pThreads

|  |
| --- |
| **Submission:**  **Show what code you changed in the matrix code, and if there were any obvious timing differences in the different schedulers (there probably aren’t).** |

(short for POSIX threads)

This lab is not really a skills lab. You’ll do a tiny bit of programming, but that isn’t the point.

There are basically 3 main parallelisation tools available, OpenMP and MPI which are intended generally for HPC in some context (OpenMP for a single node, MPI for passing between many nodes), and pthreads, which is for creating multithreaded programs on Unix environments. Pthreads operate really on a single computer at a time, and it spawns new threads as part of a single process.

Pthreads faces the same basic problems as MPI and OpenMP but because it’s an operating system concept there are a couple of interesting things we’ll be able to look at with it.

# A Trivial Program

First up though, a basic pthreads program (in C)

Probably run this to make sure it behaves but it doesn’t really do anything of importance.

(compile with gcc -pthreads filename -o outputiflename)

|  |
| --- |
| // from https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html  #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  void\* print\_message\_function(void\* ptr);  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);  }  void\* print\_message\_function(void\* ptr)  {  char\* message;  message = (char\*)ptr;  printf("%s \n", message);  } |

Ok well that was *exciting*.

The interesting part for us today is that a pthread can both set its own priority \*and\* it can request a particular scheduling algorithm.

# (The bit you do something with) Scheduling Parameters

There are both setters and getters but we care about the setter here:

int pthread\_setschedparam(pthread\_t thread, int policy, const struct sched\_param\* param);

Sets the scheduling parameters (the first param is which thread)

So what are the scheduling policy options:

1. **SCHED\_FIFO**
2. **SCHED\_RR: Round-robin scheduling**
3. **SCHED\_DEADLINE: Sporadic task model deadline scheduling**
4. **SCHED\_OTHER: Default Linux time-sharing scheduling**
5. **SCHED\_IDLE**
6. **SCHED\_BATCH**

You can also set the “nice” value – (it matters for SCHED\_BATCH and SCHED\_OTHER) as its’ the priority essentially of the process.

There are some funky details in the modern Linux scheduler too.

When scheduling non-real-time processes (i.e., those scheduled under

the **SCHED\_OTHER**, **SCHED\_BATCH**, and **SCHED\_IDLE** policies), the CFS scheduler employs a technique known as "group scheduling", if the kernel was configured with the **CONFIG\_FAIR\_GROUP\_SCHED** option (which is typical).

Under group scheduling, threads are scheduled in "task groups". Task groups have a hierarchical relationship, rooted under the initial task group on the system, known as the "root task group". Task groups are formed in the following circumstances:

Examples (if you want them):

<https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_72/apis/users_31.htm>

<http://www.yonch.com/tech/82-linux-thread-priority>

<https://computing.llnl.gov/tutorials/pthreads/>

Ok attached to this lab should be some code that will set and report the status of the thread scheduler (Threadprio.c) and a Matrix multiplication program that uses pthreads (pthreadmatrix).

I want you to

Make the parallel matrix solver much larger, and then change the scheduling algorithm it uses (try FIFO and RR)

Note that the reason this is just a ‘see how it works’ lab because there’s really no easy way to prove which scheduling algorithm it is using, or to see the benefits/drawbacks on a single machine or even a small cluster running few jobs.

Some other Pthreads stuff, you should probably have a quick read through but it’s not hugely important.

# Synchronisation

pthreads also has some support for synchronisation, with three specific constructions, Mutexes, joins and conditional variables.

* mutexes - Mutual exclusion lock: Block access to variables by other threads. This enforces exclusive access by a thread to a variable or set of variables.
* joins - Make a thread wait till others are complete (terminated).
* condition variables - data type pthread\_cond\_t

As with other parallel systems, mutexes prevent race conditions at the expense of forcible ordered exclusion.

## Mutex

|  |  |
| --- | --- |
| Without Mutext | With Mutex |

|  |  |
| --- | --- |
| int counter=0;  /\* Function C \*/  void functionC()  {  counter++  } | /\* Note scope of variable and mutex are the same \*/ pthread\_mutex\_t mutex1 = PTHREAD\_MUTEX\_INITIALIZER; int counter=0;  /\* Function C \*/ void functionC() {  pthread\_mutex\_lock( &mutex1 );  counter++  pthread\_mutex\_unlock( &mutex1 ); } |

Complete program for illustration only

|  |
| --- |
| Again, some code you don’t need to do anything with.  #include <stdio.h> #include <stdlib.h> #include <pthread.h>  void \*functionC(); pthread\_mutex\_t mutex1 = PTHREAD\_MUTEX\_INITIALIZER; int counter = 0;  main() {  int rc1, rc2;  pthread\_t thread1, thread2;   /\* Create independent threads each of which will execute functionC \*/   if( (rc1=pthread\_create( &thread1, NULL, &functionC, NULL)) )  {  printf("Thread creation failed: %d\n", rc1);  }   if( (rc2=pthread\_create( &thread2, NULL, &functionC, NULL)) )  {  printf("Thread creation failed: %d\n", rc2);  }   /\* 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);    exit(0); }  void \*functionC() {  [pthread\_mutex\_lock](http://node1.yo-linux.com/cgi-bin/man2html?cgi_command=pthread_mutex_lock)( &mutex1 );  counter++;  printf("Counter value: %d\n",counter);  [pthread\_mutex\_unlock](http://node1.yo-linux.com/cgi-bin/man2html?cgi_command=pthread_mutex_unlock)( &mutex1 ); } |

## Joins:

Joins are for when you want to wait for one thread to finish. E.g. you create many threads and then wait for them all to finish.

Again, some code you don’t need to do anything with.

|  |
| --- |
| Again, some code you don’t need to do anything with.  #include <stdio.h> #include <pthread.h>  #define NTHREADS 10 void \*thread\_function(void \*); pthread\_mutex\_t mutex1 = PTHREAD\_MUTEX\_INITIALIZER; int counter = 0;  main() {  pthread\_t thread\_id[NTHREADS];  int i, j;   for(i=0; i < NTHREADS; i++)  {  [pthread\_create](http://node1.yo-linux.com/cgi-bin/man2html?cgi_command=pthread_create)( &thread\_id[i], NULL, thread\_function, NULL );  }   for(j=0; j < NTHREADS; j++)  {  [pthread\_join](http://node1.yo-linux.com/cgi-bin/man2html?cgi_command=pthread_join)( thread\_id[j], NULL);   }    /\* Now that all threads are complete I can print the final result. \*/  /\* Without the join I could be printing a value before all the threads \*/  /\* have been completed. \*/   printf("Final counter value: %d\n", counter); }  void \*thread\_function(void \*dummyPtr) {  printf("Thread number %ld\n", [pthread\_self](http://node1.yo-linux.com/cgi-bin/man2html?cgi_command=pthread_self)());  pthread\_mutex\_lock( &mutex1 );  counter++;  pthread\_mutex\_unlock( &mutex1 ); } |

## Condition variables:

Condition variables allow a task to suspend and waiting until some condition (pthread\_cond\_t) is true.

Condition variables must be associated with a mutex to avoid race conditions around the variable itself.

There are three things a condition variable needs:

* Creation/Destruction
* Waiting condition
* Wake based on condition

|  |
| --- |
| Again, some code you don’t need to do anything with.  #include <stdio.h> #include <stdlib.h> #include <pthread.h>  pthread\_mutex\_t count\_mutex = PTHREAD\_MUTEX\_INITIALIZER; pthread\_mutex\_t condition\_mutex = PTHREAD\_MUTEX\_INITIALIZER; pthread\_cond\_t condition\_cond = PTHREAD\_COND\_INITIALIZER;  void \*functionCount1(); void \*functionCount2(); int count = 0; #define COUNT\_DONE 10 #define COUNT\_HALT1 3 #define COUNT\_HALT2 6  main() {  pthread\_t thread1, thread2;   pthread\_create( &thread1, NULL, &functionCount1, NULL);  pthread\_create( &thread2, NULL, &functionCount2, NULL);  pthread\_join( thread1, NULL);  pthread\_join( thread2, NULL);   exit(0); }  void \*functionCount1() {  for(;;)  {  [pthread\_mutex\_lock](http://node1.yo-linux.com/cgi-bin/man2html?cgi_command=pthread_mutex_lock)( &condition\_mutex );  while( count >= COUNT\_HALT1 && count <= COUNT\_HALT2 )  {  [pthread\_cond\_wait](http://node1.yo-linux.com/cgi-bin/man2html?cgi_command=pthread_cond_wait)( &condition\_cond, &condition\_mutex );  }  [pthread\_mutex\_unlock](http://node1.yo-linux.com/cgi-bin/man2html?cgi_command=pthread_mutex_unlock)( &condition\_mutex );   pthread\_mutex\_lock( &count\_mutex );  count++;  printf("Counter value functionCount1: %d\n",count);  pthread\_mutex\_unlock( &count\_mutex );   if(count >= COUNT\_DONE) return(NULL);  } }  void \*functionCount2() {  for(;;)  {  pthread\_mutex\_lock( &condition\_mutex );  if( count < COUNT\_HALT1 || count > COUNT\_HALT2 )  {  [pthread\_cond\_signal](http://node1.yo-linux.com/cgi-bin/man2html?cgi_command=pthread_cond_signal)( &condition\_cond );  }  pthread\_mutex\_unlock( &condition\_mutex );   pthread\_mutex\_lock( &count\_mutex );  count++;  printf("Counter value functionCount2: %d\n",count);  pthread\_mutex\_unlock( &count\_mutex );   if(count >= COUNT\_DONE) return(NULL);  }  } |

But that’s not the interesting part for us today.