Project 2
Hello, Synchronization!

April 1, 2021
SNU Operating Systems

Orientation matters

- Should phones turn on their displays when laid face down?
- Orientation matters when you define a device’s behavior!

Overview

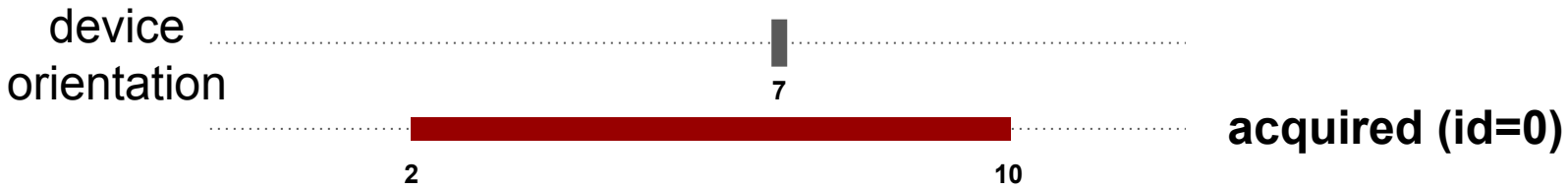
- Implement a rotation-based read/write lock
- Read/write access is claimed by processes through a syscall
- Each lock has a “degree range”
  - Access is granted when the current orientation is in the degree range
  - Otherwise, the process is put to sleep until the orientation is in-range
- Read access is granted when no existing writer range overlaps
- Write access is granted when no existing reader/writer range overlaps
  - Thus a writer has exclusive access

Orientation Degree Range

- We assume 1D device orientation
  - Actually, Tizen has three orientation axes (azimuth, pitch, roll)!
- low <= **range** <= high
- Rotation ranges are **inclusive**
  - Ex) [30, 60] and [60, 90] overlap
- Rotation ranges are **circular**
  - Ex) [330, 30] and [30, 330] are different
  - Ex) [330, 90] and [285, 345] overlap at [330, 345]

Rotation Lock Example

`rotation_lock(2, 10, ROT_WRITE) → id=0`

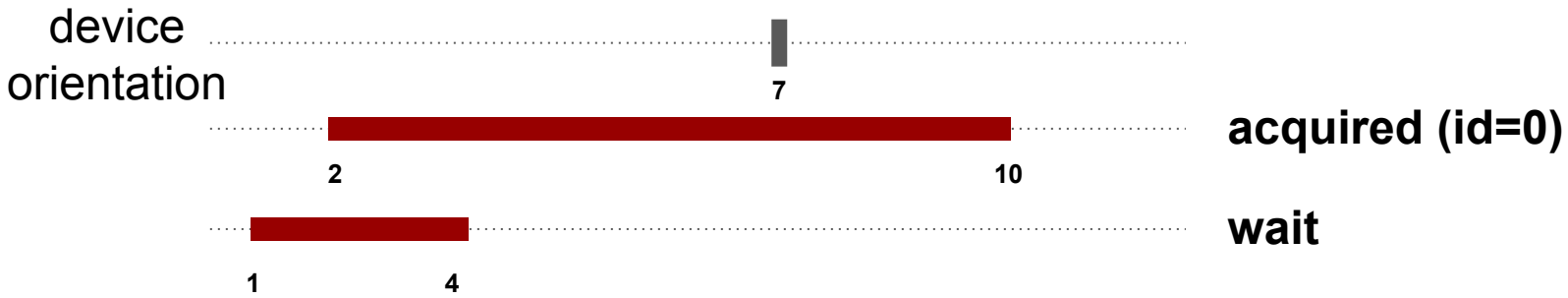


writer

reader

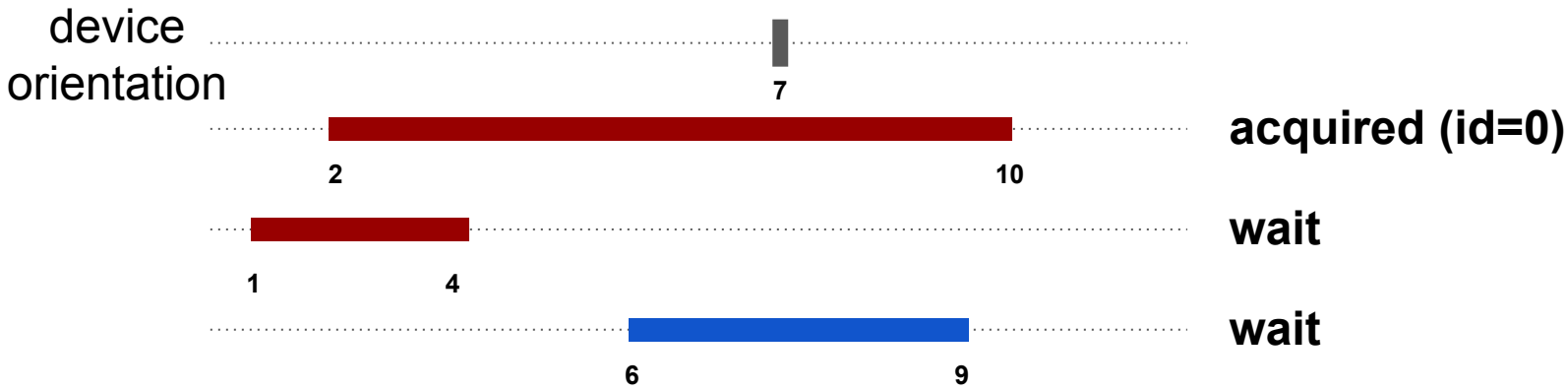
Rotation Lock Example

`rotation_lock(1, 4, ROT_WRITE)` → wait



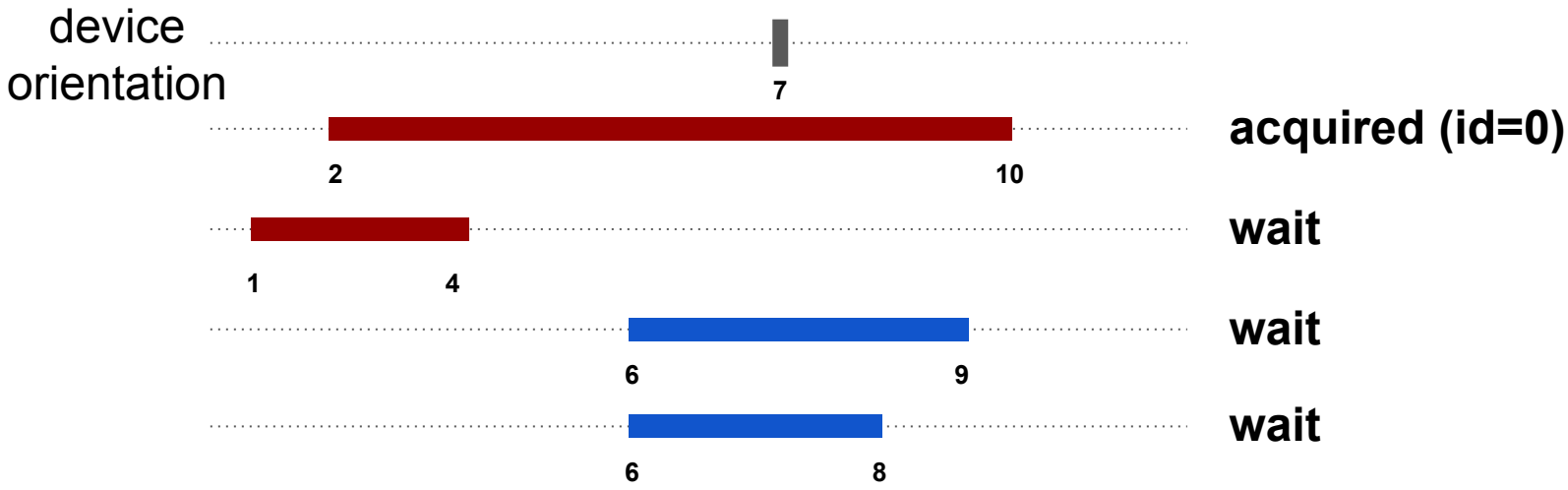
Rotation Lock Example

`rotation_lock(6, 9, ROT_READ)` → wait



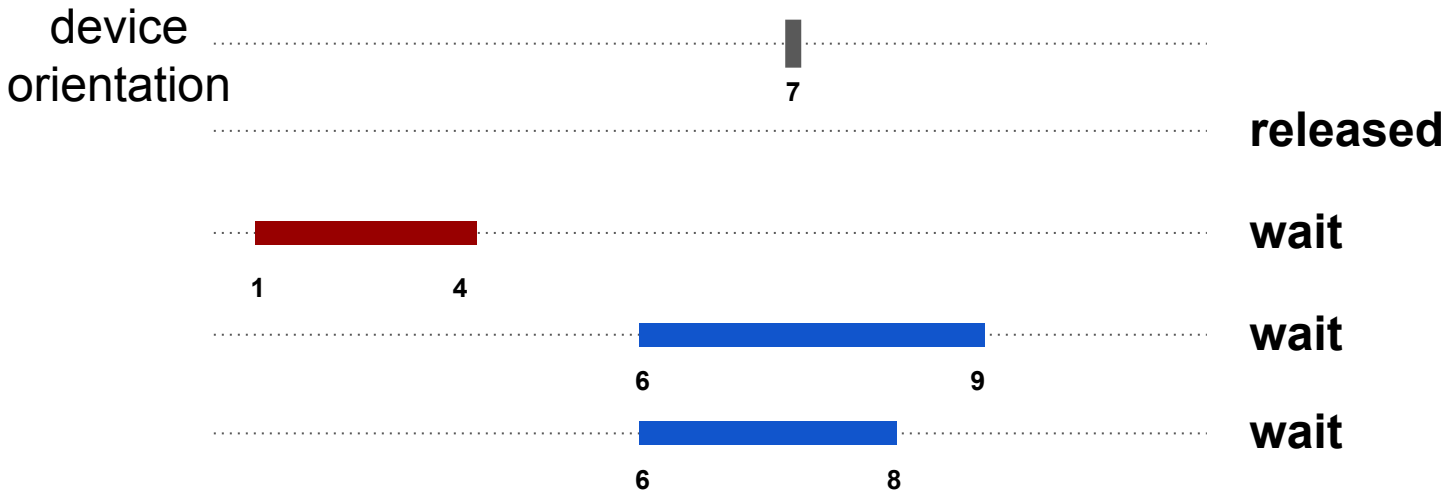
Rotation Lock Example

`rotation_lock(6, 8, ROT_READ)` → wait



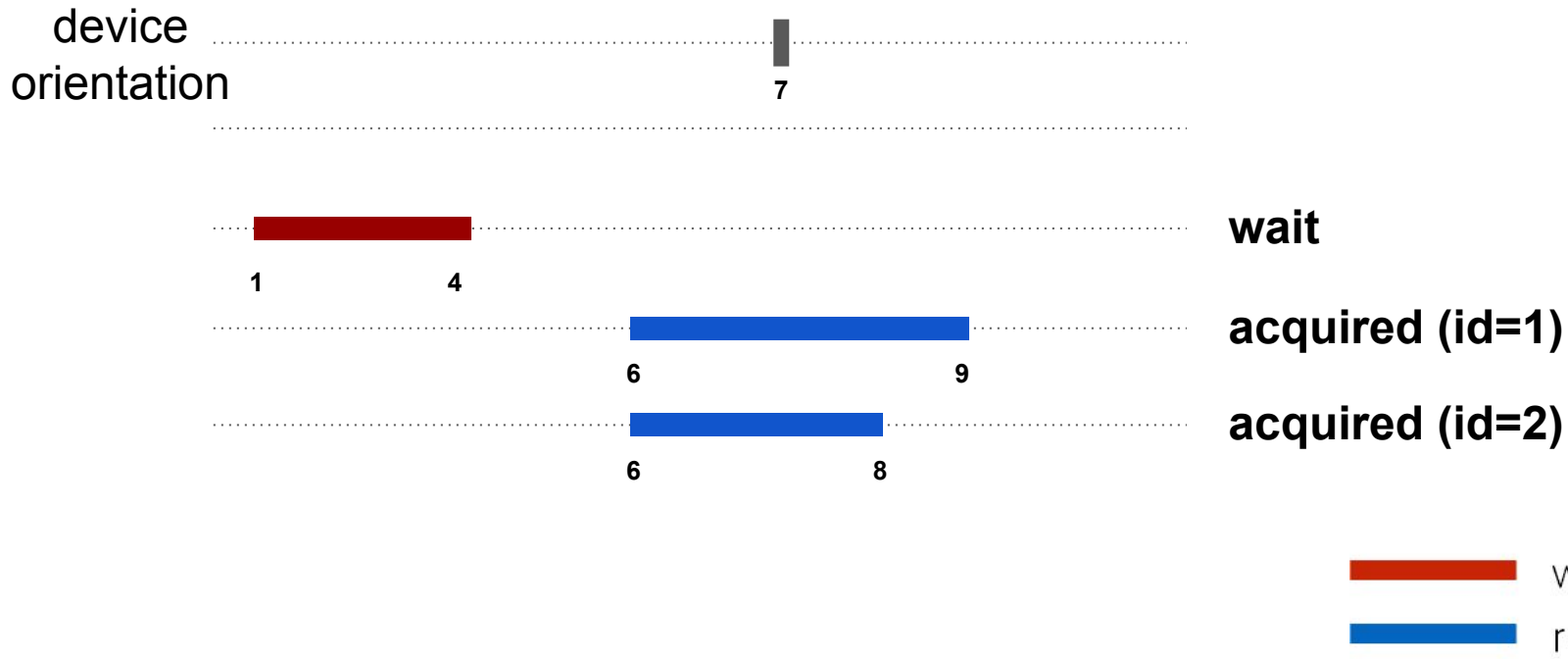
Rotation Lock Example

`rotation_unlock(0)`



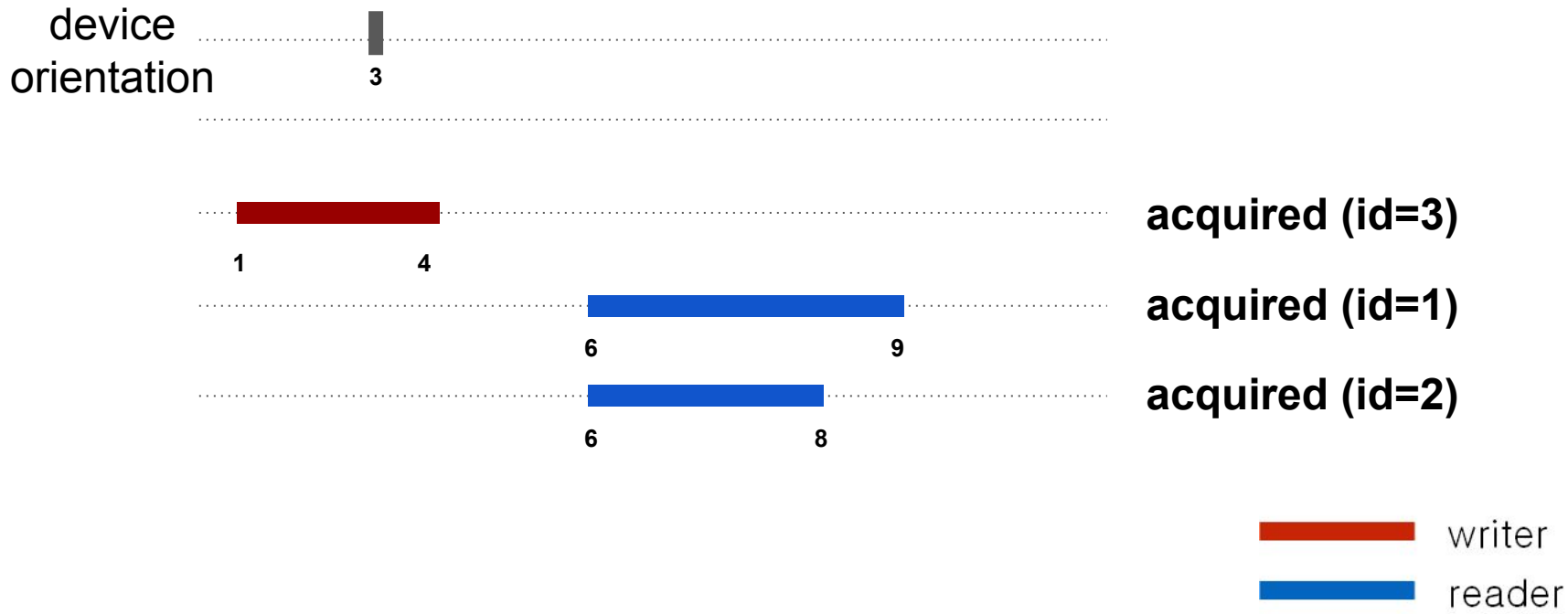
Rotation Lock Example

`rotation_unlock(0)`



Rotation Lock Example

```
set_rotation(3)
```

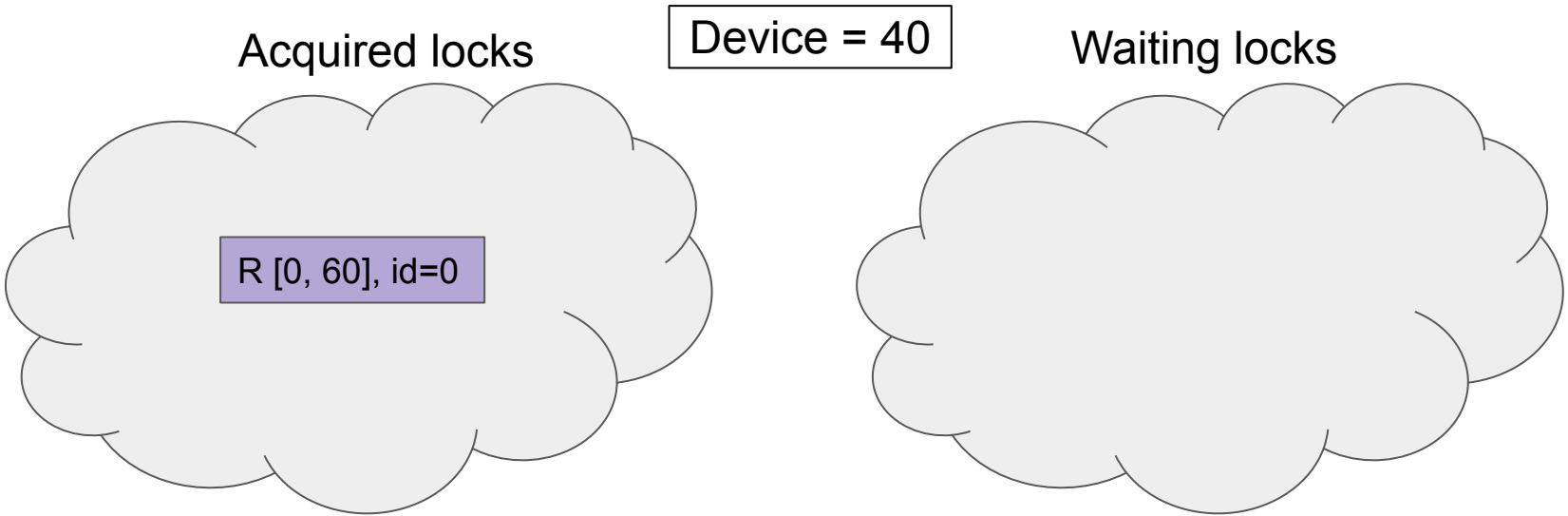


Mitigating writer starvation

- Implement the following policy
  - **Among in-range waiting locks, readers whose degree range overlap with that of a waiting writer must not be granted access before the waiting writer.**
- Note
  - This policy does not *prevent* writer starvation
  - This policy does not care about reader starvation

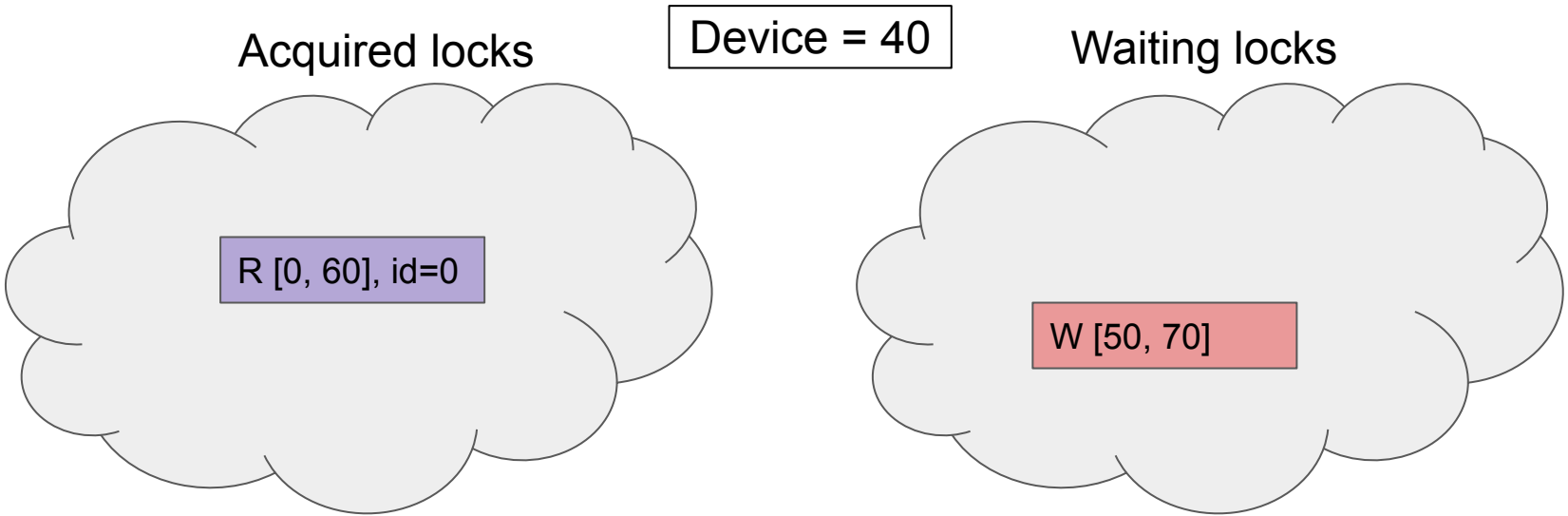
Fairness Policy

- `rotation_lock(0, 60, R)` → acquired (id=0)



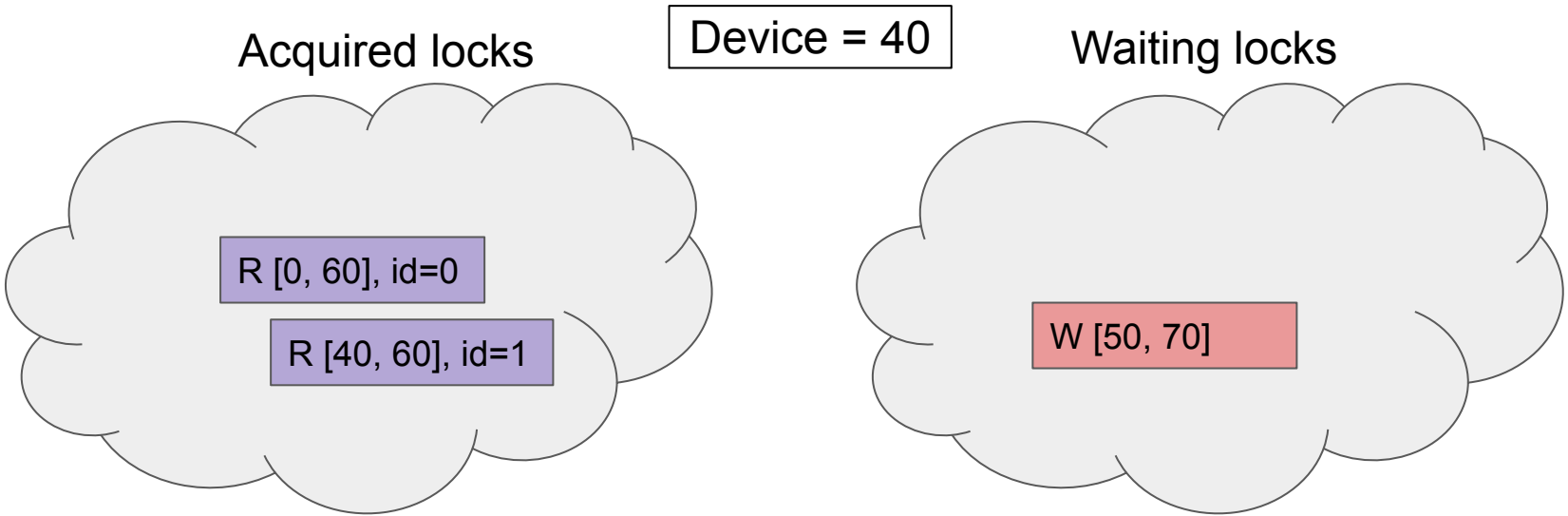
Fairness Policy

- `rotation_write(50, 70, W)` → wait



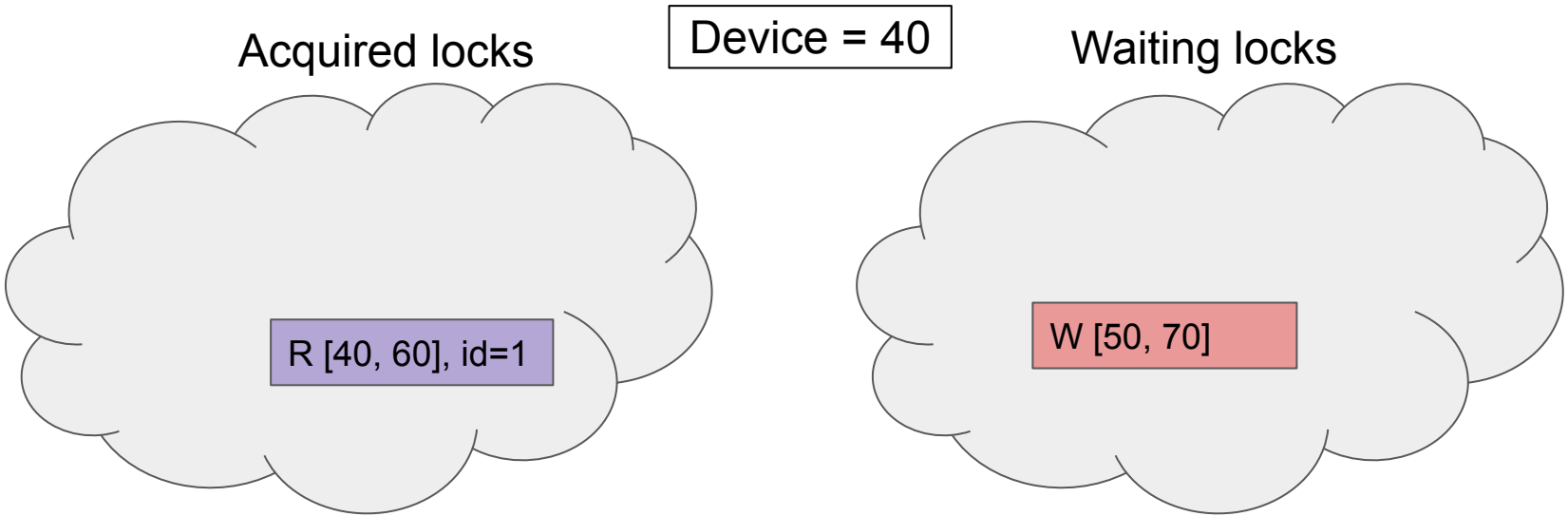
Fairness Policy

- `rotation_lock(40, 60, R)` → acquired (id=1)



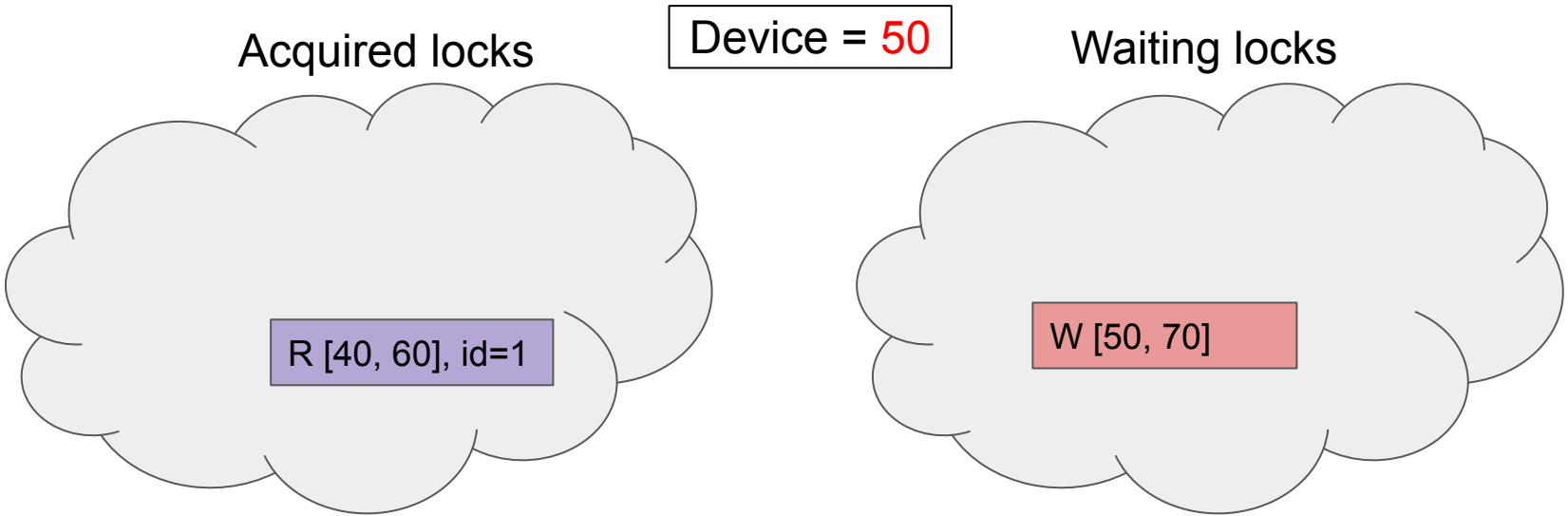
Fairness Policy

- `rotation_unlock(0)`



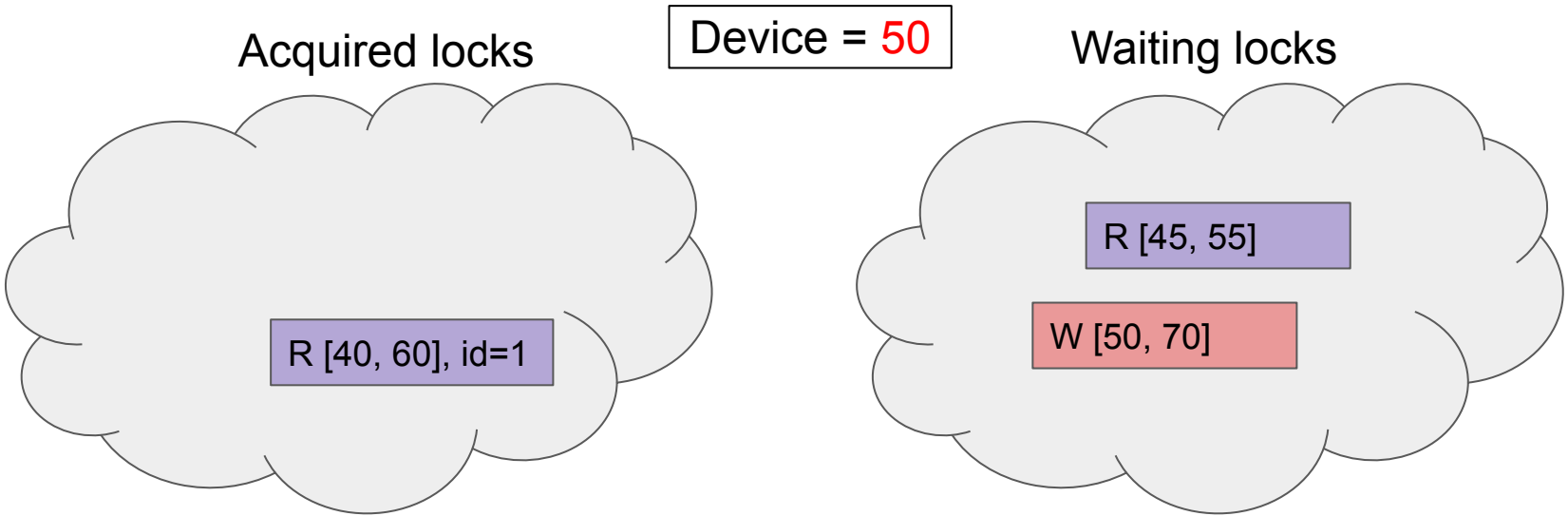
Fairness Policy

- Orientation changes 40 → 50



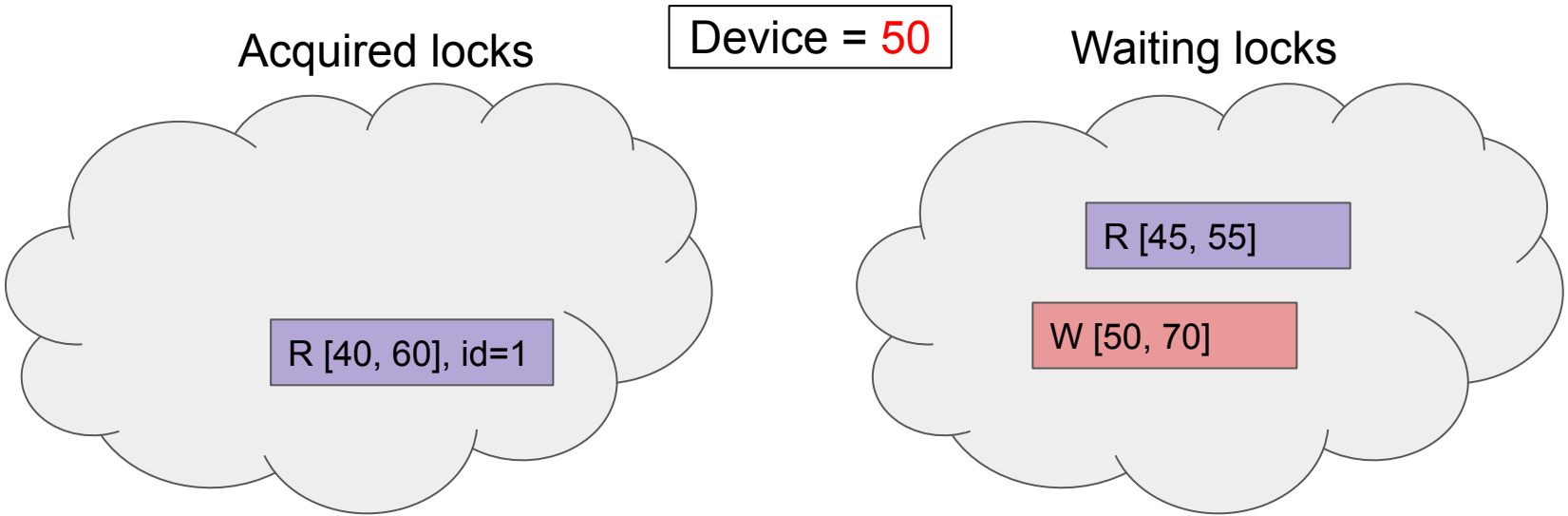
Fairness Policy

- `rotation_lock(45, 55, R)` → **wait**



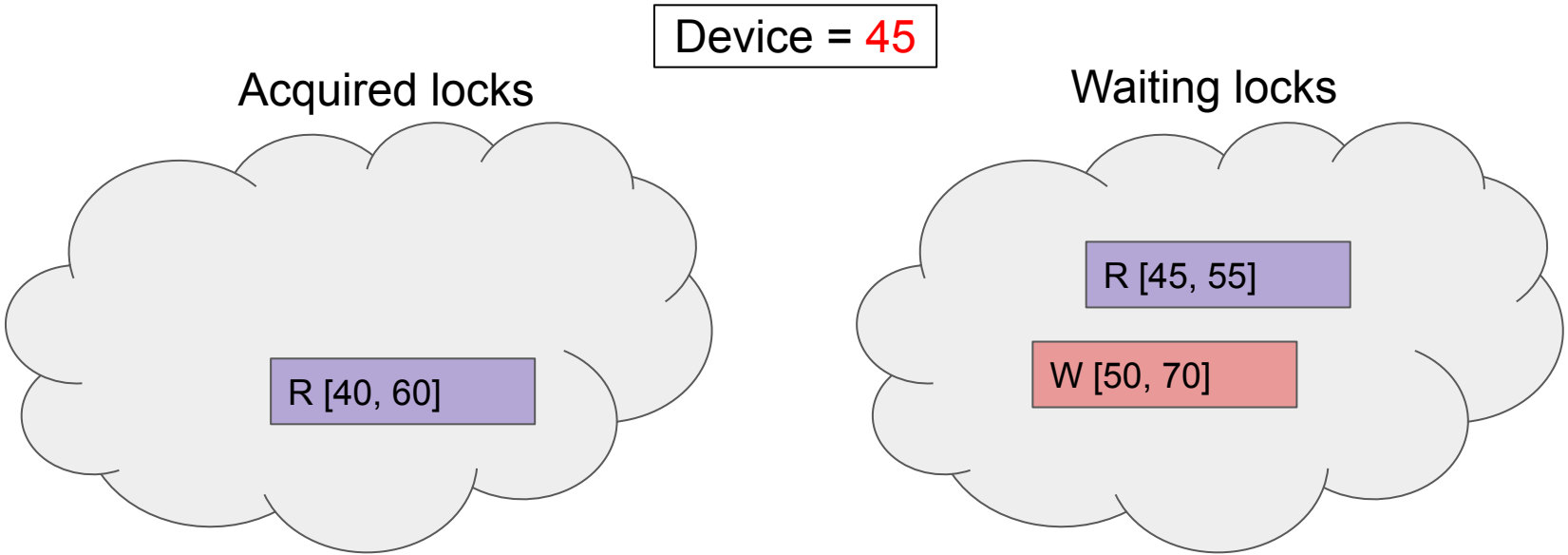
Fairness Policy

- An *in-range* writer (W [50, 70]) is waiting
- Then, all other *in-range* readers (R [45, 55]) must wait for the write lock



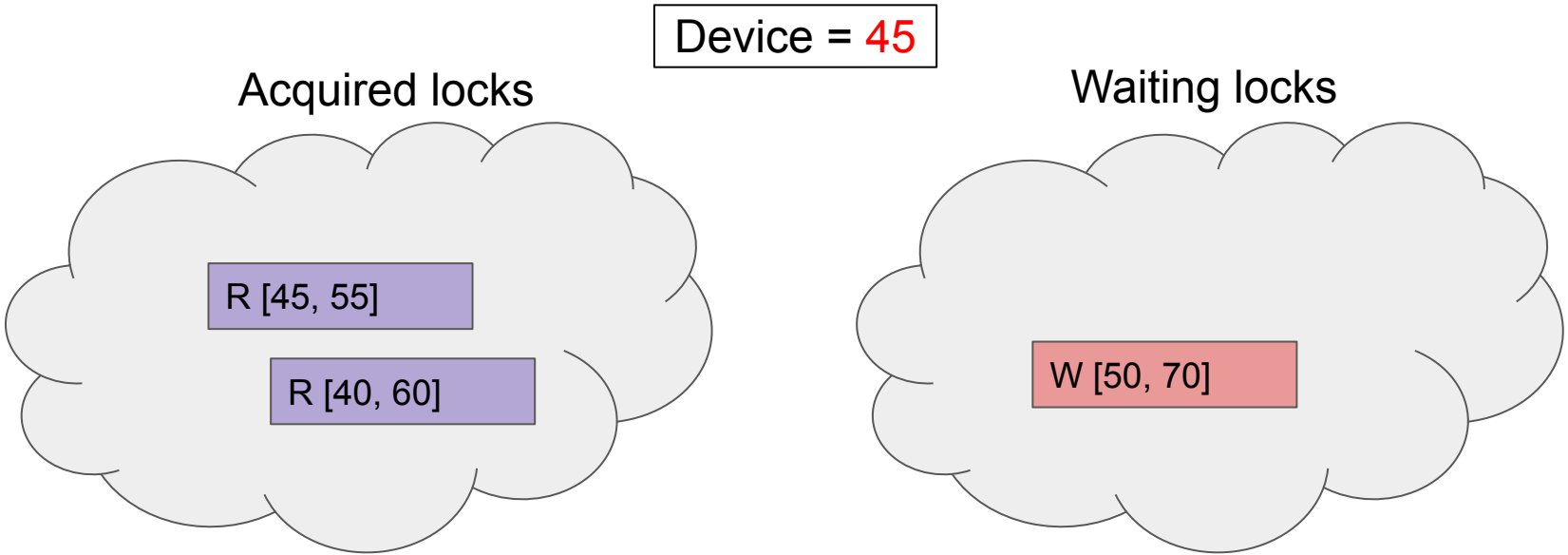
Fairness Policy

- Rotation changes 50 → 45
- W [50, 70] is no longer *in-range*



Fairness Policy

- R [45, 55] acquires the lock



Test: Professor & Student

- The two access the same file `quiz` on the filesystem
  - Professor writes, Student reads
- Synchronize access using the same degree range
  - **[0, 180]**
  - If current orientation is 240, both processes wait
- Test code
  - `test/professor.c, test/student.c`

Test: Professor & Student

Professor	Student
`rotation_lock(0,180,W)` → id=0	
	`rotation_lock(50,70,R)` → wait
`write(file, "10")`	
`rotation_unlock(0)`	read access granted (id=1)
`rotation_lock(0,180,W)` → wait	
	`read(file, &num)` → num is 10
	`write(stdout, "10 = 2 * 5")`
write access granted (id=2)	`rotation_unlock(1)`

Hints: Blocking Mechanisms

- Mutex
- Semaphore
- Condition Variable
- Wait Queue (Starts with `DECLARE_WAIT_QUEUE_HEAD`)

Hints: Synchronization

- Avoid races and deadlocks
  - We work on a 4-core machine
  - You must synchronize accesses to your internal data structures
  - Always think about the worst case scenario
    - Your mindset: *Anything that can go wrong will go wrong.*
    - e.g. While one thread is removing a lock from waiting list, another thread may (or *will*) access the list at the same time

Hints: Terminating Routine

- A process holding a write lock terminated. What happens?
- Locks should be **released** (held locks) or **removed** (waiting locks) when the process terminates.
- Instructions
  - Implement `exit_rotlock(task_struct)` in `kernel/rotation.c`
    - Release locks held by the task
    - Remove the task's entry from waiting list
  - Call it inside `do_exit()` in `kernel/exit.c`
- You'll might also need an init function. Where should you put it?

Hints: Miscellaneous

- List entries can be deleted (`list_del`) during iteration
  - `list_for_each_entry_safe`
- The rotation range is circular
  - You should implement a logic for determining two circular ranges are overlapping or not

Design Review

- Conversation with the TA about your design and plan
- Check the eTL notice
- Due: 4/5 (Mon)

About submission (IMPORTANT!)

- Don't be late!
  - TAs will clone all repositories exactly at the deadline
- Submit code
  - Your team's private project 2 repo (swsnu/project-2-hello-lock-team-n), master branch
  - README: description of your implementation, how to build, and lessons learned
- Submit slides and demo (n is your team number)
  - Email: osspr2021@gmail.com
  - Title: [Project 2] Team n
  - Attachments: `team-n-slides.{ppt,pdf}`, `team-n-demo.{mp4,avi,…}`
    - One slide file, one demo video!

Q & A