Semaphores

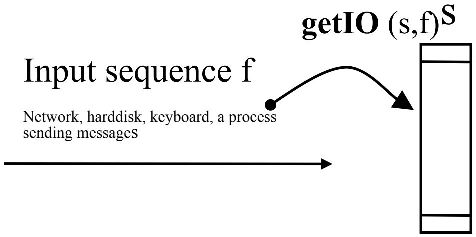
Otto J. Anshus

Concurrency: Double buffering

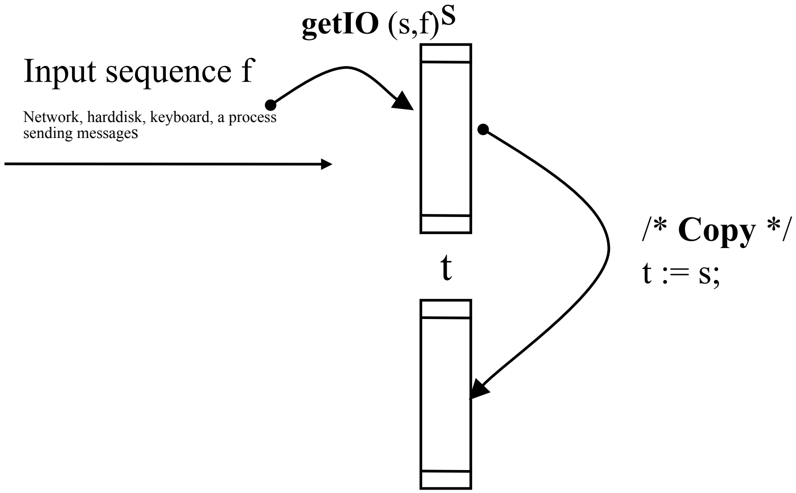
Input sequence f

Network, harddisk, keyboard, a process
sending messages

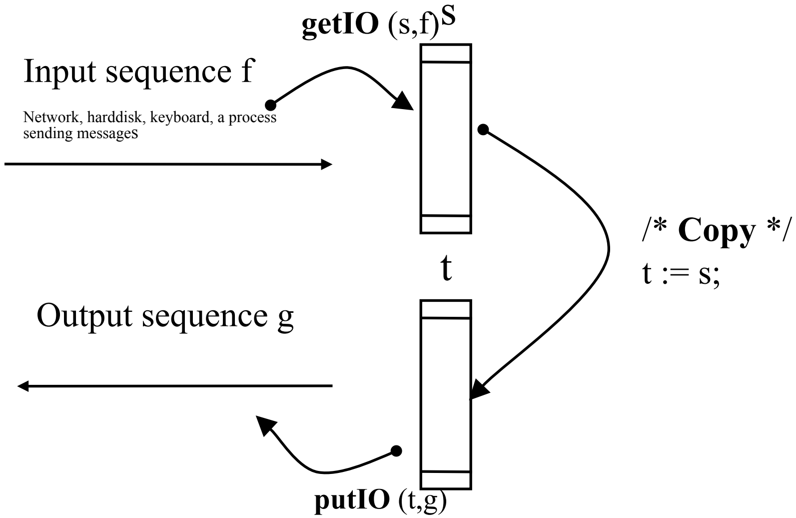
Concurrency: Double buffering



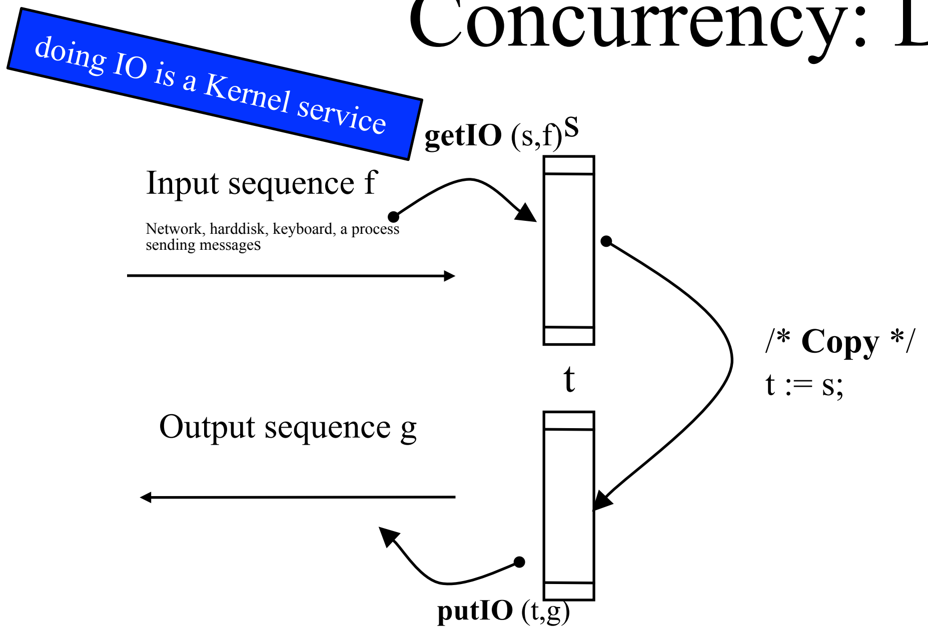
Concurrency: Double buffering



Concurrency: Double buffering

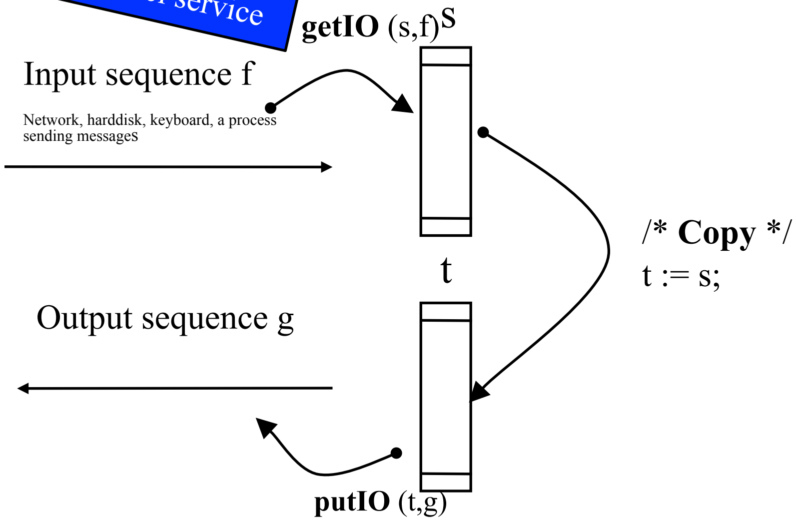


Concurrency: Double buffering



Concurrency: Double buffering

doing IO is a Kernel service



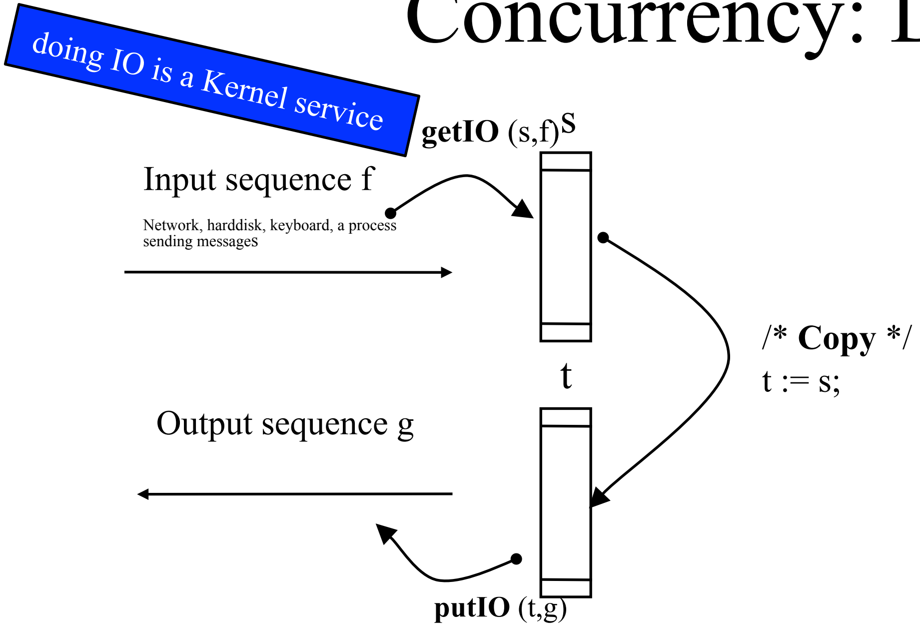
Sequential approach

```
*{
        getIO( );
        copy;
        putIO( );
}
```

What is bad with this approach?

* means loop until finished :)

Concurrency: Double buffering



Sequential approach

```
*{
     getIO( );
     copy;
     putIO( );
}
```

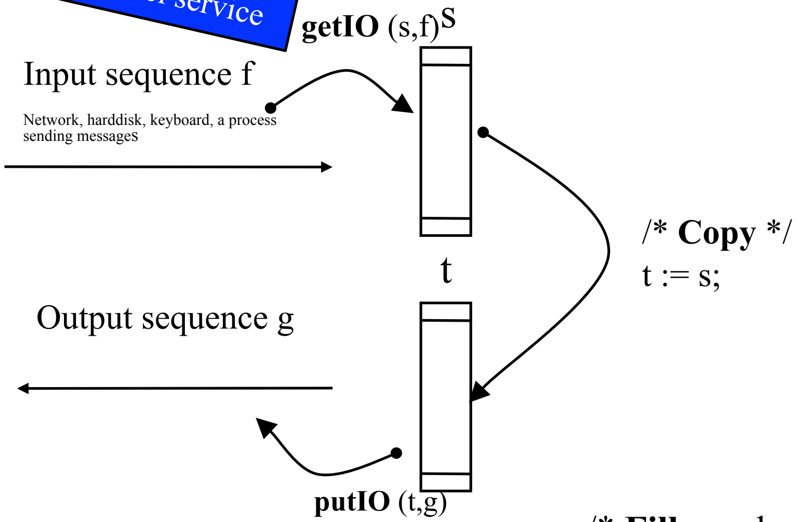
* means loop until finished :)

No concurrency so we might as well use only one buffer

What is bad with this approach?

Concurrency: Double buffering

doing IO is a Kernel service



Sequential approach

```
*{
      getIO( );
      copy;
      putIO( );
}
```

* means loop until finished :)

No concurrency so we might as well use only one buffer

What is bad with this approach?

|| specifies concurrent execution.

/* **Fill** s and **empty** t ***concurrently*** */

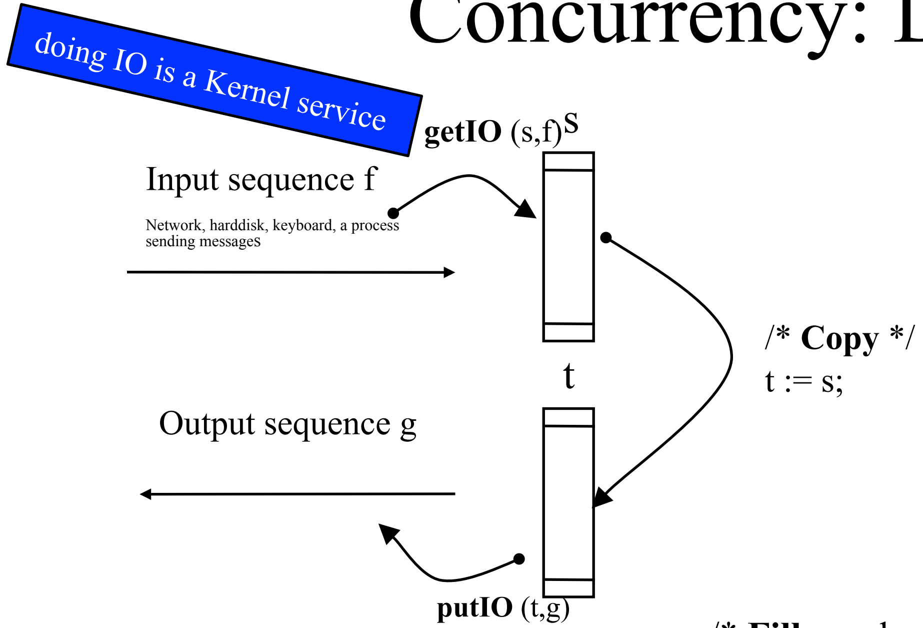
```
getIO(s,f);
*{
      Copy;
      {putIO(t,g); || getIO(s,f);}
}
```

Two concurrent threads

(Can be Interleaved or Overlapped)

(In this OS course: Interleaved)

Concurrency: Double buffering



Sequential approach

```
*{
    getIO( );
    copy;
    putIO( );
}
```

* means loop until finished :)

No concurrency so we might as well use only one buffer

What is bad with this approach?

|| specifies concurrent execution.

/* **Fill** s and **empty** t ***concurrently*** */

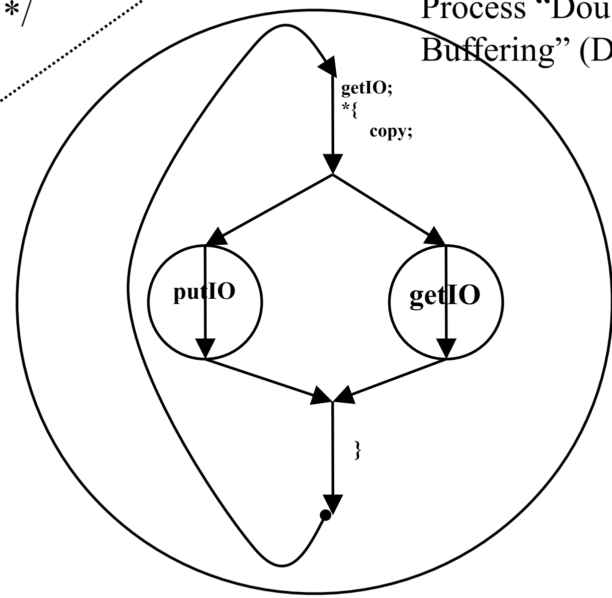
getIO(s,f);
*{
    Copy;
    {**putIO**(t,g); || **getIO**(s,f);}
}

Two concurrent threads

(Can be Interleaved or Overlapped)

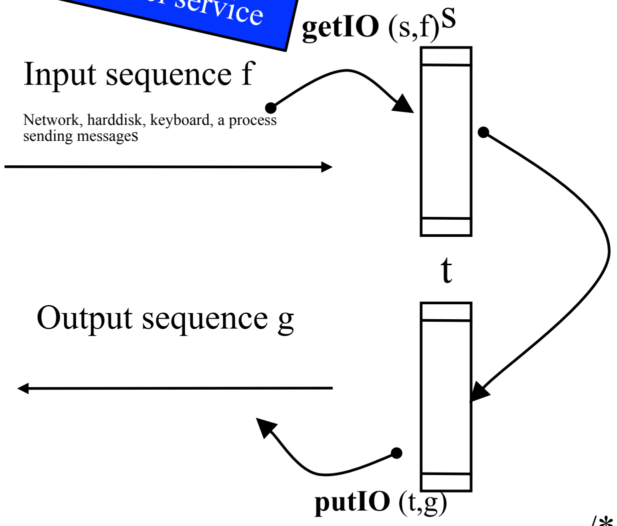
(In this OS course: Interleaved)

Process "Double Buffering" (DB)



Concurrency: Double buffering

doing IO is a Kernel service



/* **Copy** */
t := s;

Sequential approach

```
*{
      getIO( );
      copy;
      putIO( );
}
```

* means loop until finished :)

No concurrency so we might as well use only one buffer

What is bad with this approach?

|| specifies concurrent execution.

/* **Fill** s and **empty** t ***concurrently*** */

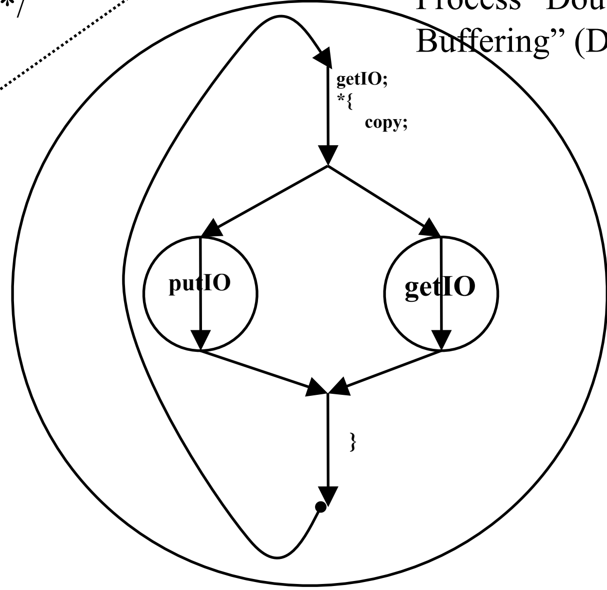
```
getIO(s,f);
*{
     Copy;
     {putIO(t,g); || getIO(s,f);}
}
```

Two concurrent threads

(Can be Interleaved or Overlapped)

(In this OS course: Interleaved)

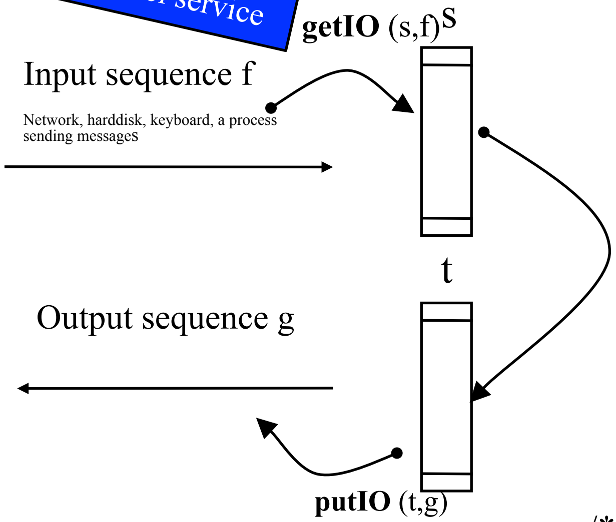
Process "Double Buffering" (DB)



- **Put and Get** are "disjoint"
  - but not with regards to **Copy**
    - **Smells like a problem...**
      - The **order** of Copy vs. Put & Get: any race conditions?
        - We are OK: order is defined by **program**

Concurrency: Double buffering

doing IO is a Kernel service



/* **Copy** */
t := s;

Sequential approach

```
*{
    getIO( );
    copy;
    putIO( );
}
```

* means loop until finished :)

No concurrency so we might as well use only one buffer

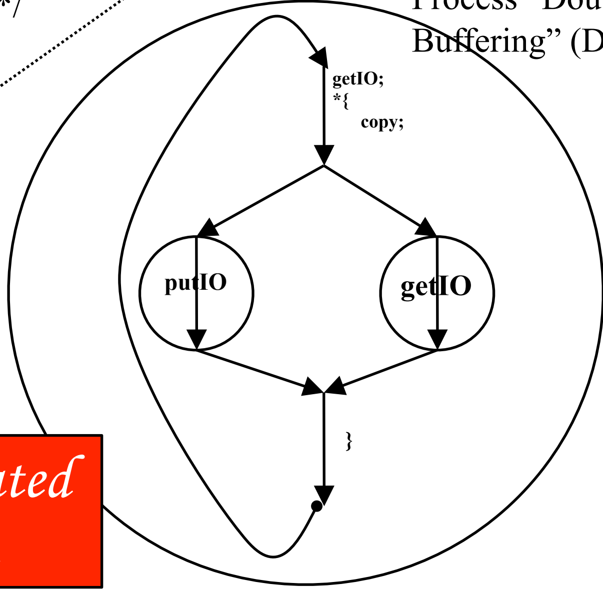
What is bad with this approach?

|| specifies concurrent execution.

/* **Fill** s and **empty** t ***concurrently*** */

```
getIO(s,f);
*{
    Copy;
    {putIO(t,g); || getIO(s,f);}
}
```

Process "Double Buffering" (DB)



- **Put and Get** are "disjoint"
  - but not with regards to **Copy**
    - **Smells like a problem...**
      - The **order** of Copy vs. Put & Get: any race conditions?
        - We are OK: order is defined by **program**

But program becomes complicated even for such a simple problem

Monday, 3.February, 2014

“Complicated Program”
OK, but can we do better?

"Complicated Program"
OK, but can we do better?

Do Better Ideas to get correct order of operations

Non-preemptive

Start all threads in a given order and maintain that order

...by OS kernel

...or at UL (yield)

Preemptive

Get the kernel scheduler to select who we want

Explicit scheduling by user level

“Complicated Program”
OK, but can we do better?

Do Better Ideas to get correct order of operations

Non-preemptive

- *Start all threads in a given order and maintain that order*
  - *...by OS kernel*
  - *...or at UL (yield)*

Complicated

Preemptive

- *Get the kernel scheduler to select who we want*
- *Explicit scheduling by user level*

"Complicated Program"
OK, but can we do better?

Do Better Ideas to get correct order of operations

Non-preemptive

- *Start all threads in a given order and maintain that order* — Complicated
  - *...by OS kernel*
  - *...or at UL (yield)*

Preemptive

- *Get the kernel scheduler to select who we want* — Complicated

- *Explicit scheduling by user level*

“Complicated Program”
OK, but can we do better?

Do Better Ideas to get correct order of operations

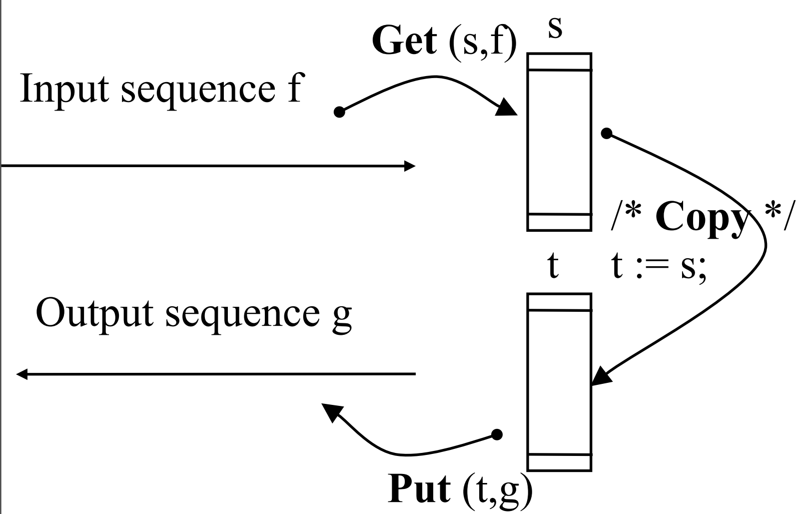
Non-preemptive
- *Start all threads in a given order and maintain that order* — Complicated
  - *...by OS kernel*
  - *...or at UL (yield)*

Preemptive
- *Get the kernel scheduler to select who we want* — Complicated

Explicit scheduling by user level — Surprisingly, this works rather well (still too complicated, though)

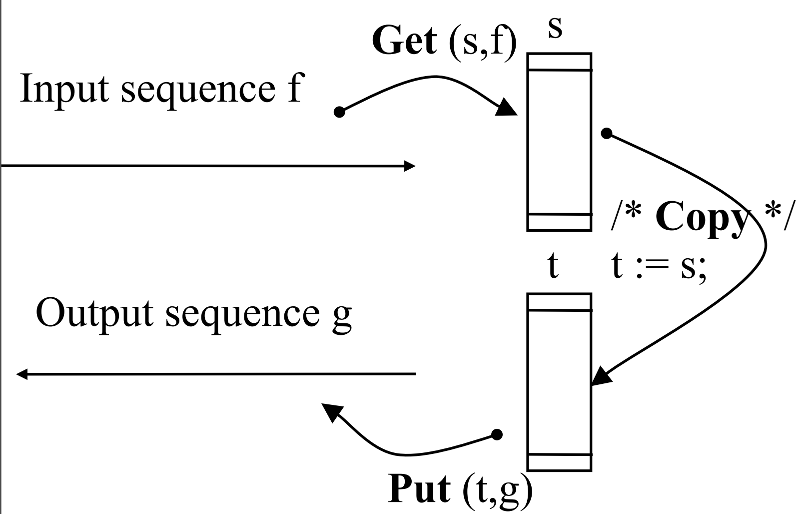
Concurrency: Double buffering

/* **Fill** s and **empty** t **concurrently**: OS Kernel will do **preemptive** scheduling of GET, COPY and PUT*/



Concurrency: Double buffering

/* **Fill** s and **empty** t **concurrently**: OS Kernel will do **preemptive** scheduling of GET, COPY and PUT*/



Three threads executing concurrently:

{put() || get() || copy} /*Assume **preemptive** sched. by kernel */

*What is **shared** between the threads?: The buffers **s** and **t**. So what can happen unless we make sure they are used by one and only one thread at a time?: **Interference** between the threads possible/likely.*

*Need **how** many **locks**?* **TWO**, one for each shared resource.

Proposed code (Not too bad, but** not **quite good enough):

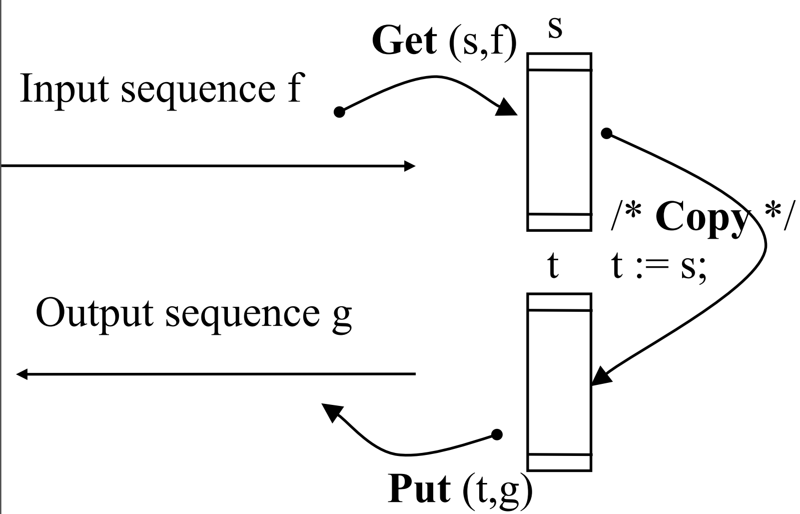
copy:: *{acq(lock_t); acq(lock_s); **t=s;** rel(lock_s); rel(lock_t);}

get:: *{acq(lock_s); **s=f;** rel(lock_s);}

put:: *{acq(lock_t): **g=t;** rel(lock_t);}

Concurrency: Double buffering

/* **Fill** s and **empty** t **concurrently**: OS Kernel will do **preemptive** scheduling of GET, COPY and PUT*/



Three threads executing concurrently:

{put() || get() || copy} /*Assume **preemptive** sched. by kernel */

*What is **shared** between the threads?: The buffers **s** and **t**. So what can happen unless we make sure they are used by one and only one thread at a time?: **Interference** between the threads possible/likely.*

*Need **how** many **locks**?* **TWO**, one for each shared resource.

Proposed code (Not too bad, but** not **quite good enough):

copy:: *{acq(lock_t); acq(lock_s); **t=s;** rel(lock_s); rel(lock_t);}

get:: *{acq(lock_s); **s=f;** rel(lock_s);}

put:: *{acq(lock_t): **g=t;** rel(lock_t);}

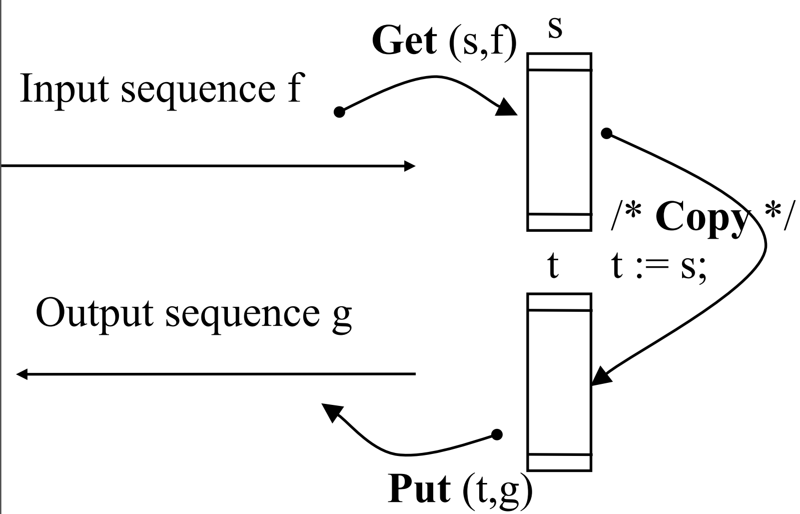
Not too bad, but the ORDER can be wrong

- Get overwrites *new* **s**
- Copy reads *old* **s**
- Copy overwrites *new* **t**
- Put reads *old* **t**

Most likely we will have a glorious mix of all of the above

Concurrency: Double buffering

/* **Fill** s and **empty** t **concurrently**: OS Kernel will do **preemptive** scheduling of GET, COPY and PUT*/



Three threads executing concurrently:

{put() || get() || copy} /*Assume **preemptive** sched. by kernel */

*What is **shared** between the threads?: The buffers **s** and **t**. So what can happen unless we make sure they are used by one and only one thread at a time?: **Interference** between the threads possible/likely.*

*Need **how** many **locks**?* **TWO**, one for each shared resource.

Proposed code (Not too bad, but** not **quite good enough):

copy:: *{acq(lock_t); acq(lock_s); **t=s;** rel(lock_s); rel(lock_t);}

get:: *{acq(lock_s); **s=f;** rel(lock_s);}

put:: *{acq(lock_t): **g=t;** rel(lock_t);}

Not too bad, but the ORDER can be wrong

- Get overwrites *new* **s**
- Copy reads *old* **s**
- Copy overwrites *new* **t**
- Put reads *old* **t**

Most likely we will have a glorious mix of all of the above

We need a way to signal conditions.

Protecting a Shared Variable (implementing locks in the OS Kernel)

- Remember: we need a *shared address space* to share variables (memory)
  - threads inside a process share an address space
  - processes: do not share address space(s) (*of course* not?)
    - (but *can* do so by exporting/importing memory regions (buffers) (not in this course))
- Assume
  - we have support in the OS kernel for user and/or kernel level threads: threads are individually scheduled without blocking the other threads (and the process itself!)
  - we have locks as an OS service, implemented by and in the Kernel.
- ***Acquire(lock_A); count++; Release(lock_A);***
  - **(1) Acquire(lock) system call**
    - User level library
      - **(2) Push parameters (acquire, lock_name) onto stack**
      - **(3) Trap to kernel (*int* instruction)**
    - Kernel level
      - Interrupt handler
        - **(4) Verify valid pointer to *lock_A***
        - Jump to code for Acquire()
          - **(5a) lock closed: *block* caller: insert(current, lock_A_wait_queue)** (and then do **out(current, Ready_Queue);** *schedule; dispatch* (to some other **thread** in same address space or even to another **process**);)
          - **(5b) lock open: close lock_A (**and *schedule: dispatch* (back library routine or to another thread or process);)
    - User level: **(6) execute count++ %**this after getting the lock
  - **(7) Release(lock) system call**
    - What should happen now if other threads are **not** waiting on lock_A?
    - ...and if other threads **are** waiting on lock_A?

Lock Performance and Cost Issues

- Should we implement the lock-mechanism waiting by *spinning* or *blocking*?
- Competition for a lock
  - *Un-contended* = rarely in use by someone else
  - *Contended* = often used by someone else
  - *Held* = currently in use by someone
- Think about the implications of these situations
  - *Contended* (**High** contention lock)
    - Spinning: **Worst** (slow in, many cpu cycles wasted)
    - Blocking: **OK** (slow in, but fewer cycles wasted vs. spinning)
  - *Un-contended* (**Low** contention lock)
    - Spinning: **Best** (fastest in, few cpu cycles wasted)
    - Blocking: **Bad** (fast in, overhead cpu cycles wasted)
- Locks done
  - by Kernel
  - by UL

Use of locks when implementing

Block/unblock

(implemented by the OS Kernel)

- **What we want to achieve**
  - **Block** thread on a queue called waitq

    q_ref *pos* *tcb_ref* *q_ref* *tcb_ref*

    - **insert** (waitq, last, **remove** (readyq, current))
  - **Unblock**

    pos is wherever the scheduler decides to insert the thread in the Ready_Queue

    - **insert** (readyq, **scheduler**, **remove** (waitq, first))
- (By the way, useful instruction:)
  - ("test and set" works both at user and kernel level)

Implementation of Block and Unblock **inside** OS Kernel

block and unblock both touch Ready_Queue and some condwait_queue so let us assume that we must protect against **concurrent** accesses

- **Block**
  - Spin until the **block_lock** is open
  - Lock lock
    - Save thread context to TCB
    - Enqueue the TCB on condwait_queue
  - Open lock
  - goto scheduler

- **UnBlock**
  - Spin until **block_lock** is open
  - Lock lock
    - -Dequeue first TCB from condwait_queue
    - -Put TCB into ready_queue
  - Open lock
  - goto scheduler

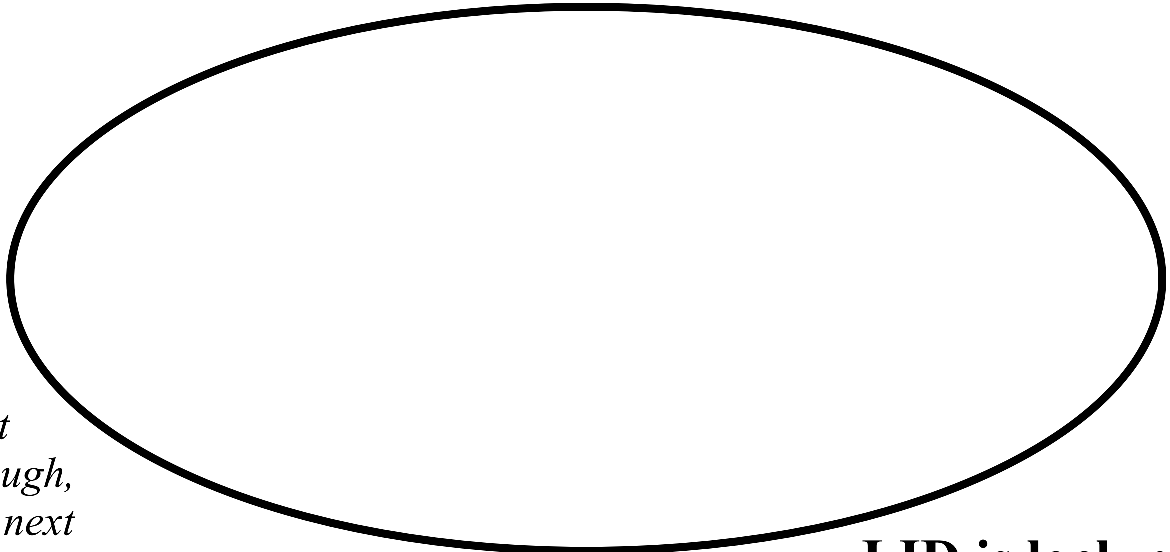
But do we really need a lock if this is implemented inside the kernel?

Is spinning such a good idea inside the kernel?

Two Styles of Synchronization

Threads inside one process: Shared address space. They can access the same variables

Process w/two threads

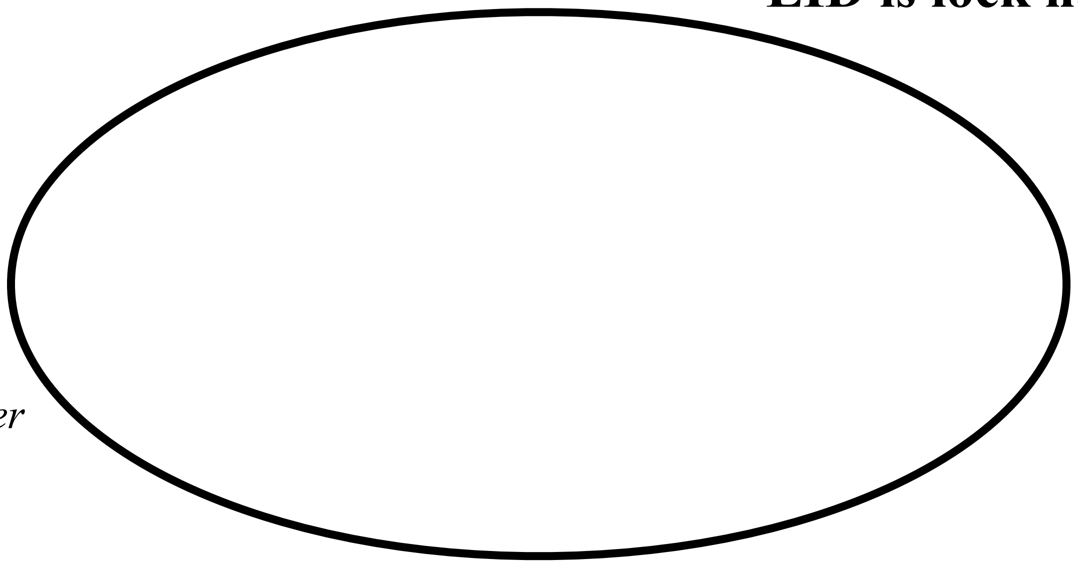


MUTEX

Acquire** will let first caller through, and then block next until **Release

LID is lock name

CONDITION SYNCHRONIZATION



Acquire** will block first caller until **Release

Two Styles of Synchronization

Threads inside one process: Shared address space. They can access the same variables

Process w/two threads

LOCK is initially **OPEN**

MUTEX

Acquire** will let first caller through, and then block next until **Release

LID is lock name

CONDITION SYNCHRONIZATION

Acquire** will block first caller until **Release

Two Styles of Synchronization

Threads inside one process: Shared address space. They can access the same variables

Process w/two threads

LOCK is initially **OPEN**

Acquire (LID);

<CR>

Release (LID);

MUTEX

Acquire *will let first caller through, and then block next until* ***Release***

LID is lock name

CONDITION SYNCHRONIZATION

Acquire *will block first caller until* ***Release***

Two Styles of Synchronization

Threads inside one process: Shared address space. They can access the same variables

Process w/two threads

LOCK is initially **OPEN**

Acquire (LID);

<CR>

Release (LID);

Acquire (LID);

<CR>

Release (LID);

MUTEX

Acquire** will let first caller through, and then block next until **Release

LID is lock name

CONDITION SYNCHRONIZATION

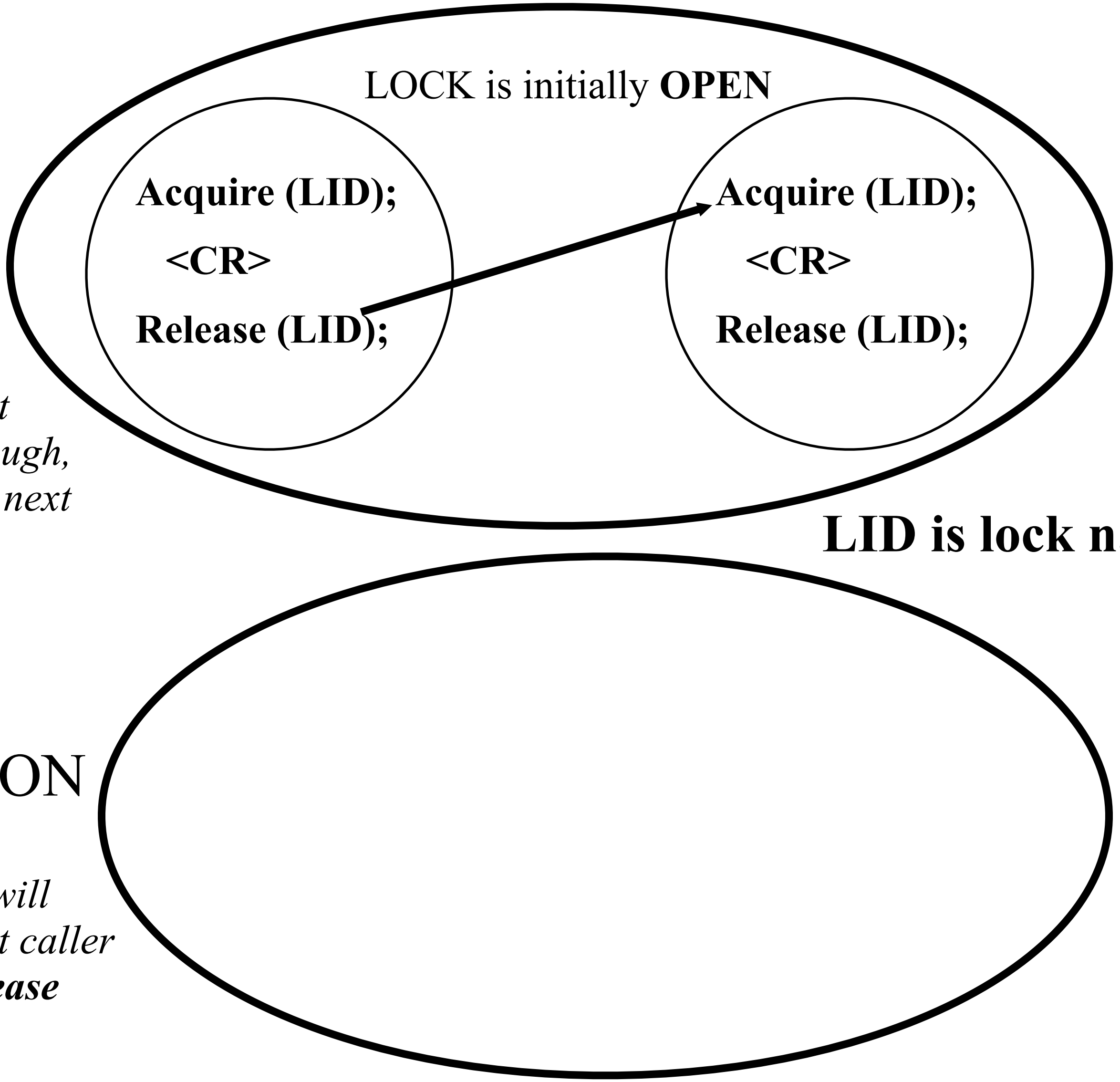
Acquire** will block first caller until **Release

Two Styles of Synchronization

Threads inside one process: Shared address space. They can access the same variables

Process w/two threads

MUTEX



Acquire *will let first caller through, and then block next until* ***Release***

LID is lock name

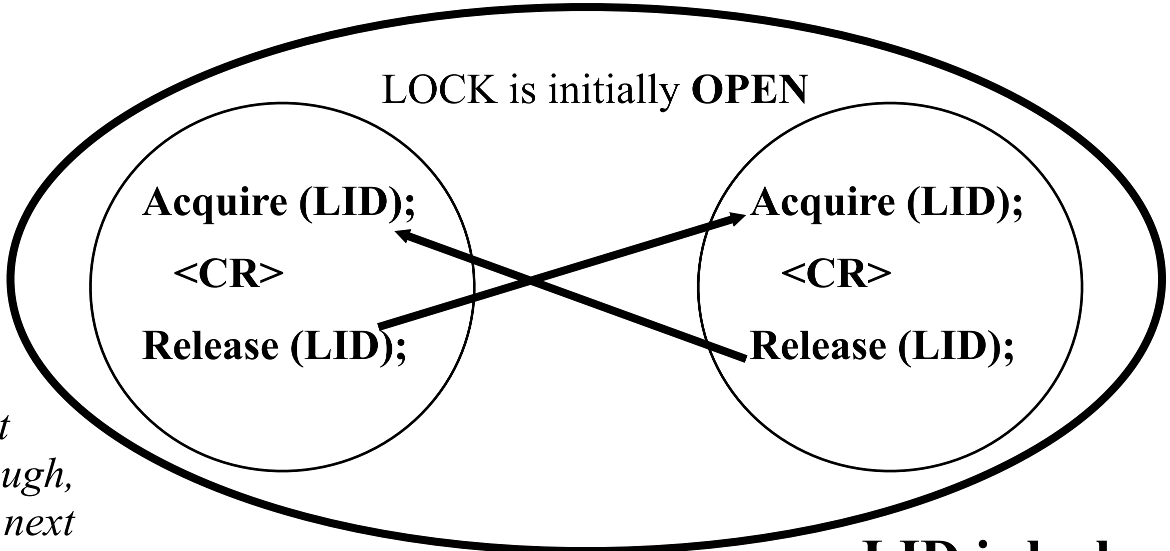
CONDITION SYNCHRONIZATION

Acquire *will block first caller until* ***Release***

Two Styles of Synchronization

Threads inside one process: Shared address space. They can access the same variables

Process w/two threads

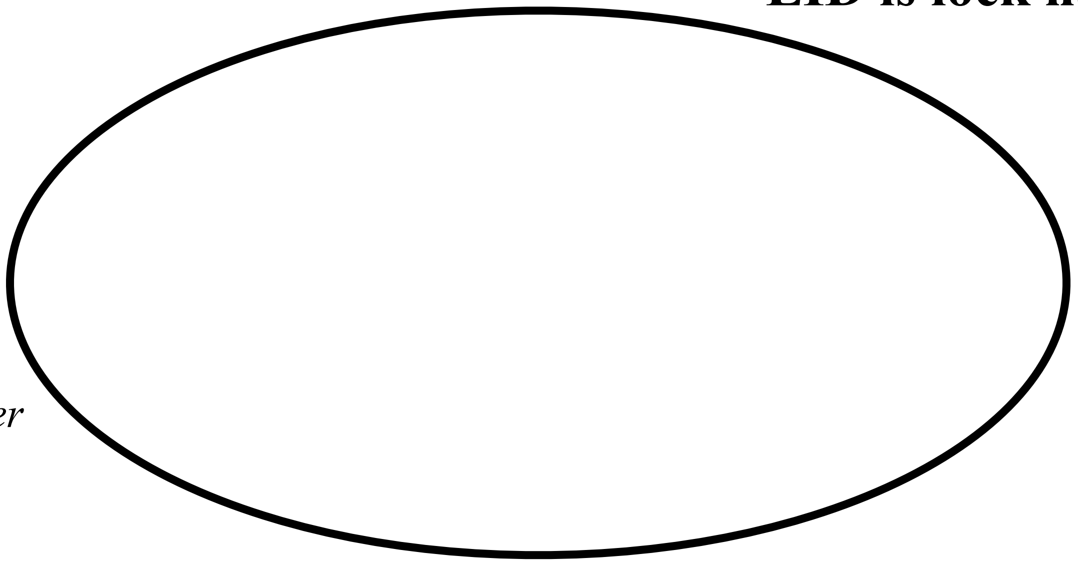


MUTEX

Acquire *will let first caller through, and then block next until* ***Release***

LID is lock name

CONDITION SYNCHRONIZATION

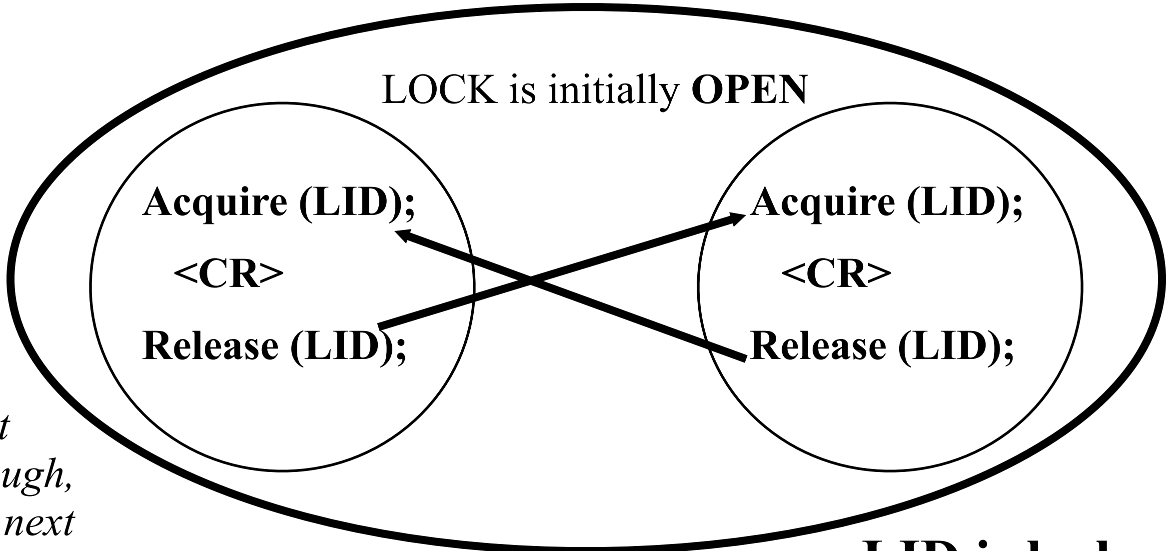


Acquire *will block first caller until* ***Release***

Two Styles of Synchronization

Threads inside one process: Shared address space. They can access the same variables

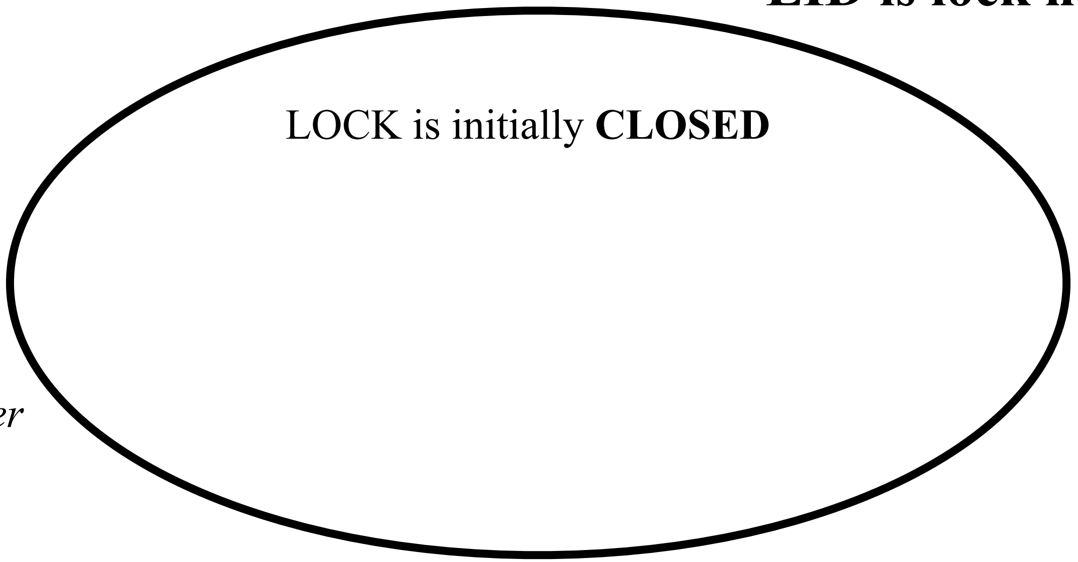
Process w/two threads



MUTEX

Acquire *will let first caller through, and then block next until* ***Release***

LID is lock name



CONDITION SYNCHRONIZATION

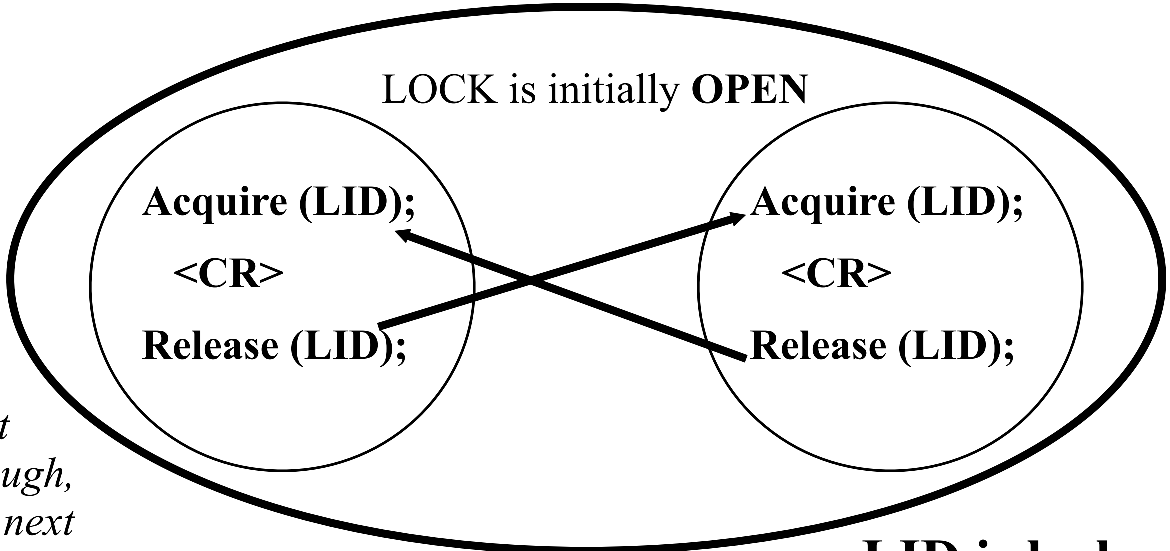
Acquire *will block first caller until* ***Release***

Two Styles of Synchronization

Threads inside one process: Shared address space. They can access the same variables

Process w/two threads

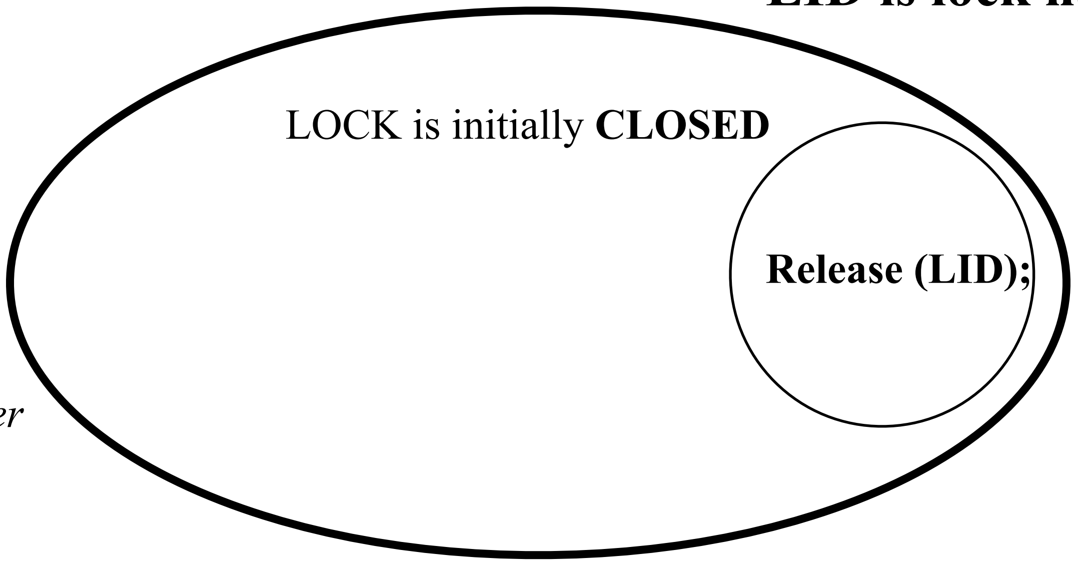
MUTEX



Acquire** will let first caller through, and then block next until **Release

LID is lock name

CONDITION SYNCHRONIZATION



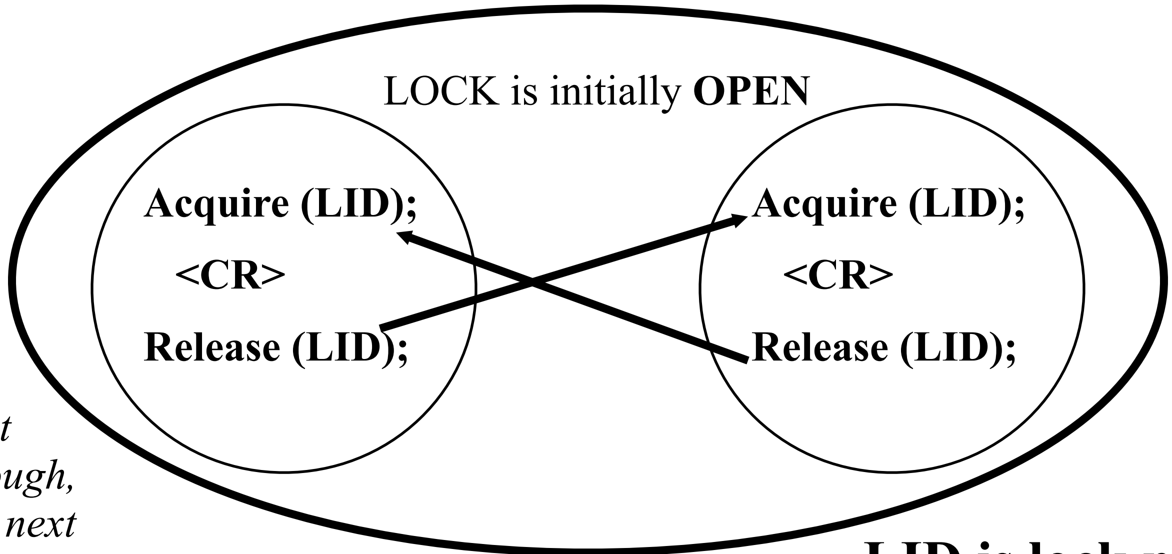
Acquire** will block first caller until **Release

Two Styles of Synchronization

Threads inside one process: Shared address space. They can access the same variables

Process w/two threads

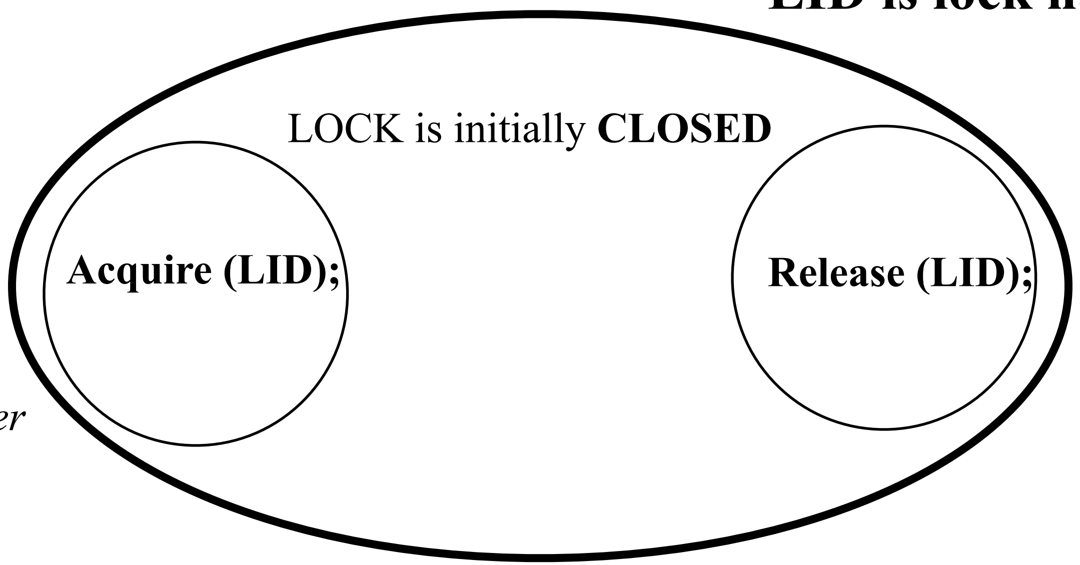
MUTEX



Acquire *will let first caller through, and then block next until* ***Release***

LID is lock name

CONDITION SYNCHRONIZATION



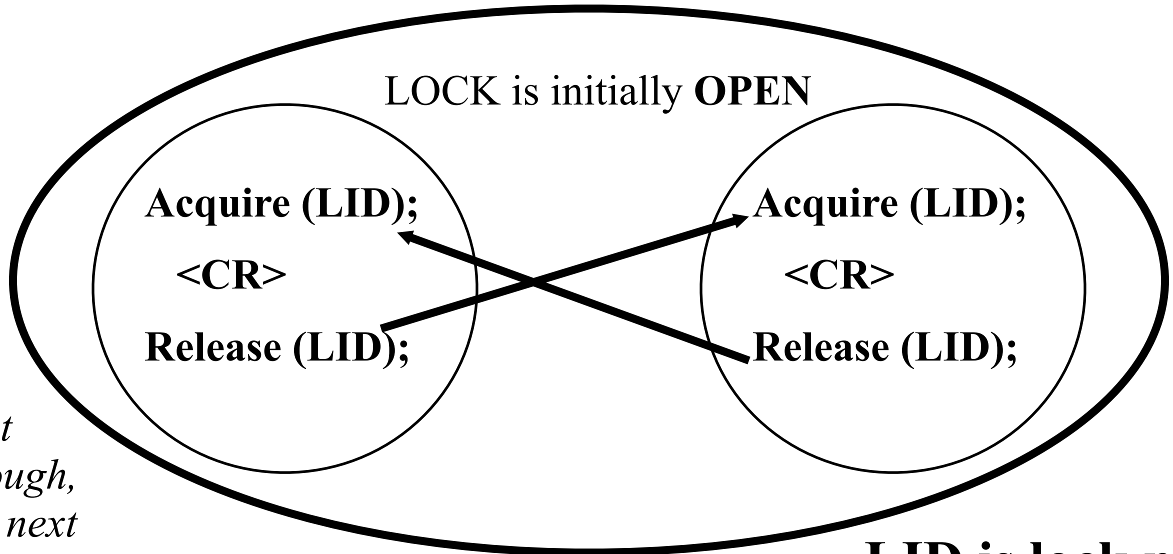
Acquire *will block first caller until* ***Release***

Two Styles of Synchronization

Threads inside one process: Shared address space. They can access the same variables

Process w/two threads

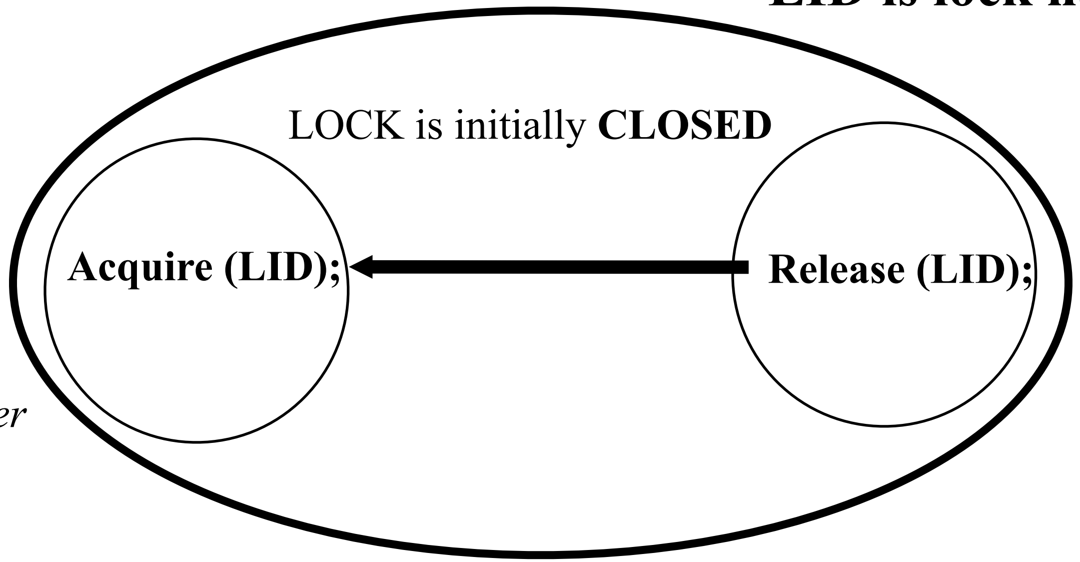
MUTEX



Acquire** will let first caller through, and then block next until **Release

LID is lock name

CONDITION SYNCHRONIZATION

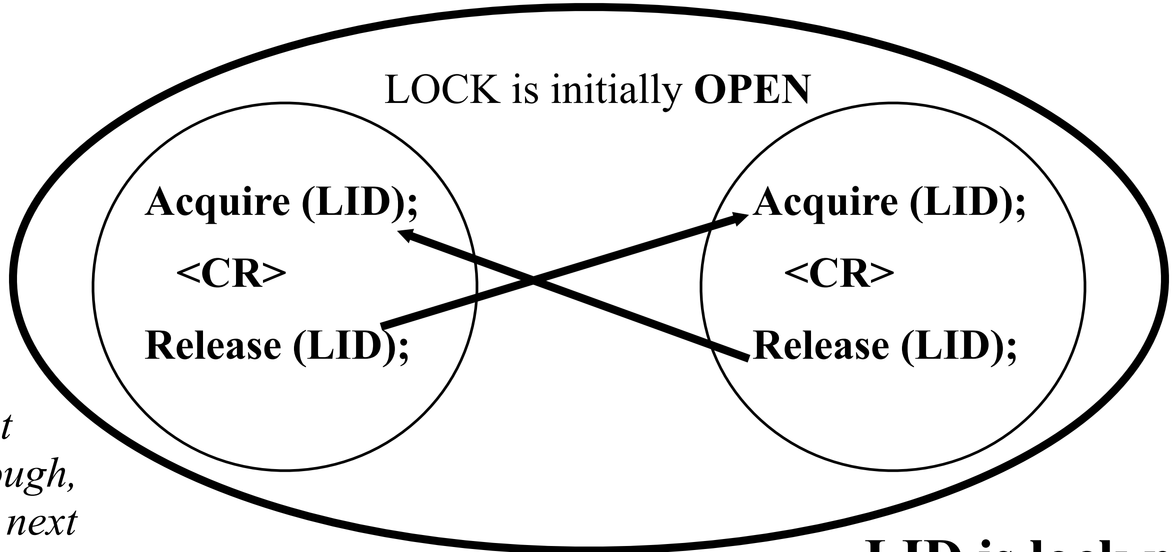


Acquire** will block first caller until **Release

Two Styles of Synchronization

Threads inside one process: Shared address space. They can access the same variables

Process w/two threads

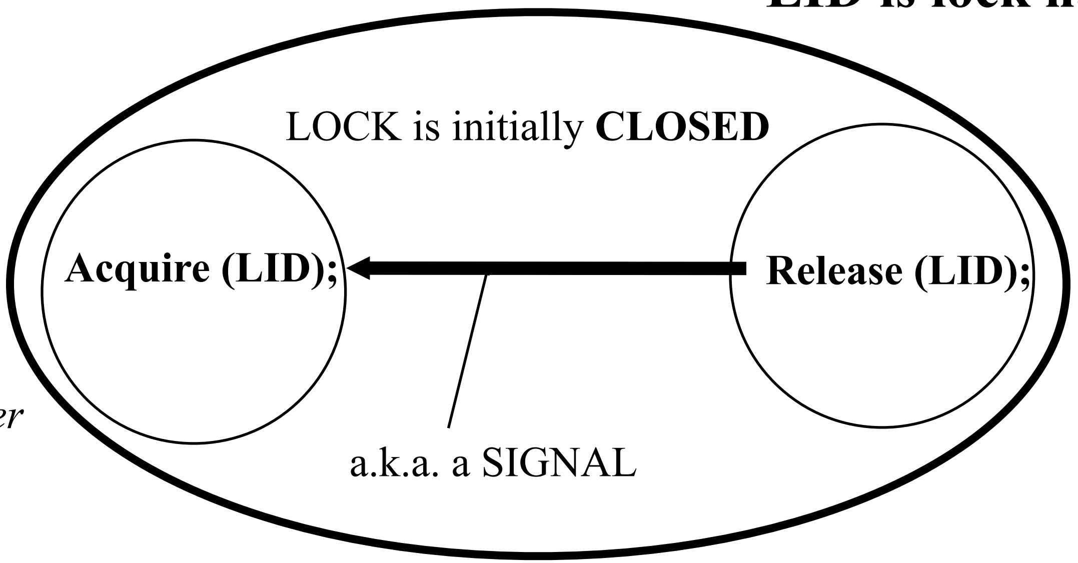


MUTEX

Acquire *will let first caller through, and then block next until* ***Release***

LID is lock name

CONDITION SYNCHRONIZATION



Acquire *will block first caller until* ***Release***

Think about ...

- Mutual exclusion using Acquire - Release:
  - Easy to forget one of them?
  - Difficult to debug?
    - must check all threads for correct use: “Acquire-CR-Release”
  - No help from the compiler?
    - It does not understand that we mean to say MUTEX
    - But could
      - check to see if we always match them “left-right”
      - associating (by specification/declaration) a variable with a Mutex, and never allow access to the variable outside of CR

Semaphores (Dijkstra, 1965)

Published as an appendix to the paper on the THE operating system

Dutch words
P: Passieren == to pass
P: Proberen == to test

V: Vrijmagen == to make free
V: Verhogen == to increment

- **Down(s)** a.k.a **Wait(s)** a.k.a **P(s)**
  - itself a critical region: MUTEX
  - delay the calling thread if s≤0
  - must decrement s by 1 for each call (and before delay!)
- **Up(s)** a.k.a **Signal(s)** a.k.a **V(s)**
  - itself a critical region: MUTEX
  - Increment semaphore by 1
  - Wake up the longest waiting thread *if any*

MUTEX

P(s)

```
{
  if (--s < 0)
    Block(s);
}
```

s must NOT be accessible through other means than calling P and V

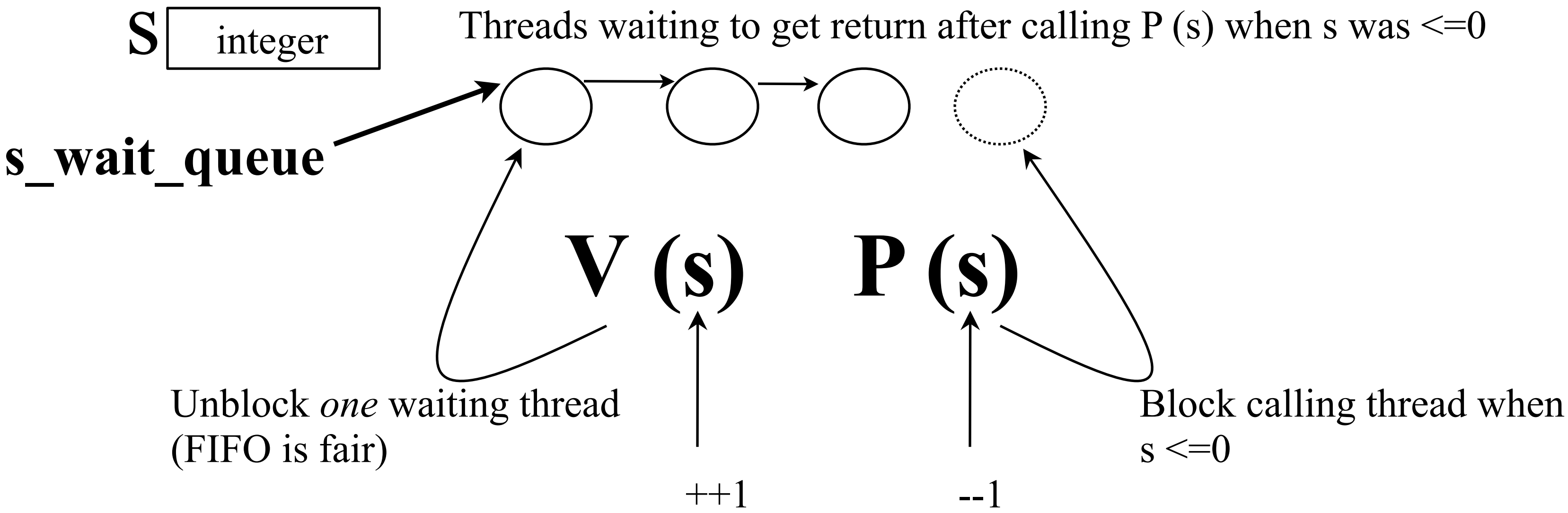
V(s)

```
{
  if (++s <= 0)
    Unblock(s);
}
```

The semaphore, s, *must* be given an *initial* value

Can get **negative** s: counts number of waiting threads

A Blocking Semaphore Implementation



- NB: **s** and **waitq** are *shared resources*

  So what?

- Approaches to achieve atomicity

  Disable interrupts

  P() and V() as System calls

  Entry-Exit protocols

A Spinning Semaphore Implementation?

MUTEX

P(s):

```
while (s <= 0) {};
s--;
```

V(s):

```
s++;
```

A Spinning Semaphore Implementation?

MUTEX

P(s):

```
while (s <= 0) {};
s--;
```

V(s):

```
s++;
```

"You Got a Problem with This?"

Spinning Semaphore

P(s):

```
while (s <= 0) {};
s--;
```

V(s):

```
s++;
```

Spinning Semaphore

P(s):

```
while (s <= 0) {};
s--;
```

V(s):

```
s++;
```

If P spinning inside mutex then V will not get in

- *Starvation possible (Lady Luck may ignore/favor some threads)*
  - *Of P's*
  - *Of V's*

Must open mutex, say, between every iteration of while() to make it possible for V to get in

- *Costly*
  - *Every 10th iteration?*
    - *Latency*

Implementation of Semaphores

- Implementing the P and V of semaphores
  - If WAIT is done by blocking
    - Expensive
    - Must open mutex
      - **But no real problems because we have a waiting queue now and we will not get starvation**
  - If done by spinning
    - Must open mutex during spin to let V in
      - Starvation of P’s and V’s possible
        - **May not be a problem in practice**
    - What can we do to “do better”?

Implementing Semaphores using **Locks**

Using **locks** to implement a **semaphore**

- mutex lock: lock is initially **open**
- "delay me" lock: lock is initially **locked**

- SEMAPHORE value is called "s.value" in the code below: Initially **0**

```
P(s) {
  Acquire(s.mutex);
  if (--s.value < 0) {
    Release(s.mutex);
    Acquire(s.delay);
  } else
    Release(s.mutex);
}
```

```
V(s) {
  Acquire(s.mutex);
  if (++s.value <= 0)
    Release(s.delay);
  Release(s.mutex);
}
```

- Kotulski (1988)
  - Two processes call P(s) (s.value is initialized to 0) and preempted after Release(s.mutex)
  - Two other processes call V(s)

Threads :)

Implementing Semaphores using **Locks**

Using **locks** to implement a **semaphore**

- mutex lock: lock is initially **open**
- “delay me” lock: lock is initially **locked**

- SEMAPHORE value is called “s.value” in the code below: Initially **0**

```
P(s) {
  Acquire(s.mutex);
  if (--s.value < 0) {
    Release(s.mutex);
    Acquire(s.delay);
  } else
    Release(s.mutex);
}
```

```
V(s) {
  Acquire(s.mutex);
  if (++s.value <= 0)
    Release(s.delay);
  Release(s.mutex);
}
```

Trouble

- Kotulski (1988)
  - Two processes call P(s) (s.value is initialized to 0) and preempted after Release(s.mutex)
  - Two other processes call V(s)

Threads :)

Implementing Semaphores using **Locks**

Using **locks** to implement a **semaphore**

- mutex lock: lock is initially **open**
- “delay me” lock: lock is initially **locked**

- SEMAPHORE value is called “s.value” in the code below: Initially **0**

```
P(s) {
  Acquire(s.mutex);
  if (--s.value < 0) {
    Release(s.mutex);
    Acquire(s.delay);
  } else
    Release(s.mutex);
}
```

```
V(s) {
  Acquire(s.mutex);
  if (++s.value <= 0)
    Release(s.delay);
  Release(s.mutex);
}
```

Trouble

- Kotulski (1988)
  - Two processes call P(s) (s.value is initialized to 0) and preempted after Release(s.mutex)
  - Two other processes call V(s)

Threads :)

Implementing Semaphores using **Locks**

Using **locks** to implement a **semaphore**

- mutex lock: lock is initially **open**
- “delay me” lock: lock is initially **locked**

- SEMAPHORE value is called “s.value” in the code below: Initially **0**

```
P(s) {
  Acquire(s.mutex);
  if (--s.value < 0) {
    Release(s.mutex);
    Acquire(s.delay);
  } else
    Release(s.mutex);
}
```

```
V(s) {
  Acquire(s.mutex);
  if (++s.value <= 0)
    Release(s.delay);
  Release(s.mutex);
}
```

“Lost” V calls: locks have no memory

Trouble

- Kotulski (1988)
  - Two processes call P(s) (s.value is initialized to 0) and preempted after Release(s.mutex)
  - Two other processes call V(s)

Threads :)

Hemmendinger's solution (1988)

```
P(s) {
  Acquire(s.mutex);
  if (--s.value < 0) {
    Release(s.mutex);
    Acquire(s.delay);
  }
  Release(s.mutex);
}
```

```
V(s) {
  Acquire(s.mutex);
  if (++s.value <= 0)
    Release(s.delay);
  else
    Release(s.mutex);
}
```

- The idea is not to release s.mutex and turn it over individually to the waiting process
- P and V are executing in locksteps

Kearn’s Solution (1988)

```
P(s) {
  Acquire(s.mutex);
  if (--s.value < 0) {
    Release(s.mutex);
    Acquire(s.delay);
    Acquire(s.mutex);
    if (--s.wakecount > 0)
      Release(s.delay);
  }
  Release(s.mutex);
}
```

```
V(s) {
  Acquire(s.mutex);
  if (++s.value <= 0) {
    s.wakecount++;
    Release(s.delay);
  }
  Release(s.mutex);
}
```

Two Release(s.delay) calls are also possible

Hemmendinger's Correction (1989)

```
P(s) {
  Acquire(s.mutex);
  if (--s.value < 0) {
    Release(s.mutex);
    Acquire(s.delay);
    Acquire(s.mutex);
    if (--s.wakecount > 0)
      Release(s.delay);
  }
  Release(s.mutex);
}
```

```
V(s) {
  Acquire(s.mutex);
  if (++s.value <= 0) {
    s.wakecount++;
    if (s.wakecount == 1)
      Release(s.delay);
  }
  Release(s.mutex);
}
```

Correct but a complex solution

Hsieh's Solution (1989)

```
P(s) {
  Acquire(s.delay);
  Acquire(s.mutex);
  if (--s.value > 0)
    Release(s.delay);
  Release(s.mutex);
}
```

```
V(s) {
  Acquire(s.mutex);
  if (++s.value == 1)
      Release(s.delay);
  Release(s.mutex);
}
```

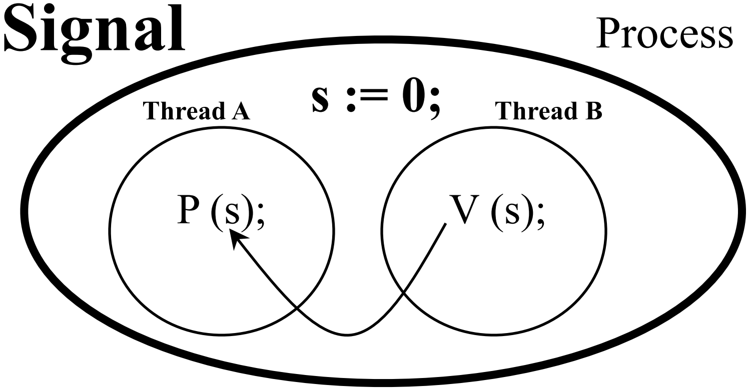
- Use Acquire(s.delay) to block processes
- Correct but still a constrained implementation

Enough

- Why don't you just implement P and V in the Kernel using blocking? :)

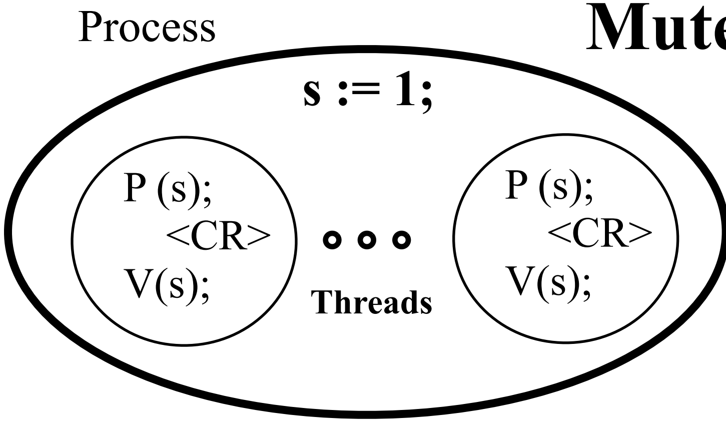
Using Semaphores

Signal



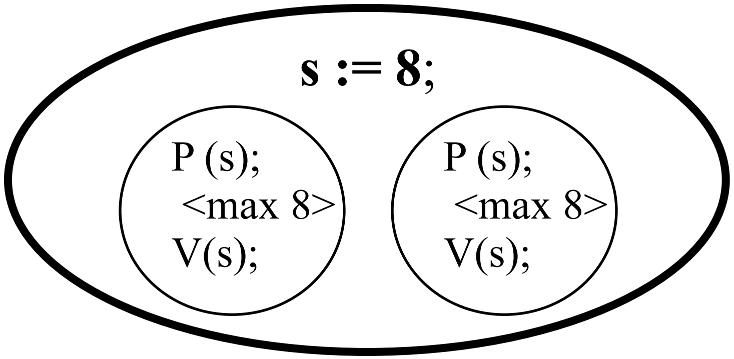
Thread A is delayed until
thread B says V(s)

Mutex



One thread gets in, next is
delayed until V is executed

NB: remember to set the initial semaphore value!

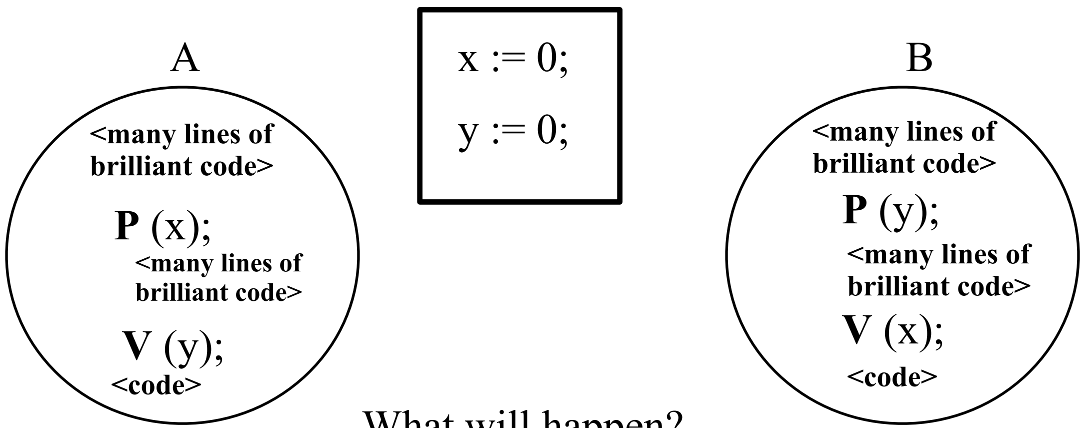


Up to 8 threads can pass P, the ninth
will block until V is said by one of
the eight already in there

Simple to debug?

*The **plan** is to have thread A wait for a signal from B and vice versa.*

Semaphores in shared memory accessible to both thread A and B

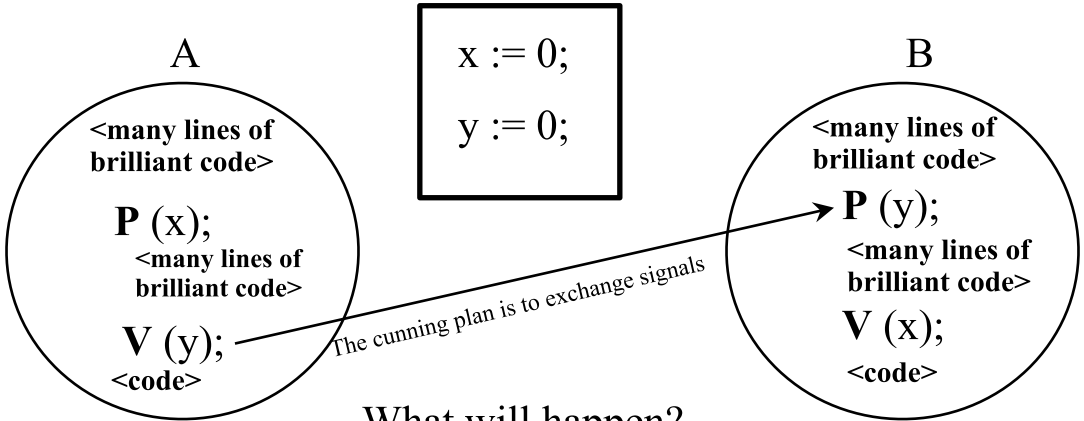


What will happen?

Simple to debug?

*The **plan** is to have thread A wait for a signal from B and vice versa.*

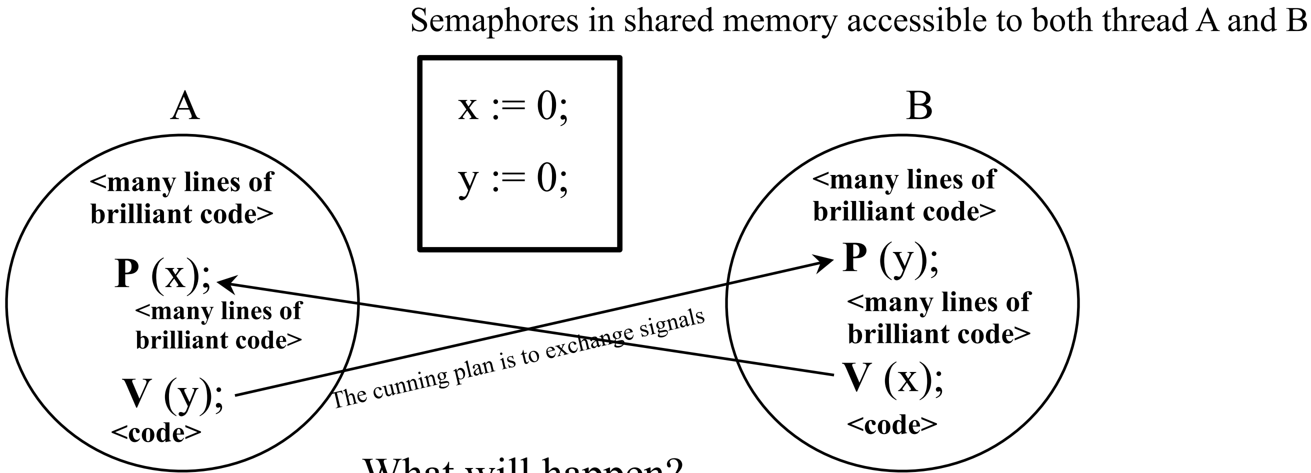
Semaphores in shared memory accessible to both thread A and B



What will happen?

Simple to debug?

*The **plan** is to have thread A wait for a signal from B and vice versa.*

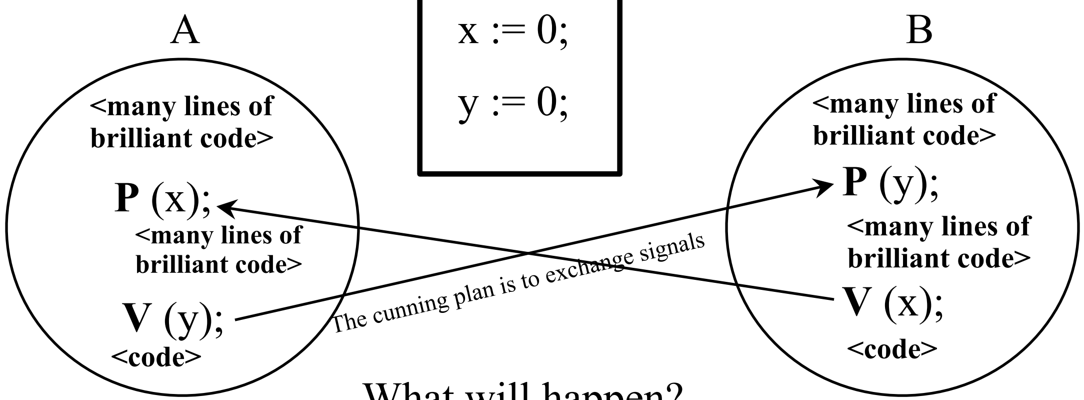


What will happen?

Simple to debug?

*The **plan** is to have thread A wait for a signal from B and vice versa.*

Semaphores in shared memory accessible to both thread A and B



What will happen?

Not all plans will come through

The two threads ARE FOREVER WAITING FOR EACH OTHERS SIGNAL

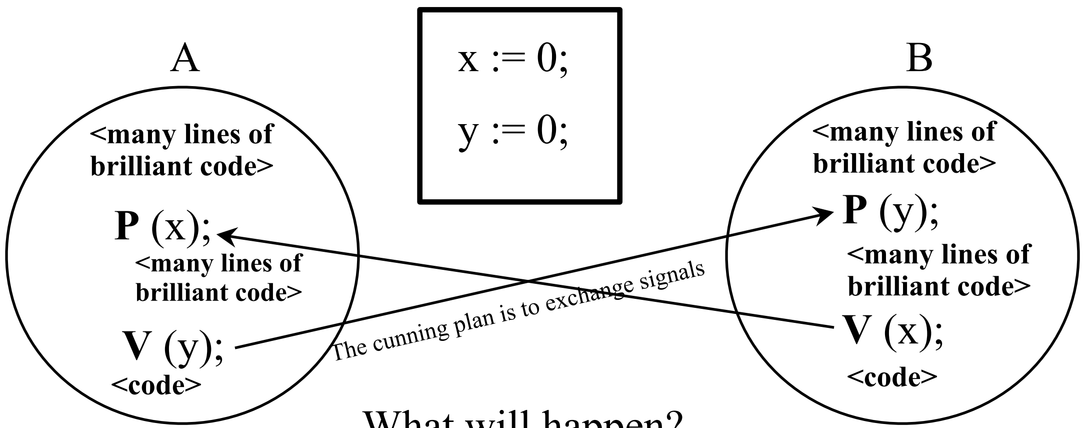
Circular Wait

A classic (*but not good*) situation resulting in a...

Simple to debug?

*The **plan** is to have thread A wait for a signal from B and vice versa.*

Semaphores in shared memory accessible to both thread A and B



What will happen?

Not all plans will come through

The two threads ARE FOREVER WAITING FOR EACH OTHERS SIGNAL

Circular Wait

A classic (*but not good*) situation resulting in a... deadlock

More to scale

A

<many

linP(x);es

of

brilliant

code>

<many

lines

of

brilliV(y);ant

code>

B

<many

lines

of

brilP(y);liant

code>

<many

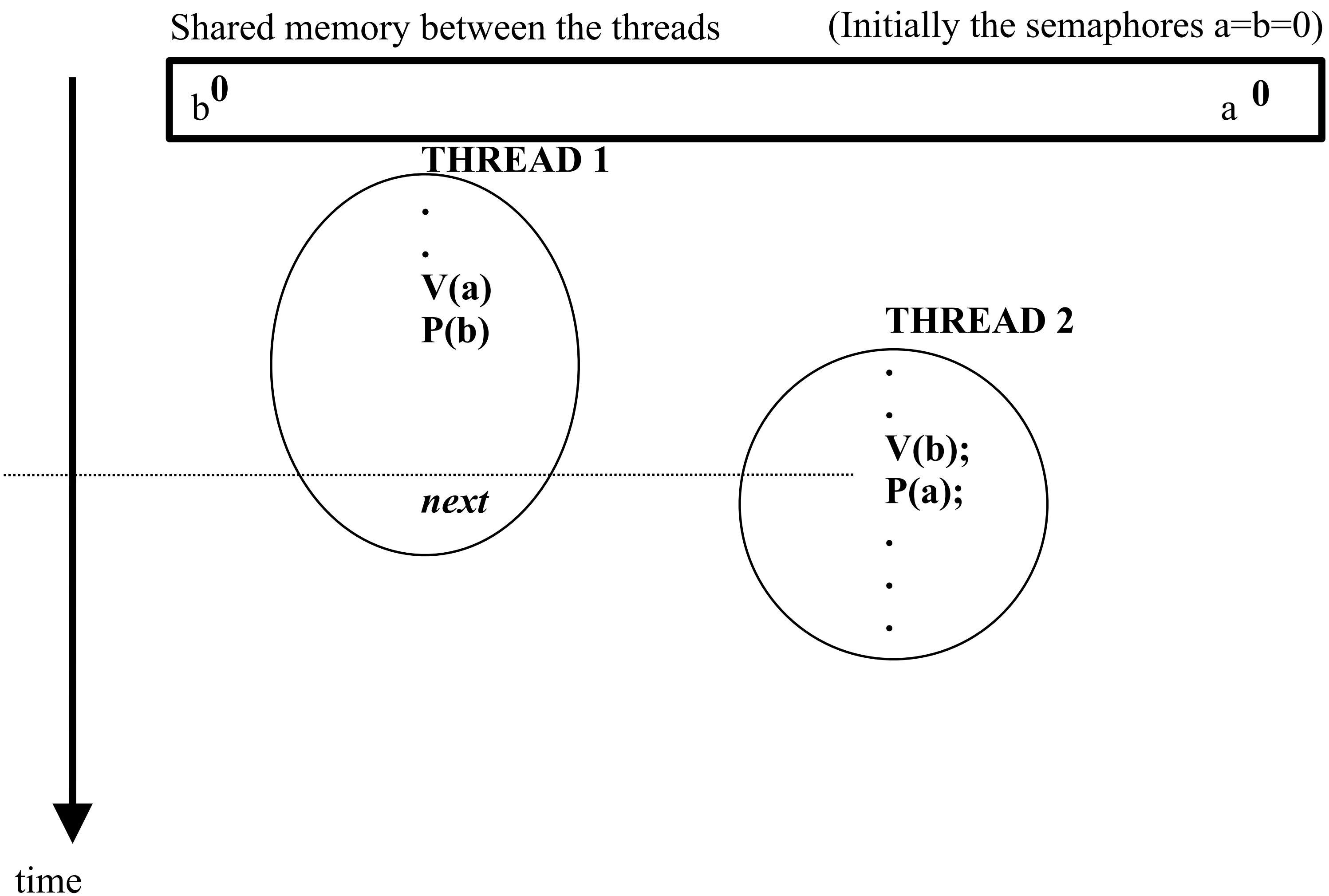
lines

of

brilliant

coV(x);de>

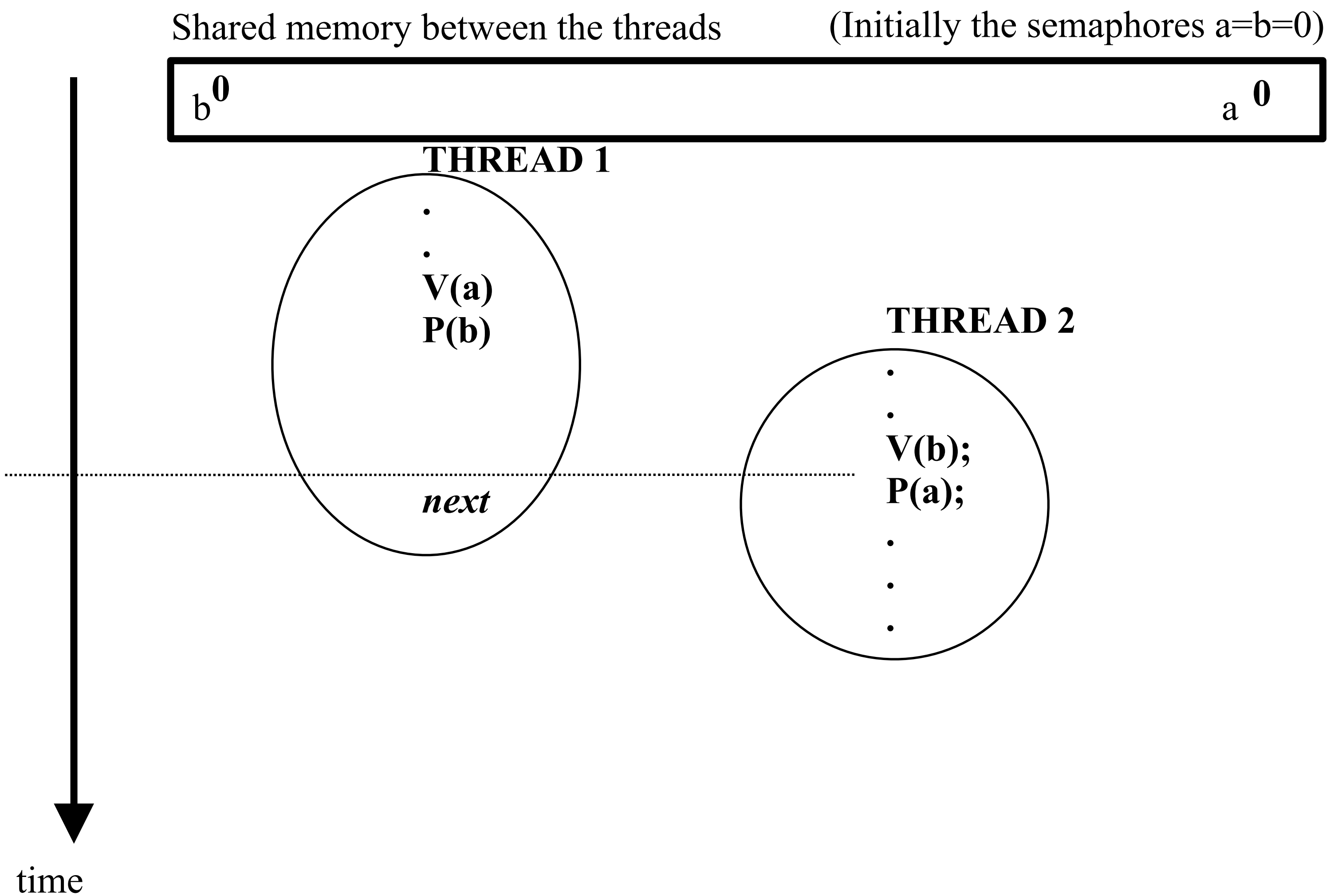
Rendezvous between two threads (or: a *Barrier* for two threads)



Rendezvous between two threads (or: a *Barrier* for two threads)

Initially both threads are in the Ready_Queue.

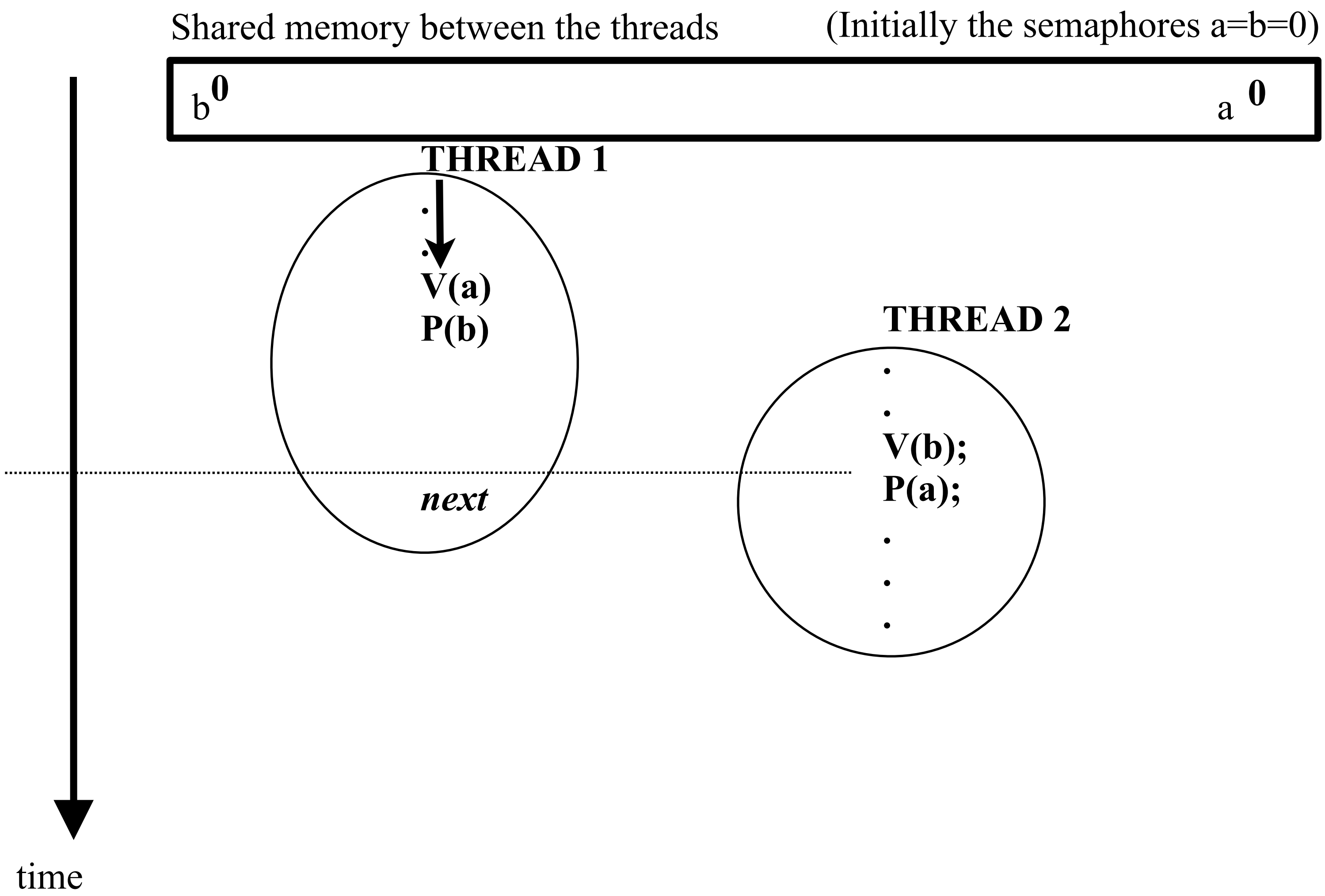
Assume that Thread 1 is scheduled to run first



Rendezvous between two threads (or: a *Barrier* for two threads)

Initially both threads are in the Ready_Queue.

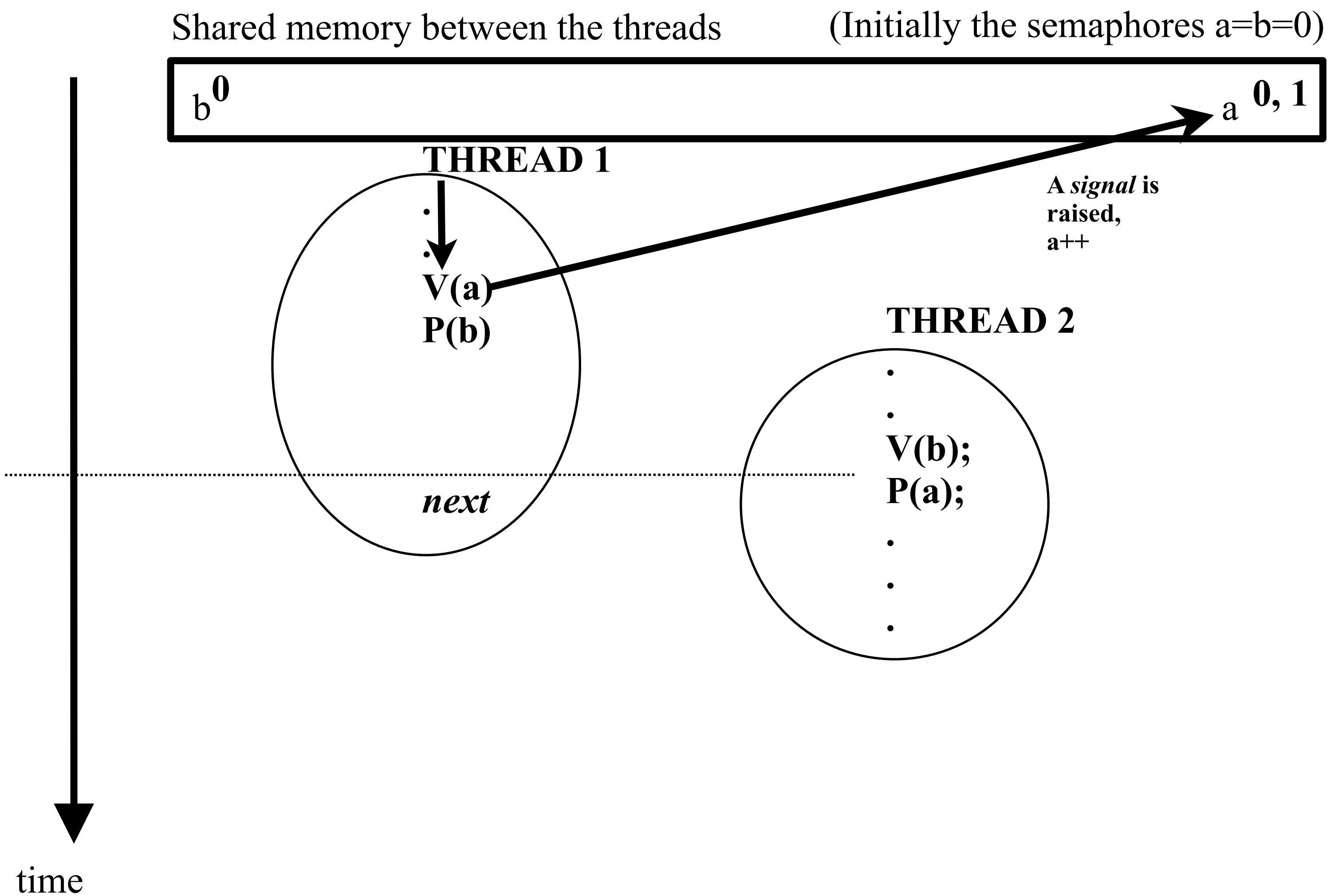
Assume that Thread 1 is scheduled to run first



Rendezvous between two threads (or: a *Barrier* for two threads)

Initially both threads are in the Ready_Queue.

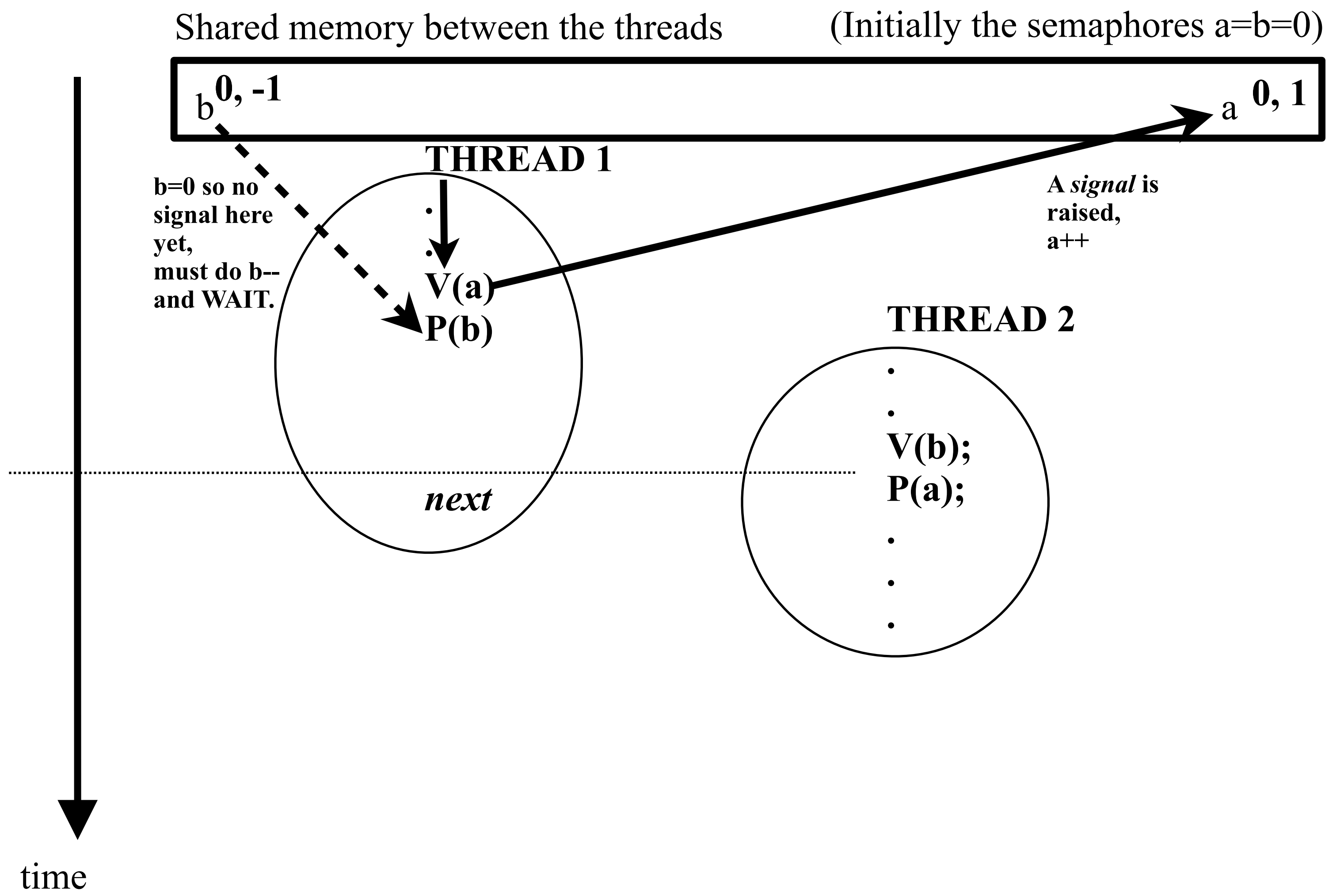
Assume that Thread 1 is scheduled to run first



Rendezvous between two threads (or: a *Barrier* for two threads)

Initially both threads are in the Ready_Queue.

Assume that Thread 1 is scheduled to run first

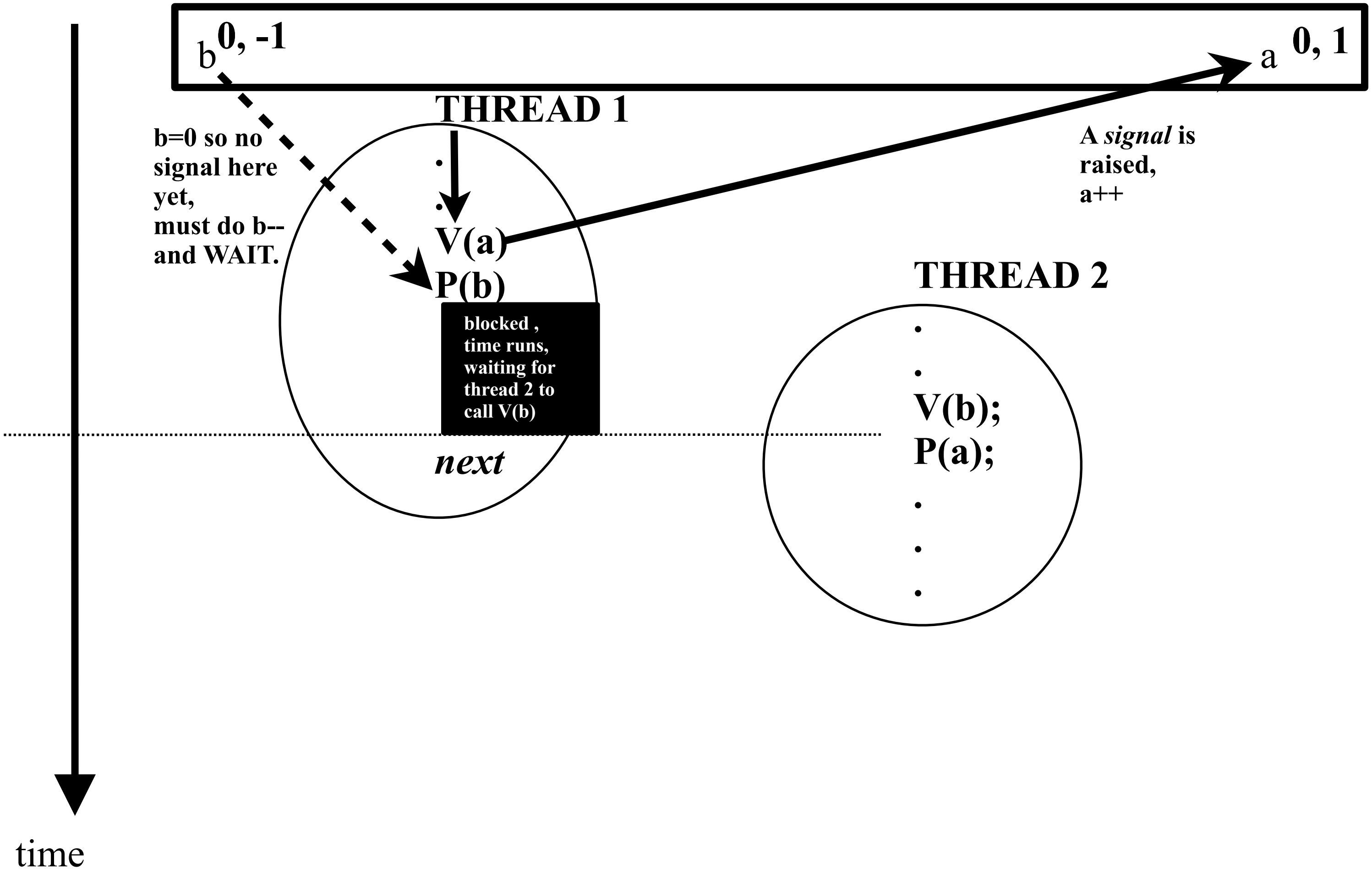


Rendezvous between two threads (or: a *Barrier* for two threads)

Initially both threads are in the Ready_Queue.

Assume that Thread 1 is scheduled to run first

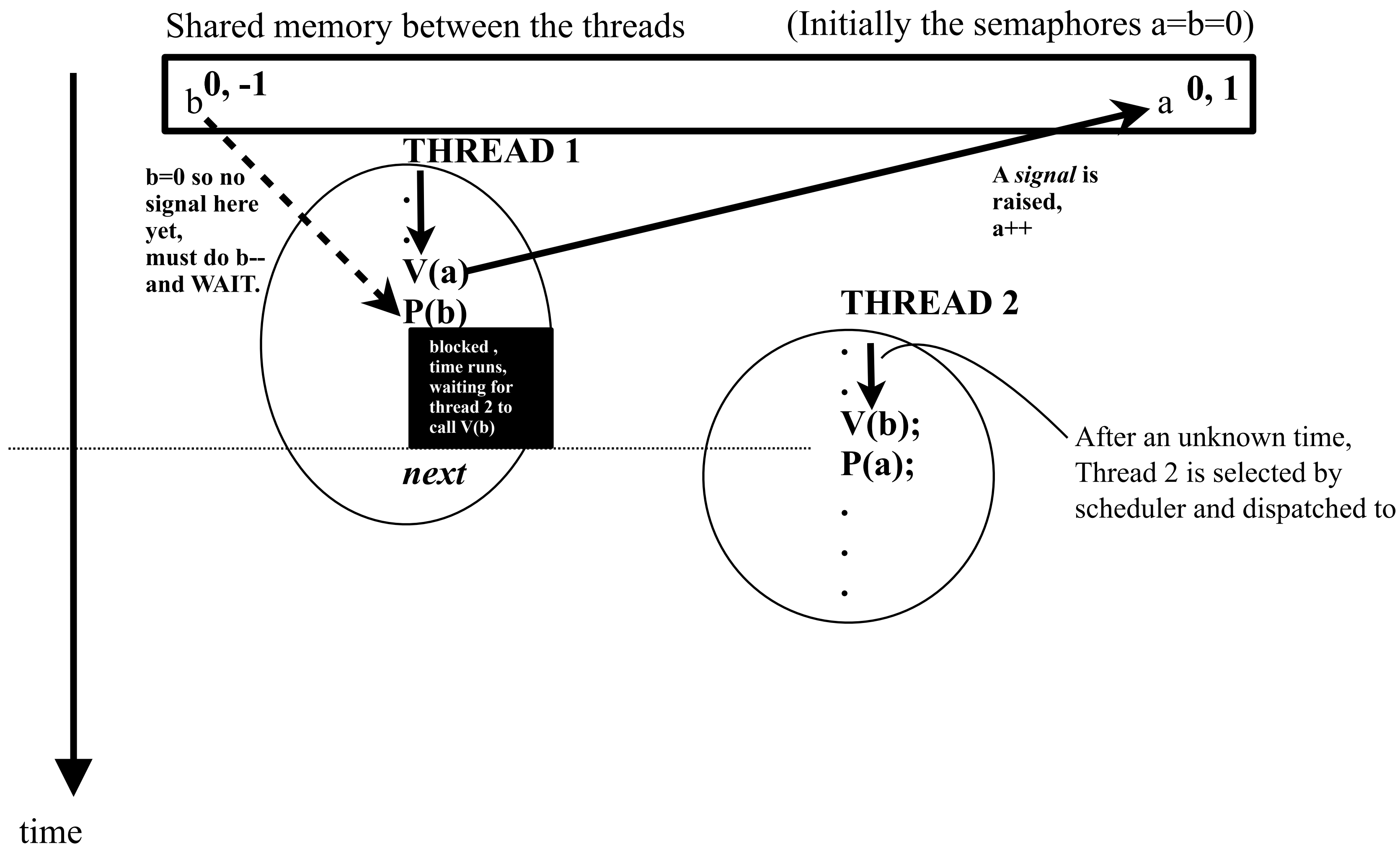
Shared memory between the threads (Initially the semaphores a=b=0)



Rendezvous between two threads (or: a *Barrier* for two threads)

Initially both threads are in the Ready_Queue.

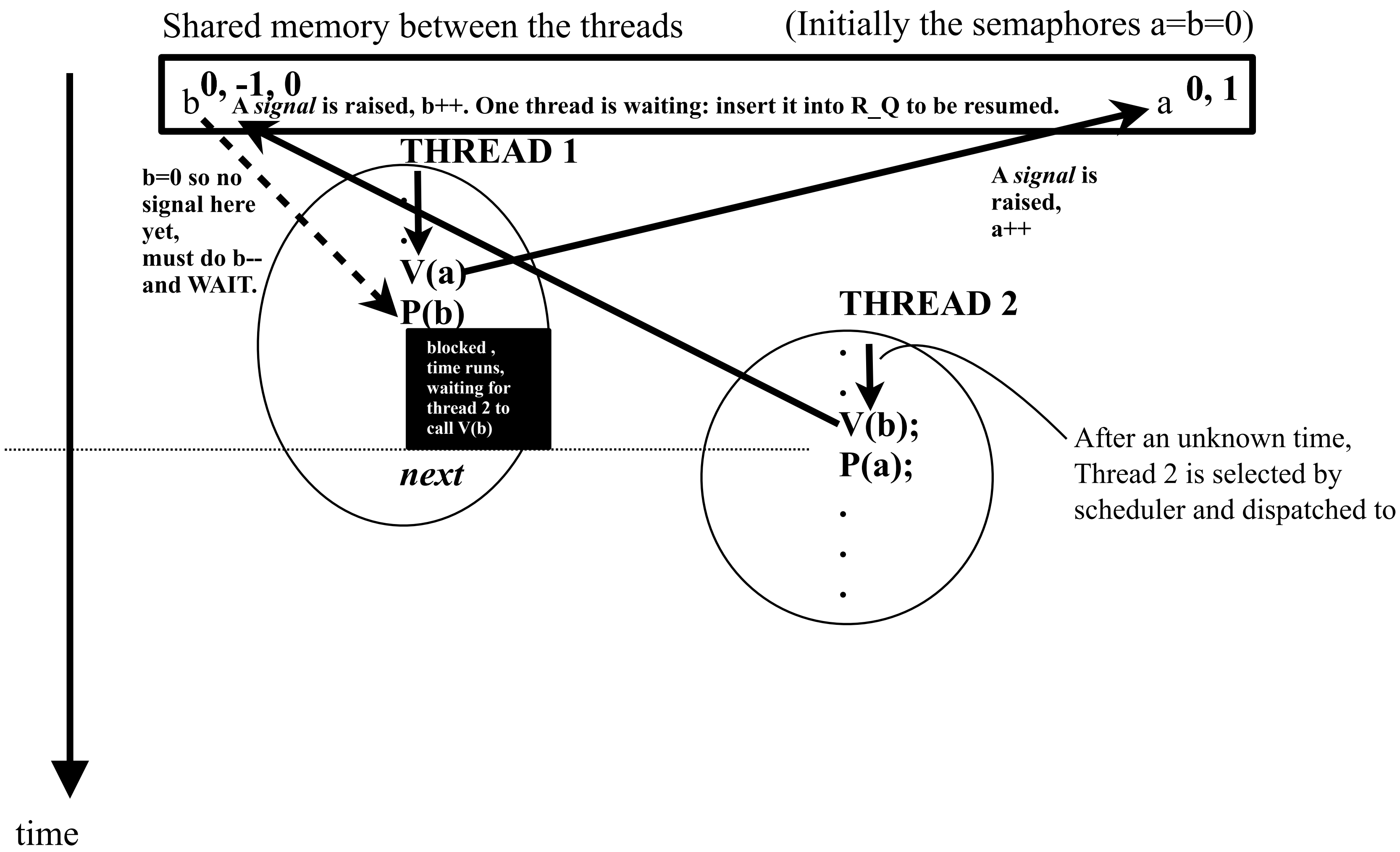
Assume that Thread 1 is scheduled to run first



Rendezvous between two threads (or: a *Barrier* for two threads)

Initially both threads are in the Ready_Queue.

Assume that Thread 1 is scheduled to run first

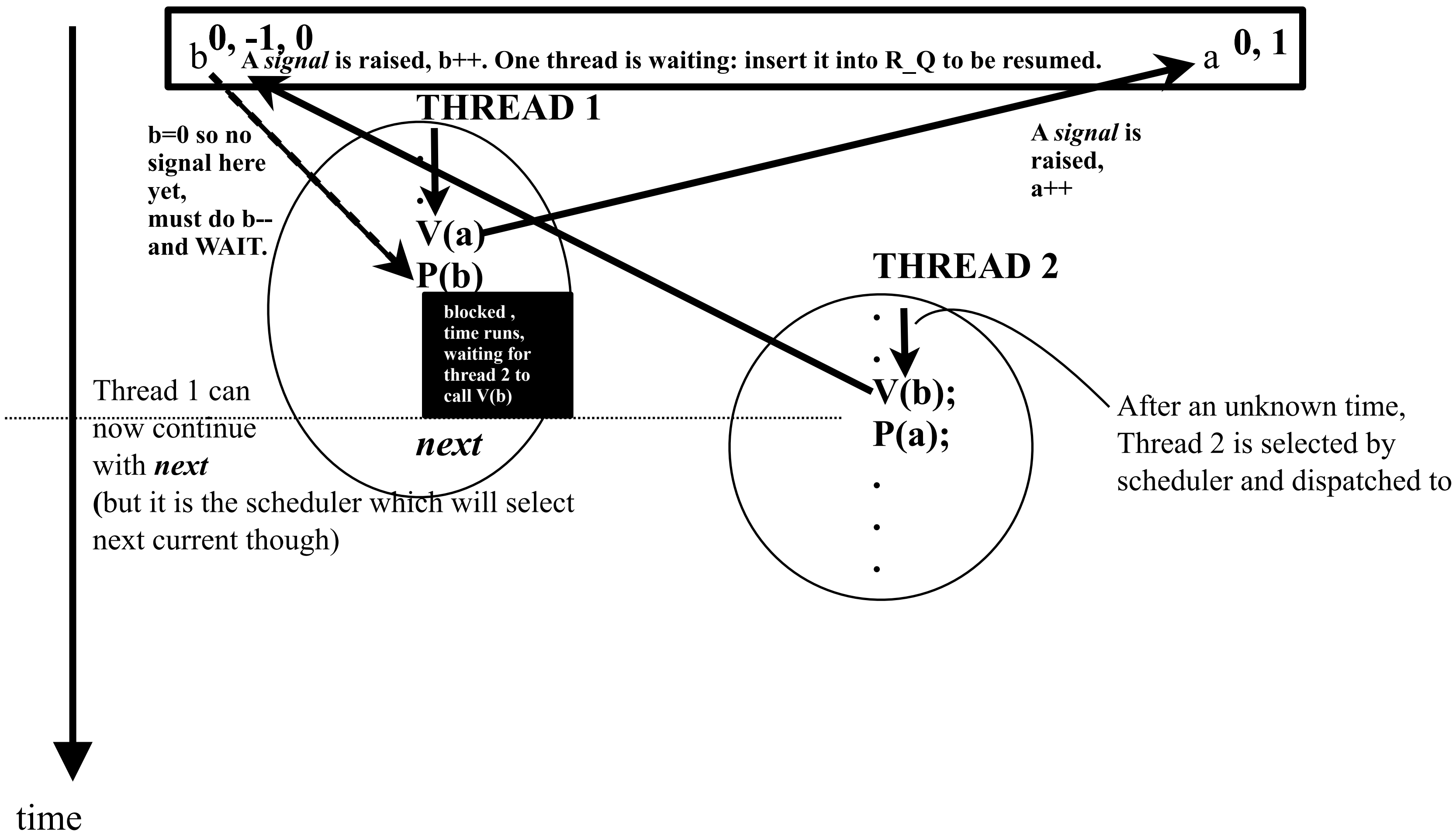


Rendezvous between two threads (or: a *Barrier* for two threads)

Initially both threads are in the Ready_Queue.

Assume that Thread 1 is scheduled to run first

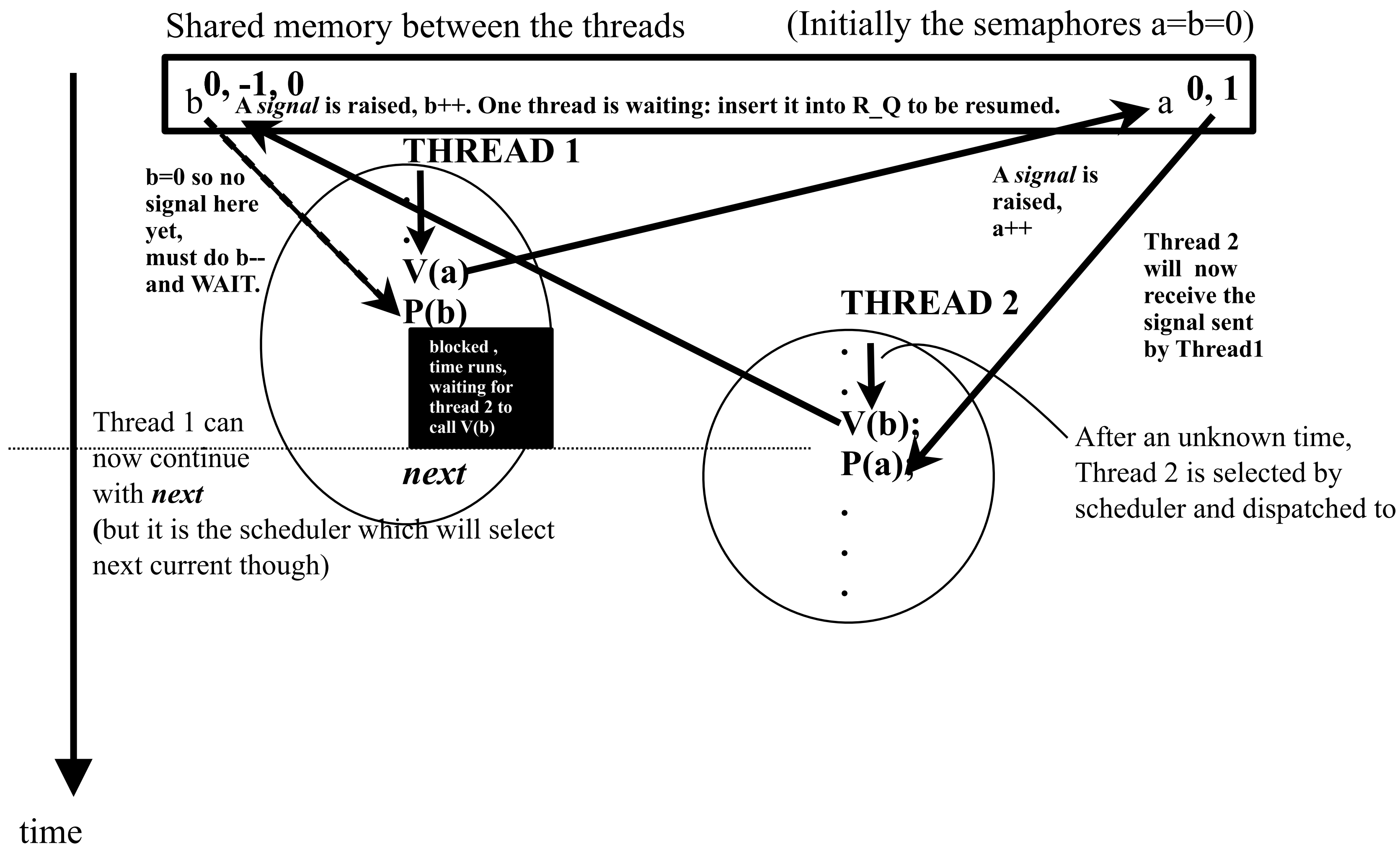
Shared memory between the threads (Initially the semaphores a=b=0)



Rendezvous between two threads (or: a *Barrier* for two threads)

Initially both threads are in the Ready_Queue.

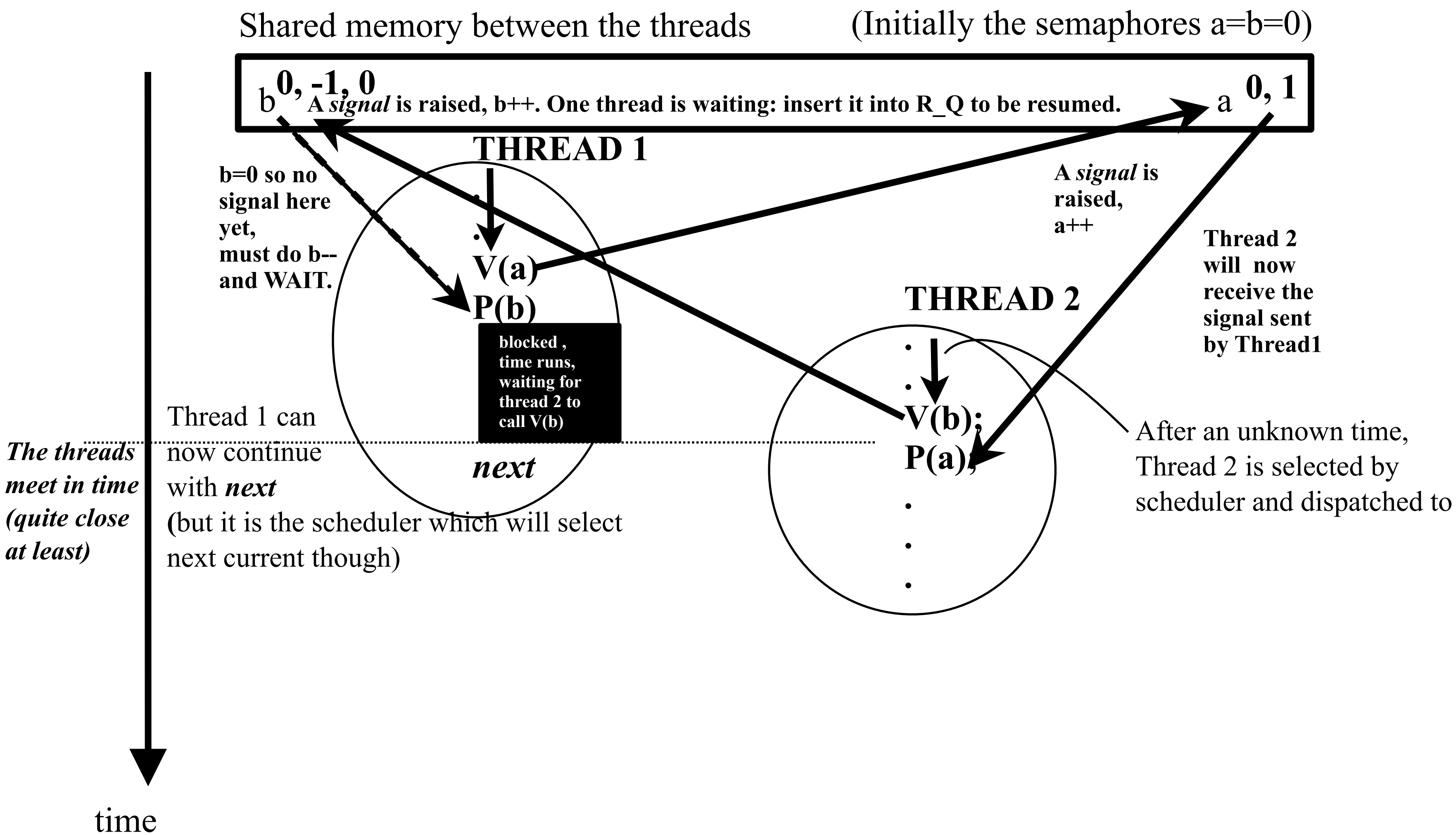
Assume that Thread 1 is scheduled to run first



Rendezvous between two threads (or: a *Barrier* for two threads)

Initially both threads are in the Ready_Queue.

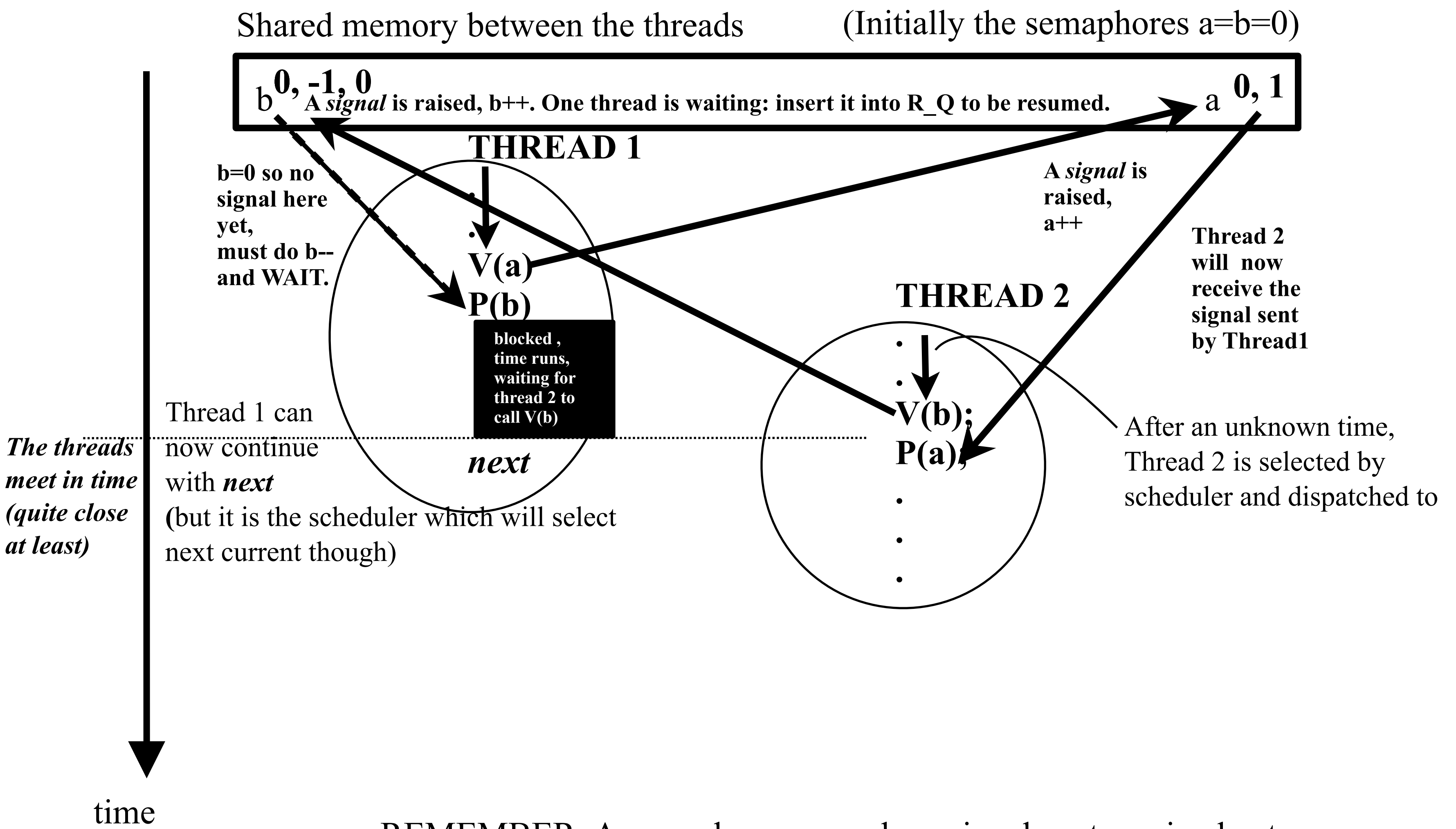
Assume that Thread 1 is scheduled to run first



Rendezvous between two threads (or: a *Barrier* for two threads)

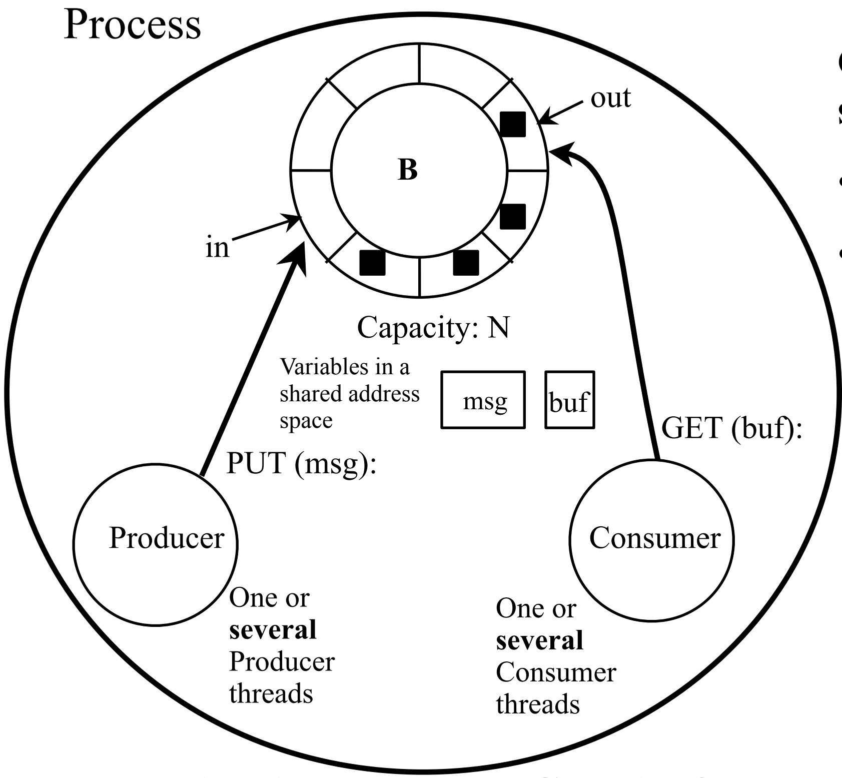
Initially both threads are in the Ready_Queue.

Assume that Thread 1 is scheduled to run first



REMEMBER: A semaphore remembers signals not received yet

Bounded Buffer using Semaphores



PUT (msg):
P(nonfull**);**
P(mutex);
<insert>
V(mutex);
V(nonempty**);**

GET (buf):
P(nonempty**);**
P(mutex);
<remove>
V(mutex);
V(nonfull**);**

Condition synchronization:

- Delay Get when empty
- Delay Put when full

Use one semaphore for *each condition* we must wait for to become TRUE:

- B empty: **nonempty**:=**0**
- B full: **nonfull**:=**N**

MUTEX:

- B and its state variables are shared between Put and Get, so should (must) have a mutex to give the threads *exclusive access* when they touch the buffer

Use one semaphore for *each shared resource* to protect it:

- B mutex: **mutex**:=**1**

- Is Mutex needed when only 1 P and 1 C?
- PUT at one end, GET at other end

Brilliant Idea

PUT (msg):

P(mutex**);**

P(nonfull**);**
<insert>
V(nonempty**);**

V(mutex**);**

GET (buf):

P(mutex**);**

P(nonempty**);**
<remove>
V(nonfull**);**

V(mutex**);**

Brilliant Idea (Not)

PUT (msg):

P(mutex**);**

P(nonfull**);**
<insert>
V(nonempty**);**

V(mutex**);**

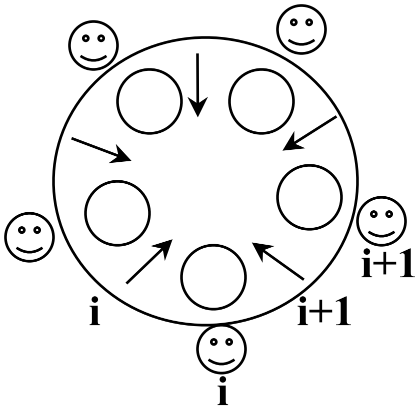
GET (buf):

P(mutex**);**

P(nonempty**);**
<remove>
V(nonfull**);**

V(mutex**);**

“Dining Philosophers”



•Each: need 2 forks to eat

•5 philosophers: 10 forks

•5 forks: 2 can eat concurrently

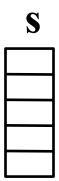
Things to observe:

•A fork can be used by one and only one at a time

•No deadlock

•No starving

•Concurrent eating



s(i): One semaphore per fork to be used in **mutex** style P-V

*{....} is while(1){...}

Mutex on whole table:

•*1 can eat at a time*

***{think;**
P(s); eat; V(s);}

T_i

Get L; Get R;

•*Deadlock possible*

S(i) = 1 initially

```
*{think;
  P(s(i));
     P(s(i+1));
        eat;
     V(s(i+1));
  V(s(i));}
```

T_i

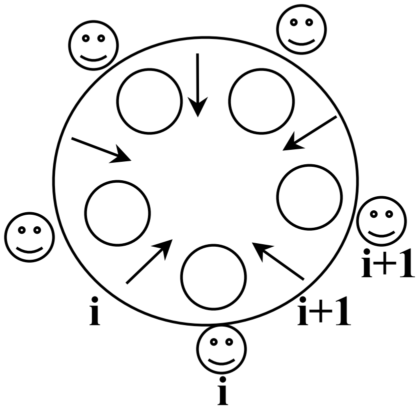
Get L; Get R if free else Put L;

•*Starvation possible*

T_i

Think about: What if we had to *clean* the forks between usage?
-where in the code?
-number of washers?

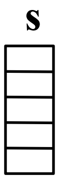
“Dining Philosophers”



- Each: need 2 forks to eat
- 5 philosophers: 10 forks
- 5 forks: 2 can eat concurrently

Things to observe:

- A fork can be used by one and only one at a time
- No deadlock
- No starving
- Concurrent eating



s(i): One semaphore per fork to be used in **mutex** style P-V

*{....} is while(1){...}

Mutex on whole table:

- *1 can eat at a time*

Initial semaphore value?

***{think;**
P(s); eat; V(s);}

T_i

Get L; Get R;

- *Deadlock possible*

S(i) = 1 initially

```
*{think;
  P(s(i));
     P(s(i+1));
       eat;
     V(s(i+1));
  V(s(i));}
```

T_i

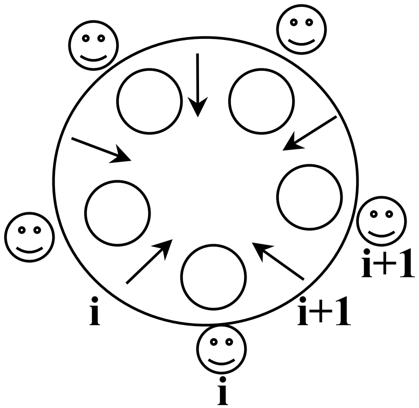
Get L; Get R if free else Put L;

- *Starvation possible*

T_i

Think about: What if we had to *clean* the forks between usage?
- -where in the code?
- -number of washers?

"Dining Philosophers"

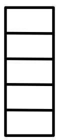


- Each: need 2 forks to eat
- 5 philosophers: 10 forks
- 5 forks: 2 can eat concurrently

Things to observe:

- A fork can be used by one and only one at a time
- No deadlock
- No starving
- Concurrent eating

s



s(i): One semaphore per fork to be used in **mutex** style P-V

*{....} is while(1){...}

Mutex on whole table:

- *1 can eat at a time*

Initial semaphore value?

```
s=1;
*{think;
  P(s); eat; V(s);}
```

T_i

Get L; Get R;

- *Deadlock possible*

S(i) = 1 initially

```
*{think;
  P(s(i));
      P(s(i+1));
        eat;
      V(s(i+1));
  V(s(i));}
```

T_i

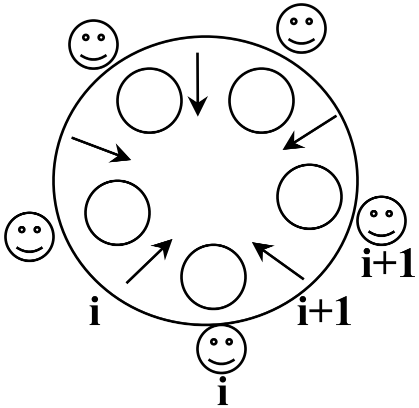
Get L; Get R if free else Put L;

- *Starvation possible*

T_i

Think about: What if we had to *clean* the forks between usage?
- -where in the code?
- -number of washers?

Dining Philosophers



To avoid starvation they could look after each other:

•**Entry**: If L and R is not eating I can

•**Exit**: If L (R) wants to eat and L.L (R.R) is not eating I start him eating

One semaphore per philosopher
Used in **signal** style

s **state**

•Thinking

•Eating

•Wanting

S(i) = 0 initially

$\mathbf{T_i}$

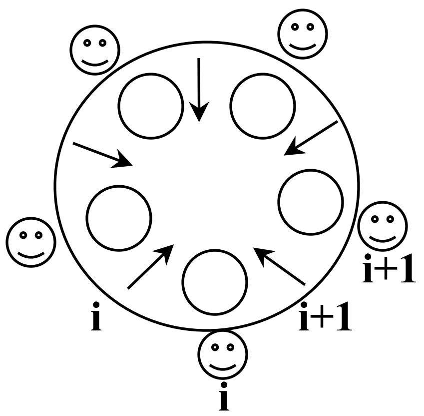
```
*{
   think;
   ENTRY;
      eat;
   EXIT;
}
```

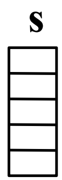
```
P(mutex);
   state(i):=Wanting;
   if (state(i-1) !=Eating AND state(i+1) != Eating)
   {/*Safe to eat*/
      state(i):=Eating;
      V(s(i));   /*Because , so I signal myself so I don't block at P below*/ }
V(mutex);
P(s(i)); /*Init was 0!! I or right (left) neighbor may have said V(i) to me!*/
                                                              What if NOT?
```

```
P(mutex);
   state(i):=Thinking;
   if (state(i-1)=Wanting AND state(i-2) !=Eating)
   {
      state(i-1):=Eating;
      V(s(i-1));  /*Start Left neighbor*/
   }
/*Analogue for Right neighbor*/
V(mutex);
```

Trouble: **starvation** pattern possible:
2&4 at table, 1&3 hungry
2 gets up, 1 sits down
4 gets up, 3 sits down
3 gets up, 4 sits down
1 gets up, 2 sits down
Ad infinitum => Phil 0 will starve

Dining Philosophers





S(i) = 1 initially

Can we in a simple way do better than this one?

Get L; Get R;

- Deadlock possible

```
P(s(i));
   P(s(i+1));
      eat;
   V(s(i+1));
V(s(i));
```

- **Remove the danger of circular waiting (deadlock)**
- T1-T4: Get L; Get R;
- **T5: Get R; Get L;**

T_1, T_2, T_3, T_4:

```
P(s(i)):
   P(s(i+1));
      <eat>
   V(s(i+1));
V(s(i));
```

T_5

```
P(s(1));
   P(s(5));
      <eat>
   V(s(5));
V(s((1));
```

- **Non-symmetric solution. Still quite elegant**