Lecture 18: Building Software, Build Automation, and Databases

1. The Painstaking Process of Building Software

- Building and delivering software is a painstaking, error-prone process.
- Historically, all steps were done by hand: writing code, compiling, linking, running.
- In early CS courses (CS31/32), students manage these steps manually.
- **Theme:** Whenever a task is boring or repetitive, automate it! If you don't, someone else will.

---

2. Automating the Build Process

2.1 Naive Automation with Scripts

- Developers write shell or Python scripts to automate builds.
- Example shell script:

```
#!/bin/sh
gcc -c foo1.c
gcc -c foo2.c
gcc -c foo3.c
echo $1 > foo4.c
gcc -c foo4.c
gcc foo1.o foo2.o foo3.o foo4.o -o myprogram
```

- **Problems:**
  - Inefficient: recompiles everything, even if only one file changed.
  - Not scalable for large projects.
  - No partial builds; always does full rebuild.

2.2 The Need for Smarter Automation

- Ideal: Only rebuild what's necessary (incremental builds).
- Desire for a tool that can "partially execute" a build script, skipping unnecessary steps.

Example: Incremental Build with `make`

Suppose you change only `foo2.c`:

1. `make` checks timestamps:
   - `foo2.c` is newer than `foo2.o` → rebuild `foo2.o`.
   - `foo1.o`, `foo3.o`, `foo4.o` are up-to-date → not rebuilt.
   - `myprogram` is now older than `foo2.o` → relink all `.o` files.
2. Only the necessary steps are performed, saving time.

Step-by-step:

```
$ touch foo2.c           # Simulate editing foo2.c
$ make
  gcc -c foo2.c         # Only foo2.o is rebuilt
  gcc foo1.o foo2.o foo3.o foo4.o -o myprogram  # Relink
```

---

3. `make`: Smarter Build Automation

3.1 What is `make`?

- A tool that automates builds, only rebuilding what's needed.
- Input: a `Makefile` describing targets, dependencies, and build commands.
- Uses file timestamps to determine what needs rebuilding.

3.2 Makefile Structure

- Rules:

```
target: dependencies
    command-to-build-target
```

- Example:

```
myprogram: foo1.o foo2.o foo3.o foo4.o
    gcc foo1.o foo2.o foo3.o foo4.o -o myprogram

foo1.o: foo1.c
    gcc -c foo1.c

foo2.o: foo2.c bar.h
    gcc -c foo2.c

foo3.o: foo3.c
    gcc -c foo3.c

foo4.c:
    echo $1 > foo4.c

foo4.o: foo4.c
    gcc -c foo4.c
```

3.3 How `make` Works

- Compares timestamps: if a target is older than any dependency, it rebuilds.
- Skips steps if target is up-to-date.
- **Dependency Graph:** Internally, `make` builds a graph of targets and dependencies.
- Can detect cycles (e.g., A depends on B, B on A) and will error out.

Example: Make Dependency Graph (ASCII Diagram)

```
   foo1.c   foo2.c   foo3.c   foo4.c
     |        |        |        |
     v        v        v        v
   foo1.o   foo2.o   foo3.o   foo4.o
     \        |        /        /
       \      |      /        /
          v   v   v   v
          myprogram
```

3.4 Common Pitfalls

- **Forgetting dependencies:** If a file includes a header (e.g., `bar.h`) but the Makefile omits it, changes to the header won't trigger rebuilds.
- **Over-specifying dependencies:** Adds unnecessary rebuilds but ensures correctness.
- **Solution:** Use compilers (GCC/Clang) to auto-generate dependencies (e.g., `-MMD` flag).

Example: Missing Dependency

```
foo2.o: foo2.c
    gcc -c foo2.c
```

If `foo2.c` includes `bar.h` and `bar.h` changes, `foo2.o` will NOT be rebuilt! Fix:

```
foo2.o: foo2.c bar.h
    gcc -c foo2.c
```

Example: Dependency Cycle (Error)

```
A: B
B: A
```

This creates a cycle. `make` will report an error:

```
make: Circular A <- B dependency dropped.
```

Table: Makefile Pitfalls and Solutions

Pitfall	Consequence	Solution
Missing dependency	Outdated builds, bugs	List all headers as dependencies
Over-specifying	Unnecessary rebuilds	Use auto-generated dependencies
Cyclic dependencies	Build fails	Refactor dependency graph

3.5 Clock Skew

- Distributed builds can suffer from inconsistent clocks (clock skew).
- Can cause unnecessary or missed builds.
- **Alternative:** Use file checksums instead of timestamps (more management overhead).

3.6 Parallel Builds

- `make -j` runs independent build steps in parallel.
- Great for large projects and multi-core machines (e.g., Linux kernel builds).

3.7 Scaling Build Systems

- **Recursive make:** Each subdirectory has its own Makefile; top-level Makefile runs `cd subdir && make`.
  - Harder to parallelize.
- **Single top-level Makefile with includes:**
  - Centralized control, better parallelism.
  - Uses `include` directives to pull in sub-makefiles.

3.8 Modern Build Systems

- **CMake:** Generates platform-specific Makefiles or Ninja files; supports complex projects and cross-platform builds.
- **Ninja:** Fast, minimal build system; often used as a backend for CMake.
- **Bazel:** Scalable, hermetic builds; used by Google and others for large codebases.
- **Meson:** Modern, fast, user-friendly; generates Ninja files.

Tool	Strengths	Typical Use Case
Make	Simple, ubiquitous	Small/medium C/C++ projects
CMake	Cross-platform, feature-rich	Large, portable projects
Ninja	Extremely fast, simple syntax	Backend for CMake/Meson
Bazel	Scalable, reproducible, language-agnostic	Monorepos, CI/CD
Meson	Modern, easy to use, fast	Modern C/C++/Python builds

---

4. Portability and System Configuration

4.1 The Problem

- Not all platforms support the same system calls (e.g., `renameat2()` on Linux, not Windows).

4.2 Conditional Compilation

- Use C preprocessor to select implementations:

```
#ifdef __linux__
    // Use renameat2
#else
    // Use fallback logic
#endif
```

- **Problem:** OS-based checks can break if OSes add/remove features.

4.3 Feature Testing (Preferred)

- Instead of checking OS, check if the feature exists.
- Write a test program to probe for the feature during build.
- Generate a config header (e.g., `config.h`):

```
#define HAS_RENAMEAT2 1
```

4.4 Build-Time Configuration Tools

Tool	Purpose
`configure`	Probes system, generates `config.h`
`autoconf`	Generates the `configure` script
`automake`	Generates portable `Makefile`s

- These tools automate and layer the build configuration process.

ASCII Diagram: Autotools Configuration Flow

```
[autoconf] ---> [configure script]
      |               |
      v               v
[configure.ac]     [User runs ./configure]
                        |
                        v
                [config.h, Makefile]
```

---

5. Packaging in Higher-Level Languages

5.1 Python Example

- Uses `pyproject.toml` for metadata and build configuration.

```
[build-system]
requires = ["setuptools >= 77.0"]
build-backend = "setuptools.build_meta"

[project]
name = "myproject"
version = "1.0.0"
authors = [{ name = "Your Name", email = "you@example.com" }]
license = "GPL-3.0"
```

- **Warning:** Faulty configuration files can break installs and compatibility.

Table: Python Packaging Tools

Tool	Purpose
pip	Install Python packages
setuptools	Build and distribute packages
wheel	Binary package format
virtualenv	Isolated Python environments
pyproject.toml	Modern build/config metadata

Example: setup.py

```
from setuptools import setup
setup(
```

```
    name='myproject',
    version='1.0.0',
    author='Your Name',
    packages=['myproject'],
    install_requires=[],
)
```

---

6. Databases: Why Not Just Use File Systems?

6.1 Motivation

- File systems are unstructured, lack consistency guarantees, and are slow for search.
- Databases provide:
  - Structured access
  - Consistency
  - Efficient search
  - Concurrency-safe access

6.2 Relational Databases

- **Data Model:** Tables (relations) with rows and columns.
- **Columns:** Typed (e.g., `INT`, `VARCHAR(255)`).
- **Rows:** Each is a record/entity. Usually unique (set semantics), but some DBs allow duplicates (multiset).
- **Keys:**
  - **Primary key:** Uniquely identifies a row (e.g., student ID).
  - **Foreign key:** References a key in another table; enforces referential integrity.
- **NULL:** Some DBs allow missing values (NULL), leading to three-valued logic (TRUE, FALSE, UNKNOWN).

6.3 Relational Algebra

- **Selection:**

```
SELECT * FROM student WHERE major = 'CS';
-- Returns only rows where major is CS
```

- **Projection:**

```
SELECT DISTINCT major FROM student;
-- Returns unique majors only
```

- **Join:**

```
SELECT student.id, student.major, class.class
FROM student JOIN class ON student.id = class.student_id;
```

Expanded Examples

- **Selection:**

```
SELECT * FROM student WHERE major = 'CS';
-- Returns only rows where major is CS
```

- **Projection:**

```
SELECT DISTINCT major FROM student;
-- Returns unique majors only
```

- **Join:**

```
SELECT student.id, student.major, class.class
FROM student JOIN class ON student.id = class.student_id;
```

Operation	SQL Example	Result Description
Selection	SELECT * FROM student WHERE major = 'CS';	Filters rows by predicate
Projection	SELECT DISTINCT major FROM student;	Selects columns, removes dups
Join	SELECT ... FROM A JOIN B ON A.x = B.y;	Combines rows on matching columns
Union	SELECT ... FROM A UNION SELECT ... FROM B;	Combines rows from both tables
Intersection	SELECT ... FROM A INTERSECT SELECT ... FROM B;	Rows common to both tables
Cartesian	SELECT * FROM A, B;	All combinations of rows

6.4 SQL: Structured Query Language

- **Schema definition:**

```
CREATE TABLE student (
    id INT PRIMARY KEY,
    family_name VARCHAR(255),
    given_names VARCHAR(255),
    major VARCHAR(64)
);

CREATE TABLE class (
    student_id INT,
    letter_grade VARCHAR(2),
    FOREIGN KEY (student_id) REFERENCES student(id)
);
```

- **Insert:**

```
INSERT INTO student VALUES (1, 'Abdul-Jabbar', 'Kareem', 'Sociology');
INSERT INTO student VALUES (2, 'Coppola', 'Francis', 'Film Studies');
```

- **Query:**

```
SELECT id, major FROM student ORDER BY major ASC;
```

---

7. Alternative Database Models

7.1 Entity-Relationship (ER) Model

- Focuses on entities (objects), attributes (properties), and relationships (references).
- Can be implemented atop relational DBs or from scratch.

7.2 Object-Oriented Databases

- Data as objects with attributes, methods, and class hierarchies.
- Integrates code (methods) with data.

7.3 NoSQL

- Rejects rigid table format; allows flexible, dynamic schemas.
- **Key-value stores:** Like hash tables, but persistent.
- **Wide-column:** Columns can differ between rows.
- **Document stores:** Store arbitrary documents (JSON, XML, etc.).
- **Graph databases:** Entities as nodes, relationships as edges; supports cycles and arbitrary graphs.
- **Trade-off:** More flexibility, but may lack strong guarantees and efficient search.

Table: NoSQL Database Types

Type	Example Systems	Data Model	Strengths	Weaknesses
Key-Value	Redis, DynamoDB	Hash table	Fast, simple	No structure, limited query
Wide-Column	Cassandra, HBase	Table, flexible	Scalable, flexible	Complex, eventual consistency
Document	MongoDB, CouchDB	JSON/BSON/XML	Flexible, nested data	Weaker consistency, joins hard
Graph	Neo4j, ArangoDB	Nodes/edges	Rich relationships	Niche, complex queries

Example: Document Store (MongoDB)

```
{
  "_id": 1,
  "name": "Alice",
  "courses": ["CS101", "CS102"]
}
```

---

8. Database Reliability: ACID Properties

Property	Description
Atomicity	Transactions are all-or-nothing
Consistency	DB always satisfies constraints/invariants
Isolation	Concurrent operations don't interfere (as if run sequentially)
Durability	Changes persist across failures

- **Example:**

  ```
  BEGIN TRANSACTION;
  INSERT INTO student VALUES (...);
  UPDATE classes SET grade='A' WHERE ...;
  COMMIT;
  ```

- Without atomicity, partial changes may persist if system crashes.
- Without isolation, concurrent users may see inconsistent data.

Example: Transaction Failure (Atomicity Violation)

Suppose a bank transfer:

```
BEGIN TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
-- System crashes here!
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT;
```

If atomicity is not enforced, $100 is lost! With atomicity, neither change is visible after crash.

ASCII Diagram: Transaction Timeline

```
[Start Transaction]
      |
      v
[Step 1: Deduct $100 from A]
      |
      v
[System Crash]
      |
      v
[Recovery: Rollback]
      |
      v
[No changes applied]
```

Example: Isolation Violation

If two users transfer from the same account at the same time, without isolation, both may see the same initial balance and overdraw the account. With isolation, transactions appear to run one after another.

8.1 Compromising ACID

- Some cloud/NoSQL DBs sacrifice consistency or isolation for scalability (eventual consistency).
- Developers must balance speed, complexity, and correctness.

8.2 Practical Tools

- **Firebase:** Popular NoSQL backend; not fully ACID-compliant, emphasizes client sync.
- **Oracle, MySQL, PostgreSQL:** Relational, support full ACID.

---

9. Summary

- Software build automation evolved from manual steps to sophisticated tools (`make`, `autoconf`, `automake`).
- Build systems must handle partial builds, parallelism, and portability.
- Databases provide structured, reliable, and efficient data management.
- Relational model, SQL, and ACID properties are foundational.
- Alternative models (ER, object, NoSQL, graph, document) offer flexibility for modern needs.
- Understanding trade-offs is key to building robust, scalable systems.