INF113: Locks

Kirill Simonov

17.10.2025

Reminder

- We aim to design a set of general **synchronization primitives** based on limited hardware instructions
- Allow to safely "block" parts of code when in use by a thread
- In other words, when entering a critical section T1 "locks" it from T2
- When T1 is done, the section should be unlocked again

```c
void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    for (int i = 0; i < 1e7; i++) {
      counter = counter + 1;
    }
    printf("%s: end\n", (char *) arg);
    return NULL;
}
```

Locks: the interface

- A datatype that stores the locked/unlocked state + possibly additional data

```
lock_t mutex;
```

- A function that sets the lock into "locked", called **before** the critical section 🔒

```
lock(&mutex);
```

- A function that sets the lock into "unlocked", called **after** the critical section 🔓

```
unlock(&mutex);
```

altogether:

```
lock_t mutex; //global var
...
  lock(&mutex);
  counter = counter + 1;
  unlock(&mutex);
```

Lock function behavior

- `mutex` is initially "unlocked"
- T1 calls `lock`: sets `mutex` to "locked" and returns
- T2 calls `lock`: while `mutex` is "locked", T2 is stuck within `lock`
- T1 calls `unlock`: sets `mutex` to "unlocked"
- T2 sets `mutex` to "locked" within `lock` and exits `lock`

🔒
```
lock_t mutex; //global var
...
  lock(&mutex);
  counter = counter + 1;
  unlock(&mutex);
```

T2

T1

POSIX locks

- `pthread.h` provides implementation for a lock, called a **mutex**

```
#include <pthread.h>
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
...
  pthread_mutex_lock(&lock);
  counter = counter + 1;
  pthread_mutex_unlock(&lock);
```

- A mutex can be dynamically initialized and destroyed

```
  pthread_mutex_init(&lock, NULL);
  ...
  pthread_mutex_destroy(&lock);
```

Locking tips

- Locking scope trade-offs
  - Larger scope: less flexibility, more waiting
  - Smaller scope: less waiting, more context switching

```
pthread_mutex_lock(&lock);
for (int i = 0; i < 1e7; ++i) {
  counter = counter + 1;
}
pthread_mutex_unlock(&lock);
```

```
for (int i = 0; i < 1e7; ++i) {
  pthread_mutex_lock(&lock);
  counter = counter + 1;
  pthread_mutex_unlock(&lock);
}
```

Locking tips

- Locking scope trade-offs
  - Larger scope: less flexibility, more waiting
  - Smaller scope: less waiting, more context switching

- Use different locks for different shared resources

T1

```
pthread_mutex_lock(&lock1);
counter1 = counter1 + 1;
pthread_mutex_unlock(&lock1);
...
pthread_mutex_lock(&lock2);
counter2 = counter2 + 1;
pthread_mutex_unlock(&lock2);
```

T2

while T1 modifies counter1, T2 is safe to access counter2

Locking tips

- Locking scope trade-offs
  - Larger scope: less flexibility, more waiting
  - Smaller scope: less waiting, more context switching

- Use different locks for different shared resources

- Less time in critical sections, less locking

```c
int local_cnt = 0;
for (int i = 0; i < 1e7; ++i) {
  local_cnt = local_cnt + 1;
  if (local_cnt > 1000) {
    pthread_mutex_lock(&lock);
    counter += local_cnt;
    pthread_mutex_unlock(&lock);
    local_cnt = 0;
  }
}
```

Example: approximate/delayed counter

Building locks

- How to build an efficient lock?
  - Hardware needs?
  - OS needs?

Evaluating the lock design:

- **Mutual exclusion:** Does it fulfill the purpose?
- **Fairness:** Does every thread get a chance of taking the lock? Alternatively, could a thread **starve**, waiting for a lock forever?
- **Performance:** What is the extra overhead of locking/unlocking?

```
void lock() {
  if (T1) {
    return;
  } else {
    while (true) {};
  }
}
```

Example: only T1 gets through the lock

Naïve implementation

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

void init(lock_t *mutex) {
  // 0 -> lock is available, 1 -> held
  mutex->flag = 0;
}

void lock(lock_t *mutex) {
  while (mutex->flag == 1) // TEST the flag
    ;// spin-wait (do nothing)
  mutex->flag = 1; // now SET it!
}

void unlock(lock_t *mutex) {
  mutex->flag = 0;
}
```

- Let's just store the state in a 0/1 variable
- Locking sets the flag to 1
  Unlocking sets to 0
- If the flag is taken, the thread will wait in a while-loop
- See the problem?

CPU

CPU
T1: TEST
T2: TEST
T1: SET
T2: SET

Another thread can sneak in-between!

Test-and-set instruction

- It seems there should be a single atomic instruction that performs `test` and `set` at the same time:

```c
int test_and_set(int *old_ptr, int new) {
  int old = *old_ptr; // fetch old value at old_ptr
  *old_ptr = new; // store 'new' into old_ptr
  return old; // return the old value
}
```

but actually implemented in hardware

- Hardware support exists indeed: e.g., called `xchg` on x86

- We can now redesign the lock implementation with `test_and_set` in mind

Test-and-set lock

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

void init(lock_t *mutex) {
  // 0 -> lock is available, 1 -> held
  mutex->flag = 0;
}

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

void unlock(lock_t *mutex) {
  mutex->flag = 0;
}
```

- If `flag` is 0, instruction sets it to 1 and reports 0 —thread proceeds
- If `flag` is 1, instruction sets it to 1 and reports 1 —thread waits

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

Compare-and-swap

- Similar to test-and-set, but also compares old value with an expected value

```
int compare_and_swap(int *old_ptr, int expected, int new) {
  int old = *old_ptr;  // fetch old value at old_ptr
  if (old == expected) // update only if value matches
    *old_ptr = new;    // store ‘new‘ into old_ptr
  return old;          // return the old value
}
```

- Can be used in the lock

```
void lock(lock_t *mutex) {
  while (compare_and_swap(&mutex->flag, 0, 1) == 1)
    ;// spin-wait (do nothing)
}
```

Compare-and-swap

- Similar to test-and-set, but also compares old value with an expected value

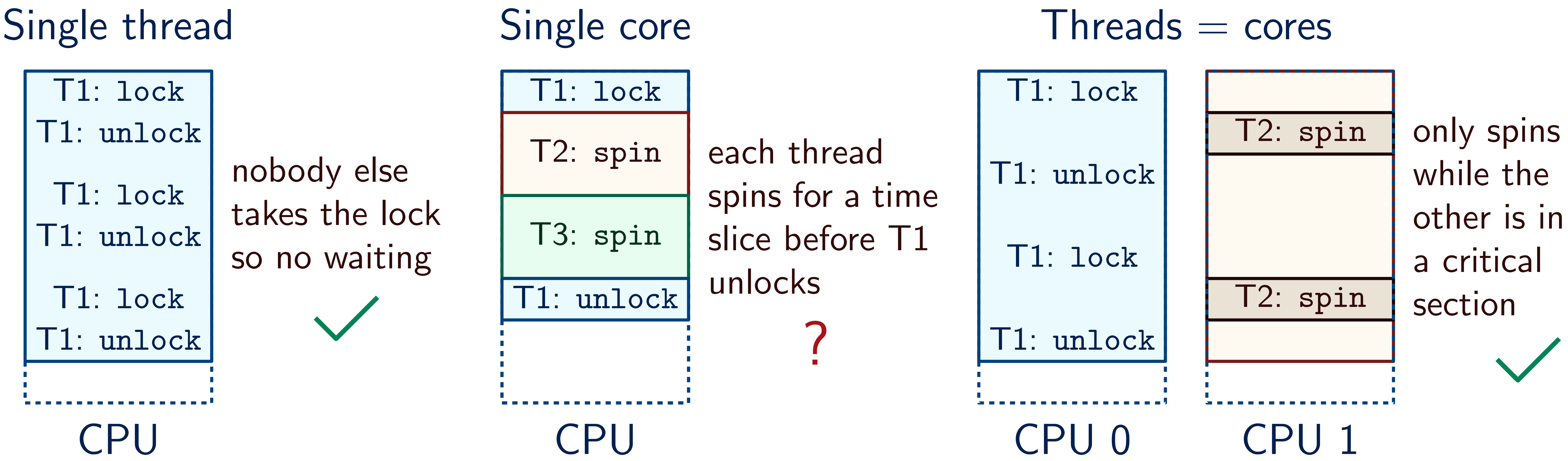
```
int compare_and_swap(int *old_ptr, int expected, int new) {
  int old = *old_ptr;  // fetch old value at old_ptr
  if (old == expected) // update only if value matches
    *old_ptr = new;    // store ‘new‘ into old_ptr
  return old;          // return the old value
}
```

- Can be used in the lock
- Also to implement other operations atomically, for example increment:

```
void inc(int *ptr) {
  while (true) {
    int old = *ptr;
    if (compare_and_swap(ptr, old, old + 1) == old) {
      break;
    }
  }
}
```

Evaluating spin locks

- **Mutual exclusion:** Done! ✓
- **Fairness:** A thread can be spinning indefinitely, if it's unlucky ?
- **Performance:** Depends on scenario, but spinning could be costly ?



Fetch-and-add

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

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

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

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

- Atomically increment and return the value of the counter:

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

- Ticket-based lock: once a thread gets a ticket, it is guaranteed to run!

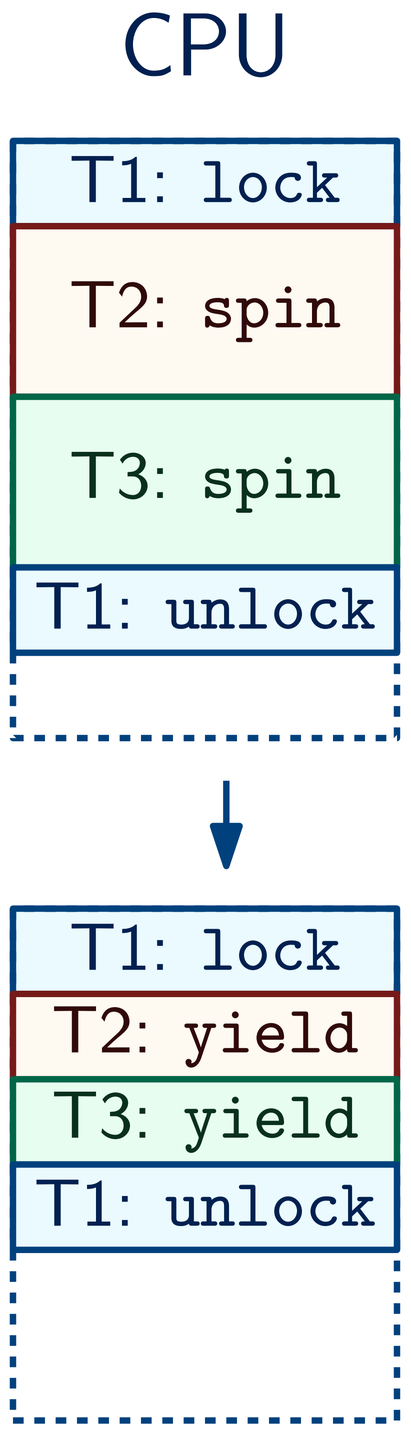
Fairness: ✓

Resolving spins

- If a thread without the lock is run, it wastes CPU for its whole timeslice
- Attempt 1: Let the thread voluntarily give up control instead of spinning

```
void lock(lock_t *mutex) {
  while (test_and_set(&mutex->flag, 1) == 1)
    yield(); // pass control back
}
```

- Still, lots of context switches before we get to unlock
- Threads can still starve



Queue-based lock

- We will put waiting threads to sleep in a queue
- OS support, Solaris example:
  - Syscall `park();` to yield control and be put to sleep,
  - Syscall `unpark(thread);` to wake `thread` and pass control
- `flag` keeps track of the lock
- `guard` is an internal lock to access the queue
- `queue` stores the waiting threads

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

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

Queue-based lock: lock

```
void lock(lock_t *m) {
  while (test_and_set(&m->guard, 1) == 1)
    ; //acquire guard lock by spinning
  if (m->flag == 0) {
    m->flag = 1; // lock is acquired
    m->guard = 0;
  } else {
    queue_add(m->q, gettid());
    m->guard = 0;
    park();
  }
}
```

- Then either get the flag and pass the main lock, or put yourself into the queue and yield

- First, try to pass the guard lock by spinning

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

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

Queue-based lock: unlock

```
void unlock(lock_t *m) {
  while (test_and_set(&m->guard, 1) == 1)
    ; //acquire guard lock by spinning
  if (queue_empty(m->q))
    m->flag = 0; // let go of lock; no
   one wants it
  else
    unpark(queue_remove(m->q)); // hold
   lock (for next thread!)
  m->guard = 0;
}
```

- First same, try to pass the guard lock by spinning

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

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

- Then either clear the flag, or put the next thread in the queue to work

Queue-based lock: benefits

- Threads only spin to access the internal queue
  - Only the small and contolled queue operation is a critical section for that, not arbitrary user code
- **Fairness:** Once a thread is put in the queue, it will clear the lock eventually ✓
- **Performance:** Less guessing by the scheduler ✓
  - When a thread is blocked, it yields and never randomly runs again
  - When a thread is done, it wakes the next waiting one

Summary

- A simple atomic instruction like test-and-set can be used to build safe locks
- Performance of a locking mechanism may depend a lot on the scenario: no. of threads, cores, time spent in critical sections, . . .
- To avoid wasting too much time on spinning, a good lock design introduces some order into thread scheduling—e.g., with the queue
  - **futex**, used in Linux, follows a similar queue-based approach
- Take a look at Chapter 28 for more details and a simulator
- **Next week:** more sync primitives, condition variables and semaphores