CS 423
Operating System Design: Synchronization

Professor Adam Bates
Fall 2018
Goals for Today

• **Learning Objectives:**
  • Understand different primitives for synchronization at the operating system layer

• **Announcements:**
  • C4 weekly summaries! **Due Friday (any time zone)**
  • MP1 is out! **Due Feb 20 (any time zone)**
  • CS Instructional Cloud is back online

**Reminder:** Please put away devices at the start of class
static __always_inline unsigned long __must_check

copy_to_user(void __user *to, const void __user *from, unsigned long n)
{
    if (likely(check_copy_size(from, n, true)))
        n = _copy_to_user(to, from, n);
    return n;
}

#ifdef CONFIG_COMPAT
static __always_inline unsigned long __must_check

copy_in_user(void __user *to, const void __user *from, unsigned long n)
{
    might_fault();
    if (access_ok VERIFY_WRITE, to, n) && access_ok VERIFY_READ, from, n))
        n = raw_copy_in_user(to, from, n);
    return n;
}
#endif
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 changes when re-run program

• Compiler/hardware instruction reordering

• Multi-word operations are not atomic
Can this panic?

Thread 1

p = someComputation();
pInitialized = true;

Thread 2

while (!pInitialized)
    ;
q = someFunction(p);
if (q != someFunction(p))
    panic
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
<table>
<thead>
<tr>
<th>Time</th>
<th>Person A</th>
<th>Person B</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:35</td>
<td>Leave for store.</td>
<td>Leave for store.</td>
</tr>
<tr>
<td>12:40</td>
<td>Arrive at store.</td>
<td>Arrive at store.</td>
</tr>
<tr>
<td>12:45</td>
<td>Buy milk.</td>
<td>Buy milk.</td>
</tr>
<tr>
<td>12:50</td>
<td>Arrive home, put milk away.</td>
<td>Arrive home, put milk away.</td>
</tr>
<tr>
<td>12:55</td>
<td></td>
<td>Oh no!</td>
</tr>
</tbody>
</table>
Too Much Milk!

SOLUTION

Make your own oat milk at home

srsly tho — https://minimalistbaker.com/make-oat-milk/
**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!)
• Correctness property
  – Someone buys if needed (liveness)
  – At most one person buys (safety)

• Try #1: leave a note
  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
Thread A	Thread B
leave note A
while (note B) // X
  do nothing;
if (!milk)
  buy milk;
remove note A
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
  – “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
Locks

• 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)
Why only Acquire/Release?

Why can’t we have an “Ask if Lock is Free” function?
Locks allow concurrent code to be much simpler:

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

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!
if (p == NULL) {
    lock.acquire();
    if (p == NULL) {
        p = newP();
    }
    lock.release();
}

use p->field1

newP() {
    p = malloc(sizeof(p));
    p->field1 = ...
    p->field2 = ...
    return p;
}
tryget() {
    item = NULL;
    lock.acquire();
    if (front < tail) {
        item = buf[front % MAX];
        front++;
    }
    lock.release();
    return item;
}

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

tryput(item) {
    lock.acquire();
    if ((tail - front) < size) {
        buf[tail % MAX] = item;
        tail++;
    }
    lock.release();
}
• 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?
Condition Variables

• 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
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();
    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
• 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
methodThat Waits() {
    lock.acquire();
    // Pre-condition: State is consistent

    // Read/write shared state

    while (!testSharedState()) {
        cv.wait(&lock);
    }
    // WARNING: shared state may have changed! But
    // testSharedState is TRUE
    // and pre-condition is true

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

methodThat Signals() {
    lock.acquire();
    // Pre-condition: State is consistent

    // Read/write shared state

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

    // NO WARNING: signal keeps lock

    // Read/write shared state
    lock.release();
}
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
  ```cpp
  while (needToWait()) {
    condition.Wait(lock);
  }
  ```

- Simplifies implementation
  - Of condition variables and locks
  - Of code that uses condition variables and locks
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
  • Nested signals possible!
FIFO Bounded Queue

(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
FIFO Bounded Queue

(Mesa Semantics)

• Create a condition variable for every waiter
• Queue condition variables (in FIFO order)
• Signal picks the front of the queue to wake up
• CAREFUL if spurious wakeups!

• Easily extends to case where queue is LIFO, priority, priority donation, …
  • With Hoare semantics, not as easy
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
• Never spin in sleep()
Implementing Synchronization

Concurrent Applications

Semaphores  Locks  Condition Variables

Interrupt Disable  Atomic Read/Modify/Write Instructions

Multiple Processors  Hardware Interrupts
Implementing Synchronization

• Take 1: using memory load/store
  • See too much milk solution/Peterson’s algorithm

• Take 2:
  • Lock::acquire()
  • Lock::release()
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();
}