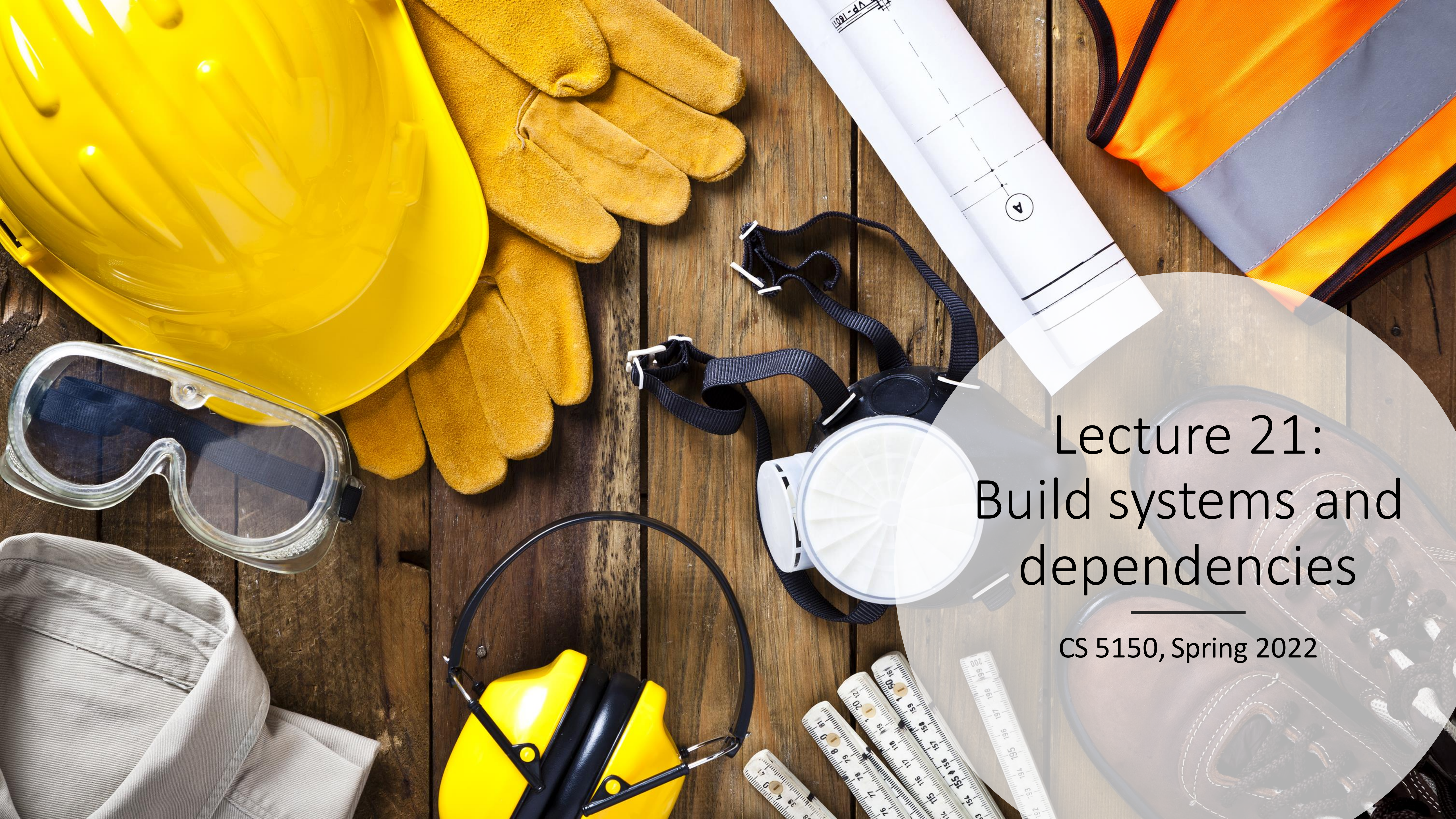

Lecture 21: Build systems and dependencies

CS 5150, Spring 2022

Administrative announcements

- Report #4 due tomorrow (Friday)
- In-class exam next Thursday (Apr 21)
  - Sample questions available on Canvas
  - E-mail instructor if you have an approved conflict outside of your control (hospitalization/isolation, religious observance)

Lecture goals

- Evaluate application performance
- Automate compilation using build systems
- Manage application dependencies and associated risks

Performance

… continued from Lecture 20

Amdahl's Law

- Speedup: $S = T\_before / T\_after$
- Identify portion p of runtime cost amenable to optimization
  - $T\_before = p*T + (1 - p)*T$
- Let s be speedup of optimization on this portion
  - Example: $s = 10$ for parallelizing on a 10-core machine
  - Often interested in limit as $s \to \infty$
- $T\_after = p*T/s + (1 - p)*T$
- $S(s) = 1/(1 - p + p/s)$
- **$S \to 1/(1 - p)$**

Profiling

- How can we estimate *p*?
- Where should our optimization efforts be focused?

- Profiling techniques
  - Sampling: Periodically interrupt process and examine stack trace
    - Low overhead
    - Incomplete data
  - Tracing: Record whenever a function is called or returns
    - High overhead
    - Complete function counts
    - Timing may be distorted
  - Instruction-level: Estimate cost of each statement
    - Requires CPU model

Profiling tools

- Native code
  - perf (Linux)
  - gprof
  - callgrind
- Python
  - cProfile
- Java
  - JProfiler
  - VisualVM
  - NetBeans
- Visualizers:
  - kcachegrind
  - Flame graphs
  - Web browser profilers

Monitoring

- To detect degradation and catch regressions, need to log and monitor performance metrics
  - Can measure duration of tests in CI, but benefits from unloaded servers
- For services, also need to monitor performance in production
  - Network conditions, load are dynamic
  - With scalable microservice architectures, counterintuitive bottlenecks may appear
    - Scaling the wrong components can remove beneficial backpressure

Soak testing

- Tests often execute for less time than a production system
  - Many production systems never turn off (e.g. embedded controllers)
  - Some defects (e.g. memory leaks, fragmentation) are innocuous for short runs
- Soak testing: Subject system to significant load for extended period of time (days, months, years)
  - Be sure to log key performance metrics (cycle time, memory usage)
  - Not particularly compatible with a rapid CI pipeline
    - Still good to run periodically to catch issues sooner

Build systems

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

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

Make example

- Built-in implicit rules
  - Knows how to compile .cc files to get .o file
  - Uses standard env vars
- 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
- Use .PHONY to declare tasks that don't produce artifacts
- First target is default

See scrambler/c++/Makefile

State-of-the-art (Bazel)

- Sandboxing to enforce artifact dependencies
- Distributed compilation, caching
- Test dependencies & caching
- Dependencies include env vars, toolchains
- Conservative header dependency extraction
- Extensible for custom languages, tools

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

Reading

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