

Operating Systems: Synchronization

Raymond Namyst

Dept. of Computer Science

University of Bordeaux, France

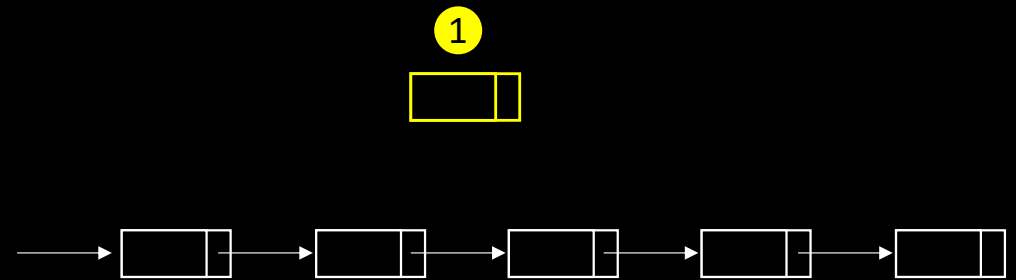
<https://gforgeron.gitlab.io/se/>

Why do we need synchronization?

- To protect data structures from being corrupted by concurrent execution of non-reentrant code
 - Mutual exclusion of critical sections
 - Reader/Writer
 - Producer/consumer
- To enforce dependencies between code regions
 - Blocking a process until an event has occurred
 - Joining a synchronization point

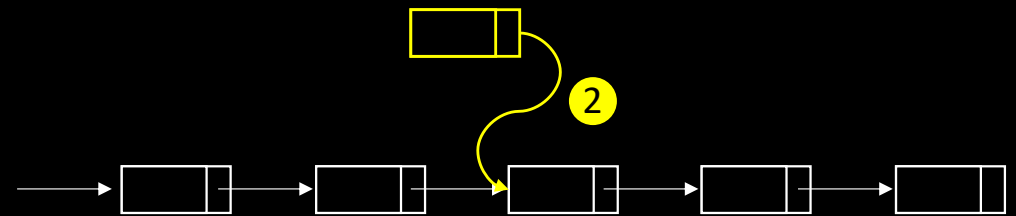
Race conditions

- Example with linked lists
 - Insertion of a new element
 - 3 steps
 1. Allocate
 2. Set next
 3. Modify previous



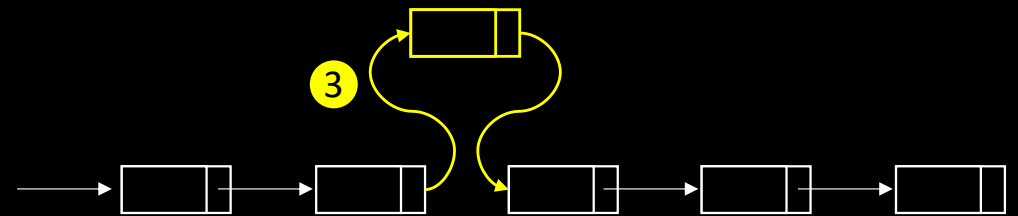
Race conditions

- Example with linked lists
 - Insertion of a new element
 - 3 steps
 1. Allocate
 2. Set next
 3. Modify previous



Race conditions

- Example with linked lists
 - Insertion of a new element
 - 3 steps
 1. Allocate
 2. Set next
 3. Modify previous



Race conditions

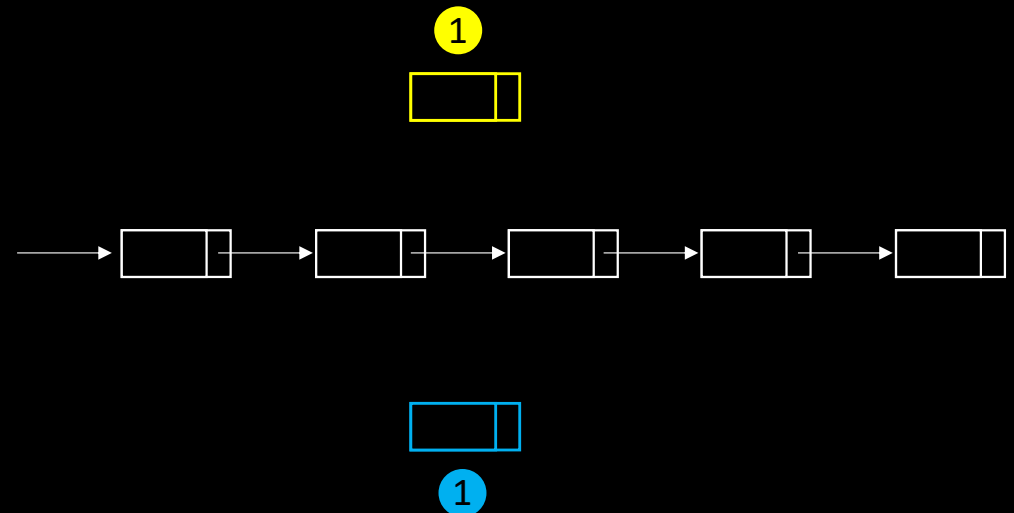
- Example with linked lists

- Insertion of a new element

- 3 steps

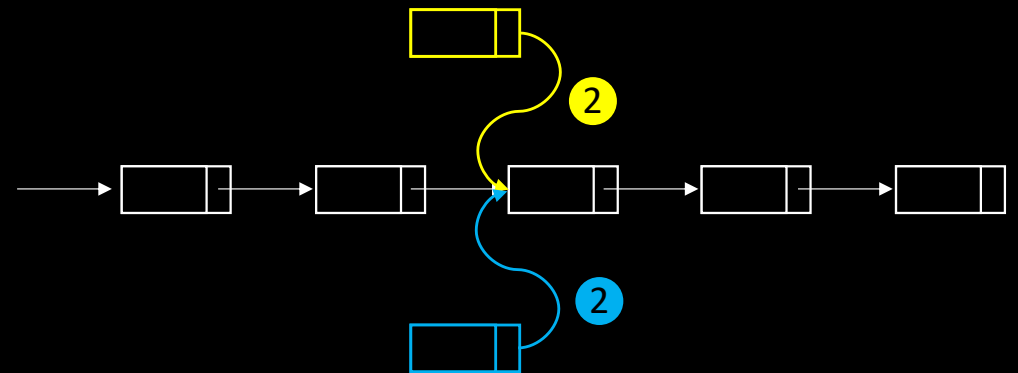
1. Allocate
2. Set next
3. Modify previous

- What if two threads perform an insert simultaneously, at the same position?



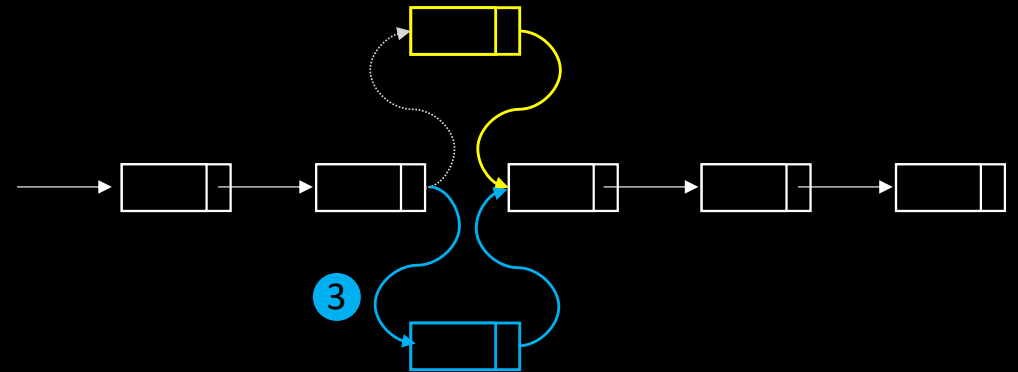
Race conditions

- Example with linked lists
 - Insertion of a new element
 - 3 steps
 1. Allocate
 2. Set next
 3. Modify previous
 - What if two threads perform an insert simultaneously, at the same position?



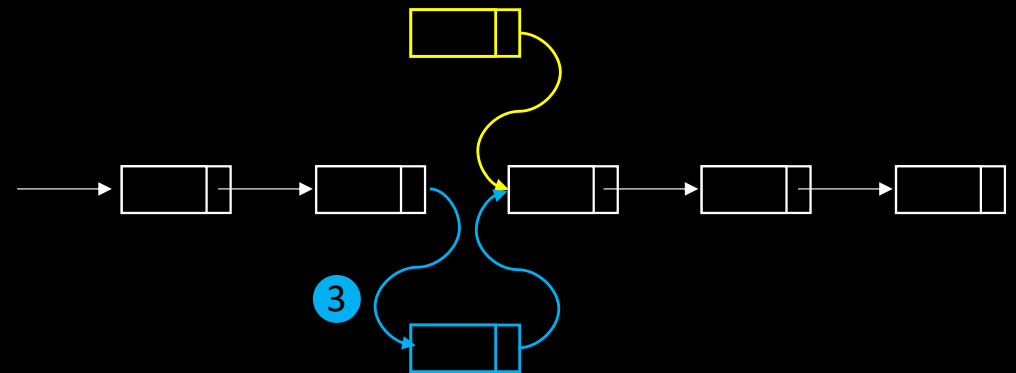
Race conditions

- Example with linked lists
 - Insertion of a new element
 - 3 steps
 1. Allocate
 2. Set next
 3. Modify previous
 - What if two threads perform an insert simultaneously, at the same position?



Race conditions

- Example with linked lists
 - Insertion of a new element
 - 3 steps
 1. Allocate
 2. Set next
 3. Modify previous
 - What if two threads perform an insert simultaneously, at the same position?
 - We may end up with a corrupted list



Enforcing mutual exclusion

- Let's say we want to wrap our critical section this way

```
enter_cs ()  
<critical section>  
exit_cs ()
```

- How make sure that
at most 1 thread at a time
can run in the critical section?

- Possible using read/write operations?

- Reading or writing a word (64 bits) is atomic on modern computers

Enforcing mutual exclusion

- Let's say we want to wrap our critical section this way

```
enter_cs ()  
<critical section>  
exit_cs ()
```

- Think about shower cabins

- Red flag = busy
- Green flag = free

```
bool busy = FALSE;
```

```
void enter_sc ()  
{  
    while (busy)  
        /* wait */ ;  
    busy = TRUE;  
}
```

```
void exit_sc ()  
{  
    busy = FALSE;  
}
```

Enforcing mutual exclusion

- Let's say we want to wrap our critical section this way

```
enter_cs ()  
<critical section>  
exit_cs ()
```

- Think about shower cabins

- Red flag = busy
- Green flag = free

```
bool busy = FALSE;
```

```
void enter_sc ()
```

```
{
```

```
    if (busy)
```



```
void exit_sc ()
```

```
{
```

```
    busy = FALSE;
```

```
}
```

Enforcing mutual exclusion

```
// Gary L. Peterson, 1981. Works with two processes : #0 and #1
bool flag [2] = { FALSE, FALSE };
unsigned turn = 0;

void enter_sc ()
{
    flag[me] = TRUE;
    turn = me;
    wait (flag[1 - me] == FALSE or turn != me);
}

void exit_sc ()
{
    flag[me] = FALSE;
}
```


Enforcing mutual exclusion

```
bool flag [2] = { FALSE, TRUE };
unsigned turn = 0;
```

```
// Thread #0
```

```
// Thread #1
```

```
void enter_sc ()
{
    flag[1] = TRUE;
}

```

Enforcing mutual exclusion

```
bool flag [2] = { FALSE, TRUE };
unsigned turn = 1;
```

```
// Thread #0
```

```
// Thread #1
```

```
void enter_sc ()
{
    flag[1] = TRUE;
    turn = 1;
}
```


Enforcing mutual exclusion

```
bool flag [2] = { FALSE, TRUE };
unsigned turn = 1;
```

```
// Thread #0
```

```
// Thread #1
```

Enforcing mutual exclusion

```
bool flag [2] = { FALSE, TRUE };
unsigned turn = 1;
```

```
// Thread #0
```

```
// Thread #1
```

```
void enter_sc ()
{
    flag[1] = TRUE;
    turn = 1;
    // Not blocked...
}
```

Enforcing mutual exclusion

```
bool flag [2] = { TRUE, TRUE };  
unsigned turn = 1;
```

```
// Thread #0
```

```
void enter_sc ()
```

```
{
```

```
    flag[0] = TRUE;
```

```
}
```

```
// Thread #1
```

```
void enter_sc ()
```

```
{
```

```
    flag[1] = TRUE;
```

```
    turn = 1;
```

```
    // Not blocked...
```

```
}
```

Enforcing mutual exclusion

```
bool flag [2] = { TRUE, TRUE };  
unsigned turn = 0;
```

```
// Thread #0
```

```
void enter_sc ()  
{  
  
    flag[0] = TRUE;  
    turn = 0;  
  
}
```

```
// Thread #1
```

```
void enter_sc ()  
{  
    flag[1] = TRUE;  
    turn = 1;  
    // Not blocked...  
}
```

Enforcing mutual exclusion

```
bool flag [2] = { TRUE, TRUE };  
unsigned turn = 0;
```

```
// Thread #0
```

```
void enter_sc ()  
{
```

```
    flag[0] = TRUE;
```

```
    turn = 0;
```

```
    wait (flag[1] == FALSE or turn != 0);
```

```
}
```

```
// Thread #1
```

```
void enter_sc ()
```

```
{
```

```
    flag[1] = TRUE;
```

```
    turn = 1;
```

```
    // Not blocked...
```

```
}
```

Enforcing mutual exclusion

```
bool flag [2] = { TRUE, TRUE };  
unsigned turn = 0;
```

```
// Thread #0
```

```
void enter_sc ()  
{
```

```
    flag[0] = TRUE;  
    turn = 0;  
    wait (flag[1] == FALSE or turn != 0);  
    // Blocked  
}
```

```
// Thread #1
```

```
void enter_sc ()  
{  
    flag[1] = TRUE;  
    turn = 1;  
    // Not blocked...  
}
```

Enforcing mutual exclusion

```
bool flag [2] = { TRUE, TRUE };  
unsigned turn = ?;
```

```
// Thread #0
```

```
void enter_sc ()  
{  
    flag[0] = TRUE;  
    turn = 0;
```

```
    wait (flag[1] == FALSE or turn != 0);  
}
```

```
// Thread #1
```

```
void enter_sc ()  
{  
    flag[1] = TRUE;  
    turn = 1;
```

```
    wait (flag[0] == FALSE or turn != 1);  
}
```

Enforcing mutual exclusion

```
bool flag [2] = { TRUE, TRUE };  
unsigned turn = 1;
```

```
// Thread #0
```

```
void enter_sc ()  
{  
    flag[0] = TRUE;  
    turn = 0;  
  
    wait (flag[1] == FALSE or turn != 0);  
}
```

```
// Thread #1
```

```
void enter_sc ()  
{  
    flag[1] = TRUE;  
  
    turn = 1;  
  
    wait (flag[0] == FALSE or turn != 1);  
}
```


Enforcing mutual exclusion

- **Peterson's algorithm limitations**
 - Only works with two processes
 - Introduces busy waiting
- **Extending Peterson's algorithm to work with N processes**
 - Idea:
 - Imagine a stairway to ~~heaven~~ critical section
 - At each step, a modified version of Peterson lets only N-1 processes access to the upper step
 - Eventually, only one process reaches the top (= critical section)

Enforcing mutual exclusion

- Coping with N processes is undoubtedly better...
 - But in a real kernel, we don't know how many processes will participate!
- We need a mechanism which works with an arbitrary number of processes/threads
 - Maybe computer architects can help us?

Enforcing mutual exclusion

- Processor atomic instructions
 - Execution cannot be interleaved with the execution of another instruction
 - Modern processors feature a large set of atomic operations
- To enforce mutual exclusion, we only need one
 - “*Test-and-Set*” (TAS)

Enforcing mutual exclusion

```
int test_and_set (int *address)
{
    int old = *address;
    *address = 1;
    return old;
}
```

- *test-and-set* is a **hardware processor instruction**
 - This C code should be considered for illustrative purposes only
 - memory side effects
 - returned value
 - Its execution wouldn't guarantee atomicity!

test-and-set

- How does it work?

```
int i = 0;  
test_and_set (&i) → ?
```

test-and-set

- How does it work?

```
int i = 1;  
test_and_set (&i) → 0
```

test-and-set

- How does it work?

```
int i = 1;  
test_and_set (&i) → 0  
test_and_set (&i) → ?
```

test-and-set

- How does it work?

```
int i = 1;  
test_and_set (&i) → 0  
test_and_set (&i) → 1
```


test-and-set

- How does it work?

```
int i = 1;  
test_and_set (&i) → 0  
test_and_set (&i) → 1  
test_and_set (&i) → ?
```

test-and-set

- How does it work?

```
int i = 1;  
test_and_set (&i) → 0  
test_and_set (&i) → 1  
test_and_set (&i) → 1
```

test-and-set

- How does it work?
 - Well... straightforward
- What happens when multiple threads use it simultaneously?

```
int i = 1;  
test_and_set (&i) → 0  
test_and_set (&i) → 1  
test_and_set (&i) → 1  
test_and_set (&i) → 1  
...
```

test-and-set

```
volatile int i = 0;
```

 `test_and_set (&i) → ?`

 `test_and_set (&i) → ?`

 `test_and_set (&i) → ?`

test-and-set

```
volatile int i = 1;
```


`test_and_set (&i) → 1`


`test_and_set (&i) → 0`


`test_and_set (&i) → 1`

Because *test-and-set* is atomic, only ONE thread gets 0

test-and-set

```
volatile int i = 1;
```



test_and_set (&i) → 1
test_and_set (&i) → 1
...



test_and_set (&i) → 0



test_and_set (&i) → 1
test_and_set (&i) → 1
...

test-and-set

```
volatile int i = 0;
```



```
test_and_set (&i) → 1  
test_and_set (&i) → 1  
...
```



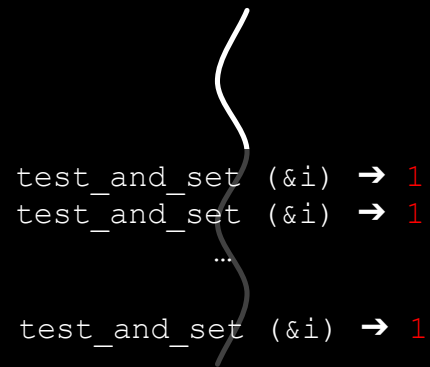
```
test_and_set (&i) → 0  
i = 0
```



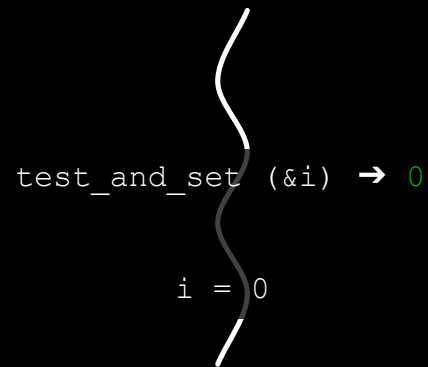
```
test_and_set (&i) → 1  
test_and_set (&i) → 1  
...
```

test-and-set

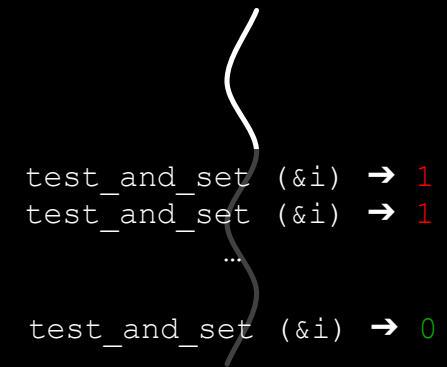
```
volatile int i = 0;
```



```
test_and_set (&i) → 1  
test_and_set (&i) → 1  
...  
test_and_set (&i) → 1
```



```
test_and_set (&i) → 0  
...  
i = 0
```



```
test_and_set (&i) → 1  
test_and_set (&i) → 1  
...  
test_and_set (&i) → 0
```

It seems we've found a simple protocol to enter/exit critical sections!

Enforcing mutual exclusion

- Let's say we want to wrap our critical section this way

```
enter_cs ()  
<critical section>  
exit_cs ()
```

- Think about shower cabins

- Red flag = busy
- Green flag = free

```
int lock = 0;  
  
void enter_sc ()  
{  
  
}  
  
void exit_sc ()  
{  
  
}
```

Enforcing mutual exclusion

- Let's say we want to wrap our critical section this way

```
enter_cs ()  
<critical section>  
exit_cs ()
```

- Think about shower cabins

- Red flag = busy
- Green flag = free

```
int lock = 0;  
  
void enter_sc ()  
{  
    while (test_and_set(&lock) == 1)  
        /* wait */ ;  
}  
  
void exit_sc ()  
{  
  
}
```

Enforcing mutual exclusion

- Let's say we want to wrap our critical section this way

```
enter_cs ()  
<critical section>  
exit_cs ()
```

- Think about shower cabins

- Red flag = busy
- Green flag = free

```
int lock = 0;  
  
void enter_sc ()  
{  
    while (test_and_set(&lock) == 1)  
        /* wait */ ;  
}  
  
void exit_sc ()  
{  
    lock = 0;  
}
```

Enforcing mutual exclusion

- Note: the following variant is usually preferred
 - Less pressure on the memory bus

```
int lock = 0;

void enter_sc ()
{
    while (test_and_set(&lock) == 1)
        while (lock)
            /* wait */ ;
}

void exit_sc ()
{
    lock = 0;
}
```

Enforcing mutual exclusion

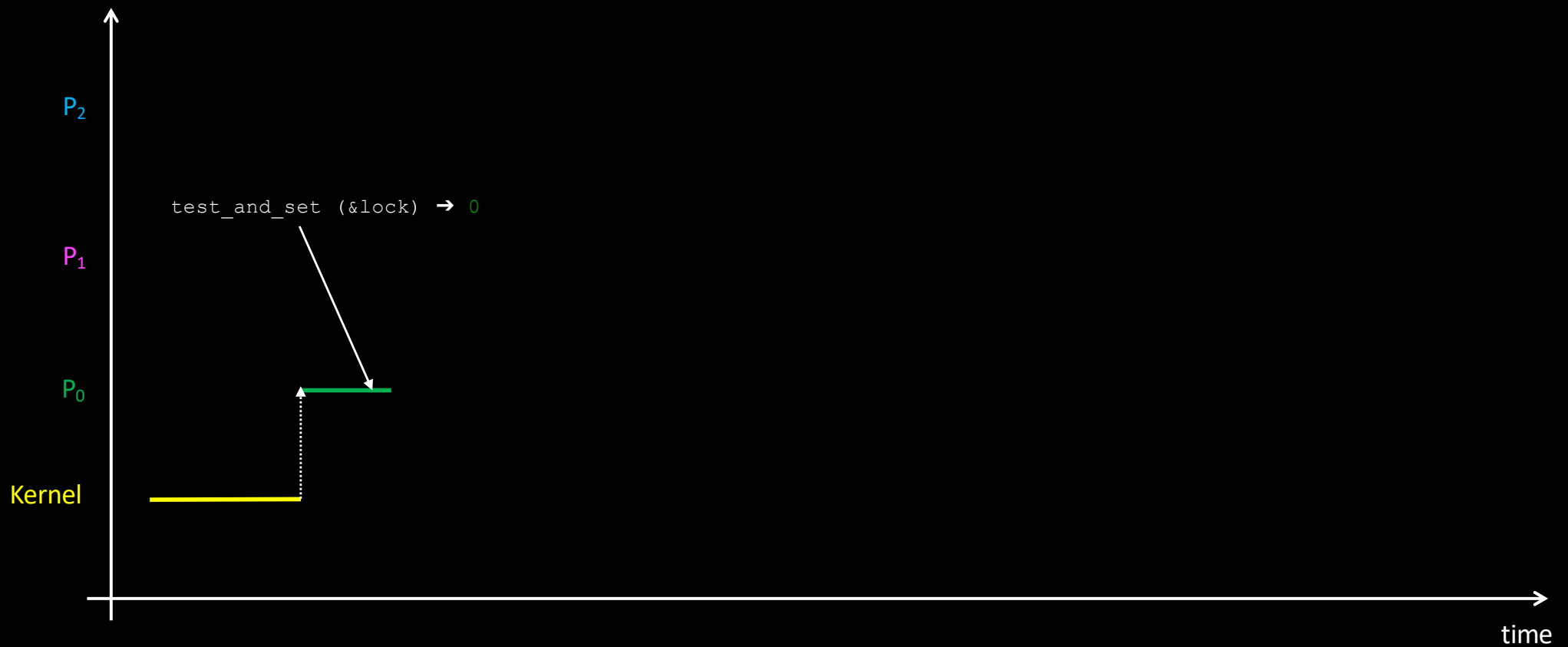
- Note: the following variant is usually preferred
 - Less pressure on the memory bus
- Still one (major ?) issue:
 - Busy waiting

```
int lock = 0;

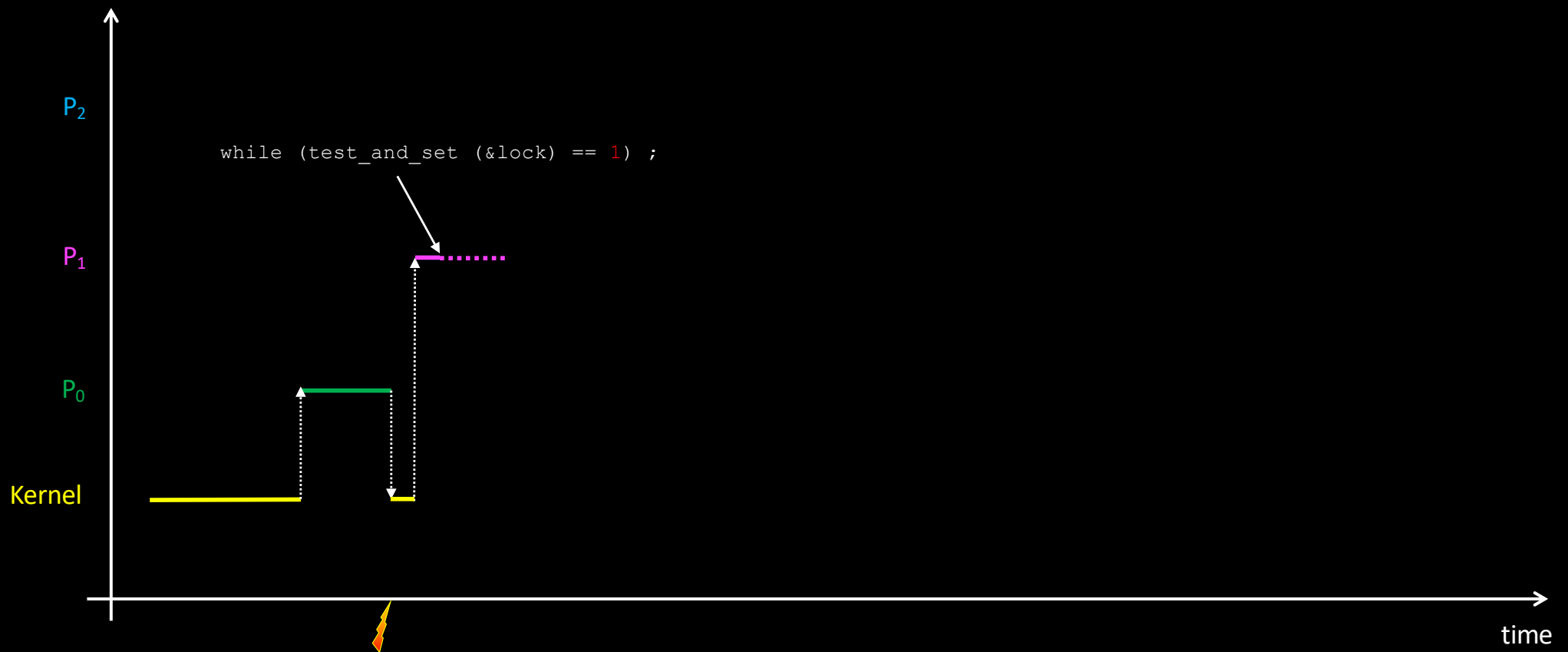
void enter_sc ()
{
    while (test_and_set(&lock) == 1)
        while (lock)
            /* wait */ ;
}

void exit_sc ()
{
    lock = 0;
}
```

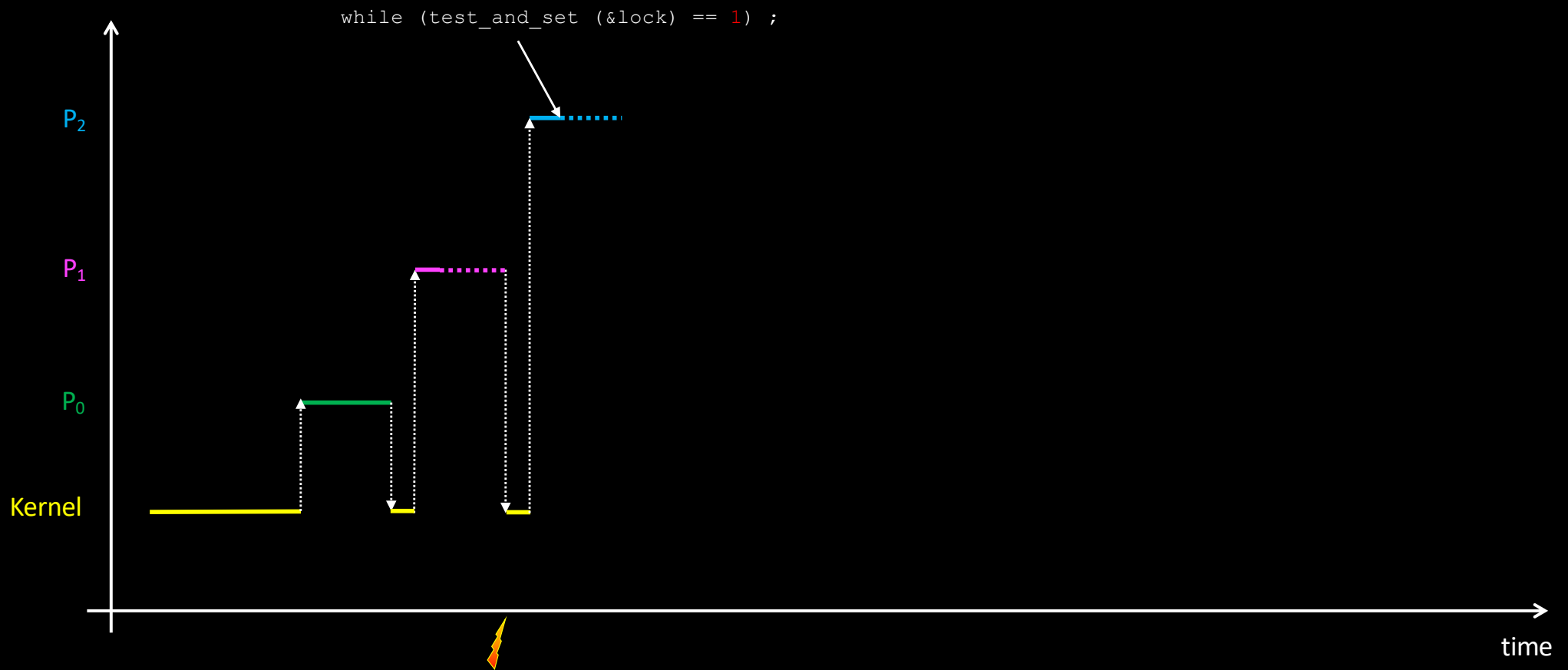
Is *busy waiting* a major issue?



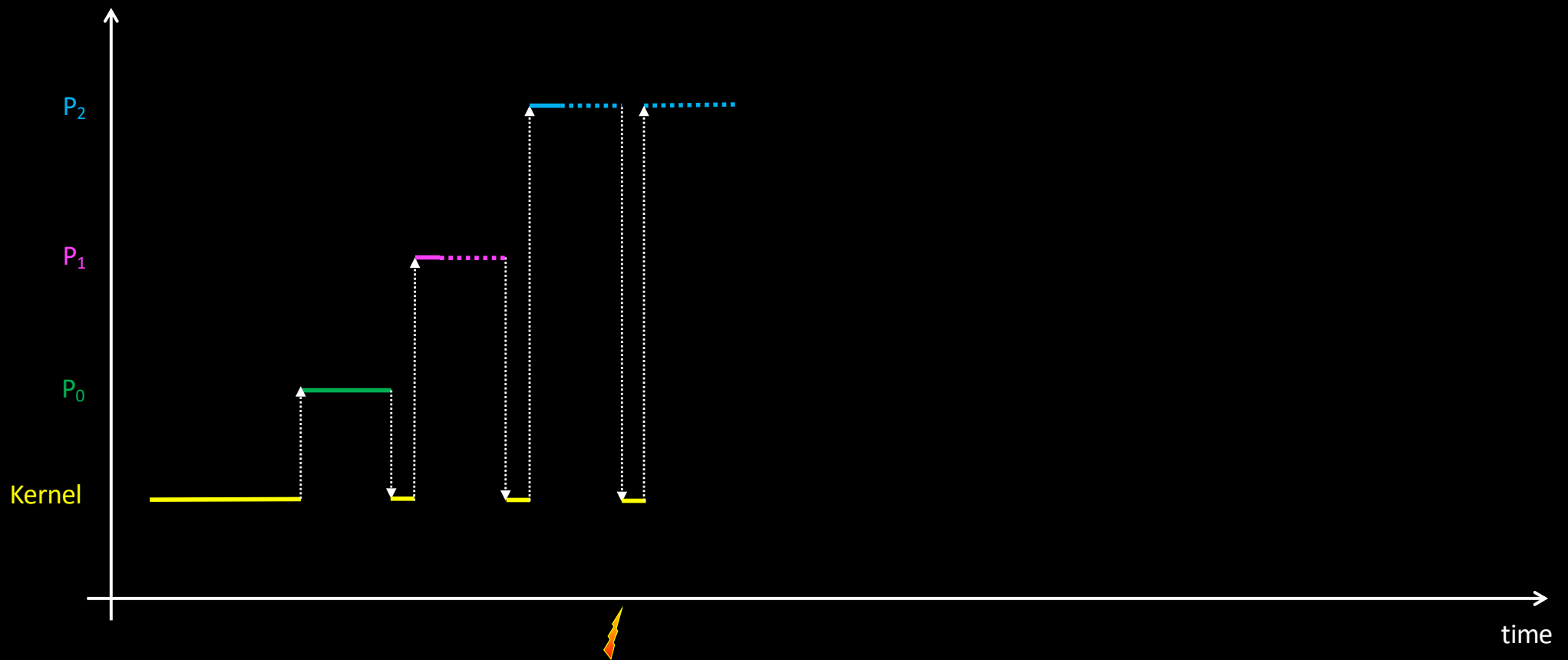
Is *busy waiting* a major issue?



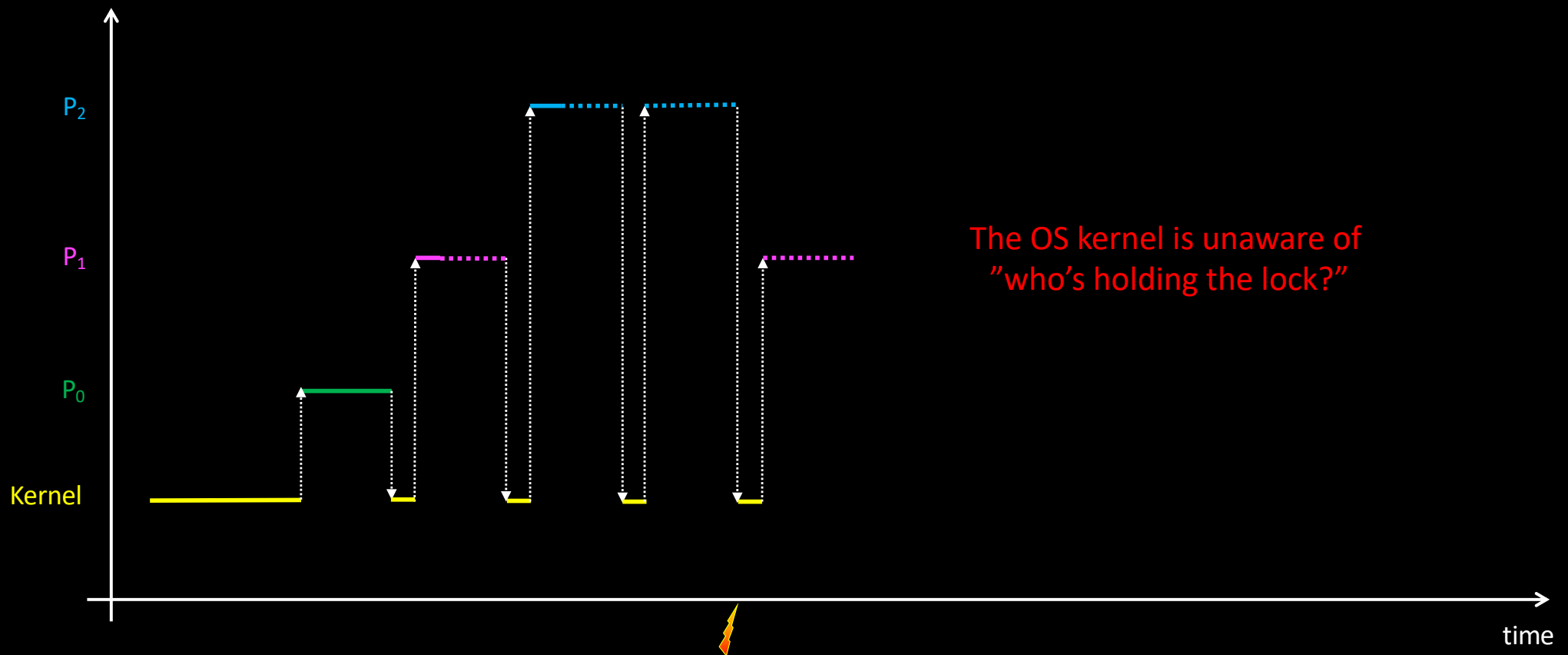
Is *busy waiting* a major issue?



Is *busy waiting* a major issue?



Is *busy waiting* a major issue?



Enforcing mutual exclusion

- Atomic instructions solve the problem at the lowest level
 - But *busy waiting* is a waste of CPU cycles
- We need to block processes which cannot enter critical section
 - Blocking processes can only be done inside the kernel

Enforcing mutual exclusion

- Atomic instructions solve the problem at the lowest level
 - But *busy waiting* is a waste of CPU cycles
- We need to block processes which cannot enter critical section
 - Blocking processes can only be done inside the kernel

```
int lock = 0;

void enter_sc ()
{
    while (test_and_set (&lock) == 1)
        syscall_enter_block_state ();
}

void exit_sc ()
{
    lock = 0;
    if (processes_are_waiting ())
        wake_up_one_process ();
}
```

Enforcing mutual exclusion

- Atomic instructions solve the problem at the lowest level
 - But *busy waiting* is a waste of CPU cycles
- We need to block processes which cannot enter critical section
 - Blocking processes can only be done inside the kernel

```
int lock = 0;

void enter_sc ()
{
    while (test_and_set (&lock) == 1)
        syscall_enter_block_state ();
}

void exit_sc ()
{
    lock = 0;
    if (processes_are_waiting ())
        wake_up_one_process ();
}
```



Semaphores [E. Dijkstra, 1962]

- Semaphores are high-level (*are you kidding me?*) synchronization objects
 - Implemented in the kernel
- A Semaphore is a structure containing **private** information
 - An integer value (initially ≥ 0)
 - A list of waiting processes (initially = NULL)
- Only 2 available operations. Both are atomic:
 - P (*probeer te verlagen*)
 - V (*vrijgave*)

Semaphores [E. Dijkstra, 1962]

```
// P is atomic
P(s)
{
    s->value--;
    if (s->value < 0)
        go_sleep_in (s->list);
}
```

Semaphores [E. Dijkstra, 1962]

```
// P is atomic
```

```
P(s)
```

```
{
```

```
    s->value--;
```

```
    if (s->value < 0)
```

```
        go_sleep_in (s->list);
```

```
}
```

```
// V is atomic
```

```
V(s)
```

```
{
```

```
    s->value++;
```

```
    if (s->value <= 0)
```

```
        wake_one_from (s->list);
```

```
}
```


Semaphores [E. Dijkstra, 1962]

```
// P is atomic
```

```
P(s)
```

```
{
```

```
    s->value--;
```

```
    if (s->value < 0)
```

```
        go_sleep_in (s->list);
```

```
}
```

```
// V is atomic
```

```
V(s)
```

```
{
```

```
    s->value++;
```

```
    if (s->value <= 0)
```

```
        wake_one_from (s->list);
```

```
}
```

Semaphores can be seen as boxes containing tokens:

- P() == wait for a token in the box, take it and continue
- V() == throw a token into the box

Semaphore initialized to zero

- Let's see how it works on a simple example
 - Two threads
 - One calls P()
 - The other calls V()

```
Semaphore s(0);
```



Semaphore initialized to zero

- Let's see how it works on a simple example
 - Two threads
 - One calls P()
 - The other calls V()

Semaphore s(0);

P(s)



Semaphore initialized to zero

- Let's see how it works on a simple example
 - Two threads
 - One calls P()
 - The other calls V()

Semaphore s(-1);



Semaphore initialized to zero

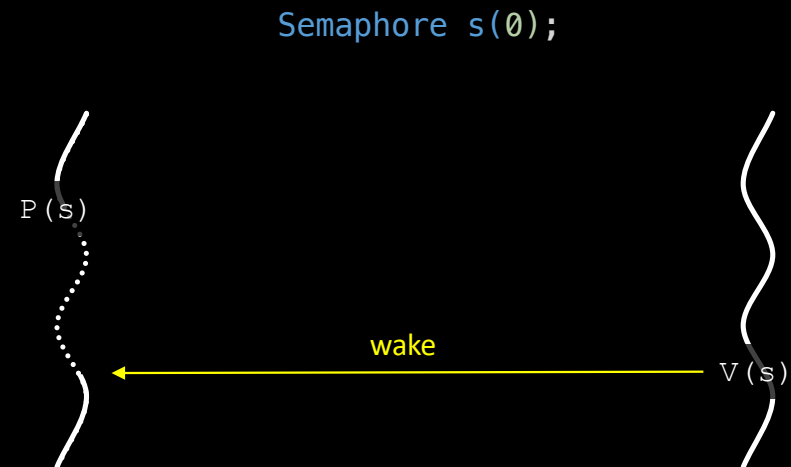
- Let's see how it works on a simple example
 - Two threads
 - One calls P()
 - The other calls V()

Semaphore s(-1);



Semaphore initialized to zero

- Let's see how it works on a simple example
 - Two threads
 - One calls P()
 - The other calls V()
- When P is called first, it blocks the caller until... someone else calls V



Semaphore initialized to zero

- Let's see how it works on a simple example
 - Two threads
 - One calls P()
 - The other calls V()

Semaphore s(0);



Semaphore initialized to zero

- Let's see how it works on a simple example
 - Two threads
 - One calls P()
 - The other calls V()

Semaphore s(1);



Semaphore initialized to zero

- Let's see how it works on a simple example
 - Two threads
 - One calls P()
 - The other calls V()

Semaphore s(1);

P(s)

A vertical wavy line representing a semaphore. A horizontal line segment intersects the wavy line, with the label 'P(s)' positioned to its left.

V(s)

A vertical wavy line representing a semaphore. A horizontal line segment intersects the wavy line, with the label 'V(s)' positioned to its left.

Semaphore initialized to zero

- Let's see how it works on a simple example
 - Two threads
 - One calls P()
 - The other calls V()
- When V is called ahead of P, P is non-blocking

Semaphore s(0);

P(s)

V(s)

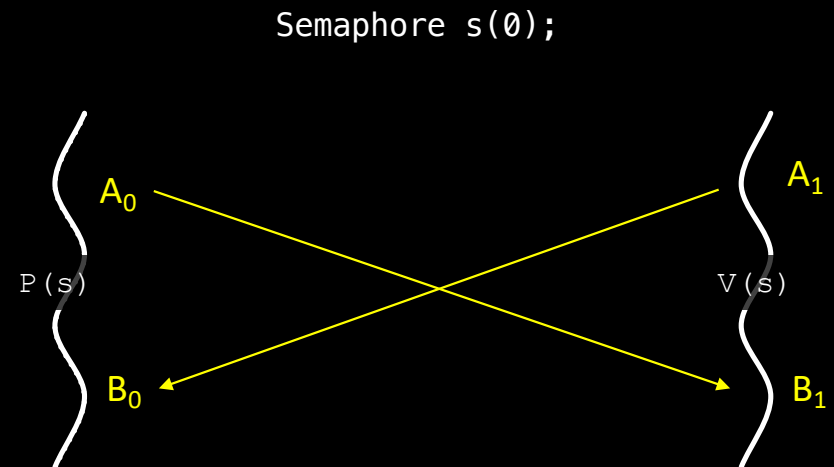
Semaphore initialized to zero

- In either case, it enforces a code dependency:
 - B_0 can only start when A_1 has completed



Semaphore initialized to zero

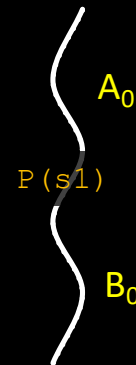
- Let us enforce:
 - B_0 can only start when A_1 has completed
- AND
- B_1 can only start when A_0 has completed



Semaphore initialized to zero

- Let us enforce:
 - B_0 can only start when A_1 has completed
- AND
- B_1 can only start when A_0 has completed

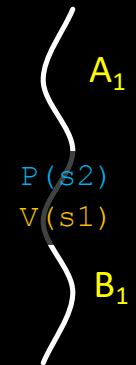
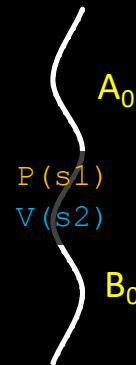
Semaphore $s1(0)$, $s2(0)$;



Semaphore initialized to zero

- Let us enforce:
 - B_0 can only start when A_1 has completed
- AND
- B_1 can only start when A_0 has completed

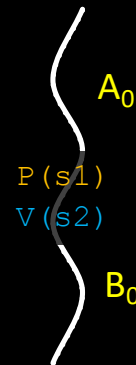
Semaphore $s1(0)$, $s2(0)$;



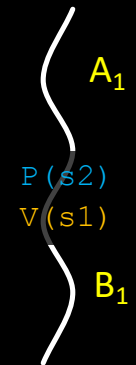
Semaphore initialized to zero

- Let us enforce:
 - B_0 can only start when A_1 has completed
- AND
- B_1 can only start when A_0 has completed

Semaphore $s1(0)$, $s2(0)$;



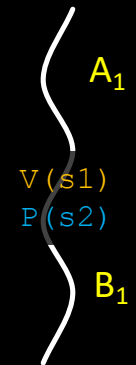
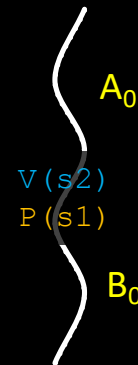
Deadlock! ☹️



Semaphore initialized to zero

- Let us enforce:
 - B_0 can only start when A_1 has completed
- AND
- B_1 can only start when A_0 has completed

Semaphore $s1(0)$, $s2(0)$;

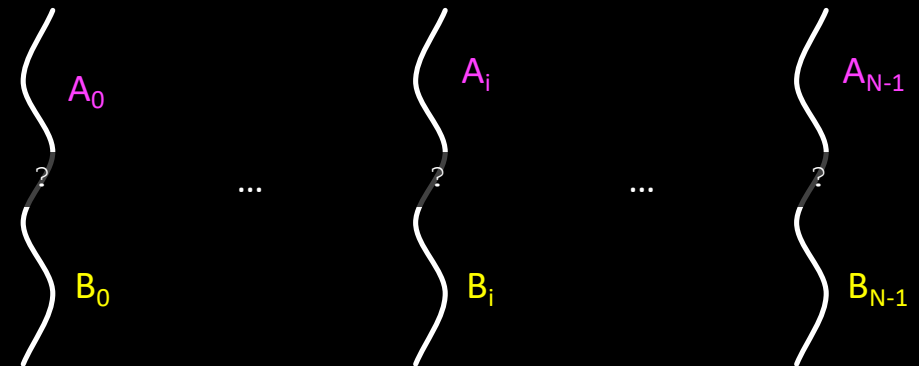


Semaphore initialized to zero

- Generalization to N processes

- B_i can only start when all A_j have completed

Semaphore $s[N](0);$

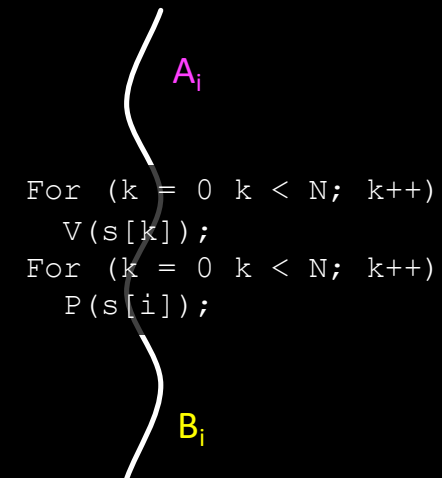


Semaphore initialized to zero

- Generalization to N processes

- B_i can only start when all A_j have completed

Semaphore $s[N](0);$



```
For (k = 0; k < N; k++)  
    V(s[k]);  
For (k = 0; k < N; k++)  
    P(s[k]);
```

Semaphore initialized to one

- Let's see how it works on a simple example
 - Three threads
 - Each one calls P(), then V()

Semaphore s(1);



Semaphore initialized to one

- Let's see how it works on a simple example
 - Three threads
 - Each one calls P(), then V()

Semaphore s(-2);



Semaphore initialized to one

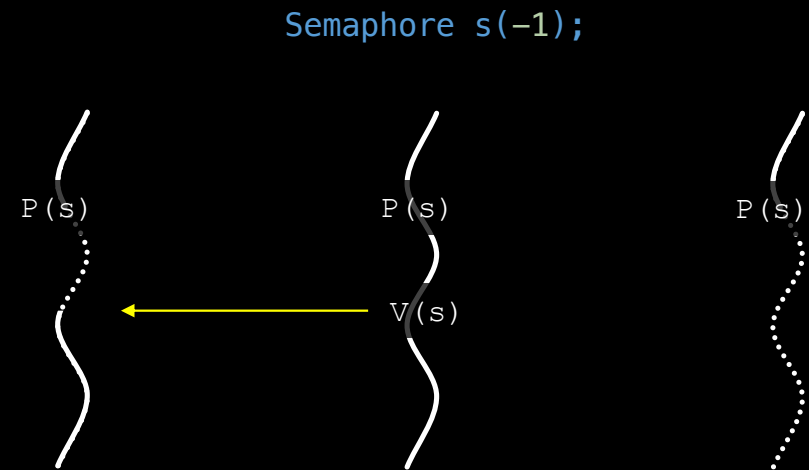
- Let's see how it works on a simple example
 - Three threads
 - Each one calls P(), then V()

Semaphore s(-2);



Semaphore initialized to one

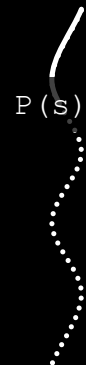
- Let's see how it works on a simple example
 - Three threads
 - Each one calls P(), then V()



Semaphore initialized to one

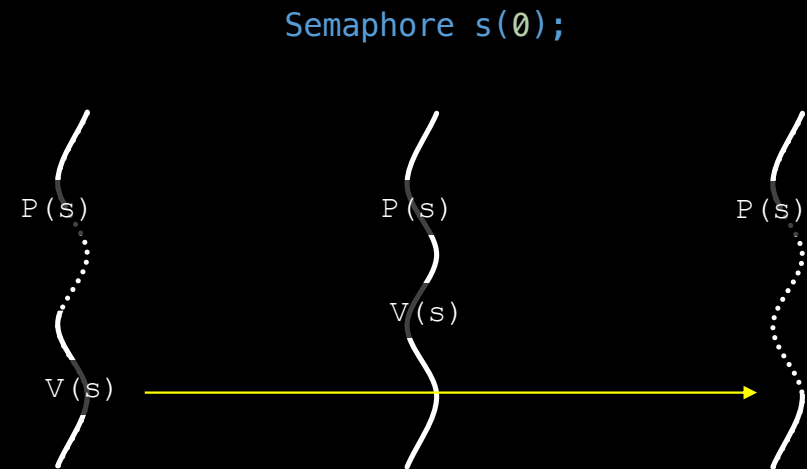
- Let's see how it works on a simple example
 - Three threads
 - Each one calls P(), then V()

Semaphore $s(-1);$



Semaphore initialized to one

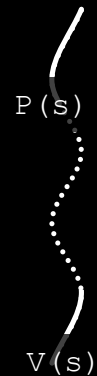
- Let's see how it works on a simple example
 - Three threads
 - Each one calls P(), then V()



Semaphore initialized to one

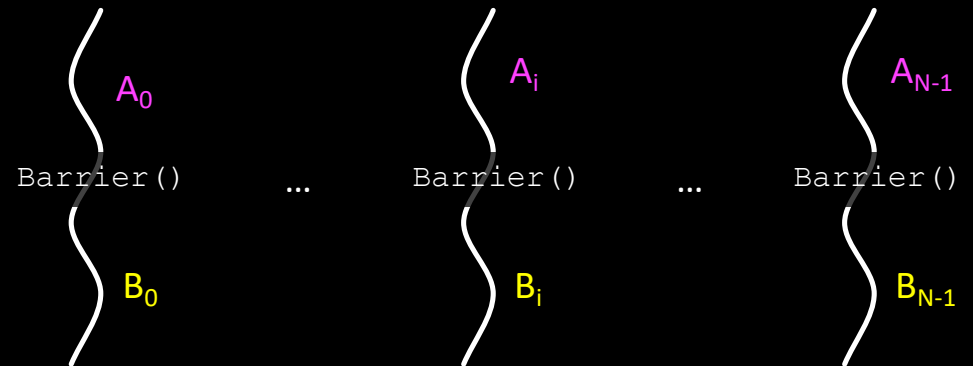
- Let's see how it works on a simple example
 - Three threads
 - Each one calls P(), then V()
- That solves the mutual exclusion problem
 - `enter_sc = P(s)`
 - `exit_sc = V(s)`

Semaphore s(1);



Back to our *rendez-vous* problem

- Can we design a solution involving less than N semaphores?



Back to our *rendez-vous* problem

- Can we design a solution involving less than N semaphores?
- Ideas
 - Block the N-1 first processes joining the barrier
 - The last process wakes everyone

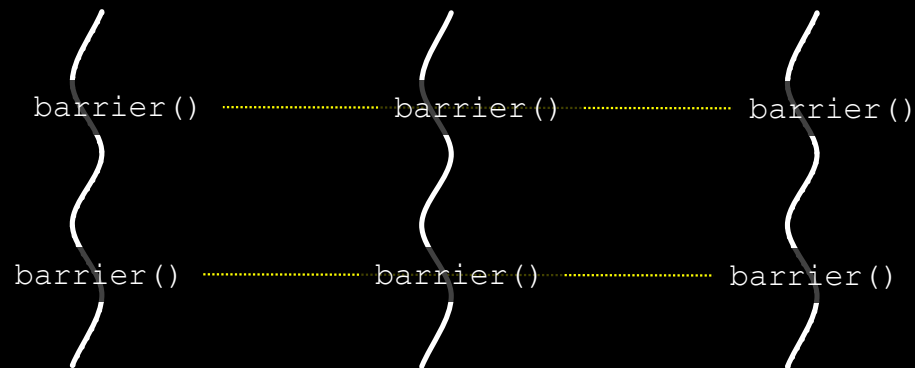
```
Semaphore wait(0), mutex(1);
int count = 0;

void barrier ()
{
    P (mutex);
    count++;
    if (count < N) {
        V (mutex);
        P(wait);
    } else {
        count = 0;
        V (mutex);
        for (int k = 0; k < N-1; k++)
            V(wait);
    }
}
```

Back to our *rendez-vous* problem

- Ok, we now have a nice barrier that should be re-usable multiple times

- Is it?

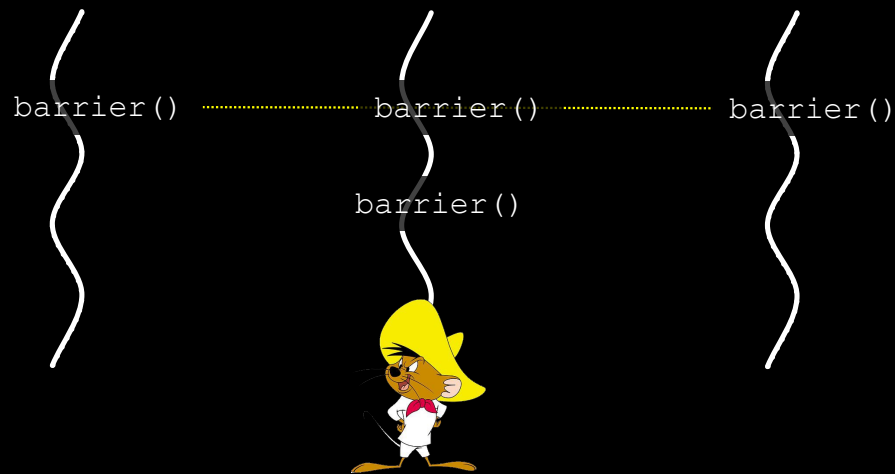


```
Semaphore wait(0), mutex(1);
int count = 0;

void barrier ()
{
    P(mutex);
    count++;
    if (count < N) {
        V(mutex);
        P(wait);
    } else {
        count = 0;
        V(mutex);
        for (int k=0; k < N-1; k++)
            V(wait);
    }
}
```

Back to our *rendez-vous* problem

- What if a thread quickly jumps from the first barrier to the second?

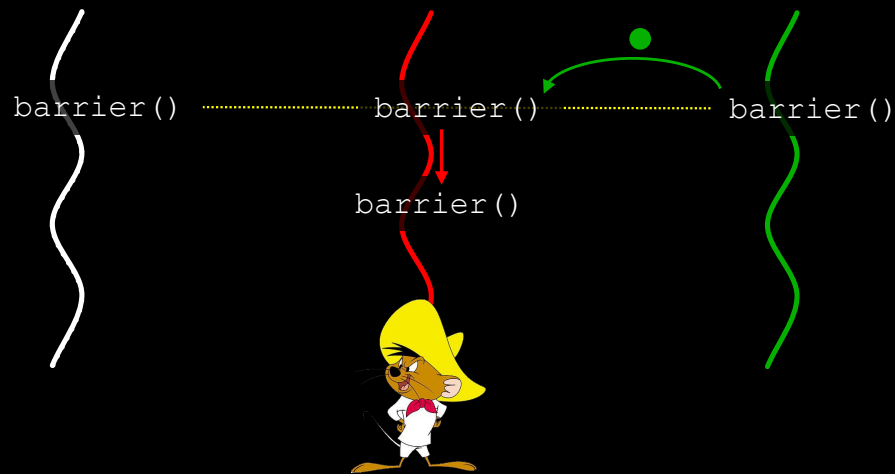


```
Semaphore wait(0), mutex(1);  
int count = 0;
```

```
void barrier ()  
{  
    P(mutex);  
    count++;  
    if (count < N) {  
        V(mutex);  
        P(wait);  
    } else {  
        count = 0;  
        V(mutex);  
        for (int k=0; k < N-1; k++)  
            V(wait);  
    }  
}
```

Back to our *rendez-vous* problem

- What if a thread quickly jumps from the first barrier to the second?

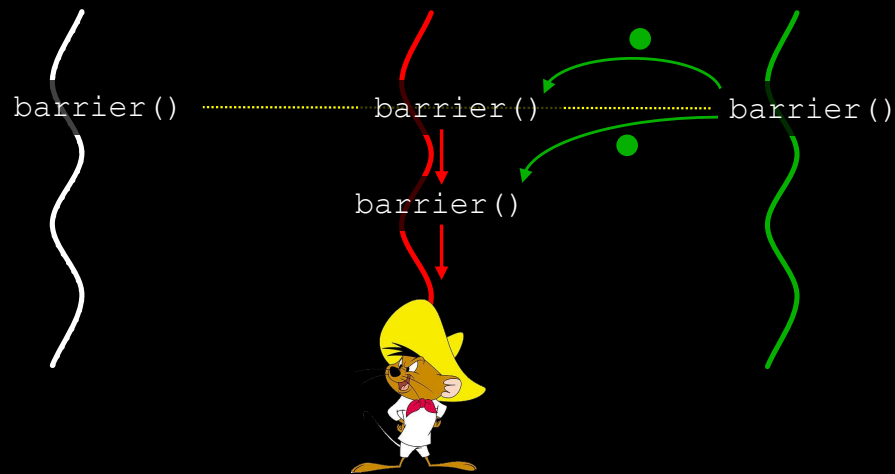


```
Semaphore wait(0), mutex(1);  
int count = 0;
```

```
void barrier ()  
{  
    P(mutex);  
    count++;  
    if (count < N) {  
        V(mutex);  
        P(wait);  
    } else {  
        count = 0;  
        V(mutex);  
        for (int k=0; k < N-1; k++)  
            V(wait);  
    }  
}
```

Back to our *rendez-vous* problem

- What if a thread quickly jumps from the first barrier to the second?



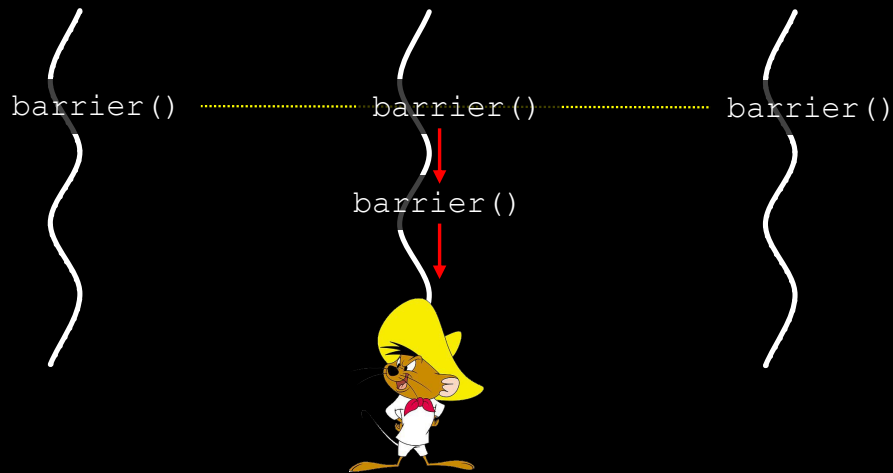
```
Semaphore wait(0), mutex(1);  
int count = 0;
```

```
void barrier ()  
{  
    P(mutex);  
    count++;  
    if (count < N) {  
        V(mutex);  
        P(wait);  
    } else {  
        count = 0;  
        V(mutex);  
        for (int k=0; k < N-1; k++)  
            V(wait);  
    }  
}
```

Back to our *rendez-vous* problem

- What if a thread quickly jumps from the first barrier to the second?

Damn!! This speedy thread can eat someone else's token... and go ahead!



```
Semaphore wait(0), mutex(1);  
int count = 0;
```

```
void barrier ()  
{  
    P(mutex);  
    count++;  
    if (count < N) {  
        V(mutex);  
        P(wait);  
    } else {  
        count = 0;  
        V(mutex);  
        for (int k=0; k < N-1; k++)  
            V(wait);  
    }  
}
```


Back to our rendez-vous problem

- What if a thread quickly jumps from the first barrier to the second?

Damn!! This speedy thread can eat someone else's token... and go ahead!

- We should probably release the mutex at the latest...

```
Semaphore wait(0), mutex(1);
int count = 0;

void barrier ()
{
    P(mutex);
    count++;
    if (count < N) {
        V(mutex);
        P(wait);
    } else {
        count = 0;
        for (int k=0; k < N-1; k++)
            V(wait);
        → V(mutex);
    }
}
```

Back to our rendez-vous problem

- What if a thread quickly jumps from the first barrier to the second?

Damn!! This speedy thread can eat someone else's token... and go ahead!

- We should probably release the mutex at the latest...

- “On a frôlé la catastrophe !”
[Hubert Bonisseur de La Bath, 2009]

```
Semaphore wait(0), mutex(1);
int count = 0;

void barrier ()
{
    P(mutex);
    count++;
    if (count < N) {
        V(mutex);
        P(wait);
    } else {
        count = 0;
        for (int k=0; k < N-1; k++)
            V(wait);
        V(mutex);
    }
}
```

Back to our rendez-vous problem

- Hmm... Unfortunately, it doesn't solve the problem 😞

```
Semaphore wait(0), mutex(1);
int count = 0;

void barrier ()
{
    P(mutex);
    count++;
    if (count < N) {
        V(mutex);
        P(wait);
    } else {
        count = 0;
        for (int k=0; k < N-1; k++)
            V(wait);
        V(mutex);
    }
}
```

Back to our rendez-vous problem

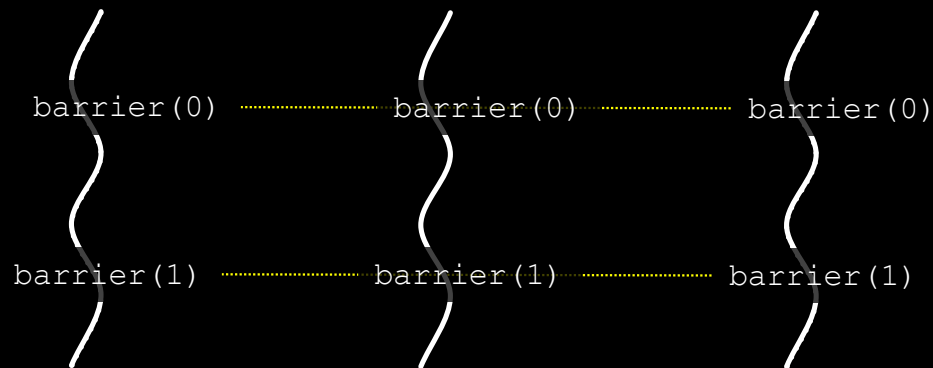
- Hmm... Unfortunately, it doesn't solve the problem 😞
- A process can have some rest before entering P(wait)
 - So we don't know WHEN he will call P(wait)...
 - The solution IS NOT to postpone V(mutex)

```
Semaphore wait(0), mutex(1);
int count = 0;

void barrier ()
{
    P(mutex);
    count++;
    if (count < N) {
        V(mutex);
        → P(wait);
    } else {
        count = 0;
        for (int k=0; k < N-1; k++)
            V(wait);
        V(mutex);
    }
}
```

Back to our *rendez-vous* problem

- Say we want to use *barrier()* only twice
 - We could duplicate all data to separate tokens
 - Two separate phases (0 and 1)

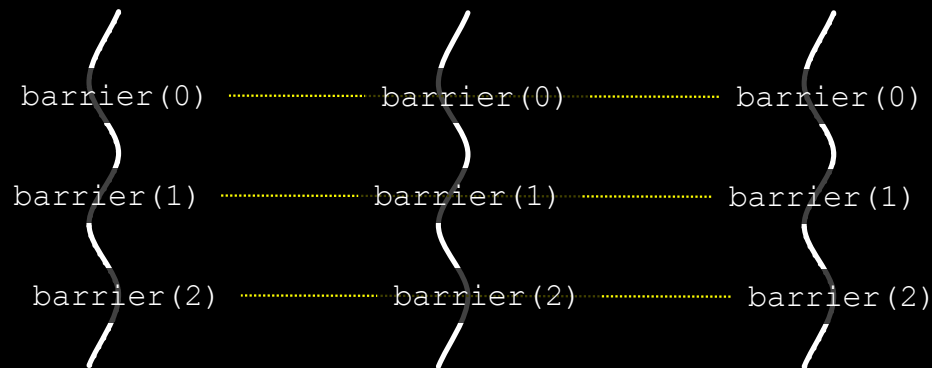


```
Semaphore wait[2](0), mutex[2](1);  
int count[2] = { 0, 0 };
```

```
void barrier (int i)  
{  
    P(mutex[i]);  
    count[i]++;  
    if (count[i] < N) {  
        V(mutex[i]);  
        P(wait[i]);  
    } else {  
        count[i] = 0;  
        V(mutex[i]);  
        for (int k=0; k < N-1; k++)  
            V(wait[i]);  
    }  
}
```

Back to our *rendez-vous* problem

- Need to use *barrier()* more than twice?

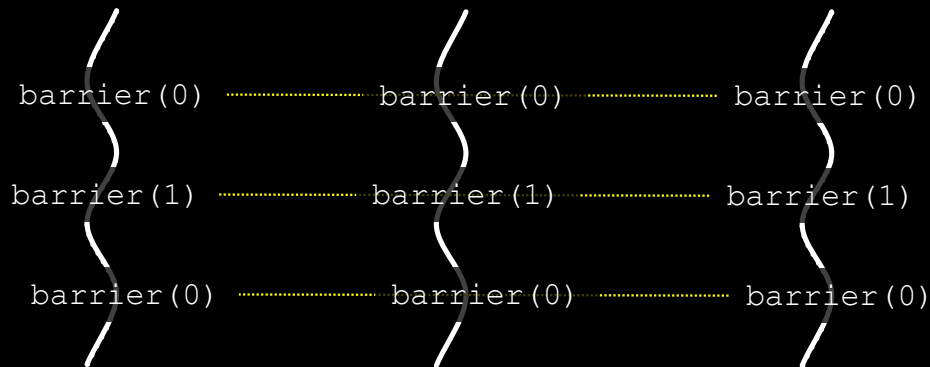


```
Semaphore wait[3](0), mutex[3](1);  
int count[3] = { 0, 0, 0 };
```

```
void barrier (int i)  
{  
    P(mutex[i]);  
    count[i]++;  
    if (count[i] < N) {  
        V(mutex[i]);  
        P(wait[i]);  
    } else {  
        count[i] = 0;  
        V(mutex[i]);  
        for (int k=0; k < N-1; k++)  
            V(wait[i]);  
    }  
}
```

Back to our *rendez-vous* problem

- Need to use *barrier()* more than twice?
 - We just need to alternate between *even* and *odd* phases

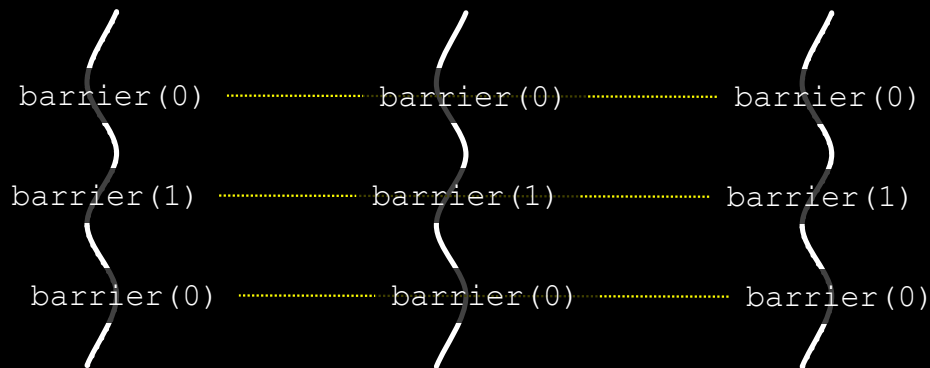


```
Semaphore wait[2](0), mutex[2](1);  
int count[2] = { 0, 0 };
```

```
void barrier (int i)  
{  
    P(mutex[i]);  
    count[i]++;  
    if (count[i] < N) {  
        V(mutex[i]);  
        P(wait[i]);  
    } else {  
        count[i] = 0;  
        V(mutex[i]);  
        for (int k=0; k < N-1; k++)  
            V(wait[i]);  
    }  
}
```

Back to our *rendez-vous* problem

- Need to use *barrier()* more than twice?
 - We just need to alternate between *even* and *odd* phases
- Threads must maintain the phase number in a “local” variable
 - Thread Local Storage (TLS)

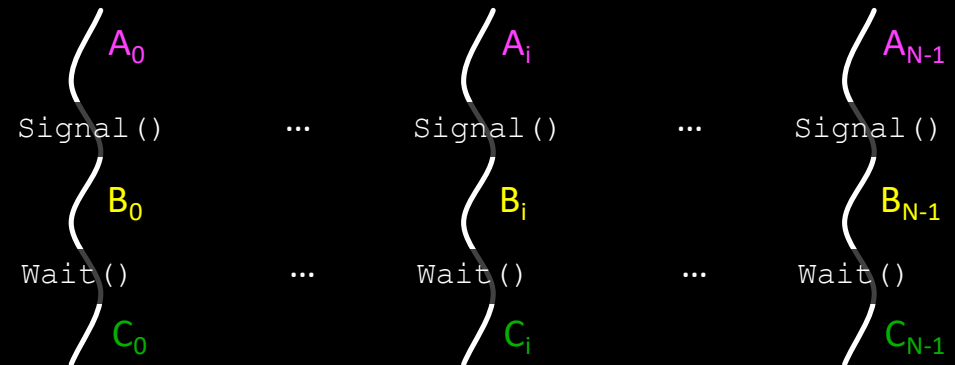


```
thread_local int phase = 0;
```

```
barrier()  
{  
...  
    phase = 1 - phase;  
}
```

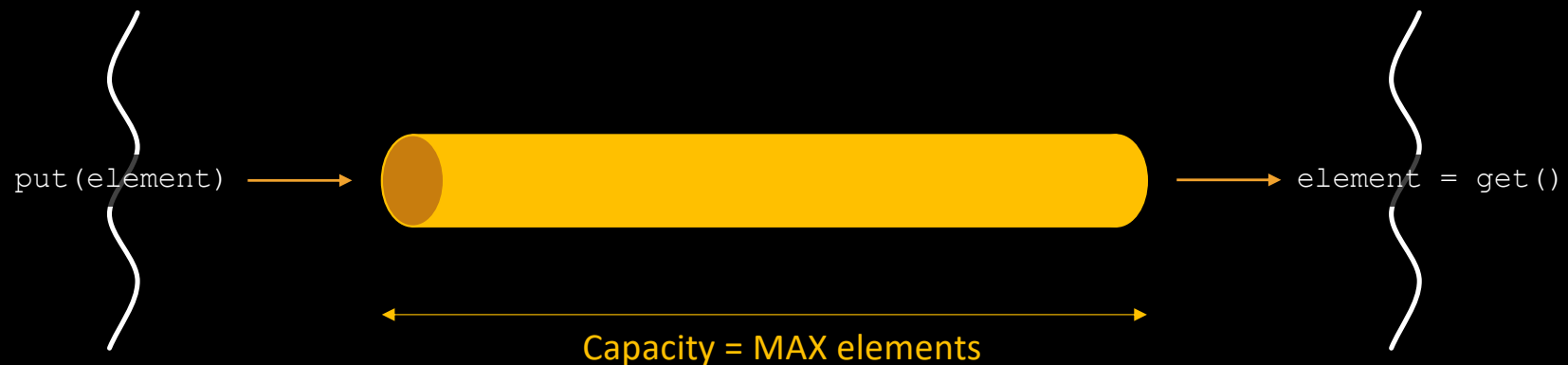

Back to our *rendez-vous* problem

- Okay, now we have a classy barrier we can be proud of!
- In some applications, two-phase barriers are more useful
 - C_k must start after A_i
 - B_j has no dependency on A_i and C_k
 - `Signal()` is non-blocking
 - `Wait()` blocks until all A_i have completed



Producers/Consumers problem

- Shared FIFO structure (initially empty)
 - `put()` and `get()` are not synchronized
 - When FIFO is full, `put` raises an error / when FIFO is empty, `get` raises an error
 - We assume `put()` and `get()` can be performed simultaneously

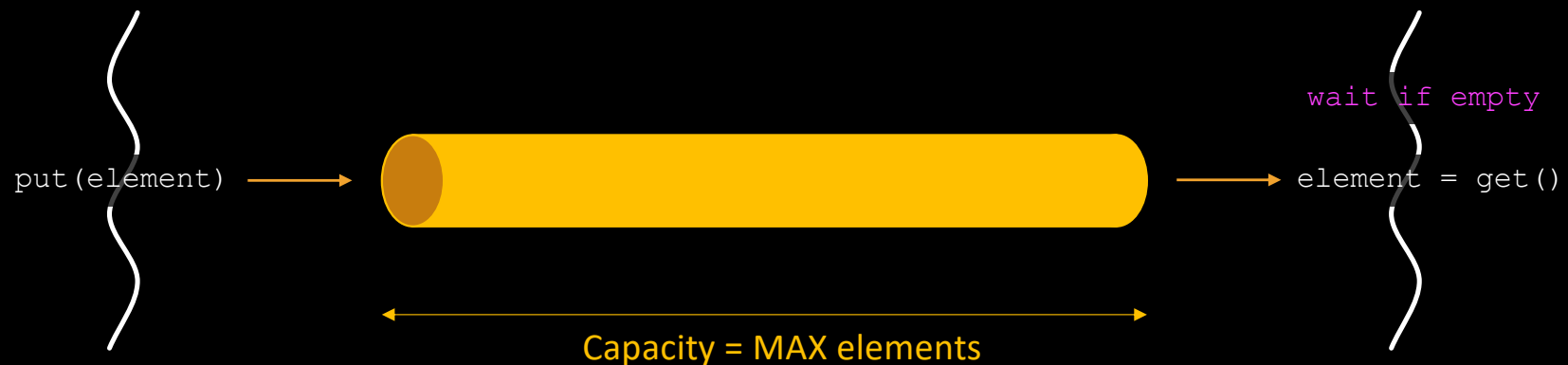


Producers/Consumers problem

- Let's look at the Consumers side first

```
semaphore cons(?);
```

- Do we need to count elements?

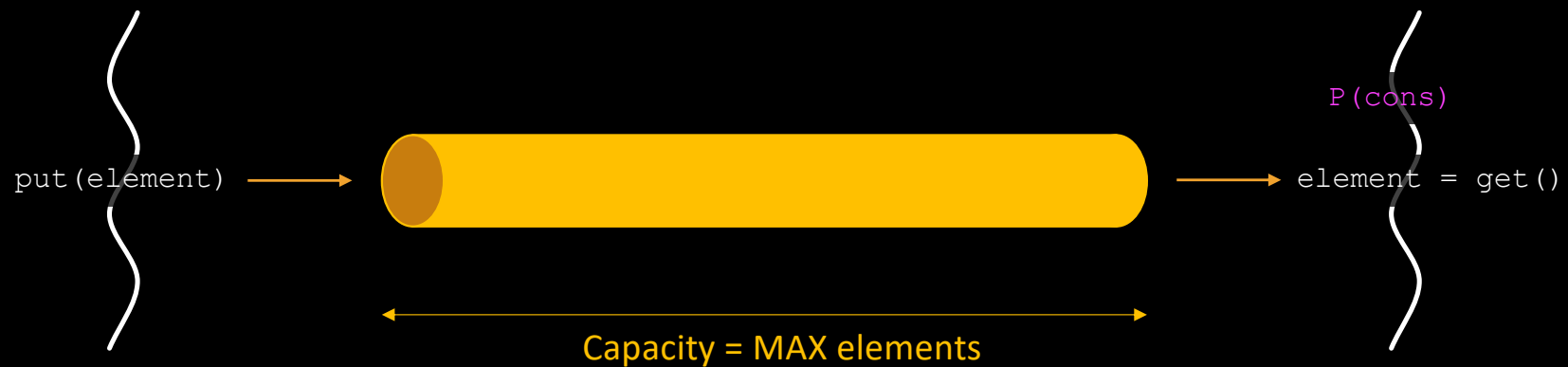


Producers/Consumers problem

- Let's look at the Consumers side first

```
semaphore cons(0);
```

- Do we need to count elements?

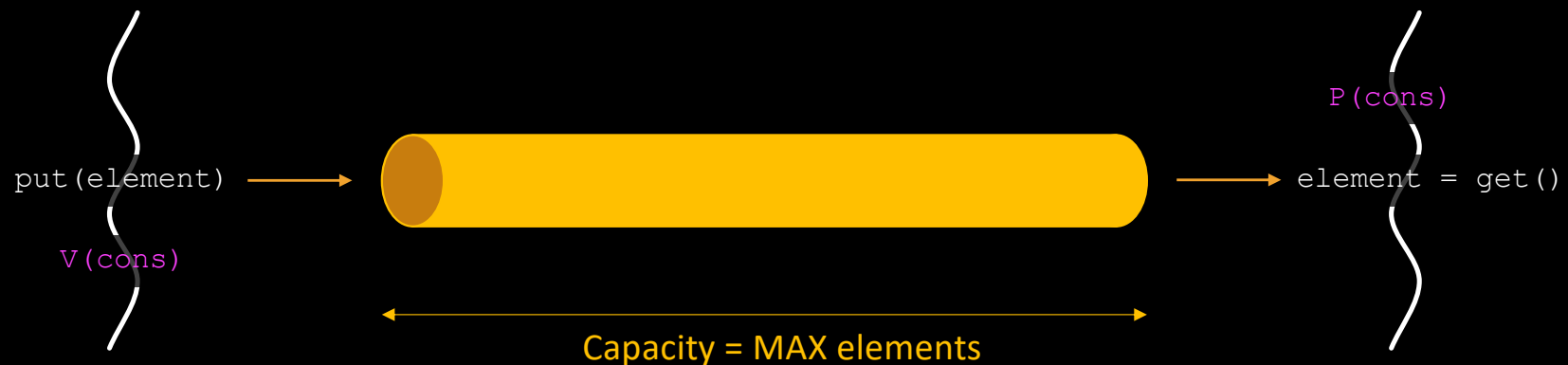


Producers/Consumers problem

- Let's look at the Consumers side first

semaphore `cons(0)`;

- N elements in FIFO
=> N tokens in `cons`

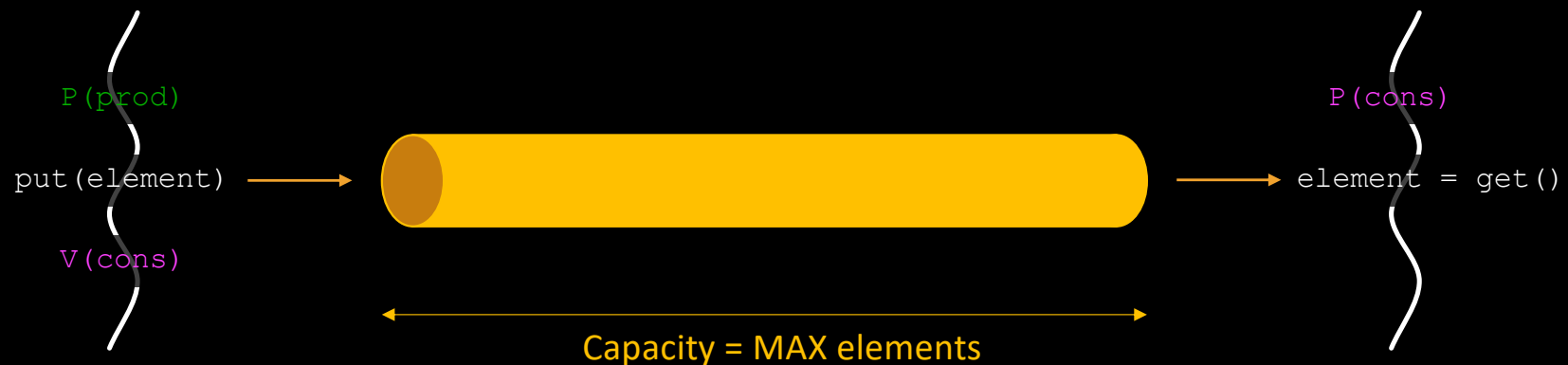


Producers/Consumers problem

- Let's look at the Consumers side first

```
semaphore cons(0), prod(?);
```

- N elements in FIFO
=> N tokens in **cons**

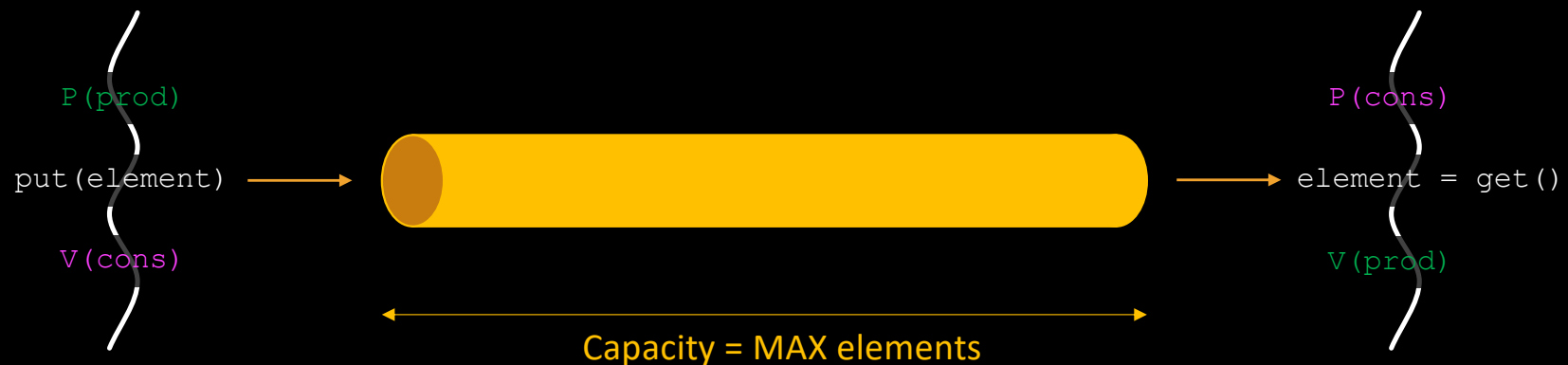


Producers/Consumers problem

- Let's look at the Consumers side first

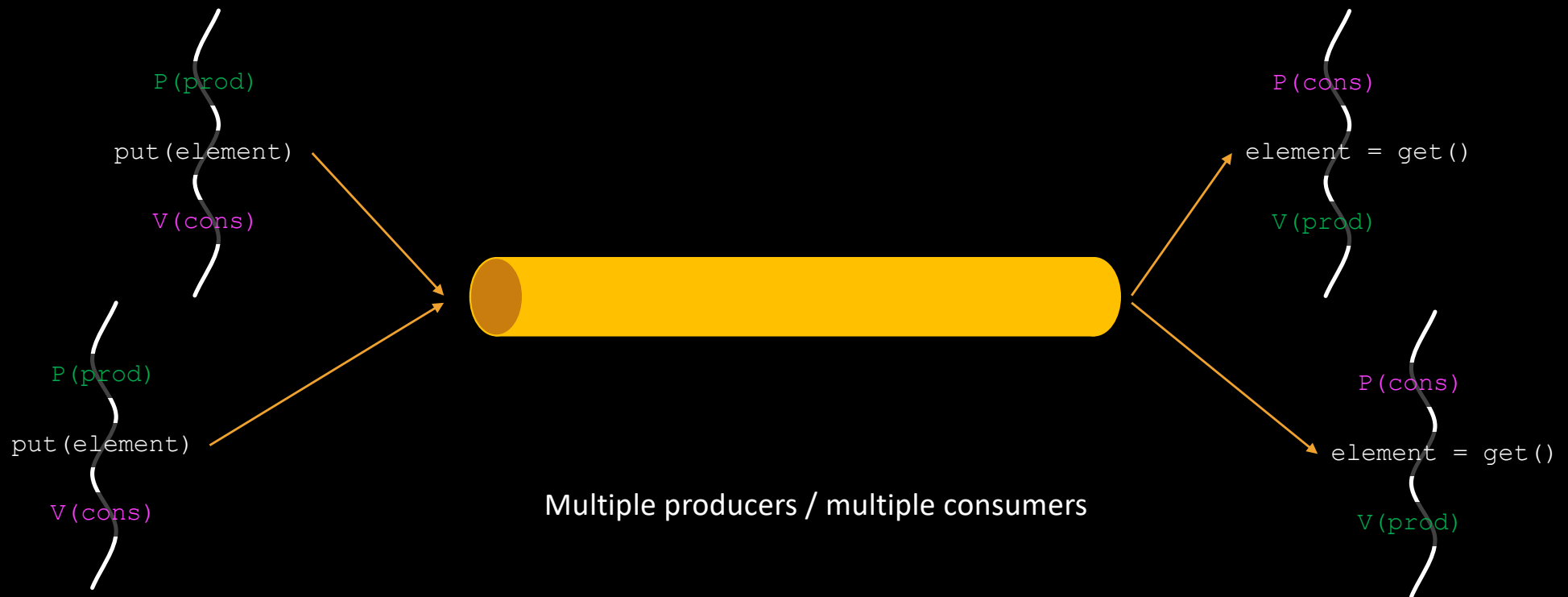
```
semaphore cons(0), prod(MAX);
```

- N free slots in FIFO
=> N tokens in `prod`



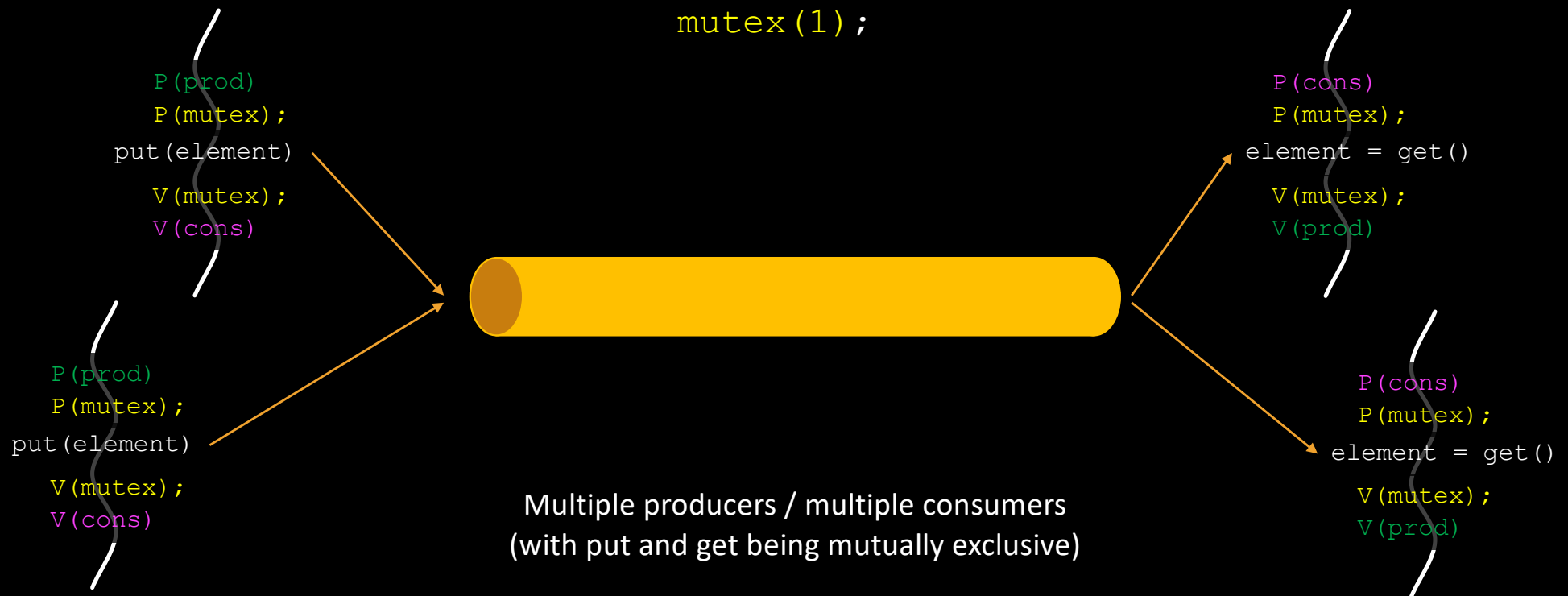
Producers/Consumers problem

```
semaphore cons(0), prod(MAX);
```



Producers/Consumers problem

```
semaphore cons(0), prod(MAX)  
mutex(1);
```



Producers/Consumers problem

```
semaphore cons(0), prod(MAX)  
mutexC(1), mutexP(1);
```



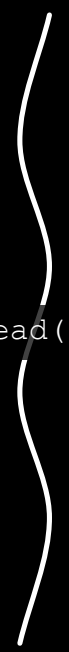
Reader/Writer problem

- A shared data structure is accessed by two types of processes
 - Readers
 - These processes only read the data
 - Writers
 - These processes can modify the data

Reader/Writer problem

- A shared data structure is accessed by two types of processes
 - Readers
 - These processes only read the data
 - Writers
 - These processes can modify the data
- Consequently, we want to enforce the following rules
 - Multiple Readers can access data concurrently (R // R // ... // R is ok)
 - Writers are mutually exclusive (no W // W)
 - Writers and Readers are mutually exclusive (no W // R)

Reader/Writer problem

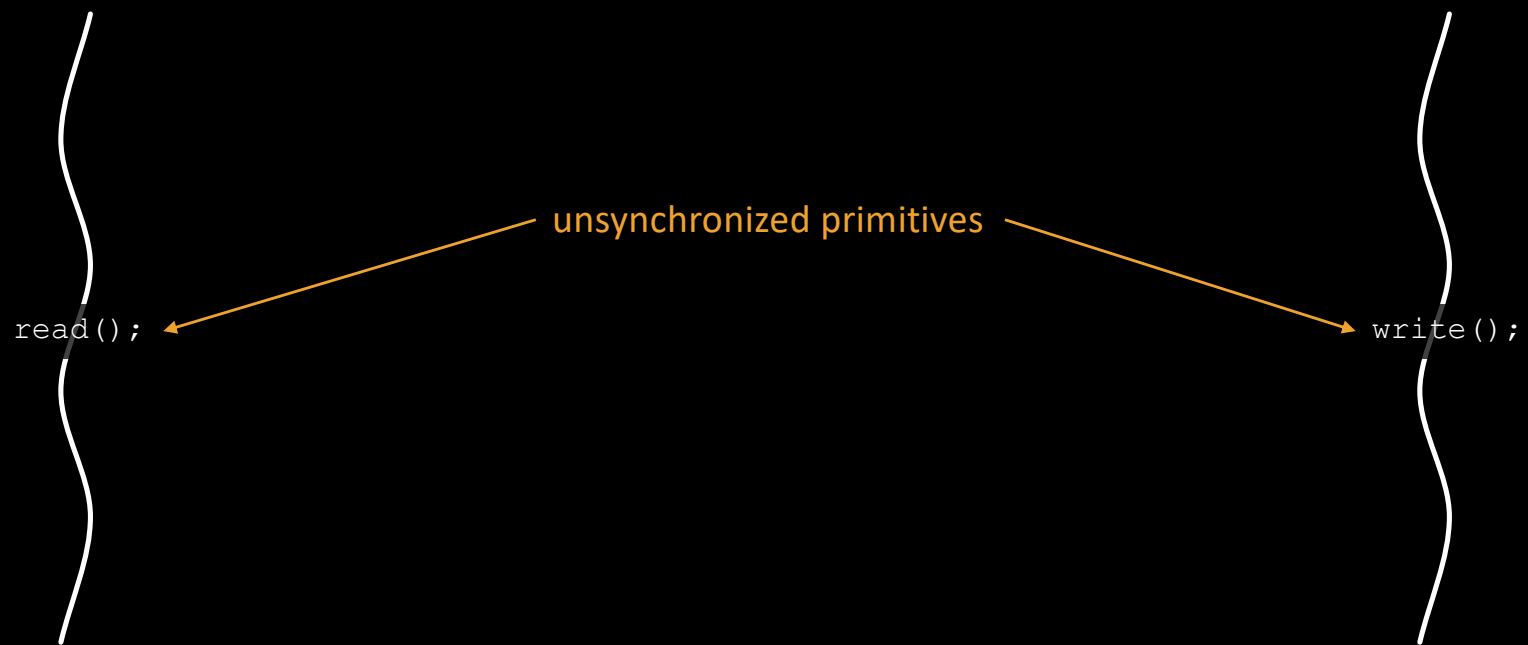


`read();`



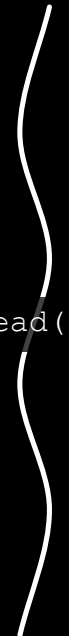
`write();`

Reader/Writer problem



Reader/Writer problem

Semaphore `writeToken(?)`;



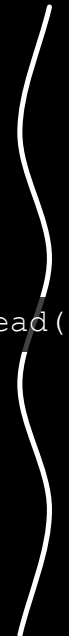
`read();`



`write();`

Reader/Writer problem

Semaphore `writeToken(1);`



```
read();
```

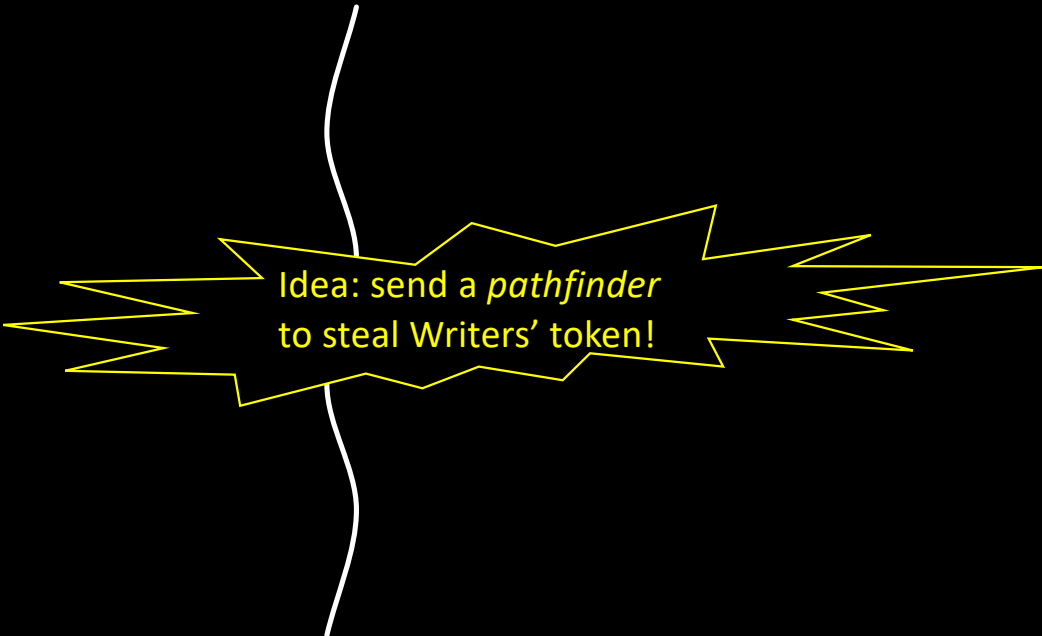
```
P(writeToken);
```

```
write();
```

```
V(writeToken);
```


Reader/Writer problem

Semaphore `writeToken(1);`



Idea: send a *pathfinder*
to steal Writers' token!



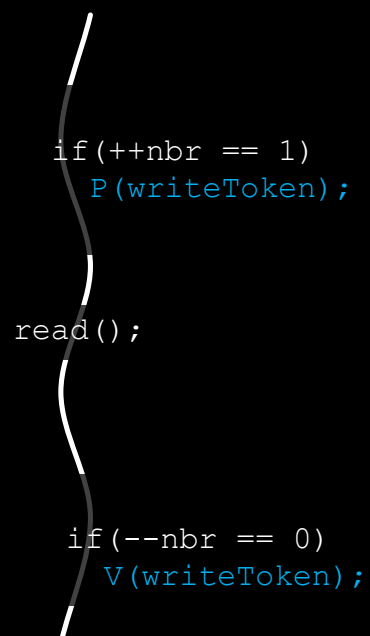
```
P(writeToken);
```

```
write();
```

```
V(writeToken);
```

Reader/Writer problem

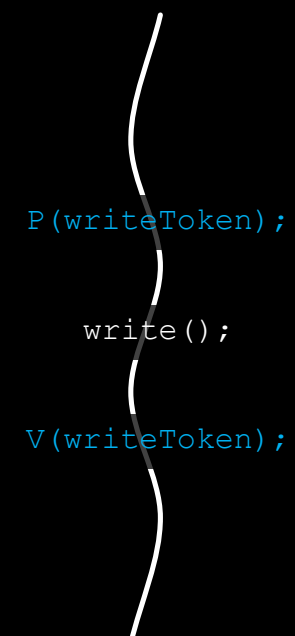
Semaphore `writeToken(1);`
`int nbr = 0;`



```
if(++nbr == 1)
    P(writeToken);

read();

if(--nbr == 0)
    V(writeToken);
```



```
P(writeToken);

write();

V(writeToken);
```

Reader/Writer problem

Semaphore `writeToken(1);`
`int nbr = 0;`

```
if(++nbr == 1)
    P(writeToken);

read();

if(--nbr == 0)
    V(writeToken);
```

“Oulala!”

```
P(writeToken);

write();

V(writeToken);
```

Reader/Writer problem

Semaphore `writeToken(1);`
`int nbr = 0;`
Semaphore `mutexR(1);`

```
P(mutexR);  
if (++nbr == 1) // pathfinder  
    P(writeToken);  
V(mutexR);  
  
read();  
  
P(mutexR);  
if (--nbr == 0) // last to leave  
    V(writeToken);  
V(mutexR);
```

```
P(writeToken);  
  
write();  
  
V(writeToken);
```

Reader/Writer problem

Semaphore `writeToken(1);`
`int nbr = 0;`
Semaphore `mutexR(1);`

```
P(mutexR);  
if (++nbr == 1) // pathfinder  
    P(writeToken);  
V(mutexR);  
  
read();  
  
P(mutexR);  
if (--nbr == 0) // last to leave  
    V(writeToken);  
V(mutexR);
```

Re-“Oulala!”

```
P(writeToken);  
  
write();  
  
V(writeToken);
```

Reader/Writer problem

Semaphore `writeToken(1);`
`int nbr = 0;`
Semaphore `mutexR(1);`

```
P(mutexR);  
if (++nbr == 1) // pathfinder  
    P(writeToken);  
V(mutexR);  
  
read();  
  
P(mutexR);  
if (--nbr == 0) // last to leave  
    V(writeToken);  
V(mutexR);
```

In this case, it's ok!

```
P(writeToken);  
  
write();  
  
V(writeToken);
```

Reader/Writer problem

Semaphore `writeToken(1);`
`int nbr = 0;`
Semaphore `mutexR(1);`

```
P(mutexR);  
if (++nbr == 1) // pathfinder  
    P(writeToken);  
V(mutexR);  
  
read();  
  
P(mutexR);  
if (--nbr == 0) // last to leave  
    V(writeToken);  
V(mutexR);
```

Can we guarantee this won't block forever?

```
P(writeToken);  
  
write();  
  
V(writeToken);
```

Reader/Writer problem

Semaphore `writeToken(1);`
`int nbr = 0;`
Semaphore `mutexR(1);`

```
P(mutexR);  
if (++nbr == 1) // pathfinder  
    P(writeToken);  
V(mutexR);  
  
read();  
  
P(mutexR);  
if (--nbr == 0) // last to leave  
    V(writeToken);  
V(mutexR);
```

Is this algorithm fair?

```
P(writeToken);  
  
write();  
  
V(writeToken);
```


Reader/Writer problem

Semaphore `writeToken(1);`
`int nbr = 0;`
Semaphore `mutexR(1);`

```
P(mutexR);  
if (++nbr == 1) // pathfinder  
    P(writeToken);  
V(mutexR);  
  
read();  
  
P(mutexR);  
if (--nbr == 0) // last to leave  
    V(writeToken);  
V(mutexR);
```

This algorithm is totally **unfair!**

Readers and Writers do not accumulate
on the same semaphores...

```
P(writeToken);  
  
write();  
  
V(writeToken);
```

Reader/Writer problem

Semaphore `writeToken(1);`
`int nbr = 0;`
Semaphore `mutexR(1);`
Semaphore `waitingRoom(1);`

```
P(waitingRoom);  
P(mutexR);  
if (++nbr == 1) // pathfinder  
    P(writeToken);  
V(mutexR);  
V(waitingRoom);  
  
    read();  
  
P(mutexR);  
if (--nbr == 0) // last to leave  
    V(writeToken);  
V(mutexR);
```

```
P(waitingRoom);  
P(writeToken);  
V(waitingRoom);  
  
    write();  
  
V(writeToken);
```

Reader/Writer problem

Semaphore `writeToken(1);`
`int nbr = 0;`
Semaphore `mutexR(1);`
Semaphore `waitingRoom(1);`

```
P(waitingRoom);  
P(mutexR);  
if (++nbr == 1) // pathfinder  
    P(writeToken);  
V(mutexR);  
V(waitingRoom);  
  
read();  
  
P(mutexR);  
if (--nbr == 0) // last to leave  
    V(writeToken);  
V(mutexR);
```

R

W

R

R

R



```
P(waitingRoom);  
P(writeToken);  
V(waitingRoom);  
  
write();  
  
V(writeToken);
```

Reader/Writer problem

```
P(waitingRoom);  
P(mutexR);  
if (++nbr == 1) // pathfinder  
    P(writeToken);  
V(mutexR);  
V(waitingRoom);  
  
read();  
  
P(mutexR);  
if (--nbr == 0) // last to leave  
    V(writeToken);  
V(mutexR);
```

```
Semaphore writeToken(1);  
int nbr = 0;  
Semaphore mutexR(1);  
Semaphore waitingRoom(1);
```

R

W

R

R

R



```
P(waitingRoom);  
P(writeToken);  
V(waitingRoom);  
  
write();  
  
V(writeToken);
```

Reader/Writer problem

```
P(waitingRoom);
P(mutexR);
if (++nbr == 1) // pathfinder
    P(writeToken);
V(mutexR);
V(waitingRoom);

read();

P(mutexR);
if (--nbr == 0) // last to leave
    V(writeToken);
V(mutexR);
```

Semaphore `writeToken(1);`
int `nbr = 0;`
Semaphore `mutexR(1);`
Semaphore `waitingRoom(1);`

R

W

R

R



R ●

```
P(waitingRoom);
P(writeToken);
V(waitingRoom);

write();

V(writeToken);
```

Reader/Writer problem

Semaphore `writeToken(1);`
`int nbr = 0;`
Semaphore `mutexR(1);`
Semaphore `waitingRoom(1);`

```
P(waitingRoom);  
P(mutexR);  
if (++nbr == 1) // pathfinder  
    P(writeToken);  
V(mutexR);  
V(waitingRoom);  
  
read();  
  
P(mutexR);  
if (--nbr == 0) // last to leave  
    V(writeToken);  
V(mutexR);
```

R ●

R

W

R

R

```
P(waitingRoom);  
P(writeToken);  
V(waitingRoom);  
  
write();  
  
V(writeToken);
```

Reader/Writer problem

Semaphore `writeToken(1);`
`int nbr = 0;`
Semaphore `mutexR(1);`
Semaphore `waitingRoom(1);`

```
P(waitingRoom);  
P(mutexR);  
if (++nbr == 1) // pathfinder  
    P(writeToken);  
V(mutexR);  
V(waitingRoom);  
  
read();  
  
P(mutexR);  
if (--nbr == 0) // last to leave  
    V(writeToken);  
V(mutexR);
```

R

W

R



R ●
R

```
P(waitingRoom);  
P(writeToken);  
V(waitingRoom);  
  
write();  
  
V(writeToken);
```

Reader/Writer problem

Semaphore `writeToken(1);`
`int nbr = 0;`
Semaphore `mutexR(1);`
Semaphore `waitingRoom(1);`

```
P(waitingRoom);  
P(mutexR);  
if (++nbr == 1) // pathfinder  
    P(writeToken);  
V(mutexR);  
V(waitingRoom);  
  
read();  
  
P(mutexR);  
if (--nbr == 0) // last to leave  
    V(writeToken);  
V(mutexR);
```

R

R

W

R

```
P(waitingRoom);  
P(writeToken);  
V(waitingRoom);  
  
write();  
  
V(writeToken);
```


Reader/Writer problem

```
P(waitingRoom);  
P(mutexR);  
if (++nbr == 1) // pathfinder  
    P(writeToken);  
V(mutexR);  
V(waitingRoom);  
  
read();  
  
P(mutexR);  
if (--nbr == 0) // last to leave  
    V(writeToken);  
V(mutexR);
```

```
Semaphore writeToken(1);  
int nbr = 0;  
Semaphore mutexR(1);  
Semaphore waitingRoom(1);
```

R

R

W

R

```
P(waitingRoom);  
P(writeToken);  
V(waitingRoom);  
  
write();  
  
V(writeToken);
```

Reader/Writer problem

```
P(waitingRoom);  
P(mutexR);  
if (++nbr == 1) // pathfinder  
    P(writeToken);  
V(mutexR);  
V(waitingRoom);  
  
read();  
  
P(mutexR);  
if (--nbr == 0) // last to leave  
    V(writeToken);  
V(mutexR);
```

```
Semaphore writeToken(1);  
int nbr = 0;  
Semaphore mutexR(1);  
Semaphore waitingRoom(1);
```

R R



R
W

```
P(waitingRoom);  
P(writeToken);  
V(waitingRoom);  
  
write();  
  
V(writeToken);
```

Reader/Writer problem

R

Semaphore `writeToken(1);`
`int nbr = 0;`
Semaphore `mutexR(1);`
Semaphore `waitingRoom(1);`

```
P(waitingRoom);  
P(mutexR);  
if (++nbr == 1) // pathfinder  
    P(writeToken);  
V(mutexR);  
V(waitingRoom);  
  
read();  
  
P(mutexR);  
if (--nbr == 0) // last to leave  
    V(writeToken);  
V(mutexR);
```

W

R R

```
P(waitingRoom);  
P(writeToken);  
V(waitingRoom);  
  
write();  
  
V(writeToken);
```

Reader/Writer problem

R

```
Semaphore writeToken(1);  
int nbr = 0;  
Semaphore mutexR(1);  
Semaphore waitingRoom(1);
```

```
P(waitingRoom);  
P(mutexR);  
if (++nbr == 1) // pathfinder  
    P(writeToken);  
V(mutexR);  
V(waitingRoom);  
  
read();  
  
P(mutexR);  
if (--nbr == 0) // last to leave  
    V(writeToken);  
V(mutexR);
```

W

R

```
P(waitingRoom);  
P(writeToken);  
V(waitingRoom);  
  
write();  
  
V(writeToken);
```

Reader/Writer problem

R

Semaphore `writeToken(1);`
`int nbr = 0;`
Semaphore `mutexR(1);`
Semaphore `waitingRoom(1);`

```
P(waitingRoom);  
P(mutexR);  
if (++nbr == 1) // pathfinder  
    P(writeToken);  
V(mutexR);  
V(waitingRoom);  
  
read();  
  
P(mutexR);  
if (--nbr == 0) // last to leave  
    V(writeToken);  
V(mutexR);
```

W

```
P(waitingRoom);  
P(writeToken);  
V(waitingRoom);  
  
write();  
  
V(writeToken);
```

Reader/Writer problem

R

```
Semaphore writeToken(1);  
int nbr = 0;  
Semaphore mutexR(1);  
Semaphore waitingRoom(1);
```

```
P(waitingRoom);  
P(mutexR);  
if (++nbr == 1) // pathfinder  
    P(writeToken);  
V(mutexR);  
V(waitingRoom);  
  
read();  
  
P(mutexR);  
if (--nbr == 0) // last to leave  
    V(writeToken);  
V(mutexR);
```



W ●

```
P(waitingRoom);  
P(writeToken);  
V(waitingRoom);  
  
write();  
  
V(writeToken);
```

Reader/Writer problem

Semaphore `writeToken(1);`
`int nbr = 0;`
Semaphore `mutexR(1);`
Semaphore `waitingRoom(1);`

```
P(waitingRoom);  
P(mutexR);  
if (++nbr == 1) // pathfinder  
    P(writeToken);  
V(mutexR);  
V(waitingRoom);  
  
read();  
  
P(mutexR);  
if (--nbr == 0) // last to leave  
    V(writeToken);  
V(mutexR);
```

R

W

```
P(waitingRoom);  
P(writeToken);  
V(waitingRoom);  
  
write();  
  
V(writeToken);
```

Reader/Writer problem

Semaphore `writeToken(1);`
`int nbr = 0;`
Semaphore `mutexR(1);`
Semaphore `waitingRoom(1);`

```
P(waitingRoom);  
P(mutexR);  
if (++nbr == 1) // pathfinder  
    P(writeToken);  
V(mutexR);  
V(waitingRoom);  
  
read();  
  
P(mutexR);  
if (--nbr == 0) // last to leave  
    V(writeToken);  
V(mutexR);
```

R

```
P(waitingRoom);  
P(writeToken);  
V(waitingRoom);  
  
write();  
  
V(writeToken);
```


Reader/Writer problem

Semaphore `writeToken(1);`
`int nbr = 0;`
Semaphore `mutexR(1);`
Semaphore `waitingRoom(1);`

```
P(waitingRoom);  
P(mutexR);  
if (++nbr == 1) // pathfinder  
    P(writeToken);  
V(mutexR);  
V(waitingRoom);  
  
read();  
  
P(mutexR);  
if (--nbr == 0) // last to leave  
    V(writeToken);  
V(mutexR);
```

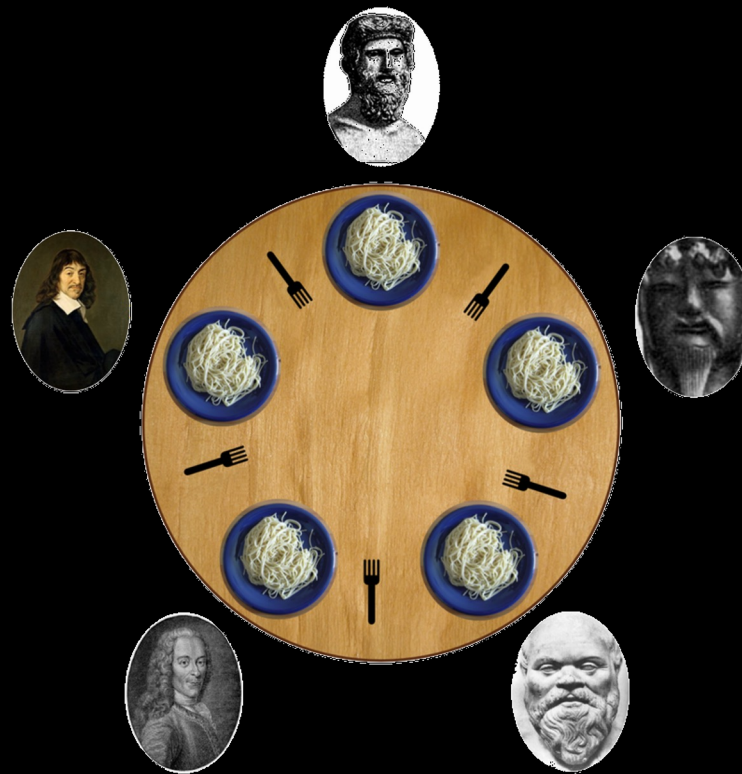
R

```
P(waitingRoom);  
P(writeToken);  
V(waitingRoom);  
  
write();  
  
V(writeToken);
```

Recap

- Most popular synchronization schemes
 - Mutual Exclusion
 - Execution Dependencies
 - Barriers
 - Producers-Consumers
 - Readers-Writers
- You'll find many, many other (theoretical) problems in OS books

The Dining Philosophers problem





Hoare Monitors (1973)

- Synchronization construct initially proposed in the context of OO Languages

- Simula67
- Concurrent Pascal

- Later used in

- Ada
- Java
- Operating Systems

```
Monitor class m {  
    private int i = ...;  
  
    public method f() {  
        ... i++; ...  
    }  
  
    public method g() {  
        ... i--; ...  
    }  
}
```

Hoare Monitors (1973)

- Principles

- Implicit *lock* associated to a monitor
 - Methods are mutually exclusive

```
Monitor class m {  
    private int i = ...;  
  
    public method f() {  
        ... i++; ...  
    }  
  
    public method g() {  
        ... i--; ...  
    }  
}
```

$\approx V(mutex)$

$\approx P(mutex)$

Hoare Monitors (1973)

- Principles

- Implicit *lock* associated to a monitor
 - Methods are mutually exclusive
- *Conditions* are synchronizing objects
 - Think about a list of waiting processes
- Wait
 - Unconditionally blocks
- Signal / Bcast
 - Wakes up one (all) process, or do nothing

```
Monitor class m {  
    private int i = ...;  
    private condition c;  
  
    public method f() {  
        if (...)   
            wait (c);  
        ...  
    }  
  
    public method g() {  
        ...  
        signal (c);  
        ...  
    }  
}
```

Hoare Monitors (1973)

- Conditions $\neq$ Semaphores

- No tokens saved
 - Signal does nothing if no process is waiting
- The wait operation is subtle
 - Sleep in the condition list

```
Monitor class m {  
    private int i = ...;  
    private condition c;  
  
    public method f() {  
        if (...)   
            wait (c);  
  
        ...  
    }  
  
    public method g() {  
        ...  
        signal (c);  
        ...  
    }  
}
```


Hoare Monitors (1973)

- Conditions $\neq$ Semaphores

- No tokens saved
 - Signal does nothing if no process is waiting
- The wait operation is subtle
 - Release the implicit lock
 - Sleep in the condition list

```
Monitor class m {  
    private int i = ...;  
    private condition c;  
  
    public method f() {  
        if (...)   
            wait (c);  
  
        ...  
    }  
  
    public method g() {  
        ...  
        signal (c);  
        ...  
    }  
}
```

Hoare Monitors (1973)

- Conditions $\neq$ Semaphores

- No tokens saved
 - Signal does nothing if no process is waiting
- The wait operation is subtle
 - Release the implicit lock
 - Sleep in the condition list
 - Re-acquire the implicit lock upon wake up

```
Monitor class m {  
    private int i = ...;  
    private condition c;  
  
    public method f() {  
        if (...)  
            wait (c);  
  
        ...  
    }  
  
    public method g() {  
        ...  
        signal (c);  
        ...  
    }  
}
```

Hoare Monitors (1973)

- Conditions $\neq$ Semaphores

- No tokens saved
 - Signal does nothing if no process is waiting

- The wait operation is subtle

- atomic** {
- Release the implicit lock
 - Sleep in the condition list
 - Re-acquire the implicit lock upon wake up

```
Monitor class m {  
    private int i = ...;  
    private condition c;  
  
    public method f() {  
        if (...)   
            wait (c);  
        ...  
    }  
  
    public method g() {  
        ...  
        signal (c);  
        ...  
    }  
}
```

Hoare Monitors (1973)

- Monitors and conditions were initially introduced as language constructs
- In operating systems or applications, the API is slightly different
 - `mutex_t` type
 - `mutex_lock (mutex_t *m)`
 - `mutex_unlock (mutex_t *m)`
 - `cond_t` type
 - `cond_wait (cond_t *c, mutex_t *m)`
 - `cond_signal (cond_t *c)`
 - `cond_bcast (cond_t *c)`

Hoare Monitors (1973)

```
Monitor class m {  
    private int i = ...;  
    private condition c;  
  
    public method f() {  
        if (...)  
            wait (c);  
        ...  
    }  
  
    public method g() {  
        ...  
        signal (c);  
        ...  
    }  
}
```



```
mutex_t m;  
cond_t c;  
  
void f() {  
    mutex_lock (&m);  
    if (...)  
        cond_wait (&c, &m);  
    mutex_unlock (&m);  
}  
  
void g() {  
    mutex_lock (&m);  
    ...  
    cond_signal (&c);  
    ...  
    mutex_unlock (&m);  
}
```

Hoare Monitors (1973)

- Caveats

- No `cond_wait` outside monitor (i.e. [lock – unlock] block)
 - “No `cond_signal/bcast`” is also a good idea
- Mutexes have an *owner*
 - `mutex_unlock` can only be done by the owner

- Recommendation

- Since conditions are “token-less”, more variables are usually needed (in comparison to using semaphores)

Back to our *rendez-vous* problem

- Solution with Hoare Monitors

- Ideas

- Block the N-1 first processes joining the barrier
- The last process wakes everyone

```
mutex_t m;
cond_t wait;
int count = 0;

void barrier ()
{
    mutex_lock (&m);
    count++;
    if (count < N)
        cond_wait (&wait, &m);
    } else {
        cond_bcast (&wait);
        count = 0;
    }
    mutex_unlock (&m);
}
```

Back to our *rendez-vous* problem

- Solution with Hoare Monitors

- Ideas

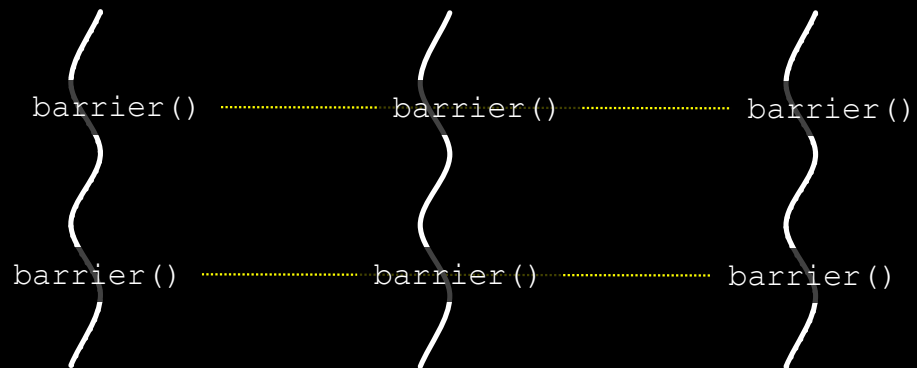
- Block the N-1 first processes joining the barrier
- The last process wakes everyone

```
mutex_t m;
cond_t wait;
int count = 0;

void barrier ()
{
    mutex_lock (&m);
    count++;
    if (count < N)
        cond_wait (&wait, &m);
    } else {
        cond_bcast (&wait);
    }
    mutex_unlock (&m);
}
```


Back to our *rendez-vous* problem

- Is it re-usable multiple times?

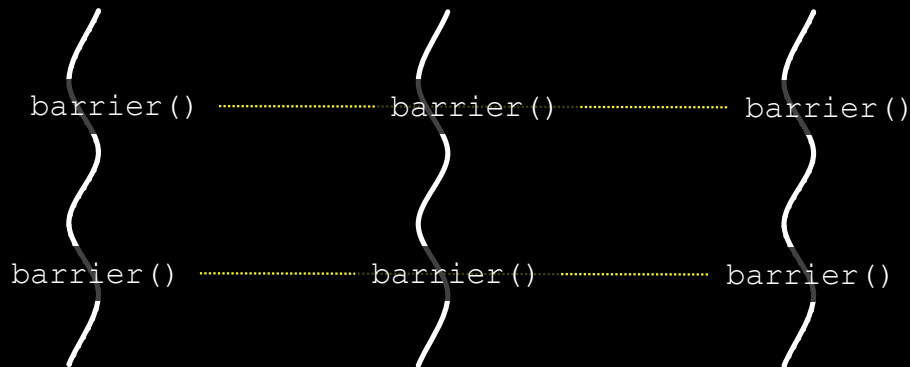


```
mutex_t m;  
cond_t wait;  
int count = 0;  
  
void barrier ()  
{  
    mutex_lock(&m);  
    count++;  
    if (count < N)  
        cond_wait (&wait, &m);  
    else {  
        cond_bcast (&wait);  
        count = 0;  
    }  
    mutex_unlock(&m);  
}
```

Back to our *rendez-vous* problem

- Is it re-usable multiple times?

- Yes! 😊



```
mutex_t m;  
cond_t wait;  
int count = 0;  
  
void barrier ()  
{  
    mutex_lock(&m);  
    count++;  
    if (count < N)  
        cond_wait (&wait, &m);  
    else {  
        cond_bcast (&wait);  
        count = 0;  
    }  
    mutex_unlock(&m);  
}
```

Producers/Consumers

```
mutex_t m;  
cond_t cons, prod;  
int nbe = 0;
```

```
Mutex_lock (&m);  
If (nbe == MAX)  
    cond_wait (&prod, &m);  
put(element) →  
Nbe++;  
Cond_signal (&cons);  
Mutex_unlock (&m);
```



Capacity = MAX elements

```
Mutex_lock (&m);  
If (nbe == 0)  
    cond_wait (&cons);  
← element = get()  
Nbe--;  
Cond_signal (&prod);  
Mutex_unlock (&m);
```

Producers/Consumers

```
mutex_t m;  
cond_t cons, prod;  
int nbe = 0;
```

Does it work well?

```
mutex_lock (&m);  
if (nbe == MAX)  
    cond_wait(&prod, &m);  
put(element) →  
nbe++;  
cond_signal(&cons);  
mutex_unlock (&m);
```



Capacity = MAX elements

```
mutex_lock (&m);  
if (nbe == 0)  
    cond_wait(&cons, &m);  
← element = get()  
nbe--;  
cond_signal(&prod);  
mutex_unlock (&m);
```

Producers/Consumers

Assuming $nbe == MAX...$

Prod₁



Prod₂



Cons



Producers/Consumers

Assuming $nbe == MAX$...

Prod₁

```
mutex_lock (&m);  
if (nbe == MAX)  
    cond_wait(&prod, m);
```



Prod₂



Cons



Producers/Consumers

Assuming $nbe == MAX$...

Prod₁

```
mutex_lock (&m);  
if (nbe == MAX)  
    cond_wait(&prod, m);
```



Prod₂



Cons

```
mutex_lock (&m);  
...
```



Producers/Consumers

Assuming $nbe == MAX...$

Prod₁

```
mutex_lock (&m);  
if (nbe == MAX)  
    cond_wait(&prod, m);
```



Prod₂

```
mutex_lock (&m);
```



Cons

```
mutex_lock (&m);  
...
```



Producers/Consumers

Prod₁

```
mutex_lock (&m);  
if (nbe == MAX)  
    cond_wait(&prod, m);
```



Prod₂

```
mutex_lock (&m);
```



Cons

```
mutex_lock (&m);  
...  
element = get();  
nbe--;  
cond_signal(&prod);  
mutex_unlock (&m);
```



Producers/Consumers

Prod₁

```
mutex_lock (&m);  
if (nbe == MAX)  
    cond_wait(&prod, m);
```



Prod₂

```
mutex_lock (&m);  
  
put(element)  
nbe++;  
cond_signal(&cons);  
mutex_unlock (&m);
```

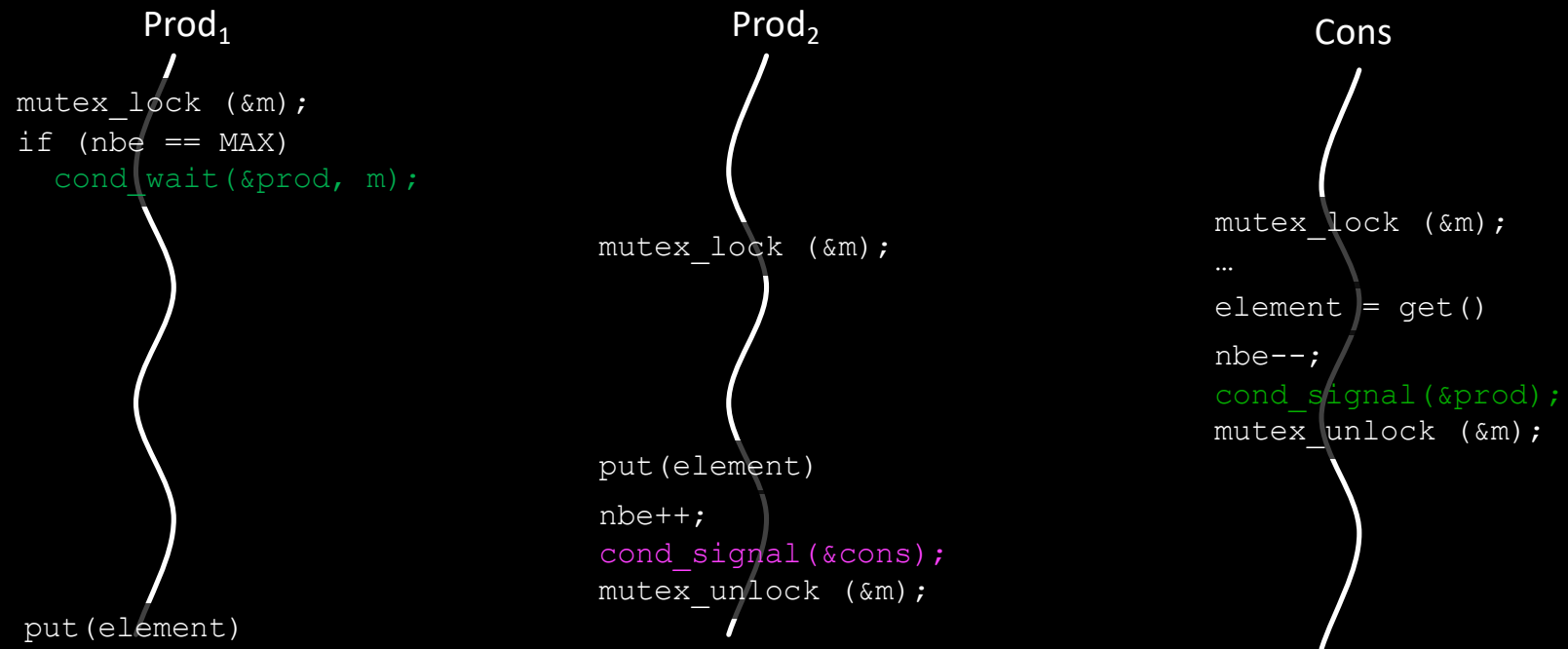


Cons

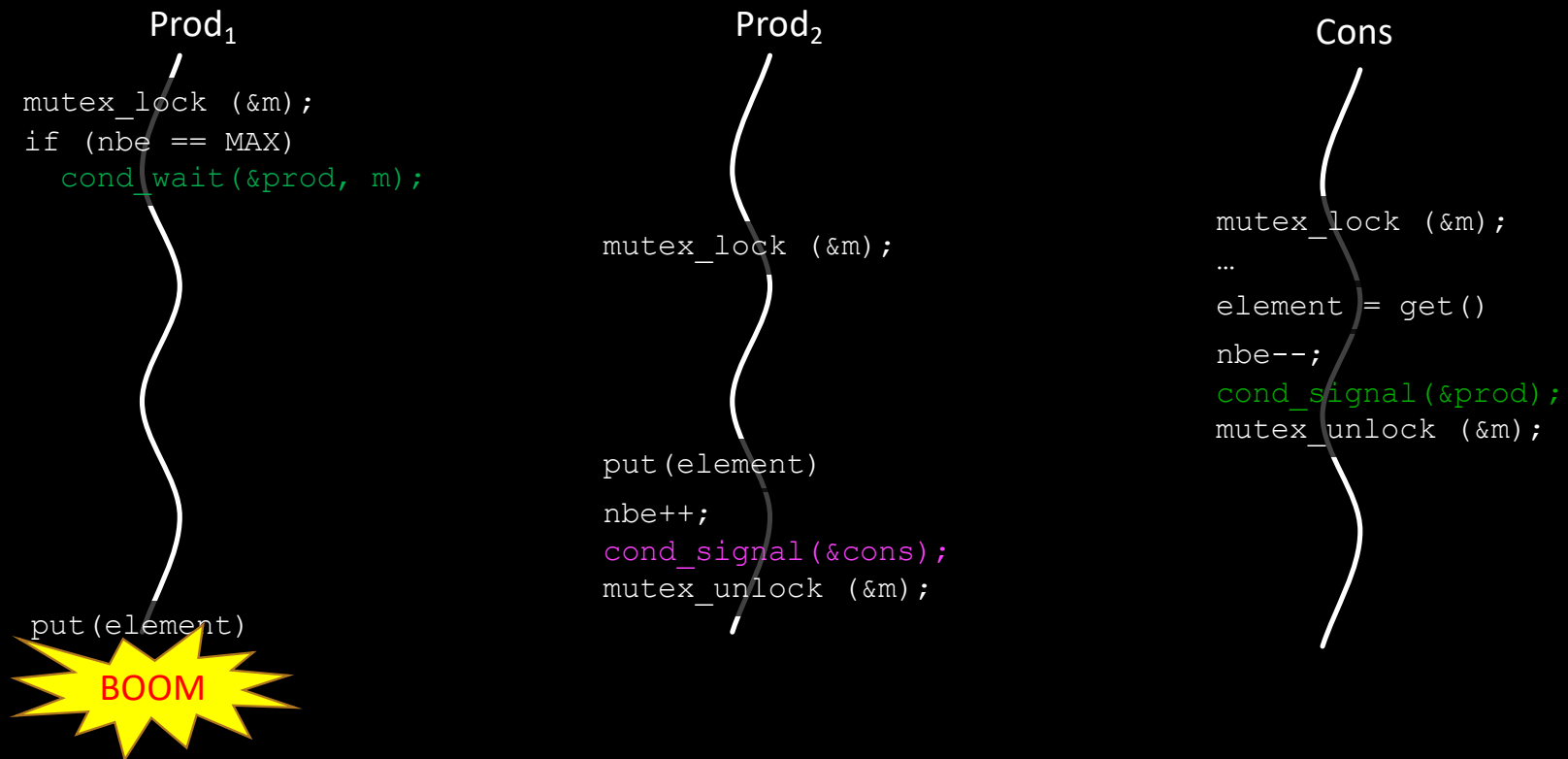
```
mutex_lock (&m);  
...  
element = get()  
nbe--;  
cond_signal(&prod);  
mutex_unlock (&m);
```



Producers/Consumers



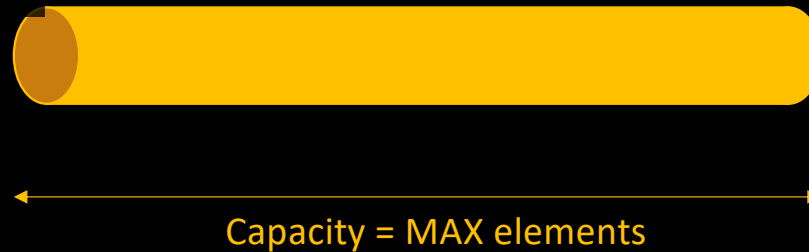
Producers/Consumers



Producers/Consumers

```
mutex_t m;  
cond_t cons, prod;  
int nbe = 0;
```

```
mutex_lock (&m);  
if (nbe == MAX)  
    cond_wait(&prod, m);  
put(element) →  
nbe++;  
cond_signal(&cons);  
mutex_unlock (&m);
```



```
mutex_lock (&m);  
if (nbe == 0)  
    cond_wait(&cons, m);  
element = get()  
nbe--;  
cond_signal(&prod);  
mutex_unlock (&m);
```

Producers/Consumers

```
mutex_t m;  
cond_t cons, prod;  
int nbe = 0;
```

```
mutex_lock (&m);  
if (nbe == MAX)  
    cond_wait(&prod, m);  
put(element) →  
nbe++;  
cond_signal(&cons);  
mutex_unlock (&m);
```



Capacity = MAX elements

```
mutex_lock (&m);  
if (nbe == 0)  
    cond_wait(&cons, m);  
element = get()  
nbe--;  
cond_signal(&prod);  
mutex_unlock (&m);
```

Producers/Consumers

```
mutex_t m;  
cond_t cons, prod;  
int nbe = 0;
```

```
mutex_lock (&m);  
while (nbe == MAX)  
    cond_wait(&prod, m);  
put(element) →  
nbe++;  
cond_signal(&cons);  
mutex_unlock (&m);
```



Capacity = MAX elements

```
mutex_lock (&m);  
while (nbe == 0)  
    cond_wait(&cons, m);  
→ element = get()  
nbe--;  
cond_signal(&prod);  
mutex_unlock (&m);
```

Readers/Writers

On souhaite disposer de verrous similaires aux « *Mutex* », mais permettant d'établir facilement une synchronisation de type « lecteurs/rédacteurs » au sein des applications. L'idée est donc de fournir un type `rwlock_t` et des primitives associées (`rwl_readlock()`, `rwl_readunlock()`, etc.) qui permettent à un processus lecteur (resp. rédacteur) d'encadrer la zone de code critique où il accèdera aux données partagées en lecture (resp. écriture).

Donnez le code associé à la gestion des verrous en lecture-écriture, en utilisant des primitives fournissant la sémantique des moniteurs de Hoare. On ne demande pas d'implémenter une version équitable du problème.

```
/* code à écrire */
typedef ... rwlock_t ;

void rwl_readlock(rwlock_t *l) ;
void rwl_readunlock(rwlock_t *l) ;
void rwl_writelock(rwlock_t *l) ;
void rwl_writeunlock(rwlock_t *l) ;
```

```
/* code disponible */
typedef ... mutex_t ;
typedef ... cond_t ;

void mutex_lock(mutex_t *m) ;
void mutex_unlock(mutex_t *m) ;
void cond_wait(cond_t *c, mutex_t *m) ;
void cond_signal(cond_t *c) ;
```


Readers/Writers

```
typedef struct {  
    unsigned nbr = 0;  
    unsigned nbw = 0;  
    mutex_t m;  
    cond_t cr, cw;  
} rwlock_t;  
  
rwlock_t cool_lock;
```

```
rwl_readlock (&cool_lock);
```

```
read();
```

```
rwl_readunlock (&cool_lock);
```

```
rwl_writelock (&cool_lock);
```

```
write();
```

```
rwl_writeunlock (&cool_lock);
```

Readers' side

```
rwl_readlock (&cool_lock);
```

```
read();
```

```
rwl_readunlock (&cool_lock);
```

```
typedef struct {  
    unsigned nbr = 0;  
    unsigned nbw = 0;  
    mutex_t m;  
    cond_t cr, cw;  
} rwlock_t;
```

```
rwlock_t cool_lock;
```

```
void rwl_readlock(rwlock_t *l)
```

```
{  
    mutex_lock (&l->m);  
    while (l->nbw > 0)  
        cond_wait (&l->cr, &l->m);  
    nbr ++;  
    mutex_unlock (&l->m);  
}
```

```
void rwl_readunlock(rwlock_t *l)
```

```
{  
    mutex_lock (&l->m);  
    nbr--;  
    if(nbr == 0)  
        cond_signal (&l->cw);  
    mutex_unlock (&l->m);  
}
```

Readers' side

```
rwl_readlock (&cool_lock);
```

```
read();
```

```
rwl_readunlock (&cool_lock);
```

```
typedef struct {  
    unsigned nbr = 0;  
    unsigned nbw = 0;  
    mutex_t m;  
    cond_t cr, cw;  
} rwlock_t;
```

```
rwlock_t cool_lock;
```

```
void rwl_readlock(rwlock_t *l)
```

```
{  
    mutex_lock (&l->m);  
    while (l->nbw > 0)  
        cond_wait (&l->cr, &l->m);  
    l->nbr++;  
    mutex_unlock (&l->m);  
}
```

```
void rwl_readunlock(rwlock_t *l)
```

```
{  
    mutex_lock (&l->m);  
    l->nbr--;  
    if (l->nbr == 0)  
        cond_signal (&l->cw);  
    mutex_unlock (&l->m);  
}
```

Writers' side

```
void rwl_writelock(rwlock_t *l)
{
    mutex_lock (&l->m);
    while (l->nbr + l->nbw > 0)
        cond_wait (&l->cw, &l->m);
    l->nbw++;
    mutex_unlock (&l->m);
}
```

```
void rwl_writeunlock(rwlock_t *l)
{
    mutex_lock (&l->m);
    l->nbw--;
    cond_signal (&l->cw);
    cond_bcast (&l->cr);
    mutex_unlock (&l->m);
}
```

```
typedef struct {
    unsigned nbr = 0;
    unsigned nbw = 0;
    mutex_t m;
    cond_t cr, cw;
} rwlock_t;
```

```
rwlock_t cool_lock;
```

```
rwl_writelock (&cool_lock);
```

```
write();
```

```
rwl_writeunlock (&cool_lock);
```

Writers' side

```
void rwl_writelock(rwlock_t *l)
{
    mutex_lock (&l->m);
    while (l->nbr + l->nbw > 0)
        cond_wait (&l->cw, &l->m);
    l->nbw++;
    mutex_unlock (&l->m);
}
```

```
void rwl_writeunlock(rwlock_t *l)
{
    mutex_lock (&l->m);
    l->nbw--;
    cond_signal (&l->cw); cond_bcast (&l->cr);
    mutex_unlock (&l->m);
}
```

```
typedef struct {
    unsigned nbr = 0;
    unsigned nbw = 0;
    mutex_t m;
    cond_t cr, cw;
} rwlock_t;
```

```
rwlock_t cool_lock;
```

```
rwl_writelock (&cool_lock);
```

```
write();
```

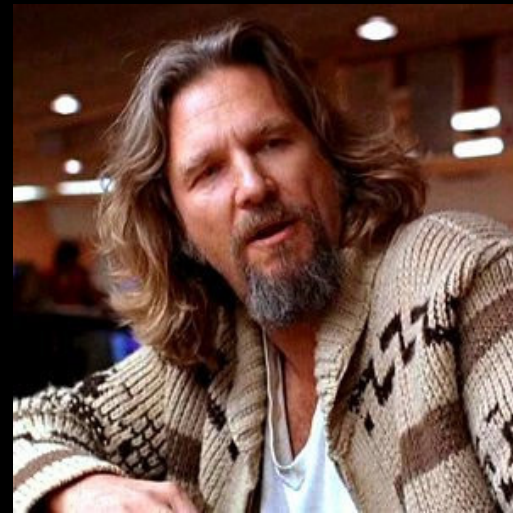
```
rwl_writeunlock (&cool_lock);
```

Final words

- Like Semaphores, Hoare monitors are implemented on top of atomic processor instructions
- User space implementations often mix polling and blocking calls
 - i.e. poll during a few iterations and then block if lock not available
- Hoare monitors are preferred by most programmers
 - For the sake of minimizing headaches...

Final words

Despite his promising initials,
Paul-Victor Semaphore
is not
the inventor of semaphores...



Final words

- FUTEXes (Fast User-Level Mutex) were introduced in Linux 2.6.x in 2003
 - Introduced in Microsoft Windows 8 under the name “WaitOnAddress”
 - Patented in 2013 😊
- Idea
 - Use atomic operations on 32bits integer variables in user space
 - If blocking or waking is needed, use
 - `sys_futex (&var, FUTEX_WAIT...)` or `sys_futex (&var, FUTEX_WAKE...)`
 - Kernel queues are associated to physical addresses using hash tables

Final words

- FUTEXes (Fast User-Level Mutex) were introduced in Linux 2.6.x in 2003
 - Introduced in Microsoft Windows 8 under the name “WaitOnAddress”
 - Patented in 2013 😊
- Idea
 - Use atomic operations on 32bits integer variables in user space
 - If blocking / waking is needed, use
 - `sys_futex (&var, FUTEX_WAIT...)` / `sys_futex (&var, FUTEX_WAKE...)`
 - Kernel queues are associated to physical addresses using hash tables
- Hmm... Isn't this what we were looking for on slide 53? 🙄

Additional resources
available on
<http://gforgeron.gitlab.io/se/>