CONCURRENCY: LOCKS

Shivaram Venkataraman
CS 537, Spring 2020
- Project 3 is out!
- Project 1b grades out today?
- Project 2a grades this week
Concurrency
   What are some of the challenges in concurrent execution?
   How do we design locks to address this?
RECAP
MOTIVATION FOR CONCURRENCY
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

```assembly
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

*lock == 0 initially

Thread 1  Thread 2
while(*lock == 1)  while(*lock == 1)
                  *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
// xchg(int *addr, int newval)
// return what was pointed to by addr
// at the same time, store newval into addr
int xchg(int *addr, int newval) {
    int old = *addr;
    *addr = newval;
    return old;
}
```

```asm
movl 4(%esp), %edx
movl 8(%esp), %eax
xchgl (%edx), %eax
ret
```
LOCK IMPLEMENTATION WITH XCHG

typedef struct __lock_t {
    int flag;
} lock_t;

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

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

void release(lock_t *lock) {
    lock->flag = ??;
}
OTHER ATOMIC HW INSTRUCTIONS

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, , ) == ) ;
    // 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

```c
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():
B lock():
C lock():

A unlock():

A lock():
B unlock():

C unlock():
A unlock():

Ticket

0
1
2
3
4
5
6
7

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

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

no yield:

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

Final values

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()
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:

WAITING:

A has 60 ms worth of work
20ms is the timeslice
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;
}
LOCK IMPLEMENTATION: BLOCK WHEN WAITING

(a) Why is guard used?

(b) Why okay to spin on guard?

(c) In release(), why not set lock=false when unpark?

(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;
        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;
}
```
Race Condition

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;

setpark() fixes race condition
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?

Spin for very long time \( t >> C \)
Ratio: \( t/C \) (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 2 \( C \));
  \( 2C / C \rightarrow 2 \)-competitive algorithm
Project 3: out now!
Discussion: Today at 5.30pm in 105 Psychology

No class on Thursday
Next Tuesday: Condition Variables