Threads and parallelism

In shared memory multiprocessor architectures, such as SMPs, threads can be used to implement parallelism. Historically, hardware vendors have implemented their own proprietary versions of threads, making portability a concern for software developers.

Recently, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard. Implementations that adhere to this standard are referred to as POSIX threads, or Pthreads.

Pthreads come in a package called **pthreads** for UNIX–based systems. Win32 versions exist but seem incomplete ((click here for details).

When you use pthreads you will need to specify that threads are used by giving a compiler flage, such as:

- gcc -pthread code.c
- gcc code.c -lthreads
- or similar.

Some references:

- Tutorial from LLNatLab: all the details, lots of colours and code.
- YoLinux tutorial: lots of informative code.
- Another tutorial
- LUPG tutorial, long

These references contain working code written in **C**.

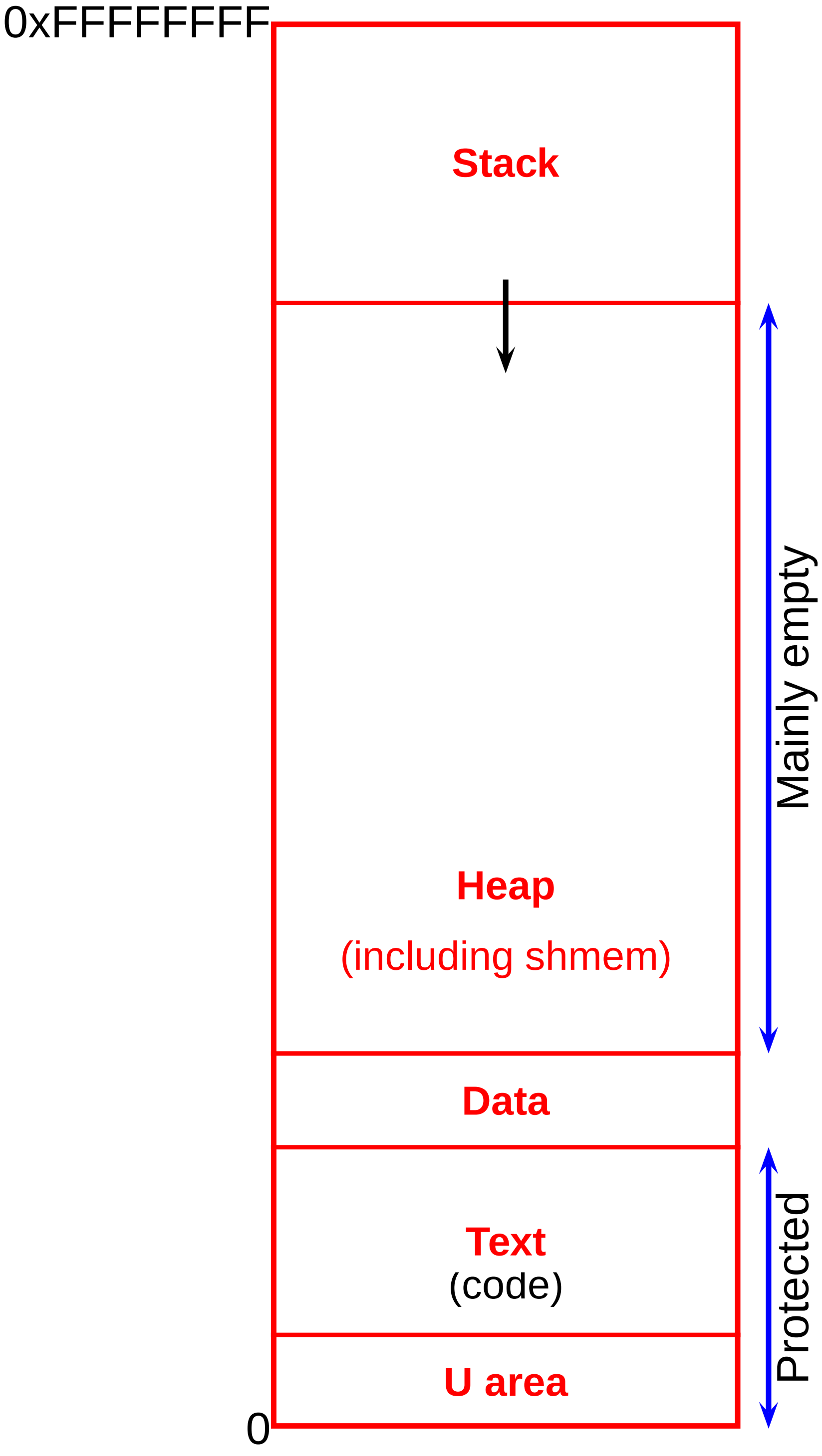
What is a thread

A **thread** is a portion of a process, a semi–process (another term is **lightweight process**) that has its own stack, and executes a given piece of code. Unlike a real process, the thread shares its global variables with other threads (where as for processes we usually have a different memory area for each one of them).

A **Thread Group is a set of threads** all executing inside the same process. They all share the same memory, and thus can access the same global variables, same heap memory (malloc()), same set of file descriptors, etc.

All these threads execute concurrently (using time slices) or in parallel, if the system has several **CPU**s.

The pthreads **API** combines shared memory and semaphores into one set of functions. Sadly, the standard uses different names for these functions than the ones used in **POSIX XSI** shared memory and semaphore standard.


0xFFFFFFFF
Stack
Mainly empty
Heap
(including shmem)
Data
Text
(code)
U area
Protected
0

Example

Suppose we have the urge to execute the function do_loop twice.

```
void do_loop(int me)
{
int i;        /* counter, to print numbers */
int j;        /* counter, for delay        */

   for (i=0; i<10; i++) {
      for (j=0; j<50000 ; j++) /* delay loop */
         ;
      printf("'%d' - Got '%d'\n", me, i);
   }
   exit(0);
}
```

I can fork a process

```
// parent process starts execution in main

int main(int argc, char∗ argv[])
{
pid_t child;        // pid of the newly created child

   if( (pid = fork()) == 0 ) {
      do_loop( 1 ) ;
   else
      do_loop( 2 );

   printf( "If you see this message, say
huh?\n" ) ;
}
```

Same using threads

The code changes because the pthread_create system call requires specific argument types.

```
// execution begins in main (single thread starts)

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

   thr_id = pthread_create(&p_thread, NULL, do_loop,
(void*)&a);
   do_loop((void*)&b);

   printf("If you see this message, say
huh?\n");
}
```

Must be compiled using **cc -pthread** or **gcc -pthread**.

pthread_create() has 4 arguments:

- The first is used by pthread_create() to return to the program information about the thread.
- The second is used to set some attributes for the new thread. In our case we supplied a NULL pointer to tell pthread_create() to use the default values.
- The third is the name of the function that the thread will start executing. It must return a void *.
- The fourth is an argument (or argument list) to pass to the function. It must be of type void *.

The function must be rewritten to match pthread_create:

```
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<50000 ; j++) /* delay loop */
         ;
      printf("'%d' - Got '%d'\n", me, i);
   }
   pthread_exit( NULL ) ;
}
```

pthread_exit() terminates the thread (note that the main process is a thread, too, so it also terminates with a pthread_exit).

Memory sharing

All the **pthreads** forming one group share all their global memory. That includes the memory placed on the **heap** (i.e. acquired using malloc).

Whenever a thread calls a function, the local variables of that function land on the private stack of the calling thread. They are not accessible by other threads. In the do_loop function there will be two sets of private variables i, j, me; one set for each thread. These variables will be different and hence will have different values in each thread.

Semaphores in threads

pthreads have semaphores which are called **mutex**es.

A mutex is declared like any other **global** variable:

```
pthread_mutex mutex = PTHREAD_MUTEX_INITIALIZER ;
```

This gives a properly initialised semaphore.

The operations on these semaphores are very simple:

P(mutex): `pthread_mutex_lock( &mutex ) ;`

V(mutex): `pthread_mutex_unlock( &mutex ) ;`

There is much more

pthreads support many other features:

- Waiting for events to happen or conditions to become true.
- Joining threads (like a return).
- Thread cancellation.

```
void Directory( char *dir )
{
    // declarations here
    struct stat filestat ;

    sprintf( buf , "ls %s > %s/.tmp%d" , dir , dir ,
getpid() ) ;
    system( buf ) ;
    fp = fopen( tmp , "r" ) ;
    while( (ret = fscanf( fp , "%s" , file )) > 0 ) {
        sprintf( buf , "%s/%s" , dir , file ) ;
        strcpy( file , buf ) ;
        if( (ret = lstat( file , &filestat )) < 0 ) {
            perror( "stat" ) ;
            kill( getpid() , SIGKILL ) ;
        } else
            handlefile( file , filestat ) ;
    }
    fclose( fp ) ; unlink( tmp ) ;
}
```

```
void handlefile( char *file , struct stat filestat )
{
    if( S_ISDIR( filestat.st_mode ) ) {
        printf( "directory\n" ) ;
        Directory( file ) ;
    }
    if( S_ISREG( filestat.st_mode ) ) {
        printf( "regular\n" ) ;
        if( filestat.st_mtime > backup→lastinc ) {
            printf( "=======> Backup needed\n" ) ;
        }
    }
    if( S_ISLNK( filestat.st_mode ) )
        printf( "link\n" ) ;
}
```

```
void handlefile( char *file , struct stat filestat )
{
    if( S_ISDIR( filestat.st_mode ) ) {
        printf( "directory\n" ) ;
        pthread_create(&p_thread , NULL ,
            Directory, (void *)&file ) ;
    }
    if( S_ISREG( filestat.st_mode ) ) {
        printf( "regular\n" ) ;
        if( filestat.st_mtime > backup→lastinc ) {
            printf( "=======> Backup needed\n" ) ;
        }
    }
    if( S_ISLNK( filestat.st_mode ) )
        printf( "link\n" ) ;
}
```