# Question

1. Process

* A program in execution is a process. It has its own memory.
* A process has a self-contained execution environment and it generally has a complete, private set of basic run-time resources i.e. each process has its own memory space. It could be also defined similar to that of programmes or applications.

1. synchronous action

* synchronous actions in concurrency involve a sequential and blocking execution of tasks, where each task must complete before the next one starts.
* The term "synchronous" refers to actions or operations that occur in a sequential and blocking manner. In other words, in synchronous execution, one operation must complete before the next one begins, and the program waits for each operation to finish before moving on to the next one.

1. Interference

* Interference is the issues and the problems that are raised when the shared resources are accessed by multiple concurrent threads simultaneously. This leads to issues such as:
  + Data Inconsistency – When the shared resource is accessed concurrently and multiple threads can lead to modify and read data at the same time leading to data loss and double consumption issues.
  + Deadlocks – A situation in which specific threads are unable to proceed as they are in waiting to other to release locks. It can occur when threads hold locks on recourses and wait for other recourses to become available. (circular dependency). A deadlock is a state in which each member of a group of actions, is waiting for some other member to release a lock.
  + Priority inversion – Instances where a low priority thread holds a resource that a high priority thread needs, this delays the output from the high priority thread

1. mutual exclusion protocol

* This refer to a set of mechanisms that are design to ensure that processes are executed in a way that they follow one at a time to access the shared resource. Mutual exclusion is executed to prevent the interference and data inconsistency in concurrent systems to access the critical code section or shared resource in exclusive manner.
* This is achieved through the use of Synchronization or though the use of locks. That is when the thread enters a critical section of the code a lock is acquired so that the shared resource could be only accessed through once process a time.
* Locks types:
  + Implicit lock – The lock is acquired with the synchronized method or synchronized block, and the lock is release when the thread exists the synchronized method or block.
  + Explicit lock – A lock object is created using the Lock interface and ReentrantLock class. This allows more flexibility as to acquire lock with the timeout. And the conditions could be specified to complex the synchronization patterns.

private ReentrantLock reentrantLock = new ReentrantLock();

private Condition notFull = reentrantLock.newCondition();

1. Livelock – When 2 or more processes or threads change their state continuously in response to the changes in the other processes without any progressing. It is a special case of resource starvation; as in the livelock situation the states of the processor’s changes but the process is not progressing.
2. Starvation - Starvation happens when “greedy” threads make shared resources unavailable for long periods. For instance, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.
3. Race conditions – race condition occurs when the shared recourses are accessed by multiple operations which result in data loss or inconsistence results of the shared resource. That is simplified as the behaviour of a program depends on the relative timing or interleaving of multiple threads or processors.
4. Synchronization – the mechanism where coordination of multiple threads or processors to ensure that the shared resource is accessed in an orderly manner, it aims to mitigate the race conditions, data losses, and other interferences.
5. Interleaving – In concurrent programming, multiple threads are their instructions are interleaved with each other that is, the instructions/ operations from different threads are altered in the execution stage which makes the concurrent program output non-deterministic.
6. What you think will be the consequences of higher priority thread pre-emps the lower priority thread and getting executed? **Starvation** => how the starvation is being avoided” The thread scheduler uses a mechanism called aging -> based on the time period for which the thread is put into **WAITING** period its priority is elevated by the scheduler $o eventually after wasting for some time the thread with lower priority will 1st the chance to execute

# Question

1. PROC1 = (act1 -> act2 -> PROC1).
2. PROC2 = ( act1 -> ( act2 -> PROC2 | act3 -> PROC2)).
3. ||TWOP1s = (PROC1 || PROC1).

a). PROC\_1 = (tick -> tock -> ERROR).

b). PROC\_ A= (d -> e -> STOP).

PROC\_B = (a->( c -> STOP | b -> END).

PROCES\_2 = (a -> (c -> STOP | b -> END) | d -> e -> STOP).

# Question

1. Thread is a light weigh process as:

* Threads runs within the context of the process, and it takes advantages of resources and memory of the process.
* They will be accessing the shared recourses and works in the shared memory but has its own resources within a running program.
* Inter communication between the threads are easier since they are in the same memory location

1. The difference between the start() and the run() method in java depends on how the threads are executed.
   1. Code 1

* The start() method is called for both myThread1 and myThread2, this method starts new threads. That is a new thread is created for each thread object.
* Then the run() method will be invoked on each of the new threads that is created and this method will run concurrently on the 2 threads.
* Output – the separate output that is obtained by the 2 threads that is executed concurrently which result in different orders of execution in each run.
  1. Code 2
* The run() method is directly called on two threads myThread1 and myThread2, this results in invoking the method on the currently thread specified.
* The method will execute on the two methods according to their order in the code therefore the two threads will run sequentially in the same thread of execution.
* Output – the run method doesn’t create a new thread therefore the method is called on the same thread of execution and is executed sequentially printing the names in order.

1. Different ways to create a Java thread:

There is main 2 ways in which the java thread could be created:

1. by extending the Thread class and overriding the run() method to define the actions of the thread.

Public class Printer extends Thread {

//write some code

@Override

Public void run() {

//body of the thread

}

}

1. by creating a object of the Runnable interface as its target i.e. using an class that implement the Runnable interface to implement the run() method

Public class Machine implements Runnable{

Public void run() {

//body of the thread

//code executed when the thread is started

}

}

1. When the Thread class is extended, the constructor of the superclass should be invoked explicitly the *super()* keyword. Using this keyword, it is ensured that the initialization of the class the Thread is defined is done. Which helps with the coordination of the Threads.

Public class Document extends Thread {

Public Document() {

Super();

//additional code

}

}

1. The body of the thread is defined by the run() method. How the run() method is defined it define what is happening in the thread. The run() method acts as a entry point for the thread execution. When the start method is called on the object of the thread class, it invokes the run() method so that the tasks and the logic of the thread is performed in order.
2. The thread is initiated using the start() method called on the object of the thread class. The start() method is invoked the thread’s run() method is executed in separate thread.

*Public class Main {*

*public static void main(String[] args) {*

*//initiating a new object of the Thread class*

*Printer printer1 = new Printer();*

*Thread printerThread = new Thread (printert1);*

*printerThread.start();*

*//initiate the new thread that is made using implementing the runnable interface*

*Thread machineThread = new Thread(new Machine());*

*machineThread.start()*

*}*

*}*

# Question

**Semaphores**

Features of semaphore:

* Semaphore maintains an internal count that represent the number of permits available.
* Uses 2 main operations as acquire (wait) and release (signal) which modify the internal counter of the semaphore.
* When the acquire operation is executed the permit count of the semaphore is decremented so that when it reaches a negative value the lock on the thread is released.
* Semaphore is initialized with an initial count which determine the permits.
* If the permit = 0 the thread will be in BLOCKED state until the permit becomes 1 with the other semaphore releasing the lock.
* These uses a mechanism to prevent deadlock situations by carefully managing the scheduling of threads.

1. Terms mutex, binary semaphore and general semaphore and generalised semaphore refers to different versions of Semaphores.
   1. Mutex (Mutual Exclusion)

* A special Semaphore designed for the mutual exclusion; it allows only one thread to access the critical section of the code/ the shared resource at a time.
* These are typically binary semaphore in states 1/0, where state 0 provide that the critical section is available to be used (unlocked), and state 1 suggest that it is in the locked state/in use
* Simple way to prevent data loss and double consumptions.
  1. Binary Semaphore
* Binary semaphore consists of 2 values either 1 or 0; where 1 mean 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 thread try to claim it will be blocked => it has only 1 permit
* Once a thread release the it permit will be restored to 1 as result competing thread will be unblocked and eventually claim the lock
* It operates similar to mutex semaphores but it is not limited to mutual exclusive cases
  1. General Semaphore
* Have a counter that can have values greater than 1, this counter represents the number of permits or resources available.
* Unlike binary semaphores, general semaphores states are not limited to 2. It can also represent count of available resources, and allow multiple threads to access the critical conditions of the codes simultaneously up to the available permits.
* In cases where multiple slots are considered the general semaphore can be used to count the freeslots and number of available items.

1. Concurrent programming problem for each of the semaphore types
   1. Mutex

* Case where multiple threads need to update a method in the shared class. That in in the Reader-Writer problem there could be many readers and many writers also. But when a writer is accessing the write method no other writer could access that method.
* Mutex is used to ensure the mutual exclusion for access to the variables No\_of\_Readers, which is s hared between Readers.
  1. Binary semaphore
* Case where producer-consumer scenario is with multiple producer threads are generating items and multiple consumer threads are consuming them. A binary semaphore is used to signal whether items are available for the consumption. Binary nature of semaphore is as available is 1 not available is 0.
  1. General semaphore
* In a case where limited databases are present that are accessed by multiple threads. A general semaphore could be used to control the number of threads accessing the resources simultaneously, ensuring maximum connections are not exceeded.
* keep track of availability for e.g. in case of producer consumer problem if their multiple slots you can use general semaphore for counting the free slots and number of available items

Two semaphores:

* 1. For freeslots
  2. For number of items available (occupied slots)

1. Main features of monitor

* Monitor consists of a collection of declarations of permanent variables, which are used to indicate the state of the resource which are a collection of procedures & functions declarations.
* The monitor could be implemented as following:
  1. Single slot monitor using a implicit lock => this uses synchronized keyword with the methods such as wait() (to make the thread enter a WAITING state) and the thread to come out of the waiting state the notify()/notifyAll() method should be called by another thread.
  2. Multi-slot monitor using implicit lock => multi-slot monitor uses a implicit lock to coordinate the access between the multiple threads and the associated condition variables of it. Implicit lock provides mutual exclusion for the threads so that one thread is executed. This uses the wait(t) operation to keep the thread in the time waiting state while the lock is said to be released from the other thread using the signal operation.
  3. Multi-slot monitor using explicit lock => Explicit approach of the acquiring a lock to the threads involves using a external lock provided by the ‘java.util.concurrent.locks’ package. This use the ReentrantLock for the mutual exclusion purposes and it could create many condition objects to handle the WAITING and the signalling of threads. Also, to ensure the proper synchronization the lock() and the unlock(0 method is used.

1. While loop is associated in the monitors to be utilized for the timed waiting on the conditions defined for the process. *While* loop is used as it could recheck the conditions after the thread is executed, and the loop is continued until the condition is satisfied, this mitigate the thread being executed without the conditions beinf satisfied. The thread exits the while loop when the condition is met, which allows the critical section of the code could be proceed with ease.

# Question

# Question

1. After a thread has been created it is in NEW thread state & it is an empty Thread object. No system resources has been allocated to it yet. When a thread is in NEW state it can be put into the runnable state using the start() method. Calling any other method beside start() raise IllegalThreadStateException.
2. Thread can execute code in RUNNABLE state

Threads can not execute code in : NEW, BLOCKED, WAITING, TIME\_WAITING, and TERMINATED states.

1. If a thread starts attempts to acquire the *synchronized* lock in the RUNNABLE state but if it fails to acquire the locked the state of the thread moved to the BLOCKED state which is a non-Runnable state, and the Thread is there until the lock it tried to acquire is released using a keyword like notifyAll(), notify().
2. Java virtual machine decide which Thread to execute using the priority that is given to specific threads. It uses Fixed priority pre-emptive thread scheduling where the Thread with the highest priority will be allocate quantum of the processor. Which is simplified as the thread with the highest priority will be moved from READY to RUNNING state in the RUNNING and if a thread is not assigned a priority (from 1 to 10) the threads are executed in the Round-robin basis.
3. In the TIME\_WAITING state the thread is in waiting for a specified time. The state could be in TIME\_WAITING due to one of the following methods:
   1. sleep(t) -> through this it waiting for a timeout after t time.
   2. wait(t) -> thread is waiting for a timeout after t time or till another thread calls notify()/notifyAll() method on the object.
   3. join(t) -> thread is waiting for a timeout t or for a specific thread to terminate.
   4. Thread can be in TIME\_WAITING is another thread interrupt the execution of the thread using the Thread.interrupt() method.

**Thread Life Cycle**

* NEW state : the state that the thread is is when it is created as:

Printer printerThread = new Printer();

In this state the Thread is an empty object, and no system resourses are allocated for it. Only the start() method could be called in this state otherwise it raises an *IllegalThreadStateException*

* RUNNABLE state : the thread is in runnable state when the thread object calls the start() method.

printerThread.start();

This creates the necessary system recourses for the thread to execute the run() method and schedule it to run. The state is called RUNNABLE rather than RUNNING state as though the thread is executed to run it could be waiting on accessing the processor with the schedule the thread have been given with.

* BLOCKED state : The current Thread is in the BLOCKED state when the thread failed to acquire the lock on the monitor (i.e. when another thread has accessed the monitor)

The BLOCKED state of the thread is continued till the release on the lock and the monitor could be accessed by the current thread to access the changes to the critical code sections.

* WAITING state : the waiting state of the THREAD is acquired when the current Thread is waiting for another thread to perform its actions.

The waiting state is acquired on the Thread using the following methods –

* + wait() – the thread will be on waiting state on the monitor object, i.e. the thread will be added to the waitset.
  + join() – this method will relinquish the resourses and goes into WAITING state until the thread on which the join() method is called complete the execution.

The when another Thread calls the notify() or notifyAll() method the thread will be brought back to the RUNNABLE state.

* TIME\_WAITING state -the state is acquired when the Thread is waiting on a specified time.

The Thread is put into the TIME waiting state by calling the one of the flowing methods –

* + sleep(t) – when the thread is sleeping for a *t* amount of time it is in the TIMED\_WAITING state and the thread is waiting for a timeout to go into the RUNNABLE state.
  + wait(t) - the wait method keep the thread in TIMED\_WAITING state till the timeout is obtained or till another method calls the notify()/notifyAll() method is called.
  + join(t) – the join method either keep the Thread waiting on the time to expire or waits the thread on which the join(ms) is called to terminate(run method to completely execute)
* TERMINATED state – thread dies when the run() method exit normally.

# Question

1. Reader-Writer problem is a generalisation of the mutual exclusion scenario. The issue raised by the case is that there is a fixed number of resources are in the database, and number of processors require the same piece of resource simultaneously.

Reader-Writer problem is designed so that many of the readers could access the shared resource at the same time, but when a Read operation is going on the there cannot be any other write operations executing simultaneously. Also, if the write operation that (0 <= no.of writers <= 1) taken in place simultaneous writing operations are not allowed also, when writing operation is executing there cannot be any other reading operations executing.

Mutual exclusion requirements are related to the read ad write operations, a sthe readers can be exclusive or 1 writer operation is executed.

1. In the Reader-Writer problem 2 binary semaphores are used:

Mutex is a binary semaphore that will used in this case study, so that the only one thread wither of reader/writer can access the critical section at a time preventing the concurrent access. That is if a Reader with permit = 1 starts accessing the critical section with claim() this blocks the Writer operation with the permits = 0. Then when the thread goes out of the critical section it will call release as a result the permit for the Writer operation becomes 1.

Counting semaphore is used by the Reader operation to keep track of the readers who are currently accessing the shared resource. It allows multiple readers to access the critical section concurrently while maininatining the exclusivity with the writer operations.

Then the mutex semaphore is used for the Writer operation to withhold the mutual exclusion among the writers, allowing one writer to access the critical section at a time.

a). the buffer mentioned in the given code section is a multi slot monitor that uses a implicit lock. The synchronized keyword is used to obtain the lock on the methods as take() and put(int i). Which is when the condition that is specified by the Boolean variable and the thread use the wait() method to wait on the conditions to satify so that the critical code section is executed in a safe manner. This Buffer is also implemented in a way the thread safety and the race conditions are mitigated.

b). The while loop on the put() method ensure the safety of the critical code section in which the thread is made to wait till the Boolean variable new-data is TRUE (i.e. data is available for the consumer to consume). One the new\_data variable becomes FALSe (i.e. no data available to consume) the thread exit the while loop to acquire the critical condition of the code in which the contents are updated to the value specified by the method call and the new\_data variable is made TRUE. Then the thread notify all the other Threads waiting on the monitor that the new\_data is available.

c).

# Question

1. Number of forks to be picked up for each philosopher is initiated with the integer array Fork[i] which is updated as 0, 1, and 2 for the philosopher i to pick up.
2. If the philosopher does not have sufficient forks to eat, i.e. < 2, then it is in put to waiting state with the wait( OK\_to\_eat[i] ), which release the fork monitor and waiting on the condition OK\_to\_eat[i]
3. Philosopher 1 and philosopher 3 will update the forks available for the philosopher 2
4. When a philosopher ‘j’ puts down the 2 forks, the number of forms in the list ‘j+1’ and ‘j-1’ the neighbouring philisophers. Then the method checks whether the adjacent philosophers have 2 forks with them selves if one does (assume j+1) then it signals OK\_to\_eat[j+1].
5. This declare an array of conditions variables, i.e. these are queues of processes that are waiting on some conditions. The intention of this Condition[] OK\_to\_eat ; is to keep the thread in WAITING so that the monitor method could be processed by one operation. In java this is called as the wait-set which the threads are added when the lock is acquired by one thread.
6. The \signal(condVar)" method is used to signal that the \condition" associated with the condition variable condVar is now true. The result is that just the process at the front of the condVar queue is signalled, i.e. woken-up & removed from the queue. The Java equivalent is Object.notify() which only \wakes up" one thread that is waiting in the monitor’s \wait-set". It is not Object.notifyAll(), as this wakes up all the threads in the monitor’s\wait-set".

**Dinning Philosophers**

The case is made providing that there are 5 philosophers and each of the philosophers need 2 chopsticks to dine. And the philosophers alternate between eating and waiting for the chance to eat. Also, to eat the philosopher mut get the 2 chopsticks adjacent to him/her.

Challenge faced by this case is when a philosopher is unable to obtain the necessary chopsticks and cannot make progress. i.e. a deadlock is occurred when a philosopher obtains one chopstick and wait for the other leading to circular waiting in dependency.

This issue could be mitigated using a monitor/butler which overseas the operations of the philosopher’s. The monitor will limit the number of philosophers that could pick chopsticks simultaneously. That is reasoned as the deadlock is led with all the philosophers trying to obtain the shared resource simultaneously.

A semaphore could be used to construct a deadlock free process for the dinning philosophers. Taking the mutex binary semaphore to ensure that only one philosopher could access the shared resource at a time. Also a general semaphore could be used to control the number of philosophers who are allowed to eat simultaneously.

Semaphore mutex = new Semaphore (1); //to acquire the mutual exclusion

Semaphore butler = new Semaphore (4); //general semaphore to count the philosophers