

15-410

“My computer is 'modern'!”

Synchronization #1

Jan. 28, 2004

Dave Eckhardt

Bruce Maggs

Synchronization

Project 0 feedback progress

- Target: tomorrow
 - Read .announce today and tomorrow
- Red ink on paper
- Test results

Project 1 alert

- “make print” *must work*

Project 0 Common Themes

Style/structure

- Constants instead of #defined tokens
 - “2” is not better than “TYPE_DOUBLE”
 - It is much much much worse
 - Don't ever do that
- Bad variable/function names
 - initialize() should not terminate
- “Code photocopier”
- Excessively long functions
- Harry Bovik did *not* help you write your P0

Project 0 Common Themes

Style/structure

- Code is *read by people*
 - Us
 - Your partner
 - Your manager
 - ...
- Don't make it painful for us
 - or else...

Project 0 Common Themes

Robustness

- Not checking syscall returns (e.g., `tmpfile()`)
- Not finding last function / not handling unnamed function
- Memory leak (no need for `malloc()` at all!)

Project 0 Common Themes

Not following spec

- Hand-verifying addresses (compare vs. 0x0804... 0xc000...)
- Approximating arg-offset info
 - Instead of getting it from the table!!
- Give up via exit() -- caller never authorized that!
- Stopping trace at hard-coded function name

Outline

Me vs. Chapter 7

- Mind your P's and Q's
- Atomic sequences vs. voluntary de-scheduling
 - “Sim City” example
- You *will* need to read the chapter
- Hopefully my preparation/review will clarify it

Outline

An intrusion from the “real world”

Two fundamental operations

Three necessary critical-section properties

Two-process solution

N-process “Bakery Algorithm”

Mind your P's and Q's

What you write

```
choosing[i] = true;  
number[i] =  
    max(number[0], number[1], ...) + 1;  
choosing[i] = false;
```

What happens...

```
number[i] =  
    max(number[0], number[1], ...) + 1;  
choosing[i] = false;
```

Mind your P's and Q's

What you write

```
choosing[i] = true;  
number[i] =  
    max(number[0], number[1], ...) + 1;  
choosing[i] = false;
```

Or maybe...

```
choosing[i] = false;  
number[i] =  
    max(number[0], number[1], ...) + 1;
```

“Computer Architecture for \$200, Dave”...

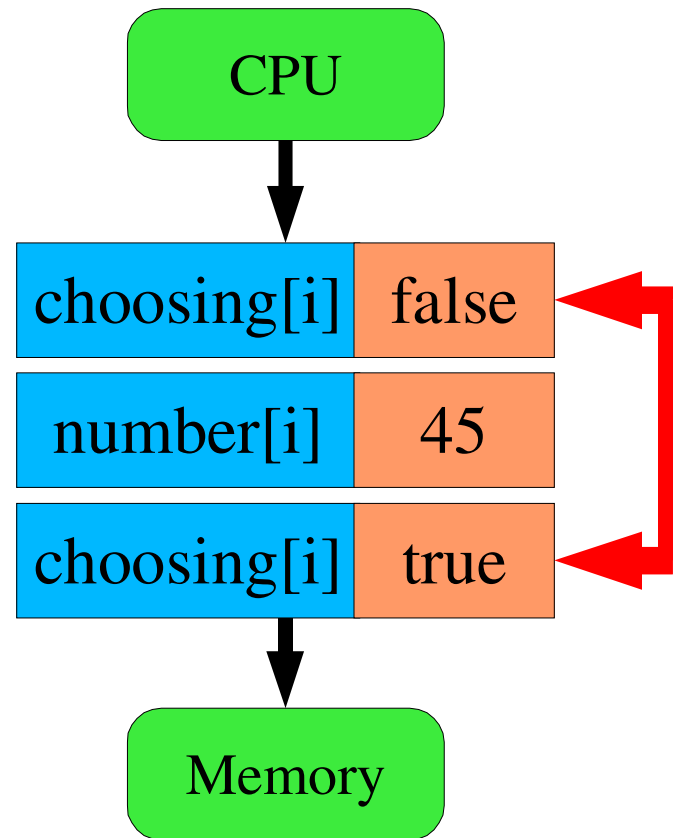
My computer is broken?!

**No, your computer is
"modern"**

- Processor "write pipe" queues memory stores
- ...and coalesces "redundant" writes!

Crazy?

- Not if you're pounding out pixels!



My computer is broken?!

Magic "memory barrier" instructions available...

- ...stall processor until write pipe is empty

Ok, now I understand

- Probably not!
 - <http://www.cs.umd.edu/~pugh/java/memoryModel/>
 - "Double-Checked Locking is Broken" Declaration
- See also "release consistency"

Textbook's memory model

- ...is "what you expect"
- Ok to use simple model for homework, exams

Synchronization Fundamentals

Two fundamental operations

- Atomic instruction sequence
- Voluntary de-scheduling

Multiple implementations of each

- Uniprocessor vs. multiprocessor
- Special hardware vs. special algorithm
- Different OS techniques
- Performance tuning for special cases

Be *very clear* on features, differences

Synchronization Fundamentals

Multiple client abstractions

Textbook covers

- Semaphore, critical region, monitor

Very relevant

- Mutex/condition variable (POSIX pthreads)
- Java "synchronized" keyword (3 uses)

Synchronization Fundamentals

Two Fundamental operations

⇒ Atomic instruction sequence

Voluntary de-scheduling

Atomic instruction sequence

Problem domain

- *Short* sequence of instructions
- Nobody else may interleave same sequence
 - or a "related" sequence
- “Typically” nobody is competing

Non-interference

Multiprocessor simulation (think: “Sim City”)

- Coarse-grained “turn” (think: hour)
- Lots of activity within turn
- Think: M:N threads, M=objects, N=#processors

Most cars don't interact in a turn...

- Must model those that do!

Commerce

<i>Customer 0</i>	<i>Customer 1</i>
<code>cash = store->cash;</code>	<code>cash = store->cash;</code>
<code>cash += 50;</code>	<code>cash += 20;</code>
<code>wallet -= 50;</code>	<code>wallet -= 20;</code>
<code>store->cash = cash;</code>	<code>store->cash = cash;</code>

Should the store call the police?

Is deflation good for the economy?

Commerce – Observations

Instruction sequences are “short”

- Ok to force competitors to wait

Probability of collision is “low”

- Many non-colliding invocations per second
- *Must not* use an expensive anti-collision approach!
 - Oh, just make a system call...
- Common (non-colliding) case must be fast

Synchronization Fundamentals

Two Fundamental operations

Atomic instruction sequence

⇒ Voluntary de-scheduling

Voluntary de-scheduling

Problem domain

- “Are we there yet?”
- “Waiting for Godot”

Example - “Sim City” disaster daemon

```
while (date < 1906-04-18) cwait(date);
while (hour < 5) cwait(hour);
for (i = 0; i < max_x; i++)
    for (j = 0; j < max_y; j++)
        wreak_havoc(i, j);
```

Voluntary de-scheduling

Anti-atomic

- We *want* to be “interrupted”

Making others wait is wrong

- Wrong for them – we won't be ready for a while
- Wrong for us – we can't be ready until *they* progress

We don't *want* exclusion

We *want* others to run - they *enable* us

CPU *de*-scheduling is an OS service!

Voluntary de-scheduling

Wait pattern

```
LOCK WORLD
```

```
while (!(ready = scan_world()))
```

```
    UNLOCK WORLD
```

```
    WAIT_FOR(progress_event)
```

Your partner/competitor will

```
SIGNAL(progress_event)
```

Standard Nomenclature

Textbook's code skeleton / naming

```
do {  
    entry section  
    critical section:  
        ...computation on shared state...  
    exit section  
    remainder section:  
        ...private computation...  
} while (1);
```


Standard Nomenclature

What's muted by this picture?

What's *in* that critical section?

- Quick atomic sequence?
- Need for a long sleep?

For now...

- Pretend critical section is brief atomic sequence
- Study the entry/exit sections

Three Critical Section Requirements

Mutual Exclusion

- At most one process executing critical section

Progress

- Choosing next entrant cannot involve nonparticipants
- Choosing protocol must have bounded time

Bounded waiting

- Cannot wait forever once you begin entry protocol
- ...bounded number of entries by others

Notation For 2-Process Protocols

Process[i] = “us”

Process[j] = “the other process”

i, j are *process-local* variables

- $\{i,j\} = \{0,1\}$
- $j == 1 - i$

Idea #1 - “Taking Turns”

```
int turn = 0;

while (turn != i)
    ;
    ...critical section...
    turn = j;
```

Mutual exclusion - yes

Progress - *no*

- *Strict* turn-taking is fatal
- If $P[i]$ never tries to enter, $P[j]$ will wait forever

Idea #2 - “Registering Interest”

```
boolean want[2] = {false, false};
```

```
want[i] = true;
```

```
while (want[j])
```

```
;
```

```
...critical section...
```

```
want[i] = false;
```

Mutual exclusion – yes

Progress - *almost*

Failing “Progress”

<i>Process 0</i>	<i>Process 1</i>
<code>want[0] = true;</code>	
	<code>want[1] = true;</code>
<code>while (want[1]) ;</code>	
	<code>while (want[0]) ;</code>

It works the rest of the time!

“Taking Turns When Necessary”

Rubbing two ideas together

```
boolean want[2] = {false, false};  
int turn = 0;  
  
want[i] = true;  
turn = j;  
while (want[j] && turn == j)  
;  
...critical section...  
want[i] = false;
```

Proof Sketch of Exclusion

Both in c.s. implies $\text{want}[i] == \text{want}[j] == \text{true}$

Thus both while loops exited because “ $\text{turn} \neq j$ ”

Cannot have $(\text{turn} == 0 \ \&\& \ \text{turn} == 1)$

- So one exited first

w.l.o.g., P0 exited first

- So $\text{turn} == 0$ before $\text{turn} == 1$
- So P1 had to set $\text{turn} == 0$ before P0 set $\text{turn} == 1$
- So P0 could not see $\text{turn} == 0$, could *not* exit loop first!

Proof Sketch Hints

want[i] == want[j] == true

“want[]” fall away, focus on “turn”

turn[] vs. loop exit...

What really happens here?

<i>Process 0</i>	<i>Process 1</i>
<code>turn = 1;</code>	<code>turn = 0;</code>
<code>while (turn == 1);</code>	<code>while (turn == 0);</code>

Bakery Algorithm

More than two processes?

- Generalization based on bakery/deli counter
 - Get monotonically-increasing ticket number from dispenser
 - Wait until monotonically-increasing “now serving” == you

Multi-process version

- Unlike “reality”, two people can get the same ticket number
- Sort by (ticket number, process number)

Bakery Algorithm

Phase 1 – Pick a number

- Look at all presently-available numbers
- Add 1 to highest you can find

Phase 2 – Wait until you hold *lowest* number

- Well, lowest (ticket, process) number
- Then it's your turn

Bakery Algorithm

```
boolean choosing[n] = { false, ... };  
int number[n] = { 0, ... } ;
```

Bakery Algorithm

Phase 1: Pick a number

```
choosing[i] = true;
```

```
number[i] =  
    max(number[0], number[1], ...) + 1;
```

```
choosing[i] = false;
```

Worst case: everybody picks same number!

But at least latecomers will pick a larger number...

Bakery Algorithm

Phase 2: Wait to hold lowest number

```
for (j = 0; j < n; ++j) {  
    while (choosing[j])  
        ;  
    while ((number[j] != 0) &&  
        ((number[j], j) < (number[i], i)))  
        ;  
}  
...critical section...  
number[i] = 0;
```

Summary

Memory is *weird*

Two fundamental operations - understand!

- *Brief exclusion* for atomic sequences
- *Long-term yielding* to get what you want

Three necessary critical-section properties

Understand these race-condition parties!

- Two-process solution
- N-process “Bakery Algorithm”