Synchronization via Semaphores

General Synchronization Task

- Operations or instructions of concurrent processes (or threads) can interleave.

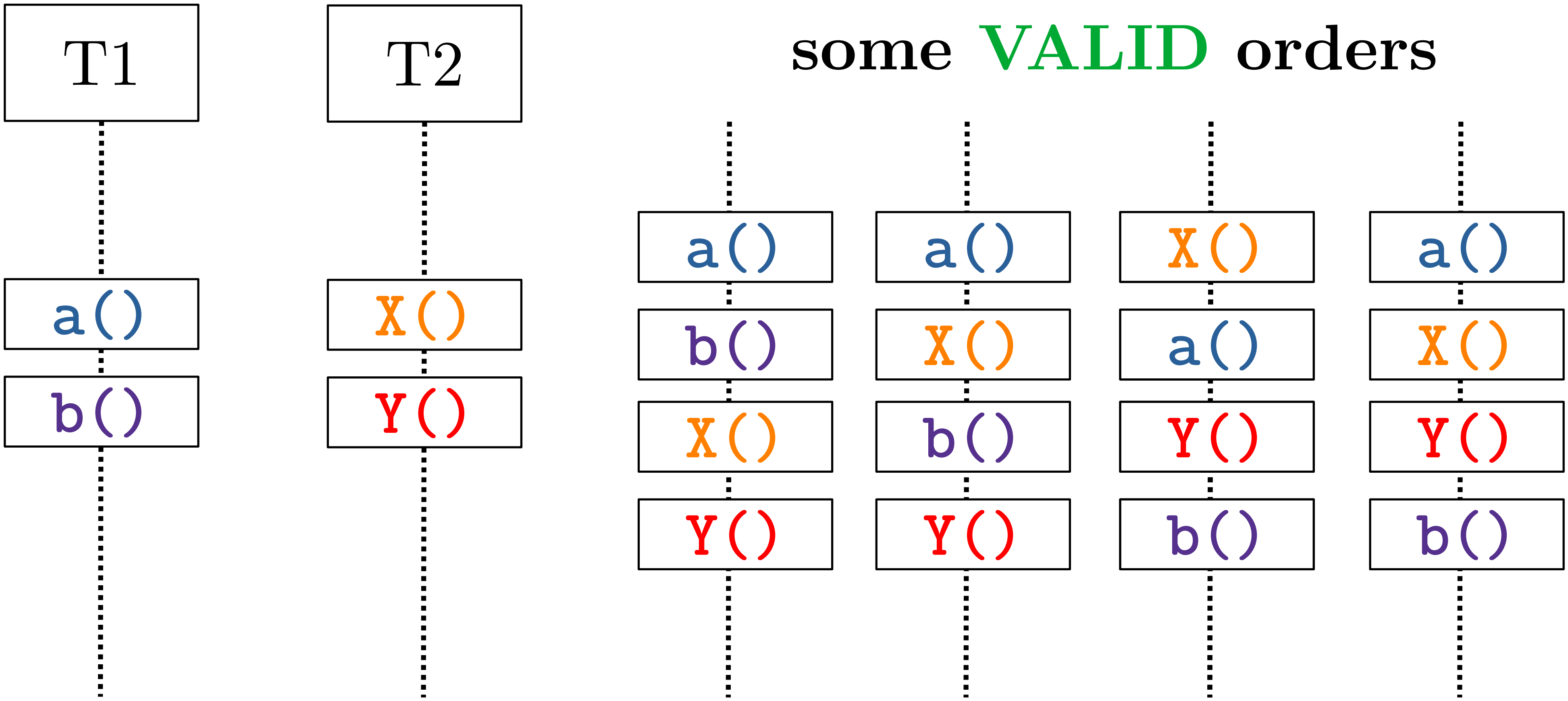
- Depending on the situation, there are **orders** of operations/instructions that can be considered **VALID** and there are orders that can be considered **INVALID**.

- **Synchronization** is about implementing different mechanisms such that only the **VALID** **orders** of operation/instructions can occur regardless of how the threads/processes are scheduled by the operating system.

Synchronization Task/Problem 1a: **(Thread) Barrier**

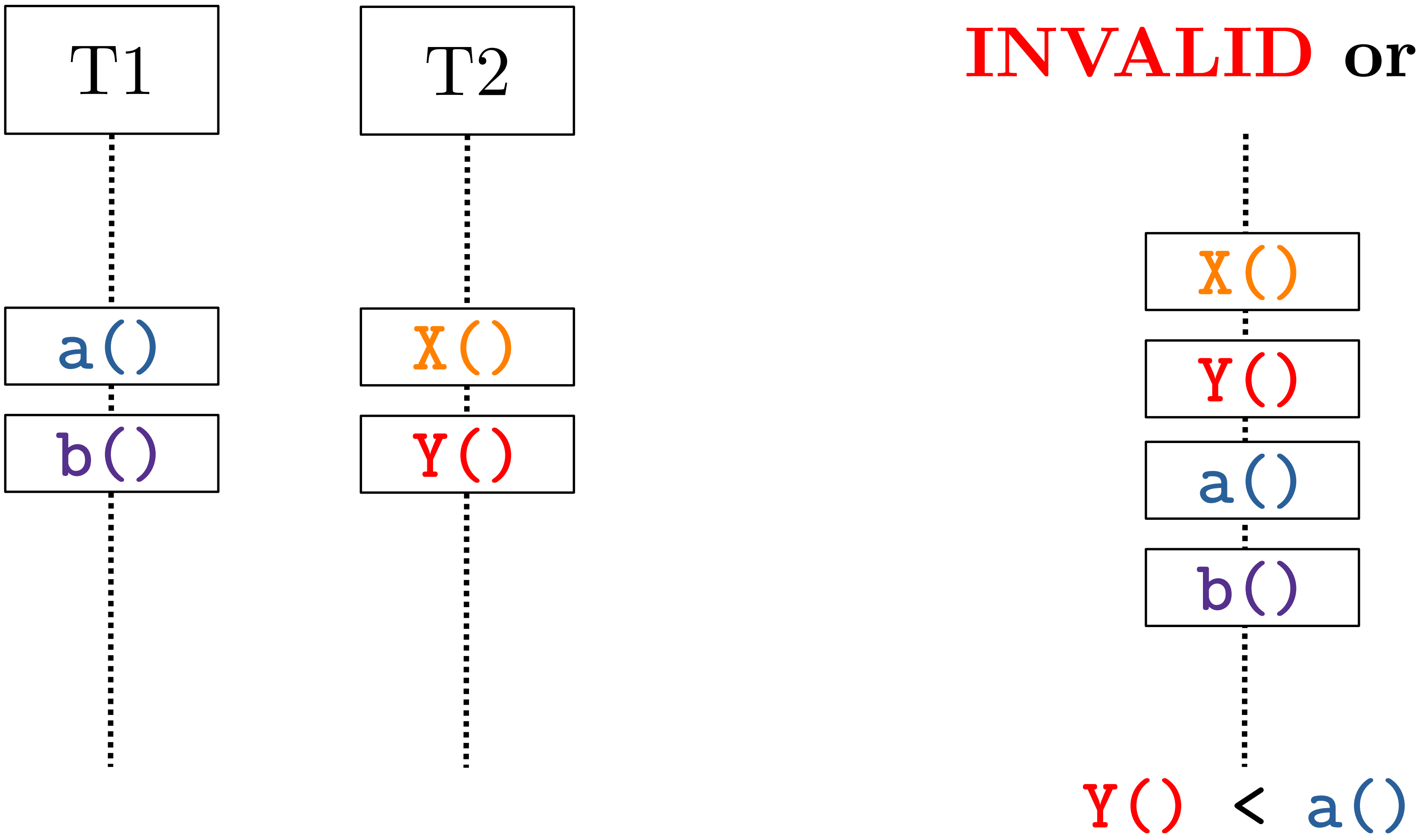
[Problem Description]

Task/Problem 1a: Barrier: `a()` < `Y()`



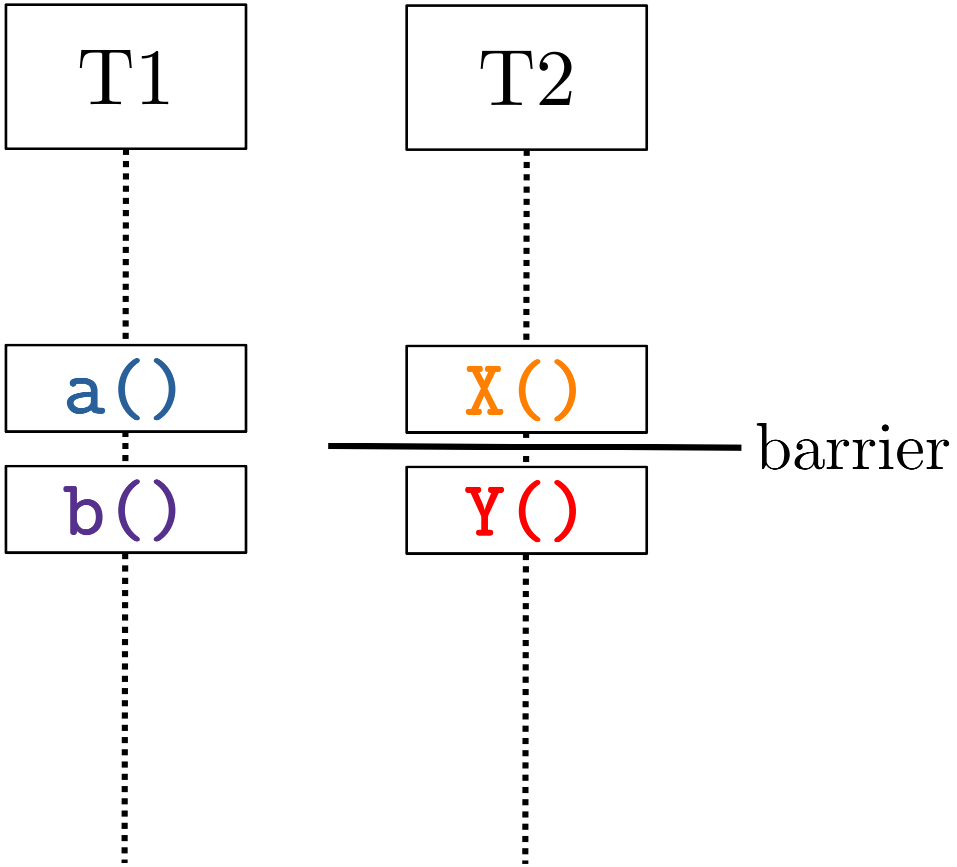
Constraint: We consider only those orders where `a()` is performed before `Y()` as VALID.

Task/Problem 1a: Barrier: a() < Y()



Constraint: We consider only those orders where a() is performed before d() as VALID.

Task/Problem 1a: Barrier: `a()` < `Y()`



A *barrier* should be placed before the call `Y()` in T2.

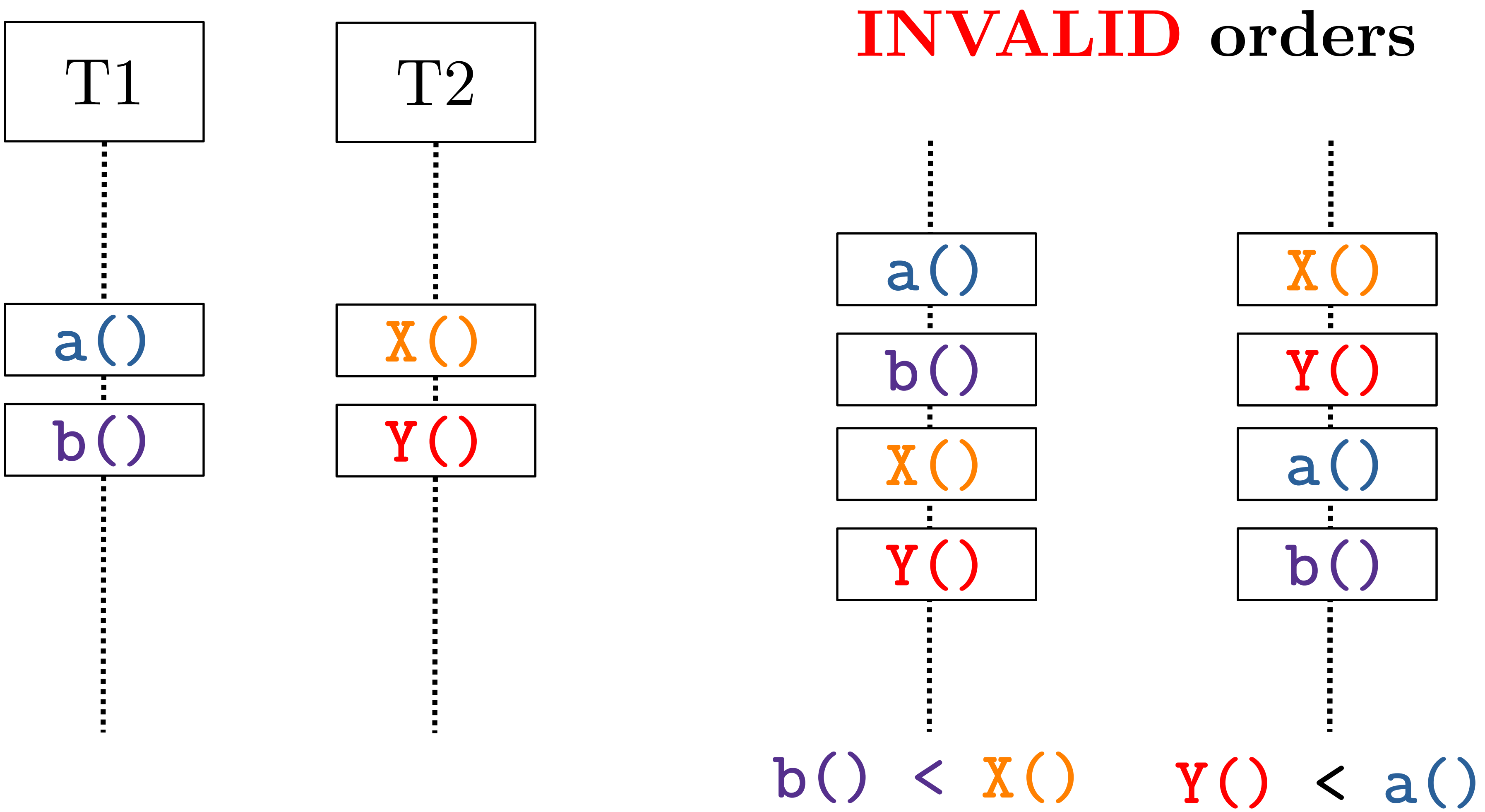
This barrier means T2 has to wait for T1 to complete the `a()` call before performing `Y()`.

Constraint: We consider only those orders where `a()` is performed before `Y()` as VALID.

Synchronization Task/Problem 1b: **(Thread) Barriers**

[Problem Description]

Task/Problem 1b: Barriers: `a()` < `Y()`, `X()` < `b()`



T1

`a()`

`b()`

T2

`X()`

`Y()`

INVALID orders

`a()`

`b()`

`X()`

`Y()`

`b()` < `X()`

`X()`

`Y()`

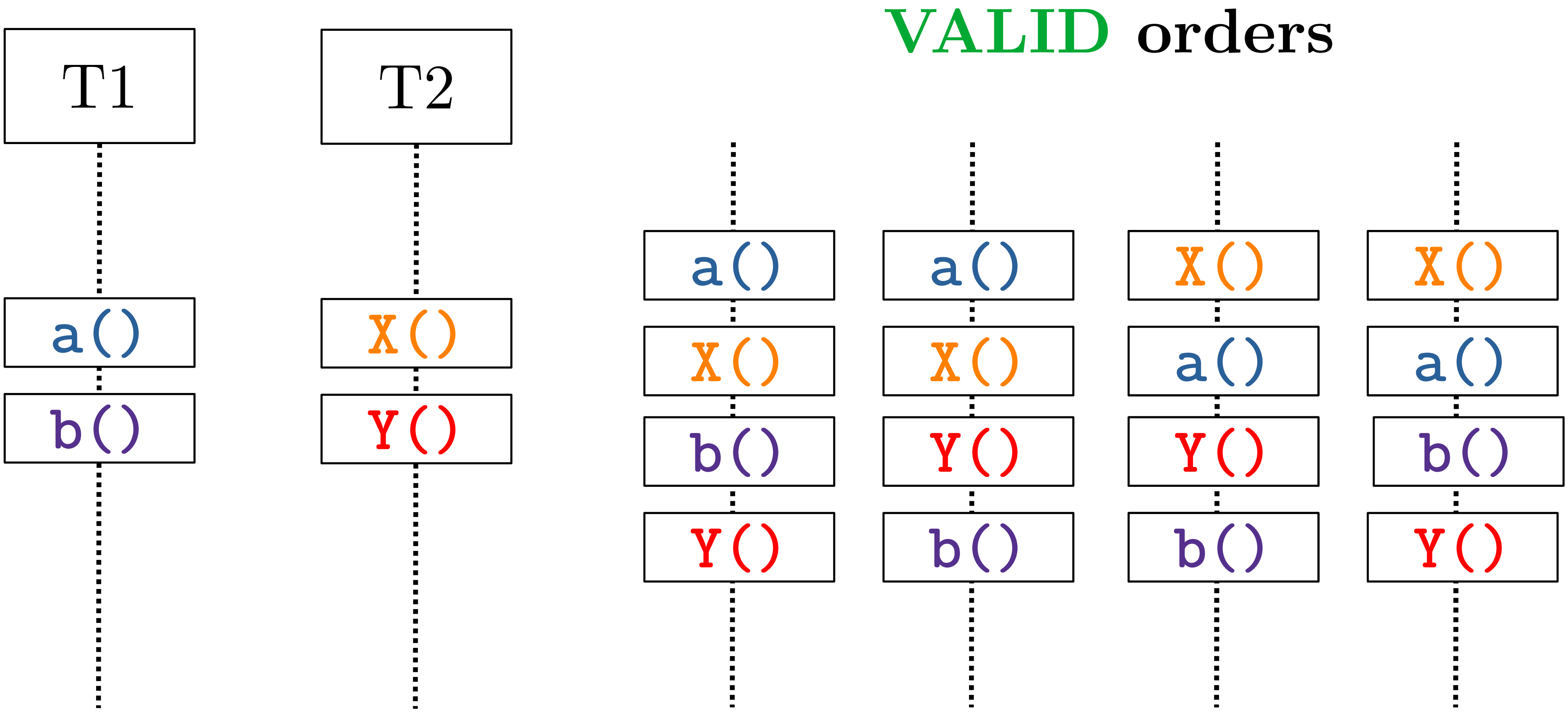
`a()`

`b()`

`Y()` < `a()`

Constraint: We consider only those orders where `a()` is performed before `Y()` and `X()` is performed before `b()` as **VALID**.

Task/Problem 1b: Barriers: `a()` < `Y()`, `X()` < `b()`



VALID orders

T1

a()

b()

T2

X()

Y()

a() X() b() Y()

a() X() Y() b()

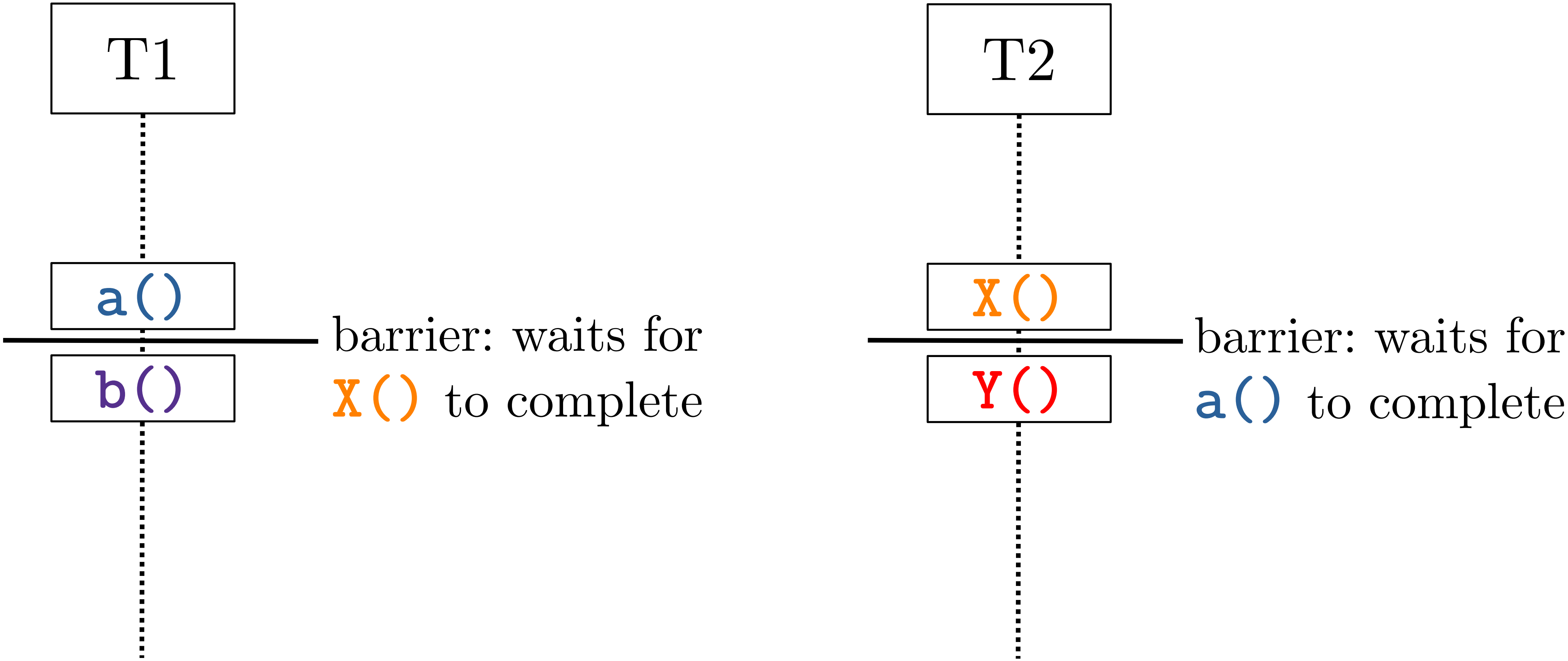
X() a() Y() b()

X() a() b() Y()

`a()` < `Y()`, `X()` < `b()`

Constraint: We consider only those orders where `a()` is performed before `Y()` and `X()` is performed before `b()` as **VALID**.

Task/Problem 1b: Barriers: `a() < Y(), X() < b()`



T1

`a()`

barrier: waits for `X()` to complete

`b()`

T2

`X()`

barrier: waits for `a()` to complete

`Y()`

Constraint: We consider only those orders where `a()` is performed before `Y()` and `X()` is performed before `b()` as **VALID**.

Semaphores

Synchronization Object: Semaphore

Semaphore `S` in an **integer** that signals permission or availability of resources.

`S = n > 0` means there are `n` permits or resources available.

`wait(S)` – a method called by a process/thread in order to request (to **wait** for) a permit or access to a resource. `Decrements S (i.e. S--;)`.

`signal(S)` – a method called by a process/thread in order to return a permit or a resource. `Increments S (i.e. S++;)`.

Synchronization Object: Semaphore

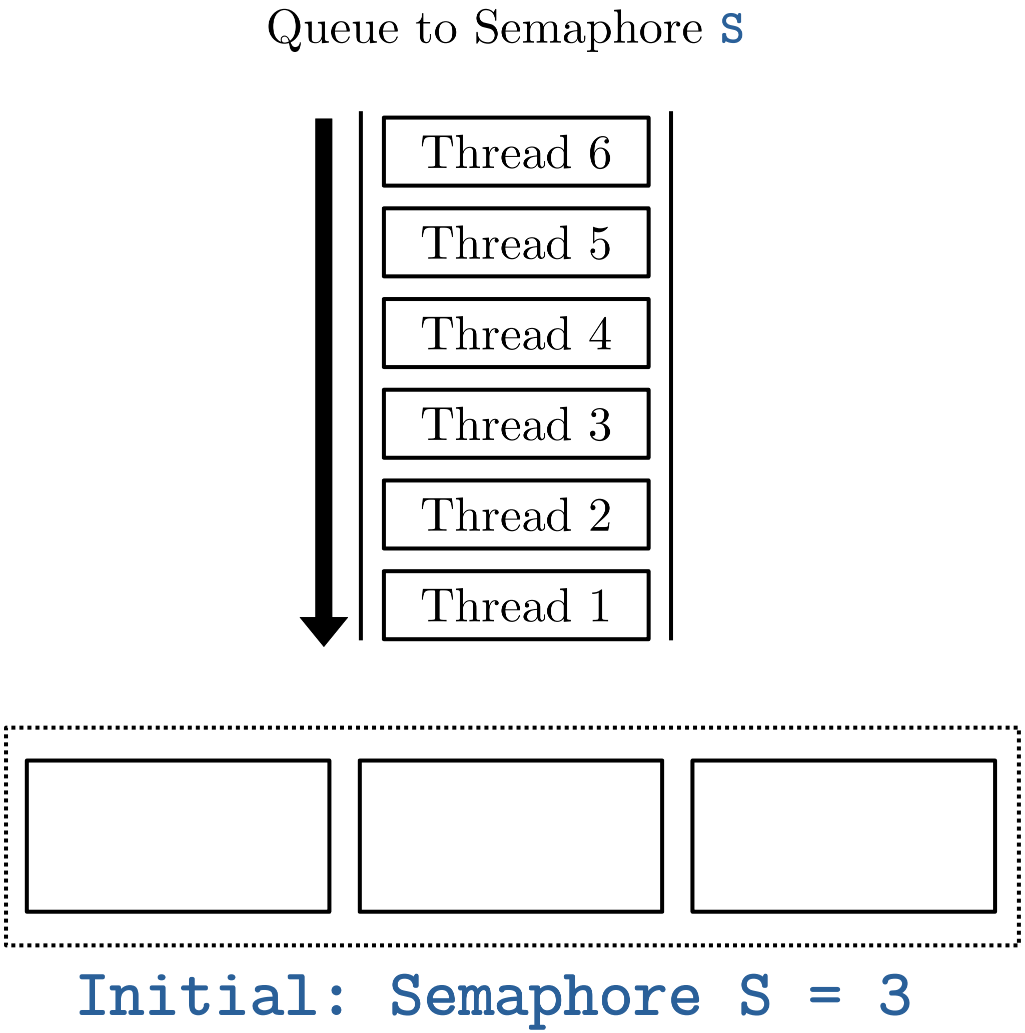
```
wait(S)
{
  while(S <= 0)
  {
    // busy wait
  }
  S--;
}
```

```
signal(S)
{
  S++;
}
```

Semaphores: Other Names for wait and signal

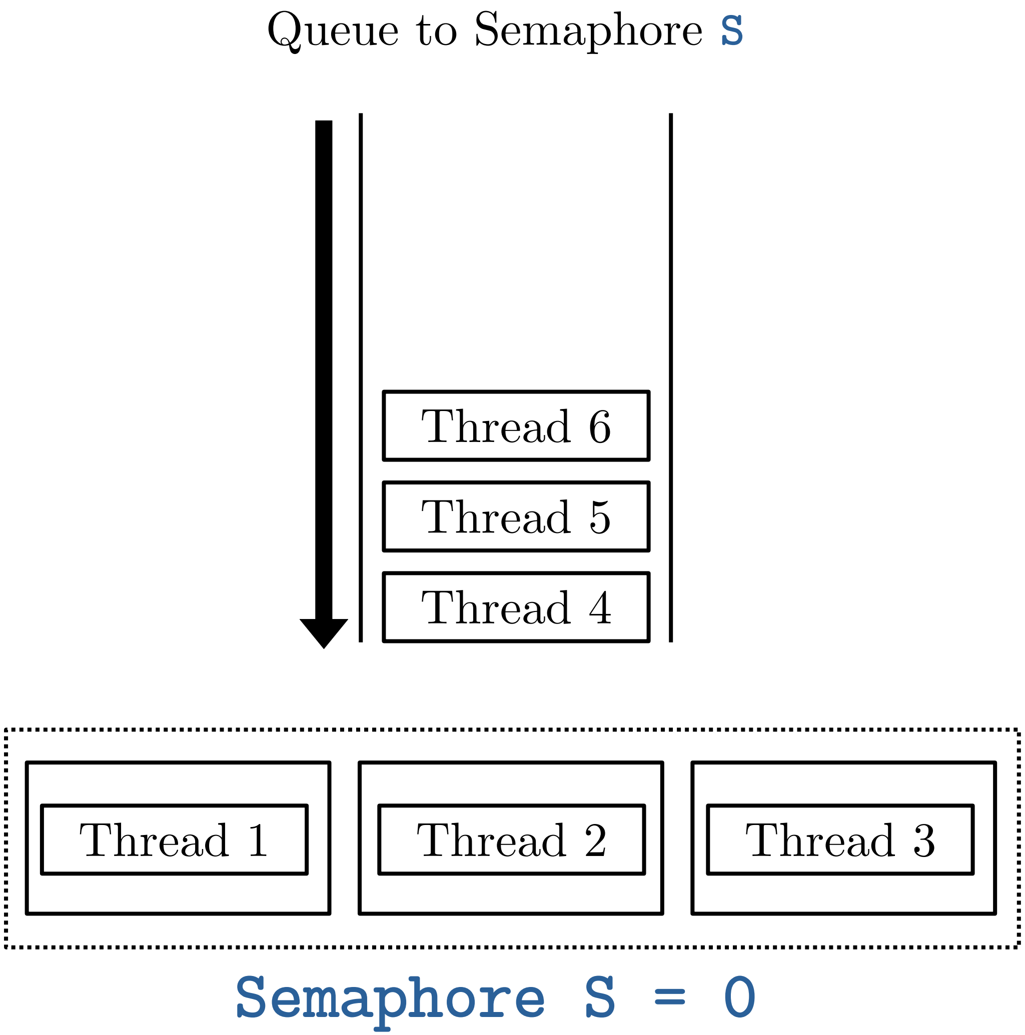
	C	C++
wait(S)	sem_wait(S)	S.acquire()
signal(S)	sem_post(S)	S.release()

Semaphore: Analogy + Some Information



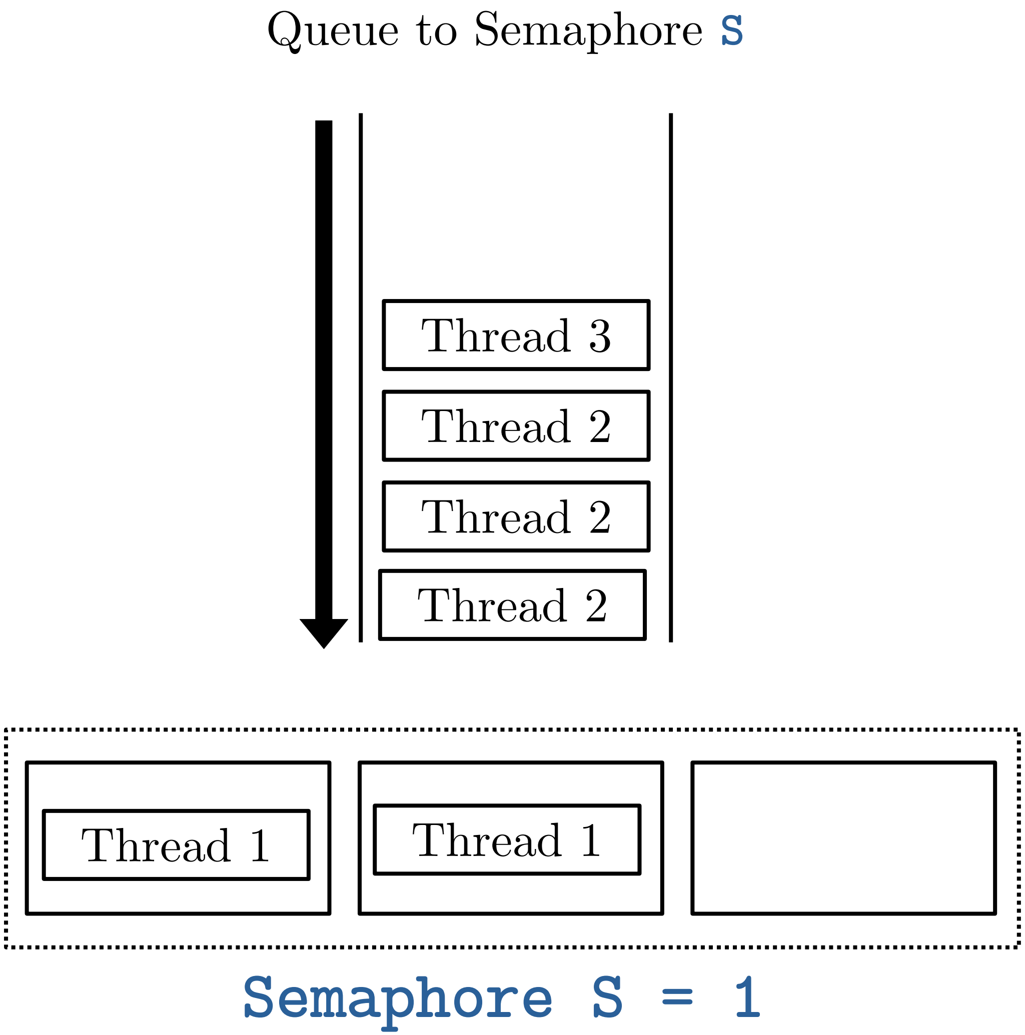
- The boxes represent the permits or available resources.
- The initial number of boxes represent the initial value of the semaphore which is the number of permits or resources available.
- The queue represent all threads that call `wait(S)`. These threads want permits or resources associated with `semaphore S`.
- ***Multiplexing*** is the idea that multiple threads can acquire permits or resources at the same time.

Semaphore: Analogy + Some Information



- The boxes represent the permits or available resources.
- The initial number of boxes represent the initial value of the semaphore which is the number of permits or resources available.
- The queue represent all threads that call `wait(S)`. These threads want permits or resources associated with `semaphore S`.
- ***Multiplexing*** is the idea that multiple threads can acquire permits or resources at the same time.

Semaphore: Some Information



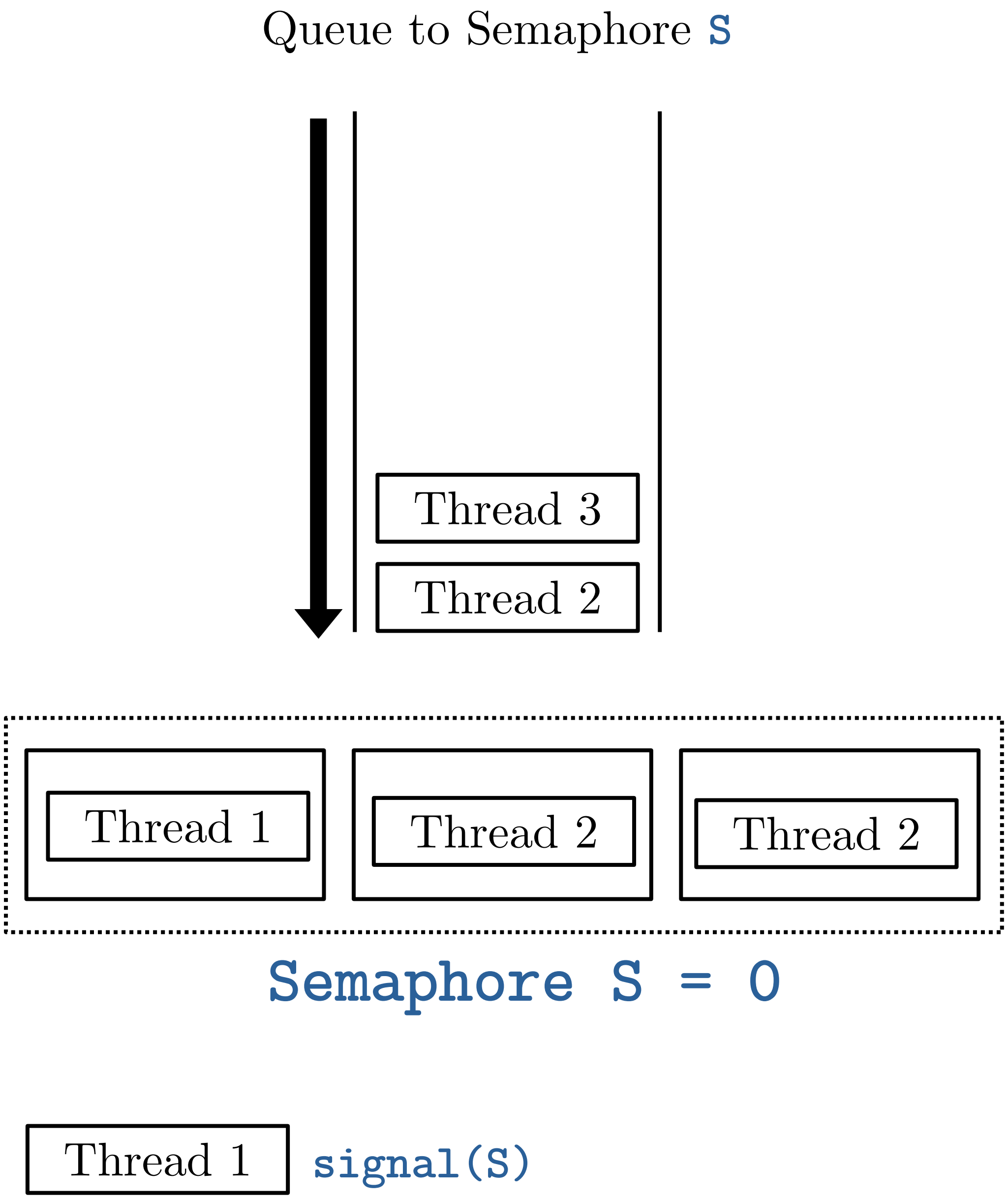
- ***Multiplexing*** is the idea that multiple threads can acquire permits or resources at the same time.

- A semaphore is called ***counting*** if it can give more than 1 permit or resource.

- A semaphore is called ***binary*** if at most it can give 1 permit or 1 resource. S ≤ 1.

- A thread calling `wait(S)` multiple times means it wants multiple permits or access to multiple resources associated with semaphore `semaphore S`.

Semaphore: Analogy + Information



Queue to Semaphore S

Thread 3

Thread 2

Thread 1

Thread 2

Thread 2

Semaphore S = 0

Thread 1 signal(S)

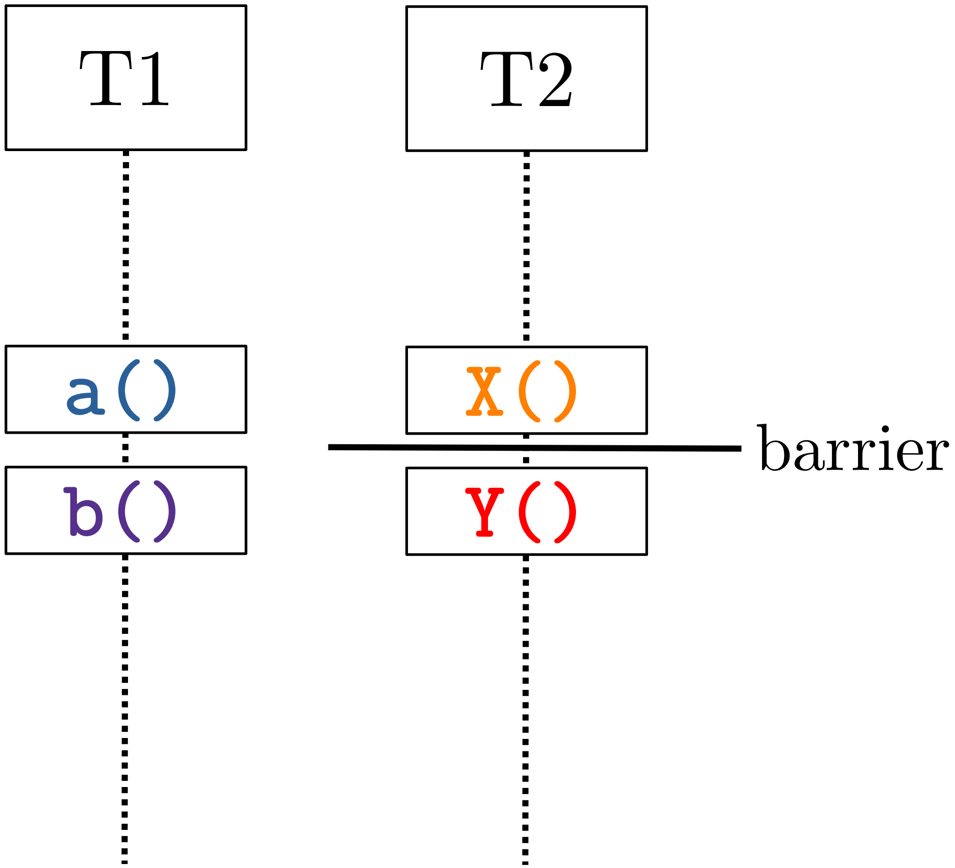
- Threads calling the signal(S) increments semaphore S and thus increasing the number of permits or resources available that are associate with semaphore S.

- In different programming languages, you can specify a semaphore's upper limit value which is the maximum number of permits or resources associated with that semaphore.

Synchronization Task/Problem 1a: **(Thread) Barrier**

[Solution Description]

Solution 1a: Barrier: `a() < Y()`

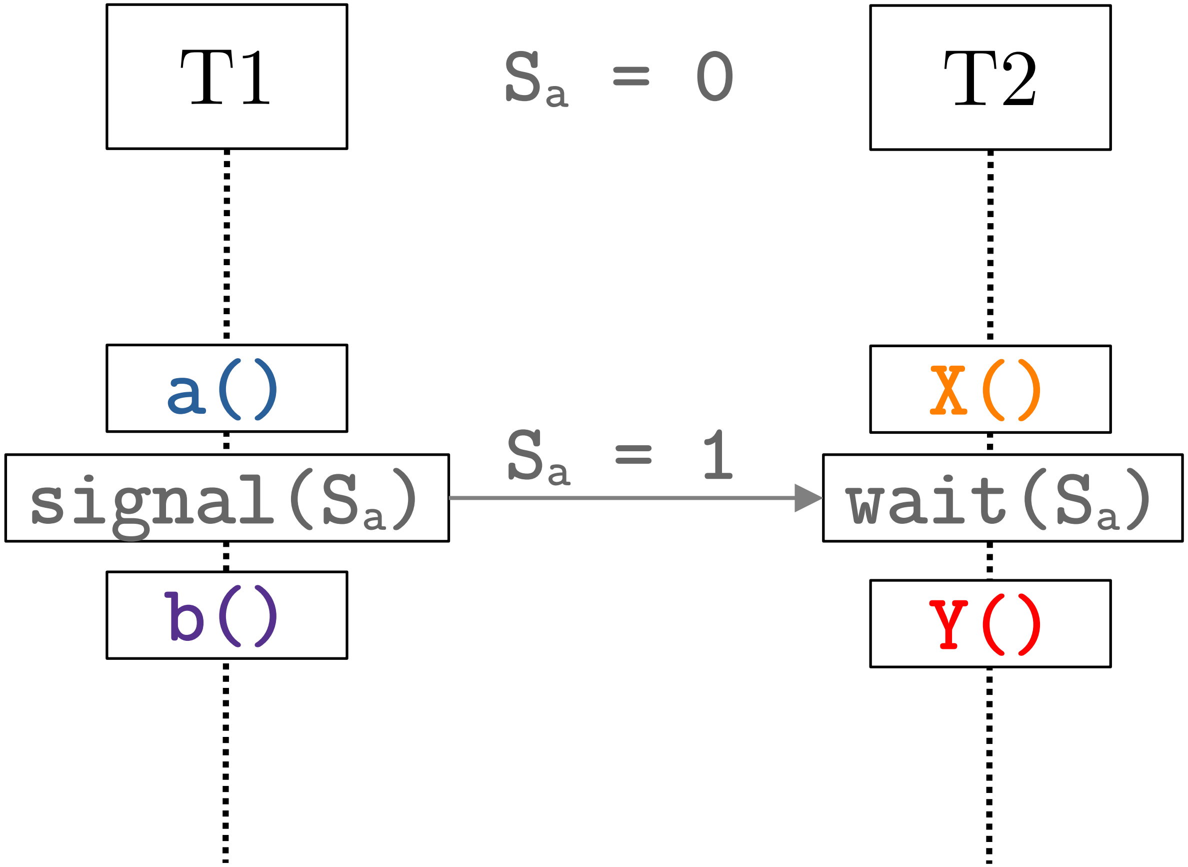


A *barrier* should be placed before the call `Y()` in T2.

This barrier means T2 has to wait for T1 to complete the `a()` call before performing `Y()`.

Constraint: We consider only those orders where `a()` is performed before `Y()` as VALID.

Solution 1a: Barrier: `a() < Y()`



S_a represents the **‘Y’** *permit*.
Initially, S_a = 0.

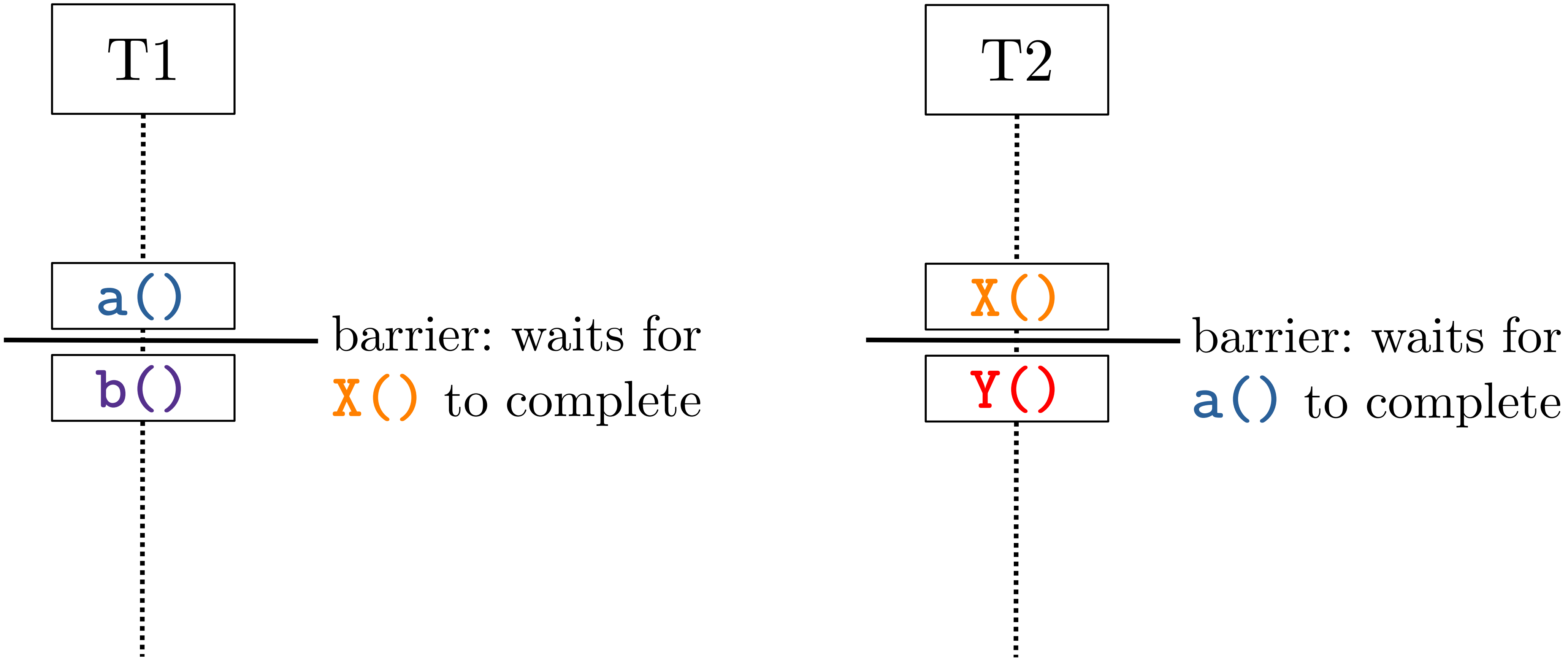
T2 can only exit `wait(`S_a`)` and call `Y()` if there is at least *1 permit*.

The *1 permit* is only available when **T1** calls `signal(`S_a`)` after `a()` executes.

Synchronization Task/Problem 1b: **(Thread) Barriers**

[Solution Description]

Solution 1b: Barriers: `a() < Y(), X() < b()`



T1

`a()`

`b()`

barrier: waits for `X()` to complete

T2

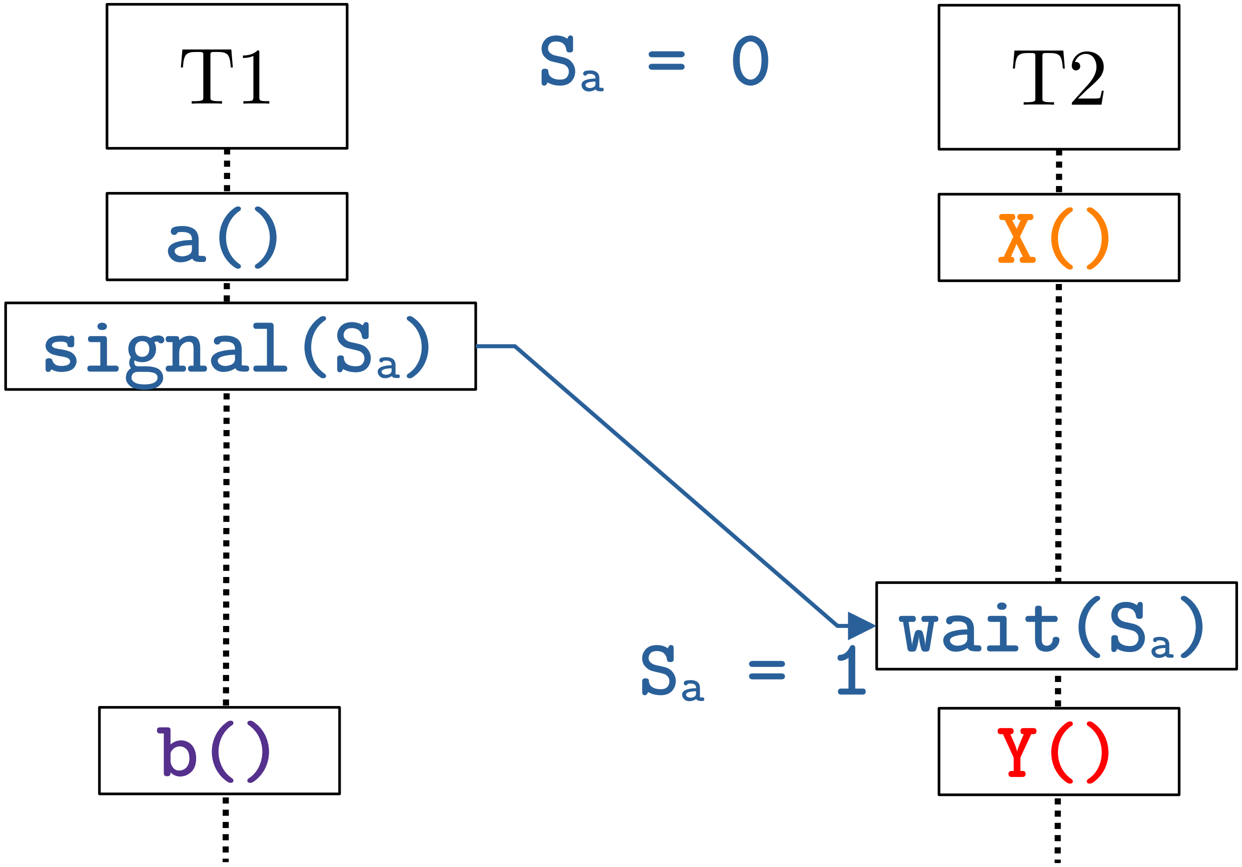
`X()`

`Y()`

barrier: waits for `a()` to complete

Constraint: We consider only those orders where `a()` is performed before `Y()` and `X()` is performed before `b()` as **VALID**.

Solution 1b: Barriers: `a() < Y(), X() < b()`

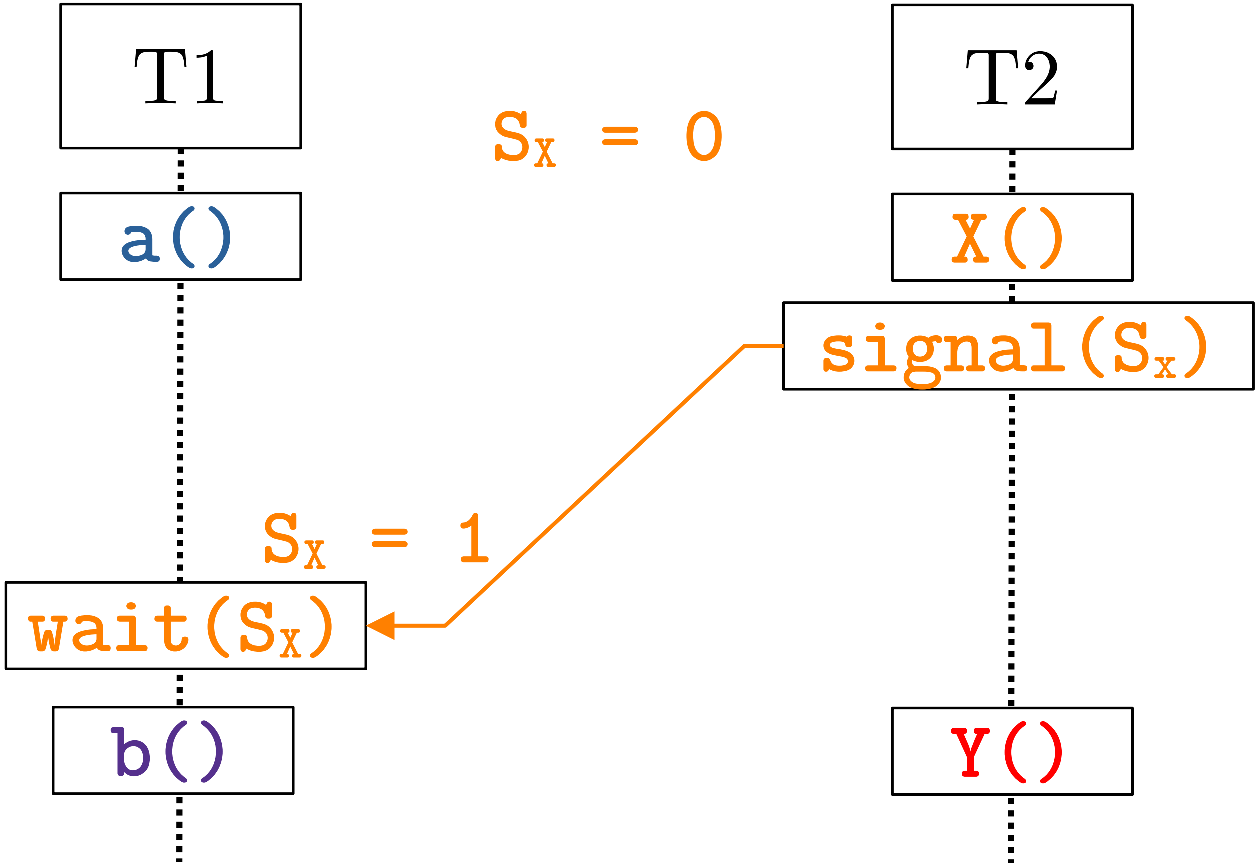


S_a represents the 'Y' *permit*. Initially, S_a = 0.

T2 can only exit `wait(`S_a`)` and call `Y()` if there is at least *1 permit*.

The *1 permit* is only available when **T1** calls `signal(`S_a`)` after `a()` executes.

Solution 1b: Barriers: `a() < Y(), X() < b()`

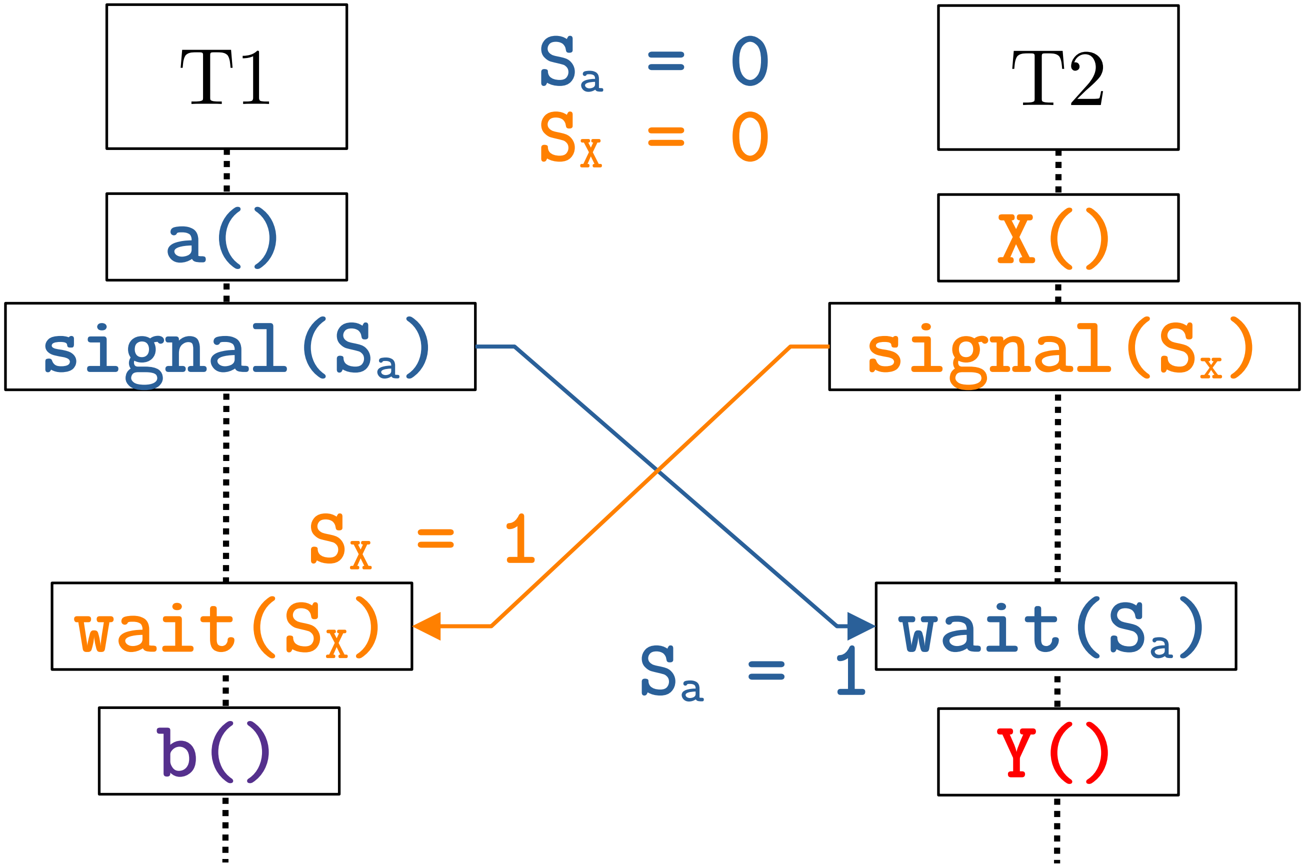


S_X represents the ‘b’ *permit*. Initially, S_X = 0.

T1 can only exit wait(S_X) and call b() if there is at least *1 permit*.

The *1 permit* is only available when **T2** calls signal(S_X) after X() executes.

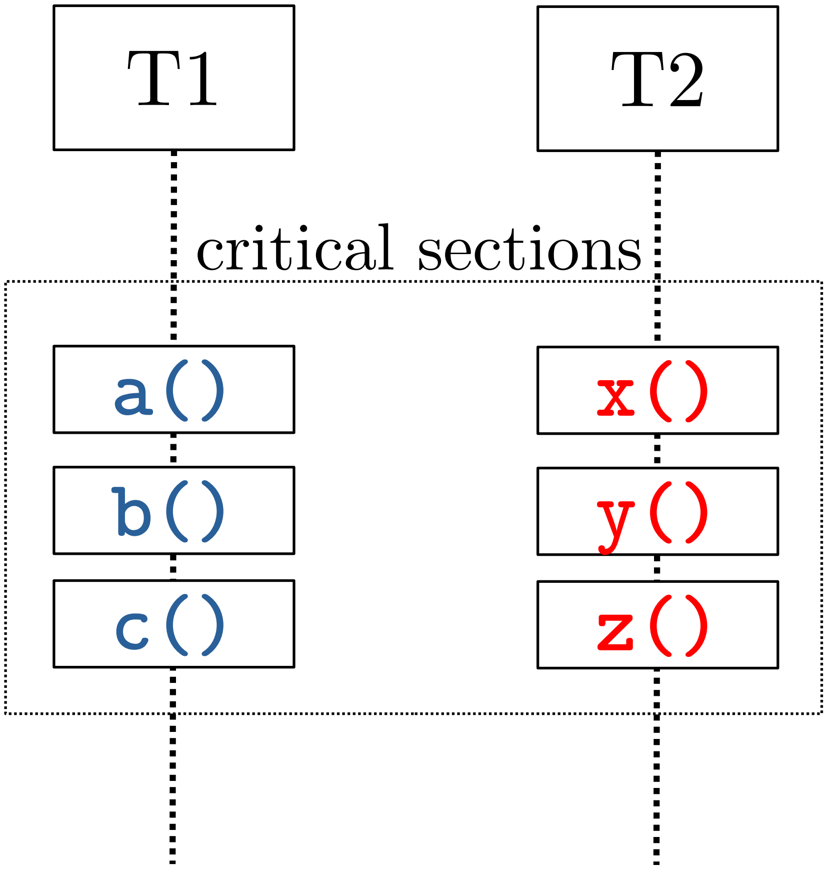
Solution 1b: Barriers: a() < Y(), X() < b()



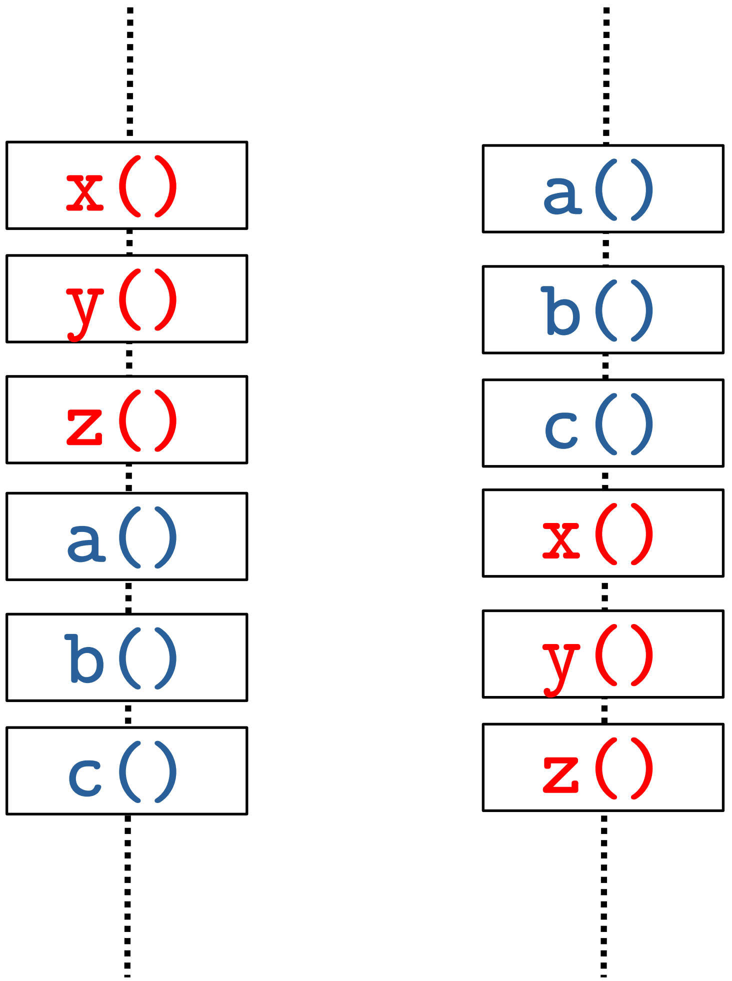
Synchronization Task/Problem 2:
Mutual Exclusion

[Problem & Solution Descriptions]

Task/Problem 2: Mutual Exclusion

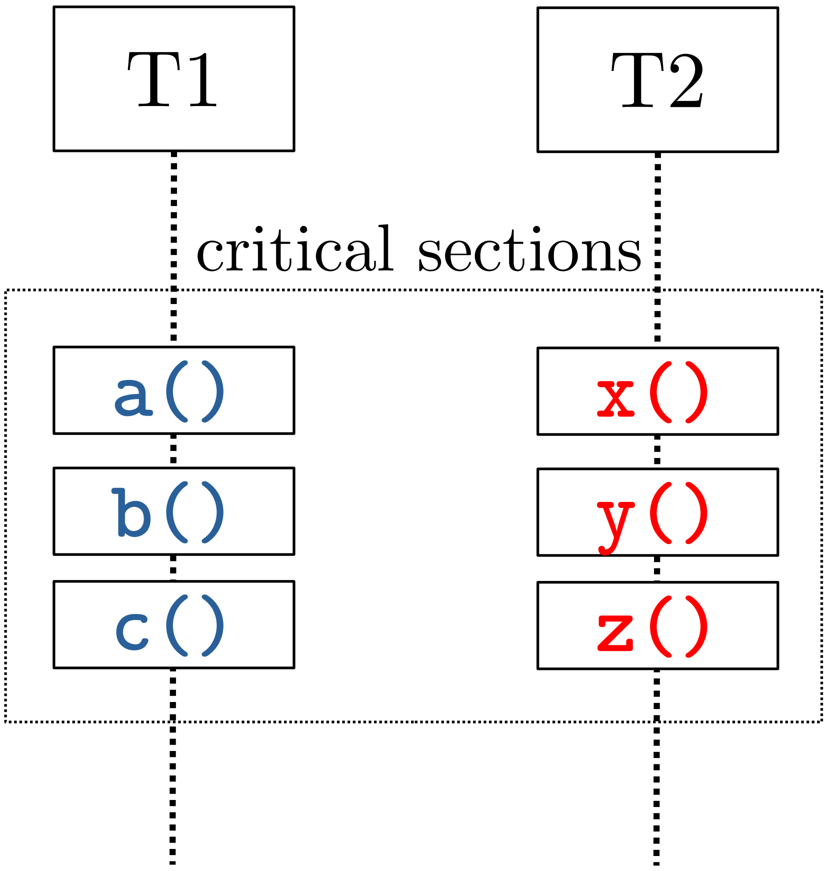


VALID orders



Constraint: Instructions/Operations in T1 and T2's critical sections should NOT interleaved.

Task/Problem 2: Mutual Exclusion



T1

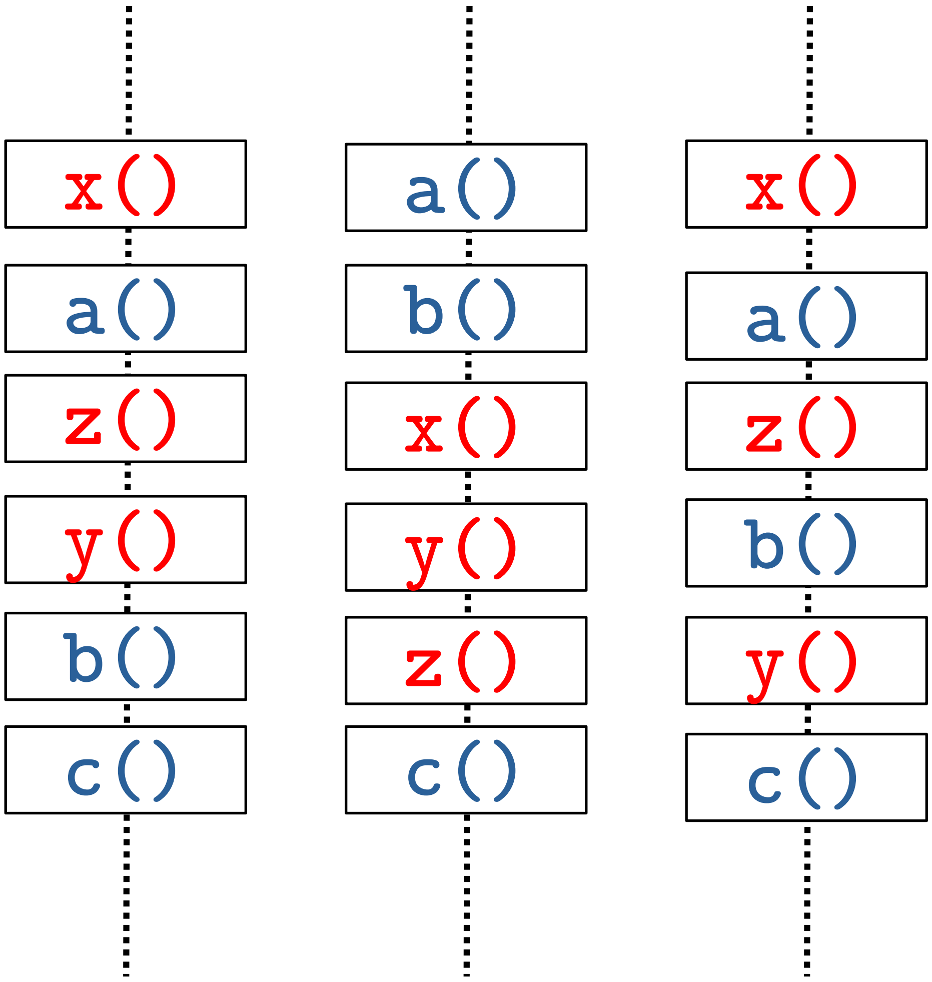
T2

critical sections

a()
b()
c()

x()
y()
z()

some INVALID orders



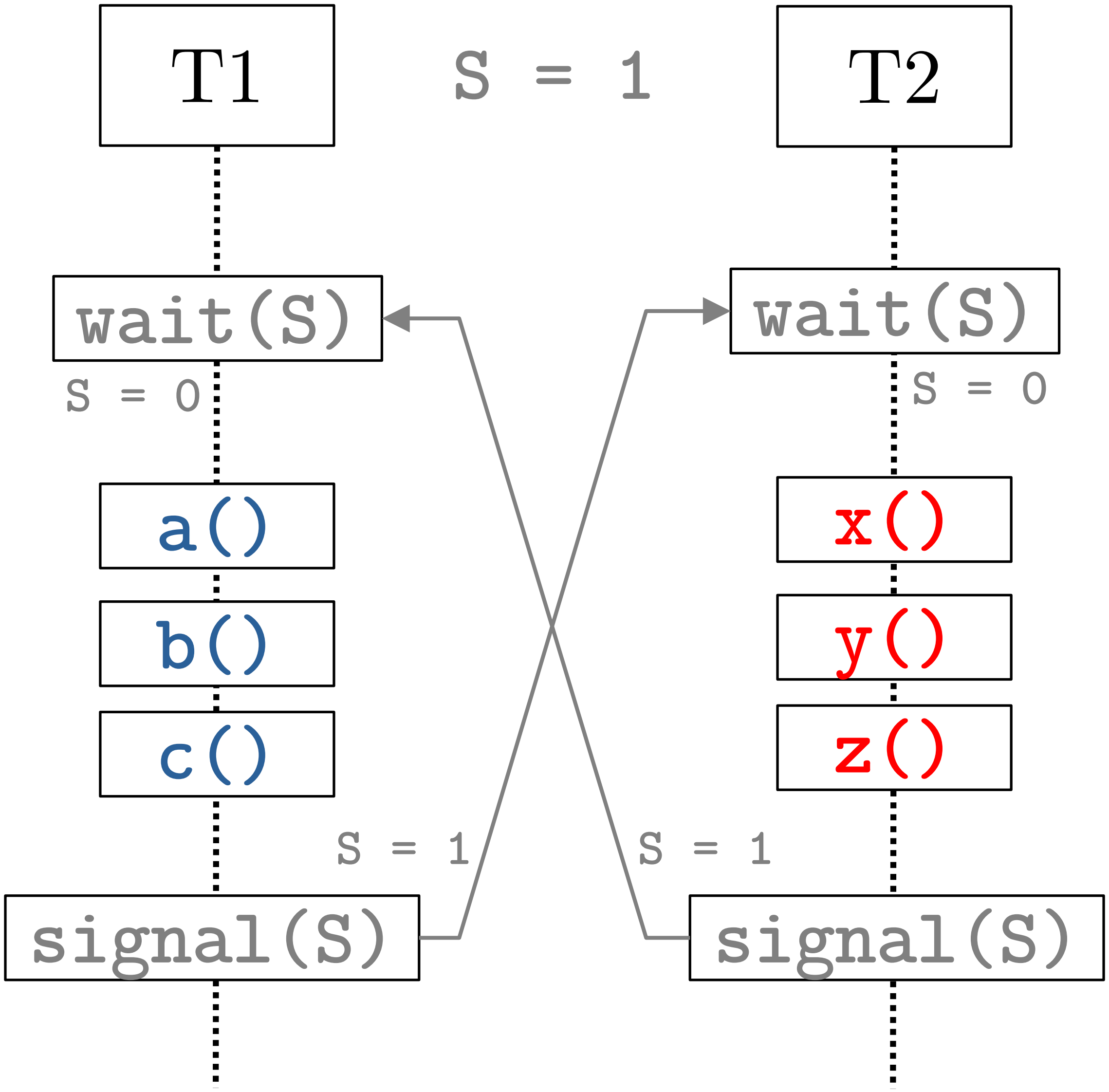
x()
a()
z()
y()
b()
c()

a()
b()
x()
y()
z()
c()

x()
a()
z()
b()
y()
c()

Constraint: Instructions/Operations in T1 and T2's critical sections should NOT interleaved.

Solution 2: Mutual Exclusion

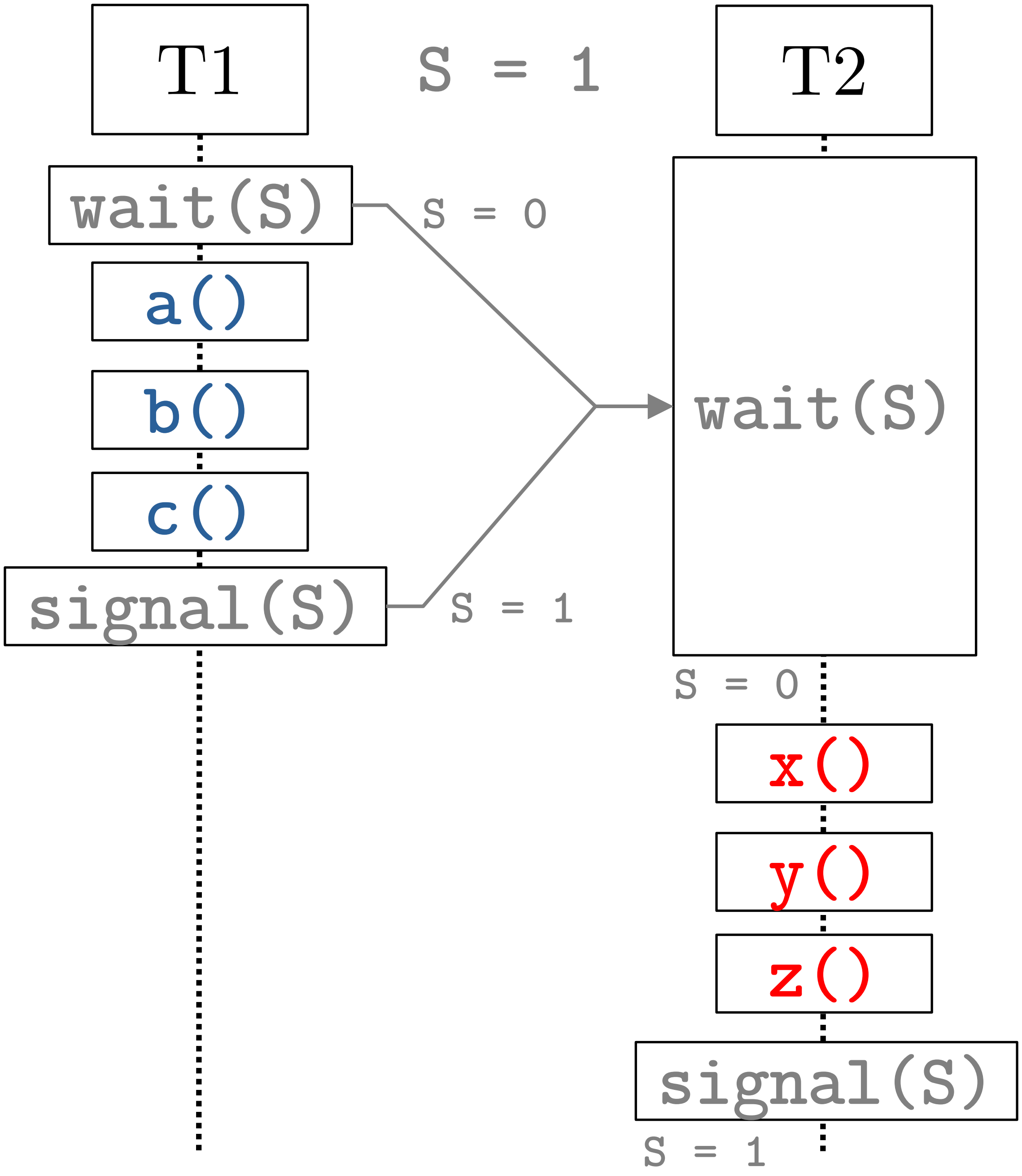


`S = 1` means *1 permit* is available.

Only one thread, **T1** or **T2**, will be given the permit and can exit the `wait(S)` loop, set `S = 0`, and enter its critical section.

After executing its critical section, the thread will now call `signal(S)` and set `S = 1` which gives the other thread a permit to enter its critical section.

Solution 2: Mutual Exclusion

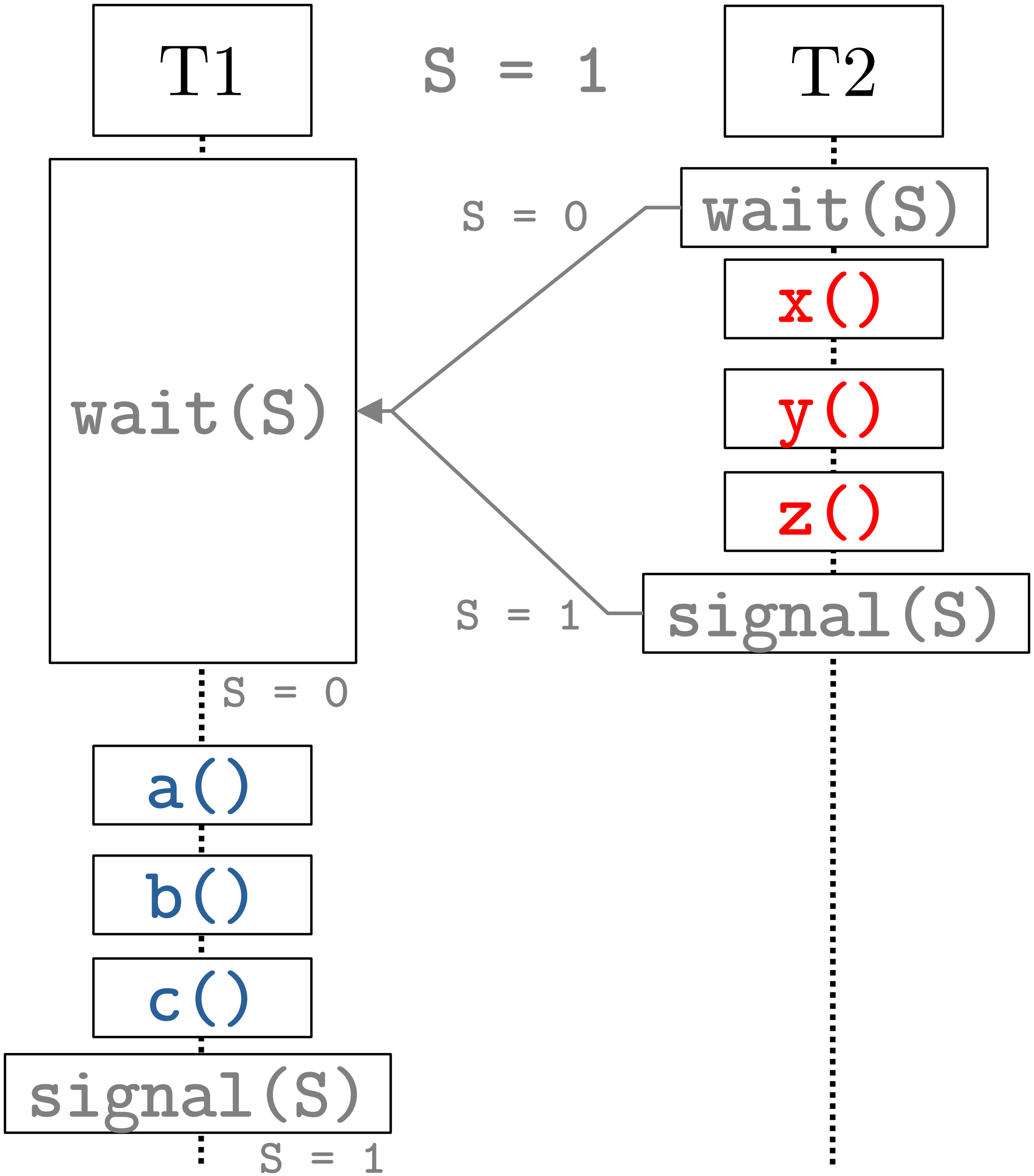


`S = 1` means *1 permit* is available.

Only one thread, **T1** or **T2**, will be given the permit and can exit the `wait(S)` loop, set `S = 0`, and enter its critical section.

After executing its critical section, the thread will now call `signal(S)` and set `S = 1` which gives the other thread a permit to enter its critical section.

Solution 2: Mutual Exclusion



`S = 1` means *1 permit* is available.

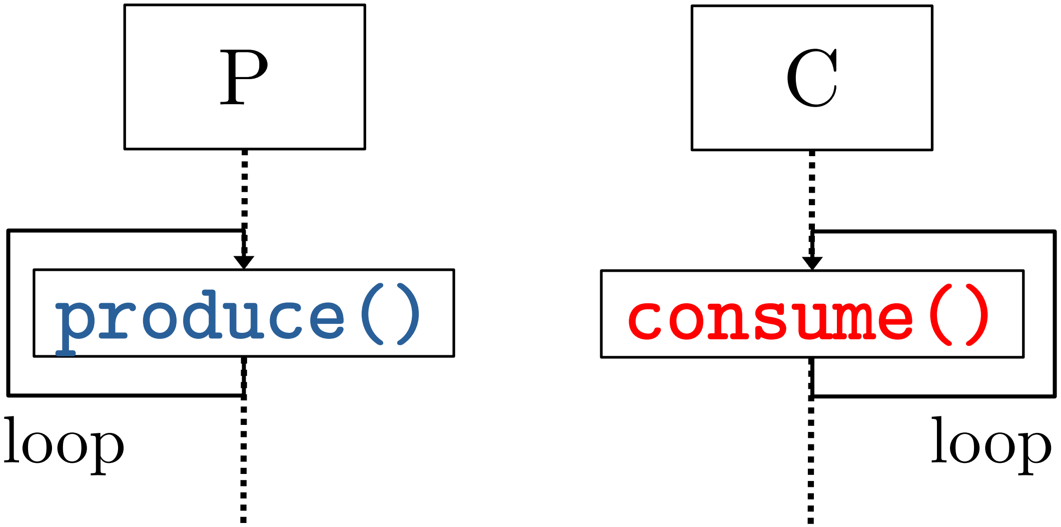
Only one thread, **T1** or **T2**, will be given the permit and can exit the `wait(S)` loop, set `S = 0`, and enter its critical section.

After executing its critical section, the thread will now call `signal(S)` and set `S = 1` which gives the other thread a permit to enter its critical section.

Synchronization Task/Problem 3a: **Producer-Consumer**

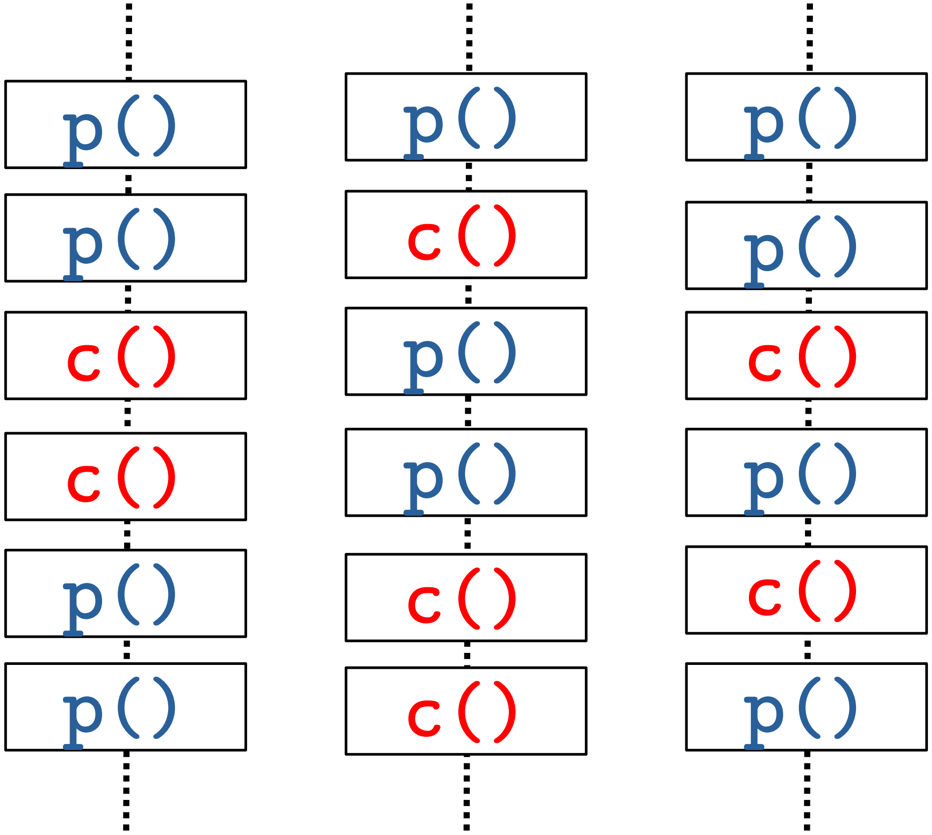
[Problem & Solution Descriptions]

Task/Problem 3a: Producer(p)-Consumer(c) Problem



c() = consume()
p() = produce()

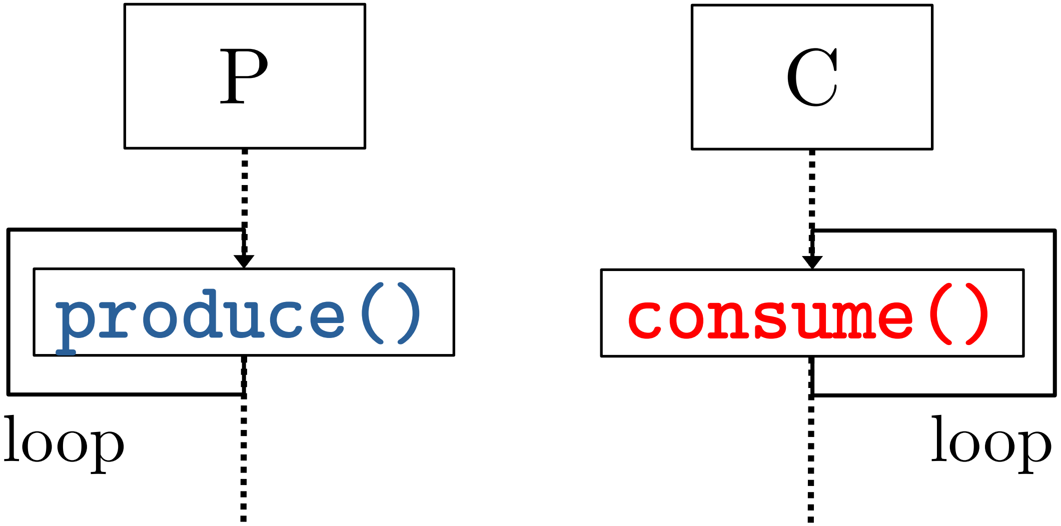
some VALID orders



p()	p()	p()
p()	c()	p()
c()	p()	c()
c()	p()	p()
p()	c()	c()
p()	c()	p()

Constraint: Before a consume() call can occur a corresponding produce() call should have occurred first.

Task/Problem 3a: Producer(p)-Consumer(c) Problem



P

produce()

loop

C

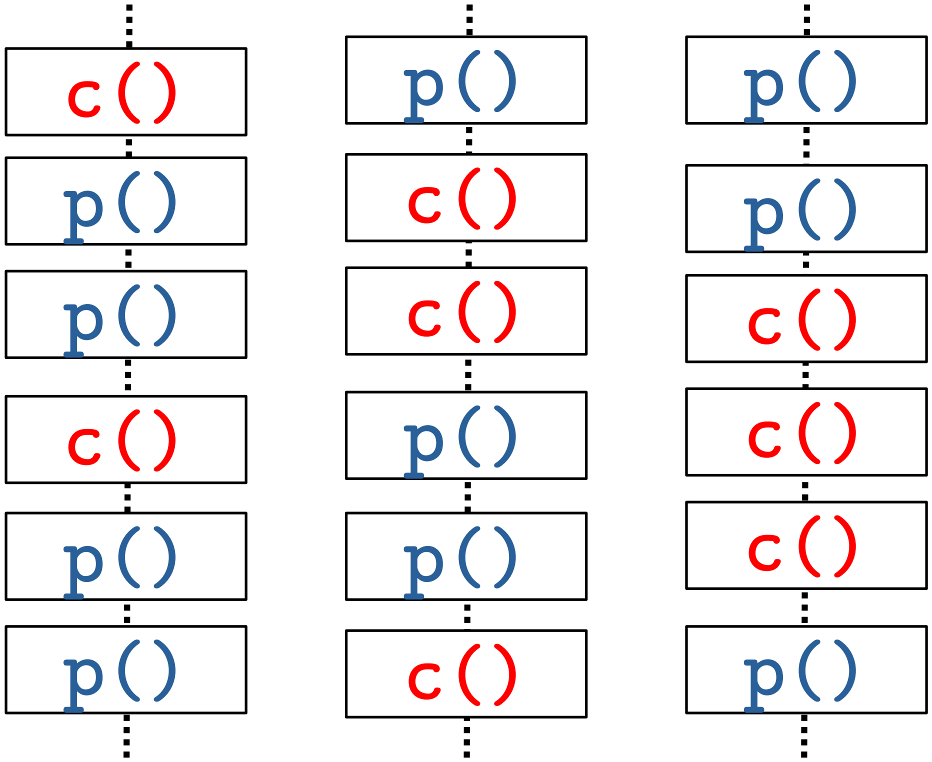
consume()

loop

c() = consume()

p() = produce()

some INVALID orders



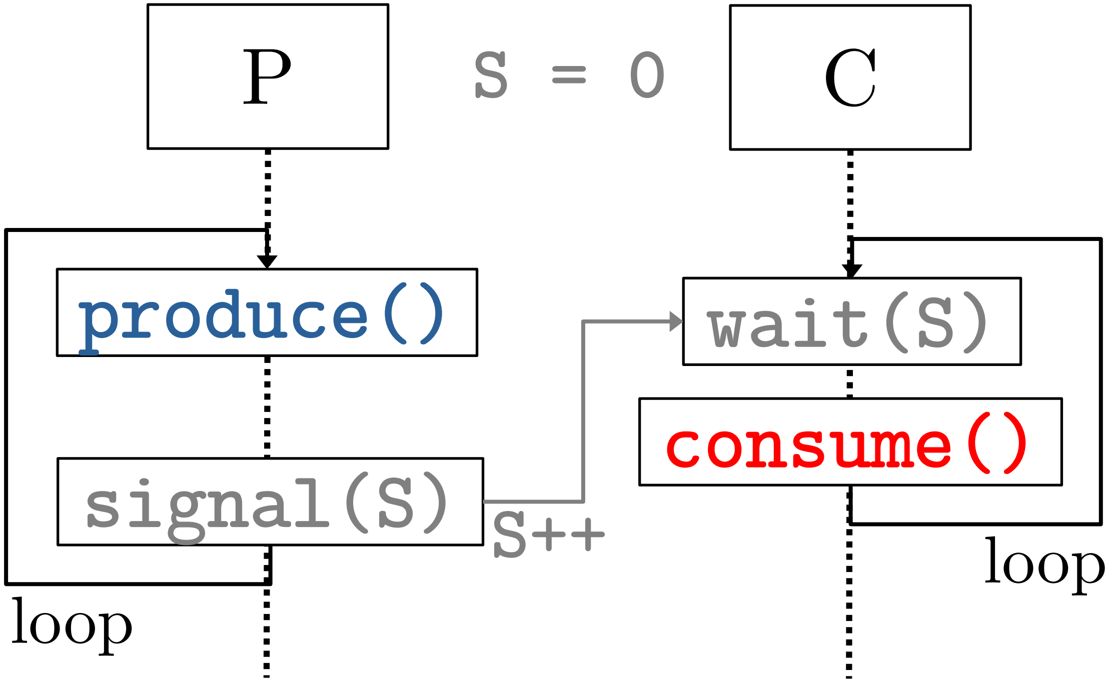
c()
p()
p()
c()
p()
p()

p()
c()
c()
p()
p()
c()

p()
p()
c()
c()
c()
p()

Constraint: Before a consume() call can occur a corresponding produce() call should have occurred first

Task/Problem 3a: Producer(p)-Consumer(c) Problem



S = 0 means 0 *permit* is available.

Thread **C** is the one calling wait(S) and is waiting for a permit to call consume().

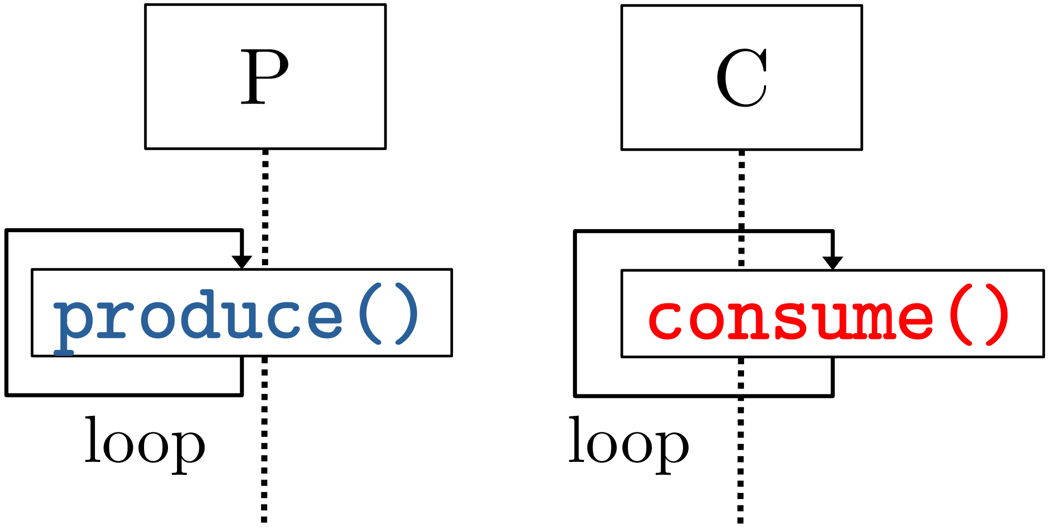
Thread **P** will call signal(S) only after calling produce(). signal(S) will increment S.

Thread C can only call consume() if there is a corresponding call to produce() prior.

Synchronization Task/Problem 3b: **Bounded Buffer**

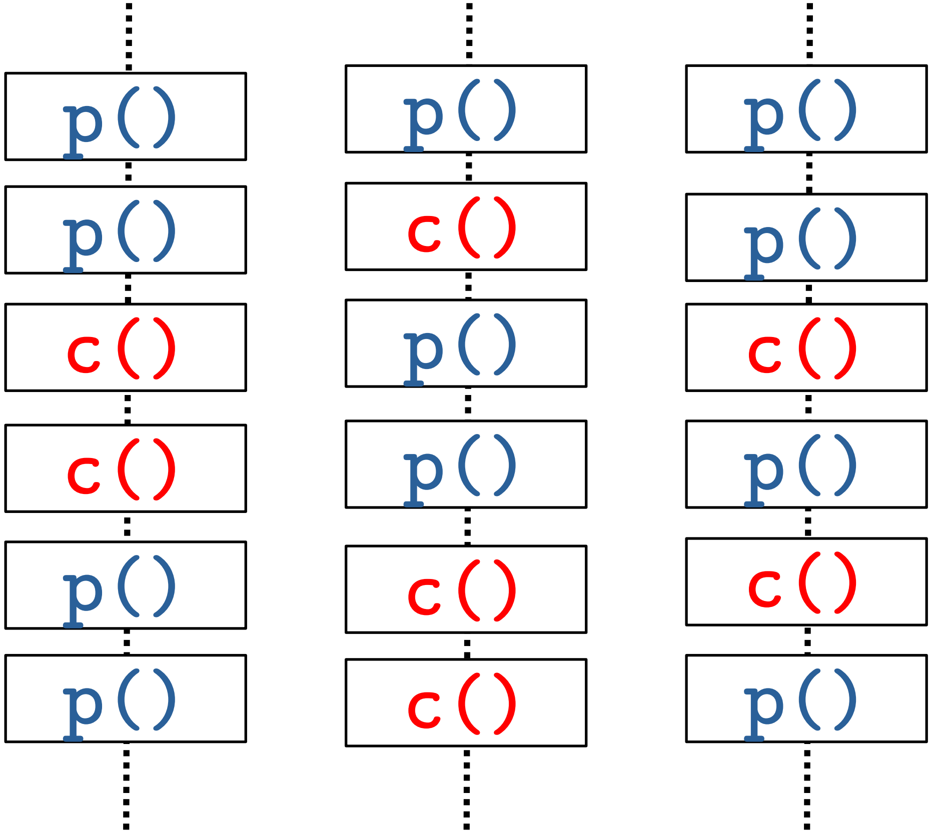
[Problem & Solution Descriptions]

Task/Problem 3b: Bounded Buffer. i.e. Buffer Size = 3



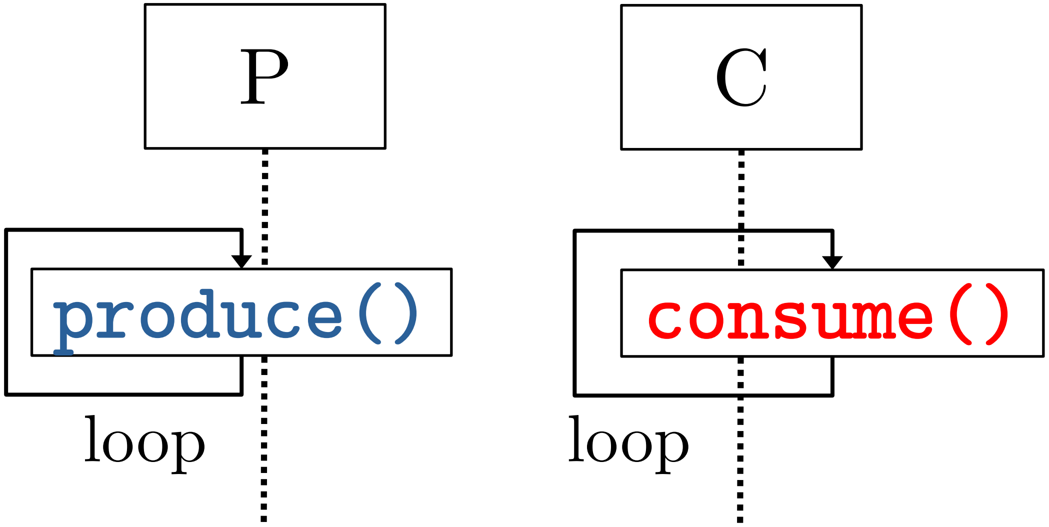
c() = consume()
p() = produce()

some VALID orders



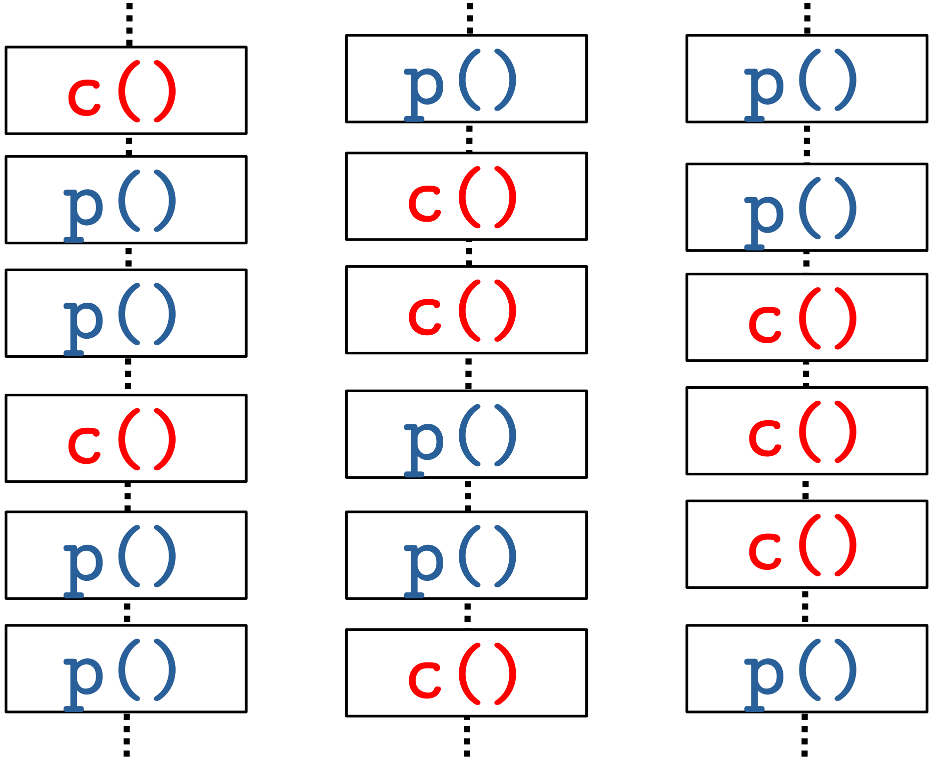
Constraint: Before a consume() call can occur a corresponding produce() call should have occurred first. At **most 3 calls** to produce() can occur that have no corresponding calls to consume(). i.e. Buffer is full after 3 calls to produce() (and 0 calls to consume()).

Task/Problem 3b: Bounded Buffer. i.e. Buffer Size = 3



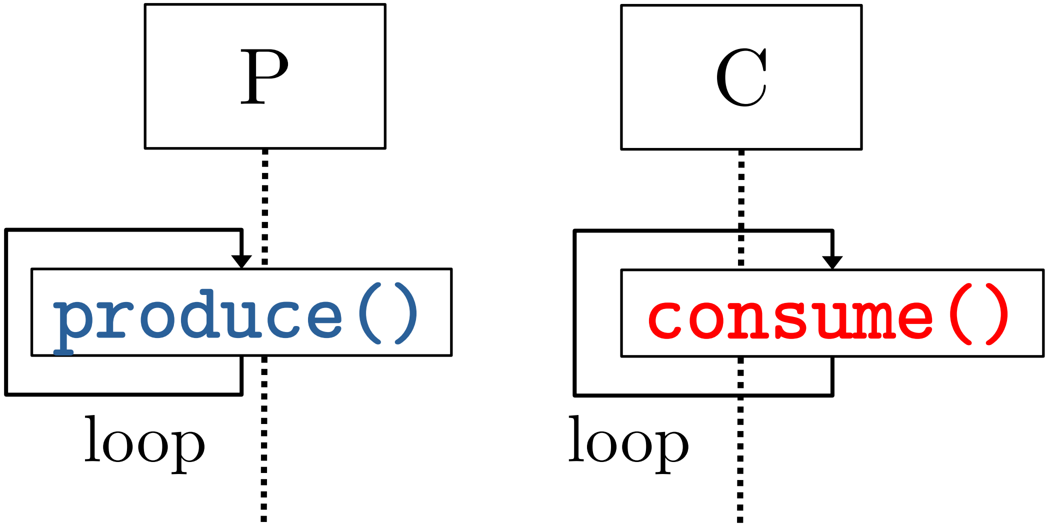
c() = consume()
p() = produce()

some INVALID orders



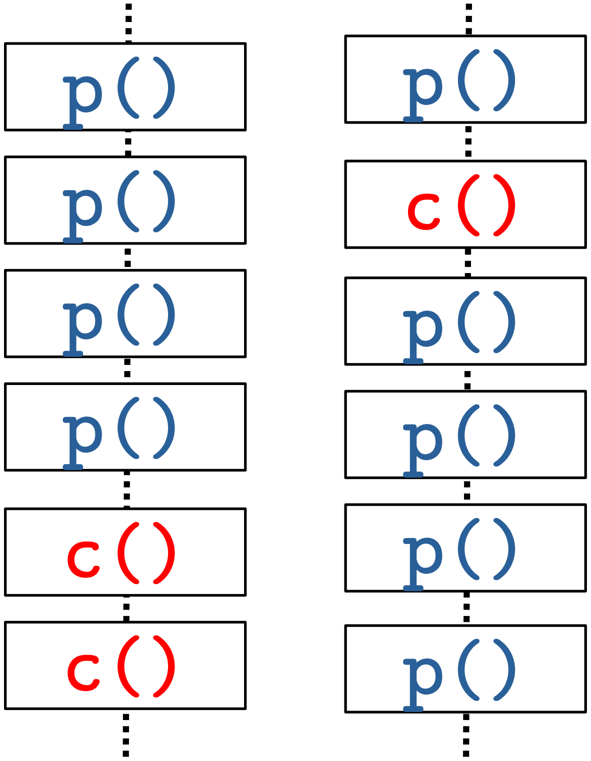
Constraint: Before a consume() call can occur a corresponding produce() call should have occurred first. At **most 3 calls** to produce() can occur that have no corresponding calls to consume(). i.e. Buffer is full after 3 calls to produce().

Task/Problem 3b: Bounded Buffer. i.e. Buffer Size = 3



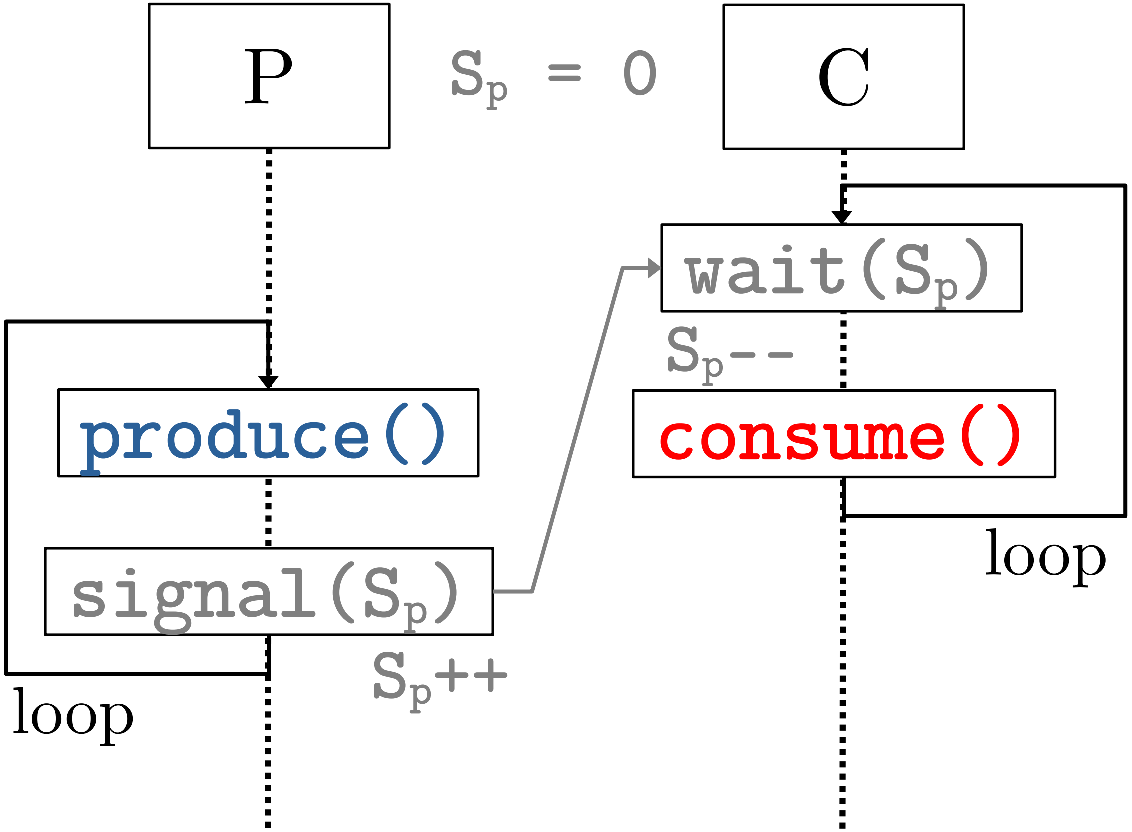
c() = consume()
p() = produce()

some INVALID orders



Constraint: Before a consume() call can occur a corresponding produce() call should have occurred first. At **most 3 calls** to produce() can occur that have no corresponding calls to consume(). i.e. Buffer is full after 3 calls to produce().

Task/Problem 3b: Bounded Buffer. i.e. Buffer Size = 3



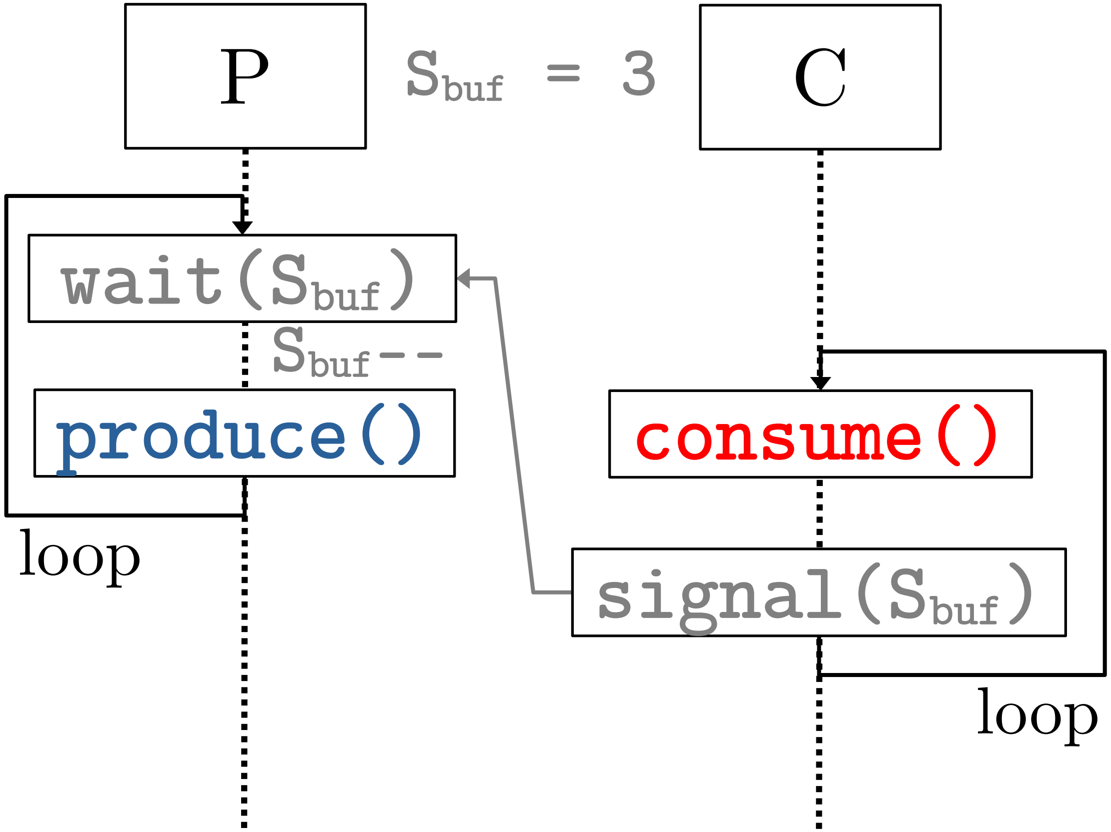
S_p = 0 means 0 ‘consume’ *permit* is available.

Thread **C** is the one calling wait(S_p) and is waiting for a permit before calling consume().

Thread **P** will call signal(S_p) only after calling produce(). signal(S_p) will increment S_p.

Thread C can only call consume() if there is a corresponding call to produce() prior.

Task/Problem 3b: Bounded Buffer. i.e. Buffer Size = 3



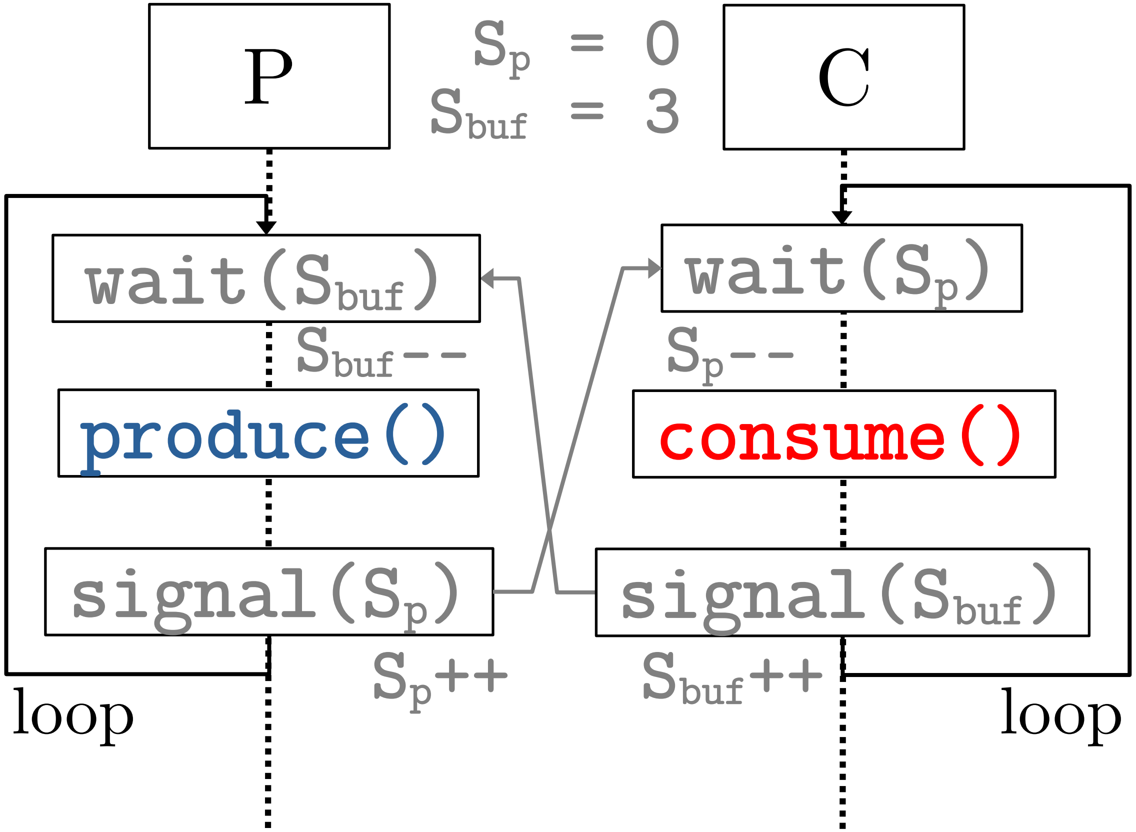
S_{buf} = 3 means 3 ‘produce’ *permit* is available.

Thread **P** is the one calling wait(S_{buf}) and is waiting for a permit before calling produce().

Thread **C** will call signal(S_p) only after calling consume() . signal(S_p) will increment S_p.

Thread C can only call consume() if there is a corresponding call to produce() prior.

Task/Problem 3b: Bounded Buffer. i.e. Buffer Size = 3



S_p represents the number of 'consume' *permits*.

S_{buf} represents the number of 'produce' *permits*.

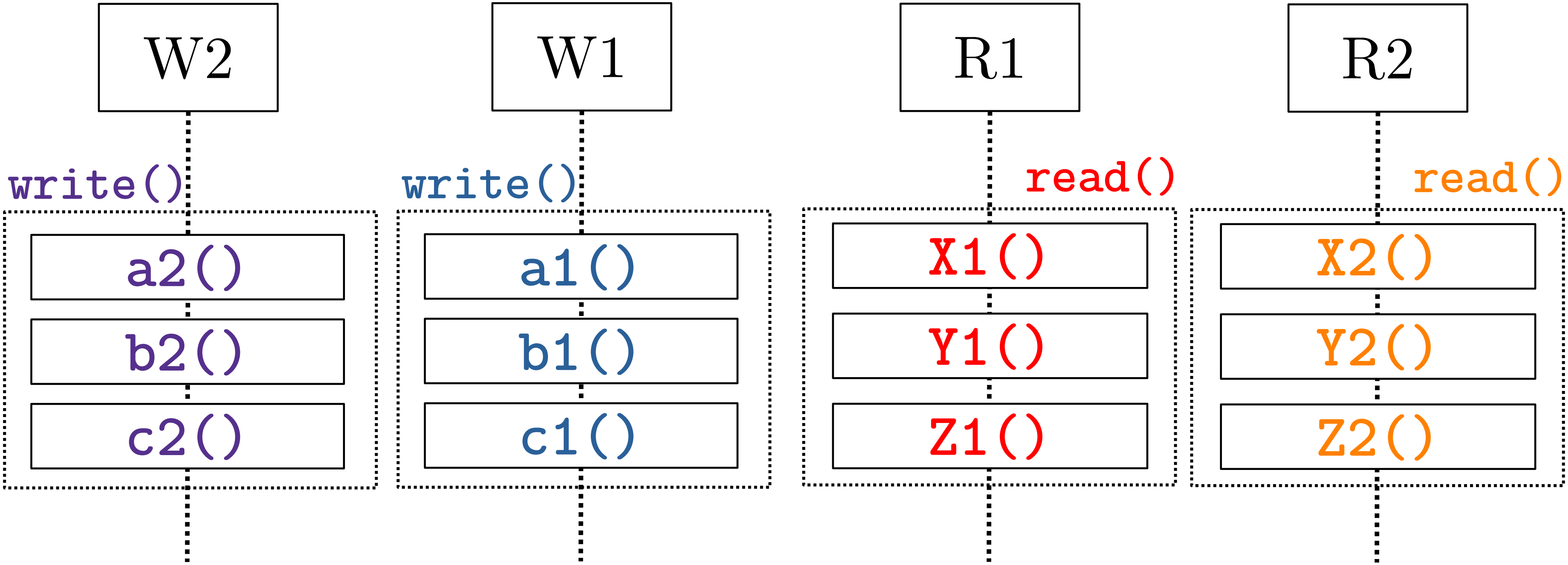
Thread **P** calls wait(S_{buf}) to wait for a 'produce' *permit* before calling produce(). After calling produce(), thread **P** calls signal(S_p) to increment the consume permit.

Thread **C** calls wait(S_p) to wait for a 'consume' *permit* before calling consume(). After calling consume(), thread **C** calls signal(S_{buf}) to increment the produce permit.

Synchronization Task/Problem 4: **Readers-Writers**

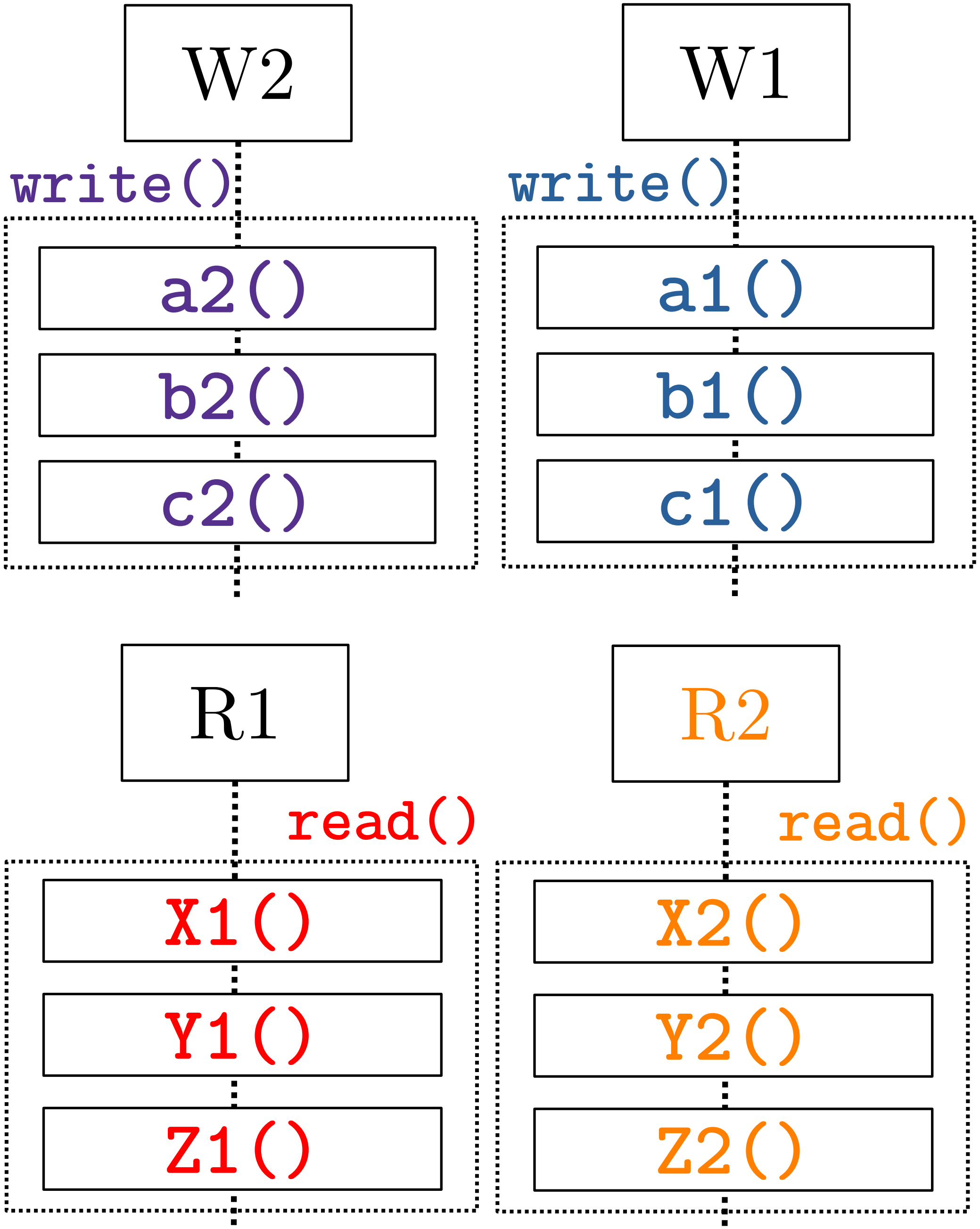
[Problem & Solution Descriptions]

Task/Problem 4: Reads-Writers



Constraint: A **writer thread** has exclusive access. No other threads (writer or reader) can access their `write()` or `read()` function if there is a writer thread calling `write()`. **Reader threads**, as a group, can access call their `read()` function concurrently and no writer threads can call their `write()` function while at least one reader thread is calling `read()`.

Task/Problem 4: Reads-Writers



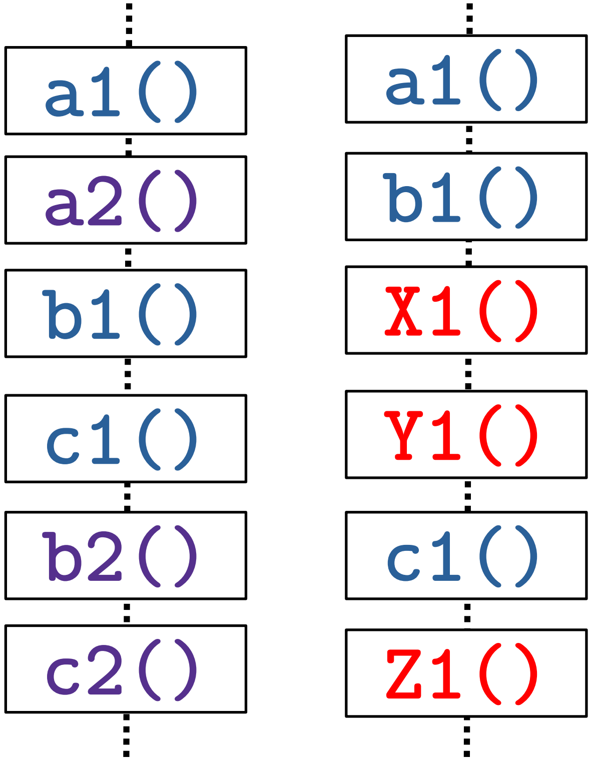
W2
write()
a2()
b2()
c2()

W1
write()
a1()
b1()
c1()

R1
read()
X1()
Y1()
Z1()

R2
read()
X2()
Y2()
Z2()

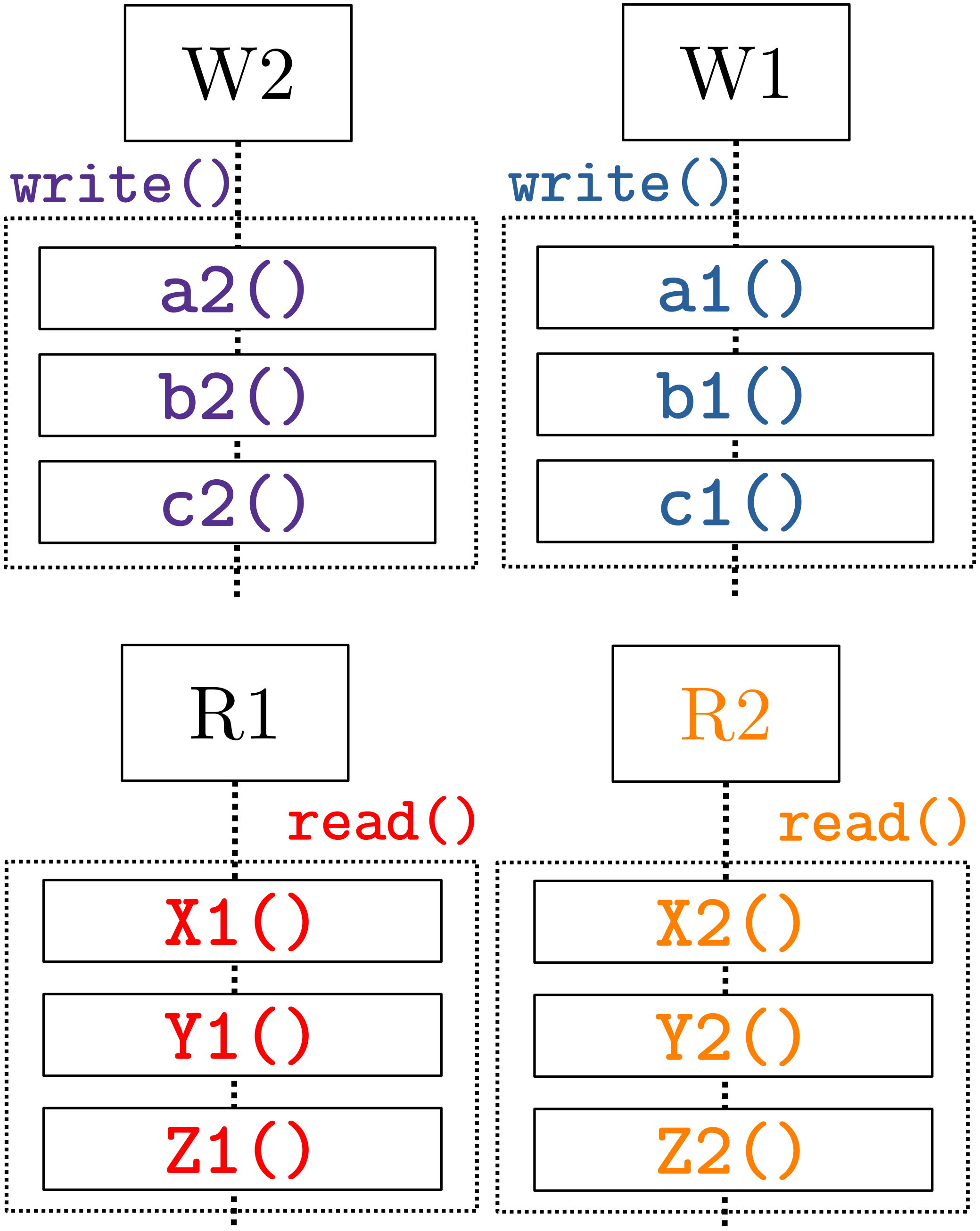
some INVALID orders



a1()
a2()
b1()
c1()
b2()
c2()

a1()
b1()
X1()
Y1()
c1()
Z1()

Task/Problem 4: Reads-Writers



W2

write()

a2()

b2()

c2()

W1

write()

a1()

b1()

c1()

R1

read()

X1()

Y1()

Z1()

R2

read()

X2()

Y2()

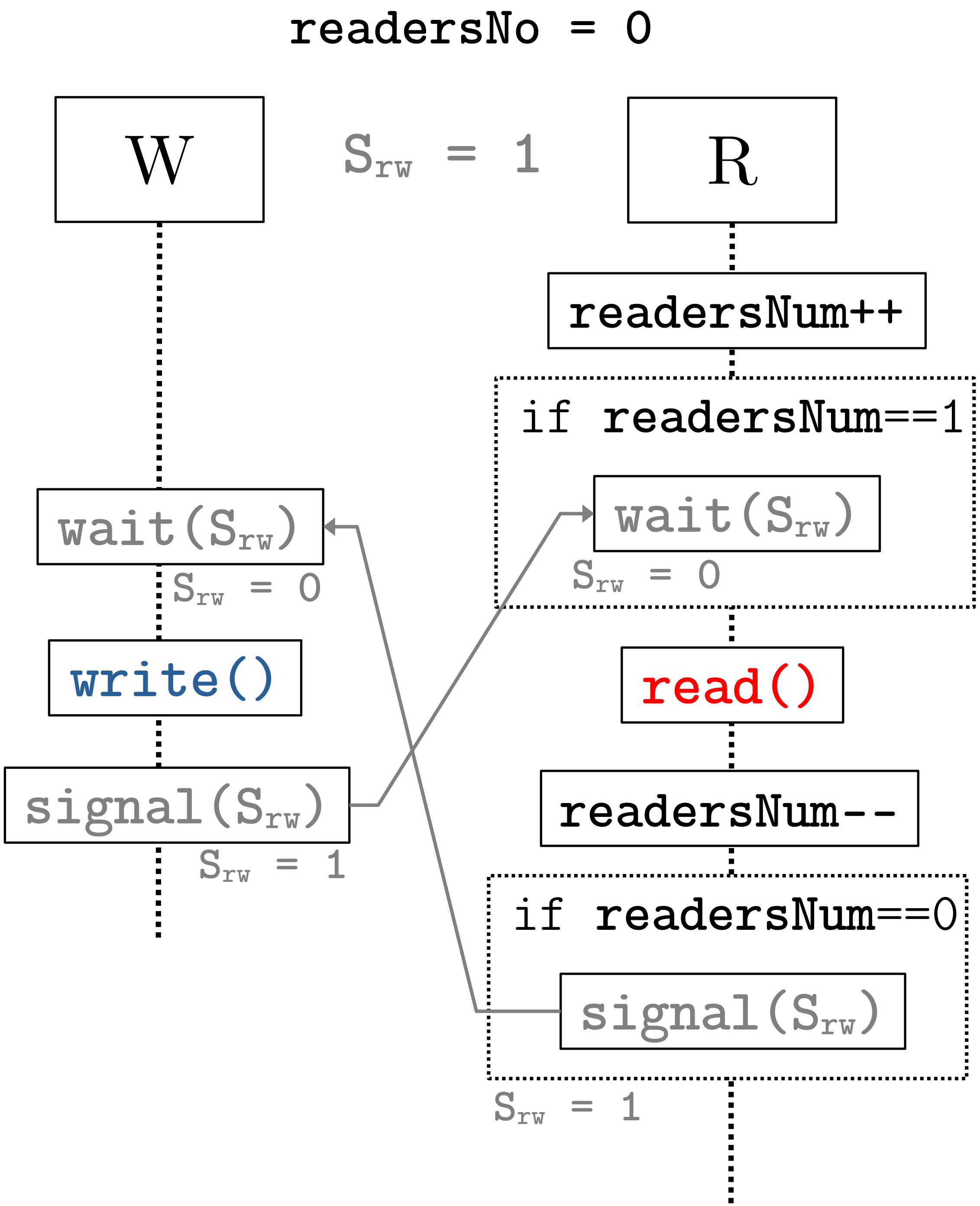
Z2()

some VALID orders

a1()
b1()
c1()
a2()
b2()
c2()

X2()
X1()
Y1()
Y2()
Z1()
Z2()
a1()
b1()
c1()

Solution 4: Reads-Writers

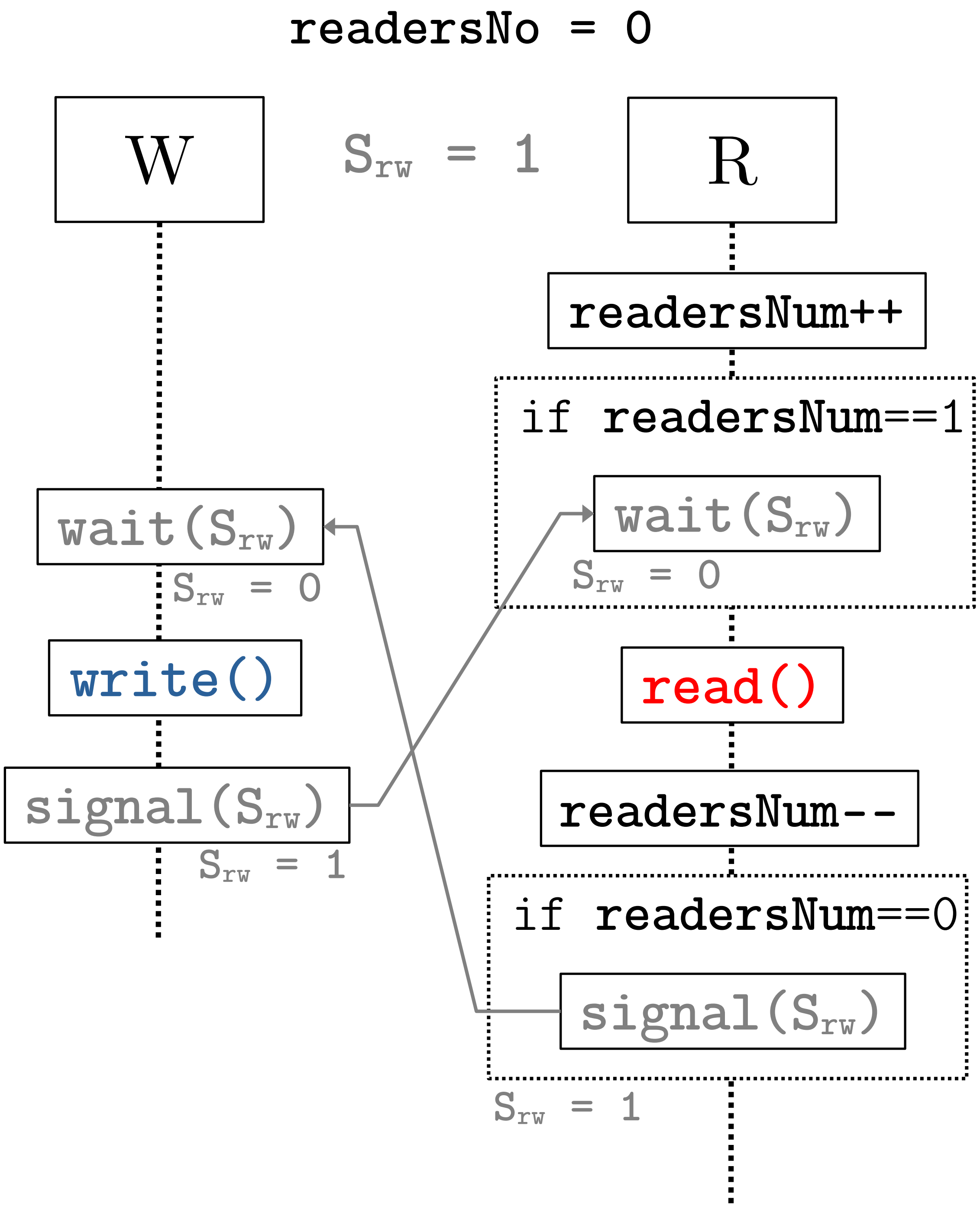


S_{rw} represents the read-write *permit*. Initially, S_{rw} = 1.

Writer **W** calls wait(S_{rw}) to wait for a read-write *permit* before calling write(). After calling write(), writer **W** calls signal(S_{rw}) to release the read-write *permit*. S_{rw} = 1

Reader **R** increments the variable readersNo. If readersNo==1 after being incremented, then Reader R is the first reader thread trying to perform the read() operation.

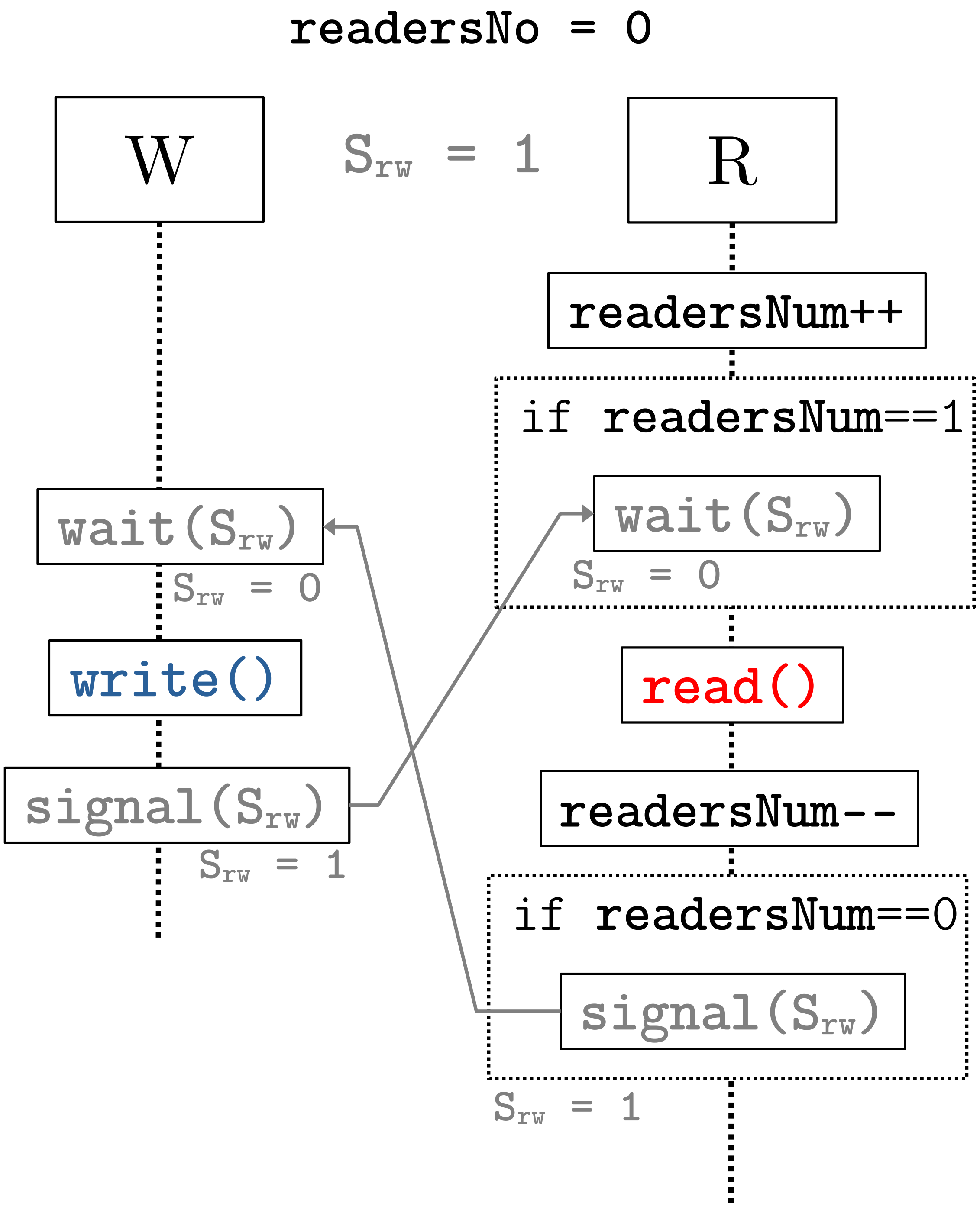
Solution 4: Reads-Writers



Reader **R** increments the variable `readersNum`. If `readersNum==1` after being incremented, then reader **R** is the first reader thread trying to perform the `read()` operation. If it is the first reader thread, then it will call `wait(`S_{rw}`)` to wait for the `read`-`write` *permit*.

If reader **R** is not the first reader (i.e. `readersNum>1`), then it does not need to wait for the `read`-`write` *permit* because some reader thread is currently calling `read()` has acquired it previously.

Solution 4: Reads-Writers

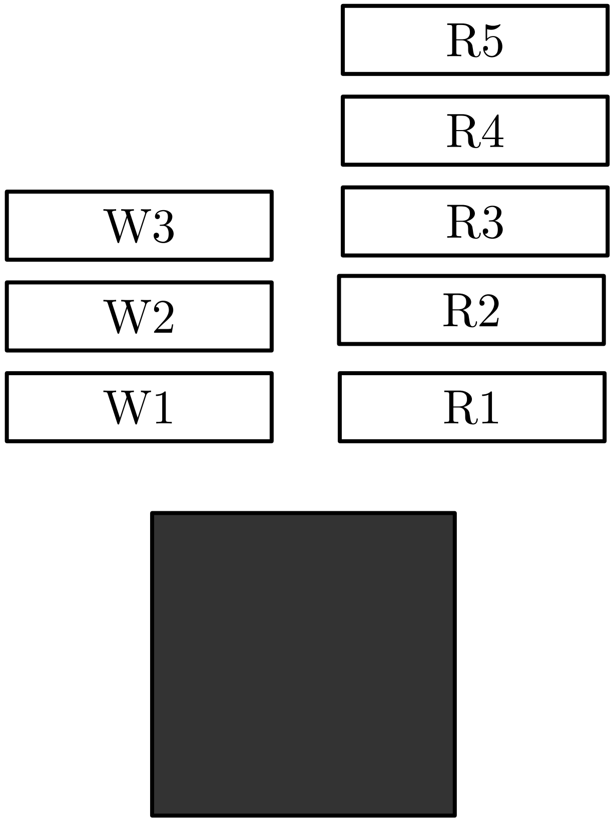


If reader **R** is not the first reader (i.e. `readersNum>1`), then it does not need to wait for the `read`-`write` *permit* because some reader thread is currently calling `read()` has acquired it previously.

After the call to `read()`, reader **R** will decrement `readersNum`, and it will release the `read`-`write` *permit* if it is the last reader that performed `read()`. i.e. If `readersNum==0`.

Solution 4: Reads-Writers: Lightswitch Design Pattern

Threads waiting for room access



Room for reading/writing

Lights OFF = Free Room

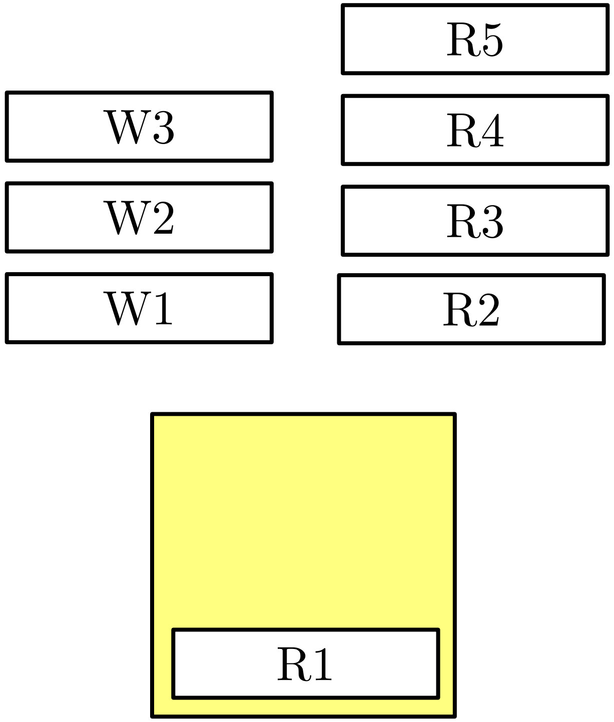
Semaphore S_{rw}=1

Analogy:

- The room is for reading/writing.
- Free Room – Lights OFF
- Room in Use – Lights ON
- Semaphore $S_{rw} = 1$ means a free room (lights OFF)
- Semaphore $S_{rw} = 0$ means a room in use (lights ON)
- Threads want to enter the room in order to `read()` or `write()`.

Solution 4: Reads-Writers: Lightswitch Design Pattern

Threads waiting for room access



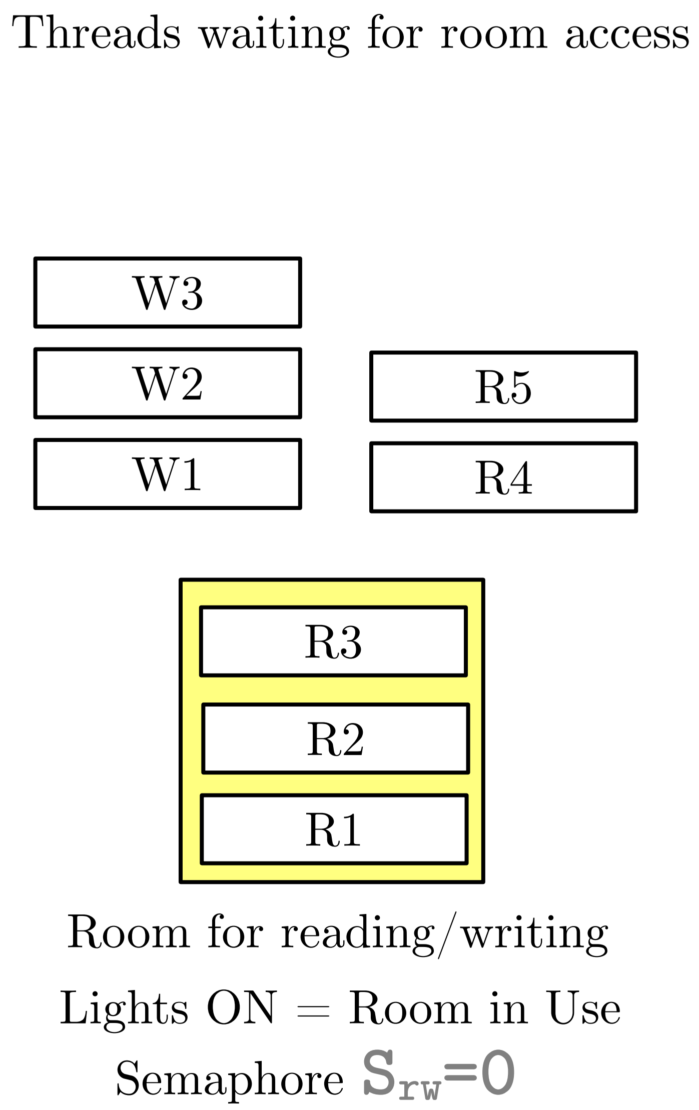
Room for reading/writing

Lights ON = Room in Use

Semaphore S_{rw}=0

- R1 enters the room and turns ON the light.
- S_{rw}=0 → **lights ON** → **Room in Use**
- When a reader is already in the room, other readers can enter the rule and perform `read()`.

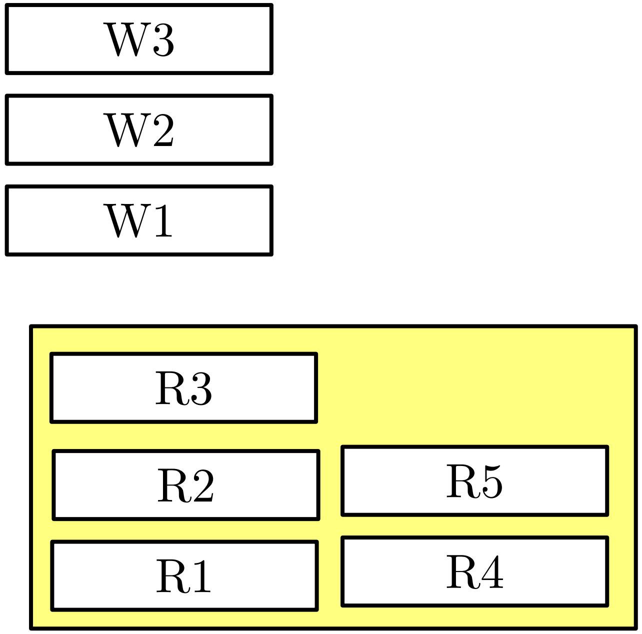
Solution 4: Reads-Writers: Lightswitch Design Pattern



- R1 enters the room and turns ON the light.
- S_{rw}=0 → **lights ON** → **Room in Use**
- When a reader is already in the room, other readers can enter the rule and perform `read()`.

Solution 4: Reads-Writers: Lightswitch Design Pattern

Threads waiting for room access



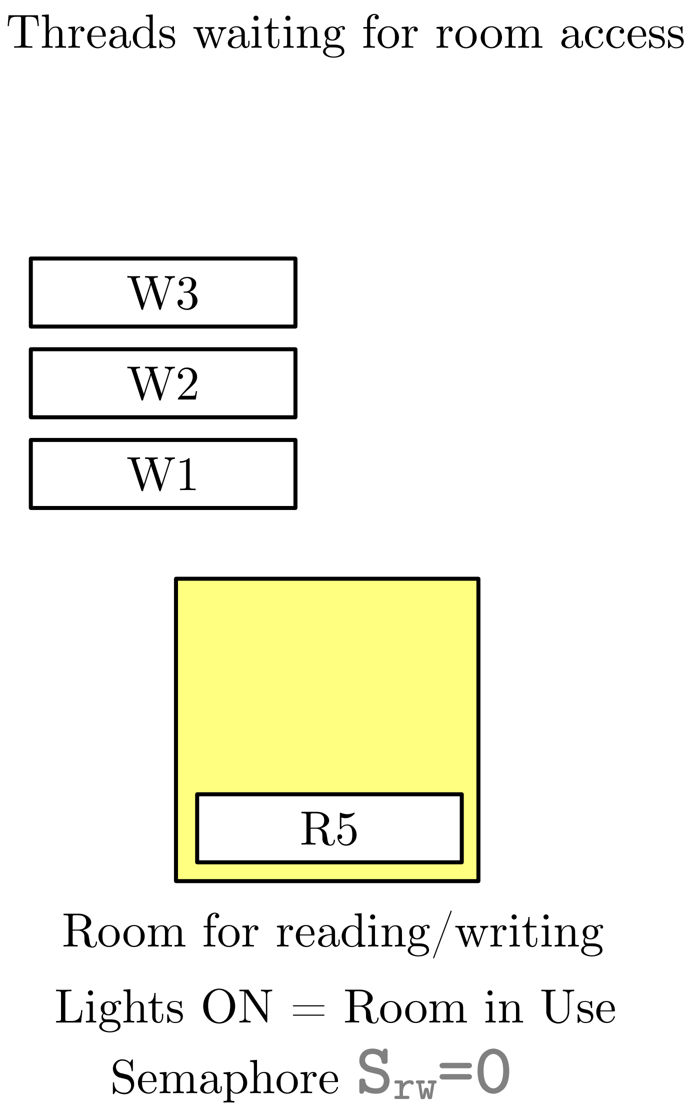
Room for reading/writing

Lights ON = Room in Use

Semaphore S_{rw}=0

- R1 enters the room and turns ON the light.
- S_{rw}=0 → **lights ON** → **Room in Use**
- When a reader is already in the room, other readers can enter the rule and perform `read()`.

Solution 4: Reads-Writers: Lightswitch Design Pattern

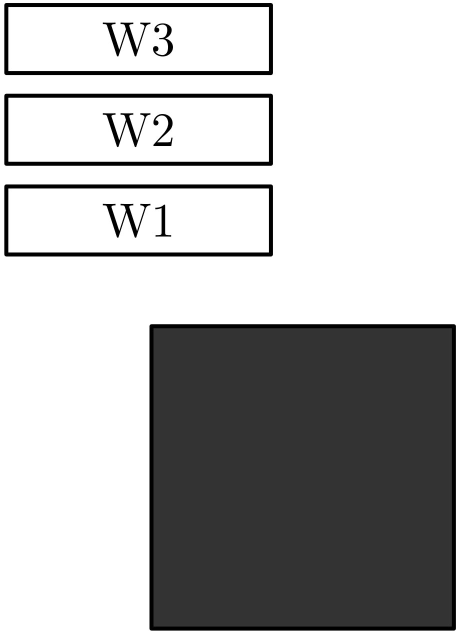


- The last reader in the room will be responsible for turning the light OFF.

- **lights OFF** → S_{rw}=0 → **Free Room**

Solution 4: Reads-Writers: Lightswitch Design Pattern

Threads waiting for room access



W3

W2

W1

Room for reading/writing

Lights OFF = Free Room

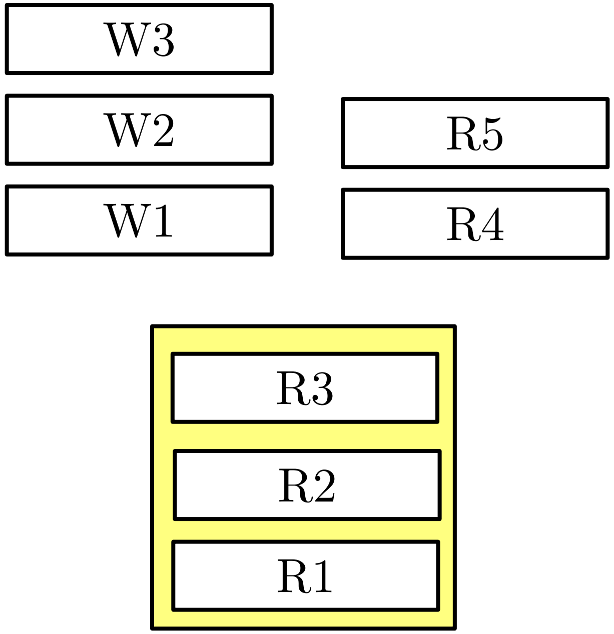
Semaphore S_{rw}=1

- The last reader in the room will be responsible for turning the light OFF.

- **lights OFF** → S_{rw}=0 → **Free Room**

Solution 4: Reads-Writers: Lightswitch Design Pattern

Threads waiting for room access



W3

W2

W1

R5

R4

R3

R2

R1

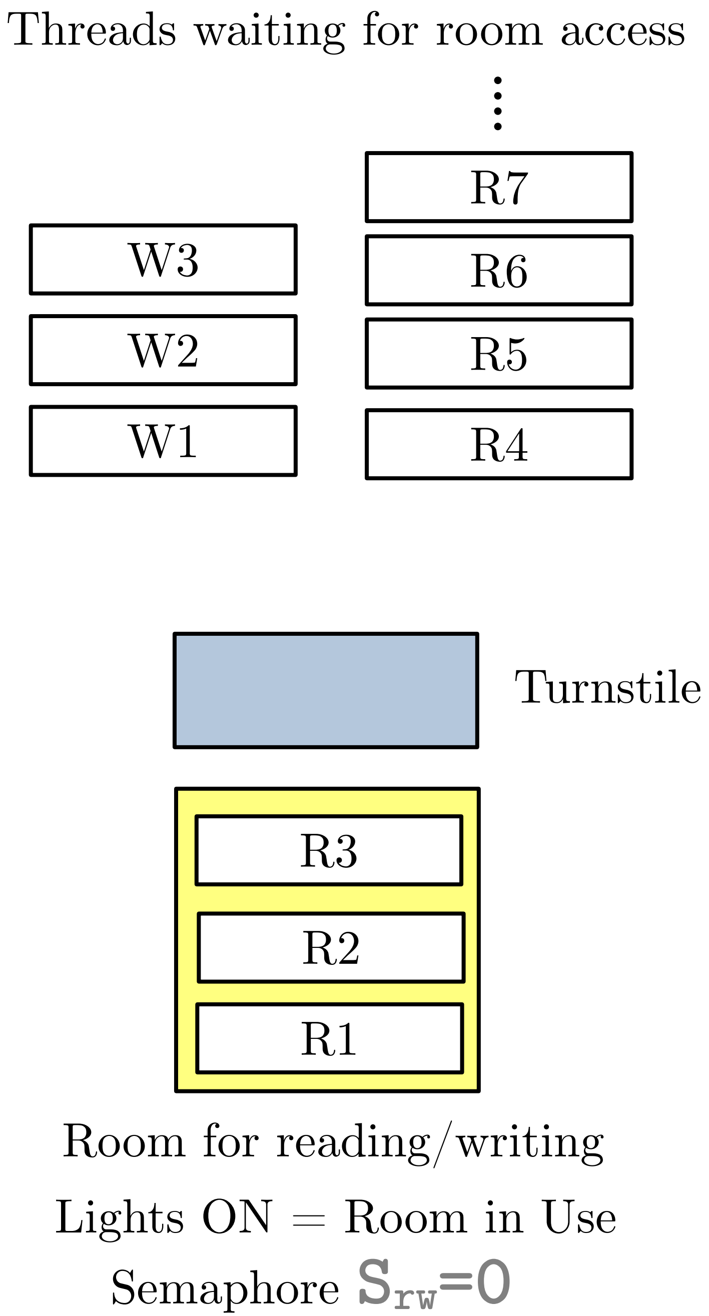
Room for reading/writing

Lights ON = Room in Use

Semaphore S_{rw}=0

- **ISSUE:** If there is a continuous stream of readers while the room is occupied by at least one reader, the writers might ‘*starve*’.

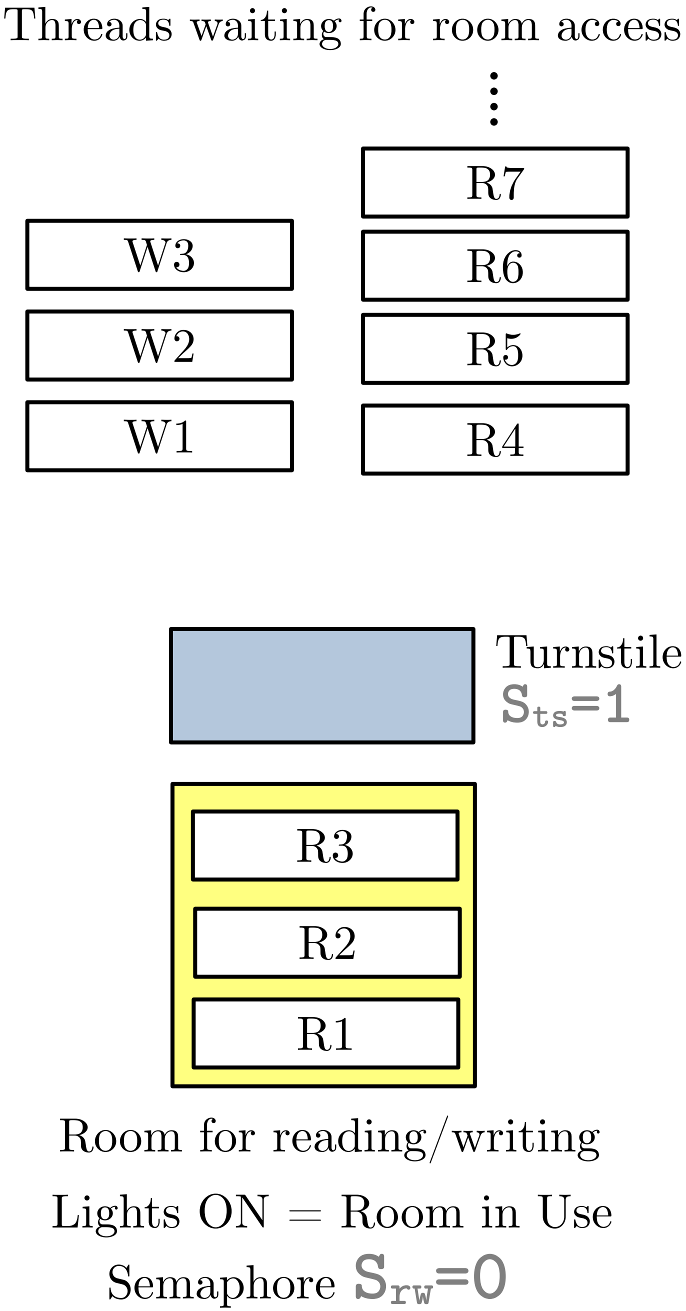
Solution 4: Reads-Writers: Turnstile Design Pattern



- **ISSUE:** If there is a continuous stream of readers while the room is occupied by at least one reader, the writers might ‘*starve*’.

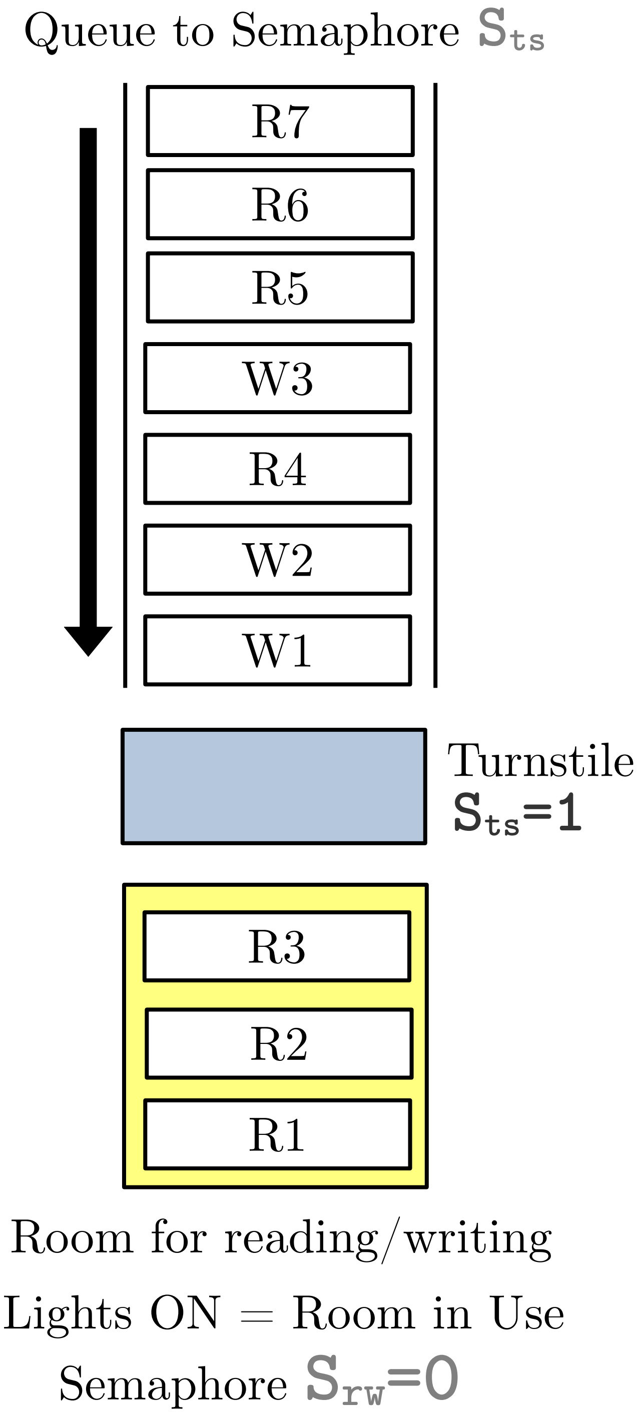
Turnstiles

Solution 4: Reads-Writers: Turnstile Design Pattern



- **ISSUE:** If there is a continuous stream of readers while the room is occupied by at least one reader, the writers might '*starve*'.
- All threads should try to access the turnstile first before trying to enter the room.
- Only one thread at a time can access the turnstile.
- S_{ts} is the semaphore that hold that one *permit* to access the turnstile. Initially, S_{ts}=1.

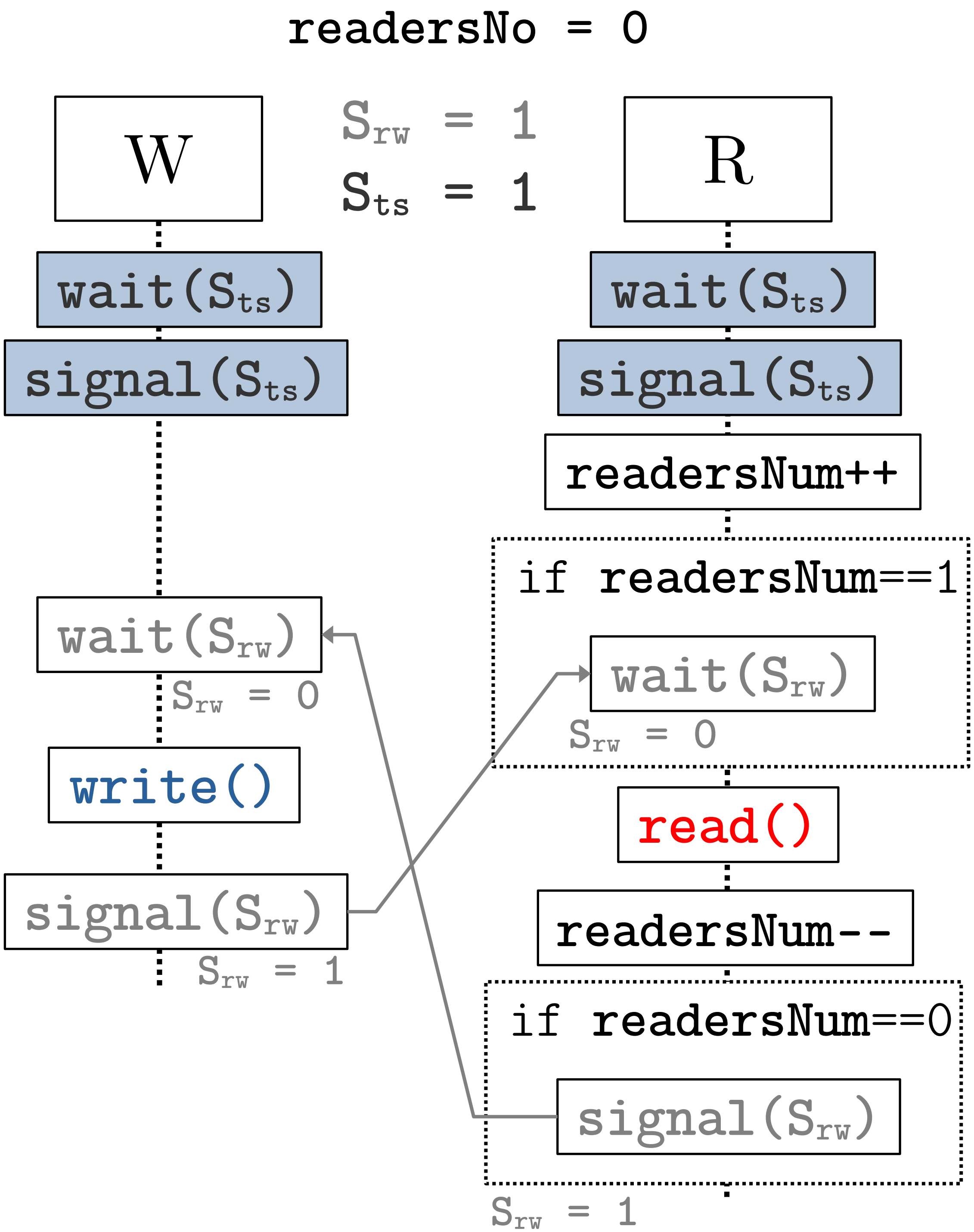
Solution 4: Reads-Writers: Turnstile Design Pattern



- The turnstile mechanism forces all threads to form a line because all of the thread need to acquire the permit to access the turnstile which means all threads will call **`wait(`**S_{ts}**`).`**

- Without the turnstile, if the room is in use by a reader (lights ON), other readers do form a queue in order to access the room and call **`read()`**. A reader can immediate use to room and call **`read()`** concurrently with other readers.

Solution 4: Reads-Writers: Turnstile Design Pattern



readersNo = 0

S_{rw} = 1

S_{ts} = 1

W

wait(S_{ts})

signal(S_{ts})

wait(S_{rw})

S_{rw} = 0

write()

signal(S_{rw})

S_{rw} = 1

R

wait(S_{ts})

signal(S_{ts})

readersNum++

if readersNum==1

wait(S_{rw})

S_{rw} = 0

read()

readersNum--

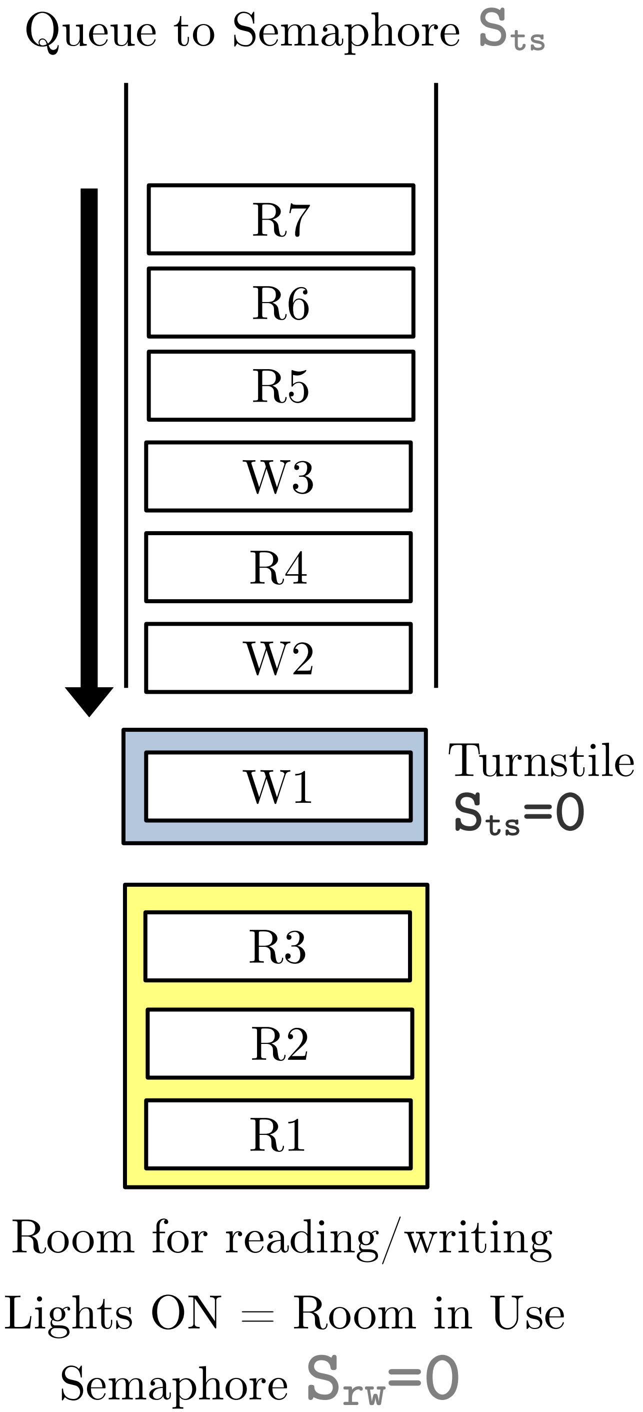
if readersNum==0

signal(S_{rw})

S_{rw} = 1

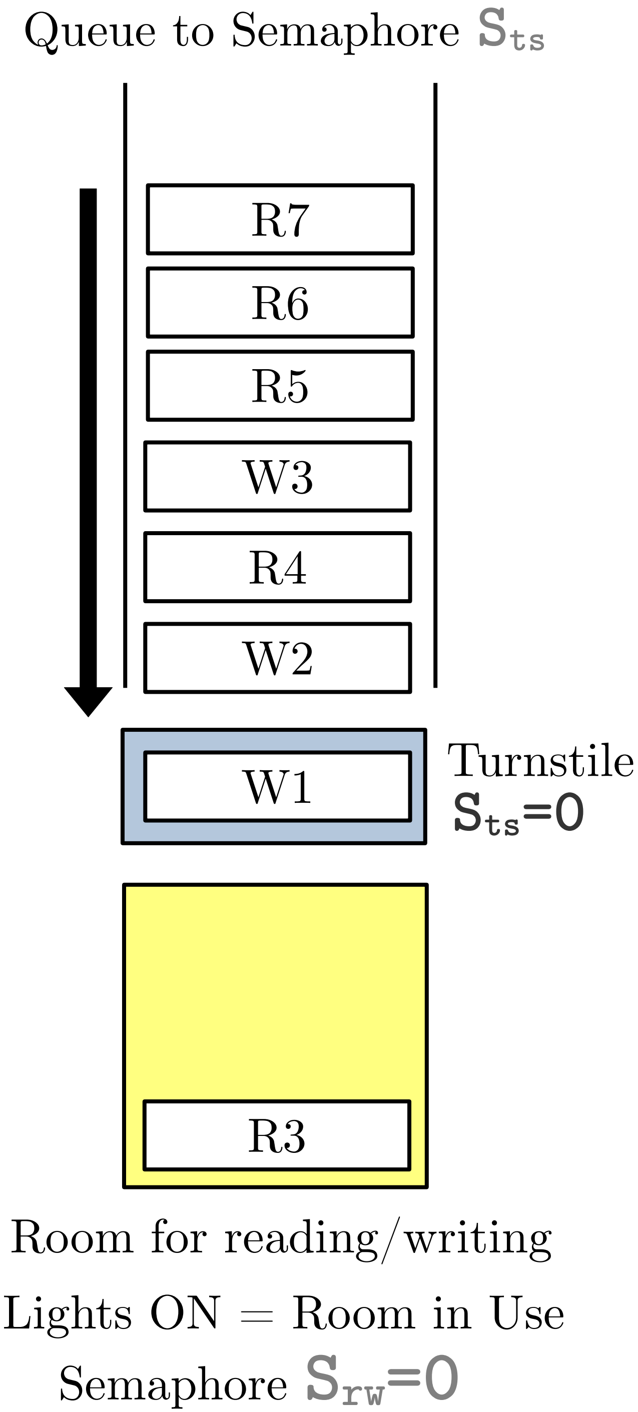
All threads now needs to go through the turnstile by calling **wait(S_{ts})** in order to ask for turnstile access permit then **signal(S_{ts})** is called afterwards.

Solution 4: Reads-Writers: Turnstile Design Pattern



- **ISSUE:** If there is a continuous stream of readers while the room is occupied by at least one reader, the writers might '*starve*'.
- All threads should try to access the turnstile first before trying to enter the room.
- Only one thread at a time can access the turnstile.
- S_{ts} is the semaphore that hold that one *permit* to access the turnstile. Initially, S_{ts}=1.

Solution 4: Reads-Writers: Turnstile Design Pattern



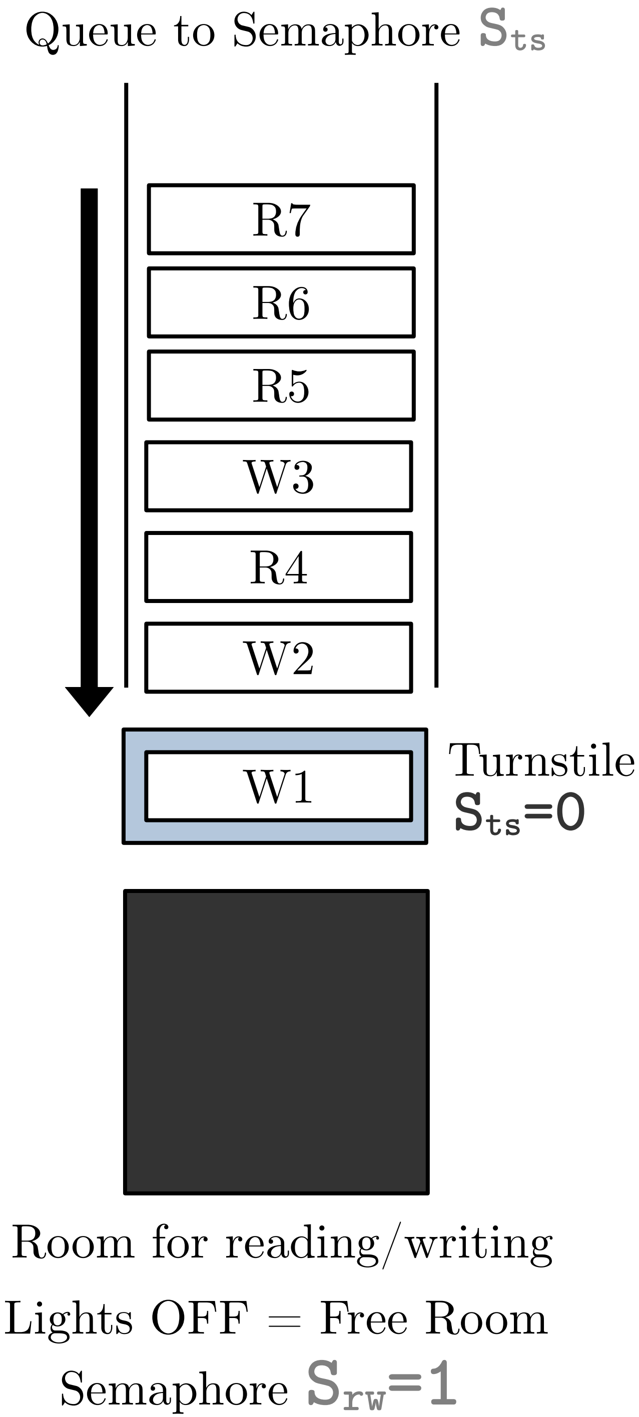
- **ISSUE:** If there is a continuous stream of readers while the room is occupied by at least one reader, the writers might '*starve*'.

- All threads should try to access the turnstile first before trying to enter the room.

- Only one thread at a time can access the turnstile.

- S_{ts} is the semaphore that hold that one *permit* to access the turnstile. Initially, $S_{ts}=1$.

Solution 4: Reads-Writers: Turnstile Design Pattern



- **ISSUE:** If there is a continuous stream of readers while the room is occupied by at least one reader, the writers might '*starve*'.

- All threads should try to access the turnstile first before trying to enter the room.

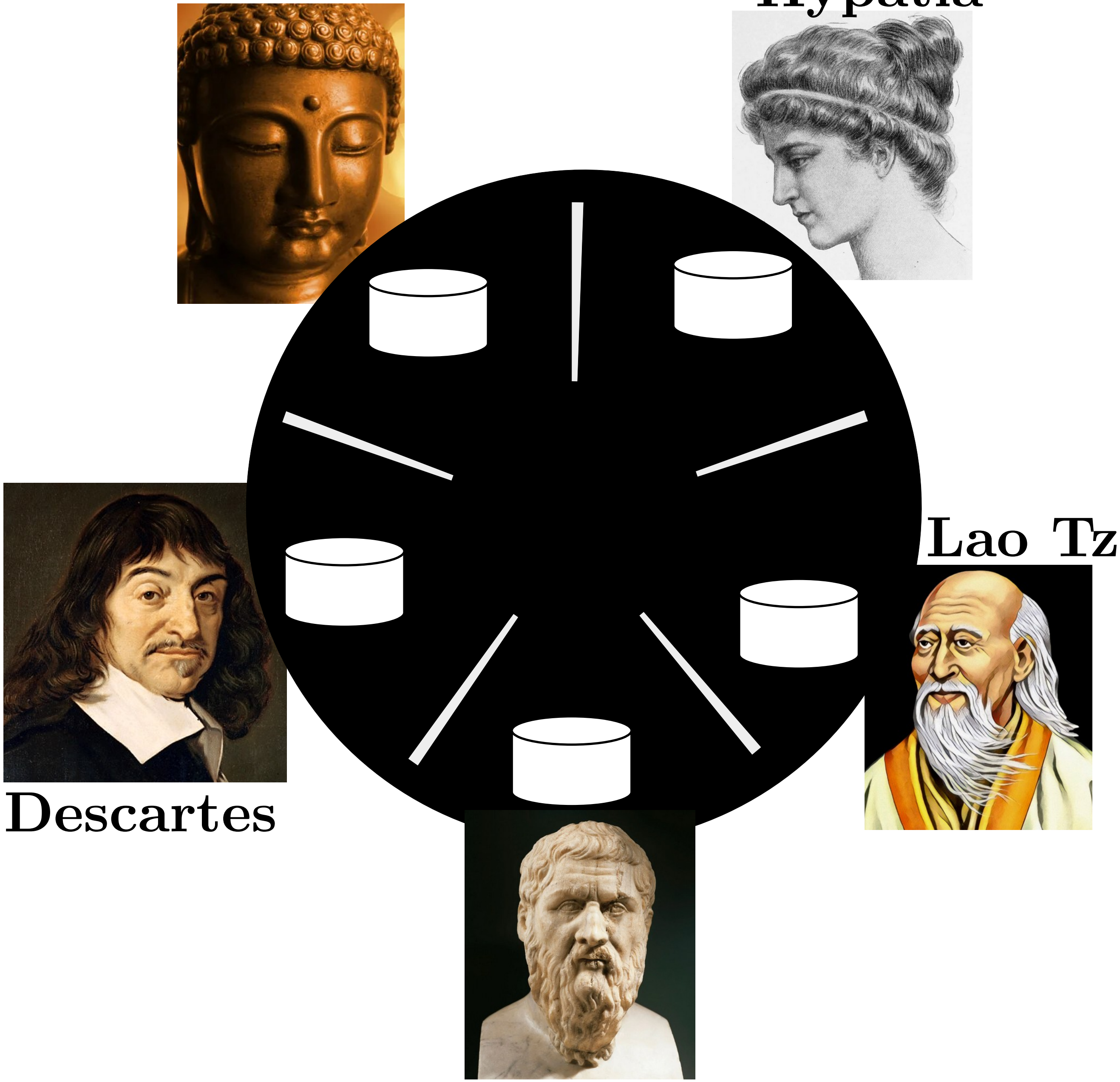
- Only one thread at a time can access the turnstile.

- S_{ts} is the semaphore that hold that one *permit* to access the turnstile. Initially, S_{ts}=1.

Synchronization Task/Problem 5:
Dining Philosophers

[Problem & Solution Descriptions]

Task/Problem 5: Dining Philosophers



Siddhartha

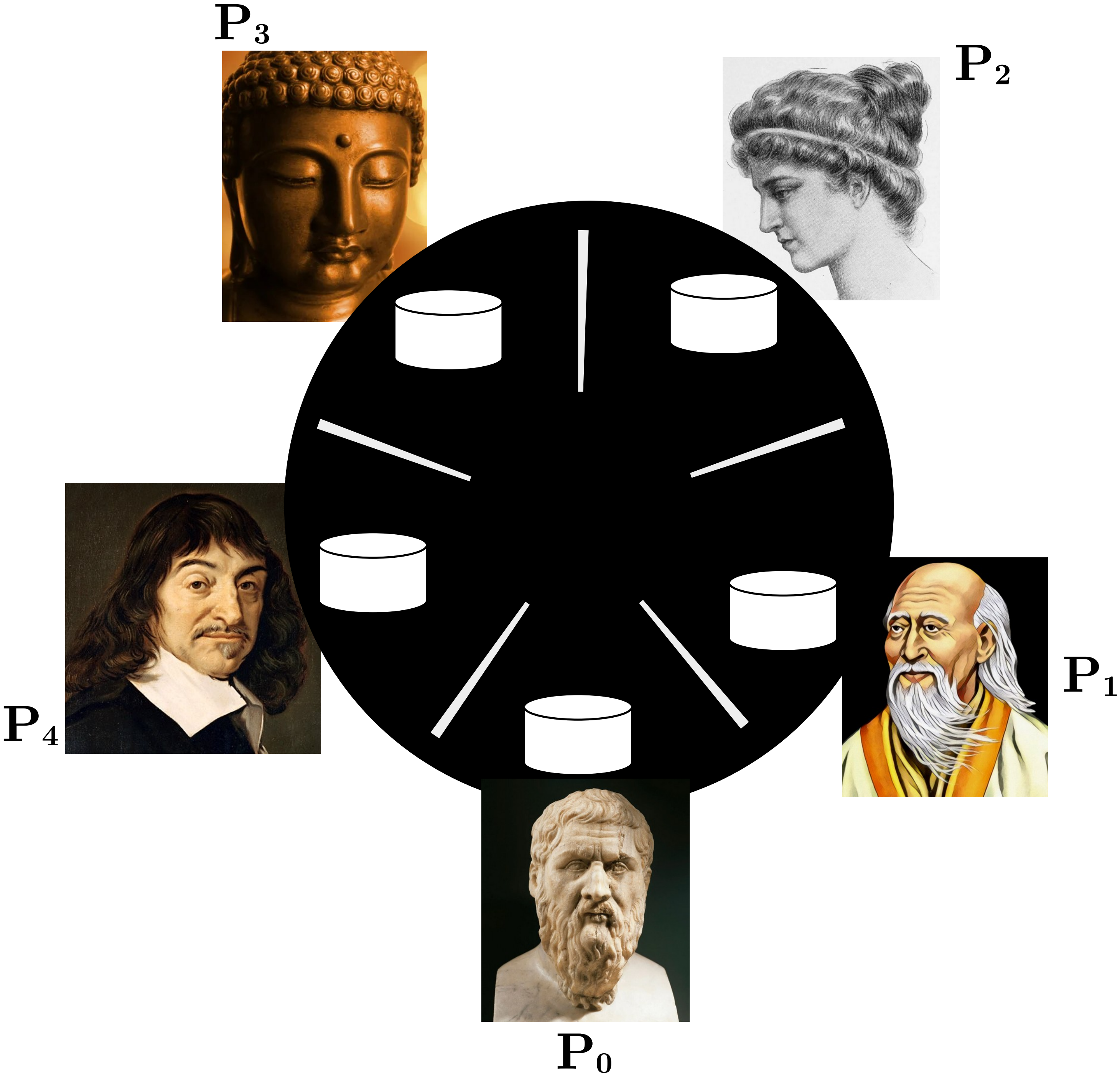
Hypatia

Lao Tzu

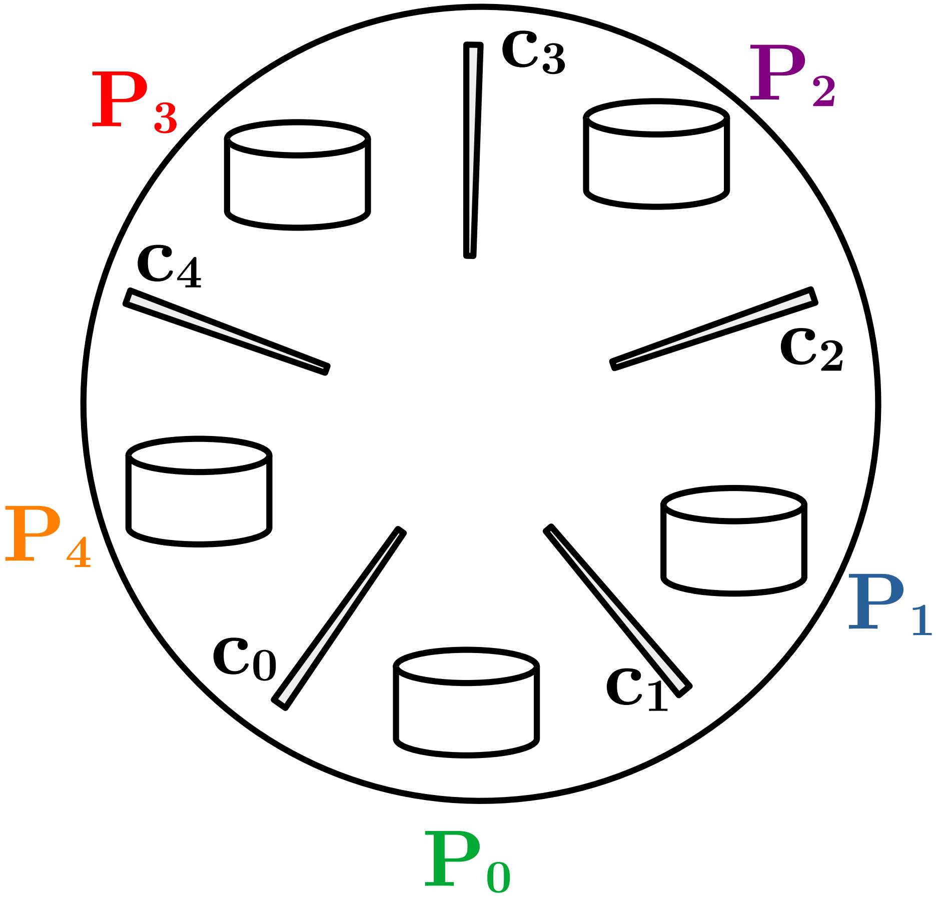
Descartes

Plato

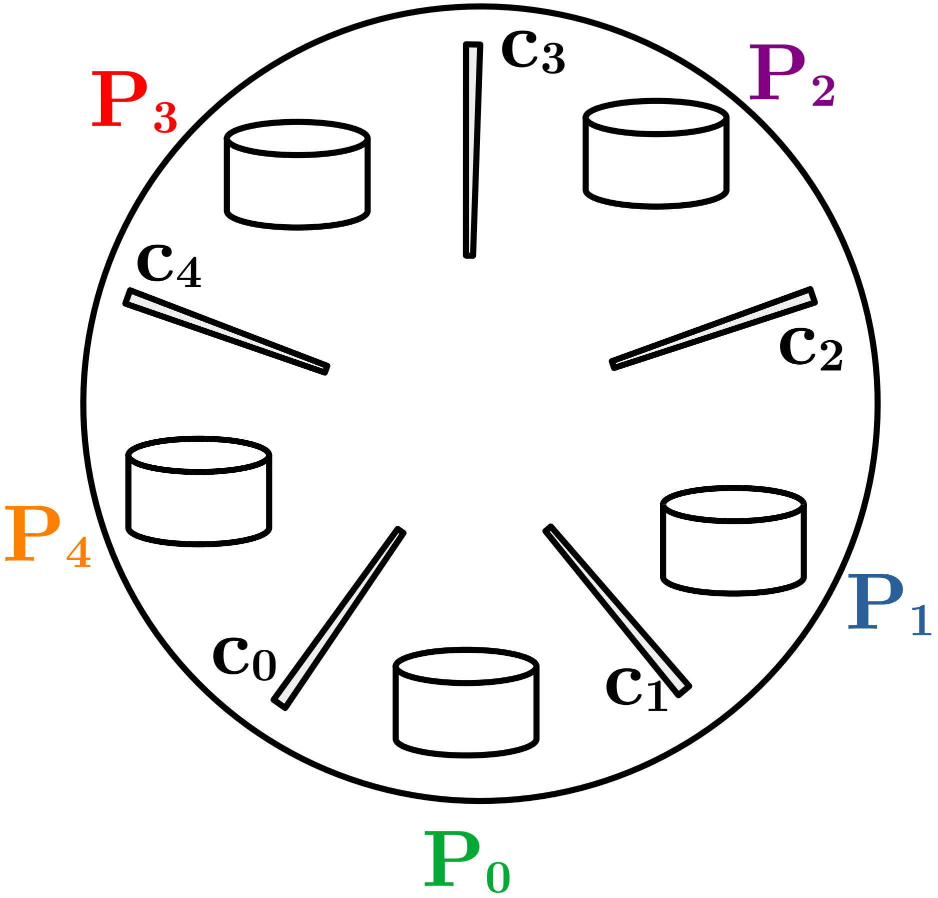
Task/Problem 5: Dining Philosophers



Task/Problem 5: Dining Philosophers



Task/Problem 5: Dining Philosophers



P_i needs c_i and c_{i+1} or more accurately $c_{(i+1)\%5}$

Constraint: Each philosopher needs two chopsticks (left & right) to eat.

P_0 needs c_0 and c_1,

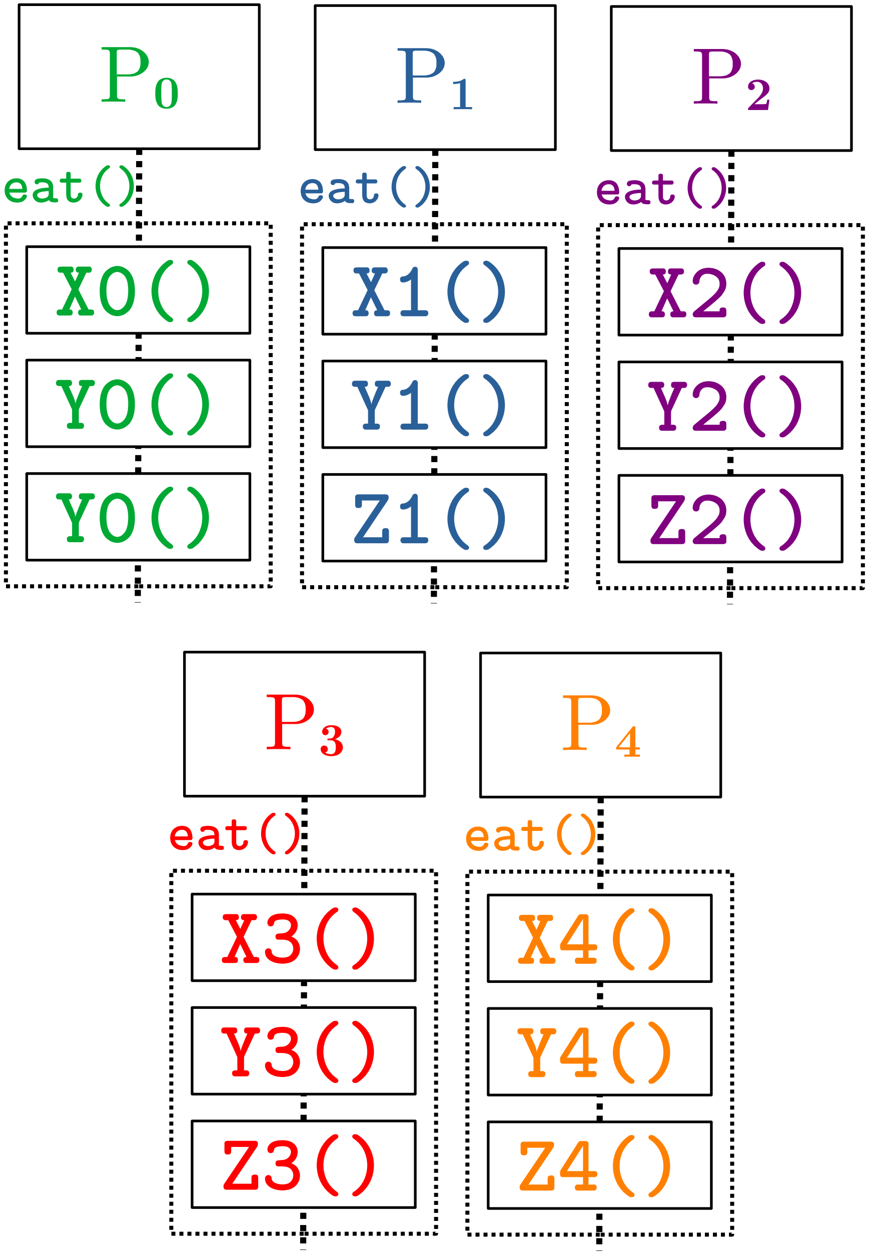
P_1 needs c_1 and c_2,

P_2 needs c_2 and c_3,

P_3 needs c_3 and c_4,

P_4 needs c_4 and c_0,

Task/Problem 5: Dining Philosophers



P_0
eat()
X0()
Y0()
Y0()

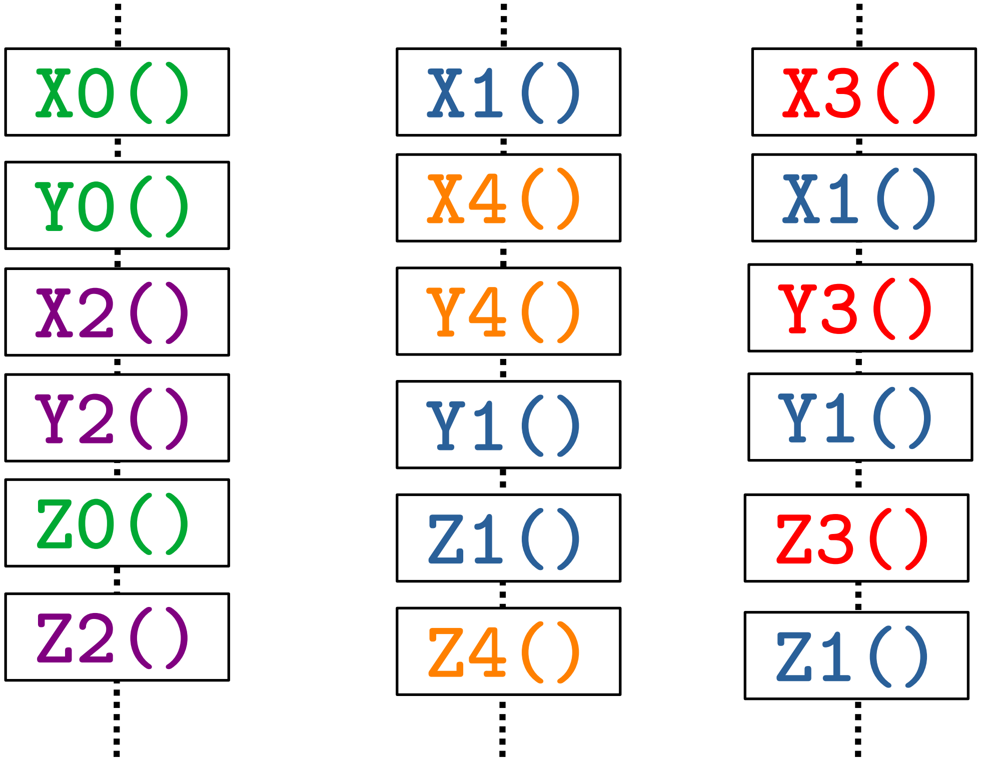
P_1
eat()
X1()
Y1()
Z1()

P_2
eat()
X2()
Y2()
Z2()

P_3
eat()
X3()
Y3()
Z3()

P_4
eat()
X4()
Y4()
Z4()

some VALID orders



X0()
Y0()
X2()
Y2()
Z0()
Z2()

X1()
X4()
Y4()
Y1()
Z1()
Z4()

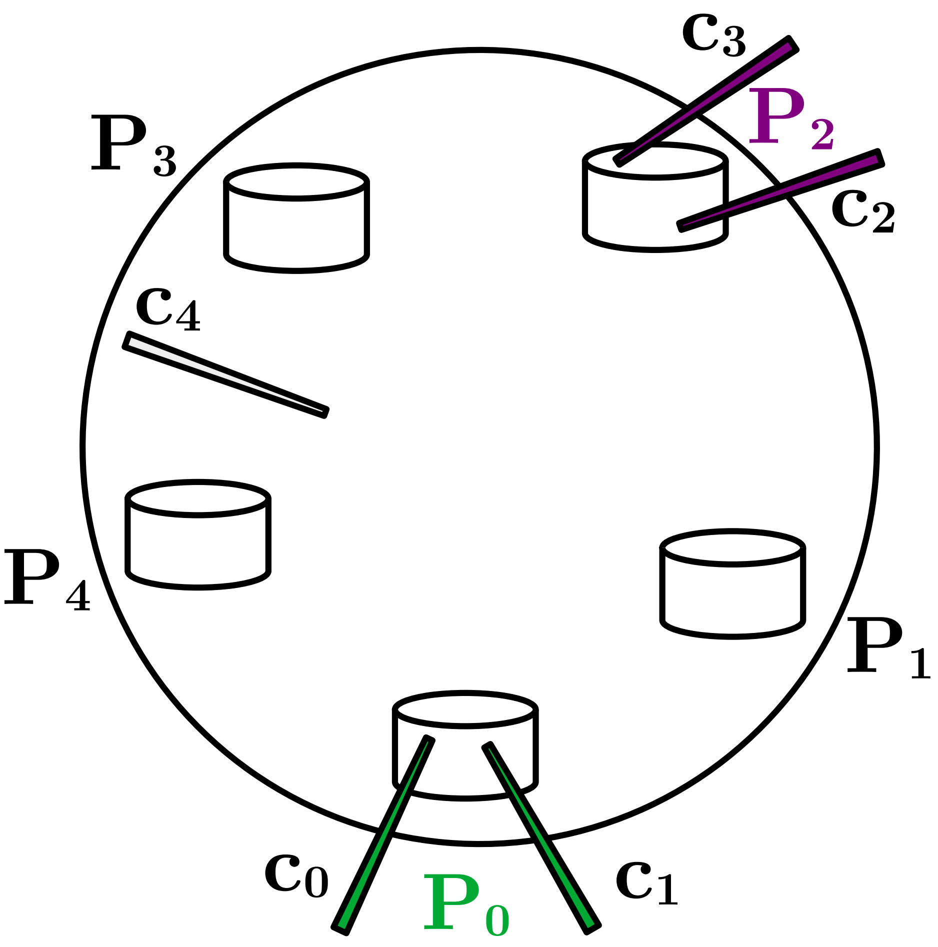
X3()
X1()
Y3()
Y1()
Z3()
Z1()

P_0 & P_2 eating in parallel

P_1 & P_4 eating in parallel

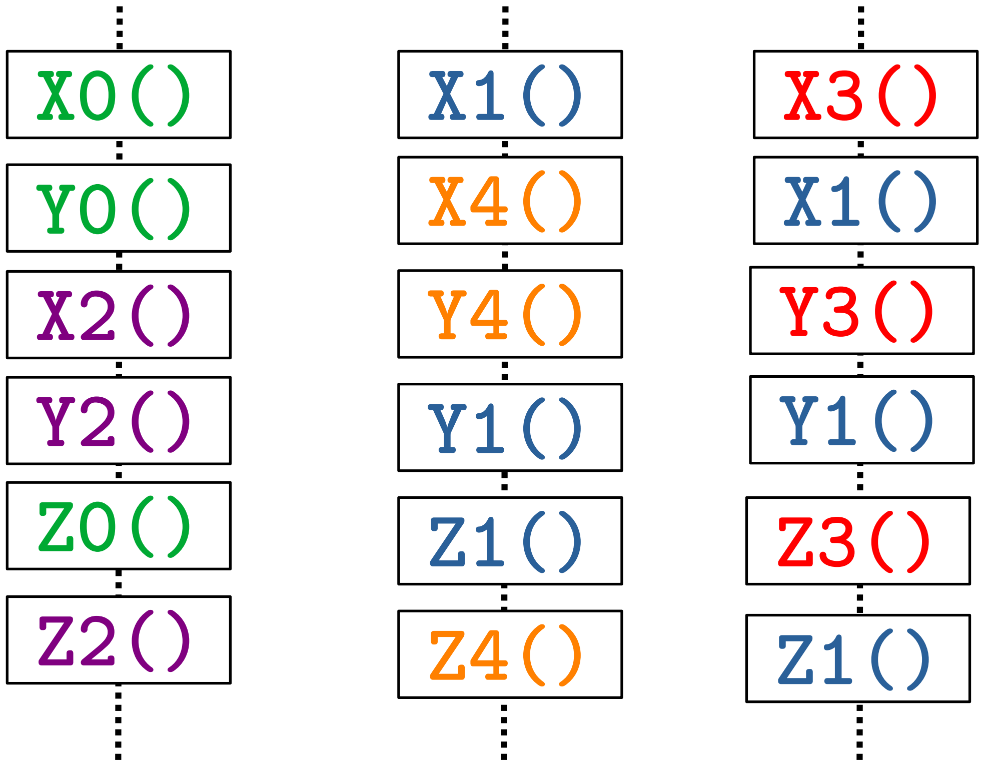
P_1 & P_3 eating in parallel

Task/Problem 5: Dining Philosophers



P_0 & P_2 eating in parallel

some VALID orders



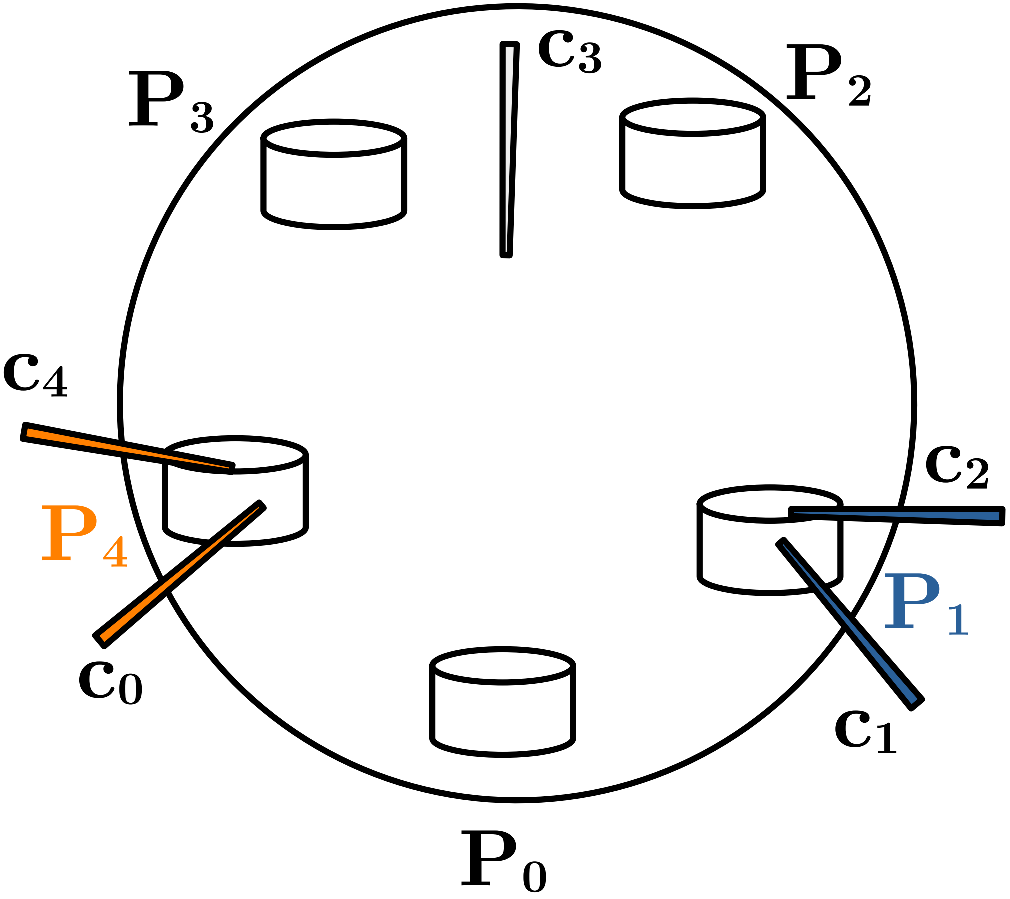
```
X0()    X1()    X3()
Y0()    X4()    X1()
X2()    Y4()    Y3()
Y2()    Y1()    Y1()
Z0()    Z1()    Z3()
Z2()    Z4()    Z1()
```

P_0 & P_2 eating in parallel

P_1 & P_4 eating in parallel

P_1 & P_3 eating in parallel

Task/Problem 5: Dining Philosophers

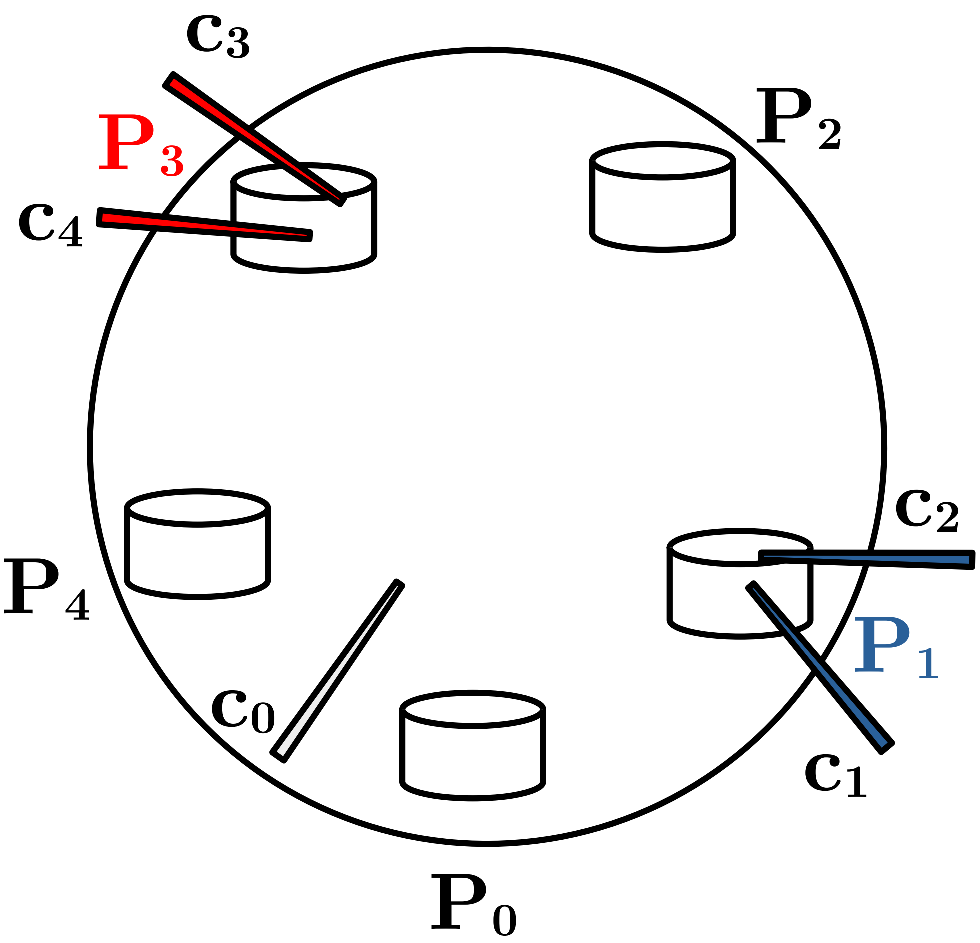


P_1 & P_4 eating in parallel

some VALID orders

X0()	X1()	X3()
Y0()	X4()	X1()
X2()	Y4()	Y3()
Y2()	Y1()	Y1()
Z0()	Z1()	Z3()
Z2()	Z4()	Z1()
P_0 & P_2 eating in parallel	P_1 & P_4 eating in parallel	P_1 & P_3 eating in parallel

Task/Problem 5: Dining Philosophers

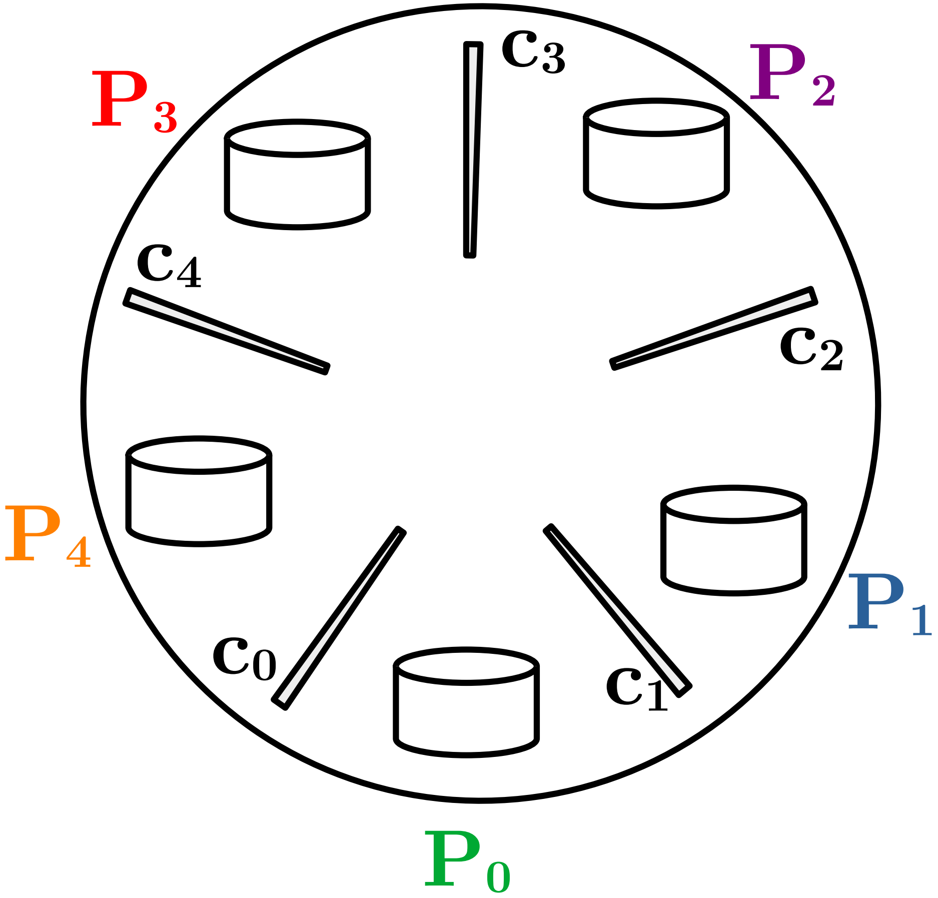


P_1 & P_3 eating in parallel

some VALID orders

X0()	X1()	X3()
Y0()	X4()	X1()
X2()	Y4()	Y3()
Y2()	Y1()	Y1()
Z0()	Z1()	Z3()
Z2()	Z4()	Z1()
P_0 & P_2 eating in parallel	P_1 & P_4 eating in parallel	P_1 & P_3 eating in parallel

Task/Problem 5: Dining Philosophers



some INVALID orders

```
X0()
Y0()
X1()
Y1()
Z0()
Z1()
```

P_0 & P_1 eating in parallel

```
X2()
Y2()
X3()
Y3()
Z3()
Z2()
```

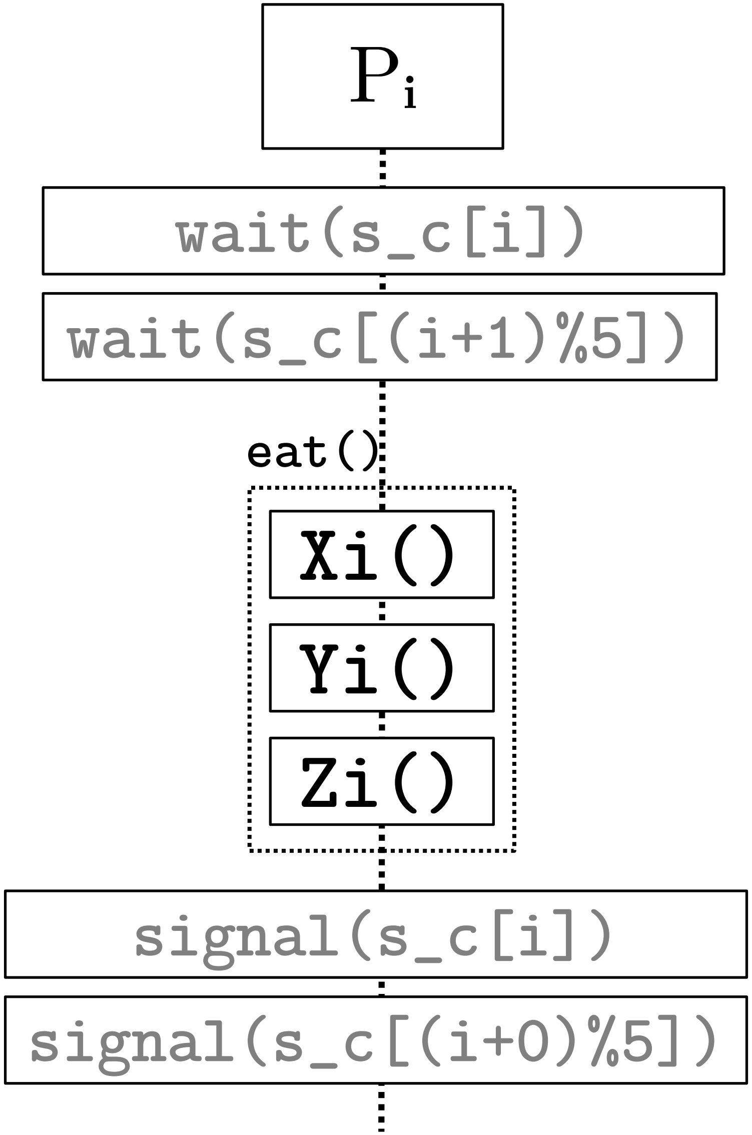
P_2 & P_3 eating in parallel

```
X4()
Y4()
X1()
X2()
Y2()
Y1()
Z1()
Z4()
Z2()
```

P_1, P_2, P_4 eating in parallel

Task/Problem 5: Dining Philosophers

```
s_c[5]={1,1,1,1,1}
```



```
Pi
wait(s_c[i])
wait(s_c[(i+1)%5])
eat()
Xi()
Yi()
Zi()
signal(s_c[i])
signal(s_c[(i+0)%5])
```

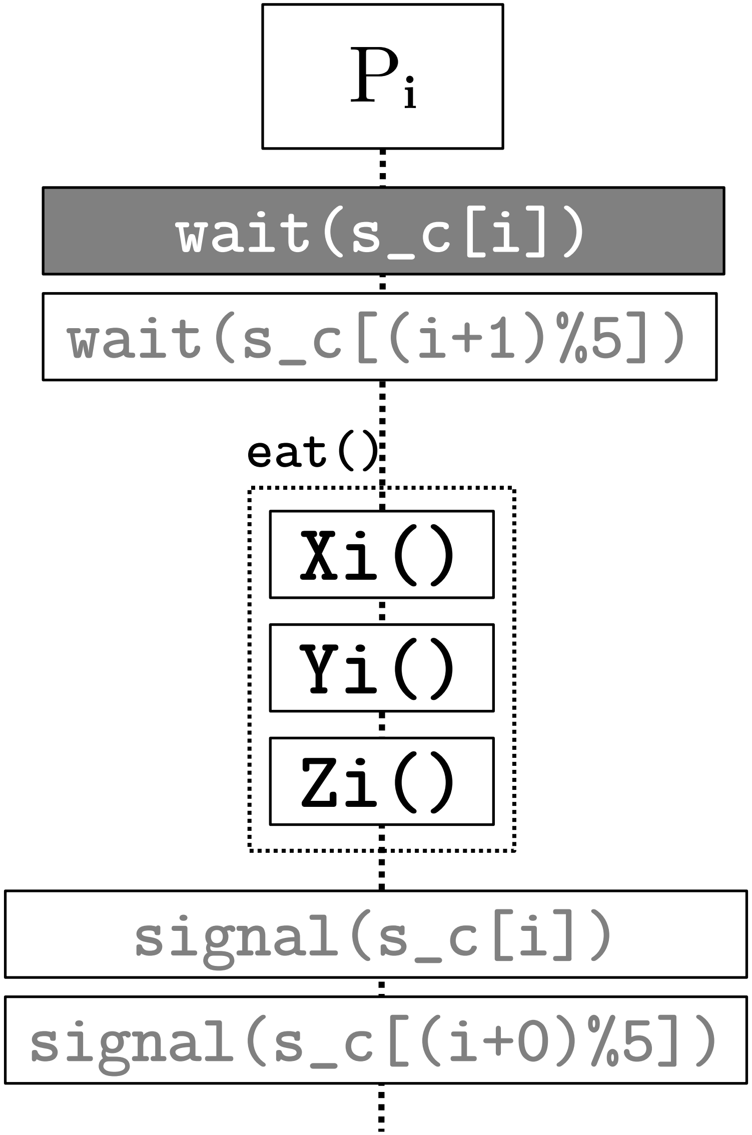
`s_c[5]` is array of semaphores of size 5, one semaphore for each chopstick. Each semaphore `s_c[i]` is initialized to `1`.

Thread **P$_i$** calls `wait()` for the semaphores `s_c[i]and` `s_c[i+1]` (more accurately `s_c[(i+1)%5]`) associated with **P$_i$**'s left and right chopsticks.

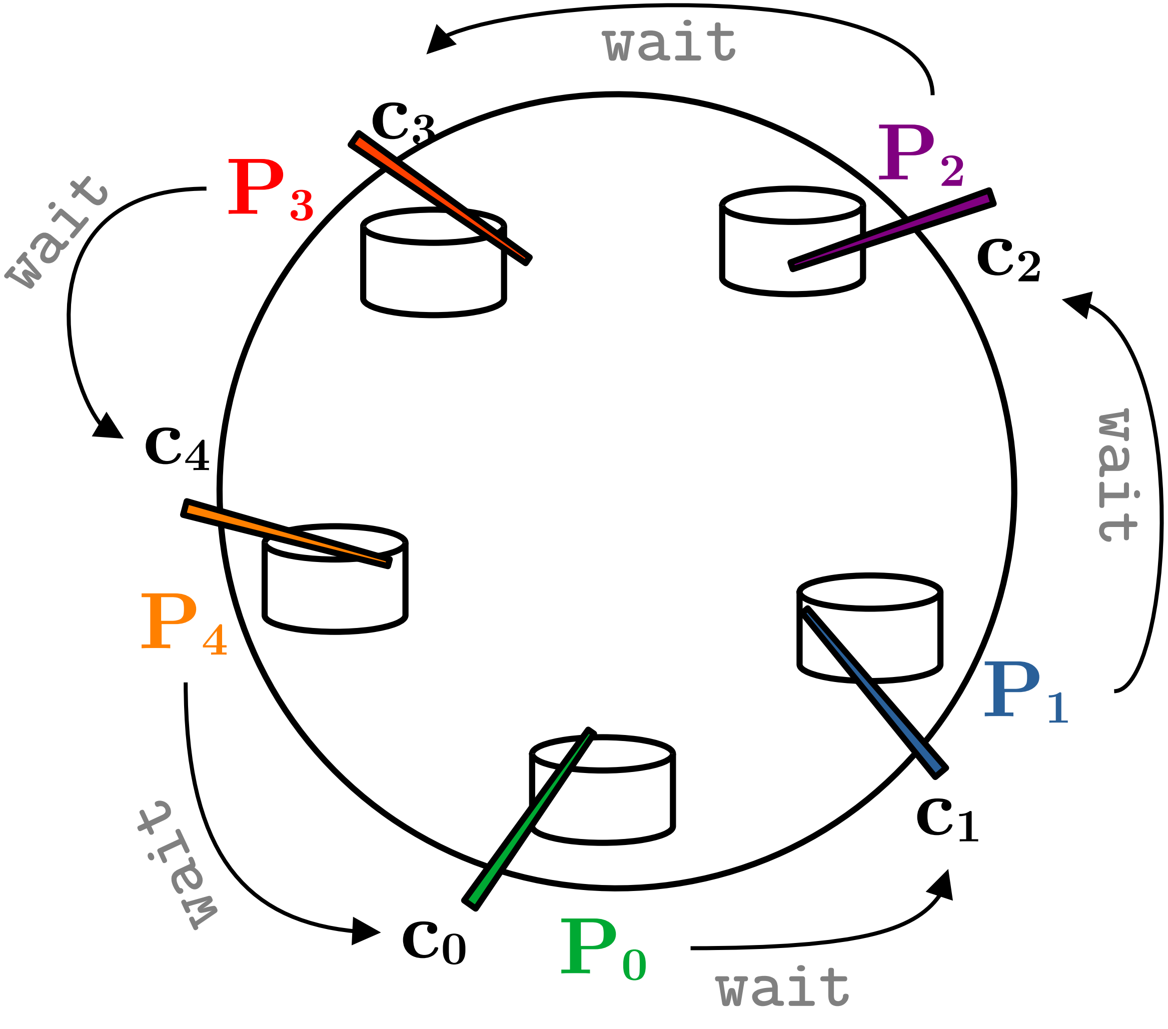
After acquire the two chopstick permits, **P$_i$** will eat and then release both permits by calling `signal()` for each semaphore.

Task/Problem 5: Dining Philosophers: Deadlock

```
s_c[5]={1,1,1,1,1}
```



```
Pi
wait(s_c[i])
wait(s_c[(i+1)%5])
eat()
Xi()
Yi()
Zi()
signal(s_c[i])
signal(s_c[(i+0)%5])
```



Each $\mathbf{P}_i$ was able to get the permission to use $\mathbf{c}_i$ (via semaphore s_c[i]) but each **Pi** is also waiting for permission to use $\mathbf{c}_{i+1}$ which is held by $\mathbf{P}_{i+1}$.

Mutual/Circular waiting = Deadlock.