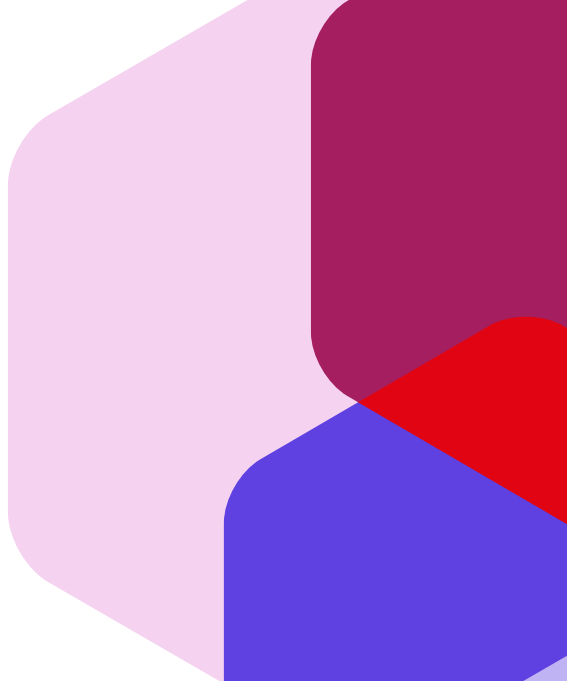


Principles, Algorithms and Models of Concurrency and Synchronization

03 Memory Primitives & Consensus Hierarchy



- Only reading and writing are atomic (indivisible) operations
- Atomic read operation: `if  $x = 10$  then ...`
Actually means `if  $read(x) = 10$  then ...`
- Atomic write operation: `$x := 10$`
Actually means `$write(x, 10)$`

Now

- Hardware supports various stronger memory operations
- Formally, we represent these operations as functions

```
function  $xyz(p_1 : \mathbf{type}_1, \dots, p_n : \mathbf{type}_n \dots)$  returns : type
```

“atomic execution of the body”

```
end function
```

Now

- Hardware supports various stronger memory operations
- Formally, we represent these operations as functions

Function name

Parameters with types

Return type

`function xyz( $p_1 : \mathbf{type}_1, \dots, p_n : \mathbf{type}_n \dots$ ) returns : type`

“atomic execution of the body”

`end function`

- **Register** = shared memory register
- **Local register** = local memory register
- **Value** = a value such as True, 1, (no further distinction)
- **Function** = a side-effect-free function on values

Read and Write in Function Notation

read(*r*)

```
function read(r : register) returns : value  
    return(r);  
end function
```

write(*r*, *v*)

```
function write(r : Register, v : Value)  
    r := v  
end function
```

Read and Write in Function Notation

read(*r*)

```
function read(r : register) returns : value  
  return(r);  
end function
```

write(*r*, *v*)

```
function write(r : Register, v : Value)  
  r := v  
end function
```



no return, as there is no return value!

“Concurrent Object”:

Concurrent data structure with atomic operations

“Concurrent Object”:

Concurrent data structure with atomic operations

Example:

Atomic memory register (as object)

Shared memory register with

- atomic *read* operation
- atomic *write* operation

test-and-set(r, v)

Value v is assigned to register r , return value: **old** value of r .

```
function test-and-set( $r$  : register,  $v$  : value) returns : value  
     $temp := r$ ;  
     $r := v$ ;  
    return( $temp$ );  
end function
```

test-and-set(r, v)

Value v is assigned to register r , return value: **old** value of r .

```
function test-and-set( $r$  : register,  $v$  : value) returns : value  
   $temp := r$ ;  
   $r := v$ ;  
  return( $temp$ );  
end function
```

Variant: register, must be 0 or 1.

```
function test-and-set( $r$  : register) returns : value  
   $temp := r$ ;  
   $r := 1$ ;  
  return( $temp$ );  
end function
```

Test-and-Set Object

Test-and-set object

A shared memory register with

- *write* operation
- *test-and-set* operations

Test-and-set bit

A shared memory register with values 0 and 1

- *test-and-set* operations set the register to 1
- *reset* operation: atomic write of a 0 to the register,

Test-and-test-and-set object

A test-and-set object that additionally supports the atomic *read* operation

swap

Swap value of local register with shared register:

```
function swap(r : register, l : local register)  
    temp := r;  
    r := l;  
    l := temp;  
end function
```

is sometimes also referred to as *fetch-and-store*.

Swap object

A shared memory register with

- *swap* operation between the register and any local register.

fetch-and-add

Increase the register value by v , return: old value

```
function fetch-and-add( $r$  : register,  $v$  : value) returns : value  
   $temp := r$ ;  
   $r := temp + v$ ;  
  return( $temp$ );  
end function
```

Variant: *fetch-and-increment*: exactly 1 is added

Fetch-and-increment object

A shared memory register with

- *fetch-and-increment* operation
- *write* operation
- *read* operation

Fetch-and-add object

A shared memory register with

- *fetch-and-add* operation
- *write* operation
- *read* operation

read-modify-write

Applying a function to the register value. Return: old value.

```
function read-modify-write(r : register, f : function)  
  returns : value  
    temp := r;  
    r := f(temp);  
    return(temp);  
end function
```

read-modify-write

Applying a function to the register value. Return: old value.

```
function read-modify-write(r : register, f : function)  
  returns : value  
    temp := r;  
    r := f(temp);  
    return(temp);  
end function
```

Note:

- *read*, *write*, *test-and-set*, and *fetch-and-add* can all be expressed using *read-modify-write*
- Function *f* must be free of side effects.

E.g.: *f* is an atomic operation such as *fetch-and-increment*

⇒ unclear what is executed atomically? Therefore: Prohibited!

Read-Modify-Write object

A shared memory register with

- *read-modify-write* operation
- *write* operation
- *read* operation

compare-and-swap

Operation expects (r,old,new):

If r=old, then new value for r and return True, otherwise return False

```
function compare-and-swap(r : register, old : value, new : value)  
  returns : value  
    if r = old then  
      r := new;  
      return(True);  
    else  
      return(False);  
  end function
```

Compare-and-swap object

A shared memory register with

- *compare-and-swap* operation
- *write* operation
- *read* operation

sticky-write

- *sticky-write*(r, v) expects a register and a value.
- Initial value of r : undefined (written as $\perp$).
- If the value of the register r is equal to v or $\perp$, then the value is set to v , True is returned as the result.
- otherwise: False is returned as the result

Variant

sticky-bit-write operation

Special case: r can only take the values 0, 1, and $\perp$

Sticky bit

A shared memory register with

- *read* operation
- *sticky-bit-write* operation

move

Value of a shared register is "moved" to another shared register

```
function move( $r_1$  : register,  $r_2$  : register)  
    temp :=  $r_2$ ;  
     $r_1$  := temp;  
end function
```

Move object

Group of shared memory registers such that

- The *move* operation is supported in pairs
- *write* operation for all registers
- *read* operation for all registers

shared swap

Two shared registers swap their values

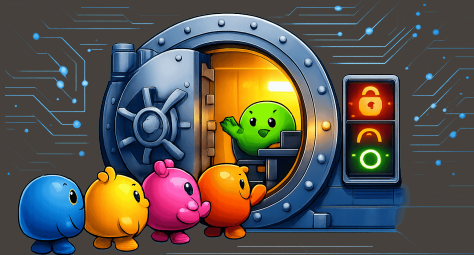
```
function shared-swap( $r_1$  : register,  $r_2$  : register)  
     $temp_1 := r_1$ ;  
     $temp_2 := r_2$ ;  
     $r_1 := temp_2$ ;  
     $r_2 := temp_1$ ;  
end function
```

Shared swap object

Group of shared memory registers, with

- in pairs: the *shared-swap* operation
- *write* operation
- *read* operation

Mutual Exclusion Algorithms with Test-and-Set-Bits



Test-and-set Bit = Lock

A shared memory register with values 0 and 1

- *test-and-set* operations set the register to 1
Operation is also called *lock*
- *reset* operation: atomic write of a 0
Operation is also called *unlock*

Spin-Lock

Algorithms often use locks as follows:

```
while test-and-set(lock)=1 do skip;
```

In other words, the algorithm “*spins*” around the lock.

Hence the name *spin lock*

Simple Algorithm with Test-and-Set Bit

Initial: Test-and-set bit x has value 0

Program of the i -th process

```
loop forever
(1)  remaining code
(2)  await (test-and-set( $x$ )=0);
(3)  critical section
(4)  reset( $x$ );
end loop
```

Example: 4 processes

Process 1

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

Process 2

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

$x :=$

Process 3

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

Process 4

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

Example: 4 processes

Process 1

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

Process 2

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

x := 1
return from t-a-s: 0

Process 3

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

Process 4

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

Example: 4 processes

Process 1

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

Process 2

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

$x := 1$

Process 3

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

Process 4

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

Example: 4 processes

Process 1

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

Process 2

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

x := 1
return from t-a-s: 1

Process 3

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

Process 4

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

Example: 4 processes

Process 1

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

Process 2

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

x := 1
return from t-a-s: 1

Process 3

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

Process 4

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

Example: 4 processes

Process 1

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

Process 2

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

x := 0

Process 3

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

Process 4

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

Example: 4 processes

Process 1

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

Process 2

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

x := 1
return from t-a-s: 0

Process 3

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

Process 4

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

Example: 4 processes

Process 1

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

Process 2

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

$x := 1$

Process 3

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

Process 4

```
loop forever
(1)  remaining code
(2)  await (test-and-set(x)=0);
(3)  critical section
(4)  reset(x);
end loop
```

- Guarantees **mutual exclusion**:
 - Only one process can atomically set x from 0 to 1 and obtain 0 as the result.
 - As long as $x=1$, all other processes obtain the result 1.
 - Only after atomic *reset* can the next process set the lock
- is **deadlock-free**
 - If several processes are in the entry code and none are in the critical section, then $x=0$ must hold after a finite number of steps
 - After that: One must set the lock after a finite number of steps
- is **not** starvation-free

Properties of the Simple Algorithm (2)

- Bad from a **practical** point of view:
With many processes: Lock is set constantly (from 1 to 1)
- Due to constant writing, processor caches must also be constantly updated!
- This leads to (unnecessary) communication overhead!

Improvement with Test-and-Test-and-Set-Bit

Initial: Test-and-test-and-set bit x has value 0

Program of the i -th process

```
loop forever
(1)  remaining code
(2)  await ( $x=0$ );
(3)  while (test-and-set( $x$ )=1) do
(4)    await ( $x=0$ );
(5)  Critical section
(6)  reset( $x$ );
end loop
```

- Waiting is done by **reading**
- Only if there is a chance to set the lock: test-and-set

(Practical) Improvement with Pause Operation

Initial: Test-and-test-and-set bit x has value 0, delay: local variable, minDelay, maxDelay: constants

Program of the i -th process

```
loop forever
(1)  remaining code
(2)  delay=minDelay;
(3)  repeat
(4)    delay := min(2*delay,maxDelay)
(5)    while (x=1) do pause(delay);
(6)  until (test-and-set(x)=0)
(7)  Critical section
(8)  reset(x);
end loop
```

- pause(x) waits x ms
- Waiting period is increased exponentially (“exponential backoff”)
- Only increase if lock was free but not acquired

Starvation-Free Algorithm with Test-and-Set

- Published: R. Alur and G. Taubenfeld 1993
- Starvation-free
- Fulfills the property n -fairness (n = number of processes)

r -fairness

A waiting process has the opportunity to enter the critical section before all other processes can enter the critical section $r + 1$ times together.

Note:

- r -fairness implies r -bounded waiting
- r -bounded waiting implies $((n - 1) \cdot r)$ -fairness

Starvation-Free Algorithm with Test-and-Set (2)

Program of the i-th process

```
loop forever
(1)  remaining code
(2)  waiting[i]:=True;
(3)  key := 1;
(4)  while (waiting[i] and key=1) do
(5)    key := test-and-set(lock);
(6)  waiting[i]:=False;
(7)  Critical section
(8)  if turn=i
(9)    then lturn := 1 + (turn mod n);
(10)   else lturn := turn
(11)   if waiting[lturn]
(12)     then turn := lturn;
(13)       waiting[lturn] := False;
(14)   else turn := 1+(lturn mod n);
(15)       reset(lock);
end loop
```

Initial:

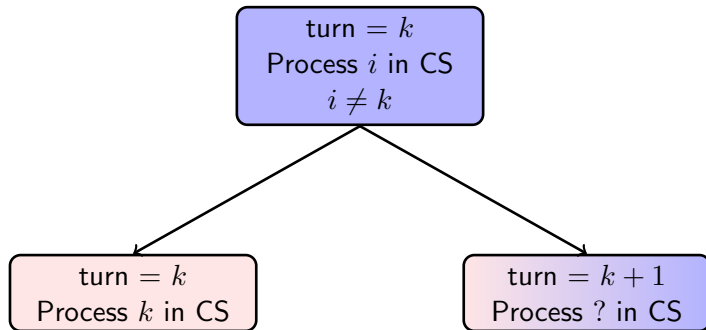
turn: atom. Register, value irrelevant (between
1,...,n);
Test-and-set bit lock: 0;
waiting[i]: False;
lturn, key: local registers

Starvation-Free Algorithm with Test-and-Set (2)

- Variable *turn* holds number of a process
- Trick: in the exit code
- Lock is “passed” to process with number *turn*,
if it is waiting
- this happens without *reset*, but via the *waiting* array
- (If *turn* is current process, then it is incremented beforehand)
- If process *turn* is not waiting, then increase *turn* and *reset*
- After at most n times exit code in total
the following applies: $\text{turn} = j$ (next slide)

Proof of n -Fairness

Two cases: Case 1:



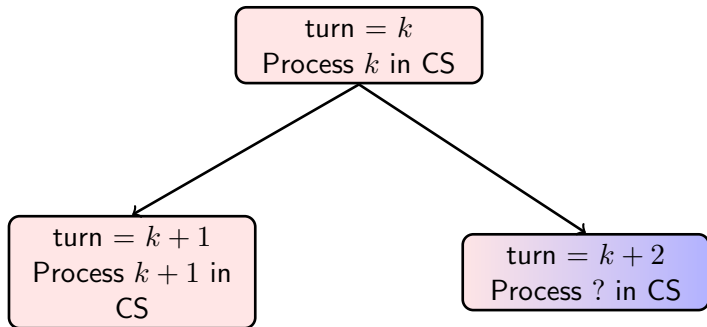
Red nodes = process u in the critical section and $\text{turn} = u$

Blue nodes = process u in the critical section and $\text{turn} \neq u$

Red/blue = both possible

Proof of n -Fairness (2)

Two cases: Case 2:



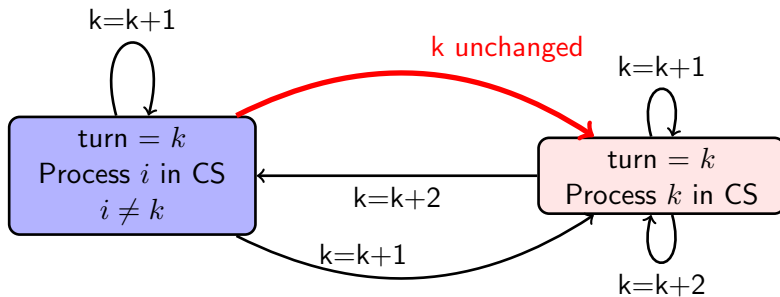
Red nodes = process u in the critical section and $\text{turn} = u$

Blue nodes = process u in the critical section and $\text{turn} \neq u$

Red/blue = both possible

Proof of n -Fairness (3)

Union of cases:



Red arrow can only be selected again once turn has been increased by 2!

Mutual Exclusion Algorithm With RMW-Object



Ticket Algorithm with RMW-Object

- M.J. Fischer, N.A. Lynch, J.E. Burns, A. Borodin 1989
- Idea similar to the bakery algorithm ("drawing numbers").
- However, a fixed number of numbers is sufficient.

Ticket Algorithm with RMW-Object (2)

Initial: (ticket,valid): Read-Modify-Write object for a pair of numbers in the range $1 \dots n$ where initially ticket = valid (ticket_i, valid_i) local register

Program of the i-th process

```
loop forever
(1)  remaining code
    // increase ticket (read old values):
(2)  (ticketi,validi) := read-modify-write((ticket,valid), inc-fst)
(3)  while ticketi ≠ validi do
(4)    validi := valid;
(5)  Critical section
(6)  read-modify-write((ticket,valid), inc-snd) // increase valid
end loop
```

Functions

```
function inc-fst((a,b))
  return (1+(a mod n), b)
end function
```

```
function inc-snd((a,b))
  return (a, 1+(b mod n))
end function
```

Example

Process 1

- (1) remaining code
- (2) $(t_1, v_1) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_1 \neq v_1$ do
- (4) $v_1 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_1 = ?$

$v_1 = ?$

Process 2

- (1) remaining code
- (2) $(t_2, v_2) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_2 \neq v_2$ do
- (4) $v_2 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_2 = ?$

$v_2 = ?$

Process 3

- (1) remaining code
- (2) $(t_3, v_3) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_3 \neq v_3$ do
- (4) $v_3 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_3 = ?$

$v_3 = ?$

$t = 1$

$v = 1$

Example

Process 1

- (1) remaining code
- (2) $(t_1, v_1) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_1 \neq v_1$ do
- (4) $v_1 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_1 = ?$

$v_1 = ?$

Process 2

- (1) remaining code
- (2) $(t_2, v_2) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_2 \neq v_2$ do
- (4) $v_2 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_2 = ?$

$v_2 = ?$

Process 3

- (1) remaining code
- (2) $(t_3, v_3) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_3 \neq v_3$ do
- (4) $v_3 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_3 = 1$

$v_3 = 1$

$t = 2$

$v = 1$

Example

Process 1

- (1) remaining code
 - (2) $(t_1, v_1) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_1 \neq v_1$ do
 - (4) $v_1 := v;$
 - (5) Critical section
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $t_1 = 2$
 $v_1 = 1$

Process 3

- (1) remaining code
 - (2) $(t_3, v_3) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_3 \neq v_3$ do
 - (4) $v_3 := v;$
 - (5) Critical section
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $t_3 = 1$
 $v_3 = 1$

Process 2

- (1) remaining code
 - (2) $(t_2, v_2) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_2 \neq v_2$ do
 - (4) $v_2 := v;$
 - (5) Critical section
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $t_2 = ?$
 $v_2 = ?$

$t = 3$
 $v = 1$

Example

Process 1

- (1) remaining code
 - (2) $(t_1, v_1) := r-m-w((t, v), inc-fst)$
 - (3) while $t_1 \neq v_1$ do
 - (4) $v_1 := v$;
 - (5) Critical section
 - (6) $r-m-w((t, v), inc-snd)$
- $t_1 = 2$
 $v_1 = 1$

Process 3

- (1) remaining code
 - (2) $(t_3, v_3) := r-m-w((t, v), inc-fst)$
 - (3) while $t_3 \neq v_3$ do
 - (4) $v_3 := v$;
 - (5) Critical section
 - (6) $r-m-w((t, v), inc-snd)$
- $t_3 = 1$
 $v_3 = 1$

Process 2

- (1) remaining code
 - (2) $(t_2, v_2) := r-m-w((t, v), inc-fst)$
 - (3) while $t_2 \neq v_2$ do
 - (4) $v_2 := v$;
 - (5) Critical section
 - (6) $r-m-w((t, v), inc-snd)$
- $t_2 = 3$
 $v_2 = 1$

$t = 1$
 $v = 1$

Example

Process 1

- (1) remaining code
- (2) $(t_1, v_1) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_1 \neq v_1$ do
- (4) $v_1 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_1 = 2$
 $v_1 = 1$

Process 2

- (1) remaining code
- (2) $(t_2, v_2) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_2 \neq v_2$ do
- (4) $v_2 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_2 = 3$
 $v_2 = 1$

Process 3

- (1) remaining code
- (2) $(t_3, v_3) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_3 \neq v_3$ do
- (4) $v_3 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_3 = 1$
 $v_3 = 1$

$t = 1$
 $v = 1$

Example

Process 1

- (1) remaining code
- (2) $(t_1, v_1) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_1 \neq v_1$ do
- (4) $v_1 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_1 = 2$
 $v_1 = 1$

Process 2

- (1) remaining code
- (2) $(t_2, v_2) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_2 \neq v_2$ do
- (4) $v_2 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_2 = 3$
 $v_2 = 1$

Process 3

- (1) remaining code
- (2) $(t_3, v_3) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_3 \neq v_3$ do
- (4) $v_3 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_3 = 1$
 $v_3 = 1$

$t = 1$
 $v = 1$

Example

Process 1

- (1) remaining code
- (2) $(t_1, v_1) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_1 \neq v_1$ do
- (4) $v_1 := v$;
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_1 = 2$
 $v_1 = 1$

Process 2

- (1) remaining code
- (2) $(t_2, v_2) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_2 \neq v_2$ do
- (4) $v_2 := v$;
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_2 = 3$
 $v_2 = 1$

Process 3

- (1) remaining code
- (2) $(t_3, v_3) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_3 \neq v_3$ do
- (4) $v_3 := v$;
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_3 = 1$
 $v_3 = 1$

$t = 1$
 $v = 1$

Example

Process 1

- (1) remaining code
- (2) $(t_1, v_1) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_1 \neq v_1$ do
- (4) $v_1 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_1 = 2$
 $v_1 = 1$

Process 2

- (1) remaining code
- (2) $(t_2, v_2) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_2 \neq v_2$ do
- (4) $v_2 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_2 = 3$
 $v_2 = 1$

Process 3

- (1) remaining code
- (2) $(t_3, v_3) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_3 \neq v_3$ do
- (4) $v_3 := v;$
- (5) **Critical section**
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_3 = 1$
 $v_3 = 1$

$t = 1$
 $v = 1$

Example

Process 1

- (1) remaining code
- (2) $(t_1, v_1) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_1 \neq v_1$ do
- (4) $v_1 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_1 = 2$
 $v_1 = 1$

Process 2

- (1) remaining code
- (2) $(t_2, v_2) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_2 \neq v_2$ do
- (4) $v_2 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_2 = 3$
 $v_2 = 1$

Process 3

- (1) remaining code
- (2) $(t_3, v_3) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_3 \neq v_3$ do
- (4) $v_3 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_3 = 1$
 $v_3 = 1$

$t = 1$
 $v = 2$

Example

Process 1

- (1) remaining code
- (2) $(t_1, v_1) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) **while** $t_1 \neq v_1$ **do**
- (4) $v_1 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_1 = 2$
 $v_1 = 1$

Process 2

- (1) remaining code
- (2) $(t_2, v_2) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) **while** $t_2 \neq v_2$ **do**
- (4) $v_2 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_2 = 3$
 $v_2 = 1$

Process 3

- (1) remaining code
- (2) $(t_3, v_3) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) **while** $t_3 \neq v_3$ **do**
- (4) $v_3 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_3 = 1$
 $v_3 = 1$

$t = 1$
 $v = 2$

Example

Process 1

- (1) remaining code
- (2) $(t_1, v_1) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_1 \neq v_1$ do
- (4) $v_1 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_1 = 2$

$v_1 = 2$

Process 3

- (1) remaining code
- (2) $(t_3, v_3) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_3 \neq v_3$ do
- (4) $v_3 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_3 = 1$

$v_3 = 1$

Process 2

- (1) remaining code
- (2) $(t_2, v_2) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_2 \neq v_2$ do
- (4) $v_2 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_2 = 3$

$v_2 = 1$

$t = 1$

$v = 2$

Example

Process 1

- (1) remaining code
 - (2) $(t_1, v_1) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) **while** $t_1 \neq v_1$ **do**
 - (4) $v_1 := v;$
 - (5) Critical section
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $t_1 = 2$
 $v_1 = 2$

Process 3

- (1) remaining code
 - (2) $(t_3, v_3) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) **while** $t_3 \neq v_3$ **do**
 - (4) $v_3 := v;$
 - (5) Critical section
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $t_3 = 1$
 $v_3 = 1$

Process 2

- (1) remaining code
 - (2) $(t_2, v_2) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) **while** $t_2 \neq v_2$ **do**
 - (4) $v_2 := v;$
 - (5) Critical section
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $t_2 = 3$
 $v_2 = 1$

$t = 1$
 $v = 2$

Example

Process 1

- (1) remaining code
- (2) $(t_1, v_1) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_1 \neq v_1$ do
- (4) $v_1 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_1 = 2$
 $v_1 = 2$

Process 2

- (1) remaining code
- (2) $(t_2, v_2) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_2 \neq v_2$ do
- (4) $v_2 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_2 = 3$
 $v_2 = 1$

Process 3

- (1) remaining code
- (2) $(t_3, v_3) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_3 \neq v_3$ do
- (4) $v_3 := v;$
- (5) Critical section
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$

$t_3 = 1$
 $v_3 = 1$

$t = 1$
 $v = 2$

Theorem

The ticket algorithm guarantees mutual exclusion, starvation freedom, and the FIFO property.

The proof is simple (we omit it).

Properties of the Ticket Algorithm (2)

Remarks:

- The ticket algorithm uses an RMW object with n^2 different values.
- With regard to this number of values, the algorithm is the **best** known algorithm among all algorithms that guarantee mutual exclusion, deadlock freedom, and the strong FIFO property.

Strong FIFO property:

FIFO also applies to initialization and exit code

- Lower bound:

Every algorithm for $n \geq 3$ processes requires at least $\frac{n^2 - 3n + 2}{2}$ different states of the shared memory.

Mutual Exclusion Algorithm with Queue and Compare-and-Swap Object



The MCS Queue-Based Algorithm

- Published by Mellor-Crummey and Scott in 1991
- Dijkstra Prize in Distributed Computing 2006
- Uses a **swap-compare-and-swap object**, i.e., an object with *swap* and *compare-and-swap* operations
- Processes line up in a queue
- Special feature: The exit code is **not wait-free**
- Properties: fulfills mutual exclusion, starvation-free, and **FIFO** property
- Only waits on **local** registers

For a pointer p , let

- $*p$ be the indirection operation, i.e., $*p$ returns the object pointed to by the pointer.
- $\&p$ returns the memory address of an object (in order to create pointers to the object)

The MCS Queue-Based Algorithm (2)

Types:

- Record Element: two attributes: value:Bool, and next: pointer to an element

Shared variables:

- nodes[i]: Array entry: Content of a record of type element
- tail: Swap and compare-and-swap object of type: Pointer to an element
Initial value: Nil

Local variables:

- mynode: pointer to an element, initially pointing to nodes[i]
- prev, succ: pointers to elements

The MCS Queue-Based Algorithm (3)

Program of the i-th process

```
loop forever
(1)   remaining code
(2)   *mynode.next := Nil;
(3)   prev := mynode;
(4)   swap(tail,prev);
(5)   if prev  $\neq$  Nil then
(6)     *mynode.value := 1;
(7)     *prev.next := mynode;
(8)     await *mynode.value = 0;
(9)   Critical section
(10)  if mynode.next = Nil then
(11)    if compare-and-swap(tail,mynode,Nil) = False then
(12)      await *mynode.next  $\neq$  Nil;
(13)      succ := *mynode.next;
(14)      *succ.value := 0;
(15)  else
(16)    succ := *mynode.next;
(17)    *succ.value := 0;
```

The MCS Queue-Based Algorithm (4)

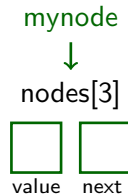
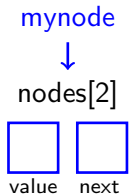
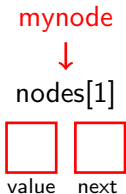
Idea:

- Processes join the queue.
- First process is allowed to enter the critical section.
- When leaving the critical section, there are several cases:
 - If another process arrived in the meantime and attached itself, then the case is simple: its value is set to 0 ($*mynode.next \neq Nil$)
 - If no process entered the initialization code in the meantime, then tail is set to Nil (case compare-and-swap returns True)
 - If a process has arrived in the meantime but has not yet finished attaching, then wait until it has attached (compare-and-swap case returns false)

The MCS Queue-Based Algorithm: Example

Only one process wants to enter the critical section

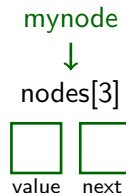
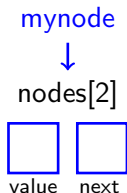
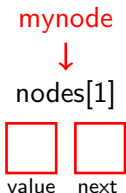
tail $\longrightarrow$ Nil



The MCS Queue-Based Algorithm: Example

Only one process wants to enter the critical section

tail $\longrightarrow$ Nil



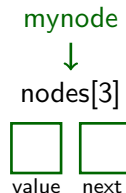
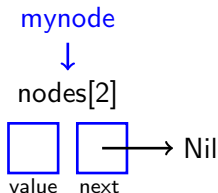
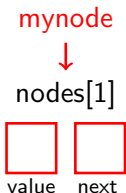
Process 2:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

The MCS Queue-Based Algorithm: Example

Only one process wants to enter the critical section

tail $\longrightarrow$ Nil



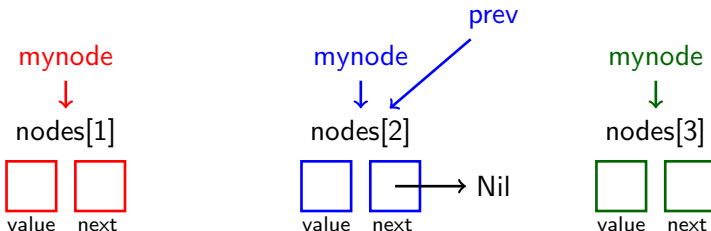
Process 2:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

The MCS Queue-Based Algorithm: Example

Only one process wants to enter the critical section

tail $\longrightarrow$ Nil

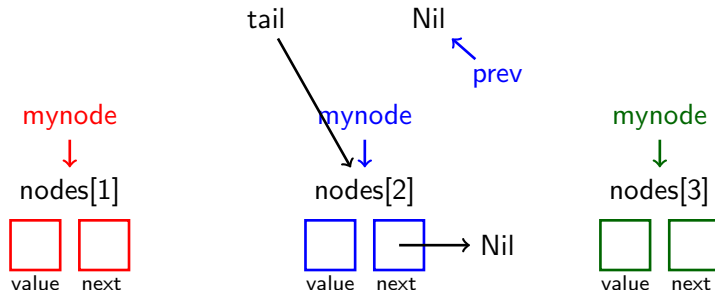


Process 2:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

The MCS Queue-Based Algorithm: Example

Only one process wants to enter the critical section

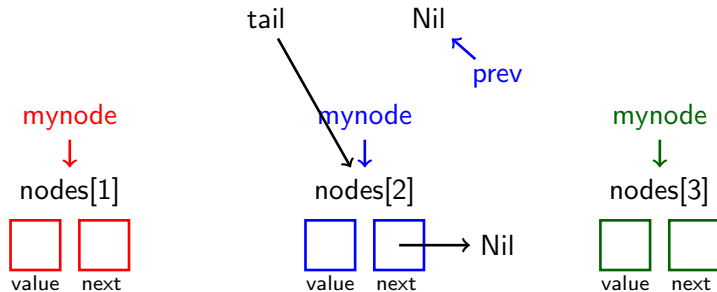


Process 2:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

The MCS Queue-Based Algorithm: Example

Only one process wants to enter the critical section



Process 2:

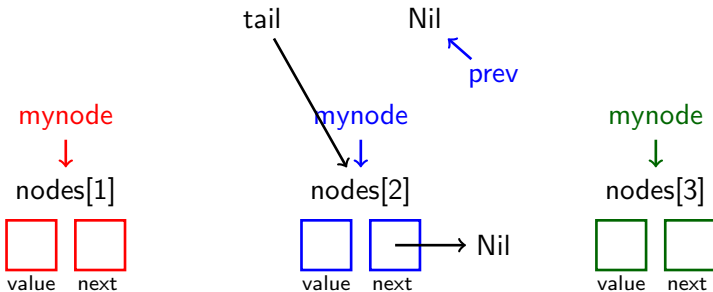
(5) if `prev` $\neq$ `Nil` then

...

(9) Critical section

The MCS Queue-Based Algorithm: Example

Only one process wants to enter the critical section



Process 2:

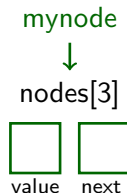
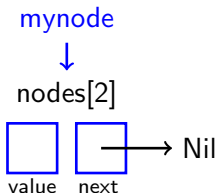
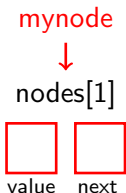
(10) if `mynode.next = Nil` then

(11) if *compare-and-swap*(`tail`, `mynode`, `Nil`) = `False` then

The MCS Queue-Based Algorithm: Example

Only one process wants to enter the critical section

tail $\longrightarrow$ Nil
 $\nwarrow$ prev



Process 2:

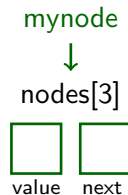
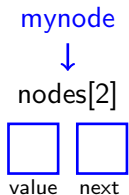
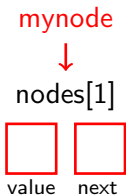
(10) if mynode.next = Nil then

(11) if *compare-and-swap*(tail, mynode, Nil) = False then
 ... returns True

The MCS Queue-Based Algorithm: Example (2)

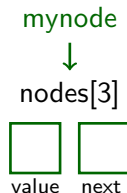
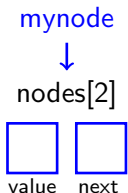
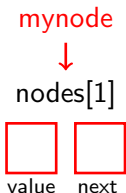
Process 2 in the CS, process 3 wants to enter the critical section

tail $\longrightarrow$ Nil



The MCS Queue-Based Algorithm: Example (2)

Process 2 in the CS, process 3 wants to enter the critical section
tail $\longrightarrow$ Nil

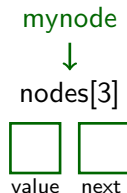
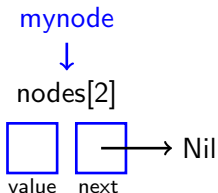
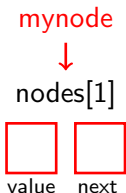


Process 2:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

The MCS Queue-Based Algorithm: Example (2)

Process 2 in the CS, process 3 wants to enter the critical section
tail $\longrightarrow$ Nil



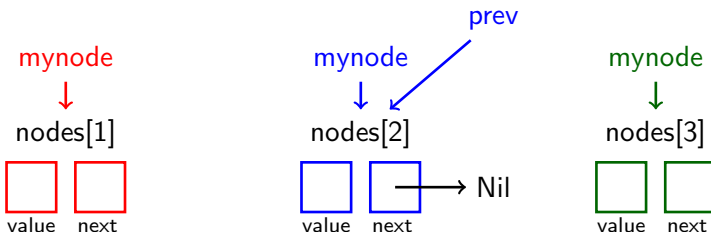
Process 2:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

The MCS Queue-Based Algorithm: Example (2)

Process 2 in the CS, process 3 wants to enter the critical section

tail $\longrightarrow$ Nil

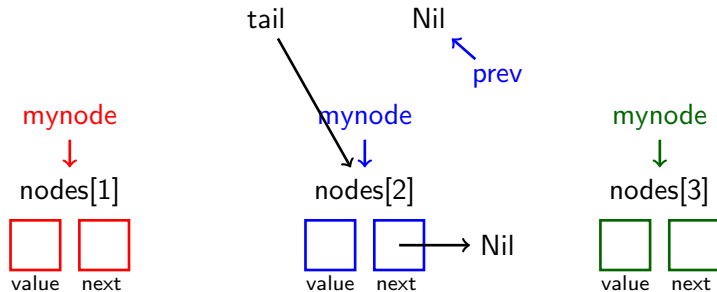


Process 2:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail, prev);`

The MCS Queue-Based Algorithm: Example (2)

Process 2 in the CS, process 3 wants to enter the critical section

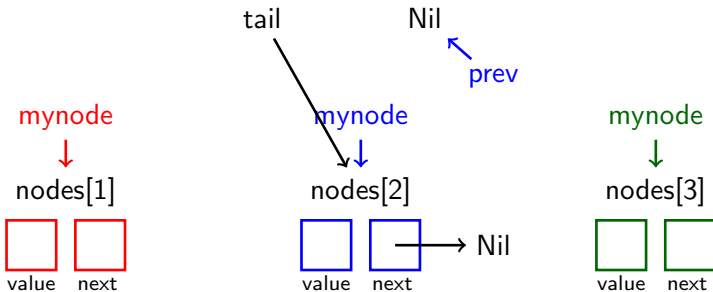


Process 2:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

The MCS Queue-Based Algorithm: Example (2)

Process 2 in the CS, process 3 wants to enter the critical section



Process 2:

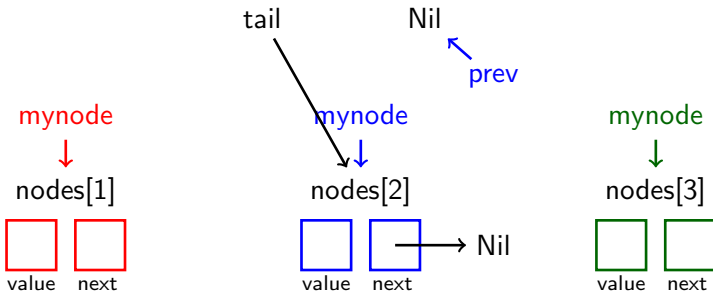
(5) if `prev` $\neq$ `Nil` then

...

(9) Critical section

The MCS Queue-Based Algorithm: Example (2)

Process 2 in the CS, process 3 wants to enter the critical section

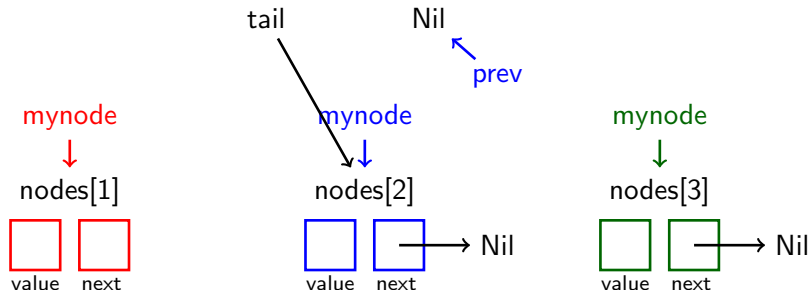


Process 3:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail, prev);`

The MCS Queue-Based Algorithm: Example (2)

Process 2 in the CS, process 3 wants to enter the critical section

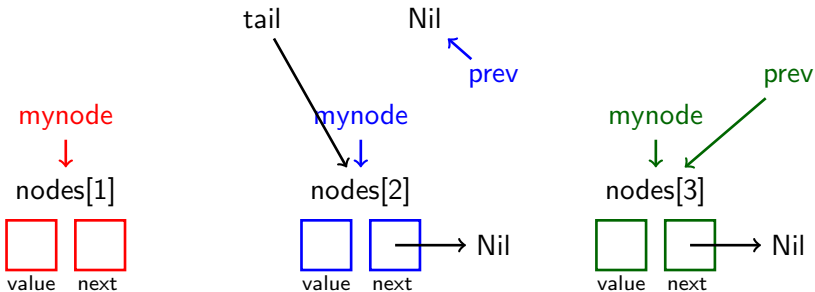


Process 3:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

The MCS Queue-Based Algorithm: Example (2)

Process 2 in the CS, process 3 wants to enter the critical section

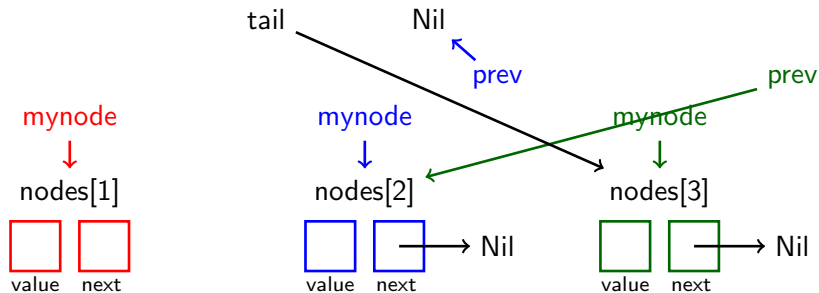


Process 3:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail, prev);`

The MCS Queue-Based Algorithm: Example (2)

Process 2 in the CS, process 3 wants to enter the critical section

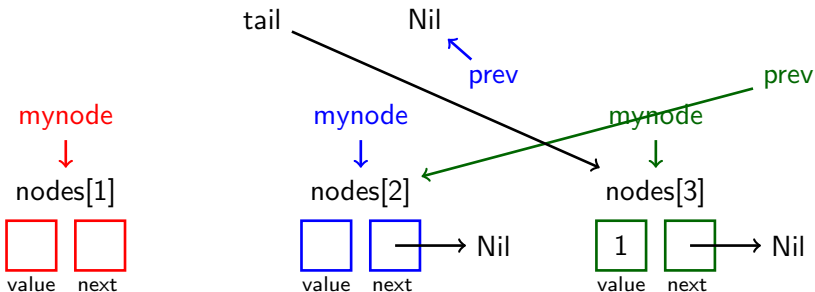


Process 3:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail, prev);`

The MCS Queue-Based Algorithm: Example (2)

Process 2 in the CS, process 3 wants to enter the critical section

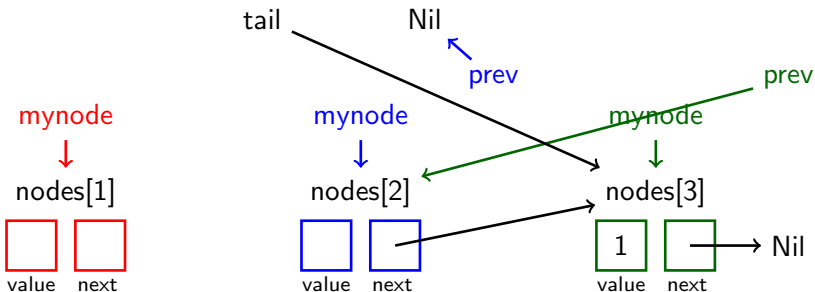


Process 3:

- (5) if `prev ≠ Nil` then
- (6) `*mynode.value := 1;`
- (7) `*prev.next := mynode;`
- (8) `await *mynode.value = 0;`

The MCS Queue-Based Algorithm: Example (2)

Process 2 in the CS, process 3 wants to enter the critical section

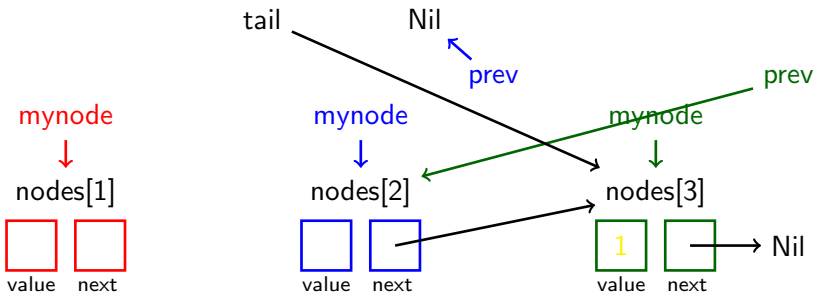


Process 3:

- (5) if `prev ≠ Nil` then
- (6) `*myNode.value := 1;`
- (7) `*prev.next := myNode;`
- (8) `await *myNode.value = 0;`

The MCS Queue-Based Algorithm: Example (2)

Process 2 in the CS, process 3 wants to enter the critical section

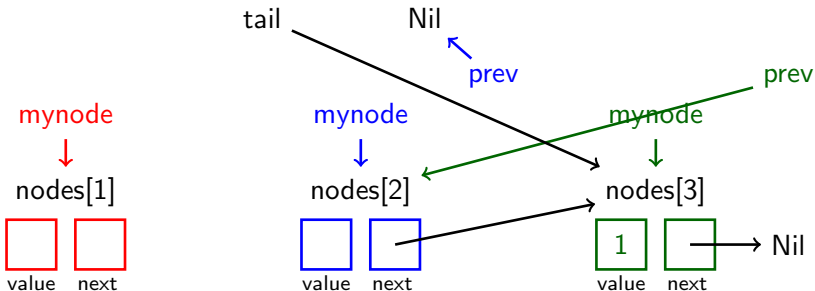


Process 3:

- (5) if `prev ≠ Nil` then
- (6) `*myNode.value := 1;`
- (7) `*prev.next := myNode;`
- (8) `await *myNode.value = 0;`

The MCS Queue-Based Algorithm: Example (2)

Process 2 in the CS, process 3 wants to enter the critical section

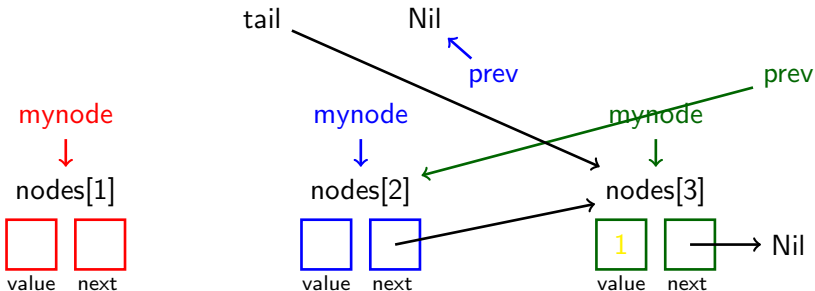


Process 3:

- (5) if `prev ≠ Nil` then
- (6) `*mynode.value := 1;`
- (7) `*prev.next := mynode;`
- (8) `await *mynode.value = 0;`

The MCS Queue-Based Algorithm: Example (2)

Process 2 in the CS, process 3 wants to enter the critical section

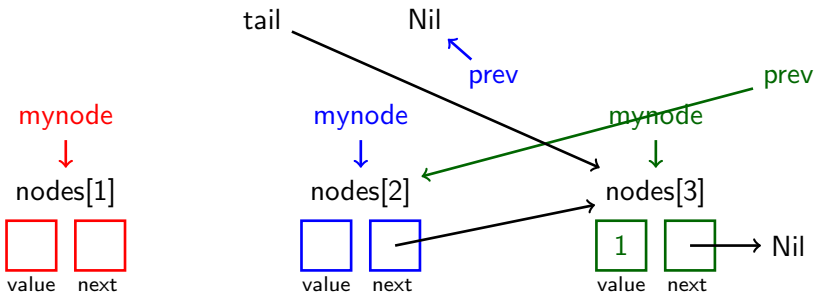


Process 3:

- (5) if `prev ≠ Nil` then
- (6) `*myNode.value := 1;`
- (7) `*prev.next := myNode;`
- (8) `await *myNode.value = 0;`

The MCS Queue-Based Algorithm: Example (2)

Process 2 in the CS, process 3 wants to enter the critical section

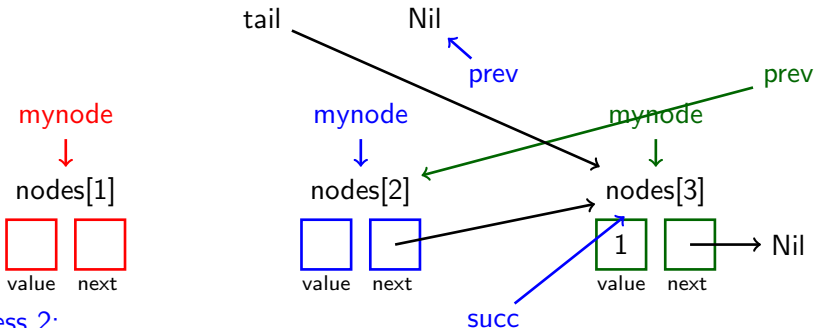


Process 3:

- (5) if `prev ≠ Nil` then
- (6) `*mynode.value := 1;`
- (7) `*prev.next := mynode;`
- (8) `await *mynode.value = 0;`

The MCS Queue-Based Algorithm: Example (2)

Process 2 in the CS, process 3 wants to enter the critical section

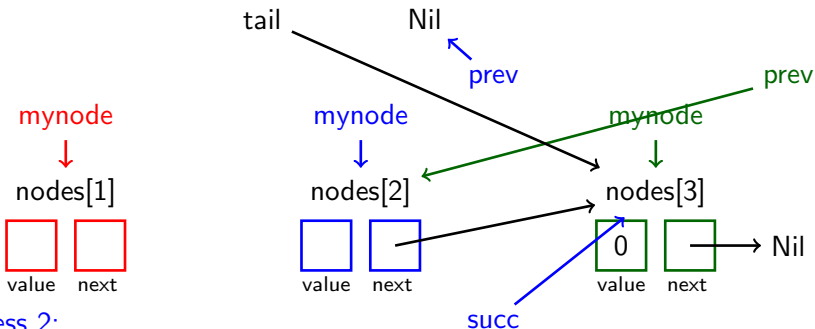


Process 2:

```
(10) if mynode.next = Nil then
(15) else
(16)   succ := *mynode.next;
(17)   *succ.value := 0;
```

The MCS Queue-Based Algorithm: Example (2)

Process 2 in the CS, process 3 wants to enter the critical section



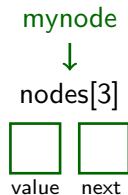
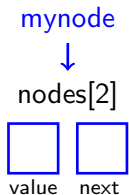
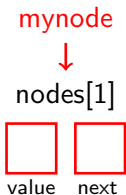
Process 2:

```
(10) if mynode.next = Nil then
(15) else
(16)   succ := *mynode.next;
(17)   *succ.value := 0;
```

The MCS Queue-Based Algorithm: Example (3)

Process 2 in CS, process 3 wants to enter CS, but 2 finishes before 3 changes the next pointer

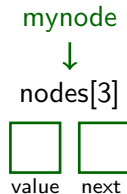
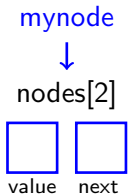
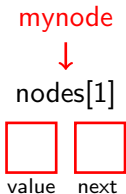
tail $\longrightarrow$ Nil



The MCS Queue-Based Algorithm: Example (3)

Process 2 in CS, process 3 wants to enter CS, but 2 finishes before 3 changes the next pointer

tail $\longrightarrow$ Nil



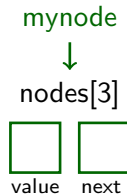
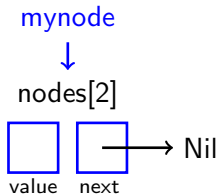
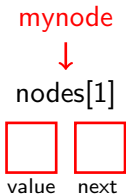
Process 2:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

The MCS Queue-Based Algorithm: Example (3)

Process 2 in CS, process 3 wants to enter CS, but 2 finishes before 3 changes the next pointer

tail $\longrightarrow$ Nil

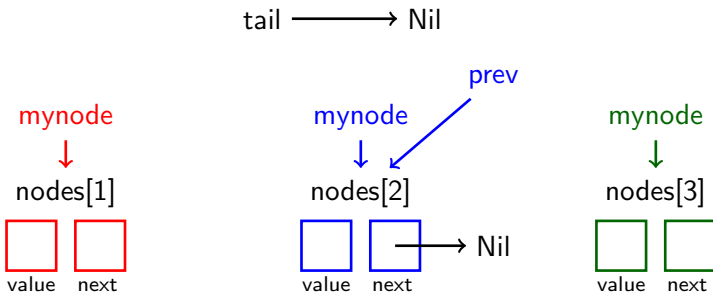


Process 2:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

The MCS Queue-Based Algorithm: Example (3)

Process 2 in CS, process 3 wants to enter CS, but 2 finishes before 3 changes the next pointer

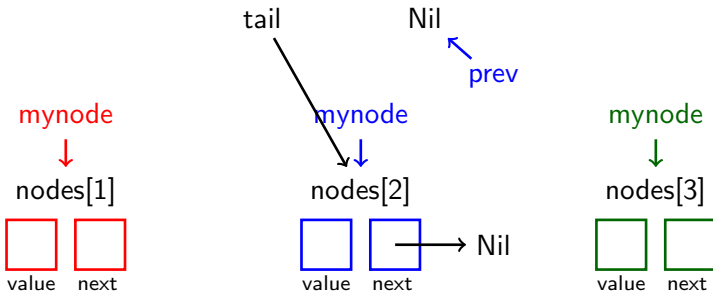


Process 2:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail, prev);`

The MCS Queue-Based Algorithm: Example (3)

Process 2 in CS, process 3 wants to enter CS, but 2 finishes before 3 changes the next pointer

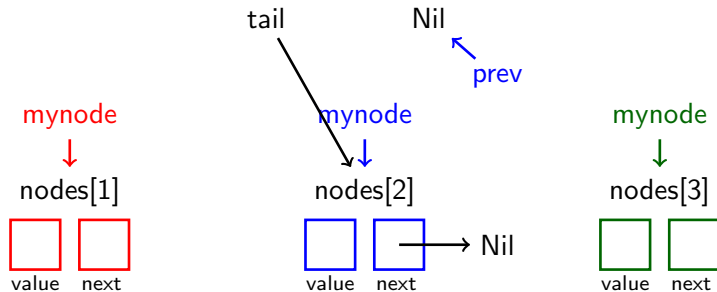


Process 2:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail, prev);`

The MCS Queue-Based Algorithm: Example (3)

Process 2 in CS, process 3 wants to enter CS, but 2 finishes before 3 changes the next pointer



Process 2:

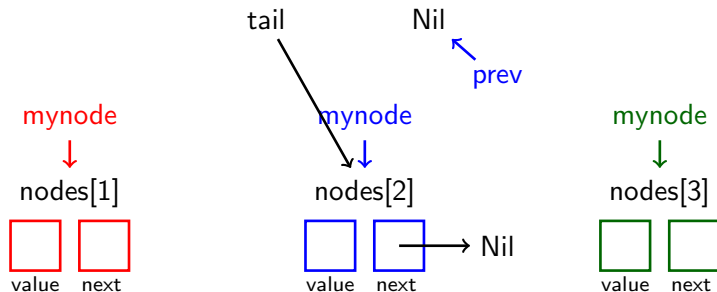
(5) if `prev` $\neq$ `Nil` then

...

(9) Critical section

The MCS Queue-Based Algorithm: Example (3)

Process 2 in CS, process 3 wants to enter CS, but 2 finishes before 3 changes the next pointer

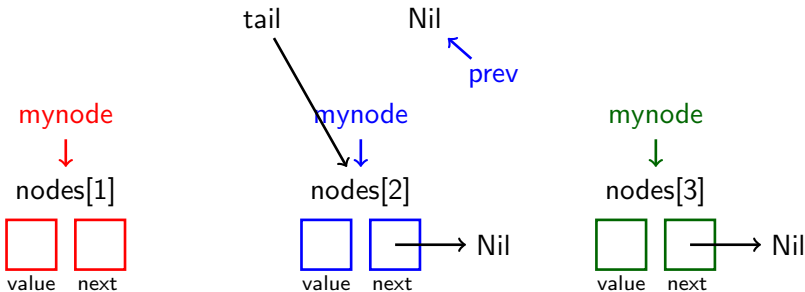


Process 3:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail, prev);`

The MCS Queue-Based Algorithm: Example (3)

Process 2 in CS, process 3 wants to enter CS, but 2 finishes before 3 changes the next pointer

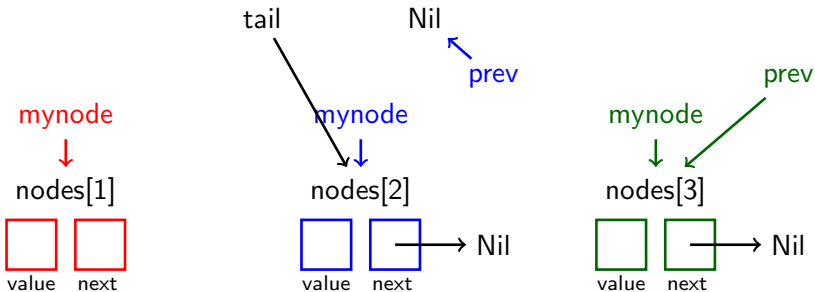


Process 3:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail, prev);`

The MCS Queue-Based Algorithm: Example (3)

Process 2 in CS, process 3 wants to enter CS, but 2 finishes before 3 changes the next pointer

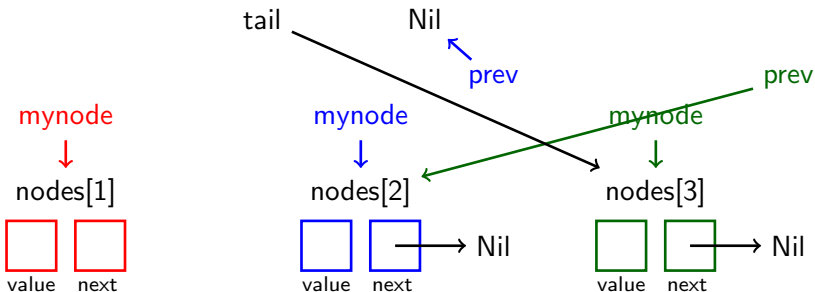


Process 3:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail, prev);`

The MCS Queue-Based Algorithm: Example (3)

Process 2 in CS, process 3 wants to enter CS, but 2 finishes before 3 changes the next pointer

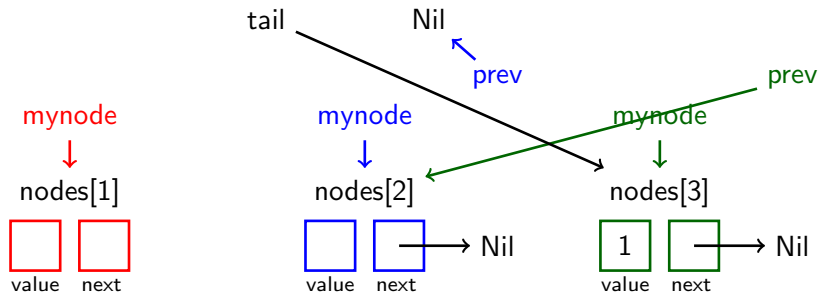


Process 3:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail, prev);`

The MCS Queue-Based Algorithm: Example (3)

Process 2 in CS, process 3 wants to enter CS, but 2 finishes before 3 changes the next pointer

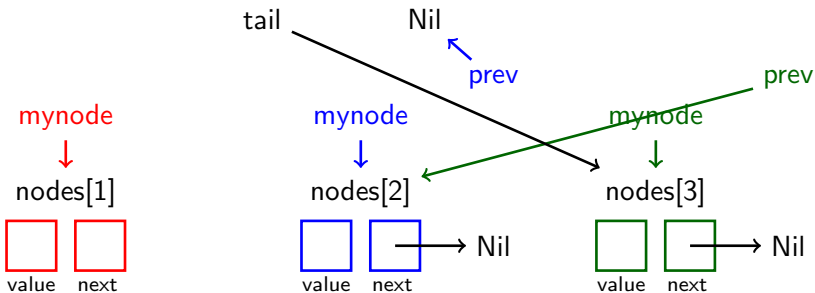


Process 3:

- (5) if `prev ≠ Nil` then
- (6) `*mynode.value := 1;`

The MCS Queue-Based Algorithm: Example (3)

Process 2 in CS, process 3 wants to enter CS, but 2 finishes before 3 changes the next pointer

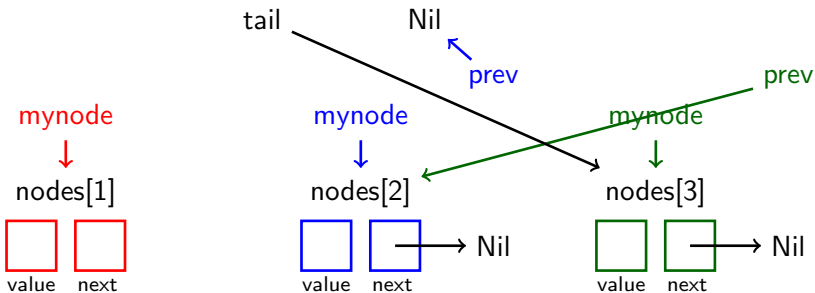


Process 3:

- (5) if `prev` $\neq$ `Nil` then
- (6) `*mynode.value := 1;`

The MCS Queue-Based Algorithm: Example (3)

Process 2 in CS, process 3 wants to enter CS, but 2 finishes before 3 changes the next pointer



Process 2:

```
(10) if mynode.next = Nil then
(11)   if compare-and-swap(tail,mynode,Nil) = False then
(12)     await *mynode.next ≠ Nil;
(13)     succ := *mynode.next;
```

- stronger memory operations than atomic read and write
- several different operations
- mutual exclusion algorithms with these operations are simple, but also non-trivial to achieve good properties (starvation freedom, FIFO properties).

Outlook:

- Which of the operations are needed?
- Which of the operations are stronger than others?

Consensus and the Herlihy Hierarchy



- Many concurrent objects: atomic register, RMW object, test-and-set object, etc.
- Which objects are "better" than the others?
- What does "better" mean?

Intuitively:

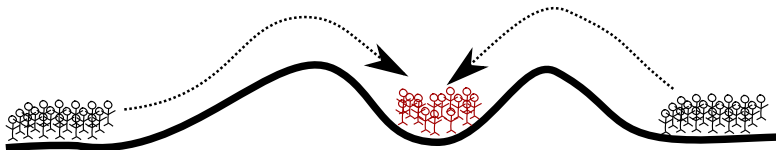
- Atomic register is rather weak (e.g., because mutual exclusion is difficult)
- RMW object is quite strong, can implement many other objects

- Maurice Herlihy investigated/formalized this in 1991
- but: Different model!
- Model: Processes can crash at any time
- Modeling: Process hangs before a command

Note: None of the mutual exclusion algorithms seen can tolerate crashes!

An Example for Crashing Processes

The two generals problem (also known as coordinated attack)

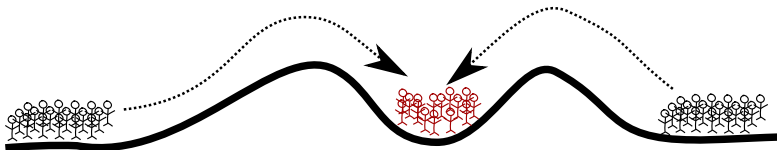


Facts:

- Attack only successful if both divisions attack **simultaneously**
- Communication only via **messengers**
- **Messengers** must pass through enemy territory and therefore **do not always** arrive

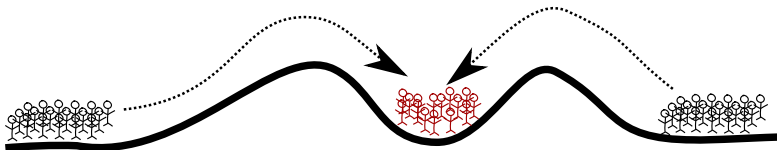
Problem: How do the generals coordinate?

Two Generals Problem



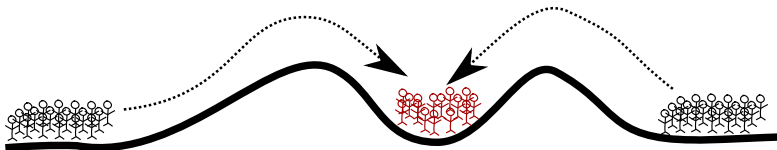
- Right general sends messenger: "Attack at 12:00 noon"

Two Generals Problem



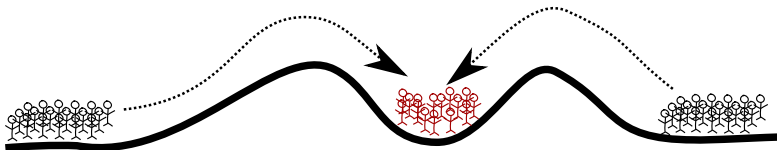
- Right general sends messenger: "Attack at 12:00 noon"
- Right general cannot be sure that the messenger will arrive
⇒ waits for confirmation from left general

Two Generals Problem



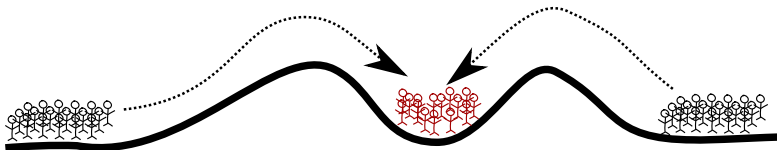
- Right general sends messenger: "Attack at 12:00 noon"
- Right general cannot be sure that the messenger will arrive
⇒ waits for confirmation from left general
- Left general sends messenger: "Message received"

Two Generals Problem



- Right general sends messenger: "Attack at 12:00 noon"
- Right general cannot be sure that the messenger will arrive
⇒ waits for confirmation from left general
- Left general sends messenger: "Message received"
- Left general cannot be sure that the messenger will arrive
⇒ waits for confirmation from right general

Two Generals Problem

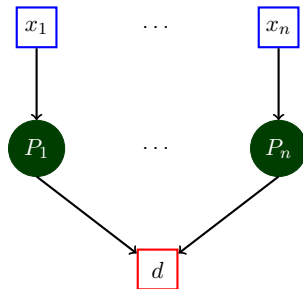


- Right general sends messenger: "Attack at 12:00 noon"
- Right general cannot be sure that the messenger will arrive
⇒ waits for confirmation from left general
- Left general sends messenger: "Message received"
- Left general cannot be sure that the messenger will arrive
⇒ waits for confirmation from right general
- Right general sends messenger: "Message received"
- etc.

In fact: Problem is unsolvable in this model!

The Consensus Problem

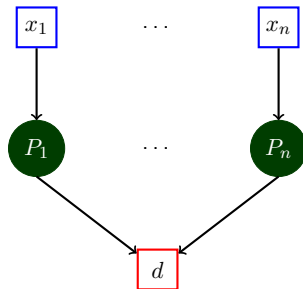
- n **processes**, which may also crash
- process i receives a **input value** $x_i \in \{0, 1\}$
- program the processes so that all (non-crashing) processes decide on a common **decision value** $d \in \{0, 1\}$



- **Agreement**: All non-crashed processes decide on the same value d .
- **Validity**: $d \in \{x_1, \dots, x_n\}$, i.e., d is one of the input values.

The Consensus Problem

- n **processes**, which may also crash
- process i receives a **input value** $x_i \in \{0, 1\}$
- program the processes so that all (non-crashing) processes decide on a common **decision value** $d \in \{0, 1\}$



- **Agreement**: All non-crashed processes decide on the same value d .
- **Validity**: $d \in \{x_1, \dots, x_n\}$, i.e., d is one of the input values.

Note: Agreement $\Rightarrow$ Algorithm is wait-free:

Each non-crashing process decides after a finite number of steps.

The Consensus Problem (2)

For non-crashing processes, the problem would be simple:

- All processes share all their input values with each other (via shared memory)
- then each process calculates $d := f(x_1, \dots, x_n)$
- where f is a fixed function known to all processes, e.g., $f(x_1, \dots, x_n) = x_1$.

Solution with Three-Valued RMW Object

Objects and initialization:

x : RMW object with the possible values $\perp, 0, 1$, initial $\perp$

x_i : Input value of process i

d_i : Decision value made by process i .

Program of the i -th process

- (1) $d_i := \text{read-modify-write}(x, f_i);$
- (2) if $d_i = \perp$ then
 $d_i := x_i;$

Function f_i of the i -th process

```
function  $f_i(v)$   
  if  $v = \perp$  then return  $x_i$   
  else return  $v$   
end function
```

Solution with Three-Valued RMW Object

Objects and initialization:

x : RMW object with the possible values $\perp, 0, 1$, initial $\perp$

x_i : Input value of process i

d_i : Decision value made by process i .

Program of the i -th process

- (1) $d_i := \text{read-modify-write}(x, f_i);$
- (2) if $d_i = \perp$ then
 $d_i := x_i;$

Function f_i of the i -th process

```
function  $f_i(v)$ 
  if  $v = \perp$  then return  $x_i$ 
  else return  $v$ 
end function
```

} first process sets its x_i as the new value,
 all other non-crashing processes read this value
 $\Rightarrow$ all d_i values are identical

Solution to the Consensus Problem

With a three-valued RMW object:

- The algorithm is a wait-free solution
- for any number of processes

Result for **atomic registers with *read* and *write***:

Even for 2 processes and only one crash, it cannot be solved:

Theorem

There is no consensus algorithm for atomic *read* and *write* that can tolerate a crash.

Conclusion: Atomic registers cannot simulate an RMW object

- Instead of arguing directly with crashes, we show:
 - A wait-free consensus algorithm for atomic registers and 2 processes does not exist
- Algorithms that tolerate crashes must be wait-free, because:
 - not wait-free
 - ⇒ process waits for condition that another process makes true
 - ⇒ infinite loop if other process crashes
 - ⇒ crash not tolerable

- We use computation trees to consider the execution of two processes P_1 and P_2
- Computation tree:
 - Nodes: States of the processes and the memory
 - Edges: Essential steps, i.e., read or write operations on the shared memory (internal steps are summarized)
 - Edge to the left = step from P_1
 - Edge to the right = step from P_2
 - ...

- Computation tree:
 - ...
 - Leaves are marked with decision value $\in \{0, 1\}$.
 - Inner nodes with the still possible decision values.
 - Bivalent node: Marking 0 and 1
 - Univalent nodes: Marking 0 (0-valent) or marking 1 (1-valent)
 - Since the algorithm is wait-free, the tree is finite!

Statement:

Every 2-process consensus algorithm has a bivalent initial state.

Proof:

- Assume P_1 has input value $x_1 = 0$ and P_2 has input value $x_2 = 1$.
- If only P_1 takes steps (leftmost path in the tree), then P_1 must compute 0 as the decision value.
- If only P_2 takes steps, then 1 must be computed as the decision value. □

Proof Sketch (5)

A node is **critical** if it is bivalent and each successor is univalent.

Statement:

Every computation tree with a bivalent initial state
for a consensus algorithm has a critical node.

Proof:

- Assume this is not true.
- Start with a bivalent initial state.
- As long as there are steps that lead to a bivalent state, execute these steps.
- If this is possible indefinitely, then the algorithm is not wait-free.

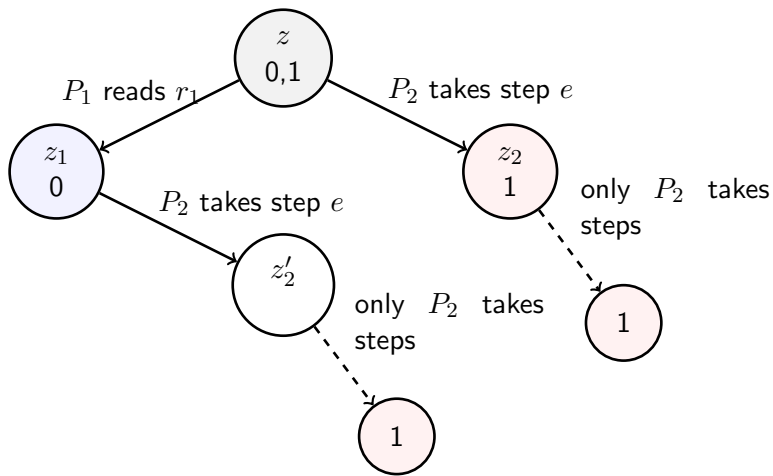
Contradiction!



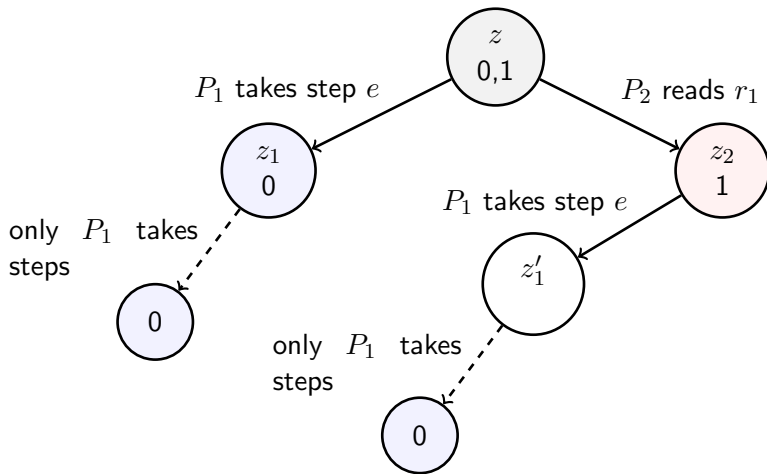
Proof Sketch (6)

- Now let there be a consensus algorithm for 2 processes.
- Go to a critical bivalent state z
- W.l.o.g. left child is 0-valent, right child is 1-valent
- Examine all cases.

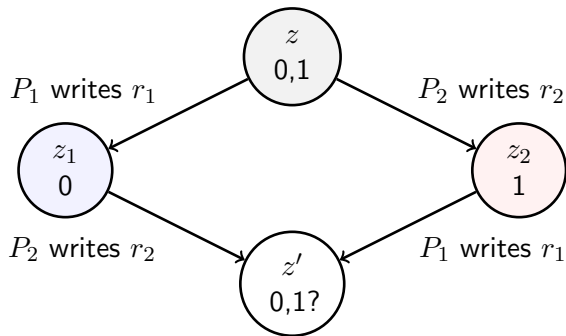
Proof Sketch (7)



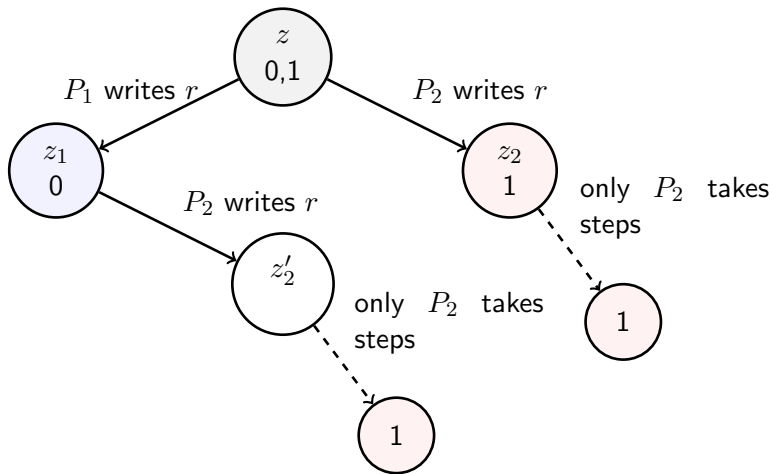
Proof Sketch (8)



Proof Sketch (9)



Proof Sketch (10)



Definition

For a concurrent object of type o , the **consensus number** $\mathbb{CN}(o)$ is the largest number of processes n for which the consensus problem for n processes can be solved by using any number of objects of type o and any number of atomic registers (with *read* and *write*).

If the number is unlimited, let $\mathbb{CN}(o) = \infty$.

$\mathbb{CN}(o)$	Object o
1	atomic register with <i>read</i> and <i>write</i>
2	test-and-set object, fetch-and-increment object, fetch-and-add object, swap object, read-modify-write bit
$\Theta(\sqrt{m})$	swap ^{m} object
$2m - 2$	m -register with m -fold assignment ($m > 1$)
∞	(three-valued) RMW object, compare-and-swap object, sticky bit

Theorem

If o_1 and o_2 are two objects with $\text{CN}(o_1) < \text{CN}(o_2)$, then the following applies to a system with $\text{CN}(o_2)$ processes:

- There is no wait-free implementation of object o_2 exclusively with objects of type o_1 and atomic registers.
 - There is a wait-free implementation of object o_1 exclusively with objects of type o_2 and atomic registers.
-
- For part 1: If there were an implementation of o_2 objects with o_1 objects, it could also be used to solve the consensus problem for $\text{CN}(o_2)$ processes. Then $\text{CN}(o_1) \geq \text{CN}(o_2)$ would hold. Contradiction!
 - For part 2: The statement follows from the next theorem (next slide), which also states: With o_2 -objects and atomic registers, any object with a sequential description for $\text{CN}(o_2)$ processes can be implemented wait-free.

Theorem

An object o with $\mathbb{CN}(o) \geq n$ is **universal** in a system with n processes. This means that every wait-free object (with a sequential description) can be implemented by objects of type o and atomic registers in a system with n processes.

Remarks:

- We will not provide the proof (this can be found in the literature).
- Key ideas of the proof:
 - Each process stores the accesses to the objects and calculates the successor states of the objects until its access takes place.
 - Computing is not a real problem due to the sequential description.
 - The hurdles are:
 - When does which process execute an operation?
The consensus algorithm is used for these decisions!
 - Ensuring that the operations are wait-free.

Consequences of the theorem:

- Universal objects can implement many other objects (regardless of the consensus problem)
- Objects with consensus number $\geq n$ are “sufficient.”