CS302

Locks, Deadlocks, Livelocks

Spring 2025

Arkaprava Basu & Babak Falsafi

parsa.epfl.ch/course-info/cs302

Adapted from slides originally developed by Profs. Falsafi, Fatahalian, Mowry, Wenisch of CMU, Michigan

Where are We?

M	T	W	T	F
17-Feb	18-Feb	19-Feb	20-Feb	21-Feb
24-Feb	25-Feb	26-Feb	27-Feb	28-Feb
3-Mar	4-Mar	5-Mar	6-Mar	7-Mar
10-Mar	11-Mar	12-Mar	13-Mar	14-Mar
17-Mar	18-Mar	19-Mar	20-Mar	21-Mar
24-Mar	25-Mar	26-Mar	27-Mar	28-Mar
	1-Apr	2-Apr	3-Apr	4-Apr
7-Apr	8-Apr	9-Apr	10-Apr	11-Apr
14-Apr	15-Apr	16-Apr	17-Apr	18-Apr
21-Apr	22-Apr	23-Apr	24-Apr	25-Apr
28-Apr	29-Apr	30-Apr	1-May	2-May
5-May	6-May	7-May	8-May	9-May
12-May	13-May	14-May	15-May	16-May
19-May	20-May	21-May	22-May	23-May
26-May	27-May	28-May	29-May	30-May

- Synchronization
  - Locks with hardware support
  - Deadlocks
- Exercise session
  - Free session
  - Ask doubts and questions to TAs and SAs
- Next Tuesday:
  - Midterm exam

Midterm Exam

- The midterm will cover upto and including Lec 6.2
  - Accounts for 20% of final grade

- The exam will be from 10:15 AM – 12:45 PM on 8th April (Tuesday)
  - Will be held in rooms CM 1 3 and INF 2
  - Exam duration will be 2 hrs 30 mins
  - Seating arrangement will be announced via Moodle
  - Please make sure to bring your Camipro card

FAQs on Ed

- Can instructions inside the LSQ bypass each other?
  - All instructions inside the LSQ are speculative
  - Instructions in the LSQ (and ROB) and can execute out of order but they commit in order
  - Instructions can only bypass other instructions the store buffer based on the memory consistency model
- In PC and WC which instructions can bypass each other?
  - In PC, loads can bypass **committed** stores and never other loads
  - In WC, committed stores can also bypass other committed stores in addition to PC

Reminder: Simple TS Lock

- Lock:

```
void Lock(int* lock){
  while (Test_and_Set(lock) != 0);
}
```

- Unlock:

```
void Unlock(volatile int* lock){
  *lock = 0;
}
```

Simple TS Lock Performance Characteristics

- Low latency (under low contention)
- High coherence traffic
- Poor scaling
- Low storage cost (one int)
- No provisions for fairness

Test-and-Test-and-Set (TTS) Lock

- Lock:

```
void Lock(int* lock){
  while (1) {
    while (*lock != 0); // additional test
    // while another processor has the lock…

    // when lock is released, try to acquire it
    if (Test_and_Set(lock) == 0)
      return;
  }
}
```

- Unlock:

```
void Unlock(volatile int* lock){
  *lock = 0;
}
```

Test-and-Test-and-Set (TTS) Lock

- Reduce coherence traffic by testing locally
  - Read from local cache without invalidating others
  - BusRd instead of BusRdx

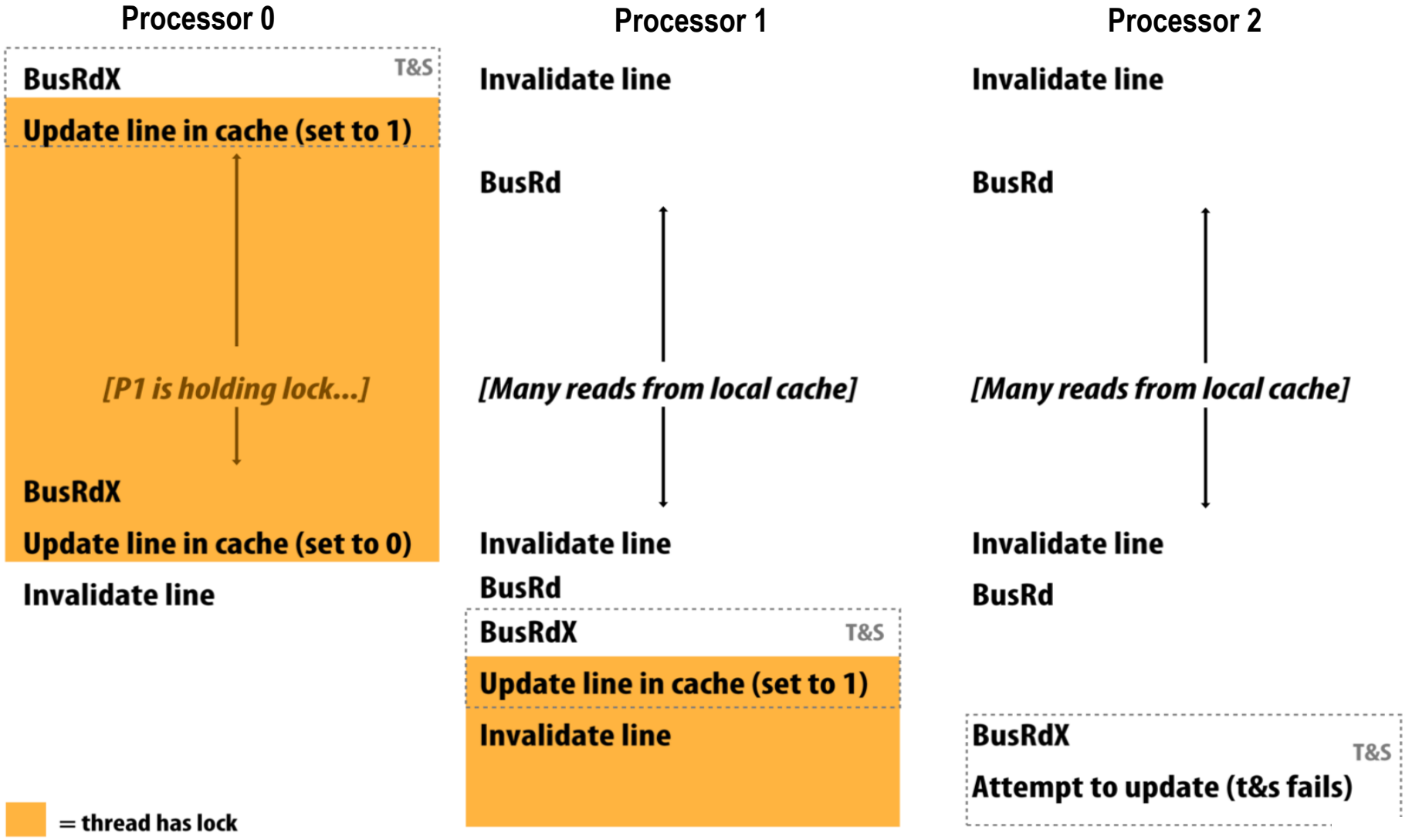
```
Test:
    ld   r1, mem[addr]        // load lock value
    cmp  r1, #0
    bnz  Test                 // compare to 0

Lock:
    ts   r1, mem[addr]        // load word into R1
    bnz  Test                     // if 0, lock obtained


Unlock:
    st   mem[addr], #0        // store 0 to
```

Test-and-Test-and-Set (TTS) Lock

- If no contention
  - To lock, read, if hit and available then RMW
  - If miss or not available, spin reading
  - To unlock, if hit, store
  - If miss, get A with write permission, then store

- With contention (multiple processors spin)
  - Most tests are read hits
  - Writes only when acquiring the lock & releasing the lock
  - Wait using reads

Test-and-Test-and-Set (TTS) Lock



TTS Lock Performance Characteristics

- Slightly higher latency than TS in uncontended case
  - Must test …. then test-and-set
- Generates much less interconnect traffic
  - One invalidation, per waiting processor, per lock release
  - O(P) invalidations = O(2*P traffic); P = #waiting cores
  - Recall: TS generated 1 invalidation per waiting processor per test
- More scalable (due to less traffic)
- Storage cost unchanged (one int)
- Still no provisions for fairness

TS Lock with Exponential Back-off

- Reduce contention traffic by adding “wait time” after failing to acquire the lock
  - Studies shown that using exponential waiting balances lock latency and contention

```
void Lock(volatile int* lock) {
  int amount = 1;
  while (1) {
    if (Test_and_Set(lock) == 0)
      return;
    delay(amount);
    amount *= 2;
  }
}
```

TS Lock with Exponential Back-off

- If no contention
  - Acts like Test & Set
- With contention (multiple processors spin)
  - Every test is a store miss!!!!
  - Every unlock is a store miss
  - But the tests are exponentially distributed

TS w/ Exp. Back-off Characteristics

- Same uncontended latency as TS
  - But potentially higher latency under contention; why?
- Generates less traffic than TS
  - Not continually attempting to acquire lock
- Improves scalability (due to less traffic)
- Storage cost unchanged
- Exponential back-off can cause severe unfairness
  - Newer requesters back off for shorter intervals

Issues w. TS and TTS

- Both locks require an instruction that is both a read and write
  - This is a challenge for pipelined processors
- Additionally, must lock the bus (block all ops) until the read-modify-write completes
  - Complicates cache controller logic
  - Two memory accesses per instruction more difficult to track in CPU pipeline

Alternative: Load-Locked & Store Conditional

- Provides atomic read-modify-write with two instrs.
  - Load-locked (LL) = `l.l` r_x`, mem[addr]`
  - Store-conditional (SC) = `s.c` r_x`, mem[addr]`
- Interacts with cache-coherence protocol to guarantee no intervening writes to `[addr]`
- Used in MIPS, DEC Alpha, and all ARM cores

How is LL/SC Atomic?

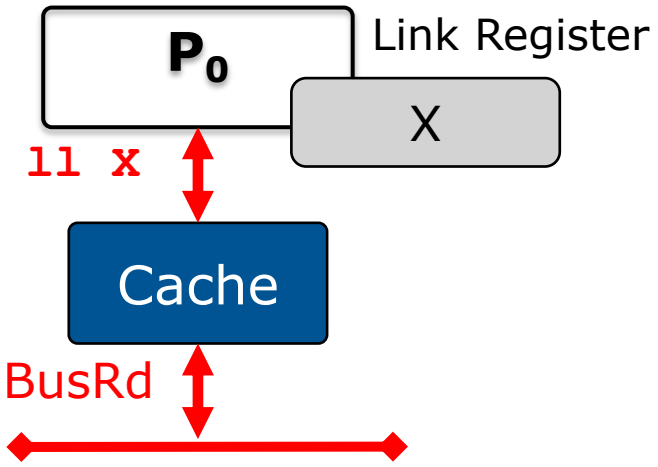
- Recall the **incorrect** first attempt:
  - Two cores could both see the lock as free, and enter the critical section

```
Lock:
     ld   r1, mem[addr]          // load word into r1
     cmp  r1, #0                 // if 0, store 1
     bnz  Lock                   // else, try again
     st   mem[addr], #1

Unlock:
     st   mem[addr], #0          // store 0 to address
```

- How does LL/SC solve the problem?

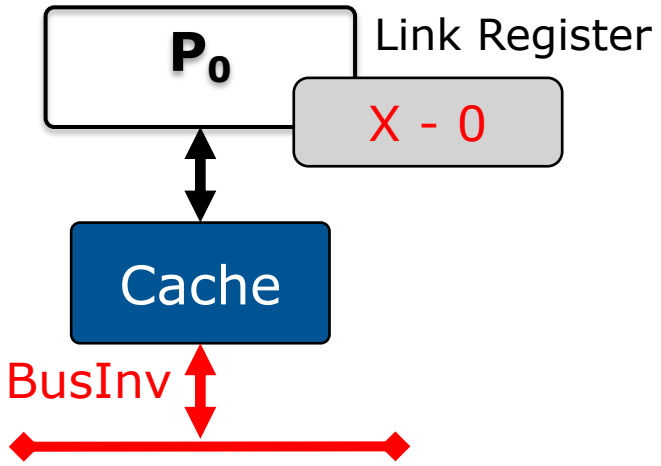
Remember and Validate the Address

- LL puts the address and flag into a link register



Remember and Validate the Address

- LL puts the address and flag into a link register
  - Invalidations or evictions for that address clear the flag, and the SC will then fail
  - Signals that another core modified the address



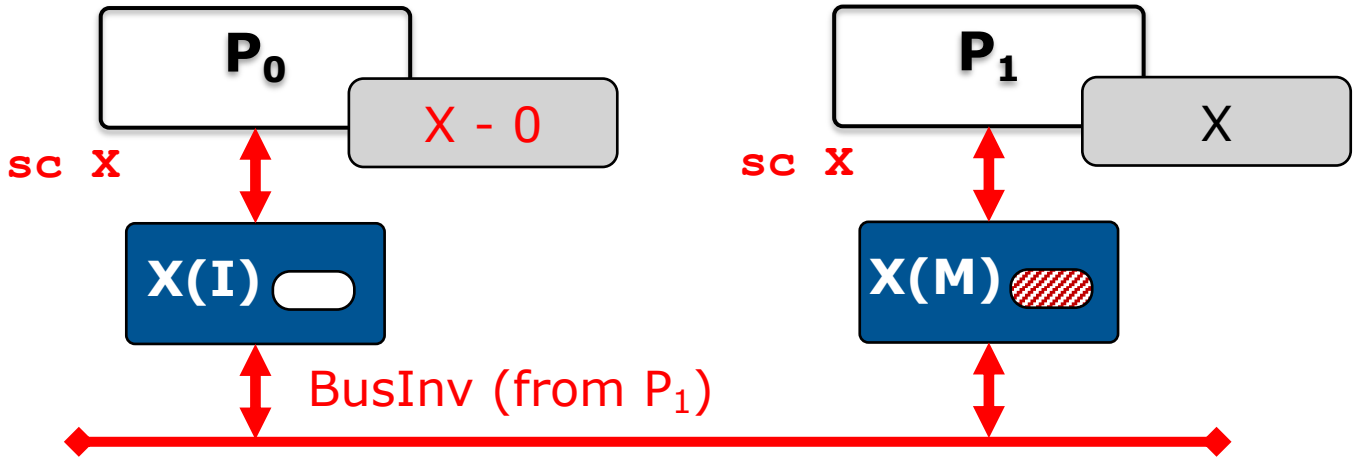
Simultaneous SCs?

- Consider the following case:
  - Processors 0 and 1 both execute the following code, with cache block X beginning in Shared
  - Both `ll [X]` read 0
  - Both begin to issue `sc [X]`
  - Will we break mutual exclusion?

```
Lock:
     ll r2, [X]
     cmp r2, #0
     bnz Lock          // if 1, spin
     addi r2, #1
     sc [X], r2
```

Simultaneous SCs?

- Will we break mutual exclusion?
  - Answer: No! Why?
    - Remember, cache coherence ensures the propagation of values to a single address.
  - So, when both processors try to BusInv, one of them will “win”, and clear the other’s link register flag
    - e.g., Say P1 wins, therefore P0’s copy of X gets invalidated, and its store will fail as well



Example: TS with LL/SC in ARM

- Can use LL/SC to build many types of locks
- This example builds TS in real ARM assembly

```
Test_and_Set:
  ldl_l   t0, 0(t1)          ; load locked, t0 = lock
  bn      t0, L1             ; if not free, loop
  lda     t0, 1(0)           ; t0 = 1
  stl_c   t0, 0(t1)          ; conditional store,
                             ; lock = 1
  beq     t0, Test_and_Set   ; if failed, loop

Release:
  stl     0, 0(t1)
```

Busy Waiting

- So far threads were spinning while waiting (busy waiting)
  - Keeping the CPU busy while making no progress

```
while(test is not successful) {
    // wait while spinning
}

compute_assuming_test_succeeded();
```

- Free up exec. resources if progress cannot be made

Blocking Synchronization

- Idea:
  - If progress cannot be made because a resource cannot be acquired
  - It's desirable to free up exec. resources for another thread
- Blocking synchronization:
  - Preempt the running thread (suspend)
  - OS schedules another thread to run
  - The other thread can run and make progress

```
if(test is not successful) {
  block_until_true();  // OS de-schedules thread
}

compute_assuming_test_succeeded();
```

Busy Waiting vs. Blocking Synchronization

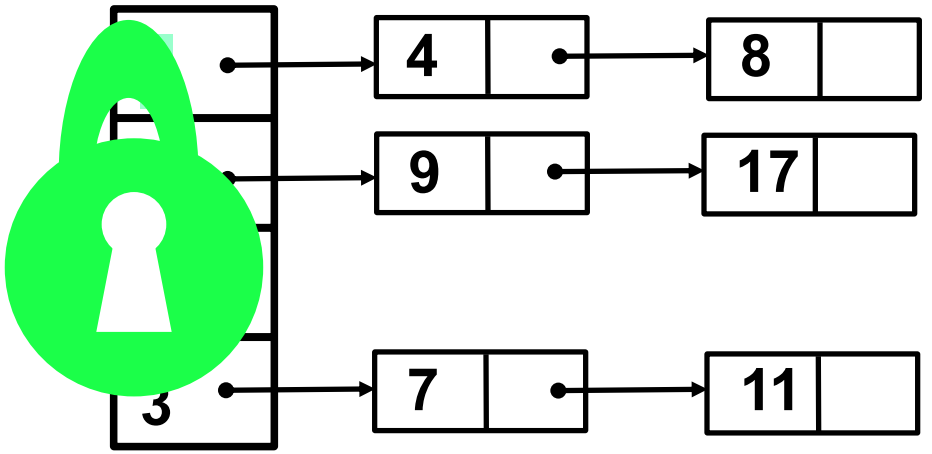
- Busy-waiting is preferable if:
  - Scheduling overhead is larger than expected wait time
    - Scheduling overhead is in the order of microseconds
    - While a few spinning iterations is in the order of nanoseconds
  - Processor's resources not needed for other tasks

- Blocking is preferred if waiting time is significant
  - And there are other tasks that need processor's resources

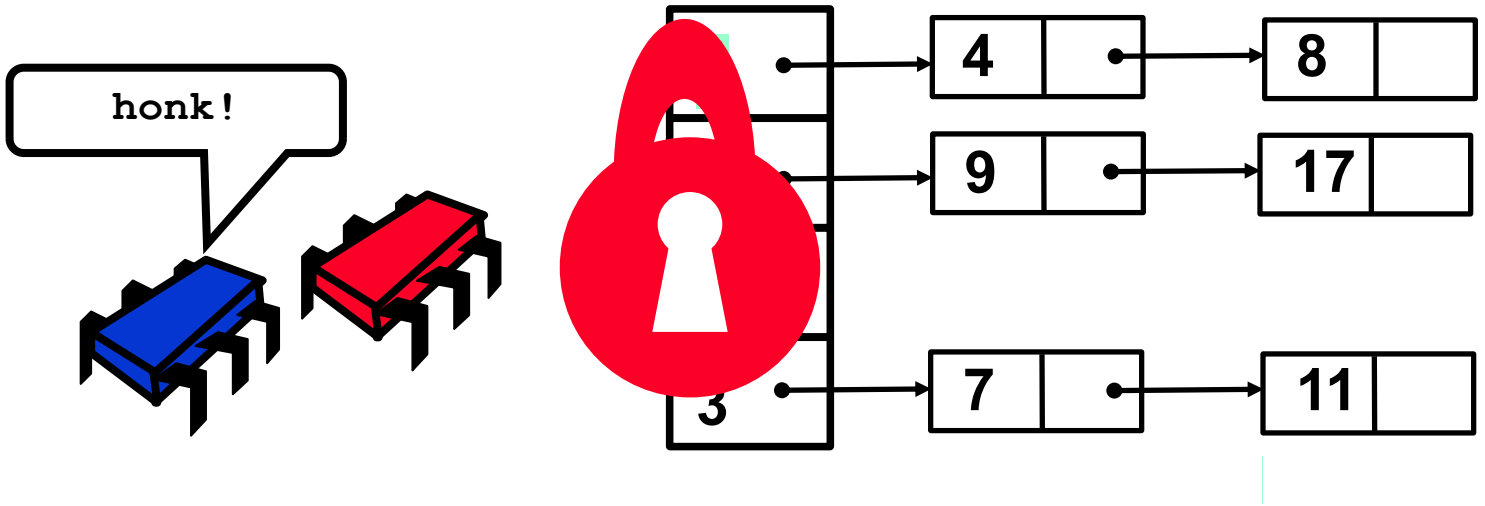
Recall: Concurrent Data Structures

- Various flavors
  - Coarse grained locks
    - easy to code, but bad performance
  - Fine grained locks
    - hard to code and reasonable performance
  - Lock-free data structures
    - very hard to code and best performance

Coarse-grained Locking



Coarse-grained Locking

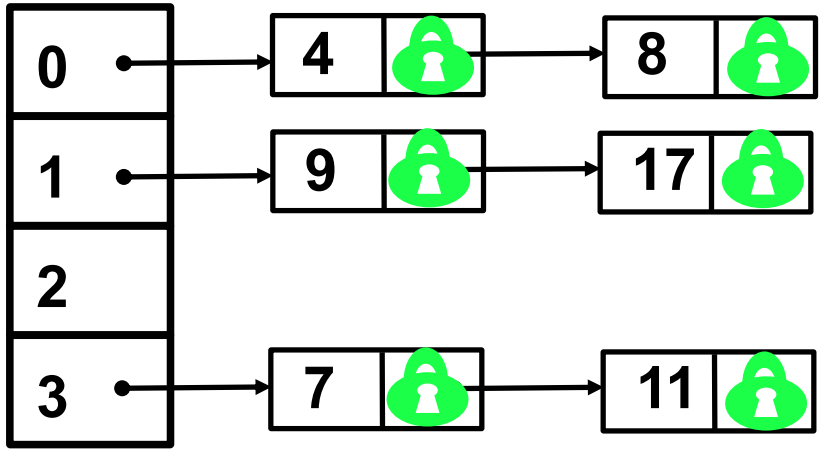


Coarse-grained Locking

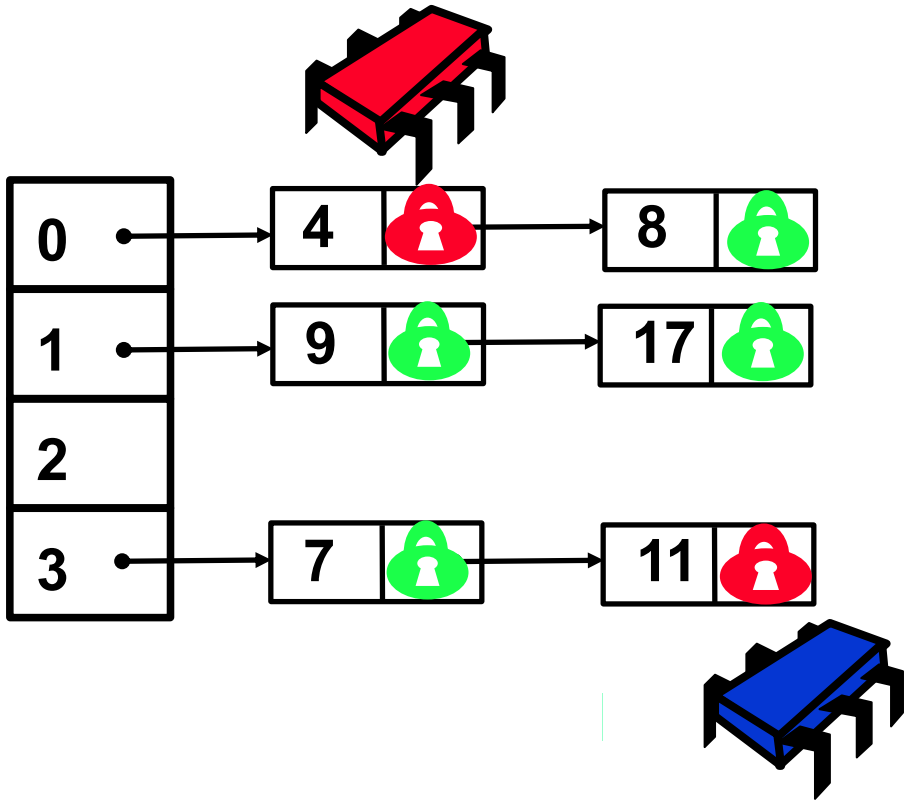
```
void editHash(HashTbl tbl, int key) {
    synchronized(tbl) {
        // read objects
        HashObj obj = tbl.get(key);
        // update
        obj.update();
    }
}
```

- Simple implementation
- Lots of contention
- No concurrency inside the data structure

Fine-grained Locking



Fine-grained Locking

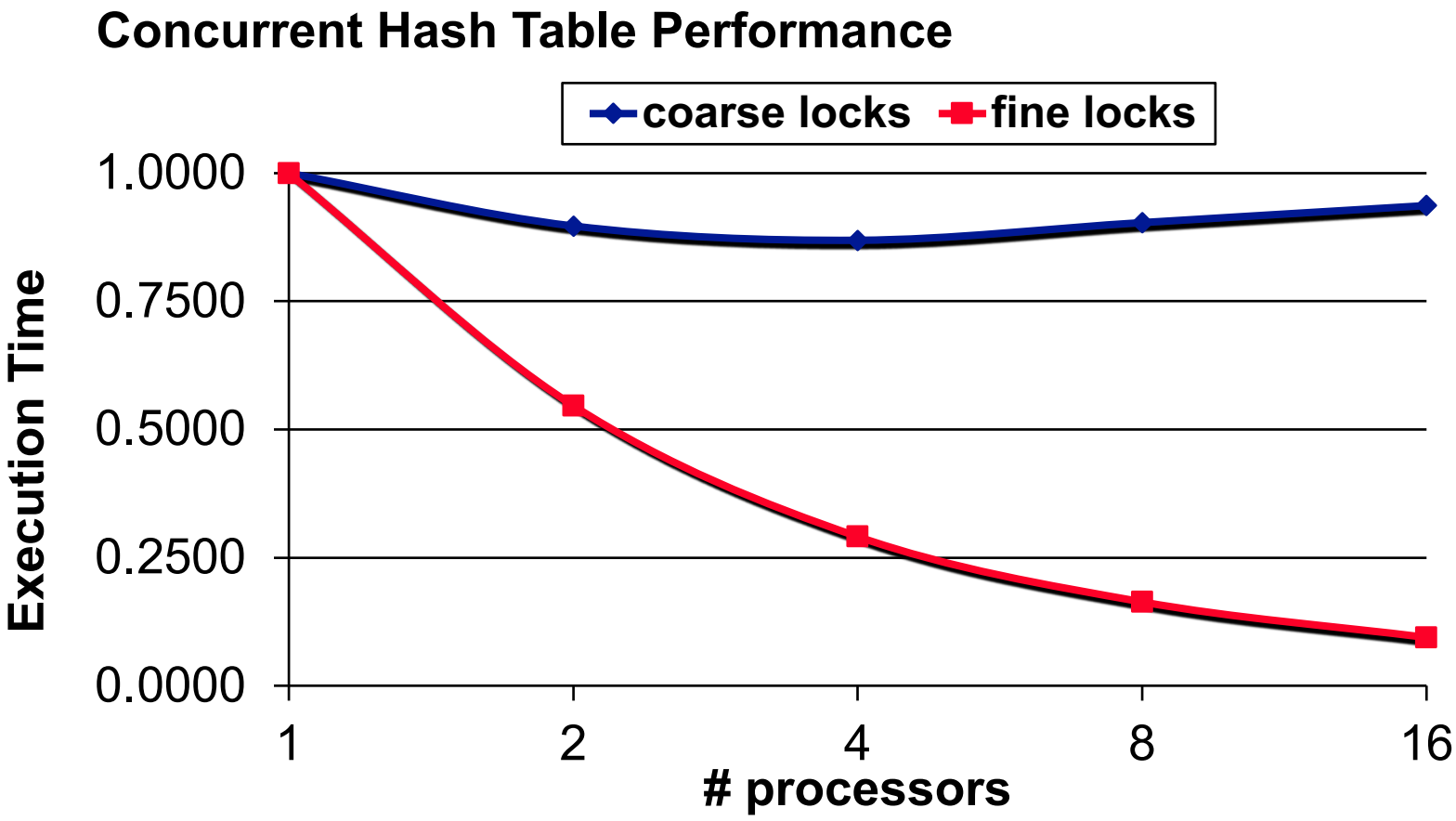


Fine-grained Locking

```
void editHash(HashTbl tbl, int key) {
  // read objects
  HashObj obj = tbl.get(key);
  synchronized(obj) {
      // update
      obj.update();
  }
}
```

- Contention for objects only
- Lots of concurrency exposed
  - Limited only to updating and reading each object
  - Cannot actually modify the hash table structure

Performance of Fine-grained locking



Reduced contention leads to better performance

Modifying the Hash Table

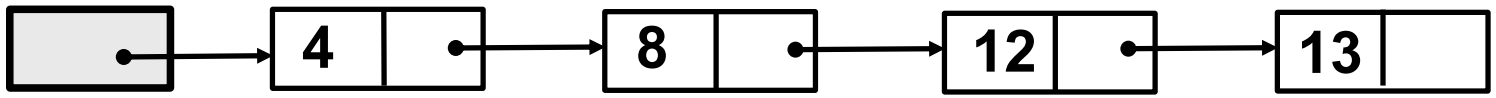
- Just saw, fine grained works well when the locks are very small, and held for a short time
  - e.g., only protecting 1 line of code

```
{obj.update();}
```

- Where do we lock if we are removing?
  - Choice 1: Just the node we are removing
    - Incorrect (Ask yourself why?)
  - Choice 2: The whole hash bucket
    - Correct, but not a lot of concurrency
  - Choice 3: All of the nodes that we modify
    - Correct, allows more concurrency in case of large lists

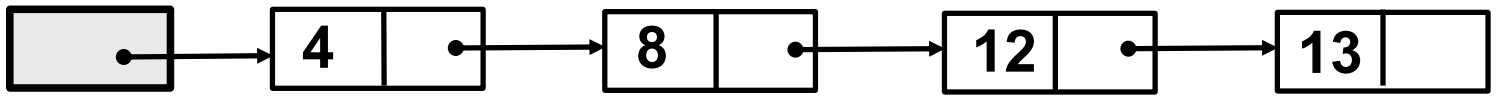
Localized Locking

- Which pointers do we edit on `delete(12)`?

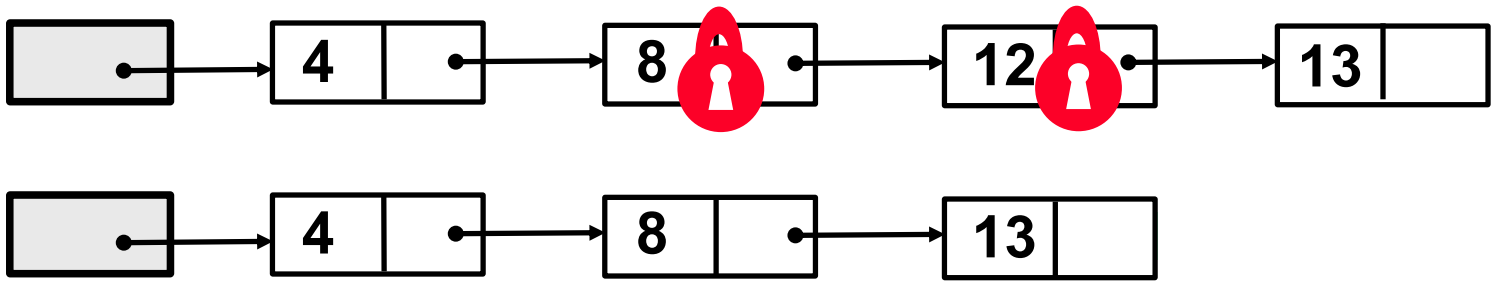


Localized Locking

- Which pointers do we edit on `delete(12)`?

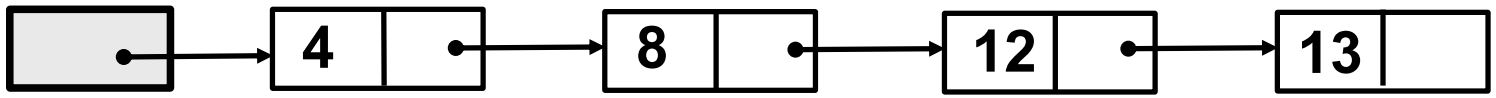


  - Answer: Deleted node, and previous node.
  - **8→next** now needs to point to **13, 12** is removed



Localized Locking

- How to ensure we get both those locks?

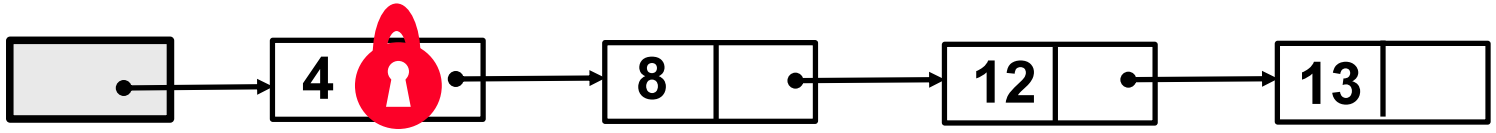


1. Grab locks as we traverse the list
2. Do not release `cur` until we have `cur→next`
   - Otherwise, another thread could delete it, making our behavior undefined

- This is called “hand over hand” locking

Localized Locking

- How to ensure we get both those locks?

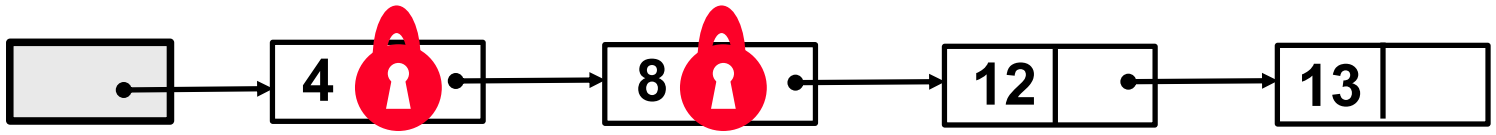


1. Grab locks as we traverse the list
2. Do not release `cur` until we have `cur→next`
   - Otherwise, another thread could delete it, making our behavior undefined

- This is called “hand over hand” locking

Localized Locking

- How to ensure we get both those locks?

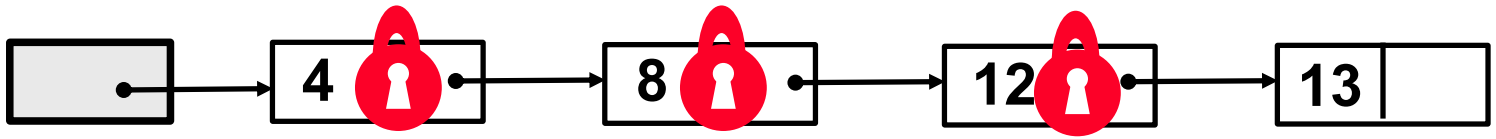


1. Grab locks as we traverse the list
2. Do not release `cur` until we have `cur→next`
   - Otherwise, another thread could delete it, making our behavior undefined

- This is called “hand over hand” locking

Localized Locking

- How to ensure we get both those locks?

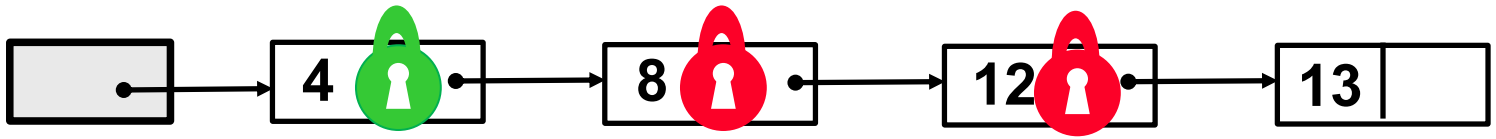


1. Grab locks as we traverse the list
2. Do not release `cur` until we have `cur→next`
   - Otherwise, another thread could delete it, making our behavior undefined

- This is called “hand over hand” locking

Localized Locking

- How to ensure we get both those locks?

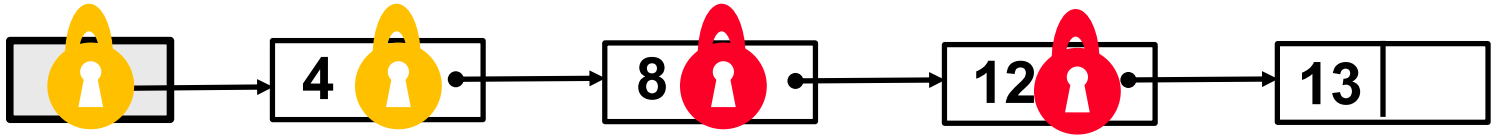


1. Grab locks as we traverse the list
2. Do not release `cur` until we have `cur→next`
   - Otherwise, another thread could delete it, making our behavior undefined

- This is called “hand over hand” locking

Performance Trade-offs

- Which specific scenarios does this allow?
  - Threads concurrently deleting elements in non-decreasing numerical order
    - e.g., `delete(12); delete(4);`

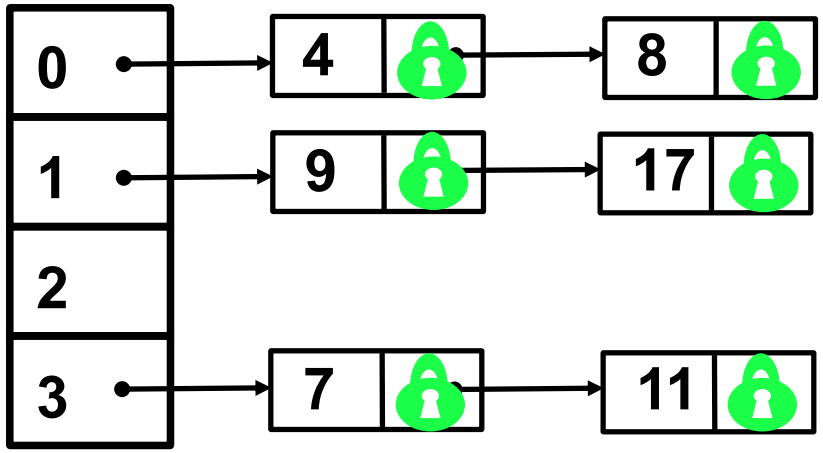


- Hand over hand forbids threads from passing
  - Probably a fairly rare scenario

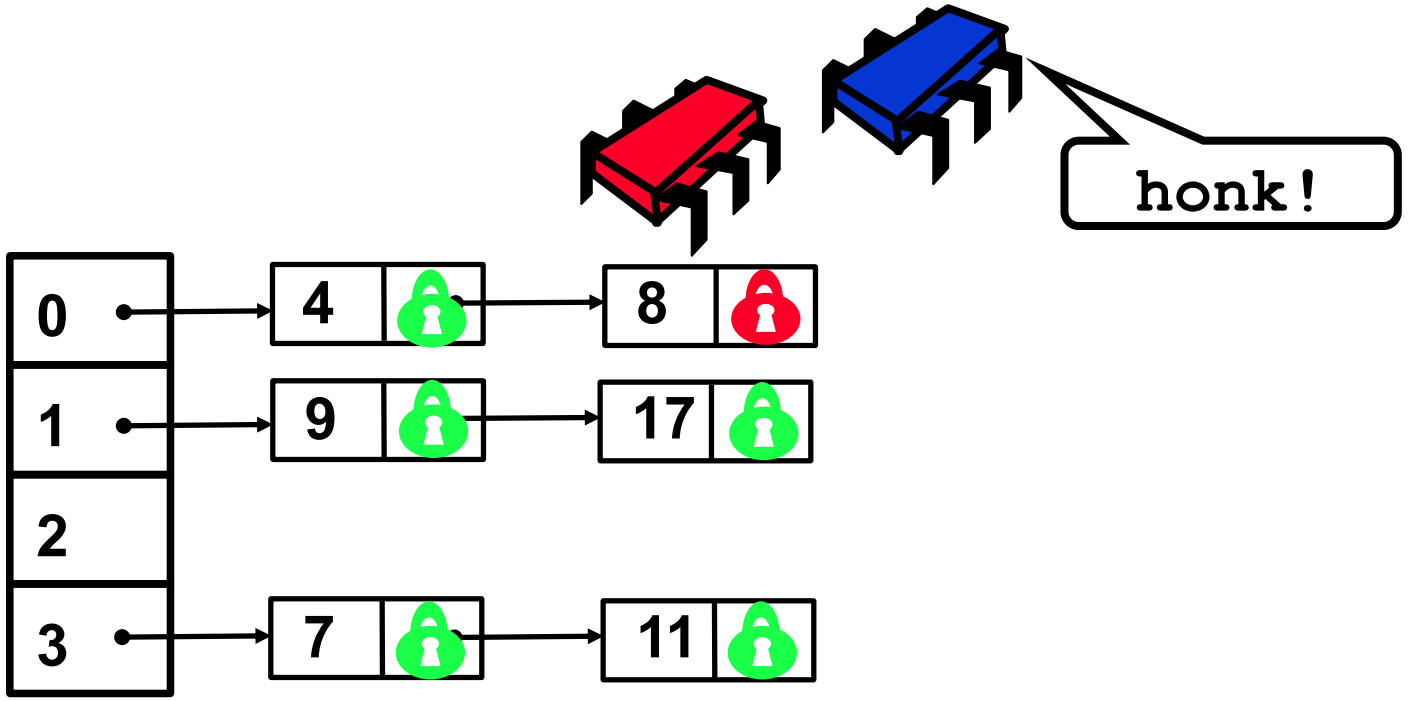
But, locks have other problems (alas 😔)

- Priority inversion
  - Higher priority threads waiting for a lock held by a lower-priority thread
- Convoying
  - Threads holding locks and are de-scheduled
- Lock composability
  - Threads holding multiple locks can lead to deadlocks or livelocks if implemented incorrectly

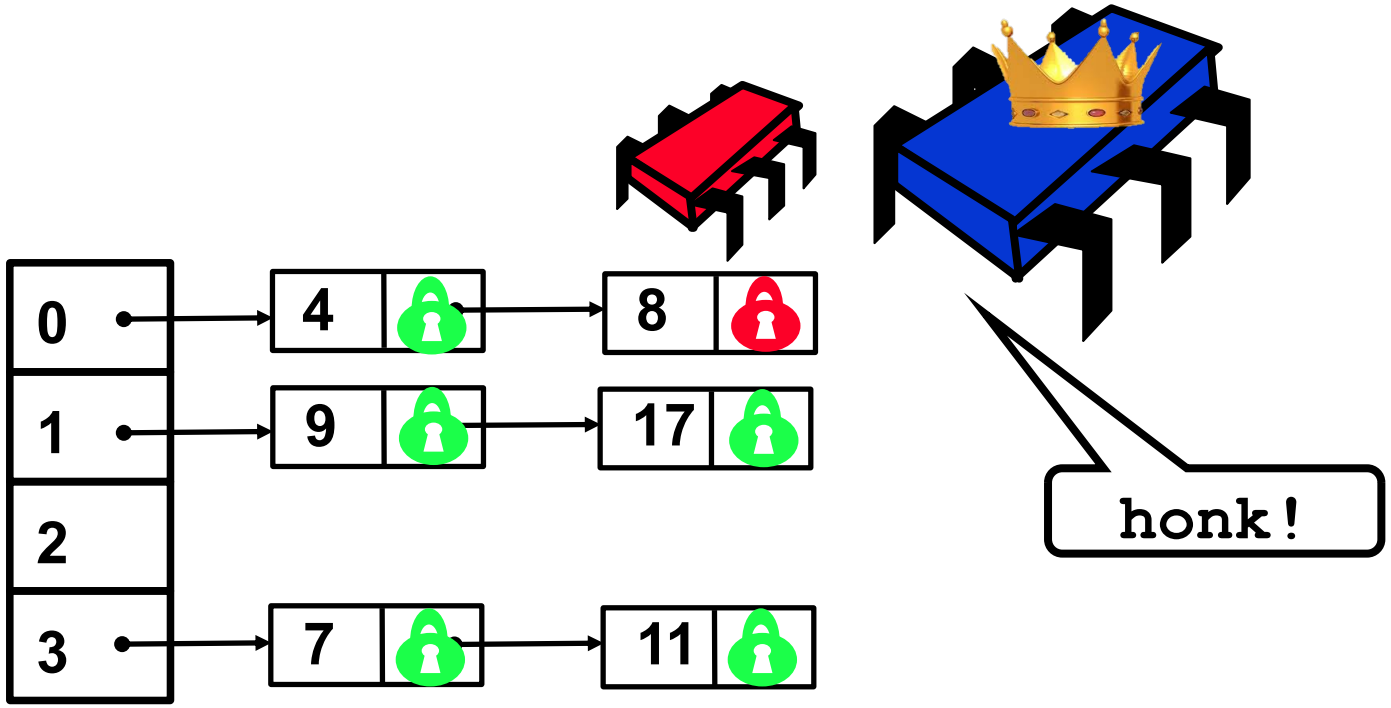
Fine-grained Locking: Good case



Fine-grained Locking: Contention



Fine-grained Locking: Priority Inversion

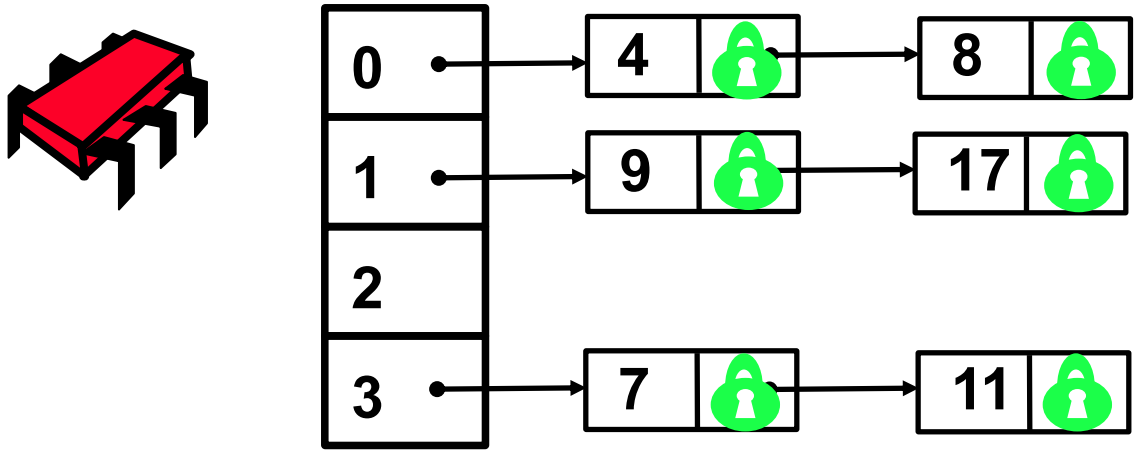


- Higher priority thread waiting behind lower priority thread
  - E.g., OS thread, or app thread in real-time systems

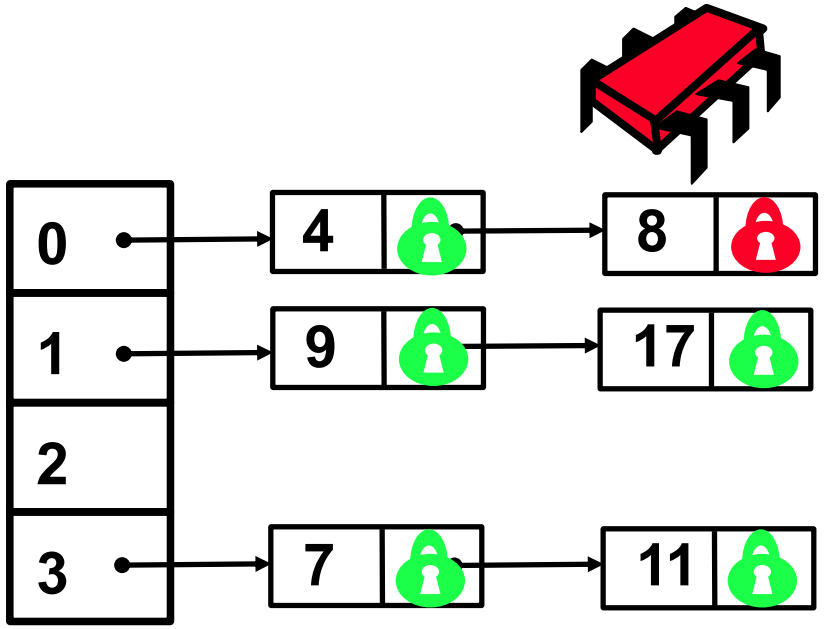
Fine-grained Locking: Priority Inversion

- Priority inversion:
  - When low-priority process is preempted while holding a lock needed by a high-priority process

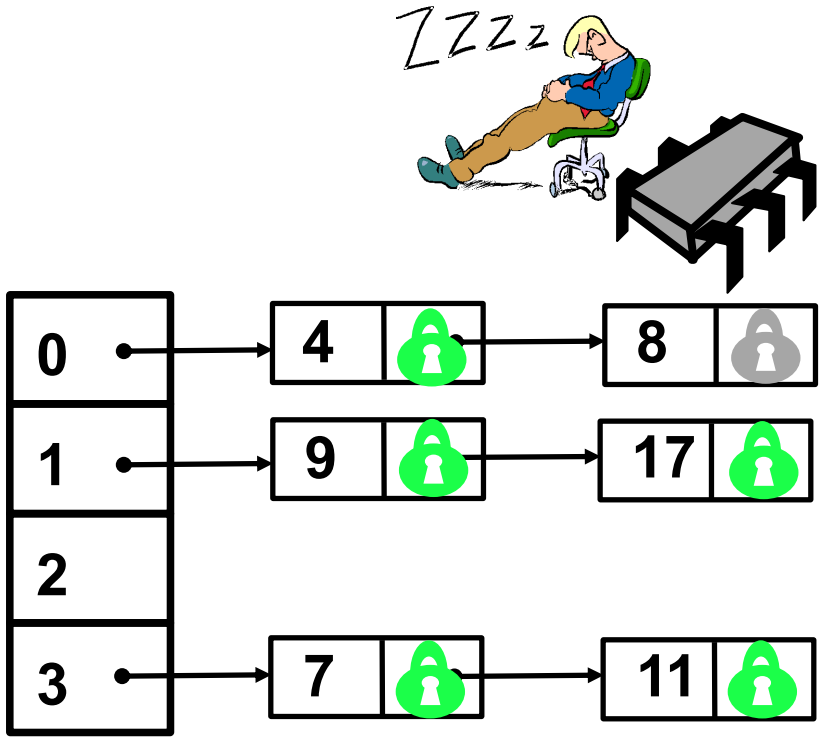
Fine-grained Locking: Convoying



Fine-grained Locking: Convoying

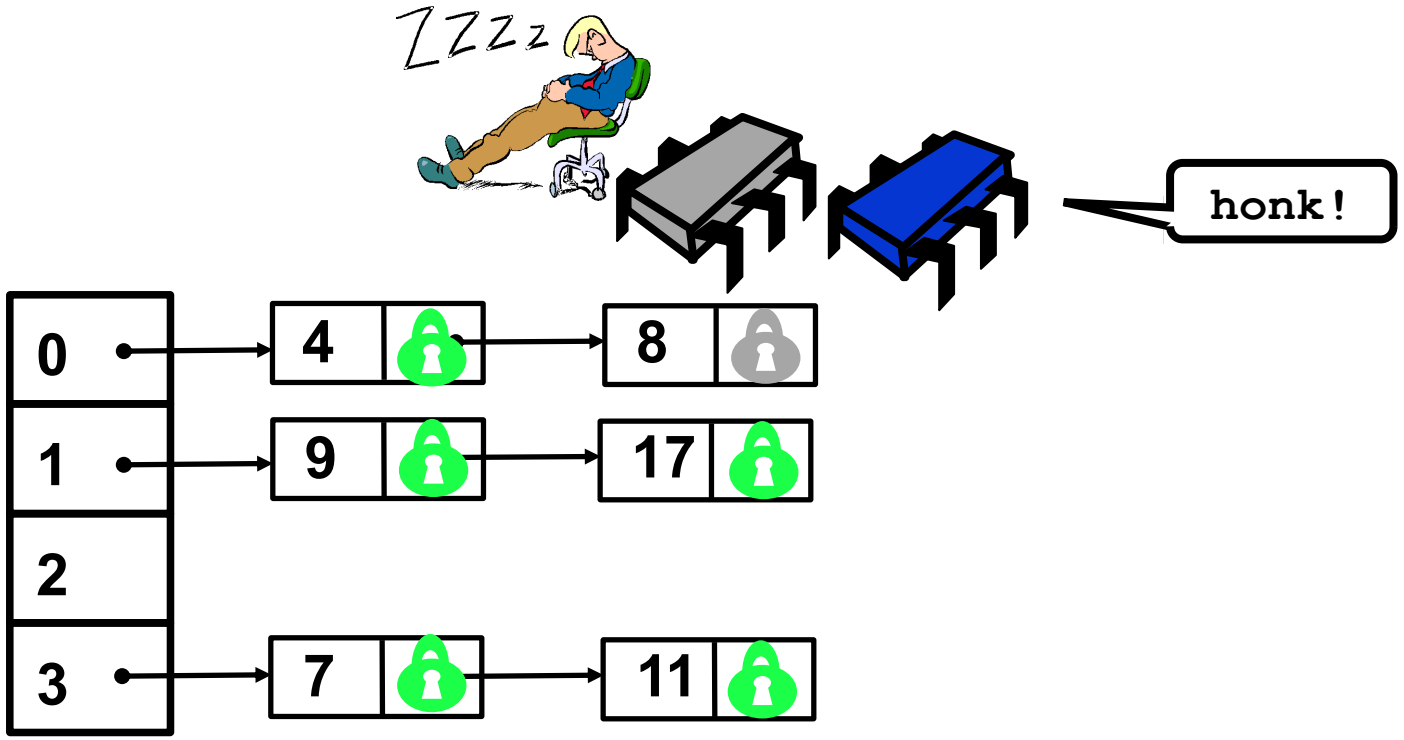


Fine-grained Locking: Convoying



- Thread holding the lock gets swapped out by OS
  - E.g., thread waiting for I/O sleeps, other threads run

Fine-grained Locking: Convoying



- Thread holding the lock gets swapped out by OS
  - E.g., thread waiting for I/O sleeps, other threads run

Fine-grained Locking: Convoying

- Convoying
  - When a process holding a lock is de-scheduled (e.g. page fault, no more quantum), no forward progress for other processes capable of running

Take a Break!

Fine-grained Locking: Lock Composability

```
void editHash2(HashTbl tbl, int keyA, int keyB)
{
    // read objects
    HashObj objA = tbl.get(keyA);
    HashObj objB = tbl.get(keyB);
    synchronized(objA) {
      synchronized(objB) {
         // update
         objA.update(objB);
         objB.update(objA);
      }
    }
}
```

P0

```
editHash2(tbl,1,2);
```

P1

```
editHash2(tbl,2,1);
```

This code deadlocks! Why?

Fine-grained Locking: Lock Composability

P0

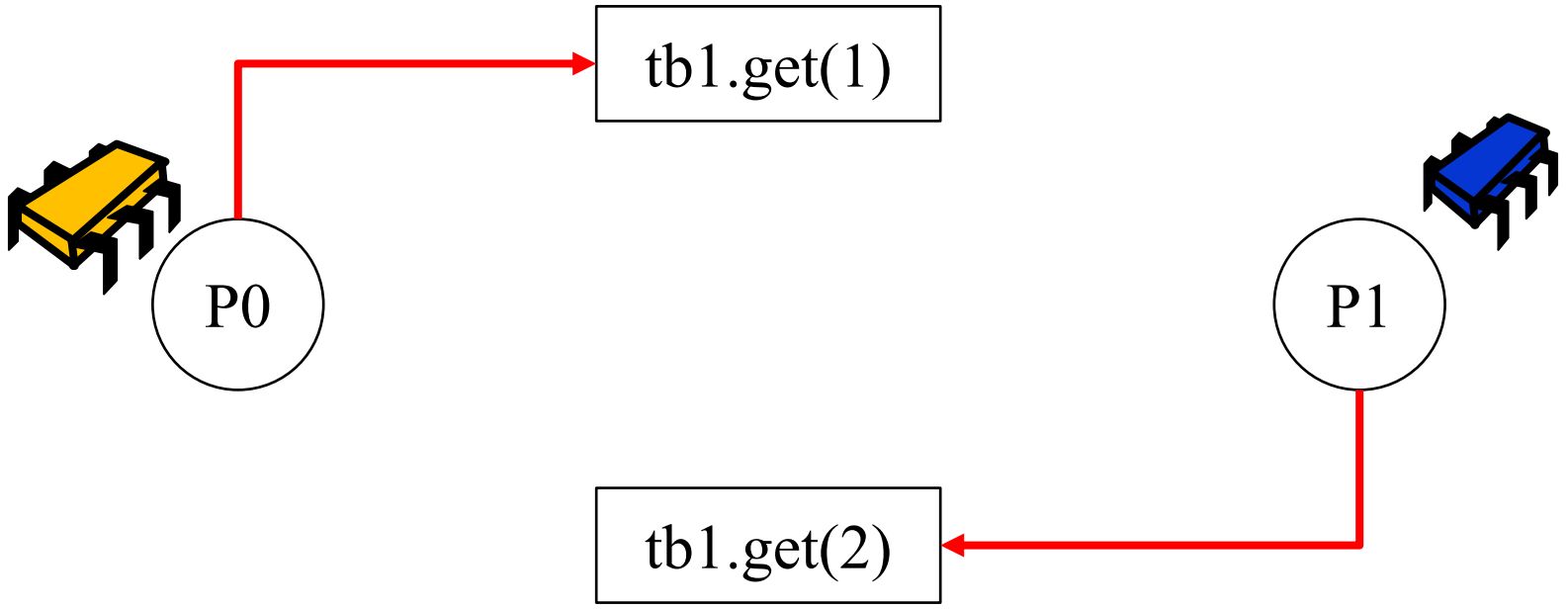
```
editHash2(tbl,1,2);
```

P1

```
editHash2(tbl,2,1);
```

```
objA = tbl.get(1);
synchronized(objA){
```

```
objA = tbl.get(2);
synchronized(objA){
```



Fine-grained Locking: Lock Composability

P0

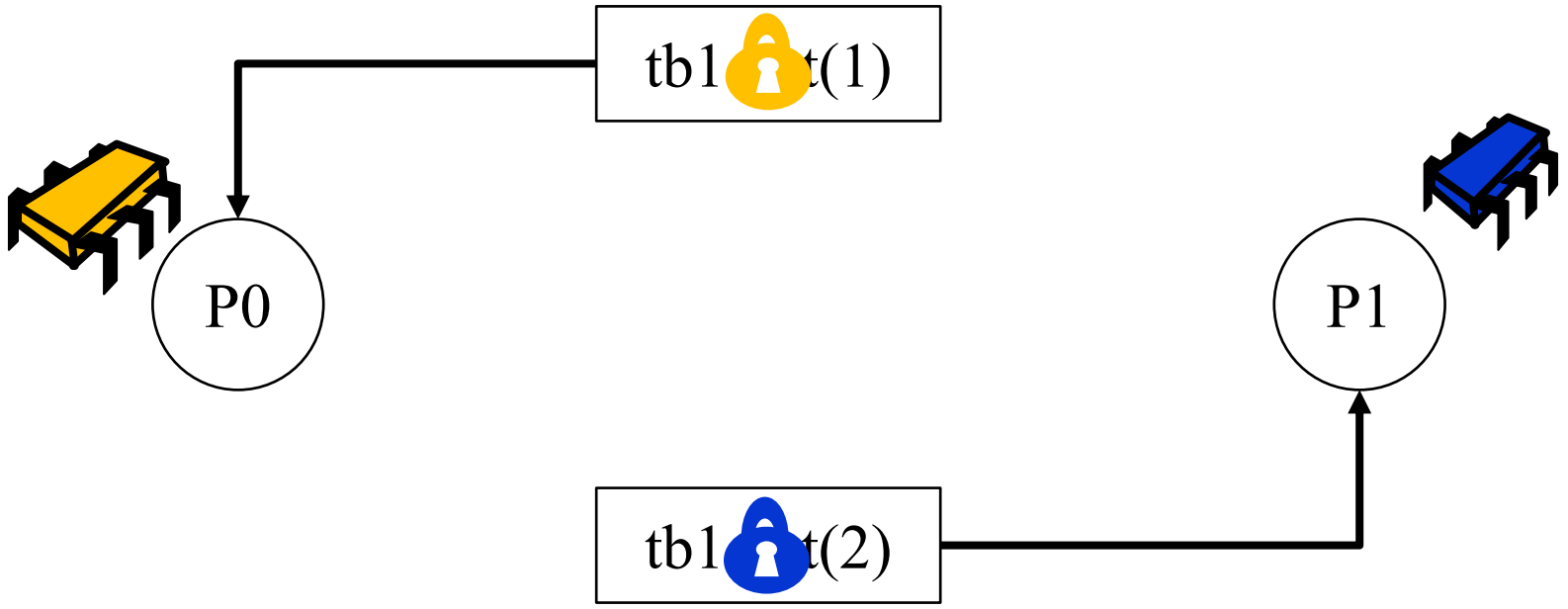
```
editHash2(tbl,1,2);
```

P1

```
editHash2(tbl,2,1);
```

```
objA = tbl.get(1);
synchronized(objA){
```

```
objA = tbl.get(2);
synchronized(objA){
```



Fine-grained Locking: Lock Composability

P0

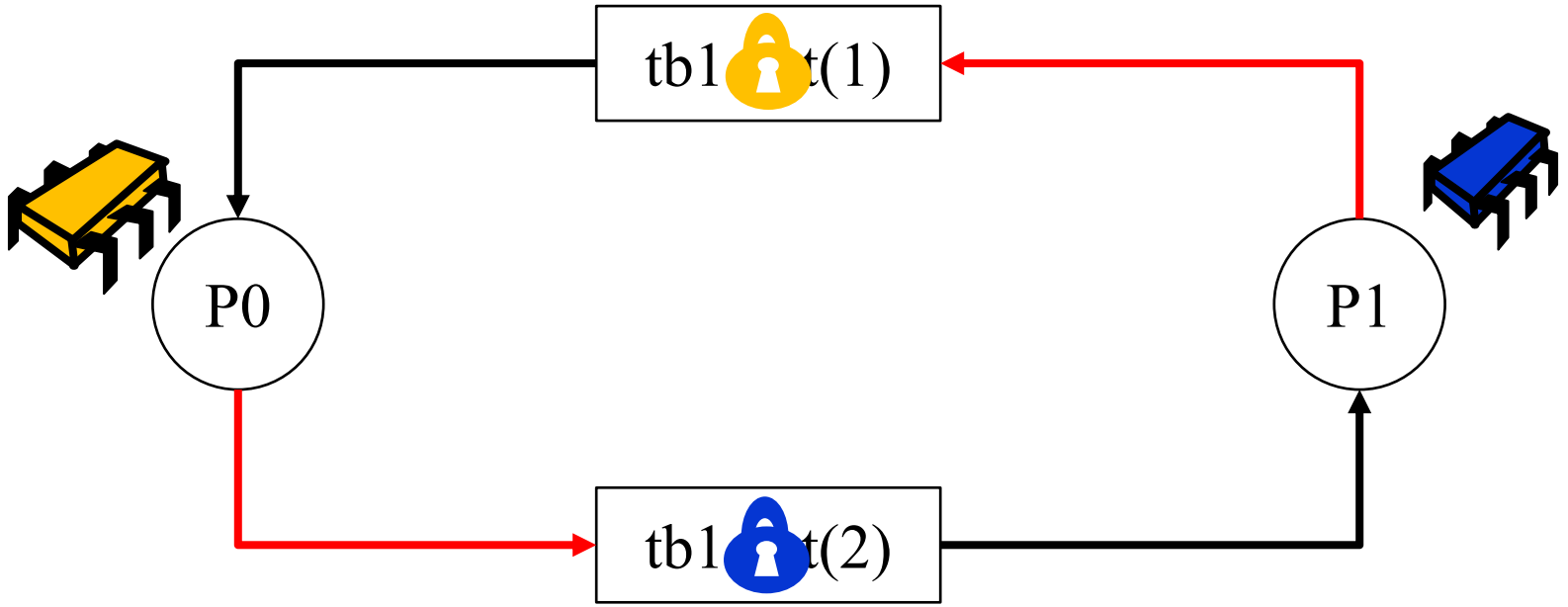
```
editHash2(tbl,1,2);
```

P1

```
editHash2(tbl,2,1);
```

```
objB = tbl.get(2);
synchronized(objA){
  synchronized(objB){
```

```
objB = tbl.get(1);
synchronized(objA){
  synchronized(objB){
```



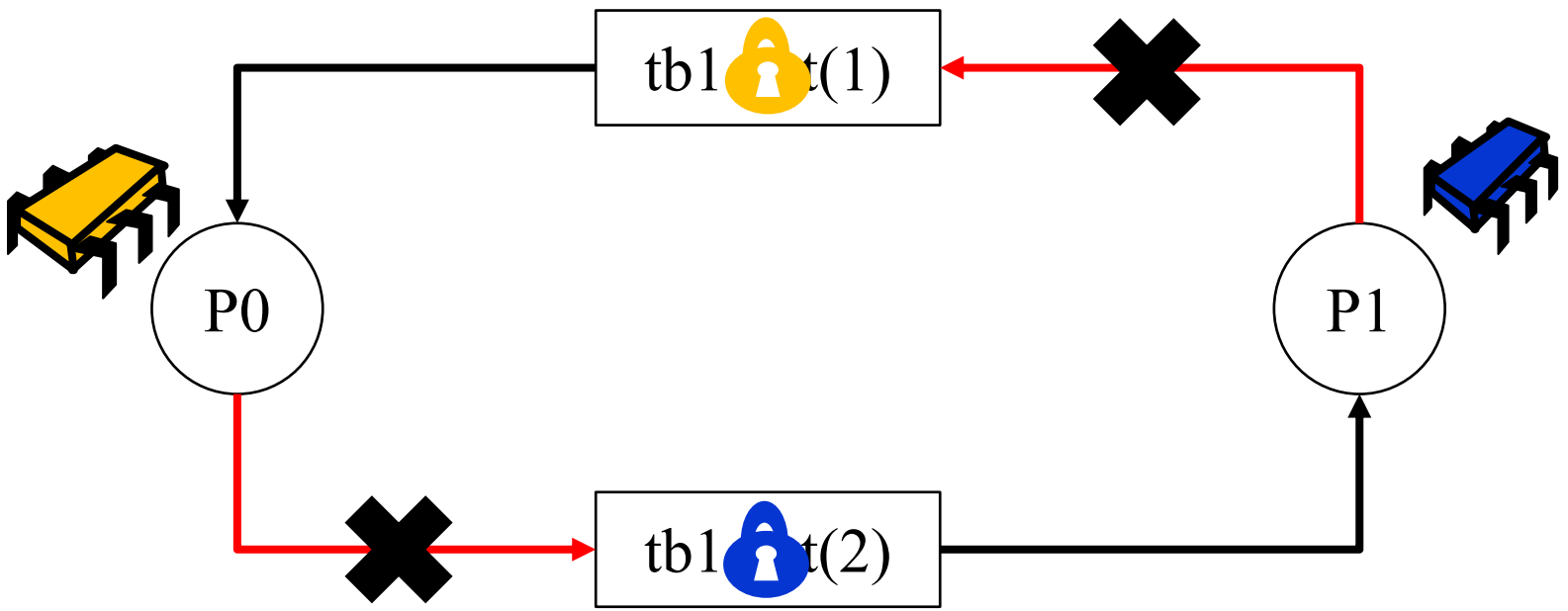
Fine-grained Locking: Lock Composability

P0

```
editHash2(tbl,1,2);
```

P1

```
editHash2(tbl,2,1);
```



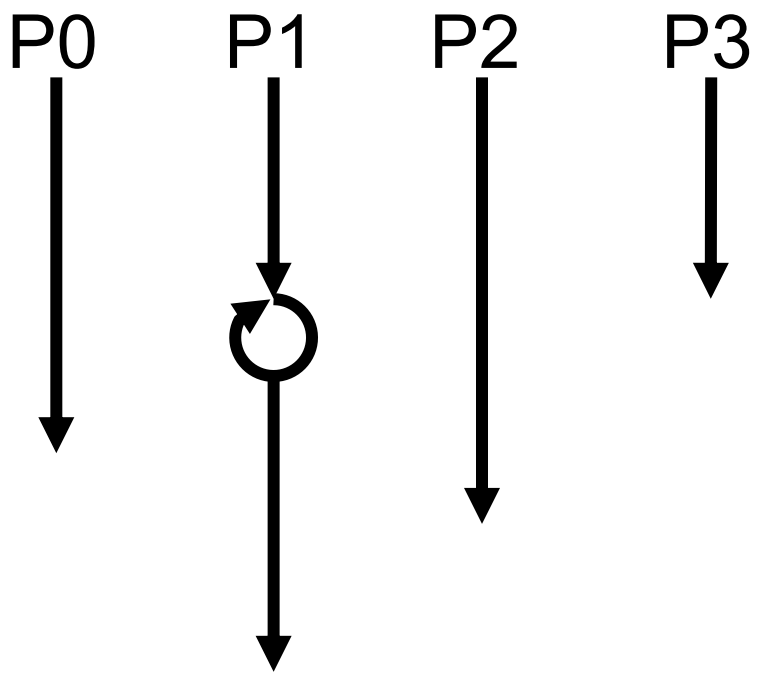
- Neither process can proceed!
- Each process waiting for the other one to release lock
- Deadlock!

Fine-grained Locking: Lock Composability

- Composing lock-based code is tricky
  - Can lead to deadlock or livelock if implemented incorrectly
  - Major problem in most modern systems

Expectations from Software

- All processes should make forward progress
  - Processes make progress toward completing their tasks
  - Processes will eventually finish what they started
  - Processes are not stuck, waiting forever, or being blocked indefinitely



Recap: What can block the progress of processes?

- Long-latency operations such as memory or I/O
  - Will eventually complete if hardware is designed correctly
- Synchronization
  - Waiting for other threads/processes to reach a barrier
  - Atomic operations
  - Waiting for locks
- A process can get stuck indefinitely because of incorrect placement of synchronization in software

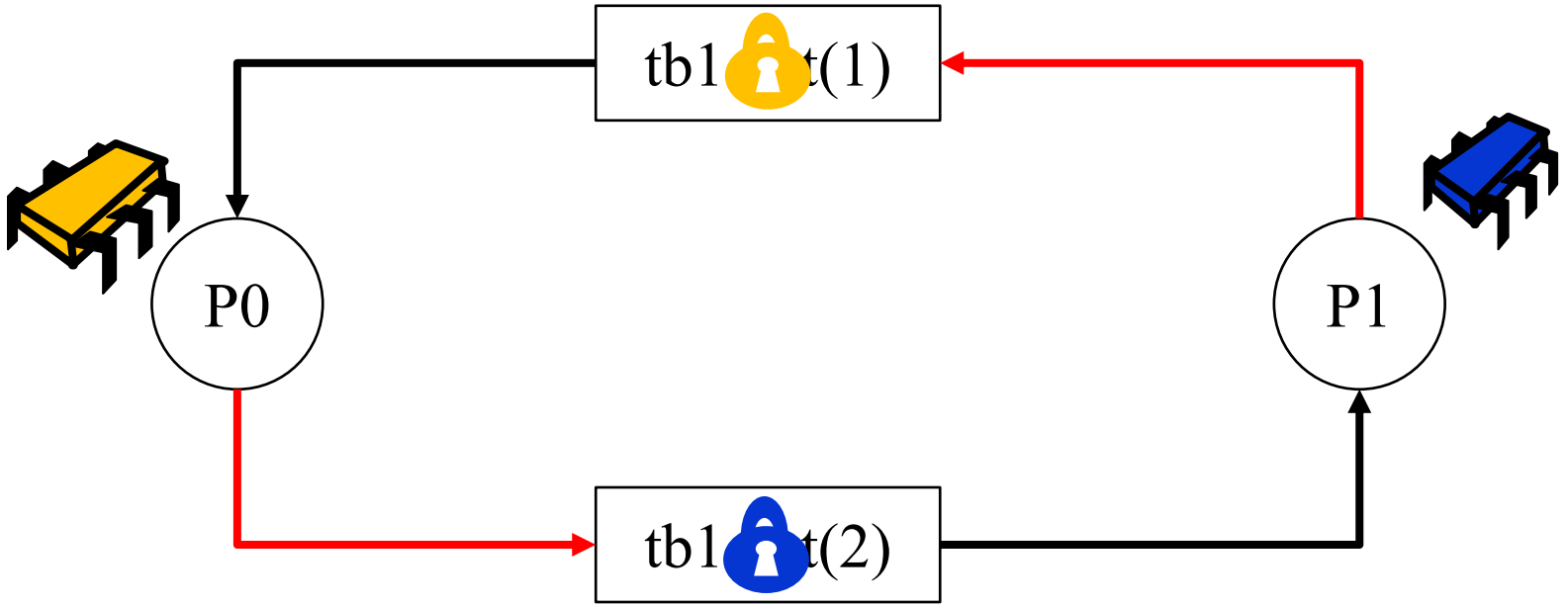
Liveness Properties

- What are deadlocks/livelocks?
  - Deadlock (processes sleep waiting for one another)
  - Livelock (processes execute but make no progress)

- We want execution to be deadlock-/livelock-free AND
  - No starvation of shared resources
  - Aim for fairness
  - Minimality (no unnecessary waiting or signaling)

Formally Defining Deadlocks

- Every process in a set is waiting for an event that can only be caused by another process in the same set
- Consider the previous example with two processes:



- Process 0 waiting for process 1
- Process 1 waiting for process 0

Necessary and Sufficient Conditions for Deadlock

1. Mutual exclusion: resources protected by locks
2. Hold and wait: hold a resource and wait for another
3. No pre-emption: no way for B to “seize the lock” from A
4. Cyclic waiting: A waits for B, B waits for A (a “circular” pattern)

If any of these four properties are missing, deadlock will not happen

Our goal: Try to prevent one of the conditions from being true

Deadlock Prevention (1)

- Mutual exclusion: resources protected by locks
  - A resource can be a variable, pointer, shared object, etc.
  - For example,

```
#pragma omp parallel {
   int result = heavy_computation_part1();

   #pragma omp atomic
   sum += result;
}
```

- Essential property to maintain data integrity
  - Cannot be prevented from being true in modern systems

Deadlock Prevention (2)

- Hold and wait: hold a resource and wait for another
- Consider the previous example with P0:

```
objA = tbl.get(1);
objB = tbl.get(2);
synchronized(objA){
  synchronized(objB){
    ...
```

- P0 holds the lock for **objA** while waiting for **objB**
- Especially prevalent when using fine-grained locks

Deadlock Prevention (2)

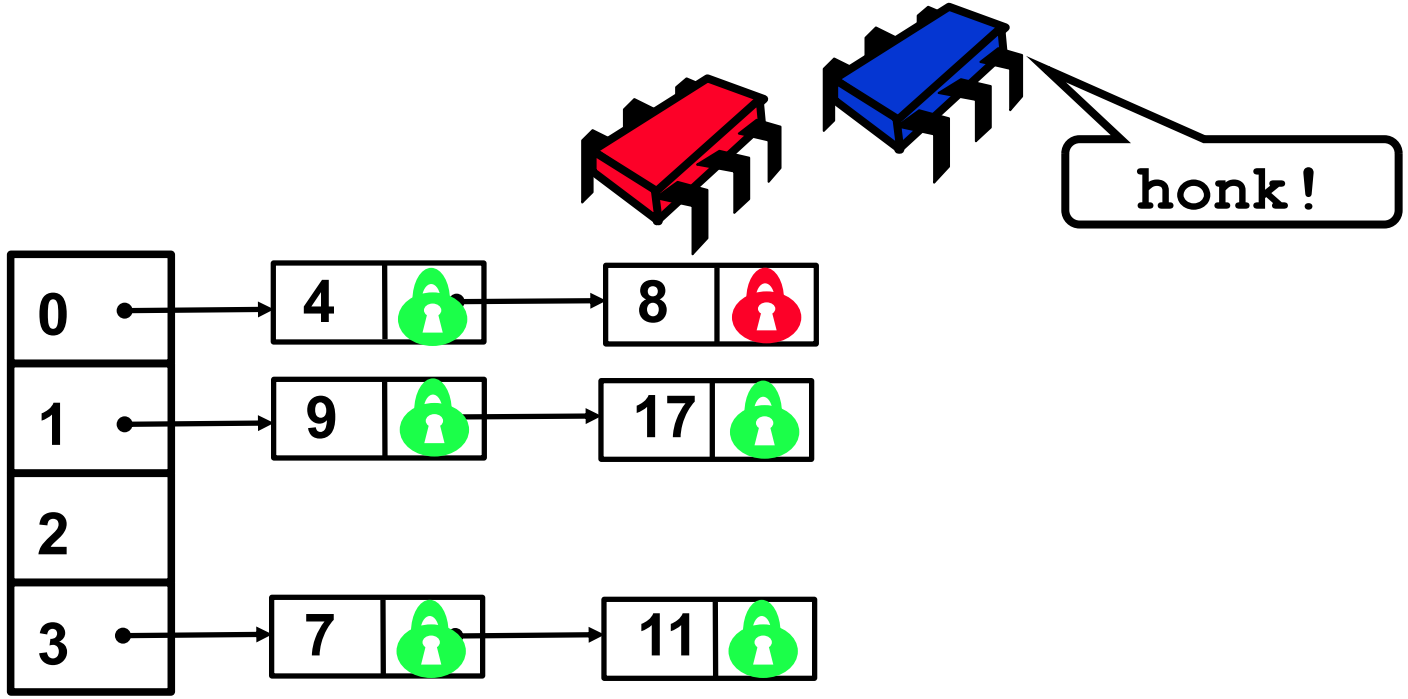
- Possible solution: acquire all resources before running
  - In the prior example, it would be equivalent to coarse-grained locking

```
synchronized(tbl) {  // Lock the entire hash table
      objA = tbl.get(1);
      objB = tbl.get(2);
      ...
}
```

- Process 0 locks the entire hash table
- Comes with all the drawbacks of coarse-grained locking as discussed

Deadlock Prevention (3)

- No pre-emption: no way for B to “seize the lock” from A



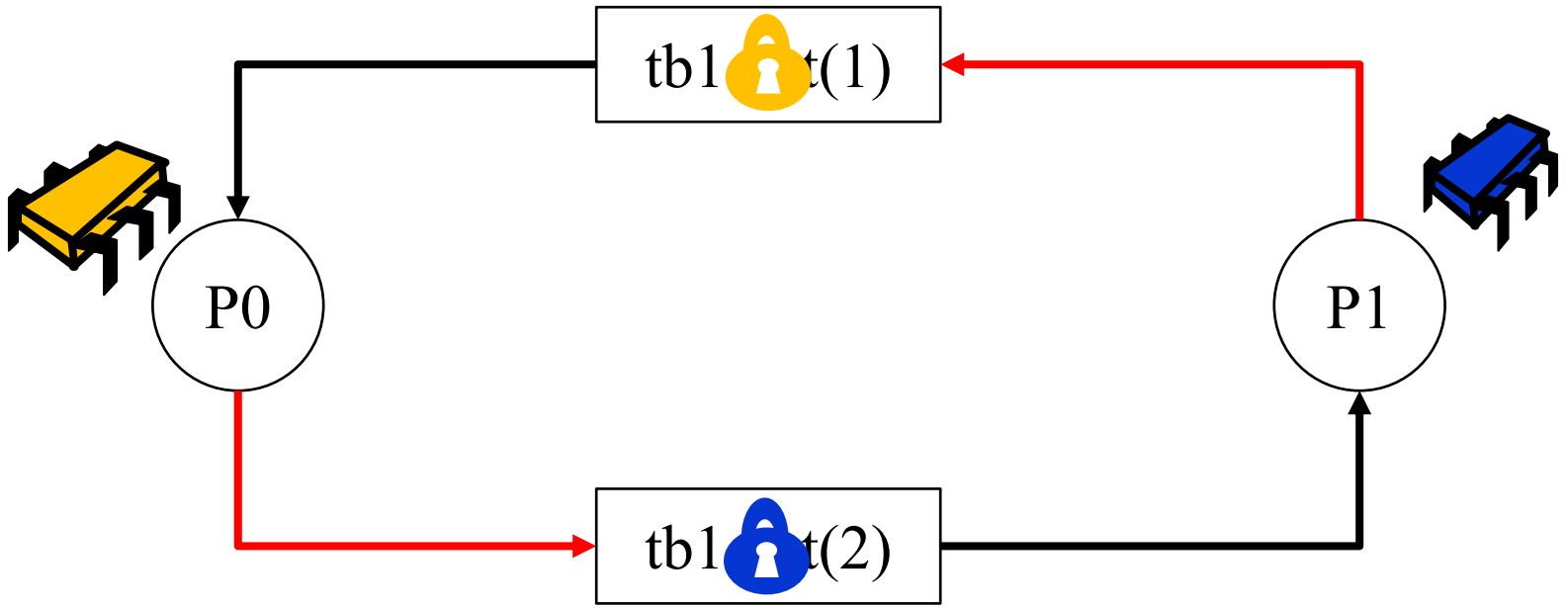
- Leads to problems such as priority inversion as discussed earlier

Deadlock Prevention (3)

- Possible solution:
  - Allocate priority to threads
  - Higher priority thread can pre-empt lower priority thread and steal all resources
- But this approach is generally unsafe
- Several problems can arise
  - Data corruption possible if a thread is pre-empted during execution
  - Performance overheads with pre-emption
  - Some resources cannot be safely pre-empted (e.g., syscalls)

Deadlock Prevention (4)

- Cyclic waiting: A waits for B, B waits for A (a “circular” pattern)



- Easiest and most practical to ensure this property never holds

Deadlock Prevention (4)

- Possible solution: only allow acquiring locks in a fixed order
- For example, allow only increasing order of locking keys

```
int small_key = (keyA < keyB) ? keyA : keyB;
int big_key = (keyA > keyB) ? keyA : keyB;

HashObj objA = tbl.get(small_key);
HashObj objB = tbl.get(big_key);
synchronized(objA) {
      synchronized(objB) {
            ...
      }
}
```

Deadlock Prevention (4)

- Solves the problem for P0 and P1
- Both P0 and P1 will try to lock the first element at the beginning
  - Regardless of the order in which their arguments are passed
  - Whichever process wins, proceeds with the execution first

P0

```
editHash3(tbl,1,2);
```

```
small_key = 1;
big_key = 2;

objA = tbl.get(1);
synchronized(objA){
```

P1

```
editHash3(tbl,2,1);
```

```
small_key = 1;
big_key = 2;

objA = tbl.get(1);
synchronized(objA){
```

Practical Challenges

- Processes often do not know ahead of time which resources to lock
  - For example, consider the following code snippet:

```
void editHash4(HashTbl tbl, int keyA) {
  HashObj objA = tbl.get(keyA);
  synchronized(objA) {
    int keyB = objA.val {
    HashObj objB = tbl.get(keyB);
    synchronized(objB) {
      ...
```

  - No way to know the value of **`objB`** prior to locking **`objA`**

- Self-imposed rule: Programmer must ensure code follows ordering

Practical Applications

- Ordering locks often applied to list-like data structures
- For example, consider:
  - Multiple threads accessing a single array
  - Sort the elements of the array in some order
  - Allow threads to lock the elements one by one in the order
- Popular approach inside the Linux kernel
- Simple but practical approach to prevent deadlocks

Livelock

- Contrary to deadlock, processes are always active
  - Constantly changing states or retrying
  - But make no progress

- Real life example:
  - Two people meet face-to-face in a corridor
  - Each move aside to let the other pass
  - They end up swaying from side to side without making any progress
  - They always move the same way at the same time

- The same four necessary and sufficient conditions for deadlocks apply

Summary

- Building fast locks with hardware support
  - TS, TTS, LL/SC
  - All of them directly interact with coherence protocol (and consistency model too!)
  - how to build one type of locking primitive out of another

- Deadlocks prevention/detection an essential part of modern systems