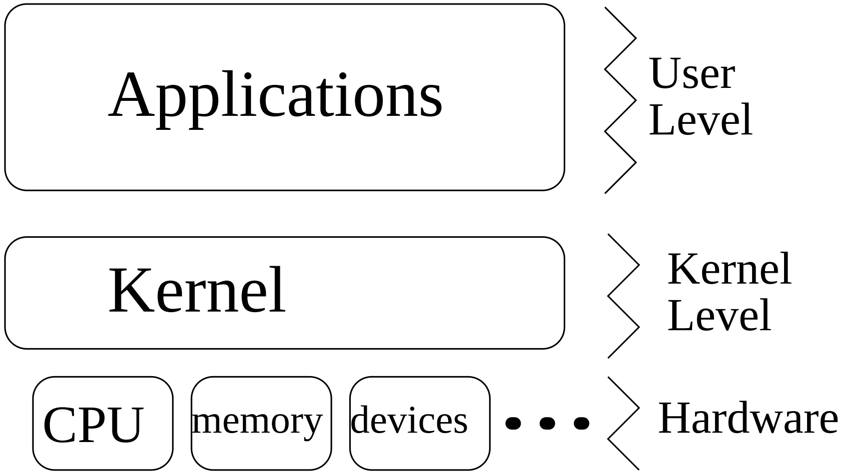
Real Time Operating Systems

Cross Compiling

Luca Abeni

The Kernel

- Kernel → component of an OS that directly interacts with the hardware
  - Runs in privileged mode (Kernel Space → KS)
  - User Level ⇔ Kernel Level switch through special CPU instructions (INT, TRAP, sysenter / sysexit, . . .)
  - User Level invokes *system calls* or IPCs
- Kernel Responsibilities
  - Process management
  - Memory management
  - Device management (Drivers...)
  - System Calls



System Libraries

- Applications generally don’t invoke system calls directly
- They generally use *system libraries* (like glibc), which
  - Provide a more advanced user interface (example: `fopen()` vs `open()`)
  - Hide the US ⇔ KS switches
  - Provide some kind of stable ABI (application binary interface)

Static vs Shared Libraries

- Libraries can be *static* or *dynamic*
- Static libraries (`.a`)
  - Collections of object files (`.o`)
  - Application linked to a static library ⇒ the needed objects are included into the executable
  - Only needed to compile the application
- Dynamic libraries (`.so`, shared objects)
  - Are not included in the executable
  - Application linked to a dynamic library ⇒ only the library symbols names are written in the executable
  - Actual linking is performed at loading time
  - `.so` files are needed to execute the application

Embedded Development

- Embedded systems are generally based on low power CPUs . . .
- . . .And have not much ram or big disks
- ⇒ not suitable for hosting development tools
    - Development is often performed by using 2 different machines: *host* and *guest*
    - Guest: the embedded machine; Host: the machine used to compile
    - Host and Guest often have different CPUs and architectures
    - ⇒ *cross-compiling* is needed

Cross-Compilers

- Cross Compiler: runs on the Host, but produces binaries for the Target
- For many embedded systems, cross-compilation is the only way to build programs (as the target hardware does not have the resources or capabilities to compile code on its own)
- Cross-Compiling environment: cross-compiler (and some related utilities) + libraries (at least system libraries, static or dynamic)
  - C compiler and C library are often strictly interconnected
  - $\Rightarrow$ building (and using) a proper cross-compiling environment is not easy

Cross-Compilers Internals - gcc

- gcc: Gnu Compiler Collection
  - Compiler transforming high-level (C, C++, etc...) code into assembly code (`.s` files, machine dependant)
  - Assembler `as`, transforming assembly in machine language (`.o` files, binary)
  - Linker `ld`, transforming a set of `.o` files and libraries in an executable (ELF, COFF, PE, . . .) file
  - `ar`, `nm`, `objdump`, . . .
- `gcc -S`: run only the compiler; `gcc -c`: run compiler and assembler, . . .

Cross-Compilers - Dependencies

- Assembler, linker, and similar programs are part of the *binutils* package
  - gcc depends on binutils
- As already seen, the compiler need standard libraries to generate working executables
  - gcc depends on glibc, or similar libraries (uclibc, etc...)
- But glibc must be compiled using gcc...
  - Circular dependency?
- ⇒ This is why building a Cross-Compiler can be tricky...

Cross-Configuring GNU Packages

- gcc, binutils, etc... → GNU tools
- `configure` script generated by automake / autoconf
  `--host=`, `--target=`, ...
- Configuration Name (configuration triplet):
  cpu-manufacturer-operating_system
- Systems which support different kernels and OSs → a fourth optional *kernel* field can be added:
  cpu-manufacturer-kernel-operating_system
- Examples:
  - mips-dec-ultrix
  - i586-pc-linux-gnu
  - arm-unknown-elf

Configuration Names

- CPU: type of processor used on the system (tipically ‘i386’, or ‘sparc’, or specific variants like ‘mipsel’)
- Manufacturer: somewhat freeform string indicating the manufacturer of the system (often ‘unknown’, ‘pc’, . . .)
- Operating System: name of the operating system (system libraries matter)
  - Typical embedded systems do not run any operating system. . .
  - ⇒ the object file format, such as ‘elf’ or ‘coff’ is used
- Kernel: mainly used for GNU/Linux systems (example: ‘i586-pc-linux-gnulibc1’)
  - the kernel (‘linux’) is separated from the OS, (ex: ‘gnu’)

Building a gcc Cross-Compiler - 1

- First of all, build binutils

  ```
  ./configure --target=arm-unknown-linux-gnu
  --host=i686-host_pc-linux-gnu --prefix=...
  --disable-nls
  ```

  - Generally, it is not needed to specify `--host=` (config.guess can guess it)
- Then, install headers needed to build gcc
  - *Sanitized* kernel headers
  - glibc headers
- Then compile gcc

Building a gcc Cross-Compiler - 2

- gcc must be built 2 times
  - First, to build glibc (no threads, no shared libraries, etc...)
  - Then, a full version after building glibc
- After building gcc the first time, glibc is built
- Then, a fully working gcc (using the glibc we just compiled) can be finally built
  - Support for threads, the shared libraries we just built, etc
- For non-x86 architectures, some patches are sometimes needed

Helpful Scripts

- As seen, correctly building a cross-compiler can be difficult, long, and boring...
- ... But there are scripts doing the dirty work for us!
  - crosstool `http://kegel.com/crosstool`
- A slightly different (but more detailed) description can be found on the eglibc web site: `www.eglibc.org`

An Example: ARM Crosscompiler

- Download it from `www.dit.unitn.it/~abeni/RTOS/Cross/cross.tgz`
- Untar it in `/tmp` and properly set the path:

```
cd /tmp
tar xvzf cross.tgz #use the right path instead of cross.tgz
PATH=$PATH:/tmp/Cross/gcc-4.1.0-glibc-2.3.2/arm-unknown-linux-gnu/bin
```

- Ready to compile: try `arm-unknown-linux-gnu-gcc -v`
- It is an ARM crosscompiler built with crosstool
    - gcc 4.1.0
    - glibc 2.3.2

The Crosscompiler

- The crosscompiler is installed in `/tmp/Cross/gcc-4.1.0-glibc-2.3.2/arm-unknown-linux-gnu`
- In particular, the `.../bin` directory contains gcc and the binutils
  - All the commands begin with `arm-unknown-linux-gnu-`
  - Compile a dynamic executable with `arm-unknown-linux-gcc hello.c`
  - Static executable: `arm-unknown-linux-gcc -static hello.c`
- We obtain ARM executables... How to run them?
  - ARM Emulator: Qemu!
  - `qemu-arm a.out`

QEMU

- QEMU is a generic (open source) CPU and machine emulator
  - It also is a virtualizer, but we are not interested in that
  - Generic: it supports different CPU models. We are interested in ARM
  - It emulates both CPU and system
- QEMU as a CPU emulator: permits to execute Linux programs compiled for a different CPU. Example: ARM → `quemu-arm`
- To execute a static ARM program, `qemu-arm <program_name>`
- What about dynamic executables?

QEMU and Dynamic Executables

- To run a dynamic executable, the system libraries must be dynamically linked to it
- This happens at load time
- QEMU can load dynamic libraries, but you have to provide a path to them
  - `-L` option
- `qemu-arm -L <path to libraries> <program name>`
  - For example:

```
qemu-arm -L \
.../gcc-4.1.0-glibc-2.3.2/arm-unknown-linux-gnu/arm-unknown-linux-gnu \
/tmp/a.out
```