Concurrent Programming Final Exam

Nguyen Xuan Binh 887799

Write a mini-essay (max. 400 words) describing semaphores.

- **What are semaphores?**
- **What variations there exists?**
- **How are they initialized and used?**

Try to give a compact and clear answer that addresses all the items.

What are semaphores?

A semaphore is a synchronization object that controls access by multiple processes or threads to shared resources in a concurrent setting. Normally, semaphore distributes access to critical sections of the program, where limited number of threads/processes are allowed to enter. Semaphore was first invented by Edsger Dijkstra in 1962

Variations of semaphores:

There are 3 variations of semaphores, which are binary, counting and mutex semaphores.

• Binary Semaphore: Binary semaphore is used when there is only one shared resource. Only one process is allowed to enter the critical region. When it finishes its job, it returns back the flag of the semaphore. Thus, binary semaphore has two methods: acquired() and released() for its binary operation

• Counting Semaphore: It is like binary semaphore but instead of allowing only 1 thread to the critical section, the counting semaphores have many keys and can distribute them to many processes. In this way, counting semaphore can handle more than one shared resource of the same type. Counting semaphore will be initialized with certain number of counts and when the count reaches 0, it blocks access to the resource.

• Mutex Semaphore: It is similar to binary semaphore, but the difference is that mutex semaphore prevents two same processes of race conditions which may cause non-deterministic result. For example, two enqueuing threads in a blocking queue will have access to the queue by the counting semaphore, but only one of them is allowed to enqueue one at a time by the mutex semaphore

How semaphores are initialized and used:

• Initializing process: Semaphores usually have two data fields, one is the number of keys and the second is a set of processes or threads waiting in a queue. When a semaphore is first initialized, its flags should be given a positive number (1 key for binary semaphore and >= 2 keys for counting semaphore) and its waiting queue is initialized empty.

- Usage: First the semaphore is tasked with "guarding" a critical section that multiple processes may want to read or write. When a process wants to use a resource, it requests the semaphore for a key and will be allocated one key if the semaphore still has available keys. If there are no keys left when

the resource is requested, the process will have to wait until other processes finish and return the key to the semaphore.

Write a mini-essay (max. 400 words) on concurrency in distributed systems.

- **What are distributed systems and how they are typically operated?**
- **What problems distributed systems exhibit that affect concurrent programming?**
- **How such problems can be resolved?**

What are distributed systems:
A distributed system is a system with various components that operate independently on different machines that communicate and coordinate actions with each other. This system appears as a single working system to the end-user. These machines have a shared state, operate concurrently and can fail independently without affecting the whole system's process.

How distributed systems are typically operated:
In a distributed system, the components use communication channels that send messages to each other instead of sharing a memory. Thus, there is no global state nor common memory, and the system state is updated by exchanging messages. Distributed systems are commonly implemented via the internet or cloud networks. A distributed system begins with a task and the distributed applications manage this task by splitting it into various subtasks. Then, each subtask is sent to their respective components. After the components finish their subtasks, they return the works back so the subtasks can be merged as a whole, finishing the original task.

What are some challenges of distributed systems?
Distributed systems have several challenges in their operations and design. Some of them are:
• Increased failure chances: If a system is not carefully designed and a single node/component crashes, the entire system can be at grave risk.
• Synchronization difficulties: Distributed systems operate without a global state. In other words, a distributed system requires that processes are appropriately synchronized to avoid errors and data corruption in transmissions
• Complex security problems: Since a distributed system works via a communication channel, it is susceptible to various security risks. An attacked single component will put the entire system at security risks
• Complex design: Distributed systems are noticeably more difficult to design and manage than a normal system of components that has a global state

How such problems can be resolved:
• Increased failure chances: a criteria of distributed systems requires the system to be fault tolerant. Each component designs should be ensured with concurrent correctness
• Synchronization difficulties: Constantly checking for the communication channels to be updated frequently and make sure they are not prone to data corruption
• Complex security problems: Make sure that each component of the distributed system is not easily prone to attack. Security of the entire system should be tested by professional technicians

• Complex design: Careful planning and documentation of the system is a must to keep track of the design.

Write a mini-essay (max. 400 words) describing memory models typically applied in concurrent programming.

- **What are memory models and they are useful in concurrent programming?**
- **What properties are important in memory models for concurrent programming?**
- **How memory models relate to program code and program execution?**

Try to give a compact and clear answer that addresses all the items.

What are memory models?

In concurrent programming, a memory model is a set of rules and protocols that dictate how the working threads interact with each other via the common shared memory and how they access and use the shared data. There are various memory models like JVM/Scala, x86-TSO, Power and ARM Memory Model

Why are memory models useful in concurrent programming?

Memory models are useful for concurrency because it is the framework that provides us knowledge on the understanding of the concurrent correctness of shared-memory. The memory model specifies how and when different threads can see values written to shared variables by other threads, and how to synchronize access to shared variables when necessary

Important properties of memory models in concurrent programming?

• Consistency model: there are several models, most common are strict consistency, sequential consistency, processor consistency and weak consistency. Knowing the concurrency model helps us understand the memory model better

• Data race definition of the memory model: in JMM, a data race occurs when a variable is read by more than one thread, and written by at least one thread, but the reads and

writes are not ordered by happens before

• happens-before relationship: a set of regulations on the relationship of two working threads, where one thread must definitely work/start before another

• Locking mechanism: in the TSO-x86 memory model, a lock prefix specifies as follows:

- Read operations are only allowed when the global lock is not held by another hardware thread

- Write operations appending to the store buffer FIFO are always allowed

How memory models relate to program code and program execution?

• In program code, the memory model defines the concurrency semantics of shared-memory systems. For reading operation, the memory model regulates which possible values that the read is allowed to return for any orderings of write operations performed by other processes within a concurrent program. Therefore, the memory model only allows a certain set of outputs of a program's reading and writing code.

• In program execution, the memory model generally permits executions whose outcome is unpredictable from the order of read and write operations in the concurrent program. Without a memory model, it’s impossible to know the result of a program execution, even if the program's implementations are available. Moreover, the memory model dictates which re-orderings are allowed and constrains the program execution to run only such re-orderings

Write an essay (max. 600 words) on correctness of concurrent programs. Focus on how the correctness can be defined and checked.

How the correctness of concurrency can be defined:

In concurrent programming, the program is defined to be correct if it satisfies two properties: safety and liveness.

• Safety Properties : during executions, nothing unexpected or out-of-thin-air behavior should be observed. A program is supposed to never enter a bad state. There are several factors that define what can make the program concurrently unsafe

- Race Conditions: they occur when two or more threads can access shared data and they try to change the shared data at the same time. Therefore, the result of the change in data is dependent on the thread scheduling algorithm or in other words, the threads are "racing" to access/change the data. This creates unexpected results and thread safety is not ensured.
- Critical Regions: because of non-determinism in the order of threads executing tasks on the shared variables, we do not know what the outcome is. These shared resources are called critical regions. Safety of the concurrent program is mainly concerned around how to distribute access to the critical region for many threads.

• Liveness properties: making sure that there is progress that must eventually take place during execution, so liveness is concerned with making progress in a program. There are three kinds of non-progress issues that affect the liveness in a concurrent program

- Deadlock: the threads are in deadlock when there is mutual exclusion, hold and wait, no preemption and circular wait between them. They made no progress and are stuck forever in busy waiting. Deadlock can be considered to impact both safety and liveness.
- Livelock occurs when the states of the processes involved in the livelock constantly change with regard to each other. So even if the threads are still working, they made no visible progress.
- Starvation: a thread is starved when it is prevented from obtaining its required resources and is made to wait, thus making it starve.

How the correctness of concurrency can be checked:

For programs that are about to terminate, we have to check that the two correctness criteria must always be true: it should produce results that are protected by the safety definition and made some progress by the liveness property

• For safety properties, we should ensure many mechanisms to carry out the concurrency so that there would be no unexpected executions of the code
- For out-of-thin-air problem: when a thread reads a variable without synchronization, it may see a stale value, but at least it sees a value that was actually placed there by some thread rather than some random value. This safety guarantee is called out-of-thin-air safety.
- For critical regions: we can check whether a region is unsafe or not when we allow many threads to access it if they produce non-deterministic results. If yes, we can apply synchronization or semaphores to protect the critical regions

• For liveness properties, there are several methods that we can check whether a system is progressing or not

- Weak Fairness: If a thread makes a request continuously, it should be granted access eventually.
- Strong Fairness: If a thread makes a request infinitely, it should be granted access eventually.
- Linear Waiting: If a thread makes a request, it will be granted access before any other thread is granted a request more than once.
- FIFO: If a thread makes a request, it will be granted access before any other thread making a later request.

Liveness properties are more difficult to be verified than safety properties because they are dependent on the scheduling in use. Moreover, liveness requires reasoning about infinite sequences of actions to prove that there is progress in the system.