Monitors

Systemes Concurrents

TD 1

Simone Gasparini
simone.gasparini@toulouse-inp.fr

Outline

- The Monitors - reminder
- The sleeping barber problem
  - Methodology
  - Variant
- The Readers-Writers problem
  - Priority policies

The Monitors

Moniteur de Hoare, Brinch-Hansen (1973)

Idée de base

La synchronisation résulte du besoin de partager convenablement un objet entre plusieurs activités concurrentes

$\rightarrow$ un moniteur est une construction qui permet de définir et de contrôler le bon usage d'un objet partagé par un ensemble d'activités

Définition

Un moniteur = un module exportant des opérations (et éventuellement des constantes et des types).

- Contrainte :
  exécution des opérations du moniteur en exclusion mutuelle
- La synchronisation des opérations du moniteur est réalisée par des opérateurs internes au moniteur.

Un moniteur est passif : ce sont les activités utilisant le moniteur qui l'activent, en invoquant ses opérations.

The Monitors

Synchronisation : type *condition*

La synchronisation est définie *au sein du moniteur*, en utilisant des variables de type *condition*, internes au moniteur

Définition

Variables de type condition définies dans le moniteur.
Opérations possibles sur une variable de type condition C :

- C.wait() (C.attendre()) : bloque l'activité appelante en libérant l'accès exclusif au moniteur.
- C.signal() (C.signaler()) : si des activités sont bloquées sur C, en réveille une ; sinon, nop.

- Une file d'attente est associée à *chaque* variable condition
- condition $\approx$ événement
  $\rightarrow$ condition $\neq$ prédicat logique
- Autre opération sur les conditions :
  C.empty() : vrai si aucune activité n'est bloquée sur C.

The Monitors

Transfert du contrôle exclusif

Les opérations du moniteur s'exécutent en exclusion mutuelle.
→ Lors d'un réveil par `signal()`, qui obtient/garde l'accès exclusif ?

Priorité au signalé — Signal-and-wait

Lors du réveil par `signal()`,

- l'accès exclusif est transféré à l'activité réveillée (signalée) ;
- l'activité signaleuse est mise en attente de pouvoir ré-acquérir l'accès exclusif

Priorité au signaleur — Signal-and-continue

Lors du réveil par `signal()`,

- l'accès exclusif est conservé par l'activité réveilleuse ;
- l'activité réveillée (signalée) est mise en attente de pouvoir acquérir l'accès exclusif
  - soit dans une file globale spécifique
  - soit avec les activités entrantes

The Sleeping Barber problem

Configuration

- 1 Barber
- 1 Barber Chair
- A barber shop with N available chairs in the waiting room

Rules

- A client waits in the waiting room if there are chairs available, otherwise it waits outside.
- When the barber is done shaving, it signals the client
  - the client exits the barbershop
  - another client is allowed in the barbershop
- If there are no clients the barber sleep.
- If a client finds the barber sleeping he wakes him up and get on the barber chair.

Methodology

Etapes

1. Déterminer l'interface du moniteur
2. Énoncer informellement les prédicats d'acceptation de chaque opération
3. Déduire les variables d'état qui permettent d'écrire ces prédicats d'acceptation
4. Formuler l'invariant du moniteur et les prédicats d'acceptation
5. Associer à chaque prédicat d'acceptation une variable condition qui permettra d'attendre/signaler la validité du prédicat
6. Programmer les opérations, en suivant le schéma précédent
7. Vérifier que
   - l'invariant est vrai à chaque transfert du contrôle du moniteur
   - le prédicat d'acceptation est vrai quand un réveil a lieu

(1) Interface definition

```
Client()
{
    loop
        EnterBarbershop()
        // wait in the waiting room
        SitOnBarberChair()
        // wait end of shaving
        GetUpAndGetOut()
    endloop
}
```

```
Barber()
{
    loop
        StartShaving()
        // shave client
        ...
        EndShaving()
    endloop
}
```

(2) Acceptance conditions

`EnterBarbershop()`	There must be a seat available in the waiting room
`SitOnBarberChair()`	The barber chair must be free
`GetUpAndGetOut()`	The client is shaved (no more beard)
`StartShaving()`	There is a bearded client on the barber chair
`EndShaving()`	nothing

(3) State variables

They allow to write the acceptance predicates

`AvailableSeats : int := N`	Number of available seats in the waiting room
`BarberChairFree : bool := true`	Whether there is a client on the barber chair or not
`HasBeard : bool := false`	Whether the client currently on the chair still has his beard

(4) Invariants and acceptance predicates

Invariants:

`(0 ≤ AvailableSeats ≤ N) ∧ (HasBeard ⇒ ¬BarberChairFree)`

Operation	Condition variable
`EnterBarbershop()`	*There must be a seat available in the waiting room* `AvailableSeats > 0`
`SitOnBarberChair()`	*The barber chair must be free* `BarberChairFree`
`GetUpAndGetOut()`	*The client is shaved (no more beard)* `¬HasBeard`
`StartShaving()`	*There is a bearded client on the barber chair* `HasBeard`
`EndShaving()`	*nothing*

(5) Condition variables

- Associate a **condition variable** with each **acceptance predicate**
- They will be used to wait for/signal the validity of the predicate.

Acceptance predicate	**Condition variable**
`AvailableSeats > 0`	**`WaitingRoom`**
`BarberChairFree`	**`BarberChair`**
`¬HasBeard`	**`ShavedClient`**
`HasBeard`	**`BeardedClient`**

Recap

Operation	Acceptance predicate	Condition Variable
`EnterBarbershop()`	*There must be a seat available in the waiting room* `AvailableSeats > 0`	`WaitingRoom`
`SitOnBarberChair()`	*The barber chair must be free* `BarberChairFree`	`BarberChair`
`GetUpAndGetOut()`	*The client is shaved (no more beard)* `¬HasBeard`	`ShavedClient`
`StartShaving()`	*There is a bearded client on the barber chair* `HasBeard`	`BeardedClient`
`EndShaving()`	*nothing*	

(6) Programming

Client

```
EnterBarbershop()
    // There must be a seat available in the waiting room
    if ¬(AvailableSeats > 0) then
        WaitingRoom.wait()
    endif
    {AvailableSeats > 0}
    AvailableSeats--
    {0 ≤ AvailableSeats < N}
```

(6) Programming

Client

```
SitOnBarberChair()
    // The barber chair must be free
    if ¬(BarberChairFree) then
        BarberChair.wait()
    endif
    {BarberChairFree == true}
    AvailableSeats++
    {0 < AvailableSeats ≤ N}
    BarberChairFree := false
    HasBeard := true
    BeardedClient.signal()
    WaitingRoom.signal()
```

(6) Programming

Client

```
GetUpAndGetOut()
    // The client is shaved (no more beard)
    if ¬(¬HasBeard) then  // or simply: if (HasBeard)
        ShavedClient.wait()
    endif
    {¬HasBeard}
    BarberChairFree := true
    BarberChair.signal()
```

(6) Programming

Barber

```
StartShaving()
    // There is a bearded client on the  barber chair
    if ¬(HasBeard) then
        BeardedClient.wait()
    endif
    {HasBeard}
```

(6) Programming

Barber

```
EndShaving()
    HasBeard := false
    ShavedClient.signal()
```

(7) Checking

- For each condition variable, check that **each signal precondition** (state at the moment when the signal is called) implies **each wait postcondition** (in this case, the acceptance predicate).

(7) Checking- signal-and-wait

```
EnterBarbershop()
      // There must be a seat available
in the waiting room
      if ¬(AvailableSeats > 0) then
            WaitingRoom.wait()
      endif
      {AvailableSeats > 0}
      AvailableSeats--
      {0 ≤ AvailableSeats < N}
```

```
GetUpAndGetOut()
      // The client is shaved (no more beard)
      if (HasBeard) then
            ShavedClient.wait()
      endif
      {¬HasBeard}
      BarberChairFree := true
      BarberChair.signal()
```

```
SitOnBarberChair()
      // The barber chair must be free
      if ¬(BarberChairFree) then
            BarberChair.wait()
      endif
      {BarberChairFree == true}
      AvailableSeats++
      {0 < AvailableSeats ≤ N}
      BarberChairFree := false
      HasBeard := true
      BeardedClient.signal()
      WaitingRoom.signal()
```

```
StartShaving()
      // There is a bearded client on the  barber chair
      if ¬(HasBeard) then
            BeardedClient.wait()
      endif
      {HasBeard}
```

```
EndShaving()
      HasBeard := false
      ShavedClient.signal()
```

(7) Checking- signal-and-wait

```
EnterBarbershop()
      // There must be a seat available
in the waiting room
      if ¬(AvailableSeats > 0) then
            WaitingRoom.wait()
      endif
      {AvailableSeats > 0}
      AvailableSeats--
      {0 ≤ AvailableSeats < N}
```

```
GetUpAndGetOut()
      // The client is shaved (no more beard)
      if (HasBeard) then
            ShavedClient.wait()
      endif
      {¬HasBeard}
      BarberChairFree := true
      BarberChair.signal()
```

```
SitOnBarberChair()
      // The barber chair must be free
      if ¬(BarberChairFree) then
            BarberChair.wait()
      endif
      {BarberChairFree == true}
      AvailableSeats++
      {0 < AvailableSeats ≤ N}
      BarberChairFree := false
      HasBeard := true
      BeardedClient.signal()
      WaitingRoom.signal()
```

```
StartShaving()
      // There is a bearded client on the  barber chair
      if ¬(HasBeard) then
            BeardedClient.wait()
      endif
      {HasBeard}
```

```
EndShaving()
      HasBeard := false
      ShavedClient.signal()
```

(7) Checking- signal-and-wait

```
EnterBarbershop()
	// There must be a seat available in the waiting room
	if ¬(AvailableSeats > 0) then
		WaitingRoom.wait()
	endif
	{AvailableSeats > 0}
	AvailableSeats--
	{0 ≤ AvailableSeats < N}
```

```
GetUpAndGetOut()
	// The client is shaved (no more beard)
	if (HasBeard) then
		ShavedClient.wait()
	endif
	{¬HasBeard}
	BarberChairFree := true
	BarberChair.signal()
```

```
SitOnBarberChair()
	// The barber chair must be free
	if ¬(BarberChairFree) then
		BarberChair.wait()
	endif
	{BarberChairFree == true}
	AvailableSeats++
	{0 < AvailableSeats ≤ N}
	BarberChairFree := false
	HasBeard := true
	BeardedClient.signal()
	WaitingRoom.signal()
```

```
StartShaving()
	// There is a bearded client on the  barber chair
	if ¬(HasBeard) then
		BeardedClient.wait()
	endif
	{HasBeard}
```

```
EndShaving()
	HasBeard := false
	ShavedClient.signal()
```

(7) Checking- signal-and-wait

```
EnterBarbershop()
      // There must be a seat available
in the waiting room
      if ¬(AvailableSeats > 0) then
            WaitingRoom.wait()
      endif
      {AvailableSeats > 0}
      AvailableSeats--
      {0 ≤ AvailableSeats < N}
```

```
GetUpAndGetOut()
      // The client is shaved (no more beard)
      if (HasBeard) then
            ShavedClient.wait()
      endif
      {¬HasBeard}
      BarberChairFree := true
      BarberChair.signal()
```

```
SitOnBarberChair()
      // The barber chair must be free
      if ¬(BarberChairFree) then
            BarberChair.wait()
      endif
      {BarberChairFree == true}
      AvailableSeats++
      {0 < AvailableSeats ≤ N}
      BarberChairFree := false
      HasBeard := true
      BeardedClient.signal()
      WaitingRoom.signal()
```

```
StartShaving()
      // There is a bearded client on the  barber chair
      if ¬(HasBeard) then
            BeardedClient.wait()
      endif
      {HasBeard}
```

```
EndShaving()
      HasBeard := false
      ShavedClient.signal()
```

(7) Checking- signal-and-wait

```
EnterBarbershop()
      // There must be a seat available
in the waiting room
      if ¬(AvailableSeats > 0) then
            WaitingRoom.wait()
      endif
      {AvailableSeats > 0}
      AvailableSeats--
      {0 ≤ AvailableSeats < N}
```

```
GetUpAndGetOut()
      // The client is shaved (no more beard)
      if (HasBeard) then
            ShavedClient.wait()
      endif
      {¬HasBeard}
      BarberChairFree := true
      BarberChair.signal()
```

```
SitOnBarberChair()
      // The barber chair must be free
      if ¬(BarberChairFree) then
            BarberChair.wait()
      endif
      {BarberChairFree == true}
      AvailableSeats++
      {0 < AvailableSeats ≤ N}
      BarberChairFree := false
      HasBeard := true
      BeardedClient.signal()
      WaitingRoom.signal()
```

```
StartShaving()
      // There is a bearded client on the  barber chair
      if ¬(HasBeard) then
            BeardedClient.wait()
      endif
      {HasBeard}
```

```
EndShaving()
      HasBeard := false
      ShavedClient.signal()
```

(7) Checking - signal-and-continue

Configuration

- Client A is on the barber chair
- Client B is waiting for the chair (`BarberChair.wait()`)
- Client C is in the waiting room doing nothing

Scenario

- Barber calls `EndShaving()`
- Client A gets the exclusive access
- Client C tries to call `SitOnBarberChair()` and waits for exclusive access
- Client A signals `BarberChair.signal()`
  - Client B is now unblocked, i.e. he can ask for exclusive access
- Client A is done and give up exclusive access. Who gets it?
  - Client B? --> OK
  - Client C?
    - it gets to the barber chair and signals the barber (`BeardedClient.signal()`)
    - B can then get the access and continue with its `SitOnBarberChair()` but the barber chair is no longer free!

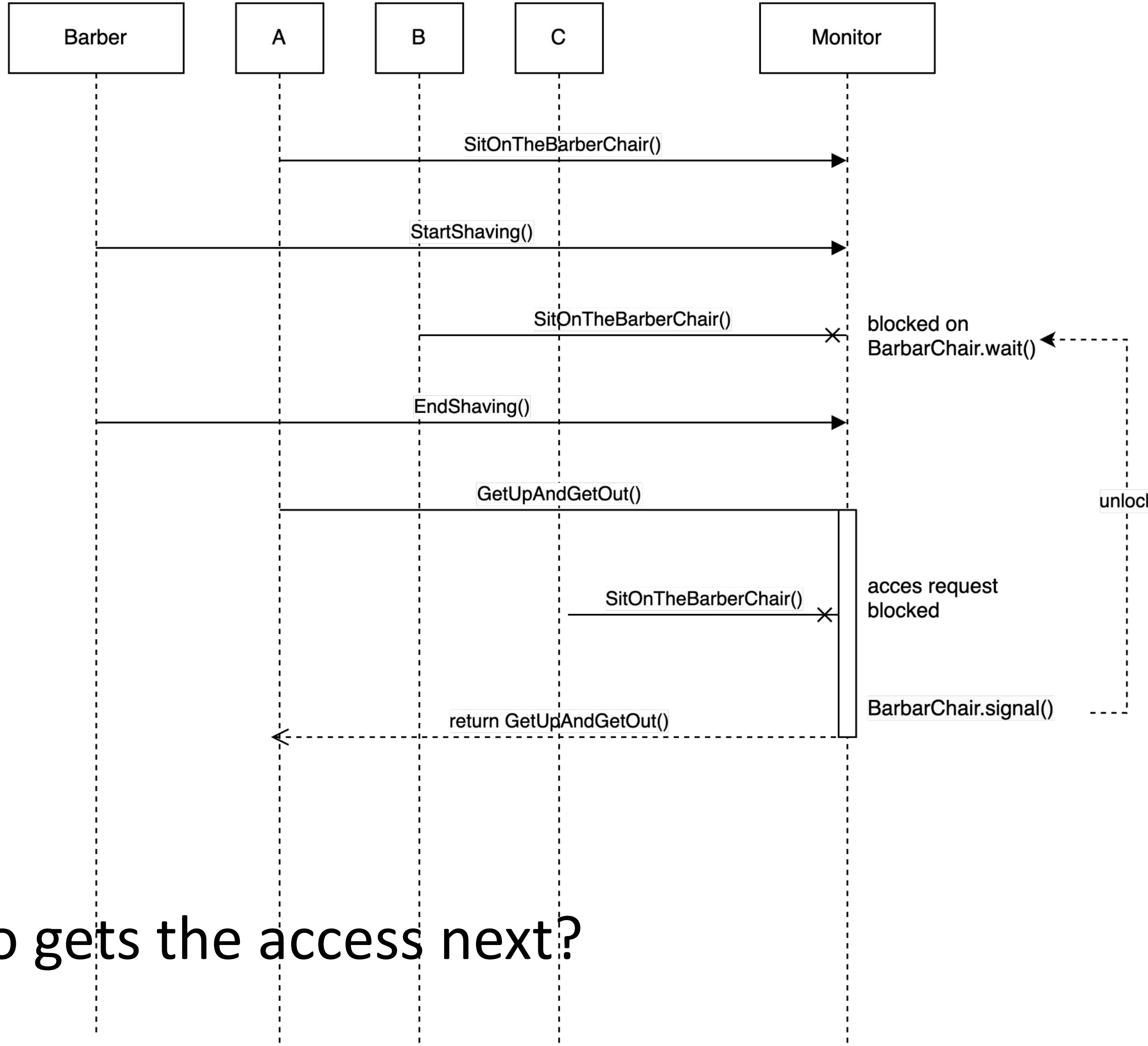
(7) Checking - signal-and-continue

Scenario

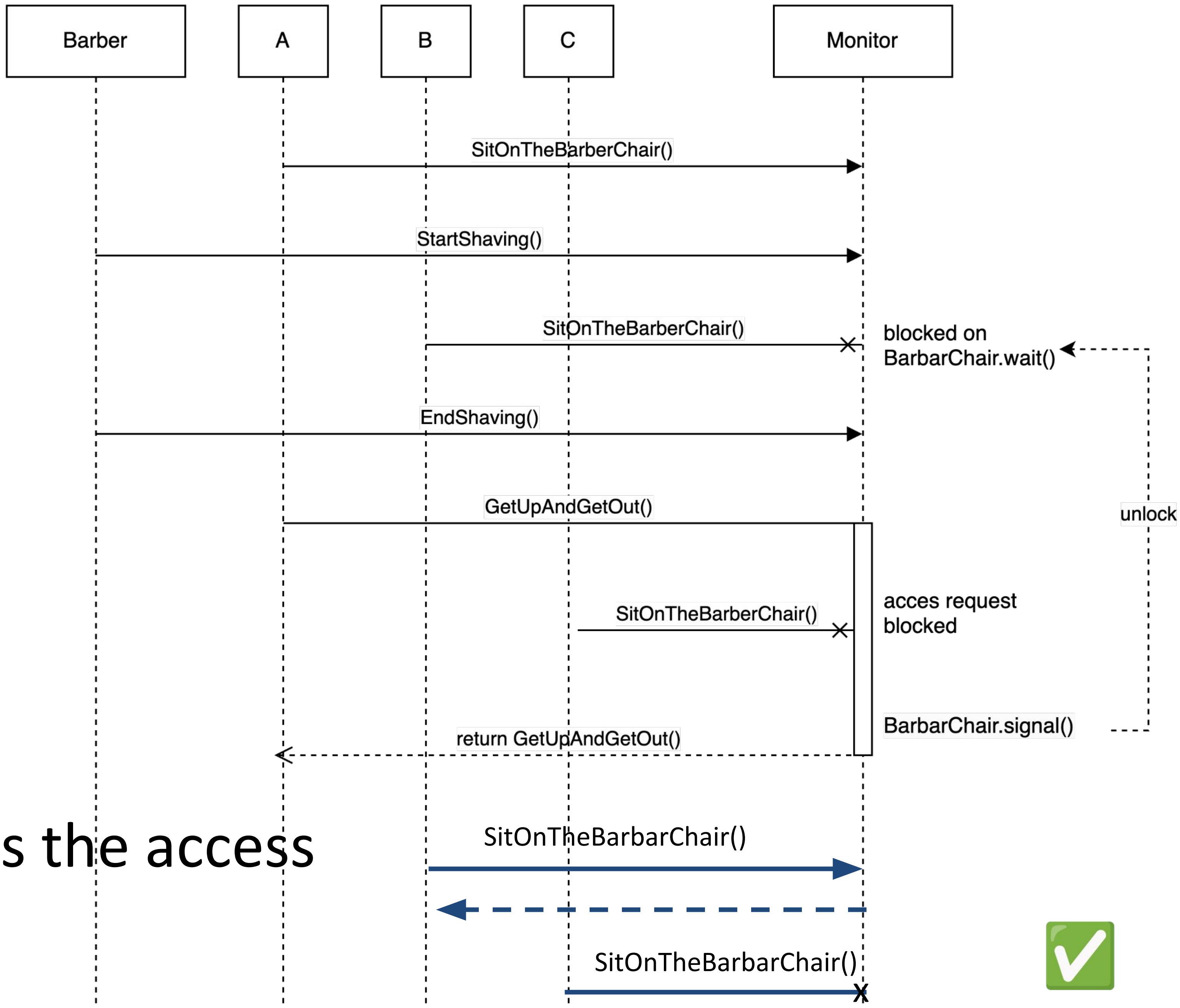
- Barber calls `EndShaving()`
- Client A gets the exclusive access
- Client C tries to call `SitOnBarberChair()` and waits for exclusive access
- Client A signals `BarberChair.signal()`
  - Client B is now unblocked, i.e. he can ask for exclusive access
- Client A is done and give up exclusive access. Who gets it?
  - Client B? --> OK
  - Client C?
    - it gets to the barber chair and signals the barber (`BeardedClient.signal()`)
    - B can then get the access and continue with its `SitOnBarberChair()` but the barber chair is no longer free!

Client needs to recheck the condition once he get the access

```
SitOnBarberChair()
      // The barber chair must be free
      while ¬(BarberChairFree)
            BarberChair.wait()
      endwhile
      {BarberChairFree == true}
      AvailableSeats++
      ....
```

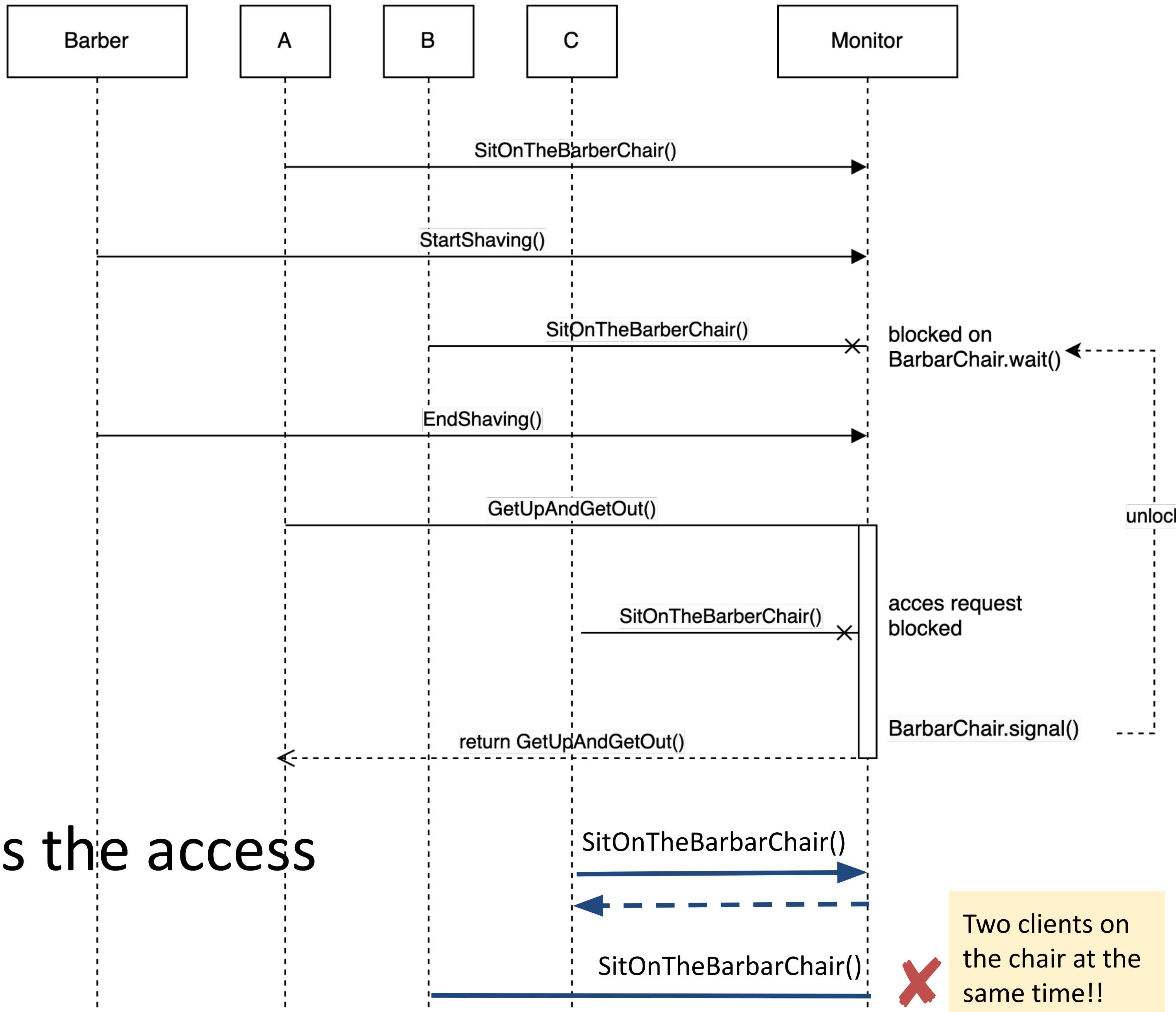


Who gets the access next?



B gets the access

when C gets the exclusive access, it gets blocked on BarberChair.wait()



C gets the access

return from BarberChair.wait() and the postcondition is false!

Variant

Configuration

- 1 Barber
- 1 Barber Chair
- A barber shop with N available chairs in the waiting room

Rules

- A client wait in the waiting room if there are chairs available, otherwise he waits outside.
- When the barber is done shaving, he signals the client
  - the client exit the barbershop
  - another client is allowed in the barbershop
- If there are no clients the barber sleep.
- If a client finds the barber sleeping he wakes him up and get on the barber chair.
- If the waiting room is full the client gives up and leave

Variant

- EnterBarbershop() need to return a boolean

```
bool EnterBarbershop()
    // There must be a seat available in the waiting room
    if ¬(AvailableSeats > 0) then
        return false
    endif
    {AvailableSeats > 0}
    AvailableSeats--
    {0 ≤ AvailableSeats < N}
    return true
```

Variant

- EnterBarbershop() need to return a boolean

```
Client()
{
    loop
        if EnterBarbershop()
            // wait in the waiting room
            SitOnBarberChair()
            // wait end of shaving
            GetUpAndGetOut()
        endif
    endloop
}
```

Readers–Writers problem

Configuration

- R readers
- W writers
- A shared file they can read or write.

Rules

- Multiple readers can read the file at the same time
- Only one writer can write at a given time
- If readers are reading no writer can write and vice versa

Policies

- Readers priority
  - no reader should wait if a reader has access to the object, while a writer waits till the last reader is done.
- Writers priority
  - No new readers allowed once a writer has asked for access, when a reader finishes it signals another writer in any, otherwise a reader

Readers–Writers problem - Writers priority

Configuration

- R readers
- W writers
- A shared file they can read or write.

Rules

- Multiple readers can read the file at the same time
- Only one writer can write at a given time
- If readers are reading no writer can write and vice versa

Policy

- Writers priority
  - No new readers allowed once a writer has asked for access, when a reader finishes it signals another writer in any, otherwise a reader

(1) Interface definition

```
Writer()
{
    loop
        AskToWrite()
        // write on the file
        ...
        EndWrite()
    endloop
}
```

```
Reader()
{
    loop
        AskToRead()
        // read file
        ...
        EndRead()
    endloop
}
```

(2) Acceptance conditions

`AskToWrite()`	No writer and no readers
`EndWrite()`	nothing
`AskToRead()`	No writer and no writer(s) waiting
`EndRead()`	nothing

(3) State variables

They allow to write the acceptance predicates

`NumWriters : int := 0`	Number of writers on the file
`NumWritersWaiting : int := 0`	Number of writers waiting to write
`NumReaders : int := 0`	Number of readers on the file

(4) Invariants and acceptance predicates

Invariants:

`(NumReaders = 0 ∨ NumWriters = 0) ∧ (NumWriters ≤ 1)`

`AskToWrite()`	*No writer and no readers* `(NumReaders = 0 ∧ NumWriters = 0)`
`AskToRead()`	*No writer and no writer(s) waiting* `(NumWriters = 0 ∧ NumWritersWaiting = 0)`

(5) Condition variables

- Associate a **condition variable** with each **acceptance predicate**
- They will be used to wait for/signal the validity of the predicate.

Operation	**Acceptance predicate**	**Condition variable**
`AskToWrite()`	`(NumReaders == 0 ∧ NumWriters == 0)`	**`CanWrite`**
`AskToRead()`	`(NumWriters == 0 ∧ NumWritersWaiting == 0)`	**`CanRead`**

(6) Programming

Writer

```
AskToWrite()
    // No writer and no readers
    if ¬(NumReaders = 0 ∧ NumWriters = 0) then
        NumWritersWaiting++
        CanWrite.wait()
        NumWritersWaiting--
    endif
    {NumReaders = 0 ∧ NumWriters = 0}
    NumWriters++
    {NumWriters = 1}
```

(6) Programming

Writer

```
EndWrite()
    {NumReaders = 0 ∧ NumWriters = 1}
    NumWriters--
    {NumReaders = 0 ∧ NumWriters = 0}
    // priority to the writers
    if (NumWritersWaiting > 0) then
        CanWrite.signal()
    else
        CanRead.signal()
    endif
```

(6) Programming

Reader

```
AskToRead()
    // No writer and no writer(s) waiting
    if ¬(NumWriters = 0 ∧ NumWritersWaiting = 0) then
        CanRead.wait()
    endif
    {NumWriters = 0 ∧ NumWritersWaiting = 0}
    NumReaders++
    {NumWriters = 0 ∧ NumReaders > 0 ∧ NumWritersWaiting =
    0}
    CanRead.signal()
```

(6) Programming

Reader

```
EndRead()
    {NumReaders > 0 ∧ NumWriters = 0}
    NumReaders--
    if (NumReaders = 0) then
        {NumReaders = 0 ∧ NumWriters = 0}
        CanWrite.signal()
    endif
```

(7) Checking

- For each condition variable, check that **each signal precondition** (state at the moment when the signal is called) implies **each wait postcondition** (in this case, the acceptance predicate).

(7) Checking - signal-and-wait

```
AskToWrite()
    // No writer and no readers
    if ¬(NumReaders = 0 ∧ NumWriters = 0) then
        NumWritersWaiting++
        CanWrite.wait()
        NumWritersWaiting--
    endif
    {NumReaders = 0 ∧ NumWriters = 0}
    NumWriters++
    {NumWriters = 1}
```

```
AskToRead()
    // No writer and no writer(s) waiting
    if ¬(NumWriters = 0 ∧ NumWritersWaiting = 0) then
        CanRead.wait()
    endif
    {NumWriters = 0 ∧ NumWritersWaiting = 0}
    NumReaders++
    {NumWriters = 0 ∧ NumReaders > 0}∧
    NumWritersWaiting = 0
    CanRead.signal()
```

```
EndWrite()
    {NumReaders = 0 ∧ NumWriters = 1}
    NumWriters--
    {NumReaders = 0 ∧ NumWriters = 0}
    // priority to the writers
    if (NumWritersWaiting > 0) then
        CanWrite.signal()
    else
        CanRead.signal()
    endif
```

```
EndRead()
    {NumReaders > 0 ∧ NumWriters = 0}
    NumReaders--
    if (NumReaders = 0) then
        {NumReaders = 0 ∧ NumWriters = 0}
        CanWrite.signal()
    endif
```

(7) Checking - signal-and-wait

```
AskToWrite()
    // No writer and no readers
    if ¬(NumReaders = 0 ∧ NumWriters = 0) then
        NumWritersWaiting++
        CanWrite.wait()
        NumWritersWaiting--
    endif
    {NumReaders = 0 ∧ NumWriters = 0}
    NumWriters++
    {NumWriters = 1}
```

```
AskToRead()
    // No writer and no writer(s) waiting
    if ¬(NumWriters = 0 ∧ NumWritersWaiting = 0) then
        CanRead.wait()
    endif
    {NumWriters = 0 ∧ NumWritersWaiting = 0}
    NumReaders++
    {NumWriters = 0 ∧ NumReaders > 0∧
    NumWritersWaiting = 0}
    CanRead.signal()
```

```
EndWrite()
    {NumReaders = 0 ∧ NumWriters = 1}
    NumWriters--
    {NumReaders = 0 ∧ NumWriters = 0}
    // priority to the writers
    if (NumWritersWaiting > 0) then
        CanWrite.signal()
    else
        CanRead.signal()
    endif
```

```
EndRead()
    {NumReaders > 0 ∧ NumWriters = 0}
    NumReaders--
    if (NumReaders == 0) then
        {NumReaders == 0 ∧ NumWriters = 0}
        CanWrite.signal()
    endif
```

(7) Checking - signal-and-wait

```
AskToWrite()
    // No writer and no readers
    if ¬(NumReaders = 0 ∧ NumWriters = 0) then
        NumWritersWaiting++
        CanWrite.wait()
        NumWritersWaiting--
    endif
    {NumReaders = 0 ∧ NumWriters = 0}
    NumWriters++
    {NumWriters = 1}
```

```
AskToRead()
    // No writer and no writer(s) waiting
    if ¬(NumWriters = 0 ∧ NumWritersWaiting = 0) then
        CanRead.wait()
    endif
    {NumWriters = 0 ∧ NumWritersWaiting = 0}
    NumReaders++
    {NumWriters = 0 ∧ NumReaders > 0∧
    NumWritersWaiting = 0}
    CanRead.signal()
```

```
EndWrite()
    {NumReaders = 0 ∧ NumWriters = 1}
    NumWriters--
    {NumReaders = 0 ∧ NumWriters = 0}
    // priority to the writers
    if (NumWritersWaiting > 0) then
        CanWrite.signal()
    else
        CanRead.signal()
    endif
```

```
EndRead()
    {NumReaders > 0 ∧ NumWriters = 0}
    NumReaders--
    if (NumReaders == 0) then
        {NumReaders == 0 ∧ NumWriters = 0}
        CanWrite.signal()
    endif
```

(7) Checking - signal-and-continue

Configuration

- There is a writer writing, no writers waiting and
- At least one reader waiting (`CanRead.wait()`)

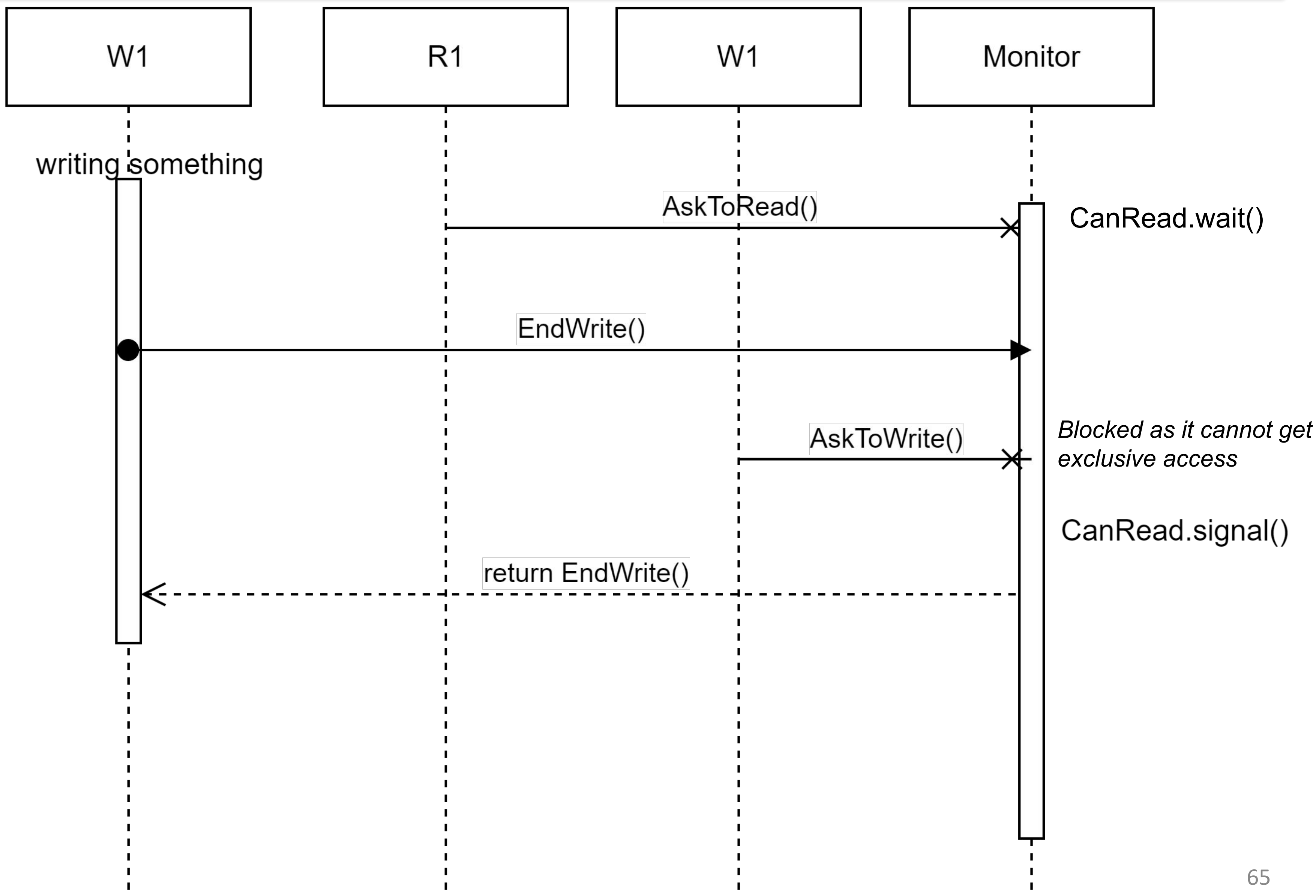
Scenario

- The writer is about to end (`EndWriting()`)
- The writer signals `CanRead.signal()`
  - Waiting reader(s) are now unblocked, i.e. they can ask for exclusive access
- Before the writer finishes a new writer arrives and asks for exclusive access for `AskToWrite()`.

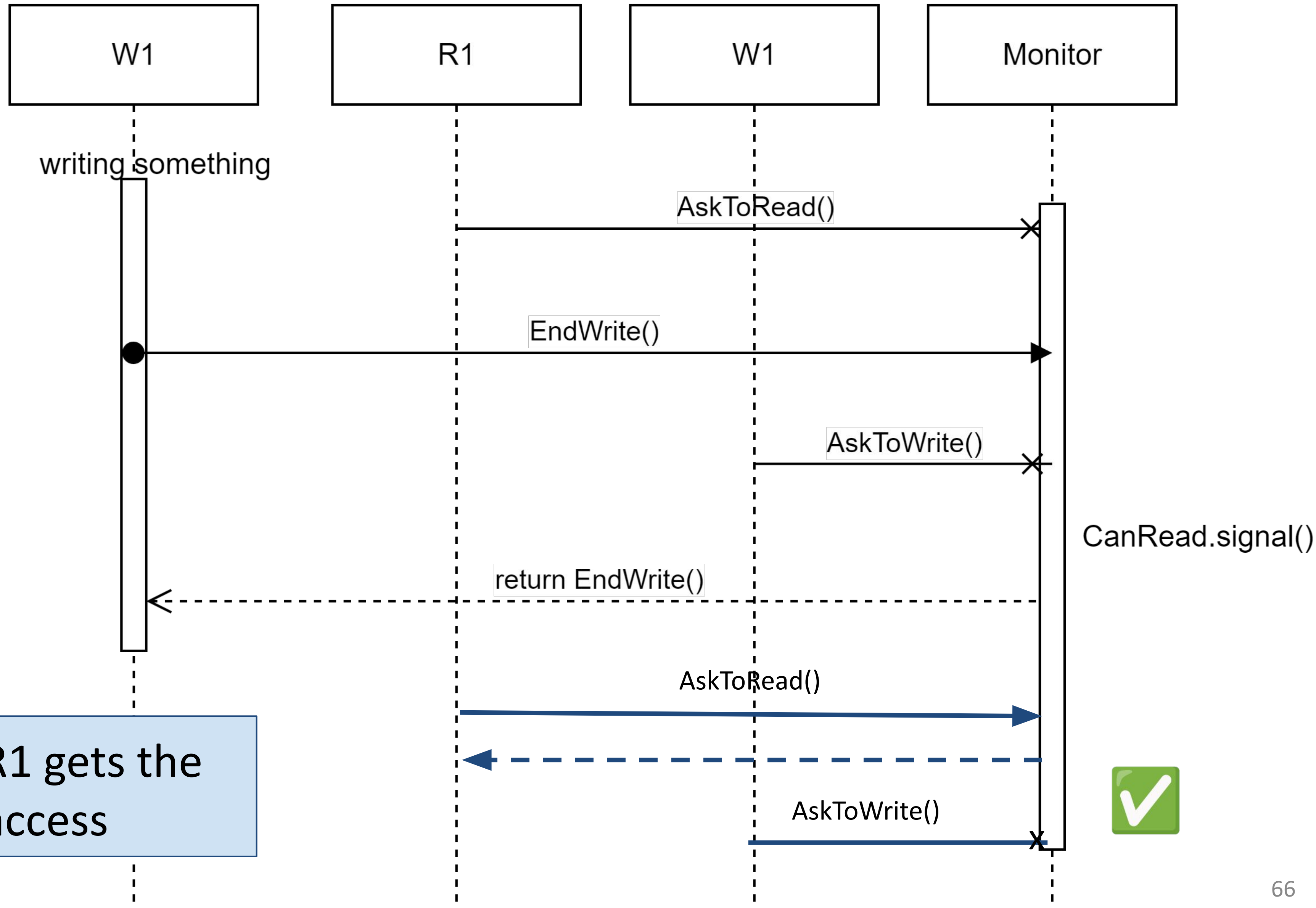
Who gets the access next?

- Reader? --> ✅ OK he will wake all the others and the writer will eventually wait
- New Writer? ✘
  - the acceptance condition for `AskToWrite()` is satisfied and starts writing.
  - One of the readers will get access and start reading!

(7) Checking - signal-and-continue

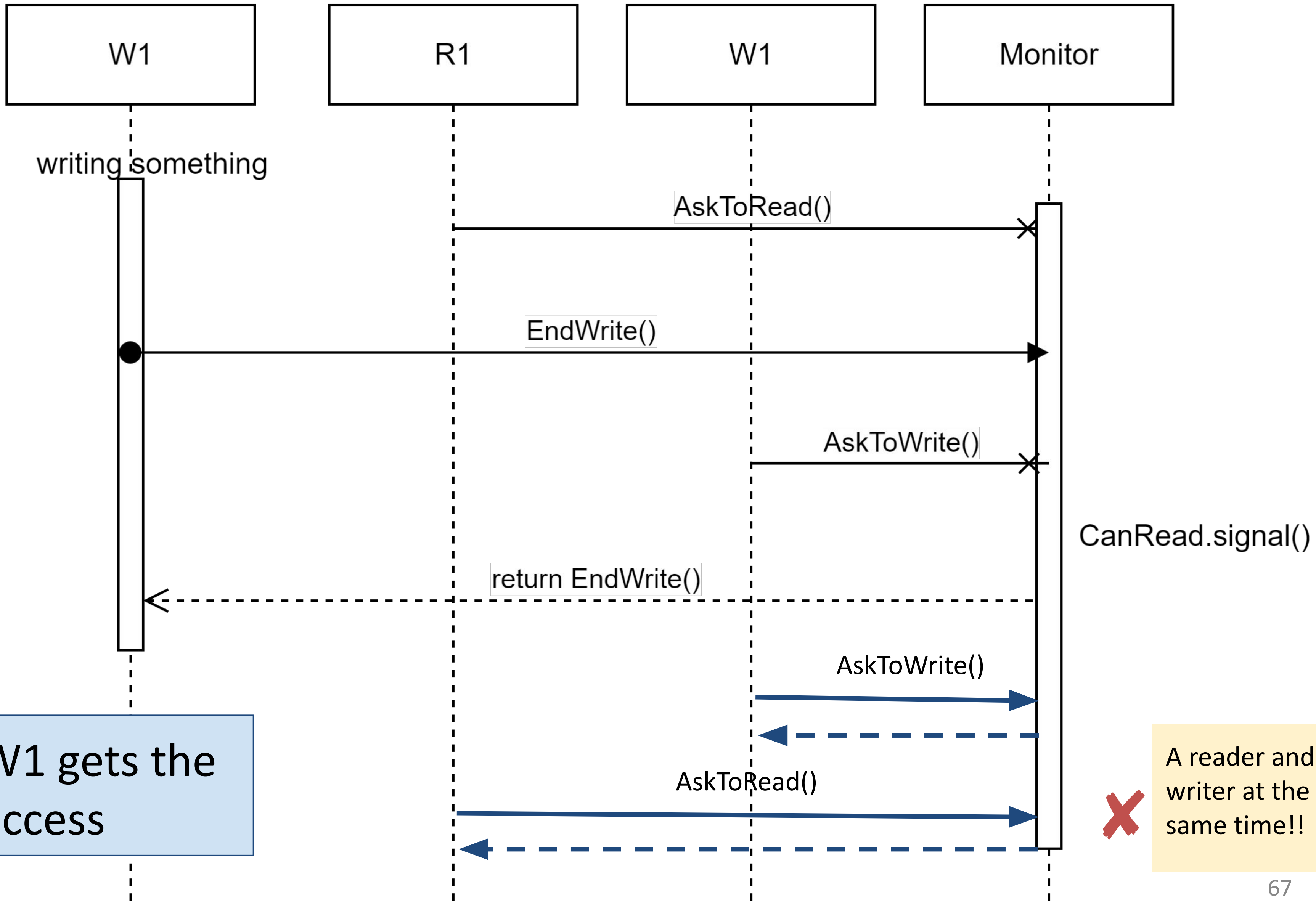


(7) Checking - signal-and-continue



R1 gets the access

(7) Checking - signal-and-continue



W1 gets the access

A reader and a writer at the same time!!

(7) Checking - signal-and-continue

Scenario

- The writer is about to end (`EndWriting()`)
- The writer signals `CanRead.signal()`
  - Waiting reader(s) are now unblocked, i.e. they can ask for exclusive access
- Before the writer finishes a new writer arrives and asks for exclusive access for `AskToWrite()`.

Who gets the access next?

- Reader? --> ✅ OK he will wake all the others and the writer will eventually wait
- New Writer? ✘
  - the acceptance condition for `AskToWrite()` is satisfied and starts writing.
  - One of the readers will get access and start reading!

It needs to check the condition again

```
AskToRead()
      // No writer and no writer(s) waiting
      while ¬(NumWriters = 0 ∧ NumWritersWaiting = 0)
            CanRead.wait()
      endwhile
      {NumWriters = 0 ∧ NumWritersWaiting = 0}
      NumReaders++
      {NumWriters = 0 ∧ NumReaders > 0}
      CanRead.signal()
```

(7) Checking - signal-and-continue

Scenario

- The same for the **writers**, there might be 2 writers at the same time!
- A writer W1 is writing
- Another W2 is waiting
- As the W1 ends it signals the waiting writer W2
  - W2 is now unblocked and can ask for exclusive access
- Meanwhile a new writer W3 calls `AskToWrite()`
- Who gets the acces after W1 finishes?
  - W2? OK, it will start writing and W3 will eventually wait on CanWrite
  - W3? W3 starts writing and at some point W2 will too when he gets the access

```
AskToWrite()
    // No writer and no readers
    while ¬(NumReaders = 0 ∧ NumWriters = 0)
        NumWritersWaiting++
        CanWrite.wait()
        NumWritersWaiting--
    endwhile
    {NumReaders = 0 ∧ NumWriters = 0}
    NumWriters++
    {NumWriters = 1}
```