Nathan Tung

CS111 Two Articles Outline #14

An Introduction to Device Drivers; Understanding Modern Device Drivers

- Device Drivers: entry point for kernel hackers to learn the code without being overwhelmed by complexity
    - Modularity: black box that hides details; enables hardware to respond to a well-defined interface
        - Easy to write drivers
    - Drivers can be built separately from the kernel and "plugged in"
    - Turnover rate of hardware technology is fast; adding a driver means more users will use a hardware
        - Also, the source can be released
- Role of Device Driver
    - Provides mechanism, not policy (and is therefore flexible)
        - Mechanism vs policy: capabilities provided vs. how capabilities are used
        - Graphic display example: server vs. window/session managers
        - ftpd: file transfer mechanism
    - Better to have policy-free drivers, since they can be used more freely for different purposes
        - Applications themselves can implement policy (floppy vs. floppy permissions/properties, etc.)
        - Easier to implement and easier to use
    - Flexible driver needs to have capabilities without constraints
    - Sometimes policy is necessary so the hardware is used correctly (ex: handle certain bits in I/O driver)
    - Device driver sits between hardware and application; programmer has control over device appearance
    - Tradeoff between providing options to user versus keeping it simple
        - Programmer has freedom in implementing concurrency, memory mapping, etc.
- Splitting the Kernel
    - Concurrent processes do different tasks
    - Kernel (executable code that handles resource requests) is split into different roles
        - Process management
            - Creates/destroys processes, connects them outside, schedules CPU sharing in processes
        - Memory management
            - Policies for dealing with memory, builds virtual address space
        - Filesystems
            - Structured filesystem on top of hardware; types include ext3, FAT, etc.
        - Device control
            - Coordinates device drivers for operations mapped to a physical device
        - Networking
            - Packets are collected, identified, and dispatched; controls programs' network aspect
- Loadable Modules
    - Ability to extend/remove kernel features at runtime
    - This code (modules) can be linked/unliked to kernel by insmod/rmmod
- Classes of Devices and Modules
    - Better use a module per functionality to allow decomposition for scalability/extendibility
    - Character devices
        - Implements behavior concerning streams of bytes (open, close, write, read)
    - Block devices
        - Device that hosts a filesystem; transfers many bytes at once
    - Network interfaces
        - Interface facilitates network transactions to exchange data between hosts
    - Universal Serial Bus (USB)
    - Filesystems can also determine the amount of info in a block device, etc.
        - Software driver that maps low-level and high-level data structures
- Security Issues
    - Security checks in system are enforced by kernel code
    - Security policy best handled under kernel at high level
    - Driver writers must know when to put security policy that prevents adverse effects
    - They must also avoid bugs (buffer overflow, etc.)

  - Be careful when running precompiled binaries
- Version Numbering
  - Every Linux software has its own release number
  - Even number kernels are stable for general distribution; odd ones are development and ephemeral
- License Terms
  - Linux is under version 2 of GNL General Public License (GPL)
    - Allows growth of knowledge by allowing everyone to modify program

- Understanding Modern Device Drivers
  - Device drivers are biggest OS contributors
  - Driver code:
    - Common assumptions: characteristics of driver code functionality?
    - Interactions: How do they interact with kernel, devices, and buses?
    - Contents: Can we reduce complexity/driver size using libraries?
- Introduction
  - Device drivers are 70% of Linux code base
  - Understanding driver code to improve reliability and reduce complexity
- Background
  - Communication request and OS service access: driver and kernel
  - Executing operations: driver and device
  - Communication management: driver and bus
  - Driver/Device
    - Character drivers (byte-stream), block drivers (random block access), network drivers (packet streams)
  - Driver Research Assumptions
    - Interaction assumptions refer to how driver interacts with kernel
    - Architecture assumptions refer to role of the driver
    - Some assumptions
      - Drivers support single chipset (like efforts to synthesize drivers from formal specs)
      - Drivers are conduit of communicating data and signaling device; it does little processing
- What do Drivers Do?
  - Translates between high level inputs to low level, hardware instructions (I/O)
  - Not fully accurate…
  - Methodology
    - DrMiner (static analysis tool) uses tagging to label entry points of driver code
  - Function breakdown of driver code
    - Mostly initialization and cleanup; only 23% of driver code request handles and interrupts
      - Optimization therefore depends on initialization and configuration
    - Research needs to look more into media, GPU, and wireless drivers (rapidly changing ones)
      - Usually we focus on generally important ones like Ethernet and sound
  - Do drivers belong to classes?
    - Most driver functionality falls into class behavior, but quite a few do not
  - Do drivers do significant processing?
    - Yes, a substantial amount do some processing
  - How many device chipsets does a single driver support?
    - Multiple chipsets per driver; efficient; avoid system that uses unique drivers for chipsets
      - This increases complexity, expands driver code
  - Most assumptions are correct, but not all the time
- Driver Interactions
  - Four topics
    - Kernel resources consumed by driver
    - How/when drivers interact with devices
    - Differences in driver structure across I/O buses
    - Threading/synchronization model used by driver code

  - Driver/kernel interaction
    - Classified into one of five
      - Kernel library (generic support routines, standard data structures)
      - Memory management (allocation)
      - Synchronization (locks)
      - Device library (subsystem libraries supporting class of device or I/O)
      - Kernel services (access to other subsystems like files, memory, scheduling)
    - A large amount is generic routines
  - Driver/device interaction
    - Perform I/O is a function that does I/O on itself or calls I/O function
    - Number and type of device interactions vary, so their cost varies as well
  - Driver/bus interaction
    - Drivers are meant for devices that attach to some PCI (or USB)
    - PCI devices' flexibility/performance come at cost of complexity and less standardized interface
  - Driver Concurrency
    - Drivers are required to multiplex access to device
    - Threaded code needs to be run in kernel (hard to do otherwise)
- Driver Redundancy
  - Related devices share code with a small amount of per-device code
  - Reducing complexity
    - Reduce driver shape down to single signature value
    - There are many opportunities to reduce volume of code by abstraction
      - Procedural abstractions for driver sub-classes
      - Better multiple chipset support
      - Table driven programming
  - Procedural abstraction is most useful (moves shared code to library and provides parameters)