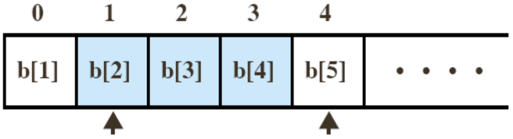
2.c Concurrency

Mutual exclusion & synchronization — mutexes

- Unbounded buffer, 1 producer, *N* consumers
  - ✓ **out** shared by all consumers → mutex among consumers
  - ✓ producer not concerned: can still add items to buffer at any time

```
item[] b;
int in, out;
```



```
mutex out_mutex;
```

```
void producer()
{
  while (true) {
    item = produce();

    b[in] = item;
    in++;
  }
}
```

but this implementation is flawed: all consumers pass the "while" at once, end up waiting at lock, then enter even if buffer is empty . . .

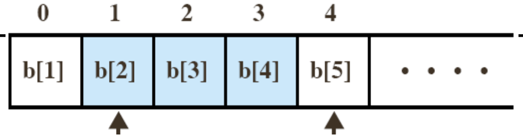
```
void consumer()
{
  while (true) {
    ~~while (out == in);~~
    ~~lock(out_mutex);~~
    item = b[out];
    out++;
    unlock(out_mutex);
    consume(item);
  }
}
```

2.c Concurrency

Mutual exclusion & synchronization — mutexes

- Unbounded buffer, 1 producer, *N* consumers
  - ✓ **out** shared by all consumers → mutex among consumers
  - ✓ producer not concerned: can still add items to buffer at any time

```
item[] b;
int in, out;
```



```
mutex out_mutex;
```

```
void producer()
{
  while (true) {
    item = produce();

    b[in] = item;
    in++;
  }
}
```

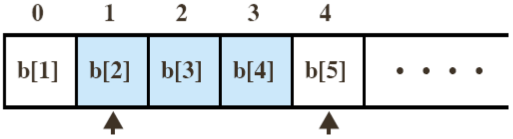
```
void consumer()
{
  while (true) {
    lock(out_mutex);
    while (out == in);
    item = b[out];
    out++;
    unlock(out_mutex);
    consume(item);
  }
}
```

this implementation is correct: even if a consumer loops inside, it means the buffer is empty anyway, so the others may as well be blocked outside . . .

2.c Concurrency

Mutual exclusion & synchronization — mutexes

- Unbounded buffer, *N* producers, *N* consumers
  - ✓ **in** shared by all producers → other mutex among producers
  - ✓ consumers and producers still (relatively) independent



```
item[] b;
int in, out;
```

```
mutex out_mutex;
mutex in_mutex;
```

```
void producer()
{
  while (true) {
    item = produce();
    lock(in_mutex);
    b[in] = item;
    in++;
    unlock(in_mutex);
  }
}
```

```
void consumer()
{
  while (true) {
    lock(out_mutex);
    while (out == in);
    item = b[out];
    out++;
    unlock(out_mutex);
    consume(item);
  }
}
```

still correct, but in all cases the consumers are busy waiting . . .

2.c Concurrency

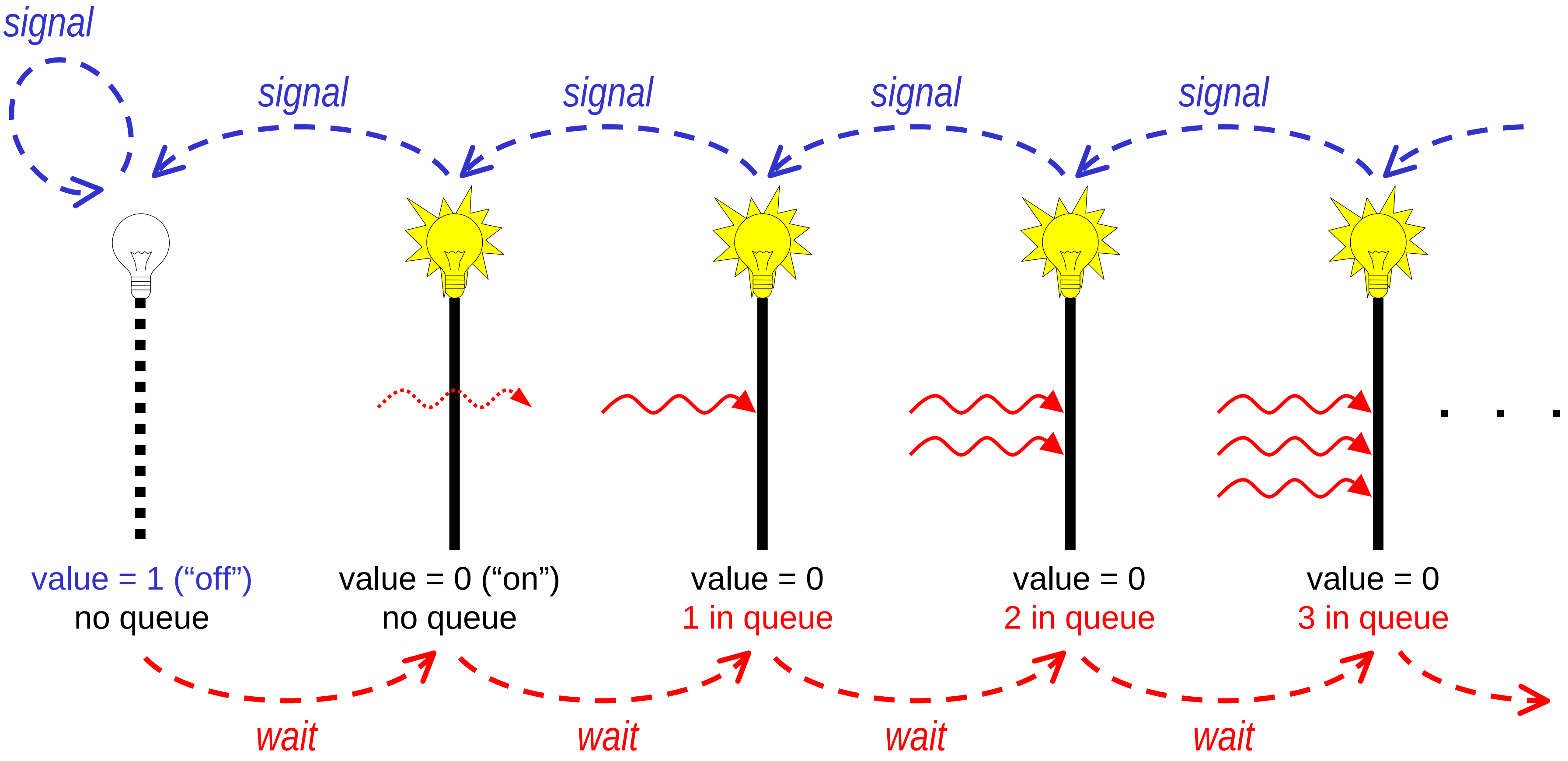
Mutual exclusion & synchronization — semaphores

- Synchronization
  - ✓ processes can also **cooperate** by means of simple signals, without defining a "critical region"
  - ✓ like mutexes: instead of looping, a process can block in some place until it receives a specific **signal** from the other process
- Binary semaphore ⇔ mutex
  - ✓ a binary semaphore is a variable that has a value 0 or 1
  - ✓ a **wait** operation attempts to <u>decrement</u> the semaphore
    - ▪ 1 → 0 and goes through; 0 → blocks
  - ✓ a **signal** operation attempts to <u>increment</u> the semaphore
    - ▪ 1 → 1, no change; 0 → unblocks or becomes 1

2.c Concurrency

Mutual exclusion & synchronization — semaphores

- Binary semaphore ⇔ mutex

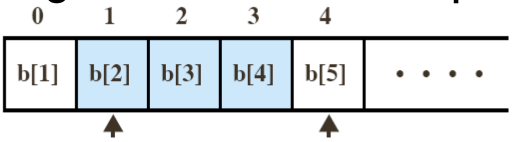


2.c Concurrency

Mutual exclusion & synchronization — semaphores

- Unbounded buffer, 1 producer, 1 consumer with sync
  - ✓ if buffer is empty, the consumer waits on a semaphore
  - ✓ if buffer just got one item, the producer signals to the consumer

```
item[] b;
int in, out;
```



```
bin_semaphore Bsem = 0;
```

```
void producer()
{
  while (true) {
    item = produce();

    b[in] = item;
    in++;
    if (out == in-1)
      signal(Bsem);
  }
}
```

unfortunately, this can lead to an inconsistent semaphore state . . .

```
void consumer()
{
  while (true) {
    ~~while (out == in);~~
    if (out == in)
      wait(Bsem);
    item = b[out];
    out++;

    consume(item);
  }
}
```

2.c Concurrency

Mutual exclusion & synchronization — semaphores

1.

```
x = produce();
b[in++] = x;
```

2.

```
x = b[out++];
consume(x);
```

3.

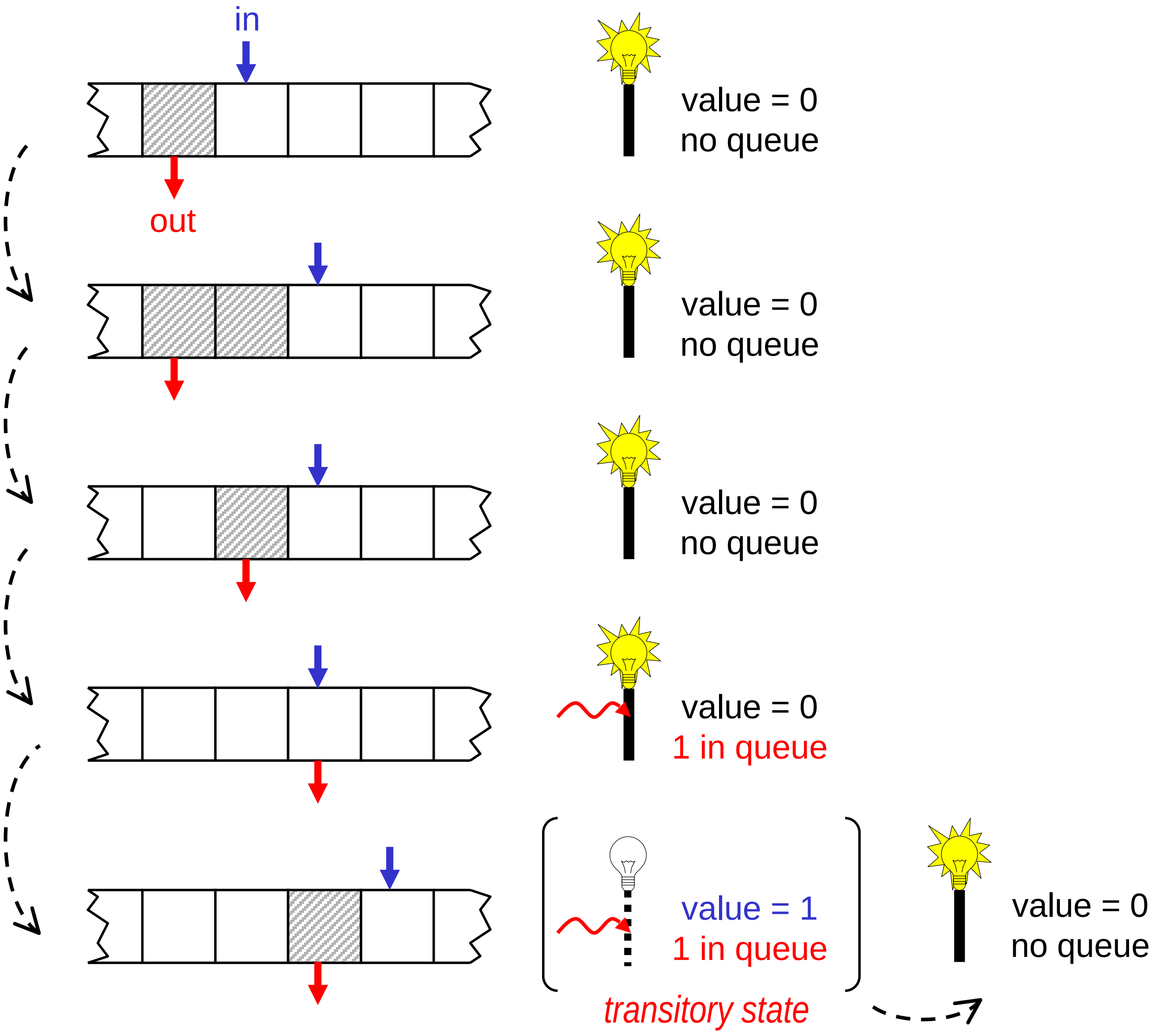
empty →

```
x = b[out++];
consume(x);
wait(sem);
```

4.

one item →

```
x = produce();
b[in++] = x;
signal(sem);
```



2.c Concurrency

Mutual exclusion & synchronization — semaphores

3.’

```
x = b[out++];
consume(x);
```

4.’

```
x = produce();
b[in++] = x;
signal(sem);
```

NOT empty →

5.’

~~wait(sem);~~

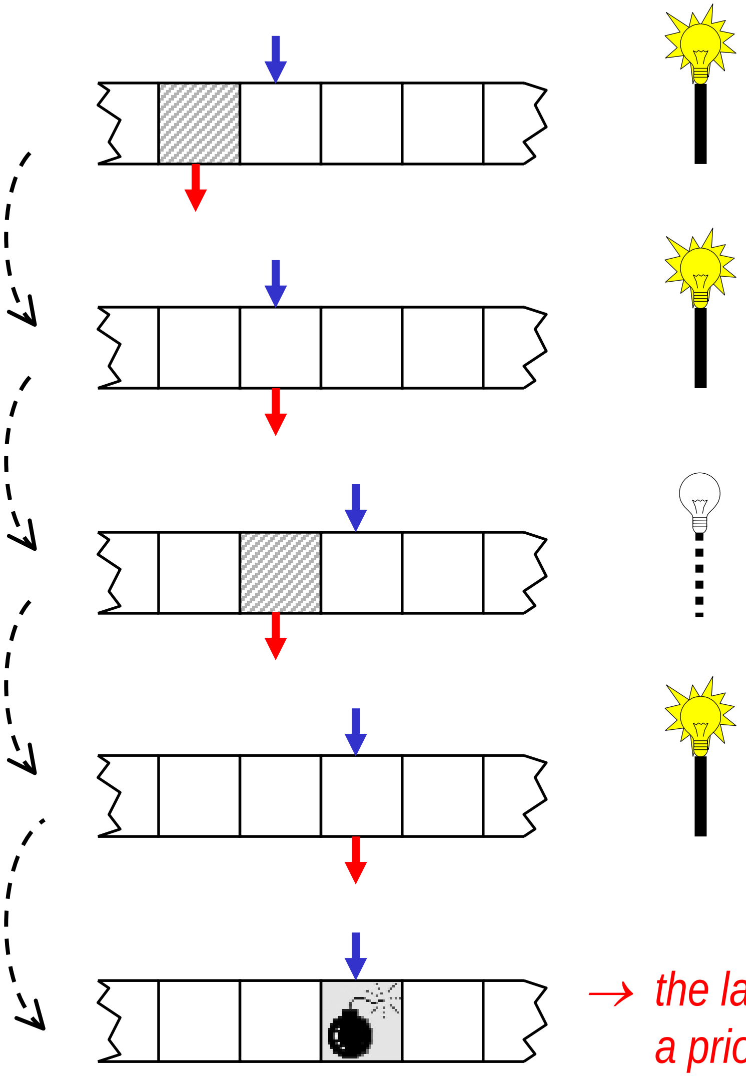
```
x = b[out++];
consume(x);
wait(sem); →
```

now empty →

. . . but goes through

6.’

```
x = b[out++];
consume(x);
```



value = 0
no queue

value = 0
no queue

value = 1
no queue

value = 0
no queue

??

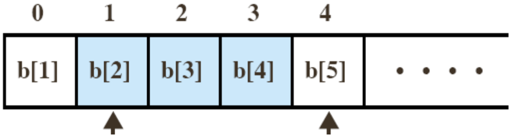
→ the last “signal” was not matched by a prior “wait”: they missed each other

2.c Concurrency

Mutual exclusion & synchronization — semaphores

- Unbounded buffer, 1 producer, 1 consumer with sync
  - ✓ we need to create critical areas to keep "consuming" and "checking the semaphore" together

```
item[] b;
int in, out;
```



```
bin_semaphore Bsem = 0;
```

```
void producer()
{
  while (true) {
    item = produce();
    lock(buf_mutex);
    b[in] = item;
    in++;
    if (out == in-1)
      signal(Bsem);
    unlock(buf_mutex);
  }
}
```

```
void consumer()
{
  while (true) {
    ~~lock(buf_mutex);~~
    ~~if (out == in)~~
      ~~wait(Bsem);~~
    item = b[out];
    out++;
    unlock(buf_mutex);
    consume(item);
  }
```

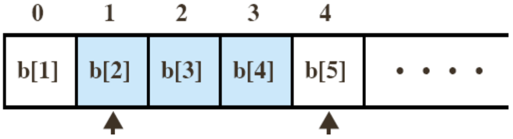
but there is a deadlock: here the consumer is blocking the producer, not other consumers . . .

2.c Concurrency

Mutual exclusion & synchronization — semaphores

- Unbounded buffer, 1 producer, 1 consumer with sync
  - ✓ the consumer needs to remember the current state of `in` & `out`, so it can exit the CR before checking the semaphore

```
item[] b;
int in, out;
```



```
bin_semaphore Bsem = 0;
```

```
void producer()
{
  while (true) {
    item = produce();
    lock(buf_mutex);
    b[in] = item;
    in++;
    if (out == in-1)
      signal(Bsem);
    unlock(buf_mutex);
  }
}
```

finally correct!

```
void consumer()
{
  while (true) {
    if (out == in0)
      wait(Bsem);
    lock(buf_mutex);
    item = b[out];
    out++; in0 = in;
    unlock(buf_mutex);
    consume(item);
  }
}
```

2.c Concurrency

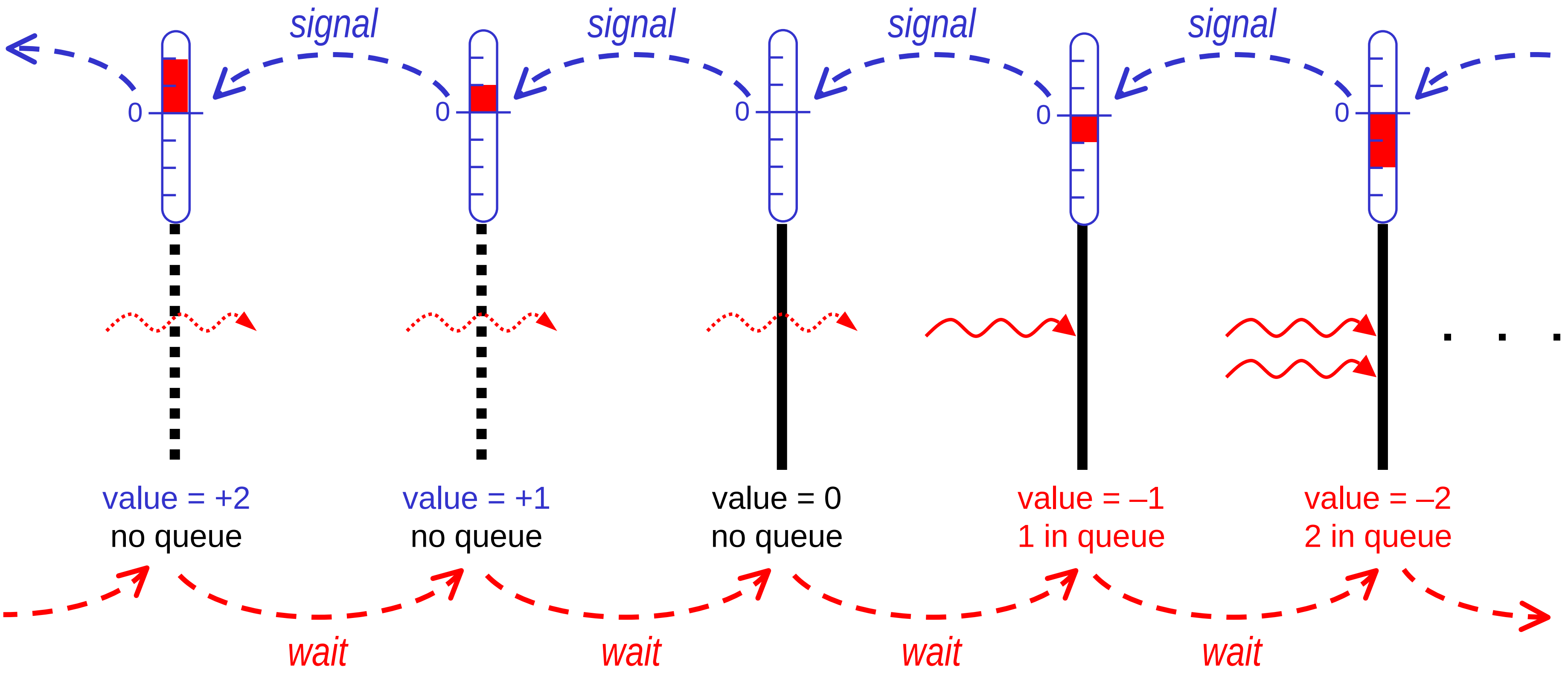
Mutual exclusion & synchronization — semaphores

- Semaphores are used for signaling between processes
  - ✓ semaphores can be used for **mutual exclusion**
  - ✓ binary semaphores are the same as mutexes
  - ✓ integer semaphores can be used to allow more than one process inside a critical region; generally:
    - ▪ the positive value of an integer semaphore corresponds to a maximum number of processes allowed concurrently inside a critical region
    - ▪ the negative value of an integer semaphore corresponds to the number of processes currently waiting in the queue
  - ✓ binary and integer semaphores can also be used for **synchronization**

2.c Concurrency

Mutual exclusion & synchronization — semaphores

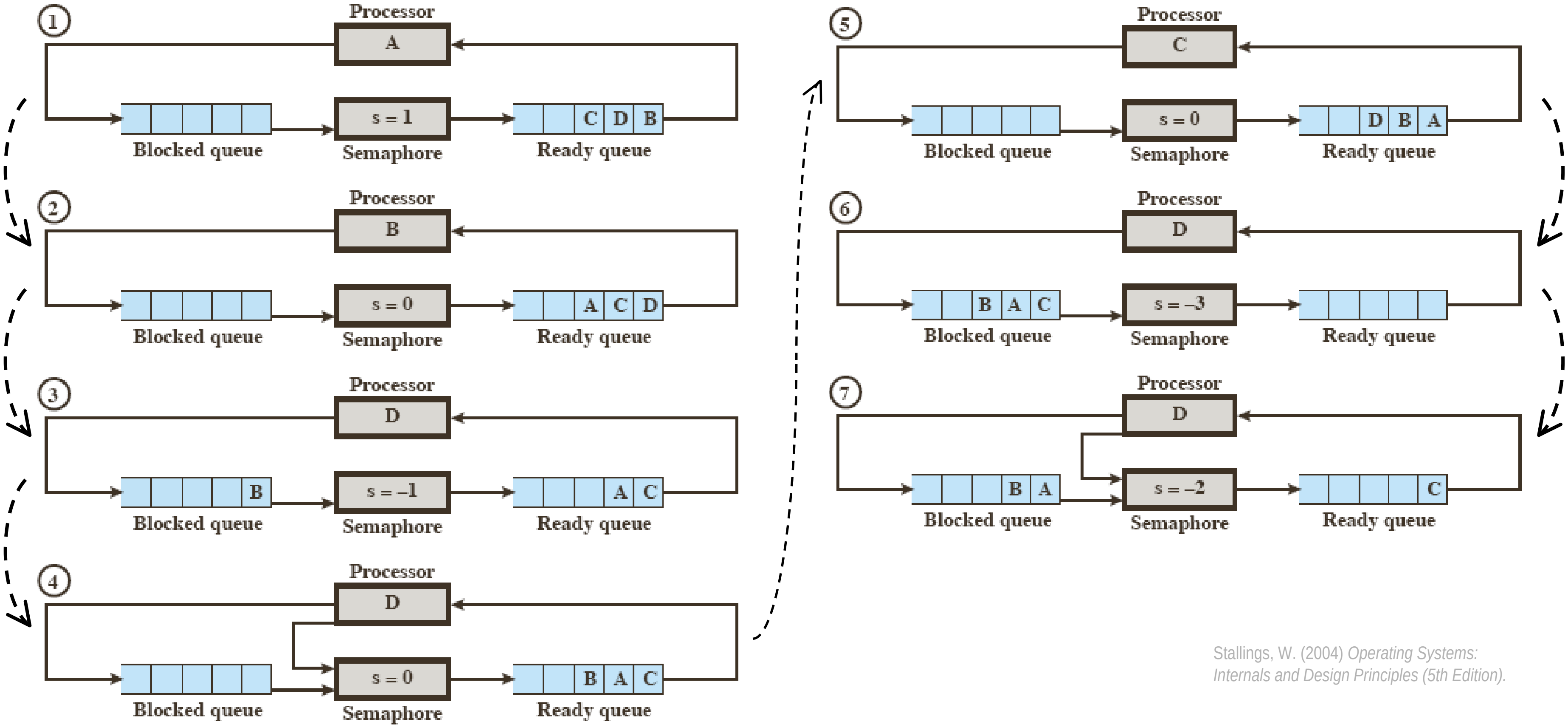
- Integer semaphore ⇔ "thermometer"



2.c Concurrency

Mutual exclusion & synchronization — semaphores

- All semaphores maintain a queue of waiting processes



Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

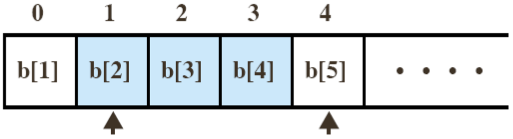
Example of semaphore mechanism

2.c Concurrency

Mutual exclusion & synchronization — semaphores

- Producer/consumer with an integer semaphore
  - ✓ no need for a condition: the semaphore itself keeps track of the size of the buffer

```
item[] b;
int in, out;
```



```
semaphore sem = 0;
```

```
void producer()
{
  while (true) {
    item = produce();
    lock(buf_mutex);
    b[in] = item;
    in++;
    ~~if (out == in-1)~~
      signal(sem);
    unlock(buf_mutex);
  }
}
```

correct!

```
void consumer()
{
  while (true) {
    ~~if (out == in0)~~
      wait(sem);
    lock(buf_mutex);
    item = b[out];
    out++;
    unlock(buf_mutex);
    consume(item);
  }
}
```

2.c Concurrency

Mutual exclusion & synchronization — semaphores

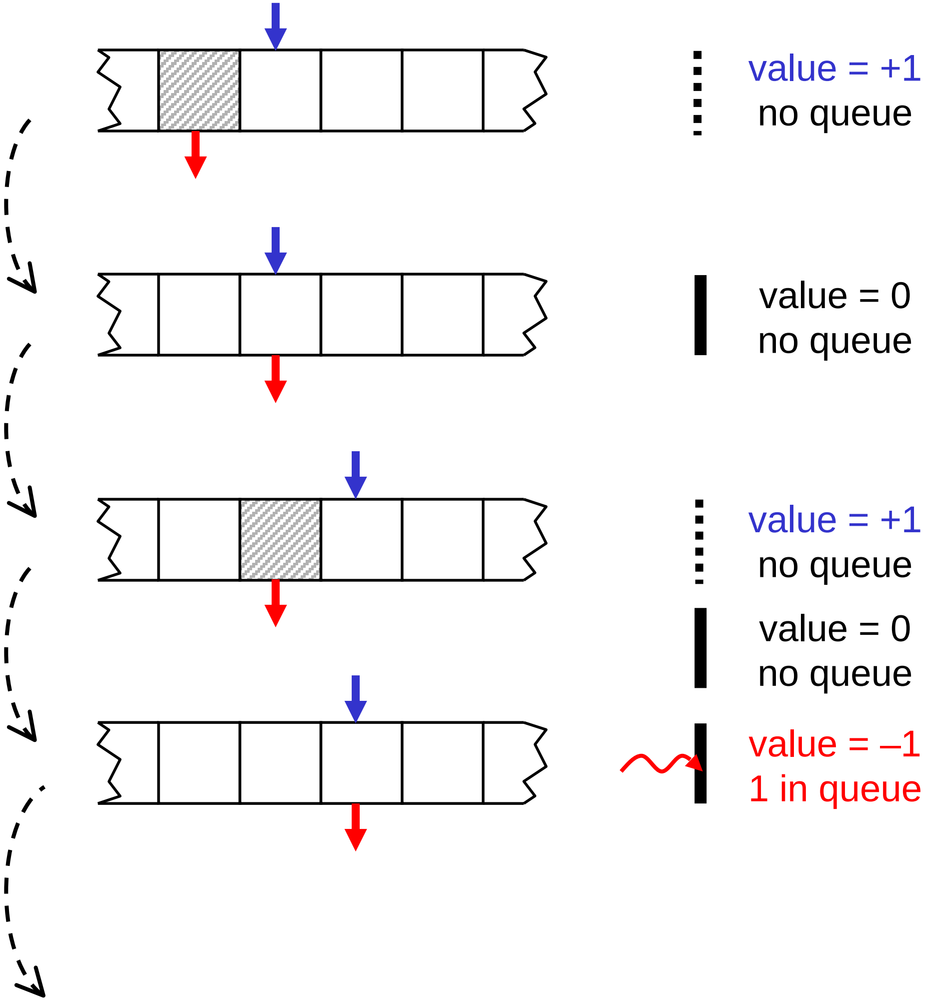
3.'
```
wait(sem);
x = b[out++];
consume(x);
```

4.'
```
x = produce();
b[in++] = x;
signal(sem);
```

5.'
```
wait(sem);
x = b[out++];
consume(x);
wait(sem);
```

6.'

~~x = b[out++];~~
~~consume(x);~~



value = +1
no queue

value = 0
no queue

value = +1
no queue

value = 0
no queue

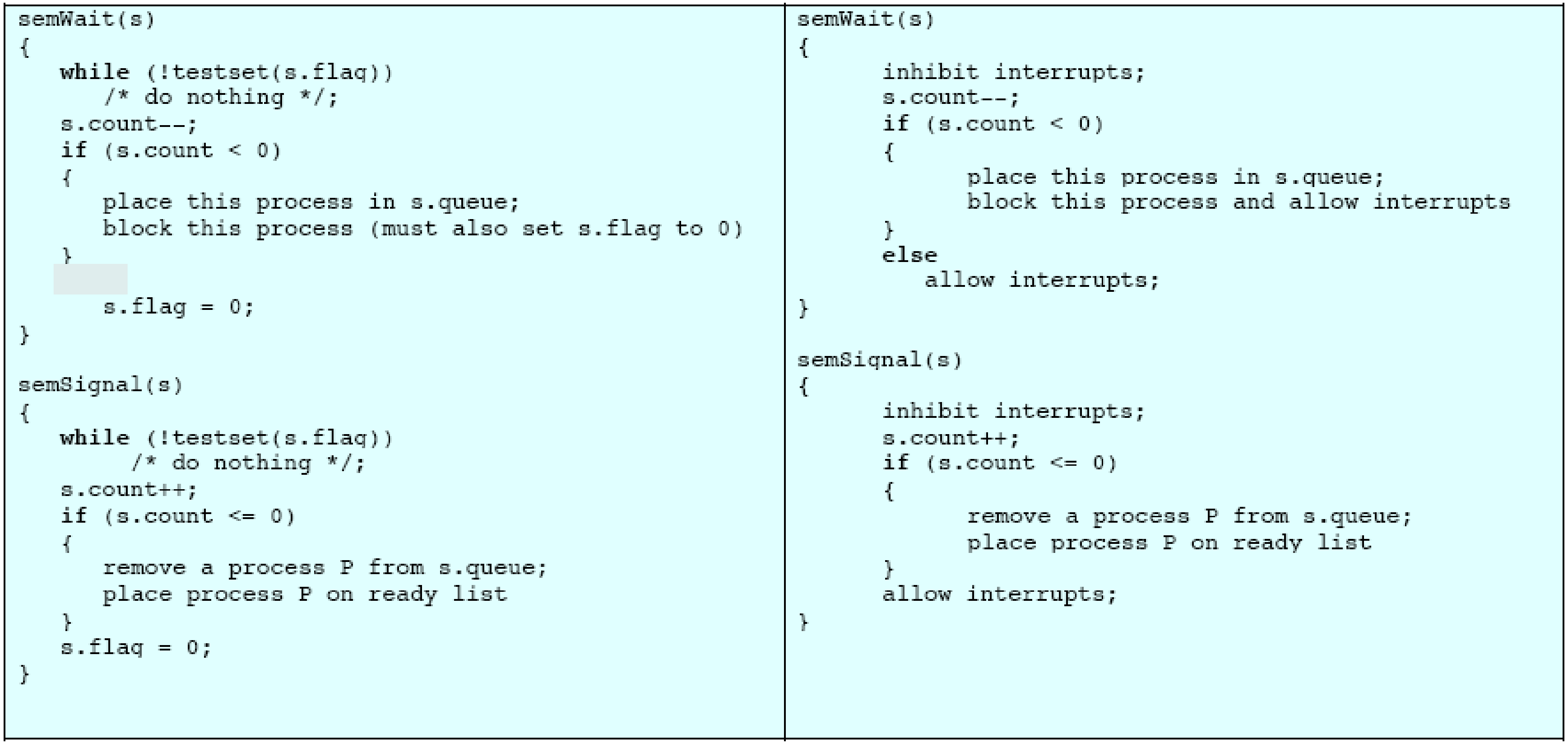
value = –1
1 in queue

the consumer is blocked, as it should be; the producer may proceed . . .

2.c Concurrency

Mutual exclusion & synchronization — semaphores

- How semaphores may be implemented



```
semWait(s)
{
   while (!testset(s.flag))
      /* do nothing */;
   s.count--;
   if (s.count < 0)
   {
      place this process in s.queue;
      block this process (must also set s.flag to 0)
   }

      s.flag = 0;
}

semSignal(s)
{
   while (!testset(s.flag))
        /* do nothing */;
   s.count++;
   if (s.count <= 0)
   {
      remove a process P from s.queue;
      place process P on ready list
   }
   s.flag = 0;
}
```

(a) Testset Instruction

```
semWait(s)
{
      inhibit interrupts;
      s.count--;
      if (s.count < 0)
      {
            place this process in s.queue;
            block this process and allow interrupts
      }
      else
         allow interrupts;
}

semSignal(s)
{
      inhibit interrupts;
      s.count++;
      if (s.count <= 0)
      {
            remove a process P from s.queue;
            place process P on ready list
      }
      allow interrupts;
}
```

(b) Interrupts

Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

Two possible implementations of semaphores