Multithreading in C

What is a Thread?

A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called *lightweight processes*.

What are the differences between process and thread?

Threads are not independent of one other like processes as a result threads shares with other threads their code section, data section and OS resources like open files and signals. But, like process, a thread has its own program counter (PC), a register set, and a stack space.

Why Multithreading?

Threads are popular way to improve application through parallelism. For example, in a browser, multiple tabs can be different threads. MS word uses multiple threads, one thread to format the text, other thread to process inputs, etc.

Threads operate faster than processes due to following reasons:

1) Thread creation is much faster.

2) Context switching between threads is much faster.

3) Threads can be terminated easily

4) Communication between threads is faster.

Can we write multithreading programs in C?

Unlike Java, multithreading is not supported by the language standard. POSIX Threads (or Pthreads) is a POSIX standard for threads. Implementation of pthread is available with gcc compiler.

A simple C program to demonstrate use of pthread basic functions

Please not that the below program may compile only with C compilers with pthread library.

```
#include <stdio.h>
#include <stdlib.h>
#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 tid;
   printf("Before Thread\n");
   pthread_create(&tid, NULL, myThreadFun, NULL);
   pthread_join(tid, NULL);
   printf("After Thread\n");
   exit(0);
}
```

In main() we declare a variable called thread_id, which is of type pthread_t, which is an integer used to identify the thread in the system. After declaring thread_id, we call

pthread_create() function to create a thread.
pthread_create() takes 4 arguments.

The first argument is a pointer to thread_id which is set by this function.
The third argument is name of function to be executed for the thread to be created.
The fourth argument is used to pass arguments to thread.

The pthread_join() function for threads is the equivalent of wait() for processes. A call to pthread_join blocks the calling thread until the thread with identifier equal to the first argument terminates.

How to compile above program?

To compile a multithreaded program using gcc, we need to link it with the pthreads library.

Following is the command used to compile the program.

```
gfg@ubuntu:~/$ gcc multithread.c -lpthread
gfg@ubuntu:~/$ ./a.out
Before Thread
Printing GeeksQuiz from Thread
After Thread
gfg@ubuntu:~/$
```

A C program to show multiple threads with global and static variables

As mentioned above, all threads share data segment. Global and static variables are stored in data segment. Therefore, they are shared by all threads. The following example program demonstrates the same.

```
#include <stdio.h>
#include <stdlib.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 *)i);

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

```
gfg@ubuntu:~/$ gcc multithread.c -lpthread
gfg@ubuntu:~/$ ./a.out
Thread ID: 1, Static: 1, Global: 1
Thread ID: 0, Static: 2, Global: 2
Thread ID: 2, Static: 3, Global: 3
```

Please note that above is simple example to show how threads work. Accessing a global variable in a thread is generally a bad idea. What if thread 2 has priority over thread 1 and thread 1 needs to change the variable. In practice, if it is required to access global variable by multiple threads, then they should be accessed using a mutex.

Mutex lock for Linux Thread Synchronization

Thread synchronization is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as critical section. Processes' access to critical section is controlled by using synchronization techniques. When one thread starts executing the critical section (serialized segment of the program) the other thread should wait until the first thread finishes. If proper synchronization techniques are not applied, it may cause a race condition where the values of variables may be unpredictable and vary depending on the timings of context switches of the processes or threads.

Thread Synchronization Problems

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* trythis(void *arg)
{
    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);
    for(i=0; i<(0xFFFFFFFF);i++);
    printf("\n Job %d has finished\n", counter);

     return NULL;
}
```

```
 int main(void)
{
    int i = 0;
    int error;

    while(i < 2)
    {
        error = pthread_create(&(tid[i]), NULL, &trythis, NULL);
        if (error != 0)
            printf("\nThread can't be created : [%s]", strerror(error));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    return 0;
}
```

How to compile above program?

To compile a multithreaded program using gcc, we need to link it with the pthreads library. Following is the command used to compile the program.

```
gfg@ubuntu:~/$ gcc filename.c -lpthread
```

In this example two threads(jobs) are created and in the start function of these threads, a counter is maintained to get the logs about job number which is started and when it is completed.

Output :

```
Job 1 has started
Job 2 has started
Job 2 has finished
Job 2 has finished
```

Problem : From the last two logs, one can see that the log '*Job 2 has finished*' is repeated twice while no log for '*Job 1 has finished*' is seen.

Why it has occurred?

On observing closely and visualizing the execution of the code, we can see find that :

- The log '*Job 2 has started*' is printed just after '*Job 1 has Started*' so it can easily be concluded that while thread 1 was processing the scheduler scheduled the thread 2.
- If we take the above assumption 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 has finished*' followed by the '*Job 2 has finished*' for the actual job 2 or vice versa as it is dependent on scheduler.
- So we see that it's 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.

How to solve 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.

Working of a mutex

5. Suppose one thread has locked a region of code using mutex and is executing that piece of code.
6. Now if scheduler decides to do a context switch, then all the other threads which are ready to execute the same region are unblocked.

7. 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.
8. 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.
9. Mutex lock will only be released by the thread who locked it.
10. 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:

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

1. **int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr) :** Creates a mutex, referenced by mutex, with attributes specified by attr. If attr is NULL, the default mutex attribute (NONRECURSIVE) is used.

   Returned value

   If successful, pthread_mutex_init() returns 0, and the state of the mutex becomes initialized and unlocked.

   If unsuccessful, pthread_mutex_init() returns -1.

2. **int pthread_mutex_lock(pthread_mutex_t *mutex) :** Locks a mutex object, which identifies a mutex. If the mutex is already locked by another thread, the thread waits for the mutex to become available. The thread that has locked a mutex becomes its current owner and remains the owner until the same thread has unlocked it.When the mutex has the attribute of recursive, the use of the lock may be different. When this kind of mutex is locked multiple times by the same thread, then a count is incremented and no waiting thread is posted. The owning thread must call pthread_mutex_unlock() the same number of times to decrement the count to zero.

Returned value

If successful, pthread_mutex_lock() returns 0.

If unsuccessful, pthread_mutex_lock() returns -1.

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

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

3. **int pthread_mutex_unlock(pthread_mutex_t *mutex) :** Releases a mutex object. If one or more threads are waiting to lock the mutex, pthread_mutex_unlock() causes one of those threads to return from pthread_mutex_lock() with the mutex object acquired. If no threads are waiting for the mutex, the mutex unlocks with no current owner.When the mutex has the attribute of recursive the use of the lock may be different. When this kind of mutex is locked multiple times by the same thread, then unlock will decrement the count and no waiting thread is posted to continue running with the lock. If the count is decremented to zero, then the mutex is released and if any thread is waiting it is posted.

   Returned value

   If successful, pthread_mutex_unlock() returns 0.

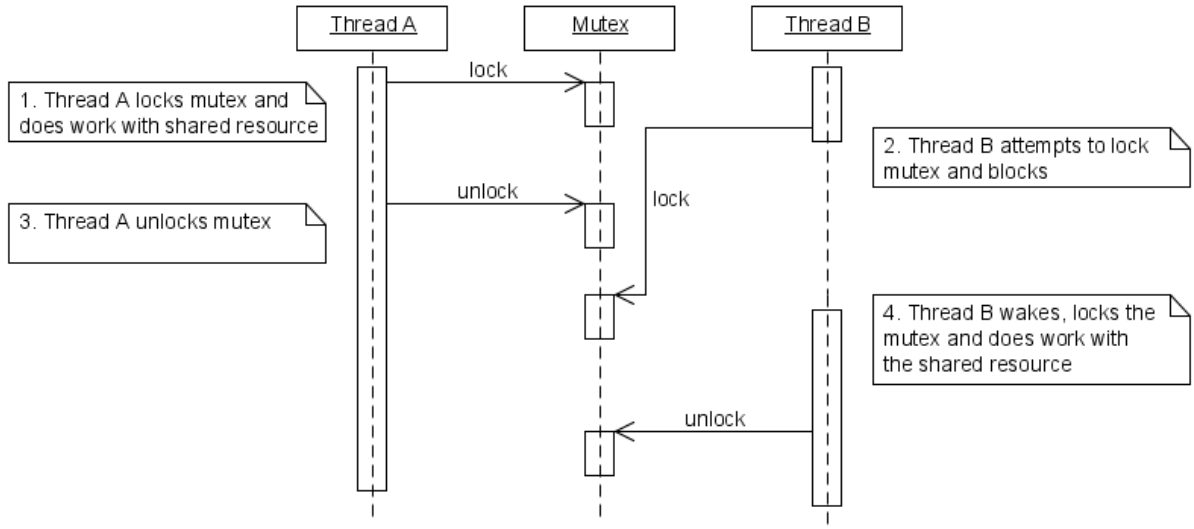
   If unsuccessful, pthread_mutex_unlock() returns -1

4. **int pthread_mutex_destroy(pthread_mutex_t *mutex) :** Deletes a mutex object, which identifies a mutex. Mutexes are used to protect shared resources. mutex is set to an invalid value, but can be reinitialized using pthread_mutex_init().

   Returned value

   If successful, pthread_mutex_destroy() returns 0.

   If unsuccessful, pthread_mutex_destroy() returns -1.



An example to show how 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* trythis(void *arg)
{
    pthread_mutex_lock(&lock);
    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);
    for(i=0; i<(0xFFFFFFFF);i++);
    printf("\n Job %d has finished\n", counter);
    pthread_mutex_unlock(&lock);
    return NULL;
}
```

```
 int main(void)
{
    int i = 0;
    int error;

    if (pthread_mutex_init(&lock, NULL) != 0)
    {
        printf("\n mutex init has failed\n");
        return 1;
    }

    while(i < 2)
    {
        err = pthread_create(&(tid[i]), NULL, &trythis,
NULL);
        if (error != 0)
            printf("\nThread can't be created :[%s]",
strerror(error));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);

    return 0;
}
```

In the above code :

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

Output :

```
Job 1 started
Job 1 finished
Job 2 started
Job 2 finished
```

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