EC527
Lab4 report
Bin Xu
Seyed Reza Sajjadinasab

Task1
By just adding this snippet of code we could print the first 3 elements of ArrayA:

```
the_data = ArrayA;
   printf("The first 3 elements of ArrayA are: %f, %f, %f", *((float*)
the_data), *((float*) the_data+1), *((float*) the_data+2));
```

Task2
By running the code a few times these different results showed up:

Hello test_create.c
Hello World! from child thread 7f8b45fd3640
Hello World! from child thread 7f8b467d4640
Hello World! from child thread 7f8b44fd1640
Hello World! from child thread 7f8b457d2640
main() after creating the thread. My id is 7f8b467d5740
Hello World! from child thread 7f8b447d0640
Hello World! from child thread 7f8b447d0640

Hello test_create.c
Hello World! from child thread 7ff4151d4640
Hello World! from child thread 7ff4141d2640
main() after creating the thread. My id is 7ff4159d6740
Hello World! from child thread 7ff4159d5640
Hello World! from child thread 7ff4139d1640
Hello World! from child thread 7ff4149d3640

Hello test_create.c
Hello World! from child thread 7fce3ef25640
Hello World! from child thread 7fce3e724640
Hello World! from child thread 7fce3d702640
main() after creating the thread. My id is 7fce3ef26740

Hello test_create.c
Hello World! from child thread 7f8ef5379640
Hello World! from child thread 7f8ef4377640
Hello World! from child thread 7f8ef3b76640
Hello World! from child thread 7f8ef4b78640

main() after creating the thread. My id is 7f8ef537a740
Hello World! from child thread 7f8ef3375640

This is because there is no guarantee that the threads are executed and finished in the same order in the code.

Task3

```
pthread_t *id = (pthread_t*) malloc((NUM_THREADS+1)*sizeof(pthread_t));
```

```
(pthread_create(++id, NULL, work, NULL))
```

These are the changes we've made to make to the code as wanted in task 3. The reason for allocating one more space to id is because it's increasing every time before creating a thread.

Task4

This is the output:

Hello test_create.c
main() after creating the thread. My id is 7fac8f4bd740

As we can see neither of the threads managed to print. It's because of the delay and that the main thread terminates the execution of the program.

Task5

This is of the output generated after adding sleep(3) before return.

Hello test_create.c
Hello World! from child thread 7fb3fd0e9640
main() after creating the thread. My id is 7fb3fd8eb740
Hello World! from child thread 7fb3fc0e7640
Hello World! from child thread 7fb3fb8e6640
Hello World! from child thread 7fb3fc8e8640
Hello World! from child thread 7fb3fd8ea640

In all of the results despite being out of order, all the child threads managed to print. It's because of the delay before terminating the program.

Task6

This time in all of the experiments the results were the same in terms of the fact that all child threads managed to be executed and printed.
It was as expected, since the main thread wouldn't terminate the program until all child thread finish executing.

Task7
As expected the result was the same. Below there is a output sample:

```
Hello test_param1.c
In main:  creating thread 0
In main:  creating thread 1
threadid before casting: 0
threadid before casting: 1
PrintHello() in thread # 1 !
PrintHello() in thread # 0 !
In main:  creating thread 2
In main:  creating thread 3
threadid before casting: 2
PrintHello() in thread # 2 !
In main:  creating thread 4
It's me MAIN -- Good Bye World!
threadid before casting: 3
PrintHello() in thread # 3 !
threadid before casting: 4
PrintHello() in thread # 4 !
```

Using a temporary variable of signed char set to -1 as the input result this:

```
Hello test_param1.c
In main:  creating thread 0
In main:  creating thread 1
In main:  creating thread 2
threadid before casting: -1
threadid before casting: -1
PrintHello() in thread # -1 !
PrintHello() in thread # -1 !
threadid before casting: -1
PrintHello() in thread # -1 !
In main:  creating thread 3
In main:  creating thread 4
threadid before casting: -1
PrintHello() in thread # -1 !
It's me MAIN -- Good Bye World!
threadid before casting: -1
PrintHello() in thread # -1 !
```

As we can see the result of casting before and after casting is the same. Trying some other integer types doesn't result in any difference. Also we check passing a double which doesn't compile at all. This is because the value of the address doesn't change and only the way the compiler is looking into it has changed.

Task8

The result of the code is:

```
in work(): f= 3, k=49999995000000, *g=10
in work(): f= 7, k=49999995000000, *g=10
in work(): f= 2, k=49999995000000, *g=10
in work(): f= 6, k=49999995000000, *g=10
in work(): f= 3, k=49999995000000, *g=10
in work(): f= 7, k=49999995000000, *g=10
in work(): f=10, k=49999995000000, *g=10
in work(): f= 4, k=49999995000000, *g=10
in work(): f=10, k=49999995000000, *g=10
in work(): f= 8, k=49999995000000, *g=10
k=4999999950000000
After creating the threads.  My id is 7fb7db7e2740, t = 10
```

As we can see f is changing despite the g is the same. That's because the location that g is pointing to is the same for all threads and it's showing the last change that has been made to it.

We added f++; The result is as below:

```
in work(): f= 7, k=49999995000000, *g=10
in work(): f= 4, k=49999995000000, *g=10
in work(): f=11, k=49999995000000, *g=10
in work(): f= 4, k=49999995000000, *g=10
in work(): f= 8, k=49999995000000, *g=10
in work(): f=11, k=49999995000000, *g=10
in work(): f= 4, k=49999995000000, *g=10
in work(): f= 7, k=49999995000000, *g=10
in work(): f= 5, k=49999995000000, *g=10
in work(): f=11, k=49999995000000, *g=10
k=4999999950000000
After creating the threads.  My id is 7fc074c43740, t = 10
```

It seems that now only the arbitrary order of f ranging below the 10 is ranging below the 12.

We did the same with (*g)++; The result is as below

```
in work(): f= 7, k=49999995000000, *g=11
in work(): f= 9, k=49999995000000, *g=12
in work(): f=10, k=49999995000000, *g=13
in work(): f= 4, k=49999995000000, *g=14
in work(): f= 5, k=49999995000000, *g=15
in work(): f= 5, k=49999995000000, *g=16
in work(): f= 1, k=49999995000000, *g=17
in work(): f= 7, k=49999995000000, *g=18
in work(): f=10, k=49999995000000, *g=19
```

in work(): f= 6, k=49999995000000, *g=20
k=4999999950000000
After creating the threads. My id is 7fe3f1bba740, t = 20

Now the *g is increasing in order!
The reason for this behavior is that *g is the pointing to the address of the value and every time it changed in the main function it affects the value of final *g. (The busy time in *g will guarantee that all threads have passed changing the value inside the address g.) On the other hand, the f is based on the exact value of the *g at the moment that its thread is reading it. So, it may have any values based on the last changes that the previous thread had made. (We can see that there are repetitive value in f)

When we are adding g++ after the busy part there is no guarantee that the value that is printed is the last change that happened to g, and we have race conditions.

Task9
We can make less threads get executed only by increasing the for loop constraints. In our experiment we set the upper bound of k to 100000000000000 of 100000000
None of the threads got executed because they were too busy! 🙂

Task10
For this part we declare global int count and initialize it with 0.
Then with this snippet of code initialize an array in the main().

```
  int *a = (int *) malloc(sizeof(int)*NUM_THREADS);
    a[0] = 0; a[1] = 1; a[2] = 2; a[3] = 3; a[4] = 4; a[5] = 5; a[6] = 6;
a[7] = 7; a[8] = 8; a[9] = 9;
```

In the work(void *i) we use this snippet of code:

```
int *gg = (int*) (i);
    int temp = gg[count];
    count++;
```

and each time print the value of temp as gg[ind]. The result is as below:
in work(): f= 0, k=49999995000000, *g=0, gg[ind]=6
in work(): f= 0, k=49999995000000, *g=0, gg[ind]=3
in work(): f= 0, k=49999995000000, *g=0, gg[ind]=0
in work(): f= 0, k=49999995000000, *g=0, gg[ind]=1
in work(): f= 0, k=49999995000000, *g=0, gg[ind]=7
in work(): f= 0, k=49999995000000, *g=0, gg[ind]=2
in work(): f= 0, k=49999995000000, *g=0, gg[ind]=4
in work(): f= 0, k=49999995000000, *g=0, gg[ind]=5
in work(): f= 0, k=49999995000000, *g=0, gg[ind]=8
in work(): f= 0, k=49999995000000, *g=0, gg[ind]=9

k=499999999500000000
After creating the threads. My id is 7fa9c9ebd740, t = 10

As we can see each thread printed a unique value.

Note: we increment the upper bound of k to make the busy time more so that all of the threads get executed.

Although this method worked, for a larger number of threads it failed to work. It's because the contention part when using a global variable and trying to increment it only in count++ is less than when we are loading from memory (int f = *((int*)(i));). Thus, for a small amount of threads it's not likely to see the effects of race condition. To handle this problem it is better to use mutex and lock it before count++ and unlock it after. Without it, there is no reliable way to ensure desired synchronization.

Task11

The result of running the program is:

```
Hello test_param3.c
In main:  creating thread 0
In main:  creating thread 1
In main:  creating thread 2
in PrintHello(), thread #1 ; sum = 28, message = Second Message
In main:  creating thread 3
in PrintHello(), thread #2 ; sum = 29, message = Third Message
in PrintHello(), thread #3 ; sum = 30, message = Fourth Message
in PrintHello(), thread #0 ; sum = 27, message = First Message
In main:  creating thread 4
in PrintHello(), thread #4 ; sum = 31, message = Fifth Message
```

As we can see every thread printed its unique values.

```
Hello test_param3.c
In main:  creating thread 0
In main:  creating thread 1
In main:  creating thread 2
in PrintHello(), thread #0 ; sum = 27, message = First Message
in PrintHello(), thread #1 ; sum = 28, message = Second Message
in PrintHello(), thread #2 ; sum = 29, message = Third Message
In main:  creating thread 3
In main:  creating thread 4
in PrintHello(), thread #3 ; sum = 30, message = Fourth Message
In main:  creating thread 5
in PrintHello(), thread #4 ; sum = 31, message = Fifth Message
in PrintHello(), thread #5 ; sum = 100, message = Sixth Message
```

We got this by changing the code as below:

```
thread_data_array[t].sum = (t<5)? t+27: 100;
```

Task12

Having the trylock, the child thread will wait until the user enters something and then print. Without it, the child thread just starts doing its stuff regardless of the input of the user.

With trylock:
Hello sync1
In main: created thread 1, which is blocked
  Press any letter key (not space) then press enter:
s
in thread ID 0 (sum = 123 message = First Message), now unblocked!
After joining
GOODBYE WORLD!

Without trylock:
Hello sync1
In main: created thread 1, which is blocked
  Press any letter key (not space) then press enter:
in thread ID 0 (sum = 123 message = First Message), now unblocked!
s
After joining
GOODBYE WORLD!

As we can see the input “s” of the user is after the printing in the version without trylock.

Task13

By increasing the number of threads the program stuck! It’s because we only have one lock and in the first iteration of creating a thread we locked it.

Task14

By running the program the result was as below:
Hello test_barrier.c
In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
Thread 1 printing before barrier 1 of 3
Thread 1 printing before barrier 2 of 3
Thread 1 printing before barrier 3 of 3

```
In main:  creating thread 3
In main:  creating thread 4
Thread 2 printing before barrier 1 of 3
After creating the threads; my id is 7f66379ab740
Thread 2 printing before barrier 2 of 3
Thread 2 printing before barrier 3 of 3
Thread 4 printing before barrier 1 of 3
Thread 4 printing before barrier 2 of 3
Thread 4 printing before barrier 3 of 3
Thread 3 printing before barrier 1 of 3
Thread 3 printing before barrier 2 of 3
Thread 3 printing before barrier 3 of 3
Thread 0 printing before barrier 1 of 3
Thread 0 printing before barrier 2 of 3
Thread 0 printing before barrier 3 of 3
Thread 1 after barrier (print 1 of 2)
Thread 4 after barrier (print 1 of 2)
Thread 1 after barrier (print 2 of 2)
Thread 0 after barrier (print 1 of 2)
Thread 3 after barrier (print 1 of 2)
Thread 0 after barrier (print 2 of 2)
Thread 4 after barrier (print 2 of 2)
Thread 2 after barrier (print 1 of 2)
Thread 2 after barrier (print 2 of 2)
Thread 3 after barrier (print 2 of 2)
After joining
```

As we can see all the “after barrier” prints are after barriers.

By uncommenting the sleep(1). The program child threads waited until all the child threads were created and then started printing. Also, the barrier still workes. This is the output:

```
Hello test_barrier.c
In main:  creating thread 0
In main:  creating thread 1
In main:  creating thread 2
In main:  creating thread 3
In main:  creating thread 4
After creating the threads; my id is 7f664b61b740
Thread 1 printing before barrier 1 of 3
Thread 2 printing before barrier 1 of 3
Thread 0 printing before barrier 1 of 3
Thread 4 printing before barrier 1 of 3
Thread 3 printing before barrier 1 of 3
```

Thread 1 printing before barrier 2 of 3
Thread 4 printing before barrier 2 of 3
Thread 0 printing before barrier 2 of 3
Thread 3 printing before barrier 2 of 3
Thread 2 printing before barrier 2 of 3
Thread 1 printing before barrier 3 of 3
Thread 4 printing before barrier 3 of 3
Thread 0 printing before barrier 3 of 3
Thread 3 printing before barrier 3 of 3
Thread 2 printing before barrier 3 of 3
Thread 1 after barrier (print 1 of 2)
Thread 0 after barrier (print 1 of 2)
Thread 1 after barrier (print 2 of 2)
Thread 2 after barrier (print 1 of 2)
Thread 2 after barrier (print 2 of 2)
Thread 4 after barrier (print 1 of 2)
Thread 4 after barrier (print 2 of 2)
Thread 3 after barrier (print 1 of 2)
Thread 3 after barrier (print 2 of 2)
Thread 0 after barrier (print 2 of 2)
After joining

By changing the value of sleep to a proponent to the taskid, the barrier works too regardless of the sleep time. (Of course it took more time!)

Task15

Hello test_sync2.c
thread #2 waiting for 1 ...
thread #4 waiting for 2 ...
Main: calling sleep(1)...
thread #5 waiting for 4 ...
thread #7 waiting for 5 and 6 ...
thread #6 waiting for 3 and 4 ...
thread #3 waiting for 1 ...
Main: created threads 2-7, type a letter (not space) and <enter>
1
Main: waiting for thread 7 to finish, UNLOCK LOCK 1
Main: Done unlocking 1
It's me, thread #2! I'm unlocking 2...
It's me, thread #3! I'm unlocking 3...
It's me, thread #4! I'm unlocking 4...
It's me, thread #6! I'm unlocking 6...
It's me, thread #5! I'm unlocking 5...
It's me, thread #7! I'm unlocking 7...

Main: After joining

Each thread depends on the other thread. join() is executed after all threads exits. The threads work in the specific order.

Task 16

```
case 8:
    printf("thread #%ld waiting for 6 and 7 ...\n", taskid);
    pthread_mutex_lock(&mutexA[6]);
    pthread_mutex_unlock(&mutexA[6]);
    pthread_mutex_lock(&mutexA[7]);
    pthread_mutex_unlock(&mutexA[7]);
    break;
```

Hello test_sync2.c
thread #2 waiting for 1 ...
thread #3 waiting for 1 ...
thread #5 waiting for 4 ...
thread #6 waiting for 3 and 4 ...
thread #4 waiting for 2 ...
thread #7 waiting for 5 and 6 ...
Main: calling sleep(1)...
thread #8 waiting for 6 and 7 ...
Main: created threads 2-8, type a letter (not space) and <enter>
1
Main: waiting for thread 7 to finish, UNLOCK LOCK 1
Main: Done unlocking 1
It's me, thread #2! I'm unlocking 2...
It's me, thread #3! I'm unlocking 3...
It's me, thread #4! I'm unlocking 4...
It's me, thread #5! I'm unlocking 5...
It's me, thread #6! I'm unlocking 6...
It's me, thread #7! I'm unlocking 7...
It's me, thread #8! I'm unlocking 8...
Main: After joining

Task17

It's not valid everytime. Sometimes it's correct as below:

Hello test_crit.c
MAIN --> final balance = 1000
qr_total = 18.035593

And sometimes it's not:

```
Hello test_crit.c
 MAIN --> final balance = 1001
        qr_total = 18.035593
```

Increasing the number of threads makes it even worse and in more experiments the results weren't the same, both for quasi_random values and balance.

This is because of the race condition. In other words, we cannot guarantee whether the balance that a thread is reading and writing after incrementing/decrementing hasn't changed by other threads.

Below you may see some of experiments:

10 threads:

```
Hello test_crit.c
 MAIN --> final balance = 1000
        qr_total = 18.035593
xu842251@vlsi28$ ./test_crit
Hello test_crit.c
 MAIN --> final balance = 1000
        qr_total = 18.035593
xu842251@vlsi28$ ./test_crit
Hello test_crit.c
 MAIN --> final balance = 999
        qr_total = 18.035593
xu842251@vlsi28$ ./test_crit
Hello test_crit.c
 MAIN --> final balance = 998
        qr_total = 18.035593
xu842251@vlsi28$ ./test_crit
Hello test_crit.c
 MAIN --> final balance = 1001
        qr_total = 18.035593
```

10000 threads:

```
Hello test_crit.c
 MAIN --> final balance = 998
        qr_total = 17457.276235
xu842251@vlsi28$ ./test_crit
Hello test_crit.c
```

```
 MAIN --> final balance = 1001
        qr_total = 17464.076758
xu842251@vlsi28$ ./test_crit
Hello test_crit.c
 MAIN --> final balance = 994
        qr_total = 17460.638721
xu842251@vlsi28$ ./test_crit
Hello test_crit.c
 MAIN --> final balance = 1001
        qr_total = 17462.146701
xu842251@vlsi28$ ./test_crit
Hello test_crit.c
 MAIN --> final balance = 1000
        qr_total = 17455.850391
xu842251@vlsi28$ ./test_crit
Hello test_crit.c
 MAIN --> final balance = 1005
        qr_total = 17463.401638
```

Task 18

After adding pthread_mutex_t, we get the below result. The balance gets the correct result.

```
Hello test_crit.c
 MAIN --> final balance = 1000
        qr_total = 1744.073537
xu842251@vlsi28$ ./test_crit
Hello test_crit.c
 MAIN --> final balance = 1000
        qr_total = 1744.073537
xu842251@vlsi28$ ./test_crit
Hello test_crit.c
 MAIN --> final balance = 1000
        qr_total = 1744.073537
xu842251@vlsi28$ ./test_crit
Hello test_crit.c
 MAIN --> final balance = 1000
        qr_total = 1744.073537
xu842251@vlsi28$ ./test_crit
Hello test_crit.c
 MAIN --> final balance = 1000
        qr_total = 1744.073537
xu842251@vlsi28$ ./test_crit
Hello test_crit.c
```

```
 MAIN --> final balance = 1000
        qr_total = 1744.073537
```

Task 19

I set the iteration as 2100000.

```
Thread for index 1 starting
Thread for index 5 starting
Thread for index 7 starting
Thread for index 8 starting
Thread for index 2 starting
Thread for index 6 starting
Thread for index 4 starting
Thread for index 9 starting
Thread for index 3 starting
Thread for index 10 starting
  0.00  9.09 18.18 27.27 36.36 45.45 54.55 63.64 72.73 81.82 90.91 100.00

real    0m0.261s
user    0m1.926s
sys     0m0.004s
```

Task 20

```
Thread for index 1 starting
Thread for index 4 starting
Thread for index 7 starting
Thread for index 5 starting
Thread for index 10 starting
Thread for index 6 starting
Thread for index 3 starting
Thread for index 8 starting
Thread for index 9 starting
Thread for index 2 starting
  0.00  9.09 18.18 27.27 36.35 45.45 54.54 63.63 72.72 81.81 90.91 100.00

real    0m0.196s
user    0m0.085s
sys     0m0.345s
```

When the iteration is 10000, the Barriers 1 reaches a steady state. It takes 0.196s.
Barriers 1 is faster. Because the pthread_barrier increases the parallelism and reduces the amount of time thread spends on waiting for each other.
Barriers 0 needs more iterations. Because it do not use the `pthread_barrier_wait`

, it causes race conditions.
Barriers 1 consistently gives the same answer. Because it uses `pthread_barrier_wait` to make the thread synchronized.
In summary, the version with USE_BARRIERS set to 1 takes fewer iterations to reach steady-state equilibrium compared to the version with USE_BARRIERS set to 0.