# W4118: concurrency errors

Instructor: Junfeng Yang

References: Modern Operating Systems (3rd edition), Operating Systems Concepts (8th edition), previous W4118, and OS at MIT, Stanford, and UWisc

# Goals

- Identify patterns of concurrency errors (so you can avoid them in your code)
- Learn techniques to detect concurrency errors (so you can apply these techniques to your code)

# Concurrency error classification

- **Deadlock**: a situation wherein two or more processes are never able to proceed because each is waiting for the others to do something
  - Key: circular wait

- **Race condition**: a timing dependent error involving shared state
  - Data race: concurrent accesses to a shared variable and at least one access is a write
  - Atomicity bugs: code does not enforce the atomicity programmers intended for a group of memory accesses
  - Order bugs: code does not enforce the order programmers intended for a group of memory accesses

# Writing correct parallel code is hard!

- Too many schedules (exponential in execution length), hard to reason about

- Correct parallel code does not compose ➔ can't divide-and-conquer
  - Synchronization cross-cuts abstraction boundaries
  - Local correctness may not yield global correctness.

- We'll see a few error examples next

# Example 1: good + bad → bad

```
deposit() // properly sycnrhonized        withdraw() // no synchronization
    lock();
    ++ balance;                               -- balance;
    unlock();
```

- Result: race between deposit() and withdraw()

# Example 2: good + good ➔ bad

```
void deposit(Account *acnt)
{
       lock(acnt->guard);
       ++ acnt->balance;
       unlock(acnt->guard);
}

int balance(Account *acnt)
{
      int b;
      lock(acnt->guard);
      b = acnt->balance;
      unlock(acnt->guard);
      return b;
}
```

```
void withdraw(Account *acnt)
{
       lock(acnt->guard);
       -- acnt->balance;
       unlock(acnt->guard);
}

int sum(Account *a1, Account *a2)
{
       return balance(a1) + balance(a2)
}
void transfer(Account *a1, Account *a2)
{
      withdraw(a1);
      deposit(a2);
}
```

- Compose single-account operations to operations on two accounts
  - deposit(), withdraw() and balance() are properly synchronized
  - sum() and transfer()? Race

# Example 3: good + good → deadlock

```
int sum(Account *a1, Account *a2)
{
        int s;
        lock(a1->guard);
        lock(a2->guard);
        s = a1->balance;
        s += a2->balance;
        unlock(a2->guard);
        unlock(a1->guard);
        return s
}
```

| T1: | T2: |
|---|---|
| sum(a1, a2) | sum(a2, a1) |

- 2nd attempt: use locks in sum()
- One sum() call, correct
- Two concurrent sum() calls? Deadlock

# Example 4: monitors don't compose as well

```
Monitor M1 {
    cond_t cv;
    foo() {
        // releases monitor lock
        wait(cv);
    }
    bar() {
         signal(cv);
    }
};'
```

```
Monitor M2 {
    f1() {M1.foo();}
    f2() {M1.bar();}
};'
```

| T1: | T2: |
|---|---|
| M2.f1(); | M2.f2(); |

- Usually bad to hold lock (in this case Monitor lock) across abstraction boundary

# Outline

- Concurrency error patterns
- Concurrency error detection
  - Deadlock detection
  - Data race detection

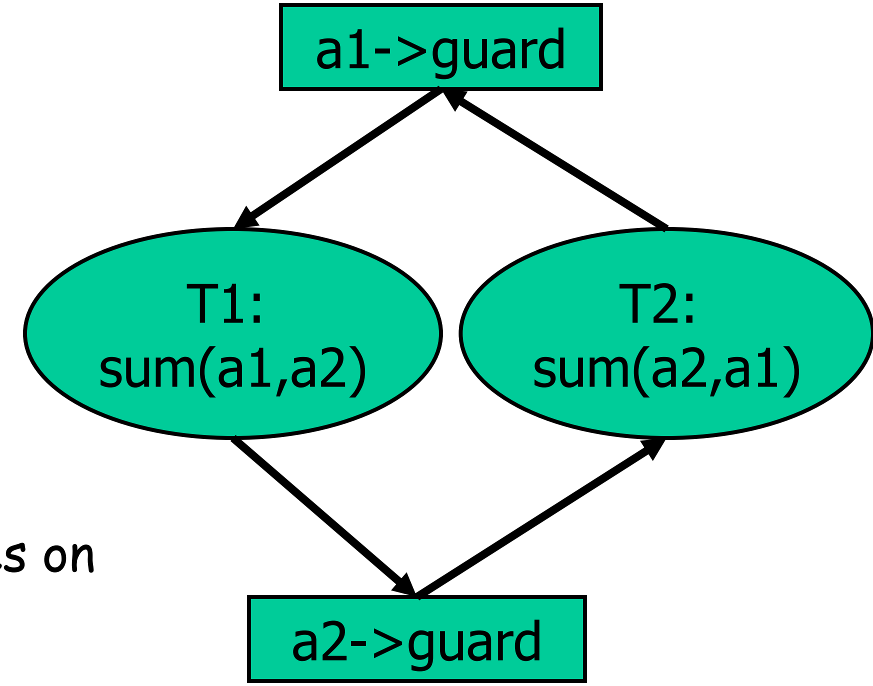
# Deadlock detection

- Root cause of deadlock: circular wait
- Detecting deadlock manually: system halts
  - Can run debugger and see the wait cycle
- Detecting deadlock automatically: resource allocation graph
- Detecting potential deadlocks automatically: lock order

# Resource allocation graph

- Nodes
  - Locks (resources)
  - Threads (processes)
- Edges
  - Assignment edge: lock->thread
    - Removed on unlock()
  - Request edge: thread->lock
    - Converted to assignment edges on lock() return
- Cycles ⇔ deadlock

- Problem: can we detect potential deadlocks before we run into them?




Resource allocation graph for example 3 deadlock

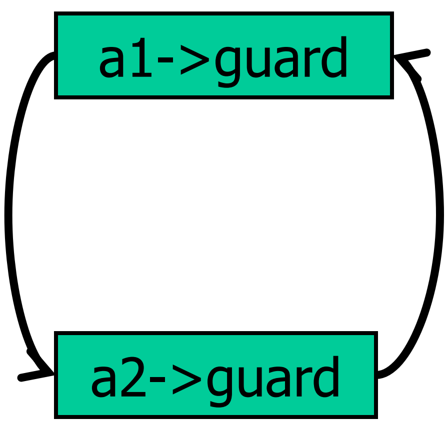
# Detecting potential deadlocks

- Can deduce lock order: the order in which locks are acquired
  - For each lock acquired, order with locks held
  - Cycles in lock order ➔ potential deadlock

```
T1:                                        
sum(a1, a2)        // locks held
lock(a1->guard) // {}
lock(a2->guard) // {a1->guard}
```

```
T2:
sum(a2, a1)        // locks held

lock(a2->guard)  // {}
lock(a1->guard) // {a2->guard}
```




Cycle ➔ Potential deadlock!

Avoid deadlock: stick to a total order of locks

# Outline

- Concurrency error patterns
- Concurrency error detection
  - Deadlock detection
  - Data race detection

# Race detection

- We will look at only data race detection
  - Techniques exist to detect atomicity and order bugs, but we won't discuss them in this class

- Two approaches to data race detection
  - Happens-before
  - Lockset (Eraser's algorithm)

# Happens-before definition

- Event A happens-before event B if
  - B follows A in the same thread
  - A inT1, and B inT2, and a synchronization event C such that
    - A happens in T1
    - C is after A in T1 and before B in T2
    - B in T2

# Happens-before race detection

- Tools before eraser are based on happens-before
- Sketch
  - Monitor all data accesses and synch operations
  - Watch for
    - Access of v in thread T1
    - Access of v in thread T2
    - No synchronization operation between the accesses
    - One of the accesses is write

# Problems with happens-before

- Problem I: expensive
  - Requires per thread
    - List of accesses to shared data
    - List of synch operations
- Problem II: false negatives
  - Happens-before looks for actual data races (moment in time when multiple threads access shared data w/o synchronization)
  - Ignores programmer intention; the synchronization op between accesses may happen to be there

```
T1:              T2:

++ y
lock(m)
unlock(m)
         ----->  lock(m);
                 unlock(m);
                 ++ y;
```

# Eraser: a different approach

- Idea: check invariants
  - Violations of invariants ➔ likely data races
- Invariant: the locking discipline
  - Assume: accesses to shared variables are protected by locks
  - Every access is protected by at least one lock
  - Any access unprotected by a lock ➔ an error
- Problem: how to find out what lock protects a variable?
  - Linkage between locks and variables undeclared

# Lockset algorithm: infer the locks

- Intuition: it must be one of the locks held at the time of access

- C(v): a set of candidate locks for protecting v
- Initialize C(v) to the set of all locks
- On access to v by thread t, refine C(v)
  - C(v) = C(v) ^ locks_held(t)
  - If C(v) = {}, report error

- Sounds good!  But …

# Implementing eraser

- Binary tool
  - Pros: does not require source
  - Cons: lose source semantics
    - Track memory access at word granularity
- How to monitor memory access?
  - Binary instrumentation
- How to track lockset efficiently?
  - A shadow word for each memory word
  - Each shadow word stores a lockset index
  - A table maps lockset index to a set of locks
  - Assumption: not many distinct locksets

# Results

- Eraser works
  - Find bugs in mature software
  - Though many limitations
    - Major: benign races (intended races)
- However, slow
  - Monitoring each memory access: costly, 10-30X slowdown
  - Can be made faster
    - With static analysis
    - Smarter instrumentation (e.g., sampling)
- Lockset algorithm is influential, used by many tools
  - E.g. Helgrind (a race detection tool in Valgrind)

# Backup slides

# Benign race examples

- Double-checking locking
  - Faster if v is often 0
  - Doesn't work with compiler/hardware reordering

```
if(v) { // race
        lock(m);
        if(v)
                ...;
        unlock(m);
}
```

- Statistical counter
  - ++ nrequests

# Problems w/ simple lockset algorithm

- Initialization
  - When shared data is first created and initialized
- Read-shared data
  - Shared data is only read (once initialized)
- Read/write lock
  - We've seen it last week
  - Locks can be held in either write mode or read mode

# Initialization

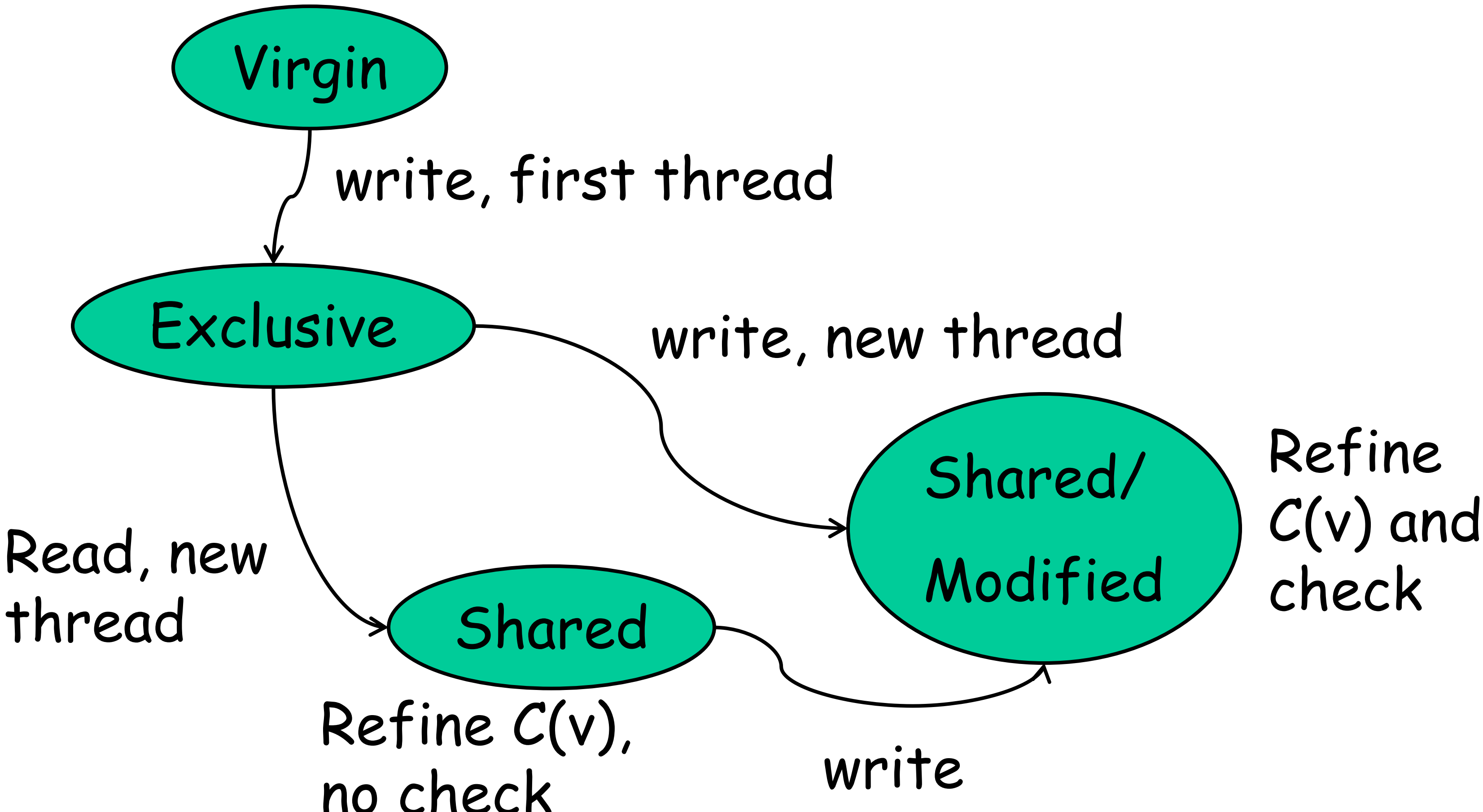
- When shared data first created, only one thread can see it ➔ locking unnecessary with only one thread
- Solution: do not refine $C(v)$ until the creator thread finishes initialization and makes the shared data accessible by other threads
- How do we know when initialization is done?
  - We don't …
  - Approximate with when a second thread accesses the shared data

# Read-shared data

- Some data is only read (once initialized) ➔ locking unnecessary with read-only data
- Solution: refine C(v), but don't report warnings
  - Question: why refine C(v) in case of read?
  - To catch the case when
    - C(v) is {} for shared read
    - A thread writes to v

# State transitions

- Each shared data value (memory location) is in one of the four states



# Read-write locks

- Read-write locks allow a single writer and multiple readers

- Locks can be held in read mode and write mode
  - read_lock(m); read v; read_unlock(m)
  - write_lock(m); write v; write_unlock(m)

- Locking discipline
  - Lock can be held in some mode (read or write) for read access
  - Lock must be held in write mode for write access
    - A write access with lock held in read mode ➔ error

# Handling read-write locks

- Idea: distinguish read and write access when refining lockset

- On each read of v by thread t (same as before)
  - C(v) = C(v) ^ locks_held(t)
  - If C(v) = {}, report error

- On each write of v by thread t
  - C(v) = C(v) ^ write_locks_held(t)
  - If C(v) = {}, report error

# Automatic software error detection

- Static analysis: inspect the code/binary without actually running it
  - E.g., gcc does some simple static analysis
    - $ gcc –Wall
- Dynamic analysis: actually run the software
  - E.g. testing
    - $ run-test

- Static v.s. dynamic
  - Static has better coverage, since compiler sees all code
  - Dynamic is more precise, since can see all values

- Which one to use for concurrency errors?