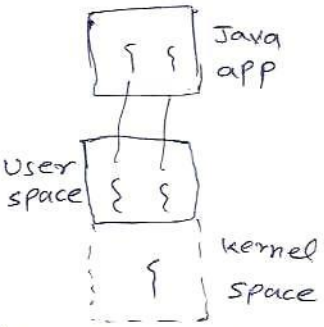
Multithreading Benefits

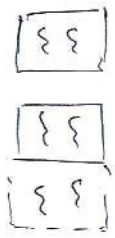
- Improved throughput. Many concurrent o/p & I/P requests within a single process.
- Simultaneous and symettric use of multiple processors.
- Applicatn responsiveness. entire app will not block.
- Improved server responsiveness.
- ~~Better~~ Program structure simplification.
- Minimized system-resource usage.
- Better communication.



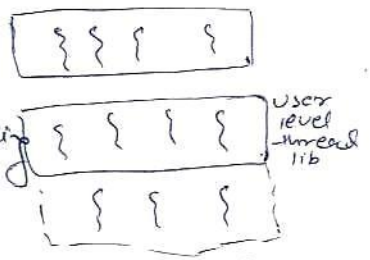
Multi-threading Models

① Many-to-One :- All thread activity is restricted to user space.
- only one thread at a time can access the kernel.
- Provides limited concurrency.

② One-to-One :
- All the thread can access the kernel at same time.
- Restriction is to be careful with thread as each thread adds more weight to the process.



③ Many-to-Many : (2-level Model)
- minimizes programming effort while reducing the cost and weight of each thread.
- The user level thread library provides scheduling of user level thread above kernel threads.
- kernel needs to maintain only the currently active thread.
- It provides a standard interface, simpler prog model, optimal performance for each process.



Ways of creating thread.

- Extending thread Class.
- Implementing Runnable Interface.
- Using executor framework along with Runnable & Callable ~~task~~

```
Public
Class MyThread implements Runnable
{
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            sop(i);
        }
    }
    p.s.v m.
    {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        MyThread t3 = new MyThread();

        Thread th1 = new Thread(t1);
        Thread th2 = new Thread(t2);
        Thread th3 = new Thread(t3);

        th1.start();
        th2.start();
        th3.start();
    }
}
```

getName , setName
getId , currentThread

```
public class MyThread extends Thread
{
    @Override
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            sop(i);
        }
    }
    P.S.V.m()
    {
        Thread t1 = new MyThread();
        Thread t2= new MyThread();
        Thread t3= new MyThread();
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Thread Scheduler

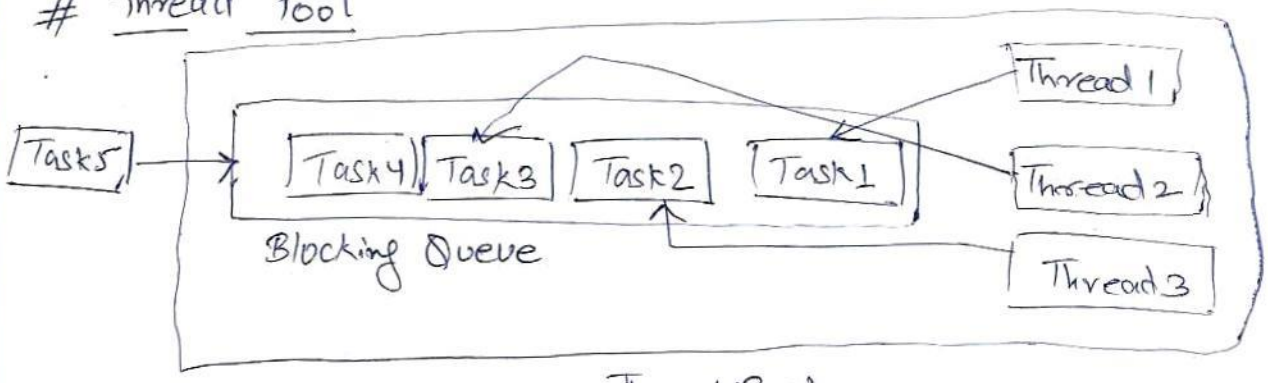
- Responsibility to execute the threads in Runnable State.
- Priority: Each thread has 1 to 10. If 2 or more threads have same priority then it select at random.
- Arrival time: If priority is same, the time is checked.

Algorithms

1. Preemptive - priority scheduling
2. First come - First serve
3. Time Slicing / Round - Robin.

* If we call start() for same thread multiple times, it will throw IllegalThreadStateException.

Thread Pool



Thread Pool

* Blocking queue supports flow control. A thread trying to enqueue an element in a full queue is blocked until some other thread makes space in a queue.
Also, it blocks if a thread is trying to delete from empty queue. until some other thread insert an item.

Create Thread using Lambda Exp.

```
p.s.v.m.{
    Runnable myThread = () ->
    {
        Thread.currentThread().setName("myThread");
    };  SOP(Thread.currentThread().getName());
    Thread run = new Thread(myThread);
    run.start();
}
```

Reentrant Lock

- It implement the lock interface and provides synchronized to methods while sharing the resources.
- It allows threads to enter into the lock on a resource more than once.
- When ~~block~~ thread enters in to the lock for first time, hold count is 1. Before unlocking, it can re-enter in the lock again and count is incremented by one.

- For every unlock, count is decreamented by 1.
- When hold count = 0, resource is unlocked.

Semaphore - It is a variable that is used to manage processes that runs in parallel. It is a non-negative variable that indicates the no. of resources in system that are available at a point of time.

By using counters, the semaphore controls the shared resources to ensure that threads running simultaneously are able to access the resources and avoid race cond.

2 types of Semaphore:

(1) Counting Semaphore: The semaphore variable is initialized with no. of available resources. When a process needs to acquire a resource, the value of semaphore is reduced by 1. (wait()) Once the process is done using the resource, value is increased by 1. (signal()) When value is 0, no resource is available and process has to wait for the resource to be released.

(2) Binary Semaphore: The value of semaphore can either be 0 or 1. The value is set to 1 in the beginning. When a process invokes a resource, value is changed to 0. When a resource is released, value is changed to 1. When value is 0 & process has to wait to acquire the shared resoure which has to be released by prev. process.

⇒ Timed Semaphore: It permits a thread to run for a specific amount of time. Once the time elapses, all permissions are released and the timer is reset.

Semaphore	Mutex.
① Allows the shared resource to be used by multiple process till it is available.	Allows multiple process to access a resource but not simultaneously.
② It is a non-negative variable	It is an object.
③ Deploys signal mechanism where the method wait() & signal() are used.	Deploys locking mechanism where process using a resource locks it & releases it after use.
④ Multiple process can change the value of semaphore but only 1 can modify at a given time to acquire a resource	The same process can acquire or ~~relase~~ release a lock at a time.

Design Patterns.

I Creational Design Patterns : provide various object creation mechanisms, which increases flexibility and resuse the existing code.

① Factory Method : provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

eg1

eg2 Shape
- Square
- Circle
- Rect

ShapeFactory

eg3. Hiring Agency and Employer

→ Applicability :

- when you don't know the exact types and dependencies of the objects your code should work with.
- when you want to provide users of your library with a way to extend its internal components.
- When we do not want to expose sub class (eg: Circle)

- when you want to save system resources by reusing existing objects instead of rebuilding them each time

Advantages
- Avoid tight coupling b/w creator and products.
- Single Responsibility Principle.
- Open/close Principle.

Disadv. - forced to send all the parameters and optional para. as NULL
- The code may become complicated.

(8) Builder Pattern: Let's us construct complex objects step by step. It allows us to produce diff. types and representations of objects using the same construction code.

eg: Building a House, Building XML file.

Applicability
- to construct complex objects.
- when you want your code to be able to create diff representations of some product.

Adv. - construct objects step-by-step.
- reuse the same cons. code
- Single Responsibility

Disadv - complexity increases

⇒ Diff b/w Factory and Builder is, builder is used when you need to do a lot of things to create an object. Factory is used when an object can be created within one method call.

#MultiMap : Collection of key-value pair, where each key is associated with multiple values.

③ Abstract Factory Pattern

- It deals with multiple families of products.
- It declares a set of methods that return diff. abstract products. These products are called family.
  A family of product have many variant, but product of one variant is incompatible with product of another variant.

Eg: Modern Furniture
- Chair
- Sofa
- Table

Old-fashioned
- Chair
- Sofa
- Table

- Code do not depend on the concrete classes of those products.

Adv. Single Responsibility Principle
Open / Close principle.
Avoid tight Coupling.
Products are compatible

#Diff b/w Factory Pattern & Abstract Factory Pattern

- Factory pattern is responsible for creating products that belong to one family. while abstract deals with multiple families of products.
- Factory uses interfaces and abstract class to decouple the client from the generator class and resulting products. Abstract Fac has a generator that is a container for several factory methods, along with interfaces decoupling the client from generator and products.

eg: of factory ⇒ A toy factory which produce only one type of toy → car.

eg. of Abstract ⇒ Pasta Maker, there can be any shape of pasta depending on the disk (factory) all of them inherit property of how to create pasta (abstract factory) but pasta maker do not have any knowledge.

* Abstract factory is consist of:

→ Abstract Product: declares an interface or abstract class for a type of product object.

→ Abstract factory: declares an interface or abstract class for operations that create abstract product objects.

→ Concrete Products: - implements Abstract Product defines a family specific product to be created.

→ Concrete factory: implements Abstract factory. - to create concreate product objects.

→ Client: It only interact with interfaces. Uses Abstract factory to get Abstract Products objects.

II. Structural Design Pattern

Explains how to assemble objects and classes into larger structures while keeping them flexible & efficient.

1. Adaptor DP: Allows ~~interfaces~~ objects with incompatible interfaces to collaborate.

eg1. Europe charging Plug and US charging plug.
Soln is Universal Adaptor

How it works.

- Adapter gets an interface, compatible with one of the existing objects.
- Using the interface, existing objects can call the adaptor's methods.
- Upon recieving a call, adaptor passes the reques to second object in the order and format it was expecting.

★ Its even possible to create 2-way adaptor

eg:

Applicability.

- when we want to use some existing class but its interface isn't compatible with code.
- when we want to reuse several existing subclass that lack some common func

III. Behavioural Pattern.

They are concerned with algorithms and assigned responsibility blw objects.

1. Strategy: It lets you define a family of algo., put them in a seprate class (encapsulate them) and make their object interchangeable.

eg1: Get to airport using bus, cab, cycle, etc.

eg2: map route for walking, cycling, car, train, etc.

Applicability.

- when you want to use diff. variants of algo. within an object and be able to switch the algo during runtime.

- you have lot of similar classes that only differ in the way of execution behaviour.

Advantages
- switch b/w algo. at runtime.
- can isolate the implementation details of an algo from the code that uses it.
- replace inheritance with composition.

Object - Level Lock

This mechanism is used when we want to synchronize non-static methods or code blocks s.t. only one thread will be able to execute the code block on given instance of class.

Eg:
```
class Demo {
        public synchronized void method() { }
        or
        public vod method() {
                synchronized (this) { }
        }
        or
        private final Object lock = new Object();
        public void method() {
                synchronized (lock) { }
        }
}
```

Class - Level lock.

Used to make static data thread-safe. It prevents multiple threads to enter syncronized block in any of all instances of the class on runtime.

Eg:
```
class Demo {
        public synchronized static void method() { }
}
        or
```

```
public class Demo{
    public void method()
    {
        synchronized (Demo.class){}
    }
    or
    private final static Object lock = new Object();
    public void method()
    {
        synchronized (lock){}
    }
}
```

Disadvantages of Static Methods.

- Static methods can't be used for abstraction and inheritance.
- Can't declare a static method in interface or static abstract method in an abstract class.
- Can't access non-static class level members.

Strategies to handle collision in HashMap?

→ Hashing is a process of transforming data and mapping it to a range of values which can be looked up.

1. Open Hashing (Seperate Chaining)

This implementation combines a linked list with a hash table in order to resolve collision. This method uses extra memory, so it is called Open hashing.

→ How to avoid open hashing?

- minimize collision
- be easy and quick to compute
- key values inserted evenly in the hash table
- have a high load factor for a given set of keys.

Avg TC = O(1 + load factor)

Worst TC = O(N)

Load factor = Total elements in hash table / Size of hash table

2. Closed Hashing (Open Addressing)

The main idea is to keep all the data in the same table to achieve it, we search for alternative slots in the hash table until it is found. There are 3 strategies: -

a) Linear Probing.

We take a fixed size hash table and every time we face a collision we linearly traverse the table in a cyclic manner to find the next empty slot.

Hash function => $H(x, i) = (H(x) + i) \% N$.

N -> size of table

i -> linearly increasing variable starting from 1.

Problem: Even though it is easy to compute, implement but it suffers from Primary Clustering. It occurs bcz table is large enough to get an empty cell. worst TC = O(n)

b) Quadratic Probing.

Main idea remains the same, only difference is search for empty slot is done quadratically. It solves the problem of clustering (Primary)

Hash func: $H(x, i) = (H(x) + i \wedge 2) \% N$

Problems:

① for 2 keys x_1 and x_2 if the hash value is same then they will have same slot causing Secondary Clustering.

② As we are searching quadratically chances are that all slots are not traversed. No guarantee to find an empty cell if the table is more than half-filled.

Double Hashing: It is one of the best technique for hashing with open addressing. The permutation formed by double hashing is like a random permutation to almost eliminate chances of cluster forming as it uses a secondary hash func.

$H(x, i) = (H1(x) + i * H2(x)) \% N$

For $H1(X) = X \% N$ a good $H2(x) = P - (X \% P)$,
P → prime no. smaller than N.

Advantages of Closed Hashing:

- Better cache performance. as all the data is stored in same table.
- Easy to implement as no pointers are involved.
- Strategies can be chosen acc. to usecase.

⇒ Linear probing provides best cache performance. but clustering is the problem

Double hashing provides poor cache performance but less or no clustering.

Quadratic lies b/w both the fields.

Q. What is Executor?

It is an interface that represents an object that execute provided tasks. It initiates and controls the execution of threads. Depending on the particular implementation a new thread or existing thread is used.

An executor can invoke the submitted task instantly in the invoking thread. If the executor can't accept the task for execution, it will throw RejectedExecutionException.

Q What is ExecutorService?

It is an interface which allows us to execute tasks on threads asynchronously. It helps in maintaining a pool of threads and assign them tasks. It also provides the facility to queue up tasks until there is a free thread available.

→ Single-Threaded

```
ExecutorService executor = Executors.newSingleThreadExecutor(ThreadFactory t)
```

→ Fixed Size

```
ExecutorService executor = Executors.newFixedThreadPool(10);
executorService.execute(new Runnable()
        {
            public void run()
            {
                sop("Asynchronous task");
            }
        });
executorservice.shutdown();
```

or `executorservice.submit(new Task());`

→ .shutdown() : waits until all submitted tasks completes execution.

→ shutdownNow(): terminate all actively executing tasks & halts the process of waiting task.

Executor	ExecutorService
- Parent Interface	- extends Executor.
- execute()	- submit()
- do not return	- return future object
- accepts runnable object.	- accepts runnable & callable objects.

Scheduled Executor Service

It can perform tasks periodically. We can schedule tasks to run after a delay or execute repeatedly with a fixed interval of time in blw each execution. Zero or -ve value signifies req needs to be executed instantly.

- scheduleAtFixedRate(Runnable command, long delay, long Period, TimeUnit unit);
- ScheduleWithFixedDelay(,,)

Future

Future is an interface used to represent the result of an asynchronous operation. It provides the methods to check if the compution is completed or not, to wait for its completions and to retrive the result of the computation. Once the task or computation is completed one cannot cancel the computation.

Cyclic Barrier

It works same a countdown latch except that we can reuse it. It allows multiple threads to wait for each other using await() before invoking the final task. for fixed no of parities.

Thread Factory

It acts as a thread pool which creates a new thread on demand

Delay Queue

It is an infinite-size blocking queue where an element can only be pulled of it's expiration time is completed. The topmost element will have the most amount delay and it will be polled first.

Phaser

It is more flexible solution that CyclicBarrier and CountDownLatch - used to act as a reusable barrier on which the dynamic no. of threads needs to wait before continuing execution.

Lock

Lock is a utility for blocking other threads from accessing a certain segment of code, apart from the thread that's executing it currently.

Q Advantage of Reentrant Lock?

- Ability to timeout while waiting for lock using Lock.tryLock()
- Ability to lock interruptibly.
- Fairness: longest waiting thread gets the lock first.
- Flexibility to try for lock w/o locking.
- API to get list of waiting threads for lock

Q Disadvantages

- wrapping method body inside try-finally block, which make code unreadable and hides business logic bcz finally is req to call unlock method.
- If programmer forget to release the lock in finally.

Q Why method Reference when we have Lambda?

- MR is same as lambda exp that refers a method w/o executing it.
- makes code more short and readable

* Constructor Overloading is done when we need to initiate the object in diff ways

* Constructor overriding is not possible bcz when we create constructor of super class in sub class it treats it as a method and expect return type but constructor do not have return type.

Thread Pool

- It is comprises of a set of pre-allocated threads that are adept at executing tasks on demand.
- It minimizes resource consumption, as new thread is not created everytime.
- A

CountdownLatch is suitable for one-time iteration with a fixed no. of parties.

Cyclic Barrier is suitable for one-time and cyclic iterations with a fixed no. of parties.

Phaser is suitable of one-time and cyclic iterations with a variable no. of parties.

Singleton Design Pattern.

Purpose is to control object creation.

=> Eager Initialization

```
class EagerSingleton
{
    private static EagerSingleton Instance = new
                                    EagerSingleton();

    Private EagerSingleton() { }

    public static EagerSingleton getInstance()
    {
        return instance;
    }
}
```

Disadvantage: If getInstance() is not called then also the Instance will be created and remain in Heap memory.

If some exception occurs while creating an object, either use try-catch or static block initialization.

⇒ Static Block Initialization

```
public class StaticSingleton
{
    private static StaticSingleton instance;
    Private Static Singleton () { }
    Static {
        try {
            instance = new staticSingleton();
        }
        catch (Exception e)
        { Throw new RuntimeException
            (" Excep occured ");
        }
    }
    Public static staticSingleton getInstance()
    { return Instance; }
}
```

⇒ Lazy Initialization.

```
Public class LazySingleton {
    private static LazySingleton instance;
    private LazySingleton () { }
    public static LazySingleton getInstance()
    { if(instance == null)
        { instance = new LazySingleton();
        }
        retun instance;
    }
}
```

This scenario is not Thread safe.

=> Using synchronized keyword.

```
public class ThreadSafeSingleton{
    Private class ThreadSafeSingleton instance;
    private ThreadSafeSingleton(){ }
    public static synchronized ThreadSafeSingleton getInstance()
    {   if(instance==null)
        { instance= new ThreadSafeSingleton();
        }
        return instance;
    }
}
```

→ Thread safe but bad performance as getInstance() gets blocked by one thread.

=> Double-Checked Locking.

```
public static DoubleLockingSingleton getInstance()
{   if(instance==null)
    {   synchronized (DoubleLockingSingleton.class)
        {   instance= new DoubleLocking();
        }
    }
    return instance;
}
```

This can also be blocked.

```
Public static DoubleLockingf getInstance()
{   if( instance == null)
    {
       synchronized(DoubleLockingf.class)
       {    if(instance == null)
            {   instance = new DoubleLockingf();
            }
       }
       return instance;
    }
}
```