Monitors

Systemes Concurrents

TD 2

Simone Gasparini
simone.gasparini@toulouse-inp.fr

Outline

- The Readers-Writers problem
  - FIFO policy
- Resource Allocation
  - FIFO policy
  - Short-Job-First policy (petit demandeurs)
  - Best-fit

Readers–Writers problem - FIFO

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

- FIFO
  - Respect the order of arrival but still allow multiple readers at the same time
    - e.g. R1 R2 R3 W R4 R5 --> first 3 Ls pass, W and R4 and R5 wait

(1) Interface definition

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

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

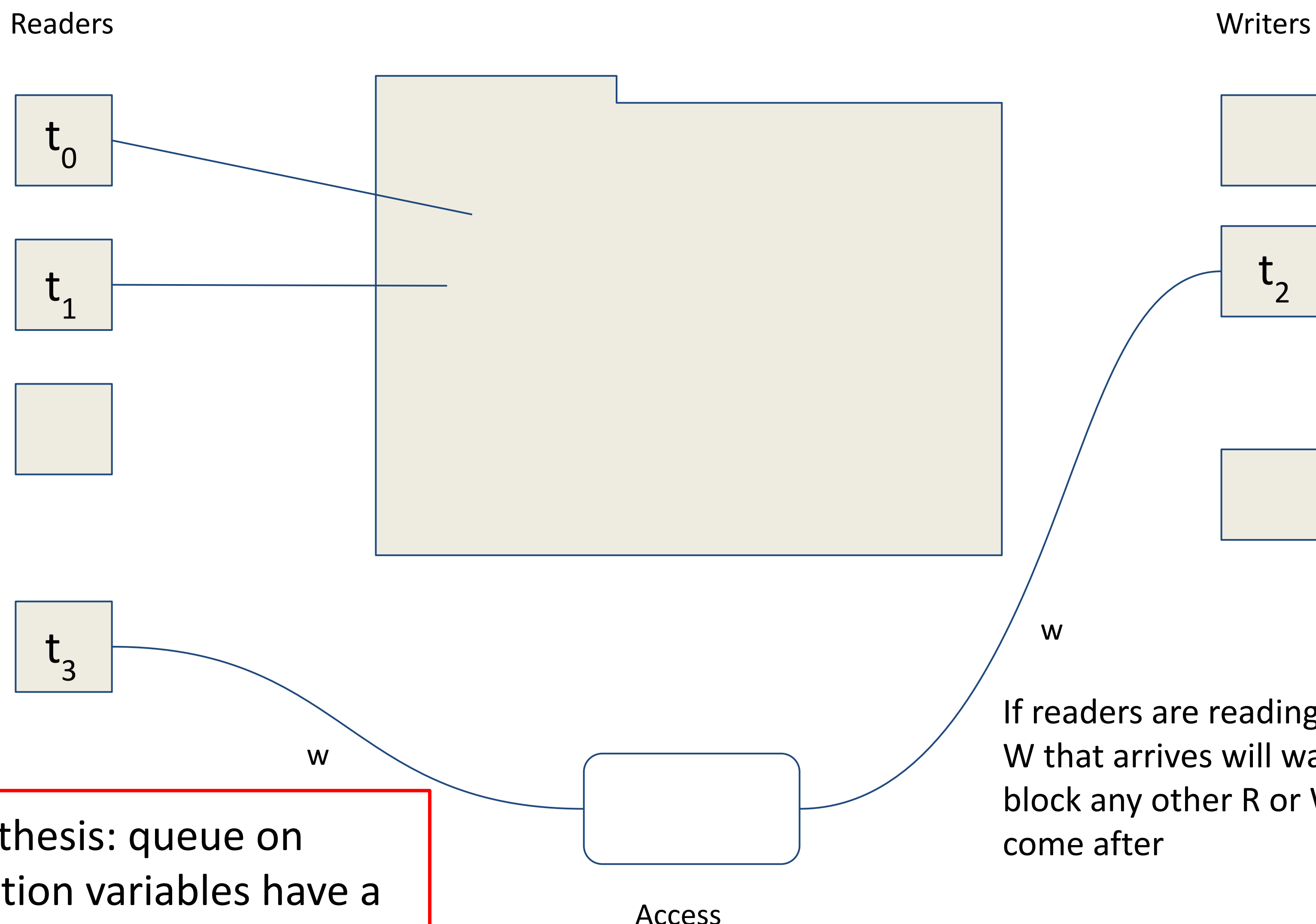
(2) Acceptance conditions

Let's start with a basic implementation from Writer Priority but without priority
Spoiler alert: it does not work!

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

Using a single condition variable

Idea: make wait any W or R that cannot do its operation

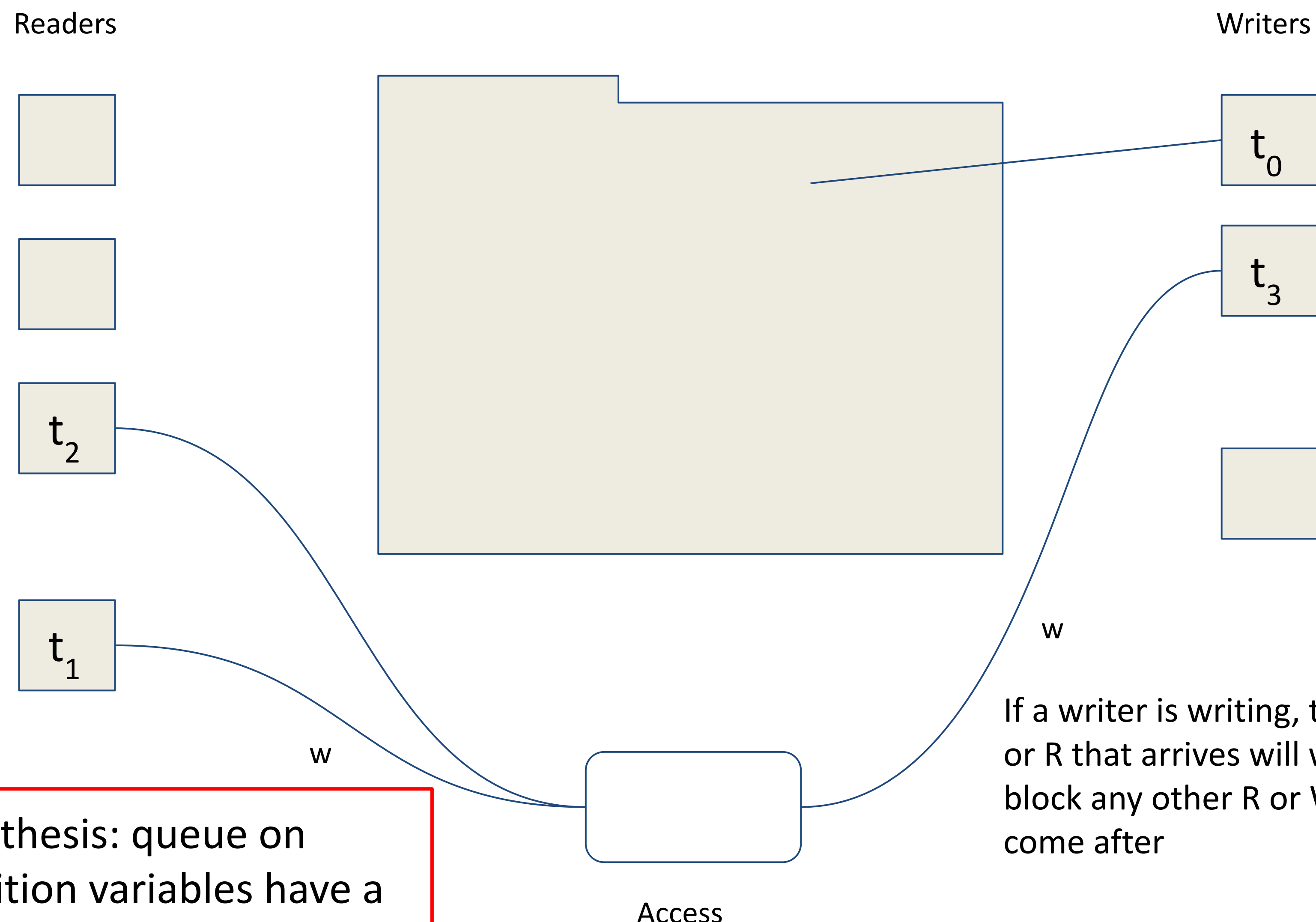


If readers are reading, the first W that arrives will wait and block any other R or W that come after

Hypothesis: queue on condition variables have a FIFO policy

Using a single condition variable

Idea: make wait any W or R that cannot do its operation



If a writer is writing, the first W or R that arrives will wait and block any other R or W that come after

Hypothesis: queue on condition variables have a FIFO policy

(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()	(Access.empty ∧ NumWriters == 0)	**Access**
AskToRead()	(NumWriters == 0 ∧ NumReaders == 0)	**Access**

(6) Programming

Writer

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

(6) Programming

Writer

```
EndWrite()
    {NumReaders = 0 ∧ NumWriters = 1}
    NumWriters--
    {NumReaders = 0 ∧ NumWriters = 0}
    // wake up whoever is waiting
    Access.signal()
```

(6) Programming

Reader

```
AskToRead()
    // No writer and no writer(s) waiting
    if ¬(NumWriters = 0 ∧ Access.empty) then
        Access.wait()
    endif
    {NumWriters = 0}
    NumReaders++
    {NumWriters = 0 ∧ NumReaders > 0}
    // chained wake up
    Access.signal()
```

(6) Programming

Reader

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

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

```
AskToRead()
    // No writer and no writer(s) waiting
    if ¬(NumWriters = 0 ∧ Access.empty) then
        Access.wait()
    endif
    {NumWriters = 0}
    NumReaders++
    {NumWriters = 0 ∧ NumReaders > 0}
    // chained wake up
    Access.signal()
```

```
EndWrite()
    {NumReaders = 0 ∧ NumWriters = 1}
    NumWriters--
    {NumReaders = 0 ∧ NumWriters = 0}
    // wake up whoever is waiting
    Access.signal()
```

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

(7) Checking

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

```
EndWrite()
    {NumReaders = 0 ∧ NumWriters = 1}
    NumWriters--
    {NumReaders = 0 ∧ NumWriters = 0}
    // wake up whoever is waiting
    Access.signal()
```

```
AskToRead()
    // No writer and no writer(s) waiting
    if ¬(NumWriters = 0 ∧ Access.empty) then
        Access.wait()
    endif
    {NumWriters = 0}
    NumReaders++
    {NumWriters = 0 ∧ NumReaders > 0}
    // chained wake up
    Access.signal()
```

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

(7) Checking

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

```
EndWrite()
    {NumReaders = 0 ∧ NumWriters = 1}
    NumWriters--
    {NumReaders = 0 ∧ NumWriters = 0}
    // wake up whoever is waiting
    Access.signal()
```

```
AskToRead()
    // No writer and no writer(s) waiting
    if ¬(NumWriters = 0 ∧ Access.empty) then
        Access.wait()
    endif
    {NumWriters = 0}
    NumReaders++
    {NumWriters = 0 ∧ NumReaders > 0}
    // chained wake up
    Access.signal()
```

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

(7) Checking

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

```
EndWrite()
    {NumReaders = 0 ∧ NumWriters = 1}
    NumWriters--
    {NumReaders = 0 ∧ NumWriters = 0}
    // wake up whoever is waiting
    Access.signal()
```

```
AskToRead()
    // No writer and no writer(s) waiting
    if ¬(NumWriters = 0 ∧ Access.empty) then
        Access.wait()
    endif
    {NumWriters = 0}
    NumReaders++
    {NumWriters = 0 ∧ NumReaders > 0}
    // chained wake up
    Access.signal()
```

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

(7) Checking

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

```
EndWrite()
    {NumReaders = 0 ∧ NumWriters = 1}
    NumWriters--
    {NumReaders = 0 ∧ NumWriters = 0}
    // wake up whoever is waiting
    Access.signal()
```

```
AskToRead()
    // No writer and no writer(s) waiting
    if ¬(NumWriters = 0 ∧ Access.empty) then
        Access.wait()
    endif
    {NumWriters = 0}
    NumReaders++
    {NumWriters = 0 ∧ NumReaders > 0}
    // chained wake up
    Access.signal()
```

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

(7) Checking

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

```
EndWrite()
    {NumReaders = 0 ∧ NumWriters = 1}
    NumWriters--
    {NumReaders = 0 ∧ NumWriters = 0}
    // wake up whoever is waiting
    Access.signal()
```

```
AskToRead()
    // No writer and no writer(s) waiting
    if ¬(NumWriters = 0 ∧ Access.empty) then
        Access.wait()
    endif
    {NumWriters = 0}
    NumReaders++
    {NumWriters = 0 ∧ NumReaders > 0}
    // chained wake up
    Access.signal()
```

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

(7) Checking

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

```
EndWrite()
    {NumReaders = 0 ∧ NumWriters = 1}
    NumWriters--
    {NumReaders = 0 ∧ NumWriters = 0}
    // wake up whoever is waiting
    Access.signal()
```

```
AskToRead()
    // No writer and no writer(s) waiting
    if ¬(NumWriters = 0 ∧ Access.empty) then
        Access.wait()
    endif
    {NumWriters = 0}
    NumReaders++
    {NumWriters = 0 ∧ NumReaders > 0}
    // chained wake up
    Access.signal()
```

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

The problem with a single condition variable

- During a wake up chain, a reader can wake up a writer

1. AW1 --> OK
2. AR1 --> Access.wait()
3. AR2 --> Access.wait()
4. AR3 --> Access.wait()
5. AW2 --> Access.wait()
6. AR4--> Access.wait()
7. EW1--> Access.signal()

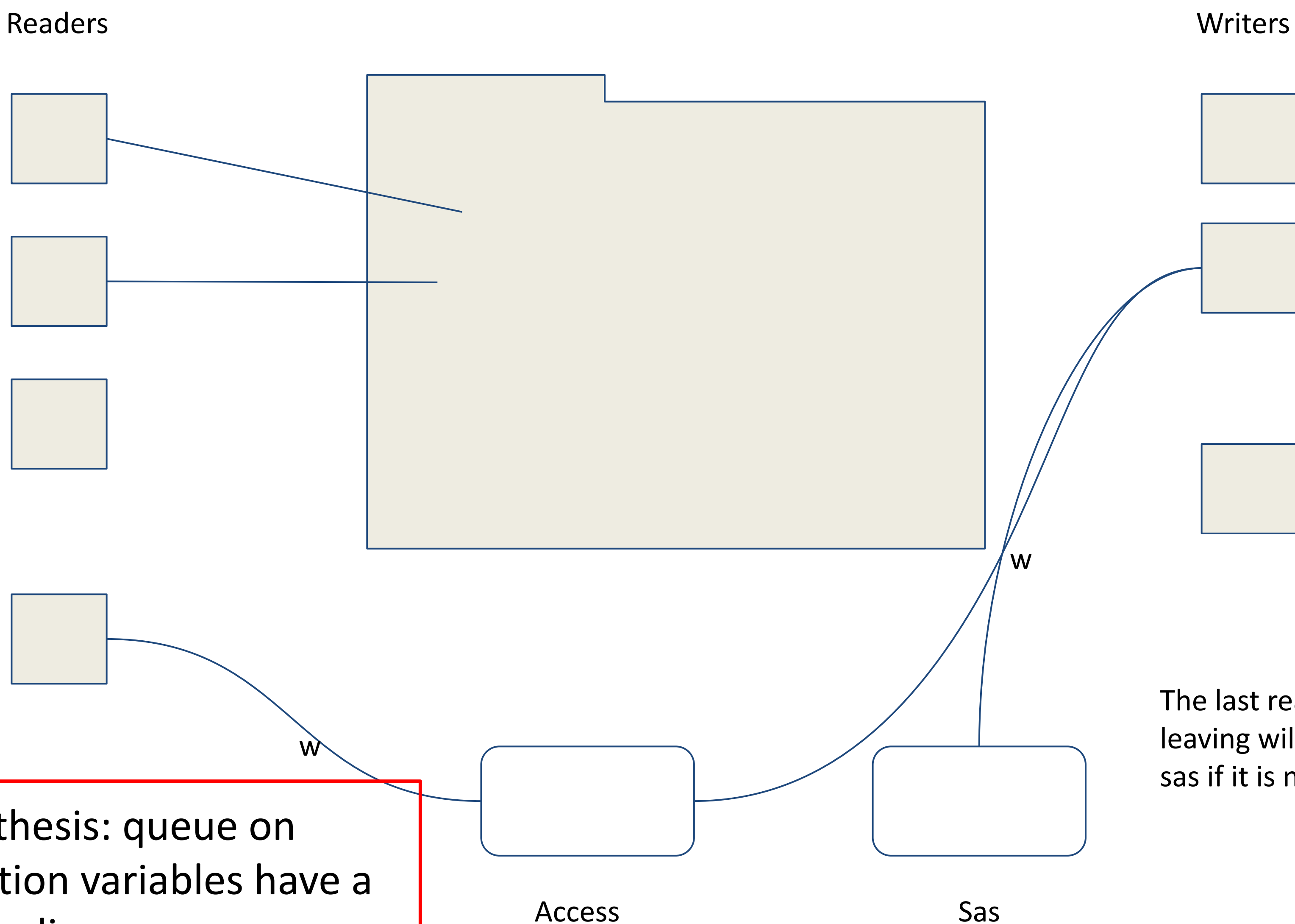
Access.signal()

Access.signal()

Access.signal()

- Need to detect the writer
  - wait again on Access? --> not fair
  - wait on a different condition variable --> Sas

Using an airlock (sas)



The last reader leaving will signal the sas if it is not empty

Hypothesis: queue on condition variables have a FIFO policy

(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()`	`(Access.empty ∧ NumWriters == 0 ∧ Sas.empty)`	**`Access`** **`Sas`**
`AskToRead()`	`(NumWriters == 0 ∧ NumReaders == 0)`	**`Access`** **`Sas`**

With a Sas

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

```
EndWrite()
    {NumReaders = 0 ∧ NumWriters = 1}
    NumWriters--
    {NumReaders = 0 ∧ NumWriters = 0}
    // wake up whoever is waiting
    Access.signal()
```

```
AskToRead()
    // No writer and no writer(s) waiting
    if ¬(NumWriters = 0 ∧ Access.empty ∧ Sas.empty ) then
        Access.wait()
    endif
    {NumWriters = 0}
    NumReaders++
    {NumWriters = 0 ∧ NumReaders > 0}
    // chained wake up
    Access.signal()
```

```
EndRead()
    {NumReaders > 0 ∧ NumWriters = 0}
    NumReaders--
    if (NumReaders == 0) then
        {NumReaders == 0 ∧ NumWriters = 0}
        if ¬(Sas.empty) then
            Sas.signal()
        else
            Access.signal()
        endif
    endif
```

Resource allocation problem

Configuration

- N resources (memory, computing time...)
- Each activity can ask for $k \leq N$ resources

Rules

- Each activity must have released the previously requested resources before asking for other resources.
- When an activity frees resources it wakes up a waiting activity (chained wake up)

Policies

- FIFO
  - Whenever a request cannot be satisfied block all the new requests until it is satisfied
- Short-Job-First policy (*petit demandeurs*)
  - Always wake up the activity waiting with the smallest request
- BestFit
  - Always wake up the activity waiting with largest request that can be satisfied

(1) Interface definition

- Just two operations
- **`request(q)`**
  - ask to reserve 1 ≤ q ≤ N resources
- **`free(q)`**
  - free the q resources previously reserved

(2) Acceptance conditions

`request(q)`	there are at least q resources available
`free(q)`	nothing

(3) State variables

They allow to write the acceptance predicates

`resAvailable : int := N`	Number of currently available resources

(4) Invariants and acceptance predicates

Invariants:

`(0 ≤ resAvailable ≤ N)`

Operation	Condition variable	Condition Variable
`request(q)`	there are at least q resources available `resAvailable ≥ q`	`EnoughResources`
`free(q)`	nothing	

Resource allocator - FIFO

- We need another condition variable for the activity that has the priority
  - `Access`
- We also need to keep track of the number of resources requested by this activity to wake it up once a free occurs
  - `pendingRequest` : int = 0
    - `0 ≤ pendingRequest ≤ N`

Hypothesis: queue on condition variables have a FIFO policy

Resource allocator - FIFO

```
request(q)
    // if there is already someone waiting, wait for your turn
    if (pendingRequest ≠ 0)
        Access.wait()
    {pendingRequest == 0}
    if ¬(resAvailable ≥ q)
        pendingRequest = q
        EnoughResources.wait()
        pendingRequest = 0
    {q ≤ resAvailable ≤ N ⋀ pendingRequest == 0}
    resAvailable = resAvailable - q
    {0 ≤ resAvailable ≤ N-q ⋀ pendingRequest == 0}
    Access.signal() // chained wake up
```

Resource allocator - FIFO

```
free(q)
    // free the resources
    resAvailable = resAvailable + q
    // if the are enough resources available to satisfy the
    // pending request signal the waiting activity
    if (resAvailable ≥ pendingRequest)
        {resAvailable ≥ pendingRequest}
        EnoughResources.signal()
```

Checking

```
free(q)
      // free the resources
      resAvailable = resAvailable + q
      // if the are enough resources available to satisfy the
      // pending request signal the waiting activity
      if (resAvailable ≥ pendingRequest)
            {resAvailable ≥ pendingRequest}
            EnoughResources.signal()
```

```
request(q)
      if (pendingRequest ≠ 0)
            Access.wait()
      {pendingRequest == 0}
      if ¬(resAvailable ≥ q)
            pendingRequest = q
            EnoughResources.wait()
            pendingRequest = 0
      {q ≤ resAvailable ≤ N ∧ pendingRequest == 0}
      resAvailable = resAvailable - q
      {0 ≤ resAvailable ≤ N-q ∧ pendingRequest == 0}
      Access.signal() // chained wake up
```

Checking

```
free(q)
    // free the resources
    resAvailable = resAvailable + q
    // if the are enough resources available to satisfy the
    // pending request signal the waiting activity
    if (resAvailable ≥ pendingRequest)
        {resAvailable ≥ pendingRequest}
        EnoughResources.signal()
```

```
request(q)
    if (pendingRequest ≠ 0)
        Access.wait()
    {pendingRequest == 0}
    if ¬(resAvailable ≥ q)
        pendingRequest = q
        EnoughResources.wait()
        pendingRequest = 0
    {q ≤ resAvailable ≤ N ∧ pendingRequest == 0}
    resAvailable = resAvailable - q
    {0 ≤ resAvailable ≤ N-q ∧ pendingRequest == 0}
    Access.signal() // chained wake up
```

Checking

```
free(q)
      // free the resources
      resAvailable = resAvailable + q
      // if the are enough resources available to satisfy the
      // pending request signal the waiting activity
      if (resAvailable ≥ pendingRequest)
            {resAvailable ≥ pendingRequest}
            EnoughResources.signal()
```

```
request(q)
      if (pendingRequest ≠ 0)
            Access.wait()
      {pendingRequest == 0}
      if ¬(resAvailable ≥ q)
            pendingRequest = q
            EnoughResources.wait()
            pendingRequest = 0
      {q ≤ resAvailable ≤ N ∧ pendingRequest == 0}
      resAvailable = resAvailable - q
      {0 ≤ resAvailable ≤ N-q ∧ pendingRequest == 0}
      Access.signal() // chained wake up
```

Resource allocator - Short-Jobs-First

- Always wake up the activity with the smallest request
- We need an array of condition variables, one for each possible request size
- `Access[N+1]`
  - N+1 to make life easier with indices, i.e. *we are allowing activity to ask* `request(0)`
- Whenever an activity frees resources, it wakes up a waiting activity starting from the ones asking less resources
- Each waken up activity will do the same (**chained wake up**)

Resource allocator - Short-Jobs-First

```
request(q)
    if ¬(resAvailable ≥ q)
        Access[q].wait()
    {q ≤ resAvailable ≤ N}
    resAvailable = resAvailable - q
    {0 ≤ resAvailable ≤ N-q}
    wakeUpNext() // chained wake up
```

Resource allocator - Short-Jobs-First

```
request(q)
    if ¬(resAvailable ≥ q)
        Access[q].wait()
    {q ≤ resAvailable ≤ N}
    resAvailable = resAvailable - q
    {0 ≤ resAvailable ≤ N-q}
    wakeUpNext() // chained wake up
```

```
wakeUpNext()
    i = 1 // because request(0) is useless
    while (i ≤ resAvailable ∧ Access[i].empty)
        ++i
    if i ≤ resAvailable
        Access[i].signal()
```

Resource allocator - Short-Jobs-First

```
free(q)
    // free the resources
    resAvailable = resAvailable + q
    wakeUpNext()
```

Resource allocator - Short-Jobs-First

Note

To optimize we can start to wake up activity at:

- `q` when `request()`
  - since the activity with q has been woken up, there should not be other activity requesting less than q waiting
- `resAvailable - q` when `free()`
  - same reason, before the activity asked to free resources there was resAvailable-q resources available, so no activity asking less than that should be waiting.

```
wakeUpNext(start)
    i = start
    while (i ≤ resAvailable ∧ Access[i].empty)
        ++i
    if i ≤ resAvailable
        Access[i].signal()
```

Optimization

```
request(q)
    if ¬(resAvailable ≥ q)
        Access[q].wait()
    {q ≤ resAvailable ≤ N}
    resAvailable = resAvailable - q
    {0 ≤ resAvailable ≤ N-q}
    wakeUpNext(q) // chained wake up
```

```
free(q)
    // free the resources
    resAvailable = resAvailable + q
    wakeUpNext(resAvailable - q)
```

Resource allocator - BestFit

- Always wake up the activity with the largest request that can be satisfied
- Same code as before, only change how the chained wake up works

```
wakeUpNext()
    i = resAvailable
    while (i > 0 ∧ Access[i].empty)
        --i
    if i > 0
        Access[i].signal()
```