Concepts of Concurrent Computation
Spring 2014
Lecture 5: Monitors

Bertrand Meyer
Sebastian Nanz
Chris Poskitt

Chair of Software Engineering
ETH zürich

Last week: semaphores

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

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

but: correct usage is still far from trivial

=> must consider the whole program to determine a semaphore's correct use
=> multiple semaphores difficult (e.g. dining philosophers)
=> missing one *down* or *up* could introduce deadlock

Today: a little more abstraction

- we will talk about monitors -- an approach that provides synchronisation in a more structured manner
- based on object-oriented principles

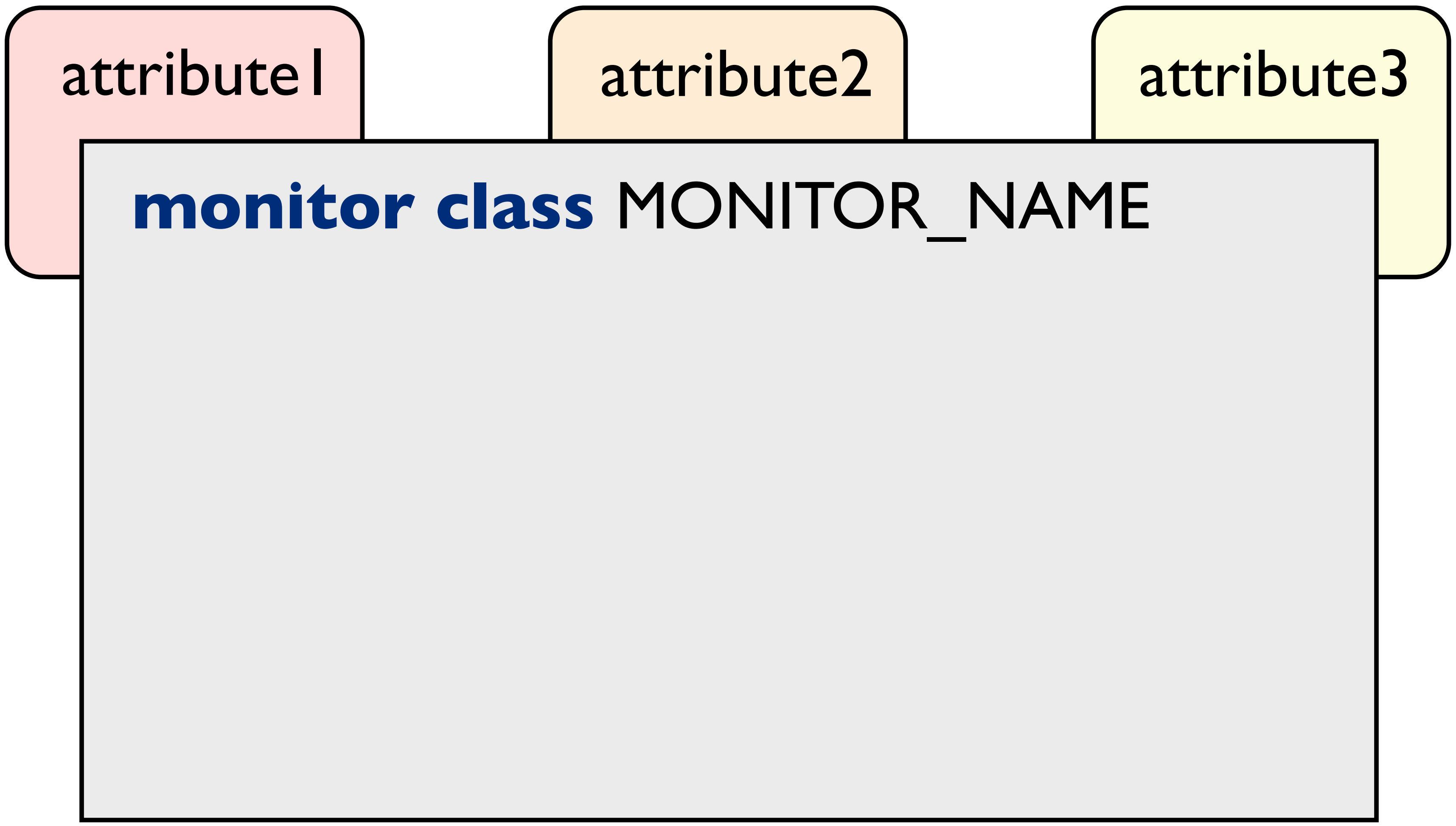
  => class
  => encapsulation

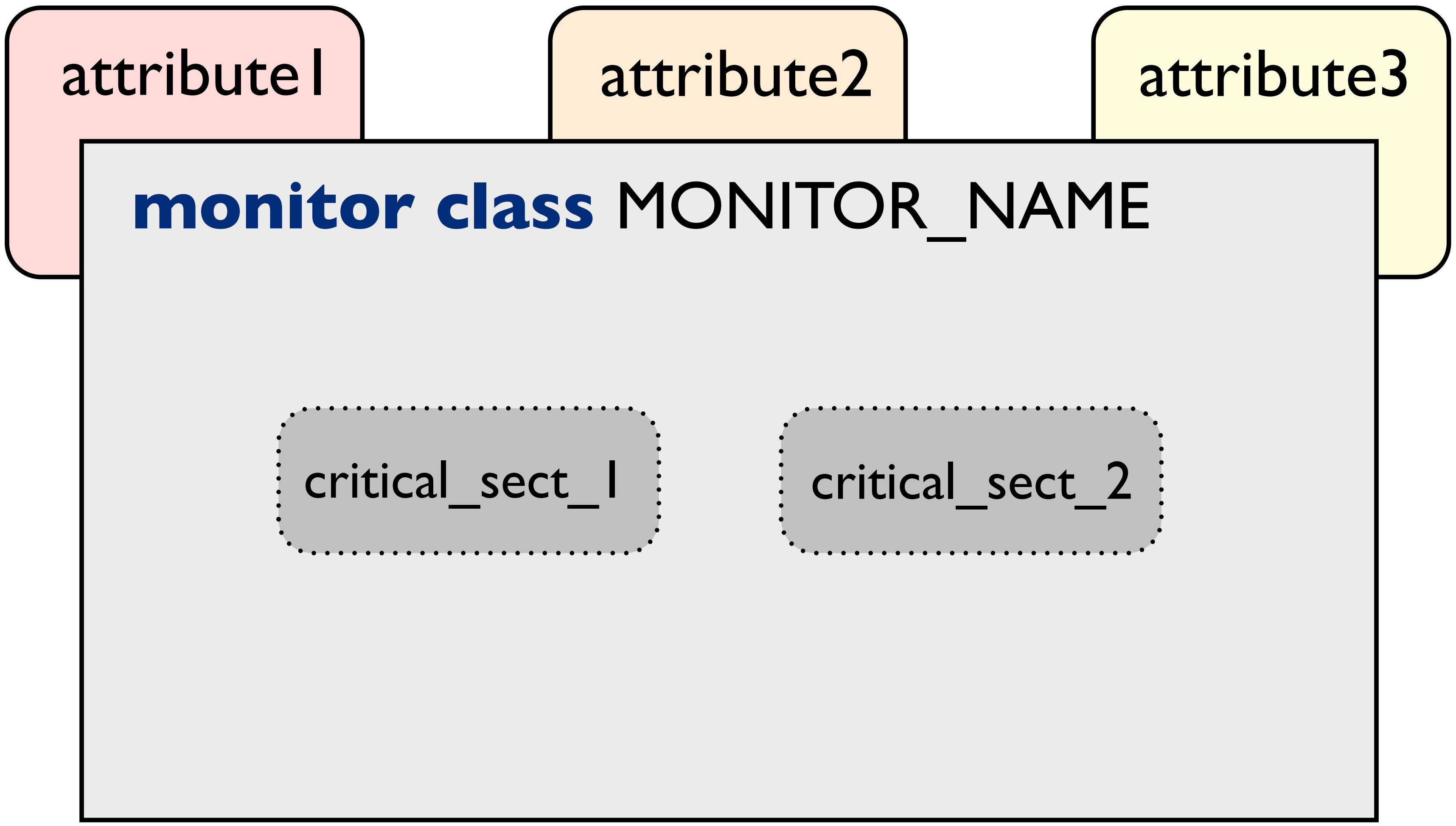
- mutual exclusion handled implicitly; or "for free"

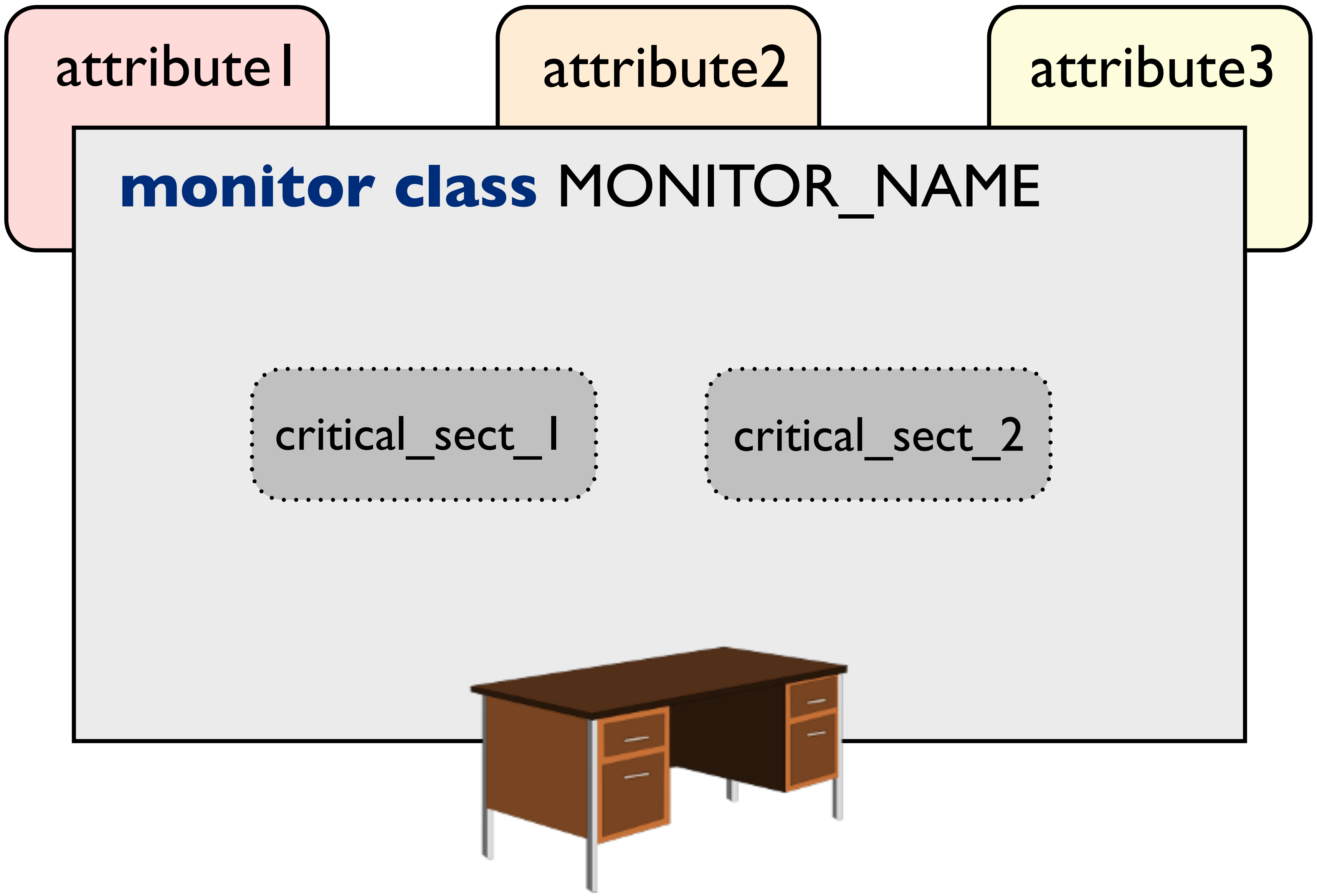
  => aims to greatly reduce the number of
      programmer errors

- invented by Hoare and Brinch Hansen


monitor class MONITOR_NAME


attribute1
attribute2
attribute3
monitor class MONITOR_NAME


attribute1
attribute2
attribute3
monitor class MONITOR_NAME
critical_sect_1
critical_sect_2



routines executed under mutual exclusion!

Next on the agenda

1. monitors and mutual exclusion
2. condition synchronisation
3. signalling disciplines
4. applications of monitors

Monitors

- a monitor class is a class that fulfills the following conditions:

  => all its attributes are declared *private*
  => its routines execute with *mutual exclusion*

- a monitor is an object instantiating a monitor class

Monitors

- a monitor class is a cla
conditions:

=> all its attributes are declared *private*
=> its routines execute with *mutual exclusion*

- a monitor is an object i

attributes correspond to shared variables

routine bodies correspond to critical sections

Monitor class notation

```
monitor class MONITOR_NAME
   feature
      -- attribute declarations
      a1 : TYPE1
       . . .

      -- routine declarations
      r1 (arg1, ..., argk) do ... end
       . . .

   invariant
      -- monitor invariant
end
```

Solution to the mutual exclusion problem

```
monitor class CS
   feature
      x_1 : TYPE₁   . . .   x_m : TYPEₘ  -- shared data
      critical_1
         do
            critical section₁
         end
      . . .
      critical_n
         do
            critical sectionₙ
         end
end
```

Solution to the mutual exclusion problem

```
monitor class CS
  feature
    x_1 : TYPE1   . . .   x_m : TYPEm  -- shared data
    critical_1
      do
        critical section1
      end
     . . .
    critical_n
      do
        critical sectionn
      end
end
```

```
while true loop
  cs.critical_i
  non-critical section
end
```

for each process

Ensuring mutual exclusion in monitors

- the requirement that at most one routine is active inside a monitor at any time is ensured by the implementation of monitors

  => not burdened on the programmer!

- can do so using strong semaphores

  => *entry* : SEMAPHORE

- intuition: *entry* is used as the monitor's lock

Ensuring mutual exclusion in monitors

- *entry* is initialised to 1

- monitor routines must acquire the semaphore before executing their bodies

```
r (arg_1, ..., arg_k)
   do
      entry.down
      body_r
      entry.up
   end
```

- the FIFO process queue *entry.blocked* acts as the entry queue of the monitor

Next on the agenda

1. monitors and mutual exclusion
2. condition synchronisation
3. signalling disciplines
4. applications of monitors

Condition variables

- monitors also support condition synchronisation through so-called condition variables
- their semantics differs to those of semaphores for condition synchronisation

  => deeply intertwined with the monitor concept

- intention: separating the concerns of mutual exclusion and condition synchronisation

  => make programs easier to read

"Programs must not be regarded as code for computers, but as literature for humans"

N. Wirth, 2014

Condition variables

- a condition variable consists of a queue *blocked* and three atomic operations:

  => *wait*

  => *signal*

  => *is_empty*

- operations *wait* and *signal* can only be called from the body of a monitor routine

Condition variables

- a condition variable consists of a queue *blocked* and three atomic operations:

  => *wait* — *releases the lock on the monitor, blocks the executing thread and appends it to blocked*

  => *signal*

  => *is_empty*

- operations *wait* and *signal* can only be called from the body of a monitor routine

Condition variables

- a condition variable consists of a queue *blocked* and three atomic operations:

=> *wait* — *releases the lock on the monitor, blocks the executing thread and appends it to blocked*

=> *signal* — *if blocked is empty, then no effect; otherwise it unblocks a thread*

=> *is_empty*

side effects possible, depending on signalling discipline

- operations *wait* and *signal* can only be called from the body of a monitor routine

Condition variables

- a condition variable consists of a queue *blocked* and three atomic operations:

=> *wait*

releases the lock on the monitor, blocks the executing thread and appends it to blocked

=> *signal*

if blocked is empty, then no effect; otherwise it unblocks a thread

=> *is_empty*

returns true if blocked is empty; false otherwise

- operations *wait* and *signal* can only be called from the body of a monitor routine

Semaphores vs. monitors

down

only blocks if *count = 0*

wait

always blocks

---

up

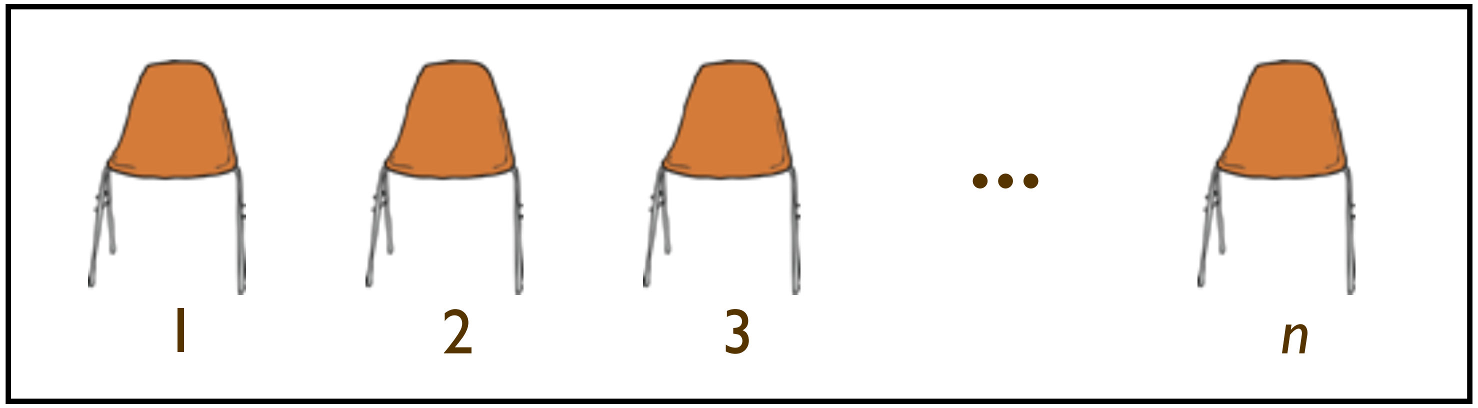
always has an effect

signal

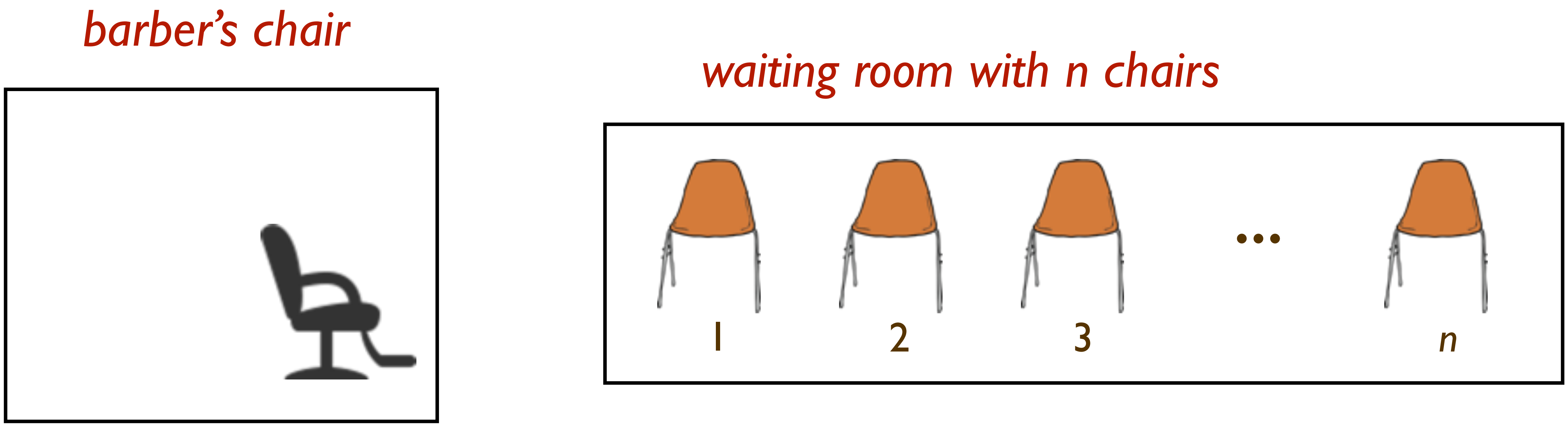
no effect if no blocked process

Sleeping barber problem

waiting room with n chairs



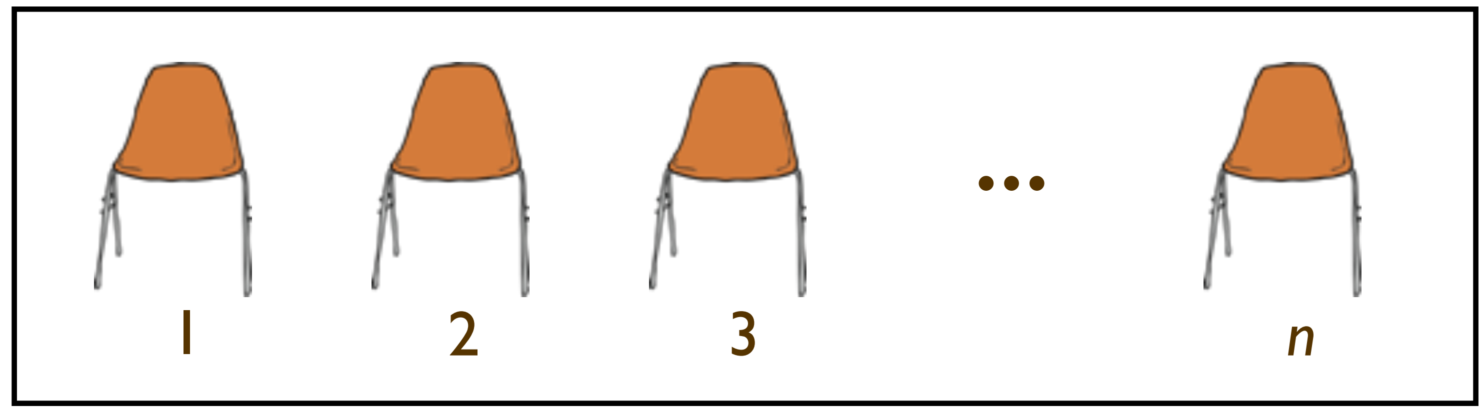
Sleeping barber problem



Sleeping barber problem

barber's chair

waiting room with n chairs



- the barber and customers abide by the following rules:

  => *if there are no customers in the waiting room, then the barber goes to sleep*

  => *if a customer enters the shop and finds the barber sleeping, they wake him up and get a haircut*

  => *if the barber is busy but there are free chairs in the waiting room, then the customer sits in a chair and waits to be called by the barber*

  => *if all chairs are occupied, then the customer leaves the shop*

Sleeping barber problem

- challenge is to find a starvation-free algorithm that observes the rules
- motivation: client-server relationships between operating system processes
- generalisation of barriers (as discussed last week)

  => *two parties must arrive before they can proceed*
  => *but the second party is not predetermined...*
  => *...could be any customer!*

Sleeping barber problem

```
monitor class SLEEPING_BARBER
  feature
    num_free_chairs : INTEGER
    barber_available : CONDITION_VARIABLE
    customer_available : CONDITION_VARIABLE

    get_haircut
      do
        if num_free_chairs > 0 then
          num_free_chairs :=
            num_free_chairs - 1
          customer_available.signal
          barber_available.wait
        end
      end
      -- get a haircut

    do_haircut
      do
        while num_free_chairs = n do
          customer_available.wait
        end
        barber_available.signal
        num_free_chairs :=
          num_free_chairs + 1
      end
      -- do a haircut
end
```

Sleeping barber problem

```
monitor class SLEEPING_BARBER
  feature
    num_free_chairs : INTEGER
    barber_available : CONDITION_VARIABLE      -- express that barber is available
    customer_available : CONDITION_VARIABLE    -- express that customer is waiting

    get_haircut
      do
        if num_free_chairs > 0 then
          num_free_chairs :=
            num_free_chairs - 1
          customer_available.signal
          barber_available.wait
        end
      end
      -- get a haircut

    do_haircut
      do
        while num_free_chairs = n do
          customer_available.wait
        end
        barber_available.signal
        num_free_chairs :=
          num_free_chairs + 1
      end
      -- do a haircut
end
```

Sleeping barber problem

```
monitor class SLEEPING_BARBER
  feature
    num_free_chairs : INTEGER
    barber_available : CONDITION_VARIABLE
    customer_available : CONDITION_VARIABLE

    get_haircut                                  do_haircut
      do                                           do
        if num_free_chairs > 0 then                   ... do
          num_free_chairs :=                            customer_available.wait
            num_free_chairs - 1                       end
          customer_available.signal
          barber_available.wait
        end
      end                                          end
      -- get a haircut                             -- do a haircut
                                  end
```

-- *if no free chairs, exit without haircut*

-- *otherwise, take a chair, signal that a customer is waiting, and block on the condition variable barber_available*

Sleeping barber problem

```
monitor class SLEEPING_BARBER
   feature
      num_free_chairs : INTEGER
      barber_available : CONDITION_VARIABLE
      customer_available : CONDITION_VARIABLE
```

-- do_haircut called in an infinite loop

```
      g
         do
            if num_free_chairs > 0 then
               num_free_chairs :=
```

-- block on customer_available if all n seats are free (no customers)
-- when customers waiting, signals to waiting customer that barber is ready

```
         -- get a haircut
```

```
      do_haircut
         do
            while num_free_chairs = n do
               customer_available.wait
            end
            barber_available.signal
            num_free_chairs :=
               num_free_chairs + 1
         end
         -- do a haircut
   end
```

Sleeping barber problem

```
monitor class SLEEPING_BARBER
  feature
    num_free_chairs : INTEGER
    barber_available : CONDITION_VARIABLE
    customer_available : CONDITION_VARIABLE

    get_haircut
      do
        if num_free_chairs > 0 then
          num_free_chairs :=
            num_free_chairs - 1
          customer_available.signal
          barber_available.wait
        end
      end
      -- get a haircut

    do_haircut
      do
        while num_free_chairs = n do
          customer_available.wait
        end
        barber_available.signal
        num_free_chairs :=
          num_free_chairs + 1
      end
      -- do a haircut
end
```

Implementing condition variables

```
class CONDITION_VARIABLE
feature
    blocked: QUEUE
    wait
        do-atomic
            entry.up             -- release the lock on the monitor
            blocked.add(P)       -- P is the current process
            P.state := blocked   -- block process P
        end
    signal deferred end
    is_empty: BOOLEAN
        do-atomic
            result := blocked.is_empty
        end
end
```

behaviour depends on signalling discpline

Next on the agenda

1. monitors and mutual exclusion ✓
2. condition synchronisation ✓
3. signalling disciplines
4. applications of monitors

Signalling disciplines

- a process that signals on a condition variable is still executing inside the monitor

- at most one process can execute within a monitor at any time

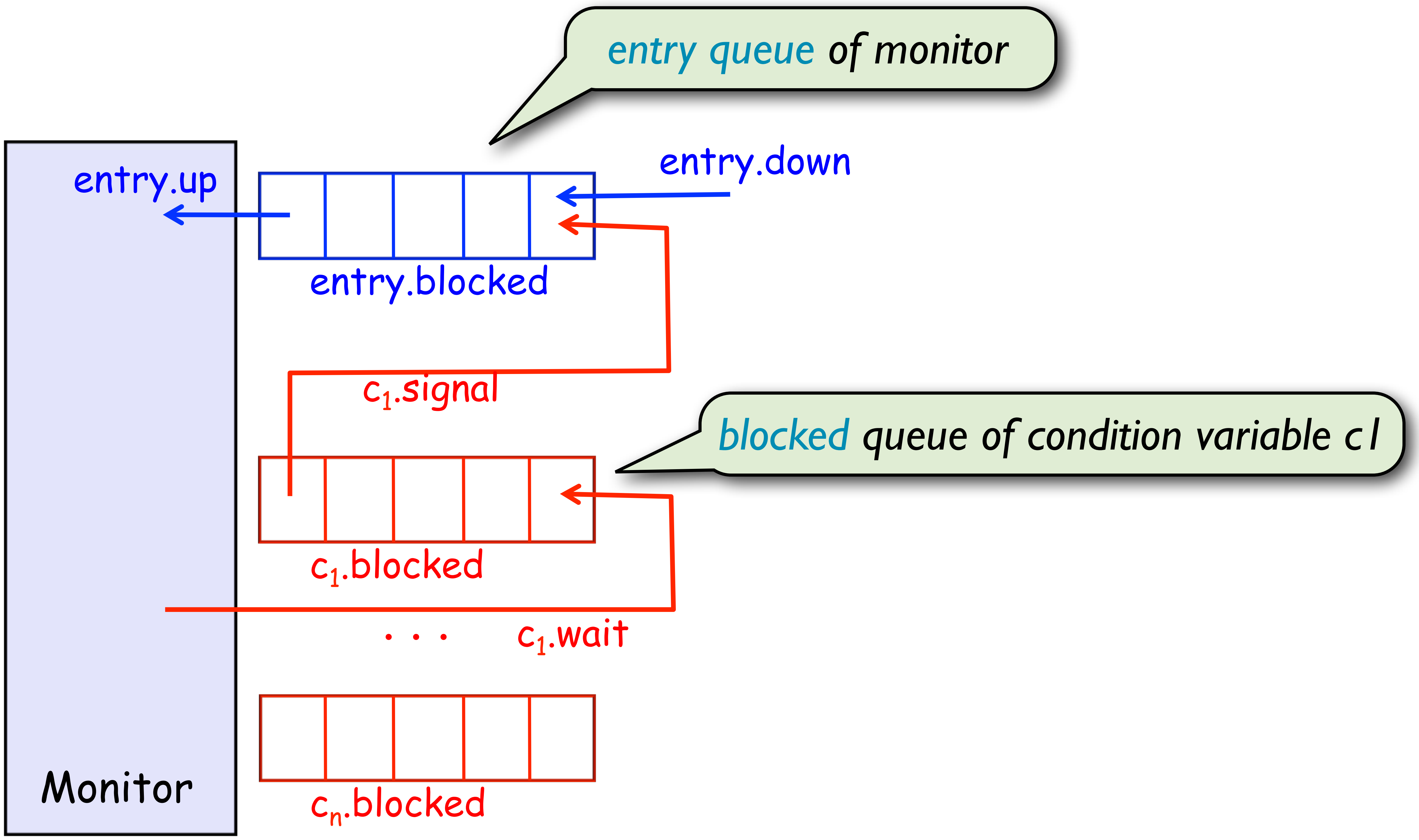
- hence an unblocked process cannot enter the monitor immediately

- we will look at two signalling disciplines

  => signalling process continues; signalled process moved to entry queue of the monitor

  => signalling process leaves the monitor; signalled process continues

Signal and continue



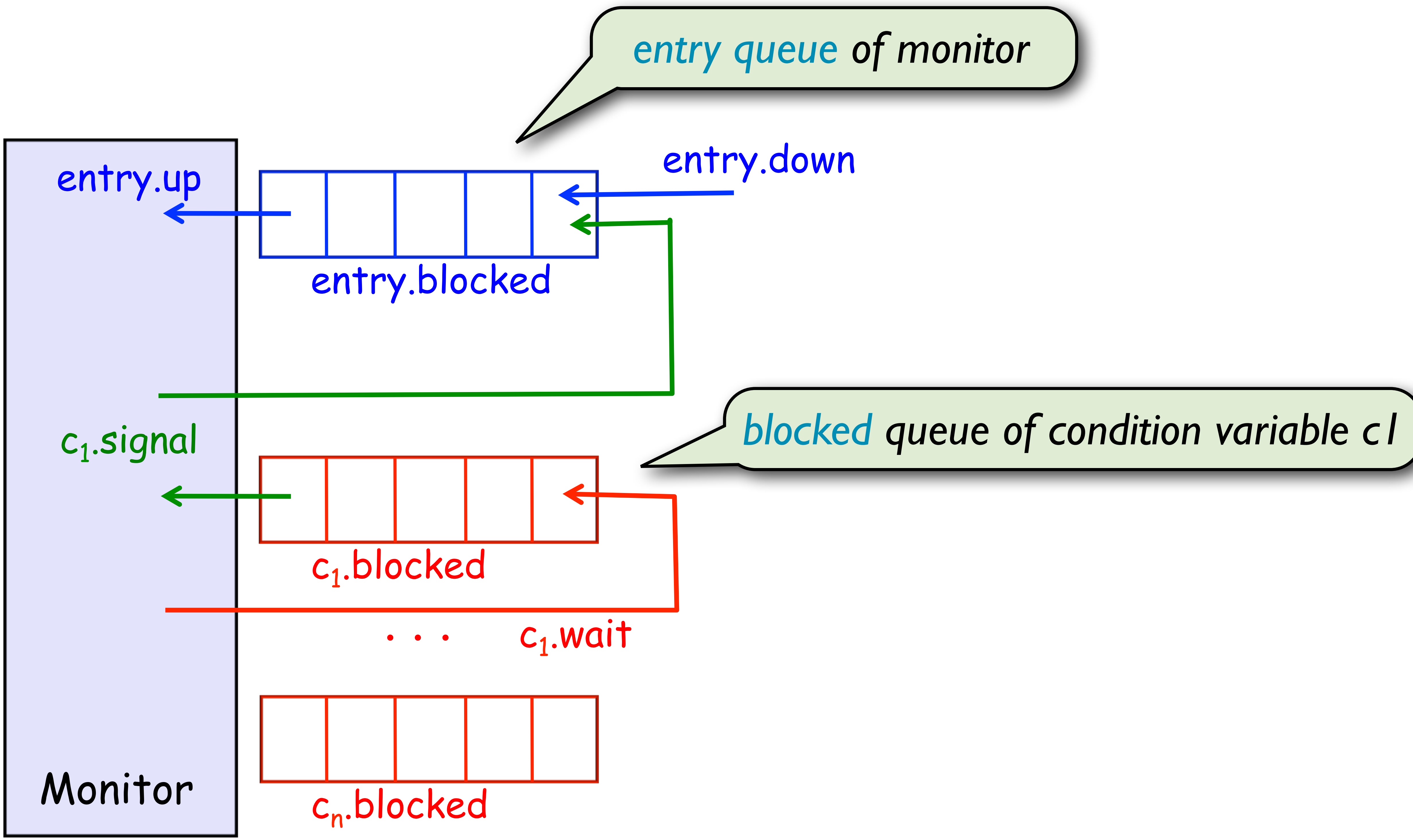
Signal and continue

- signal and continue signalling discipline:

=> the signall<u>ing</u> process continues
=> the signall<u>ed</u> process is moved to monitor's entry queue

```
signal
    do-atomic
        if not blocked.is_empty then
            Q := blocked.remove
            entry.blocked.add(Q)
        end
    end
```

Signal and wait



Signal and wait

- signal and wait signalling discipline:

  => the signalling process is moved to monitor's entry queue
  => the signalled process continues (monitor's lock is silently passed on)

```
signal
  do-atomic
    if not blocked.is_empty then
      entry.blocked.add(P)  -- P is the current process
      Q := blocked.remove
      Q.state := ready       -- unblock process Q
      P.state := blocked     -- block process P
    end
  end
```

Signal and continue vs. signal and wait

- if a process executes a signal and wait signal to indicate that a certain condition is now true, then this condition will be true for the signalled process

- not so for signal and continue: other processes may execute the monitor before the signalled process and may possibly make the condition false

=> can only take the signal as a "hint"
=> *signal and wait* monitors can thus be easier to program

Classification of signalling disciplines

- we can classify three sets of processes:

  S -- signalling processes
  U -- processes unblocked on the condition

- we write X > Y to express that processes in set X have priority over those in set Y, i.e.

=> signal and continue	S > U
=> signal and wait	U > S

Other signalling disciplines

- there are variations that differ in the way that priority is given to processes waiting due to a *signal* call vs. processes waiting in the monitor's entry queue

  S -- signalling processes
  U -- processes unblocked on the condition
  B -- blocked processes on the monitor's entry queue

- we express these other disciplines concisely:

=> signal and continue	$S > U = B$
=> urgent signal and continue	$S > U > B$
=> signal and wait	$U > S = B$
=> signal and urgent wait	$U > S > B$

Remark: monitors can simulate semaphores

- of theoretical interest -- we do not lose expressivity by using monitors instead of semaphores
- assume a signal and continue signalling discipline

```
monitor class STRONG_SEMAPHORE
feature
    count : INTEGER
    count_positive : CONDITION_VARIABLE
    down
        do
            if count > 0 then count := count – 1
            else count_positive.wait end
        end
    up
        do
            if count_positive.is_empty then count := count + 1
            else count_positive.signal end
        end
end
```

Remark: monitors in Java

- each object in Java has a mutex lock that can be acquired and released with *synchronized* blocks

```
Object lock = new Object();

synchronized (lock) {
    // critical section
}
```

- the following are equivalent:

```
synchronized type m(args) {

    // body

}
```

```
type m(args) {
    synchronized (this) {
        // body
    }
}
```

Remark: monitors in Java

- with *synchronized* methods, monitors can be emulated

- condition variables are not explicitly available, but *wait()* and *notify()* [i.e. *signal*] methods can be called on *synchronized* objects

- signal and continue signalling discipline is used

- Java "monitors" are not starvation-free; when *notify()* is invoked, an arbitrary process is unblocked

Next on the agenda

1. monitors and mutual exclusion
2. condition synchronisation
3. signalling disciplines
4. applications of monitors

The readers-writers problem

- in the readers-writers problem we consider shared data which can be accessed by two kinds of processes

  => readers: *processes that may execute concurrently with other readers, but must exclude writers*

  => writers: *processes that must exclude both readers and other writers*

- relevant for databases, shared files, heap structures

- solution should adhere to the access requirements and be starvation free

Readers-writers: the challenge

- we cannot use monitors in the classical way, i.e. encapsulating shared data as their attributes

  => wouldn't permit multiple readers

- solution: use a monitor only to coordinate access

  => shared data accesses enclosed by calls to monitor routines

readers	*writers*
rw.read_entry *read access to shared data* rw.read_exit	rw.write_entry *write access to shared data* rw.write_exit

Monitor solution to readers-writers

```
monitor class READERS_WRITERS
   feature
      num_readers : INTEGER
      num_writers : INTEGER
      ok_to_read : CONDITION_VARIABLE
                    -- signal if num_writers = 0
      ok_to_write : CONDITION_VARIABLE
                    -- signal if num_readers = 0

        . . .

   invariant
      num_writers = 0 or (num_writers = 1 and num_readers = 0)
end
```

Readers-writers: read methods

```
read_entry
        do
            if num_writers > 0 or not ok_to_write.is_empty do
                ok_to_read.wait
            end
            num_readers := num_readers + 1
            ok_to_read.signal
        end
read_exit
        do
            num_readers := num_readers - 1
            if num_readers = 0 then
                ok_to_write.signal
            end
        end
```

Readers-writers: read methods

```
read_entry
    do
        if num_writers > 0 or not ok_to_write.is_empty do
            ok_to_read.wait
        end
        num_readers := num_readers + 1
        ok_to_read.signal
    end
read_exit
    do
        num_readers := num_readers - 1
        if num_readers = 0 then
            ok_to_write.signal
        end
    end
```

preserve invariant

gives writers priority

other readers can access

Readers-writers: read methods

```
read_entry
        do
            if num_writers > 0 or not ok_to_write.is_empty do
                ok_to_read.wait
            end
            num_readers := num_readers + 1
            ok_to_read.signal
        end
read_exit
        do
            num_readers := num_readers - 1
            if num_readers = 0 then
                ok_to_write.signal
            end
        end
```

a writer can now access

Readers-writers: write methods

```
write_entry
      do
         if num_writers > 0 or num_readers > 0 do
            ok_to_write.wait
         end
         num_writers := num_writers + 1
      end
write_exit
      do
         num_writers := num_writers - 1
         if ok_to_read.is_empty then
            ok_to_write.signal
         else
            ok_to_read.signal
         end
      end
```

Readers-writers: write methods

```
write_entry
        do
            if num_writers > 0 or num_readers > 0 do
                ok_to_write.wait
            end
            num_writers := num_writers + 1
        end
write_exit
        do
            num_writers := num_writers - 1
            if ok_to_read.is_empty then
                ok_to_write.signal
            else
                ok_to_read.signal
            end
        end
```

preserve invariant

preserve invariant

Readers-writers: write methods

```
write_entry
        do
            if num_writers > 0 or num_readers > 0 do
                ok_to_write.wait
            end
            num_writers := num_writers + 1
        end
write_exit
        do
            num_writers := num_writers - 1
            if ok_to_read.is_empty then
                ok_to_write.signal
            else
                ok_to_read.signal
            end
        end
```

gives readers priority

Readers-writers: starvation

- starvation-freedom ensured by:

  => checking on *ok_to_write.is_empty* in *read_entry*; and
  => checking on *ok_to_read.is_empty* in *write_exit*

- but in certain applications may be beneficial to give either readers or writers higher priority

  => e.g. if one wants to ensure reading with minimum delay

Next on the agenda

1. monitors and mutual exclusion ✓
2. condition synchronisation ✓
3. signalling disciplines ✓
4. applications of monitors ✓

Assessment of monitors

positives:

=> *structured approach* to synchronisation
=> *separation of concerns:* mutual exclusion for free; condition synchronisation via condition variables

negatives:

=> *performance concerns:* tradeoff between programmer support and performance
=> *signalling disciplines:* source of ambiguity
=> *nested monitor calls*: semantics of wait calls?