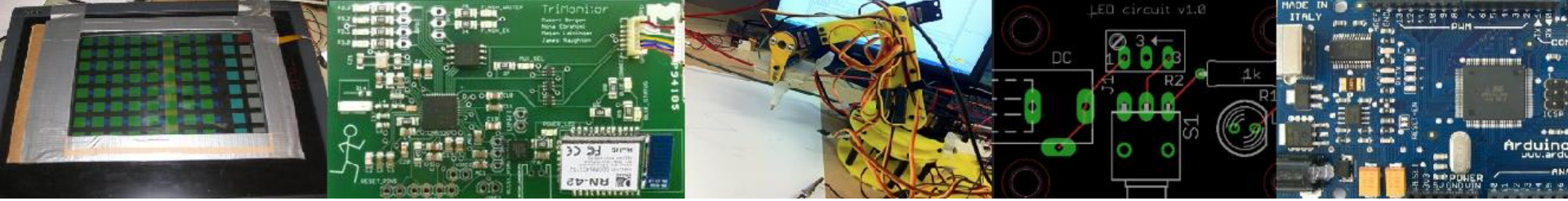
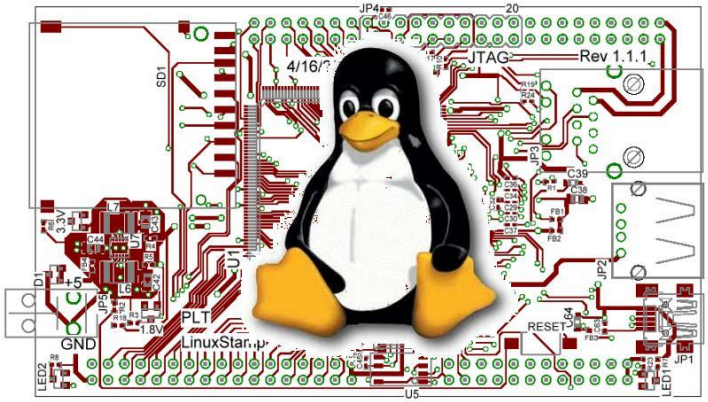


Embedded Linux
Lectures 6 and 7



MICHIGAN

Two main tasks

- Introduce Linux and in particular embedded Linux.
  - Pretty high-level.
  - Lots of material coming pretty quickly.

- How to write device drivers for Linux.
  - Much more detailed
  - Pretty complex coding
  - Will do in lab 4.

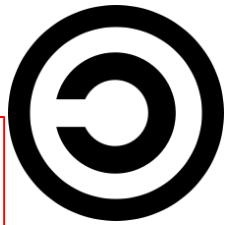
Outline

- Background
  - Legal issues, versioning, building, user basis, FHS, booting
- Embedded Linux (common but **not** required attributes)
  - Small footprint (BusyBox)
  - Flash file system
  - Real time support

Warning: We are going to do this section very quickly indeed. There is a lot of stuff I want you to see. There is enough stuff in this *section* of this lecture for 20 hours of discussion. We don't have that kind of time. Just going to lightly touch on some stuff.

Linux?

- A POSIX-compliant and widely deployed desktop/server operating system licensed under the GPL
  - POSIX
    - Unix-like environment (shell, standard programs like awk etc.)
  - Desktop OS
    - Designed for users and servers
    - Not designed for embedded systems
  - GPL
    - Gnu Public License. May mean you need to make source code available to others.
      - First “copyleft” license.
    - Linux is licensed under GPL-2, not GPL-3.



Many figures and text in this section taken from *Embedded Linux Primer*, second edition
We (kind of) have on-line access to the book.
Also http://www.freesoftwaremagazine.com/articles/drivers_linux is used a lot!

What is a kernel? (1/2)

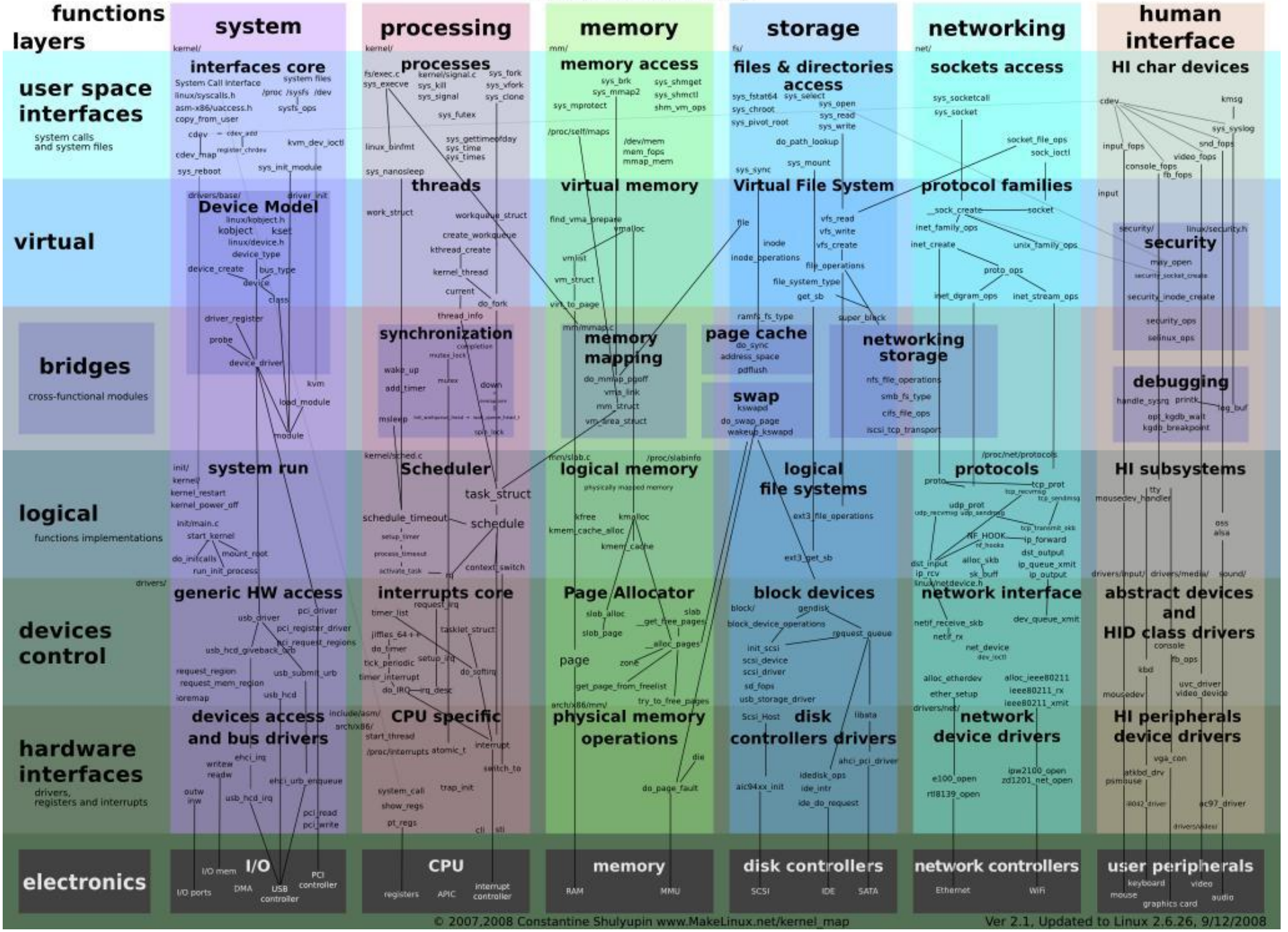
- The kernel's job is to talk to the hardware and software, and to manage the system's resources as best as possible.
  - It talks to the hardware via the drivers that are included in the kernel (or additionally installed later on in the form of a kernel module).
    - This way, when an application wants to do something (say change the volume setting of the speakers), it can just submit that request to the kernel, and the kernel can use the driver it has for the speakers to actually change the volume.

This part taken from http://www.makeuseof.com/tag/linux-kernel-explanation-laymans-terms/

What is a kernel? (2/2)

- The kernel is highly involved in resource management.
  - It has to make sure that there is enough memory available for an application to run, as well as to place an application in the right location in memory.
  - It tries to optimize the usage of the processor so that it can complete tasks as quickly as possible. It also aims to avoid deadlocks, which are problems that completely halt the system when one application needs a resource that another application is using.
  - It's a fairly complicated circus act to coordinate all of those things, but it needs to be done and that's what the kernel is for.

Linux kernel map



© 2007,2008 Constantine Shulyupin www.MakeLinux.net/kernel_map

Ver 2.1, Updated to Linux 2.6.26, 9/12/2008

Linux Kernel history

- The major numbers really aren't major.
  - That is the change from 3.x to 4.0 isn't generally much bigger than from 4.0 to 4.2
- 2.6 to 3.0 upgrade was move t for Linux's 20th anniversary.
  - **"So what are the big changes?** NOTHING. Absolutely nothing. Sure, we have the usual two thirds driver changes, and a lot of random fixes, but the point is that 3.0 is *just* about renumbering…" [https://lkml.org/lkml/2011/5/29/204]
  - And 4.0 is more of the same.
    - http://www.itworld.com/article/2887558/linus-torvalds-bumps-linux-kernel-to-version-4x.html
  - 5.0 had a bit more.

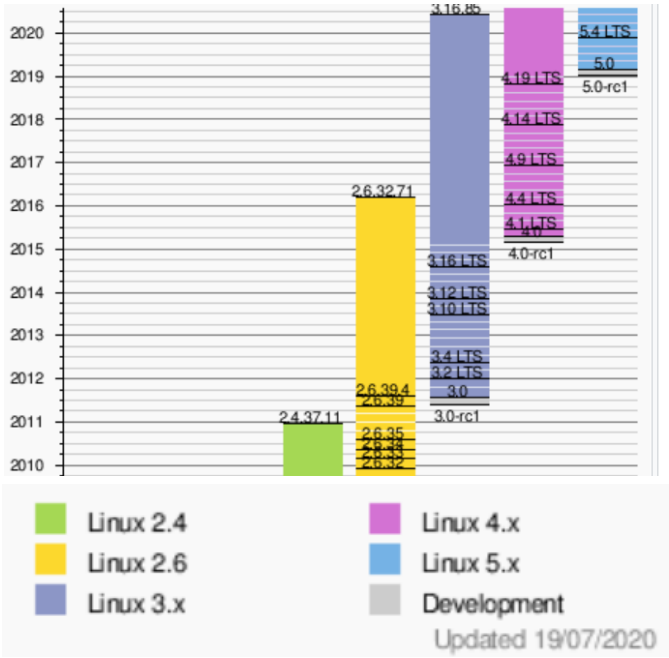


Figure modified from "Linux kernel version history" article on Wikipedia

What version am I working with?

- If running, use “uname” command
  - “uname -a” for all information
- If looking at source
  - First few lines of the top-level Makefile will tell you.

How do I download and build the kernel?

- Use git.
- Typing “make” with no target at the top-level should build the kernel.
  - Need gcc installed (no other compiler will do).
  - Should generate an ELF file called “vmlinux”
    - But lots of configuration stuff

Kernel configuration

- There is a file, “.config” which drives the build.
  - It determines the functionality (including cross-compling?) of the kernel.
    - Things like USB support, etc.
  - It is setup in any number of ways.
    - The default is to ask a huge number of questions.
    - There are editors and defaults you can use instead.
      - **`make defconfig`** should just do all defaults for example.
    - **`make help`** should give a solid overview of options
- The .config file scheme has some problems.
  - It is easy to miss, as files that start with a “.” (dot) are hidden files in Linux ("**`ls -a`**" will show them)
  - It is easy to blow away.
    - **`make distclean`** will delete it for example…

Linux user basics--shell

- You have a *shell* which handles user commands
  - May just search for an executable file (application) in certain locations
  - Allows for moving data between those applications.
    - Pipes etc.
  - Is itself a programming language.
  - There are many of these (bash, sh, tcsh, csh, ksh, zsh)
    - Most have very similar interfaces (type application name, it runs), but the programming language part varies quite a bit.
    - Geek humor:
      - **sh** is called the Bourne shell, written by Stephen Bourne
      - **bash**, often treated as an upgrade to **sh**, is the "Bourne again shell"

Linux user basics—file systems

- Linux supports a huge variety of file systems.
  - But they have some commonalities.
- Pretty much a standard directory structure with each directory holding either other directories or files.
  - Each file and directory has a set of permissions.
    - One owner (a single user)
    - One group (a list of users who may have special access)
    - There are three permissions, read, write and execute
      - Specified for owner, group, and world.
- There are also links (hard and soft)
  - So rather than copying files I can point to them.

FHS: File System Hierarchy Standard

- There is a standard for laying out file systems
  - Part of this is the standard top-level directories

TABLE 6-1 Top-Level Directories

Directory	Contents
`bin`	Binary executables, usable by all users on the system[1]
`dev`	Device nodes (see Chapter 8, "Device Driver Basics")
`etc`	Local system configuration files
`home`	User account files
`lib`	System libraries, such as the standard C library and many others
`sbin`	Binary executables usually reserved for superuser accounts on the system
`tmp`	Temporary files
`usr`	A secondary file system hierarchy for application programs, usually read-only
`var`	Contains variable files, such as system logs and temporary configuration files

A “minimal” file system

LISTING 6-1 Contents of a Minimal Root File S

```
.
|-- bin
|   |-- busybox
|   `-- sh -> busybox
|-- dev
|   `-- console
|-- etc
|   `-- init.d
|       `-- rcS
`-- lib
    |-- ld-2.3.2.so
    |-- ld-linux.so.2 -> ld-2.3.2.so
    |-- libc-2.3.2.so
    `-- libc.so.6 -> libc-2.3.2.so

5 directories, 8 files
```

- Busybox is covered later

Background: booting

1. Some circuit magic happens
   - Get clock running, reset registers etc.
2. Bootloader starts
   - Initialize devices such as I2C, serial, DRAM, cache, etc.
   - Starts the OS
3. Kernel starts
   - Might set up other things needed
4. Init gets called
   - Lots of stuff…

Background: booting

LISTING 2-1 Initial Bootloader Serial Output

```
U-Boot 2009.01 (May 20 2009 - 09:45:35)

CPU:   8548E, Version: 2.1, (0x80390021)
Core:  E500, Version: 2.2, (0x80210022)
Clock Configuration:
       CPU:990  MHz, CCB:396  MHz,
       DDR:198  MHz (396 MT/s data rate), LBC:49.500 MHz
L1:    D-cache 32 kB enabled
       I-cache 32 kB enabled
Board: CDS Version 0x13, PCI Slot 1
CPU Board Revision 0.0 (0x0000)
I2C:   ready
DRAM:  Initializing
    SDRAM: 64 MB
    DDR: 256 MB
FLASH: 16 MB
L2:    512 KB enabled
Invalid ID (ff ff ff ff)
    PCI: 64 bit, unknown MHz, async, host, external-arbiter
                Scanning PCI bus 00
PCI on bus 00 - 02

    PCIE connected to slot as Root Complex (base address e000a000)
PCIE on bus 3 - 3
In:    serial
Out:   serial
Err:   serial
Net:   eTSEC0, eTSEC1, eTSEC2, eTSEC3
=>
```

Handoff from the Kernel

```
...
  if (execute_command) {
          run_init_process(execute_command);
          printk(KERN_WARNING "Failed to execute %s.  Attempting "
                              "defaults...\n", execute_command);
  }

  run_init_process("/sbin/init");
  run_init_process("/etc/init");
  run_init_process("/bin/init");
  run_init_process("/bin/sh");

  panic("No init found.  Try passing init= option to kernel.");
```

- Kernel tries to start init.
  - Tries a few locations
    - Doesn't return if successful, so first that succeeds is all that runs.

The init process

- Init is the ultimate parent of all user-space processes.
  - Provides all default environment variables including PATH
- Init uses a text file, **`/etc/inittab`**.
  - That file includes a number of “run levels” that are used to determine what init should do.
    - Halt, reboot, general-purpose multi-user, etc.

Init (continued)

TABLE 6-2 Runlevels

Runlevel	Purpose
0	System shutdown (halt)
1	Single-user system configuration for maintenance
2	User-defined
3	General-purpose multiuser configuration
4	User-defined
5	Multiuser with graphical user interface on startup
6	System restart (reboot)

LISTING 6-4 Runlevel Directory Structure

```
$ ls -l /etc/rc.d
total 96
drwxr-xr-x  2 root root  4096 Oct 20 10:19 init.d
-rwxr-xr-x  1 root root  2352 Mar 16  2009 rc
drwxr-xr-x  2 root root  4096 Mar 22  2009 rc0.d
drwxr-xr-x  2 root root  4096 Mar 22  2009 rc1.d
drwxr-xr-x  2 root root  4096 Mar 22  2009 rc2.d
drwxr-xr-x  2 root root  4096 Mar 22  2009 rc3.d
drwxr-xr-x  2 root root  4096 Mar 22  2009 rc4.d
drwxr-xr-x  2 root root  4096 Mar 22  2009 rc5.d
drwxr-xr-x  2 root root  4096 Mar 22  2009 rc6.d
-rwxr-xr-x  1 root root   943 Dec 31 16:36 rc.local
-rwxr-xr-x  1 root root 25509 Jan 11  2009 rc.sysinit
```

LISTING 6-5 Sample Runlevel Directory

```
lrwxrwxrwx  1 root root 17 Nov 25  2009 S10network -> ../init.d/network
lrwxrwxrwx  1 root root 16 Nov 25  2009 S12syslog  -> ../init.d/syslog
lrwxrwxrwx  1 root root 16 Nov 25  2009 S56xinetd  -> ../init.d/xinetd
lrwxrwxrwx  1 root root 16 Nov 25  2009 K50xinetd  -> ../init.d/xinetd
lrwxrwxrwx  1 root root 16 Nov 25  2009 K88syslog  -> ../init.d/syslog
lrwxrwxrwx  1 root root 17 Nov 25  2009 K90network -> ../init.d/network
```

rcN.d holds symlinks to start or kill each process associated with that runlevel.

init.d holds the scripts for each service.

Outline

- Background
  - Legal issues, versioning, building, user basis, FHS, booting
- Embedded Linux (common but **not** required attributes)
  - Small footprint (BusyBox)
  - Flash file system
  - Real time support

What makes a Linux install “embedded”?

- It’s one of those poorly defined terms, but in general it will have one or more of the following
  - small footprint
  - flash files system
  - real-time extensions of some sort

Small footprint--Busybox

addgroup, adduser, ar, ash, awk, basename, blkid, bunzip2, bzcat, cat, chattr, chgrp, chmod, chown, chpasswd, chroot, chvt, clear, cmp, cp, cpio, cryptpw, cut, date, dc, dd, deallocvt, delgroup, deluser, df, dhcprelay, diff, dirname, dmesg, dos2unix, dumpkmap, dumpleases, echo, egrep, env, expr, false, fbset, fbsplash, fdisk, fgrep, find, free, freeramdisk, fsck, fsck.minix, fuser, getopt, getty, grep, gunzip, gzip, halt, head, hexdump, hostname, httpd, hwclock, id, ifconfig, ifdown, ifup, init, insmod, ip, kill, killall, klogd, last, less, linuxrc, ln, loadfont, loadkmap, logger, login, logname, logread, osetup, ls, lsmod, makedevs, md5sum, mdev, microcom, mkdir, mkfifo, mkfs.minix, mknod, mkswap, mktemp, modprobe, more, mount, mv, nc, netstat, nice, nohup, nslookup, od, penvt, passwd, patch, pidof, ping, ping6, pivot_root, poweroff, printf, ps, pwd, rdate, rdev, readahead, readlink, readprofile, realpath, reboot, renice, reset, rm, rmdir, rmmod, route, rtcwake, run-parts, sed, sed, setconsole, etfont,sh, showkey, sleep, sort, start-stop-daemon, strings, stty, su, sulogin, swapoff, swapon, switch_root, sync, sysctl, syslogd, tail, tar, tee, telnet, telnetd, test, tftp, time, top, touch, tr, traceroute, true, tty, udhcpc, udhcpd, umount, uname, uniq, unzip, uptime, usleep, vi, vlock, watch, wc, wget, which, who, whoami, xargs, yes, zcat

- A single executable that implements the functionality of a massive number of standard Linux utilities
  - `ls, gzip, ln, vi`. Pretty much everything you normally need.
  - Some have limited features
    - Gzip only does the basics for example.
  - Pick which utilities you want it to do
    - Can either drop support altogether or install real version if needed
  - Highly configurable (similar to Linux itself), easy to cross-compile.
- Example given is around 2MB statically compiled!

Using busybox

- Two ways to play
  - Invoke from the command line as busybox
    - **`busybox ls /`**
  - Or create a softlink to busybox and it will run as the name of that link.
    - So if you have a softlink to busybox named “ls” it will run as ls.

Links done for you

- Normally speaking, you'll use the softlink option.
  - You can get it to put in all the links for you with "make install"
    - Be darn careful you don't overwrite things locally if you are doing this on the host machine.
      - That would be bad.

```
|-- bin
|    |-- busybox
|    |-- cat -> busybox
|    |-- dmesg -> busybox
|    |-- echo -> busybox
|    |-- hostname -> busybox
|    |-- ls -> busybox
|    |-- ps -> busybox
|    |-- pwd -> busybox
```

```
$ make CONFIG_PREFIX=/mnt/remote install
  /mnt/remote/bin/ash -> busybox
  /mnt/remote/bin/cat -> busybox
  /mnt/remote/bin/chgrp -> busybox
  /mnt/remote/bin/chmod -> busybox
  /mnt/remote/bin/chown -> busybox
...
  /mnt/remote/usr/bin/xargs -> ../../bin/busybox
  /mnt/remote/usr/bin/yes -> ../../bin/busybox
  /mnt/remote/usr/sbin/chroot -> ../../bin/busybox
```

System initialization

- Busybot can also be “init”
  - But it’s a simpler/different version than the init material covered above.
  - More “bash” like

```
#!/bin/sh
echo "Mounting proc"
mount -t proc /proc /proc

echo "Starting system loggers"
syslogd
klogd

echo "Configuring loopback
   interface"
ifconfig lo 127.0.0.1

echo "Starting inetd"
xinetd

# start a shell
busybox sh
```

BusyBox Summary

- BusyBox is a powerful tool for embedded systems that replaces many common Linux utilities in a single multicall binary.
- BusyBox can significantly reduce the size of your root file system image.
- Configuring BusyBox is straightforward, using an interface similar to that used for Linux configuration.
- System initialization is possible but somewhat different with BusyBox

Outline

- Background
  - Legal issues, versioning, building, user basis, FHS, booting
- Embedded Linux (common but **not** required attributes)
  - Small footprint (BusyBox)
  - Flash file system
  - Real time support

Flash storage devices

- Significant restrictions on writing
  - data can be changed from a 1 to a 0 with writes to the cell's address
  - 0 to 1 requires an entire block be erased.
- Therefore, to modify data stored in a Flash memory, the block in which the modified data resides must be completely erased.
  - Write times for updating data in Flash memory can be many times that of a hard drive.
- Also very limited write cycles (100 to 1,000,000 or so) before wear out.
  - Wear leveling, conservative specifications generally make things okay.

Outline

- Background
  - Legal issues, versioning, building, user basis, FHS, booting
- Embedded Linux (common but **not** required attributes)
  - Small footprint (BusyBox)
  - Flash file system
  - Real time support
    - RT Linux patch
    - Other solutions

Real-time Kernel Patch

- The patch had many contributors, and it is currently maintained by Ingo Molnar; you can find it at:
  - **www.kernel.org/pub/linux/kernel/projects/rt/**
  - **http://tinyurl.com/rtlinux473 (linux.com)**
- Since about Linux 2.6.12, soft real-time performance in the single-digit milliseconds on a reasonably fast x86 processor is readily achieved
  - Some claim “nearly-worst-case” latency of 30us!

Features

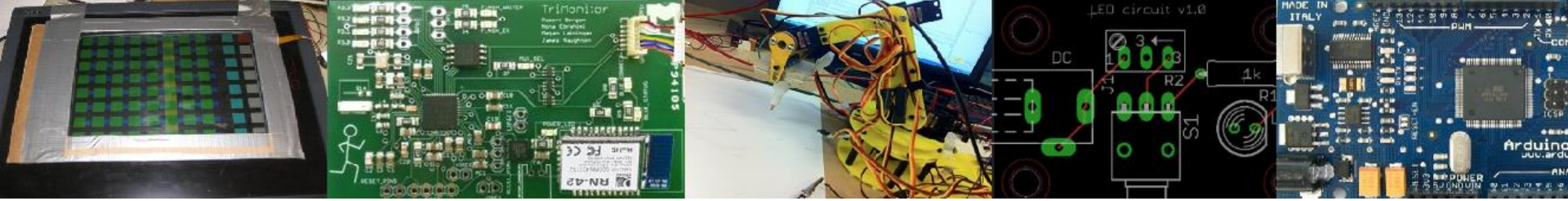
- The real-time patch adds a fourth preemption mode called PREEMPT_RT, or Preempt Real Time.
  - Features from the real-time patch are added, including replacing spinlocks with preemptable mutexes.
    - This enables involuntary preemption everywhere within the kernel except for areas protected by preempt_disable().
    - This mode significantly smoothes out the variation in latency (jitter) and allows a low and predictable latency for time-critical real-time applications.
  - The problem is…
    - Not all devices can handle being interrupted.
    - This means that with the RT patch in play you might get random crashes.

There are lots of other attempts and discussions about a real-time Linux

- RTLinux
  - Wind River had something up and running for years.
    - Ended support in 2011.
    - Seems fairly restrictive.
- Real Time Linux Foundation, Inc.
  - Holding workshops on this for 13 years.
- http://lwn.net/Articles/397422/
  - Nice overview of some of the issues
- Zephyr is an RTOS that is being developed as part of the Linux Foundation.
  - Looks like a traditional RTOS.
    - As small as 8KB memory

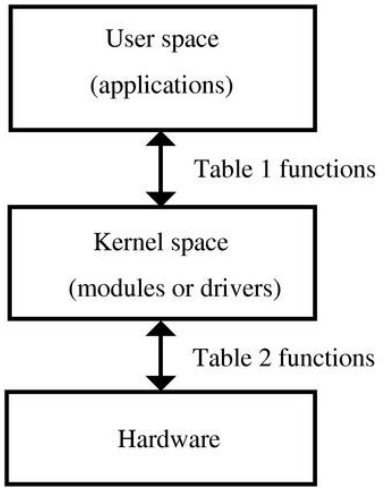
Windows 10 IoT

- A stripped down version of Windows 10 with a focus on IoT issues.
  - Appears to be fairly popular.
  - 256MB memory is the minimum
    - Sometimes 512MB (if you have a display)
  - Runs on a Pi3.
- Multiple versions all supported by Visual Studio.
  - Reasonable choice for things that are plugged in.
  - Remember, the best engineering solution isn't always the best solution
    - What I mean is that the "best" solutions require a lot of engineering time and therefore $$$$.
    - Might be best to use this (programmers familiar with the environment, lots of stuff done for you) to reduce total cost even if the parts are more expensive.



EECS 473
Advanced Embedded Systems

Linux device drivers and loadable kernel modules



MICHIGAN

Linux Device Drivers

- Overview
  - What is a device driver?
    - Linux devices
  - User space vs. Kernel space
- Modules and talking to the kernel
  - Background
  - Example
  - Some thinky stuff

A fair bit of this presentation, including some figures, comes from
http://www.freesoftwaremagazine.com/articles/drivers_linux#
Other sources noted at the end of the presentation.

Linux Device Drivers

- Overview
  - What is a device driver?
    - Linux devices
  - User space vs. Kernel space
- Modules and talking to the kernel
  - Background
  - Example
  - Some thinky stuff

Device driver

(Thanks Wikipedia!)

- A device driver is a computer program allowing higher-level computer programs to interact with a hardware device.
  - A driver typically communicates with the device through the computer bus or communications subsystem to which the hardware connects.
  - When a calling program invokes a routine in the driver, the driver issues commands to the device.
  - Drivers are hardware-dependent and operating-system-specific.

Devices in Linux (1/2)

- There are special files called “device files” in Linux.
  - A user can interact with it much like a normal file.
  - But they *generally* provide access to a physical device.
  - They are generally found in /dev and /sys
    - /dev/fb is the frame buffer
    - /dev/ttyS0 is one of the serial ports
- Not all devices files correspond to physical devices.
  - Pseudo-devices.
    - Provide various functions to the programmer
    - /dev/null
      - Accepts and discards all input; produces no output.
    - /dev/zero
      - Produces a continuous stream of NULL (zero value) bytes.
    - etc.

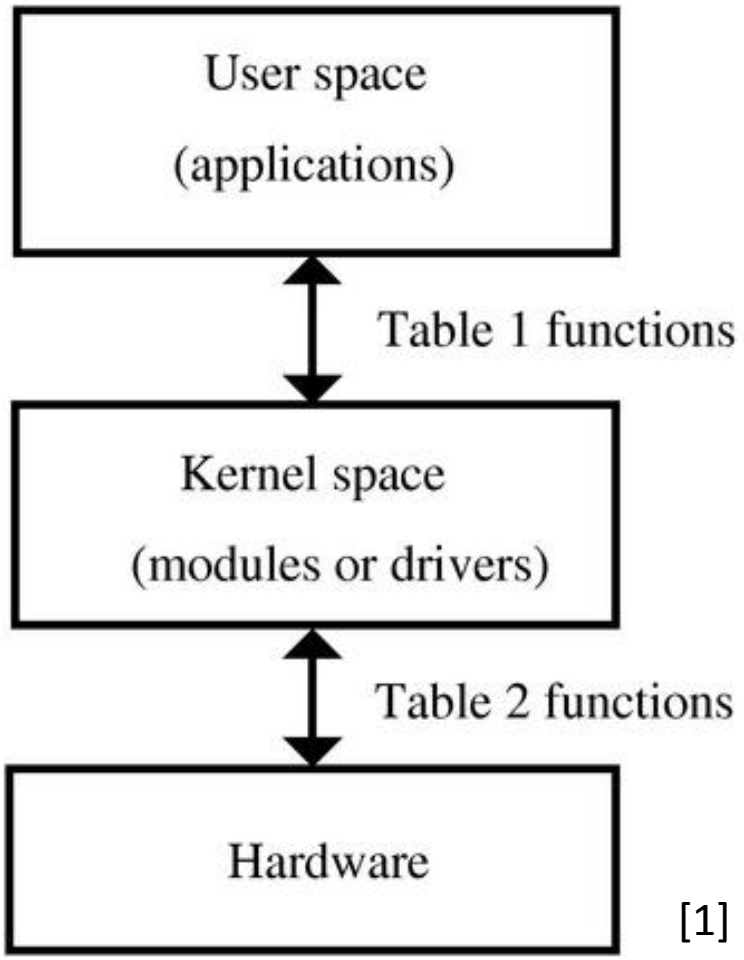
```
crw-rw---- 1 root dialout   4,  64 Jun 20 13:01 ttyS0
```

Devices in Linux (2/2)

- Pretty clearly you need a way to connect the device file to the actual device
  - Or pseudo device for that matter
- We want to be able to “fake” this by writing functions that handle the file I/O.
  - So we need to associate functions with all the things we can do with a file.
    - Open, close.
    - Read, write.
- Today we’ll talk about all that…

Kernel vs. User space

- User Space
  - End-user programs. They use the kernel to interface to the hardware.
- Kernel Space
  - Provides a standard (and hopefully multi-user secure) method of using and sharing the hardware.
    - Private function member might be a good analogy.
  - A lot of things are different here.
    - Many calls to the kernel can't be made from the kernel.
      - E.g. malloc.



[1]

Linux Device Drivers…

- Overview
  - What is a device driver?
    - Linux devices
  - User space vs. Kernel space
- Modules and talking to the kernel
  - Background
  - Example
  - Some thinky stuff

Kernel and Kernel Modules

- Often, if you want to add something to the kernel you need to rebuild the kernel and reboot.
  - A “loadable kernel module” (LKM) is an object file that extends the base kernel.
  - Exist in most OSes
    - Including Windows, FreeBSD, Mac OS X, etc.
  - Modules get added and removed as needed
    - To save memory, add functionality, etc.

[2]

Linux Kernel Modules

- In general must be licensed under a free license.
  - Doing otherwise will taint the whole kernel.
    - A tainted kernel sees little support.
    - Might be a copyright problem if you redistribute.
- The Linux kernel changes pretty rapidly, including APIs etc.
  - This can make it a real chore to keep LKMs up to date.
  - Also makes a tutorial a bit of a pain.
    - Though honestly it seems fairly stable over the last 5 years.

Creating a module

- All modules need to define functions that are to be run when:
  - The module is loaded into the kernel
  - The module is removed from the kernel
- We just write C code (see next slide)
- We need to compile it as a kernel module.
  - We invoke the kernel's makefile.
  - `sudo make -C /lib/modules/xxx/build M=$PWD modules`
    - This makes (as root) using the makefile in the path specified.
    - I think it makes all C files in the directory you started in*
    - Creates .ko (rather than .o) file
    - Xxx is some kernel version/directory

Simple module

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void) {
    printk("<1> Hello World!\n");
    return 0;
}

static void hello_exit(void) {
    printk("<1> Bye world!\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

- MODULE_LICENSE
  - Required.
  - Short list of allowed licenses.
- Printk()
  - Kernel print.
    - Prints message to console and to log.
    - <1> indicates high priority message, so it gets logged.
- Module_init()
  - Tells system what module to call when we first load the module.
- Module_exit()
  - Same but called when module released.

Modules: Listing, loading and removing

- From the command line:
  - lsmod
    - List modules.
  - insmod
    - Insert module into kernel
      - Adds to list of available modules
    - Causes function specified by module_init() to be called.
  - rmmod
    - Removes module from kernel

lsmod

```
Module                  Size  Used by
memory                 10888  0
hello                   9600  0
binfmt_misc            18572  1
bridge                 63776  0
stp                    11140  1 bridge
bnep                   22912  2
video                  29844  0
```

insmod

- Very (very) simple
  - **`insmod xxxxx.ko`**
    - Says to insert the module into the kernel

Other (better) way to load a module

- **`Modprobe`** is a smarter version of insmod.
  - Actually it's a smarter version of insmod, lsmod and rmmod…
    - It can use short names/aliases for modules
    - It will first install any dependent modules
- We'll use insmod for the most part
  - But be aware of modprobe

So?

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void) {
    printk("<1> Hello World!\n");
    return 0;
}

static void hello_exit(void) {
    printk("<1> Bye world!\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

- When insmod, log file gets a “Hello World!”
- When rmmod, that message prints to log (and console…)
- It’s not the name, it’s the module_init().

Modules?

- There are a number of different reasons one might have a module
  - But the main one is to create a device driver
  - It's not realistic for Linux to have a device driver for all possible hardware in memory all at once.
    - Would be too much code, requiring too much memory.
  - So we have devices as modules
    - Loaded as needed.

What is a “device”?

- As mentioned in the overview, Linux devices are accessed from user space in exactly the same way files are accessed.
  - They are generally found in /dev and /sys
- To link normal files with a kernel module, each device has a “major number”
  - Each device also has a “minor number” which can be used by the device to distinguish what job it is doing.

```
% ls -l /dev/fd0 /dev/fd0u1680
brwxrwxrwx   1 root  floppy   2,  0 Jul  5  2000 /dev/fd0
brw-rw----   1 root  floppy   2, 44 Jul  5  2000 /dev/fd0u1680
```

Two floppy devices. They are actually both the same bit of hardware using the same driver (major number is 2), but one is 1.68MB the other 1.44.

Creating a device

- **`mknod /dev/memory c 60 0`**
  - Creates a device named /dev/memory
  - Major number 60
  - Minor number 0
- Minor numbers are passed to the driver to distinguish different hardware with the same driver.
  - Or, potentially, the same hardware with different parameters (as the floppy example)

Linux Device Drivers

- Overview
  - What is a device driver?
    - Linux devices
  - User space vs. Kernel space
- Modules and talking to the kernel
  - Background
  - Example
  - Some thinky stuff

A somewhat real device

- We are going to create a device that is just a single byte of memory.
  - Whatever the last thing you wrote to it, is what will be read.
- For example
  - $ echo -n abcdef >/dev/memory
  - Followed by $ cat /dev/memory
    - Prints an “f”.
- Silly, but not unreasonable.
  - It’s also printing some stuff to the log.
    - Not a great idea in a real device, but handy here.

Almost entirely from http://www.freesoftwaremagazine.com/articles/drivers_linux#

includes

```
/* Necessary includes for device drivers */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>   /* printk() */
#include <linux/slab.h>     /* kmalloc() */
#include <linux/fs.h>       /* everything... */
#include <linux/errno.h>    /* error codes */
#include <linux/types.h>    /* size_t */
#include <linux/proc_fs.h>
#include <linux/fcntl.h>    /* O_ACCMODE */
#include <asm/system.h>     /* cli(), *_flags */
#include <asm/uaccess.h>    /* copy_from/to_user */
```

License and function prototypes

```
MODULE_LICENSE("Dual BSD/GPL");

int memory_open (struct inode *inode, struct
file *filp);

int memory_release (struct inode *inode, struct
file *filp);

ssize_t memory_read (struct file *filp, char
*buf, size_t count , loff_t *f_pos);

ssize_t memory_write (struct file *filp, char
*buf, size_t count , loff_t *f_pos);

void memory_exit (void);
int memory_init (void);
```

Setting up the standard interface

```
struct file_operations
memory_fops = {
        read: memory_read,
        write: memory_write,
        open:memory_open,
        release: memory_release
};
struct file_operations fops = {
        .read = memory_read,
        .write = memory_write,
        .open = memory_open,
        .release = memory_release
};
```

- This is a weird bit of C syntax.
  - Initializes struct elements.
    - So “read” member is now “memory_read”
  - Technically unsupported these days?
    - gcc supports it though

file_operations struct

```
struct file_operations {
  ssize_t(*read) (struct file *, char __user *, size_t, loff_t *);
  ssize_t(*write) (struct file *, const char __user *, size_t, loff_t *);
  int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
  int (*open) (struct inode *, struct file *);
  int (*release) (struct inode *, struct file *);
  int (*fsync) (struct file *, struct dentry *, int datasync);
  int (*aio_fsync) (struct kiocb *, int datasync);
  int (*fasync) (int, struct file *, int);
  int (*lock) (struct file *, int, struct file_lock *);
  ssize_t(*readv) (struct file *, const struct iovec *, unsigned long,
                   loff_t *);
 ssize_t(*writev) (struct file *, const struct iovec *, unsigned long,
                    loff_t *);
 ssize_t(*sendfile) (struct file *, loff_t *, size_t, read_actor_t,
                     void __user *);
 ssize_t(*sendpage) (struct file *, struct page *, int, size_t,
                     loff_t *, int);
  unsigned long (*get_unmapped_area) (struct file *, unsigned long,
  unsigned long,unsigned long,unsigned long);
};
```

file_operations: A few members

```
struct file_operations {
  ssize_t(*read) (struct file *, char __user *,
                          size_t, loff_t *);
  ssize_t(*write) (struct file *, const char __user *,
                          size_t, loff_t *);
  int (*ioctl) (struct inode *, struct file *,
                          unsigned int, unsigned long);
  int (*open) (struct inode *, struct file *);
  int (*release) (struct inode *, struct file *);
};
```

Set up init and exit
Some globals

```
module_init(memory_init);
module_exit(memory_exit);

int memory_major = 60;
char *memory_buffer;
```

memory_init

60, via global

Device name, need not be the same as in /dev

```
int memory_init(void) {
int result;
result = register_chrdev(memory_major, "memory", &memory_fops);
if (result < 0) {
    printk("<1>memory: cannot obtain major number %d\n",
            memory_major);
    return result;
}

/* Allocating memory for the buffer */
memory_buffer = kmalloc (1, GFP_KERNEL);
if (!memory_buffer) {
    result = -ENOMEM;
    goto fail;
}

memset(memory_buffer, 0, 1);  // initialize 1 byte with 0s.
printk("<1> Inserting memory module\n");
return 0;

fail:
memory_exit();
return result;
}
```

Name of file_operations structure.

Kmalloc does what you'd expect. The flag provides rules about where and how To get the memory. See makelinux.com/ldd3/chp-8-sect-1

memory_exit

```
void memory_exit(void) {

unregister_chrdev(memory_major, "memory");
if (memory_buffer) {
    kfree(memory_buffer);
}
```

Open and release (close)

```
int memory_open (struct inode *inode,
          struct file *filp) {
printk("<1> Minor: %d\n",
          MINOR(inode->i_rdev));
return 0;
}

int memory_release (struct inode *inode,
          struct file *filp) {
return 0;
}
```

Modules: single character memory example

```
ssize_t memory_read(struct file *filp, char *buf,
                    size_t count, loff_t *f_pos) {

        /* Transfering data to user space */
        copy_to_user (buf,memory_buffer,1);

        /* Changing reading position as best suits */
        if (*f_pos == 0) {
            *f_pos+=1;
            return 1;
        } else {
            return 0;
        }
}
```

f_pos is the file position.
What do you think happens
if you don't change *f_pos?

copy_to_user copies to a location in userspace (the first argument) from kernel space (the second argument), a specific number of bytes. Recall virtual memory…

memory_write

```
ssize_t memory_write( struct file *filp,
      char *buf, size_t count, loff_t *f_pos)
{
      char *tmp;

      tmp=buf+count-1;
      copy_from_user(memory_buffer,tmp,1);
      return 1;
}
```

How do you set it up?

- Make the module

```
make -C /lib/modules/2.6.28-16-generic/build M=$PWD modules
```

- Insert the module

```
insmod memory.ko
```

- Create the device

```
mknod /dev/memory c 60 0
```

- Make the device read/write

```
chmod 666 /dev/memory
```

What did all that do?

- We are going to create a device that is just a single byte of memory.
  - Whatever the last thing you wrote to it, is what will be read.
- For example
  - $ echo -n abcdef >/dev/memory
  - Followed by $ cat /dev/memory
    - Prints an “f”.

See also

- https://www.apriorit.com/dev-blog/195-simple-driver-for-linux-os looks well done.
  - For 5.0
  - I've not double-checked it all