

# CO327 LAB 03

KANEWALA U.C.H.

E/13/175

SEMESTER 6

03/09/2017

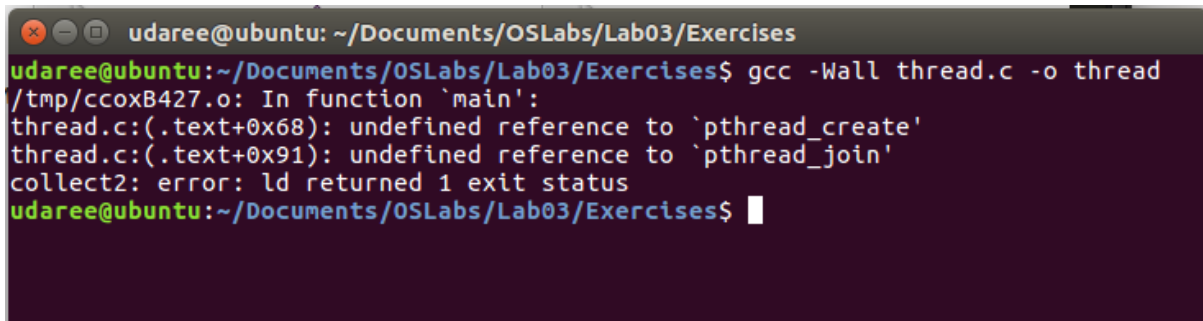
## Introduction to Multithreading

### Exercise1

1. Save the above program as thread.c and compile it as:  
`gcc -pthread -o thread thread.c -Wall`

a) Why do you need -pthread flag when you compile?

Without the -pthread flag the following errors can be seen.



```
udaree@ubuntu: ~/Documents/OSLabs/Lab03/Exercises
udaree@ubuntu:~/Documents/OSLabs/Lab03/Exercises$ gcc -Wall thread.c -o thread
/tmp/ccoxB427.o: In function `main':
thread.c:(.text+0x68): undefined reference to `pthread_create'
thread.c:(.text+0x91): undefined reference to `pthread_join'
collect2: error: ld returned 1 exit status
udaree@ubuntu:~/Documents/OSLabs/Lab03/Exercises$
```

-pthread flag adds support for multithreading with the pthreads library. This option sets flags for both the preprocessor and linker allowing to link the library to code.

### Exercise2

1. Consider the following piece of code from multiprocessing lab session:

```
int i;
for (i=0;i<3; i++)
    fork();
```

a) how many newprocesses did it create?

7 new processes

Now consider this piece of code:

```
int i;
for(i=0;i<3; i++)
    pthread_create(&myThread,NULL,function,NULL);
```

b) how many threads will it create? Compare this with a) above. Why is there a difference?

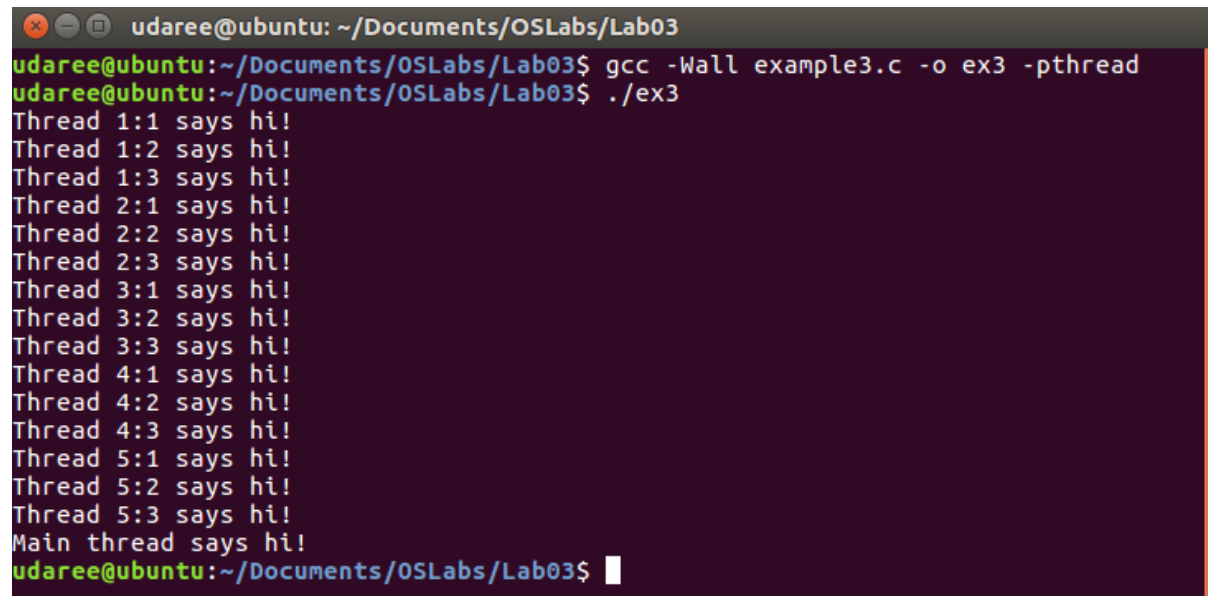
3 threads

When using the fork() command, it creates a new process (child) which runs the same copy of the current running program while having the main process (parent) running as it is. Hence, the two processes will run the same program independently afterwards. Invoking fork() command again will create new processes again on both the previous parent and child. Hence,

$2^3 - 1$  number of new processes will be created in above case when using fork(). In thread creation, only a single thread is created which runs as an independent new process after creation. Hence only 3 threads are created in this case.

### Exercise 3

1. Compile and run the above program as is.
  - a) Can you explain the results?



```
udaree@ubuntu: ~/Documents/OSLabs/Lab03
udaree@ubuntu:~/Documents/OSLabs/Lab03$ gcc -Wall example3.c -o ex3 -pthread
udaree@ubuntu:~/Documents/OSLabs/Lab03$ ./ex3
Thread 1:1 says hi!
Thread 1:2 says hi!
Thread 1:3 says hi!
Thread 2:1 says hi!
Thread 2:2 says hi!
Thread 2:3 says hi!
Thread 3:1 says hi!
Thread 3:2 says hi!
Thread 3:3 says hi!
Thread 4:1 says hi!
Thread 4:2 says hi!
Thread 4:3 says hi!
Thread 5:1 says hi!
Thread 5:2 says hi!
Thread 5:3 says hi!
Main thread says hi!
udaree@ubuntu:~/Documents/OSLabs/Lab03$
```

In the main thread, 5 threads are created. But they are not working concurrently as the `pthread_join()` function is called within the for loop where threads are generated. Hence, all five threads are running in order of the order they are created. Each thread after creation executes the `thread_function()`. The thread function takes the argument count and iterates 3 times and prints the count along with the iteration round and increments the count argument. Hence, for the first thread, in the output, the same count (1) is seen whereas the iteration number increments from 1 to 3. Then the count is incremented to 2 and the 1st thread finishes the `thread_function()` execution. Then the second thread takes the incremented count(2) and continues the same procedure. The main thread waits until all threads finish execution and exits at the end.

- b) What is the objective of the code in line 15?

Incrementing the value of the argument which is a pointer to the count variable. In order to do this, as the pointer is of type void it has to be converted to an int pointer as count variable is an integer. Then in order to increment the count variable, the value of the pointer is obtained and incremented.

- c) Deconstruct this statement, and explain how the objective you mentioned is achieved.

`void * arg` → pointer of any type(generic)

(int \*) arg → converting the generic pointer arg into an integer pointer type

\*(int\*)arg → get the value stored at the memory location pointed by the pointer arg.

**2. Comment out the code segment for the join call (lines 30 – 34). Can you explain the result?**

```
udaree@ubuntu: ~/Documents/OSLabs/Lab03
udaree@ubuntu:~/Documents/OSLabs/Lab03$ gcc -Wall example3.c -o ex3 -pthread
udaree@ubuntu:~/Documents/OSLabs/Lab03$ ./ex3
Main thread says hi!
udaree@ubuntu:~/Documents/OSLabs/Lab03$
```

The main thread exists before the execution of the threads created by the main thread. Now the main thread is not waiting till the created threads join. hence, no outputs are seen from the thread\_function().

**a) Now, comment out the sleep() statements at lines 12, 35 and 37 (only one at a time) and explain each result.**

Comment line 12:

```
udaree@ubuntu: ~/Documents/OSLabs/Lab03
udaree@ubuntu:~/Documents/OSLabs/Lab03$ gcc -Wall example3.c -o ex3 -pthread
udaree@ubuntu:~/Documents/OSLabs/Lab03$ ./ex3
Thread 1:1 says hi!
Thread 1:2 says hi!
Thread 1:3 says hi!
Thread 2:1 says hi!
Thread 2:2 says hi!
Thread 2:3 says hi!
Thread 3:1 says hi!
Thread 3:2 says hi!
Thread 3:3 says hi!
Thread 4:1 says hi!
Thread 4:2 says hi!
Thread 4:3 says hi!
Thread 5:1 says hi!
Thread 5:2 says hi!
Thread 5:3 says hi!
Main thread says hi!
udaree@ubuntu:~/Documents/OSLabs/Lab03$
```

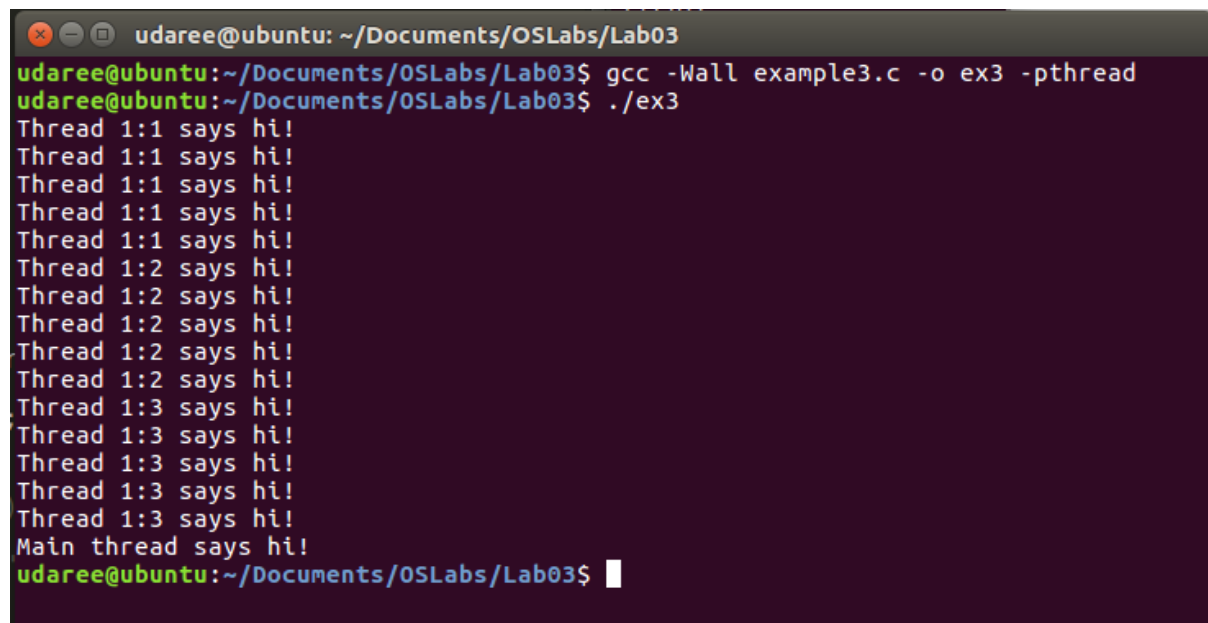
This gives a similar result to the instance seen when lines 30-40 was uncommented with a slight difference in the speed of printing the output. That is all outputs are not printed as it is run. The output is visible in print order with groups of three where each is coming from one thread and five such groups are seen in order for the 5 threads. After all five sets of output, the main thread's output is seen separately. The threads are working in a sequential manner in the output seen.

Here the outputs of the thread\_function() can be clearly seen for each thread separately due to the sleep time included in the for loop(line 35) where the threads are created. Also, the

main thread exits after all threads finish execution after waiting for another 5 ms time due to line 37. Hence can separately observe each threads functionality in this case.

The sleep time included in the for loop (line 35) where the 5 threads are created, makes the threads to function in the created order by finishing the task of the thread\_function() including the running of 3 iterations before the creation of the next thread. That is the thread\_function() in total takes less than 1 ms to finish execution. When line 12 is uncommented, the thread\_function() takes more than 1ms to finish its task as it sleeps for 1 ms 3 times resulting a total time of 3 ms sleep time. In that case, the sequential increment of the count variable cannot be observed as in this case.

Comment line 35:

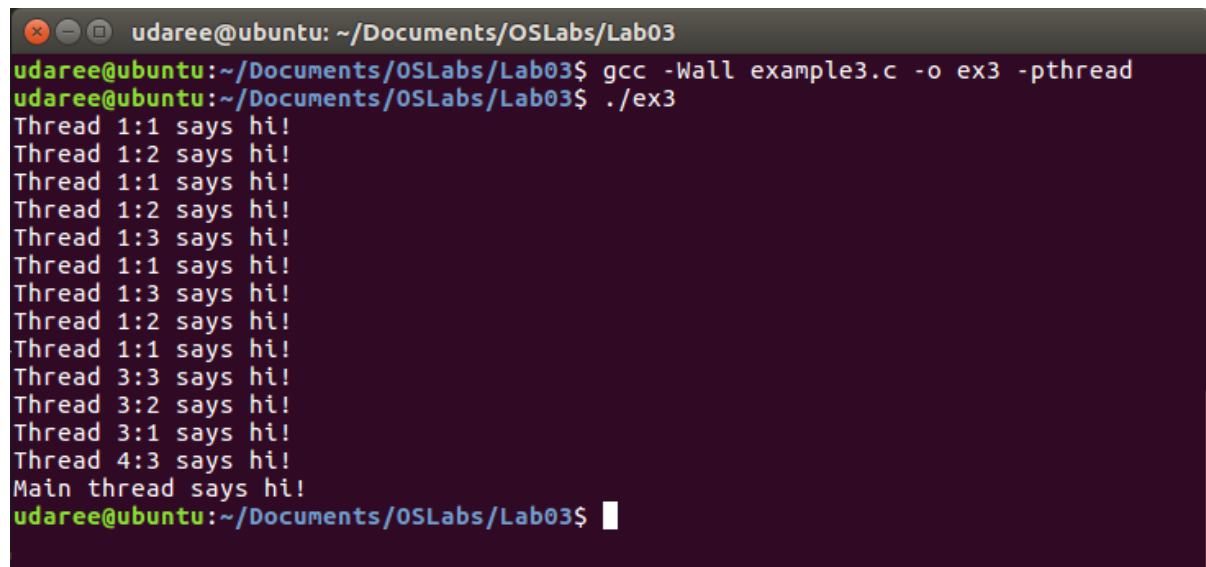


```
udaree@ubuntu: ~/Documents/OSLabs/Lab03
udaree@ubuntu:~/Documents/OSLabs/Lab03$ gcc -Wall example3.c -o ex3 -pthread
udaree@ubuntu:~/Documents/OSLabs/Lab03$ ./ex3
Thread 1:1 says hi!
Thread 1:1 says hi!
Thread 1:1 says hi!
Thread 1:1 says hi!
Thread 1:1 says hi!
Thread 1:2 says hi!
Thread 1:2 says hi!
Thread 1:2 says hi!
Thread 1:2 says hi!
Thread 1:2 says hi!
Thread 1:3 says hi!
Thread 1:3 says hi!
Thread 1:3 says hi!
Thread 1:3 says hi!
Thread 1:3 says hi!
Main thread says hi!
udaree@ubuntu:~/Documents/OSLabs/Lab03$
```

The output is visible in lines printing in groups of five and 3 such groups are printed in total where the count variable or thread number corresponds to 1 in all cases. A time gap is seen between two such groups. In each group, the iteration number is the same in each printed line in such a way that the first group, the second and the third denotes 1, 2 and 3 respectively. The output shows a behaviour where the threads are working in a concurrent manner.

The result depicts that creating all 5 threads takes less than 1 ms time. Hence, all five threads will get the initial count (count = 1) as its argument for thread\_function() during its creation. In the output, all threads are in the same iteration in the thread\_function() for a specific time as the increment happens only after a delay of 1ms due to line 12 in the thread\_function. Hence, all five threads will print the message for iteration 1 first, next iteration 2 and iteration 3 at last. This sleep time in the thread\_function() does not give any problems in this order as all five threads start execution nearly at the same time. Also, the total time for execution of all threads will be little more than 3ms due to line 12. The main thread will anyway be running until all 5 threads finish execution as the time taken for this is less than 5 ms time which is the time given for the main thread to sleep after thread creation(line 37).

### Comment 37:



```
udaree@ubuntu: ~/Documents/OSLabs/Lab03
udaree@ubuntu:~/Documents/OSLabs/Lab03$ gcc -Wall example3.c -o ex3 -pthread
udaree@ubuntu:~/Documents/OSLabs/Lab03$ ./ex3
Thread 1:1 says hi!
Thread 1:2 says hi!
Thread 1:1 says hi!
Thread 1:2 says hi!
Thread 1:3 says hi!
Thread 1:1 says hi!
Thread 1:3 says hi!
Thread 1:2 says hi!
Thread 1:1 says hi!
Thread 3:3 says hi!
Thread 3:2 says hi!
Thread 3:1 says hi!
Thread 4:3 says hi!
Main thread says hi!
udaree@ubuntu:~/Documents/OSLabs/Lab03$
```

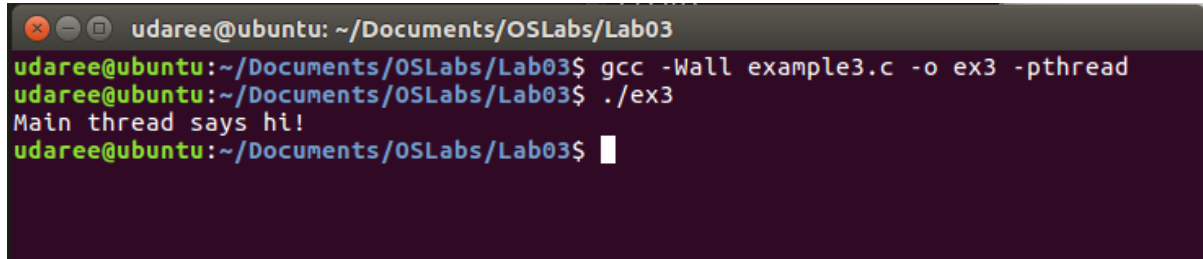
The main thread exits after 13 output print lines. The output print line of the main thread appears at the end or one before the last printed line when the program is run at different times. Hence, the output is not the same always. In the output, 8 lines denote the same count variable = 1. Single line after that shows count = 2 and last 4 lines show count variable = 3. Hence, an unexpected behaviour is seen in incrementing the count variable in this case.

The main thread exits prior to the other threads created as the main thread sleep only for a total of 5 ms which is generated by the line 35 in the for loop creating the 5 threads. This is also between the creation of each thread. Hence each thread will be created 1ms later than the previous one. Also in each thread, a total of 3 ms sleep time is included which will sum up and pass the 5 ms sleep time in the main thread at some point. Only the first 3 threads created are able to finish execution of the thread\_function completely.

The first 8 lines show the same count value as the 1<sup>st</sup> thread increments the count variable only after 3 ms time due to the sleep time enforced by line 12. During this 3ms time, the main thread is able to create 3 threads in whole while spending 1 ms sleep time in between each creation. When the 1<sup>st</sup> thread manages to increment the count, after another 1 ms the 2<sup>nd</sup> thread increments the count again to 3. Hence, only one line will show the count value = 2 in the output which is the 4<sup>th</sup> thread created. Hence, 3<sup>rd</sup>, 4<sup>th</sup> and 5<sup>th</sup> threads will get the count value as 3 afterwards. The 4<sup>th</sup> and 5<sup>th</sup> threads run only 2 iterations as the main thread gives its output line and exits after 5ms time as explained above.

a) Now, comment out pairs of `sleep()` statements as follows, and explain the results.

i. lines 35 & 37



```
udaree@ubuntu: ~/Documents/OSLabs/Lab03
udaree@ubuntu:~/Documents/OSLabs/Lab03$ gcc -Wall example3.c -o ex3 -pthread
udaree@ubuntu:~/Documents/OSLabs/Lab03$ ./ex3
Main thread says hi!
udaree@ubuntu:~/Documents/OSLabs/Lab03$
```

The output shows the main threads print line first followed by a single iteration of a created thread running the thread function. But this is not the same always when the program is run several times. Sometimes only the main thread's output line is printed while some times 4 output lines from the `thread_function()` is seen before the output line of the main function. Hence, the output varies at times.

This is due to the fact that the main thread exits just after creating five threads. The time taken to exit the main thread using `exit(0)` can vary in milliseconds due to different overheads. Hence, if the main thread remains for some time the outputs of the `thread_function()` can also be seen where at times none is seen.

ii. lines 12 & 35 (run the program multiple times, and explain changes in the results, if any)

When running several times different outputs are seen from the `thread_function` executed by the 5 threads created. One output shows the perfect sequential order of execution by all threads with both incrementing the count variable and iteration number similar to what is seen when join function was there. Another output shows all 5 threads having count value 1 at the start and showing different values in the next iterations depending on how the output of threads are directed to the terminal by the CPU scheduling. This case shows complete unexpected results as no manual scheduling is done by introducing sleep statements in the main program handling creation and execution of threads. However, the main thread exits after 5 ms time after creating the threads. All threads finish execution as they are working concurrently.

```

udaree@ubuntu:~/Documents/OSLabs/Lab03$ gcc -Wall example3.c -o ex3 -pthread
udaree@ubuntu:~/Documents/OSLabs/Lab03$ ./ex3
Thread 1:1 says hi!
Thread 1:2 says hi!
Thread 1:3 says hi!
Thread 1:1 says hi!
Thread 2:2 says hi!
Thread 2:3 says hi!
Thread 1:1 says hi!
Thread 3:2 says hi!
Thread 3:3 says hi!
Thread 1:1 says hi!
Thread 4:2 says hi!
Thread 4:3 says hi!
Thread 1:1 says hi!
Thread 5:2 says hi!
Thread 5:3 says hi!
Main thread says hi!
udaree@ubuntu:~/Documents/OSLabs/Lab03$ ./ex3
Thread 1:1 says hi!
Thread 1:2 says hi!
Thread 1:3 says hi!
Thread 1:1 says hi!
Thread 2:2 says hi!
Thread 2:3 says hi!
Thread 1:1 says hi!
Thread 3:2 says hi!
Thread 3:3 says hi!
Thread 4:1 says hi!
Thread 4:2 says hi!
Thread 4:3 says hi!
Thread 1:1 says hi!
Thread 5:2 says hi!
Thread 5:3 says hi!
Main thread says hi!
udaree@ubuntu:~/Documents/OSLabs/Lab03$ ./ex3
Thread 1:1 says hi!
Thread 1:2 says hi!
Thread 1:3 says hi!
Thread 2:1 says hi!
Thread 2:2 says hi!
Thread 2:3 says hi!
Thread 3:1 says hi!
Thread 3:2 says hi!
Thread 3:3 says hi!
Thread 4:1 says hi!
Thread 4:2 says hi!
Thread 4:3 says hi!
Thread 5:1 says hi!
Thread 5:2 says hi!
Thread 5:3 says hi!
Main thread says hi!
udaree@ubuntu:~/Documents/OSLabs/Lab03$

```

- iii. lines 12 & 37 (increase the sleep time of line 37 gradually to 5)



```
udaree@ubuntu: ~/Documents/OSLabs/Lab03
udaree@ubuntu:~/Documents/OSLabs/Lab03$ gcc -Wall example3.c -o ex3 -pthread
udaree@ubuntu:~/Documents/OSLabs/Lab03$ ./ex3
Thread 1:1 says hi!
Thread 1:2 says hi!
Thread 1:3 says hi!
Thread 2:1 says hi!
Thread 2:2 says hi!
Thread 2:3 says hi!
Thread 3:1 says hi!
Thread 3:2 says hi!
Thread 3:3 says hi!
Thread 4:1 says hi!
Thread 4:2 says hi!
Thread 4:3 says hi!
Thread 5:1 says hi!
Thread 5:2 says hi!
Thread 5:3 says hi!
Main thread says hi!
udaree@ubuntu:~/Documents/OSLabs/Lab03$
```

c) Now, uncomment all sleep() statements.

i. Can you explain the results?

```
udaree@ubuntu: ~/Documents/OSLabs/Lab03
udaree@ubuntu:~/Documents/OSLabs/Lab03$ gcc -Wall example3.c -o ex3 -pthread
udaree@ubuntu:~/Documents/OSLabs/Lab03$ ./ex3
Thread 1:1 says hi!
Thread 1:2 says hi!
Thread 1:1 says hi!
Thread 1:2 says hi!
Thread 1:3 says hi!
Thread 1:1 says hi!
Thread 1:3 says hi!
Thread 1:2 says hi!
Thread 2:1 says hi!
Thread 3:3 says hi!
Thread 3:2 says hi!
Thread 3:1 says hi!
Thread 3:2 says hi!
Thread 4:3 says hi!
Thread 4:3 says hi!
Main thread says hi!
udaree@ubuntu:~/Documents/OSLabs/Lab03$
```

This shows the same output as seen in above part a) with commenting line 37. But, total output in that manner can be seen as the main thread is not existing in the middle. The main threads wait another 5ms second time after spending 5 ms time for the creation of threads. Hence, the main thread exits after a total time of 10ms after the start. Hence, all 5 threads are able to finish executing thread\_function(). But as all threads are working concurrently after creation the count variable does not indicate the thread number in this case as it is incremented by each thread without synchronization. The same thread shows different count values in different iterations.

The correct count is received by only the first thread and displays the same count in all 3 iterations as other threads have not updated the count value during its execution due to the

sleep time is given in the `thread_function()` and the main function. But, other threads each displays incorrect count values within its own 3 iterations as all threads are updating the count variable concurrently. Different count values are seen within the same thread as there is a sleep time included in the printing for loop. Basically, a race condition occurs here and you cannot identify the threads as above uniquely using the count variable.

- ii. Does the output change if you revert the sleep value in line 37 back to 1? If so, why?

```
udaree@ubuntu:~/Documents/OSLabs/Lab03$ gcc -Wall example3.c -o ex3 -pthread
udaree@ubuntu:~/Documents/OSLabs/Lab03$ ./ex3
Thread 1:1 says hi!
Thread 1:2 says hi!
Thread 1:1 says hi!
Thread 1:3 says hi!
Thread 1:2 says hi!
Thread 1:1 says hi!
Thread 2:3 says hi!
Thread 2:2 says hi!
Thread 2:1 says hi!
Thread 3:3 says hi!
Thread 3:2 says hi!
Thread 3:1 says hi!
Thread 4:2 says hi!
Thread 4:3 says hi!
Thread 4:3 says hi!
Main thread says hi!
udaree@ubuntu:~/Documents/OSLabs/Lab03$ ./ex3
Thread 1:1 says hi!
Thread 1:2 says hi!
Thread 1:1 says hi!
Thread 1:2 says hi!
Thread 1:1 says hi!
Thread 1:3 says hi!
Thread 1:3 says hi!
Thread 1:2 says hi!
Thread 2:1 says hi!
Thread 3:3 says hi!
Thread 3:2 says hi!
Thread 3:1 says hi!
Thread 4:3 says hi!
Thread 4:2 says hi!
Main thread says hi!
udaree@ubuntu:~/Documents/OSLabs/Lab03$
```

Yes. Because now the main thread exits after 1 ms sleep time after all threads are created, the entire output above cannot be seen. Only 14 lines from the output of the `thread_function()` can be seen.

3. Consider the following statement: “you use `sleep()` statements instead of `join` calls to get the desired output of a multithreaded program.”

- a) Write a short critique of this statement expressing your views and preferences, if any.

Using `sleep()` statements instead of `join()` calls can be enough to get desired outputs of a multithreaded program only if the program is less complicated and consisting of very small executions. Otherwise, it can be very difficult to do the same for a complex program with high

execution overheads. Also, with the involvement of CPU scheduling, the output of the program becomes unpredictable in some cases. Hence, it is not a solution in such scenarios.

Even if the requirement is satisfied by using `sleep()` statements, there are some disadvantages associated with it. The main fact is that the time is given for `sleep()` is a constant value. Even if the execution intended is over, the processor will be waiting while wasting CPU cycles until the given time is elapsed. In contrast, the `join()` calls will execute other threads if available once the execution of the current thread is over. This saves CPU cycles efficiently. Also, in the other way around the given sleep time can be insufficient for the execution of the given task. In such instances, the threads are killed in the middle of execution resulting malfunctioning of the program.

The CPU clock speeds vary from processor to processor. Hence, different systems can have different sleep times relative to each other even though the constant given is the same. This can affect the functionality of the code and may give erroneous results. This reduces the compatibility and portability of the program when needed to use in different applications.

Hence, using `join()` calls is a more efficient and a better way of getting the desired output of a multi threaded program.

#### **Exercise 4**

- 1. Use the following skeleton code to implement a multi-threaded server.**

MultiServer.c

- 2. Why do you need to declare `connfd` as a variable in the heap? What problem might occur if it was declared as a local variable?**

If `connfd` is declared as a local variable, it will be stored in the stack and its value will be changed in each iteration in the while loop. As threads are executed separately from the main thread they may not finish its tasks by the time the main thread goes to the next iteration of the while loop. Hence, the `connfd` variable can be replaced while another thread is using its value. To avoid such race conditions, using `connfd` as a variable in heap is a solution. This will make sure a new memory location is allocated to store the value of `connfd` each time. Memory should be released at the end of the thread execution.