COMP5111: Fundamentals of Software Analysis

Concurrency Bug Detection (Supplementary Materials)

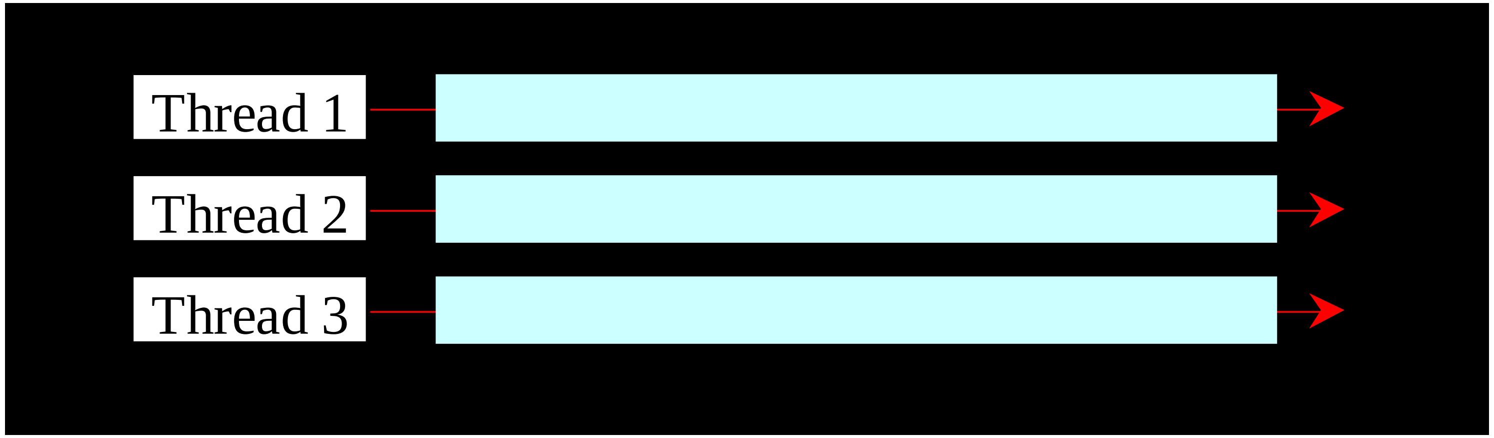
Shing-Chi Cheung

Computer Science & Engineering

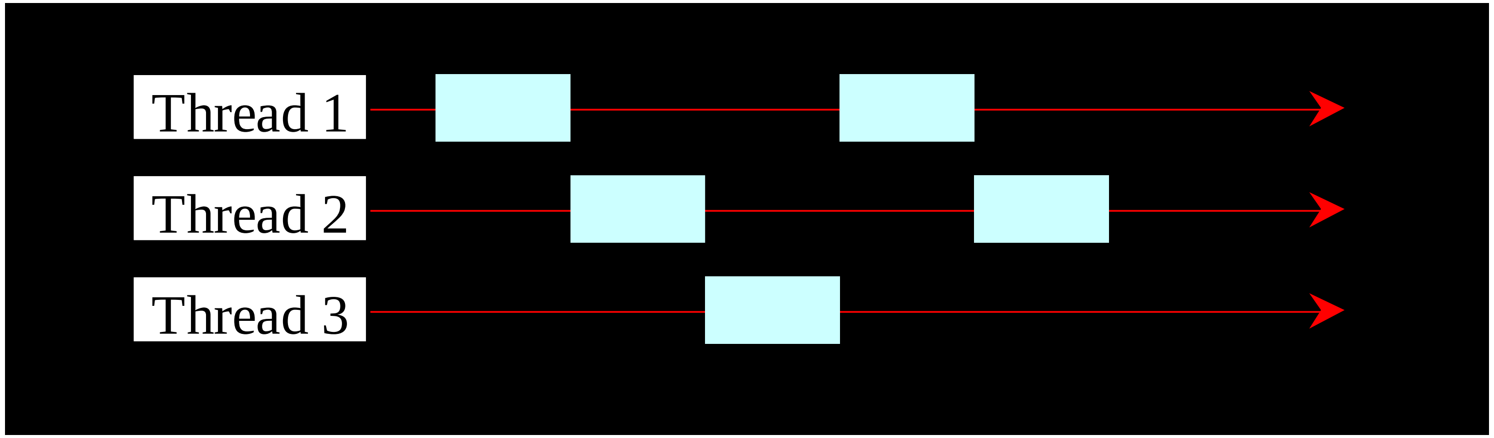
HKUST

Concurrency

Multiple threads on multiple CPUs or Cores



Multiple threads sharing a single CPU



Data Race

T1:

1 local_X = G ;

2 local_X++;

3 G = local_X;

T2:

4 local_Y = G ;

5 local_Y++;

6 G = local_Y;

Initial: G=0 → **G is 2**

Data Race

	T1:	T2:
G=0	1 local_X = G ;	
	2 local_X++;	
G=1	3 G = local_X;	
		4 local_Y = G ;
		5 local_Y++;
G=2		6 G = local_Y;

- Initial: G=0 ⟹ **G is 2**

Data Race

T1:

T2:

Output is no longer deterministic

1 local_X = G ;

G=0

2 local_X++;

4 local_Y = G ;

3 G = local_X;

G=1

5 local_Y++;

G=1

6 G =local_Y;

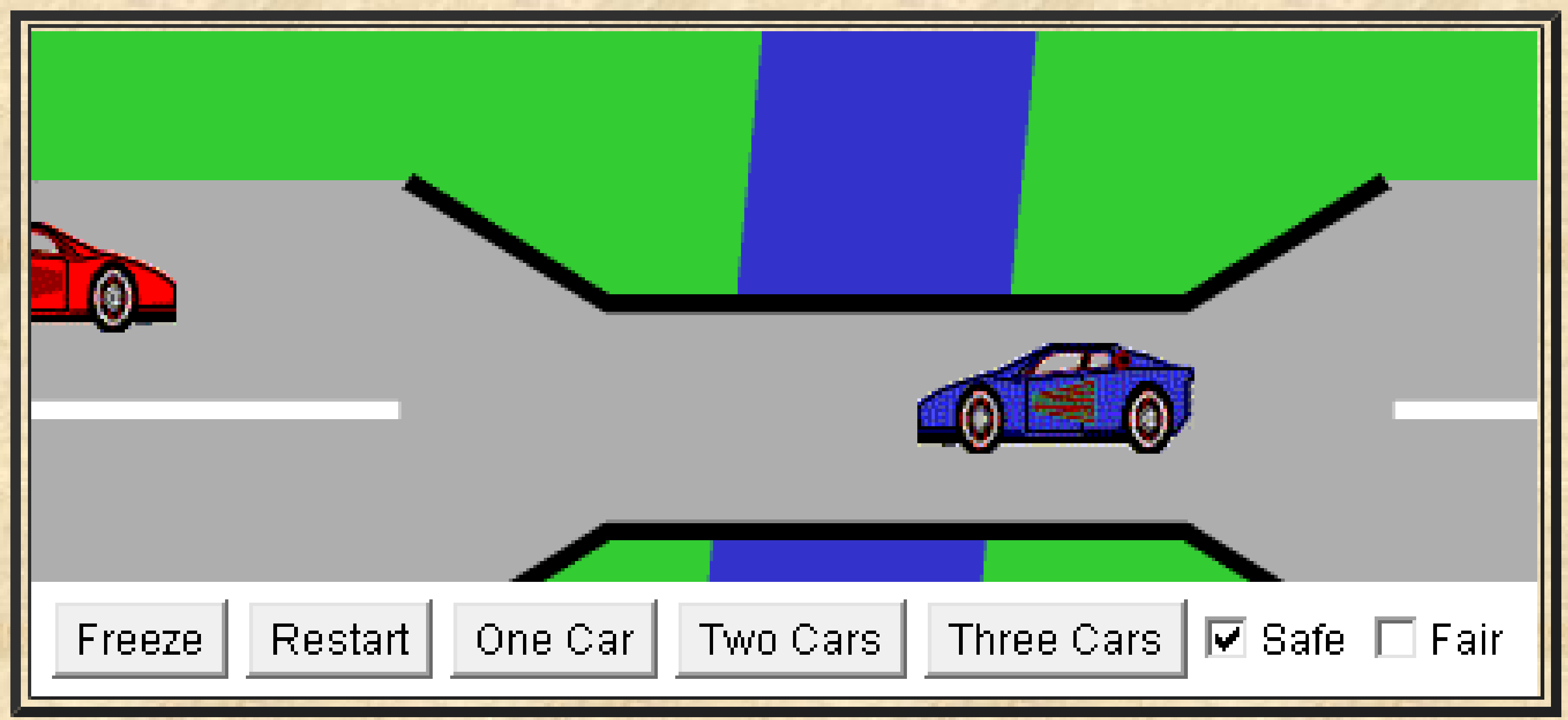
- Initial: G=0 ⇒ **G is 1**

Safety vs. Liveness

- General defintion
  - Safety: something “bad” will never happen
    - No two trains can travel with less than 50m apart.
    - The program must not violate assertions.
  - Liveness: something “good” will happen (but we don’t know when)
    - Each train will eventually reach its destination.
    - The program will eventually terminate.

Safety vs Liveness

- Safety?
- Liveness?
- Fairness?



http://www.cse.ust.hk/~scc/teaching/SingleLaneBridge.html

Safety vs. Liveness

- T1:
  1 local_X= G ;
  2 local_X++;
  3 G =local_X;
- T2:
  4 local_Y= G ;
  5 local_Y++;
  6 G =local_Y;

- Initial: G=0

G is 2

- For concurrent programs
  - Safety: read/write operations match programmer's expectation
  - Liveness: program will make progress (preferably faster)
- The rest would be a constant battle for matching programmers' expectation and making programs run faster

Operation Issues (Races and locks)

Quality Issues

- Data race/Atomicity violation
  - Program can progress
  - Results inconsistent with programmer's expectation
  - Violate safety property
- Deadlock
  - All threads cannot progress
  - Violate liveness property
- Livelock
  - Some threads may progress but the program oscillates over a number of states, not making any meaningful progress
  - Violate liveness and fairness property

Data Race

- T1:
  1 local_X= G ;
  2 local_X++;
  3 G =local_X;
- T2:
  4 local_Y= G ;
  5 local_Y++;

- Initial: G=0

G is 2

- A race condition occurs if two threads access a shared variable concurrently without synchronization, and at least one access is a write
- It is caused by non-atomic (inconsistent) execution of programmer's intent
- Types
  - Low-level data race (Language/platform deficiency)
  - High-level data race (Program semantics)

Data Race (Low Level)

```
class Ref {
  int t;
  void set (int i) {
     t = i;
  }
}

Ref x = new Ref(0);
parallel { // two calls happen in parallel
    x.set(2147483647);
    x.set(0);
}
assert x.i == 0?
assert x.i == 2147483647?
assert x.i == 2147418112?
assert x.i == 65535?
```

- Programmer's intent – Execute the instruction in a non-divisible way
- In low end devices, a 32-bit integer occupies two words on 16-bit machines
- A write operations is broken down to two instructions:

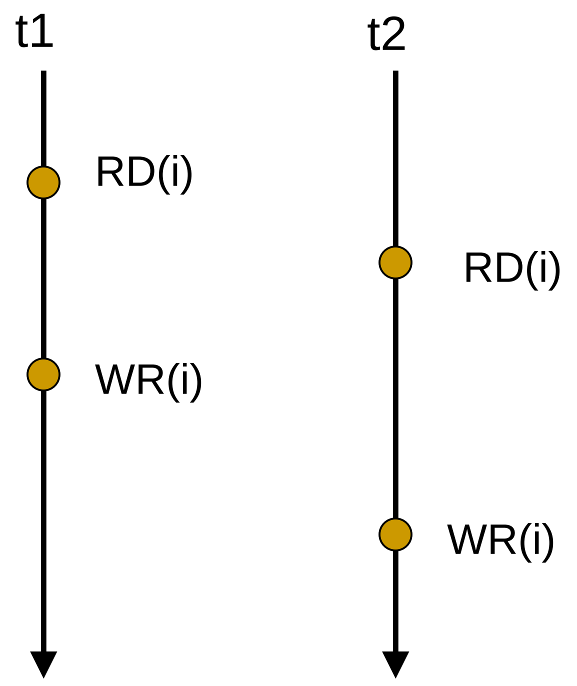
  2147483647 = 0x7FFF FFFF
  2147418112 = 0x7FFF 0000
  65535 = 0x0000 FFFF
  0 = 0x0000 0000
- Memory sees:

  Most significant half: 0x7FFF or 0x0000
  Least significant half: FFFF or 0000

Data Race (High Level)

```
class Ref {
  int i;
  void inc() {
     int t = i + 1;  // RD(i)
     i = t;          // WD(i)
  }
}

Ref x = new Ref(0);
parallel {
    x.inc();     // two calls happen
    x.inc();     // in parallel
}
assert x.i == 2;
```

A data race inducing schedule



A group of program statements that must be executed in a non-divisible manner

The synchronized keyword

```
class Ref {
  int i;
  void inc() {
    int t = i + 1;
    i = t;
  }                Critical region
}

Ref x = new Ref(0);
parallel {
    x.inc();    // two calls happen
    x.inc();    // in parallel
}
assert x.i == 2;
```

Critical Region

- To avoid race conditions, at most one thread is allowed to enter a certain part of the program, known as critical region.
- The critical region in the example is the entire inc() method.

The synchronized keyword

```
class Ref {
  int i;
  void inc() {
    int t = i + 1;
    i = t;
  }                 Critical region
}

Ref x = new Ref(0);
parallel {
    x.inc();    // two calls happen
    x.inc();    // in parallel
}
assert x.i == 2;
```

Locks

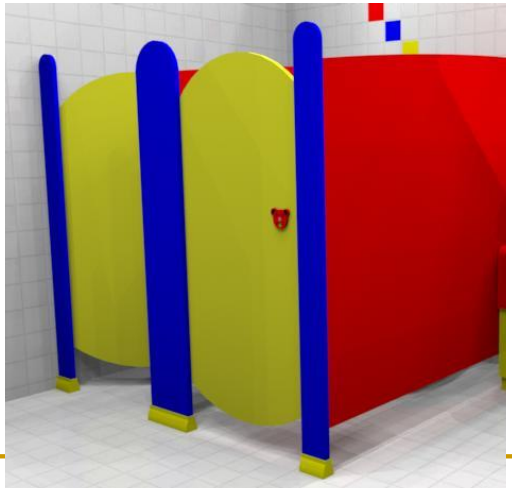
- We can use the *synchronized* keyword to synchronize the method so that only one thread can access the method at a time.
- One approach is to make Ref thread-safe by adding the synchronized keyword in the inc method as follows:

  synchronized void inc()

Critical Regions and Locks

- The concept of critical regions (CR) and locks are everywhere around us
  - Share road outside of our academic building
    - Lock → Go/Stop Sign; CR → Road
  - Shared washroom of a coffee shop
    - Lock → Physical lock; CR→ The toilet
- Every Java object is also a lock

```
synchronized(this) { … }
```



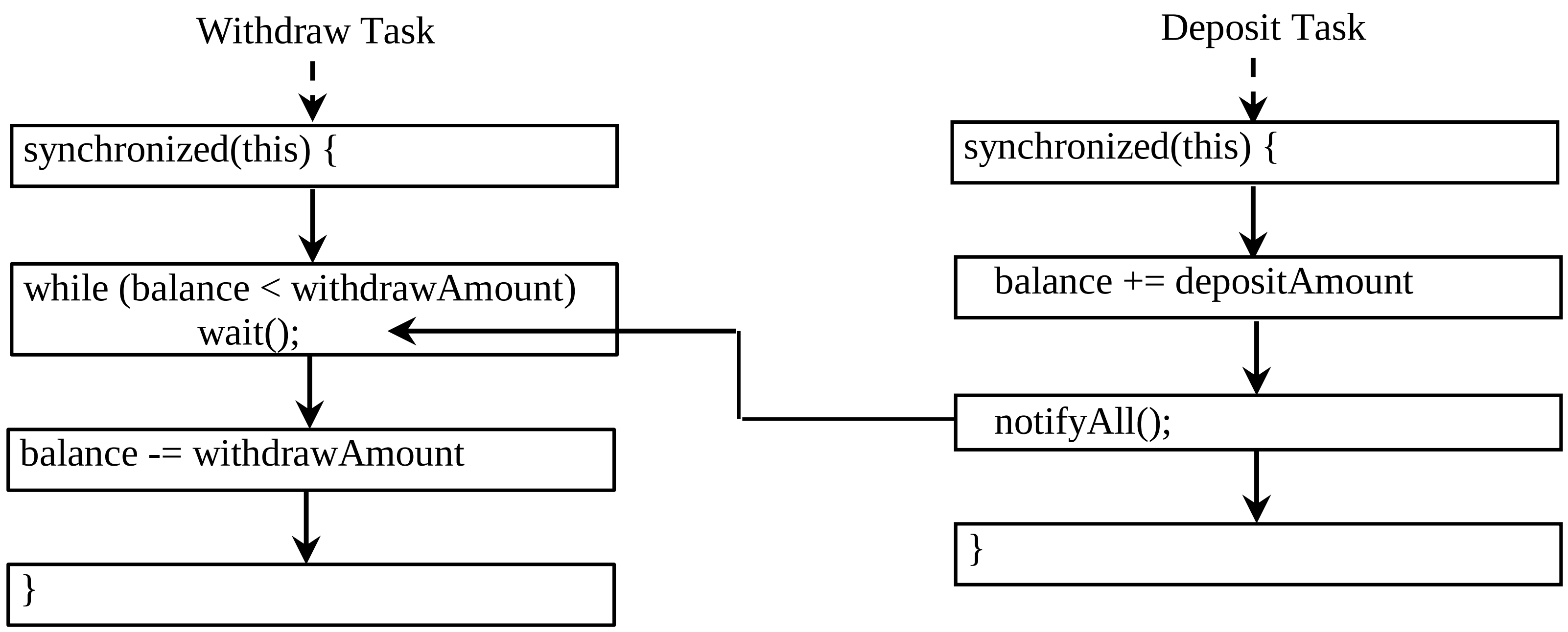
Monitor-based programming

- A thread uses “wait()” to submit our request to acquire a lock and enter a critical region.
- A thread users “notify()” or “notifyAll()” to release a lock and leave a critical region.
- A thread can only raise these three operations when executing a synchronized block.
- Otherwise, an IllegalMonitorStateException occurs.

Cooperation Among Threads

Condition: balance needs to be more than the amount withdrawn.

- Balance is less than the amount to be withdrawn, the withdraw task will wait and give up the processor.
- When the deposit task adds money to the account, the task signals the waiting withdraw task to try again.



Deadlock

- Occurs when two or more threads need to acquire the locks on several shared objects.
- Consider the scenario with two threads and two objects,
  - Thread 1 acquired a lock on object1 and Thread 2 acquired a lock on object2.
  - Now Thread 1 is waiting for the lock on object2 and Thread 2 for the lock on object1.
  - The two threads wait for each other to release the in order to get the lock, and neither can continue to run.

Thread 1

```
synchronized (object1) {
   // do something here
   synchronized (object2) {
      // do something here
   }
}
```

Wait for Thread 2 to release the lock on object2

Thread 2

```
synchronized (object2) {
   // do something here
   synchronized (object1) {
      // do something here
   }
}
```

Wait for Thread 1 to release the lock on object1

Preventing Deadlock

- Deadlock can be easily avoided by using a simple technique known as resource ordering.
  - Assign an order on all the objects whose locks must be acquired
  - Ensure that each thread acquires the locks in that order.
- Suppose the objects are ordered as object1 and then object2.
  - Thread 2 must acquire a lock on object1 first, then on object2.
  - Once Thread 1 acquired a lock on object1, Thread 2 has to wait for a lock on object1.

Thread 1

```
synchronized (object1) {
  // do something here
  synchronized (object2) {
    // do something here
  }
}
```

Do not need to wait for Thread 2 to release the lock on object2

Thread 2

```
synchronized (object1) {
  // do something here
  synchronized (object2) {
    // do something here
  }
}
```

Must acquire object 1 before acquiring object 2

Atomicity Violation (Higher level)

```
class CircularList {
  private Element[] list;
  private int size;
  …
  int synchronized getSize( ) { return size; }
  void synchronized copyAll(Element[] array) {
     …
  }
  void synchronized insert(Element m) { … }
}
```

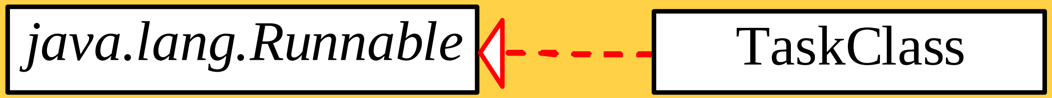
```
CircularList cl;

Thread 1:
Element[] ms = new Element[cl.getSize()];
cl.copyAll(ms);
// do other …

Thread 2:
cl.insert(new Element(0));
```

- Individual methods are safe
- Their compositions are not.
- “Composition” is a semantic thing → Determined by the client code.

- Non-deterministic results → data race
- Results inconsistent with some atomicity assumption → atomicity violation
- Atomicity violation is a form of data race

Practice: Thread Programming

Creating Tasks and Threads




```java
// Custom task class
public class TaskClass implements Runnable {
  ...
  public TaskClass(...) {
    ...
  }

  // Implement the run method in Runnable
  public void run() {
    // Tell system how to run custom thread
    ...
  }
  ...
}
```

```java
// Client class
public class Client {
  ...
  public void someMethod() {
    ...
    // Create an instance of TaskClass
    TaskClass task = new TaskClass(...);

    // Create a thread
    Thread thread = new Thread(task);

    // Start a thread
    thread.start();
    ...
  }
  ...
}
```

Monitor-based Programming

Task 1

```
synchronized (anObject) {
  try {
    // Wait for the condition to become true
    while (!condition)
      anObject.wait();

   // Do something when condition is true
  }
  catch (InterruptedException ex) {
    ex.printStackTrace();
  }
```

resume

Task 2

```
synchronized (anObject) {
  // When condition becomes true
  anObject.notify(); or  anObject.notifyAll();
  ...
}
```

- Use the wait(), notify(), and notifyAll() methods to facilitate communication among threads.
- When wait() is invoked, it pauses the thread and simultaneously releases the lock on the object. When the thread is restarted after being notified, the lock is automatically reacquired.

wait(), notify(), and notifyAll()

Task 1

```
synchronized (anObject) {
  try {
    // Wait for the condition to become true
    while (!condition)
      anObject.wait();

   // Do something when condition is true
  }
  catch (InterruptedException ex) {
    ex.printStackTrace();
  }
```

resume

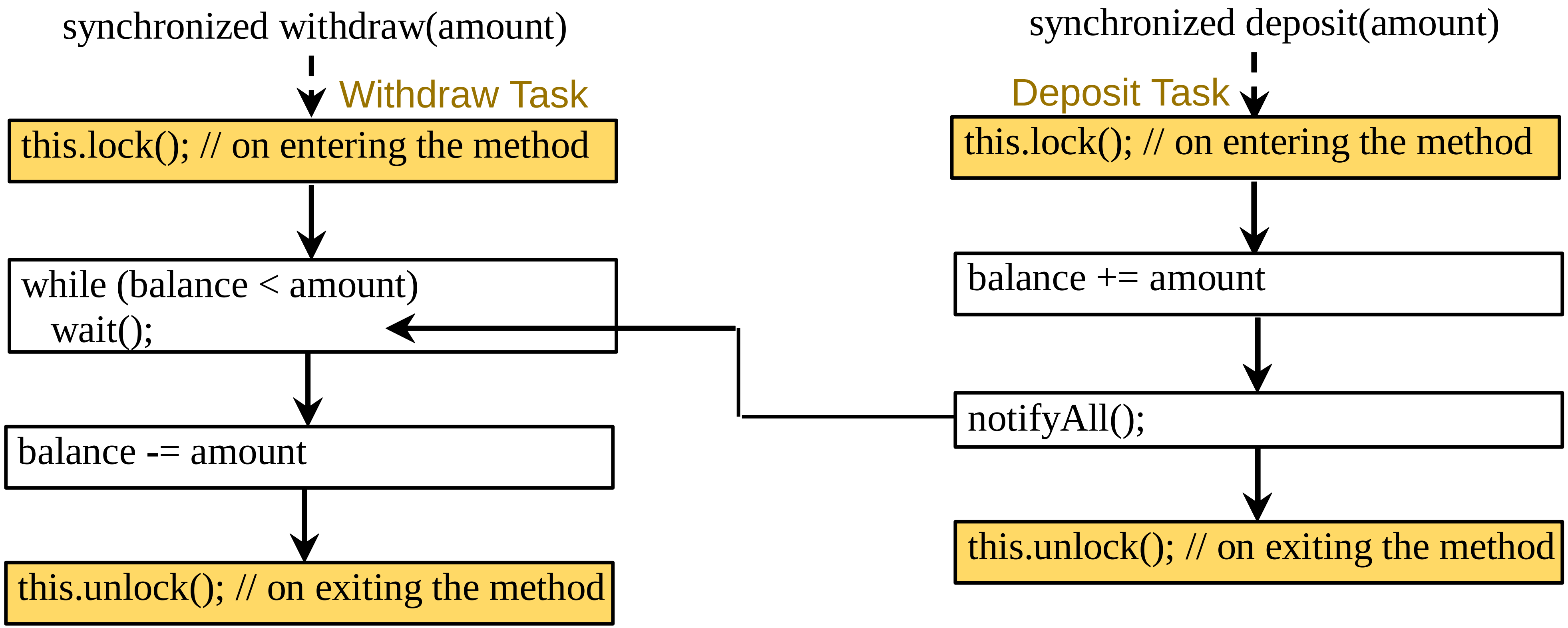
Task 2

```
synchronized (anObject) {
  // When condition becomes true
  anObject.notify(); or  anObject.notifyAll();
  ...
}
```

- The wait(), notify(), and notifyAll() methods must be called in a synchronized method or a synchronized block on the calling object of these methods. Otherwise, an IllegalMonitorStateException would occur.
- The wait() method lets the thread wait until some condition occurs. When it occurs, you can use the notify() or notifyAll() methods to notify the waiting threads to resume normal execution. The notifyAll() method wakes up all waiting threads, while notify() picks up only one thread from a waiting queue.

Thread Cooperation Using Built-in Monitor

To synchronize the operations, use a built-in lock of the account object. If the balance is less than the amount to be withdrawn, the withdraw task will wait. When the deposit task adds money to the account, the task notify the waiting withdraw task to try again. The interaction between the two tasks is shown below.



Concurrency -- Analysis

Analysis of Concurrent Software

- Algorithms and mechanisms to solve two kinds of problems
  - Q1: What will happen?
  - Q2: What has happened?
- Techniques
  - Q1: Static/dynamic bug detection, predictive analysis, testing.
  - Q2: Fault localization, bug reproduction, replay
- They are built on top of two foundations
  - Lockset algorithm
  - Causality models

Analysis : Lockset Algorithms

Are locks correctly used?

Will data race occur?

t1 ⟹ Lock(m1);

v = v + 1;

Unlock(m1);

t2 ⟹ Lock(m2);

v = v + 1;

Unlock(m2);

How to detect the data race?

Lockset [Savage et.al. 1997]
Locking Discipline

- A *locking discipline* is a programming policy that ensures the absence of data-races.
- A simple, yet common locking discipline is to require that every shared variable is consistently protected by some mutual-exclusion lock.
- The *Lockset* algorithm detects violations of locking discipline.
- The main drawback is a possibly excessive number of false alarms.

Lockset
The Basic Algorithm

- For each shared variable *v* let *C(v)* be as set of locks that protected *v* for the computation so far.
- Let *locks_held(t)* at any moment be the set of locks held by the thread *t* at that moment.
- The Lockset algorithm:

  - for each *v*, init *C(v)* to the set of all possible locks

  - on each access to *v* by thread *t*:

    - $C(v) \leftarrow C(v) \cap locks\_held(t)$

    - if $C(v) = \emptyset$, issue a warning

Lockset - Example

Program	locks_held	C(v)
	{ }	{m1, m2}
Lock(m1);		
v = v + 1;	{m1}	{m1}
Unlock(m1);		
	{ }	
Lock(m2);		
v = v + 1;	{m2}	{ } ← warning
Unlock(m2);		
	{ }	

The locking discipline for *v* is violated since no lock protects it consistently.

Lockset Explanation

$$C(v) \leftarrow C(v) \cap locks\_held(t)$$

- Clearly, a lock *m* is in *C(v)* if in execution up to that point, every thread that has accessed *v* was holding *m* at the moment of access.
- The process, called *lockset refinement*, ensures that any lock that consistently protects *v* is contained in *C(v)*.
- If some lock *m* consistently protects *v*, it will remain in *C(v)* till the termination of the program.

Lockset
Improving the Locking Discipline

- The locking discipline described above is too strict.
- There are three very common programming practices that violate the discipline, yet are free from any data-races:
  - Initialization: Shared variables are usually initialized without holding any locks.
  - Read-Shared Data: Some shared variables are written during initialization only and are read-only thereafter.
  - Read-Write Locks: Read-write locks allow multiple readers to access shared variable, but allow only a single writer to do so.
    - *Use two different locks instead of one lock.*

Lockset Initialization

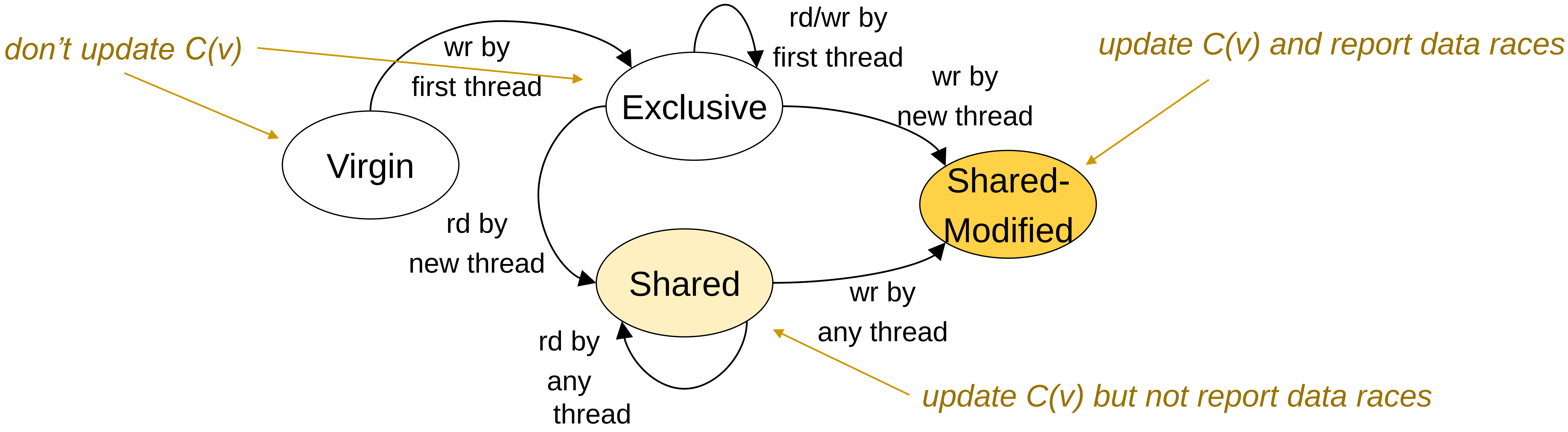
Program	locks_held	C(v)
	{ }	{m1, m2}
Lock(m1);		
v = v + 1;	{m1}	{m1}
Unlock(m1);		
	{ }	
Lock(m2);		
v = v + 1;	{m2}	{ }
Unlock(m2);		
	{ }	

- When initializing newly allocated data there is no need to lock it, since other threads cannot hold a reference to it yet.
- Unfortunately, there is no easy way of knowing when initialization is complete.
- Therefore, a shared variable is initialized when it is first accessed by a second thread.
- As long as a variable is accessed by a single thread, reads and writes don't update *C(v)*.

Lockset
Read-Shared Data

- There is no need to protect a variable if it’s read-only.
- To support unlocked read-sharing, races are not reported after an initialized variable until it has become write-shared by more than one thread.

Lockset
Initialization and Read-Sharing



- Newly allocated variables begin in the *Virgin* state. As various threads read and write the variable, its state changes according to the transition above.
- Races are reported only for variables in the *Shared-Modified* state.

Lockset
Initialization and Read-Sharing

- The states are:
  - *Virgin* – Indicates that the data are new and have not been referenced by any other thread.
  - *Exclusive* – Entered after the data is first accessed (by a single thread). Subsequent accesses don't update *C(v)* (handles initialization).
  - *Shared* – Entered after a read access by a new thread. *C(v)* is updated, but data-races are not reported. In such way, multiple threads can read the variable without causing a race to be reported (handles read-sharing).
  - *Shared-Modified* – Entered when more than one thread access the variable and at least one is for writing. *C(v)* is updated and races are reported as in original algorithm.

Lockset
Read-Write Locks

- Many programs use Single Writer/Multiple Readers (SWMR) locks as well as simple locks.
- The basic algorithm doesn't support correctly such style of synchronization.
- <u>Definition:</u> For a variable *v*, some lock *m* protects *v* if *m* is held in write mode for every write of *v*, and *m* is held in some mode (read or write) for every read of *v*.

Lockset

Read-Write Locks – Final Refinement

- When the variable enters the *Shared-Modified* state, the checking is different:
- Let *locks_held(t)* be the set of locks held in any mode by thread *t*.
- Let *write_locks_held(t)* be the set of locks held in write mode by thread *t*.

Lockset

Read-Write Locks – Final Refinement

- The refined algorithm (for *Shared-Modified*):

  - for each v, initialize $C(v)$ to the set of all locks
  - on each read of v by thread t:
    - $C(v) \leftarrow C(v) \cap locks\_held(t)$ ($\supseteq$ $write\_locks\_held(t)$)
    - if $C(v) = \emptyset$, issue a warning
  - on each write of v by thread t:
    - $C(v) \leftarrow C(v) \cap write\_locks\_held(t)$
    - if $C(v) = \emptyset$, issue a warning

- Since locks held purely in read mode don't protect against data-races between the writer and other readers, they are not considered when write occurs and thus removed from C(V).

Lockset - False Alarms

The refined algorithm works fine with this schedule:

Thread 1	Thread 2	C(v)
Lock(m1); v = v + 1; Unlock(m1);		{m1,m2}
Lock(m2); v = v + 1; Unlock(m2);		{m1,m2}
	Lock(m1); Lock(m2); v = v + 1; Unlock(m2); Unlock(m1);	{m1,m2}

enter shared-modified state

Lockset - False Alarms

The refined algorithm works fine with this schedule:

enter shared-modified state

Thread 1	Thread 2	C(v)
Lock(m1);		
v = v + 1;		{m1,m2}
Unlock(m1);		
	Lock(m1); Lock(m2);	
	v = v + 1;	{m1,m2}
	Unlock(m2); Unlock(m1);	
Lock(m2);		
v = v + 1;		{m2}
Unlock(m2);		

Lockset - False Alarms

The refined algorithm can still produce a false alarm in some schedule:

enter shared-modified state

Thread 1	Thread 2	C(v)
	Lock(m1); Lock(m2); v = v + 1; Unlock(m2); Unlock(m1);	{m1,m2}
Lock(m1); v = v + 1; Unlock(m1);		{m1}
Lock(m2); v = v + 1; Unlock(m2);		{ }

Lockset
Additional False Alarms

- Additional possible false alarms are:
  - Queue that implicitly protects its elements by accessing the queue through locked head and tail fields.
  - Thread that passes arguments to a worker thread. Since the main thread and the worker thread never access the arguments concurrently, they do not use any locks to serialize their accesses.
  - Privately implemented SWMR locks, which don't communicate with Lockset.
  - True data races that don't affect ☞ the correctness of the program (for example "benign" races).

☞
```
if (f == 0)
    lock(m);
    if (f == 0)
        f = 1;
    unlock(m);
```

Lockset
Pros and Cons

☺ Less sensitive to scheduling

☺ Detects a superset of all apparently raced locations in an execution of a program:
races cannot be missed -> sound w.r.t. the given execution

☹ Lots (and lots) of false alarms

☹ Still dependent on scheduling:
cannot prove tested program is race free

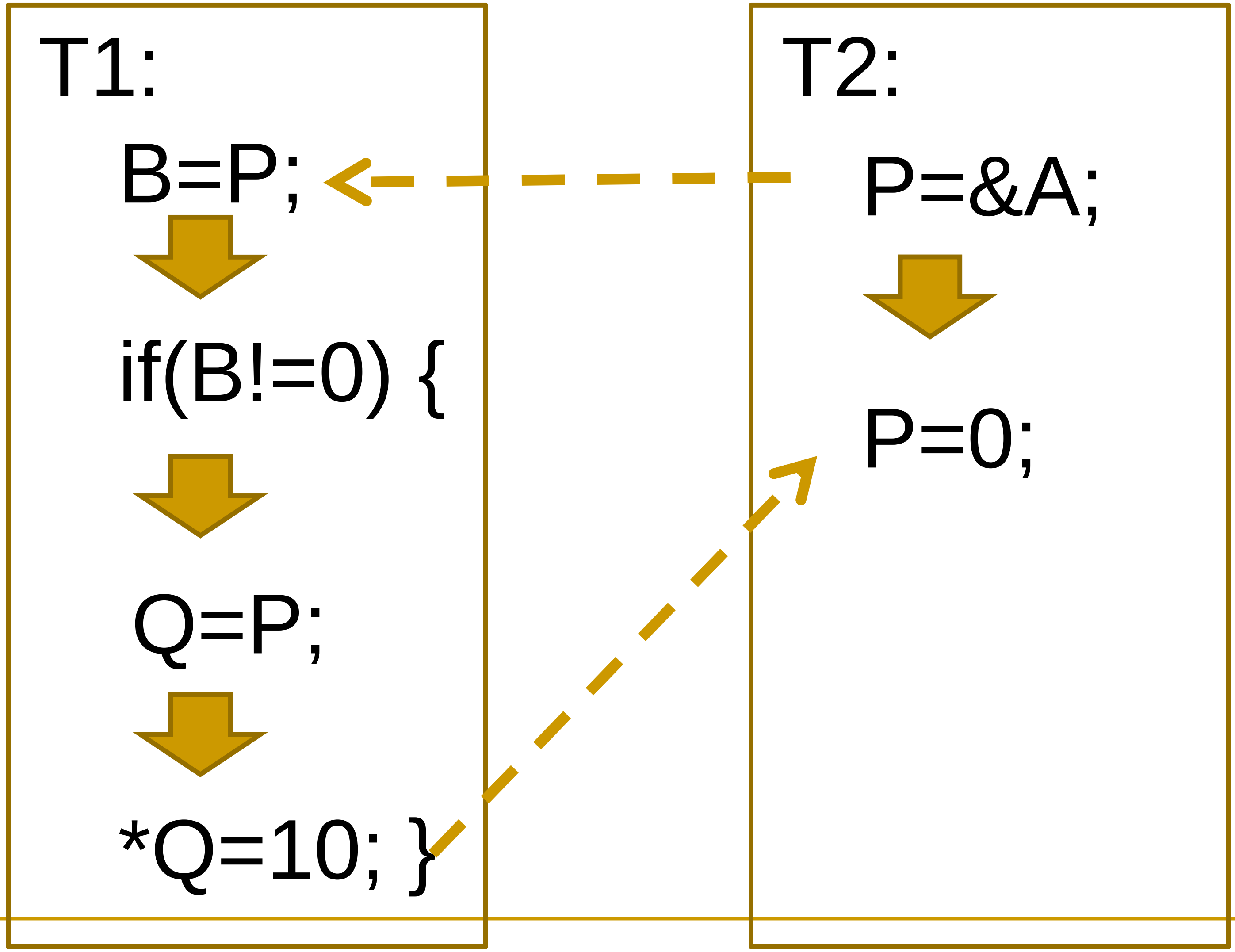
Analysis : Causality Models

Overview: A Sequential Program Execution

```
P=&A;
B=P;
if(B!=0) {
    Q=P;
    *Q=10;
}
P=0;
```

- All instructions are executed one by one.
- The execution order can be described by physical time.
- The execution is fixed when the input is fixed.
- Can we speed it up using multi-core?

One Input, One Possible Execution



Another Possible Execution

T1:

```
B=P;
if(B!=0) {
Q=P;
*Q=10; }
```

T2:

```
P=&A;
P=0;
```

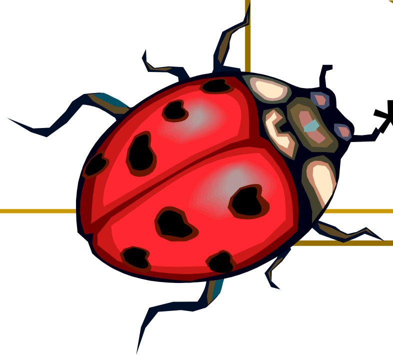
A Possible Buggy Execution

T1:

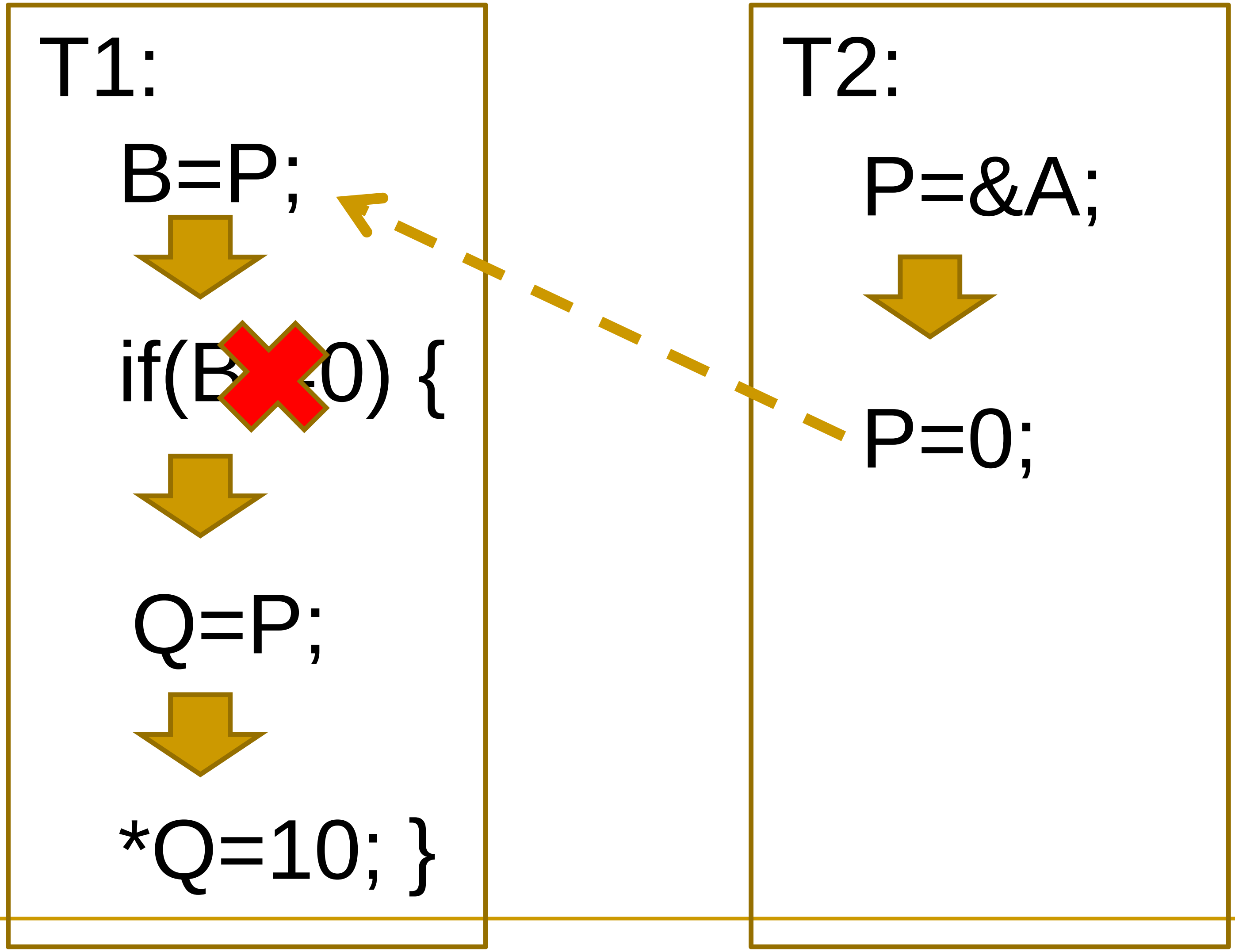
```
B=P;
if(B!=0) {
Q=P;
*Q=10; }
```

T2:

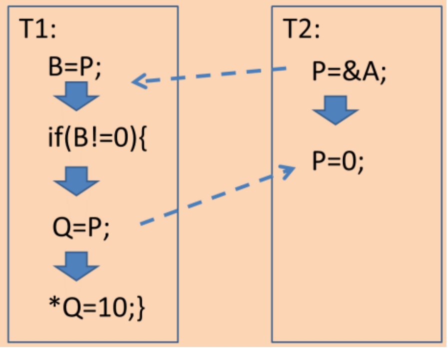
```
P=&A;
P=0;
```



Results of Sequential Run not Preserved

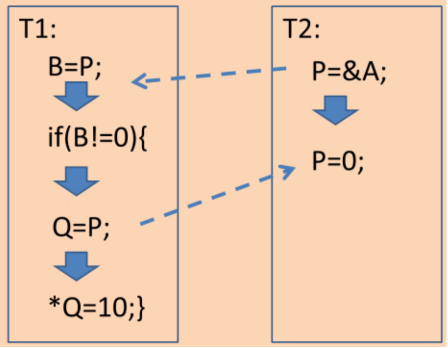


Causality models for concurrent executions



- Causality models:
  - Threads interact in computing intermediate and final program states.
  - A causality model is to encode the causal effects on share memory among threads.
- Purpose:
  - Fundamental question: given a possible run, what are the other possible runs if we still maintain the same set of causal effect.
  - Useful for finding high level errors such as data race or atomicity violations.

The Causality Relationship



- Each thread can affect the execution of other threads
  - sending messages -> lock/unlock, wait/notify, explicit communication
  - updating shared variables
- A causality model captures the effects among threads to describe a program execution
  - Why does a read operation get a certain value such as 10?
  - Why does a path condition hold?

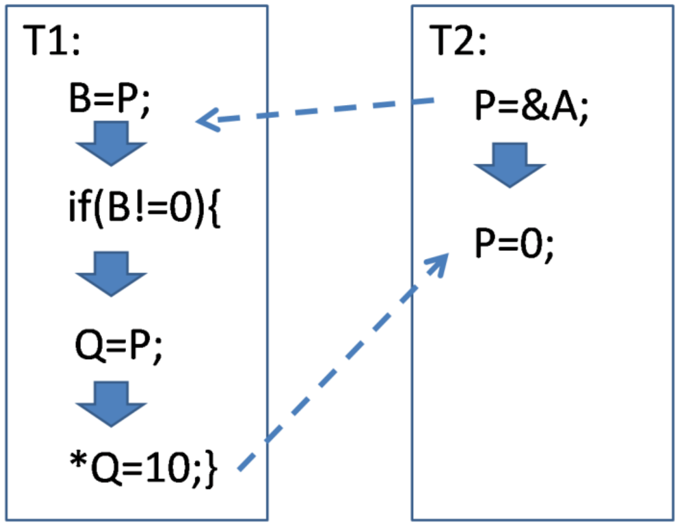
Describing a Concurrent Execution

- $T_1, T_2, \ldots, T_n$: a set of threads
- A set of variables:
  - LV_x: Local variables for each thread T_x
  - SV: Shared variables
- E_x: a sequence of events for each thread T_x
  $E_x=(e_1,e_2,\ldots,e_m)$, where e_k is an event in thread T_x

Modeling the permitted order of all the events in the event sequences for all threads.

Describing a Concurrent Execution

Observed Execution:



Happens-Before:

Order of events concerning shared variable P

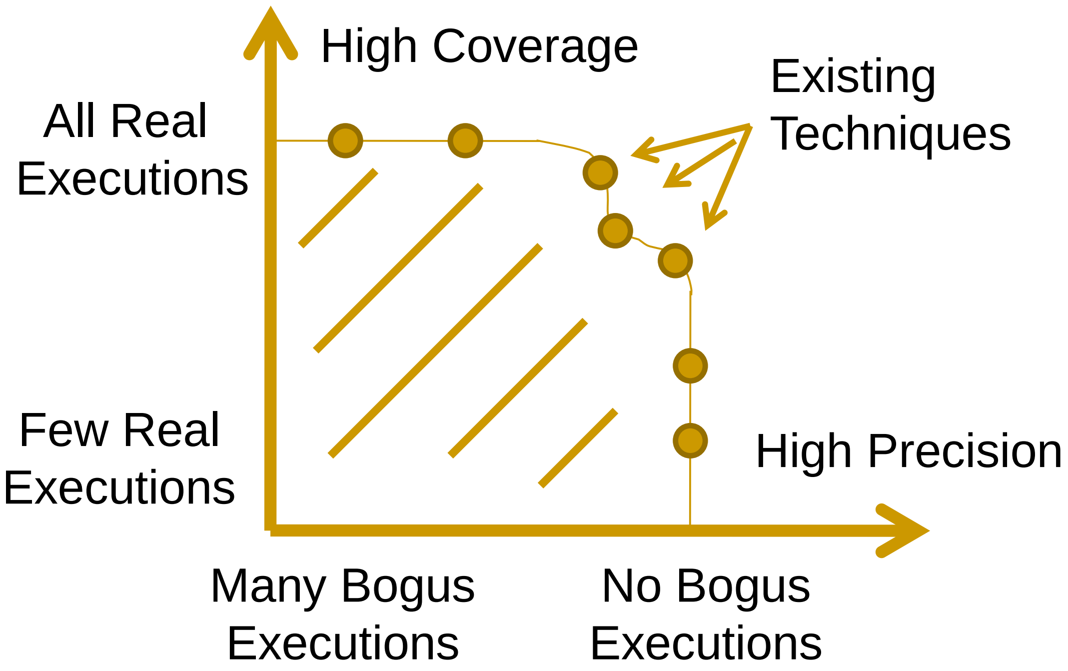
T2:P=&A;
T1:B=P;
T1:Q=P;
T2:P=0;

Causality model

Modeling the permitted order of all the events in the event sequences for all threads.

The Ideal Causality Model

- High Coverage
  - Ideal: Modeling all real executions
- High Precision
  - Ideal: Modeling no bogus executions



Existing Causality Models

- High precision but partial coverage
  - Happens-before [Lamport, 1978]
  - Weak happens-before [Sen & et al., FMOODS'05]
  - Lipton's Reduction [Lipton, POPL 1975]
- High coverage but partial precision
  - Lockset [Praun & Gross, OOPSLA'01]

Existing Causality Models

- High precision but partial coverage
  - Happens-before [Lamport, 1978]
  - Weak happens-before [Sen & et al., FMOODS'05]
  - Lipton's Reduction [Lipton, POPL 1975]
- High coverage but partial precision
  - Lockset [Praun & Gross, OOPSLA'01]

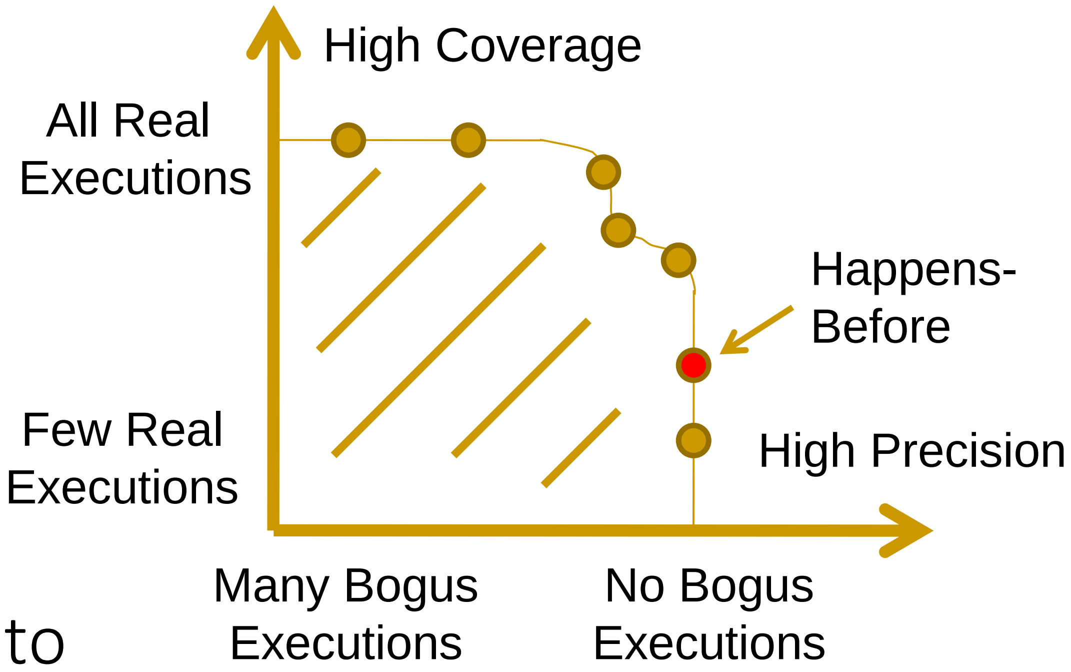
Happens-Before [Lamport, 1978]

- *Definition. The relation “Happens-Before" (<) on the set of events of* a system is the smallest relation satisfying the following three conditions:
  - (1) If e and e’ are events in the same process, and e comes before e’, then e<e’.
  - (2) If e is the sending of a message by one process and e’ is the receipt of the same message by another process, then e<e’.
  - (3) If e<e’ and e’<e’’ then e<e’’. (transitivity)

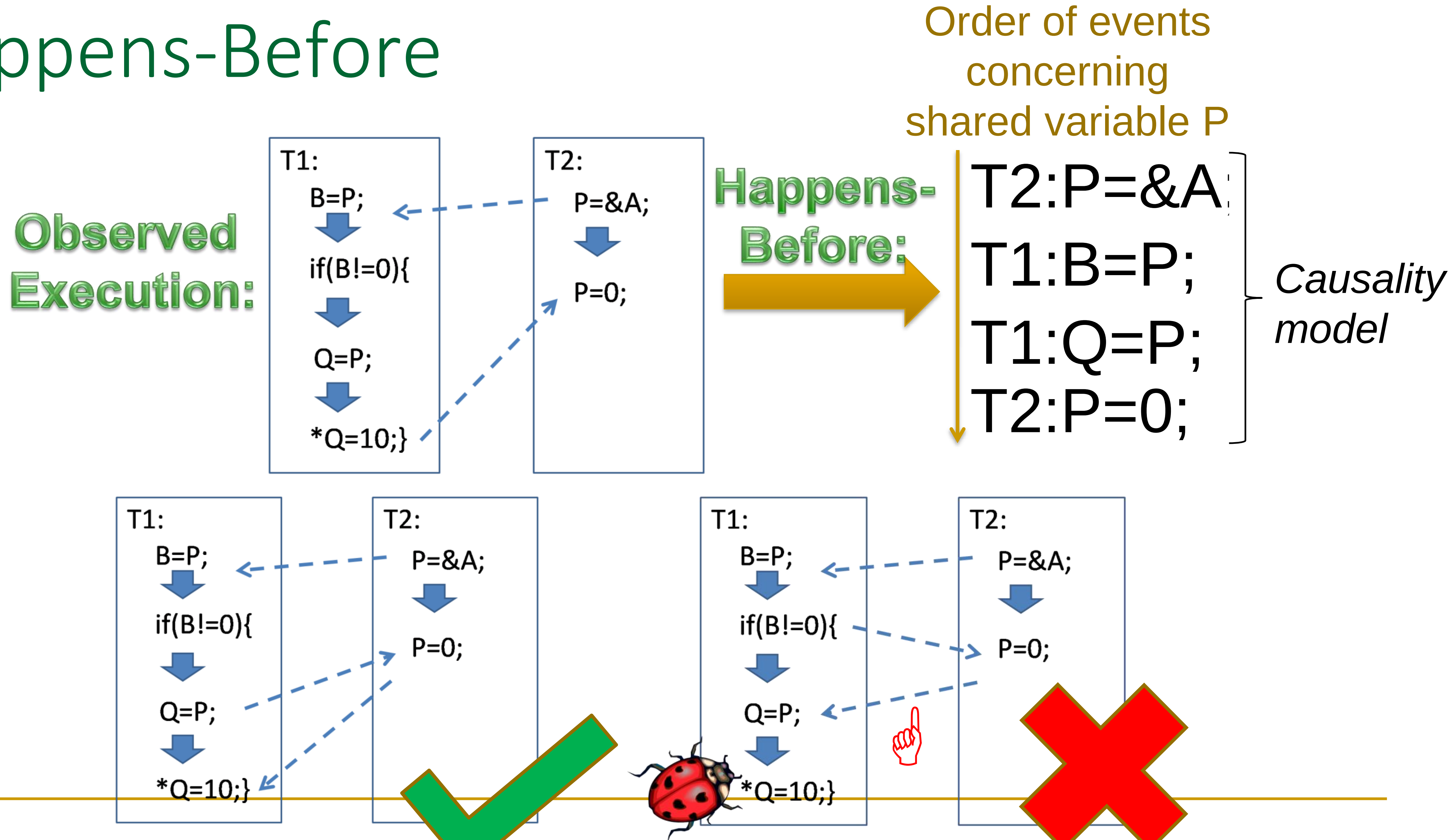
T2:P=&A;
T1:B=P;
T1:Q=P;
T2:P=0;

Happens-Before

- All operations on the same object are kept in the same order as in an observed execution
  - Full Precision
    - Modeling no bogus executions
  - Low Coverage
    - Very few traces modeled, compared to all possible execution traces.



Happens-Before



Existing Causality Models

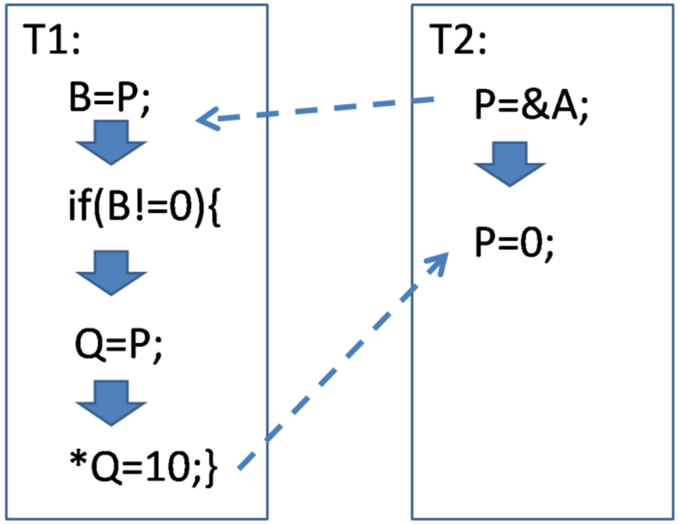
- High precision but partial coverage
  - Happens-before [Lamport, 1978]
  - Weak happens-before [Sen & et al., FMOODS’05]
  - Lipton’s Reduction [Lipton, POPL 1975]
- High coverage but partial precision
  - Lockset [Praun & Gross, OOPSLA’01]

Weak happens-before

- *Definition. The relation “Weak Happens-Before" ($\lessdot$) on the set of events of* a system is the smallest relation satisfying the following conditions:
  - If e and e’ are events of the same thread and e comes before e’, then $e \lessdot e'$.
  - Whenever there is a variable x with $e \lessdot_x e'$, $e \lessdot e'$.
    - $e \lessdot_x e'$ if e’ is a read event of variable x reading the value written by write event e.
  - If $e \lessdot e'$ and $e' \lessdot e''$ then $e \lessdot e''$

Weak happens-before

Observed Execution:

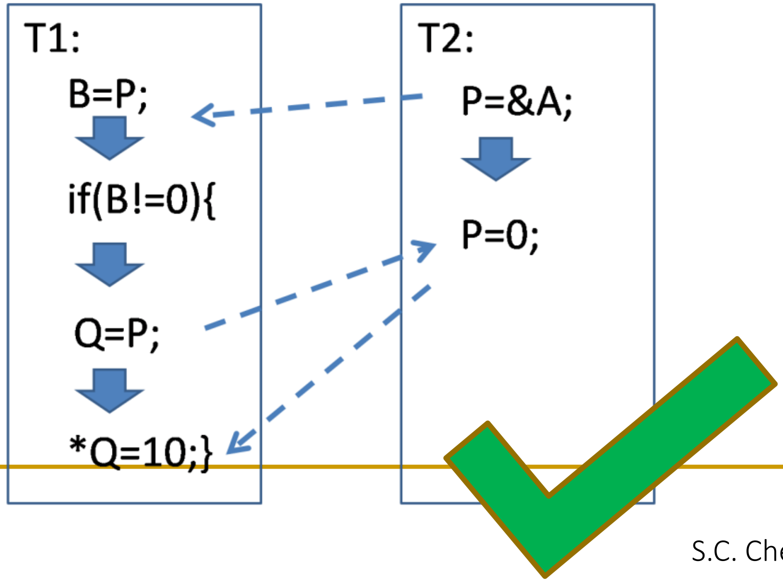


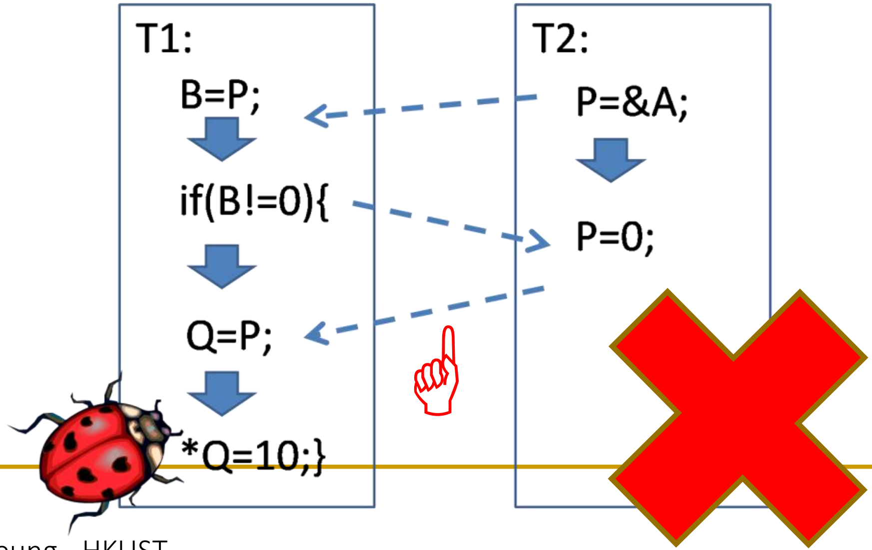
Weak Happens-Before:

Order of events concerning shared variable P

T2:P=&A;
T1:B=P;
T1:Q=P;

Causality model

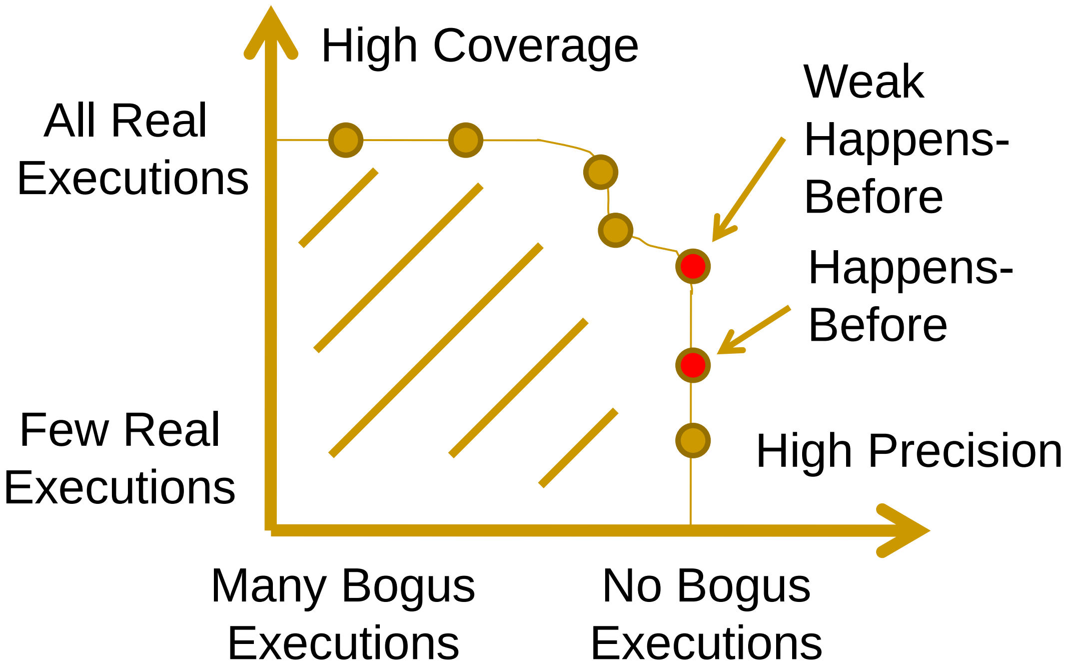




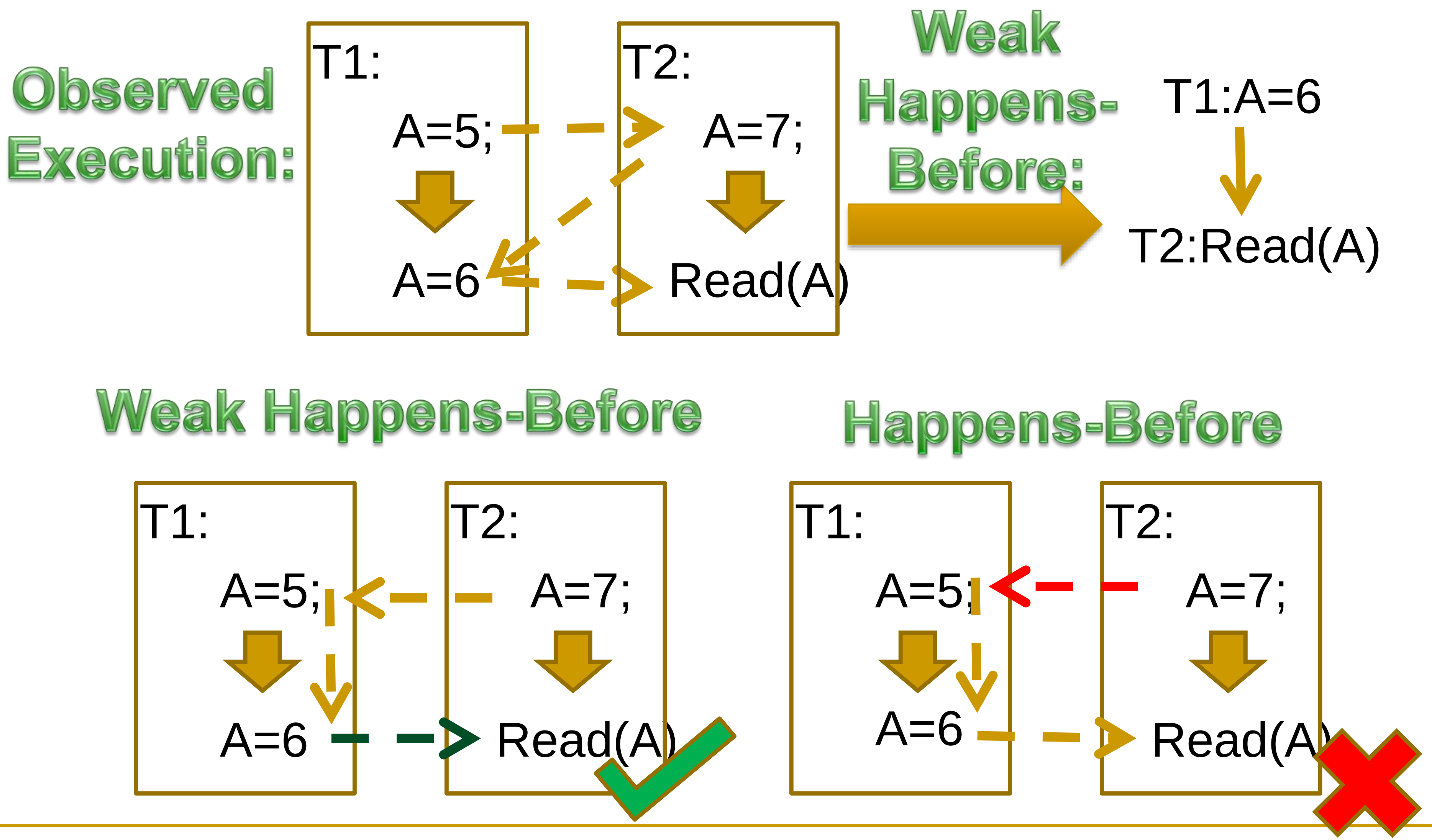
An extra event order not found in the causality model

Weak happens-before

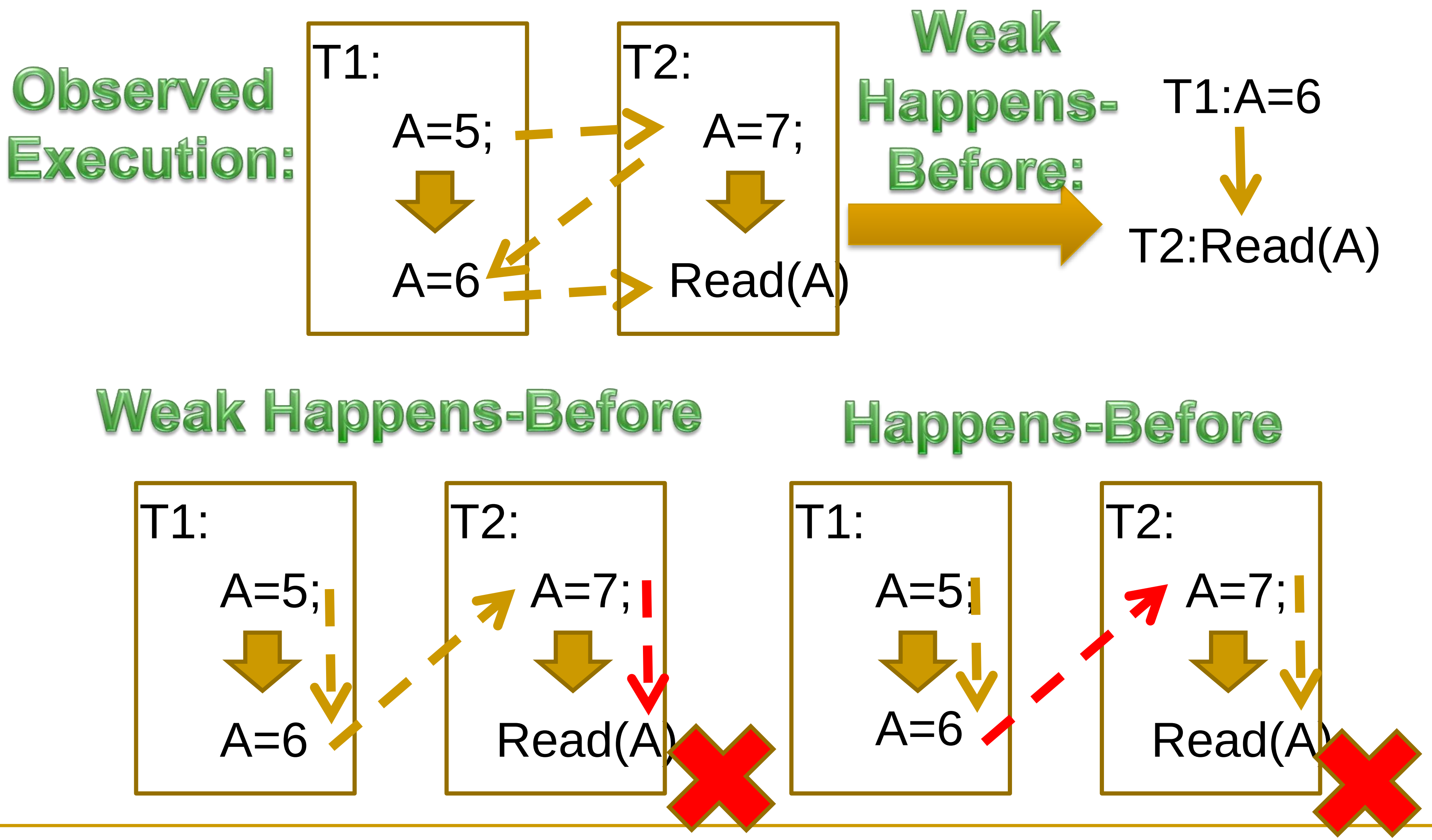
- Only the read-after-write dependency is preserved.
  - Full Precision
    - Modeling no bogus executions
  - Higher Coverage than Happens-Before.



Weak Happens-Before VS Happens-Before



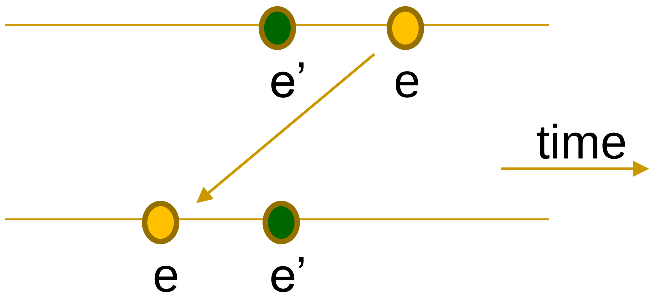
Weak Happens-Before VS Happens-Before



Existing Causality Models

- High precision but partial coverage
  - Happens-before [Lamport, 1978]
  - Weak happens-before [Sen & et al., FMOODS’05]
  - Lipton’s Reduction [Lipton, POPL 1975]
- High coverage but partial precision
  - Lockset [Praun & Gross, OOPSLA’01]

Lipton’s Reduction



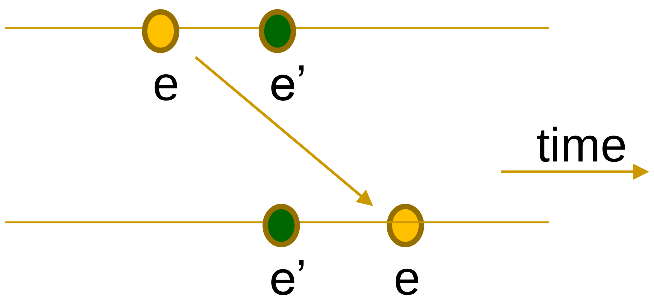
- Left-mover:

  An event e that can be moved before its previous event e’ in an execution without affecting the rest of the execution, where

  - (1) e’ and e are in different threads
  - (2) e’ can be any event

- Theorem: All events that release resources are Left-movers:
  - Unlock
  - Notify

Lipton's Reduction



- Right-mover:

  An event e that can be moved after its next event e' in an execution without affecting the rest of the execution, where
  - (1) e' and e are in different threads
  - (2) e' can be any event
- Theorem: All events that acquire resources are Right-movers:
  - Lock
  - Wait

Lipton's Reduction

- Allow the reordering of Left-movers with previous events and Right-movers with following events.
  - Full Precision
    - Modeling no bogus executions
- Focus only on synchronizations
  - Memory access operations can hardly be left-movers or right-movers
  - Better coverage applied together with Happens-Before.

Existing Causality Models

- High precision but partial coverage
  - Happens-before [Lamport, 1978]
  - Weak happens-before [Sen & et al., FMOODS'05]
  - Lipton's Reduction [Lipton, POPL 1975]
- High coverage but partial precision
  - Lockset [Praun & Gross, OOPSLA'01]

Lockset

- For each event e , we denote L(e) as the set of locks protecting e. Events e_1 and e_2 can be executed concurrently iff $L(e_1) \cap L(e_2) = \emptyset$.
- Full Coverage
  - Covering all possible executions
- Low precision

Lockset

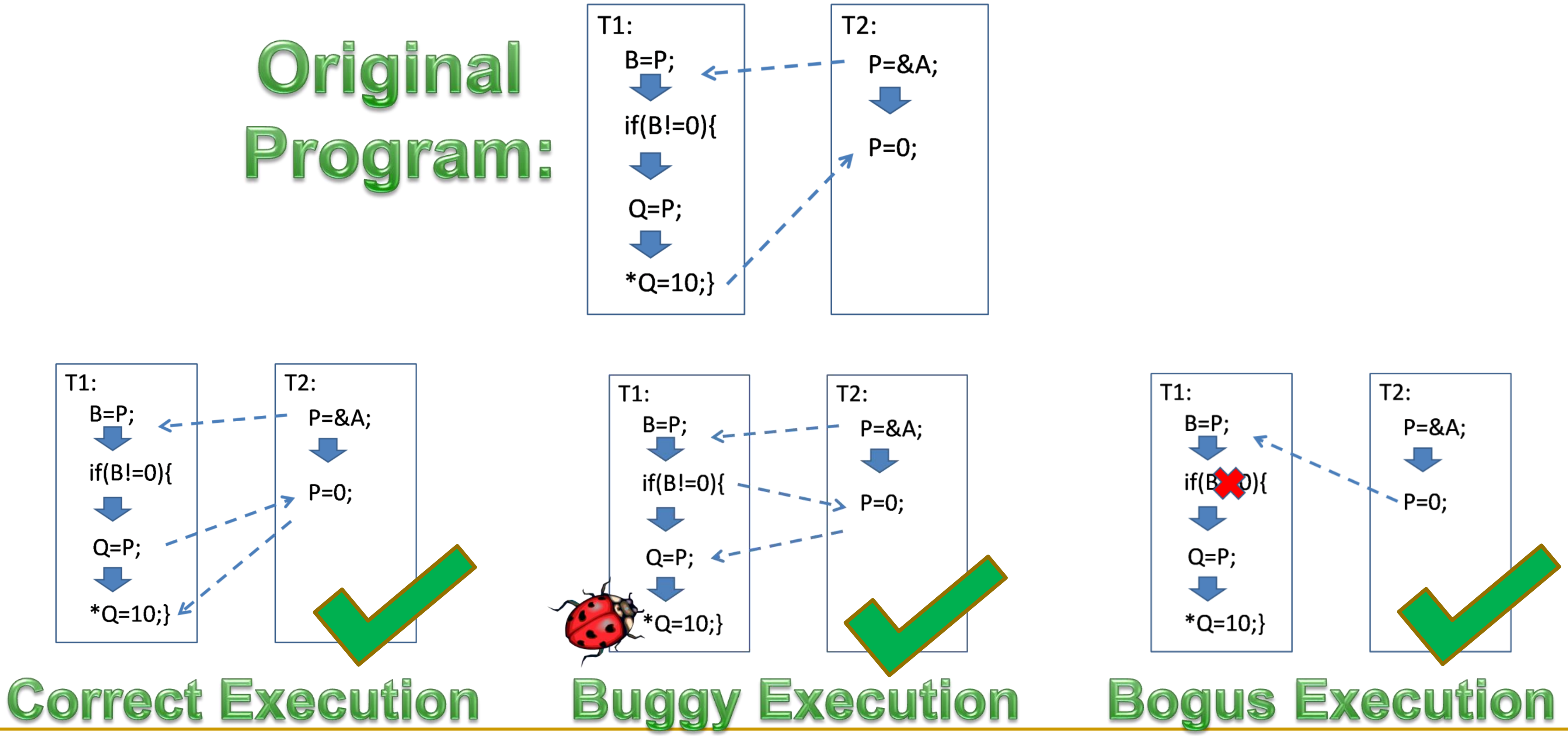
T1:	T2:
Lock(L1)	Lock(L2)
Lock(L2)	e_3: A=7;
e_1: A=5;	UnLock(L2)
Unlock(L1)	Lock(L1)
e_2: A=6	e_4: A=8
Unlock(L2)	Unlock(L1)

Compute LockSet:

$L(e_1)=\{L1,L2\}$
$L(e_2)=\{L2\}$
$L(e_3)=\{L2\}$
$L(e_4)=\{L1\}$

$Concurrent(e_1,e_3)=False$
$Concurrent(e_2,e_4)=True$

Lockset: No Locks, No Constraints



Bug Detection

- Improving Precision
  - Apply causality models with high precision
  - No False Positives: Apply causality models with full precision
    - Happens-Before, Weak Happens-Before
- Improving Recall
  - Apply causality models with high coverage
  - No False Negatives: Apply causality models with full coverage
    - Lockset
- Pattern-based (like FindBugs)
  - Heuristics (no guarantee on false positives and false negatives)
  - Compromise between precision and recall
  - Purely static, low overhead, and most popularly used
  - Keshmesh (open-source) and ThreadSafe (commercial)

Keshmesh

Concurrency Bug Pattern Detector

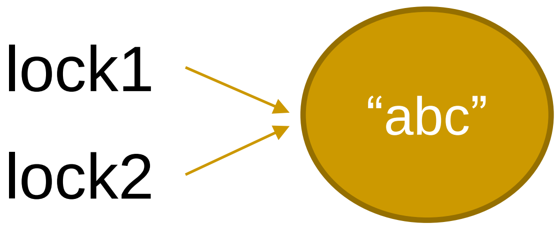
"Keshmesh: Bringing Advanced Static Analysis to Concurrency Bug Pattern Detectors" by Mohsen Vakilian, Stas Negara, Samira Tasharofi, and Ralph E. Johnson. Paper at IDEALS, 2013.

Keshmesh

- Implements 5 concurrency bug pattern detectors.
- Available from Eclipse market.
- Successfully detect 50+ unreported concurrency bugs in real-world open source projects.
- Extensible to implement more bug pattern detectors and even fixers.

BP1: Synchronizing on an object that may be reused

```
String lock1 = ”abc”;
synchronized (lock1) {
   …
}

String lock2 = “abc”;
synchronized (lock2) {
   …
}
```



- The intern String object “abc” is reused for both lock1 and lock2.
- This makes the two synchronized blocks mutually exclusive.
- E.g., Integer semaphore = 1;
- Consider using:
  - new String(“abc”);
  - new Integer(1);

BP2: Synchronizing on the class object returned by getClass()

```
class ContextWriter … {
  Context currentContext;
  Context getCurrentContext() … {
     …
     synchronized (getClass()) {
     if (currentContext == null) {
        currentContext = …;
     }
     return currentContext;
  }
}
```

```
class EnhancedContextWriter
  extends ContextWriter {
  Context getCurrentContext(int id) … {
     …
     synchronized (getClass()) {
        currentContext = …;
     }
     return currentContext;
  }
}
```

- The two getClass() methods return two different objects.
- They do not synchronize accesses to currentContext by two threads that reference an object of ContextWriter and EnhancedContextWriter, respectively.

Other Bug Patterns

- BP3: Synchronizing on a high-level concurrency object.
  - ReentrantLock lock;

    synchronized (lock) {…}

interfere with lock.lock() and lock.unlock()

- BP4: Using "this" to protect a shared static variable.
- BP5: Unprotected access to shared variables.

BP4 & BP5

```
1  public class SocketConnector ... {
2    ...
3    static int id = 0;
4    synchronized NioThread getSelector() {
5      if (selector == null) {
6        String name = "Selector-" + id++;
7        selector = new NioThread(name);
8      }
9      return selector;
10   }
11 }
```

(a) An instance of BP_4 in a Tomcat module (Subversion revision 1435416). Method **getSelector** uses an instance lock (**this**) to protect an access to the shared **static** field **id** on line 6.

```
1  public class FastQueue {
2    private boolean enabled = true;
3    boolean add(...) {
4      ...
5      if (!enabled) {
6        if (log.isInfoEnabled())
7          log.info(...);
8        return false;
9      }
10     ...
11   }
12   void setEnabled(boolean enable) {
13     enabled = enable;
14     if (!enabled) {
15       lock.abortRemove();
16       last = first = null;
17     }
18   }
19   ...
20 }
```

(b) An instance of BP_5 in Tomcat. Field **enabled** (line 2) is shared data, but, its accesses (lines 5, 13, and 14) are not mutually exclusive.

Further Readings

- Valerio Terragni and Shing-Chi Cheung. Coverage-Driven Test Code Generation for Concurrent Classes. In Proceedings of the 38th International Conference on Software Engineering (ICSE 2016), Austin, TX, USA, May 2016, pp. 1121-1132.
- Valerio Terragni, Shing-Chi Cheung and Charles Zhang. RECONTEST: Effective Regression Testing of Concurrent Programs. In Proceedings of the 37th International Conference on Software Engineering (ICSE 2015), Florence, Italy, May 16-24, 2015, pp. 246-256.
- Christian Hammer, Julian Dolby, Mandana Vaziri, and Frank Tip, Dynamic detection of atomic-set serializability violations, ICSE 2008, pp. 231-240.
- Zhifeng Lai, Shing-Chi Cheung, and W.K Chan, Detecting atomic-set serializability violations in multithreaded programs through active randomized testing, ICSE 2010, pp. 235-244.
- Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold, Falcon: Fault localization in concurrent programs, ICSE 2010, pp. 245-254.