Concurrency

- Exam 1
- 26 - Concurrency
- 27 - Overview, and POSIX threads (pthreads)
- 28 - Locks
- 29 - Concurrent Data Structures
- 30 Condition Variables
- 31 - Semaphores
- 32 - Common Problems
- 33 - Event-Based Concurrency

Concurrency

- Exam 1
- 26 - Concurrency
- 27 - Overview, and POSIX threads (`pthreads`)
- 28 - Lock Details
- 29 - Concurrent Data Structures
- 30 Condition Variable Details
- 31 - Semaphores
- 32 - Common Problems
- 33 - Event-Based Concurrency

Locks *the basic idea*

- Ensure that any critical sections executes atomically
  - Canonical update of a shared variable:

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

  - Use locks:

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


Locks *the basic idea*

- Lock variables hold the *lock state*:
  - *unlocked* (or available, or free)
    - no thread holds the lock
  - *locked* (or acquired or held)
    - exactly one thread holds the lock
    - presumably in the critical section

Locks *semantics*

- `lock()`
  - acquired if no other thread holds it
  - enter *critical section*
    - calling thread is now the lock's *owner*
  - other threads prevented from entering the critical section
    - assuming proper lock discipline

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock);
balance = balance + 1;
Pthread_mutex_unlock(&lock);
```

- using many mutexes increases concurrency
- efficient locks require help from *hardware* and the *OS*

Goal and Metrics *for locks*

- Mutual exclusion:
  - does it work?
  - *correctness*

- Fairness:
  - can threads *starve*?
  - do they get a fair share?

- Performant:
  - how much overhead

Lock implementation *controlling interrupts*

- Half-century-old approach:
  - *disable interrupts* for critical sections
  - even for single-processors:

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

  - Problems:
    - requires *trust* in applications
      - greedy program might not enable interrupts until done
    - not sufficient for multiprocessors
    - expensive
    - is the above implementation correct and complete?

Lock implementation *do we really need hardware?*

- First attempt:
  - use a *flag* to show if lock held:

```
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;
}
```

  - This code has problems:
    - correctness
    - efficiency

Lock implementation *do we really need hardware?*

- Correctness: (no mutual exclusion)

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!)	

- Performance:
  - spin-waiting
    - not doing useful work
    - might be actively *preventing* the lock from being released

Peterson's algorithm *do we really need hardware?*

```
int flag[2];
int turn;

void init() {
  // indicate you intend to hold the lock w/ 'flag'
  flag[0] = flag[1] = 0;
  // whose turn is it? (thread 0 or 1)
  turn = 0;
}

void lock() {
  // 'self' is the thread ID of caller
  flag[self] = 1;
  // make it other thread's turn
  turn = 1 - self;
  while ((flag[1-self] == 1) && (turn == 1 - self))
    ; // spin-wait while it's not your turn
}

void unlock() {
  // simply undo your intent
  flag[self] = 0;
}
```

doesn't work w/ relaxed consistency models

already had hardware support when this written

still important for under-standing synchronization

Test-and-Set *hardware support*

- single atomic hardware instruction
- pseudocode:

```
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
}
```

- *returns* old value to be tested
- *simultaneously* updates value to new

Test-and-Set *making a spin lock*

```
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;
}
```

- requires a *preemptive scheduler*, even for single processor

Test-and-Set *Goal and Metrics*

- Mutual exclusion: yes
  - does it work?
  - *correctness*
- Fairness: no
  - can threads *starve*?
  - do they get a fair share?
- Performant: not usually
  - on single CPU often quite bad
  - may be ok if:

    #threads about the same as *#processors*

Compare-and-Swap *hardware support*

- Test whether `*ptr == expected_value`
  - *if so*: update `*ptr` with `expected_value`
  - *always*: return actual value from prior to instruction
- pseudocode:

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

- Spin lock using compare-and-swap:

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

- vs test-and-set?
  - more powerful

Load-Linked Stores *hardware support*

- pseudocode:

```
int LoadLinked(int *ptr) {
    return *ptr;
}

int StoreConditional(int *ptr, int value) {
    if (*ptr not updated since the LoadLinked to this address) {
        *ptr = value;
        return 1; // success!
    } else {
        return 0; // failed to update
    }
}
```

- only succeeds if *no intervening store* to *same address*
  - success: `1` is returned, and update `*ptr` to `value`
  - failure: `0` is returned, no change to `*ptr`
- vs test-and-set?
  - more powerful
- can be efficient for hardware

Locks *so much spinning*

- Hardware-based spin locks are simple and correct
  - they can also be very inefficient….
- Address with OS support:
  - instead of spinning, just *yield*….

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

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

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

Using queues *sleeping instead of spinning*

- Use a queue to track threads waiting to enter a lock
  - `park()` : put calling thread to sleep
  - `unpark(threadID)` : wake specific thread

```
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();
    }
}
…
```


Using queues *sleeping instead of spinning*

```
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;
}
```

Using queues *sleeping instead of spinning*

- There is a race between waking up and waiting
  - Think of releasing a lock in T_A just before T_B calls `park()`
  - T_B could sleep forever…
- Solaris solves by adding a third system call: setpark()
  - indicates that a thread is *about to* park
  - if a thread is interrupted, and another thread calls `unpark()` before park actually happens, the `park()` returns immediately

```
queue_add(m->q, gettid());
setpark(); // new code
m->guard = 0;
park();
```


Concurrency

- Exam 1
- 26 - Concurrency
- 27 - Overview, and POSIX threads (`pthreads`)
- 28 - Locks
- 29 - Concurrent Data Structures
- 30 Condition Variables
- 31 - Semaphores
- 32 - Common Problems
- 33 - Event-Based Concurrency

Mutual Exclusion *mechanism summary*

- Mechanisms:
  - disabling interrupts
    - pretty much all we need if single core
    - but
      - privileged instruction
      - need to *trust* thread
      - not efficient
      - doesn't work on multiprocessors
  - atomic instructions
    - test-and-set
      - set memory location to value, returning old value
    - compare-and-swap
      - store at memory location only if it equals specific value
    - load-linked store
      - load from memory location
      - store new value to same location (only if it has not been updated)