Chih Han Yeh
EC527
Lab 4
Spring 2025

Task 1.

```c
int main()
{
  int i;
  char c;
  float ArrayA[5] = {1.1, 2.2, 3.3, 4.4, 5.5};
  void *the_data;       /* generic pointer */

  i = 6;
  c = 'a';

  the_data = &i;       /* make ptr point to a memory location */
  printf("the_data points to the integer value %d\n", *(int*) the_data);

  the_data = &c;       /* make ptr point to a different location */
  printf("the_data now points to the character %c\n", *(char*) the_data);

  /* MODIFY: use "the_data" to print out the contents of ArrayA */
  the_data = ArrayA;

  printf("Printing with array indexing: %.1f, %.1f, %.1f\n",
    ((float*)the_data)[0],
    ((float*)the_data)[1],
    ((float*)the_data)[2]);

  return 0;
}
```

Task 2.
The outputs of the ID of each iteration is different than each other.

Task 4.
Each child thread created through the work function sleeps for 3 seconds before printing. But main() does not wait for that process; instead, it finishes execution and exits before the child threads are awake and print. Therefore, with the child threads terminated when main() exits, we never receive their outputs.

Task 5.
This time the child threads would output because after execution, main() would pause for 3 seconds before exiting.

Task 6.
Instead of behaving like test_create.c in task 4, adding sleep(3) makes test_join.c behave similarly to test_create.c in task 5, which the child processes would complete and output before main() completes the program execution. This is due to pthreads_join(), which main() is paused and waits for each child process to complete before continuing execution.

Task 7.
Printing the value of threadid prints the same number as the print statement of the *PrintHello function. When passing a signed long integer to pthread_crreate() as a void*, a 64-bit sign extended hexadecimal representation of that signed char is displayed instead. In my demonstration, I passed in the value of -2, which would sign extend to 0xfffffffffffffffe.

Task 8.
The copied value f has different values (2, 4, 3) then maintained at 10 as t in main() reaches 10. g* points to t therefore the later threads all point to 10 as well. For this task I added: f += 10; and *g += 20; respectively; f would output 11 and 32 due to the result of incrementing by 10 after copying t, and *g appears as 42 as the pointer to t and maintaining for all threads.

Task 9.
Edited *work

```
void *work(void *i)
{
  long int k;
  int f = *((int*)(i));  // get the value being pointed to
  int *g = (int*)(i);    // get the pointer itself

  // only let threads with %2 == 0 values to run
  if (f % 2 != 0){
    pthread_exit(NULL);
  }

  f += 10;
  *g += 20;

  /* busy work */
  k = 0;
  for (long j=0; j < 10000000; j++) {
    k += j;
  }

  /* printf("Hello World from %lu with value %d\n", pthread_self(), f); */
  printf("in work(): f=%2d, k=%ld, *g=%d\n", f, k, *g);

  pthread_exit(NULL);
}
```

Task 10. Modified test_param2.c

```
/*******************/
void *work(void *arg)
{
  int *data = (int *)arg;
  int t_index = data[0];
  int *t_array = data + 1;

// Modify the thread’s assigned index in the array
  if (t_index >= 0 && t_index < NUM_THREADS){
    t_array[t_index] += t_index * 4;
    printf("Thread %d modified array[%d] to %d\n",
      t_index, t_index, t_array[t_index]);
    }

  free(arg);  // Free allocated memory
  pthread_exit(NULL);
}



/**************************************************************************/
int main(int argc, char *argv[])
{
  pthread_t id[NUM_THREADS];
  int t_array[NUM_THREADS];  // Create a shared array

  // Initialize array values
  for (int i = 0; i < NUM_THREADS; i++) {
    t_array[i] = i * 5;
  }

  for (long t = 0; t < NUM_THREADS; ++t) {
    int *thread_data = malloc((NUM_THREADS + 1) * sizeof(int));
    if (thread_data == NULL) {
      printf("ERROR allocating memory\n");
      exit(1);
    }

    thread_data[0] = t;  // Store thread index
    for (int i = 0; i < NUM_THREADS; i++) {
      thread_data[i + 1] = t_array[i];  // Copy the array
    }

    if (pthread_create(&id[t], NULL, work, (void *)thread_data)) {
```

```
      printf("ERROR creating the thread\n");
      exit(19);
    }
  }

  // Wait for all threads
  for (long t = 0; t < NUM_THREADS; ++t) {
    pthread_join(id[t], NULL);
  }

  // Print final array
  printf("Final array: ");
  for (int i = 0; i < NUM_THREADS; i++) {
    printf("%d ", t_array[i]);
  }
  printf("\n");

  return 0;

} /* end main */
```

Using malloc() to dynamically allocate and pass a copy of each thread's index would ensure that each thread receives a distinct value from another.

Task 11. Edited main()

```
/**************************************************************************/
int main(int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  struct thread_data thread_data_array[NUM_THREADS];
  int rc;
  char *Messages[NUM_THREADS] = {"First Message",
                                 "Second Message",
                                 "Third Message",
                                 "Fourth Message",
                                 "Fifth Message",
                                 "Sixth Message"};

  printf("Hello test_param3.c\n");

  for (long t = 0; t < NUM_THREADS; t++) {
    thread_data_array[t].thread_id = t;

    if (t == 5){
      thread_data_array[t].sum = 1000;
```

```
    } else{
      thread_data_array[t].sum = t+27;
    }
    thread_data_array[t].message = Messages[t];
    printf("In main:  creating thread %ld\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello,
                                              (void*) &thread_data_array[t]);
    if (rc) {
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }

  pthread_exit(NULL);

} /* end main */
```

Task 12 & 13.
Commenting out trylock results in the “in thread ID 0 (sum = 123 message = First Message), now unblocked!” string to output before the user is supposed to generate an input (“ Press any letter key (not space) then press enter: “). Changing NUM_THREADS to 2 enabled the program to output the messages from threads0 and thread1.

Task 14.
When executing the program, threads 0 and 1 would print their “before barrier” messages first. Threads 2, 3, and 4 starts in sequence, but since they start later than threads 0 and 1, they reach the barrier later than the two threads, but once they reach the barrier they print their “after barrier” messages simultaneously. Adding in sleep(1) forces the threads to execute together due to the delay of execution for each thread. Because each thread is printing synchronously, they don’t appear to print in sequence or order. When changing sleep(1) to sleep(taskid + 1), each subsequent print message is outputted slower than its predecessor due to linearly increasing sleep time. The threads’ “printing before” statements appear to be out of order, but the same does not happen for the thread after barrier statements.

Task 15.
The join() loop runs after most threads have finished executing as they execute independently from the main program.

Task 16.
File Submission

Task 17.
When the number of threads is low, each outcome of executing the program would result in the same balance (NUM_THREADS being 10 by default). Increasing the number to 10,000 would result in the different total values of the final balance and qr_total for each execution.

Task 19.
When iterations reach 1000,000 the solution reaches very close to steady-state. Though unreliable, a higher iteration count would ensure a guaranteed solution to reach steady-state, but the first encounter with a very close answer to the steady-state solution was 1000,000 iterations. The real time the process took was 0.517s

Task 20.
Setting USE_BARRIERS to 1, I could use an iteration as low as 5000 to get a solution very close to the steady-state. The real time it took was 0.036s, proving that setting USE_BARRIERS = 1 is a faster and and more synchronized method than its counterpart. USE_BARRIERS = 0 causes asynchronous updates to hv_to[i] of each thread leading to inconsistent updates, slowing down the process to reach steady-state, consequently it is also less consistent than USE_BARRIERS = 1 due to the asynchronous updates.