https://www.youtube.com/watch?v=NUnJc82ptd4

Announcements

- Yay snow!
- Be safe walking
- Be safe driving
- You Will Be Warm

Modifying shared data

Thread 1:
```
shared.x++;
```

Thread 2:
```
shared.x++;
```

Poll: Suppose two threads increment the same int variable. What steps are involved in each incrementation?

A) Load, increment, store
B) Only one step, it is an atomic operation
C) Access, index, increment
D) Read, write
E) None of the above

PollEv.com/leahp
text **`leahp`** to **22333**

Modifying shared data

Poll Answer: A) Load, increment, store

Thread 1:	Thread 2:
`shared.x++;`	`shared.x++;`

Scenario 1

1. T1 LOAD ($reg_1 \leftarrow 0$)
2. T1 INC ($reg_1 \leftarrow 1$)
3. T1 STORE ($x \leftarrow 1$)
4. T2 LOAD ($reg_2 \leftarrow 1$)
5. T2 INC ($reg_2 \leftarrow 2$)
6. T2 STORE ($x \leftarrow 2$)

Scenario 2

1. T1 LOAD ($reg_1 \leftarrow 0$)
2. T2 LOAD ($reg_2 \leftarrow 0$)
3. T2 INC ($reg_2 \leftarrow 1$)
4. T2 STORE ($x \leftarrow 1$)
5. T1 INC ($reg_1 \leftarrow 1$)
6. T1 STORE ($x \leftarrow 1$)

From last time: Racing for the Critical Section

- **critical section** - bit of code that accesses a shared data structure
- **race condition** - situation where the result of the program depends on which thread accesses the shared data structure first
- How can we coordinate threads to safely access the critical section?
  - ... Synchronization! (coming soon to an auditorium near you)

Lecture 27: Synchronization

CS 2110, Matt Eichhorn and Leah Perlmutter

December 2, 2025

Announcements

- Exam review session Friday 12/12 in Phillips 101 at 1-4 pm

Roadmap

Java, Complexity, OOP

- start– 9/30

ADTs I

- List, Stack, Queue, Iteration
  - 10/2 – 10/16

ADTs II

- Trees, Set, Map, Hash Table, Graph
  - Tues 10/21- 11/13

Beyond ADTs

- Graphical User Interfaces & Event-Driven Programming
  - 11/18, 11/20
- **Parallel Programming**
  - 11/25, 12/2
- Data Structures and Social Implications
  - 12/4

See Also

- Operating Systems: Three Easy Pieces. Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Arpaci-Dusseau Books: November, 2023 (Version 1.10).
  - ch 25: Dialogue
  - ch 26: Concurrency and Threads
  - ch 28: Locks
  - ch 30: Condition Variables
  - ch 32: Concurrency Bugs
- Java tutorial on Synchronized Methods

OER = Open Educational Resource!!!

Overview of 11/25 & 12/1

- Concurrent Tasks and the Operating System ✅
- The Thread Class ✅
- Race Conditions ✅
- Data Structures and Thread Safety ✅
- **Dec 1: Safely Coordinating Threads (synchronization)**
  - Race Conditions Revisited
  - Bob, Yasaman, and the Locking Refrigerator
  - Mutex Locks and Synchronized Blocks
  - Mutiple Shared Resources
  - Condition Variables
  - Monitor Pattern

Race Conditions

(the need for synchronization)

Shared Data

- **Code demo**
  - counting to 2 million
- **Question**
  - What will be the final value of shared.x?
    - When we ran the experiment, the outcomes were all over the place between 1 and 2 million.

Race Conditions

- **atomic –** describes an operation that executes fully or not at all, whose parts cannot be separated
- `shared.x++` is actually 3 steps!
  - LOAD: Load the value of shared.x from RAM into Register_1
  - INCREMENT: Add 1 to Register_1
  - STORE: Store Register_1 into RAM at the address shared.x
- Why? Compiled languages
  - Source code is compiled into machine code which runs on the CPU
  - Machine code instructions are much simpler and less powerful than lines of code in Java
  - A line of Java code is not necessarily atomic

Machine Code and Hardware

- The CPU has several registers that can store data directly on the CPU
- Machine code CANNOT
  - do arithmetic on data stored in memory (RAM)
- Machine code CAN
  - perform arithmetic on data stored in registers
  - load data from RAM into a register
  - store data from a register into RAM

Example: shared primitive

Thread 1:
```
shared.x++;
```

Thread 2:
```
shared.x++;
```

Scenario 1

1. T1 LOAD ($reg_1 \leftarrow 0$)
2. T1 INC ($reg_1 \leftarrow 1$)
3. T1 STORE ($x \leftarrow 1$)
4. T2 LOAD ($reg_2 \leftarrow 1$)
5. T2 INC ($reg_2 \leftarrow 2$)
6. T2 STORE ($x \leftarrow 2$)

Scenario 2

1. T1 LOAD ($reg_1 \leftarrow 0$)
2. T2 LOAD ($reg_2 \leftarrow 0$)
3. T2 INC ($reg_2 \leftarrow 1$)
4. T2 STORE ($x \leftarrow 1$)
5. T1 INC ($reg_1 \leftarrow 1$)
6. T1 STORE ($x \leftarrow 1$)

Racing for the Critical Section

- **critical section** - piece of code that accesses a shared variable and must not be concurrently executed by more than one thread[1]
- **race condition** - situation where the result of the program depends on which thread accesses the shared data structure first
- How can we coordinate threads to safely access the critical section?
  - ... Synchronization! (coming soon to an auditorium near you)

[1] Operating Systems: Three Easy Pieces. Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Arpaci-Dusseau Books: November, 2023 (Version 1.10). Ch. 26.

Possible solutions to race conditions

avoid concurrency 🚫

avoid shared memory 🚫

read-only access to shared memory 🚫

<--- these do prevent race conditions, but the cost is to sacrifice benefits of concurrency

synchronization ✅

<--- Necessary to fully leverage the benefits of concurrency!

good use of **synchronization** makes critical sections of code (appear to) execute atomically

Bob, Yasaman, and the Locking Refrigerator

(see additional slides)

Mutex Locks and Synchronized Blocks

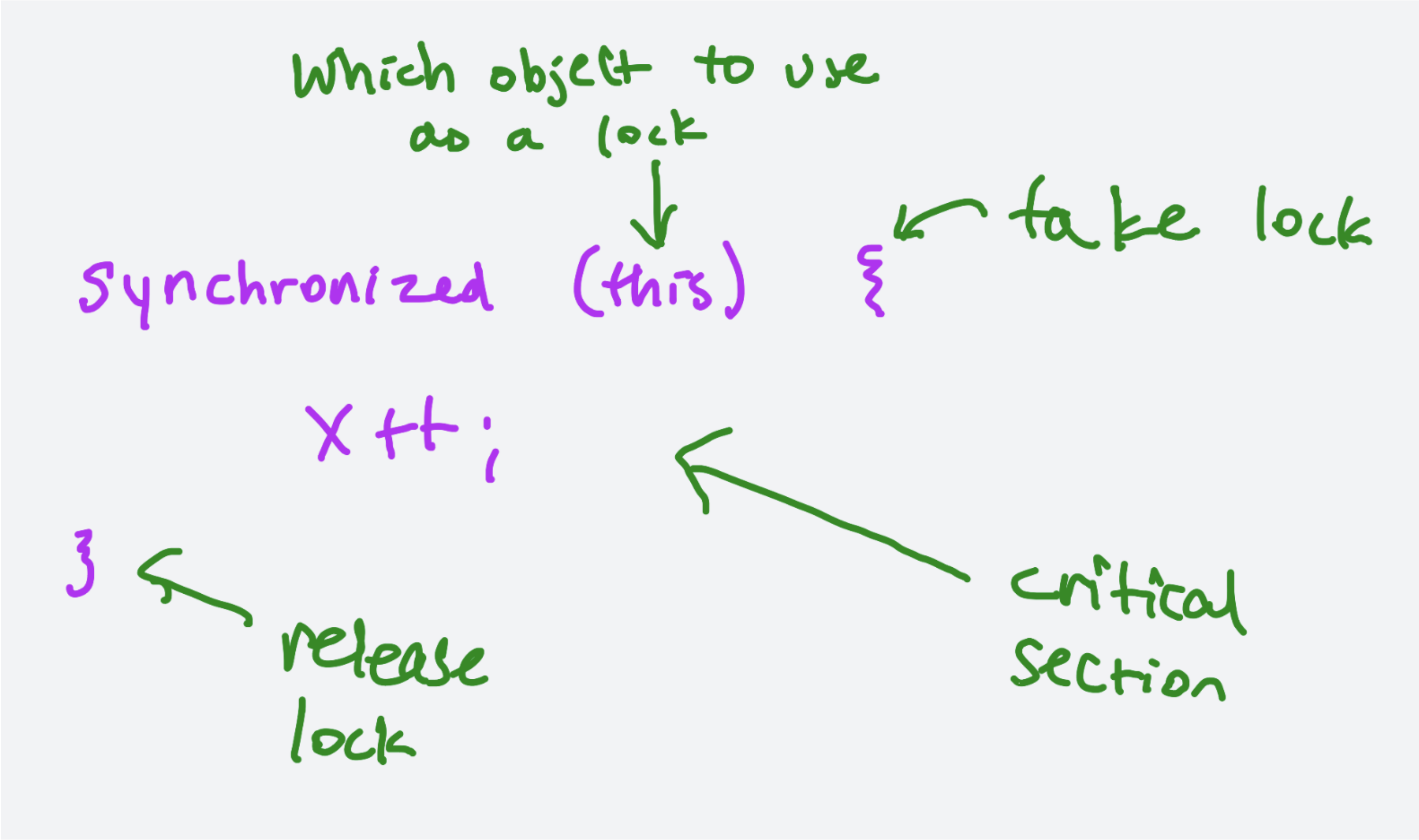
Mutex Locks

- **mutual exclusion –** the prohibition against two threads running the same critical section at the same time
- **mutex lock –** an object used to support mutual exclusion (aka “lock” or “mutex”)
- In Java any object can serve as a mutex lock
- “block B synchronized on L” means that L is used as a lock for block B

- DEMO: Synchronization

Mutex Locks

- WHITEBOARD: syntax for synchronized block in Java



Mutex Locks (abstractly)

- **state**
  - available (boolean)
- **operations**
  - **acquire**: the thread attempting to acquire will block (pause) until the lock becomes available, then it will acquire the lock
  - **release:** thread calling release will no longer hold the lock, and the lock will become available to other threads
- **thread responsibilities**
  - disciplined synchronization

Disciplined Synchronization

- DEMO: SynchronizationDemo
  - decrement method (undisciplined version & update to be disciplined)
- Key points
  - Java provides support for synchronization, but it doesn't synchronize for you
  - You have to synchronize yourself by making sure to only access shared data in a synchronized block (disciplined synchronization)
  - You can think of a lock as less like an actual padlock and more like a sticky note with a picture of a padlock

Disciplined Synchronization

- DEMO: Which object to synchronize on?
  - incLock, decLock, this
- Key points
  - Code that accesses the same shared data should synchronize on the same object
  - It's conventional to use *this* as the object to synchronize on

Multiple Shared Resources

Multiple Shared Resources

- DEMO: Deadlock
  - chefs & dishwashers
- **deadlock –** situation in which multiple threads are unable to make progress since they are all stuck waiting to acquire a mutex that another thread controls.
- Poll: What are some ways to fix the deadlock? →

PollEv.com/leahp
text **`leahp`** to **`22333`**

Multiple Shared Resources

Answer: Some ways to fix the deadlock...

- DEMO: Deadlock Soluion 1
  - one lock for everything
- DEMO: Deadlock Solution 2
  - Consistent acquisition order
- DEMO: Deadlock Solution 3
  - reduced critical section

Conditions for Deadlock & Possible Solutions

- Multiple threads attempting to acquire same lock(s)
  - unavoidable
- Each thread needs >1 lock at the same time
  - solution: make it only one lock for everything
    - beware of the correctness/performance trade-off
  - solution: reduce the locked portions so that each thread only needs one of the locks at a time (if possible)
- Circular dependency
  - standardize the order of locking

Condition Variables

Condition Variables

- DEMO: Kitchen Simulation
  - now there is a shortage of pans...

Condition Variables (abstractly)

- state
  - set of waiting threads (wait set)
- operations
  - wait: Add self to wait set.
  - signal (notify): Wake up one thread from the wait set.
  - broadcast (notify all): wake up all threads from the wait set.

Condition Variables

- DEMO: Kitchen Simulation with Condition Variables
- Q: Why a while loop?
  - A: Waking doesn't guarantee state
- Q: Why notifyAll?
  - A: to avoid deadlock if different threads have different needs
- Q: Why notify on decrement?
  - A: Best practices for correctness
- Q: Why hold mutex while waiting?
  - A: Atomicity of reading state variable and deciding whether to sleep

Condition Variables (abstractly)

- state
  - set of waiting threads (wait set)
- operations
  - wait: Add self to wait set.
    - Then release lock, and go to sleep. We will be given the lock back before wait returns.
  - signal (notify): Wake up one thread from the wait set.
    - It will acquire the lock and its wait method will return.
  - broadcast (notify all): wake up all threads from the wait set.
    - In turn, they will each acquire the lock and return from the wait method.

Condition Variables: Thread responsibilities

- establish state variable that signaling thread can modify and waiting thread can read
- waiter: <u>while</u> state variable not ready, wait on the condition variable
  - being woken does not guarantee we have exclusive access
- signaler: modify state variable and signal
- use a lock to protect the state variable to prevent race conditions with different threads reading and writing it
- use different condition variables for different states
  - if you wake the wrong kind of thread and it goes back to sleep, everyone could be asleep at once!

Data Structures and Thread Safety

Data Structures and Thread Safety

- a class follows the **monitor pattern** when each instance's public methods are protected by that instance's mutex lock
- monitor pattern follows **coarse grained locking** = one lock for the whole object (e.g. lock the fridge door)
- contrast with **fine grained locking** = several locks protecting different parts (e.g. the milk shelf, bread shelf, and yogurt shelf)
  - (not the monitor pattern)
- Pro: supports concurrency with high confidence and less error prone than fine grained locking
- Con: using one lock for all the methods degrades performance by preventing two threads from running two methods even if it would be safe

Metacognition

- Take 1 minute to write down a brief summary of what you have learned today
    - mutual exclusion –
    - mutex lock -
    - disciplined synchronization –
    - condition variable –
    - monitor pattern –

Thanks and enjoy the snow!

Announcements

- Exam review session Friday 12/12 in Phillips 101 at 1-4 pm
- Stay safe, stay warm out there!