Lab Tasks

By

Hajira Imran(44594)

Submitted to: Ma’am Kausar

Subject: Operating System

Date:11/29/2024

BSCS SEMESTER – 5

RIPHAH INTERNATIONAL UNIVERSITY

ISLAMABAD, PAKISTAN

Task 1:

Type the following and execute it.

Solution:

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

void *myThreadFun(void *vargp)
{
    sleep(1);
    printf("Printing GeeksQuiz from Thread \n");
    return NULL;
}

int main()
{
    pthread_t thread_id;
    printf("Before Thread\n");
    pthread_create(&thread_id, NULL, myThreadFun, NULL);
    pthread_join(thread_id, NULL);
    printf("After Thread\n");
    exit(0);
}
```

```
"task1.c" [New] 22L, 404B written
[root@localhost ~]# gcc task1.c -o task1 -pthread
[root@localhost ~]# ./task1
Before Thread
Printing GeeksQuiz from Thread
After Thread
[root@localhost ~]#
```

Code Explanation:

pthread_create:

This function creates a new thread that runs the function myThreadFun. The thread will sleep for 1 second and then print "Printing GeeksQuiz from Thread".

pthread_join:

This waits for the newly created thread to finish its execution before moving on with the main function.

Output:

The program first prints "Before Thread", then the thread runs and prints "Printing GeeksQuiz from Thread", and finally, the main function prints "After Thread".

Task 2:

Try to execute following code:

Solution:

```
void *myThreadFun(void *vargp)
{
    int myid = (int *)vargp;  // Store the value argument passed to this thread
    static int s = 0;         // Static variable to observe its changes

    // Change static and global variables
    ++s;
    ++g;

    // Print the argument, static, and global variables
    printf("Thread ID: %d, Static: %d, Global: %d\n", myid, ++s, ++g);
    return NULL;
}

int main()
{
    int i;
    pthread_t tid;

    // Create three threads
    for (i = 0; i < 3; i++)
    {
        pthread_create(&tid, NULL, myThreadFun, (void *)&tid);
    }

    pthread_exit(NULL);  // Wait for all threads to complete
    return 0;
}
```

This code creates three threads that each modify static and global variables and print their values:

- Global variable (g) is modified by all threads.
- Static variable (s) is local to the thread but retains its value between function calls within that thread.
- Each thread prints its unique ID along with the updated values of the global and static variables.

Task 3:

Try to execute following code:
Modify pthread_create and convert into for loop and create three threads and show output. Also removes pthread_exit and see what happens.

Solution:

Code explanation:

1. Header Files: The necessary libraries (stdio.h, stdlib.h, unistd.h, pthread.h) are included to handle input/output, memory allocation, threading, and other functions.
2. Worker Function: The workerThreadFunc prints a message identifying the thread's ID.
3. Main Function: A single thread is created using pthread_create and runs the workerThreadFunc.

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

// Worker thread function
void *workerThreadFunc(void *tid) {
    long *myID = (long *)tid;
    printf("HELLO WORLD! THIS IS THREAD %ld\n", *myID);
    return NULL;
}

int main() {
    pthread_t tids[3];  // Array to store thread IDs
    for (long i = 0; i < 3; i++) {
        pthread_create(&tids[i], NULL, workerThreadFunc, (void *)&i);
    }
    // Removed pthread_exit
    return 0;
}
```

Observation Without pthread_exit:

- Without pthread_exit, the main program may terminate before all threads complete, depending on how the OS schedules threads.
- With pthread_exit, the main thread ensures that all threads complete before the program exits.

Task 4:

Define POSIX thread and its working in your own words.

Solution:

POSIX threads (pthreads) are a standard API for creating and managing threads in C/C++. They enable parallel execution of tasks within a process, utilizing shared memory and efficient communication mechanisms.

Working of POSIX Threads:

A thread is created using pthread_create.
Each thread executes independently but shares the memory and resources of the parent process.
Synchronization is achieved using mutexes, semaphores, or condition variables to avoid race conditions.
The main thread can wait for child threads to finish using pthread_join.
