LAB-Report

Submitted by: - Parth Parashar

I had a really great experience working with these exercises. While the exercises were a little hard, but they stayed true to the concepts and it enhanced my knowledge in parallel programming with pThreads.

The first few exercises with the basic pthreads helped me build my practical knowledge with pthreads. I really loved the way pthreads handle the duties.

Conditional variables and mutexes might be tricky to get right at times but overall, if you can get a hang of it, you can grasp them easily.

I have never worked with barrier synchronization techniques and while the concept in itself is very fascinating, implementing them in programs can be a little hard when we do it for the first time. But it is great way to manage threads to thread communication. I found it very interesting. The way barrier threads reduce the overhead of joining the threads and increases performance speaks volumes for its effectiveness.

The most enjoyable part of this lab assignment though was the producer consumer problem which helped me to implement my knowledge of pthreads, conditional variables to real world problems.

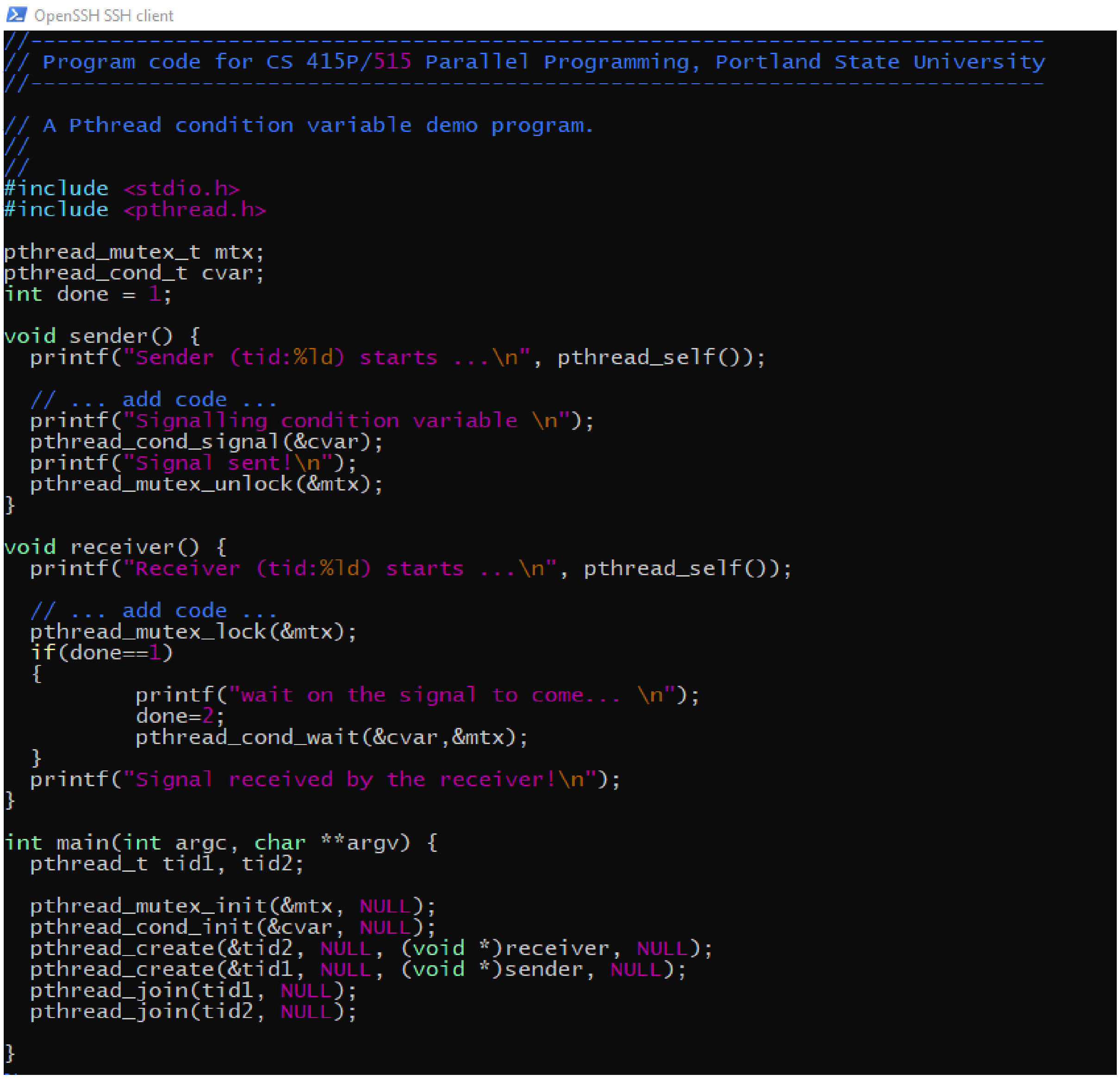
All in all, I had to work hard to get the answers but it gave me an insight as to how we can program using pthreads and how to think when working with pthreads. Sometimes I had to try different ways to incorporate the pthread programs but it is a great learning experience

I have attached the lab-experiments from the next page along with its answers.

Lab-Experiments (Condition-Variables)

Exercise-1): - Modifying and running the condvar-pthd.c program

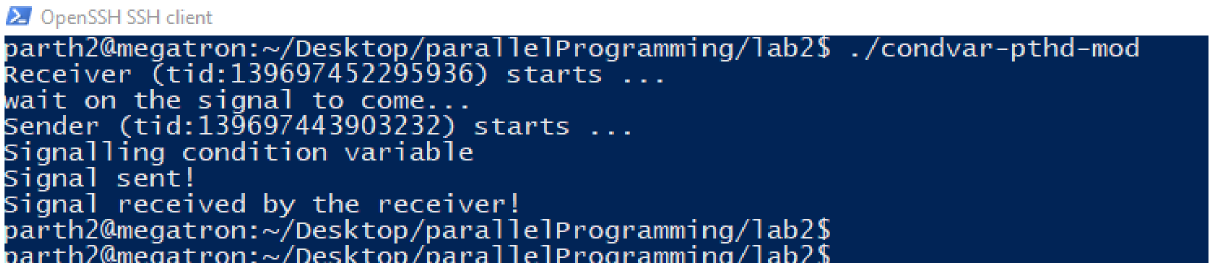
Modifications for the code



OpenSSH SSH client

```
//----------------------------------------------------------------------------
// Program code for CS 415P/515 Parallel Programming, Portland State University
//----------------------------------------------------------------------------

// A Pthread condition variable demo program.
//
//
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mtx;
pthread_cond_t cvar;
int done = 1;

void sender() {
  printf("Sender (tid:%ld) starts ...\n", pthread_self());

  // ... add code ...
  printf("Signalling condition variable \n");
  pthread_cond_signal(&cvar);
  printf("Signal sent!\n");
  pthread_mutex_unlock(&mtx);
}

void receiver() {
  printf("Receiver (tid:%ld) starts ...\n", pthread_self());

  // ... add code ...
  pthread_mutex_lock(&mtx);
  if(done==1)
  {
          printf("wait on the signal to come... \n");
          done=2;
          pthread_cond_wait(&cvar,&mtx);
  }
  printf("Signal received by the receiver!\n");
}

int main(int argc, char **argv) {
  pthread_t tid1, tid2;

  pthread_mutex_init(&mtx, NULL);
  pthread_cond_init(&cvar, NULL);
  pthread_create(&tid2, NULL, (void *)receiver, NULL);
  pthread_create(&tid1, NULL, (void *)sender, NULL);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);

}
```

When compiled, the code exhibits the following behaviour on the console: -

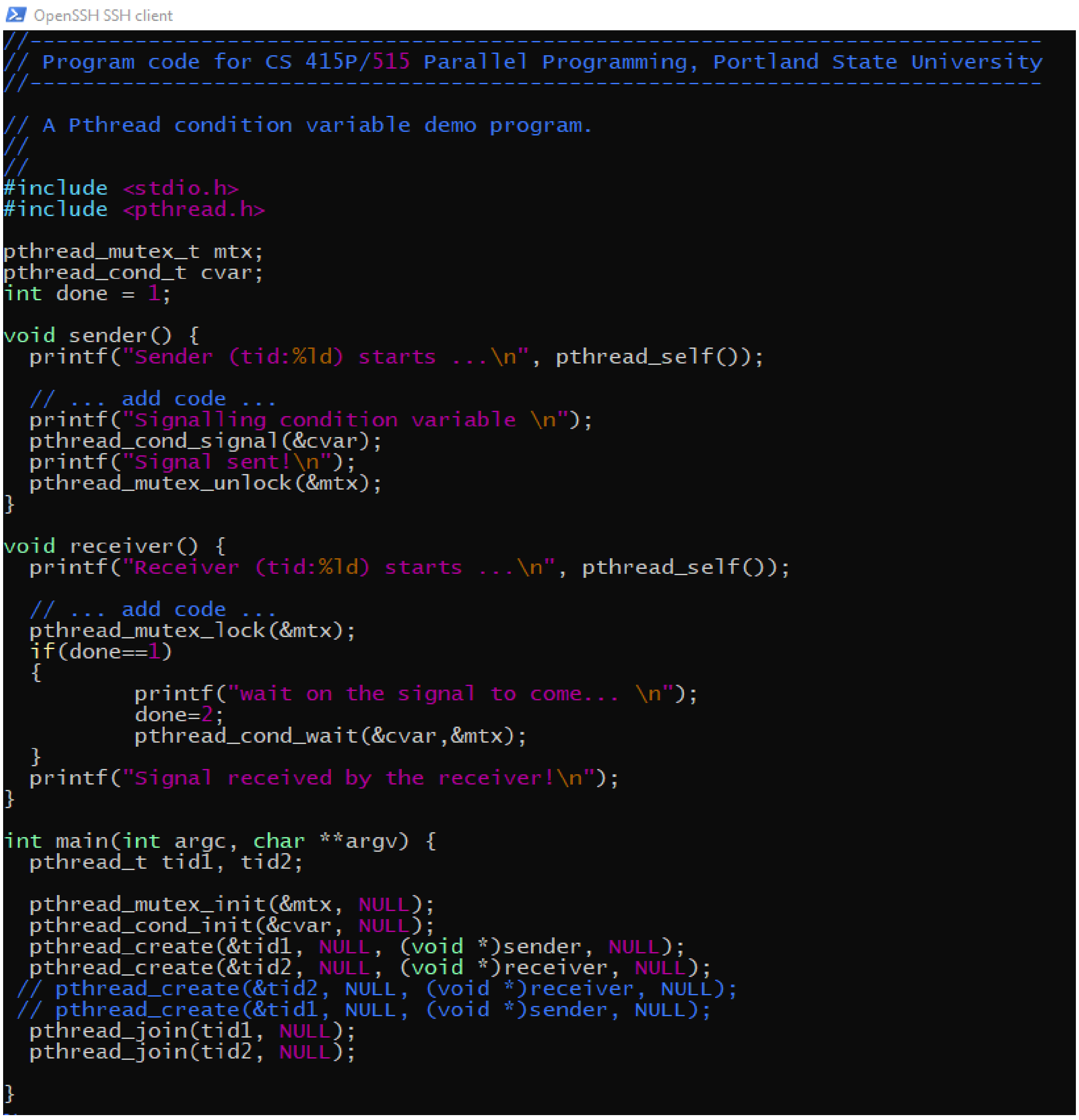


OpenSSH SSH client

```
parth2@megatron:~/Desktop/parallelProgramming/lab2$ ./condvar-pthd-mod
Receiver (tid:139697452295936) starts ...
wait on the signal to come...
Sender (tid:139697443903232) starts ...
Signalling condition variable
Signal sent!
Signal received by the receiver!
parth2@megatron:~/Desktop/parallelProgramming/lab2$
parth2@megatron:~/Desktop/parallelProgramming/lab2$
```

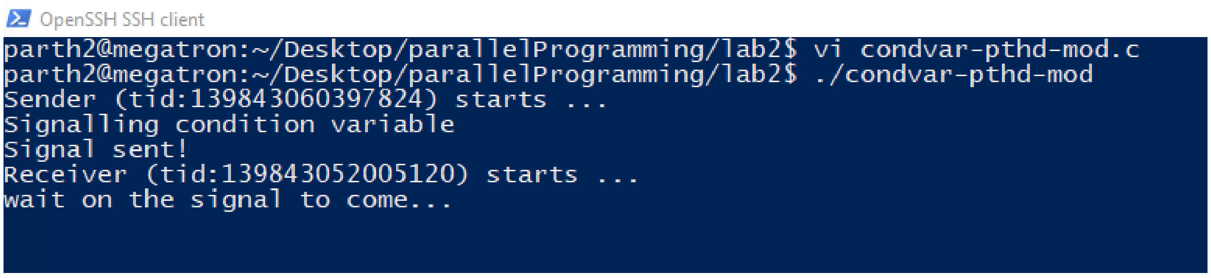
Exercise-2): - When the two create statements are swapped, then the program enters an infinite wait state. This is because the signal is sent first and then the wait condition is invoked. This means that there is no signal for the wait and the wait conditions remains true forever.

The changed code is displayed below: -



```c
//----------------------------------------------------------------------------
// Program code for CS 415P/515 Parallel Programming, Portland State University
//----------------------------------------------------------------------------

// A Pthread condition variable demo program.
//
//
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mtx;
pthread_cond_t cvar;
int done = 1;

void sender() {
  printf("Sender (tid:%ld) starts ...\n", pthread_self());

  // ... add code ...
  printf("Signalling condition variable \n");
  pthread_cond_signal(&cvar);
  printf("Signal sent!\n");
  pthread_mutex_unlock(&mtx);
}

void receiver() {
  printf("Receiver (tid:%ld) starts ...\n", pthread_self());

  // ... add code ...
  pthread_mutex_lock(&mtx);
  if(done==1)
  {
          printf("wait on the signal to come... \n");
          done=2;
          pthread_cond_wait(&cvar,&mtx);
  }
  printf("Signal received by the receiver!\n");
}

int main(int argc, char **argv) {
  pthread_t tid1, tid2;

  pthread_mutex_init(&mtx, NULL);
  pthread_cond_init(&cvar, NULL);
  pthread_create(&tid1, NULL, (void *)sender, NULL);
  pthread_create(&tid2, NULL, (void *)receiver, NULL);
 // pthread_create(&tid2, NULL, (void *)receiver, NULL);
 // pthread_create(&tid1, NULL, (void *)sender, NULL);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);

}
```

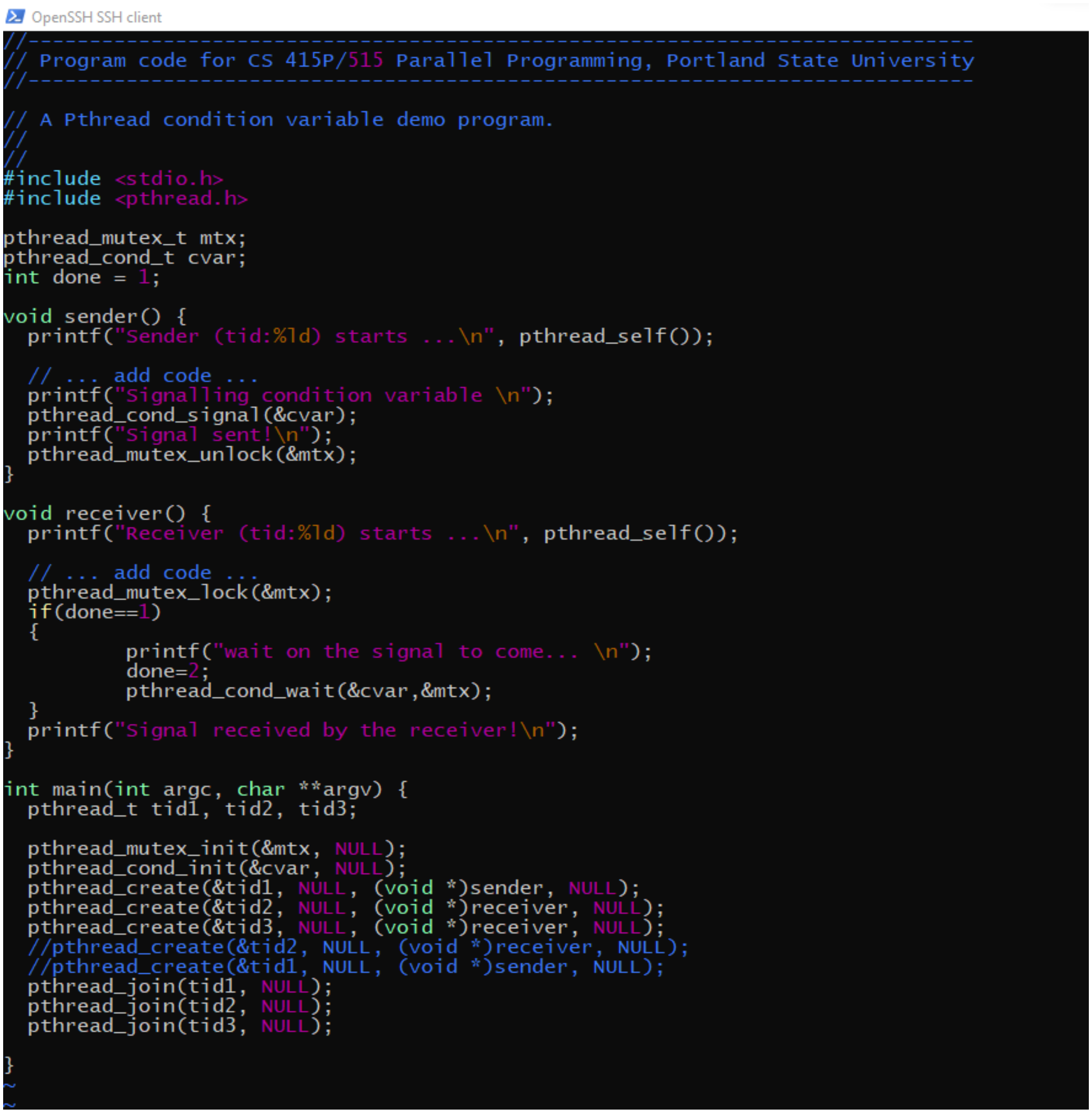
The output for the following code is given below: -



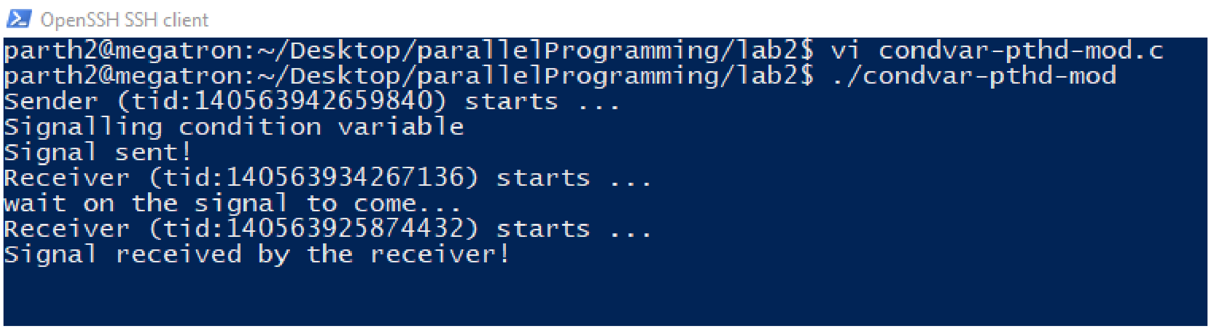
```
parth2@megatron:~/Desktop/parallelProgramming/lab2$ vi condvar-pthd-mod.c
parth2@megatron:~/Desktop/parallelProgramming/lab2$ ./condvar-pthd-mod
Sender (tid:139843060397824) starts ...
Signalling condition variable
Signal sent!
Receiver (tid:139843052005120) starts ...
wait on the signal to come...
```

Exercise-3): - When the program is compiled and run, then one signal is received properly by the receiver but since there is only one signal sent, other receiver is waiting for the signal to come.

This can be seen from the screenshots of the program below: -



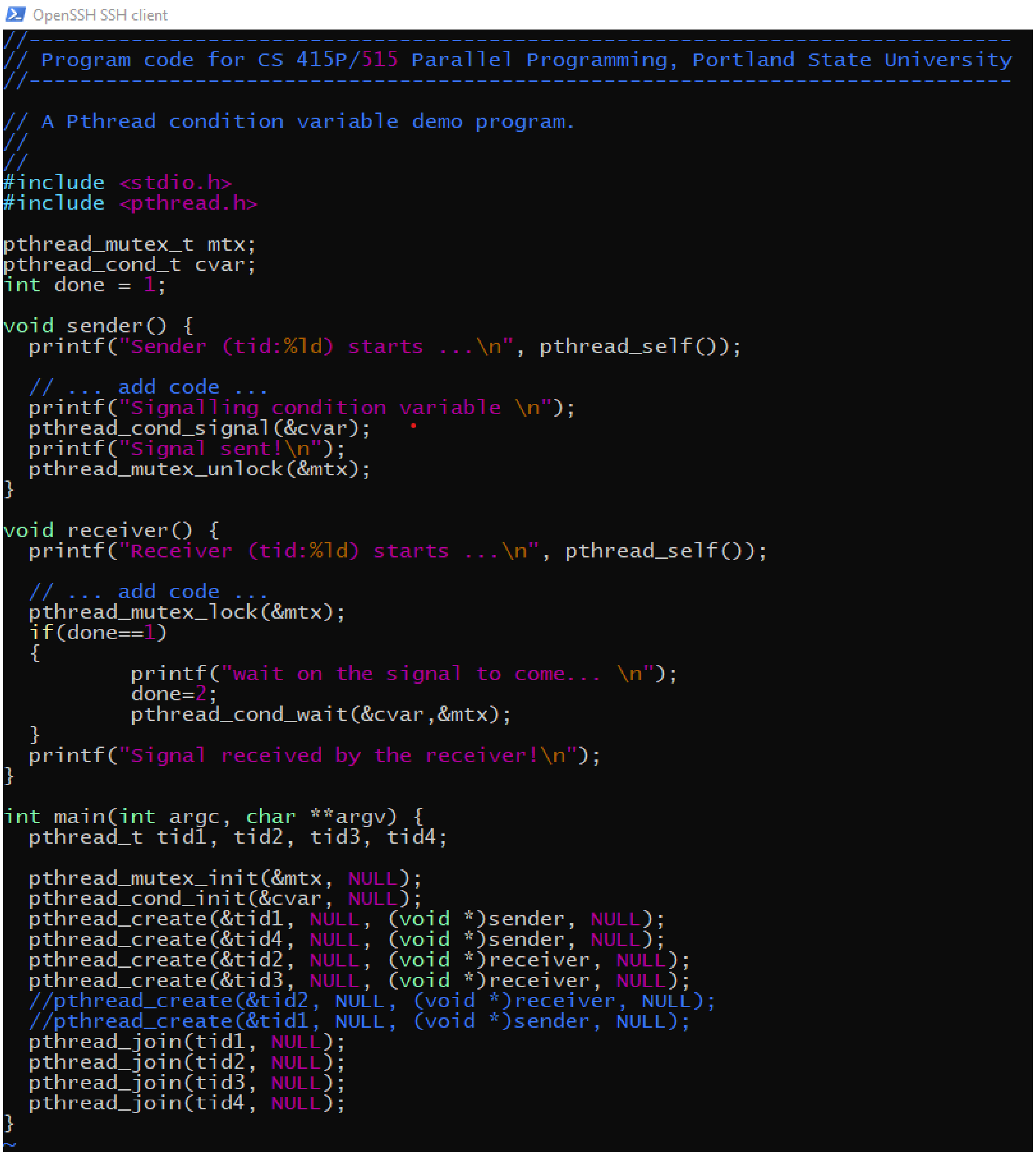
```
//--------------------------------------------------------------------------------
// Program code for CS 415P/515 Parallel Programming, Portland State University
//--------------------------------------------------------------------------------

// A Pthread condition variable demo program.
//
//
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mtx;
pthread_cond_t cvar;
int done = 1;

void sender() {
  printf("Sender (tid:%ld) starts ...\n", pthread_self());

  // ... add code ...
  printf("Signalling condition variable \n");
  pthread_cond_signal(&cvar);
  printf("Signal sent!\n");
  pthread_mutex_unlock(&mtx);
}

void receiver() {
  printf("Receiver (tid:%ld) starts ...\n", pthread_self());

  // ... add code ...
  pthread_mutex_lock(&mtx);
  if(done==1)
  {
          printf("wait on the signal to come... \n");
          done=2;
          pthread_cond_wait(&cvar,&mtx);
  }
  printf("Signal received by the receiver!\n");
}

int main(int argc, char **argv) {
  pthread_t tid1, tid2, tid3;

  pthread_mutex_init(&mtx, NULL);
  pthread_cond_init(&cvar, NULL);
  pthread_create(&tid1, NULL, (void *)sender, NULL);
  pthread_create(&tid2, NULL, (void *)receiver, NULL);
  pthread_create(&tid3, NULL, (void *)receiver, NULL);
  //pthread_create(&tid2, NULL, (void *)receiver, NULL);
  //pthread_create(&tid1, NULL, (void *)sender, NULL);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  pthread_join(tid3, NULL);

}
~
~
```



OpenSSH SSH client

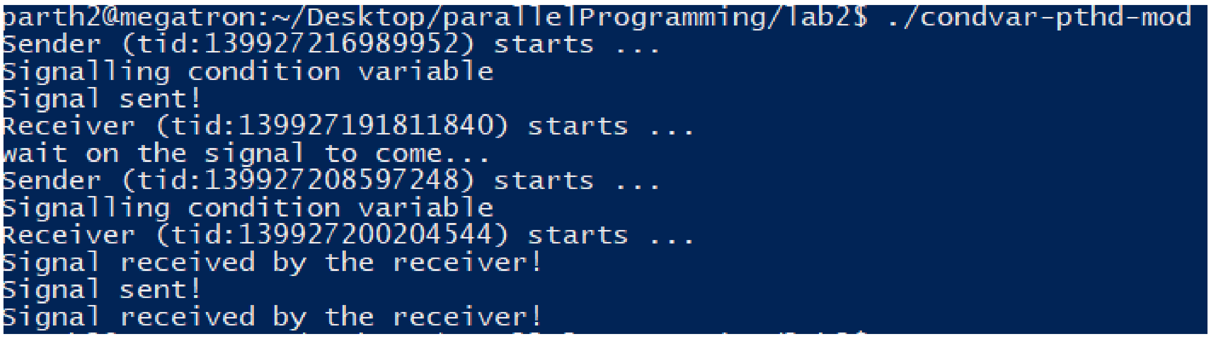
```
parth2@megatron:~/Desktop/parallelProgramming/lab2$ vi condvar-pthd-mod.c
parth2@megatron:~/Desktop/parallelProgramming/lab2$ ./condvar-pthd-mod
Sender (tid:140563942659840) starts ...
Signalling condition variable
Signal sent!
Receiver (tid:140563934267136) starts ...
wait on the signal to come...
Receiver (tid:140563925874432) starts ...
Signal received by the receiver!
```

This situation can be rectified using another sender as given below: -



OpenSSH SSH client

```
//------------------------------------------------------------------------------
// Program code for CS 415P/515 Parallel Programming, Portland State University
//------------------------------------------------------------------------------

// A Pthread condition variable demo program.
//
//
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mtx;
pthread_cond_t cvar;
int done = 1;

void sender() {
  printf("Sender (tid:%ld) starts ...\n", pthread_self());

  // ... add code ...
  printf("Signalling condition variable \n");
  pthread_cond_signal(&cvar);
  printf("Signal sent!\n");
  pthread_mutex_unlock(&mtx);
}

void receiver() {
  printf("Receiver (tid:%ld) starts ...\n", pthread_self());

  // ... add code ...
  pthread_mutex_lock(&mtx);
  if(done==1)
  {
          printf("wait on the signal to come... \n");
          done=2;
          pthread_cond_wait(&cvar,&mtx);
  }
  printf("Signal received by the receiver!\n");
}

int main(int argc, char **argv) {
  pthread_t tid1, tid2, tid3, tid4;

  pthread_mutex_init(&mtx, NULL);
  pthread_cond_init(&cvar, NULL);
  pthread_create(&tid1, NULL, (void *)sender, NULL);
  pthread_create(&tid4, NULL, (void *)sender, NULL);
  pthread_create(&tid2, NULL, (void *)receiver, NULL);
  pthread_create(&tid3, NULL, (void *)receiver, NULL);
  //pthread_create(&tid2, NULL, (void *)receiver, NULL);
  //pthread_create(&tid1, NULL, (void *)sender, NULL);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  pthread_join(tid3, NULL);
  pthread_join(tid4, NULL);
}
~
```



```
parth2@megatron:~/Desktop/parallelProgramming/lab2$ ./condvar-pthd-mod
Sender (tid:139927216989952) starts ...
Signalling condition variable
Signal sent!
Receiver (tid:139927191811840) starts ...
wait on the signal to come...
Sender (tid:139927208597248) starts ...
Signalling condition variable
Receiver (tid:139927200204544) starts ...
Signal received by the receiver!
Signal sent!
Signal received by the receiver!
```

But this also does not guarantee that the program gives the definite results as there are chances where the thread library sends the signal first and then receiver keeps on waiting for the signal to come.

Hence, by changing the order to

Receiver-1

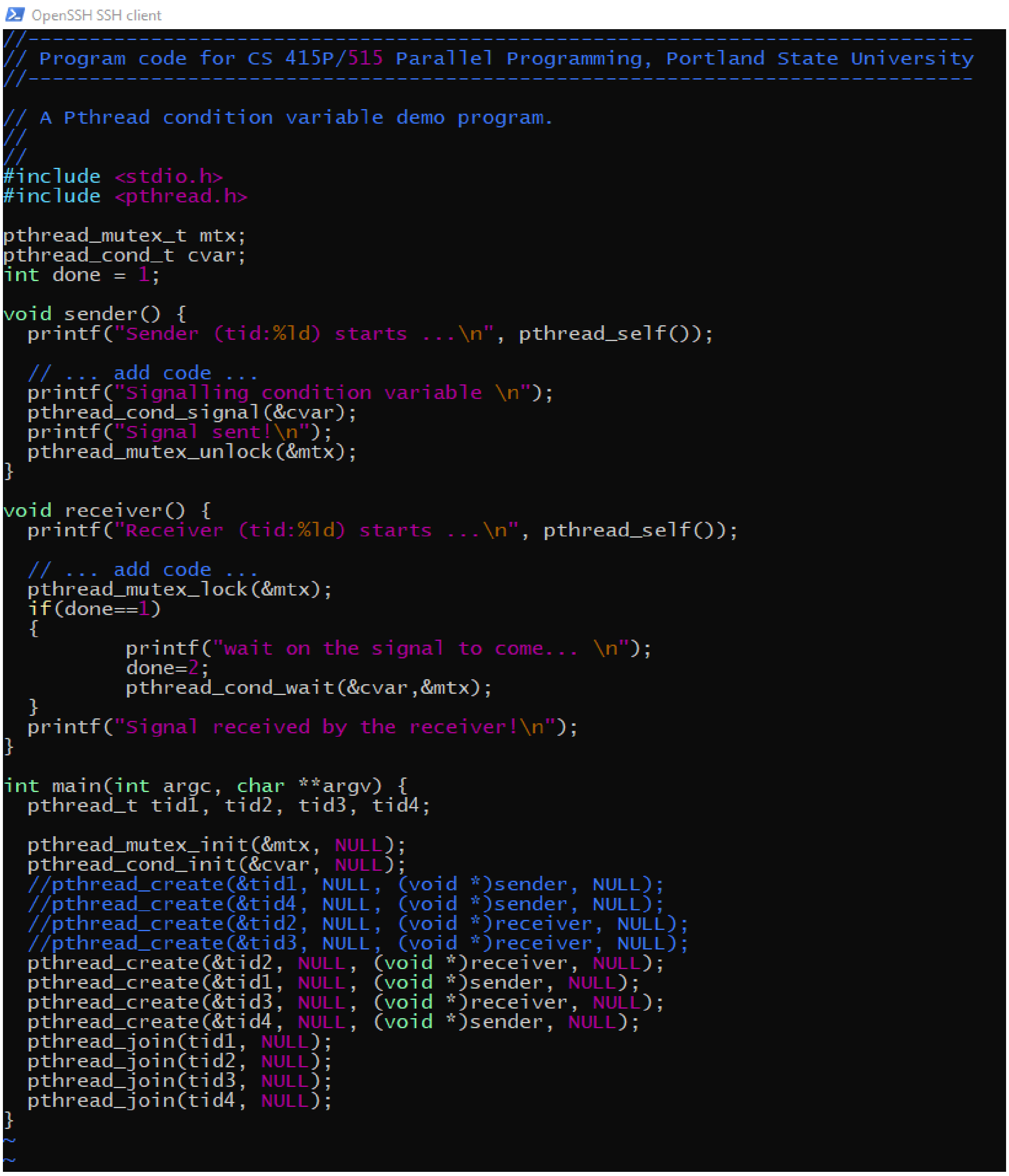
Sender-1

Receiver-2

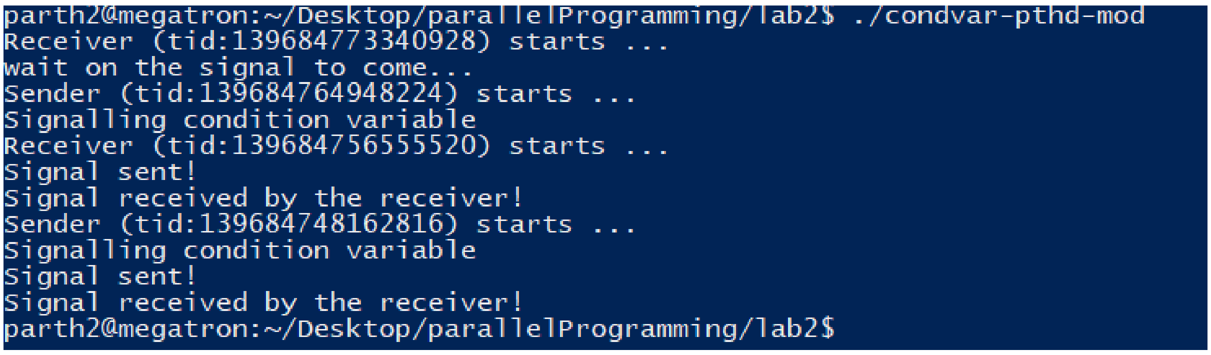
Sender-2

We can ensure that each receiver is in wait state and each sender sends the signal at the right time for the receiver to receive and perform the correct tasks as required.

The program for the same is given below: -


OpenSSH SSH client

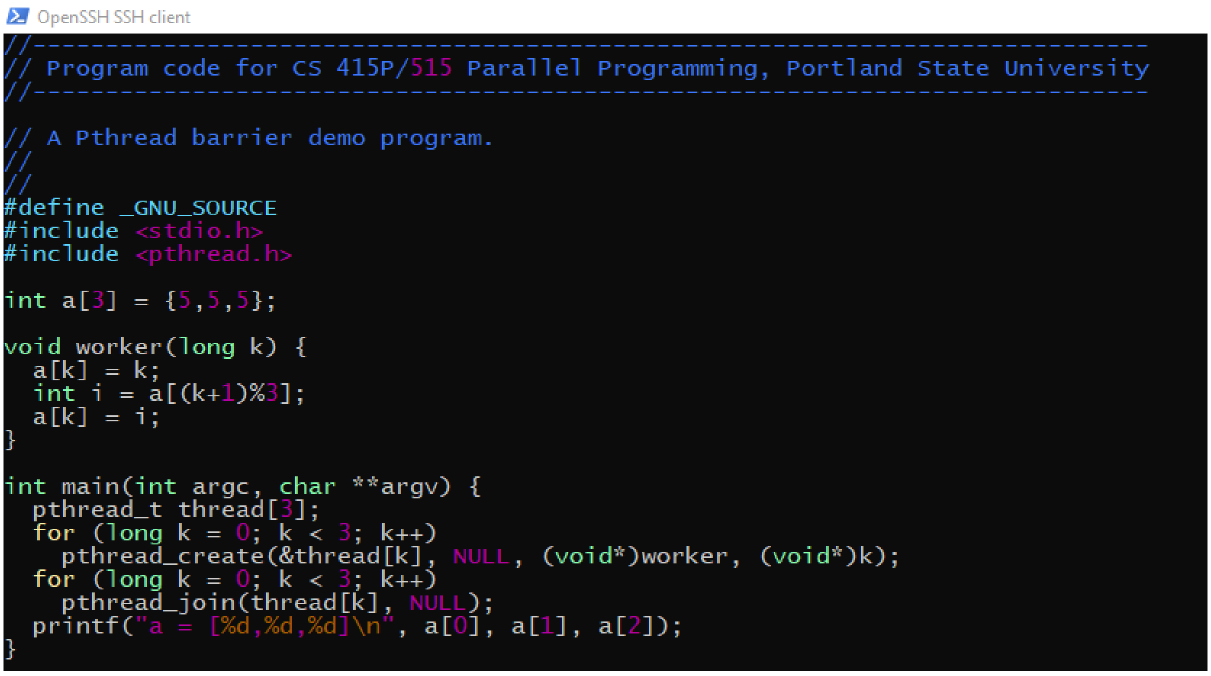
```c
//--------------------------------------------------------------------------------
// Program code for CS 415P/515 Parallel Programming, Portland State University
//--------------------------------------------------------------------------------

// A Pthread condition variable demo program.
//
//
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mtx;
pthread_cond_t cvar;
int done = 1;

void sender() {
  printf("Sender (tid:%ld) starts ...\n", pthread_self());

  // ... add code ...
  printf("Signalling condition variable \n");
  pthread_cond_signal(&cvar);
  printf("Signal sent!\n");
  pthread_mutex_unlock(&mtx);
}

void receiver() {
  printf("Receiver (tid:%ld) starts ...\n", pthread_self());

  // ... add code ...
  pthread_mutex_lock(&mtx);
  if(done==1)
  {
          printf("wait on the signal to come... \n");
          done=2;
          pthread_cond_wait(&cvar,&mtx);
  }
  printf("Signal received by the receiver!\n");
}

int main(int argc, char **argv) {
  pthread_t tid1, tid2, tid3, tid4;

  pthread_mutex_init(&mtx, NULL);
  pthread_cond_init(&cvar, NULL);
  //pthread_create(&tid1, NULL, (void *)sender, NULL);
  //pthread_create(&tid4, NULL, (void *)sender, NULL);
  //pthread_create(&tid2, NULL, (void *)receiver, NULL);
  //pthread_create(&tid3, NULL, (void *)receiver, NULL);
  pthread_create(&tid2, NULL, (void *)receiver, NULL);
  pthread_create(&tid1, NULL, (void *)sender, NULL);
  pthread_create(&tid3, NULL, (void *)receiver, NULL);
  pthread_create(&tid4, NULL, (void *)sender, NULL);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  pthread_join(tid3, NULL);
  pthread_join(tid4, NULL);
}
~
~
```



```
parth2@megatron:~/Desktop/parallelProgramming/lab2$ ./condvar-pthd-mod
Receiver (tid:139684773340928) starts ...
wait on the signal to come...
Sender (tid:139684764948224) starts ...
Signalling condition variable
Receiver (tid:139684756555520) starts ...
Signal sent!
Signal received by the receiver!
Sender (tid:139684748162816) starts ...
Signalling condition variable
Signal sent!
Signal received by the receiver!
parth2@megatron:~/Desktop/parallelProgramming/lab2$
```

Lab Experiments (Barrier Synchronization)

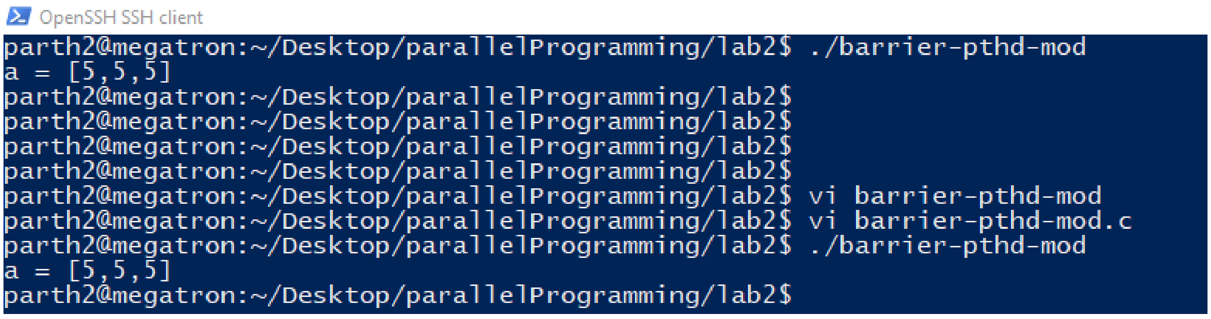
Exercise-1): - The given code is: -



```
//----------------------------------------------------------------------------
// Program code for CS 415P/515 Parallel Programming, Portland State University
//----------------------------------------------------------------------------

// A Pthread barrier demo program.
//
//
#define _GNU_SOURCE
#include <stdio.h>
#include <pthread.h>

int a[3] = {5,5,5};

void worker(long k) {
  a[k] = k;
  int i = a[(k+1)%3];
  a[k] = i;
}

int main(int argc, char **argv) {
  pthread_t thread[3];
  for (long k = 0; k < 3; k++)
    pthread_create(&thread[k], NULL, (void*)worker, (void*)k);
  for (long k = 0; k < 3; k++)
    pthread_join(thread[k], NULL);
  printf("a = [%d,%d,%d]\n", a[0], a[1], a[2]);
}
```

The code is expected to print the values of a[0], a[1], a[2] but should have been done by the threads but this cannot be guaranteed as there is no communication between the threads. There is also no guarantee that there would not be any dangling threads i.e., no guarantee that the threads would complete the execution they were set to do and might not be killed.

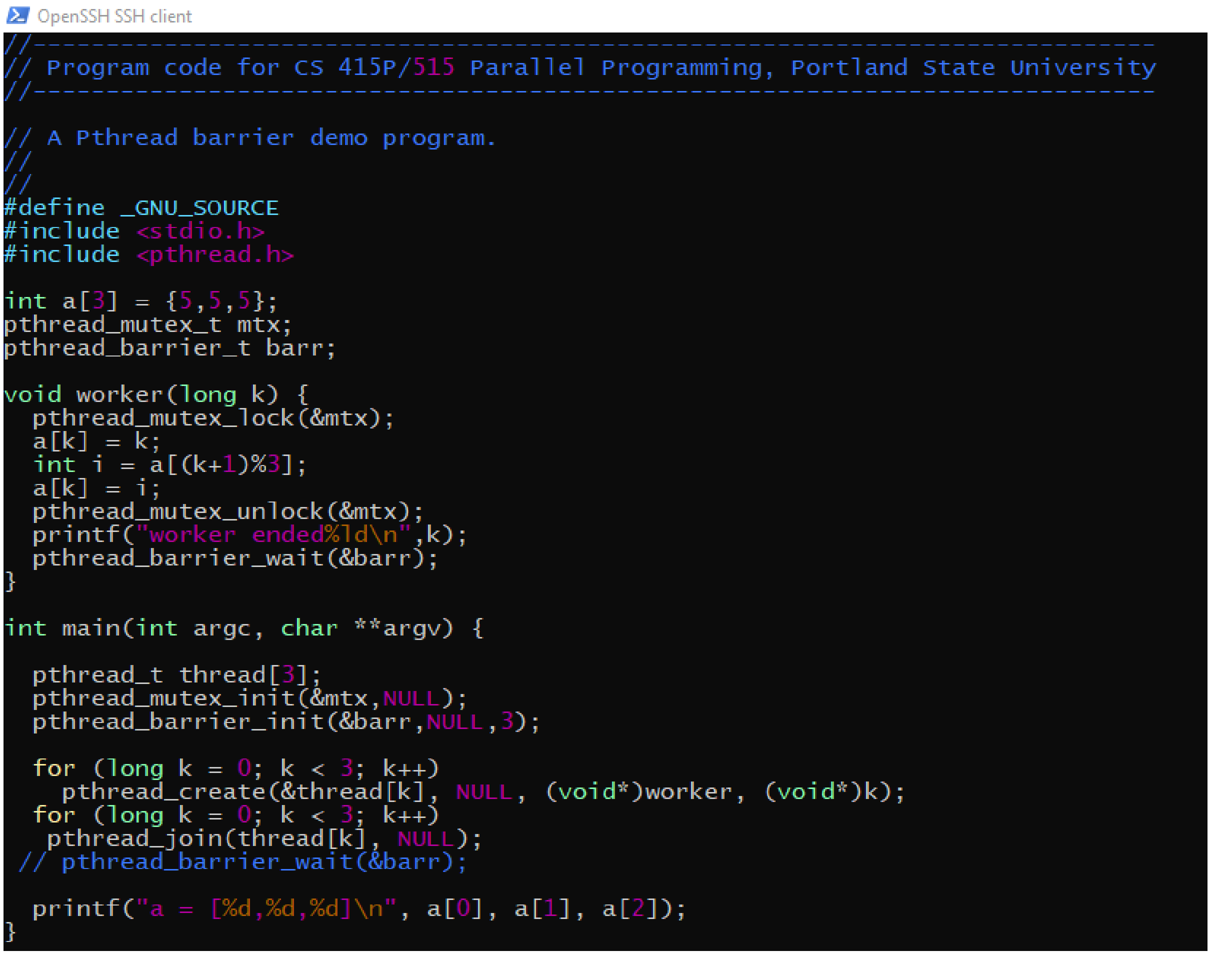
This can be verified by the output: -



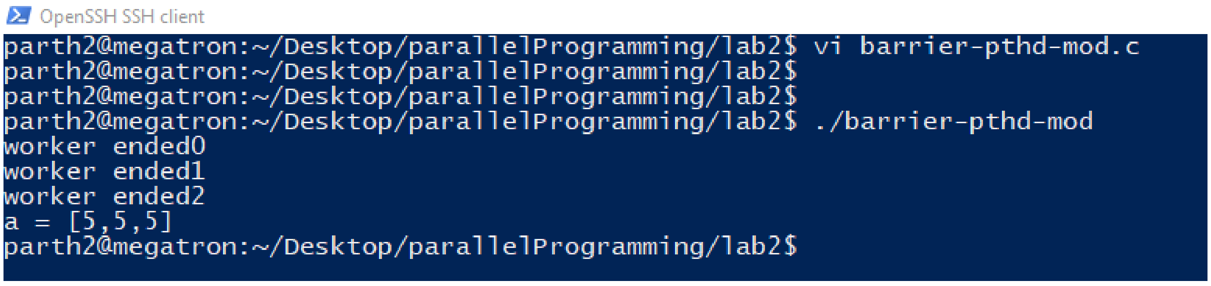
```
parth2@megatron:~/Desktop/parallelProgramming/lab2$ ./barrier-pthd-mod
a = [5,5,5]
parth2@megatron:~/Desktop/parallelProgramming/lab2$
parth2@megatron:~/Desktop/parallelProgramming/lab2$
parth2@megatron:~/Desktop/parallelProgramming/lab2$
parth2@megatron:~/Desktop/parallelProgramming/lab2$
parth2@megatron:~/Desktop/parallelProgramming/lab2$ vi barrier-pthd-mod
parth2@megatron:~/Desktop/parallelProgramming/lab2$ vi barrier-pthd-mod.c
parth2@megatron:~/Desktop/parallelProgramming/lab2$ ./barrier-pthd-mod
a = [5,5,5]
parth2@megatron:~/Desktop/parallelProgramming/lab2$
```

Exercise-2): -

Now, when we introduce barrier synchronization in this program, we get the following code and output: -


OpenSSH SSH client

```
//----------------------------------------------------------------------------
// Program code for CS 415P/515 Parallel Programming, Portland State University
//----------------------------------------------------------------------------

// A Pthread barrier demo program.
//
//
#define _GNU_SOURCE
#include <stdio.h>
#include <pthread.h>

int a[3] = {5,5,5};
pthread_mutex_t mtx;
pthread_barrier_t barr;

void worker(long k) {
  pthread_mutex_lock(&mtx);
  a[k] = k;
  int i = a[(k+1)%3];
  a[k] = i;
  pthread_mutex_unlock(&mtx);
  printf("worker ended%ld\n",k);
  pthread_barrier_wait(&barr);
}

int main(int argc, char **argv) {

  pthread_t thread[3];
  pthread_mutex_init(&mtx,NULL);
  pthread_barrier_init(&barr,NULL,3);

  for (long k = 0; k < 3; k++)
    pthread_create(&thread[k], NULL, (void*)worker, (void*)k);
  for (long k = 0; k < 3; k++)
   pthread_join(thread[k], NULL);
 // pthread_barrier_wait(&barr);

  printf("a = [%d,%d,%d]\n", a[0], a[1], a[2]);
}
```

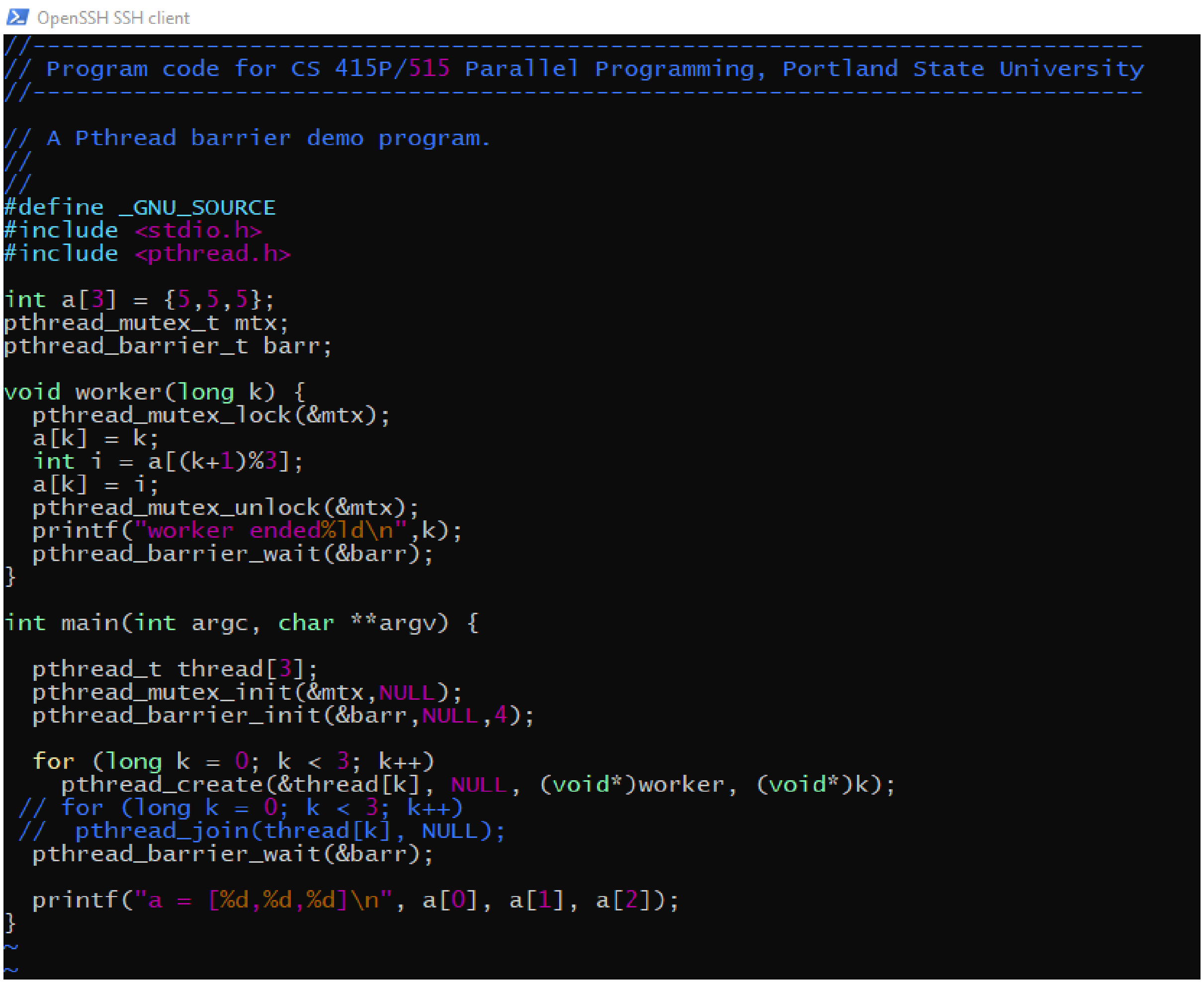

OpenSSH SSH client

```
parth2@megatron:~/Desktop/parallelProgramming/lab2$ vi barrier-pthd-mod.c
parth2@megatron:~/Desktop/parallelProgramming/lab2$
parth2@megatron:~/Desktop/parallelProgramming/lab2$
parth2@megatron:~/Desktop/parallelProgramming/lab2$ ./barrier-pthd-mod
worker ended0
worker ended1
worker ended2
a = [5,5,5]
parth2@megatron:~/Desktop/parallelProgramming/lab2$
```

The output is as expected.

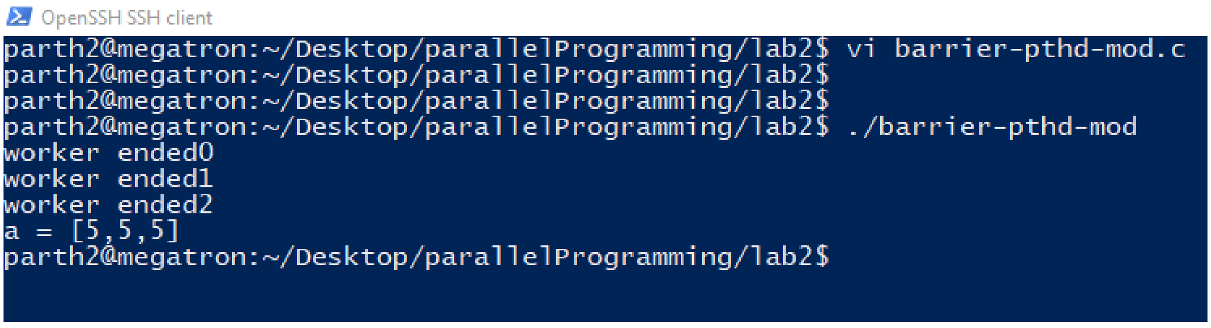
Exercise-3): -

When we replace the pthread_join() with a barrier synchronization, we will have to add a barrier wait and increase the number of threads in the barrier init block by 1 (to include the parent thread).


OpenSSH SSH client

```
//------------------------------------------------------------------------------
// Program code for CS 415P/515 Parallel Programming, Portland State University
//------------------------------------------------------------------------------

// A Pthread barrier demo program.
//
//
#define _GNU_SOURCE
#include <stdio.h>
#include <pthread.h>

int a[3] = {5,5,5};
pthread_mutex_t mtx;
pthread_barrier_t barr;

void worker(long k) {
  pthread_mutex_lock(&mtx);
  a[k] = k;
  int i = a[(k+1)%3];
  a[k] = i;
  pthread_mutex_unlock(&mtx);
  printf("worker ended%ld\n",k);
  pthread_barrier_wait(&barr);
}

int main(int argc, char **argv) {

  pthread_t thread[3];
  pthread_mutex_init(&mtx,NULL);
  pthread_barrier_init(&barr,NULL,4);

  for (long k = 0; k < 3; k++)
    pthread_create(&thread[k], NULL, (void*)worker, (void*)k);
 // for (long k = 0; k < 3; k++)
 //  pthread_join(thread[k], NULL);
  pthread_barrier_wait(&barr);

  printf("a = [%d,%d,%d]\n", a[0], a[1], a[2]);
}
~
~
```

The output is: -

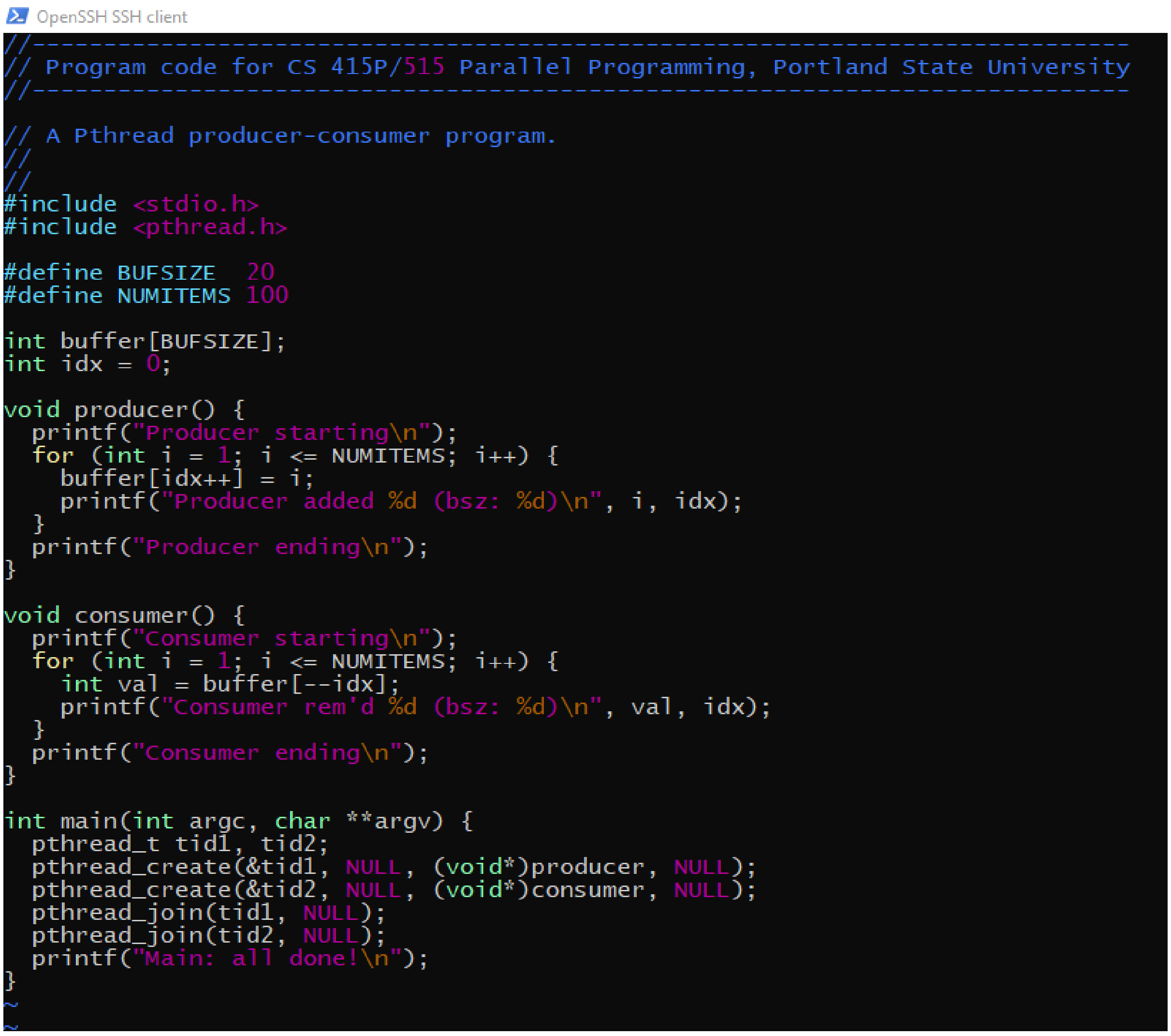

OpenSSH SSH client

```
parth2@megatron:~/Desktop/parallelProgramming/lab2$ vi barrier-pthd-mod.c
parth2@megatron:~/Desktop/parallelProgramming/lab2$
parth2@megatron:~/Desktop/parallelProgramming/lab2$
parth2@megatron:~/Desktop/parallelProgramming/lab2$ ./barrier-pthd-mod
worker ended0
worker ended1
worker ended2
a = [5,5,5]
parth2@megatron:~/Desktop/parallelProgramming/lab2$
```

Lab Experiments (Producer-Consumer Problem)

Exercise-1): -

The code is given below: -



```
//----------------------------------------------------------------------------
// Program code for CS 415P/515 Parallel Programming, Portland State University
//----------------------------------------------------------------------------

// A Pthread producer-consumer program.
//
//
#include <stdio.h>
#include <pthread.h>

#define BUFSIZE  20
#define NUMITEMS 100

int buffer[BUFSIZE];
int idx = 0;

void producer() {
  printf("Producer starting\n");
  for (int i = 1; i <= NUMITEMS; i++) {
    buffer[idx++] = i;
    printf("Producer added %d (bsz: %d)\n", i, idx);
  }
  printf("Producer ending\n");
}

void consumer() {
  printf("Consumer starting\n");
  for (int i = 1; i <= NUMITEMS; i++) {
    int val = buffer[--idx];
    printf("Consumer rem'd %d (bsz: %d)\n", val, idx);
  }
  printf("Consumer ending\n");
}

int main(int argc, char **argv) {
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, (void*)producer, NULL);
  pthread_create(&tid2, NULL, (void*)consumer, NULL);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  printf("Main: all done!\n");
}
~
~
```

This is a classic producer consumer example where the producer is producing some data and is storing it in a data structure. Producer works as a separate thread.

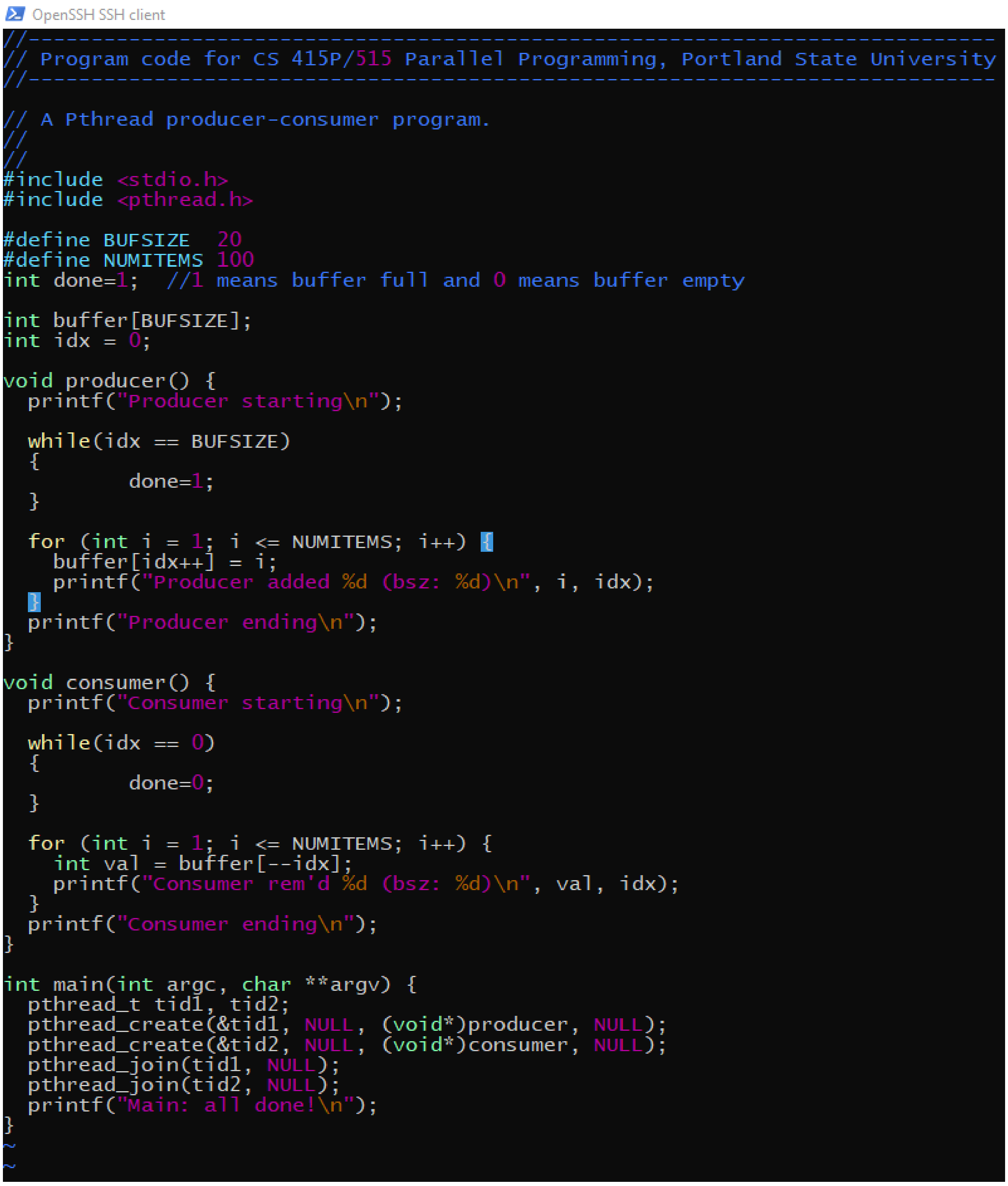
Here, Consumer is consuming the data from the data structure in which the producer is dumping the data. Consumer works as a separate thread.

Since both producer and consumer can work as separate threads, therefore, the data structure can be updated and retrieved dynamically.

In the outputs also, we can see that the producer pushes the data into the array from where the consumer starts consuming the data and prints it on the screen. The consumer may start at any point in time after the producer start pushing data into the data structure. The only problem that can occur is the out of bounds problem and also the producer and consumer can have a synchronization problem if both of them try to access the same element in the array.

Exercise-2): -

The modified code for the busy waiting synchronization is: -

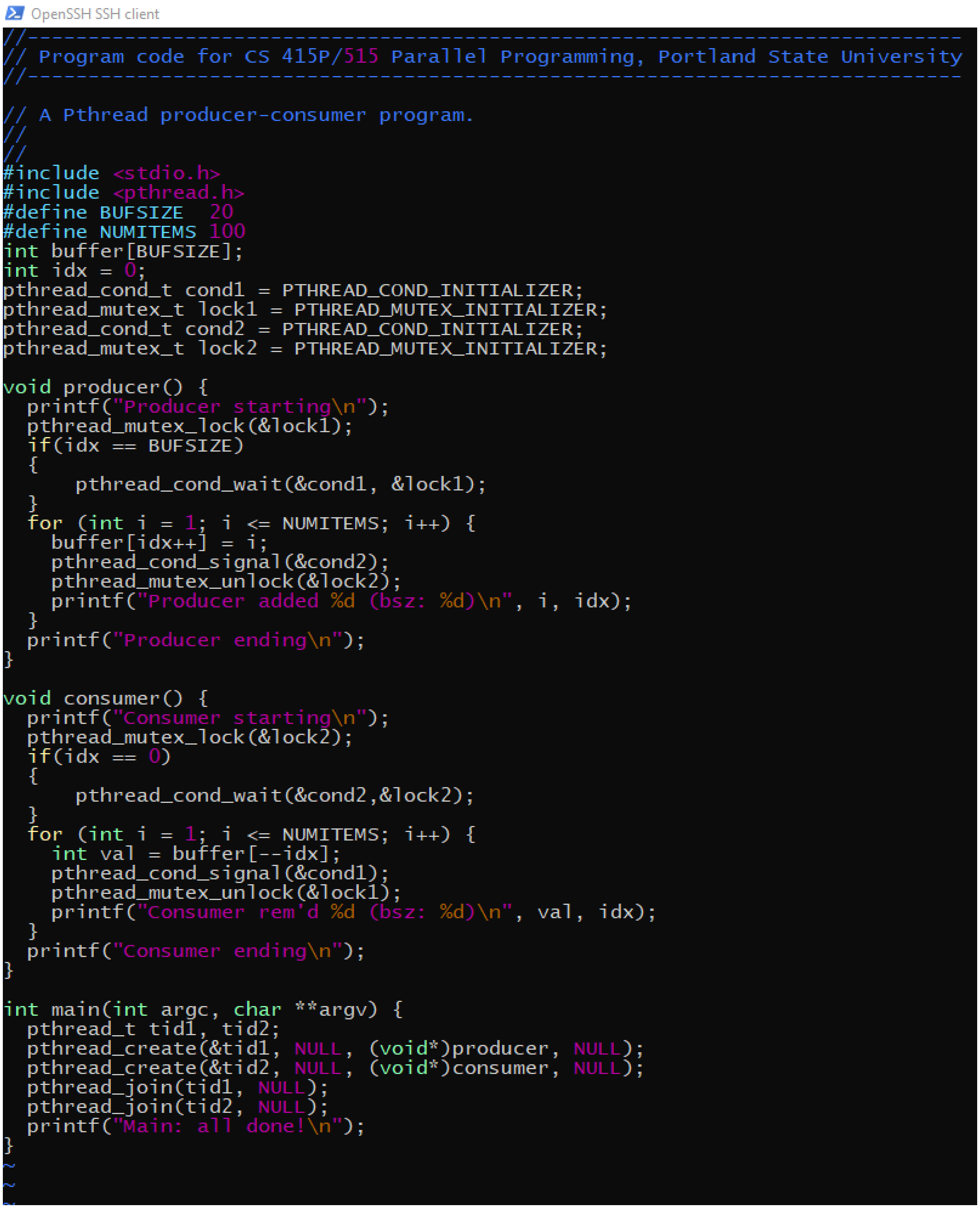


OpenSSH SSH client

```
//----------------------------------------------------------------------------
// Program code for CS 415P/515 Parallel Programming, Portland State University
//----------------------------------------------------------------------------

// A Pthread producer-consumer program.
//
//
#include <stdio.h>
#include <pthread.h>

#define BUFSIZE  20
#define NUMITEMS 100
int done=1;  //1 means buffer full and 0 means buffer empty

int buffer[BUFSIZE];
int idx = 0;

void producer() {
  printf("Producer starting\n");

  while(idx == BUFSIZE)
  {
          done=1;
  }

  for (int i = 1; i <= NUMITEMS; i++) {
    buffer[idx++] = i;
    printf("Producer added %d (bsz: %d)\n", i, idx);
  }
  printf("Producer ending\n");
}

void consumer() {
  printf("Consumer starting\n");

  while(idx == 0)
  {
          done=0;
  }

  for (int i = 1; i <= NUMITEMS; i++) {
    int val = buffer[--idx];
    printf("Consumer rem'd %d (bsz: %d)\n", val, idx);
  }
  printf("Consumer ending\n");
}

int main(int argc, char **argv) {
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, (void*)producer, NULL);
  pthread_create(&tid2, NULL, (void*)consumer, NULL);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  printf("Main: all done!\n");
}
~
~
```

The program runs perfectly fine and no, there are no out of bounds exception or error.

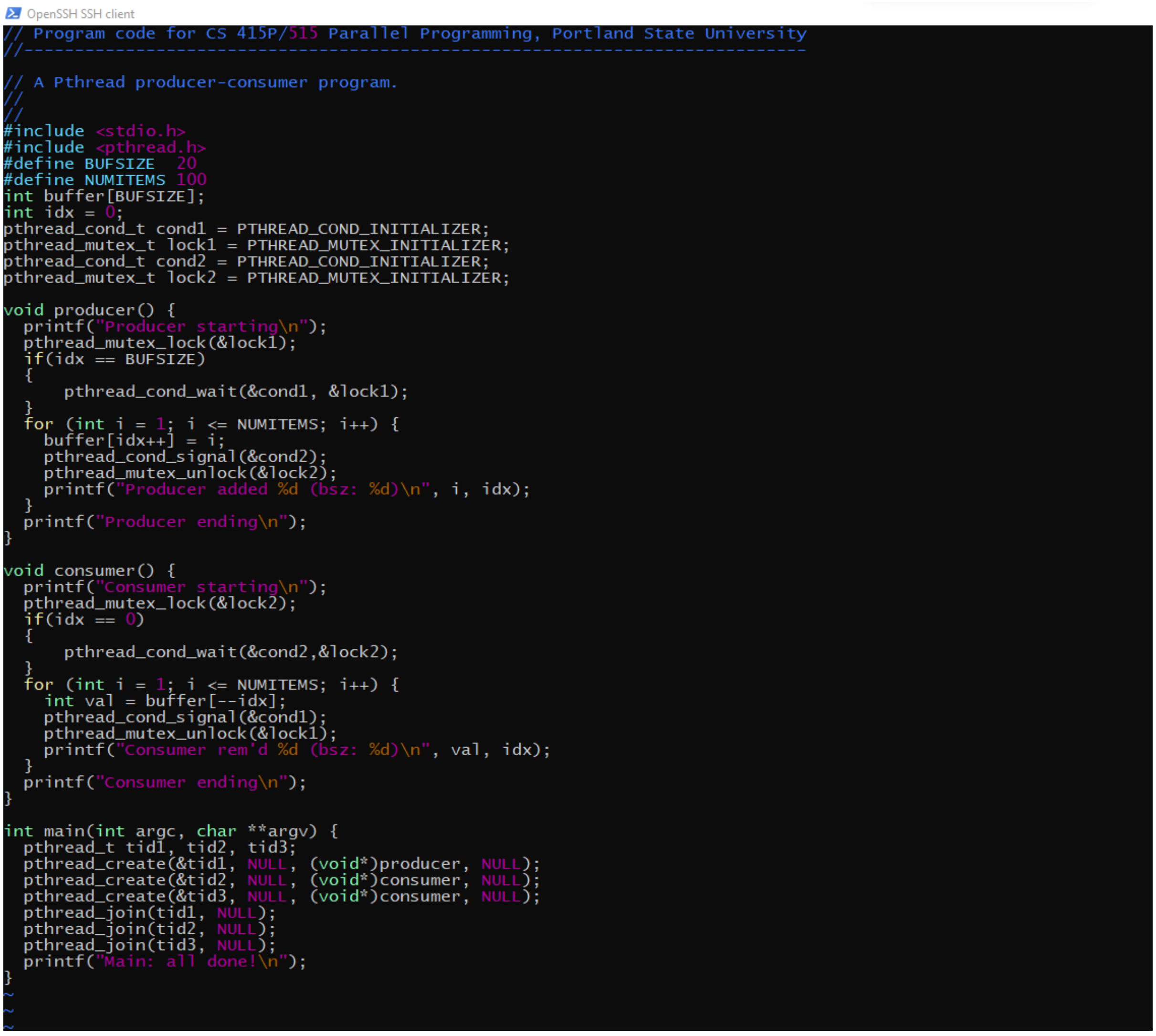
Exercise-3): - This will be like the condition variables in the exercise-1 of the condition variables. The modified code is given below: -



```
//----------------------------------------------------------------------------
// Program code for CS 415P/515 Parallel Programming, Portland State University
//----------------------------------------------------------------------------

// A Pthread producer-consumer program.
//
//
#include <stdio.h>
#include <pthread.h>
#define BUFSIZE  20
#define NUMITEMS 100
int buffer[BUFSIZE];
int idx = 0;
pthread_cond_t cond1 = PTHREAD_COND_INITIALIZER;
pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond2 = PTHREAD_COND_INITIALIZER;
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;

void producer() {
  printf("Producer starting\n");
  pthread_mutex_lock(&lock1);
  if(idx == BUFSIZE)
  {
      pthread_cond_wait(&cond1, &lock1);
  }
  for (int i = 1; i <= NUMITEMS; i++) {
    buffer[idx++] = i;
    pthread_cond_signal(&cond2);
    pthread_mutex_unlock(&lock2);
    printf("Producer added %d (bsz: %d)\n", i, idx);
  }
  printf("Producer ending\n");
}

void consumer() {
  printf("Consumer starting\n");
  pthread_mutex_lock(&lock2);
  if(idx == 0)
  {
      pthread_cond_wait(&cond2,&lock2);
  }
  for (int i = 1; i <= NUMITEMS; i++) {
    int val = buffer[--idx];
    pthread_cond_signal(&cond1);
    pthread_mutex_unlock(&lock1);
    printf("Consumer rem'd %d (bsz: %d)\n", val, idx);
  }
  printf("Consumer ending\n");
}

int main(int argc, char **argv) {
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, (void*)producer, NULL);
  pthread_create(&tid2, NULL, (void*)consumer, NULL);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  printf("Main: all done!\n");
}
~
~
```

Extra(Exercise-4)): -

The code for the same is given below: -



OpenSSH SSH client

```
// Program code for CS 415P/515 Parallel Programming, Portland State University
//-------------------------------------------------------------------------------

// A Pthread producer-consumer program.
//
//
#include <stdio.h>
#include <pthread.h>
#define BUFSIZE  20
#define NUMITEMS 100
int buffer[BUFSIZE];
int idx = 0;
pthread_cond_t cond1 = PTHREAD_COND_INITIALIZER;
pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond2 = PTHREAD_COND_INITIALIZER;
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;

void producer() {
  printf("Producer starting\n");
  pthread_mutex_lock(&lock1);
  if(idx == BUFSIZE)
  {
      pthread_cond_wait(&cond1, &lock1);
  }
  for (int i = 1; i <= NUMITEMS; i++) {
    buffer[idx++] = i;
    pthread_cond_signal(&cond2);
    pthread_mutex_unlock(&lock2);
    printf("Producer added %d (bsz: %d)\n", i, idx);
  }
  printf("Producer ending\n");
}

void consumer() {
  printf("Consumer starting\n");
  pthread_mutex_lock(&lock2);
  if(idx == 0)
  {
      pthread_cond_wait(&cond2,&lock2);
  }
  for (int i = 1; i <= NUMITEMS; i++) {
    int val = buffer[--idx];
    pthread_cond_signal(&cond1);
    pthread_mutex_unlock(&lock1);
    printf("Consumer rem'd %d (bsz: %d)\n", val, idx);
  }
  printf("Consumer ending\n");
}

int main(int argc, char **argv) {
  pthread_t tid1, tid2, tid3;
  pthread_create(&tid1, NULL, (void*)producer, NULL);
  pthread_create(&tid2, NULL, (void*)consumer, NULL);
  pthread_create(&tid3, NULL, (void*)consumer, NULL);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  pthread_join(tid3, NULL);
  printf("Main: all done!\n");
}
~
~
~
```