C++ MULTITHREADING

Multithreading is a specialized form of multitasking and a multitasking is the feature that allows your computer to run two or more programs concurrently. In general, there are two types of multitasking: process-based and thread-based.

Process-based multitasking handles the concurrent execution of programs. Thread-based multitasking deals with the concurrent execution of pieces of the same program.

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.

C++ does not contain any built-in support for multithreaded applications. Instead, it relies entirely upon the operating system to provide this feature.

This tutorial assumes that you are working on Linux OS and we are going to write multi-threaded C++ program using POSIX. POSIX Threads, or Pthreads provides API which are available on many Unix-like POSIX systems such as FreeBSD, NetBSD, GNU/Linux, Mac OS X and Solaris.

Creating Threads:

There is following routine which we use to create a POSIX thread:

```
#include <pthread.h>
pthread_create (thread, attr, start_routine, arg)
```

Here, **pthread_create** creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code. Here is the description of the parameters:

Parameter	Description
thread	An opaque, unique identifier for the new thread returned by the subroutine.
attr	An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.
start_routine	The C++ routine that the thread will execute once it is created.
arg	A single argument that may be passed to start_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.

The maximum number of threads that may be created by a process is implementation dependent. Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads.

Terminating Threads:

There is following routine which we use to terminate a POSIX thread:

```
#include <pthread.h>
pthread_exit (status)
```

Here **pthread_exit** is used to explicitly exit a thread. Typically, the pthread_exit() routine is called after a thread has completed its work and is no longer required to exist.

If main() finishes before the threads it has created, and exits with pthread_exit(), the other threads will continue to execute. Otherwise, they will be automatically terminated when main() finishes.

Example:

This simple example code creates 5 threads with the pthread_create() routine. Each thread prints a "Hello World!" message, and then terminates with a call to pthread_exit().

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>

using namespace std;

#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   cout << "Hello World! Thread ID, " << tid << endl;
   pthread_exit(NULL);
}

int main ()
{
   pthread_t threads[NUM_THREADS];
   int rc;
   int i;
   for( i=0; i < NUM_THREADS; i++ ){
      cout << "main() : creating thread, " << i << endl;
      rc = pthread_create(&threads[i], NULL,
                          PrintHello, (void *)i);
      if (rc){
         cout << "Error:unable to create thread," << rc << endl;
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

Compile the following program using -lpthread library as follows:

```
$gcc test.cpp -lpthread
```

Now, execute your program which should generate result something as follows:

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Hello World! Thread ID, 0
Hello World! Thread ID, 1
Hello World! Thread ID, 2
Hello World! Thread ID, 3
Hello World! Thread ID, 4
```

Passing Arguments to Threads:

This example shows how to pass multiple arguments via a structure. You can pass any data type in a thread callback because it points to void as explained in the following example:

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>

using namespace std;

#define NUM_THREADS     5
```

```
struct thread_data{
   int  thread_id;
   char *message;
};

void *PrintHello(void *threadarg)
{
   struct thread_data *my_data;

   my_data = (struct thread_data *) threadarg;

   cout << "Thread ID : " << my_data->thread_id ;
   cout << " Message : " << my_data->message << endl;

   pthread_exit(NULL);
}

int main ()
{
   pthread_t threads[NUM_THREADS];
   struct thread_data td[NUM_THREADS];
   int rc;
   int i;

   for( i=0; i < NUM_THREADS; i++ ){
      cout <<"main() : creating thread, " << i << endl;
      td[i].thread_id = i;
      td[i].message = "This is message";
      rc = pthread_create(&threads[i], NULL,
                          PrintHello, (void *)&td[i]);
      if (rc){
         cout << "Error:unable to create thread," << rc << endl;
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

When the above code is compiled and executed, it produces the following result:

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Thread ID : 3 Message : This is message
Thread ID : 2 Message : This is message
Thread ID : 0 Message : This is message
Thread ID : 1 Message : This is message
Thread ID : 4 Message : This is message
```

Joining and Detaching Threads:

There are following two routines which we can use to join or detach threads:

```
pthread_join (threadid, status)
pthread_detach (threadid)
```

The pthread_join() subroutine blocks the calling thread until the specified threadid thread terminates. When a thread is created, one of its attributes defines whether it is joinable or detached. Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.

This example demonstrates how to wait for thread completions by using the Pthread join routine.

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>
#include <unistd.h>
```

```
using namespace std;

#define NUM_THREADS     5

void *wait(void *t)
{
   int i;
   long tid;

   tid = (long)t;

   sleep(1);
   cout << "Sleeping in thread " << endl;
   cout << "Thread with id : " << tid << "  ...exiting " << endl;
   pthread_exit(NULL);
}

int main ()
{
   int rc;
   int i;
   pthread_t threads[NUM_THREADS];
   pthread_attr_t attr;
   void *status;

   // Initialize and set thread joinable
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   for( i=0; i < NUM_THREADS; i++ ){
      cout << "main() : creating thread, " << i << endl;
      rc = pthread_create(&threads[i], NULL, wait, (void *)i );
      if (rc){
         cout << "Error:unable to create thread," << rc << endl;
         exit(-1);
      }
   }

   // free attribute and wait for the other threads
   pthread_attr_destroy(&attr);
   for( i=0; i < NUM_THREADS; i++ ){
      rc = pthread_join(threads[i], &status);
      if (rc){
         cout << "Error:unable to join," << rc << endl;
         exit(-1);
      }
      cout << "Main: completed thread id :" << i ;
      cout << "  exiting with status :" << status << endl;
   }

   cout << "Main: program exiting." << endl;
   pthread_exit(NULL);
}
```

When the above code is compiled and executed, it produces the following result:

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Sleeping in thread
Thread with id : 0 .... exiting
Sleeping in thread
Thread with id : 1 .... exiting
Sleeping in thread
Thread with id : 2 .... exiting
Sleeping in thread
Thread with id : 3 .... exiting
Sleeping in thread
Thread with id : 4 .... exiting
Main: completed thread id :0  exiting with status :0
```

```
Main: completed thread id :1  exiting with status :0
Main: completed thread id :2  exiting with status :0
Main: completed thread id :3  exiting with status :0
Main: completed thread id :4  exiting with status :0
Main: program exiting.
```