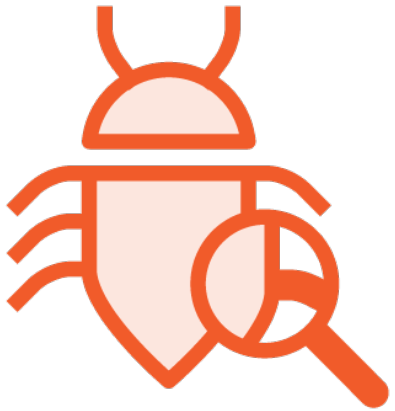
Liveness Issues: Preventing Deadlock, Livelock, and Starvation

David Flynn
SOFTWARE ENGINEER, FLYNN IT LTD

Mutexes and Misuse



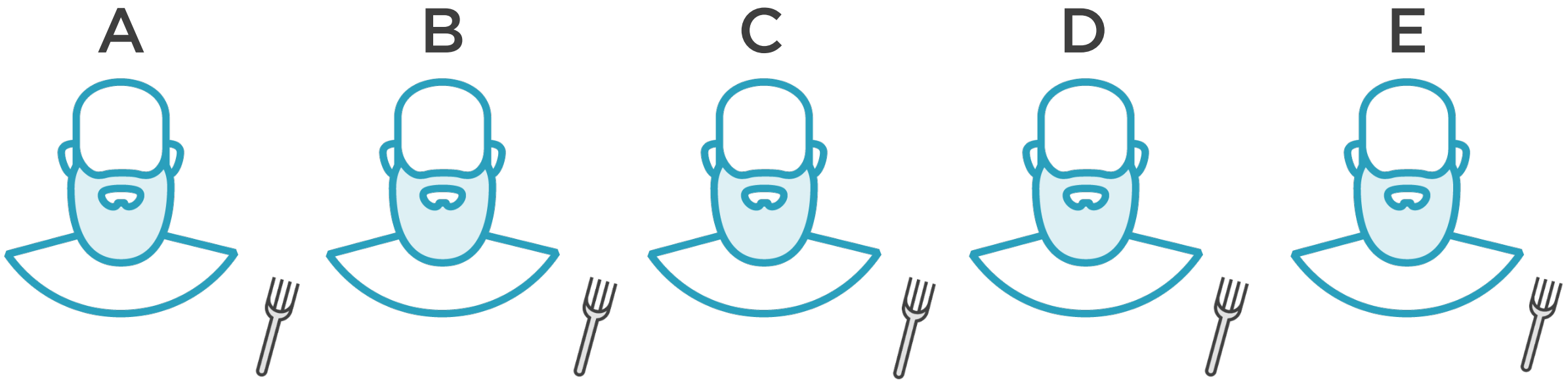
Three ‘liveness’ issues (affect how responsive the program is)

Deadlock

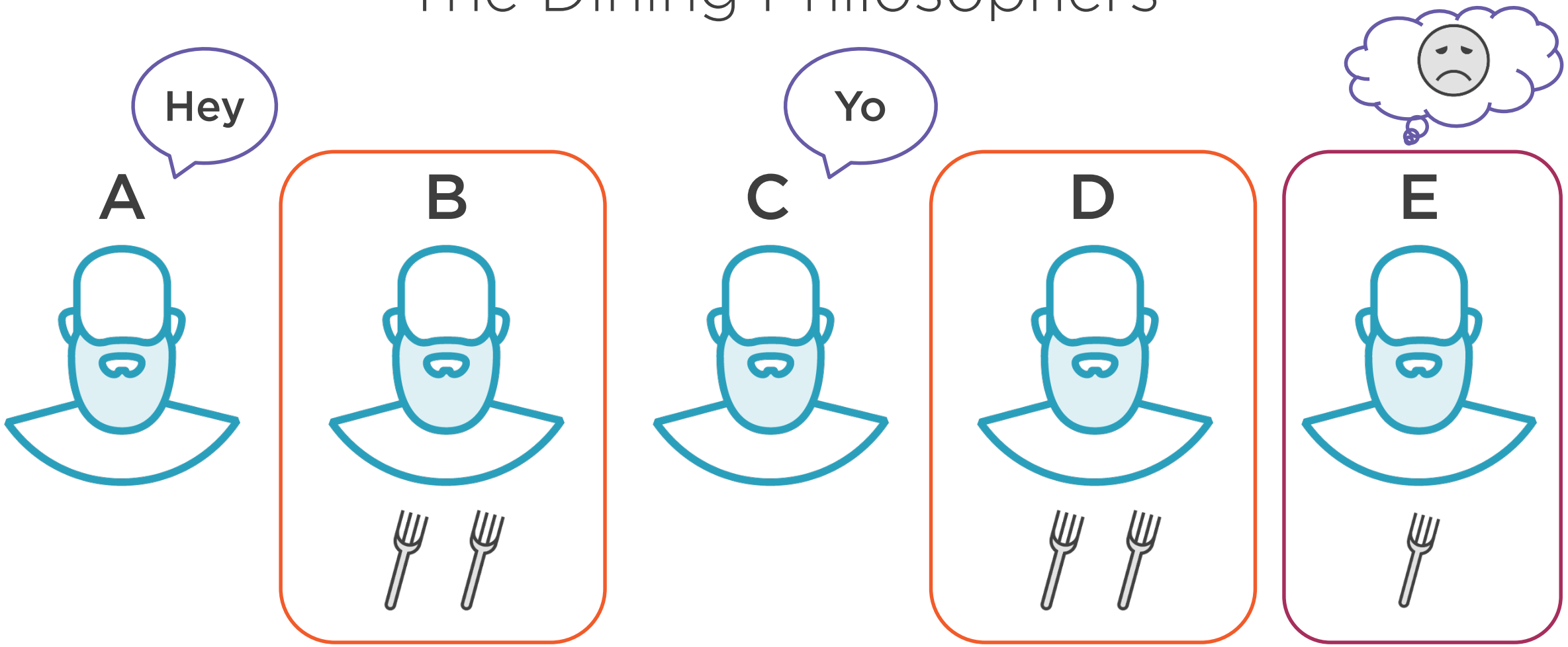
Livelock

Starvation

The Dining Philosophers



The Dining Philosophers



Philosophers Versus Mutexes

Each philosopher is like a thread

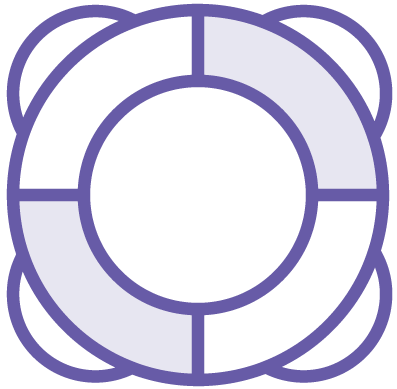
Each fork is like a mutex protecting the food

Mutexes can be used to protected resources

- Such as objects, files, databases or hardware

The philosophers example is a little different to how mutexes are normally used

Pros and Cons of Mutexes



Pros:

- Protect from race conditions and other synchronizing issues

Cons:

- May have to wait
- Incurs performance penalty

Trade-off safe vs performant

- Why shouldn't we make as much code synchronized as possible?

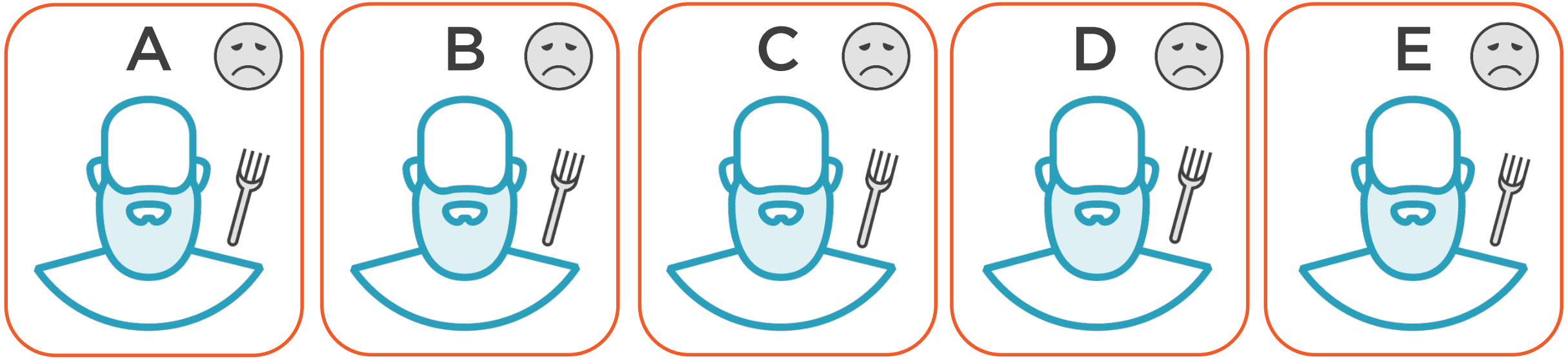
Successful Setup

Philosophers must be able to talk or eat as much of the time as possible

Minimize time attempting to acquire forks

Allow philosophers to eat soon after they feel the need

The Dining Philosophers

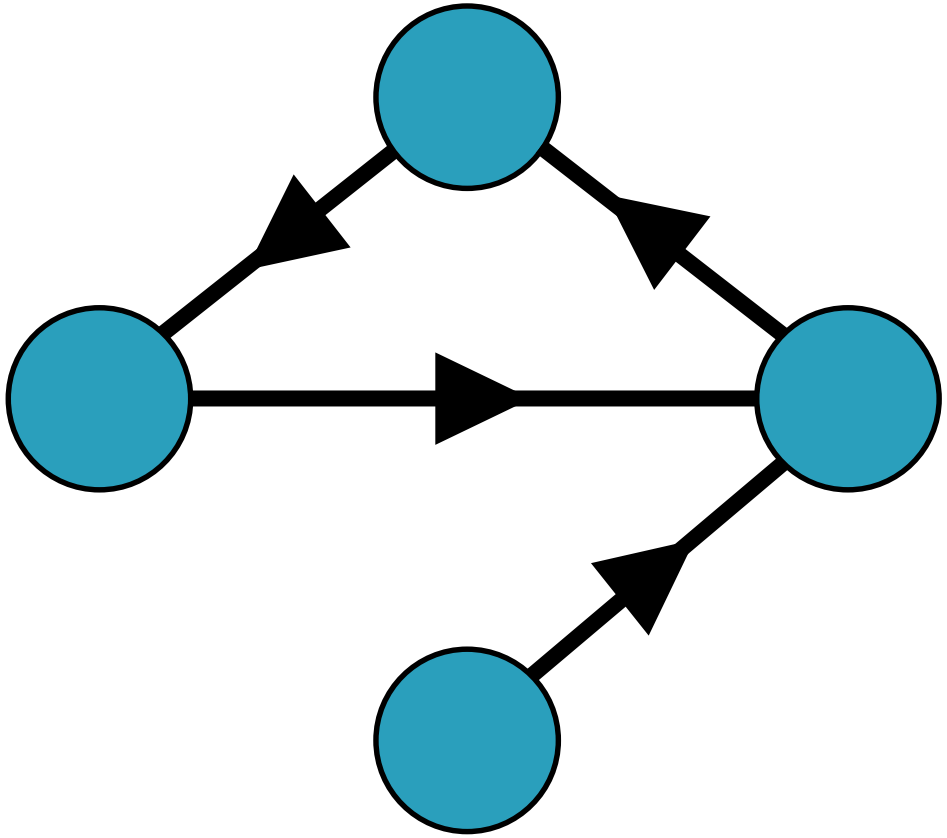


Deadlock

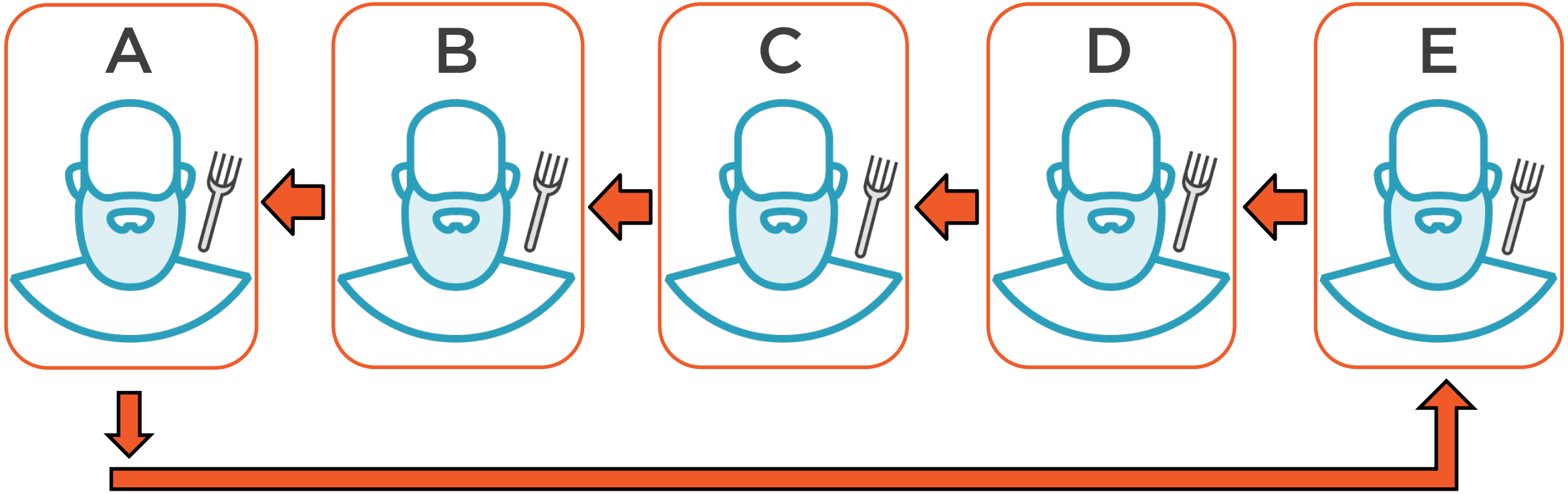
Threads cannot make progress

- The mutex is held by another thread(s)
- No thread acquires all the mutexes it requires
- No thread carries out its critical section
- Two or more threads are involved in Java deadlocks

Directed Graph



The Dining Philosophers



Consequence of Deadlock

No way to break deadlocks in Java
- Threads are permanently blocked

Application might appear locked

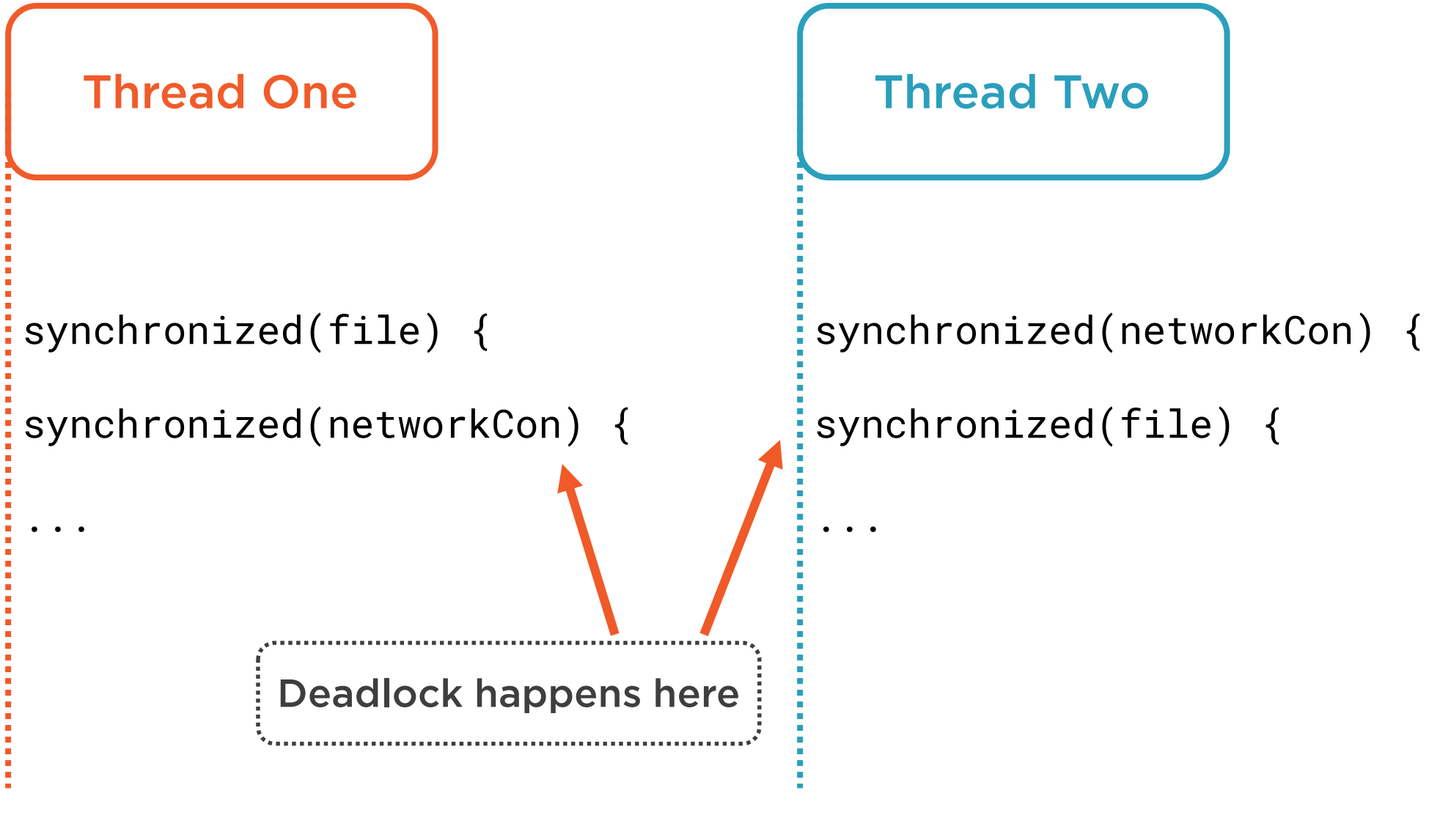
Restart will be required

Recovery

Restart the application and save logs

Users retrying might lock the system a second time

Fixing usually a high priority


Thread One
Thread Two
synchronized(file) {
synchronized(networkCon) {
...
synchronized(networkCon) {
synchronized(file) {
...
Deadlock happens here

Thread One

```
synchronized(file) {

Thread.sleep(1000)



synchronized(networkCon) {

...
```

Thread Two

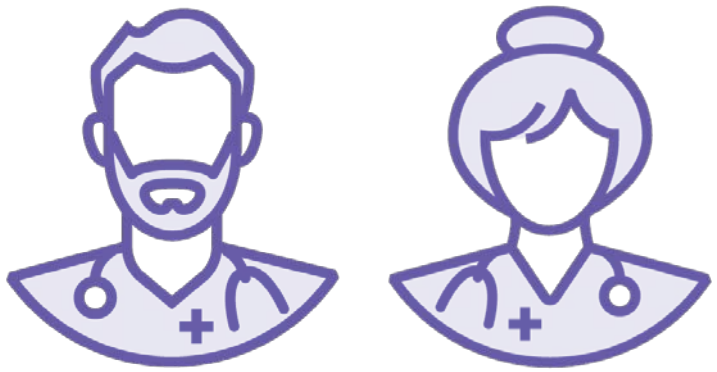
```
synchronized(networkCon) {

Thread.sleep(1000)

synchronized(file) {

...
```

Deadlock Diagnosis

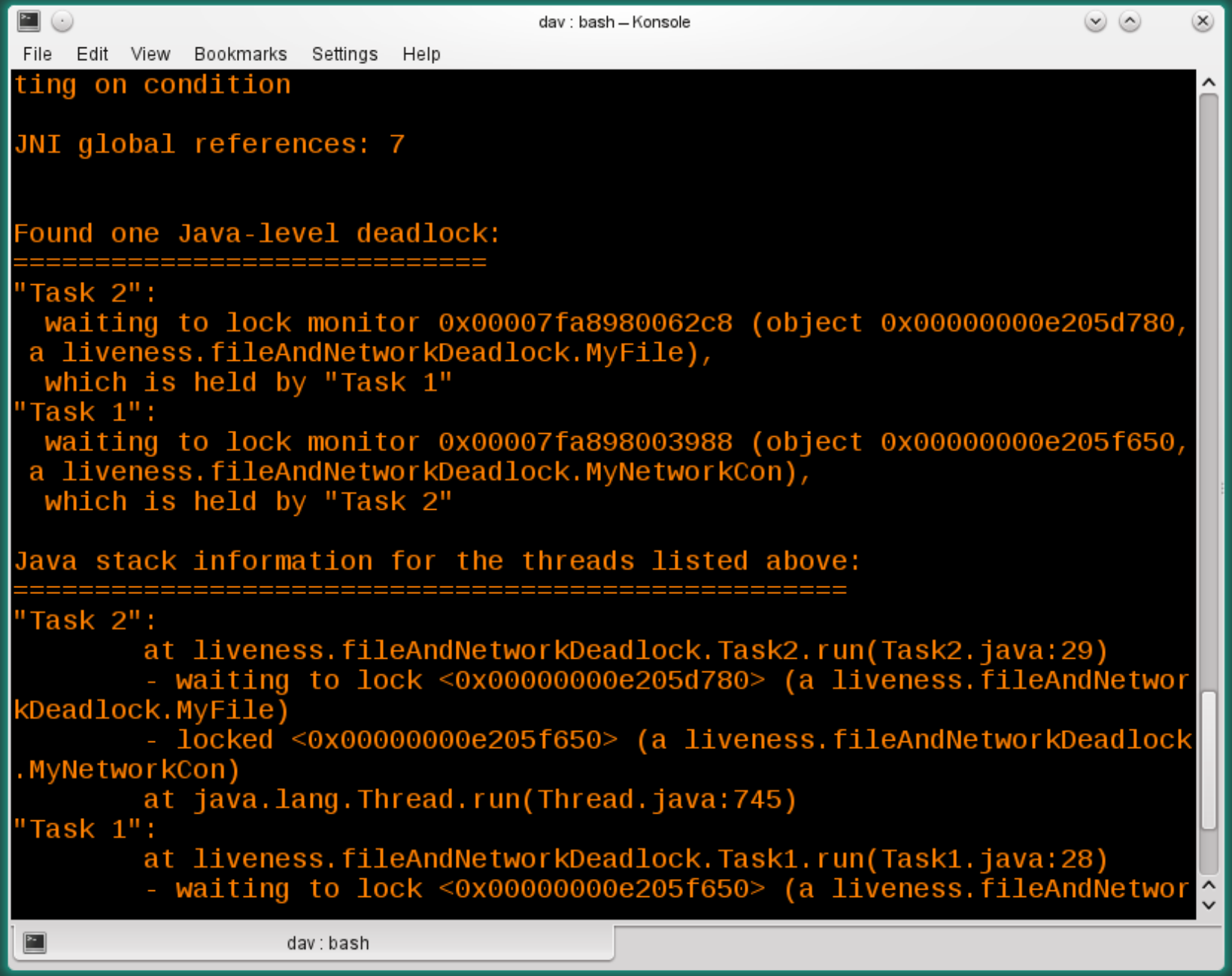


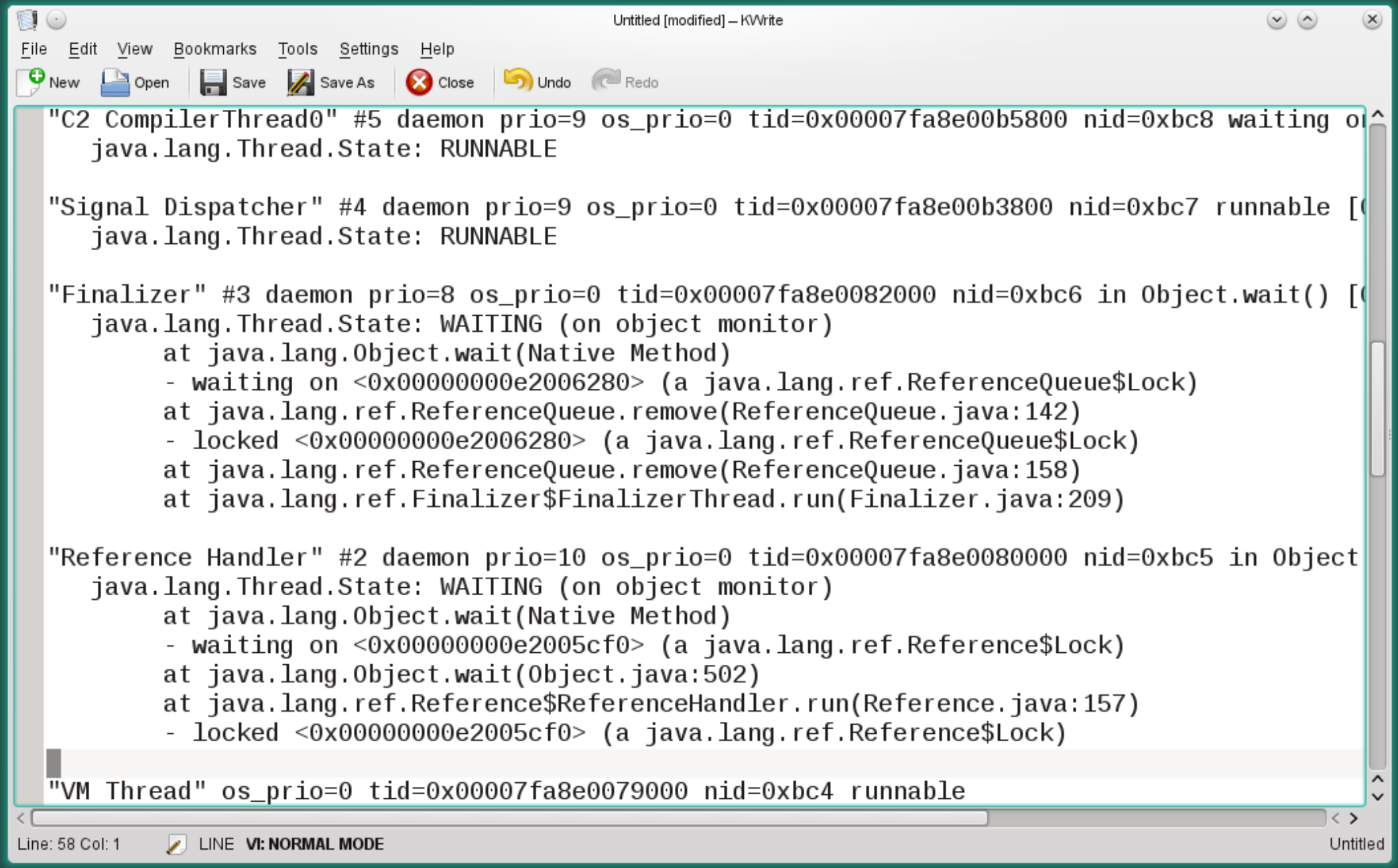
Look for no progress being made

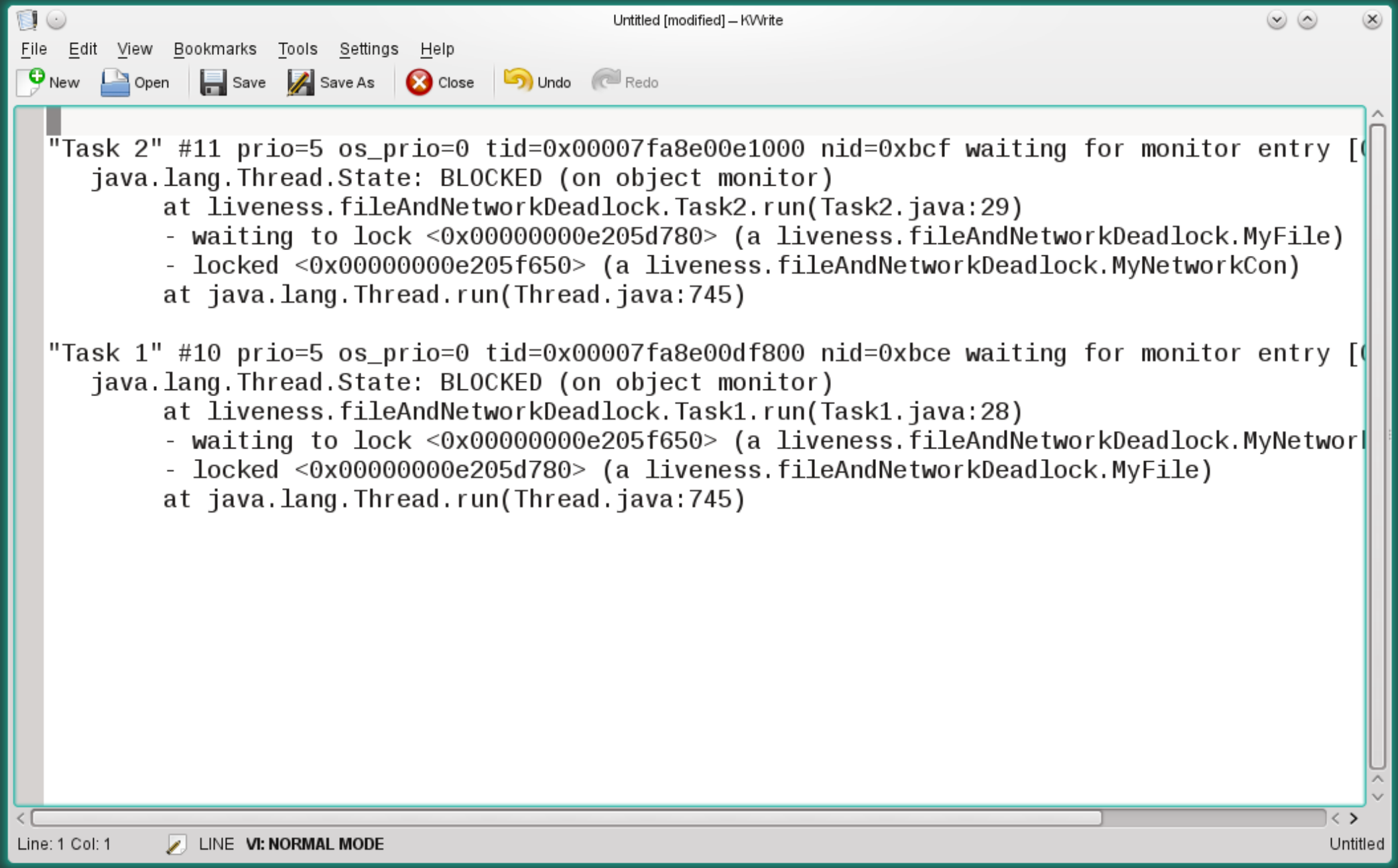
- Log files may indicate something froze and show no activity
- GUI might no longer be responsive/redrawn

Request jstacks from level one support

- Several stacks a few seconds apart


dav : bash — Konsole
File Edit View Bookmarks Settings Help
ting on condition
JNI global references: 7
Found one Java-level deadlock:
==========
"Task 2":
waiting to lock monitor 0x00007fa8980062c8 (object 0x00000000e205d780,
a liveness.fileAndNetworkDeadlock.MyFile),
which is held by "Task 1"
"Task 1":
waiting to lock monitor 0x00007fa898003988 (object 0x00000000e205f650,
a liveness.fileAndNetworkDeadlock.MyNetworkCon),
which is held by "Task 2"
Java stack information for the threads listed above:
==========
"Task 2":
at liveness.fileAndNetworkDeadlock.Task2.run(Task2.java:29)
- waiting to lock <0x00000000e205d780> (a liveness.fileAndNetwor
kDeadlock.MyFile)
- locked <0x00000000e205f650> (a liveness.fileAndNetworkDeadlock
.MyNetworkCon)
at java.lang.Thread.run(Thread.java:745)
"Task 1":
at liveness.fileAndNetworkDeadlock.Task1.run(Task1.java:28)
- waiting to lock <0x00000000e205f650> (a liveness.fileAndNetwor
dav : bash


Untitled [modified] – KWrite
File Edit View Bookmarks Tools Settings Help
New Open Save Save As Close Undo Redo
"C2 CompilerThread0" #5 daemon prio=9 os_prio=0 tid=0x00007fa8e00b5800 nid=0xbc8 waiting o
java.lang.Thread.State: RUNNABLE
"Signal Dispatcher" #4 daemon prio=9 os_prio=0 tid=0x00007fa8e00b3800 nid=0xbc7 runnable [
java.lang.Thread.State: RUNNABLE
"Finalizer" #3 daemon prio=8 os_prio=0 tid=0x00007fa8e0082000 nid=0xbc6 in Object.wait() [
java.lang.Thread.State: WAITING (on object monitor)
at java.lang.Object.wait(Native Method)
- waiting on <0x00000000e2006280> (a java.lang.ref.ReferenceQueue$Lock)
at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:142)
- locked <0x00000000e2006280> (a java.lang.ref.ReferenceQueue$Lock)
at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:158)
at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:209)
"Reference Handler" #2 daemon prio=10 os_prio=0 tid=0x00007fa8e0080000 nid=0xbc5 in Object
java.lang.Thread.State: WAITING (on object monitor)
at java.lang.Object.wait(Native Method)
- waiting on <0x00000000e2005cf0> (a java.lang.ref.Reference$Lock)
at java.lang.Object.wait(Object.java:502)
at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:157)
- locked <0x00000000e2005cf0> (a java.lang.ref.Reference$Lock)
"VM Thread" os_prio=0 tid=0x00007fa8e0079000 nid=0xbc4 runnable
Line: 58 Col: 1 LINE VI: NORMAL MODE Untitled


Untitled [modified] – KWrite
File Edit View Bookmarks Tools Settings Help
New Open Save Save As Close Undo Redo
"Task 2" #11 prio=5 os_prio=0 tid=0x00007fa8e00e1000 nid=0xbcf waiting for monitor entry [
java.lang.Thread.State: BLOCKED (on object monitor)
at liveness.fileAndNetworkDeadlock.Task2.run(Task2.java:29)
- waiting to lock <0x00000000e205d780> (a liveness.fileAndNetworkDeadlock.MyFile)
- locked <0x00000000e205f650> (a liveness.fileAndNetworkDeadlock.MyNetworkCon)
at java.lang.Thread.run(Thread.java:745)
"Task 1" #10 prio=5 os_prio=0 tid=0x00007fa8e00df800 nid=0xbce waiting for monitor entry [
java.lang.Thread.State: BLOCKED (on object monitor)
at liveness.fileAndNetworkDeadlock.Task1.run(Task1.java:28)
- waiting to lock <0x00000000e205f650> (a liveness.fileAndNetworkDeadlock.MyNetwor
- locked <0x00000000e205d780> (a liveness.fileAndNetworkDeadlock.MyFile)
at java.lang.Thread.run(Thread.java:745)
Line: 1 Col: 1
LINE VI: NORMAL MODE
Untitled

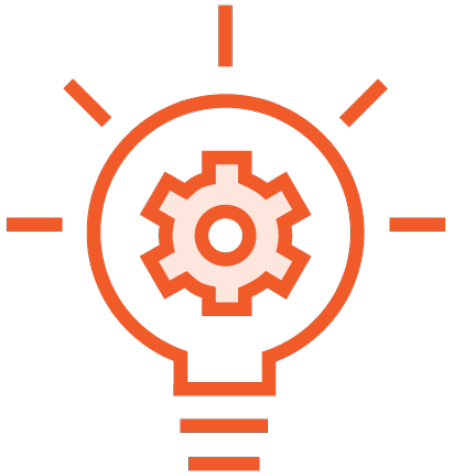
Other Deadlocks

Deadlocks don't have to involve just mutexes

One thread locking rows of tables in a database requests a mutex

Another thread has the mutex but waiting to access the locked rows

Deadlock Strategy One

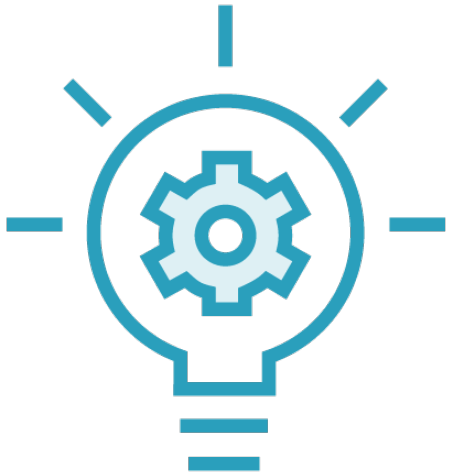


Always take the mutexes in the same order

Suspicion should always be raised if mutexes are taken in different orders

- Try opening the window to encourage a deadlock show up

Deadlock Strategy Two

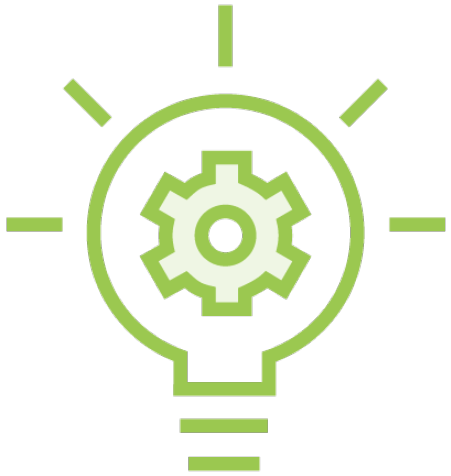


Replace two or more mutexes with a single one

If there are places where just one of those mutexes are taken

- Take a performance hit
- Now cannot be used simultaneously

Deadlock Strategy Three



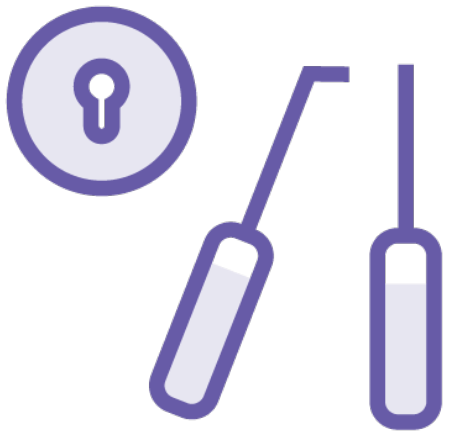
Use try-lock [back off and retry]

- If cannot get mutex immediately or after time period

Synchronized does not support this

- Locks do – see Java documentation

Try-lock Paradigm



If we fail to acquire the mutex:

- Release all mutexes
- Try again
- Hope another thread will succeed in the meantime

Good 'get of jail free card' – but ideally not first solution

Philosophers Strategy One

Place rules on order of forks taken

- Tricky as seen already

If based on what others are doing

- Risks race conditions, as in the threads in the bar example

Philosophers Strategy Two

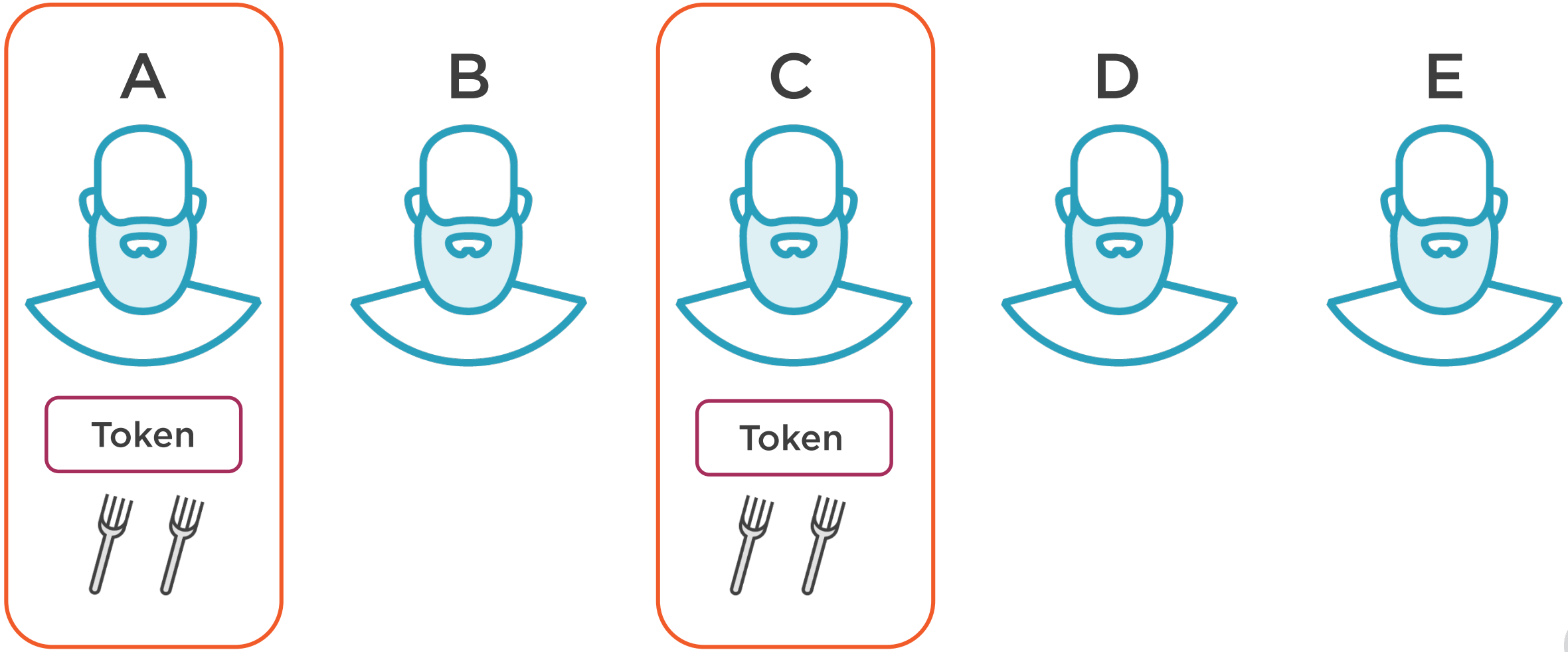
Two tokens acts as mutexes

- If token can be taken – two forks are then picked up

Guarantees two philosophers can always eat

Forks and token are replaced when done

Philosophers Strategy Two



Philosophers Strategy Three

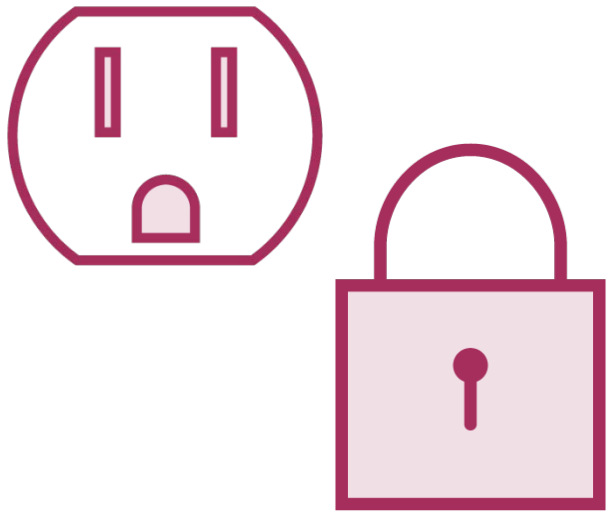
Five second rule (timeout after 5 seconds, retry after 5 seconds)

- Of course when dealing with threads the periods are a lot smaller!

Problem recurs if they all drop their forks and retry simultaneously

- Can philosophize during retry wait period [not deadlocked]
- This is livelock

Livelock



Threads are permanently blocked as in deadlock – so can:

- Back off and retry
- Wait
- Do some other work
- Try to resolve the situation

Still can't take all mutexes

- Or takes a long while

Livelock

'A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing.'

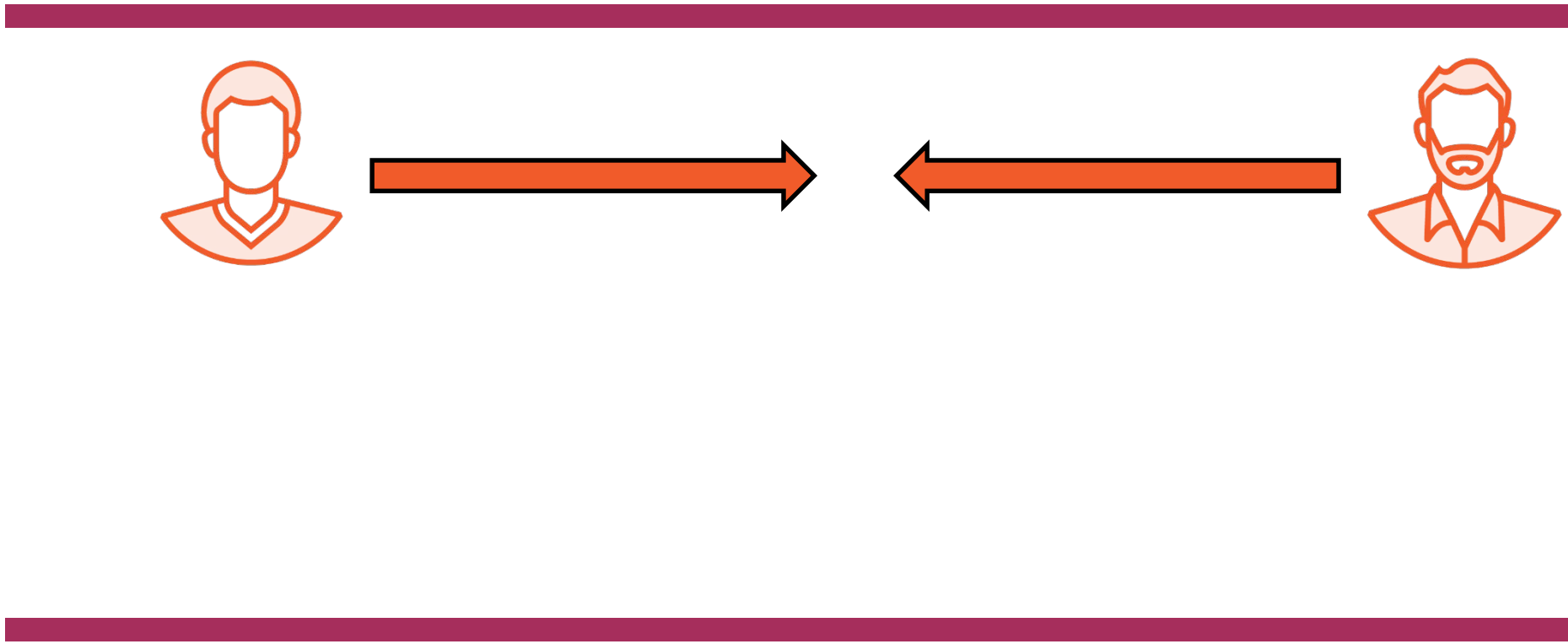
- Operating System Design, Wikibooks

Livelock

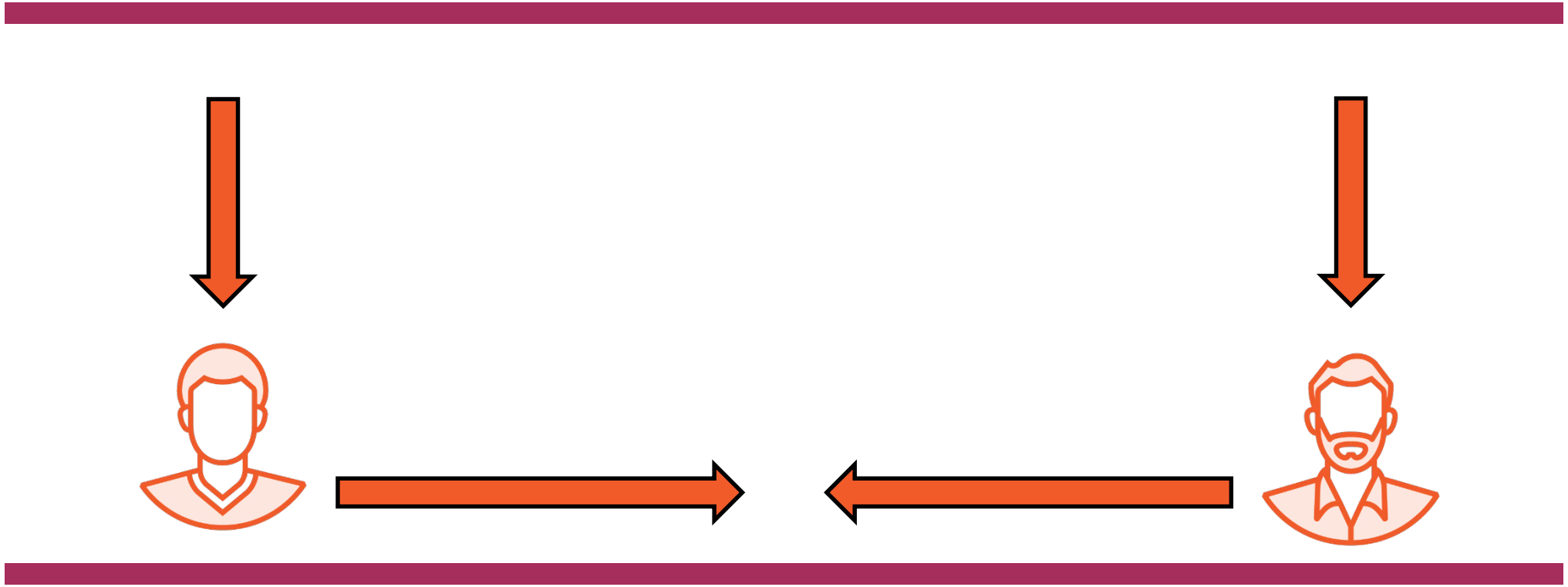
'... the threads are not blocked – they are simply too busy responding to each other to resume work.'

– Oracle, Java Tutorials

Responding Livelock



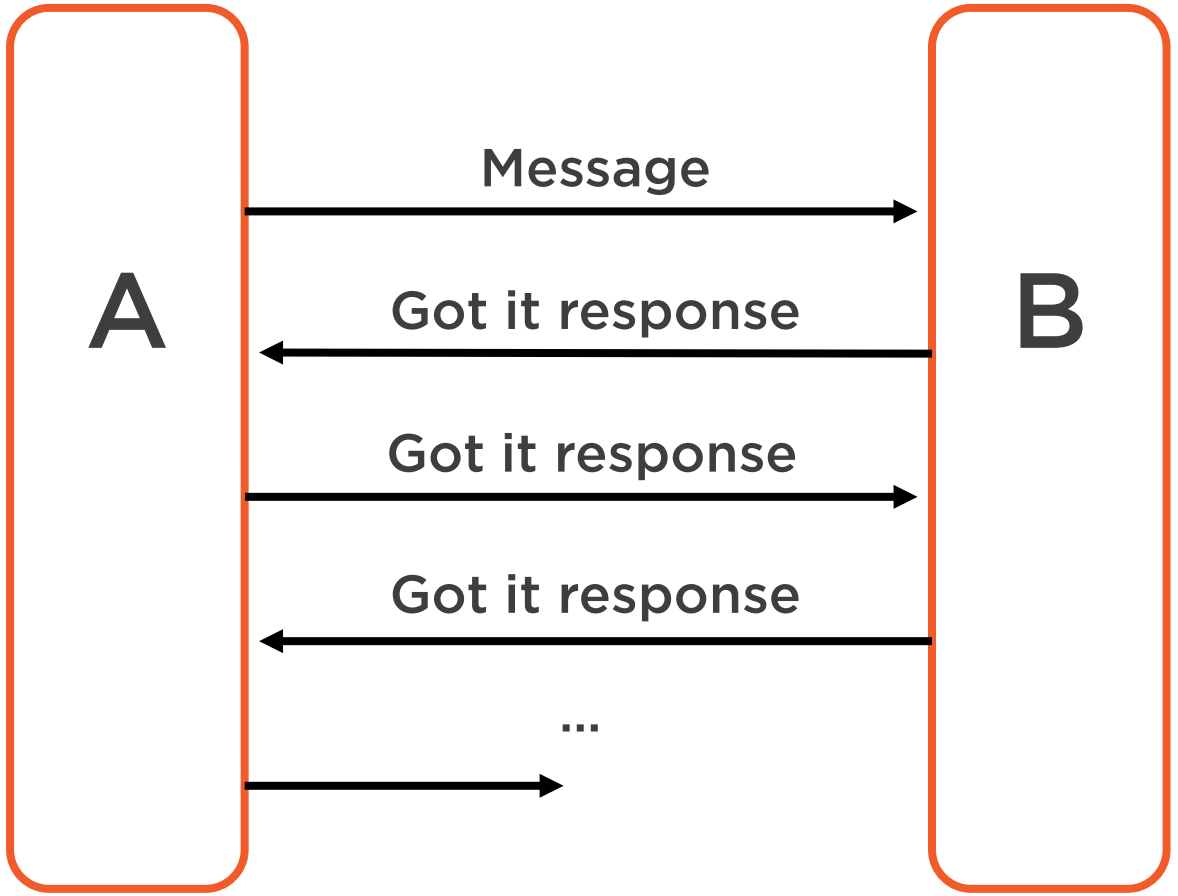
Responding Livelock



Responding Livelock [Non-mutex Issue]

Threads frustrate each other's attempt to progress

Responding Livelock



Infinite-Retry Livelock

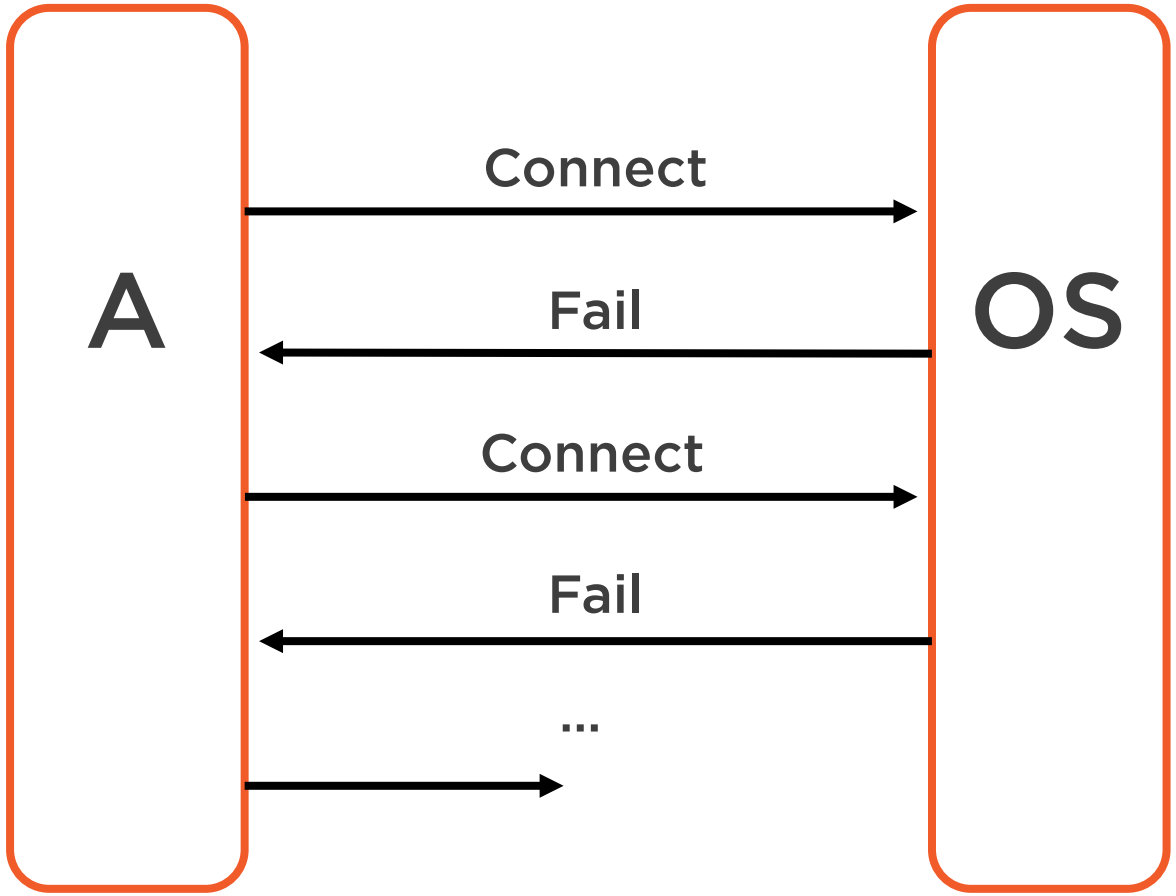
Doesn't give up quickly when retry futile

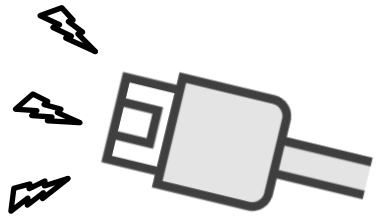
- Confusion of recoverable and unrecoverable errors

E.g.: Unrecoverable situation of cable removed

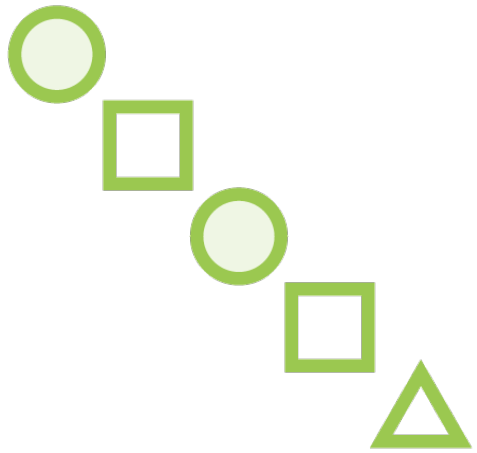
- Should send a message to the user instead of constant retry

Responding Livelock





Resource Try-Locking Livelock



As the name suggests when using try-lock

Try dropping the mutexes to allow another thread to succeed

- But if there is no break, we might pick them up again
- Or if the threads time out in sync, they pick up the same mutexes

May clear up by itself, but wastes performance

Dealing with Responding Livelock

Bad design – might need to rethink it

Arbitration may help in the corridor case

Dealing with Infinite-Retry Livelock

Should let failures happen [perhaps after a retry]

Is failure case likely to be a temporary glitch and should we fail?

Is it acceptable to disable [part of] the program?

Is the user the best one to resolve this?

Dealing with Resource Try-Locking Livelock

Three strategies to try

Should attempt deadlock strategies first

Strategy One

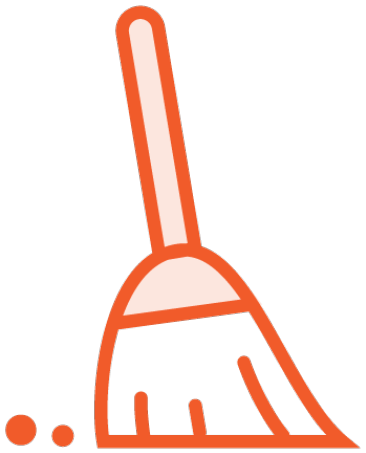
Vary try-lock time using random or predefined values

- Idea to prevent threads timing out in sync

Unlikely to be effective:

- Timing out in sync is possible but unlikely
- We might reacquire all the mutexes immediately

Strategy Two



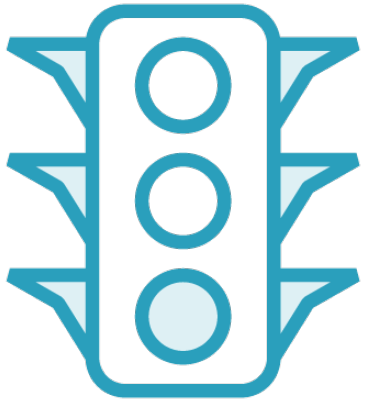
Prevent mutexes being taken again immediately

- E.g. Do some other work first
- Give time for other threads waiting to succeed

If no other work to do:

- Sleep for small period
- Log something

Strategy Three



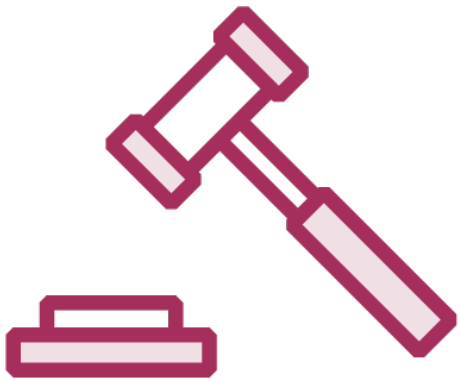
Arbitrate if cannot acquire mutexes

Another thread [arbiter] can suggest:

- Give way
- Give up
- Wait

May also help with responding livelock

Arbitration in Practice



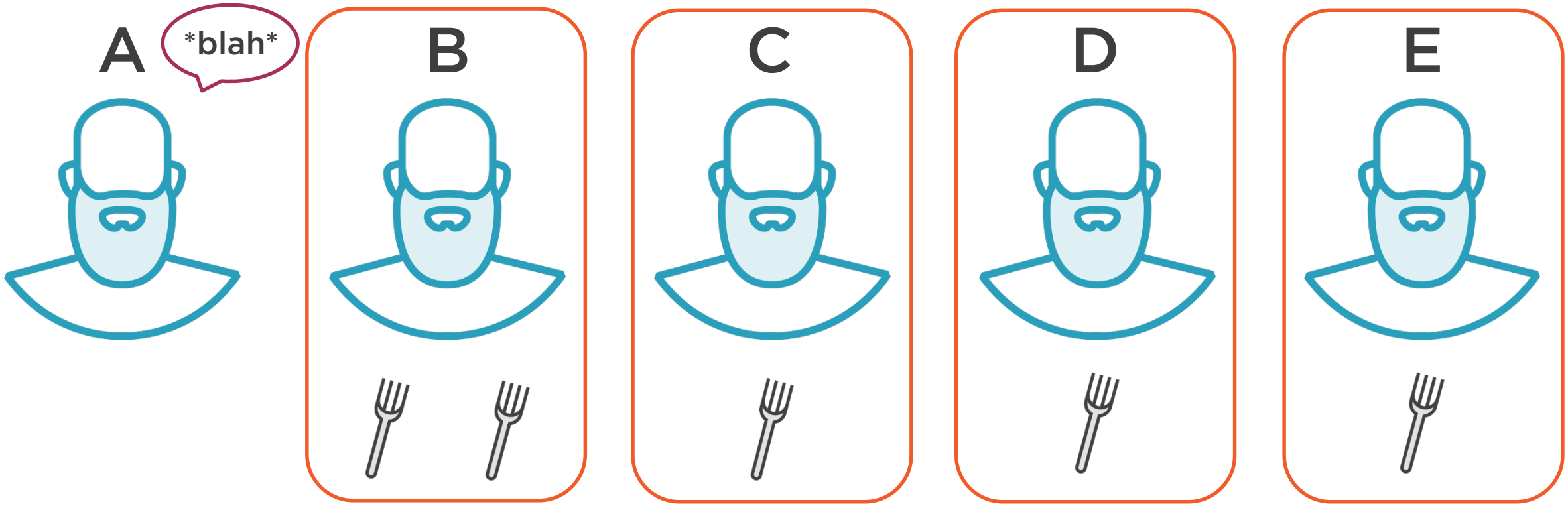
Threads inform arbiter before taking mutexes

- And then whether they succeeded or failed
- On failure, arbiter informs thread what to do

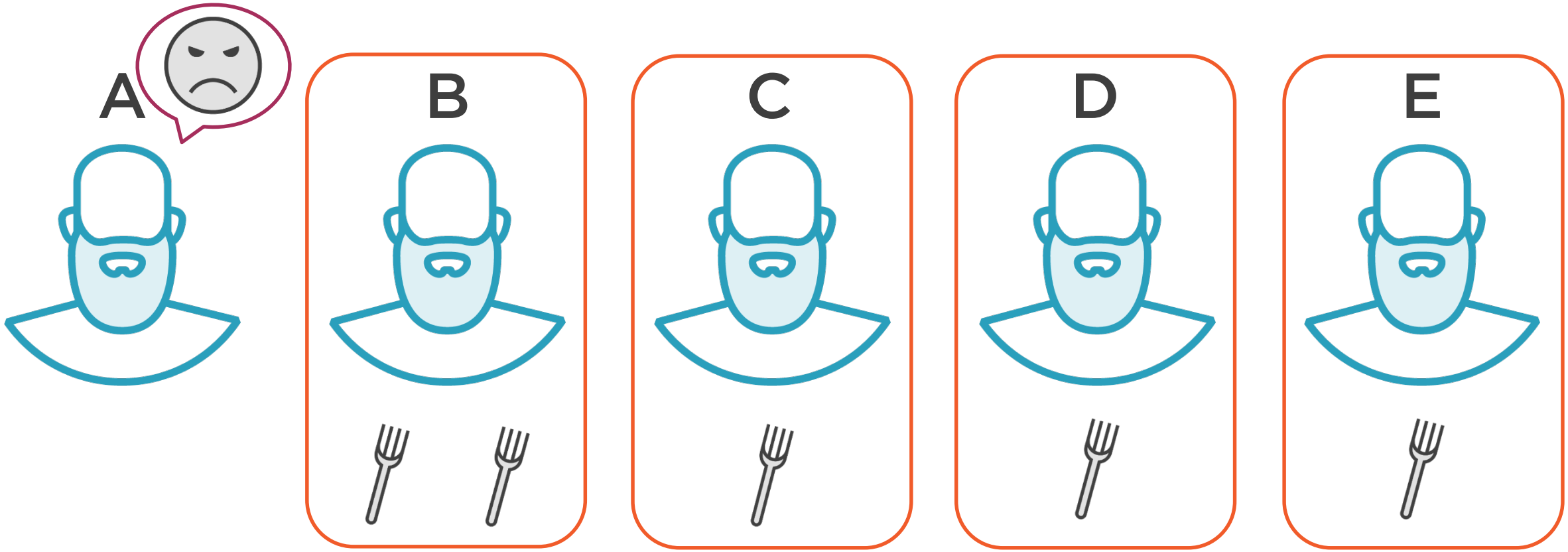
Easy solution, all other threads except one back off

Arbitration can be tricky and cause extra overhead

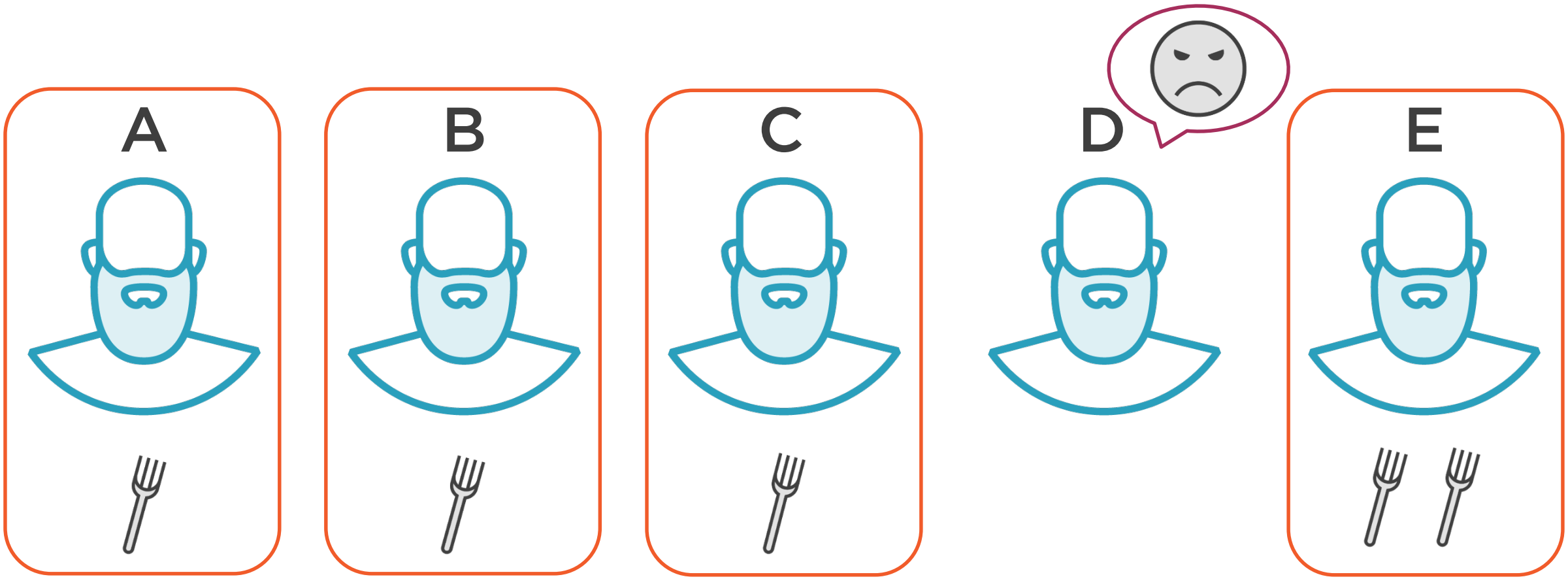
The Dining Philosophers



The Dining Philosophers



The Dining Philosophers

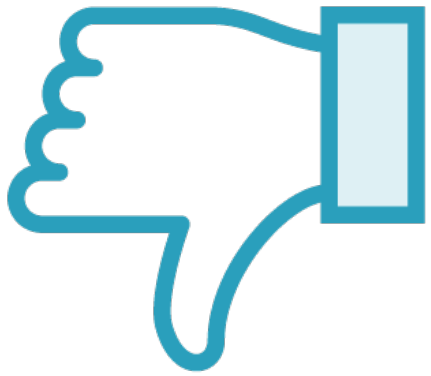


Starvation

When threads are not getting enough execution time to carry out their tasks

- Sometimes some getting significantly more than others [favored]

Favored Starvation Example



E.g. a live lossless data stream

- If threads working on earlier parts of the stream starve – stream may stop

Perhaps caused by access to a data structure which only allows writes if no readers

- If constant flow of readers, writers starve
- 'Readers/writers' problem

Solving Favored Starvation - Arbitration

Favor threads working on earlier parts of the stream

Request those working on later parts to back off – arbitration

Backed off threads sleep or be provided with less work

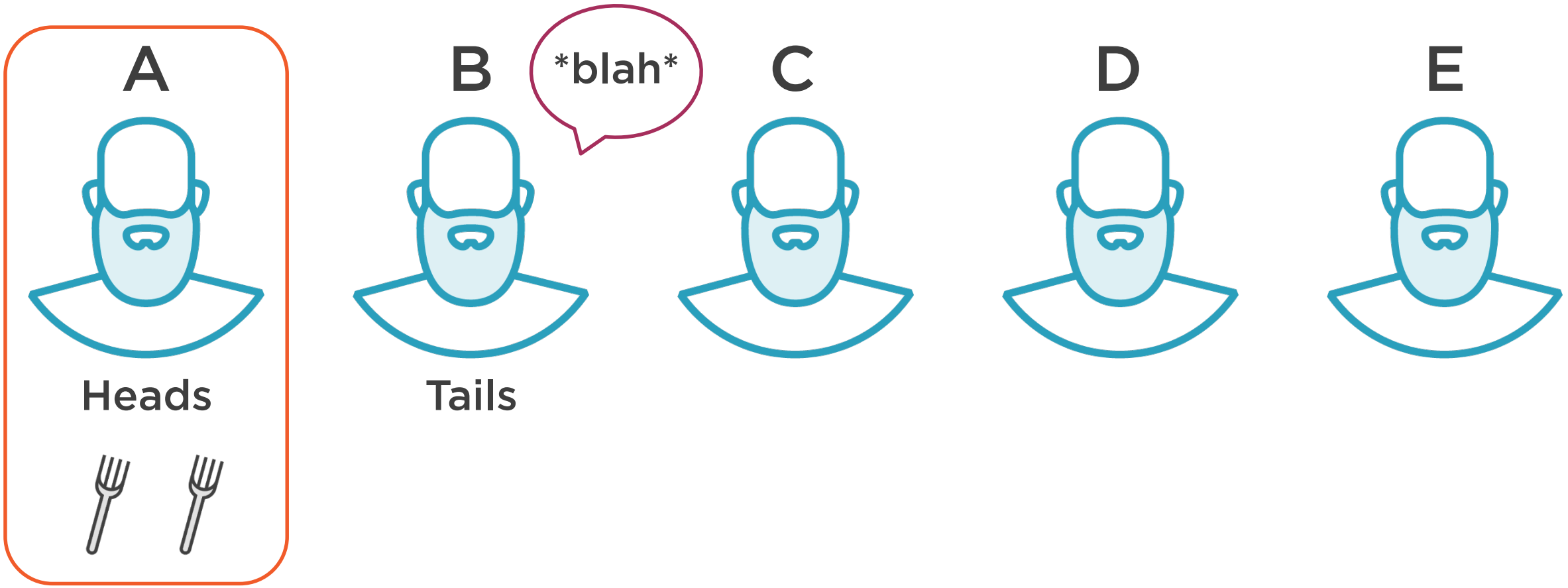
Solving Readers/Writers Issue

Could try favoring writers

- But need to be careful writers don't starve readers

Or block readers for a while to let writers succeed

The Dining Philosophers



Starvation Due to Unfair Scheduling

Out of our control – we'd hope the JVM scheduler is fair though

What if we changed priorities of threads?

- Danger of higher priority threads causing starvation of lower priority ones
- System dependent
- Should be very careful doing this

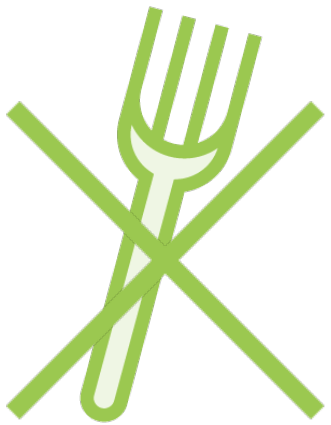
Work Stealing

If thread is falling behind

Another thread can steal some of its work

It then discards any stolen items

Starvation Due to Lack of Resources



Insufficient resources for the threads

Causes:

- Too many threads
- Too little system resources
- Attacks [e.g. Denial of Service]

Probably affect all the threads

Solving Resource Starvation

Reduce numbers of threads

- Amdahl's law and experimentation

Monitor system resources to look for shortages

- Maybe increase if possible

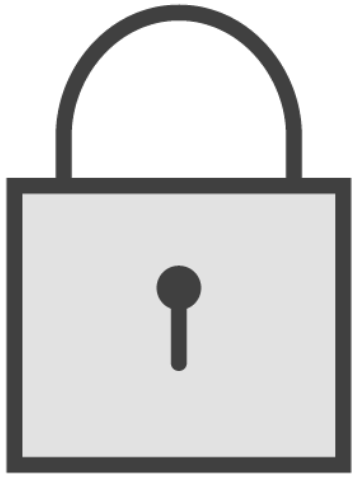
Liveness Issues from Mutexes

Deadlock

Livelock

Starvation

Deadlock



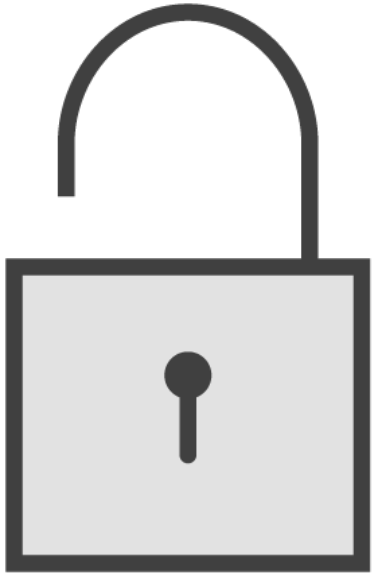
One thread waiting on a mutex another has

- That thread is waiting on a mutex another has...
- No thread can progress

Diagnose by noticing application is not progressing

- Inspect logfiles/jstack

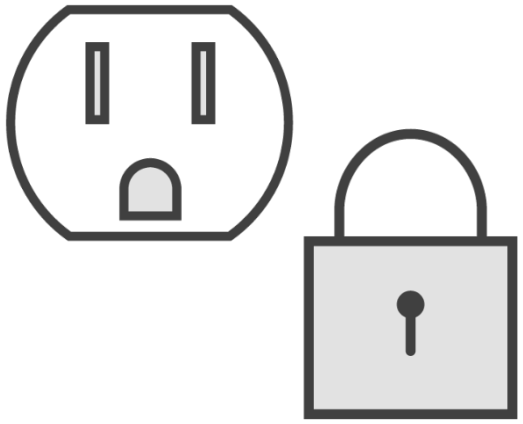
Solving Deadlock



Strategies

- Take mutexes in same order
- Reduce number of mutexes
- Use try-lock

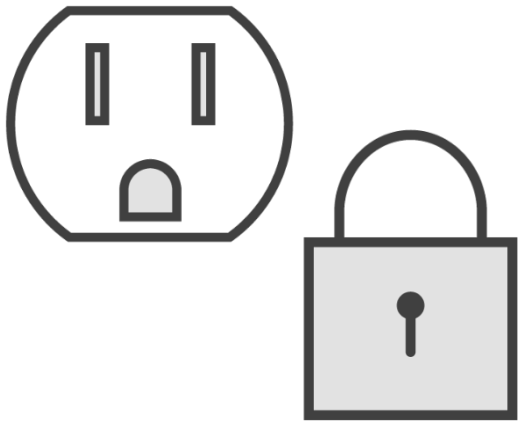
Livelock



E.g. Threads unable to get resources needed, but not permanently blocked

- Threads cannot progress, but able to back off and retry

Three Livelock Types



Responding livelock

- Rethink design, maybe arbitration

Infinite-retry livelock

- Should fail operation

Resource try-lock livelock

- Deadlock strategies, alter timeout, wait before retrying, arbitration

Starvation

Some or all threads unable to carry out work

- Not getting enough execution time

Causes:

- Unfair scheduler
- Altering thread priorities
- Insufficient resources
- Favoring threads

Solving Starvation

Favoring threads (readers/writers):

- Block threads letting others succeed

Arbitration

Work stealing

Resource starvation:

- Tune threads better
- Sort out the resourcing