Objective:

This lab examines aspects of threads and multiprocessing (and multithreading). The primary objective of this lab is to implement the Thread Management Functions using POSIX thread (pthread) libraries. We will learn how to:

1. Create Threads
2. Terminate Thread Execution
3. Pass Arguments to Threads
4. Thread Identifiers
5. JoinThreads
6. Detaching / Undetaching Threads

Introduction:

Some terminologies we will study in this lab:

- ***Process:***
  A process is an instance of a program executable (ex: vi) that is identified as a unique entity in the operating environment (ex: linux). It has its own address space and system state information (ex: handles to system resources). For two processes to communicate with one another, they need to use some form of Inter Process Communication (IPC) mechanism (ex: pipes, sockets, shared memory).
- ***Thread:***
  A thread is a path of execution that is identified by a unique entity within a process. A process can have multiple threads of execution and they all share the same process address space and system state information. Since all the threads within a process share the same address space, they can communicate with each other directly using program objects.

Pthreads:

Pthreads or **POSIX** are defined as a set of C language programming types and procedure calls. Vendors usually provide a Pthreads implementation in the form of a header/include file and a library, which you link with your program.

The Pthreads API :

The subroutines which comprise the Pthreads API can be informally grouped into three major classes:

1. **Thread management:** The first class of functions work directly on threads - creating, detaching, joining, etc. They include functions to set/query thread attributes (joinable, scheduling etc.)

2. **Mutexes**: The second class of functions deal with a coarse type of synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. They are also supplemented by mutex attribute functions that set or modify attributes associated with mutexes.

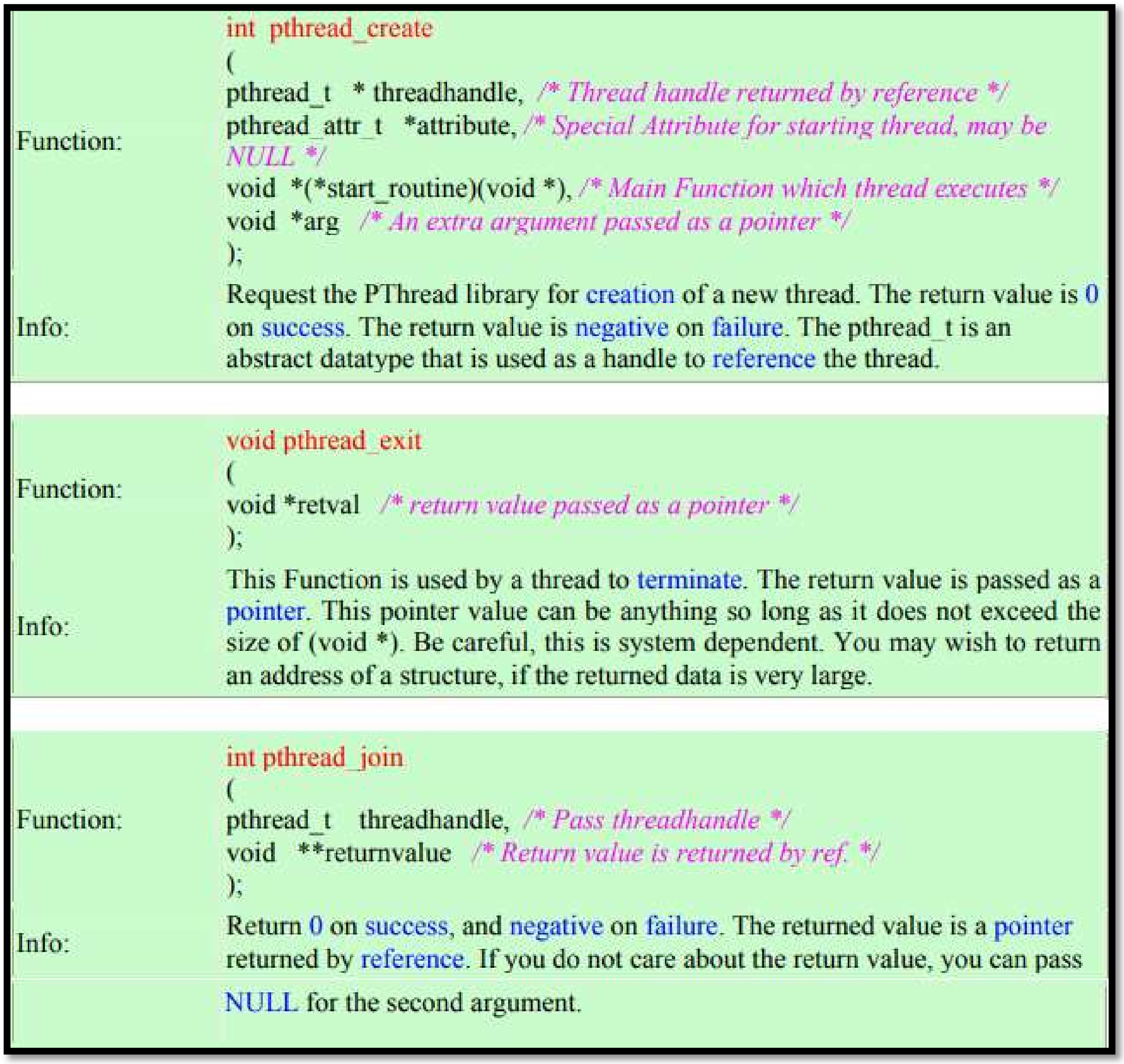
3. **Condition variables:** The third class of functions deal with a finer type of synchronization - based upon programmer specified conditions. This class includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.

Naming conventions: All identifiers in the threads library begin with pthread_

pthread_	Threads themselves and miscellaneous subroutines
pthread_t	Thread objects
pthread_attr	Thread attributes objects
pthread_mutex	Mutexes
pthread_mutexattr	Mutex attributes objects
pthread_cond	Condition variables
pthread_condattr	Condition attributes objects
pthread_key	Thread-specific data keys

Thread Management Functions:

The function **pthread_create** is used to create a new thread, and a thread to terminate itself uses the function **pthread_exit**. A thread to wait for termination of another thread uses the function **pthread_join.**



	int pthread_create
	(
	pthread_t * threadhandle, /* Thread handle returned by reference */
Function:	pthread_attr_t *attribute, /* Special Attribute for starting thread, may be NULL */
	void *(*start_routine)(void *), /* Main Function which thread executes */
	void *arg /* An extra argument passed as a pointer */
	);
Info:	Request the PThread library for creation of a new thread. The return value is 0 on success. The return value is negative on failure. The pthread_t is an abstract datatype that is used as a handle to reference the thread.
	void pthread_exit
	(
Function:	void *retval /* return value passed as a pointer */
	);
Info:	This Function is used by a thread to terminate. The return value is passed as a pointer. This pointer value can be anything so long as it does not exceed the size of (void *). Be careful, this is system dependent. You may wish to return an address of a structure, if the returned data is very large.
	int pthread_join
	(
Function:	pthread_t threadhandle, /* Pass threadhandle */
	void **returnvalue /* Return value is returned by ref. */
	);
Info:	Return 0 on success, and negative on failure. The returned value is a pointer returned by reference. If you do not care about the return value, you can pass NULL for the second argument.

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.
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.

Thread Identifiers:

pthread_self ( ) To identify threads within a process, each thread is assigned a unique identifier. This is also known as the Thread Id. To get the identifier of a thread, this function is used.

pthread_equal (thread1, thread2) Compares two thread IDs: If the two IDs are different 0 is returned, otherwise a non-zero value is returned. **NOTE:** You cannot compare two threads with (==) operator.

Thread Initialization:

- Include the pthread.h library :

  #include <pthread.h>

- Declare a variable of type pthread_t :

  pthread_t the_thread

- When you compile, add -lpthread to the linker flags, here -lpthread in tells the GCC compiler that it must link the pthread library to the compiled executable:

  ```
  gcc thread.c -o thread -lpthread
  ```

  Here:

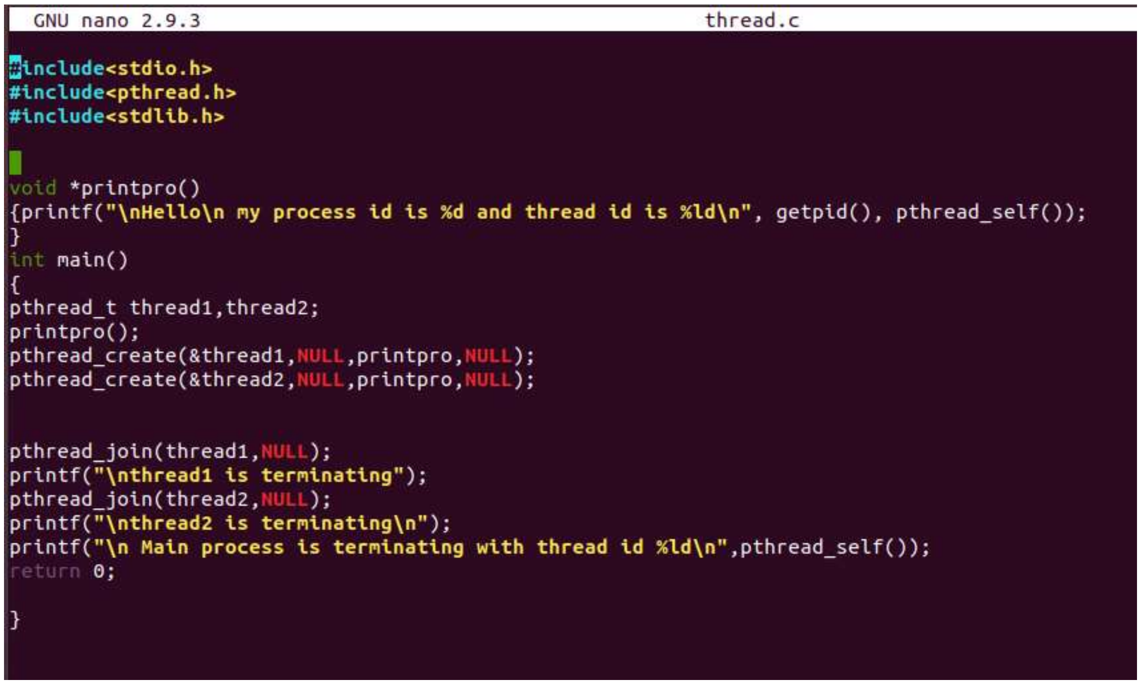
  - ➢ gcc is the compiler command.
  - ➢ thread.c is the name of c program source file.
  - ➢ -o is option to make object file.
  - ➢ thread is the name of object file.
  - ➢ -lpthread is option to execute pthread.h library file.

- Run it by using following command

  ```
  ./thread
  ```

EXAMPLES:

Example 1: Pthread creation and termination:



```
GNU nano 2.9.3                                          thread.c

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


void *printpro()
{printf("\nHello\n my process id is %d and thread id is %ld\n", getpid(), pthread_self());
}
int main()
{
pthread_t thread1,thread2;
printpro();
pthread_create(&thread1,NULL,printpro,NULL);
pthread_create(&thread2,NULL,printpro,NULL);


pthread_join(thread1,NULL);
printf("\nthread1 is terminating");
pthread_join(thread2,NULL);
printf("\nthread2 is terminating\n");
printf("\n Main process is terminating with thread id %ld\n",pthread_self());
return 0;

}
```

Example 2: Two Threads displaying two strings “Hello” and “How are you?” independent of each other

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

void * thread1()
{
        while(1){
                printf("Hello!!\n");
        }
}

void * thread2()
{
        while(1){
                printf("How are you?\n");
        }
}

int main()
{
        int status;
        pthread_t tid1,tid2;

        pthread_create(&tid1,NULL,thread1,NULL);
        pthread_create(&tid2,NULL,thread2,NULL);
        pthread_join(tid1,NULL);
        pthread_join(tid2,NULL);
        return 0;
}
```

Thread Attributes:

Threads have a number of attributes that may be set at creation time. This is done by filling athread attribute object attr of type pthread_attr_t, then passing it as second argument to pthread_create. Passing NULL is equivalent to passing a thread attribute object with all attributes set to their default values. Attribute objects are consulted only when creating a new thread. The same attribute object can be used for creating several threads. Modifying an attribute object after a call to pthread_create does not change the attributes of the thread previously created.

1. **int pthread_attr_init (pthread_attr_t *attr)**

pthread_attr_init initializes the thread attribute object attr and fills it with default values for the attributes. Each attribute attrname can be individually set using the function pthread_attr_setattrname and retrieved using the function pthread_attr_getattrname.

2. **int pthread_attr_destroy (pthread_attr_t *attr)**

pthread_attr_destroy destroys the attribute object pointed to by attr releasing any resources associated with it. attr is left in an undefined state, and you must not use it again in a call to any pthreads function until it has been reinitialized.

3. **int pthread_attr_setattr (pthread_attr_t *obj, int value)**

Set attribute attr to value in the attribute object pointed to by obj. See below for a list of possible attributes and the values they can take. On success, these functions return 0.

4. **int pthread_attr_getattr (const pthread_attr_t *obj, int *value)**

Store the current setting of attr in obj into the variable pointed to by value. These functions always return 0.

The following thread attributes are supported:

`detachstate'

Choose whether the thread is created in the joinable state (PTHREAD_CREATE_JOINABLE) or in the detached state (PTHREAD_CREATE_DETACHED).

The default is PTHREAD_CREATE_JOINABLE. In the joinable state, another thread can synchronize on the thread termination and recover its termination code using pthread_join, but some of the thread resources are kept allocated after the thread terminates, and reclaimed only when another thread performs pthread_join on that thread.

In the detached state, the thread resources are immediately freed when it terminates, but pthread_join cannot be used to synchronize on the thread termination. A thread created in the joinable state can later be put in the detached thread using pthread_detach.

`schedpolicy'

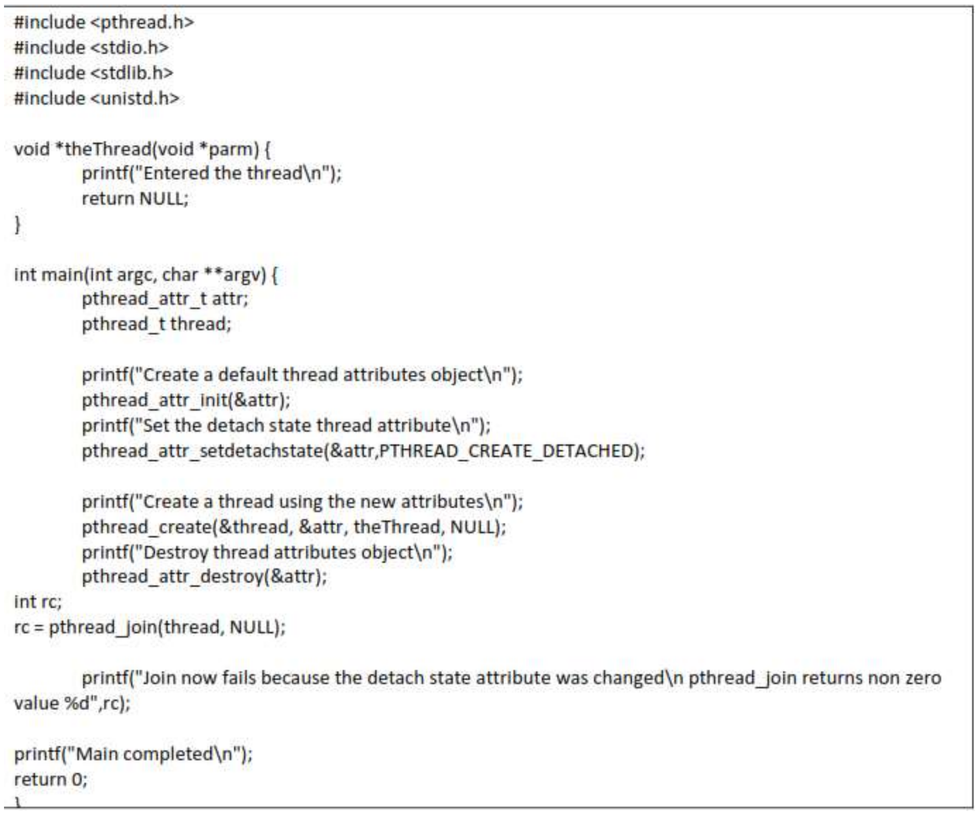
Select the scheduling policy for the thread: one of SCHED_OTHER (regular, non-realtime scheduling), SCHED_RR (realtime, round-robin) or SCHED_FIFO (realtime, first-in first-out). The default is SCHED_OTHER. The realtime scheduling policies SCHED_RR and SCHED_FIFO are available only to processes with superuser privileges. pthread_attr_setschedparam will fail and return ENOTSUP if you try to set a realtime policy when you are unprivileged. The scheduling policy of a thread can be changed after creation with pthread_setschedparam.

`schedparam'

Change the scheduling parameter (the scheduling priority) for the thread. The default is 0. This attribute is not significant if the scheduling policy is SCHED_OTHER; it only matters for the realtime policies SCHED_RR and SCHED_FIFO. The scheduling priority of a thread can be changed after creation with pthread_setschedparam.

Example:

We have used NULL in thread attr so far, now we will use thread attr in this example. Create a detached thread for a function infoThread()



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

void *theThread(void *parm) {
        printf("Entered the thread\n");
        return NULL;
}

int main(int argc, char **argv) {
        pthread_attr_t attr;
        pthread_t thread;

        printf("Create a default thread attributes object\n");
        pthread_attr_init(&attr);
        printf("Set the detach state thread attribute\n");
        pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);

        printf("Create a thread using the new attributes\n");
        pthread_create(&thread, &attr, theThread, NULL);
        printf("Destroy thread attributes object\n");
        pthread_attr_destroy(&attr);
int rc;
rc = pthread_join(thread, NULL);

        printf("Join now fails because the detach state attribute was changed\n pthread_join returns non zero
value %d",rc);

printf("Main completed\n");
return 0;
}
```

Synchronization in Multithreaded Programs:

When multiple threads access and manipulate a shared resource (ex: a variable for instance), the access to the shared resource needs to be controlled through a lock mechanism so that only one thread is allowed access to the shared resource at any point of time while the other threads are waiting to gain access the shared resource. In Posix thread (pthread) this is enforced by using a synchronization primitive called **mutex lock**.

The **mutex lock** is the simplest and most primitive synchronization variable. It provides a single, absolute owner for the section of code (aka a **critical section**) that it brackets between the calls to **pthread_mutex_lock()** and **pthread_mutex_unlock()**. The first thread that locks the mutex gets ownership, and any subsequent attempts to lock it will fail, causing the calling thread to go to sleep. When the owner unlocks it, one of the sleepers will be awakened, made runnable, and given the chance to obtain ownership.

EXAMPLE: Exploring mutex with multithreads sharing same resource:

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

volatile int counter = 0;
pthread_mutex_t myMutex;
int argc, char *argv[]
void *mutex_testing(void *param)
{
        int i;
        for(i = 0; i < 5; i++) {
                pthread_mutex_lock(&myMutex;);
                counter++;
                printf("thread %d counter = %d\n", (int)param,  counter);
                pthread_mutex_unlock(&myMutex;);
        }
}

int main()
{
        int one = 1, two = 2, three = 3;
        pthread_t thread1, thread2, thread3;
        pthread_mutex_init(&myMutex;,0);
        pthread_create(&thread1;, 0, mutex_testing, (void*)one);
        pthread_create(&thread2;, 0, mutex_testing, (void*)two);
        pthread_create(&thread3;, 0, mutex_testing, (void*)three);
        pthread_join(thread1, 0);
        pthread_join(thread2, 0);
        pthread_join(thread3, 0);
        pthread_mutex_destroy(&myMutex;);
        return 0;
}
```

Mutex Attributes:

Though mutexes, by default, are private to a process, they can be shared between multiple processes. To create a mutex that can be shared between processes, we need to set up the attributes for **pthread_mutex_init():**

```
#include <pthread.h>

int main()
{
        pthread_mutex_t myMutex;
        pthread_mutexattr_t myMutexAttr;
        pthread_mutexattr_init(&myMutexAttr;);
        pthread_mutexattr_setpshared(&myMutexAttr;, PTHREAD_PROCESS_SHARED);

        pthread_mutex_init(&myMutex;, &myMutexAttr;);
        //...

        pthread_mutexattr_destroy(&myMutexAttr;);
        pthread_mutex_destroy(&myMutex;);
        return 0;
}
```

pthread_mutexattr_setpshared() with a pointer to the attribute structure and the value **PTHREAD_PROCESS_SHARED** sets the attributes to cause a shared mutex to be created.

Mutexes are not shared between processes by default. Calling **pthread_mutexattr_setpshared()** with the value **PTHREAD_PROCESS_PRIVATE** restores the attribute to the default.

These attributes are passed into the call to **pthread_mutexattr_init()** to set the attributes of the initialized mutex. Once the attributes have been used, they can be disposed of by a call to **pthread_mutexattr_destroy().**

Lab Exercises:

1. Write a program that create 3 threads

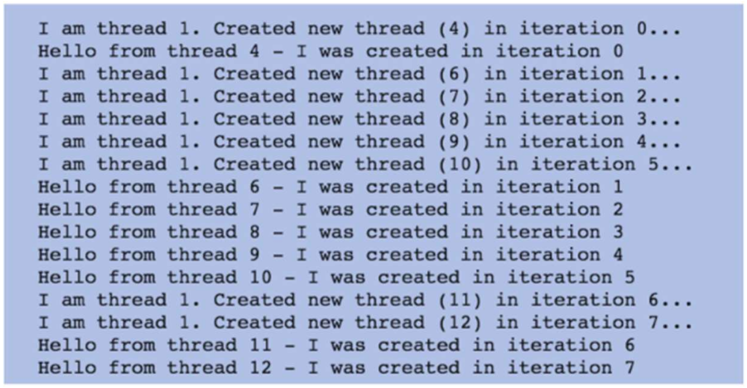
a) On successful creation, print “Thread #” in its starting routine and terminate themselves by

showing their return value.

b) On unsuccessful creation, Print “Error”.

2. Write a program, which make 4 threads. Each thread will print one table out of [5678] up to 1000.

3. Write a program that creates N number of threads specified in the command line, each prints “hello message and its own thread ID”. Sleep the main thread for 1 second and create every 4 or 5 threads. The output of your code should



```
I am thread 1. Created new thread (4) in iteration 0...
Hello from thread 4 - I was created in iteration 0
I am thread 1. Created new thread (6) in iteration 1...
I am thread 1. Created new thread (7) in iteration 2...
I am thread 1. Created new thread (8) in iteration 3...
I am thread 1. Created new thread (9) in iteration 4...
I am thread 1. Created new thread (10) in iteration 5...
Hello from thread 6 - I was created in iteration 1
Hello from thread 7 - I was created in iteration 2
Hello from thread 8 - I was created in iteration 3
Hello from thread 9 - I was created in iteration 4
Hello from thread 10 - I was created in iteration 5
I am thread 1. Created new thread (11) in iteration 6...
I am thread 1. Created new thread (12) in iteration 7...
Hello from thread 11 - I was created in iteration 6
Hello from thread 12 - I was created in iteration 7
```

alike:

4. Write a program to sum 10 elements of an array by multithreading.

5. Procom has 4 volunteers on their front desk.

• Volunteer 1 manages On day registration

• Volunteer 2 handles announcements

• Volunteer 3 handles sponsors

• Volunteer 4 resolve queries of participants

Implement this system using pthread for 10 participants.