

POSIX Threads (pthreads) – Documentation

Modern programs often need to perform several tasks simultaneously — for example, downloading data while updating the user interface, or serving many clients on a network. This is achieved through multithreading, where multiple sequences of execution (threads) share the same process resources.

In UNIX and Linux, the standard for implementing threads is POSIX Threads, commonly called pthreads.

What are Pthreads?

- **Pthreads** are a standardized **C library (POSIX.1c)** for creating and managing threads on UNIX-like systems.
- Each thread represents an independent flow of control within a process.
- All threads of a process share:
 - Code section
 - Global variables and heap
 - Open files and signals


but have **their own stack, registers, and instruction pointer**.

Why Pthreads are Needed?

Need	Explanation
Parallelism	Exploit multicore CPUs — threads can run truly in parallel.
Responsiveness	Background work (I/O, computations) can run without freezing the main program.
Resource sharing	Threads share memory, so communication is simpler and faster than inter-process communication.
Efficiency	Creating a thread is cheaper than creating a full process (fork).

The Threads API

To use pthreads in C, include:



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

and compile with:

```
gcc -pthread program.c -o program
```

Important pthread data types

Type	Description
<code>pthread_t</code>	Represents a thread ID
<code>pthread_attr_t</code>	Thread attributes (stack size, detach state, etc.)
<code>pthread_mutex_t</code>	Mutual exclusion lock
<code>pthread_cond_t</code>	Condition variable for <u>thread synchronization</u>

Thread Management Functions

Function	Purpose
<code>pthread_create()</code>	Create a new thread
<code>pthread_exit()</code>	Terminate the calling thread
<code>pthread_join()</code>	Wait for a thread to finish
<code>pthread_self()</code>	Get current thread's ID

`pthread_detach` Mark a thread as detached (no join needed)
`h()`

Example 1 — Basic Thread Creation

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

void* print_message(void* arg) {
    printf("Hello from thread! Message: %s\n", (char*)arg);
    return NULL; // returning ends thread execution
}

int main() {
    pthread_t tid;
    char* msg = "Learning pthreads";

    // Create a new thread
    pthread_create(&tid, NULL, print_message, msg);

    // Wait for the thread to finish
    pthread_join(tid, NULL);

    printf("Main thread finished.\n");
    return 0;
}
```

Explanation:

- `pthread_create()` starts a new thread executing `print_message()`.
- `pthread_join()` waits for it to complete before `main()` exits.

Compile & run:

```
gcc -pthread thread1.c -o thread1
```

```
./thread1
```

Tutorial Codes and Concepts

(a) Thread Initialization

To create a thread, you call:

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

thread → variable to store thread ID.

attr → thread attributes (usually **NULL** for default).

start_routine → function the thread will run.

arg → argument passed to the function.

Example:

```
pthread_t tid;  
pthread_create(&tid, NULL, myFunction, NULL);
```

(b) Terminating Thread Execution

A thread can terminate in several ways:

1. Return from its start function
2. Call **pthread_exit()**
3. Be canceled by another thread

Example:

```
void* worker(void* arg) {  
    printf("Worker running...\n");  
    pthread_exit(NULL); // terminates here  
}
```

If the main thread ends **before** other threads, the process terminates immediately.

Hence, always use **pthread_join()** or **pthread_exit(NULL)** in **main()** to keep it alive.

(c) Getting the Thread Identifier

Each thread has a unique ID of type **pthread_t**.

```
pthread_t tid = pthread_self();
```

```
printf("My thread ID: %lu\n", tid);
```

Note: `%lu` is used after casting to `unsigned long` for display.

(d) Creation and Termination of Multiple Threads

Example Code:

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

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

```
#include <unistd.h>
```

```
#define NUM_THREADS 3
```

```
void* print_task(void* arg) {
```

```
    int id = *(int*)arg;
```

```
    printf("Thread %d started. ID = %lu\n", id, pthread_self());
```

```
    sleep(1); // simulate some work
```

```
    printf("Thread %d finished.\n", id);
```

```
    return NULL;
```

```
}
```

```
int main() {
```

```
    pthread_t threads[NUM_THREADS];
```

```
    int ids[NUM_THREADS];
```

```
    // Create multiple threads
```

```
    for (int i = 0; i < NUM_THREADS; i++) {
```

```

        ids[i] = i + 1;

        pthread_create(&threads[i], NULL, print_task, &ids[i]);
    }

    // Wait for all threads to finish

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("All threads have completed.\n");

    return 0;
}

```

Explanation

- `pthread_t threads[NUM_THREADS];` — an array of thread handles.
- Each thread receives a unique integer ID.
- `pthread_join()` ensures the main thread waits until all finish.
- `sleep(1)` simulates independent work per thread.

Sample Output:

Thread 1 started. ID = 140393871353600

Thread 2 started. ID = 140393862960896

Thread 3 started. ID = 140393854568192

Thread 3 finished.

Thread 2 finished.

Thread 1 finished.

All threads have completed.

(Execution order may vary due to scheduling.)

