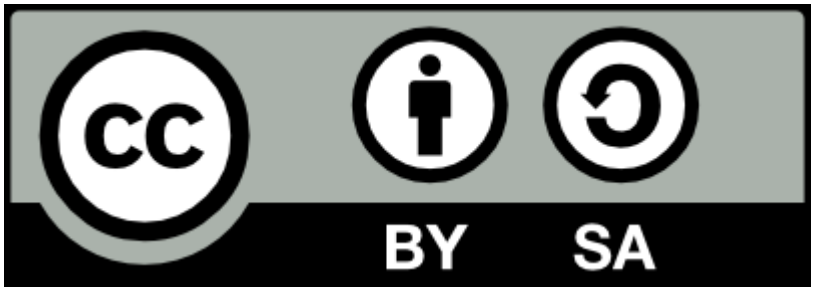
Embedded Linux



These slides created by: Karim Yaghmour

Originals at: www.opersys.com/training/embedded-linux

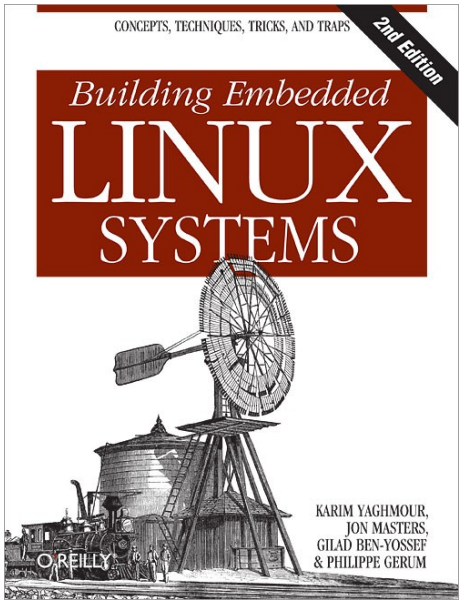
Delivered and/or Customized by:

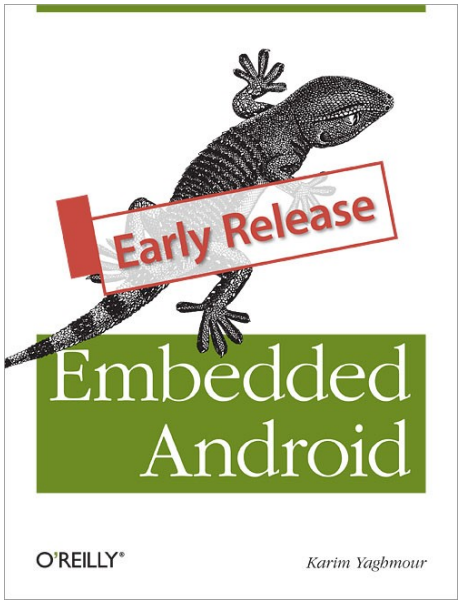
Course Structure and Presentation

1. About the instructor
2. Goals
3. Presentation format
4. Expected knowledge
5. Knowledge fields
6. Day-by-day outline
7. Courseware
8. Hardware

1. About the instructor

- Author of:





- Introduced Linux Trace Toolkit in 1999
- Originated Adeos and relayfs (kernel/relay.c)

2. Goals

- To provide an in-depth understanding of an embedded Linux system's architecture.
- To enable you to put together an embedded Linux system with as little 3rd party dependencies as possible in an architecture-independent fashion.
- To give you a hands-on experience of putting together and programming embedded Linux systems.
- To teach you how open source and free software packages are developed and how to take advantage of that development model.

3. Presentation Format

- Course has two main tracks:
  - Lecture: Instructor presents and discusses material
  - Exercises: Attendees put to practice the material presented with instructor assistance.
- Fast pace. Answers to some questions will be postponed until relevant section is covered.
- Given that there is a lot of material, the instructor will set the speed as required.

4. Expected Knowledge

- Basic embedded systems experience
- Basic understanding of operating system concepts.
- C programming experience
- Basic grasp of open source and free software philosophy.
- Good understanding of the debugging process
- Good understanding of complex project architecture

5. Knowledge fields

- Main fields:
  - Embedded systems
  - Linux kernel internals
  - Device driver development
  - Unix system administration
  - GNU software development
- Traditionally, no single person possesses complete knowledge in all these fields.
- As your experience increase, you will need to look more closely at the relevant fields.

6. Day-by-day outline

- **Day 1: booting a custom kernel and rootfs**
  1. Introduction
  2. Development workspace
  3. Kernel basics
  4. Bootloader setup
  5. Root filesystem

- **Day 2: flash, images, internals and drivers**
  6. Manipulating storage devices
  7. Choosing and installing the rootfs
  8. Kernel internals
  9. Linux hardware support
  10. Device driver overview (start)

- **Day 3: debugging, real-time and networking**
  10. Device driver overview (continued)
  11. Kernel debugging primer
  12. Real-time Linux basics
  13. Debugging applications
  14. Networking services

- **Day 4: toolchain building and human factors**

  15. User interfaces
  16. Development tools setup
  17. Linux, the human factor

- **Appendix:**

  A - Linux basics

7. Courseware

- “Embedded Linux” slides manual.
- Exercise set
- “Building Embedded Linux Systems, 2nd ed.”

8. Hardware

- Motorola EVB5200 / “Lite5200” / “Ice Cube”:
  - PowerPC-based target
  - CPU: MPC5200 (MPC603e core)
  - Freq: 400MHz
  - RAM: 64MB
  - Flash: 16MB
  - I/O: Serial, Ethernet, CAN, I2S, USB, ...
  - Manufacturer: Motorola
  - Model: Lite5200

CAREFUL: Use ESD wrist strap

- TQ Components TQM860:
  - PowerPC-based target
  - CPU: MPC860
  - Freq: 80MHz
  - RAM: 16MB
  - Flash: 8MB
  - I/O: Serial, Ethernet, CAN, ...
  - Manufacturer: TQ Components
  - Model: TQM860

CAREFUL: Use ESD wrist strap

Introduction

1. What's Linux
2. What's embedded Linux
3. What's real-time Linux
4. Generic architecture of an embedded Linux system
5. System startup

1. What's Linux

- Kernel / system / distribution
- Broad definition / many interpretations
- GNU/Linux anyone?
- Strict definition: Linux is the kernel developed and maintained by Linus Torvalds.
- Linux kernel:
  - Provides core system facilities
  - Manages system through its lifecycle (next reboot)
  - Controls all hardware
  - Provides higher-level abstractions to software

- Old kernel version identification scheme:
  - Three digits: x.y.z
  - x.y => version number
  - z => release number
  - Linux 2.4.21 is release 21 of version 2.4
  - Linux 2.5.73 is release 73 of version 2.5
  - Even version number (ex.: 2.4) => stable tree
  - Odd version number (ex.: 2.5) => development tree
- Real-life kernel version identification schemes:
  - 2.4.18-rmk4-hh24 (Handhelds.org) / 2.4.20-8 (Rhat)

- Current version identification scheme
  - Four digits: x.y.z.a
  - x.y => version number
  - z => release number
  - a => stablization increment
  - These four digit releases are not maintained by Linus, but rather by other kernel developers
  - Goal: Provide stable/fixes-only releases.
  - Starting from 2.6.11.1

- Release cycle:
  - After major stable release, 2 week window for major features.
  - 2.6.x released, merge window opened for 2.6.(x+1)
  - Merge window close = 2.6.(x+1)-rc1
  - 6 to 10 week bug fixing brings new 2.6.(x+1)-rcN
  - When stable, 2.6.(x+1) release
  - Lather, rinse, repeat
- No 2.7, 2.9 or 2.(6+n) planned for now
- Maybe when PREEMPT_RT goes fully in = 3.0?

2. What's embedded Linux

- Embedded Linux doesn't exist
- There is no specific kernel for embedded systems
- There are, nevertheless, customized kernels specially configured / customized for specific embedded hardware configurations.
- What does exist:
  - Embedded Linux system
  - Embedded Linux development distribution
  - Embedded Linux target distribution

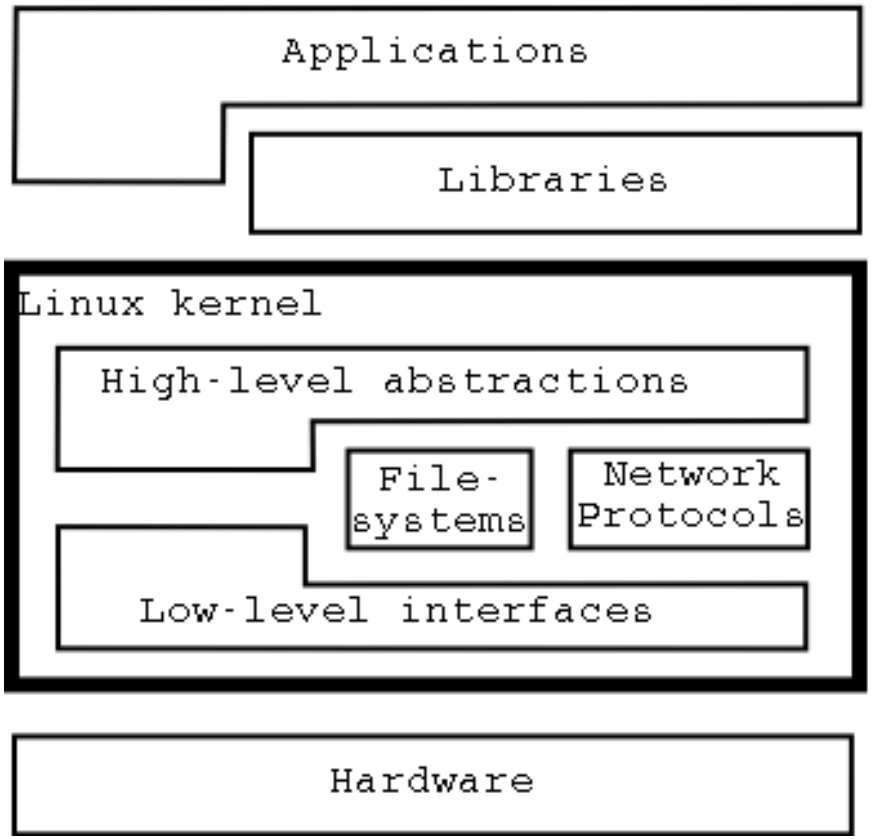
- Embedded Linux system:
  - An embedded system running the Linux kernel
  - User-space tools & configuration likely to be very different from desktop (uClibc instead of glibc, BusyBox instead of core-utils, etc.)
- Embedded Linux development distribution:
  - Includes all the tools and packages required for developing software for embedded Linux systems.
- Embedded Linux target distribution:
  - Includes binaries and related packages to be used directly in embedded Linux system.

3. What's real-time Linux

- Originally (prior to 1999), there was one main real-time Linux extension: RTLinux (first written by M. Barabanov under V. Yodaiken's supervision.)
- Today, the main players are:
  - PREEMPT_RT
  - Xenomai
- Legacy:
  - RTLinux
  - RTAI

4. Generic architecture of an embedded Linux system

- Little or no difference between architecture of a standard Linux system and that of an embedded Linux system.



- Kernel's hardware requirements:
  - 32-bit CPU
  - Memory Management Unit (MMU)
  - Minimal amount of RAM
  - ROM/flash/storage to load/mount root filesystem
- Kernel's responsibilities:
  - Drive devices
  - Manage I/O access
  - Manage memory
  - Manage storage devices
  - Control process scheduling
  - Provide Unix API to applications
  - etc.

- Low-level interfaces:
  - Very hardware specific
  - Provide hardware-independent API
  - Typically handles: CPU-specific operations, architecture-specific memory ops, basic device interfaces.
- High-level abstractions:
  - Provide Unix abstractions (processes, files, sockets, and signals.)
  - Code implementing these abstractions is identical across all architectures (with some minor exceptions.)

- Filesystems:
  - Organize storage devices into recognizable formats
  - Linux supports over 40 different filesystems
  - Virtual Filesystem layer provides transparent and uniform API to all filesystems.
  - Linux requires at least one properly structured root filesystem
- Networking protocols
  - Organize the traffic on the wires
  - Linux supports more than a dozen different protocols
  - Socket API provides uniform API to all networking

- Libraries:
  - Applications almost never access the Linux kernel's services directly.
  - Libraries provide more abstract services than those provided by the kernel.
  - The main library used with the kernel: glibc
  - Substitutes for glibc include: uClibc and eglibc
  - Dynamic linking allows only one copy of each library to be present at all times.
  - Static linking is preferable if a limited set of applications are using a limited set of a library's functionality.

5. System startup

- Software taking part in the system's startup:
  - Bootloader
  - Kernel
  - Init process
- Bootloader:
  - First to run
  - Initializes hardware to a known state
  - Places kernel parameters for the kernel to find
  - Loads kernel and jumps to it

- Kernel:
  - Early startup code is very hardware dependent
  - Initializes environment for the running of C code
  - Jumps to the architecture-independent `start_kernel()` function.
  - Initializes high-level kernel subsystems
  - Mounts root filesystem
  - Starts the init process
- The init process takes care of loading all the user applications and starting the various daemons.

Development workspace

1. Using a practical project workspace
2. Workspace setup
3. Terminal emulators
4. Integrated Development Environments
5. Version control

1. Using a practical project workspace

- Need to organize the components used during cross-platform development. Workspace layout:

bootldr:	target bootloader (s)
build-tools:	toolchain build packages and sources
debug:	debugging tools
doc:	project documentation
images:	binary images ready to be used on target
kernel:	sources and build directories for target kernels
project:	your own custom code for the target
rootfs:	root filesystem as seen on the target
sysapps:	sources for target's system applications
tmp:	temporary data and experiments
tools:	toolchain and all other tools required to build software for the target.

- Location can vary, but $HOME is highly recommended.
- Script for facilitating access to workspace:

```
export PROJECT=example-sys
export PRJROOT=/home/karim/${PROJECT}
cd $PRJROOT
```

- To use this script:

```
$ . devex
```

- **NEVER RUN AS ROOT**

2. Workspace setup

- Complete workspace script (devex)

```
export PROJECT=mpc5200
export PRJROOT=/home/stage/${PROJECT}
export TARGET=powerpc-unknown-linux-gnu
export PREFIX=${PRJROOT}/tools
export TARGET_PREFIX=${PREFIX}/${TARGET}
export PATH=${PREFIX}/bin:${TARGET_PREFIX}/bin:${PATH}
cd $PRJROOT
```

- Possible values for $TARGET:
  - ARM: arm-linux, arm-unknown-linux-gnueabi
  - MIPS: mips-linux, mipsel-unknown-linux-gnu
  - I386: i386-linux, i586-geode-linux-uclibc

- $TARGET cannot change during development. If changed, must recompile entire toolchain.
- $PREFIX is the directory for tools used on host
- $TARGET_PREFIX is the directory for binaries and files used on target.
- Some prefer $PREFIX = /usr/local
- If need to share, try /opt or /home/custom-project
- $TARGET_PREFIX has to be ${PREFIX}/${TARGET}: GNU tools expect this layout.

3. Terminal emulators

- Need to use terminal emulator to access serial port.
- Serial port must be accessible for your user account. Must have access rights to:
  - /dev/ttyS0: `crw-rw---- 1 root dialout  4,64 ... /dev/ttyS0`
- User account must be part of the tty and uucp groups:

```
...
dialout:x:20:karim
...
```

- Terminal emulators:
  - Minicom: simple interface, some problems with U-Boot
  - C-Kermit: powerful communication package
  - UUCP cu: traditional *call up* tool for UUCP
- Installing C-kermit:

```
$ sudo apt-get install ckermit
```

- Other terminal emulators:
  - GTKTerm
  - CuteCom
  - procom

- Configuration file: .kermrc

```
; Line properties
set modem type              none
set line              /dev/ttyS0
set speed                 115200
set carrier-watch            off
set handshake               none
set flow-control            none

; Communication properties
robust
set receive packet-length  1000
set send packet-length     1000
set window                   10

; File transfer properties
set file type              binary
set file names            literal
```

- Starting `kermit`:

```
$ kermit -c
Connecting to /dev/ttyS0, speed 115200
 Escape character: Ctrl-\ (ASCII 28, FS): enabled
Type the escape character followed by C to get back,
or followed by ? to see other options.
----------------------------------------------------
```

4. Integrated Development Environments (IDEs)

- **Eclipse**
- **KDevelop**
- Instructor's personal favorite: xemacs + bash

5. Version control

- Traditional:
  - CVS
  - Subversion
- Distributed:
  - Git
  - Mercurial (hg)

Kernel basics

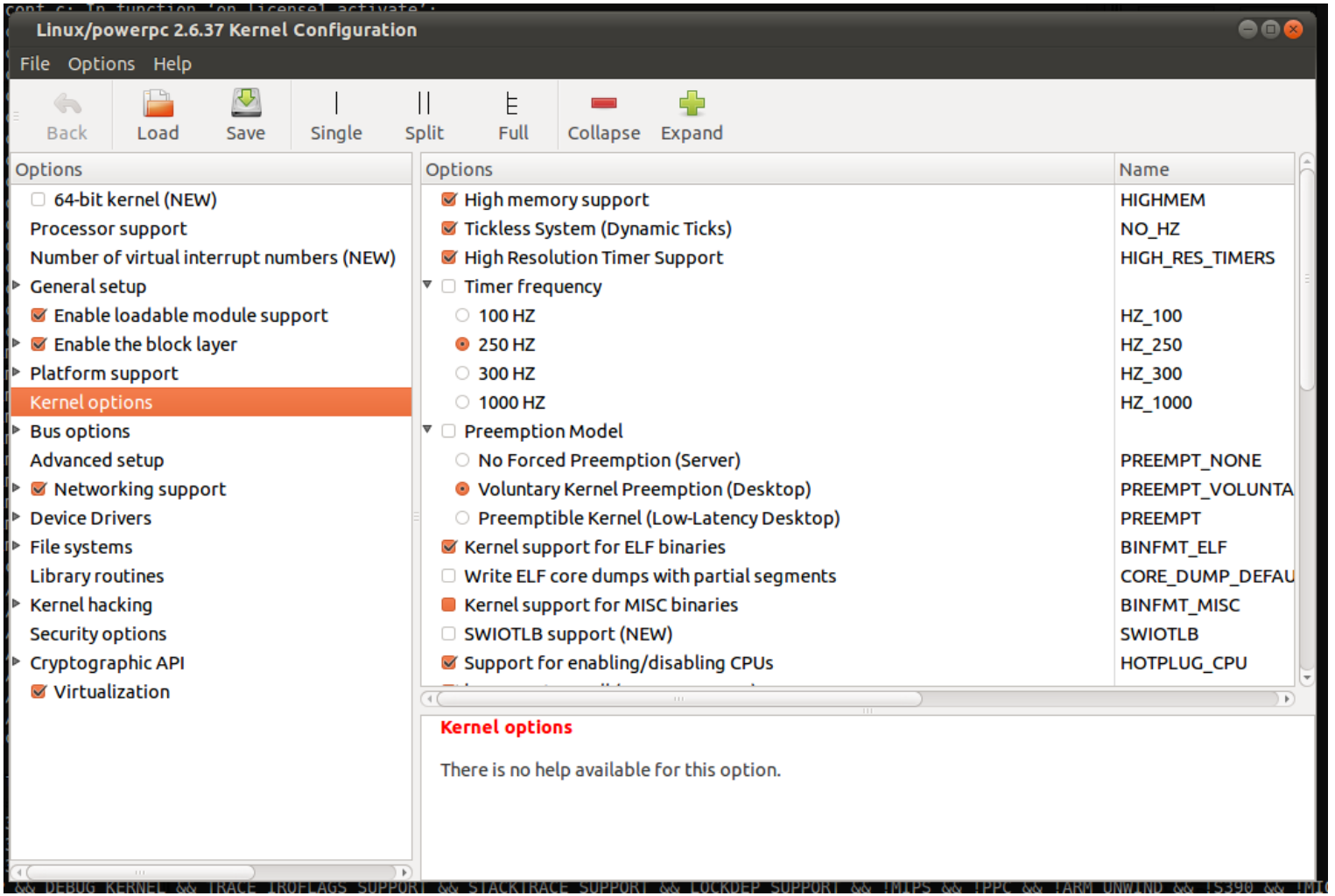
1. Selecting a kernel
2. Configuring the kernel
3. Compiling the kernel
4. Installing the kernel
5. Useful boot parameters
6. Kernel patches

1. Selecting a kernel

- Kernels from http://www.kernel.org/ = best bet
- Lots of branches to choose from – each having separate git repo, though not releases: arch-dependent, hardware-dependent, experimental, ...
- There are also kernels provided by 3rd parties. Make sure that those kernels have community support. Otherwise, you may get locked-in.
- Latest version often the best choice, but may not always work.
- Try different kernel versions

- Keep yourself in touch with the community that develops the kernel you rely on.
- http://kernelnewbies.org/LinuxChanges
- Once selected, download your kernel to ${PRJROOT}/kernel/ and extract it there.
- The exercise instructions contain the download and patch instructions for the kernel used in this course.

2. Configuring the kernel



- Main options:
  - Processor support
  - General setup
  - Enable loadable module support
  - Enable the block layer
  - Platform support
  - Kernel options
  - Bus options
  - Advanced setup
  - Networking support
  - Device Drivers
  - File systems
  - Library routines
  - Kernel hacking
  - Security options
  - Cryptographic API

- Kernel architecture name (ARCH=):
  - x86 => x86
  - ARM => arm
  - PPC => powerpc
  - mips => mips
  - sh => sh
- Some options are visible only to certain archs

- The fact that an option is displayed doesn't mean it's supported.
- The fact that an option isn't displayed doesn't mean it isn't supported.
- Configuration methods:
  - `make config`
  - `make oldconfig`
  - `make menuconfig`
  - `make xconfig`
- All config methods generate a .config file

- Configs will also generate headers & symlinks
- To start configuration menu for your target:

```
$ make ARCH=powerpc CROSS_COMPILE=${TARGET}- menuconfig
```

- Some items can be configured as modules and loaded dynamically at runtime.
- Once the configuration is done, quit the menu and save your configuration. This will create a .config file.
- Some targets have preset configurations:

```
$ make ARCH=powerpc CROSS_COMPILE=${TARGET}- \
> 52xx/lite5200b_defconfig
```

- Can use the menus provided by menuconfig and xconfig to load and save various configurations.
- Simplest way to manipulate configurations: copy the .config files by hand.
- To reuse an old configuration, simply copy the .config back to the kernel's tree and:

```
$ make ARCH=powerpc CROSS_COMPILE=${TARGET}- oldconfig
```

- Place all your .config files in your ${PRJROOT}/kernel directory for easy access.
- Rename your .config files using meaningful names: 2.6.37.config, 2.6.37-frame-buf.config, ...

- EXTRAVERSION:
  - Variable found in top of kernel's main Makefile
  - Useful means to distinguish between variants of the same kernel version.
  - The value used is recorded as part of the kernel image generated and is used to form the name of the directory where modules will be placed.
  - If you use EXTRAVERSION, then maintain naming to all kernel components generated for/by that kernel (ex.: if EXTRAVERSION is “-myco”, the config file should be saved as 2.6.37-myco.config.)

3. Compiling the kernel

1.Building the kernel:

- Command:

```
$ make ARCH=powerpc CROSS_COMPILE=${TARGET}- zImage
```

- Generates a kernel image compressed via gzip
- zImage may not be valid target. Other targets include: vmlinux, bzImage, uImage, cuImage, …
- Use “make ARCH=... help” to find out valid targets
- ARCH variable indicates which architecture subdirectory is to be used.
- CROSS_COMPILE is used to form the name of the tools. For example, `$(CROSS_COMPILE)gcc` becomes `powerpc-linux-gcc`. Hence the trailing “-”.

2.Building the modules:

```
$ make ARCH=powerpc CROSS_COMPILE=${TARGET}- modules
```

- **IF YOU NEED TO** restore kernel tree to distribution state:

```
$ make ARCH=powerpc CROSS_COMPILE=${TARGET}- distclean
```

4. Installing the kernel

- Managing multiple kernel images:
  - Images in ${PRJROOT}/images must be properly identified.
  - For each kernel, there are 4 files to put in ${PRJROOT}/images:
    - The uncompressed image => `vmlinux`
    - The compressed image => depends on arch
    - The kernel symbols => `System.map`
    - The kernel configuration file => `.config`
  - Compressed image is *usually* in the arch/YOUR_ARCH/boot directory.

- See the exercise set for target image we are using.
- Location of images can be seen by looking at arch/YOUR_ARCH/Makefile
- Images placed in ${PRJROOT}/images must follow same naming convention as .config file:

```
$ cp arch/powerpc/boot/cuImage.lite5200 \
> ${PRJROOT}/images/cuImage.lite5200-2.6.37
$ cp System.map ${PRJROOT}/images/System.map-2.6.37
$ cp vmlinux ${PRJROOT}/images/vmlinux-2.6.37
$ cp .config ${PRJROOT}/images/2.6.37.config
```

- Installing the kernel modules:
  - Must install modules in directory following the previously adopted naming convention:

```
$ make ARCH=powerpc CROSS_COMPILE=${TARGET}- \
> INSTALL_MOD_PATH=${PRJROOT}/images/modules-2.6.37 \
> modules_install
```

  - `INSTALL_MOD_PATH` is prepended to the default /lib/modules. Hence, modules installed in:

    ${PRJROOT}/images/modules-2.6.37/lib/modules

5. Useful boot parameters

- Boot parameters can be used to tell kernel about values it may not be able to determine automatically.
- There are quite a few boot parameters available, some are architecture-dependent.
- Root device:
  - Canonical: root=ROOT_DEV
  - Examples: root=/dev/hde1, root=/dev/nfs, root=/dev/ram
- Seconds before reboot on panic:
  - Examples: panic=1, panic=30

- init program:
  - Canonical: init=PATH_TO_YOUR_INIT
  - Examples: init=/bin/bash, init=/custom-prj/startup
- Default console:
  - Canonical: console=CONSOLE_DEV
  - Examples: console=/dev/ttyS0
- Location of NFS server:
  - Canonical: nfsroot=SERVER_IP:SERVER_DIR
  - Examples: nfsroot=192.168.172.222:/home/target

- Mount rootfs as read-only:
  - Canonical: ro
- Mount rootfs as read-write:
  - Canonical: rw
- Change default RAM disk size:
  - Canonical: ramdisk_size = RAMDISK_SIZE
- For the complete list of boot parameters, have a look at Documentation/kernel-parameters.txt.
- Any parameter not recognized by the kernel will be passed on to the `init` process.

6. Kernel patches

- Patch basics
  - A patch is a file containing differences between a certain “official” kernel version and a modified version.
  - Patches are most commonly created using a command line that looks like:

```
$ diff -urN original-kernel modified-kernel > my_patch
```

  - Patches can be incremental (e.g. need to apply patches A, B and C before applying patch D)
  - Patches will easily break if not applied to the exact kernel version they were created for.

- Analyzing a patch's content:

```
$ diffstat -p1 my_patch
```

- Testing a patch before applying it:

```
$ cp my_patch ${PRJROOT}/kernel/linux-2.6.37
$ cd ${PRJROOT}/kernel/linux-2.6.37
$ patch --dry-run -p1 < my_patch
```

- Applying patches:

```
$ patch -p1 < my_patch
```

Bootloader setup

1. Bootloaders galore
2. Server setup for network boot
3. Using LILO with disk and CompactFlash devices
4. Using GRUB with DiskOnChip devices
5. U-Boot
6. Boot directly from the kernel

1. Bootloaders galore

- There a slew of bootloaders out there
- BELS contains summary of open-source bootloaders that will boot Linux on one platform or another.
- Main bootloaders by architecture:
  - x86 => GRUB / LILO
  - ARM => U-Boot
  - PowerPC => U-Boot
  - MIPS => U-Boot
  - SuperH => U-Boot
  - M68k => U-Boot

- LILO: x86
  - Introduced very early in Linux's history
  - Used to be main bootloader for Linux
- GRUB: x86
  - Main bootloader for GNU project
  - Offers more capabilities than LILO (TFTP / DHCP)
  - Gradually replacing LILO
- loadlin: x86
  - Loads Linux from DOS
- Coreboot: x86
  - Linux BIOS replacement

- U-Boot: PPC, ARM, x86, MIPS, SuperH, ...
  - Evolved from forks and merges (8xxrom / PPCBoot / ARMBoot)
  - Very versatile and flexible bootloader
  - Actively maintained
  - Very mature
  - Supports a wide range of boot methods: TFTP, DHCP, IDE, SCSI, DOC, JFFS2, ...
  - Fairly well documented
  - Distributed under the terms of the GPL
  - Bootloader used in this course

- Barebox:
  - Eventual successor to U-Boot
- RedBoot: x86, ARM, PPC, MIPS, SuperH, 68k
  - Red Hat's next generation bootloader
  - Part of eCos source code
  - Doesn't require complete eCos port to run on target
  - Red Hat has dropped eCos (now maintained by eCosCentric.)
  - Should consider U-Boot for all new designs

2. Server setup for network boot

- Services to provide on host:
  - BOOTP/DHCP => target gets network config
  - TFTP => target gets binaries
  - NFS => target mounts rootfs
- DHCP:
  - Must have `dhcpd` installed
  - On Ubuntu:

```
$ sudo apt-get install dhcp3-server
```

  - Host kernel must be configured with `CONFIG_PACKET` and `CONFIG_FILTER` enabled.

- Example configuration file for dhcpd (/etc/dhcpd.conf):

```
subnet 192.168.202.0 netmask 255.255.255.0 {
      option routers 192.168.202.100;
      option subnet-mask 255.255.255.0;

      host example-sys {
            hardware ethernet 00:CF:78:44:AB:9E;
            fixed-address 192.168.202.79;
            option host-name "example-sys";
            next-server 192.168.202.100;
            filename "vmlinux-2.6.37.img";
            option root-path "/home/karim/target";
      }
}
```

- If using a daemon version later than 3.0b2pl11, need to add:

```
ddns-update-style ad-hoc;
```

- To start daemon (Ubuntu):

```
$ sudo /sbin/service dhcp3-server start
```

- Setting up the TFTP daemon:
  - Must have TFTP daemon installed
  - On Ubuntu:

```
$ sudo apt-get install tftpd
```

  - Place files in “/srv/tftp”

- Setting up NFS services:
  - Must have user-space NFS utilities installed
  - On Ubuntu:

    ```
    $ sudo apt-get install nfs-kernel-server
    ```
  - Enabling access to a given IP host (/etc/exports):

    ```
    /home/target 192.168.202.79(rw,no_root_squash,no_subtree_check)
    ```
  - May want to customize hosts.deny and hosts.allow, and/or firewall
  - Starting NFS daemon:

    ```
    $ /sbin/service nfs-kernel-server start
    ```

3. Using LILO with disk and CompactFlash devices

- Disk layout as seen on host is different from disk layout as seen on target (hdb vs. hda.)
- Steps:

1. Create *host* /dev entries on target's rootfs
2. Create LILO configuration files (/etc/target.lilo.conf):

```
boot = /dev/hdb
disk = /dev/hdb
  bios = 0x80

image = /boot/bzImage-2.6.11
  root = /dev/hda1
  label = Linux
  read-only
```

3. If needed, partition target disk

4. Create fs on disk (`mkfs`)
5. Mount target disk somewhere in /mnt
6. Copy rootfs to mounted directory (`cp -a`)
7. Install LILO on target disk

```
$ lilo -r /mnt/target-disk -C etc/target.lilo.conf
Warning: etc/target.lilo.conf should be owned by root
Added Linux *
```

8. Unmount target disk
9. Remove target disk from host
10. Connect target disk to target and boot

- BELS contains full description for CompactFlash

4. Using GRUB with DOC devices

- Initial Program Loader (IPL) started by BIOS at boot time. IPL is in ROM, cannot be replaced.
- IPL loads and starts Secondary Program Loader (SPL).
- GRUB SPL replaces int 19h with custom handler.
- Problem: original int 19h cannot be invoked
- Possible solutions to change broken DOC config:
  - Insert DOC in live circuit
  - Connect DOC address jumper to live circuit
  - Use int 18h
  - Use the CTRL-key bypass

- Main steps for installing GRUB on DOC:
  1. Download GRUB
  2. Patch with DOC patch
  3. Configure with CTRL-key bypass enabled
  4. Build
  5. Copy image to ${PRJROOT}/images
  6. Use `doc_loadbios` MTD utility to copy image to DOC
  7. Power-cycle to guarantee proper firmware install
  8. Add menu.lst configuration file to target's /boot/grub directory.
- See BELS for full details

5. U-Boot

- Download and extract U-Boot (2010.12) to your ${PRJROOT}/bootldr directory.
- Move to the extracted directory for the rest of the operations:

```
$ cd ${PRJROOT}/bootldr/u-boot-2010.12
```

- Use the pre-existing configuration for your board:

```
$ make CROSS_COMPILE=${TARGET}- icecub_5200_LOWBOOT_config
```

- Build U-Boot:

```
$ make CROSS_COMPILE=${TARGET}-
```

- Files generates:
  - u-boot.map => memory map
  - u-boot => ELF image
  - u-boot.bin => raw image for flash
  - u-boot.srec => S-Rec format
- Copy images to ${PRJROOT}/images

```
$ cp u-boot.map ${PRJROOT}/images/u-boot.System.map-2010.12
$ cp u-boot.bin ${PRJROOT}/images/u-boot.bin-2010.12
$ cp u-boot.srec ${PRJROOT}/images/u-boot.srec-2010.12
$ cp u-boot ${PRJROOT}/images/u-boot-2010.12
```

- Copy tools to ${PREFIX}/bin

```
$ cp tools/mkimage ${PREFIX}/bin
```

- Booting with U-Boot:

```
U-Boot 2010.12 (Feb 23 2011 - 16:47:04)

CPU:   MPC5200 v1.2, Core v1.1 at 396 MHz
       Bus 132 MHz, IPB 66 MHz, PCI 33 MHz
Board: Motorola MPC5200 (IceCube)
I2C:   85 kHz, ready
DRAM:  64 MiB
FLASH: 16 MiB
PCI:   Bus Dev VenId DevId Class Int
  00:1a.0     - 1057:5803 - Bridge device
In:    serial
Out:   serial
Err:   serial
Net:   FEC
IDE:   Bus 0: not available

Type run flash_nfs to mount root filesystem over NFS

Hit any key to stop autoboot:  0
=>
```

- Getting some help:

```
=> help
?       - alias for 'help'
base    - print or set address offset
bdinfo  - print Board Info structure
boot    - boot default, i.e., run 'bootcmd'
bootd   - boot default, i.e., run 'bootcmd'
bootm   - boot application image from memory
bootp   - boot image via network using BOOTP/TFTP protocol
cmp     - memory compare
coninfo - print console devices and information
cp      - memory copy
crc32   - checksum calculation
diskboot- boot from IDE device
echo    - echo args to console
editenv - edit environment variable
eeprom  - EEPROM sub-system
env     - environment handling commands
erase   - erase FLASH memory
exit    - exit script
...
```

```
=> help cp
cp - memory copy

Usage:
cp [.b, .w, .l] source target count
=> help printenv
printenv - print environment variables

Usage:
printenv
    - print values of all environment variables
printenv name ...
    - print value of environment variable 'name'
```

- .b, .w, .l, used to specify data type (cp.b, cp.w, ...)
- No need for “0x” or “h”
- Command-name subset recognition

- U-Boot has environment variables:

```
=> printenv
bootdelay=5
baudrate=115200
loads_echo=1
serial#=...
ethaddr=00:CF:78:44:AB:9E
kernel_addr=FF100000
ipaddr=192.168.172.79
serverip=192.168.172.100
stdin=serial
stdout=serial
stderr=serial

Environment size: 1029/16379 bytes
```

- Setting a few variables:

```
=> setenv rootpath /home/karim/target/rootfs
=> setenv kernel_addr FF100000
=> setenv nfscmd setenv bootargs root=/dev/nfs rw nfs→
root=\${serverip}:\${rootpath} ip=\${ipaddr}:\${serverip}→
:\${gatewayip}:\${netmask}:\${hostname}::off panic=1\; bo→
otm \${kernel_addr}
=> setenv bootcmd run nfscmd
```

- Printing variable values (ex.:boot script):

```
=> printenv nfscmd
nfscmd=setenv bootargs root=/dev/nfs rw nfsroot=${serve
rip}:${rootpath} ip=${ipaddr}:${serverip}:${gatewayip}:${
netmask}:${hostname}::off panic=1; bootm ${kernel_addr}
```

- Saving environment variables:

```
=> saveenv
Saving Environment to Flash...
Un-Protected 1 sectors
Erasing Flash...
 done
Erased 1 sectors
Writing to Flash... done
Protected 1 sectors
```

- Deleting variables:

```
=> setenv my-co-image FF400000
=> printenv my-co-image
my-co-image=FF400000
=> setenv my-co-image
=> printenv my-co-image
## Error: “my-co-image” not defined
```

- Don't use “=” when setting variables
- Setting boot command:

```
=> setenv bootcmd run nfscmd
```

- Using the `mkimage` utility to create images:

```
$ mkimage
Usage: mkimage -l image
          -l ==> list image header information
       mkimage [-x] -A arch -O os -T type -C comp -a addr -e ep
  -n name -d data_file[:data_file...] image
          -A ==> set architecture to 'arch'
          -O ==> set operating system to 'os'
          -T ==> set image type to 'type'
          -C ==> set compression type 'comp'
          -a ==> set load address to 'addr' (hex)
          -e ==> set entry point to 'ep' (hex)
          -n ==> set image name to 'name'
          -d ==> use image data from 'datafile'
          -x ==> set XIP (execute in place)
       mkimage [-D dtc_options] -f fit-image.its fit-image
```

- Creating kernel image:

```
$ cd ${PRJROOT}/images
$ mkimage -n '2.6.37 lite5200' \
> -A ppc -O linux -T kernel -C gzip -a 00000000 \
> -e 00000000 -d zImage-2.6.37 zImage-2.6.37.img
Image Name:   2.6.37 lite5200
Created:      Fri Feb 25 11:30:03 2011
Image Type:   PowerPC Linux Kernel Image (gzip compressed)
Data Size:    1683748 Bytes = 1644.29 kB = 1.61 MB
Load Address: 00000000
Entry Point:  00000000
```

- Creating RAM disk image:

```
$ mkimage -n 'initramfs' \
-A ppc -O linux -T ramdisk -C gzip \
-d initramfs.cpio.gz initramfs.img
Image Name:   initramfs
Created:      Fri Feb 25 11:32:06 2011
Image Type:   PowerPC Linux RAMDisk Image (gzip compressed)
Data Size:    3492683 Bytes = 3410.82 kB = 3.33 MB
Load Address: 00000000
Entry Point:  00000000
```

- Converting images to S-Rec format:

```
$ powerpc-unknown-linux-gnu-objcopy -I binary -O srec \
> zImage.img zImage.srec
```

- Booting using BOOTP/DHCP:

```
=> bootp
BOOTP broadcast 1
DHCP client bound to address 192.168.172.205
TFTP from server 192.168.202.100; our IP address is
  192.168.202.79
Filename 'cuImage.lite5200-2.6.37'.
Load address: 0x100000
Loading:
  ##################################################################
  ##########################################
done
Bytes transferred = 1709003 (1a13cb hex)
```

- Verifying loaded images:

```
=> imi 00100000


## Checking Image at 00100000 ...
   Legacy image found
   Image Name:   Linux-2.6.37
   Created:      2011-02-23  15:52:49 UTC
   Image Type:   PowerPC Linux Kernel Image (gzip compressed)
   Data Size:    1708939 Bytes = 1.6 MiB
   Load Address: 00400000
   Entry Point:  004005a4
   Verifying Checksum ... OK
```

- Booting loaded image:

```
=> bootm 00100000
## Booting kernel from Legacy Image at 00100000 ...
   Image Name:   Linux-2.6.37
   Created:      2011-02-23  15:52:49 UTC
   Image Type:   PowerPC Linux Kernel Image (gzip compressed)
   Data Size:    1708939 Bytes = 1.6 MiB
   Load Address: 00400000
   Entry Point:  004005a4
   Verifying Checksum ... OK
   Uncompressing Kernel Image ... OK
[    0.000000] Using lite5200 machine description
[    0.000000] Linux version 2.6.37 (stage@x200) (gcc version 4.4.5 (crosstool-NG-1.10.0) ...
[    0.000000] PCI host bridge /pci@f0000d00 (primary) ranges:
[    0.000000]  MEM 0x0000000080000000..0x000000009fffffff -> 0x0000000080000000 Prefetch
[    0.000000]  MEM 0x00000000a0000000..0x00000000affffffff -> 0x00000000a0000000
[    0.000000]   IO 0x00000000b0000000..0x00000000b0ffffff -> 0x0000000000000000
[    0.000000] Zone PFN ranges:
[    0.000000]   DMA      0x00000000 -> 0x00004000
[    0.000000]   Normal   empty
[    0.000000] Movable zone start PFN for each node
[    0.000000] early_node_map[1] active PFN ranges
[    0.000000]     0: 0x00000000 -> 0x00004000

...
[ 1823.498489] NET: Registered protocol family 17
[ 1823.503200] Registering the dns_resolver key type
[ 1823.608821] Root-NFS: no NFS server address
[ 1823.613187] VFS: Unable to mount root fs via NFS, trying floppy.
[ 1823.620364] VFS: Cannot open root device "(null)" or unknown-block(2,0)
[ 1823.627193] Please append a correct "root=" boot option; here are the available partitions:
[ 1823.635812] Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(2,0)
[ 1823.644319] Rebooting in 180 seconds..
```

- Booting using tftp

```
=> tftp 00100000 cuImage.lite5200-2.6.37
Using FEC device
TFTP from server 192.168.202.100; our IP address is
  192.168.202.79
Filename 'cuImage.lite5200-2.6.37'.
Load address: 0x100000
Loading:
  ################################################################
  ################################################################
  ############################################################ ...
done
Bytes transferred = 1709003 (1a13cb hex)
=> bootm 00100000
## Booting kernel from Legacy Image at 00100000 ...
   Image Name:   Linux-2.6.37
   Created:      2011-02-23  15:52:49 UTC
   Image Type:   PowerPC Linux Kernel Image (gzip compressed)
   Data Size:    1708939 Bytes = 1.6 MiB
   Load Address: 00400000
   Entry Point:  004005a4
   Verifying Checksum ... OK
   Uncompressing Kernel Image ... OK
[    0.000000] Using lite5200 machine description
[    0.000000] Linux version 2.6.37 (stage@x200) (gcc version 4.4.5 (crosstool-NG-1.10.0) ...
[    0.000000] PCI host bridge /pci@f0000d00 (primary) ranges:
...
```

- Setting up a boot script for network boot:

```
=> setenv bootpnfs bootp\; setenv kernel_addr 00100000\; run
  nfscmd
=> printenv bootpnfs
bootpnfs=bootp; setenv kernel_addr 00100000; run nfscmd
=> setenv bootcmd run bootpnfs
=> printenv bootcmd
bootcmd=run bootpnfs
```

- Downloading images to flash:

```
=> erase FF100000 FF2FFFFF
.................................. done
Erased 32 sectors
=> tftp 00100000 cuImage.lite5200-2.6.37
Using FEC device
TFTP from server 192.168.202.100; our IP address is 192.168.202.79
Filename 'cuImage.lite5200-2.6.37'.
Load address: 0x100000
Loading:
  ################################################################
  ################################################################
  ################################################# ...
done
Bytes transferred = 1709003 (1a13cb hex)
```

```
=> printenv filesize
filesize=1A13CB
=> cp.b 00100000 ff100000 $filesize
Copy to Flash... done
=> imi FF100000

## Checking Image at ff100000 ...
   Legacy image found
   Image Name:   Linux-2.6.37
   Created:      2011-02-23  15:52:49 UTC
   Image Type:   PowerPC Linux Kernel Image (gzip compressed)
   Data Size:    1708939 Bytes = 1.6 MiB
   Load Address: 00400000
   Entry Point:  004005a4
   Verifying Checksum ... OK
```

- Booting Using a RAM disk

```
=> tftpboot 00300000 initramfs.img
Using FEC device
TFTP from server 192.168.202.100; our IP address is 192.168.202.79
Filename 'initramfs.img'.
Load address: 0x300000
Loading:
  ###################################################################
   ...
  ###################################################################
done
Bytes transferred = 3492747 (354b8b hex)
=> imi 00300000

## Checking Image at 00300000 ...
   Legacy image found
   Image Name:   initramfs
   Created:      2011-02-25  16:32:06 UTC
   Image Type:   PowerPC Linux RAMDisk Image (gzip compressed)
   Data Size:    3492683 Bytes = 3.3 MiB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
```

```
=> bootm 00100000 00300000
## Booting kernel from Legacy Image at 00100000 ...
   Image Name:   Linux-2.6.37
   Created:      2011-02-23  15:52:49 UTC
   Image Type:   PowerPC Linux Kernel Image (gzip compressed)
   Data Size:    1708939 Bytes = 1.6 MiB
   Load Address: 00400000
   Entry Point:  004005a4
   Verifying Checksum ... OK
## Loading init Ramdisk from Legacy Image at 00300000 ...
   Image Name:   initramfs
   Created:      2011-02-25  16:32:06 UTC
   Image Type:   PowerPC Linux RAMDisk Image (gzip compressed)
   Data Size:    3492683 Bytes = 3.3 MiB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
   Uncompressing Kernel Image ... OK
   Loading Ramdisk to 03b8c000, end 03ee0b4b ... OK
[    0.000000] Using lite5200 machine description
[    0.000000] Linux version 2.6.37 (stage@x200) (gcc version ...
```

- Updating U-Boot (**CAREFUL**):

```
=> tftp 00100000 u-boot.bin-2010.12
Using FEC device
TFTP from server 192.168.202.100; our IP address is 192.168.202.79
Filename 'u-boot.bin-2010.12'.
Load address: 0x100000
Loading: ##################################################
done
Bytes transferred = 280200 (44688 hex)
=> protect off FF000000 FF04FFFF
Un-Protected 5 sectors
=> erase FF000000 FF04FFFF
... done
Erased 5 sectors
=> cp.b 00100000 FF000000 $filesize
Copy to Flash... done
=> setenv filesize
```

```
=> saveenv
Saving Environment to Flash...
Un-Protected 1 sectors
Erasing Flash...
 done
Erased 1 sectors
Writing to Flash... done
Protected 1 sectors
=> reset

U-Boot 2010.12 (Feb 23 2011 - 16:47:04)

CPU:   MPC5200 v1.2, Core v1.1 at 396 MHz
       Bus 132 MHz, IPB 66 MHz, PCI 33 MHz
Board: Motorola MPC5200 (IceCube)
I2C:   85 kHz, ready
DRAM:  64 MiB
...

Hit any key to stop autoboot:  0
=>
```

6. Booting directly from the kernel

- Can be done (ex.: coreboot)
- Not the best way to go: not very configurable
- Need to hard-code all boot options in kernel code.
- Kernel's offset must be properly set so that it is located at the CPU's startup address.
- Kernel has to take care of all hardware initialization.
- Kernel may have to copy itself (its data and bss at least) to RAM.

Root filesystem

1. Libraries
2. Kernel modules
3. Kernel images
4. Device files
5. Main system applications
6. Custom applications
7. Basic root filesystem structure
8. System initialization
9. Additional applications
10. Auto-generating filesystems

1. Basic root filesystem structure

- Unix FS structured for multi-user systems
- Some directories not necessary for embedded
- Filesystem Hierarchy Standard (FHS):
  - /bin => Essential user binaries
  - /boot => Bootloader and kernel images
  - /dev => Device files
  - /etc => System configuration
  - /home => User home directories
  - /lib => Essential shared libs and kernel modules
  - /mnt => Temporary mount point
  - /opt => Add-on software packages
  - /sbin => Essential system binaries
  - /tmp => Temporary files
  - /usr => Secondary hierarchy (mostly user apps)
  - /var => Variable data generated by daemons

- Non-essential multi-user dirs:
  - /home, /mnt, /opt, /root
- Depends on bootloader:
  - /boot
- Essential:
  - /bin, /dev, /etc, /lib, /proc, /sbin, /usr, /tmp, /var
- Contain their own hierarchy:
  - /usr, /var

- What are all these binaries directories for?
  - /bin => Essential binaries for user and admin
  - /sbin => Essential binaries for admin
  - /usr/bin => Non-essential user and admin binaries
  - /usr/sbin=> Non-essential admin binaries
- What are all those libraries directories for?
  - /lib => Essential system libraries
  - /usr/lib => Non-essential libraries
- The kernel does not force FS layout. Layout is “universally” agree upon (i.e. FHS.)

- To start working on rootfs:

```
$ cd ${PRJROOT}/rootfs
```

- Create core rootfs directories:

```
$ mkdir bin dev etc lib proc sbin tmp usr var
$ chmod 1777 tmp
```

- Create the /usr hierarchy:

```
$ mkdir usr/{bin,lib,sbin}
```

- Create the /var hierarchy:

```
$ mkdir var/{lib,lock,log,run,tmp}
$ chmod 1777 var/tmp
```

- JFYI: Linux can be run **without** a rootfs ...

2. Libraries

1. glibc
2. uClibc

2.1. glibc

- glibc components:
  - Actual shared libraries:
    - Format: libLIB_NAME-GLIBC_VER.so
    - Examples: libm-2.3.2.so, libc-2.3.2.so
  - Major revision version symbolic links:
    - Format: libLIB_NAME.so.MAJOR_REV_VER
    - Examples: libdl.so.2, libc.so.6
  - Version-independent symbolic links to the major revision version symbolic links:
    - Format: libLIB_NAME.so
    - Examples: libdl.so, libm.so
  - Static library archives:
    - Format: libLIB_NAME.a
    - Examples: libdl.a, libm.a

- For target, need:
  - The actual shared libs
  - The major revision version symbolic links
- Also need dynamic linker:
  - Actual linker: ld-GLIBC_VER.so
  - Symbolic link to linker:
    - x86, ARM, SH, m68k => ld-linux.so.MAJOR_REV_VER
    - MIPS, PPC => ld.so.MAJOR_REV_VER
- Must determine exact library components required.
- BELS table 6.2 contains complete list

- Most important components:
  - ld => the dynamic linker
  - libc => the C library
  - libm => the math library
  - libdl => the shared objects manipulation library
- Must determine exact dependencies of your applications.
- Native `ldd` is not cross-platform-capable
- Can use `readelf` or `uclibc-ldd`:

- Using readelf:

```
$ powerpc-linux-readelf -a ${PRJROOT}/rootfs/bin/busybox \
> | grep "Shared library"
 0x00000001 (NEEDED)                     Shared library: [libc.so.0]
```

- Using uclibc-ldd:

```
$ powerpc-uclibc-ldd ${PRJROOT}/rootfs/bin/busybox
libc.so.0 => /home/karim/example-sys/tools/uclibc/lib/libc.so.0
/lib/ld-uClibc.so.0 => /lib/ld-uClibc.so.0
```

- Copying important libraries to target rootfs:

```
$ cd ${TARGET_PREFIX}/lib
$ for file in libc libcrypt libdl libm \
> libpthread libresolv libutil
> do
> cp $file-*.so ${PRJROOT}/rootfs/lib
> cp -d $file.so.[*0-9] ${PRJROOT}/rootfs/lib
> done
$ cp -d ld*.so* ${PRJROOT}/rootfs/lib
```

- Copying all libraries:

```
$ cd ${TARGET_PREFIX}/lib
$ cp *-*.so ${PRJROOT}/rootfs/lib
$ cp -d *.so.[*0-9] ${PRJROOT}/rootfs/lib
$ cp libSegFault.so libmemusage.so libpcprofile.so \
> ${PRJROOT}/rootfs/lib
```

- Copying the C++ library:

```
$ cp -d libstdc++.so.* ${PRJROOT}/rootfs/lib
```

- Stripping all target libraries for space efficiency:

```
$ powerpc-linux-strip ${PRJROOT}/rootfs/lib/*.so*
```

2.2. uClibc

- Same naming conventions as glibc
- Implements most of the glibc components:
  - ld, libc, libcrypt, libdl, libm, libpthread, libresolv, libutil.
- uClibc libraries can coexist with glibc libraries in target's /lib directory.
- Copying important libraries to target rootfs:

```
$ cd ${PREFIX}/uclibc/lib
$ for file in libuClibc ld-uClibc libc libdl \
> libcrypt libm libresolv libutil
> do
> cp $file-*.so ${PRJROOT}/rootfs/lib
> cp -d $file.so.[*0-9] ${PRJROOT}/rootfs/lib
> done
```

- Copying all uClibc components:

```
$ cd ${PREFIX}/uclibc/lib
$ cp *-*.so ${PRJROOT}/rootfs/lib
$ cp -d *.so.[*0-9] ${PRJROOT}/rootfs/lib
```

- No need to strip uClibc libraries, they are stripped by the uClibc build script.

3. Kernel modules

- Kernel modules are located in /lib/modules, so they must be installed in ${PRJROOT}/rootfs/lib/modules.
- Copying modules built earlier:

```
$ cp -a ${PRJROOT}/images/modules-2.6.37/* \
> ${PRJROOT}/rootfs
```

- Module loading customization (/etc/modprobe.conf or /etc/modprobe.d/)

4. Kernel images

- Copy kernel images to rootfs only if bootloader is capable of reading fs structures.
- Copying kernel image:

```
$ mkdir ${PRJROOT}/rootfs/boot
$ cd ${PRJROOT}/images
$ cp zImage-2.6.37 ${PRJROOT}/rootfs/boot
```

- If you copied kernel, also copy config:

```
$ cp 2.6.37.config ${PRJROOT}/rootfs/boot
```

5. Device files

- All devices in Linux are seen as files (except Ethernet interfaces.)
- Typical workstation distros use udev
- Keep a copy of Documentation/devices.txt handy
- See BELS table 6.3 for core set of /dev entries
- Properties of each /dev node:
  - Filename (node name)
  - Type (char / block)
  - Major number (What type of device?)
  - Minor number (Which instance of the device?)
  - Permission bits

- Creating core /dev nodes:

```
$ cd ${PRJROOT}/rootfs/dev
$ su -m
Password:
# mknod -m 600 mem c 1 1          Physical memory access
# mknod -m 666 null c 1 3         Null device
# mknod -m 666 zero c 1 5         Null byte source
# mknod -m 644 random c 1 8       Nondeterministic rnd nbr
# mknod -m 600 tty0 c 4 0         Current virtual console
# mknod -m 600 tty1 c 4 1         First virtual console
# mknod -m 600 ttyS0 c 4 64       First UART serial port
# mknod -m 666 tty c 5 0          Current TTY device
# mknod -m 600 console c 5 1      System console
# exit
```

- Creating compulsory /dev symlinks:

```
$ ln -s /proc/self/fd fd
$ ln -s /proc/self/fd/0 stdin
$ ln -s /proc/self/fd/1 stdout
$ ln -s /proc/self/fd/2 stderr
```

6. Main system applications

- Unix systems rely on a common set of commands
- Standard distros have one binary per command
- May compile each relevant command one-by-one or use packages that provide many commands in a single binary:
  1. Complete sytem apps
  2. BusyBox
  3. TinyLogin
  4. Other choices

6.1. Complete system applications

- Very tedious and problem-prone process (unless successfully automated ...)
- Best way to go:
  - Linux From Scratch project:

    http://www.linuxfromscratch.org/
  - There is a an LFS book explaining exactly what to do, and how much time it takes.

6.2. BusyBox

- Project started to help Debian install disks
- Currently lead by the uClibc maintainer
- Main package used in embedded Linux to provide core set of Unix commands.
- Home page: http://www.busybox.net/
- Complete list of commands:

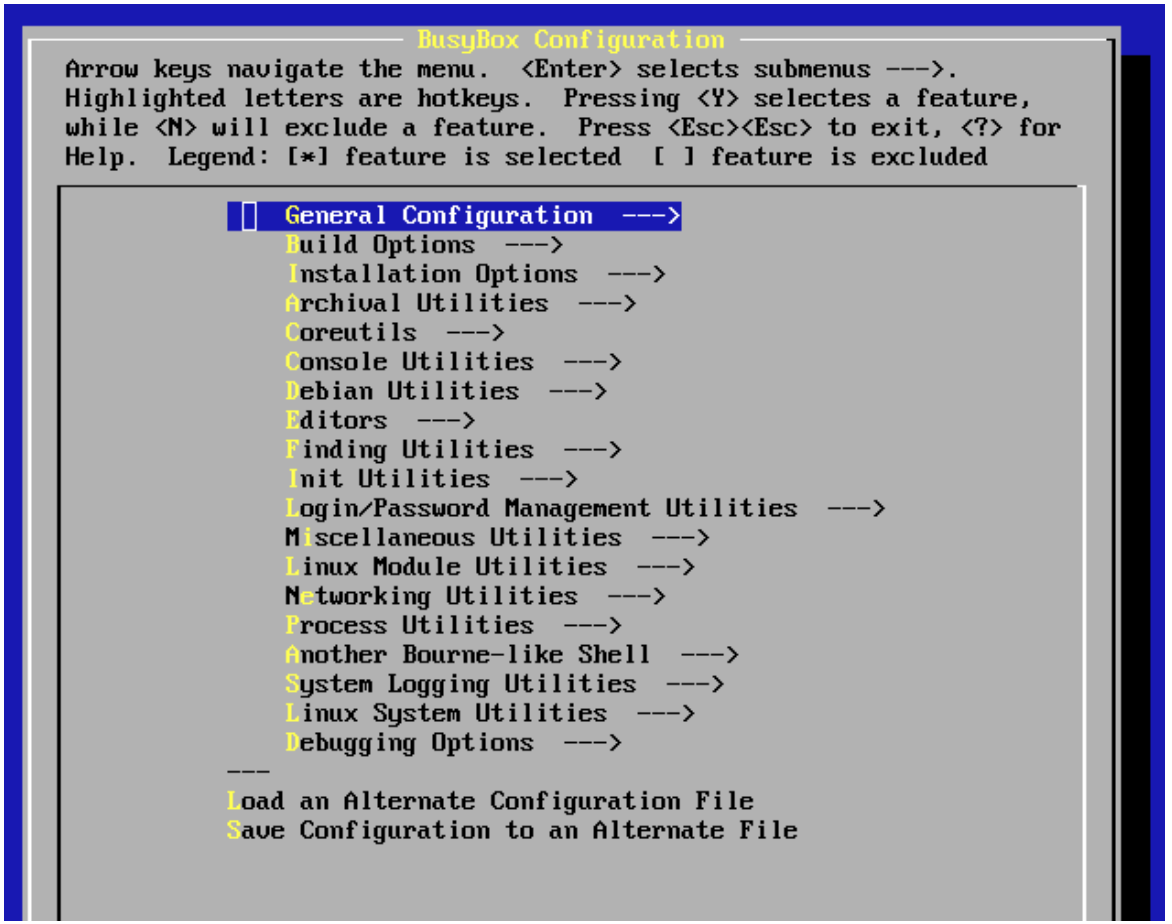
adjtimex, ar, basename, cat, chgrp, chmod, chown, chroot, chvt, clear, cmp, cp, cpio, cut, date, dc, dd, deallocvt, df, dirname, dmesg, dos2unix, dpkg, dpkg_deb, du, dumpkmap, dutmp, echo, env, expr, false, fbset, fdflush, find, free, freeramdisk, fsck_minix, getopt, grep, gunzip, gzip, halt, head, hostid, hostname, id, ifconfig, init, insmod, kill, killall, klogd, lash, length, ln, loadacm, loadfont, loadkmap, logger, logname, logread, losetup, ls, lsmod, makedevs, md5sum, mkdir, mkinfo, mkfs_minix, mknod, mkswap, mktemp, modprobe, more, mount, mt, mv, nc, nslookup, pidof, ping, pivot_root, poweroff, printf, ps, pwd, rdate, readlink, reboot, renice, reset, rm, rmdir, rmmod, route, rpm2cpio, sed, setkeycodes, sleep, sort, stty, swapoff, swapon, sync, syslogd, tail, tar, tee, telnet, test, tftp, time, touch, tr, traceroute, true, tty, umount, uname, uniq, unix2dos, update, uptime, usleep, uudecode, uuencode, vi, watchdog, wc, wget, which, whoami, xargs, yes, and zcat.

- Download BusyBox (1.18.3) to your ${PRJROOT}/sysapps directory and extract it there.
- Move to the directory for the rest of the setup:

```
$ cd ${PRJROOT}/sysapps/busybox-1.18.3
```

- Configuration of BusyBox's options:

```
$ make menuconfig
```


BusyBox Configuration
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> selectes a feature,
while <N> will exclude a feature. Press <Esc><Esc> to exit, <?> for
Help. Legend: [*] feature is selected [ ] feature is excluded
General Configuration --->
Build Options --->
Installation Options --->
Archival Utilities --->
Coreutils --->
Console Utilities --->
Debian Utilities --->
Editors --->
Finding Utilities --->
Init Utilities --->
Login/Password Management Utilities --->
Miscellaneous Utilities --->
Linux Module Utilities --->
Networking Utilities --->
Process Utilities --->
Another Bourne-like Shell --->
System Logging Utilities --->
Linux System Utilities --->
Debugging Options --->

Load an Alternate Configuration File
Save Configuration to an Alternate File

- Options that must be set:
  - “Build Options” -> Cross-compiler prefix:

    `${TARGET}-`
  - “Installation Options” -> Installation prefix:

    `${PRJROOT}/rootfs`
- Build:

  `$ make`
- Install:

  `$ make install`

- Only one binary has been installed: /bin/busybox
- All commands are symbolic links to /bin/busybox
- Determining the command issued done through main's `argv[]` and `argc`.
- Creating arbitrary links doesn't work
- BusyBox can be told to create hard-links
- Full command doc on web and in package
- Customizing the paths for the various shells:

```
# Set path
PATH=/bin:/sbin:/usr/bin:/usr/sbin
export PATH
```

6.3. TinyLogin

- Used to be BusyBox's little brother
- Now integrated into BusyBox 1.00
- Provides login and authentication utilities
- Commands provided:

```
addgroup, adduser, delgroup, deluser, login, su,
getty, passwd, sulogin, vlock
```

- See the “Login/Password Management Utilities” submenu in the BusyBox configuration.

- Must setuid BusyBox to ensure login applets will function properly:

```
$ su -m
# cd ${PRJROOT}/rootfs/bin
# chmod u+s busybox
# exit
```

- Must create: /etc/group, /etc/passwd, /etc/shadow
- Copy /etc/group and /etc/passwd from host and customize for target.
- Create dummy user on host to obtain valid passwords to put in /etc/shadow:

```
tmp:$1$3cdOSELf$XWRLoKIL7vMSfLYbRCWaf/:11880:0:99999:7:-1:-1:0
```

- Edit and copy entry to target /etc/shadow:

```
root:$1$3cdOSELf$XWRLoKIL7vMSfLYbRCWaf/:11880:0:99999:7:-1:-1:0
```

- /etc/passwd:
  - Canonical: username:x:UID:GID:user-info:home-dir:login-shell
  - Example: `root:x:0:0:root:/root:/bin/sh`
- Default paths for each user:

```
PATH=/bin:/usr/bin
```

- Customize path using /etc/profile or .profile in home directory:

```
PATH=/bin:/sbin:/usr/bin:/usr/sbin
export PATH
```

6.4. Other choices

- Embutils:
  - Written and maintained by diet libc's author
  - Provides many small binaries, one for each command, instead of a single unified binary for all commands.
  - See BELS for setup and use
- Toolbox:
  - Part of Google's Android distribution
  - BSD-licensed
  - Works in the same way Busybox works: single binary
  - Much more bare-bones than Busybox

7. Custom applications

- Can interpret FHS rules loosely
- Try /bin if your application is simple
- Use your own tree hierarchy in “/” if your project is large.
- Remember to set the PATH environment variable correctly.

8. System initialization

- Unix initialization is a world of its own
- Typical modern Unix startup similar to System V `init`.
- As we saw earlier, the kernel can be provided with a custom init (init= boot parameter.)
- A true `init` program will almost invariable be a better choice than a custom `init`.
- When `init` dies, there's a kernel panic
- 2 real choices for `init`:
  - System V `init`
  - BusyBox `init`

- System V `init`:
  - The `init` used in most mainstream distribution
  - Available from: ftp://ftp.cistron.nl/pub/people/miquels/sysvinit/
  - Package includes: halt, init, killall5, last, mesg, runlevel, shutdown, sulogin, utmpdump, wall
  - Download package (2.85) into your ${PRJROOT}/sysapps directory and extract it there.
  - Move to directory and build package:

```
$ cd ${PRJROOT}/sysapps/sysvinit-2.85/src
$ make CC=powerpc-linux-gcc
```

- Install package:

```
$ make BIN_OWNER="$(id -un)" BIN_GROUP="$(id -gn)" \
> ROOT=${PRJROOT}/rootfs install
```

- Ignore install's failure to install man pages
- BIN_OWNER and BIN_GROUP necessary to avoid Makefiles trying to install as root.
- Need properly formatted /etc/inittab (runlevels description) and properly populated /etc/rc.d (tasks to run on each runlevel).
- See Alessandro Rubini's “Take Command: Init” article for complete details of `init` mechanics.

- Runlevel description (7 runlevels):

  0 => Halt

  1 => Single user mode

  2 => Multiuser, limited networking

  3 => Full multiuser

  4 => Unused

  5 => X11 (GUI)

  6 => Reboot

- Can set embedded system to 1 or 3, according to use
- Need to create fifo for `init` to be able to talk to itself:

```
$ mknod -m 600 ${PRJROOT}/rootfs/dev/initctl p
```

- BusyBox `init`:
  - Already installed earlier, if properly configured BusyBox.
  - As other BusyBox apps, `init` is a symbolic link to /bin/busybox.
  - Main BusyBox initialization tasks:
    1. Setting up sig handlers
    2. Initialize console
    3. Parse /etc/inittab
    4. Run /etc/init.d/rcS
    5. Run all `wait` inittab commands
    6. Run all `once` inittab commands

- Once initialization done, BusyBox loops on:
  1. Run all `respawn` inittab commands
  2. Run all `askfirst` inittab commands
- BusyBox will complain if no inittab is present and /dev/tty1 through /dev/tty4 don't exist.
- See BusyBox doc for complete BusyBox behavior
- Default format of inittab entries:

  id:runlevel:action:process
  - id is controlling tty
  - runlevel is ignored (there's only one runlevel)
  - action is type of BusyBox `init` action
  - process is path to binary

- Type of BusyBox `init` actions:

sysinit	=>	path to rcS
respawn	=>	restart task when it exits
askfirst	=>	ask for confirmation on console before starting task.
wait	=>	wait on task before continuing init
once	=>	run task just once
ctrlaltdel	=>	run task on CTRL+ALT+DEL combo
shutdown	=>	run task on system shutdown
restart	=>	run task when init restarts

- Simple /etc/inittab:

```
::sysinit:/etc/init.d/rcS
::respawn:/bin/sh
::respawn:/bin/custom-app
::restart:/sbin/init
::shutdown:/bin/umount -a -r
```

- Simple /etc/init.d/rcS:

```
#!/bin/sh
#mount -n -o remount,rw /
mount -t proc none /proc
mount -t sysfs none /sys
/sbin/ifconfig eth0 192.168.202.79
/bin/sh
```

- Simple /etc/fstab:

```
# /etc/fstab
# device        directory    type     options
#
/dev/nfs        /            nfs      defaults
none            /proc        proc     defaults
```

9. Additional applications

- Autoconf-based:
  1. Unpack
  2. Configure:

```
$ CC=powerpc-linux-gcc ./configure --host=$TARGET \
> --prefix=${TARGET_PREFIX}
```

  3. Build:

```
$ make CFLAGS="..." LDFLAGS=""
```

  4. Install:

```
$ make install
```

  5. Create / copy proper configuration files

- Non-configure based:
  - Custom configuration system:
    - Some packages have custom config scripts (kernel, uClibc, etc.)
    - See package documentation for more details on how to operate scripts.
    - See packages built on same framework for more details
  - Hostile:
    - Some packages are **hostile** to cross-compilation (we will see at least one example later.)
    - Require manual editing to Makefile / config scripts
    - May require source code changes

10. Auto-generating filesystems

- Yocto
- Buildroot
- PTXdist
- OpenWRT
- LTIB

Manipulating storage devices

1. MTD subsystem concepts
2. MTD usage basics
3. Native CFI flash
4. Disk devices
5. Swapping

1. MTD subsystem concepts

- MTD support for solid-state storage devices: flash, DOC, ROM, RAM, etc.
- 3 layers:
  - MTD chip drivers
  - MTD “glue logic”
  - MTD “user modules”
- Main MTD project URL:

  http://www.linux-mtd.infradead.org

- Chip drivers:
  - NAND
  - OneNAND
  - CFI
  - NOR
  - RAM, ROM, and absent chips
  - Uncached RAM
  - Virtual devices for testing and evaluation
  - DiskOnChip (bit rot)

- “Mapping drivers” allow the MTD subsystem to locate and access the actual chips.
- When your chip is supported, you need to write the mapping driver for your board.
- MTD can manage multiple instances of a chip
- MTD can partition and concatenate devices
- User-modules provide additional functionality which allow chips to be accessed through the kernel's virtual filesystem layer.

- User-modules:
  - UBIFS:
    - Works on top of UBI (Unsorted Block Images), on top of MTD
    - UBI provides wear-leveling
    - Scalability
    - Fast mount, fast I/O, on-the-flight compression
    - Provides power-down reliability
  - YAFFS2
    - NAND flash
    - Includes wear-leveling
    - No compression
    - Provides power-down reliability (no fsck)

- JFFS2 (Journalling Flash File System version 2):
  - Implements log-structured FS on device
  - FS recreated in RAM at mount time
  - Includes wear-leveling
  - Implements data compression
  - Provides power-down reliability (no fsck)
- NFTL (NAND Flash Translation Layer) and INFTL (Inverse NFTL) for M-Systems DiskOnChip devices
- FTL (Flash Translation Layer), part of PCMCIA std.

- Char device:
  - Provides direct access to chip as a char device
  - Very useful for programming device in Linux
- Caching block device:
  - Provides a simple block device interface to MTD devices
  - Can mount a block FS straight on it for reading & writing
  - Does NOT provide power-down reliability
  - Does NOT implement smart device management
- Read-only block device:
  - Same as caching block device, but read-only

2. MTD usage basics

- MTD subsystem developed independently from kernel.
- Recommended to use latest kernel
- Bleeding edge is in MTD git repo
- MTD subsystem requires /dev entries that differ from devices.txt file specification.
- MTD /dev entries are as listed in BELS table 7.1 and 7.2.

- Types of MTD /dev entries:
  - mtdN => each instance is separate MTD device or partition.
  - mtdrN => each instance is read-only copy of mtdN entry.
  - mtdblockN => each instance is block device copy of mtdN.
  - nftlN => each instance is separate NFTL device. Further numbered like hd.
  - ftlN => similar naming as nftlN
- If MTD devices are accessible on host, can use the packaged device creation script.

- Main MTD kernel configuration options:
  - Memory Technology Device (MTD) support:
    Must be set to use MTD subsystem
  - MTD partitioning/concatenating support:
    Enables partitioning/concatenating MTD devices
  - Direct char device access to MTD devices:
    Required for accessing MTD devices as char devices
  - Caching block device access to MTD devices:
    Enables mounting MTD device as normal rw block device
  - Read-only block device access to MTD devices:
    Same as above, but read-only
  - FTL (Flash Translation Layer)
    Provides FTL MTD user module
  - NFTL (NAND Flash Translation Layer)
    Provides NFTL MTD user module

- INFTL (Inverse NAND Flash Translation Layer)
  Provides INFTL MTD user module
- Can only set mtdblock or mtdblock_ro as built-in. Can set both as modules.
- Support for JFFS2 and JFFS is found in the “File Systems” submenu.
- MTD Submenus:
  - RAM/ROM/Flash chip drivers:
    Support for CFI, JEDEC, non-CFI, RAM, ROM, etc.
  - Mapping drivers for chip access
    Default mapping drivers for quite a few boards
  - Self-contained MTD device drivers
    Uncached RAM, VM test drv, blk dev emu, DOC devices

- NAND Flash Device Drivers
  NAND flash
- During development:
  - Build MTD options as modules
- For production:
  - Select MTD options as built-in
- MTD utilities:
  - MTD subsystem requires special tools (can't use conventional disk utilities.)
  - MTD utilities are very powerful. Be careful when using them, you could destroy your device.

- Generic tools:
  - flash_info => get erase information
  - flash_erase => erase device
  - flash_eraseall => erase entire device
  - flash_unlock => unlock flash protection
  - flash_lock => lock flash protection
  - flashcp => copy file to flash
  - doc_loadbios => copy bootloader to device
  - mtd_debug => access MTD debugging

- Filesystem creation tools:
  - mkfs.jffs2 => create JFFS2 filesystem image
  - mkfs.jffs => create JFFS filesystem image
  - jffs2dump => dumps the content of a JFFS2 image (can't mount JFFS2 on loopback.)
  - mkfs.ubifs => create UBI filesystem image
- NFTL tools:
  - nftl_format => write NFTL format on device
  - nftldump => dump NFTL content to file
- FTL tools:
  - ftl_format => write FTL format on device
  - ftl_check => verify FTL format

- NAND chip tools
  - nandwrite => write to NAND device
  - nandtest => test NAND device
  - nandump => dump NAND device content
- Installing MTD tools on host:
  - Get and put in ${PRJROOT}/build-tools:
    - Zlib: http://www.zlib.net
    - LZO: http://www.oberhumer.com/opensource/lzo/
    - Libacl: http://oss.sgi.com/projects/xfs/
    - MTD: git://git.infradead.org/mtd-utils
  - Make sure you have LZO And UUID installed:

```
$ sudo apt-get install libz-dev
$ sudo apt-get install liblzo2-dev
$ sudo apt-get install uuid-dev
```

- Installing MTD tools for target:
  - Configure and compile zlib:

```
$ cd ${PRJROOT}/build-tools/
$ tar xvzf zlib-1.2.5.tar.gz
$ cd zlib-1.2.5
$ CC=${TARGET}-gcc \
> LDSHARED="${TARGET}-ld -shared" \
> ./configure --shared
$ make
$ make prefix=${TARGET_PREFIX} install
```

  - Configure and compile lzo:

```
$ cd ${PRJROOT}/build-tools/
$ tar xvzf lzo-2.03.tar.gz
$ cd lzo-2.03
$ CC=powerpc-unknown-linux-gnu-gcc ./configure \
> --enable-shared --host=${TARGET}
$ make
$ make prefix=${TARGET_PREFIX} install
```

- Got to ${PRJROOT}/build-tools
- Extract git snapshot:

```
$ git clone git://git.infradead.org/mtd-utils
```

- Copy the acl.h header:

```
$ cd ${PRJROOT}/build-tools
$ tar xvzf acl_2.2.47-1.tar.gz
$ mkdir ${PRJROOT}/build-tools/mtd-utils/include/sys
$ cp acl-2.2.47/include/acl.h \
> ${PRJROOT}/mtd-utils/include/sys
```

- Go to mtd-utils/ and build tools for host:

```
$ make
```

- Install tools:

```
$ make DESTDIR=${PREFIX} SBINDIR="/bin" install
```

3. Native CFI flash

- Relatively straight-forward
- Kernel configuration:
  - Memory Technology Device (MTD) support
  - MTD partitioning/contactenating support, if needed
  - Direct char access to MTD devices
  - Caching block device access to MTD devices
  - “RAM/ROM/...” submenu: Detect flash chip by Common Flash Interface (CFI) and Support for Intel/Sharp and AMD/Fujitsu flash chips.
  - “Mapping drivers ...” submenu: select CFI mapping for board

- Partitioning can be done in mapping driver
- Newer partitioning method using boot param:

```
mtdparts=0:256k(ppcboot)ro,768k(kernel),1m(user),-(initrd);
    1:2m(cramfs),-(jffs2)
```

- For TQM860L, partition is as follows:

```
TQM flash bank 0: Using static image partition definition
Creating 4 MTD partitions on "TQM8xxL Bank 0":
0x00000000-0x00040000 : "ppcboot"
0x00040000-0x00100000 : "kernel"
0x00100000-0x00200000 : "user"
0x00200000-0x00400000 : "initrd"
TQM flash bank 1: Using static file system partition definition
Creating 2 MTD partitions on "TQM8xxL Bank 1":
0x00000000-0x00200000 : "cramfs"
0x00200000-0x00400000 : "jffs2"
```

- Partitions are on erase block boundaries (128KB)

- Creation of necessary /dev entries:

```
$ cd ${PRJROOT}/rootfs/dev
$ su -m
Password:
# for i in $(seq 0 5)
> do
> mknod mtd$i c 90 $(expr $i + $i)
> mknod mtdblock$i b 31 $i
> done
# exit
$ ls -al mtd*
crw-rw-r-- 1 root root 90, 0 Jun 25 17:19 mtd0
crw-rw-r-- 1 root root 90, 2 Jun 25 17:20 mtd1
crw-rw-r-- 1 root root 90, 4 Jun 25 17:20 mtd2
crw-rw-r-- 1 root root 90, 6 Jun 25 17:20 mtd3
crw-rw-r-- 1 root root 90, 8 Jun 25 17:20 mtd4
crw-rw-r-- 1 root root 90,10 Jun 25 17:20 mtd5
brw-rw-r-- 1 root root 31, 0 Jun 25 17:17 mtdblock0
brw-rw-r-- 1 root root 31, 1 Jun 25 17:17 mtdblock1
brw-rw-r-- 1 root root 31, 2 Jun 25 17:17 mtdblock2
...
```

- Must erase flash before writing:

```
# flash_eraseall /dev/mtd3
Erased 2048 Kibyte @ 0 -- 100% complete.
```

- Writing:

```
# cat /tmp/initrd.bin > /dev/mtd3
```

- Reading:

```
# dd if=/dev/mtd0 of=/tmp/u-boot.img
```

4. Disk devices

- There is a lot of documentation on how to manipulate disk devices in Linux.
- Main tool: `fdisk`

```
# fdisk
Command (m for help): m
Command action
   a   toggle a bootable flag
   b   edit bsd disklabel
   c   toggle the dos compatibility flag
   d   delete a partition
   l   list known partition types
   m   print this menu
   n   add a new partition
   o   create a new empty DOS partition table
   p   print the partition table
   q   quit without saving changes
   s   create a new empty Sun disklabel
   t   change a partition's system id
   u   change display/entry units
   v   verify the partition table
   w   write table to disk and exit
   x   extra functionality (experts only)
```

- Once partitions are created, use the `mkfs` utility described earlier to create a filesystem type on the device and use `mount` to mount new filesystem.
- If need be, use LILO as described earlier to install bootloader on secondary hard-disk.

5. Swapping

- Discouraged for all types of embedded Linux systems, except large ones.
- Strongly discourage for any system relying on solid-state storage.
- Use `swapon` to enable swapping on a device and `swappoff` to disable swapping on a device.

Choosing and installing the rootfs

1. Selecting a filesystem
2. Writing images to flash using NFS
3. CRAMFS
4. ROMFS
5. Squashfs
6. UBIFS
7. JFFS2
8. Disk filesystem over RAM disk
9. Comparing filesystem image size
10. TMPFS
11. Live updates

1. Selecting a filesystem

- Important part of system design
- Requires understanding of filesystem capabilities and system's dynamics.
- Filesystem characteristics:
  - Write
  - Persistent
  - Power down reliability
  - Compression
  - Lives in RAM

- Selecting a filesystem:
  - RAM disk:
    - Large amount of RAM
    - Limited amount of flash
    - Don't need to keep changes across reboots
    - For persistence, use with additional filesystem
    - Easiest way to get self-hosting target
    - Replaced by initramfs
  - CRAMFS
    - Compressed ROM filesystem
    - Requires less RAM than a RAM disk
    - Cannot be written to (read-only)

- Maximum file-size of 16 MB
- No current or parent directories
- 16-bit UID / 8-bit GID
- All timestamps set to Epoch
- All files have link count of 1
- Requires block device

- ROMFS
  - ... read-only
  - Requires block device
  - Stores the bare-minimum details required for a filesystem
  - Does not store: dates, permissions
  - Used to be used by uClinux since no protection anyway
  - No compression

- Squashfs:
  - Compressed ROM filesystem, better than CRAMFS
  - Requires block device
- UBIFS:
  - JFFS2-replacement
  - All benefits of JFFS2 w/ better performance and scalability
  - Benefits most apparent on large images
- JFFS2
  - Compressed
  - Read / write
  - Perfect for devices that need field-updating
  - Power-down reliability (no roll-back)
  - Wear-leveling

- Using a regular disk device:
  - Have a look at the slew of disk filesystems supported by Linux. BELS contains a few pointers if you're out of inspiration.

2. Writing images to flash using NFS

- May not have access to flash device from host, but may need to put image generated on host in target flash.
- Steps:
  1. Generate image on host
  2. Cross-build MTD utilities for target and install in ${PRJROOT}/rootfs
  3. Mount target's rootfs using NFS
  4. On host, copy image to directory mounted by target as rootfs.
  5. Use MTD utilities to copy image from rootfs to appropriate /dev entry.

3. CRAMFS

- Latest available from: http://sourceforge.net/projects/cramfs/
- Download CRAMFS (1.1) utilities into your ${PRJROOT}/build-tools directory and uncompress there.
- Build and install CRAMFS utilities:

```
$ cd ${PRJROOT}/build-tools/cramfs-1.1
$ make
$ cp cramfsck mkcramfs ${PREFIX}/bin/
```

- Creating a rootfs image:

```
$ cd ${PRJROOT}
$ mkcramfs rootfs/ images/cramfs.img
Directory data: 3960 bytes
Everything: 3452 kilobytes
Super block: 76 bytes
CRC: f74fb18c
warning: gids truncated to 8 bits (this may be a security concern)
```

- Example filesystem image writing to device, if MTD device accessible on host:

```
$ su -m
Password:
# cat images/cramfs.img > /dev/mtd5
# exit
```

- Use NFS-writing if device not accessible from host

4. ROMFS

- Latest available from: http://romfs.sourceforge.net/
- Download genromfs (0.5.2) utilities into your ${PRJROOT}/build-tools directory and uncompress there.
- Build and install genromfs utilities:

```
$ cd ${PRJROOT}/build-tools/genromfs-0.5.2
$ make
$ cp genromfs ${PREFIX}/bin
```

- Creating a ROMFS image:

```
$ cd ${PRJROOT}
$ su -m
# genromfs -d rootfs/ -f images/romfs.img
# chown karim:karim images/romfs.img
```

5. Squashfs

- Latest available from: http://www.squashfs.org
- Download squashfs (4.1) into your ${PRJROOT}/build-tools directory and uncompress there.
- Make sure you have libacl1-dev installed:

```
$ sudo apt-get install libacl1-dev
```

- Build and install squashfs utilities:

```
$ cd ${PRJROOT}/build-tools/squashfs4.1/squashfs_tools
$ make
$ cp mksquashfs unsquashfs ${PREFIX}/bin
```

- Creating a SquashFS image:

```
$ cd ${PRJROOT}
$ mksquashfs rootfs/ images/squashfs.img
```

6. UBIFS

- Tools already installed as part of MTD utilities
- Creating a UBIFS image:

```
$ cd ${PRJROOT}
$ mkfs.ubifs -m 512 -e 128KiB -c 100 -x zlib \
> -r rootfs images/ubifs.img
```

7. JFFS2

- Tools already installed as part of MTD utilities.
- Creating a JFFS2 image:

```
$ cd ${PRJROOT}
$ mkfs.jffs2 -r rootfs/ -o images/jffs2.img
```

- Can't mount on loopback
- May use `jffs2dump`

8. Disk filesystem over RAM disk

- RAM disks behave like block devices
- The RAM disk driver can manage multiple RAM disk instances.
- initrd is mechanism to provide kernel with initial RAM disk for use as rootfs at startup.
- Creating blank filesystem image:

```
$ cd ${PRJROOT}
$ mkdir tmp/initrd
$ dd if=/dev/zero of=images/initrd.img bs=1k count=8192
8192+0 records in
8192+0 records out
8388608 bytes (8.4 MB) copied, 0.0425007 s, 197 MB/s
```

- Formating filesystem image and mounting it:

```
$ su -m
Password:
# /sbin/mke2fs -F -v -m0 images/initrd.img
mke2fs 1.41.12 (17-May-2010)
fs_types for mke2fs.conf resolution: 'ext2', 'small'
Calling BLKDISCARD from 0 to 8388608 failed.
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
Stride=0 blocks, Stripe width=0 blocks
2048 inodes, 8192 blocks
0 blocks (0.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=8388608
1 block group
8192 blocks per group, 8192 fragments per group
2048 inodes per group
...

# mount -o loop images/initrd.img tmp/initrd
```

- Copying rootfs, umounting, and compressing:

```
# cp -av rootfs/* tmp/initrd
`rootfs/bin' -> `tmp/initrd/bin'
`rootfs/bin/busybox' -> `tmp/initrd/bin/busybox'
`rootfs/bin/ash' -> `tmp/initrd/bin/ash'
`rootfs/bin/cat' -> `tmp/initrd/bin/cat'
`rootfs/bin/chgrp' -> `tmp/initrd/bin/chgrp'
`rootfs/bin/chmod' -> `tmp/initrd/bin/chmod'
...
# umount tmp/initrd
# exit
$ gzip -9 < images/initrd.img > images/initrd.bin
```

- The initrd mechanism is largely favored to be replaced by initramfs.

9. Disk filesystem over initramfs

- Initramfs used by kernel at boot time
- Can be linked directly into kernel image
- Creating an initramfs image:

```
$ cd ${PRJROOT}/rootfs
$ find . | cpio -o -H newc | gzip > ../images/initramfs.img
```

10. Comparing filesystem image sizes

- Let's take a look at what we've generate so far:

```
$ cd ${PRJROOT}/images
$ ls -al \
> cramfs.img romfs.img squashfs.img ubifs.img jffs2.img init*
-rw-r--r-- 1 stage stage 3719168 2011-02-25 14:49 cramfs.img
-rw-r--r-- 1 stage stage 3492683 2011-02-25 11:28 initramfs.img
-rw-r--r-- 1 stage stage 8388608 2011-02-25 15:19 initrd.img
-rw-r--r-- 1 stage stage 4237296 2011-02-25 15:13 jffs2.img
-rw-r--r-- 1 stage stage 6949888 2011-02-25 14:50 romfs.img
-rw-r--r-- 1 stage stage 3264512 2011-02-25 14:58 squashfs.img
-rw-r--r-- 1 stage stage 5898240 2011-02-25 15:13 ubifs.img
```

11. Mounting directories on TMPFS

- Dynamically resizeable RAM-based FS
- Perfect for holding transient data
- No need to create beforehand or format
- Mount:

```
# mount -t tmpfs none /tmp -o size=4m
```

12. Live updates

- Only RAM disks can be replaced in its entirety while mounted.
- CRAMFS, ROMFS, Squashfs cannot be replaced
- All rootfses based on other filesystems must be updated while mounted by modifying files and directories one-by-one.
- Four main ways:
  - `rsync`
  - Package management tools
  - `netflash`
  - Ad hoc scripts

- The `rsync` utility:
  - Replaces Unix's `rcp`
  - Allows network update
  - Can be used over SSH
  - Available from: http://samba.anu.edu.au/rsync/
  - Need to have daemon running on host and client running in the embedded system.
  - `rsync` most likely already installed by distro.
  - Download (2.5.6) and extract package in your ${PRJROOT}/sysapps directory.

- Configure compile and install `rsync`:

```
$ cd ${PRJROOT}/sysapps/rsync-2.5.6/
$ CC=powerpc-linux-gcc \
> CPPFLAGS="-DHAVE_GETTIMEOFDAY_TZ=1" ./configure \
> --host=$TARGET --prefix=${TARGET_PREFIX}
$ make
$ cp rsync ${PRJROOT}/rootfs/bin
$ powerpc-linux-strip ${PRJROOT}/rootfs/bin/rsync
```

- Use on target:

```
# rsync -e "ssh -l root" -r -l -p -t -D -v \
> --progress \
> 192.168.172.100:/home/karim/example-sys/rootfs/* /
```

- To do a dry run, add “-n” parameter

- Package management tools:
  - dpkg
  - RPM
  - BusyBox dpkg
- The `netflash` utility:
  - Part of uClinux-dist

Kernel internals

1. Brief history
2. Features
3. General architecture
4. Source layout
5. Process management
6. Filesystems
7. Memory management
8. Communication facilities and interfacing
9. Loadable modules

10. Interrupt and exception management
11. Timing
12. Locking primitives
13. Kernel startup
14. Dealing with kernel failure

1. Brief history

- Started on Minix by Linus Torvalds (who wanted to call is Freax ...)
- Rechristened “Linux” by FTP site admin
- Once public, users and contributions grew at a very rapid rate.
- Closed the GNU project's loop
- Most important events:
  - 17 September 1991: Version 0.01
  - 3 December 1991: Version 0.10
  - 8 March 1992: Version 0.95

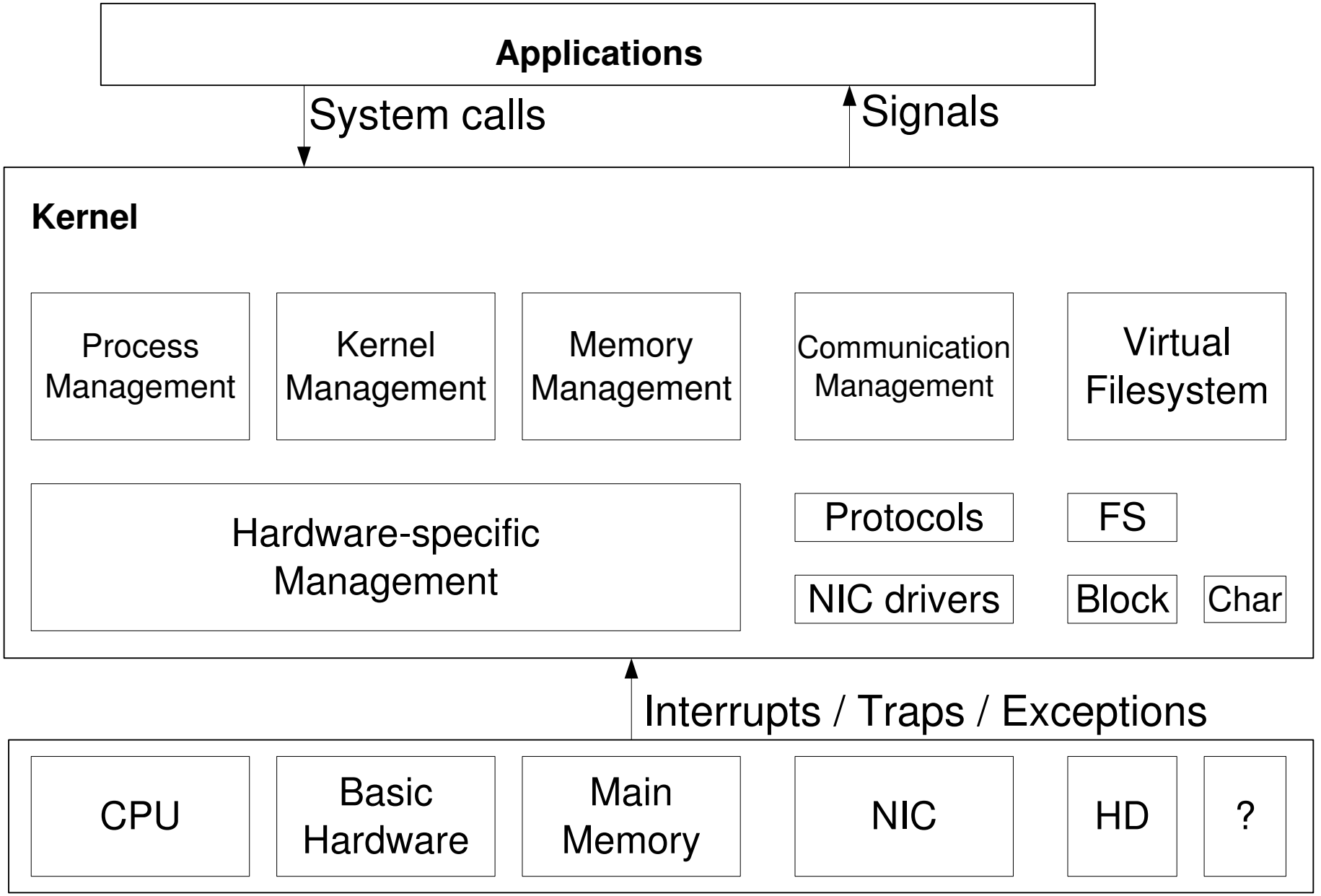
- 13 December 1992: Version 0.99
- 13 March 1994: Version 1.0
- 7 March 1995: Version 1.2.0
- 9 July 1996: Version 2.0
- 26 January 1999: Version 2.2.0
- 4 January 2001 : Version 2.4.0
- 17 December 2003: Version 2.6.0

2. Features

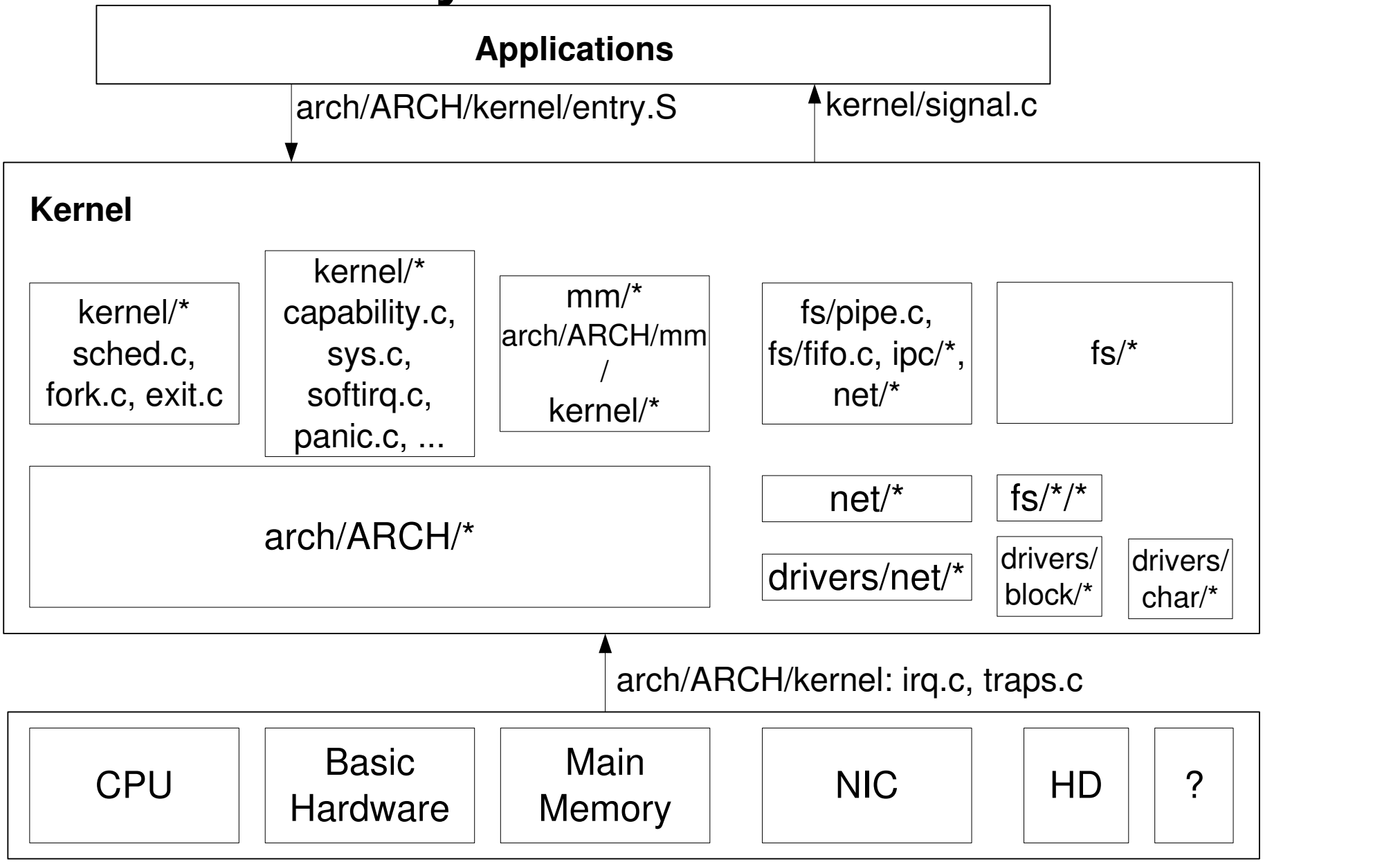
- Portable / Architecture-independent
- Scalable
- Monolithic
- Dynamically extensible (modules)
- Multi-user environment
- Multi-process / Multi-threading
- Memory protection
- Preemptable (starting in 2.5.x) ... but not real-time
- Symmetric multi-processor

- Slew of filesystems
- Slew of networking protocols / NICs
- Quite a few executable formats
- ...

3. General architecture



4. Source layout



arch	112MB =>	architecture-dependent functionality
block	600KB =>	block layer
Documentation	17MB =>	main kernel documentation
drivers	231MB =>	all drivers
fs	31MB =>	virtual filesystem and all fs types
include	20MB =>	complete kernel headers
init	150KB =>	kernel startup code
ipc	224KB =>	System V IPC
kernel	**4.7MB =>**	**core kernel code**
mm	2.2MB =>	memory management
net	20MB =>	networking core and protocols
scripts	1.1MB =>	scripts used to build kernel
tools	2.1MB =>	misc. kernel-related tools

- arch/

```
2.4M  alpha
29M   arm
1.4M  avr32
5.3M  blackfin
4.9M  cris
1.4M  frv
856K  h8300
4.6M  ia64
8.0K  Kconfig
1.4M  m32r
5.7M  m68k
1.1M  m68knommu
1.2M  microblaze
11M   mips
1.7M  mn10300
2.4M  parisc
13M   powerpc
2.4M  s390
636K  score
5.4M  sh
4.7M  sparc
1.9M  tile
1.9M  um
8.5M  x86
1.4M  xtensa
```

- arch/powerpc:

```
2.0M boot
548K configs
2.0M include
2.1M kernel
384K kvm
216K lib
204K math-emu
472K mm
156K oprofile
3.4M platforms
848K sysdev
392K xmon
```

- drivers/

```
accessibility  cpufreq   hwmon        mca       parisc     sbus       uio
acpi           cpuidle   i2c          md        parport    scsi       usb
amba           crypto    ide          media     pci        serial     uwb
ata            dca       idle         memstick  pcmcia     sfi        vhost
atm            dio       ieee802154   message   platform   sh         video
auxdisplay     dma       infiniband   mfd       pnp        sn         virtio
base           edac      input        misc      power      spi        vlynq
block          eisa      isdn         mmc       pps        ssb        w1
bluetooth      firewire  Kconfig      mtd       ps3        staging    watchdog
cdrom          firmware  leds         net       rapidio    tc         xen
char           gpio      lguest       nubus     regulator  telephony  zorro
clocksource    gpu       macintosh    of        rtc        thermal
connector      hid       Makefile     oprofile  s390       tty
```

- include/

```
acpi          config  drm     keys    math-emu  mtd  pcmcia  rxrpc  sound  video
asm-generic   crypto  Kbuild  linux   media     net  rdma    scsi   trace  xen
```

- Looking for something:
  - Try `grep`
  - Have a look at the Linux Cross-Referencing project:
    - URL: http://lxr.linux.no/
    - Code: http://lxr.sourceforge.net/
  - Advanced kernel searching/understanding:
    - CScope: http://cscope.sourceforge.net/
    - KScope front-end: http://kscope.sourceforge.net/
  - ETAGS (emacs)

5. Process management

- Process descriptor (include/linux/sched.h: task_struct):

Process state	=>	*state*
Identification	=>	*pid*, t*gid*
Relationship	=>	**parent*, *children*
Scheduling	=>	*time_slice*, sched_class, *rt_priority, prio, static_prio*
Files	=>	**files*
Memory	=>	**mm*

- Main process list:
  - Doubly-linked list
  - List head: *init_task*
- Currently running process (macro): *current*
- Task creation:

  kernel/fork.c:*do_fork()*
- Threads:

  arch/ARCH/kernel/process.c:*sys_clone()*
- *sys_clone()* ends up calling *do_fork()*

- Scheduling:
  - Process states: TASK_RUNNING, TASK_INTERRUPTIBLE, TASK_UNINTERRUPTIBLE, TASK_STOPPED, TASK_TRACED, EXIT_ZOMBIE, ...
  - Main scheduling function:

    kernel/sched.c: *schedule()*
  - Scheduling policies:
    - SCHED_NORMAL
      - Main scheduling policy for Linux processes
    - SCHED_FIFO:
      - Process has CPU until it gives it up or no other higher priority task comes along.
    - SCHED_RR:
      - CPU is shared between “real-time” tasks
    - ...

- Address space:
  - include/linux/mm_types.h:*mm_struct*
  - Fields: **mmap*, *mm_count*, *start_code*, *end_code*, *start_data*, *end_data*, *start_brk*, *brk*, *start_stack*, *arg_start*, *arg_end*, *env_start*, *env_end*
- Special tasks:
  - init => first process on system
  - ksoftirqd/0 => soft-irq thread for avoiding process starvation
  - events/0 => kernel's work queue handler (was keventd)
  - khubd => USB hub thread
  - kswapd0 => pageout daemon

6. Filesystems

- Virtual filesystem: fs/*

attr.c	=>	file attributes
block_dev.c	=>	block device access
buffer.c	=>	buffer cache
char_dev.c	=>	char device access
dcache.c	=>	dentry (directory entry) cache
notify/	=>	directory change notifications
quot/	=>	disk quota
exec.c	=>	*exec()* and its variants
fcntl.c	=>	*fcntl()*
fifo.c	=>	FIFO handling
file.c	=>	manage process' fd array
file_table.c	=>	file table manipulation
inode.c	=>	inode handling
ioctl.c	=>	*ioctl()*
locks.c	=>	file locking

namei.c	=>	pathname lookup
namespace.c	=>	filesystem mounting
open.c	=>	*open()*
pipe.c	=>	pipe management
readdir.c	=>	directory reading
read_write.c	=>	*read()* and *write(),* and variants
select.c	=>	*select()* and *poll()*
stat.c	=>	*stat()*
super.c	=>	filesystem type management

- Root filesystem mounting: init/do_mounts.c

  prepare_namespace()
- RAM disk handling (within init/):
  - do_mounts_initrd.c: initrd_load()
  - do_mounts_rd.c: *rd_load_image(), identify_ramdisk_image()*
  - do_mounts_initrd.c: *handle_initrd(),*

7. Memory management

- Arch-independent portion: mm/*

bootmem.c	=>	boot memory allocation / handling
filemap.c	=>	handling for *mmap()*'ed files
highmem.c	=>	RAM above 896MB / up to 64 GB
memory.c	=>	page and page table manipulation
mlock.c	=>	memory region locking
mmap.c	=>	*mmap()*
mprotect.c	=>	memory protection mechanisms
mremap.c	=>	*mremap()*
nommu.c (2.5)	=>	*functions for MMU-less processors*
oom_kill.c	=>	process killing when short on memory
page_alloc.c	=>	page allocation / freeing
page_io.c	=>	reading / writing swap pages
shmem.c	=>	shared memory management
slab.c	=>	memory allocation for kernel
swap.c	=>	swap default

swapfile.c	=>	swap space management
swap_state.c	=>	swap page caching
vmalloc.c	=>	memory region allocation
vmscan.c	=>	page out daemon

- Architecture-dependent portion: arch/ARCH/mm/*

fault.c	all =>	page fault handler
init.c	all =>	memory initialization
ioremap.c	x86=>	remapping of I/O range to kernel space
pageattr.c	x86=>	page attributes handling
pgtable.c	ppc=>	page table manipulation
ppc_mmu.c	ppc=>	MMU handling for PPC
tlb.c	ppc=>	TLB flushing

8. Communication facilities and interfacing

- Signals => kernel/signal.c
- Pipes => fs/pipe.c
- FIFOs => fs/fifo.c
- Sockets => net/socket.c
- System V IPC => ipc/ : msg.c, sem.c, shm.c
- System calls => arch/ARCH/kernel/entry.S
- Adding new system calls:
  1. Add entry to arch/ARCH/kernel/entry.S
  2. Add entry to arch/ARCH/include/unistd.h
  3. Add your function to the kernel's code

9. Loadable modules

- Allow dynamic loading and unloading of additional kernel functionality.
- Managed by: kernel/module.c
- In practice, modules are .ko files (.o prior to 2.6)
- A single module's source tree can have a very complex hierarchy.
- Every module must export functions using:

module_init()	=>	called on insmod
module_exit()	=>	called on rmmod

- Module macros:

MODULE_AUTHOR	=>	module's author
MODULE_LICENSE	=>	module's license (taint)
MODULE_DESCRIPTION	=>	module's description
EXPORT_SYMBOL	=>	export symbol for use by other modules
MODULE_PARM_DESC	=>	module parameter description

...

- Requesting modules from within the kernel:

*int request_module(const char * name, ...)*

10. Interrupt and exception management

- Relevant files:
  - arch/x86/kernel/

i8259.c	=>	initialization and handling of 8259
irq.c	=>	arch-dependent irq handling
entry.S	=>	main assembly entry point
traps.c	=>	CPU exception handling

  - kernel/irq/

handle.c	=>	arch-independent irq handling (exc. ARM)

- Exception examples:
  - debug, overflow, bounds, fp, fault, nmi, etc.
- Exceptions path:
  1. arch/x86/kernel/entry.S: hard-coded assembly
  2. arch/x86/kernel/traps.c: custom function or macro generated (DO_ERROR).

- Hardware interrupt code:
  - arch/x86/kernel/entry.S: irq_entries_start
- Hardware interrupt path:
  1. Assembly in entry.S
  2. *do_IRQ()* in arch/x86/kernel/irq.c
  3. *__do_IRQ()* in kernel/irq/handle.c:
     1. Acks IRQ using callback from arch/x86/kernel/i8259.c:*mask_and_ack_8259A()*
     2. Calls *handle_IRQ_event()*
  4. Handler provided by device driver is invoked

- Deferring handling with enabled interrupts:
  - Softirq:
    - Statically allocated
    - Reentrant (must use locking mechanisms to protect data)
    - Softirqs of same type can run on many CPUs in the same time.
  - Tasklets:
    - Built on top of softirqs
    - Dynamically allocatable
    - The same tasklet type can't run on 2 CPUs in the same time.
  - Bottom-halves:
    - Built on top of tasklets
    - Statically allocated
    - There can only be one BH running at one time in the entire system.

11. Timing

- Time-keeping variables in the kernel: x86

jiffies	=>	incremented at every clock tick
wall_jiffies	=>	last time xtime was updated
TSC	=>	CPU-maintained counter (64-bit)

- Time handling:
  - arch/x86/kernel/time.c
- Finding out what time it is:
  - *do_gettimeofday()*
  - *get_cycles()*
  - *jiffies*

12. Locking primitives

- Never do cli / sti
- Always use existing locks to do the dirty-work
- Variants:
  - Spinlocks with IRQ disabling: good for int handlers

    spin_lock_irqsave(&lock_var, cpu_flags);

    ....

    spin_unlock_irqrestore(&lock_var, cpu_flags);
  - Vanilla spinlocks: good for most code not in int

    spin_lock(&lock_var);

    ....

    spin_unlock(&lock_var);

- Read/write locks:
  - In reader:

    read_lock_irqsave(&lock_var, cpu_flags);

    ....

    read_unlock_irqrestore(&lock_var, cpu_flags);
  - In writer:

    write_lock_irqsave(&lock_var, cpu_flags);

    ....

    write_unlock_irqrestore(&lock_var, cpu_flags);
- For a complete list, have a look at include/linux/spinlock.h.

13. Kernel startup

Explanation for TQM860 PPC board

0.Kernel entry point:

arch/ppc/boot/common/crt0.S:*_start*

1. *_start* calls on:

arch/ppc/boot/simple/head.S:*start*

2. *start* calls on:

arch/ppc/boot/simple/relocate.S:*relocate*

3. *relocate* calls on:

arch/ppc/boot/simple/misc-embedded.c: *load_kernel()*

4. *load_kernel()* initializes the serial line and uncompresses kernel starting at address 0.

6. *relocate* jumps to address 0x00000000, where kernel start address is.
7. arch/ppc/kernel/head_8xx.S: *__start*
8. *__start* eventually calls init/main.c:*start_kernel()*
9. *load_kernel()* returns to *relocate*
10. *start_kernel()* does:
    1. Locks kernel
    2. *setup_arch()*
    3. *sched_init()*
    4. *parse_args()*
    5. *trap_init()*
    6. *init_IRQ()*

7. *time_init()*
8. *console_init()*
9. *mem_init()*
10. *calibrate_delay()* => *loops_per_jiffy*
11. *rest_init()*

11. *rest_init()* does:
    1. Start `init` thread
    2. Unlocks the kernel
    3. Becomes the idle task

12. The `init` task:

1. *lock_kernel()*
2. *do_basic_setup()* => call various *init()* fcts
3. *prepare_namespace()* => mount rootfs
4. *free_initmem()*
5. *unlock_kernel()*
6. *execve()* on the init program (`/sbin/init`)

14. Dealing with kernel failure

- Kernel is mature and stable, but can fail
- Kernel failure results in call to kernel/panic.c:*panic()*
- Reboot is as according to panic= boot param or default (180s)
- Default panic message goes to console
- Need to have your own panic handler
- Main list of callbacks on panic: *panic_notifier_list*.
- Register using *notifier_chain_register()*

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

static int on_screen_panic_event(struct notifier_block *,
                                 unsigned long,
                                 void *);

static struct notifier_block on_screen_panic_block = {
        notifier_call:   on_screen_panic_event,
        next:            NULL,
        priority:        INT_MAX
};

int __init register_on_screen_panic(void)
{
        printk("Registering on-screen panic notifier \n");

        notifier_chain_register(&panic_notifier_list,
                                &on_screen_panic_block);

        return 0;
}
```

```
void write_raw_user_screen(void)
{
          ...
}

static int on_screen_panic_event(struct notifier_block
   *this,
                                  unsigned long event,
                                  void *ptr)
{
        write_raw_user_screen(  );

        return NOTIFY_DONE;
}

__initcall(register_on_screen_panic);
```

Linux hardware support

- See kernel sources ...

Device driver overview

1. Licensing reminder
2. Device driver model
3. Writing a char device driver
4. Writing a block device driver
5. Writing a network device driver
6. Writing an MTD map file
7. Writing a framebuffer driver
8. Time-keeping
9. Memory needs

10. Hardware access
11. Interrupt handling
12. Printing out messages to console

1. Licensing reminder

- Although the use of binary-only modules is widespread, Kernel modules are not immune to kernel GPL.
- Many kernel developers have come out rather strongly against binary-only modules.
- If you are linking a driver as built-in, then you are most certainly forbidden from distributing the resulting kernel under any license other than the GPL.
- If you're wary of the GPL, push critical driver-intelligence to user-space.

2. Device driver model

- Device files
  - Everything is a file in Unix, including devices
  - All devices are located in the /dev directory
  - Only networking devices do not have /dev nodes
  - Every device is identified by major / minor number
  - Can be allocated statically (devices.txt)
  - Can be allocated dynamically
  - To see devices present: `$ cat /proc/devices`
  - Alternatives: devfs, sysfs (used to be driverfs)

- Char devices:
  - Stream-oriented devices
  - Manipulated using: *struct file_operations**
- Block devices:
  - Disk-oriented devices
  - Manipulated using: *struct block_device_operations**
- Networking devices
  - All networking devices
  - Manipulated using: *struct net_device**

- Subsystem drivers:
  - USB:

    *struct usb_driver **
  - MTD:

    Chip driver: *struct mtd_chip_driver **

    Device: *struct mtd_info **
  - Framebuffer:

    *struct fb_info **

3. Writing a char device driver

- Register char dev during module initialization
- Char dev registration: include/linux/fs.h

```
int register_chrdev(unsigned int,
                    const char *,
                    struct file_operations *);
```

- First param: Major number
- Second param: Device name (as displayed in /proc/devices)
- Third param: File-ops
  - Defined in include/linux/fs.h
  - Contains callbacks for all possible operations on a char device.

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (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 *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned ...
        int (*check_flags)(int);
        int (*dir_notify)(struct file *filp, unsigned long arg);
        int (*flock) (struct file *, int, struct file_lock *);
};
```

- Call *register_chrdev()* and pass it a valid *file_operations* structure.
- Return 0 from initialization function to tell insmod that everything is OK.
- That's it. Every time the device in /dev having the same major number as the one you registered is opened, you driver will be called.
- To remove char dev on rmmod:

```
int unregister_chrdev(unsigned int,
                      const char *);
```

4. Writing a block device driver

- Register block dev during module initialization
- Block dev registration: include/linux/fs.h

```
int register_blkdev(unsigned int,
                    const char *);
```

- First param: Major number
- Second param: Device name
- Disk allocation: include/linux/genhd.h

```
struct gendisk *alloc_disk(int minors);
```

- Block queue registration: include/linux/blkdev.h

```
extern void blk_init_queue(request_fn_proc *,
                   spinlock_t *);
```

- Queue of pending I/O operations for device

- First param: Queue handler function
- Second param: Lock for accessing queue
- Call *register_blkdev().*
- Call *alloc_disk()* and pass it the number of disks.
- Call *blk_init_queue()* and pass it a valid callback.
- Return 0 from init function to tell insmod status

- Now, all block operations on your device (/dev entry with same major number as driver) will be queued to your driver.
- To remove block dev on rmmod:

```
void blk_cleanup_queue(request_queue_t *);
void put_disk(struct gendisk *disk);
int unregister_blkdev(unsigned int, const char *);
```

5. Writing a network device driver

- Register net dev during module initialization
- Net dev registration: include/linux/netdevice.h

```
int     register_netdevice(struct net_device *dev);
```

- Param: net device ops
  - Defined in include/linux/netdevice.h
  - Contains all callbacks related to network devices
  - This is a huge structure with A LOT of fields
- Call *register_netdevice()* and pass it a valid *net_device* structure.
- Return 0 as status to insmod

- Your device will need to be *opened* by the kernel in response to an *ifconfig* command.
- Your *open()* function must allocate a packet queue to deal with packets sent to your device.
- Calling your device will depend on packet routing at the upper layers of the stack.
- To remove: `unregister_netdev(struct net_device *dev);`

6. Writing an MTD map file

- Must find device in memory and then register it
- Finding a device in memory: include/linux/mtd/map.h

```
struct mtd_info *do_map_probe(char *name,
                              struct map_info *map);
```

- First param: Type of probe (ex: “cfi_probe”)
- Second param: map-info
  - Defined in include/linux/mtd/map.h
  - Information regarding size and buswidth
  - Functions for accessing chip

```
struct map_info {
char *name;
unsigned long size;
unsigned long phys;
#define NO_XIP (-1UL)

void __iomem *virt;
void *cached;

int bankwidth;
#ifdef CONFIG_MTD_COMPLEX_MAPPINGS
map_word (*read)(struct map_info *, unsigned long);
void (*copy_from)(struct map_info *, void *, unsigned long, ssize_t);

void (*write)(struct map_info *, const map_word, unsigned long);
void (*copy_to)(struct map_info *, unsigned long, const void *, ssize_t);
#endif
void (*inval_cache)(struct map_info *, unsigned long, ssize_t);

/* set_vpp() must handle being reentered -- enable, enable, disable
   must leave it enabled. */
void (*set_vpp)(struct map_info *, int);

unsigned long map_priv_1;
unsigned long map_priv_2;
void *fldrv_priv;
struct mtd_chip_driver *fldrv;
};
```

- Once located, use *add_mtd_partitions()* to provide partition information to MTD subsystem.

- *add_mtd_partition()* is in include/linux/mtd/partitions.h

```
int add_mtd_partitions(struct mtd_info *,
                       struct mtd_partition *,
                       int);
```

- First param: pointer returned by *do_map_probe()*
- Second param: partition information as we saw earlier.
- Third param: number of partitions

7. Writing a framebuffer driver

- Register framebuffer during module init
- Framebuffer registration: include/linux/fb.h

```
int register_framebuffer(struct fb_info *fb_info);
```

- Param: fb-info
  - Defined in include/linux/fb.h
  - Contains callbacks for all framebuffer operations

```
struct fb_info {
        int node;
        int flags;
        struct fb_var_screeninfo var; /* Current var */
        struct fb_fix_screeninfo fix; /* Current fix */
        struct fb_monspecs monspecs;  /* Current Monitor specs */
        struct work_struct queue;     /* Framebuffer event queue */
        struct fb_pixmap pixmap; /* Image hardware mapper */
        struct fb_pixmap sprite; /* Cursor hardware mapper */
        struct fb_cmap cmap;          /* Current cmap */
        struct list_head modelist;      /* mode list */
        struct fb_ops *fbops;
        struct device *device;
#ifdef CONFIG_FB_TILEBLITTING
        struct fb_tile_ops *tileops;    /* Tile Blitting */
#endif
        char __iomem *screen_base;    /* Virtual address */
        unsigned long screen_size;    /* Amount of ioremapped VRAM or 0 */
        void *pseudo_palette;         /* Fake palette of 16 colors */
#define FBINFO_STATE_RUNNING    0
#define FBINFO_STATE_SUSPENDED  1
        u32 state;              /* Hardware state i.e suspend */
        void *fbcon_par;                /* fbcon use-only private area */
        /* From here on everything is device dependent */
        void *par;
};
```

- Call *register_framebuffer()* and pass it a valid *fb_info* structure.
- Return 0 from init code
- Access the /dev/fbX (where X is your framebuffer's registration order in relationship to other fb drivers) results in the functions provided in *fb_info* to be called.
- To remove fb dev on rmmod:

```
int unregister_framebuffer(struct fb_info *fb_info);
```

8. Time-keeping

- Simple ways to get the time
  - *jiffies*: updated at every kernel tick
  - *do_gettimeofday()*: good precision depending on arch
  - *get_cycles()*: arch-independent call to get CPU cycle count.
- Being notified in due time:
  - Use timers: include/linux/timer.h
    - *jiffies* resolution
    - Use *struct timer_list*: contains expiry and callback
    - Initialize timer: *init_timer()*
    - Add timer to global timer list: *add_timer()*
    - Remove timer before expiry: *del_timer()*

9. Memory needs

- Main kernel memory functions:
  - *kmalloc(size, type)*: up to 128KB of memory
  - *vmalloc(size)*: get large contiguous virtual memory
- Main types of memory allocation:
  - GFP_KERNEL => normal alloc / may sleep
  - GFP_ATOMIC => int handlers / never sleeps
  - GFP_USER => user-space / low priority
  - … See include/linux/slab.h for full list

10. Hardware access

- Main functions to be able to access hardware:
  - ISA:
    - Typical inb/outb
  - PCI:
    - Checking use of mem region: *check_mem_region()*
    - Requesting region: *request_mem_region()*
    - Releasing region: *release_mem_region()*
    - Map physical region to VM: *ioremap()*

11. Interrupt handling

- Setting up an interrupt handler:

```
extern int request_irq(unsigned int,
        irq_handler_t, unsigned long, const char *, void *);
```

- First param: IRQ number
- Second param: handler
- Third param: flags for OS int delivery
- Four param: device name
- Fifth param: provide device ID in case of shared interrupts.
- Disabling interrupt handler:

```
void free_irq(unsigned int, void *);
```

12. Printing out messages to console

- Meet the kernel's printf: *printk()*
- Defined: include/linux/kernel.h

```
int printk(const char * fmt, ...)
```

- Implemented: kernel/printk.c
- Can loose data in cases of large output
- Widely-used throughout kernel sources
- Don't call while holding lock, has lock contention of its own.

Kernel debugging primer

1. Manual techniques
2. Debugging tools
3. Performance measurement
4. Hardware tools

1. Manual techniques

- printk

```
printk(“Detected error 0x%x on interface %d:%d\n”,
       error_code, iface_bus, iface_id);
```

- /proc
  - Main functions: include/linux/proc_fs.h

```
struct proc_dir_entry
    *create_proc_read_entry(const char *name,
                            mode_t mode,
                            struct proc_dir_entry *base,
                            read_proc_t *read_proc,
                            void * data)
void remove_proc_entry(const char *name,
                       struct proc_dir_entry *parent)
```

  - *read_proc* is a callback:

```
typedef int (read_proc_t)(char *page, char **start,
                          off_t off, int count,
                          int *eof, void *data);
```

- Write data to *page*
- Careful: can only fill 1 page at a time (4K)
- Use **start* to tell OS that you have more than a page. Your function will then be called more than once with a different *offset*.
- ALWAYS return the size you wrote. If you don't, nothing will be displayed.
- You can add your own /proc tree if you want ...

- ioctl:
  - Method implemented in most device driver models (char, block, net, fb, etc.) allowing custom functionality to be coded in driver ...

- Can extend your driver's ioctl to allow a user-space application to poll or change the driver's state outside of the OS' control.

- oops messages:
  - Report printed out by kernel regarding internal error that can't be handled.
  - Sometimes last output before system freeze => must be copied by hand.
  - Contains addresses and references to function addresses which can be understood by looking at System.map
  - Can be automatically decoded with klogd/ksymoops

```
Unable to handle kernel paging request at virtual address 0007007a
  printing eip:
 c022a8f6
 *pde = 00000000
 Oops: 0000
 CPU:    0
 EIP:    0010:[<c022a8f6>]    Not tainted
 EFLAGS: 00010202
 eax: 0000000a   ebx: 00000004   ecx: 00000001   edx: e3a74b80
 esi: 0007007a   edi: e62150fc   ebp: 0007007a   esp: dfbdbc8c
 ds: 0018   es: 0018   ss: 0018
 Process ip (pid: 2128, stackpage=dfbdb000)
 Stack: 00000000 bfff0018 00000018 0000000a 00000000 e41e8c00 c02f5acd e3a74b80
        c022aec1 e3a74b80 0000000a 00000004 0007007a c02f5ac8 00000e94 00000246
        e62150b4 00000000 e41e8c00 00000001 00000000 e5365a00 c022b039 e3a74b80
 Call Trace:    [<c022aec1>] [<c022b039>] [<c022df21>] [<c022e1d0>] [<c022b6ea>]
   [<c022afb0>] [<c022b1d0>] [<c0176f7e>] [<c022b2a0>] [<c022ddba>] [<c022d623>]
   [<c022db41>] [<c021c8f5>] [<c021daf3>] [<c0118238>] [<c021d44d>] [<c021e459>]
   [<c0107800>] [<c010770f>]

 Code: f3 a5 f6 c3 02 74 02 66 a5 f6 c3 01 74 01 a4 8b 5c 24 10 8b
```

- For good measure, always save the content of /proc/ksyms before generated an oops:

```
$ cat /proc/kallsyms > /tmp/kallsyms-dump
$ sync
```

2. Debugging tools

- gdb: all archs
  - Can use standard gdb to visualize kernel variables:

    ```
    $ gdb ./vmlinux /proc/kcore
    ```
  - Get more information when using “-g” flag
  - gdb grabs kcore snapshot at startup / no dynamic update
- kgdb: Full serial-line-based kernel debugger
  - Available from http://kgdb.sourceforge.net/
  - Merged into mainline since 2.6.26:

    arch/ARCH/kernel/kgdb.c
  - Connect to remote target through gdb on host
  - Use gdb as you would for any other remote program

- Check the “Kernel hacking” kernel config submenu
- kprobes:
  - Dynamic injection of probe points
  - Check out SystemTap: sourceware.org/systemtap
- kdb: x86 / ia64
  - Available from http://oss.sgi.com/
  - Integrated kernel debugger patch
  - Requires kernel patch (Linus allergic to kdb & friends)
  - Is used directly on the host being debugged
  - Access through PAUSE/BREAK key on keyboard

3. Performance measurement

- byte-unixbench:
  - Available from https://code.google.com/p/byte-unixbench/
  - Runs heavy user-space benchmarks to determine kernel response time: Dhrystone2, double-precision Whetstone, Execl throughput, file copy, pipe ...
- perf:
  - Collects both hardware and software events

```
cpu-cycles OR cycles                    [Hardware event]
instructions                            [Hardware event]
cache-references                        [Hardware event]
cache-misses                            [Hardware event]
...
page-faults OR faults                   [Software event]
minor-faults                            [Software event]
major-faults                            [Software event]
```

- Oprofile:
  - http://oprofile.sourceforge.net
- Integrated sample-based profiler:
  - Activated upon passing “profile=” boot param
  - Profile data available in /proc/profile
  - User-space tools available from http://sourceforge.net/projects/minilop/
- Measuring interrupt latency:
  - Self-contained
  - Induced

- Self-contained:
  - System's output connected to system input
  - Write driver with 2 main functions
  - Fire-up function:
    - Records current time
    - Trigger interrupt (write to output pin)
  - Interrupt handling function:
    - Records current time
    - Toggles output pin
  - Interrupt latency is measured using time difference
  - Can connect scope and observe timing
  - No need for time recording if using scope
  - Dead angle: can't see the time it takes to get to fire-up fct

- Induced:
  - Closer to reality
  - Interrupt generated by outside source (frequency generator)
  - Driver toggles output pin
  - Measure time between wave starts to get real latency
- Linux is not an RTOS ...
  - Try `ls -R /`

4. Hardware tools

- Multimeter
- Oscilloscope
- Logic analyzer
- In-Circuit Emulator
- JTAG/BDM
  - Abatron / uses plain gdb
  - Lauterbach
  - ...

Real-time Linux basics

1. Can Linux do real-time?
2. Approaches for enhancing real-time
3. Low-latency and preemption
4. Uber-preemption
5. RTLinux
6. RTAI
7. Adeos / Xenomai

1. Can Linux do real-time?

- Linux provides soft-real-time
- Best worst-case (almost all the time) resolution for Linux: 1ms.
- Linux does not provide deterministic response times / Linux cannot guarantee hard-real-time.
- Why can't it do it?
  - Inadequate scheduling mechanisms
  - Interrupt mask manipulation in drivers
  - Fancy interrupt handling

- Ways to avoid impacting Linux's latency for PCs:
  (as listed by Andrew Morton on his site)
  - Avoid scrolling framebuffer => the fb console runs with interrupts disabled.
  - Avoid using hdparm => can eat up 50ms
  - Avoid using blkdev_close() => can eat up N*10ms
  - Avoid switching consoles => can disable ints for > 1ms.

2. Approaches for enhancing real-time

- Boat-in-a-bottle approach:
  - Careful select all system components ...
- Low-latency:
  - Adds scheduling points to portions of the kernel.
- Preemption:
  - Modifies locking mechanisms to allow scheduling right after lock release.
- Uber-preemption:
  - Thread interrupt handlers and modify locking fcts
- Dual-kernel approach:
  - Run Linux as lowest-priority task of RTOS (patented)
- Nanokernel approach:
  - Run all OSes as nanokernel clients

3. Low-latency and preemption

- Low-latency approach:
  - Example: fs/buffer.c

```
+       if (conditional_schedule_needed()) {
+           spin_unlock(&lru_list_lock);
+           unconditional_schedule();
+           spin_lock(&lru_list_lock);
+           goto repeat;
+       }
```

  - include/linux/low-latency.h:*conditional_schedule_needed()* macro

```
(enable_lowlatency && current->need_resched)
```

- Preemption approach:
  - Adds CONFIG_PREEMPT config option
  - Lock modifications implemented in include/linux/spinlock_api_*.h.
  - Requires disabling preemption when accessing structures which were safe because kernel wasn't preemptable.
  - Example lock redefinition:

```
static inline void __raw_spin_unlock(raw_spinlock_t *lock)
{
        spin_release(&lock->dep_map, 1, _RET_IP_);
        do_raw_spin_unlock(lock);
        preempt_enable();
}
```

- What does *preempt_enable()* do?

```
#define preempt_enable() \
do { \
        preempt_enable_no_resched(); \
        barrier(); \
        preempt_check_resched(); \
} while (0)
```

- Example modified code for preemption protection (old)

```
 void __global_sti(void)
 {
-    int cpu = smp_processor_id();
+    int cpu;

+    preempt_disable();
+    cpu = smp_processor_id();
     if (!local_irq_count(cpu))
         release_irqlock(cpu);
     __sti();
+    preempt_enable();
 }
```

4. Uber-preemption

- Extreme threading of the Linux kernel
- Pieces have been merged piecemeal: high-resolution timers, threaded, interrupts, priority inheritance support, ...
- Still a work-in-progress, a few years from full merge
- Consists mainly of:
  - Threading interrupt handlers so that they can be scheduled alongside tasks.
  - Changing the locking primitives to allow better threading of core kernel code.

5. RTLinux

- How it works:
  - Runs Linux as lowest priority task of RTLinux
  - Redefines cli/sti to disable/enable flag for interrupt delivery.
  - Delivers interrupts to Linux only if flag enabled
- Started mid-90's
- University project, turned patent, turned company
- Sold to WindRiver in 2007
- Not so relevant anymore

6. RTAI

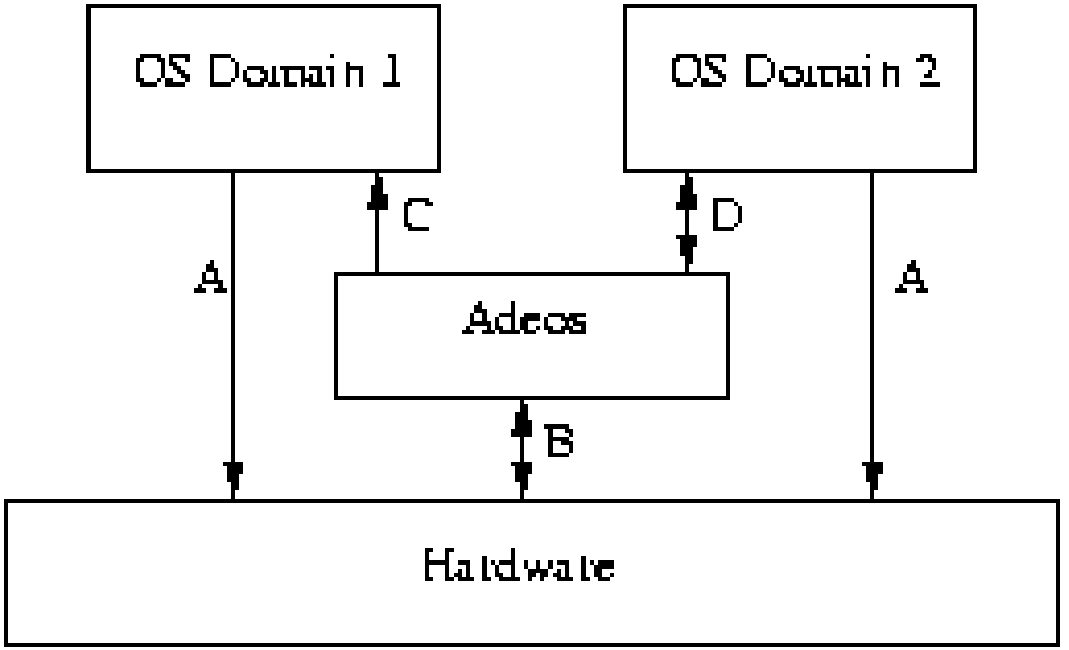
- www.rtai.org
- Based on work done Paolo Mantegazza in the '80s to obtain real-time in DOS using a TSR.
- Mostly x86-based
- Rallying point for all RTLinux patent contesters

7. Adeos / Xenomai

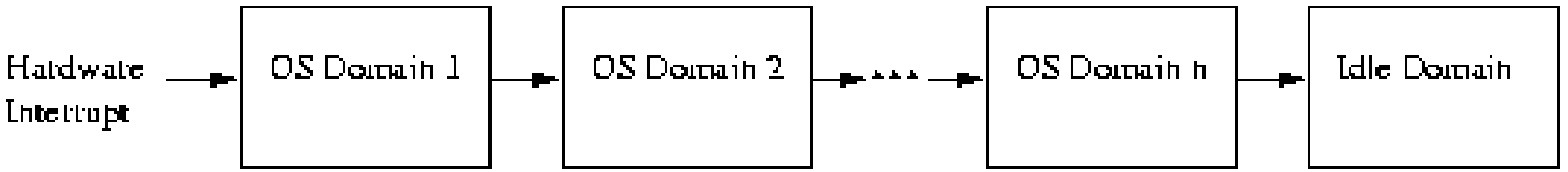
- History:
  - Architecture and background detailed in whitepaper published by K.Y. in February 2001.
  - Based entirely on nanokernel research papers published more than 1 year prior to preliminary patent application.
  - Project taken up by Philippe Gerum in late April 2002
  - First release made June 3rd 2002 / endorsed by quite a few open source organizations, including April and EuroLinux.
  - Adeos is used as the basis of Xenomai
  - Xenomai provides various real-time “skins”

- Adeos Capabilities:
  - Provide minimal environment for allowing multiple OSes to share the same hardware.
  - Forbids any OS, including the RTOS, to manipulate interrupt masks directly.
  - All OS are clients of Adeos
  - Once OS is done with CPU, it gives control back to Adeos ... not its *idle()* task.
  - Currently supports x86, ia64, MMU-full and MMU-less ARM, and PowerPC.
  - Easily portable to other architectures

- General architecture



- Interrupt pipeline



- Adeos is available as a kernel patch from http://www.adeos.org/
- API summary (old – for illustration only):

```
int adeos_register_domain(adomain_t *adp,
                          adattr_t *attr);
void adeos_unregister_domain(adomain_t *adp);
void adeos_renice_domain(int newpri);
void adeos_suspend_domain(void);
int adeos_virtualize_irq(unsigned irq,
                         void (*handler)(unsigned irq),
                         int (*acknowledge)(unsigned irq),
                         unsigned modemask);
void adeos_control_irq(unsigned irq,
                       unsigned clrmask,
                       unsigned setmask);
void adeos_stall_ipipe(void);
void adeos_unstall_ipipe(void);
unsigned adeos_alloc_irq(void);
int adeos_free_irq(unsigned irq);
int adeos_trigger_irq(unsigned irq);
```

- Xenomai:
  - Actively maintained at: http://xenomai.org
  - Provides skins for:
    - Native
    - VxWorks
    - VRTX
    - pSOS+
    - RTDM
  - Supports multiple architectures:
    - ARM
    - Blackfin
    - x86

Application debugging

1. gdb
2. Single process tracing
3. System tracing
4. Performance analysis
5. Memory debugging

... NFS-mounted root is your friend ...

1. gdb

- Standard symbolic debugger for quite a few Unix systems.
- Most important debugging tool
- Relies on *ptrace()*
- Very large binary, can't use as-is on target
- Use gdb server instead
- Available from: ftp://ftp.gnu.org/gnu/gdb/
- Built by crosstool-ng (as we'll see later)
- Build example based on 5.2.1 (old) put into ${PRJROOT}/debug

1.1. Build example based on 5.2.1 (old)

- Build, configure, and install gdb on host:

```
$ mkdir ${PRJROOT}/debug/build-gdb
$ cd ${PRJROOT}/debug/build-gdb
$ ../gdb-5.2.1/configure --target=$TARGET \
> --prefix=${PREFIX}
$ make
$ make install
```

- Generates `powerpc-linux-gdb`
- (for now) Can't read target core files on host
- Build and install gdb server:

```
$ mkdir ${PRJROOT}/debug/build-gdbserver
$ cd ${PRJROOT}/debug/build-gdbserver
$ chmod +x ../gdb-5.2.1/gdb/gdbserver/configure
$ CC=powerpc-linux-gcc \
> ../gdb-5.2.1/gdb/gdbserver/configure \
> --host=$TARGET --prefix=${TARGET_PREFIX}
$ make
$ make install
```

1.2. Using gdb / gdbserver

- Copy gdb server to target's root filesystem:

```
$ cp ${TARGET_PREFIX}/${TARGET}/debug-root/usr/bin/gdbserver \
> ${PRJROOT}/rootfs/usr/bin
```

- Can use stripped binaries on target and unstripped binaries on host.
- Building binary for debugging:

```
...
DEBUG           = -g
CFLAGS          = -O2 -Wall $(DEBUG)
...
```

- Can also use -ggdb for more debugging info
- Two ways to connect gdb server to gdb-host:
  - Serial
  - TCP/IP

- On target:

```
# gdbserver 192.168.202.100:2345 example-app
```

- Can tunnel over SSH
- On host:

```
$ powerpc-linux-gdb example-app
(gdb) target remote 192.168.202.79:2345
Remote debugging using 192.168.202.79:2345
0x10000074 in _start (  )
```

- Can now use all standard gdb commands ...
- Important commands when using gdb server:
  - file
  - dir
  - target
  - set remotebaud

  - set solib-absolute-prefix
- Providing gdb with library path (don't put “/lib”):

```
(gdb) set solib-absolute-prefix ../../tools/powerpc-linux/..
```

- Path is essentially:
  - ${TARGET_PREFIX}/${TARGET}/sysroot/
- Can use .gdbinit to automate execution of a few commands.
- Can easily interface with graphical front-end such as DDD, GDB/Insight and KDbg.
- Can integrate into IDE: Eclipse, Kdevelop, Emacs, ...

2. Single process tracing

- `strace` is the standard command to trace process execution.
- By definition: Alters application behavior because of *ptrace()*.
- Available from http://strace.sourceforge.net
- Built as part of crosstool-ng
- Download (4.5.20) to your ${PRJROOT}/debug directory and extract it there.
- Configure, build, and install:

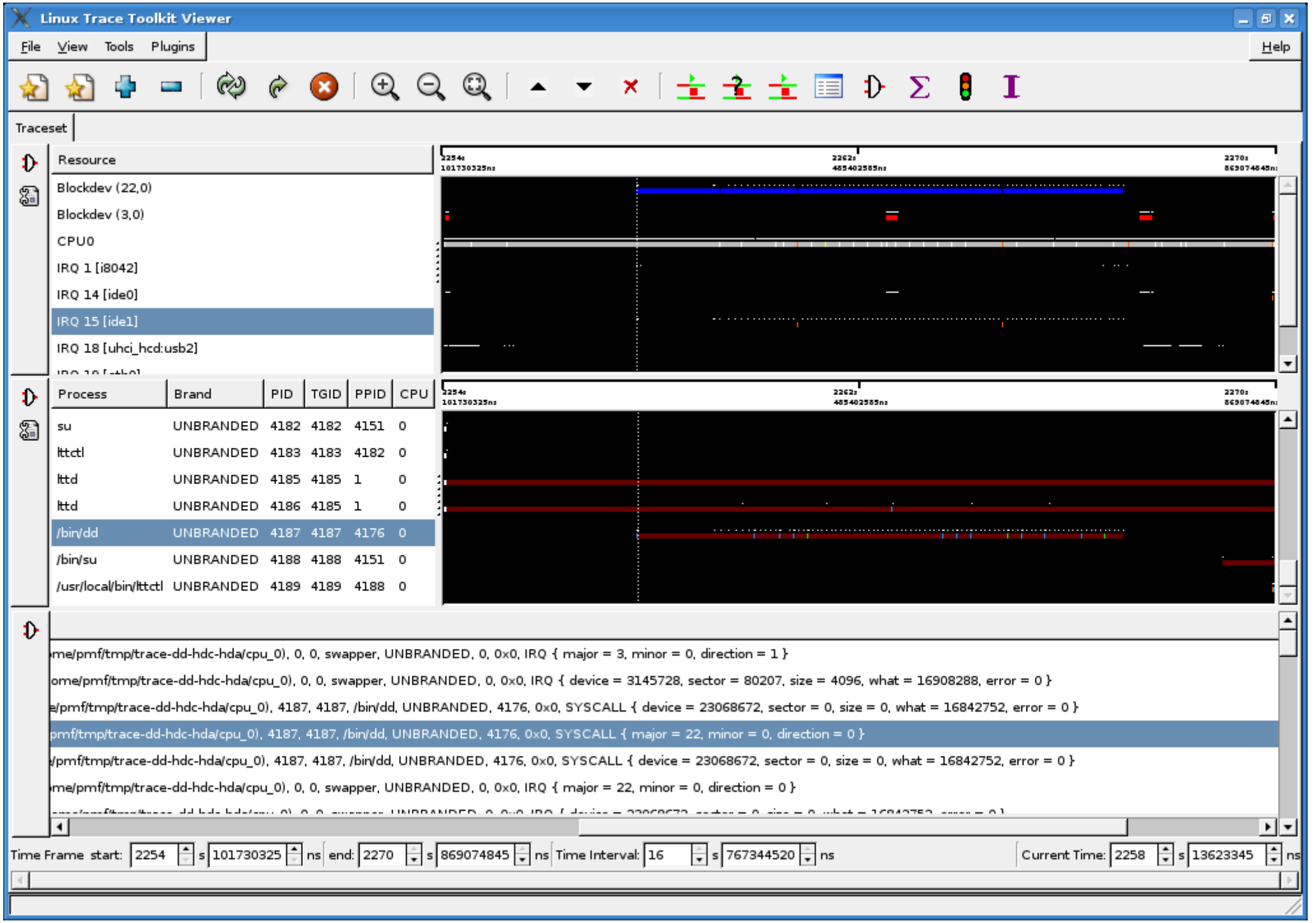
```
$ cd ${PRJROOT}/debug/strace-4.5.20
$ CC=${TARGET}-gcc ./configure --host=$TARGET
$ make
$ cp strace ${PRJROOT}/rootfs/usr/sbin
```

- Example output:

```
$ strace /bin/ls
execve("/bin/ls", ["/bin/ls"], [/* 35 vars */]) = 0
uname({sys="Linux", node="localhost.localdomain", ...
brk(0)                                  = 0x80586c8
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT ...
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=115094, ...
old_mmap(NULL, 115094, PROT_READ, MAP_PRIVATE, 3, 0) ...
close(3)                                = 0
open("/lib/libtermcap.so.2", O_RDONLY)  = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\ ...
fstat64(3, {st_mode=S_IFREG|0755, st_size=11784, ...
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVA ...
old_mmap(NULL, 14856, PROT_READ|PROT_EXEC, MAP_PRIVA ...
old_mmap(0x40037000, 4096, PROT_READ|PROT_WRITE, MAP_ ...
close(3)                                = 0
open("/lib/libc.so.6", O_RDONLY)        = 3
  ...
brk(0)                                  = 0x805a000
brk(0x805c000)                          = 0x805c000
open("/dev/null", O_RDONLY|O_NONBLOCK|O_DIRECTORY) ...
open(".", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTO ...
  ...
```

3. System tracing

- ftrace:
  - Mainline
  - Monitors function calls
  - See Documentation/ftrace.txt
- Linux Trace Toolkit:
  - Non-mainline
  - Introduced in 1999 by Karim Yaghmour
  - Maintained since 2005 at http://lttng.org/
  - Collects lots of information
  - Very low impact
  - Graphical analysis tool


Linux Trace Toolkit Viewer
File View Tools Plugins
Help
Traceset
Resource
Blockdev (22,0)
Blockdev (3,0)
CPU0
IRQ 1 [i8042]
IRQ 14 [ide0]
IRQ 15 [ide1]
IRQ 18 [uhci_hcd:usb2]
Process Brand PID TGID PPID CPU
su UNBRANDED 4182 4182 4151 0
lttctl UNBRANDED 4183 4183 4182 0
lttd UNBRANDED 4185 4185 1 0
lttd UNBRANDED 4186 4185 1 0
/bin/dd UNBRANDED 4187 4187 4176 0
/bin/su UNBRANDED 4188 4188 4151 0
/usr/local/bin/lttctl UNBRANDED 4189 4189 4188 0
me/pmf/tmp/trace-dd-hdc-hda/cpu_0), 0, 0, swapper, UNBRANDED, 0, 0x0, IRQ { major = 3, minor = 0, direction = 1 }
ome/pmf/tmp/trace-dd-hdc-hda/cpu_0), 0, 0, swapper, UNBRANDED, 0, 0x0, IRQ { device = 3145728, sector = 80207, size = 4096, what = 16908288, error = 0 }
e/pmf/tmp/trace-dd-hdc-hda/cpu_0), 4187, 4187, /bin/dd, UNBRANDED, 4176, 0x0, SYSCALL { device = 23068672, sector = 0, size = 0, what = 16842752, error = 0 }
pmf/tmp/trace-dd-hdc-hda/cpu_0), 4187, 4187, /bin/dd, UNBRANDED, 4176, 0x0, SYSCALL { major = 22, minor = 0, direction = 0 }
/pmf/tmp/trace-dd-hdc-hda/cpu_0), 4187, 4187, /bin/dd, UNBRANDED, 4176, 0x0, SYSCALL { device = 23068672, sector = 0, size = 0, what = 16842752, error = 0 }
me/pmf/tmp/trace-dd-hdc-hda/cpu_0), 0, 0, swapper, UNBRANDED, 0, 0x0, IRQ { major = 22, minor = 0, direction = 0 }
Time Frame start: 2254 s 101730325 ns end: 2270 s 869074845 ns Time Interval: 16 s 767344520 ns
Current Time: 2258 s 13623345 ns

4. Performance analysis

- Process profiling
  - Profiling process call tree
  - Relies on `gprof`
  - Building a binary with profiling information:

```
CFLAGS          = -Wall -pg
...
LDFLAGS         = -pg
```

  - Avoid -O
  - During execution, a gmon.out file will be generated
  - Generating profile information (cross-platform)

```
$ gprof example-app
```

  - Output goes to stdout. May want to redirect.

- Code coverage:
  - Identifying code portions being executed
  - Relies on `gcov` and `libgcc`
  - Building a binary with coverage information:

```
CFLAGS          = -Wall -fprofile-arcs -ftest-coverage
```

  - Avoid -O
  - After compilation, .bb and .bbg file created for each source file.
  - At runtime, .da file generated on target in original host development path (${PRJROOT}/project/...)

- Generating the coverage information:

```
$ gcov prj.c
 30.48% of 3568 source lines executed in file prj.c
Creating prj.c.gcov.
```

- .gcov file is text

- ps utility ... BusyBox implementation is very short on details and full versions are complicated to cross-compile.

5. Memory debugging

- Understanding what applications are doing with memory.
- Two main libraries:
  - DUMA
  - Valgrind
- DUMA
  - Provides substitutes for *malloc()* and *free()*
  - Implements limit testing
  - Provides exact fault location using gdb
  - Available from: http://duma.sourceforge.net/

- Valgrind Memcheck:
  - Very powerful
  - Very popular
  - Detects common errors:
    - Illegal memory access
    - Use of uninitialized values
    - Memory leaks
    - Bad heap freeing
    - Overlapping source and destination memcpy()

Networking services

1. Internet super-server
2. SNMP for remote administration
3. `telnet` for remote login
4. SSH for secure communication
5. HTTP for web serving
6. DHCP for dynamic configuration

1. Internet super-server

- In Unix, all network services listen to service requests on a particular port.
- Instead of having all network services listening at the same time, it's preferable to have one server listening to all ports.
- The Internet super-server takes care of this
- It starts network services dynamically, according to the requests it receives.
- Some services don't use the super-server because of reliability and scalability issues.

- Two main super-server implementations:
  - `inetd`
  - `xinetd`
- `inetd`:
  - The older of two super-servers
  - Used to be the standard super-server in most distributions.
  - Part of netkit-base
  - Available from:
    ftp://ftp.uk.linux.org/pub/linux/Networking/netkit/
  - BSD license

- Download (0.17) to your ${PRJROOT}/sysapps directory and extract it there.
- Move to directory for rest of operations:

```
$ cd ${PRJROOT}/sysapps/netkit-base-0.17
```

- Need to modify configure script to avoid running tests on host:

```
#                ./_ _conftest || exit 1;
```

- If using uClibc, need RPC support
- Build and install package:

```
$ CC=powerpc-linux-gcc ./configure \
> --prefix=${TARGET_PREFIX}
$ make
$ cp inetd/inetd ${PRJROOT}/rootfs/usr/sbin
$ cp etc.sample/inetd.conf ${PRJROOT}/rootfs/etc
```

- Example inetd.conf entry:

```
telnet  stream  tcp  nowait root    /usr/sbin/telnetd
```

- Modify your /etc/inittab to start `inetd`:

```
::respawn:/usr/sbin/inetd -i
```

- `xinetd`:
  - Much more elaborate and adapted to modern networks that `inetd` (more security, more logging, ...)
  - Available from:

    http://www.xinetd.org/
  - BSD license
  - Download (2.3.9) and extract in your ${PRJROOT}/sysapps directory.

- Move to directory for rest of operations

```
$ cd ${PRJROOT}/sysapps/xinetd-2.3.9
```

- If using uClibc, need RPC and C99
- Configure, build, and install package:

```
$ CC=powerpc-linux-gcc ./configure --host=$TARGET \
> --prefix=${TARGET_PREFIX}
$ make
$ make install
$ cp ${TARGET_PREFIX}/sbin/xinetd \
> ${PRJROOT}/rootfs/usr/sbin
$ powerpc-linux-strip \
> ${PRJROOT}/rootfs/usr/sbin/xinetd
$ cp xinetd/sample.conf \
> ${PRJROOT}/rootfs/etc/xinetd.conf
```

- xinetd.conf settings for the telnet daemon:

```
service telnet
{
        socket_type             = stream
        wait                    = no
        user                    = root
        server                  = /usr/sbin/telnetd
        bind                    = 127.0.0.1
        log_on_failure          += USERID
}
```

- Modify your /etc/inittab to start `xinetd`:

```
::once:/usr/sbin/xinetd
```

2. SNMP for remote administration

- Need to remotely administer / monitor a headless networked device (router, switch, etc.)?
- There are two components for SNMP:
  - SNMP Agent: runs in embedded system / allows for remote management.
  - SNMP Manager: runs on desktop / monitors SNMP-enabled devices.
- Manager can query agent for status
- Agent can notify manager of problems using SNMP traps.
- For Linux, main SNMP is Net-SNMP

- Available from:
  http://net-snmp.sourceforge.net/
- Download (5.0.6) and extract in your ${PRJROOT}/sysapps directory.
- BSD license
- Move to directory for rest of operations:
  ```
  $ cd ${PRJROOT}/sysapps/net-snmp-5.0.6
  ```
- If using uClibc, need IPv6 and need to fix agent/mibgroup/ucd-snmp/disk.c (see BELS for details.)

- Configure SNMP:

```
$ CC=powerpc-linux-gcc ./configure --host=$TARGET \
> --with-endianness=big
```

- Configure will ask a few questions about the target's configuration. These will depend on your system's architecture.
- Build and install daemon:

```
$ make
$ make prefix=${TARGET_PREFIX} \
> exec_prefix=${TARGET_PREFIX} install
$ cp ${TARGET_PREFIX}/sbin/snmpd \
> ${PRJROOT}/rootfs/usr/sbin
$ mkdir -p ${PRJROOT}/rootfs/usr/local/share
$ cp -r ${TARGET_PREFIX}/share/snmp \
> ${PRJROOT}/rootfs/usr/local/share
$ cp EXAMPLE.conf \
> ${PRJROOT}/rootfs/usr/local/share/snmp/snmpd.conf
```

- Modify /etc/inittab for the SNMP daemon:

```
::respawn:/usr/sbin/snmpd -f
```

3. `telnet` for remote login

- Simplest form of remote login
- Very insecure
- Two telnet daemons:
  - netkit-telnetd (the standard telnet daemon)
  - utelnetd
- netkit-telnetd:
  - Available from:
    ftp://ftp.uk.linux.org/pub/linux/Networking/netkit/
  - Download (0.17) and extract in your ${PRJROOT}/sysapps directory.

- As with netkit-base, need to modify configure script:

```
#                ./_ _conftest || exit 1;
```

- Build and install telnet daemon

```
$ CC=powerpc-linux-gcc ./configure \
> --prefix=${TARGET_PREFIX}
$ touch ${TARGET_PREFIX}/include/termcap.h
$ make -C telnetd
$ cp telnetd/telnetd ${PRJROOT}/rootfs/usr/sbin
```

- Don't forget to enable super-server

- utelnetd:
  - Available from:

    http://www.pengutronix.de/software/utelnetd_en.html
  - Download (0.1.3) and extract in your ${PRJROOT}/sysapps directory.

- GPL license
- Move to directory, build, and install:

```
$ cd ${PRJROOT}/sysapps/utelnetd-0.1.3
$ CC=powerpc-linux-gcc make
$ cp utelnetd ${PRJROOT}/rootfs/usr/sbin
```

- Modify /etc/inittab:

```
::respawn:/usr/sbin/utelnetd
```

4. SSH for secure communication

- Secure replacement for telnet and quite a few other Unix utilities.
- Uses public key encryption
- Main Linux package: OpenSSH
- OpenSSH requires OpenSSL and zlib
- OpenSSH available from:

  http://www.openssh.org/
- OpenSSL available from:

  http://www.openssl.org/

- Install zlib as static library for your target
- Download OpenSSL (0.9.6.g) into your ${PRJROOT}/build-tools directory and extract it there.
- Configure, build and install OpenSSL:

```
$ ./config --prefix=${TARGET_PREFIX} \
> compiler:powerpc-linux-gcc
$ make
$ make install
```

- Download OpenSSH (3.5p1) into your ${PRJROOT}/sysapps directory and extract it there.

- Need to trick OpenSSH into building
- Trick, configure, build, and install:

```
$ export PATH=./:$PATH
$ which gcc
/usr/bin/gcc
$ ln -s /usr/bin/gcc ./powerpc-linux-gcc
$ ln -s /usr/include/openssl ./fake-include
$ ln -s /usr/lib ./fake-lib
$ CC=powerpc-linux-gcc CFLAGS=-I./fake-include \
> LDFLAGS=-L./fake-lib ./configure --host=$TARGET
$ rm powerpc-linux-gcc fake-include fake-lib
$ ln -s ${TARGET_PREFIX}/include ./fake-include
$ ln -s ${TARGET_PREFIX}/lib ./fake-lib
$ make
$ cp ./sshd ${PRJROOT}/rootfs/usr/sbin
$ powerpc-linux-strip ${PRJROOT}/rootfs/usr/sbin/sshd
```

- Copy OpenSSH configuration:

```
$ mkdir -p ${PRJROOT}/rootfs/usr/local/etc
$ cp sshd_config ${PRJROOT}/rootfs/usr/local/etc
```

- Generate SSH keys:

```
$ ssh-keygen -t rsa1 -f \
> ${PRJROOT}/rootfs/usr/local/etc/ssh_host_key
$ ssh-keygen -t rsa -f \
> ${PRJROOT}/rootfs/usr/local/etc/ssh_host_rsa_key
$ ssh-keygen -t dsa -f \
> ${PRJROOT}/rootfs/usr/local/etc/ssh_host_dsa_key
```

- Create /var entries for OpenSSH:

```
$ mkdir -p ${PRJROOT}/rootfs/var/run \
> ${PRJROOT}/rootfs/var/empty
$ su -m
Password:
# chown root:root ${PRJROOT}/rootfs/var/run \
> ${PRJROOT}/rootfs/var/empty
# chmod 755 ${PRJROOT}/rootfs/var/empty
# exit
```

- Must add privilege separation user:
  - /etc/group:

    ```
    sshd:x:255:
    ```

  - /etc/passwd:

    ```
    sshd:x:501:255:sshd privsep:/var/empty:/bin/false
    ```

  - /etc/shadow:

    ```
    sshd:*:11880:0:99999:7:-1:-1:0
    ```

- Modify /etc/inittab to start OpenSSH:

  ```
  ::respawn:/usr/sbin/sshd -D
  ```

5. HTTP for web serving

- Apache, the world's most popular web server, cannot be cross-compiled.
- Two alternatives for embedded systems:
  - boa (GPL)
  - thttpd (BSD)
- Boa:
  - Available from:

    http://www.boa.org/
  - Download (0.94.13) into your ${PRJROOT}/sysapps directory and extract it there.

- Move to directory for rest of operations:

```
$ cd ${PRJROOT}/sysapps/boa-0.94.13/src
```

- Configure, build, and install boa:

```
$ ac_cv_func_setvbuf_reversed=no \
> ac_cv_struct_tm_has_tm_gmtoff=no CC=powerpc-linux-gcc \
> ./configure --host=$TARGET
$ make
$ cp boa ${PRJROOT}/rootfs/usr/sbin
$ powerpc-linux-strip ${PRJROOT}/rootfs/usr/sbin/boa
```

- See BELS for static linking
- Install configuration file:

```
$ mkdir -p ${PRJROOT}/rootfs/etc/boa
$ cp ../boa.conf ${PRJROOT}/rootfs/etc/boa
```

- Create directory for logs and errors:

```
$ mkdir -p ${PRJROOT}/rootfs/var/log/boa
```

- Create directory for HTML content and fill:

```
$ mkdir -p ${PRJROOT}/rootfs/var/www
$ cp   ... ${PRJROOT}/rootfs/var/www
```

- Modify /etc/inittab for boa:

```
::respawn:/usr/sbin/boa
```

- thttpd:
  - Available from:

    http://www.acme.com/software/thttpd/
  - Download (2.23beta1) into your ${PRJROOT}/sysapps directory and extract it there.
  - Move to directory for rest of operations:

```
$ cd ${PRJROOT}/sysapps/thttpd-2.23beta
```

  - Configure, build, and install:

```
$ CC=powerpc-linux-gcc ./configure --host=$TARGET
$ make
$ cp thttpd ${PRJROOT}/rootfs/usr/sbin
$ powerpc-linux-strip \
> ${PRJROOT}/rootfs/usr/sbin/thttpd
```

- Copy configuration file

```
$ cp contrib/redhat-rpm/thttpd.conf \
> ${PRJROOT}/rootfs/etc
```

- Create directory for HTML content and fill:

```
$ mkdir -p ${PRJROOT}/rootfs/home/httpd/html
```

- Modify /etc/inittab:

```
::respawn:/usr/sbin/thttpd -C /etc/thttpd.conf
```

6. DHCP for dynamic configuration

- ISC-DHCP, the standard DHCP package for Linux, cannot be cross-compiled.
- Must use udhcp
- Available from:

  http://udhcp.busybox.net/
- Download (0.9.8) into your ${PRJROOT}/sysapps directory and extract it there.
- Move to the directory for the rest of the operations:

```
$ cd ${PRJROOT}/sysapps/udhcp-0.9.8
```

- Cross-compile the client and the server:

```
$ make CROSS_COMPILE=powerpc-uclibc-
```

- Copy the server:

```
$ cp udhcpd ${PRJROOT}/rootfs/usr/sbin
```

- Copy the client:

```
$ cp udhcpc ${PRJROOT}/rootfs/sbin
```

- For the server:

```
$ mkdir -p ${PRJROOT}/rootfs/var/lib/misc
$ touch ${PRJROOT}/rootfs/var/lib/misc/udhcpd.leases
$ cp samples/udhcpd.conf ${PRJROOT}/rootfs/etc
```

- For the client:

```
$ mkdir -p ${PRJROOT}/rootfs/etc/udhcpc
$ mkdir -p ${PRJROOT}/rootfs/usr/share/udhcpc
$ cp samples/sample.renew \
> ${PRJROOT}/rootfs/usr/share/udhcpc/default.script
```

- Modify /etc/inittab:

```
::respawn:/usr/sbin/udhcpd
```

User interfaces

1. Hardware interfacing
2. Graphical user interfaces
3. Widget sets
4. Web browsers
5. Audio considerations
6. PVRs

1. Hardware interfacing

- Terminal:
  - Standard Unix terminal
  - Allows implementation of basic DOS-like menus
  - Though not visually-appealing, can be very powerful.
- X (http://www.xfree86.org/):
  - Standard windowing system for all Unix systems
  - Implemented entirely in user-space
  - Relies on a client-server architecture
  - There is always only one X server
  - Clients connect to server through sockets

- Server is in control of all hardware (keyboard / mouse / display).
- Client send display requests to server
- Server sends events to clients
- There is one special type of client: the window manager (WM.)
- WM is responsible for drawing the decorations of all windows.
- The WM requests that the X server forward it all events relating to all X clients (initial display, hiding, moving, resizing, activation / deactivation, destruction.)

- At startup, WM “reparents” all windows to itself
- MIT X11 license

- FrameBuffer:
  - Provides raw access to low-level display
  - Implemented as kernel driver
  - Standard mechanism for controlling embedded displays such as LCDs.
- Microwindows (http://www.microwindows.org/)
  - Replaces X Window System
  - Implements Microsoft Windows Win32 / WinCE Graphics Display Interface (GDI).

- Provides Nano-X (xlib-like interface)
- Can work within an X Window for prototyping
- No modification required when cross-compiling
- Runs directly on framebuffer on target
- Nano-X behavior very close to X Window behavior
- Quite a few default configurations already available
- There's a NanoWM
- Dual-licensed MPL/GPL
- Widely use in embedded systems

- SDL (http://www.libsdl.org/):
  - SDL = Simple DirectMedia Layer
  - Provides low-level direct access to video (including fb), keyboard, mouse, sound, media, etc.
  - Not as popular as X, but in use
  - LGPL
- DirectFB (http://www.directfb.org/):
  - Provides direct low-level programming layer for video, keyboard, and mouse.
  - Powerful engine
  - There's also a XDirectFB

  - LGPL library
- GGI (http://www.ggi-project.org/):
  - Library implementing extensive generalization of display mechanisms.
  - Runs on iPaq
  - Not really widely used
- AALib (http://aa-project.sourceforge.net/):
  - ASCII Art library
  - Nice way of doing graphics if you only have ASCII output.
  - Not really widely used

- SVGA lib (http://www.svgalib.org/):
  - Low-level user-space library for controlling video cards.
  - Not really widely used

2. Graphical user interfaces

- Familiar:
  - GPE (http://gpe.handhelds.org/):
    - Based on X Window System and GTK+ 2.2 widget toolkit
    - Contains quite a few separate components
    - Provides unified palmtop environment
  - Opie (http://opie.handhelds.org/)
    - Based on Qt widget toolkit
    - Doesn't need X
    - Fork from Trolltech's Qtopia
    - Provides unified palmtop environment
- Matchbox (http://handhelds.org/~mallum/matchbox/)

- Based on X Window System
- Does not depend on any widget toolkit
- Provides unified palmtop environment

- Packages that replace X Window, window manager, and widget toolkit:
  - MiniGUI (http://www.minigui.org/)
  - OpenGUI (http://www.tutok.sk/fastgl/)
  - PicoGUI (http://www.picogui.org/)

3. Widget sets

- tcl/tk:
  - Easy way of developing X Window applications
  - Kernel config system built on tcl/tk
- Curses:
  - Easy way of programming text-based terminal applications.
  - ncurses is the standard Linux version
- FLTK (http://www.fltk.org/):
  - X-lib based widget set
  - Optimized for size footprint
  - LGPL license

- GTK (http://www.gtk.org/):
  - Extensive widget set
  - Basis for Gnome desktop
  - X-lib based
  - LGPL license
- Qt (http://www.trolltech.com/)
  - CAREFUL: all Qt products require purchasing a license for commercial development.
  - The freely available Qt/X11 is only good for developing non-proprietary software, but can run software developed by others (free or not.)
  - The freely available Qt embedded is GPL library

4. Web browsers

- ViewML (http://www.viewml.com/):
  - Based on KDE's kfm and FLTK
  - Requires X
  - Optimized for size
- NxZilla (http://nxzilla.sourceforge.net/):
  - Set of libraries to run Mozilla on Nano-X
- Dillo (http://www.dillo.org/):
  - Based GTK+
  - Optimized for size

5. Audio considerations

- OSS API (http://www.opensound.com/):
  - Used to be main sound system for Linux
  - Programming guide at:

    http://www.opensound.com/pguide/
- ALSA (http://www.alsa-project.org/):
  - Next generation sound system for Linux
  - Integrated in 2.5 development series
  - Extensive backing and documentation

6. PVRs

- MythTV:
  - Fully-featured PVR solution
  - Features according to web site:
    - Basic 'live-tv' functionality. Pause/Fast Forward/Rewind "live" TV.
    - Support for multiple tuner cards and multiple simultaneous recordings.
    - Distributed architecture allowing multiple recording machines and multiple playback machines on the same network, completely transparent to the user.
    - Compresses video in software using rtjpeg (from Nuppelvideo) or mpeg4 (from libavcodec). Full support for Hardware MPEG-2 encoder cards (Hauppauge PVR-250 / PVR-350). Preliminary support for DVB cards and the new pcHDTV tuner card.
    - Support for the (very nice looking) hardware MPEG-2 decoder and TV out present on the Hauppauge PVR-350.
    - Completely automatic commercial detection/skipping
    - Grabs program information using xmltv.
    - A fully themeable menu to tie it all together.

- Freevo:
  - Basic/easy PVR solution for Linux
  - According to web site:
    - Freevo is an open-source home theatre PC platform based on Linux and a number of open-source audio/video tools. MPlayer and/or Xine can be used to play audio and video files in most popular formats. Freevo can be used both for a standalone PVR computer with a TV+remote, as well as on a regular desktop computer using the monitor and keyboard.
    - Freevo is easy to download and install for new users. Most hardware is supported (graphic boards, sound cards and video capture devices).

Development tools setup

1. GNU toolchain basics
2. Kernel headers setup
3. Binutils setup
4. Installing glibc headers
5. Bootstrap compiler setup
6. C library setup
7. Full compiler setup
8. Finalizing the toolchain setup
9. Using the toolchain

10. C library alternatives
11. Other programming languages
12. Cross-compiling the impossible
13. Auto-generating toolchains

1. GNU toolchain basics

0. A word of caution
1. Location of GNU packages
2. Component versions
3. Build requirements
4. Build overview
5. Preparing the build-tools directory
6. Resources
7. What about pre-built tools?

1.0. A word of caution

- Building functional cross-toolchains is an art
- It is:
  - Very tedious
  - Very error prone
  - Very lengthy
- Need to reconcile:
  - Guarantee that it'll work
  - Independence from 3rd party software suppliers
- Solution: crosstool-ng
  - Auto-builds toolchains from source

1.1. Location of GNU packages

- ftp://ftp.gnu.org/gnu/
- binutils is in binutils/
- gcc is in gcc/
- glibc is in glibc/

1.2. Component versions

- Need to select working combination of binutils, gcc, and glibc.
- Latest version not guaranteed to work for all targets.
- Asking around is always a good idea
- If nothing appropriate found, start with latest and backtrack.
- Backtracking may involve reverting to older versions of tools that have properly built.
- Target-specific web sites and mailing lists may also have patches / modifications to suggest.

- Kernel version is of no importance as long as it is 2.2.x, 2.4.x or 2.6.x. Use the latest.
- In addition to glibc, need glibc-linuxthreads package. Same version numbering as glibc.
- WATCH OUT: Some version combinations work on some targets within an architecture family and not others. (MPC8xx vs. desktop PPC is good example of this.)

1.3. Build requirements

- Need a functional native toolchain:

  Has likely been installed already by your distro
- Need a set of good kernel headers:
  - /usr/include/linux
  - /usr/include/asm
  - /usr/include/asm-generic
- Headers must be those used to build native glibc

1.4. Build overview

- Main steps:
  1. Kernel headers setup
  2. binutils setup
  3. glibc headers setup
  4. Bootstrap gcc setup
  5. glibc setup
  6. Full gcc setup
- Compiler is built twice because first time does not include C++ support since the latter needs the C library ...
- Kernel headers required for glibc build

- Each step involves:
  1. Unpacking the package
  2. Configuring the package
  3. Building the package
  4. Installing the package
- No need to build and install kernel

1.6. Preparing the build-tools dir

- Download necessary packages to the ${PRJROOT}/build-tools directory.
- Never build directly in package's source directory.
- Create separate directory for building each toolchain component:

```
$ cd ${PRJROOT}/build-tools
$ mkdir build-binutils build-glibc-headers \
> build-boot-gcc build-glibc build-gcc
```

1.7. Resources

- Each package comes with documentation:
  - binutils: not much doc, but very few problems
  - gcc: FAQ file and install/ directory
  - glibc: FAQ and INSTALL files.
- **Google is your friend**
- **List**: http://sources.redhat.com/ml/crossgcc/
- crosstool-ng source code
- See http://elinux.org/Toolchains

1.8. What about pre-built tools?

- Varying degrees of quality and pricing
- If you choose to use pre-built tools, make sure you have access to the original instructions used to build these tools directly from source.
- If any patches were used, make sure you have those too.
- Free examples:
  - Codesourcery
  - Linaro (ARM)
- Commercials offerings ...

2. Kernel headers setup

- Download kernel from http://www.kernel.org into your ${PRJROOT}/kernel directory.
- Extract kernel:

```
$ cd ${PRJROOT}/kernel
$ tar xvjf linux-2.6.11.tar.bz2
```

- Move to the kernel's directory, configure, and create required headers:

```
$ cd linux-2.6.11
$ make ARCH=ppc defconfig
$ make ARCH=ppc include/linux/version.h
```

- Copy kernel headers for toolchain build:

```
$ mkdir -p ${TARGET_PREFIX}/include
$ cp -r include/linux/ \
> ${TARGET_PREFIX}/include
$ cp -r include/asm-ppc/ \
> ${TARGET_PREFIX}/include/asm
$ cp -r include/asm-generic/ \
> ${TARGET_PREFIX}/include
```

- No need to rebuild toolchain in case of kernel change.

3. Binutils setup

- What's in binutils:

as:	**GNU assembler**
ld:	**GNU linker**
gasp:	GNU assembler preprocessor
ar:	Creates and manipulates archives
nm:	Lists symbols in an object file
objcopy:	Copies and translates object files
objdump:	Display info about content of object files
ranlib:	Generates index to archived content
readelf:	Displays info about ELF file
size:	Lists object file section sizes
strings:	Prints strings found in object file
strip:	Strips symbols from object files
addr2line:	Converts addresses to source file line nbrs
c++filt:	Converts asm-C++-labels to names

- Unpacking binutils:

```
$ cd ${PRJROOT}/build-tools
$ tar xvjf binutils-2.15.tar.bz2
```

- Configuring binutils:

```
$ cd build-binutils
$ ../binutils-2.15/configure \
> --target=$TARGET --prefix=${PREFIX}
```

- The `configure` script checks for the availability of host resources and prepares Makefiles for binutils' build.

- `configure` is controlled using parameters
- `--target=` is used to specify the target's arch
- `--prefix=` is used to specify the location where the binutils should be installed.
- When configure finishes, we can build binutils:

```
$ make
```

- Build takes is relatively fast on average hardware
- Installing binutils:

```
$ make install
```

- Looking at the tools installed:

```
$ ls ${PREFIX}/bin
powerpc-linux-objdump     powerpc-linux-ranlib
powerpc-linux-addr2line   powerpc-linux-readelf
powerpc-linux-ar          powerpc-linux-ld
powerpc-linux-size        powerpc-linux-as
powerpc-linux-nm          powerpc-linux-strings
powerpc-linux-objcopy     powerpc-linux-strip
powerpc-linux-c++filt
```

4. Installing glibc headers

- Necessary step if using gcc >3.2
- Unpacking glibc:

```
$ cd ${PRJROOT}/build-tools
$ tar xvjf glibc-2.3.2.tar.bz2
```

- Configuring glibc:

```
$ cd build-glibc-headers
$ CC=gcc32 ../glibc-2.3.2/configure \
> --host=$TARGET --build=i686-linux \
> --prefix="/usr" --disable-sanity-checks \
> --with-headers=${TARGET_PREFIX}/include
$ make sysdeps/gnu/errlist.c
$ mkdir -p stdio-common
$ touch stdio-common/errlist-compat.c
```

- Install headers:

```
$ make cross-compiling=yes \
> install_root=${TARGET_PREFIX} prefix="" \
> install-headers
```

- `cross-compiling` must be set to “yes” because we are not providing a cross-compiler to build glibc.
- Need to create dummy stubs.h file for gcc:

```
$ mkdir -p ${TARGET_PREFIX}/include/gnu
$ touch ${TARGET_PREFIX}/include/gnu/stubs.h
$ cp bits/stdio_lim.h \
> ${TARGET_PREFIX}/include/bits/
```

5. Bootstrap compiler setup

- The bootstrap compiler supports the C language only.
- Unpacking gcc:

```
$ cd ${PRJROOT}/build-tools
$ tar xvjf gcc-3.3.2.tar.bz2
```

- Configuring gcc:

```
$ cd build-boot-gcc
$ CC=gcc32 \
> ../gcc-3.3.2/configure –-target=$TARGET \
> --prefix=${PREFIX} --disable-shared \
> --disable-threads \
> --with-headers=${TARGET_PREFIX}/include \
> --with-newlib --enable-languages=c
```

- `--target` and `--prefix` same as before
- Usually, no headers for target yet means use of `--without-headers`, but new tools broken.
- Force gcc not to rely on glibc: `--with-newlib`. We can still use glibc later though.
- Tell gcc to build support for C only: `--enable-languages=`.
- Build compiler:

```
$ make all-gcc
```

- Install gcc:

```
$ make install-gcc
```

6. C library setup

- Most delicate and complicated part of the build
- For all practical purposes “C library” == glibc, though the C library is actually only one of the libraries in glibc. More on this later.
- glibc already unpacked during glibc headers install.
- Unpacking glibc-linuxthreads

```
$ cd ${PRJROOT}/build-tools
$ tar -xvjf glibc-linuxthreads-2.3.2.tar.bz2 \
> --directory=glibc-2.3.2
```

- Configuring glibc:

```
$ cd build-glibc
$ CC=powerpc-linux-gcc \
> ../glibc-2.3.2/configure \
> --host=$TARGET --build=i686-linux \
> --prefix="/usr" --enable-add-ons \
> --with-headers=${TARGET_PREFIX}/include
```

- We set the CC environment variable to point to the bootstrap compiler we built earlier.
- In this case, we use `--host` instead of `--target`, because glibc's host is the target, not our development host.

- Here, `--prefix` is used to tell glibc the location of the libraries as they will be seen on the target's root filesystem. “/usr” is a special value recognized by the glibc configure script for Linux systems.
- On the host, we will divert the library's install to a different directory, however.
- `--enable-add-ons` instructs the configuration script to take the added linuxthreads in account. Could also be explicit:

```
--enable-add-ons=linuxthreads
```

- `--with-headers` is used to instruct glibc as to the location of the system headers to use for its build. The default /usr/include is inappropriate.
- The build process creates three library sets which can be controlled to a certain extent:
  - Shared `--disable-shared`
  - Static Cannot be disabled
  - Profile `--disable-profile`
  - Use with care

- Other useful configuration options:

```
--enable-static-nss
--without-fp
CFLAGS="-msoft-float -D_SOFT_FLOAT -O2 -mcpu=X"
--disable-sanity-checks
```

- May need to apply patches to fix glibc.
- For ex.: glibc 2.3.2, must apply Dan Kegel's sscanf patch found at (copy to build-tools directory):

  http://www.kegel.com/crosstool/current/patches/glibc-2.3.2/sscanf.patch

- Applying the patch:

```
$ cd ${PRJROOT}/build-tools/glibc-2.3.2
$ patch -p1 < ../sscanf.patch
```

- Building glibc:

```
$ cd ${PRJROOT}/build-tools/build-glibc
$ make
```

- Build is fairly long
- Build failure may require `make clean` and then `make` again.
- Installing glibc:

```
$ make install_root=${TARGET_PREFIX} \
> prefix="" install
```

- The `install_root` variable is prepended to the default install path (/usr).

- By setting `prefix` to “”, we force the Makefiles to ignore the “/usr” we had used on the configuration command line.
- Using ${TARGET_PREFIX} as the value for `install_root`, all library components will be placed in ${TARGET_PREFIX}/lib.
- Final step: Correct glibc's link script
- Link script: ${TARGET_PREFIX}/lib/libc.so:

```
/* GNU ld script
   Use the shared library, but some functions are only in
   the static library, so try that secondarily.  */
OUTPUT_FORMAT(elf32-powerpc)
GROUP ( /lib/libc.so.6 /lib/libc_nonshared.a )
```

- Must remove “/lib/” to force linker to look for libraries in directory were libc.so is located: ${TARGET_PREFIX}/lib
- Make copy of original libc.so:

```
$ cd ${TARGET_PREFIX}/lib
$ cp ./libc.so ./libc.so.orig
```

- Edit and modify libc.so:

```
/* GNU ld script
   Use the shared library, but some functions are only in
   the static library, so try that secondarily.  */
OUTPUT_FORMAT(elf32-powerpc)
GROUP ( libc.so.6 libc_nonshared.a )
```

7. Full compiler setup

- Compiler with C++ support
- No need to unpack gcc since it was already unpacked for the bootstrap compiler build.
- Configuring gcc:

```
$ cd ${PRJROOT}/build-tools/build-gcc
$ CC=gcc32 \
> ../gcc-3.3.2/configure --target=$TARGET \
> --prefix=${PREFIX} --enable-languages=c,c++
```

- Notice that the command line is simpler
- Building gcc:

```
$ make all
```

- Installing gcc:

```
$ make install
```

8. Finalizing the toolchain setup

- Let's take a look at what has been installed
- Content of ${PRJROOT}/tools:

bin:	the cross-development utilities
powerpc-linux:	target-specific files
include:	headers for cross-development tools
info:	info pages for gcc
lib:	libraries for cross-development tools
man:	man pages for cross-development tools
share:	empty

- Content of ${PREFIX}/powerpc-linux:

bin:	glibc-related binaries and scripts
etc:	files for target's /etc directory
include:	target headers
info:	glibc info pages
lib:	target's libraries
libexec:	binary helpers
sbin:	files for target's /sbin directory
share:	files and subdirectories for internationalization
sys-include:	unused

- The libraries directory is fairly large (> 80 MB)
- Some host binaries have been misplaced in the ${TARGET_PREFIX}/bin directory.

- Moving misplaced binaries:

```
$ cd ${TARGET_PREFIX}/bin
$ mv as ar c++ g++ gcc ld nm ranlib strip \
> ${PREFIX}/lib/gcc-lib/powerpc-linux/3.3.2
```

- Making symbolic links to misplaced binaries:

```
$ for file in as ar c++ g++ gcc ld nm ranlib strip
> do
> ln -s ${PREFIX}/lib/gcc-lib/powerpc-linux/3.3.2/$file .
> done
```

- **The toolchain is now ready to be used!**
- Avoid the temptation of deleting ${PRJROOT}/build-tools right away ...

9. Using the toolchain

- Example Makefile:

```
# Tool names
CROSS_COMPILE = ${TARGET}-
AS            = $(CROSS_COMPILE)as
AR            = $(CROSS_COMPILE)ar
CC            = $(CROSS_COMPILE)gcc
CPP           = $(CC) -E
LD            = $(CROSS_COMPILE)ld
NM            = $(CROSS_COMPILE)nm
OBJCOPY       = $(CROSS_COMPILE)objcopy
OBJDUMP       = $(CROSS_COMPILE)objdump
RANLIB        = $(CROSS_COMPILE)ranlib
READELF       = $(CROSS_COMPILE)readelf
SIZE          = $(CROSS_COMPILE)size
STRINGS       = $(CROSS_COMPILE)strings
STRIP         = $(CROSS_COMPILE)strip

export AS AR CC CPP LD NM OBJCOPY OBJDUMP RANLIB READELF \
         SIZE STRINGS STRIP

# Build settings
CFLAGS        = -O2 -Wall
HEADER_OPS    =
LDFLAGS       =
```

```
# Installation variables
EXEC_NAME     = example-app
INSTALL       = install
INSTALL_DIR   = ${PRJROOT}/rootfs/bin

# Files needed for the build
OBJS          = example-app.o

# Make rules
all: example-app

.c.o:
        $(CC) $(CFLAGS) $(HEADER_OPS) -c $<

example-app: ${OBJS}
        $(CC) -o $(EXEC_NAME) ${OBJS} $(LDFLAGS)

install: example-app
        test -d $(INSTALL_DIR) || $(INSTALL) -d -m 755 $
(INSTALL_DIR)
        $(INSTALL) -m 755 $(EXEC_NAME) $(INSTALL_DIR)

clean:
        rm -f *.o $(EXEC_NAME) core

distclean:
        rm -f *~
        rm -f *.o $(EXEC_NAME) core
```

- To compile natively with this Makefile:

```
$ make CROSS_COMPILE=""
```

10. C library alternatives

- glibc is quite large and contains many features, some of which are of no use in an embedded system.
- Main alternatives:
  1. uClibc (LGPL)
  2. eglibc (LGPL)
- These alternatives can effectively replace glibc
- Other alternatives:
  - diet libc (GPL ... !?!)
  - Bionic (BSD)

10.1. uClibc

- Originates from uClinux effort
- Support both CPUs that have and those that lack an MMU and/or an FPU.
- Allows both static and dynamic linking
- Most applications that build with glibc will build the same with uClibc.
- Available from: http://uclibc.org/
- Download that version to your ${PRJROOT}/build-tools directory and extract it.

10.1.1 Setting up uClibc

- Move to the directory where uClibc was extracted:
  ```
  $ cd ${PRJROOT}/build-tools/uClibc-0.9.21
  ```
- Start the uClibc configuration menu:
  ```
  $ HOSTCC=gcc32 \
  > make CROSS=powerpc-linux- menuconfig
  ```
- Whereas the kernel relies on CROSS_COMPILE, uClibc relies on CROSS to get the cross-development tools prefix.
- Main configuration submenus:
  - Target Architecture Features and Options
  - General library settings
  - Networking support
  - String and stdio support
  - Big and Tall
  - Library and installation support
  - uClibc hacking options

- Use the “?” key for help
- Configuring uClibc for our workspace setup:
  - Linux kernel header location
    `${PRJROOT}/kernel/linux-2.4.20`
  - Shared library loader path
    `/lib`
  - uClibc development environment directory
    `${PRJROOT}/tools/uclibc`
  - uClibc development environment system directory
    `$(DEVEL_PREFIX)`
  - uClibc development environment tool directory
    `$(DEVEL_PREFIX)/usr`

- PREFIX is an internal uClibc variable and can't be used.
- Compiling uClibc:

```
$ make CROSS=powerpc-linux-
```

- Installing uClibc:

```
$ make CROSS=powerpc-linux- PREFIX="" install
```

10.1.2. Using uClibc

- Change path:

```
export PATH=${PREFIX}/bin:${PREFIX}/uclibc/bin:${PATH}
```

- Build command line:

```
$ make CROSS_COMPILE=powerpc-uclibc-
$ make CROSS_COMPILE=powerpc-uclibc- \
> LDFLAGS=”-static”
```

10.2. diet libc

- Written from scratch
- Favors static linking
- Does not support as many archs as uClibc
- **MAJOR PROBLEM: GPL license**
- Available from http://www.fefe.de/dietlibc/
- Have a look at BELS for complete details

10.3. Bionic

- Part of Android distribution
- BSD-licensed
- Based on BSD library
- Minimalistic C library
- Non-posix compliant
- Does not support System V IPC
- Does not fully support pthreads

11. Other programming languages

- Java:
  - OpenJDK
  - GNU java compiler
  - Dalvik
- Perl: patches, hacks available
- Python: relatively simple to cross-compile
- Ada: not used often, requires existing gnat
- ... Forth, FORTRAN, lisp, etc.

12. Cross-compiling the impossible

- Problem: Not all packages cross-compile easily
- Solution: Compile package straight on target
- Steps summary:
  - Build toolchain
  - Cross-compile the gcc compiler itself and all other necessary tools.
  - Boot target with NFS
  - Use native gcc to compile package on target
  - Remove native gcc from production rootfs

13. Auto-generating toolchains

- Toolchain generation is messy
- Best to use dedicated tool:
  - crosstool
  - crosstool-ng
  - Buildroot
  - PTXdist
  - OpenEmbedded
- Toolchain used in this class built with crosstool-ng

13.1. crosstool-ng

- Successor to crosstool
- Available at:

  http://ymorin.is-a-geek.org/projects/crosstool
- Downloads, patches, builds, installs, etc.
- Comprises **23** steps
- Menuconfig-based
- Supports uClibc, glibc and eglibc
- Supports ARM, Blackfin, MIPS, PowerPC, SH, …
- Fairly well maintained

- Must make sure the following are installed on Ubuntu in order to use crosstool-ng:
  - gawk
  - texinfo
  - automake
  - libtool
  - cvs
  - libncurses5-dev
  - bison
  - flex
- Use “sudo apt-get install” to get those

- Download and extract to ${PRJROOT}/build-tools
- Configure crosstool:

```
$ cd crosstool-ng-1.10.0/
$ ./configure
```

- Build and install crosstool-ng:

```
$ make
$ make install
```

- Configure crosstool:

```
$ cd ${PRJROOT}/build-tools
$ ct-ng menuconfig
```

- Options:
  - Paths->Prefix directory: ${PREFIX}/${CT_TARGET}
  - Target options->architecture: powerpc
  - OS->Target OS: linux
  - C library->C library: glibc
  - C library->Extra target CFLAGS:
    -U_FORTIFY_SOURCE
  - Debug facilities: gdb & strace
- Build the toolchain:

```
$ ct-ng build
```

Linux, the human factor

1. Reasons for choosing Linux
2. Players of the embedded Linux scene
3. Copyright and patent issues
4. Using distributions

1. Reasons for choosing Linux

- Quality and reliability of code
- Availability of code
- Hardware support
- Communication protocol and software standards
- Available tools
- Community support
- Licensing
- Vendor independence
- Cost

- Quality and reliability of code:
  - Quality code:
    - Modularity and structure
    - Ease of fixing
    - Extensibility
    - Configurability
  - Reliable code:
    - Predictability
    - Error recovery
    - Longevity
  - Most developers agree Linux fits this description
  - With time, quality and reliability have increased

- Availability of code:
  - No access restrictions or requirements on software
  - Distribution of the kernel and important tools (such as the GNU toolchain) under the GNU General Public License (GNU GPL).
  - Some projects under BSD (user-space networking daemons, Mozilla browser, etc.)
  - “Unaccessible” software is often or will eventually be replaced by open source and free software community.
  - Virtues of accessibility: rapid fixes, understanding of operation, vendor independence, etc.

- Hardware support:
  - Broad hardware support
  - Community maintenance means guarantee against hardware vendors' marketing strategies.
  - Support for over a dozen architectures!
  - Your hardware or similar hardware is likely already running Linux.
  - Porting “know-how” available and distributed
  - Very high software portability
  - Many instances of architecture-independent drivers

- Communication protocol and software standards:
  - Linux supports a wide range of existing communication protocols (Win emulation through Samba.)
  - Bleeding edge support for modern protocols (IPv6, Bluetooth, IrDA, etc.)
  - Provides Unix-replacement
  - Can run most Unix programs unchanged, including all GNU software.

- Available tools:
  - Slew of tools available for Linux
  - Most every tool you can think of already exists in some form.
  - These sites provide a good idea of what's out there:
    - http://www.freshmeat.net
    - http://www.sourceforge.net
  - Can't find what you're looking for:
    - Try google
    - ex.: “Linux tracing”, “Linux firewire”, etc.

- Community support:
  - Perhaps Linux's biggest advantage
  - Often others have had the same problem as you and will gladly share their solution with you.
  - Focal point for getting help with a software:
    - User and development mailing lists.
  - Expertise and help far outweighs any commercial offering.
  - Can often reach original software author directly.
  - Don't forget to provide the details though ... !

- Licensing:
  - You can: Use, Modify, and Redistribute
  - Nevertheless, must adhere to license requirements:
    - GPL, LGPL, BSD, MPL, etc.
  - Will discuss this further later
- Vendor independence:
  - No single code master (Lord of the Rings anyone?)
  - Can easily switch vendors because code is available
  - Careful about “value-added” software
  - No welded hood means you don't have to go to the dealership to get service.

- Cost:
  - None, it's free software remember
  - Traditionally, costs involve:
    - Basic developer licenses
    - Additional software development helpers
    - **Runtime royalties**
  - With Linux:
    - Basic GNU toolchain is free
    - All additional software development tools are free
    - There are no runtime royalties whatsoever

- May want to purchase distribution which provides:
  - Prepackaged toolset
  - BSP (Board Support Package)
  - Porting services
  - “Value-added” software
- IMPORTANT: You don't need a distribution to build embedded Linux systems. This course will show you how to make it on your own.

2. Players of the embedded Linux scene

- Types of players:
  - Free software and open source community
  - Industry
  - Organizations
  - Resources

- Free software and open source community:
  - Basis of all Linux development
  - Most important player
  - Members of this community are responsible for:
    - Enhancement
    - Maintenance
    - Support
  - Characteristics:
    - No central authority ... None
    - Heterogeneous
    - Care a great deal about software quality and reliability

- Industry:
  - Embrace and promote Linux's use in embedded systems.
  - Vendors *can* form an important link between users and the open source and free software community.
  - Vendors must help their clients, but must also contribute to projects to sustain them.
  - Examples of big vendors:
    - WindRiver
    - MontaVista
    - TimeSys
  - Many small players with equal or better services

- Organizations:
  - Linux Foundation:
    - Formerly OSDL
    - Big-name founders: IBM, Oracle, HP, etc.
    - Recently merged in CE Linux Forum
  - Linux / Unix:
    - Free Software Foundation (FSF) => GNU project, GPL, LGPL, etc.
    - OpenGroup => Single Unix Specification (SUS)
    - Linux Standard Base (LSB)
    - Filesystem Hierarchy Standard (FHS)
    - etc.

- Resources:
  - LinuxJournal
  - LinuxDevices.com
  - LWN
  - KernelNewbiews

3. Copyright and patent issues

- Many questions
- Textbook GPL
- Pending issues
- RTLinux patent

- Many questions:
  - What does using Linux in my system involve?
  - What exactly is in the GNU GPL?
  - What are all those other licenses?
  - Aren't there legal risks in using GPL software?
  - How do I make sure the software I develop remains the property of my company?
  - What should I look out for?
  - What about kernel extensions and proprietary drivers?
  - etc. (what questions do you have?)

- Textbook GPL:
  - Most Linux software, including the kernel, is distributed under the terms of this license.
  - Main uses:
    - Applications => GPL
    - Libraries => Lesser GPL (LGPL)
  - Beware of libraries distributed under the GPL!
  - Always make sure you know the license of every open source package you use ... **Always**.
  - GPL provides rights and imposes obligations. Essentially allowing modification and redistribution while guaranteeing the original author's rights.

- GPL provisions, rights, and requirements summary:
  - Can make unlimited copies => Must keep license and copyright intact.
  - GPL software comes with no warranty whatsoever
  - May charge for the act of copying or for providing warranty.
  - Can distribute software binaries => Must distribute source code with binaries, or provide an offer / means to get it.
  - Cannot place further restrictions on recipients other than those prescribed in the GPL.
  - Can modify and redistribute modified software as long as recipient gets same rights you received. In other words, you cannot distribute modified GPL software under any other license than the GPL.

- Confusion over the modification and distribution clauses.
- Issues to focus on:
  - Running the software
  - Modifying the software
- Running the unmodified software does not generate a derived work and does, therefore, entail any sort of “GPL contamination”.
- Modifying the software forbids you from redistributing the modified version under any license other than the GPL.
- **GPL contamination doesn't exist.**

- Some FUD plays on the confusion between running and modifying.
- Safeguard yourself by identifying your actions
- No difference between static linking and dynamic linking in regards to the law:
  - Dynamically linking your program with GPL'ed software creates a derived work
  - A program dynamically linked to GPL software cannot be distributed under any other license than the GPL.
- LGPL:
  - Allows using **unmodified** LGPL code in your application and distribution under your desired license.

- Modifying LGPL software, however, falls under similar rules as the GPL.
- Example: Modifications to glibc cannot be distributed under any license other than the LGPL, or the GPL at your choice.
- As with the GPL, there's a difference between linking and modifying.

- Static linking with LGPL software requires providing original object files for linking against alternative libraries.

- Pending issues:
  - Packaging binary copies of GPL software in your embedded system is a textbook case of GPL.
  - Some traditional embedded vendors cast doubt about packaging of GPL apps with Linux because of “tight coupling”. This is unfounded, as we shall see.
  - Linux is GPL and applications running on Linux are in some form of linking, so are they derived works?
    - NO. Linus has explicitly excluded applications from the applicability of the kernel's GPL.

- From `linux/COPYING`:

NOTE! This copyright does *not* cover user programs that use kernel services by normal system calls - this is merely considered normal use of the kernel, and does *not* fall under the heading of "derived work". Also note that the GPL below is copyrighted by the Free Software Foundation, but the instance of code that it refers to (the Linux kernel) is copyrighted by me and others who actually wrote it.

Also note that the only valid version of the GPL as far as the kernel is concerned is _this_ particular version of the license (ie v2, not v2.2 or v3.x or whatever), unless explicitly otherwise stated.

Linus Torvalds

- Up to here, everything is cristal clear
- Total lack of clarity about “binary modules”
- Modules are OS functionality component add-ons which can be loaded dynamically at runtime into the kernel's memory space / address space.
- Example modules: drivers, filesystems, networking protocol extensions, etc.
- Summary: Modules allow easy kernel extension
- Problem: They are linked as part of the kernel and should therefore be subject to the GPL.
- Reality: Many vendors have already been shipping binary-only modules for quite some time.

- L. Torvalds accepts modules as long as they are not Linux-specific.
- Some developers question his ability to allow/deny such modules.
- As of today, there is no definitive clear status for binary-only modules.
- Consensus is that it's a contentious issue :-)
- Safest recommendation: **GPL modules only!**

4. Using distributions

- To use or not to use
- How to choose a distribution
- What to avoid doing with a distribution

- To use or not to use:
  - You don't need a distribution to build an embedded Linux system.
  - All the required software is available free for download.
  - This course and the book show how to build an embedded Linux system straight from source.
  - Use a dis
  - This method can initially be more time-consuming, but does ensure autonomy.
  - Use a distribution if:

    a) You want to have an automated build procedure

    b) Would like to benefit from others' lessons learned

    c) Need to have commercial-grade support

- How to choose a distribution:
  - Open source vs. “proprietary” distribution:
    - Open source: Available from the Net and maintained by open source developers.
    - “Proprietary”: Available from vendor and maintained as part of for-profit operations.
  - Licensing issues:
    - Vendor licenses?
    - Some distros: open source and “value added”
  - Make a checklist:
    - Host: GNU toolchain, debugging tools
    - Target: Kernel, drivers, and applications

- What's the support like and who gives it?
  - Phone number, email, mailing list, or all of them?
  - Vendors are the best place to get information about their products. Support for vendor products on open source mailing lists is very low.
  - Vendors responsible for updates and bug fixing
- Reputation (Do they actually contribute?)
- Documentation, a rare commodity
- Installation
- Configurability
- Specific target support

- What to avoid doing with a distribution:

  At all cost, avoid using a distribution in a way that will force you to continue using that same distribution forever.

- Have a backup plan if you intend to use a distribution-specific capability / software.

Linux basics

1. Logging in
2. Virtual consoles
3. Basic commands
4. Shells
5. Useful keys
6. Word completion
7. Filename expansion
8. Redirect output
9. Paths
10. Background mode
11. Manual pages
12. File perms/own
13. Changing perms/own
14. Startup files
15. Important directories
16. Processes
17. Editors
18. GUIs
19. Networking
20. Serial device

1. Logging in

- Linux login:

```
localhost login:
```

- Enter your user name:

```
localhost login: my_name
```

- Password prompt (no echo):

```
Password:
```

- Session:

```
[my_name@localhost my_name] $
```

- Your home directory:

```
$ pwd
/home/my_name
```

- Changing passwords (no echo):

```
$ passwd
Changing password for user my_name.
Changing password for my_name
(current) UNIX password:
New password:
Retype new password:
passwd: all authentication tokens updated
  successfully.
```

2. Virtual consoles

- Console: Main keyboard and screen used for input and output (`/dev/console`).
- Virtual consoles: Abstractions created by the OS to allow multiple input/output sessions simultaneously.
- By default, there are 8 virtual consoles on most Linux distributions.
- Browsing through consoles:
  - ALT + F1 to F8

3. Basic commands

- Printing working directory:

```
$ pwd
/home/karim
```

- Changing directories:

```
$ cd tmp
$ pwd
/home/karim/tmp
```

- Creating directories:

```
$ mkdir ~/example-sys
$ cd ~
```

- Deleting directories:

```
$ rmdir tmp
```

- Listing directory content:

```
$ ls
tmp    example-sys
$ ls -al
-rw-r--r--  1 karim karim  191 Jun 21 00:04 .bash_profile
-rw-r--r--  1 karim karim  124 Jun 21 00:04 .bashrc
-rw-r--r--  1 karim karim  124 Jun 21 00:04 welcome
drwxrwxrw- 39 karim karim 4096 Jun 26 22:58 tmp
drwxrwxrw- 39 karim karim 4096 Jun 26 22:58 example-sys
$ ls -ld *
drwxrwxrw- 39 karim karim 4096 Jun 26 22:58 tmp
drwxrwxrw- 39 karim karim 4096 Jun 26 22:58 example-sys
```

- Dumping a file's content:

```
$ cat welcome
Hello, nice to see you!
Hope to see you soon.
```

- Creating an empty file / updating a file's timestamp:

```
$ touch bin-image
```

- Printing to standard output:

```
$ echo Hello World!
Hello World!
```

- Scrolling file contents:

```
$ less welcome
Hello, nice to see you!
Hope to see you soon.
(welcome) END
$ more welcome
```

- Creating symbolic links (files and directories):

```
$ ln -s welcome hello-message
$ ls -l hello-message
lrwxrwxrwx 1 karim karim 7 Jun 28 14:46 hello-message -> welcome
```

- Copying files:

```
$ cp bin-image bin-image-030427
```

- Deleting files and directories:

```
$ rm bin-image
$ rm -rf example-sys
```

- Moving files and directories:

```
$ mv bin-image-030427 bin-image
```

- Finding out disk usage:

```
$ du -sk example-sys
7612                    example-sys
```

- Finding out disk information:

```
$ df
Filesystem  1K-blocks      Used Available Use% Mounted on
/dev/hda1    10080488   5393032   4175388  57% /
```

- Sorting content:

```
$ sort -r welcome
Hope to see you soon.
Hello, nice to see you!
```

- Viewing environment variables:

```
$ env
```

- Modifying/adding environment variables:

```
$ export BIN_IMAGE=bin-image
$ echo $BIN_IMAGE
bin-image
```

- Creating command aliases:

```
$ alias dir="dir --color"
```

- Logging in as root:

```
$ su
```

- Logging in as another user:

```
$ su - other_user
```

- Manipulating files:

```
$ dd if=/dev/zero of=bin-image \
> count=100 skip=50
```

- Mounting filesystem (usually needs root):

```
# mount -t vfat /dev/hda2 /mnt/dos
# mount
/dev/hda1 on / type ext3 (rw)
/dev/hda2 on /mnt/doc type vfat (rw)
```

- Unmounting filesystems:

```
# umount /dev/hda2
```

- Creating filesystem type:

  # **mkfs -t ext2 /dev/hda3**

- Checking filesystem type:

  # **fsck /dev/hda3**

- Creating swap type:

  # **mkswap /dev/hda4**

- Enabling swap:

  # **swapon /dev/hda4**

- Disabling swap:

  # **swapoff /dev/hda4**

- Creating device nodes:

```
# mknod -m 660 /dev/mydev c 67 251
# ls -l /dev/mydev
crw-rw---- 1 root root  67,   251 Jan 30 05:24 /dev/mydev
```

4. Shells

- Provides command-line user interface
- Most often used shell:
  - bash, Bourne Again shell
- Other shells:
  - csh, C shell
  - ksh, Korn shell
  - sh, Bourne shell
  - tcsh, Enhanced C shell
  - zsh, Z shell
- Use `chsh` to change shells

5. Useful keys

- Allows easy control over interactive commands
- Interrupt process: CTRL-C
- Suspend process: CTRL-Z

6. Word completion

- Some filenames and directory names are tedious to type by hand.
- The shell's word completion allows automatic completion of names.
- To use word completion, use the TAB key:

```
$ ls bin[TAB]-image
```

7. Filename expansion

- Sometimes need to apply commands to many files with similar names.
- Examples of filename expansion:

```
$ ls /dev/hda*
$ rm -rf tmp/bin-image-030[1-9]*
$ ls sources/*.c
$ rm -rf sources/*.o
$ cp sources/*.[co] tmp
```

8. Redirect output

- Saving standard output (>):

```
$ df > free-space
```

- Appending standard output (>>):

```
$ df >> free-space-log
```

- Saving all output (>&):

```
$ gcc myprog.c >& gcc-output
```

- Saving standard error (2>):

```
$ gcc myprog.c 2> gcc-errors
```

- Feeding standard output to other programs (|):

```
$ cat phone-list | sort
```

- Saving standard output and error separately:

```
$ gcc myprog.c 2> gcc-errors > stdout
```

- Redirecting standard error to standard output:

```
$ gcc myprog.c 2>&1
```

- Saving all output:

```
$ gcc myprog.c 2>&1 | tee stdall
```

9. Paths

- Paths determine where the shell and other tools look for commands and files.
- Main command path:

```
$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/karim/bin
```

- Changing the path in bash:

```
$ export PATH=$PATH:new_path
```

OR

```
$ export PATH=new_path:$PATH
```

10. Background mode

- Starting processes in the background allows reusing the command line immediately without waiting for the command's completion.
- Examples of background mode:

```
$ xemacs . &
$ rm -rf tmp/* &
```

- To push an already running task in the background, use CTRL-Z and then the bg command.

11. Manual pages

- Almost every command in Linux/Unix has a man page.
- Man pages also exist for C library functions
- To read the man page for a command:

```
$ man cp
$ man rm
```

- Some names have more than one man page entry:

```
$ man 1 printf
$ man 3 printf
```

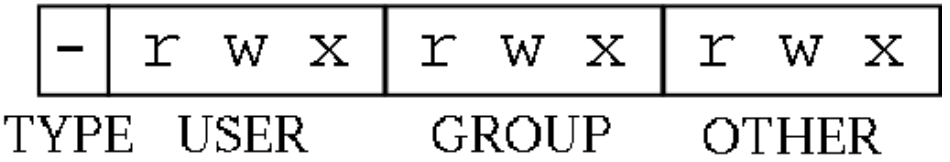
- Finding out if a man page discusses a topic you're interested in:

```
$ apropos cookies
CGI::Cookie       (3pm)  - Interface to Netscape Cookies
HTTP::Cookies     (3pm)  - Cookie storage and management
mcookie           (1)  - generate magic cookies for xauth
```

12. File permissions and ownership

- There are 3 types of permissions for Unix files:
  - Read
  - Write
  - Execute
- There are 3 levels of permissions in Unix:
  - User
  - Group
  - Other
- Each file or directory in a Unix system has permission rights for each level of permission.

- Common display of rights:



- Examples:

```
-rw-r--r--  1 karim admin     124 Jun 21 00:04 acc-info
```

RW for karim user, R for admin group, R for other

```
-rwxr-xr-x  1 root  root    67668 Feb 18 12:19 /bin/ls
```

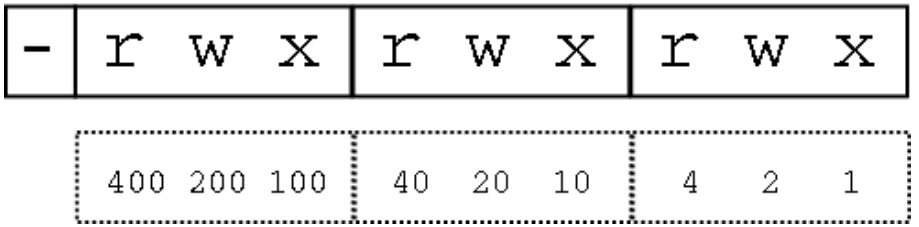
RWX for root user, RX for root group, RX for other

- Files types:
  - “-”: plain
  - “d”: directory
  - “l”: link
  - “b”: block device
  - “c”: char device
  - “s”: socket
  - “p”: named pipe

- A file always belongs to one user and one group
- Access to the file is granted according to its permissions.

13. Changing permissions and ownership

- Numerical method for describing permissions:



- Examples:
  - `-rw-r--r--` 400 + 200 + 40 + 4 = 644
  - `-rwxr-xr-x` 400 + 200 + 100 + 40 + 10 + 4 + 1 = 755
- Changing permissions:

```
$ chmod u+x ./myscript
$ chmod 755 ./myscript
$ chmod go+r tmp
```

- Changing user ownership:

```
$ chown jim example-sys
```

- Changing group ownership:

```
$ chgrp admin example-sys
```

- Changing group and ownership:

```
$ chown jim:admin example-sys
```

14. Startup files

- Some files are read and executed by applications at startup.
- .bashrc: Script for bash. Runs every time bash is started
- .bash_profile: Script for bash. Runs only when the instance of bash is started upon login.
- .cshrc: Script for C shell or tcsh. Same behavior as .bashrc.
- .login: Script for C shell or tcsh. Same behavior as .bash_profile.

- .emacs: Functions and initialization for the emacs editor.
- .exrc: For initializing the vi editor
- .Xdefaults: Configuration for programs using X Window.
- .xinitrc: Script run at X11 startup

15. Important directories

- The directories in a Linux / Unix root filesystem have specific uses.
- /bin: essential user binaries
- /sbin: essential sysadmin binaries
- /boot: bootloader and kernel images
- /etc: system configuration and startup files
- /var: variable data stored by daemons
- /usr: user programs and documentation
- /dev: device nodes
- /proc: virtual filesystem for kernel information

16. Processes

- Each task in the system is a different process
- Each process has a unique ID, the *PID*
- Each process has a parent, and *init* is the ancestor of all tasks running in the system.
- Tasks running on the system can be viewed using the `ps` command:

```
$ ps
  PID TTY          TIME CMD
 8705 pts/2    00:00:00 bash
 8758 pts/2    00:00:45 xemacs
 9029 pts/2    00:00:01 xemacs
11119 pts/2    00:00:00 ps
```

```
$ ps aux
USER PID %CPU %MEM    VSZ  RSS TTY STAT START   TIME COMMAND
root   1  0.0  0.0   1388  504 ?   S    Jun27   0:05 init
root   2  0.0  0.0      0    0 ?   SW   Jun27   0:01 [keventd]
root   3  0.0  0.0      0    0 ?   SWN  Jun27   0:00 [ksoftir...
root   4  0.0  0.0      0    0 ?   SWN  Jun27   0:00 [ksoftir...
root   5  0.0  0.0      0    0 ?   SW   Jun27   0:03 [kswapd]
root   6  0.0  0.0      0    0 ?   SW   Jun27   0:00 [bdflush]
root   7  0.0  0.0      0    0 ?   SW   Jun27   0:01 [kupdated]
root   9  0.0  0.0      0    0 ?   SW   Jun27   0:03 [kjournald]
root  59  0.0  0.0      0    0 ?   SW   Jun27   0:00 [khubd]
root 777  0.0  0.0      0    0 ?   SW   Jun27   0:00 [kjournald]
root 842  0.0  0.0      0    0 ?   SW   Jun27   0:00 [knodemg...
        ...
```

- Processes can be notified of important system events via the signal mechanism available in all Unix variants. Different signals are used for different events.
- Signals can be sent to processes by:
  - The kernel
  - Other processes
  - The operator
- The operator can send signals using the `kill` command.

- Sending the TERM signal to a processes:

```
$ kill 1247
```

- Sending the KILL signal to a process:

```
$ kill -9 1247
```

- A properly behaving process should exit when receiving the TERM signal.
- The KILL signal is handled by the kernel and will terminate the process wither it responds to other signals or whether it doesn't. **Use with care.**

17. Editors

- There are quite a few editors for use with Linux
- The easiest to use: nano
- The “most difficult” to use: vi
- The most powerful and flexible: emacs / xemacs
- If you aren't familiar already with either emacs or vi, you may want to stick with pico for the time being.
- A rudimentary understanding of vi is highly recommended.

18. GUIs

- Most Linux GUIs are based on the X Window System (X11).
- The main Linux GUIs:
  - KDE (K Desktop Environment): Most commonly used GUI. Built on Trolltech's Qt. Created a lot of debate in the OSS community a few years back because of Qt's license. One of the reasons behind the Gnome project.
  - Gnome: The GUI for the purists. Based on GTK, which is completely open source. Has been endorsed by Sun for its desktops. Has a lot of very interesting ideas behind it.

- Ubuntu's install sets the desktop to Gnome, unless you've chosen otherwise.

19. Networking

- Unlike other devices, the networking interface does not have an entry in /dev.
- Controlling the networking interface is done using the `ifconfig` command.
- Each workstation should have at least two interfaces:
  - loopback: this is a loop back to your system
  - eth0: the first Ethernet interface
- Bringing up both interfaces:

```
# ifup lo
# ifup eth0
```

20. Serial device

- What Windows usually identifies as COM1, is seen in Linux as /dev/ttyS0.
- Access rights to this file are not always properly set to enable non-root user to access the serial port.
- For now, you can try running `minicom` as root to test the serial port.