Synchronization (1)

April 2, 2018

Byung-Gon Chun

Acknowlegments. Some slides and/or picture in the following are adapted from UW, Columbia, and UC Berkeley class slides

Why Synchronization?

- If a program has independent threads that operate on completely separate subsets of memory, we can reason about each thread separately.

- But, most multi-threaded programs have both per-thread state and shared state. Cooperating threads read and write shared state

  => Writing correct multi-threaded programs becomes much more difficult.

- The sequential model of reasoning does not work in programs with cooperating threads

Why Does Not The Sequential Model of Reasoning Work?

- Program execution depends on the possible interleavings of threads' access to shared state
  - Two threads write to the same variable; which one should win?

- Program execution can be nondeterministic
  - Different runs of the same program may produce different results
  - Heisenbugs: bugs that disappear or change behavior when you try to examine them

- Compilers and processor hardware can reorder instructions

Question: Can this panic?

Thread 1

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

Thread 2

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

Roadmap

- Synchronization Challenges
- Structuring Shared Objects
- Locks: Mutual Exclusion
- Condition Variables: Waiting for a Change
- Designing and Implementing Shared Objects
- Case Studies
- Implementing Synchronization Primitives
- Semaphore
- Linux kernel synchronization
- Multiprocessor Lock Performance
- Lock Design Patterns
- Lock Contention
- Multi-Object Atomicity
- Deadlock
- Non-Blocking Synchronization

Challenges

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

- Can be very bad
  - "non-deterministic:" don't know what the output will be, and it is likely to be different across runs
  - Hard to detect: too many possible schedules
  - Hard to debug: "heisenbug," debugging changes timing so hides bugs (vs "bohr bug")

Simple Cooperating-Threads Program 1

- A program with two threads that do the following

Thread A	Thread B
x = 1;	x = 2;

What are the possible final values of x?

Simple Cooperating-Threads Program 2

- A program with two threads that do the following. Initially y = 12.

Thread A	Thread B
`x = y + 1;`	`y = y * 2;`

What are the possible final values of x?

Simple Cooperating-Threads Program 3

- A program with two threads that do the following. Initially x = 0.

Thread A	Thread B
x = x + 1;	x = x + 2;

What are the possible final values of x?

Banking Example

```
int balance = 0;
int main() {
        pthread_t t1, t2;
        pthread_create(&t1, NULL, deposit, (void*)1);
        pthread_create(&t2, NULL, withdraw, (void*)2);
        pthread_join(t1, NULL);
        pthread_join(t2, NULL);
        printf("all done: balance = %d\n", balance);
        return 0;
}
```

```
void* deposit(void *arg)
{
        int i;
        for(i=0; i<1e7; ++i)
                ++ balance;
}
```

```
void* withdraw(void *arg)
{
        int i;
        for(i=0; i<1e7; ++i)
                -- balance;
}
```

Results of the banking example

```
$ gcc –Wall –lpthread –o bank bank.c
$ bank
all done: balance = 0
$ bank
all done: balance = 140020
$ bank
all done: balance = -94304
$ bank
all done: balance = -191009
```

Why?

A closer look at the banking example

```
$ objdump –d bank
...
08048464 <deposit>:
...                              // ++ balance
8048473: a1 80 97 04 08          mov 0x8049780,%eax
8048478: 83 c0 01                add  $0x1,%eax
804847b: a3 80 97 04 08          mov %eax,0x8049780
...

0804849b <withdraw>:
...                              // -- balance
80484aa: a1 80 97 04 08          mov 0x8049780,%eax
80484af: 83 e8 01                sub  $0x1,%eax
80484b2: a3 80 97 04 08          mov %eax,0x8049780
...
```

One possible schedule

CPU 0

CPU 1

balance: 0

mov 0x8049780,%eax

eax: 0

add $0x1,%eax

eax: 1

mov %eax,0x8049780

balance: 1

mov 0x8049780,%eax

eax: 1

sub $0x1,%eax

eax: 0

mov %eax,0x8049780

balance: 0

One deposit and one withdraw, balance unchanged. Correct.

Another possible schedule

CPU 0

CPU 1

balance: 0

```
mov 0x8049780,%eax
```

eax: 0

```
add  $0x1,%eax
```

eax: 1

```
                              mov 0x8049780,%eax
```

eax: 0

```
mov %eax,0x8049780
```

balance: 1

```
                              sub  $0x1,%eax
```

eax: -1

```
                              mov %eax,0x8049780
```

balance: -1

One deposit and one withdraw, balance becomes -1. Wrong!

Simple Cooperating-Threads Program 3

- A program with two threads that do the following. Initially x = 0.

Thread A	Thread B
`x = x + 1;`	`x = x + 2;`

What are the possible final values of x?

Interleaving 1

```
load r1, x
add r2, r1, 1
store x, r2
                    load r1, x
                    add r2, r1, 2
                    store x, r2
```

Interleaving 2

```
load r1, x
                    load r1, x
add r2, r1, 1
                    add r2, r1, 2
store x, r2
                    store x, r2
```

Interleaving 3

```
load r1, x
                    load r1, x
add r2, r1, 1
                    add r2, r1, 2
                    store x, r2
store x, r2
```

Sharing a Refrigerator

- Two room mates who share a refrigerator and who make sure the refrigerator is always well stocked with milk.

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!

Simplifying Assumptions to Solve Too Much Milk

- Instructions are executed in exactly the order written.
  I.e., neither the compiler nor the architecture reorders instructions

Too Much Milk, Try #1

- Correctness property
  - Someone buys if needed (liveness)
  - At most one person buys (safety)
- Try #1: leave a note

```
if (milk == 0)
    if (note == 0) {
        note = 1; // leave note
        milk++; // buy milk
        note = 0; // remove note
    }
```

Too Much Milk, Try #1

Thread A

```
if (milk == 0) {




        if (note == 0) {
                note = 1;
                milk++;
                note = 0;
        }
}
```

Thread B

```
if (milk==0) {
        if (note == 0) {
                note = 1;
                milk++;
                note = 0;
        }
}
```

Too Much Milk, Try #2

Thread A

```
noteA = 1; // leave note A
if (noteB==0) { // if no note A1
   if (milk==0) // if no milk A2
      milk++    // buy milk  A3
}
noteA = 0; // remove note A
```

Thread B

```
noteB = 1; // leave note B
if (noteA==0) { // if no note  B1
   if (milk==0)  // if no milk    B2
      milk++;    // buy milk     B3
}                // B4
noteB=0;         // remove note B
```

Too Much Milk, Try #3

Thread A

```
noteA=1;      // leave note A
while (noteB == 1) // X: wait for no noteB
    ;             // spin
if (milk==0)    // if no milk M
    milk++;     // buy milk
noteA = 0;    // remove note A
```

Thread B

```
noteB = 1;        // leave note B
if (noteA==0) {   // Y: if no note A
    if (milk==0)    // if no milk
        milk++;       // buy milk
}
noteB = 0;        // remove note B
```

Too Much Milk, Try #3

Thread A

```
noteA=1;        // leave note A
while (noteB == 1) // X: wait for no noteB
    ;           // spin
if (milk==0)    // if no milk M
    milk++;     // buy milk
noteA = 0;      // remove note A
```

Thread B

```
noteB = 1;          // leave note B
if (noteA==0) {     // Y: if no note A
    if (milk==0)    // if no milk
        milk++;     // buy milk
}
noteB = 0;          // remove note B
```

At Y: if no Note A, safe for B to buy (means A hasn't started yet)
if note A, A is either buying, or waiting for B to quit,
so ok for B to quit

Too Much Milk, Try #3

Thread A

```
noteA=1;       // leave note A
while (noteB == 1) // X: wait for no noteB
   ;           // spin
if (milk==0)    // if no milk M
   milk++;      // buy milk
noteA = 0;     // remove note A
```

Thread B

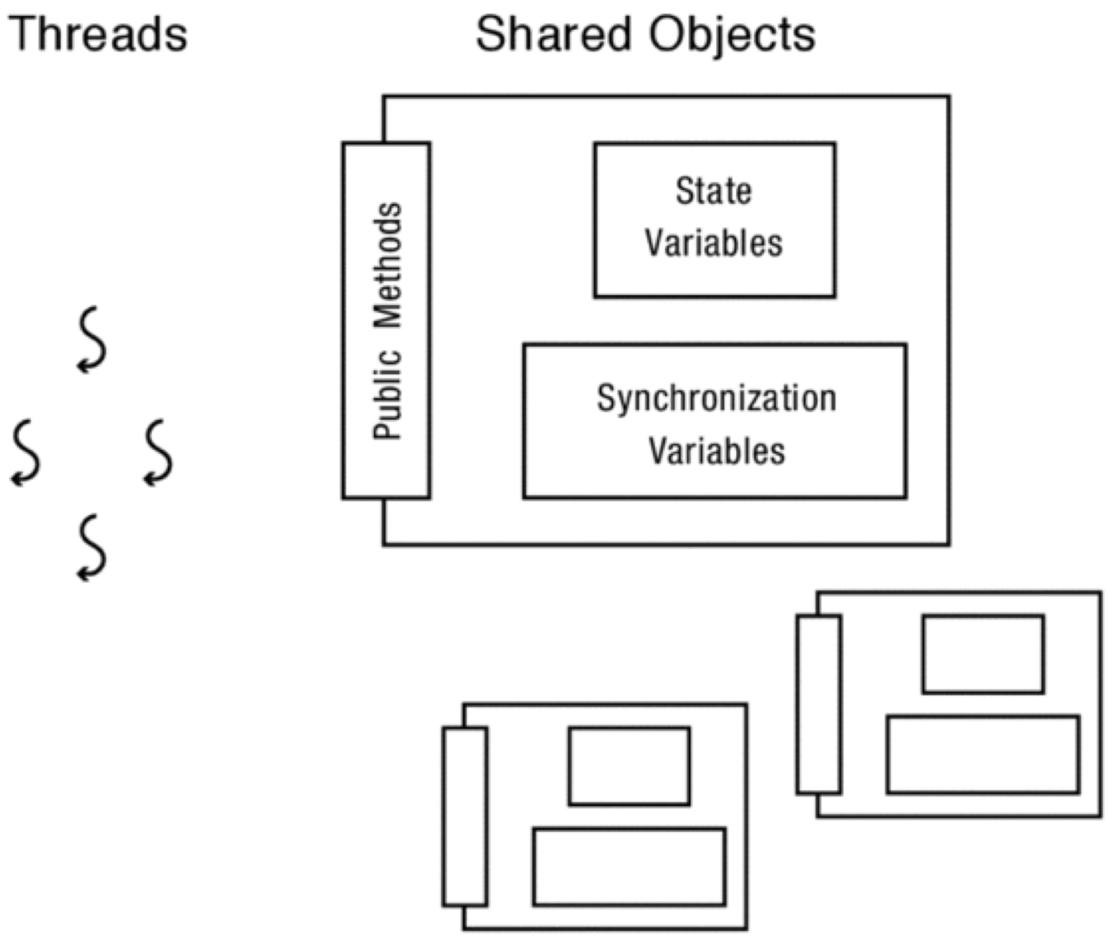
```
noteB = 1;        // leave note B
if (noteA==0) {   // Y: if no note A
   if (milk==0)   // if no milk
      milk++;     // buy milk
}
noteB = 0;        // remove note B
```

At X: if no note B, safe to buy
if note B, don't know. A hangs around. Either:
if B buys, done
if B doesn't buy, A will.

Discussion: Are You Satisfied with the Solution?

- Solution is complicated
  - “obvious” code often has bugs
- Solution is inefficient
  - While Thread A is waiting, it is busy-waiting and consuming CPU resources
- The solution may fail if the compiler or hardware reorders instructions
  - The limitation can be addressed by using memory barriers. This makes reasoning even more difficult
- Generalizing to many threads/processors
  - Even more complex: see Peterson’s algorithm

Structuring Shared Objects



Implementing Shared Objects

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 Shared Objects

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 is a synchronization variable that provides mutual exclusion – when one thread holds a lock, no other thread can hold it
- A program associates each lock with some subset of shared state and requires a thread to hold the lock when accessing that state.
- Mutual exclusion greatly simplifies reasoning about programs because a thread can perform an arbitrary set of operations while holding a lock, and those operations appear to be atomic to other threads

Locks: API

- Two methods: Lock::acquire and Lock::release

- A lock can be in one of two states: BUSY or FREE
- A lock is initially in the FREE state
- Lock::acquire waits until the lock is FREE and then takes it (**atomically** makes the lock BUSY)
- Lock::release makes the lock FREE. If there are pending acquire operations, this state change causes one of them to proceed

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?

Too Much Milk, #4

Locks allow concurrent code to be much simpler:

```
lock.acquire();
if (milk==0)
    milk++; // buy milk
lock.release();
```

Lock Example: Malloc/Free

```
char *malloc (n) {
    p = allocate memory
    return p;
}
```

```
void free(char *p) {
    put p back on free list
}
```

How to implement a thread-safe memory allocator?

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();
}
```

Fixing the Banking Example

```
void* deposit(void *arg)
{
    int i;
    for(i=0; i<1e7; ++i) {
        ++ balance;
    }
}
```

```
void* withdraw(void *arg)
{
    int i;
    for(i=0; i<1e7; ++i) {
        -- balance;
    }
}
```

Fixing the Banking Example

```
pthread_mutex_t lock;



void* deposit(void *arg)
{
    int i;
    for(i=0; i<1e7; ++i) {
        pthread_mutex_lock(&lock);
        ++ balance;
        pthread_mutex_unlock(&lock);
    }
}
```

```
void* withdraw(void *arg)
{
    int i;
    for(i=0; i<1e7; ++i) {
        pthread_mutex_lock(&lock);
        -- balance;
        pthread_mutex_unlock(&lock);
    }
}
```