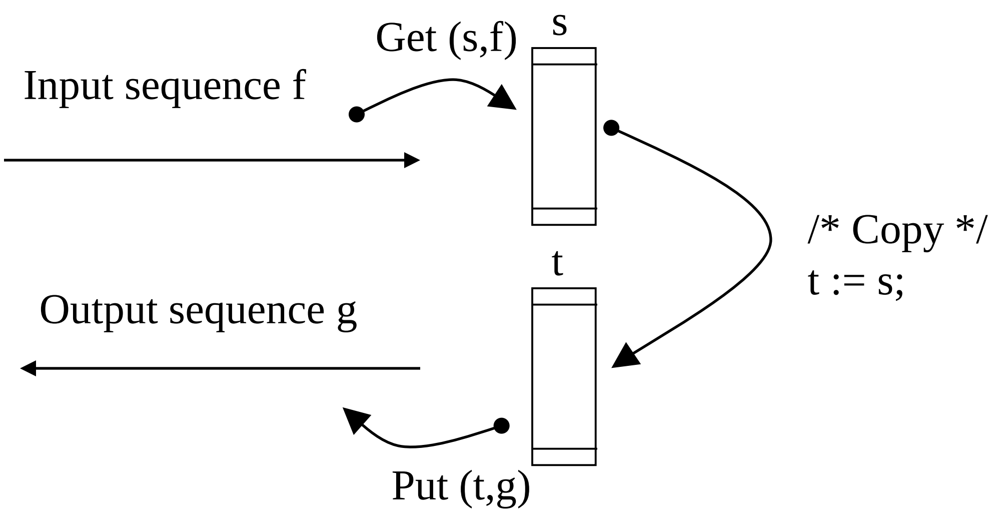
Semaphores

Tore Brox-Larsen, UiT,
Otto J. Anshus, UiT, UiO

Concurrency: Double buffering

/* Fill s and empty t concurrently */



Input sequence f

Get (s,f)

s

t

Output sequence g

Put (t,g)

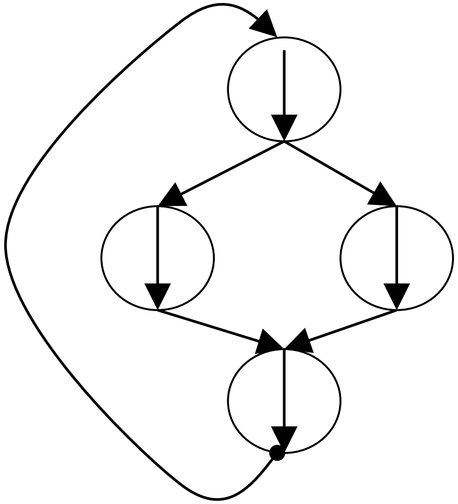
/* Copy */
t := s;

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

Specifies concurrent execution

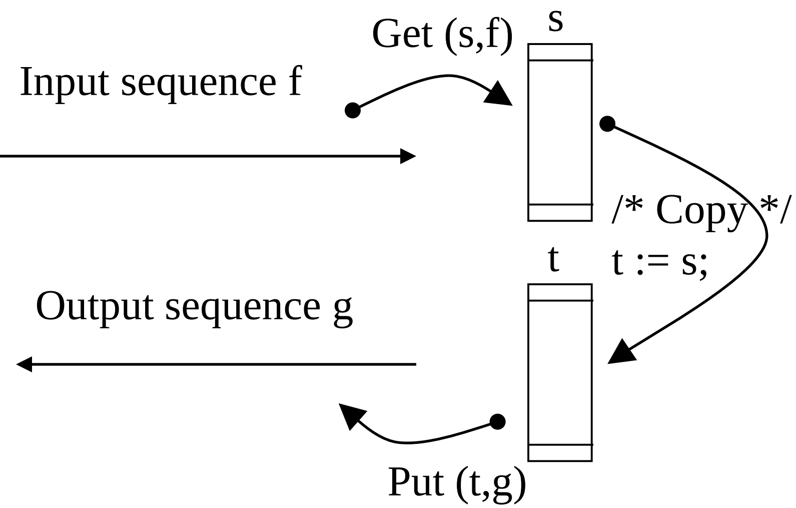
(Threads)

- **Put and Get** are disjunct
- … but not with regards to **Copy**!



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 scheduling by kernel */**

Proposed code:

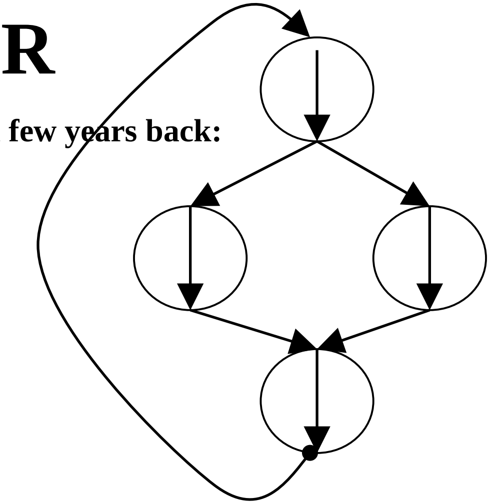
copy_thread:: *{acq(lock_t); acq(lock_s); **t=f;** rel(lock_s); rel(lock_t);**}**

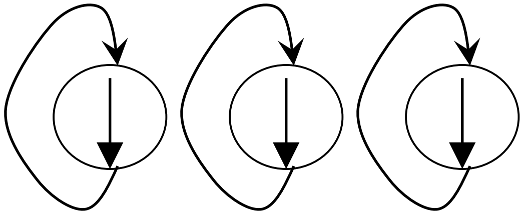
get_thread:: *{ack(lock_s); **s=f;** rel(lock_s);**}**

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

- **Not bad, but NO ORDER**

- **And as Thomas once said at the beginning of the course a few years back: Ordnung Muss Sein!**





Threads specifies concurrent execution

Protecting a Shared Variable

- Remember: we need a shared address space
  - threads inside a process share adr. space
- ***Acquire(mutex); count++; Release(mutex);***
- **(1) Acquire(mutex) system call**
  - User level library
    - **(2) Push parameters onto stack**
    - **(3) Trap to kernel (int instruction)**
  - Kernel level
    - Int handler
      - **(4) Verify valid pointer to mutex**
      - Jump to code for Acquire()
        - **(5) mutex closed: block caller: insert(current, mutex_queue)**
        - **(6) mutex open: get lock**
  - User level: **(7) execute count++**
- **(8) Release(mutex) system call**

Issues

- How “long” is the critical section?
- Competition for a mutex/lock
  - Uncontended = rarely in use by someone else
  - Contended = often used by someone else
  - Held = currently in use by someone
- Think about the results of these options
  - Spinning on low-cont. lock
  - Spinning on high-cont. lock
  - Blocking on low-cont. lock
  - Blocking on high-cont. lock

Block/unblock syscalls

- Block
  - Sleep on token
- Unblock
  - Wakes up first sleeper
- By the way
  - Remember that “test and set” works both at user and kernel level

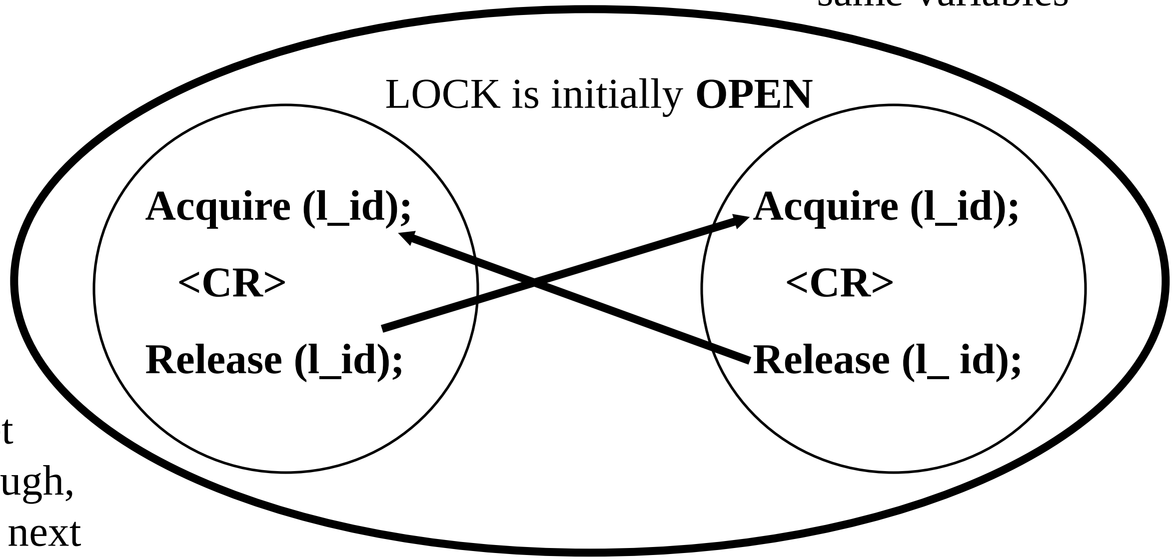
Implementing Block and Unblock

- Block (lock)
  - Spin on lock.guard
  - Save context to TCB
  - Enqueue TCB
  - Clear spin lock.guard
  - goto scheduler
- UnBlock(lock)
  - Spin on lock.guard
  - Dequeue a TCB
  - Put TCB in ready_queue
  - Clear spin lock.guard

Two Kinds of Synchronization

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

Process w/two threads



LOCK is initially **OPEN**

Acquire (l_id);

<CR>

Release (l_id);

Acquire (l_id);

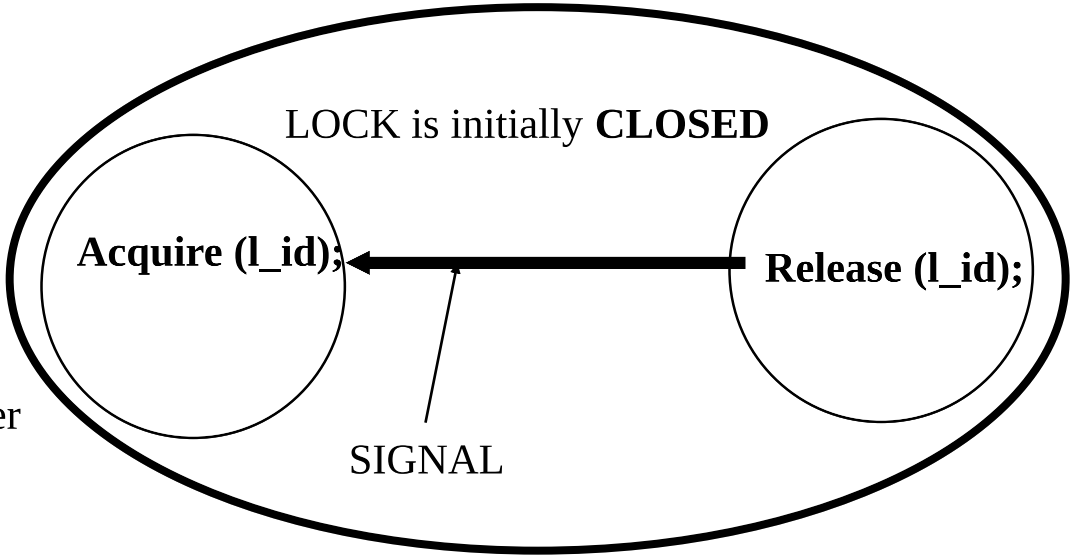
<CR>

Release (l_ id);

MUTEX

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

CONDITION SYNCHRONIZATION



LOCK is initially **CLOSED**

Acquire (l_id);

Release (l_id);

SIGNAL

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 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 T.H.E. operating system

- “Down(s)”/“Wait(s)”/“P(s)”
  - Atomic
  - DELAY (block, or busy wait) if not positive
  - Decrement semaphore value by 1

- “Up(s)”/”Signal(s)”/ “V(s)”
  - Atomic
  - Increment semaphore by 1
  - Wake up a waiting thread *if any*

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

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

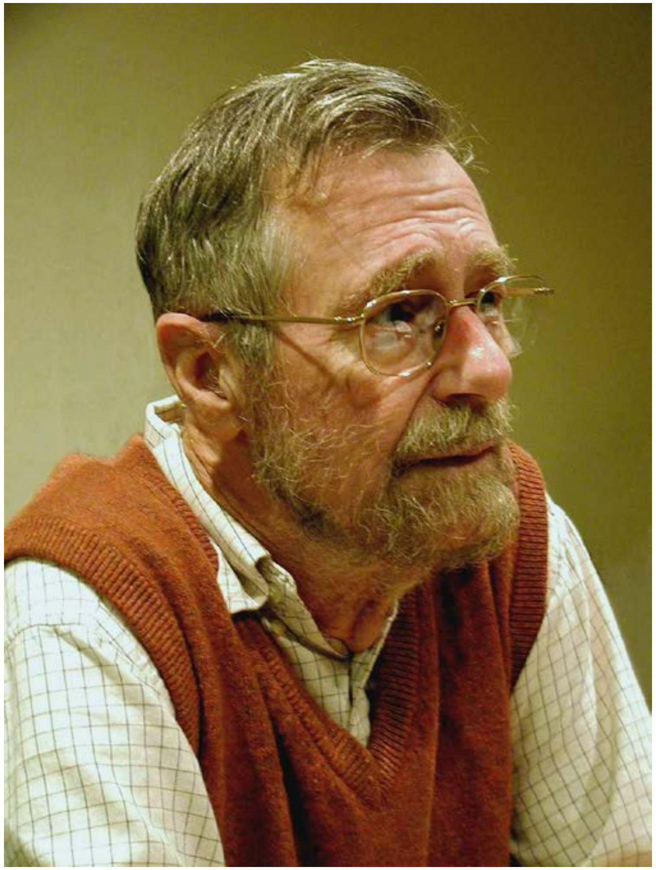
MUTEX

Can get negative s: counts number of waiting threads

s is NOT accessible through other means than calling P and V

An aside on Dijkstra

- Dutch, moved to UT/austin
- 1972 Turing Award Winner
- Go to statement considered harmful
- Homepage
- EDSAC Summer School



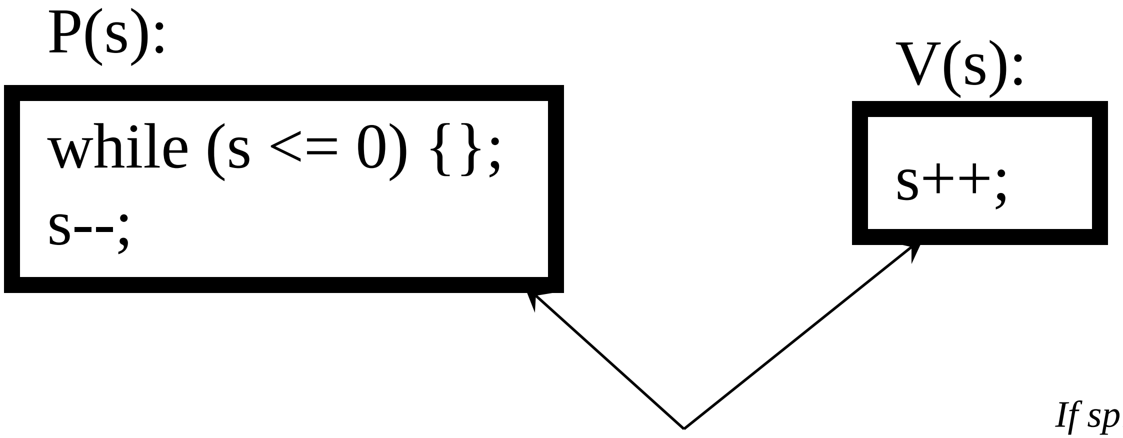
Semaphores can be used for …

- Mutual exclusion (solution of critical section problem). Binary semaphore
- Resources with multiple instances (e.g. buffer slots in producer/consumer problem. Counting semaphore
- Signaling events

Examples of classic synchronization problems

- Critical Section
- Producer/Consumer
- Reader/Writer
- Sleeping Barber
- Dining Philosophers

Semaphores w/Busy Wait



P(s):

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

V(s):

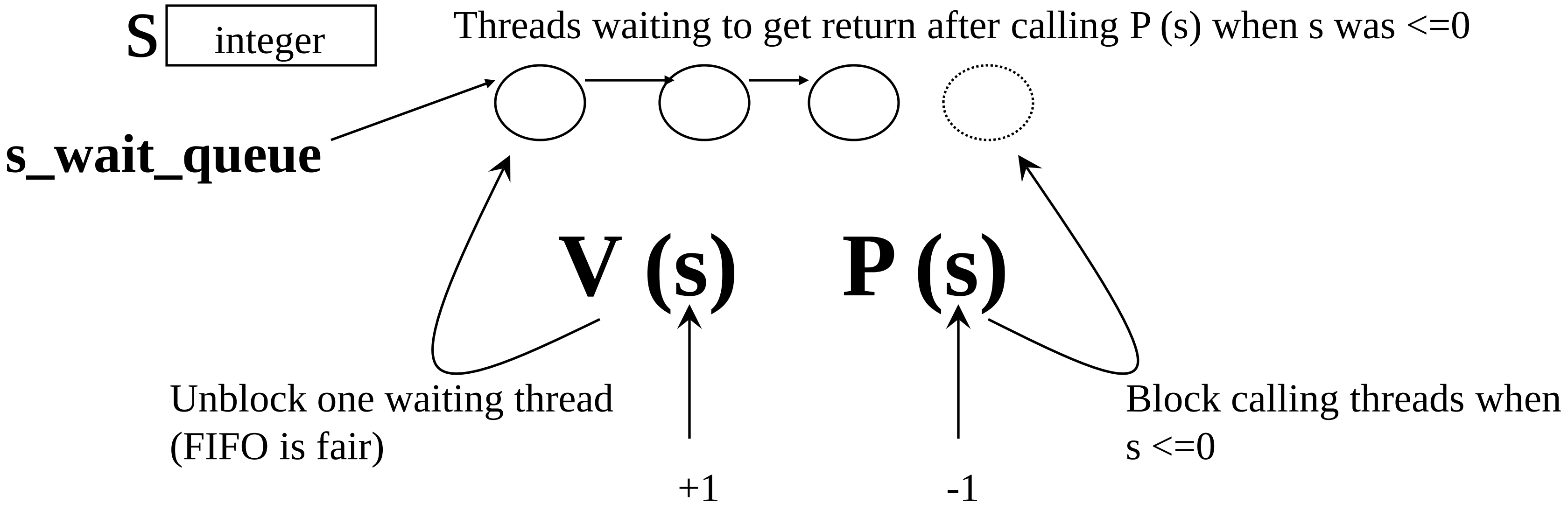
```
s++;
```

ATOMIC
(NB: mutex around
while can create a
problem…)

If spinning inside mutex
V will not get in:
- *Must open mutex, say, between every iteration of while to make it possible for V to get in*
  - *Costly*
  - *Starvation possible*
    - *Of P's*
    - *Of V's*

- Starvation possible (in theory)?
- Does it matter in practise?

The Structure of a Blocking Semaphore Implementation



S integer

Threads waiting to get return after calling P (s) when s was <=0

s_wait_queue

V (s) **P (s)**

Unblock one waiting thread
(FIFO is fair)

+1

-1

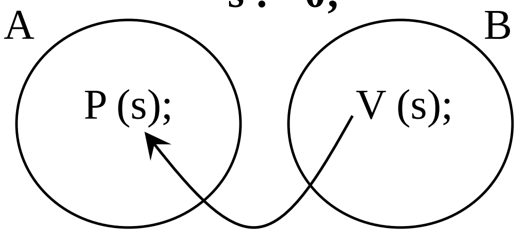
Block calling threads when
s <=0

- Atomic: Disable interrupts
- Atomic: P() and V() as System calls
- Atomic: Entry-Exit protocols

Using Semaphores

“The Signal”

s := 0;



A blocks until B says V

“The Mutex”

s := 1;

```
P (s);
   <CR>
V(s);
```

```
P (s);
   <CR>
V(s);
```

One thread gets in, next blocks until V is executed

NB: remember to set the initial semaphore value!

“The Team”

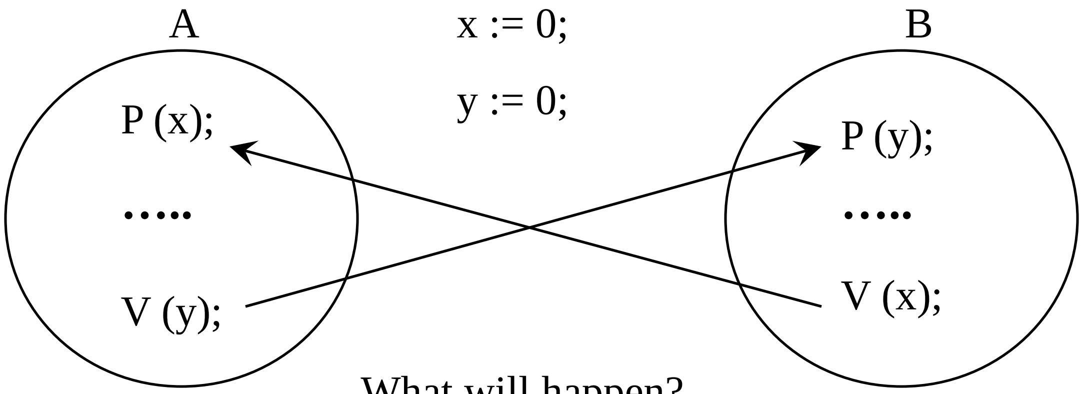
s := 8;

```
P (s);
 <max 8>
V(s);
```

```
P (s);
 <max 8>
V(s);
```

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?



A

P (x);

…..

V (y);

x := 0;

y := 0;

B

P (y);

…..

V (x);

What will happen?

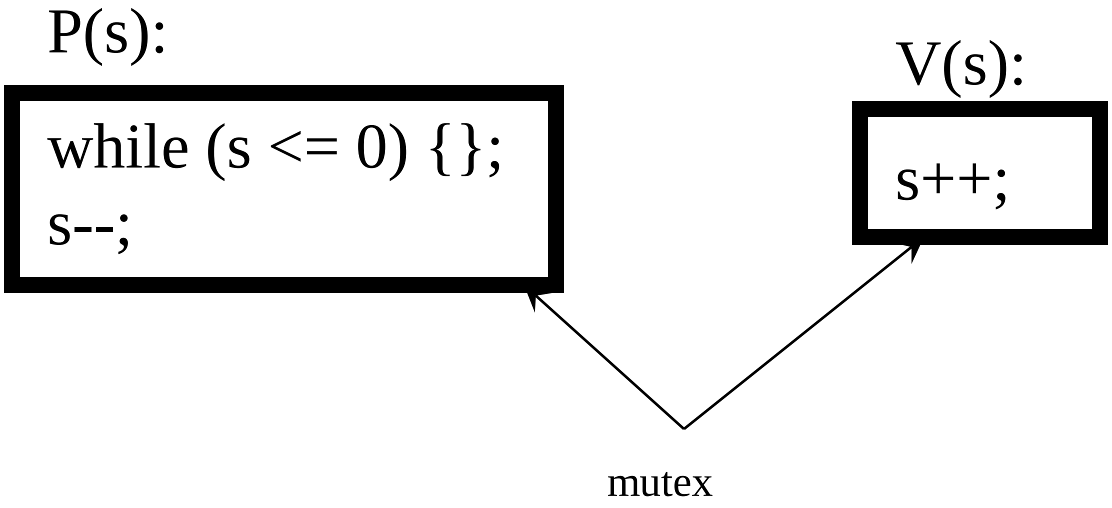
THEY ARE FOREVER WAITING FOR EACH OTHERS SIGNAL

Semaphores w/Busy Wait

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

Dutch words

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



P(s):

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

V(s):

```
s++;
```

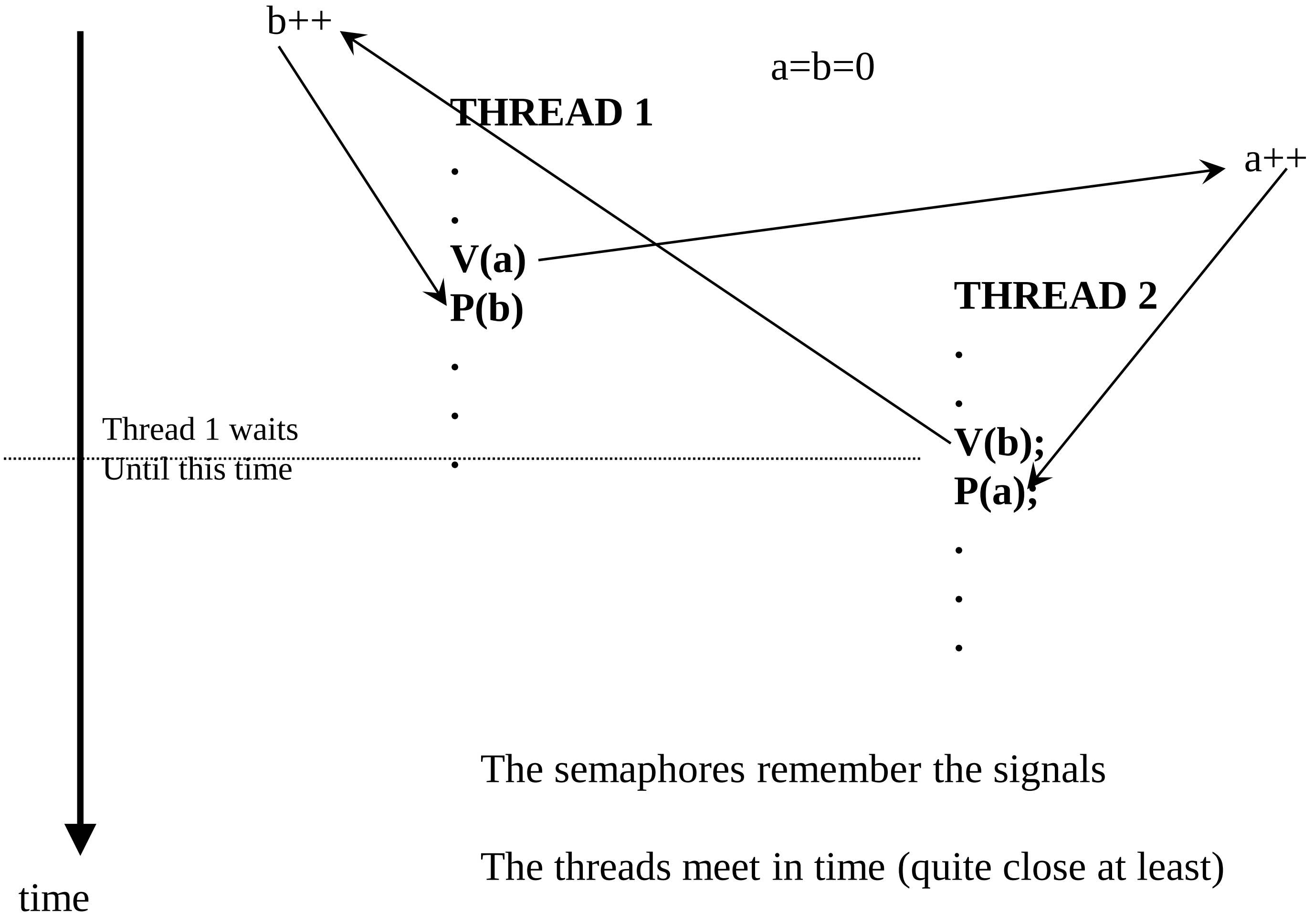
mutex

P == wait == down, V == signal == up

Why so many names?

- Down, up: what the ops *do*
- Wait, signal: what the ops are *used for*
- P, V: the original names by Dijkstra

Rendezvous between two threads



b++

a=b=0

THREAD 1

.

.

V(a)

P(b)

.

.

.

a++

THREAD 2

.

.

V(b);

P(a);

.

.

.

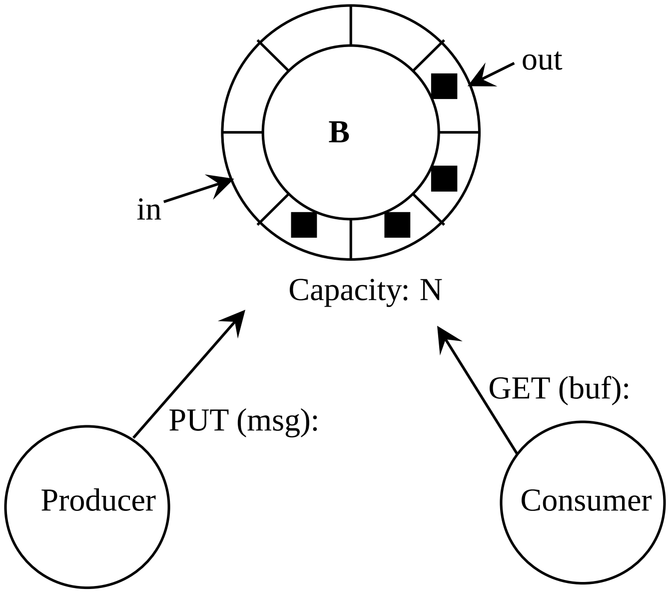
Thread 1 waits
Until this time

time

The semaphores remember the signals

The threads meet in time (quite close at least)

Bounded Buffer using Semaphores



Rules for the buffer B:

- No Get when empty
- No Put when full
- B shared, so must have mutex between Put and Get

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

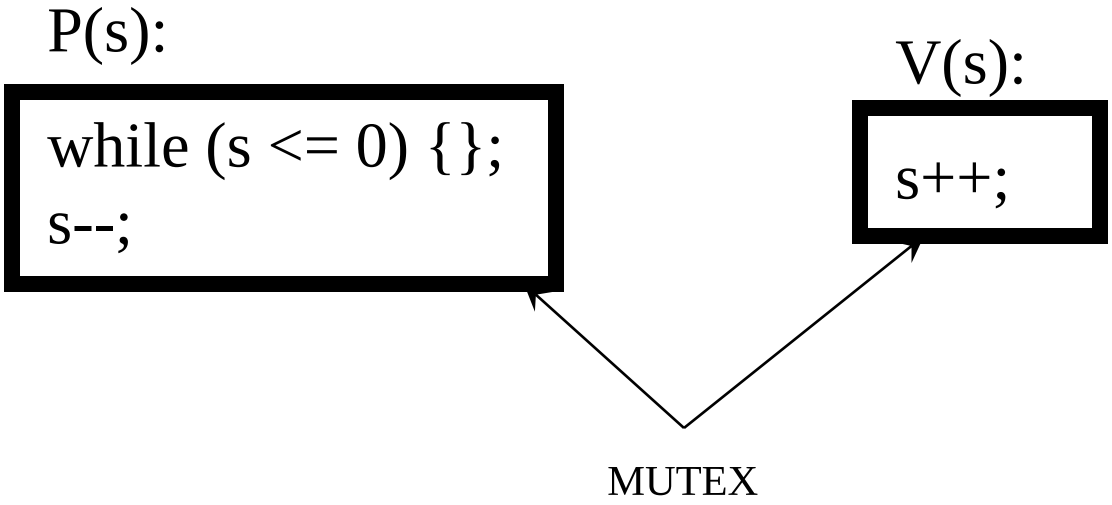
- B empty: nonempty:=0;
- B full: nonfull:=N
- B mutex: mutex:=1;

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

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

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

Semaphores w/Busy Wait



P(s):

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

V(s):

```
s++;
```

MUTEX

If P **spinning inside mutex then** V **will not get in**

- •Must *open mutex*, say, between every iteration of while to make it possible for V to get in
  - •Costly
    - •Every 10th iteration?
      - •latency
  - •Starvation possible, Lady Luck may ignore some threads
    - •Of P’s
    - •Of V’s

Hard life…

- Implementing the P and V of semaphores
  - If WAIT is done by blocking
    - Expensive
    - Must open mutex
      - But no logical issues since we now have a waiting queue and 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 practise
- What can a poor (perhaps somewhat theoretical oriented) Computer Scientist do?
  - Research ("I can do better")
  - Publish (So other people can say "I can do better")

Implementing Semaphores w/mutex

```
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)

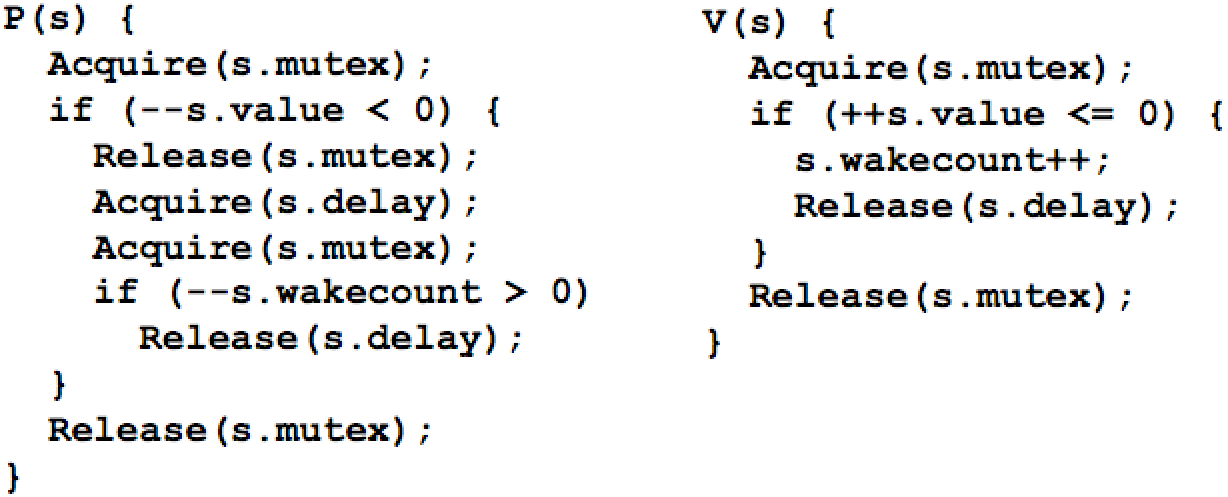
Hemmendinger’s solution (1988)

```
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)

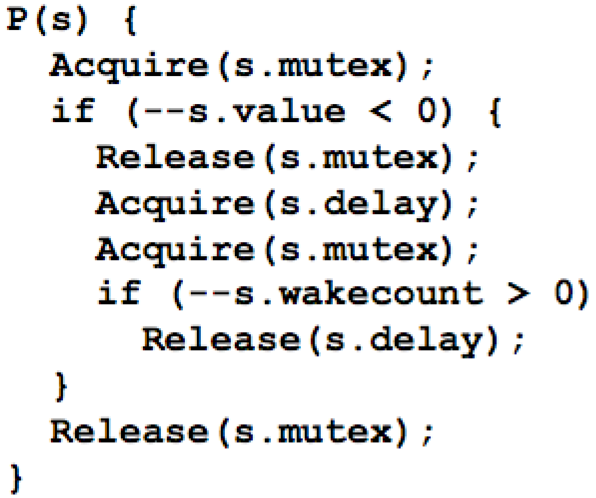


```
P(s) {
  Acquire(s.mutex);
  if (--s.value < 0) {
    Release(s.mutex);
    Acquire(s.delay);
    Acquire(s.mutex);
    if (--s.wakecount > 0)
      Release(s.delay);
  }
  Release(s.mutex);
}
```

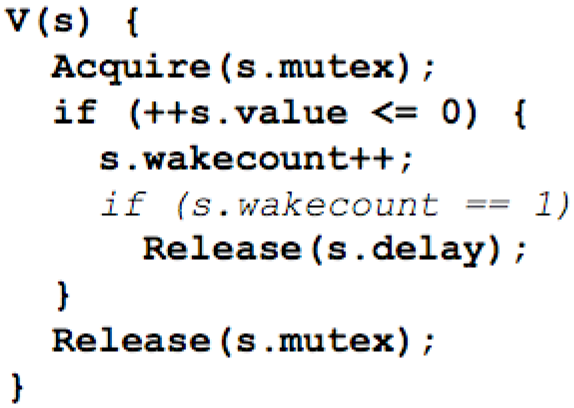
```
V(s) {
  Acquire(s.mutex);
  if (++s.value <= 0) {
    s.wakecount++;
    Release(s.delay);
  }
  Release(s.mutex);
}
```

Two Release(s.delay) calls are also possible

Hemmendinger’s Correction (1989)



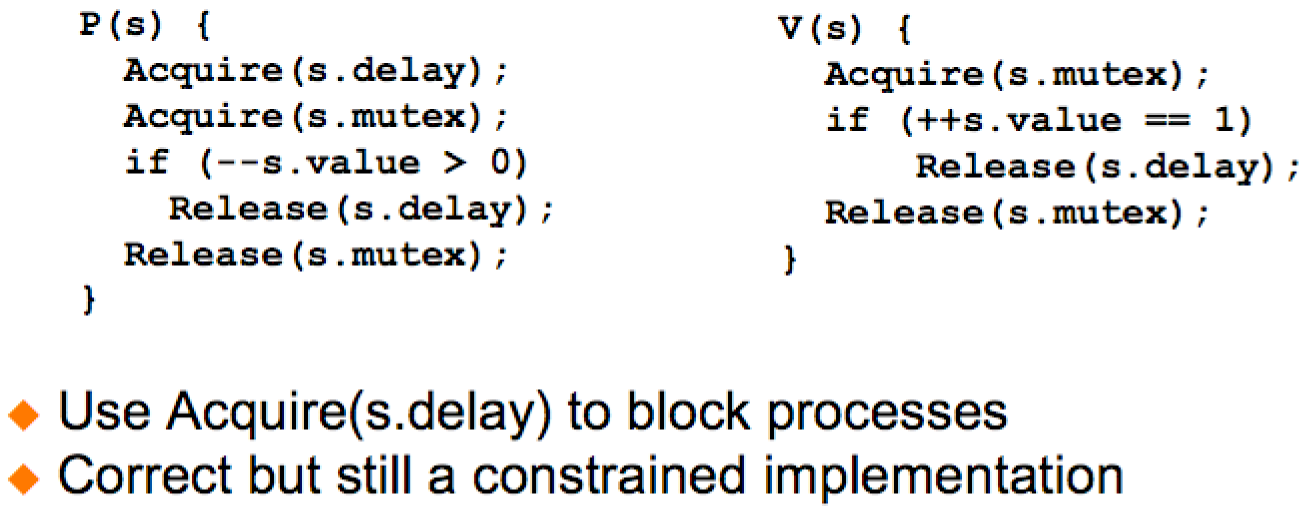
```
P(s) {
  Acquire(s.mutex);
  if (--s.value < 0) {
    Release(s.mutex);
    Acquire(s.delay);
    Acquire(s.mutex);
    if (--s.wakecount > 0)
      Release(s.delay);
  }
  Release(s.mutex);
}
```



```
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)

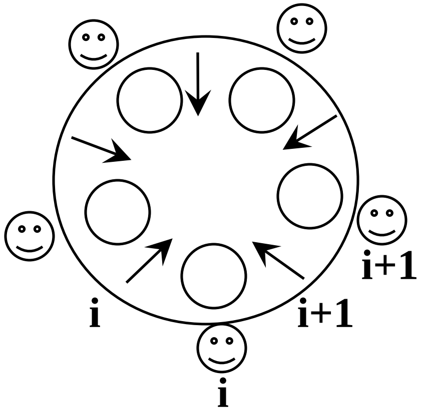


```
P(s) {
  Acquire(s.delay);
  Acquire(s.mutex);
  if (--s.value > 0)
    Release(s.delay);
  Release(s.mutex);
}
```

```
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

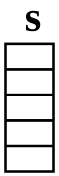
Dining Philosophers



- 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
- 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));
```

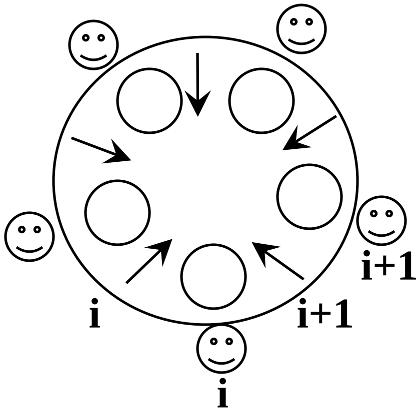
T_i

Get L; Get R if free else Put L;

- *Starvation possible*

T_i

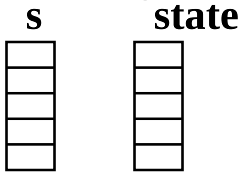
Dining Philosophers



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;
}
```

```
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 neighbor must say V(i) to myself!*/
```

```
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

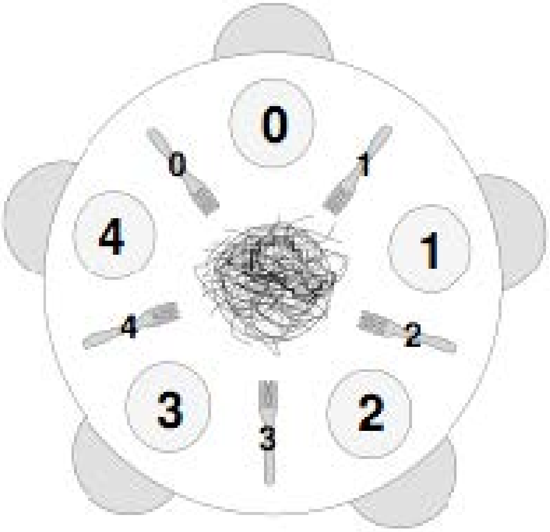
Last solution has a problem

Trouble in Tanenbaums solution:

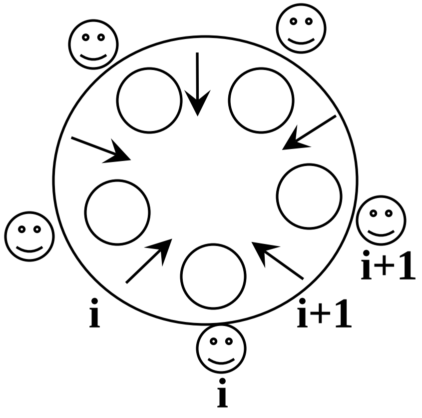
starvation pattern possible:

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



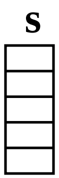
Can we in a simple way do better than this one?

Get L; Get R;

- Deadlock possible

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

- Non-symmetric solution. Still quite elegant



S(i) = 1 initially

- 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));
```

Some Links

- Wikipedia: *Semaphore*
- Alan B. Downey: *The Little Book of Semaphores*
    - *Book*
    - *Video Lecture*
- Jouni Leppäjärvi: *Master’s Thesis*