CS 423
Operating System Design: **Synchronization**

Ram Alagappan

* Thanks for Prof. Bates and Prof. Xu for the slides.

Recap - Threads

Threads – share the same address space

What does this mean?

Is the stack shared across threads?

How about the heap?

How about registers? Which are specific to threads? Which are not?

Synchronization Motivation

```
static volatile int c = 0;
void *mythread(void *arg) {
int i;
for (i = 0; i < 1000000; i++) c++;
return NULL;
}
```

Main prints the value of c

What do you expect to be printed?

With 1 thread? With 2 threads?

Synchronization Motivation

What's going on here?

c++ boils down to something like this

```
mov mem_addr(c), eax
add 1, eax
mov eax, mem_addr(c)
```

Even on an uniprocessor!

Crux of the Problem

Uncontrolled scheduling – threads can be descheduled at any point in its execution

Without synchronization, you can have

Data race – result depends on the scheduling (with untimely descheduling, can get undesired result)

Non-determinism – result vary across runs

What we want:

mutual exclusion – a common way to do this?

Other Things to Worry About

Compiler/hardware might reorder instructions

Can this panic?

Thread 1	Thread 2
p = someComputation();	while (!pInitialized)
pInitialized = true;	;
	q = someFunction(p);
	if (q != someFunction(p))
	panic

Why would they do that!?

Why Reordering?

- Why do compilers reorder instructions?
  - Efficient code generation requires analyzing control/ data dependency
  - If variables can spontaneously change, most compiler optimizations become impossible
- Why do CPUs reorder instructions?
  - Write buffering: allow next instruction to execute while write is being completed

Fix: **memory barrier**

- Instruction to compiler/CPU
- All ops before barrier complete before barrier returns
- No op after barrier starts until barrier returns

Why Study in OS Class?

OS needs to provide synchronization primitives for threads within an application to synchronize

Turns out OS was (one of) the first multi-threaded application to worry about how to manage its internal data structures when multiple threads can access it

Too Much Milk!

	Person A	Person B
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	
12:40	Arrive at store.	Look in fridge. Out of milk.
12:45	Buy milk.	Leave for store.
12:50	Arrive home, put milk away.	Arrive at store.
12:55		Buy milk.
1:00		Arrive home, put milk away. Oh no!

Desired Behaviors

At most one person buys

- this is called *safety* (the program should never do anything bad)

Someone buys milk if needed

- this is called *Liveness* (the program eventually does something good)

Too Much Milk, Try #1

- Try #1: leave a note

```
if (!note)
    if (!milk) {
        leave note
        buy milk
        remove note
    }
```

Does this work?

Too Much Milk, Try #2

Thread A

```
leave note A
if (!note B) {
    if (!milk)
        buy milk
}
remove note A
```

Thread B

```
leave note B
if (!noteA) {
    if (!milk)
        buy milk
}
remove note B
```

Too Much Milk, Try #3

Thread A

```
leave note A
while (note B) // X
    do nothing;
if (!milk)
    buy milk;
remove note A
```

Thread B

```
leave note B
if (!noteA) {   // Y
    if (!milk)
        buy milk
}
remove note B
```

Can guarantee at X and Y that either:

(i) Safe for me to buy

(ii) Other will buy, ok to quit

Takeaways

Solution is complicated…

Generalizing to many threads is complex (what if N people try to buy milk instead of 2)

As we will see, HW can simplify (still hard!)

Crux: uncontrolled scheduling

Modern compilers and hardware can reorder

makes things worse!

Synchronization Roadmap

Concurrent Applications

Shared Objects

Bounded Buffer Barrier

Synchronization Variables

Semaphores Locks Condition Variables

Atomic Instructions

Interrupt Disable Test-and-Set

Hardware

Multiple Processors Hardware Interrupts

Locks (Programmer View)

- Lock::acquire
  - wait until lock is free, then take it
- Lock::release
  - release lock, waking up anyone waiting for it

1. At most one lock holder at a time (safety)
2. If no one holding, acquire gets lock (progress)
3. If all lock holders finish and no higher priority waiters, waiter eventually gets lock (progress)

Too Much Milk, Try #4

Locks allow concurrent code to be much simpler:

```
lock.acquire();
if (!milk)
   buy milk
lock.release();
```

Rules for Using Locks

- Lock is initially free
- Always acquire before accessing shared data structure
  - Beginning of procedure!
- Always release after finishing with shared data
  - End of procedure!
  - Only the lock holder can release
  - DO NOT throw lock for someone else to release
- Never access shared data without lock
  - Danger!

Ex: Thread-Safe Bounded Queue

```
tryget() {
    item = NULL;
    lock.acquire();
    if (front < tail) {
        item = buf[front % MAX];
        front++;
    }
    lock.release();
    return item;
}
```

```
tryput(item) {
    lock.acquire();
    if ((tail – front) < size) {
        buf[tail % MAX] = item;
        tail++;
    }
    lock.release();
}
```

Initially: front = tail = 0; lock = FREE; MAX is buffer capacity

Question(s)

- If tryget returns NULL, do we know the buffer is empty?

- If we poll tryget in a loop, what happens to a thread calling tryput?

Implementing Locks

So far – programmer perspective

Now, systems perspective! How to implement/realize a lock?

Take 1: using *only* atomic memory load/store

- See too much milk solution
- Comment on Peterson's (and similar) algorithms
- (Almost) nobody does this today!

Lock Implementation for Uniprocessor?

```
Lock::acquire() {
    disableInterrupts();
}
```

```
Lock::release() {
    enableInterrupts();
}
```

What is good about this approach?

What is bad?

Lock Implementation for Uniprocessor?

```
Lock::acquire() {
    disableInterrupts();
    if (value == BUSY) {
        waiting.add(myTCB);
        myTCB->state = WAITING;
        next = readyList.remove();
        switch(myTCB, next);
        myTCB->state = RUNNING;
    } else {
        value = BUSY;
    }
    enableInterrupts();
}
```

```
Lock::release() {
    disableInterrupts();
    if (!waiting.Empty()) {
        next = waiting.remove();
        next->state = READY;
        readyList.add(next);
    } else {
     value = FREE;
    }
    enableInterrupts();
}
```

Via Atomic Instructions

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

void init(lock_t *lock) {
    // 0: lock is available, 1: lock is held
    lock->flag = 0;
}

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

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

Test – return old value

Set – set the passed in value

HW does them atomically!

Next lecture about how to implement locks using them

Condition Variables

When do you need them?

- Waiting inside a critical section
  - Called only when holding a lock
- CV::Wait — atomically release lock and relinquish processor
  - Reacquire the lock when wakened
- CV::Signal — wake up a waiter, if any
- CV::Broadcast — wake up all waiters, if any

Condition Variables

```
methodThatWaits() {
    lock.acquire();
    // Read/write shared state

    while (!testSharedState()) {
        cv.wait(&lock);
    }

    // Read/write shared state
    lock.release();
}
```

```
methodThatSignals() {
    lock.acquire();
    // Read/write shared state

    // If testSharedState is now true
    cv.signal(&lock);


    // Read/write shared state
    lock.release();
}
```

Ex: Bounded Queue w/ CV

```
get() {
    lock.acquire();
    if (front == tail) {
        empty.wait(lock);
    }
    item = buf[front % MAX];
    front++;
    full.signal(lock);
    lock.release();
    return item;
}
```

```
put(item) {
    lock.acquire();
    if ((tail - front) == MAX) {
        full.wait(lock);
    }
    buf[tail % MAX] = item;
    tail++;
    empty.signal(lock);
    lock.release();
}
```

Initially: front = tail = 0; MAX is buffer capacity
empty/full are condition variables

Ex: Bounded Queue w/ CV

```
get() {
    lock.acquire();
    if (front == tail) {
        empty.wait(lock);
    }
    item = buf[front % MAX];
    front++;
    full.signal(lock);
    lock.release();
    return item;
}
```

```
put(item) {
    lock.acquire();
    if ((tail - front) == MAX) {
        full.wait(lock);
    }
    buf[tail % MAX] = item;
    tail++;
    empty.signal(lock);
    lock.release();
}
```

Is there a problem with this code?

Ex: Bounded Queue w/ CV

```
get() {
    lock.acquire();
    while (front == tail) {
        empty.wait(lock);
    }
    item = buf[front % MAX];
    front++;
    full.signal(lock);
    lock.release();
    return item;
}
```

```
put(item) {
    lock.acquire();
    while ((tail - front) == MAX) {
        full.wait(lock);
    }
    buf[tail % MAX] = item;
    tail++;
    empty.signal(lock);
    lock.release();
}
```

Mesa vs. Hoare Semantics

- Mesa (used widely)
  - Signal puts waiter on ready list
  - Signaler keeps lock and processor
  - Not necessarily the waiter runs next
- Hoare (almost no one uses)
  - Signal gives processor and lock to waiter
  - Waiter runs when woken up by signaler
  - When waiter finishes, processor/lock given back to signaler

FIFO Bounded Queue

(Correct under Hoare Semantics)

```
get() {
    lock.acquire();
    if (front == tail) {
        empty.wait(lock);
    }
    item = buf[front % MAX];
    front++;
    full.signal(lock);
    lock.release();
    return item;
}
```

```
put(item) {
    lock.acquire();
    if ((tail – front) == MAX) {
        full.wait(lock);
    }
    buf[last % MAX] = item;
    last++;
    empty.signal(lock);
  // CAREFUL: someone else ran
    lock.release();
}
```

Initially: front = tail = 0; MAX is buffer capacity
empty/full are condition variables

Condition Variables

- ALWAYS hold lock when calling wait, signal, broadcast
  - Condition variable is sync FOR shared state
  - ALWAYS hold lock when accessing shared state

- Condition variable is memoryless
  - If signal when no one is waiting, no op
  - If wait before signal, waiter wakes up

- Wait atomically releases lock
  - What if wait, then release?
  - What if release, then wait?

Condition Variables

- When a thread is woken up from wait, it may not run immediately
  - Signal/broadcast put thread on ready list
  - When lock is released, anyone might acquire it

- Wait MUST be in a loop

```
while (needToWait()) {
    condition.Wait(lock);
}
```

- Simplifies implementation
  - Of condition variables and locks
  - Of code that uses condition variables and locks

Synchronization Best Practices

- Identify objects or data structures that can be accessed by multiple threads concurrently
- Add locks to object/module
  - Grab lock on start to every method/procedure
  - Release lock on finish
- If need to wait
  - while(needToWait()) { condition.Wait(lock); }
  - Do not assume when you wake up, signaller just ran
- If do something that might wake someone up
  - Signal or Broadcast
- Always leave shared state variables in a consistent state
  - When lock is released, or when waiting

Remember the rules…

- Use consistent structure
- Always use locks and condition variables
- Always acquire lock at beginning of procedure, release at end
- Always hold lock when using a condition variable
- Always wait in while loop