Introduction to Linux Threads – Part I

A thread of execution is often regarded as the smallest unit of processing that a scheduler works on.

A process can have multiple threads of execution which are executed asynchronously.

This asynchronous execution brings in the capability of each thread handling a particular work or service independently. Hence multiple threads running in a process handle their services which overall constitutes the complete capability of the process.

Why Threads are Required?

Now, one would ask why do we need multiple threads in a process?? Why can't a process with only one (default) main thread be used in every situation.

Well, to answer this lets consider an example :

Suppose there is a process, that receiving real time inputs and corresponding to each input it has to produce a certain output. Now, if the process is not multi-threaded ie if the process does not involve multiple threads, then the whole processing in the process becomes synchronous. This means that the process takes an input processes it and produces an output.

The limitation in the above design is that the process cannot accept an input until its done processing the earlier one and in case processing an input takes longer than expected then accepting further inputs goes on hold.

To consider the impact of the above limitation, if we map the generic example above with a socket server process that can accept input connection, process them and provide the socket client with output. Now, if in processing any input if the server process takes more than expected time and in the meantime another input (connection request) comes to the socket server then the server process would not be able to accept the new input connection as its already stuck in processing the old input connection. This may lead to a connection time out at the socket client which is not at all desired.

This shows that synchronous model of execution cannot be applied everywhere and hence was the requirement of asynchronous model of execution felt which is implemented by using threads.

Difference Between threads and processes

Following are some of the major differences between the thread and the processes :

- Processes do not share their address space while threads executing under same process share the address space.
- From the above point its clear that processes execute independent of each other and the synchronization between processes is taken care by kernel only while on the other hand the thread synchronization has to be taken care by the process under which the threads are executing
- Context switching between threads is fast as compared to context switching between processes
- The interaction between two processes is achieved only through the standard inter process communication while threads executing under the same process can communicate easily as they share most of the resources like memory, text segment etc

User threads Vs Kernel Threads

Threads can exist in user space as well as in kernel space.

A **user space** threads are created, controlled and destroyed using user space thread libraries. These threads are not known to kernel and hence kernel is nowhere involved in their processing. These threads follow co-operative multitasking where-in a thread releases CPU on its own wish ie the scheduler cannot preempt the thread. Th advantages of user space threads is that the switching between two threads does not involve much overhead and is generally very fast while on the negative side since these threads follow co-operative multitasking so if one thread gets block the whole process gets blocked.
A **kernel space** thread is created, controlled and destroyed by the kernel. For every thread that exists in user space there is a corresponding kernel thread. Since these threads are managed by kernel so they follow preemptive multitasking where-in the scheduler can preempt a thread in execution with a higher priority thread which is ready for execution. The major advantage of kernel threads is that even if one of the thread gets blocked the whole process is not blocked as kernel threads follow preemptive scheduling while on the negative side the context switch is not very fast as compared to user space threads.
If we talk of Linux then kernel threads are optimized to such an extent that they are considered better than user space threads and mostly used in all scenarios except where prime requirement is that of cooperative multitasking.

Problem with Threads

There are some major problems that arise while using threads :

- Many operating system does not implement threads as processes rather they see threads as part of parent process. In this case, what would happen if a thread calls fork() or even worse what if a thread execs a new binary?? These scenarios may have

dangerous consequences for example in the later problem the whole parent process could get replaced with the address space of the newly exec'd binary. This is not at all desired. Linux which is POSIX complaint makes sure that calling a fork() duplicates only the thread that has called the fork() function while an exec from any of the thread would stop all the threads in the parent process.

- Another problem that may arise is the concurrency problems. Since threads share all the segments (except the stack segment) and can be preempted at any stage by the scheduler than any global variable or data structure that can be left in inconsistent state by preemption of one thread could cause severe problems when the next high priority thread executes the same function and uses the same variables or data structures.

For the problem 1 mentioned above, all we can say is that its a design issue and design for applications should be done in a way that least problems of this kind arise.

For the problem 2 mentioned above, using locking mechanisms programmer can lock a chunk of code inside a function so that even if a context switch happens (when the function global variable and data structures were in inconsistent state) then also next thread is not able to execute the same code until the locked code block inside the function is unlocked by the previous thread (or the thread that acquired it).

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

In the part I of the Linux Threads 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.

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’.

How to Terminate a Thread in C Program (pthread_exit Example)

In the part-II (Thread creation and Identification) of the Linux Thread series, we discussed about thread IDs, how to compare two thread IDs and how to create a thread.

In this part, we will mainly focus on how a thread is terminated.

C Thread Example Program

If we take the same example as discussed in part-II of this series :

```
#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;

}
```

Did you observe the ‘sleep()’ function being used? Did you get a question about why sleep() is being used? Well if you did then you are at the correct place to get the answer and if you did not then also its going to be a good read ahead.

If I remove the sleep() function from the code above and then try to compile and run it, I see the following output :

```
$ ./threads

Thread created successfully

First thread processing

Thread created successfully
```

But if I run it with sleep() enabled then I see the output as :

```
$ ./threads

Thread created successfully

First thread processing
```

```
Thread created successfully


Second thread processing
```

So we see that the log ‘Second thread processing’ is missing in case we remove the sleep() function.

So, why does this happen? Well, this happened because just before the second thread is about to be scheduled, the parent thread (from which the two threads were created) completed its execution. This means that the default thread in which the main() function was running got completed and hence the process terminated as main() returned.

Thread Termination

As already discussed above that each program starts with at least one thread which is the thread in which main() function is executed. So maximum lifetime of every thread executing in the program is that of the main thread. So, if we want that the main thread should wait until all the other threads are finished then there is a function pthread_join().

```
#include <pthread.h>


int pthread_join(pthread_t thread, void **rval_ptr);
```

The function above makes sure that its parent thread does not terminate until it is done. This function is called from within the parent thread and the first argument is the thread ID of the thread to wait on and the second argument is the return value of the thread on which we want to the parent thread to wait. If we are not interested in the return value then we can set this pointer to be NULL.

If we classify on a broader level, then we see that a thread can terminate in three ways :

1. If the thread returns from its start routine.
2. If it is canceled by some other thread. The function used here is pthread_cancel().
3. If its calls pthread_exit() function from within itself.

The focus here would be on pthread_exit(). Its prototype is as follows :

```
#include <pthread.h>

void pthread_exit(void *rval_ptr);
```

So we see that this function accepts only one argument, which is the return from the thread that calls this function. This return value is accessed by the parent thread which is waiting for this thread to terminate. The return value of the thread terminated by pthread_exit() function is accessible in the second argument of the pthread_join which just explained above.

C Thread Termination Example

Lets take an example where we use the above discussed functions :

```
#include<stdio.h>

#include<string.h>

#include<pthread.h>

#include<stdlib.h>

#include<unistd.h>
```

```
pthread_t tid[2];

int ret1,ret2;



void* doSomeThing(void *arg)

{

    unsigned long i = 0;

    pthread_t id = pthread_self();



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



    if(pthread_equal(id,tid[0]))

    {

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

        ret1  = 100;

        pthread_exit(&ret1);

    }
```

```
    else

    {

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

        ret2  = 200;

        pthread_exit(&ret2);

    }



    return NULL;

}



int main(void)

{

    int i = 0;

    int err;

    int *ptr[2];
```

```
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++;

}



pthread_join(tid[0], (void**)&(ptr[0]));

pthread_join(tid[1], (void**)&(ptr[1]));



printf("\n return value from first thread is [%d]\n", *ptr[0]);

printf("\n return value from second thread is [%d]\n", *ptr[1]);
```

```
    return 0;

}
```

In the code above :

- We created two threads using pthread_create()
- The start function for both the threads is same ie doSomeThing()
- The threads exit from the start function using the pthread_exit() function with a return value.
- In the main function after the threads are created, the pthread_join() functions are called to wait for the two threads to complete.
- Once both the threads are complete, their return value is accessed by the second argument in the pthread_join() call.

The output of the above code comes out as :

```
$ ./threads

Thread created successfully

Thread created successfully

First thread processing done

Second thread processing done

return value from first thread is [100]

return value from second thread is [200]
```

So we see that both the threads execute completely and their return value is accessed in the main function.

How to Use C Mutex Lock Examples for Linux Thread Synchronization

In the Linux threads series, we discussed on the ways in which a thread can terminate and how the return status is passed on from the terminating thread to its parent thread. In this article we will throw some light on an important aspect known as thread synchronization.

Thread Synchronization Problems

Lets take an example code to study synchronization problems :

```
#include<stdio.h>

#include<string.h>

#include<pthread.h>

#include<stdlib.h>

#include<unistd.h>



pthread_t tid[2];

int counter;



void* doSomeThing(void *arg)

{

    unsigned long i = 0;

    counter += 1;

    printf("\n Job %d started\n", counter);
```

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

    printf("\n Job %d finished\n", counter);



    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));

        i++;

    }



    pthread_join(tid[0], NULL);
```

```
    pthread_join(tid[1], NULL);



    return 0;

}
```

The above code is a simple one in which two threads(jobs) are created and in the start function of these threads, a counter is maintained through which user gets the logs about job number which is started and when it is completed. The code and the flow looks fine but when we see the output :

```
$ ./tgsthreads

Job 1 started

Job 2 started

Job 2 finished

Job 2 finished
```

If you focus on the last two logs, you will see that the log ‘Job 2 finished’ is repeated twice while no log for ‘Job 1 finished’ is seen.

Now, if you go back at the code and try to find any logical flaw, you’ll probably not find any flaw easily. But if you’ll have a closer look and visualize the execution of the code, you’ll find that :

- The log ‘Job 2 started’ is printed just after ‘Job 1 Started’ so it can easily be concluded that while thread 1 was processing the scheduler scheduled the thread 2.
- If the above assumption was true then the value of the ‘counter’ variable got incremented again before job 1 got finished.
- So, when Job 1 actually got finished, then the wrong value of counter produced the log ‘Job 2 finished’ followed by the ‘Job 2 finished’ for the actual job 2 or vice versa as it is dependent on scheduler.
- So we see that its not the repetitive log but the wrong value of the ‘counter’ variable that is the problem.

The actual problem was the usage of the variable ‘counter’ by second thread when the first thread was using or about to use it. In other words we can say that lack of synchronization between the threads while using the shared resource ‘counter’ caused the problems or in one word we can say that this problem happened due to ‘Synchronization problem’ between two threads.

Mutexes

Mutexes

Now since we have understood the base problem, lets discuss the solution to it. The most popular way of achieving thread synchronization is by using Mutexes.

A Mutex is a lock that we set before using a shared resource and release after using it. When the lock is set, no other thread can access the locked region of code. So we see that even if thread 2 is scheduled while thread 1 was not done accessing the shared resource and the code is locked by thread 1 using mutexes then thread 2 cannot even access that region of code. So this ensures a synchronized access of shared resources in the code.

Internally it works as follows :

- Suppose one thread has locked a region of code using mutex and is executing that piece of code.
- Now if scheduler decides to do a context switch, then all the other threads which are ready to execute the same region are unblocked.
- Only one of all the threads would make it to the execution but if this thread tries to execute the same region of code that is already locked then it will again go to sleep.
- Context switch will take place again and again but no thread would be able to execute the locked region of code until the mutex lock over it is released.
- Mutex lock will only be released by the thread who locked it.
- So this ensures that once a thread has locked a piece of code then no other thread can execute the same region until it is unlocked by the thread who locked it.
- Hence, this system ensures synchronization among the threads while working on shared resources.

A mutex is initialized and then a lock is achieved by calling the following two functions :

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t
*restrict attr);

int pthread_mutex_lock(pthread_mutex_t *mutex);
```

The first function initializes a mutex and through second function any critical region in the code can be locked.

The mutex can be unlocked and destroyed by calling following functions :

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

The first function above releases the lock and the second function destroys the lock so that it cannot be used anywhere in future.

A Practical Example

Lets see a piece of code where mutexes are used for thread synchronization

```
#include<stdio.h>

#include<string.h>

#include<pthread.h>

#include<stdlib.h>

#include<unistd.h>
```

```
pthread_t tid[2];

int counter;

pthread_mutex_t lock;



void* doSomeThing(void *arg)

{

    pthread_mutex_lock(&lock);



    unsigned long i = 0;

    counter += 1;

    printf("\n Job %d started\n", counter);



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



    printf("\n Job %d finished\n", counter);
```

```
    pthread_mutex_unlock(&lock);



    return NULL;

}



int main(void)

{

    int i = 0;

    int err;



    if (pthread_mutex_init(&lock, NULL) != 0)

    {

        printf("\n mutex init failed\n");

        return 1;

    }
```

```
    while(i < 2)

    {

        err = pthread_create(&(tid[i]), NULL, &doSomeThing, NULL);

        if (err != 0)

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

        i++;

    }



    pthread_join(tid[0], NULL);

    pthread_join(tid[1], NULL);

    pthread_mutex_destroy(&lock);



    return 0;

}
```

In the code above :

- A mutex is initialized in the beginning of the main function.
- The same mutex is locked in the ‘doSomeThing()’ function while using the shared resource ‘counter’
- At the end of the function ‘doSomeThing()’ the same mutex is unlocked.
- At the end of the main function when both the threads are done, the mutex is destroyed.

Now if we look at the output, we find :

```
$ ./threads

Job 1 started

Job 1 finished

Job 2 started

Job 2 finished
```

So we see that this time the start and finish logs of both the jobs were present. So thread synchronization took place by the use of Mutex.