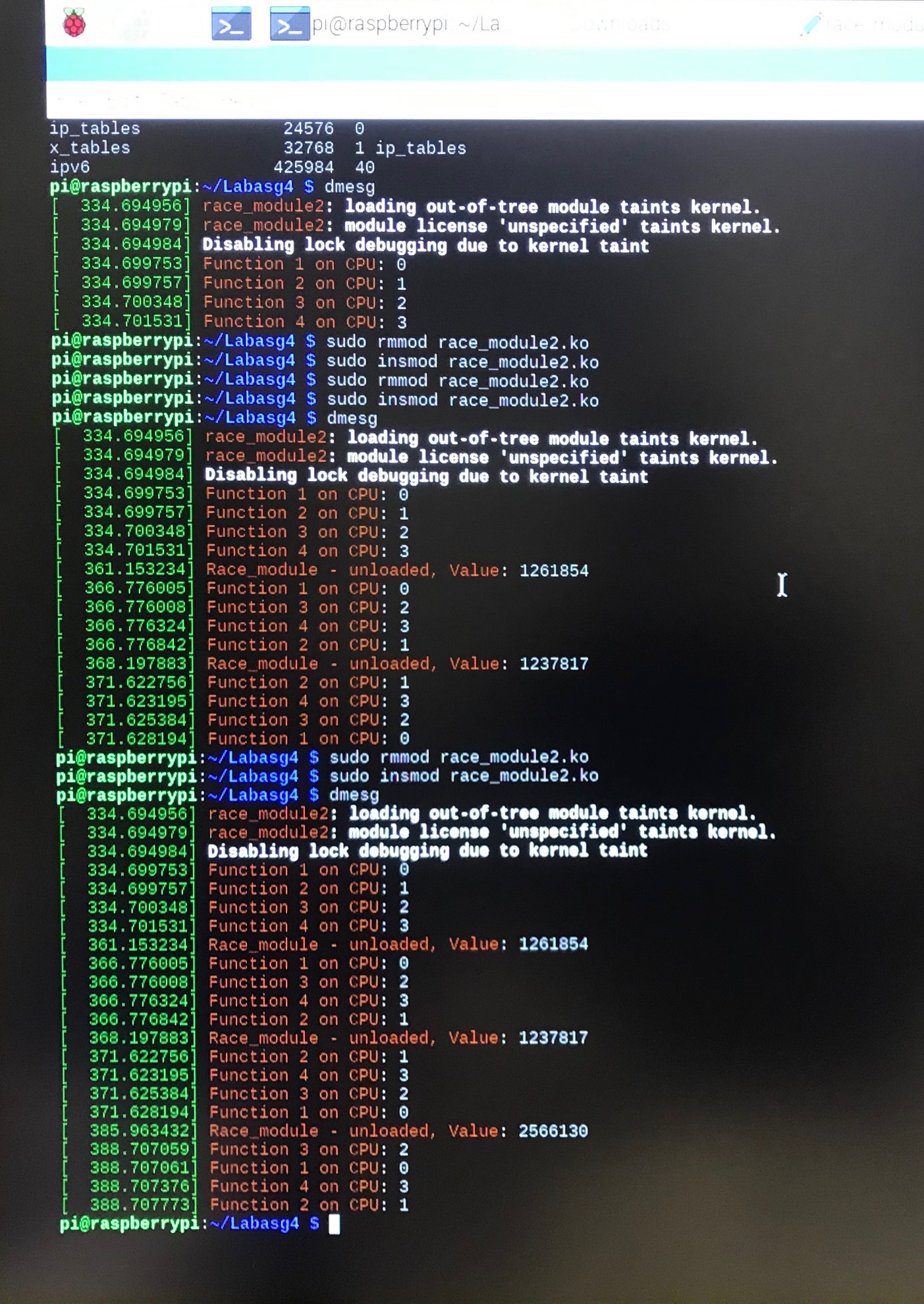
CS630 – Operating Systems Design

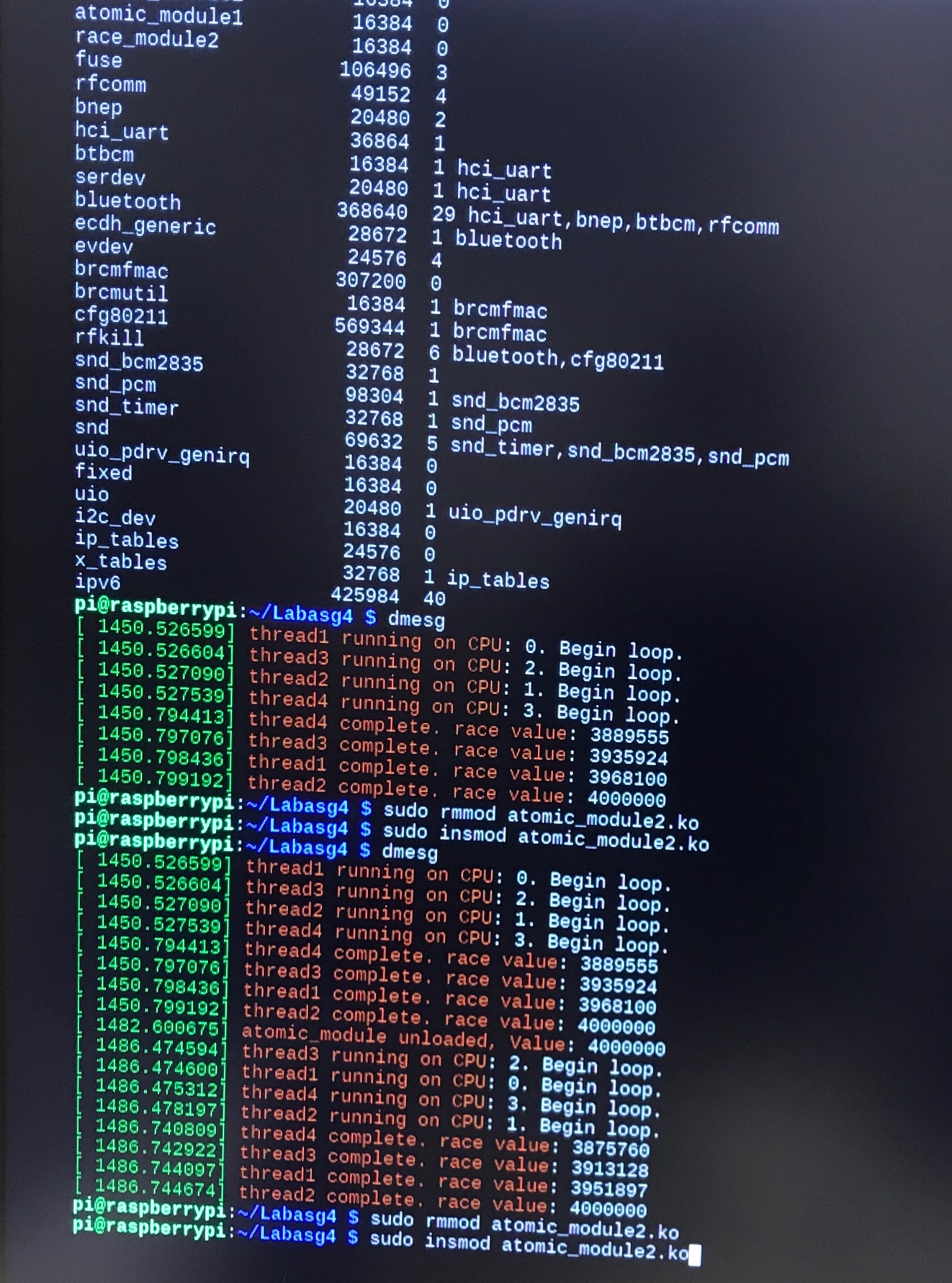
Lab 4: Kernel Synchronization

Jonathan Wolf, Soumyadeep Basu, Charanpreet Kaur (**exercise 0**)

**Exercise 1:** \_\_\_\_

**Exercise 2:**

**Exercise 3:** The final value of race in all executions is 4,000,000.



Thread1 exec time: 0.271837s

Thread2 exec time: 0.272102s

Thread3 exec time: 0.270472s

Thread4 exec time: 0.266874s

**Exercise 4:**



Thread1 exec time: 0.949174s

Thread2 exec time: 0.935208s

Thread3 exec time: 0.944843s

Thread4 exec time: 0.922531s

Mutex takes significantly longer than atomic variables. However, using a mutex allows you greater flexibility. With atomic variables, the actual operations that are atomic are pre-defined. With a mutex, you can specify any sequence of operations you want to be contained within the atomic code. For example, you might want to be able to do several additions and then calculate an average all within a single atomic segment without interruption. You could do this with mutex, but not a simple atomic variable.

Race module code:

#include <linux/kthread.h>

#include <linux/init.h>

#include <linux/module.h>

#include <linux/kernel.h>

#define iters 1000000

static int cpu\_value;

volatile int race =0; //Segmentation fault with int so changed it to int

static int function1(void\* none)

{

int i=0;

cpu\_value = get\_cpu();

printk(KERN\_INFO "Function 1 on CPU: %d\n", cpu\_value);

for(i =0; i<iters; i++)

{

race++;

}

return 0;

}

static int function2(void \*none)

{

int i=0;

cpu\_value = get\_cpu();

printk(KERN\_INFO "Function 2 on CPU: %d\n", cpu\_value);

for(i =0; i<iters; i++)

{

race++;

}

return 0;

}

static int function3(void \*none)

{

int i=0;

cpu\_value = get\_cpu();

printk(KERN\_INFO "Function 3 on CPU: %d\n", cpu\_value);

for(i =0; i<iters; i++)

{

race++;

}

return 0;

}

static int function4(void \*none)

{

int i=0;

cpu\_value = get\_cpu();

printk(KERN\_INFO "Function 4 on CPU: %d\n", cpu\_value);

for(i =0; i<iters; i++)

{

race++;

}

return 0;

}

static struct task\_struct \*thread1;

static struct task\_struct \*thread2;

static struct task\_struct \*thread3;

static struct task\_struct \*thread4;

static int simple\_init (void) {

cpu\_value= get\_cpu();

thread1=kthread\_create(function1,NULL, "thread1");

thread2=kthread\_create(function2,NULL, "thread2");

thread3=kthread\_create(function3,NULL, "thread3");

thread4=kthread\_create(function4,NULL, "thread4");

kthread\_bind(thread1, 0);

kthread\_bind(thread2, 1);

kthread\_bind(thread3, 2);

kthread\_bind(thread4, 3);

wake\_up\_process(thread1);

wake\_up\_process(thread2);

wake\_up\_process(thread3);

wake\_up\_process(thread4);

return 0;

}

static void simple\_exit (void)

{

printk(KERN\_INFO "Race\_module - unloaded, Value: %d\n", race);

}

module\_init (simple\_init);

module\_exit (simple\_exit);

Atomic module code:

#include <linux/kthread.h>

#include <linux/init.h>

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/types.h>

#define iters 1000000

static int cpu\_value;

atomic\_t race;

static int function1(void\* none)

{

int i=0;

cpu\_value = get\_cpu();

printk(KERN\_INFO "thread1 running on CPU: %d. Begin loop.\n", cpu\_value);

for(i =0; i<iters; i++)

{

atomic\_add(1, &race);

}

printk(KERN\_INFO "thread1 complete. race value: %d\n", atomic\_read(&race));

return 0;

}

static int function2(void \*none)

{

int i=0;

cpu\_value = get\_cpu();

printk(KERN\_INFO "thread2 running on CPU: %d. Begin loop.\n", cpu\_value);

for(i =0; i<iters; i++)

{

atomic\_add(1, &race);

}

printk(KERN\_INFO "thread2 complete. race value: %d\n", atomic\_read(&race));

return 0;

}

static int function3(void \*none)

{

int i=0;

cpu\_value = get\_cpu();

printk(KERN\_INFO "thread3 running on CPU: %d. Begin loop.\n", cpu\_value);

for(i =0; i<iters; i++)

{

atomic\_add(1, &race);

}

printk(KERN\_INFO "thread3 complete. race value: %d\n", atomic\_read(&race));

return 0;

}

static int function4(void \*none)

{

int i=0;

cpu\_value = get\_cpu();

printk(KERN\_INFO "thread4 running on CPU: %d. Begin loop.\n", cpu\_value);

for(i =0; i<iters; i++)

{

atomic\_add(1, &race);

}

printk(KERN\_INFO "thread4 complete. race value: %d\n", atomic\_read(&race));

return 0;

}

static struct task\_struct \*thread1;

static struct task\_struct \*thread2;

static struct task\_struct \*thread3;

static struct task\_struct \*thread4;

static int simple\_init (void) {

atomic\_set(&race, 0);

cpu\_value = get\_cpu();

thread1=kthread\_create(function1,NULL, "thread1");

thread2=kthread\_create(function2,NULL, "thread2");

thread3=kthread\_create(function3,NULL, "thread3");

thread4=kthread\_create(function4,NULL, "thread4");

kthread\_bind(thread1, 0);

kthread\_bind(thread2, 1);

kthread\_bind(thread3, 2);

kthread\_bind(thread4, 3);

wake\_up\_process(thread1);

wake\_up\_process(thread2);

wake\_up\_process(thread3);

wake\_up\_process(thread4);

return 0;

}

static void simple\_exit (void) {

printk(KERN\_INFO "atomic\_module unloaded, Value: %d\n",atomic\_read(&race));

}

module\_init (simple\_init);

module\_exit (simple\_exit);

Mutex module code:

#include <linux/kthread.h>

#include <linux/init.h>

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/mutex.h>

#define iters 1000000

static DEFINE\_MUTEX(race\_lock);

static int cpu\_value;

volatile int race =0;

static int function1(void\* none)

{

int i=0;

cpu\_value = get\_cpu();

printk(KERN\_INFO "function1 on CPU: %d\n", cpu\_value);

for(i =0; i<iters; i++)

{

mutex\_lock(&race\_lock);

race++;

mutex\_unlock(&race\_lock);

}

printk(KERN\_INFO "thread1 completed, race value: %d\n", race);

return 0;

}

static int function2(void \*none)

{

int i=0;

cpu\_value = get\_cpu();

printk(KERN\_INFO "function2 on CPU: %d\n", cpu\_value);

for(i =0; i<iters; i++)

{

mutex\_lock(&race\_lock);

race++;

mutex\_unlock(&race\_lock);

}

printk(KERN\_INFO "thread2 completed, race value: %d\n", race);

return 0;

}

static int function3(void \*none)

{

int i=0;

cpu\_value = get\_cpu();

printk(KERN\_INFO "function3 on CPU: %d\n", cpu\_value);

for(i =0; i<iters; i++)

{

mutex\_lock(&race\_lock);

race++;

mutex\_unlock(&race\_lock);

}

printk(KERN\_INFO "thread3 completed, race value: %d\n", race);

return 0;

}

static int function4(void \*none)

{

int i=0;

cpu\_value = get\_cpu();

printk(KERN\_INFO "function4 on CPU: %d\n", cpu\_value);

for(i =0; i<iters; i++)

{

mutex\_lock(&race\_lock);

race++;

mutex\_unlock(&race\_lock);

}

printk(KERN\_INFO "thread4 completed, race value: %d\n", race);

return 0;

}

static struct task\_struct \*thread1;

static struct task\_struct \*thread2;

static struct task\_struct \*thread3;

static struct task\_struct \*thread4;

static int simple\_init (void) {

cpu\_value = get\_cpu();

thread1=kthread\_create(function1,NULL, "thread1");

thread2=kthread\_create(function2,NULL, "thread2");

thread3=kthread\_create(function3,NULL, "thread3");

thread4=kthread\_create(function4,NULL, "thread4");

kthread\_bind(thread1, 0);

kthread\_bind(thread2, 1);

kthread\_bind(thread3, 2);

kthread\_bind(thread4, 3);

wake\_up\_process(thread1);

wake\_up\_process(thread2);

wake\_up\_process(thread3);

wake\_up\_process(thread4);

return 0;

}

static void simple\_exit (void) {

printk(KERN\_INFO "mutex\_module unloaded, Value: %d\n", race);

}

module\_init (simple\_init);

module\_exit (simple\_exit);