# Name: Nicholas Fotinakes

# Date: 2022-04-05

# Title: Lab 5 - Lab Report: Notes, Observations, & Questions

# Description: This pdf contains observations, answers, and test results for Lab 5

Questions:

Step 1:

Explain what happens when you run this threadHello.c program; can you list how many threads are created and what values of i are passed? Do you get the same result if you run it multiple times? What if you are also running some other demanding processes (e.g., compiling a big program, playing a Flash game on a website, or watching streaming video) when you run this program?

Ten threads are created in addition to the main thread as we can see defined by the NTHREADS constant. These threads are created within a loop and set to the go() function to print the thread and iteration id before returning, joining in another loop where the main thread waits for all threads to complete before finishing. In this step, we have indeterminate results when trying to pass the 'i' iteration number for the thread, as we are not controlling the synchronization of shared data of each thread, meaning a race condition can occur between threads when accessing this variable(address of this variable). When running multiple times, we get different results and sometimes multiple instances of the same iteration number. When other demanding processes are running, I still got similar indeterminate results, but perhaps this would be slower or different if there were even more threads or demanding processes running.

The function go() has the parameter arg passed a local variable. Are these variables per-thread or shared state? Where does the compiler store these variables' states?

The parameter arg being passed in the go() function is a pointer/address to the 'i' iteration variable. The calling of this function creates space in the particular calling threads stack, and a copy of the pointer (arg) that is passed. While this is a copy that is created, it still references the original address of 'i' which means the data can be accessed across threads.

The main() has local variable i. Is this variable per-thread or shared state? Where does the compiler store this variable?

The i variable is declared in main and stored in the stack of the main. Because each thread created is using the address of this variable in memory to pass to go(), they are sharing this variable address space without synchronization or protection from the other threads.

Write down your observations. You probably have seen that there is a bug in the program, where threads may print same values of i. Why?

As explained earlier, we see that each thread created is using the same memory address of the 'i' variable declared in main. This means this data is not protected leading to indeterminate results. Because each thread is sharing this address in memory, it could be altered or changed by a different thread before another has a chance to use it as intended. This leads to unpredictable results when each thread is passed to go() to print the thread information and iteration passed. By the time the scheduler prints this information, the variable 'i' could have already been altered by another loop.

Notes/Observations:

Step 1/Step 1 Fix

There are different ways to approach fixing the main bug of synchronization in Step 1. The way I decided was to create a separate array to store a copy of the value of i in that particular iteration, and pass that to the thread when created. This way that value will be protected and not change, meaning another thread will not access this address space. Another way to approach this could be to use malloc to dynamically allocate space for an integer on the heap for each thread to use for the iteration id. I initially was working with this approach, but I could not figure out how to free this memory within in the same scope (the for loop) that malloc was used. I was freeing it in the go() function, which after talking with Professor Lamble I learned this was bad program design and she approved the array style that I ended up using for this lab. Also, because dynamically allocating memory could lead to more complications if not handled correctly, I decided to stick with the array approach. Because we know the fixed number of threads to create, there may actually be no need to deal with dynamically allocated memory, because the number of threads is defined as a constant. A third approach could be the use of locks to give each thread mutual exclusion when working with i.

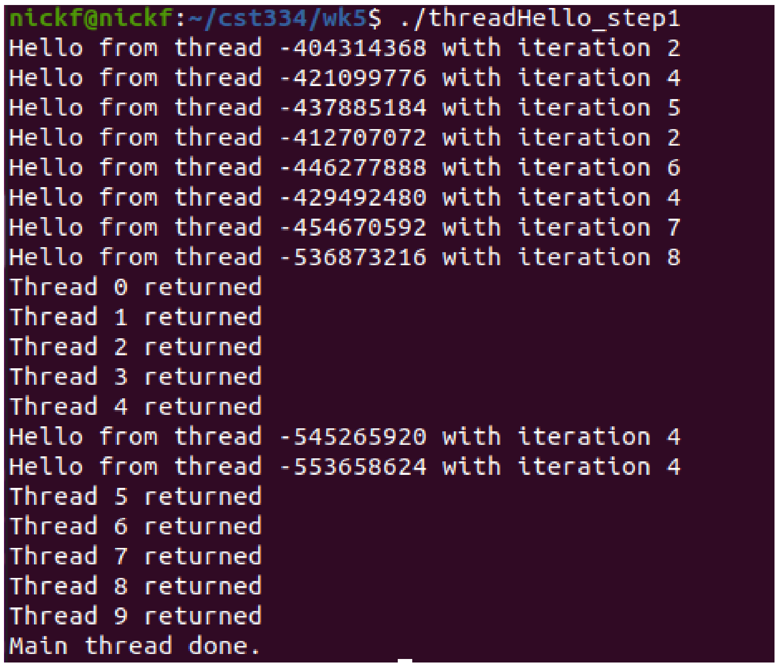
An additional bug present in Step 1 is dealing with the result of pthread_self(). By trying to print this as in integer, we could sometimes get a negative number. By using the pthread_t identifier and an unsigned long integer to print, we can prevent this from happening.

Matrix Multiplication

I used the same style of thread creation in the matrix multiplication of saving the iteration "id' for each thread in separate array. Using this, I can pass this to the multiply function for each thread to handle its dedicated portion to demonstrate using multiple threads to tackle a problem with deterministic results. I also decided to use random generated numbers between 0 and 2 to make use of a double. I rounded the printed output to 3 decimal places which makes the displayed values not necessarily exactly precise, but it demonstrates the multithreading sufficiently for the lab. While the array approach again works in this instance, if the program wants to use even larger numbers than 1024, it may make more sense to deal with dynamically allocated memory which could save the process from using an unnecessary amount of space on the stack.

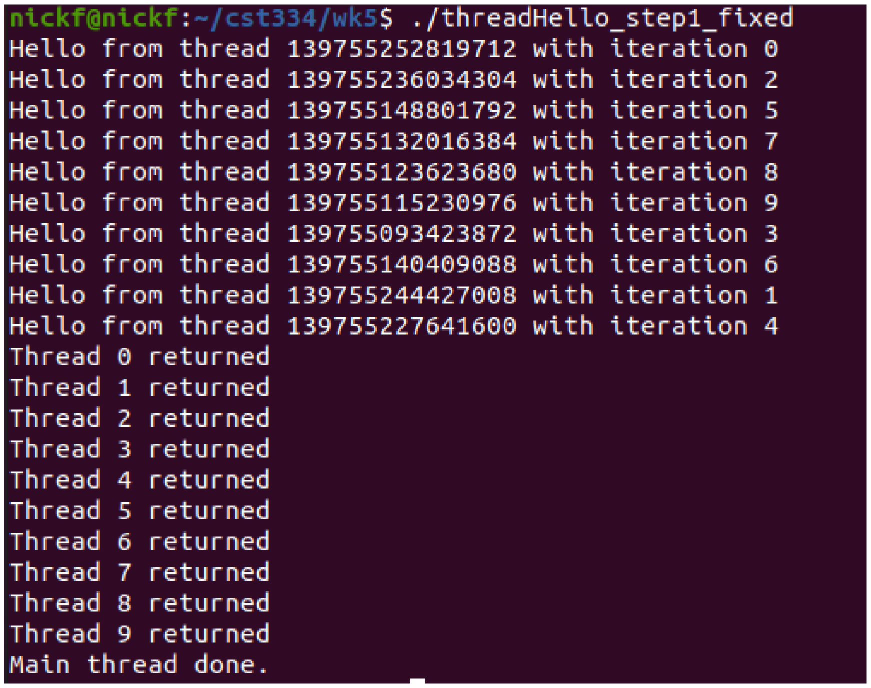
Test Verifications:

Step 1 (bug present with negative thread id and duplicate iteration values):



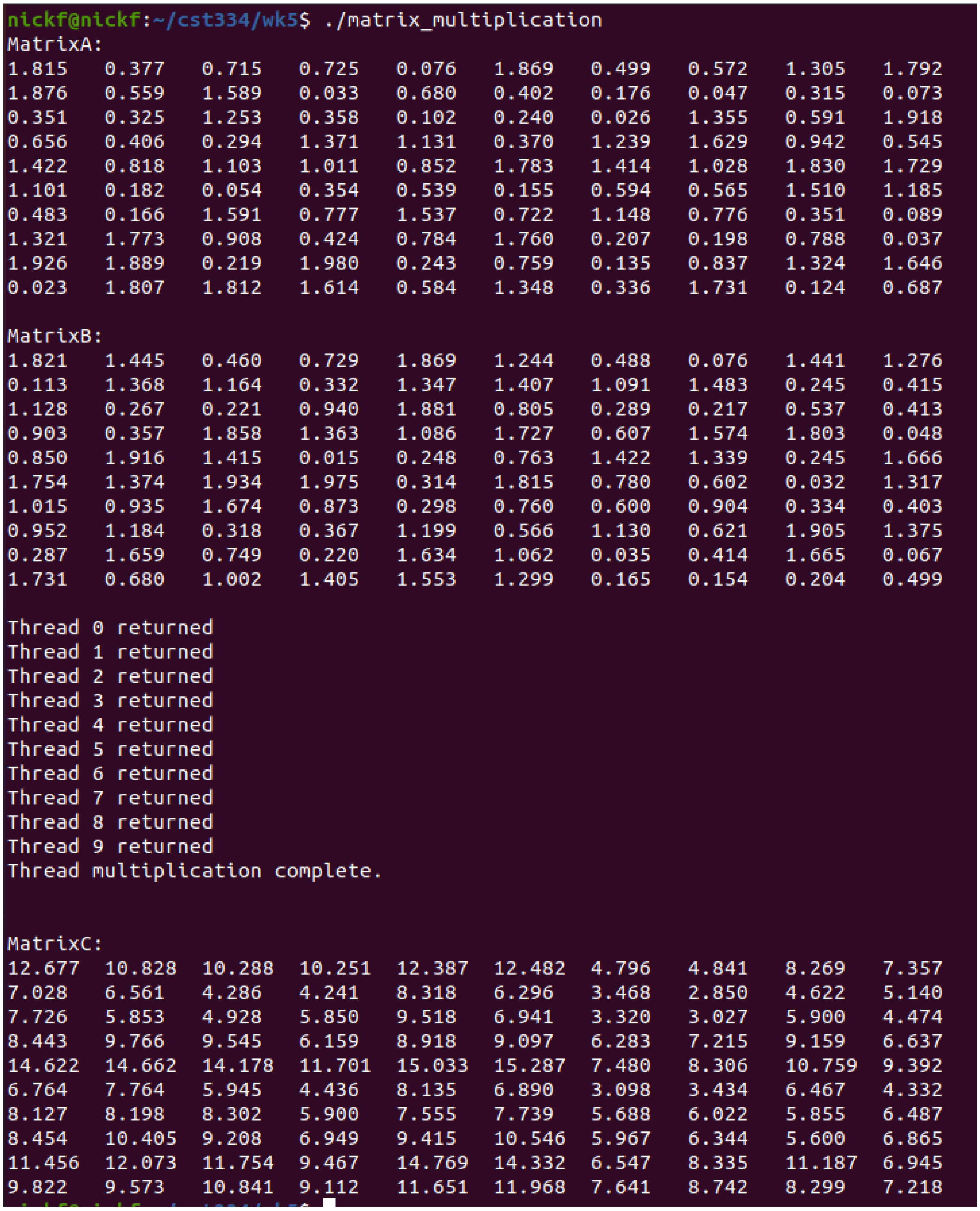
```
nickf@nickf:~/cst334/wk5$ ./threadHello_step1
Hello from thread -404314368 with iteration 2
Hello from thread -421099776 with iteration 4
Hello from thread -437885184 with iteration 5
Hello from thread -412707072 with iteration 2
Hello from thread -446277888 with iteration 6
Hello from thread -429492480 with iteration 4
Hello from thread -454670592 with iteration 7
Hello from thread -536873216 with iteration 8
Thread 0 returned
Thread 1 returned
Thread 2 returned
Thread 3 returned
Thread 4 returned
Hello from thread -545265920 with iteration 4
Hello from thread -553658624 with iteration 4
Thread 5 returned
Thread 6 returned
Thread 7 returned
Thread 8 returned
Thread 9 returned
Main thread done.
```

Step 1 Fixed (unique iteration ids 0-9 and positive thread id):



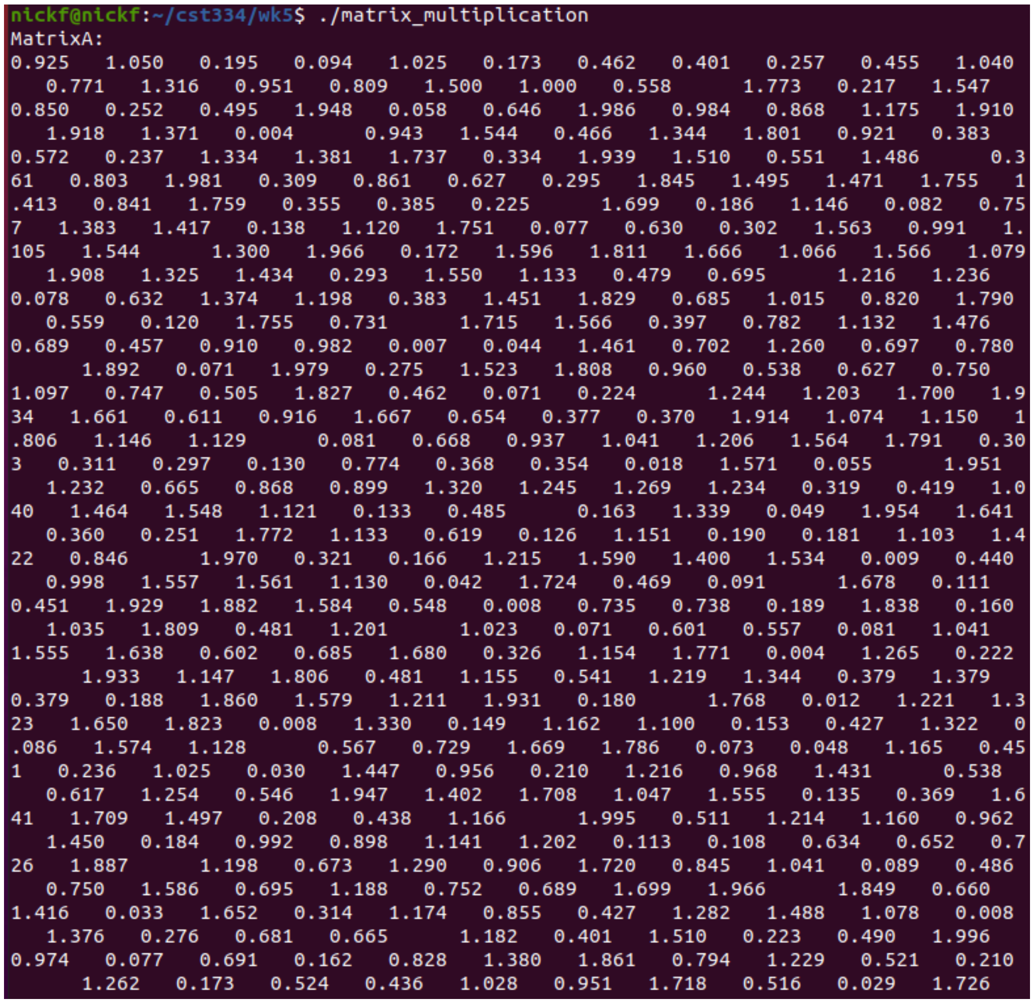
```
nickf@nickf:~/cst334/wk5$ ./threadHello_step1_fixed
Hello from thread 139755252819712 with iteration 0
Hello from thread 139755236034304 with iteration 2
Hello from thread 139755148801792 with iteration 5
Hello from thread 139755132016384 with iteration 7
Hello from thread 139755123623680 with iteration 8
Hello from thread 139755115230976 with iteration 9
Hello from thread 139755093423872 with iteration 3
Hello from thread 139755140409088 with iteration 6
Hello from thread 139755244427008 with iteration 1
Hello from thread 139755227641600 with iteration 4
Thread 0 returned
Thread 1 returned
Thread 2 returned
Thread 3 returned
Thread 4 returned
Thread 5 returned
Thread 6 returned
Thread 7 returned
Thread 8 returned
Thread 9 returned
Main thread done.
```

Matrix 10x10:



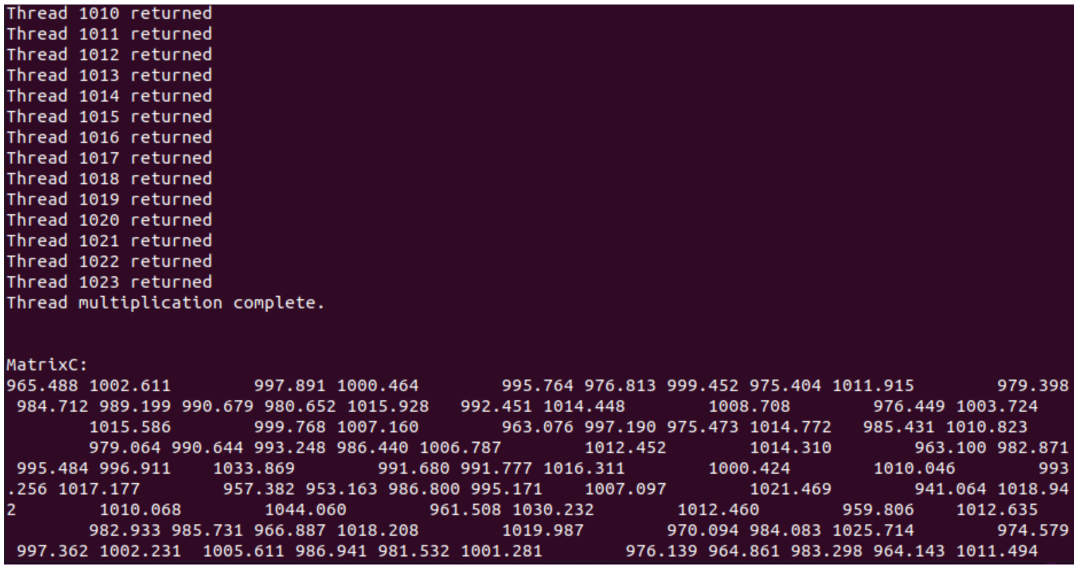
```
nickf@nickf:~/cst334/wk5$ ./matrix_multiplication
MatrixA:
1.815   0.377   0.715   0.725   0.076   1.869   0.499   0.572   1.305   1.792
1.876   0.559   1.589   0.033   0.680   0.402   0.176   0.047   0.315   0.073
0.351   0.325   1.253   0.358   0.102   0.240   0.026   1.355   0.591   1.918
0.656   0.406   0.294   1.371   1.131   0.370   1.239   1.629   0.942   0.545
1.422   0.818   1.103   1.011   0.852   1.783   1.414   1.028   1.830   1.729
1.101   0.182   0.054   0.354   0.539   0.155   0.594   0.565   1.510   1.185
0.483   0.166   1.591   0.777   1.537   0.722   1.148   0.776   0.351   0.089
1.321   1.773   0.908   0.424   0.784   1.760   0.207   0.198   0.788   0.037
1.926   1.889   0.219   1.980   0.243   0.759   0.135   0.837   1.324   1.646
0.023   1.807   1.812   1.614   0.584   1.348   0.336   1.731   0.124   0.687

MatrixB:
1.821   1.445   0.460   0.729   1.869   1.244   0.488   0.076   1.441   1.276
0.113   1.368   1.164   0.332   1.347   1.407   1.091   1.483   0.245   0.415
1.128   0.267   0.221   0.940   1.881   0.805   0.289   0.217   0.537   0.413
0.903   0.357   1.858   1.363   1.086   1.727   0.607   1.574   1.803   0.048
0.850   1.916   1.415   0.015   0.248   0.763   1.422   1.339   0.245   1.666
1.754   1.374   1.934   1.975   0.314   1.815   0.780   0.602   0.032   1.317
1.015   0.935   1.674   0.873   0.298   0.760   0.600   0.904   0.334   0.403
0.952   1.184   0.318   0.367   1.199   0.566   1.130   0.621   1.905   1.375
0.287   1.659   0.749   0.220   1.634   1.062   0.035   0.414   1.665   0.067
1.731   0.680   1.002   1.405   1.553   1.299   0.165   0.154   0.204   0.499

Thread 0 returned
Thread 1 returned
Thread 2 returned
Thread 3 returned
Thread 4 returned
Thread 5 returned
Thread 6 returned
Thread 7 returned
Thread 8 returned
Thread 9 returned
Thread multiplication complete.


MatrixC:
12.677  10.828  10.288  10.251  12.387  12.482  4.796   4.841   8.269   7.357
7.028   6.561   4.286   4.241   8.318   6.296   3.468   2.850   4.622   5.140
7.726   5.853   4.928   5.850   9.518   6.941   3.320   3.027   5.900   4.474
8.443   9.766   9.545   6.159   8.918   9.097   6.283   7.215   9.159   6.637
14.622  14.662  14.178  11.701  15.033  15.287  7.480   8.306   10.759  9.392
6.764   7.764   5.945   4.436   8.135   6.890   3.098   3.434   6.467   4.332
8.127   8.198   8.302   5.900   7.555   7.739   5.688   6.022   5.855   6.487
8.454   10.405  9.208   6.949   9.415   10.546  5.967   6.344   5.600   6.865
11.456  12.073  11.754  9.467   14.769  14.332  6.547   8.335   11.187  6.945
9.822   9.573   10.841  9.112   11.651  11.968  7.641   8.742   8.299   7.218
```

Matrix 1024x1024 (Start of execution):



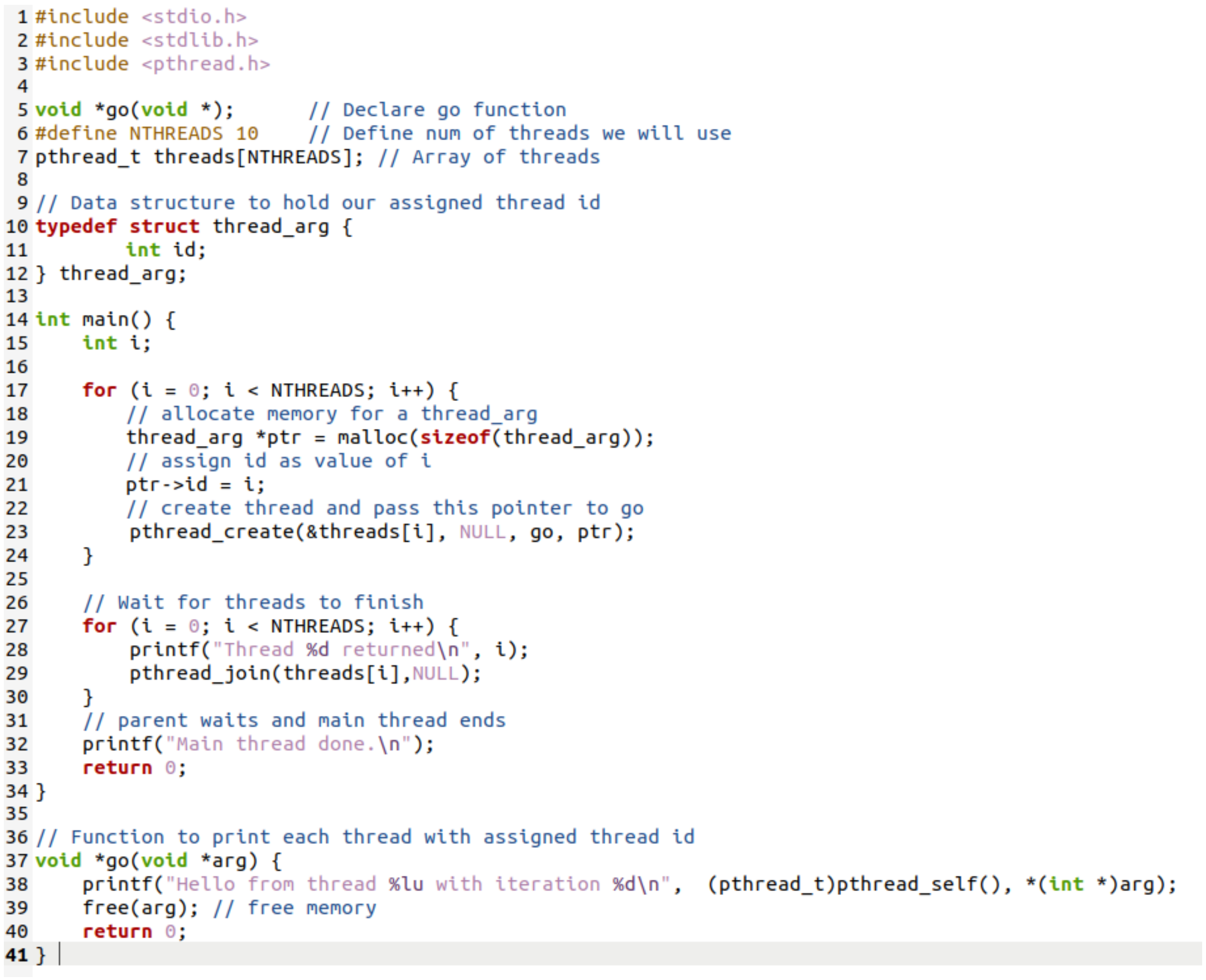
```
nickf@nickf:~/cst334/wk5$ ./matrix_multiplication
MatrixA:
0.925   1.050   0.195   0.094   1.025   0.173   0.462   0.401   0.257   0.455   1.040
   0.771   1.316   0.951   0.809   1.500   1.000   0.558       1.773   0.217   1.547
0.850   0.252   0.495   1.948   0.058   0.646   1.986   0.984   0.868   1.175   1.910
   1.918   1.371   0.004       0.943   1.544   0.466   1.344   1.801   0.921   0.383
0.572   0.237   1.334   1.381   1.737   0.334   1.939   1.510   0.551   1.486       0.3
61   0.803   1.981   0.309   0.861   0.627   0.295   1.845   1.495   1.471   1.755   1
.413   0.841   1.759   0.355   0.385   0.225       1.699   0.186   1.146   0.082   0.75
7   1.383   1.417   0.138   1.120   1.751   0.077   0.630   0.302   1.563   0.991   1.
105   1.544       1.300   1.966   0.172   1.596   1.811   1.666   1.066   1.566   1.079
   1.908   1.325   1.434   0.293   1.550   1.133   0.479   0.695       1.216   1.236
0.078   0.632   1.374   1.198   0.383   1.451   1.829   0.685   1.015   0.820   1.790
   0.559   0.120   1.755   0.731       1.715   1.566   0.397   0.782   1.132   1.476
0.689   0.457   0.910   0.982   0.007   0.044   1.461   0.702   1.260   0.697   0.780
      1.892   0.071   1.979   0.275   1.523   1.808   0.960   0.538   0.627   0.750
1.097   0.747   0.505   1.827   0.462   0.071   0.224       1.244   1.203   1.700   1.9
34   1.661   0.611   0.916   1.667   0.654   0.377   0.370   1.914   1.074   1.150   1
.806   1.146   1.129       0.081   0.668   0.937   1.041   1.206   1.564   1.791   0.30
3   0.311   0.297   0.130   0.774   0.368   0.354   0.018   1.571   0.055       1.951
   1.232   0.665   0.868   0.899   1.320   1.245   1.269   1.234   0.319   0.419   1.0
40   1.464   1.548   1.121   0.133   0.485       0.163   1.339   0.049   1.954   1.641
   0.360   0.251   1.772   1.133   0.619   0.126   1.151   0.190   0.181   1.103   1.4
22   0.846       1.970   0.321   0.166   1.215   1.590   1.400   1.534   0.009   0.440
   0.998   1.557   1.561   1.130   0.042   1.724   0.469   0.091       1.678   0.111
0.451   1.929   1.882   1.584   0.548   0.008   0.735   0.738   0.189   1.838   0.160
   1.035   1.809   0.481   1.201       1.023   0.071   0.601   0.557   0.081   1.041
1.555   1.638   0.602   0.685   1.680   0.326   1.154   1.771   0.004   1.265   0.222
      1.933   1.147   1.806   0.481   1.155   0.541   1.219   1.344   0.379   1.379
0.379   0.188   1.860   1.579   1.211   1.931   0.180       1.768   0.012   1.221   1.3
23   1.650   1.823   0.008   1.330   0.149   1.162   1.100   0.153   0.427   1.322   0
.086   1.574   1.128       0.567   0.729   1.669   1.786   0.073   0.048   1.165   0.45
1   0.236   1.025   0.030   1.447   0.956   0.210   1.216   0.968   1.431       0.538
   0.617   1.254   0.546   1.947   1.402   1.708   1.047   1.555   0.135   0.369   1.6
41   1.709   1.497   0.208   0.438   1.166       1.995   0.511   1.214   1.160   0.962
   1.450   0.184   0.992   0.898   1.141   1.202   0.113   0.108   0.634   0.652   0.7
26   1.887       1.198   0.673   1.290   0.906   1.720   0.845   1.041   0.089   0.486
   0.750   1.586   0.695   1.188   0.752   0.689   1.699   1.966       1.849   0.660
1.416   0.033   1.652   0.314   1.174   0.855   0.427   1.282   1.488   1.078   0.008
   1.376   0.276   0.681   0.665       1.182   0.401   1.510   0.223   0.490   1.996
0.974   0.077   0.691   0.162   0.828   1.380   1.861   0.794   1.229   0.521   0.210
      1.262   0.173   0.524   0.436   1.028   0.951   1.718   0.516   0.029   1.726
```

1024x1024 (Printing of Matrix C):



```
Thread 1010 returned
Thread 1011 returned
Thread 1012 returned
Thread 1013 returned
Thread 1014 returned
Thread 1015 returned
Thread 1016 returned
Thread 1017 returned
Thread 1018 returned
Thread 1019 returned
Thread 1020 returned
Thread 1021 returned
Thread 1022 returned
Thread 1023 returned
Thread multiplication complete.


MatrixC:
965.488 1002.611        997.891 1000.464        995.764 976.813 999.452 975.404 1011.915        979.398
 984.712 989.199 990.679 980.652 1015.928   992.451 1014.448        1008.708        976.449 1003.724
        1015.586        999.768 1007.160        963.076 997.190 975.473 1014.772   985.431 1010.823
        979.064 990.644 993.248 986.440 1006.787        1012.452        1014.310        963.100 982.871
 995.484 996.911    1033.869        991.680 991.777 1016.311        1000.424        1010.046        993
.256 1017.177        957.382 953.163 986.800 995.171    1007.097        1021.469        941.064 1018.94
2        1010.068        1044.060        961.508 1030.232        1012.460        959.806    1012.635
        982.933 985.731 966.887 1018.208        1019.987        970.094 984.083 1025.714        974.579
 997.362 1002.231  1005.611 986.941 981.532 1001.281        976.139 964.861 983.298 964.143 1011.494
```

Alternate Approach: This was an alternative approach I came up with for this Lab for fixing Step 1 bugs and the matrix multiplication using malloc and a structure to hold thread ids. While this approach functioned properly with the desired output, I could not figure out how to free the dynamically allocated memory outside of the for loop where malloc was called. It still worked freeing this within the go() function, but Professor Lamble indicated this is not proper design.



```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *go(void *);       // Declare go function
#define NTHREADS 10     // Define num of threads we will use
pthread_t threads[NTHREADS]; // Array of threads

// Data structure to hold our assigned thread id
typedef struct thread_arg {
        int id;
} thread_arg;

int main() {
    int i;

    for (i = 0; i < NTHREADS; i++) {
        // allocate memory for a thread_arg
        thread_arg *ptr = malloc(sizeof(thread_arg));
        // assign id as value of i
        ptr->id = i;
        // create thread and pass this pointer to go
        pthread_create(&threads[i], NULL, go, ptr);
    }

    // Wait for threads to finish
    for (i = 0; i < NTHREADS; i++) {
        printf("Thread %d returned\n", i);
        pthread_join(threads[i],NULL);
    }
    // parent waits and main thread ends
    printf("Main thread done.\n");
    return 0;
}

// Function to print each thread with assigned thread id
void *go(void *arg) {
    printf("Hello from thread %lu with iteration %d\n",  (pthread_t)pthread_self(), *(int *)arg);
    free(arg); // free memory
    return 0;
}
```