**Exercise 3 – Multithreading and pthread in C**

**Notes from Internet – Series-2**

A thread is a single sequence stream within in a process. As threads have some of the properties of processes, they are sometimes called lightweight processes. In short, thread is a unit of a process.

### What are the differences between process and thread?

Threads are not independent of one other like processes as a result threads shares with other threads their:

* code section
* data section
* OS resources like open files and signals

Like process, a thread has its own:

* program counter (PC)
* a register set
* a stack space.

### Multithreading

Thread is a programming technique to improve application performance through parallel processes. For example, in a browser, multiple tabs can be different threads. MS word uses multiple threads, one thread to format the text, other thread to process inputs and so on.

Threads operate faster than processes due to following reasons:

1. Thread creation is much faster
2. Context switching between threads is much faster
3. Threads can be terminated easily
4. Communication between threads is faster

Multithreading is not supported by the language standard in C. POSIX Threads (or Pthreads) is a POSIX standard for threads. Implementation of pthread is available with gcc compiler.

Following is a basic program using pthreads:

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**#include <pthread.h>**

**void \*myThread(void \*vargp)**

**{**

**sleep(1);**

**printf("Inside Thread \n");**

**return NULL;**

**}**

**int main()**

**{**

**pthread\_t thread\_id;**

**printf("Before Thread\n");**

**pthread\_create(&thread\_id, NULL, myThread, NULL);**

**pthread\_join(thread\_id, NULL);**

**printf("After Thread\n");**

**exit(0);**

**}**

### **Explanation of the above code**

In main(), we declare a variable called thread\_id, which is of type pthread\_t, which is an integer used to identify the thread in the system. After declaring thread\_id, we call pthread\_create() function to create a thread. pthread\_create() takes 4 arguments.

* The first argument is a pointer to thread\_id which is set by this function.
* The second argument specifies attributes. If the value is NULL, then default attributes shall be used.
* The third argument is name of function to be executed for the thread to be created.
* The fourth argument is used to pass arguments to the function, myThread.

The pthread\_join() function for threads is the equivalent of wait() for processes. A call to pthread\_join blocks the calling thread until the thread with identifier equal to the first argument terminates.

### **1.0 POSIX threads**

A [process](https://www.softprayog.in/programming/program-process-threads) is an execution environment in an operating system. A process has code and data segments which are initialized from a program during an [exec](https://www.softprayog.in/programming/creating-processes-with-fork-and-exec-in-linux) system call. A process has a thread of execution, wherein instructions are executed as per the value of the program counter register. Associated with a process is a stack and stack pointer has the address of the top of the stack.

Operating systems support multiple processes in a typical time sharing environment. They switch processes (fast) by saving the processor registers and loading another set of previously saved registers. Just like supporting multiple processes, modern operating systems support multiple threads of execution for a process. Rather than having one (saved copy of) stack pointer and one program counter for a process, they have one set of registers (including stack pointer and the program counter) for each thread of execution for a process. The global data and file pointers are shared between threads. So, it's OK if some threads wish to block (wait) for some event like arrival of data on a socket connection; other threads can do something useful in the meantime. So, essentially, we have concurrent multiple flows of execution without incurring the overhead of multiple processes.

POSIX stands for Portable Operating System Interface, a set of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems. POSIX.1c (IEEE Std 1003.1c-1995) deals with Thread extensions. POSIX threads refer to the API defined by the POSIX.1c standard. The Native POSIX Thread Library (NPTL) is an implementation of POSIX threads for Linux. NPTL was developed at Red Hat and has been a part of Linux since version 2.6. POSIX threads are commonly known as Pthreads.

### **2.0 Basic Pthread calls**

We can get started with Pthreads with four basic calls, pthread\_create, pthread\_join, pthread\_cancel and pthread\_exit. A process has the default main thread after an [exec](https://www.softprayog.in/programming/creating-processes-with-fork-and-exec-in-linux) system call. The main thread starts executing the main function of the program.

### **2.1 pthread\_create**

#include <pthread.h>

int pthread\_create (pthread\_t \*thread, const pthread\_attr\_t \*attr,

void \*(\*start\_routine) (void \*), void \*arg);

pthread\_create starts a new thread in the calling process. The first argument, thread, is a pointer to the type pthread\_t in which the thread id is returned. The second argument, attr is a pointer to desired attributes and can be NULL if you are happy with the default attributes. The last parameter, arg, is a pointer and provides a way to pass data to the start\_routine. The third parameter, start\_routine is the function that the newly created thread starts executing. The start\_routine must have the prototype,

void \*start\_routine (void \*arg);

pthread\_create returns zero on success and a nonzero error number in the case of failure.

### **2.2 pthread\_join**

#include <pthread.h>

int pthread\_join (pthread\_t thread, void \*\*retval);

pthread\_join waits for the thread identified by the first argument to terminate. The calling thread's id cannot be the first argument which means a thread cannot wait for itself to terminate. Also, two threads cannot wait for each other to terminate. In all these cases the error EDEADLK is returned. If retval is not NULL, the pointer to the exit status of the thread is copied in \*retval. However, if the thread was cancelled, PTHREAD\_CANCELED is copied in \*retval.

### **2.3 pthread\_cancel**

#include <pthread.h>

int pthread\_cancel (pthread\_t thread);

pthread\_cancel cancels the thread.

### **2.4 pthread\_exit**

#include <pthread.h>

void pthread\_exit (void \*retval);

pthread\_exit terminates the calling thread. A thread can pass an exit value by passing a pointer to the exit value in the pthread\_exit call. The exit value should have a scope such that it is visible to all threads which means that it should be a global variable at the top of the program.

### **3.0 Pthread error reporting**

Pthread calls return zero on success and if there is an error, that error is returned.

### **4.0 An example**

In the example program, the main thread creates ten threads. Then it waits for the threads to terminate, printing the status returned by each thread. The last thread was cancelled which is recorded in the output.

/\*

\*

\* threads-example.c: Program to demonstrate Pthreads in C

\*/

#include <sys/types.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <errno.h>

#include <pthread.h>

#include <unistd.h>

void \*ethread (void \*arg);

char ret\_status [10] [100];

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

{

pthread\_t tid [10];

int i, r;

void \*status;

char buffer [128];

// Create 10 threads

int thread\_no [10];

for (i = 0; i < 10; i++) {

thread\_no [i] = i;

if ((r = pthread\_create (&tid [i], NULL, ethread, (void \*) &thread\_no [i])) != 0) {

strerror\_r (r, buffer, sizeof (buffer));

fprintf (stderr, "Error = %d (%s)\n", r, buffer); exit (1);

}

}

if ((r = pthread\_cancel (tid [9])) != 0) {

strerror\_r (r, buffer, sizeof (buffer));

fprintf (stderr, "Error = %d (%s)\n", r, buffer); exit (1);

}

// Wait for threads to terminate

for (i = 0; i < 10; i++) {

if ((r = pthread\_join (tid [i], &status)) != 0) {

strerror\_r (r, buffer, sizeof (buffer));

fprintf (stderr, "Error = %d (%s)\n", r, buffer); exit (1);

}

if (status == PTHREAD\_CANCELED)

printf ("i = %d, status = CANCELED\n", i);

else

printf ("i = %d, status = %s\n", i, (char \*) status);

}

exit (0);

}

// ethread: example thread

void \*ethread (void \*arg)

{

int my\_id = \*((int \*) arg);

// Take a nap

sleep (1);

// say hello and terminate

printf ("Thread %d: Hello World!\n", my\_id);

sprintf (ret\_status [my\_id], "Thread %d: %d", my\_id, my\_id + 10);

if (my\_id == 9) sleep (10);

// pass your id to the thread waiting for you to terminate

// using pthread\_join.

pthread\_exit (ret\_status [my\_id]);

}

We can compile and run the above program as shown below.

$ gcc threads-example.c -o threads-example -lpthread

$ ./threads-example

Thread 4: Hello World!

Thread 0: Hello World!

Thread 1: Hello World!

i = 0, status = Thread 0: 10

i = 1, status = Thread 1: 11

Thread 5: Hello World!

Thread 6: Hello World!

Thread 7: Hello World!

Thread 3: Hello World!

Thread 2: Hello World!

Thread 8: Hello World!

i = 2, status = Thread 2: 12

i = 3, status = Thread 3: 13

i = 4, status = Thread 4: 14

i = 5, status = Thread 5: 15

i = 6, status = Thread 6: 16

i = 7, status = Thread 7: 17

i = 8, status = Thread 8: 18

i = 9, status = CANCELED

## Syntax

### **1. Header file**

Include the header file pthread.h.

**#include <pthread.h>**

### **2. The ID of a thread**

Each thread has an object of type pthread\_t associated with it that tells its ID. The same pthread\_t object cannot be used by multiple threads simultaneously. For multiple threads, an array can be created where each element is an ID for a separate thread:

**pthread\_t id[2];**

### **3. Creating a thread**

A thread is created and starts using the function pthread\_create(). It takes four parameters:

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| ID | pthread\_t \* | Reference (or pointer) to the ID of the thread. |
| Attributes | pthread\_attr\_t \* | Used to set the attributes of a thread(e.g., the stack size, scheduling policy, etc.) Passing NULL suffices for most applications. |
| Starting routine | void \* | The name of the function that the thread starts to execute. If the function’s return type is void \*, then its name is simply written; otherwise, it has to be type-cast to void \*. |
| Arguments | void \* | This is the argument that the starting routine takes. If it takes multiple arguments, a [struct](https://www.educative.io/edpresso/how-to-use-the-typedef-struct-in-c) is used. |

The return type of a starting routine and its argument is usually set to void \*.

pthread\_create(&id[0], NULL, printNumber, &arg);

### **4. Exiting a thread**

pthread\_exit() is used to exit a thread. This function is usually written at the end of the starting routine. If a value is returned by a thread upon ending, its reference is passed as an argument. Since a thread’s local variables are destroyed when they exit, only references to global or dynamic variables are returned.

#### Starting routine example

// Global variable:

int i = 1;

// Starting routine:

void\* foo(void\* p){

int i = \*(int\*) p;

printf("Received value: %i", i);

// Return reference to global variable:

pthread\_exit(&i);

}

### **5. Waiting for a thread**

A parent thread is made to wait for a child thread using pthread\_join(). The two parameters of this function are:

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| Thread ID | pthread\_t | The ID of the thread that the parent thread waits for. |
| Reference to return value | void \*\* | The value returned by the exiting thread is caught by this pointer. |

int\* ptr;

pthread\_join(id, &ptr);

#include <stdio.h>

#include <string.h>

#include <pthread.h>

// Global variable:

int i = 2;

void\* foo(void\* p){

// Print value received as argument:

printf("Value recevied as argument in starting routine: ");

printf("%i\n", \* (int\*)p);

// Return reference to global variable:

pthread\_exit(&i);

}

int main(void){

// Declare variable for thread's ID:

pthread\_t id;

int j = 1;

pthread\_create(&id, NULL, foo, &j);

int\* ptr;

// Wait for foo() and retrieve value in ptr;

pthread\_join(id, (void\*\*)&ptr);

printf("Value recevied by parent from child: ");

printf("%i\n", \*ptr);

}

Output

Value recevied as argument in starting routine: 1

Value recevied by parent from child: 2