Happens-before and Lockset Algorithm

Cristian Cadar

Department of Computing

Imperial College London

Imperial College London

Software Reliability Course

Autumn 2016

Race Conditions and Data Races

- A **race condition** occurs when multiple threads (or processes, systems, etc.) attempt to perform two or more operations at the same time and their timing affects program correctness.

- A **data race** occurs when multiple threads (or processes, systems, etc.) access **shared data** without an explicit mechanism to prevent simultaneous accesses and at least one access is a write.

Race Conditions vs Data Races

- Do data races imply race conditions?
- **NO!** There are benign races

```
int found = 0; //shared
int a[NThreads * N]; //shared

void  find(int val, int tid){
  for (i = tid*N; i < (tid+1)*N; i++) {
      if (a[i] == val)
         found = 1;
      if (found)
          break;
  }
```

Race Conditions vs Data Races

- Do data races imply race conditions?
- **NO!** There are benign races

```
int no_ops = 0; // shared

void  some_freq_op(){
    ...
    no_ops++;
}
```

Race Conditions vs Data Races

- Do absence of data races imply no race conditions?
- **NO!**

```
void  Extract(int acc_no, int sum)
{
  lock(l);
  int B = Acc[acc_no];
  unlock(l);
  lock(l);
  Acc[acc_no] = B - sum;
  unlock(l);
}
```

Finding Data Races

- While ideally one would like to find race conditions:
  - Data races are often a good proxy in practice
  - Data races have a precise definition and thus their detection can be automated

Finding Data Races

Two main approaches:

- Static analysis
- Dynamic analysis

With the trade-offs discussed before

In this course, we'll discuss **dynamic race detection**

Finding Data Races

Two main approaches:

- Static analysis
- Dynamic analysis

With the trade-offs discussed before

In this course, we'll discuss **dynamic race detection**

REVIEW OF SOME BASIC SYNCHRONIZATION PRIMITIVES

Disabling Interrupts

```
void Extract(int acc_no, int sum)
{
  CLI();
  int B = Acc[acc_no];
  Acc[acc_no] = B - sum;
  STI();
}
```

- Works only on single-processor systems
- Misbehaving/buggy processes may never release CPU
  - Mechanism usually only used by kernel code

Setting Interrupt Level

```
void Extract(int acc_no, int sum)
{
  l = SetInterruptLevel(n);
  int B = Acc[acc_no];
  Acc[acc_no] = B - sum;
  RestoreInterruptLevel(l);
}
```

- Variant in which we disable all interrupts of priority less than or equal to n

Locks

- Only one thread can **acquire** the lock at any time
- **Lock granularity:** the amount of data a lock is protecting
- **Lock contention:** a measure of the number of processes waiting for a lock
- **Read-write locks**
  - Locks can be acquired in either read or write mode
  - Multiple threads can acquire a lock in read mode
  - Only one thread can acquire a lock in write mode

Read/Write Locks

```
void ViewHistory(int acc_no)
{
    lock_RD(L[acc_no]);
    print_transactions(acc_no);
    unlock(L[acc_no]);
}
```

```
void Extract(int acc_no,
             int sum)
{
  lock_WR(L[acc_no]);
  Acc[acc_no] -= sum;
  add_debit(acc_no, sum);
  unlock(L[acc_no]);
}
```

Read/write locks:

- **`lock_RD(L)`** → acquire L in read mode
- **`lock_WR(L)`** → acquire L in write mode
- In write mode, the thread has exclusive access
- Multiple threads can acquire the lock in read mode!

Semaphores

- Blocking synchronization mechanism invented by Dijkstra in 1965
- Idea: Processes will cooperate by means of *signals*
  - A process will stop, waiting for a specific signal
  - A process will continue if it has received a specific signal
- Semaphores consists of a **counter** and a queue of processes currently waiting for that semaphore
- Unlike locks, semaphores don't have an owner

Semaphore Operations

```
init(s, i) ::= counter(s) = i
               queue(s) = {}
```

```
down(s) ::= if counter(s) > 0
               counter(s) = counter(s) - 1
            else
               add P to queue(s)
               suspend current process P
```

```
up(s) ::= if queue(s) not empty
             resume one process in queue(s)
          else
             counter(s) = counter(s) + 1
```

Semaphores: Mutual Exclusion

- **Binary semaphore:** counter is initialized to 1
- Similar to a lock

```
process A                          process B
   ...                                ...
   down(s)                            down(s)
      critical section                   critical section
   up(s)                              up(s)
end                                end

main() {
   var s:Semaphore
   ...
   init(s, 1)  /* initialise semaphore */
   ...
      start processes A and B in random order
   ...
}
```

Semaphores: Ordering Events

Process A must execute its critical section before process B can execute its critical section

```
process A
   ...
      critical section
   up(s)
end
```

```
process B
   ...
   down(s)
      critical section
end
```

```
var s:Semaphore
...
init(s, 0)  /* initialise semaphore */
...
   start processes A and B in random order
...
```

Traces

- Dynamic race detection operates on actual executions, which can be described by instruction traces:

```
lock(l); (T1)
x++; (T1)
unlock(l); (T1)
lock(l); (T2)
x++; (T2)
unlock(l); (T2)
```

Happens-before relationship

- Formulated by Leslie Lamport in 1976
- Partial order relation between instructions in a trace
- Denoted by a $\rightarrow$ b where a, b are instructions in a trace

- Consider a, b with a occurring before b in the trace:
  - If a, b are in the same thread, then a $\rightarrow$ b
  - If a is unlock(L) and b is lock(L), then a $\rightarrow$ b

    (can generalise for other synchronisation mechanisms)

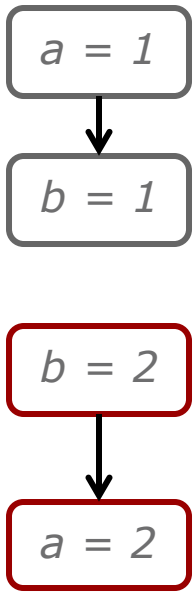
- Irreflexive: $\forall$a, a $\not\rightarrow$ a
- Antisymmetric: $\forall$a, b: a $\rightarrow$ b then b $\not\rightarrow$ a
- Transitive: $\forall$a, b, c: a $\rightarrow$ b $\wedge$ b $\rightarrow$ c then a $\rightarrow$ c

Happens-before relationship

- A data race occurs between a, b in the trace iff:
  - they access the same memory location
  - at least one of them is a write
  - they are unordered according to happens-before

Happens-before

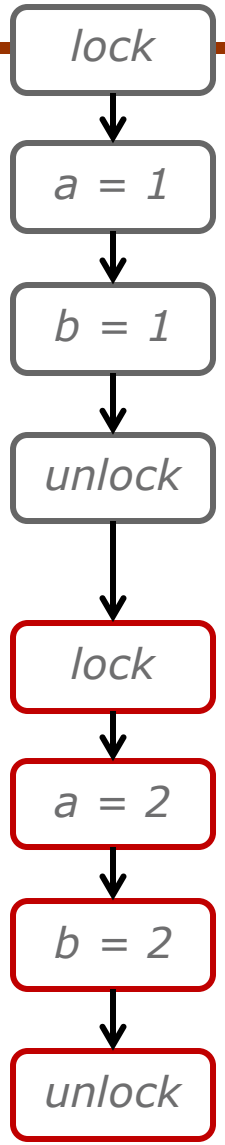
```
int a, b; // shared
void  T1()                    void T2()
{                             {
  a = 1;                        b = 2;
  b = 1;                        a = 2;
}                             }
```



Date race between a =1, a=2
and between b = 1, b =2

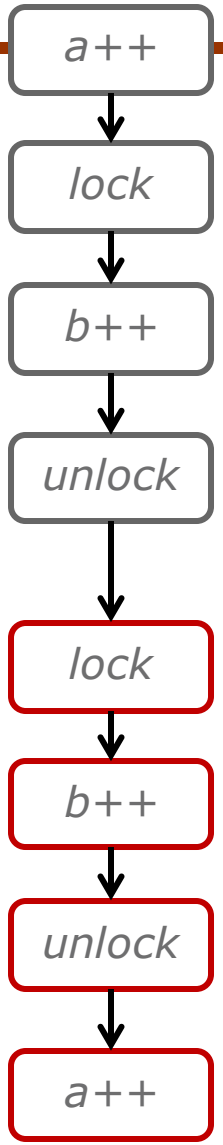
Happens-before

```
int a, b; // shared
void  T1()                  void T2()
{                           {
  lock(L);                    lock(L);
  a = 1;                      b = 2;
  b = 1;                      a = 2;
  unlock(L);                  unlock(L);
}                           }
```



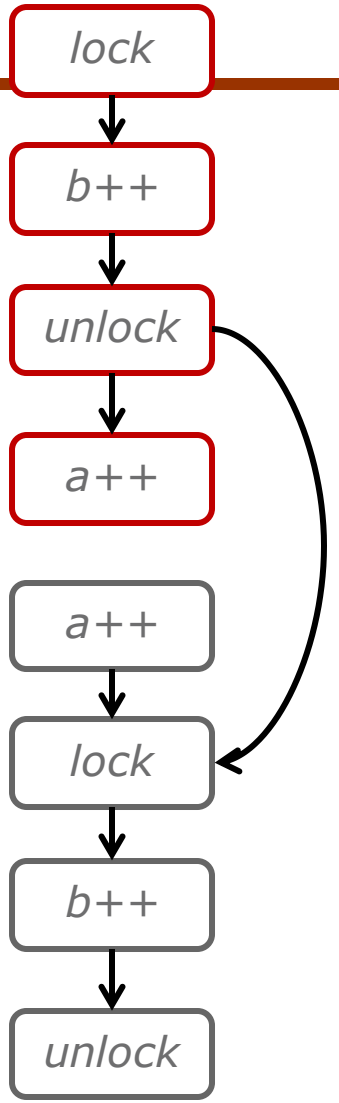
Happens-before

```
int a, b; // shared
void  T1()                  void T2()
{                           {
   a++;                        lock(L)
   lock(L);                    b++;
   b++;                        unlock(L);
   unlock(L);                  a++;
}                           }
```



Happens-before

```
int a, b; // shared
void  T1()                  void T2()
{                           {
   a++;                        lock(L)
   lock(L);                    b++;
   b++;                        unlock(L);
   unlock(L);                  a++;
}                           }
```



Lockset Algorithm and Eraser

- Happens-before is quite sensitive to the actual execution order
  - Has a high false negative rate
  - But no false positives (assuming sync primitives known)
- The lockset algorithm is an alternative dynamic analysis that
  - Has fewer false negatives than happens-before
  - But a relatively high false positive rate
- Lockset algorithm introduced in the context of Eraser by Savage et al. [SOSP 1997, TOCS 1997]
  - Recommended reading, see course website

Locking Discipline

Eraser checks the following locking discipline:

Each access to a shared variable is consistently protected by some lock L

Locking Discipline

- Does locking discipline mean no race conditions?
- **NO!**

```
void  Extract(int acc_no, int sum)
{
  lock(l);
  int B = Acc[acc_no];
  unlock(l);
  lock(l);
  Acc[acc_no] = B - sum;
  unlock(l);
}
```

Locking Discipline

- Does lack of locking discipline mean data races?
- **NO!**

T_1

```
lock(a);
lock(b);
x++;
unlock(b);
unlock(a);
```

T_2

```
lock(b);
lock(c);
x++;
unlock(c);
unlock(b);
```

T_3

```
lock(a);
lock(c);
x++;
unlock(c);
unlock(a);
```

Lockset Algorithm

locks_held(T): the set of locks currently held by thread T. Initialized to empty set.

lock(L); (T) → add L to locks_held(T)

unlock(L); (T) → remove L from locks_held(T)

C(v): the current candidate set of the locks protecting shared variable v. Initialized to the set of all locks.

On each access to v by thread T:

C(v) = C(v) ∩ locks_held(T)

if C(v) = {} issue a warning

lockset refinement

Example 1

```
lock(L);  (T1)
a++;      (T1)
b++;      (T1)
unlock(L); (T1)
   lock(L)     (T2)
   b++;        (T2)
   unlock(L);  (T2)
   a++;        (T2)
```

Example 2

```
lock(L1); (T1)
lock(L2); (T1)
a++; (T1)
unlock(L2); (T1)
  lock(L2) (T2)
  b++; (T2)
  unlock(L2); (T2)
b++; (T1)
unlock(L1)
```

Refinements

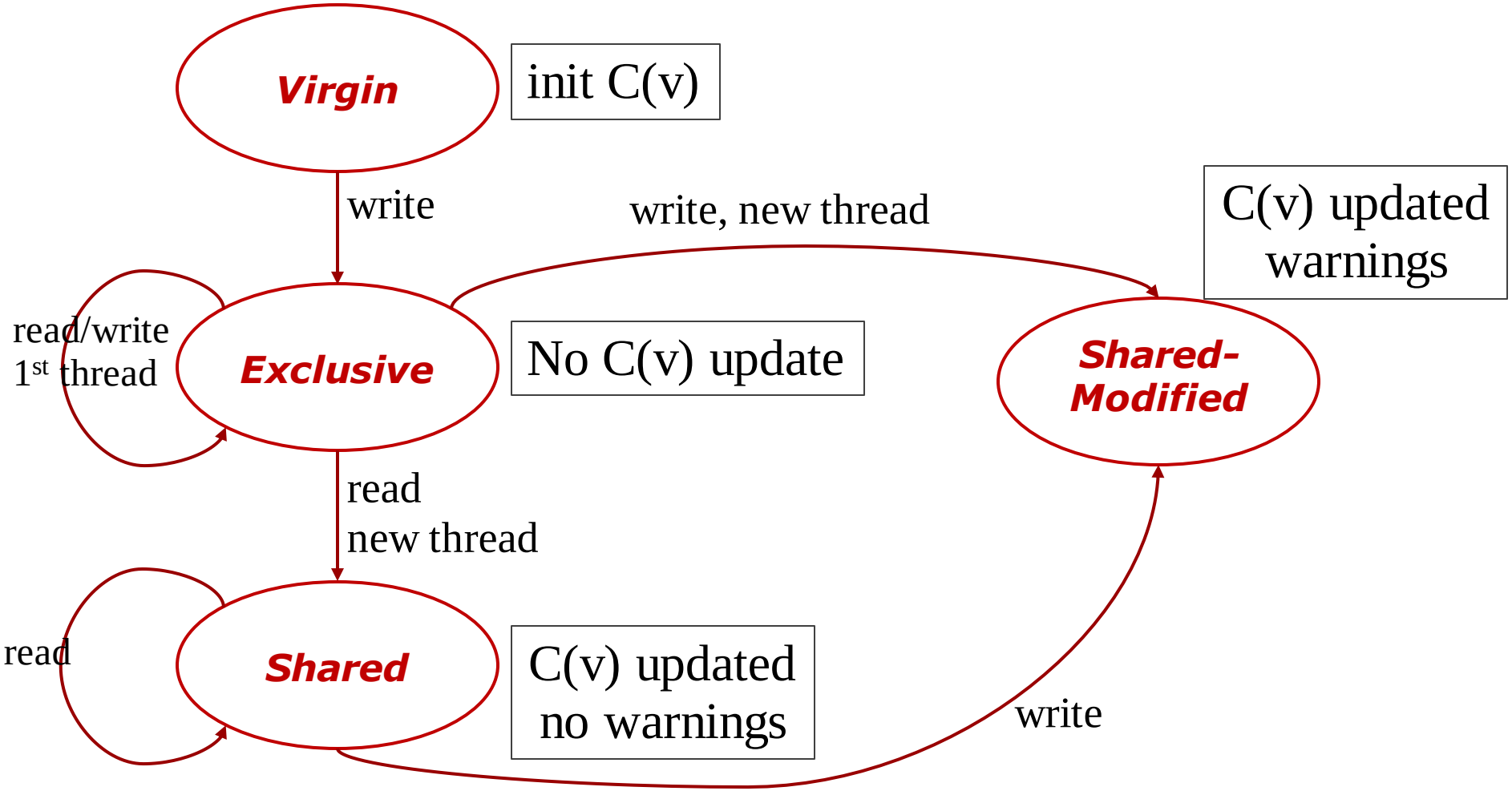
Three common programming practices that violate locking policy:

- **Initialization** can be safely done without holding a lock
- **Read-shared data**: variables are written during initialization and then shared as read-only
- **Read-write locks**: need special handling

Initialization and Read-Sharing

- No lockset refinement until after initialization
- How do we detect the end of initialization?
  - When variable first accessed by second thread
- What about unlocked read-sharing?
  - Start reporting races only when variable first written by a second thread

Eraser's state transitions



Read-Write Locks

Refined locking discipline:

For each variable v, a lock L is held in write mode for every write of v, and in either read or write mode for every read of v.

Refined Lockset Algorithm

locks_held(T): the set of locks currently held by thread T, in any mode. Initialized to {}.

wr_locks_held(T): the set of locks currently held by thread T in write mode. Init to {}.

lock_RD(L); (T) → add L to locks_held(T)

lock_WR(L); (T) → add L to locks_held(T) and wr_locks_held(T)

unlock(L); (T) → remove L from locks_held(T) and wr_locks_held(T)

Refined Lockset Algorithm

C(v): the current candidate set of the locks protecting shared variable v. Initialized to the set of all locks.

On each read to v by thread T:

```
C(v) = C(v) ∩ locks_held(T)
if C(v) = {} issue a warning
```

On each write to v by thread T:

```
C(v) = C(v) ∩ wr_locks_held(T)
if C(v) = {} issue a warning
```

Finding Races in Kernel Code

- How do you deal with Set/Restore interrupt level?
- Assign a lock to each interrupt level
- SetInterruptLevel(l) → acquire all locks associated to first n interrupt levels
- RestoreInterrupt(l) → release all locks associated to interrupt levels greater than l and up to current level

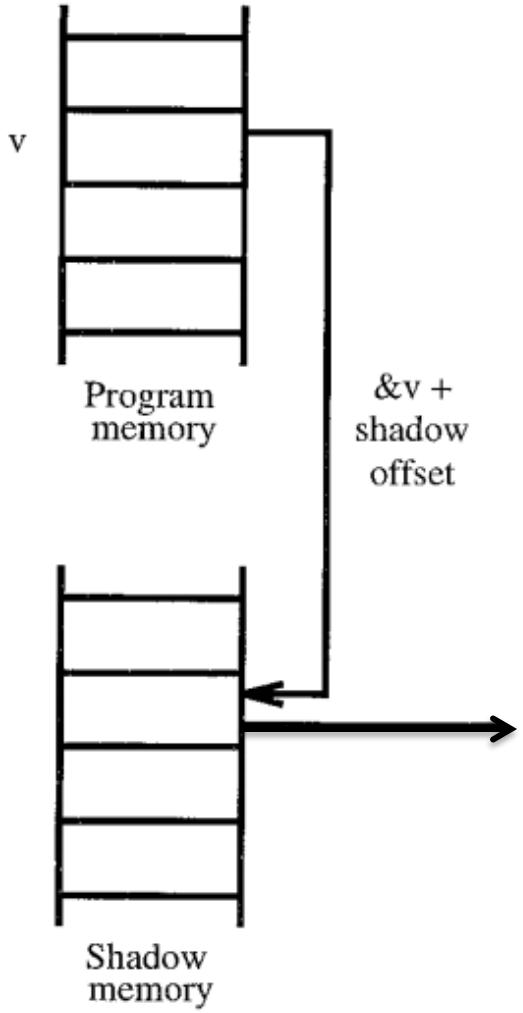
```
l = SetInterruptLevel(4); // l=2;lk(1);lk(2);lk(3);lk(4);
int B = Acc[acc_no];
Acc[acc_no] = B - sum;
RestoreInterruptLevel(l); // rl(3);rl(4);
```

Eraser – Instrumentation

- Eraser uses static binary instrumentation to intercept necessary operations (allocators, locks/unlocks, loads/stores):
  - Based on the ATOM static binary rewriting framework
  - Results in 10-30x slowdown
- What accesses to instrument?
  - Shared data assumed to be in heap or global data
  - Each 32-bit word in heap/global data assumed to be shared var

Shadow Memory

- Standard approach to keep analysis info for each memory location
  - In this case, those in heap and global segment
- In Eraser, simple scheme in which one adds a simple offset to obtain shadow word
  - Memory consumption is doubled

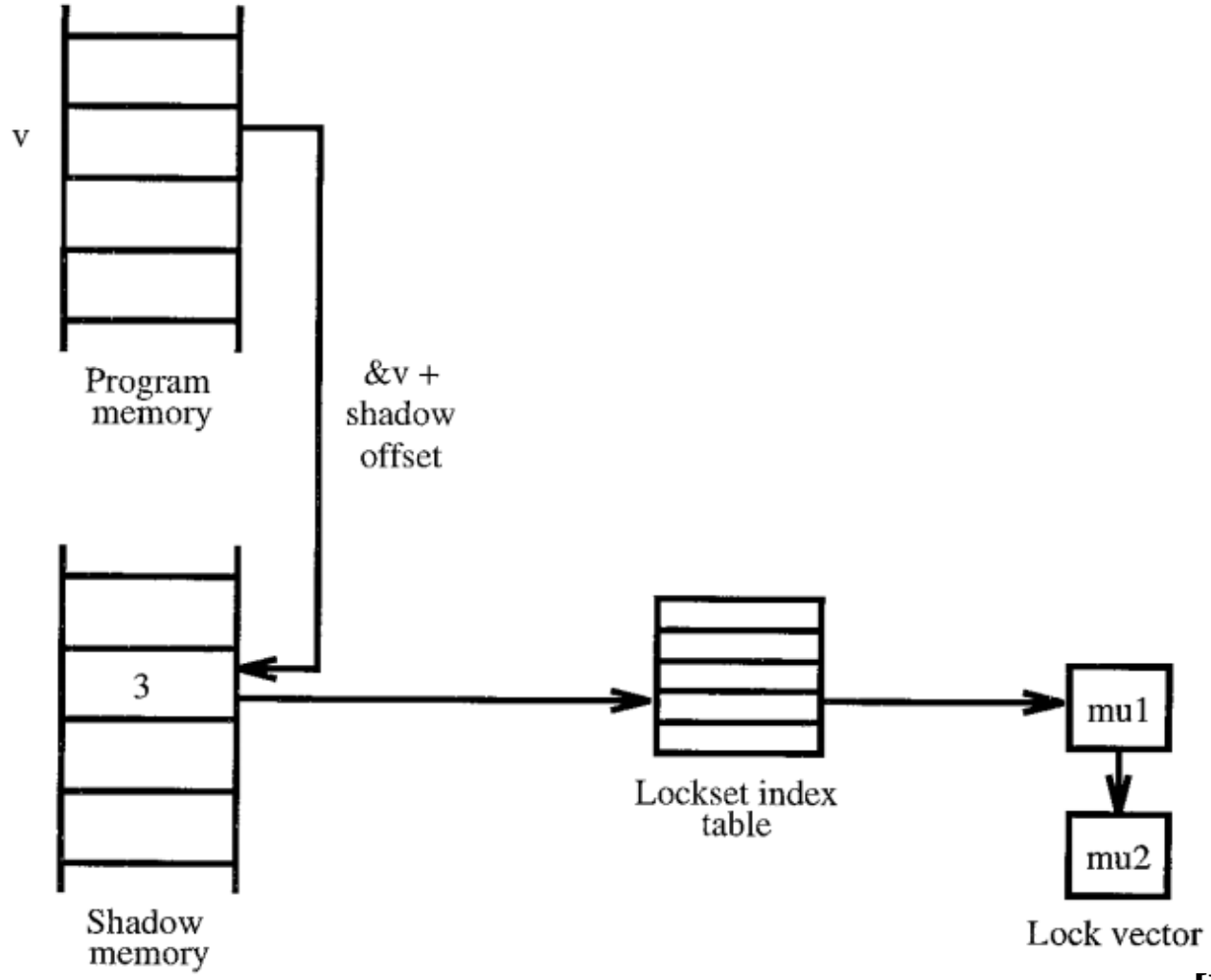


[From Eraser paper]

Eraser Implementation – Candidate Lock Sets

- Naïve implementation
  - Store list of locks for each shared memory location
  - Huge memory overhead
- In practice, number of distinct locksets small
  - < 10k in their experiments
- Map each lockset to a small integer, a "lockset index"
  - Pointing into a table whose entries represent the lockset
  - Locks sorted so that intersection is fast
  - Potentially new locksets are created after lock/unlock and lockset refinement
  - Each lockset index remains valid for the entire execution
- The result of each intersection is also cached

Eraser Implementation – Candidate Locksets



[From Eraser paper]

Evaluation

- Altavista web server and index library
  - Minor races
- Vesta cache server
  - Serious and minor races
- Petal distributed disk server
  - Minor races
- Undergraduate assignments
  - Serious races

False Positives

FPs are the main problem of Eraser, three main categories (but not all):

1) Memory reuse: due to private allocators, free lists
2) Private locks: non-standard implementation
3) Bening races: true races which don’t affect correctness

False Positives

Program annotations to eliminate FPs:

1) Memory reuse: **`EraserReuse(address, size)`**
2) Private locks: **EraserReadLock(lock), EraserReadUnlock(lock), EraserWriteLock(lock), EraserWriteUnlock(lock)**
3) Bening races: **EraserIgnoreOn(), EraserIgnoreOff()**

Needed a small number of annotations to eliminate all FPs in real apps:

- 9 in Altavista
- 10 in Vesta
- 4 in Petal

Benign race example (AltaVista)

```
if (p→ip_fp == (NI2_XFILE *) 0) {          // has file pointer been set?
    NI2_LOCKS_LOCK (&p→ip_lock);           // no? take lock for update
    if (p→ip_fp == (NI2_XFILE *) 0) {      // was file pointer set
                                           // since we last checked?
      p→ip_fp = ni2_xfopen (               // no? set file pointer
                  p→ip_name, "rb");
    }
    NI2_LOCKS_UNLOCK (&p→ip_lock);
}
                    . . .                  // no locking overhead if file
                                           // pointer is already set
```

Conclusion

- Data races vs race conditions
- Happens-before relation
  - No false positives but many false negatives
- Eraser
  - Fewer false negatives but lots of false positives
  - Based on the idea of enforcing simple locking discipline
  - Refinements for initialization, read shared and read/write locks
  - Shown effective for several real applications