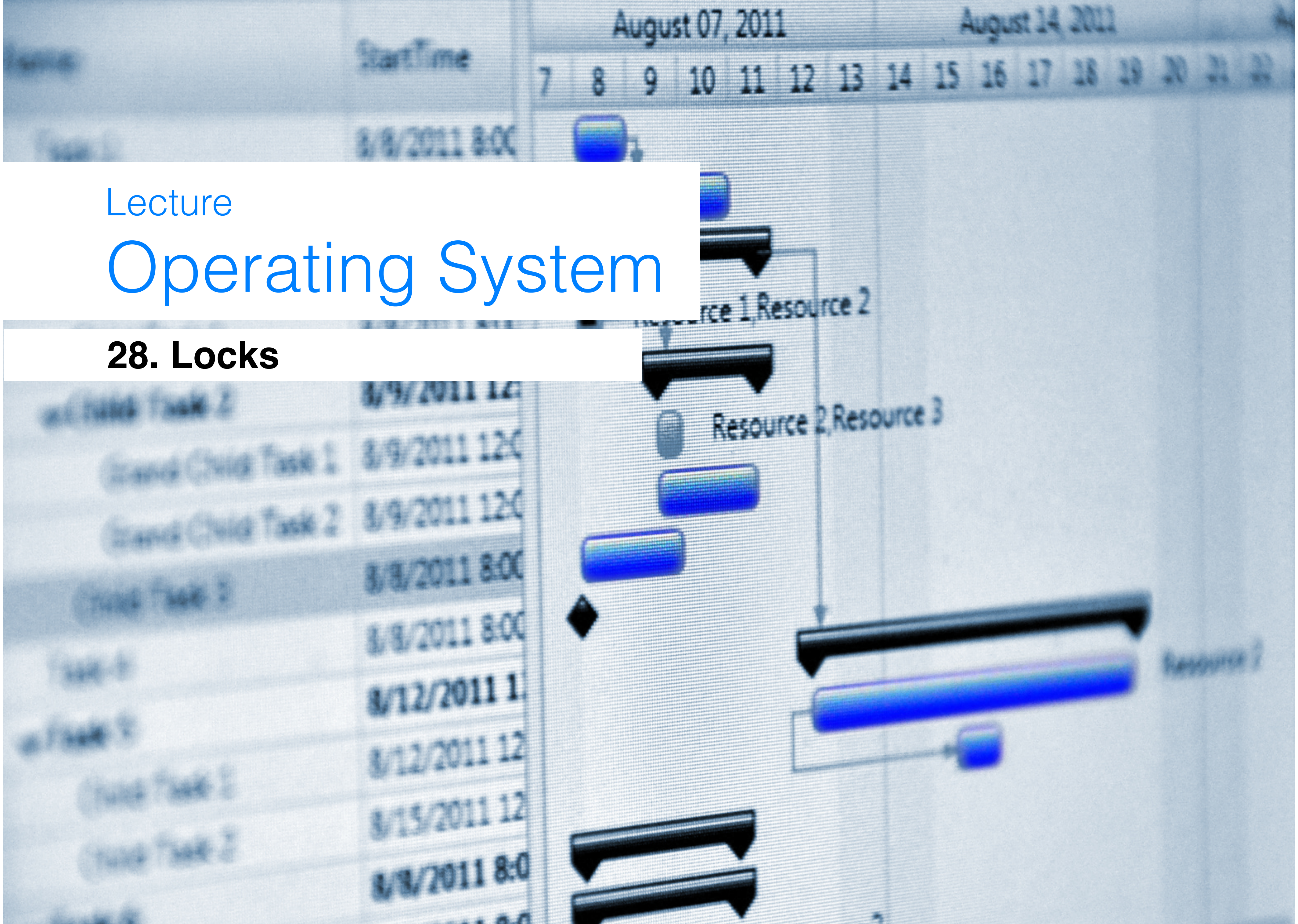


Lecture

Operating System

28. Locks

Professor Dr. Michael Mächtel

28. Locks

1. **Criteria**
2. **Solutions**
3. **Locks with Hardware Support**
4. **Spin Alternatives**

28. Locks

1. **Criteria**
2. Solutions
3. Locks with Hardware Support
4. Spin Alternatives

Locks: The Basic Idea

- Ensure that any critical section executes as if it were a single atomic instruction.
- An example: the canonical update of a shared variable
- Add some code around the critical section

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

```
lock_t mutex; // some globally-allocated lock 'mutex'
…
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Locks: The Basic Idea (Cont.)

- Lock variable holds **the state** of the lock.
  - **available** (or **unlocked** or **free**)
    - No thread holds the lock.
- **acquired** (or **locked** or **held**)
  - Exactly one thread holds the lock and presumably is in a critical section.

The semantics of the lock()

- `lock()`
  - Try to acquire the lock.
  - If no other thread holds the lock, the thread will acquire the lock.
  - Enter the critical section.
    - This thread is said to be the owner of the lock.
  - Other threads are prevented from entering the critical section while the first thread that holds the lock is in there.

Pthread Locks - mutex

- The name that the POSIX library uses for a **lock**.
  - Used to provide **mutual exclusion** between threads.
  - We may be using different locks to protect different variables
    - Increase **concurrency** (a more **fine-grained** approach).

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock); // wrapper for pthread_mutex_lock()
balance = balance + 1;
Pthread_mutex_unlock(&lock);
```

Building a Lock

- Efficient locks provided mutual exclusion at low cost.
- Building a lock need some help from the hardware and the OS.
- The Crux:
  - How can we build an efficient lock with low cost?
  - What hardware support is needed?
  - What OS support?

Evaluating locks – Basic criteria

- **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 (starvation-free)
    - Must eventually allow each waiting thread to enter
- **Fairness**
  - Each thread waits for same amount of time
- **Performance**
  - CPU is not used unnecessarily (e.g., spinning)

28. Locks

1. Criteria
2. **Solutions**
3. Locks with Hardware Support
4. Spin Alternatives

Controlling Interrupts

- **Disable Interrupts** for critical sections
  - One of the earliest solutions used to provide mutual exclusion
  - Invented for single-processor systems.
- **Problem**:
  - Require too much trust in applications
    - Greedy (or malicious) program could monopolize the processor.
  - Do **not work** on **multiprocessors**
  - Code that masks or unmasks interrupts be executed slowly by modern CPUs

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

Why hardware support needed?

- First attempt: Using a flag denoting whether the lock is held or not.
  - The code below has problems.

```
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;
 }
```

Why hardware support needed? (Cont.)

- Code has problems
  - Problem 1: **No Mutual Exclusion** (assume flag=0 to begin)
  - Problem 2: Spin-waiting wastes time waiting for another thread.
- So, we need an **atomic instruction** supported by Hardware!
  - **test-and-set instruction**, also known as atomic exchange

Thread1	Thread2
call `lock()`	
`while (flag == 1)`	
interrupt: switch to Thread 2	
	call `lock()`
	`while (flag == 1)`
	`flag = 1;`
	interrupt: switch to Thread 1
`flag = 1;` // set flag to 1 (too!)	

Why not in software?

```
/* Thread 0 /*                  /* Thread 1 */

.                               .
.                               .
while (turn != 0)               while (turn != 1)
    /*do nothing*/ ;                /*do nothing*/ ;
/* critical section*/;          /* critical section*/;
turn = 1;                       turn = 0;
.                               .
```

- thread enters critical section if **turn = thread-number**
- if **turn != thread-Nummer** -> Spin! (active waiting)
- when critical section is finished, allow other thread to enter
- Problem?
  - no race cond., but threads enter critical section **alternately**

Why not in software? (Cont.)

```
/* Thread 0 */

.
.
flag[0] = true;
while (flag[1])
    /*do nothing */ ;
/* critical section*/;
flag[0] = false;
.
```

context switch

```
/* Thread 1 */

.
.
flag[1] = true;
while (flag[0])
    /*do nothing */ ;
/* critical section*/;
flag[1] = false;
.
```

- Each thread has it's own flag
- Before reading other flag, set own to 'locked' (=true)
- Problem?
    - no race condition!
    - but **deadlock!** Thread 1 and Thread 2 "while" for ever ....

Why not in software? (Cont.)

Peterson-Algorithmus (1981)

```
int turn = 0; // shared
Boolean flag[2] = {false, false};

Void acquire() {
    flag[tid] = true;
    turn = 1-tid;
    while (flag[1-tid] && turn == 1-tid) /* wait */ ;
}

Void release() {
    flag[tid] = false;
}
```

- Assume two threads (tid = 0, 1)
  - Critical section is protected by flag (see slide before)
  - deadlock is prevented by turn

Why not in software? (Cont.)

- Evaluating Peterson's Algorithm:
  - Mutual exclusion: Enter critical section if and only if
    - Other thread does not want to enter
    - Other thread wants to enter, but your turn
  - Progress: Both threads cannot wait forever at while() loop
    - Completes if other thread does not want to enter
    - Other thread (matching turn) will eventually finish
  - Bounded waiting
    - Each thread waits at most one critical section
- **Problem**: doesn’t work on modern hardware
  - cache-consistency issues

28. Locks

1. Criteria
2. Solutions
3. **Locks with Hardware Support**
4. Spin Alternatives

Test And Set (Atomic Exchange)

- An instruction to support the creation of simple locks
    - **return** (testing) old value pointed to by the `ptr`.
    - *Simultaneously* **update** (setting) said value to `new`.
    - This sequence of operations is **performed atomically**.

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

A Simple Spin Lock using test-and-set

- **Note**: To work correctly on a **single processor**, it requires a **preemptive scheduler**.

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

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

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ;   // spin-wait
}

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

Compare-And-Swap

Compare-and-Swap hardware atomic instruction (C-style)

- Test whether the value at the address(`ptr`) is equal to `expected`.
  - *If so*, **update** the memory location pointed to by ptr with the new value.
  - *In either case*, **return** the actual value at that memory location.

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

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

Evaluating Spin Locks

- **Correctness**: yes
  - The spin lock only allows a single thread to entry the critical section
- **Fairness**: no
  - Spin locks don’t provide any fairness guarantees.
  - Indeed, a thread spinning may spin forever under contention
- **Performance**:
  - In the single CPU, performance overheads can be quire painful.
  - If the number of threads roughly equals the number of CPUs, spin locks work reasonably well.

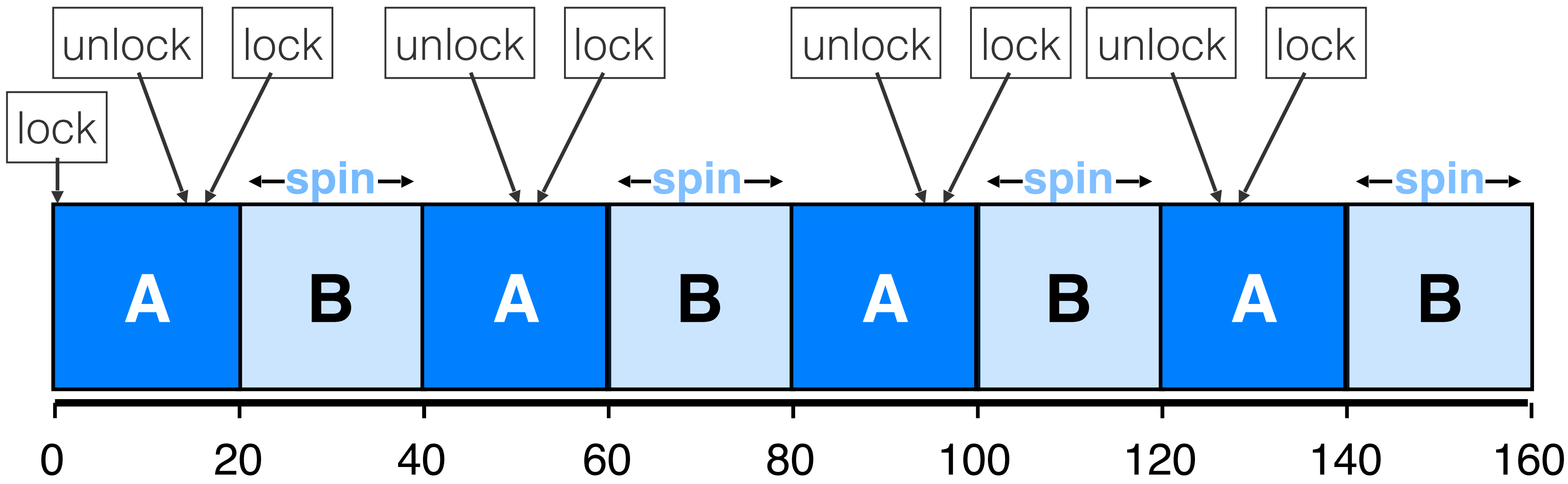
Fetch-And-Add

Fetch-And-Add Hardware atomic instruction (C-style)

- Atomically increment a value while returning the old value at a particular address.

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

Basic Spinlocks are Unfair



Ticket Lock

- Ticket lock can be built with fetch-and add.
  - Ensure progress for all threads. ⇒ **fairness**

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

void lock_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) {
    FetchAndAdd(&lock->turn);
}
```

- Idea: reserve each thread's turn to use a lock.
- Each thread spins until their turn.
- Use fetch-and-add:

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

Ticket Lock Example

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

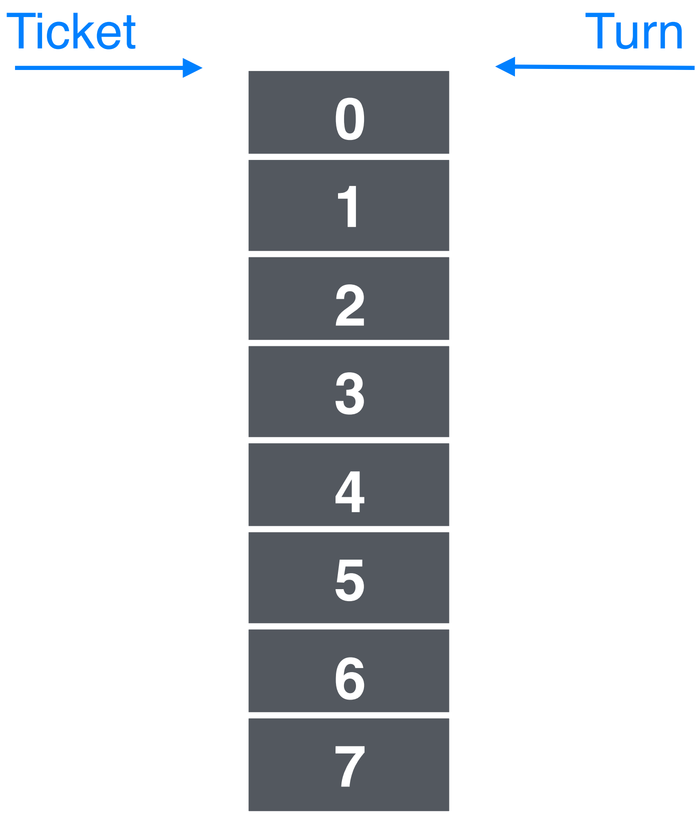
C runs

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

A runs

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

C lock(): gets ticket 4, runs



Spinlock Performance

- Hardware-based spin locks are **simple** and they work.
- Fast when…
  - many CPUs
  - locks held a short time
  - advantage: avoid context switch
- Slow when…
  - one CPU
  - locks held a long time
  - disadvantage: spinning is **inefficient**
    - Any time a thread gets caught spinning, it **wastes an entire time slice** doing nothing but checking a value.

28. Locks

1. Criteria
2. Solutions
3. Locks with Hardware Support
4. **Spin Alternatives**

A Simple Approach: Just Yield

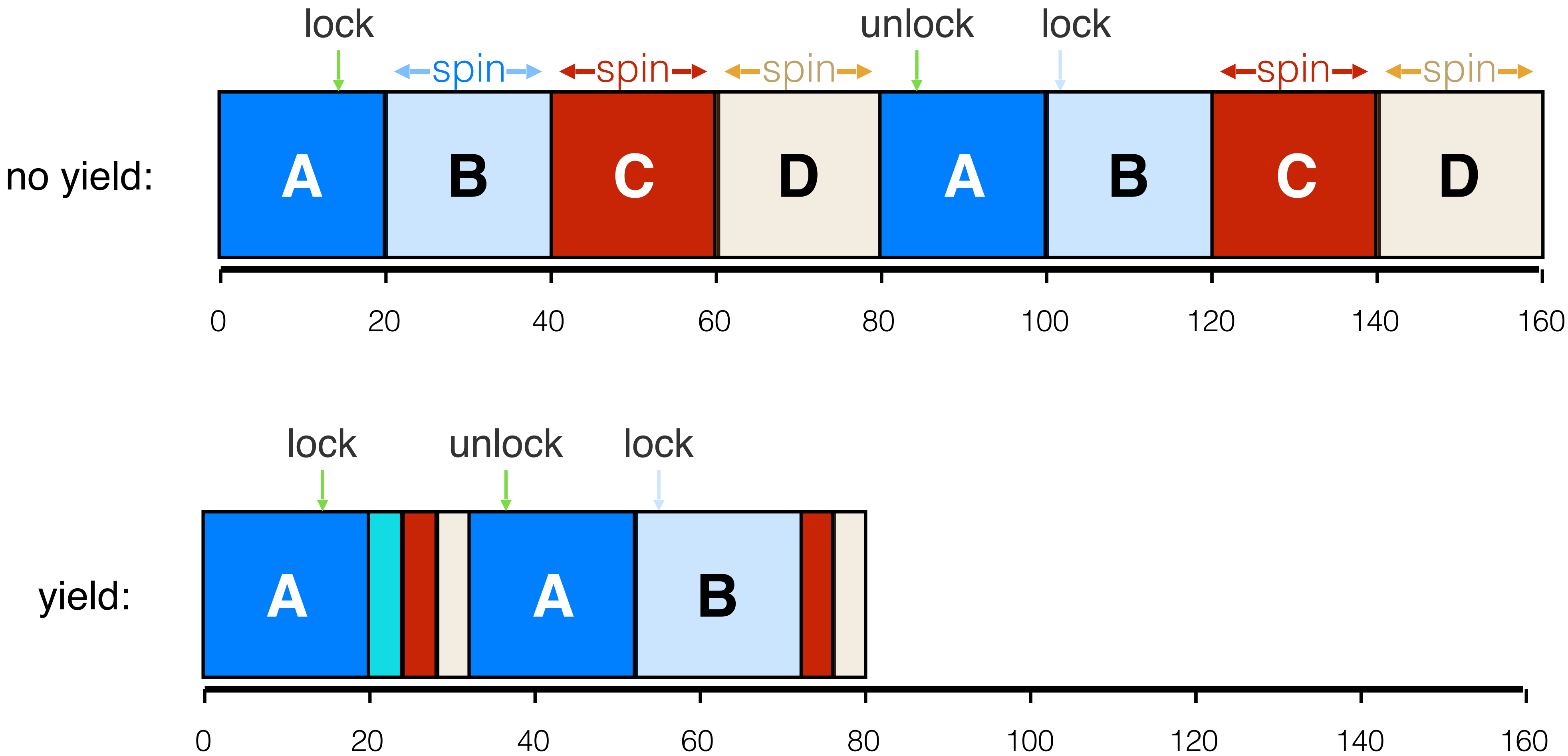
- When you are going to spin, **give up the CPU** to another thread.
  - OS system call moves the caller from the *running state* to the *ready state*.
  - The cost of a **context switch** can be substantial and the **starvation** problem still exists.

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

Yield Instead of Spin



Using Queues

Sleeping Instead of Spinning

- **Queue** to keep track of which threads are **waiting** to enter the lock.
- `park()`
  - Put a calling thread to sleep
- `unpark(threadID)`
  - Wake a particular thread as designated by `threadID`.

Using Queues

Lock With Queues, Test-and-set, Yield, And Wakeup

```
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);
 }

 void lock(lock_t *m) {
     while (TestAndSet(&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();
     }
 }
```

park() uses yield()

Using Queues

Lock With Queues, Test-and-set, Yield, And Wakeup

```
void unlock(lock_t *m) {
    while (TestAndSet(&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;
}
```

Race Condition

Wakeup/Waiting race

- In case of releasing the lock (thread 2) just before the call to `park()` (thread 1)
  - Thread 1 would sleep forever (potentially).

Thread 1 in `lock()`

```
if (m->lock) {
    queue_add(m->q, tid);
    m->guard = 0;




    park();     // block
```

Thread 2 in `unlock()`

```
while (TAS(&m->guard, 1) == 1);
if (queue_empty(m->q)) // false !!
else unpark(queue_remove(m->q));
m->guard = 0;
```

Race Condition: How to solve?

Wakeup/Waiting race:

- New system call: `setpark()`
    - By calling this routine, a thread can indicate it is about to park.
    - If by interruption another thread calls `unpark()` before `park()` is actually called, the subsequent `park()` returns immediately instead of sleeping.

Thread 1 in `lock()`

```
if (m->lock) {
     queue_add(m->q, tid);
     setpark(); // new code
     m->guard = 0;






    park();    // block
```

Thread 2 in `unlock()`

```
while (TAS(&m->guard, 1) == 1);
if (queue_empty(m->q)) // false !!
else unpark(queue_remove(m->q));
m->guard = 0;
```

Two-Phase Locks

- A two-phase lock realizes that **spinning can be useful** if the lock is about to be released.
  - **First phase**
    - The lock spins for a while, hoping that it can acquire the lock.
    - If the lock is not acquired during the first spin phase, a second phase is entered,
  - **Second phase**
    - The caller is put to sleep.
    - The caller is only woken up when the lock becomes free later.

Thanks

Questions?