CSE 451: Operating Systems
Spring 2022

Module 6
Synchronization

John Zahorjan
Temporal relations

• Machine instructions executed by a single thread are totally ordered
  • $A < B < C < \ldots$
  • This is called “program order”
  • (Interesting aside: actually, that isn’t necessarily true, physically. To go fast, each core tries to execute many instructions at once, possibly out of order. However, it does so in a way that it has the same effect as totally ordered execution. Usually.)

• Unless there is explicit synchronization, instructions executed by distinct threads must be considered unordered
  • Not $X < X’$, and not $X’ < X$

• Not $X < X’$ and not $X’ < X$ is simultaneous
  • unordered
  • at the same time

• If $X$ and $X’$ access the same memory location, and at least one of them is a write, it is a “data race”
Example

```
A

B

C

main()

pthread_create()

sub()

A'

B'

Y-axis is "time"

Could be one core, could be multiple cores.

• A < B < C
• A' < B'
• A < A'
• C == A'
• C == B'
```
Critical Sections / Mutual Exclusion / Locks

• Sequences of instructions that may get incorrect results if executed simultaneously are called critical sections
• (We also use the term race condition to refer to a situation in which the results depend on timing)
• Mutual exclusion means “not simultaneous”
  • Either A < B or B < A
  • We don’t care which
• Forcing mutual exclusion between two critical section executions is sufficient to ensure correct execution – guarantees ordering
• One way to guarantee mutually exclusive execution is using locks
Critical sections

T1 T2 T1 T2 T1 T2

Possibly incorrect

Correct

Correct

How many cores are in use here?
When do critical sections arise?

• One common pattern:
  • read-modify-write of
    • a shared value (variable)
      • in code that can be executed concurrently

(Note: There may be only one copy of the code (e.g., a procedure), but it can be executed by more than one thread at a time)

• Shared variables
  • Globals and heap-allocated variables
  • to keep your sanity, follow the convention of NOT sharing local variables (which are on the stack) across threads

(Note: Never give a reference to a stack-allocated (local) variable to another thread, unless you’re superhumanly careful ...)

• Can you pass a local as an argument to a function?
Example: buffer management

• In this example, one thread puts data into a buffer that another thread reads from

• Shared resource: buffer data structure

• Read-modify-write: each slot is either empty or free; operations get() and put() both read and modify a slot status
Why use threads in that example?
The classic shared bank account example

• Suppose we have to implement a function to withdraw money from a bank account:

```c
int withdraw(account, amount) {
    int balance = get_balance(account);  // read
    if (balance >= amount) {
        balance -= amount; // modify
        put_balance(account, balance); // write
    spit out cash;
    }
}
```

• Now suppose that you and your partner share a bank account with a balance of $500.
• What happens if you both go to separate ATM machines, and simultaneously withdraw $50 from the account?
• Assume the bank’s application is multi-threaded, and...

• A random thread is assigned a transaction when that transaction is submitted

```c
int withdraw(account, amount) {
    int balance = get_balance(account);
    if (balance >= amount) {
        balance -= amount;
        put_balance(account, balance);
        spit out cash;
    }
}
```
Interleaved schedules

• The problem is that the execution of the two threads can be interleaved, assuming preemptive scheduling:

```c
balance = get_balance(account);
balance -= amount;
```

• What’s the account balance after this sequence?
  • Who’s happy, the bank or you?
  • Suppose the two of you make simultaneous deposits?

• How often is this sequence likely to occur?

• Can this happen if there is only one physical core?
Other Execution Orders

• Which interleavings are ok? Which are not?

```c
int withdraw(account, amount) {
    int balance = get_balance(account);
    if (balance >= amount) {
        balance -= amount;
        put_balance(account, balance);
        spit out cash;
    }
}
```

```c
int withdraw(account, amount) {
    int balance = get_balance(account);
    if (balance >= amount) {
        balance -= amount;
        put_balance(account, balance);
        spit out cash;
    }
}
```
Correct critical section requirements

• Correct critical sections have the following requirements

1. **mutual exclusion**
   • at most one thread is in the critical section
   • Ridiculous solution so far: Don’t let any code execute critical section, ever

2. **progress**
   • if thread T is outside the critical section, then T cannot prevent thread S from entering the critical section
   • Ridiculous solution so far: Let there be one “chosen thread” that is allowed to execute critical sections, but no others
     • *That actually isn’t always a bad idea...*

3. **bounded waiting (no starvation)**
   • if thread T is waiting on the critical section, then T will eventually enter the critical section
     • assumes threads eventually leave critical sections

4. **performance**
   • the overhead of entering and exiting the critical section is small with respect to the work being done within it (related to granularity)
   • *High overhead solution: all threads wanting to enter critical section contact a server and the server replies when it’s your turn to enter*
Synchronization mechanisms for building critical sections

- **Locks (spinlocks)**
  - primitive, minimal semantics; used to build others

- **Mutexes** (blocking locks)

- **Semaphores**
  - basic, easy to get the hang of, somewhat hard to program with

- **Monitors**
  - higher level, “requires” language support, implicit operations
  - easier to program; Java “synchronized()” as an example

- **Messages**
  - simple model of communication and synchronization based on (atomic) transfer of data across a channel
  - direct application to distributed systems
Locking (Locks)

• Locking has two operations:
  • acquire(): obtain the right to enter the critical section
  • release(): give up the right to be in the critical section
    • (Note: terminology can vary: acquire/release, lock/unlock)

• acquire()/release() provide the four conditions required to be a critical section solution

• A lock is (usually) a memory object and code that supports those operations in a particular way (that we’ll see shortly)
Locks: Example

What happens during this time? (spinlock vs. mutex)
Acquire/Release

- Each threads pairs calls to `acquire()` and `release()`
  - between `acquire()` and `release()`, the thread holds the lock

- The `acquire()` call is the request.
  The return is the response indication that the caller now “owns” (holds) the lock
  - at most one thread can hold a lock at a time

- What happens if the calls aren’t paired (fail to call release)?

- What happens if the two threads acquire different locks?
  (I think that access to a particular shared data structure is mediated by lock A, and you think it’s mediated by lock B)

- Why is granularity of locking important
  - fine grained => not much work done between `acquire()` and `release()`
  - coarse grained => lots of work done between `acquire()` and `release()`
Using locks

```java
int withdraw(account, amount) {
    acquire(lock);
    balance = get_balance(account);
    if (balance >= amount) {
        balance -= amount;
        put_balance(account, balance);
    }
    release(lock);
    spit out cash;
}
```

- What happens when green tries to acquire the lock?
- Why is reading the balance inside the critical section?
- Why isn’t “spit out cash” inside the critical section?
  - Could it be put inside the critical section?
Roadmap ...

• Where have we just been?
  • Critical sections are a common property of concurrent/parallel code
  • Mutual exclusion is a mechanism to ensure a kind atomic execution of critical sections

• Where are we going?
  • Synchronization constructs provide the programmer with abstractions that address synchronization problems, like critical sections
  • The most primitive/fundamental abstraction is acquire()/release(): the lock
    • It can provide a solution if used correctly
    • It’s easy to mis-use it, though
  • “Higher level” synchronization abstractions provide additional semantics that can make them easier to use correctly, but usually at the cost of more overhead
  • The implementation of these higher level synchronization primitives often involves critical sections, so we layer the implementation (relying on the lock, say, for mutual exclusion)

• At the bottom of the layered implementations, it turns out we require some sort of hardware support
  • Software implementing acquire()/release “needs” to do a read-modify-write
  • Software can’t use itself to achieve that, so we need lower level support
  • So we “need” some atomic instruction that does at least two logically distinct things
    • Basically, there’s a read phase followed by a write phase
    • Done atomically
  • This hardware mechanism(s) are not intended to be utilized directly in user programs
    • They’re used to build software that implements somewhat higher abstractions that are used in user programs
Our First Primitives: Locks and Mutexes

What happens during this time?

1. Spinlock – keep using core while waiting
2. Mutex – give up core while waiting
Spinlocks

- A spinlock is a lock where the thread attempting acquire() “spins” (tries over and over without relinquishing its core)
- How do we implement spinlocks? Here’s one attempt:

```c
struct lock_t {
    int held = 0;
};

void acquire(lock) {
    while (lock->held);
    lock->held = 1;
}

void release(lock) {
    lock->held = 0;
}
```

- Why doesn’t this work?
  - where is the race condition?
  - does it work if there’s only one core?
Implementing spinlocks

• Problem is that implementation of spinlocks is itself a critical section
  • acquire/release must be atomic
    • atomic == executes as though it could not be interrupted
    • code that executes “all or nothing”

• Need help from the hardware

  1. atomic instruction
     • many instances of the instruction can be executed concurrently, because the hardware provides atomicity at the instruction level
     • test-and-set, compare-and-swap, ...

  2. disable interrupts
     • Terrible idea...
     • Used in xk...
     • Provides for atomic sequence of arbitrary instructions, when it works
Atomic Instruction: Test-and-Set

• CPU hardware provides the following operation as a single atomic instruction:

```c
bool test_and_set(bool *flag) {
    bool old = *flag;  // save value in a local (register)
    *flag = True;      // make sure value is True
    return old;        // return old value
}
```

• Remember, this is a single **atomic** instruction ...
  • *Remember, this is just one example of possible hardware support*
Implementing spinlocks using Test-and-Set

• So, to fix our broken spinlocks:

```c
struct lock {
    int held = 0;
}
void acquire(lock) {
    while(test_and_set(&lock->held));
}
void release(lock) {
    lock->held = 0;
}
```

• mutual exclusion? (at most one thread in the critical section)
• progress? (T outside cannot prevent S from entering)
• bounded waiting? (waiting T will eventually enter)
• performance? (low overhead?)
Lock instruction?

• Would a single atomic instruction whose semantics were the while loop shown on the last slide be “better” than just a test-and-set instruction?
  • The instruction would execute until it found atomically that the memory location had value 0 and had set it to 1?

• Any Pro’s?
• Any Con’s?
Reminder of use ...

```java
int withdraw(account, amount) {
    acquire(lock);
    balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
    release(lock);
    spit out cash;
}
```

- How could a thread spinning in acquire (that is, stuck in a test-and-set loop) yield its core?
  - voluntarily calls `yield() (spin-then-block lock)`
  - there’s an involuntary context switch (e.g., timer interrupt)

- When should a thread that has yielded the core be given a core again?
Problems with spinlocks

• Spinlocks work, but can be wasteful
  • if a thread is spinning on a lock, the thread holding the lock cannot make progress
    • You’ll spin for a scheduling quantum
    • (pthread_spin_t)

• Generally want to use spinlocks only as primitives to build higher-level synchronization constructs

• We’ll see later how to build blocking locks
  • But there is overhead – can be cheaper to spin
    • (pthread_mutex_t)

• Are there other “policy” choices (than spin and block)?
  • Who should make them?
    • pthread_spin_trylock()
A second approach: Disabling interrupts

```c
struct lock {
}
void acquire(lock) {
    cli();  // disable interrupts
}
void release(lock) {
    sti();  // reenable interrupts
}
```

What’s the key point about disabling interrupts?
Problems with disabling interrupts

• Available only to the kernel!
  • Can’t allow user-level to disable interrupts!

• Insufficient on a multicore!
  • Each core has its own interrupt mechanism

• “Long” periods with interrupts disabled can wreak havoc with devices!
  • “Stuff doesn’t work”

• Just as with spinlocks, you (would) want to use disabling of interrupts only when the duration of disabling is well understood (and short)
  • E.g., to build higher-level synchronization constructs
Summary

• Synchronization enforces temporal ordering constraints among instruction streams
  • Adding synchronization can eliminate races
• Synchronization can be provided by locks, semaphores, monitors, messages ...
• Spinlocks are a lowest-level mechanism
  • primitive in terms of semantics – error-prone
  • implemented by spin-waiting (crude) or by disabling interrupts (even cruder and doesn’t really work these days)
  • Make sense only when it’s “guaranteed” the lock will be released very soon
• Next...
  • Condition variables
    • Blocking as a concept/mechanism
  • Semaphores: synchronization variable
    • Importantly, they are implemented by blocking, not spinning
    • Locks can also be implemented in this way
  • Monitors: programming language support
    • are significantly higher level
    • utilize programming language support to reduce errors