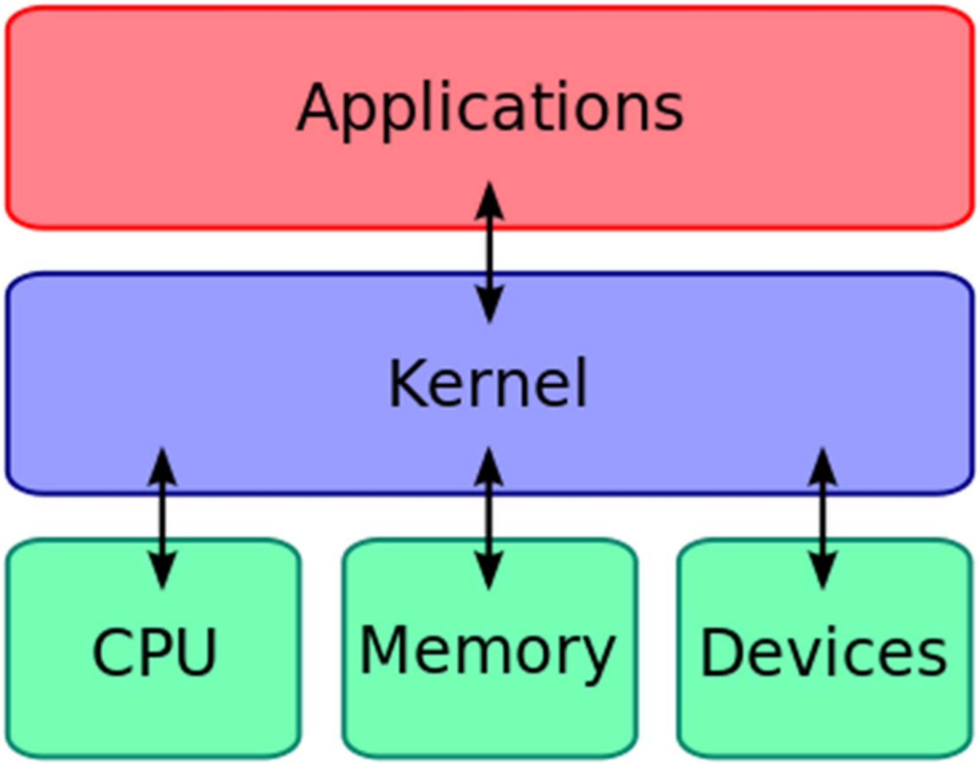
Kernel programming

Kernel

part of OS that directly operate on user application and sys hardwares

- Kernel = computer program that connects the user applications to the system hardware
- Handles:
  - Memory management
  - CPU scheduling (Process and task management)
  - Disk management
  - User access to other I/O devices (e.g., network card)

Kernel



Source: wikipedia.org

Kernel modules

- Object file that contains code to extend the kernel's functionality
- Why do we need them? Why not include all possible functionality in the kernel directly?
  - so include procedures not used often but takes up memory
  - The kernel code lies in main memory
  - Kernel should be minimal
  - Avoid functionality bloating
  - For each new functionality added => recompile kernel, reboot, .. ugh!
  - Instead, develop modules separately, load as needed
  - Modularity => Better chance to recover from buggy new code without a complete kernel crash!

Why should I bother?

- Because it’s cool!

- Better understanding on how the OS works
- Write awesome extensions to the OS
- Write your own device drivers!

Linux kernel modules

- Basic utilities:
  - `insmod`: to load a module
  - `rmmod`: to unload a module
- The `modprobe` utility manages loading modules
  - More complex, deal with module dependencies
- Module objects: `.ko` files

Kernel module example

note
+ every header file prefixed with linux/
+ no include <stdio.h> …
+ entry point module_init(fp) instead of main
    + usually for setting up params
+ exit point module_exit(fp)

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>

MODULE_DESCRIPTION("My kernel module");
MODULE_AUTHOR("John Doe");
MODULE_LICENSE("GPL");

static int mymodule_init(void) {
        printk( KERN_DEBUG "Hello world!\n" );
        return 0;
}

static void mymodule_exit(void) {
        printk( KERN_DEBUG "I'm outta here\n" );
}

module_init(mymodule_init);
module_exit(mymodule_exit);
```

Printing messages

printk outputs to logfile instead of terminal

- Use printk
  - e.g., `printk( KERN_DEBUG “Hello world\n");`

KERN_ALERT -> print to console

- Dude, where’s my output?
  - Not displayed at stdout
  - Can be retrieved from the system logs
  - Use `dmesg` command

/var/log/messages

Compiling a module

- Different than a regular C program
- Must use different headers
- Must not link with libraries. Why? linux libraries already included other libraries cant be included
- Must be compiled with the same options as the kernel in which we want to load it
- Standard method: kbuild
  - Two files: a Makefile, and a Kbuild file

Example

- Makefile:

```
KDIR=/lib/modules/`uname -r`/build
kbuild:
   make -C $(KDIR) M=`pwd`
clean:
   make -C $(KDIR) M=`pwd` clean
```

-C : change to directory before doing anything else

- Kbuild file:

```
EXTRA_CFLAGS=-g
obj-m  = mymodule.o
```

specifies object files in the directory.
obj-m stands for object for module

m -> module
y -> built-in

Loading/unloading a kernel module

- As root, or using sudo
- Loading:
  - `insmod mymodule.ko`
- Unloading:
  - `rmmod mymodule.ko` (or: `rmmod mymodule`)
- Entry point:
  - `module_init(mymodule_init);`
  - `module_exit(mymodule_exit);`

Debugging a kernel module

- More complicated than a regular program
- A bug in a module can lead to the whole OS malfunctioning
- Buggy module: can lead to a "kernel oops"
- Avoid reboot cycles => use VM for CSC369!
- Do not develop modules directly on your Linux box without a VM! – painfully slow!
- For A1, use rudimentary (yet efficient) method: printk statements

Debugging a kernel module

- You can use a debugger, but not very useful
  - Simple bugs can be tracked easily with printks
  - Use ksymoops utility
- Complex bugs – not even a debugger will help as much
  - Need to know in depth the OS structure
  - Multiple contexts, interrupts, VM, etc.
- Kernel oops message can be translated using ksymoops (memory locations, backtrace, etc.)

Linux kernel API – some differences

- Different headers – make sure to include them!

- Success/failure conventions:
  - 0 == success
  - Non-zero == failure (-ENOMEM, -EINVAL, etc.)
  - See `<include/asm-generic/errno-base.h>` and `<include/asm-generic/ernno.h>`

- Memory allocation: `kmalloc/kfree`

```
#include <linux/malloc.h>
if(!(string = kmalloc(len+1, GFP_KERNEL))) {
      return -ENOMEM;
}
...
kfree(string);
```

error shown in negative numbers

Strings and printing

- Standard string functions:
  - strcmp, str(n)cpy, str(n)cat, memcpy, etc.
- Same header: <string.h>

  only library not in linux allowed to be included
- Printing: `printk`, defined in `<linux/kernel.h>`
- Similar syntax, plus category of message:

  printk(KERN_WARNING "Uh-oh, you better check this: %s\n", buff);

  printk(KERN_DEBUG "This buffer looks spooky: %s\n", buff);

```
#define KERN_EMERG  "<0>"   /* system is unusable    */
#define KERN_ALERT  "<1>"   /* action must be taken immediately */
#define KERN_CRIT   "<2>"   /* critical conditions       */
#define KERN_ERR    "<3>"   /* error conditions      */
#define KERN_WARNING    "<4>"   /* warning conditions       */
#define KERN_NOTICE "<5>"   /* normal but significant condition */
#define KERN_INFO   "<6>"   /* informational       */
#define KERN_DEBUG  "<7>"   /* debug-level messages      */
```

Synchronization: spinlocks

- Busy-waiting synchronization: `spinlock_t` type
  - `spinlock_t myspinlock = SPIN_LOCK_UNLOCKED;`
- Operations:
  - `spin_lock_init(&myspinlock)`
  - `spin_lock/unlock(&myspinlock)`
- Can also use read/write spinlocks: `rwlock_t`
  - `rwlock_init(), read_lock(), write_lock()`
- Check out: <include/linux/spinlock.h>
- Will learn more about synchronization later in the course!