Computer Organization

Douglas Comer

Computer Science Department
Purdue University
250 N. University Street
West Lafayette, IN 47907-2066

http://www.cs.purdue.edu/people/comer

XVI

A Programmer's View Of I / O And Buffering

Device Driver

- Piece of software
- Responsible for communicating with specific device
- Usually part of operating system
- Classified as *low-level code*
- Manipulates device's CSRs
- Handles interrupts from device

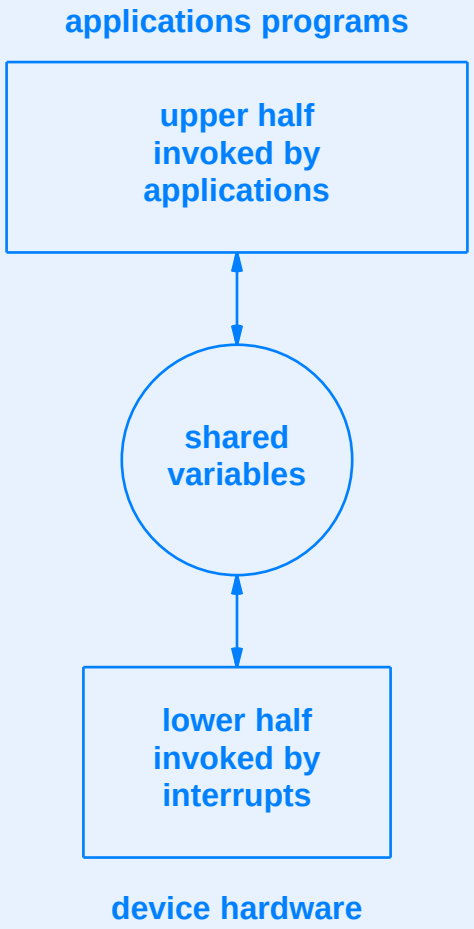
Purposes Of Device Driver

- Device independence: application is not written for specific device(s)
- Encapsulation and hiding: details of device hidden from other software

Conceptual Parts Of A Device Driver

- Lower half
  - Handler code that is invoked when the device interrupts
  - Communicates with device
- Upper half
  - Functions that are invoked by applications
  - Allow application to request I/O operations
- Shared variables
  - Used by both halves to coordinate

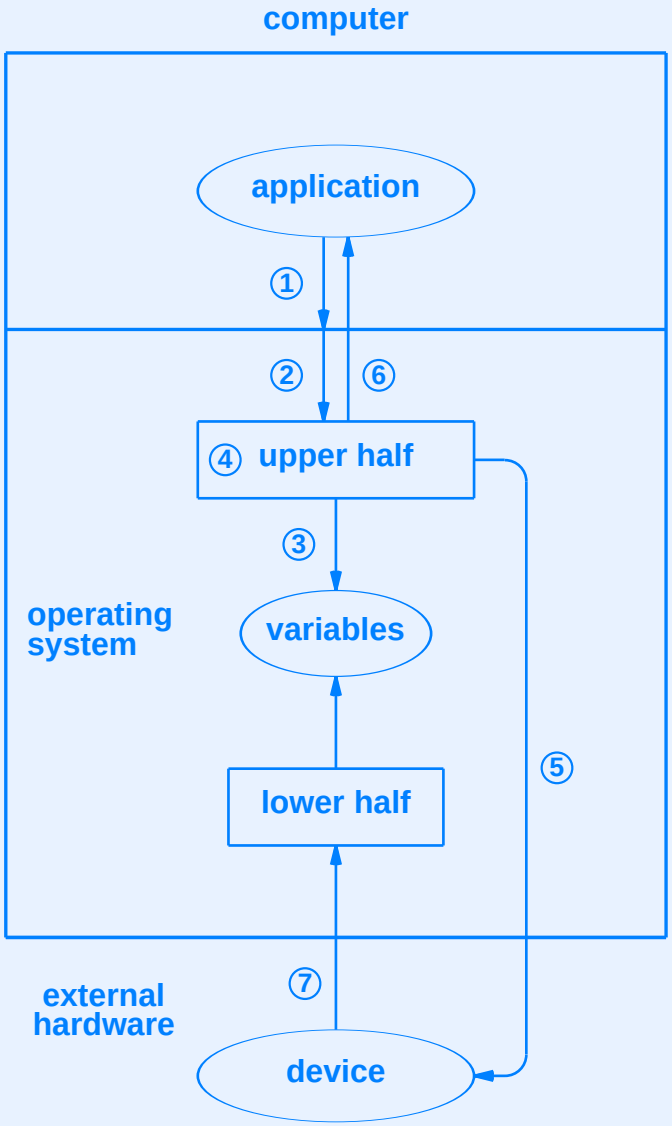
Illustration Of Device Driver Organization



Types Of Devices

- Character-oriented
  - Transfer one byte at a time
  - Examples
    - Keyboard
    - Mouse
- Block-oriented
  - Transfer block of data at a time
  - Examples
    - Disk
    - Network interface

Example Flow Through A Device Driver



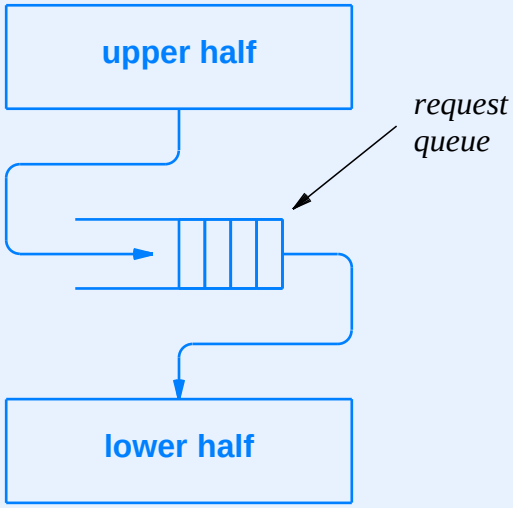
Steps Taken

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

Queued Output Operations

- Used by most device drivers
- Shared variable area contains queue of requests
- Upper-half places request on queue
- Lower-half services request from queue

Illustration Of A Device Driver Request Queue



- Queue is shared among both halves

Steps Taken On Output

- Initialization (computer system starts)
  - Initialize input queue to empty
- Upper half (application performs *write*)
  - Deposit data item in queue
  - Use the CSR to request an interrupt
  - Return to application
- Lower half (interrupt occurs)
  - If the queue is empty, stop the device from interrupting
  - If the queue is nonempty, extract an item and start output
  - Return from interrupt

Forcing An Interrupt

- A device has a CSR bit, B, that is used to force the device to interrupt
- If the device is idle, setting bit B causes the device to generate an interrupt
- If the device is currently performing an operation, setting bit B has no effect
- Above makes device driver code especially elegant

Queued Input Operations

- Initialization (computer system starts)
  - Initialize input queue to empty
  - Force the device to interrupt
- Upper half (application performs *read*)
  - If input queue is empty, temporarily stop the application
  - Extract the next item from the input queue
  - Return the item to the application
- Lower half (interrupt occurs)
  - If the queue is not full, start another input operation
  - If an application is stopped, allow the application to run
  - Return from interrupt

Devices That Support Bi-Directional Transfer

- Most devices include two-way communication
- Example: although printer is primarily an output device, most printers allow the processor to check status
- Drivers can
  - Treat device as 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 Paradigm

- Synchronous programming
  - Used for many applications
  - Processor follows single path through the code
- Asynchronous programming
  - Used for interrupts
  - Programmer writes set of handlers
  - Each handler invoked when corresponding event occurs
  - More challenging than synchronous programming

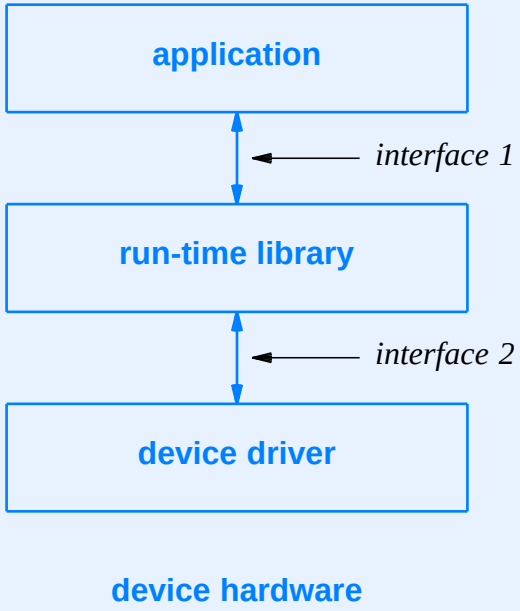
Mutual Exclusion

- Needed for asynchronous events
- Guarantees only one operation will be performed at a time
- For device drivers: must provide mutual exclusion between processor and smart device that change shared data

I/O As Viewed By An Application

- Few programmers write device drivers
- Most programmers use high-level abstractions
  - Files
  - Windows
  - Documents
- Compiler generates calls to *run-time library* functions
- Chief advantage: I/O hardware and/or device drivers can change without changing applications

Conceptual Arrangement Of Library And OS



- Well-known example
  - Standard I/O library
  - Unix kernel

Functions In the Unix Operating System's Open/Read/Write/Close Paradigm

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)

Cost Of I/O Operations

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 call used to invoke a library function.

Reducing System Call Overhead

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.

Buffering

- Important optimization
- Used heavily
- Automated and usually invisible to programmer
- Key idea: make large I/O transfers
  - Accumulate outgoing data before transfer
  - Transfer large block of incoming data and then extract items

Automating Buffering

- Typically performed with *library functions*
- Application
  - Uses functions in the library for all I/O
  - Transfers data in arbitrary size blocks
- Library functions
  - Buffer data from applications
  - Transfer data to underlying system in large blocks

Example Library Functions For Output

Operation	Meaning
setup	Initialize the buffer
input	Perform an input operation
output	Perform an output operation
terminate	Discontinue use of the buffer
flush	Force contents of buffer to be written

- Functions are analogous to those provided by operating system

Use Of Library

- *Setup*
  - Called to initialize buffer
  - May allocate buffer
  - Typical buffer sizes 8K to 128K bytes
- *Output*
  - Called when application needs to emit data
  - Data sent to OS only when buffer is full
- *Terminate*
  - Called when all data has been emitted
  - Forces remaining data to be sent to OS

Implementation Of Output Buffer Functions

Setup(N)

1. Allocate a buffer of N bytes.
2. Create a global pointer, p, and initialize p to the address of the first byte of the buffer.

Output(D)

1. Place data byte D in the buffer at the position given by pointer p, and move p to the next byte.
2. If the buffer is full, make a system call to write the contents of the entire buffer, and reset pointer p to the start of the buffer.

Implementation Of Output Buffer Functions (continued)

Terminate

1. If the buffer is not empty, make a system call to write the contents of the buffer prior to pointer p.
2. If the buffer was dynamically allocated, deallocate it.

Flushing A Buffer

- Allows a programmer to control buffering
- Needed for interactive programs
- When *flush* is called
  - If buffer contains data, library sends to OS
  - If buffer is empty, *flush* has no effect

Implementation Of Flush

Flush

1. If the buffer is currently empty, return to the caller without taking any action.
2. If the buffer is not currently empty, make a system call to write the contents of the buffer and set the global pointer p to the address of the first byte of the buffer.

Implementation Of Terminate

- Call flush
- Proceed to deallocate buffer

Summary Of Buffer Flushing

A programmer uses a flush *function to specify that outgoing data in a buffer should be sent to the device driver in the operating system. A* flush *operation has no effect if a buffer is currently empty.*

Buffering On Input

Setup(N)

1. Allocate a buffer of N bytes.
2. Create a global pointer, p, and initialize p to indicate that the buffer is empty.

Input(N)

1. If the buffer is empty, make a system call to fill the entire buffer, and set pointer p to the start of the buffer.
2. Extract a byte, D, from the position in the buffer given by pointer p, move p to the next byte, and return D to the caller.

Terminate

1. If the buffer was dynamically allocated, deallocate it.

Important Note About Implementation

- Both input and output buffering are straightforward
- Only a trivial amount of code needed

Effectiveness Of Buffering

- Buffer of size *N* reduces number of system calls by a factor of *N*
- Example
  - Minimum size buffer is typically 8K bytes
  - Resulting number of system calls is S / 8192, where *S* is the original number of system calls

Buffering In An Operating System

- Buffering is used extensively inside the OS
- Important part of device drivers
- Goal: reduce number of external transfers
- Reason: external transfers are slower than system calls

Relation Between Buffering And Caching

- Closely related concepts
- Chief difference
  - Cache handles random access
  - Buffer handles sequential access

Buffering Example
(The Unix Standard I/O Library)

- Widely used
- Speeds I/O considerably

Functions In The Unix Standard I/O Library

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

Summary

- Two aspects of I/O pertinent to programmers
  - Device details important to systems programmers who write device drivers
  - Application programmer must understand relative costs of I/O
- Device driver divided into three parts
  - Upper-half called by application
  - Lower-half handles device interrupts
  - Shared data area accessed by both halves

Summary
(continued)

- Buffering
  - Fundamental technique used to enhance performance
  - Useful with both input and output
- Buffer of size *n* reduces system calls by a factor of *N*


Questions?