Chapter 16

Input and Output
Topics

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

CS250 -- Part V
Dr. Rajesh Subramaniam, 2005
Introduction

This chapter covers

− how driver implements read and write
− device driver
− application software
− software needed to control device
− I/O from programmers perspective
Device Driver

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

Device 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
  - Application programs and relegates them to the driver

- Device Independence
  - Device driver removes all hardware details from application programs

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
  - Code to interact with an application
  - Code to handle device details
  - Code to communicate over a bus

Organizational

- Request I/O operations
  - Upper half: functions that are invoked by applications to occur
  - Lower half: handler that is invoked when an interrupt occurs
  - Shared variables: to hold state information needed to coordinate the two halves

Dr. Rajesh Subramaniam, 2005
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.

Conceptual Parts of a Device Driver

device hardware

- Interrupts
  - Invoked by lower half

- Shared variables

- Applications
  - Invoked by upper half

Applications programs
Types of Devices

- Based on the interface type, devices can be divided into two categories:
  - Block-oriented devices: Transfers a block of data at a time
  - Character-based devices: Transfers single byte of data at a time
Example Flow Through a Device Driver
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.
Initialization (computer system starts)

1. Initialize input queue to empty

Upper half (application performs write)

1. Initialize input queue to empty

2. Use the CSR to request an interrupt

Lower half (interrupt occurs)

1. If the queue is empty, stop the device from interrupting

2. If the queue is nonempty, extract an item and start output

3. Return to application

Start output

1. If the queue is nonempty, extract an item and go back to application
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 an interrupt immediately. If an interrupt is destined to occur, device finishes current operation and then generates and interrupted, if device is idle.

- Device includes a CSR bit that a processor can set to force an interrupt.
Device driver can use queuing for input operations.

Queued Input Operations
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. If an application is stopped, start another input operation

Lower half (interrupt occurs)

1. If input queue is not full, start another input operation
2. Extract the next item from the input queue
3. Return the item to the application

Return from interrupt

Looking Ahead (application performs read)

1. Initialize input queue to empty
2. Force the device to interrupt
3. Return the item to the application
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

- Bi-directional device
  - Supports data transfer in two directions, processor to device and vice versa

- Unidirectional device
  - Supports data transfer in one direction

- Bidirectional Data Transfer

Approaches to bi-directional transfer

- Treat the device as a single device that handles two types
  - Input and one used for output
  - Treat the device as if it were two separate devices, one used for input and one used for output

- 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

Asynchronous vs Synchronous programming paradigm

• Synchronous
  – polling is 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
- Processor temporarily restricts use of bus
- Test-and-set instructions

What happens if device tries to read list just before processor (command chaining)

Device follows the list and performs operations automatically

Processor creates a linked list of operations in memory, smart

Problems with Asynchronous Programming
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. Such as files or windows.
Run time I/O Libraries

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

**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. The run-time library acts as an intermediary and manages the underlying I/O mechanisms.

Advantages of using run-time libraries:

- Flexibility and ease of change: In case of changes, only the run-time library needs to be updated, the compiler remains unchanged.
- Only library function understands how to use the underlying I/O mechanisms.

Run-time I/O libraries
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?

- What is the interface between the run-time library and the operating system?

- What are the relative costs of using the two interfaces?
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

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

- Interface between run-time library and the OS

I/O Operations and OS Supports
I/O Operations and OS Supports

• Example to read/write a CD (contd ..)
  - seek to move to new position
  - ioct1 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

• Example to read/write a CD (contd ..)
I/O Operations and OS Supports

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>Prepare a device for use (e.g., power up)</td>
</tr>
<tr>
<td>read</td>
<td>Transfer data from the device to the application</td>
</tr>
<tr>
<td>write</td>
<td>Transfer data from the application to the device</td>
</tr>
<tr>
<td>close</td>
<td>Terminate use of the device</td>
</tr>
<tr>
<td>seek</td>
<td>Move to a new location of data on the device</td>
</tr>
<tr>
<td>ioctl</td>
<td>Miscellaneous control functions (e.g., change volume)</td>
</tr>
</tbody>
</table>

The names are taken from the Unix OS. Six basic I/O functions that comprise the open/read/write/close paradigm.
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
  - Cost of invoking a library function is low

- Call for an I/O operation such as read/write is extremely high
  - reason, copy of code of library function incorporated into application
  - control must pass through a system call to appropriate device driver in OS
Cost of I/O operations

- Why is system call cost high
  - Processor must change privilege mode, changing from application mode to OS privilege mode.
  - Processor must change address space from application 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 such as the calls used to invoke a library function. The key to reducing system calls involves application invokes. To reduce overhead and optimize I/O performance, a programmer must reduce the number of system calls that an application invokes. To reduce overhead and optimize I/O performance, a programmer must reduce the number of system calls that an application invokes. To reduce overhead and optimize I/O performance, a programmer must reduce the number of system calls that an application invokes.
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>fopen</td>
<td>Set up a buffer</td>
</tr>
<tr>
<td>fscanf</td>
<td>Terminates use of a buffer</td>
</tr>
<tr>
<td>fgets</td>
<td>Flushed operation for buffered output</td>
</tr>
<tr>
<td>fgets</td>
<td>Buffered output of formatted data</td>
</tr>
<tr>
<td>fwrite</td>
<td>Buffered output of multiple bytes</td>
</tr>
<tr>
<td>fprintf</td>
<td>Buffered input of one byte</td>
</tr>
<tr>
<td>fclose</td>
<td>Set up a buffer</td>
</tr>
</tbody>
</table>

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

Two aspects of I/O pertinent to programmers

Summary