CONCURRENCY: LOCKS

Shivaram Venkataraman
CS 537, Spring 2020
Project 2b is done!
- Project 3 is out!
- Project 1b grades out today?
- Project 2a grades this week

Discussion section today 5:30 pm 105 Psych
No class on Thursday
Concurrency

What are some of the challenges in concurrent execution?
How do we design locks to address this?
RECAP
Thread 1

mov 0x123, %eax

add %0x1, %eax

mov %eax, 0x123

Thread 2

mov 0x123, %eax

add %0x2, %eax

mov %eax, 0x123
WHAT DO WE WANT?

Want 3 instructions to execute as an uninterruptable group

That is, we want them to be atomic

```
mov 0x123, %eax
add %0x1, %eax
mov %eax, 0x123
```

More general: Need mutual exclusion for critical sections

if thread A is in critical section C, thread B isn’t
(okay if other threads do unrelated work)
LOCK IMPLEMENTATION GOALS

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
Race Condition with Load and Store

shared global

*lock == 0 initially

Thread 1
while(*lock == 1)
*lock = 1

Thread 2
while(*lock == 1)
*lock = 1

Both threads grab lock!

Problem: Testing lock and setting lock are not atomic
XCHG: ATOMIC EXCHANGE OR TEST-AND-SET

How do we solve this? Get help from the hardware!

```c
int xchg(int *addr, int newval) {
    int old = *addr;
    *addr = newval;
    return old;
}
```

1. Stores newval into addr
2. Returns what used to be in addr

```asm
movl 4(%esp), %edx
movl 8(%esp), %eax
xchgl (%edx), %eax
ret
```

Atomically
typedef struct __lock_t {
    int flag;
} lock_t;

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

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

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

int xchg(int *addr, int newval)
{
    // Wait until lock is held
    while (xchg(&lock->flag, 1) != 1) {}  // Spin
    lock->flag = 1; // Else grab the lock
    // Some body else has lock
}
int CompareAndSwap(int *addr, int expected, int new) {
    int actual = *addr;
    if (actual == expected)
        *addr = new;
    return actual;
}

void acquire(lock_t *lock) {
    while(CompareAndSwap(&lock->flag, 0, 1) == 1);
    // spin-wait (do nothing)
}
Basic Spinlocks are unfair

Scheduler is unaware of locks/unlocks!
**FAIRNESS: 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

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

Acquire: Grab ticket; Spin while not thread’s ticket != turn
Release: Advance to next turn
TICKET LOCK EXAMPLE

A lock(): ticket = 0 → grabs the lock
B lock(): ticket = 1 → spin
C lock(): ticket = 2 → spin

A unlock(): inc turn value
B unlock(): B gets lock, turn = 2
C unlock(): C gets lock, turn = 3
A unlock(): A gets lock, turn = 4

Avoided unfair

Ticket

0
1
2
3
4
5
6
7

RR sched

= 1

Turn
TICKET LOCK IMPLEMENTATION

```c
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);
    // spin
    while (lock->turn != myturn);
}

void release(lock_t *lock) {
    FAA(&lock->turn);
    // indicate that next ticket holder can get the lock.
}
```
Spinlock Performance

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 wasteful
CPU SCHEDULER IS IGNORANT

CPU scheduler may run B, C, D instead of A even though B, C, D are waiting for A
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();
}

void release(lock_t *lock) {
    FAA(&lock->turn);
}
YIELD INSTEAD OF SPIN

A gets a second turn at 80ms with yield vs.

no yield:

yield:

B calls yield
a = 1
int b = xchg(&a, 2)
int c = CAS(&b, 2, 3)
int d = CAS(&b, 1, 3)

Final values

\[ a = 2 \]
\[ b = 3 \]
\[ c = 1 \]
\[ d = 1 \]

Assuming round-robin scheduling, 10ms time slice. Processes A, B, C, D, E, F, G, H in the system

Timeline:

A: lock() ... compute ... unlock()
B: lock() ... compute ... unlock()
C: lock()

A will complete at 90ms
B will get the lock at 90ms → SPIN
B has 30 ms of compute, when does C get the lock?

C D E F G H

80 90 100

60 ms

160 170

60 ms

C gets the lock

29 ??

if we count context switches

B can grab the lock

Context switch = 1 ms

28 ms
Spinlock Performance

Waste of CPU cycles?
   Without yield: \( O(\text{threads} \times \text{time\_slice}) \)
   With yield: \( O(\text{threads} \times \text{context\_switch}) \)

Even with yield, spinning is slow with high thread contention

Next improvement: Block and put thread on waiting queue instead of spinning
LOCK IMPLEMENTATION: BLOCK WHEN WAITING

Remove waiting threads from scheduler runnable queue (e.g., park() and unpark(threadID))

Scheduler runs any thread that is runnable
A B D contend for lock, C is not contending

RUNNABLE: A, B, C, D

RUNNING: A

WAITING: B, D

A has 60 ms worth of work
20ms is the timeslice

B or D can be moved into Runnable

B or D were not considered
typedef struct {
    bool lock = false;
    bool guard = false;
    queue_t q;
} LockT;

void acquire(LockT *l) {
    while (XCHG(&l->guard, true));
    if (l->lock) {
        qadd(l->q, tid);
        l->guard = false;
        park();     // blocked
    } else {
        l->lock = true;
        l->guard = false;
    }
}

void release(LockT *l) {
    while (XCHG(&l->guard, true));
    if (qempty(l->q)) l->lock=false;
    else unpark(qremove(l->q));
    l->guard = false;
}
(a) Why is guard used?
   We prevent races in updating the queue

(b) Why okay to spin on guard?
   Amount of time spin is small

(c) In release(), why not set lock=false when unpark?
   l->lock = true is handed over from release to acquire

(d) Is there a race condition?

```c
void acquire(LockT *l) {
    while (XCHG(&l->guard, true));
    if (l->lock) {
        qadd(l->q, tid);
        l->guard = false;
        park();  // blocked
    } else {
        l->lock = true;
        l->guard = false;
    }
}

void release(LockT *l) {
    while (XCHG(&l->guard, true));
    if (qempty(l->q)) l->lock=false;
    else {
        unpark(qremove(l->q));
        l->guard = false;
    }
}
```
Thread 1 (in lock)
if (l->lock) {
    qadd(l->q, tid);
    l->guard = false;
}

Thread 2 (in unlock)

while (TAS(&l->guard, true));
if (qempty(l->q)) // false!!
else unpark(qremove(l->q));
l->guard = false;

park(); // block
typedef struct {
    bool lock = false;
    bool guard = false;
    queue_t q;
} LockT;

void acquire(LockT *l) {
    while (TAS(&l->guard, true));
    if (l->lock) {
        qadd(l->q, tid);
        // notify of plan
        l->guard = false;
        park(); // unless unpark()
    } else {
        l->lock = true;
        l->guard = false;
    }
}

void release(LockT *l) {
    while (TAS(&l->guard, true));
    if (qempty(l->q)) l->lock=false;
    else unpark(qremove(l->q));
    l->guard = false;
}
SPIN-WAITING VS BLOCKING

Each approach is better under different circumstances

Uniprocessor
- Waiting process is scheduled → Process holding lock isn’t
- 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, \( t \), before lock is released
  - Lock released quickly → Spin-wait
  - Lock released slowly → Block
- Quick and slow are relative to context-switch cost, \( C \)
WHEN TO SPIN-WAIT? WHEN TO BLOCK?

If know how long, $t$, before lock released, can determine optimal behavior
How much CPU time is wasted when spin-waiting?
  $t$
How much wasted when blocking?

What is the best action when $t < C$?

When $t > C$?

Problem:
  Requires knowledge of future; too much overhead to do any special prediction
Two-Phase Waiting

Theory: Bound worst-case performance; ratio of actual/optimal
When does worst-possible performance occur?

\[
\text{Spin for very long time } t \gg C \\
\text{Ratio: } t/C \text{ (unbounded)}
\]

Algorithm: Spin-wait for C then block $\rightarrow$ Factor of 2 of optimal

Two cases:

- \( t < C \): optimal spin-waits for \( t \); we spin-wait \( t \) too
- \( t > C \): optimal blocks immediately (cost of \( C \)); we pay spin \( C \) then block (cost of \( 2C \));

\[
\frac{2C}{C} \rightarrow 2\text{-competitive algorithm}
\]
NEXT STEPS

Project 3: out now!
Discussion: Today at 5.30pm in 105 Psychology

No class on Thursday
Next Tuesday: Condition Variables