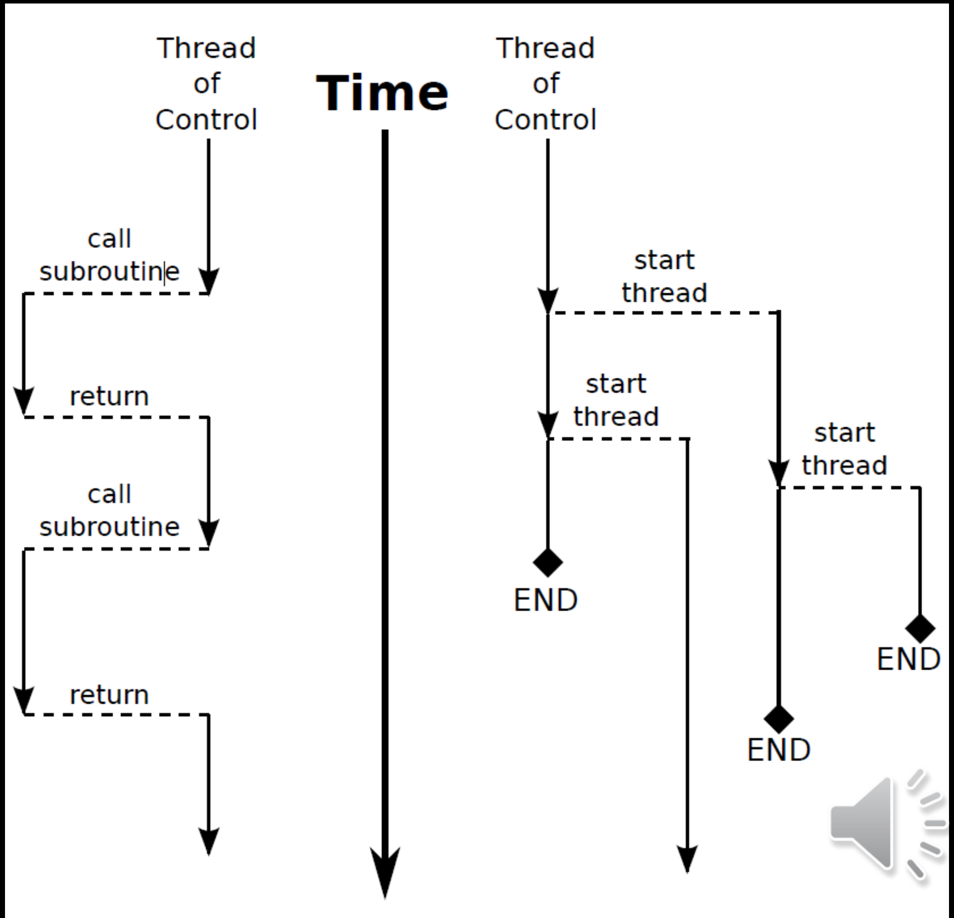
Concurrency

- Multitasking – concurrent tasks
  - Parallel processing is concurrent tasks
  - Concurrent tasks are not necessarily parallel
- Thread – thread of control or execution
  - Every program has at least one thread
  - GUI programs typically have at least three
    - Main routine
    - Drawing
    - Event handling

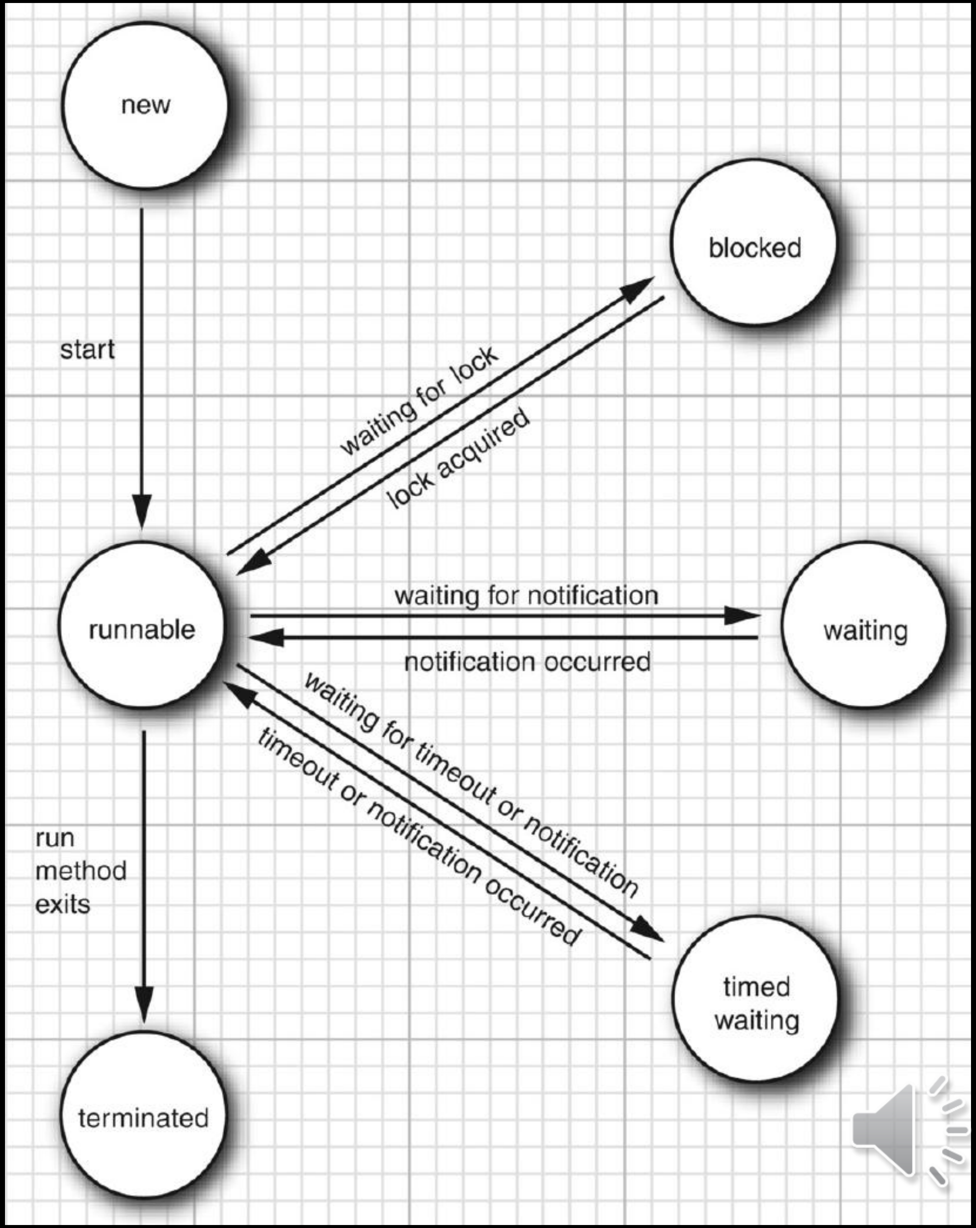
Concurrency

- Single thread
  - Subroutine method is called
  - Subroutine completes task
  - Subroutine returns
- Multiple threads
  - A thread starts another thread
  - Each thread is thread of control/execution
  - Threads run concurrently
  - Threads can call subroutines
  - Threads can start additional threads



Concurrency

- New
- Runnable
- Blocked
- Waiting
- Timed waiting
- Terminated



Concurrency

- Java Thread
  - Executes a single thread only once to perform a task
  - It cannot be restarted once it terminates
  - The object cannot be used as a thread again once terminated
- Java now has two types threads
  - Platform Threads – traditional, thinly wraps around OS thread, managed by the OS
    - OS dependent, resource intensive
    - Blocking operation causes the OS thread to block
    - Good for CPU intensive operations and applications that only require a few threads
    - Thread class and Runnable Interface
  - Virtual Threads (Java 21+) – lightweight, managed by the JVM instead of the OS
    - Blocking operating does not block the OS carrier thread
    - Good for highly threaded applications or I/O-bound tasks with high concurrency
    - Thread.ofVirtual()

Concurrency

- Common thread operations
  - run() – task to perform; start() – starts the thread
  - isAlive() – determine the status
  - interrupt() – interrupt the thread execution
  - join(), join(milliseconds) – wait for other threads to terminate
  - sleep(), sleep(milliseconds), sleep(milliseconds, int nanoseconds) – pause thread
  - setPriority(int), getPriority(), setDaemon(boolean) – prioritize the thread

Concurrency

User Thread	Daemon Thread
JVM wait until user threads finish. It never exit until all user threads finish.	JVM will not wait for daemon threads to finish. It will exit when all user threads finish.
JVM will not force user threads to terminate. It will wait for user threads to terminate themselves.	If all user threads have finished their work JVM will force the daemon threads to terminate
User threads are created by the application.	Mostly Daemon threads are created by the JVM.
Mainly, user threads are designed for a specific task.	Daemon threads are design to support user threads.
User threads are foreground threads.	Daemon threads are background threads.
User threads are high priority threads.	Daemon threads are low priority threads.
Its life independent.	Its life depends on user threads.

Concurrency

Attribute	Daemon	User
Priority	Low, JVM does not prioritize	High, JVM waits for completion
Lifecycle	No specific lifecycle, dependent on user threads	Independent lifecycle; can run independently
CPU	Not guaranteed CPU time	Guaranteed CPU time
Designed for	Background tasks (e.g., garbage collection)	Specific application tasks
Execution ground	Background execution	Foreground execution

Concurrency

- Left – not concurrent, not parallel
- Right – concurrent, not parallel
- Bottom – concurrent and parallel

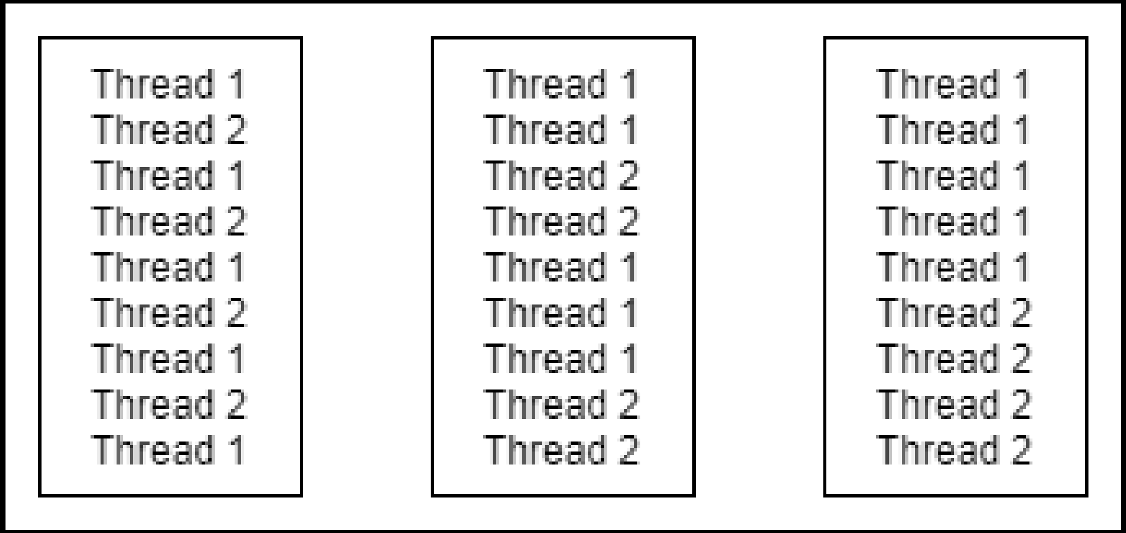
t1	balance = read(Acct1)
t2	balance = balance + deposit
t3	write(Acct1, balance)
t4	balance = read(Acct2)
t5	balance = balance - withdraw
t6	write(Acct2, balance)
t7	balance = read(Acct3)
t8	balance = balance - withdraw
t9	write(Acct3, balance)

t1	balance = read(Acct1)
t2	balance = read(Acct2)
t3	balance = balance + deposit
t4	balance = balance - withdraw
t5	balance = read(Acct3)
t6	balance = balance - withdraw
t7	write(Acct1, balance)
t8	write(Acct3, balance)
t9	write(Acct2, balance)

t1	balance = read(Acct1)	balance = read(Acct2)	balance = read(Acct3)
t2	balance = balance + deposit	balance = balance - withdraw	balance = balance - withdraw
t3	write(Acct1, balance)	write(Acct2, balance)	write(Acct3, balance)

Concurrency

- Single thread - definite and predictable order
- Multiple threads - indefinite and unpredictable order
  - Each thread is definite and predictable, but
  - Simultaneous order is unknown – each program execution can vary



Concurrency

- Threads can share resources which can lead to a race condition
- Example: Account A has $500. Two people are making transactions on the same account at the same time—one thread per transaction

1.	Thread 1: balance1 = Read(A)	load balanceA of 500 into balance1
2.	Thread 2: balance2 = Read(A)	load balanceA of 500 into balance2
3.	Thread 1: balance1 = balance1 + 100	balance1 = 500 + 100 = 600
4.	Thread 1: Write(balance1, A)	store balance1 into balanceA
5.	Thread 2: balance2 = balance2 – 100	balance2 = 500 – 100 = 400
6.	Thread 2: Write(balance2, A)	store balance2 400 into balanceA

Concurrency

- Even a simple operation can be multiple operations

```
MOV R1, 5
MOV R2, 10
ADD R1, R2
```

- Or, more specifically in our case, Java bytecode

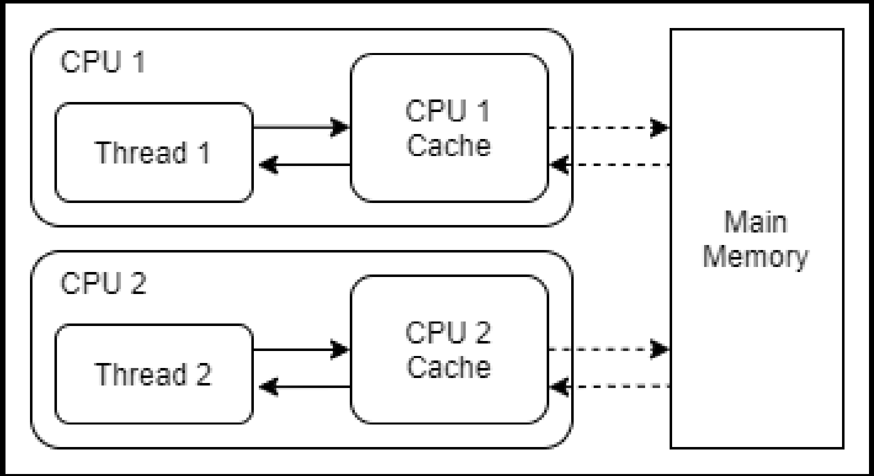
```
Bipush        5
istore_1
Bipush        10
istore_2
iload_1
iload_2
Iadd
istore_1
```

Concurrency

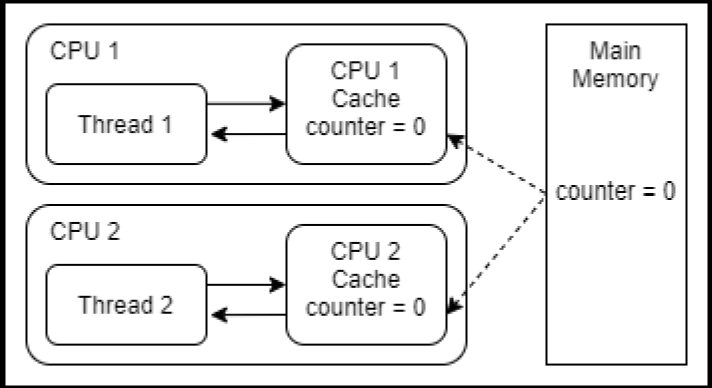
- Mutual exclusion – exclusive access to a resource
  - Thread gets access to shared resource
  - No other thread can access it until the resource is released
  - Java provides mutual exclusion
    - Synchronization – locks resource and makes other threads wait
      - Of objects – only one thread can use the object at a time
      - Of methods – only one thread can use the method at a time
        - Instance methods – only one object can use the method of a shared object at a time
        - Static method – only one thread can use the shared method at a time
      - Of blocks – only one thread can use the block of code at a time
    - Atomic variables – have operations that cannot be interrupted
    - Volatile variables – prevents local caching

Concurrency

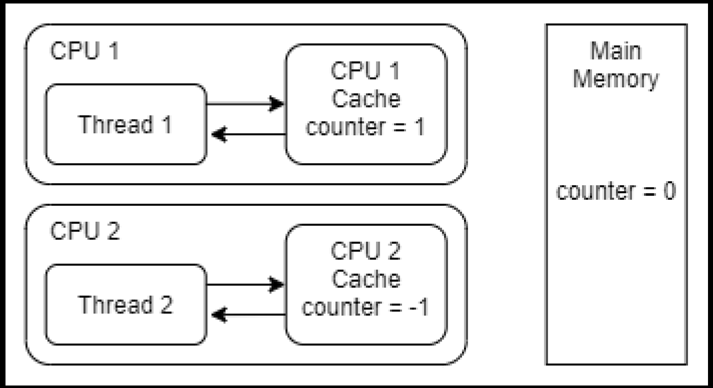
- Volatile Variables
  - Each CPU has cache
  - Threads can cache shared data locally
  - Volatile variables provides visibility
  - Note: it does not prevent race conditions



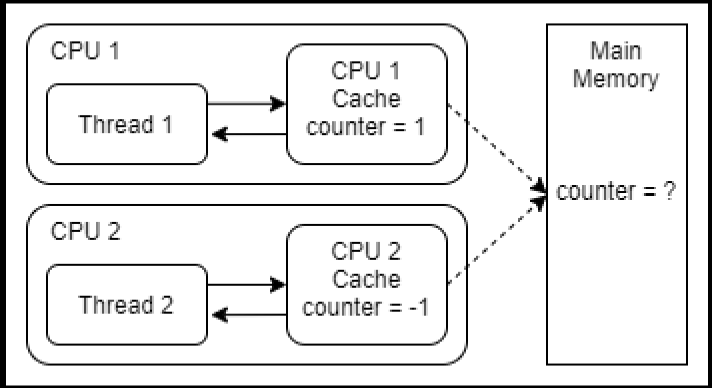
Threads



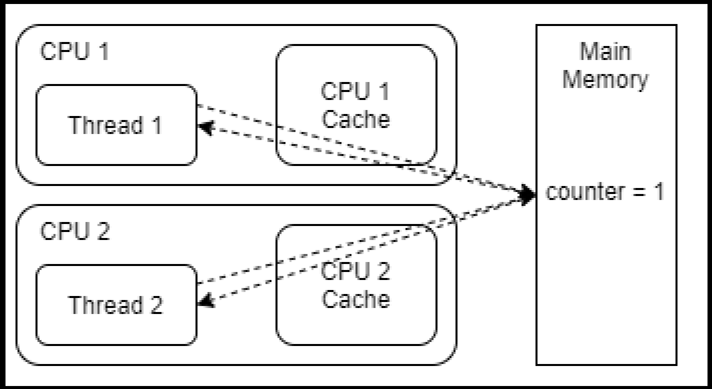
Read and cache the value locally.

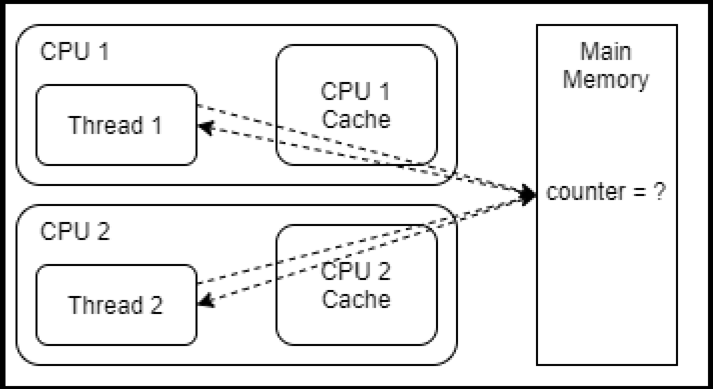


Operate on the local value.



Race to write the value back.





Conceptually, threads will use the value directly from memory. It’s essentially “write-through” caching. Unfortunately, it does not prevent race conditions by itself. Writing back is still an issue.