Concepts of Concurrent Computation
Spring 2014
Lecture 4: Semaphores

Bertrand Meyer
Sebastian Nanz
Chris Poskitt

Chair of Software Engineering
ETH zürich

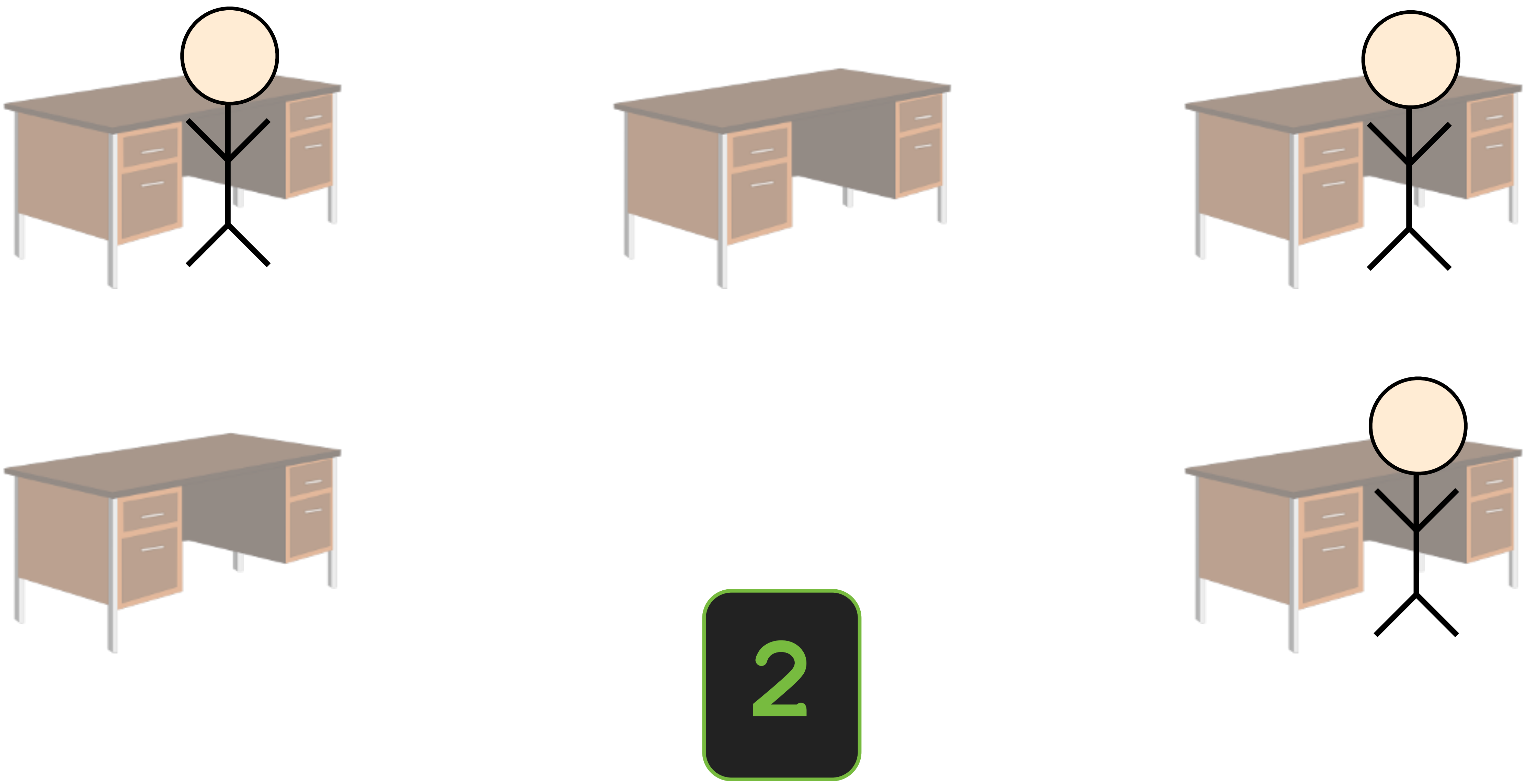
Last week: synchronisation, but lacking the simplicity

- we looked at various solutions to the mutual exclusion problem

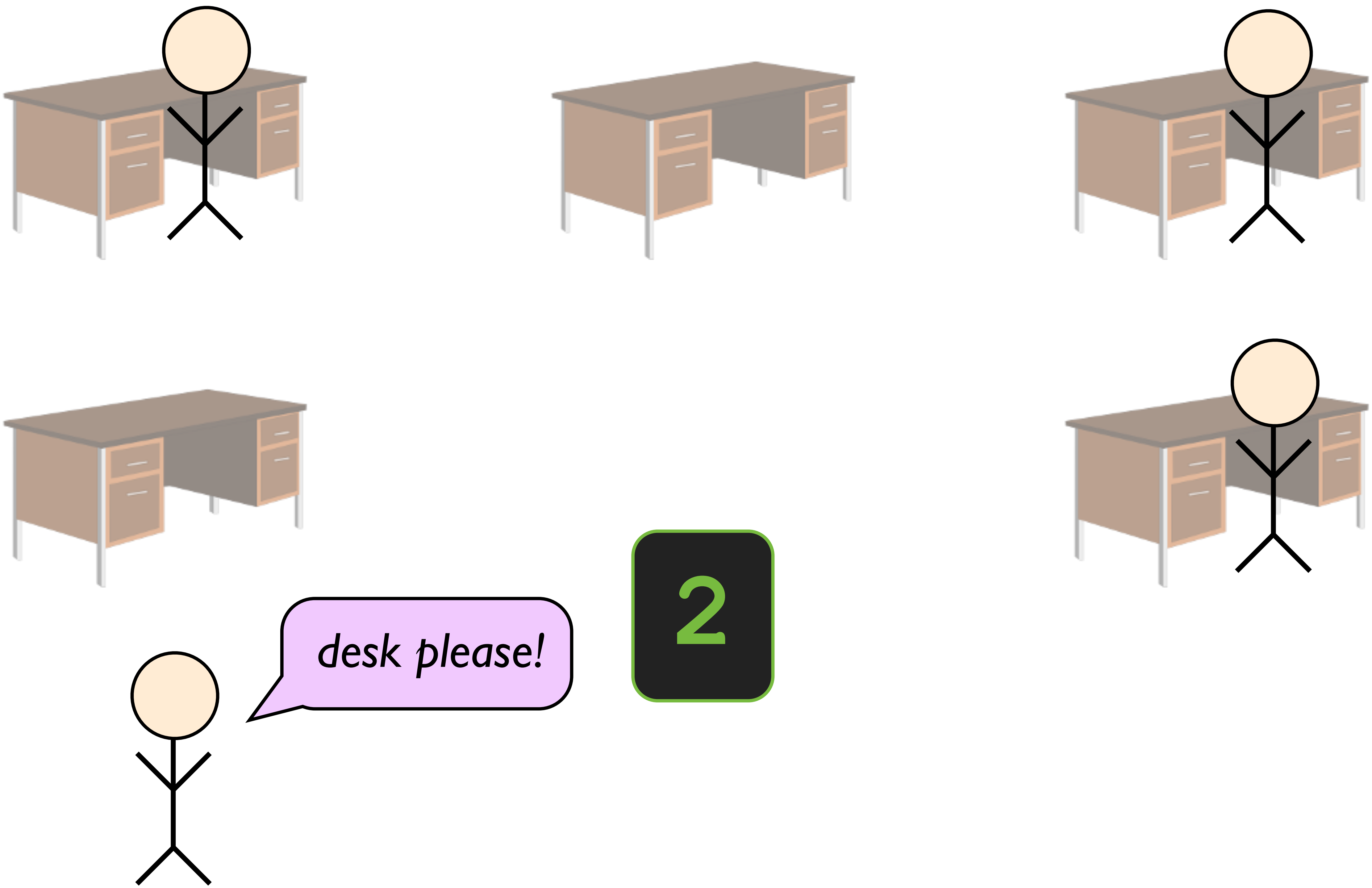
- algorithms were limited to the simplest tools - atomic read and write to shared memory

  => difficult to implement; complex
  => reliance on busy waiting
  => no encapsulation of synchronisation variables

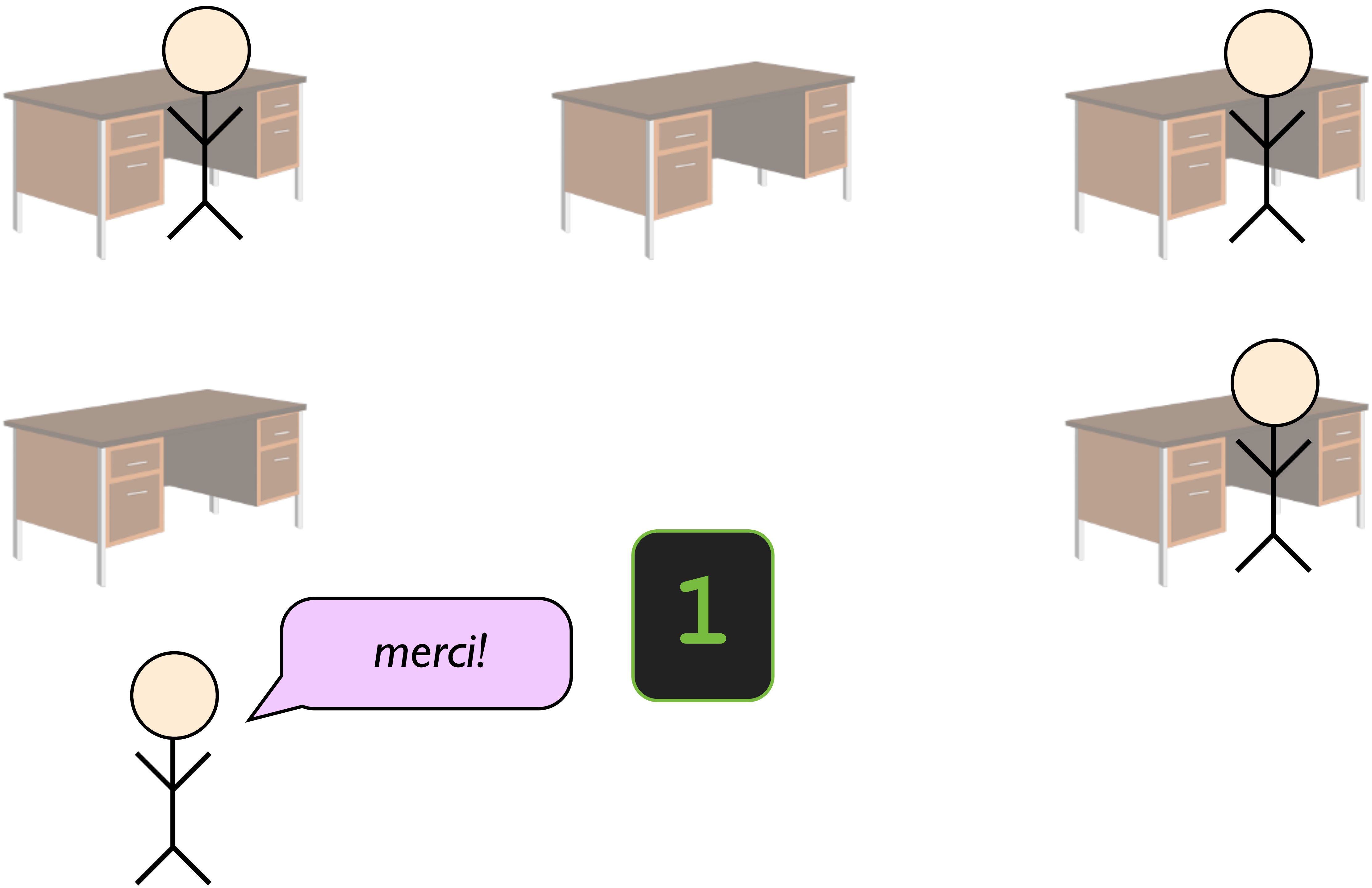
Short diversion: hot desks



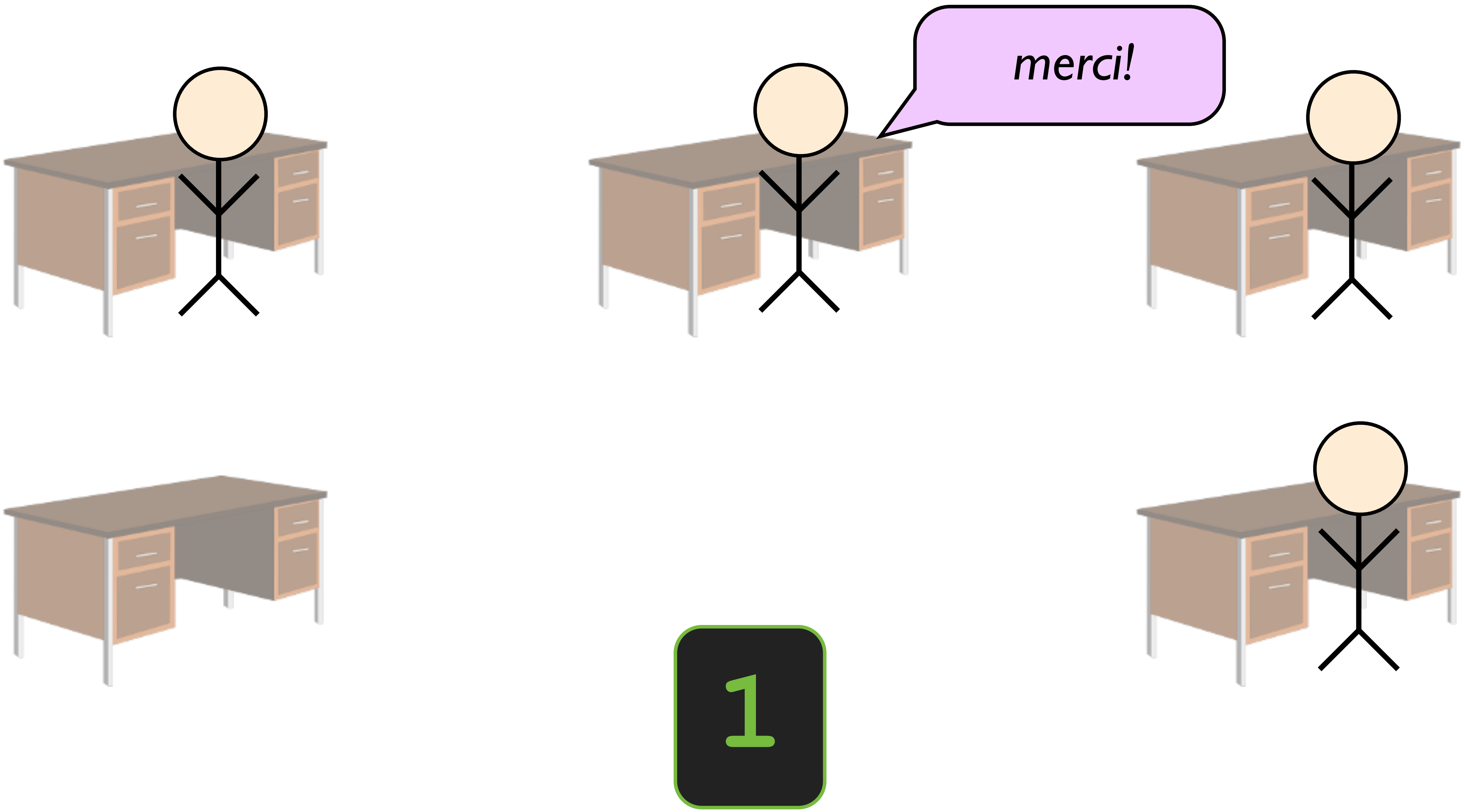
Short diversion: hot desks



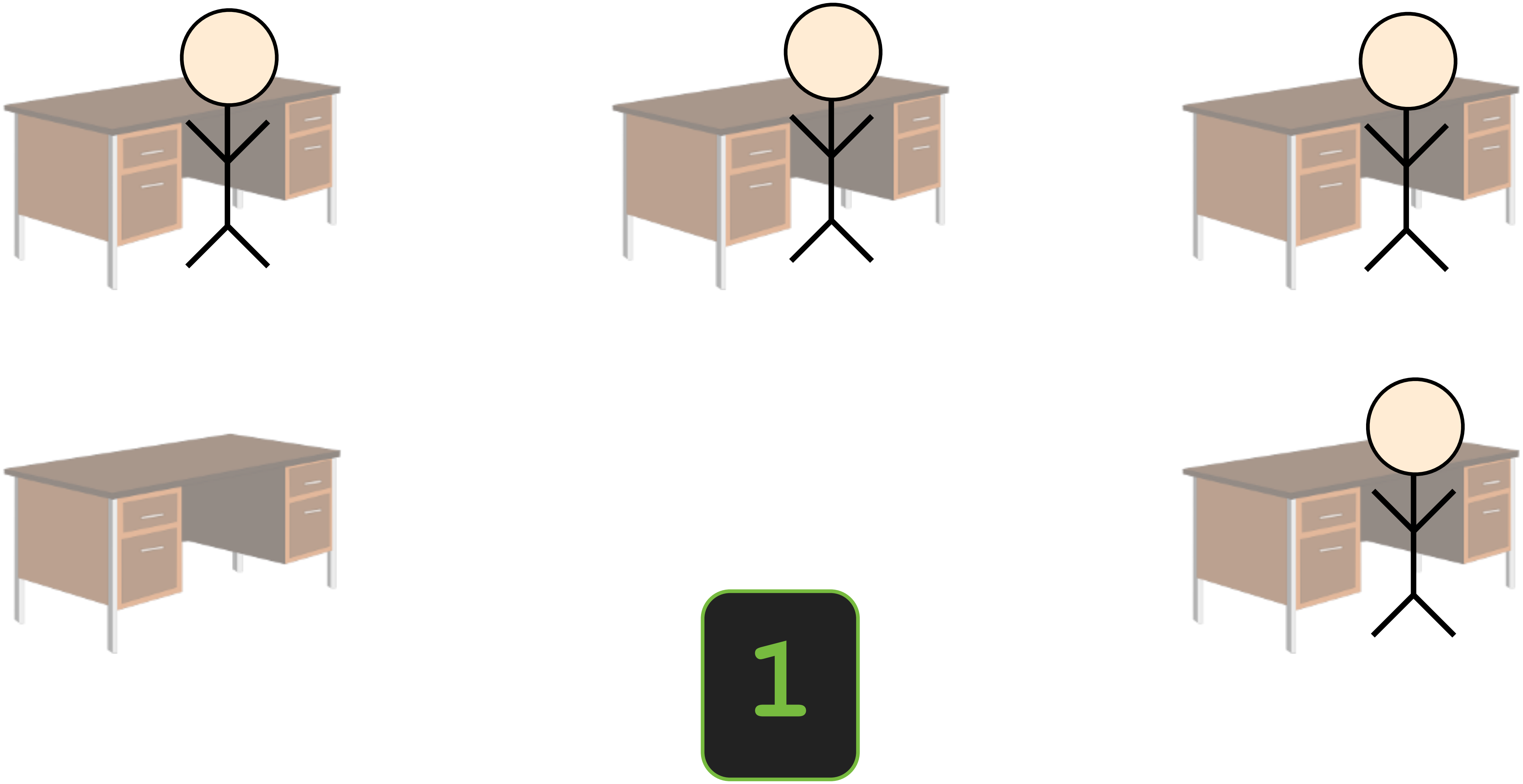
Short diversion: hot desks



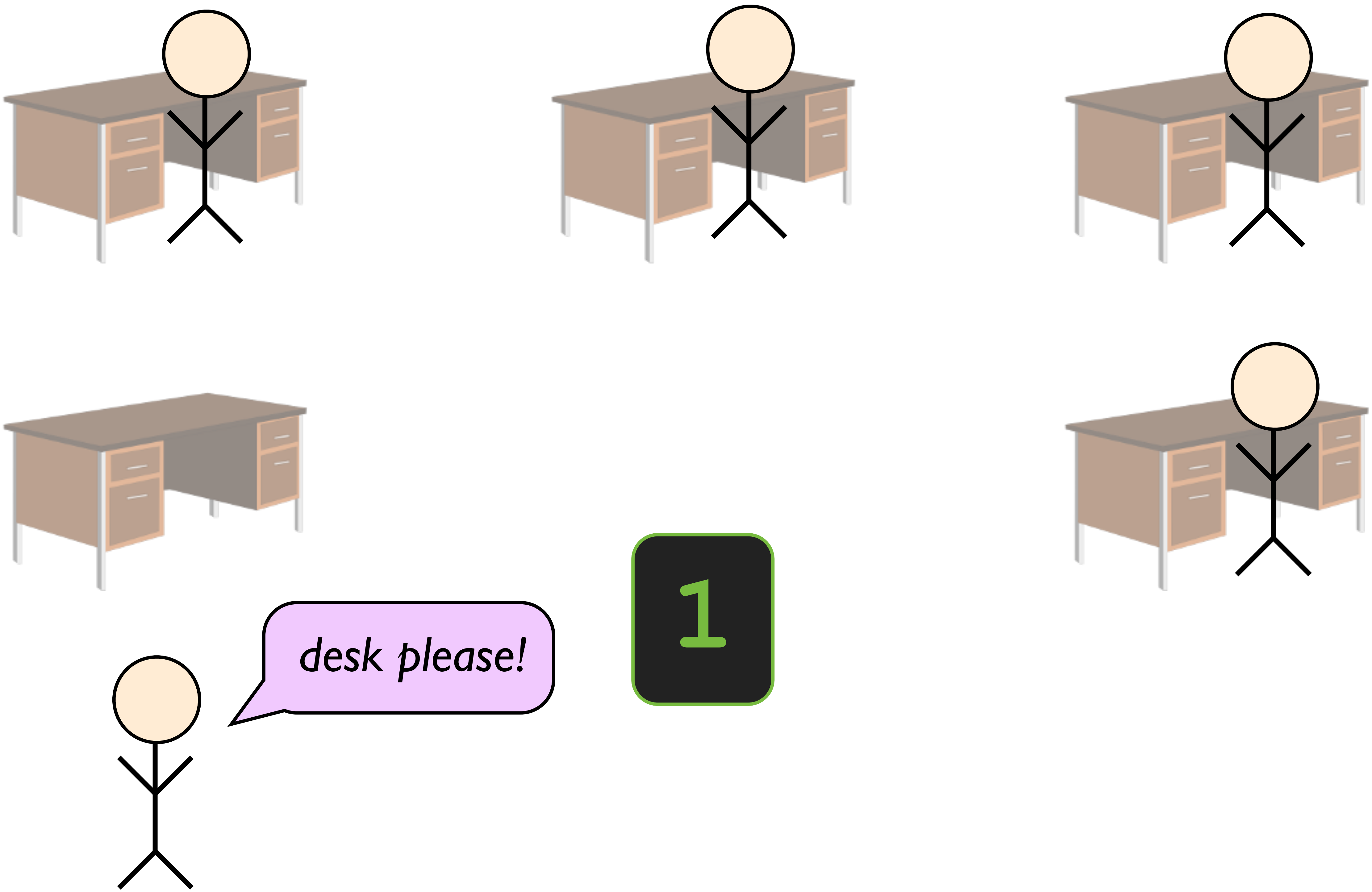
Short diversion: hot desks



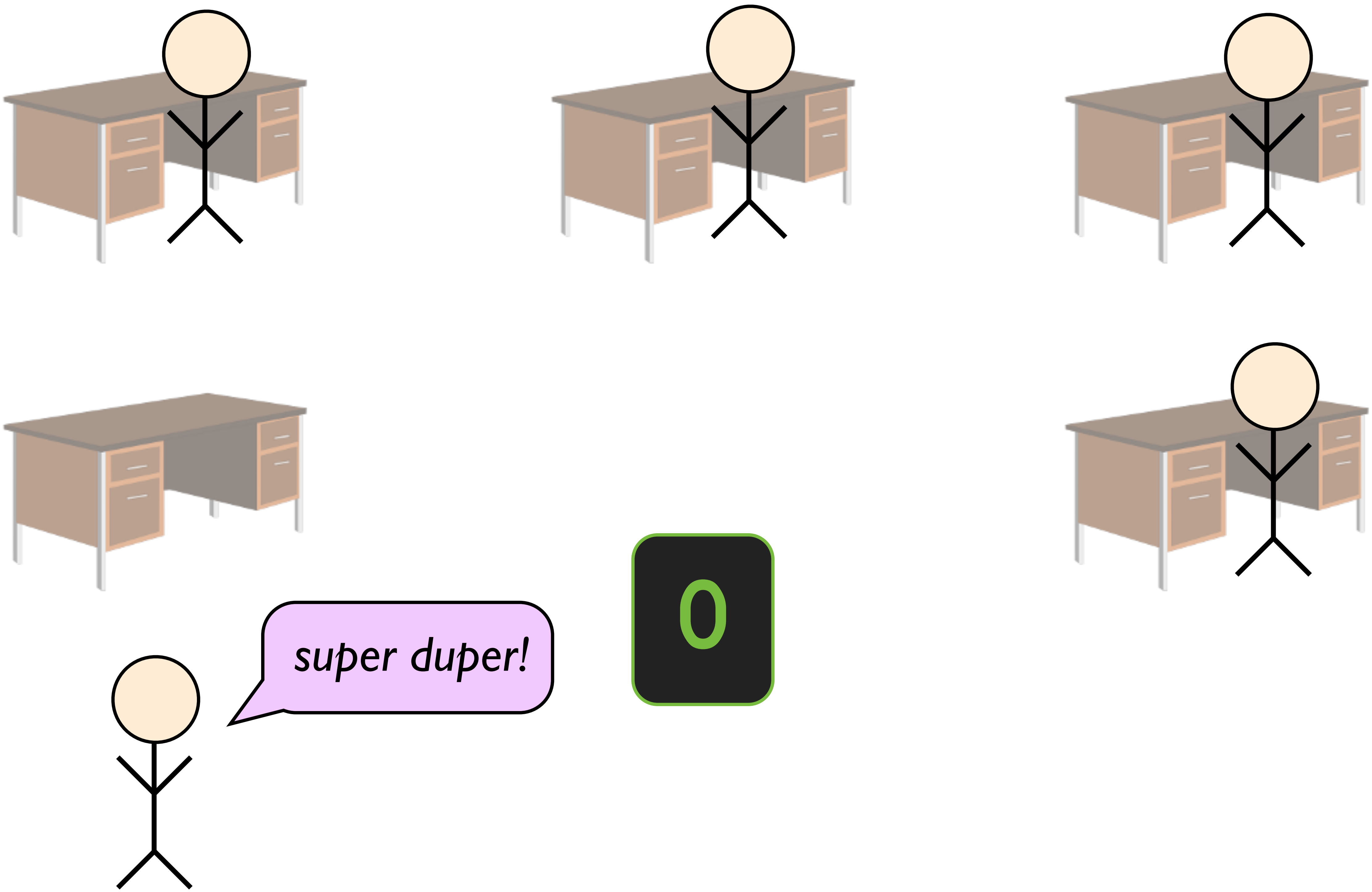
Short diversion: hot desks



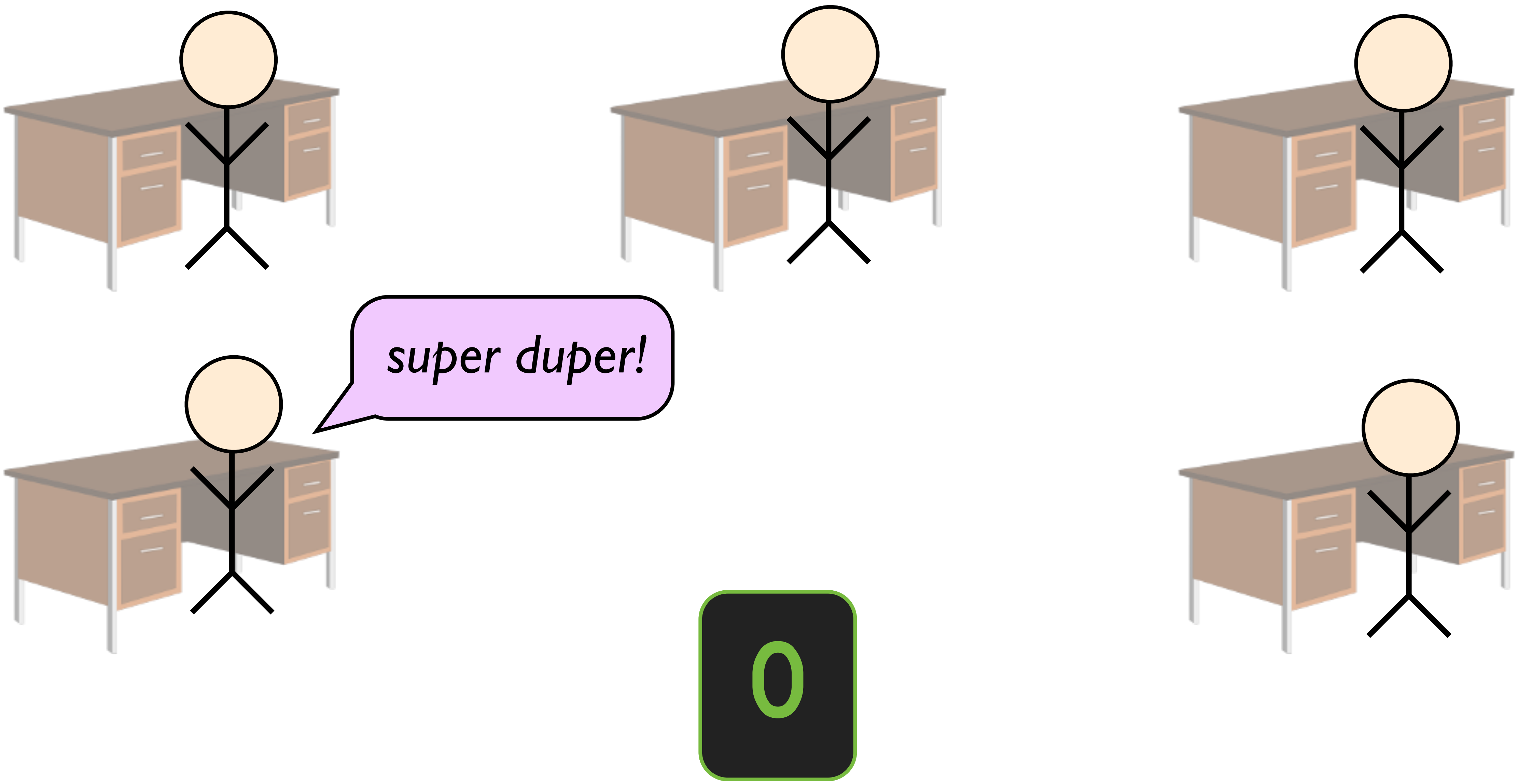
Short diversion: hot desks



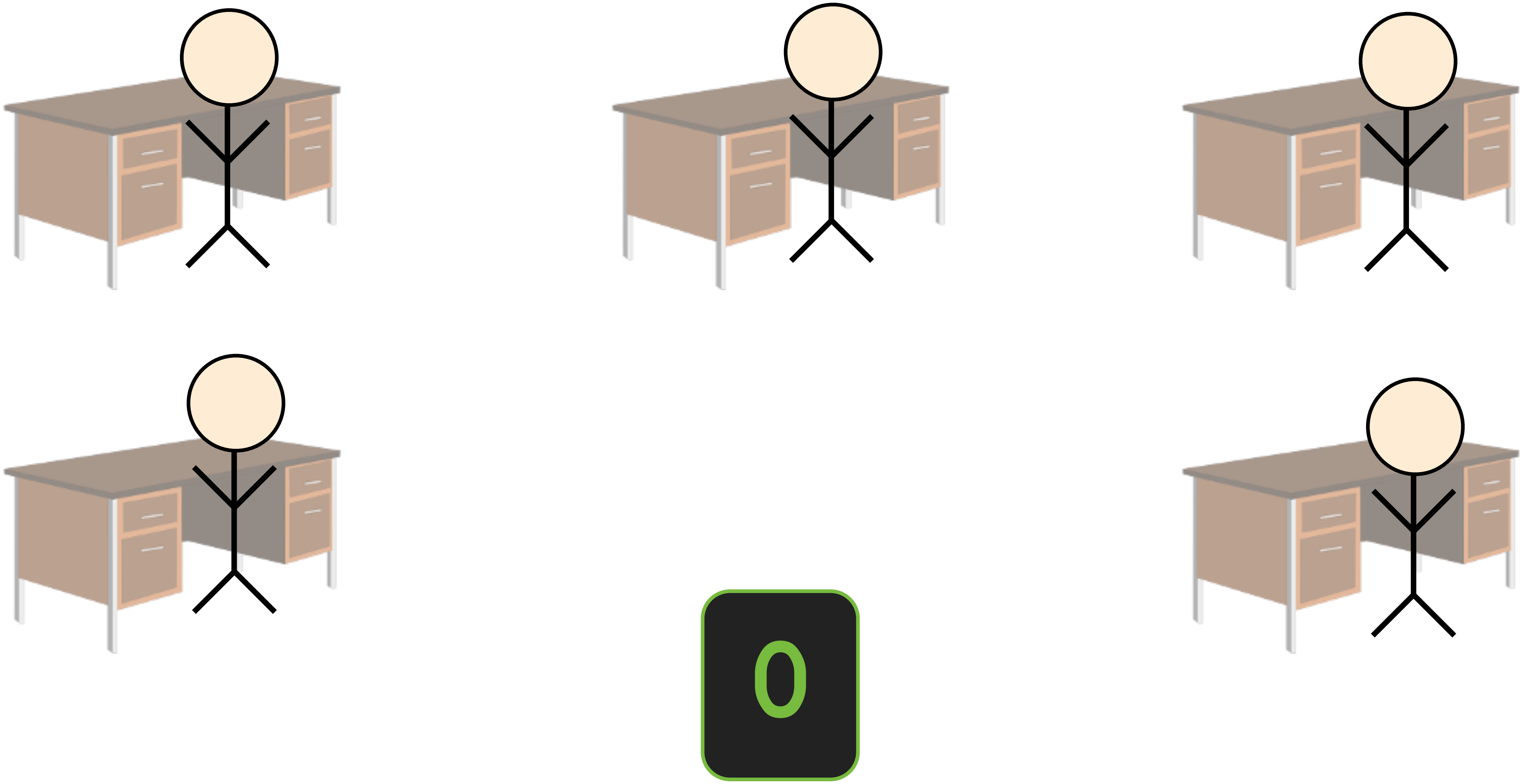
Short diversion: hot desks



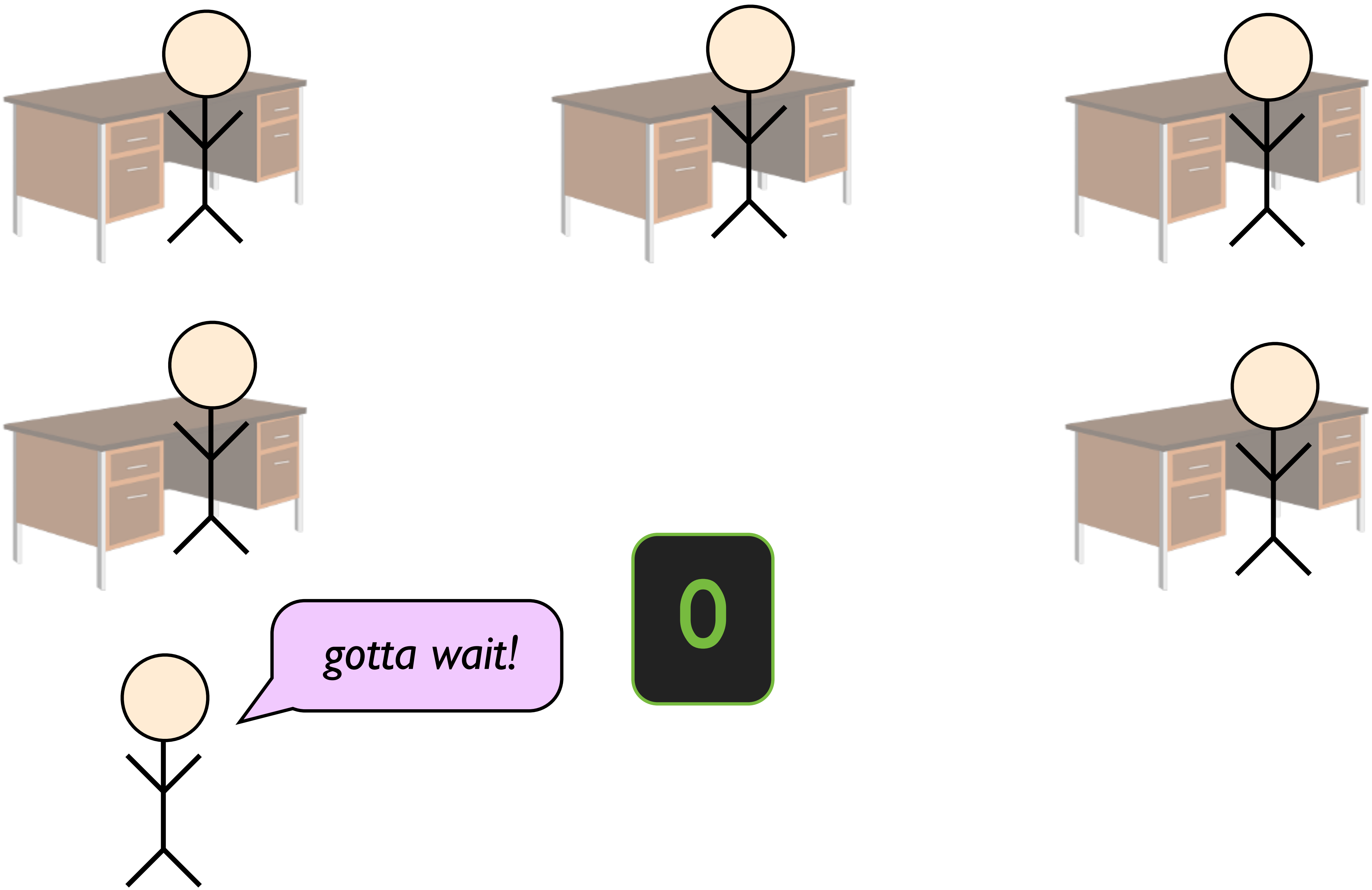
Short diversion: hot desks



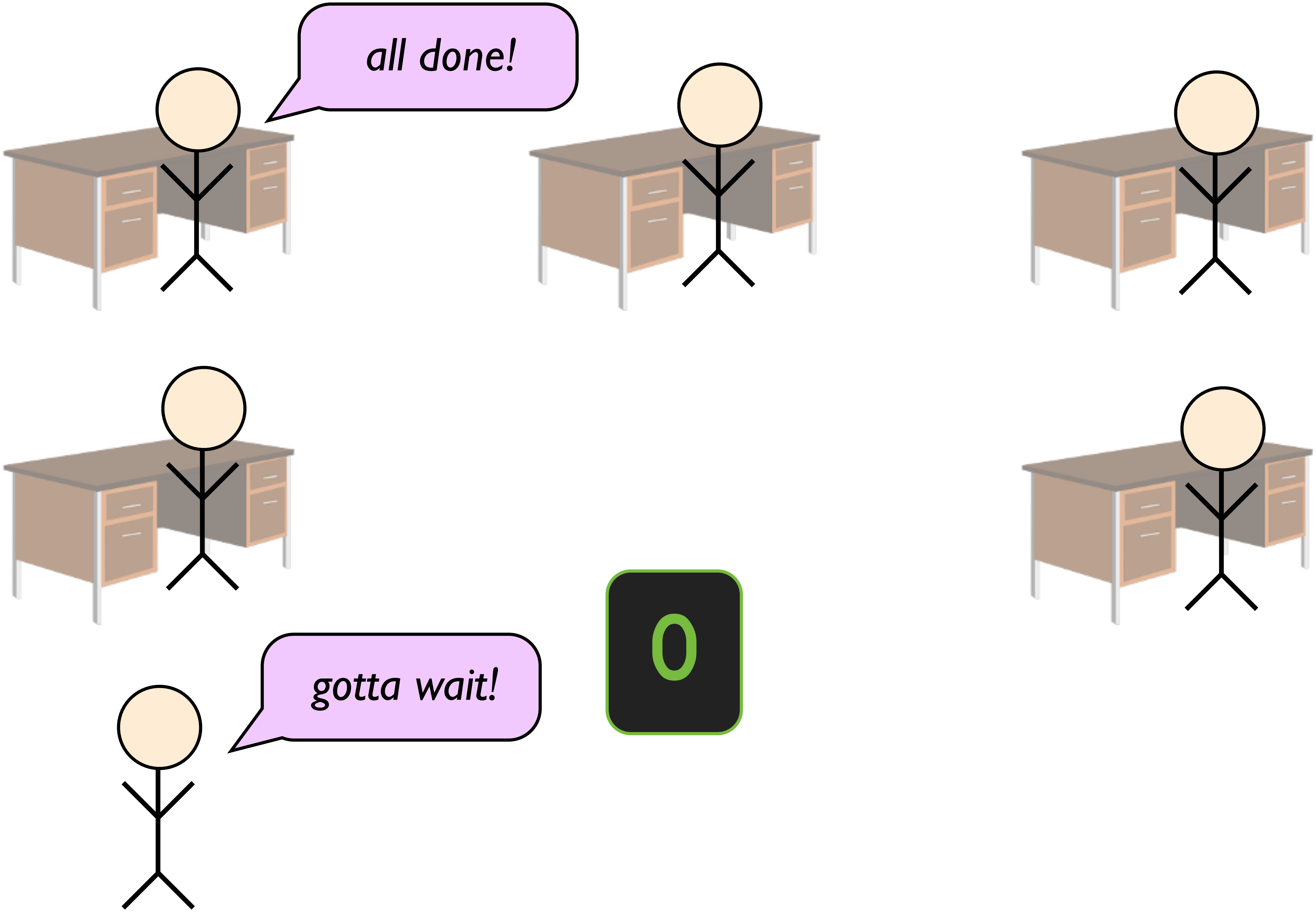
Short diversion: hot desks



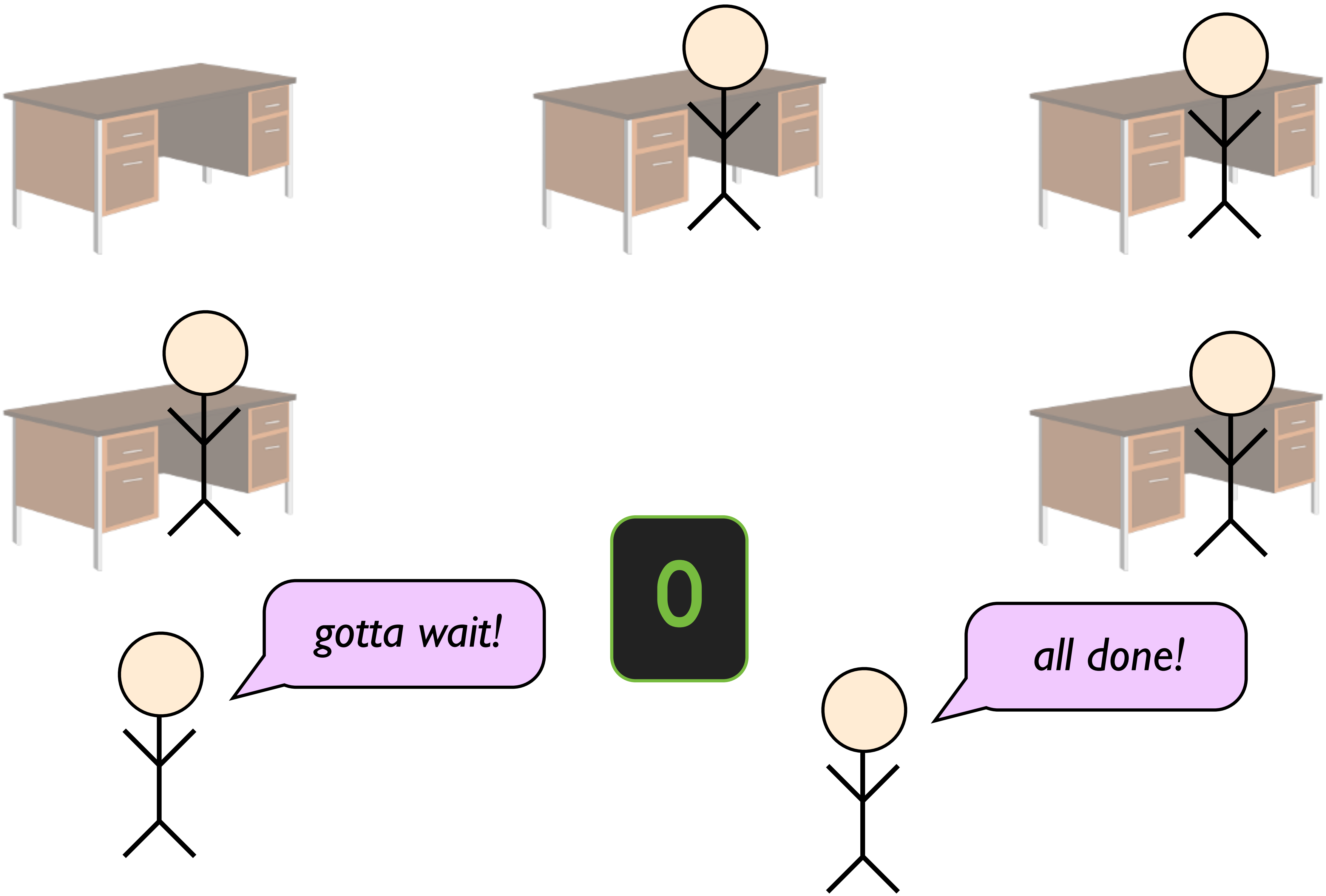
Short diversion: hot desks



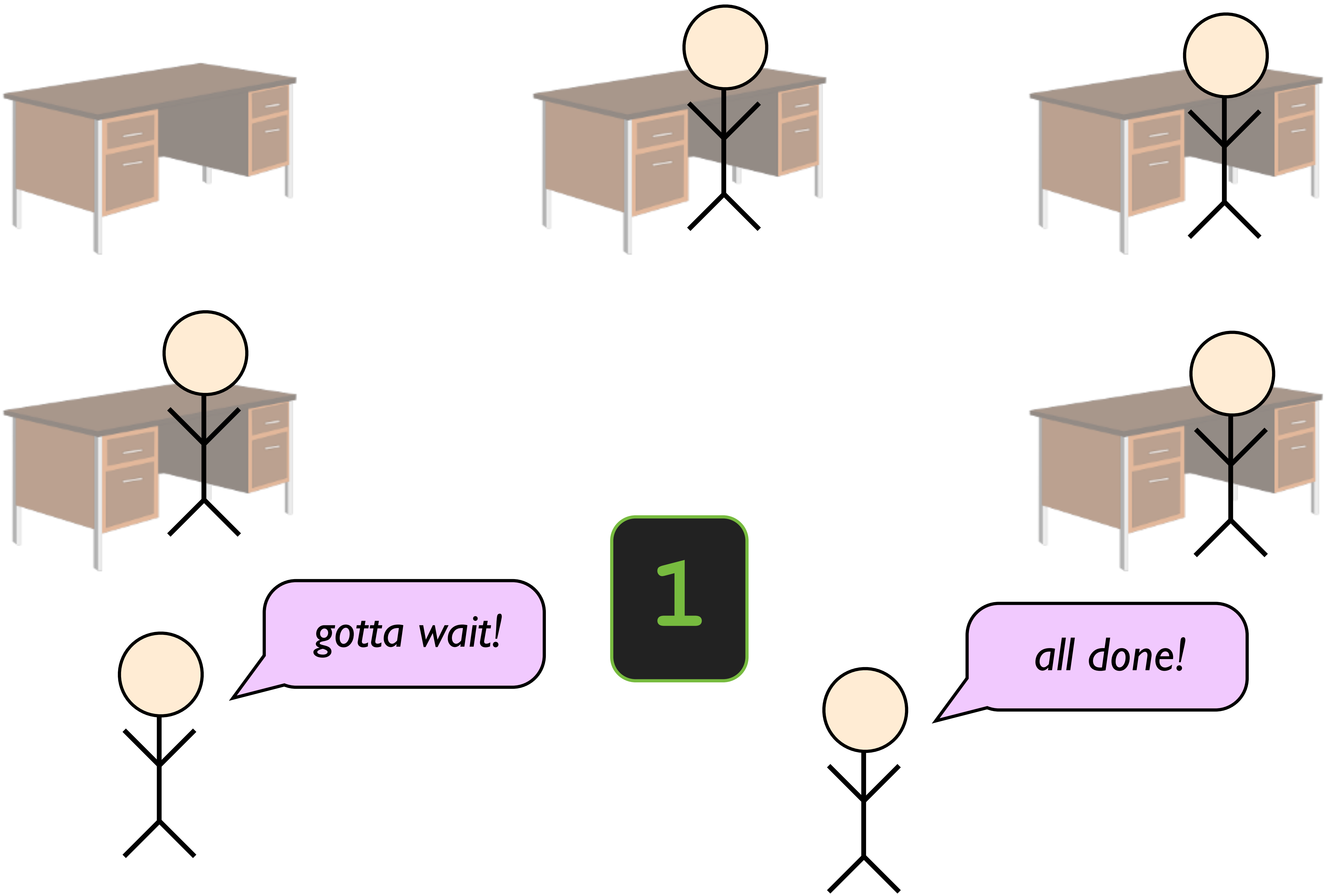
Short diversion: hot desks



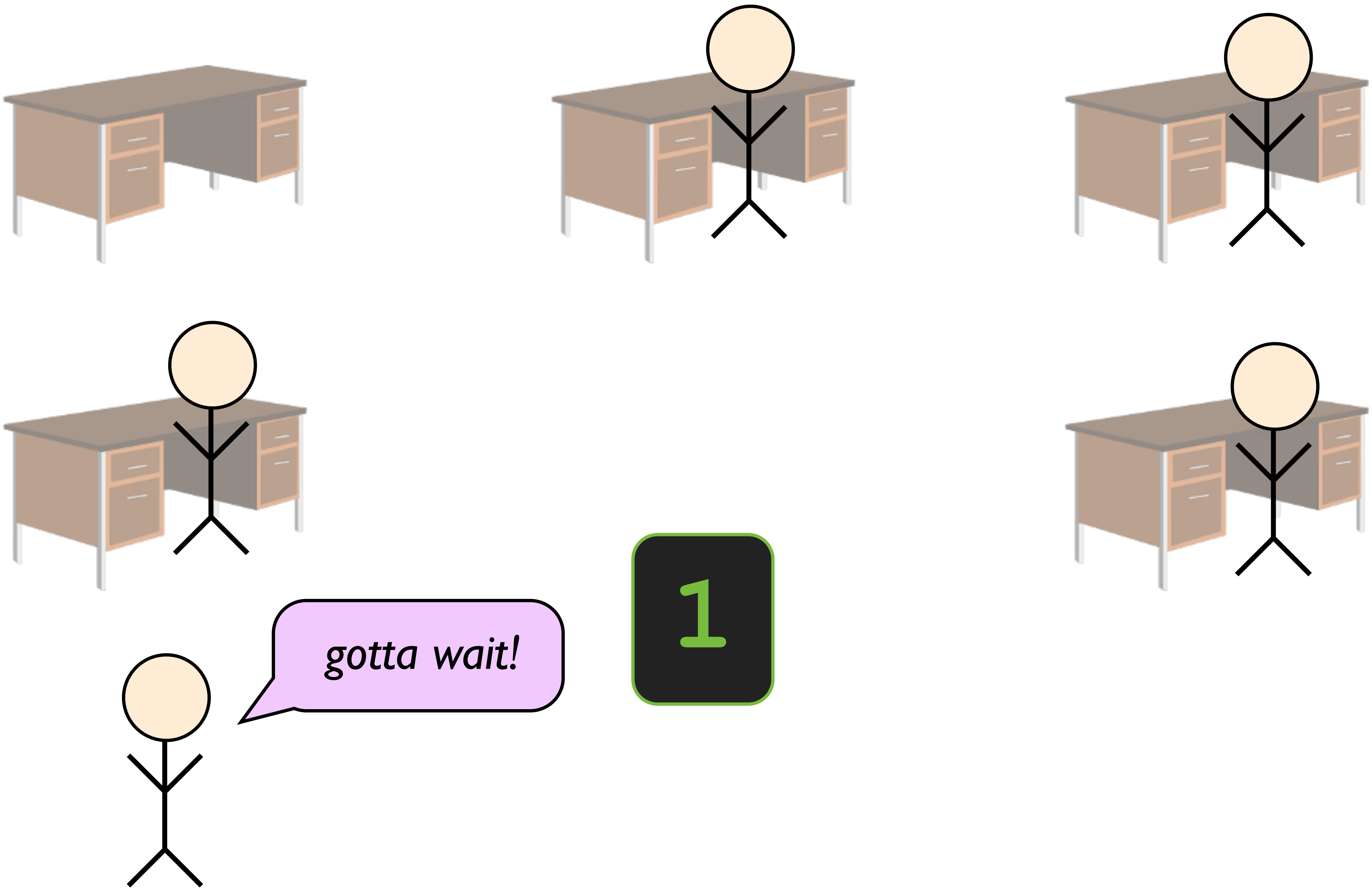
Short diversion: hot desks



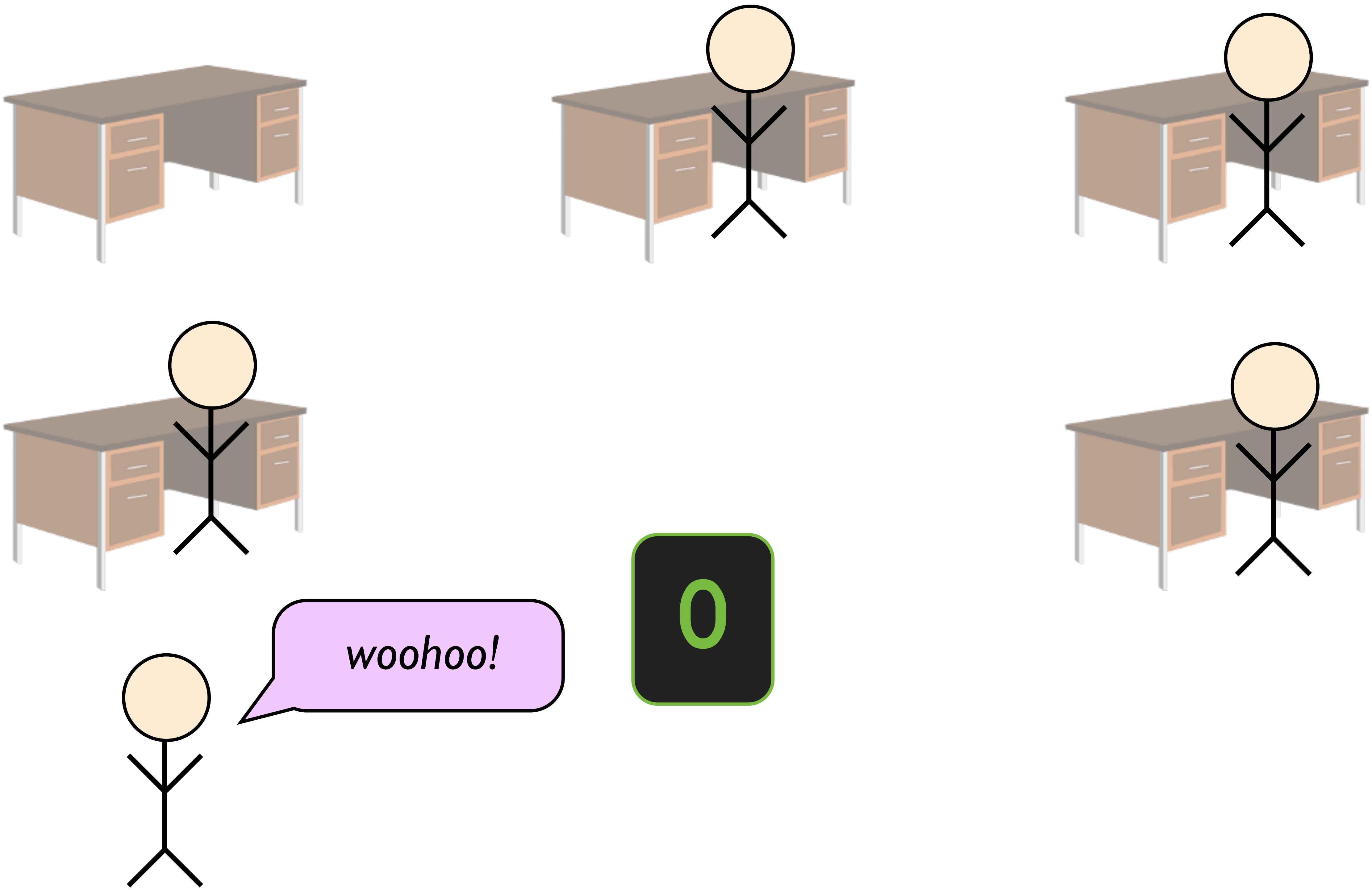
Short diversion: hot desks



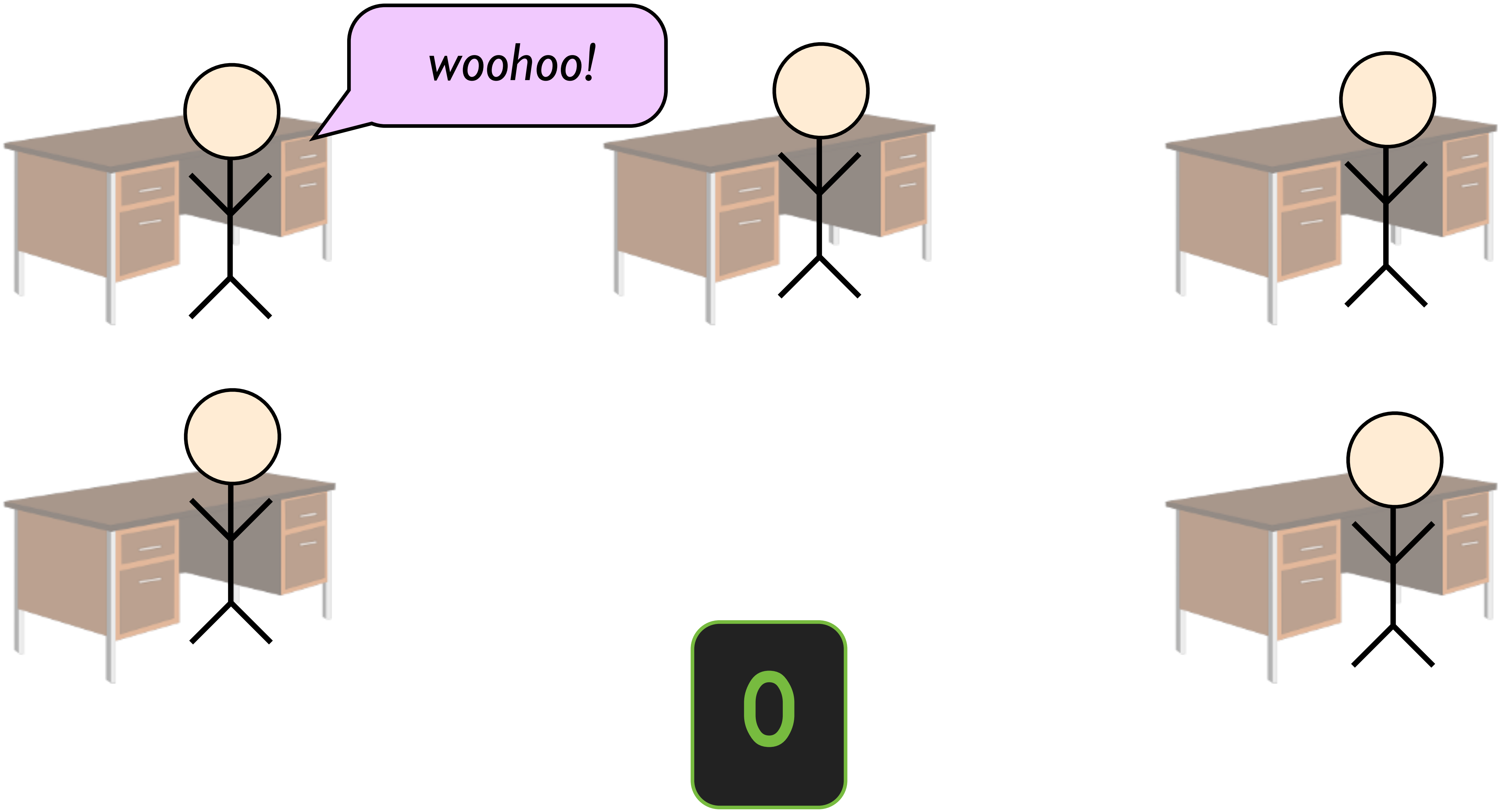
Short diversion: hot desks



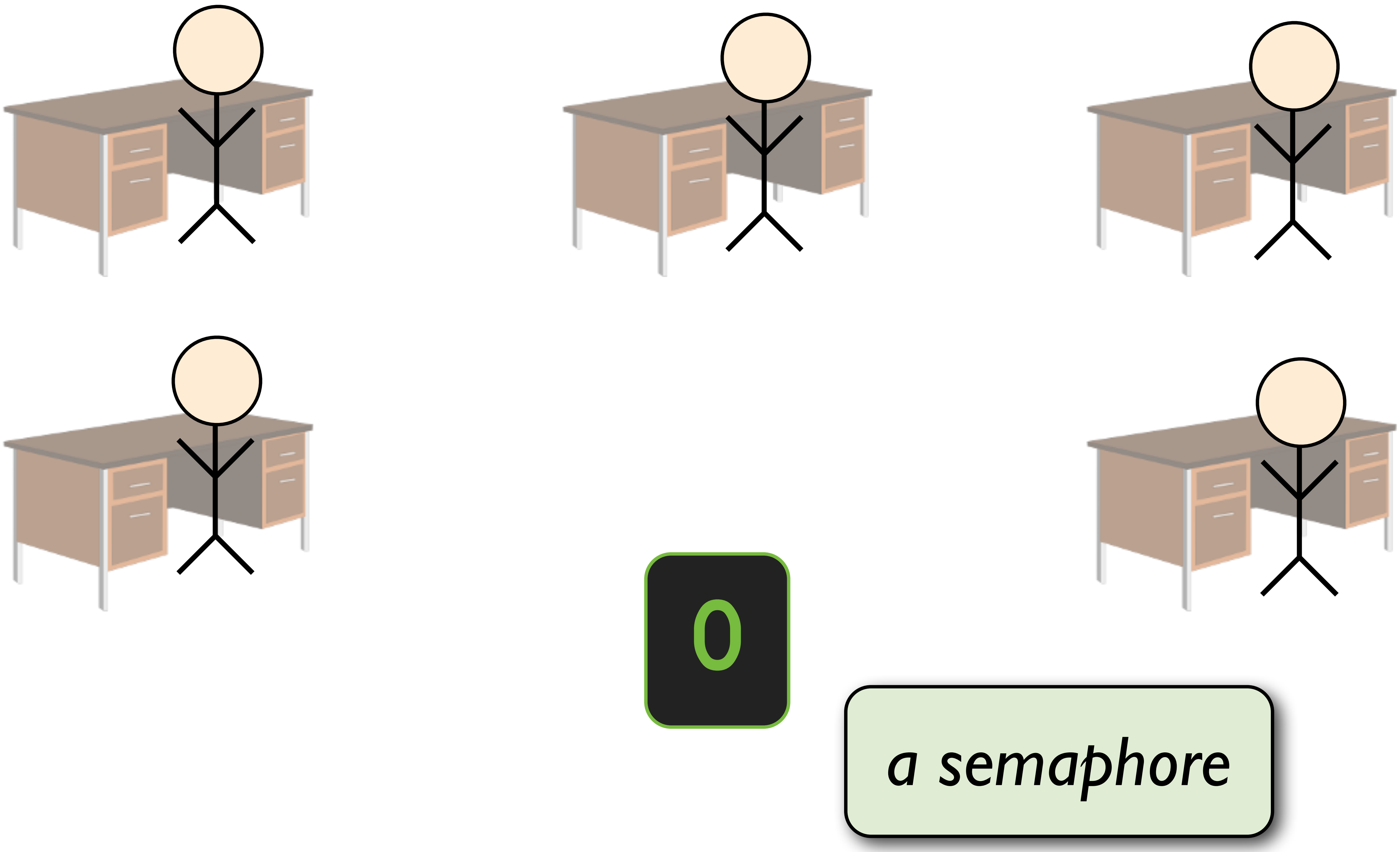
Short diversion: hot desks



Short diversion: hot desks



Short diversion: hot desks



Today's lecture: semaphores

- we will discuss semaphores, an important synchronisation primitive
- conceptually simple, although their implementations require stronger atomic operations
- widespread use in operating systems
- invented by Dijkstra in 1965

Next on the agenda

1. general and binary semaphores

2. implementing semaphores

3. beyond the mutual exclusion problem

4. simulating general semaphores

General semaphores

(aka "counting semaphores")

- a general semaphore is an object consisting of:

  (1) an integer variable *count* such that *count* ≥ 0

  (2) two atomic operations: *down* and *up*

if a process calls *down* when *count* > 0, then *count* is decremented by 1 (otherwise it first waits)

if a process calls *up*, then *count* is incremented by 1

General semaphores

(in Eiffel-like pseudocode)

```
class SEMAPHORE
feature
   count : INTEGER

   down
     do-atomic
        await count > 0
        count := count - 1
     end

   up
     do-atomic
        count := count + 1
     end
end
```

General semaphores
(in Eiffel-like pseudocode)

```
class SEMAPHORE
feature
   count : INTEGER

   down
     do-atomic
        await count > 0
        count := count - 1
     end

   up
     do-atomic
        count := count + 1
     end
end
```

will discuss how to ensure atomicity and how to avoid busy wait later!

Mutual exclusion for <u>two</u> processes

- create a semaphore *s* and initialise *s.count* to 1; then:

s.down
critical section
s.up

Mutual exclusion for <u>two</u> processes

- create a semaphore *s* and initialise *s.count* to 1; then:

s.down
critical section
s.up

one process at a time;
or one hot desk!

Mutual exclusion for <u>two</u> processes

- or in the style of last week's mutual exclusion problems:

count := 1			
P1		P2	
	while true loop		**while true loop**
1	**await** count > 0	1	**await** count > 0
	count := count – 1		count := count – 1
2	critical section	2	critical section
3	count := count + 1	3	count := count + 1
4	non-critical section	4	non-critical section
	end		**end**

Mutual exclusion for two processes

- mutual exclusion and deadlock freedom can be proven

  => remember the atomicity of *down* and *up*!

- solution does not satisfy starvation freedom

  => a different implementation later will fix this

The general semaphore invariant

- general semaphores are characterised by the following invariant -- important for proofs!

- given some semaphore, let:

  => *k* denote its <u>initial</u> value with $k \geq 0$
  => *count* denote its <u>current</u> value
  => *#down* denote the number of completed *down* operations
  => *#up* denote the number of completed *up* operations

- then the following equations are invariant:

$$
\begin{aligned}
&(1)\quad count \geq 0 \\
&(2)\quad count = k + \#up - \#down
\end{aligned}
$$

Binary semaphores

- in the previous example, *s.count* is always either 0 or 1
- such a semaphore is called a binary semaphore and can be implemented using a Boolean variable

b : BOOLEAN

down
  do-atomic
    await *b*
    b := *false*
  end

up
  do-atomic
    b := *true*
  end

Binary semaphores

- in the previous example, *s.count* is always either 0 or 1

- such a semaphore is called a binary semaphore and can be implemented using a Boolean variable

b : BOOLEAN

down
  do-atomic
    await *b*
    b := *false*
  end

up
  do-atomic
    b := *true*
  end

> *This is deceptively similar to the previous weeks' early, and wrong attempts at providing mutual exclusion. What's different?*

Next on the agenda

1. general and binary semaphores ✓
2. implementing semaphores
3. beyond the mutual exclusion problem
4. simulating general semaphores

Avoiding busy waiting

- busy-wait semaphores are not ideal

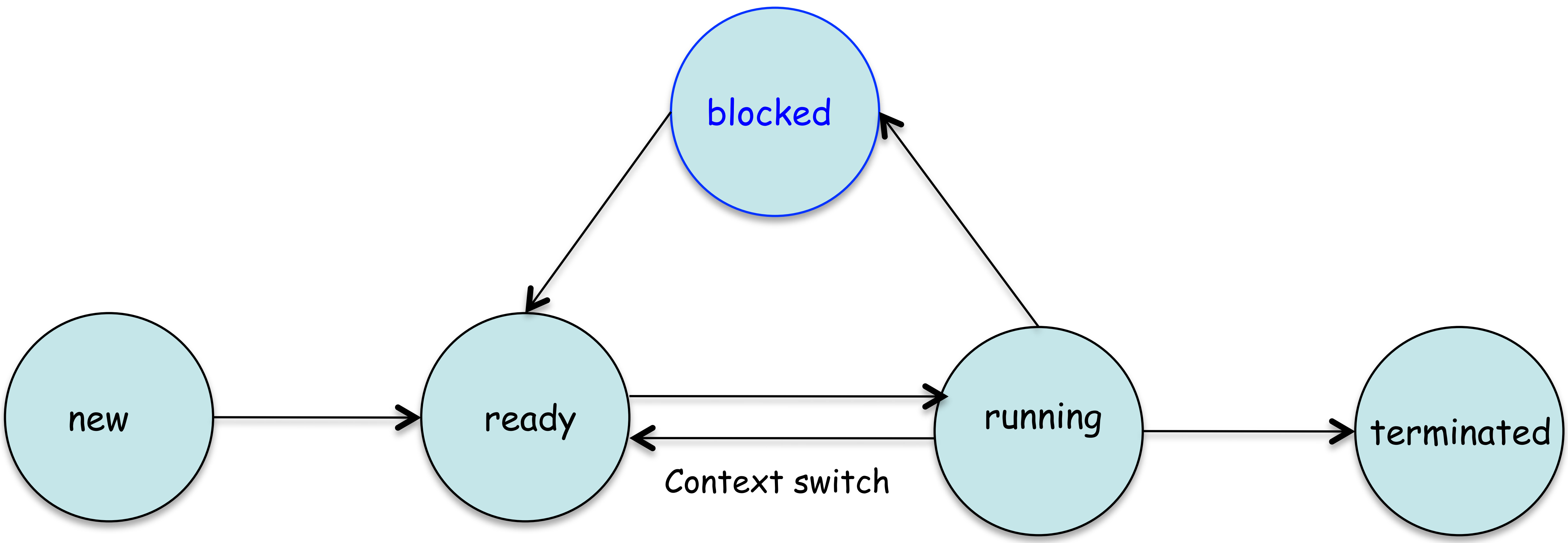
  => they are not starvation free
  => inefficient in the context of multitasking

- more preferable would be for processes to block themselves when having to wait

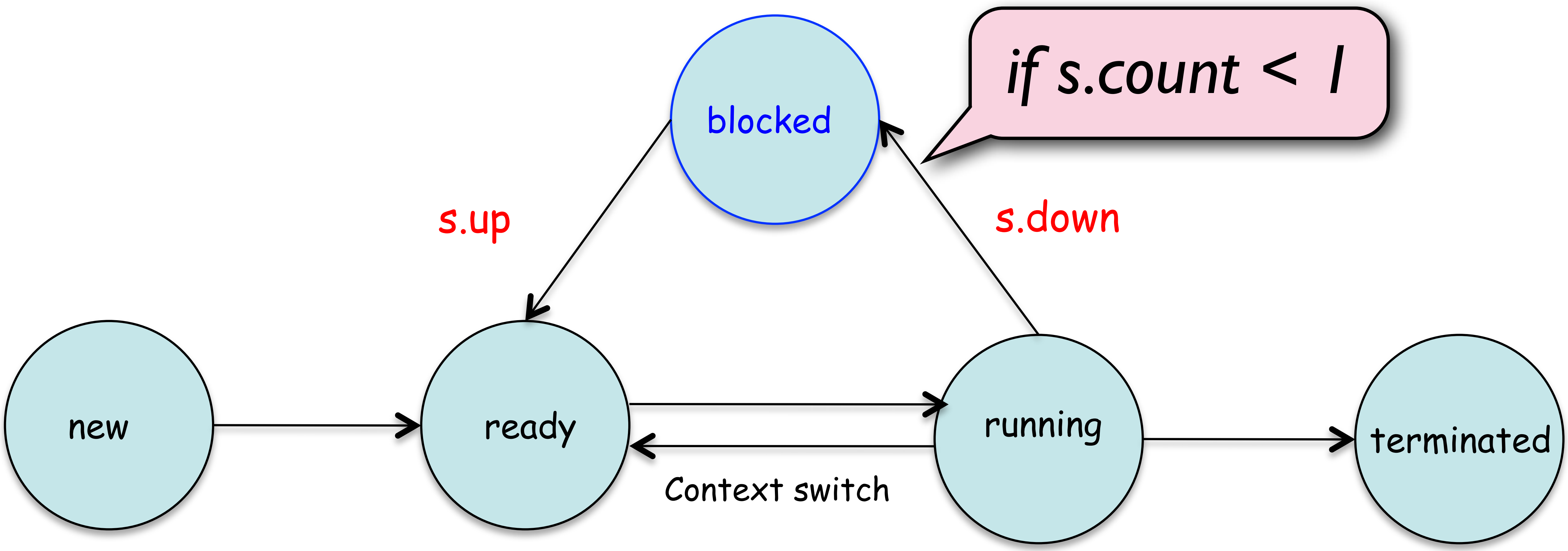
  => thus freeing processing resources as early as possible

- idea: keep track of blocked processes, "waking them" upon *up* calls on the semaphore

Avoiding busy waiting



Avoiding busy waiting



Implementing the scheme

- to avoid starvation, we will track blocked processes in a collection *blocked*

- we equip *blocked* with the following operations, which will be integrated into *down* and *up*

  => *add*(P) -- *insert process P into collection*
  => remove -- *select, remove, and return an item from the collection*
  => is_empty -- *true if collection empty; false otherwise*

- if *blocked* is implemented as a set, we call the semaphore weak; if as a FIFO queue, then strong

Weak semaphore

- a weak semaphore is a blocking semaphore in which the collection *blocked* is implemented as a set

=> *remove* will pick and remove a random element

down

```
do-atomic
   if count > 0 then
      count := count - 1
   else
     blocked.add(P)
     P.state := blocked
   end
end
```

up

```
do-atomic
   if blocked.is_empty then
      count := count + 1
   else
     Q := blocked.remove
     Q.state := ready
   end
end
```

Weak semaphore

- a weak semaphore is a blocking semaphore in which the collection *blocked* is implemented as a set

=> *remove* will pick and remove a random element

```
down
  do-atomic
    if count > 0 then
      count := count - 1
    else
      blocked.add(P)      -- add current process P to blocked
      P.state := blocked  -- block P (instead of busy wait)
    end
  end
```

```
up
  do-atomic
    if blocked.is_empty then
      count := count + 1
    else
      .remove
      ady
    end
  end
```

Weak semaphore

- a weak semaphore is a blocking semaphore in which the collection *blocked* is implemented as a set

=> *remove* will pick and remove a random element

```
down
  do-atomic
    if count > 0 then
      count := count - 1
    else
  end
```

```
-- select and remove some process Q from blocked
-- unblock Q so that it can access the resource
   (Question: why is count left unchanged?)
```

```
up
  do-atomic
    if blocked.is_empty then
       count := count + 1
    else
      Q := blocked.remove
      Q.state := ready
    end
  end
```

Mutual exclusion for <u>two</u> processes

- weak semaphores provide starvation-freedom in the two process scenario

  => why?

- what about mutual exclusion for *n* processes?

Mutual exclusion for *n* processes

- create a semaphore *s* and initialise *s.count* to 1; then:

s.down
critical section
s.up

for each process

- starvation is possible for *n* > 2 with weak semaphores because we select a process from *blocked* at random

- solution is to use a strong semaphore, in which *blocked* is implemented as a FIFO queue

Strong semaphores provide a solution to the mutual exclusion problem with *n* processes

(how to prove)

- mutual exclusion -- show that

$$\boxed{\#cs + count = 1}$$

where #cs is the number of processes in critical sections

- starvation freedom -- apply *proof by contradiction*

=> begin by assuming that a process in *blocked* is starved

A note on atomicity

- operations *down* and *up* are typically built in software using lower-level primitives (e.g. synchronisation algorithms)

- alternatively:

  => using test-and-set instructions (atomic read and write)

  => disabling interrupts (only realistic on a single processing unit)

A note on Java

- java.util.concurrent.Semaphore

  http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html

- constructors

  => Semaphore(int k) -- a weak semaphore
  => Semaphore(int k, boolean b) -- a strong semaphore if b true

- operations

  => acquire() -- corresponds to *down*
  => release() -- corresponds to *up*

Next on the agenda

1. general and binary semaphores
2. implementing semaphores
3. beyond the mutual exclusion problem
4. simulating general semaphores

The *k*-exclusion problem

- in the *k*-exclusion problem, we allow up to *k* processes to simultaneously be in their critical sections

  => mutual exclusion is the $k = 1$ instance

- use a general semaphore corresponding to the number of processes allowed to be in their critical sections

```
s.count := k
P_i
while true loop
   s.down
   critical section
   s.up
   non-critical section
end
```

The *k*-exclusion problem

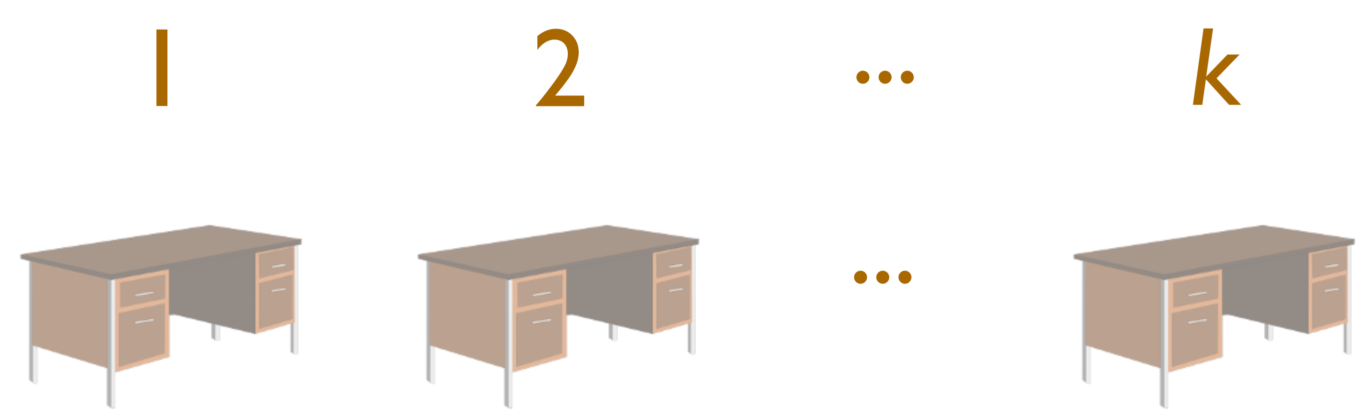
- in the *k*-exclusion problem, we allow up to *k* processes to simultaneously be in their critical sections

  => mutual exclusion is the $k = 1$ instance

- use a general semaphore corresponding to the number of processes allowed to be in their critical sections

s.count := k
P_i

```
while true loop
    s.down
    critical section
    s.up
    non-critical section
end
```



Barriers

- semaphores can be used to control the ordering of events in a system

- a barrier is a form of synchronisation that determines a point in a program's execution that all processes in a group have to reach before any of them may move on

  => important for concurrent iterative algorithms

Barriers

- semaphores can be used to control the ordering of events in a system

- a barrier is a form of synchronisation that determines a point in a program's execution that all processes in a group have to reach before any of them may move on

=> important for concurrent iterative algorithms

s1.count := 0 s2.count := 0	
P1	P2
code before the barrier s1.up s2.down code after the barrier	code before the barrier s2.up s1.down code after the barrier

Barriers

- semaphores can be used to control the ordering of events in a system

- a barrier is a form of synchronisation that determines a point in a program's execution that all processes in a group have to reach before any of them may move on

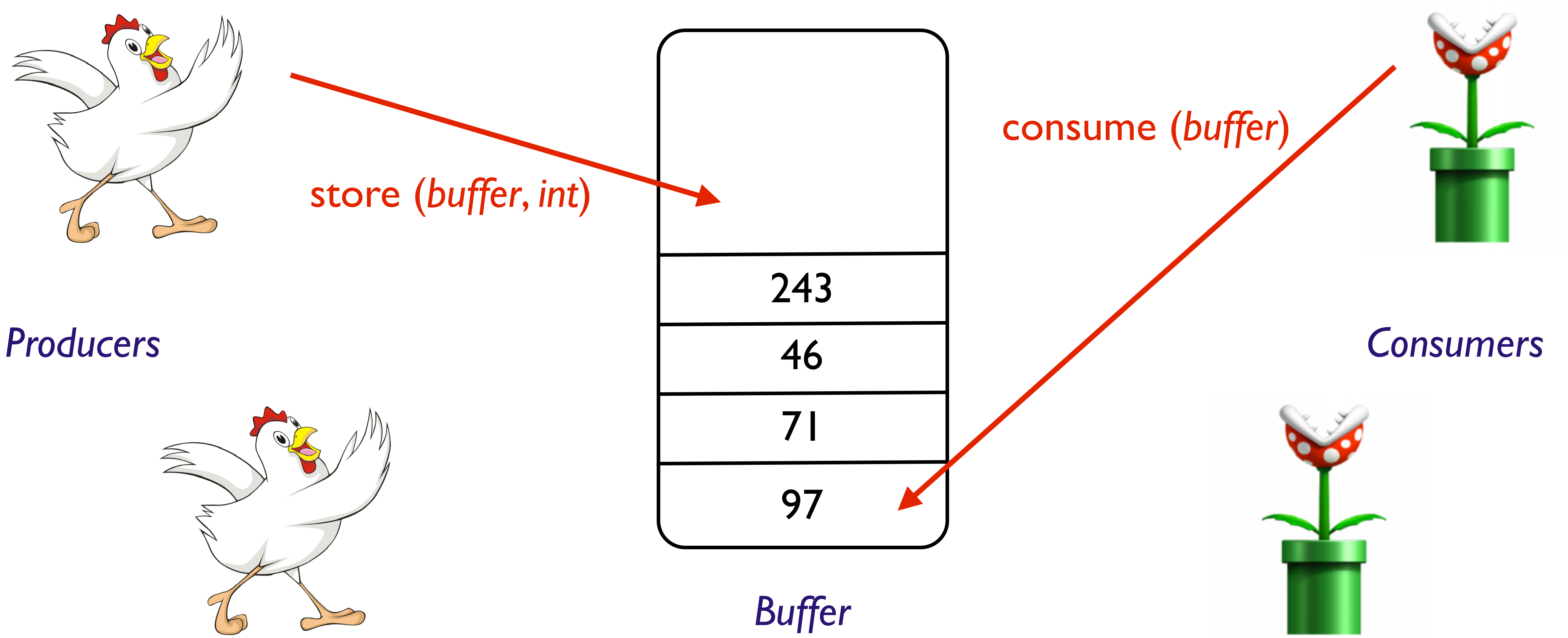
=> important for concurrent iterative algorithms

s1 is the barrier for P2; s2 is the barrier for P1 -- why are they initialised to 0?

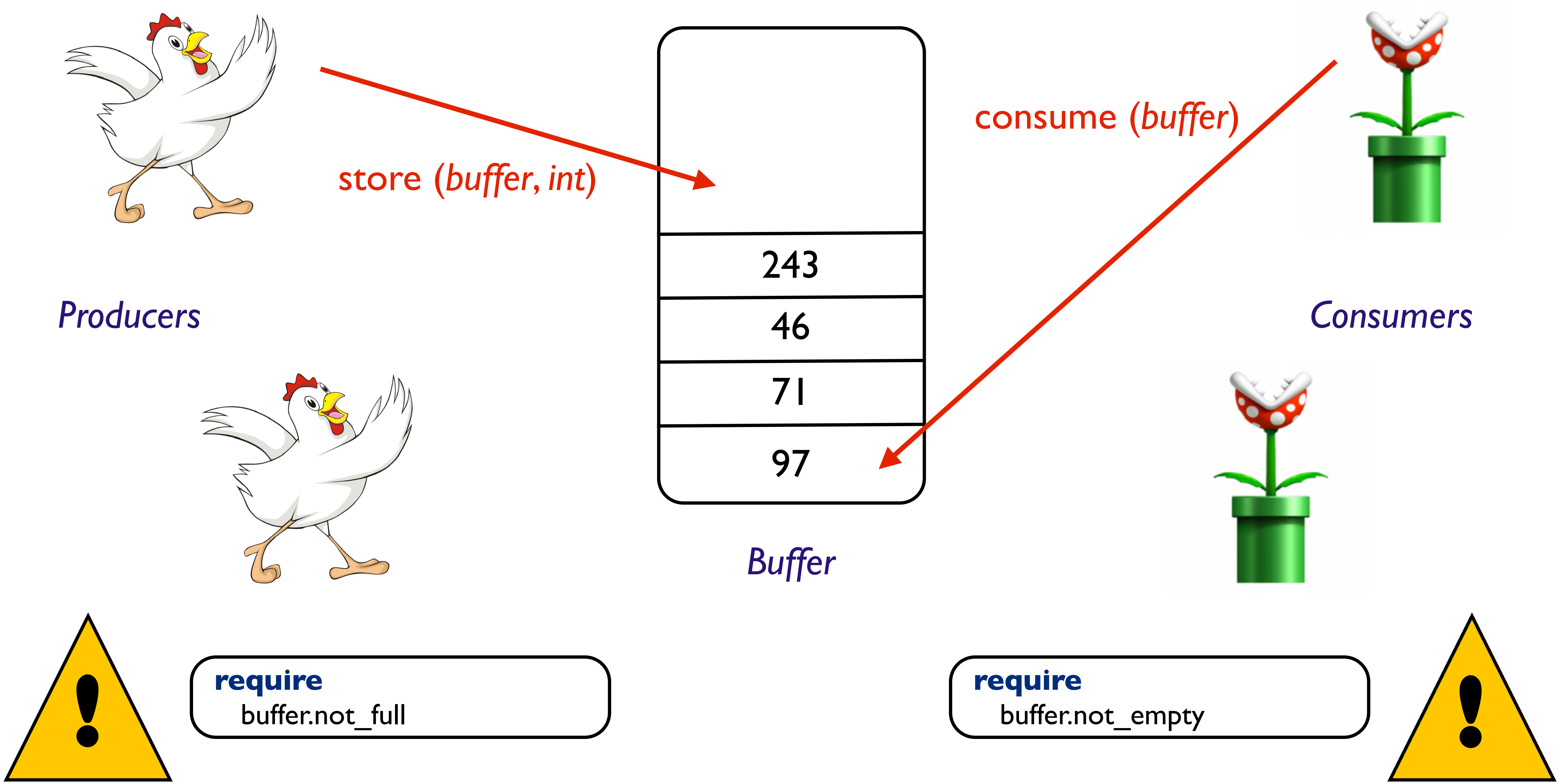
s1.count := 0
s2.count := 0

P1		P2	
1	code before the barrier	1	code before the barrier
2	s1.up	2	s2.up
3	s2.down	3	s1.down
4	code after the barrier	4	code after the barrier

The producer-consumer problem



The producer-consumer problem



The producer-consumer problem

- a good solution would:

  => ensure that every data item produced is eventually consumed
  => be deadlock-free
  => be starvation-free

- need a semaphore for mutual exclusion (the buffer)

- but additional semaphore(s) for condition synchronisation

  => e.g. consumer should block until the buffer is non-empty

Solution for an unbounded buffer

```
mutex.count := 1
not_empty.count := 0
```

Producer$_i$	Consumer$_i$
while true loop	**while true loop**
d := produce	not_empty.down
mutex.down	mutex.down
b.append(d)	d := b.remove
mutex.up	mutex.up
not_empty.up	consume(d)
end	**end**

Solution for an <u>unbounded</u> buffer

observe that not_empty.count = #items_in_buffer

mutex.count := 1 not_empty.count := 0	
Producer$_i$	Consumer$_i$

while true loop
    d := produce
    mutex.down
    b.append(d)
    mutex.up
    not_empty.up
end
	while true loop
	    not_empty.down
	    mutex.down
	    d := b.remove
	    mutex.up
	    consume(d)
	end

Solution for an <u>un</u>bounded buffer

observe that not_empty.count = #items_in_buffer

```
mutex.count := 1
not_empty.count := 0
```

Producer$_i$	Consumer$_i$
while true loop d := produce b.append(d) mutex.up not_empty.up **end**	**while true loop** not_empty.down mutex.down d := b.remove mutex.up consume(d) **end**

blocks until not_empty.count > 0

Solution for a bounded buffer

```
mutex.count := 1
not_empty.count := 0
not_full.count := k
```

where k is the size of the buffer

Producer$_i$

```
   while true loop
1      d := produce
2      not_full.down
3      mutex.down
4      b.append(d)
5      mutex.up
       not_empty.up
   end
```

Consumer$_i$

```
   while true loop
1      not_empty.down
2      mutex.down
3      d := b.remove
4      mutex.up
5      not_full.up
       consume(d)
   end
```

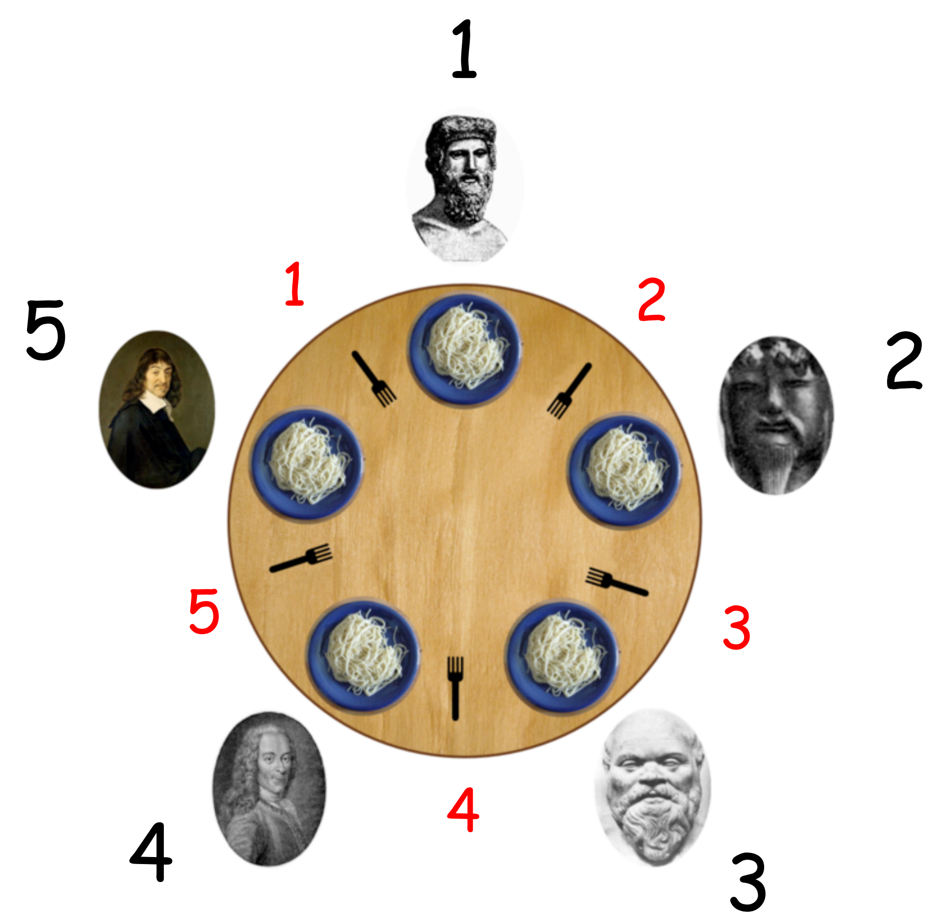
Dining philosophers problem

(a solution that can deadlock)

- multiple semaphores must be used with care -- they are prone to deadlock!

s[1].count := 1, ..., s[n].count := 1
Philosopher$_i$

```
while true loop
    think
    s[i].down
    s[(i mod n) + 1].down
    eat
    s[(i mod n) + 1].up
    s[i].up
end
```



Dining philosophers problem

(a solution that can deadlock)

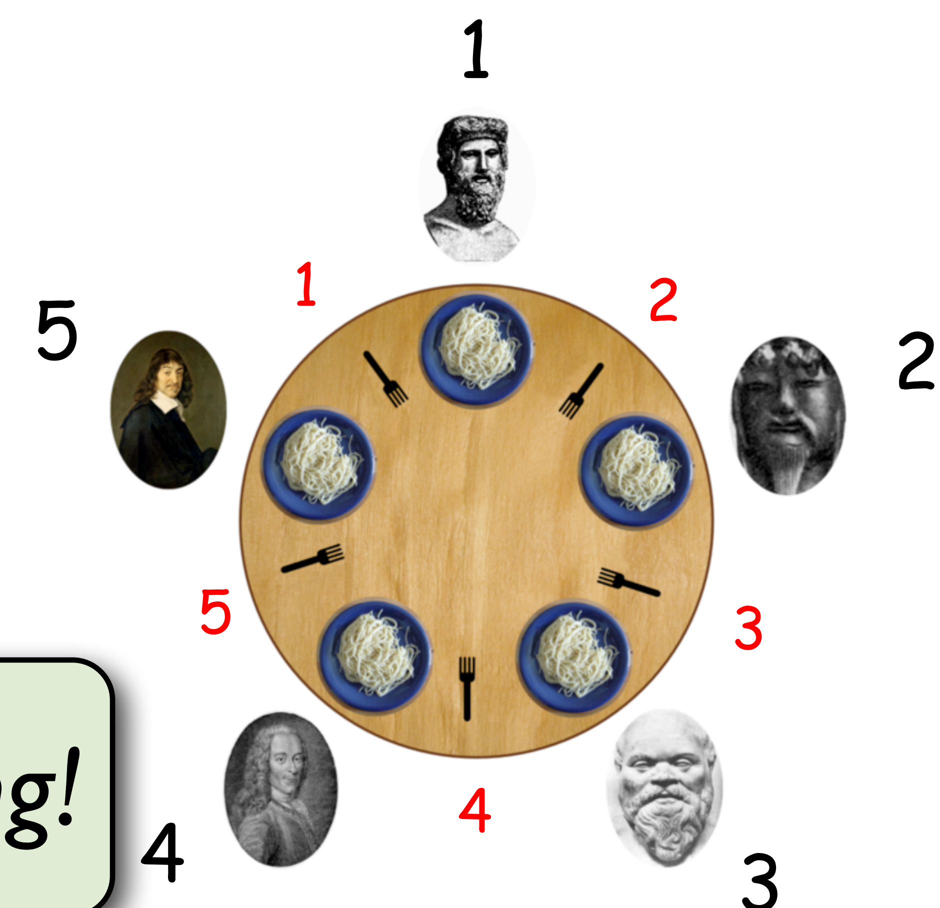
- multiple semaphores must be used with care -- they are prone to deadlock!

```
s[1].count := 1, ..., s[n].count := 1

Philosopher_i

while true loop
    think
    s[i].down
    s[(i mod n) + 1].down
    eat
    s[(i mod n) + 1].up
    s[i].up
end
```

circular waiting!



Dining philosophers problem

(a fix!)

- assume that philosopher *n* picks up the left fork before the right fork

- this breaks the circle of resource requests; there will always be one philosopher who can acquire both forks and release them again

Philosopher$_n$

```
while true loop
   think
   s[1].down
   s[n].down
   eat
   s[n].up
   s[1].up
end
```

Next on the agenda

1. general and binary semaphores
2. implementing semaphores
3. beyond the mutual exclusion problem
4. simulating general semaphores

General semaphores are superfluous

- while conceptually useful, general semaphores (theoretically) are not necessary -- they can be implemented through binary semaphores alone

General semaphores are superfluous

```
mutex.count := 1  -- binary semaphore
delay.count := 1    -- binary semaphore
count := k
```

```
general_down
   do
      delay.down
      mutex.down
      count := count - 1
      if count > 0 then
         delay.up
      end
      mutex.up
   end
```

```
general_up
   do
      mutex.down
      count := count + 1
      if count = 1 then
         delay.up
      end
      mutex.up
   end
```

General semaphores are superfluous

```
mutex.count := 1  -- binary semaphore
delay.count := 1  -- binary semaphore
count := k
```

value of the general semaphore

protects count

```
general_down
   do
      delay.down
      mutex.down
      count := count - 1
      if count > 0 then
         delay.up
      end
      mutex.up
   end
```

not called when count = 0

```
general_up
   do
      mutex.down
      count := count + 1
      if count = 1 then
         delay.up
      end
      mutex.up
   end
```

Next on the agenda

1. general and binary semaphores
2. implementing semaphores
3. beyond the mutual exclusion problem
4. simulating general semaphores

Summary

- semaphores are conceptually simple but powerful tools for solving synchronisation problems

- choice of implementation can affect starvation-freedom

- applications beyond mutual exclusion: *k*-exclusion, barriers, condition synchronisation

but: correct usage is still far from trivial

- essential reading: Chapter 4 of the CCC textbook