# 15-410

***"My computer is 'modern'!"***

# Synchronization #1
# Sep. 11, 2024

**Dave Eckhardt**

# Learning Science Alert!

**We are in a cumulative lecture series**

- Thread
- Thread synchronization
- Thread synchronization
- Thread synchronization
- A thread-synchronization problem
- Another thread-synchronization problem
- More about that second thread-synchronization problem

# Learning Science Alert!

## We are in a cumulative lecture series

- Thread
- Thread synchronization
- Thread synchronization
- Thread synchronization
- A thread-synchronization problem
- Another thread-synchronization problem
- More about that second thread-synchronization problem

## Learning science

- You might get more out of each of these if you can spend a few minutes right before class skimming the previous ~~linked-list node~~ lecture to remind yourself what it was about
- This “might” happen again later (VM)

# Notice

## Me vs. OSC 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

## Me vs. OS:P+P Chapter 5

- Philosophically very similar
- Examples and focus are different

# Notice

## Not in the book

- "Atomic sequences vs. voluntary de-scheduling"
  - "Sim City" example

## Textbook recommended!

- We will spend ~4 lectures on one chapter (~7 on two)
- This is important stuff
  - Getting a "second read" could be very useful

# 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

**Imagine you wrote this code:**

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

**How about this??**

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

**Is my computer broken???**

- **"Computer Architecture for $200, Dave"...**

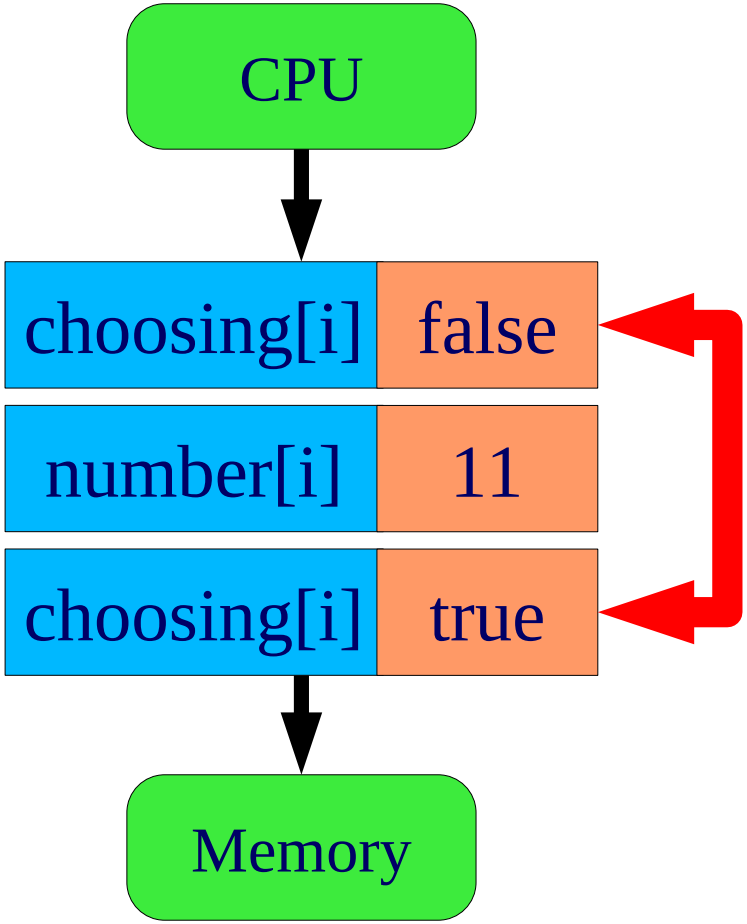
# Is my computer broken?!

**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
  - MFENCE, LFENCE, SFENCE
  - ISB, DMB, DSB
  - EIEIO

## 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 most processors you can buy today!

# 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**

- **"Critical section", semaphore, 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 (aka *key assumptions*)**

1. ***Short*** **sequence of instructions**
2. **Nobody else may interleave same sequence**
   - **Or a "related" sequence**
3. **"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

| Customer 0 | Customer 1 |
|---|---|
| `cash = store->cash;` | `cash = store->cash;` |
| `cash += 50;` | `cash += 20;` |
| `wallet -= 50;` | `wallet -= 20;` |
| `store->cash = cash;` | `store->cash = cash;` |

**Should the store call the police?**

**Is deflation good or bad for the economy?**

# Commerce – Observations

1. **Instruction sequences are "short"**
   - Ok to "mutually exclude" competitors (make them wait)
2. **Interference is disastrous**
3. **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

# Commerce – Observations

1. **Instruction sequences are "short"**
   - Ok to "mutually exclude" competitors (make them wait)
2. **Interference is disastrous**
3. **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

**These are the *mutex assumptions***

# 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

**"Traditional CS" 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 to block for a while?

**For now...**

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

# Three Critical Section Requirements

***Mutual Exclusion***

- **At most one thread is executing each critical section**

***Progress***

- **Choosing protocol must have bounded time**
  - **Common way to fail: choosing next entrant cannot wait for non-participants**

***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

## Assumptions

- Multiple threads (1 CPU with timer, or multiple CPU's)
- Shared memory, but no locking/atomic instructions
- No thread “runs at zero speed”

# Notation For 2-“Process” Protocols

## Assumptions

- Multiple threads (1 CPU with timer, or multiple CPU's)
- Shared memory, but no locking/atomic instructions
- No thread “runs at zero speed”

## Notation

- Thread `i` = “us”
- Thread `j` = “the other thread”
- `i`, `j` are *thread-local* variables
  - {i,j} = {0,1}
  - j == 1 – i

## This notation is “odd”

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

# Idea #1 - “Taking Turns”

```
int turn = 0;

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

# Idea #1 - “Taking Turns”

```
int turn = 0;

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

**Mutual exclusion – yes (make sure you see it)**

# Idea #1 - “Taking Turns”

```
int turn = 0;

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

**Mutual exclusion – yes (make sure you see it)**

**Progress - *no***

- ***Strict*** **turn-taking is fatal**
- **If T0 never tries to enter, T1 will wait forever**
  - **Violates the “depends on non-participants” rule**

# Idea #2 - “Registering Interest”

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

want[i] = true;                // entry
while (want[j])                // entry
  continue;                    // entry
...critical section...
want[i] = false;               // exit
```

# Mutual Exclusion (Intuition)

| Thread 0 | Thread 1 |
|---|---|
| `want[0] = true;` | |
| `while (want[1]) ;` | |
| `...enter...` | `want[1] = true;` |
| | `while (want[0]) ;` |
| | `while (want[0]) ;` |
| `want[0] = false;` | `while (want[0]) ;` |
| | `...enter...` |

# Mutual Exclusion (Intuition)

| Thread 0 | Thread 1 |
|---|---|
| `want[0] = true;` | |
| `while (want[1]) ;` | |
| `...enter...` | `want[1] = true;` |
| | `while (want[0]) ;` |
| | `while (want[0]) ;` |
| `want[0] = false;` | `while (want[0]) ;` |
| | `...enter...` |

**Ok, how about progress?**

# Mutual Exclusion (Intuition)

| Thread 0 | Thread 1 |
|---|---|
| `want[0] = true;` | |
| | `want[1] = true;` |
| `while (want[1]) ;` | |
| | `while (want[0]) ;` |

**It works for every *other* interleaving!**

# How Are We Doing?

**First try**

- No progress

**Second try**

- Also no progress

**Buckle your seatbelts!**

- Something strange is about to happen
  - Kind of like det(AB) = det(A)*det(B)

# “Peterson's Solution” (1981)

**(15-410 name: “Taking turns when necessary”)**

```
boolean want[2] = {false, false};
int turn = 0;

want[i] = true;                    // entry
turn = j;                          // entry
while (want[j] && turn == j)       // entry
    continue;                      // entry
...critical section...
want[i] = false;                   // exit
```

# Proof Sketch of Exclusion

**Assume contrary: two threads in critical section**

**Both in c.s. implies want[i] == want[j] == true**

**Thus both while loops exited because "turn != j"**

**Cannot have (turn == 0 && turn == 1)**

- **So one exited first**

**w.l.o.g., T0 exited first because "turn ==1" failed**

- **So turn==0 before turn==1**
- **So T1 had to set turn==0 before T0 set turn==1**
- **So T0 could not see 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?

| Thread 0 | Thread 1 |
|---|---|
| `turn = 1;` | `turn = 0;` |
| `while (turn == 1) ;` | `while (turn == 0) ;` |

# How Are We Doing?

**Two "processes"?**

- **Solved!**

**More than two processes?**

- **A mere matter of induction!**

# Bakery Algorithm (Lamport 1974)

## 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 ⇒ 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 (Lamport 1974)

**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 (Lamport 1974)

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

# Bakery Algorithm (Lamport 1974)

**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 (Lamport 1974)

**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 non-running*** **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"**