Ronald Chatelier

1. a) On a single-CPU system, under what circumstances does a multithreaded program using kernel threads provide better performance (such as faster execution time) compared to a single threaded solution (that does not use asynchronous or event-based programming) ? Explain with general principles.
Give TWO example applications.

With the use of a multi-threaded program, it provides better performance by having the threads share memory and message passing. This allows address space to be shared allowing one application as data is being transferred. This is because multithreading has multiple threads that hold the code, data, and files with their own registers and stack.

Examples of applications include document editors such as Google Docs and video games such as Mario Kart. Google Docs is checking for spelling errors, text alignment, font type, and auto-saving for document files. Mario Kart has different sound effects from the characters, items, and karts while the race tracks have their hazards and background music enabled.

b) A new operating system provides a synchronization API and a library for user-level programs (i.e. like the pthread) for which the mutex lock and unlock operation are implemented with **test_and_set** like this:

```
void mutex_lock(mutex* plock) {
        while (test_and_set(plock)) {
        };
}
void mutex_unlock(mutex* plock) {
        *plock = false;
}
```

Is this implementation of mutex synchronization correct for use in general-purpose user-level applications? What could go wrong? It helps to think of an example application, like the bounded-buffer problem or the dining philosophers.

That implementation of mutex synchronization is not correct because the mutex would be locked first and has the test_and_set shared as well as the restrictions such as 1 semaphore and 1 condvar. What could go wrong is the different threads having access to the mutex and not traversing. This can include bounded waiting, which happens when no process is waiting for a resource for an infinite amount of time.

When the process is done, there's no guarantee that another process will happen which will cause the 1st greedy process to hog the lock and go through the process of

releasing and taking. This will cause the whole process to go through starvation over time.

c) On a running Linux kernel (version > 2.6) at some point the thread_info.preempt_count field for a kernel task we call *A* is equal to 2. (Linux kernel synchronization is discussed in the textbook).

Answer these questions:

c1) Is task A currently preemptable? Explain.

Task A isn't preemptable because thread_info.preempt_count is considered a non-zero, which would make it hold 2 locks.

c2) What is new value of thread_info.preempt_count field for task A after it acquires a new lock ? Explain.

thread_info.preempt_count will have 3 as the new value if task A acquires a new lock because having the new lock available will have thread_info.preempt_count increment by 1.

c3) What is the condition for kernel task A to be safely interruptible?

Kernel task A becomes interruptible by having the lock not activated, which is done by having the thread_info.preempt_count value checked.

c4) Assuming that all locks held by task A are spinlocks, how many CPUs are on that computer?

9 CPUs would be on that computer since there are threads blocked by the spinlock. This will cause the resources in a loop that doesn't stop, which means that the CPU will require more overall.

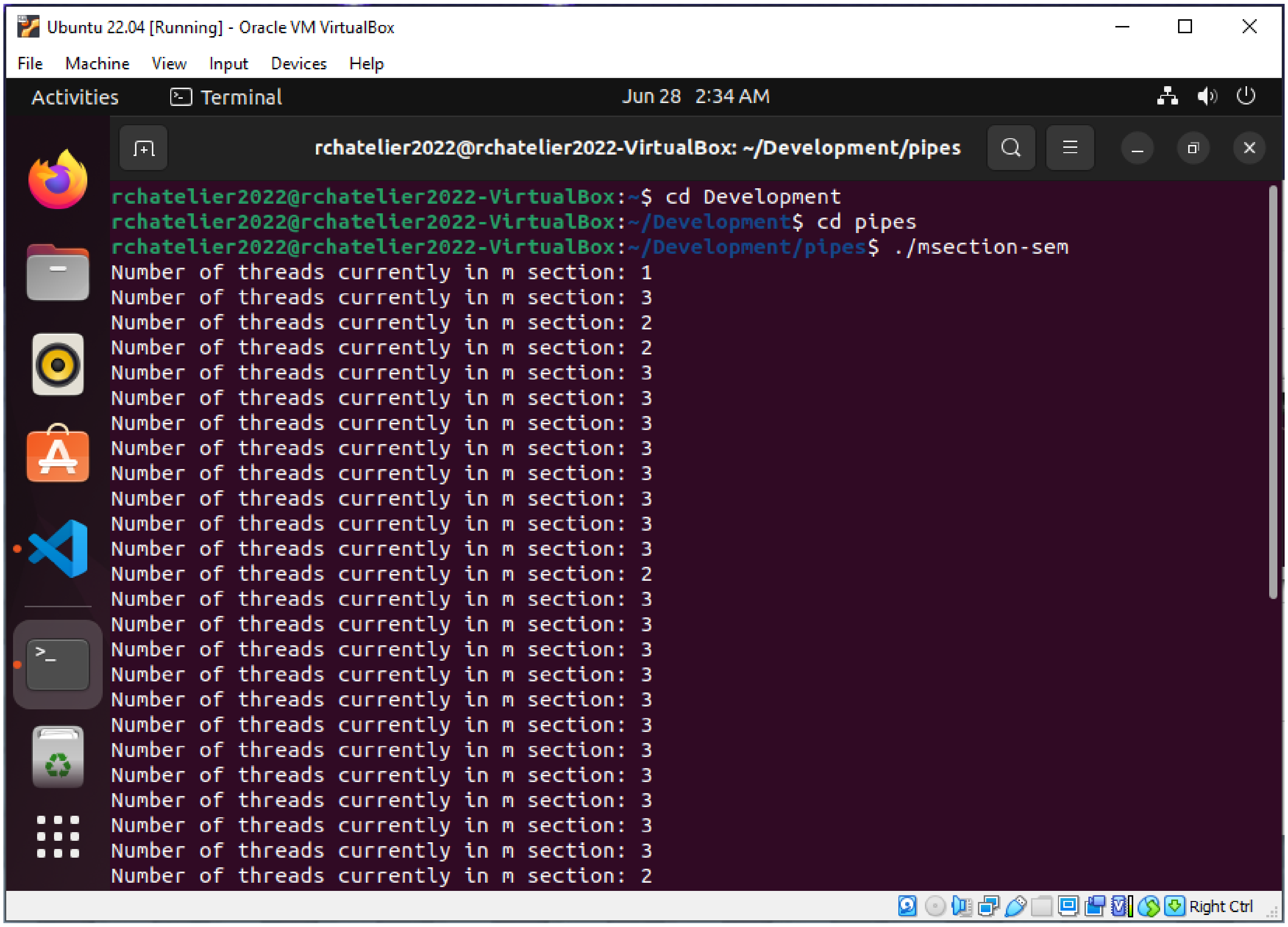
2. a) Write a C program called **msection-sem.c** using the *pthread* library that implements the algorithm above.

```
#include <string.h>

#include <pthread.h>

#include <stdbool.h>

#include <semaphore.h>
```

```
#include <stdlib.h>

#include <unistd.h>

#include <stdio.h>

void enter();

void leave();

void doWork();

void doCriticalWork();

// amount of threads

int m = 3;

int counter = 0;

// semaphore object

sem_t semObject;

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

 sem_init (&semObject, 0, 3);

long i;

// number of threads

long n = 10;

pthread_attr_t attr;

pthread_t *vtid = (pthread_t *) calloc(n, sizeof(pthread_t));

if (vtid == NULL) {

  printf("Error \n\n");

  exit(1);

}

// initializes the thread attributes object
```

```
pthread_attr_init(&attr);

// sets the detach state attribute

pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

 for (i=0; i<n; i++) {

  int r = pthread_create(&vtid[i], &attr, doWork, (void*)i);

  if (r < 0) {

    printf("Error %d\n", r);

    exit(2);

  }

}

 for (i=0; i<n; i++) {

  // waits for defined thread to terminate

  pthread_join(vtid[i], NULL);

}

// counting for completed threads

printf("all threads finished\ncounter=%ld\n", counter);

pthread_attr_destroy(&attr);

free(vtid);

return 0;

}

void doWork() {

  while (true) {



     // limited access to m threads
```

```
      enter (&semObject);
    // execute m-section
    //run by max.m threads
    doCriticalWork();
      sleep(69);
    // leave m-section
    leave (&semObject);
  }
}
void doCriticalWork() {
  pthread_t vT = pthread_self();
  printf("Number of threads currently in m section: %ld\n", counter);
 }
void enter(sem_t *semObject) {
  sem_wait (semObject);
  counter++;
}
void leave(sem_t *semObject) {
  counter--;
  sem_post(semObject);
}
```



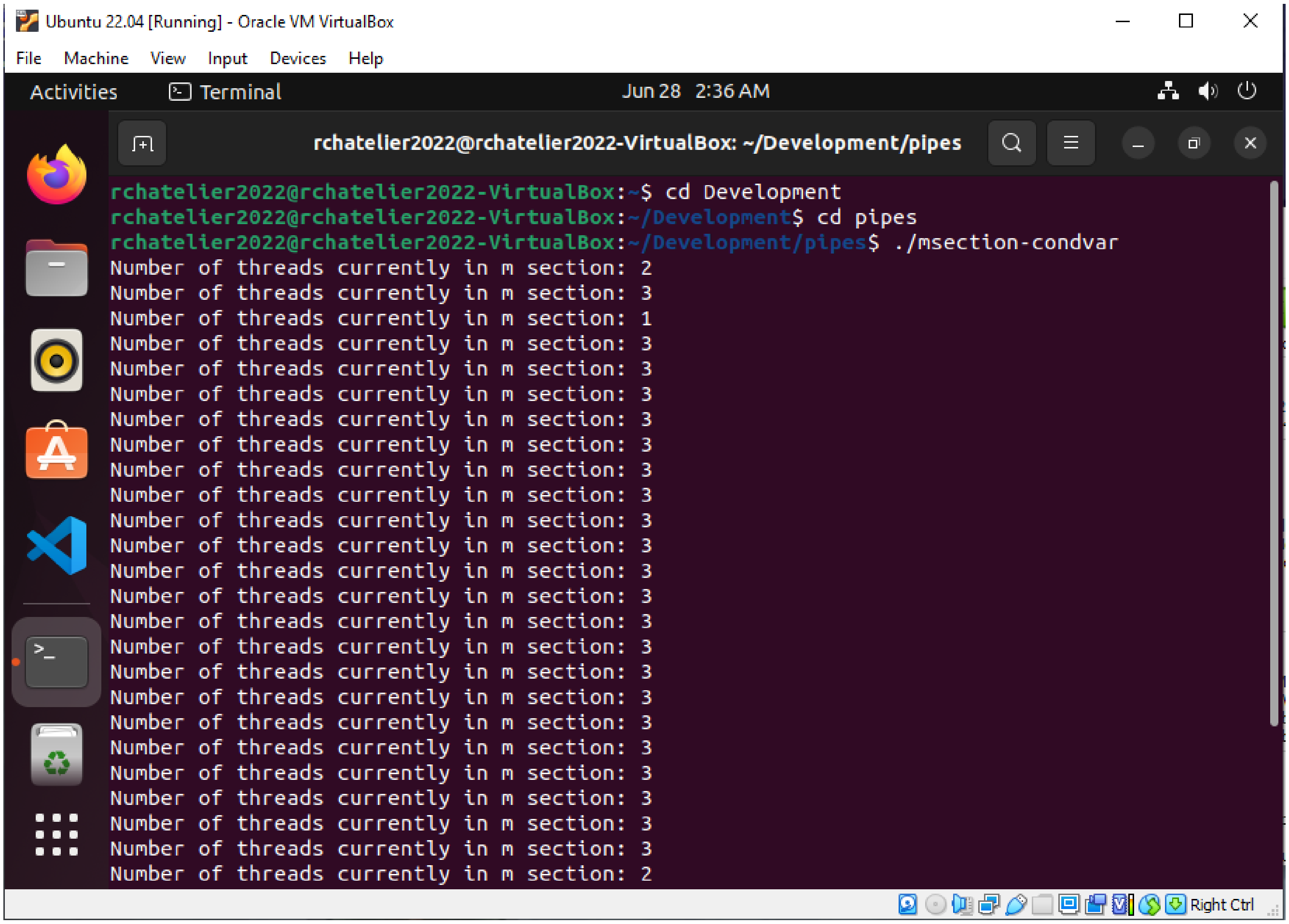
b) Write a C program called **msection-condvar.c** using the *pthread* library that implements the algorithm above. The *enter*() and *leave*() functions must use only **one condition variable** and one or more **mutex**es for synchronization.

```
#include <string.h>

#include <pthread.h>

#include <stdbool.h>

#include <stdlib.h>

#include <unistd.h>

#include <stdio.h>


void enter();

void leave();
```

```
void doWork();

void doCriticalWork();


// amount of threads

int m = 3;

int counter = 0;

pthread_mutex_t cvmutex;

pthread_cond_t condvar;


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


   pthread_mutex_init (&cvmutex, NULL);

   pthread_cond_init (&condvar, NULL);


 long i;

 // number of threads

 long n = 10;

 pthread_attr_t attr;

 pthread_t *vtid = (pthread_t *) calloc(n, sizeof(pthread_t));

 if (vtid == NULL) {

   printf("Error \n\n");

   exit(1);

 }

  // initializes the thread attributes object

 pthread_attr_init(&attr);

 // sets the detach state attribute

 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

  for (i=0; i<n; i++) {
```

```
    int r = pthread_create(&vtid[i], &attr, doWork, (void*)i);
    if (r < 0) {
      printf("Error %d\n", r);
      exit(2);
    }
  }
  for (i=0; i<n; i++) {
 // waits for defined thread to terminate
 pthread_join(vtid[i], NULL);
 }
 // counting for completed threads
 printf("all threads finished\ncounter=%d\n", counter);
 pthread_attr_destroy(&attr);
 free(vtid);
 return 0;
}

void doWork() {
   while (true) {

       enter (&cvmutex, &condvar);
       // execute m-section
       //run by max.m threads
   doCriticalWork();
       sleep(69);
       // leave m-section
   leave (&cvmutex, &condvar);
   }
```

```
}

void doCriticalWork() {
   pthread_t vT = pthread_self();
   printf("Number of threads currently in m section: %d\n", counter);

}


void enter(pthread_cond_t *cThread, pthread_mutex_t *muTex ) {
   // References with the mutex object in the unlocked state
 pthread_mutex_lock(muTex);
   while(counter >= m) {
       pthread_cond_wait(cThread, muTex);
   }
   // References with the mutex object in the locked state
   pthread_mutex_unlock(muTex);
   counter++;
}

void leave(pthread_cond_t *cThread, pthread_mutex_t *muTex) {
   counter--;
   pthread_mutex_unlock(muTex);
   pthread_cond_signal(cThread);
   pthread_mutex_unlock(muTex);

}
```



3. a) Implement a *barrier* for pthreads in C++ using pthread condition variables and mutexes in a file called ***barrier.cc***.

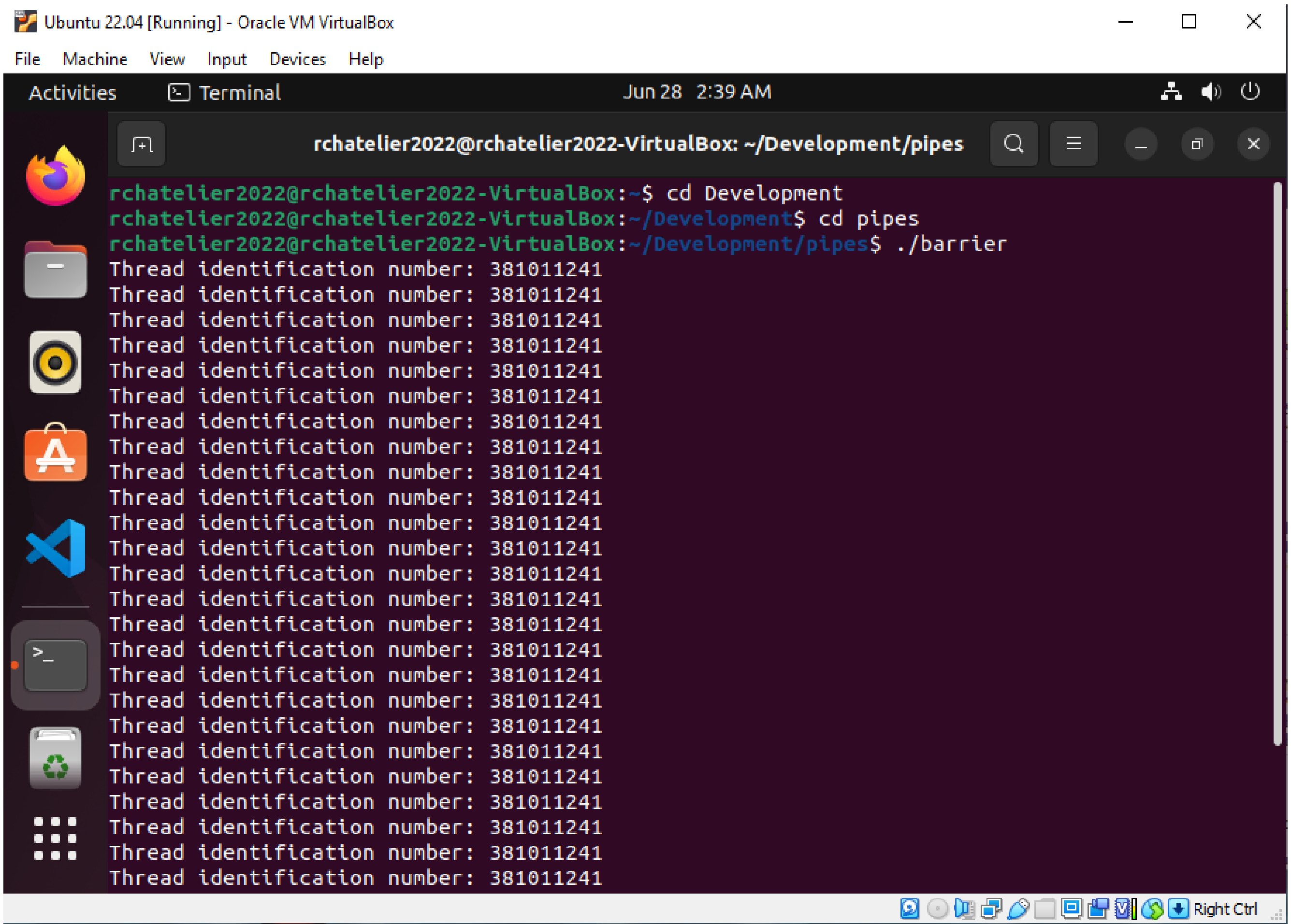
```
#include <string.h>

#include <pthread.h>

#include <stdbool.h>

#include <stdlib.h>

#include <unistd.h>

#include <stdio.h>

void thread();

void *thread_fun(void *param);
```

```
pthread_mutex_t cvmutex;

pthread_cond_t condvar;

class Barrier {

  public:

  int counter;

  // number of threads

  Barrier(int n)

  {

      pthread_mutex_init (&cvmutex, NULL);

      pthread_cond_init(&condvar, NULL);

      counter = 0;

  }

void wait() {


  if(counter < 0) {

      pthread_mutex_lock(&cvmutex);

      counter++;

      pthread_cond_wait(&condvar, &cvmutex);

      pthread_mutex_unlock(&cvmutex);

  }


  else{

      pthread_mutex_lock(&cvmutex);

      pthread_cond_broadcast(&condvar);

      counter = 0;

      pthread_mutex_unlock(&cvmutex);
```

```
  }

  };

private:

int x;

pthread_mutex_t cvmutex;

pthread_cond_t condvar;

};

// Barrier object declared as a global variable

Barrier shield(7);

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

long i;

// number of threads

long n = 10;

pthread_attr_t attr;

pthread_t *vtid = (pthread_t *) calloc(n, sizeof(pthread_t));

if (vtid == NULL) {

  printf("Error \n\n");

  exit(1);

}

// initializes the thread attributes object

pthread_attr_init(&attr);

// sets the detach state attribute

pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

 for (i=0; i<n; i++) {

  int r = pthread_create(&vtid[i], &attr, thread_fun, (void*)i);

  if (r < 0) {
```

```
    printf("Error %d\n", r);
    exit(2);
  }
}
 // waits for defined thread to terminate
 for (i=0; i<n; i++) {
  pthread_join(vtid[i], NULL);
}
// counting for completed threads
printf("all threads finished\ncounter=%d\n", shield.counter);
pthread_attr_destroy(&attr);
free(vtid);
return 0;
}
// child threads run this thread function
void *thread_fun(void *param){
  // thread runs in loop
  while(true){
    // work is done
    shield.wait();
      thread();
  }
}
void thread(){
 pthread_t results = pthread_self();
```

```
  printf("Thread identification number: %d\n", thread);

  sleep(69);



}
```



b) Write a thread function in file barrier.cc that demonstrates in a meaningful way how your Barrier object works. It should look like the code in function *thread_fun* above.

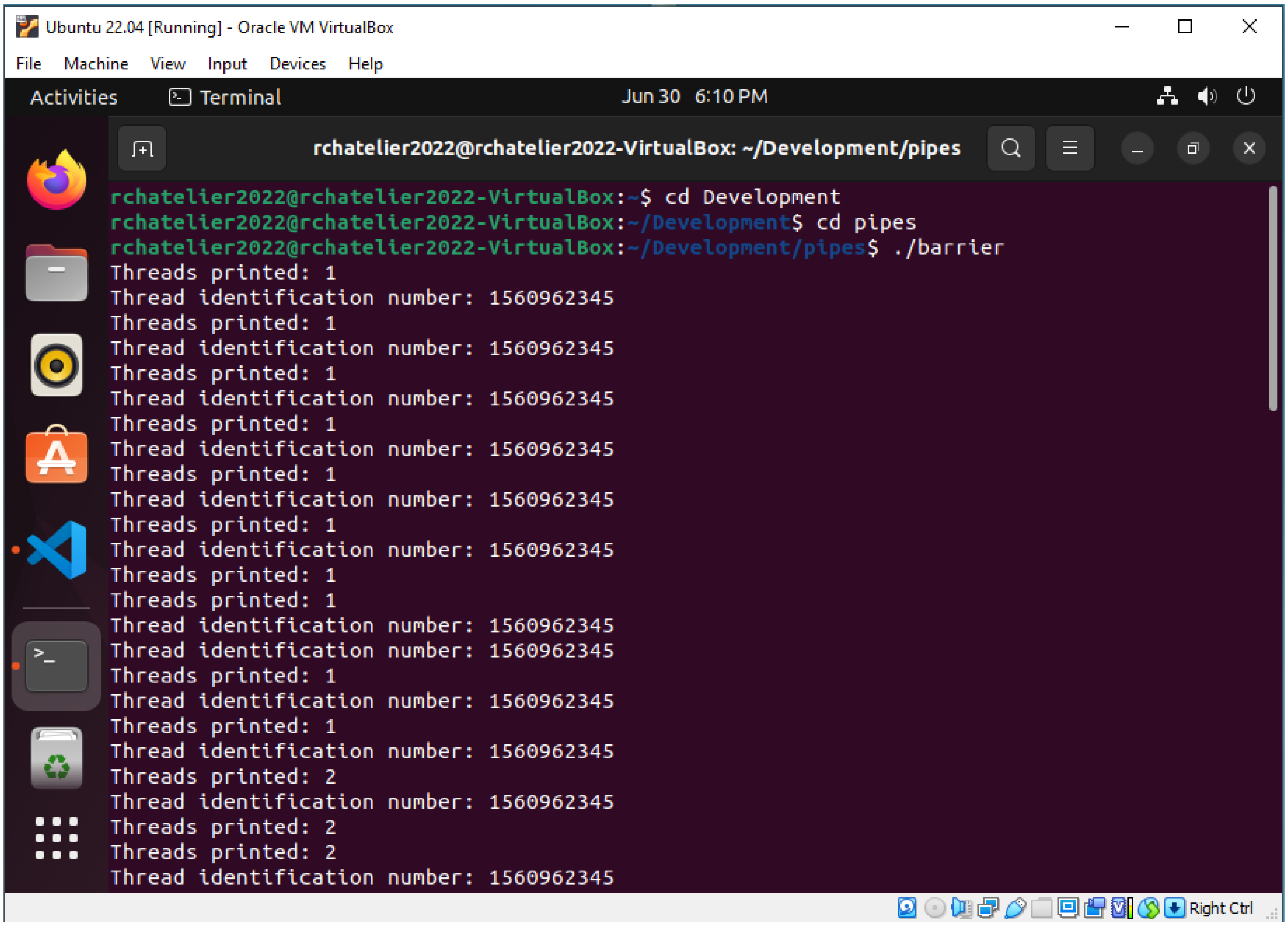
```
#include <string.h>

#include <pthread.h>
```

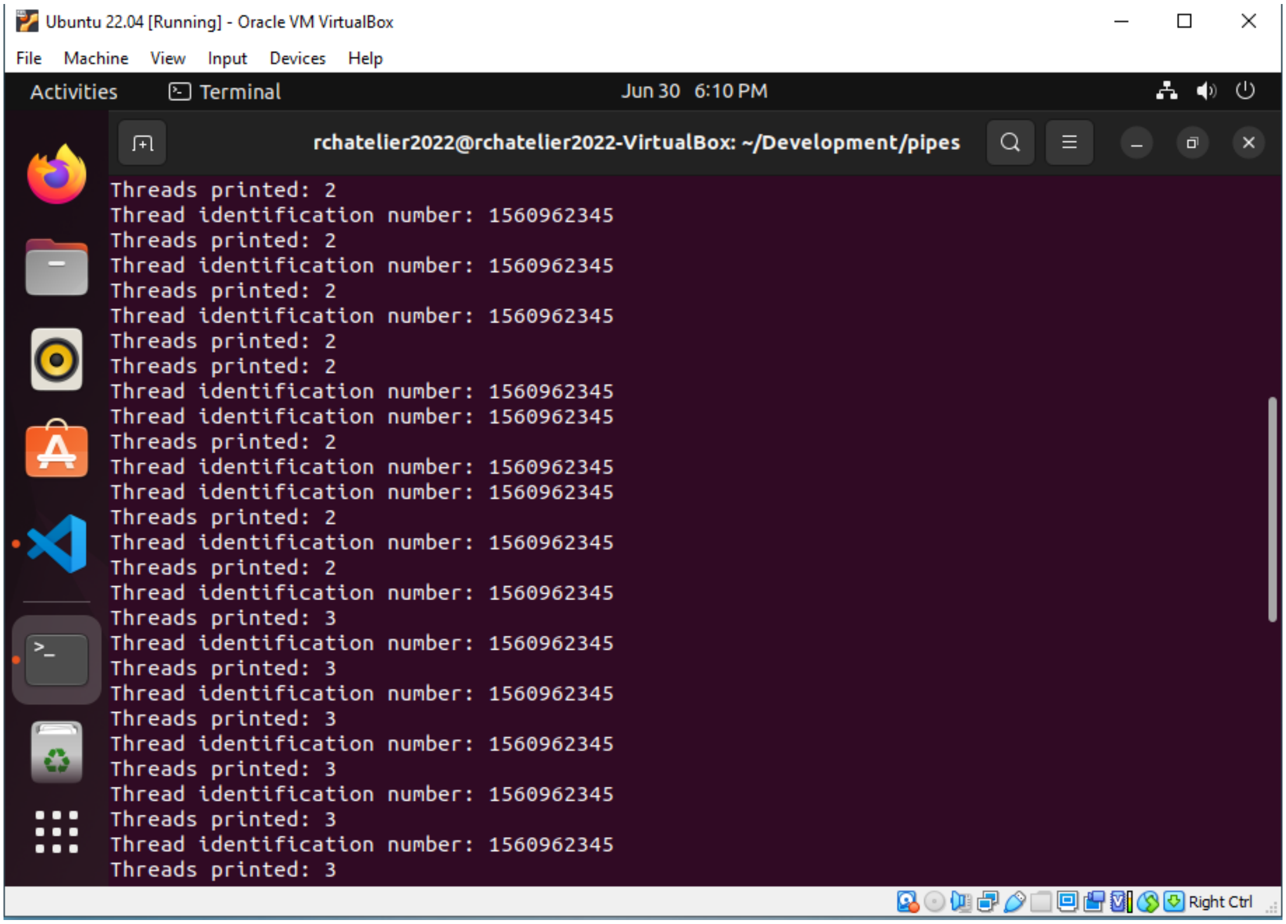
```
#include <stdbool.h>

#include <stdlib.h>

#include <unistd.h>

#include <stdio.h>

void thread();

void *thread_fun(void *param);

pthread_mutex_t cvmutex;

pthread_cond_t condvar;

class Barrier {

  public:

  int counter;

  // number of threads

  Barrier(int n)

  {

      pthread_mutex_init (&cvmutex, NULL);

      pthread_cond_init(&condvar, NULL);

      counter = 0;

  }

void wait() {


  if(counter < 0) {

      pthread_mutex_lock(&cvmutex);

      counter++;

      pthread_cond_wait(&condvar, &cvmutex);

      pthread_mutex_unlock(&cvmutex);

  }
```

```
  else{
      pthread_mutex_lock(&cvmutex);
      pthread_cond_broadcast(&condvar);
      counter = 0;
      pthread_mutex_unlock(&cvmutex);
  }
  };
private:
int x;
pthread_mutex_t cvmutex;
pthread_cond_t condvar;
};
// Barrier object declared as a global variable
Barrier shield(7);
int main(int argc, char** argv) {
long i;
// number of threads
long n = 10;
pthread_attr_t attr;
pthread_t *vtid = (pthread_t *) calloc(n, sizeof(pthread_t));
if (vtid == NULL) {
  printf("Error \n\n");
  exit(1);
}
// initializes the thread attributes object
```

```
pthread_attr_init(&attr);

// sets the detach state attribute

pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

 for (i=0; i<n; i++) {

  int r = pthread_create(&vtid[i], &attr, thread_fun, (void*)i);

  if (r < 0) {

    printf("Error %d\n", r);

    exit(2);

  }

}

 // waits for defined thread to terminate

 for (i=0; i<n; i++) {

  pthread_join(vtid[i], NULL);

}

// counting for completed threads

printf("all threads finished\ncounter=%d\n", shield.counter);

pthread_attr_destroy(&attr);

free(vtid);

return 0;

}

// child threads run this thread function

void *thread_fun(void *param){

  // thread runs in loop

  while(true){

    // work is done

    shield.wait();
```

```
      thread();

  }

}

void thread(){
 pthread_t results = pthread_self();


  int adding = 0;

  int bundle = 1;

  while(adding <= 10) {


      printf("Threads printed: %d\n", bundle);

      printf("Thread identification number: %d\n", thread);

      sleep(69);

      adding++;

      bundle++;


  }


}
```


Ubuntu 22.04 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Terminal
Jun 30 6:10 PM
rchatelier2022@rchatelier2022-VirtualBox: ~/Development/pipes
rchatelier2022@rchatelier2022-VirtualBox:~$ cd Development
rchatelier2022@rchatelier2022-VirtualBox:~/Development$ cd pipes
rchatelier2022@rchatelier2022-VirtualBox:~/Development/pipes$ ./barrier
Threads printed: 1
Thread identification number: 1560962345
Threads printed: 1
Thread identification number: 1560962345
Threads printed: 1
Thread identification number: 1560962345
Threads printed: 1
Thread identification number: 1560962345
Threads printed: 1
Thread identification number: 1560962345
Threads printed: 1
Thread identification number: 1560962345
Threads printed: 1
Threads printed: 1
Thread identification number: 1560962345
Thread identification number: 1560962345
Threads printed: 1
Thread identification number: 1560962345
Threads printed: 1
Thread identification number: 1560962345
Threads printed: 2
Thread identification number: 1560962345
Threads printed: 2
Threads printed: 2
Thread identification number: 1560962345
Right Ctrl


Ubuntu 22.04 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Terminal
Jun 30 6:10 PM
rchatelier2022@rchatelier2022-VirtualBox: ~/Development/pipes
Threads printed: 2
Thread identification number: 1560962345
Threads printed: 2
Thread identification number: 1560962345
Threads printed: 2
Thread identification number: 1560962345
Threads printed: 2
Threads printed: 2
Thread identification number: 1560962345
Thread identification number: 1560962345
Threads printed: 2
Thread identification number: 1560962345
Thread identification number: 1560962345
Threads printed: 2
Thread identification number: 1560962345
Threads printed: 2
Thread identification number: 1560962345
Threads printed: 3
Thread identification number: 1560962345
Threads printed: 3
Thread identification number: 1560962345
Threads printed: 3
Thread identification number: 1560962345
Threads printed: 3
Thread identification number: 1560962345
Threads printed: 3
Thread identification number: 1560962345
Threads printed: 3
Right Ctrl