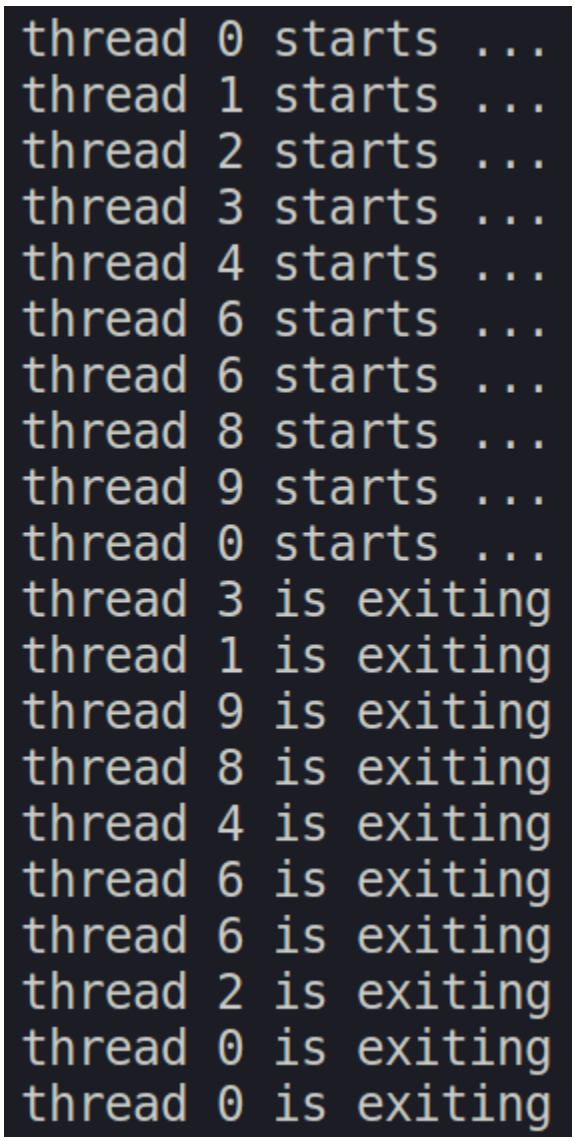
CS 425
Lab 3
Matthew Irizarry

Task 1

1. Is user defined tid correctly passed into the runner thread when there is only one? Are user defined tid correctly passed into the runner threads when there are multiple? If not, give an explanation of this phenomenon.



From the image to the side, we see that somewhere along the way, we lose 5,7, and 10. Why is this? To see what is different, let's compare the first file to the second file. Both scenarios have the same header files. Both establish the default attributes. However, we start to see a key difference when we look at how they create the threads. In the first example, the thread id passed is based on a variable **`i`**. Since the threads are created inside of the for loop, they all have access to the same **`i`** variable. This is where the issue is, because the thread may start executing after the variable **`i`** is incremented by the for loop. However, we can fix this! We can assign the ids to an array, which makes each memory location different, and thus passing the proper addresses. The

proper code is seen below.

```
pthread_attr_init(&attr);
// for (i = 0; i < NUM_THREADS; i++) {
//   pthread_create(&tid[i],&attr,runner, &i);
// }

// Create array to store the thread ids
int tids_user[NUM_THREADS];
for (i = 0; i < NUM_THREADS; i++) {
  tids_user[i] = i;
  pthread_create(&tid[i],&attr,runner, &tids_user[i
}
```

The commented code is the original code. To solve the issue we store the user thread id in an array. Then we pass a reference to that memory location. This solves our problem, as seen in this output to the right. The threads start in order, and then they finish as they do. One key difference is that every thread properly passes the thread id and all threads terminate successfully!

```
thread 0 starts ...
thread 1 starts ...
thread 2 starts ...
thread 3 starts ...
thread 4 starts ...
thread 5 starts ...
thread 6 starts ...
thread 7 starts ...
thread 8 starts ...
thread 9 starts ...
thread 3 is exiting
thread 0 is exiting
thread 6 is exiting
thread 8 is exiting
thread 1 is exiting
thread 2 is exiting
thread 7 is exiting
thread 9 is exiting
thread 4 is exiting
thread 5 is exiting
```

CS 425
Lab 3
Matthew Irizarry

Task Two

1. How does this program differ from the program in Task 1 from the perspective on how to pass user-defined tids into multiple runner threads?

As I had previously discussed with the solution to the first problem, instead of passing the address of the variable that is declared within the scope of the for loop, we instead define a list of numbers that increment through and use those addresses for our user TIDs. When threads try to access the address of the variable declared within the for loop, they might access it after the variable has been incremented by the for loop. This program differs in that the user defined TIDs are out of scope of the for loop, and each address has its own spot in an array.

2. Is the creation of threads in order consistent with the update loop variable i?

In the second example, the threads are created in the proper order, unlike the situation with the variable.

3. Do threads exit in the same order as the order of their creation? If not, what causes this phenomenon?

The threads do NOT exit in the same order as their creation, and this is just due to how threads work. They are asynchronous in that they run in the background, and then return their result when completed. This means on a lower level, they are constantly context switched on and off the CPU depending on what else is going on in the OS. Therefore, execution times will vary when using threads.

CS 425
Lab 3
Matthew Irizarry

Do the following step and answer the corresponding questions.

• Run the program three times, and record the output related to user-defined tid in each round of execution.

Thread	1st Trial (s)	2nd Trial (s)	3rd Trial (s)
0	.184	.197	.202
1	.199	.197	.197
2	.202	.200	.199
3	.203	.198	.202
4	.205	.192	.200
5	.173	.192	.196
6	.196	.183	.197
7	.202	.160	.197
8	.200	.201	.187
9	.196	.200	.191

5. Are threads created in the same order in different rounds of execution? Any explanation on your observation?

For each trial to the right, the order in which the threads were executed was the same. However, there was a difference in the exit order. This table to the left does NOT account for the order in which the threads are executed, just the time of the execution.

6. Do threads exit in the same order in different rounds of execution? Any explanation on your observation?

The threads DO NOT exit in the same order as their execution. This is because threads do not always have the same execution time. Sure, we can estimate how long a program will run based on input size, but it is just an estimation. Depending on what else is going on in the OS, performance, and many other factors, a thread will vary in its completion time. In my example, it finished on average in approximately 2 seconds. However, there was a distinguishable margin of error. At

the quickest, a thread completed in .16 seconds, and at the slowest, it took .205 seconds. This is the reason that the threads do not complete in the same order. They all take different amounts of time to complete, even if it's just off by a few milliseconds.

Thread	
5	.173
0	.184
6	.196
9	.196
1	.199
8	.200
2	.202
7	.202
3	.203
4	.205

Thread	
0	.197
7	.160
6	.183
4	.192
5	.192
1	.197
3	.198
2	.200
9	.200
8	.201

Thread	
8	.187
9	.191
5	.196
1	.197
6	.197
7	.197
2	.199
4	.200
0	.202
3	.202

The tables above represent the three trials. The first trial is on the left, the second in the middle, and the third on the right. We see that for each of these printed values, they increase towards the total execution time of the program. However, the order of completion is completely different.

Task Three

7. Explain the meaning of elapsed time in the main function by comparing the elapsed time of main thread and the elapsed time of runner threads.

In the elapsed time calculation in the runner thread, the following steps happen.

First, we “start” the clock, which means just to get the number of clock ticks since the program started. Divide by CLOCKS_PER_SEC to get the time in seconds. This is the amount of time it takes for the for loop to execute. If we look at the execution of the runner threads, they each measure how many clock cycles from the start of the for loop, and how many are at the end, and take the difference of them.

The elapsed time variable is the time in cycles until the threads complete their execution, and it computes the difference.

8. Does the workload of a thread have an impact on the order of their termination? If yes, summarize the impact.

Yes, if one thread takes longer to run because it has a heavier workload, then yes, it will terminate later than a thread that runs with a lighter workload. The impact is usually not drastic. For example, if on average a thread will take 5 minutes to complete, it's unlikely that one thread would take 10 minutes and the other take 1. It would likely have a small deviation between the top and bottom numbers, like we saw in the three tables above. In some instances, one thread would outperform another, even if it was just by a slim margin.

9. Without joining runner threads, what is the elapsed time of the main thread.

The elapsed time of the main thread is the amount of time from the start of the creation of the threads, until the other threads are done.

10. Design a solution to sum the elapsed time of all runner threads, and test it. Your solution does not have to be perfect. Working a few cases is considered an acceptable solution.

You will see the implemented code, but in short: I declared a global array to store each execution time of the runner functions. In the runner, I passed them to the array of execution times. At the end, I summed up the values of the array and

printed it to the user. There is a discrepancy between the time of the runner threads and the overall process itself due to other operations before and after the threads run.