CO327 – Lab 03

E/19/166

Jayathunga W.W.K.

Exercise 1

1. a) The -pthread flag is required when compiling a program that uses the POSIX threads (pthread) library because it ensures that the compiler and linker handle the threading correctly.

    - **Including the pthread Library:**
      The -pthread flag tells the compiler to include the pthread library during both the compilation and linking stages. This is essential because the pthread library provides the necessary functions and definitions for threading operations. Without this flag, the compiler would not be able to find the implementations of the pthread functions such as pthread_create, pthread_join, etc.

    - **Thread-Safe Compilation:**
      The flag also instructs the compiler to define _REENTRANT, which ensures that the standard library functions used in the program are thread-safe. For example, certain standard C library functions have thread-safe versions that are enabled when _REENTRANT is defined. This is crucial for preventing race conditions and ensuring that the program behaves correctly in a multi-threaded environment.

    - **Linker Configuration:**
      When linking the program, the -pthread flag ensures that the pthread library is linked correctly. This includes linking against the correct version of the library and resolving any dependencies that the pthread library might have.

Exercise 2

1. a) Total Processes = 8
   Total New Processes = 7

   b) Total Threads = 4
   Total New Threads = 3

Exercise 3

1. a) Each thread prints its messages sequentially because the main thread waits for each thread to finish before creating the next one.
   The count variable is shared among all threads, and its value is incremented by each thread before it terminates.
   As a result, each thread prints its own number (derived from count) correctly.
   The main thread prints its message after all the other threads have completed their execution.

   b) **Argument Passing:**
   The thread function thread_function is called with an argument arg, which is a pointer to the count variable in the main function. The type of arg is void*, a generic pointer type.

   Type Casting:
   The expression *(int *)arg casts arg to an int* (pointer to int) and then dereferences it to get the integer value stored at that address. This effectively accesses the value of count.

   Incrementing the Value:
   The expression (*(int *)arg)++ increments the integer value pointed to by arg. This means it increases the value of count by 1.

   c) **Unique Thread Identifiers:**

   By incrementing the value of count within each thread function, each thread is assigned a unique identifier. This identifier is used in the print statement (printf("Thread %d:%d says hi!\n", *(int *)arg, a);) to display the thread number along with the message.

   Sequential Update:

   Since each thread receives a pointer to the same count variable, but they are created and joined sequentially in the main loop, the increment operation ((*(int *)arg)++;) ensures that each thread updates count sequentially. This means that each thread gets a unique identifier based on the order in which it was created and joined.

2. **Concurrency without Synchronization:**
   Without the join call, the main thread will continue executing the loop without waiting for the created threads to finish. This means that multiple threads will be created and started simultaneously, leading to concurrent execution of multiple instances of the thread_function.

   Simultaneous Execution of Threads:
   Since multiple threads are running concurrently, each thread will access and modify the shared count variable simultaneously. This can lead to race conditions, where the behavior of the program becomes unpredictable due to inconsistent or unexpected interleavings of thread execution.

   Output Variation:

The output of the program will vary each time it is run because the order of execution of threads and their access to the shared count variable will differ. Some threads might increment count before others have finished printing their messages, leading to interleaved or overlapping output.

Possible Data Corruption:
Concurrent access to shared data without proper synchronization mechanisms like mutexes or thread barriers can result in data corruption or inconsistent state. In this case, multiple threads attempting to modify count simultaneously could lead to incorrect or unexpected values.

a) **Commenting out the ‘sleep()’ statement at line 12:**
With this sleep() statement commented out, each thread will execute its loop (for loop from lines 9 to 13) without any delay. As a result, the threads will rapidly print their messages without pausing between iterations. This will likely lead to fast and continuous printing of messages by each thread, potentially making the output difficult to read as messages may be printed very quickly.

Commenting out the sleep() statement at line 35:

With this sleep() statement commented out, the main thread will not pause after creating and joining each thread in the loop. As a result, the main thread will rapidly create and join threads without any delay between iterations of the loop. This may lead to a situation where threads are created and joined very quickly, possibly causing the program to consume CPU resources excessively and making the behavior of the program less predictable.

Commenting out the sleep() statement at line 37:

With this sleep() statement commented out, the main thread will not pause before printing its message. As a result, the message "Main thread says hi!" will be printed immediately after the loop finishes executing. Without the sleep, the message may be printed immediately after the last thread is joined, potentially causing it to be printed before some of the thread messages, depending on the scheduling of threads by the operating system.

b) I) With both sleep() statements commented out, the main thread will not pause either before or after the loop where threads are created and joined.
This means that after the loop finishes executing, the main thread immediately proceeds to print its message.
However, since the main thread does not pause before entering the loop, it will start creating and joining threads rapidly.
This can lead to the main thread printing its message before all the threads have finished executing their respective loops and printing their messages.
Depending on the scheduling of threads by the operating system, the output may vary, and the message from the main thread may appear before, after, or intermixed with the messages from the threads.

ii) **Effect on Thread Execution:**

Without the sleep() statement at line 12, threads will execute their loop (for loop from lines 9 to 13) without any delay between iterations.

Without the sleep() statement at line 35, the main thread will not pause after creating each thread in the loop. It will rapidly create and join threads without any delay between iterations of the loop.

Changes in Results:

The rapid execution of threads and the main thread without any pauses may lead to unpredictable behavior, especially in terms of the relative ordering of messages printed by threads and the main thread.

Since there are no delays introduced between iterations of the loop in the main thread, the program may consume CPU resources excessively and complete execution quickly.

Variability in Results:

The specific behavior and output of the program may vary from run to run, depending on factors such as the scheduling of threads by the operating system and the timing of thread creation and termination relative to the main thread's execution.

iii) **Without Delay in Thread Execution (Line 12 Commented Out):**

With the sleep() statement at line 12 commented out, threads execute their loop rapidly without any delay between iterations.

This results in fast and continuous printing of messages by each thread, potentially overwhelming the output.

Gradually Increasing Delay in Main Thread Execution (Line 37):

Initially, with the sleep() statement at line 37 commented out, the main thread will print its message immediately after the loop without any delay.

As we gradually increase the sleep time of line 37 to 5 seconds, the main thread will start delaying its message printing by 5 seconds after the loop completes.

This delay introduces a pause before the main thread's message is printed, allowing the messages from the threads to be displayed before the main thread's message.

c) i) **Sleeping Threads (Line 12):**

With the sleep statement at line 12 uncommented, each thread will pause for 1 second in each iteration of its loop (for loop from lines 9 to 13).

This introduces a delay between each message printed by the threads, making the output more readable and allowing for easier interpretation of each thread's activity.

Main Thread Pause after Thread Creation and Joining (Lines 35 and 37):

With the sleep statements at lines 35 and 37 uncommented, the main thread will pause for 1 second after creating and joining each thread in the loop.

This introduces a delay between each iteration of the loop in the main thread, ensuring that threads are created and joined sequentially, one after the other.
The pause also ensures that the main thread's message ("Main thread says hi!") is printed after all threads have completed their execution, providing a clear indication of the end of the program's execution.

ii) **Effect of Sleep Value in Line 37:**
When the sleep value in line 37 is set to 1 second, the main thread will pause for 1 second after creating and joining each thread in the loop.
This pause ensures that threads are created and joined sequentially, with a slight delay between each iteration of the loop in the main thread.

Impact on Thread Execution:
The sleep value in line 37 does not affect the execution of the threads themselves. They will still execute their loops with a 1-second pause between each message, as determined by the sleep value in line 12.

Overall Program Behavior:
The overall behavior of the program, including the sequence of thread execution and the printing of messages, remains unchanged regardless of the sleep value in line 37.
The main difference lies in the timing of the main thread's message relative to the thread messages. With a sleep value of 1 second in line 37, the main thread's message may be printed slightly earlier or later compared to when the sleep value is 5 seconds.

3. a) **Misuse of sleep() vs. join():**
The primary purpose of sleep() is to introduce a delay or pause in the execution of a thread, while join() is used to synchronize the execution of multiple threads, ensuring that a thread waits for another thread to finish before continuing. Using sleep() instead of join() to achieve synchronization is not appropriate and can lead to inefficient and unpredictable behavior in multithreaded programs.

Lack of Synchronization:
Using sleep() statements instead of join() calls does not provide proper synchronization between threads. Without synchronization, threads may execute concurrently and independently, leading to race conditions, data corruption, or unpredictable program behavior.

Inefficient Resource Utilization:
Using sleep() statements to delay the execution of threads introduces unnecessary waiting periods, during which CPU resources may remain idle. This can result in inefficient resource utilization, especially in scenarios where threads could be actively performing useful work instead of being artificially delayed.

Debugging and Maintainability:

Relying on sleep() statements for synchronization can make the code harder to understand, debug, and maintain. Join calls explicitly communicate the intention of waiting for a thread to complete, making the code more readable and self-explanatory.

Potential Race Conditions:
Sleep() statements introduce fixed delays, which may not accurately reflect the actual execution time of threads. Depending on system load and other factors, threads may complete their execution before the sleep duration elapses, leading to race conditions or unexpected program behavior.

Preferences:

Prefer Join() for Synchronization:
I prefer using join() calls for synchronization in multithreaded programs. Join() provides explicit synchronization between threads, ensuring orderly execution and proper resource management.

Use Sleep() Judiciously:
While sleep() statements have their use cases, such as introducing delays for testing or simulation purposes, they should not be used as a substitute for join() calls in multithreaded programming. Sleep() should be used judiciously and only when necessary, such as for implementing timeouts or periodic tasks.

Follow Best Practices:
It's essential to follow best practices in multithreaded programming, including proper synchronization mechanisms, error handling, and resource management. Using join() calls where synchronization is needed helps maintain code clarity, reliability, and scalability.

Exercise 4

1.
```
#include <unistd.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>

void* handle_client(void*);

int main() {
  int listenfd;
  struct sockaddr_in servaddr, cliaddr;
  socklen_t clilen;
```

```
    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(32000);
    bind(listenfd, (struct sockaddr*)&servaddr, sizeof(servaddr));
    listen(listenfd, 5);
    clilen = sizeof(cliaddr);
    while (1) {
        int* connfd = malloc(sizeof(int));
        *connfd = accept(listenfd, (struct sockaddr*)&cliaddr, &clilen);
        pthread_t tid;
        pthread_create(&tid, NULL, handle_client, (void*)connfd);
    }
}

void* handle_client(void* connfd) {
    int client_socket = *((int*)connfd);
    free(connfd);

    char buffer[1024];
    int bytes_received = recv(client_socket, buffer, sizeof(buffer), 0);
    if (bytes_received < 0) {
        perror("Error reading from client");
        close(client_socket);
        pthread_exit(NULL);
    }

    printf("Received from client: %s\n", buffer);

    const char* response = "Hello from the server";
    int bytes_sent = send(client_socket, response, strlen(response), 0);
    if (bytes_sent < 0) {
        perror("Error sending response to client");
    }

    close(client_socket);
    pthread_exit(NULL);
}
```

2. <u>Why Declare connfd on the Heap:</u>

Thread Safety:

In a multi-threaded environment, each thread has its own execution context and stack. If connfd were declared as a local variable within the while loop, each thread would reference the same memory location on the stack. This would lead to data race conditions and potentially incorrect behavior as multiple threads simultaneously access and modify the same memory location.

Lifetime Management:
Declaring connfd on the heap ensures that the memory associated with it remains valid until it is explicitly deallocated using free(). This allows each thread to safely access and use its own copy of the client's socket file descriptor throughout its execution.

Avoiding Stack Overflow:
If connfd were declared as a local variable on the stack within the while loop, and if a large number of threads were created, it could lead to stack overflow due to the limited stack size allocated to each thread. Heap memory allocation avoids this limitation, as the heap typically has a larger memory space available compared to the stack.

Potential Problems with Local Variable Declaration:

Data Races:
If connfd were declared as a local variable within the while loop, multiple threads would share the same memory location on the stack. This could result in data races, where concurrent access to the shared memory leads to unpredictable behavior, race conditions, or memory corruption.

Dangling Pointers:
If connfd were declared as a local variable and passed to the thread function as a pointer, the memory associated with it would be deallocated when the loop iteration ends. This would result in dangling pointers in the thread function, as the memory location pointed to by connfd would no longer be valid.