CS 423
Operating System Design: Synchronization

Professor Adam Bates
Fall 2018
Goals for Today

• Learning Objectives:
  • Dive yet further into concurrency and threading

• Announcements:
  • MP1 available on Compass2G. **Due February 19th!**
  • There will be an MP1 Walkthrough this Monday.
  • MP0 grading complete, will post to Compass this weekend.

**Reminder**: Please put away devices at the start of class
• 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
## Too Much Milk!

<table>
<thead>
<tr>
<th>Time</th>
<th>Activity Description</th>
<th>Time</th>
<th>Activity Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:40</td>
<td>Person A: Arrive at store.</td>
<td>12:45</td>
<td>Person B: Leave for store.</td>
</tr>
<tr>
<td>12:45</td>
<td>Person A: Buy milk.</td>
<td>12:50</td>
<td>Person B: Arrive at store.</td>
</tr>
<tr>
<td>12:55</td>
<td>Person A: Arrive home, put milk away.</td>
<td>1:00</td>
<td>Person B: Arrive home, put milk away.</td>
</tr>
<tr>
<td></td>
<td>Person A: Oh no!</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**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
    }
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
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
  – “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
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

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

```java
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;
}
Ex: Lock 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
• 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?