A semaphore is a counter that can be used to restrict the number of threads that can access a shared resource simultaneously. It allows a certain number of threads to acquire a lock at the same time, and all other threads are blocked until a lock becomes available. If a resource becomes available, it will increment the counter and decrement if a resource is being used by other threads.
- Concurrent Linked List: In a concurrent linked list, each node of the list is protected by its own mutex. When a thread wants to add, delete, or modify a node, it first acquires the lock for that node and performs the operation. This method allows multiple threads to access different parts of the list simultaneously, reducing contention for the same lock.

Reflection Prompt 2

Summarize what you learned about different kinds of **drives (HDD and SDD)**, particularly how their designs affect performance under different conditions. If you wish, you may also elaborate on how driving characteristics may affect how you design systems or code to have better performance. (Give 2-4 substantial bullet points / very short paragraphs.)

- Hard Disk Drives (HDDs) consist of a spinning platter with magnetic storage media and a disk head that senses or writes magnetic patterns on the disk. HDDs are best suited for tasks that involve large sequential reads and writes, such as storing media files or running backups. However, they are relatively slow for random access and I/O operations because the disk head must wait for the spinning platter to rotate to the correct position before it can access data. This waiting time is known as rotational latency and can slow down access times, particularly for random I/O operations.

- Solid State Drives (SSDs) rely on NAND-based flash technology to read and write data. SSDs can access data much faster than HDDs, particularly for random reads and writes. One major problem associated with SSDs is write amplification, which occurs when data is written to an SSD and the drive's controller must first erase any blocks that are no longer needed before writing new data. This process of erasing and rewriting data can cause additional wear on the drive's memory cells, reducing its lifespan.
- When designing a system or code for optimal performance, it's important to consider the type of tasks involved. For tasks that involve large sequential reads and writes and do not require fast I/O operation, such as media storage or backups, an HDD may be sufficient and more cost-effective than an SSD. However, for tasks that require fast I/O operations, such as manipulating data from sensors and calling interrupts, an SSD may be necessary for optimal performance. Additionally, it's important to consider the write endurance of an SSD and ensure that frequently written data is stored on a separate drive or partition to prolong the life of the SSD.

Reflection Prompt 3

Given your current OS knowledge, would you be able to implement **journaling** (write-ahead logging) in a file system, such as the Jumbo File System you implemented for P4? *If so*, what makes you sure that you could. *If not*, what would you need to research before you would be able to? Be as specific as you can.

(Assume that it's not a matter of programming skill; this question is about understanding the OS concepts in sufficient detail.)

- Given my current OS knowledge, I would be able to implement journaling in the Jumbo File System. There are four major steps of implementing a journaling file system:
    - Journal write: Write the contents of the transaction to the log.
    - Journal Commit: Write the transaction commit log to the log; wait for the write to complete.
    - Checkpoint: Write the contents of the update to their final locations within the file system.
    - Free: Free the transaction in the journal and update the journal superblock
- Currently, we have only implemented the checkpoint step of the Jumbo File System, but not the other three steps. To implement journaling, we need to create a separate superblock that contains the journal and update the journal metadata and i-node before proceeding to the checkpoint.
- The key concept of journaling is to prevent data loss in the event of system crashes. If we have to recover data by scanning the log and looking for committed transactions, it can be challenging to write out the blocks in the transaction to their final on-disk locations. This process requires practical skills and testing, but not necessarily additional knowledge.

Reflection Prompt 4

Given your current OS knowledge, would you be able to implement a **process scheduler** for a multicore / multiprocessor system, based on round robin, but that takes advantage of **cache affinity**? *If so*, what makes you sure that you could. *If not*, what would you need to research before you would be able to? Be as specific as you can.

(Assume that it's not a matter of programming skill; this question is about understanding the OS concepts in sufficient detail.)

- Based on my current knowledge of operating systems, I believe I could implement a scalable process scheduler using multiple queues that implement the round robin algorithm. When a job enters the system, it is placed on only one scheduling queue to avoid synchronization and information sharing issues.
- However, one potential challenge is migrating processes between schedulers and running them on different cores. To ensure optimal performance during migration, the process data should be available across all levels of cache in the cores before migration. This may require implementing a monitor to track data access across each core's cache levels. Additionally, the scheduler needs to monitor the cache occupancy of each CPU to prevent oversubscribing the cache. If a CPU's cache is already occupied by other processes, the scheduler should avoid scheduling a new process on that CPU to prevent cache thrashing.
- It's also important to consider the data access patterns of each process, which can be distributed across all levels of cache and cores in a multicore system. Further research may be needed to mitigate and monitor these access patterns.

Reflection Prompt 5

What general ideas were repeated in multiple topics of the course? (Not necessarily cutting across all topics, but that came up two or three different times.) What is significant about each of them that makes them come up multiple times in slightly different contexts?

- There are different ideas that have been repeated multiple times in this course are synchronization with mutual exclusion, scheduling policies, and virtual memory. These are the concepts that allow the operating system to manage resources and provide a stable and responsive computing environment efficiently and securely.
- Mutual exclusion locks are used to prevent concurrent access to shared resources by multiple processes, preventing race conditions and data inconsistencies. Without proper synchronization mechanisms, the system can become unstable and produce incorrect results.
- Scheduling policies determine how the system assigns CPU time to processes, which impacts overall performance and responsiveness. A poorly designed scheduling policy can lead to a system that is unresponsive and inefficient, while a well-designed policy can ensure that processes are executed in a fair and efficient manner.
- Virtual Memory, or even memory system in general, is a critical concept in operating systems the enable processes to share memory and allows the system to efficiently manage memory usage.
- With these concepts being so significant, it is understandable that they are being brought up many times with slightly different contexts or applications. If we are able to understand them in both high-level abstractions and low-level abstractions, we have an idea of how to manage resources and how to implement them when needed.

Reflection Prompt 6

What have I learned about my strengths and my areas in need of improvement? How will I respond to these in my work going forward from this course?

- From this course, I have learned all the fundamental concepts of operating systems and how to effectively implement them. However, the concepts that I feel I understand the most, such as mutual exclusion and virtual memory, are also the areas where I believe there is still more to learn. There are likely more advanced concepts and techniques that can be used to implement these fundamental concepts.
- Furthermore, one key area that I want to focus on in my personal assignments is implementing scheduling policies. Although we have had assignments and tests that cover these topics, I believe that practical practice would help me further enhance my knowledge. Additionally, I plan to do more research on the current scheduling policies being implemented in today's operating system market.

Reflection Prompt 7

Two-part question. Choose either one to answer (or both if you want).

- How has this course changed your perspectives?
- **What can you do with what you know now, that you could not do at the beginning of the coure?**

- There are a lot of concepts in this class that I find extremely helpful for expanding my knowledge, such as mutual exclusions, virtual memory, file systems, and I/O devices.

The programming assignments, especially P1, P3, and P4, have helped me to implement these concepts effectively. After completing this class, I feel that I have gained a lot of knowledge, but I also recognize the need to learn more about these systems. There are techniques and concepts out there that I have yet to fully grasp.

Reflection Prompt 8

What do you want to learn more about, past the end of this course, and why?

- As I move past the end of this course, I am keen on further exploring virtual memory and the different data structures used to implement mutual exclusion locks on modern systems. I believe that these are the most crucial concepts of an operating system. The programming assignments and lectures have provided me with a solid foundation, but I recognize that there is much more to learn. Additionally, I plan to review the material to ensure I have a better understanding of computer architecture in general.