ECE 531 – Advanced Operating Systems
Lecture 26

Vince Weaver

https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

7 November 2025

Announcements

- Homework #7 will be posted
- Project Topics were due, responded to them

HW#5 Notes – Code

- Converting shell to userspace
  People seemed to handle that fine
- Adding `time()` syscall
  Also seemed to handle that OK
- Aside on `time()`, why use pointer *and* return value
  - Returning r0 is actually harder than you think. Registers are saved at start of IRQ handler and restored so just setting r0 will be over-written. Need to update the saved value. This is harder when we try to let gcc

do the registers for us

- Passing back a pointer too is classic UNIX dating back to 1972. Probably because time() (Seconds since epoch) has always been a bit big and so often you are trying to return a value that won't fit in a return value. See modern 64-bit time_t

HW#5 Questions – getchar()

- Nonblocking getchar – what happens if in kernel mode waiting for `serial_read()` to finish and a timer interrupt comes in?

HW#5 Questions – Userspace

- Why run in userspace? – mostly so kernel can protect system from rogue apps
- Note for HW#5 it does help a bit, but without virtual memory protection it's still fairly unsafe

HW#5 Questions – Kernel Mode

- Changing back to kernel mode – syscall, interrupt, reset(?)

HW#5 Questions – Kernel Mode

- Security implications of kernel writing to a user-supplied pointer.
- Linux for example uses `copy_to_user()` that checks to see if area pointed to actually belongs go user process in question.
- Can we do that?
- Does the kernel know what memory belongs to a process?
- What happens if the pointer we follow is to a page that is paged out? Linux "pins" the memory in this case but

not sure what that means.

HW#5 Questions – Other Syscalls

- System call of choice
- manpages, from section “2”
- many operate on file descriptors
- chmod, inotify, exit, fork, truncate, futex, stat, wait
- surprised no one said perf_event_open

Linux Graphic Interface History

- originally, none. VGA Text only
  X11 or SVGAlib drove software directly (bypassing OS)
- Attempt at GGI/KGI, Linus nixed it
- Framebuffer devices got in. Why? Well some machines had no textmode without it
- Gradually the DRI interface (Direct Rendering Interface) became standard

Highest Level

- Applications want to draw to the screen
- Usually you do this through some sort of library
- Generally many layers of abstraction between you and hardware
- Possibly everything is running in a web-browser these days

Desktop Environments

- Provides interface and some default GUI apps
- Unlike some OSes, there's no "standard" desktop environment
- GNOME
- KDE
- XFCE4
- Many others

Graphics Toolkits

- Libraries that provide standard GUI interfaces, things like windows, tabs, buttons, dialog boxes, color pickers, etc
- Qt (c++ based, used for KDE)
- Gtk+ (c-based, used for GNOME and others)
- Motif (obsolete one used in old days)
- Various others, some cross-platform

Windowing Systems

- X11 – traditional one
- Wayland – replacing it

X11

- X11 from MIT, 1984
- Note: just X or X11, not "X windows"
- Client/Server distinction. Server draws the graphics. Takes stream of primitives like "draw box" or "draw line" or "set background color"
- Network transparent
- Really complex protocol
- Used to be many vendors, proprietary and free versions
- Now is deprecated

Wayland

- Compositing Manager
- Draw to an offscreen buffer, window manager copy to screen
- Can have 3d compositor, fancy effects
- Much better suited to modern hardware
- Does the bare minimum, a lot of X11 features are optional and now all Wayland compositors (there are many) support them all
- No network transparency by default

What if you just want to draw into a Framebuffer

- Various apps / libraries / etc support giving you some sort of canvas
- If you want to be closer to hardware (maybe for games) libSDL is often used

Linux Kernel Interface – Direct Rendering Interface (DRI)

- DRI1 – just front/backbuffer shared by all apps. Copied to screen at VSYNC. Obsolete
- DRI2 – not compatible with DRI1
  One backbuffer. Programs each have their own buffer (possibly hardware accelerated). Kernel needs to manage memory on GPU for applications. When time to draw, various buffers merged together.
- DRI3 – (2013) Apps allocate on buffers rather than let

X11 do it? (Read up on this some more)

Other Linux Kernel Interfaces

- DRM – (direct rendering manager?) event queuing?
- KMS – kernel mode setting, sets resolution, refresh, monitors, etc. Surprisingly important, what happens if application crashes, want to set back to good mode
- GEM/TTM – memory allocation

Linux 3D

- Typically Linux used OpenGL. The open-source Mesa3D library was used to implement this
- These days maybe Vulkan is taking over

Linux Device Drivers

- NVIDIA – proprietary drivers widely used. Questionable licensing. Can be a pain to keep in sync with kernel There are free reverse-engineered drivers but they have issues too
- AMD – generally has in-kernel drivers
- Intel – generally had good drivers
- Embedded (including Videocore IV) often have hit-or-miss support, sometimes reverse engineered

Raspberry Pi Framebuffer

- Pi *can* do advanced 3D GPU graphics.
  But it is complex, more than we need for a simple OS
- The GPU firmware does provide for a simple flat framebuffer mode if you ask it nicely

Pi GPU interface

http://petewarden.com/2014/08/07/how-to-optimize-raspberry-pi-code-using-its-gpu/

https://github.com/raspberrypi/firmware/wiki/Mailboxes

Pi 1B Addressing Aside

- The documentation on this can get confusing because the GPU has its own MMU separate from the ARM MMU
- The GPU MMU divides memory up into 16MB chunks
- The actual memory view from the GPU is different than the Pi, it has the idea of "bus" addresses. To it, the MMIO region is always as 0x7eff0000 (which is why the BCM2835 manual describes it like that)
- On Pi1B the memory is mirrored 4 times, with the

different regions having different cache settings

- 0x0000.0000 - 0x1FFF.FFFF – fully cached
- 0x2000.0000 – where the MMIO is viewed
- 0x4000.0000 –
- 0x8000.0000 –
- 0xC000.0000 –
- Note I will come back to this as I think it is different on Pi1B vs later models

Raspberry Pi Mailbox Interface

- How the ARM CPU communicates with the GPU that really run things
- Mailbox channels

MAILBOX_POWER	0
MAILBOX_FRAMEBUFFER	1
MAILBOX_VIRT_UART	2
MAILBOX_VCHIQ	3
MAILBOX_LED	4
MAILBOX_BUTTONS	5
MAILBOX_TOUCHSCREEN	6
MAILBOX_PROPERTY_TO_VC	8
MAILBOX_PROPERTY_FROM_VC	9

- Property tags contains a lot of the stuff we get from devicetree

- Temperature
- Clocks
- DMA
- Graphics

Raspberry Pi Mailbox Interface

- Two mailboxes in MMIO space (0x2000b880) . The Pi always reads/receives on Mailbox 0 and writes/sends on Mailbox1

Mbox0 Offset	Mbox1 Offset	Size	Name	Description	R/ W
0x00	0x20	4	Read/Write	Get/Send Mail	R/W
0x10	0x30	4	Poll/Peek	Check mail	R
0x14	0x34	4	Sender	Sender info	R
0x18	0x38	4	Status	Infor	R
0x1c	0x3c	4	Config	Settings	RW

- to send to a mailbox:
  - sender waits until the Mailbox1 Status field has a 0 in the `MAIL_FULL bit`
  - sender writes to Write such that the lowest 4 bits are

the channel to write to, and the upper 28 bits are the message to write.
How can you make the address of the message have the bottom 4 bits be zero? (align directives)

- To read a mailbox:
  - receiver waits until the Mailbox0 Status field has a 0 in the MAIL_EMPTY
  - receiver reads from Read.
  - receiver confirms the message is for the correct mailbox, and tries again if not.
- Talk to GPU through this mailbox interface. Lots of

things set in it (the GPU is in control on Pi). Things like power, clock enables, etc.

Raspberry Pi Framebuffer Interface

Discussed next lecture