**Multithreading**

**1) What do you understand by a thread and give one example in c++?**

In every application there is a default thread which is main(), inside this, we create other threads.

A thread is also known as a lightweight process. The idea is to achieve parallelism by dividing a process into multiple threads.

**1) what std::thread accepts in the constructor?**

We can attach a callback with the std::thread object, that will be executed when this new thread starts. These callbacks can be,

* + Function Pointer
  + Function Objects
  + Lambda functions

**2) Creating a thread using Function Pointer?**

**#include <iostream>**

**#include <thread>**

**void thread\_function()**

**{**

**for(int i = 0; i < 10000; i++);**

**std::cout<<"thread function Executing"<<std::endl;**

**}**

**int main()**

**{**

**std::thread threadObj(thread\_function);**

**for(int i = 0; i < 10000; i++);**

**std::cout<<"Display From MainThread"<<std::endl;**

**threadObj.join();**

**std::cout<<"Exit of Main function"<<std::endl;**

**return 0;**

**}**

**3) Creating a thread using Function Objects?**

#include <iostream>

#include <thread>

class DisplayThread

{

public:

void operator()()

{

for(int i = 0; i < 10000; i++)

std::cout<<"Display Thread Executing"<<std::endl;

}

};

int main()

{

std::thread threadObj( (DisplayThread()) );

for(int i = 0; i < 10000; i++)

std::cout<<"Display From Main Thread "<<std::endl;

std::cout<<"Waiting For Thread to complete"<<std::endl;

threadObj.join();

std::cout<<"Exiting from Main Thread"<<std::endl;

return 0;

}

**4) Creating a thread using Lambda functions ?**

#include <iostream>

#include <thread>

int main()

{

int x = 9;

std::thread threadObj([]{

for(int i = 0; i < 10000; i++)

std::cout<<"Display Thread Executing"<<std::endl;

});

for(int i = 0; i < 10000; i++)

std::cout<<"Display From Main Thread"<<std::endl;

threadObj.join();

std::cout<<"Exiting from Main Thread"<<std::endl;

return 0;

}

**5) what is std::thread join?**

Once a thread is started, another can wait for this new thread to finish. For this another need to call the join function.

**6) What is a thread using std::thread::detch?**

Detached threads are also called daemon/Background threads. To detach a thread we need to call the std::detach() function

Be careful with calling detach() and join() on Thread Handles:

**case 1: Never call join() or detach() on std::thread object with no associated executing method.**

Example:

std::thread threadObj((workerThread()));

threadaobj.join();

threadobj.join(); //It will cause program to terminate

**Case 2 : Never forget to call either join or detach on a std::thread object with associated executing**

**thread**

**7) Passing simple arguments to std::thread in c++11?**

**#include <iostream>**

**#include <string>**

**#include <thread>**

**void threadfunction(int x, std::string str)**

**{**

**std::cout<<"Passed Number = "<<x<<std::endl;**

**std::cout<<"Passed string = "<<str<<std::endl;**

**}**

**int main()**

**{**

**int x = 10;**

**std::string str = "simple string ";**

**std::thread threadobj(threadfunction, x, str);**

**threadObj.join();**

**return 0;**

**}**

**8) How not to pass arguments to threads in C++11 ?**

Don't pass the address of variables from the local stack to the thread's callback function. because it might be possible that the local variable in thread1 goes out of scope but thread2 is still trying to access it through its address.

#include <iostream>

#include <thread>

using namespace std;

using time1 = std::chrono::milliseconds;

//using Tsleep = std::this\_thread::sleep\_for;

void threadfunction(int \*p)

{

cout<<"inside thread function Id : "<<\*p<<endl;

time1 dura(1000);

std::this\_thread::sleep\_for(dura);

\*p = 20;

}

void createThread()

{

int val = 10;

std::cout<<"inside createThread : i = "<<val<<endl;

std::thread th(threadfunction, &val);

if(th.joinable())

th.detach(); // here detached the thread means this thread will not wait for another thread

std::cout<<"inside createThread : i = "<<val<<endl;

}

int main()

{

createThread();

time1 tm(1000);

std::this\_thread::sleep\_for(tm);

cout<<"Exit from main : "<<endl;

return 0;

}

**9)How to pass references to std::thread in C++11?**

#include <iostream>

#include <thread>

using namespace std;

void threadfunction(const int &p)

{

int &y = const\_cast<int&>(p);

y++;

cout<<"Inside the thread function : "<<y<<endl;

}

int main()

{

int x = 9;

std::cout<<"inside main function : "<<x<<endl;

//std::thread th(threadfunction,x); //

std::thread th(threadfunction,std::ref(x)); //

if(th.joinable())

th.join();

std::cout<<"Inside the main function : "<<x<<endl;

return 0;

}

Even if threadCallback accepts arguments as reference but still changes are not visible outside the thread.

It’s because x in the thread function threadCallback is a reference to the temporary value copied at the new thread’s stack.

**How to fix this** ?

Using std::ref()

**10) How to assign a pointer to the member function of a class as a thread function?**

#include <iostream>

#include <thread>

using namespace std;

class sample

{

public:

sample() = default;

sample(const sample& obj) = default;

void print(int &x)

{

x++;

cout<<"Print : "<<x<<endl;

}

};

int main()

{

int x = 9;

sample obj;

std::cout<<"inside main function : "<<x<<endl;

std::thread th(&sample::print,&obj,std::ref(x)); //

if(th.joinable())

th.join();

std::cout<<"Inside the main function : "<<x<<endl;

return 0;

}

**11) what is Race condition?**

The Race condition is a kind of bug that occurs in multithreaded applications.

When two or more threads perform a set of operations in parallel, that access the same memory location. Also, one or more threads out of them modify the data in that memory location, which can lead to unexpected results.

**12) How to fix the Race condition?**

To fix the race condition problem we need to use the lock mechanism i.e each thread needs to acquire a lock before modifying or reading the shared data and after modifying the data each thread should unlock the lock.

**13) what is mutex?**

we can fix the race condition by using mutex lock while the thread accessing the memory once complete the accessing then unlock the lock then the other thread can access the same data. but if forgot to unlock the mutex lock then other threads will remain to wait.

This kind of scenario can happen in case some exceptions came after locking the mutex. To avoid such scenarios we should use std::lock\_guard.

**13) what mutex try lock ?**

* try\_lock() tries to lock the mutex. Returns immediately. On successful lock acquisition return true otherwise returns false.
* If try\_lock() is not able to lock the mutex, then it doesn't get blocked that's why it is called non-blocking.
* If try\_lock is called again by the same thread which owns the mutex, the behavior is undefined. It is a deadlock situation with undefined behavior.

**Note: If you want to lock the same mutex by the same thread more than once, then go for recursive\_mutex.**

**Example :**

#include <iostream>

#include <thread>

#include <mutex>

int counter = 0;

std::mutex mut;

void increasethecounter()

{

for(int i = 0 ; i < 100000; i++)

{

if(mut.try\_lock())

{

counter++;

}

mut.unlock();

}

}

int main()

{

std::thread th(&increasethecounter);

std::thread th1(&increasethecounter);

th1.join();

th.join();

std::cout<<"counter value : "<<counter<<std::endl;

return 0;

}

**13) what is std::try\_lock() in c++11?**

* std::try\_lock() tries to lock all the lockable objects passed in it one by one in the given order.

syntax : std::try\_lock(m1,m2,m3,m4,...,mn);

* On success this function returns -1 otherwise it will return 0- based mutex index number which it could not lock.
* If it fails to lock any of the mutexes then it will release all the mutex it locked before.
* If a call to try\_lock results in an exception unlock is called for any locked objects before rethrowing.

**14) what is Timed mutex in c++?**

* std::timed\_mutex is blocked till the timeout time or the lock is acquired and returns true if successful otherwise false.
* Member functions:
* Lock
* Try\_lock
* Try\_lock\_for
* Try\_lock\_until
* Unlock

**Try\_lock\_for:** waits until the specified timeout duration have elapsed or the lock is acquired, whichever comes first.

On successful lock acquisition returns true, otherwise returns false.

**Example**:

#include <iostream>

#include <thread>

#include <mutex>

#include <chrono>

int amount = 0;

std::timed\_mutex m;

void increament(int i)

{

if(m.try\_lock\_for(std::chrono::seconds(2)))

{

amount++;

std::this\_thread::sleep\_for(std::chrono::seconds(1));

std::cout<<"Thread entered : "<<i<<std::endl;

m.unlock();

}

else

{

std::cout<<"thread timeout : "<<i<<std::endl;

}

}

int main()

{

std::thread th1(increament,1);

std::thread th2(increament,2);

th1.join();

th2.join();

std::cout<<"Amount : "<<amount<<std::endl;

return 0;

}

**15) what is recursive mutex in c++?**

* It is the same as mutex but, the same thread can lock one mutex multiple times using a recursive mutex.
* If thread T1 first call lock/try\_lock on recursive mutex ml, then ml is locked by T1, now as T1 is running in recursion T1 can call lock/try\_lock any number of times there is no issue.
* But If T1 has acquired 10 times lock/try\_lock on mutex m1 then thread t1 will have to unlock it 10 times otherwise no other thread will be able to lock mutex ml. It means recursive mutex keeps counting how many times it was locked so that many times it should be unlocked.
* How many times we can lock recursive\_mutex is not defined but when that number reaches and if we were calling lock() it will return std::system\_error or if we were calling try\_lock() then it will return false.

BOTTOM LINE:

* It is similar to mutex but has an extra facility that can be locked multiple times by the same thread.
* If we can avoid recursive\_mutex then we should because it brings overhead to the system.
* It can be used in a loop also.

**Example:**

**#include <iostream>**

**#include <thread>**

**#include <mutex>**

**#include <chrono>**

**int amount = 0;**

**std::recursive\_mutex m;**

**void recursion(int thread\_id, int count)**

**{**

**if(count < 0)**

**return;**

**m.lock();**

**std::cout<<"Thread Id : " <<thread\_id<<" "<<amount++<<std::endl;**

**recursion(thread\_id, --count);**

**m.unlock();**

**std::cout<<"Unlock thread : "<<thread\_id<<std::endl;**

**}**

**int main()**

**{**

**std::thread th1(recursion,1, 10);**

**std::thread th2(recursion,2, 10);**

**th1.join();**

**th2.join();**

**std::cout<<"Amount : "<<amount<<std::endl;**

**return 0;**

**}**

**16) what is std::lock\_guard ?**

It wraps the mutex inside the object and locks the attached mutex in the constructor. when its destructor is called it releases the mutex.

* It is a very lightweight wrapper for owning mutex on scoped basis.
* It acquired mutex to lock the moment you create the object of lock\_guard.
* It automatically removes the lock when goes out of scope.
* You can not explicitly unlock the lock\_guard.
* You cannot copy lock\_guard.

**Example:**

class Wallet

{

int mMoney;

std::mutex mutex;

public:

Wallet() :mMoney(0){}

int getMoney() { return mMoney; }

void addMoney(int money)

{

std::lock\_guard<std::mutex> lockGuard(mutex);

// In constructor it locks the mutex

for(int i = 0; i < money; ++i)

{

// If some exception occurs at this

// poin then destructor of lockGuard

// will be called due to stack unwinding.

//

mMoney++;

}

// Once function exits, then destructor

// of lockGuard Object will be called.

// In destructor it unlocks the mutex.

}

};

**16) what is std::lock in c++?**

* It is used to lock multiple mutexes at the same time.
* Std::lock(m1,m2,m3,m4);
* All the arguments are locked via a sequence of calls to lock(), try\_lock(), or unlock() on each argument.
* The Order of locking is not defined (it will try to lock provided mutex in any order and ensure that there is no deadlock). It is a blocking call.

**16) what is uniq\_lock in c++?**

i) The class unique\_lock is a mutex ownership wrapper.

ii) It allows:

* can have different locking strategies.
* time-constrained attempts at locking (try\_lock\_for, try\_lock\_until).
* recursive locking
* transfer of lock ownership (move not copy)
* condition variables.

**Locking strategies**

Type Effects

deffer\_lock do not acquire ownership of the mutex

try\_to\_lock try to acquire ownership of the mutex without blocking

adopt\_lock assumes the calling thread already has ownership of the mutex

#include <iostream>

#include <thread>

#include <mutex>

#include <chrono>

int amount = 0;

std::mutex m;

void task(int thread\_id, int count)

{

//std::unique\_lock lg(m,std::defer\_lock); // this we do not call mutex, we need to call explicitly

//lg.lock();

std::unique\_lock lg(m); // This will call the mutex no need to write lock statement here

for(int i = 0 ; i < count ; i++)

std::cout<<"lock Thread Id : " <<thread\_id<<" "<<amount++<<std::endl;

}

int main()

{

std::thread th1(task,1, 10);

std::thread th2(task,2, 10);

th1.join();

th2.join();

std::cout<<"Amount : "<<amount<<std::endl;

return 0;

}

**15) what are condition variables?**

A condition variable is a kind of event used for signaling between 2 threads. One thread can wait for it to get signaled, while the other thread can signal this.

Condition variables are used for two purposes.

* Notify other threads
* Waiting for some condition
* Condition variables allow running threads to wait on some conditions and once those conditions are met, the waiting thread is notified using:
  + - Notify\_one();
    - Notify\_all();
* You need a mutex to use the condition variable
* If some thread wants to wait on some condition, then it has to do these things
* Acquire the mutex lock using std::unique\_lock<std::mutex> lock(m)
* Execute wait, wait\_for, or wait\_until. The wait operation automatically releases the mutex and suspends the execution of the thread.
* When the condition variable is notified, the thread is awakened, and the mutex is automatically reacquired

**Example:**

#include <condition\_variable>

#include <iostream>

#include <thread>

#include <functional>

#include <mutex>

#include <condition\_variable>

using namespace std::placeholders;

class Application

{

std::mutex m\_mutex;

std::condition\_variable m\_condVar;

bool m\_bDataLoaded;

public:

Application()

{

m\_bDataLoaded = false;

}

void loadData()

{

// Make This Thread sleep for 1 Second

std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));

std::cout<<"Loading Data from XML"<<std::endl;

// Lock The Data structure

std::lock\_guard<std::mutex> guard(m\_mutex);

// Set the flag to true means data is loaded

m\_bDataLoaded = true;

// Notify the condition variable

m\_condVar.notify\_one();

}

bool isDataLoaded()

{

return m\_bDataLoaded;

}

void mainTask()

{

std::cout<<"Do Some Handshaking"<<std::endl;

// Acquire the lock

std::unique\_lock<std::mutex> mlock(m\_mutex);

// Start waiting for the Condition Variable to get signaled

// Wait() will internally release the lock and make the thread to block

// As soon as condition variable get signaled, resume the thread and

// again acquire the lock. Then check if condition is met or not

// If condition is met then continue else again go in wait.

m\_condVar.wait(mlock, std::bind(&Application::isDataLoaded, this));

std::cout<<"Do Processing On loaded Data"<<std::endl;

}

};

int main()

{

Application app;

std::thread thread\_1(&Application::mainTask, &app);

std::thread thread\_2(&Application::loadData, &app);

thread\_2.join();

thread\_1.join();

return 0;

}

**16) what are std::future and std::promise?**

std::future is a class template and its object stores the future value.

Now, what the hell is this future value?

Actually, a std::future object internally stores a value that will be assigned in the future and it also provides a mechanism to access that value i.e. using the get() member function. But if somebody tries to access this associated value of future through get() function before it is available, then get() function will block till the value is not available.

std::promise is also a class template and its object promises to set the value in the future. Each std::promise object has an associated std::future object that will give the value once set by the std::promise object.

A std::promise object shares data with its associated std::future object.

#include <iostream>

#include <thread>

#include <future>

void initiazer(std::promise<int> \* promObj)

{

std::cout<<"Inside Thread"<<std::endl; promObj->set\_value(35);

}

int main()

{

std::promise<int> promiseObj;

std::future<int> futureObj = promiseObj.get\_future();

std::thread th(initiazer, &promiseObj);

std::cout<<futureObj.get()<<std::endl;

th.join();

return 0;

}

**17) what is std::async() function?**

std::async() is a function template that accepts a callback(i.e. function or function object) as an argument and potentially executes them asynchronously.

**syntax:**

**template<class fn, class ...Args>**

**future<typename result\_of<Fn(Args...)>::type>async (launch policy, Fn&& fn, Args&& ...args);**

std::async returns a std::future<T>, that stores the value returned by function object executed by function can be passed to std::async() as arguments after the function pointer argument.

The First argument is that std::async is a launch policy, It controls the asynchronous behavior of std::async. we can create std::async with 3 different launch policies i.e.

**std::launch::async:** It guarantees the asynchronous behavior i.e passed function will be executed in a separate thread.

**std::launch::deferred:** Nonasynchronous behavior i.e. Function will be called when other thread will call get() in future to access the shared state.

**std::launch::async|std::launch::deferred:** It’s the default behavior. with this launch policy it can run asynchronously or not depending on the load on system. But we have no control over it.

**If we do not specify a launch policy. Its behavior will be similar to std::launch::async | std::launch::deferred.**

**We can pass any callback in std::async i.e.**

* **Function Pointer**
* **Function object**
* **Lambda Function.**

**18) why do we need std::async()?**

Suppose we have to fetch some data from DB and some from files in file-system. then I need to merge both the strings and print.

In a single thread we will do like this.

**#include <iostream>**

**#include <thread>**

**#include <future>**

**#include <chrono>**

**using namespace std::chrono;**

**std::string FetchDatafromDB(std::string strval)**

**{**

**std::this\_thread::sleep\_for(seconds(5));**

**return "DB\_"+strval;**

**}**

**std::string FetchDataFromFile(std::string strval)**

**{**

**std::this\_thread::sleep\_for(seconds(5));**

**return "File\_"+strval;**

**}**

**int main()**

**{**

**system\_clock::time\_point start = system\_clock::now();**

**std::string strDBData = FetchDatafromDB("DataBase");**

**std::string strFileData = FetchDataFromFile("DataFile");**

**auto end = system\_clock::now();**

**auto diff = duration\_cast < std::chrono::seconds > (end - start).count();**

**std::cout<<"Total time taken = "<<diff<<" seconds "<<std::endl;**

**std::cout<<strDBData<<std::endl;**

**std::cout<<strFileData<<std::endl;**

**return 0;**

**}**

**output:**

**Total time taken = 10 seconds**

**DB\_DataBase**

**File\_DataFile**

As both the functions fetchDataFromDB() & fetchDataFromFile() takes 5 seconds each and are running in a single thread so, the total time consumed will be 10 seconds.

Now as fetching data from DB and file are independent of each other and also time-consuming. So, we can run them in parallel.

One way to do this creates a new thread and pass a promise as an argument to the thread function and fetch data from the associated std::future object in the calling thread.

The other easy way is using std::async

Calling std::async with function pointer as callback.

**std::future<std::string> resultFromDB = std::async(std::launch::async, fetchDataFromDB, "Data");**

**// Do Some Stuff**

**//Fetch Data from DB**

**// Will block till data is available in future<std::string> object.**

**std::string dbData = resultFromDB.get();**

**std::async() does the following things,**

**It automatically creates a thread (Or picks from the internal thread pool) and a promise object for us.**

**Then passes the std::promise object to the thread function and returns the associated std::future object.**

**When our passed argument function exits then its value will be set in this promise object, so eventually return value will be available in the std::future object.**

**Now change the above example and use std::async to read data from DB asynchronously i.e.**

**#include <iostream>**

**#include <string>**

**#include <chrono>**

**#include <thread>**

**#include <future>**

**using namespace std::chrono;**

**std::string fetchDataFromDB(std::string recvdData)**

**{**

**// Make sure that function takes 5 seconds to complete**

**std::this\_thread::sleep\_for(seconds(5));**

**//Do stuff like creating DB Connection and fetching Data**

**return "DB\_" + recvdData;**

**}**

**std::string fetchDataFromFile(std::string recvdData)**

**{**

**// Make sure that function takes 5 seconds to complete**

**std::this\_thread::sleep\_for(seconds(5));**

**//Do stuff like fetching Data File**

**return "File\_" + recvdData;**

**}**

**int main()**

**{**

**// Get Start Time**

**system\_clock::time\_point start = system\_clock::now();**

**std::future<std::string> resultFromDB = std::async(std::launch::async, fetchDataFromDB, "Data");**

**//Fetch Data from File**

**std::string fileData = fetchDataFromFile("Data");**

**//Fetch Data from DB**

**// Will block till data is available in future<std::string> object.**

**std::string dbData = resultFromDB.get();**

**// Get End Time**

**auto end = system\_clock::now();**

**auto diff = duration\_cast < std::chrono::seconds > (end - start).count();**

**std::cout << "Total Time Taken = " << diff << " Seconds" << std::endl;**

**//Combine The Data**

**std::string data = dbData + " :: " + fileData;**

**//Printing the combined Data**

**std::cout << "Data = " << data << std::endl;**

**return 0;**

**}**

**Output:**

**Total Time Taken = 5 Seconds**

**Data = DB\_Data :: File\_Data**

**19) Calling std::async with Function Object as a callback?**

struct DataFetcher

{

std::string operator()(std::string recvdData)

{

// Make sure that function takes 5 seconds to complete

std::this\_thread::sleep\_for (seconds(5));

//Do stuff like fetching Data File

return "File\_" + recvdData;

}

};

//Calling std::async with function object

std::future<std::string> fileResult = std::async(DataFetcher(), "Data");

**20) Calling std::async with Lambda function as callback?**

**//Calling std::async with lambda function**

**std::future<std::string> resultFromDB = std::async([](std::string recvdData){**

**std::this\_thread::sleep\_for (seconds(5));**

**//Do stuff like creating DB Connection and fetching Data**

**return "DB\_" + recvdData; }, "Data");**

**21) what is std::packaged\_task?**

std::packaged\_task<> is a class template and represents an asynchronous task. It encapsulates,

i) A callable entity i.e either function, lambda function, or function object.

ii) A shared state that stores the value returned or thrown exception by associated callback.

Need of std::packaged\_task<>

std::string getDataFromDB(std::string token)

{

std::string data = "Data fetched from DB by filter :: "+ token;

return data;

}

**22) Using packaged\_task<> with function to create Asynchronous tasks?**

std::packaged\_task<> can wrap around a normal function and make it applicable to run as an asynchronous function.

When std::packaged\_task<> is called in a separate thread, it calls the associated callback and stores the return value/exception in its internal shared state. This value can be accessed in another thread or main function through std::future<> object.

* creating std::packaged\_task<> object.

std::packaged\_task<std::string (std::string)> task(getDataFromDB);

* Fetch the future object from it,
* std::future<std::string> result = task.get\_future();
* Passing packaged\_task<> to a thread
* std::thread th(std::move(task),"Arg");

Example:

#include <iostream>

#include <thread>

#include <future>

#include <string>

// Fetch some data from DB

std::string getDataFromDB( std::string token)

{

// Do some stuff to fetch the data

std::string data = "Data fetched from DB by Filter :: " + token;

return data;

}

struct DBDataFetcher

{

std::string operator()(std::string token)

{

// Do some stuff to fetch the data

std::string data = "Data From " + token;

return data;

}

};

int main()

{

// Create a packaged\_task<> that encapsulated the callback i.e. a function

std::packaged\_task<std::string (std::string)> task(getDataFromDB);

// Create a packaged\_task<> that encapsulated a lambda function

std::packaged\_task<std::string (std::string)> task([](std::string token){

// Do some stuff to fetch the data

std::string data = "Data From " + token;

return data;

});

// Create a packaged\_task<> that encapsulated a function pointer

std::packaged\_task<std::string (std::string)> task(std::move(DBDataFetcher()));

// Fetch the associated future<> from packaged\_task<>

std::future<std::string> result = task.get\_future();

// Pass the packaged\_task to thread to run asynchronously

std::thread th(std::move(task), "Arg");

// Join the thread. Its blocking and returns when thread is finished.

th.join();

// Fetch the result of packaged\_task<> i.e. value returned by getDataFromDB()

std::string data = result.get();

std::cout << data << std::endl;

return 0;

}

**23) what is the difference between sleep and wait?**

|  |  |
| --- | --- |
| **Sleep** | **wait** |
| Sleep is “ I am done with my timeslice, and please don’t give me another one for at least “n” milliseconds.” The OS doesn’t even try to schedule the sleeping thread until the requested time has passed | WAIT is “ I am done with my timeslice, and please don’t give me another timeslice until someone calls **notify()**” The OS won’t even try to schedule your task unless someone calls notify(). |
| **It will keep lock and sleep**  **Sleep is direct to thread, it is a thread function** | **It releases the lock and wait**  **It is on the condition variable, it is like there is a condition variable in the thread, and wait is applied to the cv but it ends up putting the thread in a waiting state.** |

**Q) What are the 6 synchronizations primitive available in Multithreading?**

Mutex

Join

Condition Variable

Barriers

Spin Lock

Semaphore

**Q) Name the design pattern for the thread?**

Some popular test cases in the current IT industry

Thread Pool (Boos and Worker)

Peer (Work Crew)

Pipeline

**Q) Brief me about the available models in Multithreading?**

Many to many relations

Many to One relation

One to one relation