UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537
Introduction to Operating Systems

Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau

Concurrency: Locks

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

Midterm Survey

- Turn up volume on microphone!
- What to do with discussion sections???
- Lecture pace is fast, but generally good
- Projects are challenging and interesting (but too much on passing tests?)
- Exam not any fun (fewer, more difficult questions)

1st Exam: Solutions posted, individual responses available

Project 3: Only xv6 part – rearrange address space of processes

- Watch parts of two videos
- Will need to submit user programs for testing

Read as we go along!

- Chapter 28

UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537
Introduction to Operating Systems

Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau

CONCURRENCY: LOCKS

Questions answered in this lecture:

Review: Why threads and mutual exclusion for critical sections?

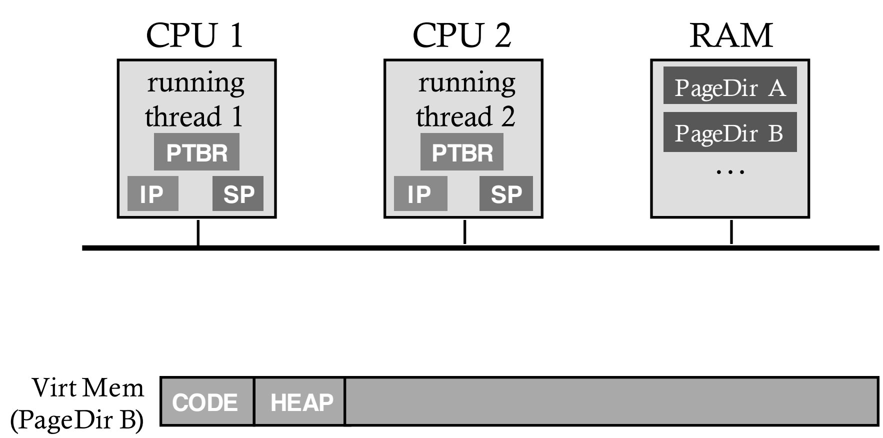
How can locks be used to protect shared data structures such as **linked lists**?

Can locks be implemented by **disabling interrupts**?

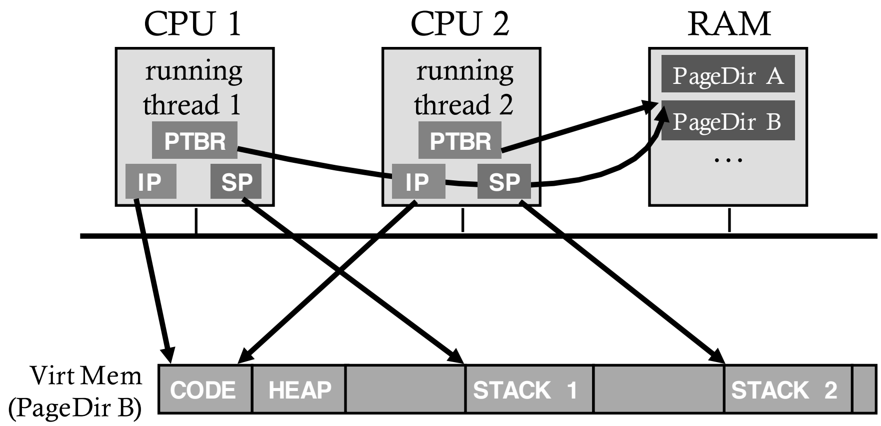
Can locks be implemented with **loads and stores**?

Can locks be implemented with **atomic hardware instructions**?

When are **spinlocks** a good idea?



Review:
Which registers store the same/different values across threads?



Review: What is needed for Correctness?

```
Balance = balance + 1;
```

Instructions accessing shared memory must execute as uninterruptable group

- Need instructions to be atomic

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

— critical section

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

```
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;
}
```

What can go wrong?
Find schedule that leads to problem?

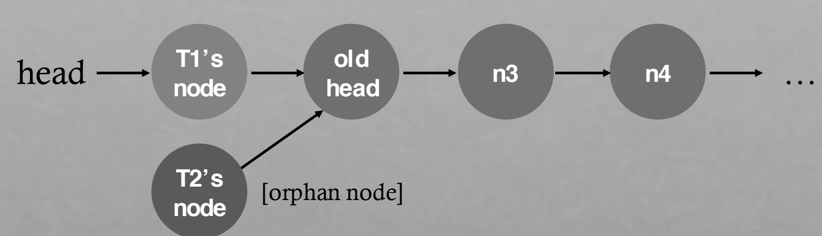
Linked-List Race

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



Locking 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?

Locking Linked Lists

```
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

```
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);
}
```

Locking Linked Lists : Approach #1

Pthread_mutex_lock(&L->lock);

Consider everything critical section
Can critical section be smaller?

Pthread_mutex_unlock(&L->lock);

Pthread_mutex_lock(&L->lock);

Pthread_mutex_unlock(&L->lock);

```
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 #2

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);

```
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()?

Pthread_mutex_lock(&L->lock);

Pthread_mutex_unlock(&L->lock);

Pthread_mutex_lock(&L->lock);

If no List_Delete(), locks not needed

Pthread_mutex_unlock(&L->lock);

```
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;
```

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

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: 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

```
Void acquire(lockT *l) {
      disableInterrupts();
}

Void release(lockT *l) {
      enableInterrupts();
}
```

Disadvantages??

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

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

Critical section not protected with faulty lock implementation

Peterson's Algorithm

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

```
int turn = 0; // shared
Boolean lock[2] = {false, false};
Void acquire() {
        lock[tid] = true;
        turn = 1-tid;
        while (lock[1-tid] && turn == 1-tid) /* wait */ ;
}
Void release() {
        lock[tid] = false;
}
```

Different Cases: All Work

Only thread 0 wants lock

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

Thread 0 and thread 1 both want lock;

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

Different Cases: All Work

Thread 0 and thread 1 both want lock

```
Lock[0] = true;
                              Lock[1] = true;
                              turn = 0;
turn = 1;

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

Different Cases: All Work

Thread 0 and thread 1 both want lock;

```
Lock[0] = true;
turn = 1;
                              Lock[1] = true;

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

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

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

Peterson's Algorithm: Intuition

Mutual exclusion: Enter critical section if and only if
- Other thread does not want to enter
- Other thread wants to enter, but your 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

Problem: doesn't work on modern hardware
(cache-consistency issues)

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 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) {
   ????;
   // 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;
}
```

Demo XCHG

Critical section protected with our lock implementation!!

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)
}
```

Other Atomic HW Instructions

```
int CompareAndSwap(int *ptr, 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)
}
```

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

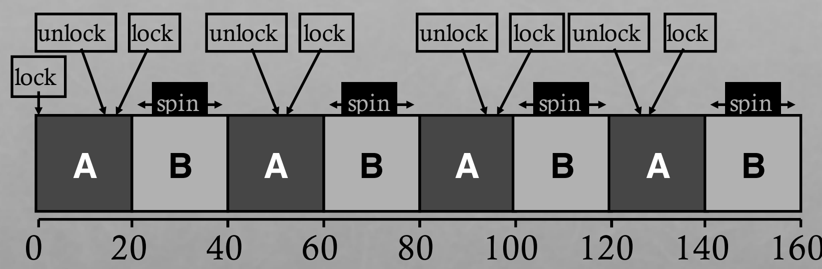
Fairness

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:

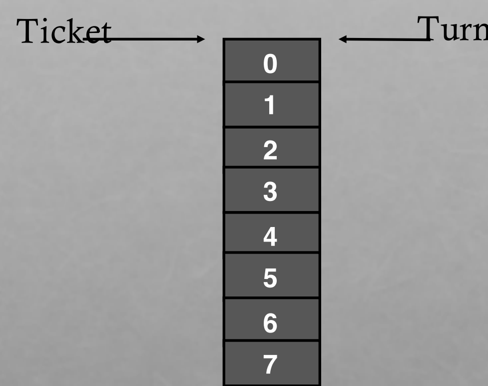
```
int FetchAndAdd(int *ptr) {
  int old = *ptr;
  *ptr = old + 1;
  return old;
}
```

```
Acquire: Grab ticket;
Spin while not thread's ticket != turn

Release: Advance to next turn
```

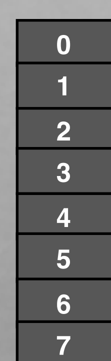
Ticket Lock Example

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



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;
}

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

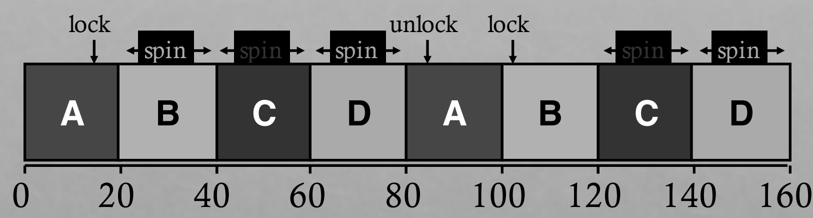
Fast when…

- 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()

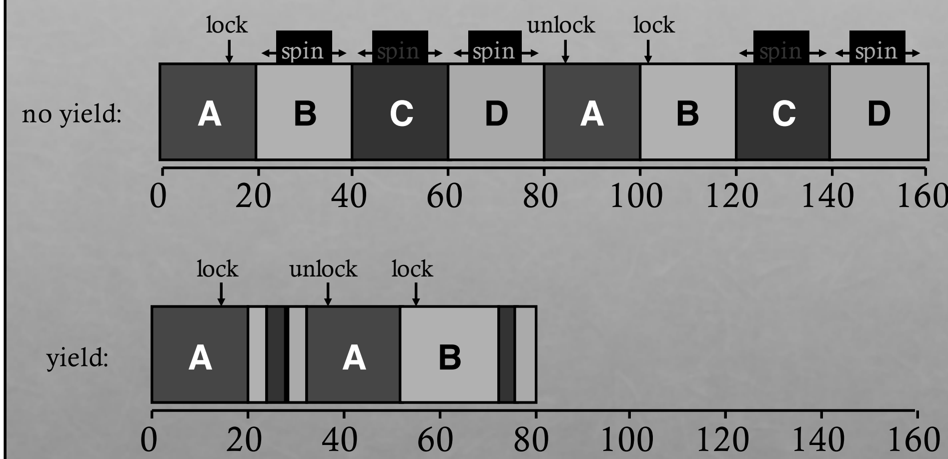
```
typedef struct __lock_t {
        int ticket;
        int turn;
}

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



Spinlock Performance

Waste…

- 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