“Thread Safety”
and
Parallel Program Correctness

Thread Safety?

- Wikipedia:

  “A piece of code is thread-safe if it can be used in a multi-threaded environment.”

  huh?

From:

www.cs.uiuc.edu/class/fa09/cs241/fa09/12-Threads_sol.ppt

- “Thread-safe means that a function can be called from multiple threads without destructive results.”

Java Concurrency In Practice
Brian Goetz, et al.

"A class is thread safe if it **behaves correctly** when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code. "

“behaves correctly”?

- How is this defined?

Java Concurrency In Practice
Brian Goetz, et al.

"Since any single threaded program is also a valid multithreaded program, it cannot be thread safe if it is not even correct in a single threaded environment. **If an object is correctly implemented, no sequence of operations calls to public methods and reads or writes of public fields should be able to violate any of its invariants or post conditions**. No set of operations performed sequentially or concurrently on instances of a thread safe class can cause an instance to be in an invalid state. "

Was that a definition of correctness?

- Notice it said “if”, not “if and only if”.

Sequential Program Correctness

- Partial Correctness
- Total Correctness

Parallel Program Correctness

- Depends on the intent:
  - Compute a specific result
  - Provide a service
  - Coordinate other programs or methods

Adapting Sequential Programs

- Assume sequential program is (or has been proved) correct.
- Show that nothing is done in the parallel realm to change this.
- Goetz, et al: No “Invalid state”

What is State?

Modeling

- States
- Transitions

State Components

Correctness Properties
“Safety Properties”

- State invariant
- Holds initially
- Preserved by all transitions
- (“No invalid states”)

Correctness Properties
“Liveness Properties”

- If a safety property is “nothing bad can happen”,

- a liveness property is “something good will happen”.

Ways to Verify Properties

- Testing
- Informal proof
- Formal verification
- Static analysis
- Dynamic analysis
- Model-checking

Testing

- Dijkstra:

  “Testing can prove the presence of errors, but not the absence”.

- Repeatability of error syndromes is harder with concurrency.

- Errors may go undiscovered for longer due to effective non-determinism in control.

Formal Verification

- Requires a formal model for states and transitions, as well as
  - proper characterization of atomicity
  - axioms for data abstractions (arrays, lists, etc.)
- How to validate the model itself?
- How to prove within the model?

Informal Proofs

- Informal proofs do not alleviate all the needs of formal proofs.

- They do increase chances for oversight.

Static Analysis

- Sufficiency conditions that can be checked mechanically, usually by examining source code.

Dynamic Analysis

- Analysis relying partly on execution to exercise states.
- The Cilk determinacy checker is an example.

Model Checking

- Applicable to finite-state systems or finite-state approximations
- Still require modeling of states and transitions
- Don’t require axiomatization of data operations (“just do it” approach)

Modeling in General

- Modeling is helpful for conceptualization,
- even if no formal implementation is planned.

Formal Models

- What are the states?
- What are the transitions?
- What are the properties?

States

- Values of all program variables and locations:
  function: Identifiers → Values

  where “Identifiers” includes

  names,

  memory locations

  instruction pointer (“program counter”)

  positions

Transition

- A mapping

  enabled: States → {True, False}

  indicating whether the transition is enabled.

- A mapping

  next: States → States

  indicating what the next state is, **if** the transition is enabled.

Why “next” is a function

- It is best if next is a function and not merely a relation (which would represent non-determinism).

- However, the theory could be extended to relations as well.

“Firing” a Transition

- When enabled(s) and the state changes to next(s), we say the transition “fires”.

Proving Invariants

- Invariants are easier to assert than to prove.

- The problem is that there are often “missing ingredients” in the invariant that are required for completing a proof.

Proof by Transition Induction

- Suppose that J is a property of states, a desired invariant.

- Prove invariance by:
  - J is true for the initial state, and
  - For an arbitrary state s:

$$(J(s) \wedge \text{enabled}(s)) \rightarrow J(\text{next}(s))$$

Problem using Transition Induction

- A state predicate J might be invariant, yet insufficient to prove

  $\forall s\ [(J(s) \wedge \text{enabled}(s)) \rightarrow J(\text{next}(s))]$

Two reasons:

$\forall s$ is over-kill, only need *reachable* states

J doesn't carry enough information to prove J(next(s))

Inductive vs. Invariant Predicates

- A predicate K is **inductive** if it satisfies the induction principle:

$$K(\text{initial}) \\ \wedge\ \forall s\ [(K(s) \wedge \text{enabled}(s)) \rightarrow K(\text{next}(s))]$$

Then k is true for all states **reachable from** initial.

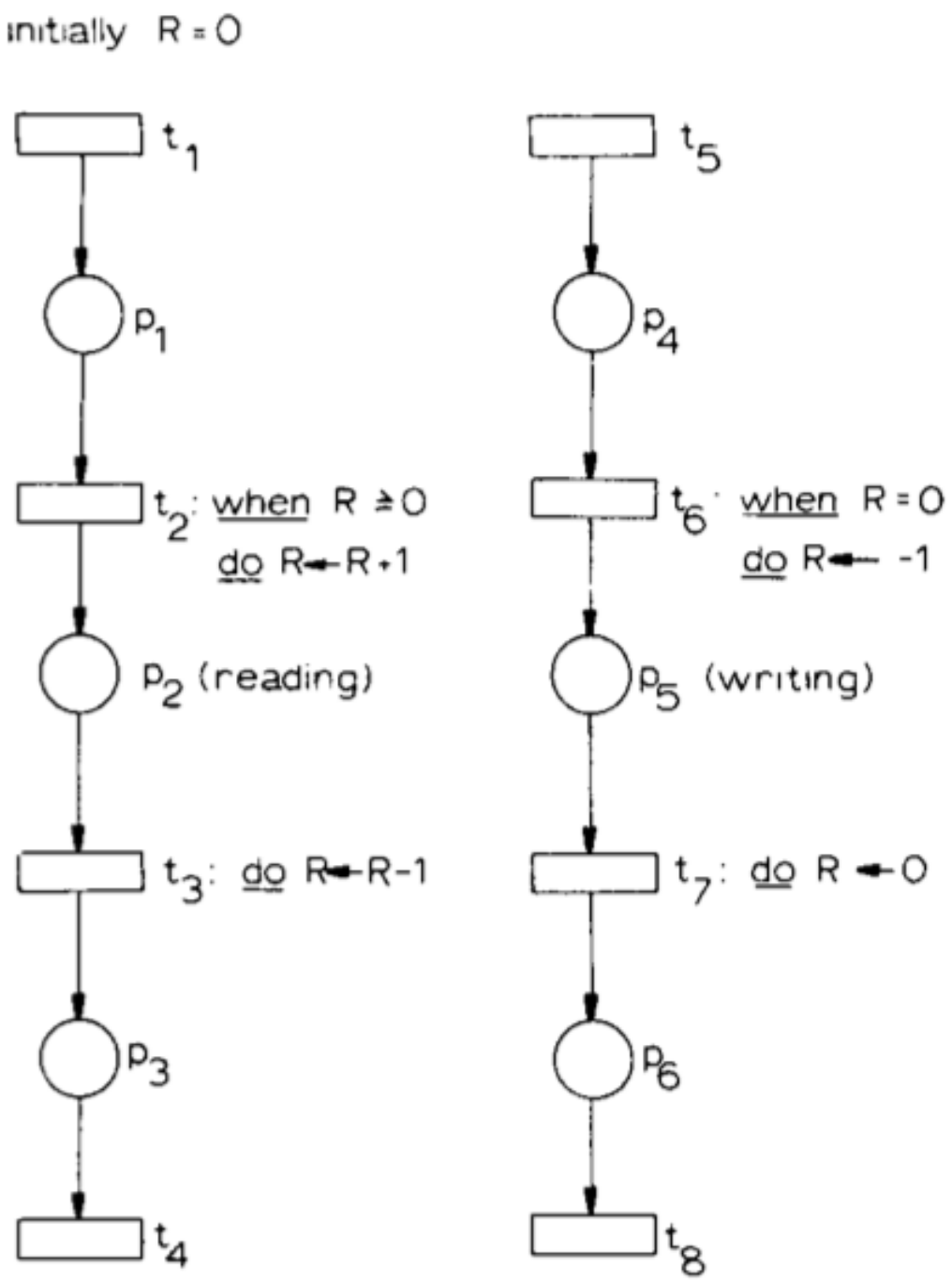
To Show Invariance

- A **sufficient** condition for J to be invariant:

  There is an inductive predicate K such that $K \rightarrow J$.

Finding the right K is often not trivial. It could be too weak or strong.

Example: Readers & Writers Synchro-nization



Invariant

Deadlock

- Is absence of deadlock a safety property or a liveness property?

Liveness Properties

- In order to guarantee something will happen, we first need some way to enforce progress of individual transitions, such as:
  - A transition cannot remain enable infinitely long without firing, or
  - A transition cannot be enabled infinitely-many times without firing.

Temporal Logic

- The need to address liveness prompts the use of temporal logic.

- Temporal logic adds to predicate logic temporal operators that deal with time or sequences of states.

Invariance

- [ ] (box, “always”, “henceforth”) represents invariance:

  [ ] P

  means P is true from the current state onward.

Eventuality

- <> (diamond, “eventually”) represents eventuality:

  <> P

  means P will eventually become true (although it might not be true in the current state).

Temporal Expressions

- [ ] <> P means P will always become true. (If it becomes true then false, it will become true again.)

- <> [ ] P means P will become true and stay true thereafter.

Equivalences

- [ ] [ ] P ≡ [ ] P
- <> <> P ≡

Equivalences

- <> ¬P ≡
- [ ] ¬P ≡

  provided the law of the excluded middle

- [ ] (P v ¬P)

Leads-To

- P --> Q means “whenever P is true, Q will eventually become true”