Lecture 2

Juni Kim

6.S913

January 21, 2026

Lecture Objectives

- Understand what a build pipeline should do and what invariants it must respect
- Understand C compilation pipelines, environment variables, and the most common options to pass to the C compiler.
- **Reminder: Do Attendance!**

Invariants you need to respect

Build script behavior should be invariant to:

- The program invoker's working directory (no relative path in your program should be undefined unless that is what you want)
- External environment variables (unless explicitly specified as a parameter)
- Filesystem state across multiple runs

You want to avoid as many required conditions for running the program as possible (your "users" don't want to suffer in setup hell)

Host vs Target Distinction (From Lecture 1)

- You are constructing a target system with different characteristics from your host system.
- Your goal is to construct a correct filesystem layout that has everything that you would expect in a basic Linux system:
  - Programs
  - Libraries
  - Config files
  - Directories (more on next slide)
- Later we will use this to create a disk image.

Source vs Artifact Distinction

- Source: all the code provided by the developers + build scripts
- Artifacts: anything generated by the build system
- If you deleted this file and rebuilt everything, do you get the same result?
- Would you commit this to your repository or gitignore it?

Blackboard exercise: classify source vs artifact.

Filesystem Hierarchy Standard

It's probably good to go over this so you can pattern match paths described later.

- Many build systems assume a conventional filesystem layout.
- We will be crafting a directory that contains a filesystem in this format (serves as the template for our disk image and initramfs)
- The most important directories for this course are:
  - `/usr/include` — headers for userspace programs
  - `/usr/lib` — libraries for userspace programs
  - `/usr/bin` — executables for userspace programs
  - `/etc` — system configuration
  - `/lib` — critical runtime libraries
- Build options like `--prefix=/usr` exist to target this layout.

C Dependency Management

GCC Parts

Crudely,

1. C Preprocessor (Source code to Source Code)
2. C Compiler (Source code to binary object code w/ symbols)
3. C Linker (Combines objects together)

Source Code:

- Is human readable C or Assembly (among other languages)
- Not readable by machines

Object Code contains:

- Symbols corresponding to functions for the linker to do relocation.
- Computer-readable binary instructions

C basics

- Headers (Preprocessor copies them to source files)
- Objects and Libraries (Linker responsibilities)

```c
// main.c
#include <helloworld.h>
int main() { hello(); }
```

```c
// helloworld.h
void hello();
```

```c
// helloworld.c, gets compiled into libhelloworld.a
#include <stdio.h>
#include "helloworld.h"
void hello() { printf("hello world!\n"); }
```

The Build Process

We first create libhelloworld.a, then tell the compiler where to find the headers and implementations when compiling main

```
#!/usr/bin/env bash
set -euo pipefail
DIR="$(realpath "$(dirname "$0" )" )"
cd "$DIR"

CFLAGS="-Os -Wall -I$DIR"
LDFLAGS="-L$DIR -static"
gcc $CFLAGS -c helloworld.c -o helloworld.o
ar rcs libhelloworld.a helloworld.o
gcc $CFLAGS $LDFLAGS main.c -lhelloworld -o main
```

GCC Flags

Flags are passed to either make (See handout) or configure as options or environment variables. They will be passed to `gcc` or `ld` as command-line arguments.

Compiler flags (CFLAGS and CPPFLAGS)

- `-I/path/to/headers`
- -nostdinc (nuke include defaults)
- -Wno-error and other warning compiler flags (**Mitigates an entire class of silent failures**)

Linker flags (LDFLAGS)

- `-L/path/to/libraries`
- -static (link statically)
- `-l{libraryname}` (usually the build system takes care of this)

FHS and C Programs

Which compiler flags would be most related to the following FHS directories?

- `/usr/include`
- `/usr/lib`
- `/usr/bin`
- `/lib`
- `/etc`
- `/include`

FHS and C Programs

Which compiler flags would be most related to the following FHS directories?

- `/usr/include` – headers, so `-I/usr/include`
- `/usr/lib` – binary libs, so `-L/usr/lib`
- `/usr/bin` – executables, so neither
- `/lib` – binary libs, so `-L/lib`
- `/etc` – config files, so neither
- `/include`– headers, so `-I/include`

Live Demo; Crack open a Debian box

Default C Flags

- Your distro's gcc will ship with a sane default of libraries and headers to check
- These tend to be anti-helpful for our assignment (except for build dependencies, see busybox and kernel).
- `-nostdinc` helps us eliminate these defaults.
- `gcc -E -x c - -v < /dev/null` can help you figure out what the compiler is seeing
  - `-E` says stop at preprocessor
  - `-x c` says interpret as c
  - `-` says look at stdin
  - `-v` says be verbose
  - `< /dev/null` says standard input comes from a device that has nothing.

Make

- A dependency scheduler for C projects
- Prevents re-compiling files whose dependencies haven't changed.
- Requires a `Makefile` in the same directory.

Below are some general patterns you can expect to use for make:

```
# note make requires being in the same directory as the source code unless
    you use -C
# make with 8 parallel jobs
make -j8
# install to the default location (probably not what you want)
make -j8 install
# what you probably want, differs by source
make -j8 install DESTDIR=/path/to/rootfs
```

The build system from before with make

You should also consult the handout about make to understand how to write Makefiles.

```
CFLAGS=-O2 -Wall -I/workspace
LDFLAGS=-L/workspace -static
main: libhelloworld.a
  # you need to put the archive import after the main.c
  gcc $(CFLAGS) $(LDFLAGS) main.c -lhelloworld -o main
libhelloworld.a: helloworld.o
  ar rcs libhelloworld.a helloworld.o
helloworld.o:
  gcc $(CFLAGS) -c helloworld.c -o helloworld.o
clean:
  rm -rf *.o main *.a
.PHONY: clean
```

Automake

- Is responsible for creating the `configure` scripts that you will often see in source tarballs
- `configure` scripts examine your system, the compiler you provided, what libraries are available, ...
- You need to provide proper environment variables and compile flags at this stage so it generates **the proper Makefile**.
- Often you want this stage to emit source files outside the source tree
- If you set flags after `./configure` it might be too late.
- Run `./configure --help` to figure out which options to use.

Files in a Standard C Project

A typical C project using autoconf / automake will contain:

- **Build system metadata**
  - `configure` (generated by autoconf)
  - `Makefile.in` → `Makefile`
- **Source code**
  - `[files].c` — implementations
  - `[files].h` — interfaces
- **Human-facing documentation**
  - `README`, `INSTALL`

A Sample Pipeline

```
#!/usr/bin/env bash
# we configure in $SCRATCHDIR for out of tree build
set -euo pipefail
cd "$SCRATCHDIR"
export CC="my-gcc-binary"
export CFLAGS="-I/my/rootfs/usr/include -nostdinc -static"
export LDFLAGS="-L/my/rootfs/usr/lib -static"
# configure will pass above compile flags into Makefile
"$SRCDIR/configure" --prefix=/usr
# workdir now has Makefile
make "-j$(nproc)"
make "-j$(nproc)" install DESTDIR=/my/rootfs/
```

- Note **out of tree builds**: build artifacts are kept separate from the source tree.
- Separation of concerns + certain programs (like gcc) only support out of tree builds.

Libc

- Basic POSIX headers like `<unistd.h>`, `<fcntl.h>`, `<stdio.h>`, are implemented by a libc.
- Glibc is on most Linux distros (including your host container)
- We will be using musl (simpler + friendly to static linking) for our static system.
- You need to control your compile flags, environment variables, and options so you don't put glibc headers and libraries as dependencies (you want musl dependencies).
- Binary linked against glibc $\rightarrow$ kaboom

“No such file or directory” Trap

If executing a binary yields this error, it’s one of the following

1. The file literally doesn’t exist
2. The executable is dynamically linked + the interpreter for the executable doesn’t exist
3. Some library the executable needs doesn’t exist (Might fail at link time?)

- You should always use `readelf -l` to inspect ELF headers and check if an interpreter is requested by an executable.
- You either want no INTERP header or `/lib/ld-musl-$ARCH.so.1`
- `/lib/ld-linux-$ARCH.so.1` is bad (glibc)

More about libc linking

helloworld.c

```
#include<stdio.h>
int main() {printf("hello world!\n");}
```

What we do in the shell:

```
$ gcc helloworld.c -o dynamic
$ ldd dynamic
  linux-vdso.so.1 (0x0000ffffa717c000)
  libc.so.6 => /lib/aarch64-linux-gnu/libc.so.6 (0x0000ffffa6f40000)
  /lib/ld-linux-aarch64.so.1 (0x0000ffffa7130000)
$ gcc helloworld.c -static -o static
$ chroot . /dynamic
chroot: failed to run command '/dynamic': No such file or directory
$ chroot . /static
hello world!
```

Note: glibc static linking is not really supported.

More Libc Linking

Slide reserved for live demo with docker image `lecture2`.

```
// helloworld.c
#include<stdio.h>
int main() {printf("hello world!\n");}
```

I have prepared a musl-based gentoo root filesystem at /src/rootfs.

```
$ gcc -static helloworld.c -o helloworld-glibc
$ gcc -nostdinc -I/src/rootfs/usr/include -L/src/rootfs/usr/include -static
    helloworld.c -o helloworld-musl
$ strings helloworld-glibc
$ strings helloworld-musl
```

Kernel User Space Headers

- The Linux kernel provides a set of headers intended for **userspace** programs.
- These headers define syscalls, ioctl numbers, constants within the kernel, ...
- They live under paths like `linux/` and `asm/` (e.g. `<linux/stat.h>`) in `/usr/include`.
- These headers are **not** part of libc, but libc depends on them.
- To build your libc, you will need to install these headers into your rootfs first (see handout for more info).

Attack Plan for Bootstrapping Userspace

1. Kernel Headers
2. Libc (we use musl)
3. Compile everything else

If you don't know how to get started:

1. Try compiling everything by hand first
2. Reproduce everything into a script that follows our guidelines.

Up Next

- Lab hours here immediately after lecture.
- We'll start going over:
  - Kernel internals and options
  - Why do you need an initramfs.
  - Actually using qemu to boot up a system