Cross Compiling

Real Time Operating Systems and Middleware

Luca Abeni
luca.abeni@unitn.it
The Kernel

- Kernel \(\rightarrow\) OS component interacting with hardware
  - Runs in privileged mode (Kernel Space \(\rightarrow\) KS)
  - User Level \(\leftrightarrow\) Kernel Level switch through special CPU instructions (INT, TRAP, ...)
  - User Level invokes system calls or IPCs

Kernel Responsibilities
- Process management
- Memory management
- Device management
- System Calls
Applications generally don’t invoke system calls directly

They generally use system libraries (like glibc), which

- Provide a more advanced user interface (example: \texttt{fopen()} vs \texttt{open()})
- Hide the US $\Leftrightarrow$ KS switches
- Provide some kind of stable ABI (application binary interface)
● Libraries can be *static* or *dynamic*
  ● `<libname>.a` vs `<libname>.so`

● 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 (\texttt{.so}, shared objects)

- Are not included in the executable
- Application linked to a dynamic library \(\Rightarrow\) only the library symbols names are written in the executable
- Actual linking is performed at loading time
- \texttt{.so} files are needed to execute the application

Linking static libraries produces larger executables...

...But these executables are “self contained”
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
- Separate the Host environment from the Target environment
- Embedded systems: sometimes, scarce resources
  - No disks / small (solid state) disks
  - Reduced computational power
  - ...
- In some cases, cross-compilation is the only way to build programs!
Cross-Compiling Environments

- Cross-Compiling environment
  - Cross-compiler (and some related utilities)
  - Libraries (at least system libraries)
    - Static or dynamic

- C compiler and C library: strictly interconnected
  - $\Rightarrow$ building (and using) a proper cross-compiling environment is not easy
• gcc: Gnu Compiler Collection

  • **Compiler**: high-level (C, C++, etc...) code → assembly code (.s files, machine dependant)

  • **Assembler** `as`: assembly → machine language (.o files, binary)

  • **Linker** `ld`: multiple .o files + libraries → 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
- *ld* needs standard libraries to generate executables
  - gcc depends on a *standard C library*
- But this library must be compiled using gcc...
  - Circular dependency?
  - Building a Cross-Compiler can be tricky...
Cross-Configuring GNU Packages

- gcc, binutils, etc... \(\rightarrow\) 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:
  `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 (typically ‘i386’, or ‘sparc’, or specific variants like ‘mipsel’)

- **manufacturer**: freeform string indicating the manufacturer of the system (often ‘unknown’, ‘pc’, …)

- **operating_system**: name of the OS (system libraries matter)
  - Some embedded systems do not run any OS…
  - ⇒ use the object file format, such as ‘elf’ or ‘coff’
Kernel vs OS

- Sometimes, no 1 ↔ 1 correspondance between OS and kernel
  - This mainly happens on linux-based systems
- The configuration name can specify both kernel and OS
  - Example: ‘i586-pc-linux-gnulibc1’ vs ‘i586-pc-linux-gnu’
  - The kernel (‘linux’) is separated from the OS
  - The OS depends on the used system libraries (‘gnu’ → glibc, …)
First of all, build binutils

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

Generally, `--host=` is not needed (config.guess can guess it)
Building a gcc Cross-Compiler - Step 2: system headers

- Then, install some header files needed to build gcc
- Some headers provided by the Linux kernel (API for syscalls)
- Other headers provided by the standard C library (API for standard C functions)
  - Sanitized kernel headers
  - glibc headers
Building a gcc Cross-Compiler - Step 3: gcc

- Rember? Circular dependency with standard C library...
  - How to break it?

- gcc must be built 2 times
  - First, to build glibc (no threads, no shared libraries, etc...)
  - Then, a full version after building glibc

- The “first gcc build” (stage1) can compile libraries, but not applications
Building a gcc Cross-Compiler - Step 4: 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](http://kegel.com/crosstool)
- A slightly different (but more detailed) description can be found on the eglibc web site: [www.eglibc.org](http://www.eglibc.org)
An Example: ARM Crosscompiler

- Download it from
  www.dit.unitn.it/~abeni/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-gnu-gcc hello.c
  
  - Static executable: arm-unknown-linux-gnu-gcc -static hello.c
Testing the Crosscompiler

- Working ARM cross-compiler
  - Runs on Intel-based PCs
  - Generates ARM executables

- So, we now have an ARM executable... How to run it?

- Can I test the generated executable without using an ARM board?
  - ARM Emulator: Qemu!
  - qemu-arm a.out
QEMU

- QEMU: **generic** (open source) emulator
  - Can also do virtualization
  - Generic: it supports different CPU models ARM
  - Can emulate CPU only or a whole system

- QEMU as a CPU emulator: executes Linux programs compiled for a different CPU. Example: ARM → `qemu-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 /tmp/Cross/gcc-4.1.0-glibc-2.3.2/arm-unknown-linux-gnu/arm-unknown-linux-gnu /tmp/a.out
```