**Carlos Alejandro Montiel Lorenzana  
15000552  
Lab08 Synchronization**

Podemos definir que un thread tiene el mismo contexto en los dos lenguajes, del lado del usuario el contexto se basa en que se requiere correr una porción de código en paralelo con otro código, incluso de esta instancia puede haber múltiples.  
Si comparamos la creación de threads de java y C podemos decir lo siguiente:

* C contiene más opciones de sincronización referentes a threads como por ejemplo Mutex, read-write locks, record locking y posix semaphores.
* La máquina virtual de Java tiene varios niveles de abstracción, las capas de un sistema operativo relativamente de un usuario, si se usa un lenguaje de programación bajo contra uno alto esto significa que hay mas capas en medio, por lo tanto si se habla de eficiencia un thread con un lenguaje de programación de nivel bajo es mucho más rápido que un thread definido en un lenguaje de programación más alto.

De una manera, siempre se encuentra un problema en los recursos compartidos, por ejemplo los thread tienen acceso al heap, de esta forma en un context switching en un tiempo con una probabilidad baja, un thread puede estar leyendo ese dato mientras otro lo acaba de escribir. Para poder eliminar ese problema se usa un bloqueo y desbloqueo.  
En C, la facilidad entra en inicializar a Mutex, donde Mutex hace un lock completo, se ejecuta el código y después de hace un unlock para que la operación o la porción de código sea atómica.

Para comenzar, podemos ver un código que explica como funciona las variables de condición, esto lo que hace es esperar una signal para poder terminar el main thread sin necesidad de hacer pthread\_join, esto se logra con variables de condiciones, también se pueden usar para hacer ciclos entre threads

|  |  |
| --- | --- |
|  | #include <pthread.h> |
|  | #include <stdio.h> |
|  | #include <unistd.h> |
|  | #include <assert.h> |
|  |  |
|  |  |
|  | /\* Compile like this: |
|  |  |
|  | gcc --std=c99 -lpthread cond.c -o cond |
|  |  |
|  | \*/ |
|  |  |
|  | const size\_t NUMTHREADS = 20; |
|  |  |
|  | /\* a global count of the number of threads finished working. It will |
|  | be protected by mutex and changes to it will be signalled to the |
|  | main thread via cond \*/ |
|  |  |
|  | int done = 0; |
|  | pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER; |
|  | pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER; |
|  |  |
|  | /\* Note: error checking on pthread\_X calls ommitted for clarity - you |
|  | should always check the return values in real code. \*/ |
|  |  |
|  | /\* Note: passing the thread id via a void pointer is cheap and easy, |
|  | \* but the code assumes pointers and long ints are the same size |
|  | \* (probably 64bits), which is a little hacky. \*/ |
|  |  |
|  | void\* ThreadEntry( void\* id ) |
|  | { |
|  | const int myid = (long)id; // force the pointer to be a 64bit integer |
|  |  |
|  | const int workloops = 5; |
|  | for( int i=0; i<workloops; i++ ) |
|  | { |
|  | printf( "[thread %d] working (%d/%d)\n", myid, i, workloops ); |
|  | sleep(1); // simulate doing some costly work |
|  | } |
|  |  |
|  | // we're going to manipulate done and use the cond, so we need the mutex |
|  | pthread\_mutex\_lock( &mutex ); |
|  |  |
|  | // increase the count of threads that have finished their work. |
|  | done++; |
|  | printf( "[thread %d] done is now %d. Signalling cond.\n", myid, done ); |
|  |  |
|  | // wait up the main thread (if it is sleeping) to test the value of done |
|  | pthread\_cond\_signal( &cond ); |
|  | pthread\_mutex\_unlock( & mutex ); |
|  |  |
|  | return NULL; |
|  | } |
|  |  |
|  | int main( int argc, char\*\* argv ) |
|  | { |
|  | puts( "[thread main] starting" ); |
|  |  |
|  | pthread\_t threads[NUMTHREADS]; |
|  |  |
|  | for( int t=0; t<NUMTHREADS; t++ ) |
|  | pthread\_create( &threads[t], NULL, ThreadEntry, (void\*)(long)t ); |
|  |  |
|  | // we're going to test "done" so we need the mutex for safety |
|  | pthread\_mutex\_lock( &mutex ); |
|  |  |
|  | // are the other threads still busy? |
|  | while( done < NUMTHREADS ) |
|  | { |
|  | printf( "[thread main] done is %d which is < %d so waiting on cond\n", |
|  | done, (int)NUMTHREADS ); |
|  |  |
|  | /\* block this thread until another thread signals cond. While |
|  | blocked, the mutex is released, then re-aquired before this |
|  | thread is woken up and the call returns. \*/ |
|  | pthread\_cond\_wait( & cond, & mutex ); |
|  |  |
|  | puts( "[thread main] wake - cond was signalled." ); |
|  |  |
|  | /\* we go around the loop with the lock held \*/ |
|  | } |
|  |  |
|  | printf( "[thread main] done == %d so everyone is done\n", (int)NUMTHREADS ); |
|  |  |
|  | pthread\_mutex\_unlock( & mutex ); |
|  |  |
|  | return 0; |
|  | } |