POSIX Threads Programming

Table of Contents

Pthreads Overview

What is a Thread?

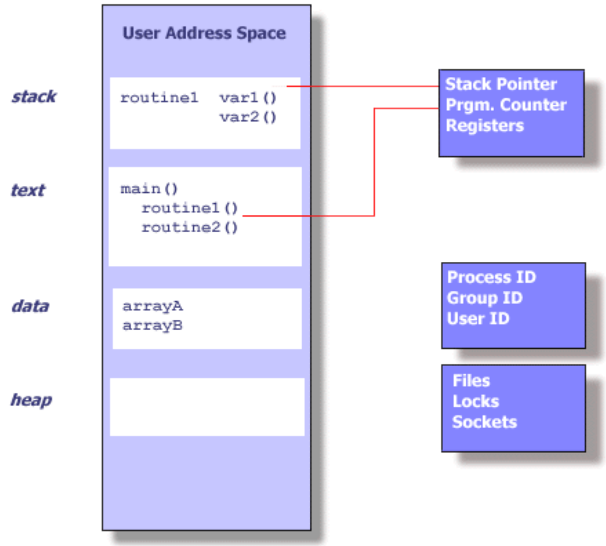
- Technically, a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system. But what does this mean?

- In the UNIX environment a thread:
    - Exists within a process and uses the process resources
    - Has its own independent flow of control as long as its parent process exists and the OS supports it
    - May share the process resources with other threads that act equally independently (and dependently)
    - Dies if the parent process dies - or something similar

- To the software developer, the concept of a "procedure" that runs independently from its main program may best describe a thread.

Process-Thread Relationship:

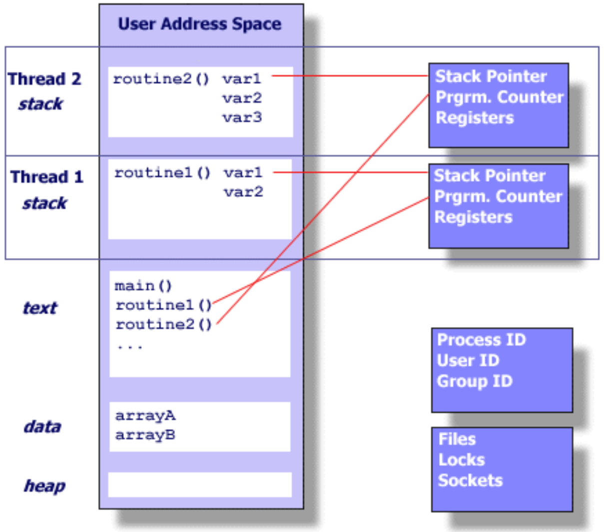
- Understanding a thread means knowing the relationship between a process and a thread. A process is created by the operating system. Processes contain information about program resources and program execution state, including:
    - Process ID, process group ID, user ID, and group ID
    - Environment
    - Working directory.
    - Program instructions
    - Registers
    - Stack
    - Heap
    - File descriptors
    - Signal actions
    - Shared libraries
    - Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).



- Threads use and exist within these process resources, yet are able to be scheduled by the operating system and run as independent entities within a process.

- A thread can possess an independent flow of control and be schedulable because it maintains its own:
    - Stack pointer

- Registers
- Scheduling properties (such as policy or priority)
- Set of pending and blocked signals
- Thread specific data.



- A process can have multiple threads, all of which share the resources within a process and all of which execute within the same address space. Within a multi-threaded program, there are at any time multiple points of execution.

- Because threads within the same process share resources:
  - Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
  - Two pointers having the same value point to the same data.
  - Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.

Pthreads Overview

What are Pthreads?

- Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.

- In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required. For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995). Implementations which adhere to this standard are referred to as POSIX threads, or Pthreads. Most hardware vendors now offer Pthreads in addition to their proprietary API's.

- Pthreads are defined as a set of C language programming types and procedure calls, implemented with a `pthread.h` header/include file and a thread library - though the this library may be part of another library, such as `libc`.

- There are several drafts of the POSIX threads standard. It is important to be aware of the draft number of a given implementation, because there are differences between drafts that can cause problems.

Pthreads Overview

Why Pthreads?

- The primary motivation for using Pthreads is to realize potential program performance gains.

- When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.

  For example, the following table compares timing results for the **`fork()`** subroutine and the **`pthreads_create()`** subroutine. Timings reflect 50,000 process/thread creations, were performed with the `time` utility, and units are in seconds.

Platform	`fork()`			`pthread_create()`		
	real	**user**	**sys**	**real**	**user**	**sys**
IBM 332 MHz 604e 4 CPUs/node 512 MB Memory AIX 4.3	92.4	2.7	105.3	8.7	4.9	3.9
IBM 375 MHz POWER3 16 CPUs/node 16 GB Memory AIX 5.1	173.6	13.9	172.1	9.6	3.8	6.7
INTEL 2.2 GHz Xeon 2 CPU/node 2 GB Memory RedHat Linux 7.3	17.4	3.9	13.5	5.9	0.8	5.3

Source fork_vs_thread.txt

- All threads within a process share the same address space. Inter-thread communication is more efficient and in many cases, easier to use than inter-process communication.

- Threaded applications offer potential performance gains and practical advantages over non-threaded applications in several other ways:

- Overlapping CPU work with I/O: For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, CPU intensive work can be performed by other threads.
- Priority/real-time scheduling: tasks which are more important can be scheduled to supersede or interrupt lower priority tasks.
- Asynchronous event handling: tasks which service events of indeterminate frequency and duration can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests.

- The primary motivation for considering the use of Pthreads on an SMP architecture is to achieve optimum performance. In particular, if an application is using MPI for on-node communications, there is a potential that performance could be greatly improved by using Pthreads for on-node data transfer instead.

- For example: IBM's MPI provides the MP_SHARED_MEMORY environment variable to direct the use of shared memory for on-node MPI communications instead of switch communications. Without this, on-node MPI communications demonstrate serious performance degradation as the number of tasks increase.

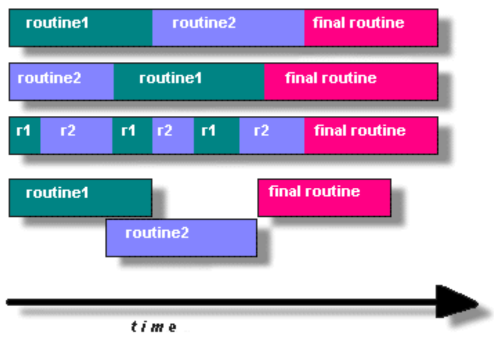
  However, even with MP_SHARED_MEMORY set to "yes", on-node MPI communications can not compete with Pthreads:

  - IBM 604e shared memory MPI bandwidth: @80 MB/sec per MPI task
  - IBM POWER3 NH-2 shared memory MPI bandwidth: @275 MB/sec per MPI task
  - Pthreads worst case: Every data reference by a thread requires a memory read. In this case, a thread's "bandwidth" is limited by the machine's memory-to-CPU bandwidth:
    604e: 1.3 GB/sec
    POWER3 NH-2: 16 GB/sec
  - Pthreads best case: Data is local to a thread's cache offering much greater cache-CPU bandwidths.

Pthreads Overview

Designing Threaded Programs

- In order for a program to take advantage of Pthreads, it must be able to be organized into discrete, independent tasks which can execute concurrently. For example, if routine1 and routine2 can be interchanged, interleaved and/or overlapped in real time, they are candidates for threading.



- Tasks that may be suitable for threading include tasks that:
  - Block for potentially long waits
  - Use many CPU cycles
  - Must respond to asynchronous events
  - Are of lesser or greater importance than other tasks
  - Are able to be performed in parallel with other tasks

- Be careful if your application uses libraries or other objects that don't explicitly guarantee thread-safeness. When in doubt, assume that they are not thread-safe until proven otherwise.

- Several common models for threaded programs exist:

  - ***Manager/worker:*** a single thread, the *manager* assigns work to other threads, the *workers*. Typically, the manager handles all input and parcels out work to the other tasks. At least two forms of the manager/worker model are common: static worker pool and dynamic worker pool.

  - ***Pipeline:*** a task is broken into a series of suboperations, each of which is handled in series, but concurrently, by a different thread. An automobile assembly line best describes this model.

  - ***Peer:*** similar to the manager/worker model, but after the main thread creates other threads, it participates in the work.

The Pthreads API

- The Pthreads API is defined in the ANSI/IEEE POSIX 1003.1 - 1995 standard. Unlike MPI, this standard is not freely available on the Web - it must be purchased from IEEE.

- The subroutines which comprise the Pthreads API can be informally grouped into three major classes:

  1. ***Thread management:*** The first class of functions work directly on threads - creating, detaching, joining, etc. They include functions to set/query thread attributes (joinable, scheduling etc.)

2. ***Mutexes:*** The second class of functions deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. They are also supplemented by mutex attribute functions that set or modify attributes associated with mutexes.

3. ***Condition variables:*** The third class of functions address communications between threads that share a mutex. They are based upon programmer specified conditions. This class includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.

- Naming conventions: All identifiers in the threads library begin with **pthread_**

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys

- The concept of opaque objects pervades the design of the API. The basic calls work to create or modify opaque objects - the opaque objects can be modified by calls to attribute functions, which deal with opaque attributes.

- The Pthreads API contains over 60 subroutines. This tutorial will focus on a subset of these - specifically, those which are most likely to be immediately useful to the beginning Pthreads programmer.

- The `pthread.h` header file must be included in each source file using the Pthreads library. For some implementations, such as IBM's AIX, it may need to be the first include file.

- The current POSIX standard is defined only for the C language. Fortran programmers can use wrappers around C function calls. Also, the IBM Fortran compiler provides a Pthreads API. See the XLF Language Reference, located with IBM's Fortran documentation for more information.

- A number of excellent books about Pthreads are available. Several of these are listed in the References section of this tutorial.

Compiling Threaded Programs

- Some of the more commonly used compile commands for pthreads codes are listed in the table below.

Platform	Compiler Command	Description
IBM AIX	`xlc_r / cc_r`	C (ANSI / non-ANSI)
	`xlC_r`	C++
	`xlf_r -qnosave` `xlf90_r -qnosave`	Fortran - using IBM's Pthreads API (non-portable)
INTEL LINUX	`icc -pthread`	C / C++
COMPAQ Tru64	`cc -pthread`	C
	`cxx -pthread`	C++
All Above Platforms	`gcc -pthread`	GNU C
	`g++ -pthread`	GNU C++
	`guidec -pthread`	KAI C
	`KCC -pthread`	KAI C++

Thread Management

Creating and Terminating Threads

Routines:

pthread_create (thread,attr,start_routine,arg)

pthread_exit (status)

pthread_attr_init (attr)

pthread_attr_destroy (attr)

Creating Threads:

- Initially, your `main()` program comprises a single, default thread. All other threads must be explicitly created by the programmer.

- `pthread_create` creates a new thread and makes it executable. Typically, threads are first created from within `main()` inside a single process. Once created, threads are peers, and may create other threads.

- `pthread_create` arguments:
  - `thread`: An opaque, unique identifier for the new thread returned by the subroutine.
  - `attr`: An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.

- `start_routine`: the C routine that the thread will execute once it is created.
- `arg`: A single argument that may be passed to *start_routine*. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.

- The maximum number of threads that may be created by a process is implementation dependent.

Question: After a thread has been created, how do you know when it will be scheduled to run by the operating system?
Answer

Thread Attributes:

- By default, a thread is created with certain attributes. Some of these attributes can be changed by the programmer via the thread attribute object.

- `pthread_attr_init` and `pthread_attr_destroy` are used to initialize/destroy the thread attribute object.

- Other routines are then used to query/set specific attributes in the thread attribute object.

- Some of these attributes will be discussed later.

Terminating Threads:

- There are several ways in which a Pthread may be terminated:
  - The thread returns from its starting routine (the main routine for the initial thread).
  - The thread makes a call to the `pthread_exit` subroutine (covered below).
  - The thread is canceled by another thread via the `pthread_cancel` routine (not covered here).
  - The entire process is terminated due to a call to either the exec or exit subroutines.

- `pthread_exit` is used to explicitly exit a thread. Typically, the `pthread_exit()` routine is called after a thread has completed its work and is no longer required to exist.

- If `main()` finishes before the threads it has created, and exits with `pthread_exit()`, the other threads will continue to execute. Otherwise, they will be automatically terminated when `main()` finishes.

- The programmer may optionally specify a termination *status*, which is stored as a void pointer for any thread that may join the calling thread.

- Cleanup: the `pthread_exit()` routine does not close files; any files opened inside the thread will remain open after the thread is terminated.

- Recommendation: Use `pthread_exit()` to exit from all threads...especially `main()`.

Example: Pthread Creation and Termination

- This simple example code creates 5 threads with the `pthread_create()` routine. Each thread prints a "Hello World!" message, and then terminates with a call to `pthread_exit()`.

Example Code - Pthread Creation and Termination

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   printf("\n%d: Hello World!\n", threadid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc, t;
   for(t=0;t < NUM_THREADS;t++){
      printf("Creating thread %d\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

Source Output

Thread Management

Passing Arguments to Threads

- The `pthread_create()` routine permits the programmer to pass one argument to the thread start routine. For cases where multiple arguments must be passed, this limitation is easily overcome by creating a structure which contains all of the arguments, and then passing a pointer to that structure in the `pthread_create()` routine.

- All arguments must be passed by reference and cast to (void *).

Question: How can you safely pass data to newly created threads, given their non-deterministic start-up and scheduling?
Answer

Example 1 - Thread Argument Passing

This code fragment demonstrates how to pass a simple integer to each thread. The calling thread uses a unique data structure for each thread, insuring that each thread's argument remains intact throughout the program.

```
int *taskids[NUM_THREADS];
```

```
for(t=0;t < NUM_THREADS;t++)
{
   taskids[t] = (int *) malloc(sizeof(int));
   *taskids[t] = t;
   printf("Creating thread %d\n", t);
   rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) taskids[t]);
   ...
}
```

Source Output

Example 2 - Thread Argument Passing

This example shows how to setup/pass multiple arguments via a structure. Each thread receives a unique instance of the structure.

```
struct thread_data{
   int  thread_id;
   int  sum;
   char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
   struct thread_data *my_data;
   ...
   my_data = (struct thread_data *) threadarg;
   taskid = my_data->thread_id;
   sum = my_data->sum;
   hello_msg = my_data->message;
   ...
}

int main (int argc, char *argv[])
{
   ...
   thread_data_array[t].thread_id = t;
   thread_data_array[t].sum = sum;
   thread_data_array[t].message = messages[t];
   rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) &thread_data_array[t]);
   ...
}
```

Source Output

Example 3 - Thread Argument Passing (Incorrect)

This example performs argument passing incorrectly. The loop which creates threads modifies the contents of the address passed as an argument, possibly before the created threads can access it.

```
int rc, t;

for(t=0;t < NUM_THREADS;t++)
{
   printf("Creating thread %d\n", t);
   rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) &t);
   ...
}
```

Source Output

Thread Management

Joining and Detaching Threads

Routines:

pthread_join (threadid,status)

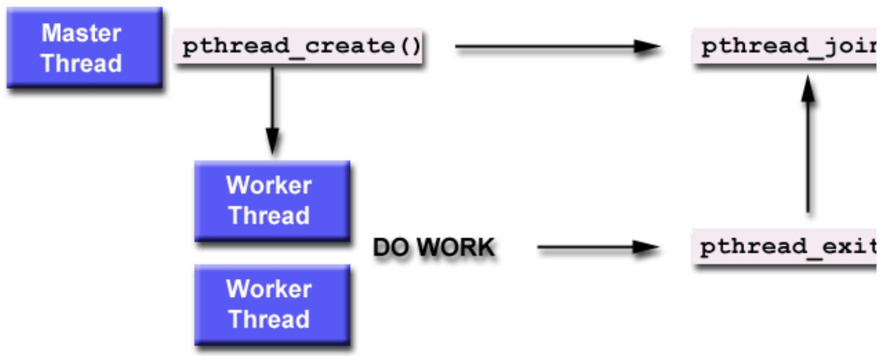
pthread_detach (threadid,status)

pthread_attr_setdetachstate (attr,detachstate)

pthread_attr_getdetachstate (attr,detachstate)

Joining:

- "Joining" is one way to accomplish synchronization between threads. For example:



- The `pthread_join()` subroutine blocks the calling thread until the specified `threadid` thread terminates.
- The programmer is able to obtain the target thread's termination return `status` if it was specified in the target thread's call to `pthread_exit()`.
- Two other synchronization methods, mutexes and condition variables, will be discussed later.

Joinable or Not?

- When a thread is created, one of its attributes defines whether it is joinable or detached. Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.
- The final draft of the POSIX standard specifies that threads should be created as joinable. However, not all implementations may follow this.
- To explicitly create a thread as joinable or detached, the `attr` argument in the `pthread_create()` routine is used. The typical 4 step process is:
  1. Declare a pthread attribute variable of the `pthread_attr_t` data type
  2. Initialize the attribute variable with `pthread_attr_init()`
  3. Set the attribute detached status with `pthread_attr_setdetachstate()`
  4. When done, free library resources used by the attribute with `pthread_attr_destroy()`

Detaching:

- The `pthread_detach()` routine can be used to explicitly detach a thread even though it was created as joinable.
- There is no converse routine.

Recommendations:

- If a thread requires joining, consider explicitly creating it as joinable. This provides portability as not all implementations may create threads as joinable by default.
- If you know in advance that a thread will never need to join with another thread, consider creating it in a detached state. Some system resources may be able to be freed.

Example: Pthread Joining

Example Code - Pthread Joining

This example demonstrates how to "wait" for thread completions by using the Pthread join routine. Since not all current implementations of Pthreads create threads in a joinable state, the threads in this example are explicitly created in a joinable state so that they can be joined later.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     3

void *BusyWork(void *null)
{
   int i;
   double result=0.0;
   for (i=0; i < 1000000; i++)
   {
     result = result + (double)random();
   }
   printf("result = %e\n",result);
   pthread_exit((void *) 0);
}

int main (int argc, char *argv[])
{
   pthread_t thread[NUM_THREADS];
   pthread_attr_t attr;
   int rc, t, status;

   /* Initialize and set thread detached attribute */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   for(t=0;t < NUM_THREADS;t++)
   {
      printf("Creating thread %d\n", t);
      rc = pthread_create(&thread[t], &attr, BusyWork, NULL);
      if (rc)
      {
         printf("ERROR; return code from pthread_create()
                is %d\n", rc);
         exit(-1);
      }
   }

   /* Free attribute and wait for the other threads */
   pthread_attr_destroy(&attr);
   for(t=0;t < NUM_THREADS;t++)
   {
      rc = pthread_join(thread[t], (void **)&status);
      if (rc)
      {
         printf("ERROR; return code from pthread_join()
                is %d\n", rc);
         exit(-1);
      }
      printf("Completed join with thread %d status= %d\n",t, status);
   }

   pthread_exit(NULL);
}
```

Source | Output

Thread Management

Stack Management

Routines:

`pthread_attr_getstacksize (attr, stacksize)`

`pthread_attr_setstacksize (attr, stacksize)`

`pthread_attr_getstackaddr (attr, stackaddr)`

pthread_attr_setstackaddr (attr, stackaddr)

Preventing Stack Problems:

- The POSIX standard does not dictate the size of a thread's stack. This is implementation dependent and varies.
- Exceeding the default stack limit is often very easy to do, with the usual results: program termination and/or corrupted data.
- Safe and portable programs do not depend upon the default stack limit, but instead, explicitly allocate enough stack for each thread by using the `pthread_attr_setstacksize` routine.
- The `pthread_attr_getstackaddr` and `pthread_attr_setstackaddr` routines can be used by applications in an environment where the stack for a thread must be placed in some particular region of memory.

Example: Stack Management

Example Code - Stack Management

This example demonstrates how to query and set a thread's stack size.

```
#include <pthread.h>
#include <stdio.h>
#define NTHREADS 4
#define N 1000
#define MEGEXTRA 1000000

pthread_attr_t attr;

void *dowork(void *threadid)
{
   double A[N][N];
   int i,j;
   size_t mystacksize;

   pthread_attr_getstacksize (&attr, &mystacksize);
   printf("Thread %d: stack size = %d bytes \n", threadid, mystacksize);
   for (i=0; i<N; i++)
     for (j=0; j<N; j++)
      A[i][j] = ((i*j)/3.452) + (N-i);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
   pthread_t threads[NTHREADS];
   size_t stacksize;
   int rc, t;

   pthread_attr_init(&attr);
   pthread_attr_getstacksize (&attr, &stacksize);
   printf("Default stack size = %d\n", stacksize);
   stacksize = sizeof(double)*N*N+MEGEXTRA;
   printf("Amount of stack needed per thread = %d\n",stacksize);
   pthread_attr_setstacksize (&attr, stacksize);
   printf("Creating threads with stack size = %d bytes\n",stacksize);
   for(t=0;t<NTHREADS;t++){
      rc = pthread_create(&threads[t], &attr, dowork, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   printf("Created %d threads.\n", t);
   pthread_exit(NULL);
}
```

Thread Management

Miscellaneous Routines

pthread_self ()

pthread_equal (thread1,thread2)

- `pthread_self` returns the unique, system assigned thread ID of the calling thread.
- `pthread_equal` compares two thread IDs. If the two IDs are different 0 is returned, otherwise a non-zero value is returned.
- Note that for both of these routines, the thread identifier objects are opaque and can not be easily inspected. Because thread IDs are opaque objects, the C language equivalence operator == should not be used to compare two thread IDs against each other, or to compare a single thread ID against another value.

pthread_once (once_control, init_routine)

- `pthread_once` executes the `init_routine` exactly once in a process. The first call to this routine by any thread in the process executes the given `init_routine`, without parameters. Any subsequent call will have no effect.
- The init_routine routine is typically an initialization routine.
- The `once_control` parameter is a synchronization control structure that requires initialization prior to calling `pthread_once`. For example:

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;
```

pthread_yield ()

- `pthread_yield` forces the calling thread to relinquish use of its processor, and to wait in the run queue before it is scheduled again.

Mutex Variables

Overview

- Mutex is an abbreviation for "mutual exclusion". Mutex variables are one of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur.

- A mutex variable acts like a "lock" protecting access to a shared data resource. The basic concept of a mutex as used in Pthreads is that only one thread can lock (or own) a mutex variable at any given time. Thus, even if several threads try to lock a mutex only one thread will be successful. No other thread can own that mutex until the owning thread unlocks that mutex. Threads must "take turns" accessing protected data.

- Mutexes can be used to prevent "race" conditions. An example of a race condition involving a bank transaction is shown below:

Thread 1	Thread 2	Balance
Read balance: $1000		$1000
	Read balance: $1000	$1000
	Deposit $200	$1000
Deposit $200		$1000
Update balance $1000+$200		$1200
	Update balance $1000+$200	$1200

- In the above example, a mutex should be used to lock the "Balance" while a thread is using this shared data resource.

- Very often the action performed by a thread owning a mutex is the updating of global variables. This is a safe way to ensure that when several threads update the same variable, the final value is the same as what it would be if only one thread performed the update. The variables being updated belong to a "critical section".

- A typical sequence in the use of a mutex is as follows:
  - Create and initialize a mutex variable
  - Several threads attempt to lock the mutex
  - Only one succeeds and that thread owns the mutex
  - The owner thread performs some set of actions
  - The owner unlocks the mutex
  - Another thread acquires the mutex and repeats the process
  - Finally the mutex is destroyed

- When several threads compete for a mutex, the losers block at that call - an unblocking call is available with "trylock" instead of the "lock" call.

- When protecting shared data, it is the programmer's responsibility to make sure every thread that needs to use a mutex does so. For example, if 4 threads are updating the same data, but only one uses a mutex, the data can still be corrupted.

Mutex Variables

Creating and Destroying Mutexes

Routines:

pthread_mutex_init (mutex,attr)

pthread_mutex_destroy (mutex)

pthread_mutexattr_init (attr)

pthread_mutexattr_destroy (attr)

Usage:

- Mutex variables must be declared with type `pthread_mutex_t`, and must be initialized before they can be used. There are two ways to initialize a mutex variable:

  1. Statically, when it is declared. For example:
     `pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;`

  2. Dynamically, with the `pthread_mutex_init()` routine. This method permits setting mutex object attributes, *attr*.

  The mutex is initially unlocked.

- The *attr* object is used to establish properties for the mutex object, and must be of type `pthread_mutexattr_t` if used (may be specified as NULL to accept defaults). The Pthreads standard defines three optional mutex attributes:
  - Protocol: Specifies the protocol used to prevent priority inversions for a mutex.
  - Prioceiling: Specifies the priority ceiling of a mutex.
  - Process-shared: Specifies the process sharing of a mutex.

  Note that not all implementations may provide the three optional mutex attributes.

- The `pthread_mutexattr_init()` and `pthread_mutexattr_destroy()` routines are used to create and destroy mutex attribute objects respectively.

- `pthread_mutex_destroy()` should be used to free a mutex object which is no longer needed.

Mutex Variables

Locking and Unlocking Mutexes

Routines:

[pthread_mutex_lock](#) **(mutex)**

[pthread_mutex_trylock](#) **(mutex)**

[pthread_mutex_unlock](#) **(mutex)**

Usage:

- The `pthread_mutex_lock()` routine is used by a thread to acquire a lock on the specified *mutex* variable. If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked.

- `pthread_mutex_trylock()` will attempt to lock a mutex. However, if the mutex is already locked, the routine will return immediately with a "busy" error code. This routine may be useful in preventing deadlock conditions, as in a priority-inversion situation.

- `pthread_mutex_unlock()` will unlock a mutex if called by the owning thread. Calling this routine is required after a thread has completed its use of protected data if other threads are to acquire the mutex for their work with the protected data. An error will be returned if:
  - If the mutex was already unlocked
  - If the mutex is owned by another thread

- There is nothing "magical" about mutexes...in fact they are akin to a "gentlemen's agreement" between participating threads. It is up to the code writer to insure that the necessary threads all make the the mutex lock and unlock calls correctly. The following scenario demonstrates a logical error:

```
Thread 1      Thread 2      Thread 3
Lock          Lock
A = 2         A = A+1       A = A*B
Unlock        Unlock
```

Question: When more than one thread is waiting for a locked mutex, which thread will be granted the lock first after it is released?

Answer

Example: Using Mutexes

Example Code - Using Mutexes

This example program illustrates the use of mutex variables in a threads program that performs a dot product. The main data is made available to all threads through a globally accessible structure. Each thread works on a different part of the data. The main thread waits for all the threads to complete their computations, and then it prints the resulting sum.

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

/*
The following structure contains the necessary information
to allow the function "dotprod" to access its input data and
place its output into the structure.
*/

typedef struct
 {
   double      *a;
   double      *b;
   double     sum;
   int     veclen;
 } DOTDATA;

/* Define globally accessible variables and a mutex */

#define NUMTHRDS 4
#define VECLEN 100
   DOTDATA dotstr;
   pthread_t callThd[NUMTHRDS];
   pthread_mutex_t mutexsum;

/*
The function dotprod is activated when the thread is created.
All input to this routine is obtained from a structure
of type DOTDATA and all output from this function is written into
this structure. The benefit of this approach is apparent for the
multi-threaded program: when a thread is created we pass a single
argument to the activated function - typically this argument
is a thread number. All  the other information required by the
function is accessed from the globally accessible structure.
*/

void *dotprod(void *arg)
{

   /* Define and use local variables for convenience */

   int i, start, end, offset, len ;
   double mysum, *x, *y;
   offset = (int)arg;

   len = dotstr.veclen;
   start = offset*len;
   end   = start + len;
   x = dotstr.a;
   y = dotstr.b;

   /*
   Perform the dot product and assign result
   to the appropriate variable in the structure.
   */

   mysum = 0;
   for (i=start; i < end ; i++)
    {
```

```
      mysum += (x[i] * y[i]);
     }

   /*
   Lock a mutex prior to updating the value in the shared
   structure, and unlock it upon updating.
   */
   pthread_mutex_lock (&mutexsum);
   dotstr.sum += mysum;
   pthread_mutex_unlock (&mutexsum);

   pthread_exit((void*) 0);
}

/*
The main program creates threads which do all the work and then
print out result upon completion. Before creating the threads,
the input data is created. Since all threads update a shared structure,
we need a mutex for mutual exclusion. The main thread needs to wait for
all threads to complete, it waits for each one of the threads. We specify
a thread attribute value that allow the main thread to join with the
threads it creates. Note also that we free up handles when they are
no longer needed.
*/

int main (int argc, char *argv[])
{
   int i;
   double *a, *b;
   int status;
   pthread_attr_t attr;

   /* Assign storage and initialize values */
   a = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
   b = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));

   for (i=0; i < VECLEN*NUMTHRDS; i++)
    {
     a[i]=1.0;
     b[i]=a[i];
    }

   dotstr.veclen = VECLEN;
   dotstr.a = a;
   dotstr.b = b;
   dotstr.sum=0;

   pthread_mutex_init(&mutexsum, NULL);

   /* Create threads to perform the dotproduct  */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

        for(i=0;i < NUMTHRDS;i++)
        {
        /*
        Each thread works on a different set of data.
        The offset is specified by 'i'. The size of
        the data for each thread is indicated by VECLEN.
        */
        pthread_create( &callThd[i], &attr, dotprod, (void *)i);
        }

        pthread_attr_destroy(&attr);

        /* Wait on the other threads */

        for(i=0;i < NUMTHRDS;i++)
        {
          pthread_join( callThd[i], (void **)&status);
        }

   /* After joining, print out the results and cleanup */
   printf ("Sum =  %f \n", dotstr.sum);
   free (a);
   free (b);
   pthread_mutex_destroy(&mutexsum);
   pthread_exit(NULL);
}
```

Source *Serial version*

Source *Pthreads version*

Condition Variables

Overview

- Condition variables provide yet another way for threads to synchronize. While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.
- Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met. This can be very resource consuming since the thread would be continuously busy in this activity. A condition variable is a way to achieve the same goal without polling.
- A condition variable is always used in conjunction with a mutex lock.
- A representative sequence for using condition variables is shown below.

Main Thread
- Declare and initialize global data/variables which require synchronization (such as "count")
- Declare and initialize a condition variable object
- Declare and initialize an associated mutex
- Create threads A and B to do work

Thread A
- Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)
- Lock associated mutex and check value of a global variable
- Call `pthread_cond_wait()` to perform a blocking wait for signal from Thread-B. Note that a call to `pthread_cond_wait()`

Thread B
- Do work
- Lock associated mutex
- Change the value of the global variable that Thread-A is waiting upon.
- Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A.
- Unlock mutex.
- Continue

automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.
- When signalled, wake up. Mutex is automatically and atomically locked.
- Explicitly unlock mutex
- Continue

Main Thread
Join / Continue

Condition Variables

Creating and Destroying Condition Variables

Routines:

pthread_cond_init (condition,attr)

pthread_cond_destroy (condition)

pthread_condattr_init (attr)

pthread_condattr_destroy (attr)

Usage:

- Condition variables must be declared with type `pthread_cond_t`, and must be initialized before they can be used. There are two ways to initialize a condition variable:

  1. Statically, when it is declared. For example:
     `pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;`

  2. Dynamically, with the `pthread_cond_init()` routine. The ID of the created condition variable is returned to the calling thread through the *condition* parameter. This method permits setting condition variable object attributes, *attr*.

- The optional *attr* object is used to set condition variable attributes. There is only one attribute defined for condition variables: process-shared, which allows the condition variable to be seen by threads in other processes. The attribute object, if used, must be of type `pthread_condattr_t` (may be specified as NULL to accept defaults).

  Note that not all implementations may provide the process-shared attribute.

- The `pthread_condattr_init()` and `pthread_condattr_destroy()` routines are used to create and destroy condition variable attribute objects.

- `pthread_cond_destroy()` should be used to free a condition variable that is no longer needed.

Condition Variables

Waiting and Signaling on Condition Variables

Routines:

pthread_cond_wait (condition,mutex)

pthread_cond_signal (condition)

pthread_cond_broadcast (condition)

Usage:

- `pthread_cond_wait()` 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. Should also unlock *mutex* after signal has been received.

- The `pthread_cond_signal()` routine is used to signal (or wake up) another thread which is waiting on the condition variable. It should be called after *mutex* is locked, and must unlock *mutex* in order for `pthread_cond_wait()` routine to complete.

- The `pthread_cond_broadcast()` routine should be used instead of `pthread_cond_signal()` if more than one thread is in a blocking wait state.

- It is a logical error to call `pthread_cond_signal()` before calling `pthread_cond_wait()`.

Proper locking and unlocking of the associated mutex variable is essential when using these routines. For example:

- Failing to lock the mutex before calling `pthread_cond_wait()` may cause it NOT to block.

- Failing to unlock the mutex after calling `pthread_cond_signal()` may not allow a matching `pthread_cond_wait()` routine to complete (it will remain blocked).

Example: Using Condition Variables

Example Code - Using Condition Variables

This simple example code demonstrates the use of several Pthread condition variable routines. The main routine creates three threads. Two of the threads perform work and update a "count" variable. The third thread waits until the count variable reaches a specified value.

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

#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int     count = 0;
int     thread_ids[3] = {0,1,2};
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *idp)
{
  int j,i;
  double result=0.0;
  int *my_id = idp;

  for (i=0; i < TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;

    /*
    Check the value of count and signal waiting thread when condition is
    reached.  Note that this occurs while mutex is locked.
    */
    if (count == COUNT_LIMIT) {
      pthread_cond_signal(&count_threshold_cv);
      printf("inc_count(): thread %d, count = %d  Threshold reached.\n",
             *my_id, count);
      }
    printf("inc_count(): thread %d, count = %d, unlocking mutex\n",
           *my_id, count);
    pthread_mutex_unlock(&count_mutex);

    /* Do some work so threads can alternate on mutex lock */
    for (j=0; j < 1000; j++)
      result = result + (double)random();
    }
  pthread_exit(NULL);
}

void *watch_count(void *idp)
{
  int *my_id = idp;

  printf("Starting watch_count(): thread %d\n", *my_id);

  /*
  Lock mutex and wait for signal.  Note that the pthread_cond_wait
  routine will automatically and atomically unlock mutex while it waits.
  Also, note that if COUNT_LIMIT is reached before this routine is run by
  the waiting thread, the loop will be skipped to prevent pthread_cond_wait
  from never returning.
  */
  pthread_mutex_lock(&count_mutex);
  while (count < COUNT_LIMIT) {
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count(): thread %d Condition signal
           received.\n", *my_id);
    }
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}
```

```
int main (int argc, char *argv[])
{
  int i, rc;
  pthread_t threads[3];
  pthread_attr_t attr;

  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);

  /*
  For portability, explicitly create threads in a joinable state
  so that they can be joined later.
  */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, inc_count, (void *)&thread_ids[0]);
  pthread_create(&threads[1], &attr, inc_count, (void *)&thread_ids[1]);
  pthread_create(&threads[2], &attr, watch_count, (void *)&thread_ids[2]);

  /* Wait for all threads to complete */
  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  printf ("Main(): Waited on %d  threads. Done.\n", NUM_THREADS);

  /* Clean up and exit */
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit(NULL);

}
```

Source Output

LLNL Specific Information and Recommendations

This section describes details specific to Livermore Computing's systems.

Implementations:

- All LC production systems include a Pthreads implementation that follows draft 10 (final) of the POSIX standard. This is the preferred implementation.
- For compatibility with earlier implementations, the IBM compilers provide a means to use the draft 7 version of Pthreads and the Compaq Tru64 compilers a means to use the draft 4 version.
- Implementations differ in the maximum number of threads that a process may create. They also differ in the default amount of thread stack space.

Compiling:

- LC maintains a number of compilers, and usually several different versions of each - see the LC's Supported Compilers web page.

- The compiler commands described in the Compiling Threaded Programs section apply to LC systems.

- Additionally, all LC IBM compilers are aliased to their thread-safe command. For example, xlc really uses xlc_r. This is only true for LC IBM systems.

Mixing MPI with Pthreads:

- Programs that contain both MPI and Pthreads are common and easy to develop on all LC systems.

- Design:
  - Each MPI process typically creates and then manages N threads, where N makes the best use of the available CPUs/node.
  - Finding the best value for N will vary with the platform and your application's characteristics.
  - For ASCI White systems with two communication adapters per node, it may prove more efficient to use two (or more) MPI tasks per node.
  - In general, there may be problems if multiple threads make MPI calls. The program may fail or behave unexpectedly. If MPI calls must be made from within a thread, they should be made only by one thread.

- Compiling:
  - Use the appropriate MPI compile command for the platform and language of choice
  - Be sure to include the required flag as in the table above (-pthread or -qnosave)
  - MPICH is not thread safe

- An example code that uses both MPI and Pthreads is available below. The serial, threads-only, MPI-only and MPI-with-threads versions demonstrate one possible progression.
  - Serial
  - Pthreads only
  - MPI only
  - MPI with pthreads
  - makefile (for IBM SP)

Topics Not Covered

Several features of the Pthreads API are not covered in this tutorial. These are listed below. See the Pthread Library Routines Reference section for more information.

- Thread Scheduling
  - Implementations will differ on how threads are scheduled to run. In most cases, the default mechanism is adequate.
  - The Pthreads API provides routines to explicitly set thread scheduling policies and priorities which may override the default mechanisms.
  - The API does not require implementations to support these features.

- Keys: Thread-Specific Data
  - As threads call and return from different routines, the local data on a thread's stack comes and goes.
  - To preserve stack data you can usually pass it as an argument from one routine to the next, or else store the data in a global variable associated with a thread.
  - Pthreads provides another, possibly more convenient and versatile, way of accomplishing this through ***keys***.

- Mutex Protocol Attributes and Mutex Priority Management for the handling of "priority inversion" problems.

- Condition Variable Sharing - across processes

- Thread Cancellation

- Threads and Signals

Pthread Library Routines Reference

Pthread Functions

Thread Management	pthread_create
	pthread_exit
	pthread_join
	pthread_once
	pthread_kill
	pthread_self
	pthread_equal
	pthread_yield
	pthread_detach
Thread-Specific Data	pthread_key_create
	pthread_key_delete
	pthread_getspecific
	pthread_setspecific

Thread Cancellation	pthread_cancel
	pthread_cleanup_pop
	pthread_cleanup_push
	pthread_setcancelstate
	pthread_getcancelstate
	pthread_testcancel
Thread Scheduling	pthread_getschedparam
	pthread_setschedparam
Signals	pthread_sigmask

Pthread Attribute Functions

Basic Management	pthread_attr_init
	pthread_attr_destroy
Detachable or Joinable	pthread_attr_setdetachstate
	pthread_attr_getdetachstate
Specifying Stack Information	pthread_attr_getstackaddr
	pthread_attr_getstacksize
	pthread_attr_setstackaddr
	pthread_attr_setstacksize
Thread Scheduling Attributes	pthread_attr_getschedparam
	pthread_attr_setschedparam
	pthread_attr_getschedpolicy
	pthread_attr_setschedpolicy
	pthread_attr_setinheritsched
	pthread_attr_getinheritsched
	pthread_attr_setscope
	pthread_attr_getscope

Mutex Functions

Mutex Management	pthread_mutex_init
	pthread_mutex_destroy
	pthread_mutex_lock
	pthread_mutex_unlock
	pthread_mutex_trylock
Priority Management	pthread_mutex_setprioceiling
	pthread_mutex_getprioceiling

Mutex Attribute Functions

Basic Management	pthread_mutexattr_init
	pthread_mutexattr_destroy
Sharing	pthread_mutexattr_getpshared
	pthread_mutexattr_setpshared
Protocol Attributes	pthread_mutexattr_getprotocol
	pthread_mutexattr_setprotocol
Priority Management	pthread_mutexattr_setprioceiling
	pthread_mutexattr_getprioceiling

Condition Variable Functions

Basic Management	pthread_cond_init
	pthread_cond_destroy
	pthread_cond_signal
	pthread_cond_broadcast
	pthread_cond_wait
	pthread_cond_timedwait

Condition Variable Attribute Functions

Basic Management	pthread_condattr_init
	pthread_condattr_destroy
Sharing	pthread_condattr_getpshared
	pthread_condattr_setpshared

This completes the tutorial.

Evaluation Form

Please complete the online evaluation form - unless you are doing the exercise, in which case please complete it at the end of the exercise.

Where would you like to go now?

- Exercise
- Agenda
- Back to the top

References and More Information

- "Pthreads Programming". B. Nichols et al. O'Reilly and Associates.
- "Threads Primer". B. Lewis and D. Berg. Prentice Hall
- "Programming With POSIX Threads". D. Butenhof. Addison Wesley
  www.awl.com/cseng/titles/0-201-63392-2
- "Programming With Threads". S. Kleiman et al. Prentice Hall
- IBM Fortran Compiler Documentation
  www-4.ibm.com/software/ad/fortran
- IBM C/C++ Compiler Documentation
  www-4.ibm.com/software/ad/caix

http://www.llnl.gov/computing/tutorials/pthreads/
Last Modified: Wed, 04 Feb 2004 16:55:55 GMT blaiseb@llnl.gov
UCRL-MI-133316