CS 423
Operating System Design: Synchronization

Tianyin Xu

* Thanks for Prof. Adam Bates for the slides.
Please post the topic you’d like me to chat about in the first 10 minutes on Piazza!
Thanks YiFei and Haoqing and many others who are helping others on Piazza!
(I promise I will buy you beers after I come back!)
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

\[
p = \text{someComputation}(); \\
p\text{Initialized} = \text{true};
\]

Thread 2

\[
\text{while} \ (\!\!p\text{Initialized}) \\
\quad ; \\
q = \text{someFunction}(p); \\
\text{if} \ (q \neq \text{someFunction}(p)) \\
\quad \text{panic}
\]
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>Person A</th>
<th>Person B</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:35</td>
<td>Leave for store.</td>
<td></td>
</tr>
<tr>
<td>12:45</td>
<td>Buy milk.</td>
<td>Leave for store.</td>
</tr>
<tr>
<td>12:50</td>
<td>Arrive home, put milk away.</td>
<td>Arrive at store.</td>
</tr>
<tr>
<td>12:55</td>
<td></td>
<td>Buy milk.</td>
</tr>
<tr>
<td>1:00</td>
<td></td>
<td>Arrive home, put milk away. 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
    }
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
  – “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)
Locks allow concurrent code to be much simpler:

```java
lock.acquire();
if (!milk)
    buy milk
lock.release();
```
Ex: Lock Malloc/Free

```c
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!
Ex: Thread-Safe Bounded Queue

```c
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?
Implementing Locks

• 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();
}
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
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
methodThatWaits() {
    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();
}

methodThatSignals() {
    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
  ```
  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)

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

```java
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
  - \[ \text{while(needToWait()) \{ condition.Wait(lock); \}} \]
  - 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