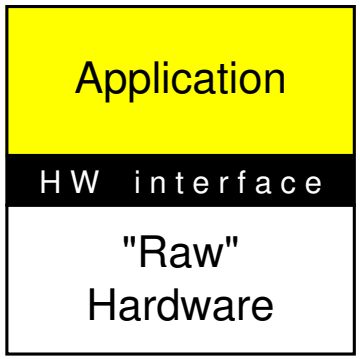
Kernels

ComS 252 — Iowa State University

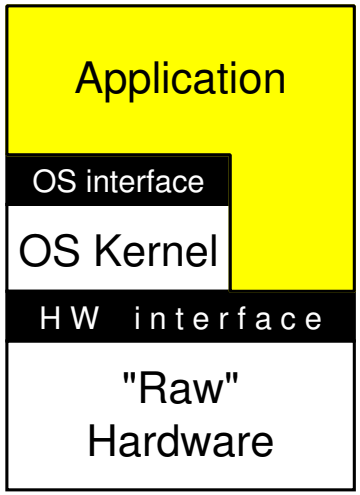
Andrew Miner

The good old days?



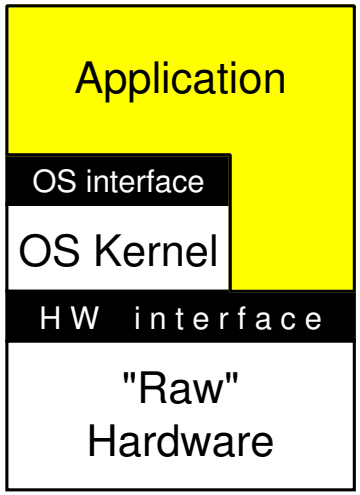
- Applications have direct control of hardware
- No "OS" to get in the way
- Application programmers need to know details of
  - I/O devices
  - Memory space
- No security, no file protections
- No clean way to have multiple processes
  - Anything you do to get multiple processes is a hack

Early PC kernels



- Applications have direct control of hardware
  - If they want it
- OS provides I/O, hardware abstractions
  - No need to know HW details
- No point for OS to provide security
  - Applications can go to HW and bypass it

Early PC kernels

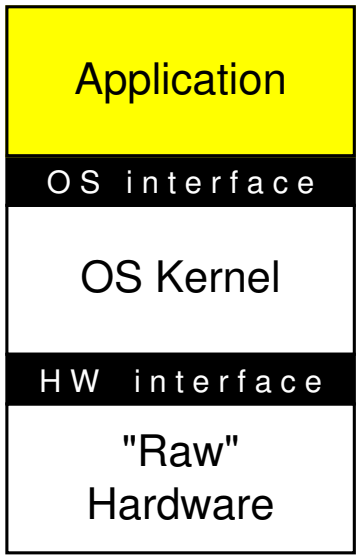


- Applications have direct control of hardware
  - If they want it
- OS provides I/O, hardware abstractions
  - No need to know HW details
- No point for OS to provide security
  - Applications can go to HW and bypass it

Most early personal computers used this model

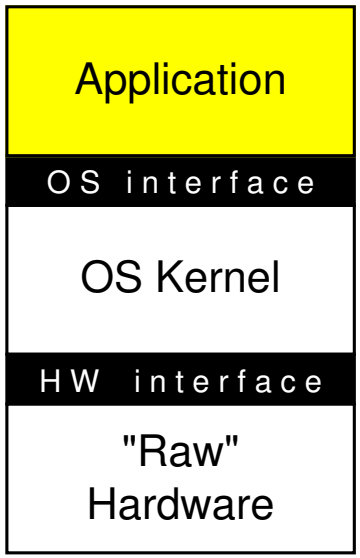
- Apple 2, IBM PC with DOS, Commodore 64, . . .
  - Some of these: "kernel" is in ROM
- This is a fault of the hardware
  - It is impossible to prevent applications from accessing hardware directly

Modern kernels



- Applications cannot access hardware directly
- All hardware access is through the kernel
  - OS interface is via "system calls"
  - More on this later ...
- Necessary for stability
- Necessary for security

Modern kernels



- Applications cannot access hardware directly
- All hardware access is through the kernel
  - OS interface is via "system calls"
  - More on this later ...
- Necessary for stability
- Necessary for security

The rest of lecture deals with this type of kernel

The kernel controls all hardware

CPU

- Processes allowed to execute on CPU only for short time

Memory

- The kernel sets up each process's address space
  - Processes may request additional memory from the kernel

I/O

- All I/O is done by the kernel
  - Processes perform I/O by asking the kernel to do it
- The kernel enforces file permissions

Other kernel responsibilities

- Knows about all devices on the system
  - May require third–party "device drivers"
- Knows how to mount filesystems
  - Can read, write files appropriately
- Schedules processes
  - Chooses the next "ready" process to run
- Handles communication between processes
  - Acts like I/O
  - Pipes are a special case of this
- Delivers signals to processes
  - Invokes the signal handler for that signal

How does the kernel do all this?

How does the kernel do all this?

Magic

How does the kernel do all this?

Magic

- Ok, not really.
- But we will only spend one lecture on the kernel
- Most things will remain “magic”
- For more information, take ComS 352

One big magic trick, revealed

I told you, applications cannot "access hardware directly".
Why not? For example,

- "The kernel sets up each process's address space"
- The kernel does this by executing instructions on the CPU
- What stops a process from executing those instructions?
  - Even if I write an application in assembly language?
  - Even if I write raw machine code?

Hardware requirement: CPU privilege levels

- Modern CPUs have at least 2 privilege levels (also known as rings)
- Applications run at a low privilege level
  - I will call this "user mode"
- The kernel runs at a high privilege level
  - I will call this "kernel mode"
- Intel processors have 4 rings, but normally only two are used
  - Ring 0 for kernel mode
  - Ring 3 for user mode
- To set a process's address space: set some memory registers
  - These instructions require kernel mode
- Performing I/O can be done only in kernel mode
  - Might be privileged instructions
  - Or requires reading/writing to protected memory space

Changing privilege levels

From kernel mode to user mode

- Done by the kernel when setting a process to run; e.g
  1. Choose the next process to run
  2. "Set it up" (details not important for this class)
  3. Switch to user mode and let it run

From user mode to kernel mode

- Done using interrupts (like sending a "signal" to the kernel)
- Doing this guarantees that the kernel gets control again
- All system calls are done in this way
- By the way — this is expensive (slow)
- Many details omitted, sorry

One little magic trick, revealed

How does the kernel schedule a process to run for "a short time"?

One little magic trick, revealed

How does the kernel schedule a process to run for "a short time"?

- Kernel sets a "timer interrupt"
  - Sets a timer to go off when we want to run the scheduler again
  - If process terminates or performs I/O before then:
    - The kernel will regain control when I/O is requested
    - Cancel the timer, run the scheduler
  - If no I/O before then:
    - Timer goes off and kernel regains control
    - Run the scheduler
- Requires hardware support

Kernel space vs. user space

Kernel space

- Memory space reserved for the kernel
- Used for kernel data structures
  - E.g., queue of pending I/O requests for each device
- Used for OS code that runs in kernel mode

User space

- Memory space for user programs and libraries
- Used for code that runs in user mode

Kernel design: two main architectures

Monolithic kernels

- Kernel is like a single software library of system calls
- All functionality provided in kernel space
- System calls run in kernel mode
- E.g., Linux, Windows

Microkernels

- Kernel is minimal
  - Memory and process management, and communication
- Other features are provided by servers
  - Run in user space and user mode; can be restarted
- E.g., MINIX

Stability

What happens when the kernel crashes?

- Kernel panic: Reboot
- No other options

What happens when a user–mode process crashes?

- Kill it if it's not already dead
- If the process is a daemon, re-start it
- Other processes, and kernel, should be unaffected
  - Except maybe for processes that were communicating with the crashed process
- Theory says Microkernels are slower but more stable

Stability, ctd.

What happens when X crashes?

- X is just another user process
- However, X likes to grab your keyboard
- So it may *appear* like the system is hung
- But this is usually not a kernel panic

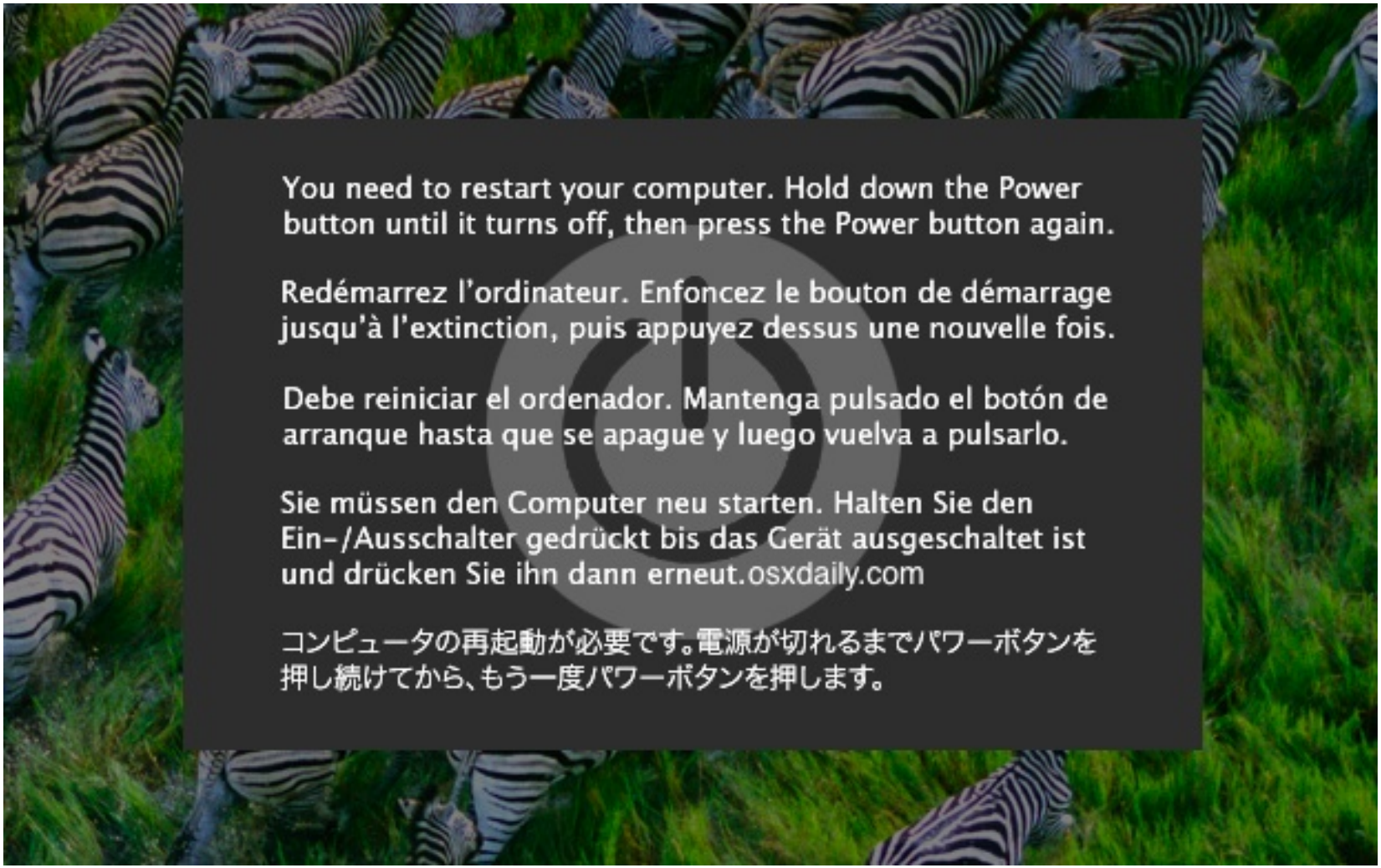
Stability, ctd.

What happens when X crashes?

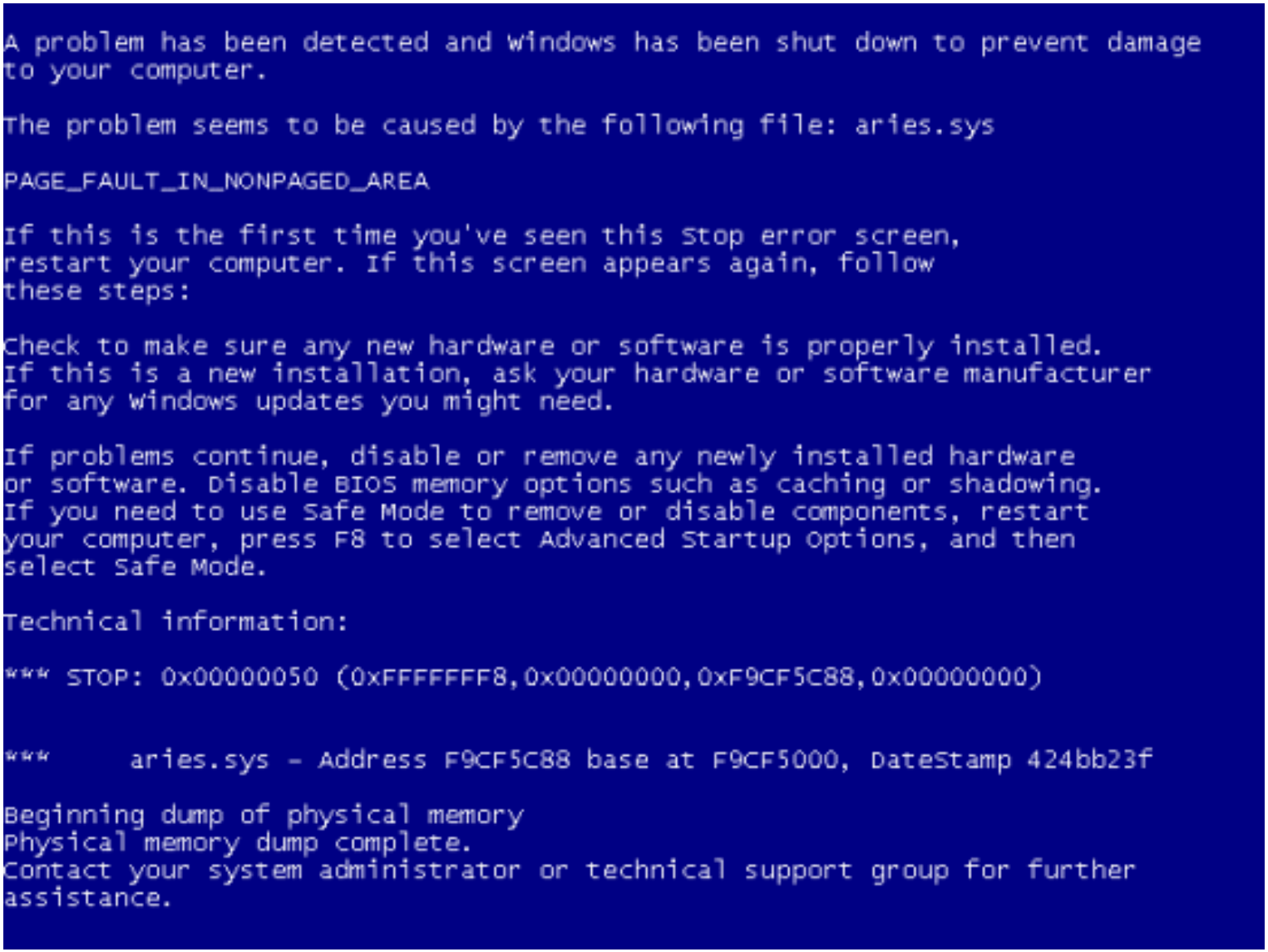
- X is just another user process
- However, X likes to grab your keyboard
- So it may *appear* like the system is hung
- But this is usually not a kernel panic
- Sometimes you can recover if you log in remotely
  - Kill your X processes

```
Call Trace:
 [<c041b7f2>] iounmap+0x9e/0xc8
 [<c053480d>] agp_generic_free_gatt_table+0x2e/0x9e
 [<c0533991>] agp_add_bridge+0x1a8/0x26f
 [<c05439eb>] __driver_attach+0x0/0x6b
 [<c04e6bf4>] pci_device_probe+0x36/0x57
 [<c0543945>] driver_probe_device+0x42/0x8b
 [<c0543a2f>] __driver_attach+0x44/0x6b
 [<c054344a>] bus_for_each_dev+0x37/0x59
 [<c05438af>] driver_attach+0x11/0x13
 [<c05439eb>] __driver_attach+0x0/0x6b
 [<c0543152>] bus_add_driver+0x64/0xfd
 [<c04e6d22>] __pci_register_driver+0x47/0x63
 [<c040044d>] init+0x17d/0x2f7
 [<c0403dee>] ret_from_fork+0x6/0x1c
 [<c04002d0>] init+0x0/0x2f7
 [<c04002d0>] init+0x0/0x2f7
 [<c0404c3b>] kernel_thread_helper+0x7/0x10
 =======================
Code: 78 29 8b 44 24 04 29 d0 8b 54 24 10 c1 f8 05 c1 e0 0c 09 f8 89 02 8b 43 0c
 85 c0 75 08 0f 0b 9c 00 77 c8 61 c0 48 89 43 0c eb 08 <0f> 0b 9f 00 77 c8 61 c0
 8b 03 f6 c4 04 0f 85 a5 00 00 00 a1 0c
EIP: [<c041bd49>] change_page_attr+0x19a/0x275 SS:ESP 0068:c14f7ec0
 <0>Kernel panic - not syncing: Fatal exception
 _
```

Linux kernel panic



Macintosh kernel panic



Windows kernel panic

The Linux kernel

- Still maintained by Linus Torvalds
  - But these days, he has a lot of help
- `http://www.kernel.org`
- Written in C
- Open source (GPL)
- Monolithic
  - But still, very stable
  - Not uncommon to have Linux systems with uptimes in years

The Linux kernel

- Still maintained by Linus Torvalds
  - But these days, he has a lot of help
- `http://www.kernel.org`
- Written in C
- Open source (GPL)
- Monolithic
  - But still, very stable
  - Not uncommon to have Linux systems with uptimes in years

When you say, "I wrote a program that crashed Windows," people just stare at you blankly and say "Hey, I got those with the system, for free."

—Linus Torvalds

Question

- Linux is a monolithic kernel
- Therefore, kernel space includes drivers
- When you run a Fedora "live" CD, you boot into Linux
  - Without specifying your devices
  - This is a general purpose kernel
  - And it includes a huge set of device drivers

Question

- Linux is a monolithic kernel
- Therefore, kernel space includes drivers
- When you run a Fedora "live" CD, you boot into Linux
  - Without specifying your devices
  - This is a general purpose kernel
  - And it includes a huge set of device drivers

How do we keep the kernel space small?

Question

- Linux is a monolithic kernel
- Therefore, kernel space includes drivers
- When you run a Fedora "live" CD, you boot into Linux
  - Without specifying your devices
  - This is a general purpose kernel
  - And it includes a huge set of device drivers

How do we keep the kernel space small?

- Linux uses loadable kernel modules

Loadable kernel modules

- Kernel modules are loaded and unloaded on demand
  - While the system is running
- Keeps the kernel space small — only load what you need
- Common for device drivers, filesystem types, network protocols
- Typical steps in Linux to enable a new feature or hardware:

Loadable kernel modules

- Kernel modules are loaded and unloaded on demand
  - While the system is running
- Keeps the kernel space small — only load what you need
- Common for device drivers, filesystem types, network protocols
- Typical steps in Linux to enable a new feature or hardware:
  1. Configure

Loadable kernel modules

- Kernel modules are loaded and unloaded on demand
  - While the system is running
- Keeps the kernel space small — only load what you need
- Common for device drivers, filesystem types, network protocols
- Typical steps in Linux to enable a new feature or hardware:
  1. Configure
  2. Load the appropriate module

Loadable kernel modules

- Kernel modules are loaded and unloaded on demand
  - While the system is running
- Keeps the kernel space small — only load what you need
- Common for device drivers, filesystem types, network protocols
- Typical steps in Linux to enable a new feature or hardware:
  1. Configure
  2. Load the appropriate module
- Allows third–party, closed–source device drivers
  - Not possible without modules according to GPL:
  - Linux kernel is covered by GPL
  - Modifications (e.g., new device drivers) are covered by GPL
  - Third party device driver code is forced to be open source

How do I manage kernel modules?

- Most of the time, the system does it automatically
- For example:
  1. You type "`mount /mnt/floppy`"
  2. The floppy disk is FAT formatted
  3. System will automatically load the kernel module for the FAT filesystem
  4. And if that module depends on others, those are loaded also
- Sometimes you need to do it "by hand"
  - There can be conflicts between modules
  - Or other reasons they are not loaded automatically

Module utilities

/sbin/lsmod

- List the modules that are currently loaded
- Any user can do this
- Output shows
  1. Module name
  2. Size
  3. Number of things using it
  4. Modules that depend on this one

/sbin/modinfo module-name

- Give information about the specified module
  - Including the license
- Any user can do this

Example

```
prompt$ /sbin/lsmod
Module                  Size  Used by
ip6t_REJECT            12826  2
nf_conntrack_ipv6      13892  3
nf_defrag_ipv6         13642  1 nf_conntrack_ipv6
nf_conntrack_ipv4      14182  2
ip6table_filter        12711  1
nf_defrag_ipv4         12601  1 nf_conntrack_ipv4
xt_state               12514  5
nf_conntrack           70557  3 xt_state,nf_conntrack_ipv4,nf_conntrack_ipv6
ip6_tables             17852  1 ip6table_filter
snd_intel8x0           33100  0
i2c_piix4              13406  0
i2c_core               28180  1 i2c_piix4
e1000                 124582  0
snd_ac97_codec        104777  1 snd_intel8x0
ac97_bus               12630  1 snd_ac97_codec
snd_pcm                81137  2 snd_ac97_codec,snd_intel8x0
snd_page_alloc         13709  2 snd_pcm,snd_intel8x0
snd_timer              23742  1 snd_pcm
snd                    62809  4 snd_timer,snd_pcm,snd_ac97_codec,snd_intel8x0
soundcore              14123  1 snd
ppdev                  17363  0
parport_pc             27403  0
parport                39143  2 parport_pc,ppdev
prompt$
```

Loading and unloading modules

`/sbin/insmod module`

- "Inserts" the specified module

`/sbin/rmmod module`

- Removes the specified module

`/sbin/modprobe [switches] module`

- Clever program to insert or remove modules
- Preferred over `/sbin/insmod` and `/sbin/rmmod`

`-r` : remove the module

Installing a new kernel

Can I install a new kernel with `yum`?

Installing a new kernel

Can I install a new kernel with `yum`?

- Yes
- `yum upgrade` will install a new kernel if one is available

Installing a new kernel

Can I install a new kernel with `yum`?

- Yes
- `yum upgrade` will install a new kernel if one is available

Can I install a new kernel with `rpm`?

Installing a new kernel

Can I install a new kernel with `yum`?

- Yes
- `yum upgrade` will install a new kernel if one is available

Can I install a new kernel with `rpm`?

- Yes
- If you can find an `.rpm` for it

Installing a new kernel

Can I install a new kernel with `yum`?

- Yes
- `yum upgrade` will install a new kernel if one is available

Can I install a new kernel with `rpm`?

- Yes
- If you can find an `.rpm` for it

Can I download, compile, and install a new kernel myself?

Installing a new kernel

Can I install a new kernel with `yum`?

- Yes
- `yum upgrade` will install a new kernel if one is available

Can I install a new kernel with `rpm`?

- Yes
- If you can find an `.rpm` for it

Can I download, compile, and install a new kernel myself?

- Yes
- Required for homework

Why build a custom kernel?

0. Because you can

Why build a custom kernel?

0. Because you can
1. You need support for something not normally available
   - E.g., crazy feature like "pretend this machine has 4 CPUs"
   - E.g., Red Hat 6 for file servers
2. You need to turn off support for some feature or device

Why build a custom kernel?

0. Because you can
1. You need support for something not normally available
   - E.g., crazy feature like "pretend this machine has 4 CPUs"
   - E.g., Red Hat 6 for file servers
2. You need to turn off support for some feature or device
3. You want to squeeze more performance out of your system
   - This is becoming less true...
4. You are trying to put Linux on some crazy device
   - So you're also using a cross compiler — good luck with that

Why build a custom kernel?

0. Because you can
1. You need support for something not normally available
   - E.g., crazy feature like "pretend this machine has 4 CPUs"
   - E.g., Red Hat 6 for file servers
2. You need to turn off support for some feature or device
3. You want to squeeze more performance out of your system
   - This is becoming less true...
4. You are trying to put Linux on some crazy device
   - So you're also using a cross compiler — good luck with that
5. You need support for something that requires a newer kernel
   - E.g., "write" support for FAT filesystems needs 2.6.*x*
6. Newer kernels may have bug or security fixes
   - Ok, but (5) and (6) just mean you need a new kernel

Why build a custom kernel?

0. Because you can
1. You need support for something not normally available
   - E.g., crazy feature like "pretend this machine has 4 CPUs"
   - E.g., Red Hat 6 for file servers
2. You need to turn off support for some feature or device
3. You want to squeeze more performance out of your system
   - This is becoming less true...
4. You are trying to put Linux on some crazy device
   - So you're also using a cross compiler — good luck with that
5. You need support for something that requires a newer kernel
   - E.g., "write" support for FAT filesystems needs 2.6.*x*
6. Newer kernels may have bug or security fixes
   - Ok, but (5) and (6) just mean you need a new kernel
7. You want to hack at the source code and see what happens
   - Allowed, but do not do this on a production system

Why not build a custom kernel?

Why not build a custom kernel?

It can be painful

- Good news: it has gotten *much* easier

Why not build a custom kernel?

It can be painful

- Good news: it has gotten *much* easier

Let's see why

Generic steps to build a kernel

Almost identical to the "generic steps to build from source code"

1. Obtain the source code
2. Read documentation
3. Configure
4. Build
5. Install
6. Test
7. Enjoy

1. Obtain the source code

- You can always get a recent kernel from `http://www.kernel.org`
  - Make sure to get a *stable* kernel
  - There are also *development* kernels
  - Current download options appear to be
    1. Compressed tarball
    2. Using `git`
    3. Using `rsync`
- You may be able to install a "source rpm" using `yum` or `rpm`
  - Will not be as new as what you can download yourself

2. Read the documentation

- These are a generic set of instructions
  - To illustrate the basic process
- There are some references to check, for more information
  - Chapter 15 of the book
  - Kwan Lowe's "Kernel Rebuild Guide"
    - A little old, but very nice
    - Ignore everything for 2.4 kernels
  - `http://kernelnewbies.org/FAQ`
- But . . .

2. Read the documentation

- These are a generic set of instructions
  - To illustrate the basic process
- There are some references to check, for more information
  - Chapter 15 of the book
  - Kwan Lowe's "Kernel Rebuild Guide"
    - A little old, but very nice
    - Ignore everything for 2.4 kernels
  - `http://kernelnewbies.org/FAQ`
- But . . .

- Read the documentation for the definitive instructions

3. Configure

- This step is totally different from what we saw before
- For "building packages from source code":
  - You typically ran "`./configure`"
  - This examined your system, automatically
  - End product was a makefile
- For "building a custom kernel":
  - Here is where you specify what kernel features you want
    - More on that, on the next slide ...
  - Turn off critical things: your kernel will be unusable
  - Turn on unneeded things:
    - Longer build time
    - More disk space required
  - This can take a lot of time to get right
  - Result of this is a configuration file

3. Configure

Choices for features

- Typical choices for a feature:

  Yes : The feature is built into the kernel
  No : The feature is turned off
  Module : The feature is built into a kernel module

- Some "features" control groups of features
  - For example, "Prompt for experimental features?"
  - Saying "no" will turn off all of these
    - And removes all those questions
  - Saying "yes" will leave those questions in

3. Configure

"Standard" ways to configure the kernel

`make config`

- A long series of Y/N/M questions
- No easy way to "go back"
- Slow and extremely painful, but always works

`make menuconfig`

- Same questions as before
- Organized into a nice menu system
- You can get help for most questions
- Requires `ncurses`

`make xconfig`

- Like `menuconfig`, but a graphical menu system
- Requires X

3. Configure

Other ways to configure the kernel

`make oldconfig`

- Uses a `.config` file from an old kernel
  - Copy this into the new kernel directory
- Series of Y/N/M questions for any new options not in the old `.config` file

`make localmodconfig`

- Reads current configuration file
- Disables any modules that are not required
  - Connect any devices you want supported
- Series of Y/N/M questions for options that cannot be determined

4. Build

- Same as before — use `make`, as ordinary user
- But let's discuss what is built

4. Build

- Same as before — use `make`, as ordinary user
- But let's discuss what is built

  kernel image : the actual kernel
  - It is not an "executable"
  - It is compressed

4. Build

- Same as before — use `make`, as ordinary user
- But let's discuss what is built

  kernel image : the actual kernel
  - It is not an "executable"
  - It is compressed

  kernel modules : the loadable modules

4. Build

- Same as before — use `make`, as ordinary user
- But let's discuss what is built

  kernel image : the actual kernel
  - It is not an "executable"
  - It is compressed

  kernel modules : the loadable modules

  kernel symbols : file `System.map`
  - Addresses of "global variables" in the kernel
  - Used by kernel modules

4. Build

- Same as before — use `make`, as ordinary user
- But let's discuss what is built

  kernel image : the actual kernel
  - It is not an "executable"
  - It is compressed

  kernel modules : the loadable modules

  kernel symbols : file `System.map`
  - Addresses of "global variables" in the kernel
  - Used by kernel modules

  initial ramdisk : a virtual disk, stored in RAM
  - Why? Next slide

4. Build

- Same as before — use `make`, as ordinary user
- But let's discuss what is built

  kernel image : the actual kernel
  - It is not an "executable"
  - It is compressed

  kernel modules : the loadable modules

  kernel symbols : file `System.map`
  - Addresses of "global variables" in the kernel
  - Used by kernel modules

  initial ramdisk : a virtual disk, stored in RAM
  - Why? Next slide
- Older kernels — some of these are built in separate steps

4. Build

What is a ramdisk for?

What happens after the kernel is loaded?

- Need to load kernel modules
  - To support various devices
  - To support filesystems
  - To support things like LVM, RAID
- Need to `mount` filesystem containing the modules

4. Build

What is a ramdisk for?

What happens after the kernel is loaded?

- Need to load kernel modules
  - To support various devices
  - To support filesystems
  - To support things like LVM, RAID
- Need to `mount` filesystem containing the modules
- But need kernel modules to `mount` the filesystem

4. Build

What is a ramdisk for?

What happens after the kernel is loaded?

- Need to load kernel modules
  - To support various devices
  - To support filesystems
  - To support things like LVM, RAID
- Need to `mount` filesystem containing the modules
- But need kernel modules to `mount` the filesystem
- The ramdisk breaks this cycle:
  1. Kernel mounts root filesystem / from the ramdisk
  2. The ramdisk contains copies of the modules
  3. Modules are loaded
  4. The "real" / is mounted

5. Install

- Copy files to the right places (as `root`)
  - Modules go in a directory under `/lib/modules`
    - Separate directory for each kernel version
  - Everything else goes in `/boot`
- Can have more than one kernel installed at a time
  - Filenames typically include version number
- Configure the bootloader to know about the new kernel

5. Install

- Copy files to the right places (as `root`)
  - Modules go in a directory under `/lib/modules`
    - Separate directory for each kernel version
  - Everything else goes in `/boot`
- Can have more than one kernel installed at a time
  - Filenames typically include version number
- Configure the bootloader to know about the new kernel
- IMPORTANT: do not remove your old kernel yet

5. Install

- Copy files to the right places (as `root`)
  - Modules go in a directory under `/lib/modules`
    - Separate directory for each kernel version
  - Everything else goes in `/boot`
- Can have more than one kernel installed at a time
  - Filenames typically include version number
- Configure the bootloader to know about the new kernel
- IMPORTANT: do not remove your old kernel yet
- `make` will do most, if not all, of this for you
  - E.g., `make modules_install`
  - E.g., `make install`
- Read the documentation

6. Test

- Boot your kernel
- Test your devices (sound, network, etc.)

Ways this can fail (1)

Compile errors

- Not as common as they used to be
- Normally this is a "dependency" problem. E.g:
  - You turn "TCP/IP" off
  - You turn "IP firewall" on
- Try to reconfigure
- Try a different version of the kernel
- Make sure you have a recent C compiler

Ways this can fail (2)

Run–time errors

- Usually happen at boot time
- Might get a kernel panic
- Might get a failed boot
  - E.g., dropped into a `dracut` shell
  - This is the ramdisk infrastructure

Ways this can fail (2)

Run–time errors

- Usually happen at boot time
- Might get a kernel panic
- Might get a failed boot
  - E.g., dropped into a `dracut` shell
  - This is the ramdisk infrastructure
- This is usually because something critical is "turned off"
- Occasionally get bugs in the kernel
  - Some device is not supported correctly (yet . . .)

`dmesg`: display kernel messages

- Messages may give you hints for "what failed"

Ways this can fail (3)

Module errors

- Good news: these are much less common now
- Did you remember to install the modules?
- "Can't find symbols" errors
  - Mismatch between `System.map` and what modules expect
  - Did you remember to install the new `System.map`?

So, why is this painful?

You can build a kernel by trial and error. But:

- There are lots of configure choices
  - Some cryptic things are absolutely critical
- When the kernel fails to boot, it is not always clear why

So, why is this painful?

You can build a kernel by trial and error. But:

- There are lots of configure choices
  - Some cryptic things are absolutely critical
- When the kernel fails to boot, it is not always clear why
- Compile times are in hours

`dmesg` : Display kernel messages

`/sbin/insmod` : Insert a module, by hand

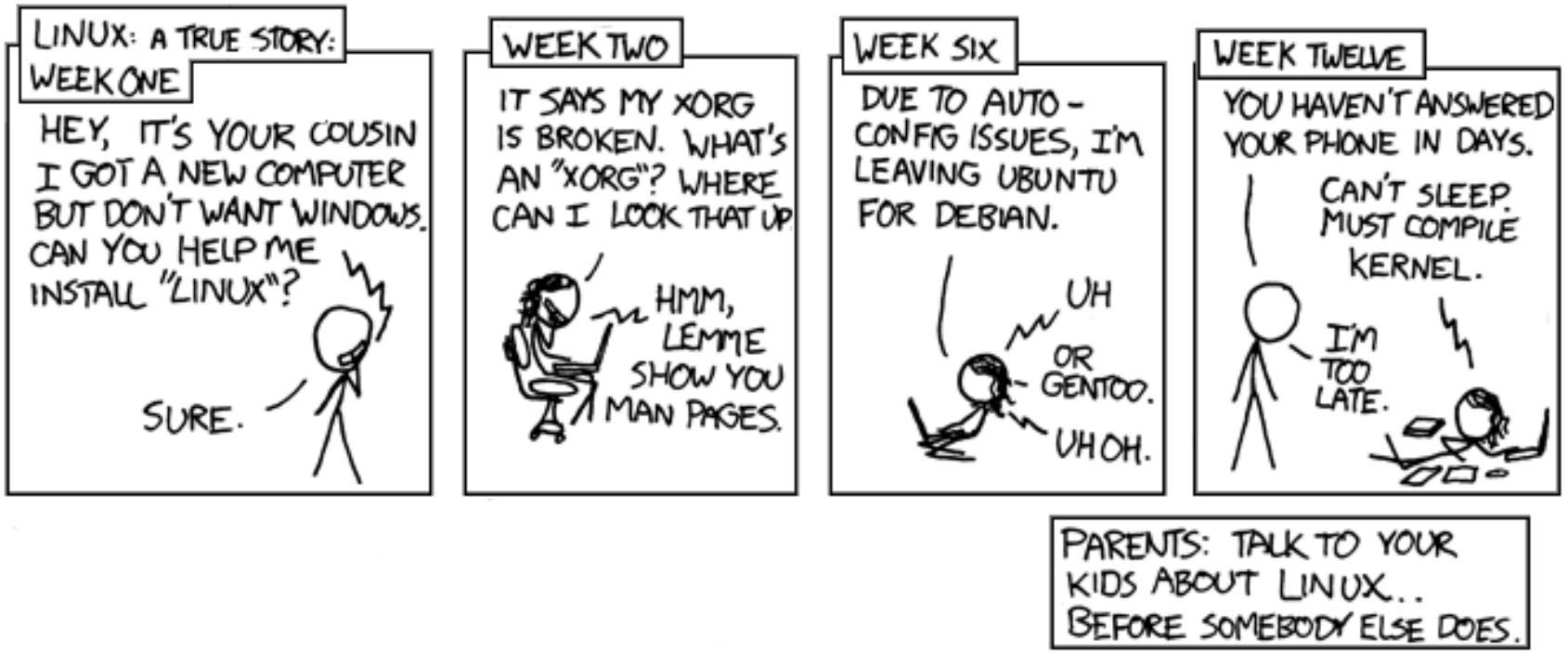
`/sbin/lsmod` : List currently–loaded modules

`/sbin/modinfo` : Display information about a module

`/sbin/modprobe` : Insert or remove modules

`/sbin/rmmod` : Remove a module, by hand

An appropriate xkcd comic: http://xkcd.com/456



End of lecture