



# System Programming Using POSIX Threads

The previous articles in this series covered files, processes and inter-process communication. Here, the discussion extends to cover the basics of multi-threading using POSIX functions.

Designing a robust multi-threaded application is quite a vast topic so we will touch upon the basics here.

A thread is an operating system entity which does work on a computer, like that of a process. But it uses less overhead, and it gives you an opportunity to use computing resources more efficiently compared to a traditional process. Some common examples of multi-threaded applications are network servers handling client requests, where individual requests can be handled by a separate thread. There are also applications where one thread is doing computations, while another thread is waiting for some I/O (maybe user input or the arrival of data over a network connection) to be completed.

A properly designed multi-threaded application can run very fast on systems that have multiple CPUs, and can be more responsive with higher throughput. So let us start with the basics of creating a thread by using a POSIX call.

Create a thread using the `pthread_create` function. Its prototype is as follows:

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

The first argument is the address of a variable where you store the identifier of the created thread. The second argument can be used to set some of the attributes of the created thread (I am not using this for the example). The starting function

(which the created thread is to execute) is specified by the third argument; the fourth can be used to pass arguments to the function that's to be run in the thread.

Please refer to the code in `lfy_thread_1.c` to understand the use of `pthread_create` and some other calls. You can find the file at [http://www.linuxforu.com/article\\_source\\_code/oct12/system\\_programming.zip](http://www.linuxforu.com/article_source_code/oct12/system_programming.zip).

To compile code using `pthread` calls, use the linker flag `-lpthread` (`gcc filename -o output -lpthread`). The program takes the number of threads to be created by a command-line argument, and creates that many of them. The function `thread_entry` is the entry point for the created threads. Notice that an argument is also passed through the `pthread_create` call.

The purpose of `pthread_join` is to wait for the specific thread, whose identifier is passed as the first argument to this function. It can also fetch the value returned by a specific thread, which can be seen in the code. Note that it is not always necessary to use the `pthread_join` call—I have used it so that readers could learn about it.

Also note the use of `pthread_exit`, which terminates the current thread—but if there are other threads, they can continue. Also, you can return some information to the caller using `pthread_exit`, and that is what I have done in the code. Please note the pointer operations for this.

Do be careful not to use `exit` instead of `pthread_exit`—the first will terminate the entire process immediately, and

all threads will die, even if that is not what you intended. Also, you will notice the type of the fourth argument in the `pthread_create` call, and the return type of the threads entry routine—whose prototype is `void * (*start_routine)(void *)`. These are `void*`, which means you can pass any type of data, and also return any type of data when using these functions.

Also, I highly recommend that you always check the return value of the functions. The return value on success for most `pthread` calls is 0, and a non-zero return value means some error has occurred; in which case, you should take appropriate action.

I would suggest you read the manual page (*man pthreads*) for general information and more details on the functions.

## Mutex and condition variables

A common problem arises when you are accessing (say reading and writing) global data shared between multiple threads. This can cause an incorrect output. This is called race conditions, and must be avoided. A classic way to solve this is shown in the pseudo-code fragment below. You can use a mutex, which is a basic form of synchronisation, as follows:

```
lock_the_mutex(&mymutex)
    critical region
unlock_the_mutex(&mymutex)
```

The critical region is one where you are accessing or updating the shared data.

The POSIX data type for a mutex is `thread_mutex_t`. It can be statically or dynamically allocated. I have used ‘&’ since the function requires that we pass the address of the mutex. In actual code, you need to use functions like `int pthread_mutex_lock(pthread_mutex_t *mutex)` and `int pthread_mutex_unlock(pthread_mutex_t *mutex)`. The mutex here allows you to ensure that at any instant, only one thread will be able to get the lock on the mutex atomically, and hence the other threads get blocked, since the mutex is now locked. Once the first thread releases the mutex, only then can another thread get the lock and proceed.

While you are using mutexes, please be cautious of deadlocks, a thread relocking a mutex (without unlocking), and similar problems. Let us look at a code fragment to illustrate one of the problems mentioned:

```
Thread1 {
    lock_the_mutex (&mutex_a);
    lock_the_mutex (&mutex_b..);
}

Thread2 {
    lock_the_mutex (&mutex_b);
    lock_the_mutex (&mutex_a..);
}
```

Note the reversed order of locking the mutexes. Suppose

both the threads complete the first step at almost the same time, what would happen? A deadlock—no thread will be able to proceed, as each is waiting to acquire a lock that the other holds. To solve this type of problem, there is the standard locking hierarchy—i.e., all programs that need `mutex_a` and `mutex_b` must always lock `mutex_a` first and then `mutex_b`. You can also first try to lock all the mutexes, and if any one of the set fails, release all previously locked mutexes. For this, you need to use a function like `int pthread_mutex_trylock(pthread_mutex_t *mutex)`. (You may download the file [http://www.linuxforu.com/article\\_source\\_code/oct12/system\\_programming.zip](http://www.linuxforu.com/article_source_code/oct12/system_programming.zip). to experiment with the mutex functions.)

Let me quickly introduce you to condition variables, which are also quite useful in certain cases. Now let’s assume a thread needs to perform some action when a condition becomes true, such as, when it is waiting for some data to arrive in a queue, so that it can consume it. This queue is shared among multiple threads. Now you need a solution that is efficient and will reduce waiting or unnecessary polling. There is a mechanism by which one thread can signal another, which is waiting for some condition to be true, through a condition variable. Some simple pseudo-code below will help you understand the benefit:

```
thread1() {
    pthread_mutex_lock(&mutex);
    done++;
    pthread_cond_signal(&done_cond);
    pthread_mutex_unlock(&mutex);
}

thread2() {
    pthread_mutex_lock(&mutex);
    while (done ==0)
        pthread_cond_wait(&done_cond, &mutex);
    Take some action;
    pthread_mutex_unlock(&mutex);
}
```

In the above example, there are two threads, `thread1` and `thread2`. There is a condition variable, which is denoted by `done_cond` associated with a mutex variable. (A condition variable is always used in conjunction with a mutex.) `thread2` needs to wait till the condition becomes true (here, the condition is the change of the variable `done` to non-zero). Now this is shared between the threads. In actual cases, this can be a queue, list or anything. If, suppose, `thread2` is scheduled first, it gets a lock of the mutex, checks the condition `done==0` (the value of `done` is not changed since `thread1` is not scheduled), and then executes `pthread_cond_wait(&done_cond, &mutex)`. This function `pthread_cond_wait` puts `thread2` to sleep, while unlocking the mutex. So you do avoid polling here.

Now, when `thread1` gets a chance to run, it locks the mutex, changes the variable `done`, calls `pthread_cond_signal(&done_cond)` and unlocks the mutex. Since `thread1`

was in conditional wait, it can now wake up automatically (the mutex will be locked) and proceed.

You may wonder why the *done* variable needs to be checked again? This is a recommended practice, since there can be spurious wake-ups too. (You may download the file [http://www.linuxforu.com/article\\_source\\_code/oct12/system\\_programming.zip](http://www.linuxforu.com/article_source_code/oct12/system_programming.zip). to experiment with the condition variables.)

## Reentrancy and thread safety

When writing code, say a function, for a multi-threaded application, the function must be thread-safe. This means that the function should be logically correct, so that you get proper output when it is executed simultaneously by multiple threads. You need to be careful that your code does not lead to a sudden runtime error, or incorrect results due to threading. These types of errors may not get detected while you are compiling your code and are usually due to the incorrect design of the problem statement. Observe the code below:

```
char *strconvert(char *string) {
    static char buffer[MAX_STRING];
    int index;
    for (index = 0; string[index]; index++)
        buffer[index] = tolower(string[index]);
    buffer[index] = '\0';

    return buffer;
}
```

The function does some conversion (here, converting to lowercase) on an input string, and

the resultant string is passed back to the caller. This is fine when you are using this in a single-threaded program. But can you use this in a multi-threaded program, where it can be called simultaneously by multiple threads? No, you cannot, which is quite interesting. The function uses a static array—so it can happen that while one thread is running this function, it gets pre-empted, and another thread gets scheduled. The second thread also calls the function. Now the array buffer is a static one, and hence its state gets changed. So when the earlier thread gets scheduled again, it will access the buffer, which is now changed—and hence yield a wrong output.

If you experiment with this code, you can see that perhaps you are getting correct results (for small strings) since one thread that was running this function did not get pre-empted. In spite of this, you cannot assume that this will work correctly for all cases.

Such functions are called non-reentrant functions, and are unsuitable for multi-threaded programs. This is quite a big problem. To solve it, you can change the interface of the function as follows:

```
char *strconvert_r(char *input, char *output) {
    int index;
```

```
    for (index = 0; in_str[index]; index++)
        output[index] = toupper(input[index]);
    output[index] = '\0';
    return output;
}
```

You will notice that the caller allocates memory here in the variable *output*. Even if multiple threads are calling this function, all the variables that are used are local—so the context can be saved for all calls, and thus no non-deterministic or incorrect result will occur. Also, in production code, you need more checks and appropriate error handling, like checking the size of the output string, if it is big enough to hold the (transformed) input string. Doing all these for a very large program is quite a large task, and we need to have a proper design to make robust multi-threaded applications. You will be surprised to know that most of the C library functions are unsuitable for use in multi-threaded programs. In the standard library, you should check for functions that are reentrant—such as *strtok*, which is commonly used for string tokenising and is a non-reentrant function. But if you use the *strtok\_r* function, it is reentrant. The interfaces of *strtok* and *strtok\_r* differ. In the manual page, I see the warning, “*The strtok() function uses a static buffer while parsing, so it's not thread safe. Use strtok\_r() if this matters to you.*”

## Debugging multi-threaded programs using GDB

Take note of some of the debugger commands that can help you debug multi-threaded programs. I have used GDB release 7.2 on Fedora 15. From the output snippets below, you can see that one breakpoint has been set in the function *thread\_entry*, and then execution has stopped. The *info threads* command shows the current state of the threads (here, three threads), with an identifier associated with each thread. The \* indicates the current thread.

```
Fedora$gcc lfy_thread_2.c -o mythread -Wall -g -lpthread
Fedora$gdb ./mythread
```

```
(gdb) set target-async on
(gdb) set non-stop on
```

The above two GDB settings need to be turned on for debugging, manipulating or examining individual threads under GDB. In *non-stop* mode, when a thread stops, only that thread is stopped. The debugger does not stop other threads as well. The implication of *target-async* on is that instead of waiting for a command to be completed, the debugger gives you the prompt back, and runs some command in the background. If you notice the snippets in the last section, I have used ‘c&’, which means that the current thread can continue executing in the background while GDB can accept subsequent commands.

```
(gdb) b thread_entry
Breakpoint 1 at 0x804858c: file lfy_thread_2.c, line 33.
(gdb) run

(gdb) info threads
  Id   Target Id         Frame
  4     Thread 0xb6feab70 (LWP 2507) "mythread" thread_entry
(arg=0x80487b7)
    at lfy_thread_2.c:33
  3     Thread 0xb77ebb70 (LWP 2506) "mythread" thread_entry
(arg=0x80487a9)
    at lfy_thread_2.c:33
  2     Thread 0xb7fecb70 (LWP 2505) "mythread" thread_entry
(arg=0x80487a0)
    at lfy_thread_2.c:33
* 1     Thread 0xb7fed6c0 (LWP 2502) "mythread" (running)
(gdb)
```

In the above snippet, you see that there are four threads in total. (In my code, there are three calls to `pthread_create`, and one main thread.) GDB associated its own integer numbers 1-4 with the threads. The current thread is marked with a \* and is in the running state. The 'arg' mentioned at the right is the argument given to the thread's starting function.

Now, you can switch your focus to some other threads by using the GDB command `thread threadnumber`.

```
(gdb) thread 2
[Switching to thread 2 (Thread 0xb7fecb70 (LWP 2505))]
#0 thread_entry (arg=0x80487a0) at lfy_thread_2.c:33
33     retstr = strconvert((char*)arg);
(gdb) info threads
  Id   Target Id         Frame
  4     Thread 0xb6feab70 (LWP 2507) "mythread" thread_entry
(arg=0x80487b7)
    at lfy_thread_2.c:33
  3     Thread 0xb77ebb70 (LWP 2506) "mythread" thread_entry
(arg=0x80487a9)
    at lfy_thread_2.c:33
* 2     Thread 0xb7fecb70 (LWP 2505) "mythread" thread_entry
(arg=0x80487a0)
    at lfy_thread_2.c:33
  1     Thread 0xb7fed6c0 (LWP 2502) "mythread" (running)
(gdb)
```

Notice that Thread 2 has become the current thread.

Let's assume you are printing some value in the current thread, and then continuing (c&) the execution of the thread. The current thread exited, and you can now switch focus to other threads, which are possibly waiting at breakpoints.


```
(gdb) printf "%s", arg
ABCDEFFG
(gdb) c&
```

```
Continuing.
Abcdeffg
(gdb) [Thread 0xb7fecb70 (LWP 2505) exited]
```

Now Thread 2 has exited, which can be seen from the `info threads` command below:

```
(gdb) info threads
  Id   Target Id         Frame
  4     Thread 0xb6feab70 (LWP 2507) "mythread" thread_entry
(arg=0x80487b7)
    at lfy_thread_2.c:33
  3     Thread 0xb77ebb70 (LWP 2506) "mythread" thread_entry
(arg=0x80487a9)
    at lfy_thread_2.c:33
  1     Thread 0xb7fed6c0 (LWP 2502) "mythread" (running)
```

```
The current thread <Thread ID 2> has terminated. See 'help thread'.
(gdb)
```

So, in this way, you can walk through a multi-threaded program using GDB. This is quite a good way to learn. I would suggest you download the GDB manual, and read the relevant section too. Also, some of the reference links suggested here are quite good for getting a better understanding of the topic. I hope you grasped the basics of writing and debugging multi-threaded applications. Now you can consider applying multi-threading to some of your existing or future applications. 

## References

- [1] Stevens, W. Richard and Rago, Stephen A. 'Advanced Programming in the Unix Environment,' 2nd Edition, Pearson Education
- [2] Butenhof, David R, 'Programming with POSIX® Threads,' Addison Wesley
- [3] Linux manual pages
- [4] <https://computing.llnl.gov/tutorials/pthreads/>
- [5] <http://red.ht/RLbrCA>
- [6] <http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>

## Acknowledgement

I would like to thank Tanmoy Bandyopadhyay and Vikas Nagpal for their help in reviewing this article.

## By: Swati Mukhopadhyay

The author has more than 12 years of experience in academics and corporate training. Her interests are digital logic, computer architecture, computer networking, data structures, Linux and programming languages like C and C++. You can contact her at [swati.mukerjee@gmail.com](mailto:swati.mukerjee@gmail.com) or <http://swatimukhopadhyay.wordpress.com/>