CL2006 - Operating Systems Spring 2024
LAB # 8 MANUAL (Common)

Please note that all labs' topics including pre-lab, in-lab and post-lab exercises are part of the theory and labsyllabus. These topics will be part of your Midterms and Final Exams of lab and theory.

Objectives:

1. **To understand and learn about implementing threads using the Pthread library already introduced in the OS theory classes.**

Lab Tasks:

1. Compile and run the code workouts to familiarize yourself with various aspects of the Pthread library.
2. Write simple multithreaded programs (In-Lab) to acquire skills to code using the Pthread library.

Delivery of Lab contents:

Strictly following the following content delivery strategy. Ask students to take notes during the lab.

1st Hour

- Compile and run example programs and related tasks.

2nd Hour

- Write multithreaded code for the two in-lab problems.

3rd Hour

- Not applicable in the holy month of Ramadan.

Created by: Hamza Yousuf, January 2019 FAST-NU, Lahore **Modified by: Nadeem Kafi (21/03/2024)**
DEPARTMENT OF COMPUTER SCEICEN, FAST-NU, KARACHI

** ChatGPT is heavily used to make the contents of this document along with other sources.

EXPERIMENT 8
Threads

Threads are an essential concept in computer science and programming, particularly in the context of multitasking and parallel computing. Some important aspects of threads are:

- **Provides** → Concurrency and parallelism: Threads allow multiple streams of execution within a single process. This means that different parts of a program can execute independently, potentially concurrently, improving the overall efficiency of the program by utilizing available CPU resources effectively. In certain applications, such as web servers or database servers, threads can be used to handle multiple client requests concurrently, improving scalability and throughput.

- **Benefits** → Responsiveness, Modularity, Resource sharing, Efficiency: Threads can be used to keep an application responsive to user input or external events while performing other tasks in the background. For example, a user interface can remain interactive even while performing complex computations in separate threads. Threads can be used to break down a complex task into smaller, more manageable units of execution, improving modularity and facilitating easier maintenance and development of software systems. Threads within the same process share resources such as memory and file descriptors, which can lead to efficient communication and data sharing between different parts of a program. While threads introduce overhead due to context switching and synchronization, they can still be more efficient than separate processes in many cases, particularly when sharing data and resources.

- **Requires** → Synchronization: Threads often need to coordinate their actions to ensure data consistency and avoid race conditions. Synchronization mechanisms such as mutexes, semaphores, and condition variables are used to control access to shared resources and coordinate the execution of threads.

Can we write multithreading programs in C?

Unlike Java, multithreading is not supported by the language standard. POSIX Threads (or Pthread) are POSIX standard for threads. Implementation of Pthread is available with gcc compiler.

Thread Library:

- POSIX Pthread
- Two general strategies for creating multiple threads.
  - Asynchronous threading:
    - Parent and child threads run independently of each other.
    - Typically, little data sharing between threads
  - Synchronous threading:
    - Parent thread waits for all of its children to terminate.
    - Children threads run concurrently.
    - Significant data sharing.
- Each thread has a set of attributes, including stack size and scheduling information.
- In a Pthread program, separate threads begin execution in a specified function.
  //runner ()
- When a program begins
  - A single thread of control begins in main ()
  - main() creates a second thread that begins control in the runner() function.
  - Both threads share the global data

Code workout # 1:

Design a multi-threaded program that performs the summation of a non-negative integer in aseparate thread using the summation function:

$$sum = \sum_{i=0}^{N} \mathrm{i}$$

For example, if N were 5, this function would represent the summation of integers from 0 to5, which is 15.

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

int sum = 0; // this data is shared by the thread(s)
void *runner(void *parameters)
{ // The thread will begin control in this funtion
    int i, upper = *((int *)parameters);
    if (upper > 0)
    {
        for (i = 1; i <= upper; i++)
            sum = sum + i;
    }
    pthread_exit(0);
}

int main(int argc, char *argv[])
{
    pthread_t threadID; // thread identifier
    pthread_attr_t attributes; // set attributes for the thread
    int num = 1000;

    pthread_attr_init(&attributes); // get the default attributes
    pthread_create(&threadID, &attributes, runner, (void *)&num); // create the thread
    pthread_join(threadID, NULL); // now wait for the thread to exit
    printf("sum=%d\n", sum);
    exit(0);
}
```

Note: use $gcc yourprogram.c -o yourprogram -lpthread (some older version use -pthread).

Observations and Discussion

a) Parameter passing from pthread_create call to runner function.
b) Pthread_attr_init() call sets different attributes of each thread. It will be covered in future labs.
c) Use the following code snippet to return values from runner function in pthread_join() call.

```c
void *thread_function(void *arg) {
    printf("Thread executing...\n");
    pthread_exit((void *)42); // Exiting thread with status 42
}

int main() {
    // Wait for the thread to finish and get its exit status
    pthread_join(thread, &exit_status);
    printf("Thread exited with status: %ld\n", (long)exit_status);

    return 0;
}
```

Code workout # 2:

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  static volatile int counter = 0;
5
6  void *mythread(void *arg) {
7      printf("%s: begin\n", (char *)arg);
8      int i;
9      //int counter = 0;
10     for (i = 0; i < 1e7; i++) {
11         counter = counter + 1;
12     }
13     printf("%s: done. Counter = %d\n", (char *)arg,counter);
14     return NULL;
15 }
16
17 int main(int argc, char *argv[]) {
18     pthread_t p1, p2;
19
20     printf("main: begin (counter = %d)\n", counter);
21     pthread_create(&p1, NULL, mythread, "A");
22     pthread_create(&p2, NULL, mythread, "B");
23
24     // join waits for the threads to finish
25     pthread_join(p1, NULL);
26     pthread_join(p2, NULL);
27
28     printf("main: done with both (counter = %d)\n", counter);
29     return 0;
30 }
```

a) Compile and run the code shown above.
b) Why both threads are calculating different values of counter? Discuss.
c) Uncomment the statement int counter = 0; at line # 9. Recompile and Re-run to review counter values. How statement at line 9 has affected the results you get in part (b) above?

In-Lab

Question 1:

Declare three float arrays A, B and C each of size 1e7 (10000000) and perform the operation C = A + B for each element of A, B and C.

a) Write, compile, and run serial code.
b) Write concurrent code where 10 worker threads will equally divide the computational workload.

Question 2:

Write a multithreaded program that calculates various statistical values for a list of numbers.This program will pass a series of numbers on the command line and will then create three separate worker threads. One thread will determine the average of the numbers, the second will determine the maximum value, and the third will determine the minimum value. For example, suppose your program is passed the integers. (The array of numbers must be passed as parameter to threads, and the thread must return the calculated value to

main thread).

90 81 78 95 79 72 85

The main thread will print:

The average value is 82.
The minimum value is 72.
The maximum value is 95.

DEMO multi-threading code (which sums an array using multiple threads)

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

#define NUM_THREADS 4
#define ARRAY_SIZE 1000000

int global_array[ARRAY_SIZE]; // Shared array

// Function to initialize the array with random values
void initialize_array() {
    for (int i = 0; i < ARRAY_SIZE; ++i) {
        global_array[i] = rand() % 1000;
    }
}

// Function to find the sum of elements in a portion of the array
void *sum_array(void *arg) {
    int thread_id = *((int *)arg);
    int start = thread_id * (ARRAY_SIZE / NUM_THREADS);
    int end = start + (ARRAY_SIZE / NUM_THREADS);
    int sum = 0;

    // Calculate the sum of elements in the assigned portion of the array
    for (int i = start; i < end; ++i) {
        sum += global_array[i];
    }

    return (void *)(long)sum; // Return the sum as a void pointer
}
int main() {
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];
    void *thread_results[NUM_THREADS];
    long total_sum = 0;

    // Initialize the array with random values
    initialize_array();

    // Create threads to compute the sum of array elements
    for (int i = 0; i < NUM_THREADS; ++i) {
        thread_args[i] = i;
        pthread_create(&threads[i], NULL, sum_array, (void *)&thread_args[i]);
    }

    // Join threads and collect results
    for (int i = 0; i < NUM_THREADS; ++i) {
        pthread_join(threads[i], &thread_results[i]);
        total_sum += (long)thread_results[i]; // Accumulate the partial sums
    }

    printf("Total sum of array elements: %ld\n", total_sum);

    return 0;
}
```