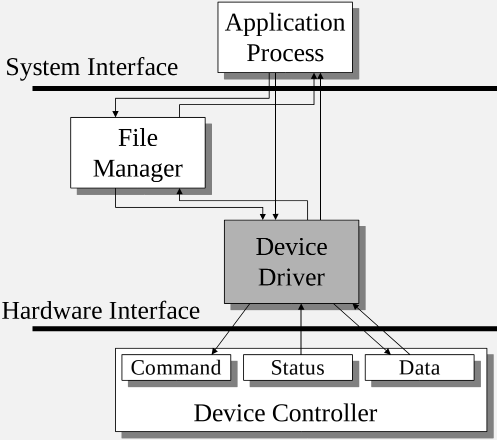
Device Management

Learning Objectives

- Understand the concept of devices and their relationship to the operating system
- Understand device I/O concepts such as direct, memory-mapped and direct memory access (DMA), polling and interrups
- Understand the concept of a compute-bound versus I/O bound process
- Understand the concept of reloadable device drivers
- Develop simple Linux modules

Device Management



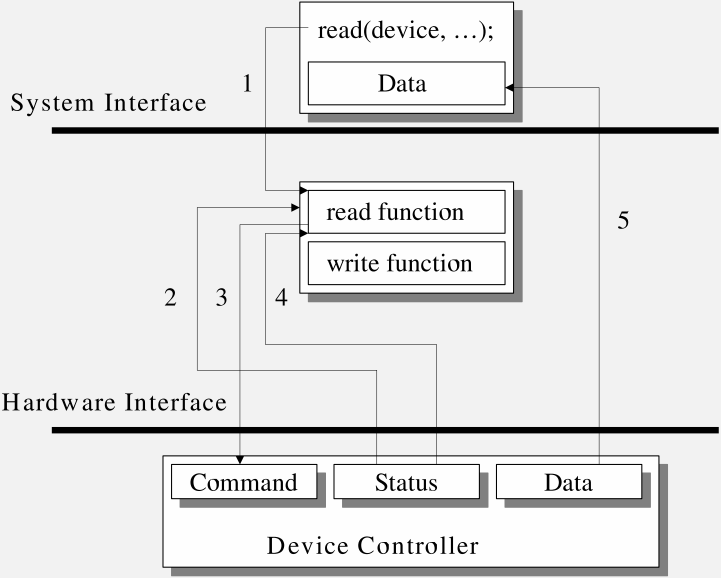
Commonly Used Device Types

- Communication devices. Examples: Serial communication devices using Universal Asynchronous Receiver Transmitter (UART) with RS 232 serial communication protocol, Universal Serial Bus (USB) (version 1 was 1.5 to 12 Mbps, version 2 is up to 480 Mbps, version 3 is up to 5 Gbps and 10Gbps in future), IEEE 1394 Firewire (100/200/400 Mbps and up to 3.2 Gbps in future), Thunderbolt (10 Gbps per channel).
- Sequentially accessed storage devices. Examples: Digital Audio Tapes (DAT) can store up to 40GB. Used for backups.
- Randomly accessed storage devices. Examples: Magnetic hard disks, Compact Disk-Read Only Memory (CD-ROM, around 700MB), Digital Versatile Disk (DVD, 4.7GB to 17GB), Jump drives, Solid State Drives (SSD).

Types of Device I/O

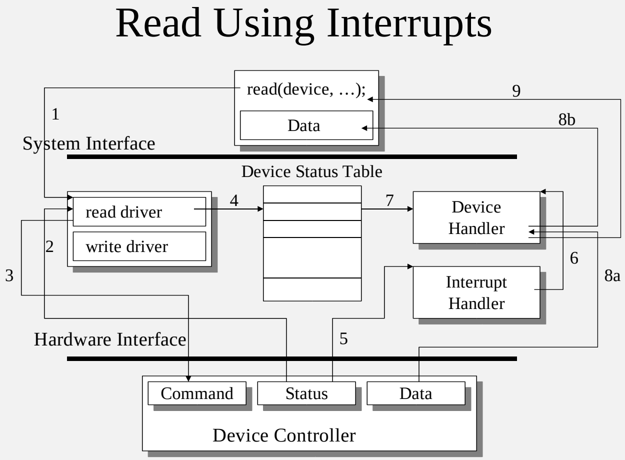
- Direct I/O.
  - with polling.
  - interrupt driven.
- Memory-mapped I/O.
  - with polling.
  - interrupt driven.
- Direct Memory Access. (DMA)

I/O with Polling



I/O with Interrupts

Device Management



top-half bottom-half

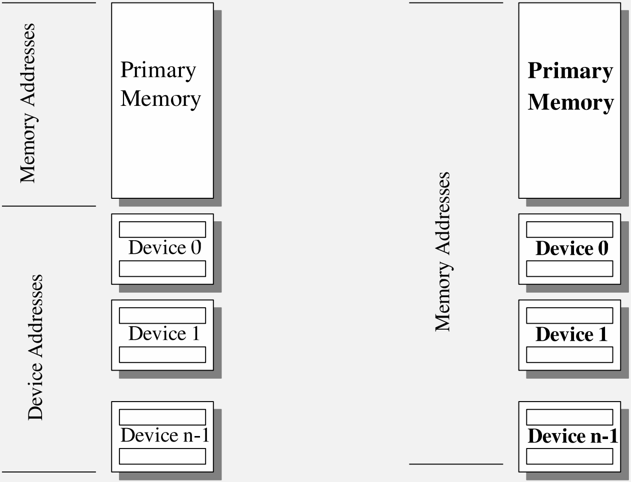
Polling versus Interrupts

- With only one process at a time, polling-based I/O would tend to be more efficient.
- With multiple processes, interrupt-based I/O would result in smaller average time for the processes.

Memory Mapped I/O

- Registers in devices are associated with logical memory addresses rather than having specialized device addresses.
- Memory-mapped I/O eliminates special I/O machine instructions to read/write to device registers.
- Memory-mapping is accomplished at the bus level by the decoding logic.

Memory Mapped I/O



Example of Interrupt Mappings

```
IRQ        CPU0                Device
  0:  223702571   XT-PIC      timer
  1:     358416   XT-PIC      keyboard
  2:          0   XT-PIC      cascade <-- connected to 2nd XT-PIC
  5:          0   XT-PIC      usb-uhci
  7:      13424   XT-PIC      soundblaster
  8:          1   XT-PIC      rtc
 11:   66502297   XT-PIC      eth0
 12:    9089768   XT-PIC      PS/2 Mouse
 14:    4513089   XT-PIC      ide0
```

`/proc/interrupts` shows the interrupt mappings under Linux

Traditional Programmable Interrupt Controllers (XT-PIC) can handle 8 Interrupt Request (IRQ) lines. Two PICs are cascaded together, with the slave PIC connected to line 2 of the controlling PIC, leaving 15 usable IRQs.

Interrupt Mappings on a Multi-processor System

```
           CPU0       CPU1       CPU2       CPU3
  0:    4948224          0          0          0    IO-APIC-edge  timer
  1:       2944          0          0          0    IO-APIC-edge  keyboard
  2:          0          0          0          0          XT-PIC  cascade
  8:          1          0          0          0    IO-APIC-edge  rtc
 12:       8551          0          0          0    IO-APIC-edge  PS/2 Mouse
 14:     396575          0          0          1    IO-APIC-edge  ide0
 15:         23          0          0          0    IO-APIC-edge  ide1
 16:          0          0          0          0   IO-APIC-level  usb-uhci
 17:    1191765          0          0          0   IO-APIC-level  eth0, Intel ICH4
 18:      24805          0          0          0   IO-APIC-level  usb-uhci, eth1
 19:          0          0          0          0   IO-APIC-level  usb-uhci
 23:          0          0          0          0   IO-APIC-level  ehci-hcd
NMI:          0          0          0          0
LOC:    4948381    4948380    4948380    4948380
ERR:          0
MIS:          0
```

- On Multi-core/processor systems, the I/O Advanced Programmable Interrupt Controller (APIC) is used. Each APIC can support 24 IRQs. On Intel CPUs, each CPU has a local APIC and there is a global I/O APIC.
- The interrupt handling can be balanced by programming the APICs. Google “irqbalance" to learn more about a utility that does load-balancing.

Example: Memory mapped I/O


```
0000-001f : dma1
0020-003f : pic1
0040-005f : timer
0060-006f : keyboard
0070-007f : rtc
0080-008f : dma page reg
00a0-00bf : pic2
00c0-00df : dma2
00f0-00ff : fpu
01f0-01f7 : ide0
0220-022f : soundblaster
02f8-02ff : serial(auto)
0330-0333 : MPU-401 UART
0378-037a : parport0
037b-037f : parport0
03c0-03df : vga+
03f6-03f6 : ide0
03f8-03ff : serial(auto)
0cf8-0cff : PCI conf1
d000-d03f : 3Com Corporation 3c905 100BaseTX [Boomerang]
d400-d41f : Intel Corp. 82371AB/EB/MB PIIX4 USB
d400-d41f : usb-uhci
d800-d80f : Intel Corp. 82371AB/EB/MB PIIX4 IDE
d800-d807 : ide0
d808-d80f : ide1
e400-e43f : Intel Corp. 82371AB/EB/MB PIIX4 ACPI
e800-e81f : Intel Corp. 82371AB/EB/MB PIIX4 ACPI
```

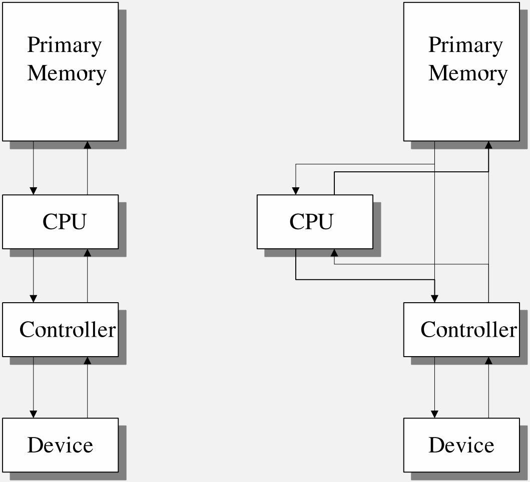
`/proc/ioports` shows the memory mapped I/O ports under Linux

Direct Memory Access (DMA)

- Once the driver has initiated an I/O operation, a DMA controller can read/write to main memory without software intervention.
- DMA frees up the CPU from copying of data from the controller registers or buffer. This leads to better performance.
- DMA controllers and the CPU may, however, compete for the bus.

Direct Memory Access (DMA)

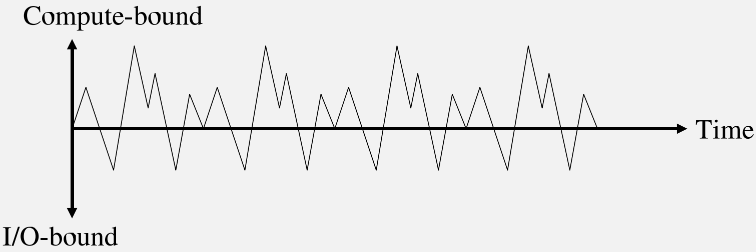
Device Management



Buffering

- Buffering improves I/O performance by allowing device managers to keep slower I/O devices busy when process do not need I/O.
- Single buffering.
- Double buffering.
- Circular buffering.
- I/O-bound versus Compute-bound processes. The effect of buffering depends a lot on the characteristics of the process.

Compute versus I/O Bound processes



How do we measure compute bound versus i/o bound? Use strace in Linux.

Examining Process Characteristics

- strace is a nifty utility for finding out more about the behavior of a process.

```
kohinoor:strace -c mycp test3.data tmp
execve("./mycp", ["mycp", "test3.data", "tt"], [/* 36 vars */]) = 0
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 56.13    0.367676         714       515           read
 42.15    0.276133         538       513           write
  1.52    0.009954        9954         1           creat
  0.10    0.000636         127         5         2 open
  0.04    0.000287          48         6           mmap
  0.02    0.000114          29         4           mprotect
  0.01    0.000080          80         1           stat
  0.01    0.000067          67         1           munmap
  0.01    0.000060          15         4           close
  0.00    0.000014          14         1           personality
  0.00    0.000013          13         1           geteuid
  0.00    0.000012          12         1           getuid
  0.00    0.000012          12         1           getgid
  0.00    0.000012          12         1           getegid
------ ----------- ----------- --------- --------- ----------------
100.00    0.655070                  1055         2 total
```

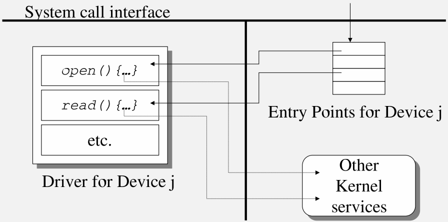
- Try `strace -r -T mycp test3.data tmp` and use the script in the examples folder `ch5/convert-strace-log` to generate plotting data from the output of the `strace` command.
- `strace -r -T` gives a time-stamp as the process enters a system call. It also prints the amount of time spent in the system call after each call.

Device Drivers

- *Application Programming Interface (API).* Conflicting goals of being able to control specific aspects of the device versus having a consistent interface for all drivers.
- The operating system tries to hide the details of the devices by using an interface common to all types of devices. The interface provides an abstract I/O paradigm. Typical operations include open, close, read, write and a general way of doing device specific operations (ioctl in Unix/Linux).
- Types of devices. *Block-oriented* versus *Character-oriented* devices. E.g. Classify these devices: network interface, keyboard, CD/DVD drive, USB key, disk drive, floppy disk drive, mouse, tape drive, printer, sound card. Other types of classifications: random-access versus sequential-access.

Kernel Interface for Device Drivers

- *Kernel Interface*. Device drivers are part of the operating system because they need to execute privileged instructions. Two ways of adding device drivers to an operating system.
  - *Built-in drivers.* Add device driver code to the operating system source code and recompile the operating system. The machine has to be then rebooted with the new version of the operating system.
  - *Reconfigurable device drivers.* Use dynamic binding of the compiled driver to the operating system code. Allows device drivers to be added on the fly without recompiling or rebooting the operating system.



Device Drivers (cont'd.)

- Process—Driver—Controller coordination.
- Optimization of I/O performance. Buffering is one common technique. Examples of where buffering is used: character and sequential access devices, printers. For random-access devices, the driver can attempt to optimize by rearranging the order in which multiple requests are performed.

Linux/Unix devices

- When the user program calls the driver, it performs a system call. The kernel looks up the entry point for the device in the block or character indirect reference table (the *jump table*) and then calls the entry point. The logical contents of the jump table are kept in the file system in the /dev directory. The files in the `/dev` directory are special files (that can only be created with the mknod command).
- Device drivers are uniquely identified by their *major numbers*. A device driver may be controlling a number of physical and virtual devices; the individual device is accessed via the *minor number*.
- Each entry point in the driver is registered at runtime by defining a structure of type file_operations, with function pointers for the defined routines. The structure is then passed to the kernel with a call to either register_chrdev(...) or to register_blkdev(...) to bind the links.

Device Files in Linux/Unix


```
$ ls -l /dev
crw-------  1 root root      5,   1 Aug 29 17:06 console
drwxr-xr-x  4 root root         100 Aug 29 17:06 cpu
brw-rw----  1 root disk      7,   0 Aug 29 17:06 loop0
brw-rw----  1 root disk      7,   1 Aug 29 17:06 loop1
crw-rw----  1 root lp        6,   0 Aug 29 17:06 lp0
brw-rw----  1 root disk      9,   0 Aug 29 17:06 md0
brw-rw----  1 root disk      9,   1 Aug 29 17:06 md1
crw-r-----  1 root kmem      1,   1 Aug 29 17:06 mem
crw-rw-rw-  1 root root      1,   3 Aug 29 17:06 null
crw-rw-rw-  1 root root    195,   0 Aug 29 17:07 nvidia0
crw-rw-rw-  1 root root    195, 255 Aug 29 17:07 nvidiactl
crw-r-----  1 root kmem     10, 144 Aug 29 17:06 nvram
crw-rw----  1 root lp       99,   0 Aug 29 17:06 parport0
crw-rw-rw-  1 root root      1,   8 Aug 29 17:06 random
crw-rw----  1 root root    254,   0 Aug 29 17:06 rtc0
brw-rw----  1 root disk      8,   0 Aug 29 17:06 sda
brw-rw----  1 root disk      8,   1 Aug 29 17:06 sda1
brw-rw----  1 root disk      8,   2 Aug 29 17:06 sda2
brw-rw----  1 root disk      8,   3 Aug 29 17:06 sda3
brw-rw----+ 1 root cdrom    11,   0 Aug 29 17:06 sr0
lrwxrwxrwx  1 root root          15 Aug 29 17:06 stderr -> /proc/self/fd/2
lrwxrwxrwx  1 root root          15 Aug 29 17:06 stdin -> /proc/self/fd/0
lrwxrwxrwx  1 root root          15 Aug 29 17:06 stdout -> /proc/self/fd/1
lrwxrwxrwx  1 root root           4 Aug 29 17:06 systty -> tty0
crw-rw-rw-  1 root tty       5,   0 Sep 14 22:18 tty
crw--w----  1 root root      4,   0 Aug 29 17:06 tty0
crw--w----  1 root root      4,   1 Aug 29 17:07 tty1
crw-rw-rw-  1 root root      1,   9 Aug 29 17:06 urandom
crw-rw-rw-  1 root root      1,   5 Aug
```

Device Files in Linux/Unix (contd.)

Some things to note in the listing of device special files in the /dev directory.

- The first character in the permissions, b or c, represents whether the device is block or character oriented.
- The major and minor numbers (5th and 6th column in the listing) are reserved for certain device types. Some major numbers are free to new devices. For example: major number 3 is for the IDE driver. See the file `Documentation/devices.txt` in the Linux kenel source for the mapping of major/minor numbers to device types.
- The console device is the same as the tty device.

A simplified device driver framework

Assumptions:

1. getBlock(device, buffer) and putBlock(device, buffer) are system calls that may be called by an application program.
2. A block is 1024 contiguous bytes pointed to by the buffer argument.
3. DMA controller can transfer block from device buffer directly to memory pointed to by buffer provided by calling application with memcpy call.
4. Device identified by kernel with major and minor integer values according to typical Unix systems.
5. Device controller command register, status register, and block buffer are memory mapped (i.e. they can be read and written with an address). The device controller has BUSY and DONE flags with the following interpretation.

busy	done	state
false	false	device is idle and ready for a new command
true	false	device is busy with command
false	true	command is done but data has not been transferred
true	true	invalid state

```
/*
This solution was prepared by Sam Siewert and provided
by Gary Nutt.  Modified by Amit Jain.
*/
/* Device Identification */
struct dev_spec {
  unsigned short major;
  unsigned short minor;
}

#define BUSYFLAG 1
#define DONEFLAG 2
#define BLKSIZE 1024

struct dev_status {
  unsigned short status;
  void *apl_return_addr;
  void *apl_buffer_addr;
}

struct dev_param {
  void *data_addr;
  void *status_addr;
  void *cmd_addr;
}

struct dev_status dev_status_table[NUM_MAJOR_DEV];
struct dev_param dev_param_table[NUM_MAJOR_DEV];
```

```
int getBlock(struct dev_spec *device, void *buffer)
{
  char cmd;
  switch(device->major) {
  case 0: ...  break;
  case 1: ...  break;
  case 2: ...  break;
  /* e.g. IDE hard-disk in Linux */
  case 3:
    switch(device->minor) {
        /* minor device is particular drive and partition */
        case 0:
        cmd = GETBLK;
        /* need to check if the device is free to execute a new command */
        while (dev_status_table[3].status != 0); /* busy wait */

        memcpy(dev_param_table[3].cmd_addr, &cmd, 1);
        dev_status_table[3].status =
                          (dev_status_table[3].status)|BUSYFLAG;
        dev_status_table[3].apl_return_addr = get_return_from_stack();
        dev_status_table[3].apl_buffer_addr = buffer;
        /* yield the CPU, blocked for IO */
        sched_yield(); /* supported in Posix */
        break;
    }
    break;
    /* ... */
  }
}
```

```
int putBlock(struct dev_spec *device, void *buffer)
{
  char cmd;
  switch(device->major) {
    case 0: ...  break;
    case 1: ...  break;
    case 2: ...  break;
    /* e.g. IDE hard-disk in Linux */
    case 3:
      switch(device->minor) {
        /* minor device is particular drive and partition */
        case 0:
        cmd = PUTBLK;
        /* need to check if the device is free to execute a new command */
        while (dev_status_table[3].status != 0); /* busy wait */
        memcpy(dev_param_table[3].cmd_addr, &cmd, 1);
        dev_status_table[3].status =
                             (dev_status_table[3].status)|BUSYFLAG;
        dev_status_table[3].apl_return_addr=get_return_from_stack();
        dev_status_table[3].apl_buffer_addr = buffer;
        /* yield the CPU, get blocked for I/O */
        sched_yield(); /* supported in Posix */
        break;
      }
      break;
    /* ... */
  }
}
```

```
void interrupt_handler(void)
{
  int i;
  unsigned short status;

  saveProcessorState();
  for(i=0;i<=LASTDEVICE;i++) {
    /* Assume that the busy flag is false and done flag is true
       after the completion of an I/O operation and before the
       data is transferred from the device controller to the buffer
       in the user space.
    */
    memcpy(&status, dev_param_table[i].status_addr, 1);
    /* Can drop the second part of the and clause below */
    if ((status&DONEFLAG) && !(status&BUSYFLAG)) {
         dev_status_table[i].status=status;
        device_handler(i);
    }
  }
  /* error if we get here */
}
```

```
void device_handler(int i)
{
  switch(i) {
    case 0: ...
    break;
    case 1: ...
    break;
    case 2: ...
    break;
    /* e.g. IDE hard-disk in Linux */
    case 3:
    /* The DMA transfer of the block happens below */
    // The following is for a getBlock, what's needed for a putBlock
    memcpy(dev_status_table[i].apl_buffer_addr,
        dev_param_table[i].data_addr, BLKSIZE);
    /* The controller clears the done flag to indicate that
       the device is again ready for the next command */
    dev_status_table[i].status = 0;
    returntoaddr();
    break;
    .
    .
    .
  }
}
```

Linux Modules

- Linux modules are pieces of code that be loaded into or unloaded from the kernel upon demand without having to reboot the system. Device drivers are a type of module that deals with hardware devices.
- Other examples of plugin use: Web browsers, Windows Media Player, Amarok etc.
- Since the kernel is written in C, you may ask how can a C program load/unload code on the fly? See examples in the plugins folder in the class examples: plugins
  - ex1: `plugin1.c`, `plugin2.c`, `runplug.c`
  - ex2: `Hello.c`, `Goodbye.c`, `Loader.h`, `Loader.c`
  - Also checkout examples in the folders `ex3` and `ex4`.

Linux Modules

- The command `modinfo` gives you info on the specified given module file.
- The command lsmod lists all currently loaded modules. (Or we can look at /proc/devices and /proc/modules)
- The command insmod allows the superuser to add a new module.
- The command rmmod allows the superuser to remove a module that is no longer in use.
- The utilities modprobe and depmod automate loading/unloading of modules under Linux.
- You may need to add `/sbin` to your PATH environment variable or prefix the commands with `/sbin` before your shell will find them.
- You will need to install the `kernel-devel` package on Fedora to be able to build a kernel module.
  `yum install kernel-devel`

Linux Module Programming

- The standard C library is not available to modules in kernel space. So a module can only use the functions that are already in the kernel.
- A list of all functions available in the kernel is in /proc/kallsyms
- Some basic string and other functions have been re-implemented in the kernel. Google for Linux kernel API.
- Name space pollution is a big concern since any module code becomes part of the kernel. A good strategy to use is to declare everything static that you can.
- If module code (or any kernel space code) dereferences a bad pointer, the results can range from annoying (having to reboot to get rid of the module) to disastrous. Memory violations in the kernel result in an oops, which is a major kernel error.
- There is no (easy) way to use floating point instructions in the kernel. *Just Don't Do It*.

The "Hello, World" Module

```
/* device-management/linux_device_drivers/hello/hello.c */
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>

/*MODULE_LICENSE("Proprietary");*/
MODULE_LICENSE("GPL");

static int __init hello_init(void) {printk("<1>Hello, world\n");
    return 0;}
static void __exit hello_cleanup(void) {printk("<1>Goodbye cruel
    world\n");}

module_init(hello_init);
module_exit(hello_cleanup);
```

All device driver usually implement an init and exit function. In older style code, these functions had a fixed name: `init_module` and `cleanup_module`. But now the init/exit functions can be named anything by using the `module_init` and `module_exit` macros. In addition, it may implement one or more of the functions listed in the `file_operations` structure, which is discussed later in this chapter.

The __init and __exit macros

- The __init macro causes the init function to be discarded and its memory freed once the init function finishes for built-in drivers, but not loadable modules.
- The __exit macro causes the omission of the function when the module is built into the kernel, and like `__init`, has no effect for loadable modules.
- Note that built-in drivers don't need a cleanup function, while loadable modules do. These macros are defined in linux/init.h and serve to free up kernel memory.

Building and Loading the “Hello, World” Module


```
[amit@localhost hello]$ ls
hello.c  Makefile

[amit@localhost hello]$ make
make -C /lib/modules/`uname -r`/build M=`pwd` modules
make[1]: Entering directory '/usr/src/kernels/4.2.3-200.fc22.x86_64'
  CC [M]  /home/amit/Documents/work/courses/cs453/lab/device-management/linux-\
           device-drivers/hello/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/amit/Documents/work/courses/cs453/lab/device-management/linux-\
           device-drivers/hello/hello.mod.o
  LD [M]  /home/amit/Documents/work/courses/cs453/lab/device-management/linux-\
           device-drivers/hello/hello.ko
make[1]: Leaving directory '/usr/src/kernels/4.2.3-200.fc22.x86_64'

[amit@localhost hello]$ sudo /sbin/insmod hello.ko
[amit@localhost hello]$ /sbin/lsmod | grep hello
hello                 16384  0
[amit@localhost hello]$ sudo /sbin/rmmod hello.ko
[amit@localhost hello]$ /sbin/lsmod | grep hello
[amit@localhost hello]$
```

Loading and Unloading the “Hello, World” Module


```
[root@kohinoor hello]# tail /var/log/messages
Sep 23 07:28:49 kohinoor sshd(pam_unix)[18098]: session closed for user amit
Sep 23 07:31:34 kohinoor sshd(pam_unix)[18310]: session opened for user amit by (uid=999)
Sep 23 08:12:17 kohinoor su(pam_unix)[18566]: session opened for user root by amit(uid=999)
Sep 23 08:12:40 kohinoor kernel: Hello, world

[root@kohinoor hello]# /sbin/rmmod hello

[root@kohinoor hello]# tail /var/log/messages
Sep 23 07:31:34 kohinoor sshd(pam_unix)[18310]: session opened for user amit by (uid=999)
Sep 23 08:12:17 kohinoor su(pam_unix)[18566]: session opened for user root by amit(uid=999)
Sep 23 08:12:40 kohinoor kernel: Hello, world
Sep 23 08:15:49 kohinoor kernel: Goodbye cruel world
[root@kohinoor hello]#
```

On some systems, `syslogd` has been replaced by `journald`. So change the above command to

```
journalctl -f

or

dmesg --follow
```

Module Licenses

```
/* From include/linux/module.h header file in the kernel source
 * The following license idents are currently accepted as indicating
 * free software modules
 *
 *  "GPL"             [GNU Public License v2 or later]
 *  "GPL v2"          [GNU Public License v2]
 *  "GPL and additional rights" [GNU Public License v2 rights and more]
 *  "Dual BSD/GPL"    [GNU Public License v2 or BSD license choice]
 *  "Dual MPL/GPL"    [GNU Public License v2 or Mozilla license choice]
 *
 * The following other idents are available
 *  "Proprietary"              [Non free products]
 *
 * There are dual licensed components, but when running with Linux
 * it is the GPL that is relevant so this is a non issue. Similarly
 * LGPL linked with GPL is a GPL combined work.
 *
 * This exists for several reasons
 * 1.   So modinfo can show license info for users wanting to vet
 *        their setup is free
 * 2.   So the community can ignore bug reports including
 *        proprietary modules
 * 3.   So vendors can do likewise based on their own policies
 */
```

The `file_operations` structure in Linux kernel

From the file `<kernel source>/include/linux/fs.h` for version 4.2.3.
Compare with your kernel source!

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
    int (*iterate) (struct file *, struct dir_context *);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*mremap)(struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, loff_t, loff_t, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
    int (*check_flags)(int);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
    ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
    int (*setlease)(struct file *, long, struct file_lock **, void **);
    long (*fallocate)(struct file *file, int mode, loff_t offset,
              loff_t len);
    void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
    unsigned (*mmap_capabilities)(struct file *);
#endif
};
```

The `file` structure in Linux kernel

From the file `<kernel source>/include/linux/fs.h` (for version 4.2.3)

```
struct file {
    union {
        struct llist_node   fu_llist;
        struct rcu_head     fu_rcuhead;
    } f_u;
    struct path     f_path;
    struct inode        *f_inode;   /* cached value */
    const struct file_operations    *f_op;

    /*
     * Protects f_ep_links, f_flags.
     * Must not be taken from IRQ context.
     */
    spinlock_t      f_lock;
    atomic_long_t       f_count;
    unsigned int        f_flags;
    fmode_t         f_mode;
    struct mutex        f_pos_lock;
    loff_t          f_pos;
    struct fown_struct  f_owner;
    const struct cred   *f_cred;
    struct file_ra_state    f_ra;

    u64         f_version;
#ifdef CONFIG_SECURITY
    void            *f_security;
#endif
    /* needed for tty driver, and maybe others */
    void            *private_data;

#ifdef CONFIG_EPOLL
    /* Used by fs/eventpoll.c to link all the hooks to this file */
    struct list_head    f_ep_links;
    struct list_head    f_tfile_llink;
#endif /* #ifdef CONFIG_EPOLL */
    struct address_space    *f_mapping;
} __attribute__((aligned(4)));  /* lest something weird decides that 2 is OK */
```

Linux Device Driver Examples

The following examples are in the linux_device_drivers directory in the examples. They are based on examples from the book *Linux Device Drivers* by *Rubini*, *Corbet* and *Kroah-Hartman* (O'Reilly publishers). You can find them in the class examples repository under the device-management/linux-device-drivers folder.

- example1
- example2
- example3
- example4

Linux Module Programming Summary (1)

/proc/kallsyms Names and addresses of all visible functions in the kernel.
/proc/ioports Memory mapped I/O device addresses.
/proc/devices Major numbers and names corresponding to device drivers loaded currently.

`#include <linux/module.h>`
: Required headers. It must be included by a module source.

```
MODULE_AUTHOR("author");
MODULE_DESCRIPTION("description");
MODULE_SUPPORTED_DEVICE("device");
```

Place documentation on the module in the object file.

```
MODULE_LICENSE("license");
```

Set a license for the module. Use “GPL” (or compatible open source licenses) to avoid tainting the kernel. Use “Proprietary” for non-free modules. If you use “Proprietary” license then other kernel developers will ignore errors in your driver since it is not open source. Recommended to use “GPL” or seek legal advice!

```
#include <linux/init.h>
module_init(init_function);
module_exit(exit_function);
```

Newer mechanism for marking a module's initialization and cleanup functions.

Linux Module Programming Summary (2)

try_module_get(THIS_MODULE)
put_module(THIS_MODULE)

Macros that act on the usage count for a module.

#include <linux/sched.h>

One of the most important header files. Contains definitions of much of the kernel API used by drivers. Contains the process descriptor structure: struct task_struct.

#include <linux/fs.h>

The file system header is required for writing device drivers. This contains the `file_operations` and the file structure declarations.

struct task_struct *current;
current->pid
current->comm

The current process. Its process id and its command name.

#include <linux/kernel.h>
int printk(const char *fmt, ...);

The analogue of *p*rintf for kernel code.

#include <linux/slab.h>
void *kmalloc(unsigned int size, int priority);
void kfree(void *object);

Analogue of *m*alloc and *f*ree for kernel code. Typical value of priority is `GFP_KERNEL` (General Free Page in the kernel space).

Linux Module Programming Summary (3)

kdev_t inode->i_rdev

The device "number" for the current device is available from the `inode` structure.

int MAJOR(kdev_t dev);
int MINOR(kdev_t dev);

These macros extract the major and minor from a device item.

int register_chrdev(unsigned int major, const char *name, struct file_operations *fops);

Registers a character device driver. If the major number is not 0, it is used. Otherwise a dynamic number is assigned for this device.

int unregister_chrdev(unsigned int major, const char *name,

Unregisters the driver at unload time. Both the major number and the name string must match what was used to register the device.

```
#include <asm/segment.h>
#include <asm/uaccess.h>
unsigned long __copy_from_user(void *to, const void *from, unsigned long count);
unsigned long __copy_to_user(void *to, const void *from, unsigned long count);
```

Copy data between user space and kernel space. Always use this instead of assigning or memcpy for transfers between user and kernel space.

```
#include <asm/semaphore.h>
void sema_init(struct semaphore *sem, int val);
int down_interruptible(struct semaphore *sem);
int up(struct semaphore *sem);
```

The semaphore data structure for preventing race conditions.