

15-410

“My computer is 'modern!'”

Synchronization #1
Sep. 11, 2009

Dave Eckhardt

Garth Gibson

Synchronization

Partner sign-up!

- Approximately 8 students un-partnered
- ~8 groups have one-way sign-up (both, please)
- I am spamming the un-signed... let's get through this...

Simics issues

- Simics doesn't simulate time with 100% accuracy
 - Mentioned in handout, but:
 - » Sometimes it runs slower (“of course”)
 - » Sometimes it runs faster (!)
- Simics doesn't blink
 - Not your fault

Outline

Me vs. Chapter 6

- I will cover 6.3 much more than the text does...
 - ...even more than the previous edition did...
 - This is a good vehicle for understanding race conditions
- 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

Imagine you wrote this code:

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

Mind your P's and Q's

Imagine you wrote this code:

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

Imagine what is sent out over the memory bus is:

```
number[i] = 11;  
choosing[i] = false;
```

Is that ok?

Mind your P's and Q's

How about this?

```
choosing[i] = false;  
number[i] = 11;
```

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

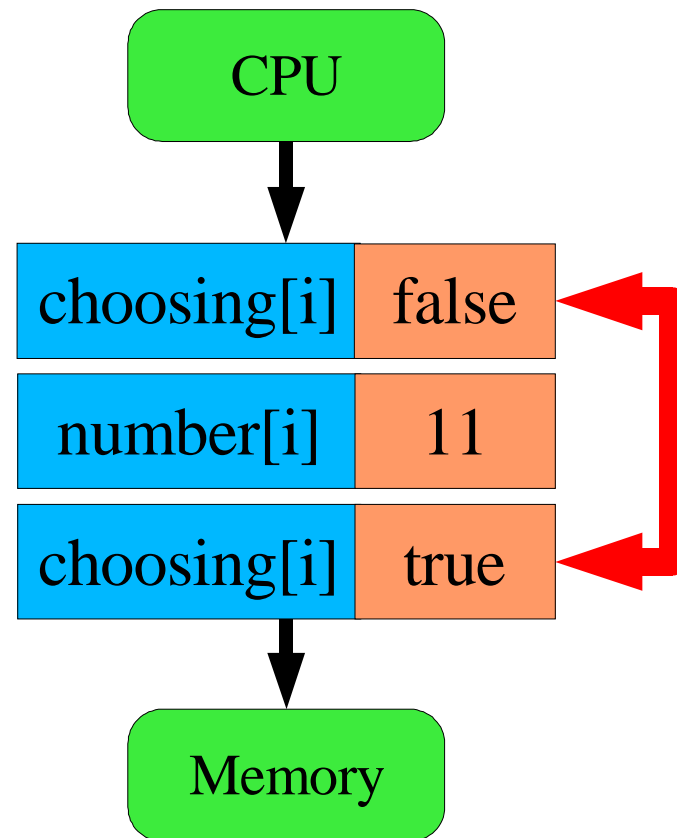
My Computer is Broken?!

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/>
 - see “Double-Checked Locking is Broken” Declaration
- See also “release consistency”

Textbook mutual exclusion algorithm memory model

- ...is “what you expect” (pre-“modern”)
- Ok to use simple model for homework, exams, P2
 - But it's not right for multi-processor Pentium-4 systems...

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

- The two operations are more “opposite” than “the same”

Synchronization Fundamentals

Multiple client abstractions use the two operations

Textbook prefers

- Semaphore, critical region, monitor

Very relevant

- Mutex/condition variable (POSIX pthreads)
- Java “synchronized” keyword (3 flavors)

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 each turn
- Think: M:N threads, M=objects, N=#processors

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

- Must model those that do
- So street intersections can't generally be “processed” by multiple cars at the same time

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 “mutually exclude” competitors (make them wait)

Probability of collision is “low”

- Many non-colliding invocations per second
 - (lots of stores in the city)
- **Must not** use an expensive anti-collision approach!
 - “Just make a system call” is **not** an acceptable answer
- 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 “maximally interleaved against”

Running and 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)
    LOCK WORLD
}
```

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 a 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 wait for non-participants
- Choosing protocol must have bounded time

Bounded waiting

- Cannot wait forever once you begin entry protocol
- ...bounded number of entries by others
 - not necessarily a bounded number of *instructions*

Notation For 2-Process Protocols

Process i = “us”

Process j = “the other process”

i, j are *process-local* variables

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

This notation is “odd”

- But it *may well appear in an exam question*

Assumptions

- Multiple threads (1 CPU with timer, or multiple CPU's)
- Shared memory, but no locking/atomic instructions

Idea #1 - “Taking Turns”

```
int turn = 0;  
  
while (turn != i)  
    continue;  
...critical section...  
turn = j;
```

Idea #1 - “Taking Turns”

```
int turn = 0;  
  
while (turn != i)  
    continue;  
...critical section...  
turn = j;
```

Mutual exclusion –yes (make sure you see it)

Idea #1 - “Taking Turns”

```
int turn = 0;

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

Mutual exclusion –yes (make sure you see it)

Progress - *no*

- *Strict* turn-taking is fatal
- If P0 never tries to enter, P1 will wait forever
 - Violates the “depends on non-participants” rule

Idea #2 - “Registering Interest”

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

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

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

```
    continue;
```

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

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

Mutual Exclusion (Intuition)

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

Mutual Exclusion (Intuition)

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

How about progress?

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 for every *other* interleaving!

“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)
    continue;
...critical section...
want[i] = false;
```

Proof Sketch of Exclusion

Assume contrary: two processes in critical section

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 because “ $\text{turn} == 1$ ” failed

- 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
 - You have lowest number \Rightarrow all people with smaller numbers have already been served

Multi-process version

- Unlike “reality”, two people can get the same ticket number
- Sort by “ticket number with tie breaker”:
 - (ticket number, process number) tuple

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

- Not strictly true: processes may have same number
- Use process-id as a tie-breaker
 - (ticket 7, process 99) > (ticket 7, process 45)
- Your turn when you hold lowest (t,pid)

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 *next wave* of arrivals will pick a larger number...

Bakery Algorithm

Phase 2: Sweep “proving” we have lowest number

```
for (j = 0; j < n; ++j) {  
    while (choosing[j])  
        continue;  
    while ((number[j] != 0) &&  
        ((number[i], i) > (number[j], j)))  
        continue;  
}  
...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 “exclusion algorithms” (which are also race-condition parties)

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