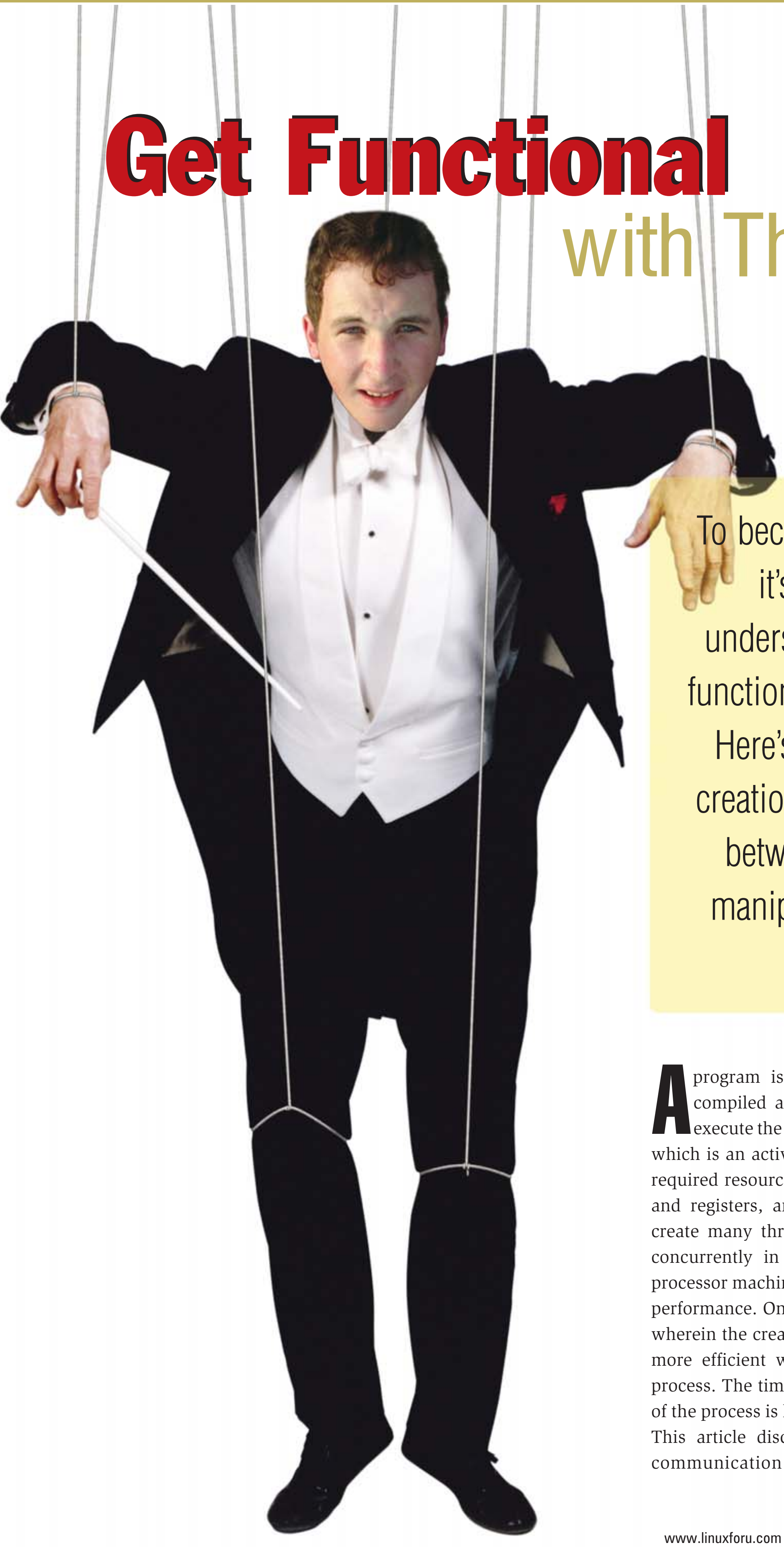
Get Functional with Threads



To become thread-savvy, it's important to understand all about its functions in greater detail. Here's a look at thread creation, communication between threads and manipulation of thread attributes.

A program is a passive entity. When a program is compiled an executable is created. Then when we execute the executable program, it becomes a process, which is an active entity. Whenever a process is created, required resources for the process, like the address space and registers, are allocated to it. If necessary, we can create many threads within a process that will execute concurrently in different processors. Even in a single processor machine, threads usage will improve the system performance. One such situation is the concurrent server, wherein the creation of a thread to service a new client is more efficient when compared to a creation of a new process. The time taken for the creation and maintenance of the process is heavy when compared to that of a thread. This article discusses the process of creating threads, communication between threads using signals, and manipulating thread attributes.

A thread is an encapsulation of the flow of control in a program. When compared to a process, threads behave very similar to a process in which they have their own thread ID, set of registers, stack pointer, stack for local variables, return addresses, signal mask, priority and return value. Threads within a single process share the common process instructions, address space, data, open files (e.g. file descriptors), signal handlers, current working directory, user identification and graphic identification.

There are two broad categories of thread implementation: user level threads and kernel level threads. In the user level threads, the thread management is done by the application; threads are independent of the kernel and are, hence, faster. The thread library manages all activities of the threads. One such library is the POSIX (Portable Operating System Interface for Unix) thread library. In case of kernel level threads, the kernel through system calls does the thread management. Both types have their own merits and demerits.

The main disadvantage of the user level threads is that, if one thread is blocked by the kernel waiting for a certain event to occur, then all the other threads in that process are also blocked because the kernel is unaware about the user level threads. But in case of the kernel level threads, a thread that blocks never causes any problem to other threads in the process, since the kernel can schedule other threads. This article is restricted to user level threads.

Thread creation

This program shows a simple way to create a thread using POSIX library and the process of compiling such a program.

```
#include <pthread.h>

void thread_func(void)
{
printf(" Thread id is %d", pthread_self());
}

main ( )
 {
  pthread_t mythread;
  pthread_create ( &mythread, NULL, (void *) thread_func, NULL);
 }
```

This needs to be compiled as follows...

```
$gcc pthread.c –lpthread
```

This program, when compiled, requires linkage to the POSIX thread library, which can be linked by specifying the pthread library to the gcc compiler.

The *pthread_t* is the data-type to declare a thread variable, here *mythread*. Actually *pthread_t* is type-defined as unsigned long int. *pthread_create* will create a new thread. It takes the thread address as the first argument, the second argument is used to set the attributes for the thread-like stack size, scheduling policy, priority; if NULL is specified, then it takes default values for the attributes. The third argument is the function that the thread should execute when created. The fourth argument is the argument for the thread function. If that function has a single argument to be passed, we can specify it here. If it has more than one argument, then we have to use a structure and declare all the arguments and pass the address of the structure.

Each thread when created will have its own thread ID. On successful execution of *pthread_create* function, the ID is stored into the thread variable, e.g., here it is *mythread*. This can also be got using the *pthread_self* function.

If we create multiple threads and the created threads are dependent, then we have to synchronise the thread activities. Otherwise, the second thread may get executed before the completion of the first thread, which leads to erroneous results. In such a case we can use *pthread_join* to synchronise thread activities that work like a wait system call in process.

Signalling a thread

Signals are notifications sent to a process in order to notify it of various important events. Similarly, threads can also send signals to each other. One such example is listed below...

```
#include <pthread.h>
#include <signal.h>

pthread_t mythread1, mythread2;

void signal_hand (int signo)
{
   printf (" I have received the %d signal\n", signo);
   exit(0);
}

void thread_function1 (void)
{
   printf("The first tid = %d\n", pthread_self());
   signal(SIGINT, signal_hand);
   printf ("Waiting for the signal\n");
   getchar();
}

void thread_function2 (void)
{
  printf("The second tid = %d\n", pthread_self());
  printf (" Sending signal to the %ld thread\n", mythread1);
  pthread_kill(mythread1, SIGINT);
}

int main (void)
{
   pthread_create (&mythread1, NULL, (void *)thread_function1,
NULL);
   pthread_create (&mythread2, NULL, (void *)thread_function2,
NULL);
   pthread_join (mythread1, NULL);
}
```

This program creates two threads: *mythread1* and *mythread2*. Execution of *thread_function2* sends a signal SIGINT to *mythread1* using *pthread_kill. pthread_kill* takes the thread argument and the send signal number. The signal is delivered to the thread specified in the thread argument (in our case *mythread1*). This works similar to *kill.* The *kill* command or system call will not terminate a process; it will actually send a given signal to the specified process. The signal handler specified in *thread_function1* takes care of handling the signal delivered to it.

Thread attributes

While creating a thread, we generally pass NULL as the second argument. By doing so, the thread takes up the default values for the attributes. Now we'll try and set some of these attributes, such as the priority, stack size and so on.

Setting the stack size

The following program helps us to change the stack size of the thread that will be created. Here, we set the stack size to minimum and cause an overflow of the stack. The program works well for small numbers and gives segmentation fault for large numbers, because it tries to access illegal areas of the memory in the recursive function.

```
#include <limits.h>
#include <pthread.h>

float stat = 1;
int n;
float recur (int n)
{
  if ( n == 0 || n == 1)
    return 1;
  else
    {
      stat = stat * n;
      n = n-1;
      recur(n);
    }
  return stat;
}

void thread_routine(int arg)
{
  float res;
  n = arg;
  res = recur(n);
  printf (“\n Factorial of %d is %f\n”, n, res);
}

int main (void)
{
  pthread_t mythread;
  pthread_attr_t thread_attr;
  struct sched_param thread_param;
  size_t stk_size;
  int status;
  int myno;

  printf (“ Enter the number: “);
  scanf (“%d”, &myno);

  pthread_attr_init(&thread_attr);
  pthread_attr_getstacksize(&thread_attr, &stk_size);
  printf (“Default stack size is %u: Minimum size is %u\n”,
stk_size,
         PTHREAD_STACK_MIN);
  pthread_attr_setstacksize(&thread_attr, PTHREAD_STACK_MIN);
  pthread_attr_setguardsize(&thread_attr, 0);

  pthread_create(&mythread, &thread_attr, (void
*)thread_routine, (void* )myno);

}
```

Setting the attributes for a thread is achieved by filling the thread attribute object *thread_attr* of type *pthread_attr_t* and passing it to *pthread_create.*

pthread_attr_init initialises the thread attribute object *thread_attr* and fills it with default values for the attribute. Then each attribute can be set by the user defined value using *pthread_attr_set* <attrname> (attrname=stack size in this particular case). To know the value of the current attribute of a thread we use *pthread_attr_get<attrname>*.

By default, at the overflow end of each thread's stack, the thread's library allocates an overflow warning area, followed by a guard area. These two areas can help a multi-threaded program detect overflow of a thread's stack.

The guard size attribute represents the minimum size (in bytes) of the guard area for the stack of a thread. A guard area can help a multi-threaded program detect overflow of a thread's stack. A guard area is a region of no-access memory that is allocated at the overflow end of the thread's writable stack. When the thread attempts to access a memory location within the guard area, a memory addressing violation occurs, giving a SIGSEGV signal.

To set the guard size attribute of a thread attribute's object, call the *pthread_attr_setguardsize ( )* routine. To obtain the value of the guard size attribute in a thread attribute's object, call the *pthread_attr_getguardsize ( )* routine.

After initialising the attribute object, we pass the object into the thread create function as the second argument.

Setting the priority of a thread

Sometimes, we need the threads to behave in a different manner. The behaviour of the threads can be changed by setting the attributes of the thread. If we need to change the order of the execution of the threads despite the order in which they are created, it can be done by changing the priority of the thread. Sometimes, we even need to change the policy of scheduling required—maybe a round-robin type scheduling, FIFO (first in first out) scheduling or, sometimes, the default policy.

POSIX threads provide APIs to the changes for the thread attributes. They provide a structure *sched_param* that can be instantiated and can be used to assign the priority of the thread to its member *sched_priority*. Then we need to set the scheduling parameter by using *pthread_attr_setschedparam*, to which we pass the attribute object and the *sched_param* variable to set the priority for the thread.

To change the policy of execution of the thread, POSIX provides *pthread_attr_set* policy, which takes the thread attribute object as its first argument and the new policy as its second. The new policy can be directly mentioned as *SCHED_RR* for round-robin, *SCHED_FIFO* for first-in first-out and *SCHED_OTHER* for default.

This article should help you understand the basic concept of user level threads and some of its importance. You will get a clear picture on how to create pthread, get thread ID, communication between threads using signals, and finding and changing the thread attributes with the help of the given programs. LFY

By: B. Thangaraju. The author is tech lead, at Wipro Technologies, Bangalore.
B.B. Ramya and V. Pavithra. The authors are project trainees, Linux Focus Group, Talent Transformation, Wipro Technologies, Bangalore.