Methods to protect proprietary components in device drivers

Matt Porter

Embedded Alley Solutions, Inc.

Embedded Alley

Introduction

- Why the interest in closed drivers on Linux?
  - Competition
    - Advantage perception
  - Upsell
  - Policy
- Disadvantages of closed drivers
  - Can't move upstream
    - Lack of review
    - No 3rd party bug fixes
    - FOSS alternatives will be preferred
  - User forced to rely on vendor updates

Implementing a closed driver

- Why not take the easy route?
  - Binary modules
  - Not really easy
    - Don't forget EXPORT_SYMBOL_GPL
- What are the risks?
  - Legal
    - Many legal opinions against binary only modules
  - Community
    - Unpopularity of binary only modules

Implementing a closed driver

- User-space drivers
  - Low legal risk
  - Unaffected by EXPORT_SYMBOL_GPL
  - A lot of infrastructure is available
    - Mmap()
    - Select()/Poll()/Read()
    - UIO

The approach

- Design the driver properly
  - Driver can be split into kernel and user portions
  - Kernel portion has
    - Memory alloc/free
      - Bootmem, kmalloc, dma_alloc*()
    - Exports access to registers
    - Provides address translation information for memory regions.
    - Top-half interrupt handling

The approach

- User portion has
  - Driver specific memory management of kernel allocated regions
    - DMA buffers
  - Register access
  - Bottom-half interrupt handling
- User-space portion can be distributed in binary form as with any application.

Register Access

- Mmap()
  - Kernel portion exports all necessary register regions
    - Pgprot_*() settings are critical
  - User portion uses mmap() to map all register regions into user address space for direct access.

Memory Allocation

- Devices often have specific buffer characteristic requirements
  - Physically contiguous
  - Alignment restrictions
- Allocate memory in the kernel portion using appropriate kernel allocation technique
- User portion may manage subsections of a larger contiguous allocation as need.

Managing DMA

- Use of kernel side memory allocation as described
- Kernel driver portion provides information on bus addresses and cache mode of memory allocations.
- User driver portion implements buffer cache management routines as required by buffer cache mode.

Handling Interrupts

- Kernel driver handles interrupt work that must be done at all costs (error handling, etc.)
- User driver handles the interrupt workload
- User interrupt event signaled by kernel driver by unblocking a system call (read() or ioctl())

Role of UIO

- Mainline kernel user-space driver infrastructure
  - Provides
    - Standard kernel side driver infrastructure for user-space driver clients
    - Interrupt event notification
    - Standardized access to multiple memory regions via mmap().
  - Does not have
    - DMA support
      - Hooks for specialized buffer allocation
        - on demand allocation
        - Large physically contiguous allocations
        - Alignment restrictions
      - Cache mode and bus address information

GPU Driver – Introduction

- Production user-space driver example
- Platform
  - ARM926-based SoC with GPU
- Goal
  - Port an existing OpenGL ES and GPU driver implementation from an RTOS to Linux

GPU Driver – Requirements

- Top performance
- Deliver functional and accelerated OpenGL ES on Linux as fast as possible
- Maintain common code base with RTOS implementation
- Driver must be kept proprietary

GPU Driver – RTOS implementation

- OpenGL ES library and GPU driver are tightly coupled
- Assumes 1:1 mapping of bus and virtual addresses for GPU DMA buffer management
- Manages interrupt hardware directly

GPU Driver – Linux Implementation

- Linear FB driver exists already
  - Allocate contiguous GPU memory space from FB driver (cmdline options to control size)
  - mmap() access to base regs, gpu regs, linear FB mem, and GPU mem
  - ioctl provides memory region offset info and physical address of GPU mem.
  - Driver handles GPU irq error events and provides irq events via a FB driver specific ioctl

GPU Driver – Linux Implementation

- Original GPU driver now lives in Linux user-space
  - Extended with OS init routine to provide GPU memory virtual and bus address base.
  - Translation function used to convert buffer address to bus address.
  - User-space DMA buffer allocator manages kernel-allocated GPU mem pool
  - GPU command list complete interrupt handled using ioctl event notification.

GPU Driver – Results

- Automobile model render demo
  - 60 FPS
  - 30% CPU

User-space driver future

- UIO is a good start
- UIO primarily needs support for DMA to enable a broader range of drivers
  - Driver-specific memory allocation and region characteristic info
- Plan is to convert this driver to a UIO driver with DMA helper extensions
  - Some thought still required on how to properly abstract these extensions
  - Need an open user-space DMA driver example to help drive acceptance of any extensions.

Conclusions

- User-space drivers aren't just for simple automation drivers
- With proper design, DMA-driven devices can be supported in user-space

Q&A

- Questions?