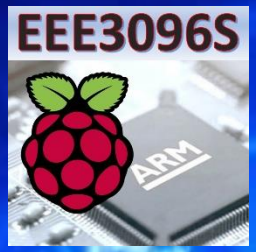
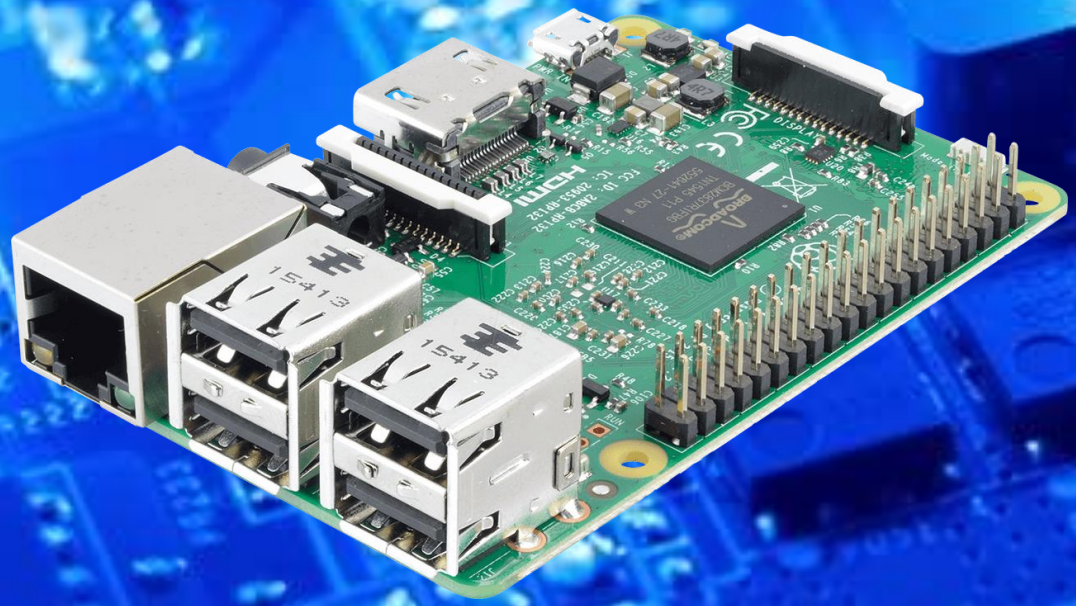




The Linux Kernel Design & Raspbian P02

Embedded Systems II

P2

Dr Simon Winberg

Electrical Engineering
University of Cape Town

Most of this content won't be examined but it could help with the pracs

Outline of Lecture P02

- The design of Linux and the Linux Kernel
- Linux Kernel Source Directory Tree
- About Prac 2
- Building the Raspbian kernel from sources

History of Linux

- UNIX: 1969 Thompson & Ritchie AT&T Bell Labs
- BSD: 1978 Berkeley Software Distribution
- Commercial: Sun, HP, IBM, SGI, DEC.
- GNU: 1984 Richard Stallman, Free Software Foundation (FSF).
- POSIX compatability standard: 1986
  IEEE Portable Operating System unIX
- Minix: 1987 Andy Tannenbaum
  A mini (bare essentials) POSIX compatible Unix
  Still available, maintained and in use (open source)
- SVR4: 1989 AT&T and Sun
  (prior versions but this was the most popular)
  SVR5 cancelled, the product range largely abandoned since 2005
- **Linux: 1991 Linus Torvalds Intel 386 (i386)**
- Open Source: GPL.

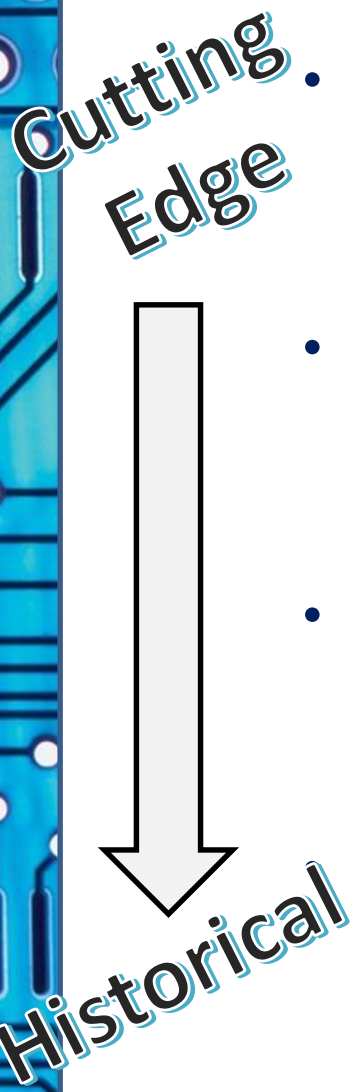
Maintainers of Linux

- The Linux kernel is currently maintained by
  - Linus Torvalds and a few hundred developers
  - Releases are numbered in a very ordered fashion:

    major.minor.patchlevel

  Odd minor number = development kernels
  latest version: 4.17.11 (i.e. dev version)

Linux Kernel Categories



- Pre-patch / RC “Release Candidate” kernels
  - Pre-releases for the mainline kernel (aimed at other kernel developers and Linux enthusiasts).
  - Compiled from source, usually contain new features.
- Mainline
  - The tree where all new features are introduced and where all the exciting new development happens.
  - New mainline kernels released +- every 2-3 months.
- Stable
  - A period of time after each mainline kernel is released, it becomes "stable." Bug fixes for stable are backported to the mainline tree
- Longterm
  - These are mainly for backporting bugfixes to older kernel trees.

Find out the kernel are you using??

uname r

Brief Aside: Superfast booting

Superfast booting that you can achieve with Linux

Short Video

Features of Linux

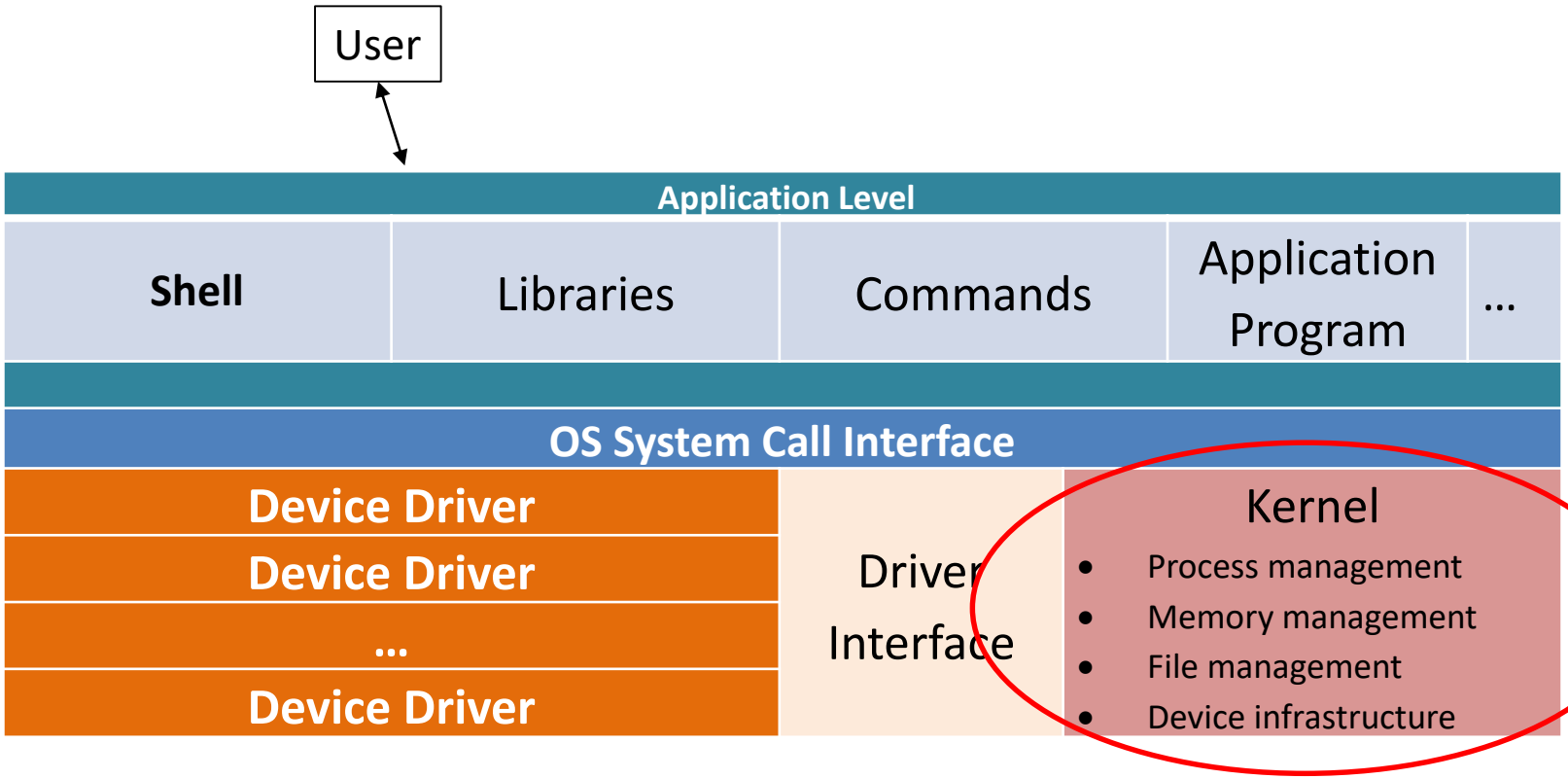
- UNIX-like operating system
- Features:
  - Pre-emptive multitasking
  - Virtual memory (protected memory, paging)
  - Shared libraries
  - Demand loading, dynamic kernel modules
  - Shared copy-on-write executables
  - TCP/IP networking
  - SMP* support
  - Open source

* SMP = symmetric multiprocessing

What exactly is an OS kernel?

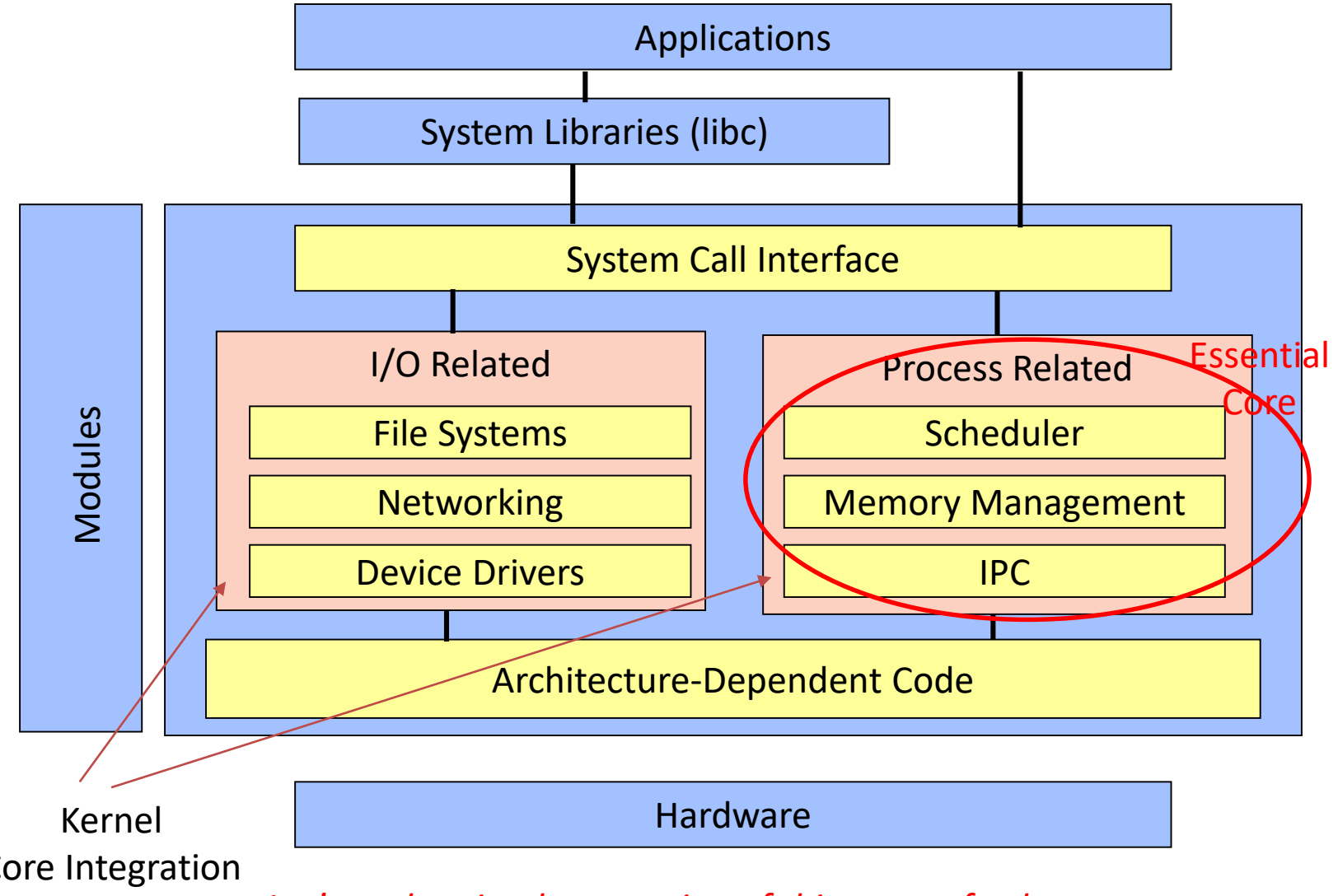
- You could call it the processes ‘executive’ or system arbitration core
- The Kernel Core
  - Controls and mediates access to hardware
- Implements and supports fundamental abstractions:
  - Processes, files, devices etc.
- Schedules / allocates system resources:
  - Memory, CPU, disk, descriptors, etc.
- Enforces security and protection.
- Responds to requests for service (system calls).
- Among other things – which one may put inside the kernel or decide to move outside the kernel

Linux OS Simplified View



Let's explore design considerations of this aspect further…

Example Kernel Design



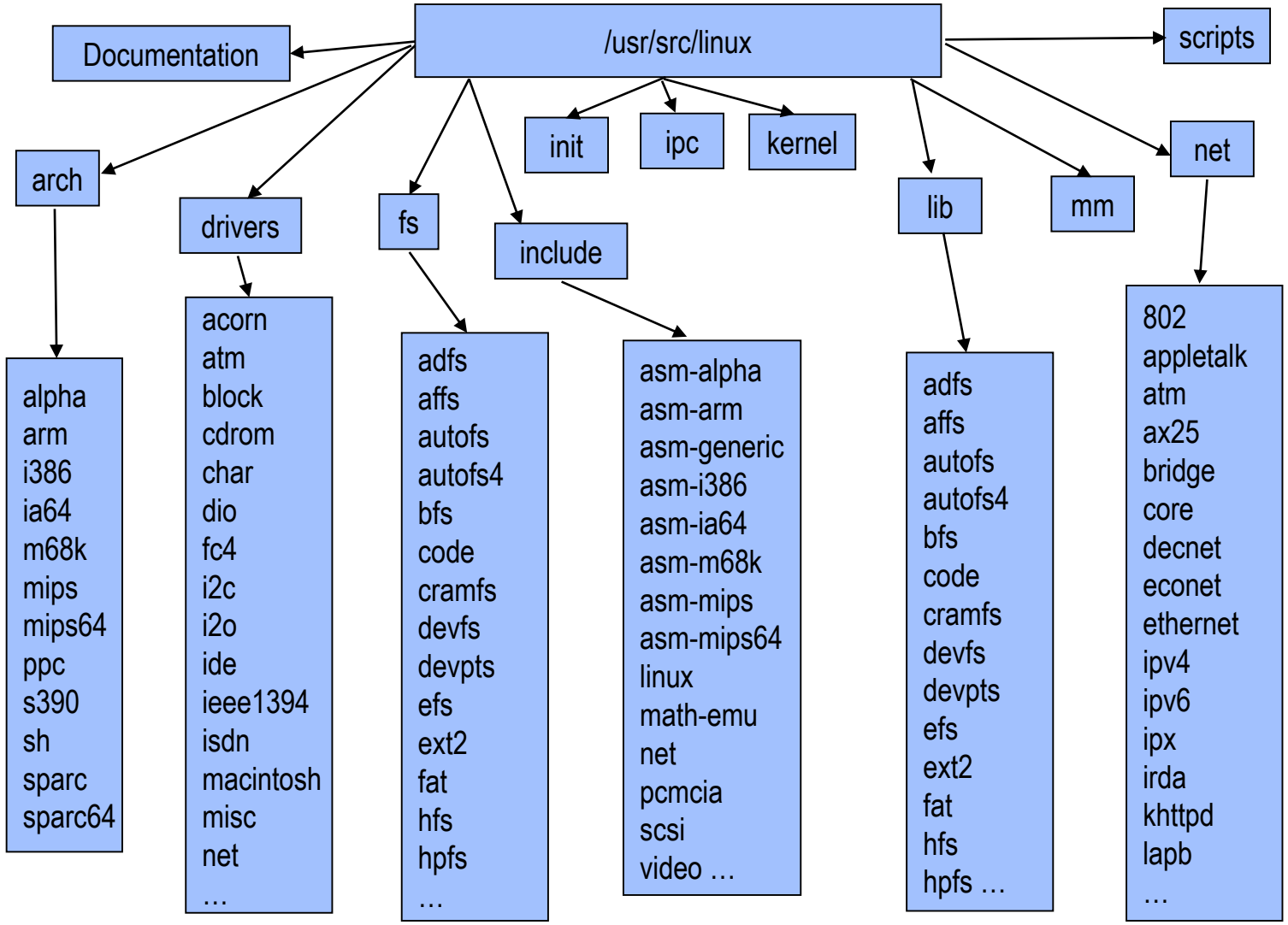
Let's explore implementation of this aspect further…

Kernel Structure

The Linux design follows a separation of concerns principle, as is usual for big systems.

That is to say an effort has been made to: categorize modules according to services they support, how the work and their security level.

Linux Kernel Source Directory Tree



Let's explore each of these top-level directories…

Directory: linux/arch

- This contains subdirectories for kernel porting overrides. Essentially works like subclasses.
- Each contains
  kernel, **lib**, **mm**, **boot** and other directories whose contents override code stubs in architecture independent code.
- **lib** contains highly-optimized common utility routines such as memcpy, checksums, etc.
- **arch** has overrides for processor architectures such as:
  - arm, i386, ia64, m68k, mips, mips64 and others.

Directory: linux/drivers

- The driver directory has the most code of the kernel (few Mb)
- Includes **device**, **bus**, **platform** and **general** directories.
- drivers/char
  - Character devices (e.g. n_tty.c basic serial driver)
- drivers/block
  - Block transfer devices (e.g. ll_rw_blk.c r/w requests to a block device)
- drivers/net
  - Network or streaming typed devices, e.g. Ethernet drivers
- drivers/scsi
  - SCSI devices
- General
  - Basically like a “misc” category. Includes: cdrom, ide, isdn, parport, pcmcia, pnp, sound.
- Buses
  - fc4, i2c, nubus, pci, sbus, tc, usb

Directory: linux/fs

- Directory contains file system implementations
- Contains:
  - Linux's virtual filesystem (VFS) framework
  - subdirectories for actual filesystems
- vfs-related files: (note these closely match Linux POSFIX file operations)
  - exec.c, binfmt_*.c - files for mapping new process images.
  - open.c, read_write.c, select.c, pipe.c, fifo.c.
  - fcntl.c, ioctl.c, locks.c, dquot.c, stat.c. …

Directory: linux/include

- Common includes needed by various modules
- include/asm-*:
  - Architecture-dependent include subdirectories for assembly code
- include/linux:
  - Header info needed both by the kernel and user apps (e.g. process configuration, etc).
- Other directories:
  - math-emu : floating point math emulation
  - net, pcmcia, scsi, video

Directory: linux/init

- This has only two files:
  - version.c: contains the version banner that prints at boot.
  - main.c: architecture-independent boot code.
- start_kernel is the primary entry point.

Change at your peril!

Of course you may have to change things here to make it boot

Directory: linux/kernel

- This directory has the core kernel code
- sched.c
  - Essentially the main kernel file, its '*main*' function. Implements:
  - Scheduler, timers, task queues, alarms (e.g. core dump)
- Process control:
  - fork.c, exec.c, signal.c, exit.c etc…
- Kernel module support:
  - kmod.c, ksyms.c, module.c.
- Other operations:
  - time.c, resource.c, dma.c, softirq.c, itimer.c.
  - printk.c, info.c, panic.c, sysctl.c, sys.c.

* Note there is not really a main() function, the processing will basically jump around, the threads won't really 'return' to this main function but rather go through an intermediate context switching operation not necessarily in main().

Directory: linux/lib

- These are commonly used routines
- The kernel code cannot call the standard C, because it is at a lower level (otherwise you could have a chicken and egg problem).
- Linux Lib files:
  - brlock.c – “Big Reader” just a for-loop to read
  - cmdline.c – kernel command line parsing routines (e.g. in kernel only boot gives a basic interface)
  - errno.c – global definition of errno (basically a lookup table)
  - inflate.c – “gunzip” part of gzip.c used during boot.
  - string.c – portable string code.
  - vsprintf.c – libc replacement.

Other directories

- linux/ipc: IPC operations and semaphores
- linux/mm: Memory management, paging
- linux/scripts: Scripts to help configure the kernel (e.g. menu for making the kernel)

linux/scripts

- Scripts for:
  - Menu-based kernel configuration.
  - Kernel patching.
  - Generating kernel documentation.

BusyBox Libraries

BusyBox

- Library with very small footprint that implements commonly used functions
  - A single executable that implements the functionality of a massive number of standard Linux utilities, like a monolithic kernel of library functions
  - Provides:
    - ls, gzip, ln, vi. The basics everything you often need, but some have limited features (e.g. limited ls output options)
  - Can drop support altogether for these basic operations or install some of the full version if needed
  - Highly configurable (similar to Linux itself), easy to cross-compile.
  - Tycally static compiled ARM version takes around 2MB

Using BusyBox

Two ways to use the commands:

- Invoke busybox from command line:

  e.g.: busybox ls /tmp

- Or create a aliases that appends busybox to the usual commands e.g.

  alias ll='busybox ls -al'

BusyBox Summary

- BusyBox is a powerful tool for embedded systems, replaces many common Linux utilities with a multicall binary.
- BusyBox significantly reduces the size of your file system image.
- Configuring BusyBox is straightforward, using a makemenu type interface similar to that used for Linux configuration.
- Calling shell commands from user programs will likely need changes to invoke BusyBox unless the shell started from the application has alias set

About Prac 2

The objective of Prac 2 is:
Cross-compile and build Raspbian Kernel successfully using the Raspbian source code.

Local Building or Cross-compiling Linux

- There are two methods to build the Linux kernel:
  - local building: building the kernel on the machine that will boot it
  - Cross-compiling: building the kernel on a different machine from the one that will boot the image
- Local build on the RPi may take too long (the processor isn't as powerful as e.g. an i5) so we will do the cross-compiling approach

Cross-compiling Linux

- The Prac is set up for running the **cross-compilers on Linux**, but there are Linux emulators, such as the Windows Bash Shell
  - Windows Bash Shell essentially provides an Ubuntu system call interface, which allows Linux programs to run, even though behind the scenes there is no Linux kernel, just wrappers to connect from Linux system calls to Windows system calls. This is technically not a virtual machine.

Compiling Raspbian

- The prac descriptions explains the recipe to follows
  - Getting compilers installed on the PC
  - Getting the Raspbian sources
  - Configuring the settings to compile for ARM

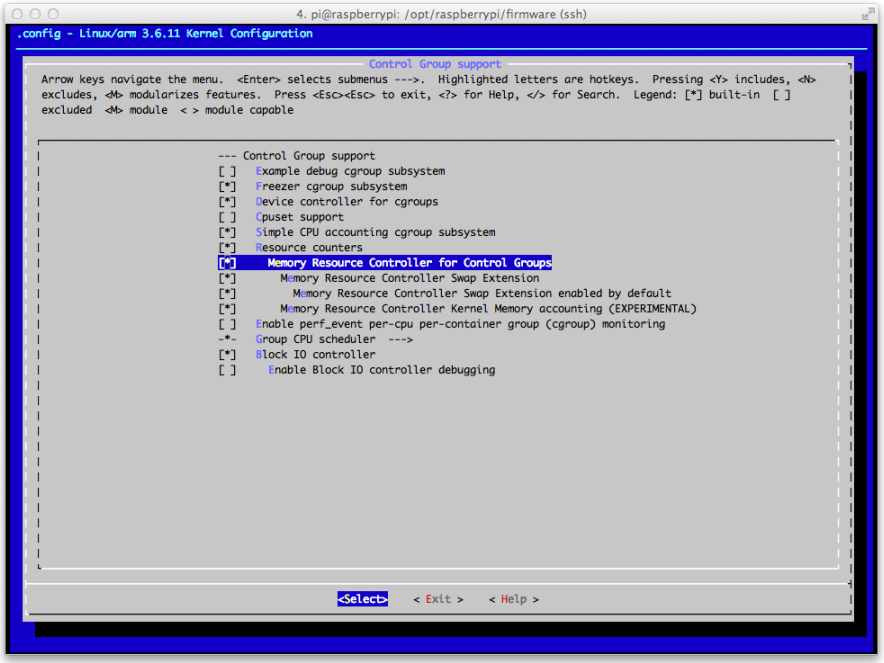
Taking Prac2 further

- Prac 2 just gives a basic start to kernel building, due to the pracs time limits
- If you can find time I strong recommend doing some of the following experiments…

Taking Prac2 further

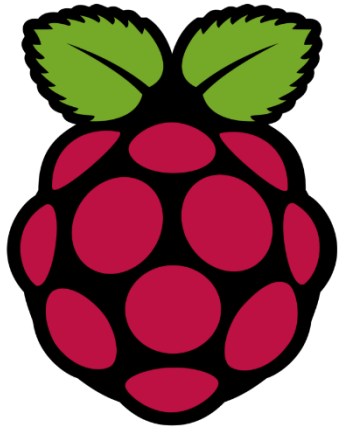
Experiment with **menuconfig**

Type **make manuconfig** from the sources directory. This allows you to configure the kernel, e.g. add/remove drivers and services. Try removing some drivers and see if it still builds, and what size the kernel is.



Minimal Kernel

See if you can make a minimal kernel, to see what how small you can make the image while still using SD card to boot and network to interact



You’re now all set for Prac2 !!

The Next Episode…

Mini Test!

Please see announcement for the test syllabus

References

- Lecture material adapted from the following sources:
  - Greg Kroah-Hartman, “Linux Kernel in a Nutshell”, O’Reilly, 2007
  - Daniel P. Bovet, Marco Cesati, “Understanding the Linux Kernel: From I/O Ports to Process Management”, O’Reilly, 2006
  - R. West Operating System notes for CS 591
  - The Linux Kernel Archives FAQ