Lab 4: Kernel Synchronization

Like many user space programs, the Linux kernel is designed to run concurrently with itself. In this case, kernel code must be careful not to create race conditions and concurrency bugs by accessing shared data without protection.

This lab assignment, you will:

1. Create a race condition in a kernel module
2. Use kernel synchronization to resolve the race condition

You will need Raspberry Pi (or a Linux machine) to complete the lab.

Make sure that the name of each person who worked on these exercises is listed in the first answer, and make sure you number each of your responses so it is easy to match your responses with each exercise.

This lab has 8 points in total.

Submission:

When finished, ***submit your report in the PDF form to Moodle***. In addition to the report, please also submit the `race_module.c`, `atomic_module.c` and `mutex_module.c` files.

Required Exercises:

0. As the answer to the first exercise, list the names of the team members working on this lab.

1. Create a kernel module called `race_module` that creates a team of threads using `kthread_create`, one on each core, that call a thread function and return.

You can pin a kernel thread to a core using `kthread_bind`. To verify that pinning thread is successful, you can use `smp_processor_id` or `get_cpu` to get the CPU that a thread is executing on. Print the CPU id of each thread in the kernel log.

Now create a race condition in this kernel module, so that we can later solve it. Declare a **global** `int` called `race` with the `volatile` qualifier, as so:

```
volatile int race = 0;
```

The volatile qualifier tells the compiler that this variable will be modified outside of the current execution context, and has the effect of disabling certain optimizations.

Within the thread function for each of your threads, write a for loop that increments the variable race one million (1,000,000) times, as so:

```
#define iters 1000000
for( i=0; i<iters; i++){
        race++;
}
```

In module's exit function, print out the value of race. Leave the answer to this exercise blank.

2. Execute your code three times, what is the value of race? If you get 4,000,000 consistently then you have not successfully created a race condition. Make sure your variable is declared as volatile and that your threads are executing simultaneously (you may need to use `wake_up_process` to make sure the threads are running). As the answer to this exercise, take the **screenshots of three different outputs** of your module execution.

3. Make a new copy of your kernel module, name it as `atomic_module` and replace your global integer with a global `atomic_t` type, which is defined in `include/linux/types.h`. This is an opaque type that is only accessed through special mutators and accessors. Initialize this new type with the function `atomic_set()`, increment it with the function `atomic_add()`, and access it's value with the function `atomic_read()`. The function prototypes are found in `include/asm-generic/atomic.h`.

Run your code three times. What is the new value of race? If you get anything other than 4,000,000 you have not successfully resolved your race condition.

Have your threads print a statement to the system log before they start their for loop and when they finish. Use kernel dmesg timestamp ([x.y] means x sec and y microsec) as a crude timestamp to determine how long it takes your code to execute. As the answer to this exercise, take the **screenshot of one execution** of your module and calculate: how long does it take?

4. Make another copy of your original kernel module `race_module` and name it as `mutex_module`. This time, rather than modifying your global integer to be atomic, we will use a mutex to protect it. Use the macro `DEFINE_MUTEX(mutex_name)` to statically declare a mutex at the global scope and use the functions `mutex_lock(&mutex_name)` and `mutex_unlock(&mutex_name)` to protect access to the race variable.

Again, use kernel print statements as a crude timestamp for your thread functions. As the answer to this exercise, first take the **screenshot of one execution** of your module and calculate: how long does it take your mutex code to complete? Then answer the following questions: given the time difference between using atomic variables versus mutexes, why are mutexes useful? Give an example where you might prefer a mutex over an atomic variable.