# **Lecture 1 -Note**

1. What is the difference between sequential programming and concurrent programming?

Sequential programming- one task at a time

Concurrent programming- many tasks at a time

1. Concurrent vs Parallel programming?

No need for multi-core for concurrent. A single core is sufficient.

Need multiple cores for parallel programming.

1. What are the issues that concurrent programming leads to?

Livelocks, deadlocks, race conditions, starvation

1. Bank account which is shared by wife and husband. The wife is a career-minded wife, and the husband is a house-based husband for whom he gets a daily allowance.

20,000 (Deposit)



Balance= 20,000



20,000 (Withdraw)

CMW is trying to deposit 20,000. At the same time, HBH is trying to withdraw 20,000.

CMW is one thread and HBH is another thread and balance is the shared resource.

CMW reads the balance= 20,000 HBH reads the balance=20,000

Deposit= 20,000 Withdraw= 20,000

Total= 40,0000 Total=0

Shared resources 🡪 Bank account object 🡪 attribute called balance is the shared resource 🡪 that is the critical section as well.

Race condition is happening. Balance can be lost.

Solution🡪 Lock

20,000 (Deposit)



Balance= 20,000



20,000 (Withdraw)



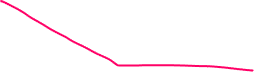
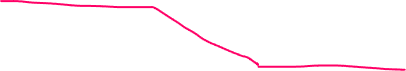
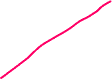
CMW acquired the lock HBH will be made to wait in the blocked state.



By using a locking mechanism, we avoid,

1. Racing condition overlapping is avoided🡪 Mutual exclusive is ensured (mutual exclusion refers to the property that ensures that only one process or thread can access a shared resource at any given time. Mutual exclusion is often implemented using synchronization techniques such as locks, semaphores, or monitors. These mechanisms allow threads to request exclusive access to a shared resource, ensuring that only one can access the resource at any time. Once a thread has finished using the shared resource, it releases the lock, allowing other threads to acquire it).
2. Data corruption.

Trains- Threads



Tracks- Resource

Critical section where overlapping is possible.

Should be mutually exclusive.

Accessing shared resources by multiple threads is known as a critical section. If multiple threads are allowed to access the critical section at the same time will lead to overlapping and, we call it racing conditions therefore the access to the critical section must be mutually exclusive.

How to ensure mutual exclusiveness between two threads? Use a locking mechanism.

Must use Synchronization. Where shared resource methods have been used. If one of the synchronized methods acquired the lock, only one can be called at that time until the lock is released. So, there won’t be any overlapping. Therefore, no data corruption.

Can the locking mechanism lead to any problem? Deadlock and livelocks🡪 which can also lead to starvation. Basically, the shared resource object is the monitor.

FSP🡺 State chart diagram

Correct behavior of a concurrent program by using categories are called desirable properties. These categories are known as.

1. Safety properties- Ensure mutual exclusion, avoid deadlock
2. Liveness properties- Avoid livelock (when one fork is accessible other fork is not accessible all the time), Ensure accessibility, Ensure fairness, avoid individual starvation

## **Lecture 2 - Note**

Concurrency Concepts

1. Process🡪 A program in execution is the process. It has its own memory. Application is called a process. Application can be made up of more than one process which cooperates with each other.

A process has a self-contained execution environment. Each process has its own memory space. However, what the user sees as a single application may in fact be a set of cooperating processes. To facilitate communication between processes, most operating systems support Inter Process Communication (IPC) resources, such as pipes & sockets.

1. Thread🡪 Within the process, each thread shares the resources and memory of the process.

Threads are sometimes called lightweight processes. Both processes & threads provide an execution environment. However, creating a new thread requires fewer resources than creating a new process. Threads exist within a process – every process has at least one. Threads share the process’s resources, including memory & open files. This can result in the efficient use of resources & communication between the threads. But it also leads to the introduction of the associated problems that arise with concurrency, e.g., interference, deadlock, etc.

Thread🡪 Way to achieve concurrent programming

1. Pseudo concurrency- single core- means only one single thread can be executed at any point in time- how come it is concurrent? - time slicing.
2. Real concurrency- using multi-core with time slicing or using multi-processor.

When you create an application by default it is a single threaded apartment. But you can add more threads into your application🡪 multi-threaded apartment.

In java there are two ways to create multi-threaded application,

1. Inheriting (extending) Thread class
2. Implementing Runnable Interface

What is the best option and why?

If we want the behavior, we select the interface.

We don’t need any specialization. So, we don’t need inheritance.

Because we need only the method run -OOP view

Thread priority is between 1 to 10

1. Low priority

5- Mid priority 🡪 default priority

10- High priority

The priority queue will be in the ready state as below

Tail

1

2

3

4

5- T3

6

7

8

9

10- T1, T2

Head

# **Lecture 3- Note**

**The life cycle of Java Threads**

What are the various states at which the java thread exists and how does the transition occur (How does it go from one state to another state and from a state what possible action can be performed on the thread)

A simple producer-consumer application.

1. There is a shared variable that is accessed by both the producer and consumer in a coordinated manner.
2. Coordinated manner🡪 Producer will produce something and put it into the shared variable 🡪 Once the production is over, the producer will wait until the consumer consumes.
3. Once the consumer consumes, the consumer thread will notify the producer to come and produce more and meanwhile, the consumer will be gone to the waiting state because there is nothing to consume further.
4. The producer will come out from the wait state to the runnable state and eventually, if the producer is able to hold the processor, time will produce again and notify the consumer again to the runnable state to consume again.

Producer- setData()

Consumer- getData()

If there is no communication between the producer and consumer it will raise the following problems.

1. Data loss
2. Double consumption

**wait ()** method will only call inside a synchronized block. Otherwise, IllegalMonitorStateException is called.

When the wait () method is called, the thread enters a WAITING state. To come out from the WAITING state another thread needs to call notify() or notifyAll() method to bring back to the RUNNABLE state. Threads will be in a WAITING state on the monitor object.

In our example, there will be a queue on the SimpleMailBox object.

wait(millisecond/nanosecond) 🡪 waits until time expires or notify()/notifyAll() is called. Whatever happens first.

**notify(**)🡪 one thread will get from the queue

**notifyAll()🡪** Informs all the threads to come out from the waiting state.

**Join (), join(ms)**

If a thread goes to Time waiting state because of wait() method called, there are two possibilities. It will wait time to expire or notify(), notifyAll() methods call.

When a thread calls the join() on another thread, the thread which called join() method will resigns the resources and goes into WAITING state until the thread on which the join () method called completes the execution.

Main thread and there is another thread that is created by the main thread called t1.

If the main thread calls the join () method in the t1, (t1.join() ) 🡪 main thread goes to WAITING state until the t1 complete the execution.

Join(ms)🡪 either waits for the time to expire or waits for the thread on which join (ms) is called to terminate (run method to complete the execution) whichever the earliest.

Depreciated methods:

stop()🡪 which used to stop a method🡪 problem with this method is invoked a ThreadDeath object is thrown and the thread will stop at the point of receiving the ThreadDeath object and it is getting killed (asynchronously) 🡪 danger part is while the ThreadDead object is being received the thread might be in the critical section.

So the object on which thread was working might be left in the inconsistent state 🡪 otherwise the monitor object might be damaged.

suspend(), resume(), destroy()

In case of HBH, CMW and bank account example > BankAccount object might be left in damaged state if the CW or HBH thread is killed while it is in the critical section => Code inside deposit() method and/or withdraw() method is the critical section.

Thread will terminate naturally when the run method completes the execution but what if the run() method has an infinite loop (Timer Applet example)

Applet has a life cycle method;

Init()🡪 start() 🡪 stop() 🡪 destroy()

Stop()🡪 thread to NULL.

# **Lecture 4- Note**

**Thread Scheduling🡪Thread priority and Thread Groups**

**Thread Scheduling**

Concurrent can be categorized into 2 categories based on the number of cores that machines have.

1. Real concurrent programming - where there are **multi-core processors** where each thread will get executed truly simultaneously in different cores or multiple processors each thread could run in different processes. In that case, also it is truly concurrent.
2. Pseudo concurrency- there are multiple threads that want to get executed but there is only **one single core**. At the given time only one thread gets executed. Then how does it come to concurrent programming? This is where time slicing (quantum) comes into play. Each of the threads gets a time slice (quantum) executed as a result it looks like they are executing concurrently though in reality, only one thread runs at any given time.

Why do we need thread scheduling?

Because there are multiple threads, and someone must pick one of them and give them the processor (time slice) to execute.

JVM uses fixed-priority pre-emptive thread scheduling 🡪 The thread with highest priority in the ready state will be allocated quantum of the processor to execute when the processor falls vacant 🡪 The highest priority of the thread will be moved from READY to RUNNABLE state.

What if the low priority thread is now in the RUNNING state and higher priority thread enters to the READY state. What do you think will happen? The high priority thread will block the lower priority thread. 🡪Now the lower priority thread will be pre-empted. 🡪 Kicked out from the processor. 🡪 Goes from RUNNING to READY while the higher priority thread will take up the processor. 🡪 goes from READY to RUNNING.

Based on the above fact we call the scheduler uses a scheduling algorithm of Fixed priority pre-emptive scheduling.

What you think will be the consequences of higher priority thread pre-empts the lower priority thread and getting executed? Leads to **starvation** 🡪 how been avoided the starvation? The thread scheduler uses a mechanism called **aging**🡪 Based on the time for which the thread is put into WAITING period its priority is elevated by the scheduler. So eventually after waiting for some time the thread with the lower priority will get a chance to execute. Once the execution is over, the original priority will be taken.

What if two threads with **same priority** enters to the READY state who will get chance to get executed.

They will follow the round robin basis and execute one after the other taking the time slice.

Thread priority 🡪 Priority is an attribute or instance variable of a thread object. Each thread has a priority, and its values range from 1 to 10.

Thread.MIN\_PRIORITY🡪 1

Thread.NORM\_PRIORITY 🡪 5 (This is the default value of the priority attribute of the thread object. Why? Because the main thread has the priority 5)

Thread.MAX\_PRIORITY 🡪 10

Priority attribute can be accessed and mutated by getters and setters:

getPriority(); 🡪 int 🡪return the priority value between 1-10

setPriority(int priority); 🡪 void🡪 sets the priority attribute to the parameter. 🡪 once a thread is created its priority can be changed at any time during the setPriority() method.

Any thread that is created inherits the value for the priority attribute from the thread creates it.

That means Thread (child thread-created) inherits its priority from the thread it is created (parent thread-creator).

What is thread scheduling?

Giving a thread quantum of the processor (a time slice) to get executed (to decide the order in which threads should run).

# **Lecture 5- Note**

**Thread Groups**

1. You can create thread groups and place threads and thread groups into it.
2. Thread group, if the parent thread group is not specified at the time creating it belong to the thread group of the thread creating it.

public ThreadGroup (String name)

1. Thread if created using one of the following constructors when the thread is placed in the thread group to which the creating thread belongs to:

Thread ()

Thread (Runnable target)

Thread (Runnable target, String name)

Thread (String name)

1. However, when you are creating a thread, you can specify the parent thread group using the following constructor.

public ThreadGroup (ThreadGroup parent, String name)

1. When you are creating a thread, you can explicitly specify the thread group.



|  |  |  |  |
| --- | --- | --- | --- |
| **threadArray** | **Index** | **threadGroupArray** | **Index** |
| main thread | 0 | main thread group | 0 |
| Thread 1 | 1 | ThreadGroupA | 1 |
| Thread 2 | 2 | ThradGroupB | 2 |
| Thread 3 | 3 | ThreadGroupB1 | 3 |
| Thread 5 | 4 | ThreadGroupB2 | 4 |
| Thread 6 | 5 | ThreadGroupC | 5 |
| Thread 7 | 6 |  |  |
| Thread 4 | 7 |

A screenshot of a computer

Description automatically generated

# **Lecture 6- Note**

**Semaphores**

There are two types of semaphores.

1. Binary Semaphore🡪 Mutex
2. General Semaphore🡪 Counting semaphore

**Binary semaphore**🡪 locking and unlocking critical section.

1. It will be having the value 1 or 0 🡪 1 means available to lock, so you can claim the lock. Once you claim the lock it becomes 0. So that competing threads cannot claim if any of the competing threads try to claim it will be locked. 🡪 It has only one permit.
2. Once a thread releases, its permit will be restored to 1 as a result competing thread will be unblocked and eventually claim the lock.

Permit=0🡪 If the claim () is successful.

permit=1🡪 Therefore the thread will be blocked.

When the thread goes out of a critical section, it will call the release () as a result permit=0.

All the blocked threads will be kept in the FIFO queue.

**General Semaphore**🡪 Keep track of availability. For ex: In the case of a producer-consumer problem, if there are multiple slots you can use a general semaphore for counting the free slots and the number of available items.

Two semaphores:

1. Free slots
2. Occupied

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |

Initially, there are 6 free slots.

Semaphore freeSemaphore = new Semaphore (6);

Number of occupied slots.

Semaphore occupiedSemaphore = new Semaphore (0);

Consumer 🡪 to consume it must successfully claim () in occupiedSemaphore semaphore.

Initially, occupiedSemaphore has 0 permit, therefore if consumer tries to claim (), it will be blocked.

freeSemaphore has 6 permits, therefore if producer tries to claim (), it will be allowed to claim (). If the claim is successful, then permit will be reduced by 1. It will be allowed to produce once the production is over, producer will call release () occupiedSemaphore which will increase the permit 0 to 1.

After the production🡪

freeSemaphore = 5 permit

occupiedSemaphore = 1 permit

mutex.claim()

Access the critical section



mutex.claim()

Such mistakes should be avoided.

freeslot.claim();

produce



itemsolt.release();

If you make mistake here such as:

freeslot.claim();

produce

freeslot.release();



This will lead to data loss.

In java there is only one Semaphore class. **Difference between fair setting and non-fair setting Semaphore** is basically, fair semaphore uses a FIFO queue. Where non fair setting semaphore let the waiting threads to jump queue.

# **Lecture 7- Note**

**Java Thread Synchronization and Monitor**

1. **Single slot monitor🡪 Monitor can hold one value at a time. Using implicit lock🡪synchronized keyword, wait(), and notifyAll() method. 🡪SimpleMailBox**
2. **Multi slot monitor🡪 MultislotMaliBox**
3. **Queue**
4. **We are going to look at 2 types of lock**
5. **Implicit lock- synchronized keyword and wait and notify method🡪 MultislotMaliBoxImpl**
6. **Explicit lock- Lock interface and ReentrantLock class that are available in the java.util.concurrency.lock interface🡪 MultislotMaliBoxExpl**

work for any given data type-generics.

Monitors are used by threads. Monitor is a passive object. It won’t do anything by own.

**wait() method can be only called in synchronized method.**

Queue in java has two implementations.

1. **LinkedList🡪 First in first out (FIFO)**
2. PriorityQueue🡪 Priority In Priority Out

Synchronized🡪 Avoid double consumption and production.

# **Lecture 8- Note**

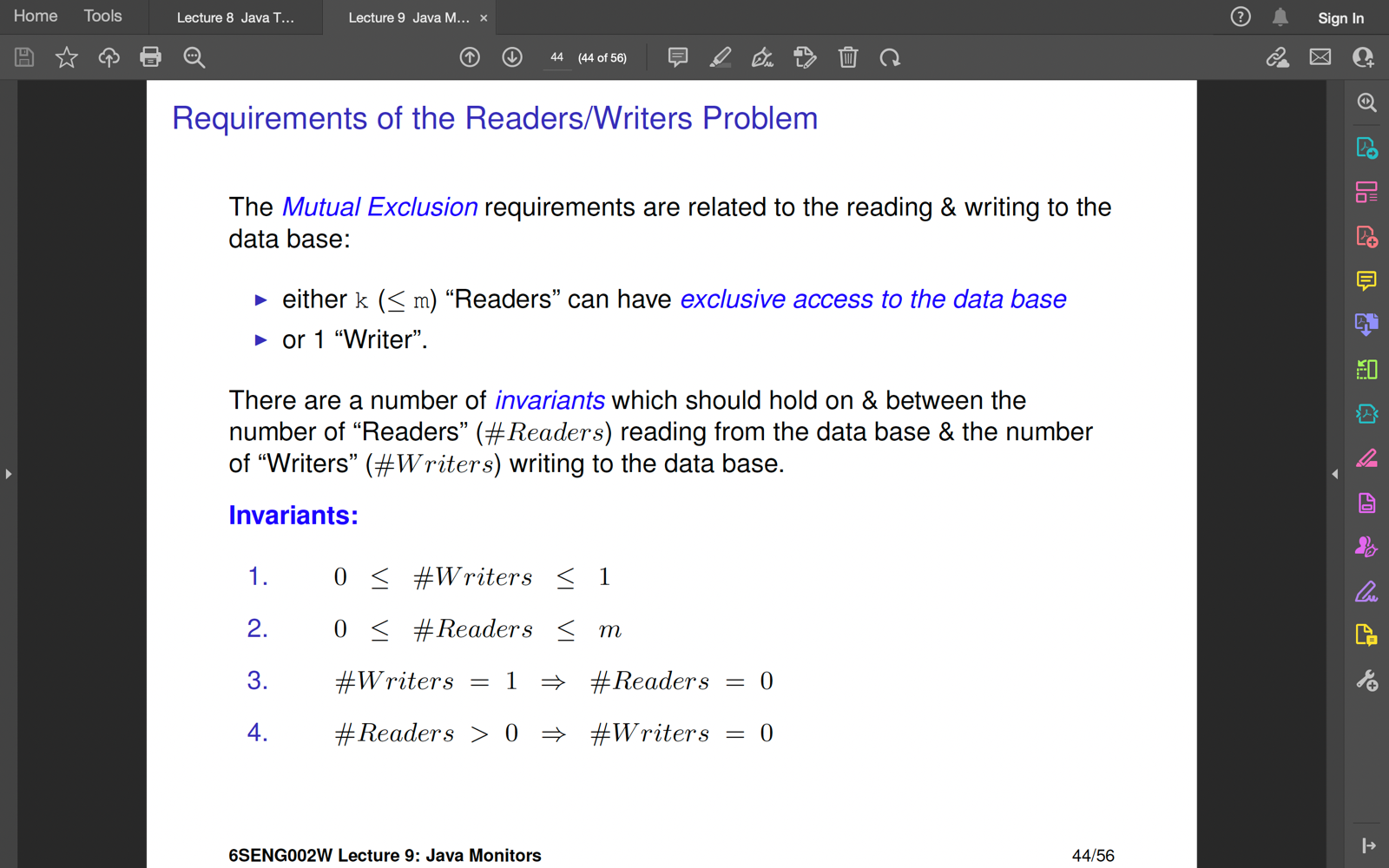
Monitor concept and Reader Writer problem

Reader Writer problem

Read operation Writer operation



Resource



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  | With synchronized | With no synchronized for readData () |
| Read Operation | Read Operation | Possible | Not Possible | Possible |
| Read Operation | Write Operation | Not Possible |  | Not Possible |
| Write Operation | Read Operation | Not Possible |  | Not Possible |
| Write Operation | Write Operation | Not Possible |  | Possible |

n no of other read operation is allowed. **Multiple simultaneous read operations are allowed.** Because there will be no change in read method.

Can we do this using synchronized? No not possible.

We cannot use synchronized block or method because the movement we enter a synchronized block cannot do another read or write why? A lock is established.