CSE 511: Operating Systems Design

Lectures 22

TAS and CAS

The slides are partially based on Maurice Herlihy's slides from "The Art of Multiprocessing"

What Should you do if you can't get a lock?

Keep trying

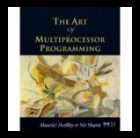
"spin" or "busy-wait"

Good if delays are short

Yield the processor

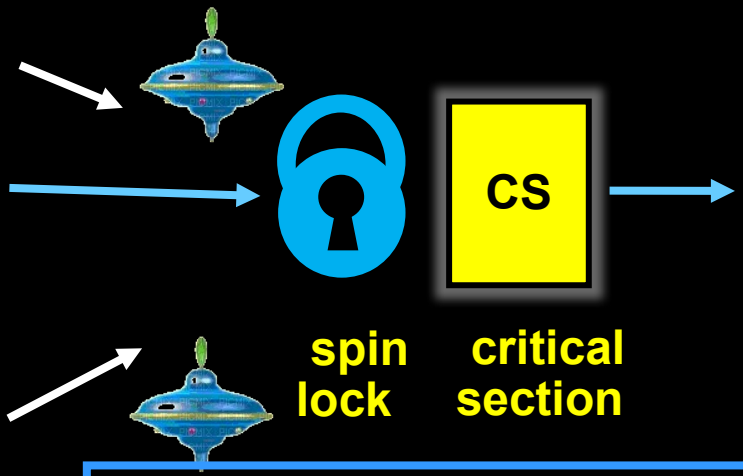
Good if delays are long

Always good on uniprocessor

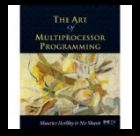


Basic Spin-Lock

lock means sequential bottleneck



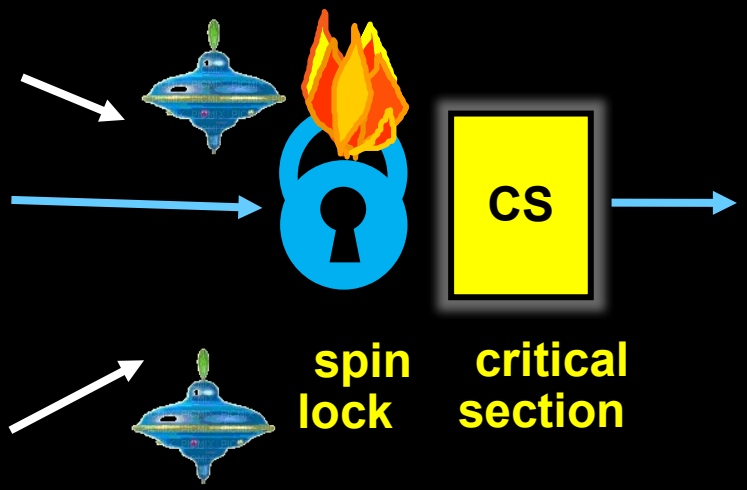
Bottleneck means no parallelism



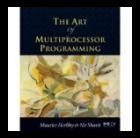
Basic Spin-Lock

Lock suffers from contention

Honk!



Honk!



Test-and-Set Lock

Boolean state

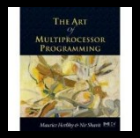
Lock: Test-and-set (TAS)

Swap *true* with current value

Returns prior value

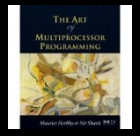
Unlock: write *false*

TAS sometimes called “getAndSet”



Test-and-Set

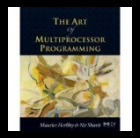
```
bool getAndSet(bool *globalVal, bool newValue) {
    /* Done atomically (as a single instruction)! */
    bool prior = *globalValue;
    *globalValue = newValue;
    return prior;
}
```



Test-and-Set

```
bool getAndSet(bool *globalVal, bool newValue) {
    /* Done atomically (as a single instruction)! */
    bool prior = *globalVal;
    *globalVal = newValue;
    return prior;
}
```

Swapping in *true* is called "*test-and-set*" or TAS
(usually a HW machine instruction)

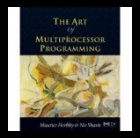


TAS-based Lock

```
struct spinlock {
   bool value;
};

struct spinlock lock = { false };

void spinlock_lock(struct spinlock *lock) {
    while (getAndSet(&lock->value, true)
             == true) {
        /* Retry */
    }
    /* The previous value was false! */
}
```

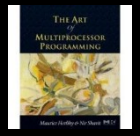


TAS-based Lock

```
struct spinlock {
   bool value;
};

struct spinlock lock = { false };

void spinlock_unlock(struct spinlock *lock) {
    lock->value = false;
}
```



Test-and-Set Locks

Lock is free: value is *false*

Lock is taken: value is *true*

Spin: repeatedly call TAS

When result is *false*, stop

While result is *true*, try again

Release lock by writing *false*

Space Complexity

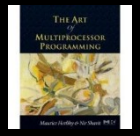
TAS spin-lock has small “footprint”

n thread spin-lock uses O(1) space

As opposed to O(*n*) Peterson/Bakery*

* Please review these algorithms for classical RW-approaches to implement locks

TAS is RMW, not RW …



Compare-and-Set

```
bool compareAndSet(long *globalVal, long oldValue,
      long newValue)
{
   /* Done atomically (as a single instruction)! */
   long prior = *globalVal;
   if (prior == oldValue) {
      *globalVal = newValue;
      return true;
   } else {
      return false;
   }
}
```