Mikhail Andreev

EC527: Assignment 5

Task 1: test_generic.c

Using the following code, ArrayA can print the float values.

```
// use "the_data" to print out the contents of ArrayA
  the_data = &ArrayA;

  printf("the_data now points to the float %f, %f, %f\n", *(float*) the_data,
*(((float *) the_data) + 1), *(((float *) the_data) + 2));

  printf("the_data now points to the float %f, %f, %f\n", ((float*)
the_data)[0], ((float*) the_data)[1], ((float*) the_data)[2]);
```

Task 2: test_create.c base code

The output from the base code is given below. The id's on the thread have definitely changed:

```
main() after creating the thread.  My id is 139783900706560
 Hello World! from child thread 139783850694400
 Hello World! from child thread 139783892653824
 Hello World! from child thread 139783882163968
 Hello World! from child thread 139783871674112
 Hello World! from child thread 139783861184256


main() after creating the thread.  My id is 139783900706560
 Hello World! from child thread 139783850694400
 Hello World! from child thread 139783892653824
 Hello World! from child thread 139783882163968
 Hello World! from child thread 139783871674112
 Hello World! from child thread 139783861184256


main() after creating the thread.  My id is 140664013453056
 Hello World! from child thread 140663963440896
 Hello World! from child thread 140664005400320
 Hello World! from child thread 140663994910464
 Hello World! from child thread 140663984420608
 Hello World! from child thread 140663973930752
```

Task 3: change id to be a pointer

Using the following code, this is achieved:

```
pthread_t *id;
  id = (pthread_t*) malloc(NUM_THREADS);

  for (i = 0; i < NUM_THREADS; ++i) {
    if (pthread_create((id++), NULL, work, NULL)) {
      printf("ERROR creating the thread\n");
      exit(19);
    }
  }
```

Task 4: sleep(3)

Code for work:

```
void *work(void *i)
{
      sleep(3);
  printf(" Hello World! from child thread %lu\n",
       pthread_self());

  pthread_exit(NULL);
}
```

The output is a single line stating:

```
main() after creating the thread.  My id is 140664013453056
```

with different id values. The child threads do not print anything. This is happening because the main thread terminates before the child threads reach the print statements. This causes them to also be terminated.

Task 5: sleep(3) in main

Code has been changed to:

```
printf("\n main() after creating the thread.  My id is %lu\n",
       pthread_self());
      sleep(3);

  return(0);
```

This causes all the thread statements to be printed, as well as the main thread statement. Execution terminates 3 seconds after the last statement. This happens because all the threads now have time to finish executing.

Task 6: sleep with join

The code has been changed to the following:

```
void *work(void *i)
{
  sleep(3);
  printf(" Hello World!  It's me, thread #%lu --\n",
       pthread_self());
  pthread_exit(NULL);
}
```

The output starts by outputting the main thread statement. Then after a 3 second delay, all the child threads output their print statements. After this the main thread outputs its final statement and terminates.

This occurs because since the main thread blocks while the child threads are running, it has to wait for the threads to complete before finishing and joining. This causes the full body of the child threads to be executed, resulting in the delays and prints.

Task 7: passing parameter

Using the code:

```
void *PrintHello(void *threadid)
{
  printf("%d\n", threadid);
  long unsigned int tid = (long unsigned int) threadid;

  printf(" Hello World!  It's me, thread # %lx ! \n", tid);

  pthread_exit(NULL);
}
```

We get the output:

```
 Hello World!  It's me, MAIN!
In main:  creating thread 0
In main:  creating thread 1
0
 Hello World!  It's me, thread # 0 !
In main:  creating thread 2
1
 Hello World!  It's me, thread # 1 !
In main:  creating thread 3
2
 Hello World!  It's me, thread # 2 !
In main:  creating thread 4
3
 Hello World!  It's me, thread # 3 !
4
 Hello World!  It's me, thread # 4 !

 It's me MAIN -- Good Bye World!
```

Clearly, in this case the output is the same regardless of what the pointer is cast to.

If we change the code use the signed char:

```
signed char t_c = -4;

  printf("\n Hello World!  It's me, MAIN!\n");

  for (t = 0; t < NUM_THREADS; t++) {
    printf("In main:  creating thread %ld\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void*) t_c);
```

The code does compile and the output is:

```
 Hello World!  It's me, MAIN!
In main:  creating thread 0
In main:  creating thread 1
-4
 Hello World!  It's me, thread # fffffffffffffffc !
In main:  creating thread 2
-4
 Hello World!  It's me, thread # fffffffffffffffc !
In main:  creating thread 3
-4
 Hello World!  It's me, thread # fffffffffffffffc !
In main:  creating thread 4
-4
 Hello World!  It's me, thread # fffffffffffffffc !
-4
 Hello World!  It's me, thread # fffffffffffffffc !

 It's me MAIN -- Good Bye World!
```

The variable is printed out correctly, but when converting to an unsigned long, we get the unsigned representation of the variable instead.

Task 8: print out

The printout will sometimes output messages in different orders, just based on what manages to finish processing first. However, the content is the same.

Task 9: changing variables

Without any changes we have:

```
Hello World! 1  1
Hello World! 7  10
Hello World! 3  10
Hello World! 10  10
Hello World! 10  10
Hello World! 10  10
Hello World! 2  10
Hello World! 10  10
Hello World! 4  10
Hello World! 10  10
After creating the thread.  My id is 139876371523328, i = 10
```

Changing f in the code with:

```
long int j, k;
  int f = *((int*)(i))+5;  // get the value being pointed to
  int *g = (int*)(i);    // get the pointer itself
```

We get the output:

```
Hello World! 6  1
Hello World! 10  10
Hello World! 15  10
Hello World! 15  10
Hello World! 7  10
Hello World! 15  10
Hello World! 15  10
Hello World! 8  10
Hello World! 15  10
Hello World! 10  10
After creating the thread.  My id is 140696733218560, i = 10
```

Changing the code to modify g, we have:

```
int *g = (int*)(i);    // get the pointer itself
  printf("g = %lu - %d\n", g, *g);
  *g = (*g) + 89;
  printf("g = %lu - %d\n", g, *g);
```

The output after this is:

```
Hello World! 1  182
g = 140735472104460 - 3
g = 140735472104460 - 2
g = 140735472104460 - 93
g = 140735472104460 - 1
g = 140735472104460 - 182
g = 140735472104460 - 272
g = 140735472104460 - 272
g = 140735472104460 - 361
g = 140735472104460 - 361
g = 140735472104460 - 450

Hello World! 3  450
Hello World! 2  450
Hello World! 272  450
Hello World! 361  450
After creating the thread.  My id is 139657121900288, i = 450
```

As more threads are created, segmentation faults start to appear. This occurs because the changes to the loop variable eventually propagate down to the main thread which is still creating threads. If the loop variable exceeds the length of the pthread_t array, a segmentation fault will occur. This also causes less than NUM_THREADS threads to be created.

Task 10: passing an array

Using the following code we can pass an array to work:

```
for(i = 0; i < NUM_THREADS; i++)
    values[i] = i;

  printf("\n\nValues:\n");
  for(i = 0; i < NUM_THREADS; i++){
    printf("%d\t", values[i]);
  }
  printf("\n\n");

  for (i = 0; i < NUM_THREADS; ++i) {
    if ((i < NUM_THREADS) && pthread_create(&id[i], NULL, work, (void
*)(values+i))) {
      printf("ERROR creating the thread\n");
      exit(19);
    }
    len++;
  }
```

This gives the output:

```
Values:
0     1     2     3     4     5     6     7     8     9


Hello World! 0  1
Hello World! 2  3
Hello World! 6  7
Hello World! 3  4
Hello World! 8  9
Hello World! 4  5
Hello World! 7  8
Hello World! 5  6
Hello World! 1  2
Hello World! 9  10
After creating the thread.  My id is 140359201400576, i = 10


Values:
1     2     3     4     5     6     7     8     9     10
```

As can be seen the values increment. The way this happens is because we pass the pointer to the array with an offset to the correct location. This method will work as long as the array is continuous.

Task 11: passing a struct

The output from the given code is:

```
 Hello World!  It's me, MAIN!
In main:  creating thread 0
In main:  creating thread 1
 Hello World!  It's me, thread #0! sum = 28 message = First Message
In main:  creating thread 2
 Hello World!  It's me, thread #1! sum = 29 message = Second Message
In main:  creating thread 3
 Hello World!  It's me, thread #2! sum = 30 message = Third Message
 Hello World!  It's me, thread #3! sum = 31 message = Fourth Message
In main:  creating thread 4
 Hello World!  It's me, thread #4! sum = 32 message = Fifth Message
```

Using this code we can make a 6th thread:

```
for (t = 0; t < NUM_THREADS; t++) {
    thread_data_array[t].thread_id = t;
    thread_data_array[t].sum = t+28;
    if(t == 5)
      thread_data_array[t].sum = 1000;
    thread_data_array[t].message = Messages[t];
    printf("In main:  creating thread %ld\n", t);
```

Which gives us the output:

```
 Hello World!  It's me, MAIN!
In main:  creating thread 0
In main:  creating thread 1
 Hello World!  It's me, thread #0! sum = 28 message = First Message
In main:  creating thread 2
 Hello World!  It's me, thread #1! sum = 29 message = Second Message
In main:  creating thread 3
 Hello World!  It's me, thread #2! sum = 30 message = Third Message
In main:  creating thread 4
 Hello World!  It's me, thread #3! sum = 31 message = Fourth Message
In main:  creating thread 5
 Hello World!  It's me, thread #4! sum = 32 message = Fifth Message
 Hello World!  It's me, thread #5! sum = 1000 message = (null)
```

Task 12:

Output without changes:

```
 Hello World!  It's me, MAIN!
In main: created thread 1, which is blocked -- press a char and enter
s

   I, thread #0 (sum = 28 message = First Message) am now unblocked!

After joining

GOODBYE WORLD!
```

After commenting out the lock, we have the output:

```
 Hello World!  It's me, MAIN!
In main: created thread 1, which is blocked -- press a char and enter

   I, thread #0 (sum = 28 message = First Message) am now unblocked!
s
After joining

GOODBYE WORLD!
```

In both cases the s is the character entered. So in the first case, we had to wait to enter the character to pass the lock. In the second case the child thread ignores the lock and just runs, then we have to enter the character to finish the main thread.

Task 13:

After changing the thread count, the output from the main thread no longer responds and everything is blocked. This indicates having multiple threads on the same mutex does not work.

Task 14:

Code output before edits verify that all before barrier statements occur prior to after barrier prints:

```
 Hello World!  It's me, MAIN!
In main:  creating thread 0
In main:  creating thread 1
Thread 0 printing before barrier 1 of 3
Thread 0 printing before barrier 2 of 3
Thread 0 printing before barrier 3 of 3
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
Thread 2 printing before barrier 1 of 3
Thread 2 printing before barrier 2 of 3
Thread 2 printing before barrier 3 of 3
In main:  creating thread 4
Thread 3 printing before barrier 1 of 3
Thread 3 printing before barrier 2 of 3
Thread 3 printing before barrier 3 of 3

After creating the threads.  My id is:  140411790587648
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 0 printing after barrier 1 of 2
Thread 0 printing after barrier 2 of 2
Thread 1 printing after barrier 1 of 2
Thread 1 printing after barrier 2 of 2
Thread 2 printing after barrier 1 of 2
Thread 2 printing after barrier 2 of 2
Thread 3 printing after barrier 1 of 2
Thread 3 printing after barrier 2 of 2
Thread 4 printing after barrier 1 of 2
Thread 4 printing after barrier 2 of 2

After joining
GOODBYE WORLD!
```

After sleep is uncommented, the print statement sequence is still the same, however, the statements are outputted in stages where each second 1 of 3 statements are outputted for each thread:

```
 Hello World!  It's me, MAIN!
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:  140339845330688
Thread 0 printing before barrier 1 of 3
Thread 1 printing before barrier 1 of 3
Thread 2 printing before barrier 1 of 3
Thread 3 printing before barrier 1 of 3
Thread 4 printing before barrier 1 of 3
Thread 0 printing before barrier 2 of 3
Thread 1 printing before barrier 2 of 3
Thread 2 printing before barrier 2 of 3
Thread 3 printing before barrier 2 of 3
Thread 4 printing before barrier 2 of 3
Thread 0 printing before barrier 3 of 3
Thread 1 printing before barrier 3 of 3
Thread 2 printing before barrier 3 of 3
Thread 3 printing before barrier 3 of 3
Thread 4 printing before barrier 3 of 3
Thread 4 printing after barrier 1 of 2
Thread 4 printing after barrier 2 of 2
Thread 2 printing after barrier 1 of 2
Thread 2 printing after barrier 2 of 2
Thread 3 printing after barrier 1 of 2
Thread 3 printing after barrier 2 of 2
Thread 1 printing after barrier 1 of 2
Thread 1 printing after barrier 2 of 2
Thread 0 printing after barrier 1 of 2
Thread 0 printing after barrier 2 of 2

After joining
GOODBYE WORLD!
```

After changing the sleep time to tasked we have the lower id threads finishing execution much faster than higher id threads:

```
 Hello World!  It's me, MAIN!
In main:  creating thread 0
In main:  creating thread 1
Thread 0 printing before barrier 1 of 3
Thread 0 printing before barrier 2 of 3
Thread 0 printing before barrier 3 of 3
In main:  creating thread 2
In main:  creating thread 3
In main:  creating thread 4

After creating the threads.  My id is:  139847352055552
Thread 1 printing before barrier 1 of 3
Thread 2 printing before barrier 1 of 3
Thread 1 printing before barrier 2 of 3
Thread 3 printing before barrier 1 of 3
Thread 1 printing before barrier 3 of 3
Thread 4 printing before barrier 1 of 3
Thread 2 printing before barrier 2 of 3
Thread 3 printing before barrier 2 of 3
Thread 2 printing before barrier 3 of 3
Thread 4 printing before barrier 2 of 3
Thread 3 printing before barrier 3 of 3
Thread 4 printing before barrier 3 of 3
Thread 4 printing after barrier 1 of 2
Thread 4 printing after barrier 2 of 2
Thread 1 printing after barrier 1 of 2
Thread 1 printing after barrier 2 of 2
Thread 3 printing after barrier 1 of 2
Thread 3 printing after barrier 2 of 2
Thread 2 printing after barrier 1 of 2
Thread 2 printing after barrier 2 of 2
Thread 0 printing after barrier 1 of 2
Thread 0 printing after barrier 2 of 2

After joining
GOODBYE WORLD!
```

Task 15:

Output without any changes:

```
 Hello World!  It's me, MAIN!

It's me, thread #2! I'm waiting for 1 ...
It's me, thread #3! I'm waiting for 1 ...
It's me, thread #4! I'm waiting for 2 ...
It's me, thread #5! I'm waiting for 4 ...
It's me, thread #6! I'm waiting 3 and 4 ...
It's me, thread #7! I'm waiting 5 and 6 ...
 Created threads 2-7, type any character and <enter>

Waiting for thread 7 to finish, UNLOCK LOCK 1

 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...
After joining
GOODBYE WORLD!
```

The main thread waits to join the remaining threads because a lock is set to wait for the 7th thread to unlock before unlocking, which causes the join statements to wait.

Task 16:

Code that adds 8th thread:

```
  case 8:
    printf("\nIt's me, thread #%ld! I'm waiting 6 and 7 ...", taskid);
    pthread_mutex_lock(&mutexA[6]);
    pthread_mutex_unlock(&mutexA[6]);
    pthread_mutex_lock(&mutexA[7]);
    pthread_mutex_unlock(&mutexA[7]);
  default:
```

Task 17:

Under normal code execution the balance is the same as the start. However, when upgrading to 10000 threads causes the balance to get skewed. The reason the normal code is fine is because the accesses to the shared variable are not frequent enough to trigger corruption. However, when the number of threads increases, this causes an error.

Task 18: