[537] Locks and Condition Variables

Tyler Harter
Review: using and designing basic locks
Problem 1

Do it.
Lock Evaluation

How to tell if a lock implementation is good?
Lock Evaluation

How to tell if a lock implementation is good?

**Fairness**: does everybody get a chance to use the lock?

**Performance**
- high contention (many threads per CPU, each contending)
- low contention (fewer threads, fewer locking attempts)
Lock Evaluation

How to tell if a lock implementation is good?

**Fairness**: does everybody get a chance to use the lock?

**Performance**
- high contention (many threads per CPU, each contending)
- low contention (fewer threads, fewer locking attempts)

*which are spinlocks better for?*
Ticket Lock Review

\[\text{turn} = 6\]
\[\text{ticket} = 6\]
turn = 6

ticket = 6
A `lock()` gets ticket 6, runs.

```
7 0
6 A
5
4 3
```

- `turn = 6`
- `ticket = 7`
A lock(): gets ticket 6, runs
B lock(): gets ticket 7, spins until turn=7

turn = 6
ticket = 0
A lock(): gets ticket 6, runs
B lock(): gets ticket 7, spins until turn=7
C lock(): gets ticket 0, spins until turn=0
A `lock()`: gets ticket 6, runs
B `lock()`: gets ticket 7, spins until `turn=7`
C `lock()`: gets ticket 0, spins until `turn=0`
A `unlock()`: `turn++`
B runs

```
6  7  0  1
B  C
5  4  3
```

`turn = 7`
`ticket = 1`
A lock(): gets ticket 6, runs
B lock(): gets ticket 7, spins until turn=7
C lock(): gets ticket 0, spins until turn=0
A unlock(): turn++
B runs
A lock(): gets ticket 1, spins until turn=1
A lock(): gets ticket 6, runs
B lock(): gets ticket 7, spins until turn=7
C lock(): gets ticket 0, spins until turn=0
A unlock(): turn++
B runs
A lock(): gets ticket 1, spins until turn=1
B unlock(): turn++
C runs

turn = 0
ticket = 2
A lock(): gets ticket 6, runs
B lock(): gets ticket 7, spins until turn=7
C lock(): gets ticket 0, spins until turn=0
A unlock(): turn++
B runs
A lock(): gets ticket 1, spins until turn=1
B unlock(): turn++
C runs
C unlock(): turn++
A runs

turn = 1
ticket = 2
A lock(): gets ticket 6, runs
B lock(): gets ticket 7, spins until turn=7
C lock(): gets ticket 0, spins until turn=0
A unlock(): turn++
B runs
A lock(): gets ticket 1, spins until turn=1
B unlock(): turn++
C runs
C unlock(): turn++
A runs
A unlock(): turn++

turn = 2
ticket = 2
A lock(): gets ticket 6, runs
B lock(): gets ticket 7, spins until turn=7
C lock(): gets ticket 0, spins until turn=0
A unlock(): turn++
B runs
A lock(): gets ticket 1, spins until turn=1
B unlock(): turn++
C runs
C unlock(): turn++
A runs
A unlock(): turn++
C lock(): gets ticket 2, runs
Problem 2

Do it.
typedef struct __lock_t {
    int ticket;
    int turn;
}!

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) {
    FAA(&lock->turn);
}
typedef struct __lock_t {
    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) {
        yield(); // spin
    }
}

void release(lock_t *lock) {
    FAA(&lock->turn);
}
Spinlock Performance

Waste…

Without yield: $O(\text{threads} \times \text{time_slice})$

With yield: $O(\text{threads} \times \text{context_switch})$
Spinlock Performance

Waste...

Without yield: $O(\text{threads} \times \text{time\_slice})$

With yield: $O(\text{threads} \times \text{context\_switch})$

So even with yield, we’re slow with high contention.
Problem 3

**Park**: put thread to sleep

**Unpark**: wakeup thread

(a) This spins on guard — why? (what is protected? what is not?)
(b) This still spins. Why is it better than a simple spin lock?
(c) In unlock, there is no setting of flag=0 when we unpark. Why?
(d) What is the race-condition bug in this code?
Problem 3

**Park**: put thread to sleep
**Unpark**: wakeup thread

(a) This spins on guard — why? (what is protected? what is not?)
(b) This still spins. Why is it better than a simple spin lock?
(c) In unlock, there is no setting of flag=0 when we unpark. Why?
(d) What is the race-condition bug in this code?
Race Condition (2 had lock)

**Thread 1**
if (lock->flag == 0)
queue_push(lock->q, gettid());
lock->guard = 0;
park();

(in lock)

**Thread 2**
while (xchg(&lock->guard, 1) == 1)
if (queue_empty(lock->q))
unpark(queue_pop(lock->q));
lock->guard = 0;

(in unlock)
Incorrect Code

```c
void lock(lock_t *lock) {
    while (xchg(&lock->guard, 1) == 1) // spin
        ;
    if (lock->flag == 0) { // lock is free: grab it!
        lock->flag = 1;
        lock->guard = 0;
    } else { // lock not free: sleep
        queue_push(lock->q, gettid());
        lock->guard = 0;
        park(); // put self to sleep
    }
}
```
void lock(lock_t *lock) {
    while (xchg(&lock->guard, 1) == 1)  
        ; // spin
    if (lock->flag == 0) { // lock is free: grab it!
        lock->flag = 1;
        lock->guard = 0;
    } else { // lock not free: sleep
        queue_push(lock->q, gettid());
        setpark();
        lock->guard = 0;
        park(); // put self to sleep
    }
}
Queue Lock

RUNNABLE: A, B, C, D

RUNNING: <empty>

WAITING: <empty>
Queue Lock

RUNNABLE: B, C, D

RUNNING: A

WAITING: <empty>
Queue Lock

RUNNABLE: C, D, A

RUNNING: B

WAITING: <empty>
Queue Lock

RUNNABLE: C, D, A

RUNNING: 

WAITING: B
Queue Lock

RUNNABLE: D, A

RUNNING: C

WAITING: B
Queue Lock

RUNNABLE: A, C

RUNNING: D

WAITING: B

lock

try lock (sleep)
Queue Lock

RUNNABLE:  A, C

RUNNING: 

WAITING:  B, D
Queue Lock

RUNNABLE: C

RUNNING: A

WAITING: B, D
Queue Lock

RUNNABLE: A

RUNNING: C

WAITING: B, D
Queue Lock

RUNNABLE: C
RUNNING: A
WAITING: B, D
Queue Lock

RUNNABLE: C
RUNNING: A
WAITING: B, D

A
lock
B
try lock (sleep)
C
try lock (sleep)
D
A
unlock
C
A
Queue Lock

RUNNABLE: B, D, C

RUNNING: A

WAITING:
Queue Lock

RUNNABLE: B, D, C

RUNNING: A

WAITING: 

A

lock

B

try lock (sleep)

C

try lock (sleep)

D

A

C

unlock

A
Queue Lock

RUNNABLE: D, C, A

RUNNING: B

WAITING:

A

B

try lock (sleep)

try lock (sleep)

D

A

C

A

B

unlock

lock

B

lock
Condition Variables
Concurrent Objectives

**Mutual exclusion** (e.g., A and B don’t run at same time)
- solved with *locks*

**Ordering** (e.g., B runs after A)
- solved with *condition variables*
Ordering Example: Join

```c
pthread_t p1, p2;

Pthread_create(&p1, NULL, mythread, "A");
Pthread_create(&p2, NULL, mythread, "B");

// join waits for the threads to finish
Pthread_join(p1, NULL);
Pthread_join(p2, NULL);

printf("main: done\n [balance: %d]\n [should: %d]\n", balance, max*2);
return 0;
```
Ordering Example: Join

```c
pthread_t p1, p2;

Pthread_create(&p1, NULL, mythread, "A");
Pthread_create(&p2, NULL, mythread, "B");

// join waits for the threads to finish
Pthread_join(p1, NULL);
Pthread_join(p2, NULL);

printf("main: done\n [balance: %d]\n [should: %d]\n", balance, max*2);
return 0;
```
Spin Join (Bad!)

volatile int done = 0;

void *child_func(void *arg) {
    ... do some work ...
    done = 1;
}

int main(...)
{
pthread_t child;
Pthread_create(....child_func...);
while (!ready)
    ; // spin
printf("child is done\n");
}
Spin Join (Bad!)

volatile int done = 0;

void *child_func(void *arg) {
    ... do some work ...
    done = 1;
}

int main(...) {
    pthread_t child;
    Pthread_create(…child_func…);
    while (!ready)  
        ; // spin  
    printf("child is done\n");
}
Spin Join (Bad!)

volatile int done = 0;

void *child_func(void *arg) {
    ... do some work ...
    done = 1;
}

int main(…) {
    pthread_t child;
    Pthread_create(…child_func…);
    while (!ready) // spin
        // spin
    printf(“child is done\n”);
}
Spin Join (Bad!)

volatile int done = 0;

void *child_func(void *arg) {
    ... do some work ...
    done = 1;
}

int main(...) {
    pthread_t child;
Pthread_create(...child_func...);
    while (!ready) // spin
        ;
    printf("child is done\n");
}

want parent to sleep until done=1

need way for child to “nudge” parent
Condition Variables

CVs are used to wait until a some variable meets some condition.

CV’s are more like channels than variables. B waits for a signal on channel before running. A sends signal when it is time for B to run.

A CV also has a queue of waiting threads.
Broken CV’s

**wait**(cond_t *cv)
- puts caller to sleep (and on queue)

**signal**(cond_t *cv)
- wake a single waiting thread (if >= 1 thread is waiting)
- if there is no waiting thread, just return w/o doing anything
Broken CV’s

**wait** (cond_t *cv)
- puts caller to sleep (and on queue)

**signal** (cond_t *cv)
- wake a single waiting thread (if >= 1 thread is waiting)
- if there is no waiting thread, just return w/o doing anything
if (!ready) {
    wait(&cv);

    lock(&mutex);
    // critical section
    unlock(&mutex);
}
if (!ready)
  wait(&cv);

lock(&mutex);
// critical section
unlock(&mutex);

what if another thread sets ready=1 here?
lock(&mutex);
// critical section
if (!ready)
    wait(&cv);
unlock(&mutex);
lock(&mutex);
// critical section
if (!ready)
    wait(&cv);
unlock(&mutex);

nobody can wake us up because we hold mutex
Broken CV's

**wait**(*cond_t *cv*)
- puts caller to sleep

**signal**(*cond_t *cv*)
- wake a single waiting thread (if >= 1 thread is waiting)
- if there is no waiting thread, just return w/o doing anything
Correct CV’s

**wait**(cond_t *cv, mutex_t *lock)
- assumes the lock is held when wait() is called
- puts caller to sleep + releases the lock (atomically)
- when awoken, reacquires lock before returning

**signal**(cond_t *cv)
- wake a single waiting thread (if >= 1 thread is waiting)
- if there is no waiting thread, just return w/o doing anything
Correct CV’s

wait(cond_t *cv, mutex_t *lock)
- assumes the lock is held when wait() is called
- puts caller to sleep + releases the lock (atomically)
- when awoken, reacquires lock before returning

signal(cond_t *cv)
- wake a single waiting thread (if >= 1 thread is waiting)
- if there is no waiting thread, just return w/o doing anything

requires kernel support!
Weird CV Behavior 1

Condition Variable

wait → thread

wait → thread
Weird CV Behavior 1

Condition Variable \rightarrow \text{wait} \rightarrow \text{thread}
Weird CV Behavior 1

only one thread gets a signal
Weird CV Behavior 2

Condition Variable

wait → thread

wait → thread
Weird CV Behavior 2

?q signal
Condition Variable

wait
thread

wait
thread
Weird CV Behavior 2

Condition Variable

wait → thread
Weird CV Behavior 2

Condition Variable

wait

thread

spurious signals are possible
Weird CV Behavior 3

Condition Variable
Weird CV Behavior 3

thread $\xrightarrow{\text{signal}}$ Condition Variable
Weird CV Behavior 3

Condition Variable
Weird CV Behavior 3

Condition Variable

wait

thread
Weird CV Behavior 3

Condition Variable

wait

thread

waits forever…
Weird CV Behavior 3

Condition Variable

wait

thread

waits forever...

signals lost if nobody waiting at that time
 Guarantee

upon signal, at least one thread will wake (if there is at least one waiting)
upon signal, at least one thread will wake (if there is at least one waiting)
Guarantee

upon signal, at least one thread will wake
(if there is at least one waiting)
Guarantee

upon signal, at least one thread will wake
(if there is at least one waiting)
Do code examples

goto terminal
“Condition Variables” in xv6 code

proc.c:
- **sleep()** is like cond_wait()
- **wakeup()** is like cond_signal()

Example use case:
- piperead() and pipewrite() in pipe.c
Summary

Need both mutual exclusion AND ordering!

Condition variables provide ordering.