Userspace Drivers

ECE 373

What is a userspace driver?

What is a userspace driver?

- A way to drive hardware from outside kernel
- User space program
- Accesses various resources
  - I/O ports
  - Memory regions
  - Control interfaces
- Resources presented by kernel scanning hardware in standardized ways
  - e.g. PCI bus scan
- Not a .ko file!

What isn't a userspace driver?

- Scripts/programs using only sysfs or proc
- Scripts/programs using devfs
  - If /dev entry exists, likely a kernel driver is present
- Exceptions do exist though
  - Graphics drivers can be combination of user and kernel mode
  - Nvidia kmod-nvidia plus Xorg userspace driver
  - More on this later

Drivers of the rings

- Kernel runs in ring 0 on CPU
- Hardware is protected by ring 0 access
- Userspace runs in ring 3 on CPU
- Userspace drivers run in userspace

- But system calls...

What's not available?

- Direct access to all kernel memory
  - Not always a bad thing
  - Requires more maps if you need access
- Interrupts
  - Only fire in kernel space
  - Priority is lost in userspace
- Schedule priority can be a problem
  - Just “another“ userspace process

But first...

What is mapping memory?

- First, man `mmap()`
- Maps files into memory...obviously
- Access with array/pointer operations
- Why is this useful?
- Why not just use `open(), read(), write()`?
- What's the tradeoff?

Files in memory

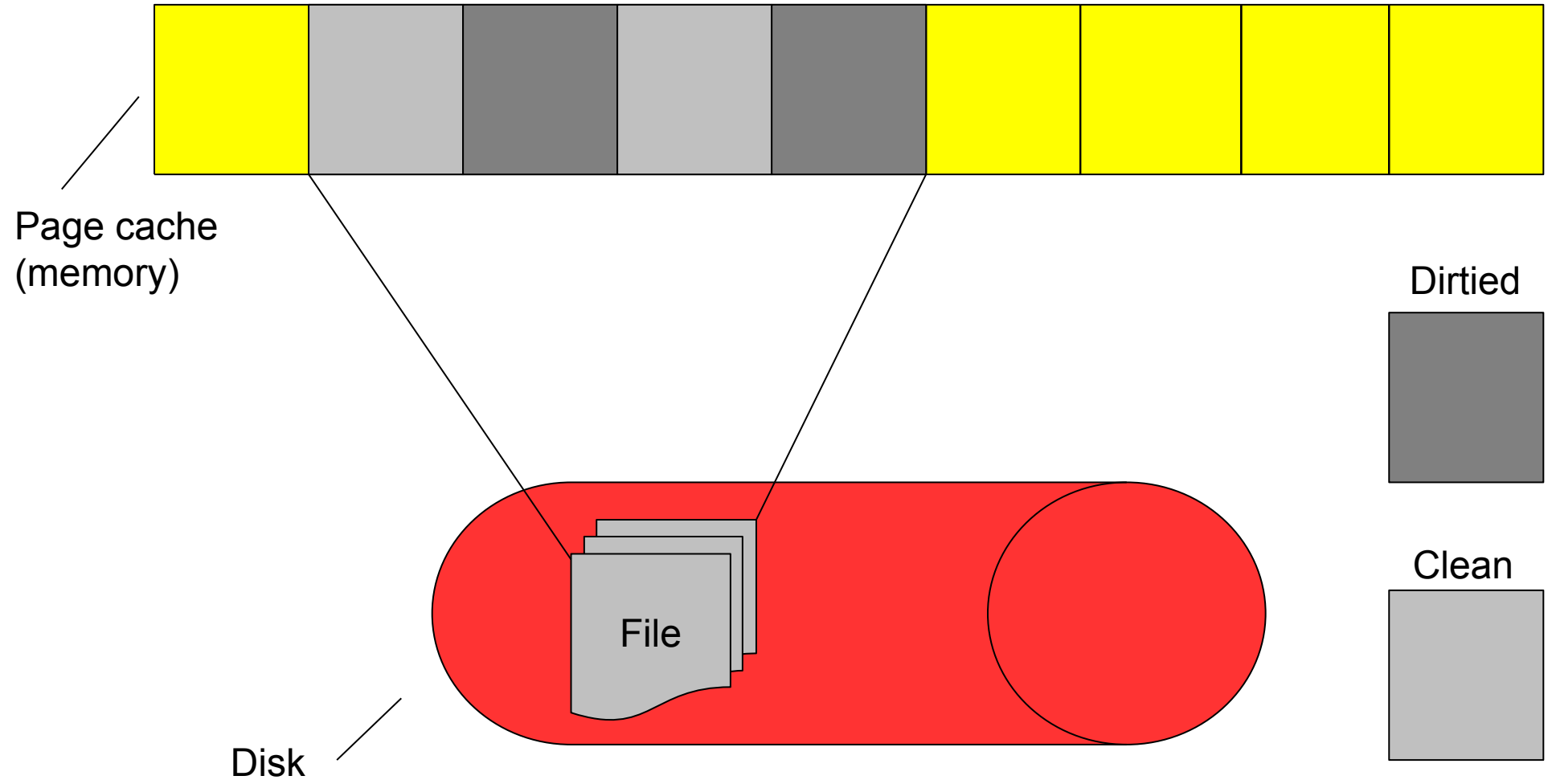
- Advantage to access large files
  - Reading from a file using memory operations can be quick
- Kernel can manage page resources better
  - mmap'd pages backed by page cache
  - Can swap pieces out quickly, versus file page cache writing back before swapping
- Disadvantage – uses up memory map space
  - Limited space in 32-bit addressing

Files in memory, cont.

- Great for sharing file data between processes
  - MAP_SHARED, MAP_ANONYMOUS attributes
  - Page cached once, rewritten on updates, everyone sees update
- Think web server...
  - Multiple threads on multiple files
- Other examples?

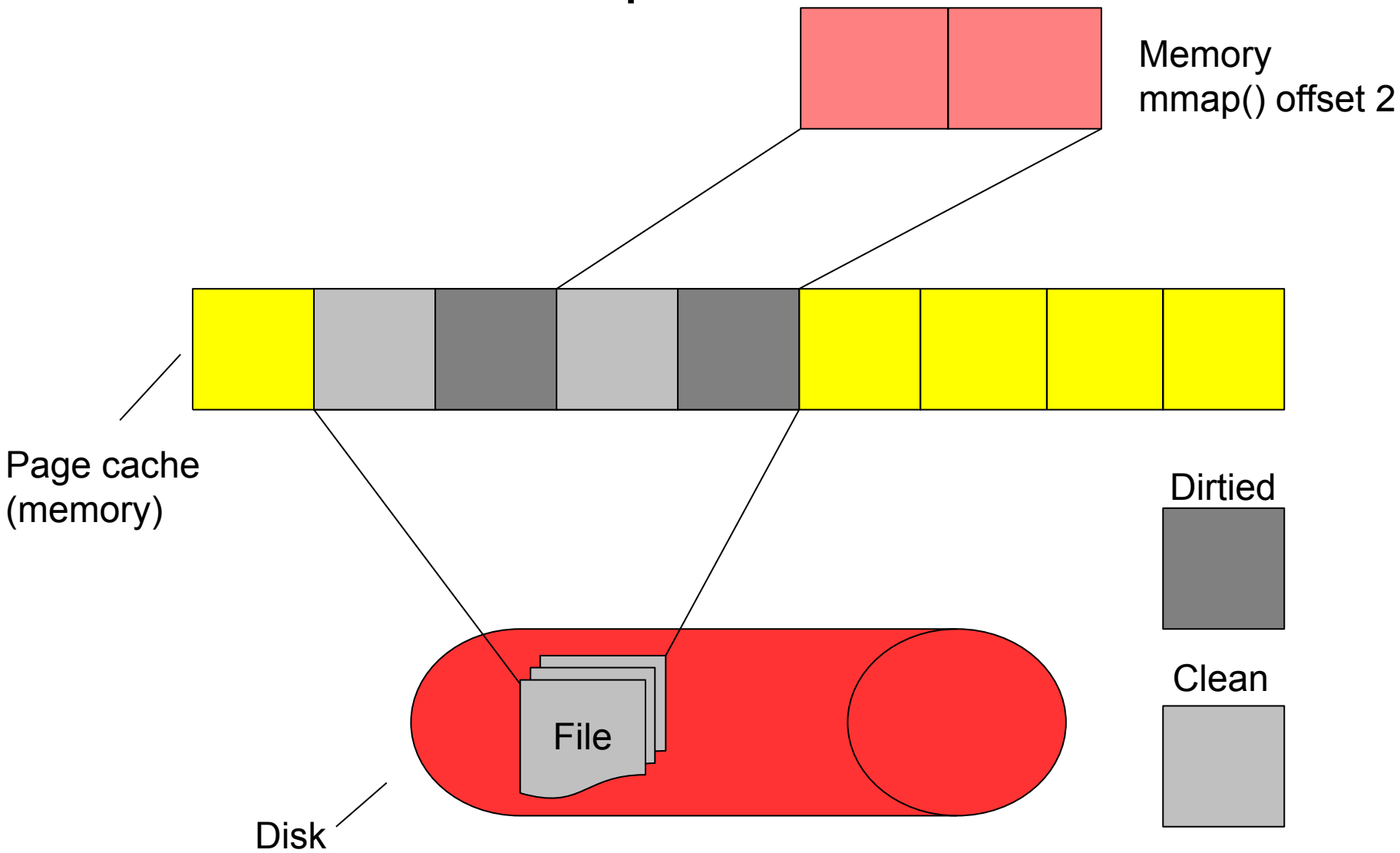
What's it look like?

- What a normal file looks like when opened



What's it look like?

- When a file is mmap'd



Mapping a file is as simple as...

- `open()` the file
- `mmap()` the file
- Inspect the memory region returned from `mmap()`
- `munmap()` the file
- `close()` the file

- Example!

Why use a userspace driver?

Why use a userspace driver?

- Quick prototyping
- Free from kernel ABI changes
  - Application Binary Interface
- Typically doesn't blow up the machine
  - Constrained to mapped space
- Ease of use
  - Can be written in most compiled and scripted languages
- May not have ability to change the kernel

Why not to use a userspace driver?

Why not to use a userspace driver?

- Performance and latency on wake-ups (more on this in a bit...)
- Power efficiency
- No interrupts
- No direct access to kernel infrastructure
  - Kernel times, jiffies, physical memory, traditional DMA, etc.

Userspace drivers in embedded

- Popular for fully-encapsulated devices
  - Touchscreen
  - Battery driver maybe?
- Popular for simple devices
  - LEDs, GPIOs, etc.
  - Arduino, BeagleBone, etc
- Not popular when interfacing with subsystems
  - Networking, audio
- DPDK – Data Plane Development Kit
  - Intel SDK to enable switching infrastructure, kernel bypass

The hybrid approach

- UIO driver a good example
  - https://elixir.bootlin.com/linux/latest/source/drivers/uio
- RDMA drivers
  - OFED stack
- Graphics a perfect example
  - Lightweight portion in kernel, hides some HW interfaces, power management, etc.
  - Heavyweight portion in userspace, interfacing with X server

Performance considerations

- Pure userspace drivers can be quick
  - No interrupts means...?
- Polling can have negative side effects (hurry up and wait!)
- Polling very good for latency-sensitive operations
  - Good for stock trading
  - Good for stop-lights working properly
  - Good for car computers getting the “brake pressed“ signal

What about the kernel?

- Pure userspace approach
  - Must break into the kernel somehow
  - Kernel provides mechanism to get at memory
- Hybrid approach
  - Small shim driver in kernel space
  - Exports driver interface file (/dev) for userspace to drive

Mapping into the kernel

- `/dev/mem`
- It's a file, right?
- man mem...
- `open()` and `mmap()`
- But where to map?

- May require some pleading with the kernel...
  - Linux boot parameter: iomem=relaxed

Mapping sensitive areas

- strace
- dmidecode
- SMBIOS
  - http://en.wikipedia.org/wiki/System_Management_BIOS
- Follow dmidecode...

SecureBoot, oh my

- Attempt at securing EFI to bootloader security
- Trying to prevent rootkit access to the bootloader and OS
- “Signed“ modules used, signed by a CA
  - Guess who's the CA...
- Required for Win8/10 certification on ARM only
- Red Hat, SuSE, others still moving forward with SecureBoot compliance on x86

What does this mean to me?

- SecureBoot wants to lock access to kernel space down
- /dev/mem accesses likely to be prohibited
- Kernel modules (e.g. drivers) will need to be signed
- Details still being worked

Let's mess with hardware

- To `mmap()` successfully on `/dev/mem`, need a place to map
- What are we trying to map on our PCI device to control it?
  - Hint: what do we map in the kernel?
- Where do we find the BAR address?
  - Lspci…
  - Can look in /sysfs...

The basics...

- Great examples exist out there
- E1000regs, aka ethregs, scans bus for devices
- Find device we want
- Grab BAR address from struct, and away we go

- Basic example

What about interrupts??

- Poll, poll, poll
  - Maybe assign one core in a multi-core machine for this
- Using hybrid approach, interrupt routine could only consist of waking userspace thread
- Can generate interrupts if underlying HW supports it
  - Writing register to fire SW-originated interrupt
- New Sapphire Rapids (Intel) CPUs have userspace-aimed interrupts…TBD how well they work!

Reading and an assignment!

- LDD3 pages 37-39
- Essential pages 558-564

- Man pages on `mmap()`
- Read the source!
  - e1000regs (aka ethregs) on e1000.sourceforge.net

- Assignment #5