# Locks: Implementation

ICS432
Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

# Implementing Lock?

- At this point we know how to use lock() and unlock() to create critical sections
- Question: how does one implement lock()?
  - Granted, you will probably never need to as languages/ systems provide them
  - But it's interesting to have some idea of how things work
  - **And it will be our first attempt at truly reasoning about concurrency**
- The first natural attempt is to try to implement lock() and unlock() in software, like any other method/function
  - Following the pseudo-code in the previous set of lecture notes

# Spinlocks

- We'll use the following basic idea
- A lock will be a boolean variable, initially set to 0
  - 0 means: nobody "has the lock", i.e., no thread is in the critical section defined by the lock
- lock():
  - While lock == 1, keep testing until the lock is == 0
  - When lock == 0, then set the lock to 1
    - So that other threads can't get in
- unlock()
  - set lock to 0
- This is called spinning because if a thread is already in the critical section, another will keep testing the lock over and over

# Assumptions

- To simplify we'll assume
  - A single core (false concurrency)
  - An OS with a scheduler that does some type of round-robin scheduling (time-slicing via context-switching)
- We're going to go through a series of implementations
  - Re-tracing the history of "software spinlocks"
- We'll analyze each implementation for correctness
- We assume that the OS scheduler is an adversary
  - It tries to place context-switches inconveniently so as to break correctness
- If there is one case, **no matter how unlikely,** in which the execution is incorrect, then we declare the code broken

# Software Spin Locks: v0

- The simplest (but wrong) possible implementation

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

```
void lock(int *lock) {
  while (*lock) {}  // spin
  *lock = 1;
}
```

- What’s wrong with this implementation?

# Software Spin Locks: v0

```
void lock(int *lock) {
  while (*lock) {} // spin
  *lock = 1;
}
```

- Assume the lock is unlocked, and we have two threads
- Thread A calls lock, and doesn't spin because *lock = 0
- Before thread A gets a chance to set *lock to 1, it is context-switched out
- Thread B is context-switched in, calls lock(), doesn't spin because *lock = 0, sets *lock to 1, enters the critical section protected by the lock, and get context-switched out
- Thread A is context-switched back in, sets *lock to 1 (which it already is!), and enters the critical section
- We have two threads in the critical section, therefore we don't have mutual execution, therefore our lock() implementation is broken

# Software Spin Lock: v0

- There is a race condition in the lock() function on the boolean lock variable itself!
  - Ironically, our lock() function is not thread-safe!
  - Adding another lock on the lock would only push the problem down one level, and so on...
- One possible solution could be to used a “turn-based” system
  - A variable alternates between 0 and 1
  - A value of 0 indicates that Thread #1 should get access to the critical section
  - A value of 1 indicates that Thread #2 should get access to the critical section
  - Initially the value is (arbitrarily) set to 0
- Let’s look at the code

# Software Spin Lock: v1

```
void unlock(int *lock, int id) {
  *lock = 1 - id;
}
```

```
void lock(int *lock, int id) {
  while (*lock != id) {}  // spin
  *lock = id;
}
```

- Thread #1 calls the functions passing 0 as an argument, and thread #2 calls the functions passing 1 as an argument
- The code above solves the problem of the previous implementation
  - The two threads cannot enter the critical section because only a single thread can have its id equal to the lock
- What is the problem?

# Software Spin Lock: v1

```
void unlock(int *lock, int id) {
  *lock = 1 - id;
}
```

```
void lock(int *lock, int id) {
  while (*lock != id) {}  // spin
  *lock = id;
}
```

- The problem is starvation
- Consider the following sequence of locks and unlocks:

Thread A: lock(0);

Thread A: unlock(0);

Thread A: lock(0);      // blocks!

- Thread A is blocked until Thread B goes into the critical section
  - Thread B may not even do anything for the next hour
- Threads **are forced to alternate** in the critical section
  - Because it's turn-based
- This goes against the principle of "no unnecessary delays"
- Let's look at another idea...

# Software Spin Lock: v2

```
void unlock(lock_t lock, int id) {
  lock->flag[id] = false;
}
```

```
void lock(lock_t lock, int id) {
  while (lock->flag[1-id] == true) {} // spin
  lock->flag[id] = true;
}
```

- Use two variables inside the lock:

```
typedef struct {
    boolean flag[2];   // initialized to {false, false}
} *lock_t;
```

- The idea: when a thread wants to acquire the lock, it looks at whether the other thread has it
- This avoids the “forced alternation” problem of the previous solution
- But is it correct?  Anybody?

# Software Spin Lock: v2

```
void unlock(lock_t lock, int id) {
  lock->flag[id] = false;
}
```

```
void lock(lock_t lock, int id) {
  while (lock->flag[1-id] == true) {} // spin
  lock->flag[id] = true;
}
```

- Incorrect, for the same reason as v0 was broken: race condition!
  - The two threads enter lock() “at the same time”
  - They both see the other’s flag set to false and proceed
  - We now have two threads in the critical section!
- This is a very typical problem
  - You cannot test for a condition and then take action based on the test in a way that is atomic
    - We saw this a few times already
  - More plainly: if (cond) { do_something; } is not atomic
- Let’s look at yet another idea....

# Software Spin Lock: v3

```
void unlock(lock_t lock, int id) {
  lock->flag[id] = false;
}
```

```
void lock(lock_t lock, int id) {
  lock->flag[id] = true;
  while (lock->flag[1-id] == true) {} // spin
}
```

- To fix the problem we swap the two statements in function lock()
  - The idea is to right away (atomically) say “I want to enter the critical section” by setting lock->flag[id]
  - There is no interleaving of the executions that can lead to both threads entering the critical section simultaneously

```
lock->flag[0] = true;                      lock->flag[1] = true;
while(lock->flag[1] == true) yield();      while(lock->flag[0] == true) yield();
. . .                                      . . .
lock->flag[0] = false;                     lock->flag[1] = false;
```

- But now we have a new problem...

# Software Spin Lock: v3

```
void unlock(lock_t lock, int id) {
  lock->flag[id] = false;
}
```

```
void lock(lock_t lock, int id) {
  lock->flag[id] = true;
  while (lock->flag[1-id] == true) {} // spin
}
```

- Deadlock!
  - Both threads set their variables to true “at the same time”
    - Thread #1 sets his to true
    - Context-switch
    - Thread #2 sets his to true
    - And at this point both threads spin forever
- Again, unlikely but possible
  - Remember that we consider the OS scheduler as an adversary
- Let’s look at yet another idea…

# Software Spin Lock: v4

```
void unlock(lock_t lock, int id) {
  lock->flag[id] = false;
}
```

```
void lock(lock_t lock, int id) {
  lock->flag[id] = true;
  while (lock->flag[1-id] == true) { // spin
      lock->flag[id] = false;
      lock->flag[id] = true;
  }
}
```

- The idea here is to fix the problem from v3 by having threads back off when they realize they're both entering the function at the same time
  - If the other's flag is set to true, I set mine to false, let the other run for a while (which should happen due to OS scheduling), and set mine to true again and check on the other's flag
- There is STILL a problem here! (**really** unlikely)

# Software Spin Lock: v4

```
void unlock(lock_t lock, int id) {
  lock->flag[id] = false;
}
```

```
void lock(lock_t lock, int id) {
  lock->flag[id] = true;
  while (lock->flag[1-id] == true) { // spin
      lock->flag[id] = false;
      lock->flag[id] = true;
  }
}
```

- The problem is **livelock!**
  - A kind of deadlock in which threads are in an infinite (or very long) sequence of blocking and unblocking, like people in a hallway
- Threads could be in locked step
  - They both set their flags to true
  - They both set their flags to false
  - Repeat . . .
- With false concurrency, this is virtually impossible (but probability $\neq 0$)
  - With true concurrency, the livelock is a bit likelier
- Let's look at another idea…

# Software Spin Lock: v5

```
void unlock(lock_t lock, int id) {
  lock->flag[id] = false;
  lock->turn = 1-id;
}
```

```
void lock(lock_t lock, int id) {
  lock->flag[id] = true;
  while (lock->flag[1-id] == true) {
    if (lock->turn != id) {
      lock->flag[id] = false;
      while (lock->turn != id) {} // spin
      lock->flag[id] = true;
    }
  }
}
```

- We add a "turn" variable to the lock structure

```
typedef struct {
  boolean flag[2];
  int turn;
} *lock_t;
```

- The threads take turns backing off
- This is a very good solution [Dekker, 1960's]
  - But it does allow starvation in some situations

# Software Spin Lock: v6

- In 1981 Peterson came up with a complete *and simpler* solution:

```
typedef struct {
    boolean flag[2];
    int last;
} *lock_t;
```

- The *last* field tracks which thread last tried to enter the CS
- This is the thread that is delayed if both threads compete
  - Removes the starvation problem of v5

```
void unlock(lock_t lock, int id) {
  lock->flag[id] = false;
}
```

```
void lock(lock_t lock, int id) {
  lock->flag[id] = true;
  lock->last = id;
  while (lock->flag[1-id] == true && lock->last == id) {} // spin
}
```

# Software Locks: Bottomline

- Producing a good solution requires a lot of thought
- Thanks to Peterson we have one
  - Formally proving that it is a correct solution is not easy
    - But in this course we don’t touch theory
  - Just know that detecting race conditions, deadlocks and starvations by analyzing code is NP-hard
- But what about more than 2 threads?
- Turns out things get much more complicated but doable
- The **bakery algorithm** (by Lamport)
  - Analogous to a bakery with a machine dispensing tickets to customers
  - Cleverly designed to avoid all the problems we have seen with v1, v2, v3, v4, and v5
  - Accommodates an arbitrary number of threads

# Asking the Hardware for Help

- The software solutions are interesting
  - Especially because the same principles and reasoning applies when writing concurrent applications that use locks
  - You’re not expected to remember these solutions in this course
  - But we will do similar analyses of user-level code for correctness (good luck everyone!)
- But they can be time/memory consuming
  - lock() has quite a few instructions
  - lock_t has quite a few bytes
- Common trend in the history of computing: hardware solutions are simpler and faster than software solutions
  - e.g., hardware floating point, virtualization hardware support

# Atomic instructions

- Let's look at our first naive implementation

```
void lock(int *lock) {
    while (*lock) {}   // spin
    *lock = 1;
}
```

- The assembly in RISC-like x86 assembly:

```
spin:     mov R1, [lock]     // Load lock
          cmp R1, 0          // compare to 0
          jnz  spin          // if not 0, loop
          mov [lock], 1      // set lock to 0
```

- Therefore, between the *loading*, the *testing* and the *setting* the value may have changed, because a sequence of instructions is not atomic
- We need an atomic "test and act" instruction!

# Compare-and-Swap Instruction

- Most processors provide atomic instructions that do multiple things at once
- One such instruction is Compare and Swap (CAS)
- CAS(location, old, new) does atomically:
  - if [location] == old, then [location] = new;
  - return true if value was changed;
- You could think of this implemented in hardware by locking the memory bus so that no other memory access can occur in between the load, the test, and store
  - That is, the content of memory cannot be changed by another thread while a thread is doing a CAS
  - In reality, the implementation is a bit more clever and leverages “cache coherency protocols”, so that not all memory operations are blocked

# Spinlock with CAS

- With the CAS instruction, one can then write the pseudo-code for lock():

  **while (CAS(lock, 0 , 1) == false) { }**

- In words: if the lock is set to 0 then set it to 1 and break from the loop, otherwise try again
- Fixes our first, simplest implementation with the help of the hardware
  - It only works because CAS is atomic
- And it's really fast!

# Spinning?

- In everything we've talked about so far, our implementation of the lock() function "spins" in loop
- That's why our lock is called a **spinlock**
- Spinning is good because one gets the lock as soon as it is released
- But since it's always a good idea to have short critical sections, then spinning isn't bad since no thread will spin for a long time
  - If the critical section were to be long the threads will spin for a log time, wasting of CPU cycles (and power / heat)
  - Think of a bathroom analogy again: if the person in there will be there for an hour, it's wasteful to stay by the door and keep trying to turn the handle!
- So we're all good and don't need anything else?

# Spinning is Bad?

- Unfortunately, critical sections cannot always be made short
  - e.g., they involve some network operation, some I/O operation
- We really, really don’t want to spin for a long time due to waste of CPU cycles
- And so, this is why we have Blocking Locks
  - You should have seen them in ICS332

# Blocking Locks (Mutexes)

- A radically different option in which the OS is involved
- The lock() function is modified so that if the lock is taken, instead of spinning, the thread is put to "sleep" by the OS
  - More precisely, the thread is removed from the ready queue and put in a queue associated to the lock
- When the lock is released via unlock(), the OS puts the thread back into the ready queue
  - The thread will eventually re-attempt to acquire the lock and may get it, or will be put back to sleep
- If the critical section is short, a blocking lock has very high overhead
  - Essentially, a system call + context-switch is involved when you could have instead been spinning for only a few cycles
- But, if the lock is taken for a long time, then no CPU cycles are wasted spinning

# Spinlock vs. Blocking Locks

| | short critical section | long critical section |
|---|---|---|
| **spinlocks** | ✔ | many wasted CPU cycles |
| **blocking locks** | high overhead | ✔ |

- Both types of locks are available on most systems

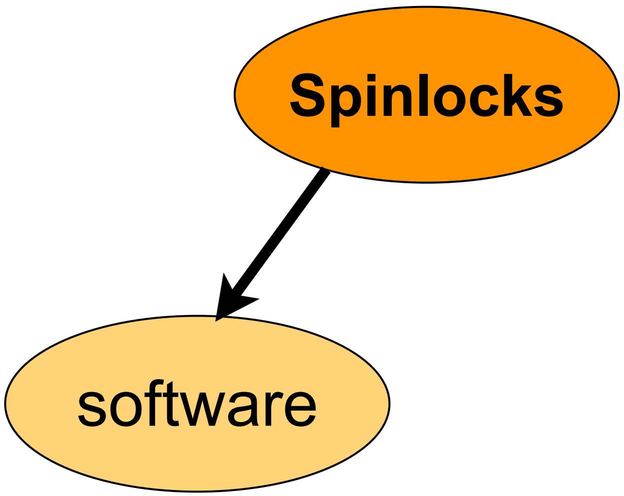
# Choosing?

- Sometimes the duration of a critical section is clear:
  - add 1 to a counter: short -> use a spinlock
  - update a database: long -> use a blocking lock
- But in many cases it's not easy to tell
- For this reason, most systems provide hybrid locks
  - First behaves like a spinlock
  - If spinning too long, then behaves like a blocking lock
  - Plus other custom behaviors that aim to strike a good compromise between CPU waste and responsiveness
- Typically, it's a great idea to use the provided hybrid locks
  - What Java provides by default, pthread_mutex_t in C/Pthreads, ...
- For instance, one some systems, even with short "x++" critical sections, I've found hybrid locks to be better than spinlocks in terms of performance!

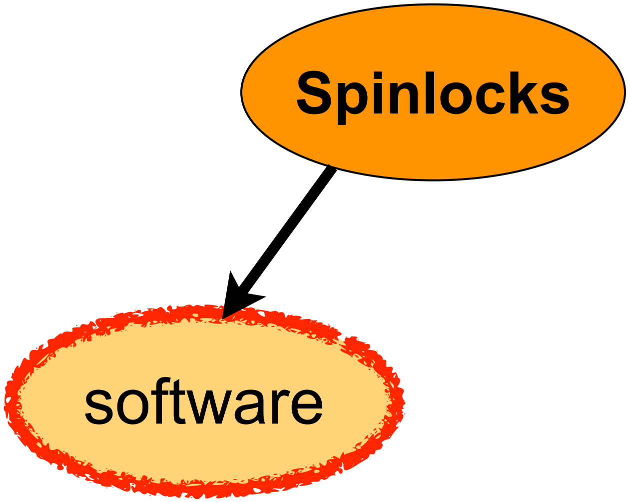
# Recap

Spinlocks

# Recap

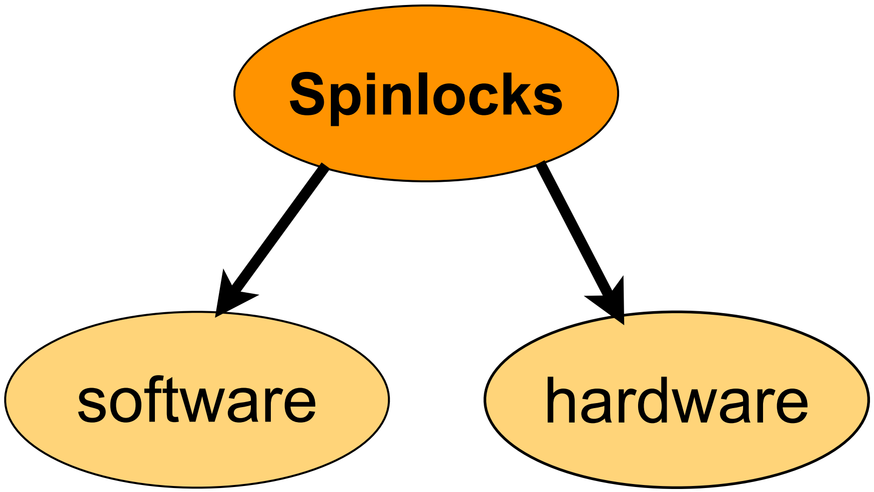


# Recap

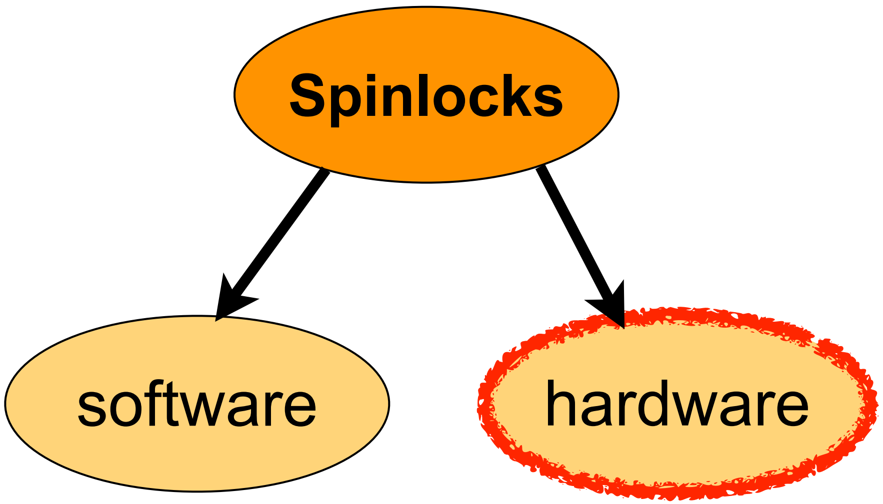



- ➡ Complicated, but solved by smart people decades ago
- ➡ Not efficient in terms of CPU and RAM

# Recap

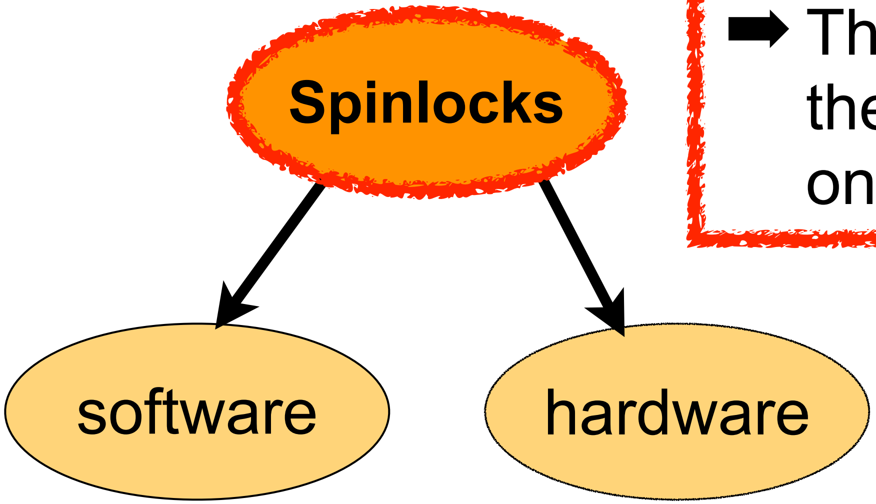


# Recap



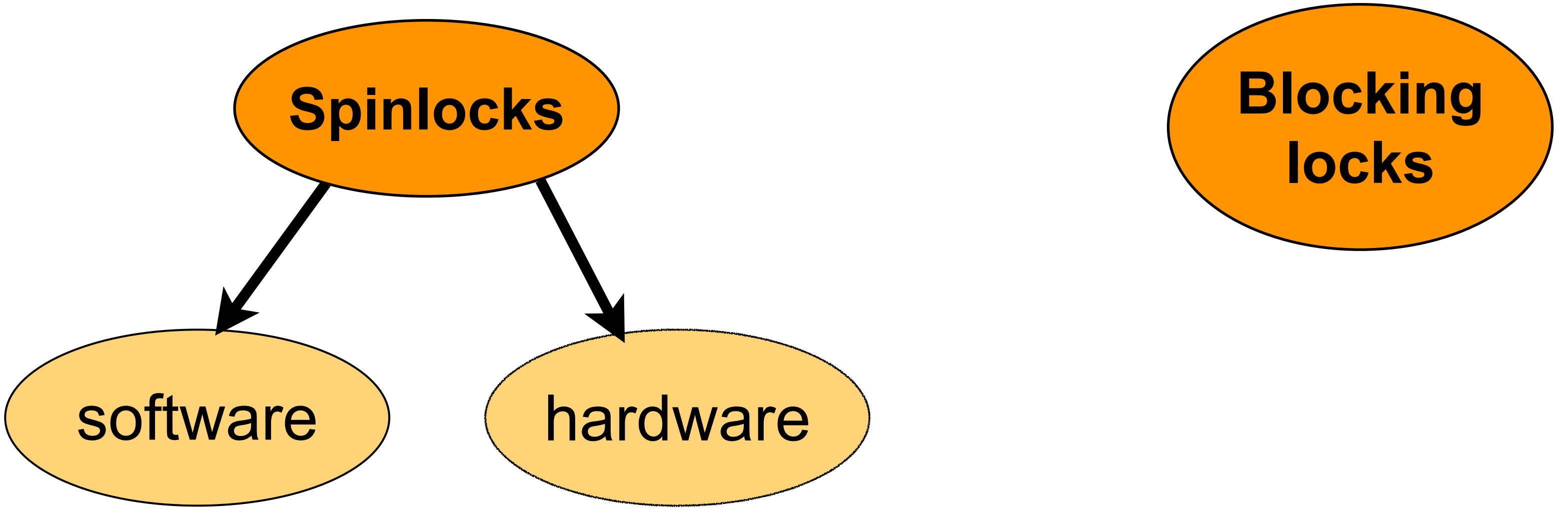

- ➡ Easy once processors provided atomic “compare and swap” instructions
- ➡ Efficient in terms of CPU and RAM

# Recap

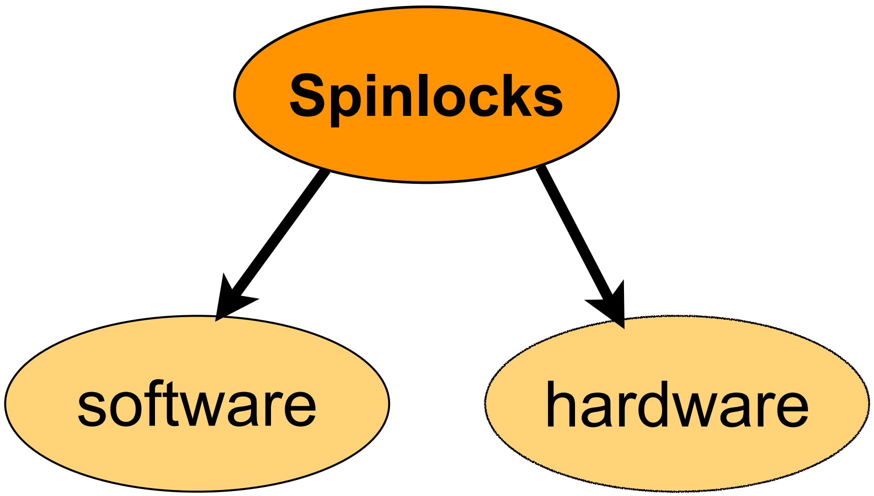



- The Good: one gets the lock as soon as it becomes available
- The Bad: while waiting for the lock to become available one wastes CPU cycles

# Recap



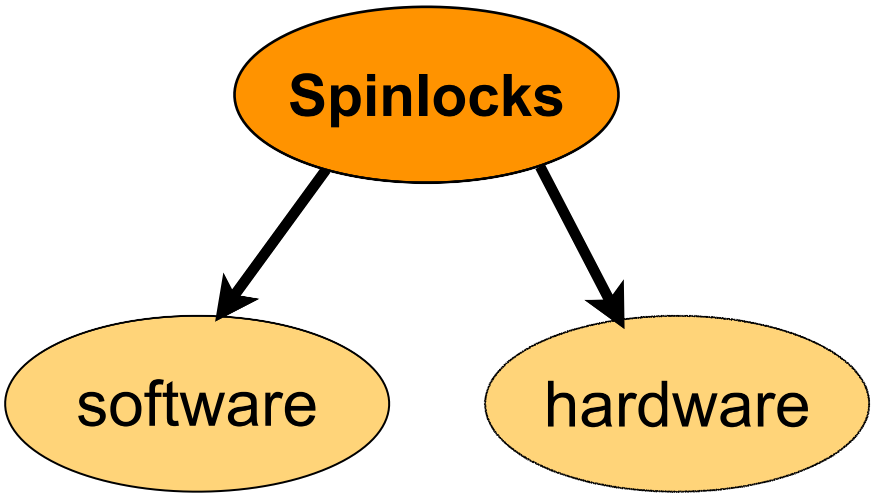
# Recap




**Blocking locks**

- Instead of spinning, the thread is put to sleep by the OS and re-awakened when the lock becomes available

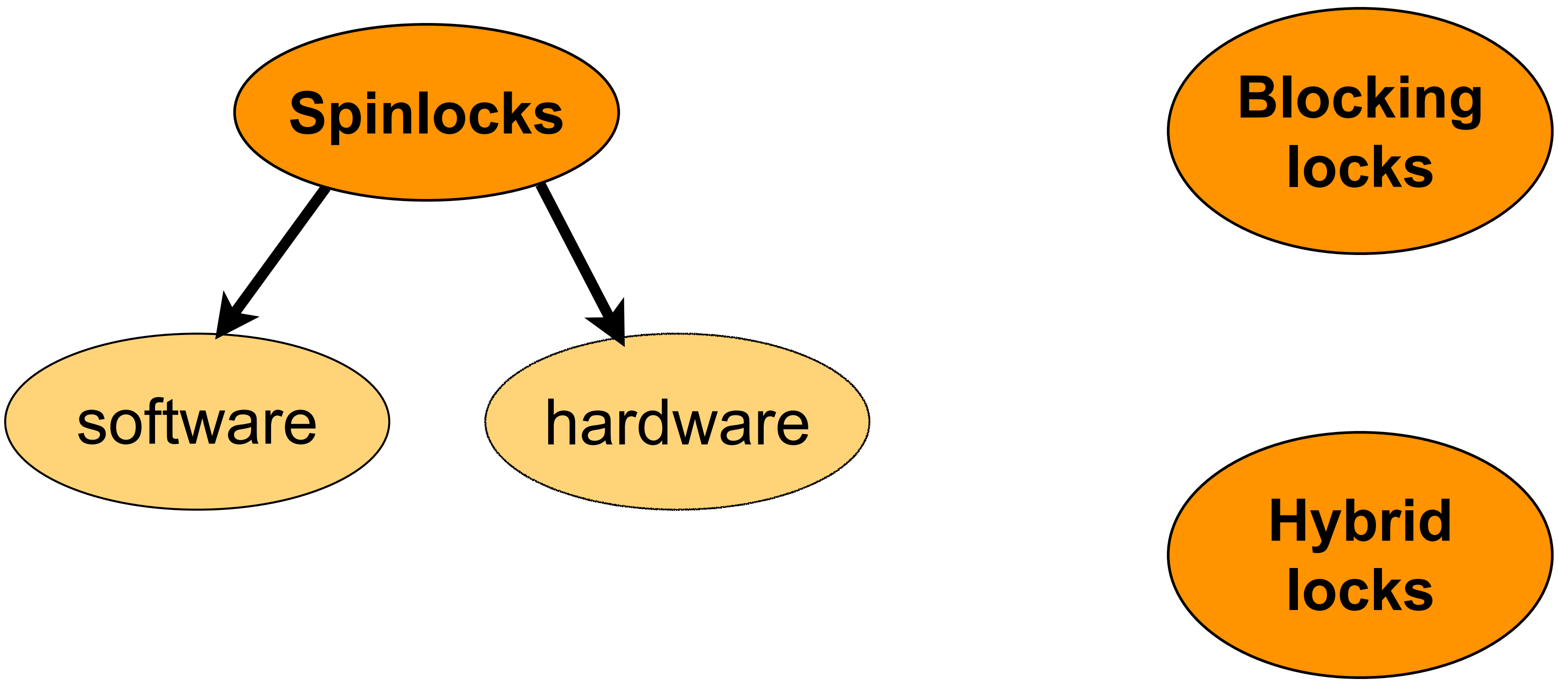
# Recap



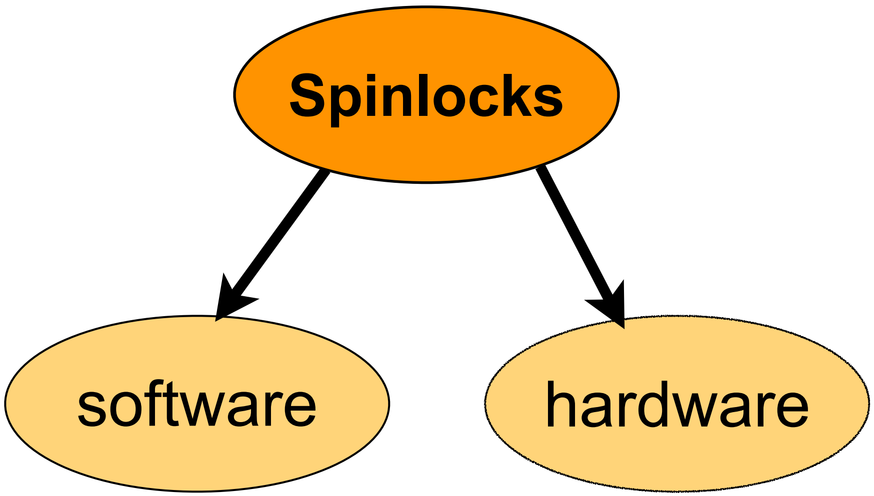

**Blocking locks**

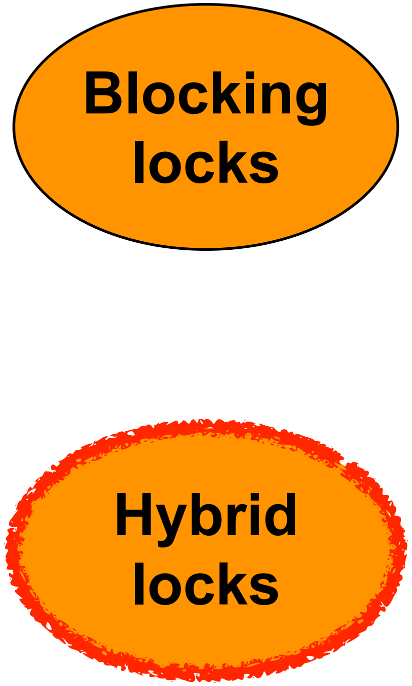
- ➡ The Good: no waste of CPU cycles while a thread is sleeping
- ➡ The Bad: high overhead

# Recap



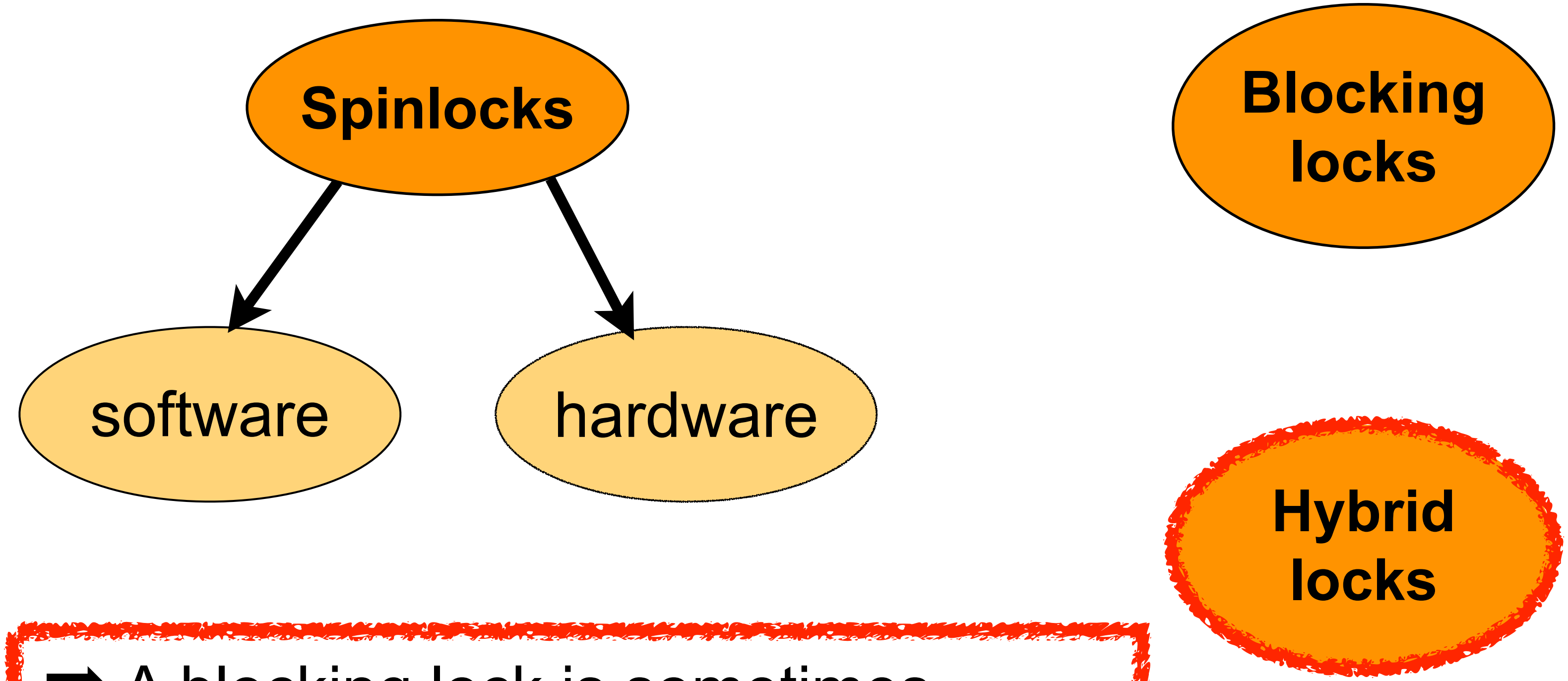
# Recap




➡ Adaptive locks that spin and/or block threads to achieve a good compromise between overhead and responsiveness

# Recap




➡ A blocking lock is sometimes called “mutex”
(due to the Pthread API in C…)

# A real-life metaphor

- You're a thread and you are in a coffee shop with a single bathroom, and many other threads
- Spinlock:
  - I go to the bathroom, I wait in line, when I get first in line I keep turning the handle until it opens and get in immediately(-ish)
- Blocking lock
  - I go to the bathroom, I see it's busy, I go to the barista and say "Can you come get me when the bathroom is free" and I go back to my table where I take a nap. Later, the barista comes by and tells me I can go in the bathroom (provided nobody got in there in the meantime… more on this later)
- Hybrid lock
  - I go to the bathroom, try the "spinlock" thing for 5 seconds in case I am lucky and the person inside is just about to finish. After 5 seconds I give up and try the "blocking lock" thing

# Conclusion

- Locks are used to create critical sections
- Three kinds of locks
  - Spinlock
  - Blocking locks
  - Hybrid locks
- These locks ALL do the same things, but achieve different levels of responsiveness and CPU cycle consumption
- Different locks are good for different critical section lengths
- In practice, systems provide good hybrid locks that most users use
- Onward to what Java does!