Data Race Bug

Shin Hong

Race Condition in Multithreaded Program

- A multithreaded program has a *race condition* if (1) execution order of certain operations are not fixed, and (2) their execution results are decided by their non-deterministic execution orders.

- Race conditions sometime cause serious errors in SW
  - e.g. Therac-25

Q: Is a race condition in a program always problematic?

Race Condition Harmful? (1/2)

Ex. Parallel adder

```
long cnt=0 ;
long arr[100] ;
long sum1=0, sum2=0;

main() {
  read(arr, 100) ;
  start(work, &sum1);
  start(work, &sum2);
  print(sum1 + sum2) ;
}

work(long * sum) {
  while (cnt < 100) {
    *sum += arr[cnt] ;
    cnt++ ;
  }
}
```

Has a race condition?

Is this race condition harmful?

Race Condition Harmful? (2/2)

Ex. Seminar room reservation system

```
1   service() {
2     input(&room, &timeslot) ;
3     if(isAvailable(room, timeslot){
4       print("available. continue?") ;
5       input(&continue) ;
6       if(continue)
7         if(reserve(room, timeslot))
8           print("reserved.") ;
9         else
10          print("not available.") ;
11  } }
```

Is this race condition harmful?

User#1: "Rm01", "7PM Today"
System: available. continue?

User#2: "Rm01", "7PM Today"
System: available. continue?
User#2 "yes"
System: reserved.

User#1: "Yes"
System: not available.

Race Bug

- A *race bug* is a multithreaded program fault that causes race conditions leading to *unintended* program behaviors (i.e. invalid states)

- Race bug detectors detect (predict) race conditions that violate concurrency requirements

Data Race

- A *data race* is a pair of concurrent operations that read and write (or both write) data on a same memory location without synchronization

- A data race may violate *sequential consistency** of a target program execution

* L. Lamport: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs, IEEE Transactions on Computers, 1978

Sequential Consistency

- Lamport's definition

  "A multiprocessor is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor occur in this sequence in the order specified by its program"

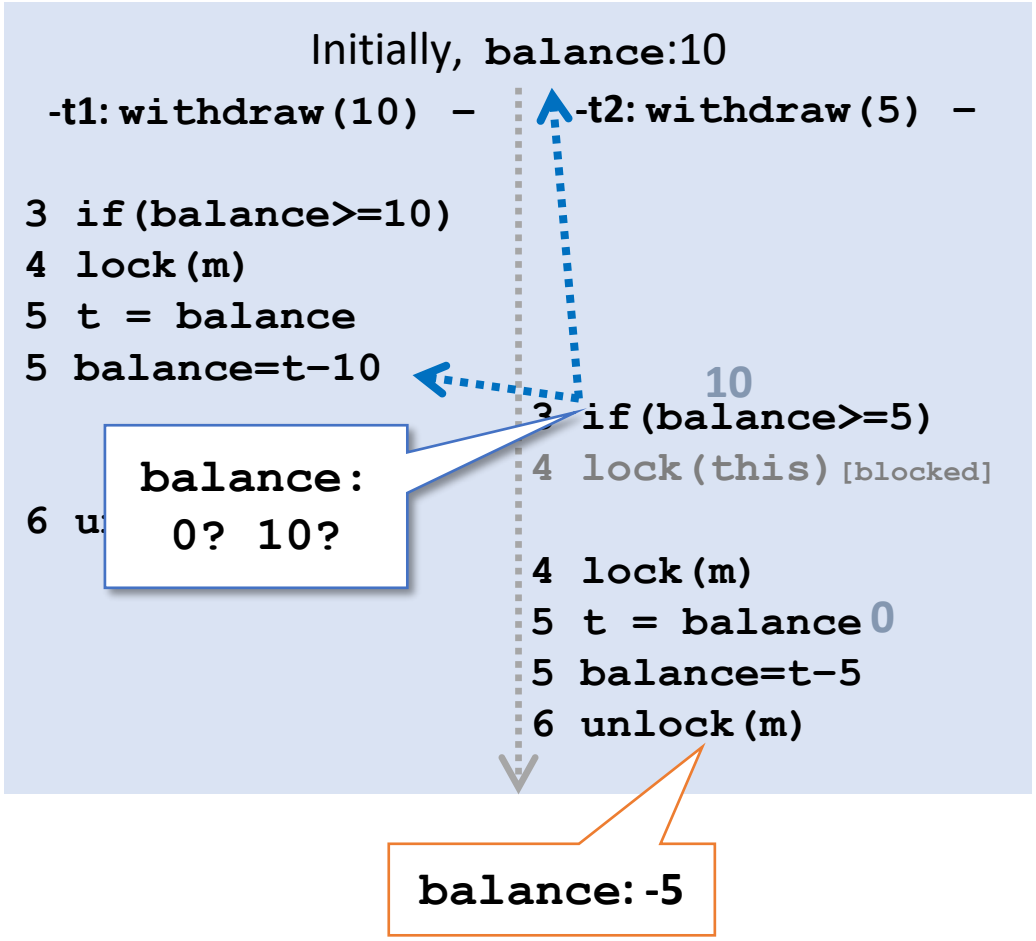
- Most intuitive consistency model for programmers
  - Processors see their own loads and stores in program order
  - Processors see others' loads and stores in program order
  - All processors see same global load and store ordering

Prof. Huh's lecture note on "Consistency"

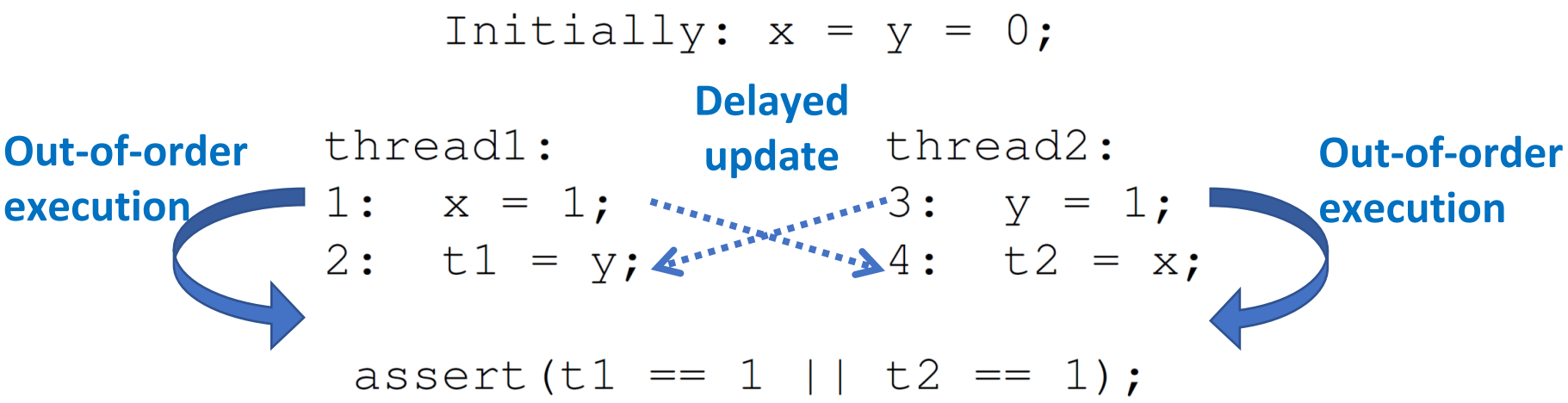
Data Race Example

```
   class Account {
1   long balance;
    //must be non-negative

2   void withdraw(long x){
3    if(balance >= x){
4     lock(m) ;
5      balance=balance-x ;
6     unlock(m) ;
7    }
8   }
   }
```



Data Race Breaks Sequential Consistency



Does the assertion hold with SC memory model?

Does the assertion hold with weak memory model?

* J. Burnim, K. Sen, C. Stergiou: Testing concurrent programs on relaxed memory models. ISSTA 2011

Why Data Race is Bug?

Data races are harmful in most cases

- Execution results are (almost) unpredictable with weak memory models
- Compilers may reorder statements around data races*
- Performance benefit of benign race is really marginal*
- It is bad for maintenance

* H. J. Boehm: Nondeterminism is unavoidable, but data races are pure evil, RACES Workshop, 2012

Detecting Data Race

- Data races are notoriously difficult to detect
  - Unlike deadlock, the program behavior change by a data race may not be noticeable to users
  - Data races induce errors only under specific thread schedules
  - There are too many shared variables

- There have been two approaches:
  1. Happens-before based detection technique
  2. **Lockset algorithm based detection technique**

Lockset Based Data Race Detection

- Lock discipline
  - Every access to a shared variable MUST be guarded by at least one lock consistently
- Dynamic data race detector *Eraser* [Savage, SOSP 97]
  - Check that every shared memory location follows a lock discipline
    - Consider memory locations for global variables, and heap memory locations as shared memory locations

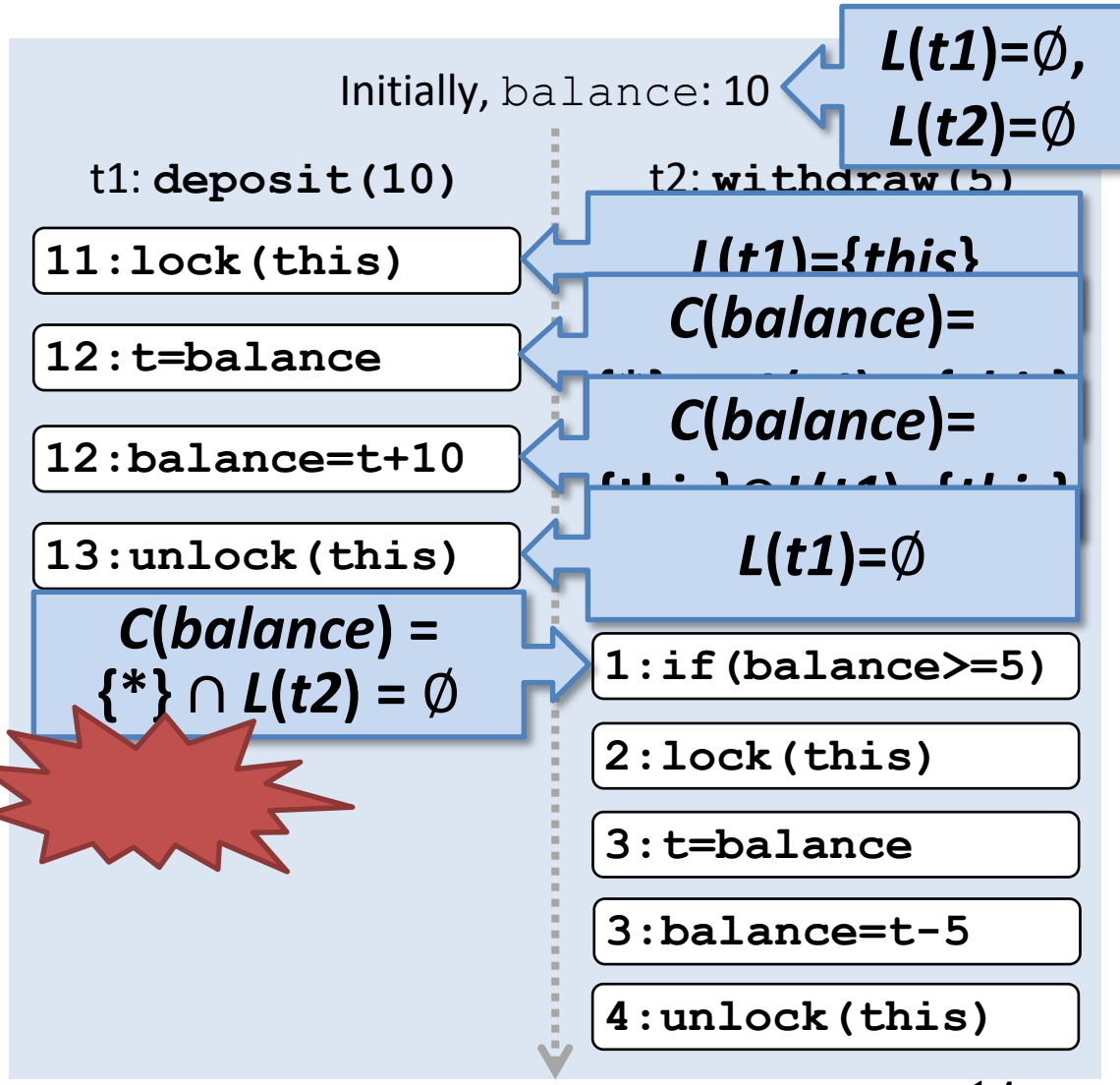
Lockset Algorithm

- Eraser monitors every read/write operation and every lock/unlock operation in an execution
- For each variable v, Eraser maintains the lockset $C(v)$, candidate locks for the lock discipline
  - Let $L(t)$ be the set of locks held by thread t
  - For each v, initialize $C(v)$ to the set of all locks
- For each read/write on variable v by thread t
  - $C(v) := C(v) \cap L(t)$
  - If $C(v) = \emptyset$, report that there is a data race for v

Lockset Algorithm Example

```
  class Account {
    long balance;
    //must be non-negative
   void withdraw(long x){
 1   if(balance >= x){
 2    lock(this);
 3     balance=balance - x;
 4    unlock(this)
 5   }
   }

   void deposit(long x){
11   lock(this);
12    balance=balance + x;
13   unlock(this);
   }
  }
```



Improving Lockset Algorithm

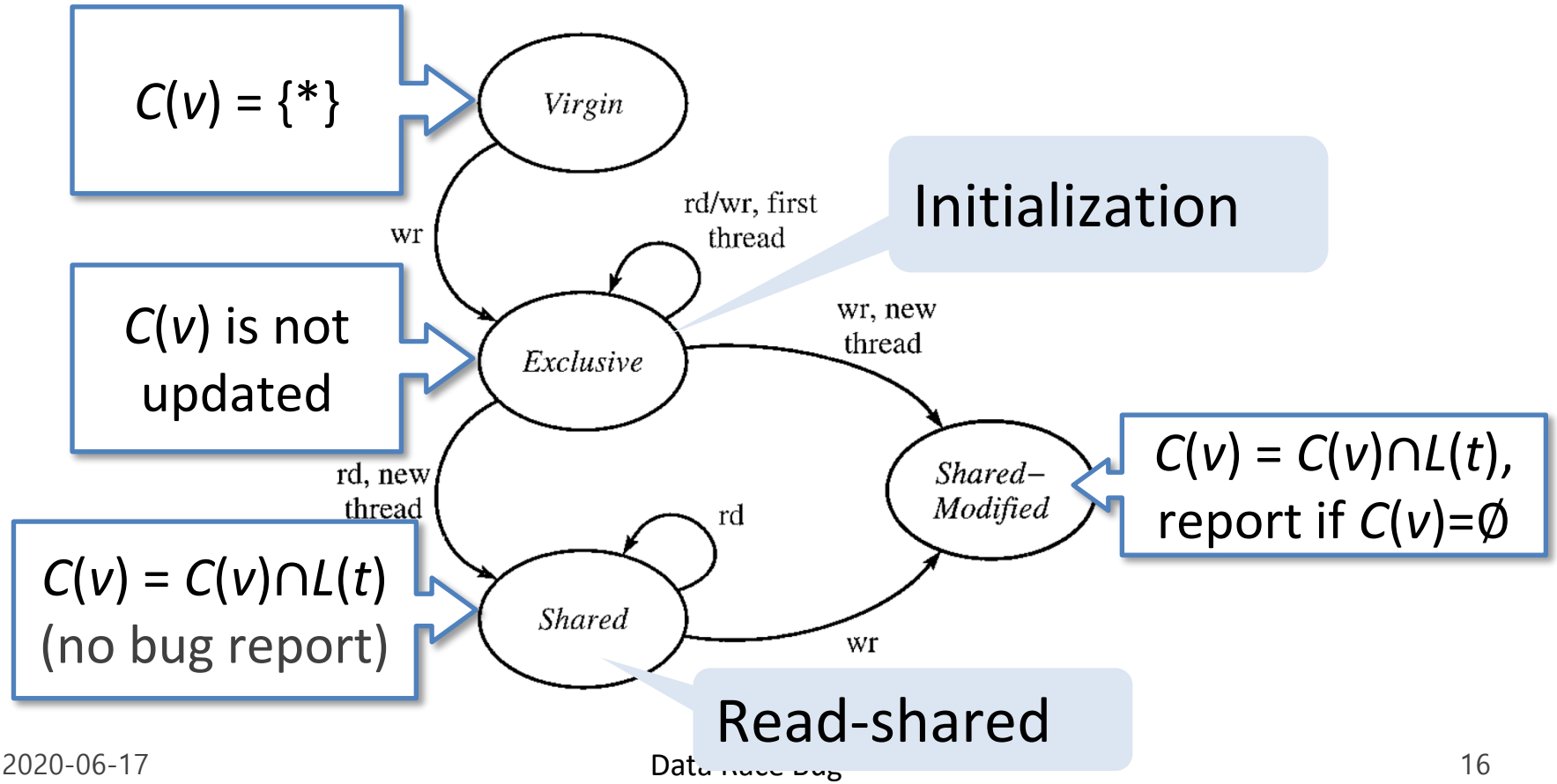
- The naïve locking discipline generate many false positives

- Three popular cases for generating false positive

  1. Initialization
     - A thread writes data on the variable without locking before is makes the variable accessible by other threads
  2. Read-shared variable
     - After initialization, the variable is only read, and never updated.
  3. Readers-writer lock

Memory Location State

- Eraser maintains the state for each memory location to check if it is in initialization, and if read-shared.



Reducing False Positives

- Use happens-before relation induced by wait/notify and thread start/join to reduce false positives
- Check if one memory location is once used for a variable, and then re-used for another variable
- Track all references to a memory location to precisely check if multiple threads can access the memory location

Reducing False Negative

- Check for a set of memory locations assigned for a single variable rather than a single memory location
  - E.g. `long`, `double`, array, compound data (`struct`)

Performance Improvement (1/2)

- Dynamic data race detection tools are still too slow to be practical
  - Inter ThreadChecker incurs 100—200x slow down, Google ThreadSanitizer 30--40x, and FastTrack 8.5x in average*
- Approach
  - **Pre-processing**: use static analyses to filter out non-shared variables and read-only variables before runtime monitoring
  - **Hardware assisted monitoring**: use a customized hardware to monitor memory accesses and synchronization with low cost

* T. Sheng et al.: RACEZ: A Lightweight and Non-invasive Race Detection Tool for Production Applications, ICSE 2011

Performance Improvement (2/2)

- Approach (cond.)
  - **Sampling:** monitor only a subset of operations, or a subset of memory locations
    - *LiteRace* [Marino, PLDI 09] assumes the cold region hypothesis
      "data races are likely to occur when a thread is executing **cold** (infrequently accessed) region in the program"
    - *Pacer* [Bond, PLDI 10] allows users to configure sampling ratio, and guarantees higher detection ratio for higher sampling ratio.
    - *RACEZ* [Sheng, ICSE 11] exploits performance monitoring unit to obtain partial information on memory accesses with low cost