# Concurrency (Wk 6-10)

# Week 6 - Executing MIPS code in single-core processor

(Concurrency in hardware)

## Pipelining

**From C to execution in MIPS processor**

- Consider the following code segment in C:

```
a = b + e;
c = b + f;
```

- Assume that variables' values are in memory and are addressable as offsets from `$t0`
- How is the code executed within a pipelined MIPS processor?

To approach this question:

- Translate the C code segment to MIPS instructions
- Simulate the execution of MIPS instructions within a pipelined processor

`a = b + e`
in MIPS we need to:

- load values of two variables into 2 registers
  - b in `0($t0)`, e in 4( `$t0` ), f in `8($t0)` as variables' values are assumed to be in memory and addressable as offsets from $t0
- store sum in 3rd register
- save sum into main memory

`c = b + f`
similar procedure

- So for the code segment

```
a = b + e;
c = b + f;
```

- The MIPS code is:

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

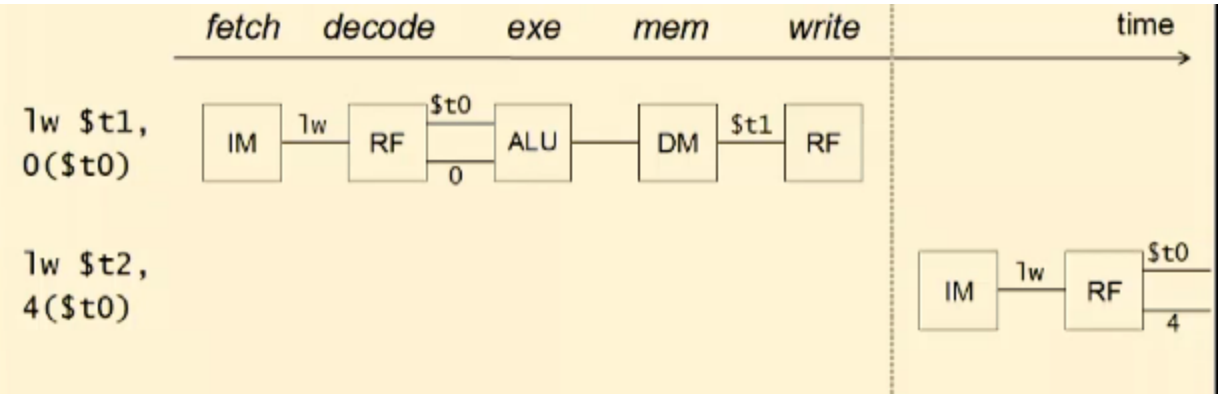
9

simulating these MIPS instructions in a pipelined processor:

note that each MIPS instruction has the same 5 stages:

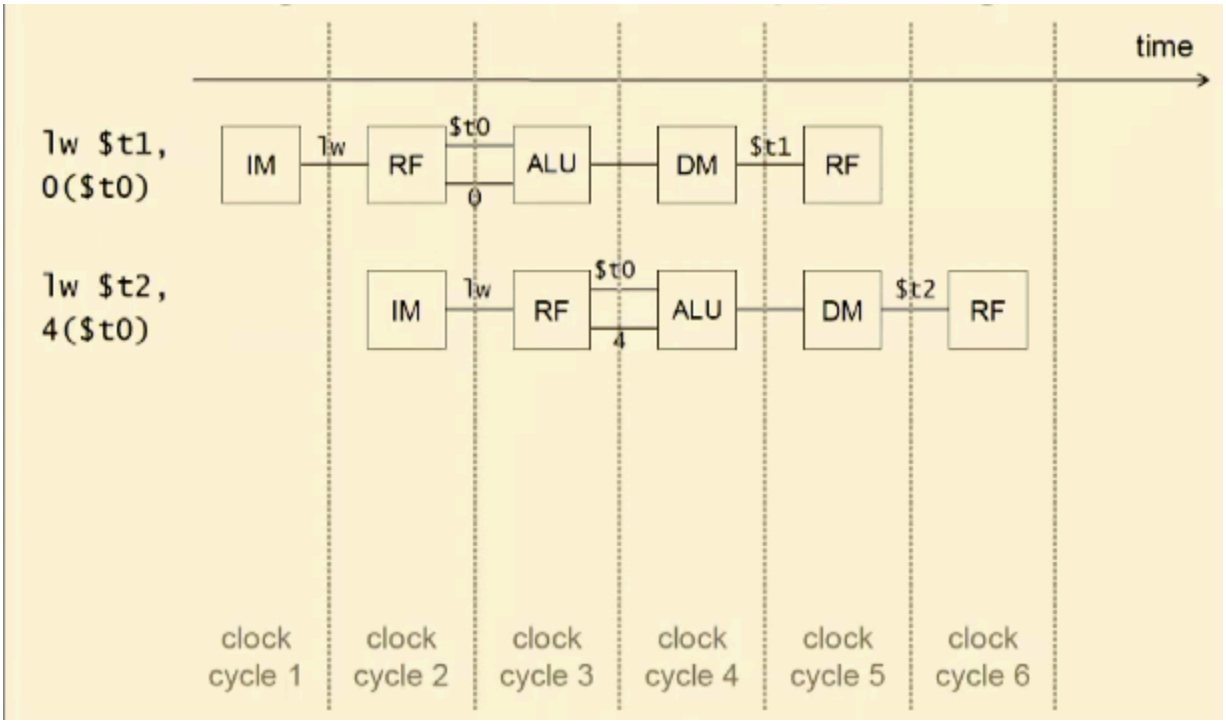
- **IF** = fetch - load the instruction from memory
- **ID** = decode
- **EX** = execute
- **MEM** = memory access
- **WB** = write back (to register file - file that gives access to registers that are in processors, not in main memory)

without pipelining:




5 clock cycles to execute one instruction. total 10 to execute 2 (sequential) - where each compoent is used once in clock cycle

pipelining - overlap the execution of different instructions (subdivide instructions to different phases)

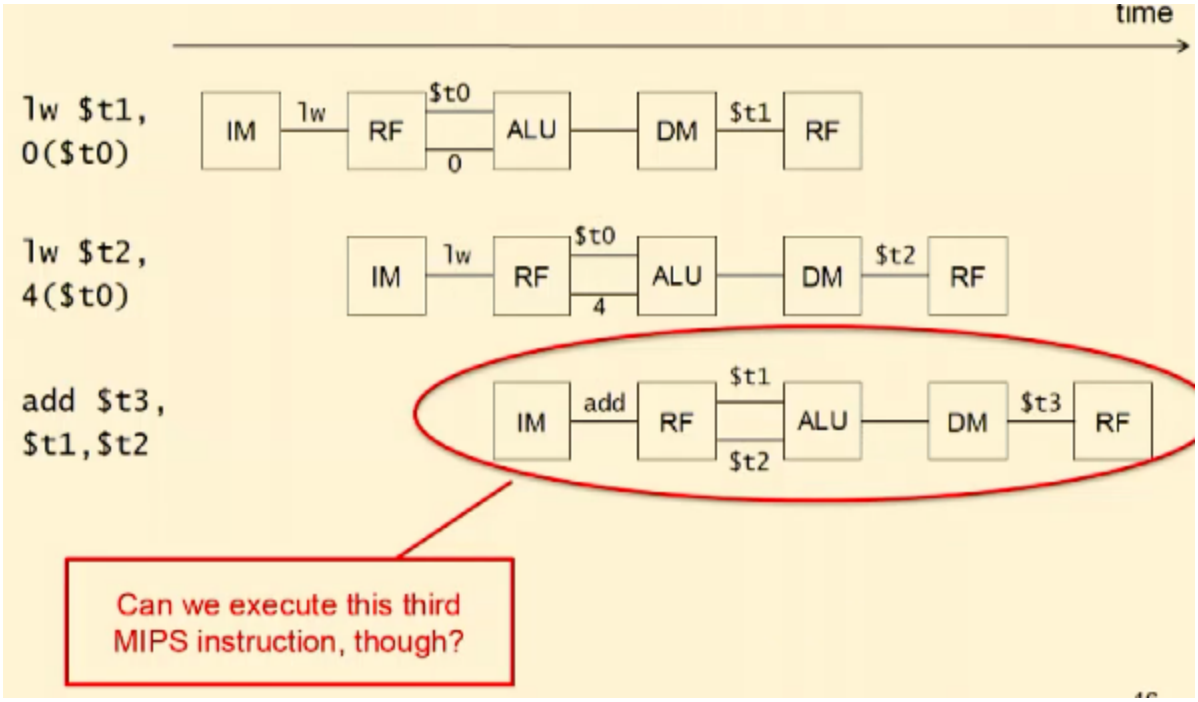



example of overlapping: as IM is not in use after clock cycle 1 by first instruction, IM can be used by second instruction in clock cycle 2 to enable overlapping (hence pipelining )

pipelining increases throughput (number of instructions executed per unit time)

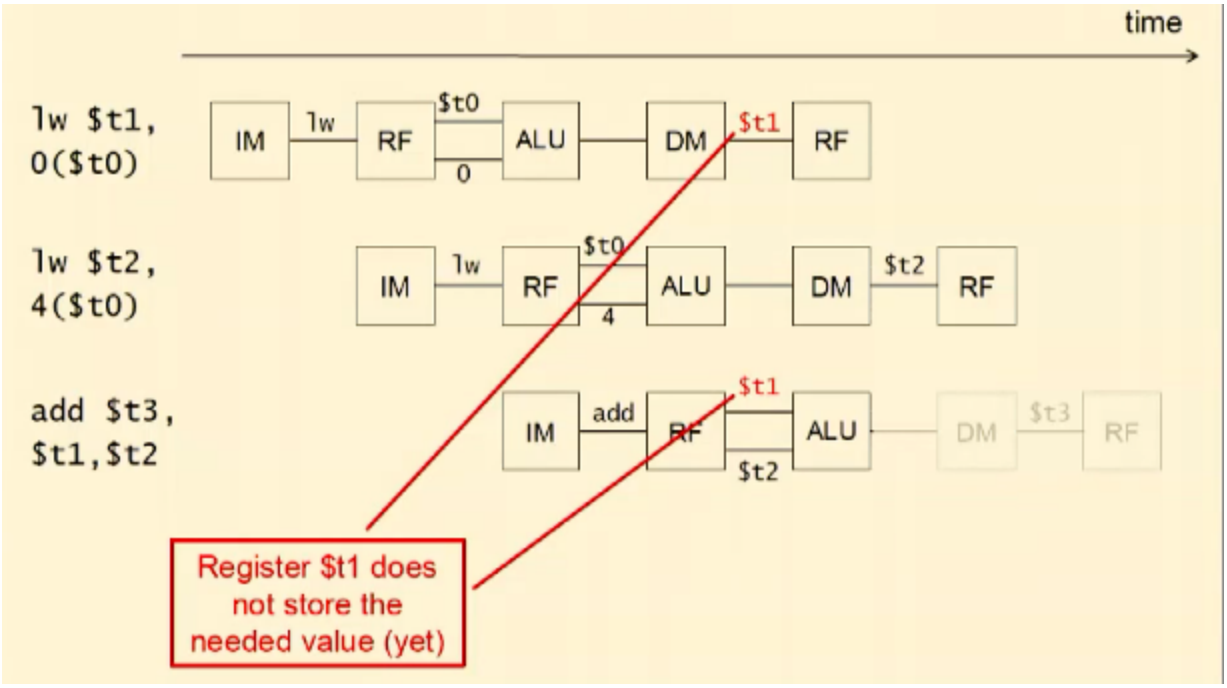
cannot start more than 1 instruction at same time as each component can be accessed once one at a time (esp in. a single core processor)

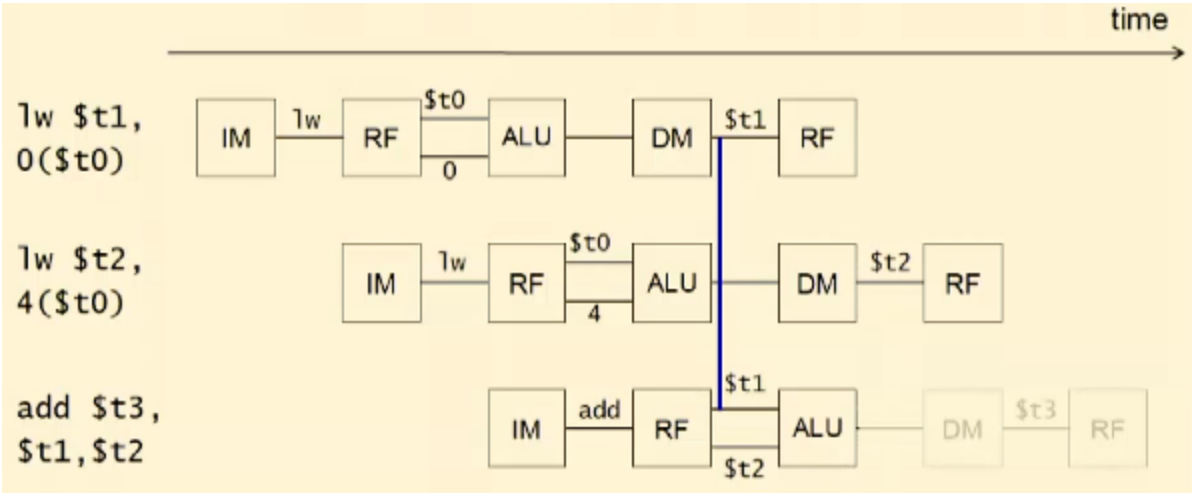
question:



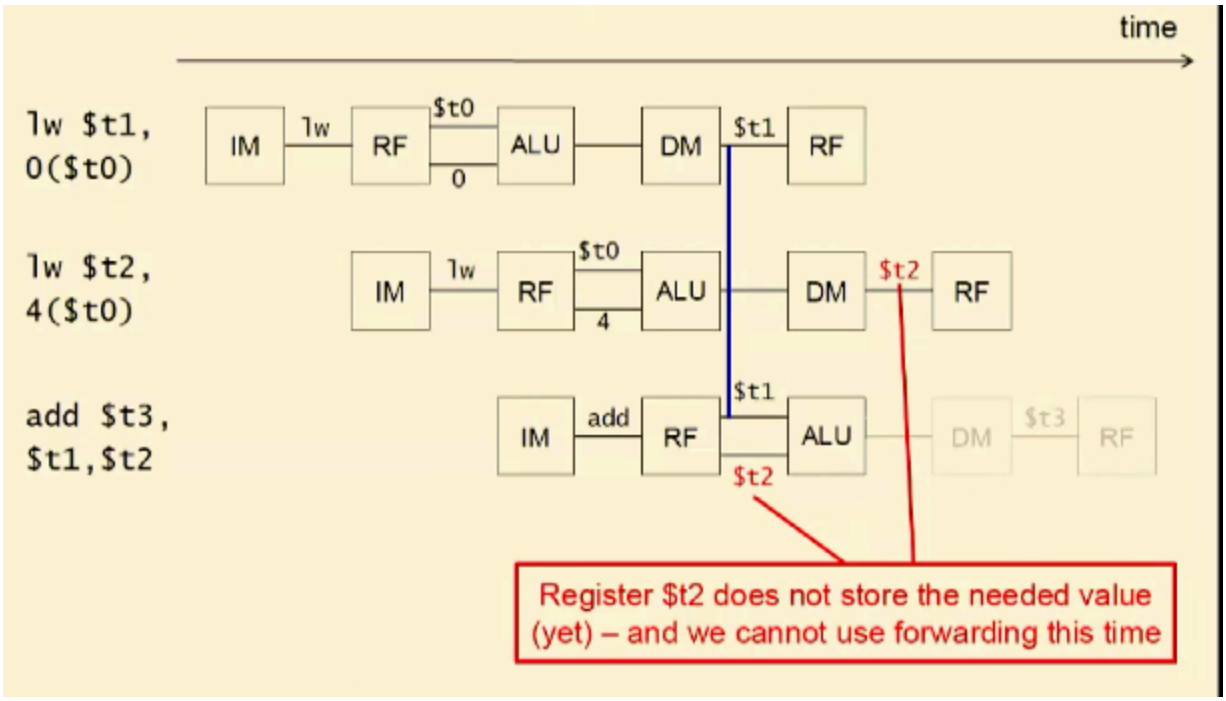

answer:

- **no**

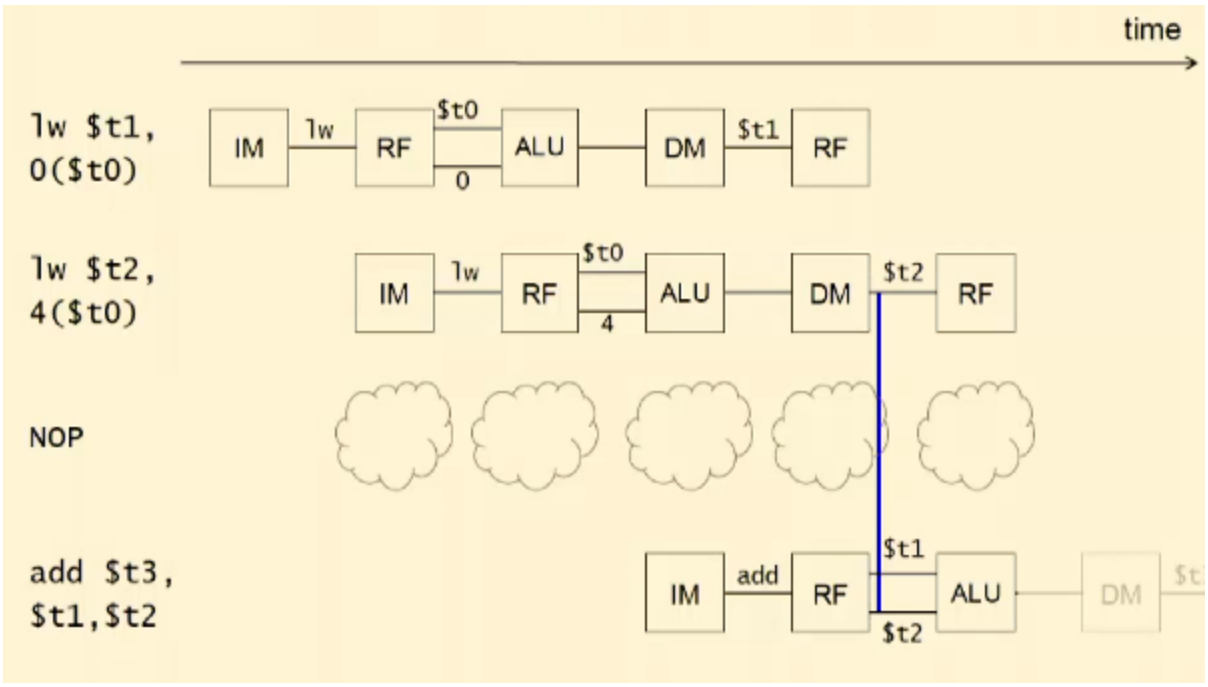
- third instruction is dependent on the first 2 instructions (raises a data hazard)
- register t1 has not stored the needed value yet by instruction 3
- 

- we can use forwarding to solve the data hazard for t1
- 

- however, register t2 also hasn't stored the needed value (yet) but cannot use forwarding

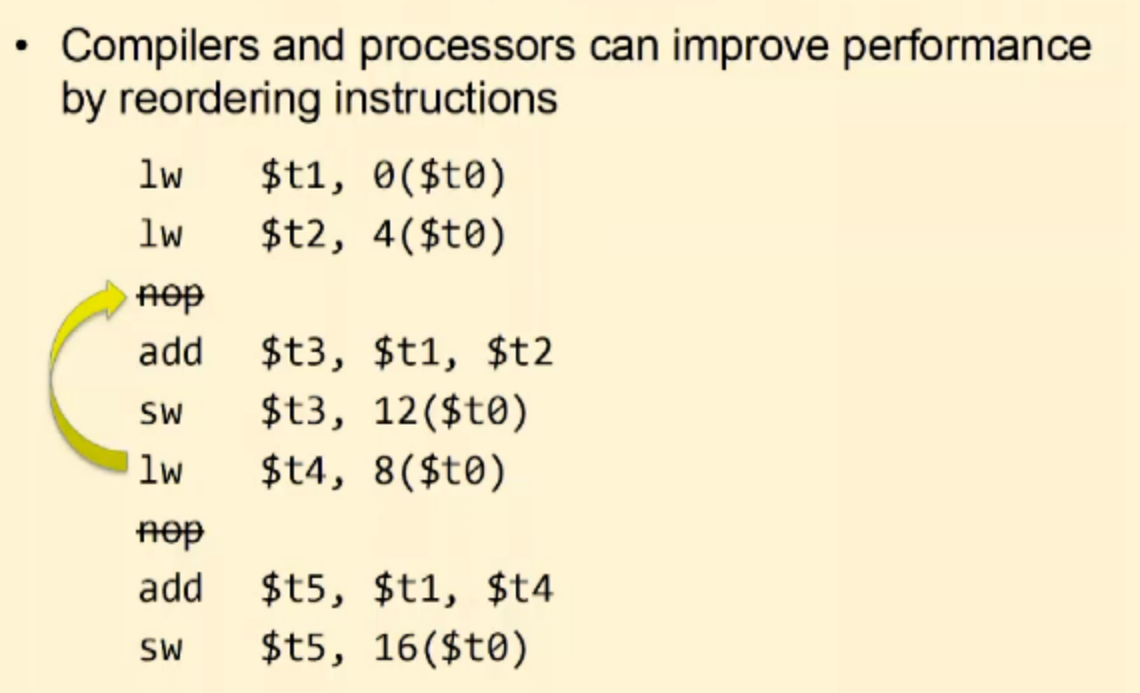



- it is not possible to forward as we cannot go back in time, hence we **stall the pipeline** where we delay teh execution of the third instruction by 1 clock cycle - allows the value to load to register t2 first before forwarding as as value to ALU (EX stage in 3rd instruction)




how to improve performance where we reduce number of stalls?

- reordering (change order which operations are executed)

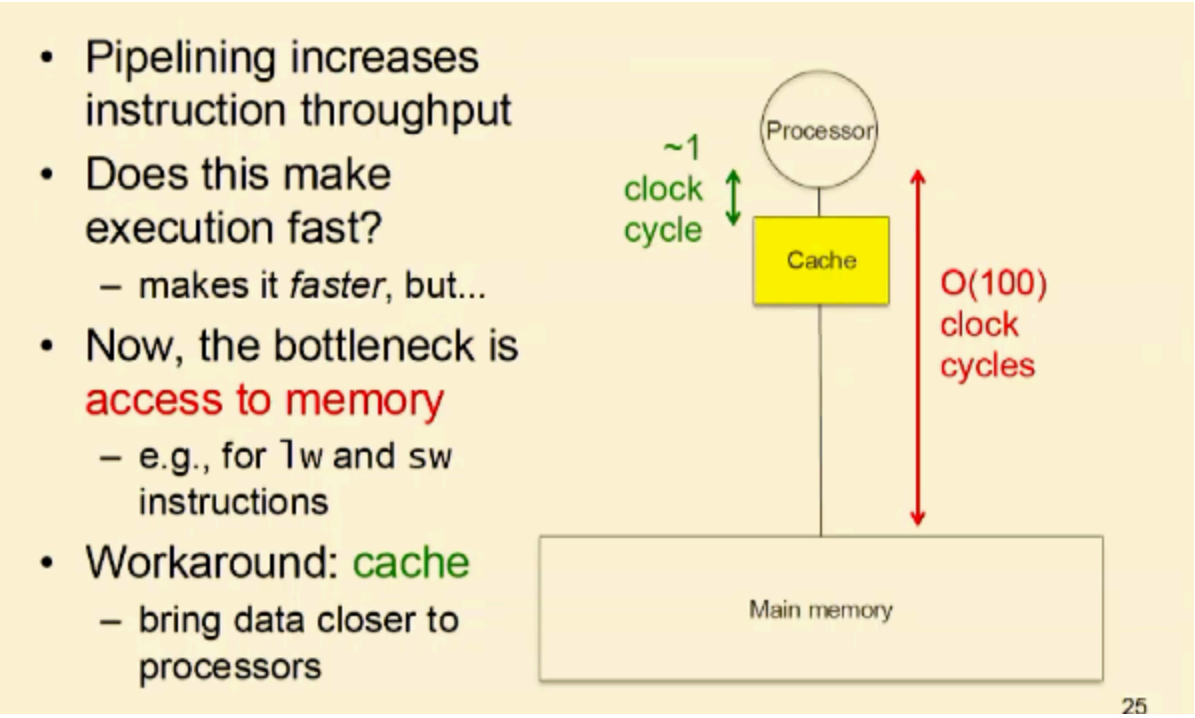



control hazards - occurs during branch operations (basically if statements)

- processor has to wait for evaluation of condition to understand what is in next instructions
- processor doesn't know which instruction to fetch
- procesors can do delay branching or predictions (predict the instructions to be executed, undo if prediction is wrong - can be static or dynamic predictions) - to solve control hazards

# Caching

It can take a long time to read from main memory




Caching allow data access to take place with just 1 clock cycle (provided they are already in the cache)

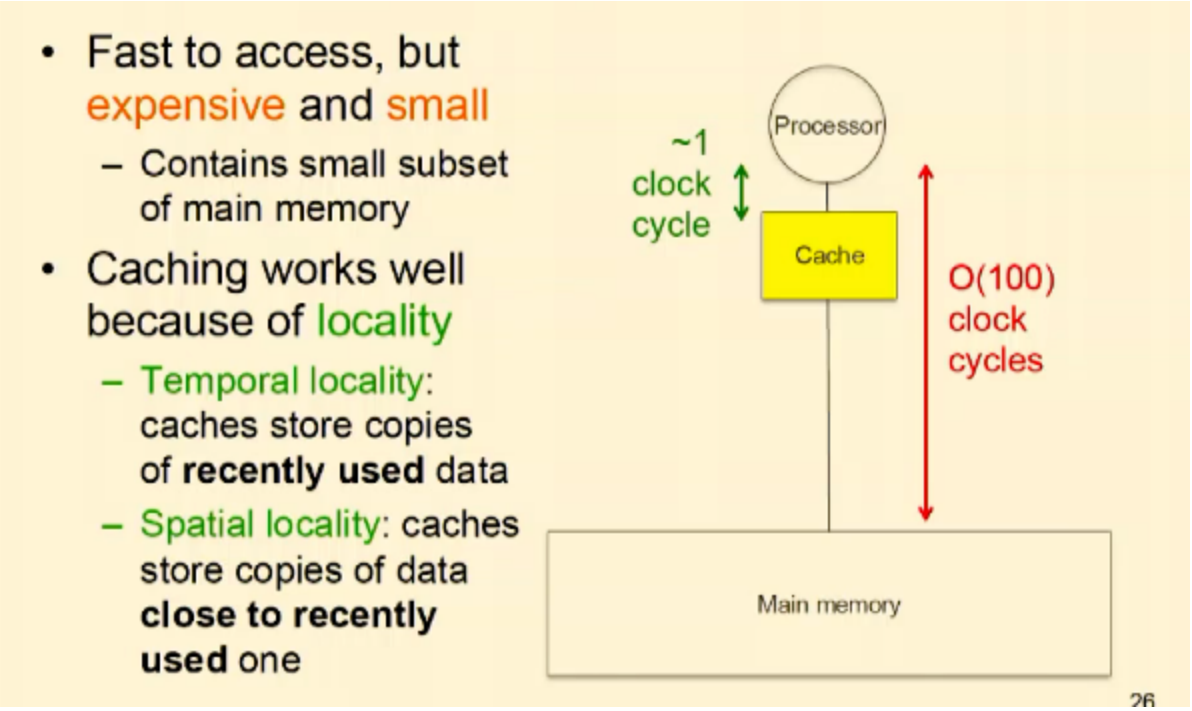
cahce - small portion of memory that is part of main memroy that is very fast to access (can be one clock cycle)

reading/writing from memory can take 1 clock cycle provided they are read/written from cache

data needed by programs is small (important/frequently used data can be accessed via cache instead of reading/writing it from main memory - you pay a miss penalty if you have to do that as more time is needed to access from main memory)

what data is loaded to cache

- data that is most often used by programs (temporal/spatial locality)




spatial locality can bring all the data close to each other from cache to processor (as data close together tend to be more likely used)

## Accessing data (in cache/not in cache)

If processor loads a word:
check cache first

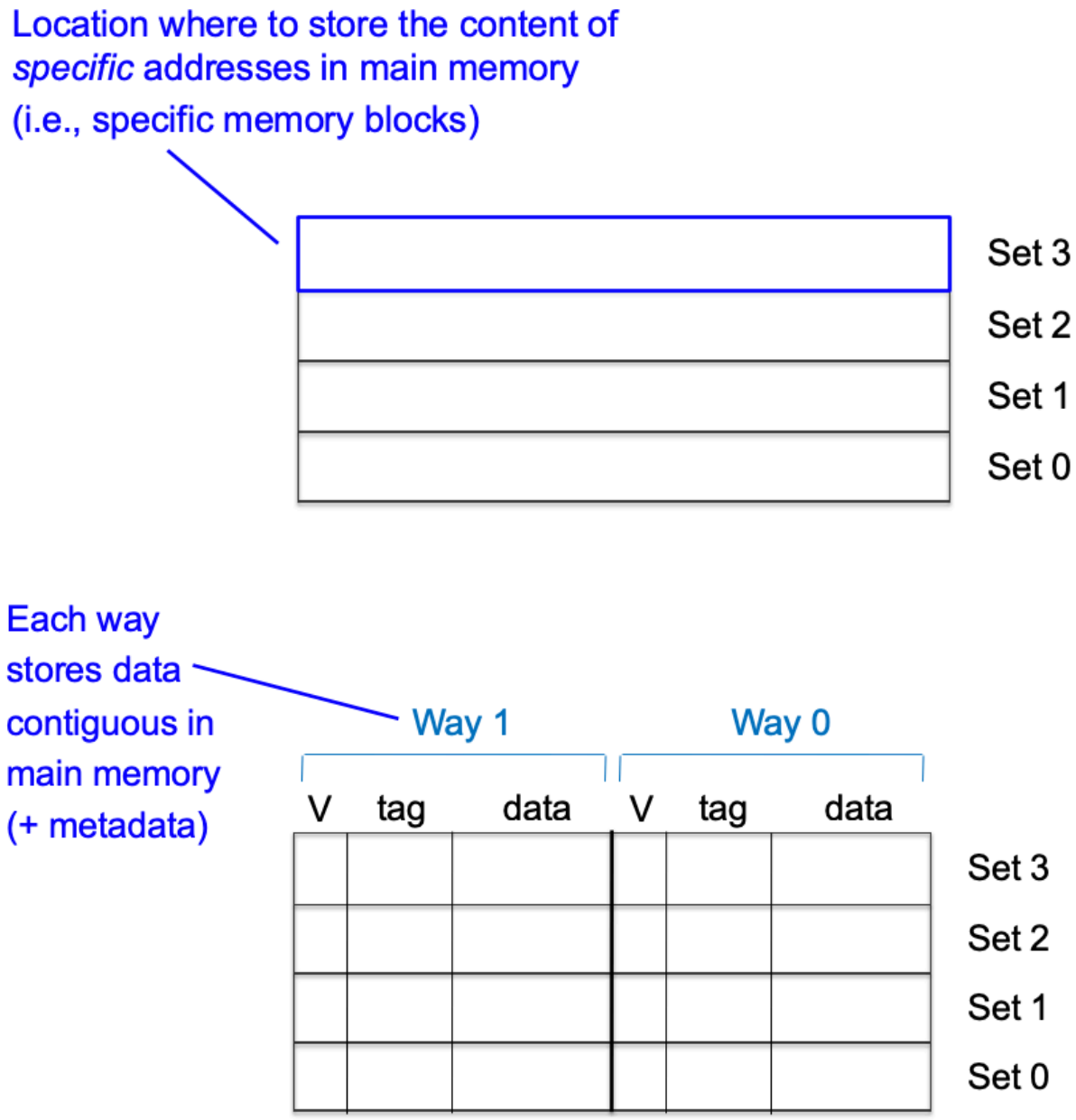
- if in cache (**cache hit**) - just return data in cache
- not in cache (**cache miss**) - retrieve data from main memory onto the cache and update the cache (miss penalty = time taken to retrieve from memory and update cache)

cache will update:

- **missed data** - temporal locality (likely to access the same data in future)
- **close memory locations** - spatial locality (would need content of memory close to the memory updated)

**cache internals**

- partition addresses in main memory into sets




tag - identifier of memory location that contains the data
validity - if the tag and data are valid (you set the valid bits to 1 after loading data to cache)

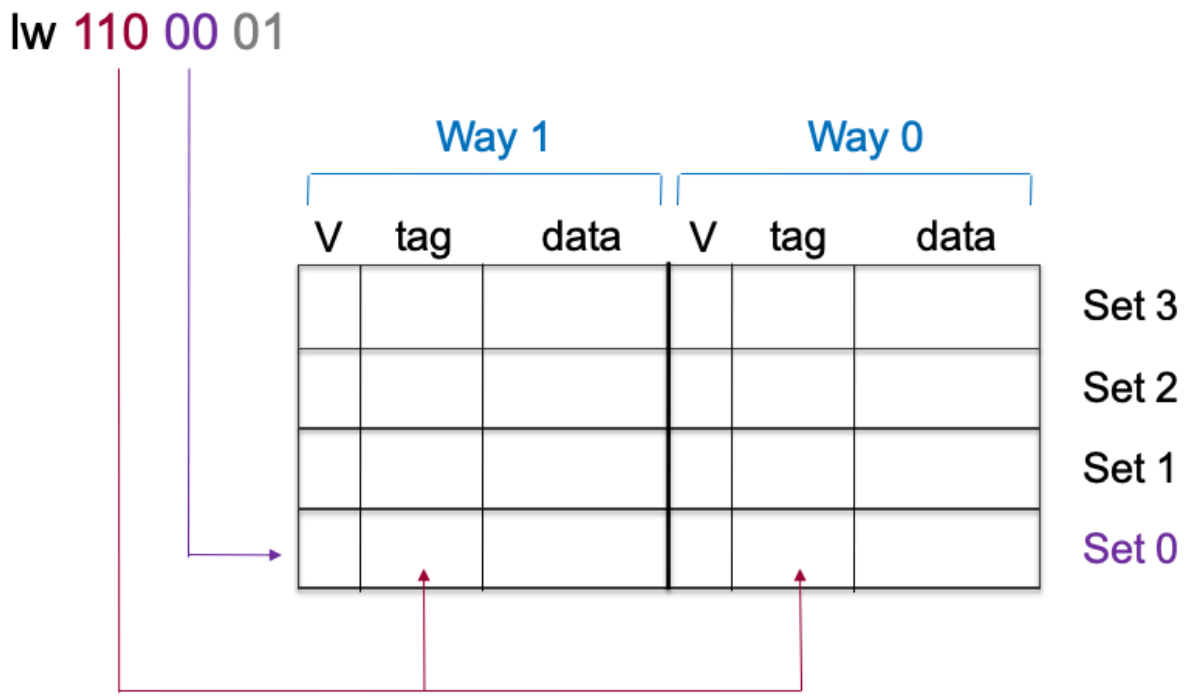
example: loading a word from memory
`lw 110 00 01`

- 110 - tag
- 00 - set
- 01 - byte offset

if cache is empty, then store the data corresponding to the memory address (set 0, tag 110)

so when loading same word again, processor knows where to find the word in cache via set/tag/byte offset

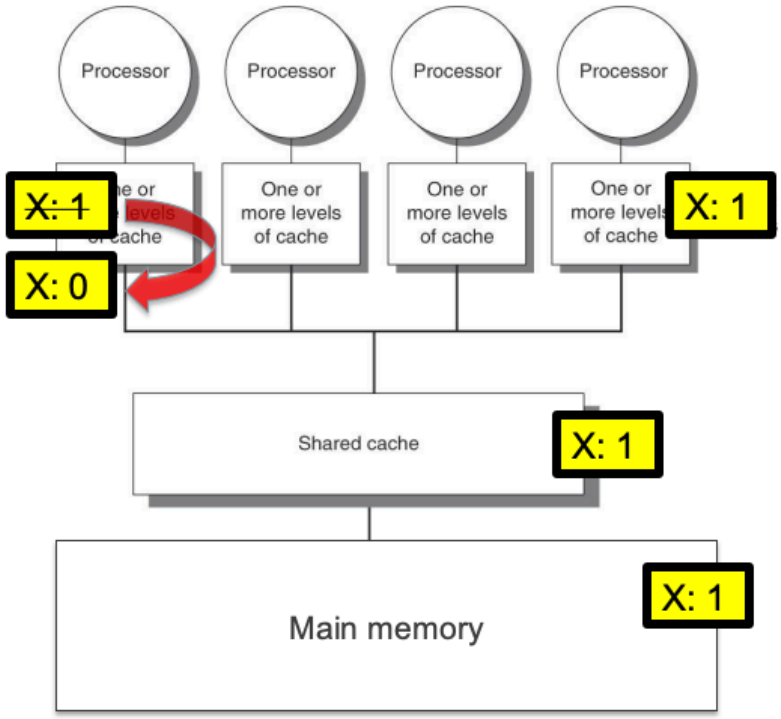
tag is used to understand if data stored corresponds to the memory address specified




you have to see all data in the specified set to be able to check the data exist in cache where you check the tags inside the set.

there are tradeoffs like dimensioning sets, having larger/smaller blocks where larger blocks => larger spatial locality => lesser cache misses BUT more time to read/write data, resulting in greater miss penalty

- Suppose four processors and the following events
  - t0: main memory location X stores 1
  - t1: processor 1 caches X = 1
  - t2: processor 2 caches X = 1
- What happens if X is modified in one cache at time t3?




42

inconsistency between x values - so when thinking about concurrent program, cannot assume that setting a value of variable will set the value of the variable in all caches (different locations might get different values depending on what is stored in cache/memory)

# Week 7 - Multi-core processors, threads and concurrency abstraction

(Concurrency in software)

## Introduction to Threads

### Example case: Designing a web service

Problem statement:

- We are asked to implement a server supporting Web-accessible functionalities
  - many clients can simultaneously ask for different services
- Processing clients' messages requires resources
  - some functionalities require long calculations and/or computations (CPU intensive)
  - other functionalities may need many memory reads/writes (I/O intensive) and network interactions (network intensive)

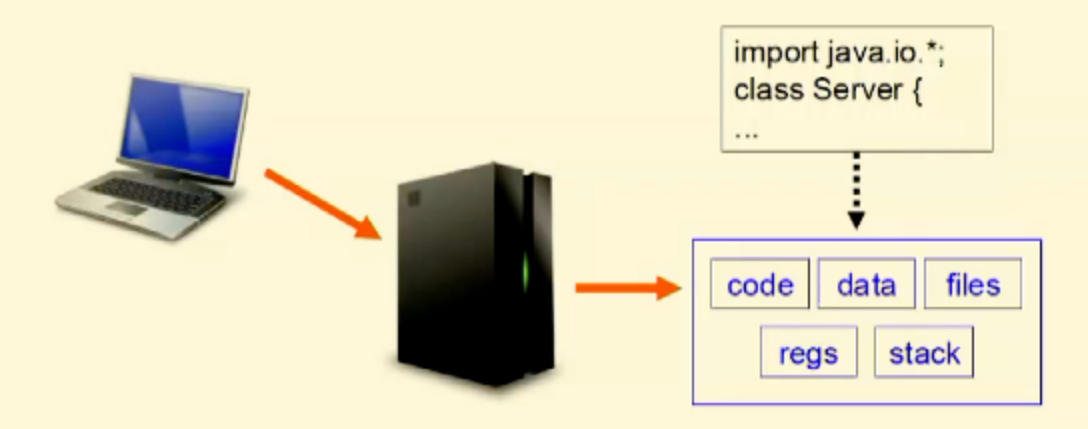
Possible designs for the server-side application based on the functionalities mentioned

**Design 1 - single process** - single-threaded applications
Example of a process that only uses one thread

Writing a monolithic program (self-contained, standalone application) and let OS execute it - OS will allocate resources to run the program by creating a process, separately to others running in server. The process will include:

- instructions to run the program - actual code needed to be executed
- pointer to file handler (used to open/close and write/read files)
- pointer to data (heap memory) during execution
- pointer to registers/stack - for function calls, local varialbes and execution context
- pointer to the code segment - where the executable code of the program is stored in.

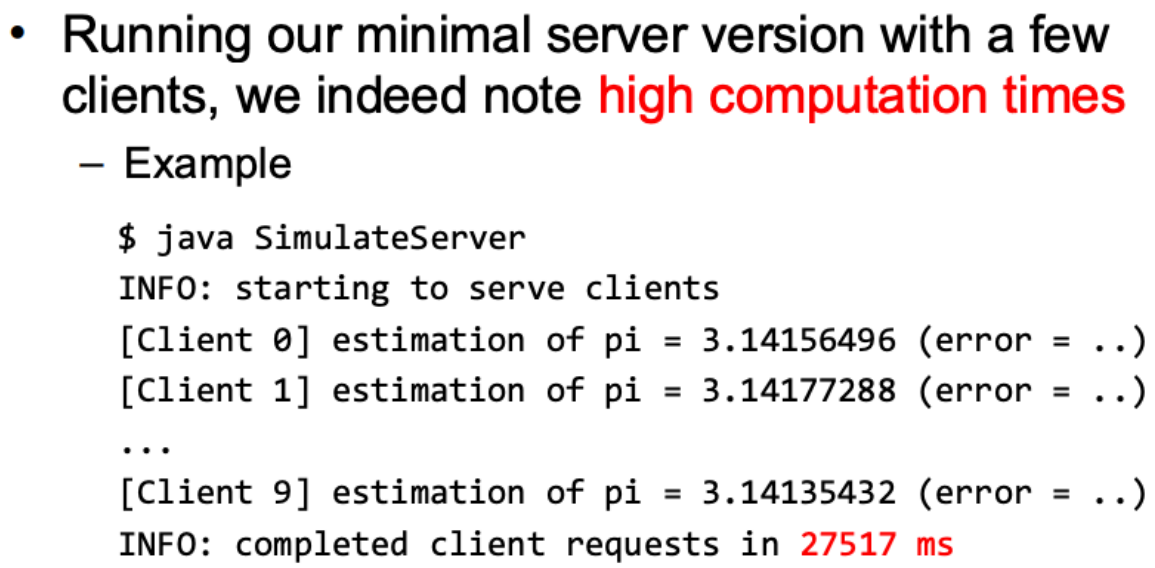
Basically OS will create a new environment to allow the program to run in isolation (no interaction between different process) where OS will handle potential overlaps between different processes




Client sends a message to our server which passes the message to the application running. Same thing happens every time there is a new client

problem:

- very slow, blocks at time, but no crashes/bugs/network bottlenecks
- reason is process could be busy handling another client (it only interacts with the new client once the process is done for the previous client)
  - example in lecture - server serving new clients where each client has to do the monte carlo method to calculate pi (takes a while to compute). in the example, clients have to wait in a queue for current process to finish in server, before the server serves the next client - clients are served one at a time (if there are too many clients - creating a bottleneck)
    - same computation is running over and over again
    - not possible to cache the results (each client might want its own computation - different computation for different clients)




- main problem is that processor is only using 1 of 64 cores of server machine (other cores remain idle) to run the server (thus only one client can have their request processed at the same time) - cannot process clients simultaneously. server is prevented from serving other clients as a result (underutilisation of cpu cores)

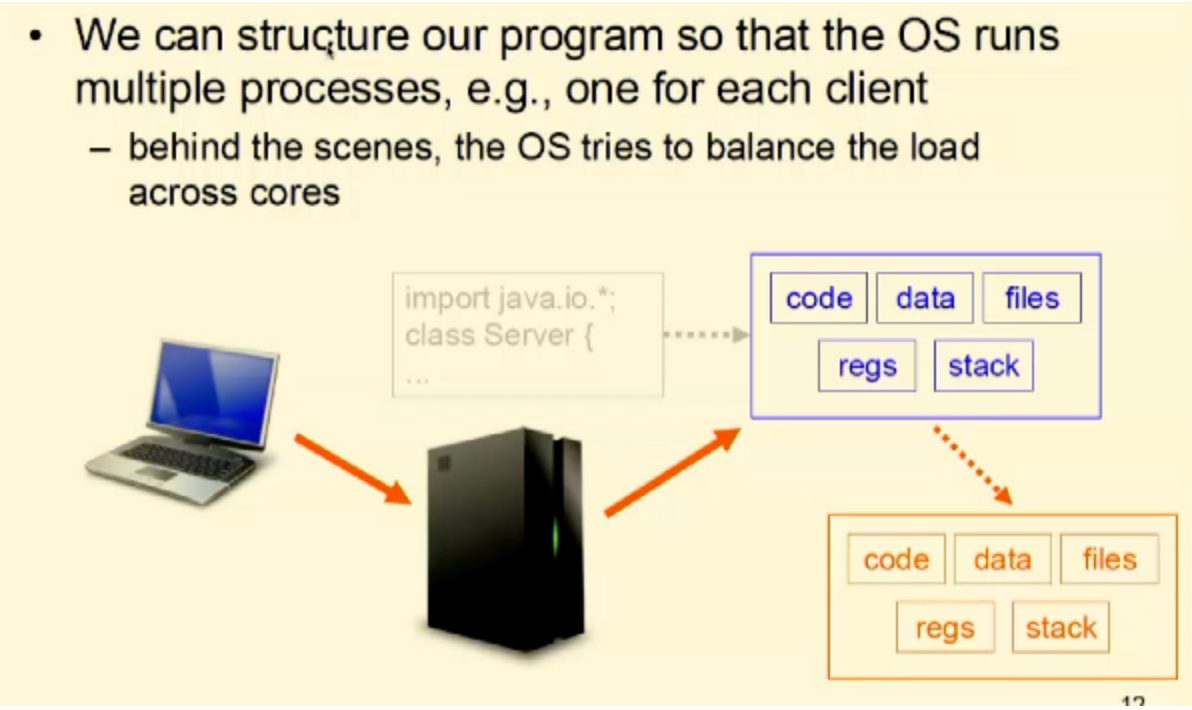
solution:

- serve all the clients at the same time by creating multiple processes

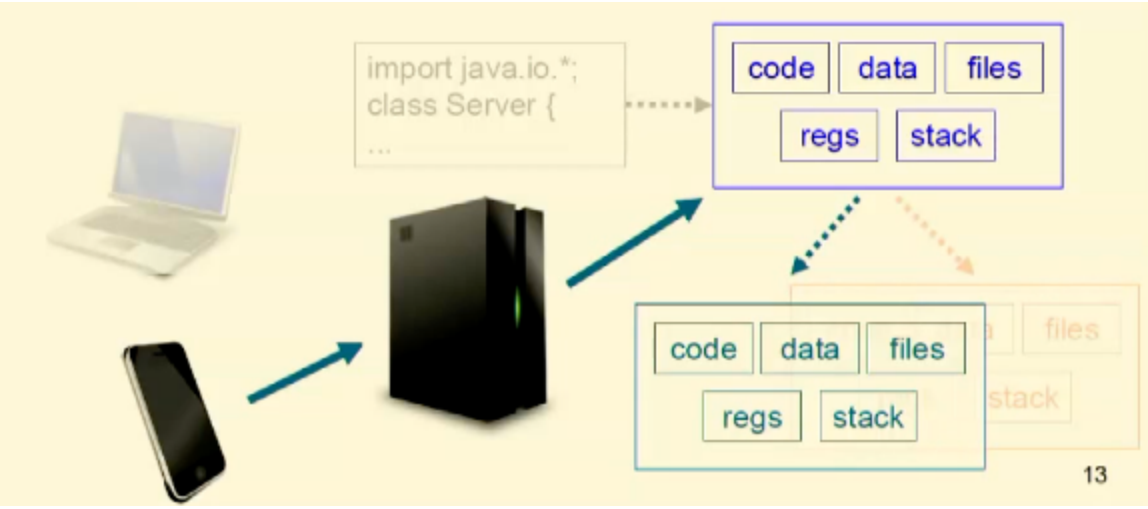
**Design 2 - multiple processes**
When server starts a main process is started, where main process can create new processes to handle a new client

Different processes are running at the same time (with different runtime environment separate from main environment). Each process has its own isolated environement to run an instance of

the program (i.e. the web service)




When there is a new client connecting to server, server passes message to the main process whcih then creates a new process to serve new clients




There is better utilisation of CPU cores (one core is allocated to process new processes created for each client)

example code:

```
import java.util.*;

public class MultiProcessServer extends Server {
  protected List<Process> spawnProcesses = new ArrayList<Process>();

  public void serveNewClient() {
    ProcessBuilder pb = new ProcessBuilder("java", "SimulateServer", "--num-clients=1", "--start-client-ID=" + Integer.toString(cur
rentClient++),"--quiet").inheritIO();
    try {
      Process newProc = pb.start();
      spawnProcesses.add(newProc);
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }

  public void stop() {
    try {
      for (Process p: spawnProcesses) {
        p.waitFor();
      }
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

When serving a new client, a new process is started to serve that new client where main process will keep track of the new process

example execution:

```
Pro:0008-demo ste$
Pro:0008-demo ste$ java SimulateServer --multiprocess
INFO: will run the multi-process version of the server
INFO: starting to serve clients
[Client 0] estimation of pi = 3.14168736 (error = 0.0030145986653876693)
[Client 2] estimation of pi = 3.14173848 (error = 0.0046417988083550869)
[Client 3] estimation of pi = 3.14120016 (error = 0.012493458989494348)
[Client 1] estimation of pi = 3.14144232 (error = 0.0047852666785667951)
[Client 4] estimation of pi = 3.1419932 (error = 0.012749788224425709)
[Client 5] estimation of pi = 3.14150432 (error = 0.0028117454913211736)
[Client 6] estimation of pi = 3.14174352 (error = 0.0048822269861878715)
[Client 7] estimation of pi = 3.14156 (error = 0.0010393960450497508)
[Client 8] estimation of pi = 3.14174952 (error = 0.0049932129178965 7)
[Client 9] estimation of pi = 3.14175816 (error = 0.0052682326595701 01)
INFO: completed client requests in 4075 ms

Pro:0008-demo ste$ java SimulateServer --multiprocess
INFO: will run the multi-process version of the server
INFO: starting to serve clients
[Client 0] estimation of pi = 3.14147248 (error = 0.0038252441689340623)
[Client 4] estimation of pi = 3.14140848 (error = 0.0058624274405 1236)
[Client 1] estimation of pi = 3.14166512 (error = 0.0019883675923199947)
[Client 5] estimation of pi = 3.14153504 (error = 0.0018338975209703757)
[Client 2] estimation of pi = 3.14157848 (error = 4.511593753807398E-4)
[Client 3] estimation of pi = 3.14166736 (error = 0.0023779788935015917)
[Client 7] estimation of pi = 3.14158384 (error = 2.80545276391217E-4)
[Client 6] estimation of pi = 3.14168024 (error = 0.0027879620264168797)
[Client 9] estimation of pi = 3.14128832 (error = 0.0096872390328 9086)
[Client 8] estimation of pi = 3.1417324 (error = 0.00444826639275404)
INFO: completed client requests in 3585 ms

Pro:0008-demo ste$
```

When creating multiple processes, order in which cleints are served is not guaranteed where first client has been served first (as processes are running in parallel in different processors - there can be different times for each core to process each client)

Advantages:

- Faster time to process all clients (better than processing one client a time)

Problem:

- Server is **over-utilised** - that design can lead to more than 100% of CPU usage (where different processes is running on different cores)
- Processes tend to be heavier than threads (threads are lightweight processes)

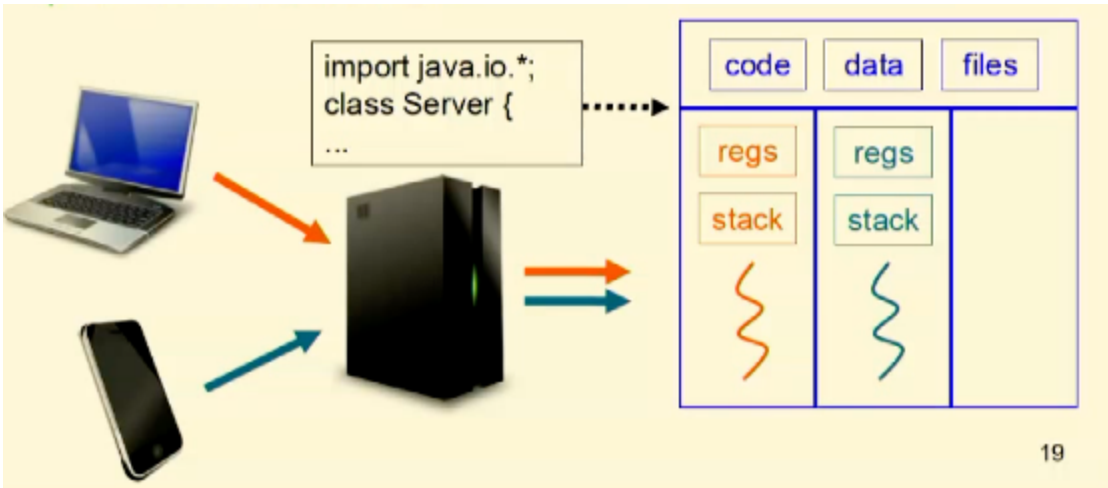
- OS has to manage different processes (and their allocated resources) at the same time where each process is running in isolation

Solution:

- Threads! (used for performing the same task or sharing the same process-wide resources)
- Have a single process with multiple threads

**Design 3 - multiple threads**
Refactor the program so OS can run a single process which spawns one thread per client

- a single thread is part of program (i.e. sequence of instructions) that programmer flags as runnable in parallel to other threads
- threads sha**re process-wide resource**s, but each thread can be scheduled independently of each other - thus multiple threads can run on different cores




Only one process containing code, data, files, registers, stacks (its own resources)

Whenever a request is received from a new client and is passed to server where server passes the request to the main process where it will create a new thread to handle that client's request

Note that all threads have access to same shared resources e.g. global variables.

Each thread has its own copy of registers and its own stack - as threads run in parallel and indepdently

Same code can be run by all threads created by the main process. But different threads can run different instructions within the code in each moment in time - hence the need for different registers/its own stack

Implementation:

```
import java.util.*;

public class MultiThreadServer extends Server {
  protected List<Worker> workers = new ArrayList<Worker>();

  public void serveNewClient() {
    String clientId = Integer.toString(currentClient++);
    Worker newOne = new Worker(clientId);
    workers.add(newOne);
    newOne.start();
  }

  public void stop() {
    try {
      for (Worker w: workers){
        w.join();
      }
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

Worker class is a thread - where whenever we want to serve a new client, new thread is created, and we keep reference to the new thread created and then start the thread

```
import java.util.*;

public class Worker extends Thread {
  private String id;

  public Worker (String clientId) {
    id = clientId;
  }

  public void run() {
    new ClientComputation().computePI(id);
  }
}
```

Worker class implementation (it inherits from Thread class - shows that Worker is a Thread, where it takes an id as a constructor)

example of execution (multithread vs multi process)
multithread:

```
CHECK "SimulateServer" processes 15:43.10
USER     PID     %CPU    %MEM     TIME     COMMAND
ste      69466   757.2   0.5      0:23.67 /usr/bin/java SimulateServer --multithread
```

multiprocess:

```
CHECK "SimulateServer" processes 15:28.22
USER     PID     %CPU    %MEM     TIME     COMMAND
ste      69170   72.1    0.5      0:01.39 /usr/bin/java SimulateServer --num-clients=1 --start-client-ID=6 --quiet
ste      69171   70.9    0.5      0:01.38 /usr/bin/java SimulateServer --num-clients=1 --start-client-ID=7 --quiet
ste      69169   70.3    0.5      0:01.41 /usr/bin/java SimulateServer --num-clients=1 --start-client-ID=5 --quiet
ste      69165   70.3    0.5      0:01.40 /usr/bin/java SimulateServer --num-clients=1 --start-client-ID=1 --quiet
ste      69173   70.2    0.5      0:01.33 /usr/bin/java SimulateServer --num-clients=1 --start-client-ID=9 --quiet
ste      69166   70.0    0.5      0:01.43 /usr/bin/java SimulateServer --num-clients=1 --start-client-ID=2 --quiet
ste      69172   69.9    0.5      0:01.37 /usr/bin/java SimulateServer --num-clients=1 --start-client-ID=8 --quiet
ste      69164   69.9    0.3      0:01.51 /usr/bin/java SimulateServer --num-clients=1 --start-client-ID=0 --quiet
ste      69167   69.0    0.5      0:01.41 /usr/bin/java SimulateServer --num-clients=1 --start-client-ID=3 --quiet
ste      69168   68.8    0.5      0:01.41 /usr/bin/java SimulateServer --num-clients=1 --start-client-ID=4 --quiet
ste      69156   0.0     0.4      0:00.10 /usr/bin/java SimulateServer --multiprocess
```

shows that threads are more lightweight and uses up less memory as compared to creating many processes (makes a huge difference esp when having lots of threads vs having lots of processes running simultaenously)

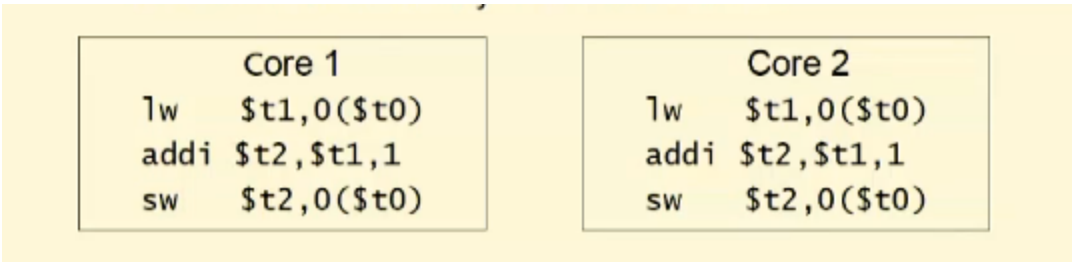
**Threads vs Process**
Threads:

- created within a process
- lightweight
- shares process-wide resources with other threads i.e. memory space, file handlers, data (heap), global variables, code segments but has their own PC, stack and local variables
- not completely isolated from each other
- requires explicit synchronisation

Process:

- runtime environment completely isolated from other processes
- has its own resources i.e. memory space, ifle handlers, stack, registers, heap and code segment
- can cause resource overutilisation (>100% cpu usage across cores)
- more exepnsive as context of entire process must be saved/restored (context switching)

**Problems**

- can result in unintentional bugs
- multi-threaded software is harder to write than single-threaded software - there is a cost when it comes to sharing resources
  - because multiple threads can cause unexpected interactions with underlying (performance-optimised) hardware
  - example is when there is two threads that are running on different cores on different porcessors which are doing the same thing
  - 

  - what happens if you access/run the same code segment at the same time as they might run in parallel
  - they are accessing the same shared variable in memroy
  - Both cores/threads execute the same code segment simultaneously
  - Both threads read the original value from memory before either has updated it
  - Each thread performs its addition independently, unaware of the other thread's operation
  - Both threads calculate the same new value (original + 1)
  - The second thread's write operation overwrites the first thread's update

    - The end result is that only one increment happens, even though two increment operations were executed
    - these 3 step operation isnt atomic
    - in general, sharing resources (e.g. variables) canm create risks of incorrectness (think about data hazards in ILP)

**Solution**:

- synchronisation of threads - to ensure correctness of code especialaly during parallel processing with multi-threading

NOTE 1:
Multithreading is never the best design as there is a tradeoff between simplicity, correctness and performance (you will want to target the easiest design that fits your requirements)

- good way is to start from single threaded, single process application
- if there is a need for multi threading then refactor the app as a multithreaded application
- don't jump to opstimisation as it is harder to develop multithreaded application as compared to a single threaded one
- concucrrency bugs can affect simultaneous execution of code sharing resources (race conditions)
    - e.g. by multiple resouirces, on memory/files they share

NOTE 2:

- multithreading support depends on programming langugae (e.g. python doesn't support true parallelism by default - as it is simpler to guarantee safety with a single theraded application or an app that runs 1 thread at a time)

# Concurrency Abstraction

Concurrency abstraction - model to show how hardware works in multi-threaded applications we might not know what is the specific hardware/software that is running our application

**programmer's viewpoint**: hardware can **unpredictably interleave actions from all threads**

Each thread corresponds totally ordered sequence of atomic actions (which are indivisible in terms of processing)

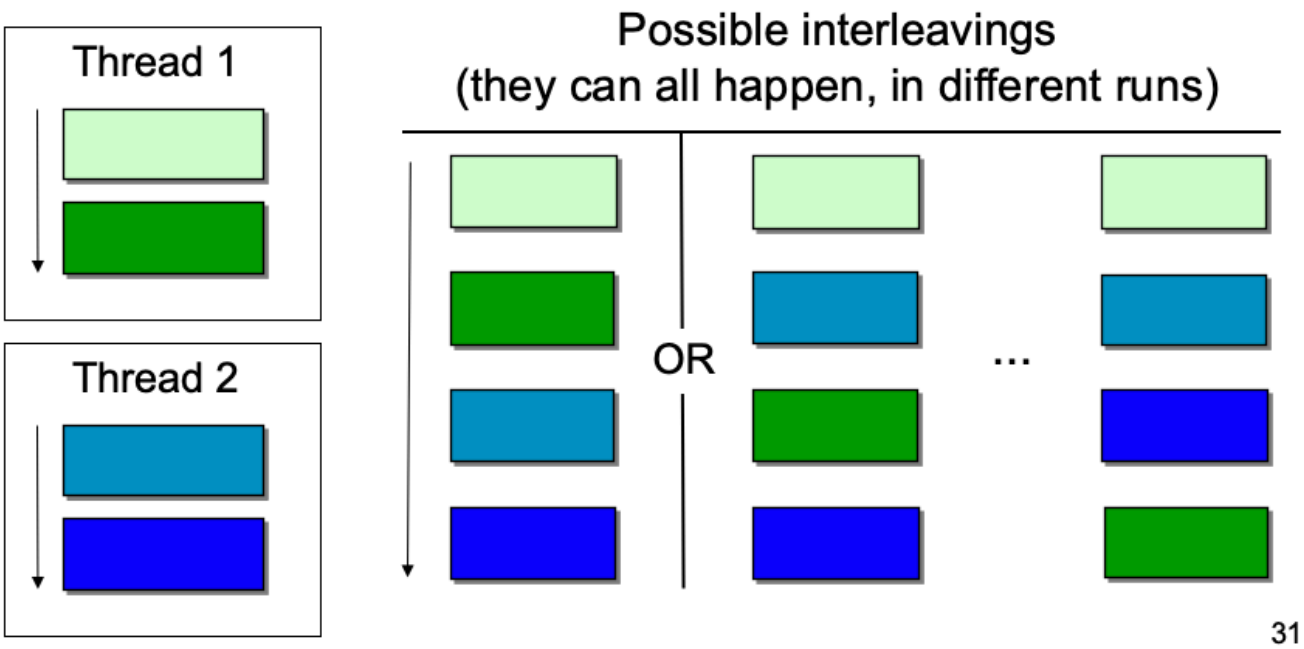
**Interleaving** - totally ordered sequence of executed actions, across all threads where:

- basically an execution of multi-threaded application with multiple threads
- only 1 action at a time is executed

- actions from same thread are executed in order defined by the thread itself - **we do NOT reorder actions within threads**
- overall sequence of executed actions can **arbitrarily mix actions from different threads**
- shows the unpredictability in execution order which shows the importance of synchronisation to solve race conditions

illustration:
consider two threads (both threads made by their own atomic actions - the order in which actions are defined in the thread where second atomic action starts executing only after the first atomic action is done - **ORDER MATTERS**)

**possible interleavings when running the two threads in paralllel**




**first interleaving:**
first action from first thread -> second action from first thread -> first action from second thread -> second action in second thread

**second interleaving** - comes from the idea that we can mix actions between different threads as per the concurrency model:
first action from first thread -> first action from second thread -> second action from first thread -> second action in second thread

**third interleaving :**
first action from first thred -> first action from second thread -> first action from second thread -> second action from first thread

note that: first action of first thread MUST finish before second action of first thread and so on, but we mix atomic actions in all possible ways when finding out possible interleavings

this model shows how the different atomic actions of different threads are executed in practice

Is the abstraction correct?

- we need to dfine the scope of the model (we want to model problems arising from actions from different threads (e.g. executed from different cores))
  concerns:
- we do not model time, hardware detials (e.g. instruction reordering, caching, thread-to-core allocation, ..

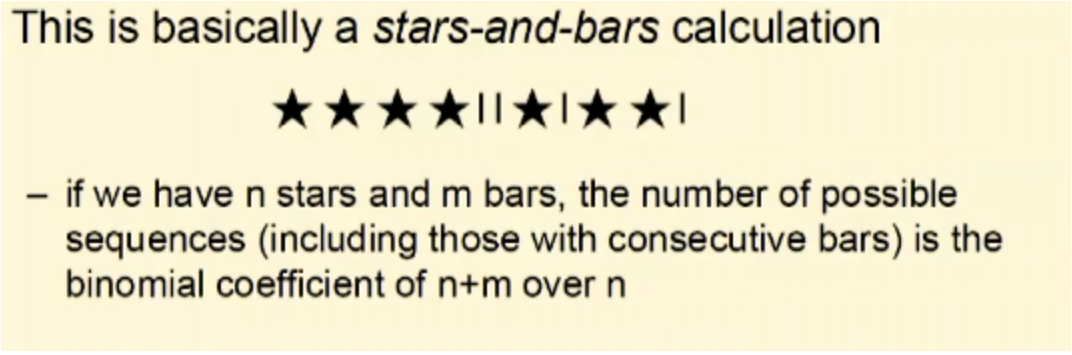
abstraction is correct as the reserving observation:

- only 1 processor at a time can access shared resources - hardware imposes **total order**
- two actions cannot finish at the same time
- hardware also guarantees correctness per thread
  - each thread is guarnateed to provide consistent results which is always the same (guaranteed by hardware)
  - want to guarantee correct result is achieved
- no bad interleaving == no misbehaviour for any duration of individual actions

**Building upon concurrency abstraction**

- Concurrency abstraciton helps understand/solve concurrency bugs via primitives or programming languages (e.g. Java)
- Depends on the constructs of programming languages used
  - allow us to define the code blocks aka what are actual atomic actions that we can execute in multithreaded application
  - restrict the set of interleavings where we remove interleavings that we don't want via keywords
- It also defines the guarantees the constructs provided by the programming language

**How many possible different interleavings ?**
Proportional to binomial coefficient (stars and bars calculation)



Calculation:

- Given two threads A and B where A has X atomic actions and B has Y atomic actions
- FOrmula: $\frac{(X+Y)!}{X!Y!}$

Number of interleavings grow very fast if X or Y increases or if there are more threads

Combinatorial explosion of interleavings can lead to unpredictability and non-determinism

- can lead to many possible results that are hard to enumerate for large programs
- some interleavings may be more likely than others: concurrenmcy bugs are hard to uncover even with testing
- other interleavings can happen but very rarely -> concurrency bugs are latent

# Additional Material

## Thread Level Programming

- Run multiple threads of execution in parallel via multicore arhitecture, with different instruction streams to keep individual cores/resources busy
- threads can come from:
  - different processes scheduled by OS onto different cores
  - from a single multithreaded program running
- consequences:
  - potential for better throughput
  - need to start designing multithreaded code

## Processes vs Threads

From additional material

**Processes**
Instance of computer program being executed and consist of

- one address space containing memory segments
- shared I/O resources
- multiple threads
  - each with their own set of registers, i.e. PC, stack pointer and general purpose registers
  - each has their own stack in address space
  - **BUT** machine code (text segment) and data (static/dynamic data segments) are shared

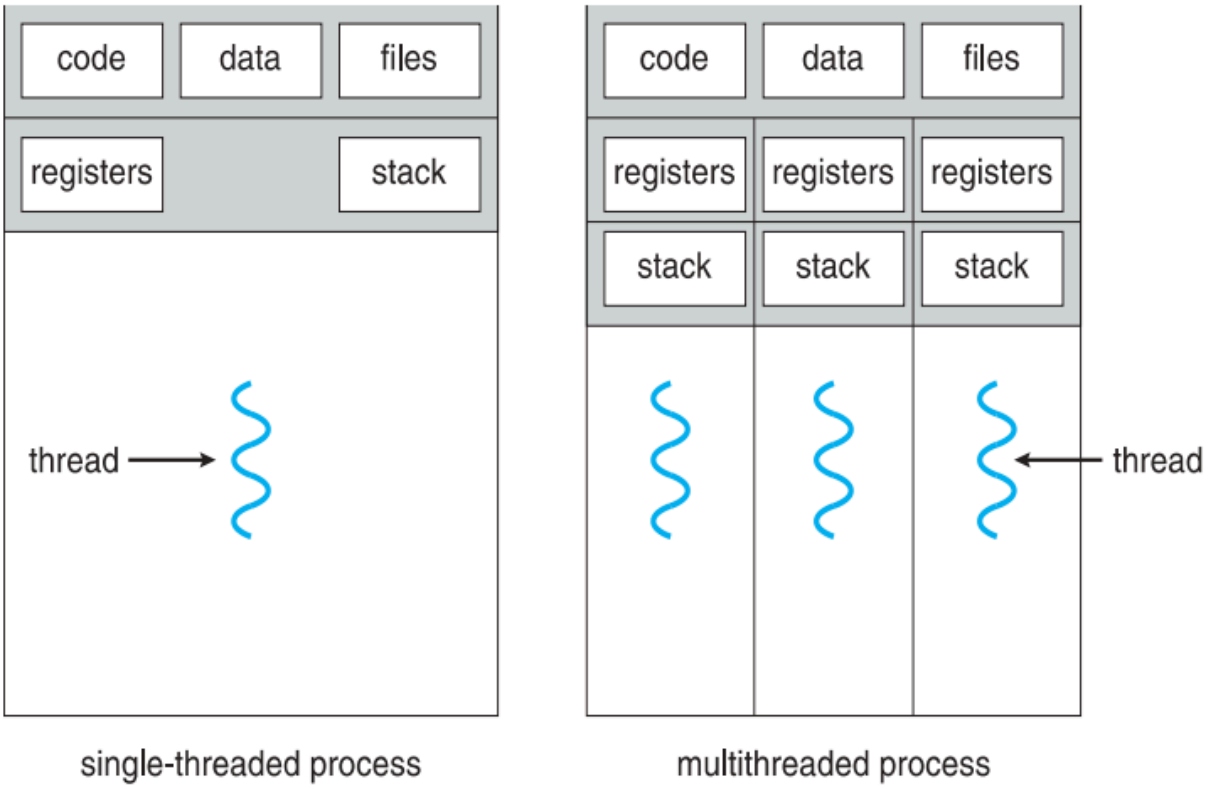
**Threads**

Threads in same processes **share**:

- code
- heap space memory

BUT each thread has its own copy of:

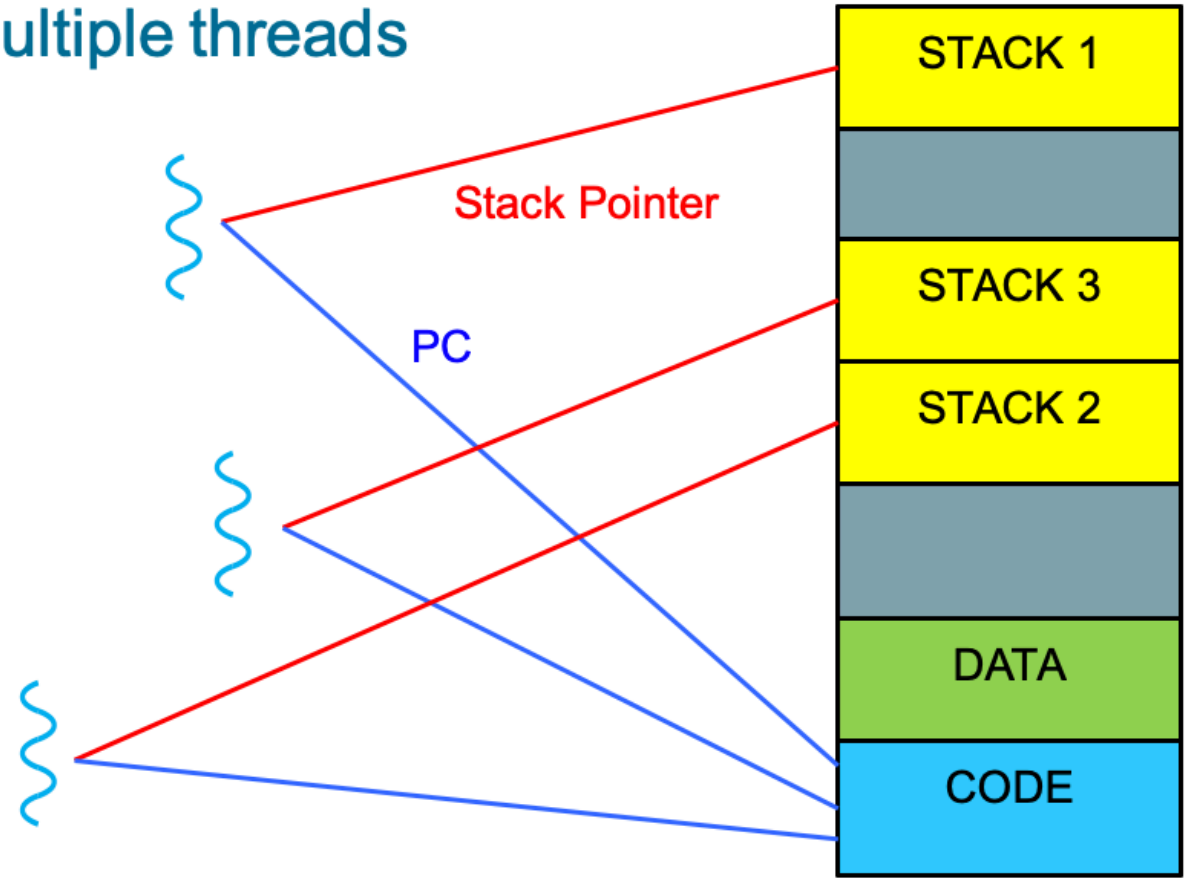
- program counter register, fro location of current instruction in running program
- other registers, for local variables being worked upon
- stack space memory, for procedure call parameters and other local variables




single-threaded process

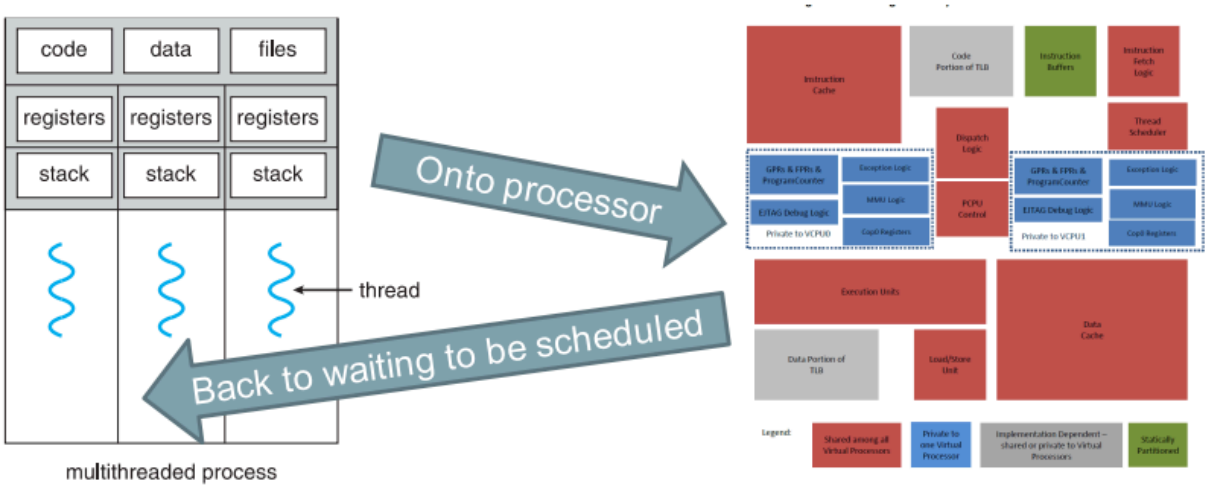
multithreaded process

# Memory layout with multiple threads



**OS support for multithreading**

- Context switching - scheduling different threads for execution, especially if a thread "stalls" so OS can switch to run another non-stalled thread
- Performed via interrupts, in the OS kernel
- To schedule threads, OS stores info on all threads and processes via:
  - process control block - contain all info about each process
  - thread control block - contain all info about each thread (i.e. current register values, etc)




# Week 8 - Threads and thread synchronization in Java

(Concurrency in Java)

# Thread Safety

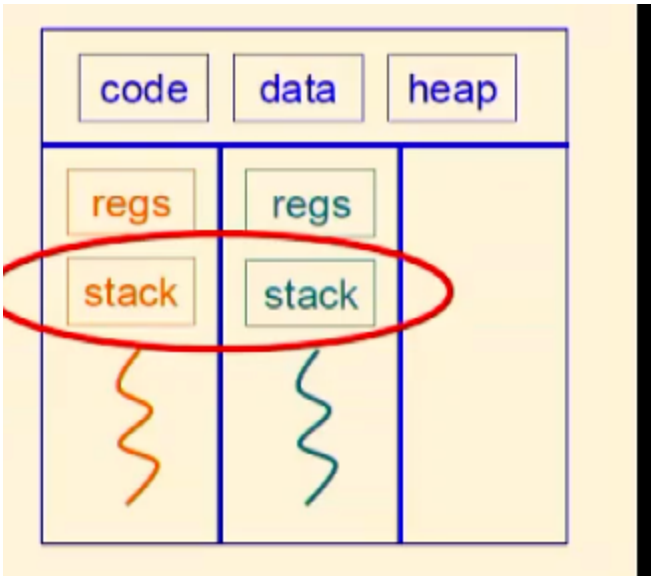
Thread Safety - correctness (never in invalid state) irrespective of interactions (interleavings) between threads

- where correctness (formalised as pre/post conditions, invariants) implies that there are NO race conditions
- it is always valid

Module definition of thread safety:
An application (respective class) is **thread safe** if it allows the **same set of outputs** (respective states) irrespective of the number/timing of threads. Same outputs as if the application is singlethreaded, while you get the same results even with multiple threads and all possible interleavings

Threads DON'T share local variables (it stores them in its own stack), thus they are not accessible by other threads

It is important to know what is shared/unique between threads (code/data/heap is shared, registers/stack/local variables are unique to threads)




# Race Conditions

Race condition: incorrect computations (e.g. invalid state) in specific interleavings (e.g. unlucky timings) - correctness is bwoken

# Synchronization

Key observation - multiple threads have access to the same variables

**Problem**: shared variables are modified by different threads that run concurrently

- results in inconsistent modifications of instance/shared variables (state variables)

**Solution**: coordinate threads by enforcing atomicity of actions on **shared variables** (mutable states) by synchronisation

- access to shared variables can be serialised so that only one thread can access at a time
- sequences of operations that must be atomic to ensure thread safety are called **compound actions**

Lock is a way to ensure synchronisation where if a thread acquires a lock (linked to the object) before executing a method denoted by `synchronized`, other threads trying to execute the method have to wait for the lock to be released, i.e. that thread to finish executing the method

What do we need to protect by synchronisation?

- **any access to mutable state/instance variables** so multiple threads can't touch the same variables at the same time - they MUST be properly synchronised with same lock to maintain thread safety

In concurrency, we MUST protect the state of the object

- Object state: internal data that affects the object's externally visible behaviour
  - typically instance and static fields
  - can also include fields from dependent objects
- threads that share/arbitrarily modify state can cause concurrency problems called **interfenrece**
  - e.g. state inconsistently mixes updates from 2 threads
- we MUST synchronise threads' access to shared mutable state; otherwise program is broken
  - general rule: whenever multiple threads access a given mutable state variable, we MUST coordinate their access to it using synchronisation

REtrofitting thread-safety is hard

- it involves evlauting all possible accesses to any shared resources by any set of spawn threads where it requires checking all code paths and interleavings
  Design with thread safety in mind where:
- for each state variable, consider if:
  - it needs to be shared
  - it must be mutable
  - access to it MUST be synchronised
- we face tradeoffs between correctness, simpklicity and performance
  - e.g. easier to use synchronisation everywhere when writing multithreaded code BUT can hurt performance and even correctness

From Claude AI - Locking

When you declare a method as `synchronized`, like:

```java
public synchronized void myMethod() {
    // method body
}
```

It's equivalent to:

```java
public void myMethod() {
    synchronized(this) {
        // method body
    }
}
```

In this case, `this` refers to the current object instance that the method is being invoked on. The thread that calls this method must acquire the intrinsic lock of that object before executing the method, and it releases the lock when the method completes (whether normally or by throwing an exception).

This is why synchronized instance methods of the same object cannot execute concurrently - they all use the same object (`this`) as their lock. However, synchronized methods of different object instances can run concurrently since they're using different locks.

It's also worth noting that for static synchronized methods, the lock is the Class object rather than any instance, since static methods don't have a `this` reference.

It's equivalent to:

```java
public void methodA() {
    synchronized(this) {
        // code block A
    }
}

public void methodB() {
    synchronized(this) {
        // code block B
    }
}
```

In both cases, the same lock (`this`) protects both code blocks. This means:

1. If thread T1 is executing methodA and thread T2 tries to execute methodB on the same object instance, T2 will be blocked until T1 completes methodA and releases the lock.
2. Two threads cannot simultaneously execute any synchronized methods that use the same lock (`this` in this case) on the same object instance.
3. However, threads can concurrently execute synchronized methods on different object instances, as each instance has its own intrinsic lock.

This is a fundamental mechanism of Java's synchronization model for ensuring thread safety when accessing shared mutable state within an object.

If you're synchronizing different sections of code using `synchronized(this)` blocks rather than making entire methods synchronized, the same principle applies - these blocks are protected by the same lock.

For example:

```java
public void methodX() {
    // non-synchronized code

    synchronized(this) {
        // critical section 1
    }

    // more non-synchronized code

    synchronized(this) {
        // critical section 2
    }
}

public void methodY() {
    // some code

    synchronized(this) {
        // critical section 3
    }
}
```

In this code:

1. Critical sections 1, 2, and 3 are all protected by the same lock (the `this` reference).
2. Only one of these critical sections can execute at any time, even if they're in different methods.
3. If thread T1 is executing critical section 1, thread T2 cannot execute critical sections 2 or 3 on the same object until T1 releases the lock.
4. The non-synchronized portions of code can run concurrently with any other code.

This approach improves performance by:

- reducing scope of synchronised blocks to only critical sections that need locking protection
- allow non-critical code to run concurrently
- minimising time that thrads hold locks

# Locking

Synchronize keyword enables use of Java implict locks that enforce **mutual exclusion** where:

- blocks of instructions guarded yb same lock cannot be exercuted by more htan 1 thread at a time
- basis for supporting synchronisation policies

Useful design practices for locking:

- **encapsulation/data hiding** - internal state of an object is directly accessible only by the object itself (cannot be accessed by outside of the object where no instance variable is declared **public**)
- **immutability** - make a variable immutable when their variable should not change over time
- **documentation** - especially of invariants

# Book Examples

```
public class SimpleFactorizer implements Servlet
{
  public void service(ServletRequest req,
                      ServletResponse resp) {
    BigInteger i = extract(req);
    BigInteger[] factors = factorize(i);
    encodeInResponse(resp, factors);
  }
  ...
}
```

Stateless servlet as no info is carried from 1 request to following request
Servlet extracts integer from request and factorise the integer

- not possible for different threads to alter variables (as they are local)
- this SImpleFactorizer servlet is **thread-safe** (result is correct irrespective of interleavings)
- the biginteger variables / factors are local and only accessible by that thread (other threads cannot access that)

**Iteration 2 - adding application-level caching**
Try and improve server's performance as factorising is expensive operation - avoid factorising a number if that same number has recently factorised where we remember the last factorisation in a 'cache'

We have to add instance fields to our servlets

```
public class CachingFactorizer implements Servlet {
  private BigInteger lastNumber;
  private BigInteger[] lastFactors;
  ...
```

```
  ...
  public void service(ServletRequest req,
                      ServletResponse resp) {
    BigInteger i = extract(req);
    BigInteger[] factors = null;
    if(i.equals(lastNumber)){
      factors = lastFactors.clone();
    }
    if(factors == null){
      factors = factorize(i);
      lastNumber = i;
      lastFactors = factors;
    }
    encodeInResponse(resp, factors);
  }
}
```

Don't need to factorize

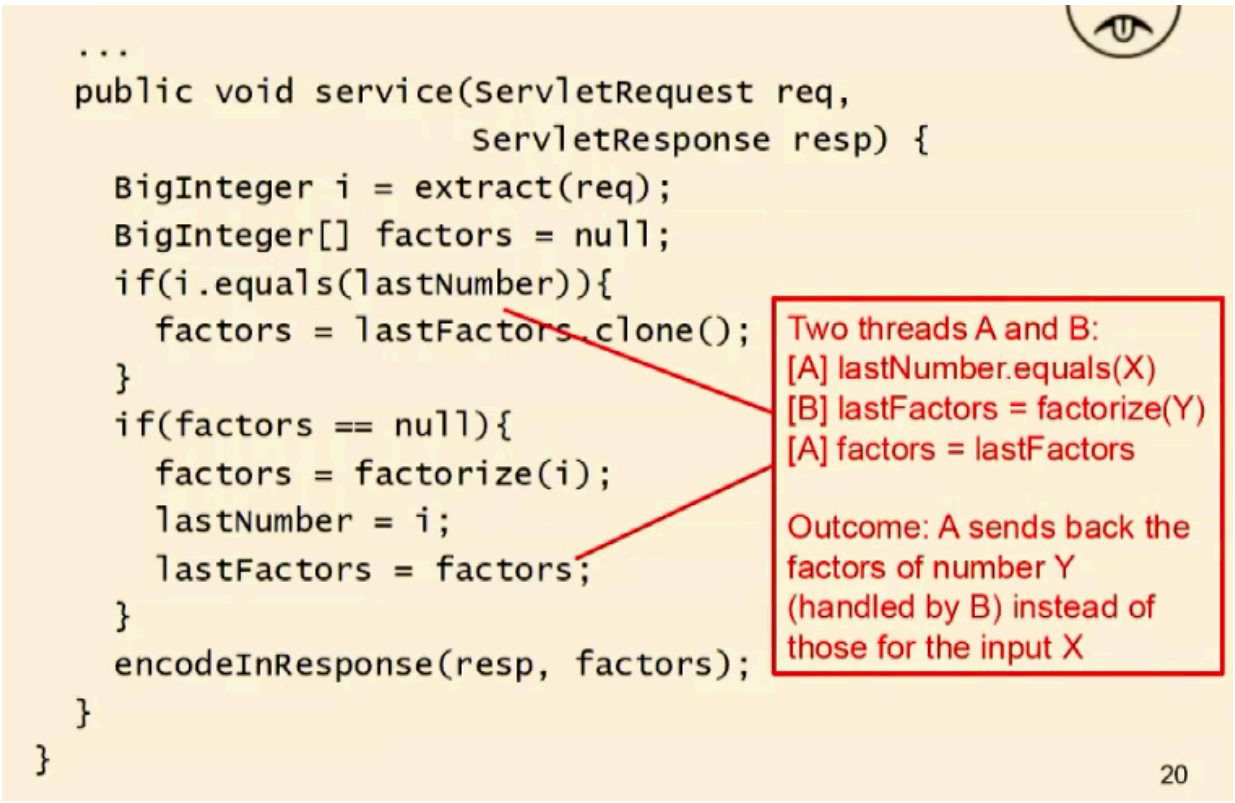
Need to factorize, then update instance fields

16

Still extract number from current request where we check if that number was factorised previosuly, if yes then no need to factorise, just get from instance variable, otherwise factorise and update the instnace variables

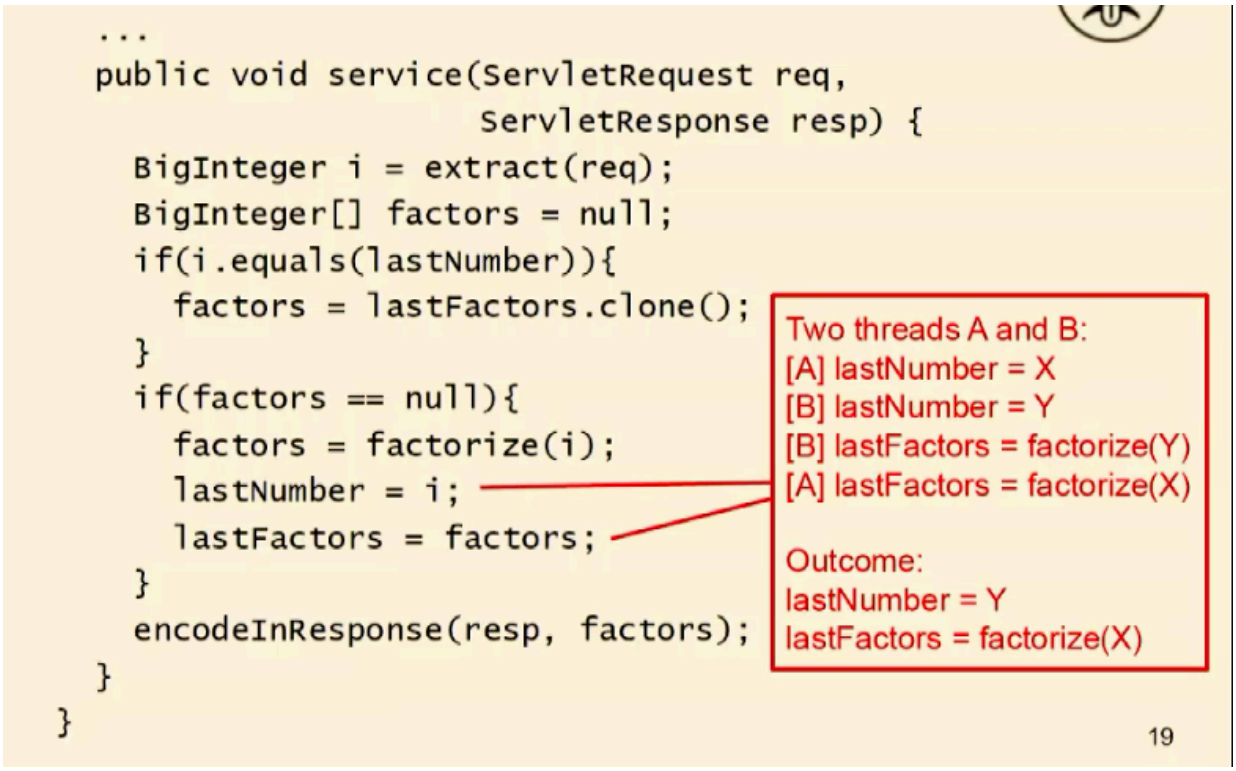
is CachingFactorizer thread safe?

- NO
- there are instance variables (non-local) used by the methods that are accessed/shared by multiple threads
- doesn't guarantee any of the 2 post conditions: product of lastfactors equal last number, each response encodes the factors for the number in the corresponding request
- raises a race condition:




- first thread checks if number equal to last number (check is correct so clones the last factors to be encoded)
- in the meantime, second thread already factorised number and updated the last factor to the factor it has computed, right after first thread does its check (state changes after B runs `lastFactors = factorize(Y)`)
- so first thread will clone the last factors comptued by second thread - encoding the factors computed by B (WRONG ANSWER), not the last number checked (as state was changed before cloning)

- another race condition:



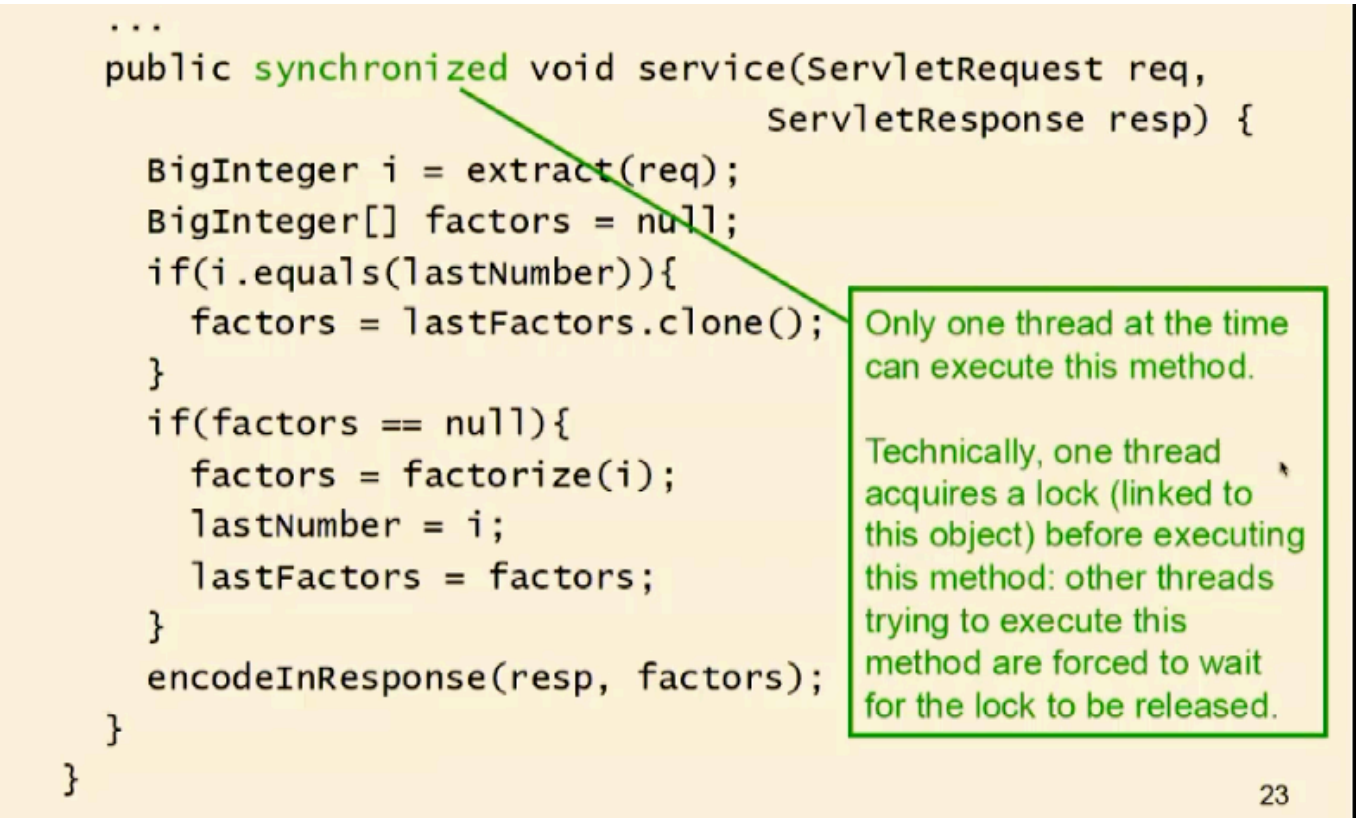

- when both threads are in the same block of code so theyre overwriting lastNumber/lastFactors (Interleavings where lastNumber gets overwritten by B, and lastFactors ovewrritten by A after lastFactors gets factorised in the B thread)
- Servlet states are wrong - wrong cache results stored
- one thread overwriting whatever was written by previous thread
- lastFactors set to factors of X, NOT factors of Y (causing inconsistent states)
- serve client incorrectly by providing the incorrect factors

- lastFactors and lastNumber are instance fields which are shared across multiple threads - they can be modified by threads running concurrently (inconsistent modifications as per examples above)

**Iteration 3 - Adding Synchronization**

Java `synchronized` keyword - only 1 thread at a time can run inside a `synchronized` code

body




All other threads running inside the process are competing to acquire that lock
Once a thread gets the lock, it will run the code that is locked and will release the lock once its done - so other threads can acquire it to run that code block

THe lock used is linked the object that is running in the method invoked

Before running that method, the thread must get the lock first, so other threads cant run that same method at the same time while the lock has been in use.

Problem: it is slow - it is factorising one thread at atime

```
...
public synchronized void service(ServletRequest req,
                                 ServletResponse resp) {
  BigInteger i = extract(req);
  BigInteger[] factors = null;
  if(i.equals(lastNumber)){
    factors = lastFactors.clone();
  }
  if(factors == null){
    factors = factorize(i);
    lastNumber = i;
    lastFactors = factors;
  }
  encodeInResponse(resp, factors);
}
}
```

Only one thread at the time performs the expensive operation.

Outcome: despite we used multi-threading *and* caching, performance is very close to single-threaded application!

25

It is acting like a single threaded application

how can we resolve?

```
public void service(ServletRequest req,
                    ServletResponse resp) {
  BigInteger i = extract(req);
  BigInteger[] factors = null;
  synchronized(this){
    if(i.equals(lastNumber))
      factors = lastFactors.clone();
  }
  if(factors == null){
    factors = factorize(i);
    synchronized(this) {
      lastNumber = i; lastFactors = factors;
    }
  }
  encodeInResponse(resp, factors);
}
```

Only reads and updates to lastNumber and lastFactors are serialized.

Multiple threads can factorize integers in parallel.

26

We only synchronise sections that is needed (don't synchronise critical sections that are too large or take lots of computational time)

Only protect selected lines by locks so that mutiple threads cannot run that lines at the same time (they are blocked by a lock acquired by a thread)

In this example `this` is used as the lock (the argument to synchronised), where `this` refers to the current object instance that the method is being invoked on - `this` is implicitly used as the lock in a synchronised method. When you specify `this`, and therea re two synchroised(this) lbocks, multiple threads cannot run both blocks at the same time, i.e. they are protected by the same lock ( `this` )

e.g. in the example, you cannot have 1 thread executing `i.equals lastNumber` and another different thread executing `lastNumber = i; lastFactors = factors;` at the same time as these blocks are protected by the **same lock**. (Important to note)

code outside the synchronised block can be executed by multiple threads at the same time (if the method is declared `synchronised` then only one thread can run the entire method at a time)

- in the example, this will allow multiple threads to perform factorising operations at the same time, which takes up a lot of computational time

**Iteration 5 -** possible refactoring**

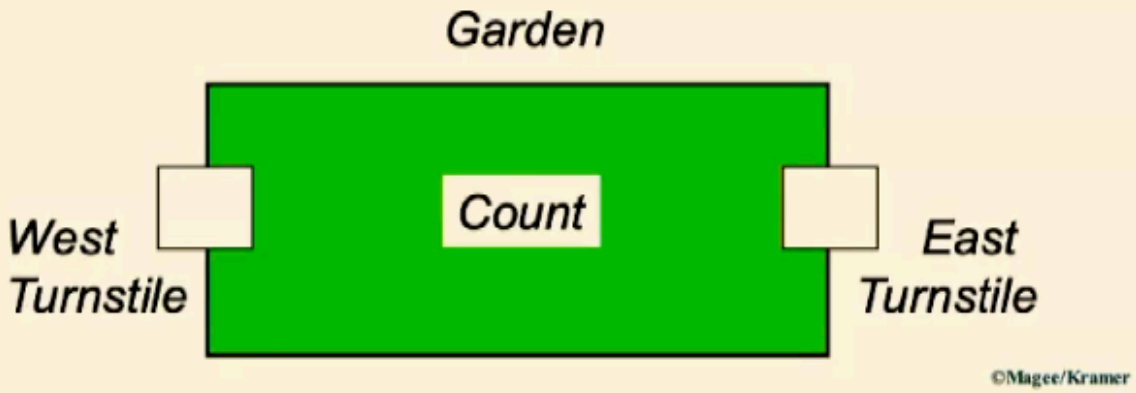
- We can finally isolate critical sections of the code in synchronized methods

```
...
public void service(ServletRequest req,
                    ServletResponse resp) {
  BigInteger i = extract(req);
  BigInteger[] factors = getSavedFactors(i);
  if(factors == null){
    factors = factorize(i);
    saveState(i, factors);
  }
  encodeInResponse(resp, factors);
}
...
```

# Big Garden Example

Big Garden application:

- Inspired by Ornamental Garden [Magee & Kramer ch.4]
  - Garden open to the public, with people entering through either one of two turnstiles
  - Visitors counted at each turnstile and for the entire garden




- Big Garden is a generalization for N turnstiles
  - … with simplified and updated code

Code implementation:

```
public class BigGardenMain {
  private static int numTurnstiles = 2;
  private static int peoplePerTurnstile = 10;
  private static boolean fixit = false;

  public static void main(String[] args) throws InterruptedException {
    for (String arg: args) {
      if (arg.startsWith("--num-turnstiles=")) {
        numTurnstiles = Integer.parseInt(arg.split("=")[1]);
      }
      else if (arg.startsWith("--num-people=")) {
        peoplePerTurnstile = Integer.parseInt(arg.split("=")[1]);
      }
      else {
        System.out.println("Unknown option " + arg);
        System.out.println("Aborting...");
        System.exit(1);
      }
    }
```

```
    BigGarden garden = new BigGarden();

    Turnstile[] allTurnstiles = new Turnstile[numTurnstiles];
    for (int i = 0; i < numTurnstiles; i++) {
      allTurnstiles[i] = new Turnstile("Turnstile" + i, garden, peoplePerTurnstile);
    }
    System.out.println();
    for (int i = 0; i < numTurnstiles; i++) {
      allTurnstiles[i].start();
    }

    // This just prints the garden counter when the other threads have terminated
    for (int i = 0; i < numTurnstiles; i++) {
      allTurnstiles[i].join();
    }
    System.out.println("\n[Garden] The total counter is set to " + garden.getCount() + "\n");
  }
}
```

Creates number of threads where each threads counts garden and starts each threads and waits for all threads to finsih

```
public class Turnstile extends Thread {
  private String id;
  private BigGarden garden;
  private int numAdmitted;

  public Turnstile(String id, BigGarden turnstileGarden, int totalToAdmit) {
    this.id = id;
    this.garden = turnstileGarden;
    this.numAdmitted = totalToAdmit;
  }

  public void run() {
    for (int i = 0; i < this.numAdmitted; i++) {
      this.garden.count();
    }
    System.out.println("[" + this.id + "] I've admitted " + this.numAdmitted + " people");
  }
}
```

Turnstile is a Thread that contains 3 local variables

```
public class BigGarden {
  protected int people = 0;

  public void count() {
    people++;
  }

  public int getCount() {
    return people;
  }
}
```

BigGarden class has a method to counts the number of people

```
Pro:BigGarden ste$ java BigGardenMain --num-people=1000

[Turnstile0] I've admitted 1000 people
[Turnstile1] I've admitted 1000 people

[Garden] The total counter is set to 1913

Pro:BigGarden ste$ java BigGardenMain --num-people=1000

[Turnstile1] I've admitted 1000 people
[Turnstile0] I've admitted 1000 people

[Garden] The total counter is set to 1690

Pro:BigGarden ste$ java BigGardenMain --num-people=1000

[Turnstile0] I've admitted 1000 people
[Turnstile1] I've admitted 1000 people

[Garden] The total counter is set to 1700
```

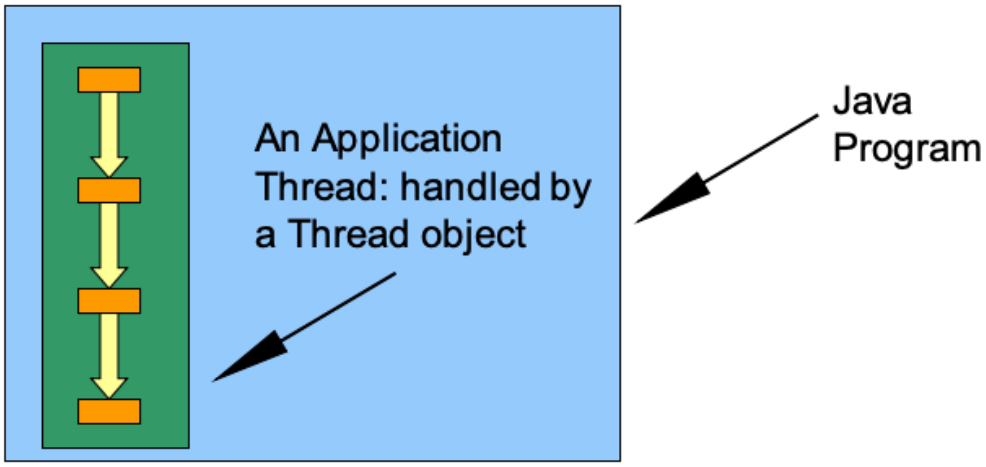
Is Big Garden thread-safe?

- NO - the threads access
- There are interferences when you increase arugmetn of number of people
- `people` is shared among other threads - there is no synchronisation/locking to serialise access to that state, and that incrementing number of people is not synchronised

# Additional Material - Java Threads

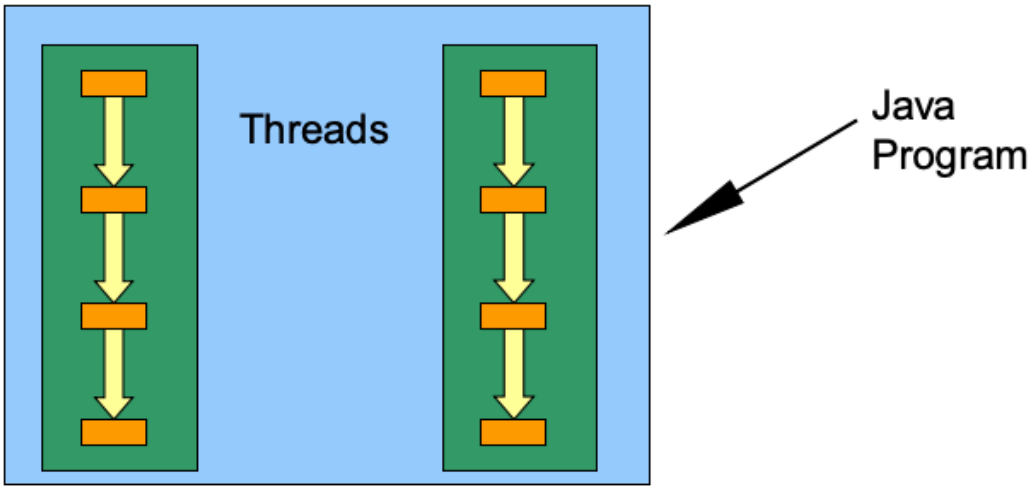
In Java, a thread is represented by a thread object

- provides a single sequential flow of control within a program
  All Java programs have a Main Thread object which is created by default



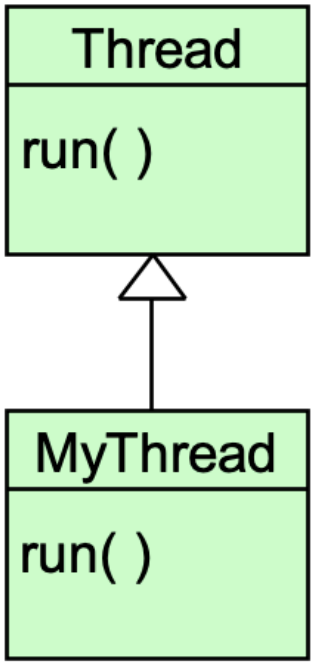

Any Java program can create more Thread objects in addition to the main thread, leading to

multi-threaded applications




**How to create a custom thread in Java?**

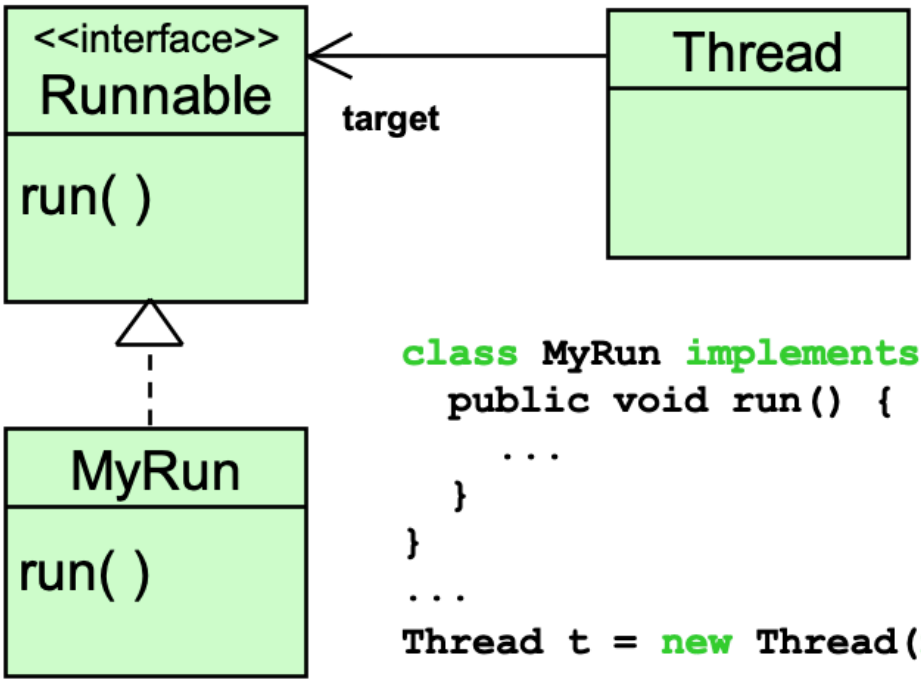
1. **Inherit from Thread class**




```
class MyThread extends Thread {
  public void run() {
     ...
  }
}
...
MyThread t = new MyThread();
```

*caveat*: Java doesn't allow multiple inheritance, so MyThread cannot inherit from other classes

2. **Implement Runnable interface**




```
class MyRun implements Runnable {
  public void run() {
     ...
  }
}
...
Thread t = new Thread(new MyRun());
```

**Runnable vs Thread**

- Runnable interface is simple and generic

```
public interface Runnable {
       public void run();
}
```

- Thread class implements Runnable and offers many additional methods

```
public class Thread implements Runnable {
   public void    start();
   public void    run();
   public boolean isAlive();
   public Thread.State getState();
   public static void sleep(long millis);
   public void join() throws InterruptedException;
    ...
}
```

**Thread states - from getState() method**

### NEW
A thread that has not yet started is in this state.

### RUNNABLE
A thread executing in the Java virtual machine is in this state.

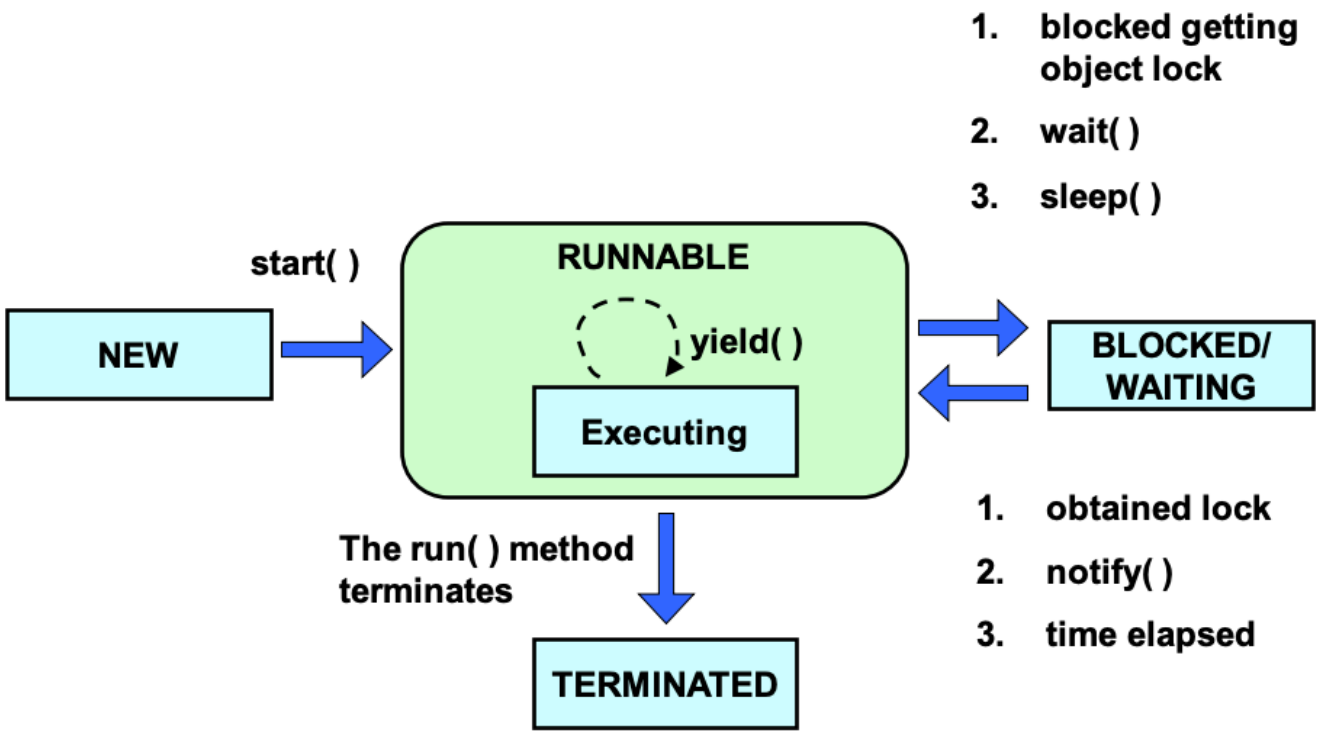
### BLOCKED
A thread that is blocked waiting for a monitor lock is in this state.

### WAITING
A thread that is waiting indefinitely for another thread to perform a particular action is in this state.

### TIMED_WAITING
A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.

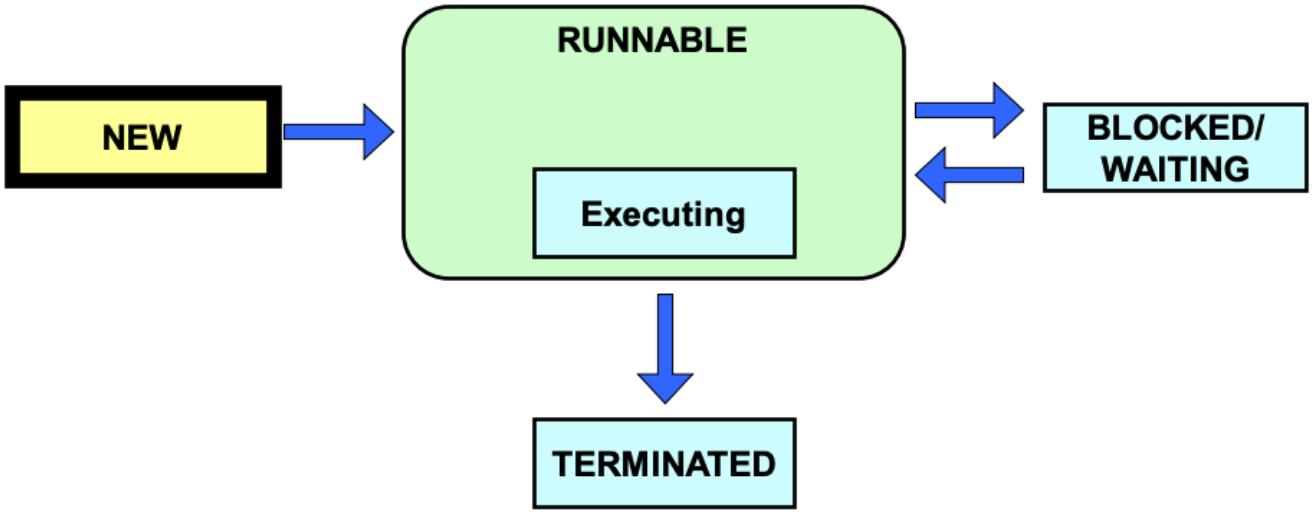
### TERMINATED
A thread that has exited is in this state.

**Thread lifecycle**



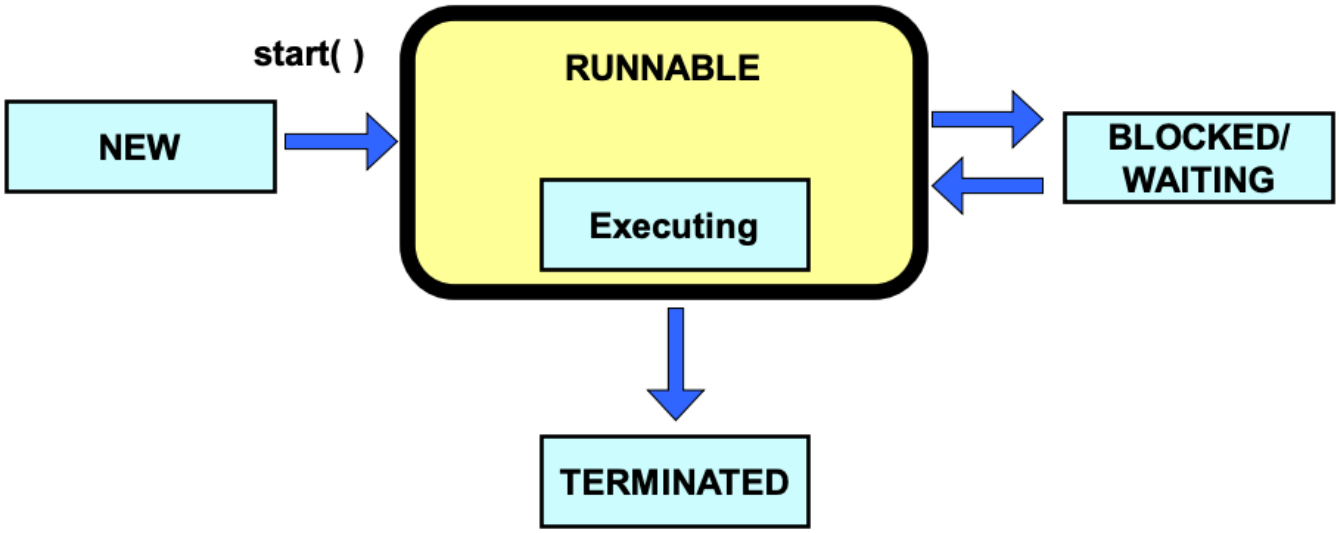

NEW thread - newly created object that is empty with no allocated resources

```
MyThread t = new MyThread();
```



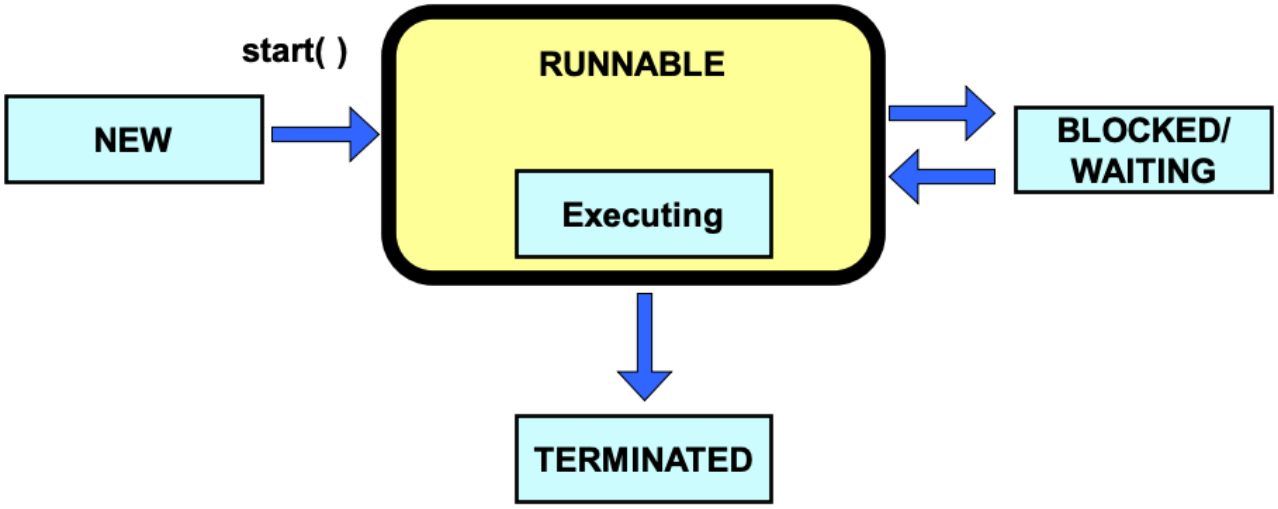

a started thread is in RUNNABLE state when reosurces are allocated to it and run() is executed

```
t.start();
```



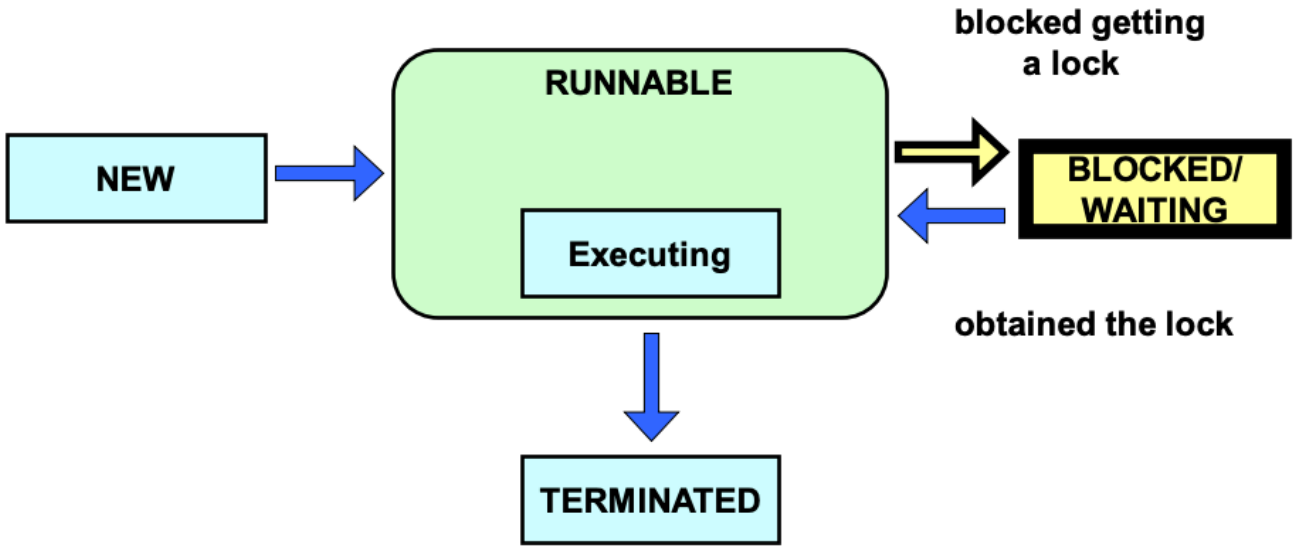

A RUNNABLE thread may not be executing on CPU but waiting for other resources

- e.g. CPU cores or I/O operations

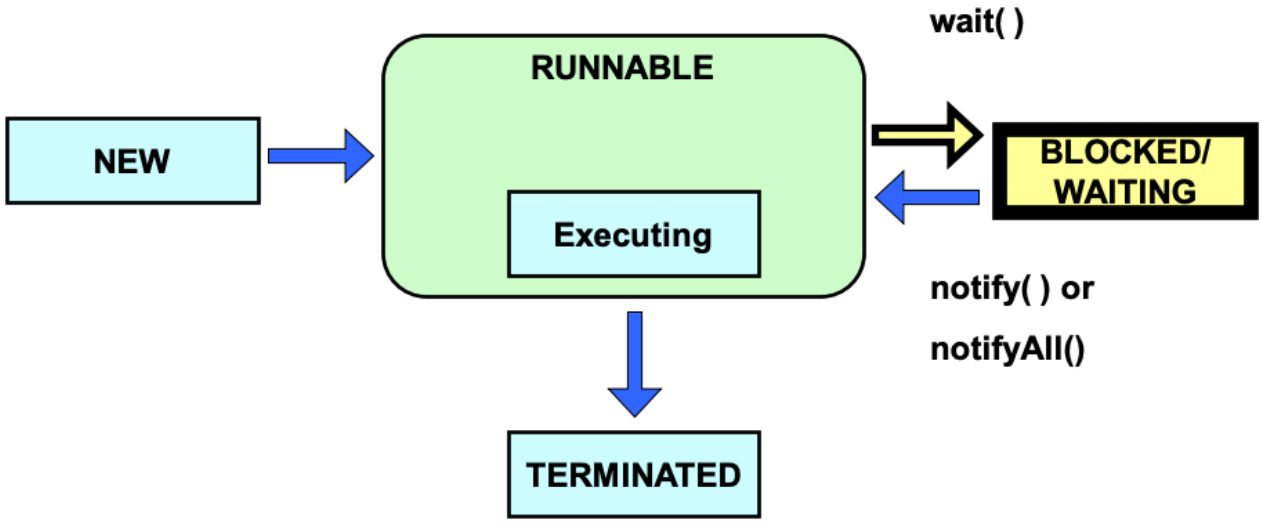



A thread can be programtically BLOCKED

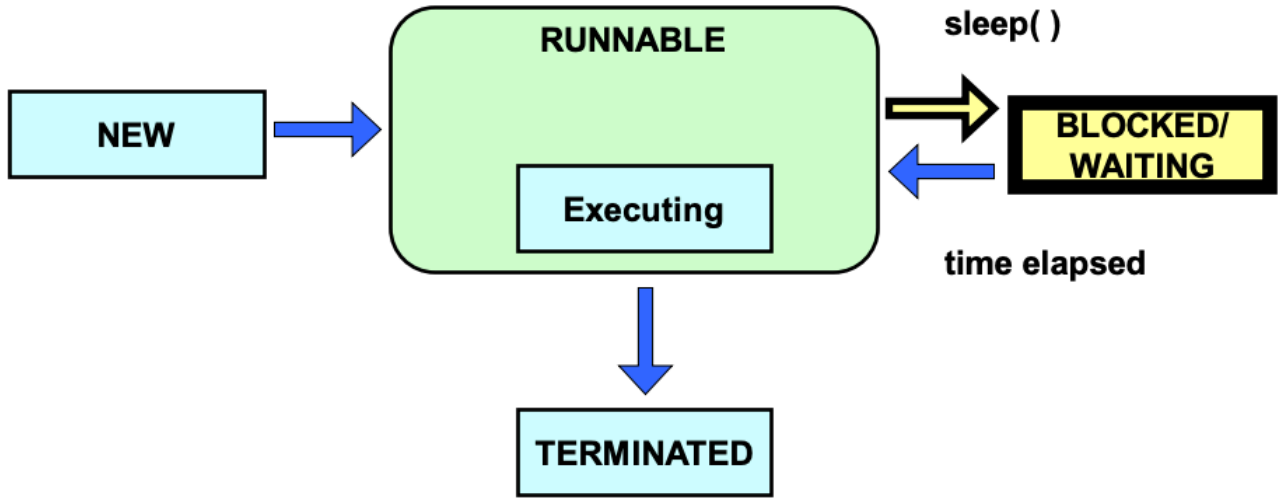
- case 1: waiting for a lock




- case 2: waiting for a notification from other threads

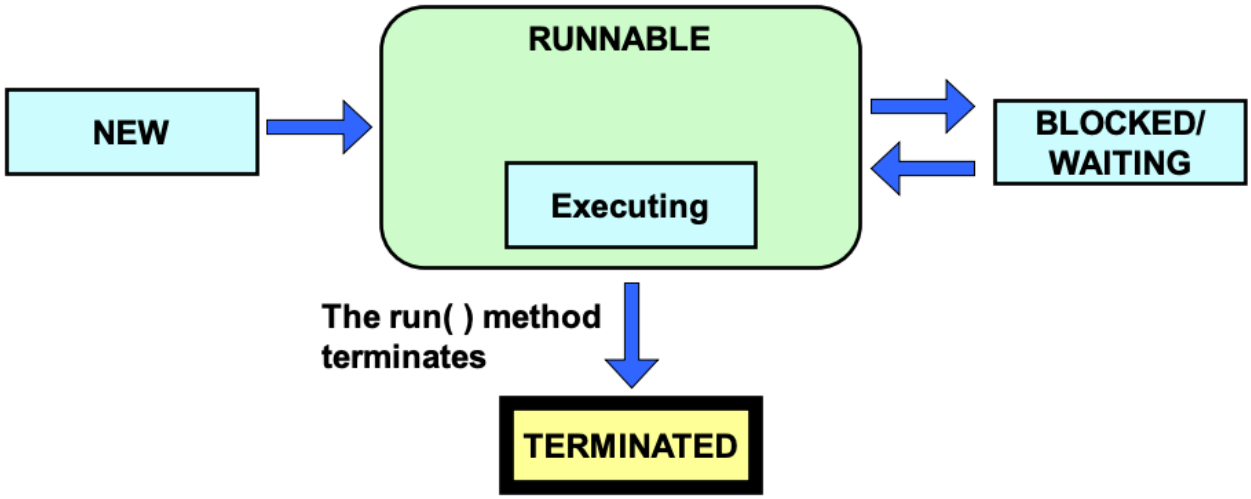


- case 3: instructed by programmer to sleep for X ms




A thread is TERMINATED when run() returns

- thread's resources are released




# Week 9 - Java memory model, monitors and conditional synchronization

(Concurrency in Java continued)

Example:

```
public class Example {
  private static boolean ready;
  private static int number;
  private static class Reader extends Thread {
    public void run() {
      while(!ready)
        Thread.sleep(1000);
      System.out.println(number);
    }
  }
  public static void main(String[] args){
    new Reader().start();
    number = 42;
    ready = true;
  }
}
```

A Reader thread spins until ready is true: when this happens, it simply prints the value of the number field.

Main thread creates a Reader thread, sets number to 42, and then allows Reader to print (and exit). **Expectation**: 42 is printed.

4

is this thread safe?

```
public class Example {
  private static boolean ready;
  private static int number;
  private static class Reader extends Thread {
    public void run() {
      while(!ready)
        Thread.sleep(1000);
      System.out.println(number);
    }
  }
  public static void main(String[] args){
    new Reader().start();
    number = 42;
    ready = true;
  }
}
```

shared among Main and Reader threads

The Reader thread *only reads* the shared variables: the two threads do **not interfere**, nor induce spurious state changes

5

Main is writing, Reader thread is reading.

Zero may be printed, as there is no guarantee that consecutive assignments ( `number = 42;` `ready = true` ) are executed in order, nor that all caches are coherent at any moment in time.

This is because instructions within one thread can be reordered by compiler (is like what happened in ILP/MIPS where instructions are reordered).

Thus if ready = true runs first due to reordering, and the thread ends while loop and prints the number, there is a chance that it will print 0 as number hasn't been updated yet (as it is default to 0) or that the print statement is executed before number gets updated as 42.

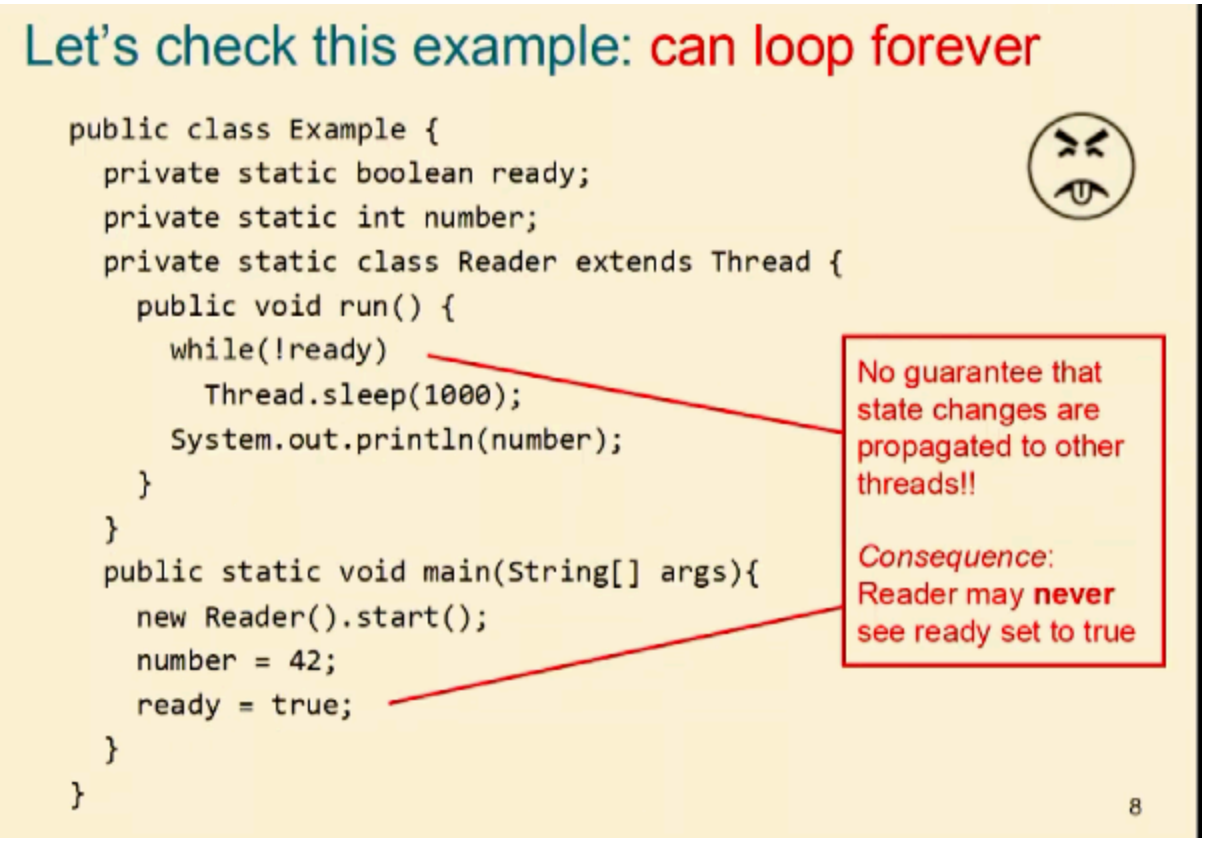
The actions of one thread can be seen by other threads as reordered

- when reordering doesn't affect per-therad correctness

This case doesnt follow the concurrency abstraction model - it abstracts from reality (i.e. the hardware used in concurrency)

- still useful for interference problems e.g. if we are sure that there are no other problems
- if you have other problems (e.g. visibility problems), you should fix these problems first if you want to use concurrency abstraction - as there are limitataions to what we can model with concurrency abstraction if there are problems
- it may be extended to accoutn for reordering by modelling independent instructions as different threads
- it is still useful to reason about all possbile outcomes of multi-threaedd system

another error from example - can cause an infinite loop




reader thread may not be able to see that ready is set to true - problem is beyond control of java programmers (depends on how JVM execute the code) - visibility problem (ready is a state that is modified in the main function)

# Visibility problems

Problem: when one thread changes the state, but other thread doesnt see the entire new state

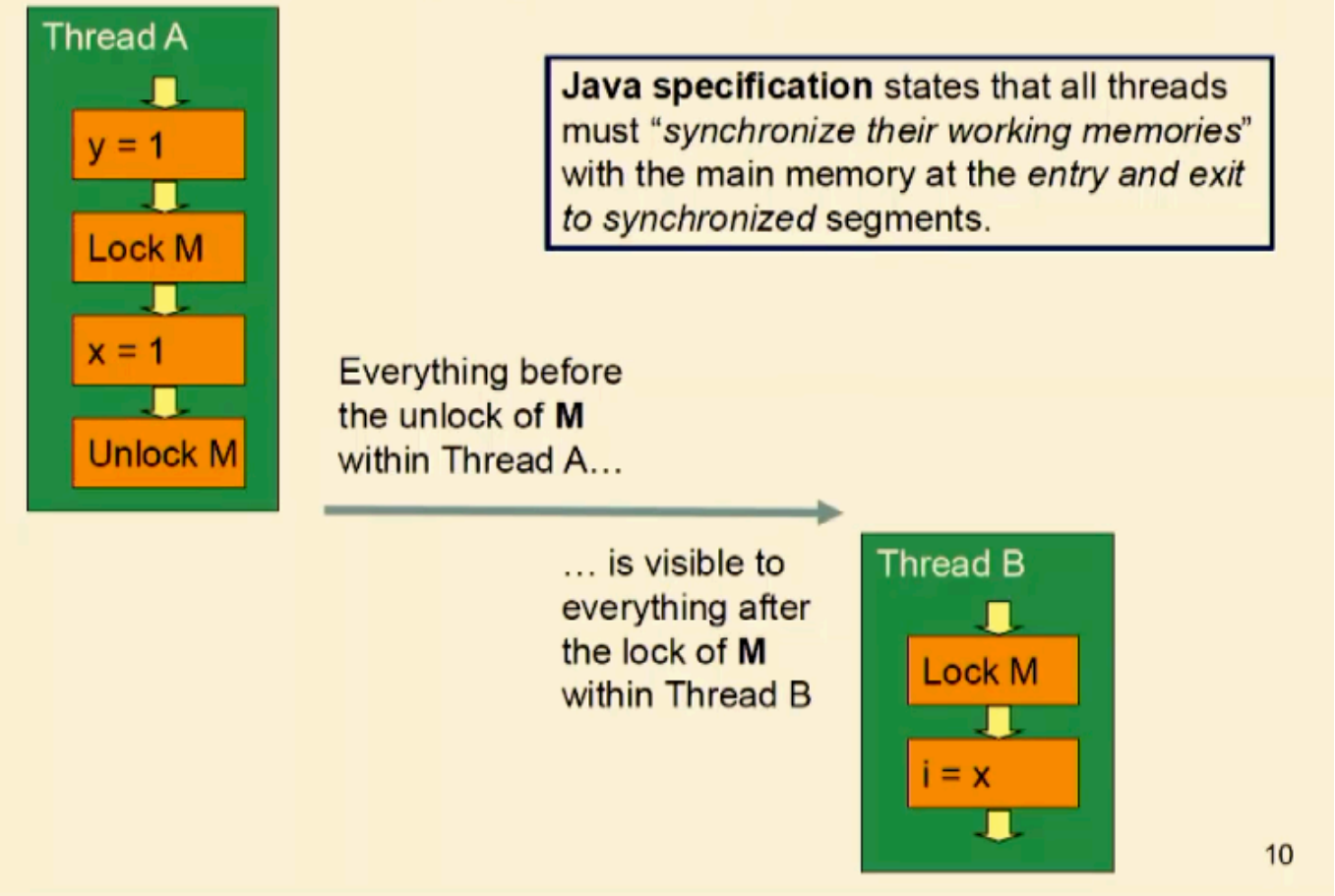
- reader sees a partial change when it prints zero
- or reader sees no change when it doesn't terminate

Stems from having too little interaction between threads (opposite of interference)

SOlution: synchronization whenver one thread accesses data shared with other threads

- i.e. both read/writes must b esynchronzied

## Why synchronization:

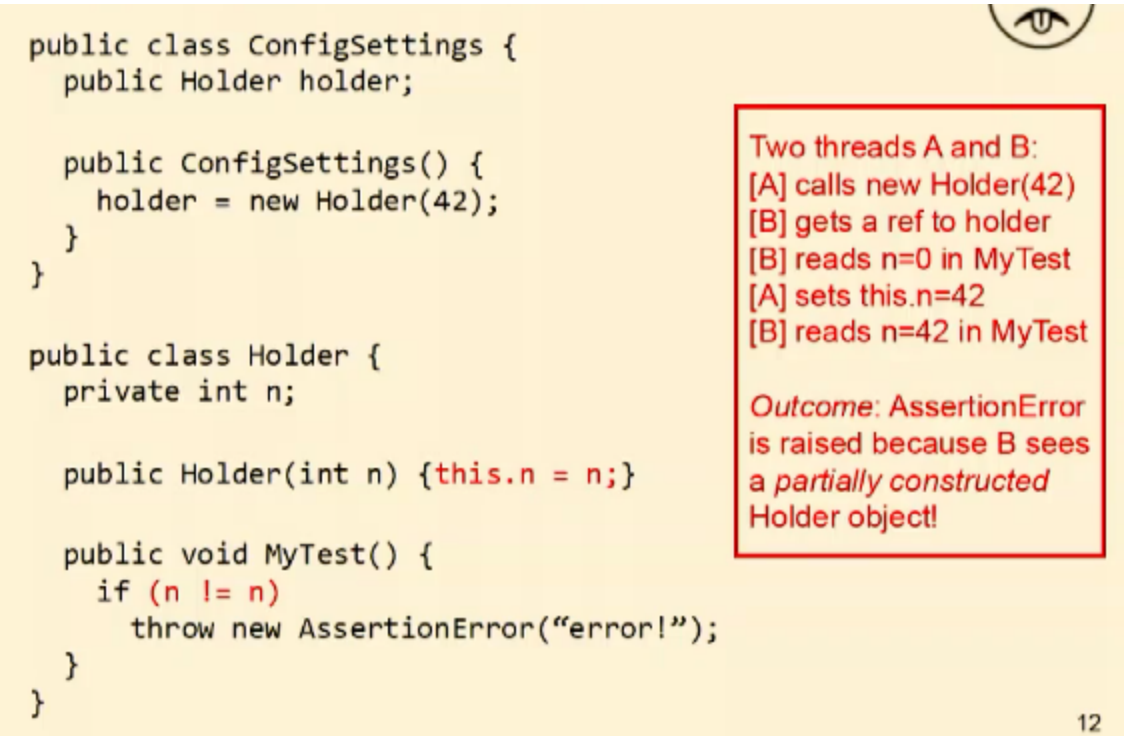



Synchronization enables all assignments/changes done by previous thread to be updated/synchronized

in the diagram, it is guaranteed that B sees everything done by A prior to A unlocking same lock thread B acquired (i.e. `x = 1, y = 1`) - Java is enforcing these constraints

Synchronization will resolve any visibility issues in thread B for all changes done by thread A

## Example

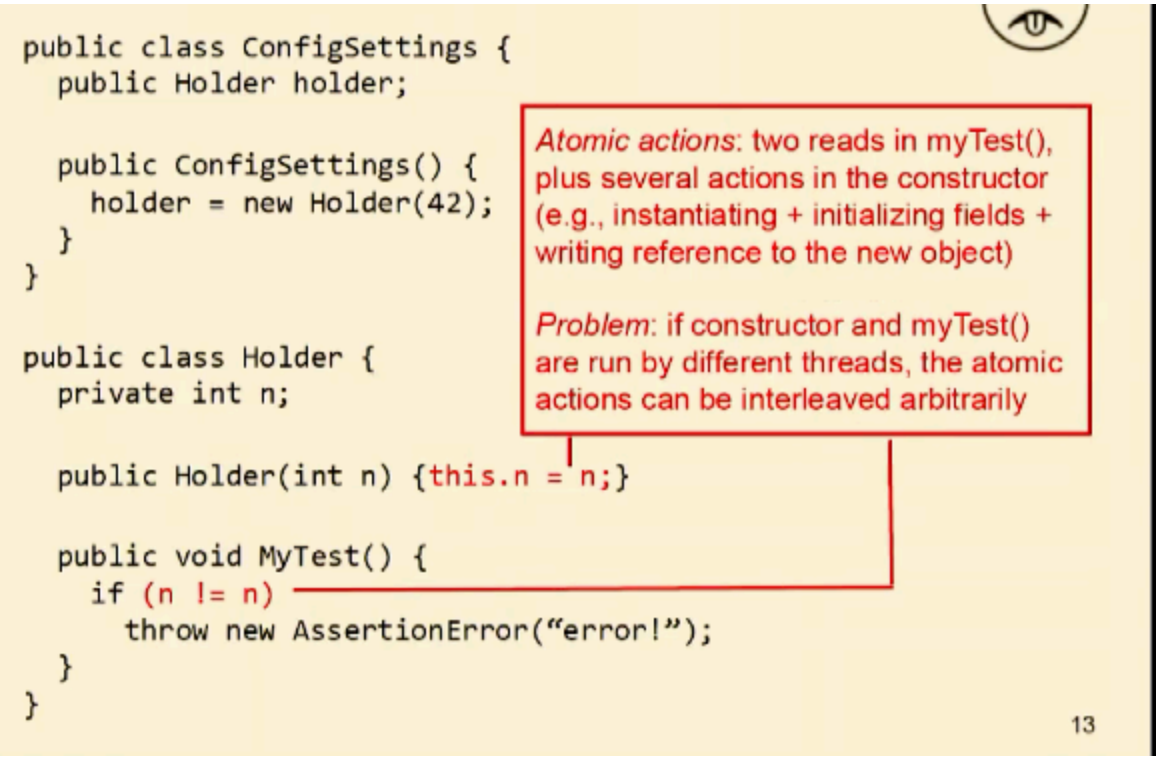



Holder class is not synchronised and there are visibility problems due to one thread having a stale reference of Holder which is a partially constructed object (fields not finished initailsing so n could have been set to its default 0 before assignment by constructor by Thread A). holder is also **published**

MyTest: tests if value put in private instance field is different from itself
This test won't run in a single-threaded program (no race conditions)

what visibility problems that can emerge:

- thread A calls the constructor, B gets a reference to holder as it is public (before `holder` finishes constructing - can happen due to interleavings/no synchronisation)
- B reads `n=0` as n hasn't been set to 42 yet via constructor
- A sets `n=42` via constructor
- B then reads `n = 42` in MyTest as `n` is accessed twice to be compared
- main problem is that how thread B could get the reference to the object if the constructor hasn't finished (partially constructed object)




constructor is not atomic by itself and that some actions are run in constructor

the constructor has to

- load the class
- instantiate the object
- initialise the fields
- write a reference to the new object

there is no constraints between these actions, and that there is a chance these operations can be reordered - so thread B could have seen the object in an inconsistent state (partially constructed object) where not all fields are properly initialised (which throws the assertionerror)

## How to avoid visibility problems

- For any given shared objects, we need to ensure
  - **no thread sees a partially constructed object**
  - **all threads sees changes to the object state**
- we need to understand the JMM (Java Memory Model) to achieve these goals:
  - JMM spells out guarantees that JVM provides about visiblity of one thread's writes to other threads
  - JMM also says that atomic instructions can be partially reordered - which can affect thread safety (other thread can see a partially constructed object)
  - synchronization prevents any instruction reordering that would violate JMM guarantees
  - JMM guarantees that all operations in a thread starts after thread is created

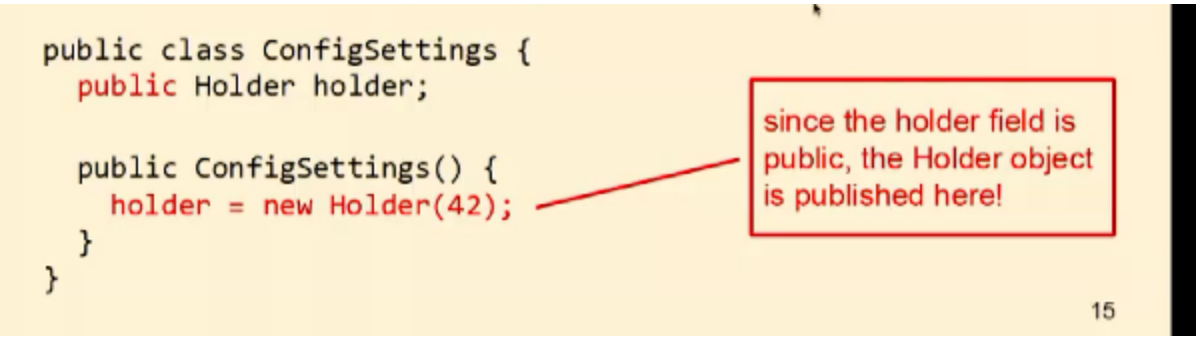
    - within a single thread, all operations are happening one after another - it is not true across multiple threads
- publication / safe publication stems from the JMM

## Publication

An object is **published** when it is made avaialble to code outside its correct context

- e.g.:
  - storing it in a pbulci field
  - returning it from a public method
  - passing it as a parameter

  hazard: publishing an object in a constructor can make a partially constructed object avaialble to external code (e.g. other threads) - anyone can access the `holder` object as a reference to the constructing object is published to




  Reference to the `holder` object can be made public/available to other threads via the constructor as holder field is public

### Safe Publication

An object is **safely published** if both the reference to the object and its state are made visible to other threads at the same time

Achieving goals to avoid visibility problems depend on the object's type (it can be application specific)

- For any given shared object, we have *two* high-level goals, really:
  - G1: no thread sees a partially constructed object
  - G2: all threads see changes to the object's state

how to avoid visibility problems for these object types:

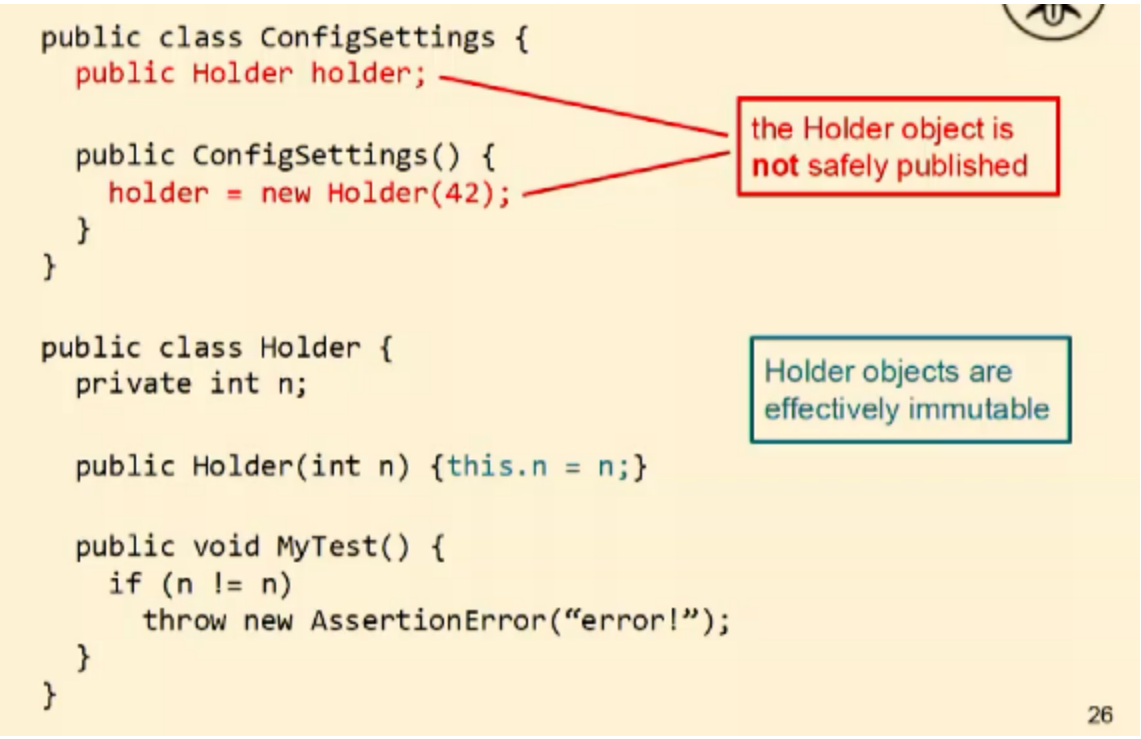
| Object type | Goal G1 | Goal G2 |
|---|---|---|
| Local to thread | ✓ | ✓ |
| Immutable | ✓ | ✓ |
| Effectively immutable | must be safely published | ✓ |
| Mutable | must be safely published | must be thread-safe or guarded by lock |

- If an object is only stored as a local variable, the object remains local to the thread - both goals are fulfilled
- If an object is not local to thread but is immutable, the object's state never changes and the object is properly constructed (reference to the object is not made available to other threads within the constructor) with all fields declared `final` - both goals are fulfilled automatically
- if an **object only has one state but doesnt change after publication** but the object is not immutable, then the object is **effectively immutable** (some fields are not final) - goal 2 is satisfied as it has only one state, but safe publishing is needed to fulfill goal 1
- if a shared object has mutable states (that changes over time) and that multiple threads can read/write these states, to achieve goal 1, we need to safely publish the object, and goal 2 requires the object to be thread-safe or guarded by lock (synchronisation)

**How to safely publish an object**
Safe publication idioms

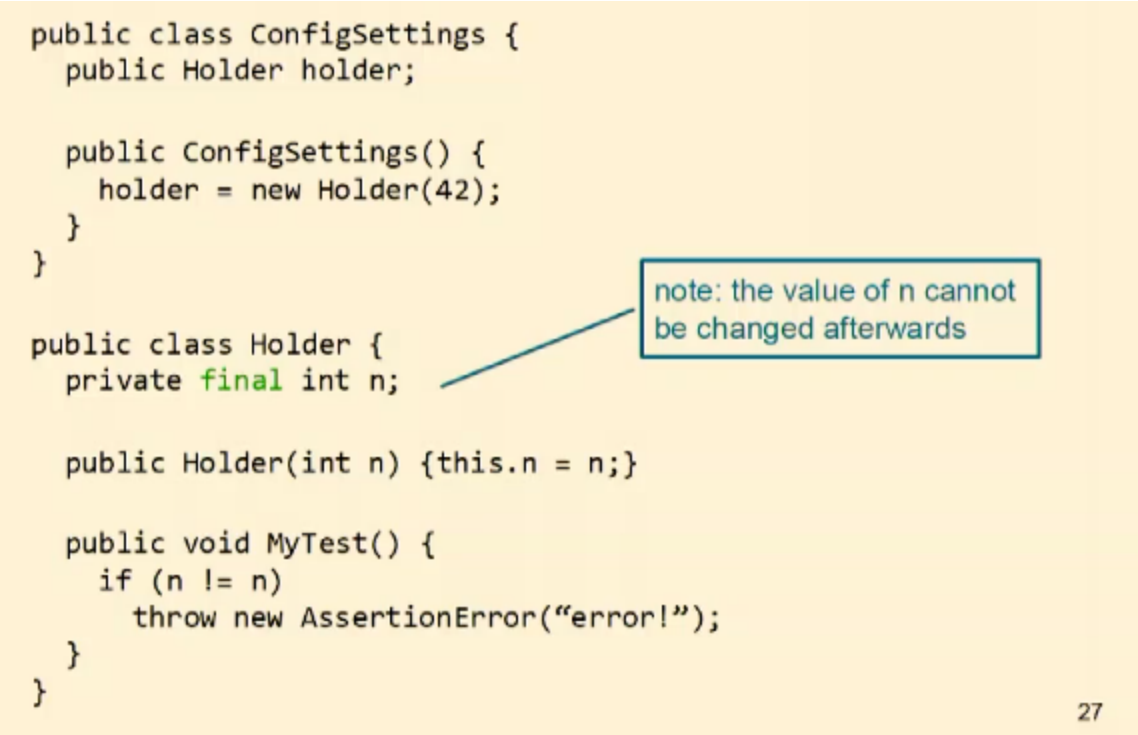
- intiialise object reference from static initilaizer
- store reference to it in a `volatile` field or atomic reference
- store reference to it into a `final` field of a properly constructed object
- store a reference to it into a field properly guarded by a **lock**

from previous example:




object is effectively immutable (one state that doesn't change after publication), but the object is not **safely published** because it is made public

**option 1: make Holder immutable**




making it immutable means we can publish Holder anywhere

**option 2: safely publish holder**

- use a static initailiser

```
public class ConfigSettings {
  public static Holder holder = new Holder(42);
}

public class Holder {
  private int n;

  public Holder(int n) {this.n = n;}

  public void MyTest() {
    if (n != n)
      throw new AssertionError("error!");
  }
}
```

note: holder becomes a static property of ConfigSettings

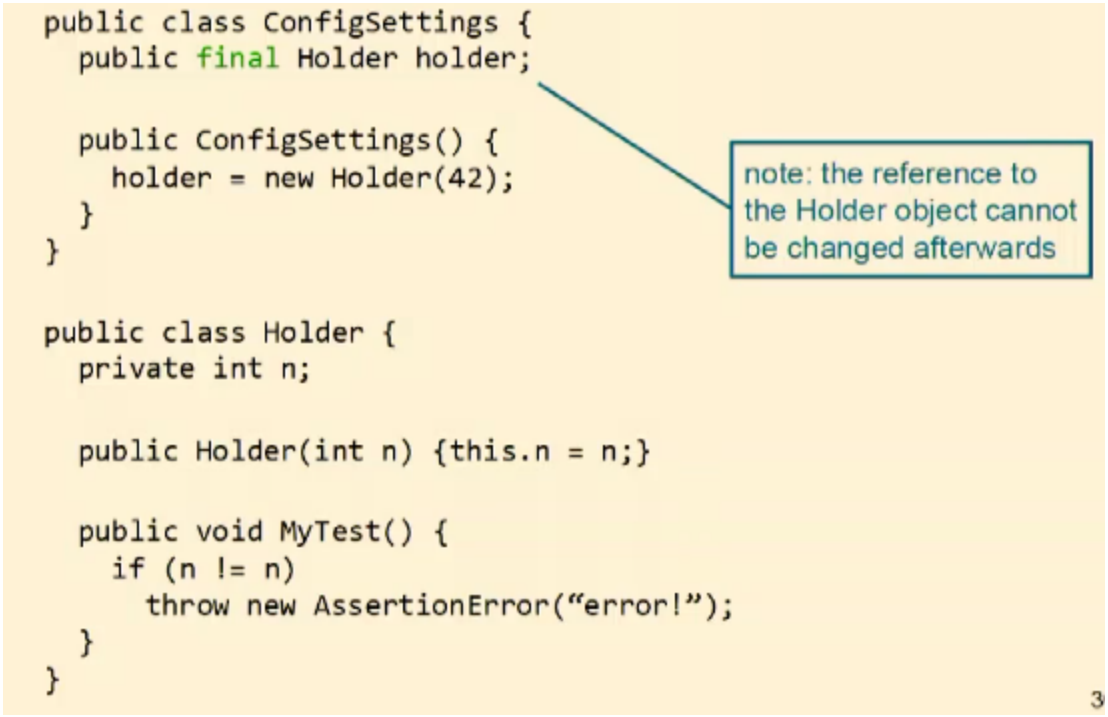
- store in volatile (forcing code to read from main memory)

```
public class ConfigSettings {
  public volatile Holder holder;

  public ConfigSettings() {
    holder = new Holder(42);
  }
}

public class Holder {
  private int n;

  public Holder(int n) {this.n = n;}

  public void MyTest() {
    if (n != n)
      throw new AssertionError("error!");
  }
}
```

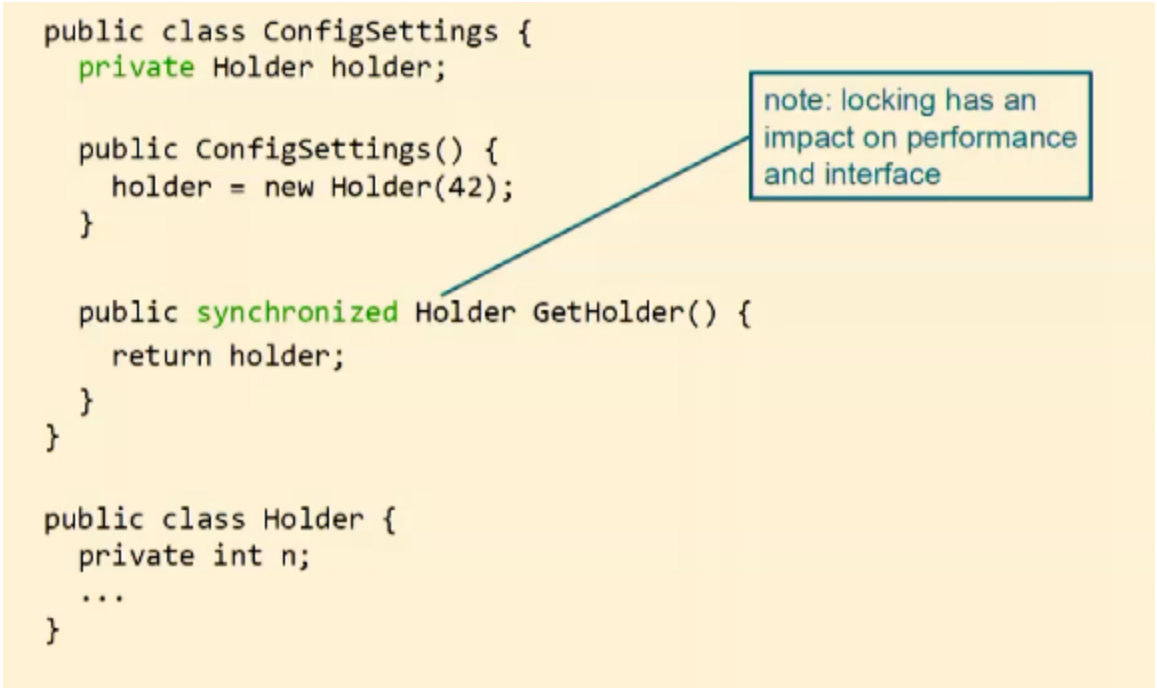
note: volatile ensures that reads always return the most recent write by any thread (weak synchronization)

29

- declare holder as `final`




- guard with a lock (encapsulate the `holder` state in `ConfigSettings`) - can impact performance




volatile is a **very weak form of synchronisation that doesn't guarantee atomicity** (doesn't guarantee that there is no interference, where that multiple threads can't interfere if they access for read/write of the same state variable at the same time)

volatile only **guarantees** that if there is a write to a state variable, a thread will then read the updated variable (**visibility**)

for a shared variable that can have multiple mutable values, it is better to use better synchronisation i.e. locks

you can use volatile for flags (true/false) as there is a lesser chance of interference

if a block of code is not declared `synchronized`, other threds can access them completely asynchronously

**bonus exercise** - where are the fields published

```
public class ConfigSettings {
  private int id;
  private Holder holder;

  public ConfigSettings() {
    SetValue();
    holder = new Holder(42);
  }
  private void SetValue() {
    id = 10;
  }
  public synchronized void Configure (UI interface) {
    interface.SetConfigId(id);
    interface.SetValue(holder.GetValue());
  }
}
```

method in Holder that simply returns the value of private int n

32

answer:

`id`:

- in `Configure()` - as id/holder made private
- id/holder are not published in setValue or construtor
- `id` is published in configure as private field id is published when passed to `interface.SetConfigId(id)`

`holder`:

- `holder` field is NOT published
- In configure, we are not passing the reference to `holder` object but holder's fields - so it is not published

Configure is a public method that can expose the values of private fields where they are passed to external interface methods

# Design Patterns for multithreaded code

Reasoning about interference and visibility problems for each object/state is complex and error prone

- but correctness must be ensured in all concurrent applications
  thus we need design patterns

1. Monitor pattern
   - to make an object thread-safe, we encapsulate all its mutable state and guard the stater with the object's intrinsic lock
     - i.e. all methods that access the stae are synchronised
   - how to use the pattern:
     - wrap non thread-safe objects, so you transform them into a thread-safe object by using a wrapper (there are thread-safe libraries in Java to allow that)
     - model data as classes implementing the moniotor pattern: they can be safely accessed by multiple threads such that the data is accessed in a synchronised way by different threads
   - example:

```
public class MutableHolder {
  private int n = 0;

  public synchronized void SetValue(int x) {
     n = x;
  }

  public synchronized int GetValue() {
    return n;
  }

  public synchronized void MyTest() {
    if (n != n)
      throw new AssertionError("error!");
  }
}
```

2. conditional synchronisation
   - problem: what if a thread cant work on current state of a monitor(e.g. read an empty buffer or write to full buffer)
     - fail/raise an error (only possibility for single-threaded apps)
     - wait for another thread to change the monitor's state
   - conditional sync allows the 2nd possibility to happen (we can write the code to handle that ourselves) - it simplifies the waiting/waking up part of threads

- basically if precondition is not met, suspend the thread and wake the thread up once something changes in the state
- pseudocode:
  - *PSEUDO-CODE*

```
acquire lock
while (predicate) {
  release lock
  wait to be notified
  reacquire lock
}
perform action
notify other threads
release lock
```

  - acquire lock that guards the shared object
  - look at the predicate (base/precondition)
    - release the lock - let another thread change the state
    - wait to be notified
    - reacquire the lock once a state chanegs (mutual exclusion)
  - perform action once the thread wakes up
  - notifty other threads once action is done
  - release the lock so other threads can acquire the lock
- example code:
  - *EXAMPLE*

```
public synchronized V read(){
  while(isEmpty()){
    wait();
  }
  V val = doRead();
  notifyAll();
  return val;
}
```

38

  - each object is a condition queue, not just a lock
  - wait and notifyall are methods that each object in Java implements

## Conditional Sync

- condition queues give threads a way to subscribe to specific updates on a given condition
  - waiting threads form a so-called wait set
- condition queues offer methods to
  - wait() - to allow threads to enter a condition queue

    - notify() - wake up only one thread in the wait set
    - notifyAll() - wakes up all threads waiting in the condition queue
- imporant: notifications signal that something has changed, not what has changed (so need to re-check the condition once the thread has waken up)
- each object behaves as a condition queue
    - `this.wait()` makes a thread wait until `this.notifyAll()` is called
    - these methods are called on the intrinsic condition queue of each other
    - there is a relationsihp between condition predicate, condition queue and intrinsic lock used - hence there are assumptions
- what lock and conditional queue to use?
    - condition predicate - condition on state variables
    - before testing condition predicate, we must hold the lock guarding the corresponding state variables
        - we must also hold that lock when calling wait and notification methods on the condition queue
    - hence, `lock object == condition queue object`
- entry/exit protocols (covered in book)

example:

```
public synchronized void put(E o) throws InterruptedException {
  while (count==size) {
    System.out.println("[Buffer] Adding " + o + " to buffer needs to wait");
    wait();
  }
  buf.set(in, o);
  ++count;
  in=(in+1) % size;
  notifyAll();
}

public synchronized E get() throws InterruptedException {
  while (count==0) {
    System.out.println("[Buffer] Reading from buffer needs to wait");
    wait();
  }
  E o = buf.get(out);
  buf.set(out,null);
  --count;
  out=(out+1) % size;
  notifyAll();
  return (o);
}
```

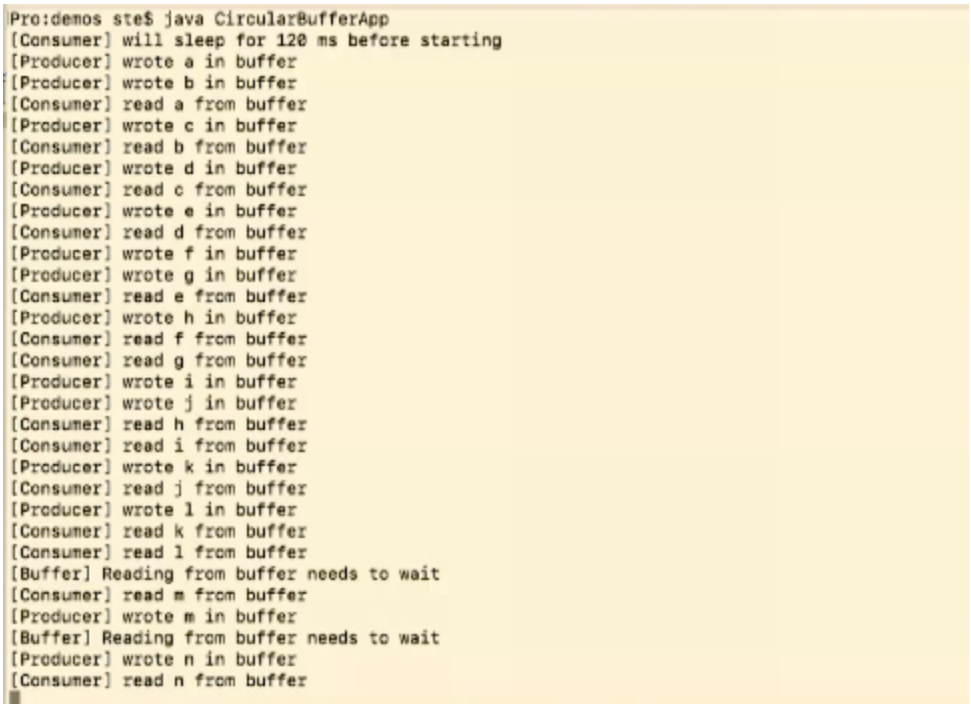
put condition checks where if it doesn't hold, the thread has to wait until it gets notified

possible interleavings:
1 thread being consumer/other thread being producer

consumer - only performs `get`
producer - only performs `put`

consumer thread has to wait as buffer is empty, producer performs put, and then performs `notifyAll()` to wake up the consumer thread and checks the condition again before performing action and notifying all threads in condition queue, and vice versa happens

example execution:



```
Pro:demos ste$ java CircularBufferApp
[Consumer] will sleep for 120 ms before starting
[Producer] wrote a in buffer
[Producer] wrote b in buffer
[Consumer] read a from buffer
[Producer] wrote c in buffer
[Consumer] read b from buffer
[Producer] wrote d in buffer
[Consumer] read c from buffer
[Producer] wrote e in buffer
[Consumer] read d from buffer
[Producer] wrote f in buffer
[Producer] wrote g in buffer
[Consumer] read e from buffer
[Producer] wrote h in buffer
[Consumer] read f from buffer
[Consumer] read g from buffer
[Producer] wrote i in buffer
[Producer] wrote j in buffer
[Consumer] read h from buffer
[Consumer] read i from buffer
[Producer] wrote k in buffer
[Consumer] read j from buffer
[Producer] wrote l in buffer
[Consumer] read k from buffer
[Consumer] read l from buffer
[Buffer] Reading from buffer needs to wait
[Consumer] read m from buffer
[Producer] wrote m in buffer
[Buffer] Reading from buffer needs to wait
[Producer] wrote n in buffer
[Consumer] read n from buffer
```

consumer wakes up after producer write something to buffer, and vice versa. there are instances when buffer is empty so consumer has to wait (timeout is 120ms)

# Week 10 - Reasoning on correctness of multi-threaded systems

(Concurrency on larger applications)
Note:
if multiple threads access the same mutable state variable without appropriate synchronization, **your program is broken** as multi-threaded systems are prone to interference/visibility problems hence the need for synchronization

Does synchroniastion guarantee correctness? **NO**

Correctness - expected, useful behaviour of the program where it entails two properties:

- safety - nothing bad happens
- liveness - something good eventually happens

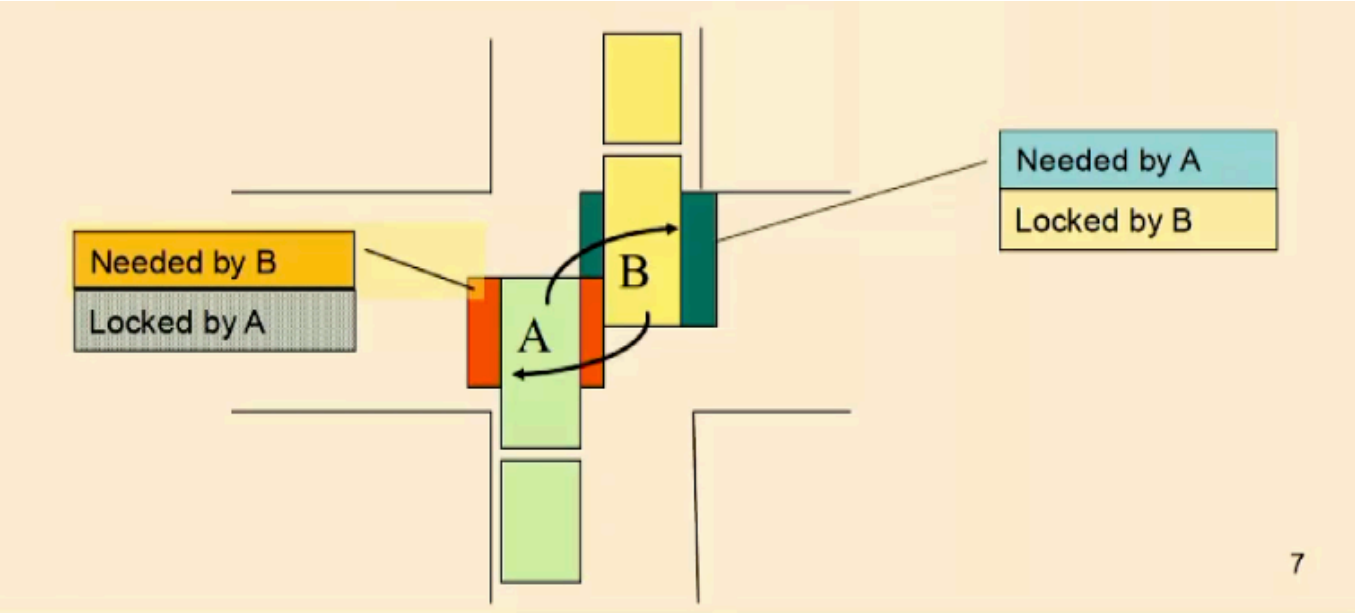
does an appropriate synchronized thread-safe program ensure liveness too?

- NO - safety is generally at odds with liveness as locking can prevent anything good from happening

# Liveness Problems

## Deadlock

- All threads are lbocked
  - typical case: multiple threads wait forever for each other due to **cyclic dependencies in resource acquisiton**
- example irl: traffic gridlock




both threads cannot make progress as they need each others' locks

example: Transfermoney

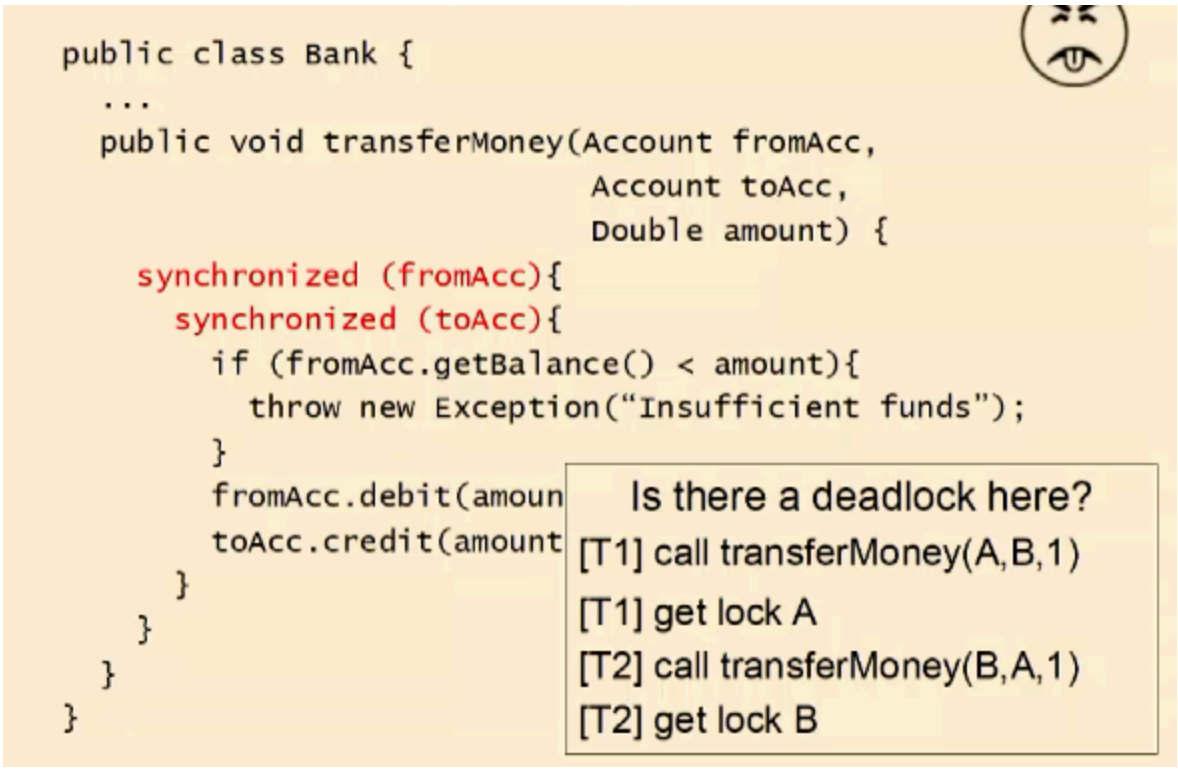
```
public class Bank {
  ...
  public void transferMoney(Account fromAcc,
                            Account toAcc,
                            Double amount) {
    synchronized (fromAcc){
      synchronized (toAcc){
        if (fromAcc.getBalance() < amount){
          throw new Exception("Insufficient funds");
        }
        fromAcc.debit(amount);
        toAcc.credit(amount);
      }
    }
  }
}
```

8

Method simulates transfer between one account to another
you want transfer to be synchronised where you syncrhonise both from account and to account. you use two locks as we wnat to allow othjer accounts to transfer to same account if fromAcc is different

Is there a deadlock?

- Maybe - depends onw how methods is called by threads (depends on which fromAcc and toAcc is called/passed to the methods, and when these methods are called)
- possible interleaving:




- next instruction thread A has to execute is to synchronise `toAcc` which is B, so it is tryign to take the lock acquired by B
- thread B has lock B and is trying to take the lock acquired by A
- this causes a cyclic locking dependencies meaning that both threads are blocked, causing a **deadlock**

Deadlock examples (from textbook)

| Type | Example | Reference |
|---|---|---|
| Single object, conflicting methods | LeftRightDeadlock | [GPBBHL 10.1.1] |
| | Multiple Databases | [GPBBHL 10.1.5] |
| Single object, conflicting external calls | transferMoney | [GPBBHL 10.1.2] |
| | Dining Philosophers | [Moodle] |
| Cross objects | Taxi dispatcher | [GPBBHL 10.1.3] |

dining philosopherrs problem:

- Dining Philosophers is a classic concurrency problem
  - Examined by Dijkstra in 1968
  - Employed as test case by many synchronization algorithms
- Problem formulation
  - 5 philosophers around a table
    - each philosopher is a thread
  - Each philosopher spends time alternating thinking and eating
  - To eat, each philosopher needs access to 2 forks
  - Only 5 forks on the table
    - each fork is on the left of one philosopher and on the right of another philosopher

(not examined)

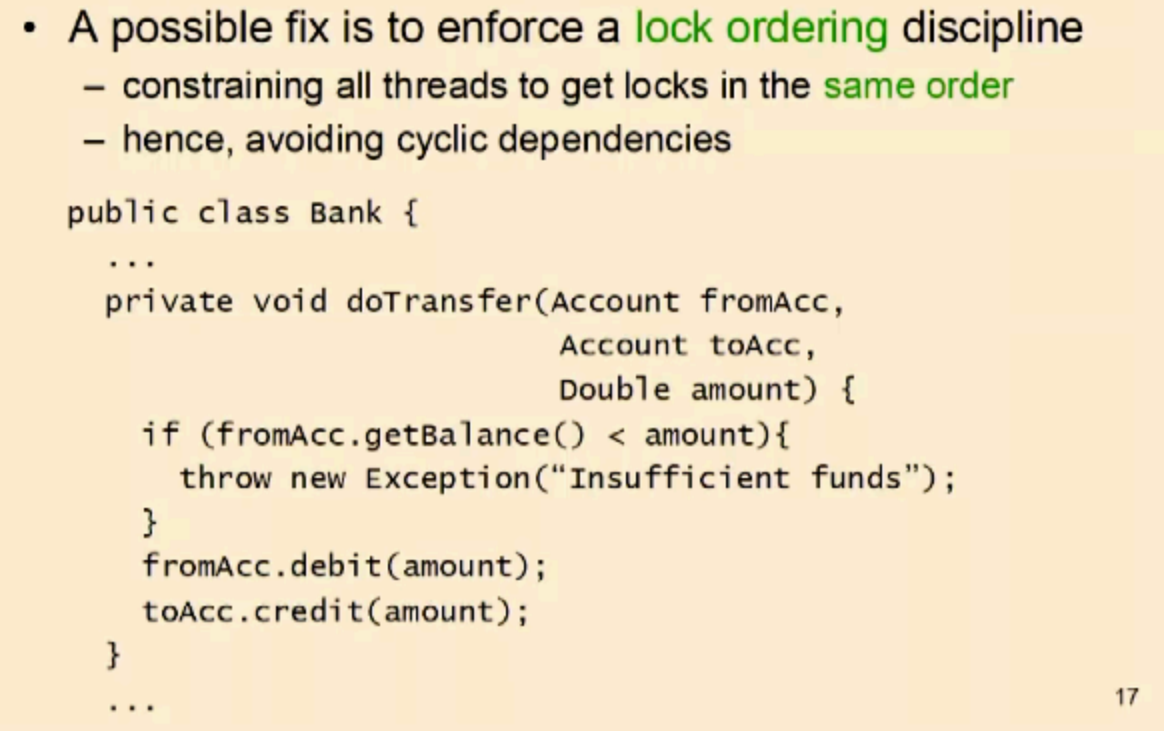
deadlock happens when all philosophers have one fork each and are all trying to acquire another fork

it can be complicated to analyse your application for a deadlock as it depends on the objects/methods

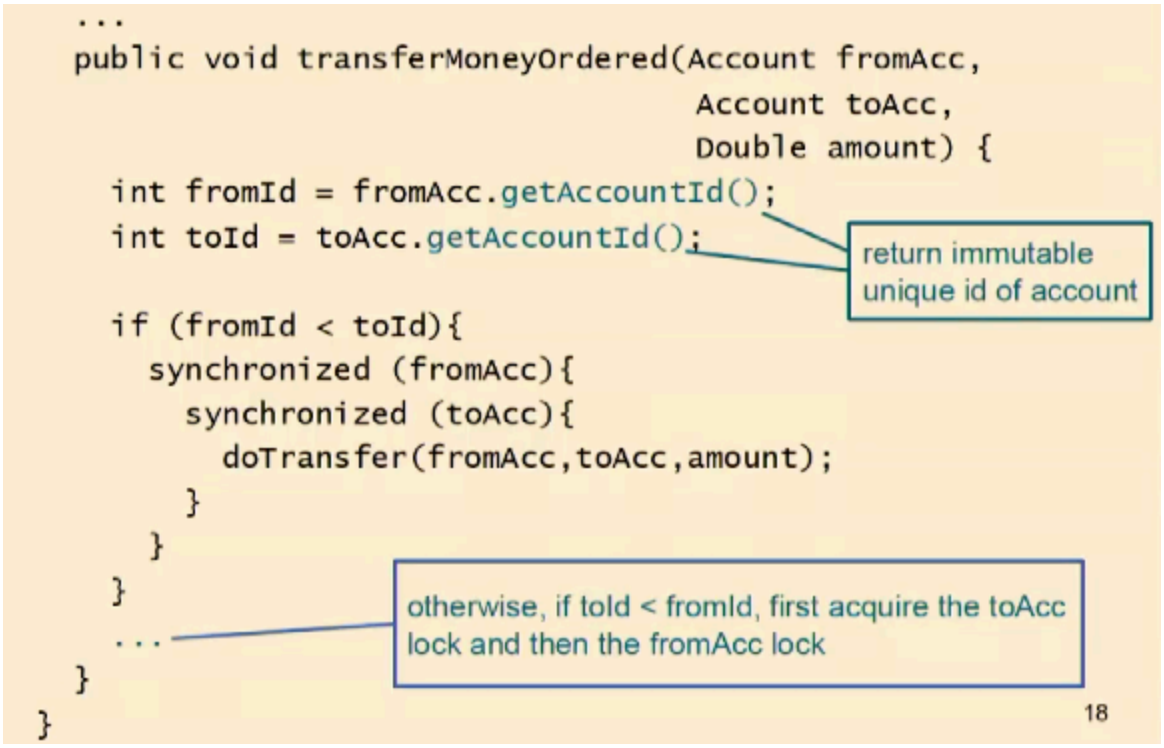
**cross objects** - how can it cause deadlock
you can get deadlock where if one object that has its own intrinsic block nd from one synchronised method in its own class calls a synchronised method in another class, the object is acquiring its own lock and tries to acquire the lock of another ojbect. if you do the same from other object, there is a potential conflict

How to avoid deadlocks in transferMoney?

- enforce a lockl order ing on how threads acquire locks to avoid cyclic dependencies




doTransfer - private function checks balance and then credits if sufficient money




we can order the ID (assumed to be immutable, unique and sortable)

you always synchronise the account with lowest ID first - eliminates deadlock as a reuslt

## Open calls

In other cases deadlocks may be caused by calling an external method while holding a lock

**Open call**: call to an external method (possibly of another object) with no acquired lock / without holding a lock - to avoid deadlocks

- as calling an external method with an acquired lock can risk a deadlock as you don't know what the other method is doing

Using open calls makes it easier to ensure locks are acquired in consistnet order

- they restrict number of code paths acquiring locks => easier to follow (similar role to encapsulation)

drawback - open calls may indue loss of atomicity

## Avoiding cyclic dependencies

How? identify where ressources are acquired, then ensure ordering is **always** consistent (KEEP IT SIMPLE)

| Technique | Example | Reference |
|---|---|---|
| Threads never acquire >1 resource at the time | - | [GPBBHL10.2] |
| Order resource acquisition within individual methods | transferMoneyOrdered | [GPBBHL 10.1.2] |
| Use open calls across objects | ThreadSafe Taxi dispatcher | [GPBBHL 10.1.4] |
| Acquire with timeout | transferMoney w/ tryLock | [GPBBHL 10.2.1] |
| Detect-and-abort | DBMS | [GPBBHL10.1] |

for timeout, you would not know what actually if your thread reaches timeout, but you can avoid being stuck forever

there is no implementation in Java that support detecting deadlocks

# Starvation

- a thread is perpetually denied access to resources it needs to progress

| Resource waited for | Example | Reference |
|---|---|---|
| CPU cycles | low-priority threads starved by high-priority ones | [GPBBHL 10.3.1] |
| Object or Lock | lock held by a thread running an infinite loop | [GPBBHL 10.3.1] |
| | Readers and Writers | [Moodle] |

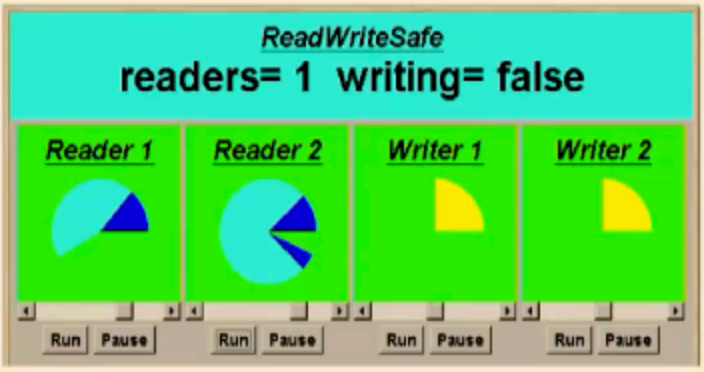
Light version of starvation: unfair allocation of resources (e.g., CPU), possibly causing poor responsiveness

high priority threads can take longer time/ more cpu cycles - they can starve lower priority threads

alternatively if a thread never releases a lock due to infinite loop such that other threads waiting for that lock hve to wait forever

starvation examples: readers/writers

- Inspired by Readers and Writers [Magee & Kramer ch.7]
  - Shared database (DB) accessed by two types of threads: Readers that read the DB, and Writers that update it.
  - Writers must have exclusive access to the DB, while any number of Readers may simultaneously access it.




example implementation:

```
/****************** READERS ************************/

class Reader extends Thread {
  ReadWriteController ctrl;
  String id;

  Reader(ReadWriteController controller, String name) {
    ctrl = controller;
    id = name;
  }

  public void run() {
    try {
      while(true)  {
        System.out.println("[" + id + "] waiting for permission to read");
        ctrl.acquireRead();
        System.out.println("[" + id + "] reading...");
        Thread.sleep(1000);
        ctrl.releaseRead();
      }
    }
    catch (InterruptedException e){}
  }
}
```

```
/****************** WRITERS ************************/

class Writer extends Thread {
  ReadWriteController ctrl;
  String id;

  Writer(ReadWriteController controller, String name) {
    ctrl = controller;
    id = name;
  }

  public void run() {
    try {
      while(true)  {
        System.out.println("[" + id + "] waiting for permission to write");
        ctrl.acquireWrite();
        System.out.println("[" + id + "] writing...");
        Thread.sleep(1000);
        ctrl.releaseWrite();
      }
    }
    catch (InterruptedException e){}
  }
}
```

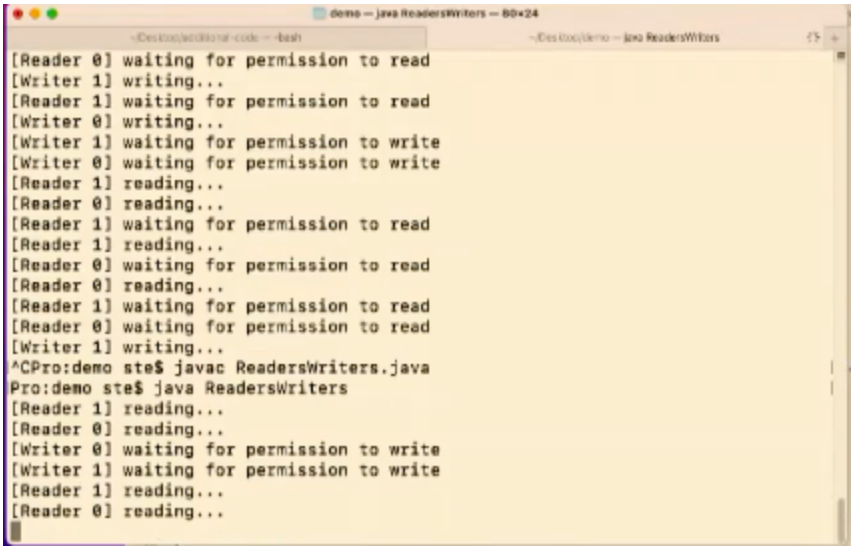
```
interface ReadWriteController {
   public void acquireRead() throws InterruptedException;
   public void releaseRead();
   public void acquireWrite() throws InterruptedException;
   public void releaseWrite();
}
```

```
class ReadWriteControllerSafe implements ReadWriteController {
  private int readers = 0;
  private boolean writing = false;

  public synchronized void acquireRead() throws InterruptedException {
    while (writing) wait();
    ++readers;
  }

  public synchronized void releaseRead() {
    --readers;
    if(readers==0) notifyAll();
  }

  public synchronized void acquireWrite() throws InterruptedException {
    while (readers>0 || writing) wait();
    writing = true;
  }

  public synchronized void releaseWrite() {
    writing = false;
    notifyAll();
  }
}

/****************** MAIN ***********************/

public class ReadersWriters {
  private static int numReaders = 2;
  private static int numWriters = 2;

  public static void main(String[] args) throws InterruptedException {
    ReadWriteController controller = new ReadWriteControllerSafe();

    for (int i=0; i<numReaders; i++) {
      new Reader(controller,"Reader " + i).start();
    }
```

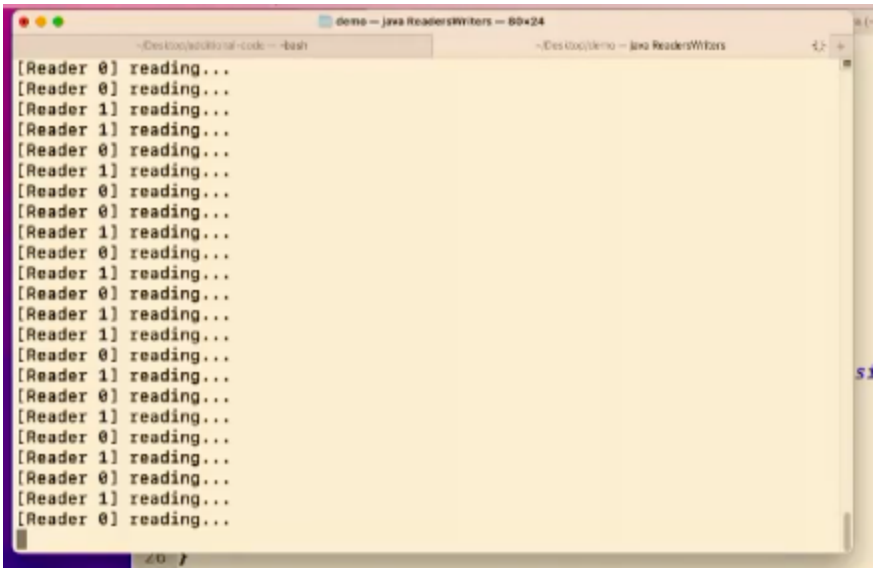
```
/****************** MAIN ***********************/

public class ReadersWriters {
  private static int numReaders = 2;
  private static int numWriters = 2;

  public static void main(String[] args) throws InterruptedException {
    ReadWriteController controller = new ReadWriteControllerSafe();

    for (int i=0; i<numReaders; i++) {
      new Reader(controller,"Reader " + i).start();
    }

    for (int i=0; i<numWriters; i++) {
      new Writer(controller,"Writer " + i).start();
    }
  }
}
```

we create two reader with a Reader thread for each reader, and two writer threads. both will ahve reference to controller

execution:



what happen sif we comment out permission to reaad:



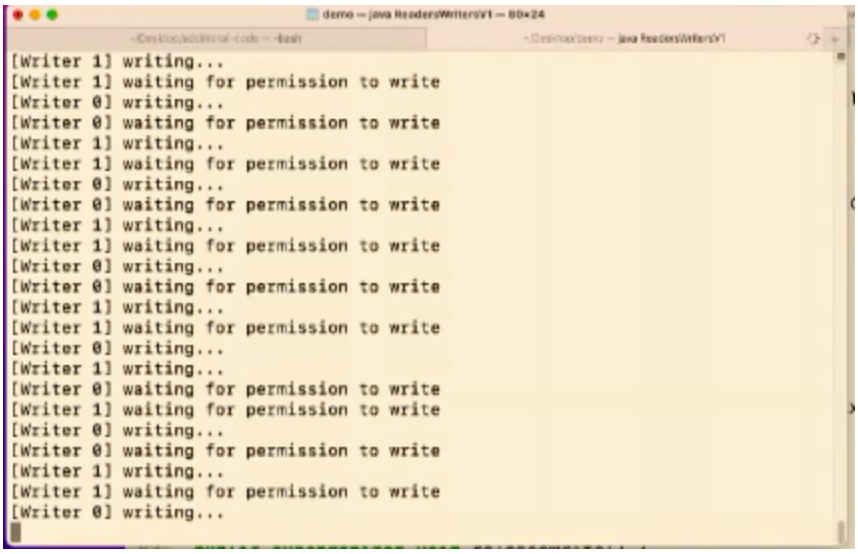

will it cause a starvation problem?

- writers are starved because there is never a case there are zero readers
- there is higher priority for the readers
- possibility for writers to get the running lock is very small
- the readers all alternate reading between each other - there is always a reader reading the database
- notifyAll only invoked if 0 readers, but if there is another reader, the other reader will take the reading and keeps getting another reading lock

**how about you give writer priority** - whenever there is a writer waiting to acquire the lock, reader cannot acquire the lock

you track number of writers that are waiting to write on database
when 1 reader thread tries to acquire access database, it checks for number of writers waiting where if there is writers waiting, the reader thread has to wait

```java
class ReadWriteControllerPriority implements ReadWriteController {
  private int readers = 0;
  private boolean writing = false;
  private int numWaitingWriters = 0;

  public synchronized void acquireRead() throws InterruptedException {
    while (writing || numWaitingWriters>0) wait();
     ++readers;
  }

  public synchronized void releaseRead() {
    --readers;
    if (readers==0) notifyAll();
  }

  public synchronized void acquireWrite() throws InterruptedException {
    ++numWaitingWriters;
    while (readers>0 || writing) wait();
    --numWaitingWriters;
    writing = true;
  }

  public synchronized void releaseWrite() {
    writing = false;
    notifyAll();
  }
}

/******************* MAIN ************************/

public class ReadersWritersV1 {
  private static int numReaders = 2;
  private static int numWriters = 2;
```




- writers create a queue
- increment number of waiting writers when writer thread tries to get
- when there are two threads reading and a writer thread arrives, then number of waiting writers get incremented
- readers will keep waiting until writers have finished being served (numWaitingWriters is 0)
- **problem**: prioritising writers (readers start and they can keep)
- as there are two writers can alternate between getting the lock and accessing the database in write mode
- readers never read (writers always write) - need to give a fair chance to everyone to get access to the database for both readers/writers
- how to do that? (irl you can use a traffic light to regulate flow)
- can alternate between readers/writers to regulate/give fair chance to everyone

**alternating between reader/writers**

```
class ReadWriteControllerFair implements ReadWriteController {
  private int readers = 0;
  private boolean writing = false;
  private int numWaitingWriters = 0;
  private boolean readersTurn = false;

  synchronized public void acquireRead() throws InterruptedException {
    while (writing || (numWaitingWriters>0 && !readersTurn)) wait();
    ++readers;
  }

  synchronized public void releaseRead() {
    --readers;
    readersTurn=false;
    if(readers==0) notifyAll();
  }

  synchronized public void acquireWrite() throws InterruptedException {
    ++numWaitingWriters;
    while (readers>0 || writing) wait();
    --numWaitingWriters;
    writing = true;
  }

  synchronized public void releaseWrite() {
    writing = false;
    readersTurn=true;
    notifyAll();
  }
}

/****************** MAIN ***********************/

public class ReadersWritersV2 {
  private static int numReaders = 2;
  private static int numWriters = 2;
-- INSERT --
```

- take turns for reader/writers to access the database
- when reader tries to get access to data, it checks if there is already a writer already writing or if there are writers in the queue that want to write or if its the reader's turn. - thread wait if all conditions are satisfies, otherwise requerst goes throufgh
- when first reader gets thnrough and releases the read, it notifies that its not the reader's turn anymore - signal to admit writers to access the database
- we don't admit new readers after first reader releases Read lock
- whenever writer finishes writing, we let readers read, and so on vice versa

```
[Reader 1] reading...
[Reader 0] reading...
[Writer 1] writing...
[Writer 1] waiting for permission to write
[Reader 0] reading...
[Reader 1] reading...
[Writer 0] writing...
[Writer 0] waiting for permission to write
[Reader 1] reading...
[Reader 0] reading...
[Writer 1] writing...
[Writer 1] waiting for permission to write
[Reader 0] reading...
[Reader 1] reading...
[Writer 0] writing...
[Writer 0] waiting for permission to write
[Reader 1] reading...
[Reader 0] reading...
[Writer 1] writing...
[Writer 1] waiting for permission to write
[Reader 0] reading...
[Reader 1] reading...
[Writer 0] writing...
```

- execution shows working that there is a good alternate between readers/writers
- it is not fair in the thread level - fairness only comes from taking turns (hardcoded)

- if there is an imbalance bewteen numer of readers/writers, there can be some reader/writers that might not have access (depending on the surplus) e.g. some wruiters cannot write if there are more writers than readers so these writers might not get their chance to access

## Livelock

- Despite not blocked, a thread does not make any useful work
  - e.g. keeps retrying operations that always fail -don't make any progress
- 

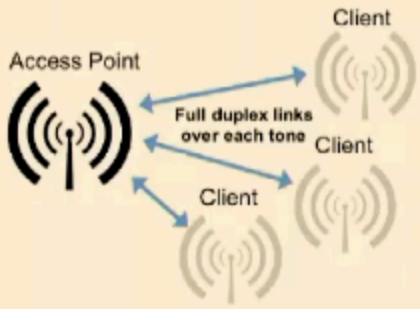
| Failing operation | Example | Reference |
|---|---|---|
| Change state (in response to other thread's state change) | Polite people in a hallway | [GPBBHL 10.3.3] |
| Faulty error recovery | Poison message problem | [GPBBHL 10.3.3] |
| | WifiLink | [Moodle] |

  -solution: add randomness to retry mechanism

posion message problem - message from a queue cannot be processed as it has an error, and it is put back in queue, repeats processing that message and putting back in queue => no useful work done

**WifiLink**

- Consider a simulation of transmissions on a Wi-Fi link
  - Wi-Fi clients transmit messages to an access point (AP)
  - Messages simultaneously transmitted by clients "collide", and they are not received correctly (i.e., signals mix up)
- Problem formulation
  - 1 AP and N clients
  - The AP waits for messages, and confirms correct receptions
  - Each client sends one message to AP, waits for confirmation, and resends if no confirmation
  - Messages and AP confirmations take time to reach destinations




## Missed Signals

- Happens due to incorrect **conditional sync** (refer to week 9)
- A thread keeps waitinfg for a condition that is already true, or was true in the past
- incorrectness often stems from inappropriate use of conditional synchronisation and/or code bugs
  - eg: condition predicate not checked before waiting
  - thread not waken up from a wait

recap - conditional sync

- Mechanism: condition queues give threads a way to subscribe to specific updates on a given condition
  - waiting threads form a so-called wait set
- Condition queues offer methods to
  - wait() allows threads to enter a condition queue
  - notify() wakes up one thread in the wait set
  - notifyAll() wakes up *all* threads in the wait set
- Important: notifications signal that "something has changed", ***not what*** has changed
  - threads must re-check condition when woken up