Introduction to Kernel Programming

Luca Abeni
luca.abeni@santannapisa.it

About the Course

- Goal: understand how to code an OS kernel
  - This course will introduce the students to the pains and joys of kernel programming...
  - Of course, this is only an introduction!
- First question: so, why is kernel programming so different?
  - But, first of all, what is a kernel?
  - ...And what is an Operating System?
  - ...And, what is a computer? Where are we coming from? Where are we going?
    - No, I will not answer the last questions...

Practical Details

- The course is 20 hours long (2 credits)
- Organized in lessons by 2:30 hours each
- Some theory and some practice
  - There cannot be practice without theory...
  - ...But theory is useless without practice!
- For informations, email
  `luca.abeni@santannapisa.it`
  - Office: TeCIP (via Moruzzi 1)

Overview

- Quick recap about architecture, OSs, kernels
  - Privileged instructions $\rightarrow$ kernel
- The kernel execution environment
- Kernel development: DIY, or existing systems (Linux)?
- Introduction to the Linux kernel
  - Kernel modules, concurrency, synchronization, ...
- Some examples

The Operating System

- Operating System: set of computer program acting as an interface between applications and hardware
  - “Set of computer programs”: the OS is not a single program!
  - “Acting as an interface... ”: applications do not directly access the hardware (must use the OS)
- So, the OS:
  - Hides the hardware details to user applications
  - Controls the execution of user programs
  - Manages the hardware and software resources
- Applications running on an OS can use the CPU Assembly extended with some additional instructions: the *system calls*

The OS Kernel

- Modern CPUs: different privilege levels (user level and privileged level)
  - Actually, it can be even more complex than this $\leftarrow$ hypervisor mode, ...
- Security / protection $\rightarrow$ only a small amount of trusted code should run with a high privilege level
  - OS Kernel: part of the OS executing with a high privilege level
- Regular user applications execute at a lower privilege level
  - To protect the system from malicious programs

The OS Kernel - Again

- Kernel: part of the OS running at high privilege level
  - Can do (almost!) everything: even crash the system
- This is why it must be trusted... Very critical component of the system
  - Security and stability depend on it!
  - But also the system performance depends on it...
- “With Great Power Comes Great Responsibility”
- Applications rely on the kernel to do everything important / critical
  - How is kernel execution invoked? Interrupts / exceptions (hw or sw)

Kernel Responsibilities

- Multiprogramming:
  - Multiple tasks (processes or threads) on few CPUs
  - Memory protection: multiple address spaces (paging, segmentation)
  - CPU privilege levels: system and user
- Low level hardware details:
  - Interrupt handling, boot, device drivers, system calls, ...
  - Important data structures (memory page tables, ...)
- Kernel address space: can see all the system memory

Kernel Functionalities

- System boot $\rightarrow$ configure and set-up the system so that virtual memory and multitasking can work
  - Configure memory, page tables & friends
  - Once it is done, start the first user-space process: in Unix, it is `init` (PID 1)
  - Then, execution returns to the kernel only through interrupts
- Hardware interrupt handlers (ISRs, used by drivers)
- System calls (software interrupts, ...)
  - Interrupts cause a privilege change (user $\rightarrow$ system)
  - Syscalls and ISRs can cause context switches

Task Handling

- The kernel handles processes and threads
- Each task is characterised by:
  - A Task Descriptor (TSS)
  - A Task Body (code implementing the task)
  - Some (public or private) data (Address Space)
- Task Descriptor $\rightarrow$ contains copy of the CPU registers, including:
  - Pointers to user-space and kernel stack
  - Address Space configuration (CR3, ...)
  - Protection level (CPL)
- The task body is technically part of the address space

Task Address Space

- Divided in segments:
  - Code Segment (task body)
  - Data Segment
    - Initialised Data, BSS, Heap
  - Stack Segment
- Recently, some additional segments (RO data, etc...)

- Before starting a task, the OS kernel has to:
  - Initialise memory segments (allocate virtual memory)
  - Allocate and set up a stack, initialise the stack pointer, etc...
  - Allocate and initialise (to 0) the BSS

The Rest of the OS

- The kernel is only part of the OS
- There also are many user-space components
  - System libraries
  - System programs
  - ...
- System libraries $\rightarrow$ needed to properly invoke kernel functionalities (hide the syscall mechanism)
- System programs $\rightarrow$ needed to properly boot and use the system

How does a Kernel Look Like

- No single entry point
  - “Boot entry point” + system calls
- Kernel Memory Address Space: all the memory can be accessed
  - No memory protection!!!
  - Kernel-space code can easily corrupt important data structures!
- No standard runtime
  - C code cannot include `<stdio.h>` and friends...
  - No standard C library!

Kernel Programming

- The kernel must provide the utility functions to be used
  - Example, no `printf()`, but `printk()`...
- Errors do not result in segmentation faults...
- ...But can cause system crashes!
- Other weird details
  - No floating point (do not use `float` or `double`)
  - Small stack ($4KB$ or $8KB$)
  - Atomic contexts, ...

Kernel Programming Language

- OS kernels can be coded in many different languages
  - But some amount of Assembly is needed...
- For some languages, additional restrictions apply (example: for C++, generally no RTTI)
  - And in order to use some languages the kernel must implement a large runtime...
- Kernels are generally coded in C or C++ (with restrictions)
  - After all, the C language has been invented exactly for this purpose!
- Simplest choice: C + some Assembly (inline and not)

Example: the Linux Kernel

- The Linux kernel uses C
- Subset of C99 + some extensions (`likely()` / `unlikely()` annotations, etc...)
- As said, no access to standard libraries
  - Different set of header files and utility functions
- Strict coding style to control the quality (`Documentation/CodingStyle`, `scripts/checkpatch.pl`)
- Some Assembly is used (for entry points, etc...)
- Example: Linked Lists (`include/linux/list.h`)

First Adventures in Kernel Land

- First experiments with kernel programming
  - Should we write our own kernel?
  - Or use an existing kernel as a basis?
- Our own didactic kernel: simpler, maybe we learn more...
  - ...But we can easily get lost in low-level hw details!!!
- Work on an existing kernel: it might be more complex...
  - ...But we can focus on the aspects we are interested in
  - The rest of the kernel already exists!