Concurrency:  
Mutual Exclusion (Locks)

Questions Answered in this Lecture:
• What are locks and how do we implement them?
• How do we use hardware primitives (atomics) to support efficient locks?
• How do we extend locks to multiprocessors?
• How do we use locks to implement concurrent data structures?
Announcements

• P2b is out; p1b grades should be posted tonight
• Final exam date set (Monday 5/6, 2-4PM, this room)
Review: which registers are shared between threads? Which are different?
Review: What do we need for correctness?

• Want 3 instructions to execute as an uninterruptable group
• That is, we want them to be an *atomic unit*

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

More general:

Need mutual exclusion for critical sections
• if process A is in critical section C, process B can’t be
  (okay if other processes do unrelated work)
Other Examples

• Consider multi-threaded programs that do more than increment a shared balance

• E.g., multi-threaded program with a shared linked-list
  • All concurrent operations:
    • Thread A inserts element a
    • Thread B inserts element b
    • Thread C looks up element c
Shared Linked List

```c
void list_insert(list_t *L, int key) {
    node_t *new = malloc(sizeof(node_t));
    assert(new);
    new->key = key;
    new->next = L->head;
    L->head = new;
}

int list_lookup(list_t *L, int key) {
    node_t *tmp = L->head;
    while (tmp) {
        if (tmp->key == key)
            return 1;
        tmp = tmp->next;
    }
    return 0;
}
```

typedef struct __node_t {
    int key;
    struct __node_t *next;
} node_t;

typedef struct __list_t {
    node_t *head;
} list_t;

```c
void list_init(list_t *L) {
    L->head = NULL;
}
```

What can go wrong?
What schedule leads to a problem?
Linked-List Race

Thread 1                                        Thread 2
new->key  =  key
new->next = L->head                             new->key  =  key
                                                      new->next = L->head
                                                      L->head = new
L->head = new

Both entries point to old head
Only one entry (which one?) can be the new head.
Resulting Linked List

head → T1’s node → old head → n3 → n4 → ...

T2’s node [orphan node]
Concurrent Linked List

```c
void list_insert(list_t *L, int key) {
    node_t *new = malloc(sizeof(node_t));
    assert(new);
    new->key = key;
    new->next = L->head;
    L->head = new;
}

int list_lookup(list_t *L, int key) {
    node_t *tmp = L->head;
    while (tmp) {
        if (tmp->key == key)
            return 1;
        tmp = tmp->next;
    }
    return 0;
}
```

```c
typedef struct __node_t {
    int key;
    struct __node_t *next;
} node_t;

typedef struct __list_t {
    node_t *head;
} list_t;

void list_init(list_t *L) {
    L->head = NULL;
}
```

How do we add locks to this?
Concurrent Linked List

```c
void list_insert(list_t *L, int key) {
    node_t *new = malloc(sizeof(node_t));
    assert(new);
    new->key = key;
    new->next = L->head;
    L->head = new;
}

int list_lookup(list_t *L, int key) {
    node_t *tmp = L->head;
    while (tmp) {
        if (tmp->key == key)
            return 1;
        tmp = tmp->next;
    }
    return 0;
}
```

typedef struct __node_t {
    int key;
    struct __node_t *next;
} node_t;

typedef struct __list_t {
    pthread_mutex_t lock;
    node_t *head;
} list_t;

void list_init(list_t *L) {
    L->head = NULL;
    pthread_mutex_init(&L->lock, NULL);
}

pthread_mutex_t lock;

One lock per list
Locking Linked Lists: Approach #1

```c
Void list_insert(list_t *L, int key) {
    node_t *new = malloc(sizeof(node_t));
    assert(new);
    new->key = key;
    new->next = L->head;
    L->head = new;
}

int list_lookup(list_t *L, int key) {
    node_t *tmp = L->head;
    while (tmp) {
        if (tmp->key == key)
            return 1;
        tmp = tmp->next;
    }
    return 0;
}
```

Consider everything critical section
Can critical section be smaller?

```c
pthread_mutex_lock(&L->lock);
...
```

```
pthread_mutex_unlock(&L->lock);
```
Locking Linked Lists: Approach #2

Critical section as small as possible

Void list_insert(list_t *L, int key) {
    node_t *new = malloc(sizeof(node_t));
    assert(new);
    new->key = key;
    new->next = L->head;
    L->head = new;
}

int list_lookup(list_t *L, int key) {
    node_t *tmp = L->head;
    while (tmp) {
        if (tmp->key == key)
            return 1;
        tmp = tmp->next;
    }
    return 0;
}
Locking Linked Lists: Approach #3

What about lookup?

If no list_delete(), locks not necessary

void list_insert(list_t *L, int key) {
    node_t *new = malloc(sizeof(node_t));
    assert(new);
    new->key = key;
    new->next = L->head;
    L->head = new;
}

int list_lookup(list_t *L, int key) {
    node_t *tmp = L->head;
    while (tmp) {
        if (tmp->key == key)
            return 1;
        tmp = tmp->next;
    }
    return 0;
}

#include <pthread.h>

pthread_mutex_lock(&L->lock);

pthread_mutex_unlock(&L->lock);

Synchronization

Build higher-level synchronization primitives in OS

- Operations that ensure correct ordering of instructions across threads

Motivation: Build them once and get them right

- Monitors
- Locks
- Semaphores
- Condition Variables
- Loads
- Stores
- Test&Set
- Disable Interrupts
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 (e.g., spinning)
Implementing Synchronization

• To implement, need atomic operations
• Atomic operation: guarantees no other instructions can be interleaved
• Examples of atomic operations
  • Code between interrupts on uniprocessors
    • Disable timer interrupts, don’t do any I/O
  • Loads and stores of words
    • Load r1, B
    • Store r1, A
  • Special hardware instructions
    • atomic test & set
    • atomic compare & swap
Implementing Locks: Using Interrupts

*Turn off interrupts for critical sections*

- Prevent dispatcher from running another thread
- Code between interrupts executes atomically

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

void release(lock_t *l) {
    enableInterrupts();
}
```

*Disadvantages??*

- Only works on uniprocessors
- Process can keep control of CPU for arbitrary length
- Cannot perform other necessary work
Implementing Locks: Using Load+Store

Code uses a single *shared* lock variable

```c
bool lock = false; // shared variable
void acquire(bool *lock) {
    while (*lock); /* wait */
    *lock = true;
}
void release(bool *lock) {
    *lock = false;
}
```

Why doesn’t this work? Example schedule that fails with 2 threads?
Race Condition with LOAD and STORE

*lock == 0 initially

Thread 1

while (*lock == 1);

Thread 2

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

// 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;
}

static inline unsigned
xchg(volatile unsigned int *addr, unsigned int newval)
{
    unsigned result;
    asm volatile("lock; xchgl %0, %1" :
        "+m" (*addr), "=a" (result) :
        "1" (newval) : "cc");
    return result;
}
XCHG Implementation

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)
XCHG Implementation

typedef struct __lock_t {
    int flag;
} lock_t;

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

void acquire(lock_t *lock) {
    while (xchg(&lock->flag, 1) == 1);
    // spin-wait (do nothing)
}

void release(lock_t *lock) {
    lock->flag = 0;
}
Other Atomic HW Instructions

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

void acquire(lock_t *lock) {
    while(CompareAndSwap(&lock->flag, ?, ?) == ?);
    // spin-wait (do nothing)
}
```
Other Atomic HW Instructions

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

```c
void acquire(lock_t *lock) {
    while(CompareAndSwap(&lock->flag, 0, 1) == 1) ;
    // spin-wait (do nothing)
}
```
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 independent 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()
B runs
A lock()
B unlock()
C runs
C unlock()
A runs
A unlock()
C lock():

Diagram:

Ticket
0
1
2
3
4
5
6
7

Turn
Ticket Lock Example

A lock():
B lock():
C lock():
A unlock():
B runs
A lock():
B unlock():
C runs
C unlock():
A runs
A unlock():
C lock():

Ticket

0
1
2
3
4
5
6
7

Turn
Ticket Lock Example

A lock():
B lock():
C lock():
A unlock():
B runs
A lock():
B unlock():
C runs
C unlock():
A runs
A unlock():
C lock():

Ticket

0
1
2
3
4
5
6
7

Turn
Ticket Lock Example

A lock():
B lock():
**C lock():**
A unlock():
B runs
A lock():
B unlock():
C runs
C unlock():
A runs
A unlock():
C lock():

Ticket

<table>
<thead>
<tr>
<th>Turn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>
Ticket Lock Example

A lock():
B lock():
C lock():
**A unlock():**
B runs
A lock():
B unlock():
C runs
C unlock():
A runs
A unlock():
C lock():

Ticket

<table>
<thead>
<tr>
<th>Ticket</th>
<th>Turn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>
Ticket Lock Example

A lock():
B lock():
C lock():
A unlock():
B runs
A lock():
B unlock():
C runs
C unlock():
A runs
A unlock():
C lock():

Ticket

Turn

0
1
2
3
4
5
6
7
Ticket Lock Example

A lock():
B lock():
C lock():
A unlock():
B runs
**A lock():**
B unlock():
C runs
C unlock():
A runs
A unlock():
C lock():
Ticket Lock Example

A lock():
B lock():
C lock():
A unlock():
B runs
A lock():
B unlock():
C runs
C unlock():
A runs
A unlock():
C lock():
Ticket Lock Example

A lock():
B lock():
C lock():
A unlock():
B runs
A lock():
B unlock():
C runs
C unlock():
A runs
A unlock():
C lock():

Ticket

0
1
2
3
4
5
6
7

Turn
Ticket Lock Example

A lock():
B lock():
C lock():
A unlock():
B runs
A lock():
B unlock():
C runs
C unlock():
A runs
A unlock():
C lock():

Ticket
0
1
2
3
4
5
6
7

Turn
Ticket Lock Example

A lock():
B lock():
C lock():
A unlock():
B runs
A lock():
B unlock():
C runs
C unlock():
**A runs**
A unlock():
C lock():

Ticket

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

Turn

- Ticket
- Turn
Tick Lock Example

A lock():
B lock():
C lock():
A unlock():
B runs
A lock():
B unlock():
C runs
C unlock():
A runs
A unlock():
C lock():
Ticket Lock Example

A lock():
B lock():
C lock():
A unlock():
B runs
A lock():
B unlock():
C runs
C unlock():
A runs
A unlock():
C lock():
Ticket Lock Implementation

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); // spin
}

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 instead of A even though B is waiting for A
Ticket Lock with `yield()`

typedef struct __lock_t {
    int ticket;
    int turn;
} __lock_t;

void acquire(lock_t *lock) {
    int myturn = FAA(&lock->ticket);
    while (lock->turn != myturn)
        yield();
}

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

void release (lock_t *lock) {
    FAA(&lock->turn);
}
Yield Instead of Spin

(no yield:)

(yield:)

A | B | C | D | A | B | C | D

lock | spin | spin | spin | unlock | lock | spin | spin

0 20 40 60 80 100 120 140 160

lock | unlock | lock

0 20 40 60 80 100 120 140 160

Yield Instead of Spin
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, spinning is slow with high thread contention

**Next improvement**: Block and put thread on waiting queue instead of spinning