CS492B Analysis of Concurrent Programs

Data Race Detection Technique

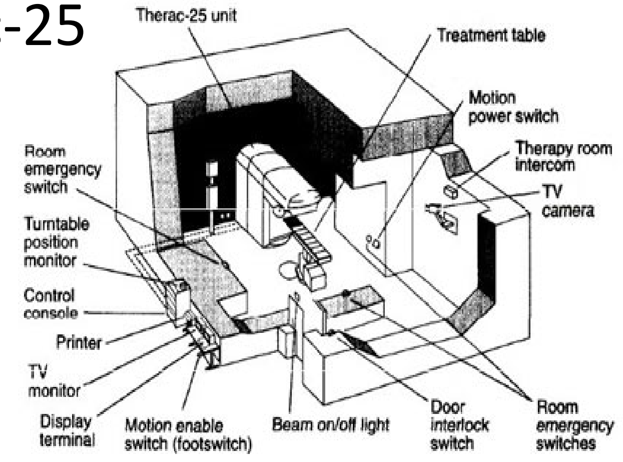
Prof. Moonzoo Kim

Computer Science, KAIST

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. Radiation therapy machine: Therac-25



Q: Is a race condition always problematic?

Harmful Race Condition

Ex. Parallel adder

```
// adding all numbers in arr[]
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?

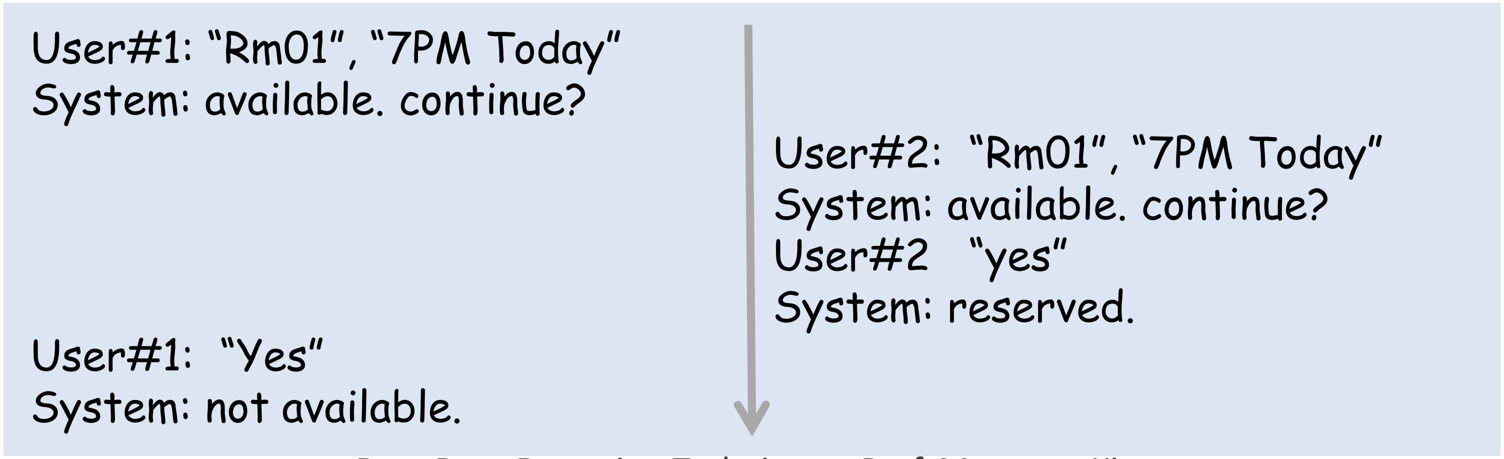
Is this race condition harmful?

Not Harmful Race Condition

Ex. Seminar room reservation system

```
service() {
  input(&room, &timeslot) ;
  if(isAvailable(room, timeslot){
    print("available. continue?") ;
    input(&continue) ;
    if(continue)
      if(reserve(room, timeslot))
        print("reserved.") ;
      else
        print("not available.") ;
} }
```

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 may violate <u>common concurrency requirements</u>

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 (i.e., concurrently without any coordination)

- 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

  (1) the result of **any** execution is the same as if the operations of all the processors were executed in **some** sequential order, and

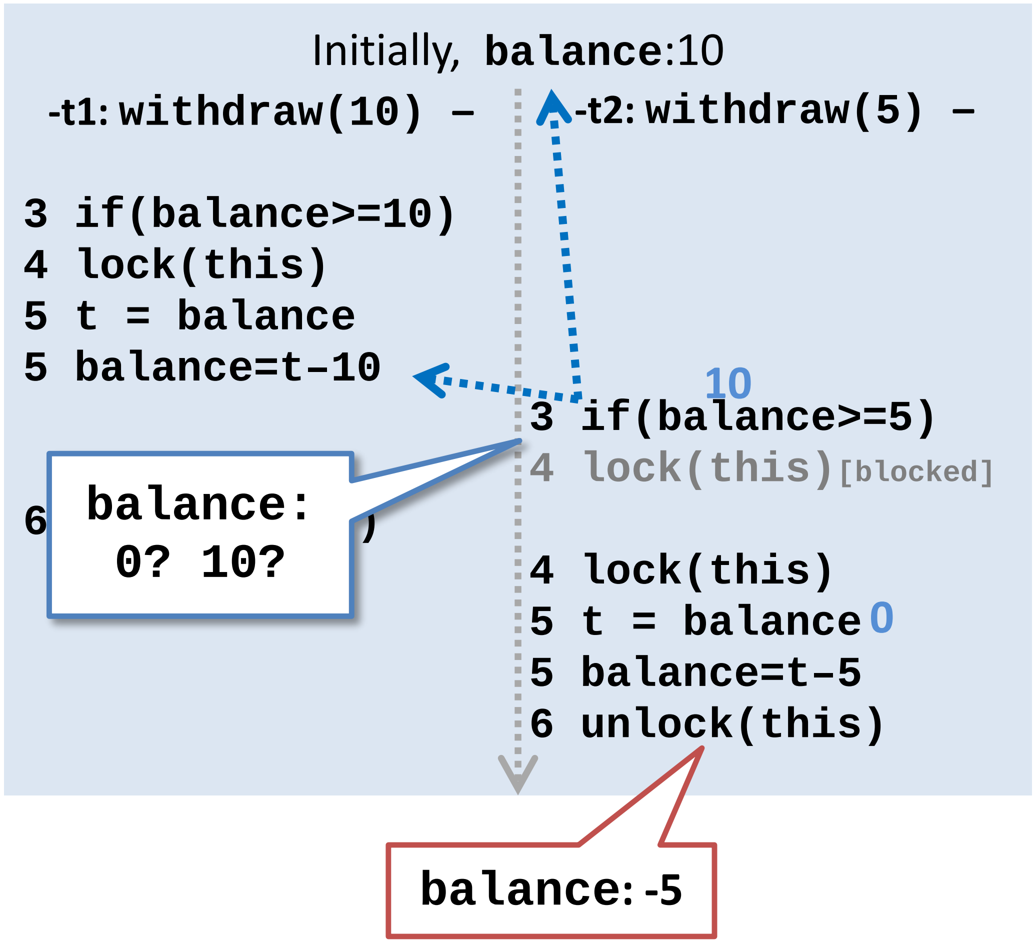
  (2) the operations of each individual processor occur in this sequence in the order specified by its program”
- Most intuitive consistency model for programmers
  - Processors see their own loads and stores in program order
  - Processors see others’ loads and stores in program order
  - All processors see same global load and store ordering

A SC preserved program is easy to understand but architects and compiler writers want to violate it for performance

Excerpts from the Prof. Huh’s lecture note on “Consistency”

Data Race Example

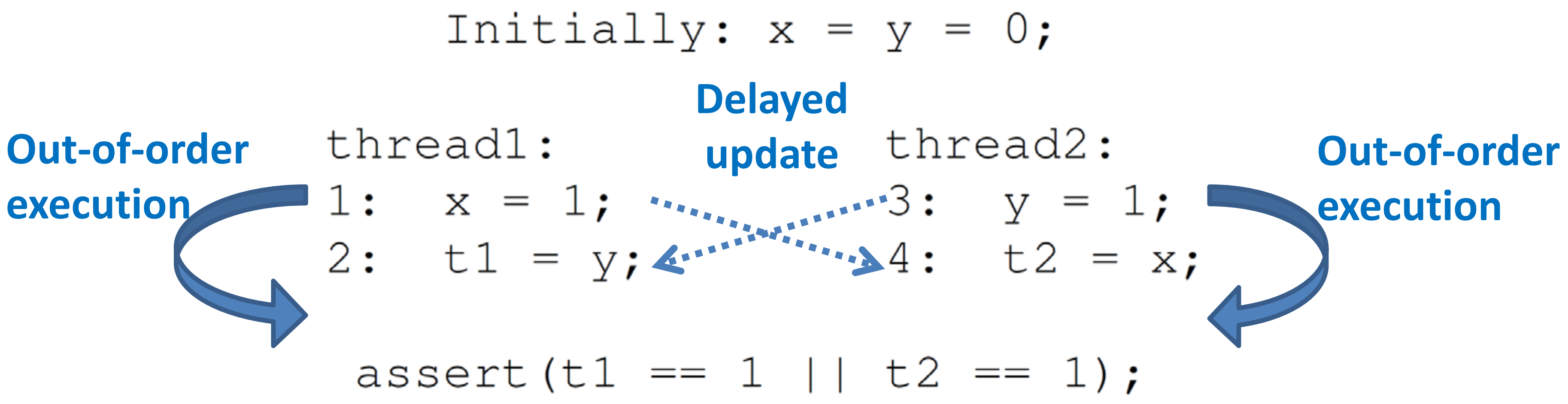
```
  class Account {
1  long balance;
   //must be non-negative

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



Initially, **balance**:10

```
-t1: withdraw(10) –                 -t2: withdraw(5) –

3 if(balance>=10)
4 lock(this)
5 t = balance
5 balance=t-10
                                    3 if(balance>=5)
                                    4 lock(this) [blocked]
6                      )
                                    4 lock(this)
                                    5 t = balance
                                    5 balance=t-5
                                    6 unlock(this)
```

Data Race can Break Sequential Consistency



```
Initially: x = y = 0;

thread1:          thread2:
1:  x = 1;        3:  y = 1;
2:  t1 = y;       4:  t2 = x;

 assert(t1 == 1 || t2 == 1);
```

Does the assertion hold with a SC memory model?

Does the assertion hold with a weak memory model?

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

Memory Model

- A **memory model** describes the interactions of threads through memory and their shared use of the data (necessary for compiler optimization)
  - A memory model allows a compiler to perform many important optimizations.
- Ex. The Java memory model (a weak memory model) stipulates that changes to the values of shared variables only need to be made visible to other threads when a synchronization barrier (e.g. lock/monitor) is reached.
  - the compiler needs to make sure only that the values of (potentially shared) variables at synchronization barriers are guaranteed to be the same in both the optimized and unoptimized code. In particular, reordering statements in a block of code that contains no synchronization barrier is assumed to be safe by the compiler.
- Most research in the area of memory models revolves around:
  - Designing a memory model that allows a maximal degree of freedom for compiler optimizations while still giving sufficient guarantees about race-free and (perhaps more importantly) race-containing programs.
  - Proving program optimizations that are correct with respect to such a memory model.
- https://en.wikipedia.org/wiki/Memory_model_(programming)

Excerpt from Wikipedia

Why is Data Race Harmful? (1/2)

- Sometimes developers intentionally induce data races for efficient read on shared variables (*benign race* or *dirty read*)
  - e.g. test-test-and-set pattern (a.k.a., double-checked locking)

```
if(balance>=x){
 synchronized(this){
 if(balance>=x){
   balance=balance-x ;
 }
}
```

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

Why is Data Race Harmful? (2/2)

However, 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

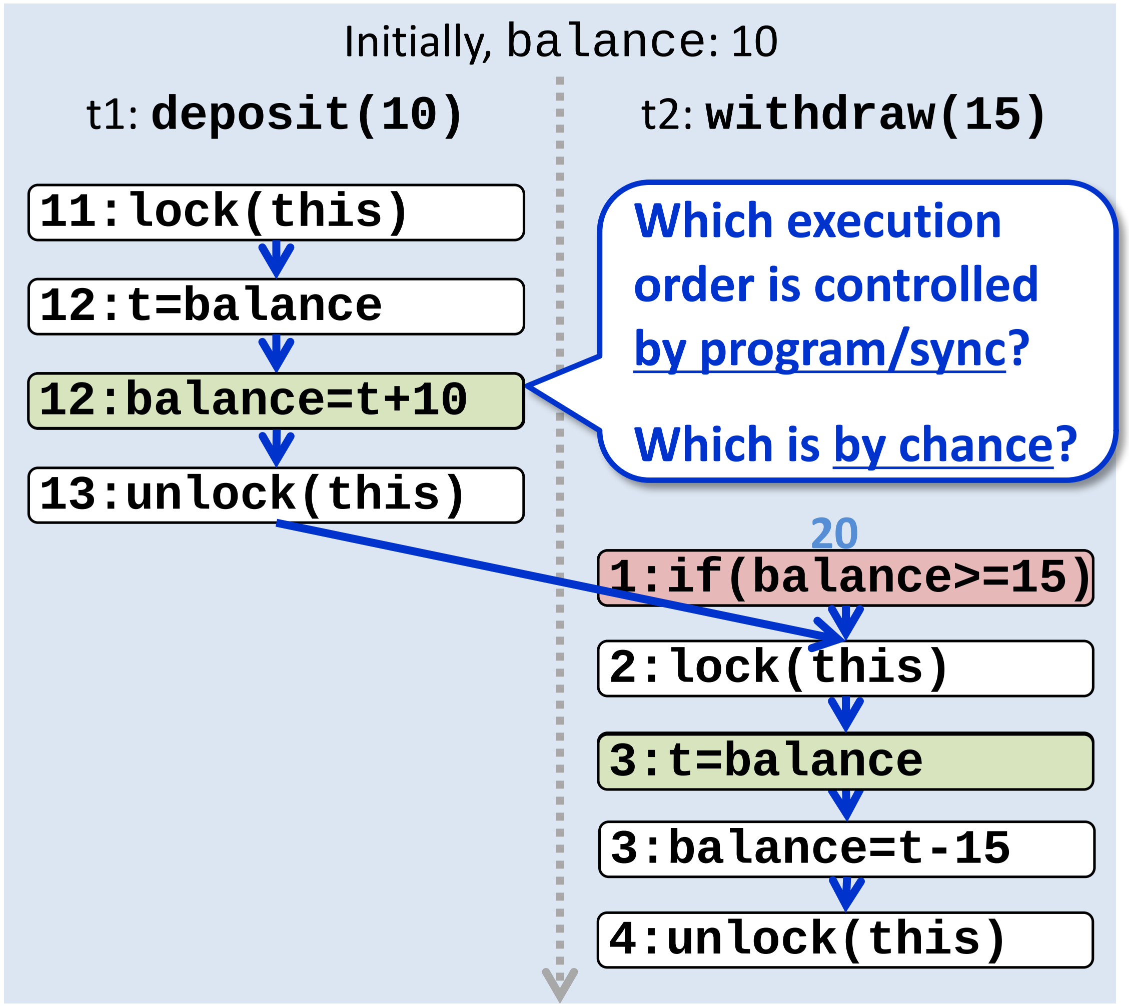
Data Race Detection/Prediction

- 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. <u>Happens-before</u> based detection technique
  2. <u>Lockset algorithm</u> based detection technique

Happens-Before Example

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

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



Happens-before Relation (1/2)

- The *happens-before relation* $\prec$ is a smallest relation over operations in an execution that satisfies the following conditions:

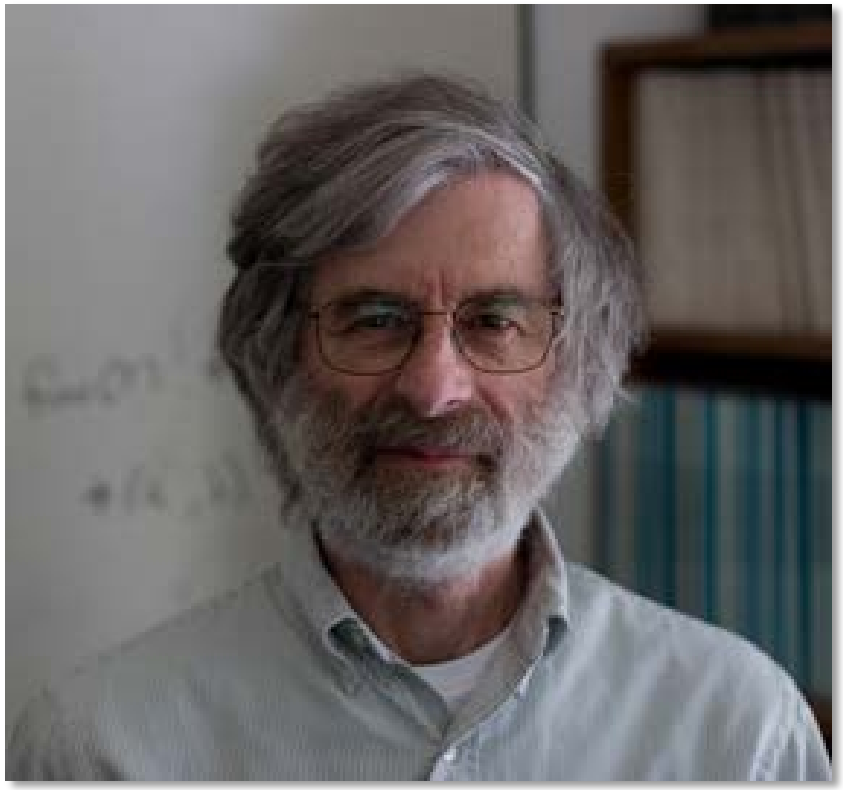
  (1) $a \prec b$ when a and b are executed by the same thread, and a comes before b

  (2) $a \prec b$ when a and b are ordered by the same synchronization entity, and a comes before b (e.g. lock, wait/notify, join)

  (3) If $a \prec b$ and $b \prec c$ then $a \prec c$

  → **a and b are *concurrent* if $a \nprec b$ and $b \nprec a$**

Happens-before Relation (2/2)

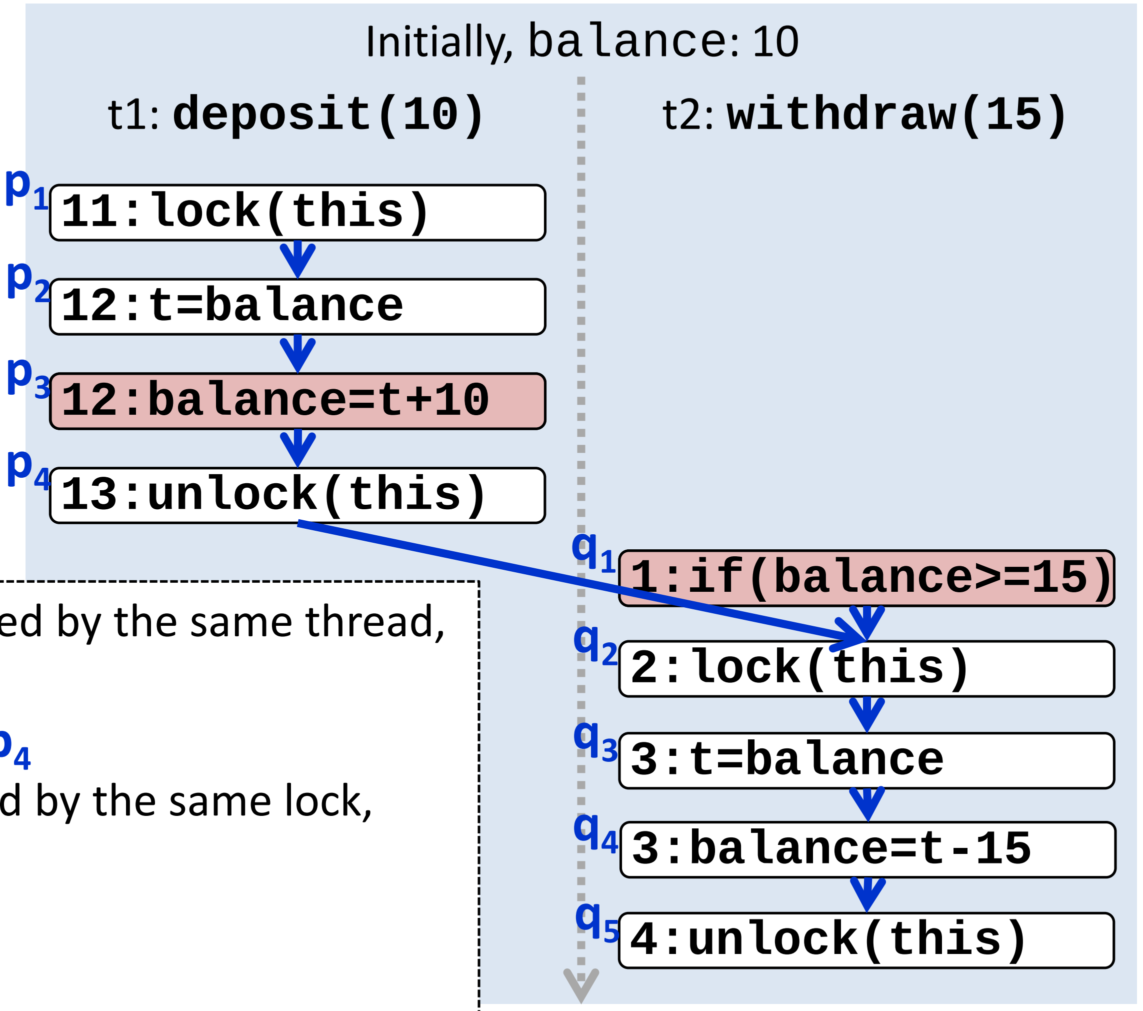


- Leslie Lamport (Microsoft research)
  - Winner of the 2013 Turing award for advances in reliability of distributed/ concurrent systems
  - Happens-before relation, sequential consistency, Bakery algorithm, TLA (temporal logic of actions), and LaTeX

http://amturing.acm.org/
http://research.microsoft.com/apps/video/default.aspx?id=210551

Happens-Before Example

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



(1) $a \prec b$ when a and b are executed by the same thread, and a comes before b

E.g. $p_1 \prec p_2$, $p_1 \prec p_3$, $p_1 \prec p_4$

(2) $a \prec b$ when a and b are ordered by the same lock, and a comes before b

E.g. $p_1 \prec q_2$

(3) If $a \prec b$ and $b \prec c$ then $a \prec c$

E.g. $p_1 \prec q_2 \land q_2 \prec q_3 \rightarrow p_1 \prec q_3$

Happens-before Based Detection (1/2)

- The pair of operations a and b is *data race* if all of the following conditions hold:
  - (1) a and b access the same variable, and
  - (2) at least one operation is writing, and
  - (3) $a \nprec b$ and $b \nprec a$

- Several tools such as MultiRace[1] and FastTrack[2] use this definition for data race detection

[1] E. Pozniansky et al.: MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs, PPoPP, 2003

[2] C. Flanagan et al.: FastTrack: Efficient and Precise Dynamic Race Detection, PLDI, 2009

Happens-before Based Detection (2/2)

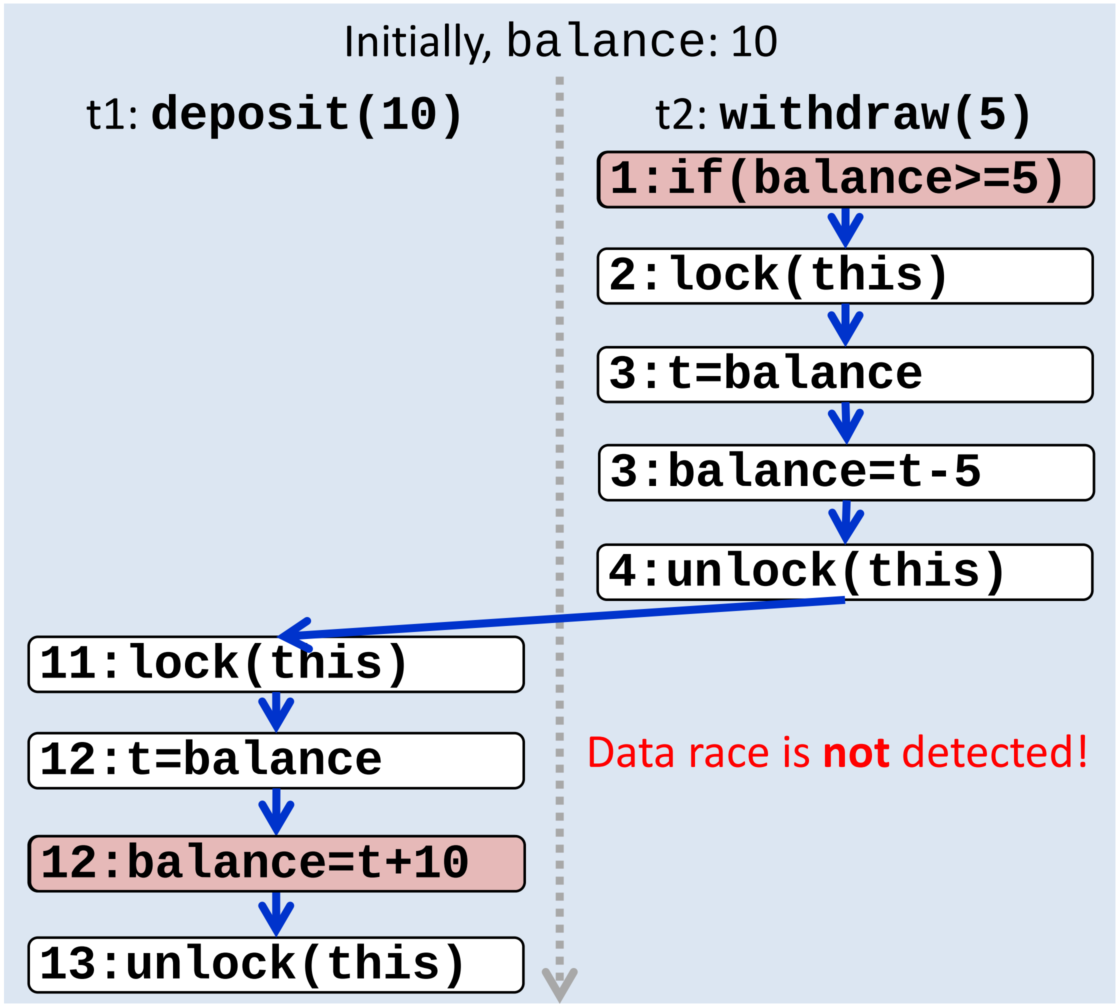
- Happens-before relation provides precise reasoning of concurrency of operations (i.e., no false positives)
- However, these techniques may or may not detect data races depending on observed execution scenario (i.e., false negatives)
- In addition, tracking happens-before relation induces heavy runtime overhead

Happens-before Based Detection (2/2)

- Happens-before relation provides precise reasoning of concurrency of operations (i.e., no false positives)
- However, these techniques may or may not detect data races depending on observed execution scenario (i.e., false negatives)
- In addition, tracking happens-before relation induces heavy runtime overhead

Another Execution Scenario

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

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



Lockset Based Data Race Detection

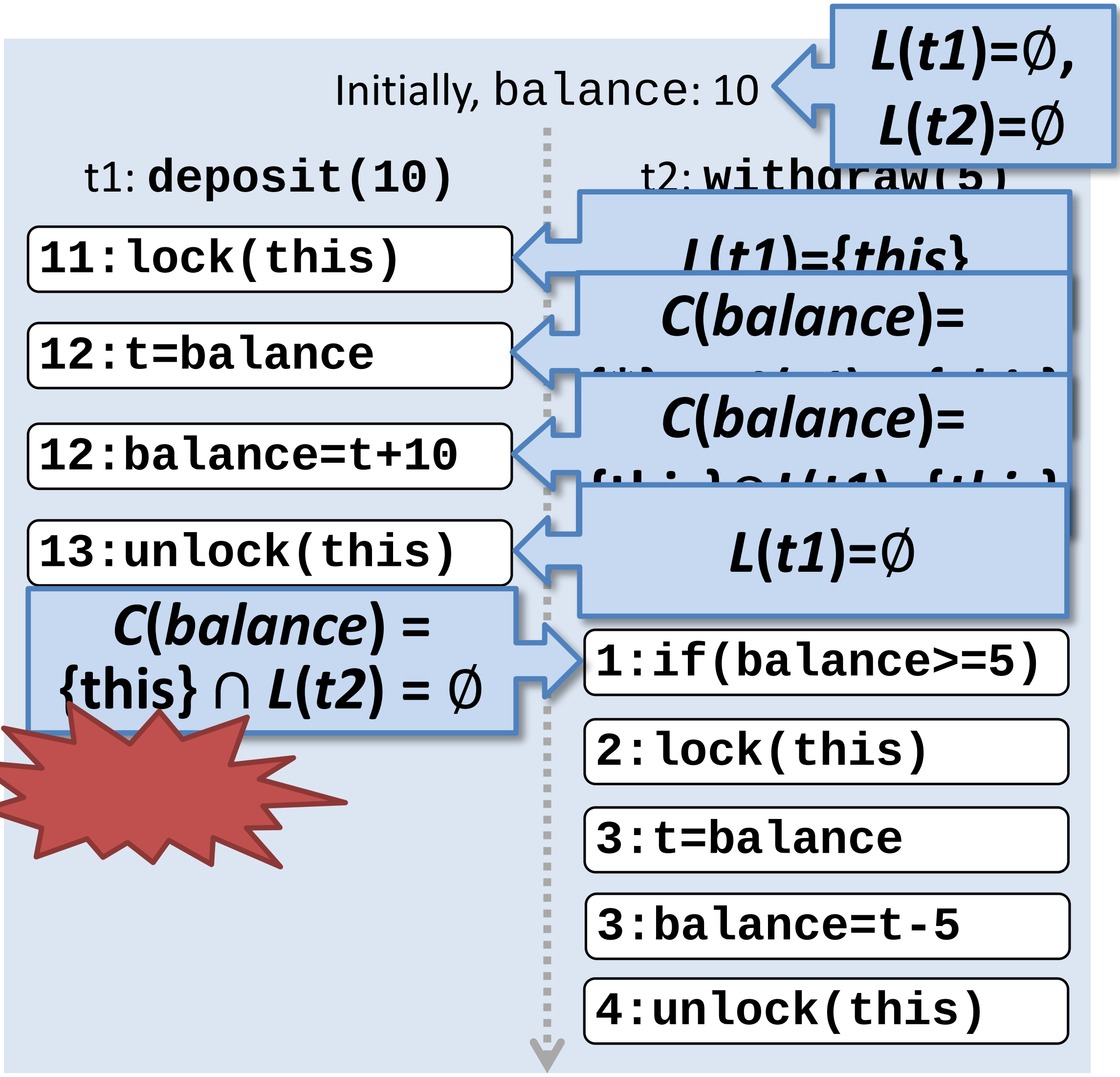
- Lock discipline
  - Every access to a shared variable MUST be guarded by at least one lock consistently
- Dynamic data race detector *Eraser* [Savage, SOSP 97]
  - Checks that every shared memory location follows the lock discipline
    - Consider memory locations for global variables, and heap memory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

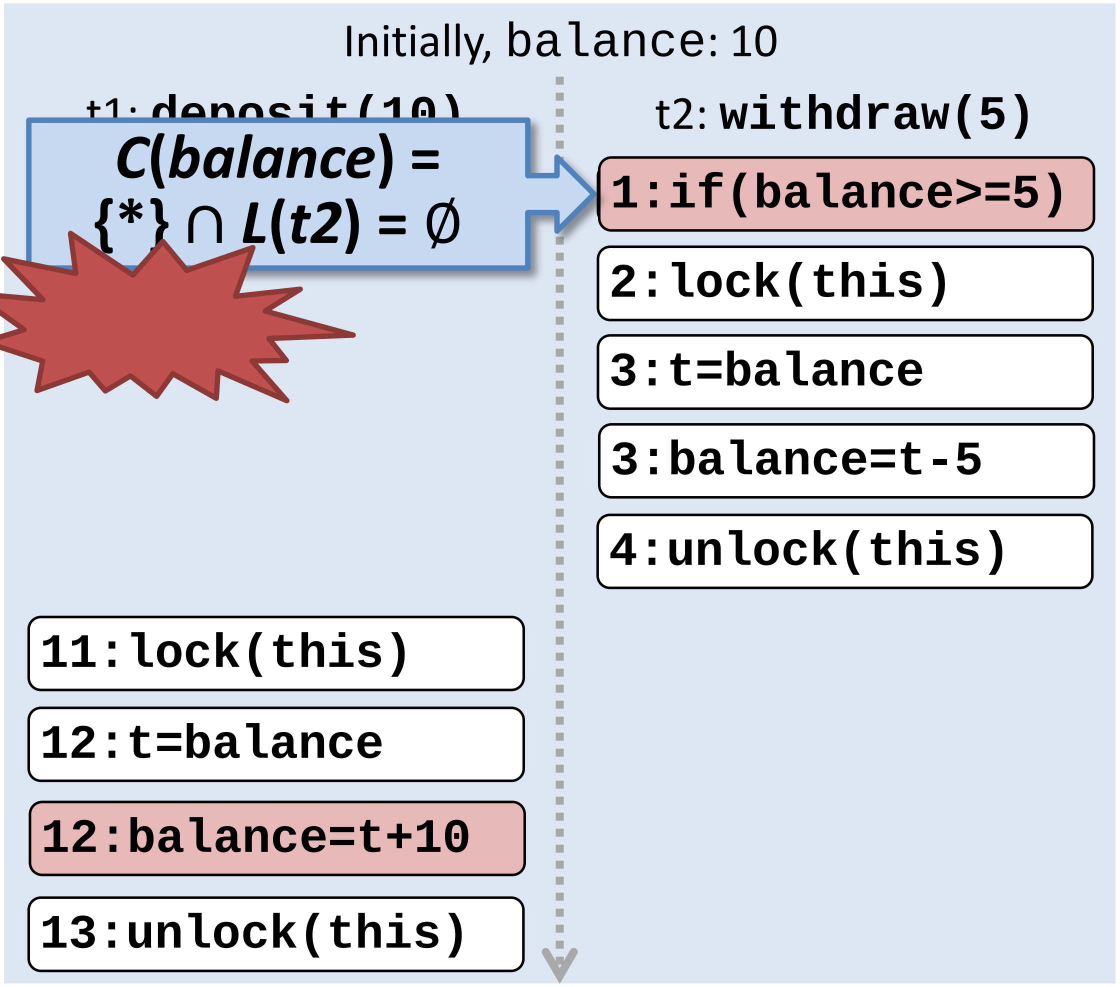
Lockset Algorithm Example

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

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



Revisiting False Negative Example

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

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



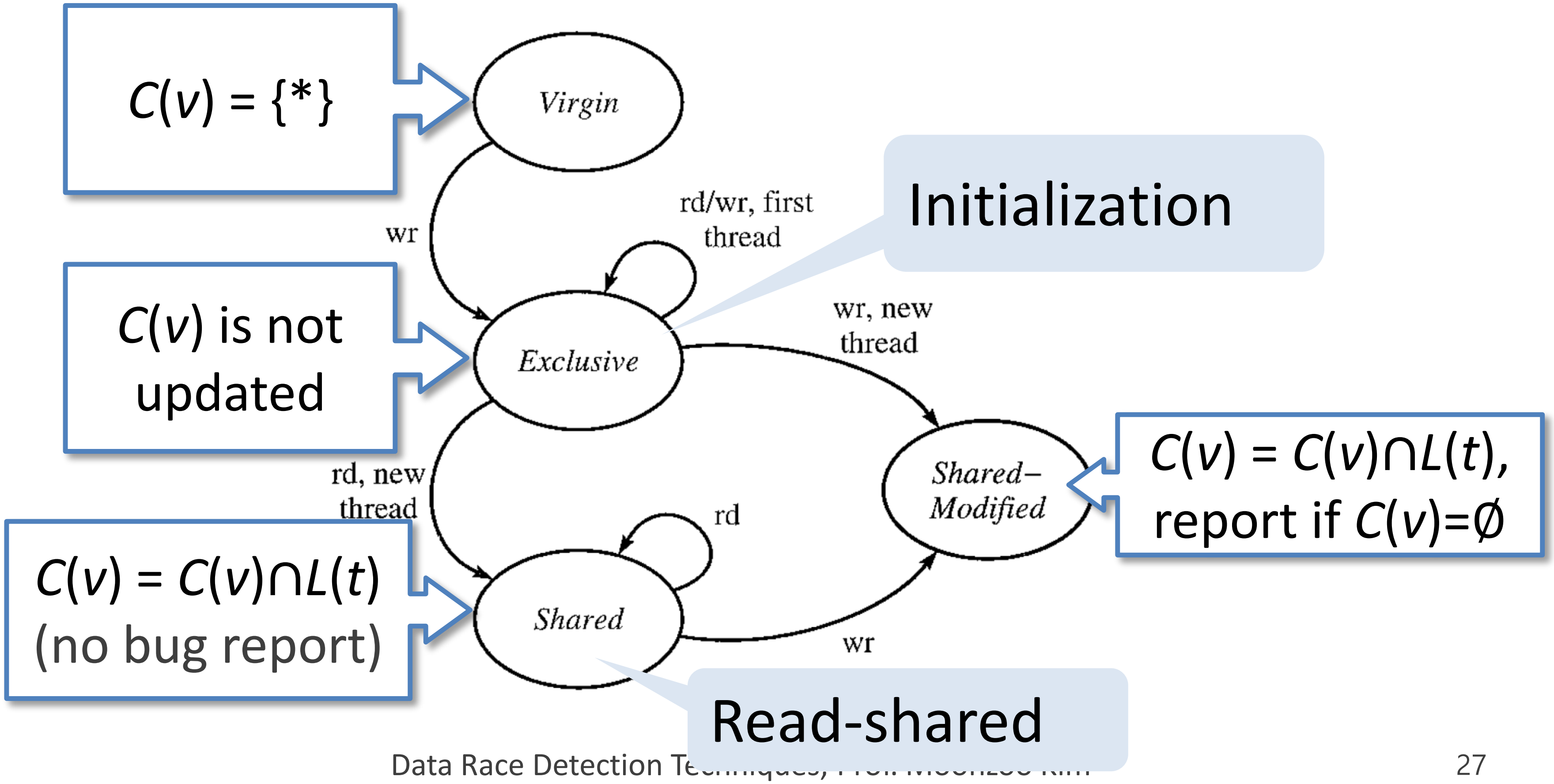
Improving Lockset Algorithm

- The naïve lockset algorithm may generate many false positives and false negatives

- 2 cases that cause false positives
  - Initialization
    - A thread writes data on the variable without locking before it makes the variable accessible by other threads
  - Read-shared variable
    - After initialization, the variable is only read, and never updated.
- 1 case that causes false negatives
  - Readers-writer lock (aka. shared-exclusive lock, or multi-reader lock)

Memory Location State

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



Example

```
int max, iter ;
Lock m ;

main(){
 iter= 10;
 max = 0 ;
 start(f);
 start(f);
}

f() {
 int i, t;
 for(i=0;i<iter;i++){
  t = input();
  lock(m);
  if (t>max)
   max = t ;
  unlock(m);
 }
}
```

t1:main()	t2:f()	t3:f()
iter= 10		
max = 0		
start(t2)		
start(t3)		
	if(0 < iter)	
	t = 10	
	lock(m)	
	if(t > max)	**if**(0 < iter)
	max = t	t = 5
	unlock(m)	
		lock(m)
	…	**if**(t>max)
		unlock(m)
		…

Example

t0	t1	t2	L(t0)	L(t1)	L(t2)	C(iter)	S(iter)	C(max)	S(max)
(initial state)									
`iter=10`									
`max = 0`									
`start``(f)`									
`start``(f)`									
	`if``(0<iter)`								
		`if``(0<iter)`							
	`lock``(m)`								
	`if``(t>max)`								
	`max = t`								
	`unlock``(m)`								
		`lock``(m)`							
		`if``(t>max)`							
		`unlock``(m)`							
	...	...							

Reducing More 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
  - For cases where malloc() reuses allocated memory
- Track all references to a memory location to precisely check if multiple threads can access the memory location
  - For cases where global variables become local (e.g., an element of a global list which is removed from the list)

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`)

Considering Readers-Writer Locks

- A thread acquires a readers-writer lock either in read-mode or write-mode
- For each variable, Eraser additionally checks if there is a lock consistently held in write-mode for write accesses
  - In Shared-Modified state
    - For each **read** 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
    - For each **write** on variable v by thread t
      - $C(v) := C(v) \cap \boldsymbol{LW(t)}$
        - » $LW(t)$ is a set of locks held in a write mode by t
      - If $C(v) = \emptyset$, report that there is a data race for v

Performance Improvement (1/2)

- Dynamic data race detection tools are still too slow to be practical
  - Intel 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

Next Class: Race Bug Which Is Not a Data Race

```
  class Account {
    long balance;
    //must be non-negative
   void getBalance(){
1   synchronized(this){
2    return balance;
3   }
   }
   void withdraw(long x){
4   if(getBalance()>x){
5    synchronized(this){
6     balance=balance-x;
7    }
    }
   }
  }
```

Initially, balance: 10

```
t1: withdraw(10)              t2: withdraw(10)
1: lock(this)
2: tmp = balance
3: unlock(this)
                              1: lock(this)
                              2: tmp = balance
                              3: unlock(this)
4: if(tmp>10)
5: lock(this)
6: tmp = balance
6: balance = tmp-10
7: unlock(this)
                              4: if(tmp>10)
                              5: lock(this)
                              6: tmp = balance
                              6: balance = tmp-10
                              7: unlock(this)
```

Data race free,
but race bug