COMP 4735: Operating Systems Concepts

Lesson 14: Input / Output & Devices

Rob Neilson

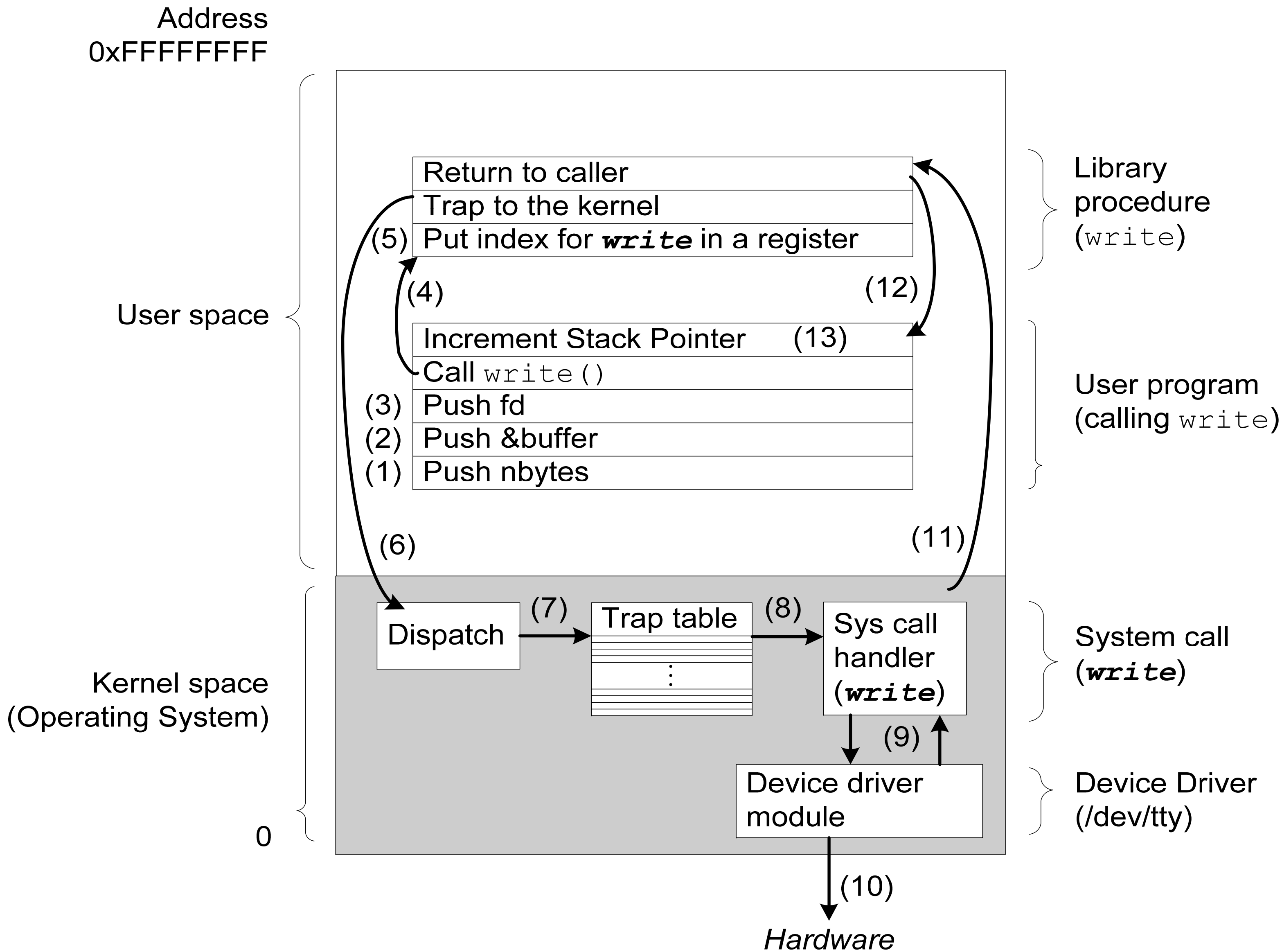
rneilson@bcit.ca

Schedule for remaining classes ...

Week	Date	Lesson and Lab Topics	Textbook
13	Lab 12	File Systems	4.1, 4.2, 4.3
	Apr 6	Quiz 8: Chapters 3.4	3.4
	Apr 8	Input/Output (software principles)	5.1, 5.2
	Apr 9	Input/Output (Interrupts / Drivers)	5.2, 5.3
14	Lab 13	Input / Output (take home lab)	5.1, 5.2, 5.3
	Apr 13	*No Classes - Easter Monday*	
	Apr 15	Quiz 9: Chapters 4.1, 4.2, 4.3	4.1, 4.2, 4.3
	Apr 16	Final Exam Info + Optional Quiz	5.1, 5.2, 5.3
15	***exam week***		

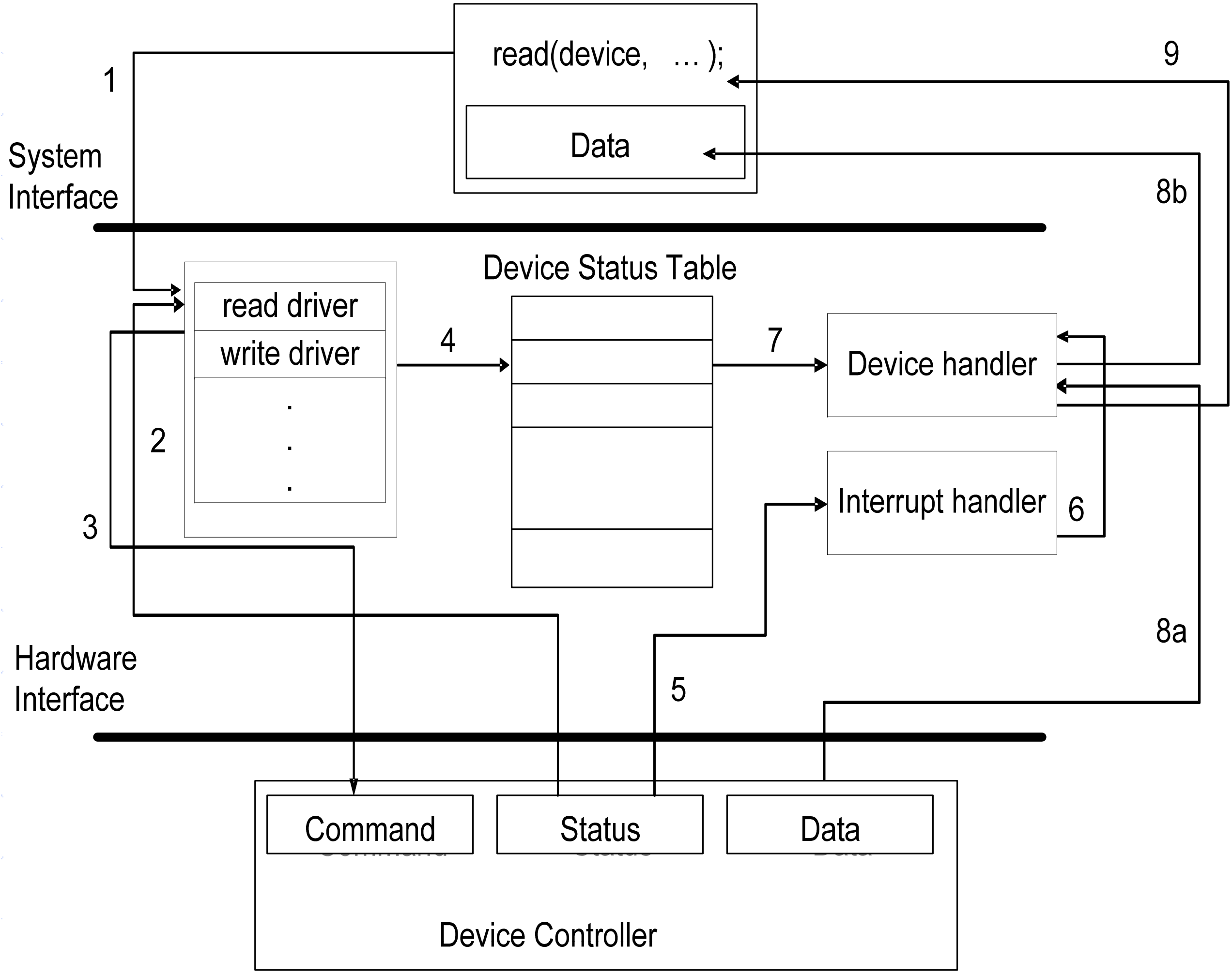
IO & Device Drivers

Recall our System Call Model ...

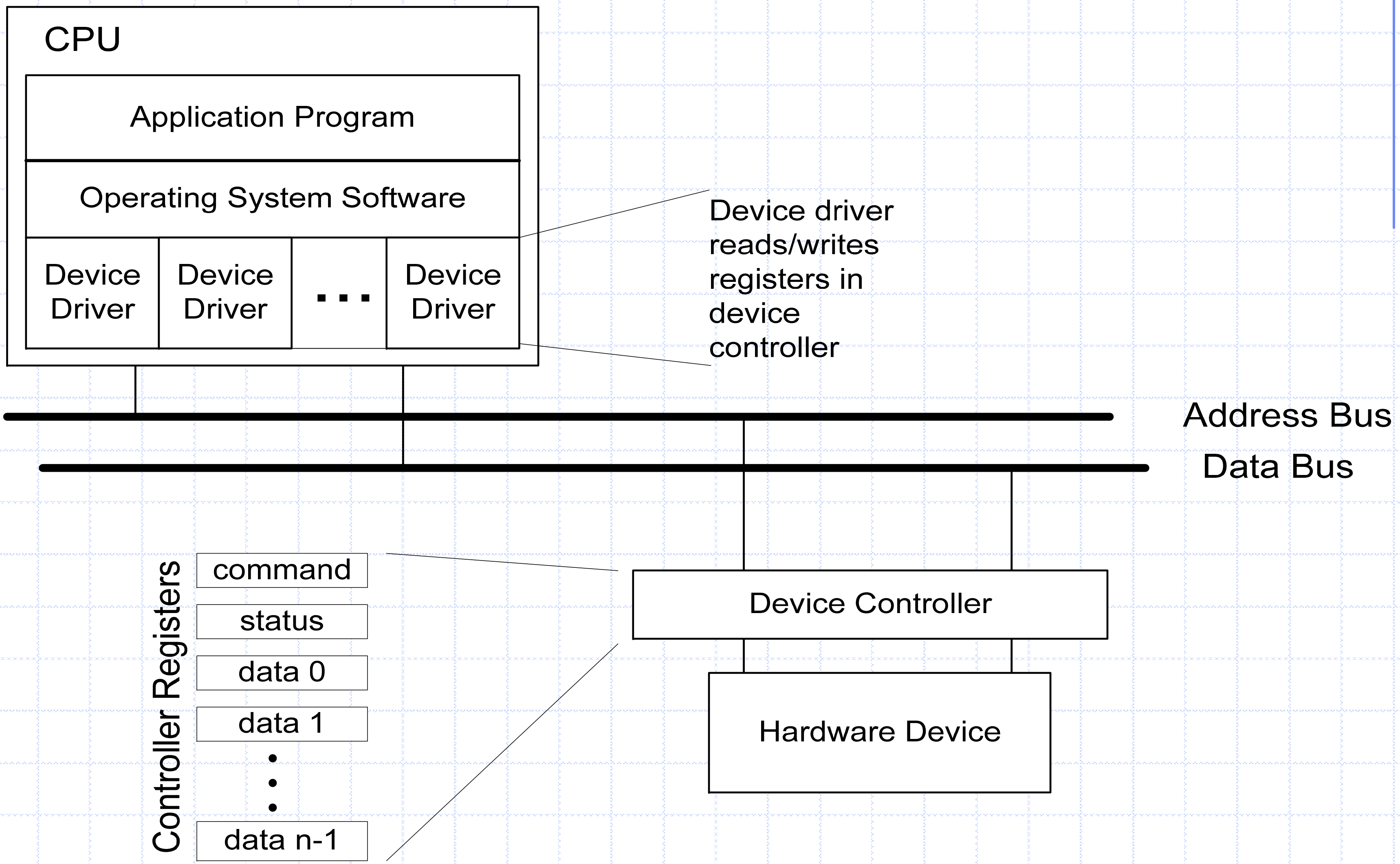


Let's add more detail (for Interrupt driven IO) ...

1. read sys call
2. is dev idle?
3. send cmd to dev
4. save device status and yield CPU
5. dev interrupts CPU when done
6. int hndlr jmps to dev handlr
7. get IO status info
8. copy data from dev to user
9. return control to user app



And Recall How Devices Work ...

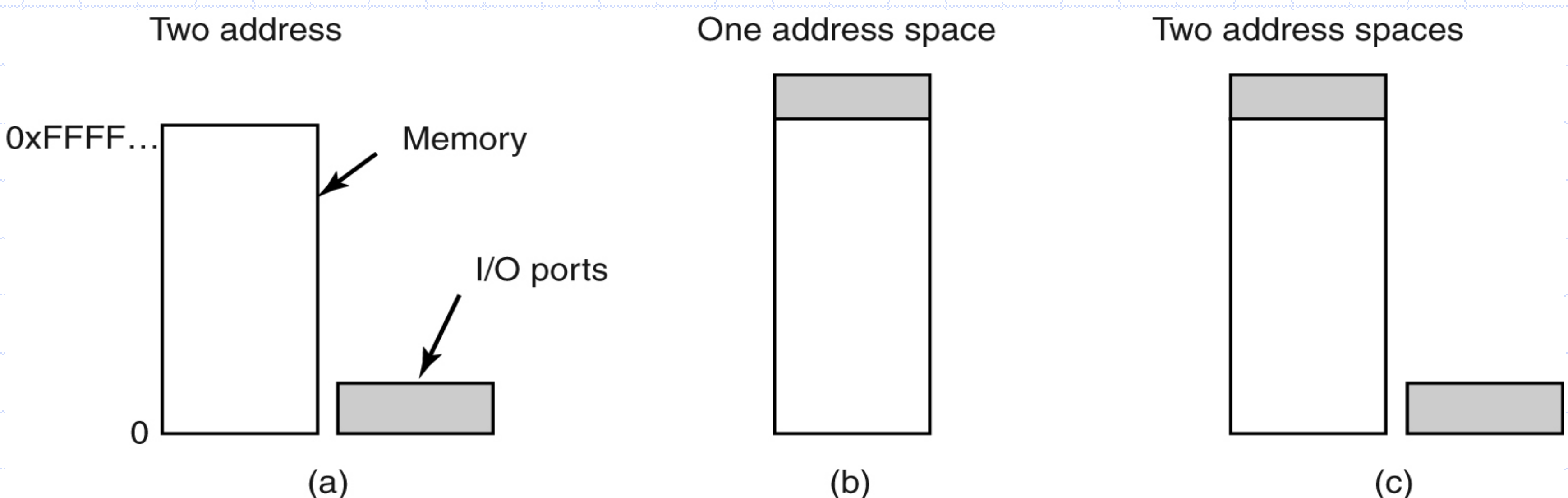


IO Ports and Memory Mapped IO

How does the CPU read and write the device controller registers?

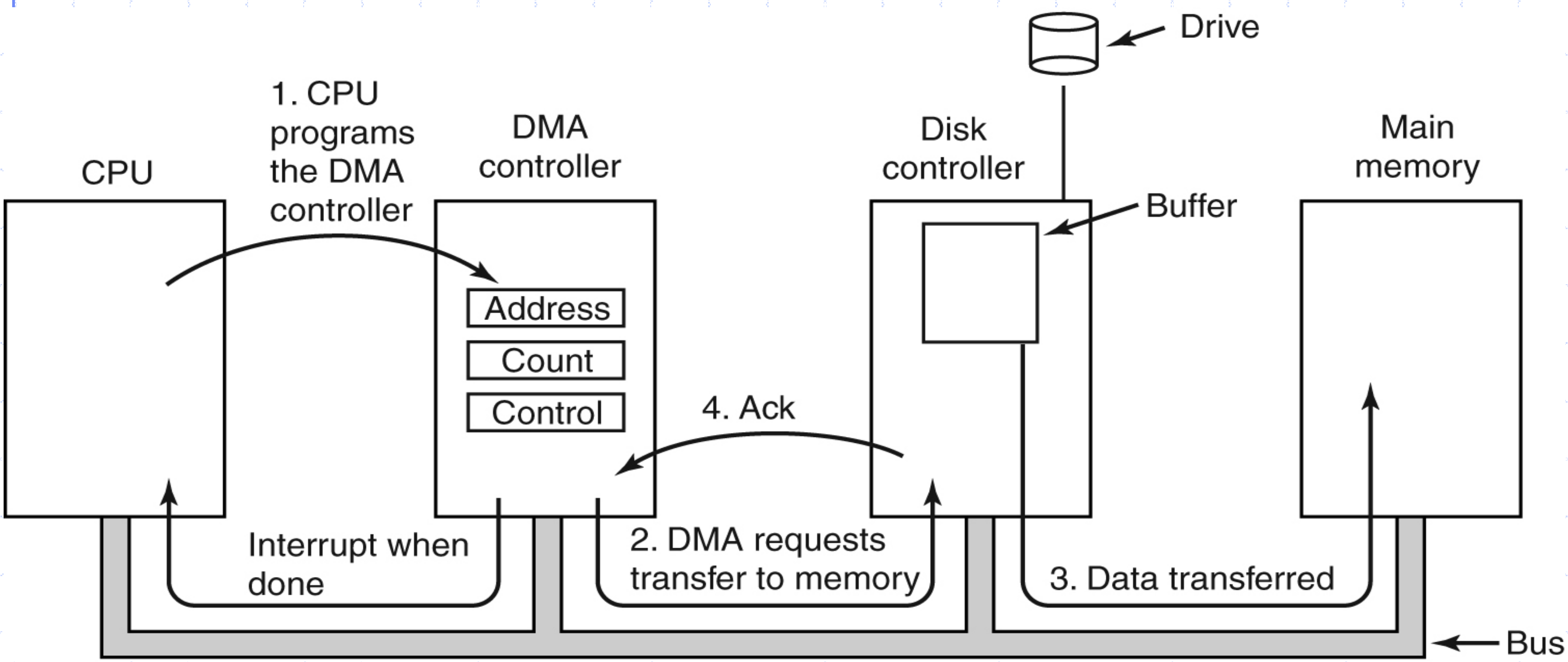
Two approaches:

1. I/O Ports
   - each register is assigned a unique address (port)
   - the ports numbers are known by the OS and used in drivers etc
2. Memory Mapped I/O
   - all control registers are mapped into a portion of the address space
   - device drivers write to the correct address in memory
   - the read/write does not go to RAM, instead it is mapped to the corresponding device control register



Direct Memory Access (DMA)

- DMA uses a separate controller (DMA controller) to manage transfers of data between device control registers and RAM
- This technique offloads IO operations from the CPU, allowing the CPU to do other work



- *Note: without DMA the CPU must do the copying of bytes from the device controller , ie, it needs to process interrupts to move the data registers to RAM*

IO Software

- On the preceding slides we looked at hardware aspects of IO … namely … how do we get bytes from the CPU to a device and vice-versa

  Of course all this has to be driven by software …

- There are three different ways of using software to perform IO, namely:
  1. Programmed I/O
     - CPU uses executes a loop that copies/reads from the IO device, polling to see when IO is complete
  2. Interrupt-Driven IO
     - CPU issues a read/write, and does other stuff until an interrupt signals that the operation is complete, then it does the next read/write, and so on
  3. IO Using DMA
     - CPU sets up the DMA controller to do all the read/write operations, and then does other work until it receives an interrupt saying all IO operations are done

Writing to a device using Programmed IO

```
// this code is executed when system call is made

copy_from_user(buffer, p, count);        // move data to kernel
for (i=0; i<count; i++) {                // let device respond
   while (*device_status_reg != READY); // ... BUSY WAIT ...
   *device_data_register = p[i];        // write next byte
}
return_to_user( );                       // exit system call
```

Writing to a device using Interrupt-Driven IO

```
// this code is executed when system call is made

copy_from_user(buffer, p, count);        // move data to kernel
enable_interrupts( );                    // let device respond
while (*device_status_reg != READY) ;    // is device available?
*device_data_register = p[0];            // write to device
scheduler( );                            // let another proc run
```

```
// this code is the ISR that executes when the device has
// finished the write operation and is ready for more data

if (count == 0) {                        // are we finished ...
   unblock_user( );                      // Yes -> unblock
} else {                                 // No  ...
   *device_data_register = p[i];         // ... write next byte
   count = count - 1;
   i = i + 1;
}
acknowledge_interrupt( );                // Ack this interrupt
return_from_interrupt( );                // let scheduler run
```

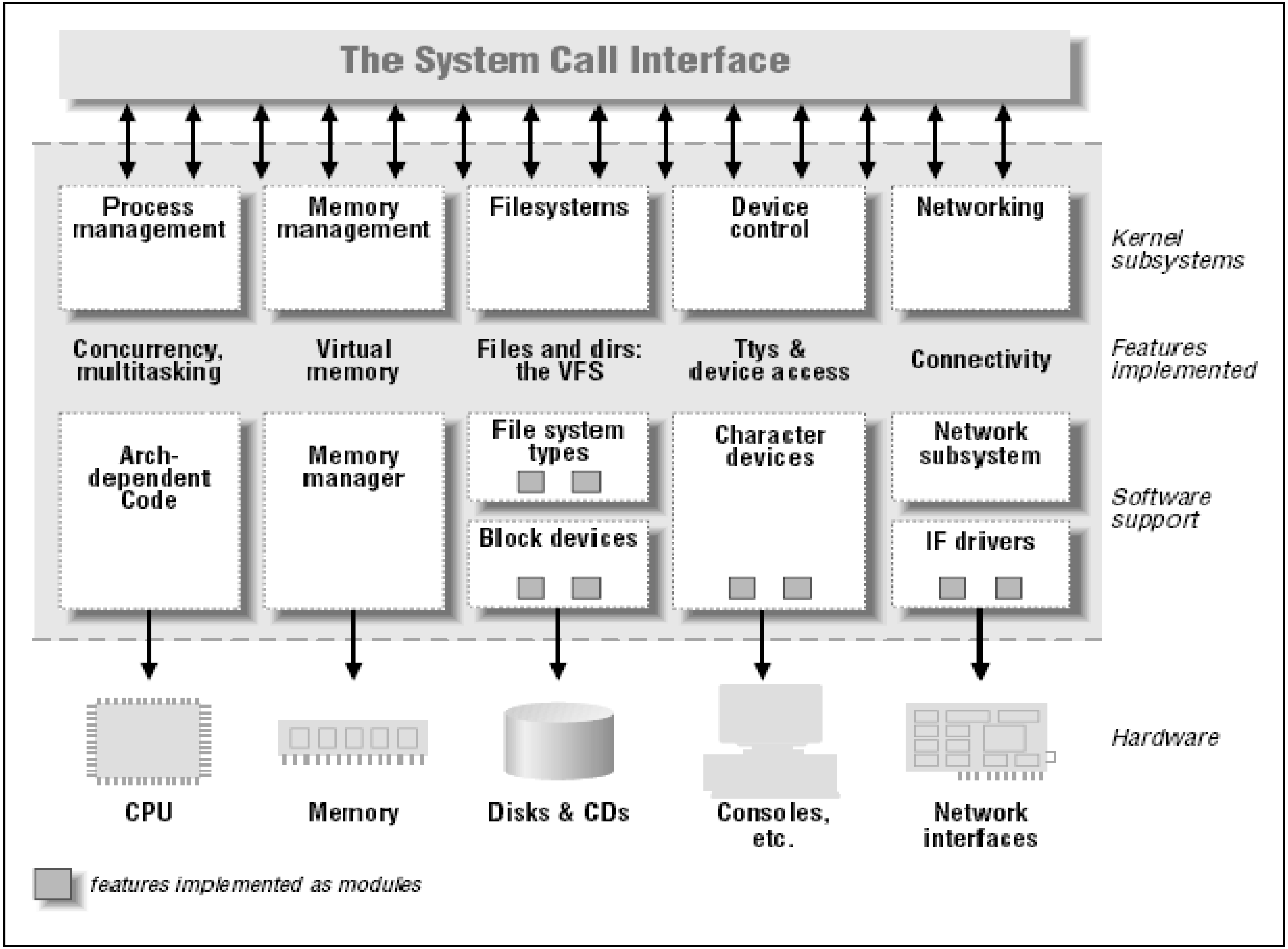
Writing to a device using DMA

```
// this code is executed when system call is made

copy_from_user(buffer, p, count);   // move data to kernel
set_up_DMA_controller();            // where is data? how much?
scheduler( );                       // let another proc run
```

```
// this code is the ISR that executes when the DMA CONTROLLER
// has finished writing all the data
//
// essentially the DMA controller is doing Programmed IO in
// parallel with the CPU, letting the CPU do other work

acknowledge_interrupt( );          // DMA done; all bytes copied
unblock_user;                      // IO operation is done
return_from_interrupt( );          // let scheduler run
```

Review of Devices ...

Device Types ...



Character Devices ...

- *reads or writes one byte per operation*
- typically accessed as using a “stream” abstraction
  - eg: terminal, keyboard, printer
- char drivers typically implement
  - open()
  - close()
  - read()
  - write()
- usually these devices are accessed *sequentially*, ie, one byte after another (no jumping back and forth)
- most older peripherals (eg: joystick) are char devs

Block Oriented Devices

- *read or write a block of data in one operation*
- a block is usually 1 KB (or some value of 2^n)
- block devices typically contain filesystems
  - ie: in linux it can be *mounted*
  - eg: storage devices – disks, cdroms etc
- block drivers tend to be more complicated as operations take longer and device implements more operations
- block interfaces are often standardized (eg: for hard disks, cd-roms etc), so we do not usually have to write these drivers ourselves (the drive manufacturers write them)

Other I/O interfaces ...

- many types of devices are accessed through controllers that have a richer interface, example:
  - network drivers
  - PCI drivers
  - USB drivers
  - SCSI drivers
- these interfaces use the kernel driver framework, but have their own functions/commands/framework extensions that the driver developer needs to learn

Sample Device Interface

- the following examples are is for the interface to a parallel port device on x86
- the interface is as defined in Linux
- remember … what we need is a way to read and write to the device controller registers …
  - ie … data, control, status registers in the parallel port device controller
- Note: this is a **`char`** type device (stream oriented)

Parallel Port - Data Register

- parallel port's base address is:
  - `0x3bc` for /dev/lp0, `0x378` for /dev/lp1, and `0x278` for /dev/lp2
    - (these are port numbers for an x86 PC)

- there are kernel functions to read and write these registers
  - `unsigned inb(unsigned port);`
  - `void outb(unsigned char byte, unsigned port);`

- *data* port (BASE+`0`) is for the actual data signals (wires) on the port
  - lines D0 to D7 for bits 0 to 7, respectively
  - states: 0 = low (0 V), 1 = high (5 V))
  - a write to this register latches the data on the pins
  - a read returns the data latched on the pins
    - (which could be the last thing you wrote)

Parallel Port - Status Register

- *status* port (BASE+1) is read-only, and returns the state of the following input signals
  - bit 0: reserved
  - bit 1: reserved
  - bit 2: IRQ status
  - bit 3: ERROR
  - bit 4: SLCT
  - bit 5: PE
  - bit 6: ACK
  - bit 7: BUSY

Parallel Port - Control Register

- *control* port (BASE+2) is write-only
  - read returns the data last written
  - controls the following status signals
    - bit 0: STROBE
    - bit 1: AUTO_FD_XT
    - bit 2: INIT
    - bit 3: SLCT_IN
    - bit 4: enable IRQ
      - which occurs on the low-to-high transition of ACK when this bit is set to 1
    - bit 5: controls the extended mode direction (0 = write, 1 = read)
      - write-only (a read returns nothing useful for this bit)
    - bit 6: reserved
    - bit 7: reserved

Adding a new device to `/dev/lp0`

- when we build a device, we make sure it conforms to the pinout for the port connector we are using
- the pinout for a 25-pin parallel port is (i=input, o=output):

Pin	Usage	Pin	Usage	Pin	Usage
1io	STROBE	9io	D7	17o	SLCT_IN
2io	D0	10i	ACK	18-25	GROUND
3io	D1	11i	BUSY		
4io	D2	12i	PE		
5io	D3	13i	SLCT		

Using the Device Interface

- the device driver needs to perform the reads and writes to the device
- it uses the interface (for LPT is uses the interface as described on previous slides)

- here is a quick outline of what the driver software does …

Example parallel port driver output operation ...

- When no output activity is going on (and after startup)
  - driver sets the output signal lines as follows:
    - STROBE (pin 1) high, data not valid
    - AUTOFD (pin 14) high, do not feed paper
    - INIT (pin 16) high, do not initialize
    - SELCTIN (pin 17) low, printer selected
- To print a character
  - the driver waits for BUSY to go low
  - then outputs the new data
  - delay at least 0.5 microsecond, then STROBE (pin 1) is pulsed low for at least 0.5 microsecond
  - the driver waits for ACK after printing each character
- To use the interrupt mode
  - a rising edge on ACK will cause an interrupt request if interrupts are enabled

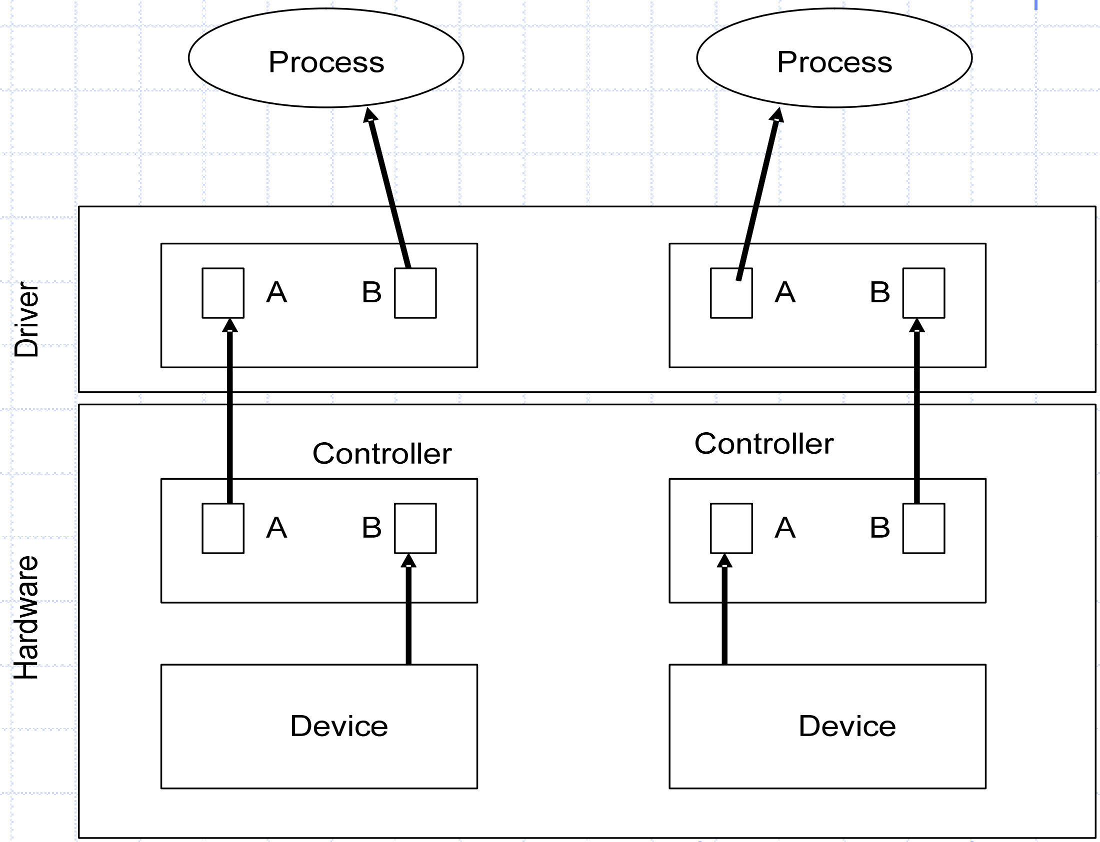
Buffering

Device IO and buffering

- buffering can be on input or output
- allows some delay on behalf of the IO device or the process
  - ie: CPU and/or device doesn't have to always be ready for the IO event
  - this is important, as most IO events are asynchronous (they happen at arbitrary times), and the CPU could be busy doing something else
- hardware buffering
  - the idea is to add a buffer to the device controller
  - the buffer will store some number of bytes so we don't have to read them as soon as they are input

Double Buffering

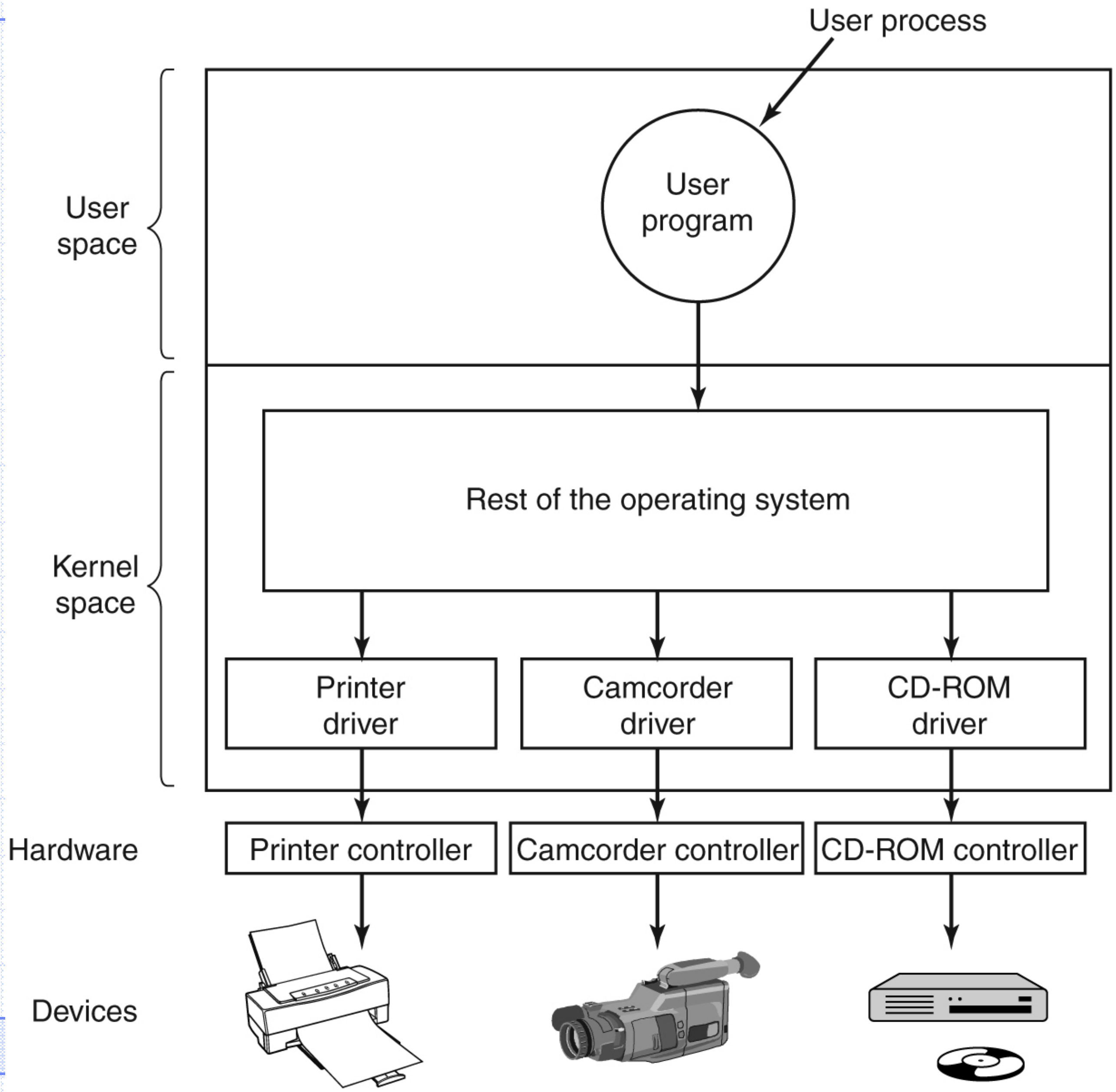
- there is a hardware buffer as well as a second buffer in the driver
- can be used to facilitate pre-processing by the driver before contents are passed to user-mode pgm
  - eg: linux ‘cooked tty’
  - linux tty devices operate in raw mode (each char is sent to application as it arrives) or in cooked mode, where chars are buffered and sent when a line feed occurs)



- *this strategy can be generalized to use n buffers*
  - *called circular buffering*

A General Device Driver Framework

Device Driver Framework



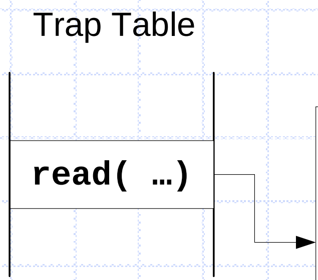
System Call Interface to Device Drivers (1)

- make device functions available to application programs
- abstract all devices to a few interfaces
- make interfaces as similar as possible
- device driver implements functions
  - one entry point per API function

- for example, a driver might to implement **`write(), read(), open(), close()`**
  - drivers only need to implement the functions that apply to the device
  - for instance, a printer would likely not implement **`read()`**

System Call Interface to Device Drivers (2)

- OS exports a common interface to user pgms
  - eg: **`read(dev_id, buffer, size)`**
- kernel maps the common function calls to device-specific implementations of these functions




```
dev_read(devID  , … ) {
  // Processing common to all devices
      …
  switch(devID  ) {
  case dev0:      dev0_read(  … );
                  break;
  case dev1:      dev1_read(  … );
                  break;
      …
  case devM :     devM_read  (… );
                  break;
 };
 // Processing common to all devices
      …
}
```

Kernel Interface to Device Drivers

- drivers are distinct from main part of kernel
  - sometimes linked in when kernel is built
  - sometimes linked dynamically at run-time
- kernel code uses the drivers
  - the kernel makes calls to specific functions, drivers implement these functions
- drivers use common kernel functions for things like
  - device allocation
  - resource (e.g., memory) allocation
  - scheduling
  - etc. (varies from OS to OS)

Linux Driver Example

Linux Device Driver Framework - 1

- drivers are identified by *special* files in the **`/dev`** filesystem
  - all char and block devices have entries in /dev

```
crw------- 1 root tty     4,1  Aug 16 22:22  tty1
crw-rw-rw- 1 root dialout 4,64 Jun 30 11:19  ttyS0
crw-rw-rw- 1 root dialout 4,65 Aug 16 00:00  ttyS1
crw------- 1 root sys     7,1  Feb 23 19:66  vcs1
crw------- 1 root sys     7,12 Feb 23 19:66  vcsa1
```

- each driver has a major number and a minor number
  - major number: identifies the type of driver
    - major numbers are defined in **`include/linux/major.h`**, for example
      - **`#define FLOPPY_MAJOR 2`**
      - **`#define TTY_MAJOR    4`**
      - **`#define LOOP_MAJOR   7`**

Linux Device Driver Framework - 2

- in the `ls` output on the previous page, the major numbers are 4, 7
  - the kernel uses the major number at *open* time to dispatch execution to the appropriate driver
- minor number: identifies the specific driver (of each type)
  - this number is only used by the driver code, so that it can identify which device it is working with
  - the driver code may or may not need to use this number
  - on the previous slide, the minor numbers are 1, 64, 65, 12
- device special files are created (manually) with the linux command

  `mknod /dev/<dev_name> <type> <major#> <minor#>`
  - for example:

```
% mknod /dev/mem c 60 0
```

Linux Device Driver Framework - 3

- in addition to identifying the device (in /dev), we need to load the driver code into the kernel
  - this is done with the linux command **`insmod`** for example

    `% insmod ./mem.o`

  - this command causes the driver code to be dynamically linked to the kernel (via the dispatch table)
  - the file with the driver code is compiled, but not linked (ie: you run `gcc`, but not `ld`)
- there is a corresponding **`rmmod`** command to unlink the module from the kernel

Linux Device Driver Framework - 4

- to bind the module to the kernel, the **`insmod`** command runs a function called **`<driver>_init()`**, which must be coded in the **`driver.c`** file by the driver developer
- your **`driver_init`** function must make a call to **`register_chrdev()`**
  - which is the kernel function to link the driver code to the kernel. The prototype is …

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

Note: the **`rmmod`** command requires that you code a **`<driver>_exit`** function, which will make a call to **`unregister_chrdev()`** to unlink the module from the kernel

Linux Device Driver Framework - 5

File Operations: defines the entry points for the driver

- when `insmod` runs, the entry points for the supported driver operations (eg: `read, write`) need to be linked to the kernel
- this is done by defining a `file_operations` structure in `<driver>_init()`
- a prototype `file_operations` structure is shown on the next slide ...

File Operations Prototype

```
struct file_operations {
  loff_t   (*llseek) (struct file *, loff_t, int);
  ssize_t  (*read)    (struct file *, char *, size_t, loff_t *);
  ssize_t  (*write)   (struct file *, char *, size_t, loff_t *);
  int      (*readdir)(struct file *, void *, filldir_t);
  uint     (*poll)    (struct file *, struct poll_table *);
  int      (*ioctl)   (struct inode *, struct file *, uint,ulong);
  int      (*mmap)    (struct file *, struct vm_area_struct *);
  int      (*open)    (struct inode *, struct file *);
  int      (*flush)   (struct file *);
  int      (*release)(struct inode *, struct file *);
  int      (*fsync)   (struct file *, struct dent *, int dsync);
  int      (*fasync) (int, struct file *, int);
  int      (*lock)    (struct file *, int, struct file_lock *);
                    ... etc ...
};
```

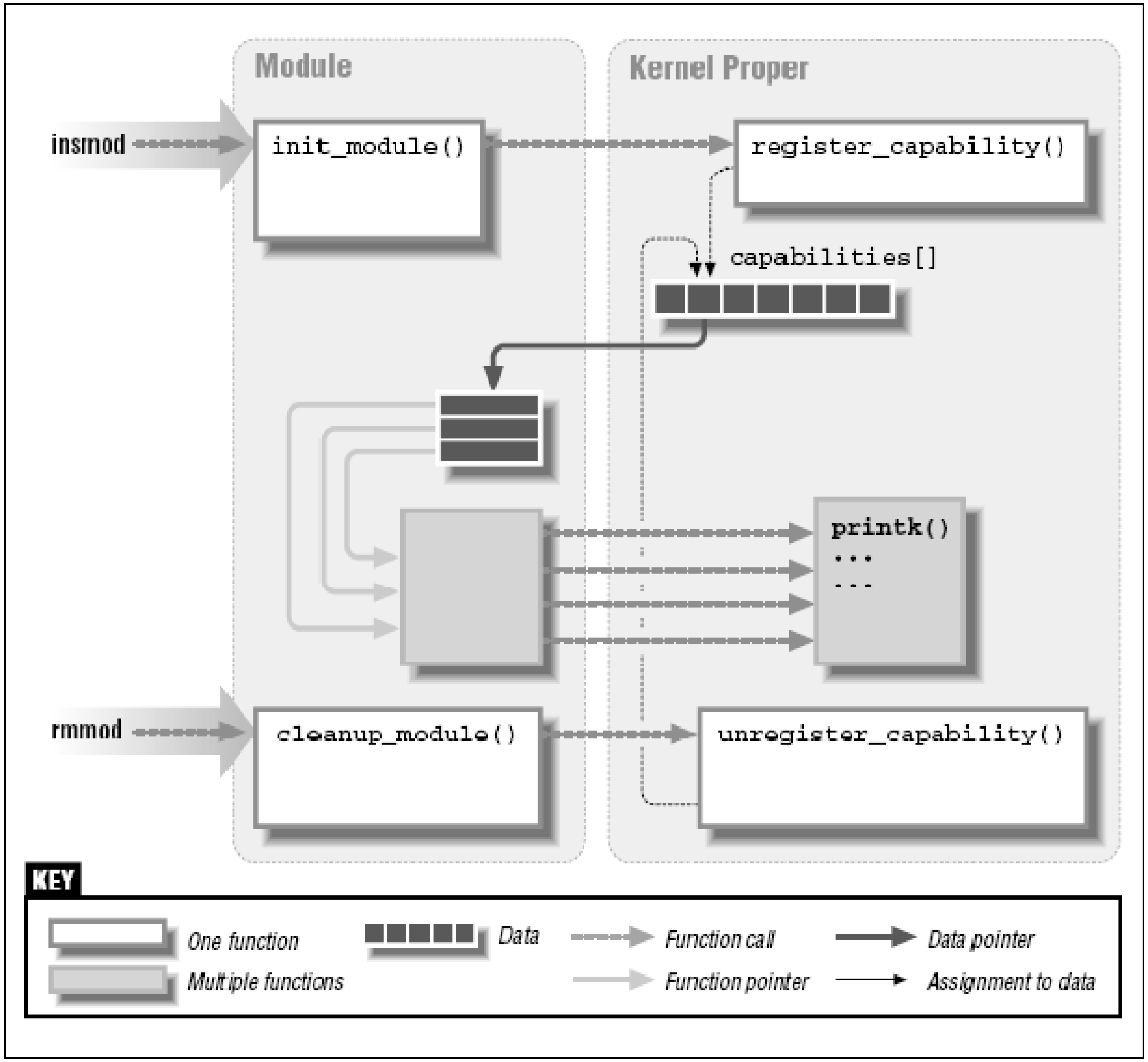
Sample **file_operations** structure

- this declaration of **file_operations** is for a driver named foo, coded in (foo.c); assume the driver supports …
  - **read, write, open, close, ioctl**

```
struct file_operations foo_ops = {
  NULL;
  foo_read;
  foo_write;
  NULL;
  foo_ioctl;
  NULL;
  foo_open;
  NULL;
  foo_close;
  NULL;
  ... etc ...
};
```

> this structure identifies the actual functions that implement the file operations so that **insmod** can link them in the dispatch table

Linking a module to the kernel



A Sample Linux Device Driver

A sample driver:

- now let’s code a driver

Our driver (called mem) will be able to:

- load into the kernel
- be removed from the kernel
- let us write a character to it
- let us read a character from it

Sample user session:

```
% mknod /dev/mem
% chmod 666 /dev/mem
% insmod mem.ko
% echo –h abcdef > /dev/mem
% cat /dev/mem
%
```

```c
/* these are the prototypes for functions we will code */

int mem_open      (struct inode *inode, struct file *filp);
int mem_close     (struct inode *inode, struct file *filp);
ssize_t mem_read  (struct file *filp, char *buf,
                   size_t count, loff_t *f_pos);
ssize_t mem_write (struct file *filp, char *buf,
                   size_t count, loff_t *f_pos);
void mem_exit     (void);
int mem_init      (void);

module_init(mem_init);     /* identify init function */
module_exit(mem_exit);     /* identify exit function */

/* Global variables of the driver */

int mem_major = 60;        /* Major number          */
char *mem_buffer;          /* Buffer for IO data    */
```

```
/* Declaration of supported file operations */

struct file_operations mem_fops = {
   NULL;
   mem_read;
   mem_write;
   NULL;
   mem_ioctl;
   NULL;
   mem_open;
   NULL;
   mem_close;
   NULL;
   NULL;
   NULL;
   NULL;
};
```

```
int mem_init(void) {                        /* run by insmod command */
   int result;

   result = register_chrdev(mem_major, "mem", &mem_fops);
   if (result < 0) {
      printk("<1>mem: cannot get major %d\n", mem_major);
      return result;
   }

   memory_buffer = kmalloc(1, GFP_KERNEL);  /* Alloc 1 byte   */
   if (!memory_buffer) {                    /* of kernel mem  */
      result = -ENOMEM;                     /* to be the read */
      goto fail;                            /* write buffer.  */
   }
   memset(memory_buffer, 0, 1);             /* init buff to 0 */
   return 0;                                /* normal return  */

fail:
   mem_exit();
   return result;
}
```

```
/* this method run by the rmmod system call */

void mem_exit(void) {

   /* Freeing the major number */
   unregister_chrdev(mem_major, "mem");

   /* Freeing buffer memory */
   if (memory_buffer) {
      kfree(memory_buffer);
   }
   printk("<1>Removing module: mem\n");
}
```

```
/* mem_open() - called when user executes fopen() system call */

int mem_open (struct inode *inode, struct file *filp) {
   return 0;
}
```

Some notes:

A) Why doesn't it do anything?

- this is a simple driver; it is an example
- the driver-independent parts of open will create and init the file struct , which is all we need to be able to use this file
- in most drivers, *open* should perform the following tasks:
  - increment the usage count
  - check for device-specific errors (such as device-not-ready or similar hardware problems)
  - initialize the device, if it is being opened for the first time
  - identify the minor number and update the **f_op** pointer, if necessary
  - allocate and fill any data structure to be put in **filp->private_data**

```
/* mem_open() - continued ... */

int mem_open (struct inode *inode, struct file *filp) {
   return 0;
}
```

Notes: (continued)

C) What is struct file used for

- the **`struct file`** is a kernel structure
- it is different that the FILE typedef that is used by user pgms
- there is one **`struct file`** in the kernel for each *open file*
- the **`struct file`** is created by the kernel *open()* function, and released by the kernel *close()* function
- **`struct file`** is passed (by the kernel) to every kernel function that operates on the file
- **`struct file`** contains transient information about the file, such as

`mode_t f_mode;`	- the file mode: read, write, or both
`loff_t f_pos;`	- current offset in the file (for reading or writing)
`uint f_flags;`	- O_RDONLY, O_NONBLOCK etc
`struct file_operations *f_op;`	- a pointer to the f_op structure - kernel uses this for dispatch

```
/* mem_open() - continued ... */

int mem_open (struct inode *inode, struct file *filp) {
   return 0;
}
```

Notes: (continued)

D) What is struct inode used for

- the **`struct inode`** is a structure that identifies a physical file
- it contains fields to specify where the file is on disk, the owner, permissions etc
- the inode also contains a reference to a kernel structure called **`kdev_t`**
  - **`kdev_t`** is where the major and minor device numbers are actually stored

```
int mem_close (struct inode *inode, struct file *filp) {
    return 0;
}
```

Some notes:

A) this is the function that gets run when **`fclose()`** is run on device /dev/mem

B) typically, this function should perform the following

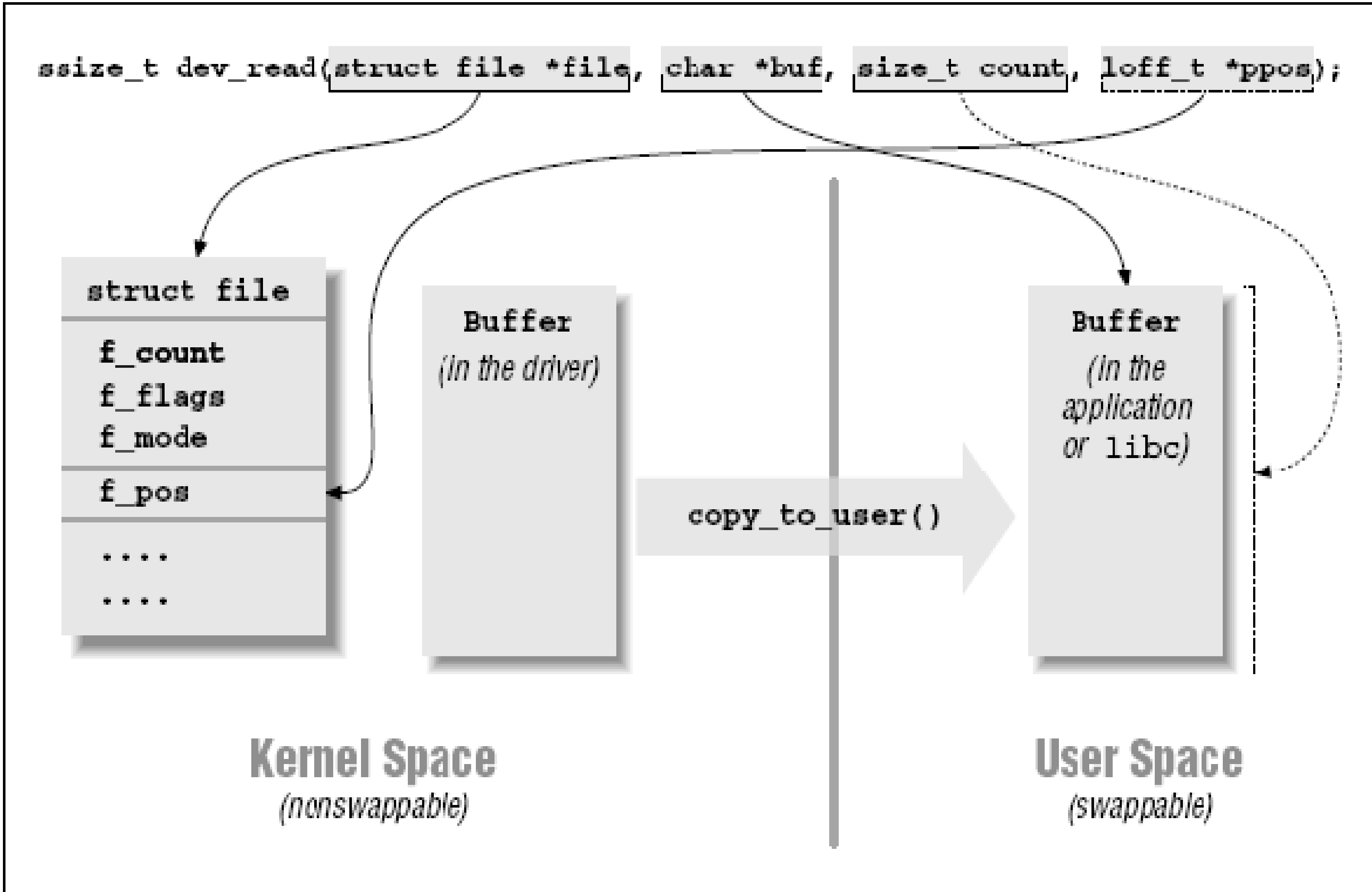
- deallocate anything that *open* allocated in **`filp->private_data`**
- shutdown the device on last close
- decrement the usage count

```
ssize_t mem_read (struct file *filp, char *buf,
                  size_t count, loff_t *f_pos)  {

   copy_to_user(buf,memory_buffer,1); /* copy data to ...    */
                                      /* ... user space      */

   if (*f_pos == 0) {
       *f_pos+=1;                    /* advance reading position */
       return 1;
   } else
       return 0;
}
```

Some notes:

- this is the function that gets run when fread()is run on device /dev/mem
- the function only every transfers 1 byte to the user's buffer
- the file position is incremented (to end of file)
- if the function is called when the current buffer contents have already been read, end of file (0) is returned
- the function copy_to_user() is a kernel routine that copies data from kernel space to user space
  - it ensures that the memory addresses are in the correct areas etc

How the argumeants to **mem_read()** are used ...



```
ssize_t mem_write (struct file *filp, char *buf,
                   size_t count, loff_t *f_pos) {
   char *tmp;
   tmp=buf+count-1;
   copy_from_user(memory_buffer,tmp,1);
   return 1;
}
```

Some notes:

- this is the function that gets run when **`fwrite()`** is run on device /dev/mem
- the function only ever writes 1 byte to our memory buffer
- it writes the last byte from the user supplied buffer
- depending in the semantics that we want for this device, we might want to reset the *f_pos=0, so that we can read the new byte with a subsequent call to mem_read

Driver event to function mapping

Events	*User functions*	*Kernel functions*	*mem.c functions*
Load module	`insmod`	`module_init()`	`mem_init()`
Open device	`fopen()`	`file_operations: open`	`mem_open()`
Close device	`fread()`	`file_operations: read`	`mem_read()`
Write device	`fwrite()`	`file_operations: write`	`mem_write()`
Close device	`fclose()`	`file_operations: release`	`mem_close()`
Remove module	`rmmod`	`module_exit()`	`mem_exit()`

The End