OPERATING SYSTEMS

WORKSHEET

Assume `a1` writes to a file, and `b1` prints a line from the file (hence reads from the file)

Goal : We want `a1` to complete before `b1` begins

Use semaphore "sem" to achieve this

Already run code

```
sem = Semaphore(0)
```

Thread A

```
a1
```

Thread B

```
b1
```

WORKSHEET

Assume `a1` writes to a file, and `b1` prints a line from the file (hence reads from the file)

Goal : We want `a1` to complete before `b1` begins

Use semaphore "sem" to achieve this

Already run code

```
sem = Semaphore(0)
```

Thread A

```
a1
sem.increment()
```

Thread B

```
sem.decrement()
b1
```

What happens if we execute while S=0?

WORKSHEET

Assume `a1` writes to a file, and `b1` prints a line from the file (hence reads from the file)

Goal : We want `a1` to complete before `b1` begins

Use semaphore "sem" to achieve this

Already run code

```
sem = Semaphore(0)
```

Thread A

```
a1
sem.increment()
```

Thread B

```
sem.decrement()
b1
```

Blocks until S=1

WORKSHEET

Assume `a1` writes to a file, and `b1` prints a line from the file (hence reads from the file)

Goal : We want `a1` to complete before `b1` begins

Use semaphore "sem" to achieve this

Already run code

```
sem = Semaphore(0)
```

Thread A

```
a1
sem.increment()
```

Thread B

```
sem.decrement()
b1
```

Blocks until S=1

What happens after executing this?

WORKSHEET

Assume `a1` writes to a file, and `b1` prints a line from the file (hence reads from the file)

Goal : We want `a1` to complete before `b1` begins

Use semaphore "sem" to achieve this

Already run code

```
sem = Semaphore(0)
```

Thread A

```
a1
sem.increment()
```

Thread B

```
sem.decrement()
b1
```

S will increment to 1

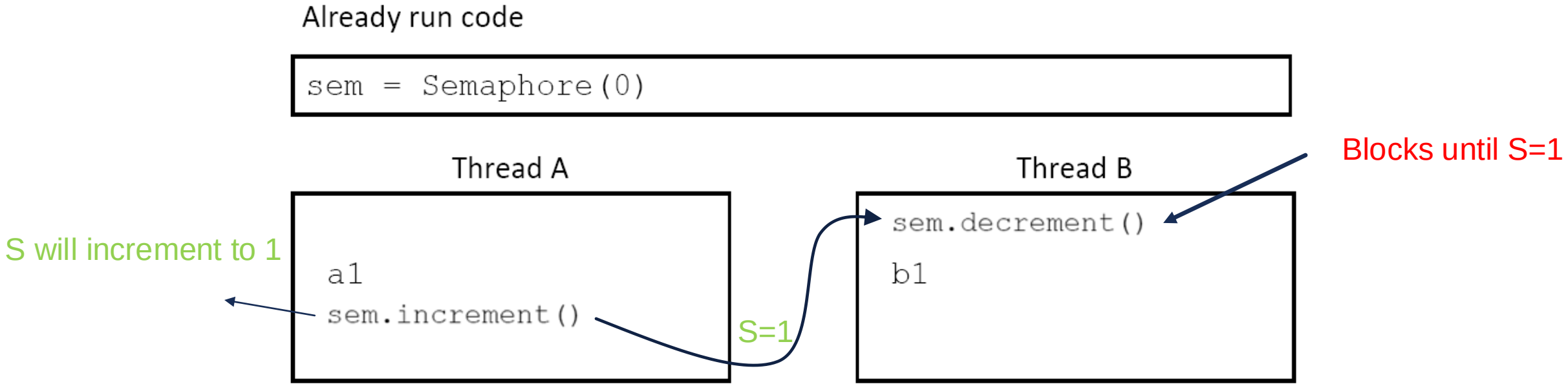
Blocks until S=1

WORKSHEET

Assume `a1` writes to a file, and `b1` prints a line from the file (hence reads from the file)

Goal : We want `a1` to complete before `b1` begins

Use semaphore "sem" to achieve this

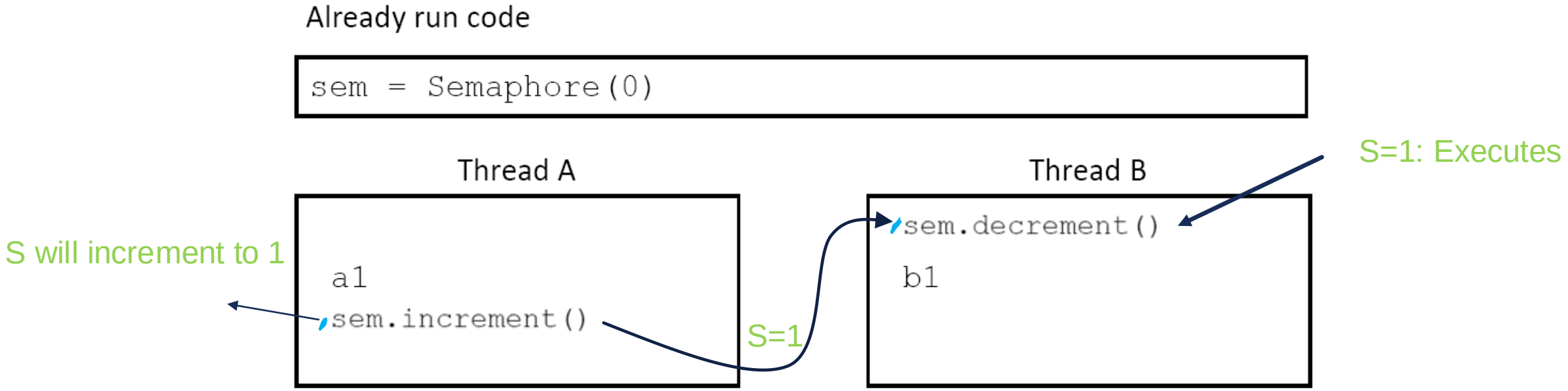


WORKSHEET

Assume `a1` writes to a file, and `b1` prints a line from the file (hence reads from the file)

Goal : We want `a1` to complete before `b1` begins

Use semaphore "sem" to achieve this



WORKSHEET Q2

Already run code

Thread A

```
a1

a2
```

Thread B

```
b1

b2
```

Goals

- a1 must happen before b2
- b1 must happen before a2

You can use more than one semaphore (hint: you should)

WORKSHEET Q2

Already run code

```
sem1=Semaphore(0)
sem2=Semaphore(0)
```

Thread A

```
a1
sem1.increment()
sem2.decrement()
a2
```

Thread B

```
b1
sem2.increment()
sem1.decrement()
b2
```

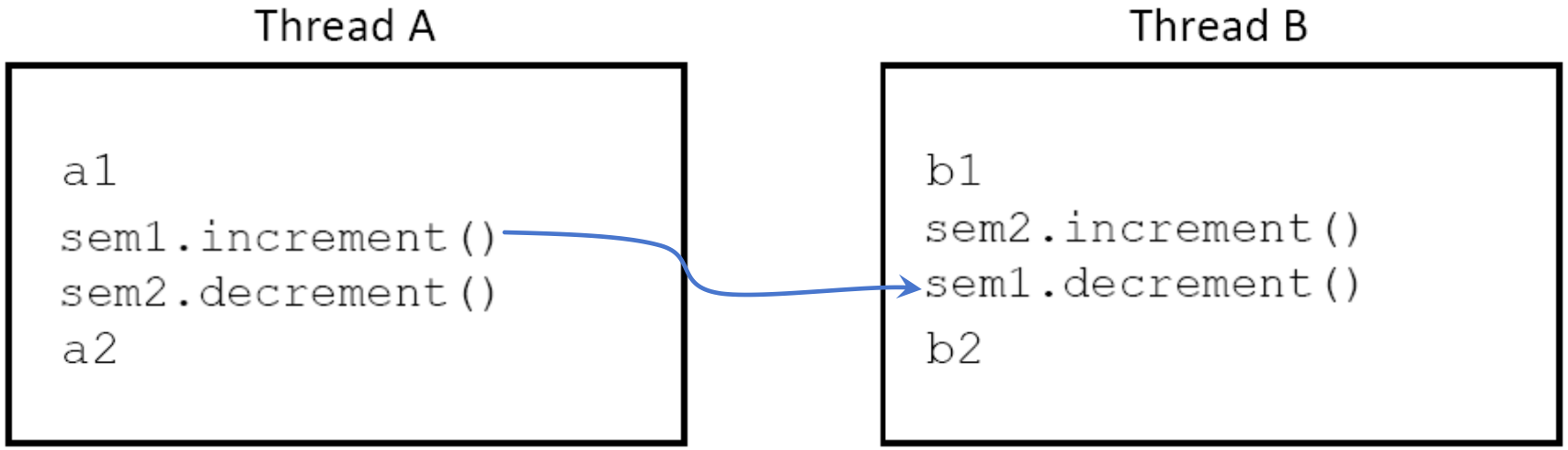
Goals

- a1 must happen before b2
- b1 must happen before a2

WORKSHEET Q2

Already run code

```
sem1=Semaphore(0)
sem2=Semaphore(0)
```



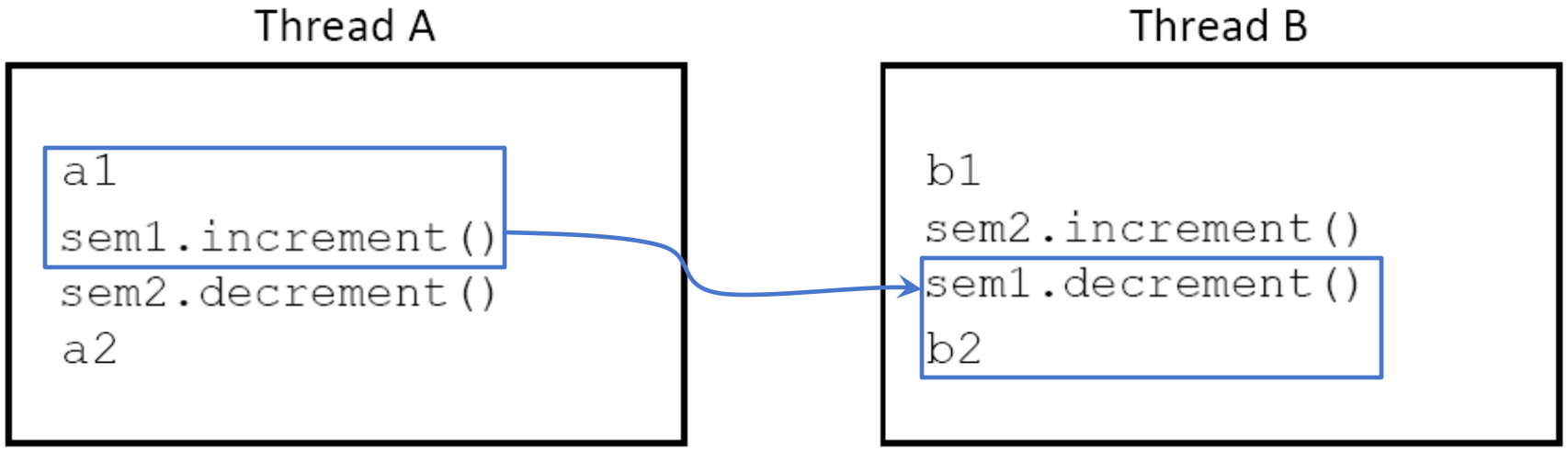
Goals

- a1 must happen before b2
- b1 must happen before a2

WORKSHEET Q2

Already run code

```
sem1=Semaphore(0)
sem2=Semaphore(0)
```



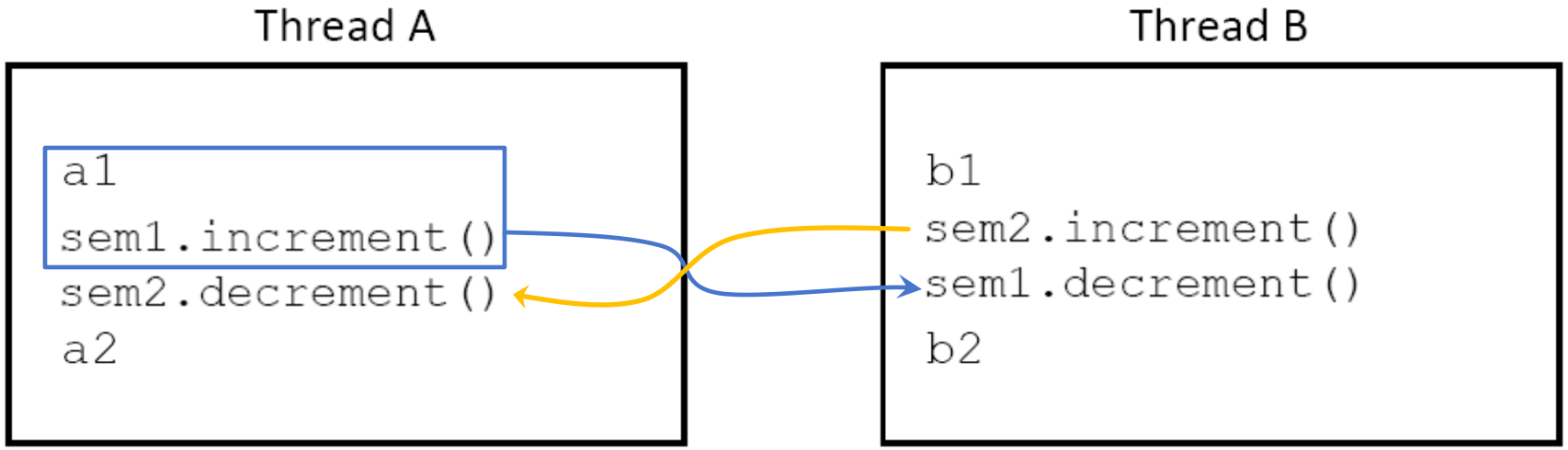
Goals

- a1 must happen before b2
- b1 must happen before a2

WORKSHEET Q2

Already run code

```
sem1=Semaphore(0)
sem2=Semaphore(0)
```



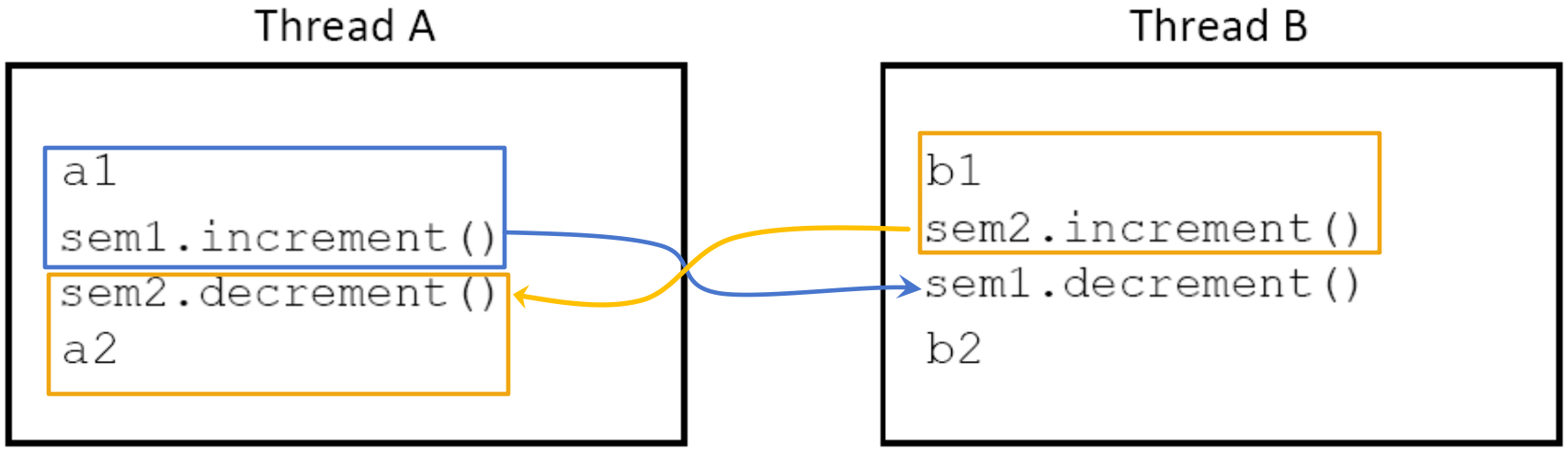
Goals

- a1 must happen before b2
- b1 must happen before a2

WORKSHEET Q2

Already run code

```
sem1=Semaphore(0)
sem2=Semaphore(0)
```



Goals

- a1 must happen before b2
- b1 must happen before a2

WORKSHEET Q2

Already run code

```
sem1=Semaphore(0)
sem2=Semaphore(0)
```

Thread A

```
a1
sem1.decrement()
sem2.increment()
a2
```

Thread B

```
b1
sem1.decrement()
sem2.increment()
b2
```

Goals

- a1 must happen before b2
- b1 must happen before a2

What happens with this solution?

WORKSHEET Q2

Already run code

```
sem1=Semaphore(0)
sem2=Semaphore(0)
```

Thread A

```
a1
sem1.decrement()
sem2.increment()
a2
```

Thread B

```
b1
sem1.decrement()
sem2.increment()
b2
```

Goals

- a1 must happen before b2
- b1 must happen before a2

Both threads will be blocked forever! (Deadlock)

SEMAPHORE IMPLEMENTATION

To implement semaphores in BLITZ we need to:

SEMAPHORE IMPLEMENTATION

To implement semaphores in BLITZ we need to:

1. Enforce atomic operation. This is done by disabling interrupts in Blitz or similar systems.
2. Increment/Decrement the value of the counter/semaphore.
3. Either put a thread to sleep or let it wake up another thread depending whether we're using increment/decrement.
4. Reset interrupt enable/disable.

SEMAPHORE IMPLEMENTATION

Worksheet Q1

```
1. readyList.AddToEnd (t)
2. oldIntStat = SetInterruptsTo (DISABLED)
3. waitingThreads.AddToEnd (currentThread)
4. t = waitingThreads.Remove ()
5. currentThread.Sleep ()
```

```
----------  Semaphore.Up  ----------
method Up ()
    var
      oldIntStat: int
      t: ptr to Thread

    ____________________________

    if count == 0x7fffffff
      FatalError ("Semaphore count overflowed during 'Up' operation")
    endIf
    count = count + 1
    if count <= 0

     ____________________________

      t.status = READY
      readyList.AddToEnd (t)
    endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
  endMethod
```

SEMAPHORE IMPLEMENTATION

Worksheet Q1

```
1. readyList.AddToEnd (t)
2. oldIntStat = SetInterruptsTo (DISABLED)
3. waitingThreads.AddToEnd (currentThread)
4. t = waitingThreads.Remove ()
5. currentThread.Sleep ()
```

```
----------  Semaphore.Up  ----------
method Up ()
    var
      oldIntStat: int
      t: ptr to Thread
    oldIntStat = SetInterruptsTo (DISABLED)
    ____________________________

    if count == 0x7fffffff
      FatalError ("Semaphore count overflowed during 'Up' operation")
    endIf
    count = count + 1
    if count <= 0


     ____________________________

      t.status = READY
      readyList.AddToEnd (t)
    endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
  endMethod
```

SEMAPHORE IMPLEMENTATION

Worksheet Q1

```
1. readyList.AddToEnd (t)
2. oldIntStat = SetInterruptsTo (DISABLED)
3. waitingThreads.AddToEnd (currentThread)
4. t = waitingThreads.Remove ()
5. currentThread.Sleep ()
```

```
----------  Semaphore.Up  ----------
method Up ()
    var
      oldIntStat: int
      t: ptr to Thread
    oldIntStat = SetInterruptsTo (DISABLED)
    ---------------------------------

    if count == 0x7fffffff
      FatalError ("Semaphore count overflowed during 'Up' operation")
    endIf
    count = count + 1
    if count <= 0
      t = waitingThreads.Remove ()
      ---------------------------------

      t.status = READY
      readyList.AddToEnd (t)
    endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
  endMethod
```

SEMAPHORE IMPLEMENTATION

Worksheet Q2

```
1. readyList.AddToEnd (t)
2. oldIntStat = SetInterruptsTo (DISABLED)
3. waitingThreads.AddToEnd (currentThread)
4. t = waitingThreads.Remove ()
5. currentThread.Sleep ()
```

```
----------  Semaphore . Down  ----------
method Down ()
    var
      oldIntStat: int

     ____________________________

    if count == 0x80000000
      FatalError ("Semaphore count underflowed during 'Down' operation")
    endIf
    count = count - 1
    if count < 0

     ____________________________

     ____________________________

     endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
  endMethod
```

SEMAPHORE IMPLEMENTATION

Worksheet Q2

```
1. readyList.AddToEnd (t)
2. oldIntStat = SetInterruptsTo (DISABLED)
3. waitingThreads.AddToEnd (currentThread)
4. t = waitingThreads.Remove ()
5. currentThread.Sleep ()
```

```
----------  Semaphore . Down  ----------
method Down ()
    var
      oldIntStat: int

    oldIntStat = SetInterruptsTo (DISABLED)
    ____________________________________

    if count == 0x80000000
      FatalError ("Semaphore count underflowed during 'Down' operation")
    endIf
    count = count - 1
    if count < 0

    ____________________________

    ____________________________

    endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
  endMethod
```

SEMAPHORE IMPLEMENTATION

Worksheet Q2

```
1. readyList.AddToEnd (t)
2. oldIntStat = SetInterruptsTo (DISABLED)
3. waitingThreads.AddToEnd (currentThread)
4. t = waitingThreads.Remove ()
5. currentThread.Sleep ()
```

```
----------  Semaphore . Down  ----------
method Down ()
    var
      oldIntStat: int

    oldIntStat = SetInterruptsTo (DISABLED)
    ___________________________________

    if count == 0x80000000
      FatalError ("Semaphore count underflowed during 'Down' operation")
    endIf
    count = count - 1
    if count < 0

    waitingThreads.AddToEnd (currentThread)
    ______________________________________

    ___________________________

    endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
  endMethod
```

SEMAPHORE IMPLEMENTATION

Worksheet Q2

```
1. readyList.AddToEnd (t)
2. oldIntStat = SetInterruptsTo (DISABLED)
3. waitingThreads.AddToEnd (currentThread)
4. t = waitingThreads.Remove ()
5. currentThread.Sleep ()
```

```
----------  Semaphore . Down  ----------
method Down ()
    var
      oldIntStat: int

    oldIntStat = SetInterruptsTo (DISABLED)

    if count == 0x80000000
      FatalError ("Semaphore count underflowed during 'Down' operation")
    endIf
    count = count - 1
    if count < 0

    waitingThreads.AddToEnd (currentThread)

    currentThread.Sleep ()

    endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
  endMethod
```

SEMAPHORE IMPLEMENTATION

```
1. readyList.AddToEnd (t)
2. oldIntStat = SetInterruptsTo (DISABLED)
3. waitingThreads.AddToEnd (currentThread)
4. t = waitingThreads.Remove ()
5. currentThread.Sleep ()
```

```
----------  Semaphore.Up  ----------
method Up ()
    var
      oldIntStat: int
      t: ptr to Thread
    oldIntStat = SetInterruptsTo (DISABLED)
    ------------------------------

    if count == 0x7fffffff
      FatalError ("Semaphore count overflowed during 'Up' operation")
    endIf
    count = count + 1
    if count <= 0
      t = waitingThreads.Remove ()
      ------------------------------

      t.status = READY
      readyList.AddToEnd (t)
    endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
  endMethod
```

SEMAPHORE IMPLEMENTATION

```
1. readyList.AddToEnd (t)
2. oldIntStat = SetInterruptsTo (DISABLED)
3. waitingThreads.AddToEnd (currentThread)
4. t = waitingThreads.Remove ()
5. currentThread.Sleep ()
```

```
----------  Semaphore.Up  ----------
method Up ()
    var
      oldIntStat: int
      t: ptr to Thread
    oldIntStat = SetInterruptsTo (DISABLED)
    ------------------------------

    if count == 0x7fffffff
      FatalError ("Semaphore count overflowed during 'Up' operation")
    endIf
    count = count + 1
    if count <= 0
      t = waitingThreads.Remove ()
      ------------------------------

      t.status = READY
      readyList.AddToEnd (t)
    endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
  endMethod
```

- We said that a semaphore needs to be > 0 to avoid sleeping/waiting after a decrement …
- Why does this work?

SEMAPHORE IMPLEMENTATION

```
1. readyList.AddToEnd (t)
2. oldIntStat = SetInterruptsTo (DISABLED)
3. waitingThreads.AddToEnd (currentThread)
4. t = waitingThreads.Remove ()
5. currentThread.Sleep ()
```

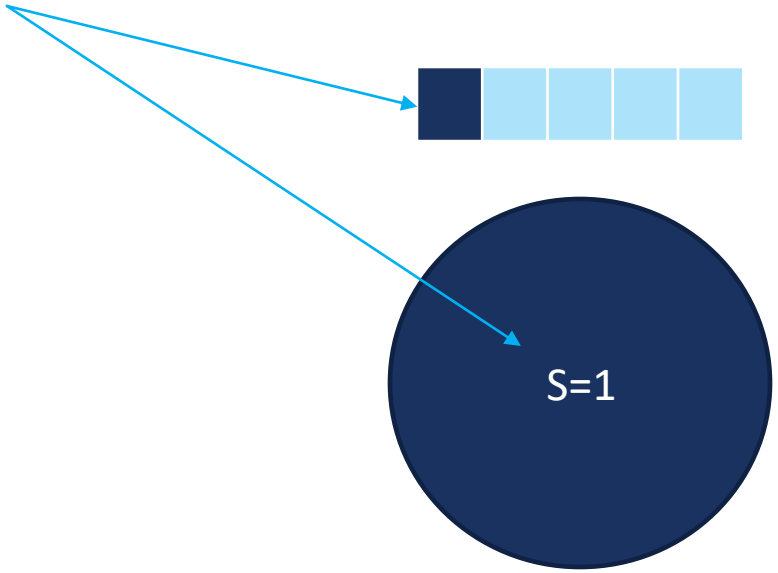
```
----------  Semaphore.Up  ----------
method Up ()
    var
      oldIntStat: int
      t: ptr to Thread
    oldIntStat = SetInterruptsTo (DISABLED)
    ------------------------------

    if count == 0x7fffffff
      FatalError ("Semaphore count overflowed during 'Up' operation")
    endIf
    count = count + 1
    if count <= 0
      t = waitingThreads.Remove ()
      ------------------------------

      t.status = READY
      readyList.AddToEnd (t)
    endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
  endMethod
```

- We said that a semaphore needs to be > 0 to avoid sleeping/waiting after a decrement …
- Why does this work?

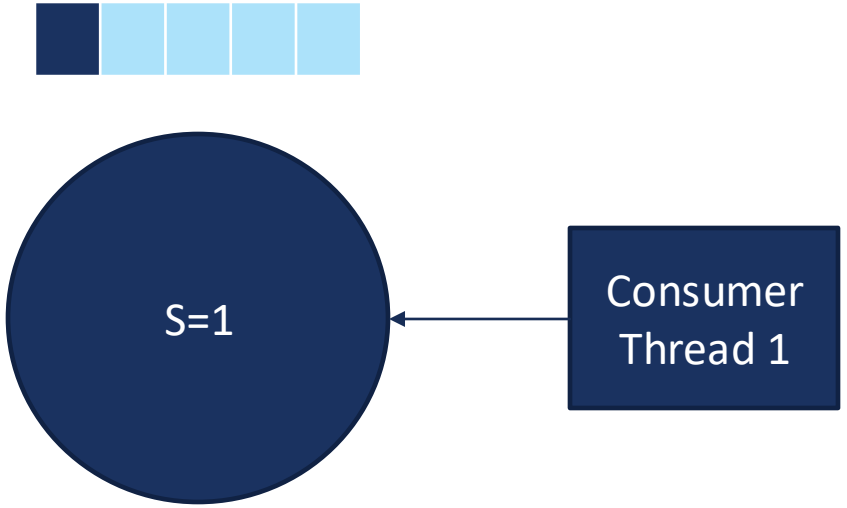
Semaphore needs to be > 0 after decrement to avoid sleeping … BUT it can "wake up" even if semaphore is < 0.

PRODUCER/CONSUMER SEMAPHORE

1 item ready to be consumed.

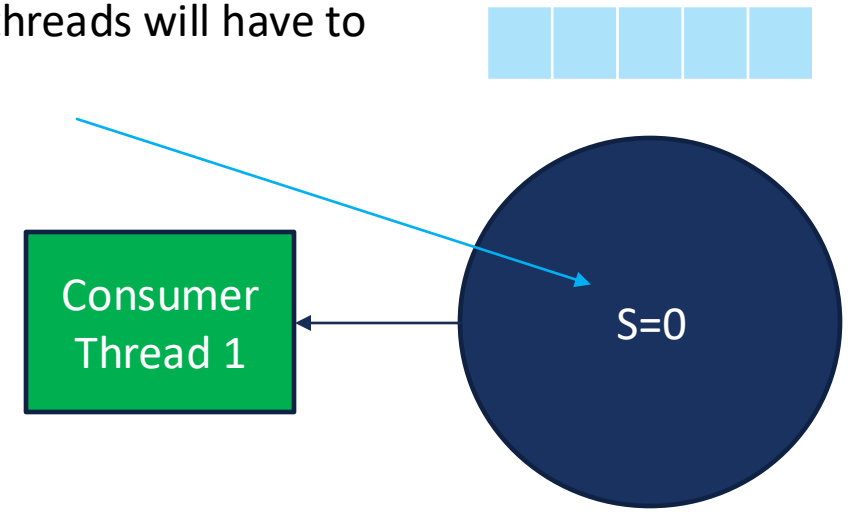


PRODUCER/CONSUMER SEMAPHORE



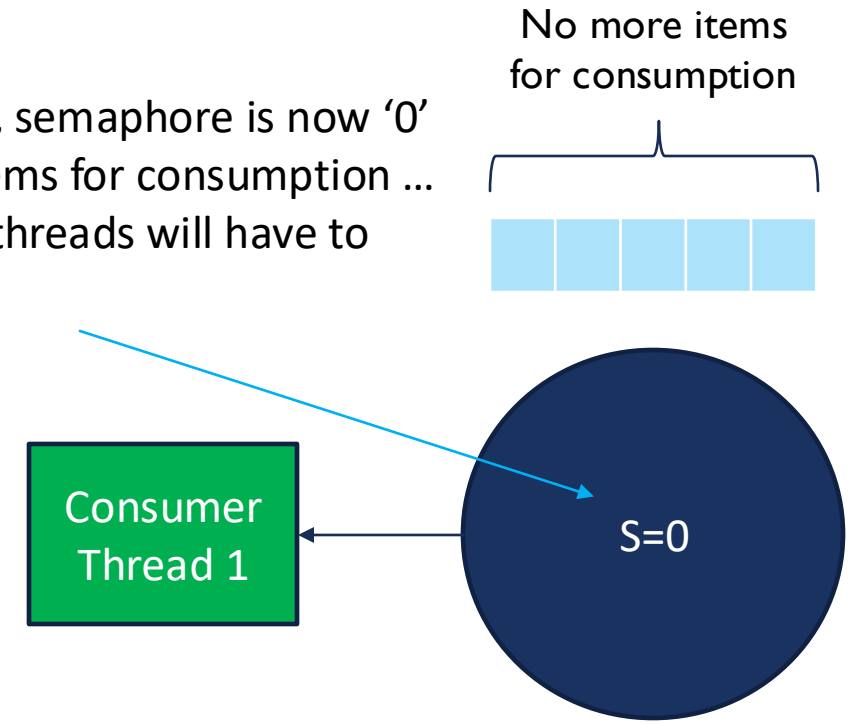
PRODUCER/CONSUMER SEMAPHORE

Item consumed, semaphore is now ‘0’ indicating no items for consumption … new consumer threads will have to wait.

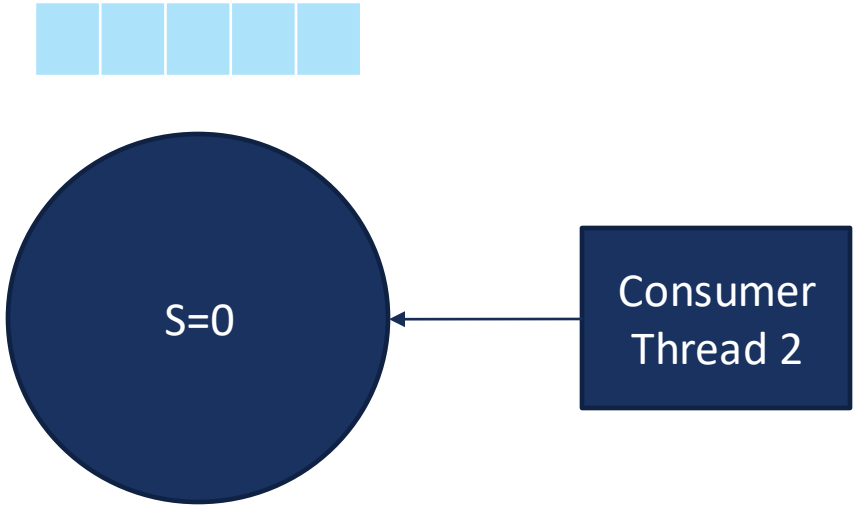


PRODUCER/CONSUMER SEMAPHORE

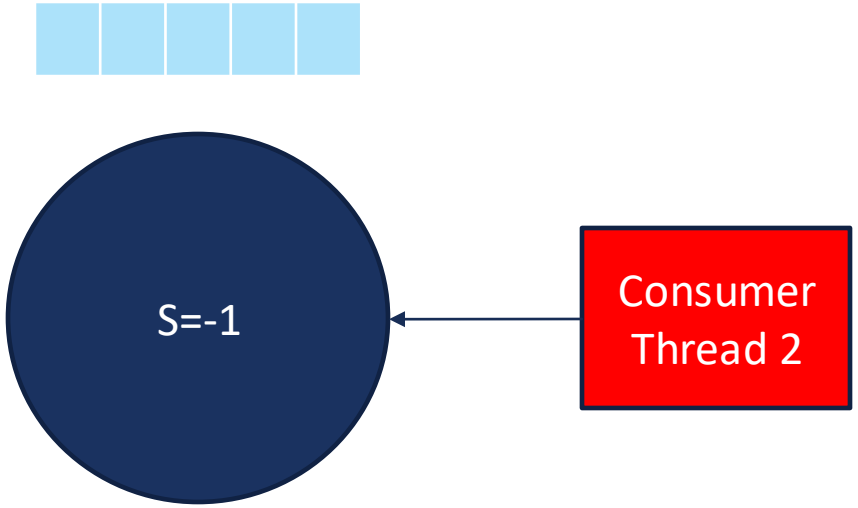
Item consumed, semaphore is now ‘0’ indicating no items for consumption … new consumer threads will have to wait.



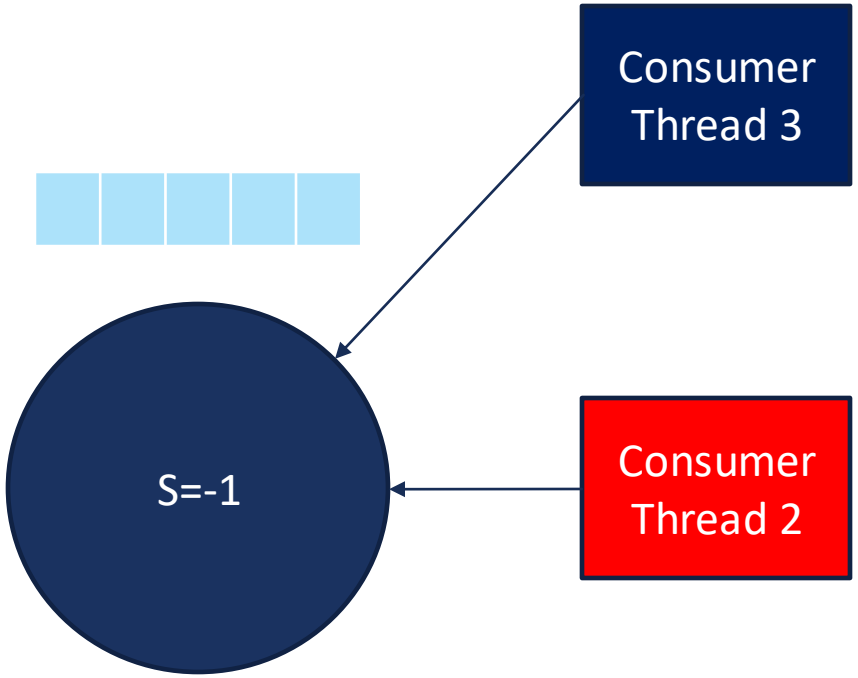
PRODUCER/CONSUMER SEMAPHORE



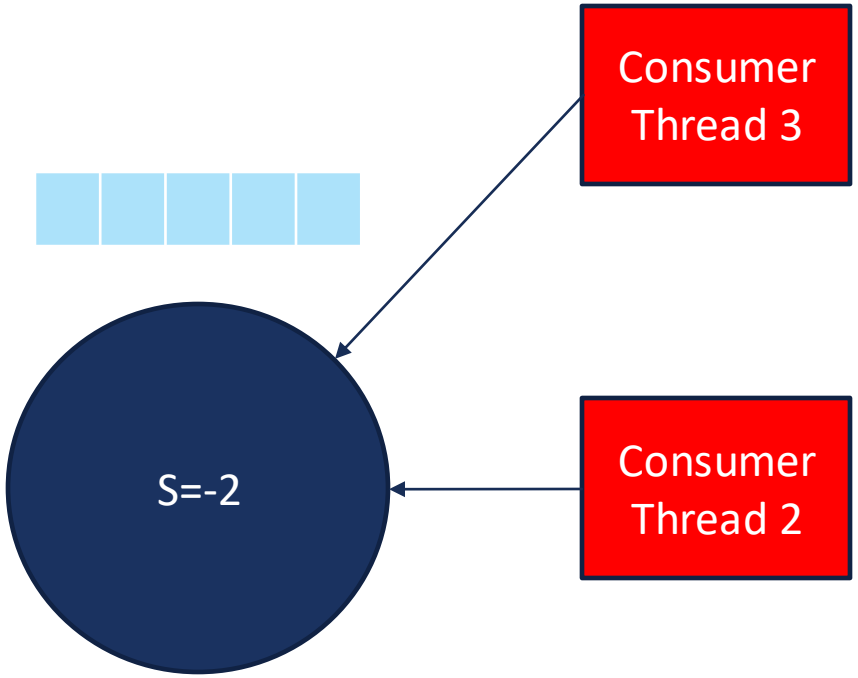
PRODUCER/CONSUMER SEMAPHORE



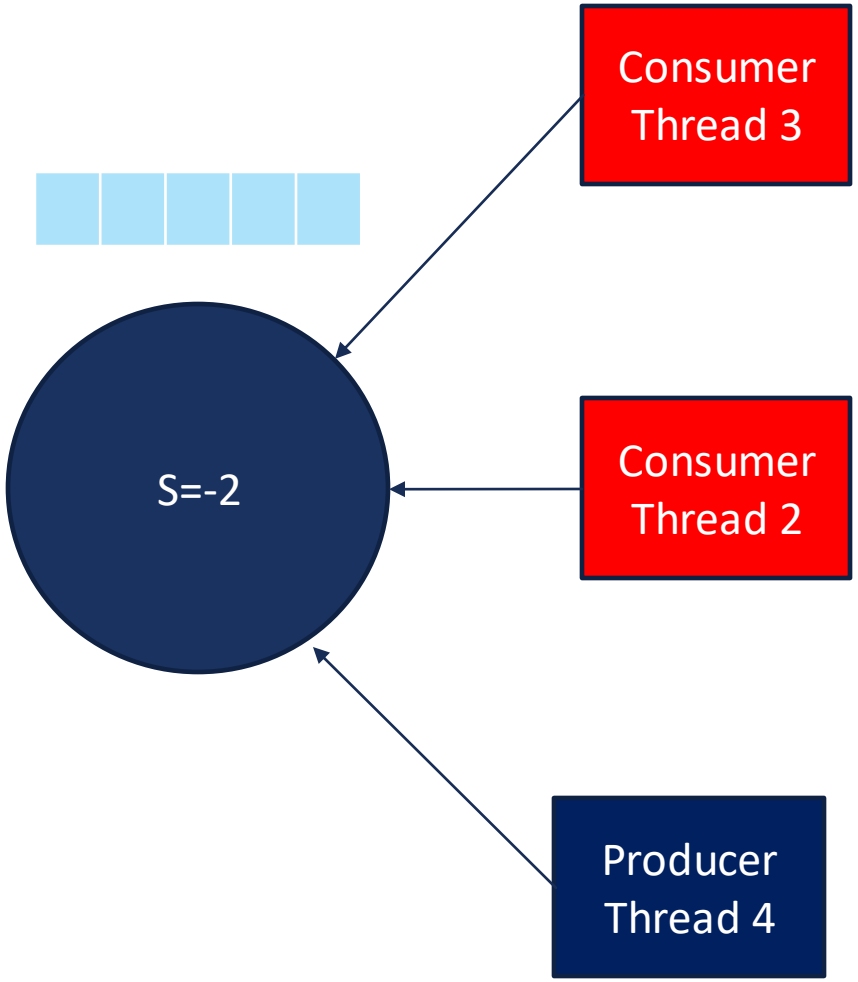
PRODUCER/CONSUMER SEMAPHORE



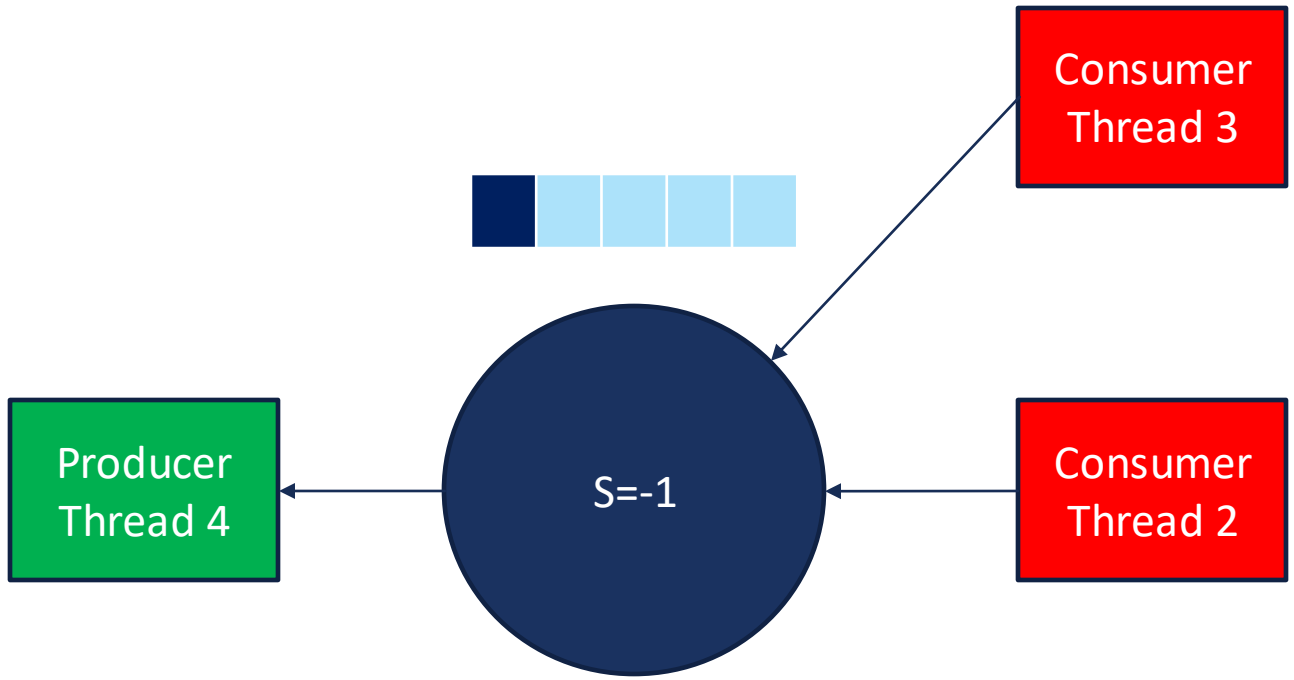
PRODUCER/CONSUMER SEMAPHORE



PRODUCER/CONSUMER SEMAPHORE

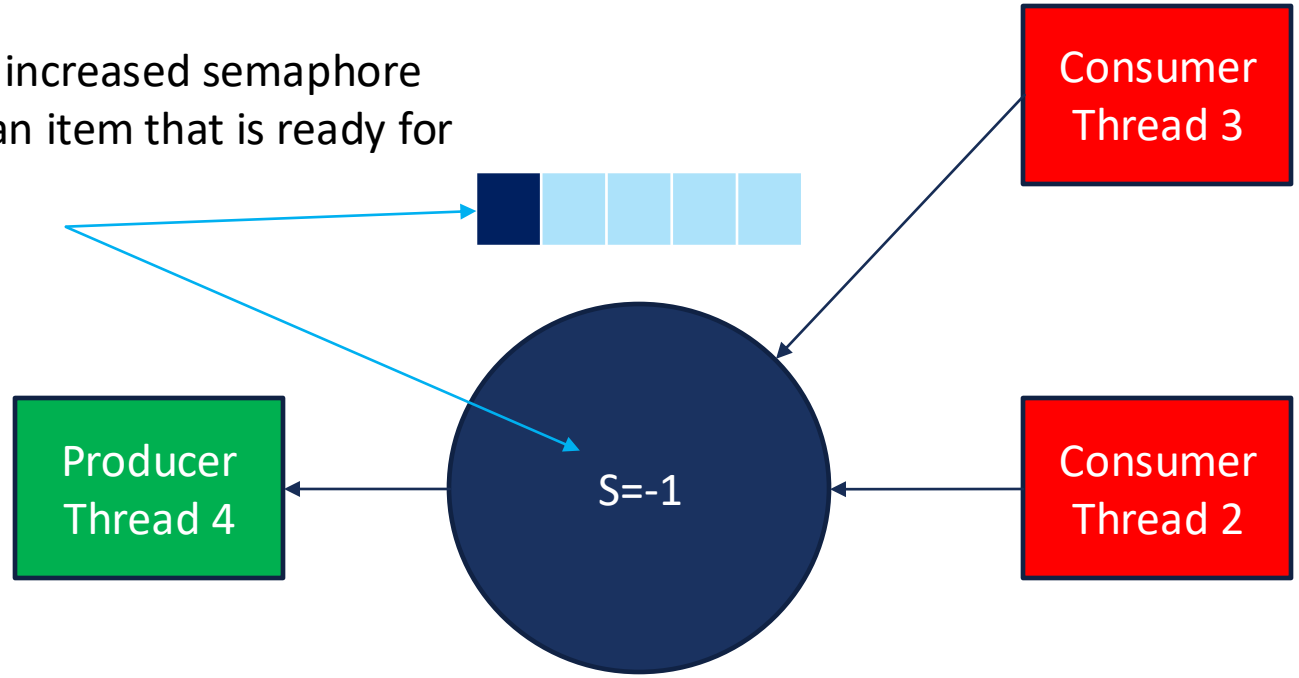


PRODUCER/CONSUMER SEMAPHORE



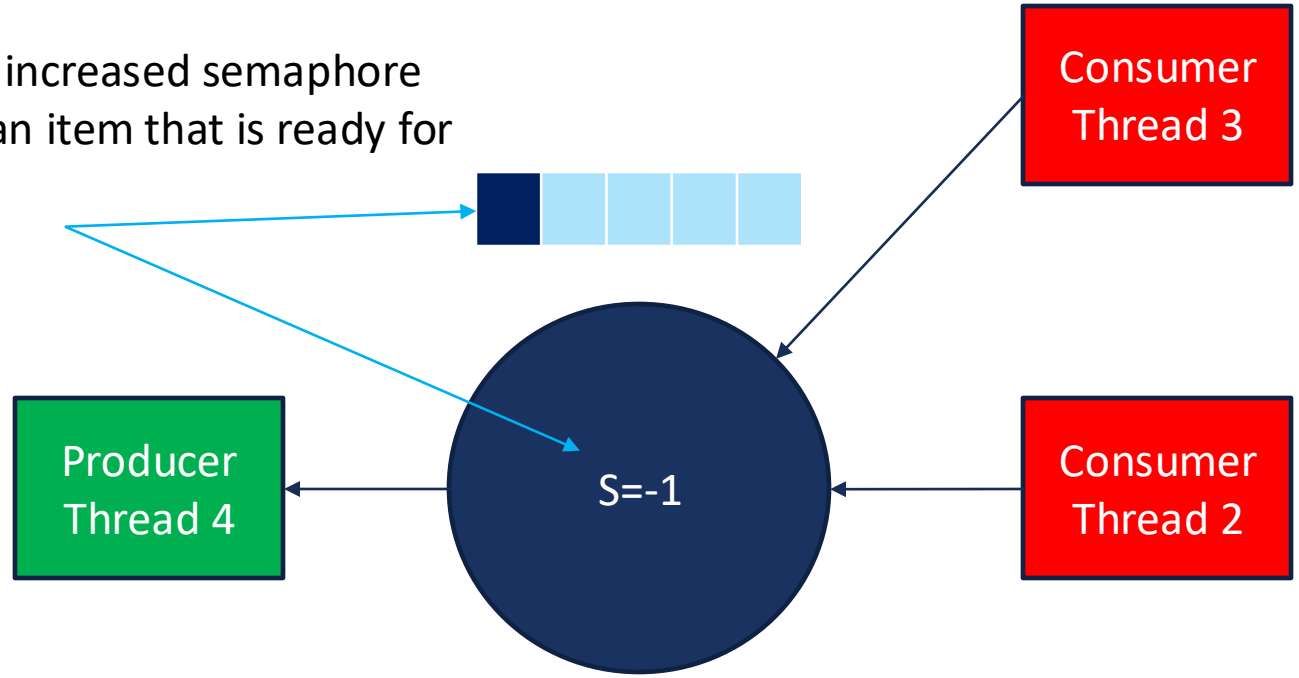
PRODUCER/CONSUMER SEMAPHORE

Producer thread increased semaphore by 1 and added an item that is ready for consumption.



PRODUCER/CONSUMER SEMAPHORE

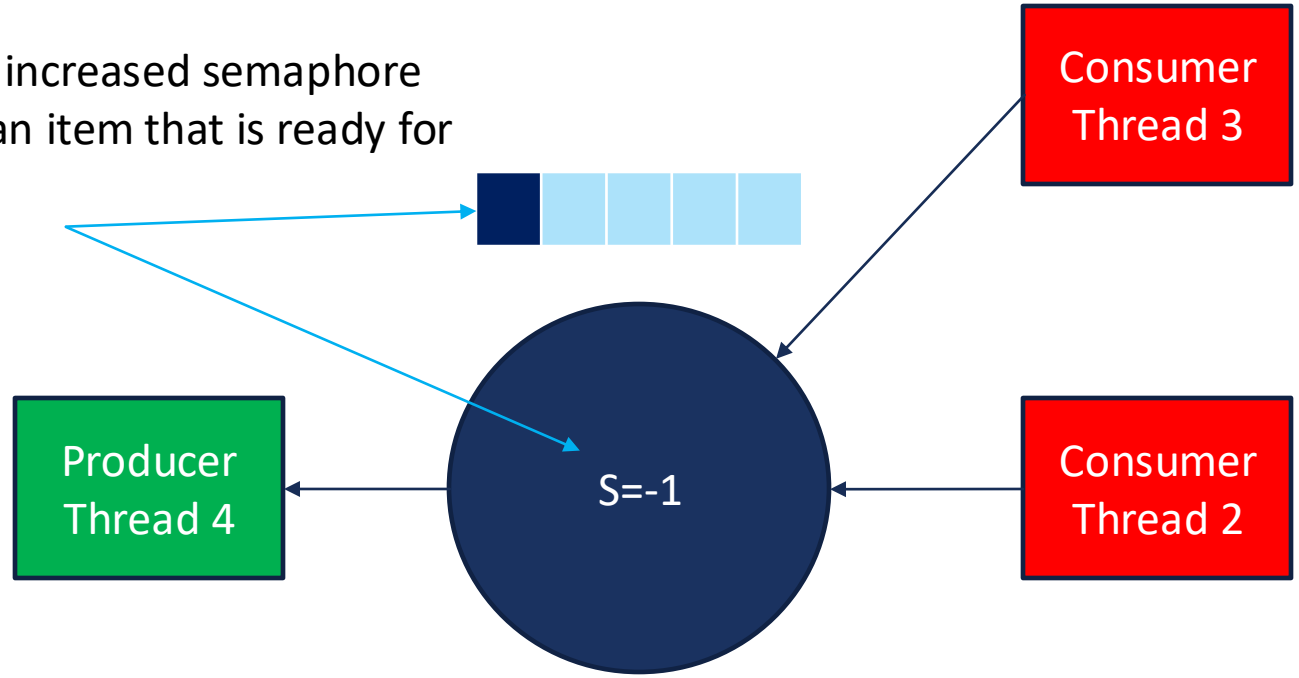
Producer thread increased semaphore by 1 and added an item that is ready for consumption.



- Even though the semaphore 'S' is negative, we have an item that is ready for consumption …

PRODUCER/CONSUMER SEMAPHORE

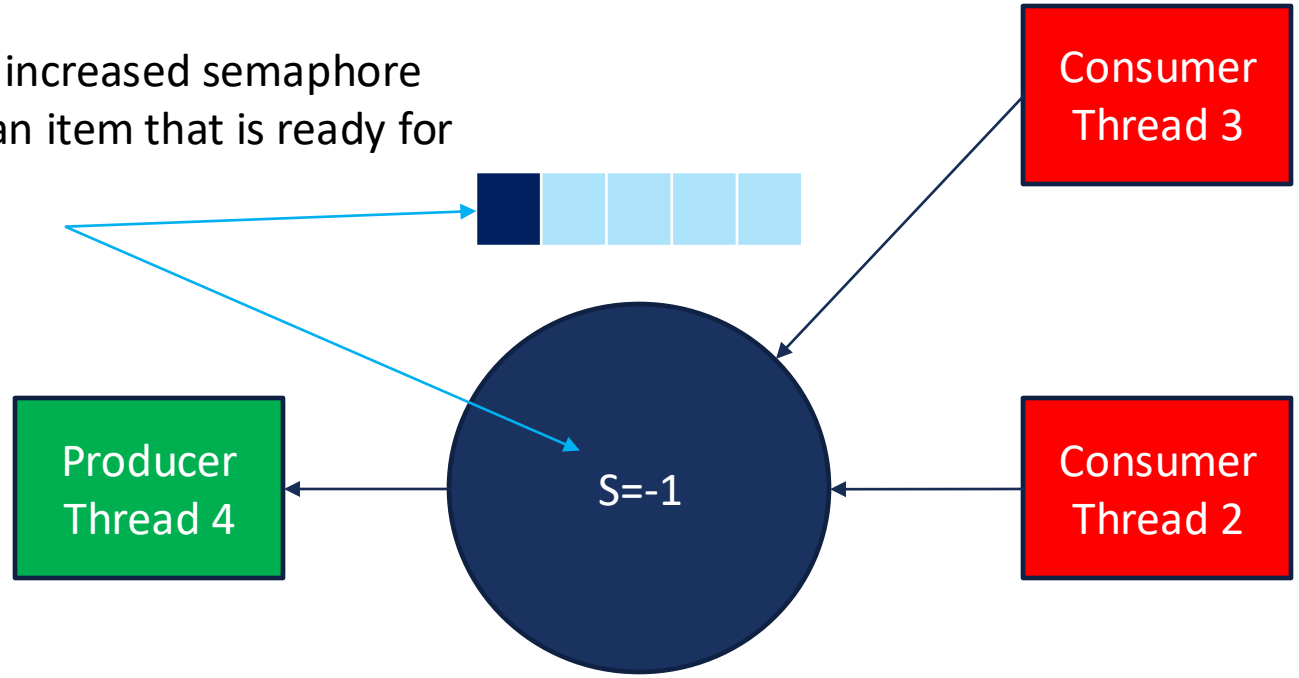
Producer thread increased semaphore by 1 and added an item that is ready for consumption.



- Even though the semaphore 'S' is negative, we have an item that is ready for consumption …
- We can let of the consumer threads execute!

PRODUCER/CONSUMER SEMAPHORE

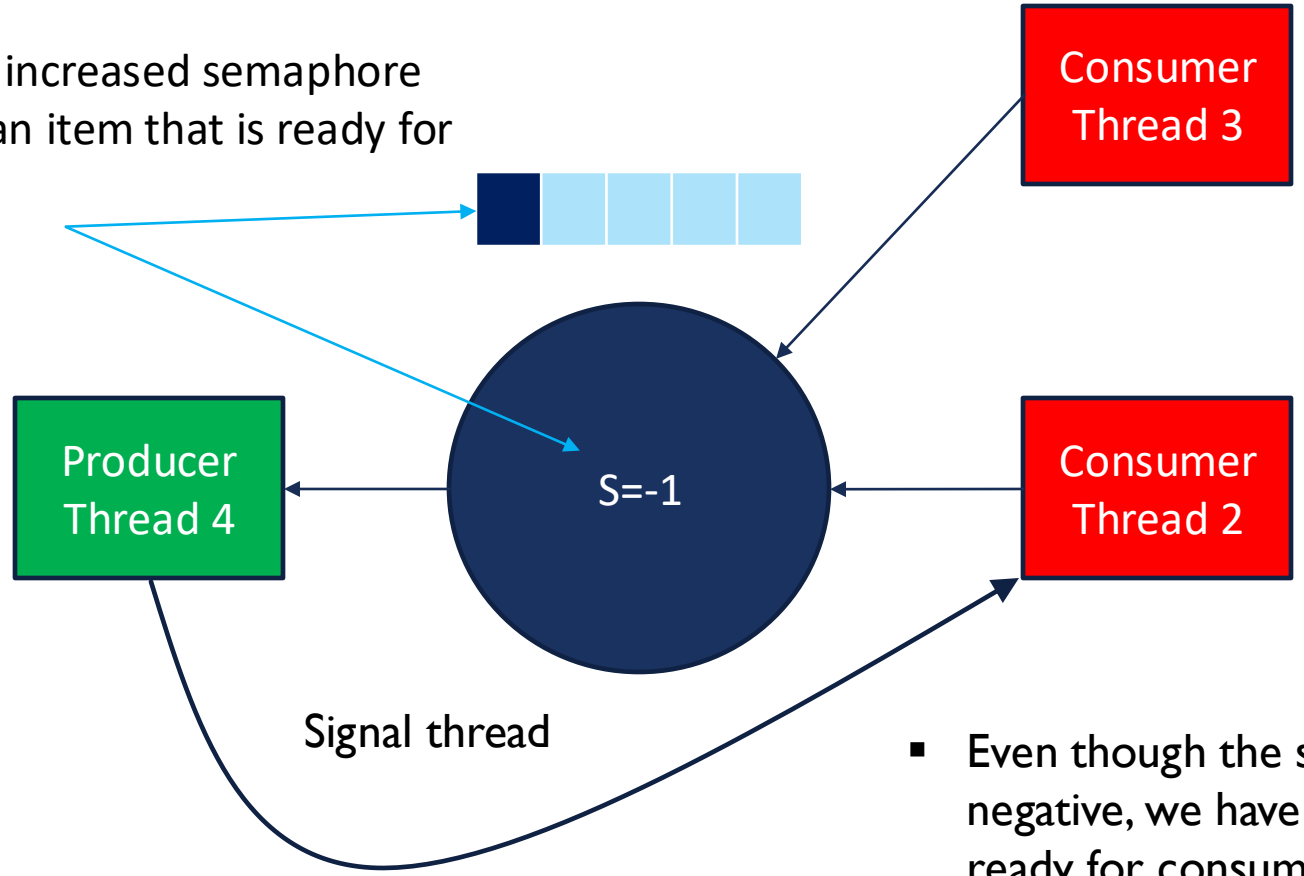
Producer thread increased semaphore by 1 and added an item that is ready for consumption.



- Even though the semaphore 'S' is negative, we have an item that is ready for consumption …
- We can let of the consumer threads execute!

PRODUCER/CONSUMER SEMAPHORE

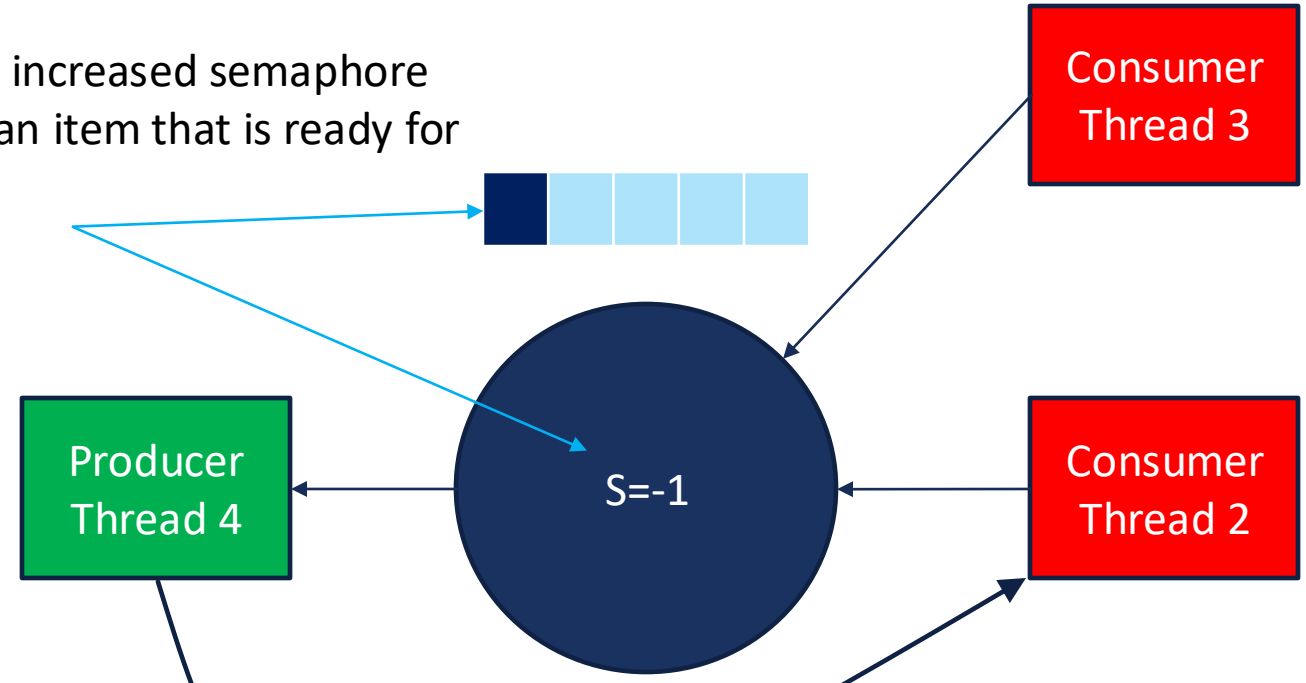
Producer thread increased semaphore by 1 and added an item that is ready for consumption.



- Even though the semaphore 'S' is negative, we have an item that is ready for consumption …
- We can let of the consumer threads execute!

PRODUCER/CONSUMER SEMAPHORE

Producer thread increased semaphore by 1 and added an item that is ready for consumption.



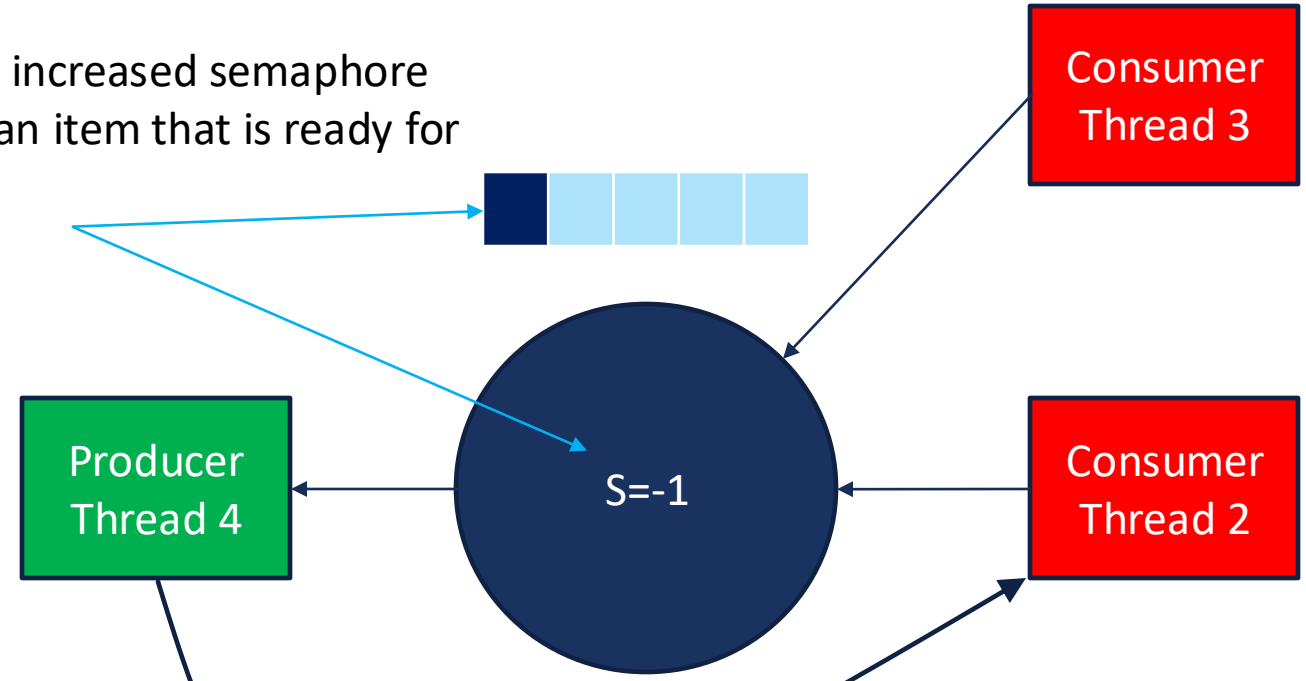

```
if count <= 0
    t = waitingThreads.Remove ()
    ------------------------------

    t.status = READY
    readyList.AddToEnd (t)
```

Signal thread

- Even though the semaphore 'S' is negative, we have an item that is ready for consumption …
- We can let of the consumer threads execute!

PRODUCER/CONSUMER SEMAPHORE

Producer thread increased semaphore by 1 and added an item that is ready for consumption.

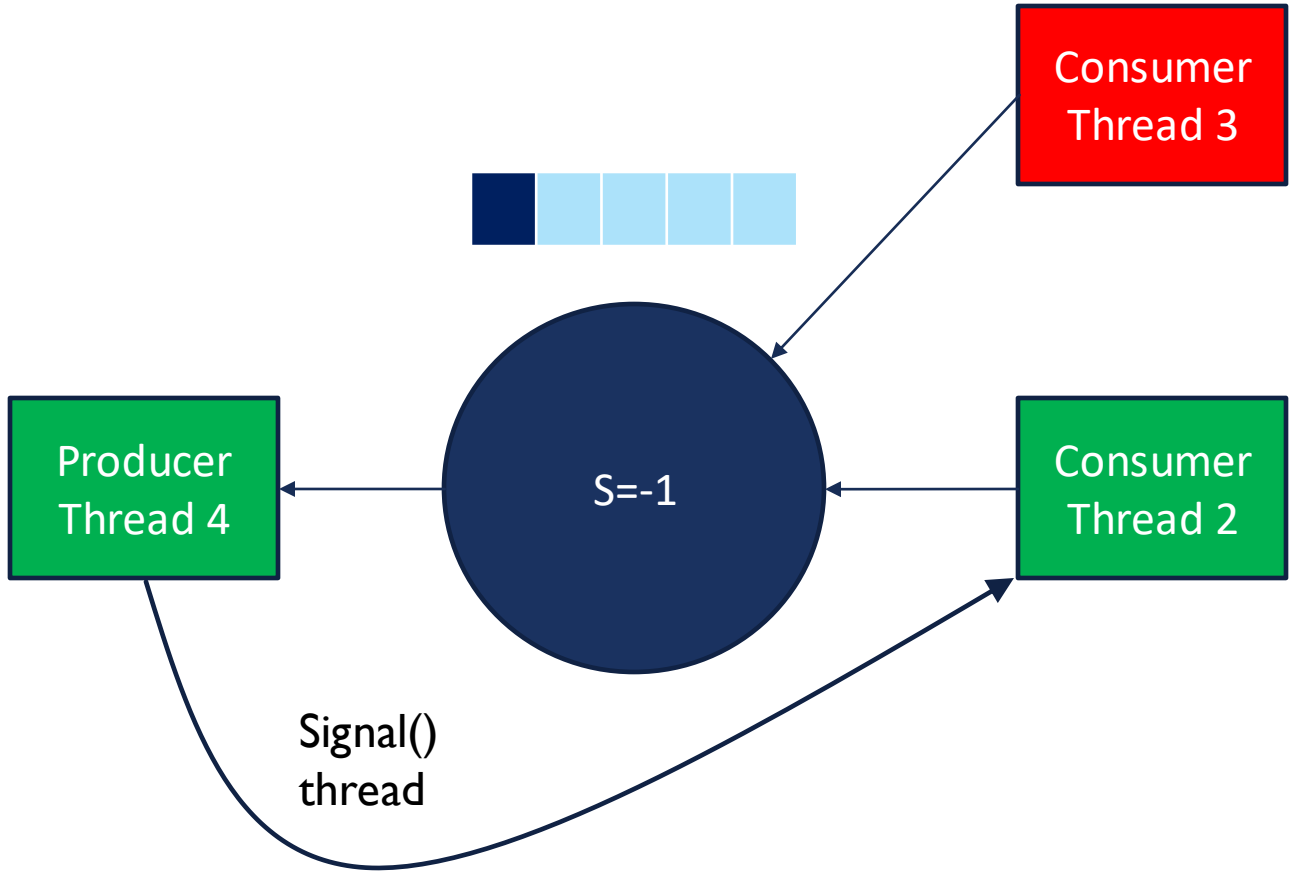



```
if count <= 0
    t = waitingThreads.Remove ()
    ------------------------------

    t.status = READY
    readyList.AddToEnd (t)
```

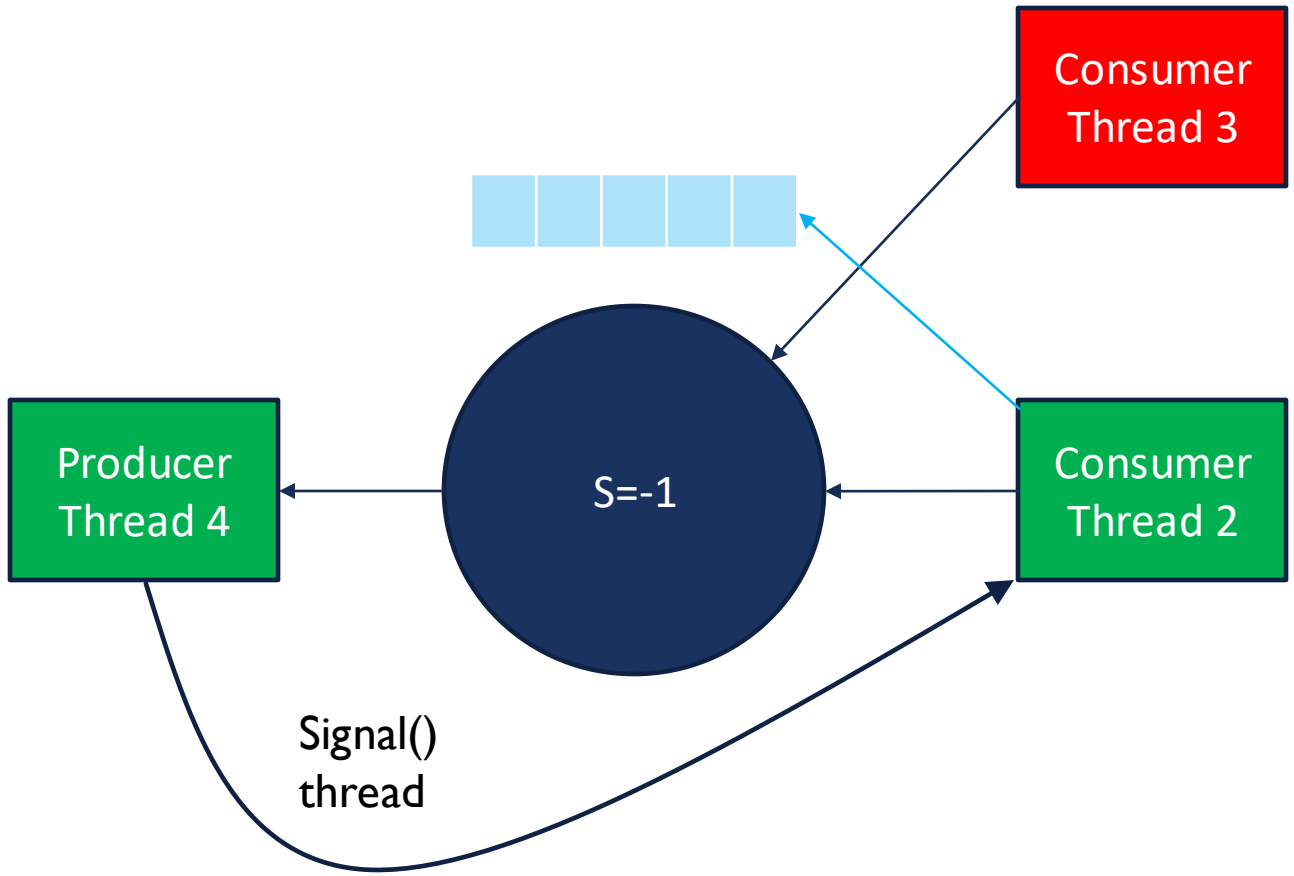
Signal() thread

- Even though the semaphore 'S' is negative, we have an item that is ready for consumption …
- We can let of the consumer threads execute!

PRODUCER/CONSUMER SEMAPHORE

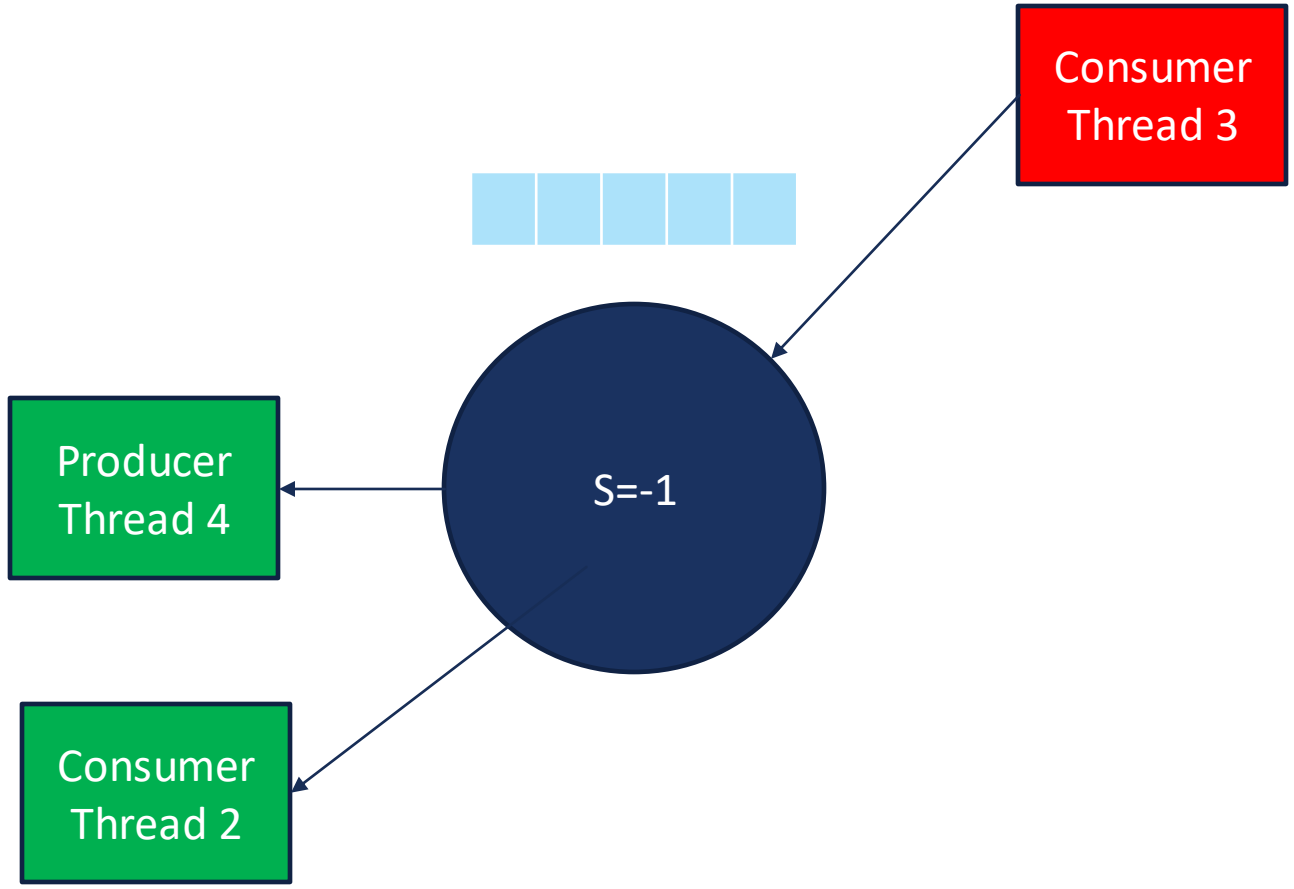


PRODUCER/CONSUMER SEMAPHORE

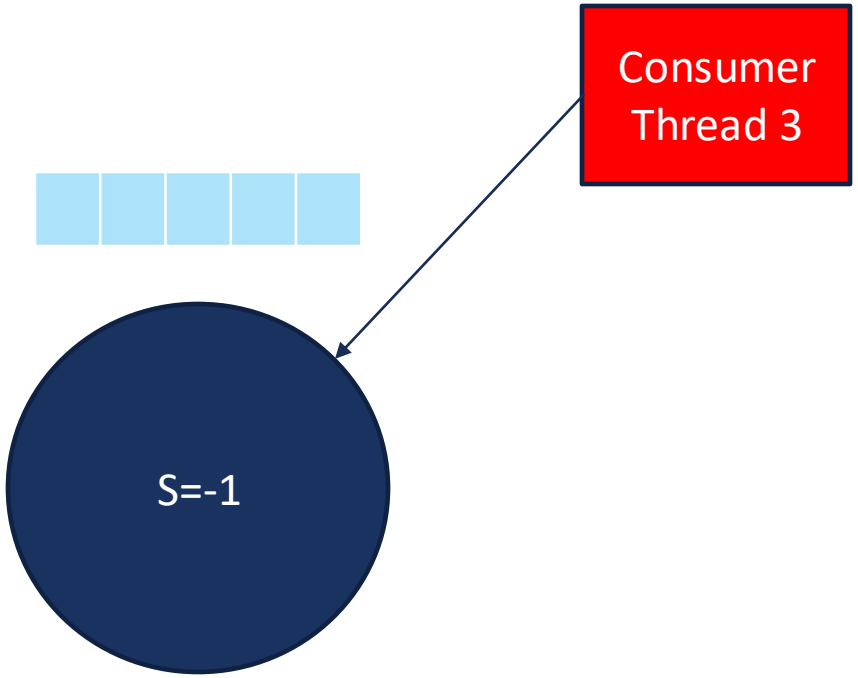


Consumer Thread 2 consumes the item

PRODUCER/CONSUMER SEMAPHORE



PRODUCER/CONSUMER SEMAPHORE



SEMAPHORE IMPLEMENTATION

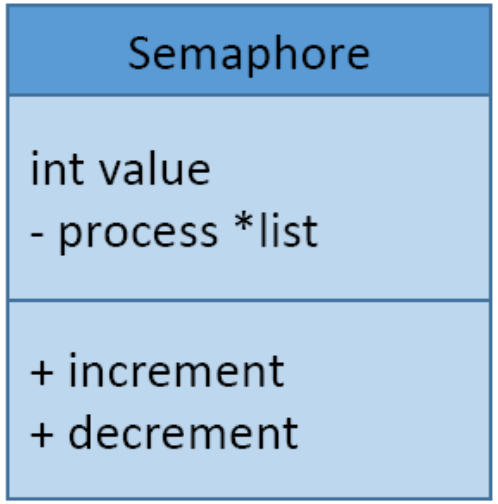
```
1. readyList.AddToEnd (t)
2. oldIntStat = SetInterruptsTo (DISABLED)
3. waitingThreads.AddToEnd (currentThread)
4. t = waitingThreads.Remove ()
5. currentThread.Sleep ()
```

```
----------  Semaphore.Up  ----------
method Up ()
    var
      oldIntStat: int
      t: ptr to Thread
    oldIntStat = SetInterruptsTo (DISABLED)
    ------------------------------

    if count == 0x7fffffff
      FatalError ("Semaphore count overflowed during 'Up' operation")
    endIf
    count = count + 1
    if count <= 0
      t = waitingThreads.Remove ()
      ------------------------------

      t.status = READY
      readyList.AddToEnd (t)
    endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
  endMethod
```

- This simply a check to make sure that there are waiting threads …

SEMAPHORE WITH WAITING QUEUE

```
typedef struct {
     int value;
     struct process *list;
} semaphore;

 wait(semaphore *S) {
            S->value--;
            if (S->value < 0) {
                  add this process to S->list;
                  block();
            }
 }

signal(semaphore *S) {
         S->value++;
         if (S->value <= 0) {
               remove a process P from S->list;
               wakeup(P);
         }
}
```

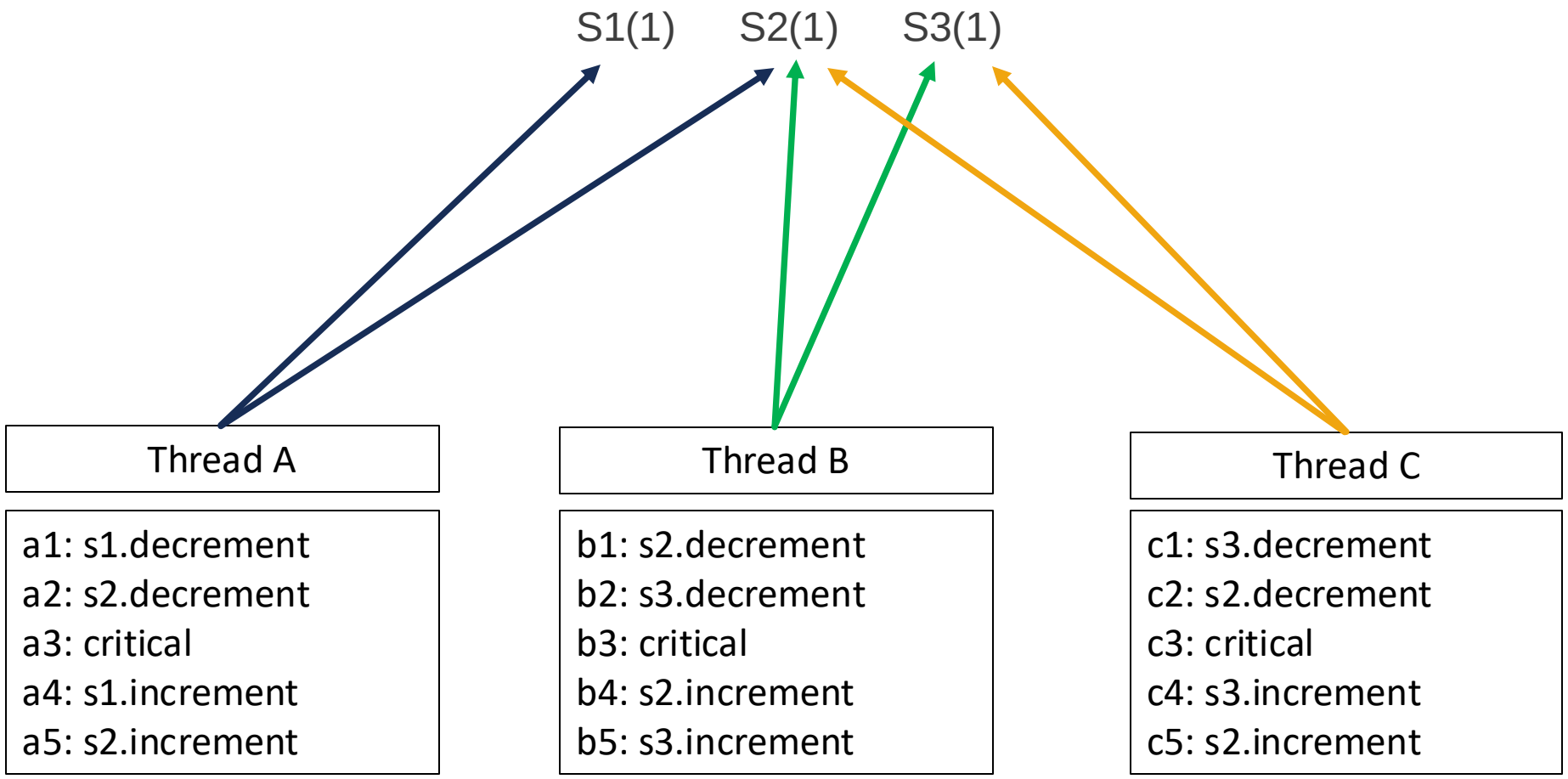


DEADLOCK POSSIBILITY

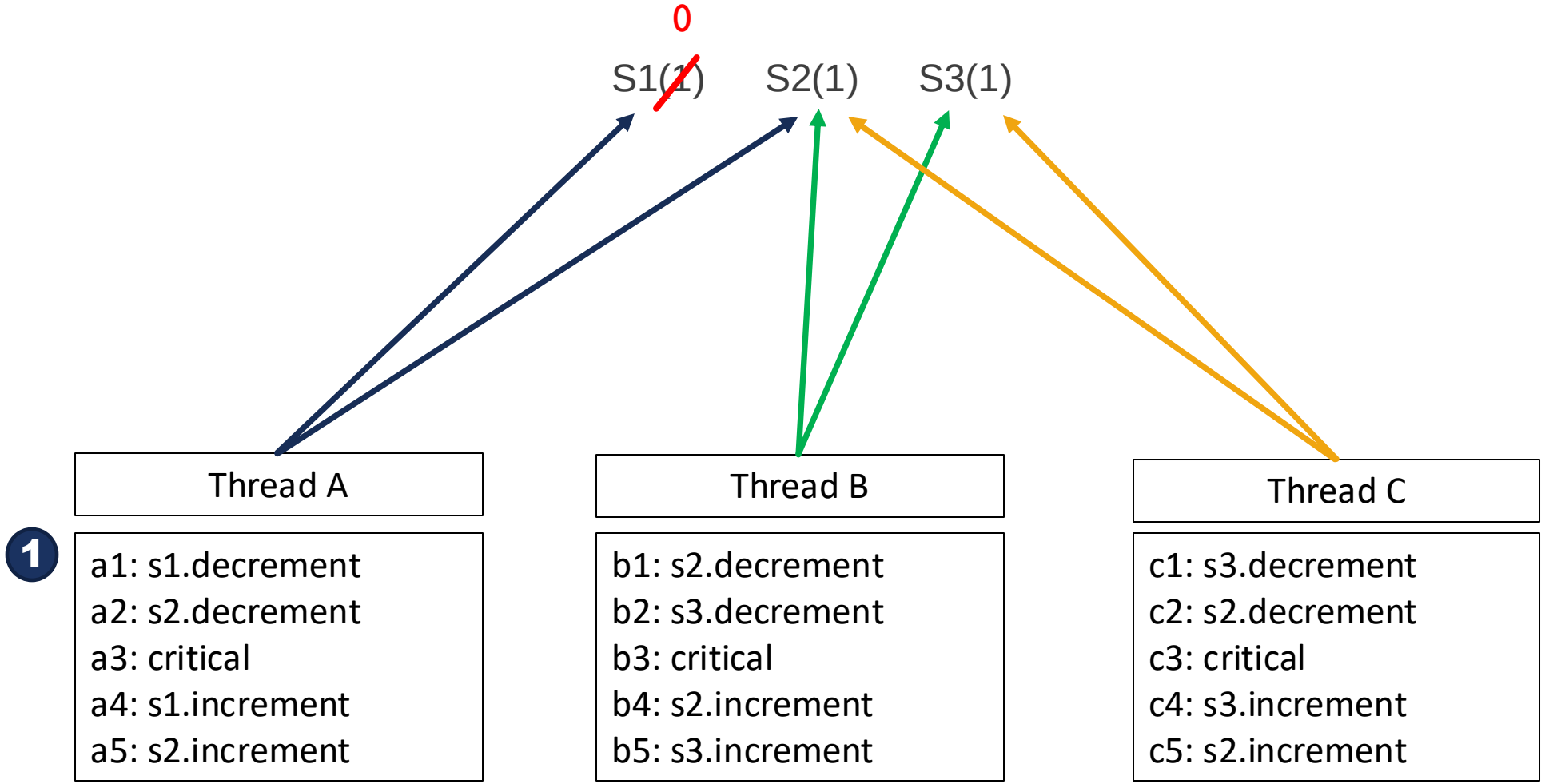
S1(1) S2(1) S3(1)

Thread A	Thread B	Thread C
a1: s1.decrement a2: s2.decrement a3: critical a4: s1.increment a5: s2.increment	b1: s2.decrement b2: s3.decrement b3: critical b4: s2.increment b5: s3.increment	c1: s3.decrement c2: s2.decrement c3: critical c4: s3.increment c5: s2.increment

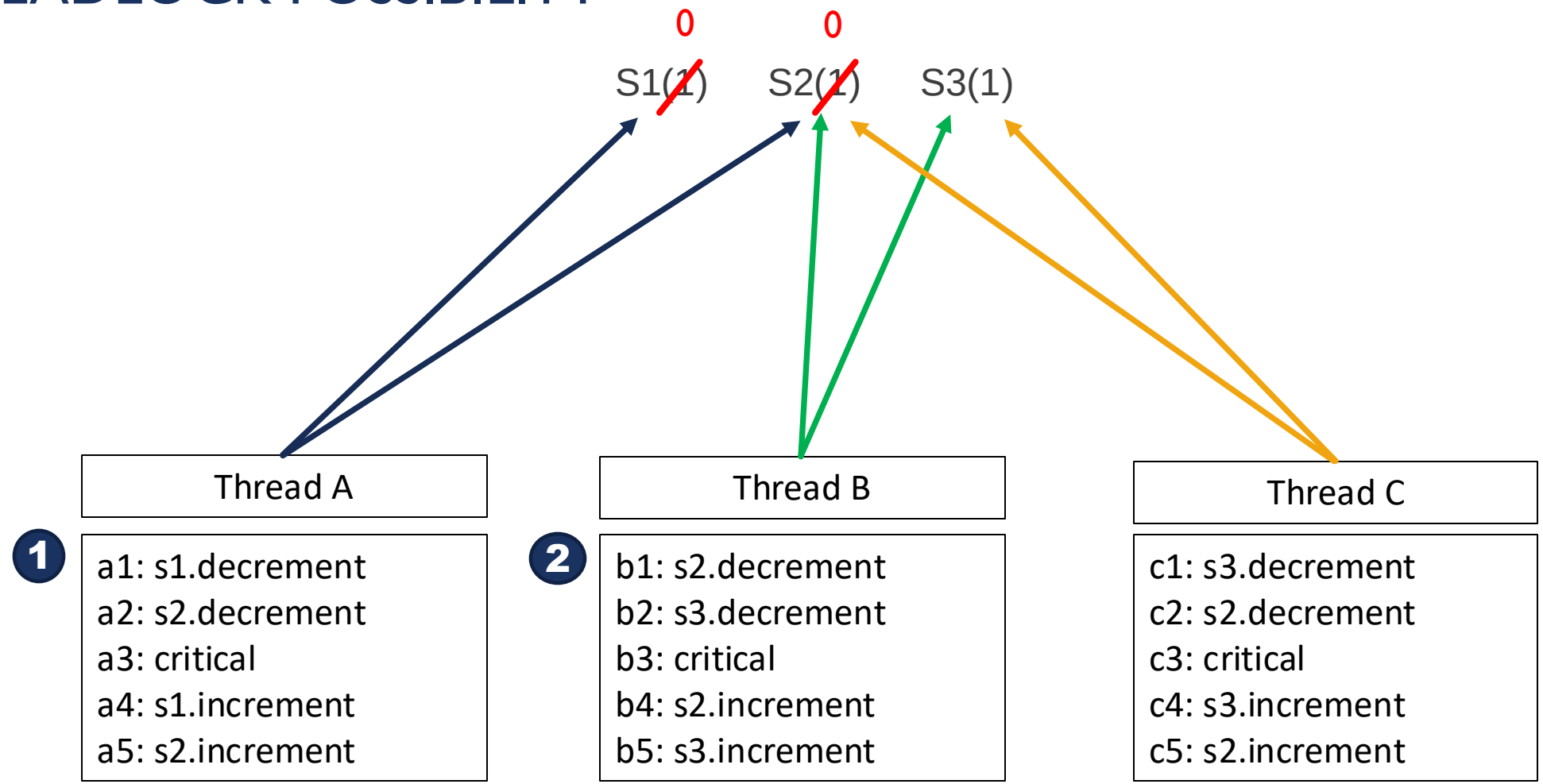
DEADLOCK POSSIBILITY



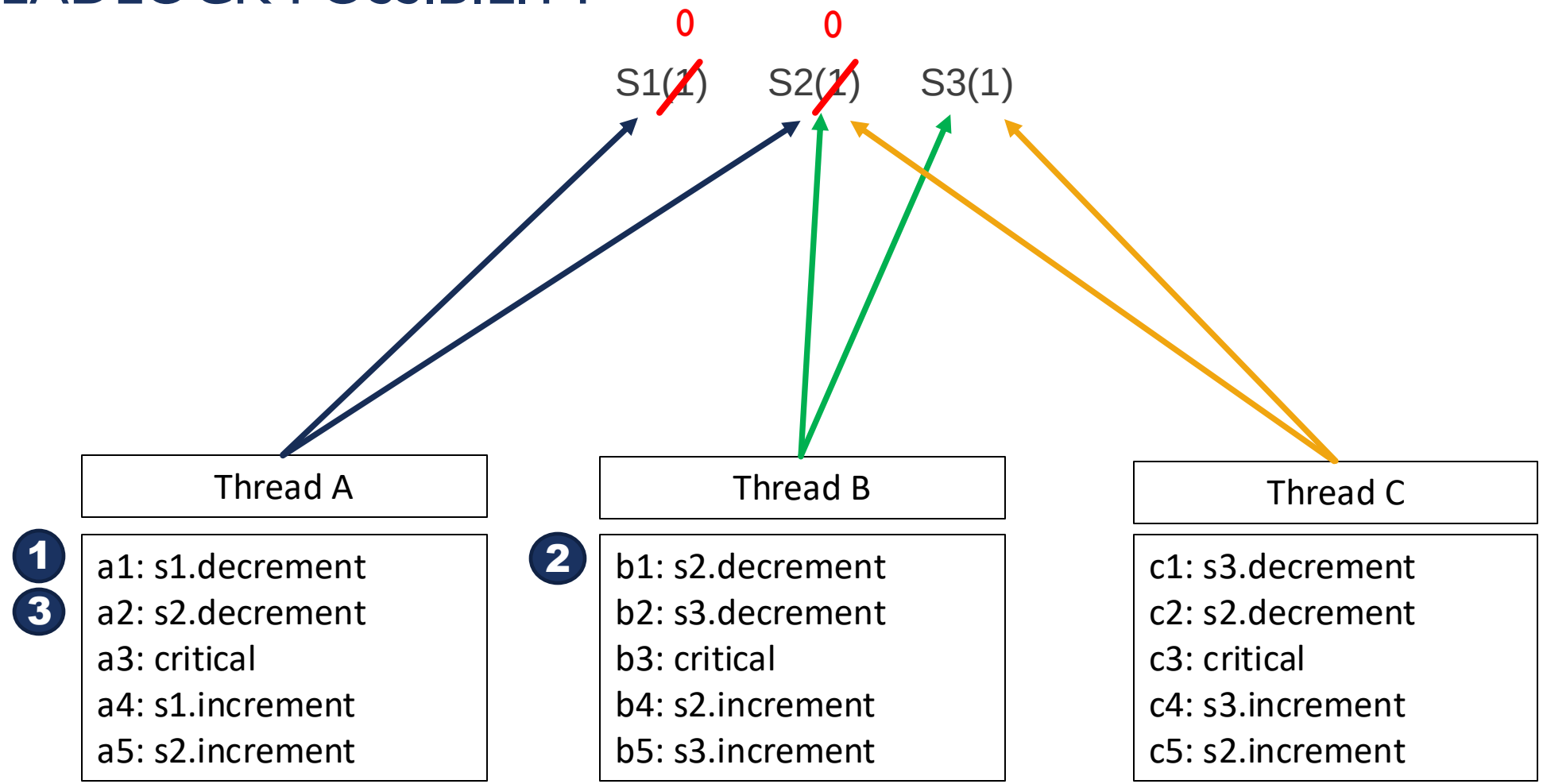
DEADLOCK POSSIBILITY



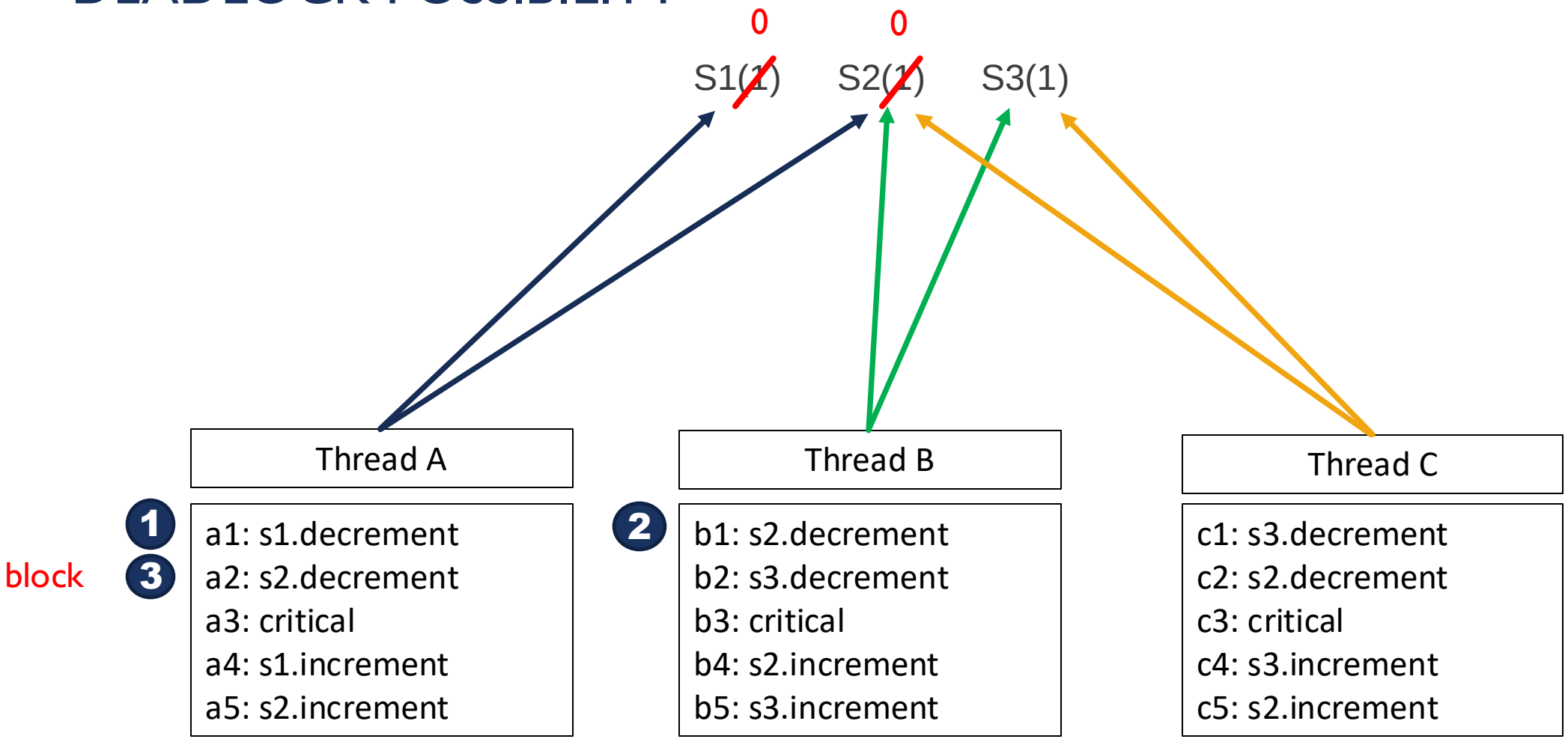
DEADLOCK POSSIBILITY



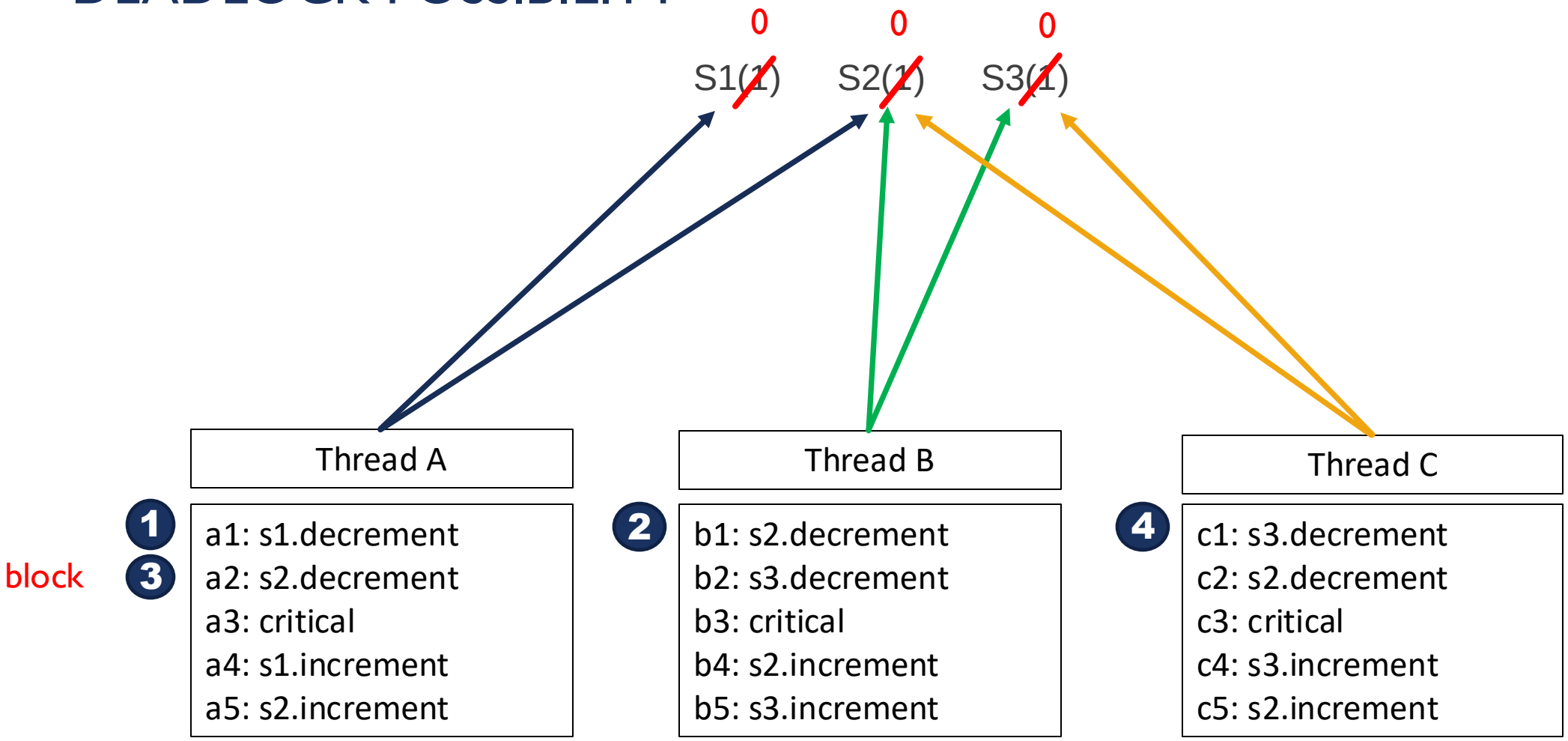
DEADLOCK POSSIBILITY



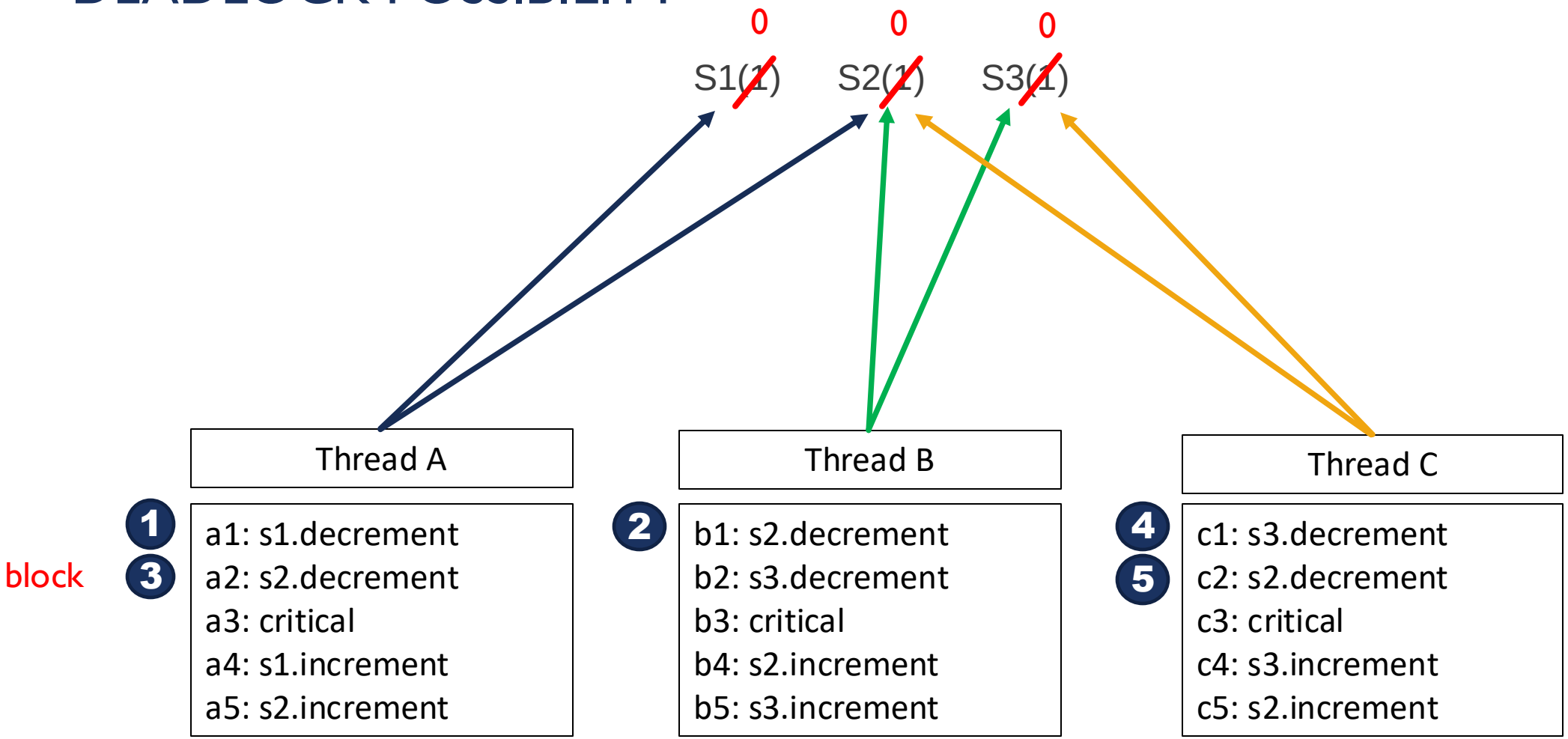
DEADLOCK POSSIBILITY



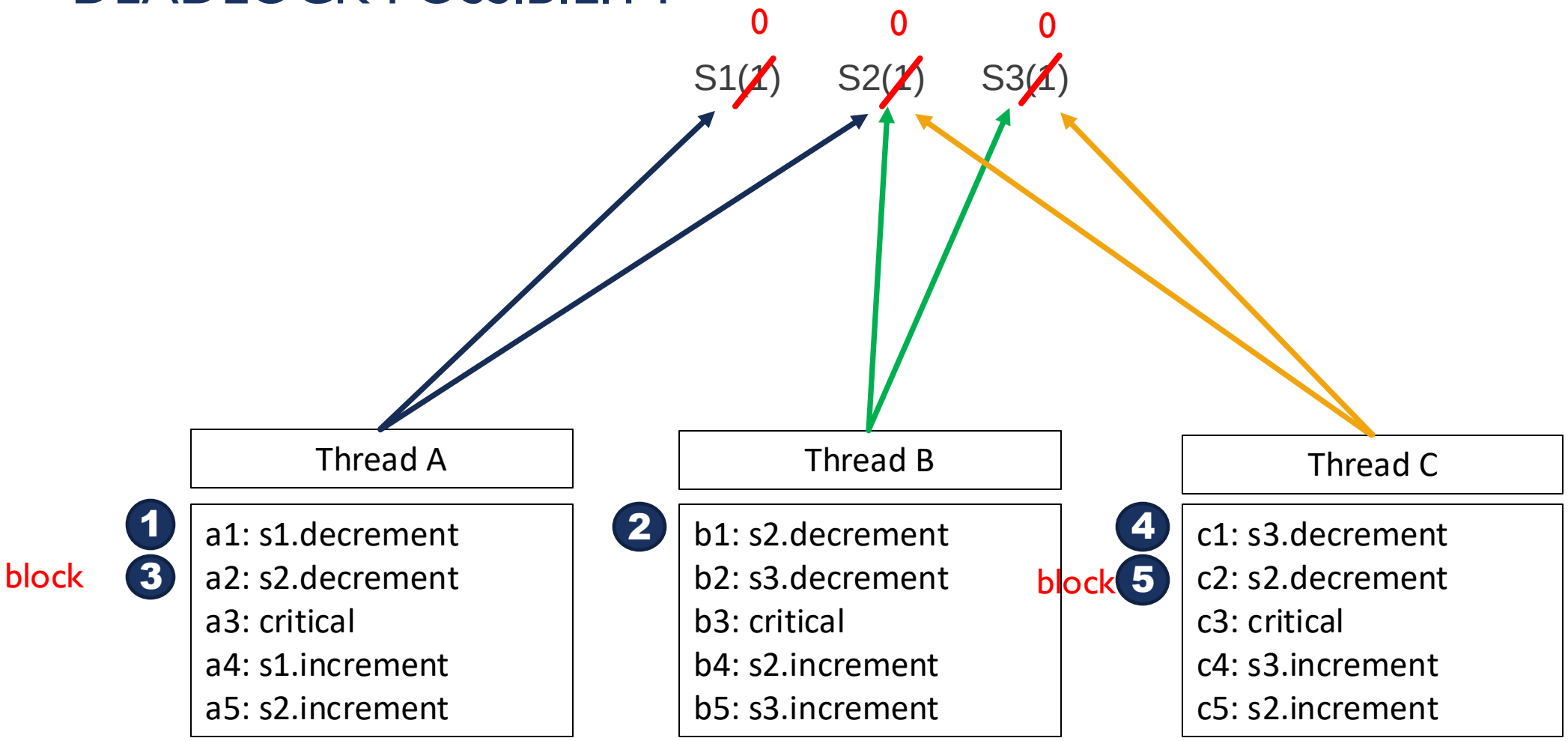
DEADLOCK POSSIBILITY



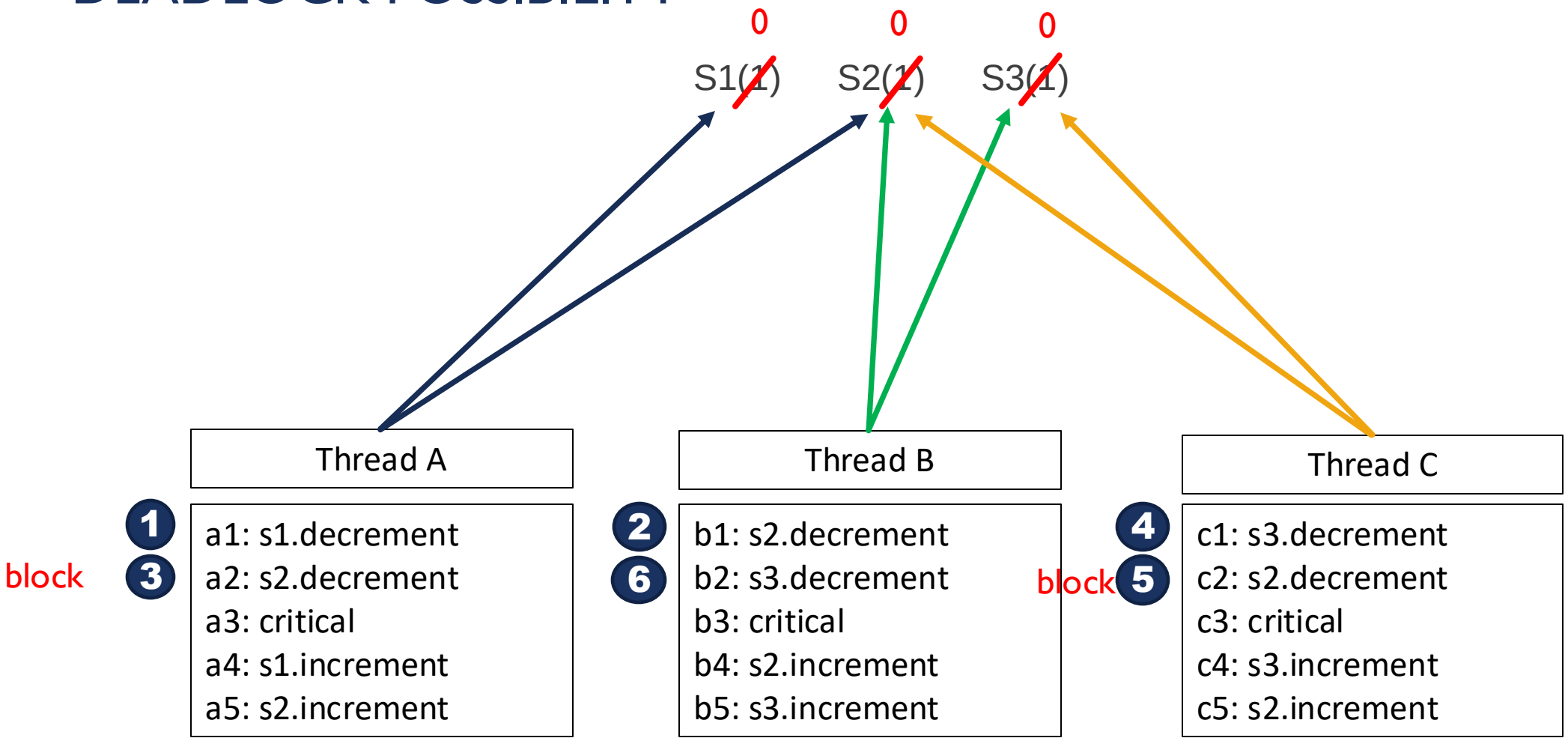
DEADLOCK POSSIBILITY



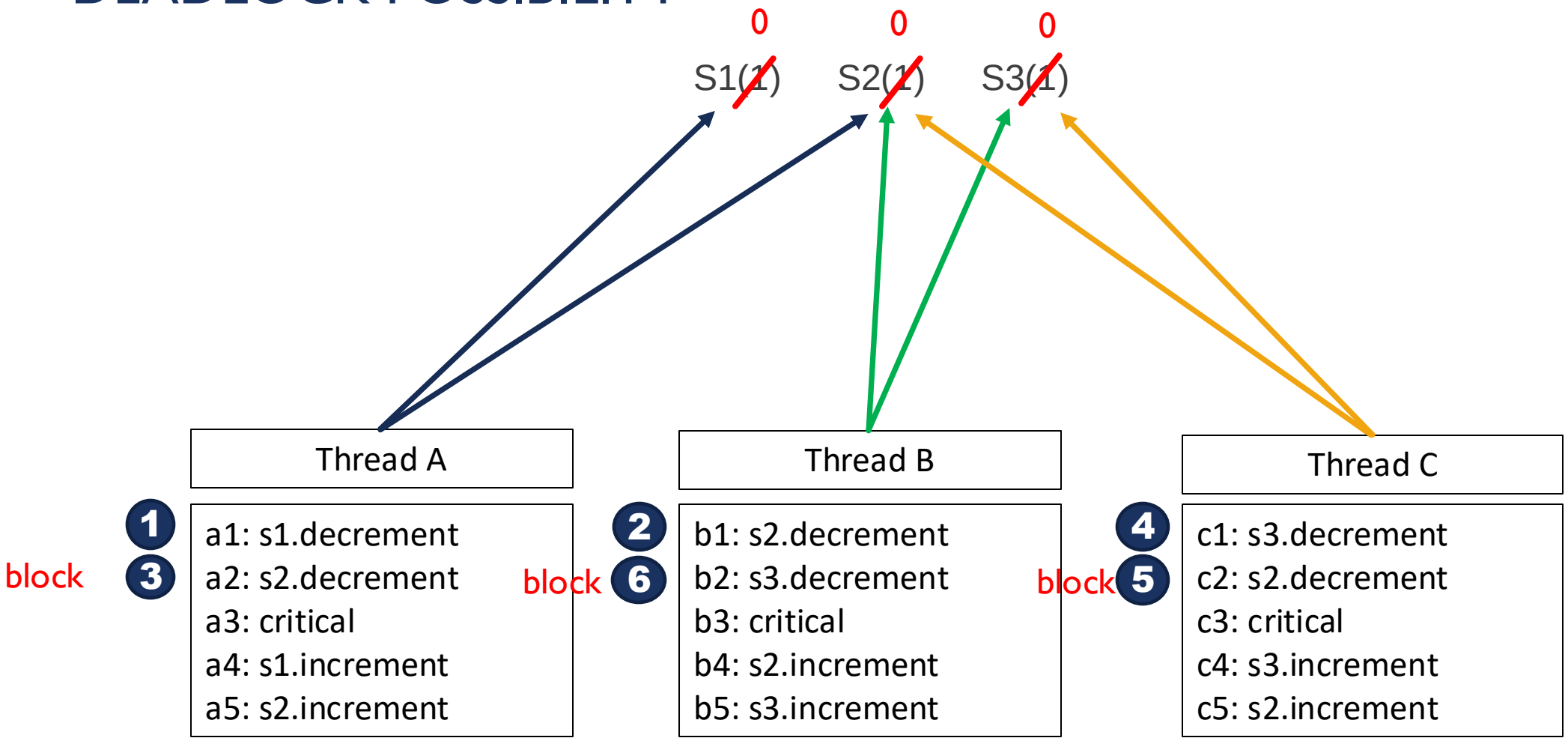
DEADLOCK POSSIBILITY



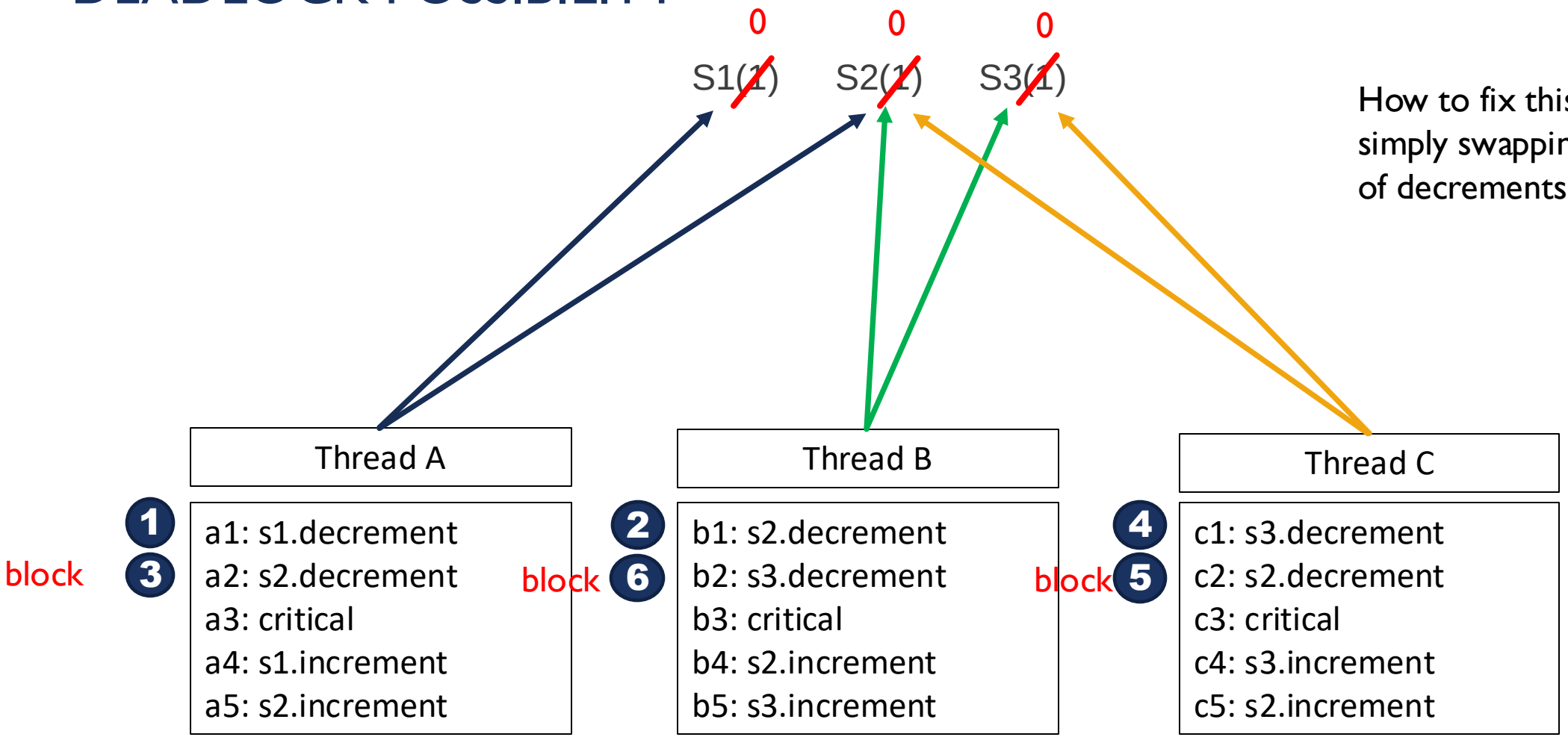
DEADLOCK POSSIBILITY



DEADLOCK POSSIBILITY

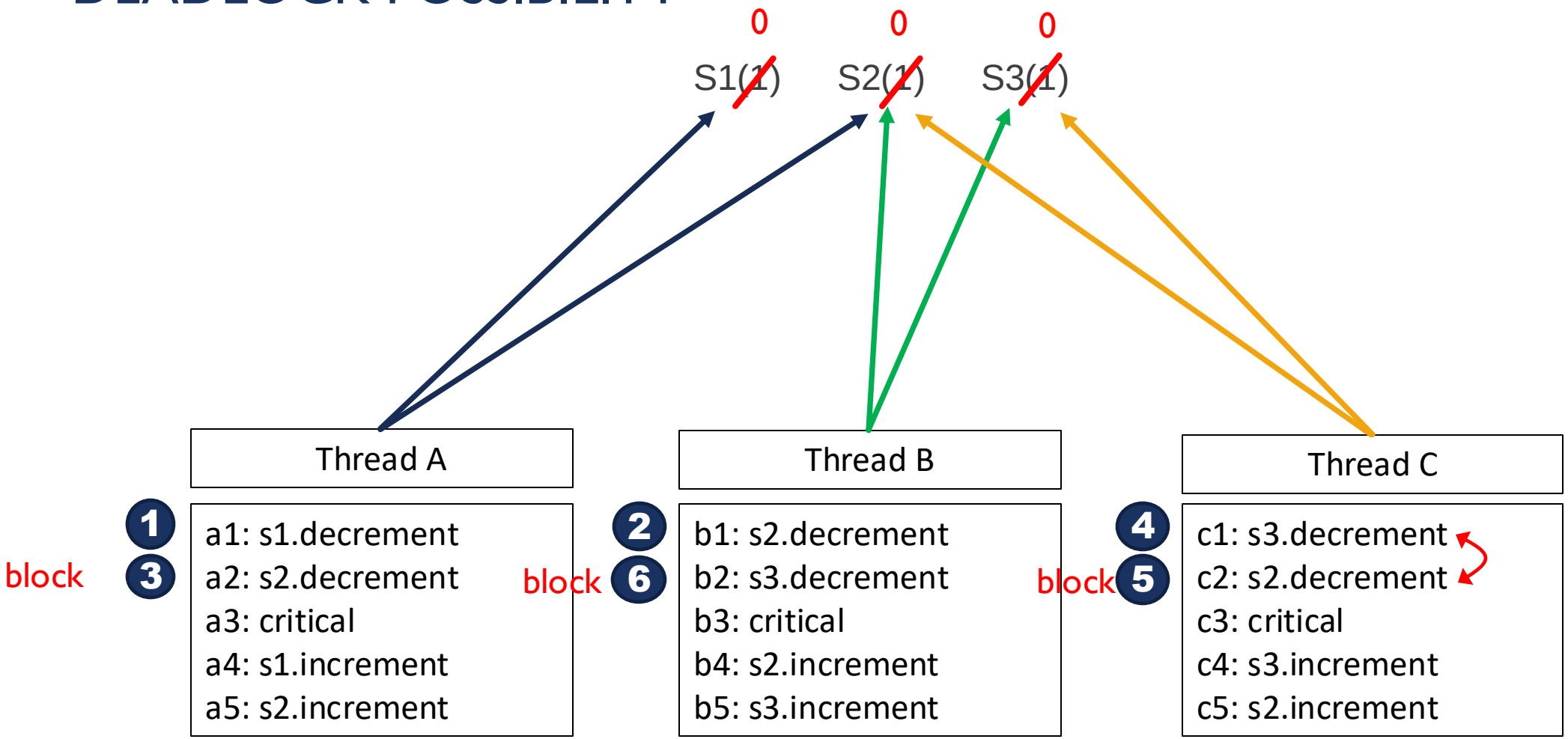


DEADLOCK POSSIBILITY



How to fix this by simply swapping order of decrements?

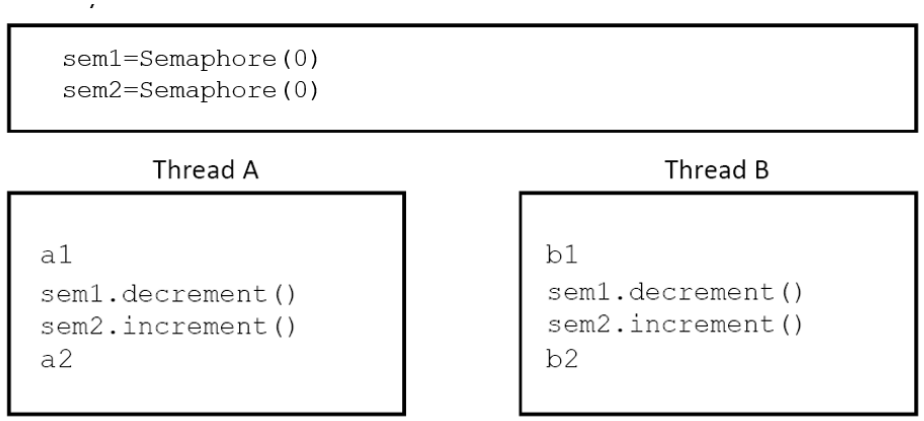
DEADLOCK POSSIBILITY



Multiple shared resources should be accessed in the same order.

SEMAPHORES COMPLICATION

- Each thread is responsible for writing its critical section.
- Semaphores are shared among all threads.
- Threads needed to manually increment and decrement semaphores while ensuring correct usage.
- With large programs and many threads, this becomes unmanageable and more prone to error.

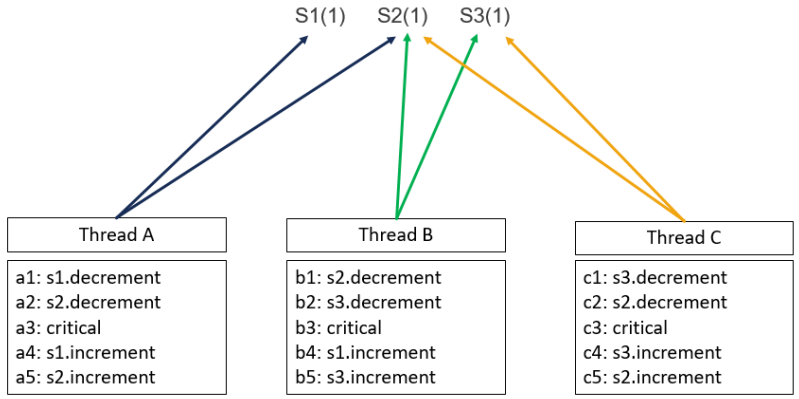



```
while (true) {
item = produce_item;
while
(counter == BUFFER_SIZE) {}/*
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
counter++;
}
```

Critical Section

```
while (true) {
while (counter == 0) {}/* do r
item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
counter--;
consume_item(item);
```

Critical Section



MONITOR

```
in = out = 0;
Mutex mutex=Mutex.Init();
```

Thread A

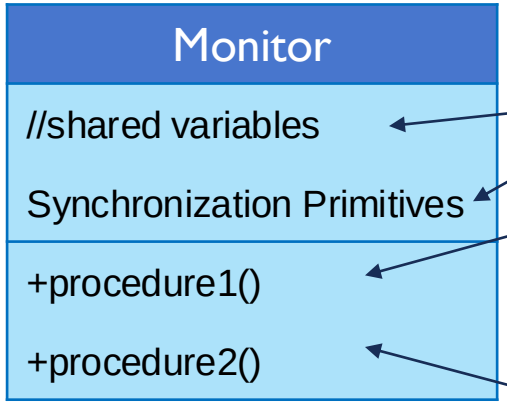
```
while (true) {
      item = produce_item;
      while
      (counter == BUFFER_SIZE) {}/* do nothing */;
      buffer[in] = item;
      in = (in + 1) % BUFFER_SIZE;
      mutex.acquire()
      counter++;
      mutex.release()
}
```

Thread B

```
while (true) {
      while (counter == 0) {}/* do nothing
      item = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      mutex.acquire()
      counter--;
      mutex.release()
      consume_item(item);
      }
```

THE MONITOR SOLUTION

Whatever synch primitives we need, mutex locks, semaphores …




```
in = out = 0;
Mutex mutex=Mutex.Init();

while (true) {
      item = produce_item;
      while
      (counter == BUFFER_SIZE) {}/* do nothing */;
      buffer[in] = item;
      in = (in + 1) % BUFFER_SIZE;
      mutex.acquire()
      counter++;
      mutex.release()
}

while (true) {
      while (counter == 0) {}/* do nothing
      item = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      mutex.acquire()
      counter--;
      mutex.release()
      consume_item(item);
      }
```

THE MONITOR SOLUTION

```
SharedBufferMonitor sbm = new SharedBufferMonitor();
```

Thread A

```
while (true) {
item = produce_item;
sbm.placeItem(item);
}
```

Thread B

```
while (true) {
item = sbm.extractItem(item)
consume_item(item);
}
```

```
in = out = 0;
Mutex mutex=Mutex.Init();
```

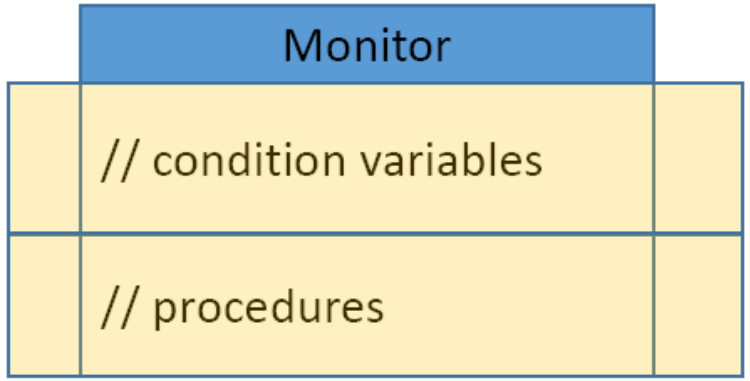
```
while (true) {
      item = produce_item;
      while
      (counter == BUFFER_SIZE) {}/* do nothing */;
      buffer[in] = item;
      in = (in + 1) % BUFFER_SIZE;
      mutex.acquire()
      counter++;
      mutex.release()
}
```

```
while (true) {
      while (counter == 0) {}/* do nothing
      item = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      mutex.acquire()
      counter--;
      mutex.release()
      consume_item(item);
      }
```

MONITORS

The object's state … **not** accessible from the "outside"

The "methods" that are accessible from the outside



MONITORS

- Provide a more object-oriented, abstract style approach to mutex and synchronization
- More structure than a semaphore

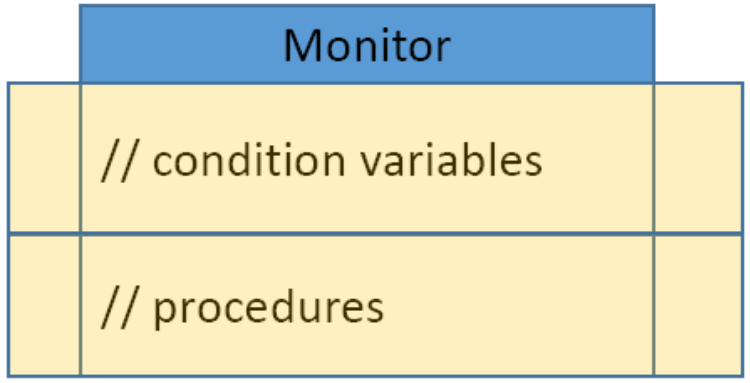
- A data abstraction mechanism
- Can be implemented multiple ways

Defining Properties

- Access to monitor variables is only through interface
- Mutual exclusion of all monitor procedures is implicit (procedures in the same monitor cannot be executed concurrently)
- Condition synchronization is via condition variables

The object's state … **not** accessible from the "outside"

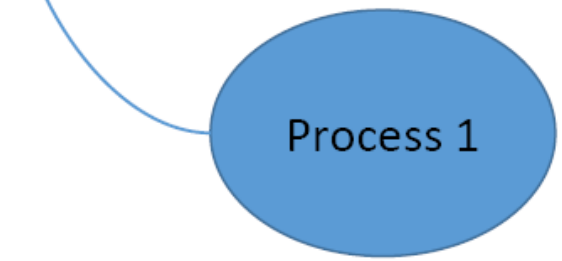
The "methods" that are accessible from the outside



MONITOR PROCEDURE

- A monitor procedure is called by an external process
- A procedure is active if some process is executing a statement in the monitor

```
monitor mName{

    //variables

    procedure1(){
      // statement
    }
    procedure2(){
      // statement
    }

}
```



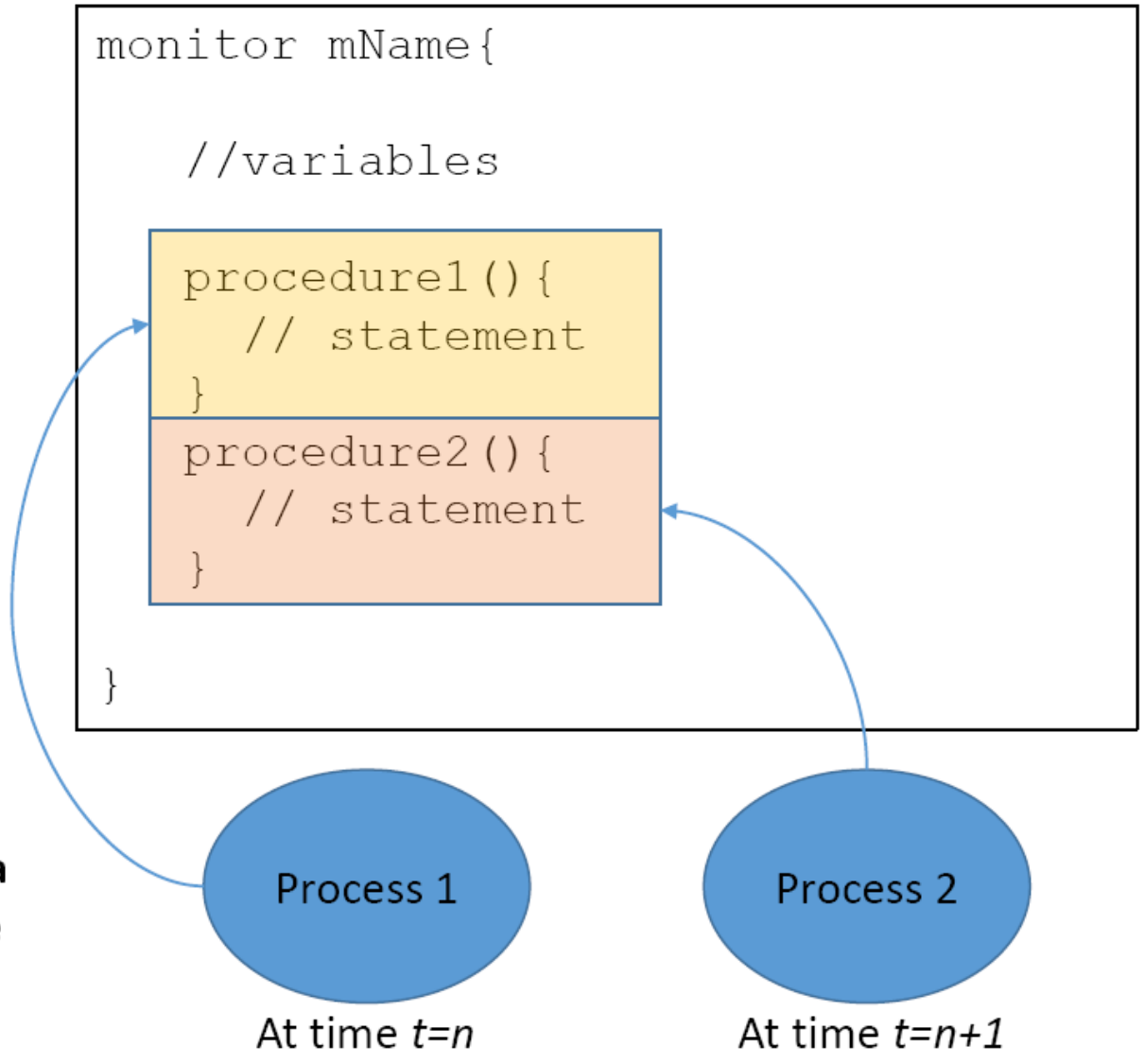
At time *t=n*

Assume Process 1 begins executing `procedure1`. No other processes are accessing the monitor `mName`.

MONITOR PROCEDURE

- A monitor procedure is called by an external process
- A procedure is active if some process is executing a statement in the monitor
- At most one instance of any one of a monitor's procedure may be active at the same time

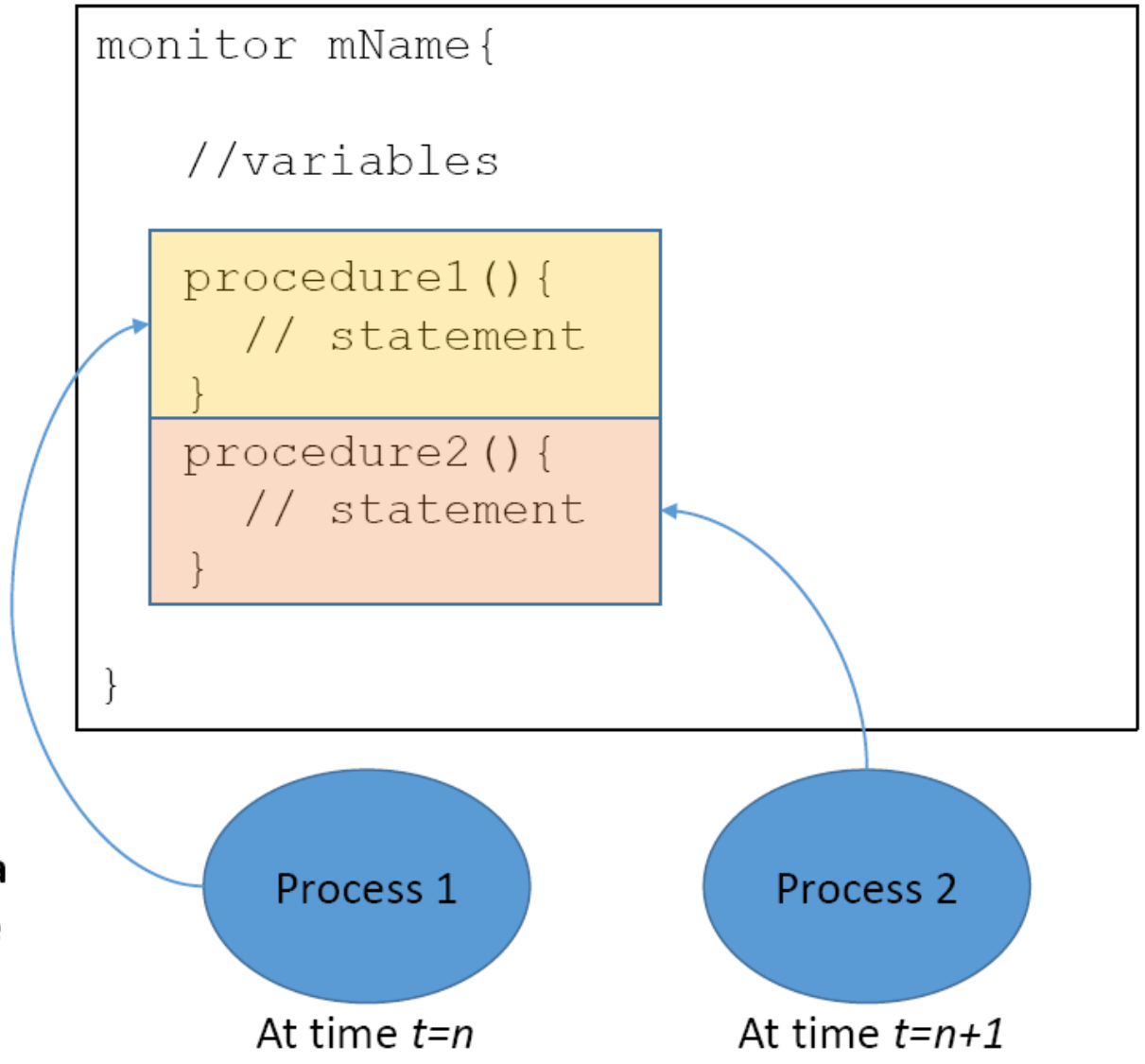
Q: Is this allowed? Two processes, each invoking a separate procedure in the same monitor?



MONITOR PROCEDURE

- A monitor procedure is called by an external process
- A procedure is active if some process is executing a statement in the monitor
- At most one instance of any one of a monitor's procedure may be active at the same time

Q: Is this allowed? Two processes, each invoking a separate procedure in the same monitor?



Only one will be running though.

The user doesn't know and doesn't care about any synchronization.

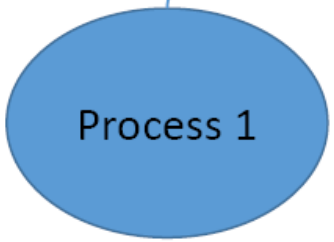
CONDITION VARIABLES

```
monitor mName{

   cond cv;

   procedure1(){
      // statement
   }
   procedure2(){
      // statement
   }

}
```

Q: Can an outside processes access the value of cv?

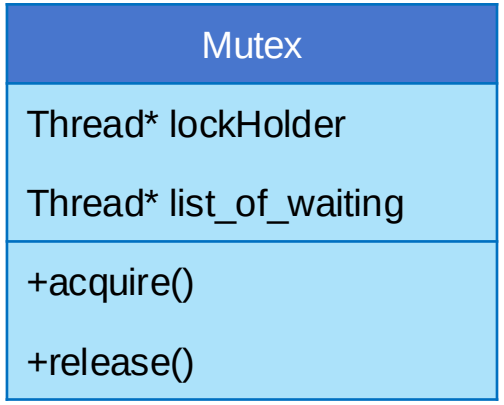
PROCEDURES TO READ CONDITION VARIABLES

- A process can indirectly query the state of a condition variable by calling publicly available methods, such as `empty`




```
monitor mName{

    cond cv;

    empty(cv){
      // return true if empty
    }
    wait(cv){
      // a process blocks; its ID is
      // placed at the rear of the
      // cv queue
    }
}
```

MUTEX LOCKS, SEMAPHORES AND MONITORS

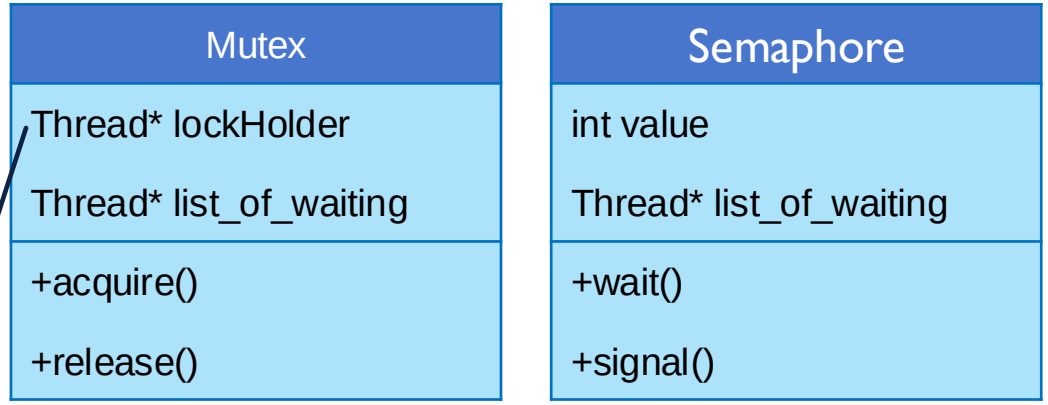


MUTEX LOCKS, SEMAPHORES AND MONITORS

Mutex
Thread* lockHolder Thread* list_of_waiting
+acquire() +release()

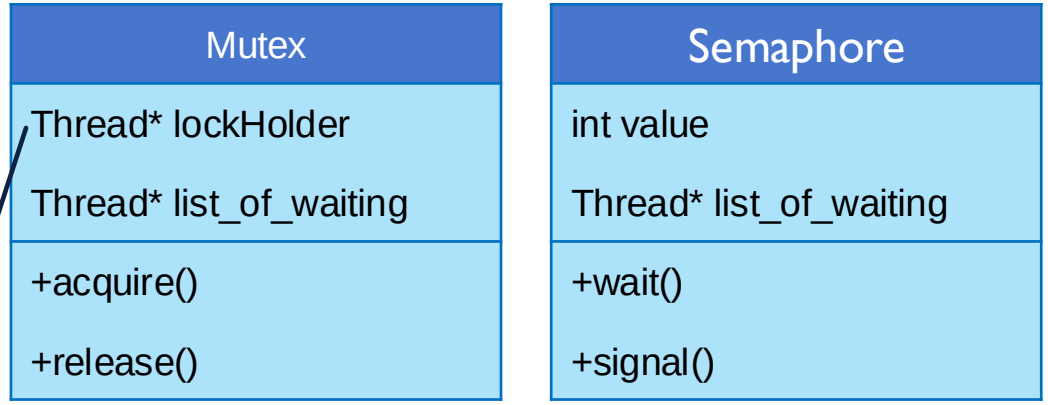
Semaphore
int value Thread* list_of_waiting
+wait() +signal()

MUTEX LOCKS, SEMAPHORES AND MONITORS



Notice there is a lock holder for mutex but not for Semaphore. Any guesses why?

MUTEX LOCKS, SEMAPHORES AND MONITORS



Notice there is a lock holder for mutex but not for Semaphore. Any guesses why?

For a mutex lock, only the lock holder can release the lock. For a Semaphore, threads using increment() could be different than those using decrement().

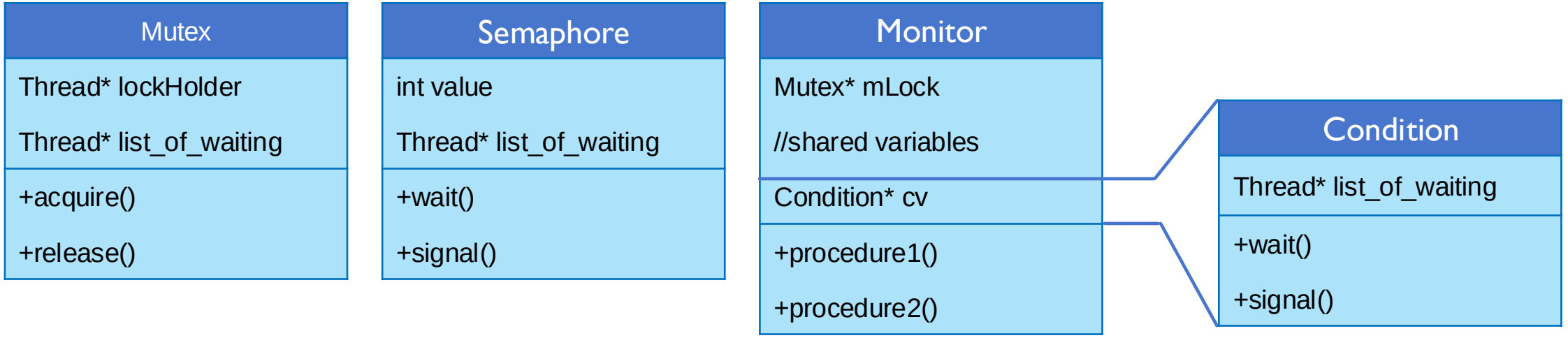
MUTEX LOCKS, SEMAPHORES AND MONITORS

Mutex
Thread* lockHolder Thread* list_of_waiting
+acquire() +release()

Semaphore
int value Thread* list_of_waiting
+wait() +signal()

Monitor
Mutex* mLock //shared variables Condition* cv
+procedure1() +procedure2()

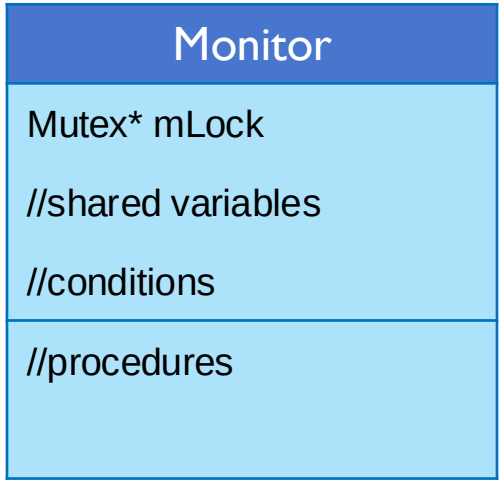
MUTEX LOCKS, SEMAPHORES AND MONITORS



PRODUCER/CONSUMER MONITOR

PRODUCER/CONSUMER MONITOR

- Shared Variables:




```
in = out = 0;
```

```
while (true) {
item = produce_item;
while
(counter == BUFFER_SIZE) {}/* do nothing */;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
counter++;
}
```

```
while (true) {
while (counter == 0) {}/* do nothing
item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
counter--;
consume_item(item);
```