Announcements

- Midterm this Friday
- Where?
  - Canvas quiz
- When?
  - You’ll have 50 minutes between 11:20 am and 12:30 pm on Friday
Temporal relations

• Machine instructions executed by a single thread are totally ordered
  • A < B < C < ...
  • *(Okay, we know they’re not, because that’s too slow. But however they’re executed it has the same effect as totally ordered execution, usually.)*

• Absent 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
Example

main()
A

pthread_create()
B

foo()
A'

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
Possibly incorrect

T1 T2
Correct

T1 T2
Correct
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, NOT local variables (which are on the stack)
  (Never give a reference to a stack-allocated (local) variable to another thread, unless you’re superhumanly careful …)
Example: buffer management

- In this example, one thread puts data into a buffer that another thread reads from.
- Shared resource: buffer
- Read-modify-write: each slot is either empty or free, and operations `get()` and `put()` both read the status and modify it.
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 $100.

• What happens if you both go to separate ATM machines, and simultaneously withdraw $15 from the account?
• Assume the bank’s application is multi-threaded
• A random thread is assigned a transaction when that transaction is submitted

```java
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:

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

balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
spit out cash;

put_balance(account, balance);
spit out cash;
```

• 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?
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;
    }
}
```
How About Now?

int xfer(from, to, amt) {
    withdraw( from, amt );
    deposit( to, amt );
}

• Morals:
  • Interleavings are hard to reason about
    • We make lots of mistakes
    • Control-flow analysis is hard for tools to get right
  • Identifying critical sections and ensuring mutually exclusive access is … “easier”

  • We’d like it to be easier still!
Another example

Moral?
Correct critical section requirements

• Correct critical sections have the following requirements
  • **mutual exclusion**
    • at most one thread is in the critical section
  • **progress**
    • Ridiculous solution so far: Don’t let any code execute critical section, ever
    • if thread T is outside the critical section, then T cannot prevent thread S from entering the critical section
  • **bounded waiting (no starvation)**
    • Ridiculous solution so far: Let there be one “chosen thread” that is allowed to execute critical sections, but no others
      • *That actually isn’t such a bad idea…*
    • if thread T is waiting on the critical section, then T will eventually enter the critical section
      • assumes threads eventually leave critical sections
  • **performance**
    • the overhead of entering and exiting the critical section is small with respect to the work being done within it
    • High overhead solution: all threads wanting to enter critical section contact a server and the server replies when it’s your turn to enter
Mechanisms for building critical sections

• Spinlocks (locks)
  • 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 with; 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
Locks

- A lock is a (memory) object with two operations:
  - `acquire()`: obtain the right to enter the critical section
  - `release()`: give up the right to be in the critical section

- `acquire()` prevents progress of the thread until the lock can be acquired

- (Note: terminology varies: acquire/release, lock/unlock)
Locks: Example

What happens during this time?
Acquire/Release

• Threads pair up calls to acquire() and release()
  • between acquire() and release(), the thread holds the lock
• acquire() does not return until the caller “owns” (holds) the lock
  • at most one thread can hold a lock at a time

• What happens if the calls aren’t paired (I acquire, but neglect to 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 => plenty of work done between acquire() and release()
Using locks

```c
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?
Roadmap ...

• Where we are eventually going:
  • The OS and/or the user-level thread package will provide some sort of efficient primitive for user programs to achieve mutual exclusion
    • for example, locks or semaphores, used with condition variables
  • There may be higher-level constructs provided by a programming language to help you get it right
    • for example, monitors – which utilize condition variables

• But somewhere, underneath it all, it turns out we require some sort of hardware support
  • Some “atomic instruction” that does at least two logically distinct things in a way that every other instruction operating on the same data executes either before or after, but not during
  • This hardware mechanism will not be utilized by user programs
  • It will be utilized in implementing somewhat higher abstractions for user programs
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 has critical sections, too!
  • the 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

• atomic instruction
  • test-and-set, compare-and-swap, ...

• atomic sequence of instructions, in some cases
  • just don’t let any other code run...
  • disable/reenable interrupts to prevent context switches
    • used in xk
Hardware Test-and-Set

- CPU hardware provides the following a single atomic instruction:

```c
bool test_and_set(bool *flag) {
    bool old = *flag;
    *flag = True;
    return old;
}
```

- 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?)
• How does a thread blocked on an “acquire” (that is, stuck in a test-and-set loop) yield the CPU?
  • voluntarily calls yield( ) (spin-then-block)
  • there’s an involuntary context switch (e.g., timer interrupt)
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)
A second approach: Disabling interrupts

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

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

• Insufficient on a multiprocessor!
  • Each processor has its own interrupt mechanism

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

• Just as with spinlocks, you want to use disabling of interrupts only to build higher-level synchronization constructs
  • Except maybe in xk...
Summary

• Synchronization enforces temporal ordering constraints
• 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)
• In our next exciting episode ...
  • semaphores are a slightly higher level abstraction
    • Importantly, they are implemented by blocking, not spinning
    • Locks can also be implemented in this way
  • monitors are significantly higher level
    • utilize programming language support to reduce errors