Multi-threading Programming

Note Title 9/28/2007

```
/* like any C program, program's execution begins in main */
int
main(int argc, char* argv[])
{
    int        thr_id;         /* thread ID for the newly created thread */
    pthread_t  p_thread;       /* thread's structure                     */
    int        a        = 1;   /* thread 1 identifying number            */
    int        b        = 2;   /* thread 2 identifying number            */

    /* create a new thread that will execute 'do_loop()' */
    thr_id = pthread_create(&p_thread, NULL, do_loop, (void*)&a);
    /* run 'do_loop()' in the main thread as well */
    do_loop((void*)&b);

    /* NOT REACHED */
    return 0;
}
```

pass value or structures

argument to the function

function name

```
/* function to be executed by the new thread */
void*
do_loop(void* data)
{
    int i;                      /* counter, to print numbers */
    int j;                      /* counter, for delay        */
    int me = *((int*)data);     /* thread identifying number */

    for (i=0; i<10; i++) {
        for (j=0; j<500000; j++) /* delay loop */
            ;
        printf("'%d' - Got '%d'\n", me, i);
    }
```

Compile : g++

```
gcc pthread_create.c -o pthread_create -lpthread
```

What Is A Mutex? (Semaphore)

A basic mechanism supplied by the pthreads library to solve this problem, is called a mutex. A mutex is a lock that guarantees three things:

1. Atomicity - Locking a mutex is an atomic operation, meaning that the operating system (or threads library) assures you that if you locked a mutex, no other thread succeeded in locking this mutex at the same time.
2. Singularity - If a thread managed to lock a mutex, it is assured that no other thread will be able to lock the thread until the original thread releases the lock.
3. Non-Busy Wait - If a thread attempts to lock a thread that was locked by a second thread, the first thread will be suspended (and will not consume any CPU resources) until the lock is freed by the second thread. At this time, the first thread will wake up and continue execution, having the mutex locked by it.

From these three points we can see how a mutex can be used to assure exclusive access to variables (or in general critical

Here is some pseudo-code that updates the two variables we were talking about in the previous section, and can be used by the first thread:

```
lock mutex 'X1'.            → disallow others
set first variable to '0'.  }
set second variable to '0'. }
unlock mutex 'X1'.          → allow others
```

0, 1 semaphore is mutex : mutual exclusion.

n semaphore : all threads share totally n number of resources.

Meanwhile, the second thread will do something like this:

```
lock mutex 'X1'.
set first variable to '1'.
set second variable to '1'.
unlock mutex 'X1'.
```

Creating And Initializing A Mutex

```
pthread_mutex_t a_mutex = PTHREAD_MUTEX_INITIALIZER;
```

Lock operation :

```
int rc = pthread_mutex_lock(&a_mutex);
if (rc) { /* an error has occurred */
    perror("pthread_mutex_lock");
    pthread_exit(NULL);
}
/* mutex is now locked - do your stuff. */
```

Unlock operation :

```
rc = pthread_mutex_unlock
(&a_mutex);
if (rc) {
    perror("pthread_mutex_unlock");
    pthread_exit(NULL);
}
```

Destroy :

```
rc = pthread_mutex_destroy(&a_mutex);
```

To avoid "Deadlock".