Questions answered in this lecture:

Review threads and mutual exclusion for critical sections
How can locks be used to protect shared data structures such as linked lists?
Can locks be implemented by disabling interrupts?
Can locks be implemented with loads and stores?
Can locks be implemented with atomic hardware instructions?
Are spinlocks a good idea?

ANNOUNCEMENTS

P2: Due this Friday ➔ Extension to Sunday evening…
- Test scripts and handin directories available
- Purpose of graph is to demonstrate scheduler is working correctly

1st Exam: Congratulations for completing!
- Grades posted to Learn@UW: Average around 80%
  90% and up: A
  85 - 90: AB
  80 - 85: B
  70 - 80: BC
  60 - 70: C
  Below 60: D
- Return individual sheets in discussion section
- Exam with answers will be posted to course web page soon…

Read as we go along!
- Chapter 28
Review:
Which registers store the same/different values across threads?

All general purpose registers are virtualized
→ each thread given impression of own copy
**REVIEW: WHAT IS NEEDED FOR CORRECTNESS?**

Balance = balance + 1;

Instructions accessing shared memory must execute as uninterruptable group
- Need group of assembly instructions to be atomic

```
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 (okay if other processes do unrelated work)

**OTHER EXAMPLES**

Consider multi-threaded applications that do more than increment shared balance

Multi-threaded application with shared linked-list
- All concurrent:
  - Thread A inserting element a
  - Thread B inserting element b
  - Thread C looking up element c
**SHARED LINKED LIST**

```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;
}

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;
}
```

**LINKED-LIST RACE**

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>new-&gt;key = key</td>
<td>new-&gt;key = key</td>
</tr>
<tr>
<td>new-&gt;next = L-&gt;head</td>
<td>new-&gt;next = L-&gt;head</td>
</tr>
<tr>
<td></td>
<td>L-&gt;head = new</td>
</tr>
</tbody>
</table>

**Both entries point to old head**

Only one entry (which one?) can be the new head.
RESULTING LINKED LIST

LOCATING LINKED LISTS

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;

Void List_Init(list_t *L) {
    L->head = NULL;
}

How to add locks?
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 to add locks?

pthread_mutex_t lock; 

One lock per list – Fine if add to OTHER lists concurrently

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

VOID LOCKING LINKED LISTS

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

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

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

How to add locks?

pthread_mutex_t lock; 

One lock per list – Fine if add to OTHER lists concurrently

VOID LOCKING LINKED LISTS :

APPROACH #1

Pthread_mutex_lock(&L->lock);

Consider everything critical section
Can critical section be smaller?

Pthread_mutex_unlock(&L->lock);

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 #2

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;
}

Pthread_mutex_lock(&L->lock);
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;
}

Pthread_mutex_unlock(&L->lock);

Critical section small as possible

Pthread_mutex_lock(&L->lock);

Pthread_mutex_unlock(&L->lock);

Pthread_mutex_lock(&L->lock);

Pthread_mutex_unlock(&L->lock);

What about Lookup()?

If no List_Delete(), locks not needed

Pthread_mutex_unlock(&L->lock);

LOCKING LINKED LISTS:  
APPROACH #3

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;
}

Pthread_mutex_lock(&L->lock);
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;
}

Pthread_mutex_unlock(&L->lock);

Pthread_mutex_lock(&L->lock);

Pthread_mutex_unlock(&L->lock);

If no List_Delete(), locks not needed

Pthread_mutex_unlock(&L->lock);
IMPLEMENTING 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

<table>
<thead>
<tr>
<th>Monitors</th>
<th>Locks</th>
<th>Semaphores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Condition Variables</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>

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 waiting (starvation-free)
    • Must eventually allow each waiting thread to eventually enter

Fairness
  Each thread waits in some defined order

Performance
  CPU is not used unnecessarily (e.g., spinning)
  Fast to acquire lock if no contention with other threads
IMPLEMENTING SYNCHRONIZATION

To implement, need atomic operations

**Atomic operation**: 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 hw instructions**
  - Test&Set
  - Compare&Swap

IMPLEMENTING LOCKS: W/ INTERRUPTS

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

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

```c
void release(lockT *l) {
    enableInterruptions();
}
```

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

To implement, need atomic operations

Atomic operation: 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 hw instructions
  - Test&Set
  - Compare&Swap

IMPLEMENTING LOCKS: W/ LOAD+STORE

Code uses a single shared lock variable

Boolean lock = false; // shared variable

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

void release(Boolean *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               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

DEMO

Main-thread-3.c

Critical section not protected with faulty lock implementation
PETE RSON’S ALGORITHM

Assume only two threads (tid = 0, 1) and use just loads and stores

```c
int turn = 0; // shared across threads – PER LOCK

Void acquire() {
    lock[tid] = true;
    turn = 1-tid;
    while (lock[1-tid] && turn == 1-tid) /* wait */;
}

Void release() {
    lock[tid] = false;
}
```

Example of spin-lock

DIFFERENT CASES:
ALL WORK

Only thread 0 wants lock initially

```
Lock[0] = true;
turn = 1;
while (lock[1] && turn ==1)
;

In critical section

Lock[1] = true;
turn = 0;
while (lock[0] && turn == 0)

lock[0] = false;
while (lock[0] && turn == 0)
;```
DIFFERENT CASES:
ALL WORK

Thread 0 and thread 1 both try to acquire lock at same time

\begin{verbatim}
Lock[0] = true;
turn = 1;
while (lock[1] && turn == 1) {
}
Finish critical section
lock[0] = false;
\end{verbatim}

\begin{verbatim}
Lock[1] = true;
turn = 0;
while (lock[0] && turn == 0) {
}
\end{verbatim}

Thread 0 and thread 1 both want lock

\begin{verbatim}
Lock[0] = true;
turn = 1;
while (lock[1] && turn == 1) {
}
\end{verbatim}

\begin{verbatim}
Lock[1] = true;
turn = 0;
while (lock[0] && turn == 0) {
}
\end{verbatim}
DIFFERENT CASES: ALL WORK

Thread 0 and thread 1 both want lock;

```
Lock[0] = true;

turn = 1;

while (lock[1] && turn ==1)
    Lock[1] = true;

    turn = 0;
    while (lock[0] && turn == 0)

while (lock[1] && turn ==1)
```

PETEYERSON’S ALGORITHM: INTUITION

Mutual exclusion: Enter critical section if and only if
Other thread does not want to enter OR
Other thread wants to enter, but your turn (only 1 turn)

Progress: Both threads cannot wait forever at while() loop
Completes if other process does not want to enter
Other process (matching turn) will eventually finish

Bounded waiting (not shown in examples)
Each process waits at most one critical section
(because turn given to other)

Problem: doesn’t work on modern hardware
(doesn’t provide sequential consistency due to caching)
IMPLEMENTING SYNCHRONIZATION

To implement, need atomic operations

**Atomic operation**: 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 hw instructions
  - Test&Set
  - Compare&Swap

XCHG: ATOMIC EXCHANGE, OR TEST-AND-SET

```c
// xchg(int *addr, int newval)
// ATOMICALLY 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;
}

Need hardware support

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

typedef struct __lock_t {
    int flag;
} lock_t;

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

void acquire(lock_t *lock) {
    xchg(#addr, #newval)  
    // spin-wait (do nothing)
}

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

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;
}
DEMO: XCHG

Critical section protected with our lock implementation!!

Main-thread-5.c

BREAK
Other Atomic HW Instructions

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

Example of spin-lock

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 eventually

Fairness
Each thread waits in some determined ordered

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**;
**Wait** 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():

```
Ticket
0
1
2
3
4
5
6
7
```

Turn
**TICKET LOCK EXAMPLE**

A lock(): gets ticket 0, spins until turn = 0 → runs
B lock(): gets ticket 1, spins until turn=1
C lock(): gets ticket 2, spins until turn=2
A unlock(): turn++ (turn = 1)
B runs
A lock(): gets ticket 3, spins until turn=3
B unlock(): turn++ (turn = 2)
C runs
C unlock(): turn++ (turn = 3)
A runs
A unlock(): turn++ (turn = 4)
C lock(): gets ticket 4, runs

**TICKET LOCK IMPLEMENTATION**

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

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

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

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

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

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

FAA() used in textbook → conservative
Try this modification in Homework simulations

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);
}

Remember: yield() voluntarily relinquishes CPU for remainder of timeslice, but process remains READY
YIELD INSTEAD OF SPIN

Yield Instead of Spinlock Performance

Without yield: O(threads * time_slice)
With yield: O(threads * 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