**What are Pthreads?**

POSIX Threads, or Pthreads, is a POSIX standard for threads. The standard, POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995), defines an API for creating and manipulating threads.

Implementations of the API are available on many Unix-like POSIX systems such as FreeBSD, NetBSD, GNU/Linux, Mac OS X and Solaris, but Microsoft Windows implementations also exist. For example, the pthreads-w32 is available and supports a subset of the Pthread API for the Windows 32-bit platform.

The POSIX standard has continued to evolve and undergo revisions, including the Pthreads specification. The latest version is known as IEEE Std 1003.1, 2004 Edition.

Pthreads are defined as a set of C language programming types and procedure calls, implemented with a pthread.h header file. In GNU/Linux, the pthread functions are not included in the standard C library. They are in libpthrea, therefore, we should add -lpthread to link our program.

**The Pthread API**

Pthreads API can be grouped into four:

**Thread management:**

Routines that work directly on threads - creating, detaching, joining, etc. They also include functions to set/query thread attributes such as joinable, scheduling etc.

**Mutexes:**

Routines that deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. These are supplemented by mutex attribute functions that set or modify attributes associated with mutexes.

**Condition variables:**

Routines that address communications between threads that share a mutex. Based upon programmer specified conditions. This group includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.

**Synchronization:**

Routines that manage read/write locks and barriers.

**Creating Threads**

1. Our **main()** program is a single, default thread. All other threads must be explicitly created by the programmer.
2. **pthread\_create** creates a new thread and makes it executable. This routine can be called any number of times from anywhere within our code.
3. **pthread\_create (pthread\_t \*thread, pthread\_attr\_t \*attr, void \*(\*start\_routine)(void \*), void \*arg)**arguments:

**1->thread:**   
An identifier for the new thread returned by the subroutine. This is a pointer to **pthread\_t** structure. When a thread is created, an identifier is written to the memory location to which this variable points. This identifier enables us to refer to the thread.

**2->attr:**   
An attribute object that may be used to set thread attributes. We can specify a thread attributes object, or NULL for the default values.

**3->start\_routine:**   
The routine that the thread will execute once it is created.

1. void \*(\*start\_routine)(void \*

We should pass the address of a function taking a pointer to void as a parameter and the function will return a pointer to void. So, we can pass any type of single argument and return a pointer to any type.   
While using **fork()** causes execution to continue in the same location with a different return code, using a new thread explicitly provides a pointer to a function where the new thread should start executing.

4->**arg:**   
A single argument that may be passed to **start\_routine**. It must be passed as a **void pointer**. NULL may be used if no argument is to be passed.

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

5-> Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads.

6->Here is a sample of creating a child thread:

// thread0.c

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

void \*worker\_thread(void \*arg)

{

printf("This is worker\_thread()\n");

pthread\_exit(NULL);

}

int main()

{

pthread\_t my\_thread;

int ret;

printf("In main: creating thread\n");

ret = pthread\_create(&my;\_thread, NULL, &worker;\_thread, NULL);

if(ret != 0) {

printf("Error: pthread\_create() failed\n");

exit(EXIT\_FAILURE);

}

pthread\_exit(NULL);

}

In the code, the main thread will create a second thread to execute **worker\_thread()**, which will print out its message while main thread prints another. The call to create the thread has a NULL value for the attributes, which gives the thread default attributes. The call also passes the address of a **my\_thread** variable for the **worker\_thread()** to store a handle to the thread. The return value from the **pthread\_create()** call will be zero if it's successful, otherwise, it returns an error.

To Run.: $ gcc -o thread0 thread0.c -lpthread

$ ./thread0

In main: creating thread

This is worker\_thread()

**We can create several child threads:\***

// thread01.c

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#define N 5

void \*worker\_thread(void \*arg)

{

printf("This is worker\_thread #%ld\n", (long)arg);

pthread\_exit(NULL);

}

int main()

{

pthread\_t my\_thread[N];

long id;

for(id = 1; id <= N; id++) {

int ret = pthread\_create(&my;\_thread[id], NULL, &worker;\_thread, (void\*)id);

if(ret != 0) {

printf("Error: pthread\_create() failed\n");

exit(EXIT\_FAILURE);

}

}

pthread\_exit(NULL);

}

Output.:

$ ./thread01

This is worker\_thread #5

This is worker\_thread #4

This is worker\_thread #3

This is worker\_thread #2

This is worker\_thread #1

**Note that, in the code, we pass the parameter (thread id) to the child thread.**

If we do (void\*)&id, it's a wrong way of passing data to the child thread. It passes the address of variable **id**, which is shared memory space and visible to all threads. As the loop iterates, the value of this memory location changes, possibly before the created threads can access it.

**Attributes of Threads**

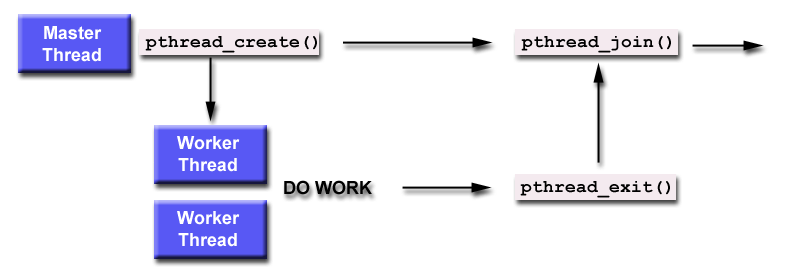
1. By default, a thread is created with certain attributes. Some of these attributes can be changed by the programmer via the thread attribute object.
2. **pthread\_attr\_init()** and **pthread\_attr\_destroy()** are used to initialize/destroy the thread attribute object.
3. Other routines are then used to query/set specific attributes in the thread attribute object.

Terminating Threads

1. There are several ways in which a Pthread may be terminated:
   1. The thread returns from its starting routine (the main routine for the initial thread).
   2. The thread makes a call to the **pthread\_exit** subroutine.
   3. The thread is canceled by another thread via the **pthread\_cancel** routine
   4. The entire process is terminated due to a call to either the exec or exit subroutines.`
2. **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.   
   So, if we comment out the line **pthread\_exit()** in **main()** in the **thread01.c** of the previous example code, the threads created may not have a chance to execute their work before being terminated.
3. The programmer may optionally specify a termination **status**, which is stored as a void pointer for any thread that may join the calling thread.
4. Cleanup: the **pthread\_exit()** routine does not close files; any files opened inside the thread will remain open after the thread is terminated.

before aquiring the lock just check for queue count :)   
  
**Join**

1. **int pthread\_join (pthread\_t th, void \*\*thread\_return)**  
   The first parameter is the thread for which to wait, the identified that **pthread\_create**filled in for us. The second argument is a pointer to a pointer that itself points to the return value from the thread. This function returns zero for success and an error code on failure.
2. When a thread is created, one of its attributes defines whether the thread 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.
3. The final draft of the POSIX standard specifies that threads should be created as joinable.
4. 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()**
5. Here is the summary for the join related functions:  
   1. **pthread\_join (threadid,status)**
   2. **pthread\_detach (threadid)**
   3. **pthread\_attr\_setdetachstate (attr,detachstate)**
   4. **pthread\_attr\_getdetachstate (attr,detachstate)**



A thread can execute a thread join to wait until the other thread terminates. In our case, you - the main thread - should execute a thread join waiting for your colleague - a child thread - to terminate. In general, thread join is for a parent (**P**) to join with one of its child threads (**C**). Thread join has the following activities, assuming that a parent thread **P** wants to join with one of its child threads **C**:

1. When **P** executes a thread join in order to join with **C**, which is still running, **P** is suspended until **C** terminates. Once **C** terminates, **P** resumes.
2. When **P** executes a thread join and **C** has already terminated, **P** continues as if no such thread join has ever executed (i.e., join has no effect).

A parent thread may join with many child threads created by the parent. Or, a parent only join with some of its child threads, and ignore other child threads. In this case, those child threads that are ignored by the parent will be terminated when the parent terminates.

1. The **pthread\_join()** subroutine **blocks the calling thread** until the specified thread terminates.
2. 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()** as show here:
3. void \*worker\_thread(void \*arg)
4. { pthread\_exit((void\*)911); }
5. int main()
6. { int i;
7. pthread\_t thread;
8. pthread\_create(&thread;, NULL, worker\_thread, NULL);
9. pthread\_join(thread, (void \*\*)&i);
10. printf("%d\n",i); // will print out 911

3->A joining thread can match one **pthread\_join()** call. It is a logical error to attempt multiple joins on the same thread.

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()**.

**#include <pthread.h>**

**#include <stdio.h>**

**#define NUM\_THREADS 5**

**void \*PrintHello(void \*threadid)**

**{**

**long tid;**

**tid = (long)threadid;**

**printf("Hello World! It's me, thread #%ld!\n", tid);**

**pthread\_exit(NULL);**

**}**

**int main (int argc, char \*argv[])**

**{**

**pthread\_t threads[NUM\_THREADS];**

**int rc;**

**long t;**

**for(t=0; t<NUM\_THREADS; t++){**

**printf("In main: creating thread %ld\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);**

**}**

**}**

**/\* Last thing that main() should do \*/**

**pthread\_exit(NULL);**

**}**