Embedded Linux

CPE-555 Real-Time and Embedded Systems

Stevens Institute of Technology

Spring 2017

Richard Prego

Overview

- Introduction to Linux
- Why use Linux for embedded?
- The Linux kernel
- Kernel modules
- Device drivers
- POSIX threads
- Real-time Linux

First, a disclaimer…

- Linux is part of a large and complex set of system components used in many embedded systems
  - Includes filesystems, networking, bootloaders, build systems, drivers, etc.
- These topics could easily fill an entire semester (or more) in detail
  - We will only go over some of the basics to provide an introduction in this course

Introduction to Linux

- In 1969, Dennis Ritchie and Ken Thomson, along with other engineers at AT&T Bell Labs, began designing an early operating system that ultimately evolved into Unix
- Throughout the 1970s and 1980s, AT&T and other companies continued to develop Unix

Introduction to Linux

- Unix has an elegant, yet still powerful design:
  - Rather than implementing thousands of system calls, Unix implemented only a few hundred
  - Almost everything is a file: this makes manipulating data and devices easy, using a single simple interface
  - Unix has simple interprocess communication and low overhead in creating new processes
  - Unix is written in C - making it easy to port between processor architectures

Introduction to Linux

- Linux Torvalds created Linux in 1991 as a free operating system for computers using the Intel 80386 processor
- Linus was a student at the time, and was disappointed at the lack of a powerful free version of Unix
- Linus posted his code on the Internet and opened the door for others to contribute – many developers started to add to the project

Introduction to Linux

- Today, Linux runs on many different processor architectures (x86, ARM, PowerPC, and more)
- Linux powers all sorts of devices:
  - Watches
  - Phones (Android)
  - Washing machines
  - TVs
  - Desktops
  - Servers
  - Datacenters

Introduction to Linux

- Linux is a Unix-like system, but is not actually a descendent of Unix
  - Linux borrows many ideas from and follows the main design goals of Unix
- Linux itself is just a kernel, not a full operating system
  - Full systems often include a desktop environment, C library, a shell, a window system, and more in addition to the Linux kernel

Why use Linux for embedded?

- Linux is open-source: anyone can add to or change the source code
- Component re-use: many standard features already exist (filesystems, networking stacks, graphic libraries, USB library, etc.) – no need to “reinvent the wheel”
- Low cost: many distributions of Linux are provided at no cost, including the development tools

Why use Linux for embedded?

- Full control: with open-source components, developers can modify the software as needed
- Quality: the Linux kernel and open-source libraries are used in millions of systems, so they have been well tested
- Community support: there are many knowledgeable Linux users and developers in the community

Why use Linux for embedded?

- Supports many architectures: no need to port an operating system to a new processor architecture
- Supports many common communications buses: I2C, SPI, CAN, USB, etc.
- Extensive networking support: Ethernet, WiFi, Bluetooth, IPv4, IPv6, TCP, UDP, FTP, etc.

The Linux kernel

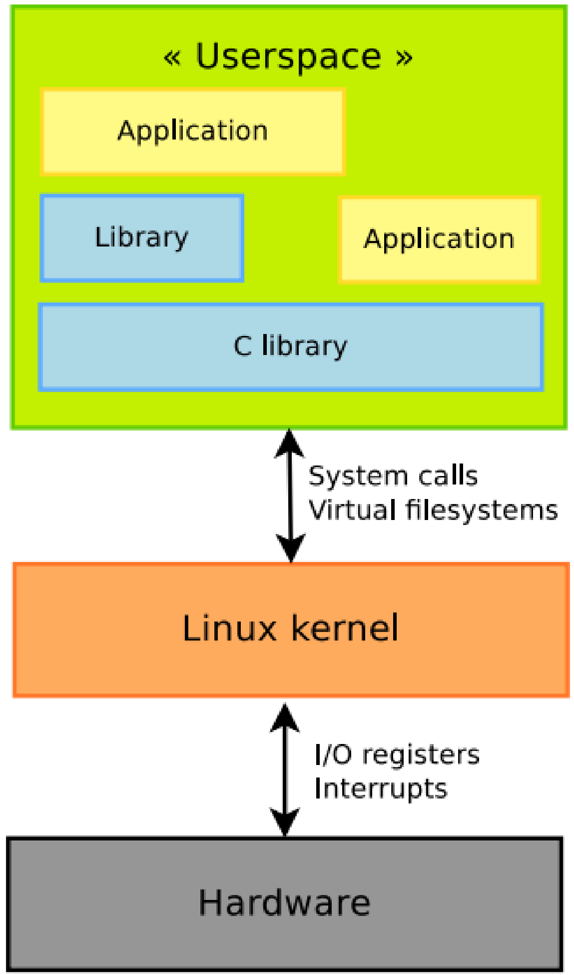
- The kernel is the core part of the operating system
- The kernel provides these major features:
  - Process management
  - Memory management
  - Inter-process communication
  - Timers
  - Device drivers for hardware
  - Filesystems
  - Power management
  - Etc.

STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

The Linux kernel

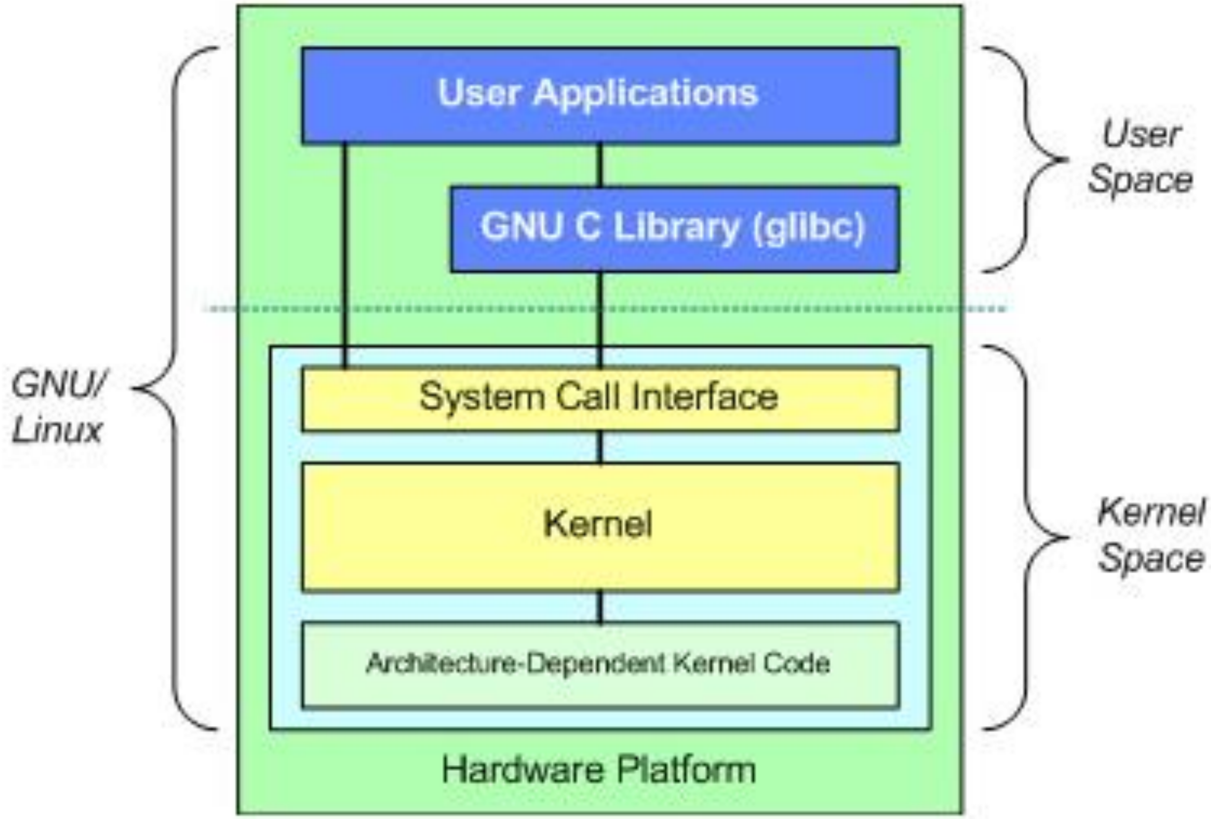
- The kernel runs in privileged mode
  - Kernel can access and control the hardware
  - Kernel controls access to shared resources (such as memory, processor time, communications busses, etc.)
- User applications run in userspace, or unprivileged mode
  - Applications must go through the kernel to access resources, including the hardware

The Linux kernel



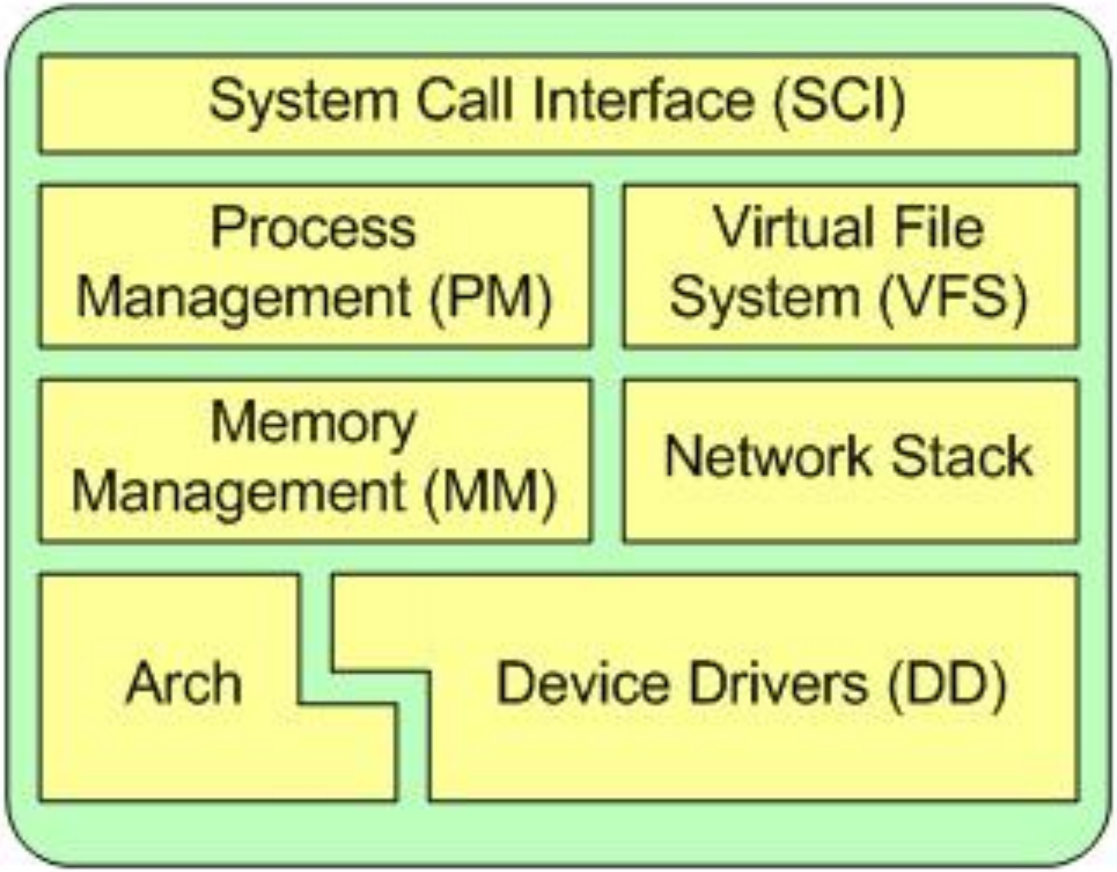
Source: Petazzoni

The Linux kernel



Source: Jones

The Linux kernel



Source: Jones

Kernel modules

- The Linux kernel is a monolithic kernel
  - All services in the kernel execute in a single address space (large process) in kernel mode
  - In contrast to a microkernel, which is broken down into several processes, some of which run in kernel mode and others which run in user mode

Kernel modules

- Kernel modules allow us to add and remove software components to the kernel on-the-fly while the system is running
  – Modules are loaded or unloaded while the kernel is running
- Device drivers are most often implemented as kernel modules

Kernel modules

- Advantages of kernel modules:
  - Makes it easy to develop drivers without having to reboot the system
  - Keeps kernel size to a minimum: only include what is needed in the kernel, make everything else a loadable module
  - Reduces boot time: devices and kernel features that aren't needed immediately at boot can be initialized later
- Caution: once a kernel module is loaded, it has full access to the system in kernel mode, since it is becomes part of the running kernel

Device drivers

- Device drivers allow userspace programs to access and use the hardware
- Drivers are commonly implemented as kernel modules
  – Access to hardware goes through the kernel
- There are three main classes of device drivers in Linux:
  – Character devices
  – Block devices
  – Network devices

Device drivers

- Character devices
  - Devices that can be accessed as a stream of bytes, similar to a file
  - A character driver is used to access this type of device
  - Driver usually implements open(), read(), write(), close(), and ioctl() system calls
  - Character devices are accessed as filesystem nodes: /dev/tty1, /dev/lp0
  - Usually cannot move backwards in a character device, since it is usually a type of data channel

Device drivers

- Block devices
  - Devices that can host a filesystem, such as a disk or memory device
  - A block driver is used to access this type of device
  - Block devices are also accessed as filesystem nodes: /dev/sda1, /dev/sr0
  - Difference between character devices and block devices is how data is managed in the kernel and the kernel/driver interface - the difference is transparent to the user

Device drivers

- Network devices
  - Network transactions are made through an interface
  - Interfaces are usually physical hardware devices, but can also be a pure software device, like a loopback interface or virtual connection
  - Network interface is in charge of sending and receiving data packets: interface has no concept of a connection since it deals only in packets
  - Kernel calls used for accessing network devices are entirely different than those used for character and block devices

POSIX threads

- Portable Operating Systems Interface (POSIX) is a set of standards defined by IEEE to establish compatibility between operating systems
  - Defines Application Programming Interface (API), shells, and utilities for compatibility with Unix, Linux, and other operating systems
- IEEE POSIX 1003.1c standard defines a standardized C language thread programming interface we call "Pthreads"

POSIX threads

- We have already discussed threads v. processes
  - Threads share resources (memory, open files, etc.)
  - Processes each have their own address space
- Pthreads allows us to create threads within a program
  - fork() system call is used to create a new process as a clone of a currently running process (we won’t go into the details here)

POSIX threads

- To use Pthreads, include the header pthread.h:

```
#include <pthread.h>
```

```
int pthread_create(pthread_t *restrict thread,
              const pthread_attr_t *restrict attr,
              void *(*start_routine)(void*),
              void *restrict arg);
void pthread_exit(void *value_ptr);
int pthread_cancel(pthread_t thread);
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

POSIX threads: creation/termination

- When your program starts (and executes main()), the program is running a single thread
- `pthread_create(thread, attr, start_routine, arg)` creates a new thread
  - `thread`: a unique identifier, returned by the subroutine
  - `attr`: an object used to specify thread attributes
  - `start_routine`: the C function the thread will execute when it is created
  - `arg`: pointer to an argument to pass to the thread

POSIX threads: creation/termination

- A thread can be terminated in different ways:
  - The thread finishes and returns from its function
  - The thread calls `pthread_exit()`
  - The thread is canceled by another thread with `pthread_cancel()`
  - main() finishes before the threads, without calling `pthread_exit()`

STEVENS INSTITUTE of TECHNOLOGY THE INNOVATION UNIVERSITY®

POSIX threads: attributes

- When a thread is created, an attribute object can be passed to `pthread_create()`
- Attributes that can be set include:
  - Whether thread is joinable
  - Scheduling inheritance
  - Scheduling policy
  - Stack size
  - Stack address
  - Etc.

POSIX threads: creating/termination

- Example 1: example_1_creating_threads.c

POSIX threads: joining/detaching

```
int pthread_join(pthread_t
thread, void **value_ptr);
```

```
int pthread_detach(pthread_t
thread);
```

- pthread_join() will block until the specified thread returns (completes)

POSIX threads: joining/detaching

- Example 2: example_2_joining_threads.c

POSIX threads: mutexes

```
int pthread_mutex_init(pthread_mutex_t *restrict
   mutex, const pthread_mutexattr_t *restrict attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

POSIX threads: mutexes

- Mutexes are initially unlocked when they are created
- `pthread_mutex_lock()` will block if the mutex is locked by another thread
- `pthread_mutex_trylock()` will return with an error if the mutex is locked by another thread
  - Useful to prevent deadlock situations

POSIX threads: mutexes

- Example 3: example_3_mutexes.c

POSIX threads: condition variables

- Condition variables are another synchronization mechanism
  - Mutexes control access to a resource or data
  - Condition variables allow threads to signal different conditions to each other
- Condition variables are always used with a mutex

POSIX threads: condition variables

```
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t *restrict cond,
      const pthread_condattr_t *restrict attr);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
      pthread_mutex_t *restrict mutex,
      const struct timespec *restrict abstime);
int pthread_cond_wait(pthread_cond_t *restrict cond,
      pthread_mutex_t *restrict mutex);
```

POSIX threads: condition variables

- Example 4: example_4_condition_vars.c

Real-time Linux

- Unix and Linux were both designed for fairness in their schedulers
  - Goal is to allocate resources across all processes that require the CPU and guarantee processes can all make progress
- Unfortunately, this design does not fit real-time applications

Real-time Linux

- In early days of Linux (1.x), when a userspace process made a system call to use kernel services, no other task was able to run until the process blocked or completed its system call
- Kernel preemption was added to allow the kernel to be preempted when a higher-priority task becomes ready

Real-time Linux

- In a kernel with preemption enabled (and on multi-processor systems), shared kernel data structures must be protected
- The approach used in Linux is a spin lock
  - Spin locks poll waiting for the lock to be free
  - Thread does not block waiting for the lock
- Since there are many kernel services that use many kernel data structures "simultaneously" the latency in waiting for a spin lock is difficult to predict

Real-time Linux

- The PREEMPT_RT kernel patch tries to address the shortcomings of using spin locks in the kernel
  - Goal is to replace spin locks with semaphores so that the threads are able to be blocked/sleep
  - Patch also moves interrupt processing from ISRs to high-priority threads, so that the interrupt handler work may also be blocked when a semaphore is needed

Real-time Linux

- Even with the PREEMPT_RT patch, latencies in the kernel cannot be guaranteed
  - However, worst-case latencies are more tightly bounded
- PREEMPT_RT improves performance for soft real-time services, but still is not appropriate for hard real-time systems

Summary

- Linux was developed as a free alternative to Unix
- Wide support and open community provide many reasons to use Linux for embedded systems
- Kernel modules allow us to add features to the kernel on-the-fly
- Three types of device drivers: character devices, block devices, network devices
- POSIX threads (Pthreads) are used for multi-threading in Linux (and other POSIX-compliant systems)
- PREEMPT_RT kernel patch improves kernel performance for soft real-time

References and additional reading

- R. Love, Linux Kernel Development, Third Edition. Upper Saddle River, NJ: Pearson Education, 2010.
- C. Hallinan, Embedded Linux Primer, Second Edition. Upper Saddle River, NJ: Pearson Education, 2011.
- S. Siewert and J. Pratt, Real-Time Embedded Components and Systems With Linux and RTOS. Dulles, VA: Mercury Learning, 2016.
- T. Petazzoni, "Embedded Linux Introduction". Free Electrons, 2011. Available online: http://free-electrons.com/pub/conferences/2011/limoges-clermont/presentation.pdf
- "Embedded Linux system development". Free Electrons, 2016. Available online: http://free-electrons.com/doc/training/embedded-linux/embedded-linux-slides.pdf
- M. Jones, "Anatomy of the Linux kernel". IBM, 2007. Available online: http://www.ibm.com/developerworks/linux/library/l-linux-kernel/
- J. Corbet, A. Rubini, G. Kroah-Hartman, Linux Device Drivers, Third Edition. Sebastopol, CA: O'Reilly, 2005. Available online: https://lwn.net/Kernel/LDD3/
- B. Barney, "POSIX Threads Programming". Lawrence Livermore National Laboratory, 2016. Available online: https://computing.llnl.gov/tutorials/pthreads/

STEVENS INSTITUTE of TECHNOLOGY THE INNOVATION UNIVERSITY®