THREADS

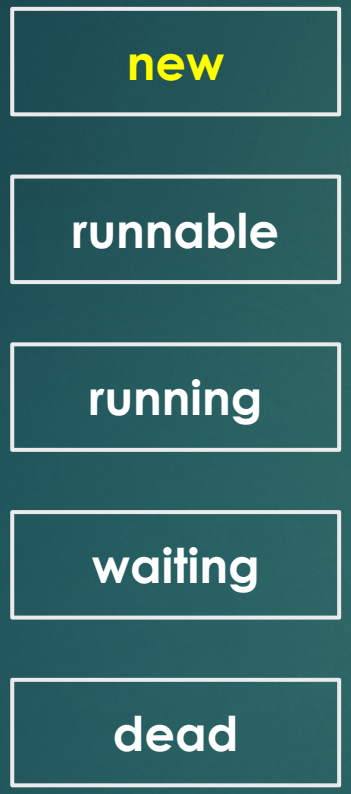
LIFE CYCLE

Threads: there are various stages → thread is born, it is started, it runs and it dies

Threads: there are various stages → thread is born, it is started, it runs and it dies

- **new**
- **runnable**
- **running**
- **waiting**
- **dead**

Threads: there are various stages → thread is born, it is started, it runs and it dies



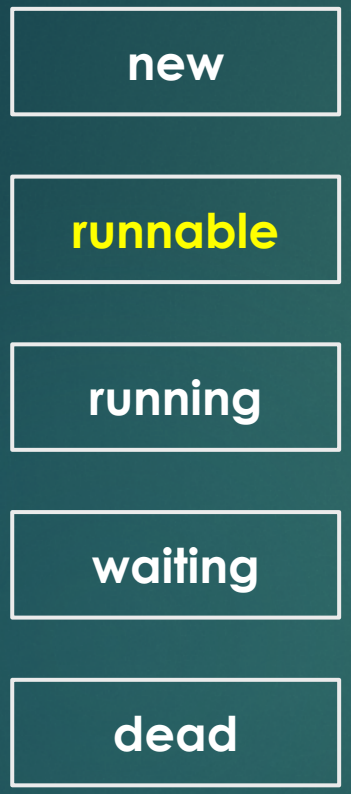
1.) **NEW** when we instantiate a thread
It is in this state until we start it
~ **start()**

2.) **RUNNABLE** after we have started the thread
The thread is executing its task in this state

3.) **WAITING** when a thread is waiting: for example for another thread to finish its task
When other thread signals → this thread goes back to the ‚runnable' state
~ **wait()** and **sleep()**

4.) **DEAD** after the thread finishes its task

Threads: there are various stages → thread is born, it is started, it runs and it dies



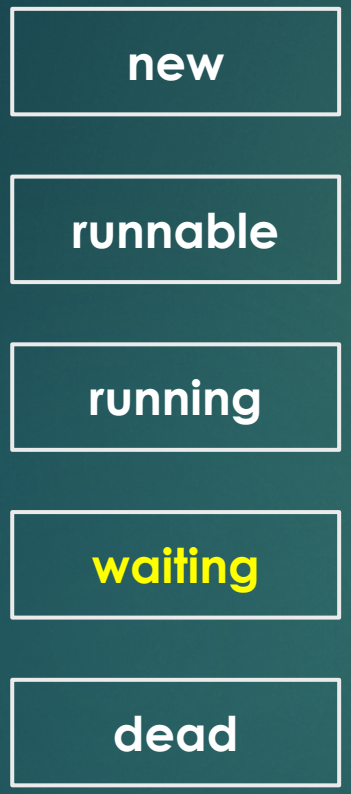
1.) **NEW** when we instantiate a thread
It is in this state until we start it
~ **start()**

2.) **RUNNABLE** after we have started the thread
The thread is executing its task in this state

3.) **WAITING** when a thread is waiting: for example for another thread to finish its task
When other thread signals → this thread goes back to the ‚runnable' state
~ **wait()** and **sleep()**

4.) **DEAD** after the thread finishes its task

Threads: there are various stages → thread is born, it is started, it runs and it dies



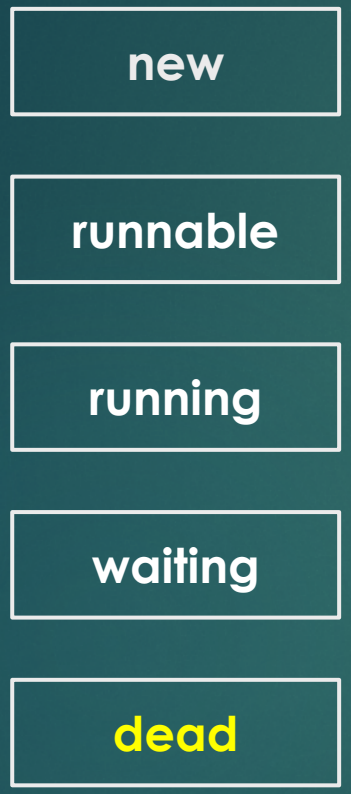
1.) **NEW** when we instantiate a thread
It is in this state until we start it
~ **start()**

2.) **RUNNABLE** after we have started the thread
The thread is executing its task in this state

3.) **WAITING** when a thread is waiting: for example for another thread to finish its task
When other thread signals → this thread goes back to the ‚runnable' state
~ **wait()** and **sleep()**

4.) **DEAD** after the thread finishes its task

Threads: there are various stages → thread is born, it is started, it runs and it dies



1.) **NEW** when we instantiate a thread
It is in this state until we start it
~ **start()**

2.) **RUNNABLE** after we have started the thread
The thread is executing its task in this state

3.) **WAITING** when a thread is waiting: for example for another thread to finish its task
When other thread signals → this thread goes back to the ‚runnable' state
~ **wait()** and **sleep()**

4.) **DEAD** after the thread finishes its task

MULTITHREADING

ADVANTAGES

Advantages

- We can design more responsive softwares → do several things at the same time
- We can achieve better resource utilization
- We can improve performance !!!

Downloading images
from the web

IO operations: copying files
and parse the content

Doing heavy calculations
For example: simulations, numerical methods

Downloading images
from the web

THREAD #1

IO operations: copying files
and parse the content

THREAD #2

Doing heavy calculations
For example: simulations, numerical methods

THREAD #3

MULTITHREADING

DISADVANTAGES

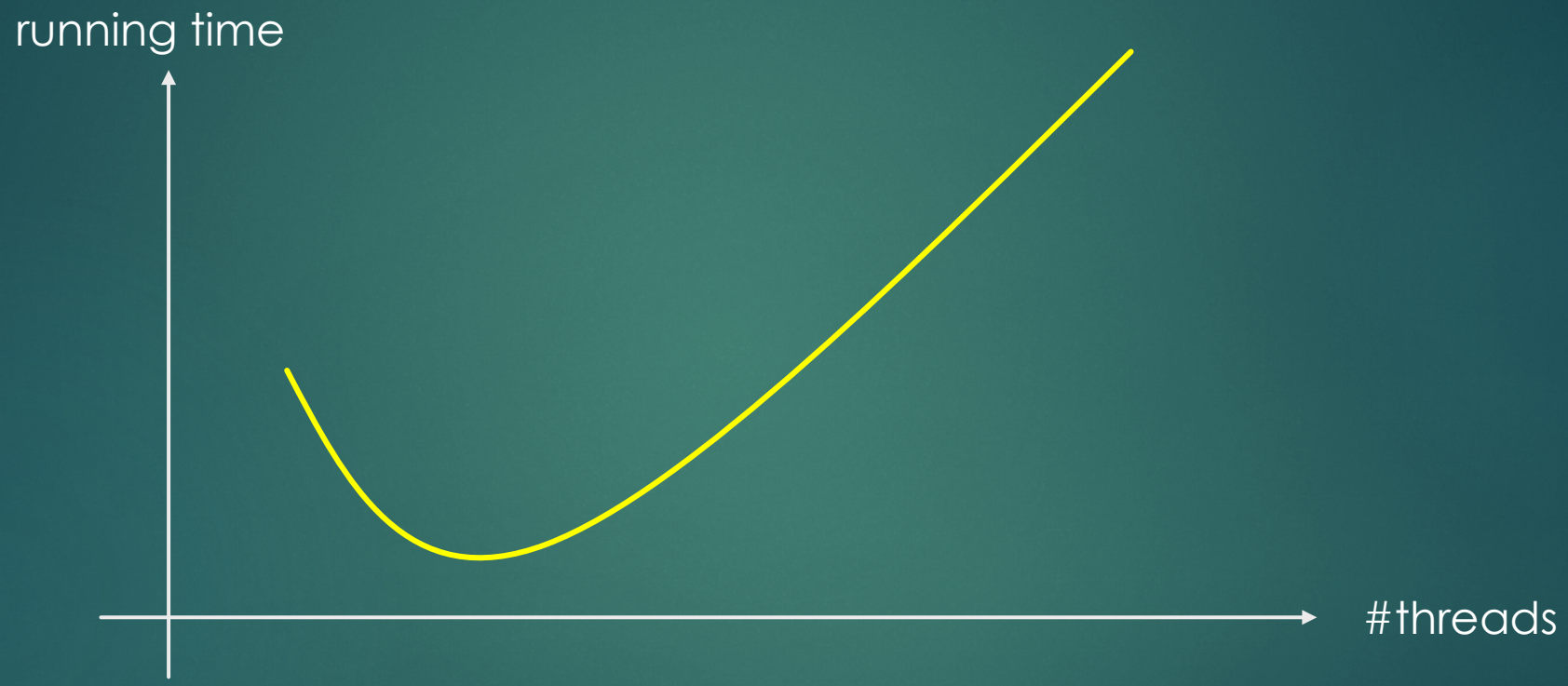
Disadvantages

- Of course there are some costs involved in multithreading
- Multithreading is not always better !!!
- Threads manupulate data located on the same memory area → we have to take it into consideration
- Difficult to design multithreaded software
- Hard to detect errors
- **EXPENSIVE OPERATION**: switcing between threads is exspensive !!!

  ~ CPU save local data, pointers ... of the current thread

  + loads the data of the other thread

Rule of thumb: for small problems it is unnecessary to use multithreading,
it may be slower than single threaded applications
~ multithreaded sorting is slower for small number of items



MULTITHREADING

DEADLOCK AND LIVELOCK

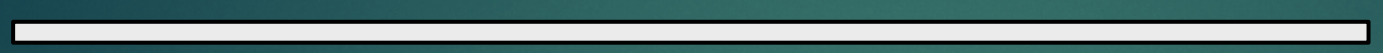
Deadlock

- Deadlock is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does
- Databases → deadlock happens when two processes each within its own transaction updates two rows of information but in the opposite order. For example, process **A** updates row **1** then row **2** in the exact timeframe that process **B** updates row **2** then row **1** !!!
- Operating system → a deadlock is a situation which occurs when a process or thread enters a waiting state because a resource requested is being held by another waiting process, which in turn is waiting for another resource held by another waiting process

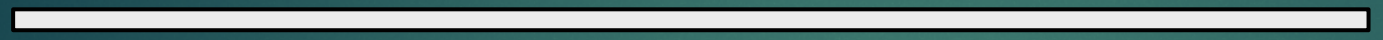
Livelock

- A thread often acts in response to the action of another thread
- If the other thread's action is also a response to the action of another thread → livelock !!!
- Livelocked threads are unable to make further progress. However, the threads are not blocked → they are simply too busy responding to each other to resume work
- Like two people attempting to pass each other in a narrow corridor: **A** moves to his left to let **B** pass, while **B** moves to his right to let **A** pass. They are still blocking each other, **A** moves to his right, while **B** moves to his left ... still not good

Parallel *versus* multithreading

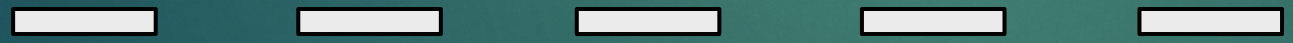


thread #1

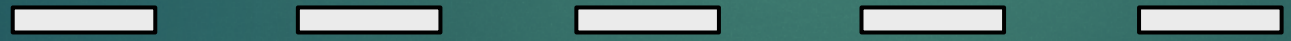


thread #2

„parallel execution"



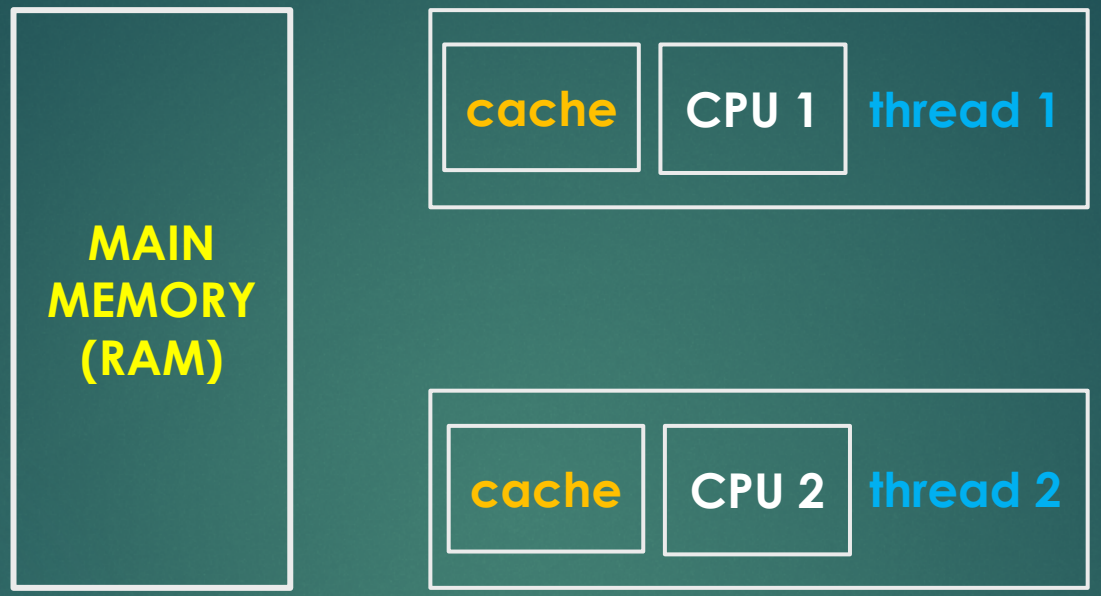
thread #1



thread #2

„multithreaded execution" (with time-slicing)

Volatile

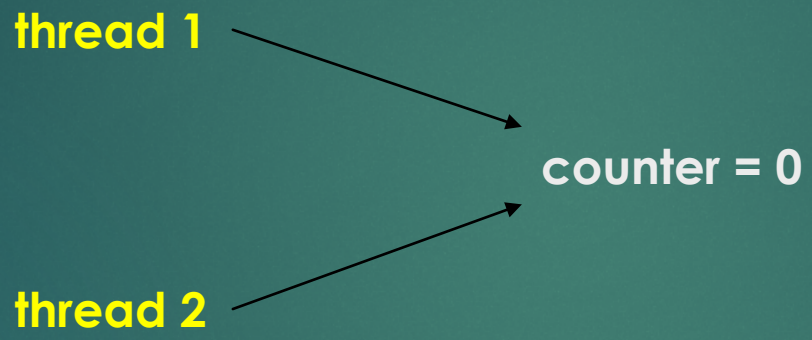


Every read of a volatile variable will be read from the **RAM** so from the main memory

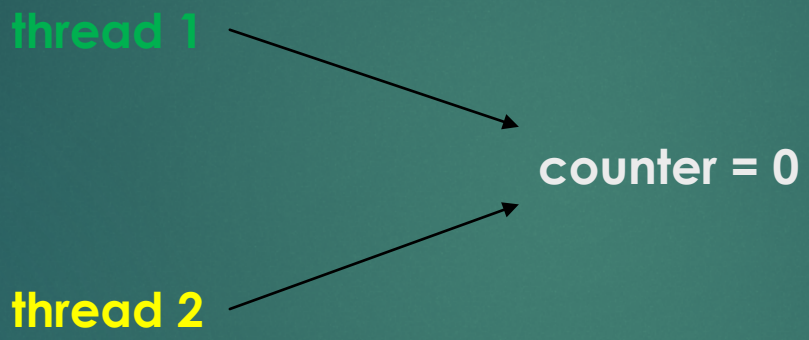
// not from cache, usually variables are cached for performance reasons

Caches are faster → do not use ***volatile*** keyword if not necessary (+ it prevents instruction reordering which is a performance boost technique)

Volatile

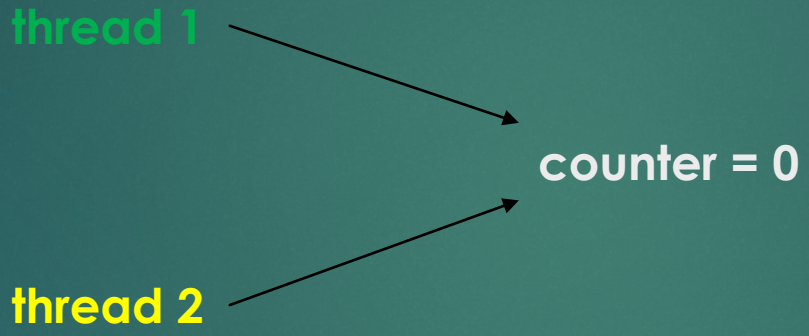


Volatile



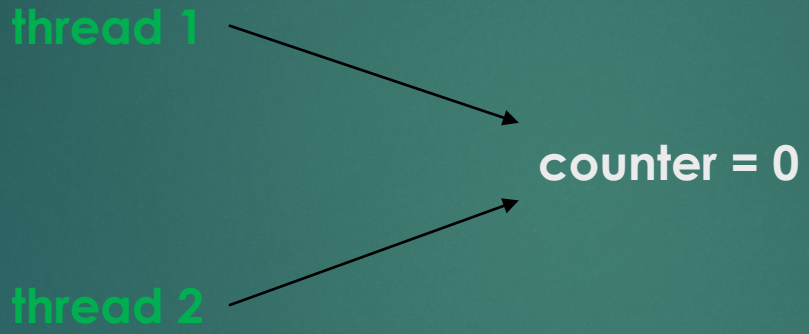
Volatile

counter = counter + 1

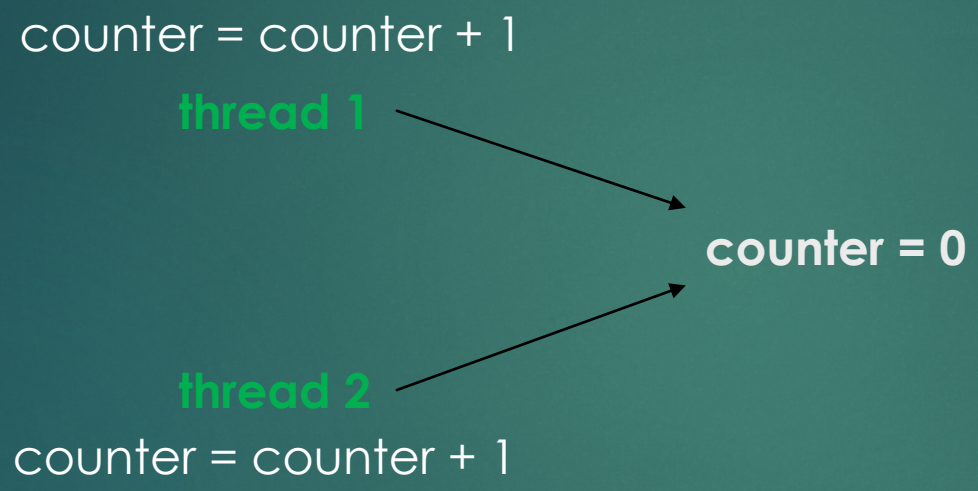


Volatile

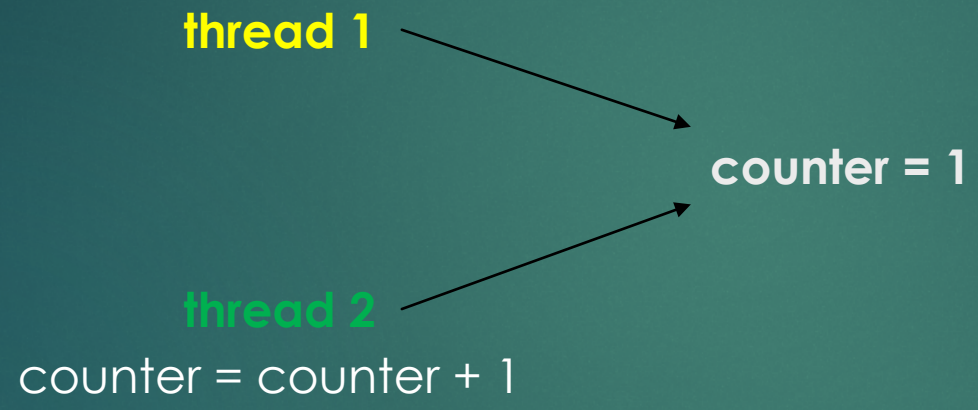
counter = counter + 1



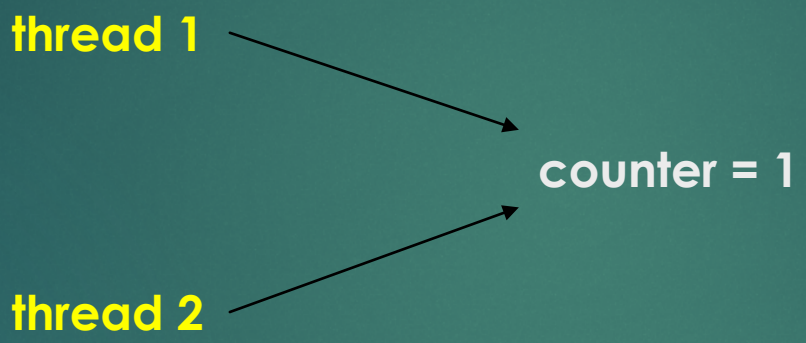
Volatile


counter = counter + 1
thread 1
counter = 0
thread 2
counter = counter + 1

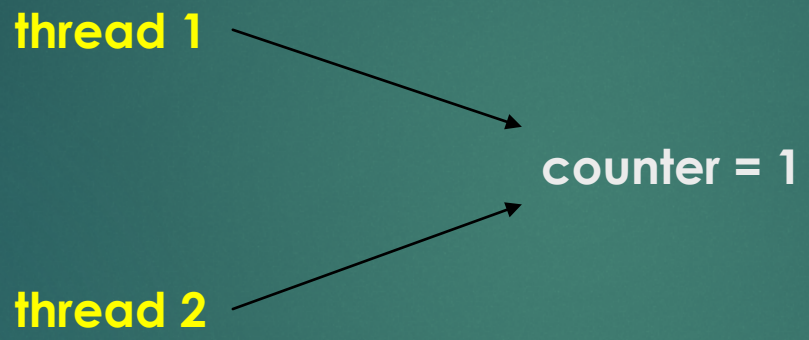
Volatile



Volatile

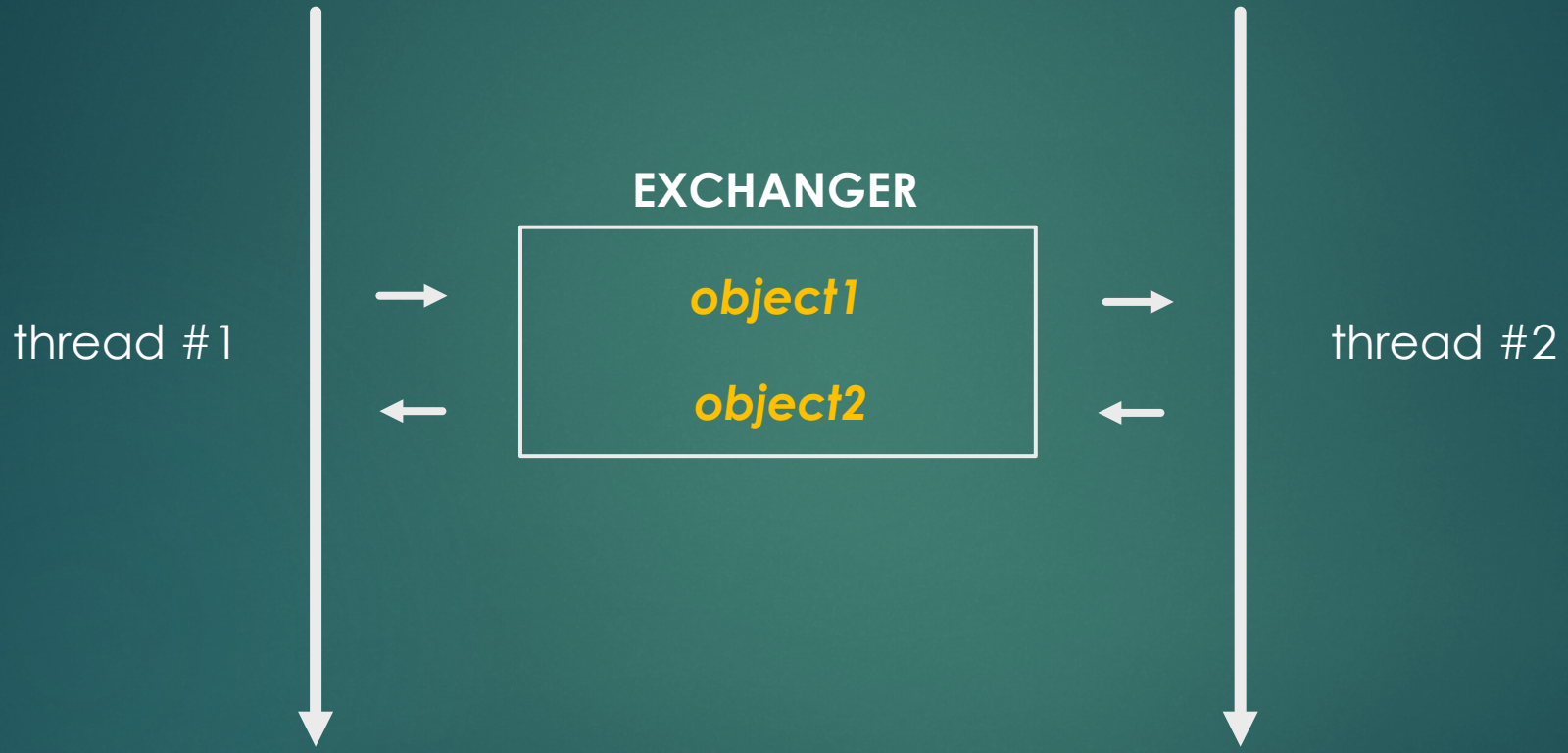


Volatile



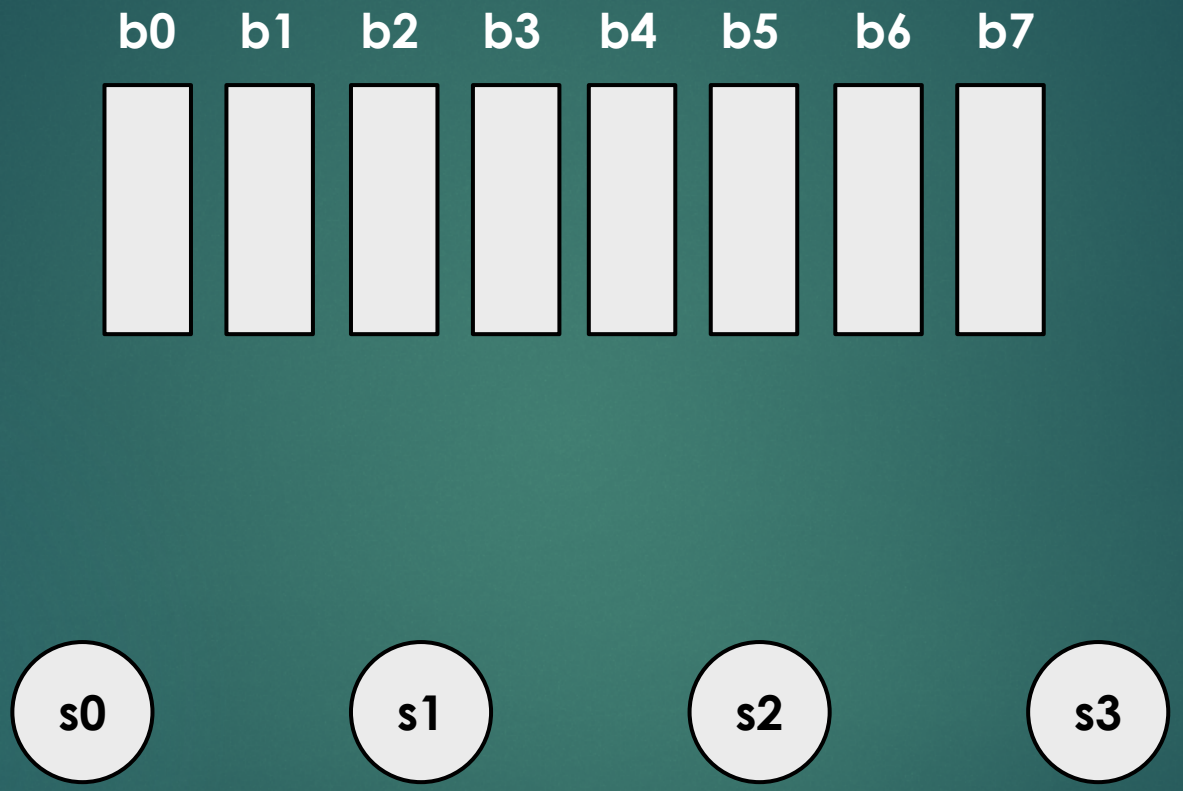
Counter remained 1 instead of 2

~ we should make sure the threads are going to wait for each other to finish the given task on the variables !!!


EXCHANGER
object1
object2
thread #1
thread #2

MULTITHREADING

STUDENT LIBRARY SIMULATION

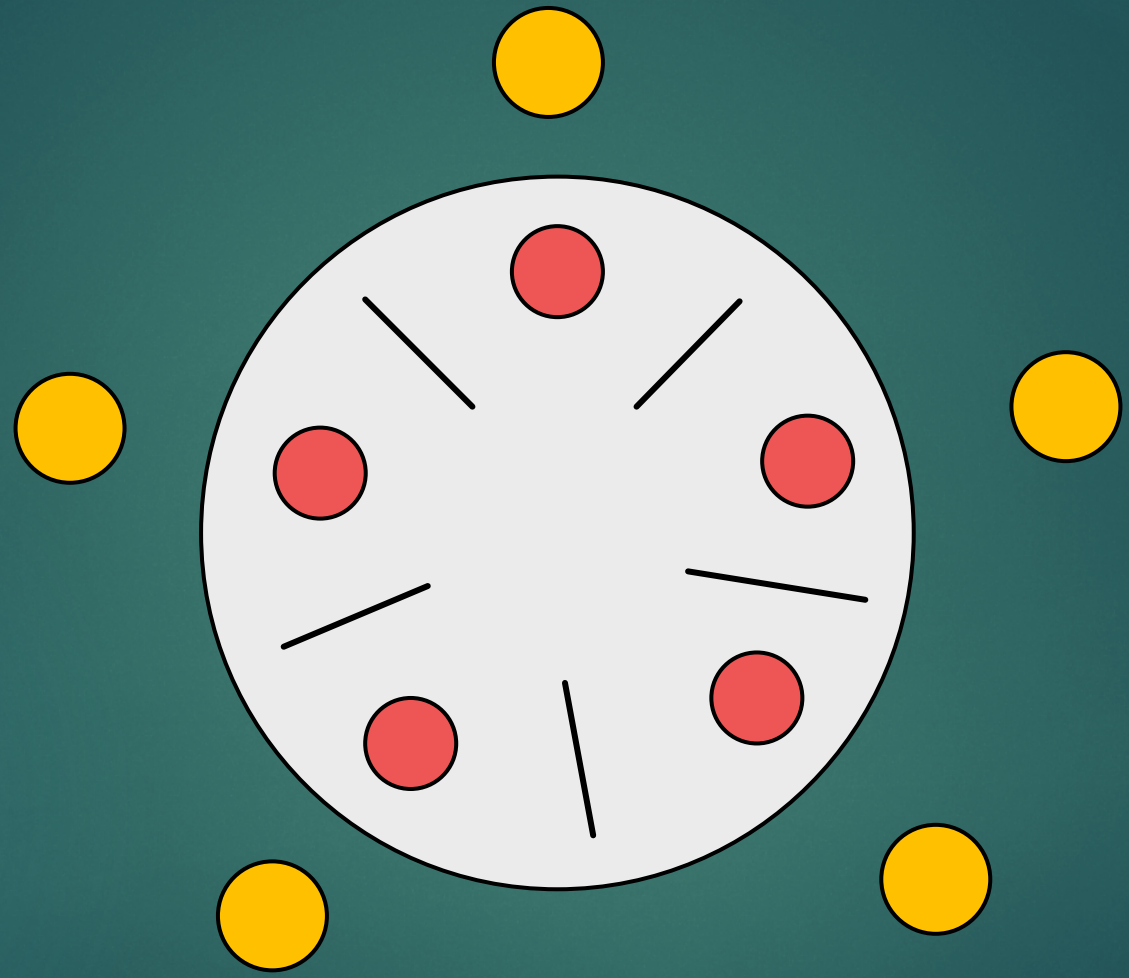

b0
b1
b2
b3
b4
b5
b6
b7
s0
s1
s2
s3

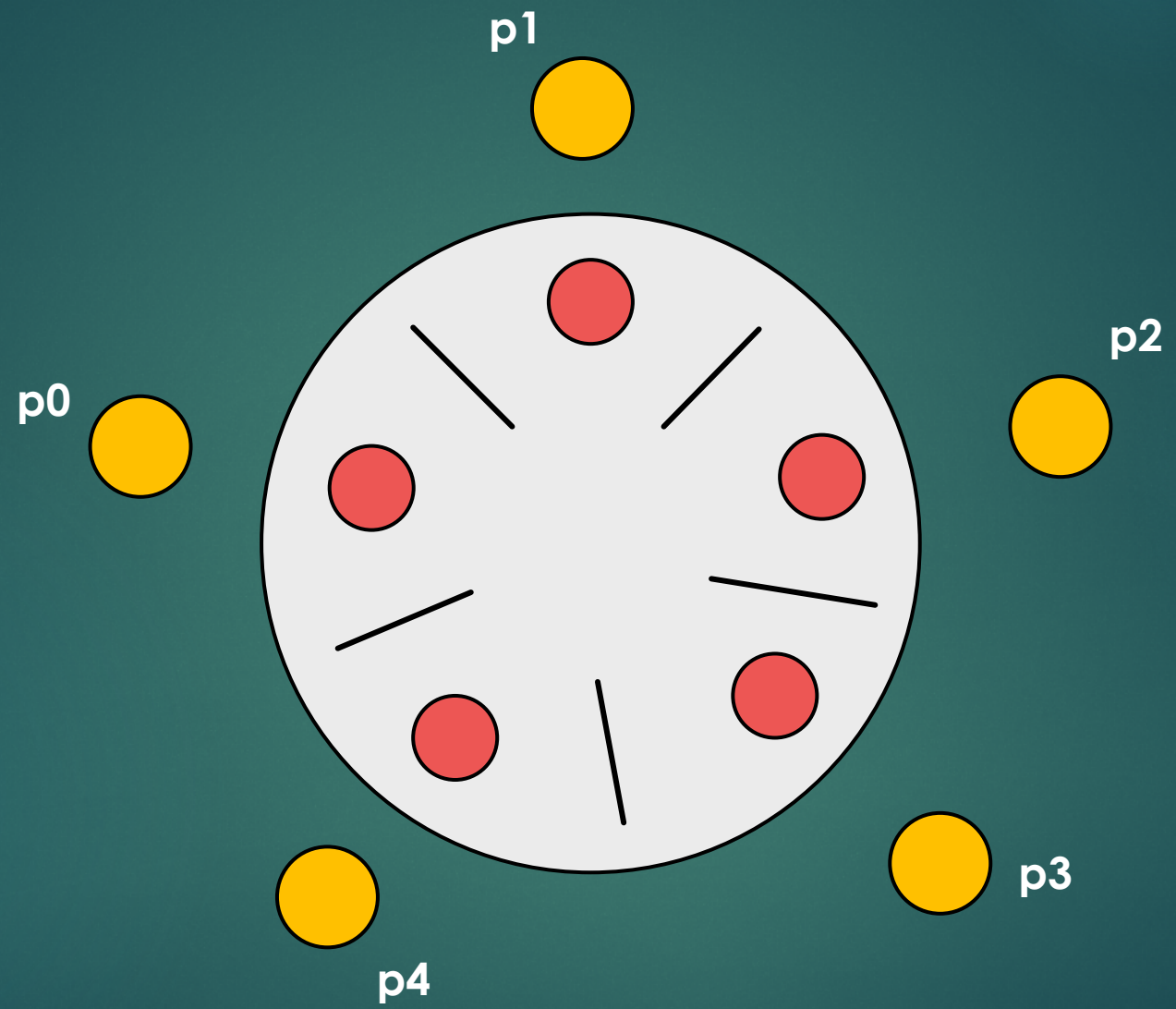
MULTITHREADING

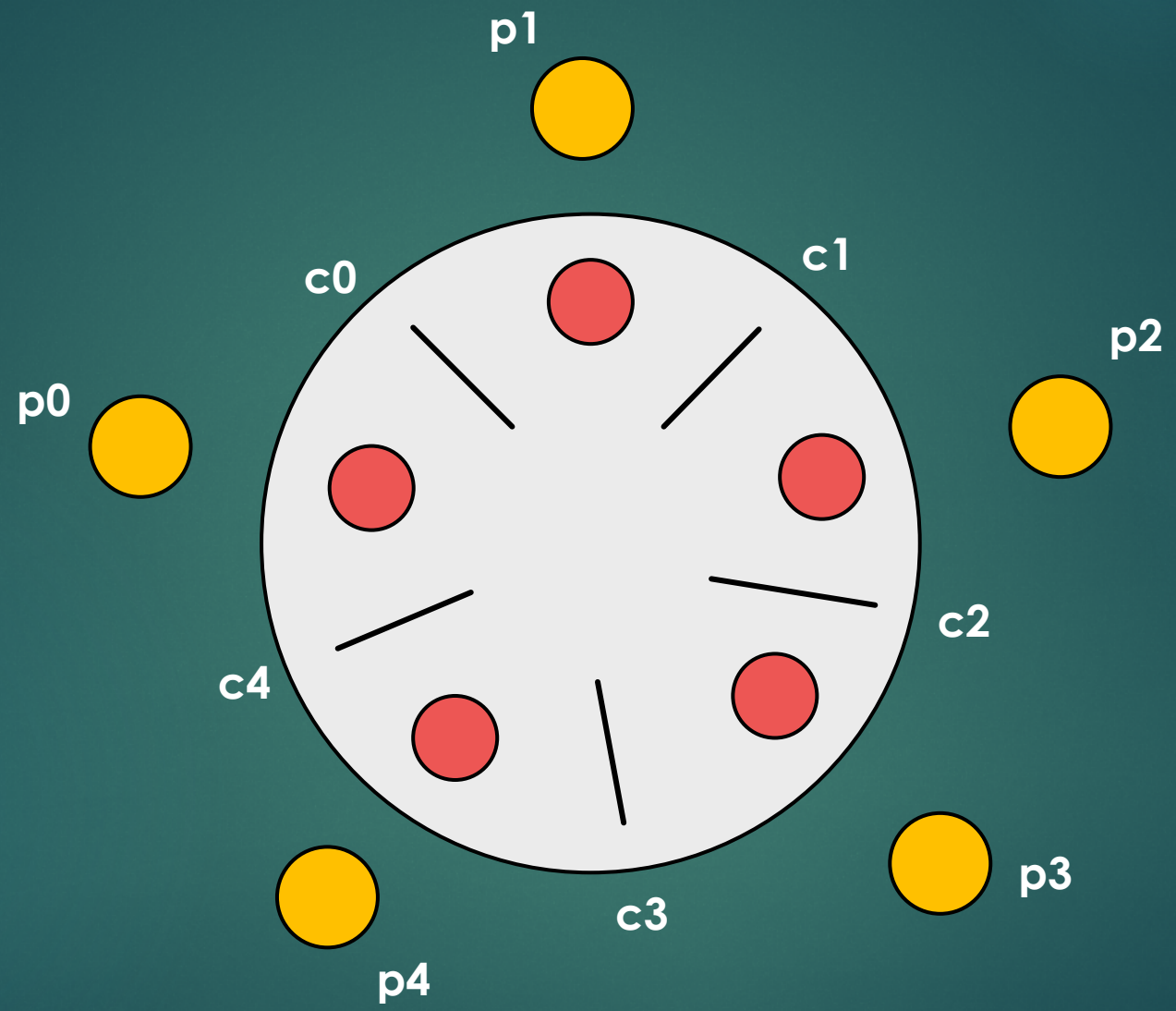
DINING PHILOSOPHERS

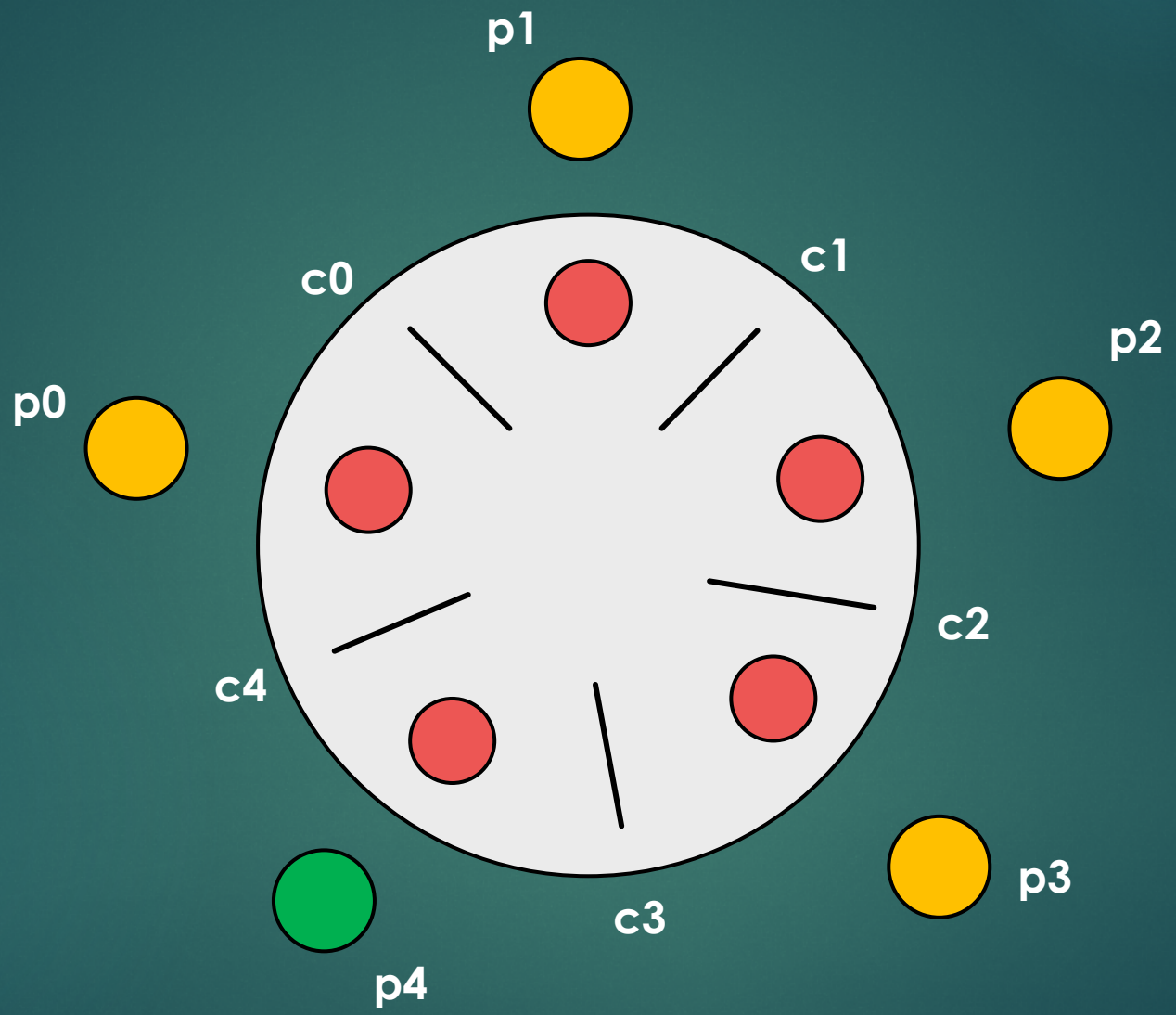
Dining philosopher

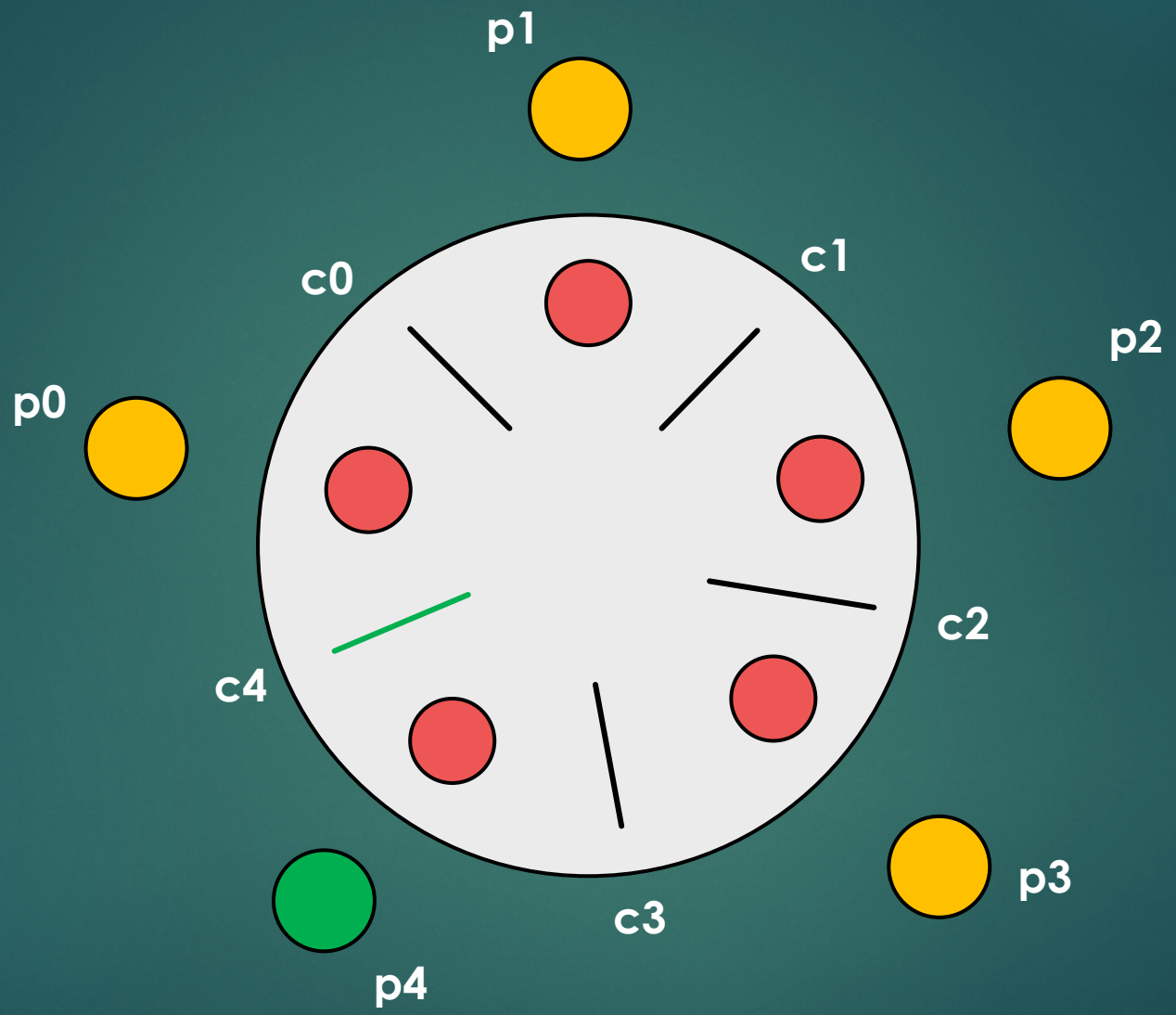
- It was formulated by Dijkstra in 1965
- 5 philopshers are present at a table + 5 forks / chopsticks
- They can eat or think
- Philosophers can eat when they have both left and right chopsticks
- A chopstick can be hold by one philosopher at a given time
- The problem: how to create a concurrent algorithm such that no philosopher will starve

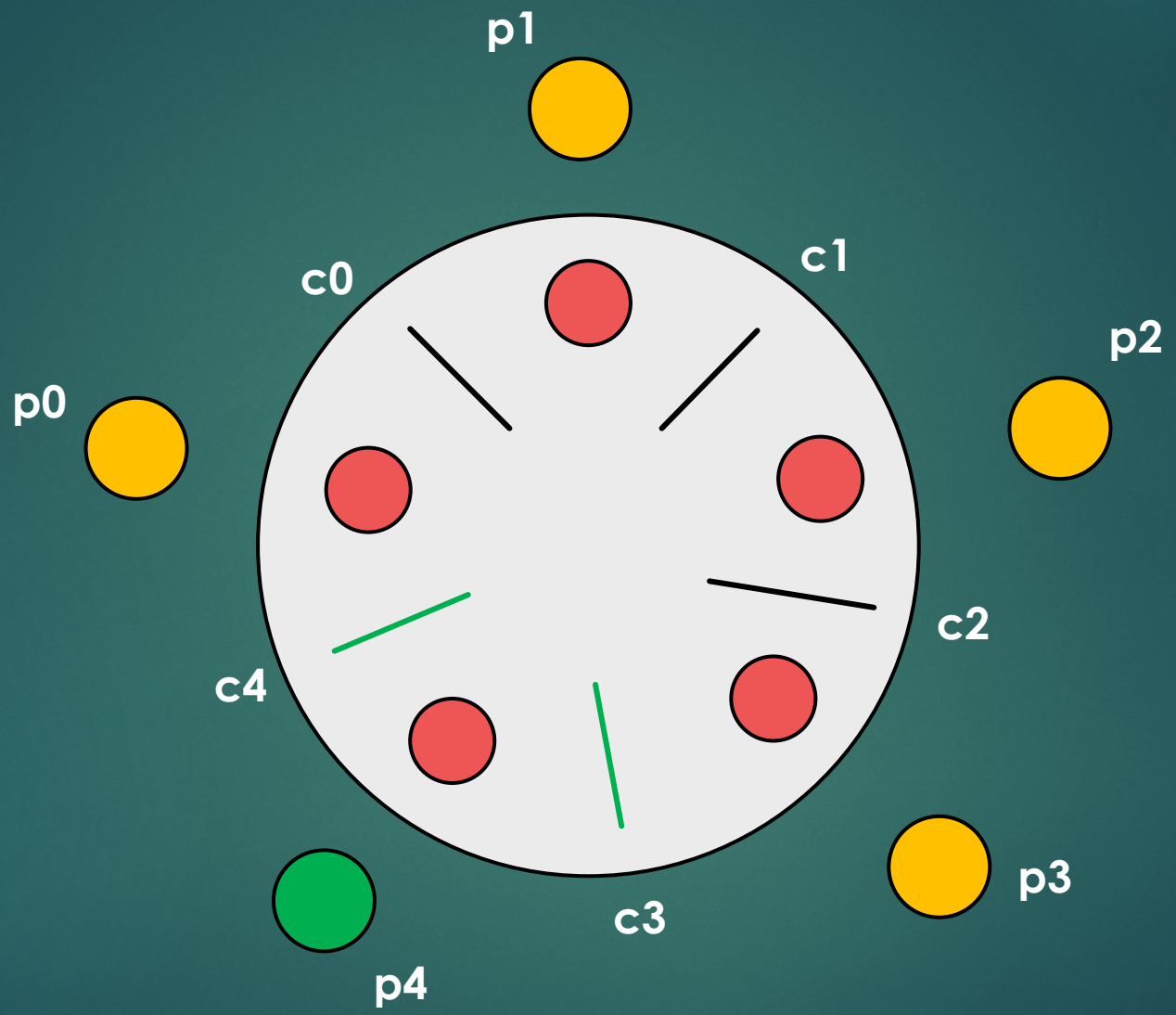



p1
p0
p2
p3
p4


p1
c0
c1
p2
p0
c2
c4
c3
p3
p4


p1
c0
c1
p2
p0
c2
c4
c3
p3
p4


p1
c0
c1
p2
p0
c2
c4
c3
p3
p4


p1
c0
c1
p2
p0
c2
c4
c3
p3
p4

MULTITHREADING

LOCKS AND SYNCHRONIZATION

A reentrant lock has the same basic behavior as we have seen for synchronized blocks

~ of course there are some extended features !!!

→ We can make a lock fair: prevent thread starvation
Synchronized blocks are unfair by default

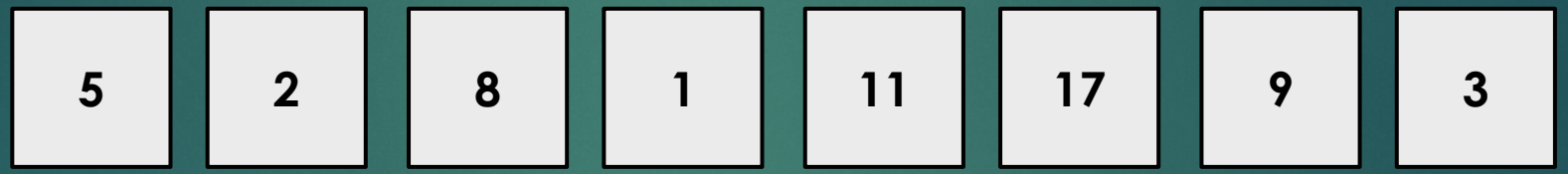
→ We can check whether the given lock is held or not with reentrant locks

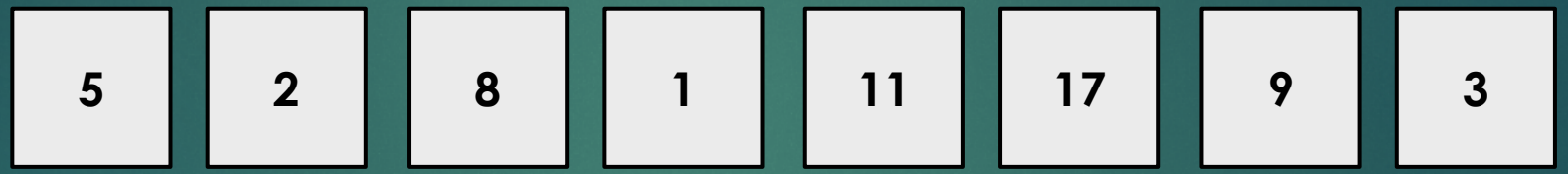
→ We can get the list of threads waiting for the given lock with reentrant locks

→ Synchronized blocks are nicer: we do not need the try-catch-finally block

MULTITHREADING

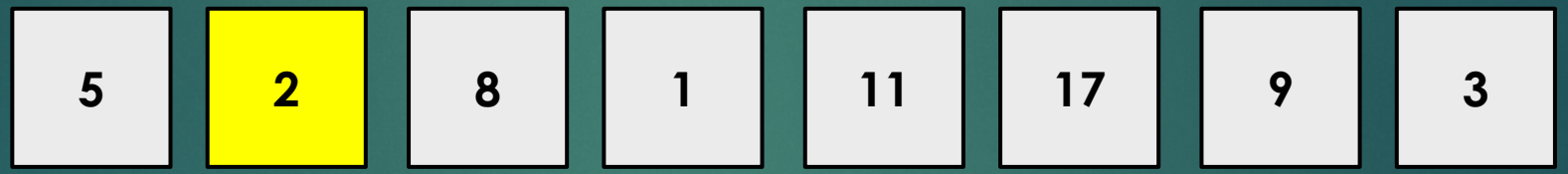
PARALLEL SUM


5
2
8
1
11
17
9
3

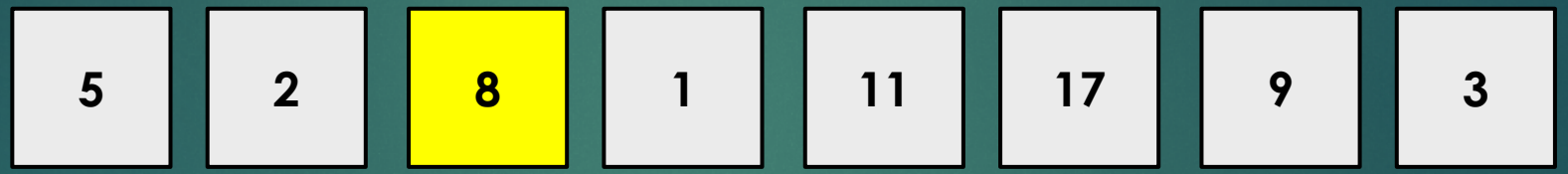



```
for i in nums
        total = total + nums[i]
```

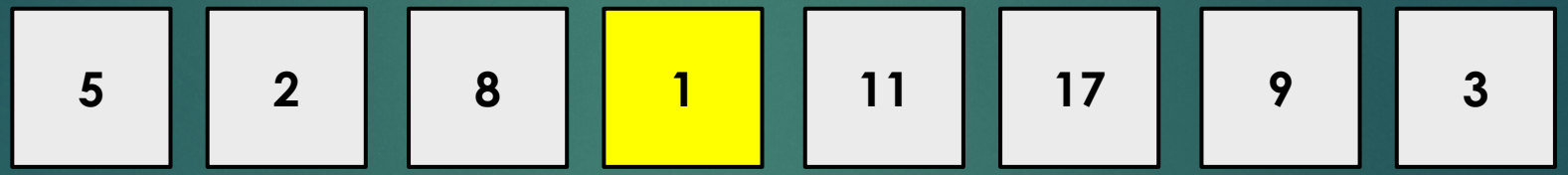

```
for i in nums
        total = total + nums[i]
```




```
for i in nums
        total = total + nums[i]
```

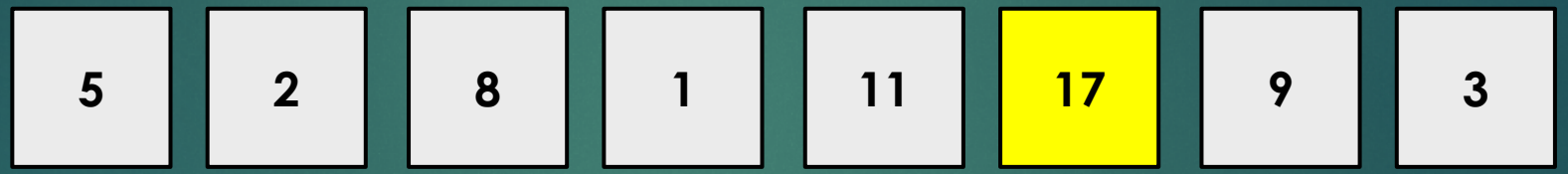



```
for i in nums
        total = total + nums[i]
```

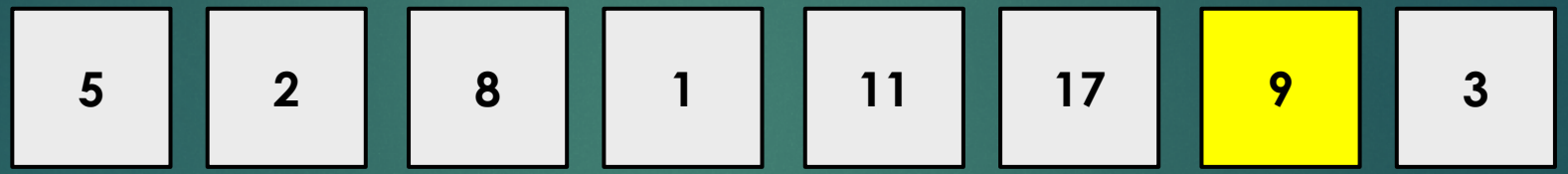



```
for i in nums
        total = total + nums[i]
```


```
for i in nums
        total = total + nums[i]
```



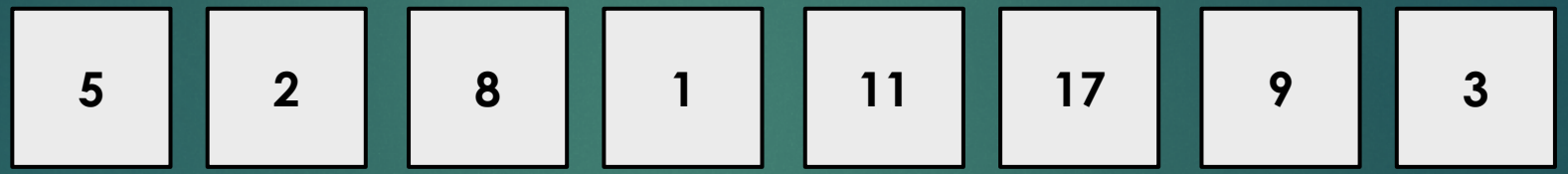

```
for i in nums
        total = total + nums[i]
```



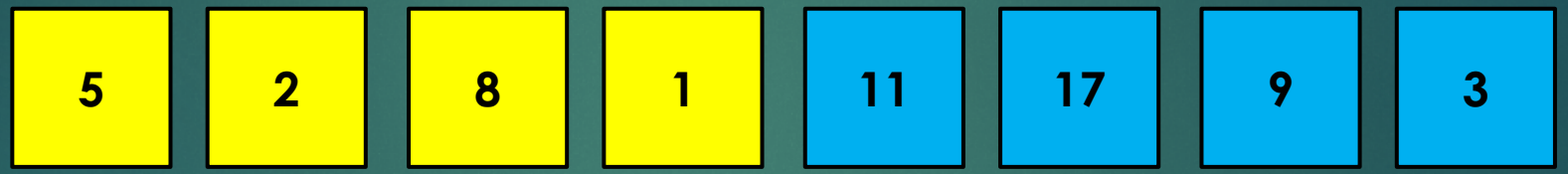

```
for i in nums
        total = total + nums[i]
```


```
for i in nums
        total = total + nums[i]
```

Parallel sum → with multiple processors or multicore processor
we can assign a task to every processor
~ parallel computing



Parallel sum → with multiple processors or multicore processor
we can assign a task to every processor
~ parallel computing



thread #1

sum1 = 16

thread #2

sum2 = 40

MULTITHREADING

FORK-JOIN FRAMEWORK

What is fork-join framework?

Concrete implementation for parallel execution !!!

Fork-join framework

- This framework helps to make an algorithm parallel
- We do not have to bother about low level synchronizations or locks
- Divide and conquer algorithms !!!
- A larger task → it can be divided into smaller ones + the subsolutions can be combined !!!
- **IMPORTANT** subtasks have to be independent in order to be executed in parallel
- So the main concept→ fork-join framework breaks the task into smalles subtasks until these subtasks are simple enough to solve without further breakups
- For example: parallel merge sort, parallel maximum finding

RecursiveTask<T> it will return a T type

All the tasks we want to execute in parallel is a subclass of this class
We have to override the computer method that will return the solution of the subproblem

RecursiveAction it is a task, but without any return value

ForkJoinPool

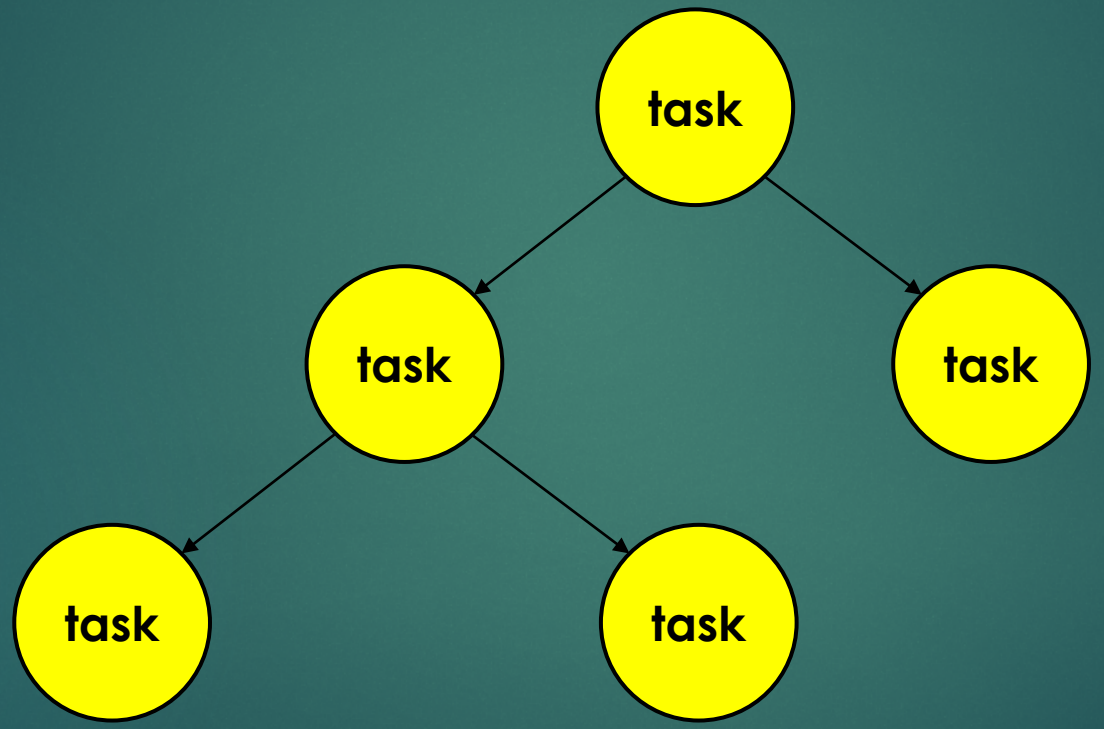
Basically it is a thread pool for executing fork-join tasks

work-stealing a task is not equivalent to a thread !!!
Tasks are lightweight threads → so fork-join will be efficient even when there are a huge number of tasks

So ForkJoinPool creates a fix number of threads → usually the number of CPU cores

These threads are executing the tasks but if a thread has no task: it can „steal" a task from more busy threads
~ tasks are distributed to all threads in the thread pool !!!

fork → split the given task into smaller subtasks that can be executed in a parallel manner



join → the splitted tasks are being executed and after all of them are finished they are merged into one result