What are Pthreads?

1. **POSIX Threads**, or **Pthreads**, is a POSIX standard for threads.
2. 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:
   * **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.
   * **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.
   * **start\_routine:**   
     The routine that the thread will execute once it is created.

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.
  + **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.

1. The maximum number of threads that may be created by a process is implementation dependent.
2. Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads.
3. Here is a sample of creating a child thread:
4. // thread0.c
5. #include <pthread.h>
6. #include <stdio.h>
7. #include <stdlib.h>
8. void \*worker\_thread(void \*arg)
9. {
10. printf("This is worker\_thread()\n");
11. pthread\_exit(NULL);
12. }
13. int main()
14. {
15. pthread\_t my\_thread;
16. int ret;
17. printf("In main: creating thread\n");
18. ret = pthread\_create(&my;\_thread, NULL, &worker;\_thread, NULL);
19. if(ret != 0) {
20. printf("Error: pthread\_create() failed\n");
21. exit(EXIT\_FAILURE);
22. }
23. pthread\_exit(NULL);
24. }

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 it:

$ 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 is:

$ ./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.