Developing Soft and Parallel Programming Skills Using Project-Based Learning

Spring 2020

The Commuters
Alaya Shack, Miguel Romo, Arteen Ghafourikia, Andre Nguyenphuc, Joan Galicia

Planning and Scheduling: Arteen Ghafourikia

Assignee Name	Email	Task	Duration (hours)	Dependency	Due Date	Note
Arteen Ghafourikia	aghafourikia1@student.gsu.edu	Do the Planning and Scheduling. Complete individual parallel programming skills task and ARM assembly programming task.	9 hours	(none)	03/25	Review report for grammatical/ spelling errors
Miguel Romo	mromo1@student.gsu.edu	Get the report formatted correctly (fonts, page numbers, and sections) .Complete individual parallel programming skills task and ARM assembly programming task.	7hours	(none)	03/25	Review report for grammatical/ spelling errors. 24 hours before the due date, please have the report ready for all team members to review.
Joan Galicia (Team Coordinator)	jgalicia2@student.gsu.edu	Team Coordinator. Edit the video and include the link in the report. Complete individual	7 hours	(none)	03/25	Review report for grammatical/ spelling errors.

		parallel programming skills task and ARM assembly programming task.				
Alaya Shack	ashack1@student.gsu.edu	Complete columns in Github. Complete individual parallel programming skills task and ARM assembly programming task.	8 hours	Github	03/25	Review report for grammatical/ spelling errors.
Andre Nguyenphuc	anguyenphuc1@student.gsu.edu	Facilitator and getting everyone together. Complete individual parallel programming skills task and ARM assembly programming task.	8 hours	(none)	03/25	Review report for grammatical/ spelling errors.

Parallel Programming Skills Foundation: Alaya Shack

- ➔ Race Condition:
  - ◆ What is race condition?
    - Race condition, also known as a race hazard, is the behavior of a system where the output is dependent on the timing of oher uncontrollable events.
  - ◆ Why is race condition difficult to reproduce and debug?
    - Race conditions are difficult to reproduce and debug because the end result is nondeterministic and is dependent on the relative timing between interfering threads. Also, when running in debugging mode or when attaching a debugger, problems that occur in production systems sometimes disappear.
  - ◆ How can it be fixed? Provide an example from your Project_A3 (see spmd2.c)
    - Race conditions can be fixed by carefully crafting the code, this includes sometimes making sure that variables have full declarations like how we did with spmd2.c in project 2.
- ➔ Summarize the Parallel programming Patterns section in the "Introduction to Parallel Computing_3.pdf" (two pages).
  - ◆ Parallel programs have many patterns, but they are grouped into two main groups, which are strategies and concurrent execution mechanisms. The different types of strategies that should be considered when writing a program are algorithmic strategies and implementation strategies. Parallel algorithmic strategies mostly deal with deciding what tasks can be done simultaneously. Parallel programs use several implementation strategies that contribute to the overall structure of the program, whereas others deal with how data is computed. Concurrent execution mechanisms have the following two main categories: Process/Thread control patterns and coordination patterns. Process/Thread control patterns dictate how processing units are controlled at runtime, but coordination patterns set up how tasks coordinate. There are also two major coordination patterns: message passing, which occurs between concurrent processes, and mutual exclusion, which occurs between threads executing on a single shared memory system.
- ➔ In the section "Categorizing Patterns in the "Introduction to Parallel Computing_3.pdf" **compare** the following:
  - ◆ Collective synchronization (barrier) with Collective communication (reduction)
    - Collective synchronization(barrier) involves blocking processes until a specific synchronization point is met. With this pattern, it is said to function like a barrier because it will block a process until all other processes reach that point. However, collective communication (reduction) involves all processes reaching a specific point before

continuing execution. With this pattern, it acts as a reduction because one process collects data from the other processes.

- ◆ Master-worker with fork join
  - Master worker involves a master or a main process that is split up into different pieces that are then distributed to several worker processes. However, fork-join pattern is used to execute light weight processes and threads in parallel.

➔ Dependency: Using your own words and explanation, answer the following:

- ◆ Where can we find parallelism in programming?
  - We can find parallelism in programming from different viewpoints. However, from a programming viewpoint, we can find parallelism between statements, specifically statements that can be executed concurrently. Also, we can find parallelism at the block, loop, routine, and process level in larger-grained program statements, specifically with which ones can be executed concurrently.
- ◆ What is dependency and what are its types (provide one example for each)?
  - Dependency happens when an event relies on a previous operation to complete before it can perform its operation.
  - True(flow) dependence is when the second operation depends on the first operation. An example would be:
    - Statement 1: x=2
    - Statement 2: y=x
  - Output Dependence is also when the second operation is dependent on the first operation, but there is typically some type of operation involved with another variable.An example would be:
    - Statement 1: x=2*a
    - Statement 2: y=x/4
  - Anti-dependence is when the first operation is dependent on a later operation. An example of this would be:
    - Statement 1: x=y
    - Statement 2: y=2
- ◆ When a statement is dependent and when it is independent (provide two examples)?
  - Statements are dependent when the order of their execution matters or if the order of their execution affects the output.
    - An example of this would be statement 1: a=5 and statement 2: b=a*2. This example has dependency because b cannot be computed without a, and if a is changed b's computational output will be affected.

- A statement or statements are independent when the execution of the statements does not matter, that is, the execution of one statement has no affect on the other statement.
    - An example of this would be statement 1: a=2 and statement 2: b=3 are independent because no matter the order in which the statements are presented, the output will be the same.

◆ When can two statements be executed in parallel?
- Two statements can be executed in parallel when statement 1 and statement 2 have no dependencies(true,anti,output).

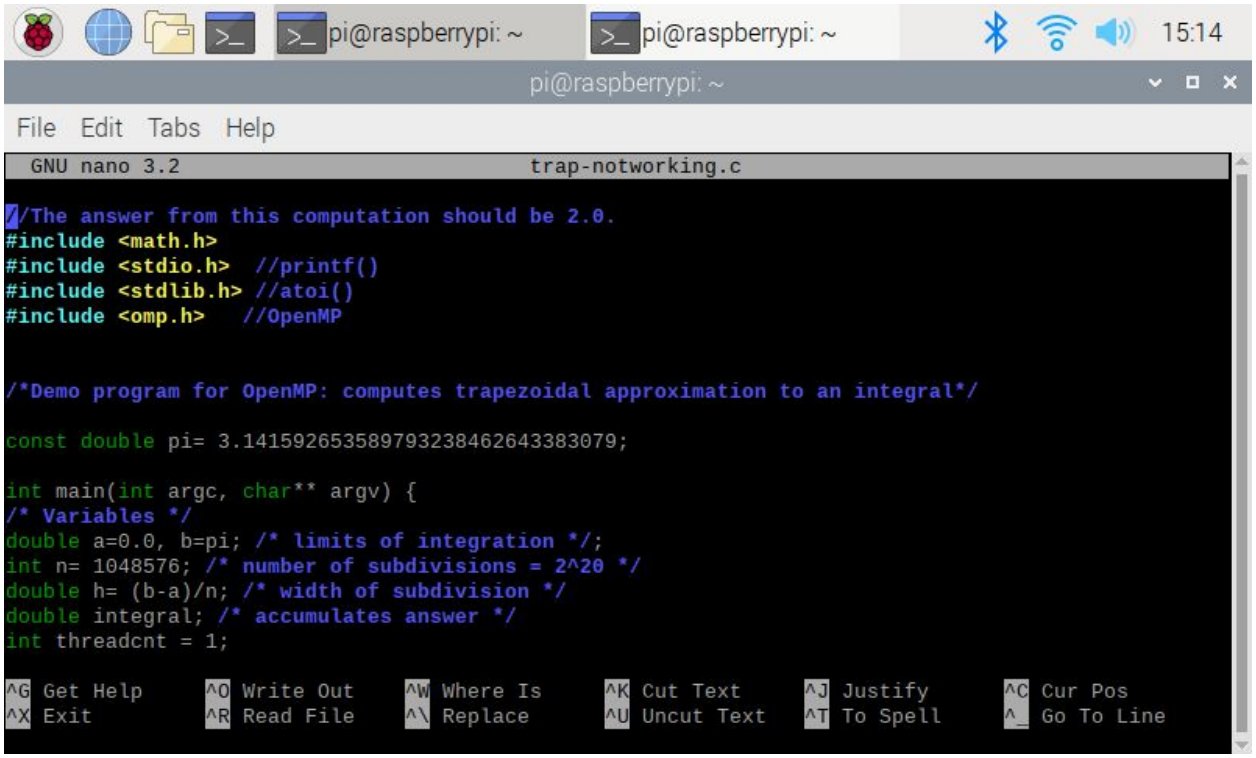
◆ How can dependency be removed?
- Dependency can be removed by revising the program, which can include rearranging or eliminating statements.

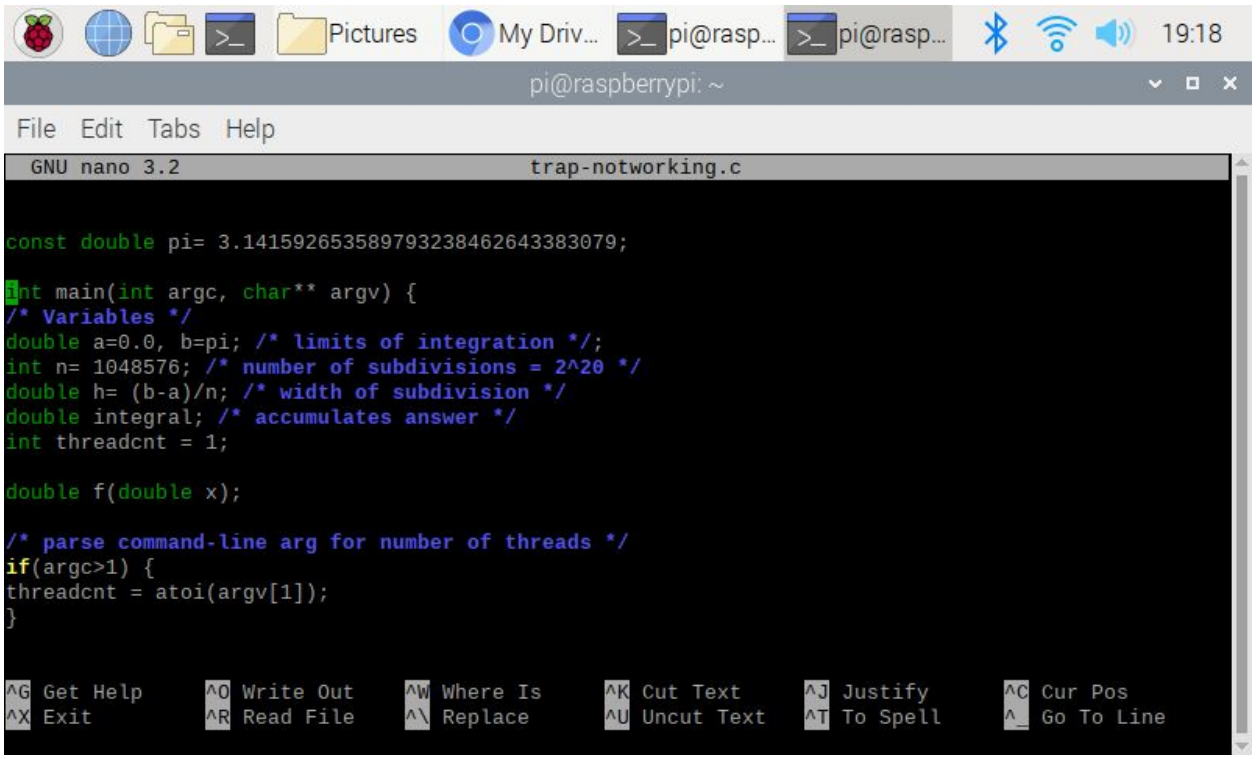
◆ How do we compute dependency for the following two loops and what type/s of dependency?
- We compute dependency by comparing the IN and OUT sets of each node. They are both defined as the set of variables that are used in a statement.
- The first loop is true (flow) dependence.
- The second loop is true (flow) dependence.

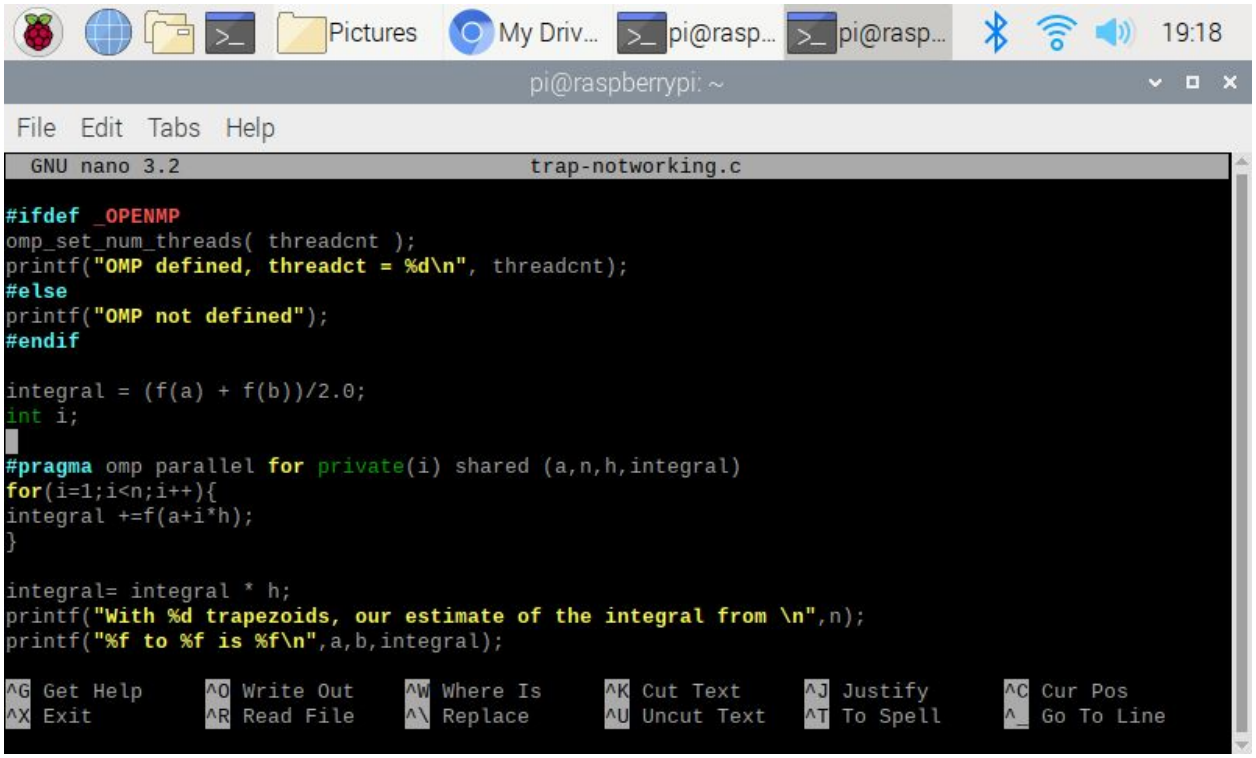
```
for(i=0;i<100;i++)          for(i=0;i<100;i++){
   S1: a[i]=i;                 S1:a[i]=i;
                               S2:b[i]=2*i;
                              }
```

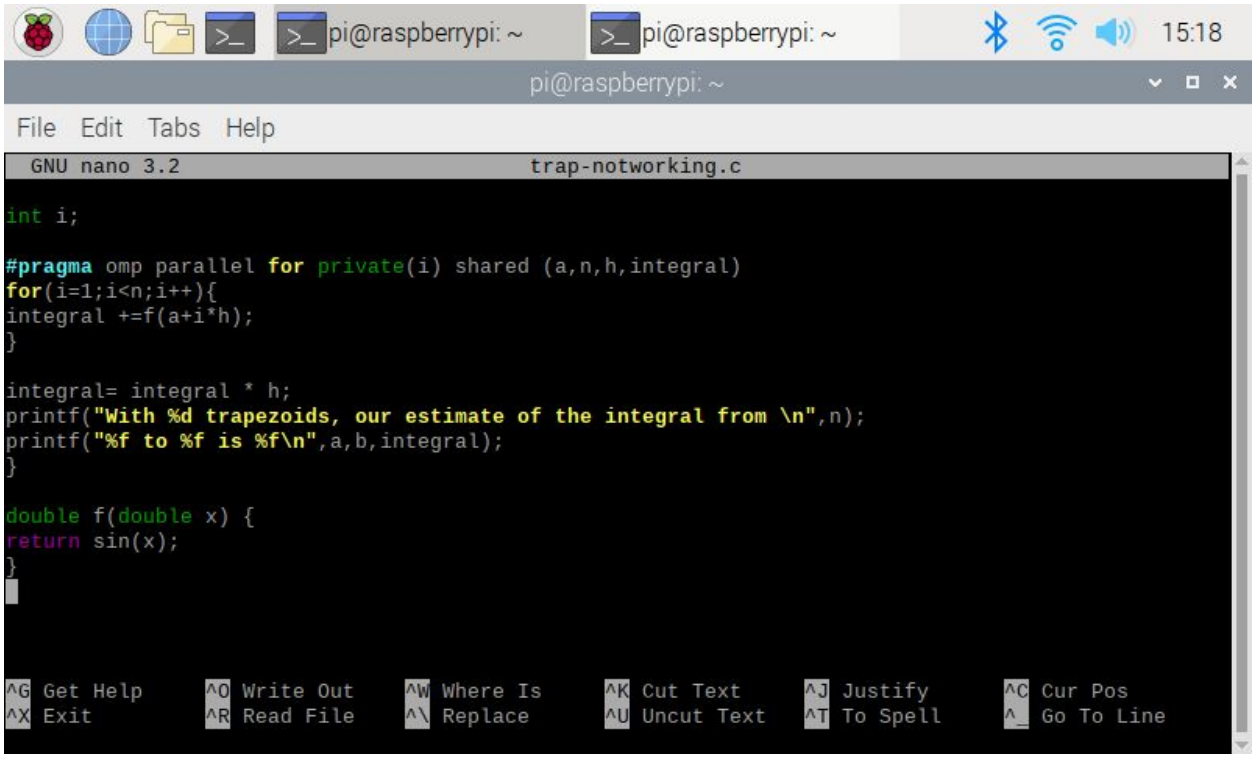
Parallel Programming Basics: Alaya Shack

Part 1


pi@raspberrypi: ~
pi@raspberrypi: ~
15:14
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 trap-notworking.c
//The answer from this computation should be 2.0.
#include <math.h>
#include <stdio.h> //printf()
#include <stdlib.h> //atoi()
#include <omp.h> //OpenMP
/*Demo program for OpenMP: computes trapezoidal approximation to an integral*/
const double pi= 3.14159265358979323846264338307 9;
int main(int argc, char** argv) {
/* Variables */
double a=0.0, b=pi; /* limits of integration */;
int n= 1048576; /* number of subdivisions = 2^20 */
double h= (b-a)/n; /* width of subdivision */
double integral; /* accumulates answer */
int threadcnt = 1;
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line

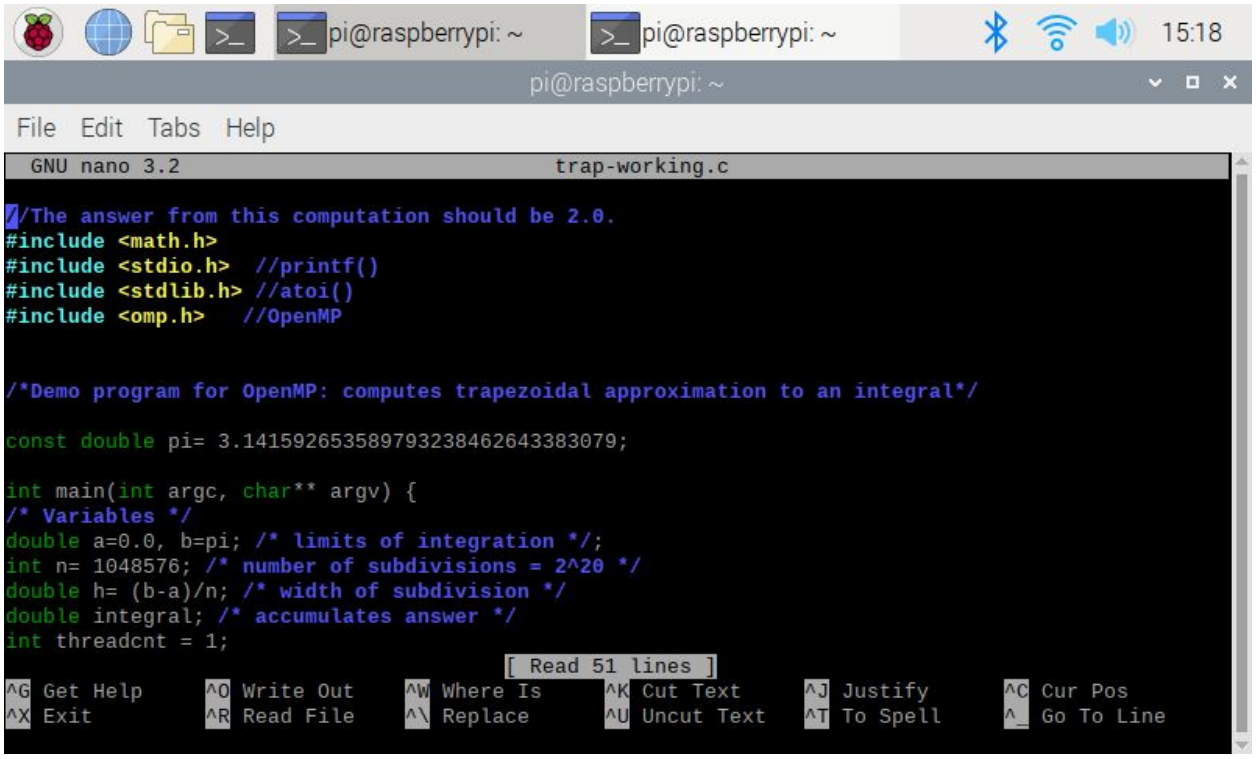

Pictures
My Driv...
pi@rasp...
pi@rasp...
19:18
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 trap-notworking.c
const double pi= 3.14159265358979323846264338307 9;
int main(int argc, char** argv) {
/* Variables */
double a=0.0, b=pi; /* limits of integration */;
int n= 1048576; /* number of subdivisions = 2^20 */
double h= (b-a)/n; /* width of subdivision */
double integral; /* accumulates answer */
int threadcnt = 1;
double f(double x);
/* parse command-line arg for number of threads */
if(argc>1) {
threadcnt = atoi(argv[1]);
}
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line

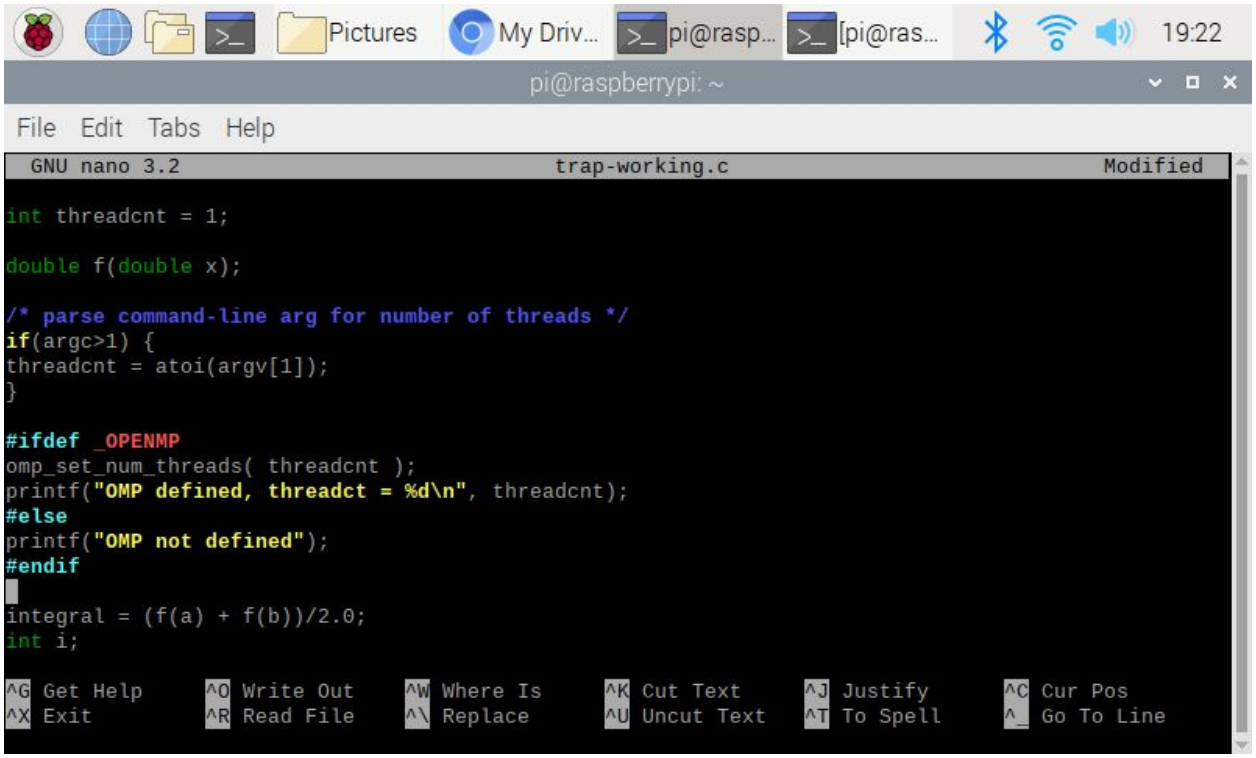


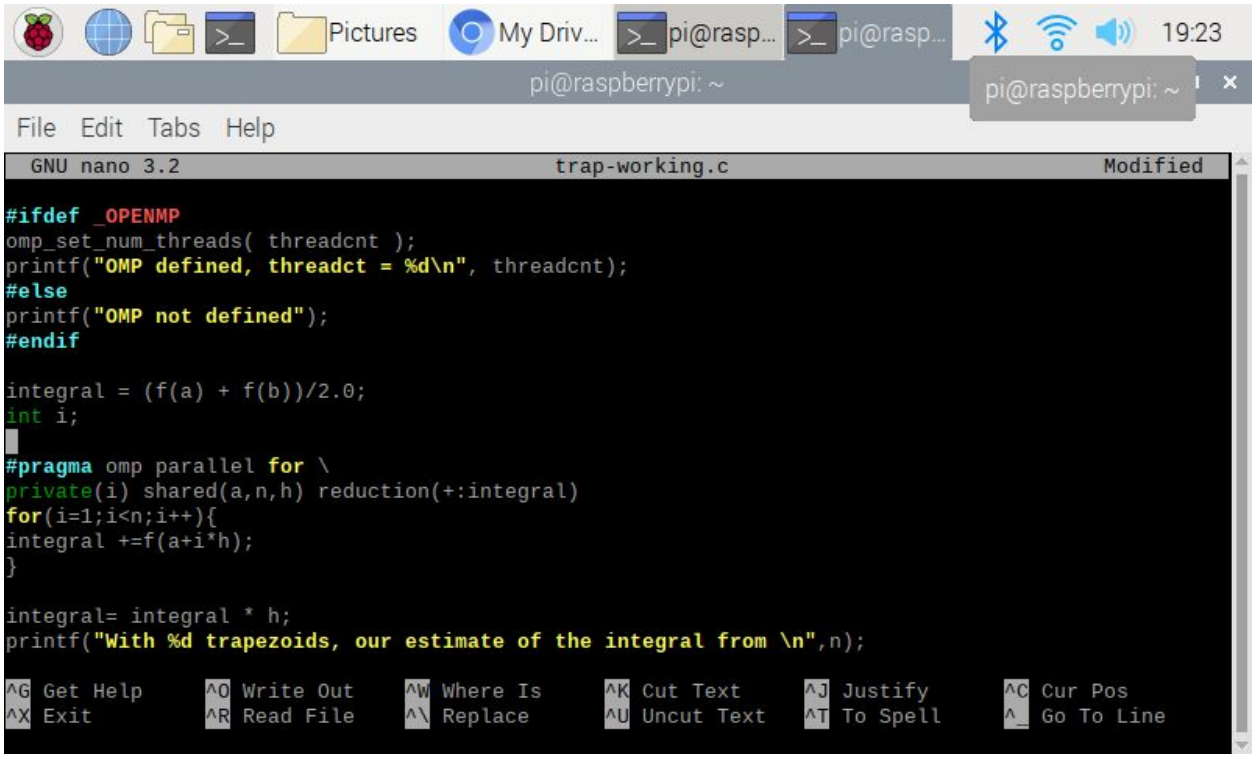


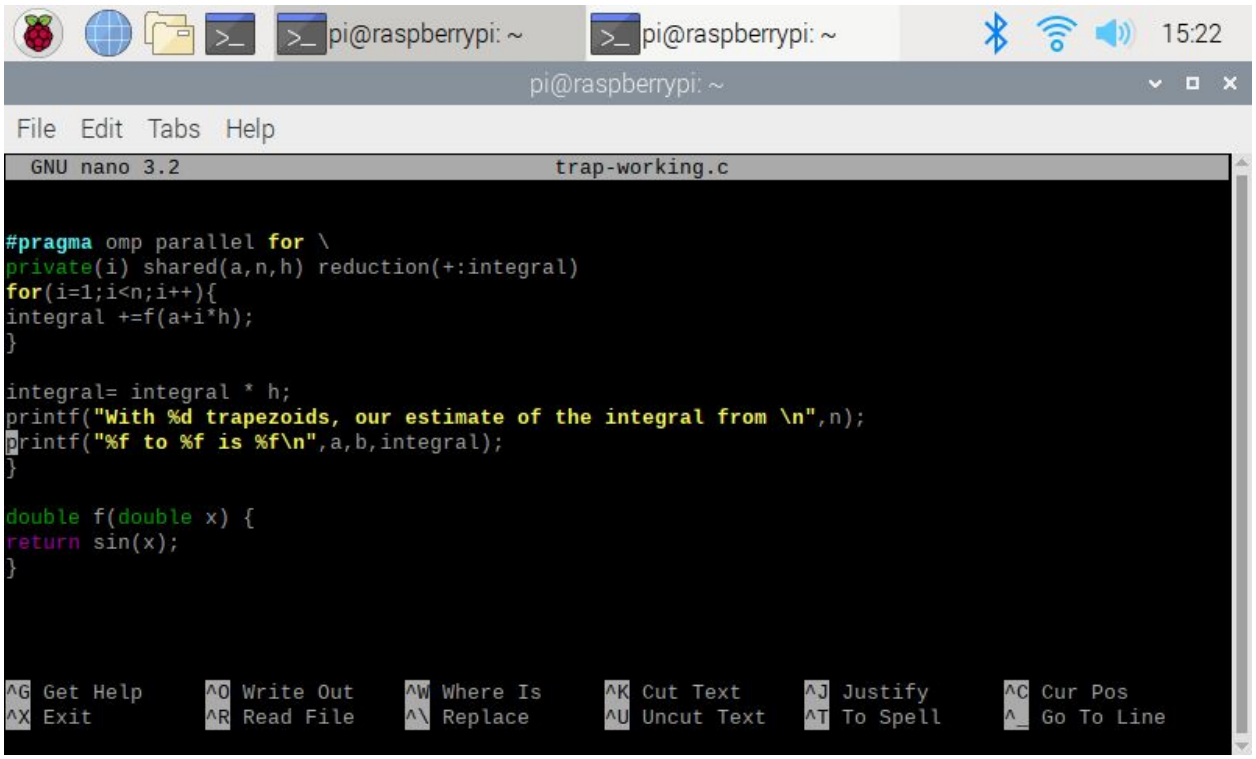
These four screenshots display the code for the program trap-notworking, which computes the integral of sin(x) on the interval 0 to pi using the trapezoidal rule. With this program, we use the "#include <math.h>" to include the math library functions because we are dealing with sin. This

code has a problem with the line that has “#pragma omp parallel for private(i) shared (a,n,h,integral)”. Also, with this code, variables are explicitly declared private or shared.

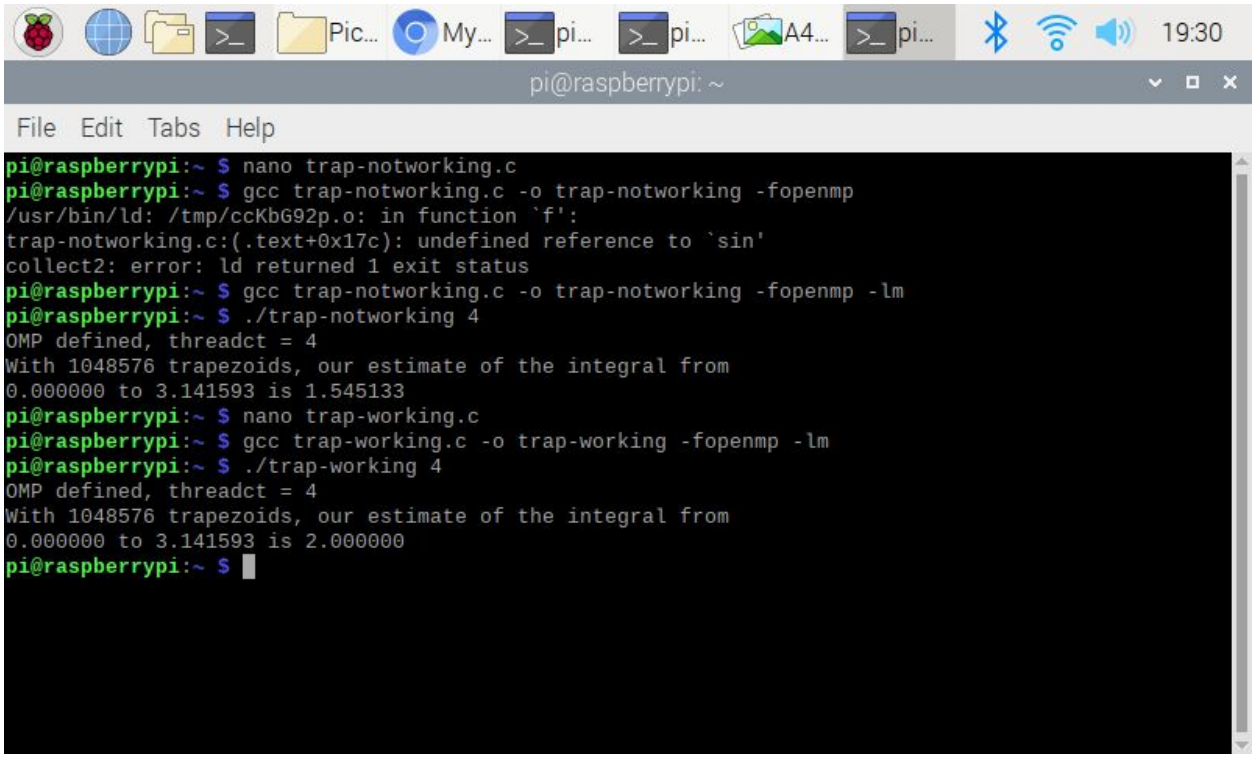





Pictures
My Driv...
pi@rasp...
pi@rasp...
19:23
pi@raspberrypi: ~
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 trap-working.c Modified
#ifdef _OPENMP
omp_set_num_threads(threadcnt);
printf("OMP defined, threadct = %d\n", threadcnt);
#else
printf("OMP not defined");
#endif
integral = (f(a) + f(b))/2.0;
int i;
#pragma omp parallel for \
private(i) shared(a,n,h) reduction(+:integral)
for(i=1;i<n;i++){
integral +=f(a+i*h);
}
integral= integral * h;
printf("With %d trapezoids, our estimate of the integral from \n",n);
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line

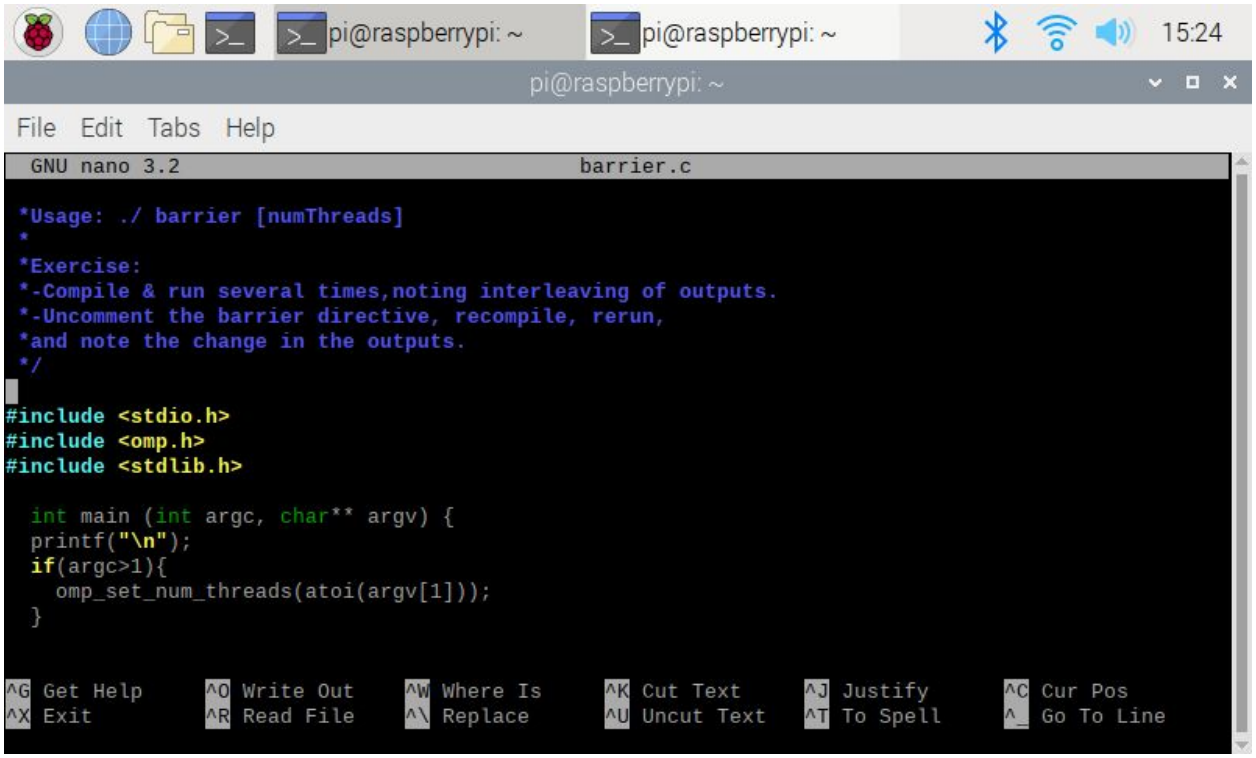

pi@raspberrypi: ~
pi@raspberrypi: ~
15:22
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 trap-working.c
#pragma omp parallel for \
private(i) shared(a,n,h) reduction(+:integral)
for(i=1;i<n;i++){
integral +=f(a+i*h);
}
integral= integral * h;
printf("With %d trapezoids, our estimate of the integral from \n",n);
printf("%f to %f is %f\n",a,b,integral);
}
double f(double x) {
return sin(x);
}
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line

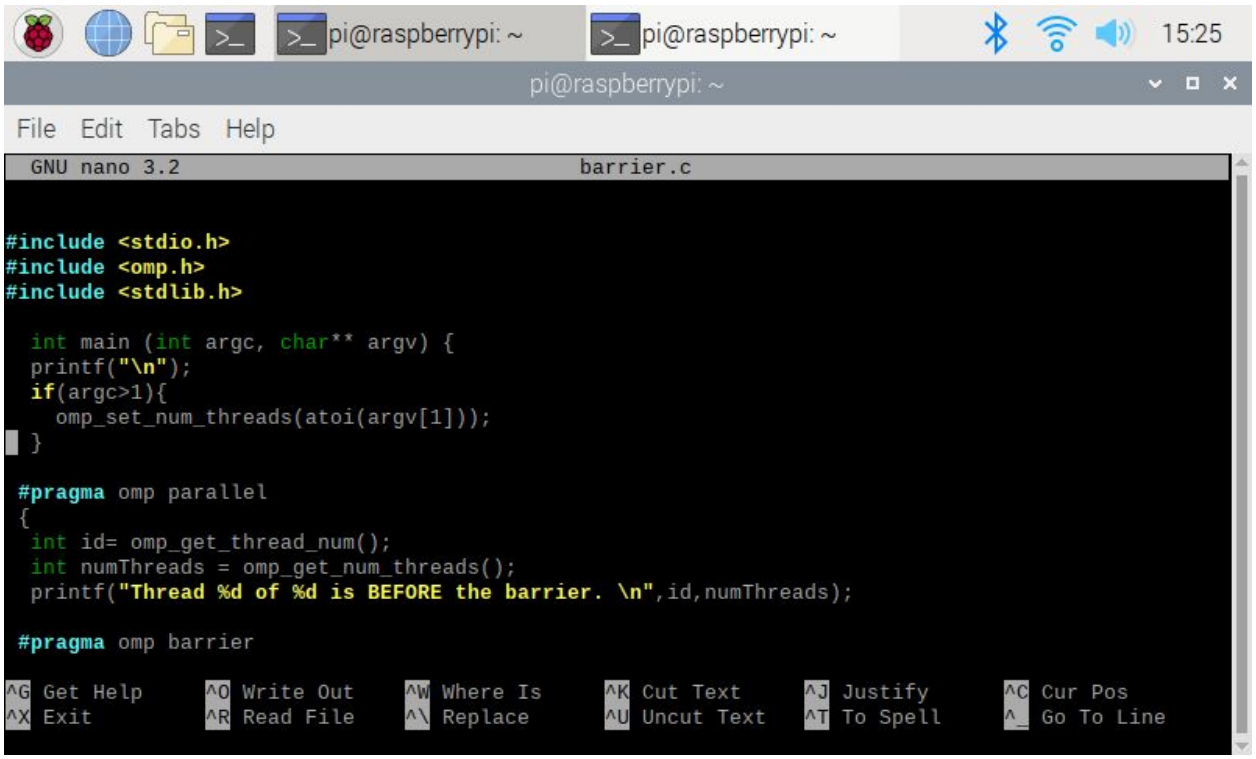
These four screenshots display the code for trap-working, which computes the integral of sin on the interval 0 to pi using the trapezoidal rule. This program is the same as the program mentioned above, but this program actually computes the correct answer for the integral. This is because we have the accumulator variable named integral, which indicates we need to include the reduction clause for that variable.

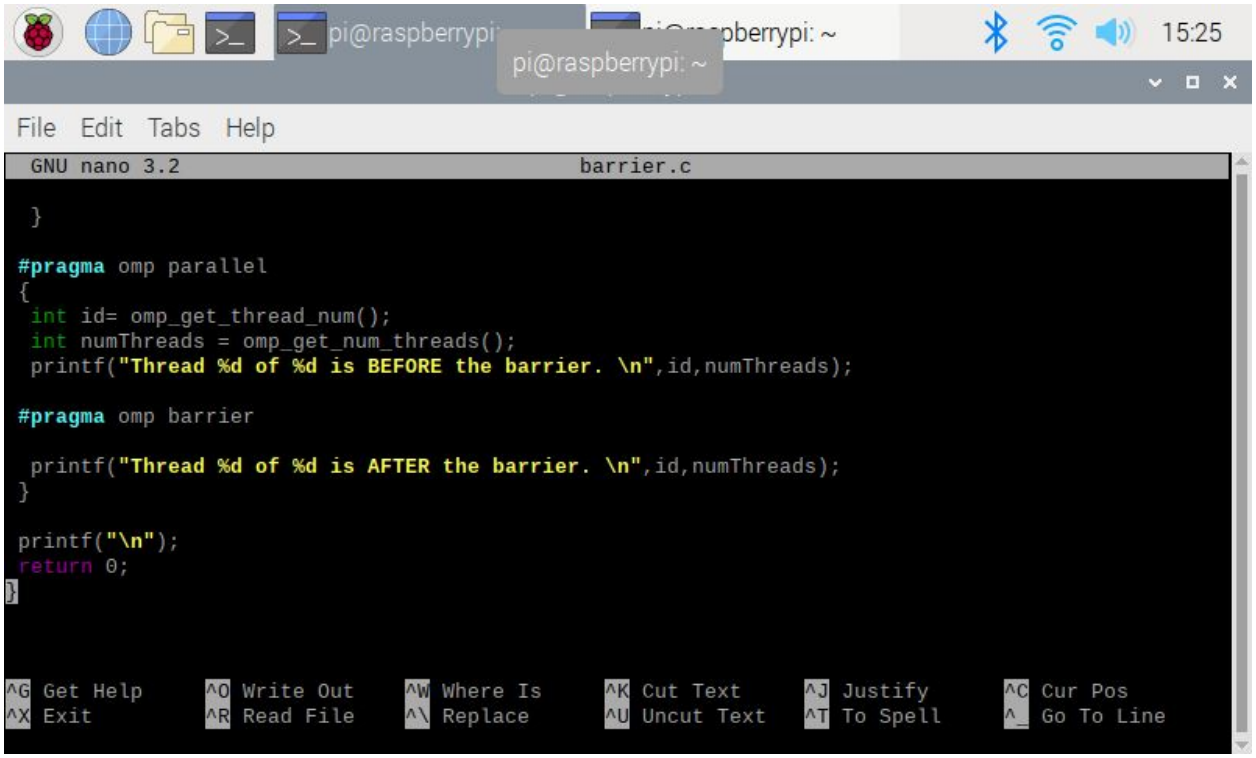


In this screenshot, I edited the trap-notworking, and then I made the program executable with the "gcc trap-notworking.c -o trap-notworking -fopenmp". However, once I used that command, I got an error that said undefined reference to sin, so then I researched the problem. Upon my research, I saw that "-lm" should be appended to the end of the command so that it can be linked with the math library. I appended "-lm", and there was no error. I then ran trap-working and trap-notworking to compare the results. Trap-notworking gave the incorrect answer of 1.545133, but trap-working gave the correct answer of 2.000000.

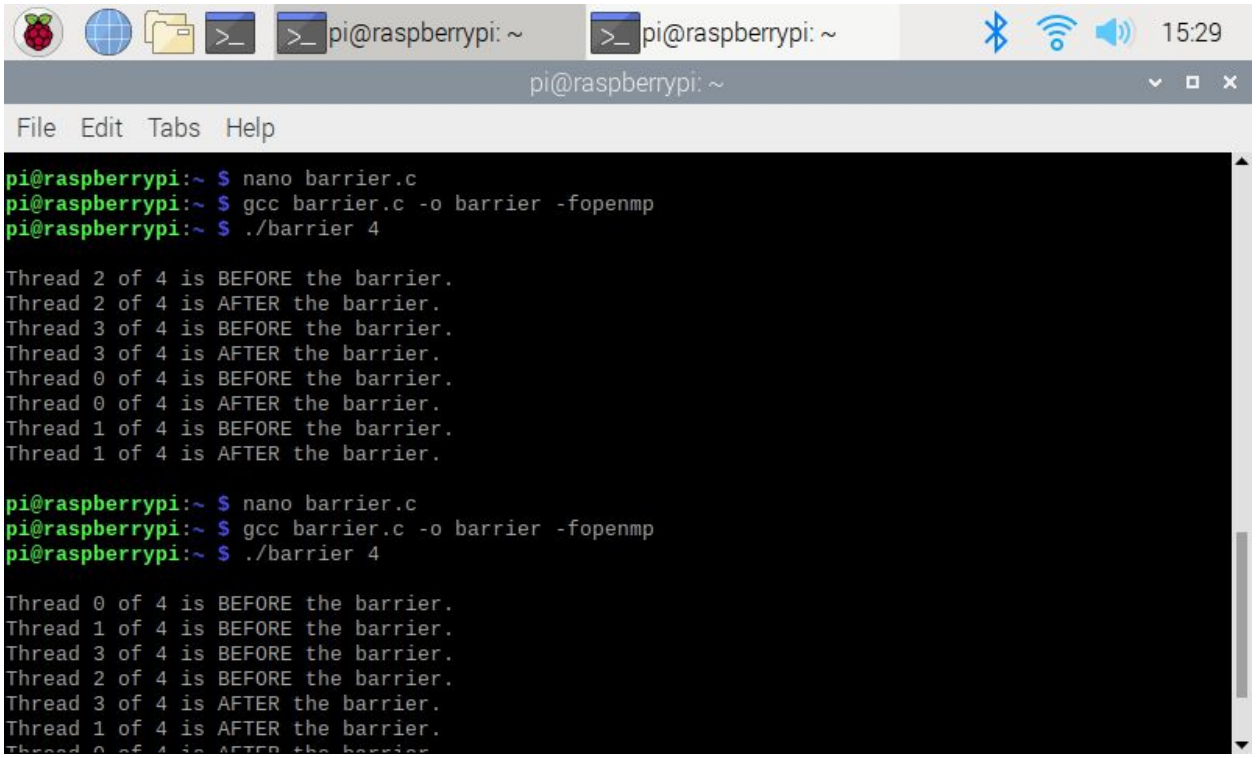
Part 2


pi@raspberrypi: ~
pi@raspberrypi: ~
15:24
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 barrier.c
*Usage: ./ barrier [numThreads]
*
*Exercise:
*-Compile & run several times,noting interleaving of outputs.
*-Uncomment the barrier directive, recompile, rerun,
*and note the change in the outputs.
*/
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
int main (int argc, char** argv) {
printf("\n");
if(argc>1){
omp_set_num_threads(atoi(argv[1]));
}
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line

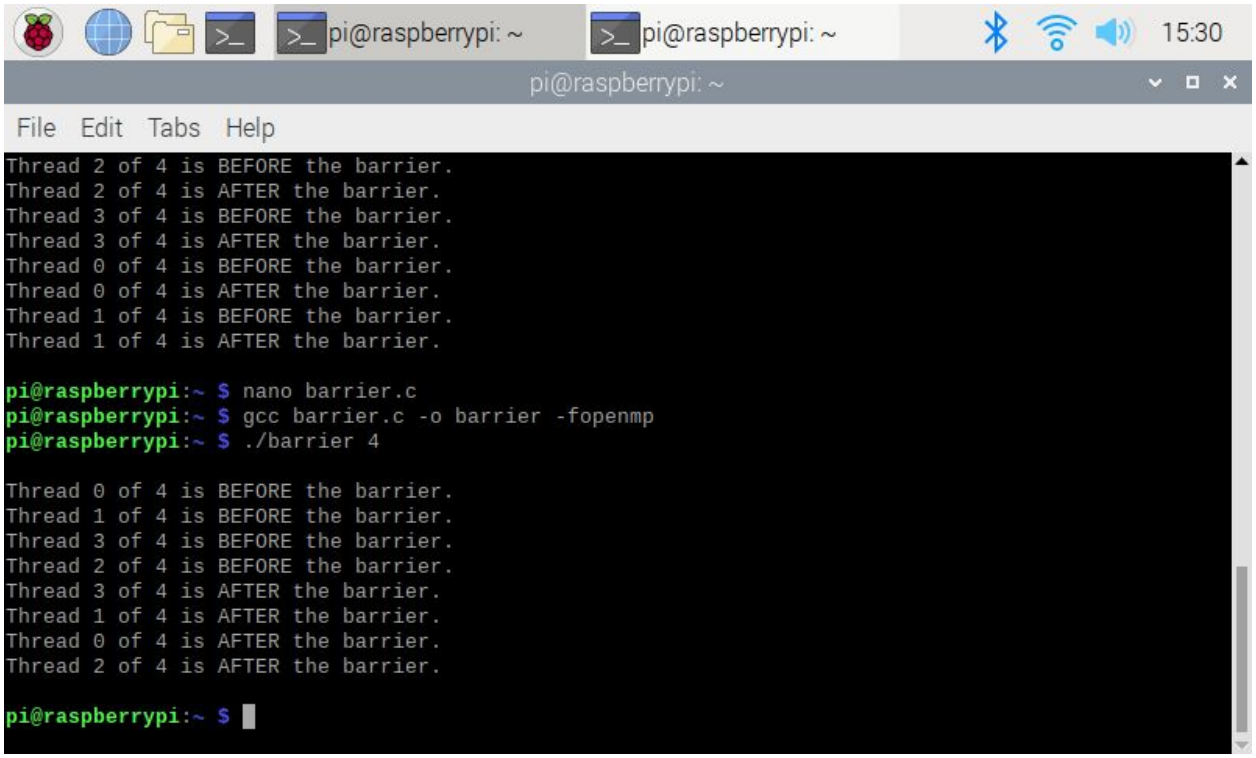

pi@raspberrypi: ~
pi@raspberrypi: ~
15:25
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 barrier.c
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
int main (int argc, char** argv) {
printf("\n");
if(argc>1){
omp_set_num_threads(atoi(argv[1]));
}
#pragma omp parallel
{
int id= omp_get_thread_num();
int numThreads = omp_get_num_threads();
printf("Thread %d of %d is BEFORE the barrier. \n",id,numThreads);
#pragma omp barrier
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line



These three screenshots display the code for the barrier program. This particular screenshot shows when I uncommented the "#pragma omp parallel" line. The barrier pattern ensures that all threads complete a parallel section of code before execution continues.

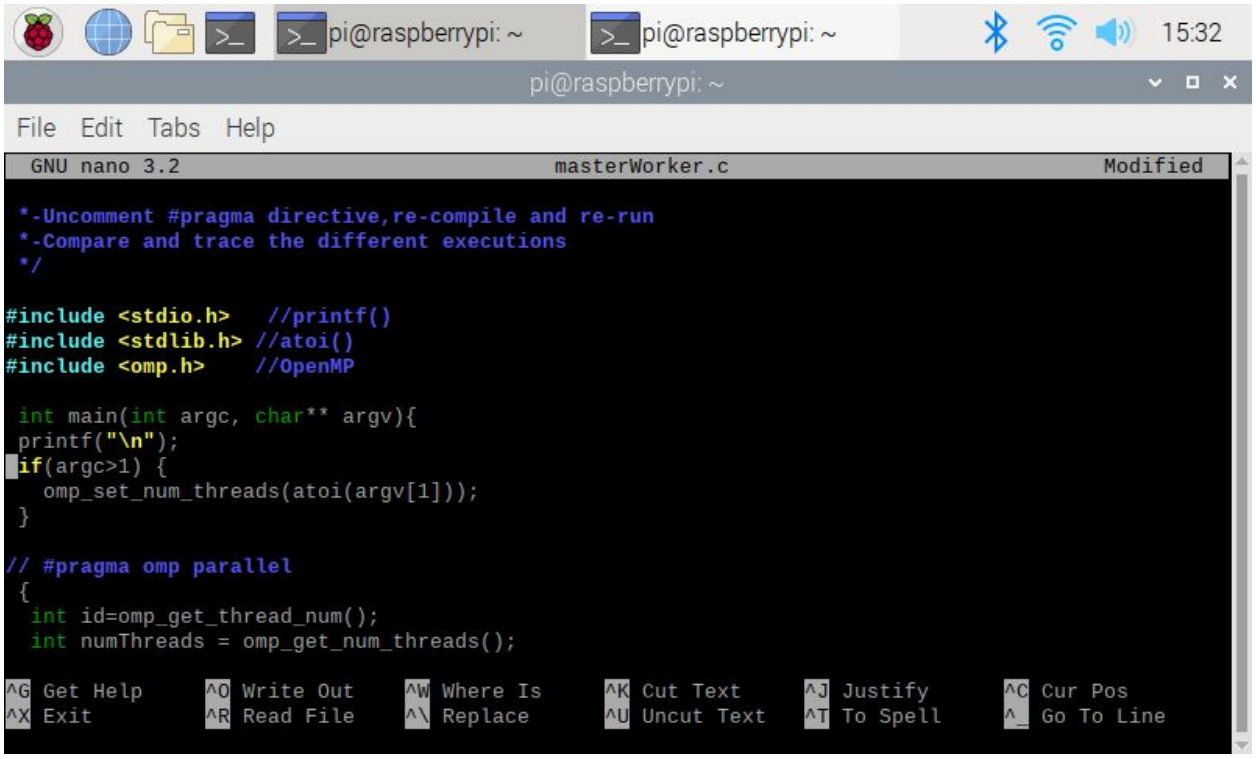


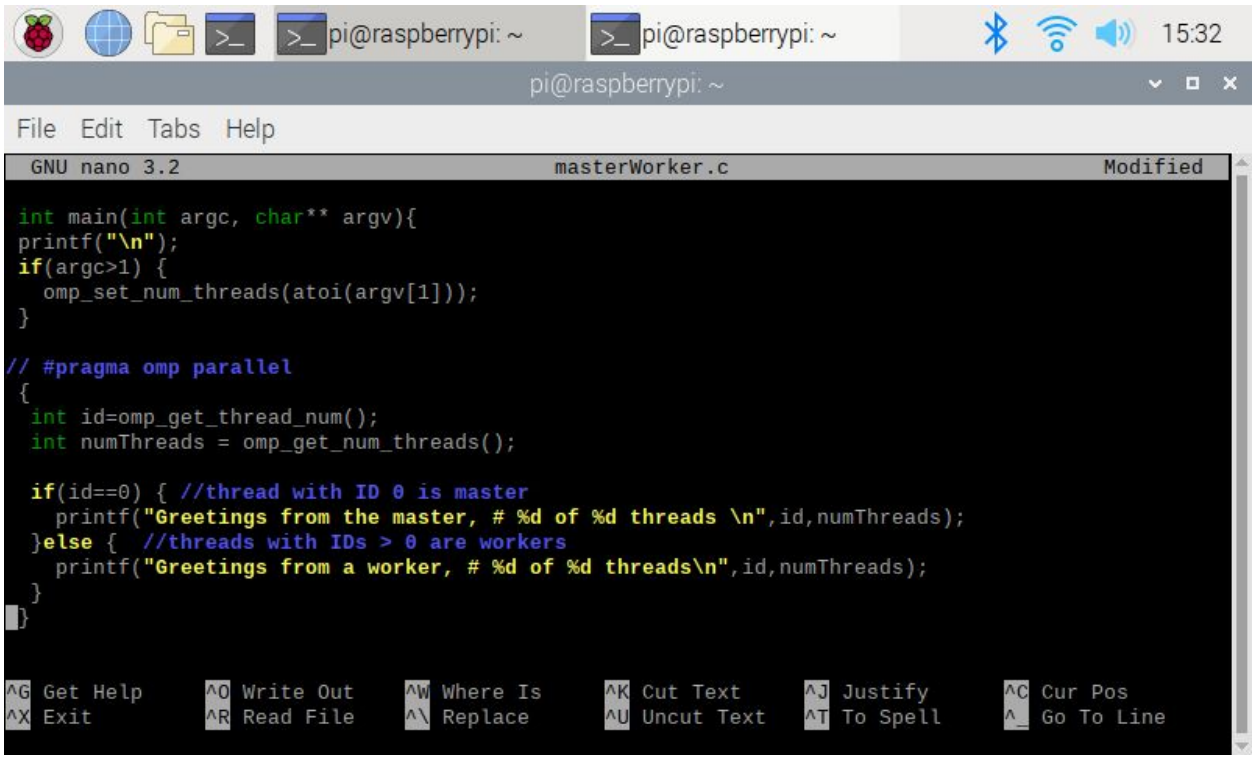
In this screenshot, I edited the barrier with nano command. Then, I used gcc to make the program executable, and I ran the program with "/.barrier 4", I chose 4 because raspberry pi is quad-core, so it is natural to try this program with the value 4. Once I ran the program without uncommenting the "#pragma omp barrier", I noticed the pattern followed a before after pattern, in which each thread would have a before and then an after would follow immediately.

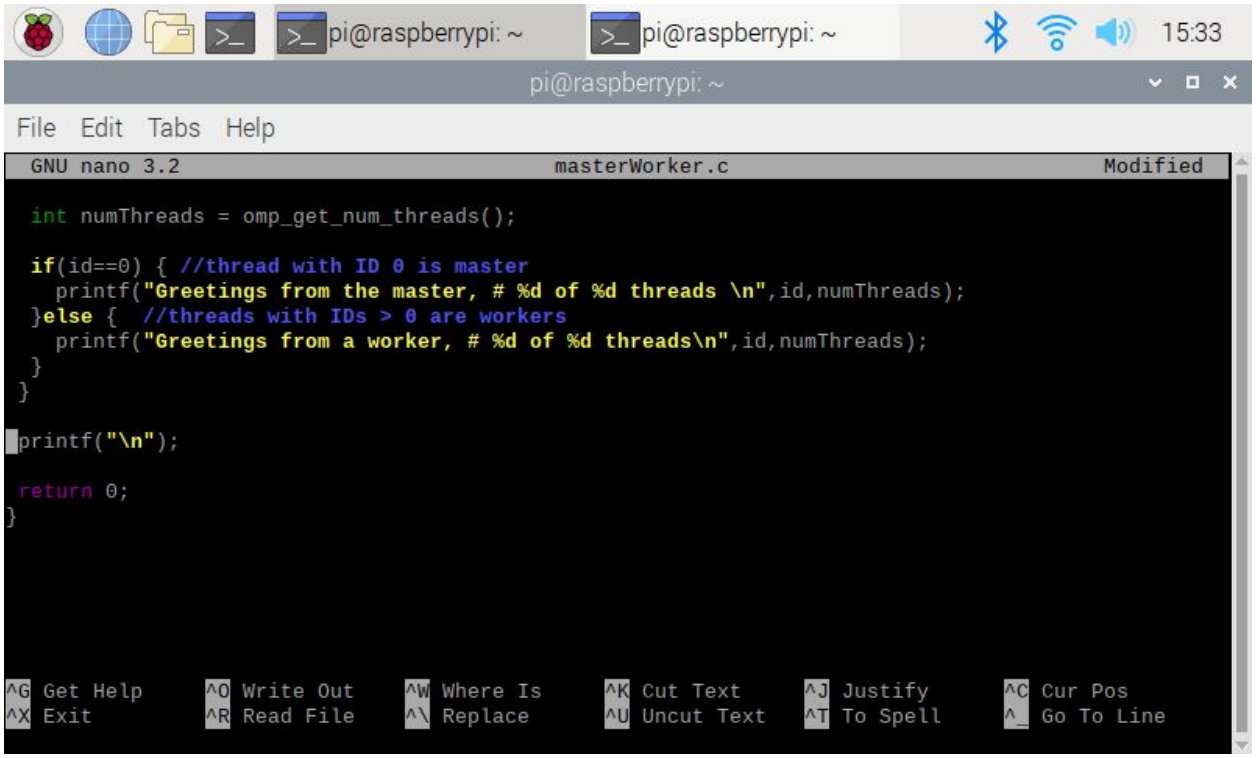


In this screenshot, I uncommented the "#pragma omp barrier" line, and I edited the program with nano. Then, I made the program executable with gcc, and I ran the program with "./barrier 4". Once, I ran the program, I noticed that the threads would execute each thread's before the barrier and then it executed each thread's after the barrier.

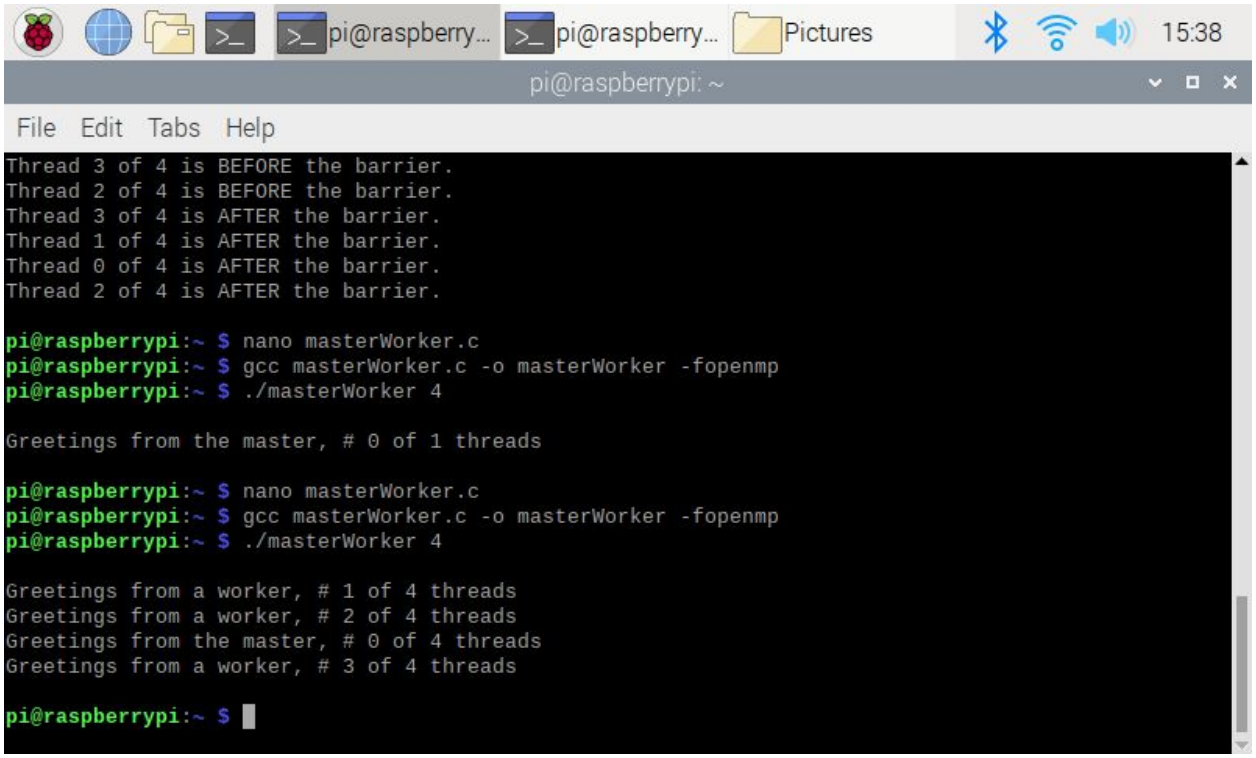
Part 3


pi@raspberrypi: ~
pi@raspberrypi: ~
15:32
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 masterWorker.c Modified
*-Uncomment #pragma directive,re-compile and re-run
*-Compare and trace the different executions
*/
#include <stdio.h> //printf()
#include <stdlib.h> //atoi()
#include <omp.h> //OpenMP
int main(int argc, char** argv){
printf("\n");
if(argc>1) {
omp_set_num_threads(atoi(argv[1]));
}
// #pragma omp parallel
{
int id=omp_get_thread_num();
int numThreads = omp_get_num_threads();
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line


pi@raspberrypi: ~
pi@raspberrypi: ~
15:32
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 masterWorker.c Modified
int main(int argc, char** argv){
printf("\n");
if(argc>1) {
omp_set_num_threads(atoi(argv[1]));
}
// #pragma omp parallel
{
int id=omp_get_thread_num();
int numThreads = omp_get_num_threads();
if(id==0) { //thread with ID 0 is master
printf("Greetings from the master, # %d of %d threads \n",id,numThreads);
}else { //threads with IDs > 0 are workers
printf("Greetings from a worker, # %d of %d threads\n",id,numThreads);
}
}
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line



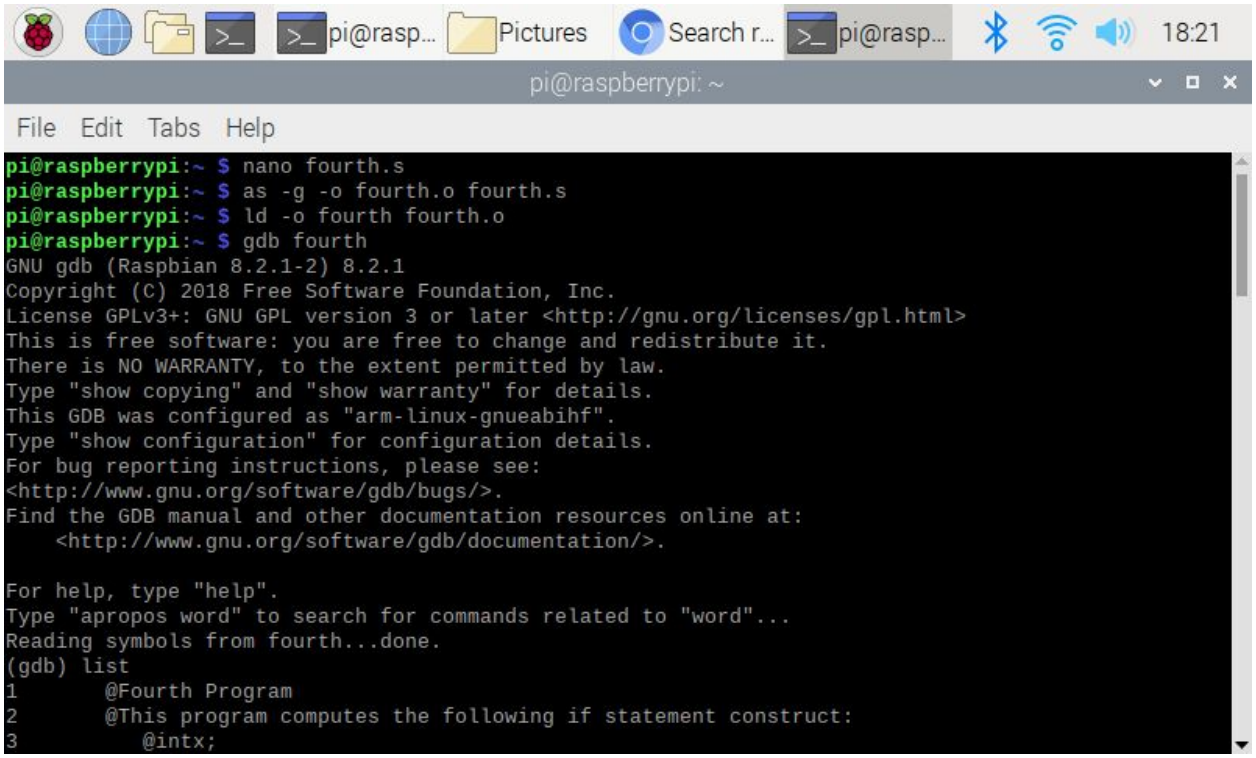
These three screenshots display the code for masterWorker. This particular snippet has the "#pragma omp parallel" commented. This program helps with understanding the Master-Worker Implementation Strategy. In this strategy, the master executes one block of code when it forks, and the other threads, known as workers, execute a different block of code when they fork.



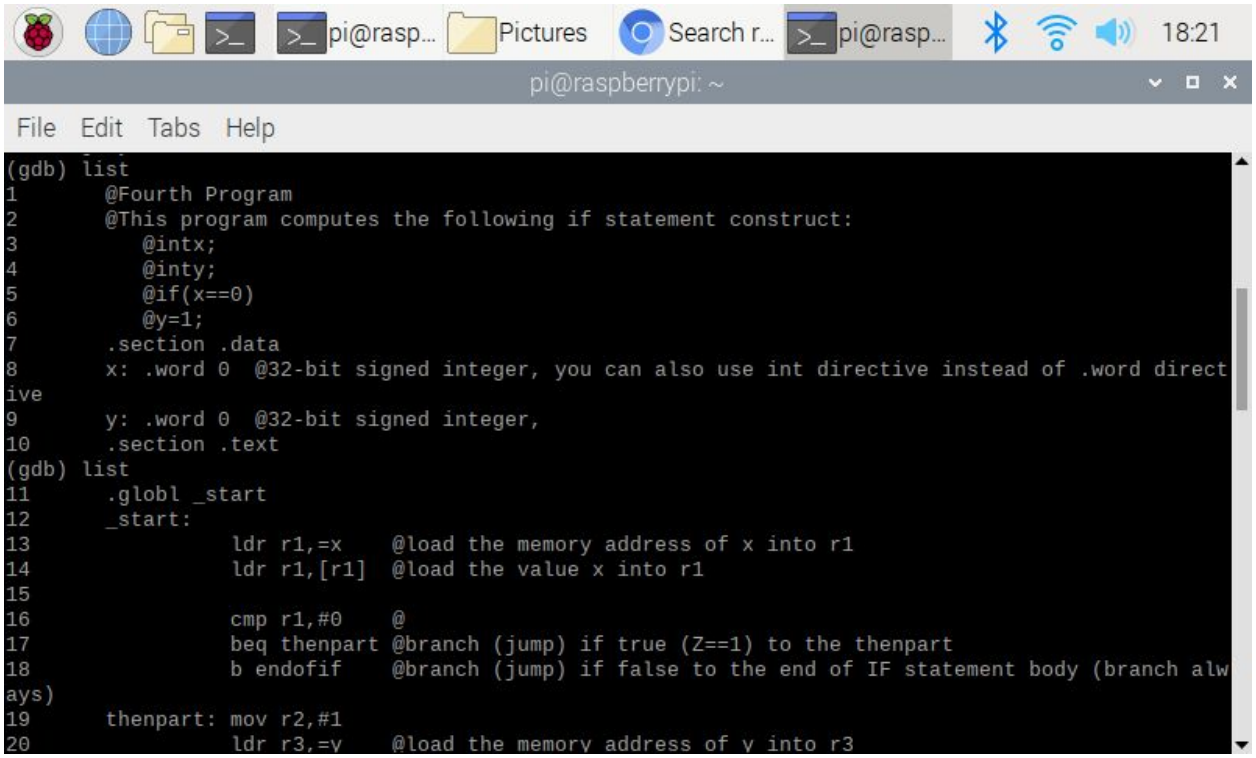
In this screenshot, I edited the masterWorker code with nano. Then, I made the program executable with gcc, and I ran the program with "./masterWorker 4". I was surprised to see that that even with the "#pragma omp parallel" commented, the output was only one single line, so then I reviewed the code to make sure I understood it correctly. Then, I ran the program with "#pragma omp parallel" uncommented, and I noticed that the code was now printing out the correct information, and I realized that the uncommented phrase is essential to the program running in parallel.

ARM Assembly Programming: Alaya Shack

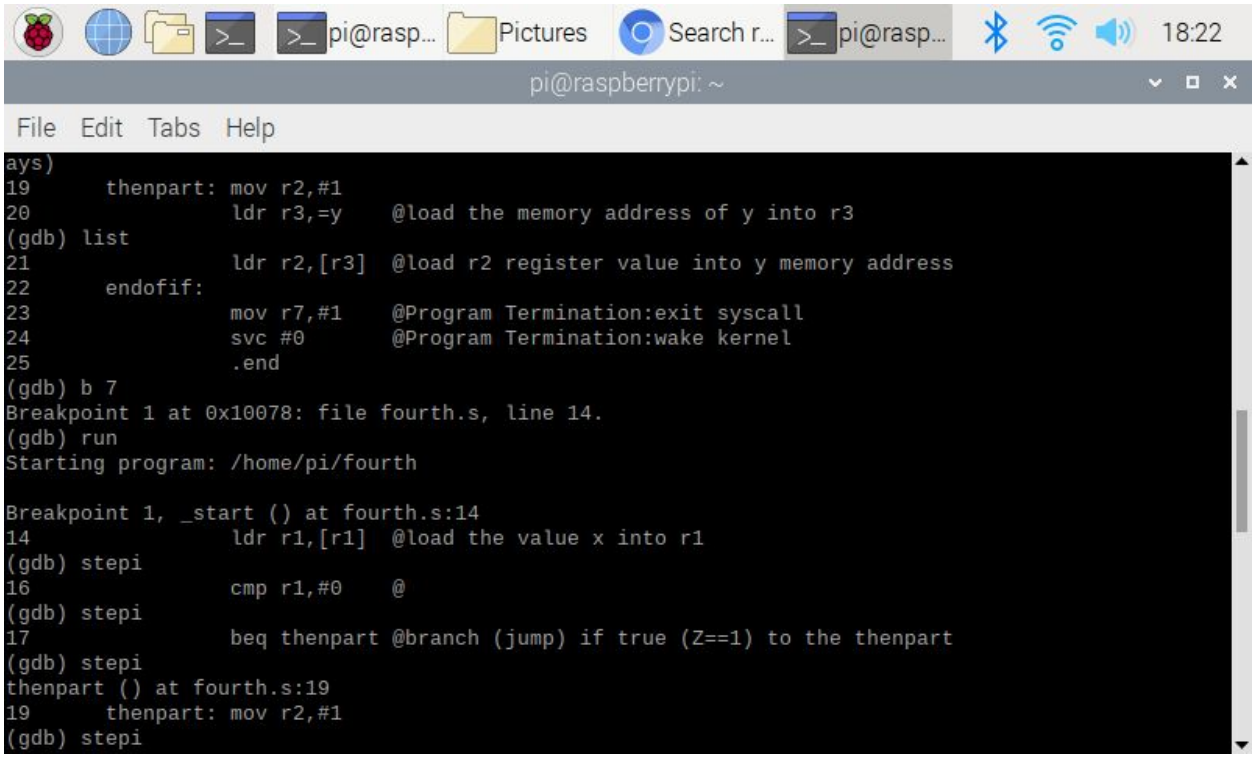
Part 1



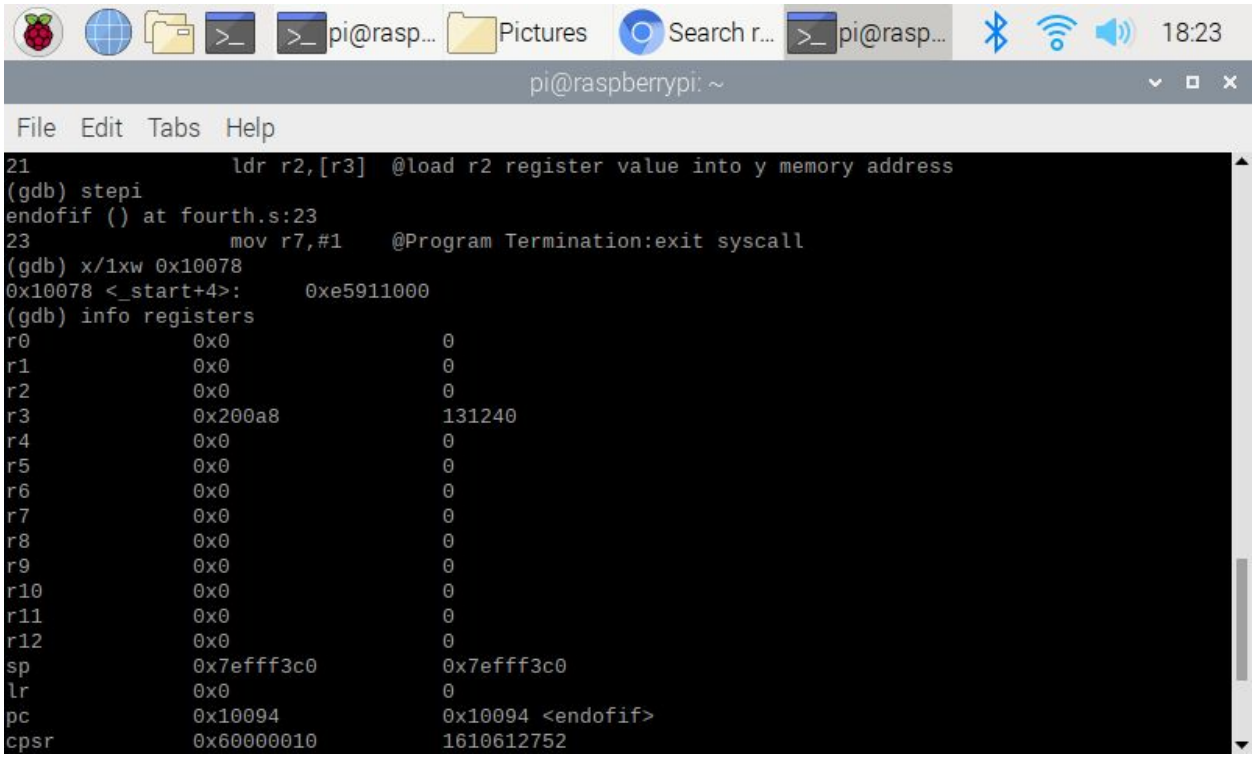
In this screenshot, I edited, assembled, and linked the fourth program. Then, I launched the debugger.



In this screenshot, I used the "gdb list" command to list the code. I had to use the command three times to list all of the code. In this part of the code, we are dealing with if/else statements. The comments help to explain what happens with beq and b. When translating if/else statements from high-level to assembly, cmp is used because it is nondestructive, but it still affects the flags.

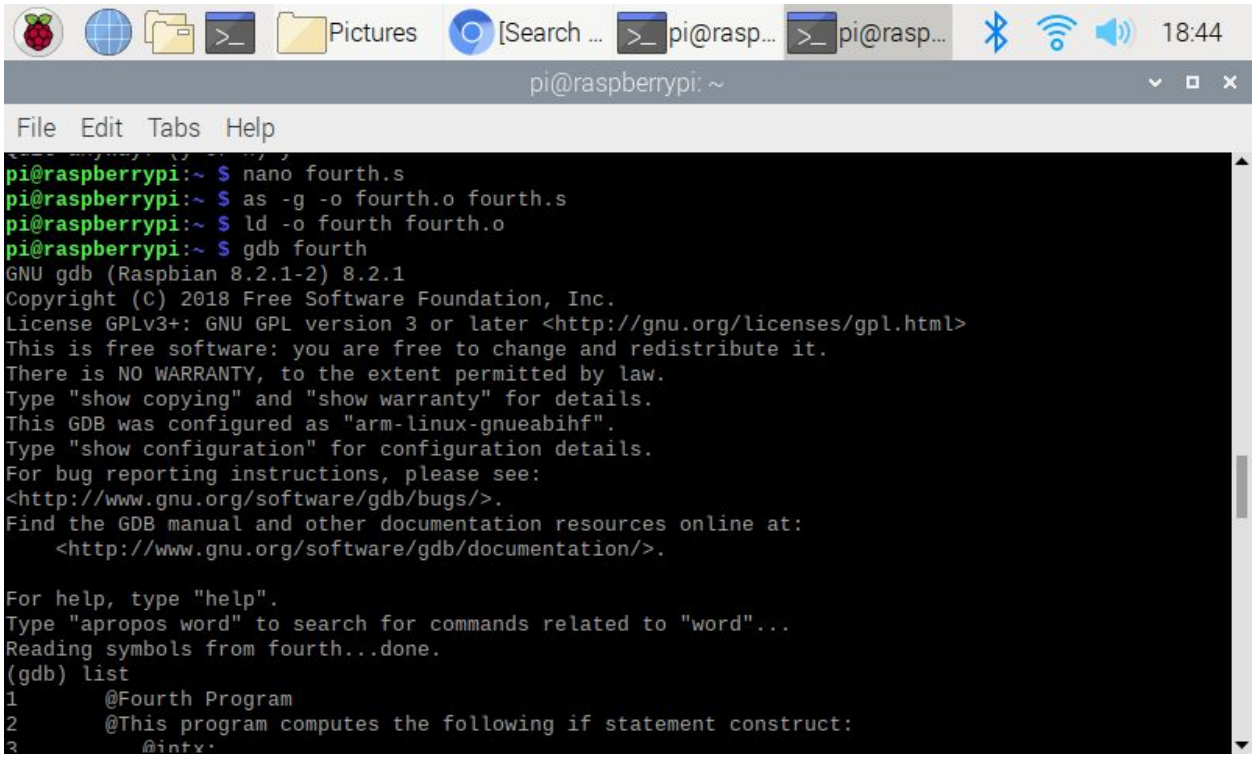


In this screenshot, I created a breakpoint at line 7. Then, I started the program with “gdb run”. Once I started the program, I used “gdb stepi” to step through the code one line at a time, until I reached line 23.

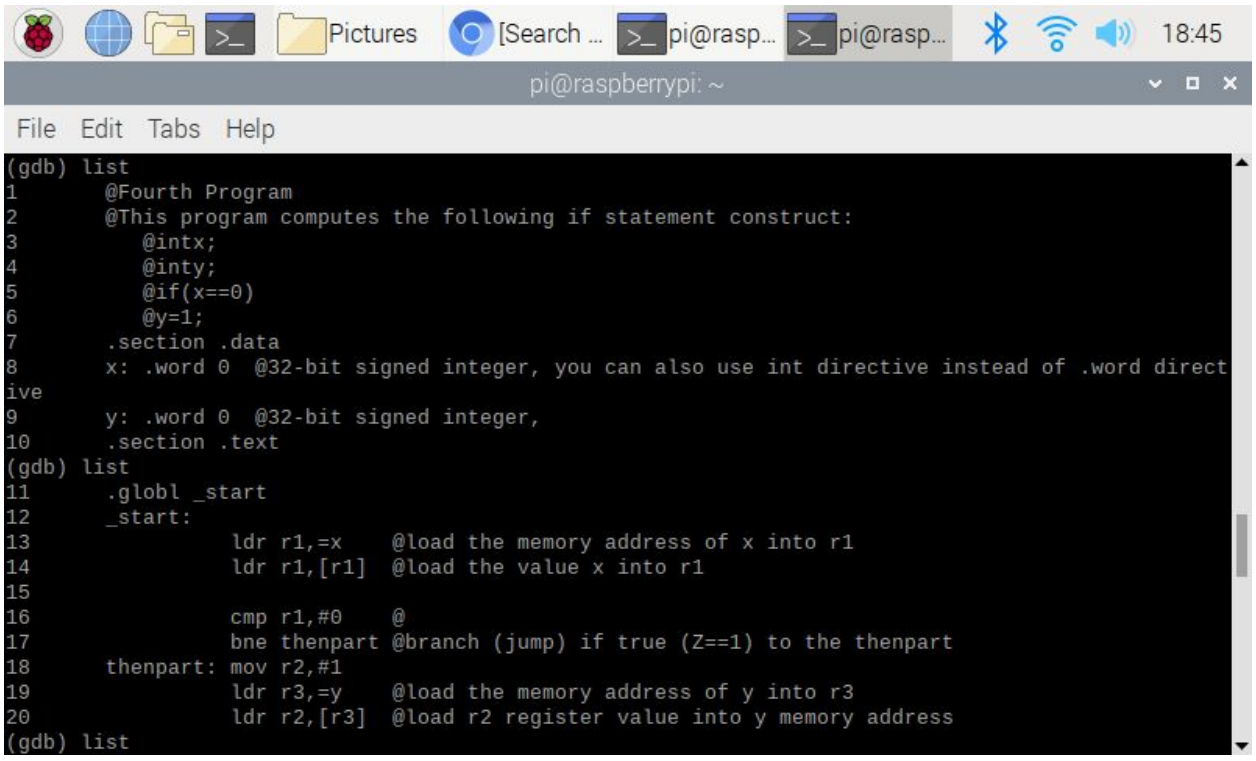


In this screenshot, I used “gdb x/1xw 0x10078” to examine the y memory location. Then, I used “gdb info registers” to display the registers. In r3, the value 0x200a8 is displayed, and the zero flag is 1 because cmp is the only instruction that affects the flags, and 0-0 is 0, which raises the z flag.

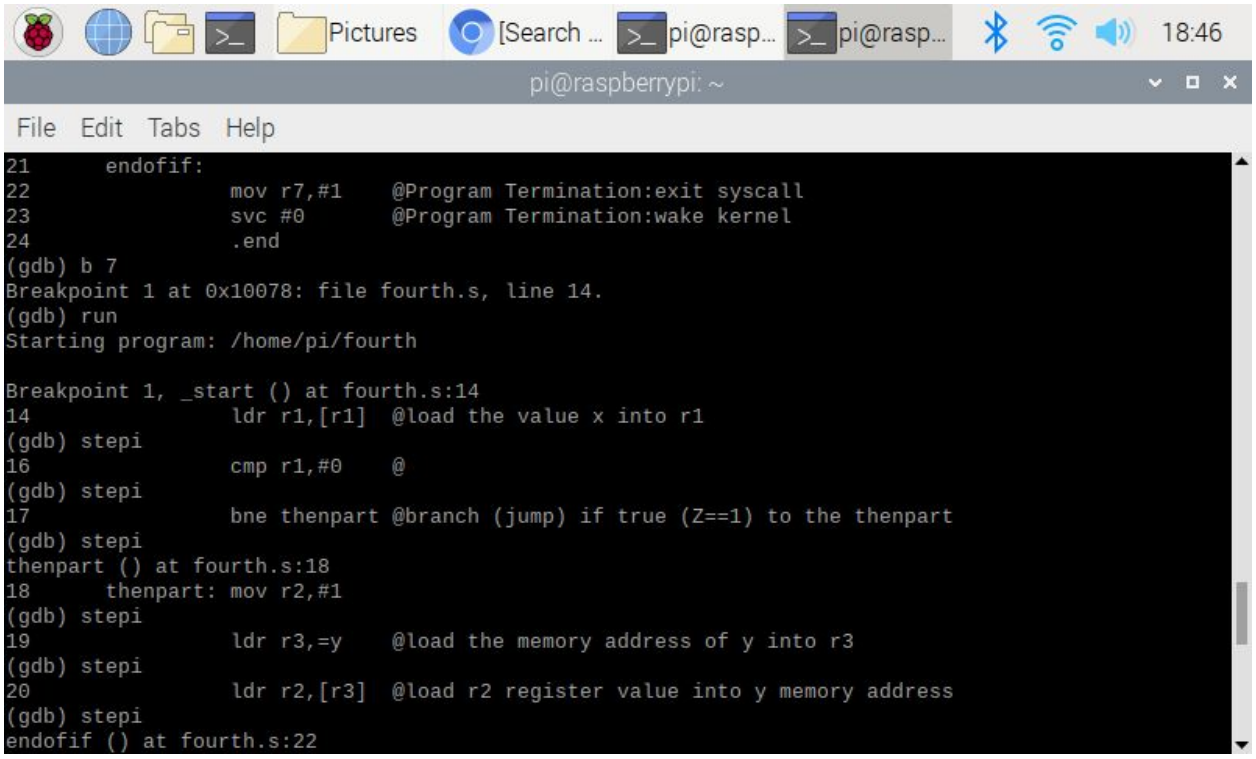
Part 2



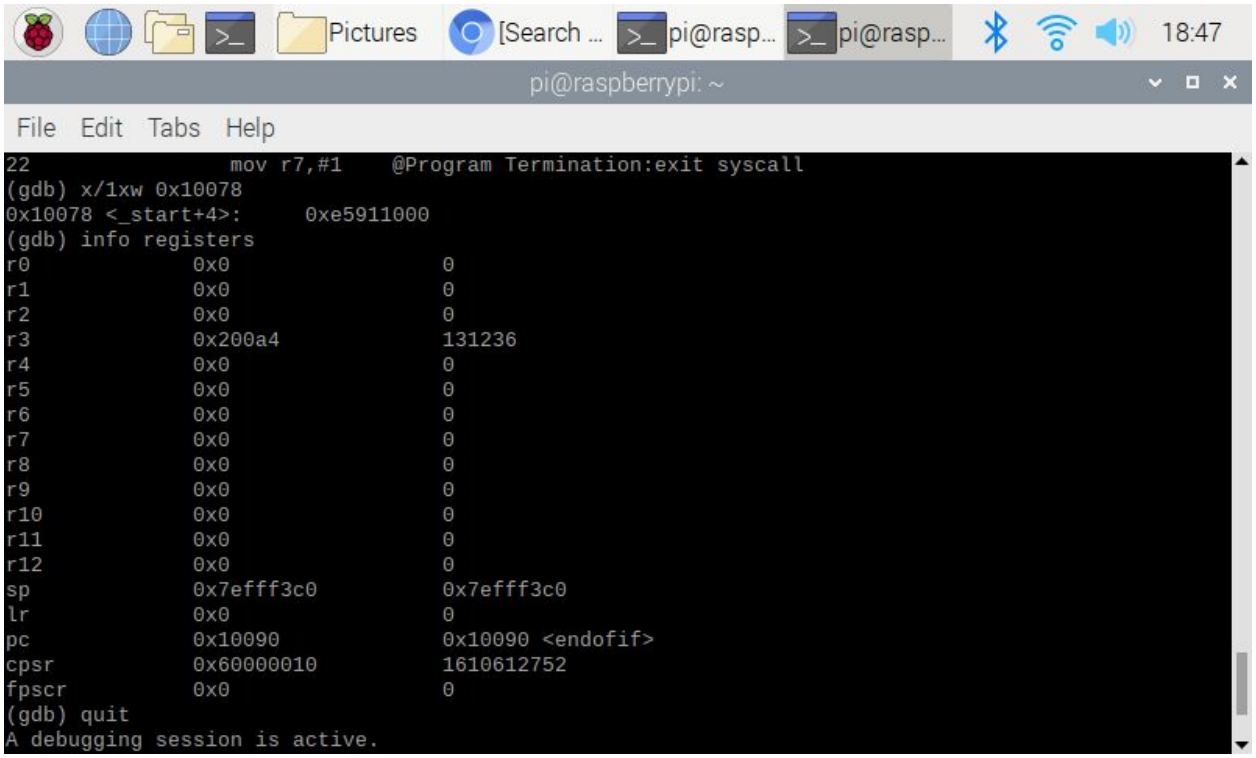
In this screenshot, I corrected the fourth program. Then, I assembled, linked, and launched the debugger.



In this screenshot, I used “gdb list” three times so that I could display all of the code. The code shows where I replaced beq with bne to make the program more efficient.

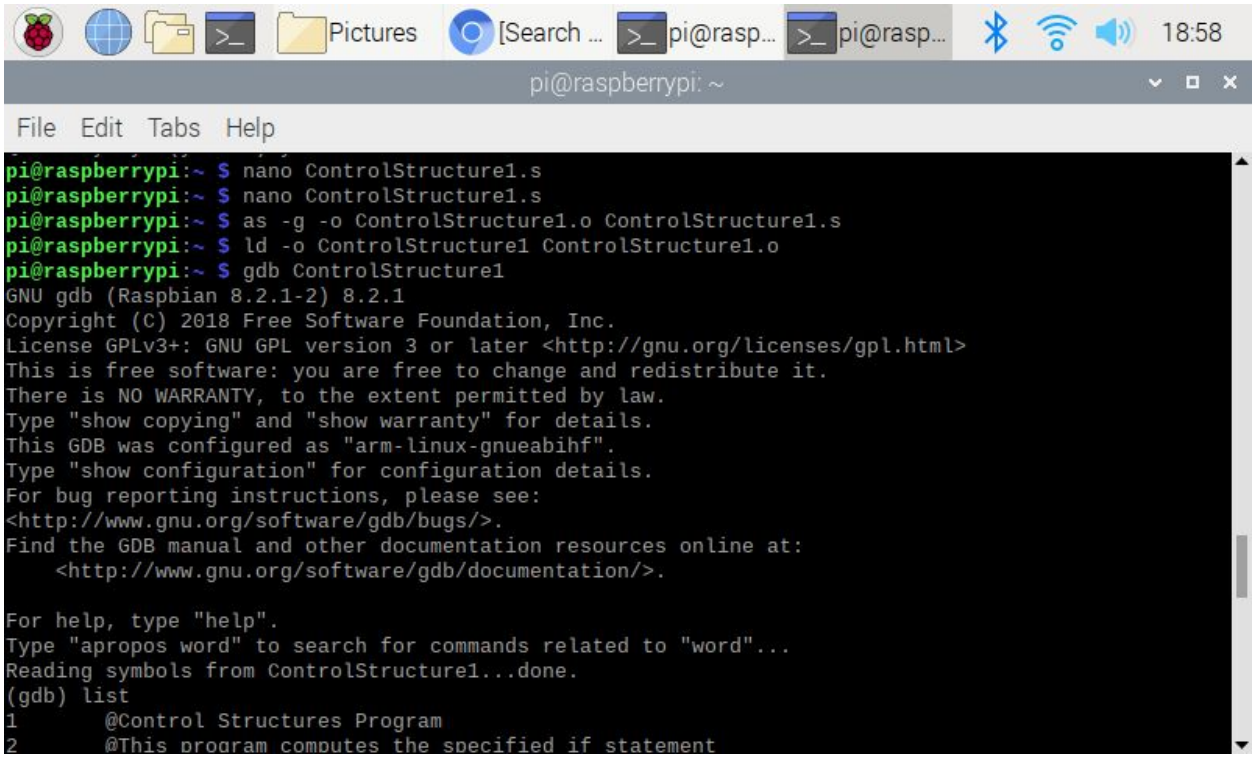


In this screenshot, I created a breakpoint at line 7 and I ran the program. Then, I used “gdb stepi” to step through each line of code until I reached line 22.

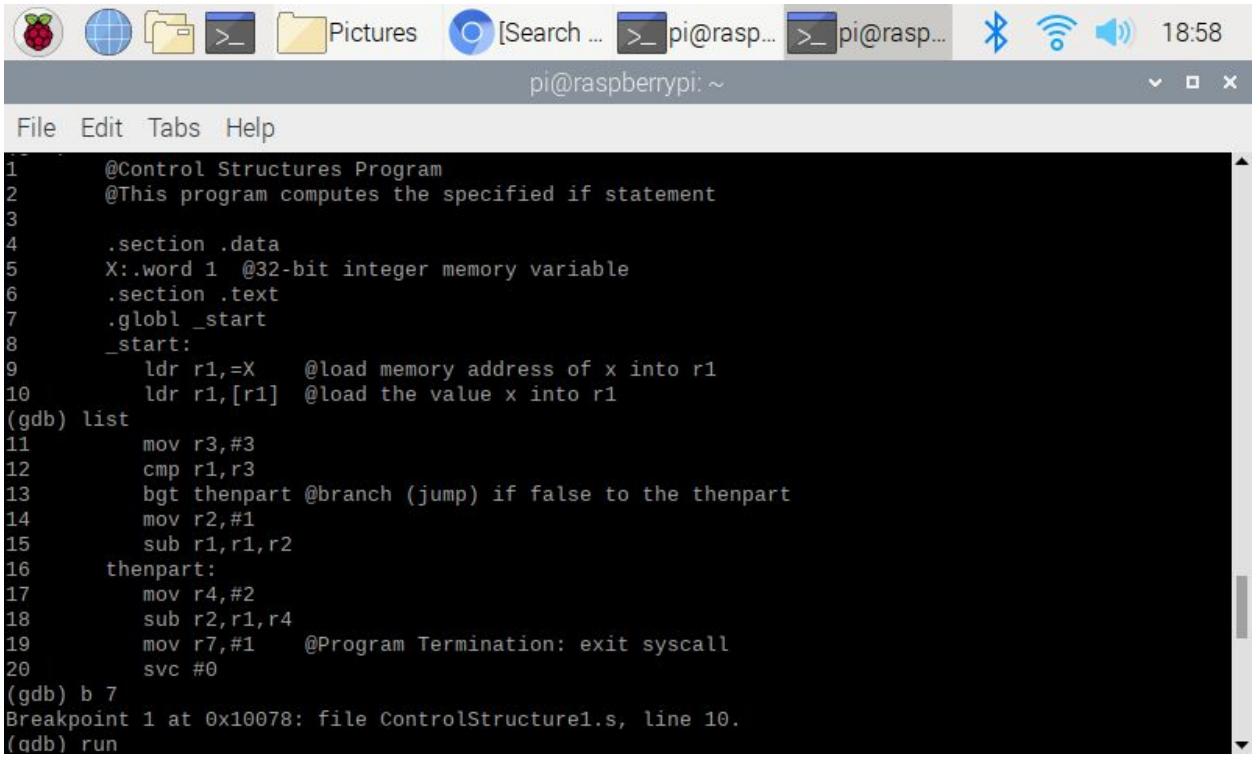


In this screenshot, I displayed the registers with “gdb registers”. This time r3 has the value 0x200a4, and the zero flag is still 1 because cmp is the only instruction that affects the flags, and the result of cmp is 0, so the z flag is raised. Lastly, I used “gdb quit” to stop the debugger.

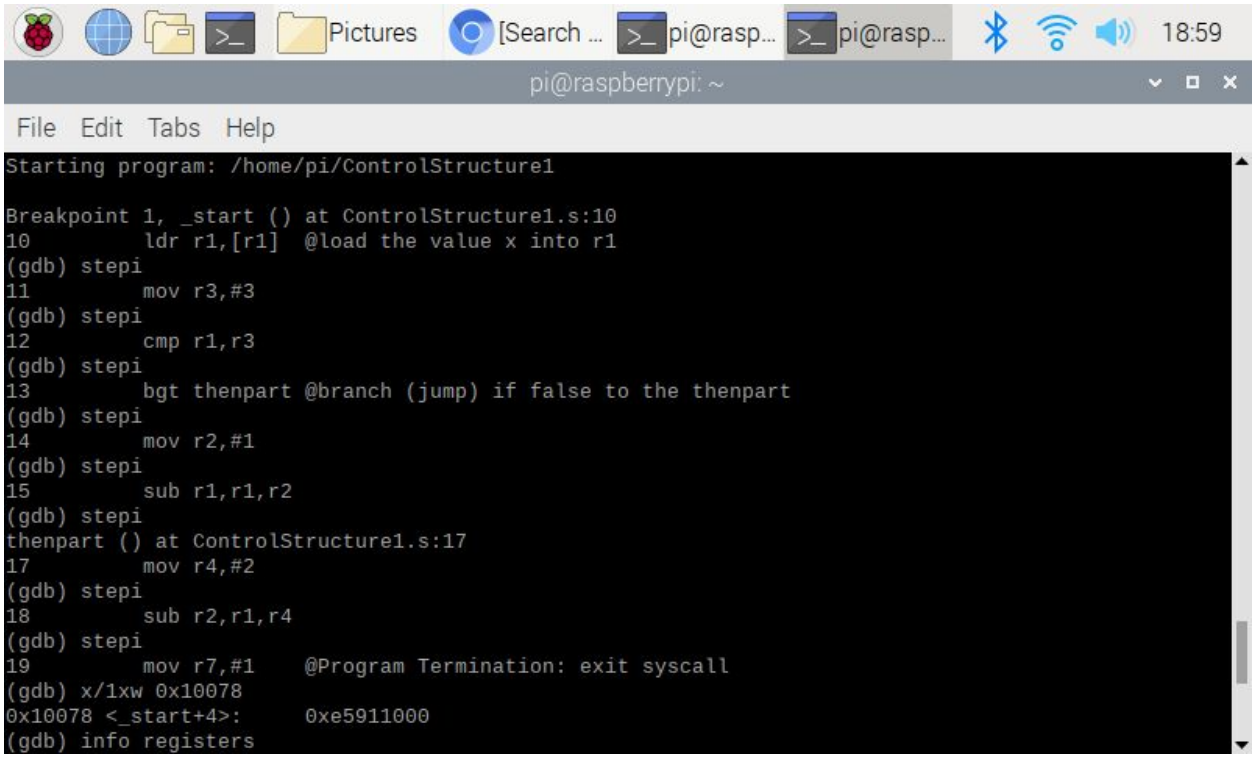
Part 3



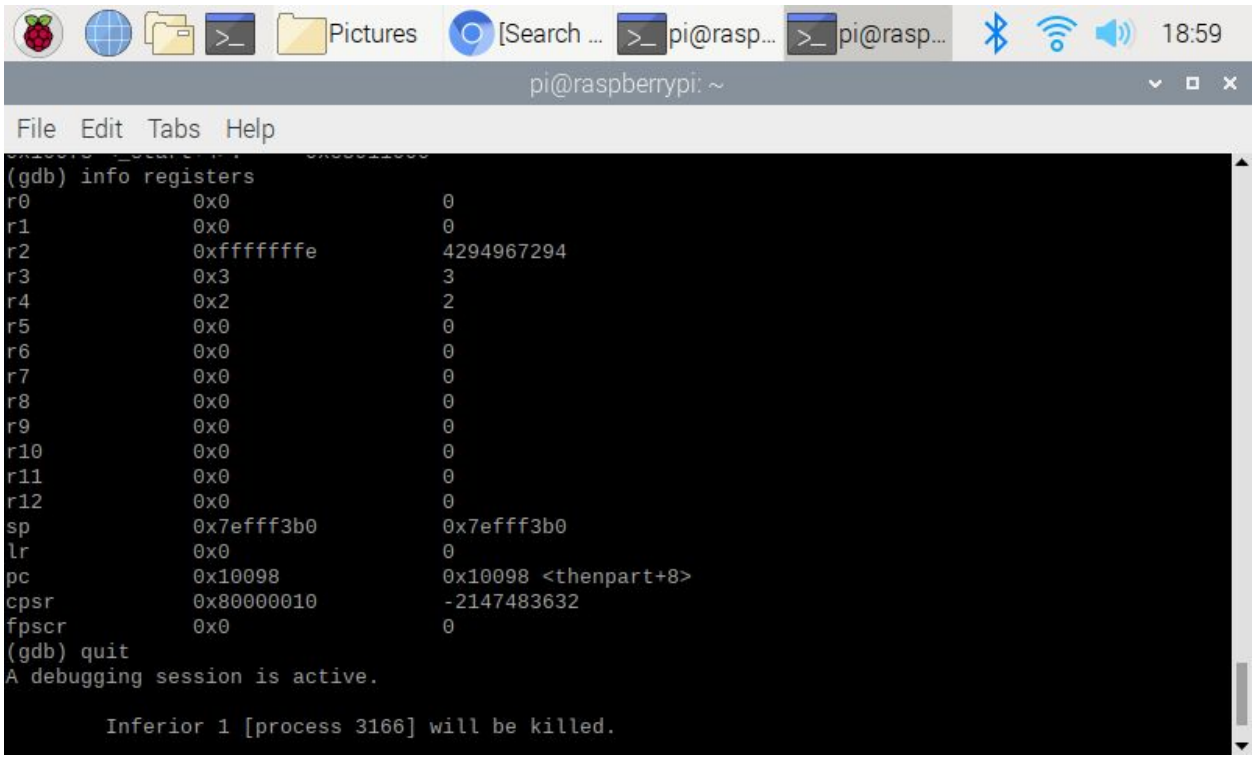
In this screenshot, I edited my “ControlStructure1” program with “nano”. Then, I assembled, linked, and launched the debugger.



In this screenshot, I used “gdb list” twice to display my code. In the .data, I declared a 32-bit memory variable and set it to 1. Then, in the .text, I loaded the memory address of X into r1, and then I loaded the actual value of X into r1. I moved the value r3 into a register, and I used cmp with r1 and r3. I used DeMorgan’s law to get bgt(branch if greater than) rather than using ble(branch is less than or equal to), so that the program would be more efficient.



In this screenshot, I used “gdb stepi” to step through the code one line at a time.Then, I used “gdb x/1xw 0x10078” to examine the memory.



In this screenshot, I used “gdb info registers” to display the registers. In r1, the value is 0 because 1-1 is 0. In r3, the value is 3 because I stored 3 in the register. In r4, the value is 2 because I stored 2 in the register. The zero flag is set because the last instruction of sub r1,r1,r2 is equal to 0, which raises the z flag.

Parallel Programming Skills Foundation: Arteen Ghafourikia

- ➔ Race Condition:
  - ◆ What is race condition?
    - A race condition is the performance of electronics, software or other systems where the final result relies on the order and timing of events you have no control over.
  - ◆ Why is race condition difficult to reproduce and debug?
    - The final result is non deterministic since there are different timings which will produce different results and it is dependent on the timing of those threads. It will be very difficult to produce the same results.
  - ◆ How can it be fixed? Provide an example from your Project_A3 (see spmd2.c)
    - It can be fixed by initializing the variables inside the pragma, because it will then reinitialize each time. In the spmd2.c example this is prevalent because we commented the initialization which was outside of the pragma, and then initialized it within the pragma to get different thread numbers.
- ➔ Summarize the Parallel programming Patterns section in the “Introduction to Parallel Computing_3.pdf” (two pages). In your own words (one paragraph, no more than 150 words).
  - ◆ There are two main patterns that are used and they are Strategies and Concurrent Execution Mechanisms. Strategies generally come in algorithmic strategies and implementation strategies. They are used to make decisions on what tasks should be done simultaneously by the different processing units. Parallel programs are made up of implementation strategies. Concurrent Execution Mechanisms are what allows parallelism and how the execution is processed allowing communication .
- ➔ In the section “Categorizing Patterns in the “Introduction to Parallel Computing_3.pdf” **compare** the following:
  - ◆ Collective synchronization (barrier) with Collective communication (reduction)
    - A barrier is when the threads and processes have to wait until all the other threads and processes reach the barrier to continue. Reduction is when the communicator gets the data from all the other processes to get the result..
  - ◆ Master-worker with fork-join

- Master-worker is when there is one thread that is the master and it executes a block of code when it diverges. The other threads which are the workers execute different blocks of code when they fork. Fork join is when the different points of execution merge together.

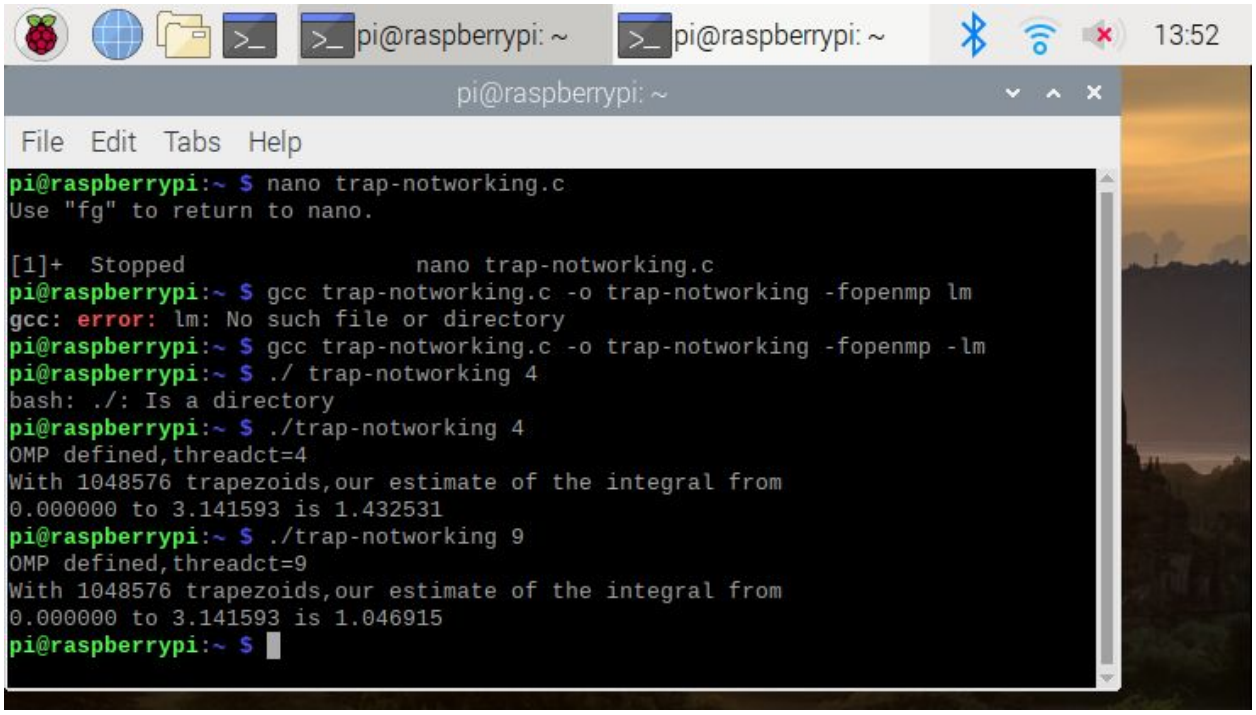
➔ Dependency: Using your own words and explanation, answer the following:

- ◆ Where can we find parallelism in programming?
  - We find parallelism in process/thread control and coordination patterns because they both imply how multiple programs can run simultaneously. This can be found in databases, servers, simulation applications, and etc.
- ◆ What is dependency and what are its types (provide one example for each)?
  - A dependency is when a step requires an earlier step to produce a result before the next step can happen. (Order of execution is important)
    - True(flow)- s1:a=3, s2: b=a
    - Anti- s1:a=b*3, s2:b=a+2
    - Output- s1:a=c, s2: a=b
- ◆ When a statement is dependent and when it is independent (provide two examples)?
  - A statement is dependent when the order of the execution matters, it is independent when the order does not matter.
    Dependent example
    STATEMENT 1: a=6, STATEMENT 2:b=7
    Independent example
    STATEMENT 1: a=5, STATEMENT2: b=a (STATEMENT 2 is now dependent on "a" being 5)
- ◆ When can two statements be executed in parallel?
  - When the statement's order doesn't matter and the statements don't have any dependencies.
- ◆ How can dependency be removed?
  - They can be removed by removing the dependent statement(eliminating) or writing it in a way where there are no dependencies (rearranging).
- ◆ How do we compute dependency for the following two loops and what type/s of dependency?
  - To determine the dependency of these two loops you have to find the IN and OUT sets. These are both examples of True(flow) dependency.

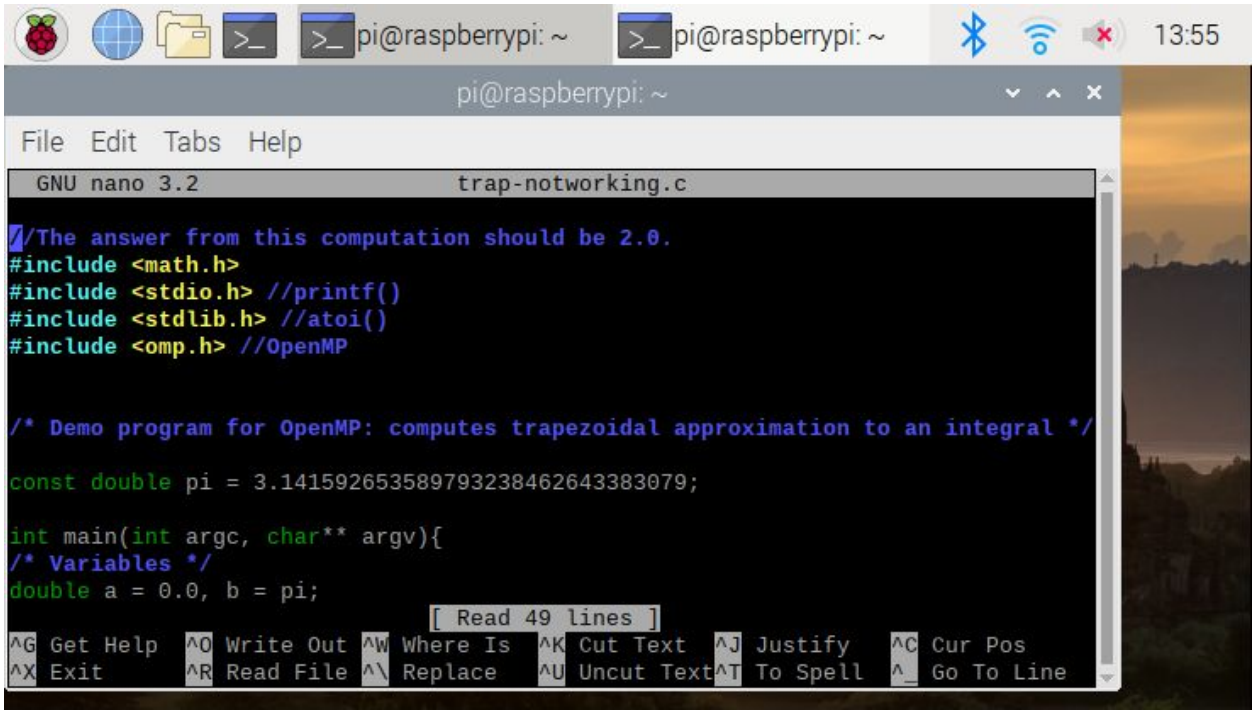
```
for(i=0;i<100;i++)          for(i=0;i<100;i++){
   S1: a[i]=i;                 S1:a[i]=i;
                               S2:b[i]=2*i;
                              }
```

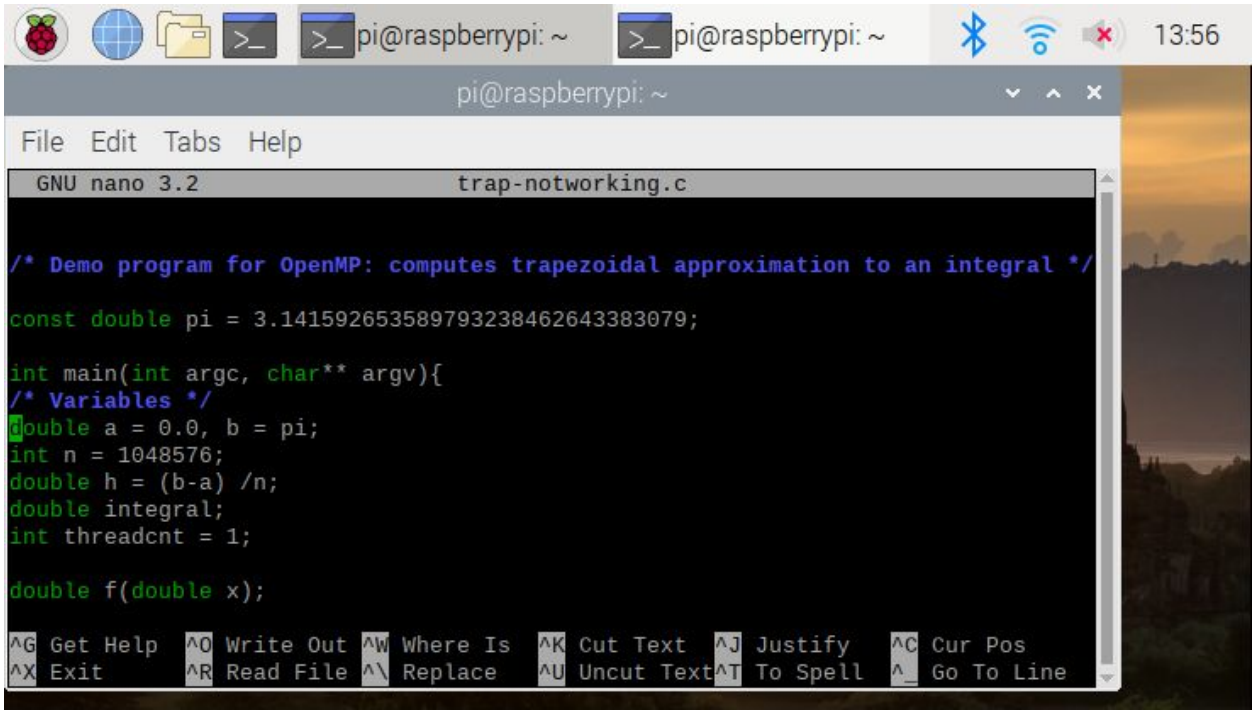
Parallel Programming Basics:

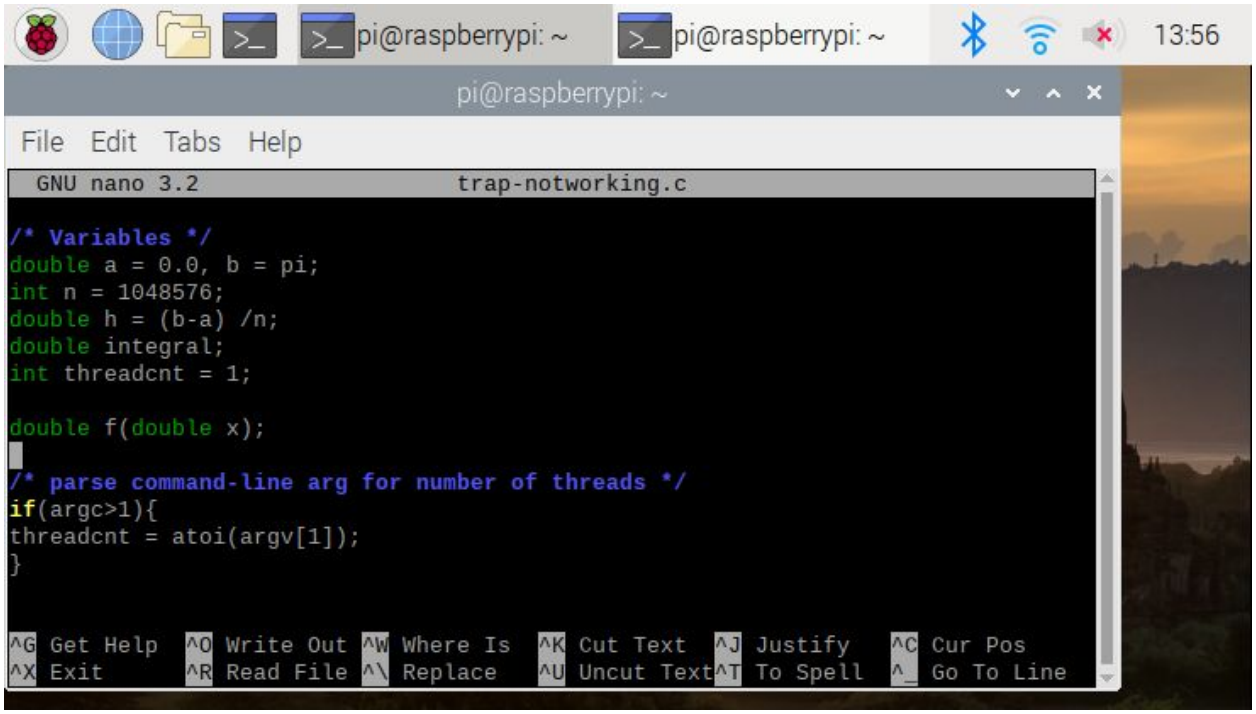
Part 1

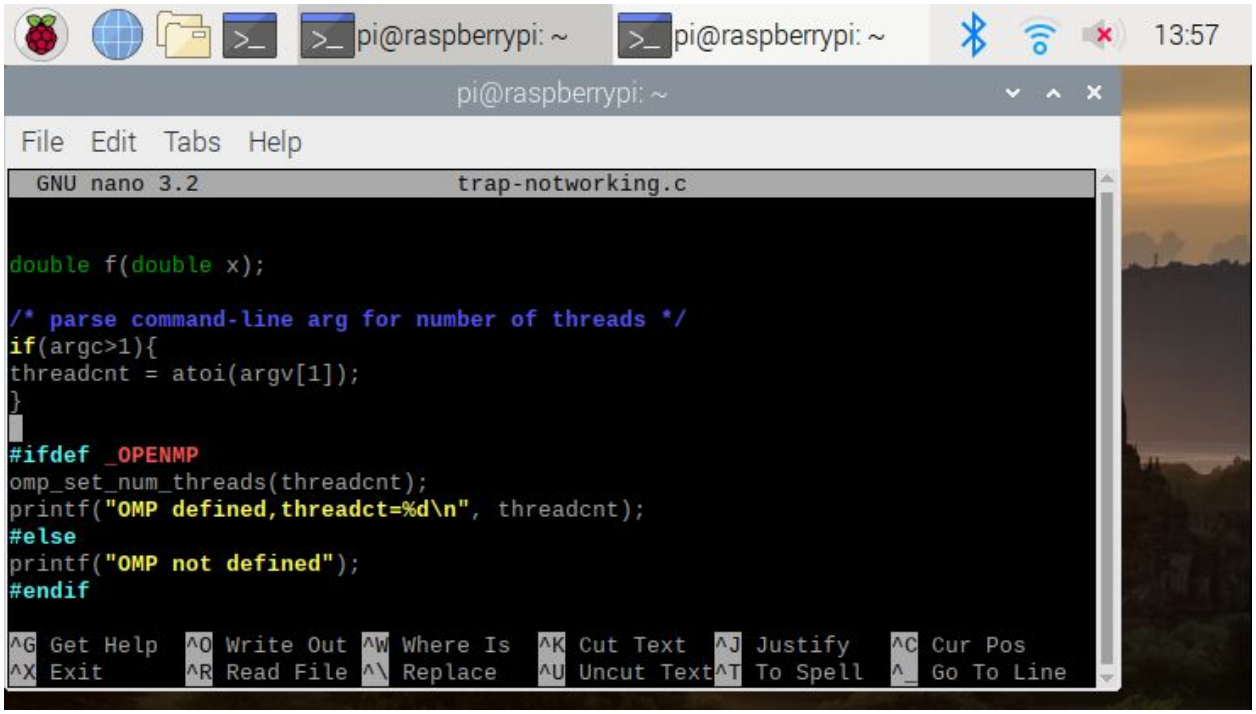


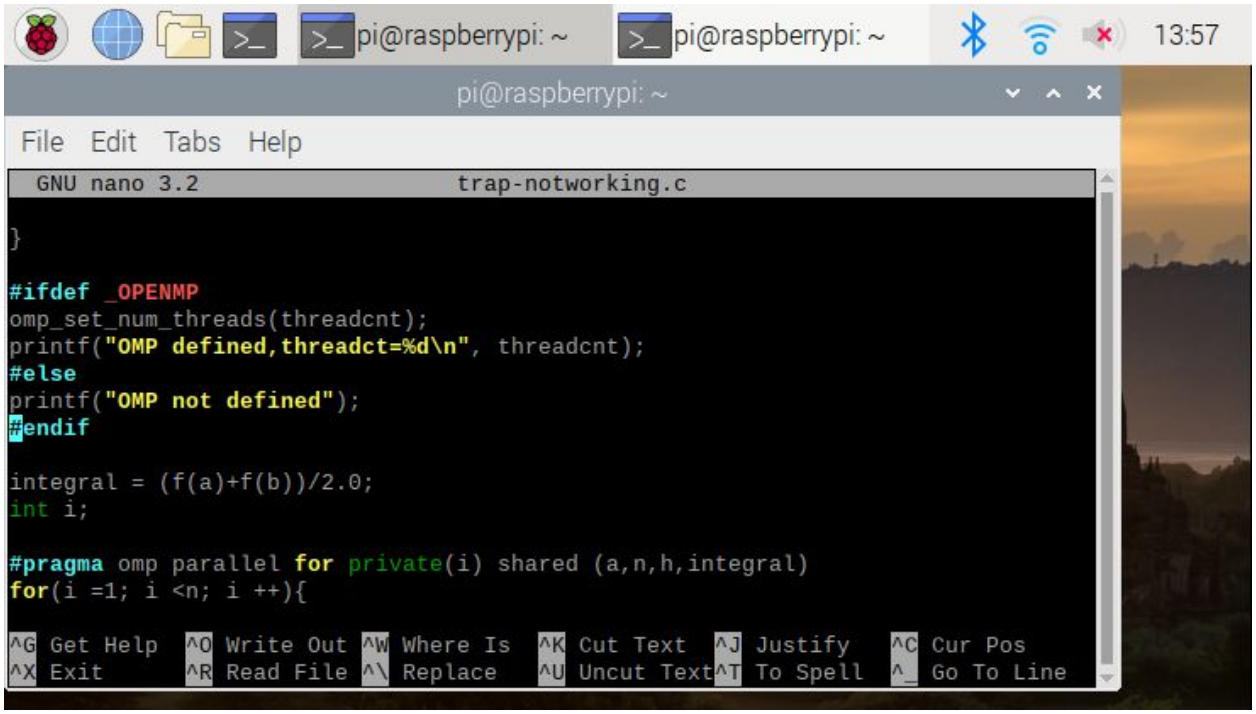
In this screenshot I am running trap-notworking.In this program we are doing calculus and as you can see the output value is not 2.0. The code did not display the correct answer because it was not encapsulated.

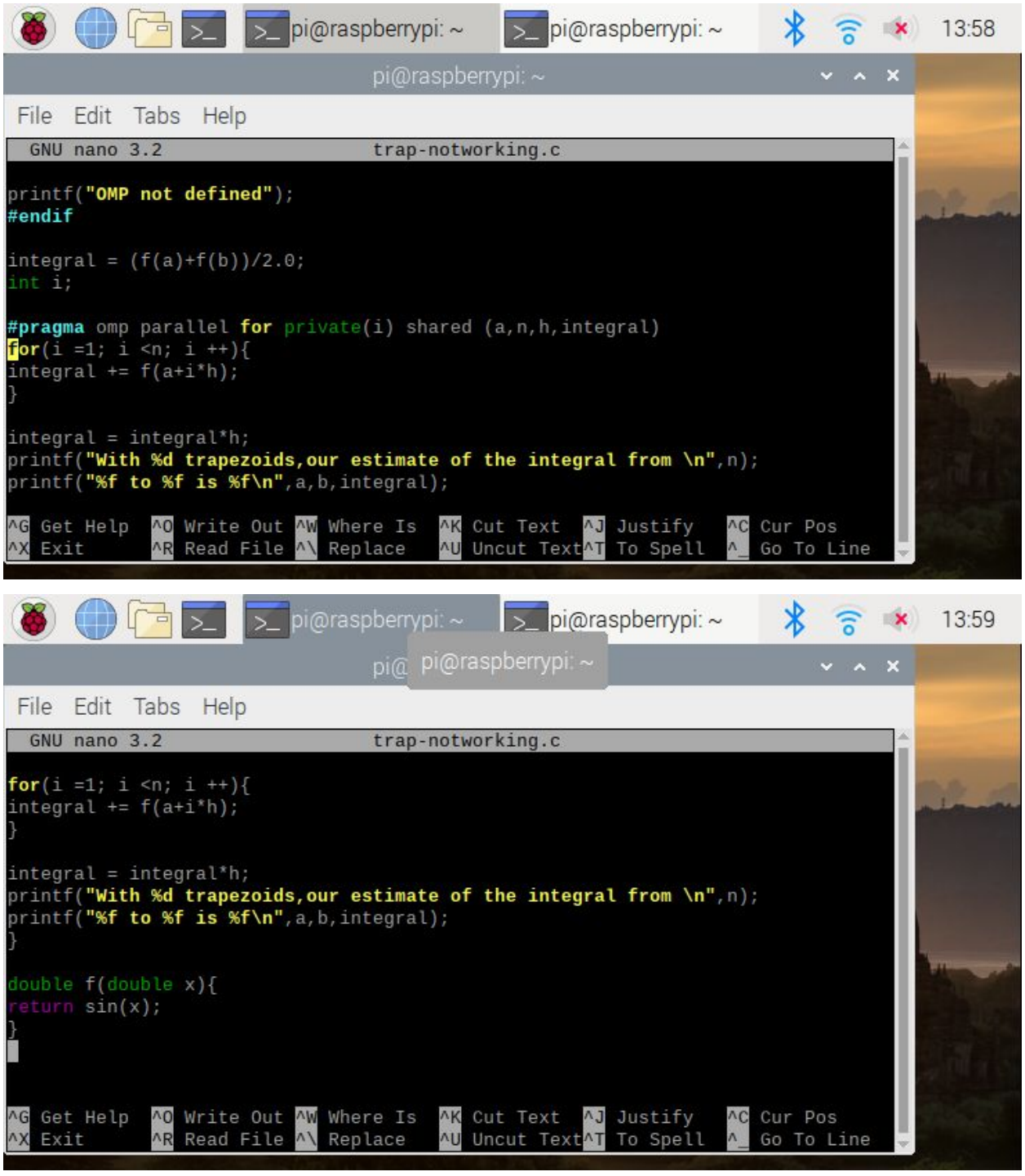

pi@raspberrypi: ~
pi@raspberrypi: ~
13:55
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 trap-notworking.c
//The answer from this computation should be 2.0.
#include <math.h>
#include <stdio.h> //printf()
#include <stdlib.h> //atoi()
#include <omp.h> //OpenMP
/* Demo program for OpenMP: computes trapezoidal approximation to an integral */
const double pi = 3.14159265358979323846264338307 9;
int main(int argc, char** argv){
/* Variables */
double a = 0.0, b = pi;
[Read 49 lines]
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line


pi@raspberrypi: ~
pi@raspberrypi: ~
13:56
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 trap-notworking.c
/* Demo program for OpenMP: computes trapezoidal approximation to an integral */
const double pi = 3.14159265358979323846264338307 9;
int main(int argc, char** argv){
/* Variables */
double a = 0.0, b = pi;
int n = 1048576;
double h = (b-a) /n;
double integral;
int threadcnt = 1;
double f(double x);
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line

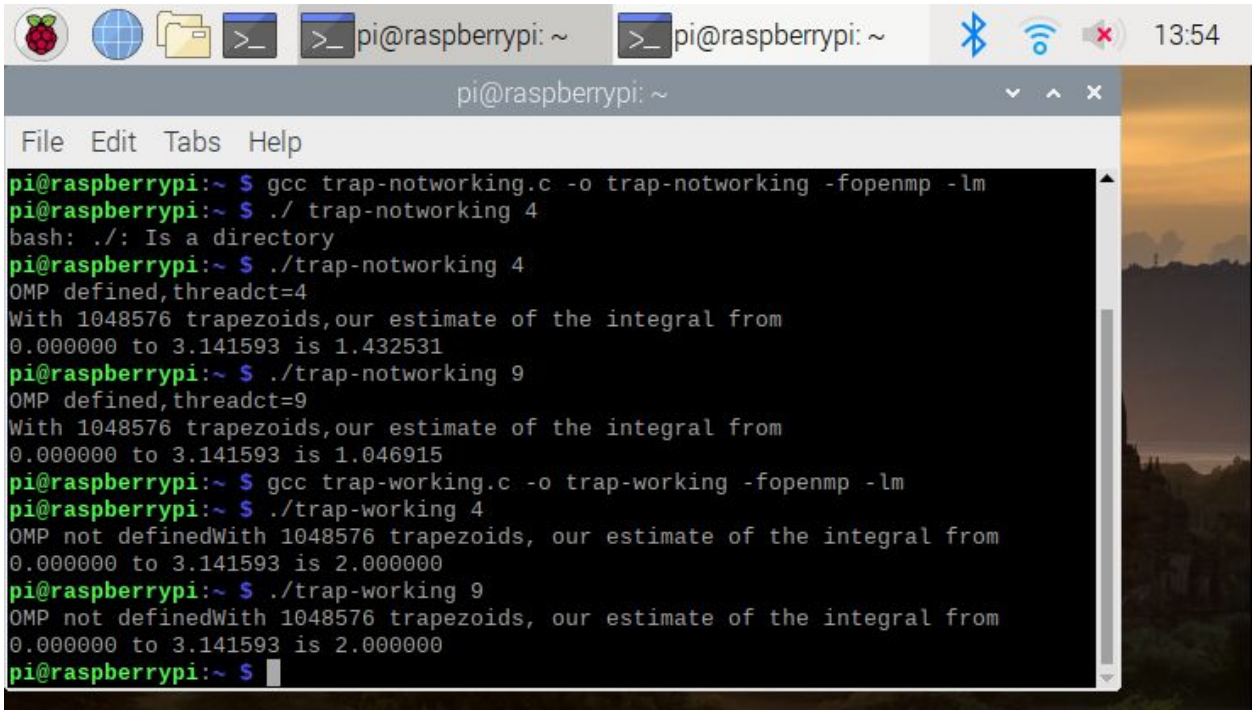

pi@raspberrypi: ~
pi@raspberrypi: ~
13:56
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 trap-notworking.c
/* Variables */
double a = 0.0, b = pi;
int n = 1048576;
double h = (b-a) /n;
double integral;
int threadcnt = 1;
double f(double x);
/* parse command-line arg for number of threads */
if(argc>1){
threadcnt = atoi(argv[1]);
}
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line


pi@raspberrypi: ~
pi@raspberrypi: ~
13:57
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 trap-notworking.c
double f(double x);
/* parse command-line arg for number of threads */
if(argc>1){
threadcnt = atoi(argv[1]);
}
#ifdef _OPENMP
omp_set_num_threads(threadcnt);
printf("OMP defined,threadct=%d\n", threadcnt);
#else
printf("OMP not defined");
#endif
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line

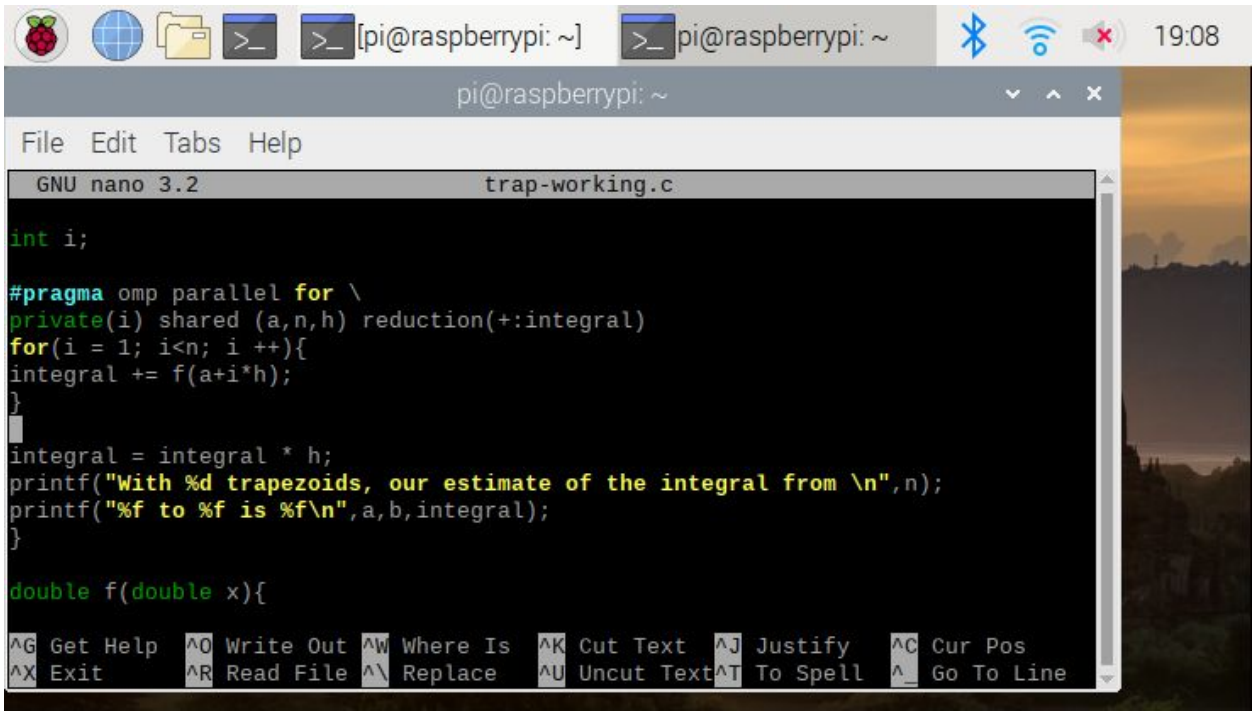

pi@raspberrypi: ~
pi@raspberrypi: ~
13:57
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 trap-notworking.c
}
#ifdef _OPENMP
omp_set_num_threads(threadcnt);
printf("OMP defined,threadct=%d\n", threadcnt);
#else
printf("OMP not defined");
#endif
integral = (f(a)+f(b))/2.0;
int i;
#pragma omp parallel for private(i) shared (a,n,h,integral)
for(i =1; i <n; i ++){
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line



These are the screenshots for the code for trap-notworking.c.

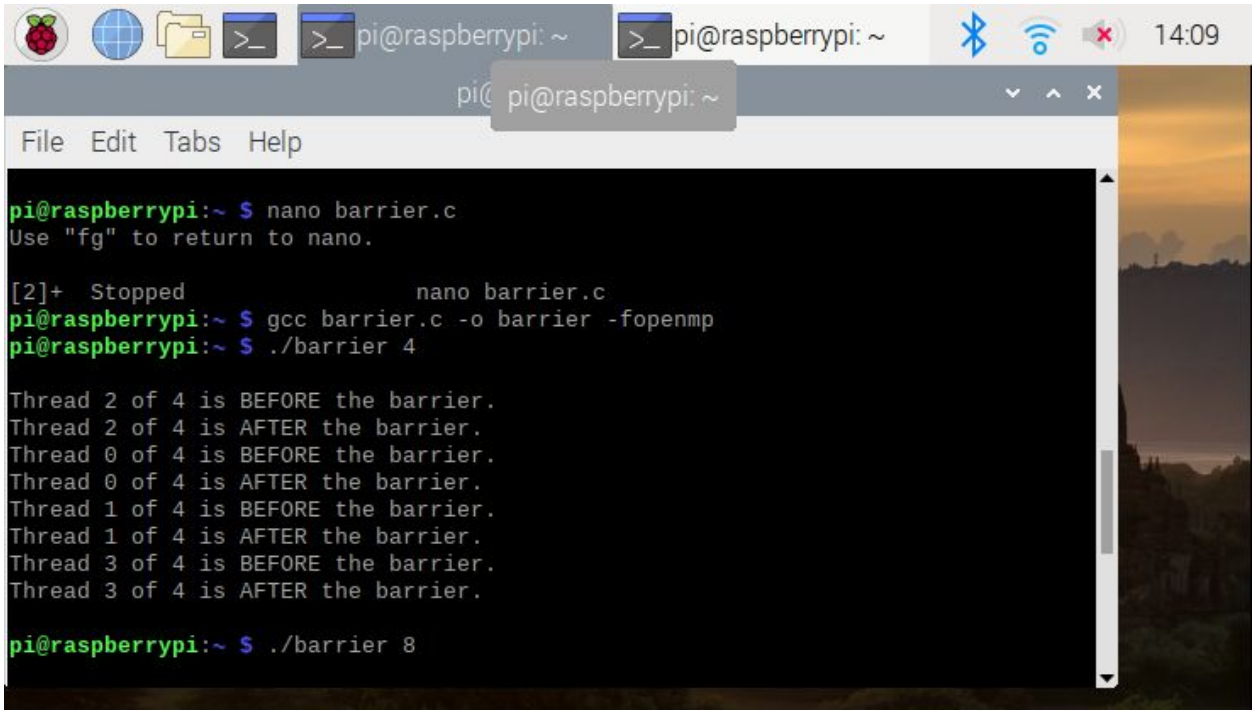


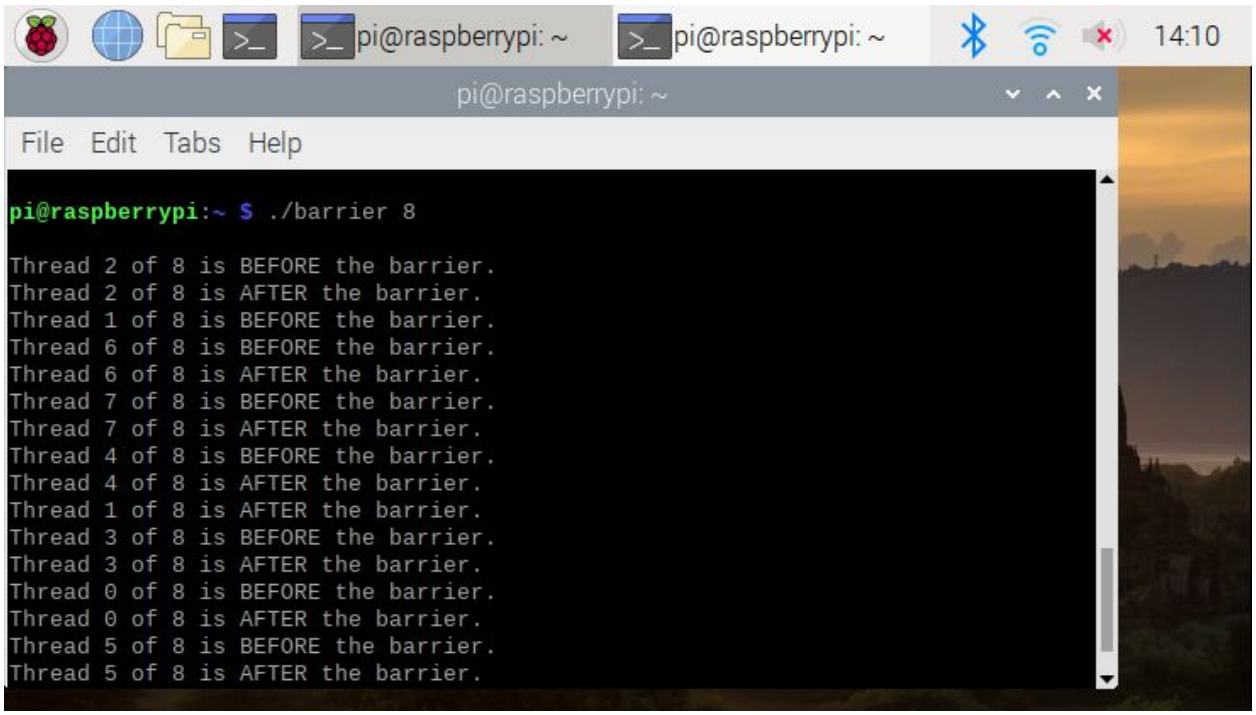
In this screenshot I am running trap-working,c and as you can see the output value is 2.0 which is the intended result. This code works after I added the given code segment, because now the code is encapsulated and we are implementing the reduction.

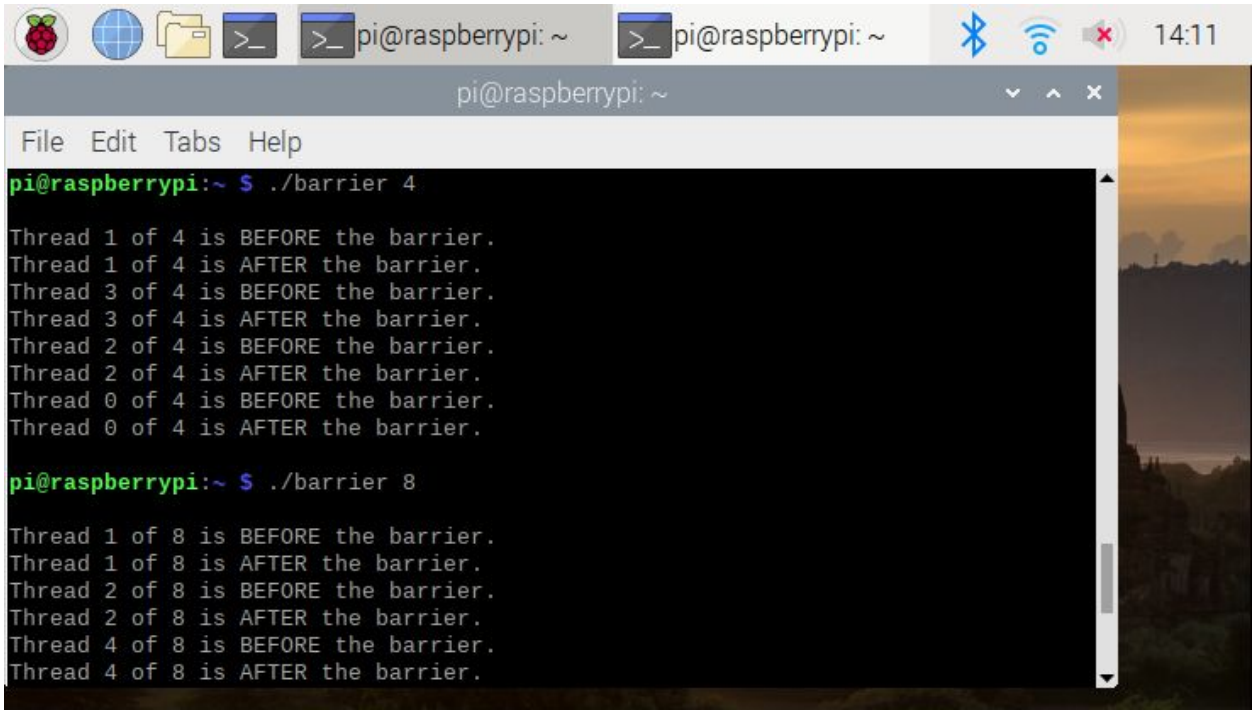


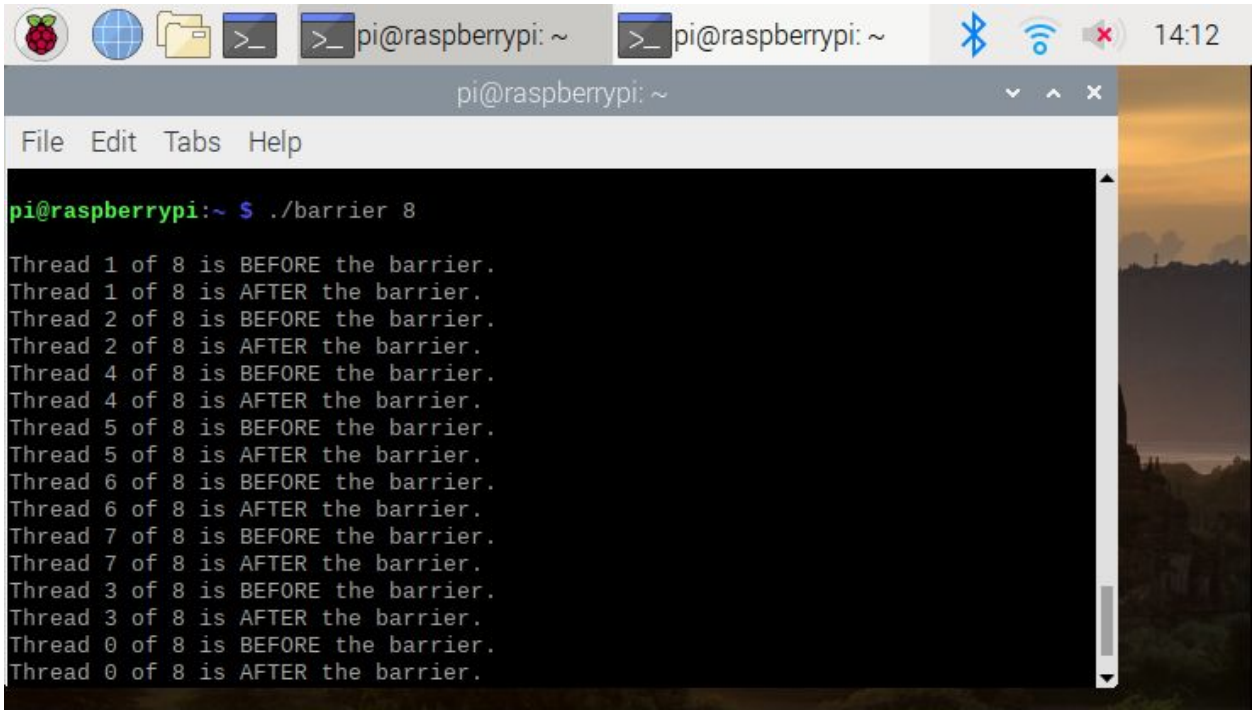
In this screenshot you can see the code for trap-working.c and the code that is different from trap-notworking.c.

Part 2

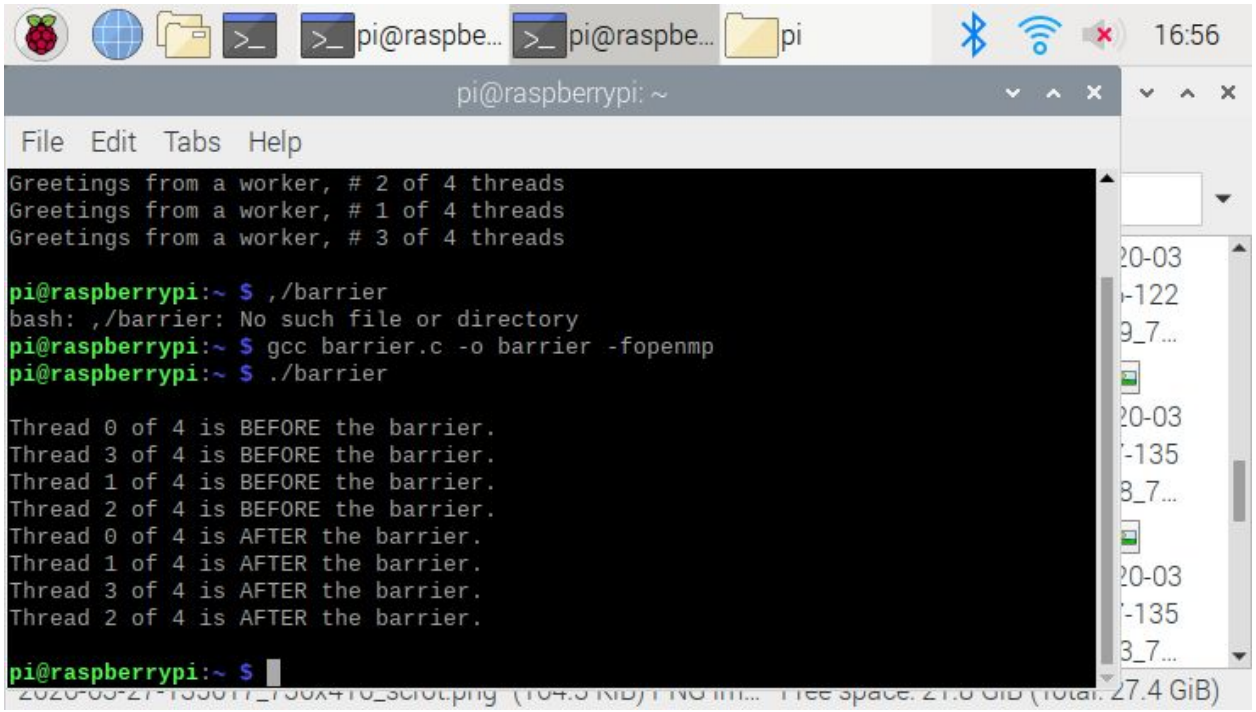

pi@raspberrypi: ~
pi@raspberrypi: ~
14:09
pi@raspberrypi: ~
File Edit Tabs Help
pi@raspberrypi:~ $ nano barrier.c
Use "fg" to return to nano.
[2]+ Stopped nano barrier.c
pi@raspberrypi:~ $ gcc barrier.c -o barrier -fopenmp
pi@raspberrypi:~ $ ./barrier 4
Thread 2 of 4 is BEFORE the barrier.
Thread 2 of 4 is AFTER the barrier.
Thread 0 of 4 is BEFORE the barrier.
Thread 0 of 4 is AFTER the barrier.
Thread 1 of 4 is BEFORE the barrier.
Thread 1 of 4 is AFTER the barrier.
Thread 3 of 4 is BEFORE the barrier.
Thread 3 of 4 is AFTER the barrier.
pi@raspberrypi:~ $ ./barrier 8

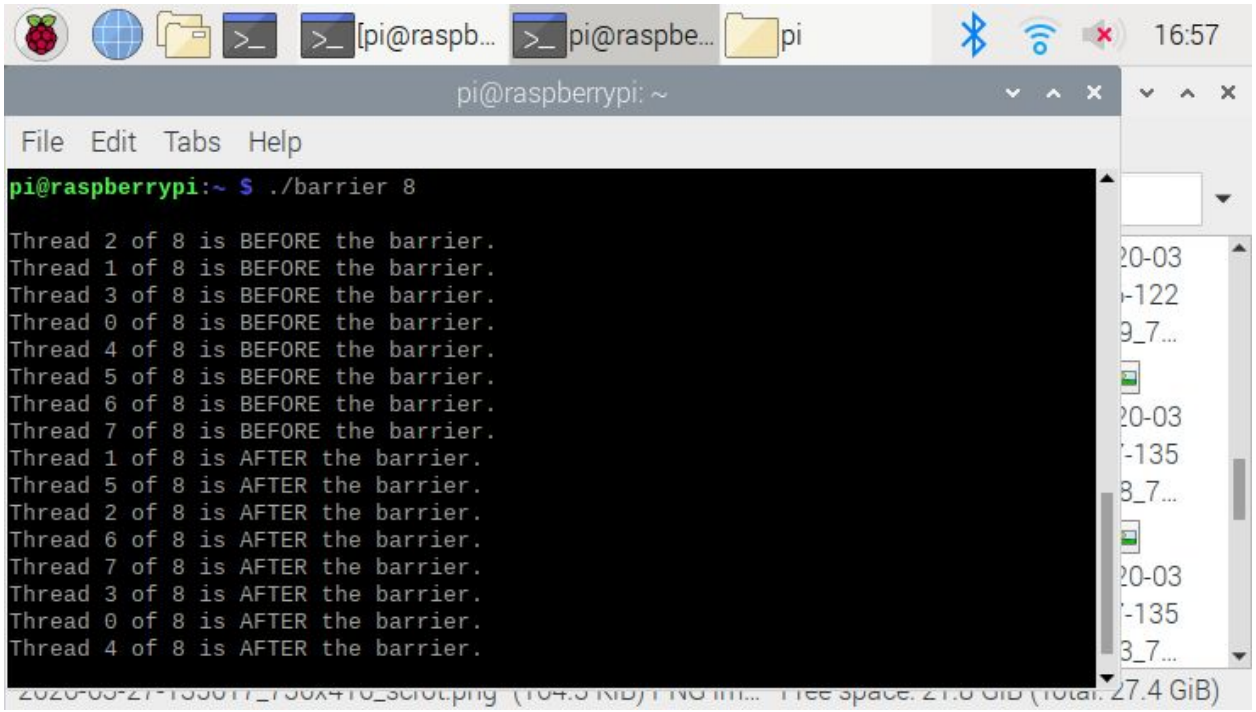

pi@raspberrypi: ~
pi@raspberrypi: ~
14:10
pi@raspberrypi: ~
File Edit Tabs Help
pi@raspberrypi:~ $ ./barrier 8
Thread 2 of 8 is BEFORE the barrier.
Thread 2 of 8 is AFTER the barrier.
Thread 1 of 8 is BEFORE the barrier.
Thread 6 of 8 is BEFORE the barrier.
Thread 6 of 8 is AFTER the barrier.
Thread 7 of 8 is BEFORE the barrier.
Thread 7 of 8 is AFTER the barrier.
Thread 4 of 8 is BEFORE the barrier.
Thread 4 of 8 is AFTER the barrier.
Thread 1 of 8 is AFTER the barrier.
Thread 3 of 8 is BEFORE the barrier.
Thread 3 of 8 is AFTER the barrier.
Thread 0 of 8 is BEFORE the barrier.
Thread 0 of 8 is AFTER the barrier.
Thread 5 of 8 is BEFORE the barrier.
Thread 5 of 8 is AFTER the barrier.



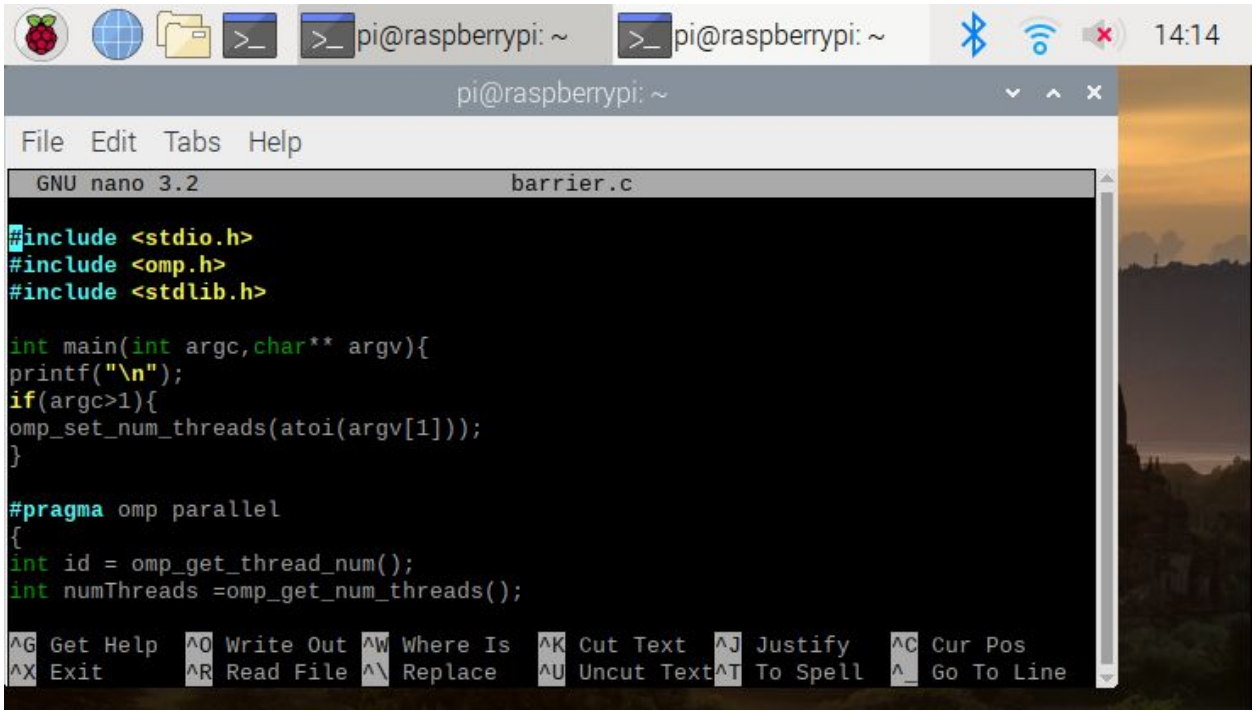


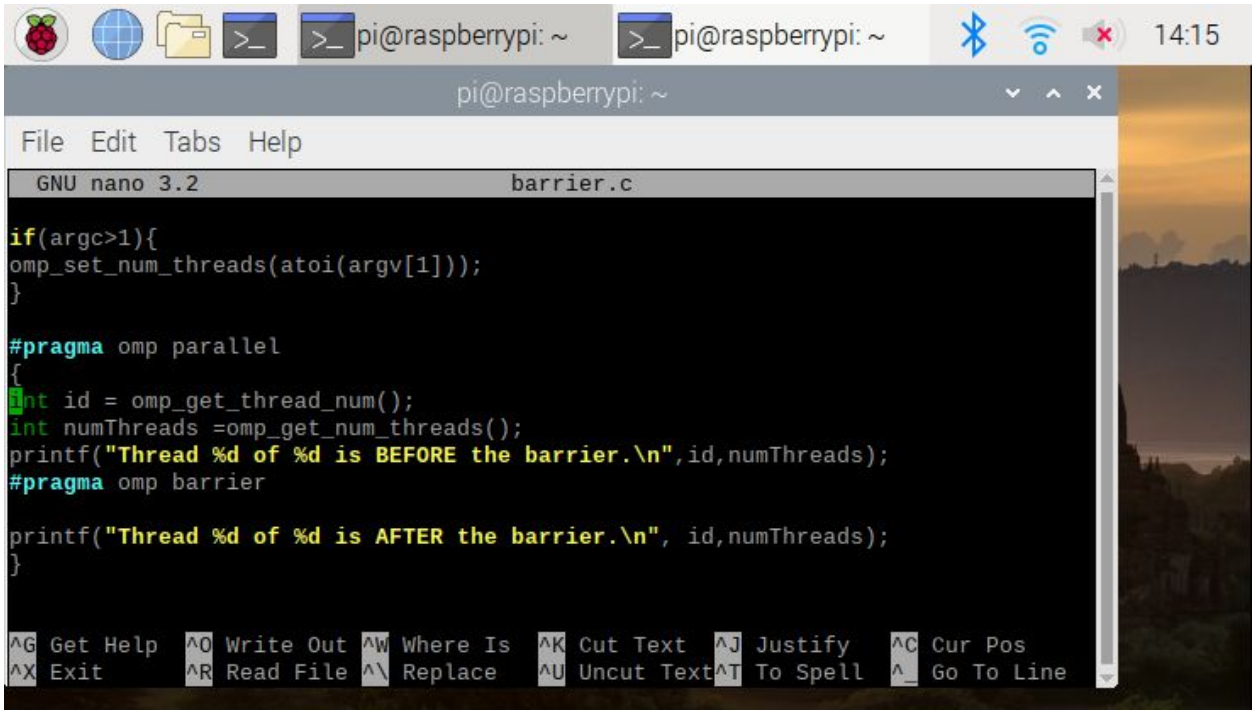
These screenshots are showing the commented version of barrier.c. As you can see I tried out different values to see if I can notice any patterns. This is not the intended result of the program because it should display all the BEFOREs and then all the AFTERs. All the threads have to complete their task before they can move on to the next task.

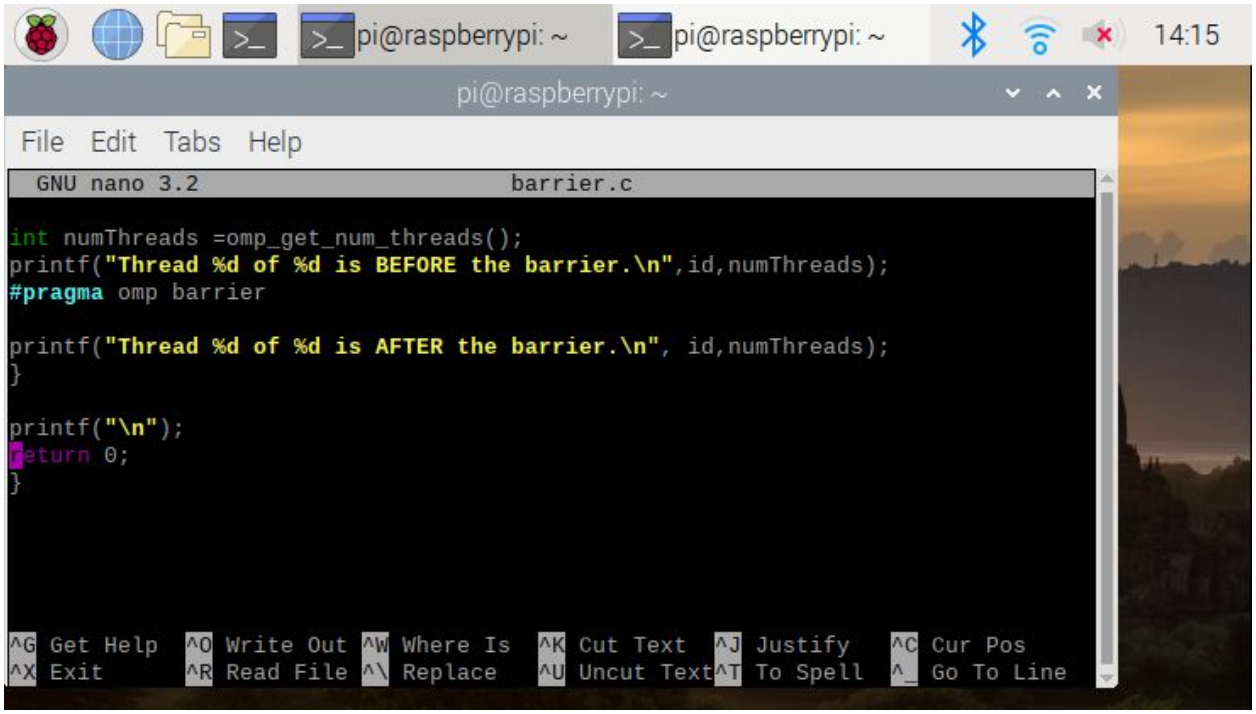




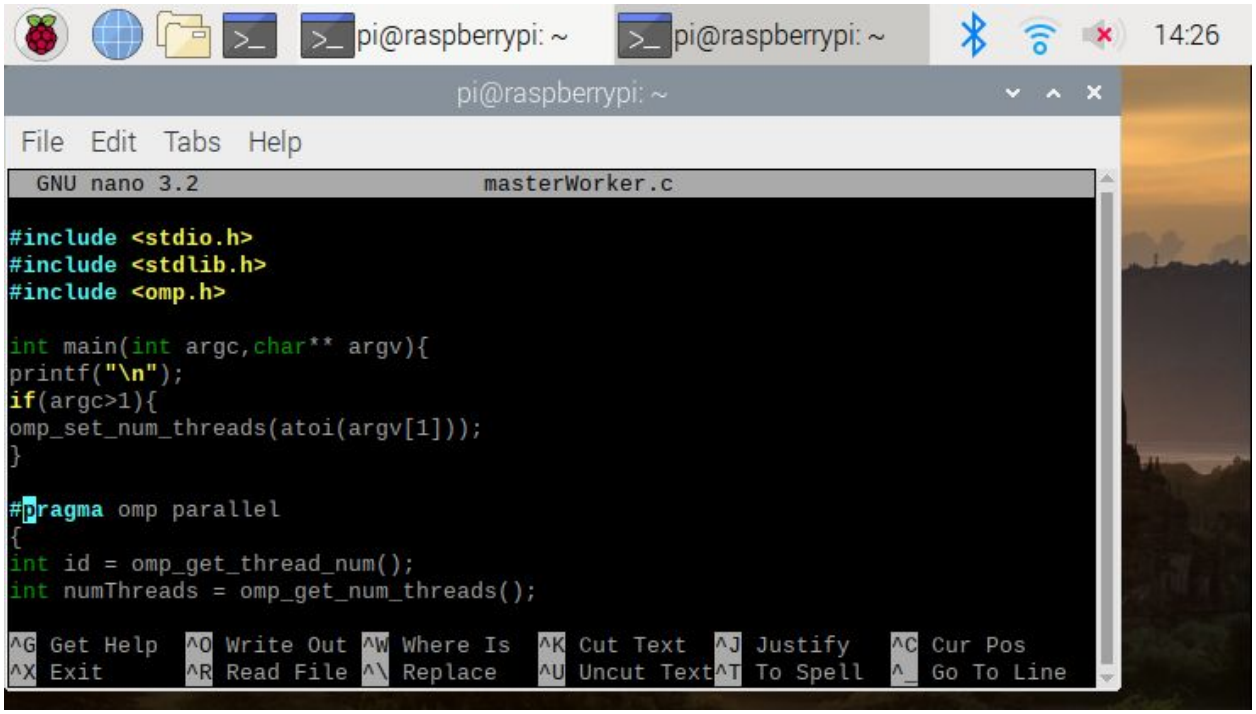
These screenshots are showing the uncommented version of the code. As you can see the code worked as it was intended to. This is because the code is finishing the sequence before it continues to execute their next task effectively using barriers.

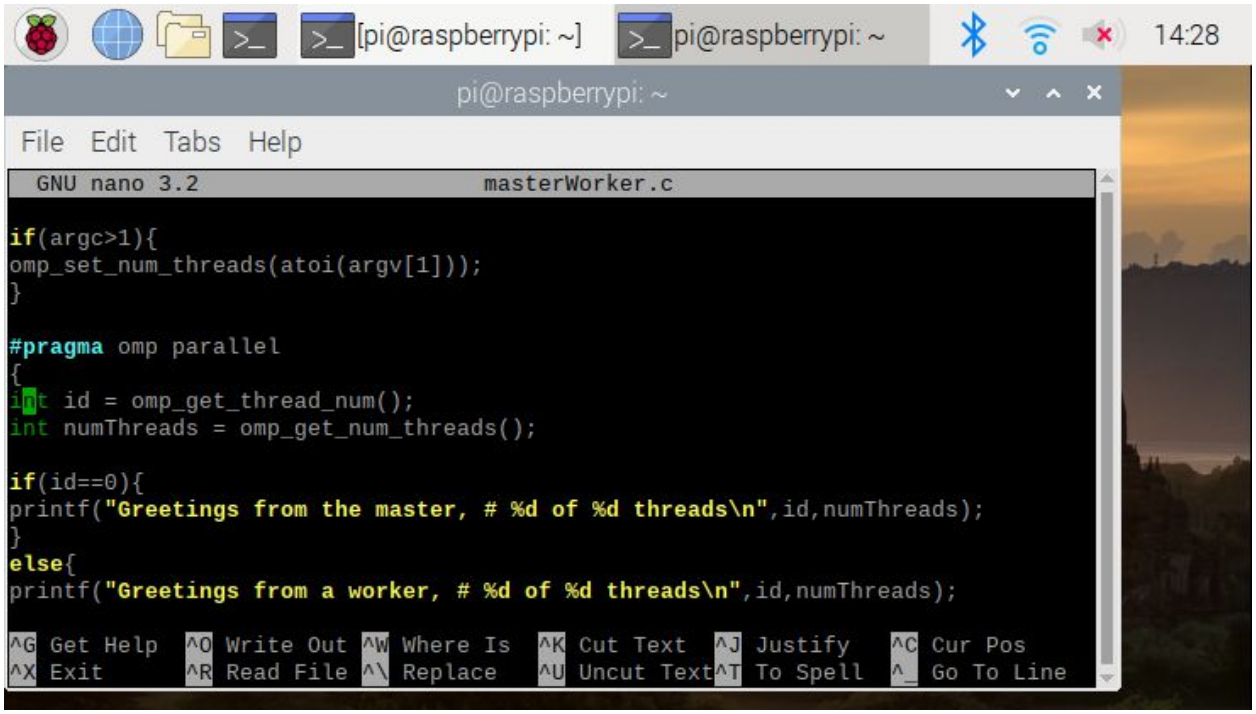

pi@raspberrypi: ~
pi@raspberrypi: ~
14:14
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 barrier.c
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
int main(int argc,char** argv){
printf("\n");
if(argc>1){
omp_set_num_threads(atoi(argv[1]));
}
#pragma omp parallel
{
int id = omp_get_thread_num();
int numThreads =omp_get_num_threads();
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line

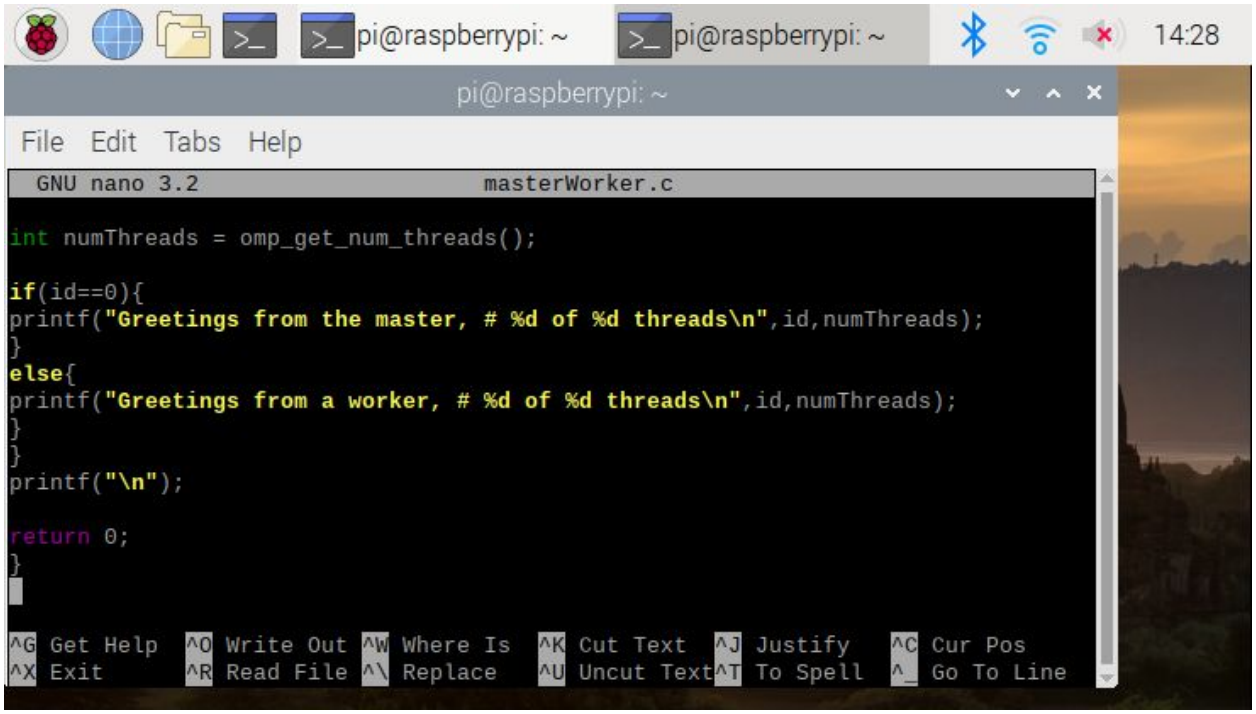

pi@raspberrypi: ~
pi@raspberrypi: ~
14:15
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 barrier.c
if(argc>1){
omp_set_num_threads(atoi(argv[1]));
}
#pragma omp parallel
{
int id = omp_get_thread_num();
int numThreads =omp_get_num_threads();
printf("Thread %d of %d is BEFORE the barrier.\n",id,numThreads);
#pragma omp barrier
printf("Thread %d of %d is AFTER the barrier.\n", id,numThreads);
}
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line



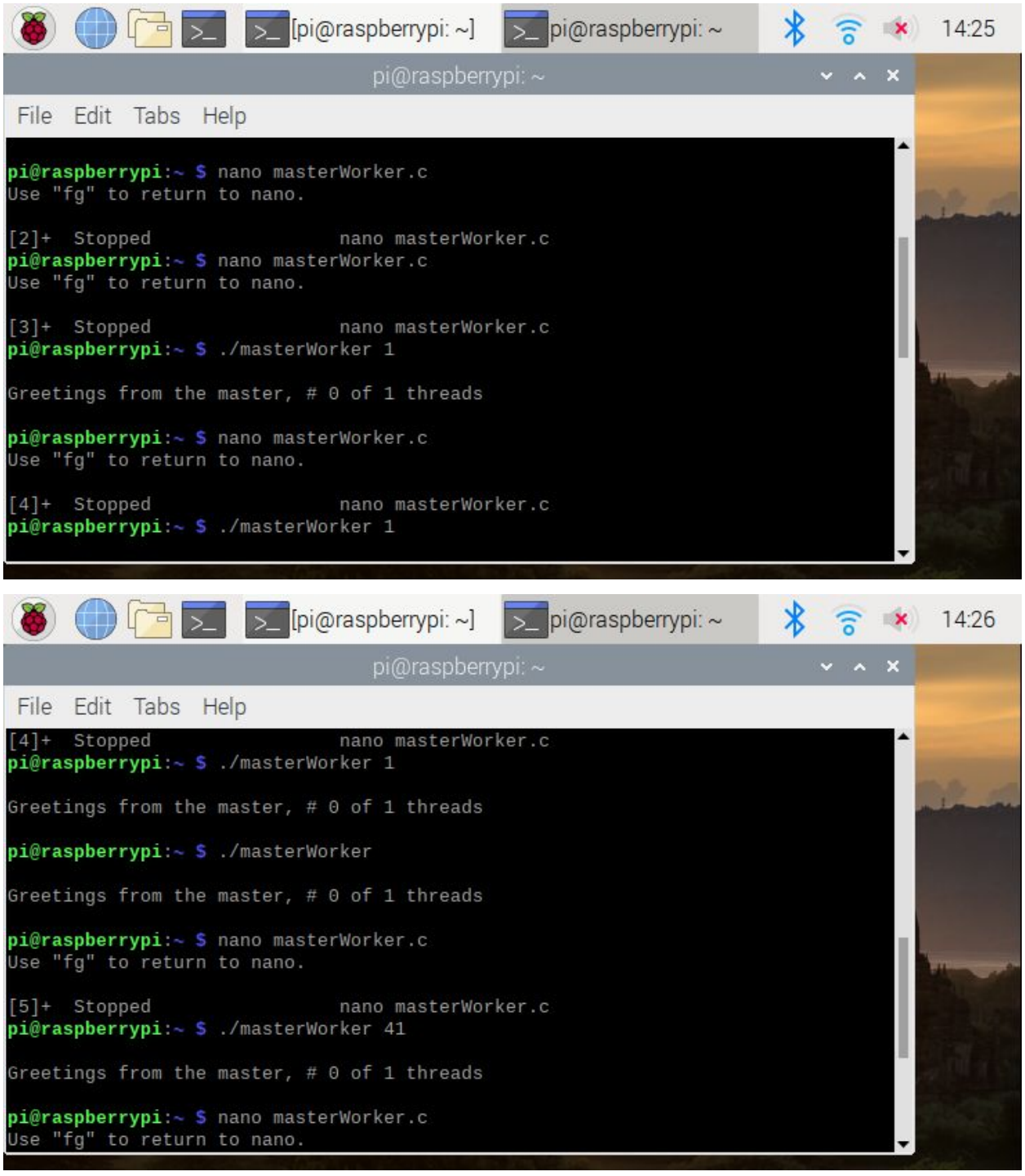
These screenshots are the code for barrier.c.


pi@raspberrypi: ~
pi@raspberrypi: ~
14:26
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 masterWorker.c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(int argc,char** argv){
printf("\n");
if(argc>1){
omp_set_num_threads(atoi(argv[1]));
}
#pragma omp parallel
{
int id = omp_get_thread_num();
int numThreads = omp_get_num_threads();
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line

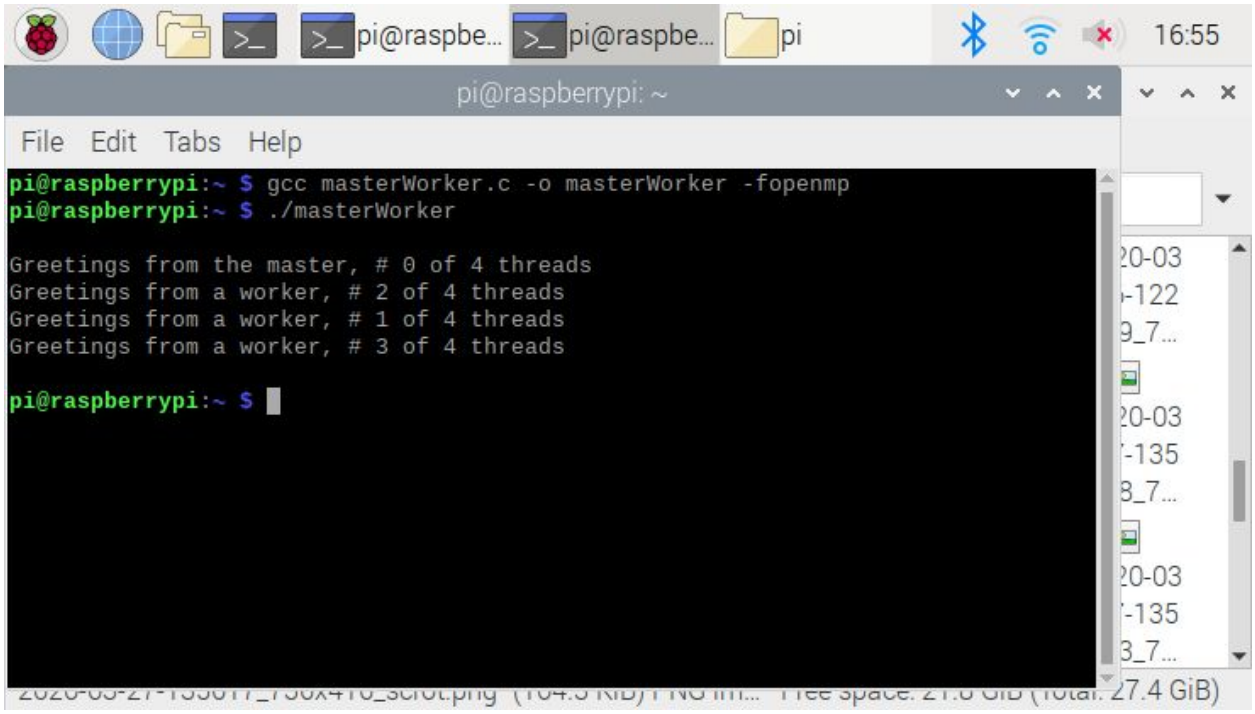

[pi@raspberrypi: ~]
pi@raspberrypi: ~
14:28
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 masterWorker.c
if(argc>1){
omp_set_num_threads(atoi(argv[1]));
}
#pragma omp parallel
{
int id = omp_get_thread_num();
int numThreads = omp_get_num_threads();
if(id==0){
printf("Greetings from the master, # %d of %d threads\n",id,numThreads);
}
else{
printf("Greetings from a worker, # %d of %d threads\n",id,numThreads);
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line



These screenshots are the code for masterWorker.c



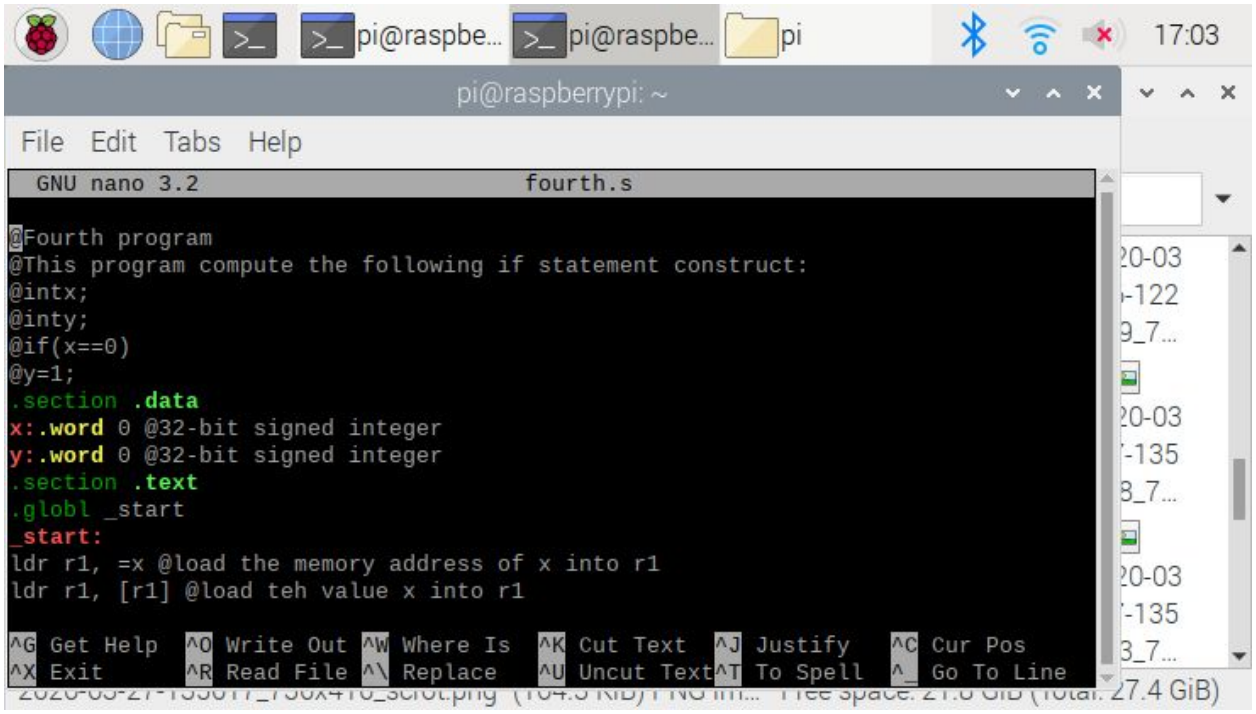
This is the commented version of the code. As you can see no matter what value I put afterwards it gives the same result. This is because it is only showing us the master thread.

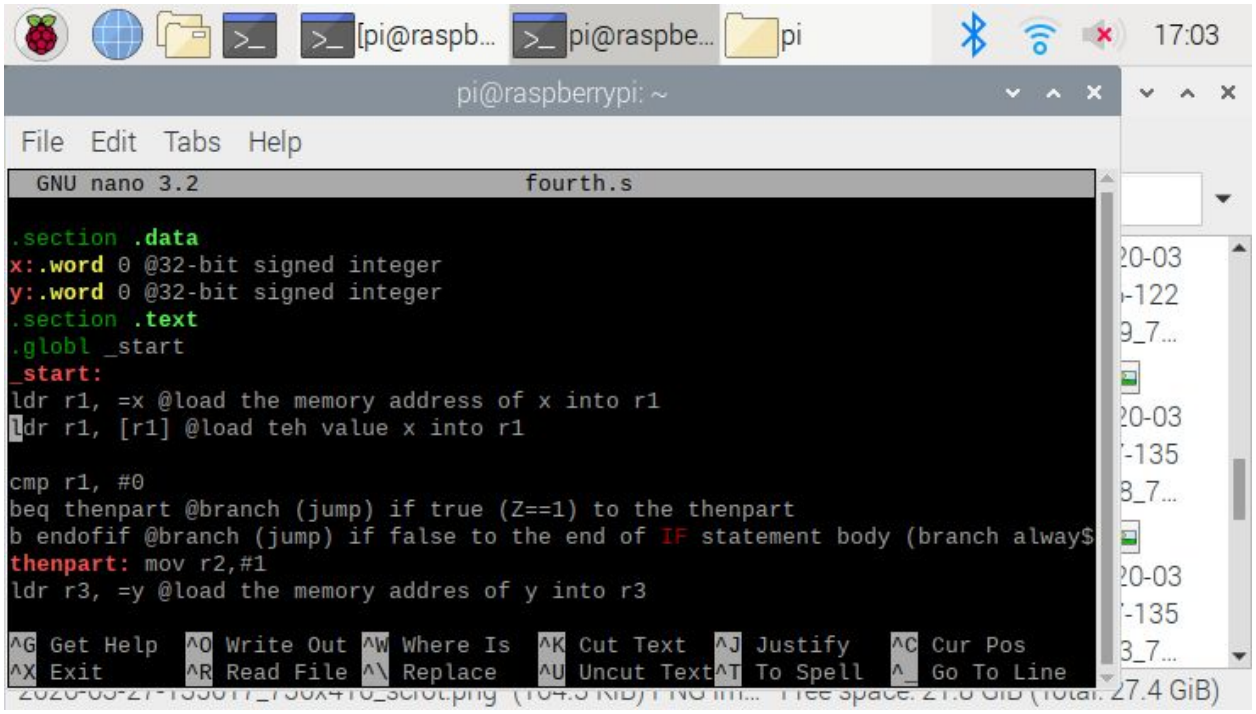


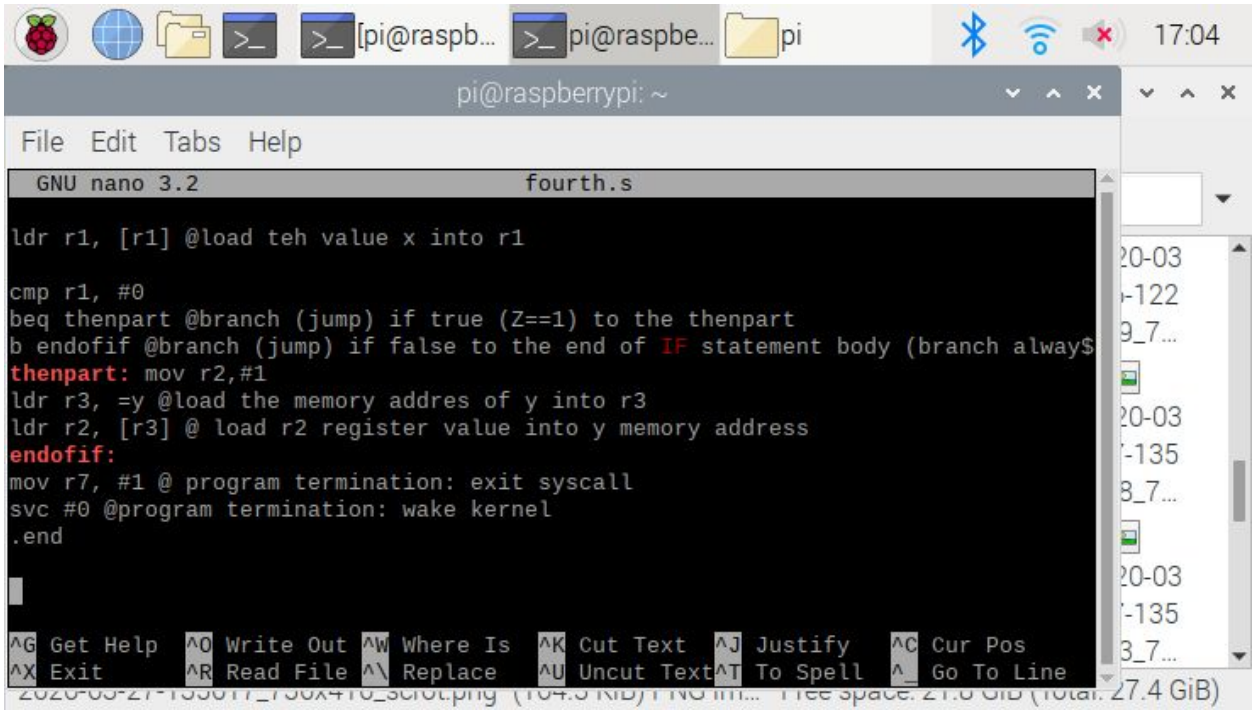
This is the uncommented version of the code. In the uncommented version of the code you can see that the code is working as intended and you have the master and the workers saying their phrases. In this screenshot you can see that the master is executing it's designated segment and then it forks out to the workers where they execute different blocks as well. This will greatly help out when it comes to the organization of the code.

ARM Assembly Programming:

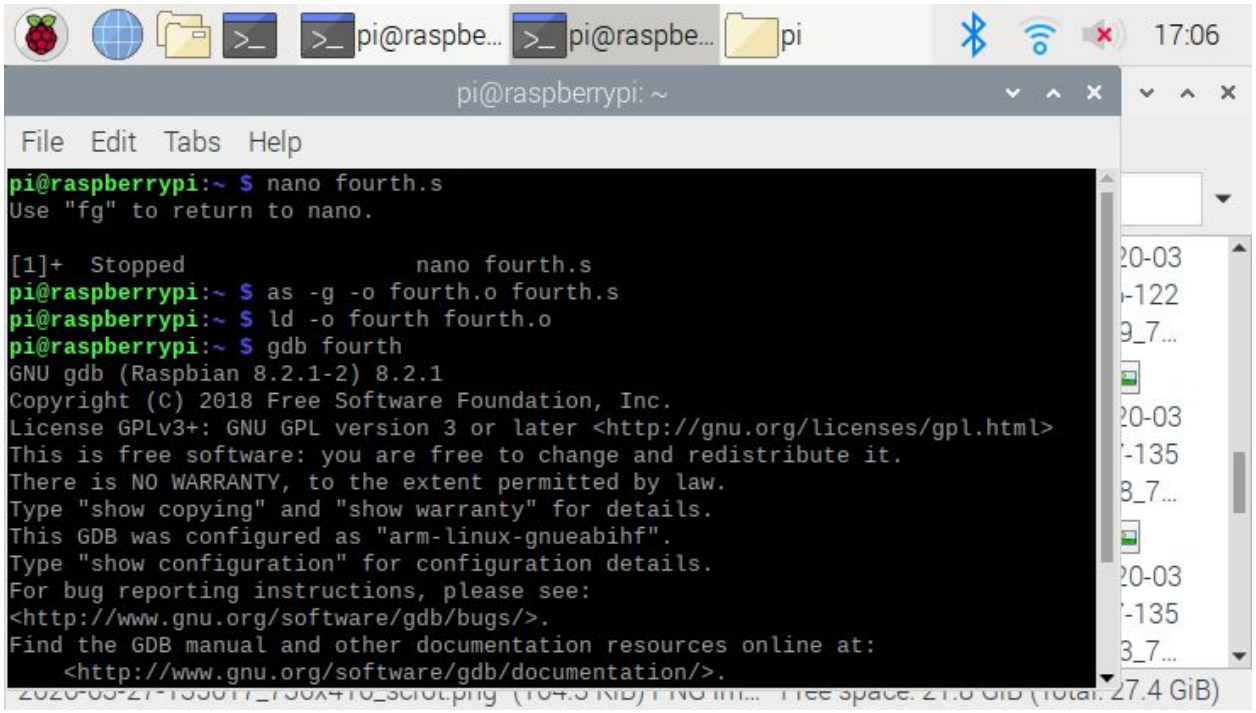
Part 1

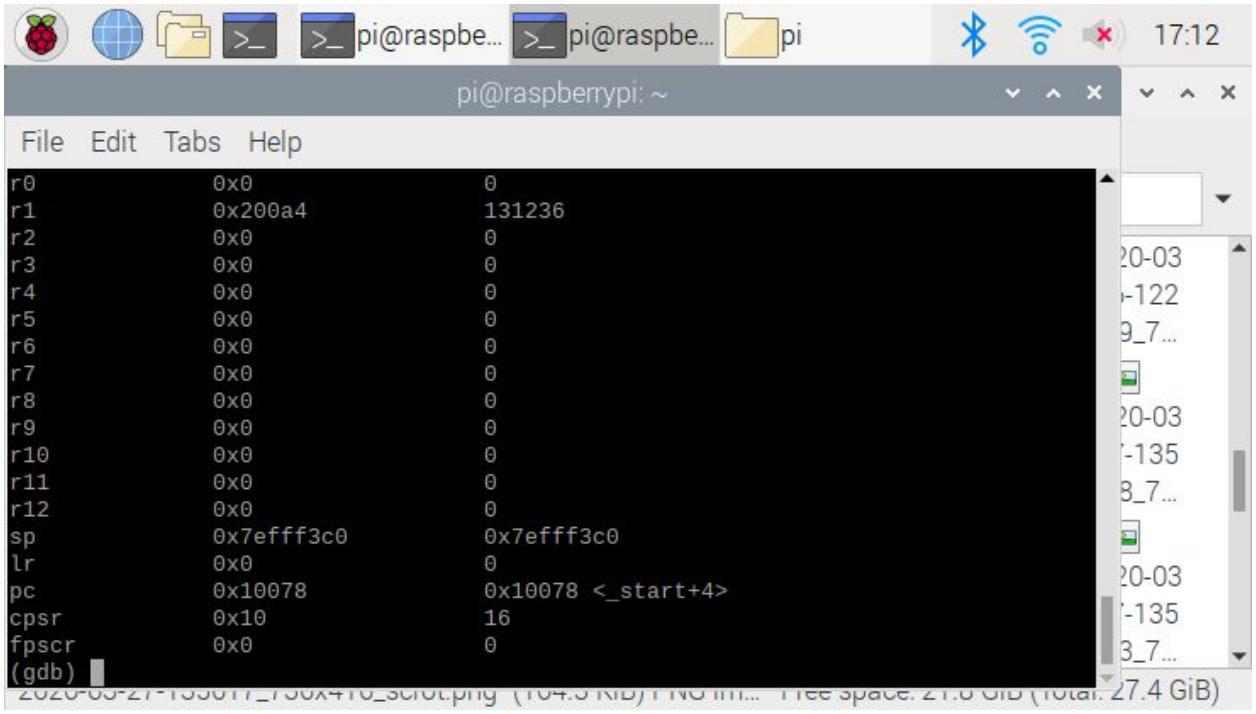


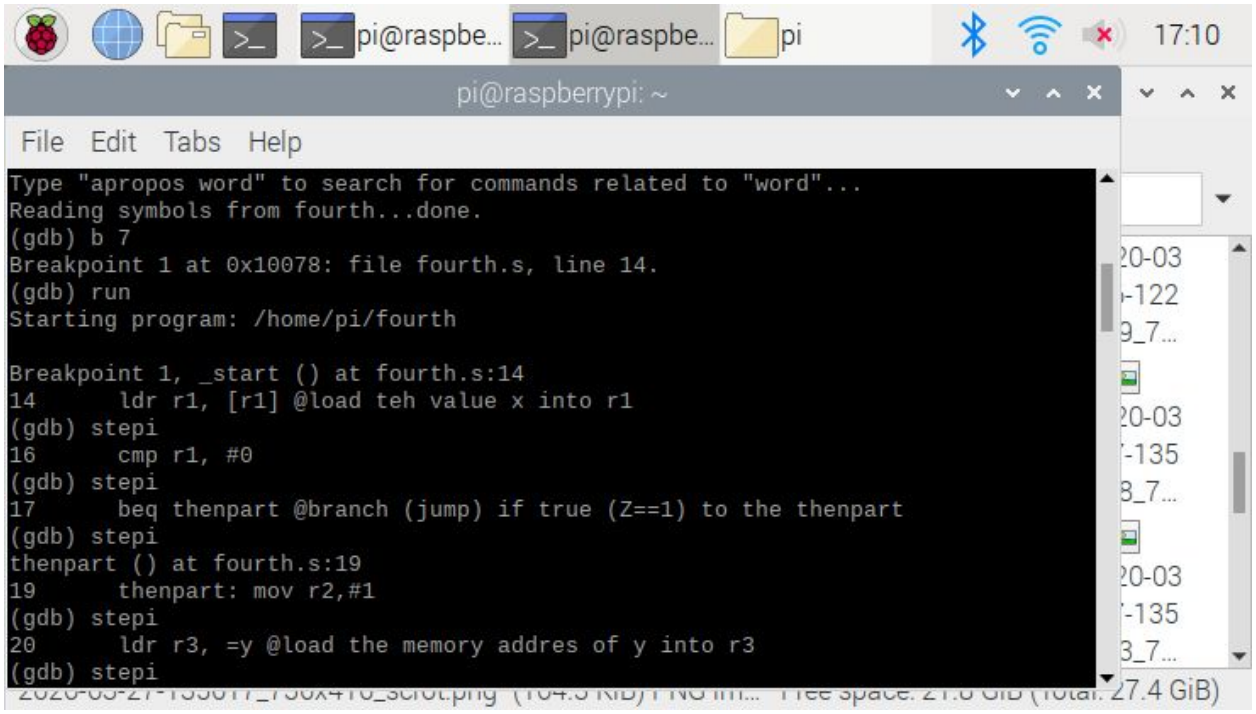


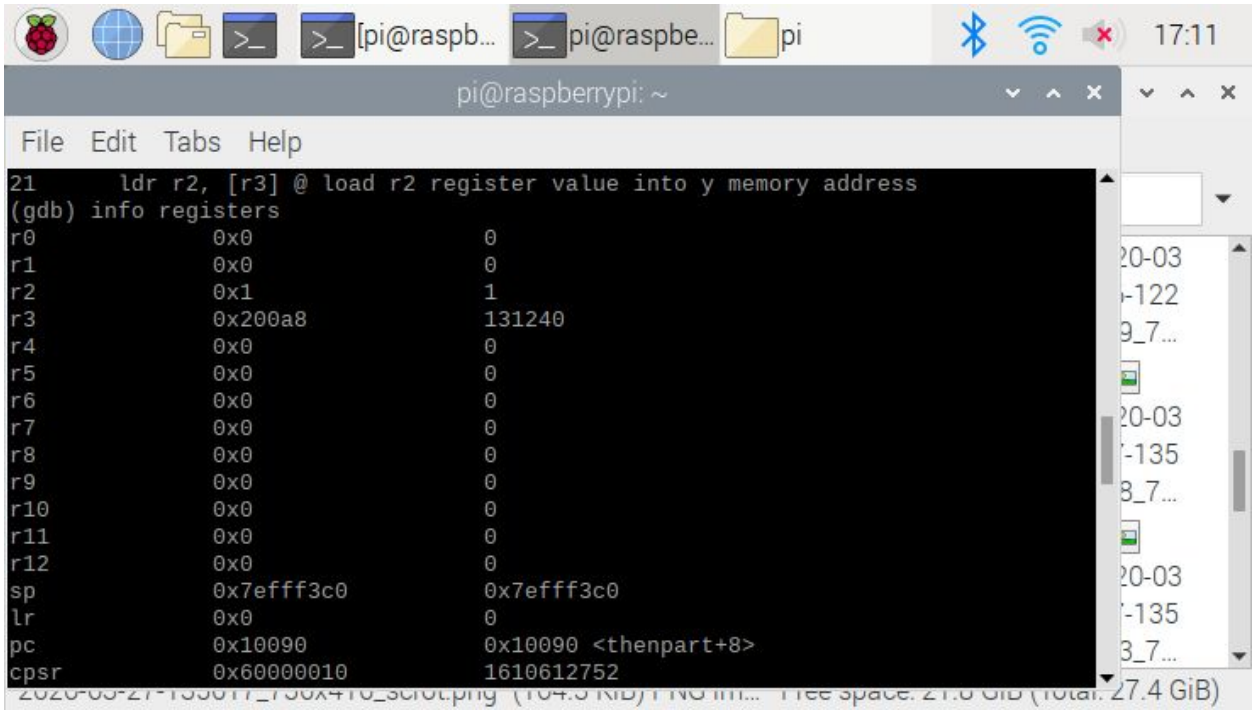


These screenshots are the code for part 1.

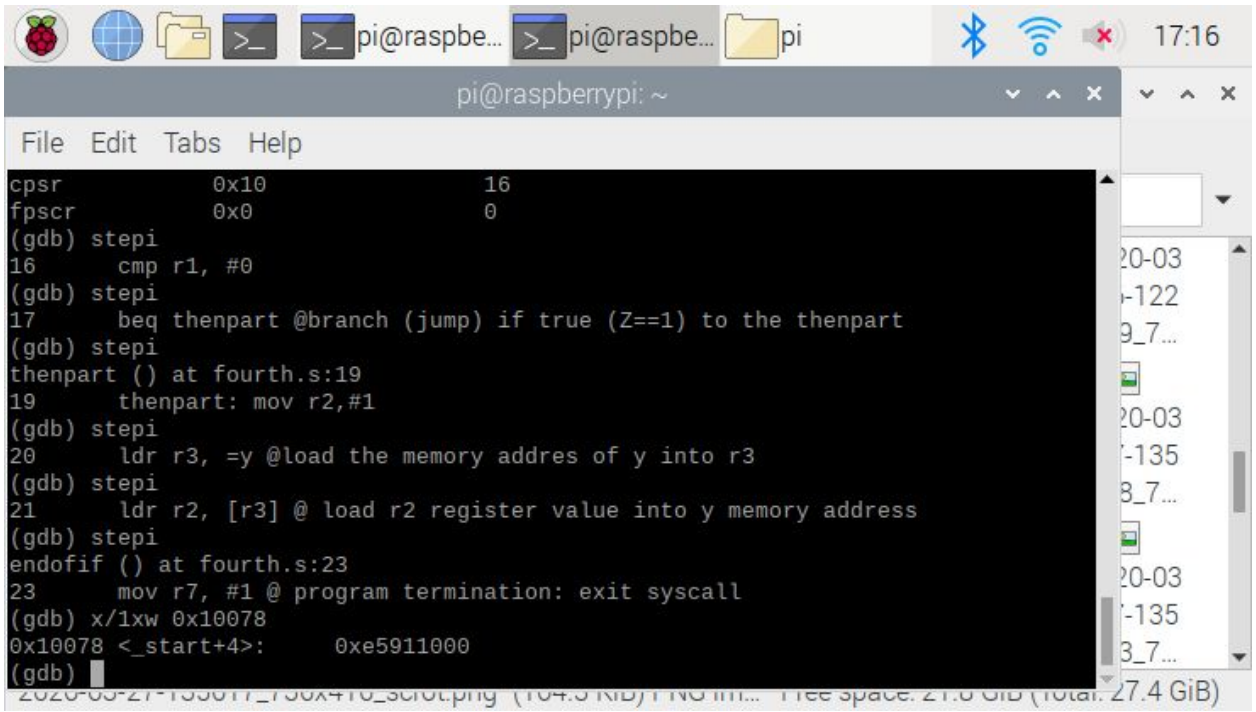


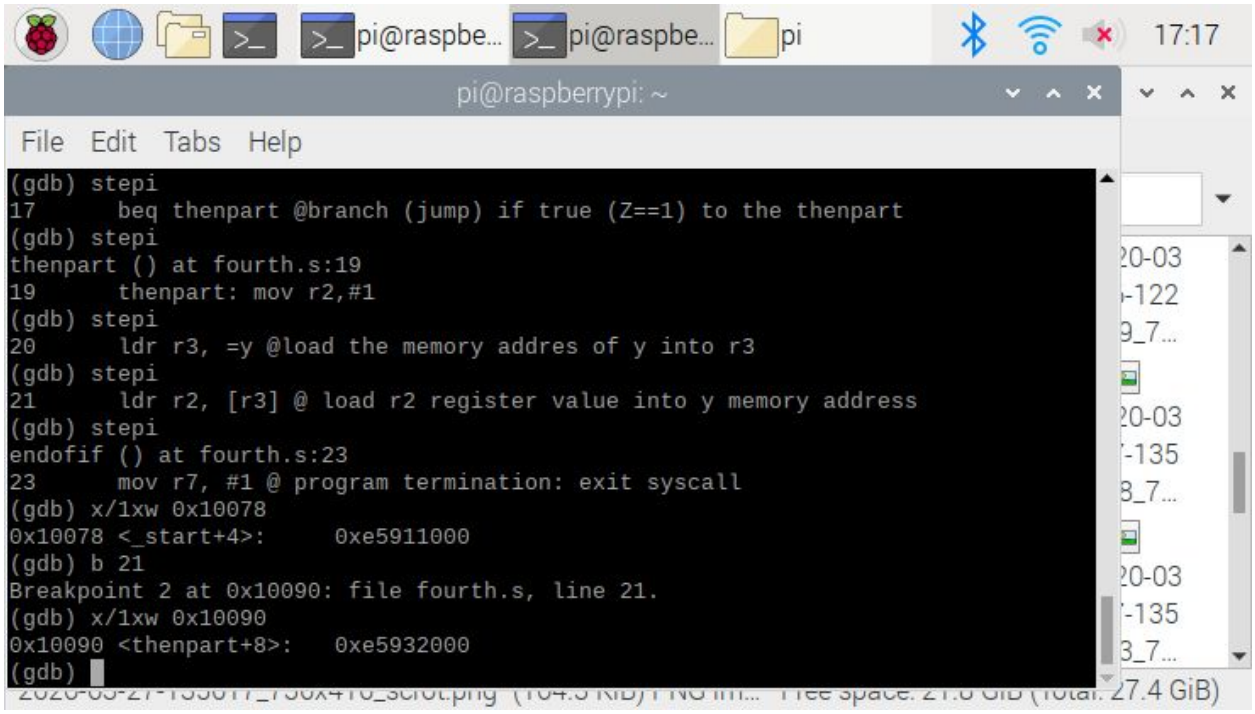

pi@raspbe...
pi@raspbe...
pi
17:12
pi@raspberrypi: ~
File Edit Tabs Help
r0 0x0 0
r1 0x200a4 131236
r2 0x0 0
r3 0x0 0
r4 0x0 0
r5 0x0 0
r6 0x0 0
r7 0x0 0
r8 0x0 0
r9 0x0 0
r10 0x0 0
r11 0x0 0
r12 0x0 0
sp 0x7efff3c0 0x7efff3c0
lr 0x0 0
pc 0x10078 0x10078 <_start+4>
cpsr 0x10 16
fpscr 0x0 0
(gdb)


pi@raspbe...
pi@raspbe...
pi
17:10
pi@raspberrypi: ~
File Edit Tabs Help
Type "apropos word" to search for commands related to "word"...
Reading symbols from fourth...done.
(gdb) b 7
Breakpoint 1 at 0x10078: file fourth.s, line 14.
(gdb) run
Starting program: /home/pi/fourth
Breakpoint 1, _start () at fourth.s:14
14 ldr r1, [r1] @load teh value x into r1
(gdb) stepi
16 cmp r1, #0
(gdb) stepi
17 beq thenpart @branch (jump) if true (Z==1) to the thenpart
(gdb) stepi
thenpart () at fourth.s:19
19 thenpart: mov r2,#1
(gdb) stepi
20 ldr r3, =y @load the memory addres of y into r3
(gdb) stepi



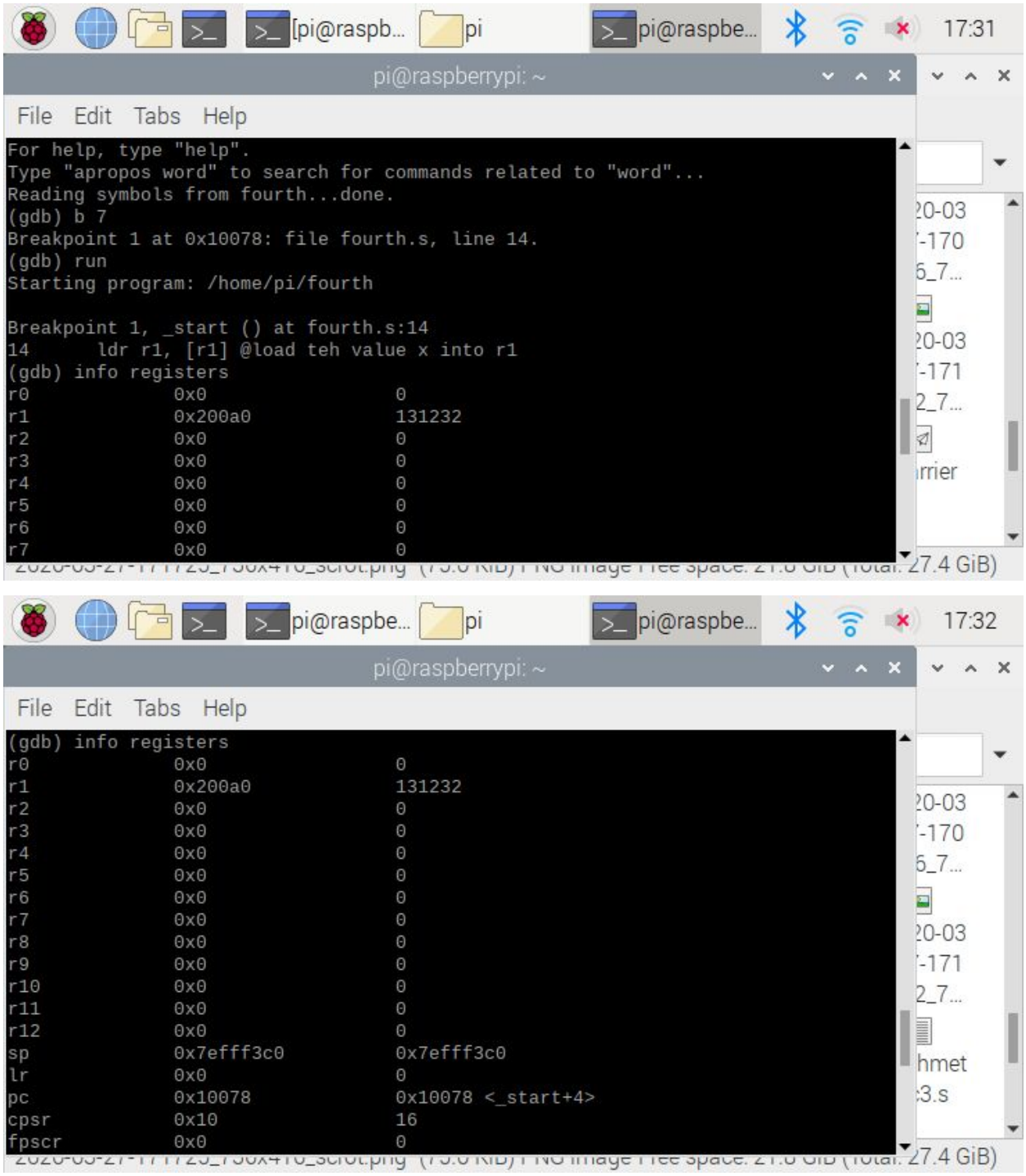
In these screenshots I assembled and linked the program. I then placed breakpoints and started stepping through the different lines of code to see what was in each register. The final value that r3 ends with is 0x200a8. The Z flag that I got was 0 because the value is not 0.



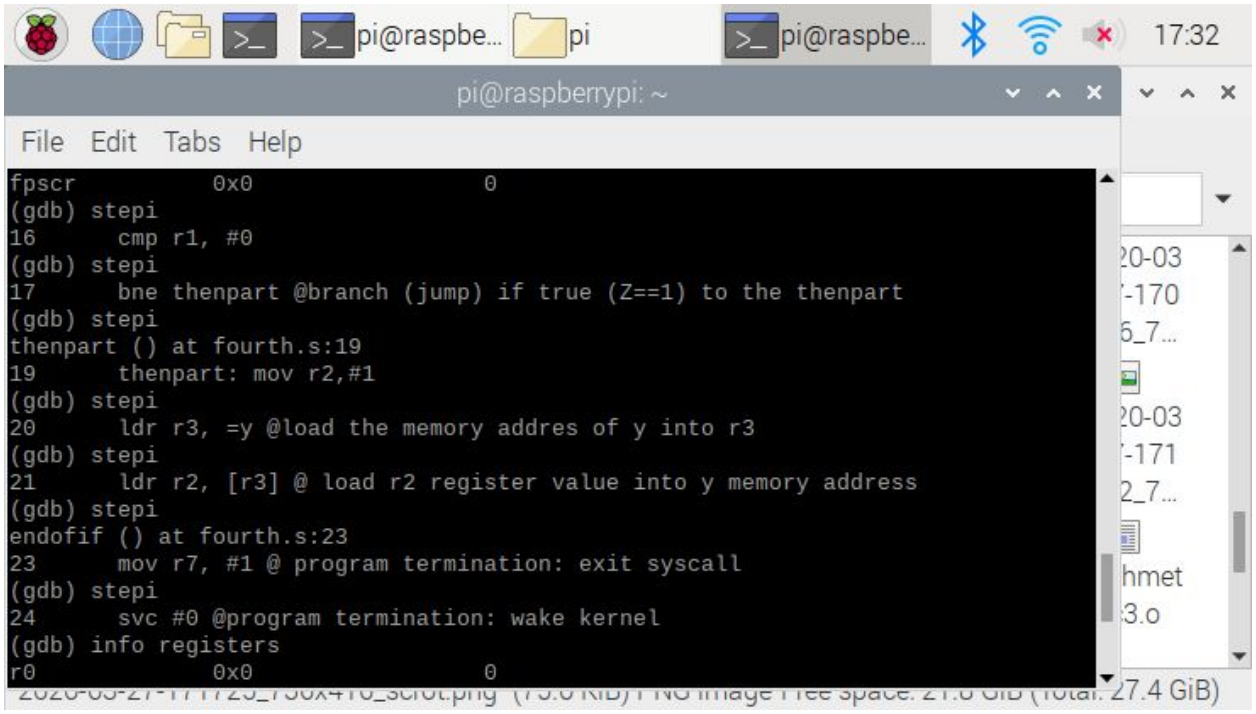


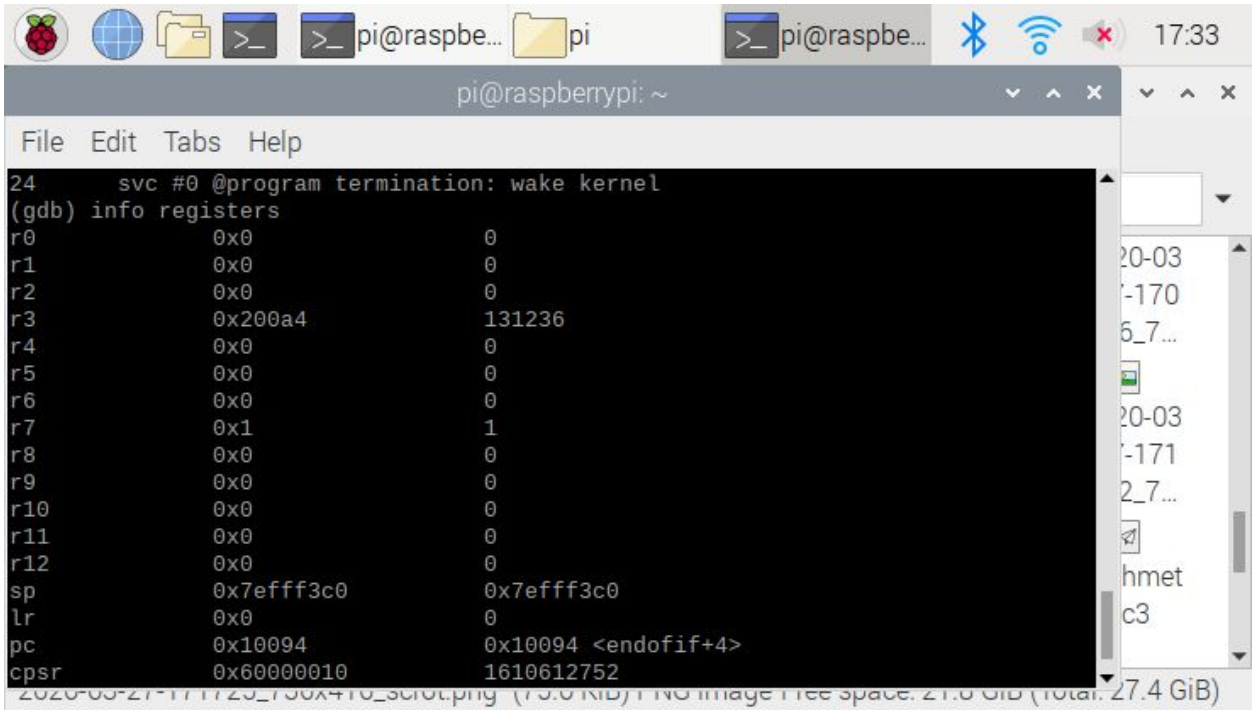
In these screenshots I displayed the memory address at location 0x10078 which is 0xe5911000 and the memory address at location 0x10090 which is 0xe5932000.

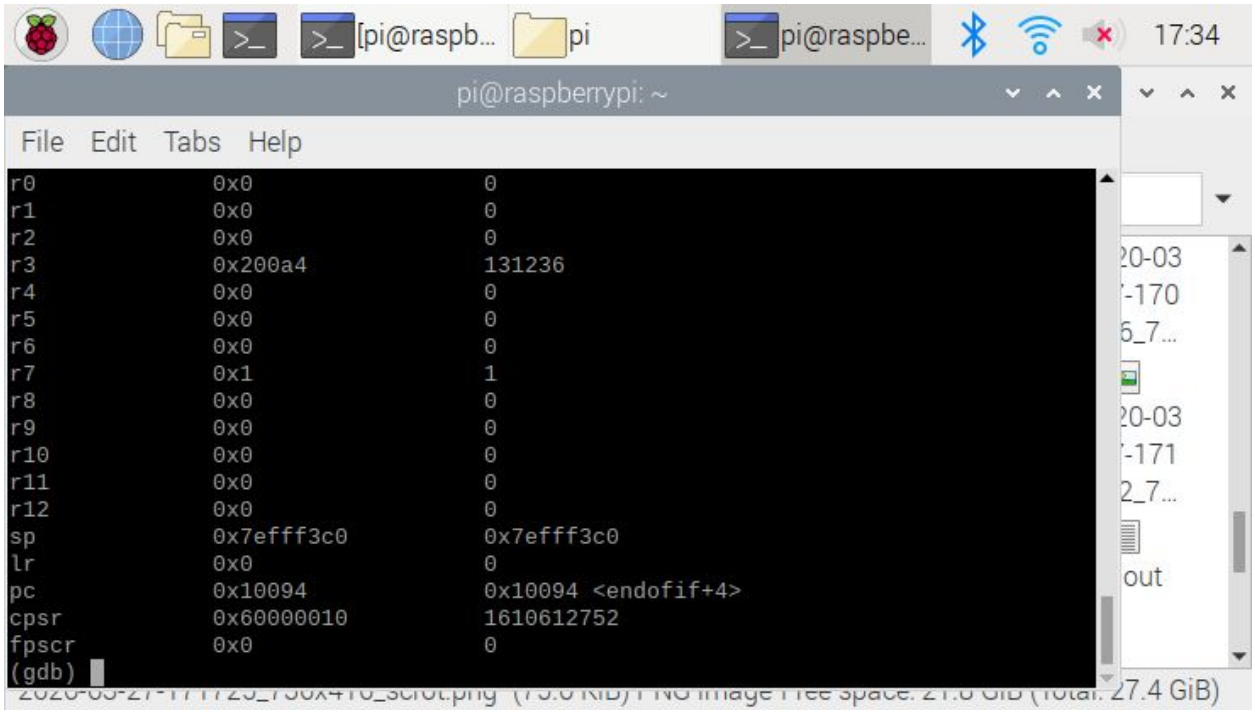
Part 2



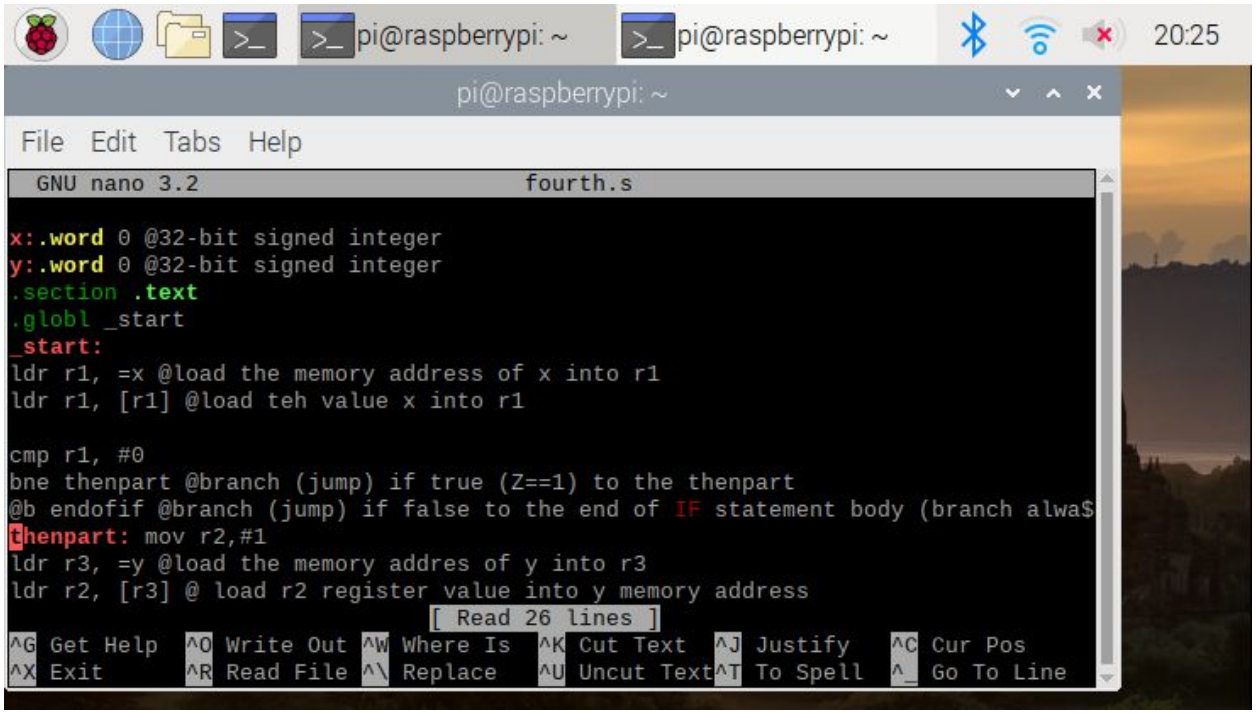
These are the registers at the first breakpoint.


pi@raspbe...
pi
pi@raspbe...
17:32
pi@raspberrypi: ~
File Edit Tabs Help
fpscr 0x0 0
(gdb) stepi
16 cmp r1, #0
(gdb) stepi
17 bne thenpart @branch (jump) if true (Z==1) to the thenpart
(gdb) stepi
thenpart () at fourth.s:19
19 thenpart: mov r2,#1
(gdb) stepi
20 ldr r3, =y @load the memory addres of y into r3
(gdb) stepi
21 ldr r2, [r3] @ load r2 register value into y memory address
(gdb) stepi
endofif () at fourth.s:23
23 mov r7, #1 @ program termination: exit syscall
(gdb) stepi
24 svc #0 @program termination: wake kernel
(gdb) info registers
r0 0x0 0



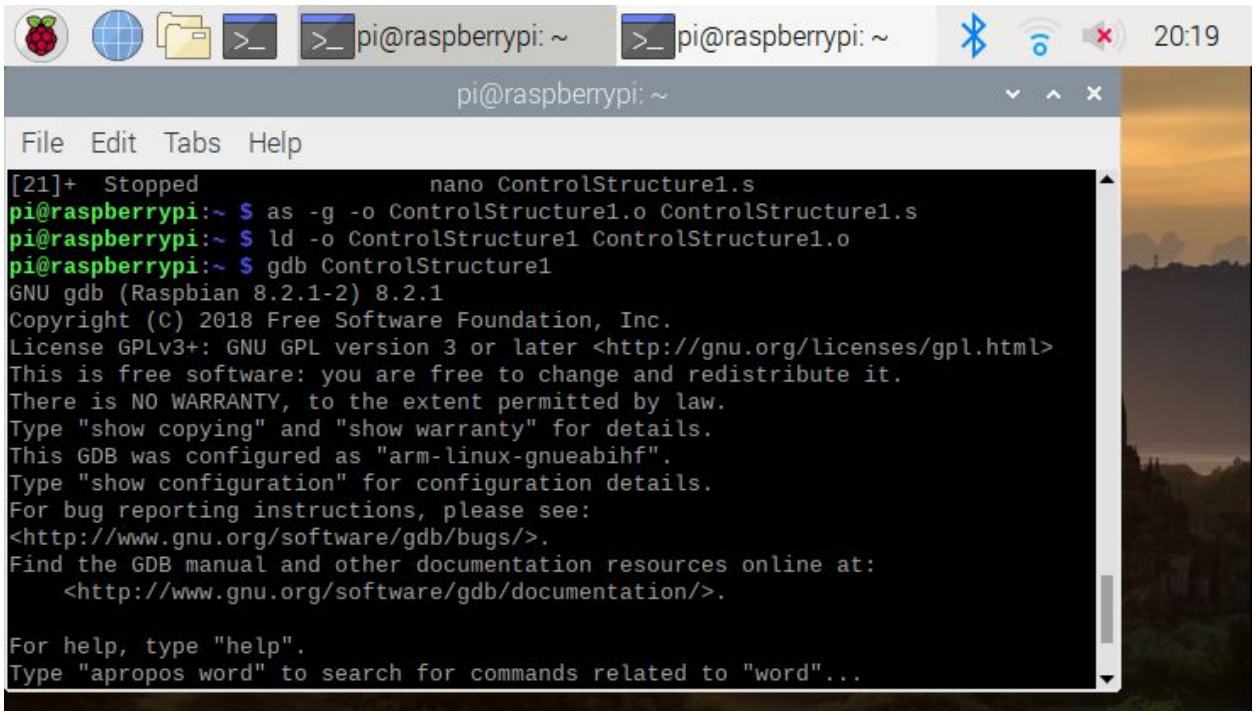


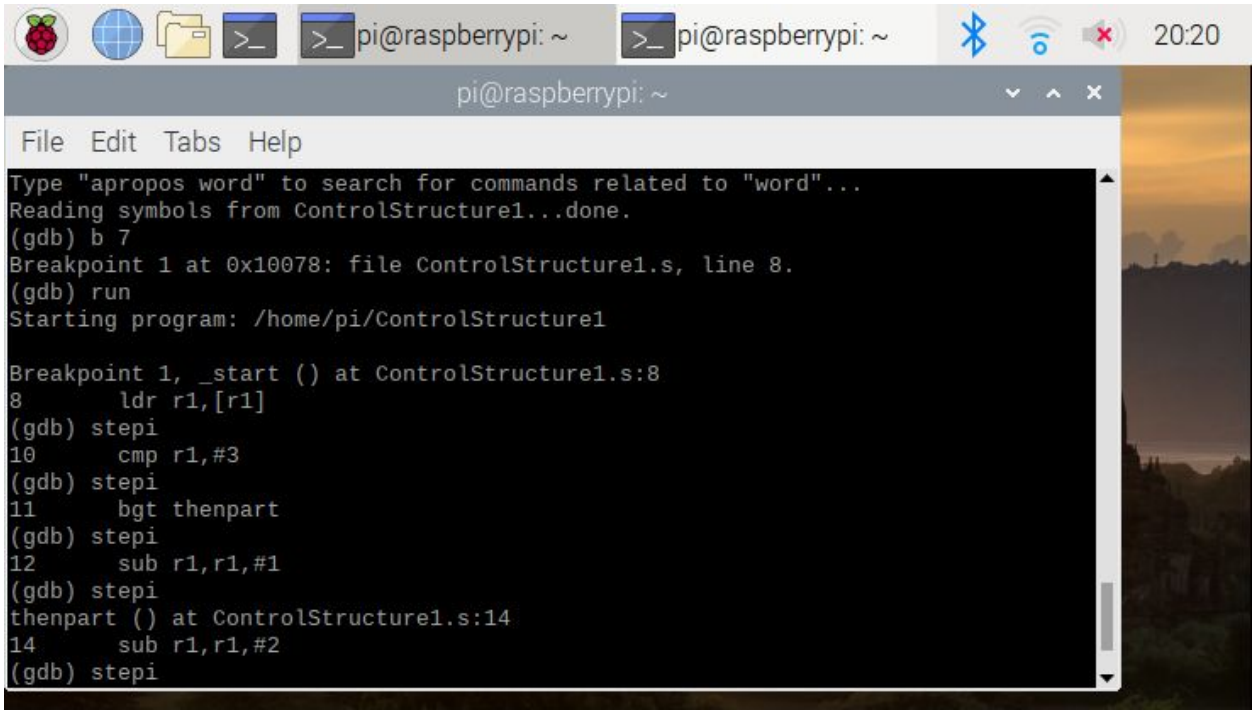
These are the registers at the end of the program. As you can see the value in r3 is 200a4 now, which is different from before I made the change. The Z flag for this is also 0 because the value is not 0.



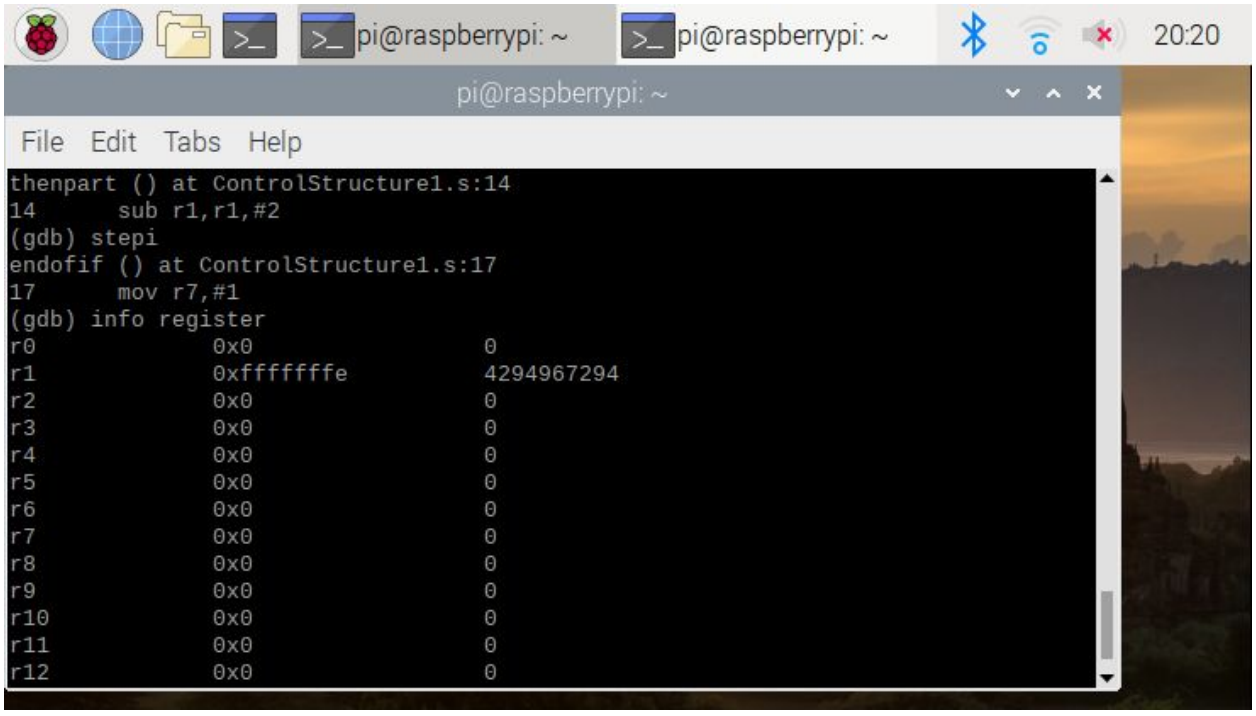
The change that was made for part 2. I changed beq to bne and then commented out the b instruction. This makes the program more efficient because it is using De Morgan's Law.

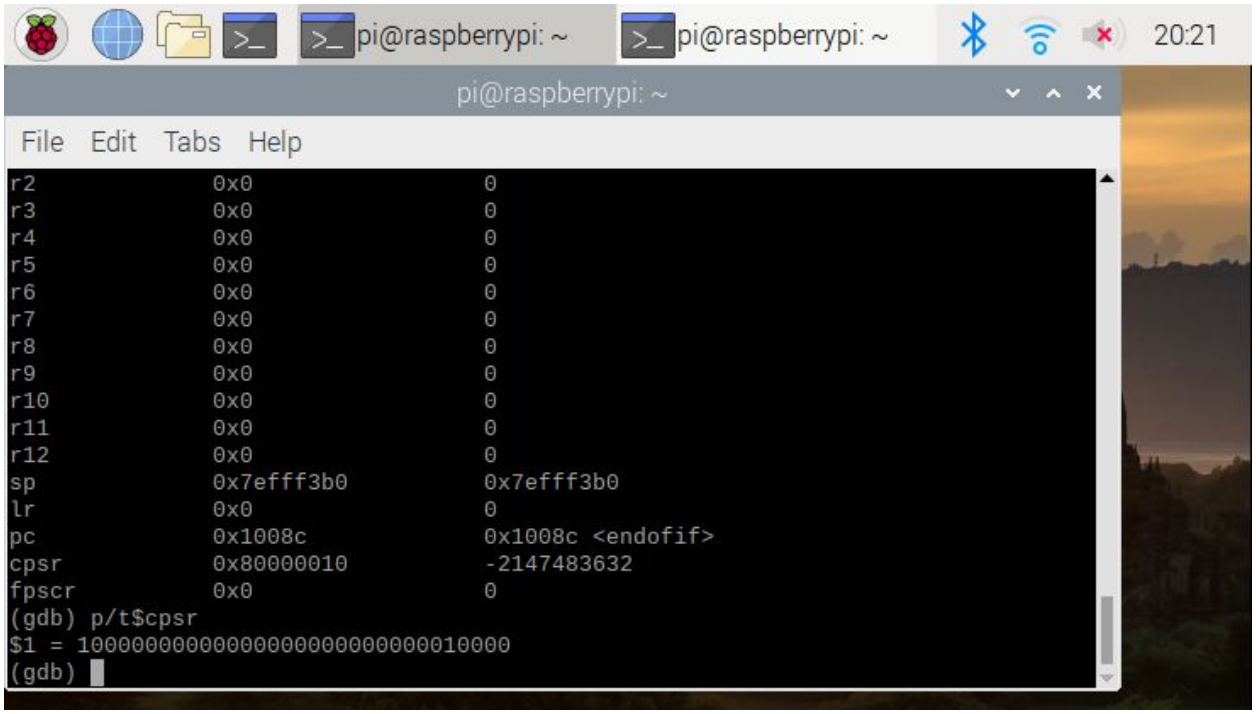
Part 3



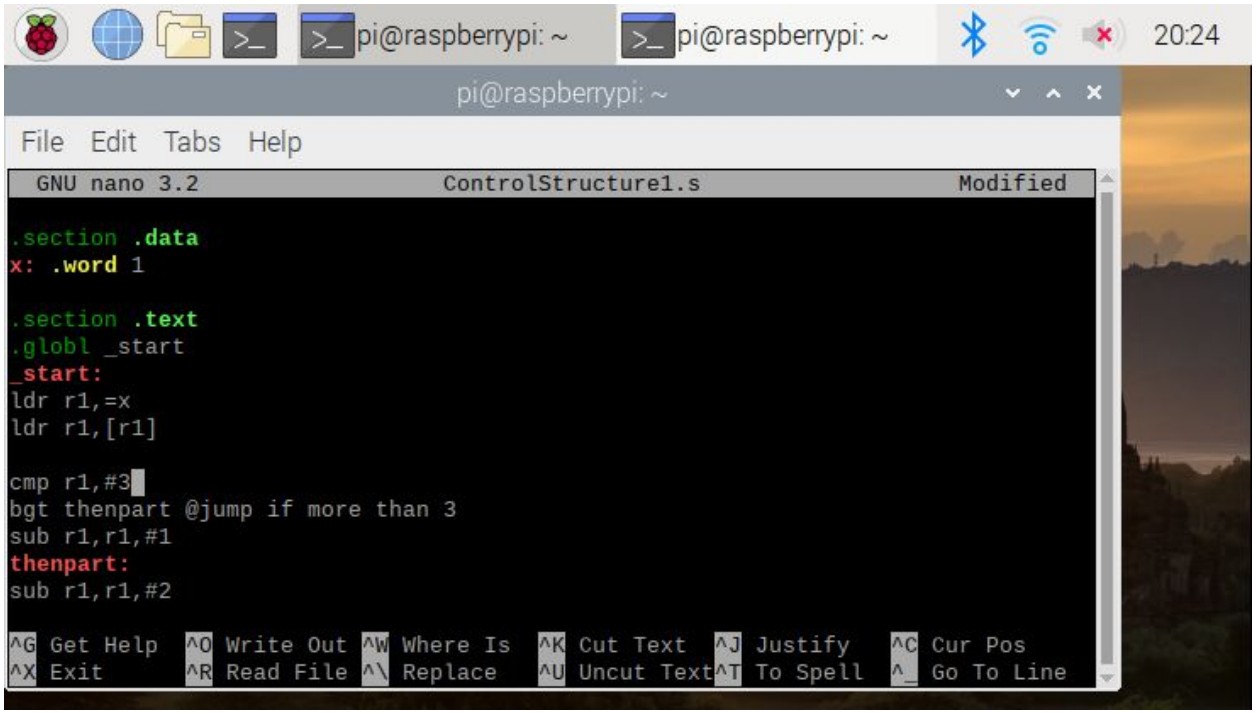


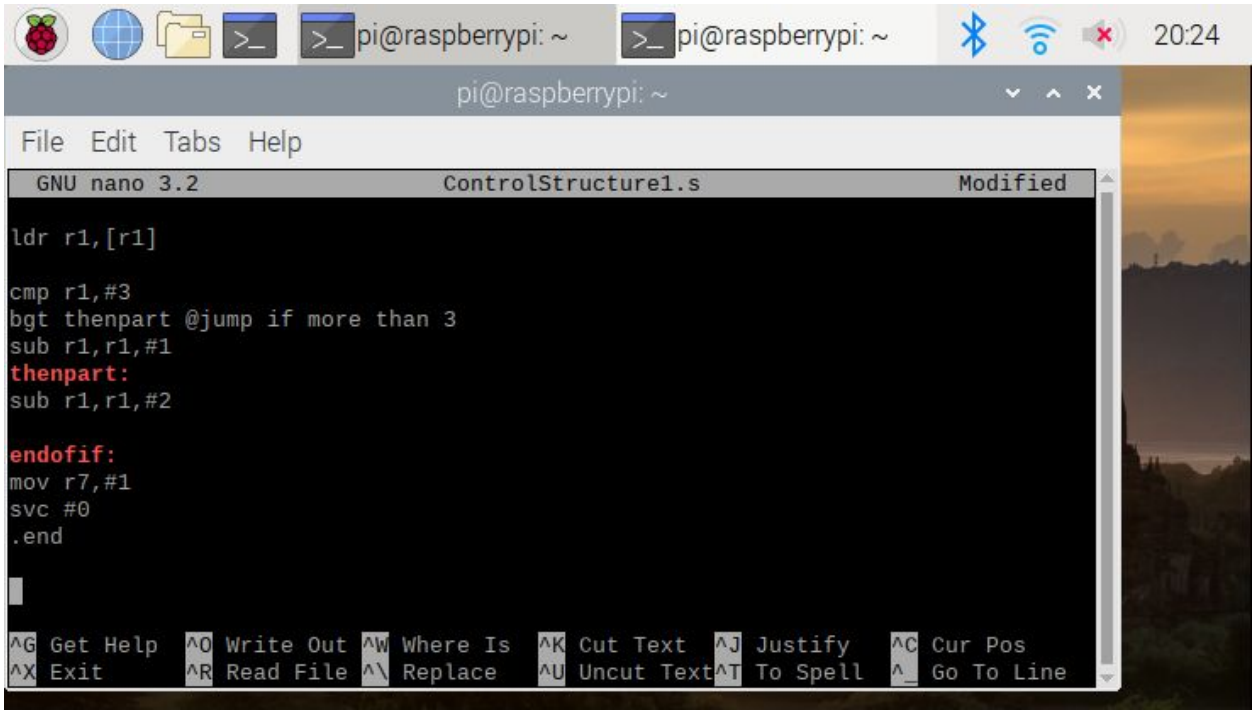
In these screenshots I assembled and linked the program and then placed my breakpoints and started to step into each line.





In these screenshots I displayed the registers and the values in the registers. The value in r1 is fffffffe.The z flag for this is 0.





This is my code for ControlStructure1.s. In this code you can see that I used De Morgan's law to jump to thenpart if X is greater than 3 and if it does not jump it will just subtract 1.

Parallel Programming Skills Foundation: Joan Galicia

- ➔ Race condition:
  - ◆ What is the race condition?
    - The behavior of an electronics, software, or other system where the output is dependent on the sequence or timing of other uncontrollable events. Occurs usually in logic circuits and become a bug when events do not happen according to how the programmer wanted them to.
  - ◆ Why is race condition difficult to reproduce and debug?
    - The end result is non deterministic and depends on the relative timing between between interfering threads
  - ◆ How can it be fixed? Provide an example from your Project_A3 (see spmd2.c)
    - To fix race conditions a solution is to initialize variable within a function
    - From the spmd2.c program done back in project 2. The race condition found was when the variables were initialized outside of the #pragma function. So I had to initialize the variables inside the #pragma function so that the variable could obtain the different thread numbers.

- ➔ Summaries the Parallel Programming Patterns section in the "Introduction to Parallel Computing_3.pdf" (two pages) in your own words (one paragraph, no more than 150 words).
  - ◆ Parallel programming is the use of multiple processors that help t o solve tasks simultaneously, and it is not bound strictly to one use only. Parallel programming has an abundance of patterns and helps identify which and how to use them. The article conveys a method of categorizing the patterns one is the algorithmic strategies and the other is how the hardware is used to process tasks simultaneously, which is known as concurrent execution mechanisms. Algorithmic strategies are just a way of deciding how the task in the parallel programming will be executed concurrently.
- ➔ In the section "Categorizing Patterns" in the "Introduction to Parallel Computing_3.pdf" compare the following:
  - ◆ Collective synchronization (barrier) with Collective communication (reduction)
    - Collective Synchronisation blocks all the processes until a specific synchronisation point is reached. It acts as a barrier as it blocks the processes until all other processes reach the synchronisation point. Collective communication uses all of its processes to reach a specific point before executing. It acts as a reduction since one process of the communicator collects data from all other processes and performs an operation to find the result.
  - ◆ Master worker with fork join
    - The master-worker pattern divides the main process into small chunks and distributes it to other worker processes. The Fork-Join pattern is used to execute parallel light weight processes and threads.
- ➔ Dependency: Using your own words and explanation, answer the following:
  - ◆ Where can we find parallelism in programming?
    - Parallelism enables programs to run faster by performing multiple computations at the same time by integrating hardware with multiple CPUs. This can be found in mobile phones, databases, and servers.
  - ◆ What is dependency and what are its types (provide one example for each)?
    - Dependency is an operations reliance of another operation to be completed for its other operations to experience the same thing.
  - ◆ When a statement is dependent and when it is independent (Provide two examples)?
    - A statement is dependent if given below:
      - S1: a =1, s2 b =a +5
        - ◆ This is dependent because for statement 2 to execute it requires statement 1 to assign a value.

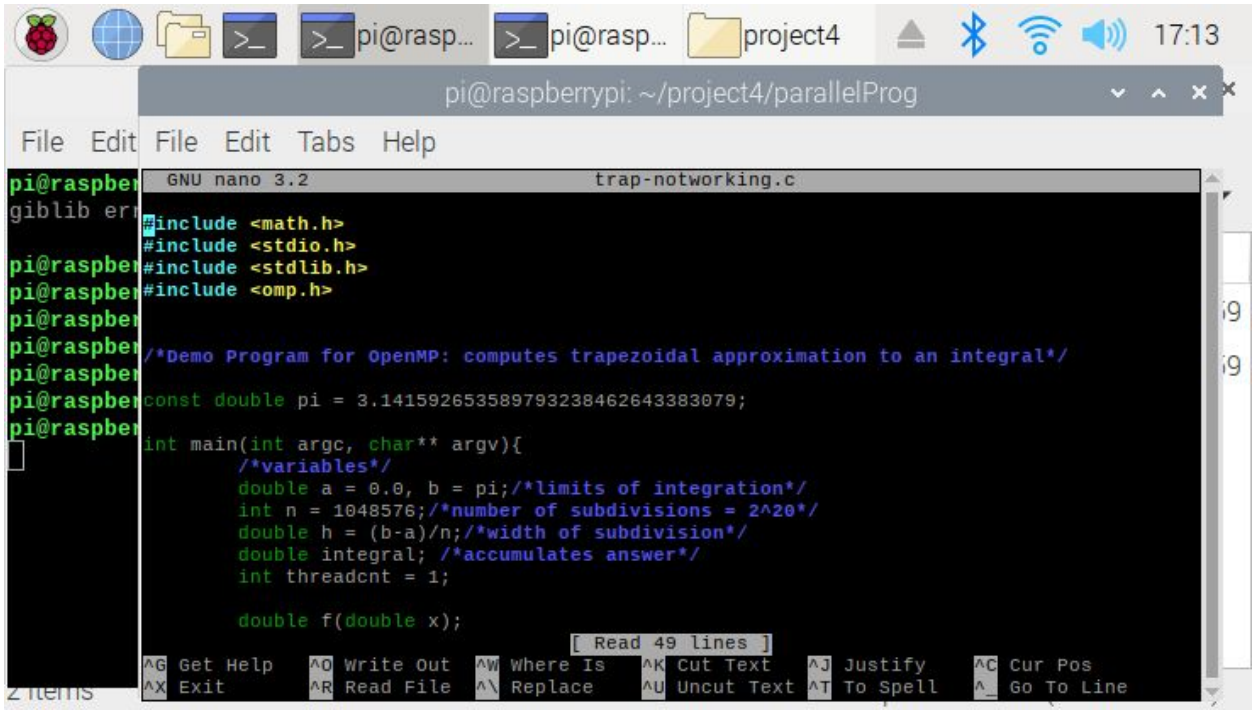
- A statement is dependent which is given below:
    - S1: a =20, s2: b = 21
        - This is independent because both statements are being assigned a value but are uncorrelated with each other.

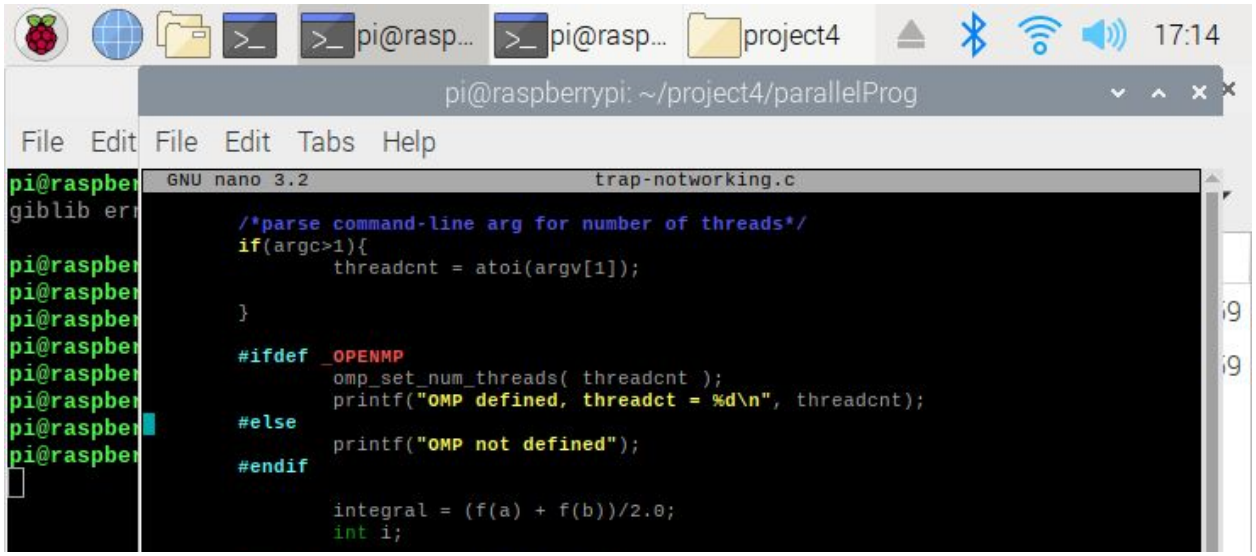
- When can two statements be executed in parallel?
    - Two statements can be executed in parallel when there are no dependencies involved
- How can dependency be removed?
    - To remove dependencies, the program must be modified by either rearranging statements or eliminating statements.
- How do we compute dependency for the following two loops and what type/s of dependency?
    - To compute dependencies we must find the IN and OUT set of each node.
    - For this case both for loops have flow dependencies

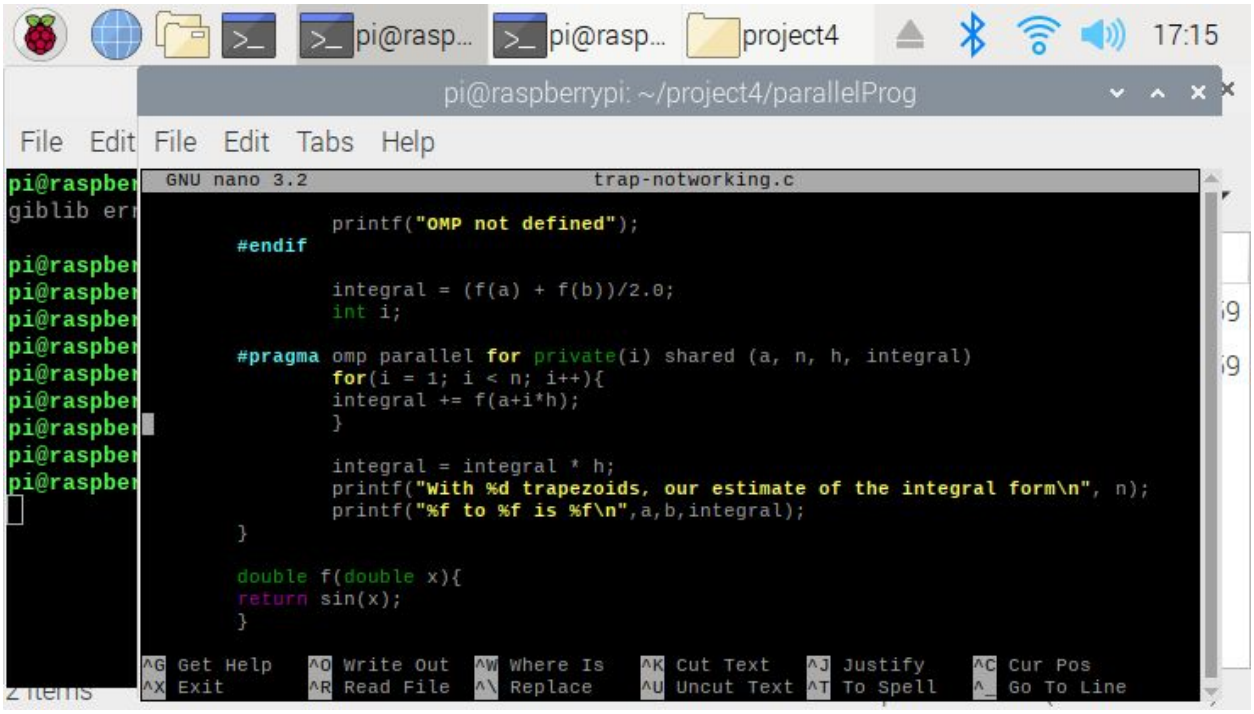
```
for(i=0;i<100;i++)          for(i=0;i<100;i++){
  S1: a[i]=i;                 S1:a[i]=i;
                              S2:b[i]=2*i;
                             }
```

Parallel Programming Basics:

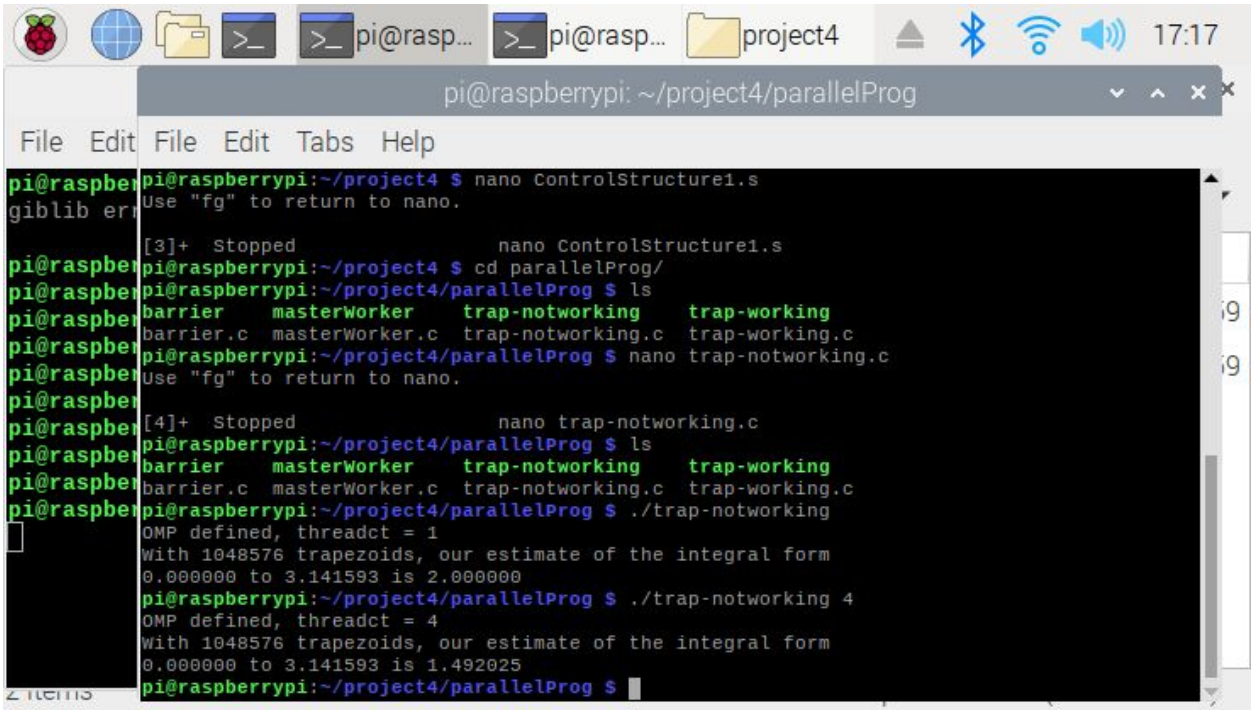
Part 1


pi@rasp...
pi@rasp...
project4
17:13
pi@raspberrypi: ~/project4/parallelProg
File Edit File Edit Tabs Help
GNU nano 3.2 trap-notworking.c
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
/*Demo Program for OpenMP: computes trapezoidal approximation to an integral*/
const double pi = 3.14159265358979323846264338307 9;
int main(int argc, char** argv){
/*variables*/
double a = 0.0, b = pi;/*limits of integration*/
int n = 1048576;/*number of subdivisions = 2^20*/
double h = (b-a)/n;/*width of subdivision*/
double integral; /*accumulates answer*/
int threadcnt = 1;
double f(double x);
[Read 49 lines]
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line


pi@rasp...
pi@rasp...
project4
17:14
pi@raspberrypi: ~/project4/parallelProg
File Edit File Edit Tabs Help
GNU nano 3.2 trap-notworking.c
/*parse command-line arg for number of threads*/
if(argc>1){
threadcnt = atoi(argv[1]);
}
#ifdef _OPENMP
omp_set_num_threads(threadcnt);
printf("OMP defined, threadct = %d\n", threadcnt);
#else
printf("OMP not defined");
#endif
integral = (f(a) + f(b))/2.0;
int i;

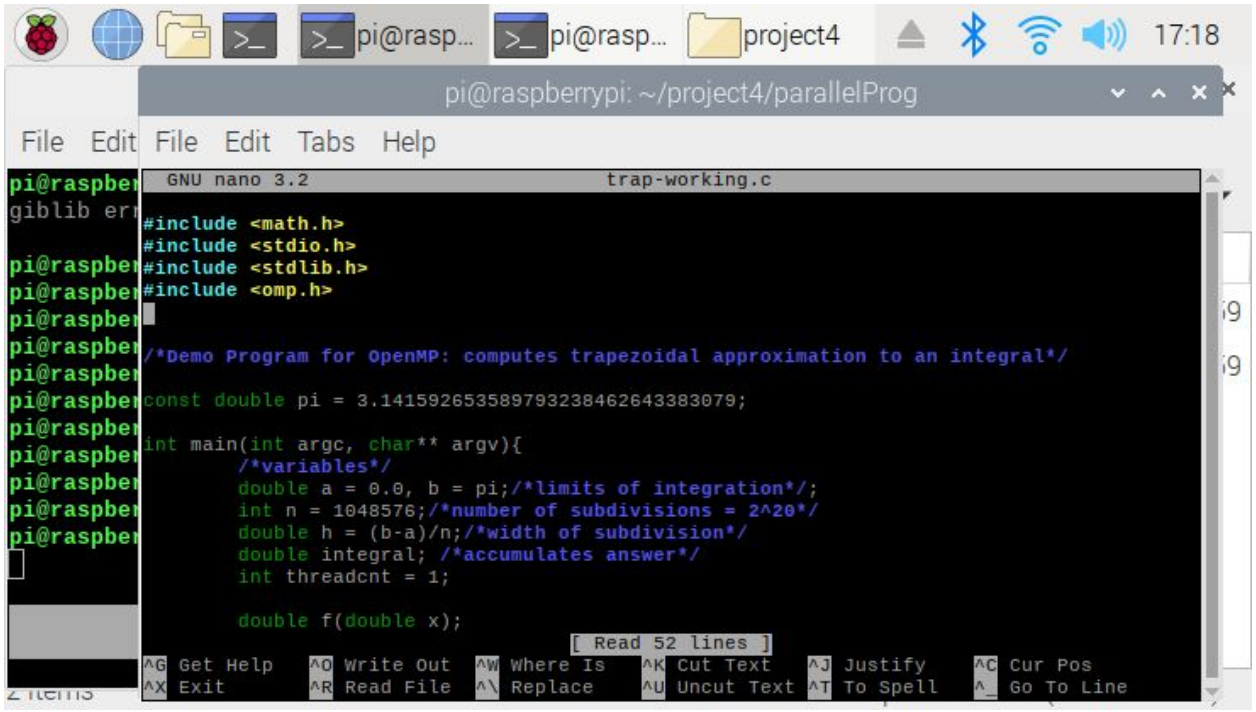


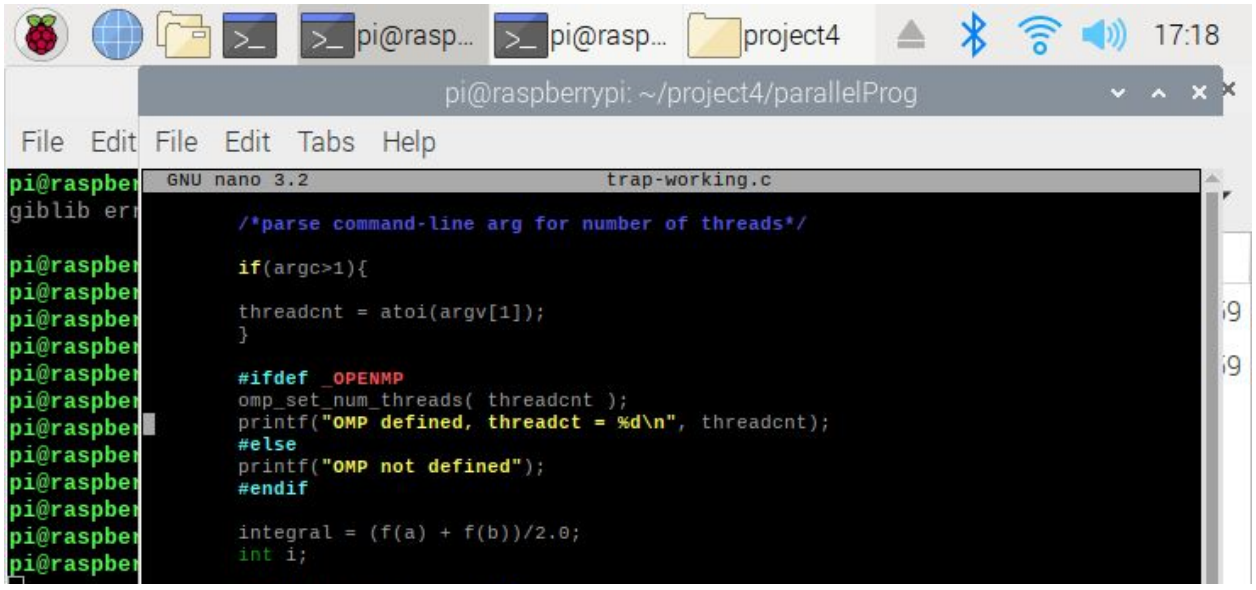
All three of these screenshots above are of the program “trap-notworking” and this is the first program where I encounter the use of calculus. In this program, the threads will contain a copy of each variable in memory. The issue I found when compiling this program using gcc is that it would not recognize sin(x). This is because i was not using the correct library at the time, the solution was to add “-lm” to the end of what I was compiling.

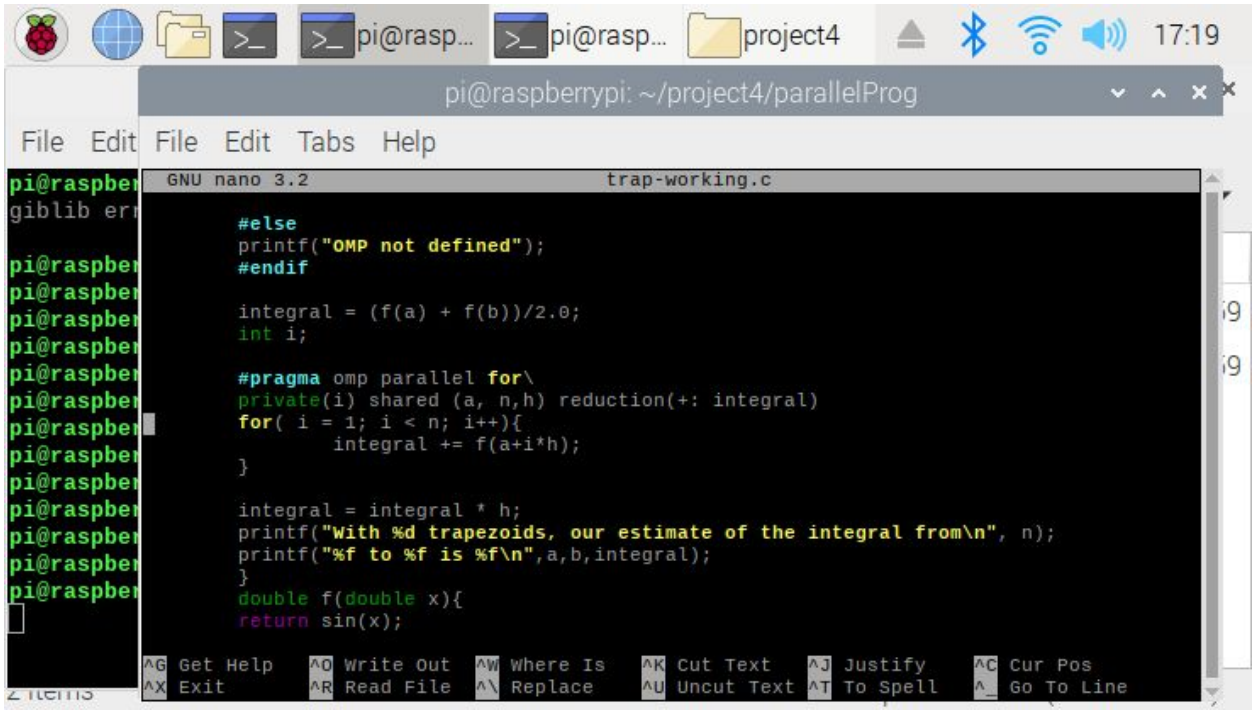


This screenshot is the executed program “trap-not working” the answer is supposed to be 2.0 when executed as “./trap-notworking 4”. This is not the case as the output is given as 1.492025 so this has something to do with the single line #pragma.

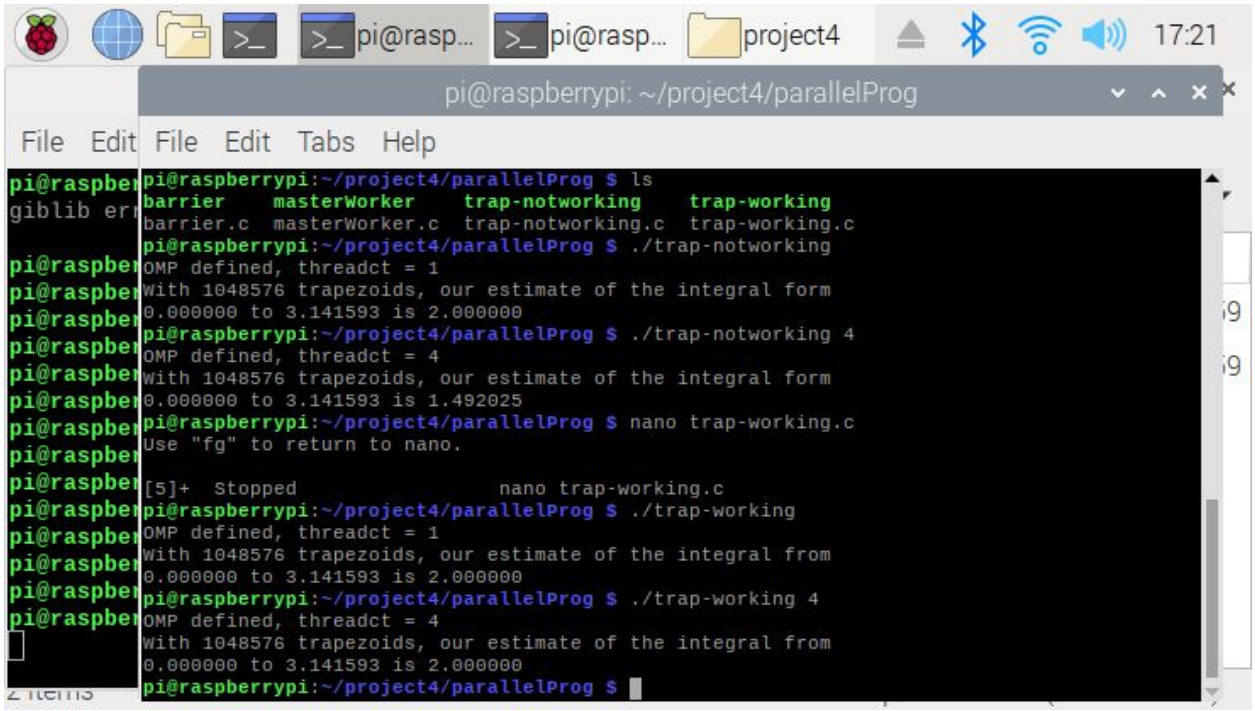
Part 2


pi@rasp...
pi@rasp...
project4
17:18
pi@raspberrypi: ~/project4/parallelProg
File Edit File Edit Tabs Help
GNU nano 3.2 trap-working.c
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
/*Demo Program for OpenMP: computes trapezoidal approximation to an integral*/
const double pi = 3.14159265358979323846264338307 9;
int main(int argc, char** argv){
/*variables*/
double a = 0.0, b = pi;/*limits of integration*/;
int n = 1048576;/*number of subdivisions = 2^20*/
double h = (b-a)/n;/*width of subdivision*/
double integral; /*accumulates answer*/
int threadcnt = 1;
double f(double x);
[Read 52 lines]
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line


pi@rasp...
pi@rasp...
project4
17:18
pi@raspberrypi: ~/project4/parallelProg
File Edit File Edit Tabs Help
GNU nano 3.2 trap-working.c
/*parse command-line arg for number of threads*/
if(argc>1){
threadcnt = atoi(argv[1]);
}
#ifdef _OPENMP
omp_set_num_threads(threadcnt);
printf("OMP defined, threadct = %d\n", threadcnt);
#else
printf("OMP not defined");
#endif
integral = (f(a) + f(b))/2.0;
int i;

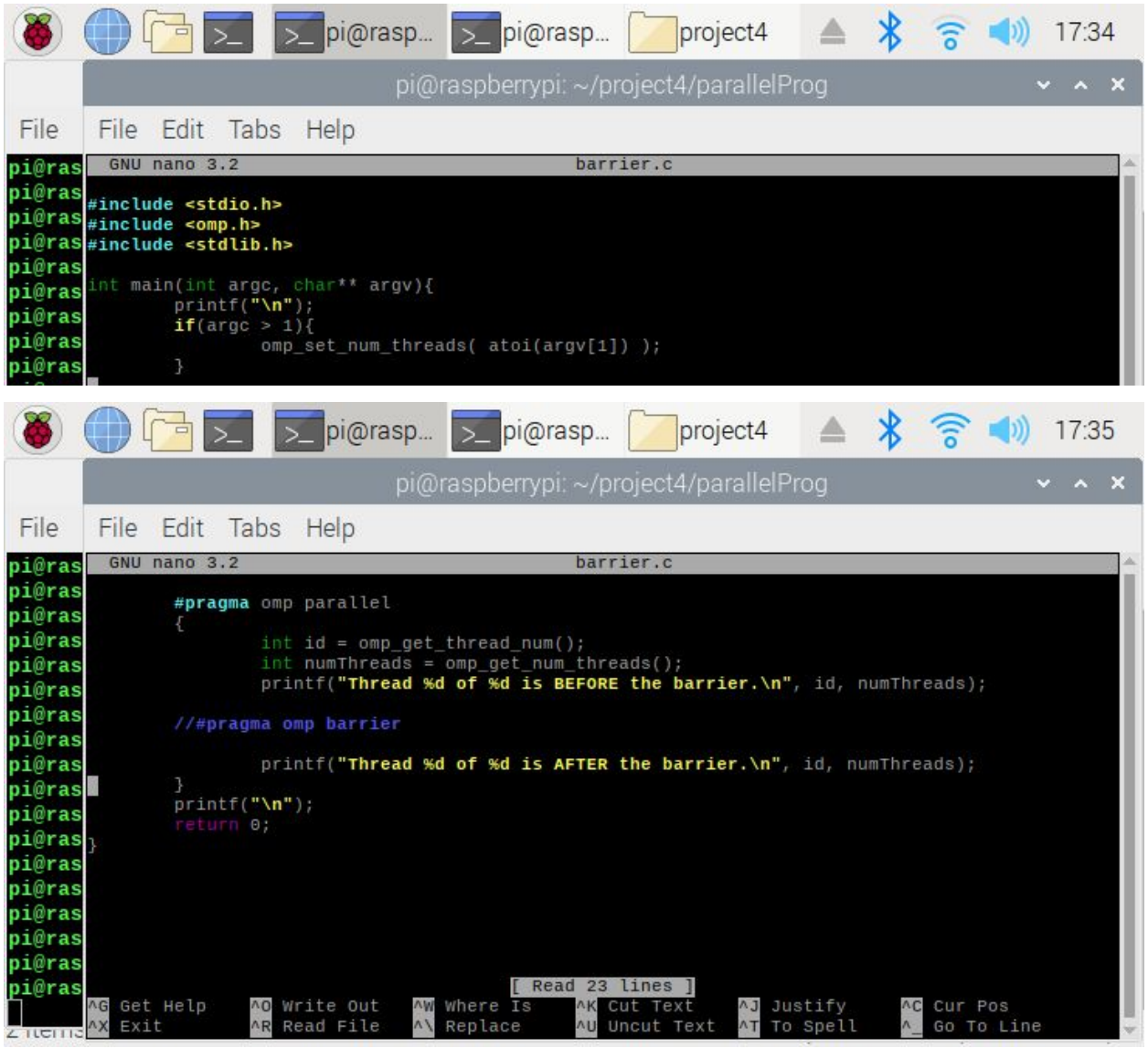


These three screenshots are of the program “trap-working” this is the corrected version for the integration of the trapezoidal rule program. The error found in this is in line 37 or in this case where the first “#pragma” is found. This corrected version allows for the CPU to know that we are setting the variables as private, and we are also using the reduction clause for the variable “integral”.

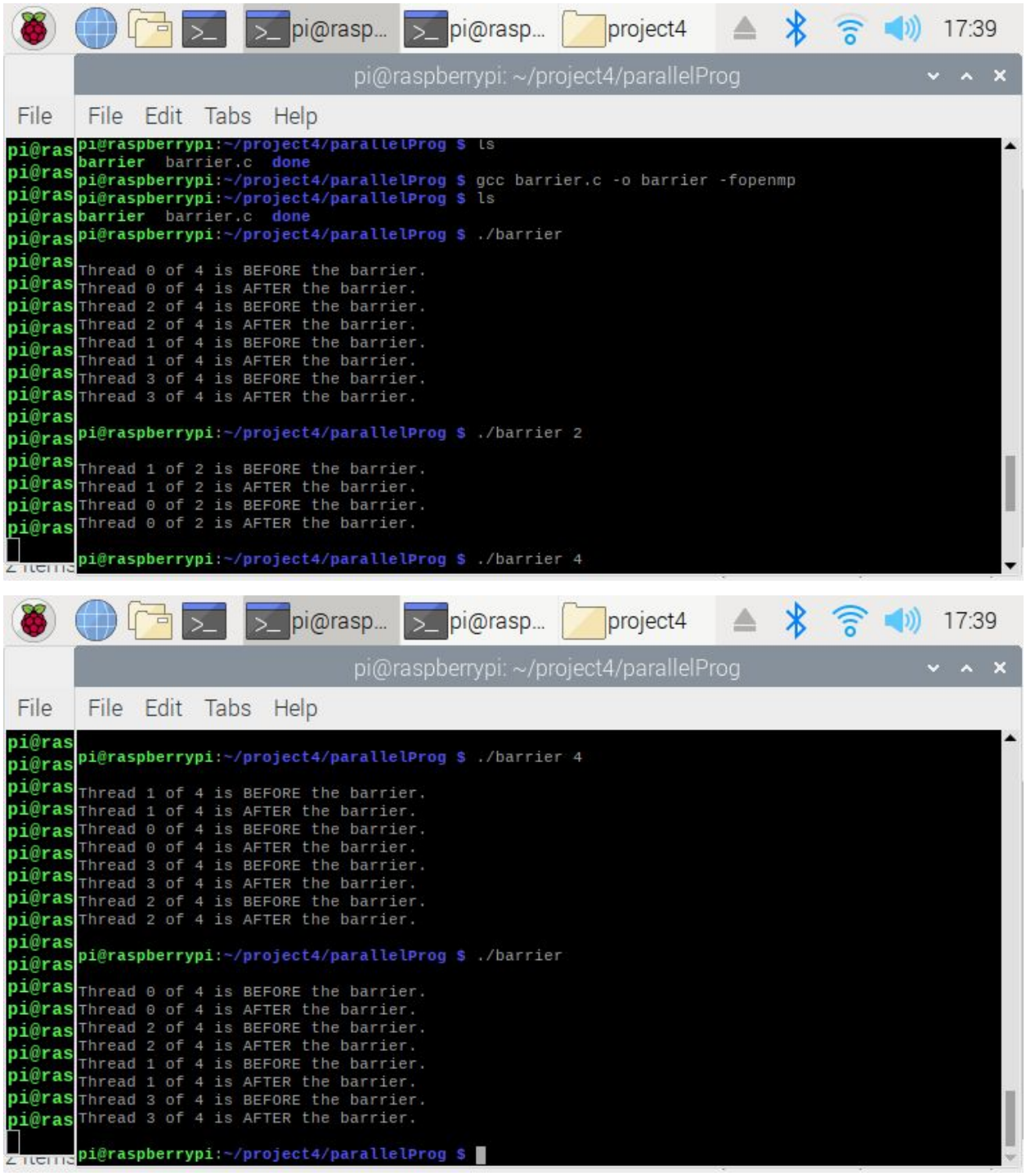


This screenshot is the correct output of the integration of the trapezoidal rule program. As we set the number of threads being used as 4 the answer to the integration of the trapezoidal rule should be 2.0.

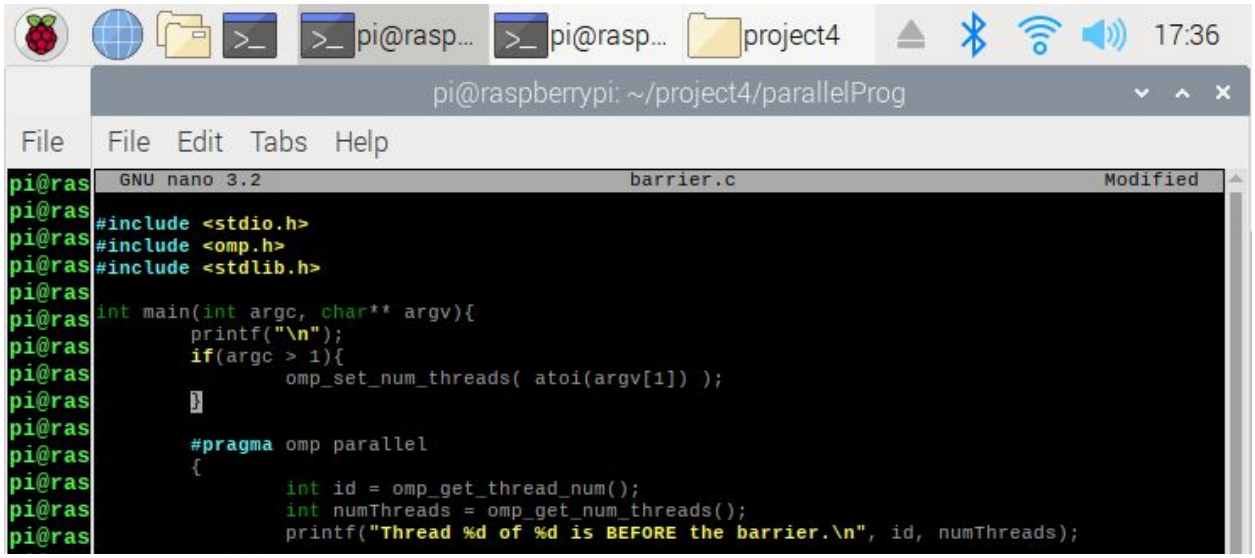
Part 3

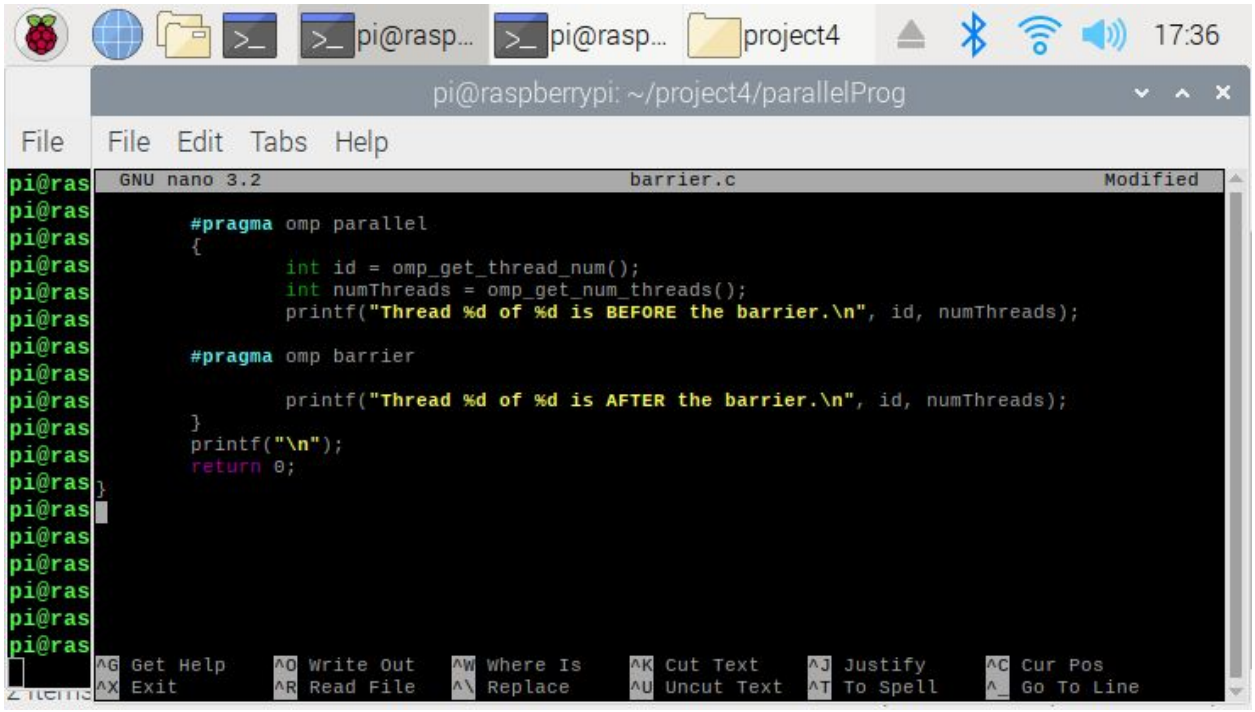


These two screenshots are of the program “barrier” which ensures that all threads complete a parallel section of code before execution continues on to the next task. This first version is the program where it does not use the barrier function so this means that the program will not function correctly.

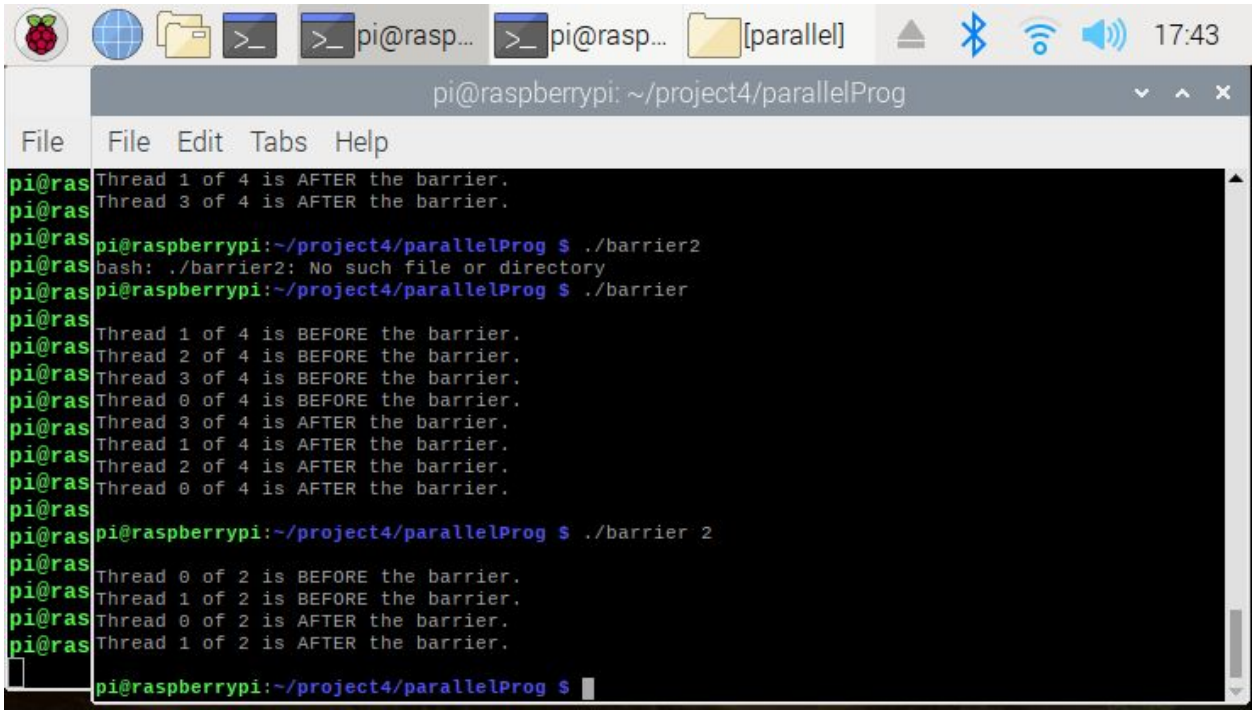


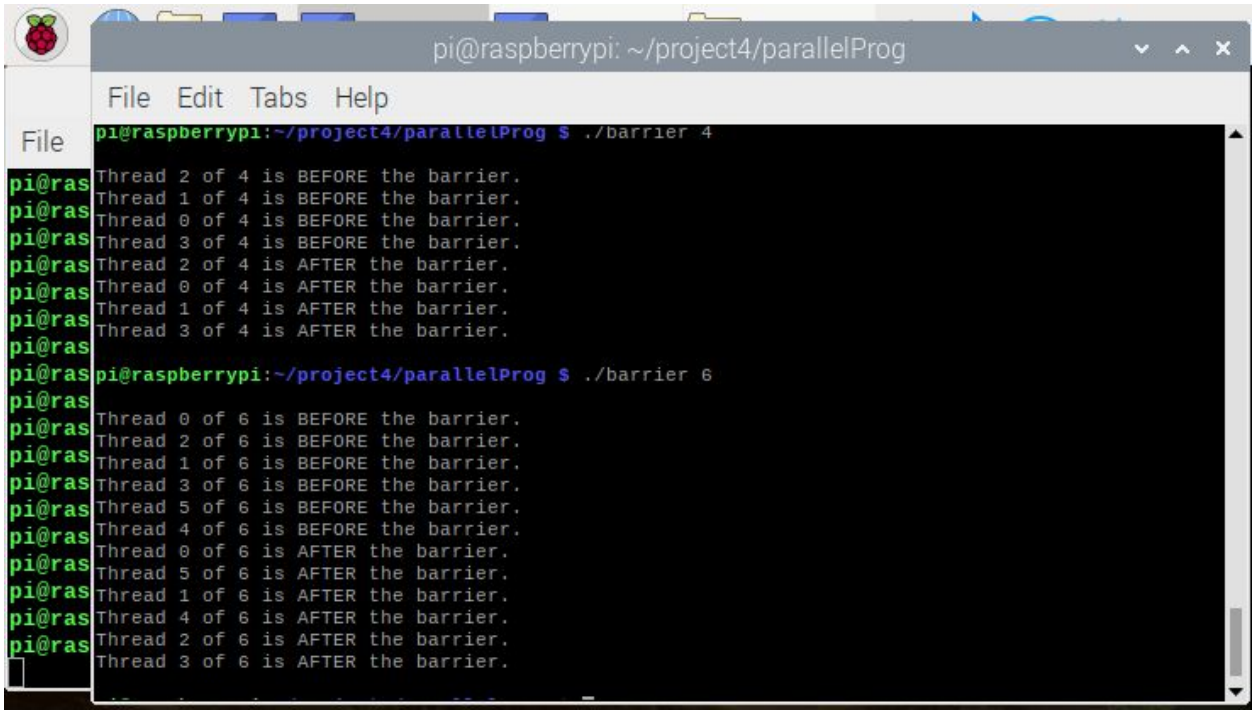
These two screenshots are of the executed program “barrier” with the commented barrier function. This shows the barrier is not working correctly as the before and after are displayed one after the other. The actual output is supposed to display all the before together and then the after as seen on the corrected version of the “barrier program”.





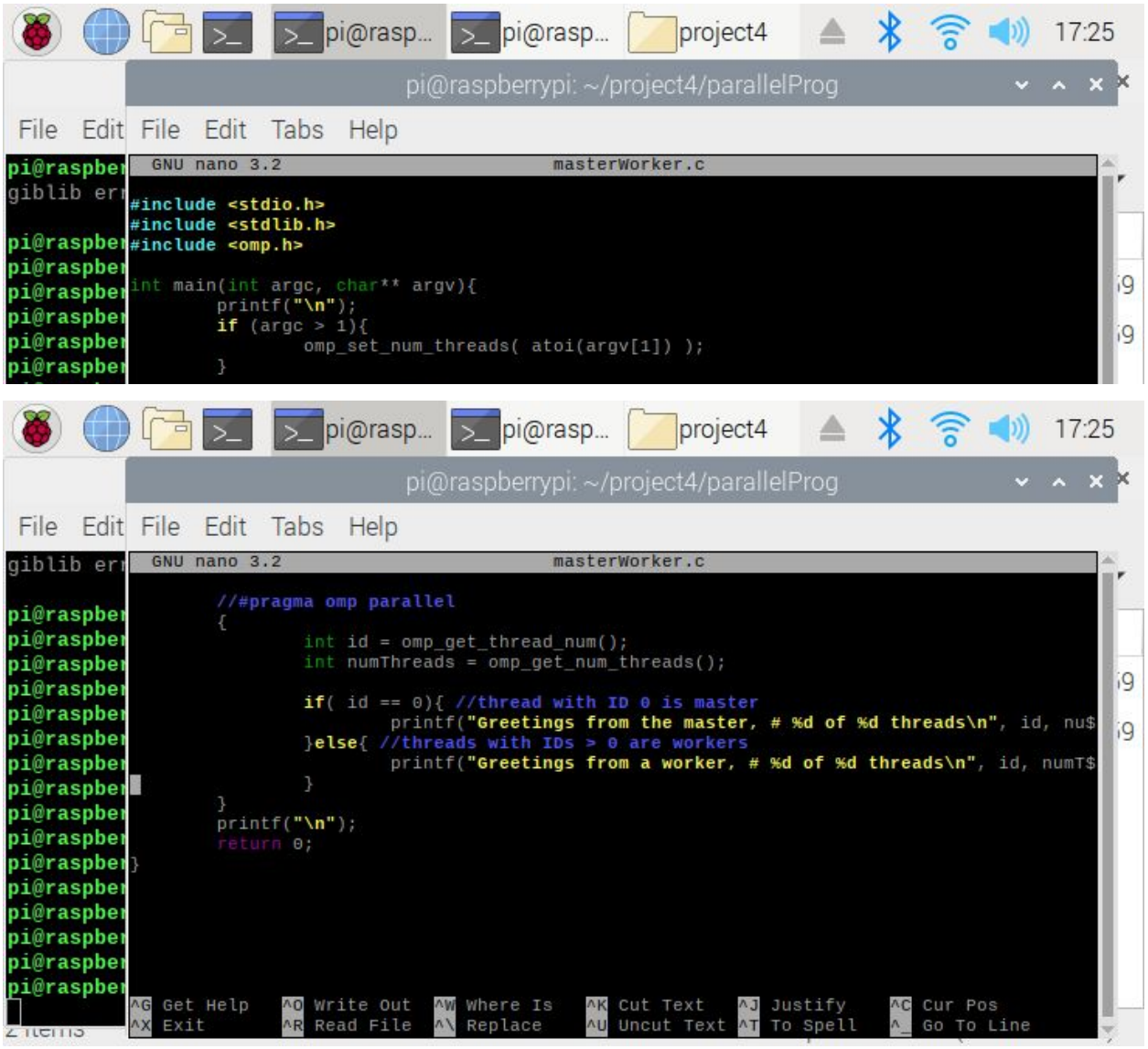
These two screenshots are of the corrected version of the “barrier ” program where the “#pragma omp barrier” function is uncommented. This allows for the computer to know that there will be a barrier for the after thread.



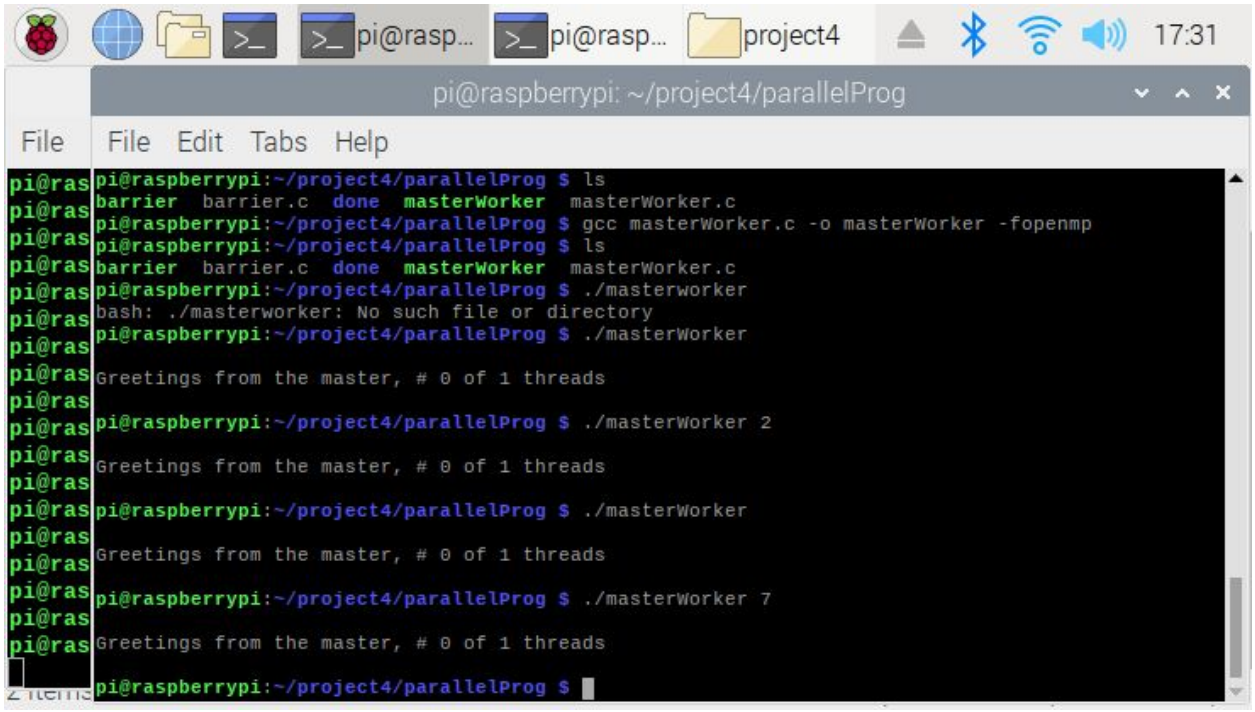


These two screenshots are of the executed program “barrier” without the commented barrier function. The output shows that the threads executed the first tasks first and then the second tasks. I also set a higher number of threads to show what it executed and it was the same when I assigned the set of threads to 4.

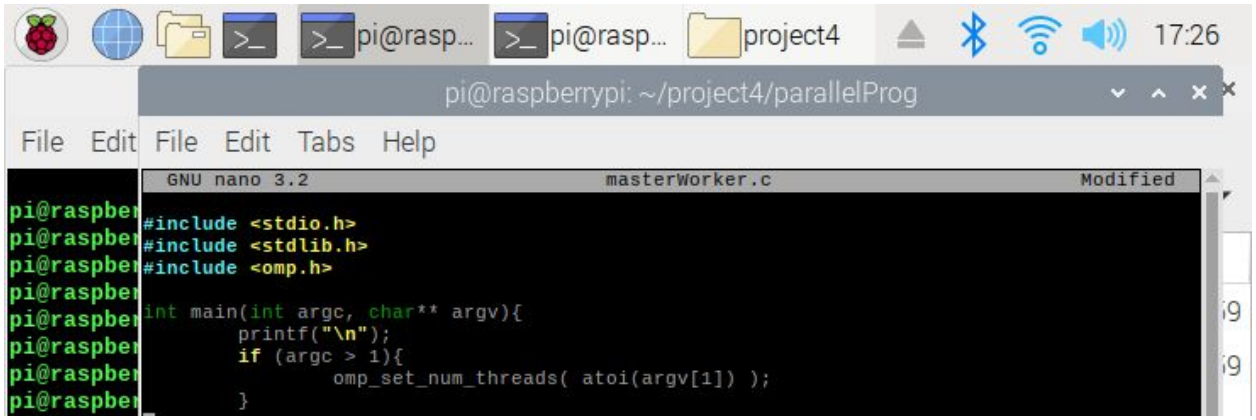
Part4

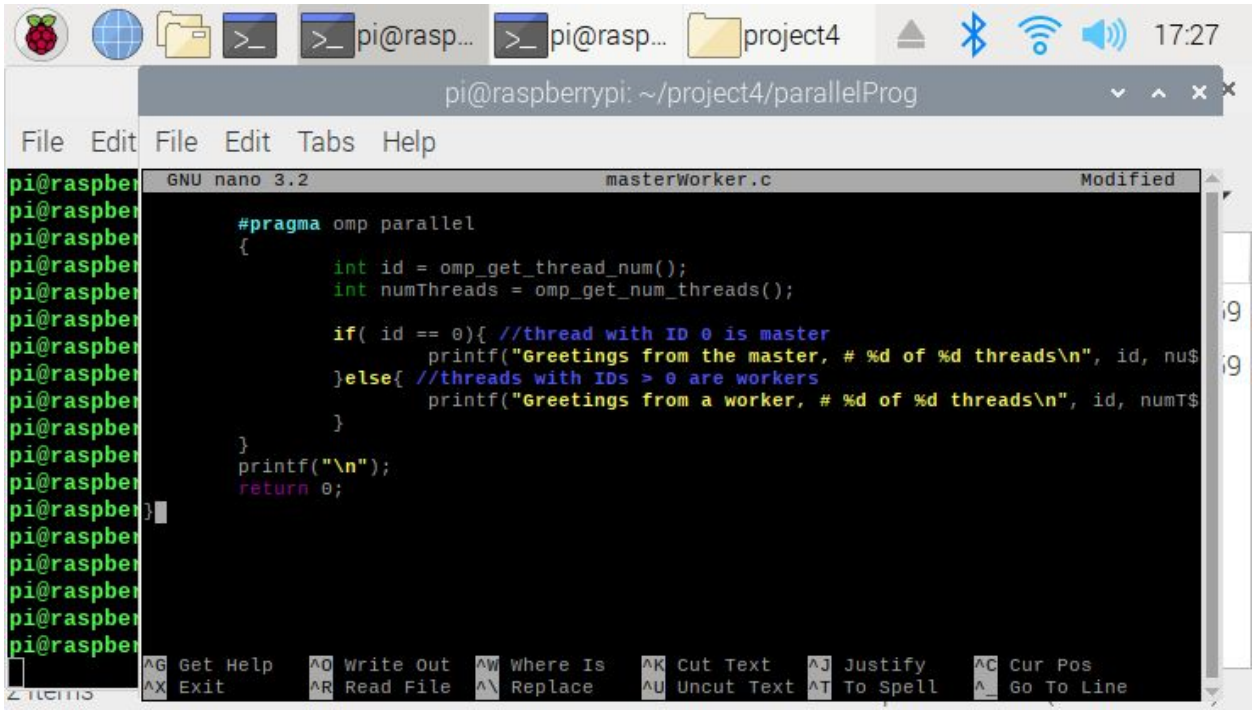


These two screenshots are of the program “masterworker” with the “#pragma” function commented. This program uses one thread that is called the master and it executes to one block of code, and then forks out. These forked threads are called workers which when they execute on a different block then they also fork. This version did not work correctly and the corrected version is shown below.

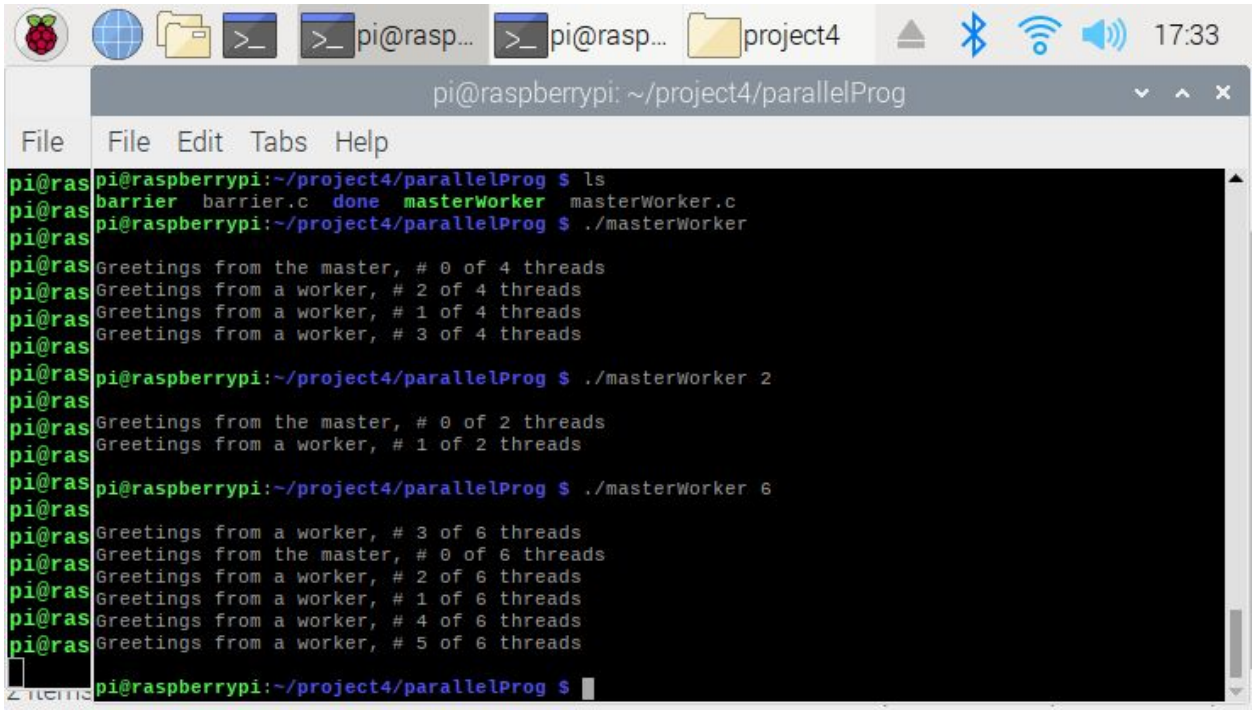


This screenshot is of the executed program “masterworker” where it just outputs the master thread, and none of the worker threads are displayed. Even with setting a higher number of threads the result did not change.





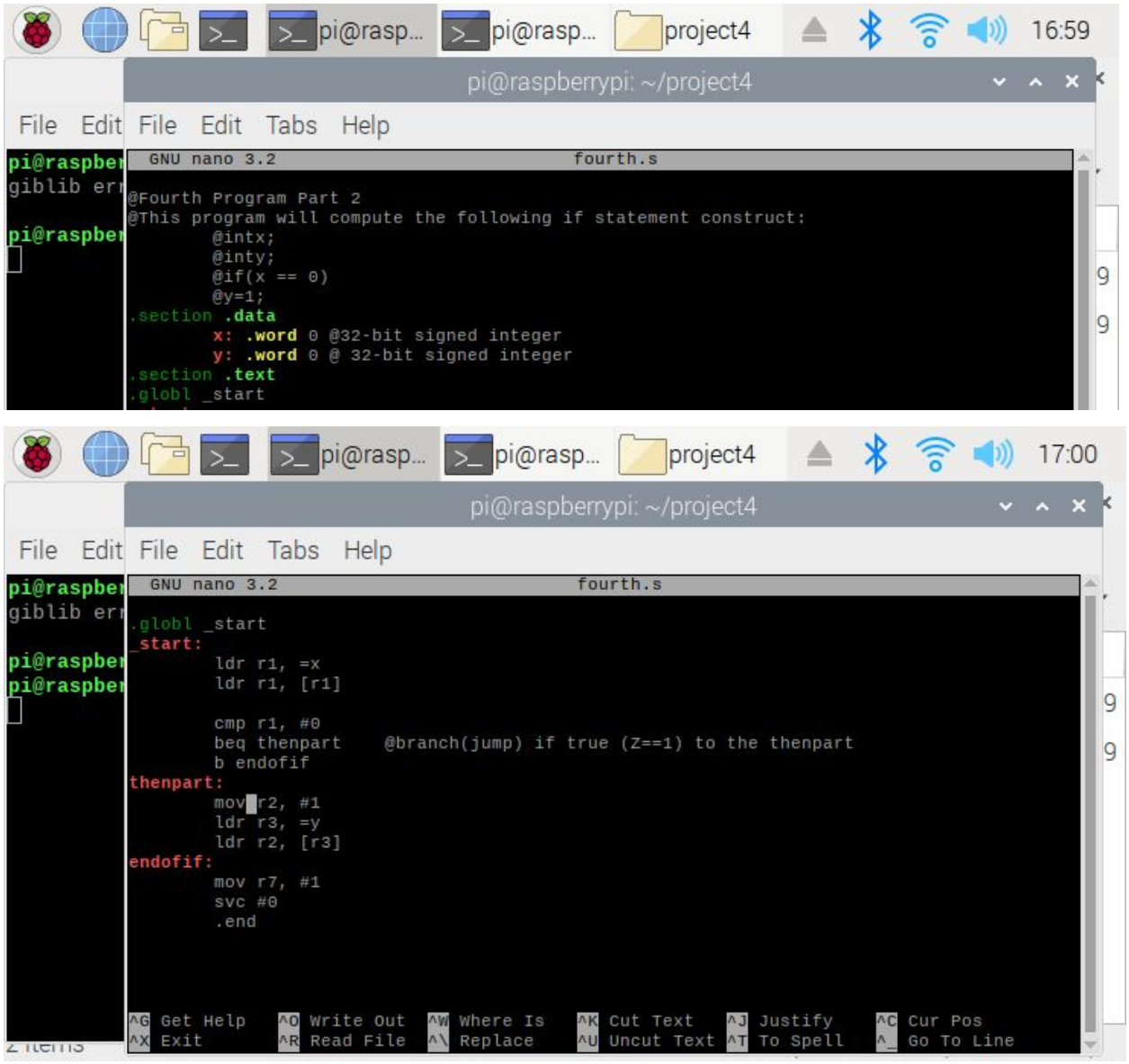
These two screenshots are of the program “masterworker” with the “#pragma” function uncommented.



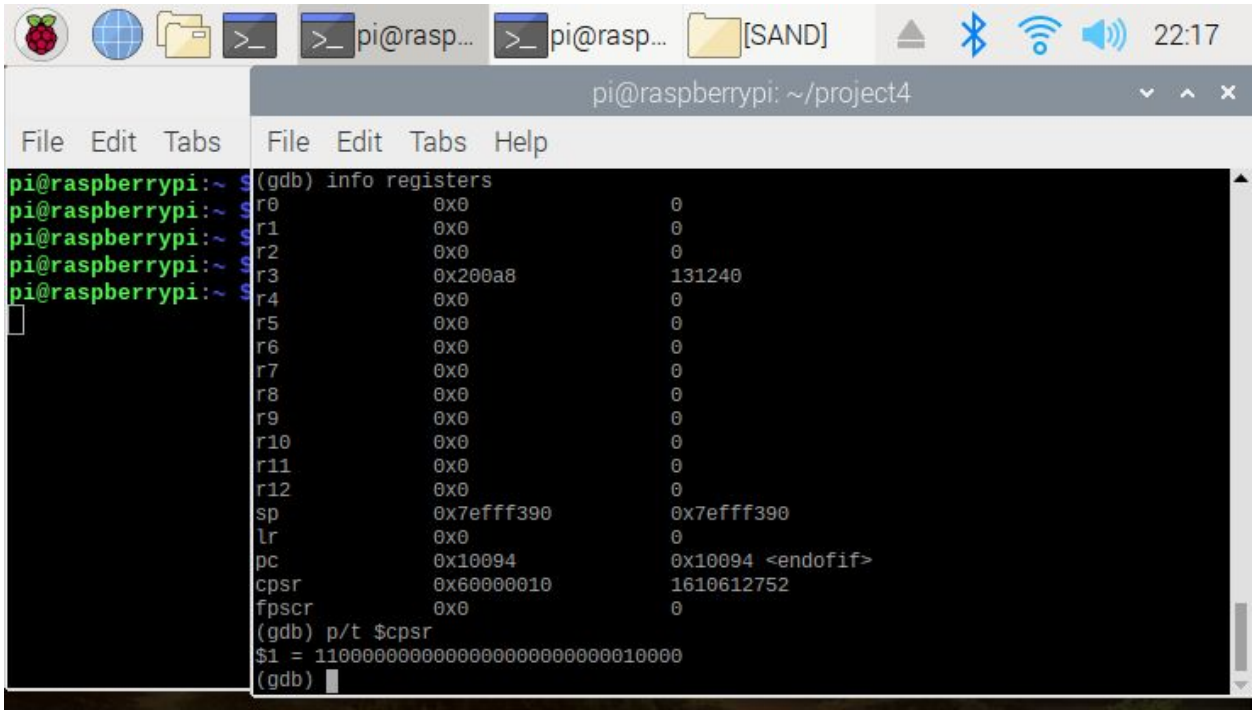
This screenshot is of the correct version of the executed program “masterworker”. As shown in the output it displays the worker and master thread. I have also executed the program to show what would happen if I assigned multiple threads.

ARM Assembly Programming:

Part 1

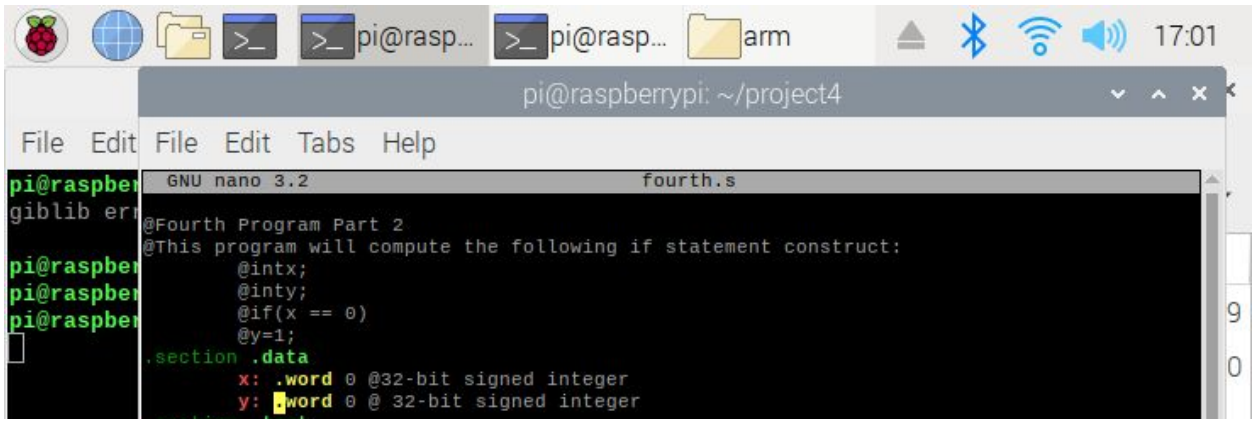


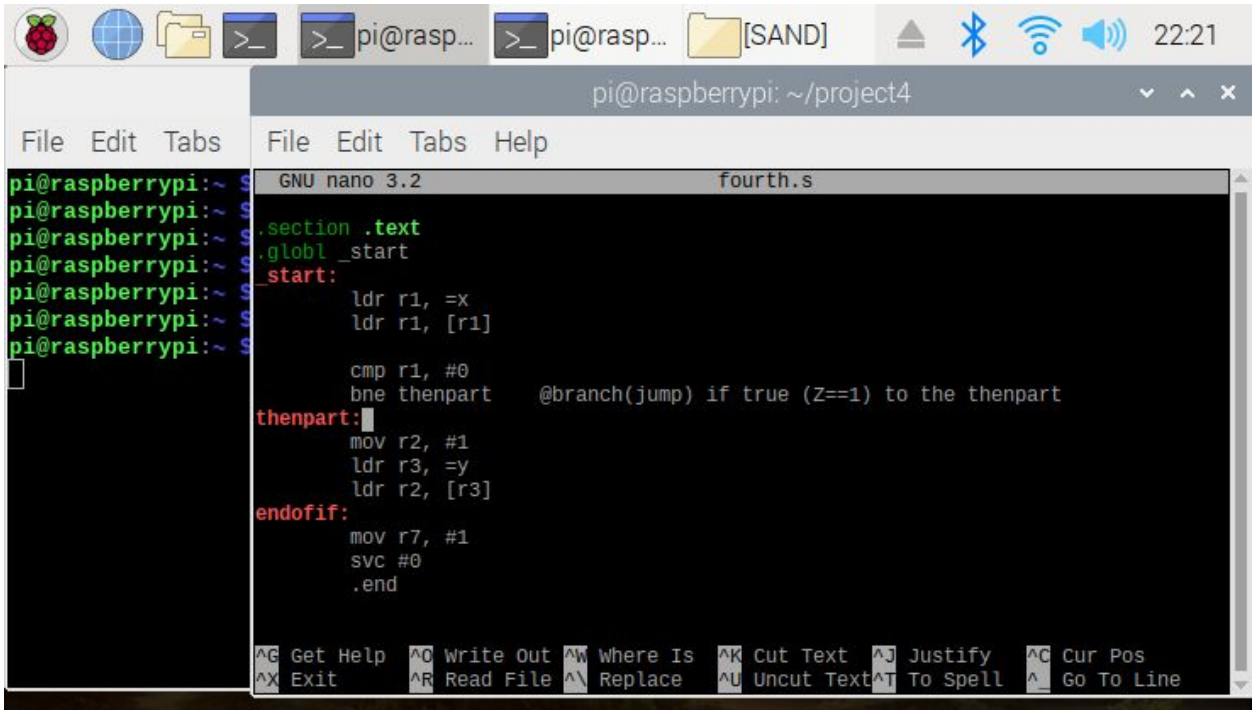
The two screenshots shown above are of the first version of the “fourth” program. There were no issues with this program as it just helped to identify how if statements are used in arm assembly language. The variable x is loaded to register 1 and is then compared to 0 to see if the zero flag is cleared or set.



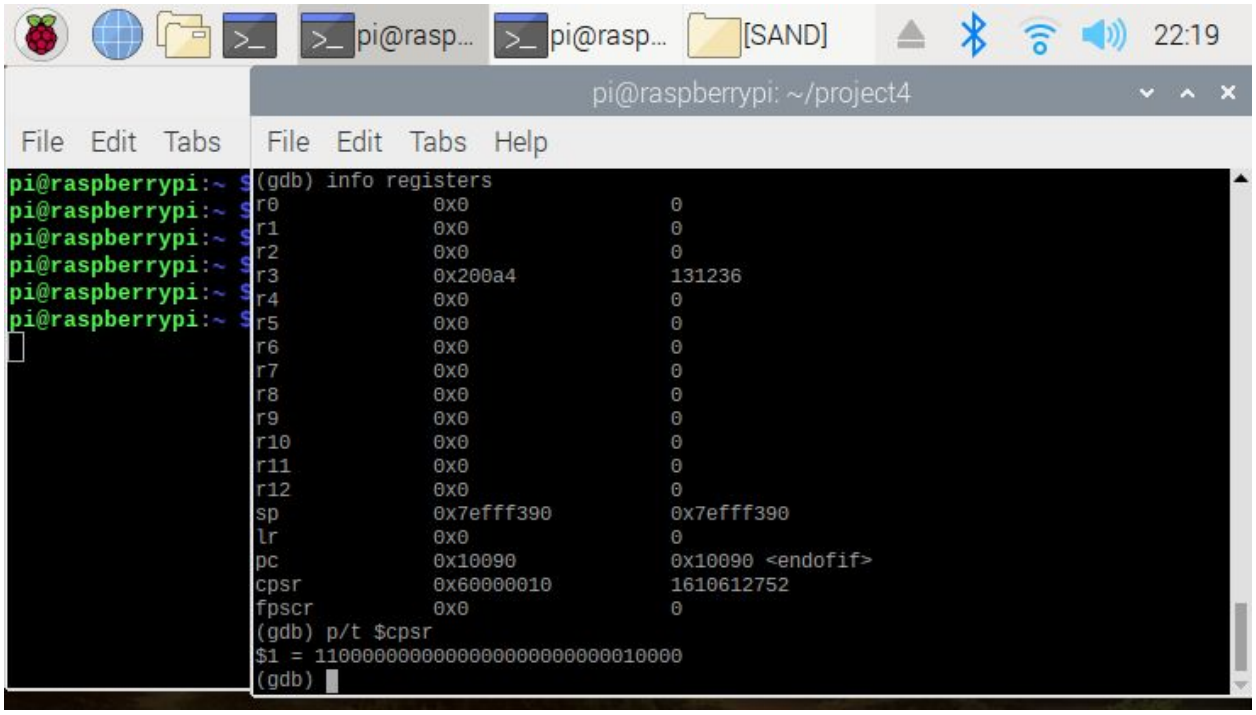
This screenshot is of the executed program "fourth" version 2 where it is a more inefficient version as it has back to back branches.

Part 2



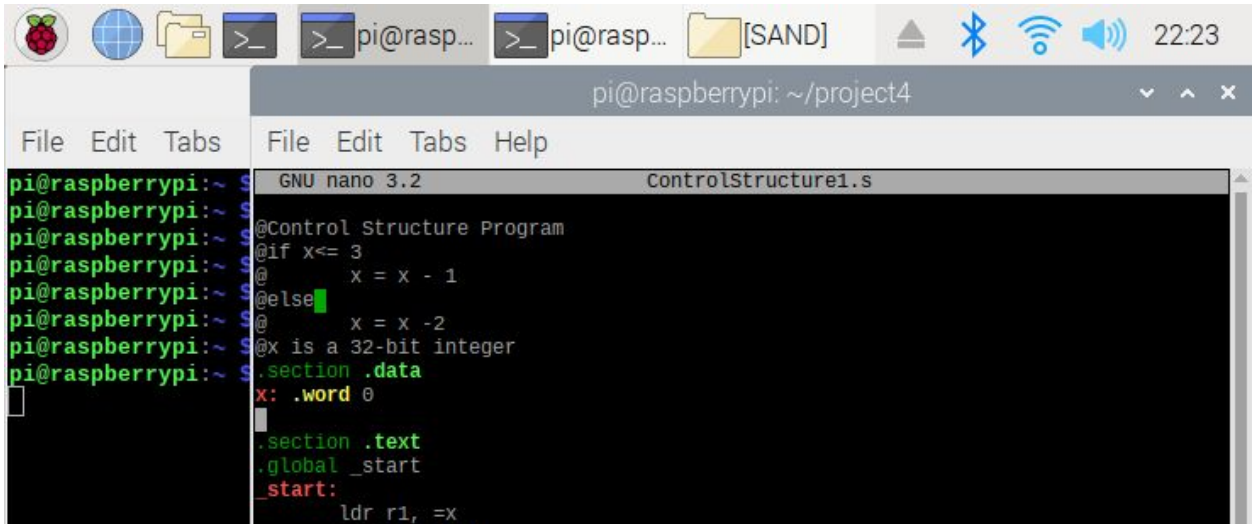


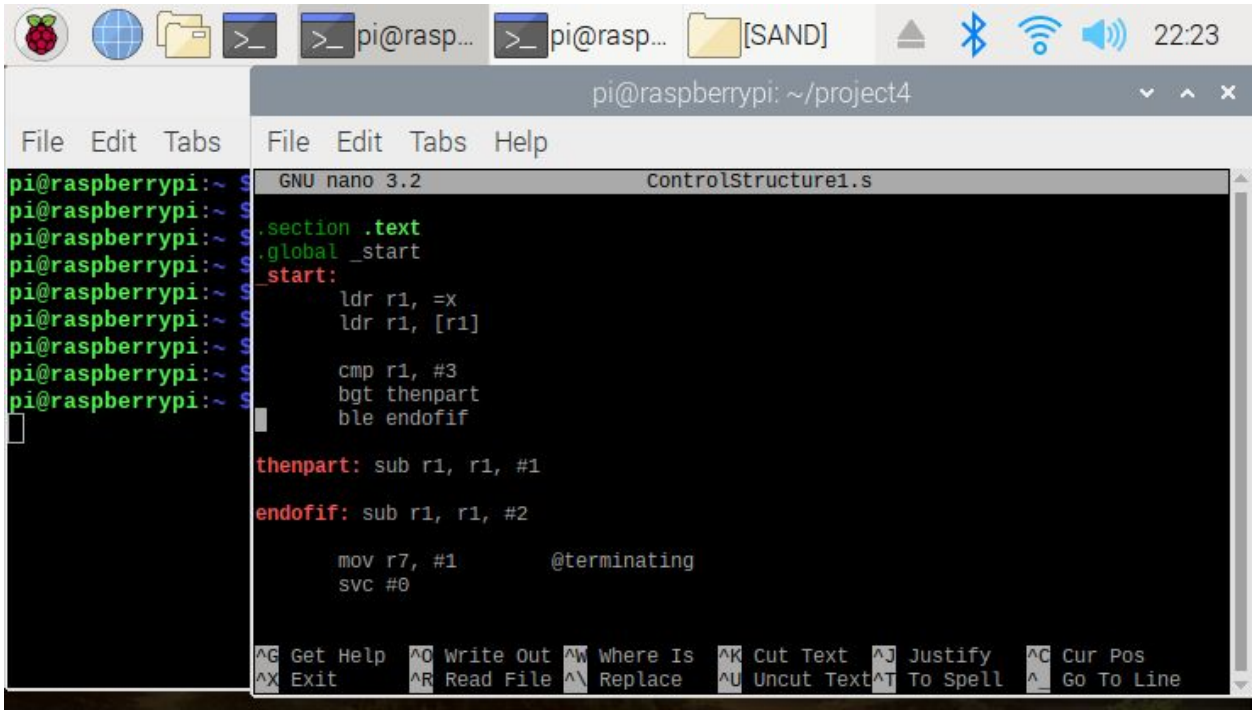
These 2 screenshots shown above are of the "fourth" program where it is a more efficient program as it does not require the use of back to back branches.



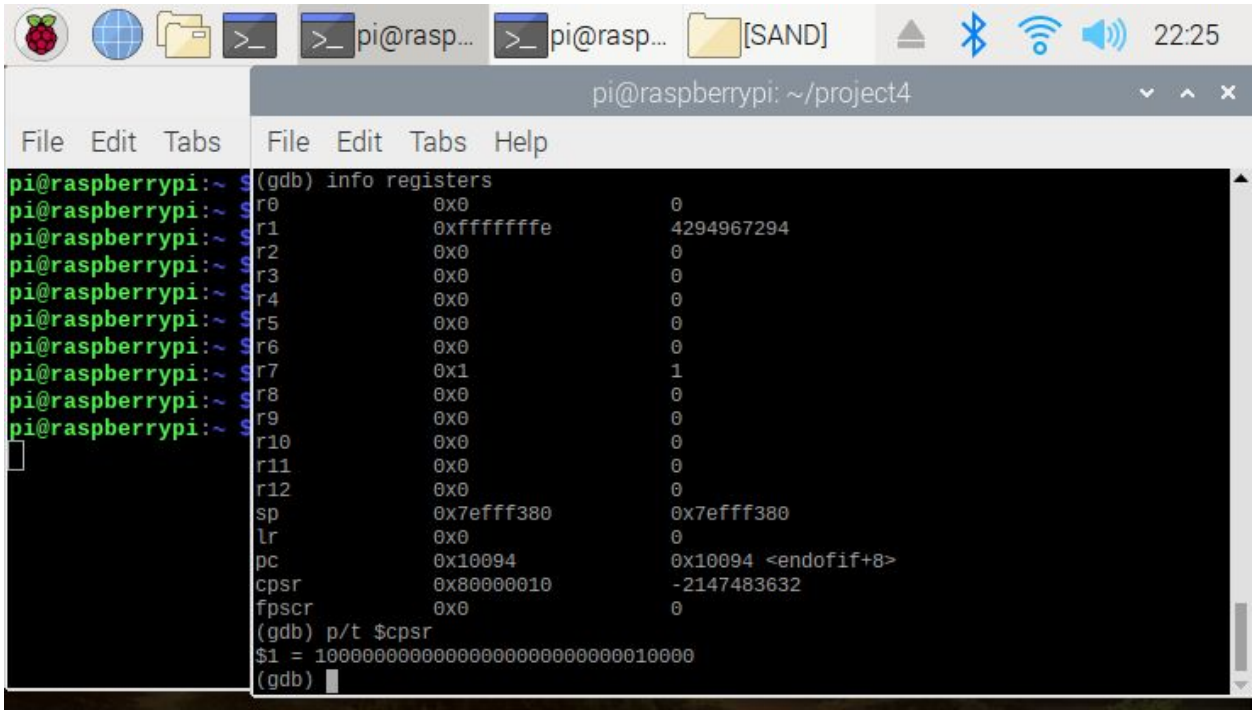
This screenshot is of the second version of the "fourth" program where the z flag is displayed at the bottom of the screenshot. The z flag is cleared so that means that the value is not zero. The answer is shown to be in register 3.

Part 3





These two screenshots are of the program “ControlStructures1”. This is a program where I have to translate a if else statement to arm assembly. The issue I encountered with this is getting the program to branch to the correct statement when x is less than or equal to 3.



This screenshot is the execution of the "ControlStructure1" program. The answer is in register 1 which is -2. The reason for why it is -2 is that x is zero and that is less than 3 so the program directs itself to the else statement where x=x-2. The z flag is also shown at the bottom of the screenshot and it is clear indicating that the value is not zero.

Parallel Programming Skills Foundation: Andre Nguyenphuc

- ➔ Race Condition:
  - ◆ What is race condition?
    - When the result of a software or system is dependent on the timing of other uncontrollable events
  - ◆ Why is race condition difficult to reproduce and debug?
    - The end result is nondeterministic and depends on the timing between interfering threads. There will be different timings everytime which leads to different results so it is hard to reproduce and get the same results.
  - ◆ How can it be fixed? Provide an example from your Project_A3 (see spmd2.c)
    - It can be fixed by initializing within the pragma.In spmd2.c you do not need to declare the id and numThreads variables separately at the start instead you will declare it in the #pragma omp parallel which would allow you to get unique thread numbers printed out.
- ➔ Summarize the Parallel programming Patterns section in the "Introduction to Parallel Computing_4.pdf" (two pages).

- There are two main categories that patterns can be grouped into which are Strategies and Concurrent Execution Mechanisms. Concurrent Execution Mechanisms is the parallel code patterns which are being written for the hardware to allow parallelism. Within this there are two categories of patterns which are Process/Thread control patterns and Coordination Patterns. Strategies is when a decision has to be made on how tasks can be done concurrently by multiple CPUs. Within this there are two categories which are algorithmic strategies and implementation strategies. Implementation strategy is determined by the algorithmic strategy.

➔ In the section "Categorizing Patterns" in the "Introduction to Parallel Computing_3.pdf" **compare** the following:

- Collective synchronization (barrier) with Collective communication (reduction)
  - A barrier is when threads/processes have to stop at the barrier and cannot continue until all the threads/processes reach the barrier. Reduction is when communication is coordinated among a group of processes in a communicator who collects data from all the processors and gets the result.
- Master-worker with fork join
  - Master-worker is when one thread which is the master executes a block of code when it forks while the rest of the threads which are the workers execute a different block of code when they fork. Fork-join is when execution branches off in parallel at designated points which then merge at a subsequent point and resume execution.

➔ Dependency: Using your own words and explanation, answer the following:

- Where can we find parallelism in programming?
  - Parallelism in statement level can be found between program statements and at block level/loop level/routine level/process level can be found in larger-grained program statements.
- What is dependency and what are its types (provide one example for each)?
  - Dependency is when one operation depends on an earlier operation to complete and produce a result before its own operation can be performed.
  - The types of dependency are flow,anti,output,and control
    - Flow (True) - A=2, B=A, C=B
    - Anti - B = 2, A = B + 3, B =5
    - Output- B = 2 , B2 = B, A = B2 + 3, B = 5
    - Control - if(A == B), A = A + B, B = A + B
- When a statement is dependent and when it is independent (provide two examples)?
  - Dependent is when it the order of the execution affects the result

- S1: a = 3 , S2: b= 4
- Independent is when the order of the execution does not affect the result
  - S1: a=3, S2: b=a

◆ When can two statements be executed in parallel?
- Two statements can be executed in parallel when there is no dependencies between the two statements which means when they are independent

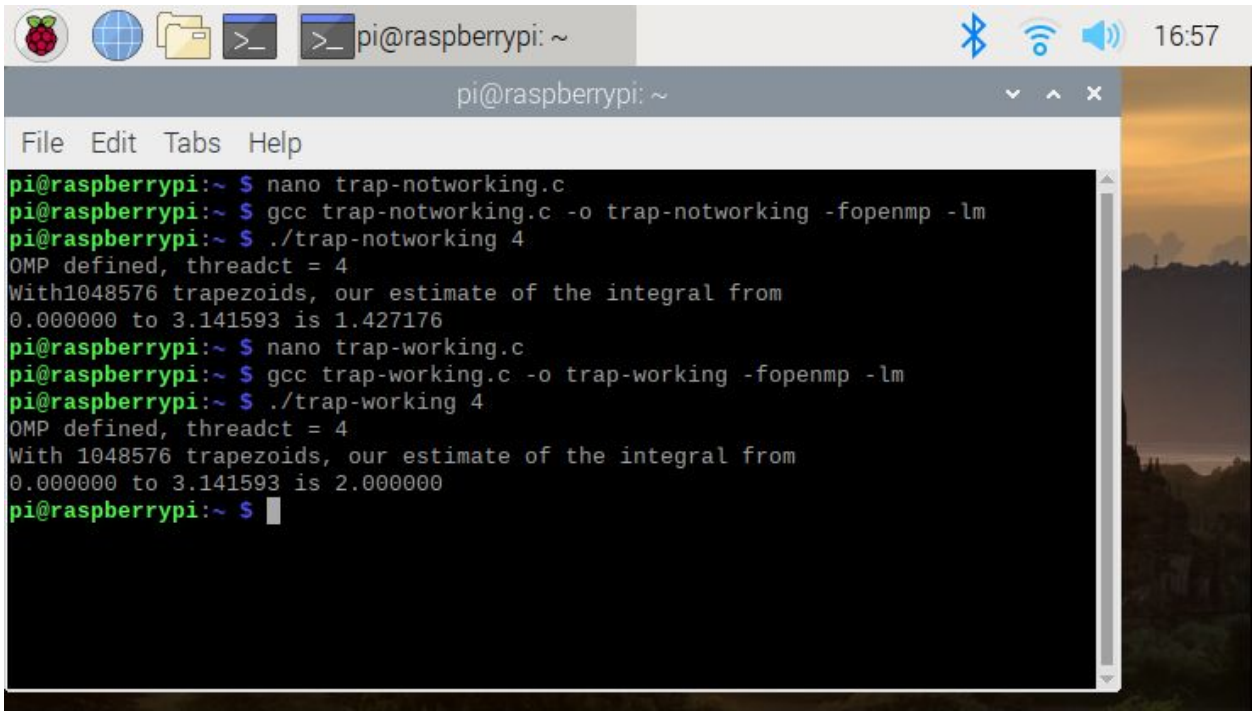
◆ How can dependency be removed?
- By either rearranging the statements so that there is no more dependency or erasing the statement so that the statement that was dependent is no longer there

◆ How do we compute dependency for the following two loops and what type/s of dependency?
- In the example below we can compute dependency by finding the IN(S) and OUT(s). They are both Flow (True) dependency because both rely on the i from the for loop
-

```
for(i=0;i<100;i++)          for(i=0;i<100;i++){
  S1: a[i]=i;                  S1:a[i]=i;
                               S2:b[i]=2*i;
                              }
```

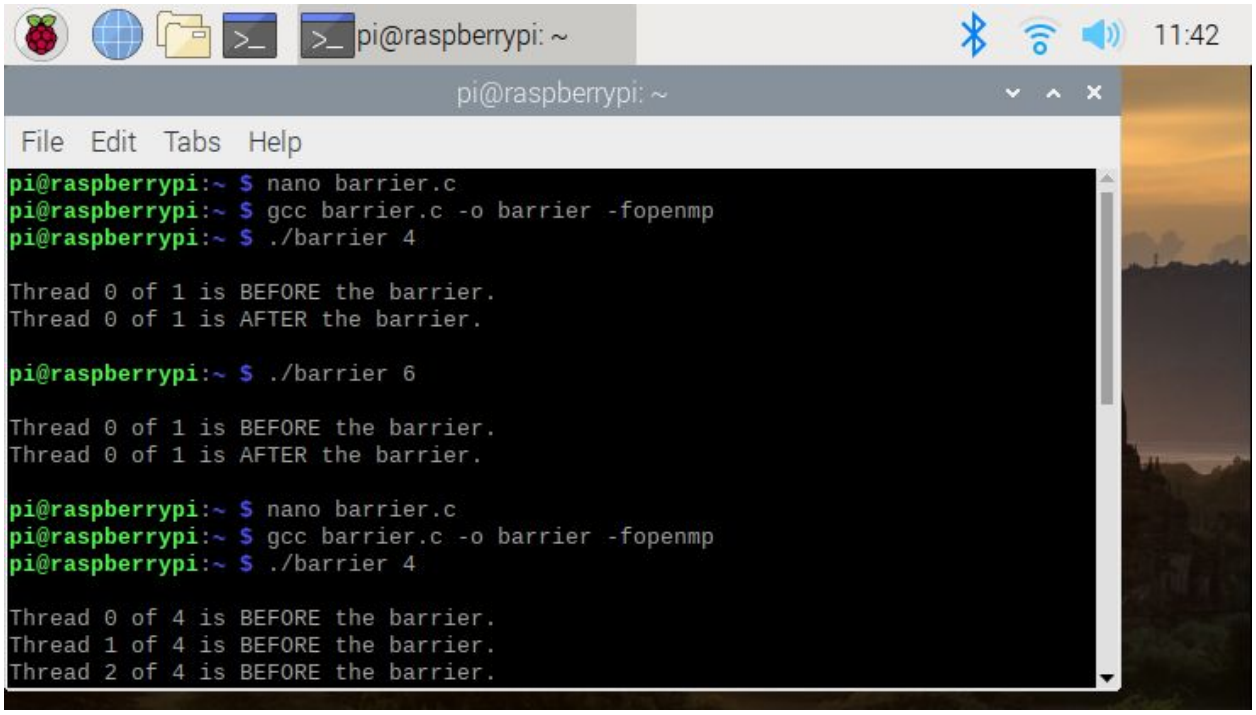
Parallel Programming Basics: Andre Nguyenphuc

Part 1

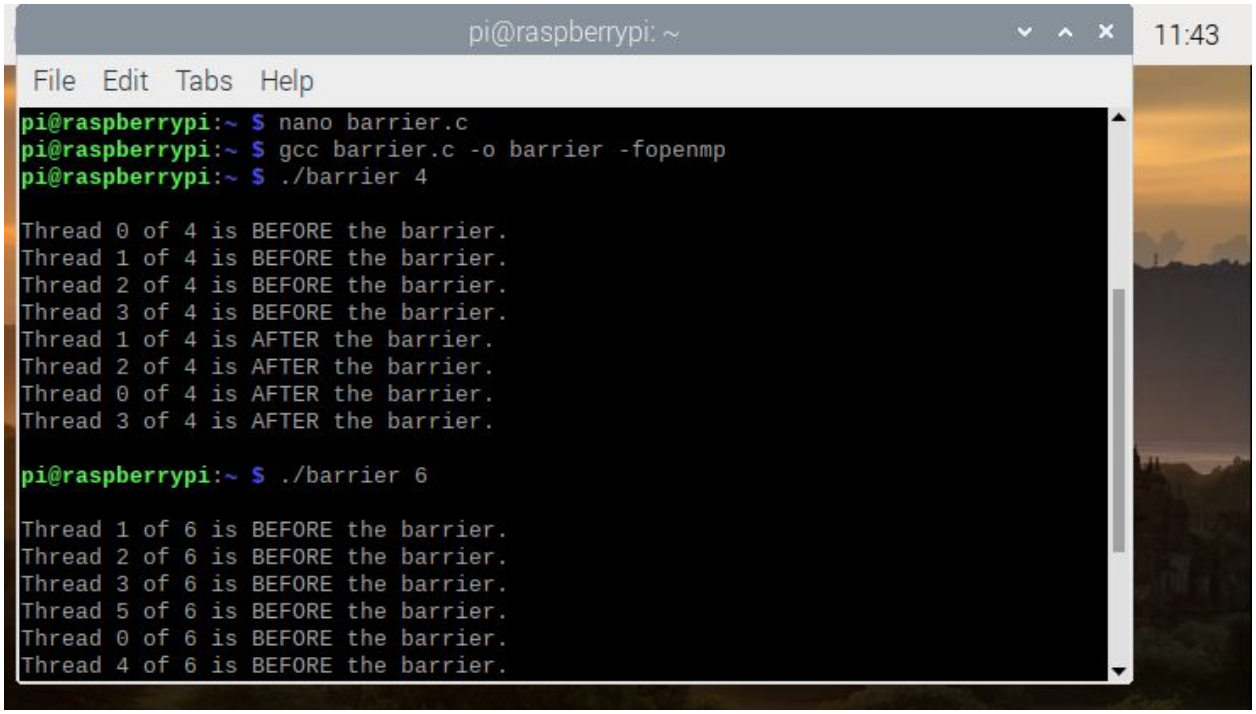


In this picture are the results for both trap-not working and trap-working. In trap-not working code I had everything the example had including #pragma omp parallel for private(i) shared (a,n,h,integral). This code did not work because the answer I got was 1.427176. I believe I did not get 2.0 because the variables are being declared to be either private or shared. I then worked on the trap-working which has similar code as trap-not working however the difference is that it has #pragma omp parallel for \ private(i) shared(a,n,h) reduction(+: integral). This gave me the answer 2.0.

Part 2

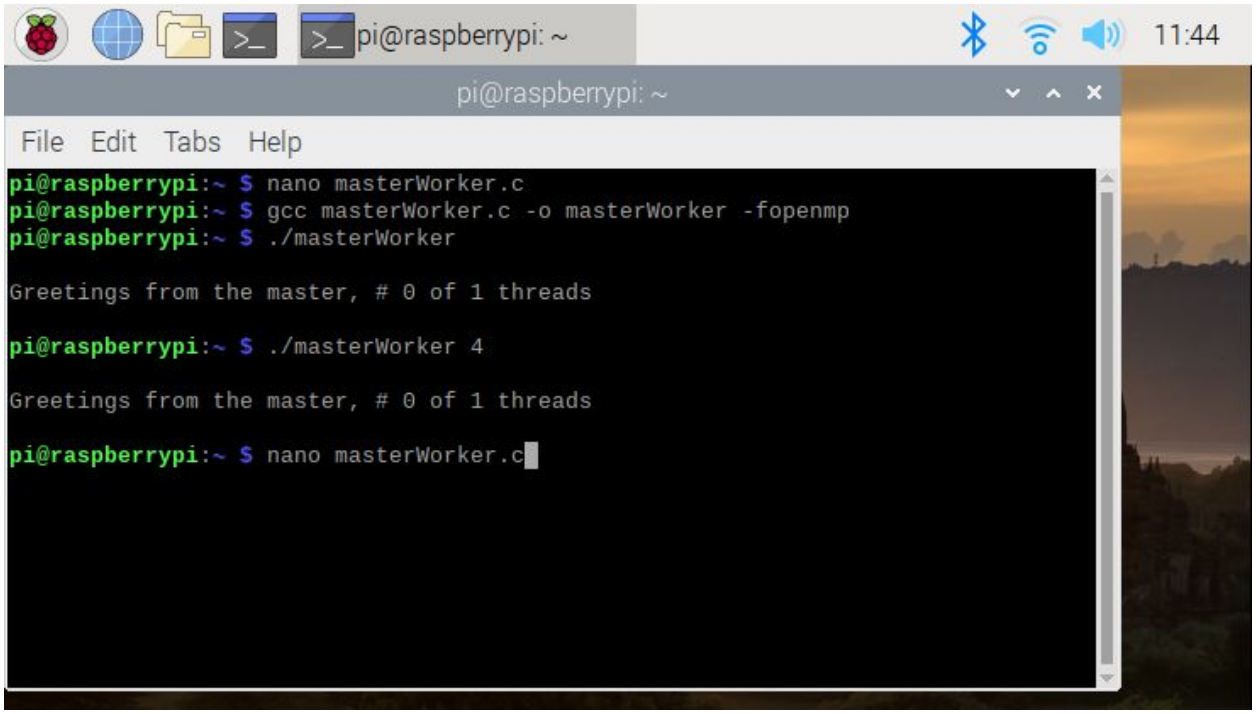


In this picture is my result for barrier with a comment still before the #pragma omp parallel at line 31 in the given code. I noticed that no matter what value I put in the numThreads I still get Thread 0 of 1 is BEFORE the barrier and Thread 0 of 1 is AFTER the barrier.

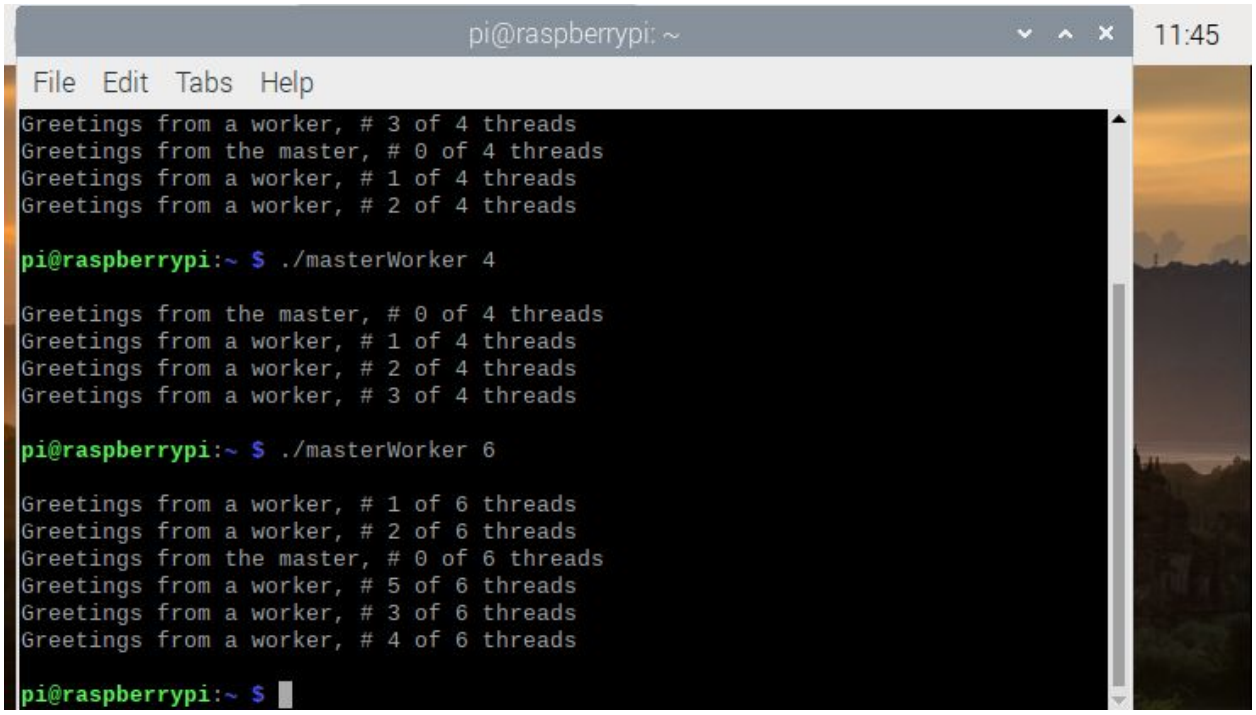


In this picture is my result for barrier after I removed the comments from #pragma omp barrier as instructed. I noticed that in this picture the values I put in will be the value of the thread unlike before. For example I put 4 as my value and I will get the intended result which is 4 BEFOREs and 4 AFTERs.

Part 3



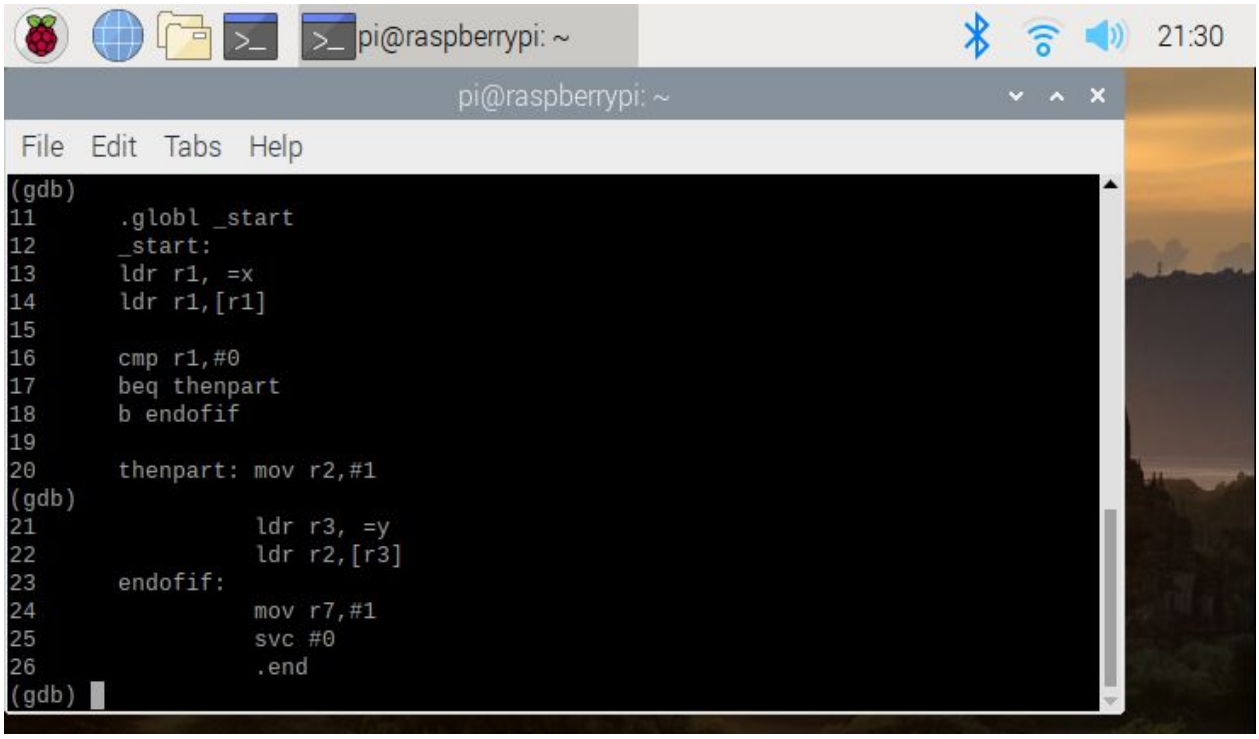
In this picture is the result of the masterWorker code with the #pragma omp parallel at line 24 still commented. This gave me the result of Greetings from the master, #0 of 1 threads. This is still the output I get even when I change the value to 4.



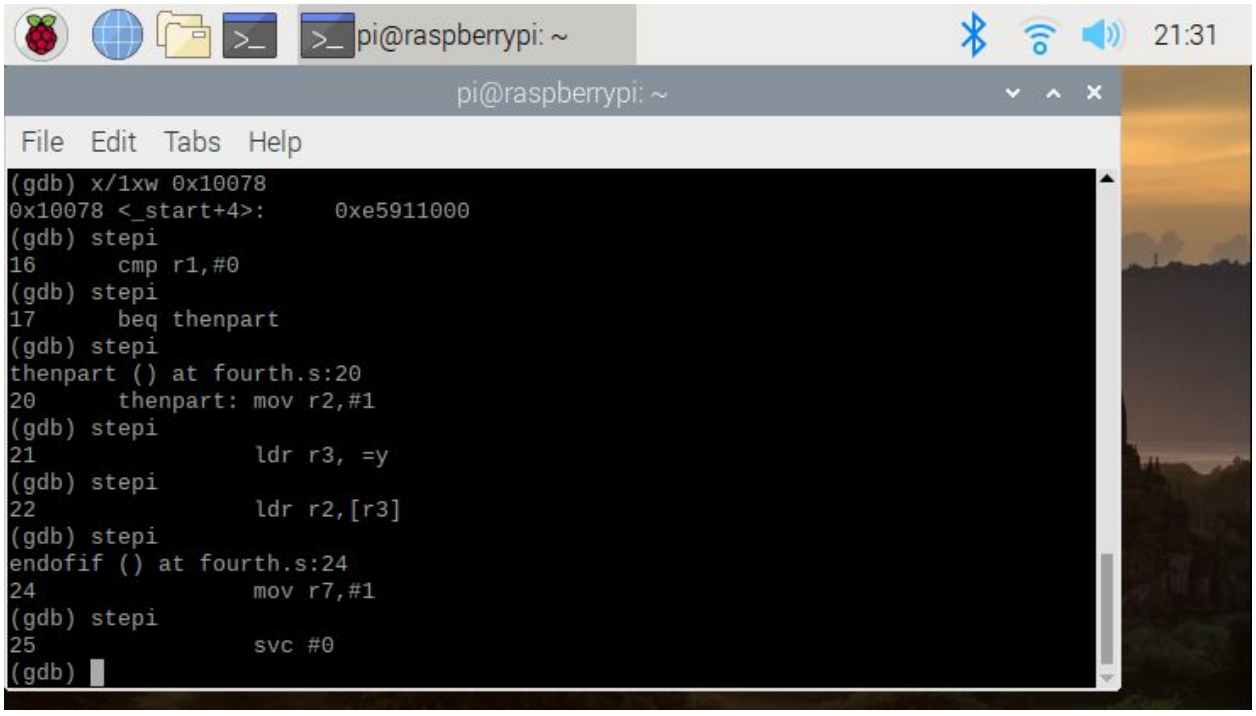
In this picture is the results when I removed the comments from the #pragma omp parallel. Instead of just getting the master thread I also get the worker threads. Also instead of just one line of output, the output corresponds to the value I put. An example of this would be 4 instead of just getting the single master thread I also get three worker threads. I then tried 6 to be sure and I got one single master thread and 5 worker threads.

ARM Assembly Programming: Andre Nguyenphuc

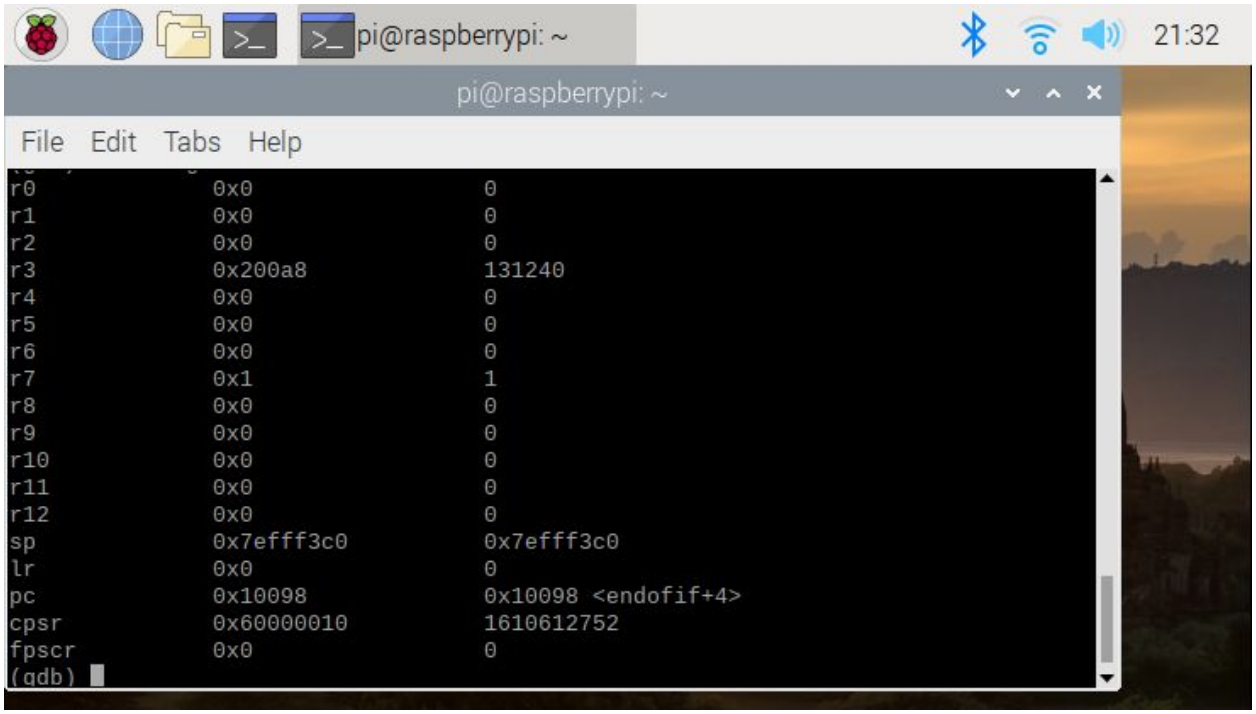
Part 1



Code for the fourth program

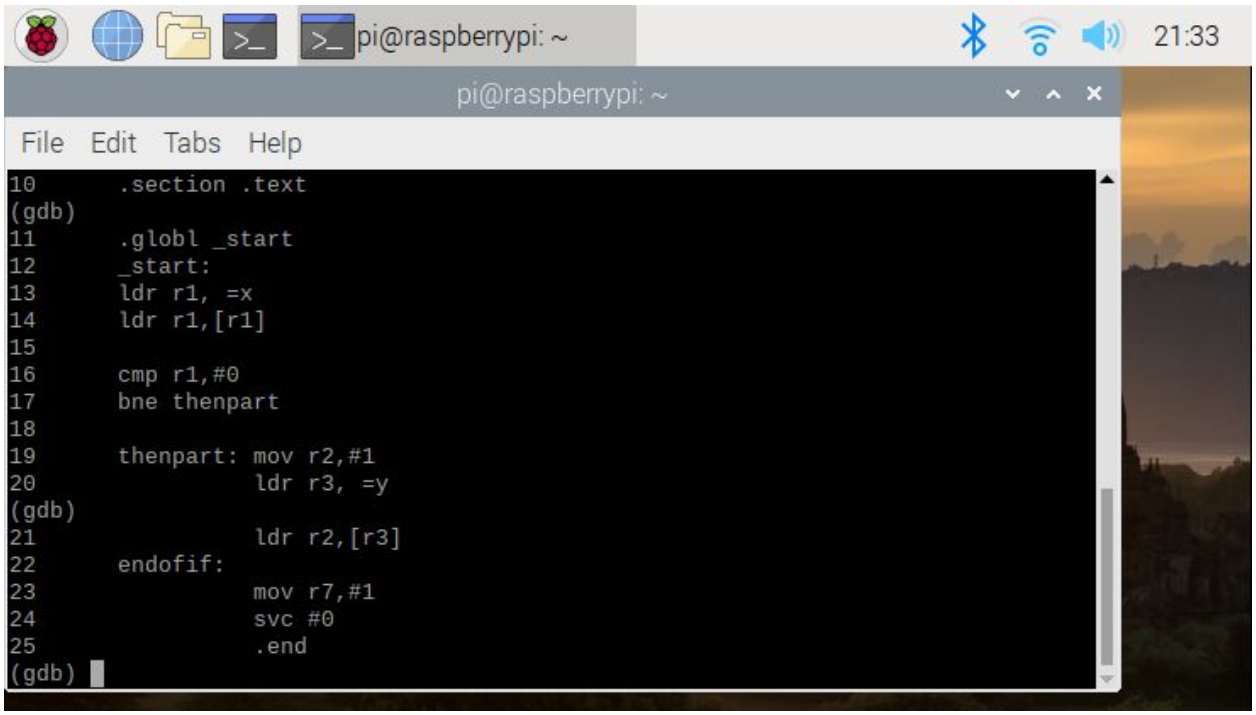


In this picture I show the memory location at 0x10078 and I got 0xe5911000 and I continued to step over until I got to the end

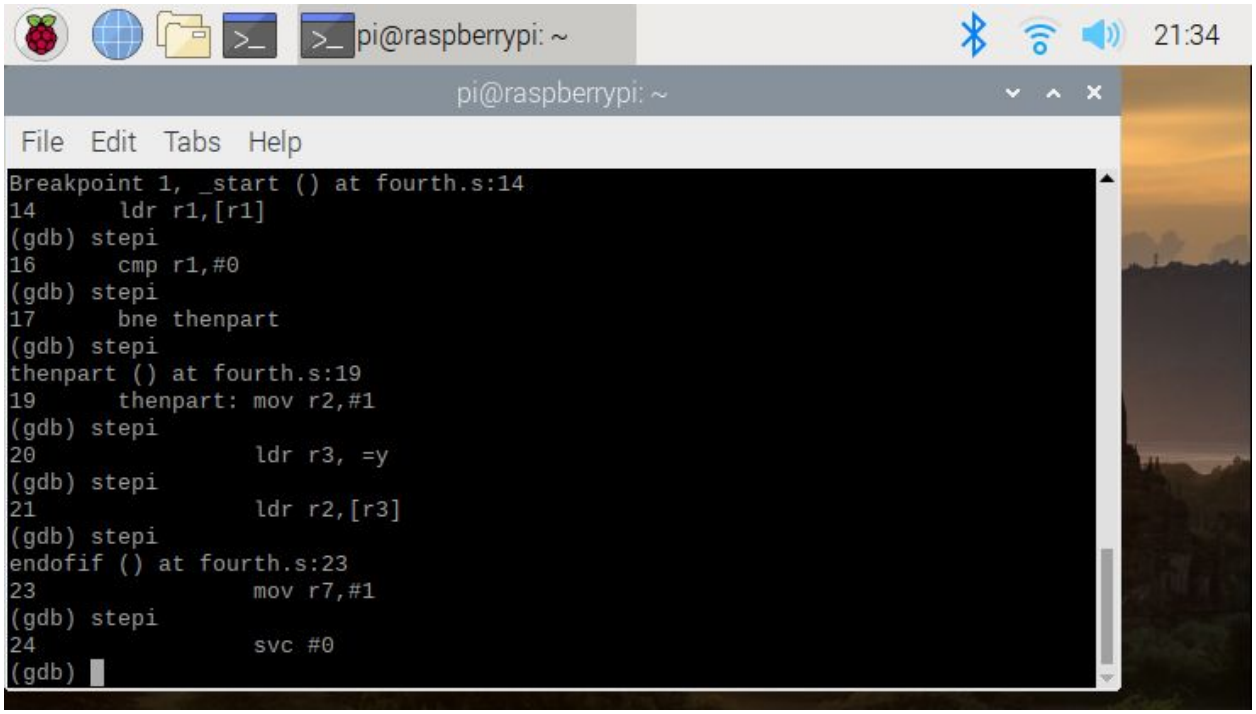


In this picture I am showing the info registers and I get 200a8 inside register 3 and for the Z flag I get 0 because the value is not 0 so the flag is not set

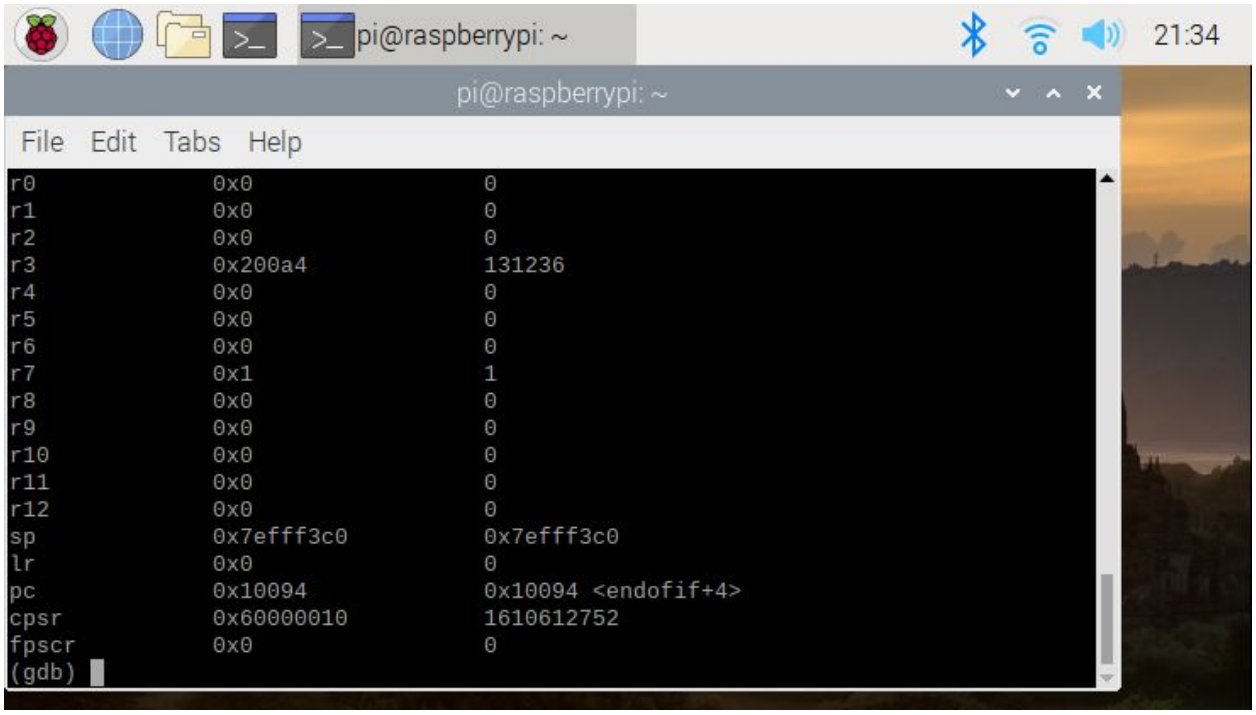
Part 2



The program in part one is not efficient because it contains back-to-back branches so in this code I change beq with bne and then I removed the b instruction. I use bne because I am jumping when the conclusion is false and this is using De Morgan's Law.

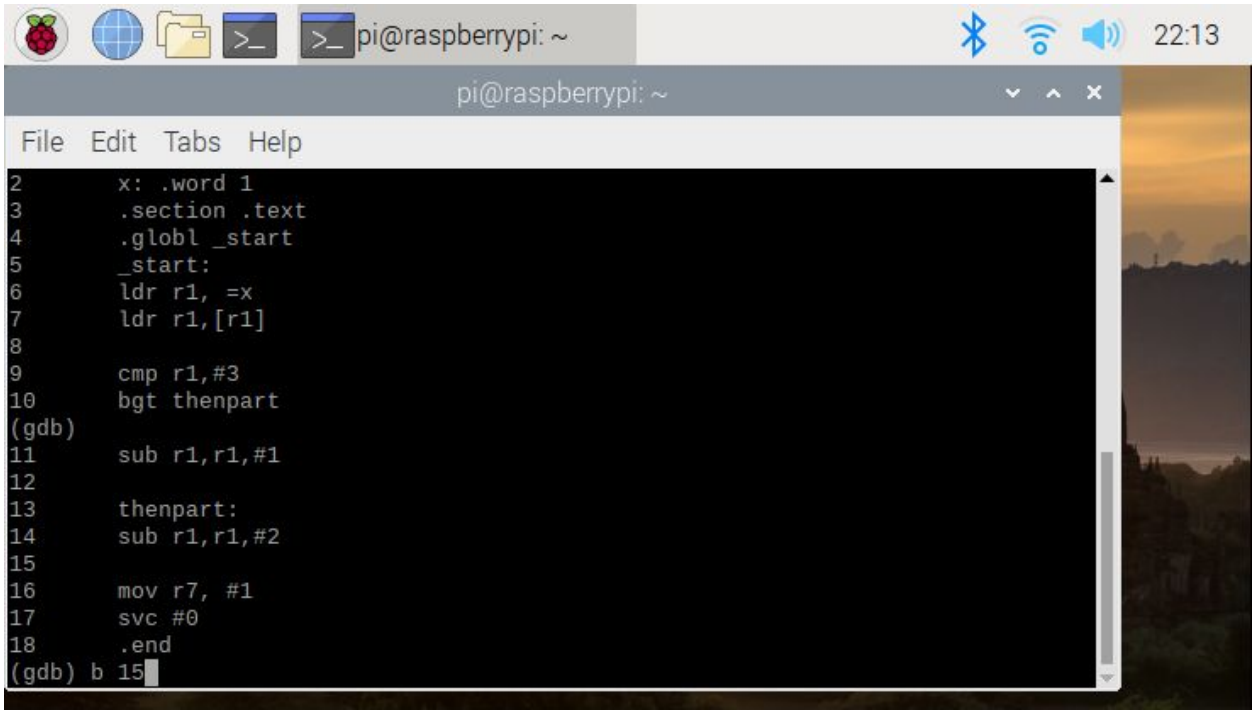


In this picture I set my breakpoints and then run the code. I then continue to step over until I reach the end of the code

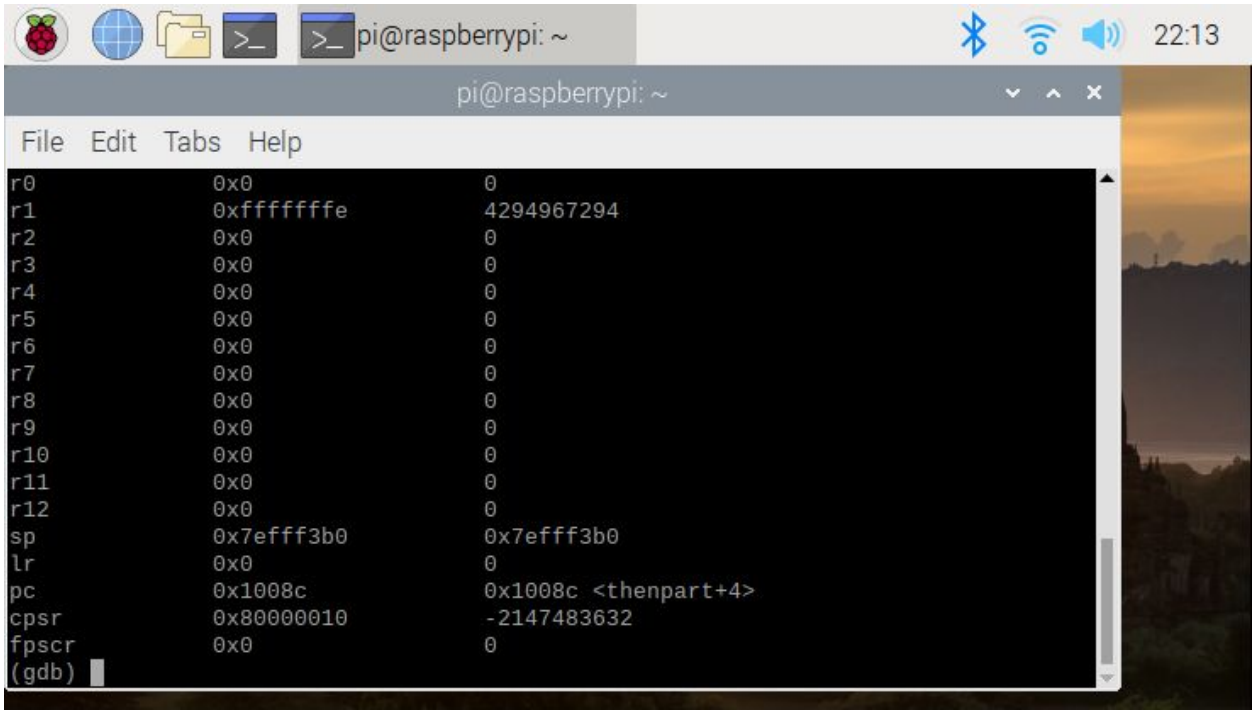


In this picture I am displaying the info registers and in register 3 I get 200a4 which is different from the first program. For the Z Flag I get 0 because the value is not 0 so the flag is not set

Part 3



In this picture is my code for part 3 of the ARM assembly. I first put 1 as a 32 bit integer for x. I then loaded the memory address of x into r1 and loaded the value of x into r1. I then compare r1 with 3. I used De Morgan's law so I used bgt thenpart which means if it is greater than it jumps. If it does not jump I subtract 1 from r1 and set that value to r1. If it does jump I subtract 2 from r1 and set that value to r1.



In this picture I am displaying the info registers and I get fffffffe in register 1 and I get 0 for the Z flag because the answer is not 0 so the flag is not set.

Parallel Programming Skills Foundation Miguel Romo:

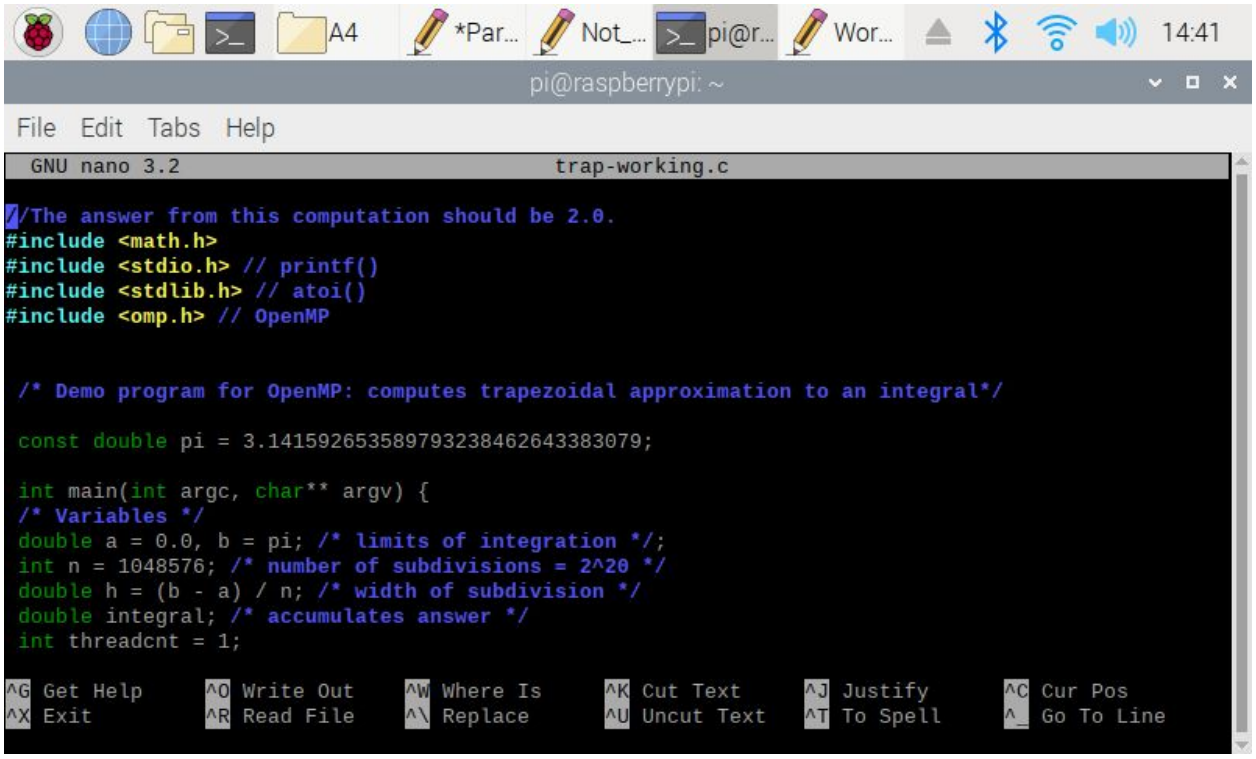
- ➔ Race condition:
  - ◆ What is race condition?
    - A race condition or race hazard is the behavior of an electronics, software, or other system where the output is dependent on the sequence or timing of other uncontrollable events
  - ◆ Why is race condition difficult to reproduce and debug?
    - Race conditions are difficult to reproduce and debug because the result is nondeterministic and depends on the relative timing between interfering threads.
  - ◆ How can it be fixed? Provide an example from your Project_A3 (see spmd2.c).
    -). It is difficult to fix race conditions because problems disappear when running in debug mode, when additional logging is added or when attaching a debugger. Therefore, it is better to avoid race conditions by careful software design.
- ➔ Summaries the Parallel Programming Patterns section in the "Introduction to Parallel Computing_3.pdf" (two pages) in your own words (one paragraph, no more than 150 words).
  - ◆
- ➔ In the section "Categorizing Patterns" in the "Introduction to Parallel Computing_3.pdf" compare the following:
  - ◆ Collective synchronization (barrier) with Collective communication (reduction) A collective barrier is a logical point in the control flow of an algorithm at which all processes in a process group must arrive before any processes in the group are allowed to proceed further. Reduction collective operators act as a common communicative operator across versions of the same variable of all the processes.
  - ◆ Master-worker with fork join. In parallel computing, the fork–join model is a way of setting up and executing parallel programs, such that execution branches off in parallel at designated points in the program, to "join" (merge) at a subsequent point and resume sequential execution. In master-worker, The pattern consists of one process, called the master, executing one block of code while the rest of the processes, called workers, are executing a different block of code.
- ➔ Dependency: Using your own words and explanation, answer the following:
  - ◆ Where can we find parallelism in programming?

- Parallelism can be found when several computations need to be performed at the same time.

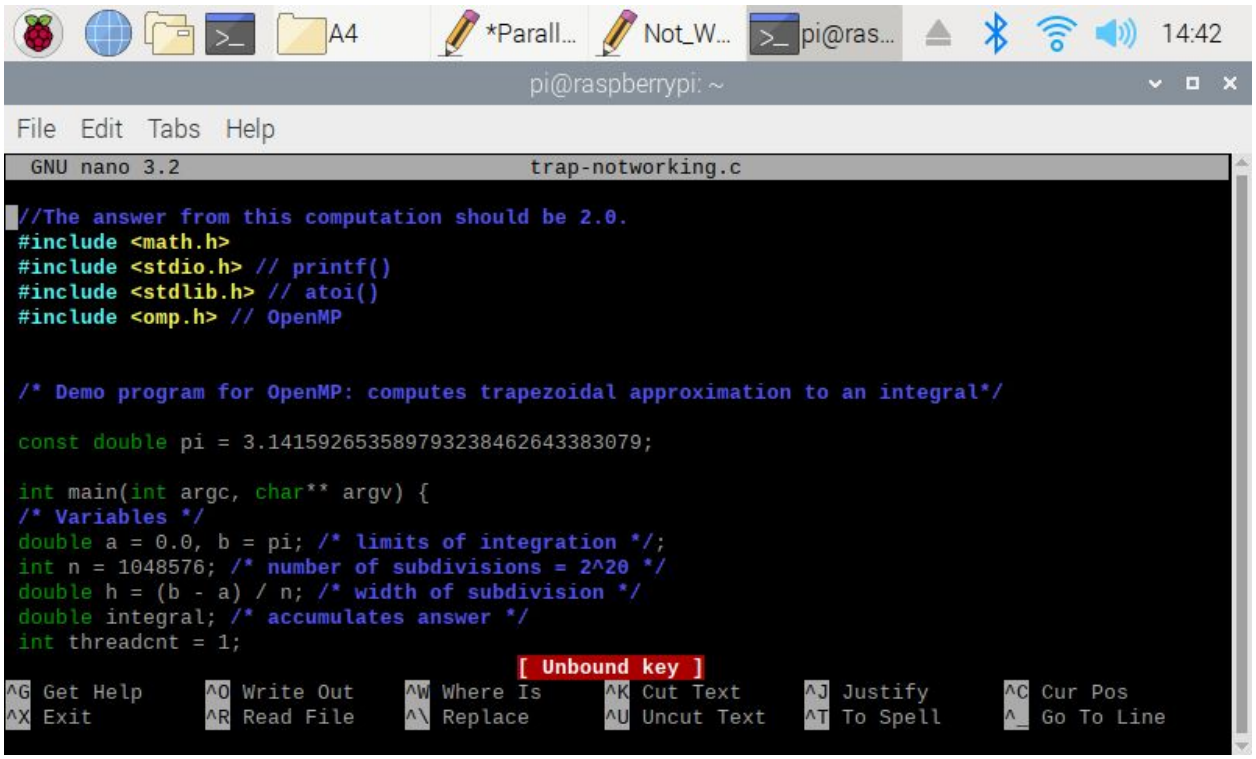
- What is dependency and what are its types (provide one example for each)?
  - Dependency is when a software relies on another one. They type are Class, Interface, and Method Dependencies,
    - Class Dependencies - A method in a class takes a String as parameter, therefore it depends on the String class.
    - Interface Dependencies - A method takes a CharSequence as parameter.
    - Method Dependencies - Dependencies on concrete methods or fields of an object.
- When a statement is dependent and when it is independent (Provide two examples)? Statements are dependent when the outcome of the program is dependent on another. Statements are independent of each other if they are true.
- When can two statements be executed in parallel?
  - Two statements can be executed in parallel if and only if there are no dependencies.
- How can dependency be removed?
  - Dependency can be removed by modifying the program.
- How do we compute dependency for the following two loops and what type/s of dependency?
  - You unroll the loop into separate iterations.
  - Both loops are flow dependency.

Parallel Programming Basics Miguel Romo:

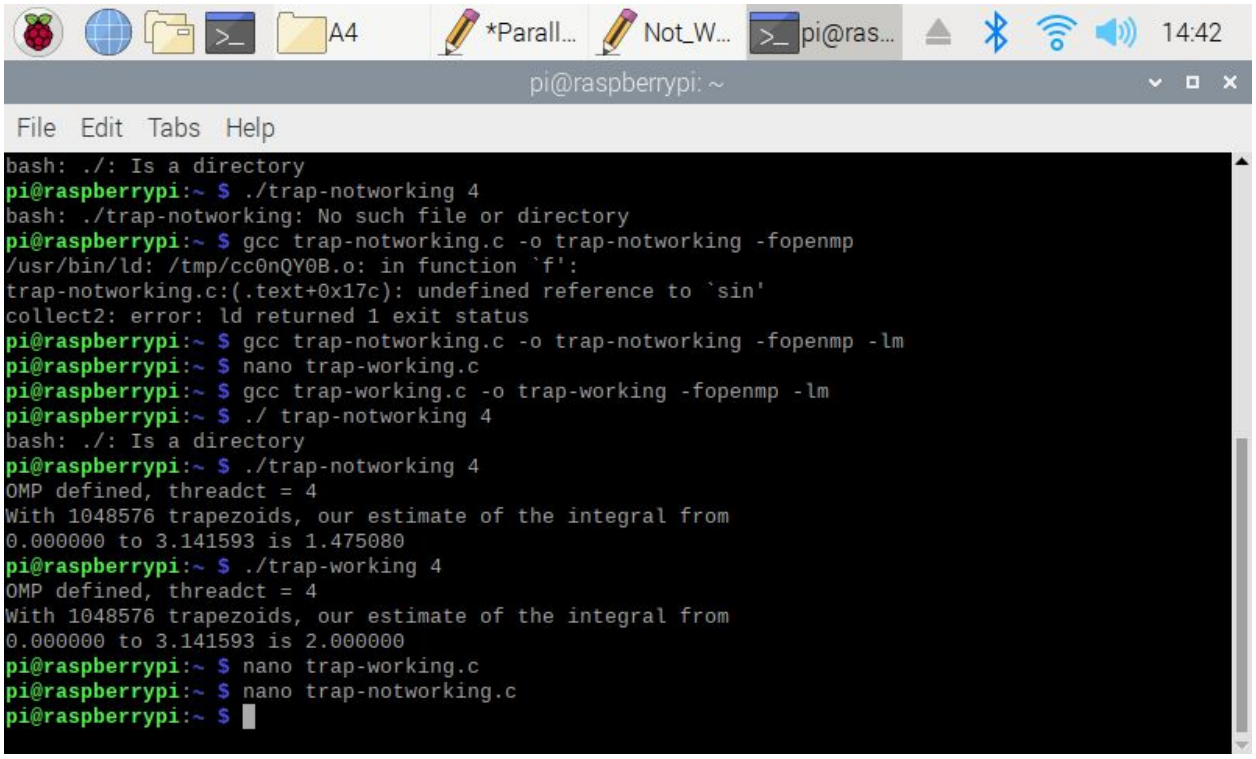
Part 1



Here is the code for trap-working.

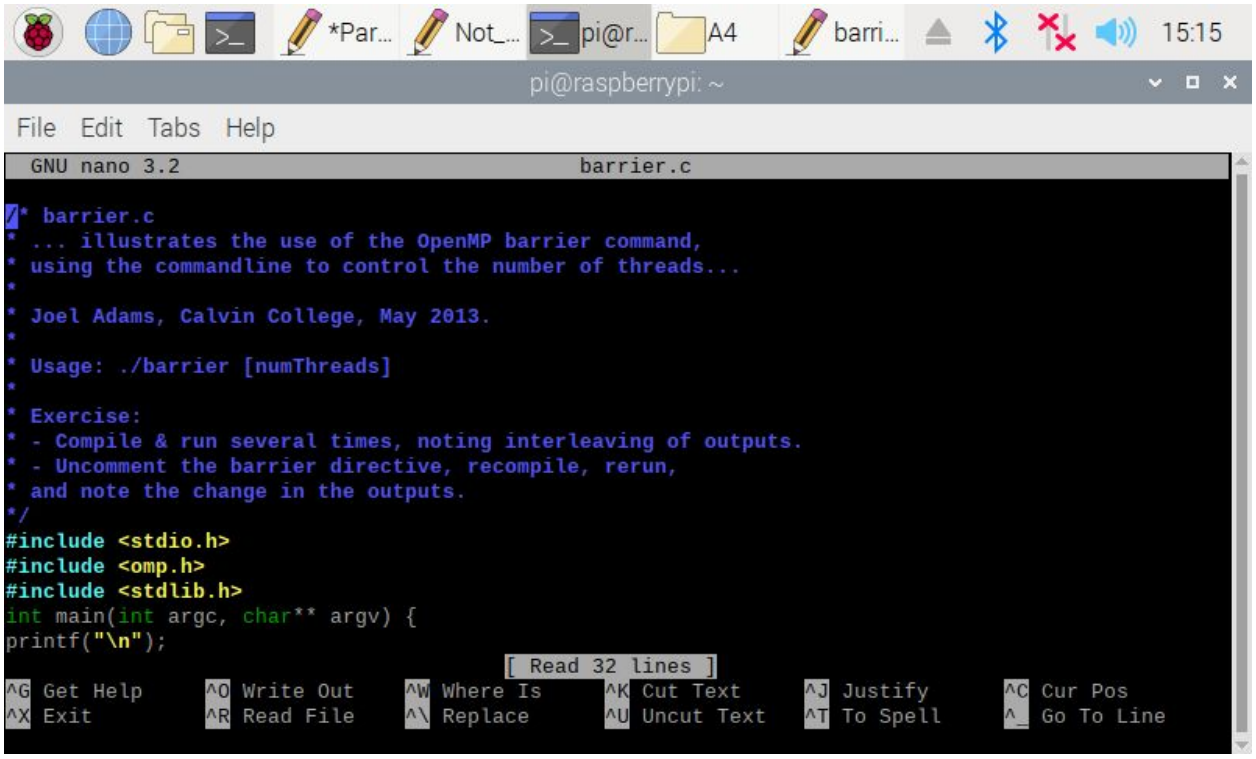


And this is the code for trap-notworking.

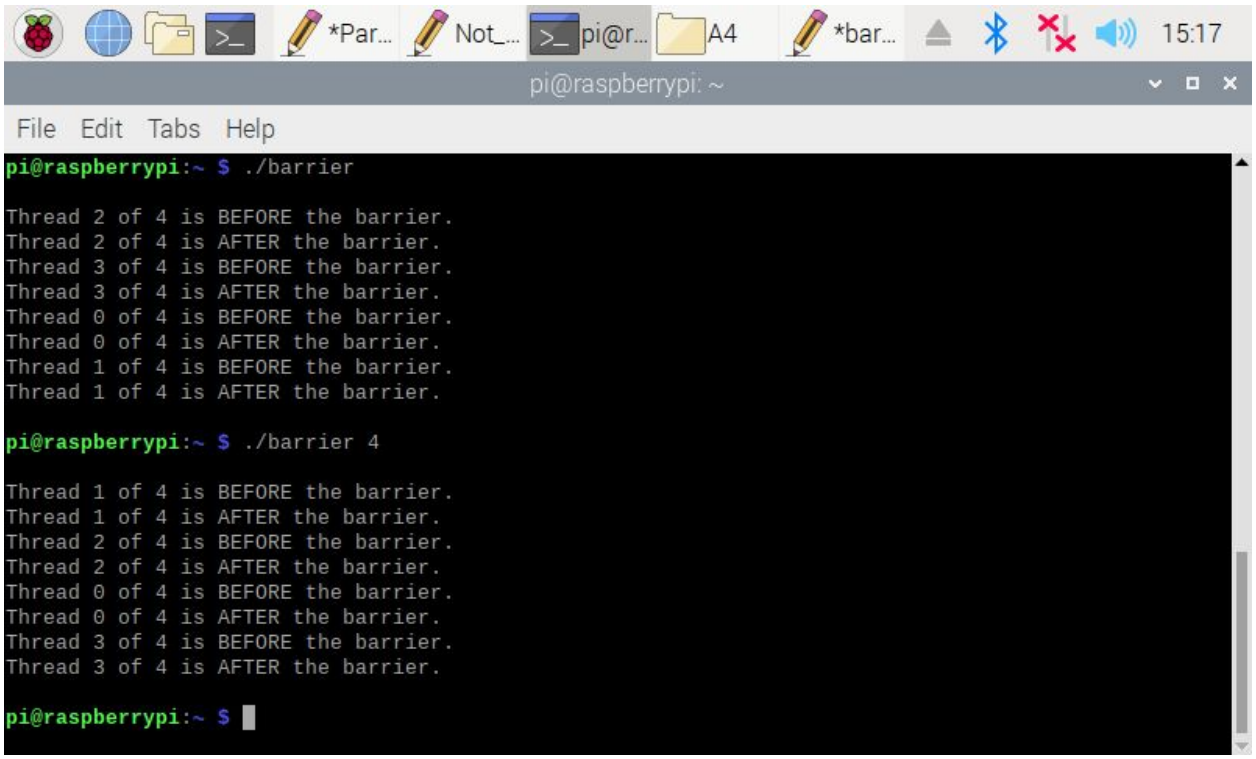


Here we have the values that both codes produced. trap-notworking 4 gave us a value of 1.4475080. And trap-working 4 gave us a value of 2.000000.

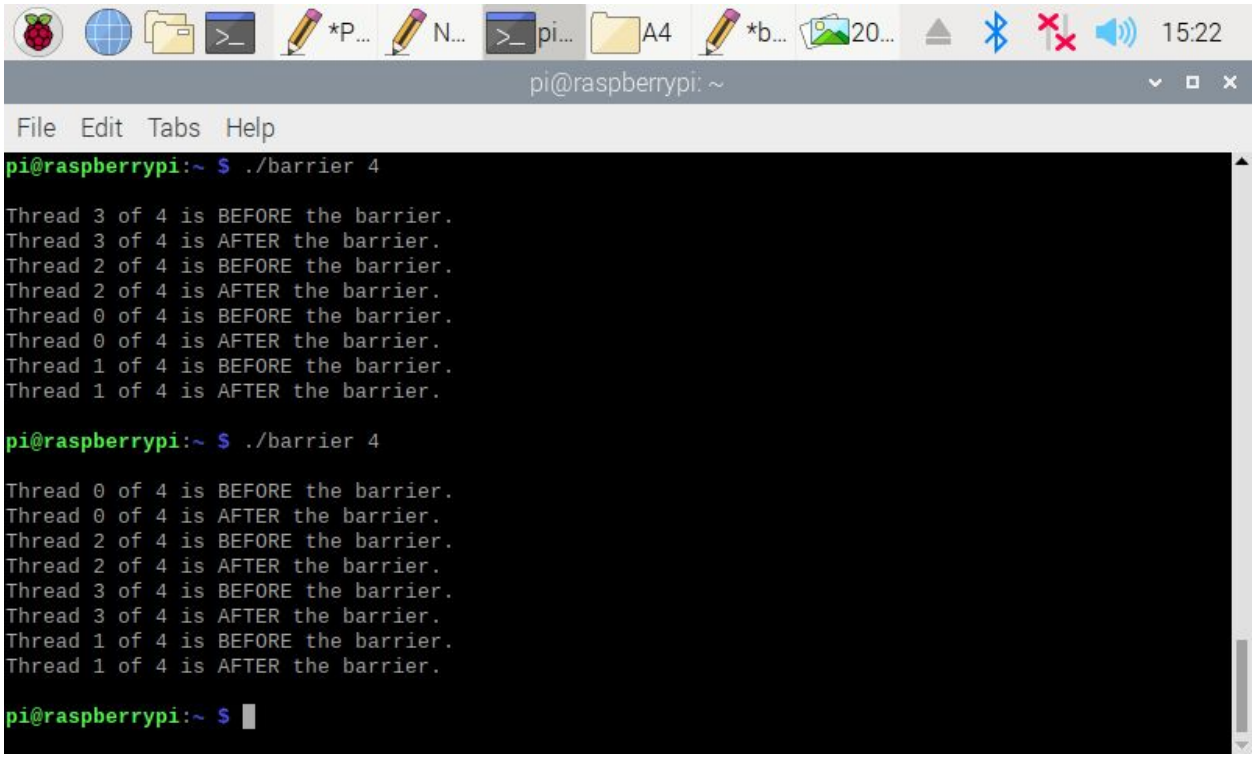
Part 2



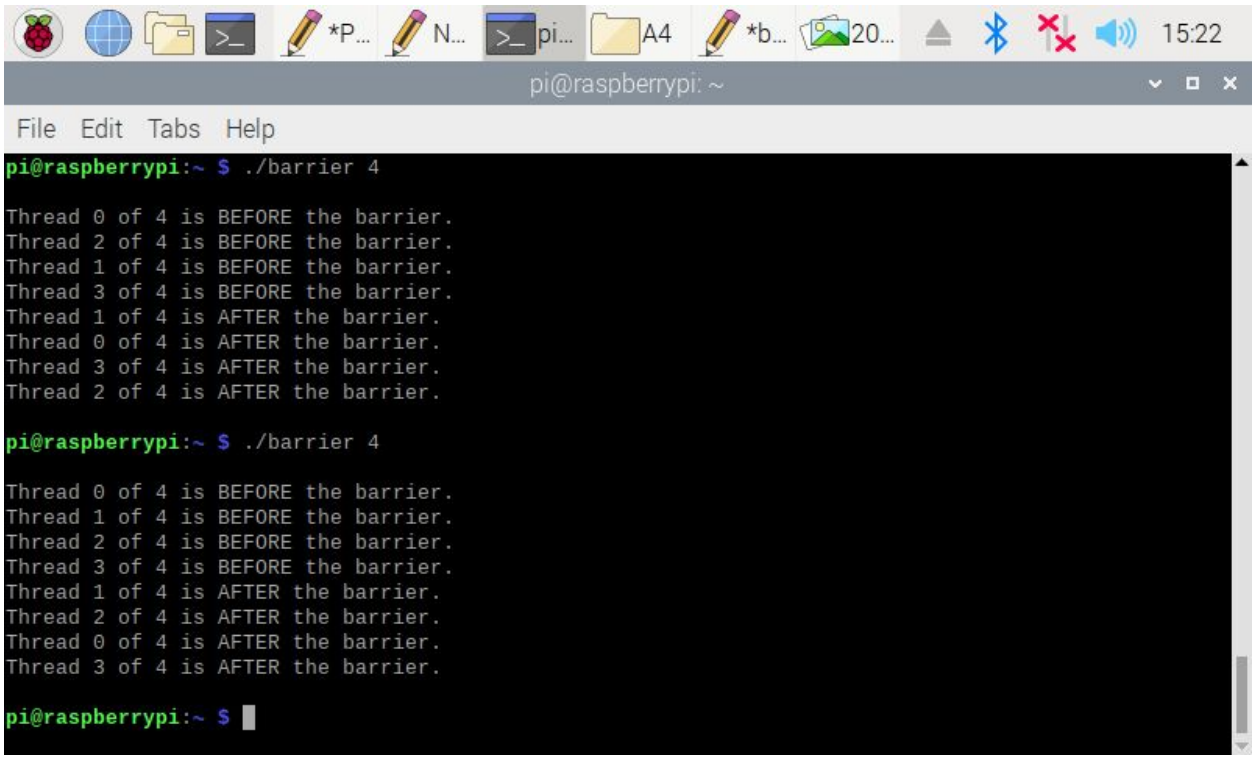
Here is the code for the barrier.



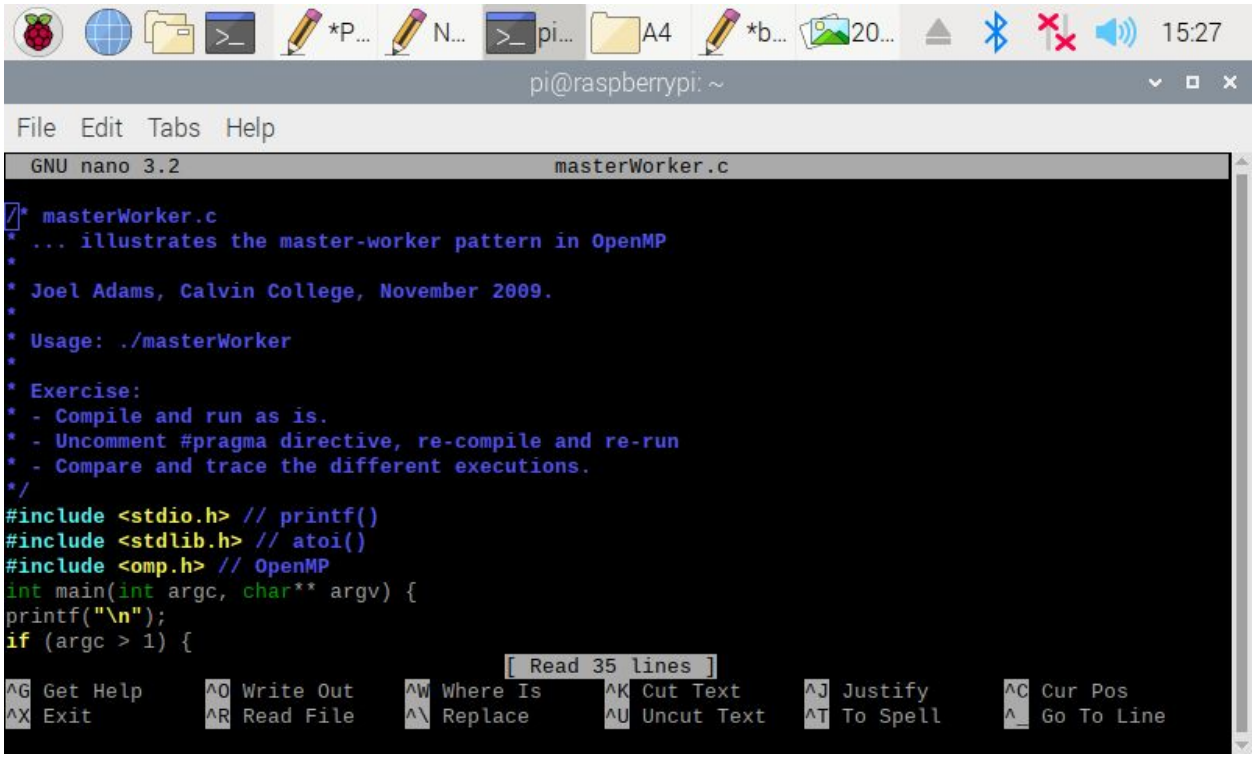
This is what the code produces with line 31 commented.



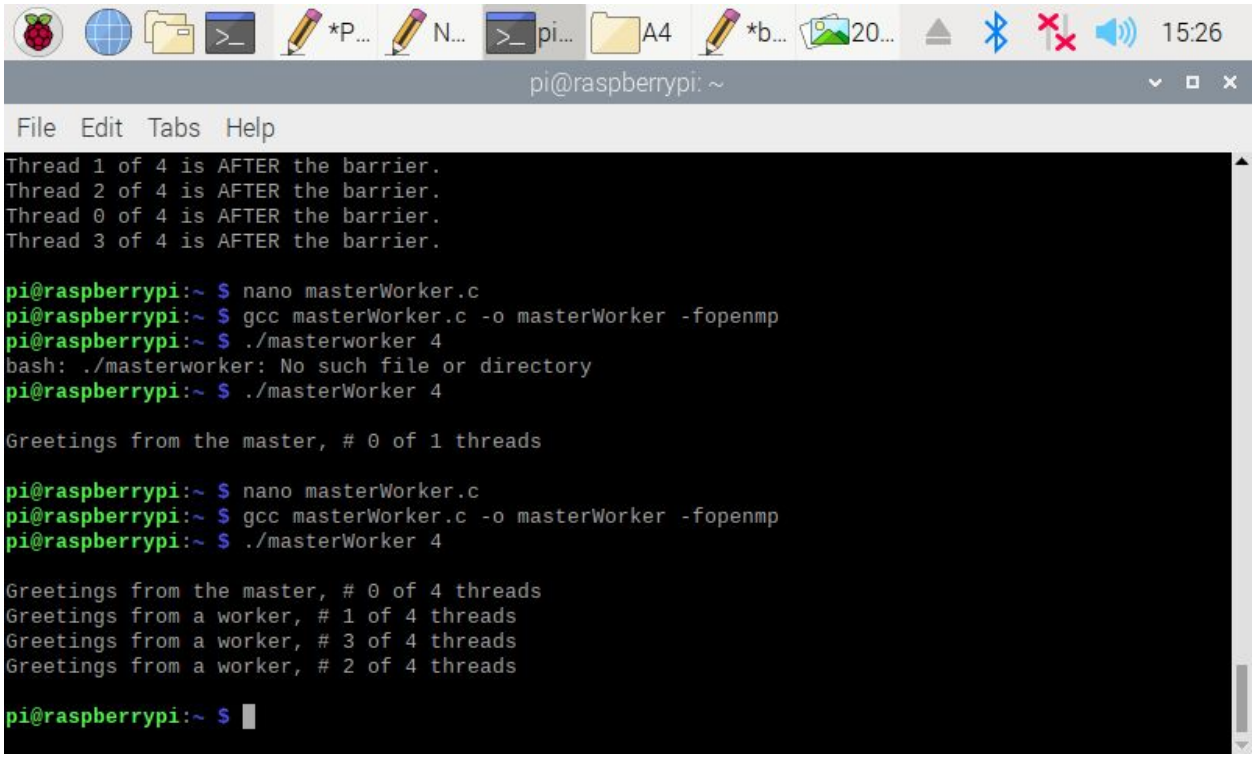
And again, we can observe that the before and after barrier alternate.



Here are the results after uncommenting line 31. We can see all the befores complete prior to the afters.

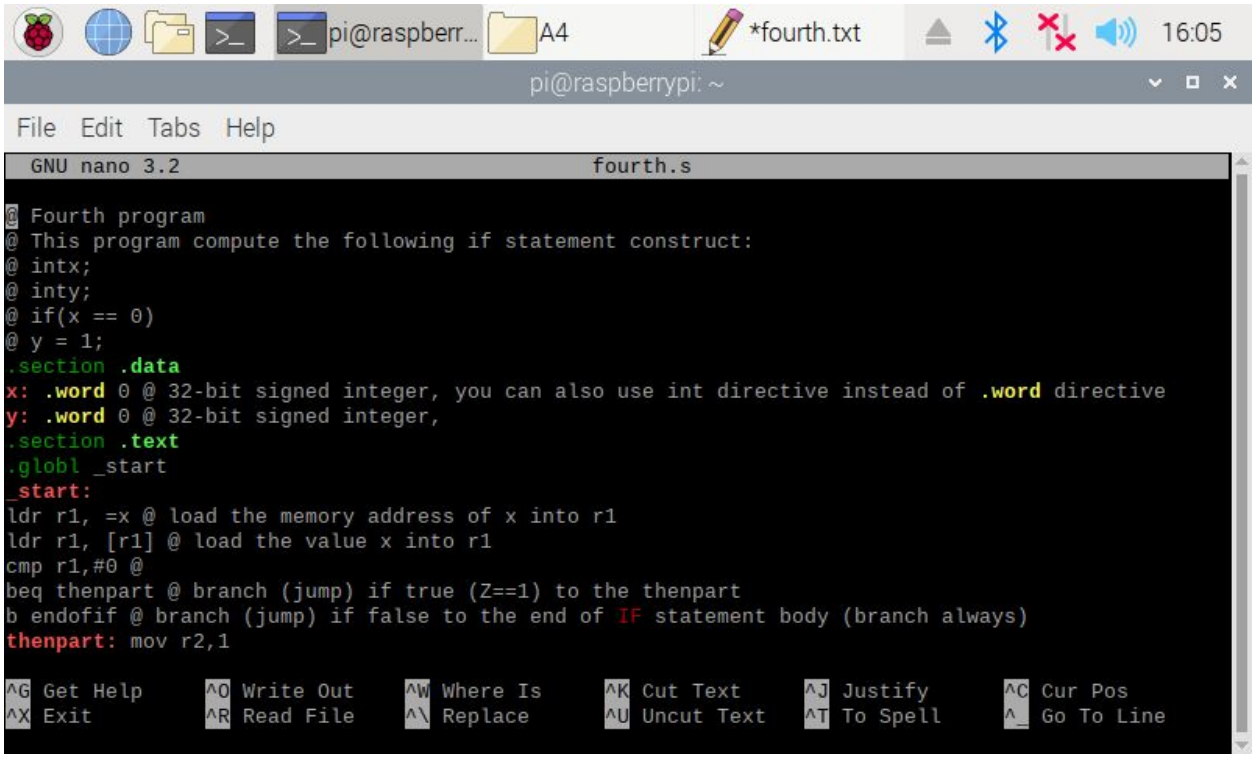


This is a screenshot of the masterWorker program.

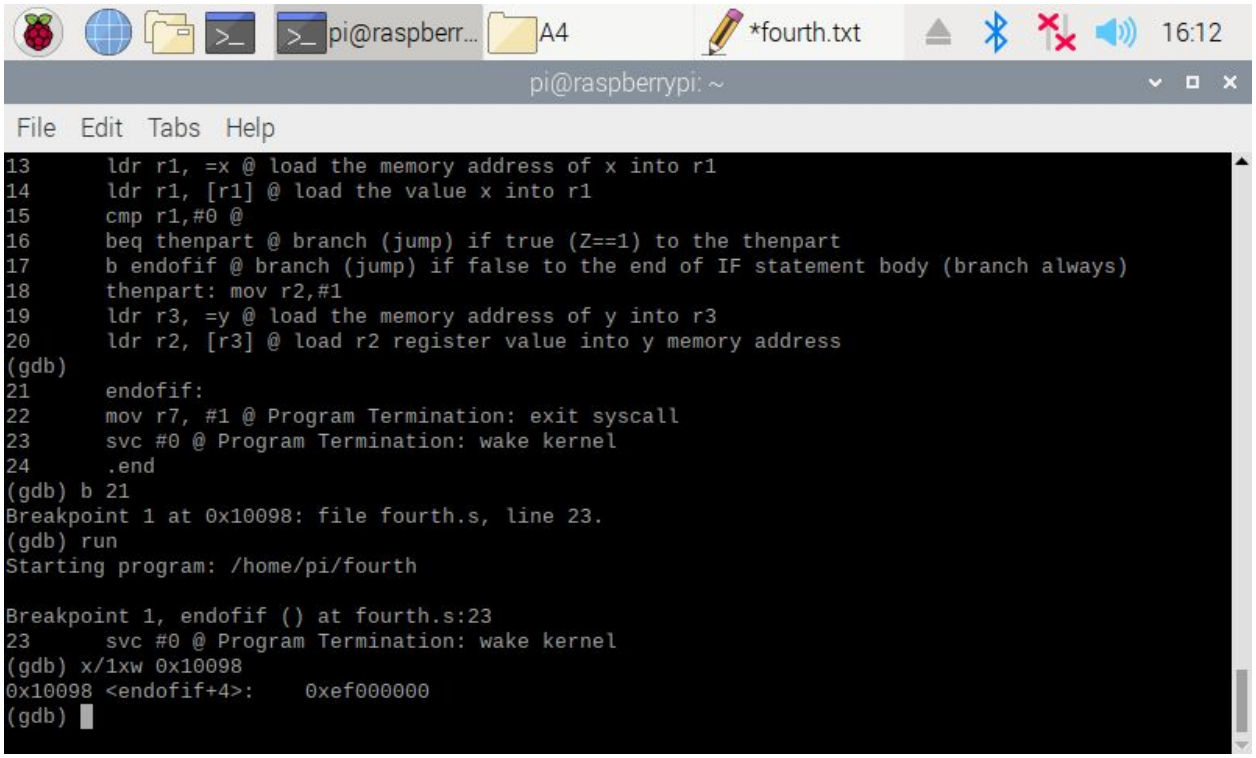


ARM Assembly Programming Miguel Romo:

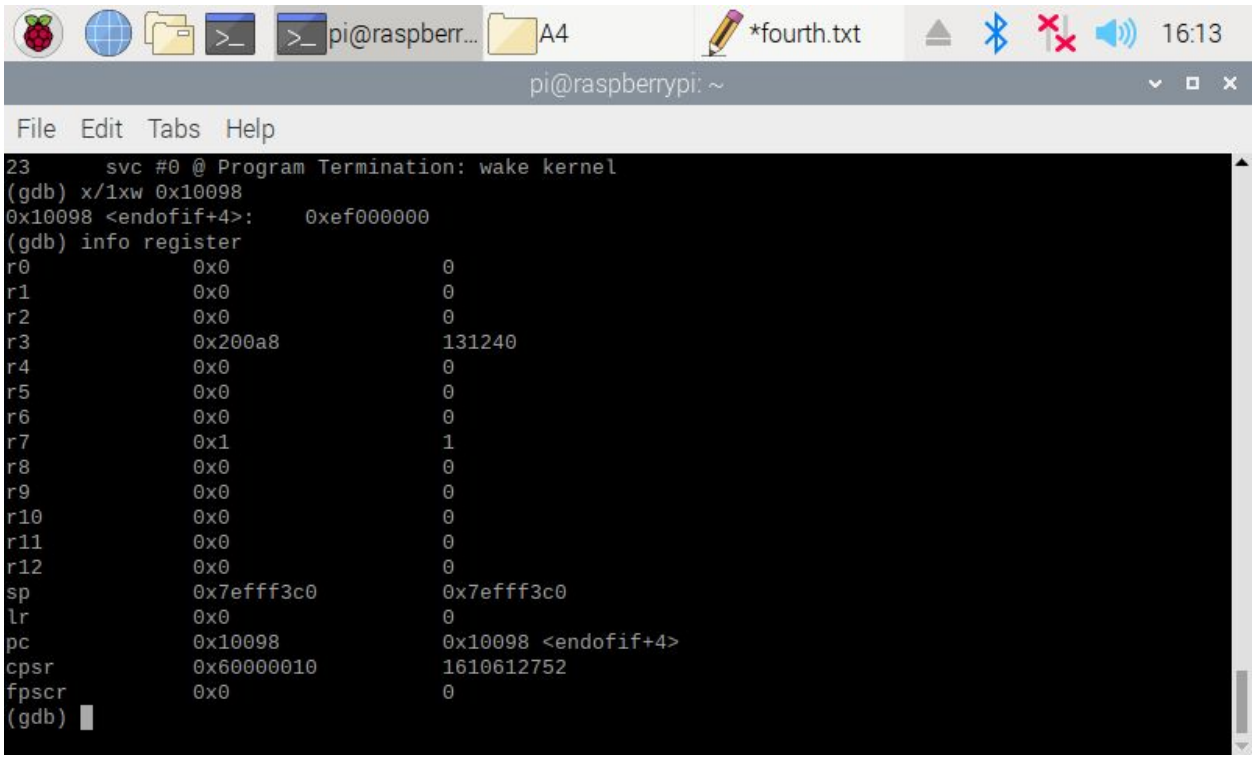
Part 1



Here is the code for Fourth program with beq and a second b instruction for a second brach.

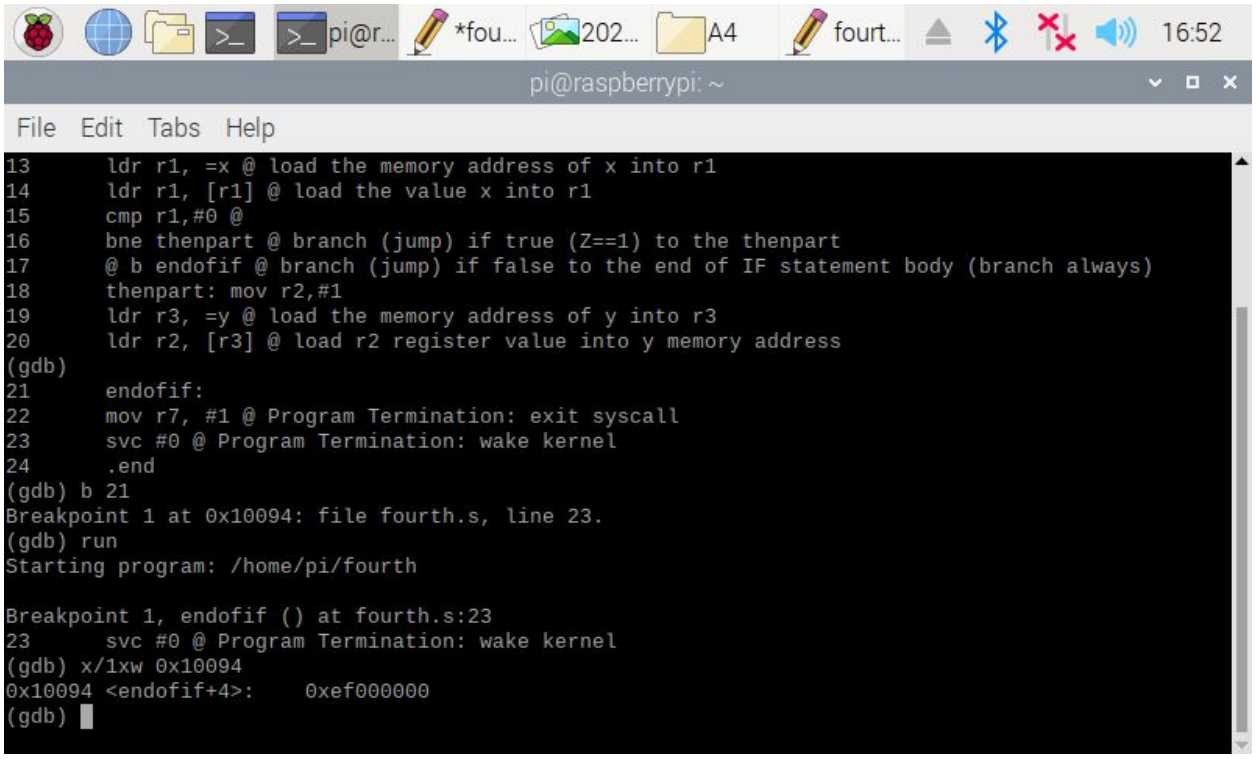


In this screen shot we run the program in debug mode and set a breakpoint on line 21. We also examine the memory address, which is 0xef000000.

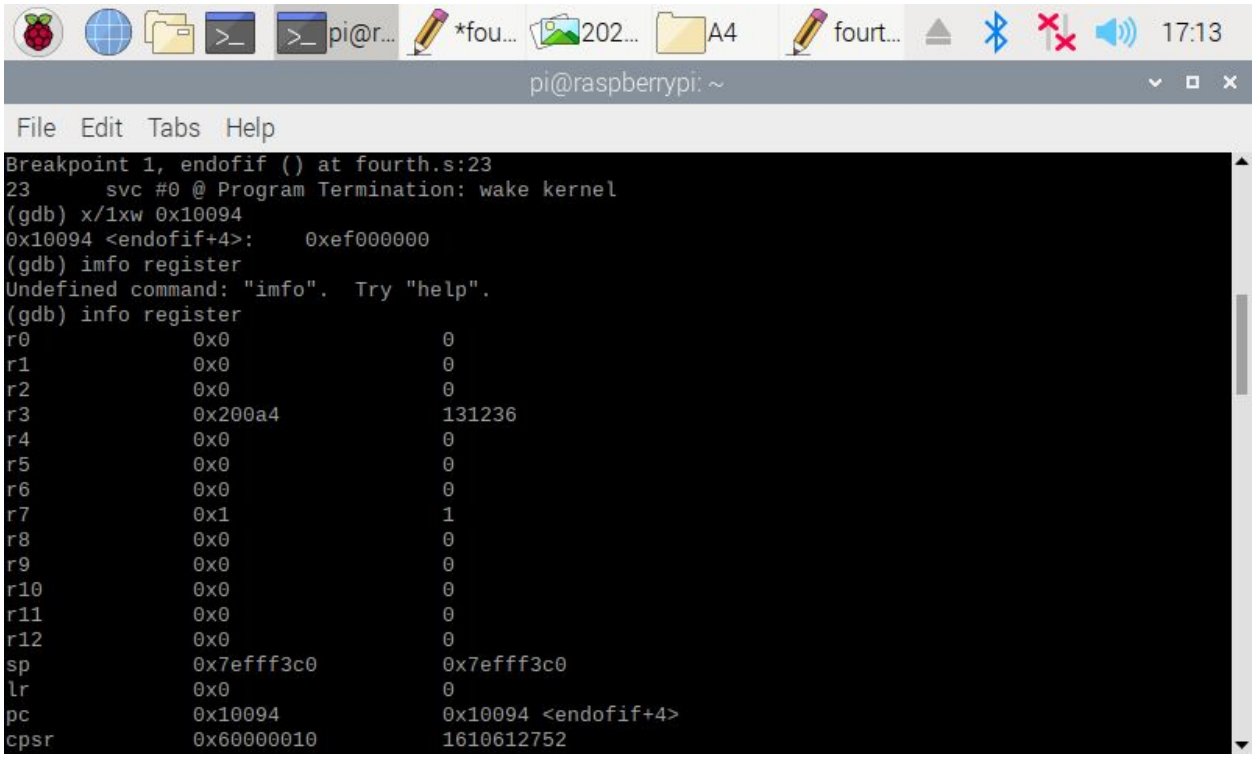


We check the registers and the Z flag value. The Z flag value = 0x60000010.

Part 2

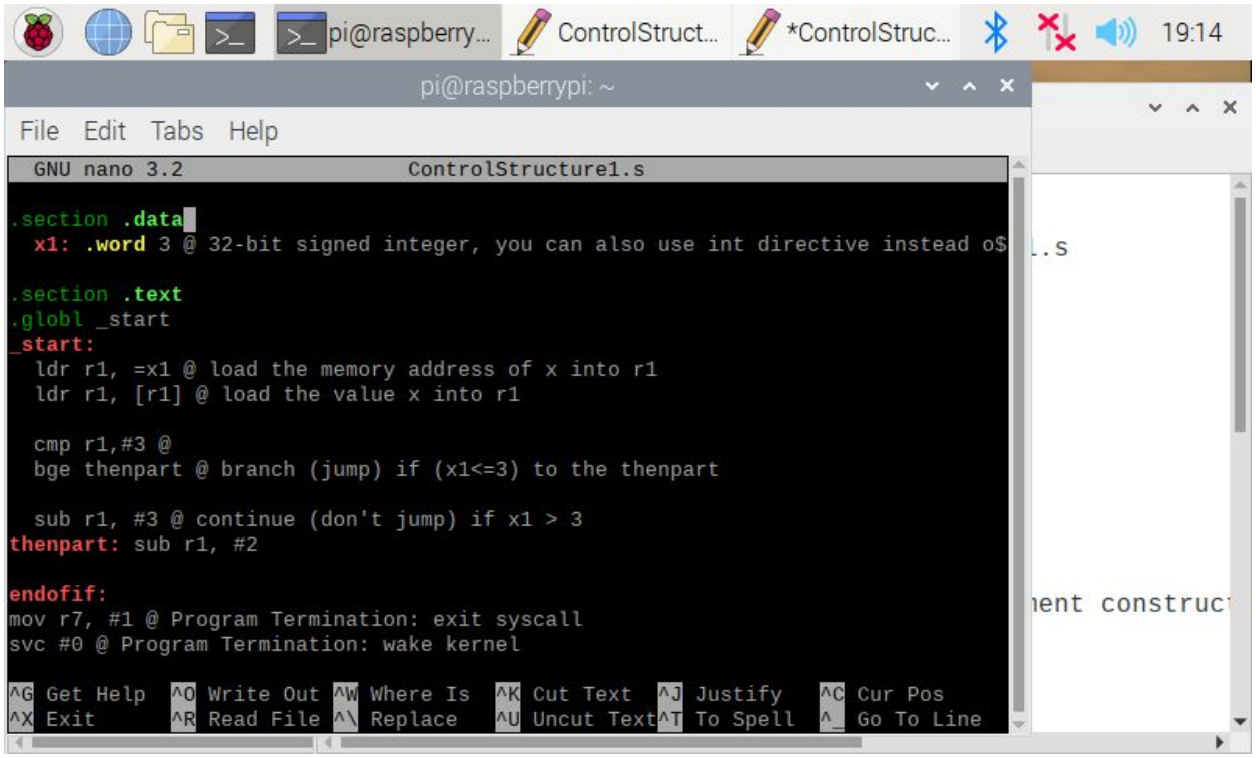


In part 2 of the ARM Assembly we make the Fourth program more efficient by replacing beq with bne and removing the back-to-back branch by commenting out line 17, the b instruction.



After updating the program, we assemble, link, run, and debug the code. We set a breakpoint on line 23 and examine the memory. The memory address is 0xef000000.

Part 3



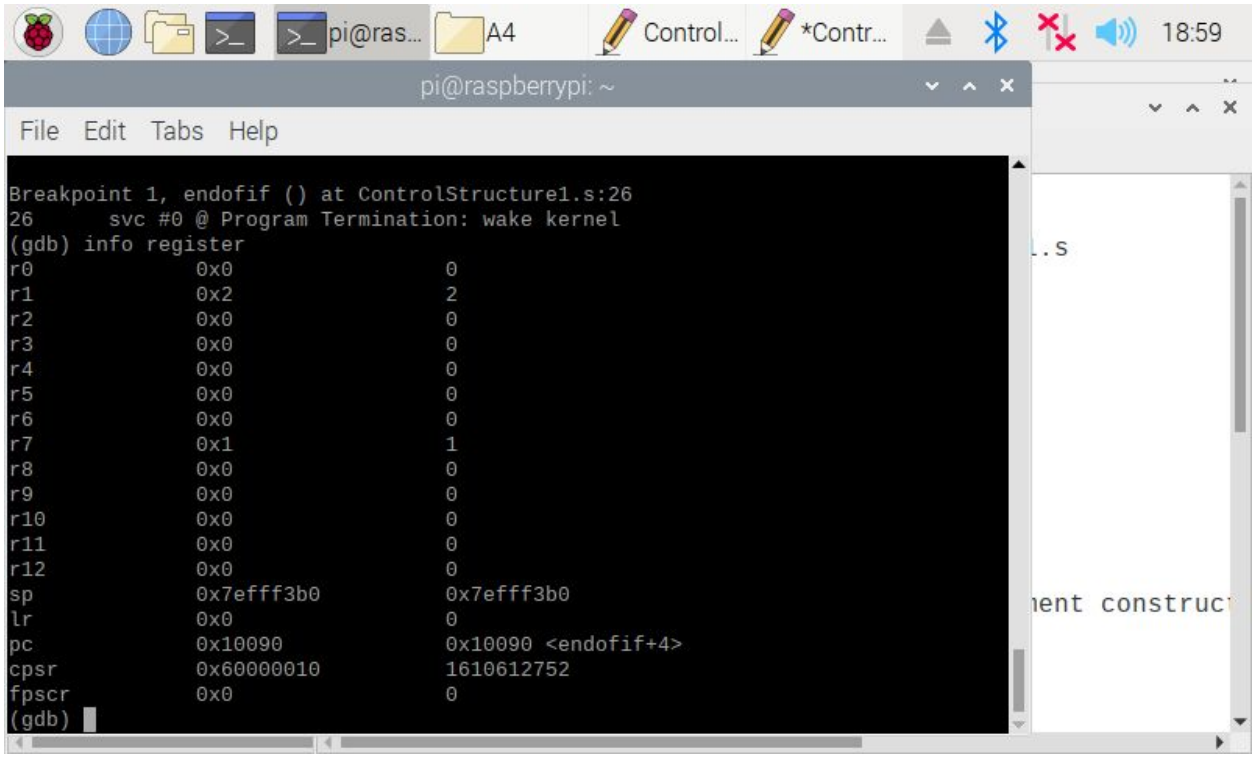
Here is the program for ControlStructure1. Base on the Note from the ARM Assembly document, you should use DE Morgan's Law, and the code should look like

bgt thenpart

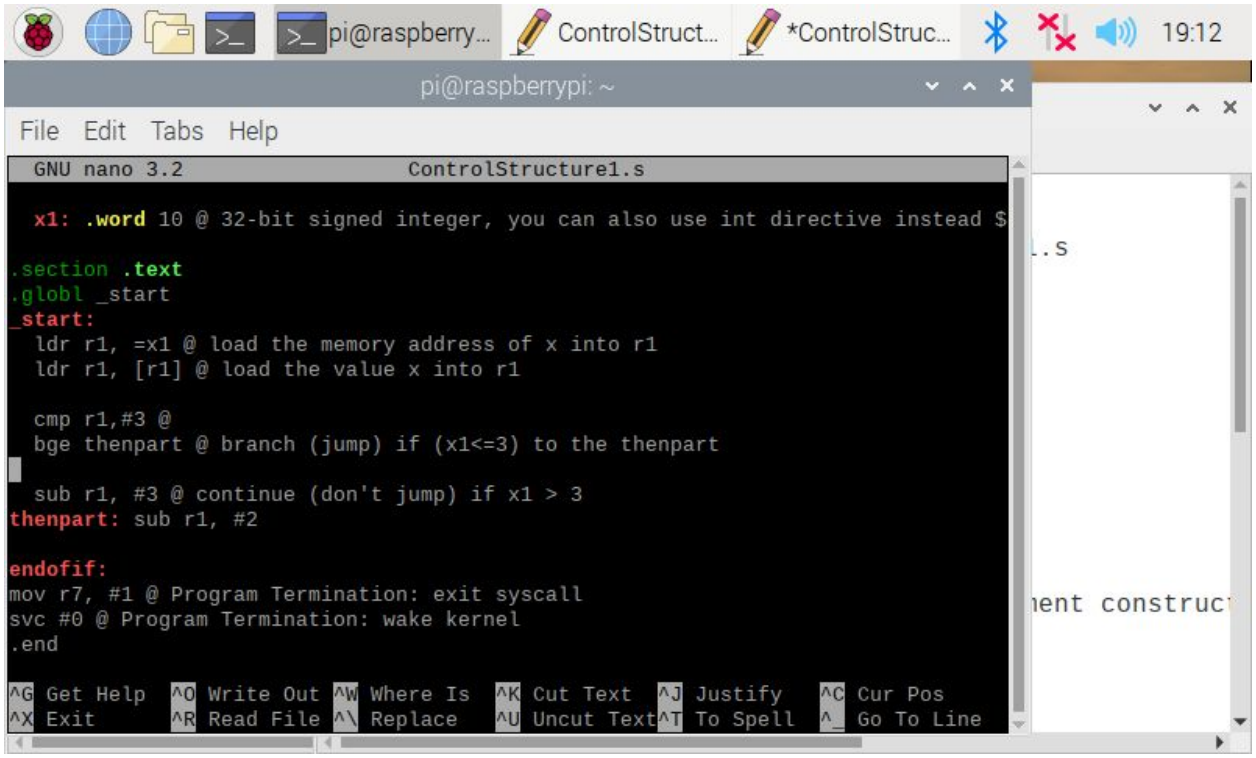
sub r1, #2

thenpart: sub r1, #2

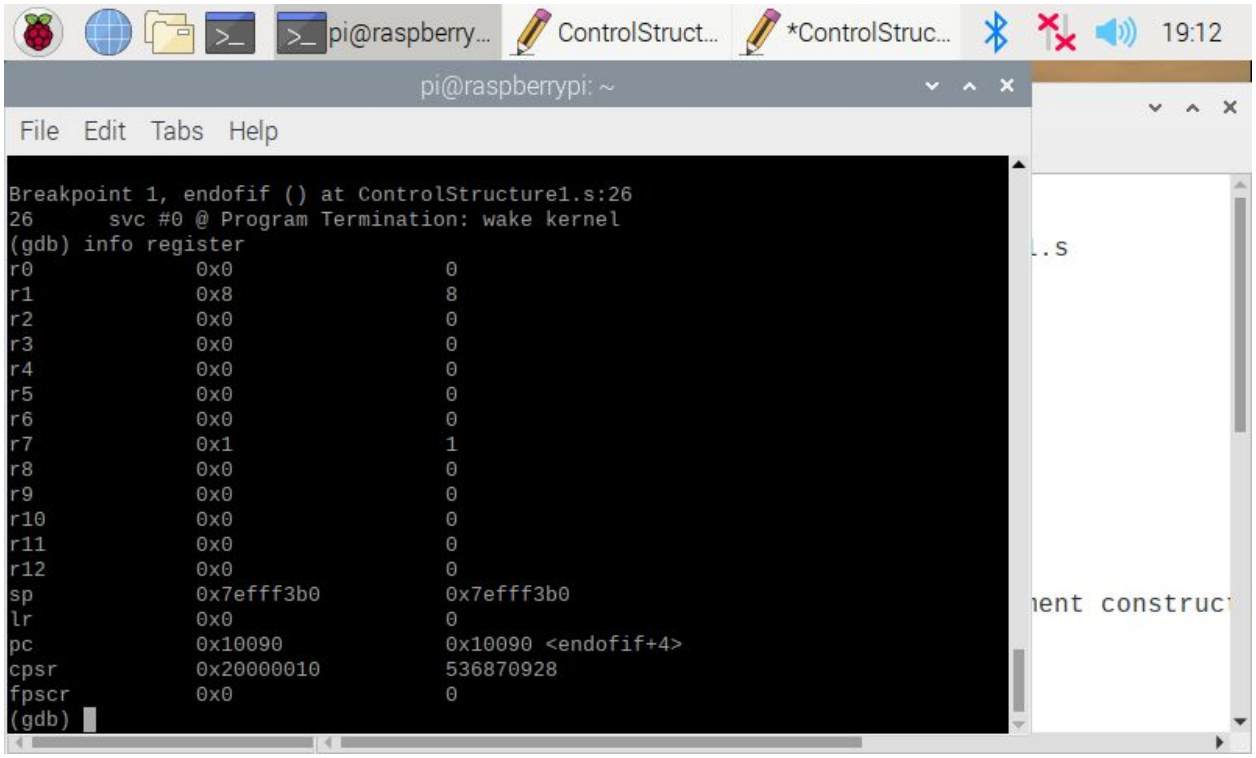
But this code does not produce the correct results. So I used bge, which gave me the correct values in the register each time I tested it with a new value of x.



In this test x1 = 3. I expected the value in r1 to be equal to 2, and that is what I got. You can all see the z flag value is 0x60000010.



In this screenshot, x1 = 10. So I expect the value of register r1 to equal 8.



After running the program and debugging it, we check the register and see r1 has the value of 8, which is what I expected. The Z flag = 0x20000010.

Appendix:

Slack:https://app.slack.com/client/TSWLWS9LK/CT8581ZU0

Youtube: https://youtu.be/Qw_G9VnX2kA

Github:https://github.com/Arteenghafourikia/CSC3210-TheCommuters