Synchronization

Chapter 5 OSPP

Part I

Synchronization Motivation

- When threads concurrently read/write shared memory, program behavior is undefined
  - Two threads write to the same variable; which one should win?
- Thread schedule is non-deterministic
  - Behavior may change when program is re-run
- Compiler/hardware instruction reordering
- Multi-word operations are not atomic

  e.g. i = i + 1

Question: Can this panic?

Thread 1

```
p = someComputation();
pInitialized = true;
```

Thread 2

```
while (!pInitialized)
    ;
q = someFunction(p);
if (q != someFunction(p))
    panic
```

Can p change?

Why Reordering?

- Why do compilers reorder instructions?
  - Efficient code generation requires analyzing control/data dependency
- Why do CPUs reorder instructions?
  - Out order execution for efficient pipelining and branch prediction

Fix: **memory barrier**

- Instruction to compiler/CPU, x86 has one
- All ops before barrier complete before barrier returns
- No op after barrier starts until barrier returns

Too Much Milk Example

	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!

Definitions

Race condition: output of a concurrent program depends on the order of operations between threads

Mutual exclusion: only one thread does a particular thing at a time

- **Critical section:** piece of code that only one thread can execute at once
-

Lock: prevent someone from doing something

- Lock before entering critical section, before accessing shared data
- Unlock when leaving, after done accessing shared data
- Wait if locked (all synchronization involves waiting!)

Desirable Properties

- Correctness property
  - Someone buys if needed (liveness)
  - At most one person buys (safety)

Too Much Milk, Try #1

- Try #1: leave a note
- Both threads do this …

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

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

Lessons

- Solution is complicated
  - “obvious” code often has bugs
- Modern compilers/architectures reorder instructions
  - Making reasoning even more difficult
- Generalizing to many threads/processors
  - Even more complex: see Peterson’s algorithm

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

- Lock::acquire
  - wait until lock is free, then take it, atomically
- 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 or fairness)

Atomicity

- All-or-nothing
- In our context:
  - Set of instructions that are executed as a group OR
  - System will ensure that this appears to be so

Question: Why only Acquire/Release

- Suppose we add a method to a lock, to ask if the lock is free. Suppose it returns true. Is the lock:
  - Free?
  - Busy?
  - Don’t know?

- Very risky!

```
if (test lock)
  acquire …
```

Too Much Milk, #4

Locks allow concurrent code to be much simpler:

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

Lock Example: Malloc/Free

```
char *malloc (n) {
    heaplock.acquire();
    p = allocate memory
    heaplock.release();
    return p;
}
```

```
void free(char *p) {
    heaplock.acquire();
    put p back on free list
    heaplock.release();
}
```

Synchronization

Chapter 5 OSPP

Part II

Example: Bounded Buffer

```
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

Condition Variables

- Waiting inside a critical section
  - Called only when holding a lock

- Wait: atomically release lock and relinquish processor
  - Reacquire the lock when wakened
- Signal: wake up a waiter, if any
- Broadcast: wake up all waiters, if any

Example: Bounded Buffer

```
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();
}
```

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

empty/full are condition variables

Condition Variable Design Pattern

```
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())
       cv.signal(&lock);

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

not all impls require

Pre/Post Conditions

- What is state of the bounded buffer at lock acquire?
  - front <= tail
  - front + MAX >= tail
- These are also true on return from wait
- And at lock release
- Allows for proof of correctness

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 (i.e. block), then release?
  - What if release, then wait (i.e. block)?

Condition Variables, cont'd

- 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

Spurious Wakeup

- Thread can be woken up “prematurely”
  - Unclear when exactly this can ever happen?
  - E.g. signal arrives when holding a user level lock …
- Postels Law
- Assumption of spurious wakeups forces thread to be *conservative in what it does*: set condition when notifying other threads, and *liberal in what it accepts*: check the condition upon any return
- Java claims this is possible!

Structured Synchronization

- 1. Identify objects or data structures that can be accessed by multiple threads concurrently

- 2. Add locks to object/module
  - Grab lock on start to every method/procedure
  - Release lock on finish

- 3. If need to wait
  - `while(needToWait()) { condition.Wait(lock); }`
  - Do not assume when you wake up, signaller just ran

- 4. If do something that might wake someone up (hint)
  - Signal or Broadcast

- 5. Always leave shared state variables in a consistent state
  - When lock is released, or when waiting

Mesa vs. Hoare semantics

- Mesa
  - Signal puts waiter on ready list
  - Signaller keeps lock and processor

- Hoare
  - Signal gives processor and lock to waiter
  - When waiter finishes, processor/lock given back to signaller

FIFO Bounded Buffer
(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();
}
```

Pitfalls

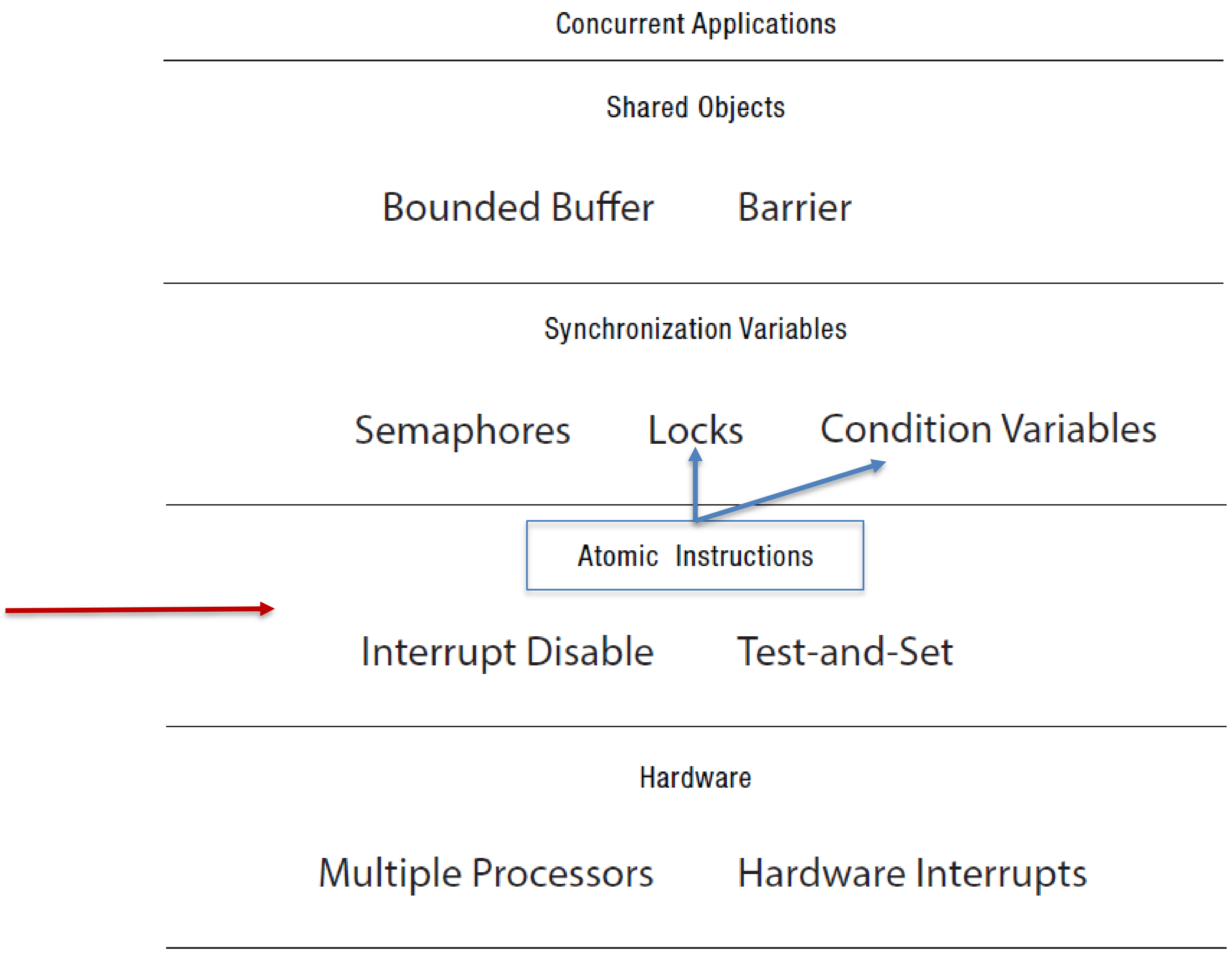
Common Case Rules

Synchronization

Chapter 5 OSPP

Part III

Implementing Synchronization



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

Implementing Synchronization

Take 1: using memory load/store

- See too much milk solution/Peterson’s algorithm

Take 2:

```
Lock::acquire()
   { disable interrupts }
Lock::release()
   { enable interrupts }
```

Two variations

Limitations

- Keep code short
- Trust the kernel to do this
- User threads: not so much
- Multiprocessors? Problem

- Spin or Block?
  - If lock is busy on a uniprocessor, why should acquire keep trying?

Lock Implementation, 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();
}
```

Why only switch in acquire?

If we suspend with interrupts turned off, what must be true?

Multiprocessor

- Interrupts won't work on a multiprocessor
- Read-modify-write instructions: h/w support
  - Atomically read a value from memory, operate on it, and then write it back to memory
  - + Can be called from user code
  - Intervening instructions prevented in hardware
- Examples
  - Test and set
  - Compare and swap
- Any of these can be used for implementing locks and condition variables!
- Since we cannot disable interrupts, there must be some amount of busy-waiting

Spinlocks

A spinlock is a lock where the processor waits in a loop for the lock to become free

- Assumes lock will be held for a short time
- Used to protect the CPU scheduler and to implement locks

```
Spinlock::Spinlock() { lockValue = FREE; }

Spinlock::acquire() {
  // TSL returns old value, sets new value to BUSY as a side-effect
  while (testAndSet(&lockValue) == BUSY);  }
     ;

Spinlock::release() { lockValue = FREE; }
```

How many spinlocks?

- Various data structures to protect
  - Protect user data A: use Lock X
  - Protect Lock X internals
  - Protect List of threads ready to run
- One spinlock
- Bottleneck!
- Instead:
  - Want higher-level lock to block
  - One spinlock per lock to protect access to lock internal state
  - One spinlock for the scheduler ready list

Lock Implementation, Multiprocessor

```
Lock::acquire() {
    disableInterrupts();
    spinLock.acquire();
    if (value == BUSY) {
        waiting.add(myTCB);
        suspend(&spinLock);
    } else {
        value = BUSY;
    }
    spinLock.release();
    enableInterrupts();
}
```

why do I pass spinLock?

```
Lock::release() {
    disableInterrupts();
    spinLock.acquire();
    if (!waiting.Empty()) {
        next = waiting.remove();
        scheduler->makeReady(next);
    } else {
        value = FREE;
    }
    spinLock.release();
    enableInterrupts();
}
```

Is this lock implemented in kernel or user space?

Why disable ints?

Lock Implementation, Multiprocessor

```
Sched::suspend(SpinLock ∗lock) {
    TCB ∗next;

    disableInterrupts();
    schedSpinLock.acquire();
    lock−>release();
    myTCB−>state = WAITING;
    next = readyList.remove();
    thread_switch(myTCB, next);
    myTCB−>state = RUNNING;
    schedSpinLock.release();
    enableInterrupts();
}
```

```
Sched::makeReady(TCB ∗thread) {

    disableInterrupts ();
    schedSpinLock.acquire();
    readyList.add(thread);
    thread−>state = READY;
    schedSpinLock.release();
    enableInterrupts();
}
```

next_thread needs to release schedSpinLock

Lock Implementation, Linux

- Most locks are free most of the time
  - Why?
  - Kernel and good programmers keep critical sections short!
  - Linux implementation takes advantage of this fact
- Fast path (common case)
  - If lock is FREE, and no one is waiting, two instructions to acquire the lock: no spinlock or disabling interrupts
  - If no one is waiting, two instructions to release the lock
  - load/store solution ~ no more milk
- Slow path
  - If lock is BUSY or someone is waiting, use multiprocessor version

Lock Implementation, Linux

```
struct mutex {
 /∗ 1: unlocked ; 0: locked;
    negative : locked,
    possible waiters ∗/
 atomic_t count;
 spinlock_t wait_lock;
 struct list_head wait_list;
};
```

```
// atomic decrement
// %eax is pointer to lock->count

lock decl (%eax)
jns 1f // jump if not signed
          // (i.e. if value is now 0)
call slowpath_acquire
1:
```

Semaphores

- Please look at them
- They are more for historical reasons as CVs are the synchronization of choice
- Rarely better: Ex. P 250