Threads & Synchronization

Arvind Krishnamurthy
Spring 2004

Outline

- Previous lectures: concurrency
  - threads vs. processes
  - how to implement threads?
- Next few lectures:
  - how to write multithreaded programs?
    - Main challenge: how to eliminate race conditions? how to synchronize?
    - Solutions: locks, semaphore, conditional variables, monitors, ...

Independent vs. Cooperating

- Independent threads
  - no state shared with other threads
  - deterministic --- input state determines result
  - reproducible
  - scheduling order does not matter
- Cooperating threads
  - shared state
  - non-deterministic
  - non-reproducible

Non-reproducibility and non-determinism means that bugs can be intermittent. This makes debugging really hard.

Why allow cooperating threads?

Computer programs at some level have to cooperate

- Share resources/information
  - one computer many users/programs
  - one bank balance, many ATMs
- Speedup
  - overlap I/O and computation
  - multiprocessors -- chop up programs into smaller pieces
- Modularity
  - chop large problem up into simpler pieces
  - For example: “latex foo.tex | spell | sort | uniq | wc”

Example: Shared counter

- Yahoo gets millions of hits a day. Uses multiple threads (on multiple processors) to speed things up.
- Simple shared state error: each thread increments a shared counter to track the number of hits today:

```c
... hits = hits + 1;
...```

- What happens when two threads execute this code concurrently?

Problem with shared counters

- One possible result: lost update!
  - `hits = 0`
  - `T1` read `hits (0)`
  - `T2` read `hits (0)`
  - `T1` `hits = 0 + 1`
  - `T2` `hits = 0 + 1`
  - `hits = 1`

- One other possible result: everything works.
  - Bugs are frequently intermittent. Makes debugging hard.
  - This is called “race condition”
Race conditions

- Race condition: timing dependent error involving shared state.
  - whether it happens depends on how threads are scheduled
- "Hard" because:
  - must make sure all possible schedules are safe. Number of possible schedule permutations is huge.

```java
if (n == stack_size) /* A */
return full; /* B */
stack[n] = v; /* C */
n = n + 1; /* D */
```

Some bad schedules:
- AA'CC'DD' → overwrites
- ACA'DC'D' → overflow
- How many???

Bugs are intermittent. Timing dependent = small changes (adding a print stmt, different machine) can hide bug.

Stack Race Conditions

```java
if (n == stack_size) /* A' */
return full; /* B' */
stack[n] = v; /* C' */
n = n + 1; /* D' */
```

Preventing race conditions: atomicity

- atomic unit = instruction sequence guaranteed to execute indivisibly (also, a "critical section")
  - if two threads execute the same atomic unit at the same time, one thread will execute the whole sequence before the other begins.

```
hits = hits + 1
```

T1 T2

- How to make multiple instructions seem like one atomic one?

A few definitions

- Critical section:
  - piece of code that only one thread can execute at once. Only one thread at a time will get into the section of code.
- Mutual exclusion:
  - ensuring that only one thread does a particular thing at a time. One thread doing it excludes the other, and vice versa.
- Lock: prevents someone from doing something
  - lock before entering critical section, before accessing shared data
  - unlock when leaving, after done accessing shared data
  - wait if locked
- Synchronization:
  - using atomic operations to ensure cooperation between threads

Example: the Too-Much-Milk problem

Consider a bunch of roommates in a house

```
Person() {
    while (1) {
        Dosomething();
        if (!CheckMilk) BuyMilk();
    }
}
```

Goal:
- 1. never more than one person buys
- 2. someone buys if needed (otherwise "starvation")

Example: the Too-Much-Milk problem

Person A

- 3:00 Look in fridge. Out of milk
- 3:05 Leave for store
- 3:10 Arrive at store
- 3:15 Buy milk
- 3:20 Arrive home, put milk away
- 3:25
- 3:30

Person B

- 3:00 Look in fridge. Out of milk
- 3:05 Leave for store
- 3:10 Arrive at store
- 3:15 Buy milk
- 3:20 Arrive home, put milk away
- Oh no!
Too much milk: solution #1

Basic idea:
- leave a note (kind of like "lock")
- remove note (kind of like "unlock")
- don't buy if there is a note (wait)

```java
if (!noMilk) {
  if (!noNote) {
    leave Note;
    buy milk;
    remove Note
  }
}
```

Why solution #1 does not work?

Threads can get context-switched at any time!

Too much milk: solution #2

```java
Thread A
leave NoteA
if (!noNoteB) {
  if (!noMilk)
    buy milk
} remove NoteA

Thread B
leave NoteB
if (!noNoteA) {
  if (!noMilk)
    buy milk
} remove NoteB
```

Problem with Solution #2

```
Problem: neither thread to buy milk --- think other is going to buy --- starvation!
```

Too much milk: solution #3

```java
Thread A
leave NoteA
while (NoteB)
do nothing;
if (!noMilk)
  buy milk;
} remove NoteA

Thread B
leave NoteB
if (!noNoteA) {
  if (!noMilk)
    buy milk;
} remove NoteB
```

Solution #3: a scenario

```
Either safe for me to buy or others will buy!
```
Solution #3: another scenario

Thread A
leave NoteA
while (NoteB) do nothing;

if (noMilk) buy milk;
remove NoteA

Thread B
leave NoteB
if (noNoteA) {
if (noMilk) buy milk;
}
remove NoteB

Question: any criticisms on this style of providing mutual exclusion?

Locks using load/store
- Dekker’s algorithm, later simplified by Peterson
- No hardware support required

Scenario 1
lockedA = true;
turn = B;
while (lockedB && turn != A);
<critical section>
lockedA = false;
lockedB = true;
turn = A;
while (lockedA && turn != B);
<critical section>
lockedB = false;

Scenario 2
lockedA = true;
turn = B;
while (lockedB && turn != A);
<critical section>
lockedA = false;
lockedB = true;
turn = A;
while (lockedA && turn != B);
<blocks>

Scenario 3
lockedA = true;
turn = B;
while (lockedB && turn != A);
<blocks>
lockedB = true;
turn = A;
while (lockedA && turn != B);
<critical section>
lockedB = false;
A better solution

- Have hardware provide better primitives than simple load and store.
- Build higher-level programming abstractions on this new hardware support.
- Example: using locks as an atomic building block
  ```
  Lock::Acquire --- wait until lock is free, then grabs it
  Lock::Release --- unlock, waking up a waiter if any
  ```
  These must be atomic operations --- if two threads are waiting for the lock, and both see it is free, only one grabs it!
  ```
  lock -> Acquire();
  if (!nomilk)
      buy milk;
  lock -> Release();
  ```

Announcements

- Assignment 1 will be online tonight:
- Send us groupings by Wednesday/Thursday
- Design due by next Tuesday