Implementing Locks

Nima Honarmand

(Based on slides by Prof. Andrea Arpaci-Dusseau)

Lock Implementation Goals

- We evaluate lock implementations along following lines
- **Correctness**
    - **Mutual exclusion**: only one thread in critical section at a time
    - **Progress** (deadlock-free): if several simultaneous requests, must allow one to proceed
    - **Bounded wait** (starvation-free): must eventually allow each waiting thread to enter
- **Fairness**: each thread waits for same amount of time
    - Also, threads acquire locks in the same order as requested
- **Performance**: CPU time is used efficiently

Building Locks

- Locks are variables in shared memory
  - Two main operations: `acquire()` and `release()`
  - Also called `lock()` and `unlock()`
- To check if locked, read variable and check value
- To acquire, write “locked” value to variable
  - Should only do this if already unlocked
  - If already locked, keep reading value until unlock observed
- To release, write “unlocked” value to variable

First Implementation Attempt

- Using normal load/store instructions

```
Boolean lock = false; // shared variable

Void acquire(Boolean *lock) {
    while (*lock) /* wait */ ;
    *lock = true;
}

Void release(Boolean *lock) {
    *lock = false;
}
```

} Final check of `while` condition & write to `lock` should happen atomically

- This does not work. Why?
- Checking and writing of the lock value in acquire() need to happen atomically.

Solution: Use Atomic RMW Instructions

- **Atomic Instructions** guarantee atomicity
  - Perform **Read, Modify, and Write** atomically (**RMW**)
  - Many flavors in the real world
    - **Test and Set**
    - **Fetch and Add**
    - **Compare and Swap (CAS)**
    - **Load Linked / Store Conditional**

Example: Test-and-Set

Semantic:

```
// return what was pointed to by addr
// at the same time, store newval into addr atomically
int TAS(int *addr, int newval) {
    int old = *addr;
    *addr = newval;
    return old;
}
```

Implementation in x86:

```
int TAS(volatile int *addr, int newval) {
    int result = newval;
    asm volatile("lock; xchg %0, %1"
                 : "+m" (*addr), "=r" (result)
                 : "1" (newval)
                 : "cc");
    return result;
}
```

Lock Implementation with TAS

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

void init(lock_t *lock) {
    lock->flag = ??;
}

void acquire(lock_t *lock) {
    while (????)
        ; // spin-wait (do nothing)
}

void release(lock_t *lock) {
    lock->flag = ??;
}
```

Lock Implementation with TAS

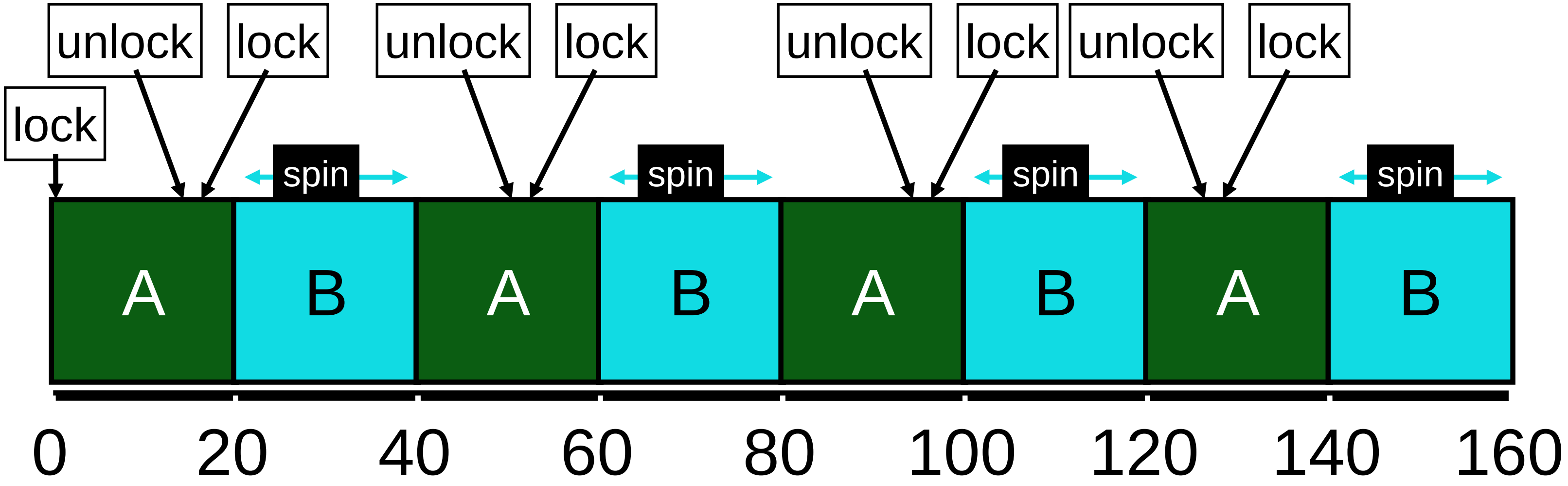
```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    lock->flag = 0;
}

void acquire(lock_t *lock) {
    while (TAS(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void release(lock_t *lock) {
    lock->flag = 0;
}
```

Evaluating Our Spinlock

- Lock implementation goals
    1) **Mutual exclusion**: only one thread in critical section at a time
    2) **Progress** (deadlock-free): if several simultaneous requests, must allow one to proceed
    3) **Bounded wait**: must eventually allow each waiting thread to enter
    4) **Fairness**: threads acquire lock in the order of requesting
    5) **Performance**: CPU time is used efficiently

- Which ones are NOT satisfied by our lock impl?
    - 3, 4, 5

Our Spinlock is Unfair



Scheduler is independent of locks/unlocks

Fairness and Bounded Wait

- Use **Ticket Locks**
- Idea: reserve each thread's turn to use a lock.
  - Each thread spins until their turn.
- Use new atomic primitive: **fetch-and-add**
- Acquire: Grab ticket using fetch-and-add
- Spin while not thread's ticket != turn
- Release: Advance to next turn

Semantics:

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

Implementation:

```
// Let's use GCC's built-in
// atomic functions this time around
__sync_fetch_and_add(ptr, 1)
```

Ticket Lock Example

Initially, turn = ticket = 0

A lock(): gets ticket 0, spins until turn == 0

→ A runs

B lock(): gets ticket 1, spins until turn == 1

C lock(): gets ticket 2, spins until turn == 2

A unlock(): turn++ (turn = 1)

→ B runs

A lock(): gets ticket 3, spins until turn == 3

B unlock(): turn++ (turn = 2)

→ C runs

C unlock(): turn++ (turn = 3)

→ A runs

A unlock(): turn++ (turn = 4)

C lock(): gets ticket 4

→ C runs

Ticket Lock Implementation

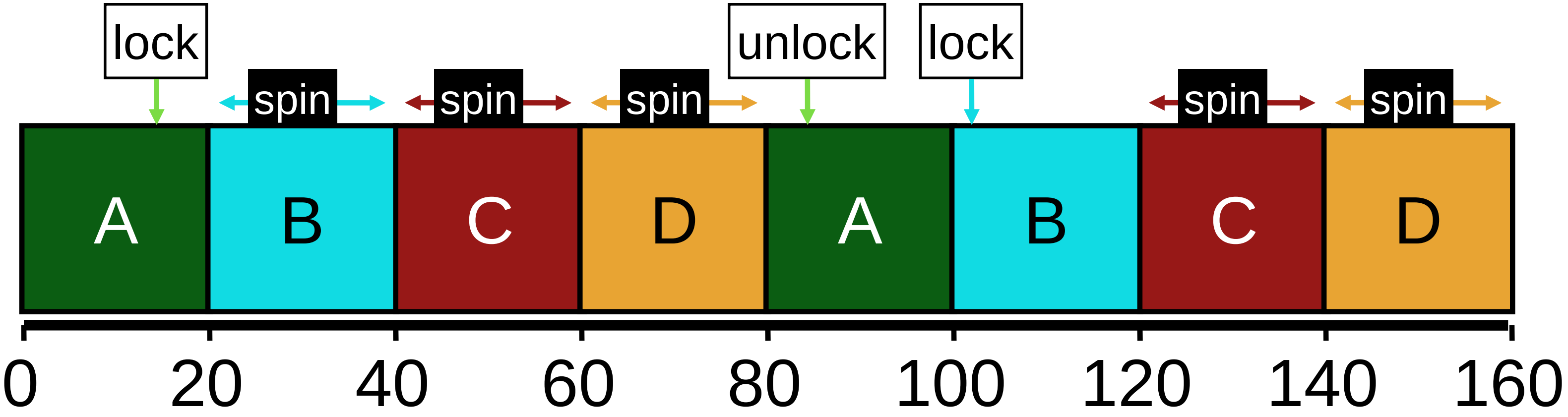
```
typedef struct {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

void acquire(lock_t *lock) {
    int myturn = FAA(&lock->ticket);
    while (lock->turn != myturn); // spin
}

void release(lock_t *lock) {
    lock->turn += 1;
}
```

Busy-Waiting (Spinning) Performance

- Good when…
  - many CPUs
  - locks held a short time
  - advantage: avoid context switch
- Awful when…
  - one CPU
  - locks held a long time
  - disadvantage: spinning is wasteful

CPU Scheduler Is Ignorant

- …of busy-waiting locks

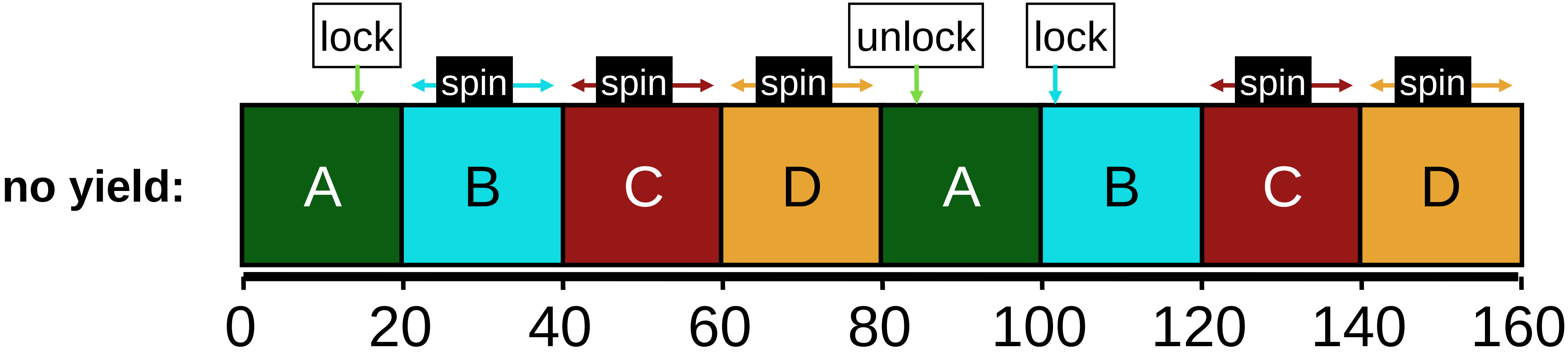


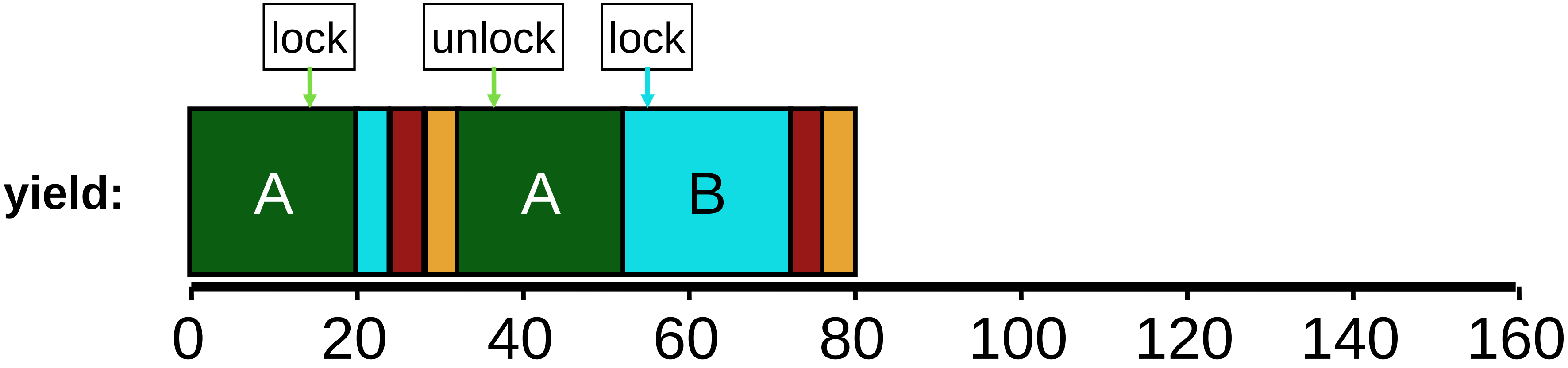
CPU scheduler may run **B** instead of **A** even though **B** is waiting for **A**

Ticket Lock with `yield()`

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

…

void acquire(lock_t *lock) {
    int myturn = FAA(&lock->ticket);
    while (lock->turn != myturn)
        yield();
}

void release(lock_t *lock) {
    lock->turn += 1;
}
```

Yielding instead of Spinning





Evaluating Ticket Lock

- Lock implementation goals
  1) **Mutual exclusion**: only one thread in critical section at a time
  2) **Progress** (deadlock-free): if several simultaneous requests, must allow one to proceed
  3) **Bounded wait**: must eventually allow each waiting thread to enter
  4) **Fairness**: threads acquire lock in the order of requesting
  5) **Performance**: CPU time is used efficiently

- Which ones are NOT satisfied by our lock impl?
  - 5 (even with yielding, too much overhead)

Spinning Performance

- Wasted time
  - Without yield: *O*(*threads* × ***time_slice***)
  - With yield: *O*(*threads* × ***context_switch_time***)
- So even with yield, spinning is slow with high thread contention
- Next improvement: instead of spinning, block and put thread on a wait queue

Blocking Locks

- `acquire()` removes waiting threads from run queue using special system call
  - Let’s call it **`park()`** — removes current thread from run queue
- `release()` returns waiting threads to run queue using special system call
  - Let’s call it **`unpark(tid)`** — returns thread `tid` to run queue

- Scheduler runs any thread that is ready
  - No time wasted on waiting threads when lock is not available
- Good separation of concerns
  - Keep waiting threads on a wait queue instead of scheduler’s run queue

- Note: `park()` and `unpark()` are made-up syscalls — inspired by Solaris’ `lwp_park()` and `lwp_unpark()` system calls

Building a Blocking Lock

```
typedef struct {
    int lock;
    int guard;
    queue_t q;
} lock_t;
```

1) What is `guard` for?
2) Why okay to spin on `guard`?
3) In `release()`, why not set `lock=false` when unparking?
4) Is the code correct?
   - Hint: there is a race condition

```
void acquire(lock_t *l) {
   while (TAS(&l->guard, 1) == 1);

   if (l->lock) {
      queue_add(l->q, gettid());
      l->guard = 0;
      park();          // blocked
   } else {
      l->lock = 1;
      l->guard = 0;
   }
}

void release(lock_t *l) {
   while (TAS(&l->guard, 1) == 1);

   if (queue_empty(l->q))
      l->lock=false;
   else
      unpark(queue_remove(l->q));
   l->guard = false;
}
```

Race Condition

```
Thread 1      in acquire()        Thread 2      in release()

if (l->lock) {
    queue_add(l->q, gettid());
    l->guard = 0;
                                  while (TAS(&l->guard, 1) == 1);
                                  if (queue_empty(l->q))
                                      l->lock=false;
                                  else
                                      unpark(queue_remove(l->q));

    park();
```

- Problem: `guard` not held when calling `park()`
    - Thread 2 can call `unpark()` before Thread 1 calls `park()`

Solving Race Problem: Final Correct Lock

```
typedef struct {
    int lock;
    int guard;
    queue_t q;
} lock_t;
```

- `setpark()` informs the OS of my plan to `park()` myself

- If there is an `unpark()` between my `setpark()` and `park()`, `park()` will return immediately (no blocking)

```
void acquire(lock_t *l) {
   while (TAS(&l->guard, 1) == 1);
   if (l->lock) {
      queue_add(l->q, gettid());
      setpark();
      l->guard = 0;
      park();          // blocked
   } else {
      l->lock = 1;
      l->guard = 0;
   }
}

void release(lock_t *l) {
   while (TAS(&l->guard, 1) == 1);

   if (queue_empty(l->q))
      l->lock=false;
   else
      unpark(queue_remove(l->q));
   l->guard = false;
}
```

Different OS, Different Support

- `park`, `unpark`, and `setpark` inspired by Solaris
- Other OSes provide different mechanisms to support blocking synchronization
- E.g., Linux has a mechanism called ***futex***
  - With two basic operations: **wait** and **wakeup**
  - It keeps the queue in kernel
  - It renders `guard` and `setpark` unnecessary
- Read more about futex in OSTEP (brief) and in an optional reading (detailed)

Spinning vs. Blocking

- Each approach is better under different circumstances

- Uniprocessor
  - Waiting process is scheduled → Process holding lock can't be
  - Therefore, waiting process should always relinquish processor
  - Associate queue of waiters with each lock (as in previous implementation)

- Multiprocessor
  - Waiting process is scheduled → Process holding lock might be
  - Spin or block depends on how long before lock is released
  - Lock is going to be released quickly → Spin-wait
  - Lock released slowly → Block

Two-Phase Locking

- A hybrid approach that combines best of spinning and blocking
- Phase 1: spin for a short time, hoping the lock becomes available soon
- Phase 2: if lock not released after a short while, then block
- Question: how long to spin for?
  - There’s a nice theory (next slide) which is in practice hard to implement, so just spin for a few iterations

Two-Phase Locking Spin Time

- Say cost of context switch is ***C*** cycles and lock will become available after ***T*** cycles

- Algorithm: spin for ***C*** cycles before blocking

- We can show this is a <u>2-approximation</u> of the optimal solution

- Two cases:
  - ***T*** < ***C***: optimal would spin for ***T*** (cost = ***T***), so do we (cost = ***T***)
  - ***T*** ≥ ***C***: optimal would immediately block (cost = ***C***), we spin for ***C*** and then block (cost = ***C*** + ***C*** = 2***C***)
  - So, our cost is at most twice that of optimal algorithm

- Problems to implement this theory?
  1) Difficult to know ***C*** (it is non-deterministic)
  2) Needs a low-overhead high-resolution timing mechanism to know when ***C*** cycles have passed