Threads

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.

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.