Shared Memory Synchronization

Lecture 8
March 6, 2025

To Dos

Reading for next time

Program 3 submission today

Questions about program 3?

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

Question: What is the outcome of this code?

x = 0

Thread 1	Thread 2
`x=x+1;`	`x=x+1;`
`print x;`	`print x;`

Question: Will `p` have result of `someComputation()` before `someFunction()`?

Thread 1

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

Thread 2

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

Why Reordering?

- Why do compilers reorder instructions?
  - Efficient code generation requires analyzing control/data dependency
  - Problem: If variables can spontaneously change (due to threads), 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 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!)

Definitions

- Safety – program never enters a bad state
- Liveness – program eventually enters a good state

Too Much Milk, Try #1

- 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
```

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:

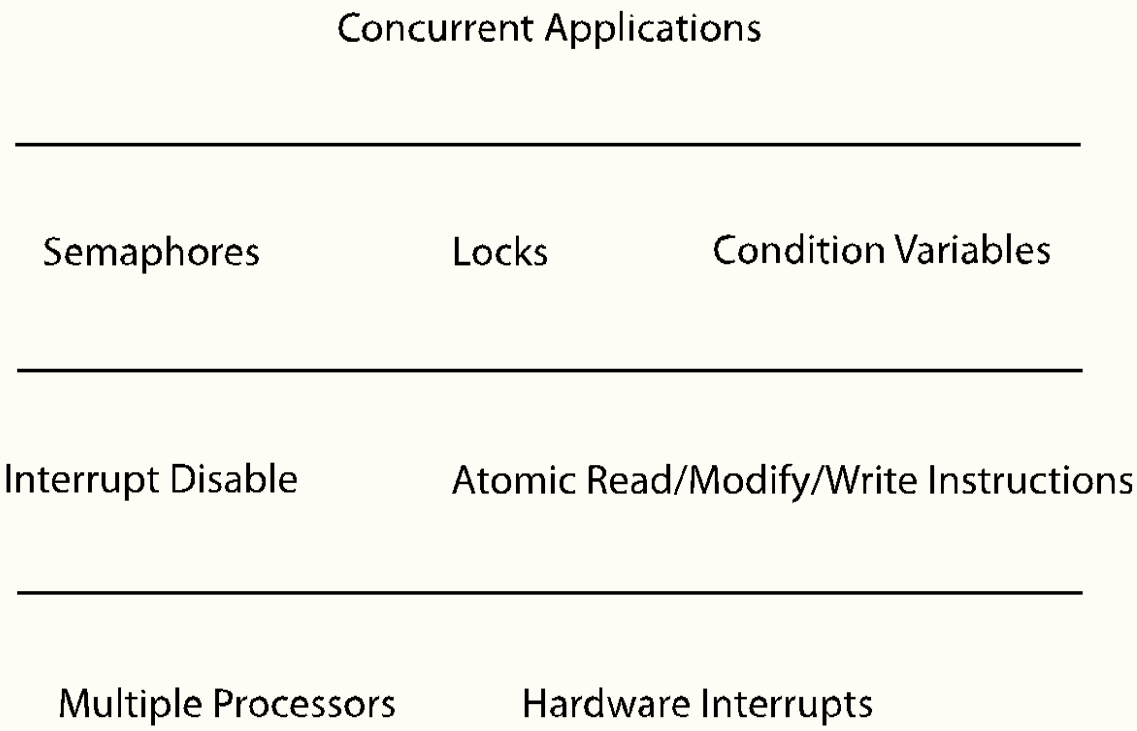
(i) Safe for me to buy

(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
- Inefficient due to **busy waiting**
  - Generalizing to many threads/processors even more complex and still inefficient

Implementing Synchronization



Locks

- Lock has two states: `free, busy`
- `Lock::acquire()`
  - wait until lock is `free`, then take it (make `busy`)
  - done atomically
- `Lock::release()`
  - release lock, waking up one thread waiting for it which will now have the lock
- Properties:
  1. At most one lock holder at a time (safety/mutual exclusion)
  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/bounded waiting)

Too Much Milk, #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
- Never access shared data without lock
  - Danger!

Example: Bounded Buffer

```
Class BoundedBuffer {
 private:
   Lock lock;

   int buf[MAX];
   int front;
   int tail;
 public:
   int tryget();
   void tryput(int item);
}
```

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)< MAX){
      buf[tail % MAX]=item;
      tail++;
    }
    lock.release();
}
```

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

Question

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

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

Condition Variables

- Waiting inside a critical section
  - Called only when holding a lock
  - Waiting for another thread to do something
  - Alternative to polling/busy-waiting
- `wait(lock)`: atomically release lock and relinquish processor
  - Reacquires the lock when wakened
- `signal(lock)`: wake up a waiter, if any
- `broadcast(lock)`: wake up all waiters, if any

Shared Object

- Contains shared data
- Contains lock
- Contains 0+ condition variables
  - Can wait for some shared data state to change while not preventing other threads from getting lock

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

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

Example: Bounded Buffer

```
Class BoundedBuffer {
 private:
   Lock lock;
   ConditionVariable full;
    ConditionVariable empty;

   int buf[MAX];
   int front;
   int tail;
 public:
   int get();
   void put(int item);
}
```

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

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`

Pre/Post Conditions

```
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
```

```
methodThatSignals() {
    lock.acquire();
    // Pre-condition: State is
  consistent

    // Read/write shared state


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

    // NO WARNING: signal keepslock


    // 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
  - Has queue of waiting threads
  - If signal when no one is waiting, no op
  - If wait before signal, waiter wakes up on signal
- Wait atomically releases lock
  - What if release, then wait?

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

Structured Synchronization

- 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, signaler just ran
- If you 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