Programm- & Systemverifikation

Concurrency

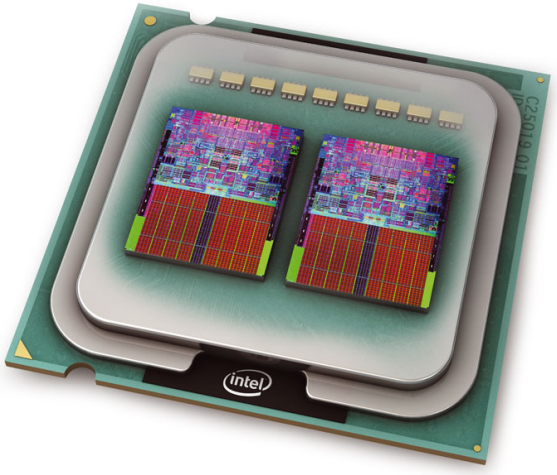
Georg Weissenbacher

184.741

What happened so far

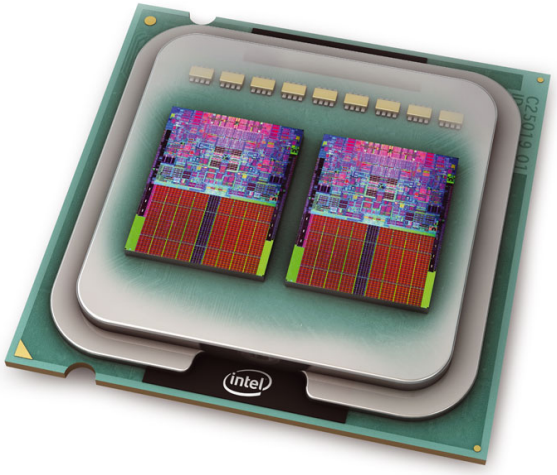
- How bugs come into being:
  - Fault – cause of an error (e.g., mistake in coding)
  - Error – *incorrect* state that may lead to failure
  - Failure – deviation from *desired* behaviour
- We specified *intended* behaviour using assertions
- We proved our programs correct (inductive invariants).
- We learned how to test programs.
- We heard about logical formalisms:
  - Propositional Logic
  - First Order Logic
  - Temporal Logic
- . . . and tools to reason in/about these logics.
- Hoare's Calculus for reasoning about programs

Concurrency



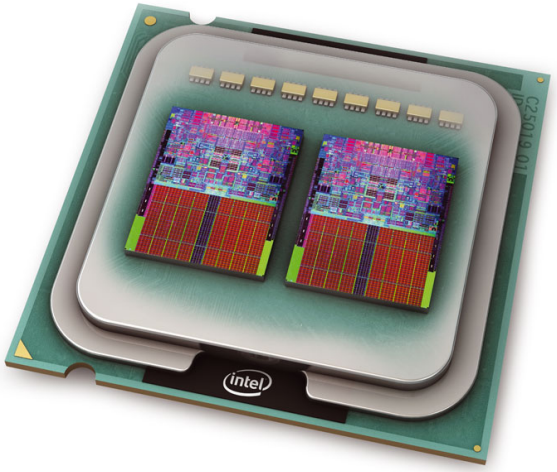
- Upper limit for processor frequency has been reached
- Chip manufacturers now increase number of cores instead

Concurrency



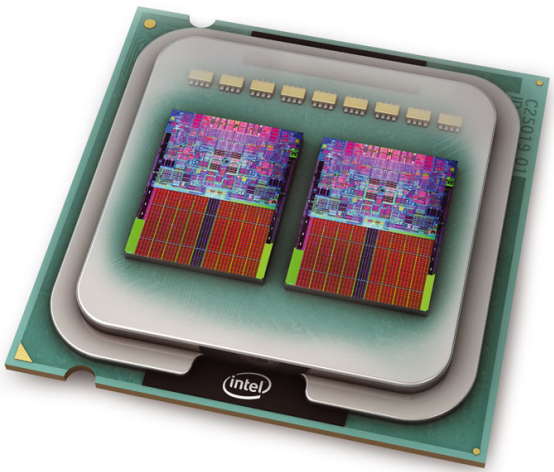
- Upper limit for processor frequency has been reached
- Chip manufacturers now increase number of cores instead
- Performance improvements depend on multi-threaded programming

Concurrency



- Upper limit for processor frequency has been reached
- Chip manufacturers now increase number of cores instead
- Performance improvements depend on multi-threaded programming
- Opens Pandora's Box of new bugs (e.g., Heisenbugs)

What kinds of *concurrency* bugs are there?



Concurrency Bugs

- deadlock

  (two tasks wait for same resource)

- livelock/starvation

  (thread makes no progress)

- race condition

  (two threads accessing resource at same time)

- order violation

  (statements executed in unintended order)

- atomicity violation

  (interruption of supposedly atomic action)

Synchronization Primitives

- *Locks* can be used to prevent simultaneous or concurrent access to critical regions or resources
- Simplified API:
  - `lock(A)` succeeds if lock `A` is available
  - `lock(A)` blocks if lock is already held/acquired (by this or another thread)
  - `unlock(A)` releases a lock previously acquired
  - `unlock(A)` never blocks

Deadlocks

- Deadlocks can happen if locks are acquired in *wrong order*

Thread 1

```
lock (A);
  lock (B);
  unlock (B);
unlock (A);
```

Thread 2

```
lock (B);
  lock (A);
  unlock (A);
unlock (B);
```

Deadlocks

- Deadlocks can happen if locks are acquired in *wrong order*
  - Thread one acquires lock A

Thread 1

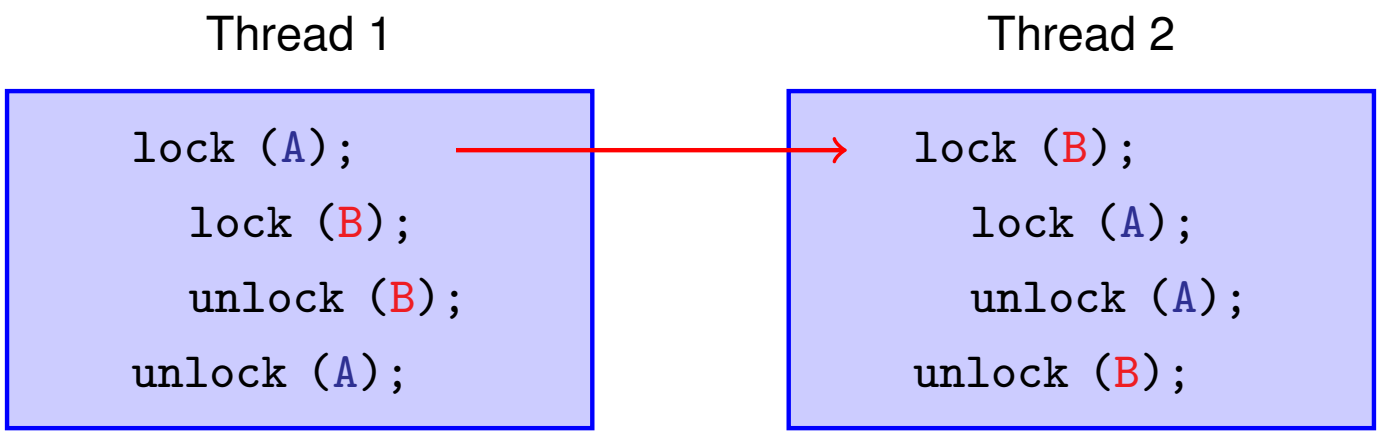
```
lock (A);
  lock (B);
  unlock (B);
unlock (A);
```

Thread 2

```
lock (B);
  lock (A);
  unlock (A);
unlock (B);
```

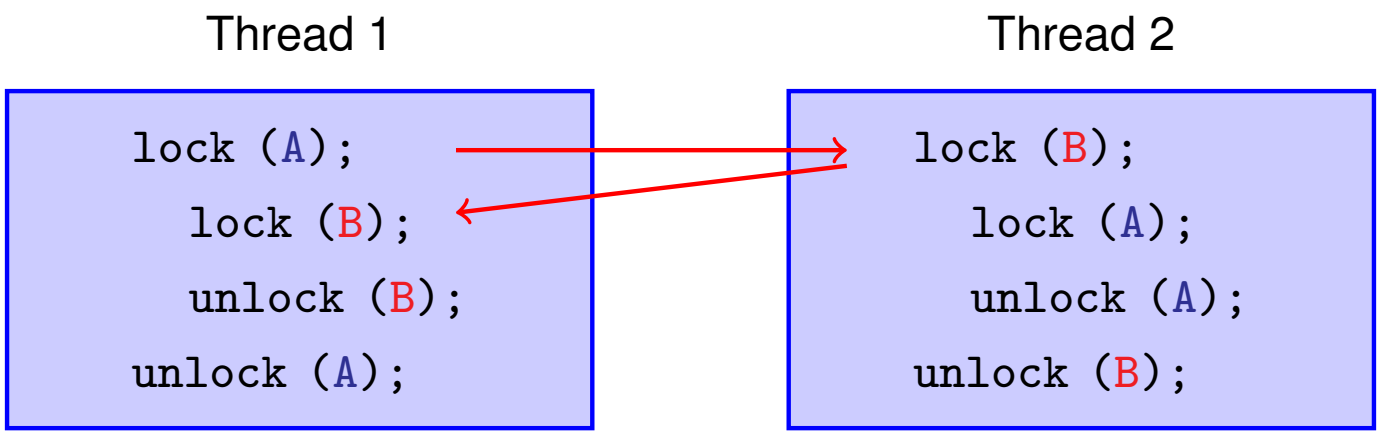
Deadlocks

- Deadlocks can happen if locks are acquired in *wrong order*
  - Thread one acquires lock A
  - Thread two acquires lock B



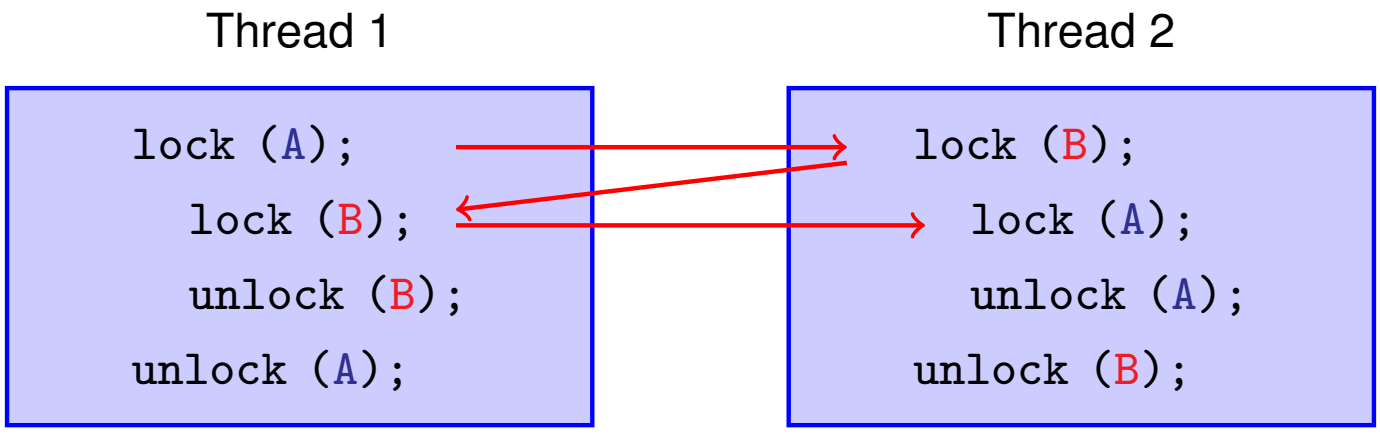
Deadlocks

- Deadlocks can happen if locks are acquired in *wrong order*
  - Thread one acquires lock A
  - Thread two acquires lock B
  - Thread one waits for lock B (thread two still running)



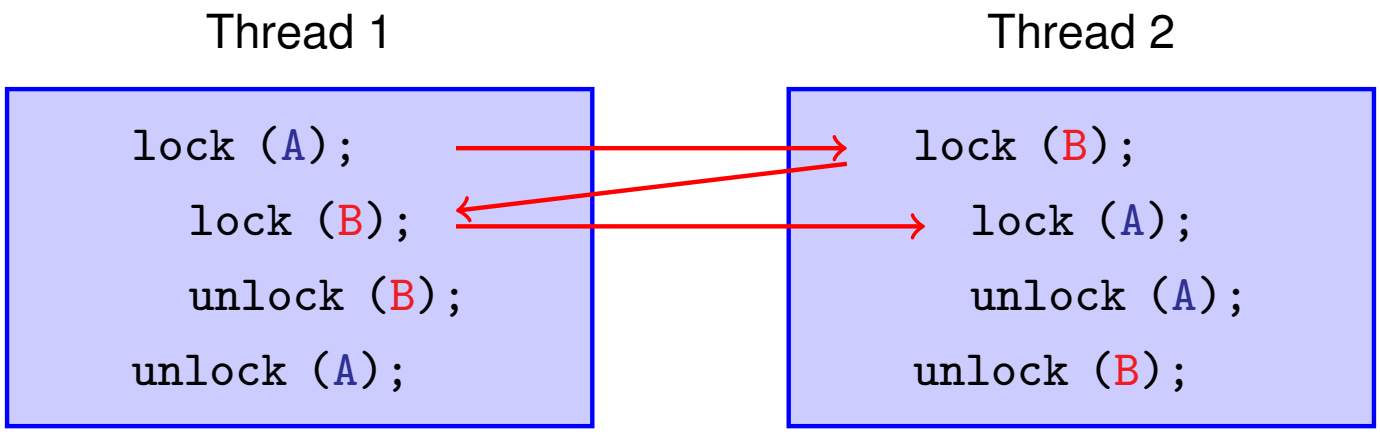
Deadlocks

- Deadlocks can happen if locks are acquired in *wrong order*
  - Thread one acquires lock A
  - Thread two acquires lock B
  - Thread one waits for lock B
  - Thread two waits for lock A



Deadlocks

- Deadlocks can happen if locks are acquired in *wrong order*
  - Thread one acquires lock A
  - Thread two acquires lock B
  - Thread one waits for lock B
  - Thread two waits for lock A
  - Now both threads are stuck...



Attempt to Fix Deadlock

- Assume `lock` returns true on success, false otherwise

```
1: lock (A);
     if (!lock (B))
       goto 2;
     do some work
     unlock (B);
2: unlock (A);
   goto 1;
```

```
3: lock (B);
     if (!lock (A))
       goto 4;
     do some work
     unlock (A);
4: unlock (B);
   goto 3;
```

Attempt to Fix Deadlock

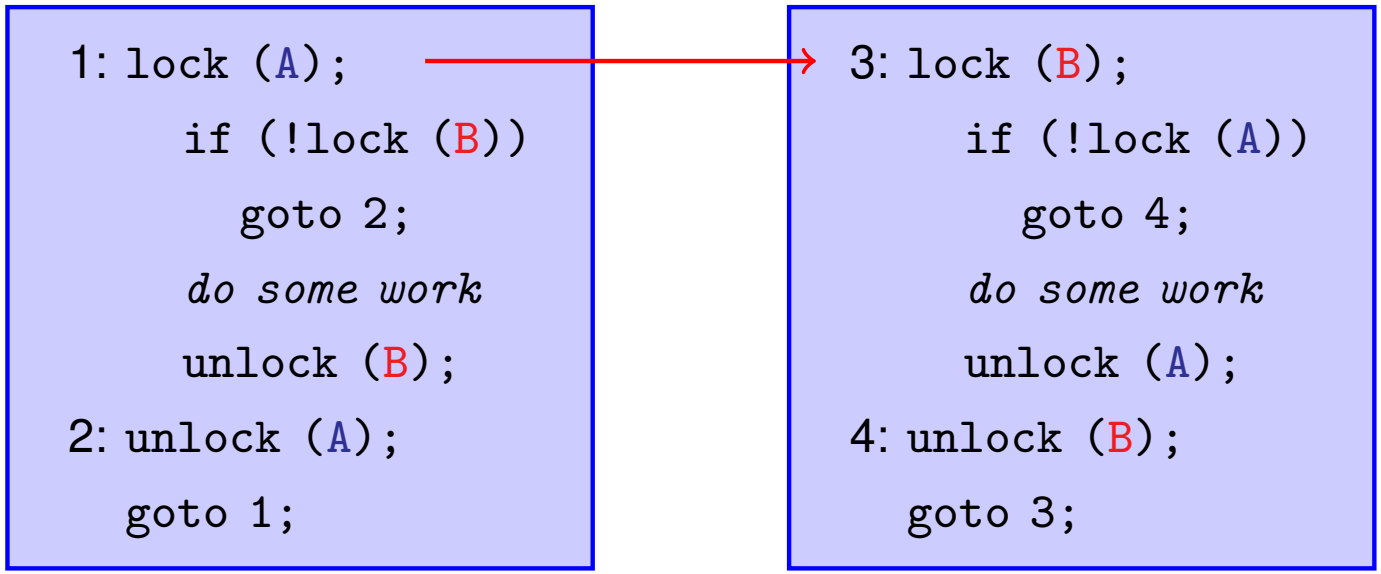
- Assume `lock` returns true on success, false otherwise
  - Thread one acquires lock A

```
1: lock (A);
     if (!lock (B))
       goto 2;
     do some work
     unlock (B);
2: unlock (A);
   goto 1;
```

```
3: lock (B);
     if (!lock (A))
       goto 4;
     do some work
     unlock (A);
4: unlock (B);
   goto 3;
```

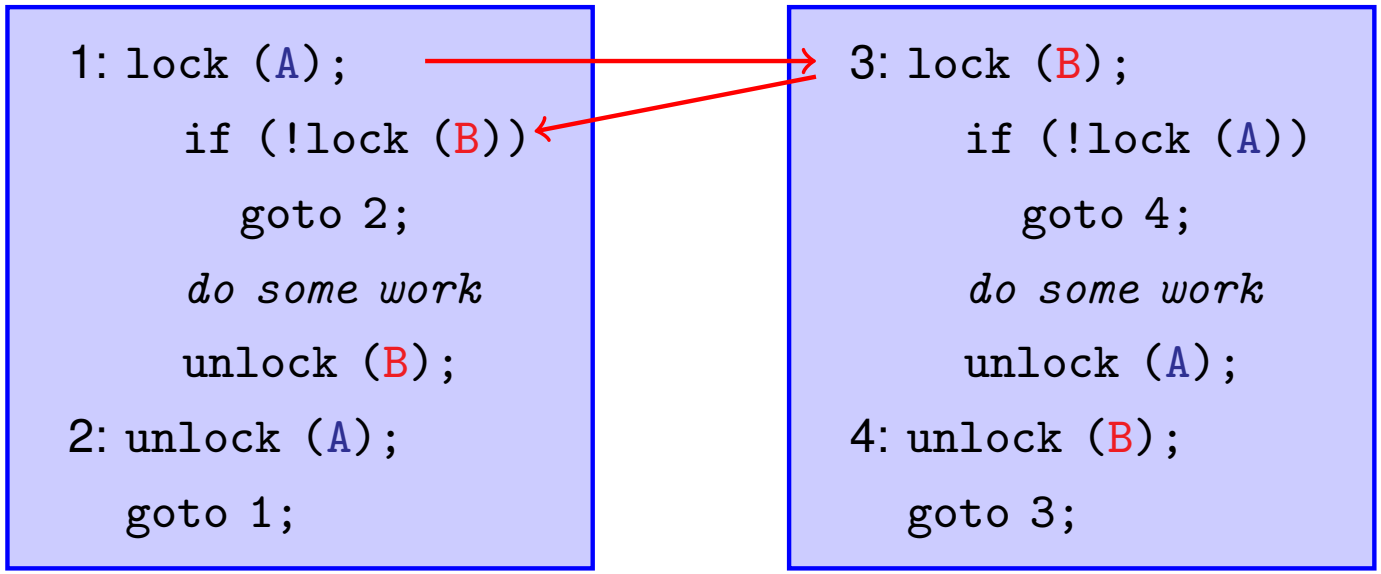
Attempt to Fix Deadlock

- Assume `lock` returns true on success, false otherwise
  - Thread one acquires lock A
  - Thread two acquires lock B



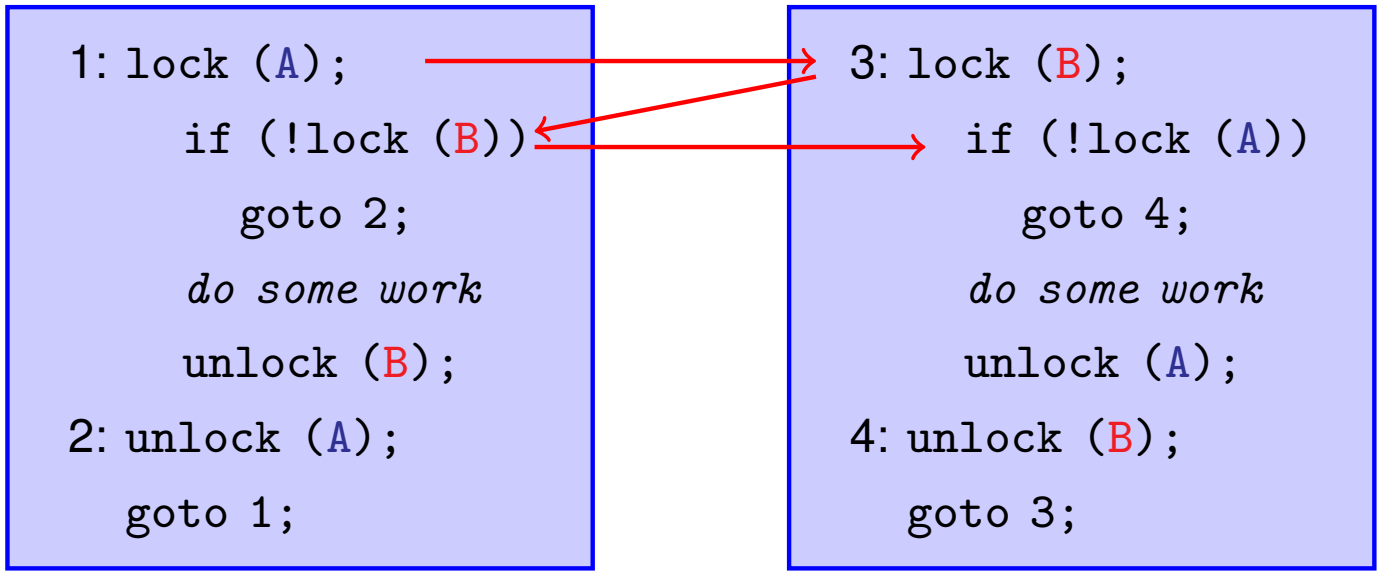
Attempt to Fix Deadlock

- Assume `lock` returns true on success, false otherwise
  - Thread one acquires lock A
  - Thread two acquires lock B
  - Thread one fails to acquire lock B



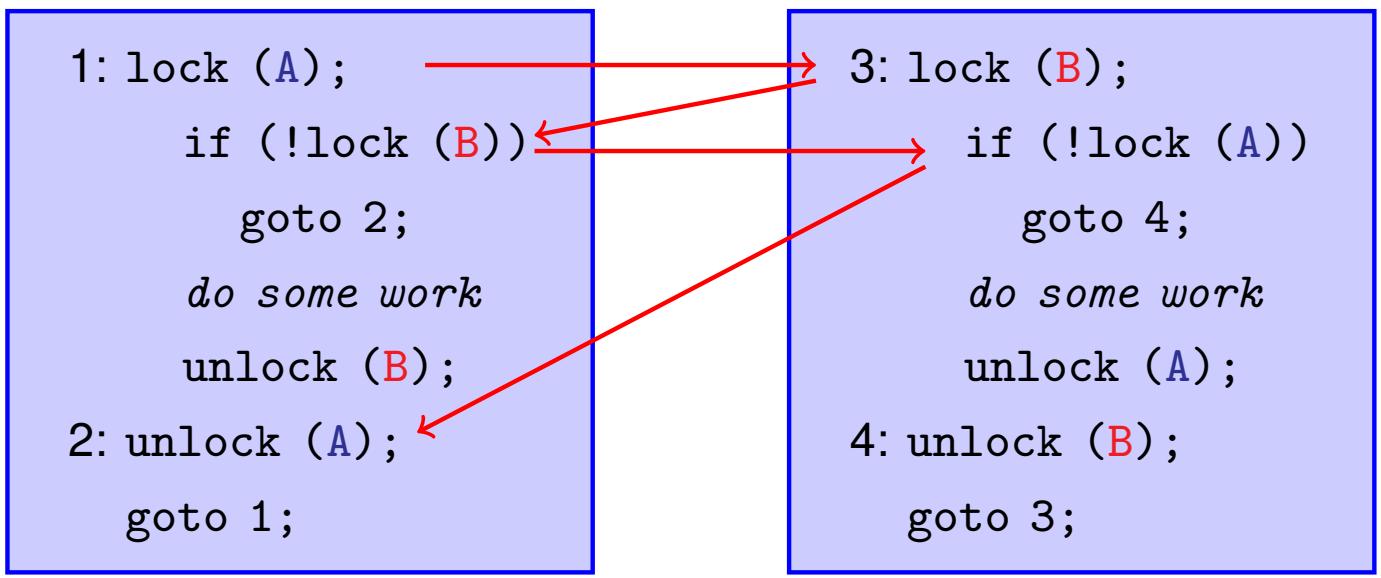
Attempt to Fix Deadlock

- Assume `lock` returns true on success, false otherwise
  - Thread one acquires lock A
  - Thread two acquires lock B
  - Thread one fails to acquire lock B
  - Thread two fails to acquire lock A



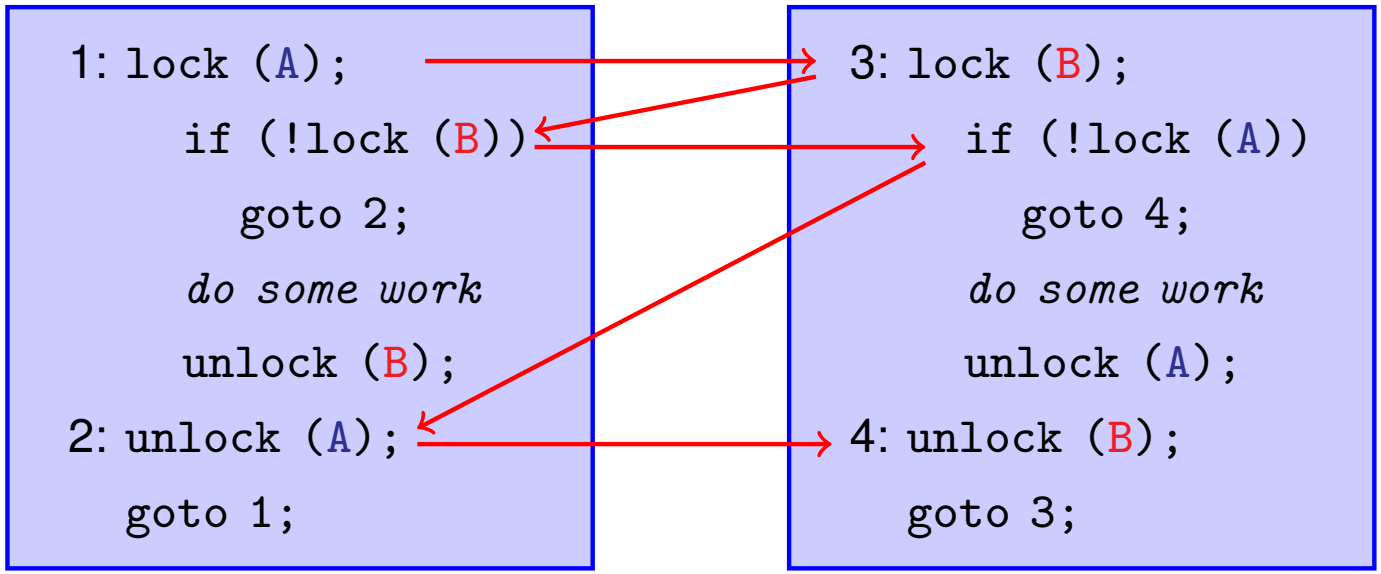
Attempt to Fix Deadlock

- Assume `lock` returns true on success, false otherwise
  - Thread one acquires lock A
  - Thread two acquires lock B
  - Thread one fails to acquire lock B
  - Thread two fails to acquire lock A
  - Thread one releases lock A



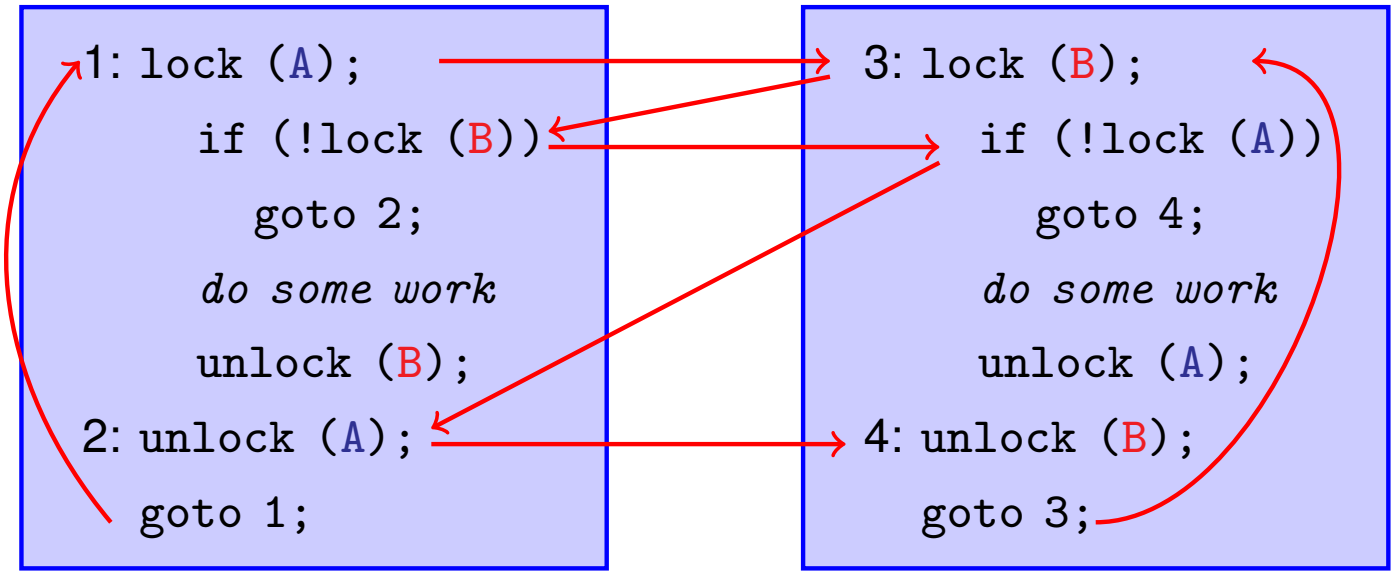
Attempt to Fix Deadlock

- Assume `lock` returns true on success, false otherwise
  - Thread one acquires lock A
  - Thread two acquires lock B
  - Thread one fails to acquire lock B
  - Thread two fails to acquire lock A
  - Thread one releases lock A
  - Thread two releases lock B



Attempt to Fix Deadlock

- Assume `lock` returns true on success, false otherwise
  - Thread one acquires lock A
  - Thread two acquires lock B
  - Thread one fails to acquire lock B
  - Thread two fails to acquire lock A
  - Thread one releases lock A
  - Thread two releases lock B
  - Scenario repeats (*livelock*)



Livelock

- Livelock can occur when algorithm detects and recovers from deadlock
- Deadlock detection can be repeatedly triggered
- Solution: ensure only one process takes action
  - randomized, priority, random timing (as in ethernet), ...

Race Conditions

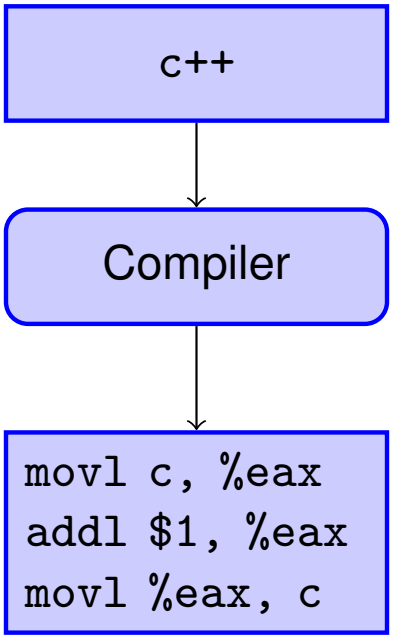
```
#include <stdio.h>
#include <pthread.h>

int c = 0;
void *count (void *parg)
{
  for (unsigned i=0; i<500000; i++)
    c++;
  return NULL;
}

int main (int argc, char** argv)
{
  pthread_t thread1, thread2;
  pthread_create (&thread1, NULL, count, NULL);
  pthread_create (&thread2, NULL, count, NULL);
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);

  printf ("%d\n", c);
  return 0;
}
```

Race Conditions

Race Conditions

Race Conditions

- Compile with `gcc -S threads.c`



Race Conditions

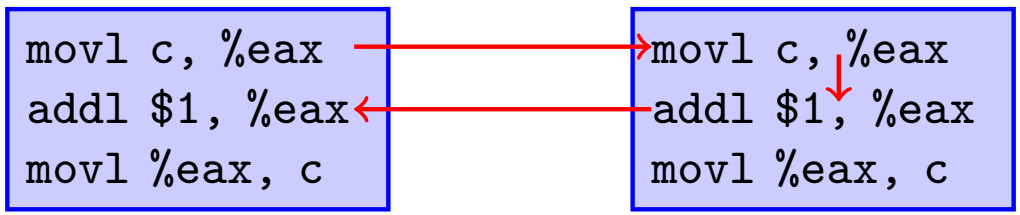
```
movl c, %eax
addl $1, %eax
movl %eax, c
```

```
movl c, %eax
addl $1, %eax
movl %eax, c
```

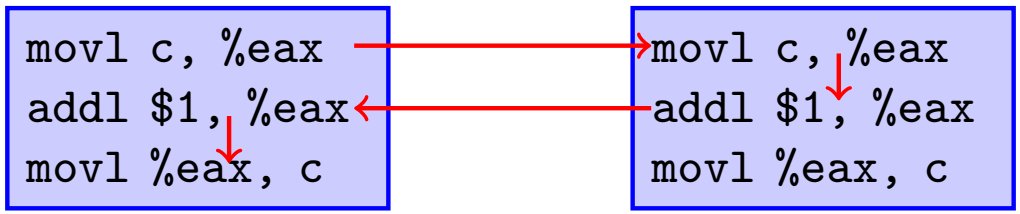
Race Conditions

Race Conditions

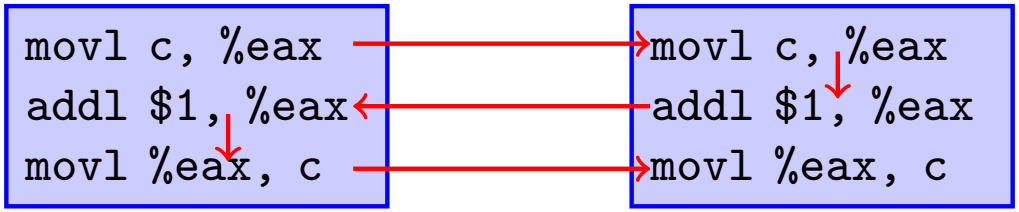
Race Conditions



Race Conditions



Race Conditions



Problem: Conflicting Variable Accesses

- ISO/IEC 14882:2011 §1.7 (The C++ Memory Model)

  “3 [. . .] Two threads of execution (1.10) can update and access separate memory locations without interfering with each other”

Problem: Conflicting Variable Accesses

- ISO/IEC 14882:2011 §1.7 (The C++ Memory Model)

  "3 [...] Two threads of execution (1.10) can update and access separate memory locations without interfering with each other"
- ISO/IEC 14882:2011 §1.10
  (Multi-threaded executions and data races)

  "3 [...] Two expression evaluations *conflict* if one of them modifies a memory location and the other one accesses or modifies the same memory location."

Problem: Conflicting Variable Accesses

- ISO/IEC 14882:2011 §1.7 (The C++ Memory Model)

  "3 [...] Two threads of execution (1.10) can update and access separate memory locations without interfering with each other"
- ISO/IEC 14882:2011 §1.10
  (Multi-threaded executions and data races)

  "3 [...] Two expression evaluations *conflict* if one of them modifies a memory location and the other one accesses or modifies the same memory location."
- Race condition:
  - two threads access same *unprotected* memory location
  - at least one of them is writing

Fixing Race Conditions

Thread 1

```
lock (A);
   c = c+1;
unlock (A);
```

Thread 2

```
lock (A);
   c = c+1;
unlock (A);
```

Fixing Race Conditions

Thread 1

```
lock (A);
   c = c+1;
unlock (A);
```

Thread 2

```
lock (A);
   c = c+1;
unlock (A);
```

- Does absence of race conditions mean program is free of (non-deadlock) concurrency bugs?

Order Violations

- Instructions can still be executed in unintended order

Protecting Access to Shared Variables

- Concurrent account deposit and withdrawal
- *Fine-grained* locking for performance reasons

```
lock (A);
   tmp1 = balance;
unlock (A);
tmp1 = tmp1 + deposit;
lock (A);
   balance = tmp1;
unlock (A);
```

```
lock (A);
   tmp2 = balance;
unlock (A);
tmp2 = tmp2 - withdrawal;
lock (A);
   balance = tmp2;
unlock (A);
```

Protecting Access to Shared Variables

- Concurrent account deposit and withdrawal
- *Fine-grained* locking for performance reasons
  - Thread one reads shared variable `balance`

```
lock (A);
   tmp1 = balance;
unlock (A);
tmp1 = tmp1 + deposit;
lock (A);
   balance = tmp1;
unlock (A);
```

```
lock (A);
   tmp2 = balance;
unlock (A);
tmp2 = tmp2 - withdrawal;
lock (A);
   balance = tmp2;
unlock (A);
```

Protecting Access to Shared Variables

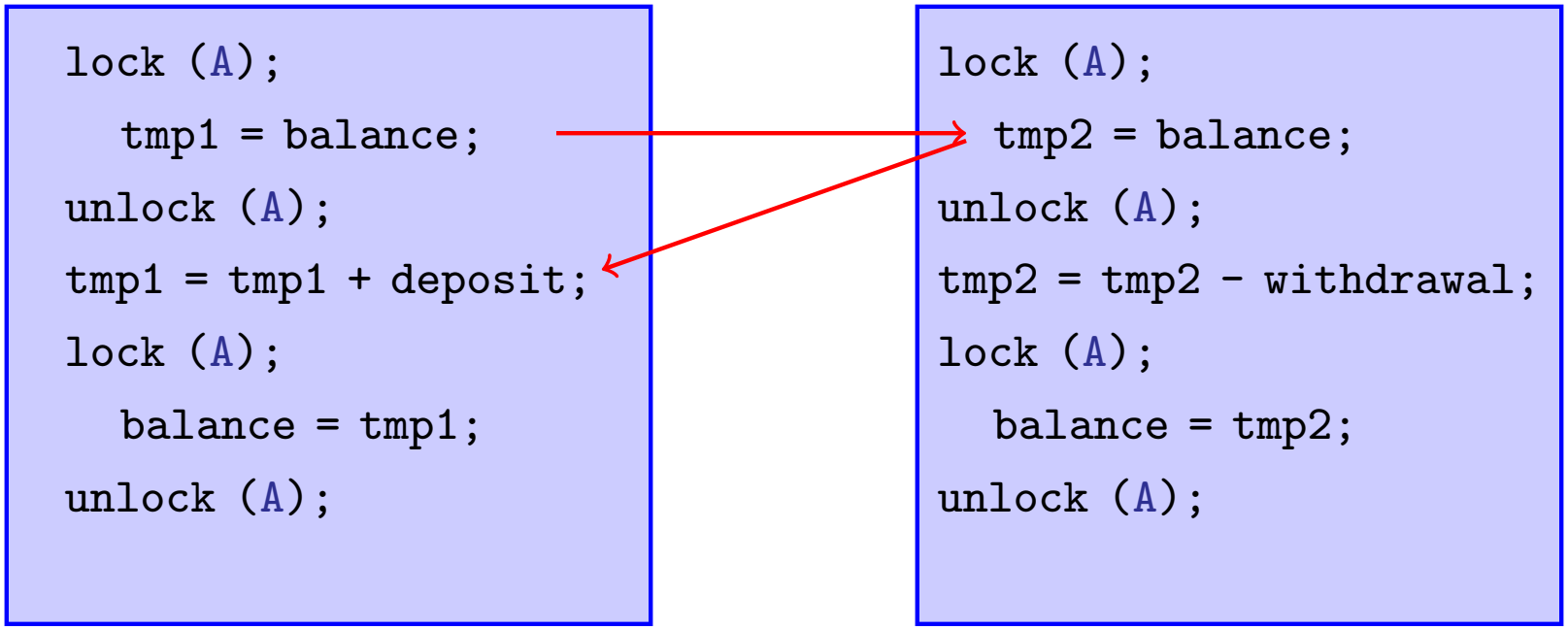
- Concurrent account deposit and withdrawal
- *Fine-grained* locking for performance reasons
  - Thread one reads shared variable `balance`
  - Thread two reads shared variable `balance`

```
lock (A);
  tmp1 = balance;
unlock (A);
tmp1 = tmp1 + deposit;
lock (A);
  balance = tmp1;
unlock (A);
```

```
lock (A);
  tmp2 = balance;
unlock (A);
tmp2 = tmp2 - withdrawal;
lock (A);
  balance = tmp2;
unlock (A);
```

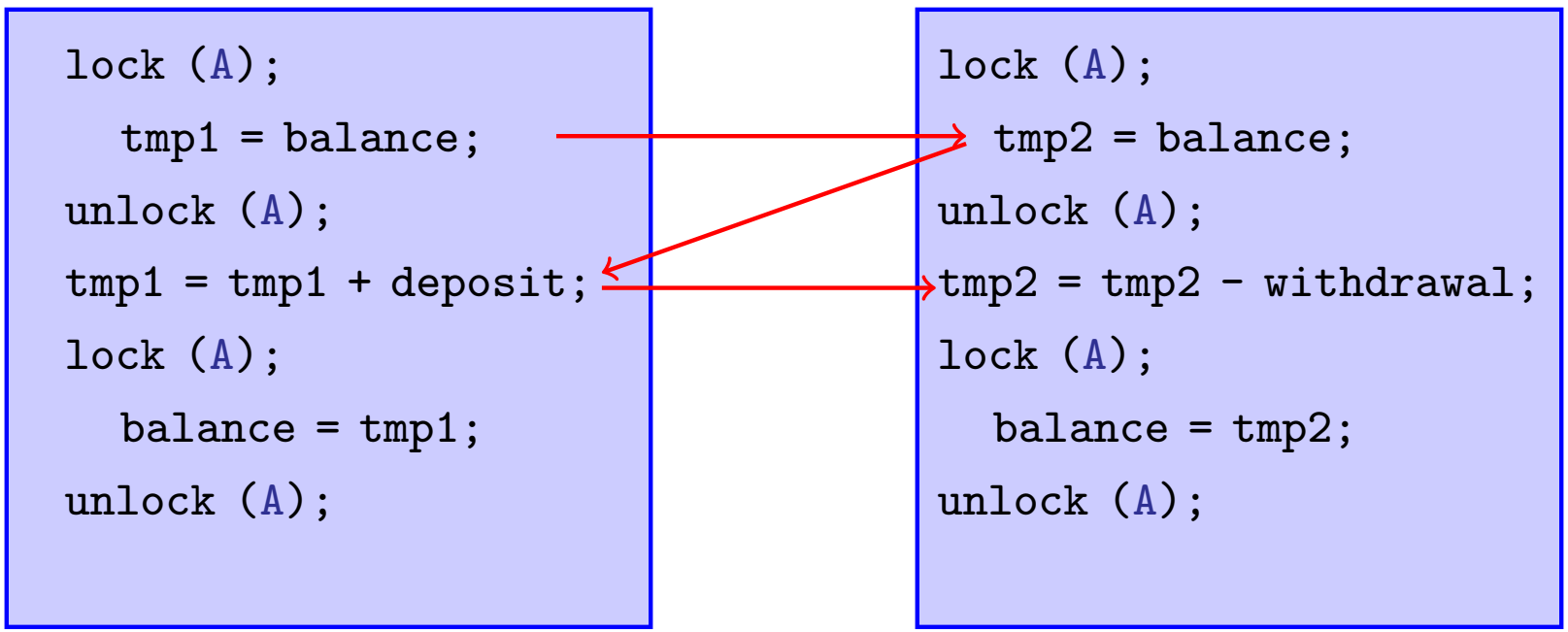
Protecting Access to Shared Variables

- Concurrent account deposit and withdrawal
- *Fine-grained* locking for performance reasons
  - Thread one reads shared variable `balance`
  - Thread two reads shared variable `balance`
  - Thread one adds deposit to *local copy* of `balance`



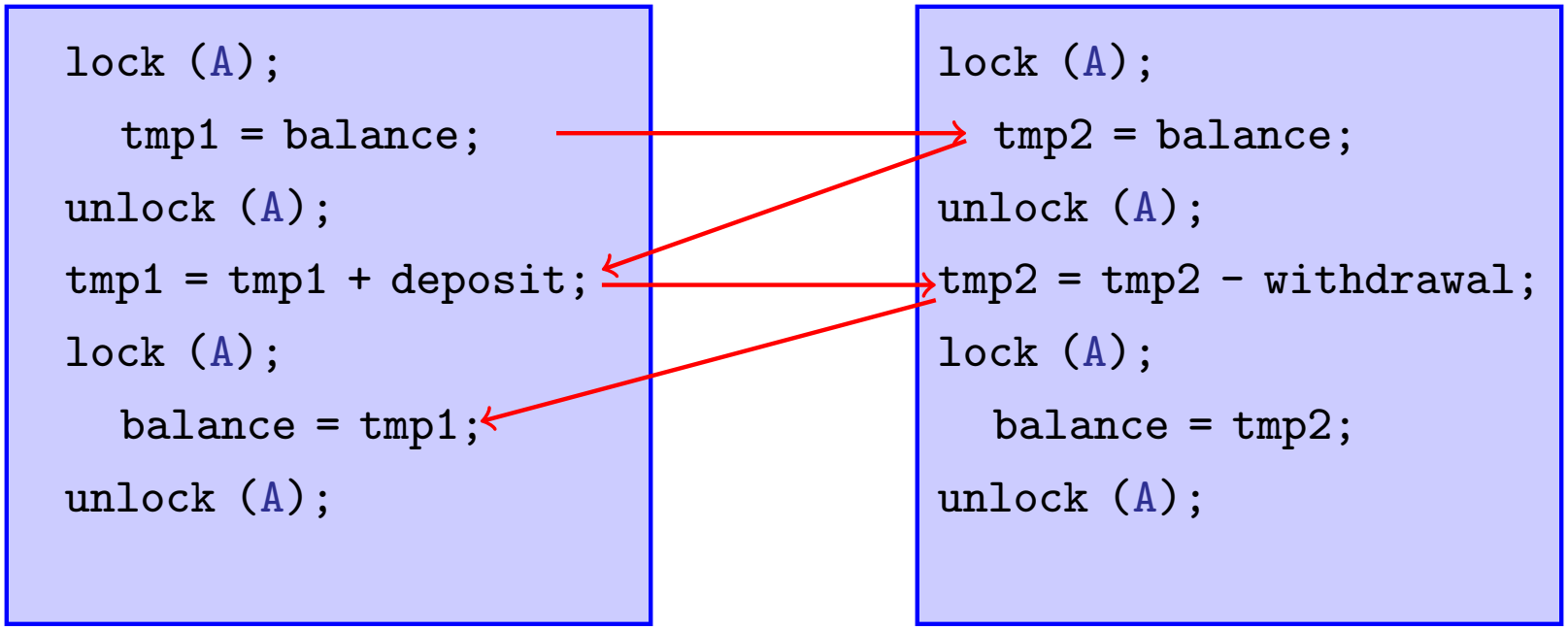
Protecting Access to Shared Variables

- Concurrent account deposit and withdrawal
- *Fine-grained* locking for performance reasons
  - Thread one reads shared variable `balance`
  - Thread two reads shared variable `balance`
  - Thread one adds deposit to *local copy* of `balance`
  - Thread two subtracts withdrawal from *local copy* of `balance`



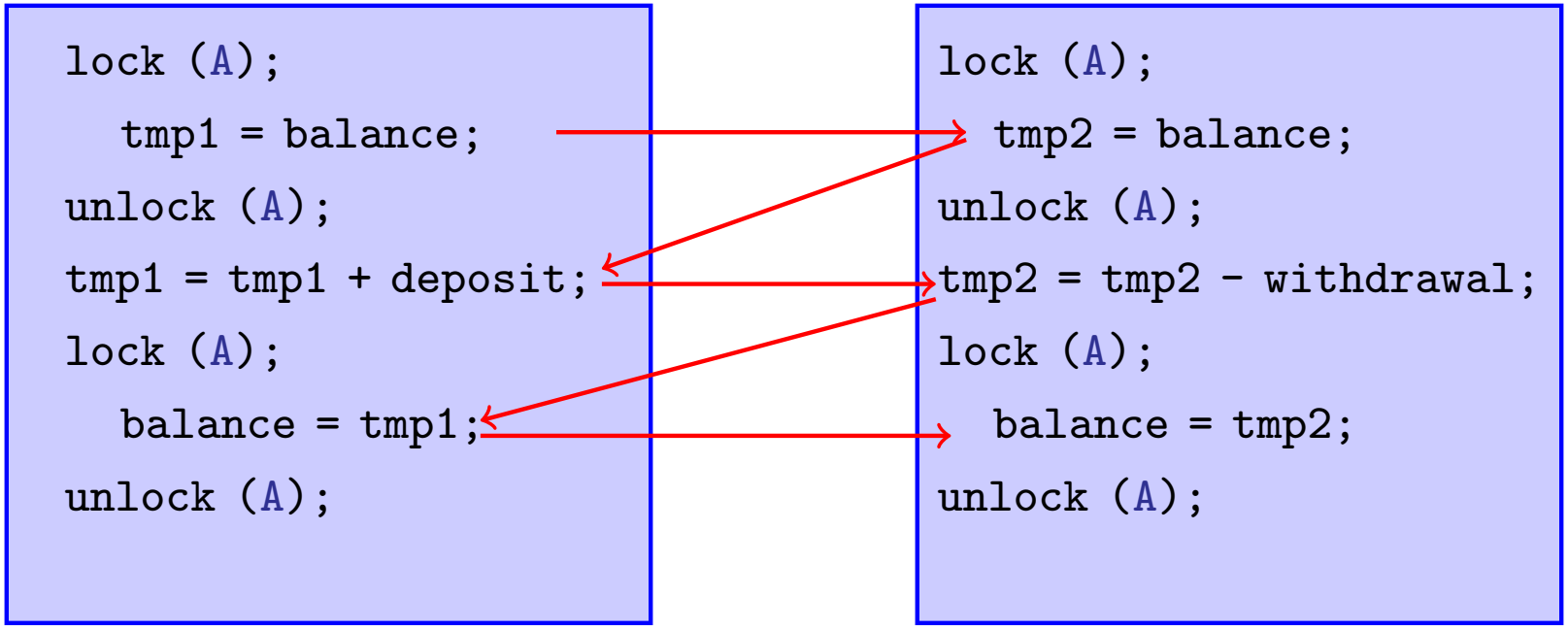
Protecting Access to Shared Variables

- Concurrent account deposit and withdrawal
- *Fine-grained* locking for performance reasons
  - Thread one reads shared variable `balance`
  - Thread two reads shared variable `balance`
  - Thread one adds deposit to *local copy* of `balance`
  - Thread two subtracts withdrawal from *local copy* of `balance`
  - Thread one stores result of transaction in `balance`



Protecting Access to Shared Variables

- Concurrent account deposit and withdrawal
- *Fine-grained* locking for performance reasons
  - Thread one reads shared variable `balance`
  - Thread two reads shared variable `balance`
  - Thread one adds deposit to *local copy* of `balance`
  - Thread two subtracts withdrawal from *local copy* of `balance`
  - Thread one stores result of transaction in `balance`
  - Thread two overwrites result of transaction of thread one



Atomicity Violation

- No race condition (since no conflicting access)
- Program disregards “intended” isolation/atomicity

Atomicity Violation

- No race condition (since no conflicting access)
- Program disregards “intended” isolation/atomicity
- Unlike race condition, depends on programmer’s intention
  - Cannot be detected automatically without annotations

Atomicity Violation

- No race condition (since no conflicting access)
- Program disregards “intended” isolation/atomicity
- Unlike race condition, depends on programmer’s intention
  - Cannot be detected automatically without annotations
  - Unrealistic to ask programmer to indicate “intended” atomic region (if s/he knew, there’d probably be no bug)

Atomicity Violation

- No race condition (since no conflicting access)
- Program disregards “intended” isolation/atomicity
- Unlike race condition, depends on programmer’s intention
  - Cannot be detected automatically without annotations
  - Unrealistic to ask programmer to indicate “intended” atomic region (if s/he knew, there’d probably be no bug)
  - Assert result instead (i.e., testing):

```
assert (balance ==
                old_balance + deposit - withdrawal);
```

Atomicity Violation

- No race condition (since no conflicting access)
- Program disregards “intended” isolation/atomicity
- Unlike race condition, depends on programmer’s intention
  - Cannot be detected automatically without annotations
  - Unrealistic to ask programmer to indicate “intended” atomic region (if s/he knew, there’d probably be no bug)
  - Assert result instead (i.e., testing):

```
assert (balance ==
                old_balance + deposit - withdrawal);
```

- Alternatively, use *sequential* reference implementation and compare results! (cf. Pex for Fun!)

Heisenbugs

```
old_balance = balance;
```

```
lock (A);
  tmp1 = balance;
unlock (A);
tmp1 = tmp1 + deposit;
lock (A);
  balance = tmp1;
unlock (A);
```

```
lock (A);
  tmp2 = balance;
unlock (A);
tmp2 = tmp2 - withdrawal;
lock (A);
  balance = tmp2;
unlock (A);
```

```
assert (balance == old_balance + deposit - withdrawal);
```

Heisenbugs

```
old_balance = balance;
```

```
lock (A);
  tmp1 = balance;
unlock (A);
tmp1 = tmp1 + deposit;
lock (A);
  balance = tmp1;
unlock (A);
```

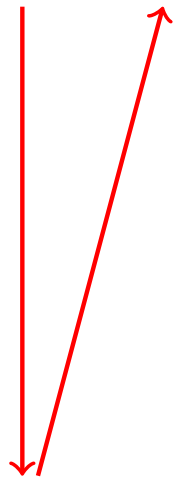
```
lock (A);
  tmp2 = balance;
unlock (A);
tmp2 = tmp2 - withdrawal;
lock (A);
  balance = tmp2;
unlock (A);
```

```
assert (balance == old_balance + deposit - withdrawal);
```

Heisenbugs

```
old_balance = balance;
```

```
lock (A);
  tmp1 = balance;
unlock (A);
tmp1 = tmp1 + deposit;
lock (A);
  balance = tmp1;
unlock (A);
```



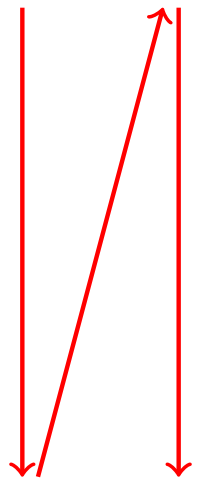
```
lock (A);
  tmp2 = balance;
unlock (A);
tmp2 = tmp2 - withdrawal;
lock (A);
  balance = tmp2;
unlock (A);
```

```
assert (balance == old_balance + deposit - withdrawal);
```

Heisenbugs

```
old_balance = balance;
```

```
lock (A);
  tmp1 = balance;
unlock (A);
tmp1 = tmp1 + deposit;
lock (A);
  balance = tmp1;
unlock (A);
```



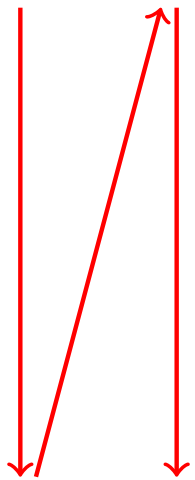
```
lock (A);
  tmp2 = balance;
unlock (A);
tmp2 = tmp2 - withdrawal;
lock (A);
  balance = tmp2;
unlock (A);
```

```
assert (balance == old_balance + deposit - withdrawal);
```

Heisenbugs

```
old_balance = balance;
```

```
lock (A);
  tmp1 = balance;
unlock (A);
tmp1 = tmp1 + deposit;
lock (A);
  balance = tmp1;
unlock (A);
```



```
lock (A);
  tmp2 = balance;
unlock (A);
tmp2 = tmp2 - withdrawal;
lock (A);
  balance = tmp2;
unlock (A);
```

```
assert (balance == old_balance + deposit - withdrawal);
```

Bug does not happen in every execution!

Heisenbugs

"a software bug that seems to disappear or alter its behavior when one attempts to study it"

- Concurrency bugs are one example of Heisenbugs
- Why?

Heisenbugs

"a software bug that seems to disappear or alter its behavior when one attempts to study it"

- Concurrency bugs are one example of Heisenbugs
- Why? No control over scheduler!

Heisenbugs: Caused by Scheduler

- Assume all inputs of the program are *fixed* (i.e., test case)
- Then what causes variation of program behaviour?

Heisenbugs: Caused by Scheduler

- Assume all inputs of the program are *fixed* (i.e., test case)
- Then what causes variation of program behaviour?
  - Change of schedule results in change of data-flow
  - Remember from lecture on Coverage Criteria:

$$\underbrace{\mathtt{x}}_{\text{defined}} := \underbrace{\mathtt{y} + \mathtt{z}}_{\text{used}}$$

  - Execution results in (ordered) sequence of read/write events:

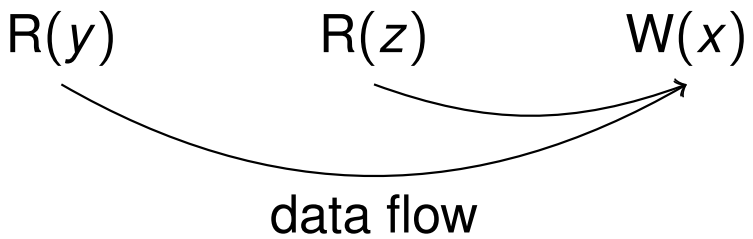
$$\mathrm{R}(y) \qquad \mathrm{R}(z) \qquad \mathrm{W}(x)$$

Heisenbugs: Caused by Scheduler

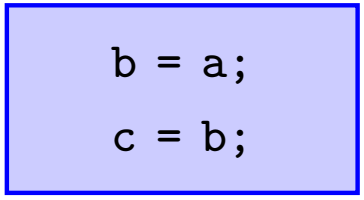
- Assume all inputs of the program are *fixed* (i.e., test case)
- Then what causes variation of program behaviour?
  - Change of schedule results in change of data-flow
  - Remember from lecture on Coverage Criteria:

$$\underbrace{\mathtt{x}}_{\text{defined}} := \underbrace{\mathtt{y} + \mathtt{z}}_{\text{used}}$$

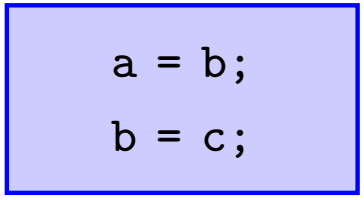
  - Execution results in (ordered) sequence of read/write events:



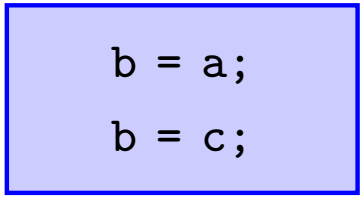
Data Dependencies *within* a Thread



Flow dependency



Anti-dependency

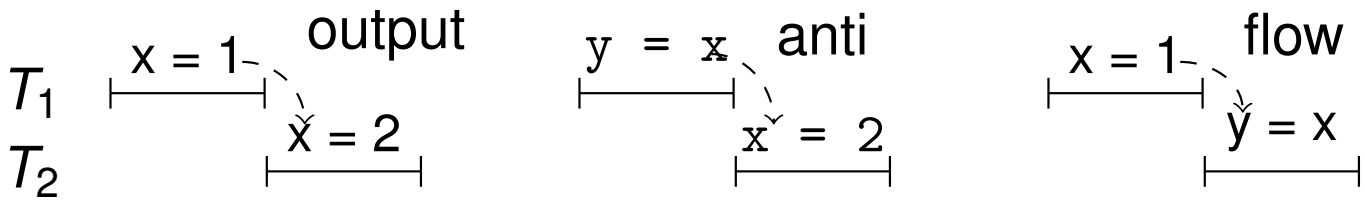


Output-dependency

- **Flow dependency**: R(a) W(b) R(b) W(c)
  - Read-after-Write (RAW)
  - `c = b` depends on result of `b = a`
- **Anti-dependency**: R(b) W(a) R(c) W(b)
  - Write-after-Read (WAR)
  - `b = c` must happen after `a = b`
- **Output-dependency**: R(a) W(b) R(c) W(b)
  - Write-after-Write (WAW)
  - `b = c` overwrites result of `b = a`

Inter-Thread Data Dependencies

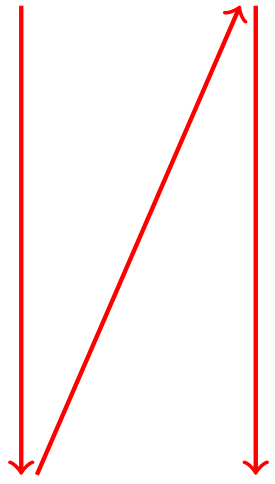
Data-dependencies (“hazards”) between threads are similar:



Inter-Thread Data Dependencies

- *Intra-Thread* (or *thread-local*) data flow and dependencies are determined by program/instruction order
- *Inter-Thread* data dependencies may vary from execution to execution ("data hazard"):

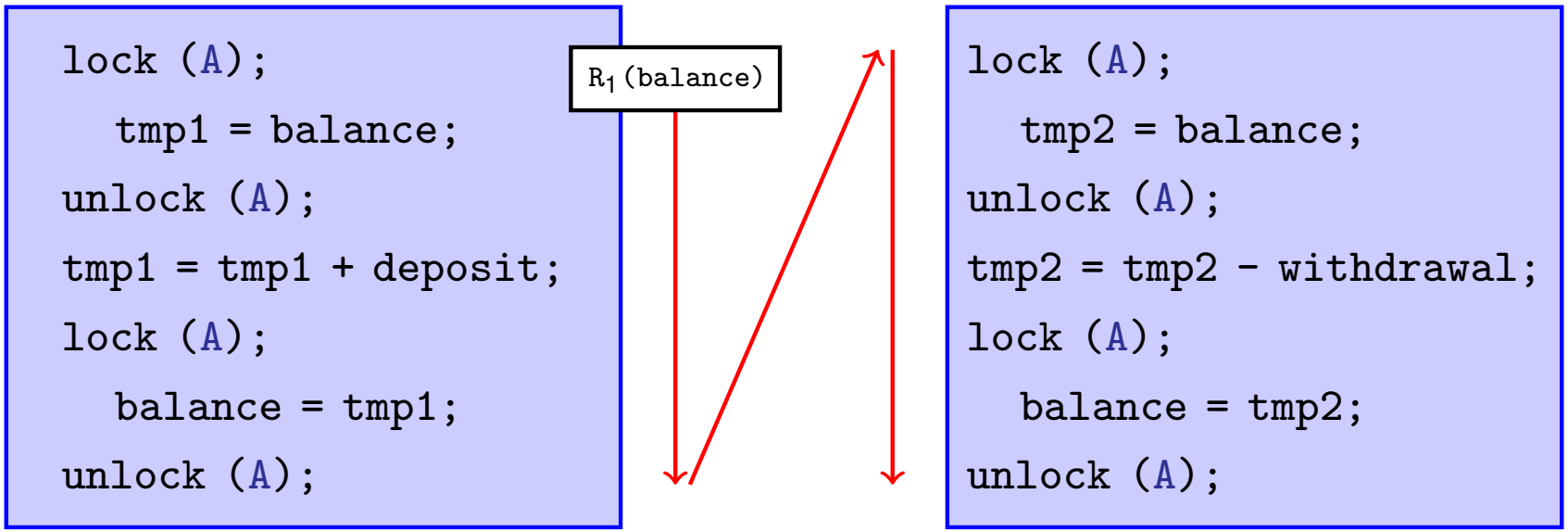
```
lock (A);
  tmp1 = balance;
unlock (A);
tmp1 = tmp1 + deposit;
lock (A);
  balance = tmp1;
unlock (A);
```



```
lock (A);
  tmp2 = balance;
unlock (A);
tmp2 = tmp2 - withdrawal;
lock (A);
  balance = tmp2;
unlock (A);
```

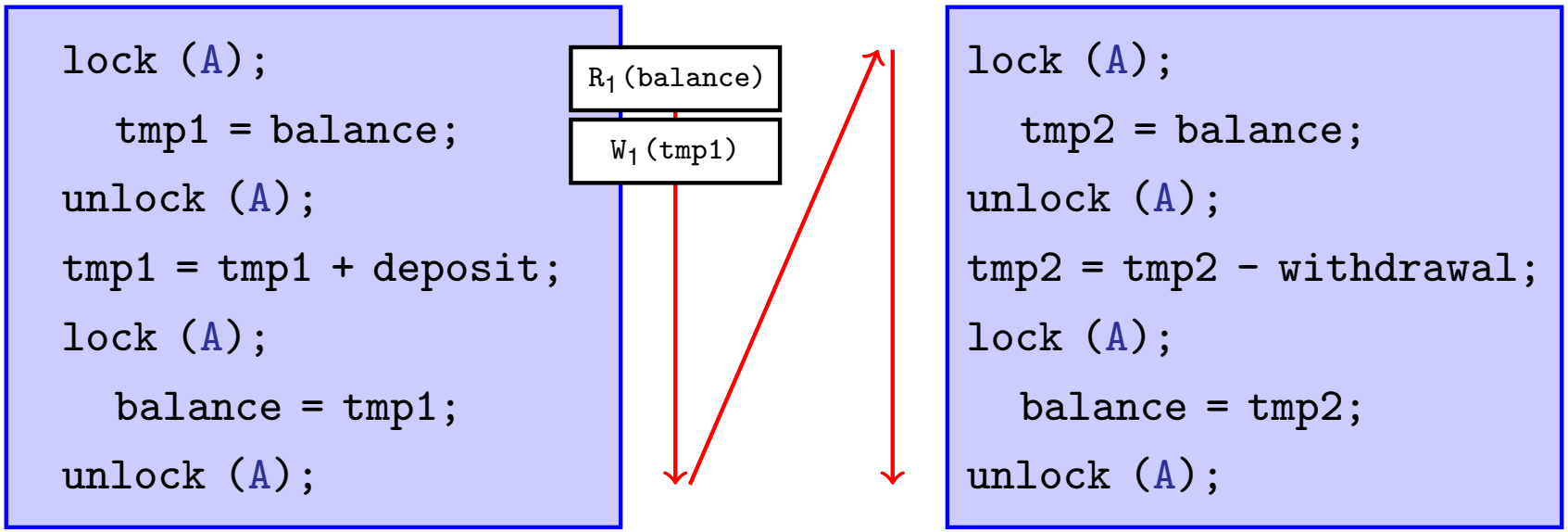
Inter-Thread Data Dependencies

- *Intra-Thread* (or *thread-local*) data flow and dependencies are determined by program/instruction order
- *Inter-Thread* data dependencies may vary from execution to execution (“data hazard”):



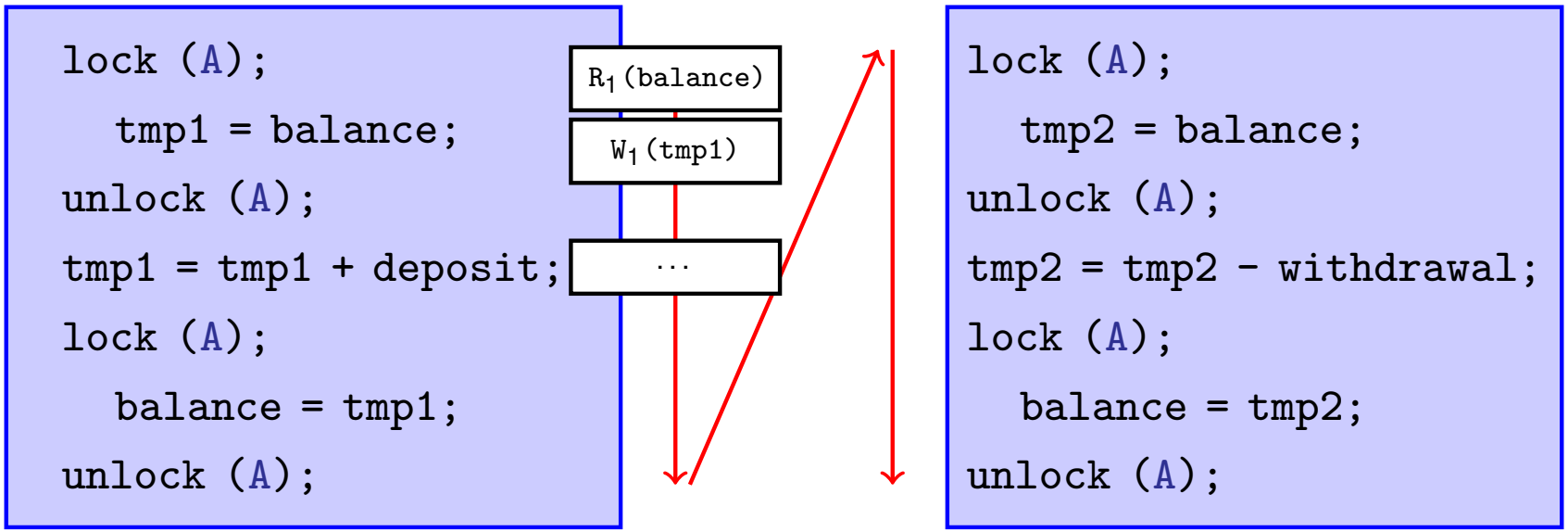
Inter-Thread Data Dependencies

- *Intra-Thread* (or *thread-local*) data flow and dependencies are determined by program/instruction order
- *Inter-Thread* data dependencies may vary from execution to execution (“data hazard”):



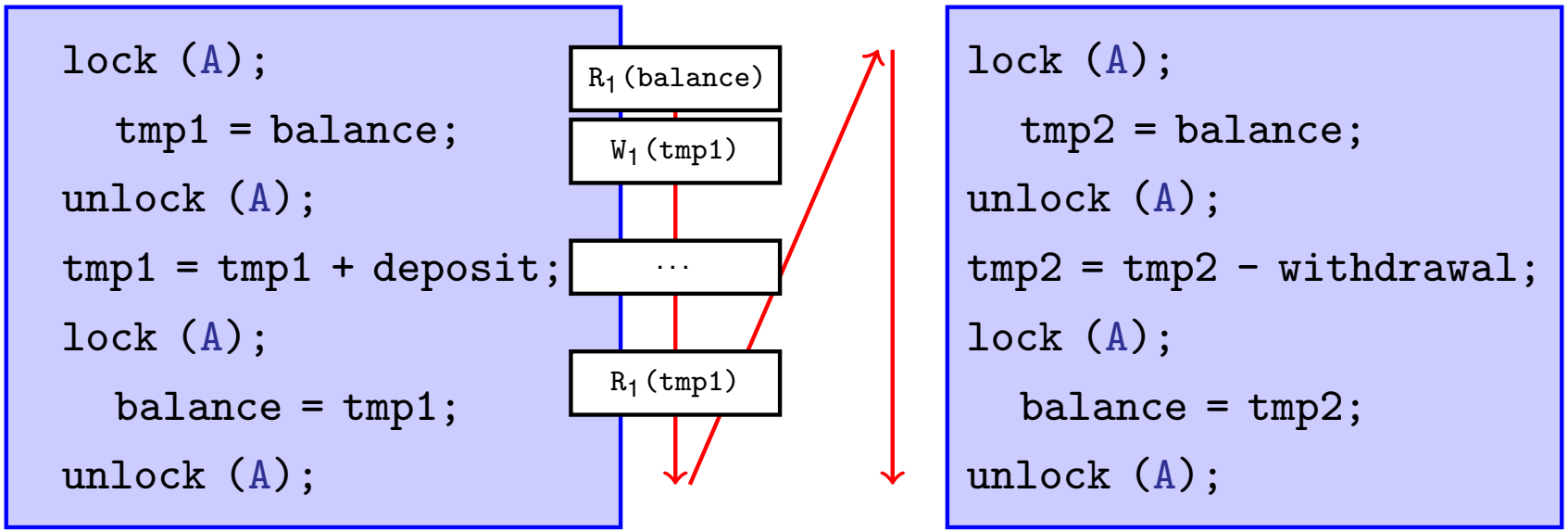
Inter-Thread Data Dependencies

- *Intra-Thread* (or *thread-local*) data flow and dependencies are determined by program/instruction order
- *Inter-Thread* data dependencies may vary from execution to execution (“data hazard”):



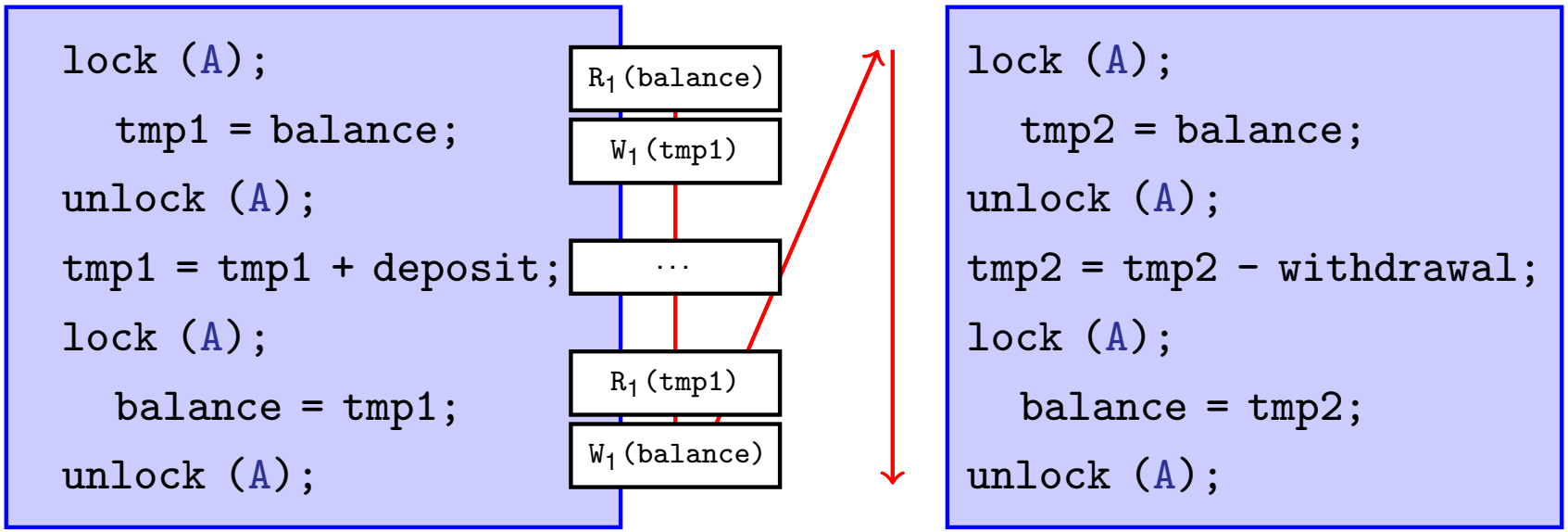
Inter-Thread Data Dependencies

- *Intra-Thread* (or *thread-local*) data flow and dependencies are determined by program/instruction order
- *Inter-Thread* data dependencies may vary from execution to execution (“data hazard”):



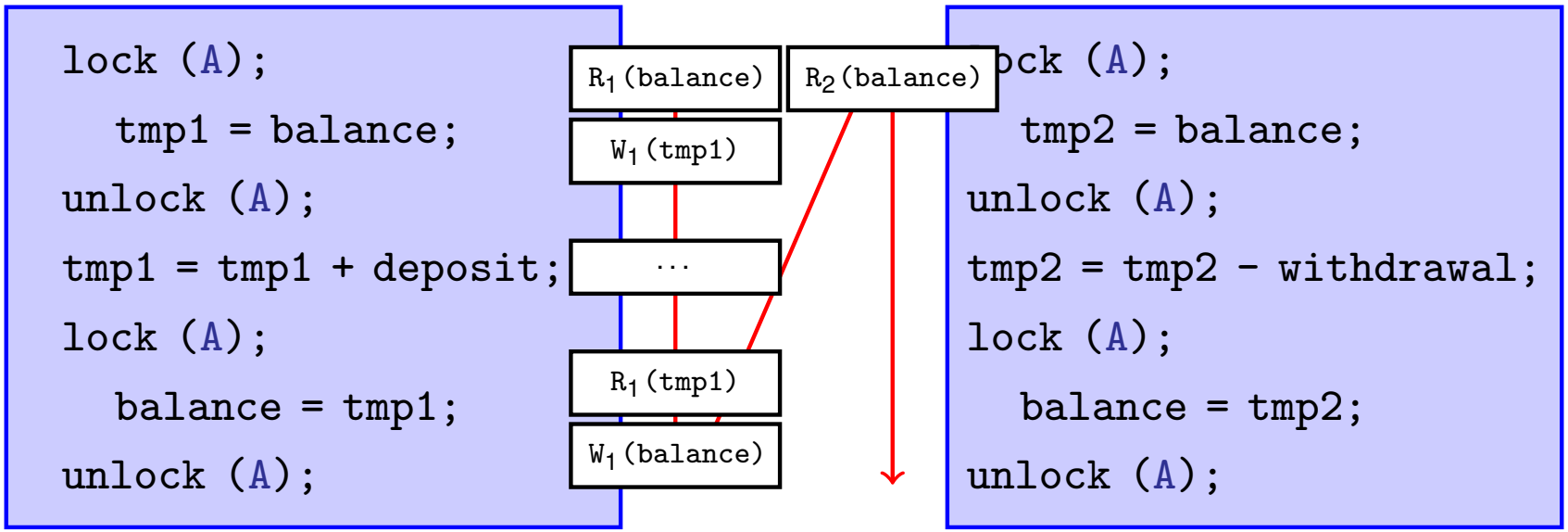
Inter-Thread Data Dependencies

- *Intra-Thread* (or *thread-local*) data flow and dependencies are determined by program/instruction order
- *Inter-Thread* data dependencies may vary from execution to execution (“data hazard”):



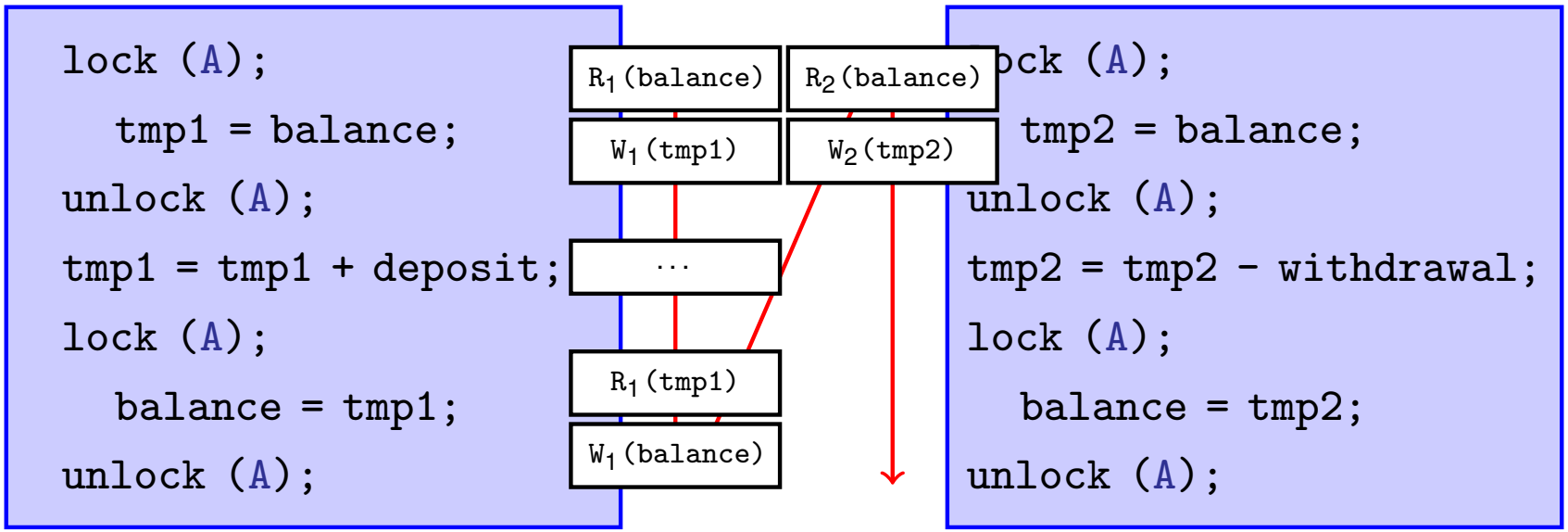
Inter-Thread Data Dependencies

- *Intra-Thread* (or *thread-local*) data flow and dependencies are determined by program/instruction order
- *Inter-Thread* data dependencies may vary from execution to execution (“data hazard”):



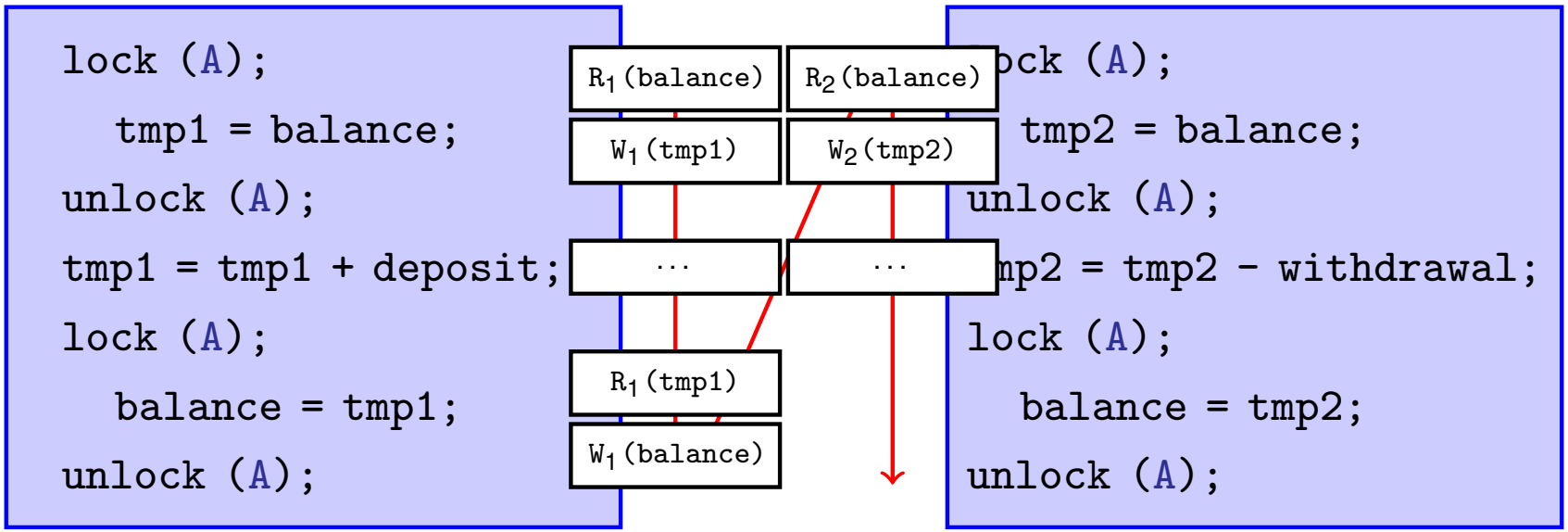
Inter-Thread Data Dependencies

- *Intra-Thread* (or *thread-local*) data flow and dependencies are determined by program/instruction order
- *Inter-Thread* data dependencies may vary from execution to execution (“data hazard”):



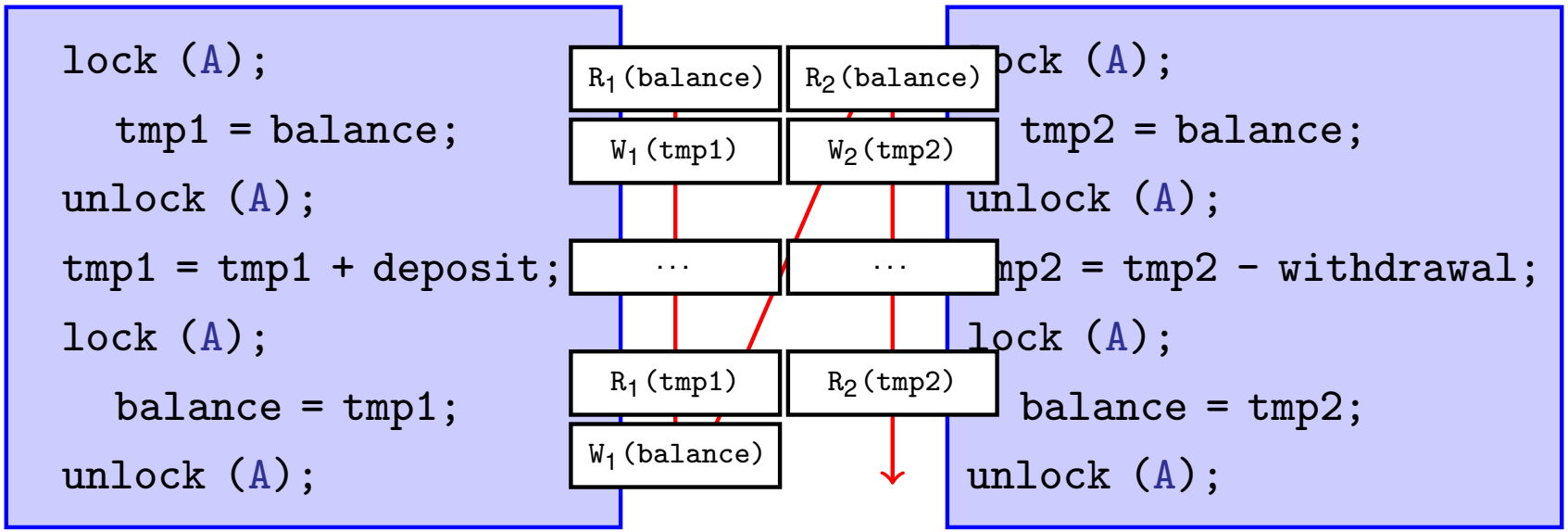
Inter-Thread Data Dependencies

- *Intra-Thread* (or *thread-local*) data flow and dependencies are determined by program/instruction order
- *Inter-Thread* data dependencies may vary from execution to execution (“data hazard”):



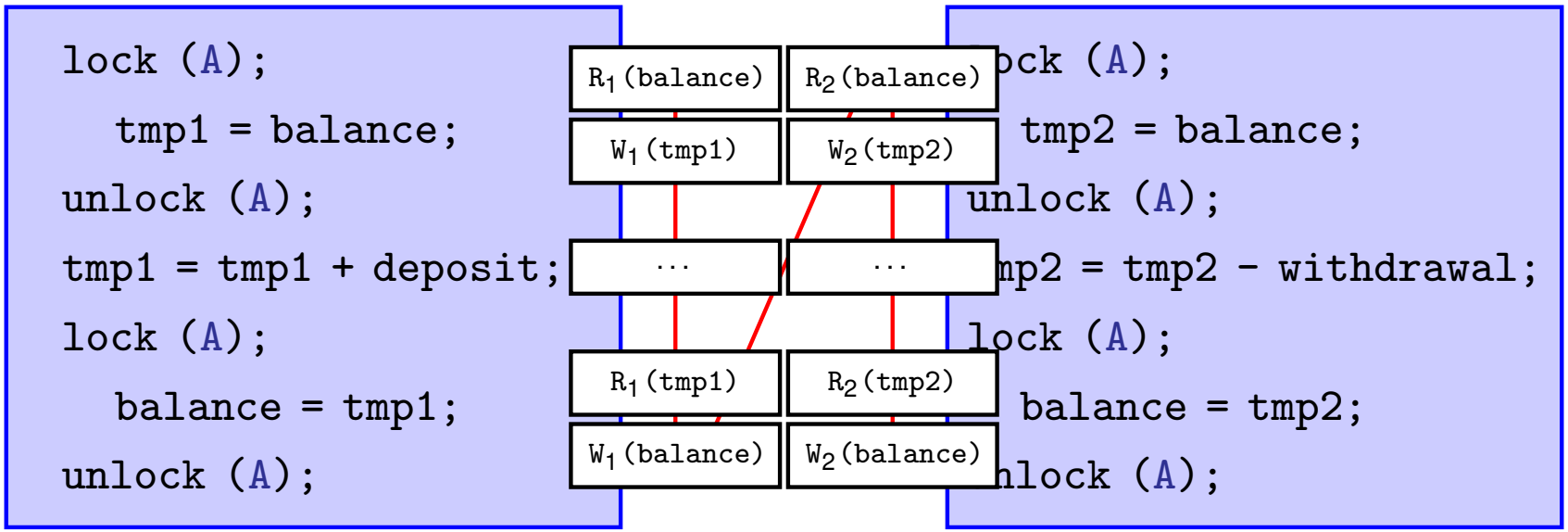
Inter-Thread Data Dependencies

- *Intra-Thread* (or *thread-local*) data flow and dependencies are determined by program/instruction order
- *Inter-Thread* data dependencies may vary from execution to execution (“data hazard”):



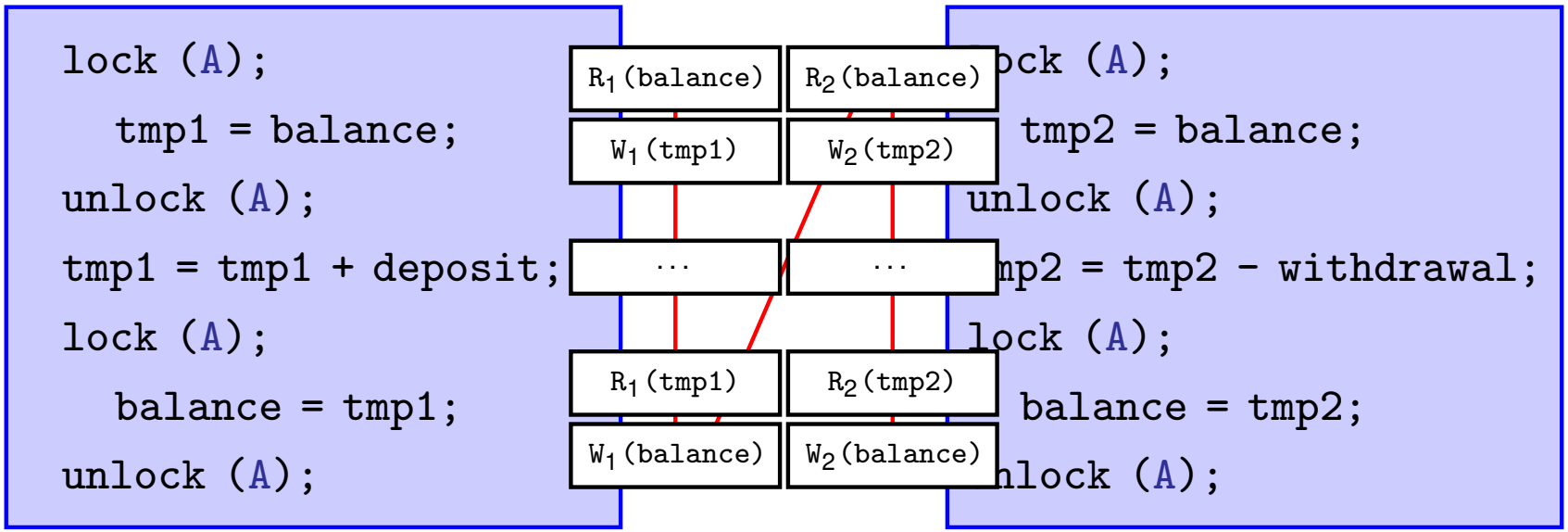
Inter-Thread Data Dependencies

- *Intra-Thread* (or *thread-local*) data flow and dependencies are determined by program/instruction order
- *Inter-Thread* data dependencies may vary from execution to execution (“data hazard”):



Inter-Thread Data Dependencies

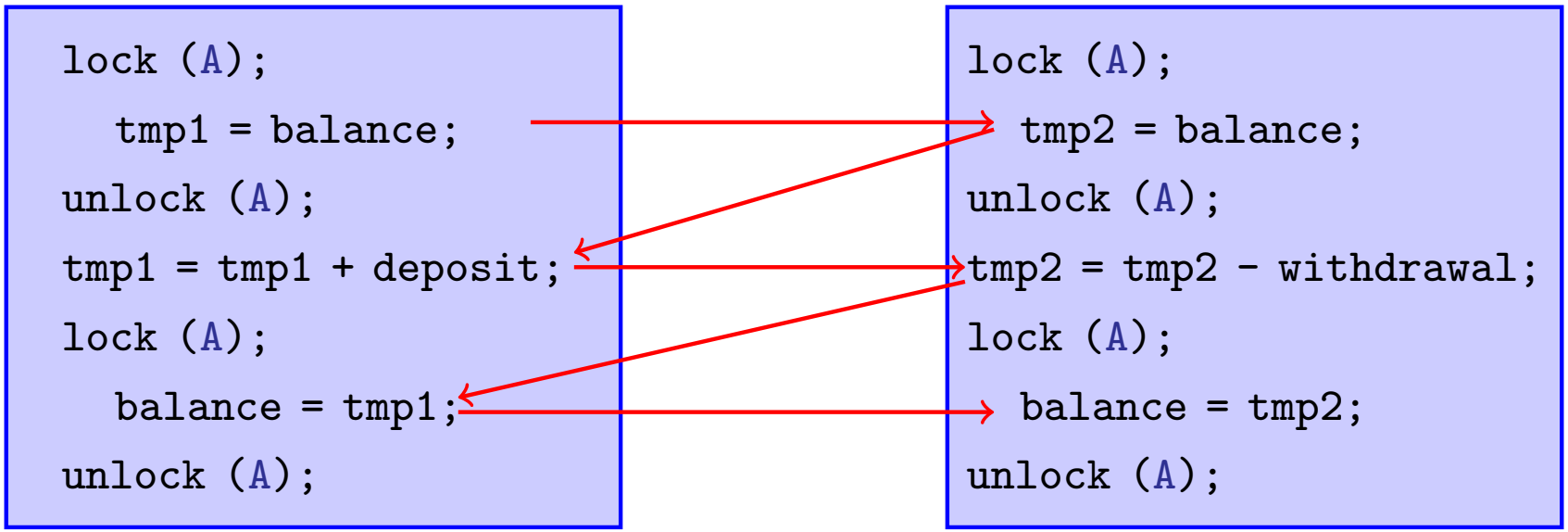
- *Intra-Thread* (or *thread-local*) data flow and dependencies are determined by program/instruction order
- *Inter-Thread* data dependencies may vary from execution to execution (“data hazard”):



- Subscripts of R_1, W_1, R_2, W_2 indicate thread!
- Intra-thread order not relevant for outcome!

Inter-Thread Data Dependencies

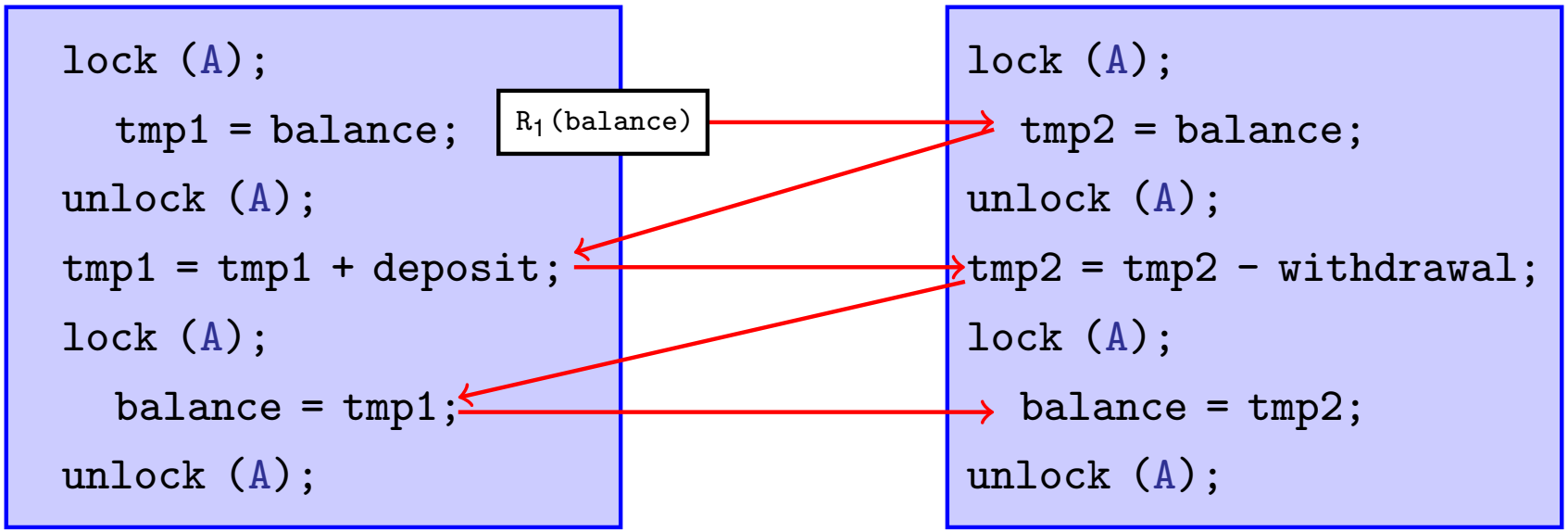
- *Intra-Thread* (or *thread-local*) data flow and dependencies are determined by program/instruction order
- *Inter-Thread* data dependencies may vary from execution to execution ("data hazard"):



- Subscripts of R_1, W_1, R_2, W_2 indicate thread!
- Intra-thread order not relevant for outcome!

Inter-Thread Data Dependencies

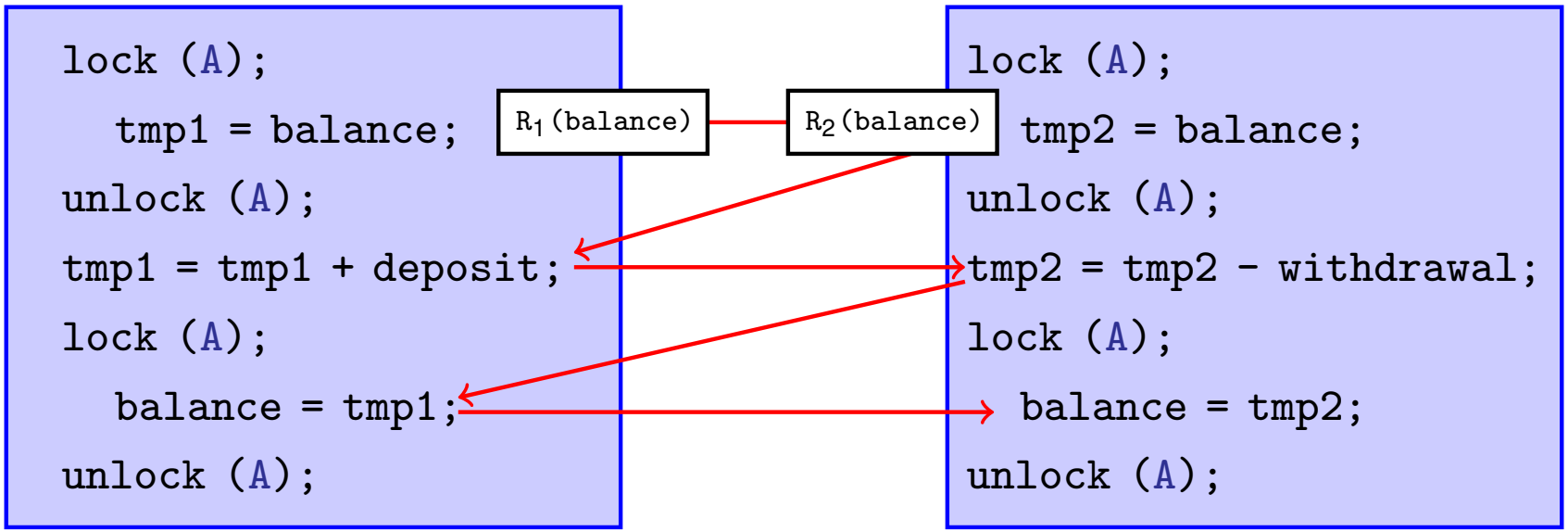
- *Intra-Thread* (or *thread-local*) data flow and dependencies are determined by program/instruction order
- *Inter-Thread* data dependencies may vary from execution to execution (“data hazard”):



- Subscripts of R_1, W_1, R_2, W_2 indicate thread!
- Intra-thread order not relevant for outcome!

Inter-Thread Data Dependencies

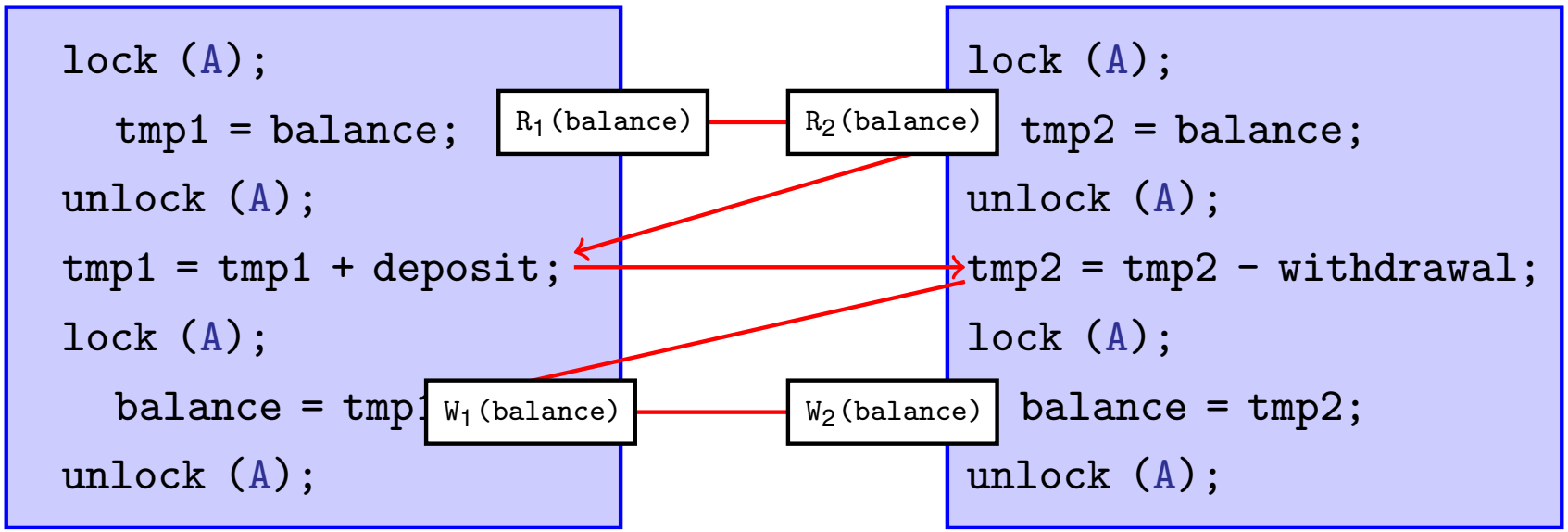
- *Intra-Thread* (or *thread-local*) data flow and dependencies are determined by program/instruction order
- *Inter-Thread* data dependencies may vary from execution to execution (“data hazard”):



- Subscripts of R_1, W_1, R_2, W_2 indicate thread!
- Intra-thread order not relevant for outcome!

Inter-Thread Data Dependencies

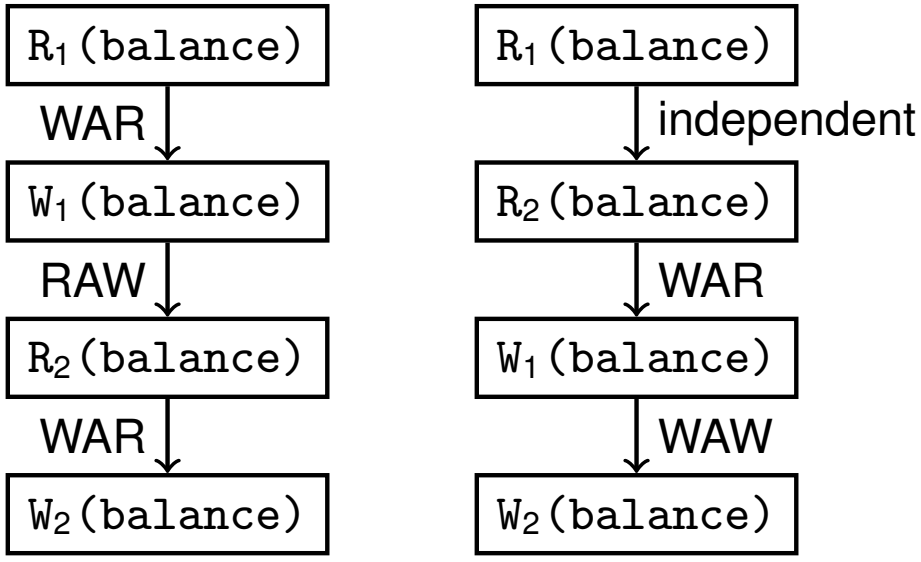
- *Intra-Thread* (or *thread-local*) data flow and dependencies are determined by program/instruction order
- *Inter-Thread* data dependencies may vary from execution to execution (“data hazard”):



- Subscripts of R_1, W_1, R_2, W_2 indicate thread!
- Intra-thread order not relevant for outcome!

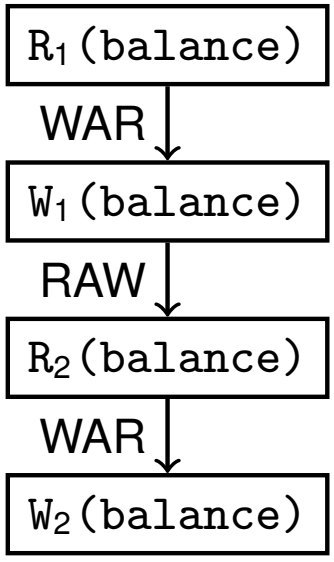
Inter-Thread Data Dependencies

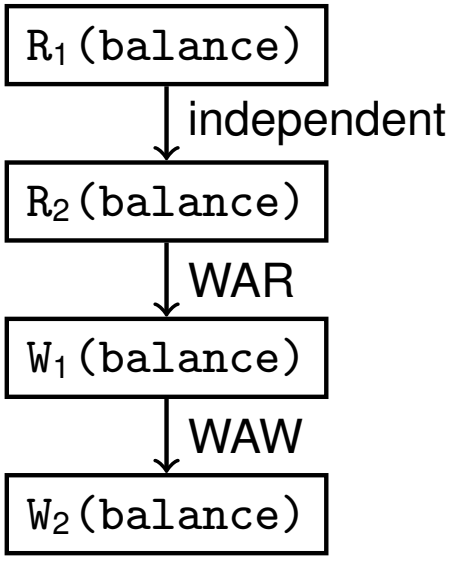
- Outcome depends on *inter-thread* order of R/W Accesses to *shared* variables



Inter-Thread Data Dependencies

- Outcome depends on *inter-thread* order of R/W Accesses to *shared* variables

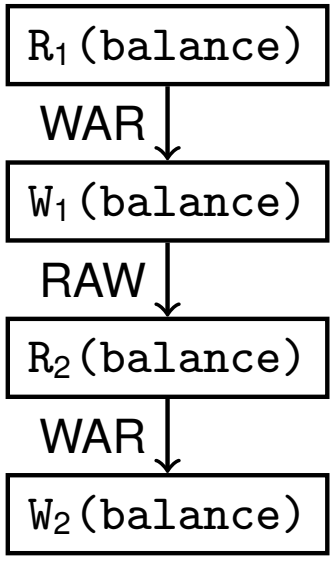




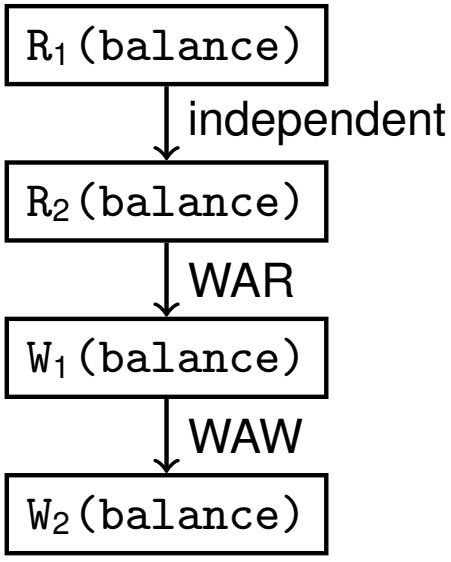
✗

Inter-Thread Data Dependencies

- Outcome depends on *inter-thread* order of R/W Accesses to *shared* variables



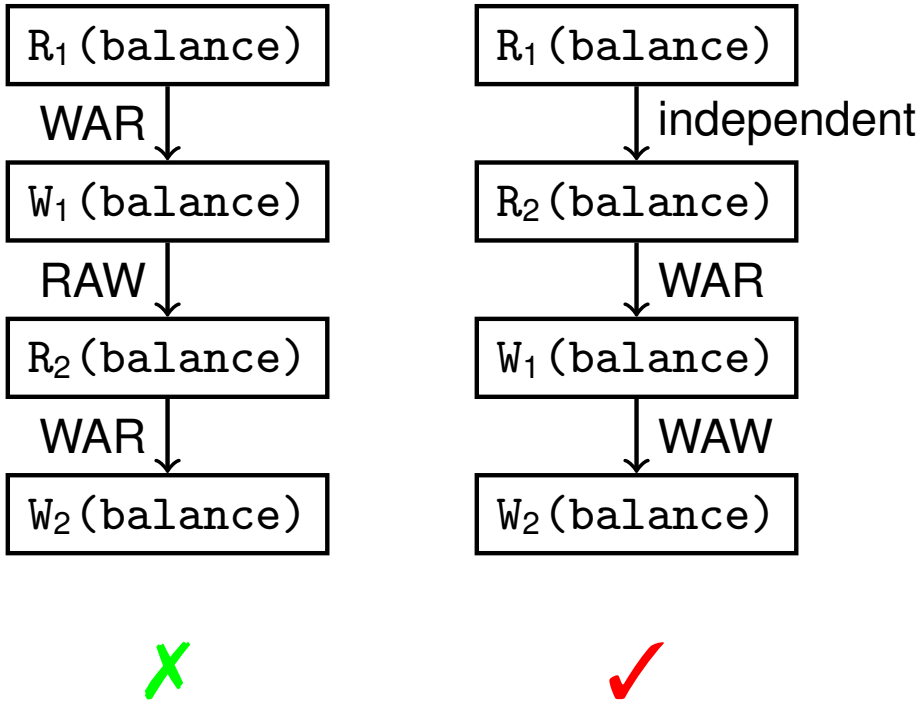
✗



✓

Inter-Thread Data Dependencies

- Outcome depends on *inter-thread* order of R/W Accesses to *shared* variables



- Note: same input values, different output values

Reproducing Heisenbugs?

- For same input, different schedules produce different result
- Assertion may fail in some executions, not in others
- Resulting Challenges:
  1. Reproducing Heisenbugs
  2. Understanding Heisenbugs

Reproducing Heisenbugs?

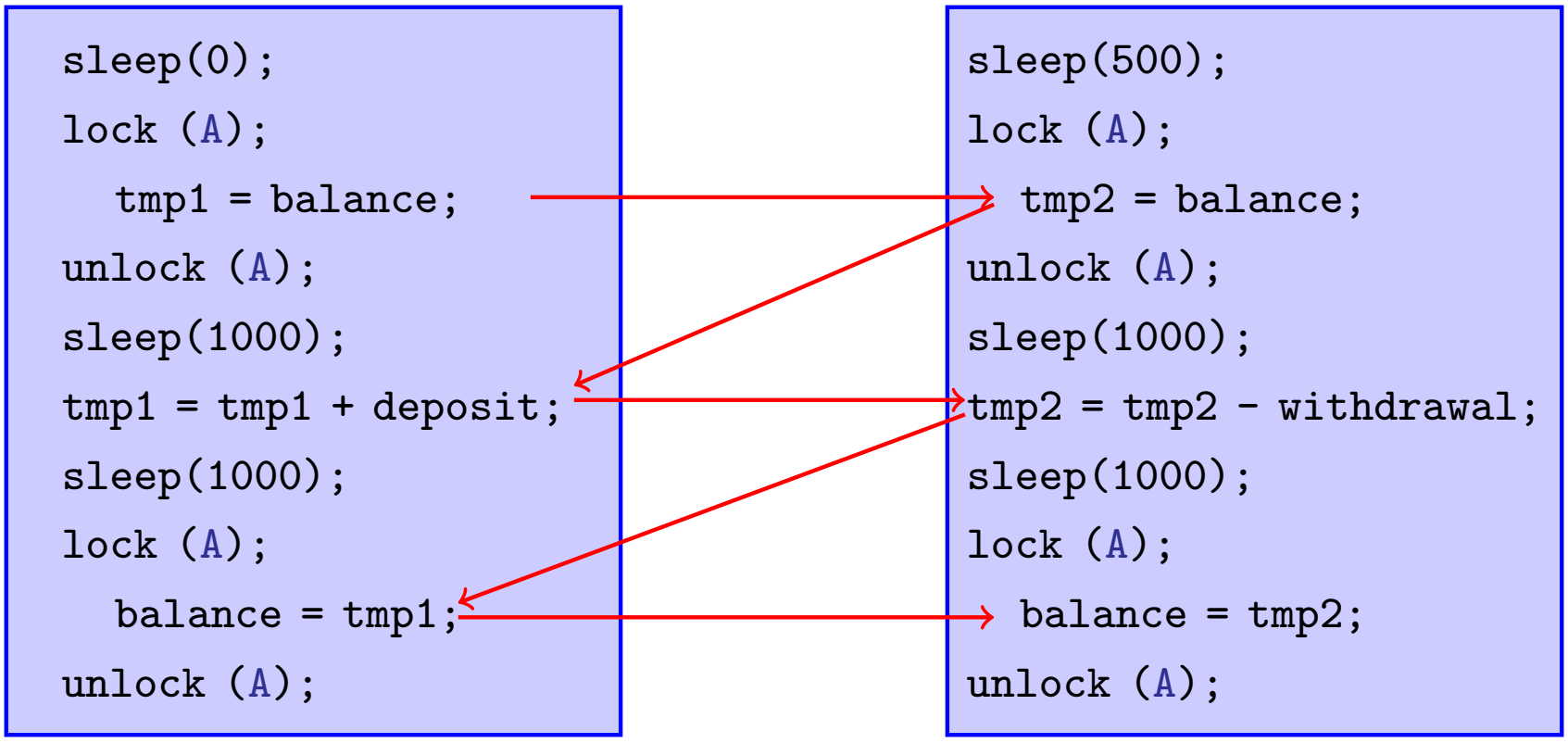
"Poor man's" strategies:

- Stress testing
  - Increase number of threads
  - Run program/system with heavy computational load

Reproducing Heisenbugs?

"Poor man's" strategies:

- Stress testing
  - Increase number of threads
  - Run program/system with heavy computational load
- Change schedule by adding `sleep` statements



Reproducing Heisenbugs?

Systematic approaches:

- "Take over" the scheduler
  - Can be achieved by instrumenting synchronization primitives

```
instr_lock(A) {
    ask scheduler to:
        perform previously unexplored context switch
        acquire lock
}
```

  - Implemented in
    - CHESS: `http://research.microsoft.com/en-us/projects/chess/`
    - INSPECT: `http://formalverification.cs.utah.edu/Inspect/`

Reproducing Heisenbugs?

Systematic approaches:

- "Take over" the scheduler
  - Can be achieved by instrumenting synchronization primitives

```
instr_lock(A) {
    ask scheduler to:
        perform previously unexplored context switch
        acquire lock
}
```

  - Implemented in
    - CHESS: `http://research.microsoft.com/en-us/projects/chess/`
    - INSPECT: `http://formalverification.cs.utah.edu/Inspect/`

Disadvantage of all techniques considered so far:

- Probe effect (bugs may be masked)

Reproducing Heisenbugs?

Systematic approaches:

- Model Checking
  - Exhaustively explore all possible program interleavings
  - Let's look at a (familiar) example!

Assertions and Concurrency: Solution

```
flagA = 0;
lock (A);
  flagA = 1;
  assert (!flagB);
  lock (B);
  flagA = 0;
  unlock (B);
unlock (A);
```

```
flagB = 0;
lock (B);
  flagB = 1;
  assert (!flagA);
  lock (A);
  flagB = 0;
  unlock (A);
unlock (B);
```

- Add assertions that fail *if and only if* a deadlock is about to occur!

Note:

- If `flagA` and `flagB` are reset after the inner locks are released, then there's a potential assertion failure even if the deadlock doesn't happen

The SPIN Model Checker

`http://spinroot.com/spin/whatispin.html`

- Open Source verification tool
- Targets verification of multi-threaded software
- Modelling language PROMELA

SPIN Model of Solution

```
int flagA = 0; int flagB = 0;
int lockA = 0; int lockB = 0;
int histA = 1; int histB = 1;
```

- Locks `lockA` and `lockB` are modelled using Boolean values
- `histA` and `histB` are set to 0 if assertion fails

SPIN Model of Solution

```
active proctype A() {
  flagA = 0;
  atomic { lockA == 0; lockA == 1;}
  flagA = 1;
  histA = (!flagB);
  if
    :: atomic { lockB == 0; lockB = 1; }
    :: timeout -> assert(!histA || !histB); goto end;
  flagA = 0;
  lockB = 0;
  lockA = 0;

  assert(histA && histB);
end:
}
```

SPIN Model of Solution

```
active proctype B() {
  flagB = 0;
  atomic { lockB == 0; lockB == 1;}
  flagB = 1;
  histB = (!flagA);
  if
    :: atomic { lockA == 0; lockA = 1; }
    :: timeout -> assert(!histA || !histB); goto end;
  flagB = 0;
  lockA = 0;
  lockB = 0;

  assert(histA && histB);
end:
}
```

Model Checking with SPIN

- Run SPIN:

```
spin -a model.pml
gcc -o pan pan.c
./pan
```

- SPIN checks all possible executions
- Condition at beginning of atomic block “waits” until it’s true
- The condition `timeout` is true if a deadlock happens
- If no assertion in SPIN model fails, then solution is correct
- If there is a counterexample, `./pan -r` reports it

Explaining Heisenbugs

- Once we've reproduced the Heisenbug, we need to explain it!
- Recall:

1. programmer introduces a **fault** in the code

Explaining Heisenbugs

- Once we've reproduced the Heisenbug, we need to explain it!
- Recall:

1. programmer introduces a **fault** in the code
2. fault gets excited during execution, results in **error**

Explaining Heisenbugs

- Once we've reproduced the Heisenbug, we need to explain it!
- Recall:

1. programmer introduces a **fault** in the code
2. fault gets excited during execution, results in **error**
3. error propagates, results in system **failure**

Explaining Heisenbugs

We want to identify

- **Fault**
- **Error**:
- **Failure**:

Explaining Heisenbugs

We want to identify

- **Fault**
- **Error**:
- **Failure**: Determined by assertion failure/failed test case

Explaining Heisenbugs

We want to identify

- **Fault** ?
- **Error**: ?
- **Failure**: Determined by assertion failure/failed test case

Explaining Heisenbugs

We want to identify

- **Fault** ?
- **Error**: ?
- **Failure**: Determined by assertion failure/failed test case

“Poor man’s” strategy:

- Debugger (e.g., `gdb`): reproduction challenging/context switches performed manually
- `printf`-debugging: instrument program with logging statements, then analyze/narrow down problem manually

Explaining Heisenbugs

Systematic approaches (incomplete list)

- Run-time monitoring:
  - Try to identify “problematic” access patterns (e.g. race conditions) during run-time
  - Disadvantage: relies on patterns (might be incomplete), false positives

Explaining Heisenbugs

Systematic approaches (incomplete list)

- Run-time monitoring:
  - Try to identify “problematic” access patterns (e.g. race conditions) during run-time
  - Disadvantage: relies on patterns (might be incomplete), false positives
- Trace analysis/data mining:
  - Record several failing and passing traces
  - Report R/W combinations frequently occurring in bad traces, but not in good ones
  - Disadvantage: requires several traces, may report false positives

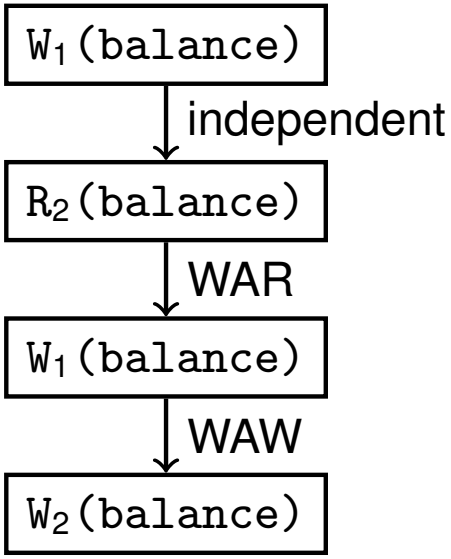
Explaining Heisenbugs

Systematic approaches (incomplete list)

- Run-time monitoring:
  - Try to identify “problematic” access patterns (e.g. race conditions) during run-time
  - Disadvantage: relies on patterns (might be incomplete), false positives
- Trace analysis/data mining:
  - Record several failing and passing traces
  - Report R/W combinations frequently occurring in bad traces, but not in good ones
  - Disadvantage: requires several traces, may report false positives
- Slicing:
  - Perform *symbolic* analysis of execution trace
  - *Slice* away statements irrelevant for assertion failure
  - Disadvantage: sliced traces may still be long

Problematic Access Patterns

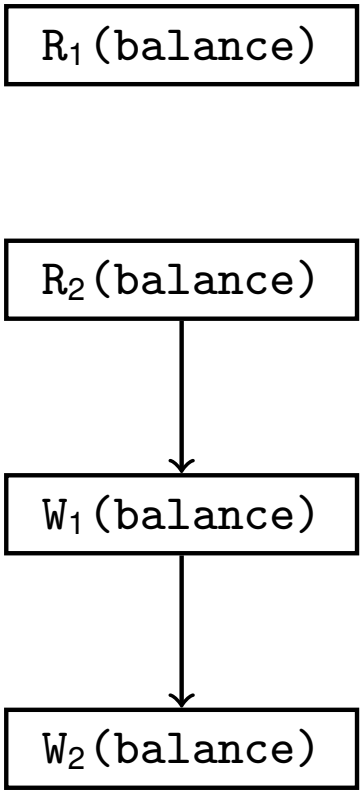
- Recall: Data dependencies



- Access pattern can be mapped back to program

Problematic Access Patterns

- Access pattern can be mapped back to program
  - Locks only shown for context




```
lock (A);
  tmp1 = balance;
unlock (A);
```

```
lock (A);
  tmp2 = balance;
unlock (A);
```

```
lock (A);
  balance = tmp1;
unlock (A);
```

```
lock (A);
  balance = tmp2;
unlock (A);
```

Problematic Access Patterns

Explanation derived from pattern:

- **Fault**: Update of balance is not performed atomically
- **Error**: Value of `balance` written by Thread 2 is “stale”
- **Failure**: Balance on account deviates from expected value

Problematic Access Patterns

Explanation derived from pattern:

- **Fault**: Update of balance is not performed atomically
- **Error**: Value of `balance` written by Thread 2 is “stale”
- **Failure**: Balance on account deviates from expected value

Still requires intuition, but pattern “explanation” helps.

- For more complex programs, however, access pattern may contain variables not directly related to bug!

Sequential Slicing

- Debugging technique that cuts away irrelevant code
- Backwards, starting from the assertion
  - Eliminate statements where there's no flow dependency
  - (For sequential setting)

```
x = 42;

y = 15;

z = y + 5;

assert (z ≤ 10);
```

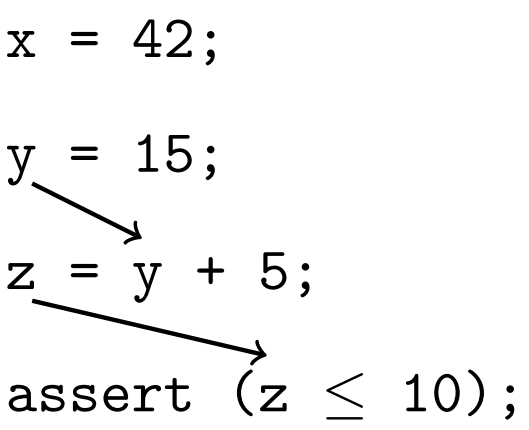
Sequential Slicing

- Debugging technique that cuts away irrelevant code
- Backwards, starting from the assertion
  - Eliminate statements where there's no flow dependency
  - (For sequential setting)

```
x = 42;

y = 15;

z = y + 5;

assert (z ≤ 10);
```

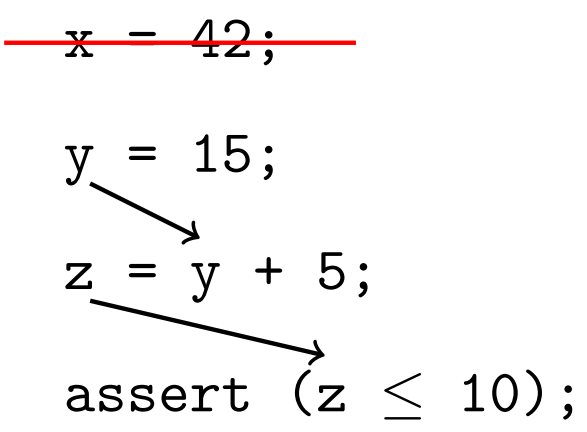
Sequential Slicing

- Debugging technique that cuts away irrelevant code
- Backwards, starting from the assertion
  - Eliminate statements where there's no flow dependency
  - (For sequential setting)



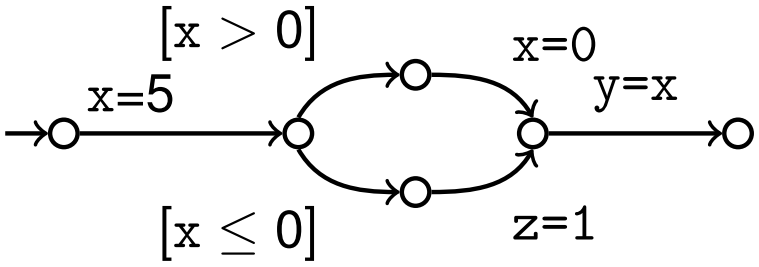
Sequential Slicing

- Debugging technique that cuts away irrelevant code
- Backwards, starting from the assertion
  - Eliminate statements where there's no flow dependency
  - (For sequential setting)

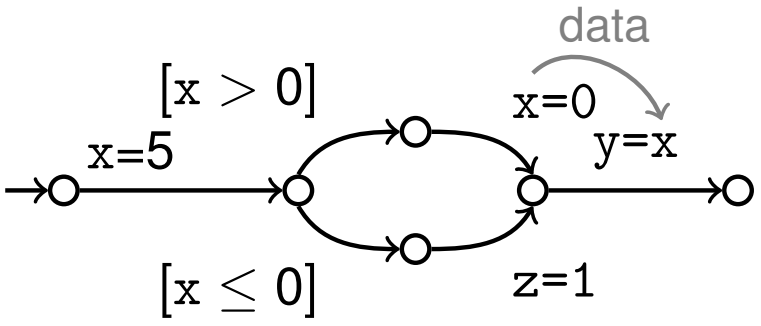


- Assignment `x = 42` has no impact on assertion

Sequential Slicing and Control-Dependency

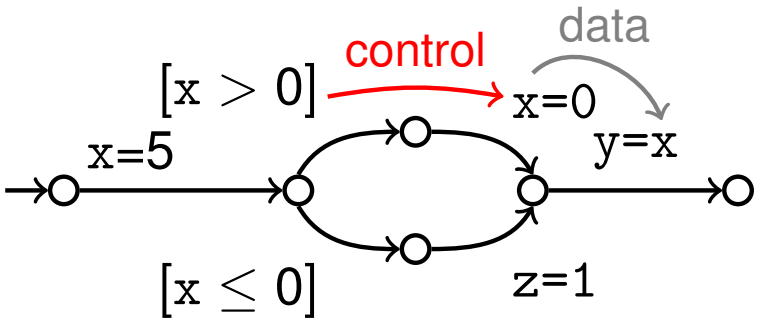


Sequential Slicing and Control-Dependency



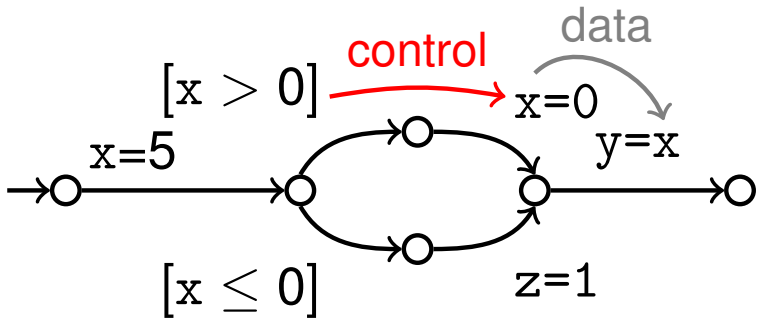
- y=x data-depends on x=0 (but not on z=1)

Sequential Slicing and Control-Dependency



- `y=x` data-depends on `x=0` (but not on `z=1`)
- `x=0` is only executed if (`x` > 0)-branch is taken
  - Therefore, branching condition must be included in slice
  - “control-flow sensitive slice”

Sequential Slicing and Control-Dependency



- `y=x` data-depends on `x=0` (but not on `z=1`)
- `x=0` is only executed if $(x > 0)$-branch is taken
  - Therefore, branching condition must be included in slice
  - "control-flow sensitive slice"
- $(x > 0)$ data-depends on `x=5`, therefore `x=5` must be included

Semantic Slicing

- “Syntactic” slicing results in large slices
  - many statements included
  - doesn’t take semantics of statements into account

```
x = 5;

y = 15;

z = y % x;

assert (z ≥ 5);
```

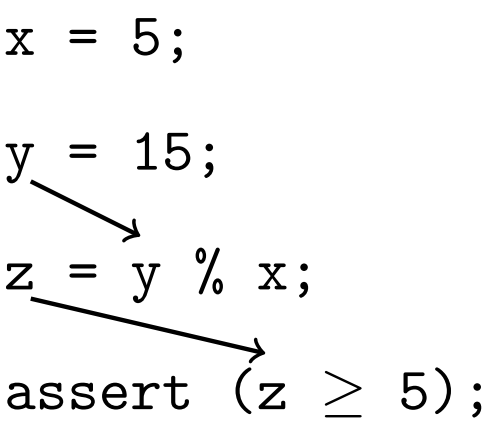
Semantic Slicing

- “Syntactic” slicing results in large slices
  - many statements included
  - doesn’t take semantics of statements into account

```
x = 5;

y = 15;

z = y % x;

assert (z ≥ 5);
```

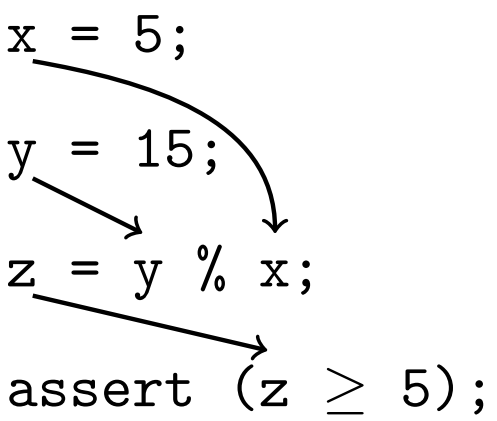
Semantic Slicing

- “Syntactic” slicing results in large slices
  - many statements included
  - doesn’t take semantics of statements into account



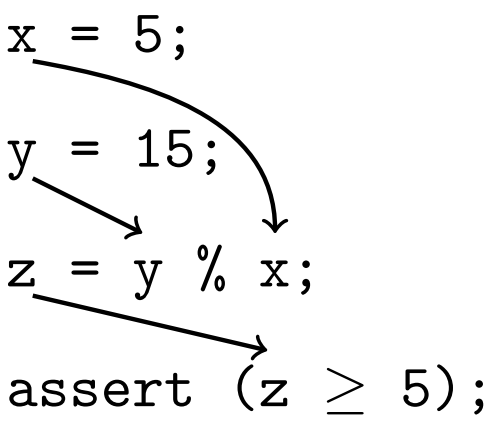
Semantic Slicing

- “Syntactic” slicing results in large slices
  - many statements included
  - doesn’t take semantics of statements into account



Semantic Slicing

- “Syntactic” slicing results in large slices
  - many statements included
  - doesn’t take semantics of statements into account



- But `y=15` is *irrelevant*, since $(y\%5) < 5$ independently of `y`!

Semantic Slicing

- Hoare Logic to the rescue!
- Construct Hoare Proof showing that assertion fails
  - Execution traces do not contain loops
  - Can be automated (using SMT solvers and *Craig interpolation*)

```
x = 5;

y = 15;

z = y % x;

assert (z ≥ 5);
```

Semantic Slicing

- Hoare Logic to the rescue!
- Construct Hoare Proof showing that assertion fails
  - Execution traces do not contain loops
  - Can be automated (using SMT solvers and *Craig interpolation*)

```
x = 5;

y = 15;

z = y % x;
{z < 5}
assert (z ≥ 5);
```

Semantic Slicing

- Hoare Logic to the rescue!
- Construct Hoare Proof showing that assertion fails
  - Execution traces do not contain loops
  - Can be automated (using SMT solvers and *Craig interpolation*)

```
x = 5;

y = 15;
{x ≤ 5}
z = y % x;
{z < 5}
assert (z ≥ 5);
```

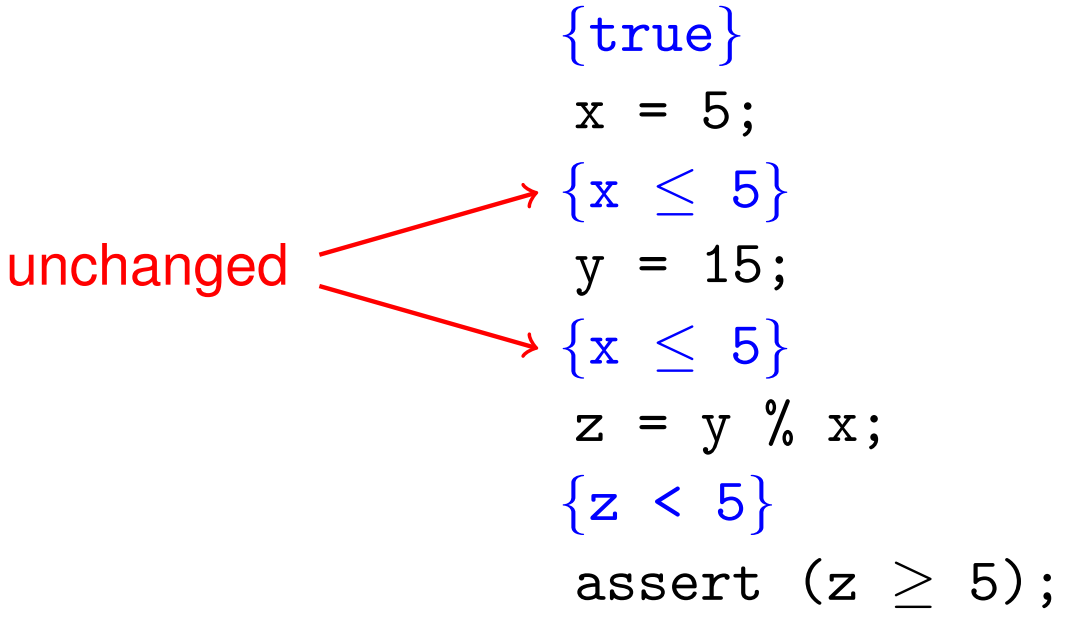
Semantic Slicing

- Hoare Logic to the rescue!
- Construct Hoare Proof showing that assertion fails
  - Execution traces do not contain loops
  - Can be automated (using SMT solvers and *Craig interpolation*)

```
{true}
x = 5;
{x ≤ 5}
y = 15;
{x ≤ 5}
z = y % x;
{z < 5}
assert (z ≥ 5);
```

Semantic Slicing

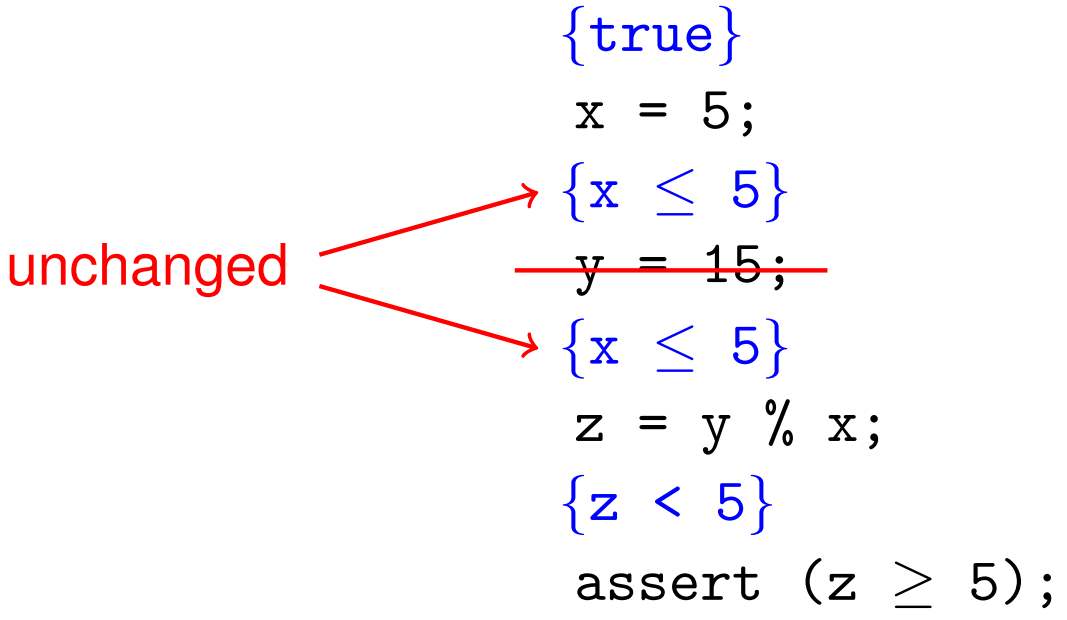
- Hoare Logic to the rescue!
- Construct Hoare Proof showing that assertion fails
  - Execution traces do not contain loops
  - Can be automated (using SMT solvers and *Craig interpolation*)



- `y=15` does not affect surrounding Hoare assertions

Semantic Slicing

- Hoare Logic to the rescue!
- Construct Hoare Proof showing that assertion fails
  - Execution traces do not contain loops
  - Can be automated (using SMT solvers and *Craig interpolation*)



- `y=15` does not affect surrounding Hoare assertions
- Therefore, `y=15` is irrelevant

Semantic Slicing

- Semantic slicing can eliminate irrelevant statements
- Hoare assertions act as annotation, aid understanding

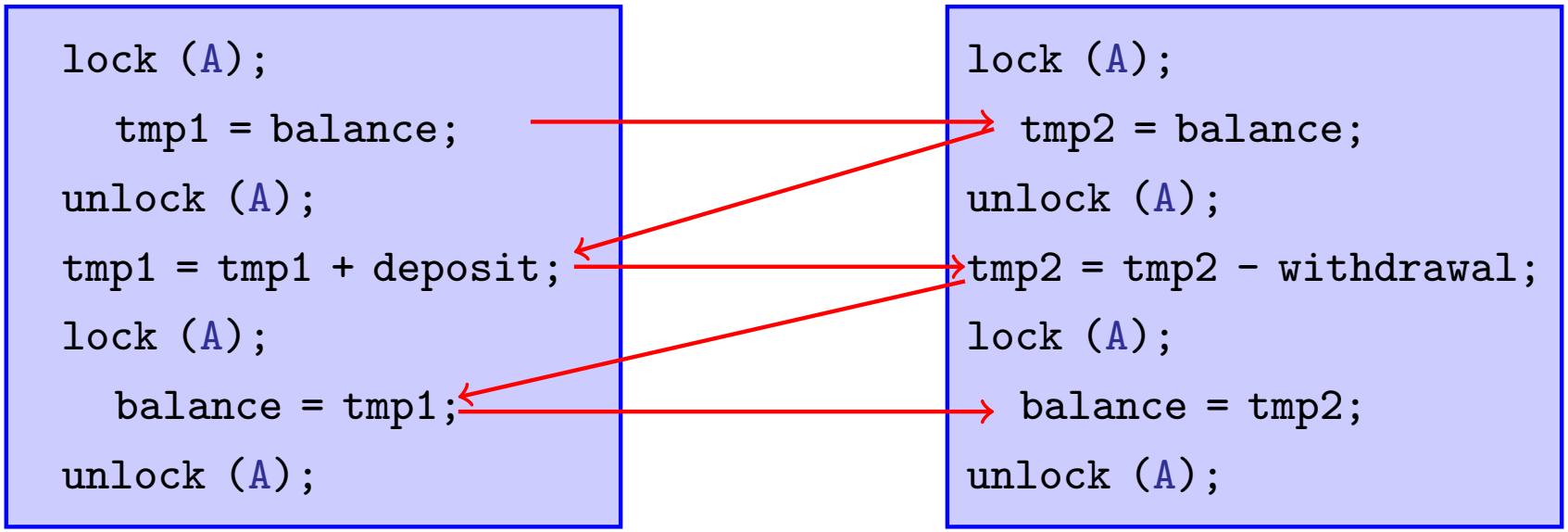
```
{true}
x = 5;
{x ≤ 5}
...
{x ≤ 5}
z = y % x;
{z < 5}
assert (z ≥ 5);
```

Concurrent Slicing (First Attempt)

let's apply sequential slicing to concurrent trace.

- Ignore locks for the time being

```
old_balance = balance;
```




```
assert (balance == old_balance + deposit - withdrawal);
```

Concurrent Slicing (First Attempt)

```
tmp1 = balance;
                        tmp2 = balance;
tmp1 = tmp1 + deposit;
                        tmp2 = tmp2 - withdrawal;
balance = tmp1;
                        balance = tmp2;
assert (balance == old_balance + deposit - withdrawal);
```

Concurrent Slicing (First Attempt)

```
tmp1 = balance;
                        tmp2 = balance;
tmp1 = tmp1 + deposit;
                        tmp2 = tmp2 - withdrawal;
balance = tmp1;
                        balance = tmp2;
{ balance == old_balance - withdrawal }
assert (balance == old_balance + deposit - withdrawal);
```

Concurrent Slicing (First Attempt)

```
                tmp1 = balance;
                                        tmp2 = balance;
                tmp1 = tmp1 + deposit;
                                        tmp2 = tmp2 - withdrawal;
                balance = tmp1;
               { tmp2 == old_balance - withdrawal }
                                        balance = tmp2;
             { balance == old_balance - withdrawal }
assert (balance == old_balance + deposit - withdrawal);
```

Concurrent Slicing (First Attempt)

```
                tmp1 = balance;
                                        tmp2 = balance;
                tmp1 = tmp1 + deposit;
                                        tmp2 = tmp2 - withdrawal;
              { tmp2 == old_balance - withdrawal }
              balance = tmp1;
              { tmp2 == old_balance - withdrawal }
                                        balance = tmp2;
            { balance == old_balance - withdrawal }
assert (balance == old_balance + deposit - withdrawal);
```

Concurrent Slicing (First Attempt)

```
                tmp1 = balance;
                                        tmp2 = balance;
                tmp1 = tmp1 + deposit;
                      { tmp2 == old_balance }
                                        tmp2 = tmp2 - withdrawal;
               { tmp2 == old_balance - withdrawal }
               balance = tmp1;
               { tmp2 == old_balance - withdrawal }
                                        balance = tmp2;
             { balance == old_balance - withdrawal }
assert (balance == old_balance + deposit - withdrawal);
```

Concurrent Slicing (First Attempt)

```
                tmp1 = balance;

                                    tmp2 = balance;
                      { tmp2 == old_balance }
               tmp1 = tmp1 + deposit;
                      { tmp2 == old_balance }
                                    tmp2 = tmp2 - withdrawal;
              { tmp2 == old_balance - withdrawal }
              balance = tmp1;
              { tmp2 == old_balance - withdrawal }
                                    balance = tmp2;
            { balance == old_balance - withdrawal }
assert (balance == old_balance + deposit - withdrawal);
```

Concurrent Slicing (First Attempt)

```
                tmp1 = balance;

                                    tmp2 = balance;
                      { tmp2 == old_balance }
               tmp1 = tmp1 + deposit;
                      { tmp2 == old_balance }
                                    tmp2 = tmp2 - withdrawal;
              { tmp2 == old_balance - withdrawal }
              balance = tmp1;
              { tmp2 == old_balance - withdrawal }
                                    balance = tmp2;
            { balance == old_balance - withdrawal }
assert (balance == old_balance + deposit - withdrawal);
```

Concurrent Slicing (First Attempt)

```
                tmp1 = balance;
                    { balance == old_balance }
                                tmp2 = balance;
                     { tmp2 == old_balance }
               tmp1 = tmp1 + deposit;
                     { tmp2 == old_balance }
                                tmp2 = tmp2 - withdrawal;
              { tmp2 == old_balance - withdrawal }
              balance = tmp1;
              { tmp2 == old_balance - withdrawal }
                                balance = tmp2;
            { balance == old_balance - withdrawal }
assert (balance == old_balance + deposit - withdrawal);
```

Concurrent Slicing (First Attempt)

```
                    { balance == old_balance }
          tmp1 = balance;
                    { balance == old_balance }
                                  tmp2 = balance;
                     { tmp2 == old_balance }
          tmp1 = tmp1 + deposit;
                     { tmp2 == old_balance }
                                  tmp2 = tmp2 - withdrawal;
          { tmp2 == old_balance - withdrawal }
          balance = tmp1;
          { tmp2 == old_balance - withdrawal }
                                  balance = tmp2;
        { balance == old_balance - withdrawal }
assert (balance == old_balance + deposit - withdrawal);
```

Concurrent Slicing (First Attempt)

- Thread 1 sliced away *entirely*
- Slice doesn't reflect concurrency bug (atomicity violation)!

Concurrent Slicing (First Attempt)

- Thread 1 sliced away *entirely*
- Slice doesn't reflect concurrency bug (atomicity violation)!
- Problem: Sequential slicing ignores *data hazards*

Concurrent Slicing (First Attempt)

- Thread 1 sliced away *entirely*
- Slice doesn't reflect concurrency bug (atomicity violation)!
- Problem: Sequential slicing ignores *data hazards*
- Solution: Consider data hazards during slicing

Concurrent Slicing (With Data Hazards)

```
                    { balance == old_balance }
          tmp1 = balance;
                    { balance == old_balance }
                                  tmp2 = balance;
                     { tmp2 == old_balance }
          tmp1 = tmp1 + deposit;
                     { tmp2 == old_balance }
                                  tmp2 = tmp2 - withdrawal;
          { tmp2 == old_balance - withdrawal }
          balance = tmp1;
          { tmp2 == old_balance - withdrawal }
                                  balance = tmp2;
        { balance == old_balance - withdrawal }
assert (balance == old_balance + deposit - withdrawal);
```

Concurrent Slicing (With Data Hazards)

```
                    { balance == old_balance }
tmp1 = balance;
                    { balance == old_balance }
                                  tmp2 = balance;
                     { tmp2 == old_balance }
tmp1 = tmp1 + deposit;
                     { tmp2 == old_balance }
                                  tmp2 = tmp2 - withdrawal;
                  { tmp2 == old_balance - withdrawal }
                   balance = tmp1;
                  { tmp2 == old_balance - withdrawal }
                            WAW ──────────→ balance = tmp2;
                { balance == old_balance - withdrawal }
assert (balance == old_balance + deposit - withdrawal);
```

Concurrent Slicing (With Data Hazards)

- `tmp1 = tmp1 + deposit` not included, since eliminated by *semantic* slicing
- Hoare assertions show that value of `balance` is stale
- Disadvantage: slice contains $>$ 50% of program
  - This ratio is better for larger programs

Concurrent Slices as Explanations

Error explanation identifies:

- **Fault** – derived from statements in slice
- **Error** – reflected by Hoare assertions
- **Failure** – determined given by assertion/specification

A More Involved Example

```
26 int withdraw(int amount)
27 {
28   int tmpBalance;
29   int applied = 0;

31   pthread_mutex_lock(balance_lock);
32   tmpBalance = balance;
33   pthread_mutex_unlock(balance_lock);

35   if(tmpBalance - amount >= MIN)
36   {
37     tmpBalance -= amount;
38     applied = 1;
39   }

41   pthread_mutex_lock(balance_lock);
42   balance = tmpBalance;
43   pthread_mutex_unlock(balance_lock);

45   return applied;
46 }
```

```
48 int deposit(int amount)
49 {
50   int tmpBalance;
51   int applied = 0;

53   pthread_mutex_lock(balance_lock);
54   tmpBalance = balance;
55   pthread_mutex_unlock(balance_lock);

57   if(tmpBalance + amount <= MAX)
58   {
59     tmpBalance += amount;
60     applied = 1;
61   }

63   pthread_mutex_lock(balance_lock);
64   balance = tmpBalance;
65   pthread_mutex_unlock(balance_lock);

67   return applied;
68 }
```

A More Involved Example

```
108 int sum_thread1 = 0; // shared vars
109 int sum_thread2 = 0; // shared vars

111 void* thread1(void *np) {
112   transactions * t;
113   t = (transactions*)np;
114   int upper_bound = t->num / 2;
115   int i;
116   for(i = 0; i < upper_bound; i++){
117     if (do_transaction(&(t->t_array[i
            ]))) {
118       if (t->t_array[i].type ==
              DEPOSIT) {
119         sum_thread1 += t->t_array[i].
              amount;
120       }
121       else {
122         sum_thread1 -= t->t_array[i].
              amount;
123       }
124     }
125   }

127   pthread_exit(NULL);
128   return NULL;
129 }
```

```
131 void* thread2(void *np) {
132   transactions * t;
133   t = (transactions*)np;
134   int upper_bound = t->num / 2;
135   int i;
136   for(i = upper_bound; i < t->num; i
          ++){
137     if (do_transaction(&(t->t_array[i
            ]))) {
138       if (t->t_array[i].type ==
              DEPOSIT) {
139         sum_thread2 += t->t_array[i].
              amount;
140       }
141       else {
142         sum_thread2 -= t->t_array[i].
              amount;
143       }
144     }
145   }

147   pthread_exit(NULL);
148   return NULL;
149 }
```

A More Involved Example

```
151 void unit_test() {
152   transactions * trs = new_transactions(5);
153   balance_lock = (pthread_mutex_t *) malloc(sizeof(pthread_mutex_t));
154   pthread_mutex_init(balance_lock,NULL);

156   balance = 40;

158   int orgBalance = balance;

160   pthread_t t1, t2;
161   pthread_create(&t1,NULL,thread1,(void*)trs);
162   pthread_create(&t2,NULL,thread2,(void*)trs);

164   pthread_join(t1,NULL);
165   pthread_join(t2,NULL);

167   int expBalance = orgBalance + sum_thread1 + sum_thread2;
168   assert(expBalance == balance);
169 }
```

A More Involved Example: Bug Explanation Pattern

Bug Explanation Pattern R: read, W:Write Subscript of R/W: thread id	Line number Source code (T**1**)	Line number Source code (T**2**)

R_2(**t->num**)
R_2(**t->array**)
R_2(**t->array[4].type**)
R_2(**t->array[4].amount**)
R_2(**balance**)
		136 for(i = upper_bound; i < **t->num**; i++){
		137 if (do_transaction(&(**t->t_array**[i]))) {
		74 if (**t->type** == DEPOSIT)
		76 applied = deposit(**t->amount**);
		54 tmpBalance = **balance**;

W_2(**balance**)		64 **balance** = tmpBalance;
W_1(**balance**)	64 **balance** = tmpBalance;	

Summary

- Heisenbugs are hard to reproduce
  - caused by lack of control over schedule
  - excessive instrumentation can *hide* bugs (probe effect)
- Goal of error explanation is to identify:
  - **Fault**
  - **Error**

  for a given **Failure** (determined by assertion/specification)

Bonus Exercise

June 11, 2pm-4pm in HS17

- Task: 3 concurrency bugs to explain
- Incentive: 15 additional points (count towards grade)
- Sign up via TUWEL until June 10, 4pm!