Lecture 24: Build systems and dependencies

CS 5150, Spring 2025

Administrative Reminders

- In class exam 2: Topics Lecture 14-24 (today)

Previously in 5150…

Dependencies

Internal vs. external dependencies

Internal

- Maintainers' goals are (hopefully) aligned
- Can audit for all uses of a library
- Can coordinate large-scale changes of all code using library (facilitated by monorepo)
- Can manage with source control tools, policies

External

- Cannot assume coordination between library and users
- Cannot enforce compatibility, maintenance policies
- Cannot control release schedule
- Danger of diamond dependency problem
- Domain of dependency management

Where to get dependencies from?

- Defer to users / distributors
  - E.g., List of Debian packages to install
  - Common for libraries, system software (C/C++); often used for "standard" dependencies
  - Build system should confirm that dependencies are satisfied
  - May assume elevated privileges, may mask portability
- "**Vendoring**"
  - Copy third-party source code (or artifacts) into your repository
  - Increases project size, hard to maintain
- Artifact repositories
  - Download binary artifacts and their transitive dependencies
  - E.g., **Maven Central**, **PyPi (Python)**, **Debian packages**
- Source code repositories
  - Download the source code and compile locally
  - E.g., Cargo.io, BSD ports, npm
- Private Cloud Registry

Dependency Management Practices

- **Version pinning**: select the exact dependency version
  - But only applies to direct dependencies
  - Problems?
- Signature and hash verification
- **Lockfiles**: pinning+sig/hash verification for full dependency tree
  - Compiles all dependencies and sub-dependencies (entire dependency tree)
  - Better reproducibility and consistency
- **Dependency confusion attack**: publishing projects with the same name as an internal project to open-source
- **Vulnerability scanning**: scan lock files to check the artifact versions

https://cloud.google.com/blog/topics/developers-practitioners/best-practices-dependency-management

Supply Chain Attack (Example)

- https://pytorch.org/blog/compromised-nightly-dependency
- PyTorch-nightly Linux packages installed via pip during that time installed a dependency, **torchtriton**, which was compromised on the Python Package Index (PyPI) code repository and ran a malicious binary. This is what is known as a **supply chain attack** and directly affects dependencies for packages that are hosted on public package indices.
- A malicious dependency package (**torchtriton**) that was uploaded to the Python Package Index (PyPI) code repository with the same package name as the one shipped on the PyTorch nightly package index.
- This malicious package was being installed instead of the version from the official repository.
- This malicious package contains code that uploads sensitive data from the machine.

Compromised PyTorch-nightly dependency chain between December 25th and December 30th, 2022.

If you installed PyTorch-nightly on Linux via pip between December 25, 2022 and December 30, 2022, please uninstall it and torchtriton immediately, and use the latest nightly binaries (newer than Dec 30th 2022).

```
$ pip3 uninstall -y torch torchvision torchaudio torchtriton
$ pip3 cache purge
```

PyTorch-nightly Linux packages installed via pip during that time installed a dependency, torchtriton, which was compromised on the Python Package Index (PyPI) code repository and ran a malicious binary. This is what is known as a supply chain attack and directly affects dependencies for packages that are hosted on public package indices.

NOTE: Users of the PyTorch **stable** packages **are not** affected by this issue.**

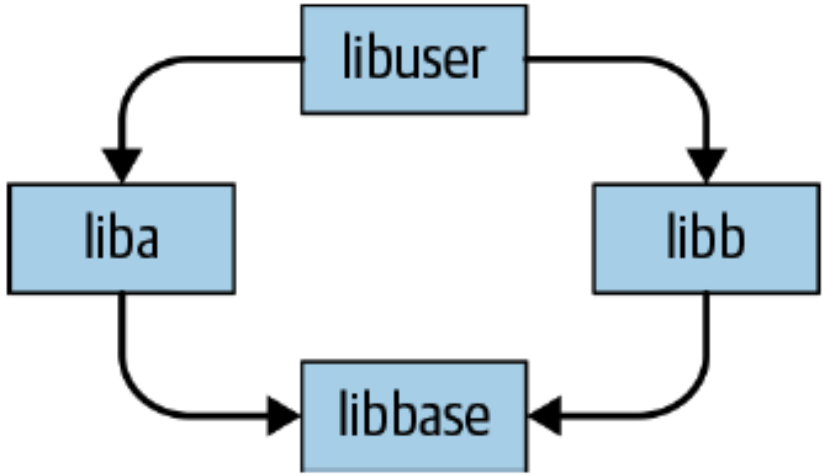
Repository mirrors

- Depending on public repositories is risky
  - What if their servers are not available?
  - What if packages are removed?
  - Do you trust that an artifact will never change?
  - Does your employer's firewall block binaries? Do they need to scan for viruses?

- Can point build tools to an internal repository mirror, rather than the public Internet
  - Tradeoff between maintenance and control

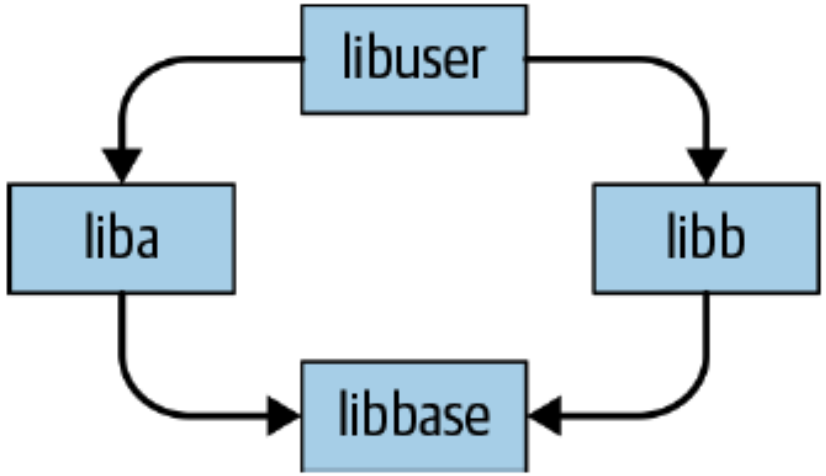
Diamond dependency problem

- Consider an application that uses a computer vision library and a GUI toolkit
- Suppose the CV library depends on libpng-1.4, but the GUI toolkit is linked against libpng-1.2. These versions are incompatible
- What version of libpng can your application link against?



Diamond dependency problem

- Solutions:
  - Downgrade a library
  - Upgrade both (Delta debug ☺)
  - Manually patch



Dependency management

- What versions of dependencies should you import?
- When should you upgrade dependency versions?

- SwE@Google book outlines four options:
  1. Never upgrade
  2. Semantic versioning
  3. Bundled distributions
  4. "Live at HEAD"

Never upgrade (Static Dependency Model)

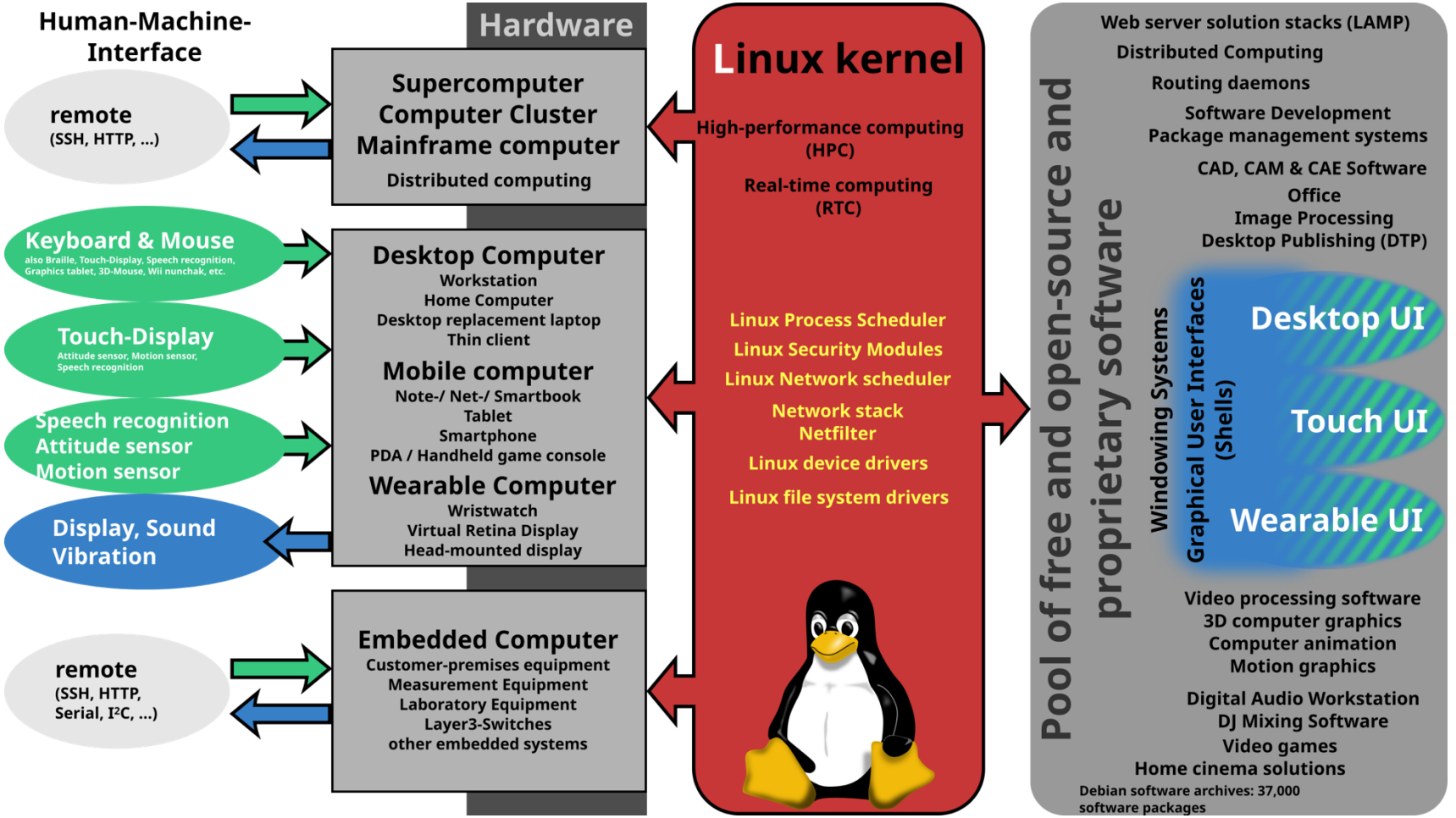
- Predictable
  - Avoids failures due to changes outside of your control
- Natural when starting out, or for short-lived projects
  - Compatible with "vendoring"
- What happens when a dependency has a security vulnerability?
- What happens when a new dependency depends on newer versions of old dependencies?

Bundled distributions

- Defer dependency management to the distribution maintainer
  - Responsible for maintaining compatibility while incorporating security updates
- Depend on the bundle and whatever dependency versions it provides
  - Common for commercial applications
- **Example**:
  - **Linux distributions**: Debian (non-commercial), Fedora Linux (Red Hat), OpenSUSE
  - **Android, ChromeOS**: Built on Linux Kernel
  - More niche: **Raspberry PI OS**
- **Distributors**: responsible for finding, patching, and testing a mutually compatible set of versions to include.
- Limitations:
  - Limits (verified) portability
  - Can't leverage latest features

Linux Distributions

- Typically includes
  - Linux Kernel
  - Package manager
  - Init system
  - GNU tools
  - Networking
  - GUI
  - …



Semantic versioning (SemVer)

- Dependency version numbers obey MAJOR.MINOR.PATCH format
  - Changes to PATCH should be fully compatible (bug fixes, security fixes)
  - Changes to MINOR may add functionality in a backwards-compatible manner
  - Changes to MAJOR indicate API changes (potentially breaking)
- Assumed by many build tools
  - Depend on a specific MAJOR version and a minimum MINOR version
- Challenges
  - Not all dependencies follow this scheme
  - Human maintainers make mistakes
  - **Hyrum's Law**: one person's "bug" is another's "feature"
  - Can be over-constraining (no solution to SAT problem)
    - Heuristics for relaxing some requirements

Hyrum's Law

"With a sufficient number of users, every observable behavior of your system will be depended upon by someone"

- SemVer's patch versions may not be "safe" (beyond input-output spec of an API)
  - Adding a delay in time-sensitive API
  - Logging format changes
  - Change order of results in a stream
  - Changing the order of importing dependencies…
- Many of these patches can be "breaking"

SemVer works when…

- Your dependency providers are **accurate** and **responsible** (to avoid human error in SemVer bumps)
- Your dependencies are **fine-grained** (to avoid falsely over-constraining when unused/unrelated APIs in your dependencies are updated, and the associated risk of unsatisfiable SemVer requirements)
- All usage of all APIs **is within the expected usage** (to avoid being broken in surprising fashion by an assumed-compatible change, either directly or in code you depend upon transitively)

Hard to satisfy these when operating at large (Google) scale…

Minimum Version Selection (MVS)

- SemVer: Chooses the newest possible versions of dependencies that satisfy requirements
- **MVS**: Select the lowest satisfiable version
- **Intuition**: Produce **high-fidelity builds** in which dependencies are as close as possible to what the developer used
- Proposed by Russ Cox for Go: https://research.swtch.com/vgo-mvs

PollEv.com/cs5150sp25: Which version would you upgrade to?

You are maintaining a Java web service that uses the library **json-utils** for parsing and generating JSON. Your current version is: 2.4.1

Upgrade reason: Your team needs better performance when serializing large JSON payloads, especially due to recent load testing that revealed bottlenecks. You heard that newer versions have optimized this.

Version	Change Type	Release Notes Summary
2.4.2	Patch	Fixed a memory leak in edge-case deserialization. No API changes.
2.5.0	Minor	Improved JSON serialization performance by 30%. Backward-compatible.
3.0.0	Major	Rewritten core APIs; significantly faster, but deprecated several classes and changed behavior for null values.
3.1.0	Minor	Adds new streaming APIs and improves documentation. Still same breaking changes as 3.0.0.

Compatibility

API

- Names of public functions and data types
- Recompilation should succeed
  - May be required to incorporate updates

ABI (Application Binary Interface)

- Function calling conventions
- Data structure layout
- Instructions, inlined system functions
- Dependent code does not need to be recompiled to incorporate updates

Compatibility

Backward compatibility

- Code that worked with an older version of a dependency will work with a newer version
  - Preserved across MINOR versions

- Implies that public types and functions cannot be removed
- For ABI compatibility, public data structures cannot change outside of "reserved" fields

Forward compatibility

- Code built with a newer version of a dependency will also work with an older version
  - Preserved across PATCH versions

- Implies that no new public types, fields, or functions may be added

"Live at HEAD"

- Dependency management analogue of trunk-based development
- Principles:
  - Always depend on current stable version of everything
  - Never change anything in a way that is difficult for dependents to adapt
- Dependency maintainer responsible for not breaking all users
  - Effectively requires continuous integration for all software in the world (except closed-source dependents)
  - If compatibility cannot be maintained, maintainer will provide an upgrade tool
- **API providers**: Ensure smooth migration; **API consumers**: Provide tests

"Live at HEAD"

- Some of this infrastructure already exists
  - "Rolling" Linux distributions (e.g., Gentoo) integrate tens of thousands of packages continuously
  - Programming languages (e.g. Scala, Rust) proactively test all changes against major libraries/applications
- **Version selection**: What is the latest stable version of everything?

Dependency vulnerabilities

- NPM has a history of dependency-related disasters
  - **left-pad** unpublished
  - Bitcoin theft transitive dependency in **event-stream**
  - Ukraine war **"protestware"** in **node-ipc**
- Why was impact so large?
  - Tools depended on external repository services rather than internal mirror
  - Projects depended on **floating** instead of fixed versions
  - Projects were built "too continuously"
  - Fine-grained dependencies depended upon by many other libraries (cascading)

https://en.wikipedia.org/wiki/Npm_left-pad_incident

https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/hacker-infects-node-js-package-to-steal-from-bitcoin-wallets

https://orca.security/resources/blog/cve-2022-23812-protestware-malicious-code-node-ipc-npm-package

Vulnerabilities

- CVE: Common Vulnerabilities and Exposures
  - Common identifier for specific vulnerabilities (not vulnerable systems)
  - CWE: Common Weakness Enumeration (type of vulnerability)
  - May be crosslinked with other databases (e.g., severity, product, weakness category)
    - NIST's National Vulnerability Database (NVD) includes common links and history
    - Common Vulnerability Scoring System (CVSS) standardizes measures of severity
- Example CVE: https://nvd.nist.gov/vuln/detail/CVE-2025-32955
- Others: https://mvnrepository.com/artifact/org.opencms/opencms-core/16.0

Reading

- *Software Engineering at Google*, Chapter 21: Dependency Management

Build systems

Build System Objectives

- Automate compilation & linkage of all components
- Rebuild necessary components when things change
- Manage multiple configurations
- Manage external dependencies
- Automate testing
- Automate release actions
  - Strip debugging symbols
  - Minify web assets
  - Generate installers

Also relevant for interpreted languages

Desirable properties of build system

- Fast:
  - Run a single command to build and get the output binary in a short time (few seconds)
- Correct:
  - Reproducible: Should output same result for any developer/machine for the same input files

What does a build system even do?

- Why something like `javac *.java` is not enough?
- How to handle:
  - Building libraries stored in different directories (shared libraries)
  - Code written in different programming languages (dependencies)
  - Third-party jar files (how to store them, version management)
  - Rebuilding part of the codebase after dependency upgrade
  - Target different systems/release builds (build configs)
  - (Implicit dependencies) Managing related artifacts/tasks: documentation, latest library version
- Write a shell script?

Options

- Write your own scripts
  - Lots of redundant effort to provide flexibility and functionality
  - Maintenance cost of bespoke system
- Follow conventions
  - Easy way for new projects to take advantage of build tool features with minimal effort
  - Good IDE support
  - Hard to adapt for large, heterogeneous, legacy projects
  - Difficult to diagnose implicit rules
  - Can lead to bloated dependencies
- Configure a build tool
  - Must learn a complicated tool & configuration syntax
    - But knowledge is transferrable
  - Must maintain build configuration
    - But being explicit is often good, avoids dependency bloat
  - Can accommodate custom procedures
    - Code generation
    - Multiple languages
  - IDE may require additional configuration

Common build tools

- Make [1976]
- Autoconf
- CMake
- Ant + Ivy, Maven, Gradle (Java)
- sbt (Java, Scala)
- Pip, setuptools (Python)
- npm, Bower (Javascript)
- Cargo (Rust)
- latexmk (LaTeX)
- Bazel

- Responsible for constructing dependency graph
  - Task-oriented: Targets can execute arbitrary commands
    - Hard to correctly specify when a task does not need to be rerun
    - Hard to parallelize safely
  - Artifact-oriented: Targets must declare inputs, outputs
    - Enables safe caching, parallelization

Task-Based Build Systems

- Task: Fundamental Unit of Work
- Tasks can have other tasks as dependencies
- Major build systems: Ant, Maven, Gradle, Grunt, Rake, …

Make example

- Built-in implicit rules
  - Knows how to compile .cc files to get .o file
  - Uses standard env vars (CXX, CXXFLAGS)
- Compiler provides header dependencies for future use
  - But what if a header with the same name is created elsewhere?
- Does not depend on variable values (static)

- Use .PHONY to declare tasks that don't produce artifacts
- First target is default

- Uses timestamp to detect changes

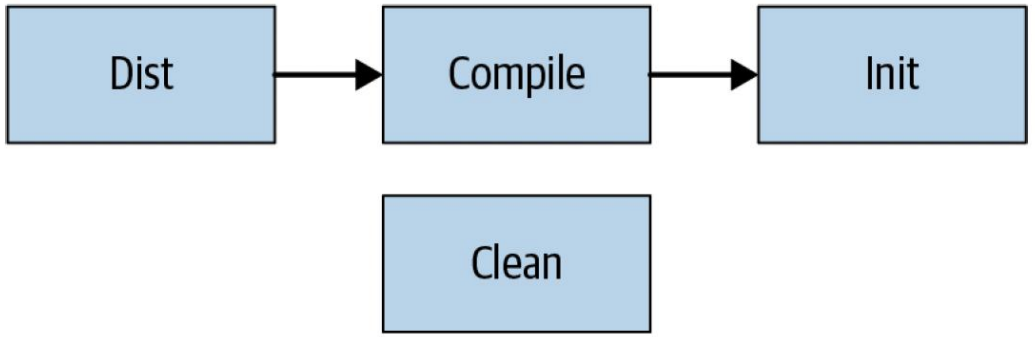
See scrambler/c++/Makefile

Example: Ant Build File

```
<project name="MyProject" default="dist"
  basedir=".">
<description>
simple example build file
</description>
<!-- set global properties for this build -->
<property name="src" location="src"/>
<property name="build" location="build"/>
<property name="dist" location="dist"/>
<target name="init">
<!-- Create the build directory structure used
  by compile -->
<mkdir dir="${build}"/>
</target>
<target name="compile" depends="init"
description="compile the source">
<!-- Compile the Java code from ${src} into
  ${build} -->
<javac srcdir="${src}" destdir="${build}"/>
</target>
<target name="dist" depends="compile"
description="generate the
  distribution">
<!-- Create the distribution directory
  -->
<mkdir dir="${dist}/lib"/>
<!-- Put everything in ${build} into
  the MyProject-${DSTAMP}.jar file -->
<jar jarfile="${dist}/lib/MyProject-
  ${DSTAMP}.jar" basedir="${build}"/>
</target>
<target name="clean"
description="clean up">
<!-- Delete the ${build} and ${dist}
  directory trees -->
<delete dir="${build}"/>
<delete dir="${dist}"/>
</target>
</project>
```

Example: Ant Build File

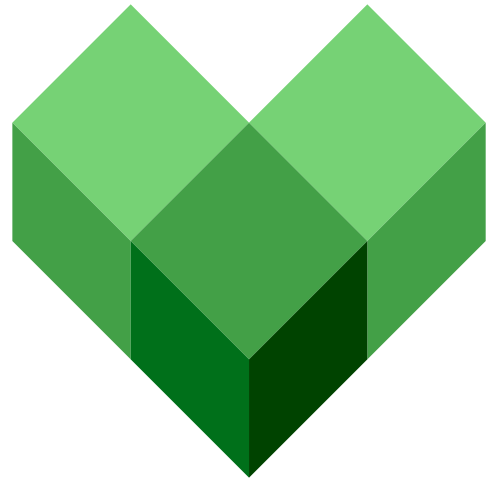
- "`ant [task name]`" executes the given task and its dependent tasks
- Advantage:
  - Modularized builds
- Disadvantages:
  - One more file to debug! (tricky)
  - Hard to parallelize
  - Incremental builds are difficult



Artifact-based build system

- Build: Tell the system “what” to build instead of “how”
- Implemented in Blaze/Bazel (Google), Pants, Buck
- Build files are declarative: specify set of artifacts to build, their dependencies, some build options (instead of exact steps)
- Blaze has full control over “how” build is run

- *(Stronger) correctness guarantee while being more efficient*



Example Bazel BUILD file

```
java_binary(
name = "MyBinary",
srcs = ["MyBinary.java"],
deps = [ ":mylib",],)
java_library(
name = "mylib",
srcs = ["MyLibrary.java", "MyHelper.java"],
visibility =
  ["//java/com/example/myproduct:__subpackage
  s__"],
deps = [
"//java/com/example/common",
"//java/com/example/myproduct/otherlib",
"@com_google_common_guava_guava//jar",
],)
```

- **Targets/Artifacts**: java_binary, java_library
- **Workspace**: Source hierarchy for artifacts

Bazel BUILD Steps

```
bazel build :MyBinary
```

- Parse all build files and create graph of artifacts and dependencies
- Determine transitive dependencies of MyBinary
- Build each dependency (in order).
  - Start with artifacts with no dependencies.
  - Keep track of artifacts that need dependencies to be built
  - Build a target **as soon as** its dependencies are built
- Build final MyBinary executable binary

Bazel Advantages/Differences

- Parallelization:
  - Targets that only require java compiler (vs custom script)
- Reuse/caching:
  - If MyBinary.java changes, it will rebuild **MyBinary** but reuse **mylib**
  - If a source file for **//java/com/example/common** changes, Bazel knows to rebuild that library, **mylib**, and **MyBinary**, but reuse **//java/com/example/myproduct/otherlib**

PollEv.com/cs5150sp25

Which of the following best describes an advantage of Bazel over traditional build systems like Make or custom Java build scripts?

A. Bazel always rebuilds the entire project to ensure consistency.
B. Bazel parallelizes tasks using custom shell scripts instead of native compilers.
C. Bazel tracks fine-grained dependencies, enabling it to rebuild only what's necessary and reuse cached outputs.
D. Bazel relies on environment variables for dependency tracking and compilation.

Other Bazel Features

- Tools as dependencies, **toolchains** (platform-specific tool usage)
- Custom user-defined **actions:** specify inputs, outputs, and steps
- **Sandboxing**: isolating filesystem for each action
- Remote caching
- **Distributed build**: Remote build
- Making remote/external dependencies deterministic
  - Manifest file: Create **cryptographic hash** for each ext dependency, only redownload when hash changes, build fails if hash changes
  - What can go wrong?

Read more details in SWE@Google Chapter 18 build systems

Dependency Management @ Google
Scaling to Billions of Lines of Code

- Strict Transitive Dependency:
  - A cannot use a symbol for C without declaring direct dependency
- External Dependencies: Uses semantic versioning
  - One-Version Rule (eliminates diamond dependency problem)
- Transitive External Dependencies:
  - Bazel does not allow automatic download of such dependencies
- Shared cache for external artifacts that require building
- Security/Reliability: Mirroring servers, Vendoring