Locks (ch 28) ptr. 1

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

Moshe Sulamy

Tel-Aviv Academic College

Locks

- **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:

```
1 lock_t mutex; // lock variable
2 ...
3 lock(&mutex);
4 balance = balance + 1;
5 unlock(&mutex);
```

Pthread Locks

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

```
1 pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2 ...
3 pthread_mutex_lock(&lock); // may fail!
4 balance = balance + 1;
5 pthread_mutex_unlock(&lock);
```

Pthread Locks

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

```
1 pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2 ...
3 pthread_mutex_lock(&lock); // may fail!
4 balance = balance + 1;
5 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

```
1  typedef struct __lock_t { int flag; } lock_t;
2
3  void init(lock_t* mutex) {
4      // 0: available, 1: locked
5      mutex->flag = 0;
6  }
7  void lock(lock_t* mutex) {
8      while (mutex->flag == 1)
9          ; // spin-wait
10     mutex->flag = 1;
11 }
12 void unlock(lock_t* mutex) {
13     mutex->flag = 0;
14 }
```

Just Using Loads/Stores

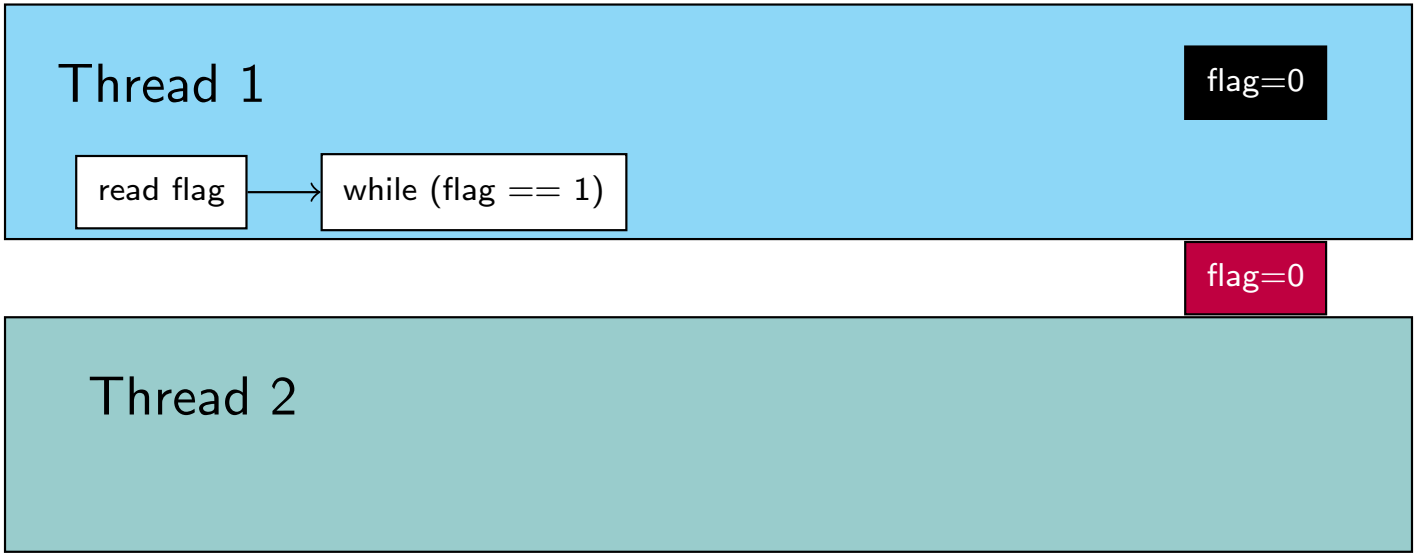
- No mutual exclusion

Just Using Loads/Stores

- No mutual exclusion

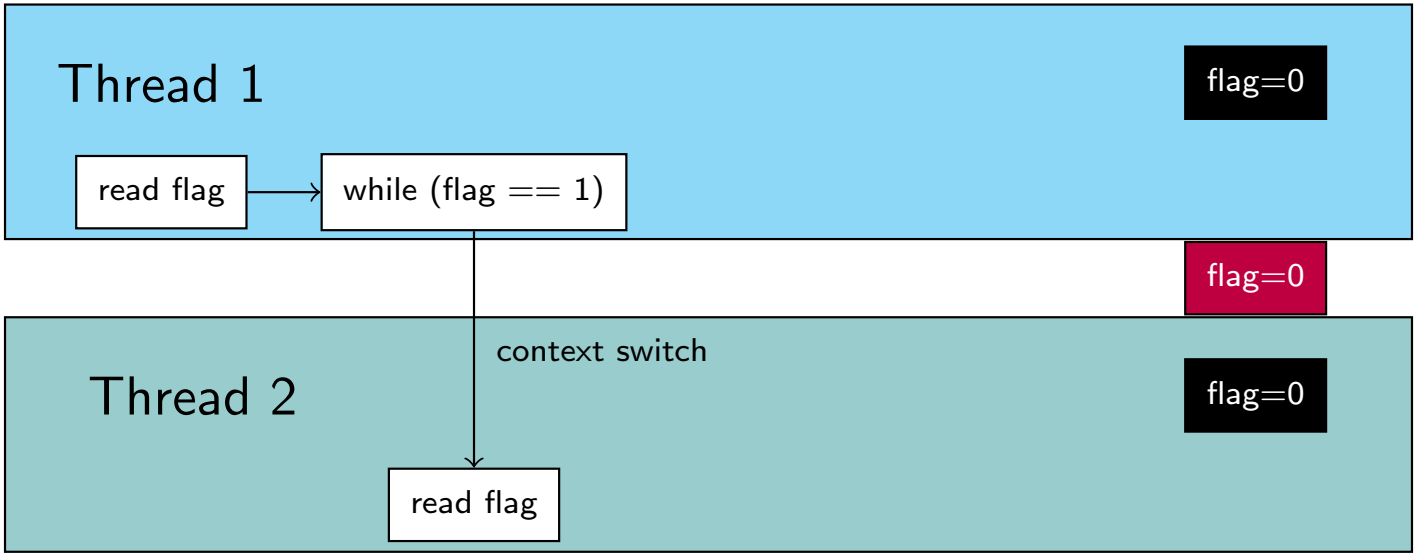
Just Using Loads/Stores

- No mutual exclusion



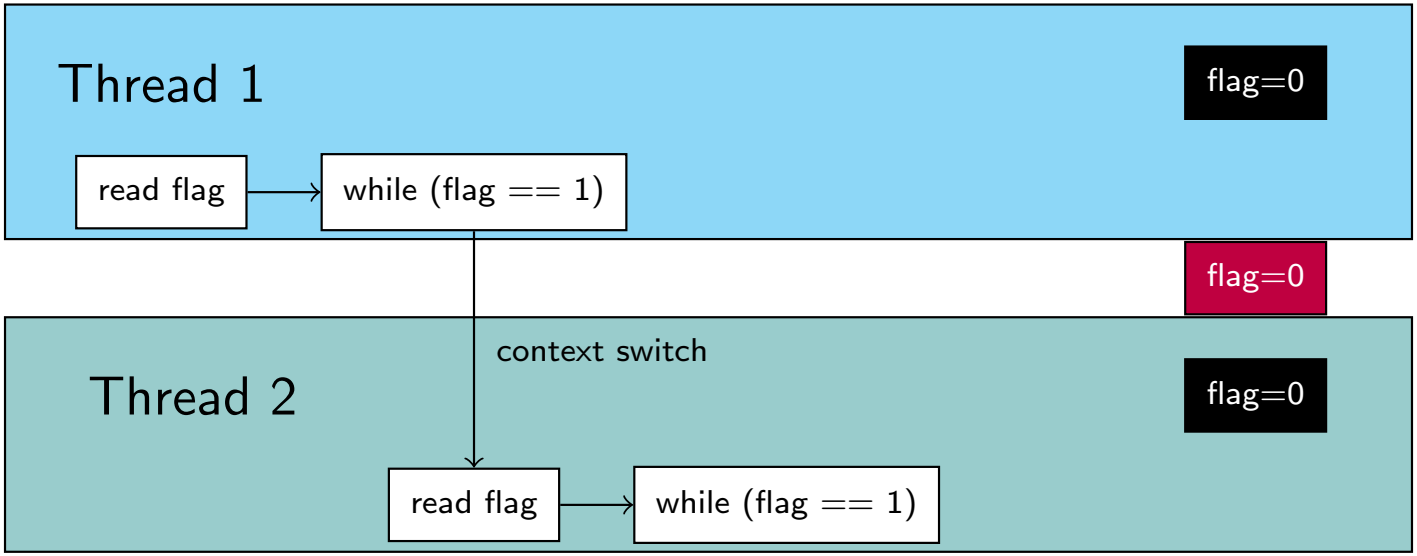
Just Using Loads/Stores

- No mutual exclusion



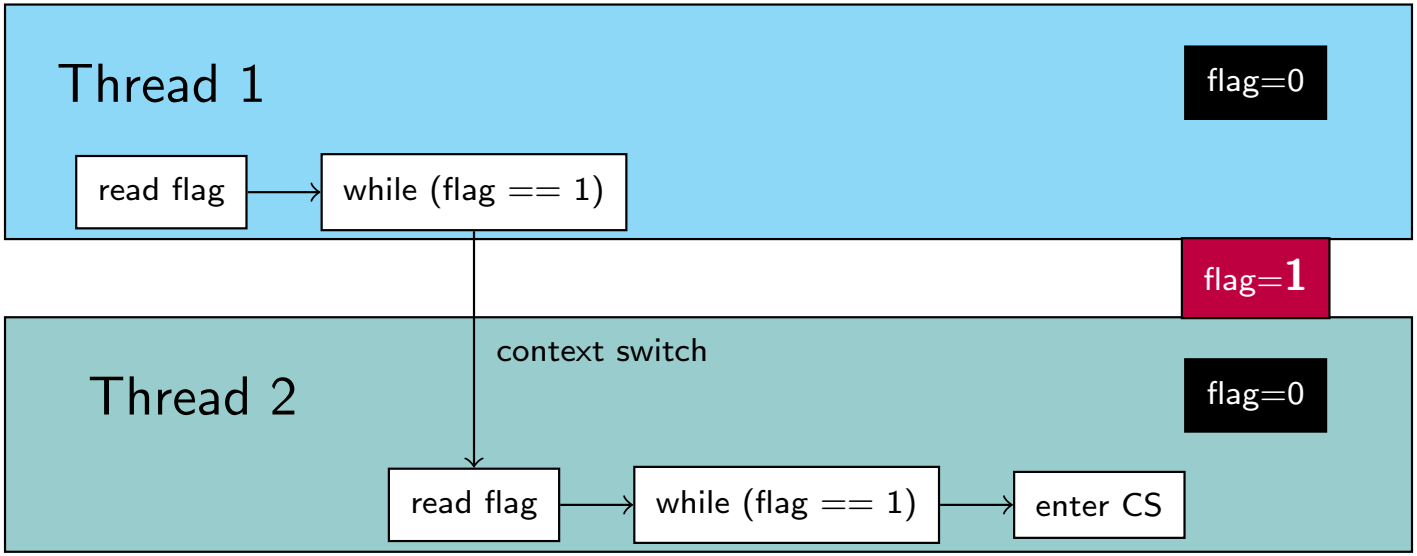
Just Using Loads/Stores

- No mutual exclusion



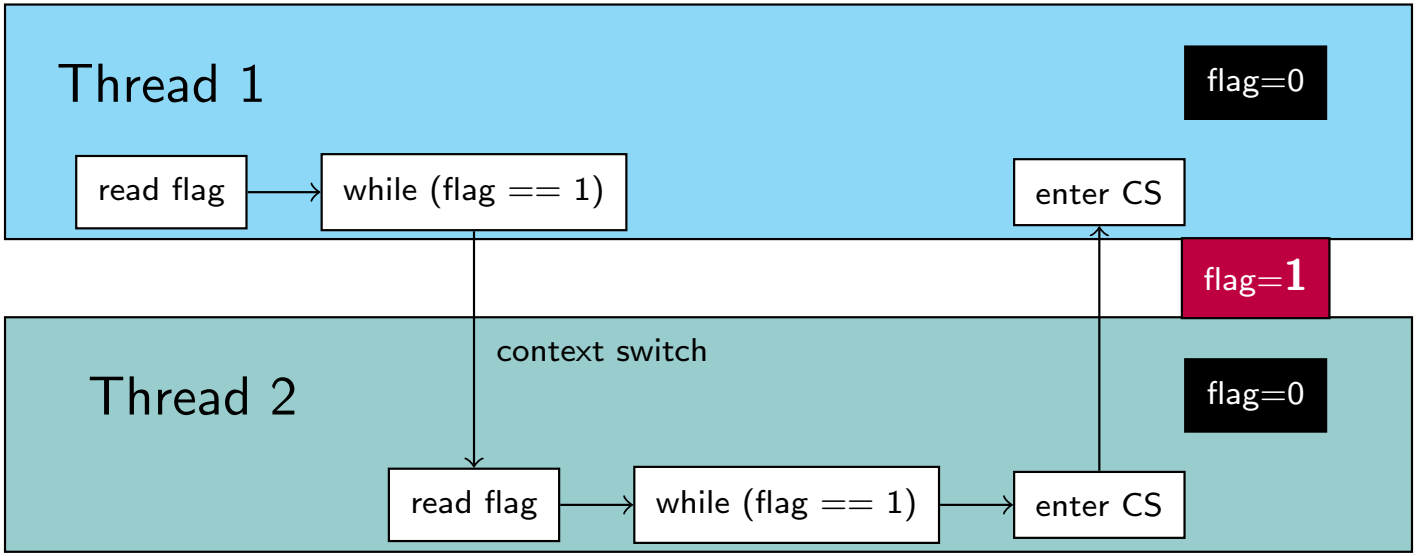
Just Using Loads/Stores

- No mutual exclusion



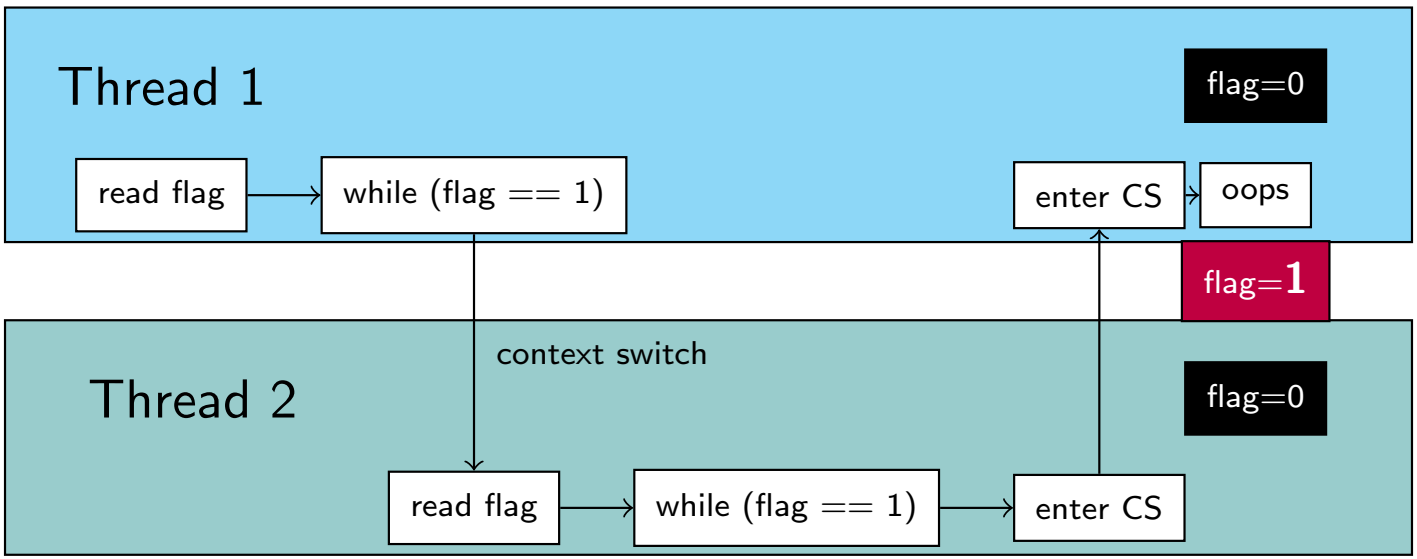
Just Using Loads/Stores

- 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:

```
1 int TestAndSet(int* old_ptr, int new) {
2     int old = *old_ptr;
3     *old_ptr = new;
4     return old;
5 }
```

New Spin Lock

```
1  typedef struct __lock_t { int flag; } lock_t;
2
3  void init(lock_t* mutex) {
4      // 0: available, 1: locked
5      mutex->flag = 0;
6  }
7  void lock(lock_t* mutex) {
8      while (TestAndSet(&mutex->flag, 1))
9          ; // spin-wait
10     mutex->flag = 1;
11 }
12 void unlock(lock_t* mutex) {
13     mutex->flag = 0;
14 }
```

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: reasonably well

Compare-And-Swap

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

```
1 int CompareAndSwap(int* ptr, int expected, int new) {
2     int original = *ptr;
3     if (original == expected)
4         *ptr = new;
5     return original;
6 }
```

Compare-And-Swap

- Spin-lock with CAS:

```
1 void lock(lock_t* lock) {
2     while (CompareAndSwap(&mutex->flag, 0, 1) == 1)
3         ; // spin
4 }
```

- Fairness? performance?

Compare-And-Swap

- Spin-lock with CAS:

```
1 void lock(lock_t* lock) {
2     while (CompareAndSwap(&mutex->flag, 0, 1) == 1)
3         ; // spin
4 }
```

- Fairness? performance?
  - Pretty much the same