**Multithreading**

**What?**

Multithreading is a specialized form of multitasking and a multitasking is the feature that allows your computer to run two or more programs concurrently.

**Why?**

A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

**How?**

Multithreading support was introduced in C+11. Prior to C++11, we had to use POSIX threads or p threads library in C.

#include <thread>

thread th1(foo, 3);

Here foo is place for function object/ function pointer/ lambda expression.

Launching thread using function pointer

void foo(param)

{

// Do something

}

// The parameters to the function are put after the comma

std::thread thread\_obj(foo, params);

Launching thread using lambda expression

// Define a lamda expression

auto f = [](params) {

// Do Something

};

// Pass f and its parameters to thread

// object constructor as

std::thread thread\_object(f, params);

Launching threads using function objects

// Define the class of function object

class fn\_object\_class {

// Overload () operator

void operator()(params)

{

// Do Something

}

}

// Create thread object

std::thread thread\_object(fn\_class\_object(), params)

**Waiting for thread to finish**

1. Join

2. Detach

**Join** is used to make the main program wait until the execution of the other thread are completed.

**Detach,** when the main program end that threads that are associated with it are detached so that they continue their flow. But once detached no operation can be perfomed over the detached thread.

**Example:**

// CPP program to demonstrate multithreading

// using three different callables.

#include <iostream>

#include <thread>

using namespace std;

// A dummy function

void foo(int Z)

{

for (int i = 0; i < Z; i++) {

cout << "Thread using function"

" pointer as callable\n";

}

}

// A callable object

class thread\_obj {

public:

void operator()(int x)

{

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

cout << "Thread using function"

" object as callable\n";

}

};

int main()

{

cout << "Threads 1 and 2 and 3 "

"operating independently" << endl;

// This thread is launched by using

// function pointer as callable

thread th1(foo, 3);

// This thread is launched by using

// function object as callable

thread th2(thread\_obj(), 3);

// Define a Lambda Expression

auto f = [](int x) {

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

cout << "Thread using lambda"

" expression as callable\n";

};

// This thread is launched by using

// lamda expression as callable

thread th3(f, 3);

// Wait for the threads to finish

// Wait for thread t1 to finish

th1.join();

// Wait for thread t2 to finish

th2.join();

// Wait for thread t3 to finish

th3.join();

return 0;

}

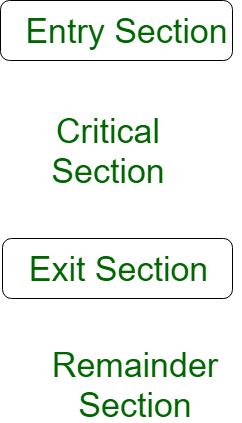
To compile:

g++ pgm.cpp -o pgm -lpthread -std=c++11

**Critical section:**

During concurrency, if one thread tries to change the value of shared data at the same time as another thread tries to read the value (i.e. data race across threads), the result is unpredictable.

So the area where these shared data are modified is known as critical section.



**------------------------------------------------------------------------------------------------------------------------**

**Race condition:**

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.

Example:

if (x == 5) // The "Check"

{

y = x \* 2; // The "Act"

// If another thread changed x in between "if (x == 5)" and "y = x \* 2" above,

// y will not be equal to 10.

}

How to handle them?

Race condition can be handled by Mutex or Semaphores. They act as a lock allows a process to acquire a resource based on certain requirements to prevent race condition

How do you prevent them from occurring?

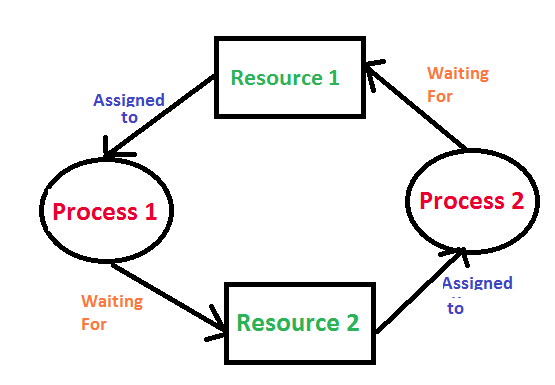
There are various ways to prevent race condition, such as Critical Section Avoidance.

1. No two processes simultaneously inside their critical regions. (Mutual Exclusion)
2. No assumptions are made about speeds or the number of CPUs.
3. No process running outside its critical region which blocks other processes.
4. No process has to wait forever to enter its critical region. (A waits for B resources, B waits for C resources, C waits for A resources)

--------------------------------------------------------------------------------

**Dead lock:**

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.



A deadlock can occur if and only if all the following conditions in a system are fulfilled simultaneously. These conditions are called the **Coffman conditions**:

* **Mutual exclusion** states that each resource can be assigned to only one process at a time
* **Circular wait** means that a process is holding a resource and requires more of the resources which are being held by other processes
* **Resource holding** is when one or more processes can hold and wait for other resources to become available for use
* **No preemption** means the resources that have already been granted access in advance and cannot be taken away at that time

If you are aware of these four conditions, then you can follow them and hopefully avoid a deadlock from occurring altogether. There can also be a slight variation in a deadlock situation in which there are two or more processes that are in a constantly changing state, which is known as a **livelock**. The critical difference here is that the processes at play have not actually stopped at all but have instead just become too busy to respond to each other.

------------------------------------------------------------------------------------------------------------------------

**Mutex:**

A *mutex* is a *lockable objec*[*t*](http://www.cplusplus.com/Lockable) that is designed to signal when critical sections of code need exclusive access, preventing other threads with the same protection from executing concurrently and access the same memory locations.

Example:

// mutex example

#include <iostream> // std::cout

#include <thread> // std::thread

#include <mutex> // std::mutex

std::mutex mtx; // mutex for critical section

void print\_block (int n, char c) {

// critical section (exclusive access to std::cout signaled by locking mtx):

mtx.lock();

for (int i=0; i<n; ++i) { std::cout << c; }

std::cout << '\n';

mtx.unlock();

}

int main ()

{

std::thread th1 (print\_block,50,'\*');

std::thread th2 (print\_block,50,'$');

th1.join();

th2.join();

return 0;

}

------------------------------------------------------------------------------------------------------------------------

**Semaphore:**

Semaphore is simply a variable. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.  
The two most common kinds of semaphores are counting semaphores and binary semaphores. Counting semaphore can take non-negative integer values and Binary semaphore can take the value 0 & 1. only.