# Kernel Modules to list all processes and their states

## Objective

You will learn how to develop and use kernel modules and character devices. You will also learn how the Linux OS maintains list of processes and their execution states.

## Part A: Accessing your Virtual Machines

This assignment will be carried out on a Linux virtual machine. Each student has been assigned a VM with a unique IP address. The login credentials have been emailed to you. You can either use the VM assigned to you or you can use your own Linux VM on your personal computer.

Instructions to access your virtual machine remotely

## Part B: Learn how to write and execute a Linux kernel module.

Kernel modules allow the code to be dynamically added to the kernel, dynamically. Try compiling and inserting the kernel modules hello.c and hellon.c from kernel module slides.

1. Save the Makefile, hello.c, and hellon.c in your home directory. The files hello.c has initialization and cleanup functions that are invoked upon loading and unloading the kernel module. The second file hellon.c is similar, except that it takes command-line arguments with insmod command to print a custom message.
2. Compile the kernel module:

   ```
   make
   ```

3. Load the kernel module:

   ```
   sudo insmod hello.ko
   ```

   If the module was successfully inserted, you will see "mymodule: Hello World!" message in kernel log. Use commands "dmesg" or "cat /var/log/kern.log" to see the kernel log messages.

4. Unload the kernel module:

   ```
   sudo rmmod hello
   ```

   Upon removing the kernel module, use the "dmesg" command to check if kernel log prints the message "mymodule: Goodbye, cruel world!!"

5. For hellon.c, module insertion command is `sudo insmod hellon.ko whom=class howmany=10` (or any other argument values you want to give). Removal is the same as for hello.c.

## Part C: Learn how to write a miscellaneous character device driver

- Follow this example to learn how to create a miscellaneous character device in Linux kernel.
- Modify the example to implement the `.read` file operation to return "Hello World!" message to a user

space program that invokes the `read()` system call on your miscellaneous device.

# Part D: How to browse Linux source code

There are a couple of ways you can browse linux source code.

- Use the Linux cross reference website
- Download the entire linux source code to your VM (or any other Linux computer) and use the `cscope` tool to browse. (slightly faster as you get familiar with cscope).
  - `wget https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.19.80.tar.xz` (replace with whatever kernel version you are developing your module against)
  - `sudo apt install cscope`
  - Cscope tutorial

# Part E: Write a kernel module to return list of processes to user space via a character device

Implement a kernel module that creates a `/dev/process_list` character device. The character device should support the `read()` operation. When the `read()` system call is invoked on your character device from a user space process, your kernel module should return to the following information about all currently running processes:

- process ID
- parent process ID
- the CPU on which the process is running
- its current state.

Note that the state field in task_struct can be -1, 0 or greater than 0. A state value of 1 or more indicates a combination (bitwise OR) values listed here.

So you will have to parse the `task->state` value to figure out which of the states apply to a process. You can pass the raw state value (as a `long` value) from kernel to user space and decode the state values in your user space program.

Beware that bugs in kernel code may either crash your kernel immediately or may have no immediate visible effect, but may have a delayed effect. Therefore, you cannot assume that the thing you did most recently is necessarily the cause of a crash.

# Part F: Write a user space program to retrieve the list of processes from the kernel module

Implement a user-space C program that opens your character device and outputs the list of processes retrieved from your character device.

One such application could be written as follows (please fill in the missing code):

```
char *buffer;

/* allocate memory for character buffers HERE before you use them */
```

```
fd = open("/dev/process_list", O_RDONLY);
/* check for errors HERE */

while(!some termination condition)
{
        bytes_read = read(fd, buffer, buffer_length);
        /* check for errors HERE. Exit loop if all processes have been retrieved. */
        /* print the output you have read so far. */
}

close(fd);
```

Your program's output could be as follows:

```
PID=1 PPID=0 CPU=4 STATE=TASK_RUNNING
PID=2 PPID=0 CPU=2 STATE=TASK_INTERRUPTIBLE
PID=10 PPID=2 CPU=0 STATE=TASK_UNINTERRUPTIBLE
PID=16434 PPID=16424 CPU=10 STATE=TASK_DEAD
PID=14820 PPID=16424 CPU=8 STATE=TASK_WAKEKILL,TASK_UNINTERRUPTIBLE
...and so forth
```

# Grading Guidelines

```
Part A, B, C = 20
Part D, E, F = 80

Total = 100
```

# References

- Example kernel modules hello.c and hellon.c
- Kernel Cross Reference
- for_each_process() macro can help you iterate over all processes in the system.
- `struct task_struct` definition can be found here
- Process states are defined here
- The function task_cpu() can help you extract the currently assigned CPU for a process.
- Introductory material on Linux Kernel
    - Chapters 1 and 2 of the following online book provide a good introduction to the kernel, though with a bias towards device-driver development. http://lwn.net/Kernel/LDD3/
    - For more kernel programming help, just google "Linux Kernel" and you'll get lots more.