Lecture 1

Juni Kim

6.S913

January 20, 2026

Course Logistics

- **Registered Students**: Final Assignment due Friday January 30, 11:59PM, no extensions
- Website at `https://junic.kim/en/6s913`
- Check the portal at `https://6s913.mit.junic.kim` for submission.
- Communication over Piazza `https://piazza.com/mit/spring2026/6s913`
- Lectures on Tuesday, Wednesday, Friday, Monday, Thursday
- First four lectures come with an associated handout. You are expected to read through this handout as needed.
- Alternative grading available if you attend all 4 lectures.

What this Course is Not

- A “How to use the command line” tutorial - many other courses for this.
- “How to use Git” - You will be responsible for cloning, committing, and recovering from mistakes, although we may post resources later
- Operating systems kernels class (Go check out 6.1810!)

Course Goals

- Assignment: compose a build system for a Linux-based OS (a lot of other courses will abstract this away)
- Reasoning out bugs and writing deterministic programs in a stateful environment (hostile to deterministic behavior)
- Along the way, understand the most basic components of a modern operating system

This course will move fast! Get started and ask for help ASAP.

Special Note on Cargo Culting

- This is deceptively easy to do in this course, we want to strongly discourage it!
- You copy paste instructions because they worked somewhere else without understanding exactly why.
- For every program you use, you should understand all options and what the program will assume.
- Your scripts will be built in hostile environments. “It works on my machine” is unacceptable.
- Note: concretely understanding “setup issues” is the *entire point* of the class. It is expected that you give a faithful effort to triage and resolve such issues.
- Note on AI: it’s everywhere, aim to use it like a search engine or to assist in debugging work. Agentic IDE’s are discouraged.

Today's Lecture

- Writing "correct" shell scripts that exhibit defined behavior in a hostile environment (your users)
- Host vs Target distinction
- The role of Docker (host system standardization)

This is a lot of content! If you're stuck please ask questions.

Mental Models you should remember

- Shell scripts ingest and manipulate environment state.
- Shell scripts accumulate effects that need explicit checking.
- Locating yourself in the filesystem.
- Shell control flow as exit-code driven
- Host-Target distinction
- (Additional) Image vs Containers in Docker

Shell Scripts and State

- The shell is your main interface with the operating system. You can run other programs, manage the filesystem, etc.
- Usually you interface via the CLI, but you can automate this with scripts.
- Side effects are the **entire point** of many shell scripts (e.g. setup scripts).

External State

- Working Directory (!!!! if you use relative paths)
- Filesystem contents
- Environment variables passed by the system

- Do not assume shell scripts are like 6.101 assignments.
- You are responsible for writing your own reset button (reverting any changes the shell script made if necessary).

Understanding the File System

Locating Yourself in the File System

- You **must** internalize this for the class to make sense.
- There are relative paths and absolute paths (know which one you use).
- You must reason correctly about paths or else you will get `no such file` errors everywhere.
- ASK: where is the file I need, and where is it relative to where I am?

Relative Path Examples

- `../myfile.sh`
- `./myfile.sh`
- `myfile.sh`
- `some/dir/../myfile.sh`

Absolute Path Examples

- `/workspace/myfile.sh`
- `/home/myuser/myfile.sh`

Working Directory

- Relative paths are relative **to what**??
- `pwd` gets the current working dir, `cd` changes the working directory.
- Most of the programs and arguments you invoke will be working directory dependent. (relative paths)
- If you have a script that is attached to a certain repository, you need to work to prevent this.

Why is this so emphasized?

- It helps lock down assumptions about the state of the other system. They didn't mess with your repo means your script works.
- A lot of your scripts will be tightly coupled with other parts of your source code (e.g. copying over config files or running other scripts in the same repo).
- Naive `cp ./myconfig $DIST/config` might not work

Live Demo with paths!

Working Directory Demo

Let's say you have the following project structure:

```
/home/junikim/projects/myproject/ <-- repository root
├── .git
├── .gitignore
├── requirements.txt    <-- data file (referenced by scripts)
├── devserver.py        <-- program entry point
└── setup.sh            <-- script entry point (THIS FILE)
```

Bad `setup.sh` (Why is it bad?):

```
#!/usr/bin/env bash
pip install -r ./requirements.txt
python3 ./devserver.py
```

Working Directory Demo

Better `setup.sh`:

```
#!/usr/bin/env bash
DIR="$(realpath "$(dirname "$0" )" )"
cd "$DIR"
# now we can run this knowing all relative paths are relative to repository
    root.
pip install -e ./requirements.txt
python3 ./devserver.py
```

What is dirname and realpath (symlink resolution)?

File System Permissions

Regular Linux files have three main permissions attached to them:

1. Reading (`cat myfile` does this)
2. Writing (If you edit a file with a text editor)
3. Executing (When you run a script)

The kernel can grant or deny any of these permissions to you. `chmod` modifies these permissions.

A Tale of Two Scripts

```python
#!/usr/bin/env python3
def sumn(n):
  return (n * (n+1))//2
print(sumn(int(input("number: "))))
```

```bash
#!/bin/bash
mkdir out
cd out
grep foo text.txt > result
echo "done"
```

Running Processes

"Commands" as Programs

- Some set of shell built-ins
- But most commands are **names for executable files on your system**
- The filesystem has marked them with executable permissions (`chmod +x`)
- They are just shortened for convenience (we'll get to this)

```
# this is a valid shell script (albeit very brittle).
/bin/mkdir myfolder
/usr/bin/gcc file.c -o file
```

- You can execute your own shell scripts from inside other shell scripts.
- Programs take in options which they can then interpret internally.

Exit Codes

- Some runtimes use sophisticated exceptions, result types
- Errors in UNIX-like operating systems are handled via exit codes.
- Successful programs exit with 0, non-zero means some error.
- Most shell interpreters will store the exit status of the last exit code in `$?` (special bash variable).

```
mkdir -p /tmp/hello
# prints 0
echo "$?"
# prints 1 (because it already exists)
mkdir /tmp/hello
echo "$?"
```

Exit Codes and Propagation

- Programs tend to depend on effects from the last program
- Bash will not exist just because the last program exited unsuccessfully (consider implications on the cli)
- We use `set -euo pipefail` to ensure we fail rather than go into UB.

```
/project/root
├── myfile.txt
└── src
    └── helloify.sh
```

Now consider src/helloify.sh:

```
# what happens if cd succeeds, and what happens if cd fails?
# reason out working directory
cd src/
echo "hello world!" > ../myfile.txt
```

Pipes and Redirection

- The I/O streams to know are stdin (0), stdout(1), stderr(2)
- They are actually called file descriptors (see 6.1810), but just think of them as streams that can be redirected.
- The redirection gets read left to right.
- You can redirect or pipe program stdout into the stdin of another program (pipes).

```
# count the number of files and output that
ls | wc -l
# redirects standard output into a file (truncate file)
ls | wc -l > count.txt
# Same as above, except don't truncate
ls | wc -l >> count.txt
# this will show two lines.
cat count.txt
```

Q: How might we use truncate vs append behavior when constructing a log?

Stdout vs Stderr Demo

```c
// hello.c
#include <unistd.h>
int main() {
  char out[] = "stdout\n";
  char err[] = "stderr\n";
  // write to standard output
  write(1, out, sizeof(out));
  // write to standard error
  write(2, err, sizeof(err));
}
```

```
gcc hello.c -o hello
# this will output "hello" twice
./hello
# this will output one "hello" and one 1
./hello | wc -l
# this will output 2 because we merge stderr into stdout
./hello 2>&1 | wc -l
```

(Blackboard for how the streams get modified)

Environment Variables

Shell Variables are:

- Confined to the process
- Set by the syntax `VAR=value`
- Referenced with dollar sign syntax `VAR="$ANOTHERVAR"`

Environment Variables are:

- External variables that can influence processes
- Passed from parent to child.
- You can access them as variables when writing shell scripts.
- Ordinary shell variables will not become environment variables unless you use the `export` keyword.
- You will often manipulate `PATH`, `CC`, `LDFLAGS`, and a bunch of other environment variables to get the build results you want.
- In python, you extract process env vars via `os.environ`

Environment Variable Demo

tester.sh

```
DIR="$(realpath "$(dirname "$0" )" )"
# regular bash variable (will not get exported)
MYENV=hello
# an environment variable
export MYENV2=hello2
"$DIR/reader.sh"
```

reader.sh

```
# this program will only get env vars marked with "export"
echo "MYENV=$MYENV"
echo "MYENV2=$MYENV2"
```

When tester.sh is executed:

```
$ chmod +x reader.sh tester.sh
$ ./tester.sh
MYENV=
MYENV2=hello2
```

So what is PATH?

- The bane of intro CS students trying to set up their environments
- Basically tells you which executables you can shorthand (instead of writing out a full path).
- Use `command -v` to figure out which executable is being called.
- A colon-separated series of paths which the shell will lookup.
- Homebrew, Virtual Envs, Conda, ...change the world by messing with this variable (`python3` points to a different interpreter)!

Live Demo!

```
export PATH="/usr/bin:/bin"
# bash will search /usr/bin/myprogram, followed by /bin/myprogram.
myprogram
# you can see whether it is /usr/bin/myprogram
command -v myprogram
export PATH="/my/installation:/usr/bin:/bin"
# now bash will search /my/installation first.
command -v myprogram
```

Control Flow

- if, for, and while all exist, and are driven by exit codes
- conditionals are determined by exit code (0 is true, non zero is false!)
- You probably want to use && and || more
- Functions have no scope! They just help section code
- `test` can be used to check for file existence and other utilities.

```
if ! test -f myfile.txt; then
  echo 'hello' > myfile.txt
fi
# below does the same thing as above
! test -f myfile.txt && echo 'hello' > myfile.txt
# so does below
test -f myfile.txt || echo 'hello' > myfile.txt
```

Don't forget `set -euo pipefail` to prevent undefined effects!

Defensive Scripting

A Tale of Two Scripts: Part 2

So what's wrong with the below? What are the possible failure modes?

```
#!/bin/bash
mkdir out
cd out
grep foo text.txt > result
echo "done"
```

A Tale of Two Scripts: Part 2

So what's wrong with the below? What are the possible failure modes?

```
#!/bin/bash
mkdir out
cd out
grep foo text.txt > result
echo "done"
```

- dependent on the script invoker's working directory! (Behavior outside of project root is undefined behavior)
- mkdir or grep not found (The user screwed up `$PATH`)
- errors propagate (no "exceptions" unless forced)
- Bonus: `/bin/bash` is not standard! Only `/bin/sh` is. There are systems that don't have `/bin/bash`

A More Defensive Script

```
#!/usr/bin/env bash
# exit if any part encounters a bad exit code or undefined env var
set -euo pipefail
# parent directory of our script
DIR="$(realpath "$(dirname "$0" )" )"
# -p for idempotency, our working directory is now deterministic.
mkdir -p "$DIR/out" && cd "$DIR/out"
# test for existence first.
test -r text.txt && grep foo text.txt > result
```

Design Decisions

- Do we want silent failure or be loud?
- Do we destroy history of result? (> versus >>)
- We just need to be deliberate with what behavior we allow

Some General Principles

- Even if a script works, you need to be skeptical.
- If your entire working environment and build artifacts were destroyed and then reconstructed, your script must still work or error cleanly.

Host vs Target Systems

Host vs Target

- Will be a huge part of lab pain
- You are on a **Host System**, building artifacts that will go in a separate **Target System** (Blackboard time)
- The target system must be wholly self-sufficient. No dependence on:
  - Host libraries (This one's subtle), **especially the libc**
  - Host services
  - Host configuration files
- You need to carefully manipulate environment variables and options because cross-compiling is not assumed!

```
# working dir is the root of some source tree
export CC="/path/to/custom/cc"
export CFLAGS="-nostdinc -I/custom/include/prefix -static"
export LDFLAGS="-L/custom/lib/prefix -static"
./configure --prefix="/usr"
make
make install DESTDIR="/path/to/your/rootfs"
```

Docker

- Containerization platform we will be using for the class.
- Everyone will be working on a standard Debian Linux environment.
- This standardizes the **host** compiling system.
- Assignment README also has further instructions on usage.

Core Concepts Regarding Docker

- A *container* is a process managed by docker with its own view of the filesystem and network. (`docker run`)
- An *image* is the template for a container (debian)
- You build images and then run containers based off that image.
- You can also execute commands inside a running container like an interactive shell (`docker exec`)
- Live Demo! (Image pull, container start, monitoring, deletion)

For this assignment, there will be:

- Two images (builder, grader)
- Four long-running containers (one for each part, you will run your scripts here)
- Two ephemeral containers (grading)

Next Lecture (Tomorrow same time)

- Build tools
- Understanding the gcc toolchain and general pipeline for building C programs
- Tightly controlling inputs for cross compilation