CS350 Handout - Multithreaded Programming Basics (POSIX)

PART I - Programming with Threads

=================================

1. Create a Thread

`pthread_create()`

Thread creation adds a new thread of control to the current Process. The procedure `main()`, itself, is a single thread of control. Each thread executes simultaneously with all the other threads within the calling process, and with other threads from other active processes.

A newly created thread shares all of the calling process' global data with the other threads in this process; however, it has its own set of attributes and private execution stack. The new thread inherits the calling thread's signal mask, possibly, and scheduling priority. Pending signals for a new thread are not inherited and will be empty.

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

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

thread - address where the Thread Identification of the newly created thread is placed. If an error occurs, the value at this address is undefined.

attr - address where the attributes the newly created thread should inherit. If this argment is NULL, then the default attributes are inherited.

`start_routine` - contains the function with which the new thread begins execution. If *`start_routine`* returns, the thread exits with the exit status set to the value returned by *`start_routine`* (see `pthread_exit()`).

`arg` - the argument passed to *`start_routine`*. Only one argument can be passed. If more than one argument needs to be passed to *start_routine*, the arguments can be packed into a structure and the address of that structure can be passed to *arg*.

2. Get the Thread Identifier

`pthread_self()` - Gets the ID of the calling thread.

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

pthread_t pthread_self( void );
----------------------------
```

3. Yield Thread Execution

`pthread_yield()` - Causes the current thread to yield its execution in favor of another thread with the same or greater priority.

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

void pthread_yield( void );
-----------------------------
```

5. Send a Signal to a Thread

`pthread_kill()` - sends a signal to a thread.

```
--------------------------------------------------
#include <sys/types.h>
#include <signal.h>

int kill( pid_t target, int sig );
int tkill( pid_t tid, int sig );
--------------------------------------------------
```

`kill()` sends the signal *`sig`* to the thread specified by *`target`*. *`target`* must be a process. The *`sig`* argument must be from the list given in signal.

`tkill()` sends the signal *`sig`* to the thread specified by *`tid`*. *`tid`* must be a thread within the same process as the calling thread. The *`sig`* argument must be from the list given in signal. `tkill()` is **Linux specific**. Do not use `tkill()` if you plan on making your program portable.

6. Terminate a Thread

`pthread_exit()` - terminates a thread.

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

void pthread_exit( void *status );
-------------------------------
```

The `pthread_exit()` function terminates the calling thread. All thread-specific data bindings are released. If the calling thread is not detached, the thread's ID and the exit status specified by *`status`* are retained until the thread is waited for. Otherwise, *`status`* is ignored and the thread's ID can be reclaimed immediately.

7. Wait for Thread Termination

`pthread_join()` - waits for a thread to terminate.

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

int pthread_join( pthread_t wait_for, void **status );
-------------------------------------------------------------------
```

The `pthread_join()` function blocks the calling thread until the thread specified by *`wait_for`* terminates. The specified thread must be in the current process and must not be detached.
When *`status`* is not NULL, it points to a location that is set to the exit status of the terminated thread when `pthread_join()` returns successfully.

*** *Even though **pthread_join()** can be used to wait for a specific thread; there is no way to wait for "any" thread by using the Pthreads library in Linux.*

8. Related Functions

`thr_sigsetmask()` – accesses the signal mask of the calling thread
`pthread_key_create(), thr_setspecific(), thr_getspecific()` - maintain thread-specific data
`thr_getconcurrency(), thr_setconcurrency()` - get and set thread concurrency level
`thr_getprio(), thr_setprio()` - get and set thread priority

-- See manual pages for details.

PART II - Programming with Synchronization Objects

===

2.1 Mutual Exclusion Locks

1. Lock a Mutex

`pthread_mutex_lock()`

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

int pthead_mutex_lock( pthread_mutex_t *mp );
---------------------------------
```

Use `pthead_mutex_lock()` to lock the mutex pointed by *mp*. When the mutex is already locked, the calling thread blocks until the mutex becomes available (blocked threads wait on a prioritized queue). When `pthread_mutex_lock()` returns, the mutex is locked and the calling thread is the owner.

2. Lock with a Nonblocking Mutex

pthread_mutex_trylock()

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

int pthread_mutex_trylock( pthread_mutex_t *mp );
---------------------------------
```

Use `pthread_mutex_trylock()` to attempt to lock the mutex pointed to by *mp*. This function is a nonblocking version of `pthread_mutex_lock()`. When the mutex is already locked, this call returns with an error number (a positive integer). Otherwise, the mutex is locked (the calling thread becomes the owner of the lock) and a ‘0’ (zero value) is returned.

3. Unlock a Mutex

pthread_mutex_unlock()

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

int pthread_mutex_unlock( pthread_mutex_t *mp );
---------------------------------
```

Use the `pthread_mutex_unlock()` to unlock the mutex pointed to by *mp*. The mutex must be locked and the calling thread must be the one that last locked the mutex (the owner). When any other threads are waiting for the mutex to become available, the thread at the head of the queue is unblocked.

4. Code Example

```
-------------------------------------
pthread_mutex_t count_mutex;
int count;

increment_count()
{
   pthread_mutex_lock( &count_mutex );
   count = count + 1;
   pthread_mutex_unlock( &count_mutex );
}

int get_count()
{
   int c;

   pthread_mutex_lock( &count_mutex );
   c = count;
   pthread_mutex_unlock( &count_mutex );
   return c;
}
---------------------------------------
```

In this example, `increment_count()` uses the mutex lock simply to assure an atomic update of the shared variable. `get_count()` uses the mutex lock to guarantee the synchronization when it refers to `count`.

5. Related Functions

`pthead_mutex_init()` - initializes a mutual exclusion lock
`pthread_mutex_destroy()` - destroy mutex state

see manual pages for details

2.2 Condition Variables

Use condition variables to atomically block threads until a particular condition is true. Always use condition variables together with a mutex lock.

1. Block on a Condition Variable

`pthread_cond_wait()`

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

int pthread_cond_wait( pthread_cond_t *cvp, pthread_mutex_t *mp );
-------------------------------------------
```

Use `pthread_cond_wait()` to atomically release the mutex pointed by *`mp`* and to cause the calling thread to block on the condition variable pointed to by *`cvp`*. The blocked thread can be awakened by `pthread_cond_signal()`, `pthread_cond_broadcast()`, or when interrupted by delivery of a signal.

In typical use, a condition expression is evaluated under the protection of a mutex lock. When the condition expression is false, the thread blocks on the condition variable. The condition variable is then signaled by another thread when it changes the condition value. This causes one or all of the threads waiting on the condition to unblock and to try to reacquire the mutex lock.

Because the condition can change before an awakened thread returns from `pthread_cond_wait()`, the condition that caused the wait must be retested before the mutex lock is acquired. The recommended test method is to write the condition check as a while loop that calls `pthread_cond_wait()`.

```
---------------------------------
pthread_mutex_lock();
   while( condition_is_false )
      pthread_cond_wait();
pthread_mutex_unlock();
---------------------------------
```

2. Unblock a Specific Thread

`pthread_cond_signal()`

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

int pthread_cond_signal( pthread_cond_t *cvp );
-------------------------------
```

Use `pthead_cond_signal()` to unblock one thread that is blocked on the condition variable pointed to by *`cvp`*. Call `pthread_cond_signal()` under the protection of the same mutex used with the condition variable being signaled. Otherwise, the condition variable could be signaled between the test of the associated condition and blocking in `pthread_cond_wait()`, which can cause an infinite wait.

When no threads are blocked on the condition variable, the `pthread_cond_signal()` has no effect.

3. Code Example

```
-------------------------------------------------
pthread_mutex_t count_mutex;
pthread_cond_t count_nonzero;
unsigned int count;

decrement_count()
{
   pthread_mutex_lock( &count_lock );
   while( count == 0 )
      pthread_cond_wait( &count_nonzero, &count_lock );
   count = count - 1;
   pthread_mutex_unlock( &count_lock );
}

increment_count()
{
   pthread_mutex_lock( &count_lock );
   if( count == 0 )
      pthread_cond_signal( &count_nonzero );
   count = count + 1;
   pthread_mutex_unlock( &count_lock );
}
-------------------------------------------------
```

4. Related Functions

`pthread_cond_init()` - initializes a condition variable
`pthread_cond_timedwait()` - blocks until a specified event
`pthread_cond_broadcast()` - unblocks all threads

`pthread_cond_destroy()` - destroys condition variable state

--See manual pages for details

References

1. POSIX online manual pages