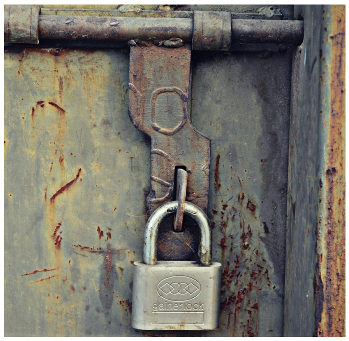
Software Solution: Semaphores
("lock & counter")



Semaphore in User Space

Semaphores: Edsger Dijkstra (1965)

Also invented
Dijkstra Graph Shortest Path Algorithm

```
void wait(int s) {
  while (s <= 0) { /* None */ }
  s--;
}
```

```
void signal(int s) {
  s++;
}
```

two ATOMIC operations on INTEGER s

Using Dijkstra Semaphore

```
// initial value
int s = 1;
```

```
while (s <= 0) { /* do nothing */ }
s--;
```

Critical Section

```
s++;
```

```
down(s);   // or wait(s)
```

Critical Section

```
up(s);     // or signal(s)
```

Types of Semaphore

- Binary Semaphores
  - Use: mutex locks, wait(L) to obtain the lock, signal(L) to release the lock
- Counting Semaphores: value can be any number (including **NEGATIVE**)
  - Common use: control access to resources with finite number of availability
  - Initialized to number of available resources
  - wait(R): request ONE unit of resource, signal(R): release the resource
- “Event Notification” Semaphores
  - Initialized to ZERO
  - wait(E): block until event took place, signal(E): notify that event has taken place

Binary Semaphore Details

```
while (s <= 0) {/* none */}
s--;
```

/* Critical Section */

```
s++;
```

s = ___ (initial value 1)

Counting Semaphore Details

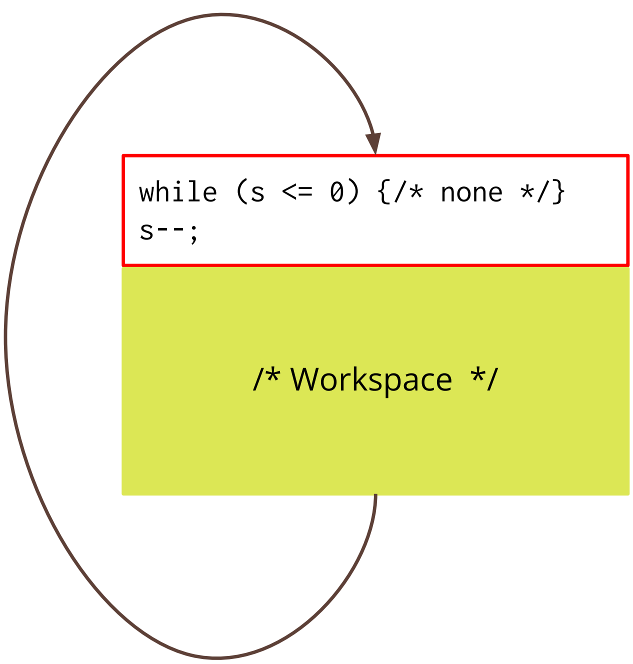
```
while (s <= 0) {/* none */}
s--;
```

/* Shared Space */

```
s++;
```

s = ____ (initial value 3)

Counting Semaphore Details



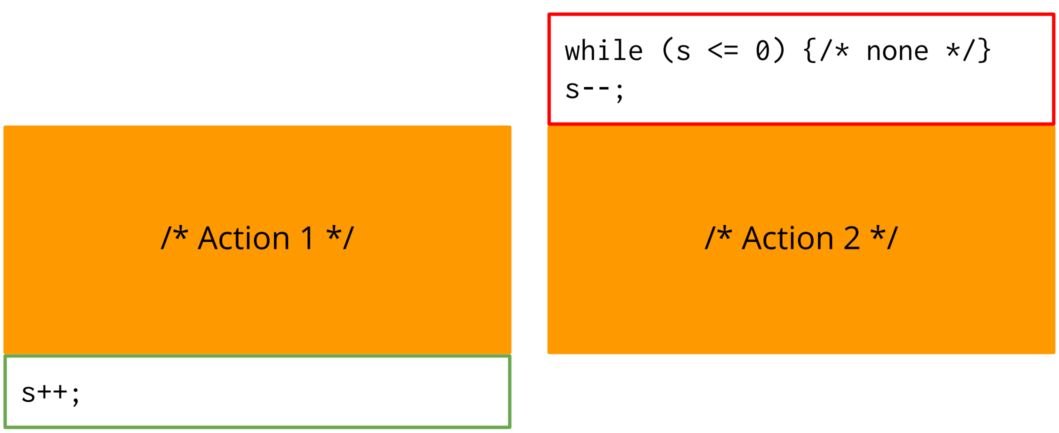
s = ____ (initial value 3)

Typical Use of Semaphores

- # of `sem.wait()` calls must == # of `sem.signal()` calls
- Protect Shared Resources (control the "**room capacity"**)
    - Invoke `sr.wait()` and `sr.signal()` pair **within one process** (the wait-signal pair creates a *virtual room* of capacity N)
    - Initialize the semaphore sr to the "room" capacity
- Event Counters (notify "**events**")
    - `ev.wait()` and `ev.signal()` calls are **split across two processes**, the pair create a *notification channel* between the two processes
    - Typically initialize the semaphore ev to zero (to indicate no *events have taken place*), or positive number (to indicate *some events have happened*)

Event Notification Semaphore Details

s = ___ (initial value 0)



Counting/Binary Semaphore

initialization

```
// counting
bigroom_sem = 5
```

```
wait(bigroom_sem)
```

Room of capacity 5

```
signal(bigroom_sem)
```

initialization

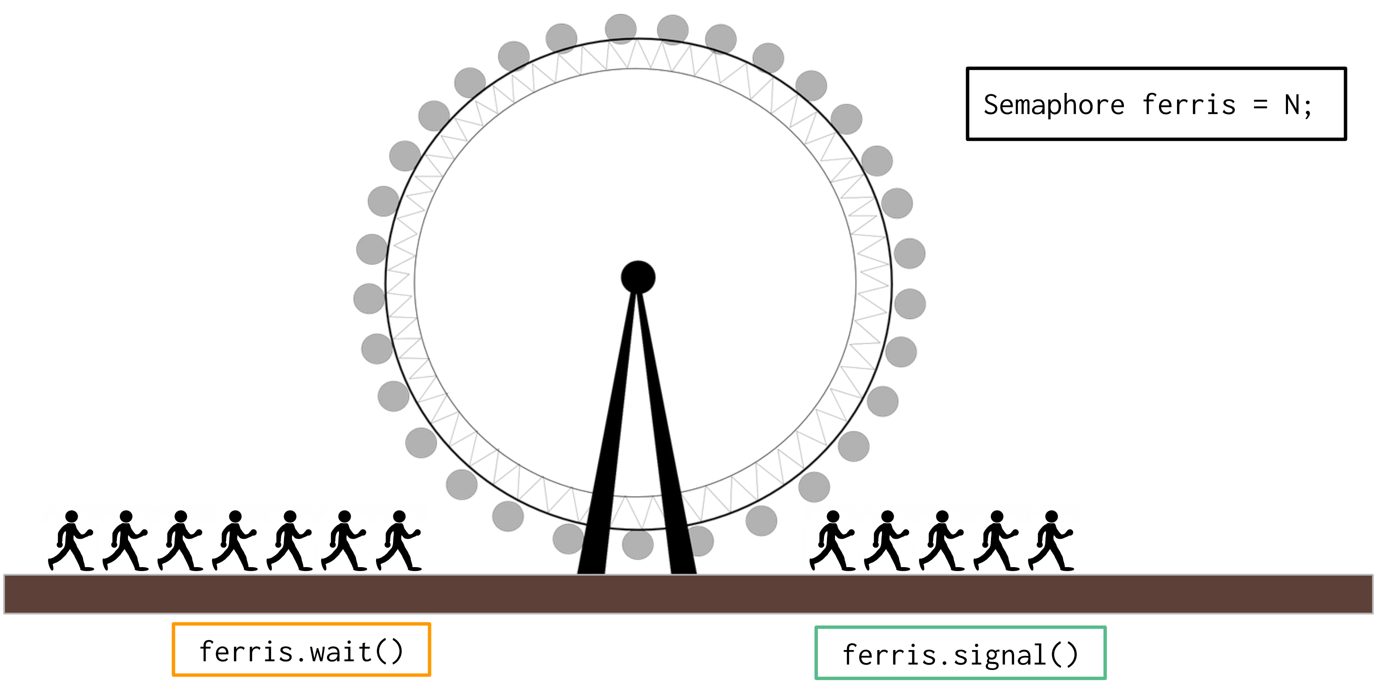
```
// binary
smallroom_sem = 1
```

```
wait(smallroom_sem)
```

Room of capacity 1

```
signal(smallroom_sem)
```


Ferris Wheel: “Critical Section” of Capacity N



The Rapids Bus: “Critical Section” of Capacity N


```
bus.wait()
// hop on
```

sit or sleep inside bus

```
// hop off
bus.signal()
```


number of wait() = number of signal()

Semaphore Implementation

Classical (With Busy Wait)	Modern (*Minimum* Busy Wait)
Without OS assistance	With OS assistance
*the counter becomes a lock when its value is **zero or negative***	*the counter becomes a lock when its value is **negative***

Implementation: classical vs. modern

```
wait(s) {
   spin_lock;
   while (s <= 0)
     ; // BUSY WAIT

   S--;
   spin_unlock;
}
```

```
signal(s) {
   spin_lock;
   s++;
   spin_unlock;
}
```

Using **classical** definitions, semaphore values **are never negative**

```
class Semaphore { // "MODERN" semaphore,
private:          // spin_lock is used, but not explicitly shown
  int value;
  queue<Process> list; // queue of processe                    e
public:
  void wait() {   // ATTN: no while loop!!!
    value--;
    if (value < 0) {
      list.push_back(_this_process_);
      change the state of _this_process_ to "blocked"
    }
  }

  void signal() {
    value++;
    if (value <= 0) {
      Process p = list.pop_front();
      change the state of p to "ready"
    }
  }
}
```

value = 1 (init 1)

*Using "modern" definition, semaphore value **may be negative** and its magnitude is the number of processes blocked on the semaphore*

Implementation: classical vs. modern

```
Semaphore x;

x.wait();

// Shared Section

x.signal();
```

```
class Semaphore { // "MODERN" semaphore,
                  // spin_lock is used, but not explicitly shown
private:
  int value;
  queue<Process> blocked; // queue of processes blocked on this semaphore
public:
  void wait() {   // ATTN: no while loop!!!
    value--;
    if (value < 0) {
      blocked.push_back(_this_process_);
      change the state of _this_process_ to "blocked"
    }
  }

  void signal() {
    value++;
    if (value <= 0) {
      Process p = blocked.pop_front();
      change the state of p to "ready"
    }
  }
}
```

value = ____ (init __)

*Using "modern" definition, semaphore value **may be negative** and its magnitude is the number of processes blocked on the semaphore*

"Modern" Semaphore Implementation

- **value** and **list** are *shared variables* themselves
- Operations inside `Semaphore::wait()` and `Semaphore::signal()` must be ATOMIC
  - Increment / decrement s
  - Add / remove processes/threads from the queue
- Use spinlock to guarantee atomic operation **throughout both functions**
  - **We can't avoid busy wait altogether**!
  - Classical semaphores require **much longer busy wait**
  - Modern semaphores run the spinlock only for a **fraction of time**

Integer Value of Modern Semaphores

`Semaphore sem;`

- sem.value ≥ 0: the number of processes that can run `sem.wait()` without getting blocked
- sem.value < 0: **abs(sem.value)** is the number of processes blocked on the semaphore

"Notification" Semaphores: Chef to Server

initialization
taco_done = 0

`signal(taco_done)`

`wait(taco_done)`

"Notification" Semaphores: Server to Chef

`wait(empty_tray)`

initialization
empty_tray = 5;

Current value
empty_tray is 2

`signal(empty_tray)`

"Notification" Semaphores

`wait(empty_tray)`

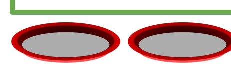
`signal(taco_done)`

initialization
taco_done = 0 empty_tray = 5;

Current value
taco_done is 3 empty_tray is 2

`wait(taco_done)`

`signal(empty_tray)`



Python Playground

Taco Restaurant using Semaphores

```
/* Chef */
while (true) {

  t = make_taco();

  empty_tray.wait();
  put_taco_on_tray(t)
  mutex.wait();
  put_tray_on_kitchen_counter()
  mutex.signal();

  taco_done.signal();


}
```

```
/* Server */
while (true) {



  taco_done.wait();
  mutex.wait();
  get_tray_from_kitchen_counter()
  mutex.signal();
  serve_taco_to_customer();
  put_empty_tray_on_kit_counter();
  empty_tray.signal();

}
```

Semaphore values:

```
empty_tray = __ (init 5)
 taco_done = __ (init 0)
     mutex = __ (init 1)
```


Producer/Consumer Solution using Semaphores

```
/* producer (append @ aend) */
while (true) {

  p_item = produce_item();

  empty_bin.wait();
  mutex.wait();
    buff[in] = p_item;
    in++;
    in %= BUFF_SIZE;
  mutex.signal();
  filled_bin.signal();


}
```

```
/* consumer (remove from front) */
while (true) {



  filled_bin.wait();
  mutex.wait();
    c_item = buff[out];
    out++;
    out %= BUFF_SIZE;
  mutex.signal();
  empty_bin.signal();

  consume_item (c_item);
}
```

Semaphore values:

```
 empty_bin = __ (init 3)
filled_bin = __ (init 0)
     mutex = __ (init 1)
```

Producer/Consumer using Semaphores

- Shared buffer with N bins
- Two "event counters"
  - an item is placed in a bin (bin_filled similar to "taco_done")
  - an item is removed from a bin (bin_emptied similar to "empty_tray")
- One mutex lock (binary semaphore)
  - shared buffer manipulated concurrently by both producer and consumer

Producer: counters & busy wait ⇔ semaphores

```
/* producer (append @ the end */
while (true) {

  p_item = produce_item();

  while (counter == BUFF_SIZE)
    /* do nothing */;
  buff[in] = p_item;
  in++;
  in %= BUFF_SIZE;

  counter++; /* unblock consumer *//

}
```

→

```
/* producer (append @ the end) */
while (true) {

  p_item = produce_item();

  empty_bin.wait();

  buff[in] = p_item;
  in++;
  in %= BUFF_SIZE;

  filled_bin.signal();

}
```

Consumer: counters & busy wait ⇔ semaphores

```
/* consumer (remove from front) */
while (true) {



  while (counter == 0)
    /* do nothing */;
  c_item = buff[out];
  out++;
  out %= BUFF_SIZE;

  counter--; /* unblock producer */

  consume_item (c_item);

}
```

```
/* consumer (remove from front) */
while (true) {



  filled_bin.wait();

  c_item = buff[out];
  out++;
  out %= BUFF_SIZE;

  empty_bin.signal();

  consume_item (c_item);

}
```

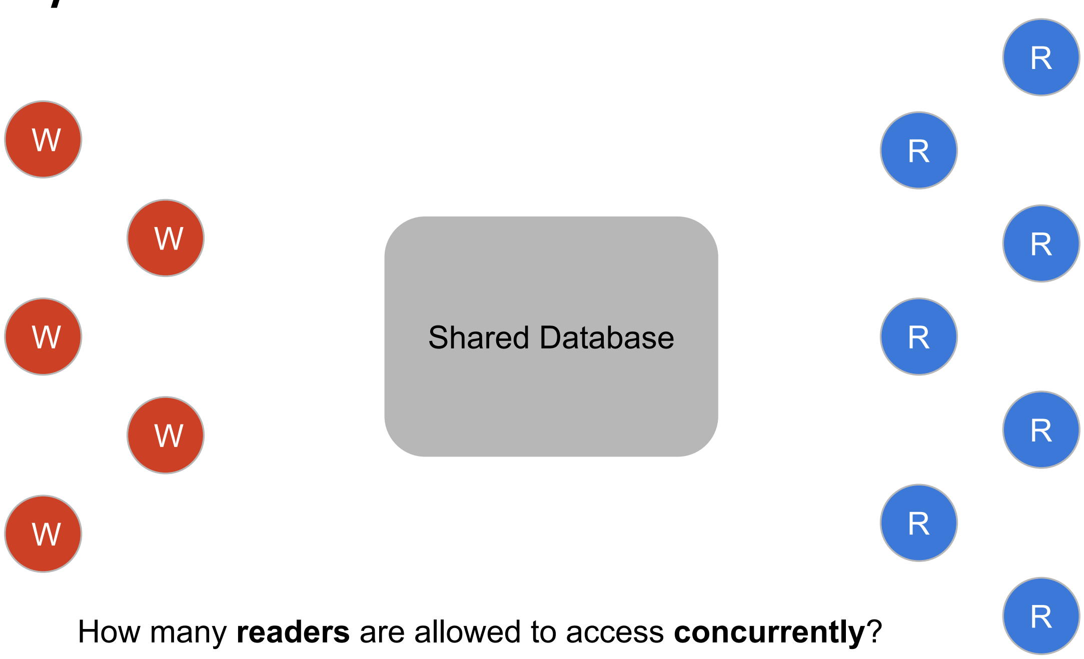

Semaphore wait() vs Process wait()

- Semaphore wait() **becomes blocking** *only when* the value of the semaphore is
  - Non-positive (classic semaphores)
  - Negative (modern semaphores)

- Process wait() **becomes blocking** when its child process has not terminated
- Process wait() does not block when the child it is waiting for has terminated

Semaphores for Classic CS Problems

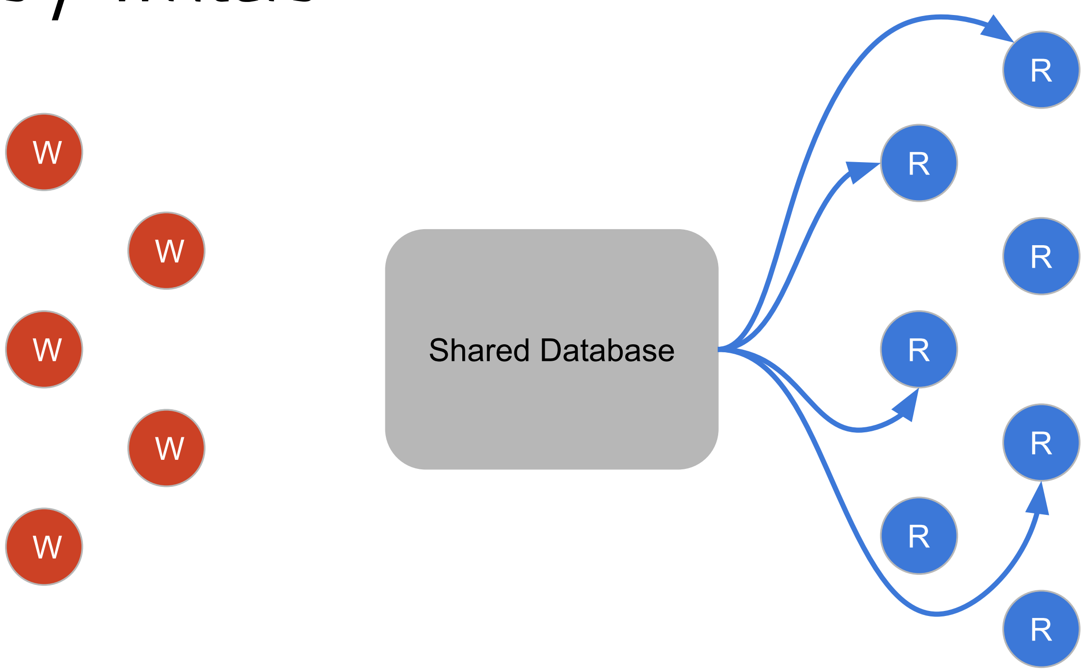
Readers / Writers



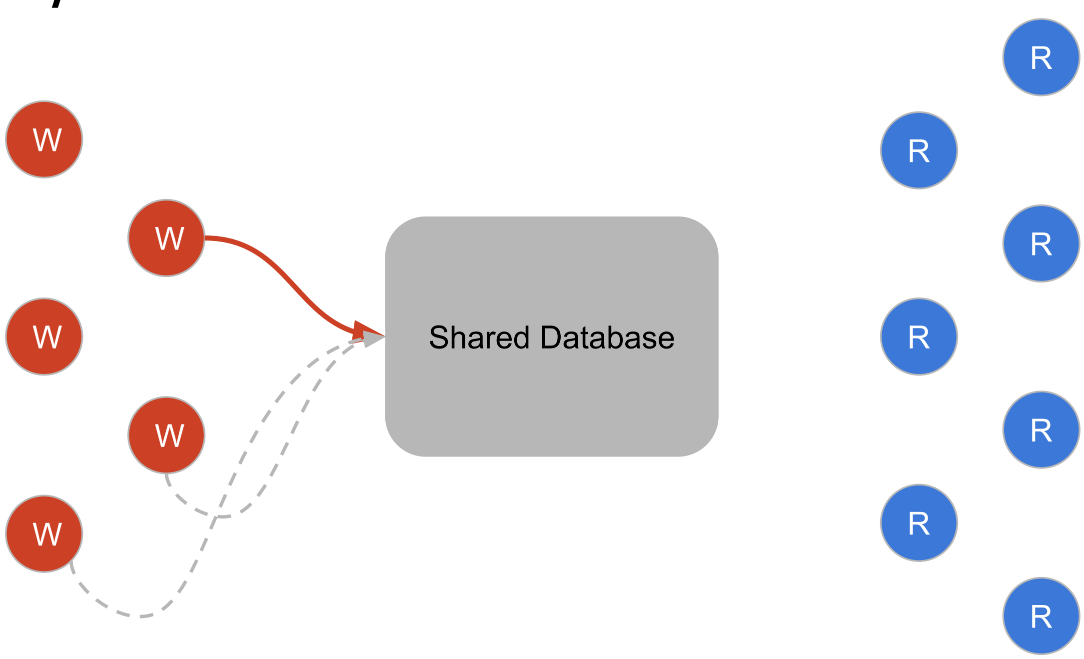
How many **readers** are allowed to access **concurrently**?

How many **writers** are allowed to access **concurrently**?

Readers / Writers



Readers / Writers



Readers/Writers

- More challenging than Producer/Consumer problem
  - ONE producer and ONE consumer
  - MANY readers and MANY writers
- **Asymmetrical access**
  - Only one writer is allowed at any time (destructive operation)
  - Multiple readers are allowed at any time (non-destructive operation)
- Reading and writing are mutually exclusive operations
  - When the DB is being written, no readers shall be allowed access
  - When the DB is being read (by multipler readers), no writers shall be allowed access

Semaphores for Reader/Writers

- An **“active”** writer must exclude other writers and other readers
  - Simpler synchronization code for writers
- An “active” reader should exclude any writers but allow other readers to join reading the DB
  - More complicated logic in readers’ code
- Solution Strategy
  - Assign one reader to be the “group leader”
  - Let the “group leader” prevent other writers from using the DB but allow other readers

Readers/Writers Handout

Readers/Writers (First attempt)

```
/* writer */
while (true) {

  rw_access.wait();

  // update the DB
  // Update the DB
  // Update the DB

  rw_access.signal();

}
```

```
/* reader */
while (true) {

  rw_access.wait();

  // read the DB
  // read the DB
  // read the DB

  rw_access.signal();

}
```

rw_access: __ (init 1)

#1: Readers/Writers (incomplete)

```
/* writer */
while (true) {

  rw_access.wait();


  // update the DB
  // Update the DB
  // Update the DB

  rw_access.signal();


}
```

```
/* reader */
while (true) {

  reader_counter++;
  if (reader_counter == 1)
    rw_access.wait(); // only the leader holds the access key

  // read the DB
  // read the DB
  // read the DB

  reader_counter--;
  if (reader_counter == 0)
    rw_access.signal(); // the leader releases the access key

}
```

rw_access:	__	(init 1)
reader_count:	__	(init 0)

#2: Readers/Writers: First R locks (last R unlock)

```
Semaphore rw_access:  __    (init 1)
Semaphore mutex:      __    (init 1)
int reader_counter:   __    (init 0)
```

```
/* writer */
while (true) {

  rw_access.wait();


  // update the DB
  // Update the DB
  // Update the DB

  rw_access.signal();


}
```

```
/* reader */
while (true) {
  rmutex.wait();
  reader_counter++;
  rmutex.signal();

  if (reader_counter == 1)
    rw_access.wait(); // only the leader holds the access key
  // read the DB
  // read the DB
  rmutex.wait();
  reader_counter--;
  rmutex.signal();
  if (reader_counter == 0)
    rw_access.signal(); // the leader releases the access key
}
```

#3: Readers/Writers: Fair Competition

```
/* writer */
while (true) {

  writer_counter++;
  if (writer_counter == 1)
    rw_permit.wait();
  rw_access.wait();

  // update the DB
  // Update the DB

  rw_access.signal();
  writer_counter--;
  if (writer_counter == 0)
    rw_permit.signal();
}
```

```
/* reader */
while (true) {

  rw_permit.wait();
  reader_counter++;
  if (reader_counter == 1)
    rw_access.wait();
  rw_permit.signal();
  // read the DB
  // read the DB
  // read the DB

  reader_counter--;
  if (reader_counter == 0)
    rw_access.signal();
```

Semaphore rw_access	= __	(init 1)
Semaphore rw_permit	= __	(init 1)
int reader_count	= __	(init 0)
int writer_couint	= __	(init 0)