Locks (ch. 28) pt. 2

Operating Systems

Based on: Three Easy Pieces by Arpaci-Dusseaux

Moshe Sulamy

Tel-Aviv Academic College

User mode

Pthread Locks

- POSIX library: **mutex (mutual exclusion)**

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
...
pthread_mutex_lock(&mutex); // may fail!
balance = balance + 1;
pthread_mutex_unlock(&mutex);
```

Pthread Locks

- POSIX library: **mutex (mutual exclusion)**

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
...
pthread_mutex_lock(&mutex); // may fail!
balance = balance + 1;
pthread_mutex_unlock(&mutex);
```

- Variable passed to lock and unlock
  - May use different locks for different sections
  - **Coarse-grained** locking: one big lock
  - **Fine-grained**: use various locks for different sections

Building A Lock

- Efficient locks provide mutual exclusion at low cost (overhead)
  - Support from hardware and the OS

How can we build an efficient lock?

Issues compared to kernel

- Can not disable interrupts
- Can not use busy-waits
  - Scheduling out while in busy-wait is catastrophic.
- So, what do we do? We ask for the kernel for help

yield

- Assume there is a yield system call.

```
lock(int *mutex) {
  while (tas(mutex,1)) yield();
}

unlock(int *mutex) {
  *mutex = 0;
}
```

Good?

- It works.
- We depend on the scheduler.
- Quite a lot of system calls (yield) might occur.
- So? If we call the kernel, lets call it for actual work.

Kernel support for mutexes

Three system calls:

```
int handle = createMutex();

lockMutex(handle);

unlockMutex(handle);
```

Kernel Implementation: Data structure

```
struct userMutex {
  int lock;
  int hardMutex;
  struct queue *queue;
}
```

Kernel Implementation: Lock syscall

```
sysLock(struct userMutex *userMutex) {
    hardLock(&userMutex->hardMutex);
    if (userMutex->lock == 1) {
        queueAdd(&userMutex->queue, proc);
        BLOCK(&userMutex->hardMutex);
    }
    userMutex->lock = 1;
    hardUnlock(&userMutex->hardMutex);
}
```

Kernel Implementation: unlock syscall

```
sysUnock(struct userMutex *userMutex) {
    hardLock(&userMutex->hardMutex);

    if (userMutex->queue != NULL)
        int p = queueRemove(&userMutex->queue;
        UNBLOCK(p);
    }
    userMutex->lock = 0;
    hardUnlock(&userMutex->hardMutex);
}
```

Good?

- It works.
- (For many years this was the standard way)
- What is the problem?
  - Many switches to the kernel.
  - This is considerable overhead for nowadays applications.
  - Most of the time there is no lock contention.
    - Can we exploit this phenomenon?

Futex

- Linux: **futex**
  - `futex_wait(address,expected)`
    - Puts calling thread to sleep if `address` is equal to `expected`
  - `futex_wake(address)`
    - Wakes one thread waiting on `address`

Using Queues: Different OS

- Snippet from POSIX thread library:

```
void mutex_lock(int *mutex) {
    int v;
    // Bit 31 was clear, we got the mutex (fastpath)
    if (atomic_bit_test_set(mutex, 31) == 0)
        return;
    atomic_increment(mutex);
    while (1) {
        if (atomic_bit_test_set(mutex, 31) == 0) {
            atomic_decrement(mutex);
            return;
        }
        v = *mutex;
        if (v >= 0)
            continue;
        futex_wait(mutex, v);
    }
}
void mutex_unlock(int *mutex) {
    if (atomic_add_zero(mutex, 0x80000000))
        return; // zero iff no other interested threads

    // there are other threads waiting
    futex_wake(mutex);
}
```

Good?

- It works.
- Moreover, if there is no contention then we stay in user mode!

Just Using Loads/Stores

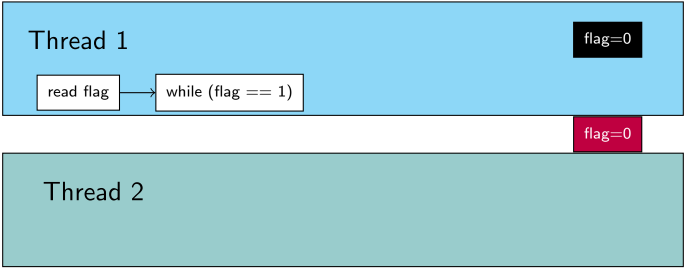
- No mutual exclusion

Just Using Loads/Stores

- No mutual exclusion

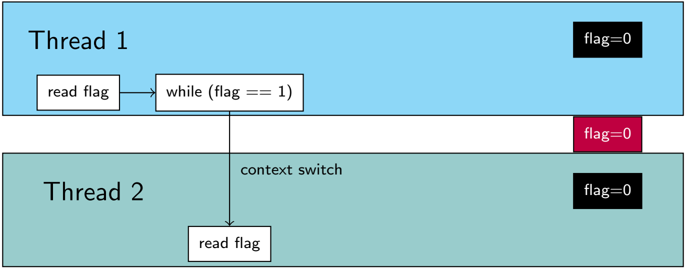
Just Using Loads/Stores

- No mutual exclusion



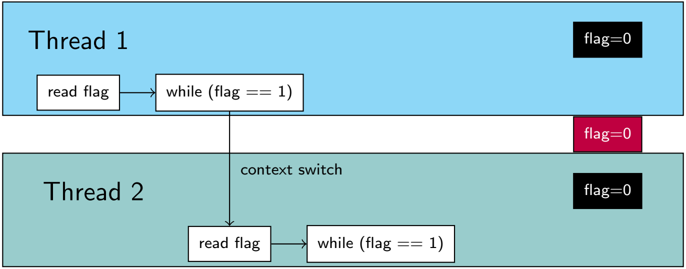
Just Using Loads/Stores

- No mutual exclusion



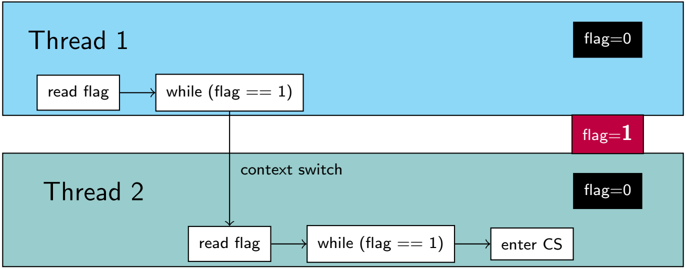
Just Using Loads/Stores

- No mutual exclusion



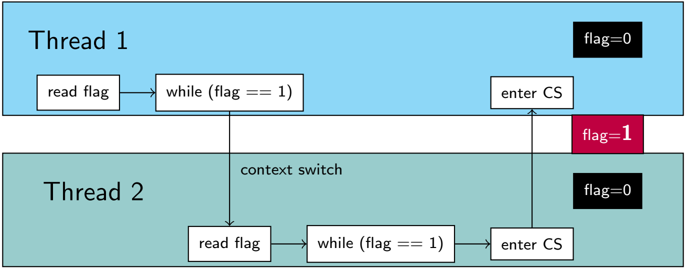
Just Using Loads/Stores

- No mutual exclusion



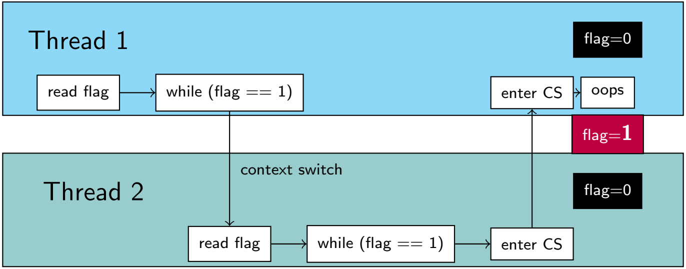
Just Using Loads/Stores

- No mutual exclusion



Just Using Loads/Stores

- No mutual exclusion



Evaluating Spin Locks

- Correctness (**mutual exclusion**)?

Evaluating Spin Locks

- Correctness (**mutual exclusion**)? yes
- **Deadlock-freedom**?

Evaluating Spin Locks

- Correctness (**mutual exclusion**)? yes
- **Deadlock-freedom**? yes
- **Fairness**?

Evaluating Spin Locks

- Correctness (**mutual exclusion**)? yes
- **Deadlock-freedom**? yes
- **Fairness**? no
- Performance?
  - Single CPU:

Evaluating Spin Locks

- Correctness (**mutual exclusion**)? yes
- **Deadlock-freedom**? yes
- **Fairness**? no
- Performance?
  - Single CPU:painful
    - Owner thread is preempted, all $N-1$ others spin-wait needlessly
  - Multiple CPUs:

Evaluating Spin Locks

- Correctness (**mutual exclusion**)? yes
- **Deadlock-freedom**? yes
- **Fairness**? no
- Performance?
    - Single CPU:painful
        - Owner thread is preempted, all $N-1$ others spin-wait needlessly
    - Multiple CPUs: Might be reasonably well

Fetch-And-Add

- Final hardware primitive: **fetch-and-add**
- Atomically increment a value and return old value
- Defined as:

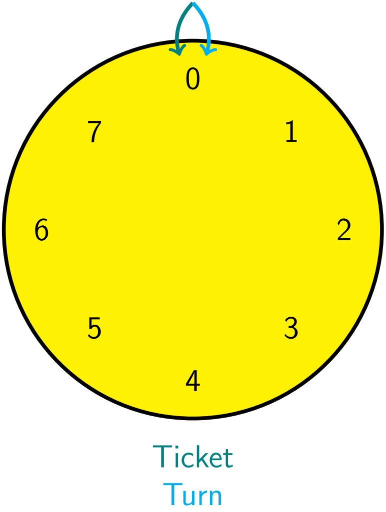
```
int FetchAndAdd(int* ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

Fetch-And-Add

- We can now build a fair **ticket lock**:

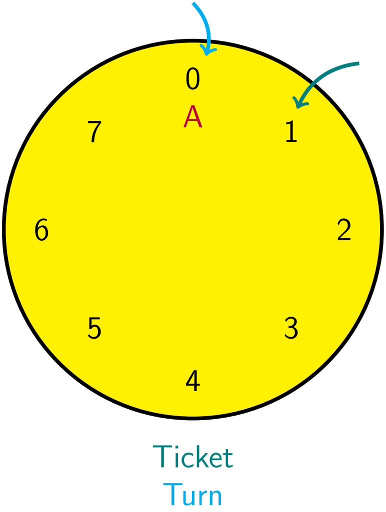
```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void init(lock_t* lock) {
    lock->ticket = 0;
    lock->turn = 0;
}
void lock(lock_t* lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
}
void unlock(lock_t* lock) {
    lock->turn = lock->turn + 1;
}
```

Ticket Lock



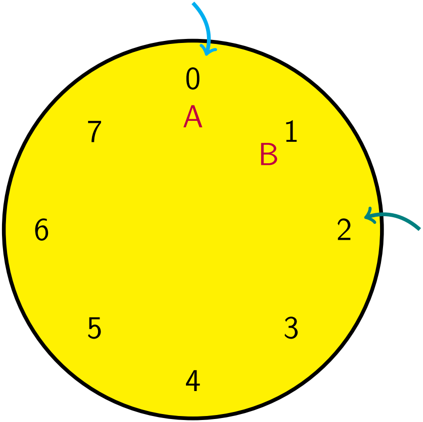
Ticket Lock

- A: lock(), gets ticket 0 & runs



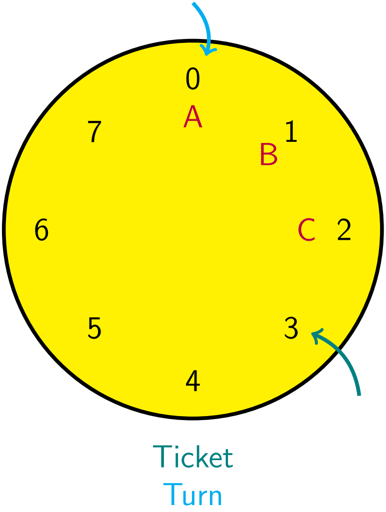
Ticket Lock

- A: lock(), gets ticket 0 & runs
- B: lock(), gets ticket 1, spins



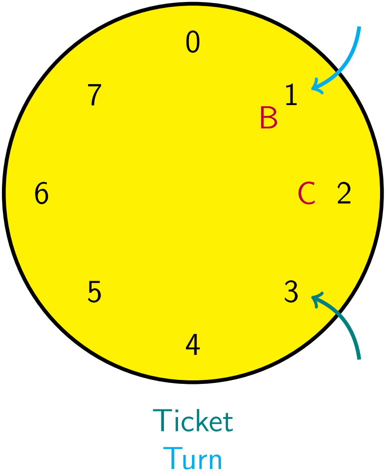
Ticket Lock

- A: lock(), gets ticket 0 & runs
- B: lock(), gets ticket 1, spins
- C: lock(), gets ticket 2, spins



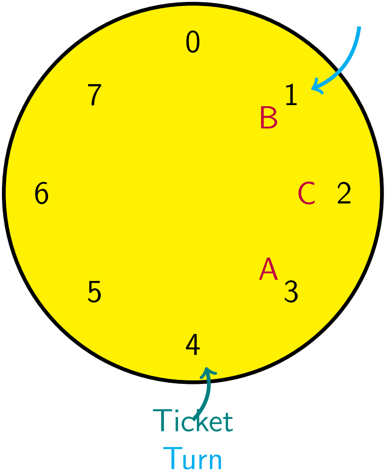
Ticket Lock

- A: lock(), gets ticket 0 & runs
- B: lock(), gets ticket 1, spins
- C: lock(), gets ticket 2, spins
- A: unlock(), turn++, B runs



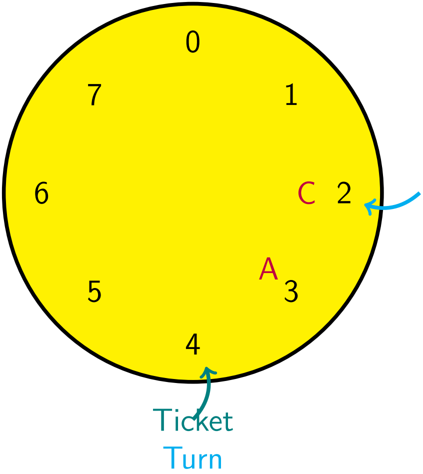
Ticket Lock

- A: lock(), gets ticket 0 & runs
- B: lock(), gets ticket 1, spins
- C: lock(), gets ticket 2, spins
- A: unlock(), turn++, B runs
- A: lock(), gets ticket 3, spins



Ticket Lock

- A: lock(), gets ticket 0 & runs
- B: lock(), gets ticket 1, spins
- C: lock(), gets ticket 2, spins
- A: unlock(), turn++, B runs
- A: lock(), gets ticket 3, spins
- B: unlock(), turn++, C runs
- ...



Two-Phase Locks

- Hybrid approach: **two-phase lock**
  - Spinning can be useful
  - Particularly if lock is about to be released
- **First phase**: lock spins for a while
- **Second phase**: caller put to sleep, wakes up when lock becomes free

Summary

- **Lock**
  - Execute a series of actions **atomically**
  - Evaluated by: **Mutual exclusion**, **Deadlock-freedom**, **fairness**, **performance**
  - POSIX library: **mutex**, **futex**
- Disabling interrupts: problematic, used by OS
- Hardware support: **test&set**, **compare&swap**, **fetch&add**
- Spin-locks: TAS lock & CAS lock
  - Avoid spinning with `yield()`
- Fairness: **ticket lock** or queue lock
- **Condition variables**