Input and Output

Chapter 16

A Programmer’s View Of Devices, I/O, and Buffering.

Topics

- Introduction
- Definition of a device driver
- Device independence, encapsulation, and hiding
- Conceptual parts of a device driver
- Summary

Introduction

- This chapter covers
  - I/O from programmers perspective
  - software needed to control device
  - application software
  - device driver
  - how driver implements read and write

Device Driver

- Software that provides an interface between an application program and an external hardware device.
  - device driver understands details of hardware, hence called low level code
  - all applications accessing a given device us the same driver

Device Driver Properties

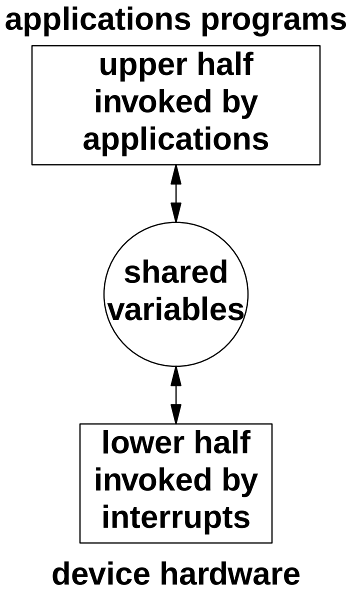
- Device independence
  - device driver removes all hardware details from application programs and relegates them to the driver
- Encapsulation
  - driver hides (encapsulates) device details

    A device driver consists of software that understands and handles all the low-level details of communication with a particular device. Because the driver provides a high-level interface to applications, an application program does not need to change if a device changes.

Conceptual Parts of a Device Driver

- Device driver contains multiple functions that work together
  - code to communicate over a bus
  - code to handle device details
  - code to interact with an application
- Organization
  - Lower half: handler that is invoked when an interrupt occurs
  - Upper half: functions that are invoked by applications to request I/O operations
  - Shared variables: to hold state information needed to coordinate the two halves

Conceptual Parts of a Device Driver



applications programs

upper half invoked by applications

shared variables

lower half invoked by interrupts

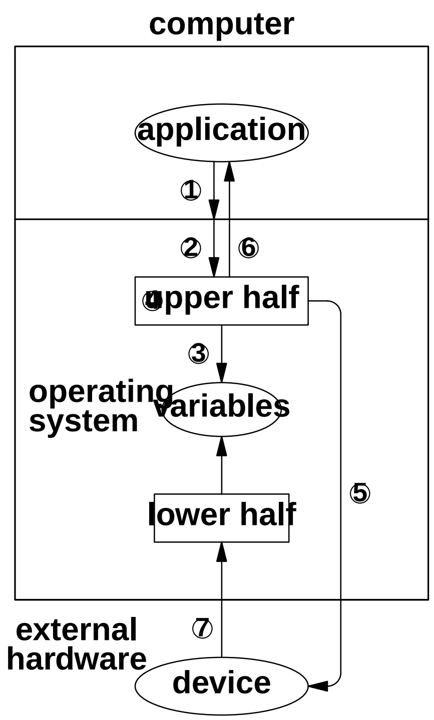
device hardware

The conceptual organization of device driver software into three parts. A driver provides the interface between applications that operate at a high level and the underlying device hardware.

Types of Devices

- Based on the interface type, devices can be divided into two categories
  - Character based devices: transfers single byte of data at a time
  - Block-oriented devices: transfers a block of data at a time

Example Flow Through a Device Driver



Steps Taken

1. The application writes data
2. The OS passes control to the driver
3. The driver records information
4. The driver waits for the device
5. The driver starts the transfer
6. The driver returns to the application
7. The device interrupts

Simplified example of the steps that occur when an application requests an output operation. A device driver located in the operating system handles all communication with the device.

Queued Output Operations

- Driver can waste time with polling, instead a queue is used

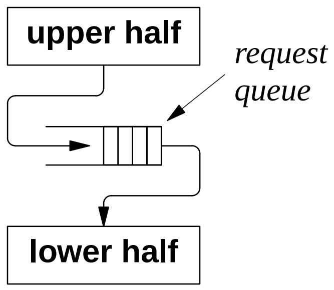


Illustration of a device driver that uses a request queue. On output, the upper half deposits items in a queue without waiting for the device, and the lower half controls the device

Queued Output Operations

scale= 5/7.25

Initialization (computer system starts)

1. Initialize input queue to empty

Upper half (application performs write)

1. Deposit data item in queue
2. Use the CSR to request an interrupt
3. Return to application

Lower half (interrupt occurs)

1. If the queue is empty, stop the device from interrupti
2. If the queue is nonempty, extract an item and start output

3. Return from interrupt

The steps that the upper and lower half of a driver take for an output operation when queuing is used. The upper half forces an interrupt, but does not start output on the device.

Forcing an Interrupt

- Device includes a CSR bit that a processor can set to force the device to interrupt
- If interrupt destined to occur, device finishes current operation and then generates and interrupt, if device is idle, setting the bit causes device to generate an interrupt immediately

Queued Input Operations

- Device driver can use queuing for input

  A production device driver uses an input or output queue to store items. The upper half places a request in the queue, and the lower half handles the details of communication with the device

Queued Input Operations

Initialization (computer system starts)

1. Initialize input queue to empty
2. Force the device to interrupt

Upper half (application performs read)

1. If input queue is empty, temporarily stop the application
2. Extract the next item from the input queue
3. Return the item to the application

Lower half (interrupt occurs)

1. If the queue is not full, start another input operation
2. If an application is stopped, allow the application to run
3. Return from interrupt

The steps that the upper and lower half of a driver take for an input operation when queuing is used. The upper half temporarily stops an application until data becomes available.

Bi-directional Data Transfer

- Bidirectional device
  - supports data transfer in two directions, processor to device and vice versa
- Unidirectional device
  - supports data transfer in one direction
  - may still provide feedback in the other direction
- Approaches to bi-directional transfer
  - Treat the device in two separate devices, one used for input and one used for output
  - Treat the device as a single device that handles two types of commands, one for input and one for output

Asynchronous vs Synchronous programming paradigm

- Synchronous
  - polling is a a synchronous activity, control passes through the code from beginning to end
- Asynchronous
  - interrupt is an asynchronous activity, the user writes separate pieces of code that responds to events
  - more challenging, events occur in any order, even simultaneously
  - programmer uses shared variables to encode current state of computation

Problems with Asynchronous Programming

- Processor creates a linked list of operations in memory, smart device follows the list and performs operations automatically (command chaining)
- What happens if device tries to read list just before processor adds
- What happens if both try to manipulate pointers simultaneously
- Solution: mutual exclusion
  - use special CSR values that processor can temporarily set to prevent device from accessing command list
  - processor temporarily restricts use of bus
  - test-and-set instructions

I/O as Viewed by an Application

In many programming systems, I/O is hidden from the programmer - instead of manipulating hardware devices such as disks and display screens, a programmer only uses abstractions such as files or windows.

Run time I/O libraries

- Where application programmers aren’t allowed to directly control I/O device, the compiler maps each high level I/O operation to sequence of low level steps.
- Compiler translation done indirectly, compiler generates code that invokes library functions to perform I/O operations
- Library functions are called run-time library

Run time I/O libraries

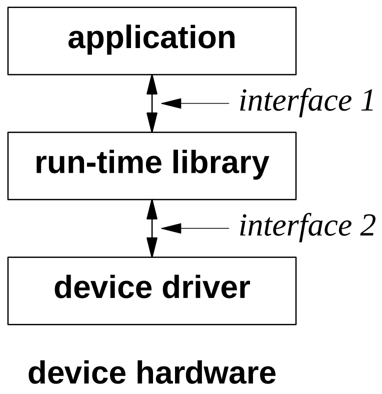
Instead of encoding I/O details into a program, a compiler relies on a run-time library to act as an intermediary. When the application performs an I/O operation, the generated code invokes a library function, which then performs the actual I/O operations

- Advantage of using run-time library
  - flexibility and ease of change
  - in case of changes, only run rime library needs to be updated, compiler remains unchanged
  - only library function understands how to use the underlying I/O mechanisms

Library/OS Dichotomy

- Device driver resides in the OS, and run-time library functions reside outside the OS
- What software does each layer provide?
- What is the interface between an application and the run-time library or the interface between the run-time library and the operating system.
- What are the relative costs of using the two interfaces?

Library/OS Dichotomy



Conceptual arrangement of application code, run-time library code, and a device driver. The run-time library acts as an intermediary.

I/O Operations and OS Supports

- Interface between run-time library and the OS
  - in C, OS interface is directly available to applications
  - programmer can choose an I/O library or make OS calls directly
- Example to read/write a CD
  - open to start the drive motor and ensure disk has been inserted
  - read to read data from disk; write to write data to disk
  - close to power down the disk

I/O Operations and OS Supports

- Example to read/write a CD (contd ..)
  - seek to move to new position
  - ioctl for all other functions (e.g. eject)
- Note each operation takes arguments that specify details
  - example write arguments could specify device to use, location of data, and amount of data

I/O Operations and OS Supports

Operation	Meaning
open	**Prepare a device for use (e.g., power up)**
read	**Transfer data from the device to the application**
write	**Transfer data from the application to the device**
close	**Terminate use of the device**
seek	**Move to a new location of data on the device**
ioctl	**Miscellaneous control functions (e.g., change volume)**

Six basic I/O functions that comprise the open/read/write/close paradigm. The names are takes form the Unix OS.

Cost of I/O operations

- Cost of invoking a library function is low
  - calling a function in run time, the cost is same as calling a procedure
  - reason, copy of code of library function incorporated into application
- Call for an I/O operation such as read write is extremely high
  - control must pass through a system call to appropriate device driver in OS

Cost of I/O operations

- Why is a system call cost high
  - processor must change privilege mode, changing from application to OS mode
  - processor must change address space from application virtual address space to OS address space
  - processor must copy data between the application address space and the OS address space
- System call overhead is associated with the call rather than the work performed by the driver.
- To optimize performance, minimize the number of system calls

Reducing the System Call Overhead

The overhead involved in using a system call to communicate with a device driver is extremely high; a system call is much more expensive than a conventional procedure call such as the calls used to invoke a library function.

To reduce overhead and optimize I/O performance, a programmer must reduce the number of system calls that an application invokes. The key to reducing system calls involves transferring more data per system call.

The Buffering Concept

The buffering principle: to reduce the number of system calls on output, accumulate data in a buffer, and transfer mode data each time a system call is made.

- To automate buffering, we need a scheme that works for any application. Use fixed size buffer and a set of library functions.

The Buffering Concept

Function	**Meaning**
fopen	**Set up a buffer**
fgetc	**Buffered input of one byte**
fread	**Buffered input of multiple bytes**
fwrite	**Buffered output of multiple bytes**
fprintf	**Buffered output of formatted data**
fflush	**Flush operation for buffered output**
fclose	**Terminate use of a buffer**

Examples of functions included in the standard I/O library used with the Unix operating system. The library includes additional functions .

Summary

- Two aspects of I/O pertinent to programmers
  - system programmers writing device driver code must understand low-level details of device
  - application programmer must understand relative costs
- Device driver is divided into three parts
  - upper half that interacts with application program
  - lower half that interacts with the device itself
  - set of shared variables
- Buffering is used to optimize I/O performance
  - can be used at input and output
  - reduces system call overheads substantially