Lecture 2 Handout

Juni Kim

January 21, 2026

Contents

§1 On Make

§1.1 Parallelizing Make

We strongly encourage you to enable parallelization in your build scripts. If you allow for compilation with a certain number of jobs, make will distribute the work so that all artifacts are built in the correct order. As discussed in lecture, we recommend should use `make -j$(nproc)`. nproc is a program that will roughly tell you how many cpu cores you have on your system.

§1.2 Notice about working directories

Make is also sensitive to the current working directory. By default, it looks for a `Makefile` in the current directory and interprets all relative paths with respect to that directory.

To avoid relying on the caller's working directory, make provides the `-C` option, which instructs make to change directories before reading the Makefile:

```
make -C /path/to/source
```

This is functionally equivalent to:

```
cd /path/to/source
make
```

but is preferred in scripts because it avoids modifying global shell state and makes directory assumptions explicit.

The `-C` option composes cleanly with other make options and variable assignments:

```
make -C "/kernel/source/dir" O="/kernel/build" -j8
```

In this course, the use of `make -C` is strongly encouraged whenever you are invoking make on a source tree that is not the script's current working directory. This ensures build scripts remain working-directory invariant and robust to changes in how they are invoked.

§1.3 Deeper Dive into how make works

This course does not expect you to write Makefiles, but it is hoped that this section aids in understanding how make gets used.

Make operates over a declarative configuration language, usually written in a file called `Makefile`. A Makefile consists of *targets*, their *dependencies*, and the *commands* needed to produce those targets.

A minimal example looks like this:

```
output.o: input.c
  gcc -c input.c -o output.o
```

Here, `output.o` is the target, `input.c` is a dependency, and the indented line is the command that produces the target. Make will only run the command if the target does not exist or is older than one of its dependencies.

Targets are resolved recursively. When you invoke `make`, it selects a *default target*, which is simply the **first target listed** in the Makefile, and attempts to build it by first building all of its dependencies.

Makefiles also support variables. Variables can be defined inside the Makefile:

```
CC = gcc
CFLAGS = -O2
```

and referenced using `$(VAR)`. Variables may be overridden either via environment variables or via arguments passed directly to make:

```
make CC=clang CFLAGS="-O0 -g"
# the top and bottom do the same thing, given the Makefile does not override
    variables.
export CC=clang CFLAGS="-O0 -g"
make
```

Variables passed on the command line take precedence over those defined in the Makefile. This is how flags such as `CFLAGS`, `CPPFLAGS`, and `LDFLAGS` are commonly injected into third-party build systems.

Some targets do not represent real files. These are marked as `.PHONY` targets to prevent make from becoming confused if a file with the same name exists. A common example is `clean`:

```
.PHONY: clean
clean:
  rm -rf build
```

Without `.PHONY`, make may incorrectly believe the target is already up-to-date and skip the command.

§1.4 Debugging Tools

When debugging build failures or runtime issues related to dependency management, it is often necessary to inspect the resulting binaries directly.

The `nm` utility lists symbols contained in object files or binaries. This is useful when diagnosing unresolved symbols or confirming that a binary was linked against the expected implementation of a function.

```
nm mybinary | grep malloc
```

The `ldd` command lists dynamic dependencies of an executable. Note that `ldd` works by invoking the program's dynamic linker; it does *not* simply read metadata. As a result, it may fail if the dynamic linker path is incorrect or missing.

If `ldd` prints:

```
not a dynamically linked executable
```

This indicates a statically linked binary, which is expected and acceptable in this course.

If `ldd` reports a dynamic linker path associated with glibc (for example `/lib64/ld-linux-x86-64.so.2`), you are almost certainly mixing host and target libc environments, which will lead to runtime failures.

For safer inspection, you can use `readelf` to inspect ELF headers without executing the binary:

```
readelf -l mybinary
```

This allows you to verify the interpreter (dynamic linker) path and other ELF metadata directly.

§1.5 Warnings, `-Werror`, and Autoconf Pitfalls

Many upstream projects treat compiler warnings as errors by enabling `-Werror`. While this can be useful during development, it is often *actively harmful* in constrained or nonstandard build environments.

In this course, you will frequently build software under non-standard conditions, including a non-host libc, static linking, and headers installed into a synthetic root fs. In such environments, warnings are expected and do *not* necessarily indicate incorrect behavior. Treating warnings as fatal errors can cause otherwise-correct software to fail to build.

¶ **Autoconf Failure Mode** Many projects use `./configure` scripts generated by autoconf. These scripts test for system features by compiling small test programs. If a warning is promoted to an error, a feature test may fail even though the feature is actually available.

This failure is often silent and appears as:

```
checking for feature X... no
```

even though the real cause was a warning treated as an error.

For the purposes of this course:

- You are *allowed and encouraged* to disable warnings-as-errors.
- Adding `-Wno-error` to `CFLAGS` is acceptable.
- Removing `-Werror` from upstream build systems is acceptable.

§1.6 Dynamic Linking

You are not expected to build dynamically linked binaries in this course, but this section exists to help you understand them better.

Dynamic linking refers to resolving library dependencies at runtime rather than embedding all required code into the executable at link time. When building shared libraries, the compiler is typically invoked with the `-shared` flag, producing `.so` files instead of static `.a` archives.

Executables that are dynamically linked contain ELF metadata that specifies an absolute path to the dynamic linker. This path is fixed at link time and must exist on the system where the binary is executed.

For example, glibc-linked binaries typically reference:

```
/lib64/ld-linux-x86-64.so.2
```

while musl-linked binaries reference a different absolute path, such as:

```
/lib/ld-musl-x86_64.so.1
```

Because this path is absolute and fixed, dynamically linked binaries are tightly coupled to the filesystem layout of the system they were built for.

The `LD_LIBRARY_PATH` environment variable can be used to influence where the dynamic linker searches for shared libraries at runtime. This mechanism is primarily intended for debugging and should be used sparingly, as it introduces implicit and global state.

In this course, you are not required to rely on dynamic linking, but understanding how it works is helpful when interpreting linker errors, runtime failures, or unexpected host contamination.

§2 Build system exceptions

You can expect many of the software packages in our class to use the autoconf conventions as described in class. Below are the notable exceptions.

§2.1 Kernel Headers

You will almost certainly have to preemptively install the Linux headers (compiling musl may require those headers). Header installation follows the conventions below. Most important to notice are how we supply `O` and `INSTALL_HDR_PATH` as options to make.

```
make -C "/kernel/source" O="/kernel/build/directory" INSTALL_HDR_PATH="/path/to/
    your/install/prefix" headers_install
```

§2.2 The Kernel

Building up the kernel requires generating a config first, which you can do by running `make menuconfig`. Once you save all your options, you should get a `.config` file which you can then copy into your own codebase so you can automatically copy it over later.

Note that the above step is not optional, as our automated scripts will run with the assumption your build scripts inject a `.config` into the kernel build tree.

Likewise, for building up the kernel, you can do something similar (with the `{your target}` placeholder being replaced with something or nothing at all if you want the default target)

```
make -C "/kernel/source" O="/kernel/build/directory" {your target}
```

There is no separate `make install` for the kernel; you have to fish out the kernel image from the kernel build directory. For x86, you should expect your kernel to be at `$KERNEL_BUILD_DIR/arch/x86/boot/` while for aarch64, your kernel will be at `$KERNEL_BUILD_DIR/arch/arm64/boot/Image`.

§2.3 Busybox

Busybox has a similar `make menuconfig` convention for generating a config. Just like the kernel, you can `make menuconfig`, save your options, and copy the newly created `.config` file into your own codebase for copying later.

When installing busybox you use `CONFIG_PREFIX` instead of `DESTDIR`, as shown below:

```
make install CONFIG_PREFIX=/path/to/your/rootfs
```