Sistemas Operativos: Input/Output
I/O SW Layers

Pedro F. Souto (`pfs@fe.up.pt`)

April 20, 2012

Topics

Topics

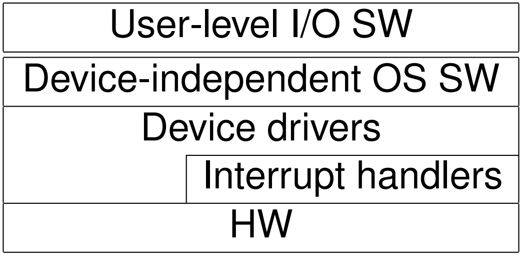
Device Drivers

Device Independent Layer

User-Level Layer

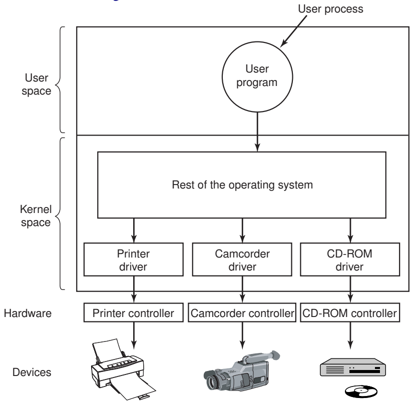
Additional Reading

I/O SW Layers



- Each layer has a well defined interface to the adjacent layers
  - And provides a well defined service to the layer above it

Device Dependent Layers



Device Drivers

- Comprises all the code that is device specific
  - Code that interfaces with the device controller

Interrupt Handlers Code that is executed upon interrupt of a device

- Usually, an interrupt handler is part of the driver of the corresponding device

Interrupt Handlers (1/3)

- Interrupts may occur as a result of:
  1. Either an I/O operation iniciated by some process, e.g. a read or a write from a disk;
  2. Or as a result of an asynchronous event, e.g. the reception of a key's scancode from the keyboard
- In case 1) there is usually a process waiting for completion of the I/O operation, e.g. by doing a down on a semaphore, or a receive on a message
  - Upon an interrupt, the handler does what it has to and then wakes up the process
- In case 2) the kernel has usually to buffer the input data until some process reclaims it
  - It may well be that there is a process blocked waiting for data

Interrupt Handlers (2/3)

1. Save registers that are not automatically saved by the HW upon an interrupt
2. Setup a context for the interrupt handler (related to memory management)
3. Setup a stack for the interrupt handler
4. Acknowledge the interrupt controller
5. Copy the registers from where they were saved to the process table
   - Basically, this is the initial part of a context switch
6. Run the interrupt service procedure
   - Accesses the I/O controller registers
   - Reenables the interrupts if they were not reenabled before
7. Choose the next process to run, if the interrupt has awaken a higher priority process
8. Set up the context for the new process to run, and run ti
   - Thus concluding a context switch

Interrupt Handlers (3/3)

- Sometimes servicing of an interrupt is split in two parts

  Top half The first one, is executed by the interrupt handler, possibly with the interrupts disabled. Typically:
    - Saves data to a device specific buffer
    - Schedules the corresponding bottom half

  Bottom half The second one, is executed out of the interrupt handler, with the interrupts enabled
    - Initiates another I/O operation
    - Does the required processing
    - Unblocks processes

- Linux provides two mechanisms for implementing the bottom half:

  Tasklets The kernel guarantees that they are handled always before the following clock tick
    - But they cannot block

  Workqueues The code can block, but the kernel may take longer to schedule it

Device Driver

- Code that interfaces with I/O controllers
- Usually, each device driver handles:
  - One device type, e.g. PS/2 mouse, AT Keyboard
  - At most one class of devices, e.g. SATA disk driver
- Most DD are part of the kernel
  - I/O access must be done by trusted code, which in most OSs must be part of the kernel
  - There are some OSs that allow DD to be implemented as special privileged user processes
    - This usually requires the kernel to export I/O related operations, such as input/output operations and enabling/disabling instructions
- Usually, device drivers are provided by the device's manufacturer
  - device drivers are dynamically linked to the kernel
  - the interface between a device driver and the kernel must be well-defined

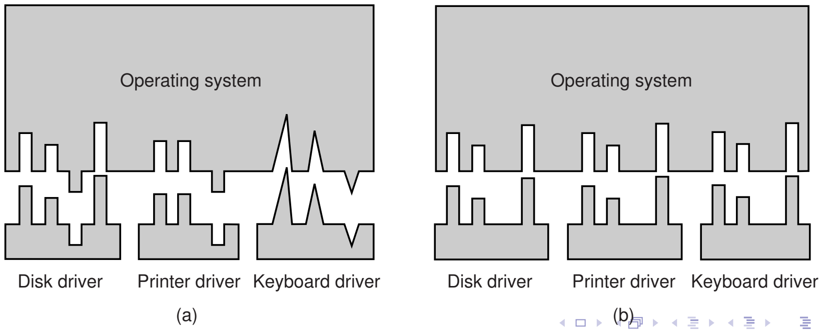
DD Interface

Problem How does a DD interfaces with the rest of the OS?

- There are so many different I/O devices that this may be a real problem.

Solution Each OS defines a uniform programming interface for all DDs. I.e.:

- A set of functions that a DD has to provide for I/O
- A set of functions provided by the kernel that can be used by the DD



Classes of I/O Devices (in Unix/Linux)

Block Devices such as disks, which support the I/O of data blocks

- Whose size can be "large", e.g. 512 bytes or even 4096 bytes
- Which can be accessed independently, i.e. randomly rather than sequentially

Character Devices such as keyboards and audio cards, which generate or accept a stream of bytes/characters

- Usually, support only sequential access

Network Interfaces such as Ethernet or Wifi, whose characteristics are sufficiently different from characer and from block devices

Video Cards which provide essentially a memory-mapped interface for I/O

- Actually, modern video cards include a Graphic Processing Unit with 100's cores which can be programmed not only for graphics processing but also general processing

Character Device Programming Interface in Linux

- To simplify adding new devices, most devices are implemented as kernel modules
  - init function called by the kernel when the module is loaded into memory
  - cleanup/exit function called by the kernel when the module is unloaded
- Character device specific functions include (they are members of the `file_operations` structure)
  - `open` which is called everytime an application opens the special/device file associated to the device
  - `flush` which is called everytime an application closes the special/device file
  - `release` which is called on the *last* close of a file
  - `read/write` which are invoked for reading from/writing to the device
  - `ioctl` which is used for “controlling” the device
    - Most of these functions are invoked upon the invocation of a corresponding system call on the device

Kernel Programming Interface

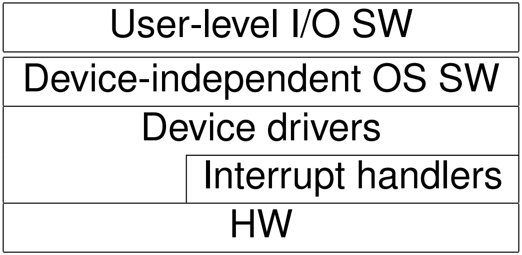
- To help the development of deviced drivers, the kernel provides an extensive set of functions, including
  - `kprintf, kmalloc, kfree` which are similar to their user-level counterparts
  - `container_of` a macro that returns the address of a structure that contains a given address of one of its fields
  - `module_init`, `module_exit` macros for declaring the module initialization and cleanup functions
  - `copy_to_user`, `copy_from_user` to copy data from kernel space to user space and vice-versa
  - `request_region/release_region` for reserving and releasing a region of the I/O address space
  - `inb`, `outb` for reading from/writing to the I/O address space (there are also functions for larger operands)
- Functions specific to character devices include:
  - `alloc_chrdev_region` for dynamically allocating a device major number, and its inverse `unregister_chrdev_region`

Device Driver (DD): Error Handling and Logging

- Upon completion of an I/O operation (whether explicitly requested or asynchronous) a DD has to check for errors
- Devices interface with the real world and errors are quite frequent
    - Damaged disk sectors
    - Communication errors resulting from EMI
        - Serial communication
        - (Wireless)LAN
        - Keyboard
        - Mouse
    - Removal of an hot pluggable device while an I/O request is being processed
- Often these abnormal events have also to be logged
    - Can be done with the help of a general OS logging service

Topics

Device Independent Layer



- Its functionality includes:
  - Uniform interfacing for I/O
  - Buffering
  - Error reporting
  - Allocating and releasing devices
  - Providing a device-independent block size
- Sometimes, for reasons of efficiency, the device dependent layer provides some of this functionality

Uniform Interfacing for I/O: Naming

- E.g. in Unix/Linux block and char devices are named using file pathnames corresponding to entries of the `/dev` directory, or its subdirectories
  - Each entry contains the corresponding:

    major device number which is used to locate the driver

    minor device number which is driver specific, but usually identifies the particular device
  - Again, network devices and graphics cards use a different naming scheme

Uniform Interfacing for I/O: API

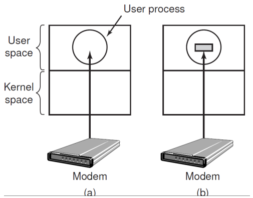
- The integration of device names in the filesystem name space makes it possible to use the filesystem interface
- An application can use the usual system calls to `open`, `read`, `write` and `close` a device
  - The computational model of block and char devices is very close to that of a file, in particular before accessing a device an application must open it
  - In order to test your device driver you'll have to develop short test programs using the file system system calls
- Furthermore, Unix/Linux use the file system access control mechanisms, and rules, also to protect the access to I/O devices
  - The system administrator can set the proper permissions for each device according to a security policy
- The API for networking devices is different
  - Usually, they are accessed via communication protocols that are implemented at the kernel level
  - Nevertheless, the BSD sockets API can also be used to access at least partially the network interface card

Uniform Interfacing for I/O: Block Size

- Different block devices, i.e. disks, may have different sector sizes
- The device-independent SW layer hides this fact by providing a uniform block size to higher layers
  - For example, by treating consecutive sectors as a logical block
- Similar issues arise:
  - In the case of networking devices
    - The size of a network packet may vary
  - And also for graphical cards
    - Video cards may support different resolutions and number of colors

Buffering

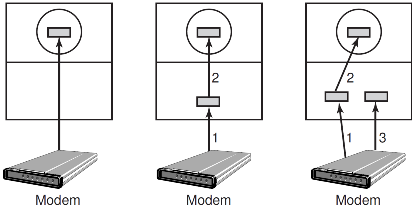
- Refers to the use of a buffer to temporarily hold the data transferred from an I/O device
- Decouples the program that requests the I/O operation from its execution
  - Speed mismatch between processor and I/O devices
  - Many I/O operations are asynchronous to user-process execution



No buffering The data has to be transferred directly to the user process

- Very inefficient, because the user process must read one character/byte at a time leading to many context switches

Buffering



In user space The user process allocates a buffer used to transfer data directly to/from the I/O device

- Need to pin-down the page to prevent it from being paged out. If too many pages are pinned down ...

In kernel space Data are transferred between the I/O device and a buffer in kernel space

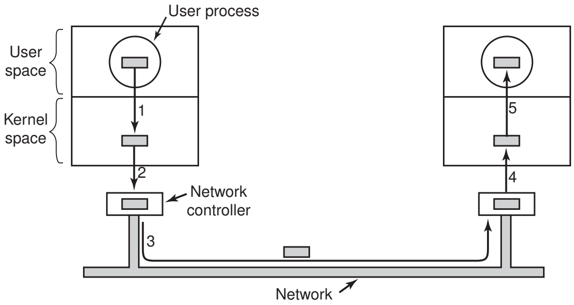
- And from that buffer to user space, sequentially

Double buffering Allows data to be transferred between the I/O device and the kernel, and between the user and the kernel simultaneously

- May use a buffer pool rather than just two buffers

Buffering and Networking

- For transmission, buffering allows the `send()` system call to return immediately after copying the data to the kernel
- For reception, buffering allows the user processes to operate asynchronously wrt communication



- Buffering inside the network controller is essential because the memory bus may be too busy, and DMA may not be responsive enough to prevent the loss of data
- But each buffer in the chain adds some overhead
  - Many tricks have been proposed to reduce this overhead

Buffering and Computer Graphics

- Use of a single buffer in computer graphics may lead to a poor image with, among other artifacts:
  - Flicker
  - Tearing

  This is because the application modifies the buffer with the screen image (frame) while it is displayed by the video card
- Double buffering allows an application to modify the image on one buffer while the video card displays the content on another buffer

  Problem The application may have to wait for the video card before it starts a new frame

  Solution Use triple buffering
  - One buffer has the frame that is being displayed by the video card
  - A second frame has the frame that is being created by the application
  - The third frame has the most recent completed frame not yet passed to the video card, if any

Error Handling and Reporting

I/O Errors These are errors that occur at the device level, e.g. trying to read a damaged sector

- Usually, the device driver (DD) tries to handle these
- If the error persists, the DD has to report it to the device independent layer
- Some actions that may be taken by the device independent SW layer are:
  - Retry the operation up to a maximum number of times
  - Ignore the error or report it to the user level (via the return value of the system call)
  - Terminate the calling process
  - Logging the error and shutdown, in extreme cases that affect the entire system

Programming errors Such as:

- Invalid device, i.e. a device that does not exist
- Invalid operation, e.g. read from an output device
- Invalid operands, e.g. buffer address

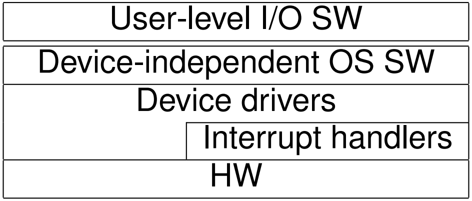
In this case, the kernel just reports back an error code

Allocating and Releasing Dedicate Devices

- Some devices, e.g. CD-ROM recorders, can be used only by a single process at any given moment
- It is up to the OS to ensure that. Some alternatives are:
  - Require the process to call `open` on the device's special file that will return an error if the device is already being used by another process
    - A `close` will release the device
  - Use some other mechanism to request/release such devices
    - And put a process in a waiting queue, if the request cannot be satisfied immediately

Topics

User-level Layer



Library functions

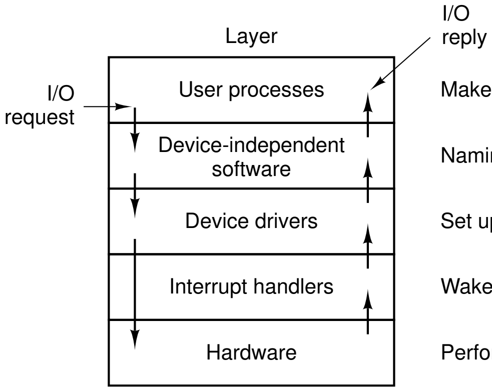
System call stub functions e.g. `read()`, `write()`

Formatted output/input e.g. `printf()` and `scanf()`

Spoolers I.e. user processes that run in the background (**daemons**) and manage devices whose access must be done in mutual exclusion, such as a printer

- A process wishing to print a document creates a file with the data to print in the printer's spooling directory
- The print spooler reads these files and writes their content to the printer
  - It is the only process that can access the printer

Summary of the Functions of the I/O System Layers



Layer	I/O functions
User processes	Make I/O call; format I/O; spooling
Device-independent software	Naming, protection, blocking, buffering, allocation
Device drivers	Set up device registers; check status
Interrupt handlers	Wake up driver when I/O completed
Hardware	Perform I/O operation

Synchronous vs. Asynchronous I/O

Synchronous I/O Can have two modes

Blocking The user process blocks until the operation is completed.

- For some writing operations, the system call may return immediately after copying the data to kernel space and enqueueing the output request

Non blocking The user process does not block

- Not even in input operations: the call returns immediately with whatever data is available at the kernel

Asynchronous The system call just enqueues the I/O request and returns immediately

- The user process may execute while the requested I/O operation is being executed
- The user process learns about the termination of the I/O operation
  - either by polling
  - or via event notification (signals in Unix/Linux)

POSIX Asynchronous I/O

- POSIX.1b specifies several functions for asynchronous I/O

```
int aio_read(struct aiocb *racbp);
int aio_write(struct aiocb *racbp);
int aio_cancel(int fd, struct aiocb *acbp);
ssize_t aio_return(const struct aiocb *acbp);
int aio_error(const struct aiocb *acbp);
```

- The asynchronous I/O operations are controlled by an AIO control block stucture (`struct aiocb`)

```
struct aiocb {
   int           aio_fildes;
   off_t         aio_offset; /* no file position */
   volatile void *aio_buf;
   size_t        aio_nbytes;
   struct sigevent aio_sigevent; /* on completion */
   int           aio_lio_opcode; /* for list op. */
   int           aio_reqprio; /* AIO priority */
}
```

Asynchronous I/O: Operation Termination

Problem How does the user process learn that the operation has terminated?

Solution There are two alternatives, which are specified in the `sigev_notify` member of the `struct sigevent`:

- **Polling (`SIGEV_NONE`)** The process can invoke `aio_error()`
  - It returns `EINPROGRESS` while it has not completed
- **Notification** Here there are also some alternatives
  - **Signal (`SIGEV_SIGNAL`)** the signal is specified in field of the `struct sigevent` of of the `struct aiocb` argument
    - Process must register the corresponding handler via the `sigaction()` system call
  - **Function (`SIGEV_THREAD`)** to be executed by a thread created for that purpose

Asynchronous I/O: `struct sigevent`

```
union sigval {          /* Data passed with notification */
    int     sival_int;         /* Integer value */
    void   *sival_ptr;         /* Pointer value */
};
struct sigevent {
    int             sigev_notify; /* Notification method */
    int             sigev_signo;  /* Notification signal */
    union sigval sigev_value;  /* Data passed with
                                  notification */
    void         (*sigev_notify_function) (union sigval);
                     /* Function used for thread
                        notification (SIGEV_THREAD) */
    void          *sigev_notify_attributes;
                     /* Attributes for notification thread
                      (SIGEV_THREAD) */
    pid_t          sigev_notify_thread_id;
                     /* ID of thread to signal (SIGEV_THREAD_I
};
```

Topics

Additional Reading

Sistemas Operativos

- Section 11.2: *Arquitectura das E/S*
- Section 11.3: *Modelo Computacional das E/S*
- Section 11.4: *Modelo de Acesso à Interface ...*
- Section 11.5: *Gestores de Periféricos*
- Section 11.6: *E/S no Unix*

Modern Operating Systems, 2nd. Ed.

- Section 5.3: *I/O Software Layers*

Operating Systems Concepts, 7th. Ed.

- Section 13.3: *Application I/O Interface*
- Section 13.4: *Kernel I/O Subsystem*
- Section 13.5: *Transforming I/O Requests to HW ...*