POSIX thread (pthread) libraries

The POSIX thread libraries are a standards based thread API for C/C++. It allows one to spawn a new concurrent process flow. It is most effective on multi-processor or multi-core systems where the process flow can be scheduled to run on another processor thus gaining speed through parallel or distributed processing. Threads require less overhead than "forking" or spawning a new process because the system does not initialize a new system virtual memory space and environment for the process. While most effective on a multiprocessor system, gains are also found on uniprocessor systems which exploit latency in I/O and other system functions which may halt process execution. (One thread may execute while another is waiting for I/O or some other system latency.) Parallel programming technologies such as MPI and PVM are used in a distributed computing environment while threads are limited to a single computer system. All threads within a process share the same address space. A thread is spawned by defining a function and it's arguments which will be processed in the thread. The purpose of using the POSIX thread library in your software is to execute software faster.

Table of Contents:

Thread Basics:

- Thread operations include thread creation, termination, synchronization (joins,blocking), scheduling, data management and process interaction.
- A thread does not maintain a list of created threads, nor does it know the thread that created it.
- All threads within a process share the same address space.
- Threads in the same process share:
  - Process instructions
  - Most data
  - open files (descriptors)
  - signals and signal handlers
  - current working directory
  - User and group id
- Each thread has a unique:
  - Thread ID
  - set of registers, stack pointer
  - stack for local variables, return addresses
  - signal mask
  - priority
  - Return value: `errno`
- pthread functions return "0" if OK.

Thread Creation and Termination:

Example: `pthread1.c`

```
#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);
}
```

Compile:

- C compiler: `cc -lpthread pthread1.c`
  or
- C++ compiler: `g++ -lpthread pthread1.c`

Run: `./a.out`
Results:

```
Thread 1
Thread 2
Thread 1 returns: 0
Thread 2 returns: 0
```

Details:

- In this example the same function is used in each thread. The arguments are different. The functions need not be the same.

- Threads terminate by explicitly calling `pthread_exit`, by letting the function return, or by a call to the function `exit` which will terminate the process including any threads.

- Function call: **`pthread_create`**

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

  Arguments:
  - `thread` - returns the thread id. (unsigned long int defined in bits/pthreadtypes.h)
  - `attr` - Set to NULL if default thread attributes are used. (else define members of the struct `pthread_attr_t` defined in bits/pthreadtypes.h) Attributes include:
    - detached state (joinable? Default: PTHREAD_CREATE_JOINABLE. Other option: PTHREAD_CREATE_DETACHED)
    - scheduling policy (real-time? PTHREAD_INHERIT_SCHED,PTHREAD_EXPLICIT_SCHED,SCHED_OTHER)
    - scheduling parameter
    - inheritsched attribute (Default: PTHREAD_EXPLICIT_SCHED Inherit from parent thread: PTHREAD_INHERIT_SCHED)
    - scope (Kernel threads: PTHREAD_SCOPE_SYSTEM User threads: PTHREAD_SCOPE_PROCESS Pick one or the other not both.)
    - guard size
    - stack address (See unistd.h and bits/posix_opt.h _POSIX_THREAD_ATTR_STACKADDR)
    - stack size (default minimum PTHREAD_STACK_SIZE set in pthread.h),
  - `void * (*start_routine)` - pointer to the function to be threaded. Function has a single argument: pointer to void.
  - `*arg` - pointer to argument of function. To pass multiple arguments, send a pointer to a structure.

- Function call: **`pthread_exit`**

```
void pthread_exit(void *retval);
```

  Arguments:
  - `retval` - Return value of thread.

  This routine kills the thread. The `pthread_exit` function never returns. If the thread is not detached, the thread id and return value may be examined from another thread by using pthread_join.
  Note: the return pointer `*retval`, must not be of local scope otherwise it would cease to exist once the thread terminates.

- [C++ pitfalls]: The above sample program **will** compile with the GNU C **and** C++ compiler `g++`. The following function pointer representation below will work for C but not C++. Note the subtle differences and avoid the pitfall below:

```
void print_message_function( void *ptr );
...
...
iret1 = pthread_create( &thread1, NULL, (void*)&print_message_function, (void*) message1);
...
...
```

Thread Synchronization:

The threads library provides three synchronization mechanisms:

- 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`

Mutexes:

Mutexes are used to prevent data inconsistencies due to race conditions. A race condition often occurs when two or more threads need to perform operations on the same memory area, but the results of computations depends on the order in which these operations are performed. Mutexes are used for serializing shared resources. Anytime a global resource is accessed by more than one thread the resource should have a Mutex associated with it. One can apply a mutex to protect a segment of memory ("critical region") from other threads. Mutexes can be applied only to threads in a single process and do not work between processes as do semaphores.

Example threaded function:

<table>
<tr><th colspan="2">Without Mutex</th><th colspan="2">With Mutex</th></tr>
<tr><td colspan="2"><pre>
int counter=0;

/* Function C */
void functionC()
{

   counter++

}
</pre></td><td colspan="2"><pre>
/* 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);
}
</pre></td></tr>
<tr><td colspan="4">Possible execution sequence</td></tr>
<tr><th>Thread 1</th><th>Thread 2</th><th>Thread 1</th><th>Thread 2</th></tr>
<tr><td>counter = 0</td><td>counter = 0</td><td>counter = 0</td><td>counter = 0</td></tr>
<tr><td>counter = 1</td><td>counter = 1</td><td>counter = 1</td><td>Thread 2 locked out.
Thread 1 has exclusive use of variable <code>counter</code></td></tr>
<tr><td></td><td></td><td></td><td>counter = 2</td></tr>
</table>

If register load and store operations for the incrementing of variable `counter` occurs with unfortunate timing, it is theoretically possible to have each thread increment and overwrite the same variable with the same value. Another possibility is that thread two would first increment `counter` locking out thread one until complete and then thread one would increment it to 2.

Sequence	Thread 1	Thread 2
1	counter = 0	counter=0

2	Thread 1 locked out. Thread 2 has exclusive use of variable `counter`	counter = 1
3	counter = 2	

Code listing: `mutex1.c`

```
#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( &mutex1 );
   counter++;
   printf("Counter value: %d\n",counter);
   pthread_mutex_unlock( &mutex1 );
}
```

Compile: `cc -lpthread mutex1.c`
Run: `./a.out`
Results:

```
Counter value: 1
Counter value: 2
```

When a mutex lock is attempted against a mutex which is held by another thread, the thread is blocked until the mutex is unlocked. When a thread terminates, the mutex does not unless explicitly unlocked. Nothing happens by default.

Joins:

A join is performed when one wants to wait for a thread to finish. A thread calling routine may launch multiple threads then wait for them to finish to get the results. One wait for the completion of the threads with a join.

Sample code: `join1.c`

```
#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( &thread_id[i], NULL, thread_function, NULL );
   }

   for(j=0; j < NTHREADS; j++)
   {
      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());
   pthread_mutex_lock( &mutex1 );
   counter++;
   pthread_mutex_unlock( &mutex1 );
}
```

Compile: `cc -lpthread join1.c`
Run: `./a.out`
Results:

```
Thread number 1026
Thread number 2051
Thread number 3076
Thread number 4101
Thread number 5126
Thread number 6151
Thread number 7176
Thread number 8201
Thread number 9226
```

```
Thread number 10251
Final counter value: 10
```

Condition Variables:

A condition variable is a variable of type `pthread_cond_t` and is used with the appropriate functions for waiting and later, process continuation. The condition variable mechanism allows threads to suspend execution and relinquish the processor until some condition is true. A condition variable must always be associated with a mutex to avoid a race condition created by one thread preparing to wait and another thread which may signal the condition before the first thread actually waits on it resulting in a deadlock. The thread will be perpetually waiting for a signal that is never sent. Any mutex can be used, there is no explicit link between the mutex and the condition variable.

Functions used in conjunction with the condition variable:

- Creating/Destroying:
  - pthread_cond_init
  - pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
  - pthread_cond_destroy
- Waiting on condition:
  - pthread_cond_wait
  - pthread_cond_timedwait - place limit on how long it will block.
- Waking thread based on condition:
  - pthread_cond_signal
  - pthread_cond_broadcast - wake up all threads blocked by the specified condition variable.

Example code: `cond1.c`

```
#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( &condition_mutex );
       while( count >= COUNT_HALT1 && count <= COUNT_HALT2 )
       {
          pthread_cond_wait( &condition_cond, &condition_mutex );
       }
       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( &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);
    }

}
```

Compile: `cc -lpthread cond1.c`
Run: `./a.out`
Results:

```
Counter value functionCount1: 1
Counter value functionCount1: 2
Counter value functionCount1: 3
Counter value functionCount2: 4
Counter value functionCount2: 5
Counter value functionCount2: 6
Counter value functionCount2: 7
Counter value functionCount1: 8
Counter value functionCount1: 9
Counter value functionCount1: 10
Counter value functionCount2: 11
```

Note that `functionCount1()` was halted while count was between the values COUNT_HALT1 and COUNT_HALT2. The only thing that has been ensures is that `functionCount2` will increment the count between the values COUNT_HALT1 and COUNT_HALT2. Everything else is random.

The logic conditions (the "if" and "while" statements) must be chosen to insure that the "signal" is executed if the "wait" is ever processed. Poor software logic can also lead to a deadlock condition.

Note: Race conditions abound with this example because count is used as the condition and can't be locked in the while statement without causing deadlock. I'll work on a cleaner example but it is an example of a condition variable.

Thread Scheduling:

When this option is enabled, each thread may have its own scheduling properties. Scheduling attributes may be specified:

- during thread creation
- by dynamically by changing the attributes of a thread already created
- by defining the effect of a mutex on the thread's scheduling when creating a mutex
- by dynamically changing the scheduling of a thread during synchronization operations.

The threads library provides default values that are sufficient for most cases.

Thread Pitfalls:

- Race conditions: While the code may appear on the screen in the order you wish the code to execute, threads are scheduled by the operating system and are executed at random. It cannot be assumed that threads are executed in the order they are created. They may also execute at different speeds. When threads are executing (racing to complete) they may give unexpected results (race condition). Mutexes and joins must be utilized to achieve a predictable execution order and outcome.

- Thread safe code: The threaded routines must call functions which are "thread safe". This means that there are no static or global variables which other threads may clobber or read assuming single threaded operation. If static or global variables are used then mutexes must be applied or the functions must be re-written to avoid the use of these variables. In C, local variables are dynamically allocated on the stack. Therefore, any function that does not use static data or other shared resources is thread-safe. Thread-unsafe functions may be used by only one thread at a time in a program and the uniqueness of the thread must be ensured. Many non-reentrant functions return a pointer to static data. This can be avoided by returning dynamically allocated data or using caller-provided storage. An example of a non-thread safe function is `strtok` which is also not re-entrant. The "thread safe" version is the re-entrant version `strtok_r`.

- Mutex Deadlock: This condition occurs when a mutex is applied but then not "unlocked". This causes program execution to halt indefinitely. It can also be caused by poor application of mutexes or joins. Be careful when applying two or more mutexes to a section of code. If the first pthread_mutex_lock is applied and the second pthread_mutex_lock fails due to another thread applying a mutex, the first mutex may eventually lock all other threads from accessing data including the thread which holds the second mutex. The threads may wait indefinitely for the resource to become free causing a deadlock. It is best to test and if failure occurs, free the resources and stall before retrying.

```
...
pthread_mutex_lock(&mutex_1);
while ( pthread_mutex_trylock(&mutex_2) )  /* Test if already locked   */
{
   pthread_mutex_unlock(&mutex_1);  /* Free resource to avoid deadlock */
   ...
   /* stall here   */
   ...
   pthread_mutex_lock(&mutex_1);
}
count++;
pthread_mutex_unlock(&mutex_1);
pthread_mutex_unlock(&mutex_2);
```

```
    ...
```

The order of applying the mutex is also important. The following code segment illustrates a potential for deadlock:

```
void *function1()
{
   ...
   pthread_mutex_lock(&lock1);           - Execution step 1
   pthread_mutex_lock(&lock2);           - Execution step 3 DEADLOCK!!!
   ...
   ...
   pthread_mutex_lock(&lock2);
   pthread_mutex_lock(&lock1);
   ...
}

void *function2()
{
   ...
   pthread_mutex_lock(&lock2);           - Execution step 2
   pthread_mutex_lock(&lock1);
   ...
   ...
   pthread_mutex_lock(&lock1);
   pthread_mutex_lock(&lock2);
   ...
}

main()
{
   ...
   pthread_create(&thread1, NULL, function1, NULL);
   pthread_create(&thread2, NULL, function1, NULL);
   ...
}
```

If `function1` acquires the first mutex and `function2` acquires the second, all resources are tied up and locked.

- Condition Variable Deadlock: The logic conditions (the "if" and "while" statements) must be chosen to insure that the "signal" is executed if the "wait" is ever processed.

Thread Debugging:

- **GDB:**
  - GDB: Stopping and starting multi-thread programs
  - GDB/MI: Threads commands
- **DDD:**
  - Examining Threads

Thread Man Pages:

- pthread_atfork - register handlers to be called at fork(2) time
- pthread_attr_destroy [pthread_attr_init] - thread creation attributes

- pthread_attr_getdetachstate [pthread_attr_init] - thread creation attributes
- pthread_attr_getinheritsched [pthread_attr_init] - thread creation attributes
- pthread_attr_getschedparam [pthread_attr_init] - thread creation attributes
- pthread_attr_getschedpolicy [pthread_attr_init] - thread creation attributes
- pthread_attr_getscope [pthread_attr_init] - thread creation attributes
- pthread_attr_init - thread creation attributes
- pthread_attr_setdetachstate [pthread_attr_init] - thread creation attributes
- pthread_attr_setinheritsched [pthread_attr_init] - thread creation attributes
- pthread_attr_setschedparam [pthread_attr_init] - thread creation attributes
- pthread_attr_setschedpolicy [pthread_attr_init] - thread creation attributes
- pthread_attr_setscope [pthread_attr_init] - thread creation attributes
- pthread_cancel - thread cancellation
- pthread_cleanup_pop [pthread_cleanup_push] - install and remove cleanup handlers
- pthread_cleanup_pop_restore_np [pthread_cleanup_push] - install and remove cleanup handlers
- pthread_cleanup_push - install and remove cleanup handlers
- pthread_cleanup_push_defer_np [pthread_cleanup_push] - install and remove cleanup handlers
- pthread_condattr_destroy [pthread_condattr_init] - condition creation attributes
- pthread_condattr_init - condition creation attributes
- pthread_cond_broadcast [pthread_cond_init] - operations on conditions
- pthread_cond_destroy [pthread_cond_init] - operations on conditions
- pthread_cond_init - operations on conditions
- pthread_cond_signal [pthread_cond_init] - operations on conditions
- pthread_cond_timedwait [pthread_cond_init] - operations on conditions
- pthread_cond_wait [pthread_cond_init] - operations on conditions
- pthread_create - create a new thread
- pthread_detach - put a running thread in the detached state
- pthread_equal - compare two thread identifiers
- pthread_exit - terminate the calling thread
- pthread_getschedparam [pthread_setschedparam] - control thread scheduling parameters
- pthread_getspecific [pthread_key_create] - management of thread-specific data
- pthread_join - wait for termination of another thread
- pthread_key_create - management of thread-specific data
- pthread_key_delete [pthread_key_create] - management of thread-specific data
- pthread_kill_other_threads_np - terminate all threads in program except calling thread
- pthread_kill [pthread_sigmask] - handling of signals in threads
- pthread_mutexattr_destroy [pthread_mutexattr_init] - mutex creation attributes
- pthread_mutexattr_getkind_np [pthread_mutexattr_init] - mutex creation attributes
- pthread_mutexattr_init - mutex creation attributes
- pthread_mutexattr_setkind_np [pthread_mutexattr_init] - mutex creation attributes
- pthread_mutex_destroy [pthread_mutex_init] - operations on mutexes
- pthread_mutex_init - operations on mutexes
- pthread_mutex_lock [pthread_mutex_init] - operations on mutexes
- pthread_mutex_trylock [pthread_mutex_init] - operations on mutexes
- pthread_mutex_unlock [pthread_mutex_init] - operations on mutexes
- pthread_once - once-only initialization
- pthread_self - return identifier of current thread
- pthread_setcancelstate [pthread_cancel] - thread cancellation
- pthread_setcanceltype [pthread_cancel] - thread cancellation
- pthread_setschedparam - control thread scheduling parameters
- pthread_setspecific [pthread_key_create] - management of thread-specific data
- pthread_sigmask - handling of signals in threads
- pthread_testcancel [pthread_cancel] - thread cancellation

Links:

- Fundamentals Of Multithreading - Paul Mazzucco
- Native Posix Thread Library for Linux
- Introduction to Programming Threads
- Getting Started With POSIX Threads
- ITS: Introduction to Threads
- GNU Portable Threads
- Introduction of threads for Solaris, Linux, and Windows
- Comparison of thread implementations
- comp.programming.threads FAQ
- An in-depth description of PMPthread internal queue functions.
- Examples
- Pthreads tutorial and examples of thread problems - by Andrae Muys
- Valgrind KDE thread checker: Helgrind
- Sun's Multithreaded Programming Guide - Not Linux but a good reference.
- FSU Pthreads (POSIX Threads)
- Linux-mag.com: Concurrent Programming Topics - semaphores, condition variables
- Linux-mag.com: The Fibers of Threads - Discussion of how Linux threads work
- Platform independent threads:
  - Gnome GLib 2.0 threads - Thread abstraction; including mutexes, conditions and thread private data. [example]
  - OmniORB (CORBA) Thread Library
  - zThreads
- **C++ Thread classes:**
  - GNU: Common C++ - support for threading, sockets, file access, daemons, persistence, serial I/O, XML parsing and system services
  - ACE: Adaptive Communication Environment - C++ interface
    - ACE programmers guide: [pdf] (see page 29 for threads)
    - Thread management examples using ACE
  - Hood - A C++ Threads Library for Multiprogrammed Multiprocessors
  - C++ Thread classes - sourceforge
  - QpThread

News Groups:

- comp.programming.threads
- comp.unix.solaris

Books:

	Pthreads Programming A POSIX Standard for Better Multiprocessing By Bradford Nichols, Dick Buttlar, Jacqueline Proulx Farrell ISBN #1-56592-115-1, O'Reilly	Amazon.com
	Programming with POSIX(R) Threads By David R. Butenhof ISBN #0201633922, Addison Wesley Pub. Co.	Amazon.com
	C++ Network Programming Volume 1 By Douglas C. Schmidt, Stephen D. Huston ISBN #0201604647, Addison Wesley Pub. Co.	Amazon.com

	Covers ACE (ADAPTIVE Communication Environment) open-source framework view of threads and other topics.	
	Dr. Dobb's Journal Free subscription to the premier resource for professional programmers and software developers. Multi-language and multi-platform with program listings, coding tips, design issue discussions and algorithms. Subscribe here!	Free Subscription

See http://YoLinux.com for more Linux information and tutorials
Return to YoLinux Tutorial Index
Feedback Form