Concurrency: Locks

Kai Mast
CS 537
Fall 2022

RECAP: THREADS

Threads enable concurrency within the same process

Each thread has its own

- Thread ID (TID)
- Set of registers, including program counter and stack pointer
- Stack for local variables and return addresses
  - There are now multiple stacks in same address space

(Book Chapter 26)

EXAMPLE: CONCURRENT INCREMENT

Data

```
balance at 0x9cd4 = 100
```

Code (C)

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

Code (Assembly)

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4
```

thread control blocks	Thread 1	Thread 2
	%eax ... %rip	%eax ... %rip

Registers are virtualized by OS;
Each thread “thinks” it has own

- Both threads run the above code
- Many possible **execution timelines** due to concurrency

Problem: Non-Determinism

Concurrency can lead to non-deterministic results

- Different results even with same inputs
- Race conditions

Whether bug manifests depends on CPU schedule!

How write concurrent programs: pretend scheduler is malicious

- Processes/threads can be stopped and resumed at any point
- Processes/threads might execute at different "speeds"

ATOMIC EXECUTION

We want instructions to execute as an uninterruptible 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 one thread is in critical section C, other threads cannot

(okay if other threads do unrelated work)

LOCKS

Goal: Provide mutual exclusion (mutex)

Allocate and Initialize

```
pthread_mutex_t mylock;
pthread_mutex_init(&mylock, NULL);
```

Acquire (or Lock)

- Acquire exclusive 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 (or Unlock)

- Release exclusive access to lock; let another process enter critical section
- `pthread_mutex_unlock(&mylock);`

(Book Chapter 28)

EXECUTION TIMELINE WITH LOCKING

Time

Thread 1	Thread 2
mutex_lock()	**mutex_lock()**
mov 0x9cd4, %eax	
add %0x1, %eax	
mov %eax, 0x9cd4	
mutex_unlock()	
	mov 0x9cd4, %eax
	add %0x1, %eax
	mov %eax, 0x9cd4
	mutex_unlock()

Thread 2 waits while thread 1 holds the lock

Once thread 1 is done, thread 2 can enter the critical section

OTHER EXAMPLES

Consider multi-threaded applications that do more than incrementing a single integer value

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

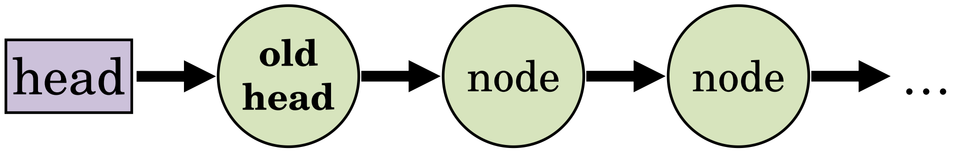
EXAMPLE: SHARED LINKED LIST

```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;
}
```

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

```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) {
    for(node_t *curr = l->head; curr;
            curr = curr->next) {
        if (curr->key == key) {
            return 1;
        }
    }
    return 0;
}
```

LINKED-LIST RACE




```
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;
}
```

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

LOCKING LINKED LISTS

```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;
}
```

```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) {
    for(node_t *curr = l->head; curr;
            curr = curr->next) {
        if (curr->key == key) {
            return 1;
        }
    }

    return 0;
}
```

How to add locks?

A SIMPLE THREAD-SAFE LINKED 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);
}
```

A SIMPLE THREAD-SAFE LINKED LIST (CONT.)

```
void list_insert(list_t *l, int key) {
    pthread_mutex_lock(l);
    node_t *new =
        malloc(sizeof(node_t));
    new->key = key;
    new->next = l->head;
    l->head = new;
    pthread_mutex_unlock(l);
}
```

```
int list_lookup(list_t *l, int key) {
    pthread_mutex_lock(l);
    for (node_t *curr = l->head;
        curr!=NULL; curr = curr->next) {
        if (curr->key == key) {
            pthread_mutex_unlock(l);
            return 1;
        }
    }
    pthread_mutex_unlock(l);
    return 0;
}
```

What is the impact of critical section size?

SYNCHRONIZATION PRIMITIVES

- OS provides higher-level synchronization primitives
- Rely on hardware features if possible

Motivation: Build them once and get them right

High-Level Userspace API	Monitors, Locks, Semaphores, Condition Variables
Hardware-Specific Implementation	Loads, Stores, Test&Set, Disable Interrupts

LOCK IMPLEMENTATION GOALS

Correctness

- *Mutual exclusion:* Only one thread in critical section at a time
- *Progress:* If several simultaneous requests, must allow one to proceed

Performance:

- CPU is not used unnecessarily (e.g., no spin-waiting or busy-waiting)
- Fast to acquire lock if no contention with other threads (common case!)

Fairness: Each thread eventually gets a turn

- Wait times are *bounded* (starvation-free)
- Must eventually allow each waiting thread to enter (assuming others eventually exit)

Approach 1: Disable Interrupts

```
void acquire_lock() {
    disable_interrupts();
}
```

```
void release_lock() {
    enable_interrupts();
}
```

Turn off interrupts while in a critical section

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

Problems:

- Only works on systems with a single CPU
- OS yields control to the process
  - What if the process is malicious? What if it runs forever?
- Not fair: process might disable interrupts for a long time

APPROACH 2: USE LOAD AND STORE

```
void acquire_lock(lock_t *l) {
    while (l->flag != 0) {}
    l->flag = 1;
}
```

```
void release_lock(lock_t *l) {
    l->flag = 0;
}
```

Approach

- Each lock has a single flag indicating its state
- `acquire_lock` **spins** until the lock is available

Does this work? What situation can cause this to not work?

PROBLEMS WITH APPROACH 2

Time

Thread 1	Thread 2	Thread 3
`l->flag = 1;`		
	`while (l->flag != 0) {}`	
`[critical section]`		
`l->flag = 0;`		
		`while (l->flag != 0) {}`
	`l->flag = 1;`	
		`l->flag = 1;`
	`[critical section]`	
		`[critical section]`
	`l->flag = 0;`	
		`l->flag = 0;`

When T1 releases lock, both T2 and T3 acquire it

THE NEED FOR HARDWARE SUPPORT

Software-based solutions exist but do not well on modern hardware

- See "Dekker's and Peterson's Algorithms" in the book

Better: Use a hardware primitive that can load and store atomically

- `test_and_set(&val, 1)` sets `val` to 1 and returns previous value
- Also called atomic exchange
- Specific instruction differs depending on CPU-architecture
  - e.g., `xchg` on x86

APPROACH 3: USE TEST-AND-SET

```
typedef struct __lock_t {
    int flag;
} lock_t;

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

void acquire(lock_t *lock) {
    while (test_and_set(&lock->flag, 1) == 1) {}
}

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

```
int test_and_set(int *var,
    int new_value) {
  int old_value = *var;
  *var = new_value;
  return old_value;
}
```

OTHER POSSIBLE HARDWARE INSTRUCTIONS

```
int compare_and_swap(int *val, int cmp, int new)
```

- Set `val` to `new` if `val==cmp`
- Otherwise, does nothing
- Always returns `val`'s previous value

```
int fetch_and_add(int *val)
```

- Increments `val` and returns its previous value

USE SPINNING WITH CAUTION

Consider the following scenario:

- Thread 1 is holding a lock for five time slices
- Ten other threads want to acquire the lock and spin
- The scheduler is Round-Robin with time slice length 10ms

What is the worst-case time spent on spinning? 500ms!

We need to avoid spinning as much as possible!
What could we do instead of spinning?

APPROACH 4: YIELD

Idea: Introduce a new system call `yield()` that voluntarily gives up CPU control

```
void acquire_lock(lock_t *l) {
    while ((test_and_set(&l->flag,1) == 1) {
        yield();
    }
}

void release_lock(lock_t *l) {
    l->flag = 0;
}
```

LOCK IMPLEMENTATION GOALS REVISITED

Yield-based locks (Approach 4)

- Correctness: Yes (only one thread can be in the CS)
- Performance: Better (still too many context switches)
- Fairness: No (scheduler does not know/respect wait order)

APPROACH 5: USE A QUEUE

Intuition: Two-step approach

- First lock a wait-queue and add current thread to it
  - Queue-lock is only held for a very short time
- When calling `release()`, pick the first thread from queue
- Needs to more system class
  - `park()` pauses/deschedules the current thread
  - `unpark(int thread_id)` resumes/schedules the specified thread

APPROACH 5: ACQUIRING LOCKS

```
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void init(lock_t *l) {
    l->flag = 0;
    l->guard = 0;
    queue_init(l->q);
}
```

```
void acquire(lock_t *l) {
    while(test_and_set(&l->guard,1) == 1)
        ; //acquire guard lock by spinning
    if (l->flag == 0) {
        l->flag = 1; // lock is acquired
        l->guard = 0;
    } else {
        queue_add(l->q, gettid());
        l->guard = 0;
        park();
    }
}
```

APPROACH 5: RELEASING LOCKS

```
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void init(lock_t *l) {
    l->flag = 0;
    l->guard = 0;
    queue_init(l->q);
}
```

```
void release(lock_t *l) {
    while(test_and_set(&l->guard,1) == 1)
        ; //acquire guard lock by spinning
    if (queue_empty(l->q)) {
        // let go of lock; no one wants it
        l->flag = 0;
    } else {
        // hold lock (for next thread!)
        unpark(queue_remove(m->q));
        l->guard = 0;
    }
}
```

SUMMARY

- Locks are a fundamental building block for concurrent applications and data structures
- Efficient locking mechanism require hardware and OS support
  - `compare_and_swap()`, atomic exchange, etc.
  - `yield()`, `park()`, etc.