ECEN 5803

Mastering Embedded Systems Architecture

Be Boulder.

University of Colorado Boulder

Embedded Linux – What is it?

- It's **not desktop Linux** – It's configured to be smaller and for an embedded target.
- Consists of **4 elements**: Tool Chain, Boot Loader, Kernel, User Applications or File System
- Technically speaking, Linux refers only to an operating system kernel originally written by Linus Torvalds, starting in 1991. Embedded Linux applies this Kernel to an OS for an embedded system.
- Embedded Linux typically refers to a complete system, or in the context of an embedded Linux vendor, to a distribution targeted at embedded devices.

Embedded Linux – Why use it?

"I just had a great idea! Our next product will run Linux"

Embedded Linux – Why use it?

- **Free source code**
- the source code for today's Linux kernel is the result of **contributions from thousands** of **private and corporate** developers.
- The Linux kernel is **written in the C** programming language, and it supports all major computer architectures.
- It includes **device drivers for thousands of devices**, and it is easily extended to support new devices and protocols.
- Linux has an excellent reputation for **security and reliability**, and sees widespread use in military applications, such as drone fighters and other autonomous vehicles
- the kernel is usually built with support for physical address extension (PAE). PAE supports **physical memory sizes up to 64GB**.
- The main advantage to using a VM OS like Linux in an embedded system is that the burden of **testing may be reduced** considerably.

Embedded Linux – Why use it?

- The **source code is free**
- Linux, as opposed to commercial OSes, is **royalty-free**. Even if you buy your Linux from a commercial vendor, you typically pay by the number of developer seats, not the volume of products shipped.
- #1 - **Linux Rules the Net**. Networking was once an add-on you paid extra for. With Linux, it is built right in, fully configurable, as simple or cutting-edge as you wish.
- #2 - Linux has **multiple display options**, from SVGALib, to Simple Direct Layer (SDL) to Xorg.
- #3 - Linux **plays multi-media** well. The kernel is highly configurable for multimedia quality-of-service. There are several codec frameworks to pick from, like gstreamer, helix, and mencoder. There are many media player options, like mplayer, xine, totem, alsaplayer, aplay, mpc; There are various audio routing layers (esd, PulseAudio and Jack).

Embedded Linux – Why use it?

- The source code is free
- Source code with high quality and reliability.
- Source code with high availability.
- **Broad hardware support**, both for processors and peripherals
- **Many communication protocols**
- Many good support tools, see Freshmeat (*http://www.freshmeat.net*) and SourceForge (*http://www.sourceforge.net*)
- Great community support
- **Vendor independence** – distributions are available from many vendors instead of just one.

Embedded Linux – Why NOT use it?

- The **GPL License can force you to share** your source code with competitors.
- Linux is **not real-time** without extensive modifications that lead to driver incompatibility.
- Linux is **a moving target** – with releases every 6 months, ongoing support of a product becomes an issue, especially driver changes. product companies want one thing – stability and long product life-cycles -- and open source developers (including Linux developers) want rapid progress. When these goals collide, embedded product companies may feel like they've hitched their wagon to a rocket-ship.
- Linux is **not free** – tools cost, support costs, software packaged with it costs, etc. Development can be time consuming and expensive.
- To be useful, Linux needs to be **large** – 8 to 16 MB, and requires a 32-bit processor. The total available source code has become very large and complex and hard to understand, leading to the need for consultants and their high fees.

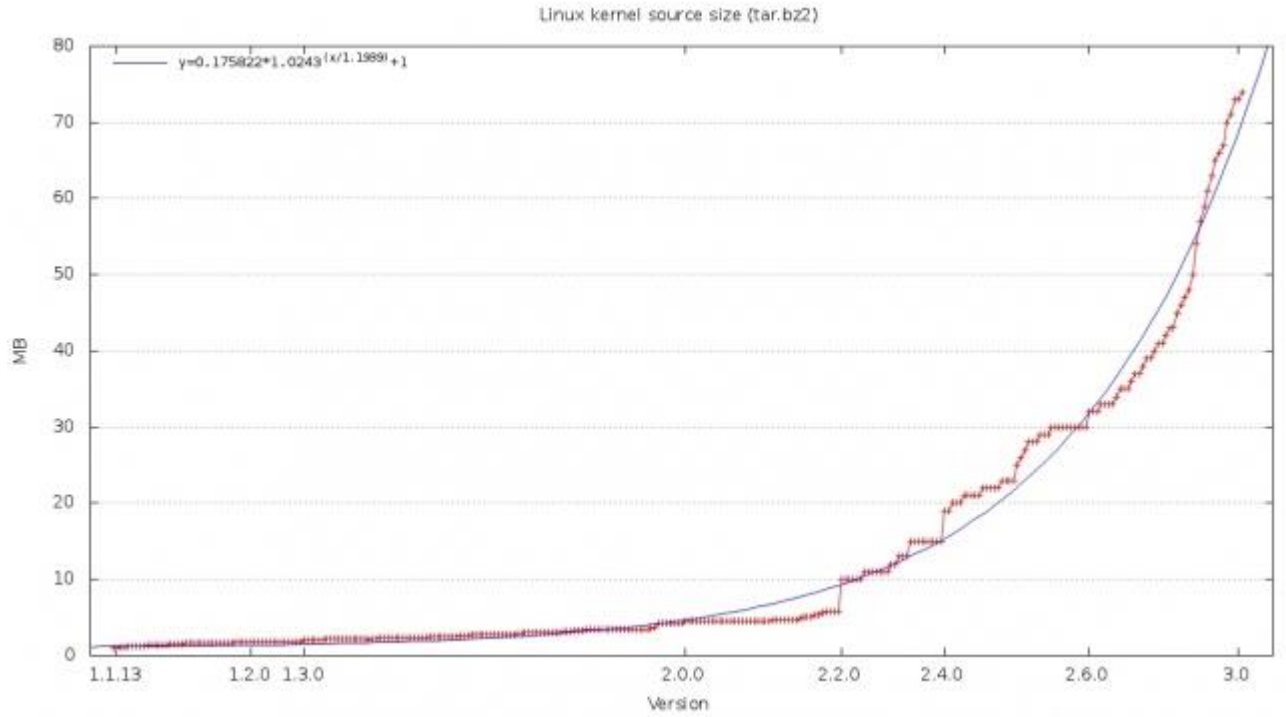
Embedded Linux - Size

- David A. Wheeler studied the Red Hat distribution of the Linux operating system, and reported that Red Hat Linux version 7.1[5] (released April 2001) contained over **30 million physical SLOC**. He also extrapolated that, had it been developed by conventional proprietary means, it would have required about 8,000 person-years of development effort and would have cost over $1 billion (in year 2000 U.S. dollars). SLOC = source lines of code.
- A similar study was later made of Debian GNU/Linux version 2.2 (also known as "Potato"); this operating system was originally released in August 2000. This study found that Debian GNU/Linux **2.2 included over 55 million SLOC**, and if developed in a conventional proprietary way would have required 14,005 person-years and cost $1.9 billion USD to develop. Later runs of the tools used report that the following release of Debian had 104 million SLOC, and as of year 2005, the newest release is going to include over 213 million SLOC.

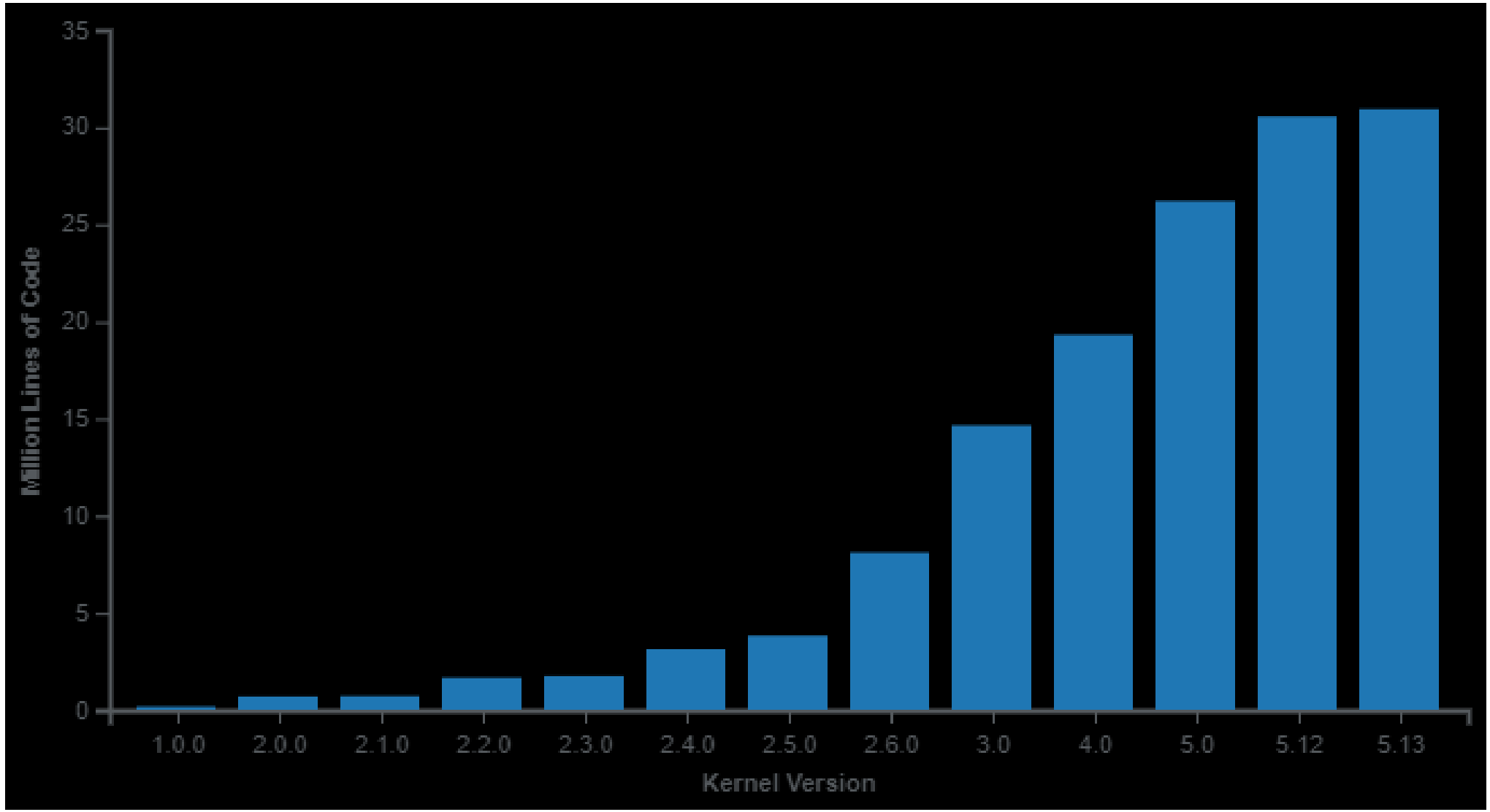
Embedded Linux - Size

Year	Operating System	SLOC (Million)
2000	Debian 2.2	55–59
2002	Debian 3.0	104
2005	Debian 3.1	215
2007	Debian 4.0	283
2009	Debian 5.0	324
2012	Debian 7.0	419
2009	OpenSolaris	9.7
	FreeBSD	8.8
2005	Mac OS X 10.4	86
2001	Linux kernel 2.4.2	2.4
2003	Linux kernel 2.6.0	5.2
2009	Linux kernel 2.6.29	11.0
2009	Linux kernel 2.6.32	12.6
2010	Linux kernel 2.6.35	13.5
2012	Linux kernel 3.6	15.9
2015-06-30	Linux kernel pre-4.2	20.2

2018 – Kernel 4.13.9 20,323,379
2018 - 25,584,633 Kernel 4.15
2021 – Kernel 5.11 30.34 million



Embedded Linux - Size



Learning Embedded Linux

1. Become familiar with desktop Linux. Rather than create a Linux for the target, it's easier to use a VM on your development machine to become familiar with Linux. Install VMware or Virtual Box and install a popular Linux like Ubuntu or CentOS inside the VM.
2. Learn shell commands to perform actions from the command line.
3. Practice building and debugging programs with GCC and GDB.

Good learning resources: https://www.embedded.com/electronics-blogs/open-mike/4420567/Learning-Linux-for-embedded-systems

https://elinux.org/Main_Page

Embedded Linux Element – Toolchain

A tool chain consists of at least

- binary utilities: GNU assembler, linker, etc.
- Compiler: gcc, GNU C compiler
- C library (libc): the interface to the operating system
- Debugger: gdb

A tool chain is Native if you run the compiler on a target board. If your target board is not fast enough or doesn't have enough memory or storage, use an emulator e.g. qemu

Cross Compiler: compile on one machine, run on another; this is the most common option.

Embedded Linux Element – Toolchain

C Library options are

Main options are

- GNU glibc – big but fully functional
- GNU eglibc – glibc but more configurable; embedded-friendly
- uClibc – small, lacking up-to-date threads library and other POSIX functions

- dietlibc — a small standard C library used to produce static executable that do not use shared libraries.
- Newlib — a small standard C library that is used in many free, commercial, and vendor-distributed toolchains for embedded development, and is also used by the popular Cygwin project, which is a set of tools and API layer for Linux applications in the Microsoft Windows environment.

Embedded Linux Element – Toolchain

Criteria for selecting a toolchain

- Good support for your processor e.g. for ARM A-8 core, armv4 compilers work OK but armv7 works better
- Appropriate C library
- Up-to-date
- Good support (community or commercial)
- Other goodies, libraries and development tools for tracing, profiling, etc.

Embedded Linux Element – Toolchain Examples

Free, minimal

Source	URL	Architectures
Codesourcery G++ Lite	www.codesourcery.com	MIPS, PPC, SH
Bootlin	https://toolchains.bootlin.com	ARM, PPC, SH, MIPS, x86, RISCV
Linaro	https://wiki.linaro.org/WorkingGroups/ToolChain	ARM

Free, Binary

Source	URL	Architectures
Angstrom	www.angstrom-distribution.org	ARM, PPC, AVR32, SH
Debian	www.debian.org	ARM, PPC
Ubuntu	www.Ubuntu.org	ARM
Denx ELDK	www.denx.de	PPC, ARM, MIPS

Toolchain examples

Free, integrated build environment

	URL	Architectures
Buildroot	www.buildroot.org	ARM, PPC, MIPS
OpenEmbedded	www.openembedded.org	ARM, PPC, AVR32, SH
LTIB	www.bitshrine.org	ARM, PPC

Commercial

	URL	Architectures
MontaVista Linux	www.mvista.com	
Timesys LinuxLink	linuxlink.timesys.com	
Windriver Linux	www.windriver.com	
LynuxWorks BlueCat Linux	www.lynuxworks.com	
Sysgo ElinOS	www.sysgo.com	

Embedded Linux Element – Bootloader

Functions of the Bootloader:

Initialize the hardware

- Set up SDRAM controller
- Map memory
- Set processor mode and features

Load a kernel

Set the kernel command line (see later)

Jump to kernel start vector, passing pointers to

– information about hardware

– kernel command line

Optional (but very useful)

- Load images via Ethernet, serial, SD card
- Erase and program flash memory
- Display splash screen

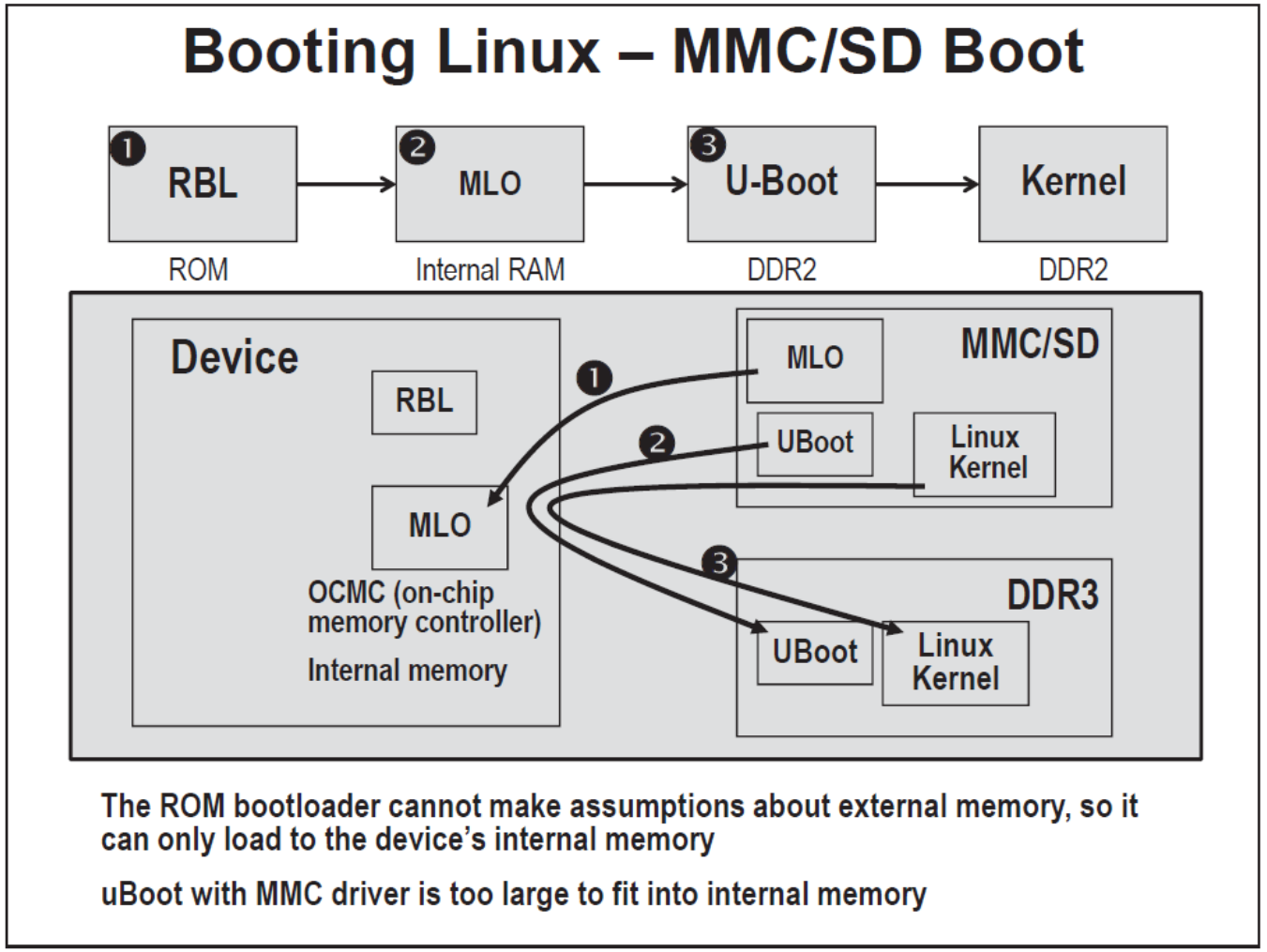
Considerable debugging of hardware can be done in the bootloader!

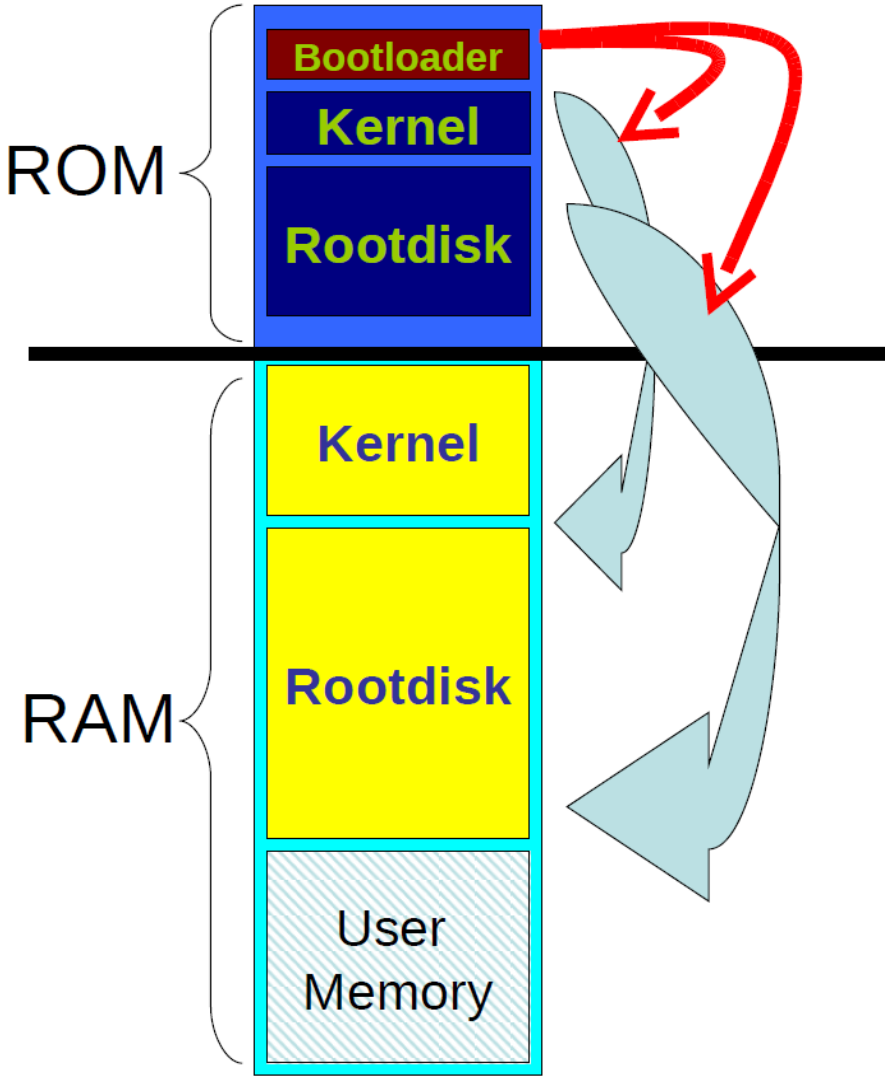
Embedded Linux Element – Bootloader

Examples of boot loaders

- (Das) U-Boot
  - PPC, ARM, MIPS, SH4
  - http://www.denx.de/wiki/U-Boot/WebHome
- Redboot
  - PPC, ARM, MIPS, SH4
  - http://sources.redhat.com/redboot/
- Barebox
  - PPC, ARM, MIPS, x86, Blackfin, NIOS2
  - https://www.barebox.org/
- For PC hardware use
  - BIOS together with GRUB or LILO

Embedded Linux Element – Bootloader





Jim Huang and Matt Hsu. (2015 Nov. 11) Qi Bootloader. [Online] Available: http://www.slideshare.net/jserv/qi-lightweight-boot-loader-applied-in-mobile-and-embedded-devices

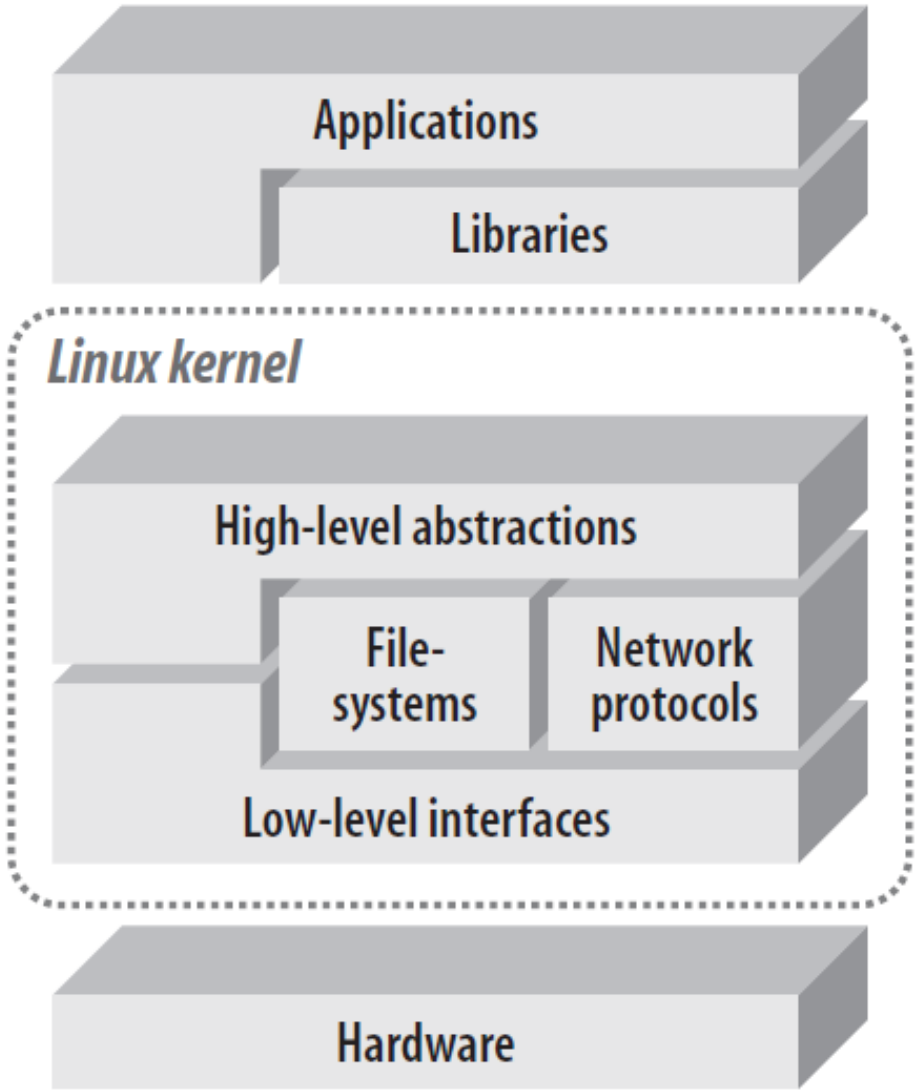
Embedded Linux Element – Kernel

The Linux kernel is the heart of any Linux installation, embedded or otherwise. The kernel is responsible for:

- memory allocation,
- process and thread creation, management and scheduling
- data exchange with onboard hardware and peripheral devices
- supports the protocols necessary for interaction with other systems, as needed.

TimeSys. (2015 Nov. 11). *Embedded Linux Development Tutorial*. [Online] Available: https://linuxlink.timesys.com /docs/wiki/embedded_linux_development_tutorial

Embedded Linux Element – Kernel as part of the Arch.



Karim Yaghmour, Jon Masters, Gilad Ben-Yossef, and Philippe Gerum, Building Embedded Linux Systems, 2nd Ed. Cambridge, MA: O'REILLY, 2008, pp 33-40.

Embedded Linux Element – Kernel

The Linux kernel can use device drivers in one of two ways. They can either be compiled into the kernel or can be dynamically loaded on demand when delivered as loadable kernel modules (LKMs).

Loadable kernel modules are compiled object files that are compiled to work with a specific kernel and that can be dynamically loaded into a running instance of that kernel in response to detecting new hardware or protocol requirements. LKMs are stored outside the kernel as files in a physical or in-memory filesystem that the kernel has access to, and have the ".ko" (kernel object) extension to make it easy to identify them.

TimeSys. (2015 Nov. 11). *Embedded Linux Development Tutorial*. [Online] Available: https://linuxlink.timesys.com/docs/wiki/embedded_linux_development_tutorial

Embedded Linux Element – Kernel Scheduling

Scheduling Methodologies

Time-Slicing Threads

Scheduler shares processor run time between all threads with greater time for higher priority

- ✓ No threads completely starve
- ✓ Corrects for non-"good citizen" threads
- × Can't guarantee processor cycles even to highest priority threads.
- × More context switching overhead

Realtime Threads

Lower priority threads won't run unless higher priority threads block (i.e. pause)

- × Requires "good citizen" threads
- × Low priority threads may starve
- ✓ Lower priority threads never break high priority threads
- ✓ Lower context-switch overhead

- Time-sliced threads have a "nieceness" value by which administrator may modify relative loading
- Linux dynamically modifies processes' time slice according to process behavior
- With realtime threads, the highest priority thread always runs until it blocks itself

Texas Instruments. (2015 Nov. 12). *Introduction to Embedded Linux*. [Online]. Available: https://training.ti.com/system/files/docs/bbb_1day_student_guide_0.pdf

Embedded Linux Element – Kernel Scheduling

Scheduling Policy Options

	SCHED_OTHER	SCHED_RR	SCHED_FIFO
Sched Method	Time Slicing	Real-Time (RT)	
RT priority	0	1 to 99	1 to 99
Min niceness	+20	n/a	n/a
Max niceness	-20	n/a	n/a
Scope	root or user	root	root

- Time Sliced scheduling is specified with **SCHED_OTHER**:
  - Niceness determines how much time slice a thread receives, where higher niceness value means less time slice
  - Threads that block frequently are rewarded by Linux with lower niceness
- Real-time threads use preemptive (i.e. priority-based) scheduling
  - Higher priority threads always preempt lower priority threads
  - RT threads scheduled at the same priority are defined by their policy:
    - **SCHED_FIFO**: When it begins running, it will continue until it blocks
    - **SCHED_RR**: "Round-Robin" will share with other threads at it's priority based on a deterministic time quantum

Texas Instruments. (2015 Nov. 12). *Introduction to Embedded Linux*. [Online]. Available: https://training.ti.com/system/files/docs/bbb_1day_student_guide_0.pdf

Embedded Linux Element – Kernel Scheduling

The Usefulness of Processes

Option 1: Audio and Video in a single Process

```
// audio_video.c
// handles audio and video in
//     a single application

int main(int argc, char *argv[])
{
      while(condition == TRUE){
            callAudioFxn();
            callVideoFxn();
      }
}
```

Option 2: Audio and Video in separate Processes

```
// audio.c, handles audio only

int main(int argc, char *argv[]) {
      while(condition == TRUE)
            callAudioFxn();
}
```

```
// video.c, handles video only

int main(int argc, char *argv[]) {
      while(condition == TRUE)
            callVideoFxn();
}
```

Splitting into two processes is helpful if:

1. audio and video occur at different rates
2. audio and video should be prioritized differently
3. multiple channels of audio or video might be required (modularity)
4. memory protection between audio and video is desired

Texas Instruments. (2015 Nov. 12). *Introduction to Embedded Linux*. [Online]. Available: https://training.ti.com/system/files/docs/bbb_1day_student_guide_0.pdf

Embedded Linux Element – Kernel Scheduling

Terminal Commands for Processes

ps Lists currently running user processes

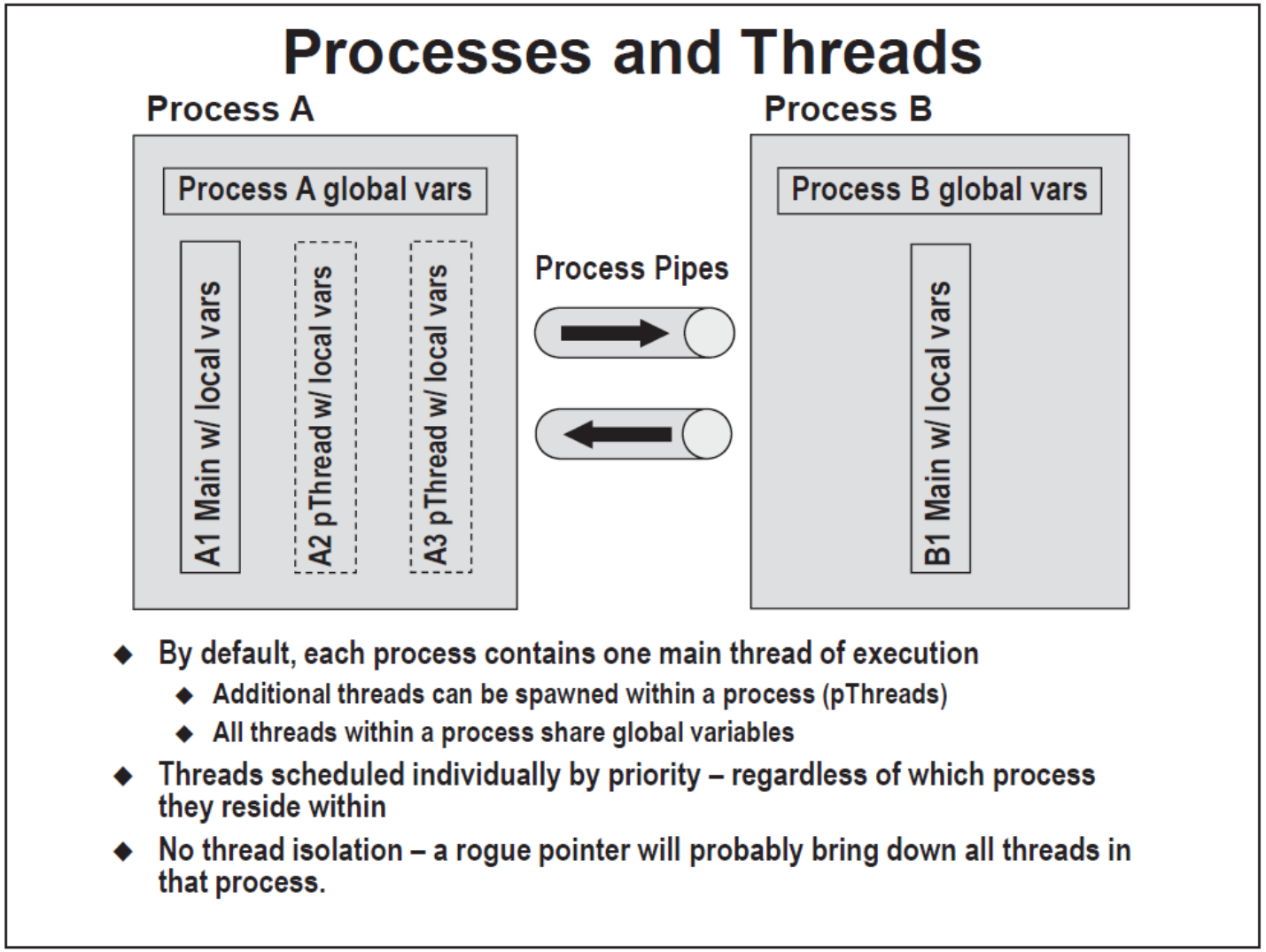
ps -e Lists all processes

top Ranks processes in order of CPU usage

kill <*pid*> Ends a running process

renice +5 –p <*pid*> Changes time-slice ranking of a process (range+/- 20)

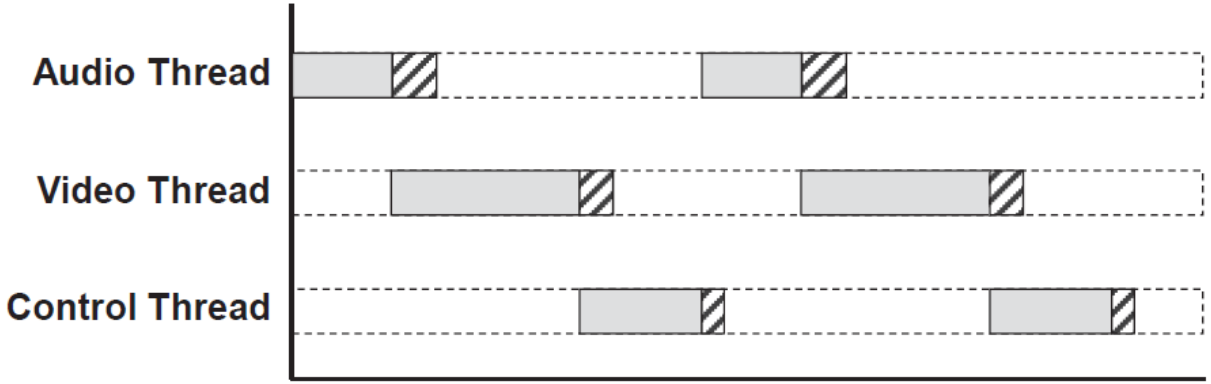
Embedded Linux – Threads



Texas Instruments. (2015 Nov. 12). *Introduction to Embedded Linux*. [Online]. Available: https://training.ti.com/system/files/docs/bbb_1day_student_guide_0.pdf

Embedded Linux – Threads

Time-Sliced A/V Application, >100% load

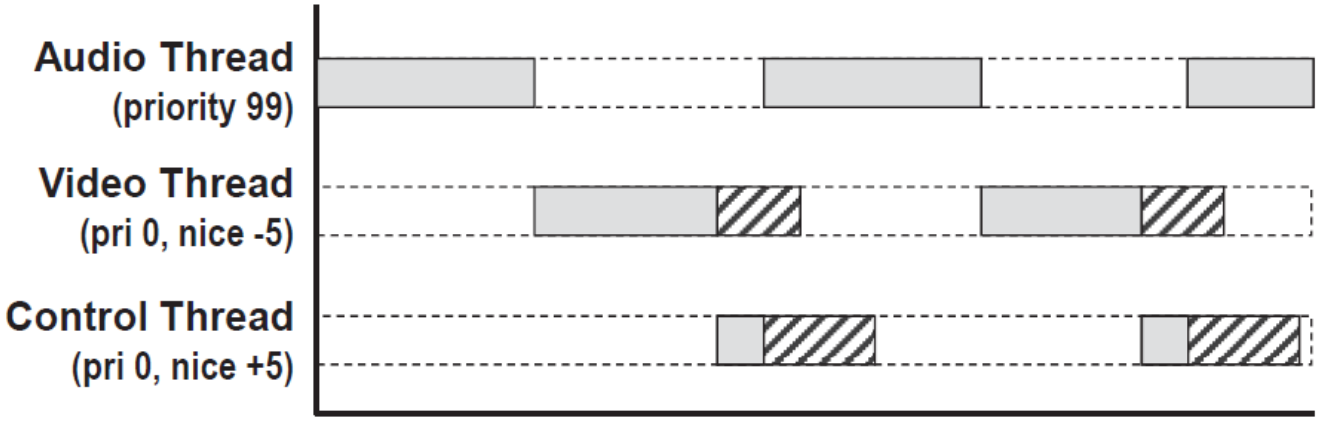


- Adding a new thread of the highest "niceness" (smallest time slice) may disrupt lower "niceness" threads (higher time slices)
- All threads share the pain of overloading, no thread has time to complete all of its processing
- Niceness values may be reconfigured, but system unpredictability will often cause future problems
- In general, what happens when your system reaches 100% loading? Will it degrade in a well planned way? What can you do about it?

Texas Instruments. (2015 Nov. 12). *Introduction to Embedded Linux*. [Online]. Available: https://training.ti.com/system/files/docs/bbb_1day_student_guide_0.pdf

Embedded Linux – Threads

Hybrid A/V Application, >100% load



- Audio thread is guaranteed the bandwidth it needs
- Video thread takes *most* of remaining bandwidth
- Control thread gets a small portion of remaining bandwidth

Texas Instruments. (2015 Nov. 12). *Introduction to Embedded Linux*. [Online]. Available: https://training.ti.com/system/files/docs/bbb_1day_student_guide_0.pdf