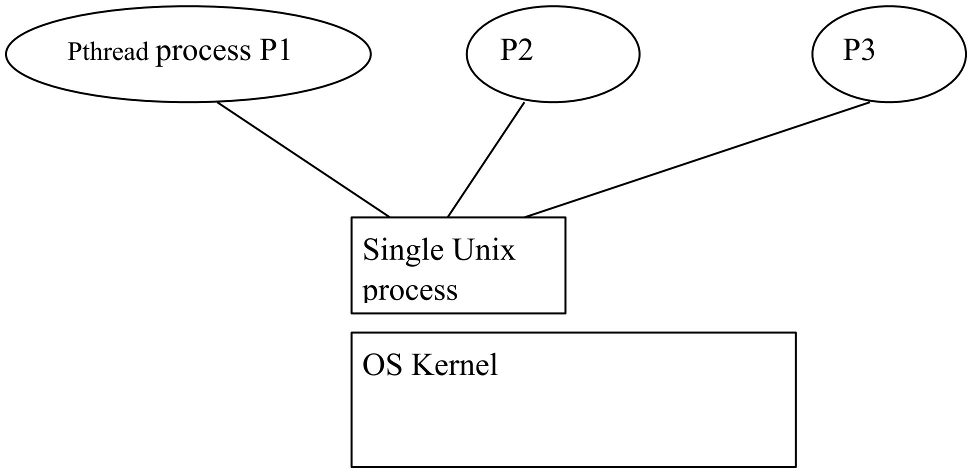
CMPT 300: Pthreads

Pthreads

Pthreads is thread management system implemented at the kernel level. Pictorially it looks like this:



Generally, there are two types of thread systems, user-level threads or kernel-level threads (which we are going to use):

User-level threads	**Kernel-level threads**
- kernel does not know about the threads	- kernel knows about each thread
- if one thread blocks for I/O, the process blocks and all threads block	- if one thread blocks other threads can run
- low cost of creation because we don't have to ask the kernel to create the threads	- high cost of creation because we must ask the kernel to create threads (must use system calls that go through protection boundaries)
- e.g. the GNU PTH system	- e.g. Windows and Linux have kernel level threads

Process Creation

Consider the following code to create two pthread processes:

```
void *print_message_function( void *ptr )
{
     char *message;
     message = (char *) ptr;
     printf("%s \n", message);
}
```

```
main()
{
     pthread_t thread1, thread2;
     char *message1 = "Thread 1";
     char *message2 = "Thread 2";
     int  iret1, iret2;

    /* Create independent threads each of which will execute function */

     iret1 = pthread_create( &thread1, NULL, print_message_function, (void*)
message1);
     iret2 = pthread_create( &thread2, NULL, print_message_function, (void*)
message2);

     /* Wait till threads are complete before main continues. Unless we  */
     /* wait we run the risk of executing an exit which will terminate   */
     /* the process and all threads before the threads have completed.   */

     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);

     printf("Thread 1 returns: %d\n",iret1);
     printf("Thread 2 returns: %d\n",iret2);
     exit(0);
}
```

The pthread_create call returns 0 on success, non-zero error code otherwise. The arguments to pthread_create are as follows:

- argument 1 is a pointer to a location that pthread_create will store the ID of the created thread
- argument 2 specifies the attributes to be applied to the new thread, passing NULL means use the default set of attributes (this is what you would usually want)
- argument 3 is the pointer to the function that will act as the entry point for the thread
- argument 4 is the argument to the process, it must be cast to void *

How do I use pthreads?

What do we need to add to the makefile to use pthreads? Something like the following:

```
gcc -c app.c
gcc -o run -pthread app.o
```

In other words you need to link to the pthread library.

What about include files? You must include the following in your code:

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

CMPT 300: Pthreads

Synchronization using pthreads

Remember that now that we have all these threads running concurrently, we can come across synchronization problems. Pthreads provides the following synchronization constructs:

`pthread_mutex_init(mutex, attr)` – creates a mutex

`pthread_mutex_destroy(mutex)` – destroys a mutex

`pthread_mutex_lock(mutex)` – acquire a lock on the mutex

`pthread_mutex_unlock(mutex)` – unlock the mutex

`pthread_cond_init(condition, attr)` – creates a condition variable

`pthread_cond_destroy(condition)` – destroys a condition variable

`pthread_cond_wait(condition, mutex)` – blocks the calling thread until the specified condition is signalled. This routine should be called while mutex is locked, and it will automatically release the mutex while it waits. After signal is received and thread is awakened, mutex will be automatically locked for use by the thread.

`pthread_cond_signal(condition)` – used to signal another thread which is waiting on the condition variable. It should be called after mutex is locked, and must unlock mutex in order for blocked thread to complete.

See the man pages for the above functions for more details on how they work and what the parameters are. Also see the link provided in the assignment #2 description for more details and examples.

Interprocess Communication using pthreads

Since all of the threads created in one process share memory, the simplest form of IPC between threads is to share variables/data structures. Of course, if you have different threads accessing the same data concurrency issues may arise, hence you need to use mutexes/condition variables to mediate access.