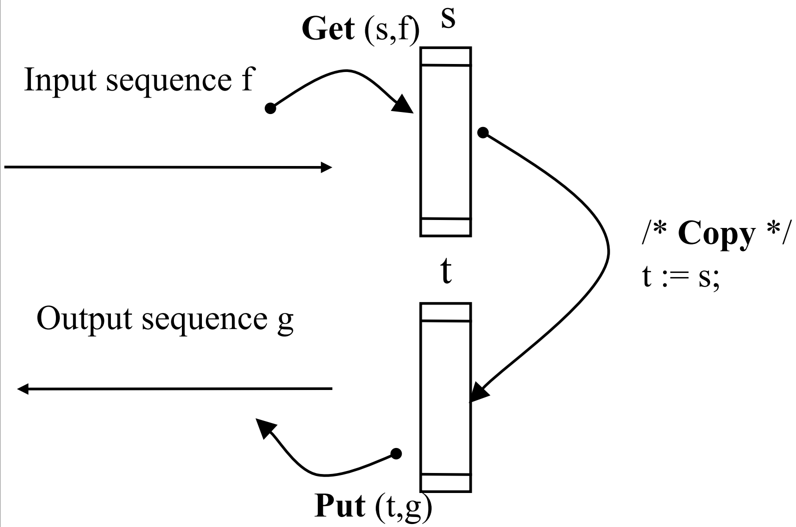
Semaphores

Otto J. Anshus
University of {Tromsø, Oslo}

Concurrency: Double buffering

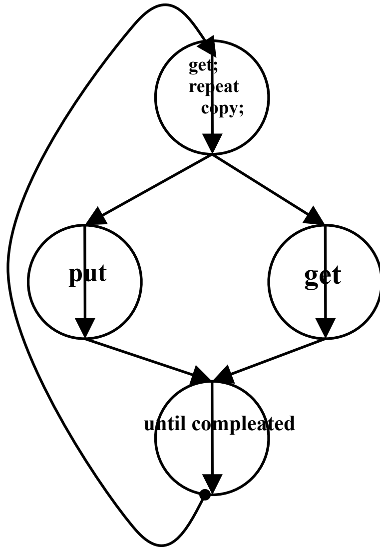
/* **Fill** s and **empty** t ***concurrently*** */



cobegin & *coend* specifies concurrent execution.

```
Get(s,f);
Repeat
    Copy;
    cobegin
        Put(t,g);
        Get(s,f);
    coend;
until completed;
```

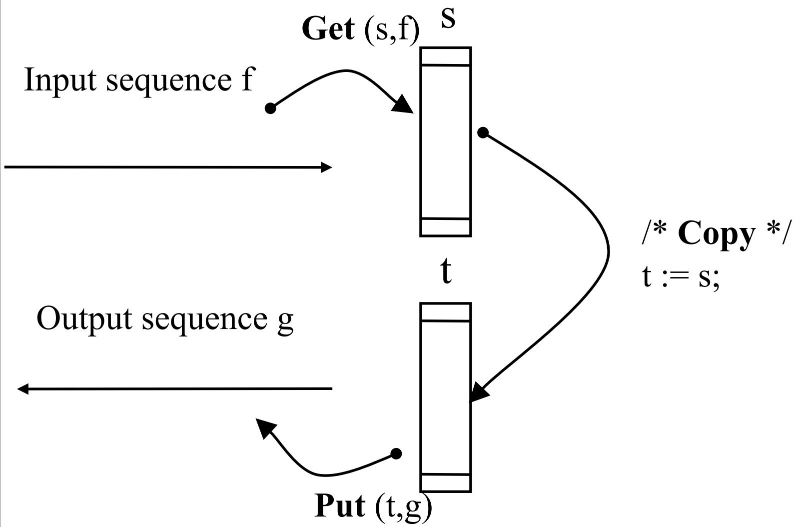
(Two threads)
Alternative syntax:
{put() || get()}



- **Put and Get** are disjoint
  - but not with regards to **Copy**
    - The **order** of Copy vs. Put & Get:
      - OK, defined by program

Concurrency: Double buffering

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



Get(s,f);

Repeat

 Copy;

 cobegin

  Put(t,g);

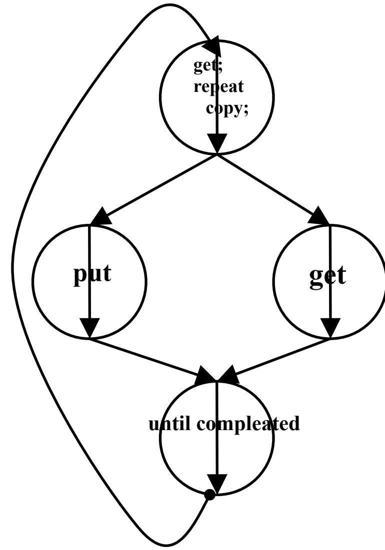
  Get(s,f);

 coend;

until completed;

cobegin & *coend* specifies concurrent execution.

(Two threads)
Alternative syntax:
{put() || get()}

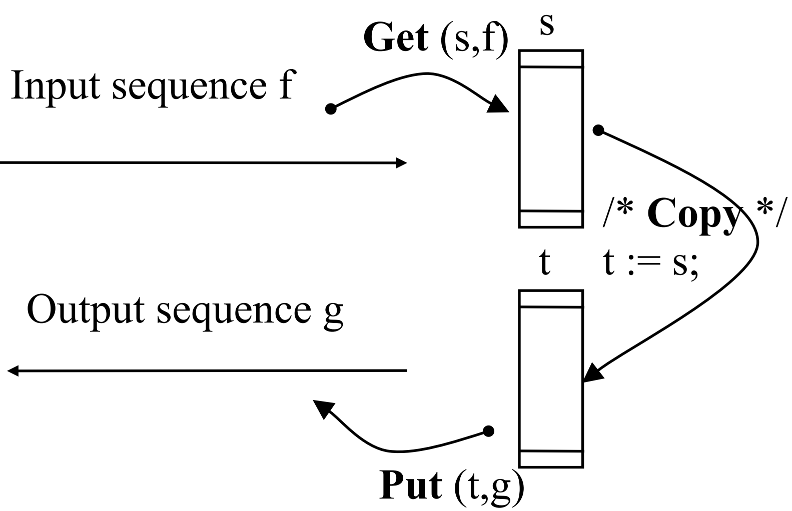


- **Put and Get** are disjoint
  - but not with regards to **Copy**
    - The **order** of Copy vs. Put & Get:
      - OK, defined by program

Think about non-preemptive vs. preemptive scheduling by OS

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_thread || **get**_thread || **copy**_thread} /*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?* **2**, one for each shared resource.

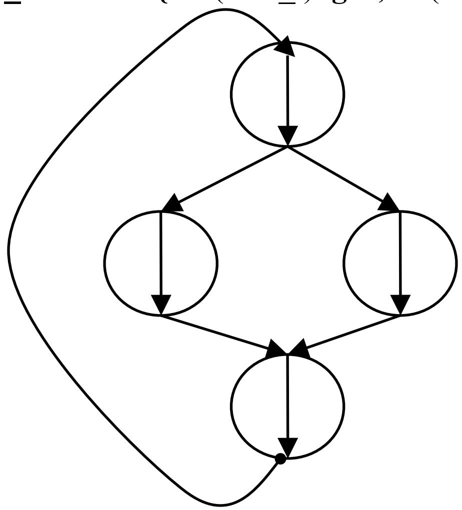
Proposed code (but** not **quite good enough):

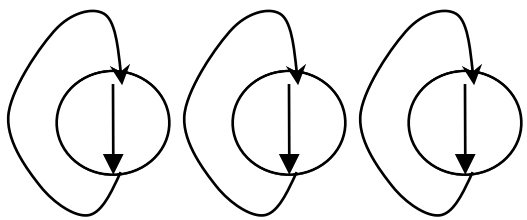
copy_thread:: *{acq(lock_t); acq(lock_s); **t=f;** rel(lock_s); rel(lock_t);}

get_thread:: *{ack(lock_s); **s=f;** rel(lock_s);}

put_thread:: *{ack(lock_t): **g=t;** rel(lock_t);}

- **Not too bad, but NO ORDER**
  - what can happen?
    - same/old **s** values copied again
    - **s** values never copied because Get overwrites
    - same/old **t** values read by Put
    - **t** values lost because Copy overwrites





Threads specifies concurrent execution

Protecting a Shared Variable

- 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, that is the point)
    - (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: they can be individually scheduled
- ***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 *schedule* and *dispatch* to some other **thread** in same address space or even to another **process**)
          - **(5b) lock open: close lock_A (**and *schedule* and *dispatch* to library routine (or even to another thread or process)
    - User level: **(6) execute count++**
  - **(7) Release(lock) system call**

Lock Performance and Cost Issues

- Implement the lock-mechanism 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 *relative*)
  - *Un-contended* (**Low** contention lock):
    - Spinning: **Best** (fastest in, few cpu cycles wasted)
    - Blocking: **Bad** (fast in, overhead cpu cycles wasted)

Use of locks when implementing

Block/unblock syscalls

(implemented by the OS Kernel)

- **What we want to achieve**

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

  – **Block** thread on a queue called waitq
    - **insert** (waitq, last, **remove** (readyq, current))

  – **Unblock**
    - **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 (lock)
  - Spin until lock is open %*Why*?
    - Save context to the TCB
    - Enqueue the TCB
  - Open lock
  - goto scheduler

- UnBlock (lock)
  - Spin until lock is open
    - -Dequeue first TCB
    - -Put TCB into ready_queue
  - Open lock
  - goto scheduler

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

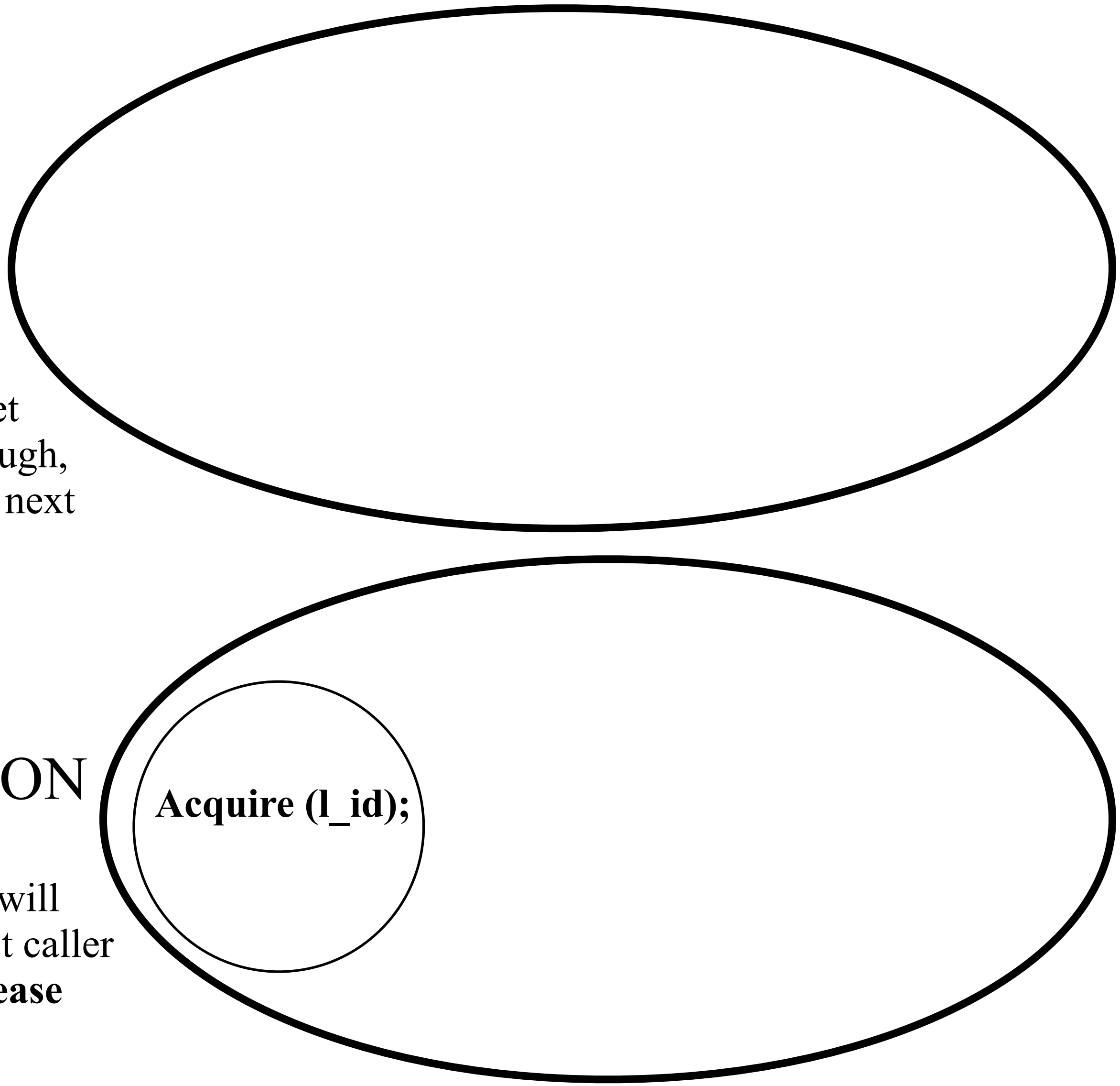
Threads inside one process: Shared address space. They can access the same variables

Process w/two threads

MUTEX

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

CONDITION SYNCHRONIZATION

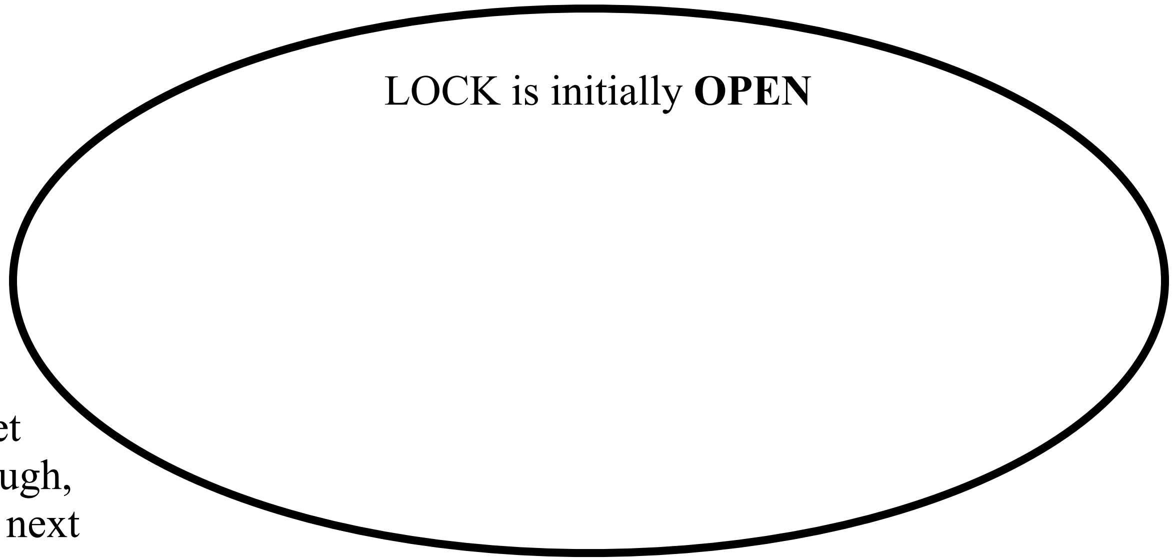


Acquire will block first caller until **Release**

Two Styles of Synchronization

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

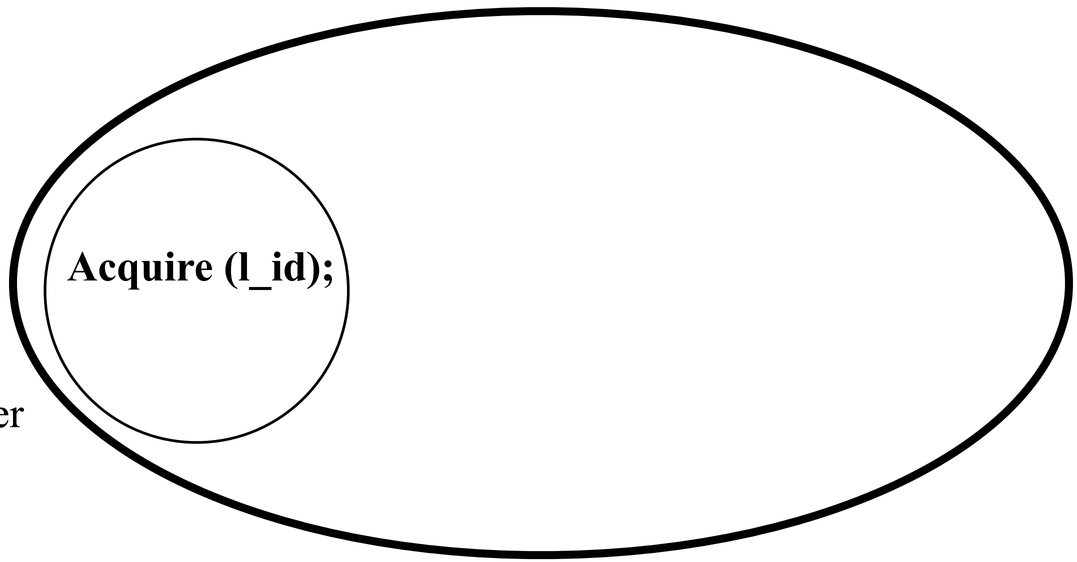
Process w/two threads



MUTEX

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

CONDITION SYNCHRONIZATION



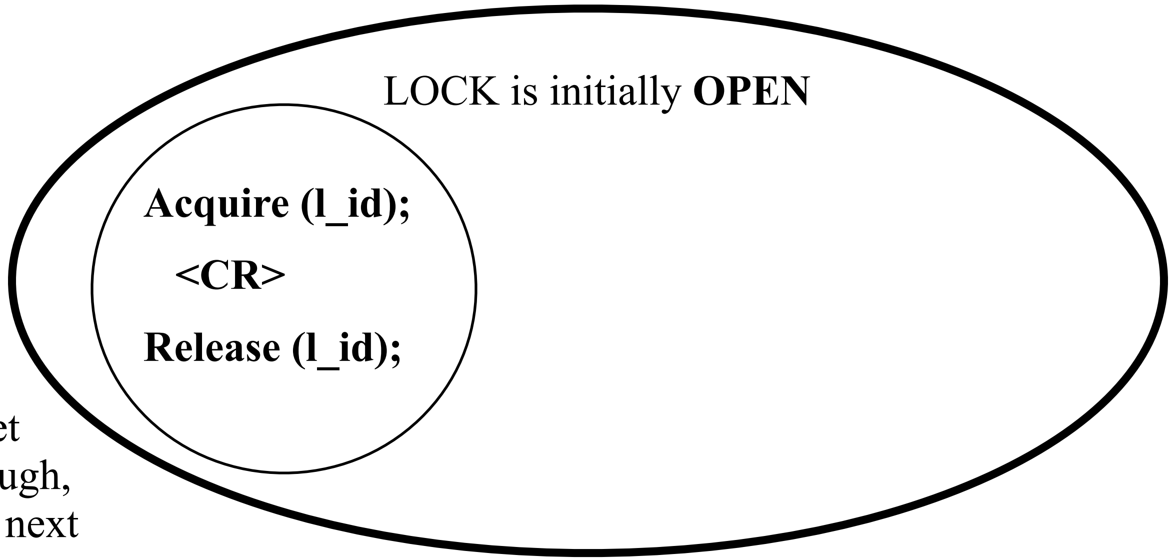
Acquire will block first caller until **Release**

Two Styles of Synchronization

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

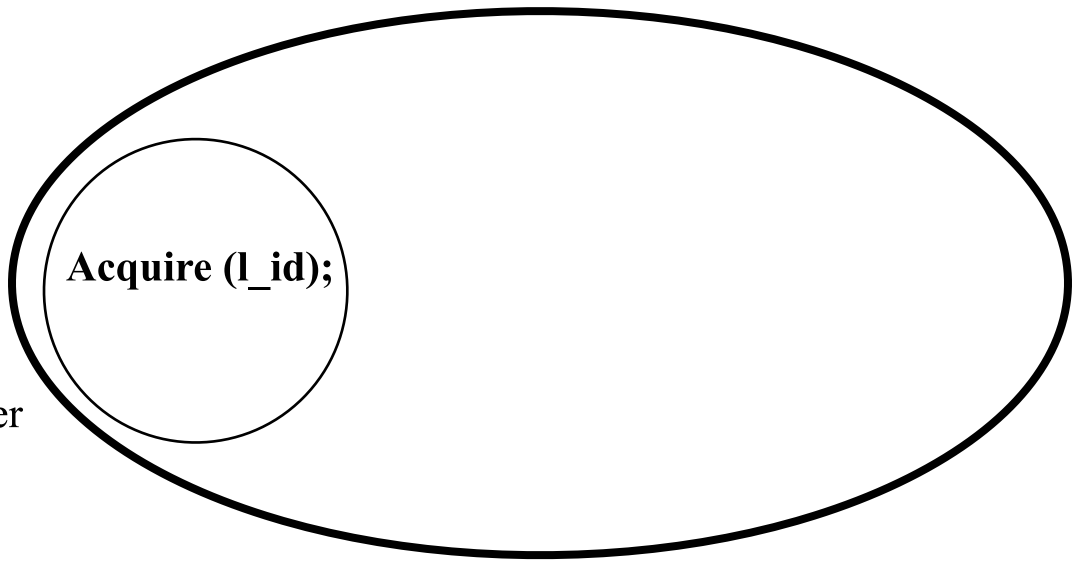
Process w/two threads

MUTEX



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

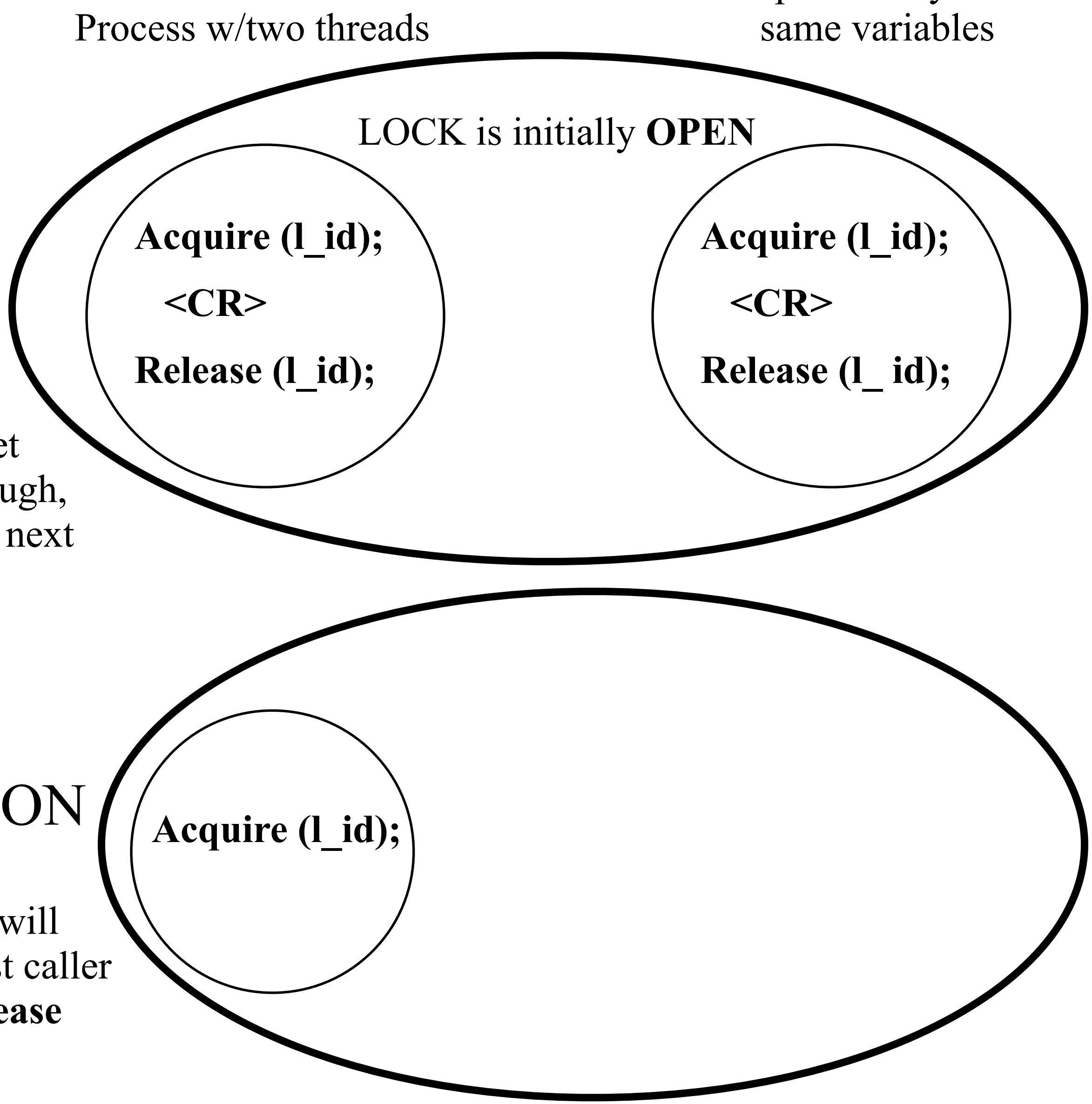
CONDITION SYNCHRONIZATION



Acquire will block first caller until **Release**

Two Styles of Synchronization

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



MUTEX

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

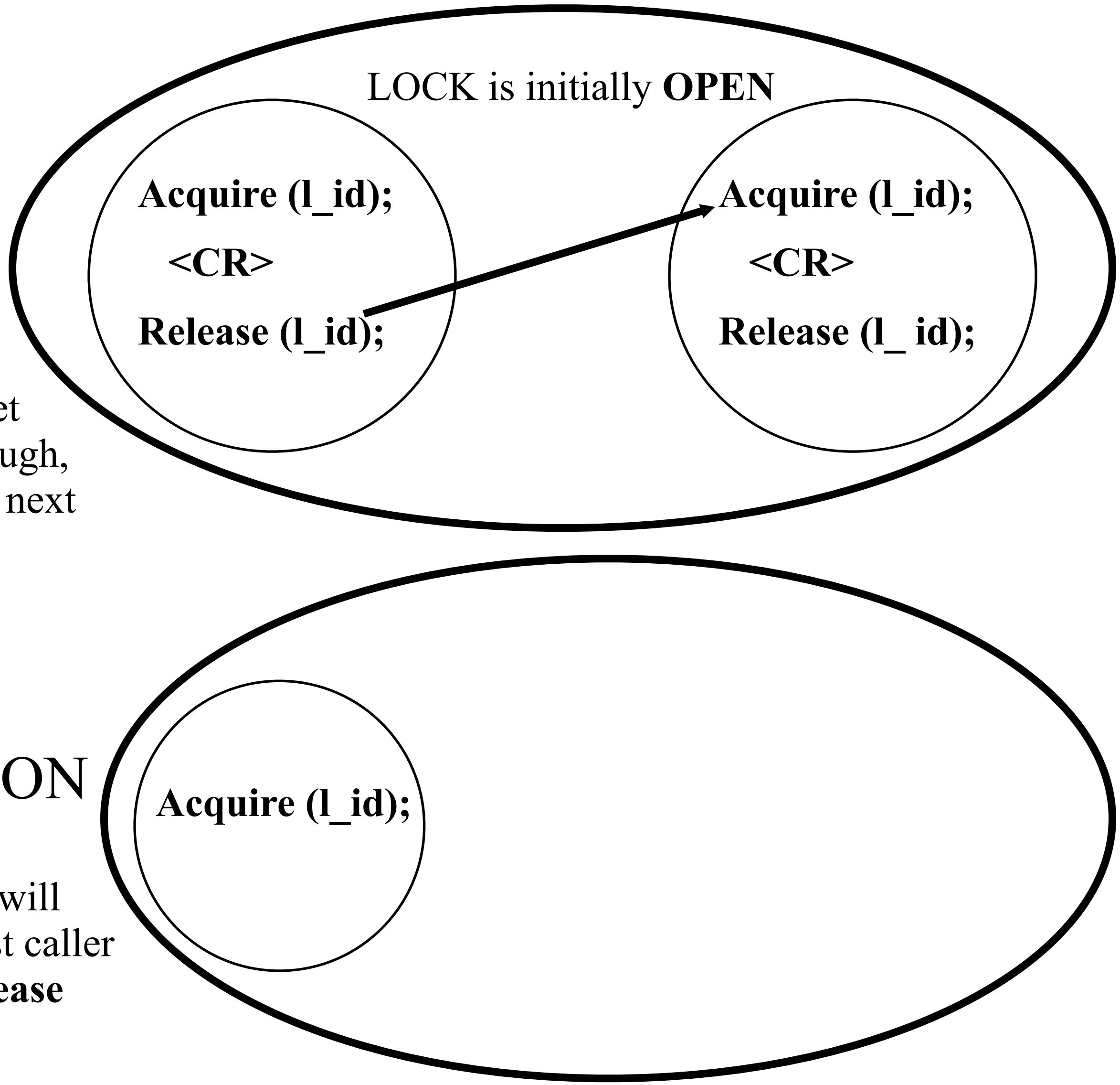
CONDITION SYNCHRONIZATION

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**

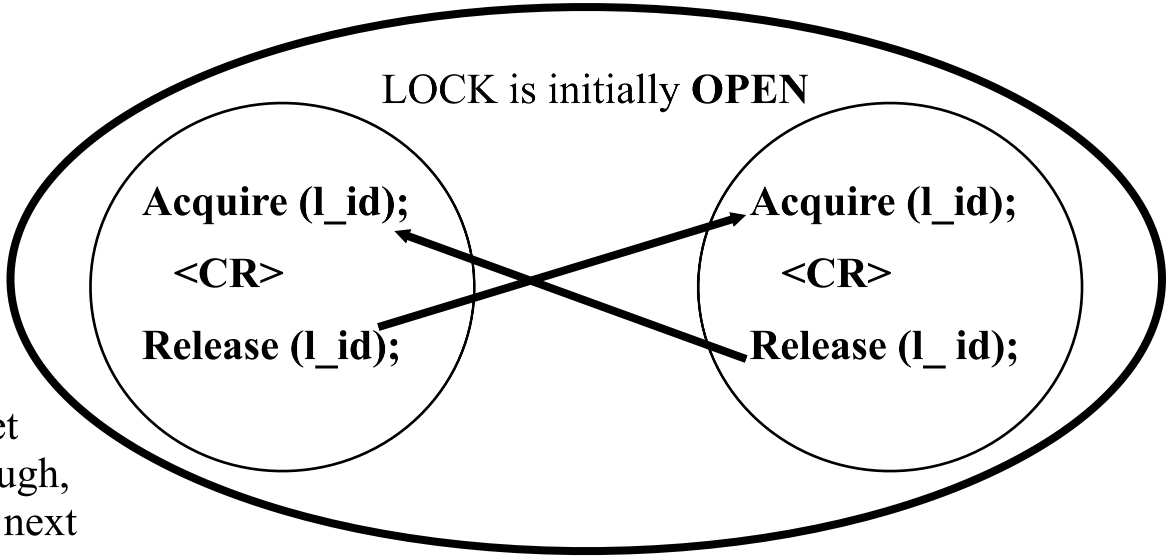
CONDITION SYNCHRONIZATION

Acquire will block first caller until **Release**

Two Styles of Synchronization

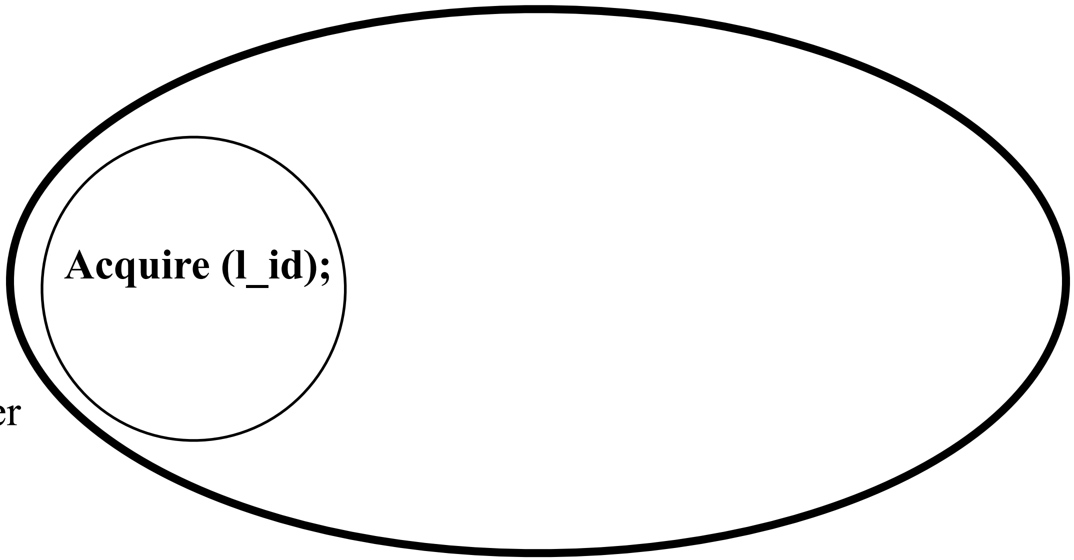
Threads inside one process: Shared address space. They can access the same variables

Process w/two threads



MUTEX

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



CONDITION SYNCHRONIZATION

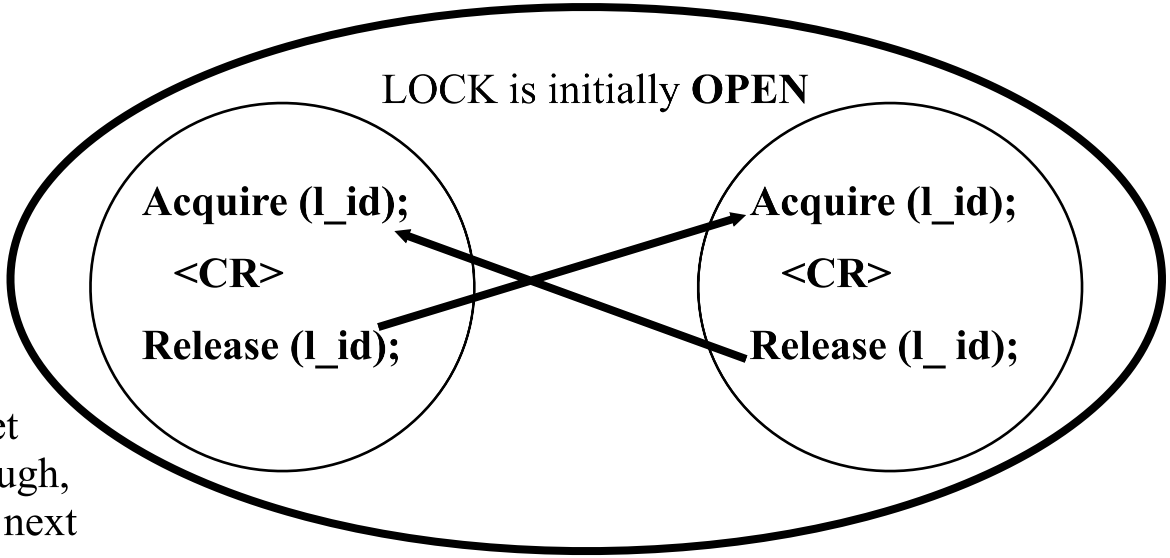
Acquire will block first caller until **Release**

Two Styles of Synchronization

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

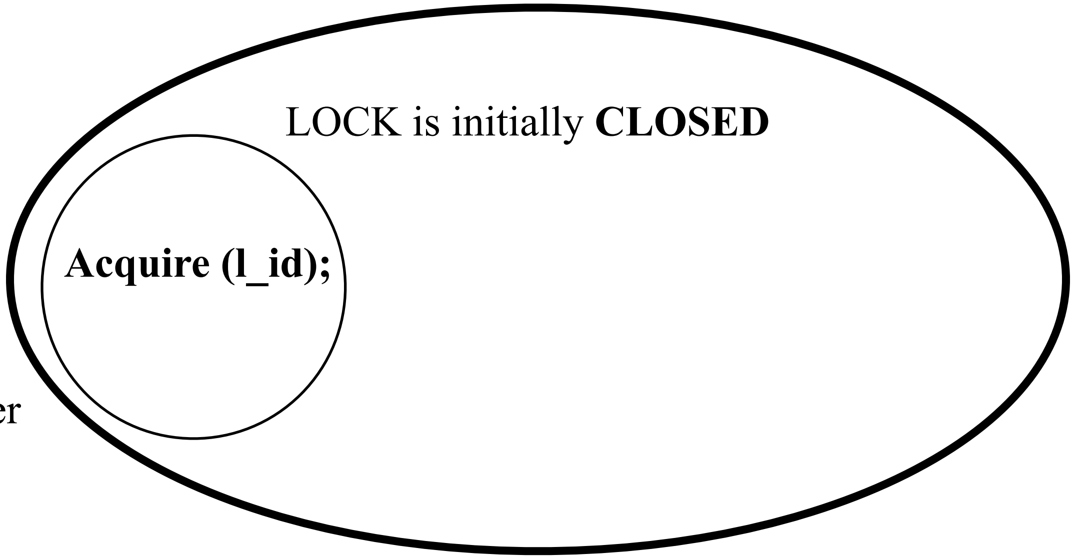
Process w/two threads

MUTEX



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

CONDITION SYNCHRONIZATION



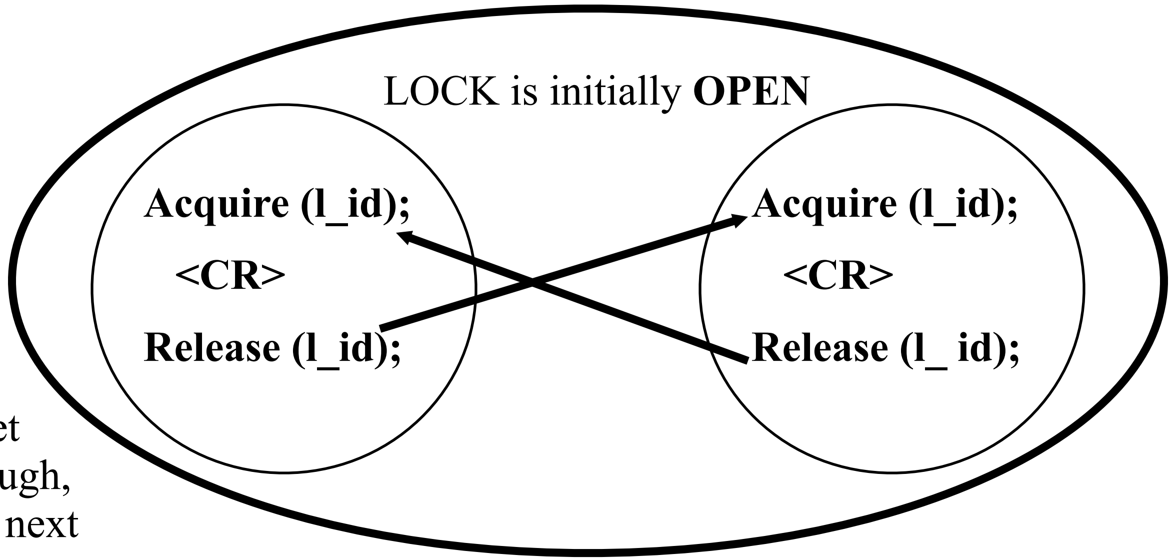
Acquire will block first caller until **Release**

Two Styles of Synchronization

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

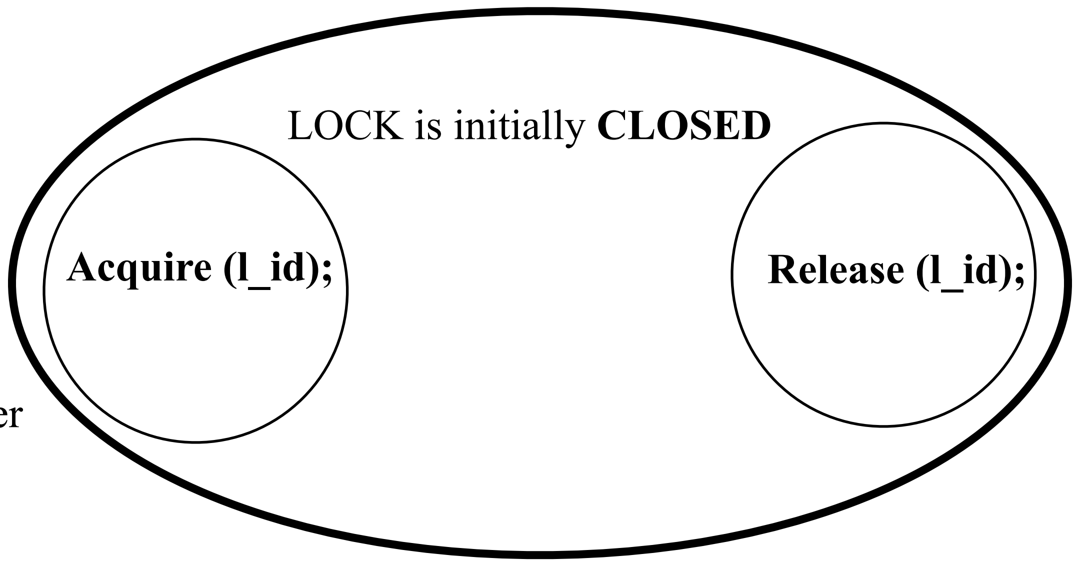
Process w/two threads

MUTEX



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

CONDITION SYNCHRONIZATION



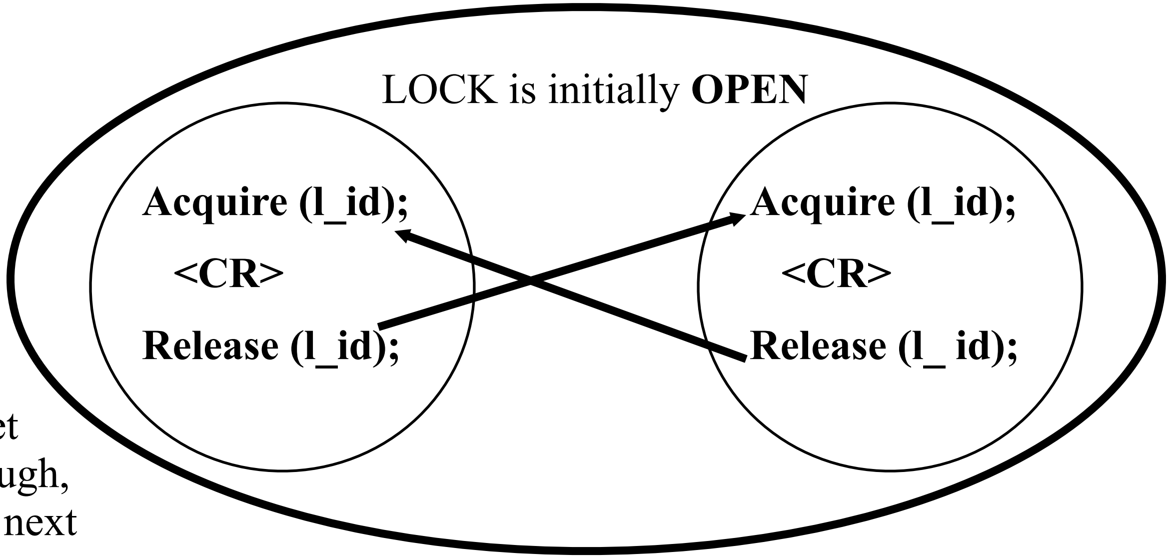
Acquire will block first caller until **Release**

Two Styles of Synchronization

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

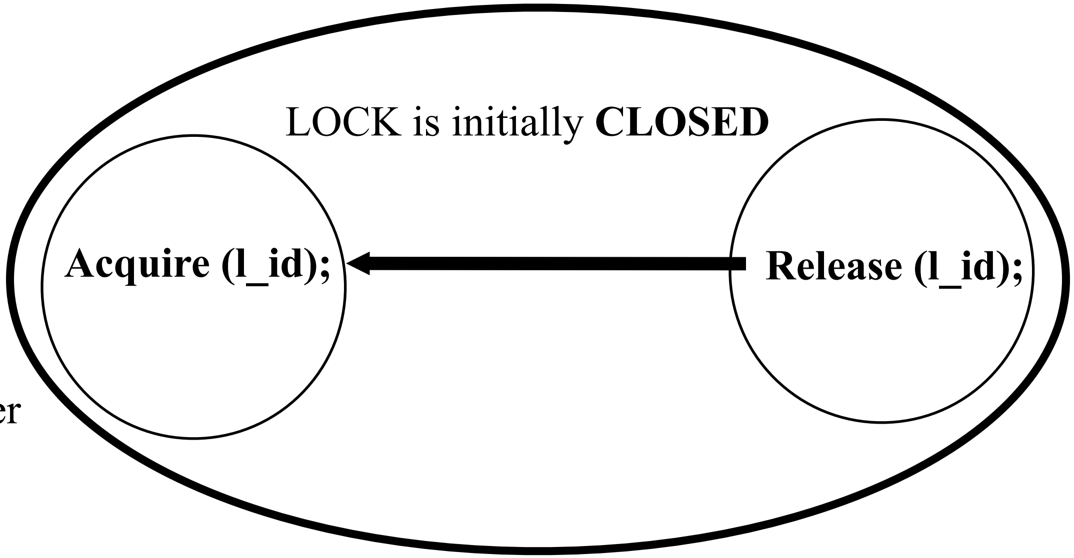
Process w/two threads

MUTEX



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

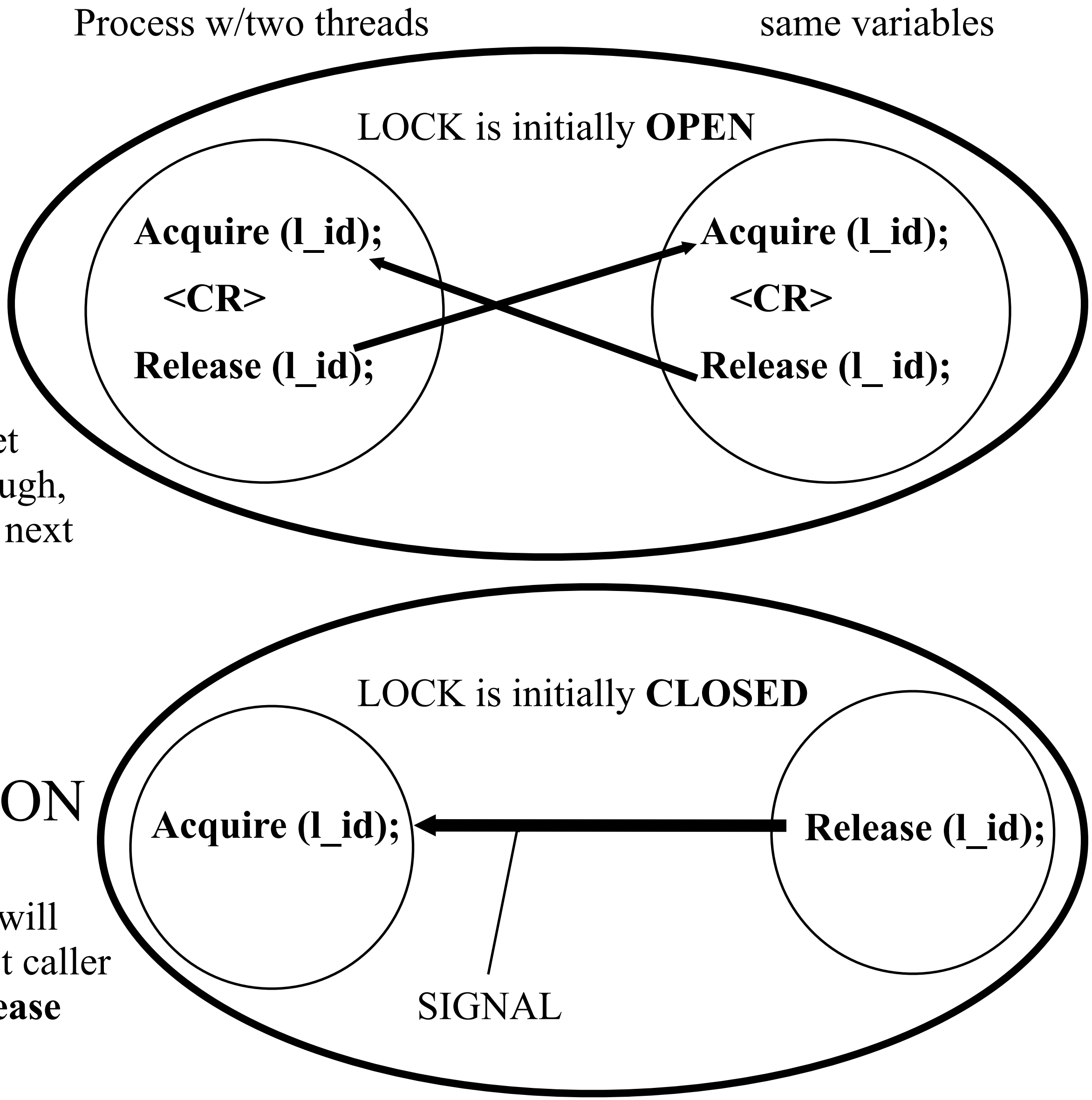
CONDITION SYNCHRONIZATION



Acquire will block first caller until **Release**

Two Styles of Synchronization

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



MUTEX

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

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

- **Down(s)** a.k.a **Wait(s)** a.k.a **P(s)**
  - itself a critical region: MUTEX
  - DELAY (block, or busy wait) if not positive (s<1)
  - Decrement semaphore value by 1
- **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

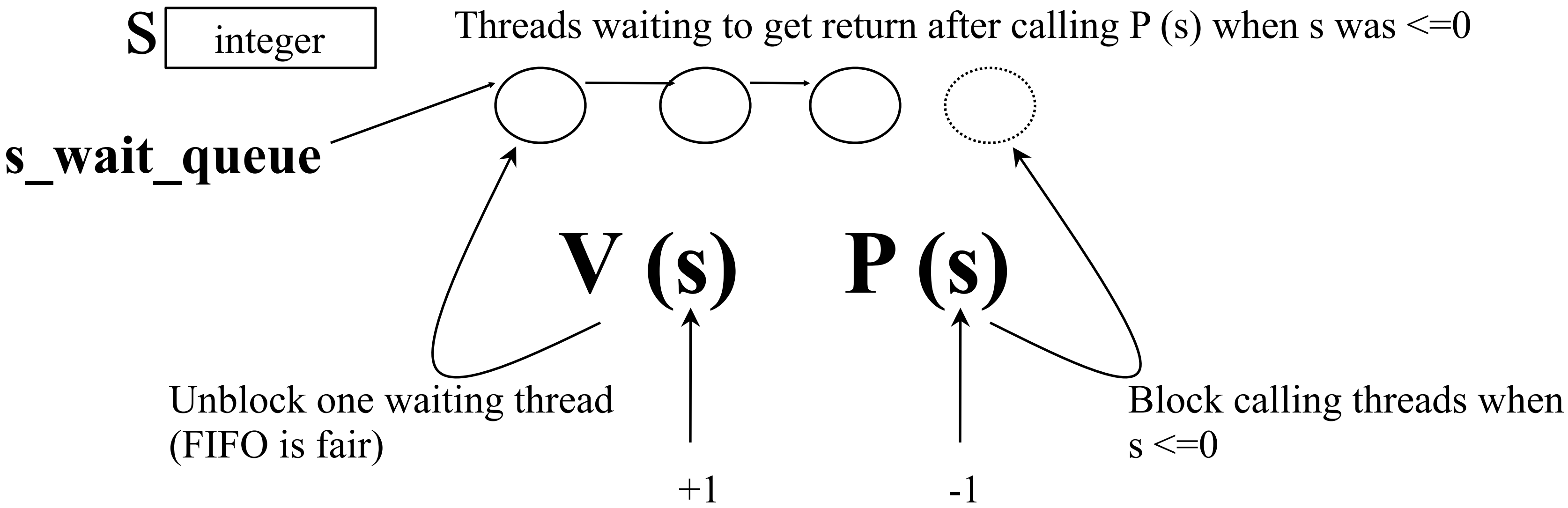
Can get **negative** s: counts number of waiting threads

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

Dutch words

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

A Blocking Semaphore Implementation



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

  So what?

- Approaches to achieve atomicity

  Disable interrupts

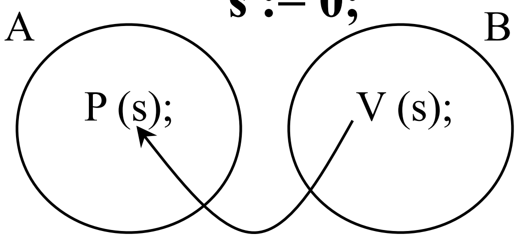
  P() and V() as System calls

  Entry-Exit protocols

Using Semaphores

"The Signal"

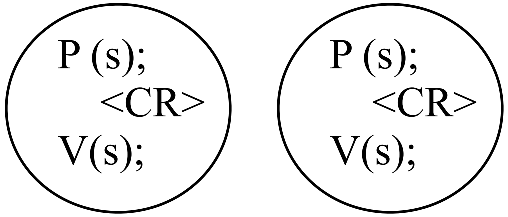
s := 0;



A is delayed until B says V

"The Mutex"

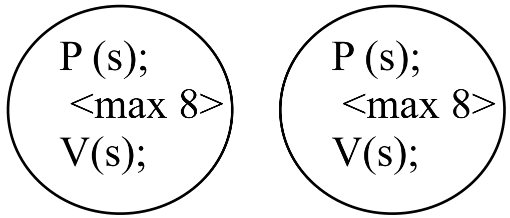
s := 1;



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

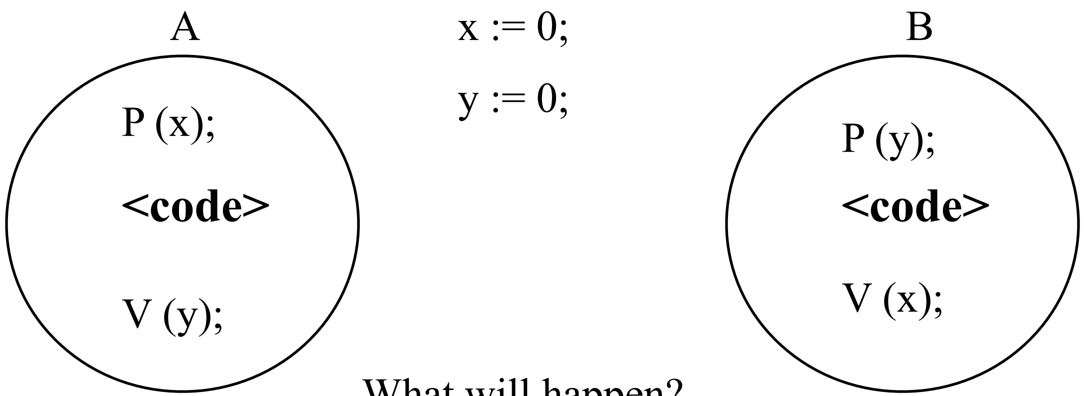
NB: remember to set the initial semaphore value!

s := 8;



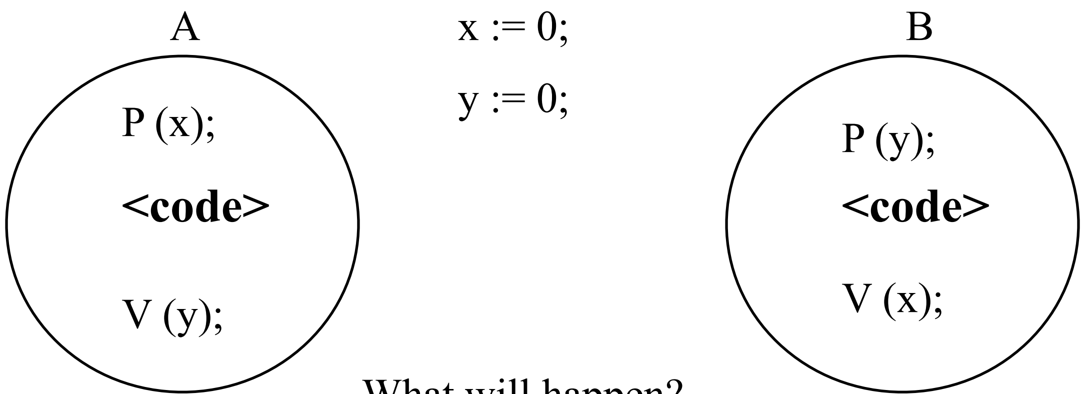
Up to 8 threads can pass P, the ninth will block until V is said by one of the eight already in there

Simple to debug?



What will happen?

Simple to debug?



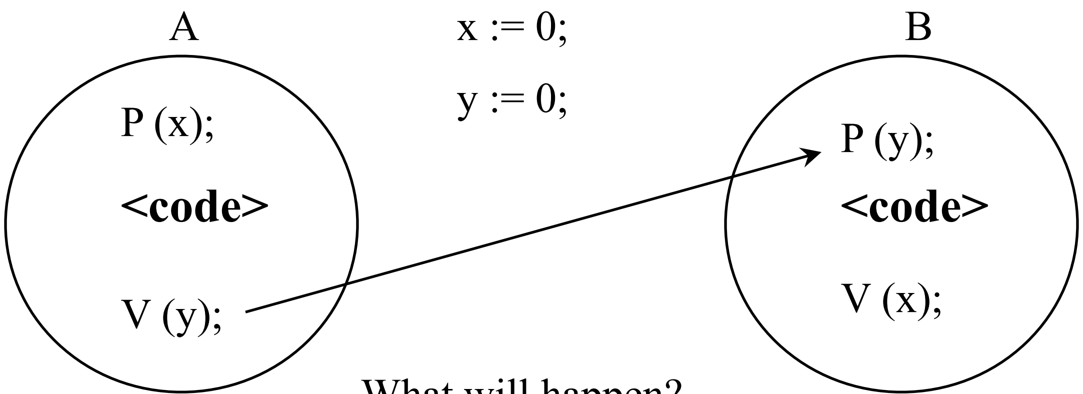
What will happen?

THEY ARE FOREVER WAITING FOR EACH OTHERS SIGNAL

Circular Wait

Classic (but not good) situation resulting in a *Deadlock*

Simple to debug?



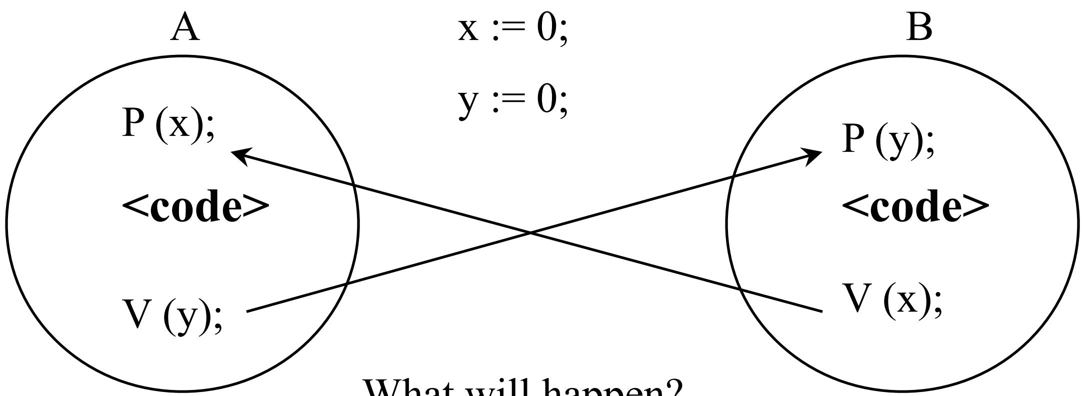
What will happen?

THEY ARE FOREVER WAITING FOR EACH OTHERS SIGNAL

Circular Wait

Classic (but not good) situation resulting in a *Deadlock*

Simple to debug?



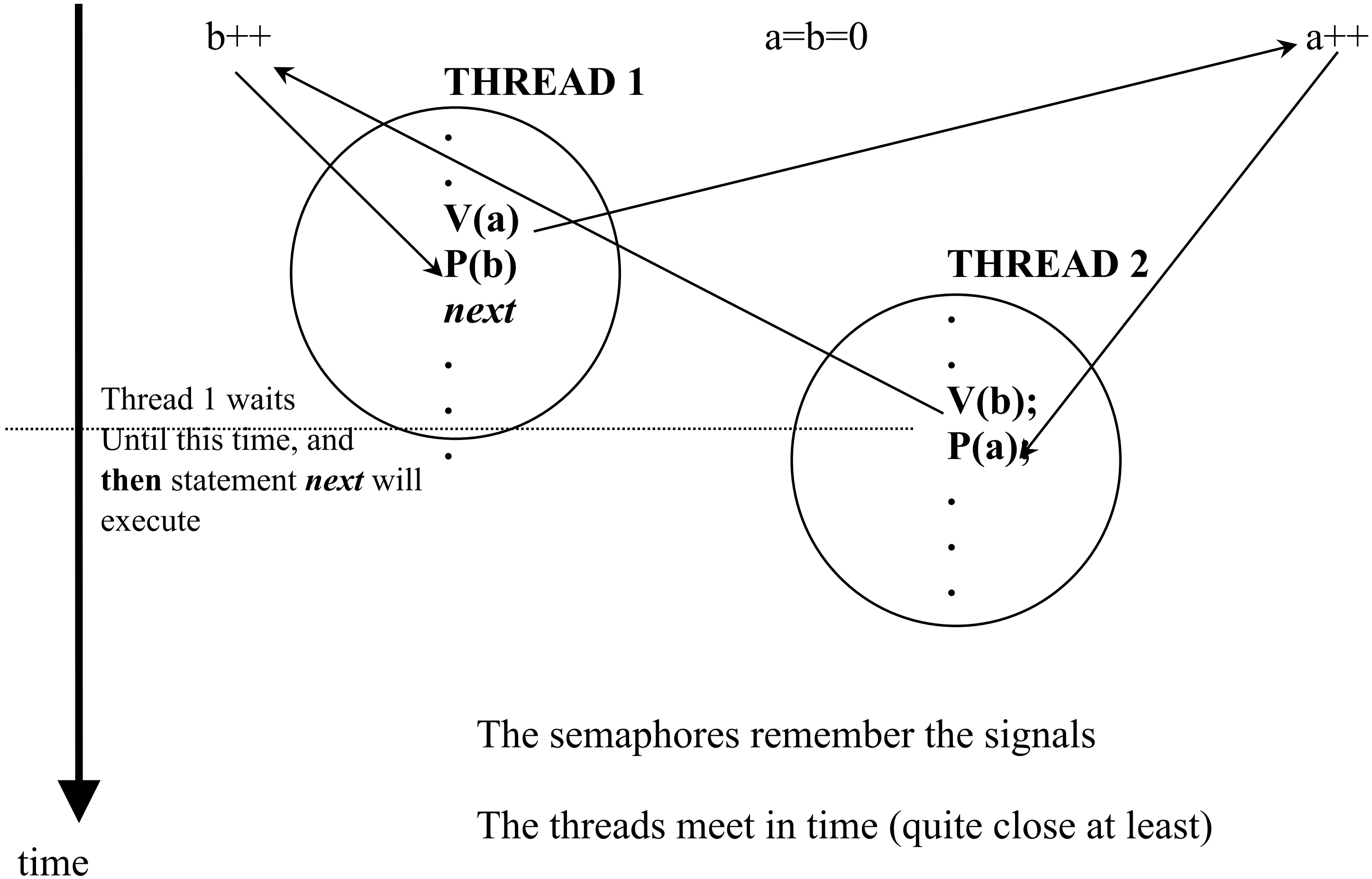
What will happen?

THEY ARE FOREVER WAITING FOR EACH OTHERS SIGNAL

Circular Wait

Classic (but not good) situation resulting in a *Deadlock*

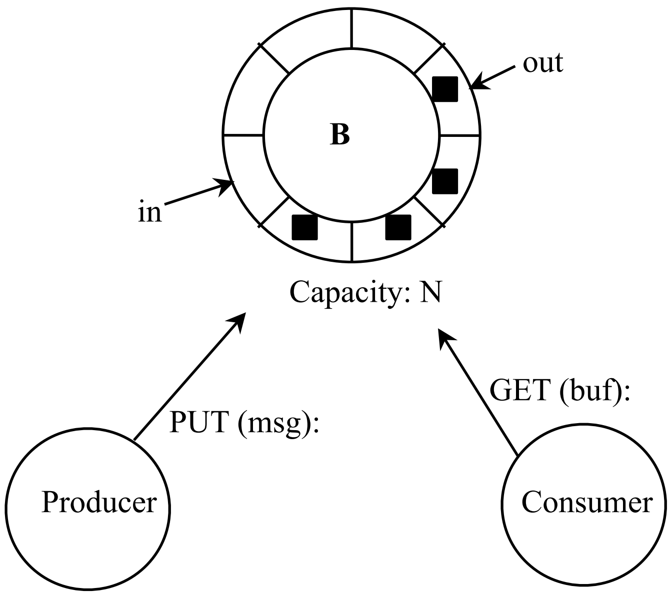
Rendezvous between two threads (or: a *Barrier* for two threads)



The semaphores remember the signals

The threads meet in time (quite close at least)

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:

•No Get when empty

•No 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 shared, so must have mutex between Put and Get

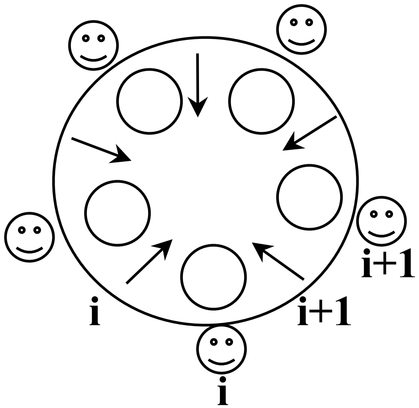
Use one semaphore for *each shared resource* toproytect it from i:

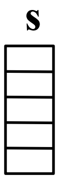
•B mutex: **mutex**:=**1**

•Is Mutex needed when only 1 P and 1 C?

•PUT at one end, GET at other end

“Dining Philosophers”





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

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

Things to observe:

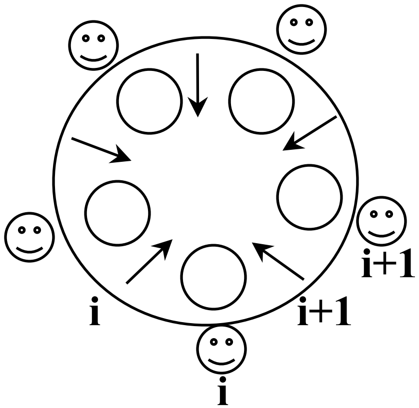
- A fork can only be used by one at a time, please
- No deadlock, please
- No starving, please
- Concurrent eating, please

$\mathbf{T_i}$

$\mathbf{T_i}$

$\mathbf{T_i}$

“Dining Philosophers”



•Each: need 2 forks to eat

•5 philosophers: 10 forks

•5 forks: 2 can eat concurrently

Things to observe:

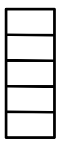
•A fork can only be used by one at a time, please

•No deadlock, please

•No starving, please

•Concurrent eating, please

s



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

Mutex on whole table:

•*1 can eat at a time*

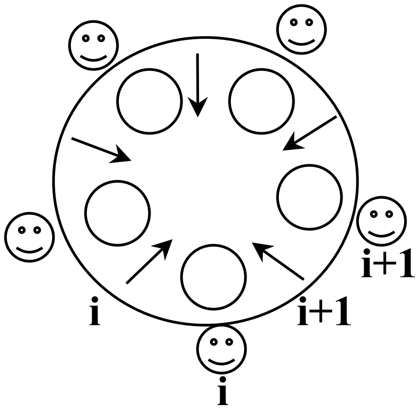
P(mutex);
eat;
V(mutex);

T_i

T_i

T_i

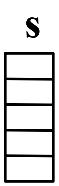
“Dining Philosophers”



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

Things to observe:

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



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

Mutex on whole table:

- *1 can eat at a time*

```
P(mutex);
  eat;
V(mutex);
```

T_i

Get L; Get R;

- *Deadlock possible*

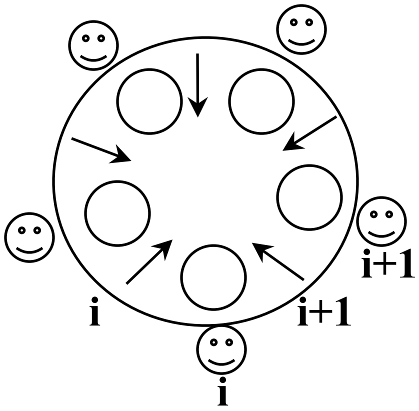
S(i) = 1 initially

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

T_i

T_i

"Dining Philosophers"



•Each: need 2 forks to eat

•5 philosophers: 10 forks

•5 forks: 2 can eat concurrently

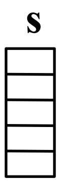
Things to observe:

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

•No deadlock, please

•No starving, please

•Concurrent eating, please



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

Mutex on whole table:

•*1 can eat at a time*

```
P(mutex);
  eat;
V(mutex);
```

T_i

Get L; Get R;

•*Deadlock possible*

S(i) = 1 initially

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

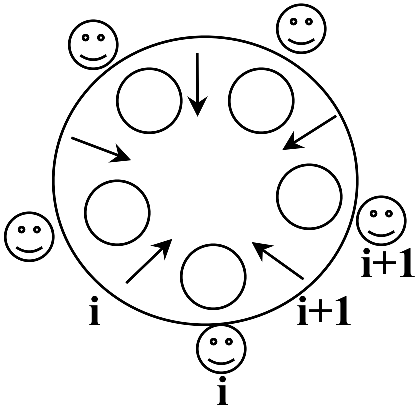
T_i

Get L; Get R if free else Put L;

•*Starvation possible*

T_i

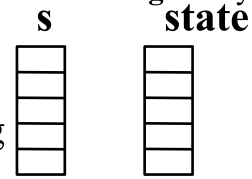
Dining Philosophers



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



- Thinking
- Eating
- Want

S(i) = 0 initially

T_i

```
While (1) {
   <think>
   ENTRY;
      <eat>
   EXIT;
}
```

ENTRY:

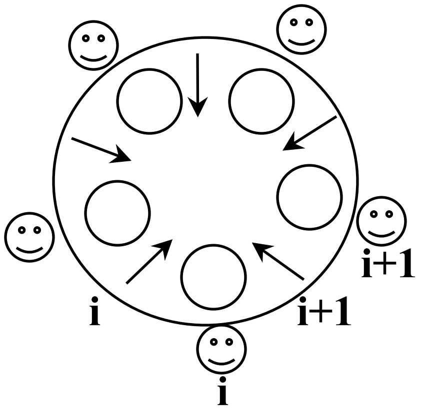
```
P(mutex);
   state(i):=Want;
   if (state(i-1) !=Eating AND state(i+1) != Eating)
   {/*Safe to eat*/
      state(i):=Eating;
      V(s(i));   /*Because  */    }
V(mutex);
P(s(i)); /*Init was 0!! I or right (left) neighbor may have said V(i) to me!*/
```

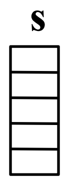
EXIT:

```
P(mutex);
   state(i):=Thinking;
   if (state(i-1)=Want 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**

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);
}
```

Trouble

Lost V calls

- 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)

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

Example: Condition Synchronization between Interrupt Handler and Device Driver

- A device thread and the interrupt handler
  - need to handle shared data between them

semaphore s; s=0;