Kernels

* The core of an operating system, manages the most critical resources
* Works as a bridge between application software and the computer’s hardware

Kernel vs Application Programming

* No memory protection – so when you write an application, you’re messing with all the memory in your kernel space, whereas normal c code is in its own environment (segmentation faults etc). So basically you can fuck shit up
* Easily crash the system
* Very hard to debug
* Sometimes no preemption
* Can hog the CPU
* No libraries – so no printf, fopen
* No access to files
* Direct access to hardware

Kernel modules

* Code that can be loaded and unloaded from the kernel on demand
* Have .ko extension
* Typically reside inside /lib/modules/<something …
* Kernel modules offer an easy way to extend the functionality of the base kernel without having to recompile the kernel again
* Most of the drivers are implemented as Linux Kernel modules
  + When those drivers are not needed, we can unload only that specific driver, which will reduce the kernel image size
* Advantages
* There is no necessity to rebuild the ernel when a new kernel functionality is added
* If you have something bad, you can try to unplug it from the kernel or reboot your system
* Utilities
  + Lsmod: lists out every loaded kernel module on a system
  + Modinfo: display info about a kernel module
  + Insmod: insert a module into the kernel “sudo insmod modulename.ko”
  + Rmmod: remove a module from the kernel “sudo insmod modulename”

Dmesg

* Slide 32
* Modules cannot directly print to the screen like printf() can. But they can log info and warnings that are eventually displayed on the screen
* **The ring buffer** stores info about hardware, device driver initialization, and messages from kernel modules that take place during system startup.
  + Dmesg | tail -10
    - Displays the last 10 messages from the ring buffer

Note

* All class examples are tested on Linux Kernel 5.10
* Recommended: Ubuntu 20.04 or Debian 11
* Make sure my system can run the example from slide 39

What does a kernel module look like? Lecture 11 Slide 8

* The headers are common headers that you need for writing kernel modules
* The modules start with 2 basic functions
  + Module\_init: the first function block that the module will execute. Takes in a pointer to the starting function of the module.
  + Module\_exit: tells which function block to execute when we unload the module.
* Makefile: slide 9
* Licensing: it is good practice to define these fields
  + Module\_license
  + Module\_author
  + Module\_description

Slide 12: set up machine for kernel programming

--------- new stuff--------

Read and write a /proc file (slide 17)

* Ex:
  + Proc file: /proc/procbuf
  + Read and write to a kernel buffer from user space
    - Echo “Cougs” > /proc/procbuf
    - Cat /proc/procbuf
* To move data between kernel/user space:
  + Copy\_to\_user()
    - Copies a block of data from the kernel into user space
    - Accepts a pointer to a user space buffer, a pointer to a kernel buffer, and a length defined in bytes
    - Returns zero on success or non zero to indicate the number of bytes
  + Copy\_from\_user()
    - Copies a block of data from user space into a kernel buffer
    - Accepts a destination buffer, a source buffer, and a length in bytes
* Linux provides a special struct proc\_ops, which has proc\_read and proc\_write

--------- new stuff ----------

Linux kernel lists

* Linux kernel list is a widely used data structure in linux kernel
  + It’s a linked list with next and prev ponters
* 3-5 slide 8 for list of functionality
* 3-5 slide 12 for list of commands

Kernel timer

* When you want to do some operations, but you don’t want then to start immediately
* Uses time units called **jiffies** rather than seconds
  + Use msec\_to\_jiffies() to convert ms to jiffies
* Slide 14 for example program using timer
  + Struct timer\_list is defined in timer.h
  + When calling timer\_setup() pass in a reference to your timer struct, a pointer to a function, and a flag for something
  + Mod\_timer() actually triggers the timer. Pass in current time (jiffies) + your delay
* Slide 15 example
  + This time inside of the callback function we call mod\_timer again, so it reinitializes the timer, making a periodic timer that keeps repeating
  + This recursive timer call wont stop until you unload the timer using del\_timer()
  + Say you want to only run the timer 5 times. Then you need a static variable for count, then in callback() only reinitialize mod\_timer if count is less than 5.

Scheduling: workqueue

* Use a workqueue to add a task to the scheduler
  + The kernel then uses the completely fair scheduler to execute work within the queue
* Slide 16 for implementation of worqueue in a kernel program

Kernel spinlock

* The linux kernel is a preemptible kernel
  + Contexts can be interrupted
* You need to protect your data structures through the use of appropriate locks
* Spinlocks lock up the CPU that the code is running on, taking 100% of its resources
* Should only use the spinlock mechanism around code, which is likely to take no more than a few milliseconds to run so it will not noticeably slow anything down from the user’s perspective.
* Ex slide 18
  + Define spinlock macro
  + The flags variable stores the current state of your system
  + Do some code that does not take very long
  + Then get back the previously saved state of your system

**Solution to In class 4** is on slide 21