G52OSC
OPERATING SYSTEMS AND CONCURRENCY

Readers and Writers

Dr Jason Atkin

This lecture

- Readers-Writers problems
- Fairness vs correctness

An example

- A simple library catalogue has two kinds of users:
  - *Borrowers* who use the catalogue to search for books or to find out whether a particular book is on loan (readers)
  - *Library staff* who update the catalogue when new books are added to the library stock or to record which books are on loan or returned (writers)
- If we allow all users unrestricted access to the catalogue, a borrower may see the catalogue when it is an inconsistent state, e.g.:
  - A borrower is looking for a book which has just arrived at the library and not all details have been added
  - A librarian is updating the catalogue to include the book, e.g., by copying and editing an existing entry
  - The borrower sees an inconsistent state of the catalogue, e.g., the entry contains an incorrect shelf number

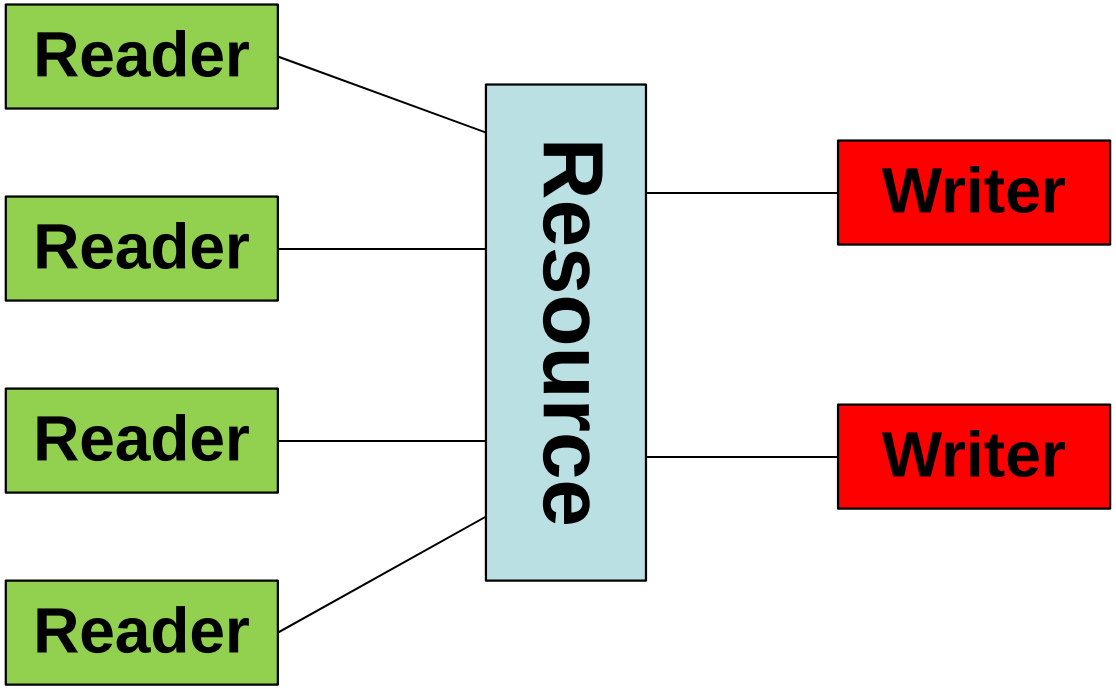
Correctness vs efficiency

- **Readers and Writers problem: special case** of general problem where processes may all ***both*** read **and** write:
  - Any solution to the general problem will also be a ***correct*** solution to the Readers and Writers problem
  - However (much) more ***efficient*** solutions are possible for the Readers and Writers problem
- In general, an efficient solution depends on the specifics of the problem – can we take advantage of structure?
- One solution to the readers and writers problem would be to make **all** accesses to the file **mutually exclusive**:
  - At most one process/thread (reader or writer) would be able to access the file at a time (e.g. binary semaphore, monitor, mutex)
  - **This is simple and correct** but, in general, the performance of such an approach is unacceptable (only one borrower/reader could search the catalogue at a time)

Synchronisation requirements

- To ensure correctness **and for maximum efficiency** (concurrency) and we require that:
  - If a writer is writing to the file:
    - No other writer may write to the file
    - No reader may read it
  - Any number of readers may simultaneously read the file (if no writers)

How could we do this?



Tools:

Monitor

Semaphore

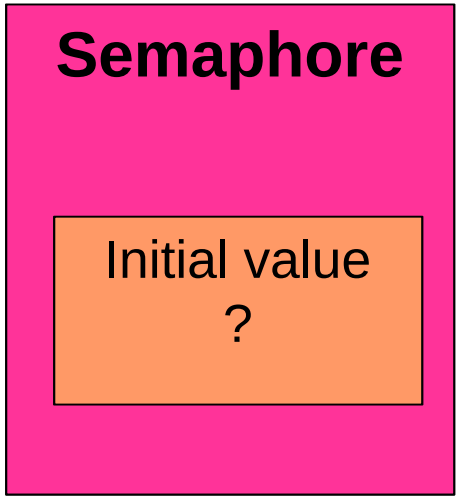
Mutex / Critical Section

Condition Variable / Event

- How could we build a protocol to enforce mutual exclusion for a single writer, without being inefficient?

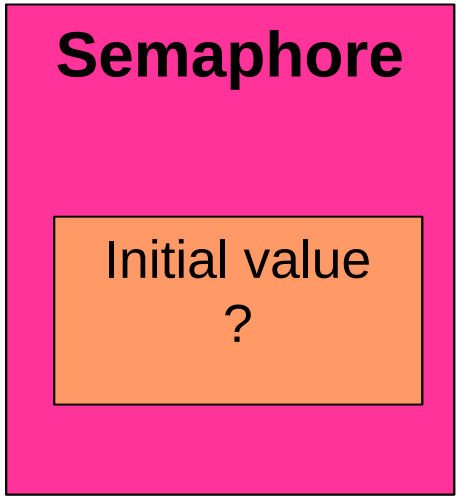
Semaphore

- A counter with a waiting mechanism
- p() wait() : ‘lock’ it
  - If count is zero then wait on the semaphore
  - Otherwise decrement count and continue
- v() notify() signal() :
  - If there are waiting processes, wake one up
  - Otherwise increment the count

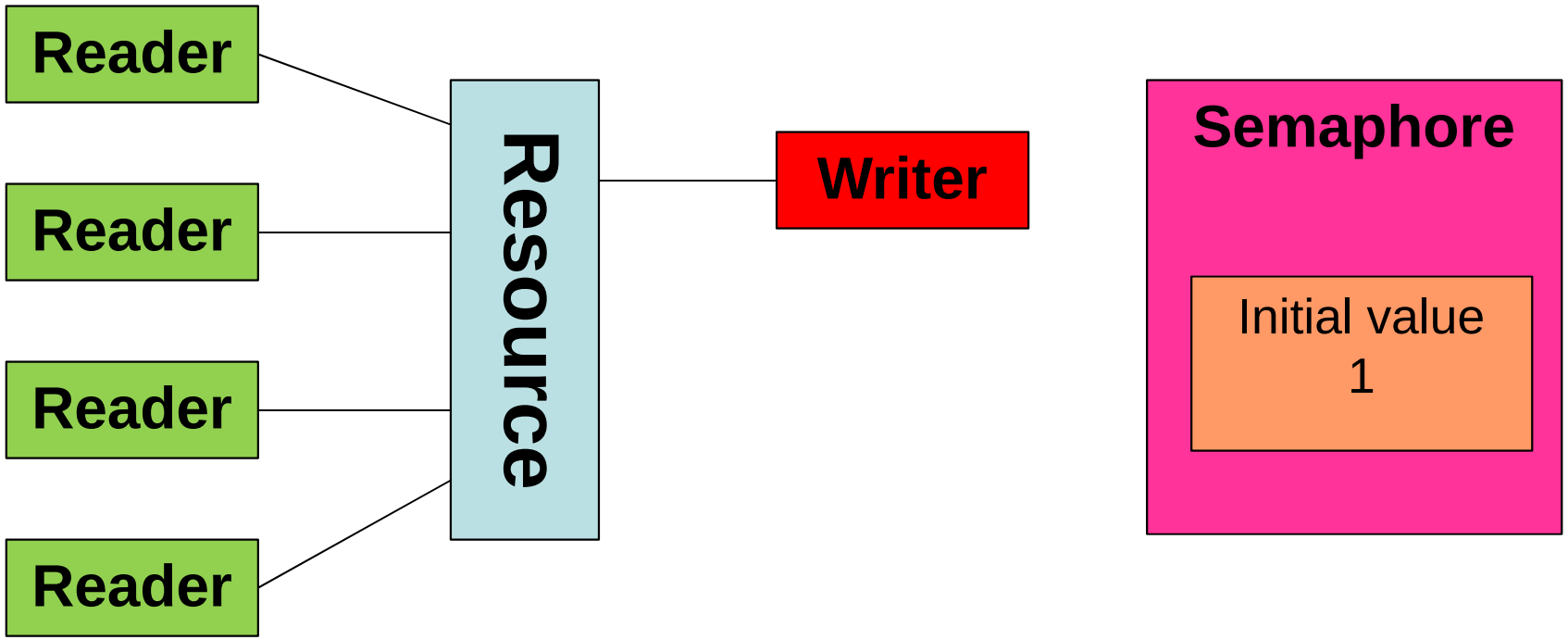


Equivalent Semaphore Implementation

- A counter with a waiting mechanism
- p() wait() : ‘lock’ it
  - Decrement count
  - If count is now negative
    then wait on the semaphore
- v() notify() signal() :
  - Increment semaphore value
  - If count was negative **before**
    the increment (i.e. a process was waiting)
    then wake up a waiting process

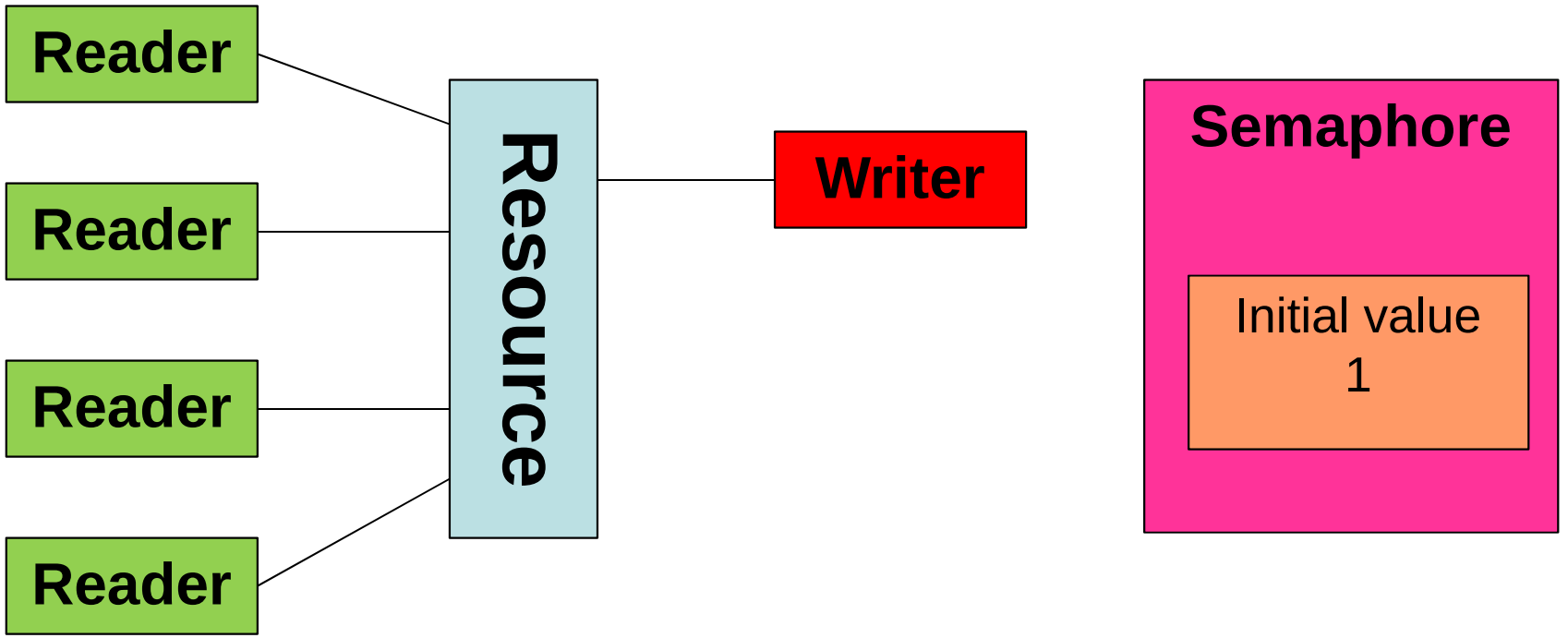


Single writer binary semaphore?



- Assume four readers and one writer
- What can we do with a BINARY semaphore?

Single writer binary semaphore

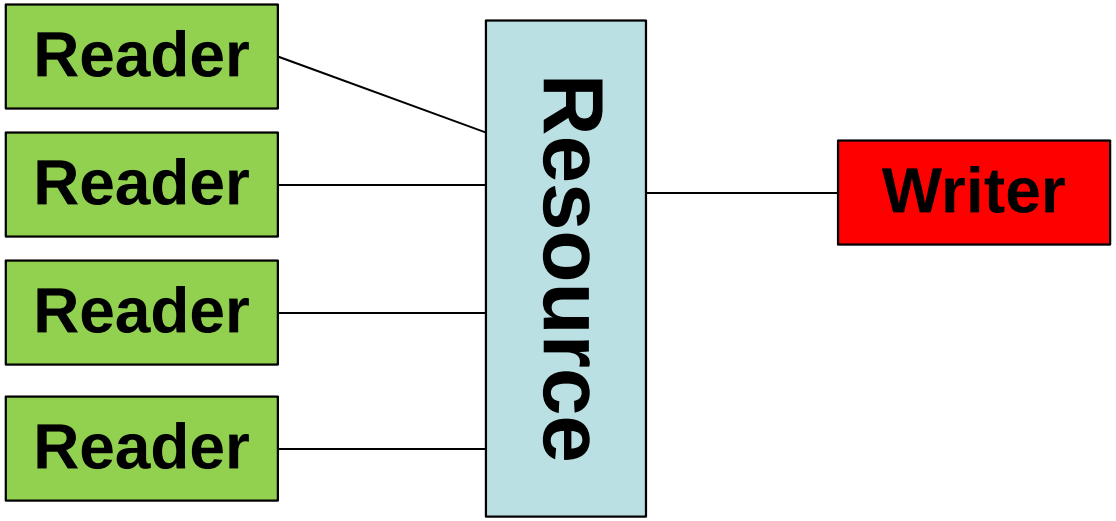


- Each thread does p(), uses it, then v()
- Mutual exclusion between everything - **inefficient**

Could we allow multiple readers?

Hint: using a second binary semaphore

Allowing multiple readers



Semaphore Lock

Initial value 1

Semaphore Readers

Initial value 1

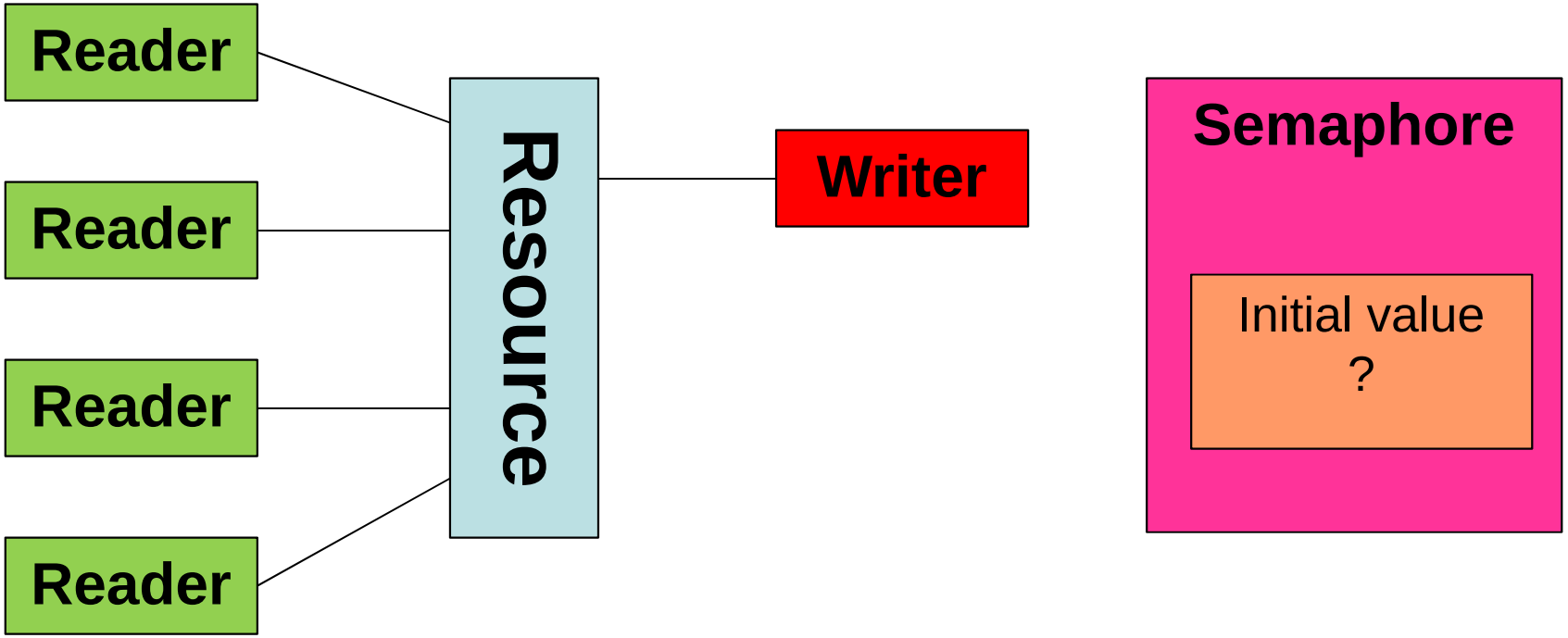
Readcount = 0

- **Writer:** `lock.p()` …. `lock.v()`
- **Reader:**

```
reader.p(), readcount++,
if (readcount==1) lock.p(),
reader.v()
…
reader.p(), readcount--,
if (readcount==0) lock.v(),
reader.v()
```

Using a generic semaphore

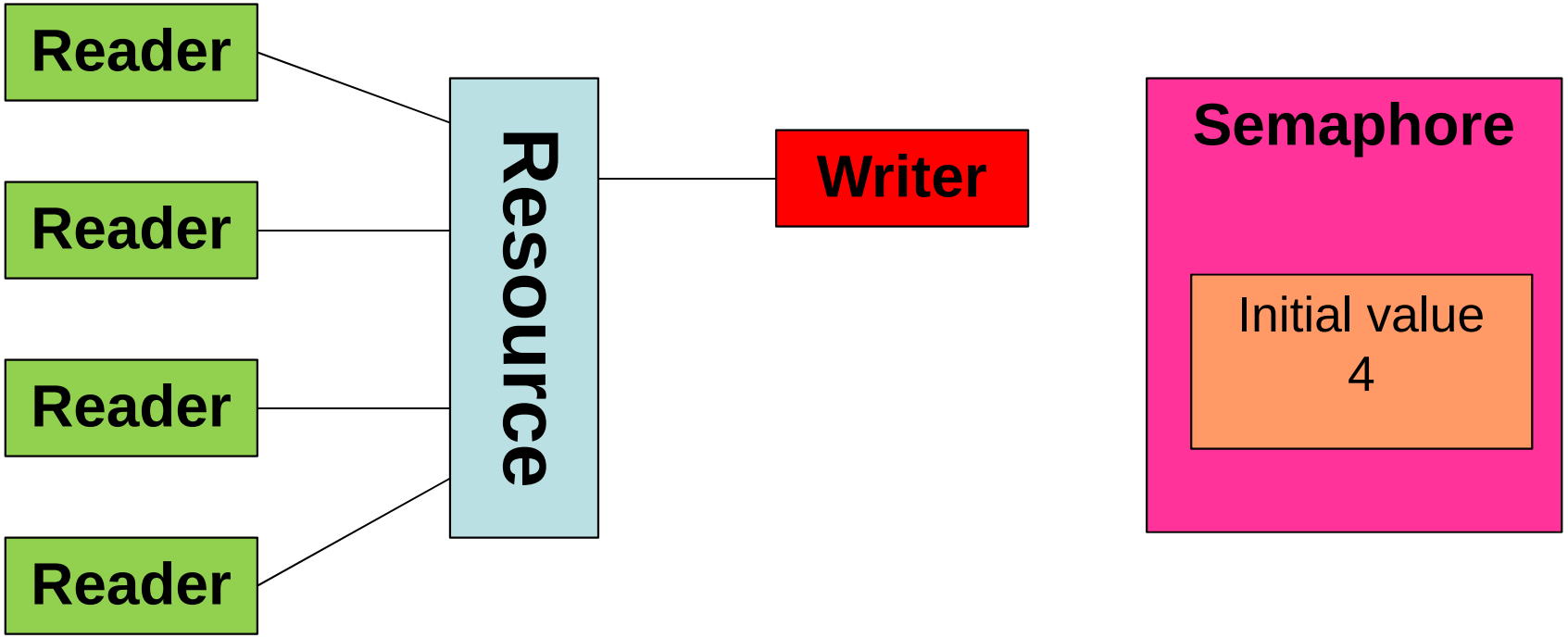
Single writer semaphore solution?



- Assume four readers and one writer
- What can we do with a generic semaphore?

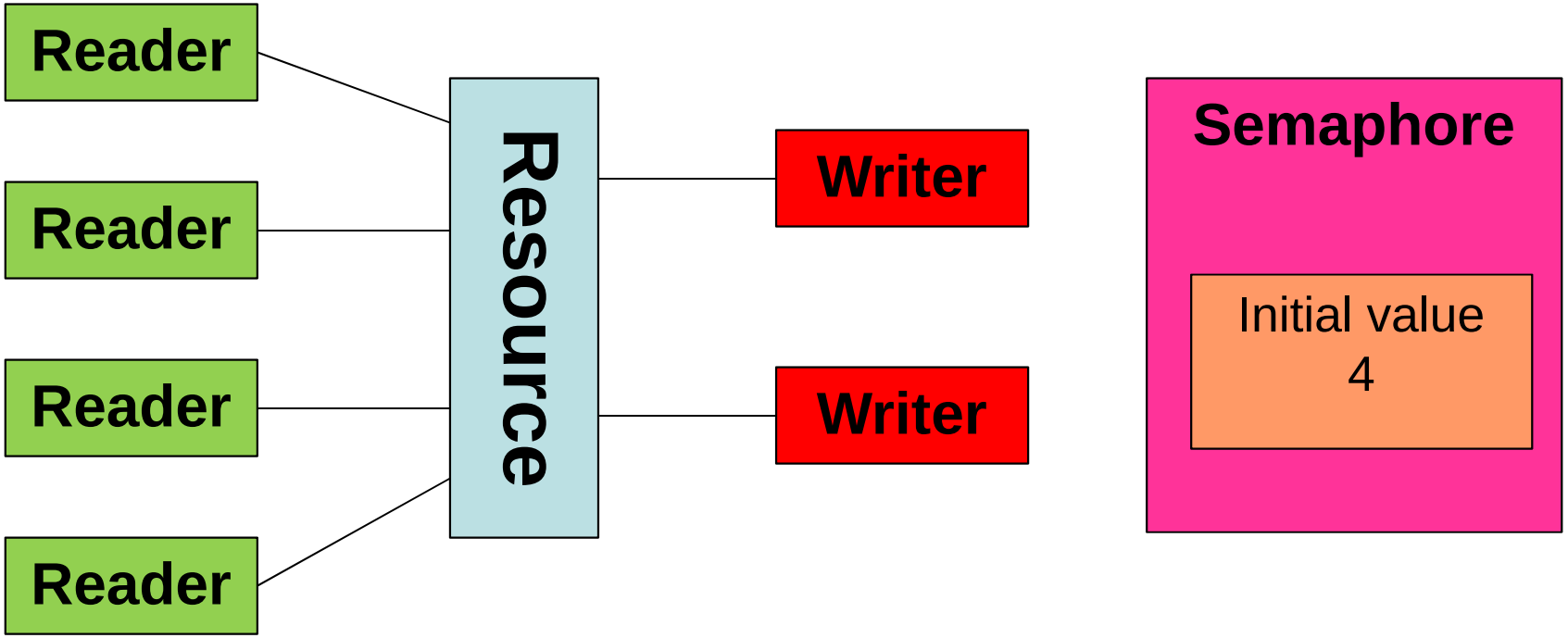
ANSWER

Simple solution



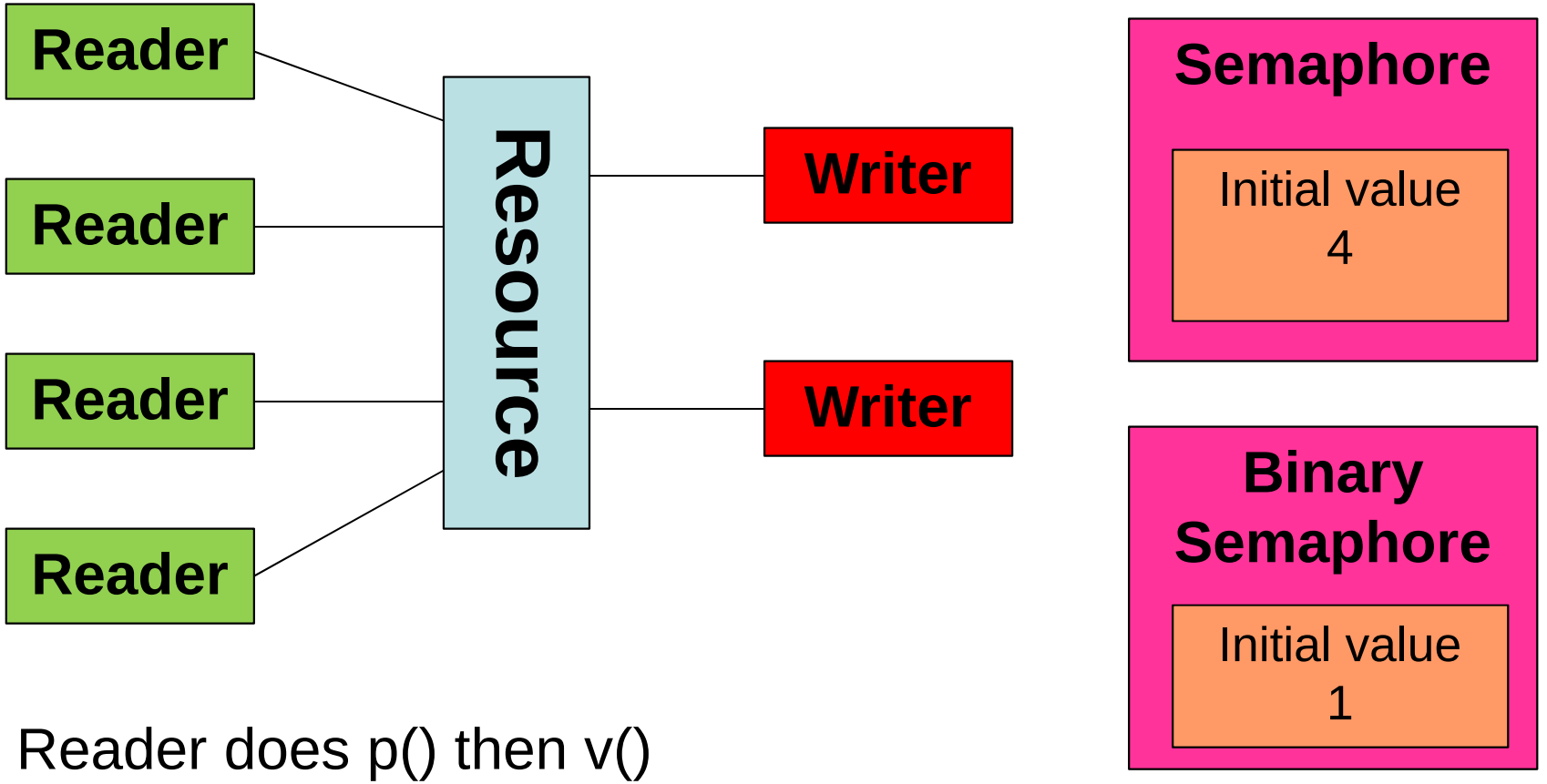
- Initialise semaphore to 4 (or n = #readers)
- Each reader does p() then v()
- Writer does p() 4 times, then v() 4 times

What if there were 2 writers?



- Initialise semaphore to 4 (or n)
- Reader does p() then v()
- Each writer does p() 4 times, then v() 4 times
- **Does it work? If not how could we fix it?**

Potential fix: lock for the writers?



- Reader does p() then v()
- Writer does p() on binary semaphore, then p() 4 times on original, then 4x v(), then v() on binary semaphore
- Any problem with this?

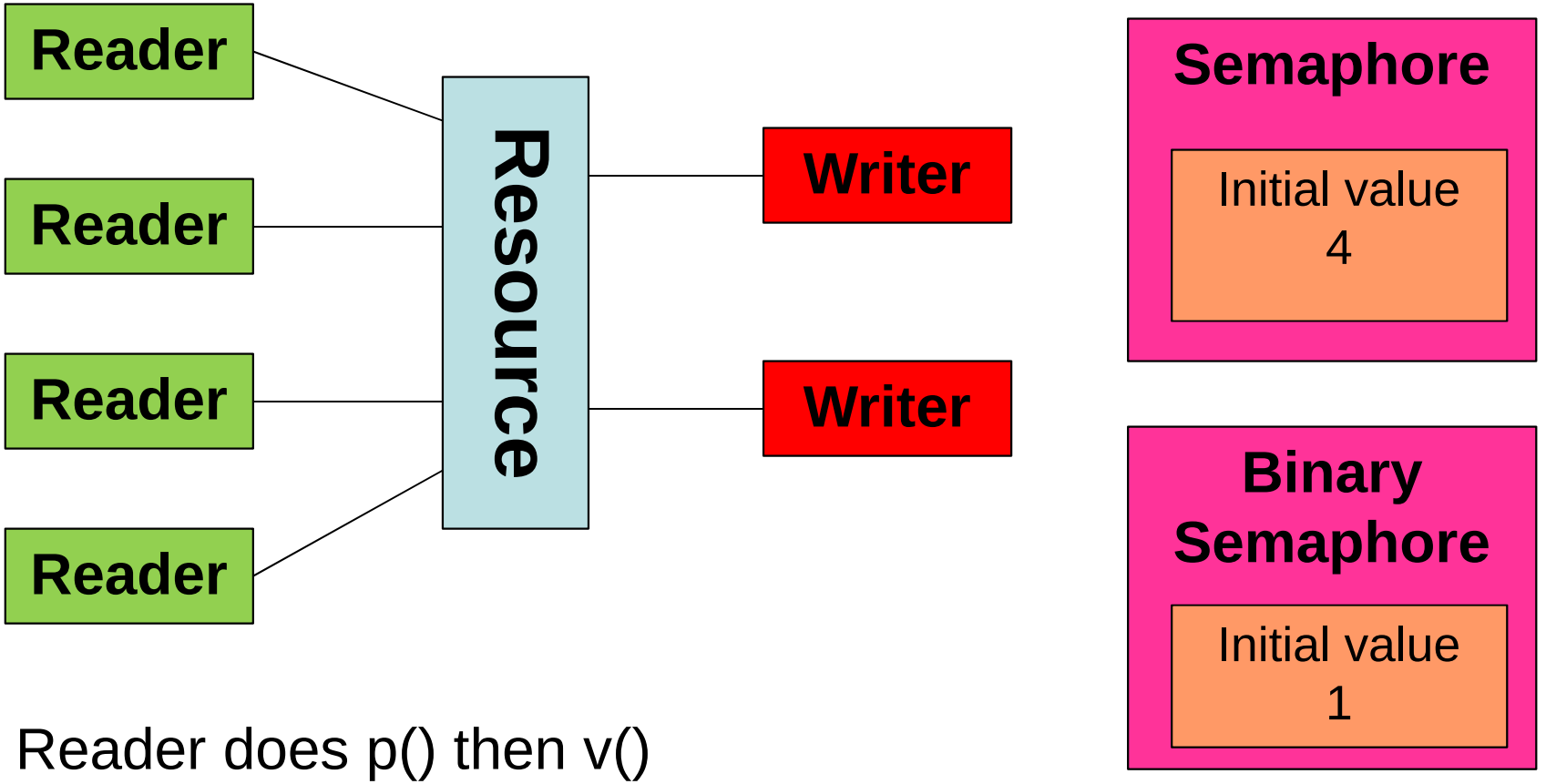
Consider various cases

- Like traces at the thread rather than operation level

- **W1** R1 R2 R3 R4 R5 R6 **W2**
- **W1 W2** R1 R2 R3 R4 R5 R6
- R1 R2 R3 **W1 W2** R4 R5 R6

- Assume FIFO queue on the semaphore
  - Which is probably not realistic in most implementations

Reverse the unlock order



- Reader does p() then v()
- Writer does p() on binary, then p() 4 times on original, then v() on binary semaphore, then 4x v() on original
- Consider: **W1 W2** R1 R2 R3 R4 R5 R6

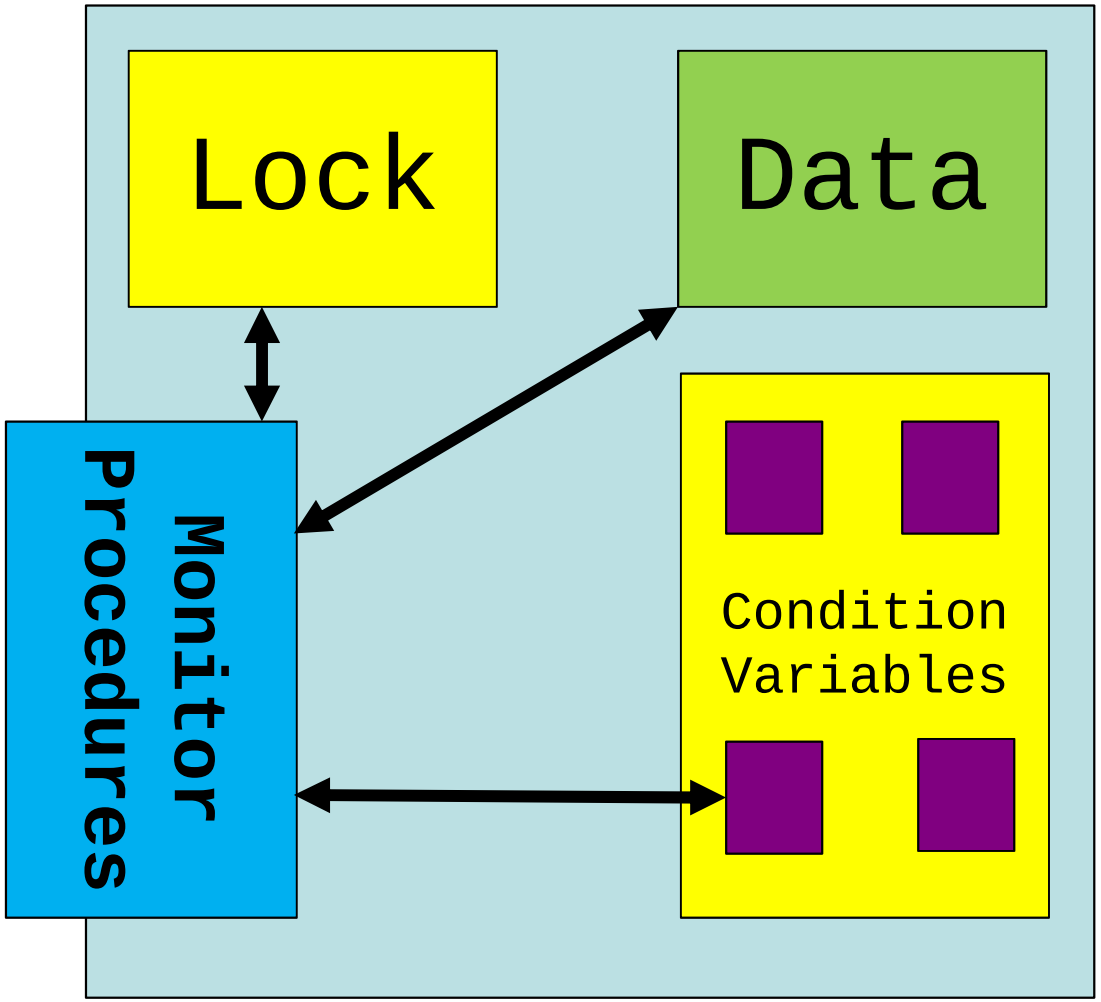
Overview of semaphore solution

- With one writer: a single semaphore works for multiple readers or one writer at once
- As soon as you have multiple writers there are problems
  - E.g. They can get half of the semaphore count each
- You could implement mutual exclusion around the writers, so that only one can go for it at once
  - But that can deprioritise the multiple writers

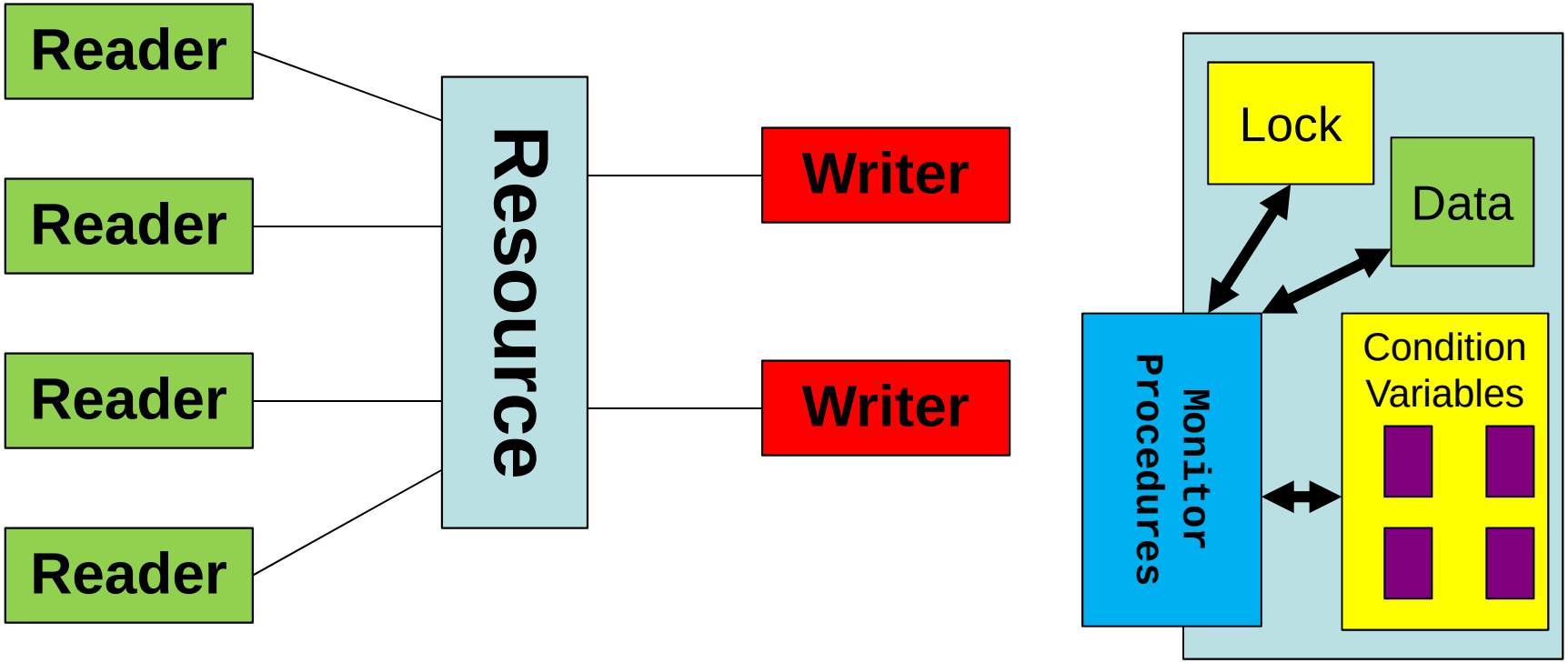
Monitor solution?

Structure of a monitor

- Private data
- Public methods
- Locks
- Condition variables

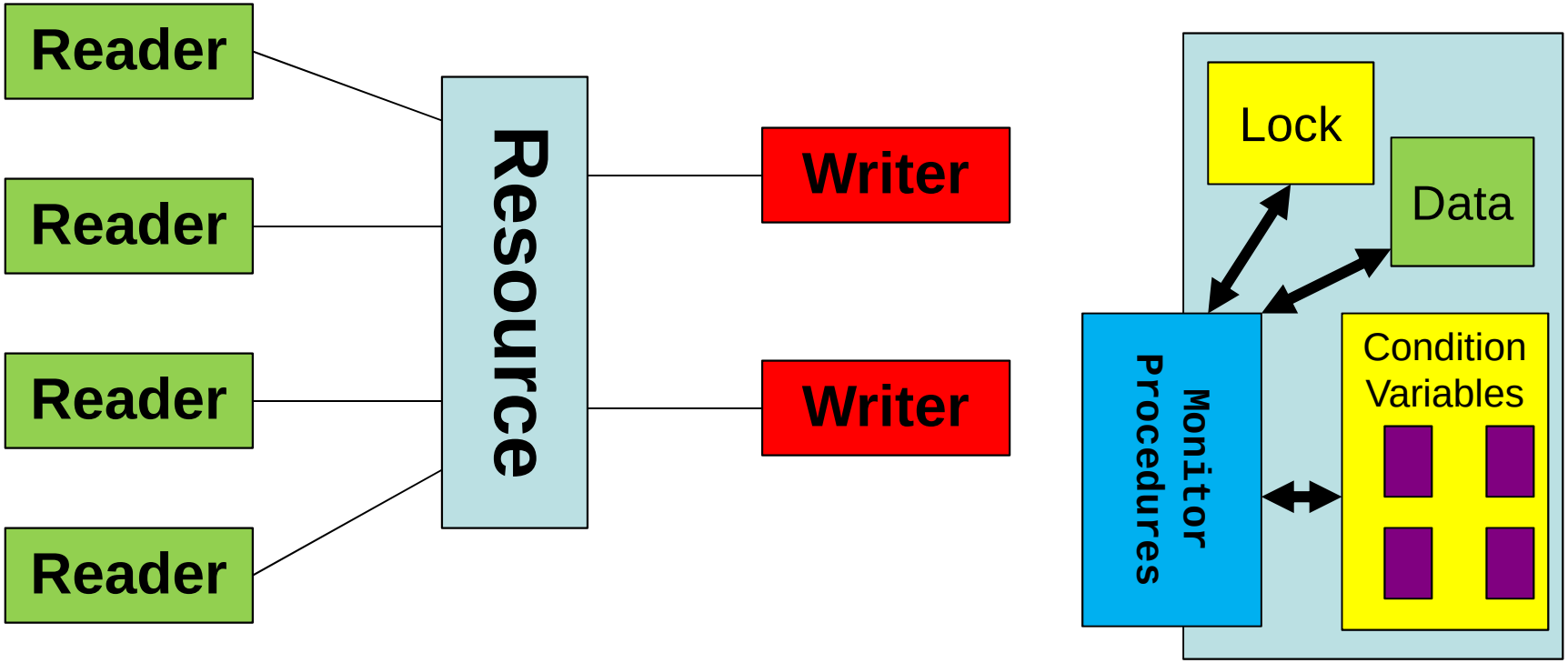


Could we use a monitor for this?



- Simple implementation:
  - Data = the resource to protect
  - Mutual exclusion is guaranteed by the monitor
  - Could we do better?

Use the monitor to arbitrate access



- Better implementation – monitor just arbitrates access
- Resource is global and shared
- All access has to go through the monitor at the start and end, to request access and to release access

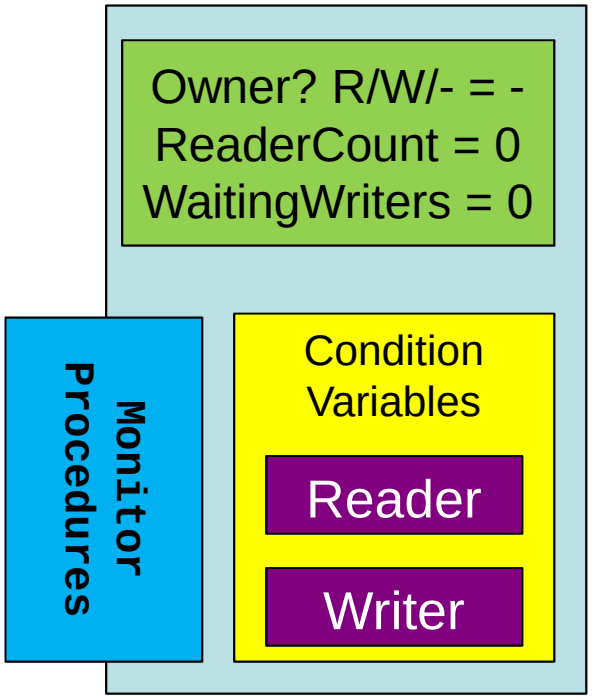
Using a monitor to arbitrate access

- All of these are within the monitor procedures, to ensure mutual exclusion

- Four protocols/procedures are needed:
  - Reader entry protocol – reader wants to read
  - Reader exit protocol – reader finishes reading
  - Writer entry protocol – writer wants to write
  - Writer exit protocol – writer finishes writing

Reader protocols

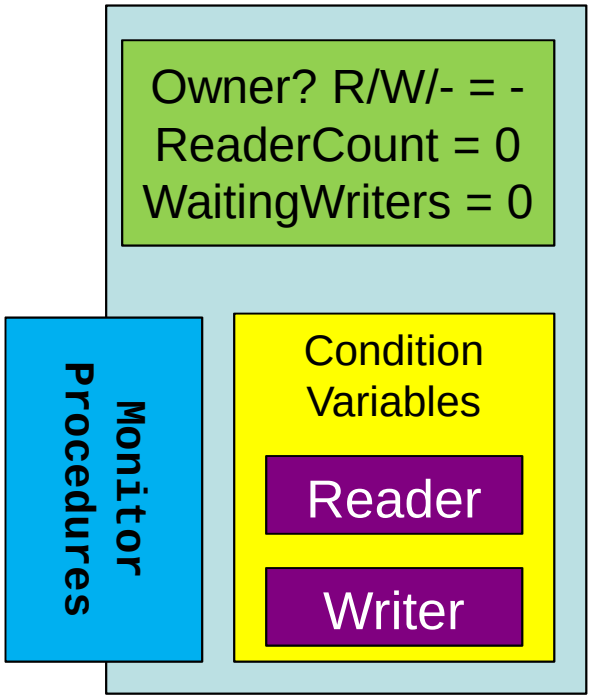
- Reader requests access: (entry protocol)
  - If a writer currently has the resources:
    - Wait on condition variable for readers
    - When awoken re-check this condition
  - If no reader currently has the resources:
    - Record that a reader has it now
  - Increment the count of active readers



- Reader finishes: (exit protocol)
  - Decrement count of readers
  - If count is now zero then:
    - Clear that a reader has it
    - Notify the writer condition variable (may be a writer waiting)

Writer protocols

- **Writer requests access: (entry protocol)**
  - If a reader or writer currently has the resource:
    - Increment count of waiting writers
    - Wait on condition variable for writers
    - When awoken, decrement count of waiting writers then re-check the condition above
  - If we got here then no thread has access
    - Record that a writer has it



- **Writer finishes: (exit protocol)**
  - Clear that a writer has it
  - Notify (all) reader and writer condition variables

- Note: We could prefer to signal one condition variable over the other

Consider various cases

- **W1** R1 R2 R3 R4 R5 R6 **W2**
- **W1 W2** R1 R2 R3 R4 R5 R6
- R1 R2 R3 **W1 W2** R4 R5 R6

Readers' Preference protocol

- Given a sequence of read and write requests which arrive in the following order:
  - $R_1 R_2 W_1 R_3 \ \ldots$
- In a *Readers' Preference* protocol: R_3 takes priority over W_1
  - $R_1 R_2 R_3 W_1 \ \ldots$
- Writer may have to wait arbitrarily long time before getting access

Next Lecture

- Overview of what we have seen so far
  - And of hardware support for features
- Considering larger problems
  - And how we can solve them using what we have seen so far