



**SYSTEMS
GOSSIP
MEETUP**

Learning large systems using peer-to-peer gossip

Policy Against Harassment at ACM Activities

<https://www.acm.org/about-acm/policy-against-harassment>

OS Meetup wants to encourage and preserve this open exchange of ideas, which requires an environment that enables all to participate without fear of personal harassment. We define harassment to include specific unacceptable factors and behaviors listed in the ACM's policy against harassment. Unacceptable behavior will not be tolerated.

Today

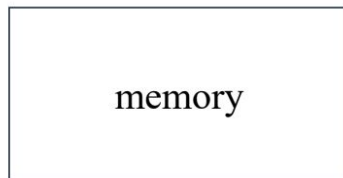
- Multicore
- Concurrency
- Lock
 - Abstraction
 - Implementation
- Xv6 Locks
- Locks and Interrupts
- Deadlock

Multicore Processing

Modern Multi-core Processors

CPU	CPU	CPU	CPU
L1	L1	L1	L1
L2	L2	L2	L2
L3			

on-chip
cache(s)



main memory



disk

disk

Concurrency

- Concurrency is a situation where multiple instructions are interleaved, due to multicore parallelism, thread switching or interrupts.
- Instructions are interleaved arbitrarily and at different rates.
- We need a way to control concurrency for the correctness of the program

Concurrency Problem

CPU 1



```
withdraw(account, amount) {  
    balance = get_balance(account);  
    balance = balance - amount;  
    put_balance(account, balance);  
    return balance;  
}
```

CPU 2



```
withdraw(account, amount) {  
    balance = get_balance(account);  
    balance = balance - amount;  
    put_balance(account, balance);  
    return balance;  
}
```

Concurrency Problem

CPU 1

CPU 2

CPU 1



```
withdraw(account, amount) {  
    balance = get_balance(account);  
    balance = balance - amount;  
  
    balance = get_balance(account);  
    balance = balance - amount;  
    put_balance(account, balance);  
  
    put_balance(account, balance);  
    ...  
}
```

Concurrency Problem

```
1  struct element {
2      int data;
3      struct element *next;
4  };
5
6  struct element *list = 0;
7
8  void
9  push(int data)
10 {
11     struct element *l;
12
13     l = malloc(sizeof *l);
14     l->data = data;
15     l->next = list;
16     list = l;
```

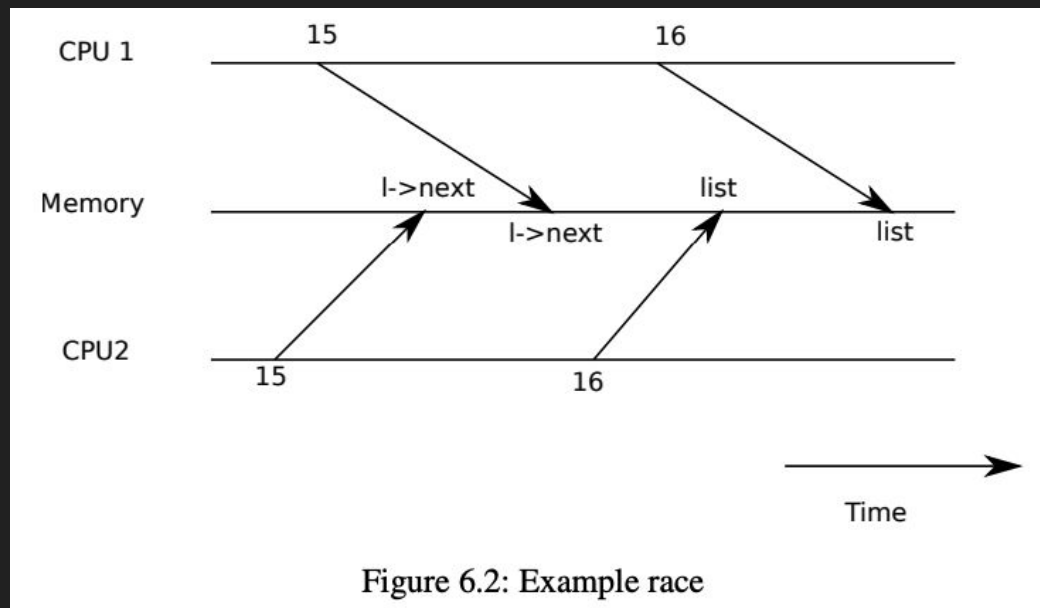


Figure 6.2: Example race

Concurrency Control Mechanism

- Lock is a concurrency control mechanism. It is an abstraction to support correctness under concurrency. It must provide **mutual exclusion**, meaning that only one CPU can hold a lock at a time.
- The downside of holding a lock is that execution of a sequence of instructions within the lock section is **serialized**. This prevents parallelism and kills **performance**.
- We often call the sequence of instructions between lock a **critical section**.

Lock Abstraction



```
struct Lock { uint locked }  
  
// only one process can acquire lock.  
acquire(&l)  
//  
//critical section  
//  
release(&l)
```

Lock Abstraction

Locked section can be viewed in different perspectives:

- Avoid **lost updates**
- Make multi-step operations **atomic**
- Helps concurrency data structures to maintain **invariants**.



```
struct Lock { uint locked }  
  
// only one process can acquire lock.  
acquire(&l)  
//  
//critical section  
//  
release(&l)
```

Lock Abstraction

Locked section can be viewed in different perspectives:

- Avoid **lost updates**
- Make multi-step operations **atomic**
- Helps concurrency data structures to maintain **invariants**.




```
struct Lock { uint locked }  
  
// only one process can acquire lock.  
acquire(&l)  
//  
//critical section  
//  
release(&l)
```

(spin)Lock Implementation (Wrong)



```
struct spinlock { uint locked; }  
void acquire(struct spinlock *l) {  
    for(;;){  
        if(l->locked == 0){ // A  
            l->locked = 1;    // B  
            break;  
        }  
    }  
}
```

(spin)Lock Implementation (Wrong)




```
struct spinlock { uint locked; }  
void acquire(struct spinlock *l) {  
    for(;;){  
        if(l->locked == 0){ // A  
            l->locked = 1;    // B  
            break;  
        }  
    }  
}
```

Context
switch



(spin)Lock Implementation (Wrong too)



```
struct lock { uint locked; }  
void acquire(struct lock *l) {  
    //disable interrupt  
  
    for(;;) {  
        if (l->locked == 0) {  
            l->locked = 1;  
            break;  
        }  
    }  
}
```

(spin)Lock Implementation (Wrong too)

```
struct lock { uint locked; }  
void acquire(struct lock *l) {  
    //disable interrupt  
  
    for(;;) {  
        if (l->locked == 0) {  
            l->locked = 1;  
            break;  
        }  
    }  
}
```

This might be sufficient for locking single core multi-threading, but not for multicore

(spin)Lock Implementation

- The problem: lock implementation also has critical sections.
- What we need? Atomic read-modify-write operation.
- What's atomic? A sequence of instructions that cannot be interrupted
- How do we make them atomic?
- Hardware support
 - RISC-V has **amoswap r, a** instruction
 - It reads memory a, writes the content of register r to that address

(spin)Lock Implementation (Correct)

```
// Acquire the lock.
// Loops (spins) until the lock is acquired.
void
acquire(struct spinlock *lk)
{
    push_off(); // disable interrupts to avoid deadlock.
    if(holding(lk))
        panic("acquire");

    // On RISC-V, sync_lock_test_and_set turns into an atomic swap:
    //  a5 = 1
    //  s1 = &lk->locked
    //  amoswap.w.aq a5, a5, (s1)
    while(__sync_lock_test_and_set(&lk->locked, 1) != 0)
        ;

    // Tell the C compiler and the processor to not move loads or stores
    // past this point, to ensure that the critical section's memory
    // references happen strictly after the lock is acquired.
    // On RISC-V, this emits a fence instruction.
    __sync_synchronize();

    // Record info about lock acquisition for holding() and debugging.
    lk->cpu = mycpu();
}
```

```
// Release the lock.
void
release(struct spinlock *lk)
{
    if(!holding(lk))
        panic("release");

    lk->cpu = 0;

    // Tell the C compiler and the CPU to not move loads or stores
    // past this point, to ensure that all the stores in the critical
    // section are visible to other CPUs before the lock is released,
    // and that loads in the critical section occur strictly before
    // the lock is released.
    // On RISC-V, this emits a fence instruction.
    __sync_synchronize();

    // Release the lock, equivalent to lk->locked = 0.
    // This code doesn't use a C assignment, since the C standard
    // implies that an assignment might be implemented with
    // multiple store instructions.
    // On RISC-V, sync_lock_release turns into an atomic swap:
    //  s1 = &lk->locked
    //  amoswap.w zero, zero, (s1)
    __sync_lock_release(&lk->locked);

    pop_off();
}
```

Reentrant/Recursive Locks



```
acquire(&l);  
acquire(&l);
```

- If the lock is held by a process, and if that process attempts to acquire the lock again, then the kernel could just allow this.
- Xv6 uses non-reentrant locks



```
struct spinlock l;  
int data = 0;  
  
f() {  
    acquire(&l);  
    if (data == 0) {  
        call_once();  
        h();  
        data = 1;  
    }  
    release(&l);  
}  
  
g() {  
    acquire(&l);  
    if (data == 0) {  
        call_once();  
        data = 1;  
    }  
    release(&l);  
}
```

Locks and Interrupts

- Why do we need to disable interrupt in `acquire(&l)`?
- Spinlock protects data that is used by both processes and interrupt handlers.
- An example: tickslock in `sys_sleep()`

Deadlocks



```
lockA → acquire();  
...  
lockB → acquire();
```



```
lockB → acquire();  
...  
lockA → acquire();
```

Deadlock exists among a set of processes if every process waiting for an event that can be caused only by another process in the set

Ordering of locks

Deadlock exists among a set of processes if every process waiting for an event that can be caused only by another process in the set

“Callers must invoke functions in a way that causes locks to be acquired in the agreed-on order.”

Maintaining a global order of locks is complex!

Next time

1. Lab 4 Traps due today!
2. We will briefly cover Lab 4 next week
3. Lab 5 will be released today!
4. Lecture 11: Scheduling