Locks

Operating Systems
Based on: Three Easy Pieces by Arpaci-Dusseaux

Moshe Sulamy

Tel-Aviv Academic College

Locks (ch. 28)

- **Concurrency** issues
  - Execute a series of instructions **atomically**
  - With interrupts and concurrent processors
- Introducing: a **lock**
  - Critical section seemingly executes atomically

Basic Idea

- **Lock variable**
  - Holds lock state
  - **Available** (or unlocked or **free**)
    - No thread holds the lock
  - **Acquired** (or **locked** or **held**)
    - Exactly one thread (**owner**) holds the lock
    - In a critical section

Basic Idea

- `lock()`
  - Try to acquire the lock
  - Will not return (or fail) if held by another thread
- `unlock()`
  - Lock is available again

Basic Idea

- Critical section:

```
balance = balance + 1;
```

- To use lock:

```
lock_t mutex; // lock variable
...
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Pthread Locks

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

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

Pthread Locks

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

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

- 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?

Evaluating Locks

- **Mutual exclusion**
  - At most one thread in the CS
- **Deadlock-freedom**
  - Some thread eventually enters CS
- **Fairness** (**starvation-freedom**)
  - Each thread eventually enters CS
- **Performance**
  - Time overhead for using the lock
  - Single thread: overhead for grab & release
  - Multiple threads and CPUs

Controlling Interrupts

- Early solution: disable interrupts
  - For single-processor systems
  - No clock interrupt / context switch in critical section
- The negatives:

Controlling Interrupts

- Early solution: disable interrupts
  - For single-processor systems
  - No clock interrupt / context switch in critical section
- The negatives:
  - Trust arbitrary (greedy, malicious, or faulty) programs
  - Does not work on multiprocessors
  - Lost interrupts
- Used by OS

Just Using Loads/Stores

```c
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t* mutex) {
    // 0: available, 1: locked
    mutex->flag = 0;
}
void lock(lock_t* mutex) {
    while (mutex->flag == 1)
        ; // spin-wait
    mutex->flag = 1;
}
void unlock(lock_t* mutex) {
    mutex->flag = 0;
}
```

Just Using Loads/Stores

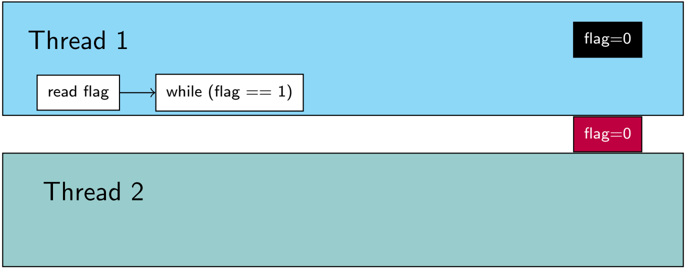
- No mutual exclusion

Just Using Loads/Stores

- No mutual exclusion

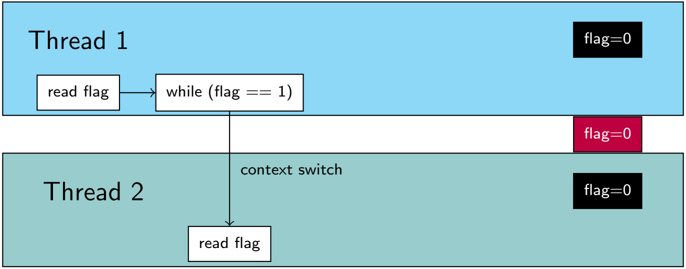
Just Using Loads/Stores

- No mutual exclusion



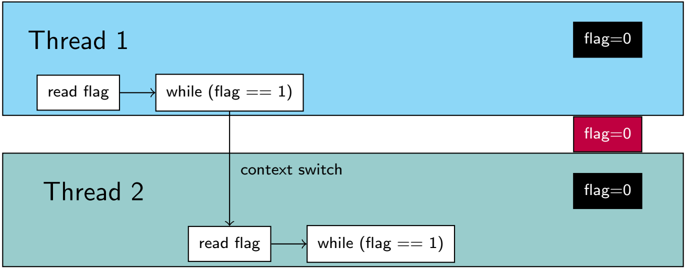
Just Using Loads/Stores

- No mutual exclusion



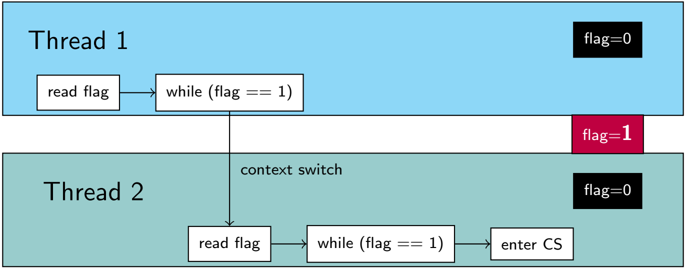
Just Using Loads/Stores

- No mutual exclusion



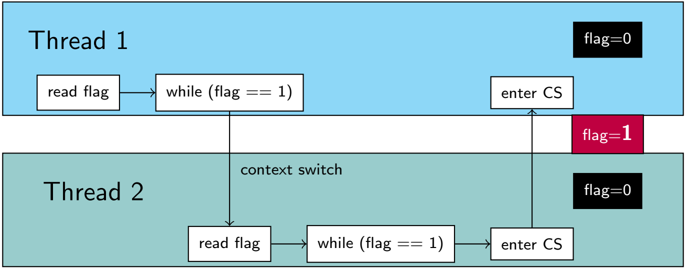
Just Using Loads/Stores

- No mutual exclusion



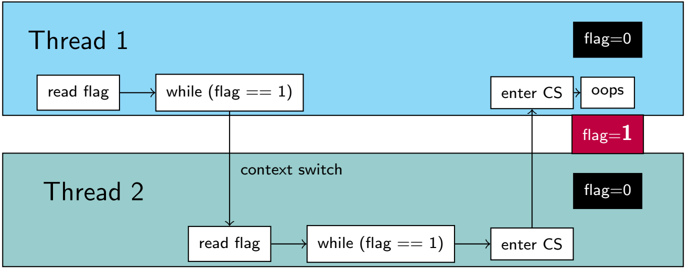
Just Using Loads/Stores

- No mutual exclusion



Just Using Loads/Stores

- No mutual exclusion



Test-And-Set

- Hardware support: a new instruction **test-and-set**
  - Update value and return previous, **atomically**
- Defined as:

```c
int TestAndSet(int* old_ptr, int new) {
    int old = *old_ptr;
    *old_ptr = new;
    return old;
}
```

New Spin Lock

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t* mutex) {
    // 0: available, 1: locked
    mutex->flag = 0;
}
void lock(lock_t* mutex) {
    while (TestAndSet(&mutex->flag, 1))
        ; // spin-wait
    mutex->flag = 1;
}
void unlock(lock_t* mutex) {
    mutex->flag = 0;
}
```

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

Compare-And-Swap

- Another hardware primitive: **compare-and-swap**
- Compare to `expected`, update <u>only</u> if equal, return previous
- Defined as:

```
int CompareAndSwap(int* ptr, int expected, int new) {
    int original = *ptr;
    if (original == expected)
        *ptr = new;
    return original;
}
```

Compare-And-Swap

- Spin-lock with CAS:

```
void lock(lock_t* lock) {
    while (CompareAndSwap(&mutex->flag, 0, 1) == 1)
        ; // spin
}
```

- Fairness? performance?

Compare-And-Swap

- Spin-lock with CAS:

```
void lock(lock_t* lock) {
    while (CompareAndSwap(&mutex->flag, 0, 1) == 1)
        ; // spin
}
```

- Fairness? performance?
  - Pretty much the same

Fetch-And-Add

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

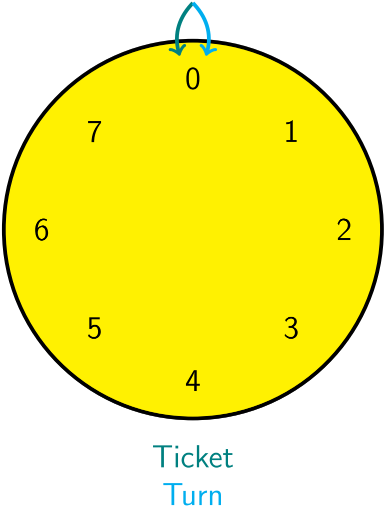
```
int FetchAndAdd(int* ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

Fetch-And-Add

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

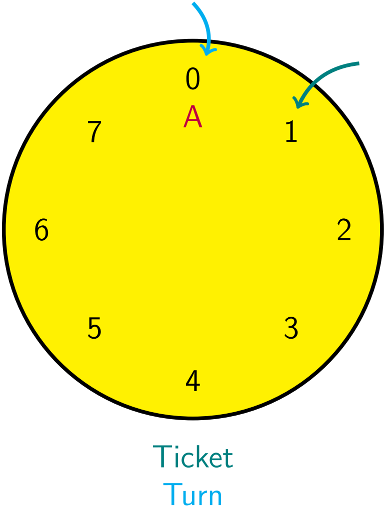
```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

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



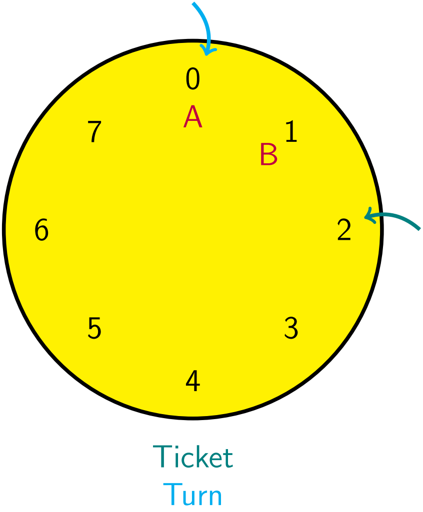
Ticket Lock

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



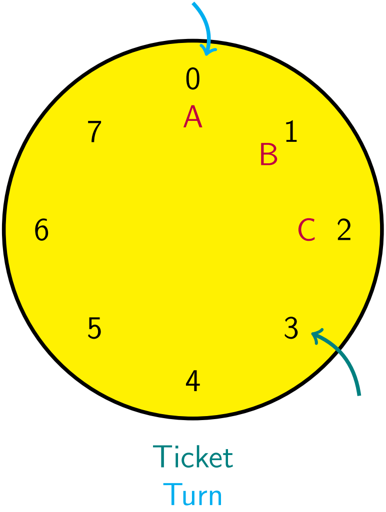
Ticket Lock

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



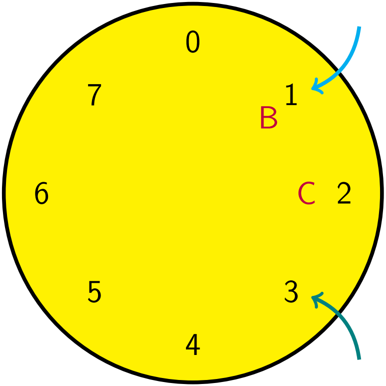
Ticket Lock

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



Ticket Lock

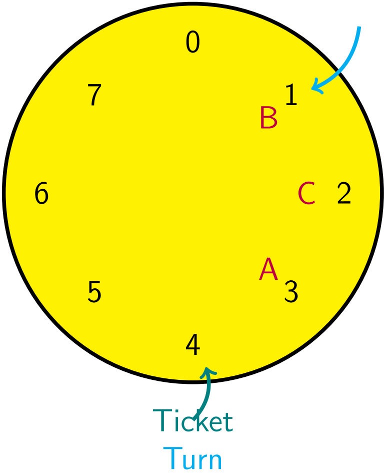
- A: lock(), gets ticket 0 & runs
- B: lock(), gets ticket 1, spins
- C: lock(), gets ticket 2, spins
- A: unlock(), turn++, B runs



Ticket
Turn

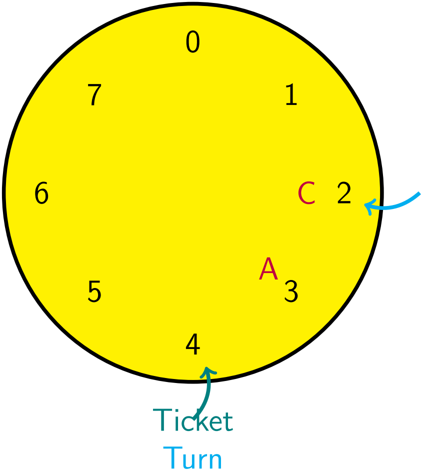
Ticket Lock

- 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
- ...



Decker's Algorithm

- What about a lock without hardware support?

```
int flag[2];     // wants to grab lock?
int turn;        // whose turn?

void init() {
    flag[0] = flag[1] = 0;
    turn = 0;
}
void lock(int self) {
    flag[self] = 1;
    turn = 1 - self;     // let other run
    while ((flag[1-self] == 1) && (turn == 1-self))
        ; // spin-wait
}
void unlock(int self) {
    flag[self] = 0;
}
```

- Various issues → concurrency course

Too Much Spinning

- Locks so far used spinning
  - Quite inefficient
- Consider N threads
  - Thread 1 grabs lock, $N - 1$ threads waiting for lock
  - Timer interrupt $\rightarrow$ context switch
  - $N - 1$ threads execute, waste $N - 1$ time slices
- Solution? hardware again!

Too Much Spinning

- When you are going to spin, give up CPU
  - `yield()`: system call to change caller state
    - From **running** to **ready**
    - Essentially **deschedules** itself

```
void lock(lock_t* mutex) {
    while (TestAndSet(&mutex->flag, 1))
        yield(); // give up the CPU
    mutex->flag = 1;
}
```

Too Much Spinning

- When you are going to spin, give up CPU
  - `yield()`: system call to change caller state
    - From **running** to **ready**
    - Essentially **deschedules** itself

```
void lock(lock_t* mutex) {
    while (TestAndSet(&mutex->flag, 1))
        yield(); // give up the CPU
    mutex->flag = 1;
}
```

- Still costly
  - $N - 1$ system calls and context switches
  - Does not handle **starvation**

Using Queues

- Use **queue** to keep track of threads waiting for lock
- In **Solaris**: `park()`
  - Put calling thread to sleep
  - Until another thread calls `unpark(threadID)`

Using Queues

- Use **queue** to keep track of threads waiting for lock
- In **Solaris**: `park()`
  - Put calling thread to sleep
  - Until another thread calls `unpark(threadID)`

```
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void init(lock_t *m) {
    m->flag = 0;
    m->guard = 0;
    queue_init(m->q);
}
```

Using Queues

```c
void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; // acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1;
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}
void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; // acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0;
    else
        unpark(queue_remove(m->q));
    m->guard = 0;
}
```

Using Queues

- Race condition:
  1. Thread adds itself to queue, releases lock
  2. Context switch just before call to `park()`
  3. Owner thread is done, removes new item from queue
  4. Calls `unpark()` for new thread: not yet parked!

Using Queues

- Also in Solaris: `setpark()`
  - Thread about to call `park()`
  - If interrupted and `unpark()` is called for it:
  - Subsequent `park()` returns immediately

```
void lock(lock_t *m) {
    ...
    } else {
        setpark(); // <- new code
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}
```

Using Queues: Different OS

- Support details vary between OS
- Linux: **futex**
  - Similar to Solaris
  - `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:

```c
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);
}
```

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**