CONCURRENCY: LOCKS

Shivaram Venkataraman
CS 537, Spring 2019
ADMINISTRIVIA

- Project 2b is due Wed Feb 27th, 11:59pm
- Project 2a grades out by tonight
Concurrency

What are some of the challenges in concurrent execution?
How do we design locks to address this?
RECAP
MOTIVATION FOR CONCURRENCY

Performance (vs. VAX-11/780)

Intel D850EMVR motherboard (3.06 GHz, Pentium 4 processor with Hyper-Threading Technology)
Intel V820 motherboard, 1.0 GHz Pentium III processor
Professional Workstation XP1000, 667 MHz 21264A
Digital AlphaServer 8400/6/575, 575 MHz 21264
AlphaServer 4000 5/600, 600 MHz 21164
Digital Alphastation 5/500, 500 MHz 21164
Digital Alphastation 5/300, 300 MHz 21164
Digital Alphastation 4/266, 266 MHz 21164
IBM POWERStation 100, 150 MHz
Digital 3000 AXP/500, 150 MHz
HP 9000/750, 66 MHz
IBM RS6000/540, 30 MHz
MIPS M2000, 25 MHz
MiPS M/120, 16.7 MHz
Sun-4/260, 16.7 MHz
VAX 8700, 22 MHz
AX-11/780, 5 MHz


25%/year 52%/year 23%/year 12%/year 3.5%/year
Thread 1
mov 0x123, %eax
add %0x1, %eax
mov %eax, 0x123

Thread 2
mov 0x123, %eax
add %0x2, %eax
mov %eax, 0x123

Variable
a = 0
Thread 1
mov 0x123, %eax
add %0x1, %eax
mov %eax, 0x123

Thread 2
mov 0x123, %eax
add %0x2, %eax
mov %eax, 0x123

Final value of a is incorrect.
NON-DETERMINISM

Concurrency leads to non-deterministic results

- Different results even with same inputs
- Race conditions

Whether bug manifests depends on CPU schedule!

How to program: imagine scheduler is malicious?!
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)
Synchronization

Build higher-level synchronization primitives in OS
Operations that ensure correct ordering of instructions across threads
Use help from hardware

Motivation: Build them once and get them right

<table>
<thead>
<tr>
<th>Monitors</th>
<th>Locks</th>
<th>Semaphores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Condition Variables</td>
<td></td>
</tr>
<tr>
<td>Loads</td>
<td>Stores</td>
<td>Test&amp;Set</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Disable Interrupts</td>
</tr>
</tbody>
</table>
Concurrency is needed for high performance when using multiple cores.

Threads are multiple execution streams within a single process or address space (share PID and address space, own registers and stack).

Context switches within a critical section can lead to non-deterministic bugs.
LOCKS
LOCKS

Goal: Provide mutual exclusion (mutex)

Allocate and Initialize
– `Pthread_mutex_t` mylock = PTHREAD_MUTEX_INITIALIZER;

Acquire
– Acquire exclusion access to lock;
– Wait if lock is not available (some other process in critical section)
– Spin or block (relinquish CPU) while waiting
– `Pthread_mutex_lock(&mylock);`

Release
– Release exclusive access to lock; let another process enter critical section
– `Pthread_mutex_unlock(&mylock);`
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
Implementing Synchronization

Atomic operation: No other instructions can be interleaved

Approaches
- Disable interrupts
- Locks using loads/stores
- Using special hardware instructions
IMPLEMENTING LOCKS: W/ INTERRUPTS

Turn off interrupts for critical sections
- Prevent dispatcher from running another thread
- Code between interrupts executes atomically

Disadvantages?
- Only works on uniprocessors
- Process can keep control of CPU for arbitrary length
- Cannot perform other necessary work

```c
void acquire(lockT *l) {
    disableInterrupts();
}

void release(lockT *l) {
    enableInterrupts();
}
```
Implementing LOCKS: w/ LOAD+STORE

Code uses a single **shared** lock variable

```c
// shared variable
boolean lock = false;

void acquire(Boolean *lock) {
    while (*lock) /* wait */;
    *lock = true;
}

void release(Boolean *lock) {
    *lock = false;
}
```

Does this work? What situation can cause this to not work?
LOCKS WITH VARIABLE DEMO
Race Condition with Load and Store

*lock == 0 initially

Thread 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!

// 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;
}
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 = ??;
}

int xchg(int *addr, int newval)
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, 0, 1) == 1) ;
    // spin-wait (do nothing)
}
a = 1
int b = xchg(&a, 2)
int c = CompareAndSwap(&b, 2, 3)
int d = CompareAndSwap(&b, 1, 3)
XCHG, CAS

\[
\begin{align*}
& a = 1 \\
& \text{int } b = \text{xchg}(\&a, 2) \\
& \text{int } c = \text{CompareAndSwap}(\&b, 2, 3) \\
& \text{int } d = \text{CompareAndSwap}(\&b, 1, 3)
\end{align*}
\]
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
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

Time
A runs
B runs
C runs
A runs

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 in terms of CPU
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;
} ;

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

Timeline
A: lock() ... compute ... unlock()
B: lock() ... compute ... unlock()
C: lock() ...
YIELD VS SPIN

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

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

If A’s compute is 20ms long, starting at t = 0, when does B get lock with spin?

\[ 110 \text{ ms} \]

If B’s compute is 30ms long, when does C get lock with spin?

If context switch time = 1ms, when does B get lock with yield?
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 ready queue (e.g., park() and unpark(threadID))

Scheduler runs any thread that is ready
RUNNABLE:  A, B, C, D
RUNNING:    
WAITING:    

0  20  40  60  80  100  120  140  160
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?
Race Condition

Thread 1 (in lock)
if (l->lock) {
    qadd(l->q, tid);
    l->guard = false;
}
park(); // block

Thread 2 (in unlock)
while (TAS(&l->guard, true));
if (qempty(l->q)) // false!!
else unpark(qremove(l->q));
l->guard = false;
typedef struct {
    bool lock = false;
    bool guard = false;
    queue_t q;
} LockT;

setpark() fixes race condition

void acquire(LockT *l) {
    while (TAS(&l->guard, true));
    if (l->lock) {
        qadd(l->q, tid);
        setpark();  // 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 block?

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 $2C$);
  $2C / C \rightarrow$ 2-competitive algorithm
NEXT STEPS

Project 2b: Due tomorrow!

Next class: Condition Variables