ILLINOIS TECH

College of Computing

CS 450 Operating Systems

Mutual Exclusion

Yue Duan

Recap

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

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

Another Example

- 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;
}
```

```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?

What schedule leads to a problem?

Shared Linked List

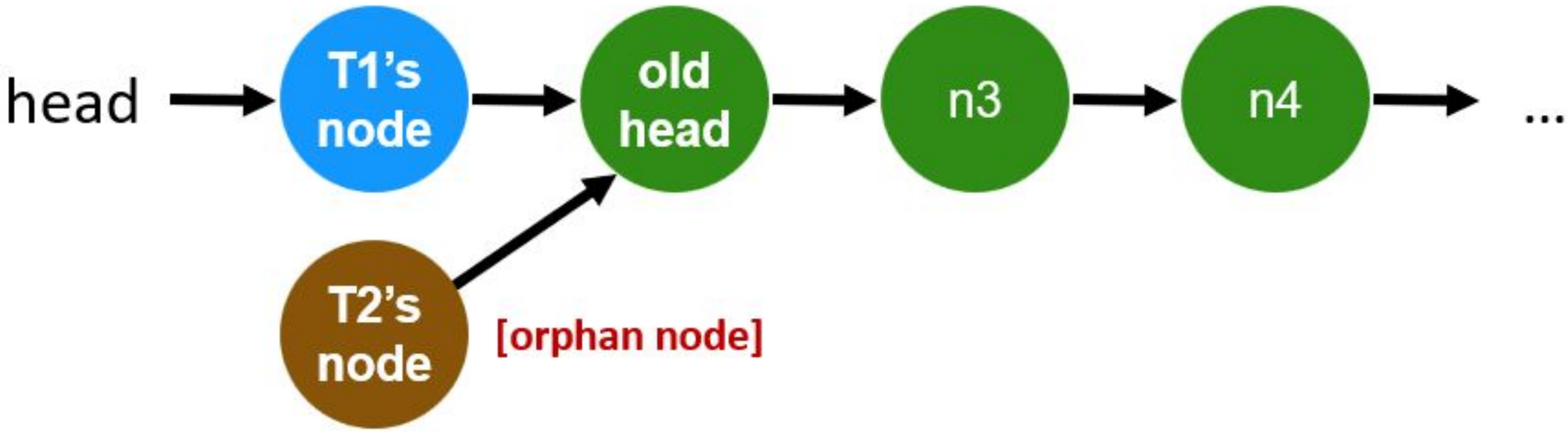
Thread 1	Thread 2
`new->key  = key`	
`new->next = L->head`	
	`new->key  = key`
	`new->next = L->head`
	`L->head   = new`
`L->head = new`	

time

Both entries point to old head

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

Shared Linked List



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;
}
```

```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;
}
```

```c
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

- Consider everything critical section
- Can critical section be smaller?

```
pthread_mutex_lock(&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;
pthread_mutex_unlock(&L->lock);
}
```

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

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

Locking Linked Lists : Approach #3

- What about list_lookup()?

```
pthread_mutex_lock(&L->lock);
pthread_mutex_unlock(&L->lock);
pthread_mutex_lock(&L->lock);
```

- If no list_delete(), locks not necessary

```
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;
}
```

Critical Section Requirements

- Mutual exclusion (mutex)
  - If one thread is in the critical section, then no other is
- Progress
  - A thread in the critical section will eventually leave the critical section
  - If some thread T is not in the critical section, then T cannot prevent other thread from entering the critical section
- Bounded waiting (no starvation)
  - If some thread T is waiting on the critical section, then T will eventually enter the critical section
- Performance
  - The overhead of entering and exiting the critical section is small with respect to the work being done within it

Building Critical Sections

- 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

Building Critical Sections

- **Locks**
  - Primitive, minimal semantics, used to build others
- Semaphores
  - Basic, easy to get the hang of, but hard to program with
- Monitors
  - High-level, requires language support, operations implicit
- Architecture help
  - Atomic read/write » Can it be done?

How to implement a lock? First try

```
pthread_trylock(mutex) {
    if (mutex==0) {
     mutex= 1;
     return 1;
    } else return 0;
}
```

```
Thread 0, 1, …

…//time to access critical region
while(!pthread_trylock(mutex); // wait
<critical region>
pthread_unlock(mutex)
```

- Does this work?
    - Assume reads/writes are atomic
- The lock itself is a critical region!
    - Chicken and egg
- Computer scientist struggled with how to create software locks

How to implement a lock? Second try

```
int turn = 1;
```

```
while (true) {
    while (turn != 1) ;
    critical section
    turn = 2;
    outside of critical section
}
```

```
while (true) {
    while (turn != 2) ;
    critical section
    turn = 1;
    outside of critical section
}
```

- This is called **alternation**
- It satisfies mutex:
  - If blue is in the critical section, then turn == 1 and if yellow is in the critical section then turn == 2
  - (turn == 1) ≡ (turn != 2)
- **Is there anything wrong with this solution?**

How to implement a lock? Second try

```
int turn = 1;
```

P1

```
while (true) {
    while (turn != 1) ;
    critical section
    turn = 2;
    outside of critical section
}
```

P2

```
while (true) {
    while (turn != 2) ;
    critical section
    turn = 1;
    outside of critical section
}
```

- **Is there anything wrong with this solution?**
    - break the progress requirement
    - **how**?
    - strictly alternating order; may not map well to application needs
    - consider the situation where P2 wants to enter its CS earlier than P1

Third try - Two Variables

```
Bool flag[2]
```

```
while (flag[1] != 0);
flag[0] = 1;
critical section
flag[0]=0;
outside of critical section
```

```
while (flag[0] != 0);
flag[1] = 1;
critical section
flag[1]=0;
outside of critical section
```

- We added two variables to try to break the race for the same variable
- **Is there anything wrong with this solution?**
  - break the mutual exclusion requirement
  - **how?**

Third try - Two Variables

```
Bool flag[2]
```

```
while (flag[1] != 0);
flag[0] = 1;
critical section
flag[0]=0;
outside of critical section
```

```
while (flag[0] != 0);
flag[1] = 1;
critical section
flag[1]=0;
outside of critical section
```

- **Is there anything wrong with this solution?**
  - break the mutual exclusion requirement

Fourth try – set before you check

```
Bool flag[2]
```

```
flag[0] = 1;
while (flag[1] != 0);
critical section
flag[0]=0;
outside of critical section
```

```
flag[1] = 1;
while (flag[0] != 0);
critical section
flag[1]=0;
outside of critical section
```

- **Is there anything wrong with this solution?**
  - **deadlock**
  - both threads could set their flag as true simultaneously and both will wait infinitely later on

Fifth try – double check and back off

```
Bool flag[2]
```

```
flag[0] = 1;
while (flag[1] != 0) {
        flag[0] = 0;
        wait a short time;
        flag[0] = 1;
}
critical section
flag[0]=0;
outside of critical section
```

```
flag[1] = 1;
while (flag[0] != 0) {
        flag[1] = 0;
        wait a short time;
        flag[1] = 1;
}
critical section
flag[1]=0;
outside of critical section
```

- **Is this a correct solution?**
  - indefinite postponement

Six try – Dekker's Algorithm

```
Bool flag[2]l
Int turn = 1;
```

```
flag[0] = 1;
while (flag[1] != 0) {
        if(turn == 2) {
        flag[0] = 0;
         while (turn == 2);
        flag[0] = 1;
       } //if
}//while
critical section
flag[0]=0;
turn=2;
outside of critical section
```

```
flag[1] = 1;
while (flag[0] != 0) {
        if(turn == 1) {
        flag[1] = 0;
         while (turn == 1);
        flag[1] = 1;
       } //if
}//while
critical section
flag[1]=0;
turn=1;
outside of critical section
```

Another solution: Peterson's Algorithm

```
int turn = 1;
bool try1 = false, try2 = false;
```

```
while (true) {
    try1 = true;
    turn = 2;
    while (try2 && turn != 1) ;
    critical section
    try1 = false;
    outside of critical section
}
```

```
while (true) {
    try2 = true;
    turn = 1;
    while (try1 && turn != 2) ;
    critical section
    try2 = false;
    outside of critical section
}
```

Some observations

- This stuff (software locks) is hard
  - Hard to get right
  - Hard to prove right
- It also is inefficient
  - A spin lock – waiting by checking the condition repeatedly
- Even better, software locks don't really work
  - Compiler and hardware reorder memory references from different threads
  - So, we need to find a different way
- Hardware help

Using Interrupts

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

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

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

Using Interrupts

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

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

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

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: atomic exchange, or test-and-set

```
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: atomic exchange, or 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 (xchg(&lock->flag, 1) == 1);
        // spin-wait (do nothing)
}

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

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)
}
```

THANK YOU!