Thread functions in C/C++

In a **Unix/Linux operating system**, the **C/C++ languages** provide the POSIX thread(pthread) standard API(Application program Interface) for all thread related functions. It allows us to create multiple threads for concurrent process flow. It is most effective on multiprocessor or multi-core systems where threads can be implemented on a kernel-level for achieving the speed of execution. Gains can also be found in uni-processor systems by exploiting the latency in IO or other system functions that may halt a process.
We must include the pthread.h header file at the beginning of the script to use all the functions of the pthreads library. To execute the c file, we have to use the -pthread or -lpthread in the command line while compiling the file.

```
cc -pthread file.c or

cc -lpthread file.c
```

The **functions** defined in the **pthreads library** include:

a. ***pthread_create:*** used to create a new thread
   Syntax:

```
int pthread_create(pthread_t * thread,
                   const pthread_attr_t * attr,
                   void * (*start_routine)(void *),
                   void *arg);
```

   Parameters:

- **thread:** pointer to an unsigned integer value that returns the thread id of the thread created.
- **attr:** pointer to a structure that is used to define thread attributes like detached state, scheduling policy, stack address, etc. Set to NULL for default thread attributes.
- **start_routine:** pointer to a subroutine that is executed by the thread. The return type and parameter type of the subroutine must be of type void *. The function has a single attribute but if multiple values need to be passed to the function, a struct must be used.
- **arg:** pointer to void that contains the arguments to the function defined in the earlier argument

b. ***pthread_exit:*** used to terminate a thread
   Syntax:

```
void pthread_exit(void *retval);
```

   Parameters: This method accepts a mandatory parameter **retval** which is the pointer to an integer that stores the return status of the thread terminated. The

scope of this variable must be global so that any thread waiting to join this thread may read the return status.

c. ***pthread_join:*** used to wait for the termination of a thread.
Syntax:
```
int pthread_join(pthread_t th,
                 void **thread_return);
```
Parameter: This method accepts following parameters:
- **th:** thread id of the thread for which the current thread waits.
- **thread_return:** pointer to the location where the exit status of the thread mentioned in th is stored.

d. ***pthread_self:*** used to get the thread id of the current thread.
Syntax:
```
pthread_t pthread_self(void);
```

e. ***pthread_equal:*** compares whether two threads are the same or not. If the two threads are equal, the function returns a non-zero value otherwise zero.
Syntax:
```
int pthread_equal(pthread_t t1,
                  pthread_t t2);
```
Parameters: This method accepts following parameters:
- t1: the thread id of the first thread
- t2: the thread id of the second thread

f. ***pthread_cancel:*** used to send a cancellation request to a thread
Syntax:
```
int pthread_cancel(pthread_t thread);
```
Parameter: This method accepts a mandatory parameter **thread** which is the thread id of the thread to which cancel request is sent.

g. ***pthread_detach:*** used to detach a thread. A detached thread does not require a thread to join on terminating. The resources of the thread are automatically released after terminating if the thread is detached.
Syntax:
```
int pthread_detach(pthread_t thread);
```
Parameter: This method accepts a mandatory parameter **thread** which is the thread id of the thread that must be detached.

Example: A simple implementation of threads may be as follows:

```
// C program to show thread functions



#include <pthread.h>
```

```
#include <stdio.h>
#include <stdlib.h>

void* func(void* arg)
{
    // detach the current thread
    // from the calling thread
    pthread_detach(pthread_self());
    printf("Inside the thread\n");
    // exit the current thread
    pthread_exit(NULL);
}

void fun()
{
    pthread_t ptid;

    // Creating a new thread
    pthread_create(&ptid, NULL, &func, NULL);
    printf("This line may be printed"
           " before thread terminates\n");
```

```
    // The following line terminates
    // the thread manually
    // pthread_cancel(ptid);

    // Compare the two threads created
    if(pthread_equal(ptid, pthread_self())
        printf("Threads are equal\n");
    else
        printf("Threads are not equal\n");

    // Waiting for the created thread to terminate
    pthread_join(ptid, NULL);
   printf("This line will be printed"
           " after thread ends\n");

    pthread_exit(NULL);
}

// Driver code
int main()
{
    fun();
```

```
    return 0;

}
```

Output:

```
This line may be printed before thread terminates

Threads are not equal

Inside the thread

This line will be printed after thread ends
```

Explanation: Here two threads of execution are created in the code. The order of the lines of output of the two threads may be interchanged depending upon the thread processed earlier. The main thread waits on the newly created thread for exiting. Therefore, the final line of the output is printed only after the new thread exits. The threads can terminate independently of each other by not using the *pthread_join* function. If we want to terminate the new thread manually, we may use *pthread_cancel* to do it.

Note: If we use exit() instead of **pthread_exit()** to end a thread, the whole process with all associated threads will be terminated even if some of the threads may still be running.

PROGRAM1

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
void* func(void* arg)
{
   // detach the current thread
   // from the calling thread
   pthread_detach(pthread_self());

   printf("Inside the thread\n");

   // exit the current thread
   pthread_exit(NULL);
```

```
}

void fun()
{
    pthread_t ptid;

    // Creating a new thread
    pthread_create(&ptid, NULL, &func, NULL);
    printf("This line may be printed"
           " before thread terminates\n");

    // The following line terminates
    // the thread manually
    // pthread_cancel(ptid);

    // Compare the two threads created
    if(pthread_equal(ptid, pthread_self()))
       printf("Threads are equal\n");
    else
       printf("Threads are not equal\n");

    // Waiting for the created thread to terminate
    pthread_join(ptid, NULL);

    printf("This line will be printed"
           " after thread ends\n");

    pthread_exit(NULL);

// Driver code
int main()
```

```
{
   fun();
   return 0;
}

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>  //Header file for sleep(). man 3 sleep for details.
#include <pthread.h>

// A normal C function that is executed as a thread
// when its name is specified in pthread_create()
void *myThreadFun(void *vargp)
{
   sleep(1);
   printf("Printing GeeksQuiz from Thread \n");
   return NULL;
}

int main()
{
   pthread_t thread_id;
   printf("Before Thread\n");
   pthread_create(&thread_id, NULL, myThreadFun, NULL);
   pthread_join(thread_id, NULL);
   printf("After Thread\n");
   exit(0);
}
```

PROGRAM3:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

// Let us create a global variable to change it in threads
int g = 0;

// The function to be executed by all threads
void *myThreadFun(void *vargp)
{
   // Store the value argument passed to this thread
   int *myid = (int *)vargp;

   // Let us create a static variable to observe its changes
   static int s = 0;

   // Change static and global variables
   ++s; ++g;

   // Print the argument, static and global variables
   printf("Thread ID: %d, Static: %d, Global: %d\n", *myid, ++s, ++g);
}
int main()
{
   int i;
   pthread_t tid;

   // Let us create three threads
   for (i = 0; i < 3; i++)
      pthread_create(&tid, NULL, myThreadFun, (void *)&tid);
```

```
   pthread_exit(NULL);
   return 0;
}
```

PROGRAM4

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
#include<stdlib.h>
void *PrintHello(void *threadid)
{
 printf("\n%d: Hello World!\n", threadid);
 pthread_exit(NULL);
}
int main( )
{
 pthread_t threads [NUM_THREADS];
 int rc, t;
 for(t=0; t < NUM_THREADS; t++) {
 printf("Creating thread %d\n", t);
 rc = pthread_create(&threads[t], NULL, PrintHello, (void *) t );
 if (rc)
{
 printf("ERROR; return code from pthread_create() is %d\n", rc);
 exit(-1);
 }
 }
 pthread_exit(NULL);
```

}