**How to Create Threads in Linux (With a C Example Program)**

by Himanshu Arora on April 6, 2012

In the part I of the [Linux Threads](http://www.thegeekstuff.com/2012/03/linux-threads-intro/) series, we discussed various aspects related to threads in Linux.

In this article we will focus on how a thread is created and identified. We will also present a working C program example that will explain how to do basic threaded programming.  
  
Linux Threads Series: [part 1](http://www.thegeekstuff.com/2012/03/linux-threads-intro/), part 2 (this article), [part 3](http://www.thegeekstuff.com/2012/04/terminate-c-thread/).

**Thread Identification**

Just as a process is identified through a process ID, a thread is identified by a thread ID. But interestingly, the similarity between the two ends here.

* A process ID is unique across the system where as a thread ID is unique only in context of a single process.
* A process ID is an integer value but the thread ID is not necessarily an integer value. It could well be a structure
* A process ID can be printed very easily while a thread ID is not easy to print.

The above points give an idea about the difference between a process ID and thread ID.

Thread ID is represented by the type ‘pthread\_t’.  As we already discussed that in most of the cases this type is a structure, so there has to be a function that can compare two thread IDs.

#include <pthread.h>

int pthread\_equal(pthread\_t tid1, pthread\_t tid2);

So as you can see that the above function takes two thread IDs and returns nonzero value if both the thread IDs are equal or else it returns zero.

Another case may arise when a thread would want to know its own thread ID. For this case the following function provides the desired service.

#include <pthread.h>

pthread\_t pthread\_self(void);

So we see that the function ‘pthread\_self()’ is used by a thread for printing its own thread ID.

Now, one would ask about the case where the above two function would be required. Suppose there is a case where a link list contains data for different threads. Every node in the list contains a thread ID and the corresponding data. Now whenever a thread tries to fetch its data from linked list, it first gets its own ID by calling ‘pthread\_self()’ and then it calls the ‘pthread\_equal()’ on every node to see if the node contains data for it or not.

An example of the generic case discussed above would be the one in which a master thread gets the jobs to be processed and then it pushes them into a link list. Now individual worker threads parse the linked list and extract the job assigned to them.

**Thread Creation**

Normally when a program starts up and becomes a process, it starts with a default thread. So we can say that every process has at least one thread of control.  A process can create extra threads using the following function :

#include <pthread.h>

int pthread\_create(pthread\_t \*restrict tidp, const pthread\_attr\_t \*restrict attr, void \*(\*start\_rtn)(void), void \*restrict arg)

The above function requires four arguments, lets first discuss a bit on them :

* The first argument is a pthread\_t type address. Once the function is called successfully, the variable whose address is passed as first argument will hold the thread ID of the newly created thread.
* The second argument may contain certain attributes which we want the new thread to contain.  It could be priority etc.
* The third argument is a function pointer. This is something to keep in mind that each thread starts with a function and that functions address is passed here as the third argument so that the kernel knows which function to start the thread from.
* As the function (whose address is passed in the third argument above) may accept some arguments also so we can pass these arguments in form of a pointer to a void type. Now, why a void type was chosen? This was because if a function accepts more than one argument then this pointer could be a pointer to a structure that may contain these arguments.

**A Practical Thread Example**

Following is the example code where we tried to use all the three functions discussed above.

#include<stdio.h>

#include<string.h>

#include<pthread.h>

#include<stdlib.h>

#include<unistd.h>

pthread\_t tid[2];

void\* doSomeThing(void \*arg)

{

unsigned long i = 0;

pthread\_t id = pthread\_self();

if(pthread\_equal(id,tid[0]))

{

printf("\n First thread processing\n");

}

else

{

printf("\n Second thread processing\n");

}

for(i=0; i<(0xFFFFFFFF);i++);

return NULL;

}

int main(void)

{

int i = 0;

int err;

while(i < 2)

{

err = pthread\_create(&(tid[i]), NULL, &doSomeThing, NULL);

if (err != 0)

printf("\ncan't create thread :[%s]", strerror(err));

else

printf("\n Thread created successfully\n");

i++;

}

sleep(5);

return 0;

}

So what this code does is :

* It uses the pthread\_create() function to create two threads
* The starting function for both the threads is kept same.
* Inside the function ‘doSomeThing()’, the thread uses pthread\_self() and pthread\_equal() functions to identify whether the executing thread is the first one or the second one as created.
* Also, Inside the same function ‘doSomeThing()’ a for loop is run so as to simulate some time consuming work.

Now, when the above code is run, following was the output :

$ ./threads

Thread created successfully

First thread processing

Thread created successfully

Second thread processing

As seen in the output, first thread is created and it starts processing, then the second thread is created and then it starts processing. Well one point to be noted here is that the order of execution of threads is not always fixed. It depends on the OS scheduling algorithm.

Note: The whole explanation in this article is done on Posix threads. As can be comprehended from the type, the pthread\_t type stands for POSIX threads. If an application wants to test whether POSIX threads are supported or not, then the application can use the macro \_POSIX\_THREADS for compile time test. To compile a code containing calls to posix APIs, please use the compile option ‘-pthread’.