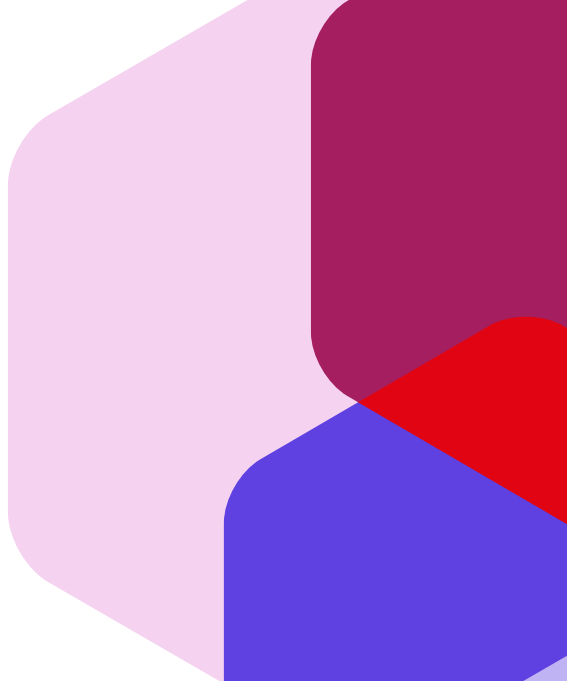


Principles, Algorithms and Models of Concurrency and Synchronization

04 Programming Primitives



In the last lectures:

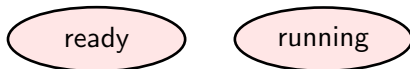
- Primitive atomic operations implemented by hardware
- Solutions to the mutual exclusion problem with the various primitives

Now:

- Software solutions
- Primitives provided by concurrent programming languages
- Applications (classical problems)

Process Model (1)

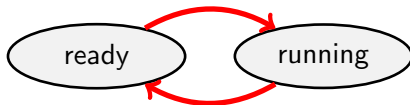
Processes P have a state $P.state$:



- **running**: Process executes steps
- **ready**: Process wants to execute steps, but is not allowed to
- at least one process is always running (e.g., **idle process**)

Process Model (1)

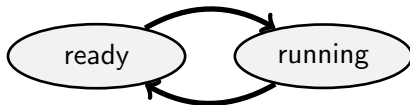
Processes P have a state $P.state$:



- **running**: Process executes steps
- **ready**: Process wants to execute steps, but is not allowed to
- at least one process is always running (e.g., **idle process**)
- **scheduler** performs **context switch**:
ready processes become running processes and vice versa

Process Model (1)

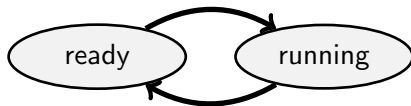
Processes P have a state $P.state$:



- **running**: Process executes steps
- **ready**: Process wants to execute steps, but is not allowed to
- at least one process is always running (e.g., **idle process**)
- **scheduler** performs **context switch**:
ready processes become running processes and vice versa
- **Fairness**: Every ready process becomes running after a finite number of steps

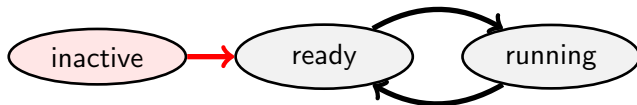
Process Model (2)

Processes P have a state $P.state$:



Process Model (2)

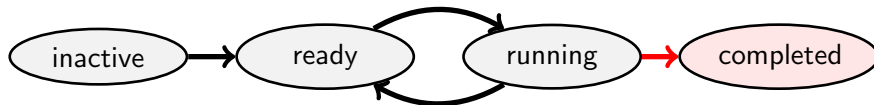
Processes P have a state $P.state$:



- **inactive**: not yet ready (e.g., code is loading)

Process Model (2)

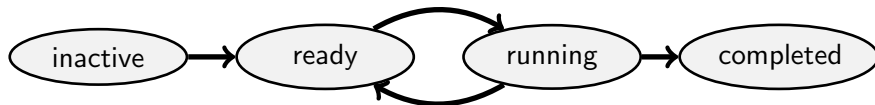
Processes P have a state $P.state$:



- **inactive**: not yet ready (e.g., code is loading)
- **completed**: process terminated

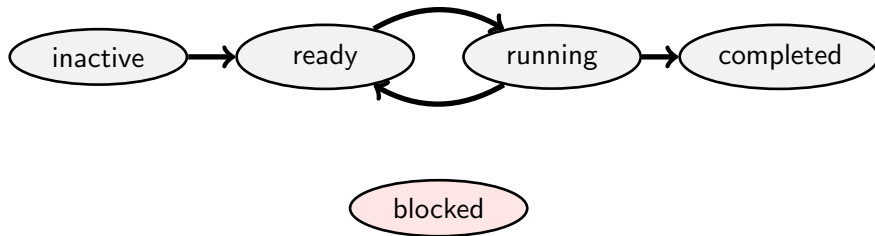
Process Model (3)

Processes P have a state $P.state$:



Process Model (3)

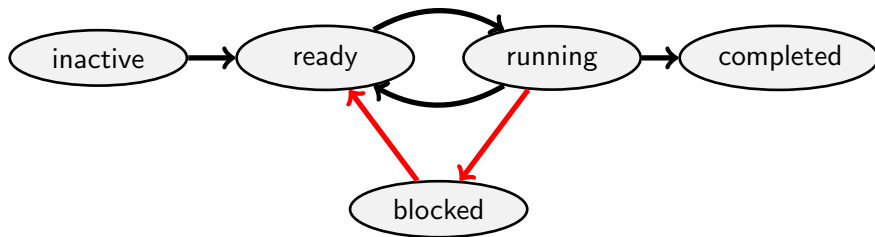
Processes P have a state $P.state$:



- blocked: process may not execute any steps

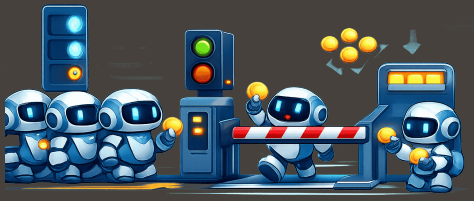
Process Model (3)

Processes P have a state $P.state$:



- blocked: process may not execute any steps
- block/unblock via program commands,
not via scheduler

Semaphores

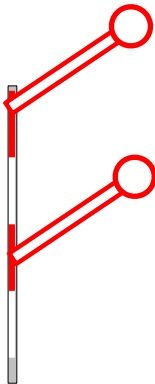


Semaphore

Origin of term:

Semaphore =

Mechanical signal device in rail transport



Semaphore

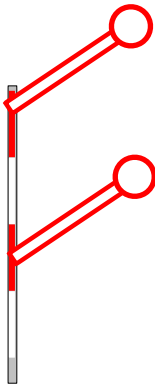
Origin of term:

Semaphore =

Mechanical signal device in rail transport

In computer science:

Abstract data type with operations



Semaphore S

Attributes (in general):

- V = Non-negative integer
- M = Set of processes

Notation for semaphore S : $S.V$ and $S.M$

Semaphore S

Attributes (in general):

- V = Non-negative integer
- M = Set of processes

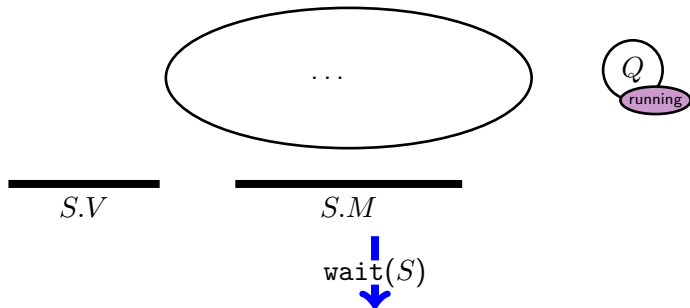
Notation for semaphore S : $S.V$ and $S.M$

Operations:

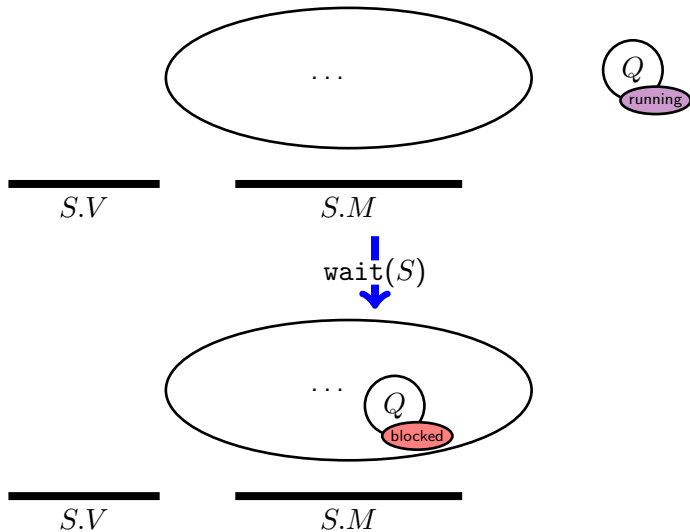
- `newSem( $k$ )` creates a new semaphore with $S.V = k$ and $S.M = \emptyset$
- `wait( $S$ )` (alternatively: `P( $S$ )` (Dijkstra, *prolaag* (artificial word, instead of *verlaag* (Dutch for “lower”))) or `down( $S$ )`)
- `signal( $S$ )` (alternatively: `V( $S$ )` (Dijkstra, *verhoog* (Dutch for “increase”)) or `up( $S$ )`)
- are executed **atomically** (from the programmer’s point of view, i.e., the programming language ensures “correct” implementation)

Semaphore: Wait from Process Q if $S.V = 0$

hsrm*

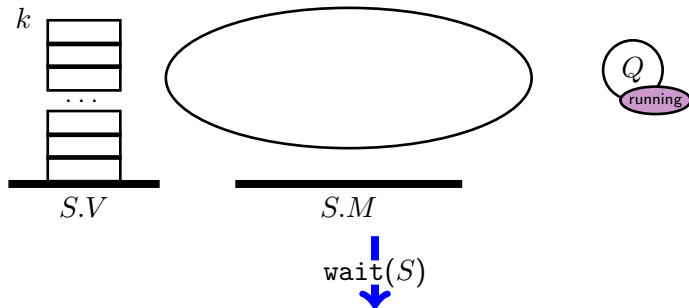


Semaphore: Wait from Process Q if $S.V = 0$

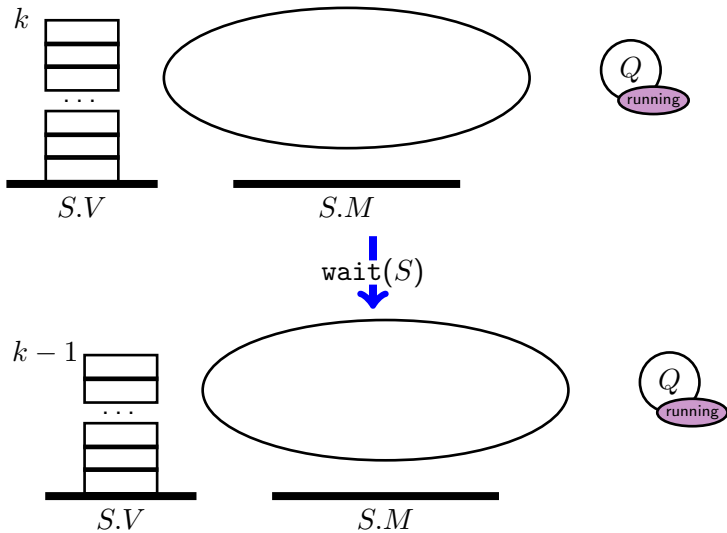


Semaphore: Wait from Process Q if $S.V > 0$

hsrm*



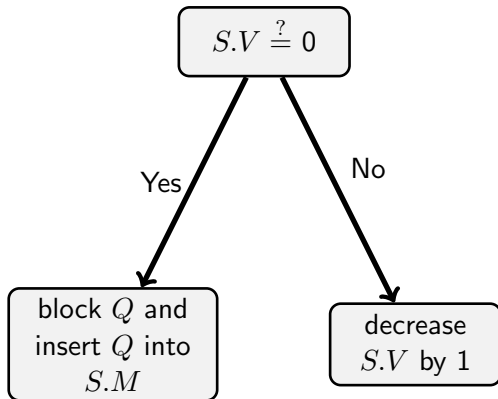
Semaphore: Wait from Process Q if $S.V > 0$



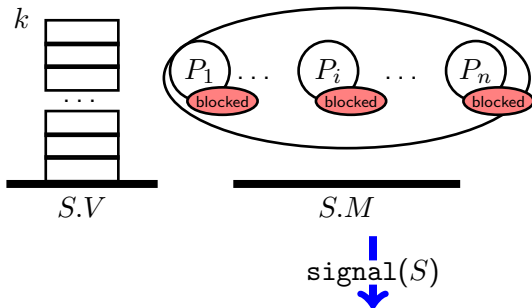
wait(S)

Let Q be the calling process:

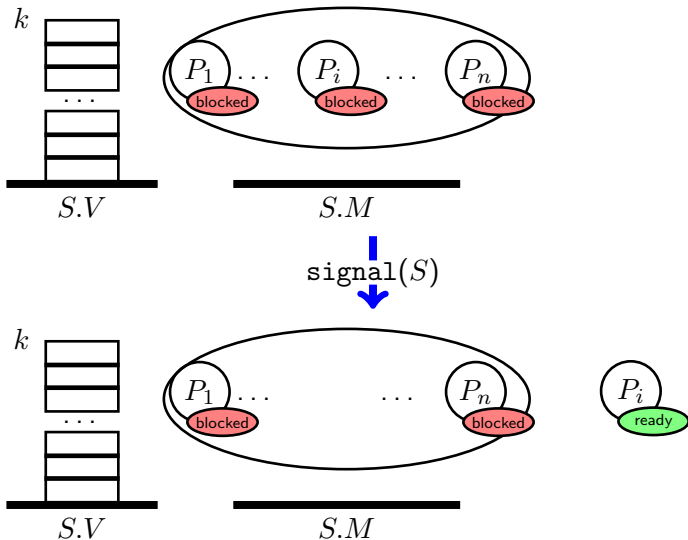
```
procedure wait( $S$ )  
  if  $S.V > 0$  then  
     $S.V := S.V - 1$ ;  
  else  
     $S.M := S.M \cup \{Q\}$ ;  
     $Q.state := blocked$ ;
```



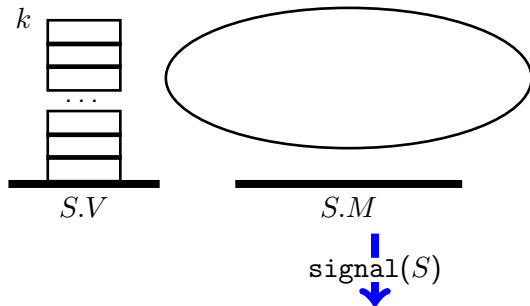
Semaphore: Signal if $S.M \neq \emptyset$



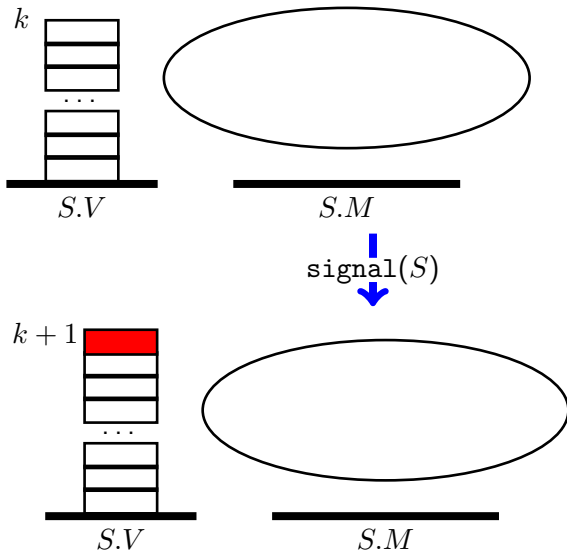
Semaphore: Signal if $S.M \neq \emptyset$



Semaphore: Signal if $S.M = \emptyset$

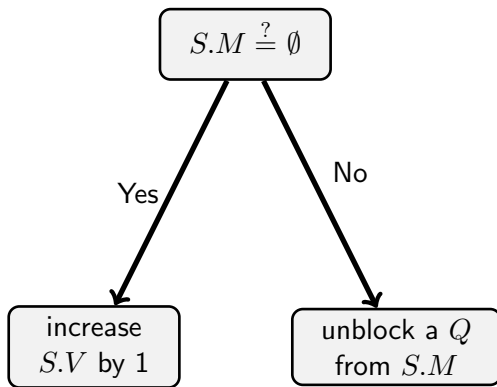


Semaphore: Signal if $S.M = \emptyset$



signal(S)

```
procedure signal( $S$ )  
  if  $S.M = \emptyset$  then  
     $S.V := S.V + 1$ ;  
  else  
    select an element  $Q$  from  $S.M$ ;  
     $S.M := S.M \setminus \{Q\}$ ;  
     $Q.state := ready$ ;
```



Let S be a semaphore initialized with k . After executing each computation sequence $\mathcal{P}$, the following applies:

- $S.V \geq 0$
- $S.V = k + |S.M| + \#_{\text{signal}}(S, \mathcal{P}) - \#_{\text{wait}}(S, \mathcal{P})$

where

- $\#_{\text{signal}}(S, \mathcal{P})$ = number of `signal` operations in $\mathcal{P}$
- $\#_{\text{wait}}(S, \mathcal{P})$ = number of `wait` operations in $\mathcal{P}$

Let S be a semaphore initialized with k . After executing each computation sequence $\mathcal{P}$, the following applies:

- $S.V \geq 0$
- $S.V = k + \underset{-1}{|S.M|} + \underset{+1}{\#_{signal}(S, \mathcal{P})} - \underset{+1}{\#_{wait}(S, \mathcal{P})}$ wait(S)

where

- $\#_{signal}(S, \mathcal{P})$ = number of `signal` operations in $\mathcal{P}$
- $\#_{wait}(S, \mathcal{P})$ = number of `wait` operations in $\mathcal{P}$

Let S be a semaphore initialized with k . After executing each computation sequence $\mathcal{P}$, the following applies:

- $S.V \geq 0$
 - $S.V = k + |S.M| + \#_{signal}(S, \mathcal{P}) - \#_{wait}(S, \mathcal{P})$
- -1

 $+1$

 -1

$+1$

 -1

$+1$

 $+1$

$+1$

 $+1$

$wait(S)$

 $signal(S)$

where

- $\#_{signal}(S, \mathcal{P})$ = number of `signal` operations in $\mathcal{P}$
- $\#_{wait}(S, \mathcal{P})$ = number of `wait` operations in $\mathcal{P}$

Binary Semaphores

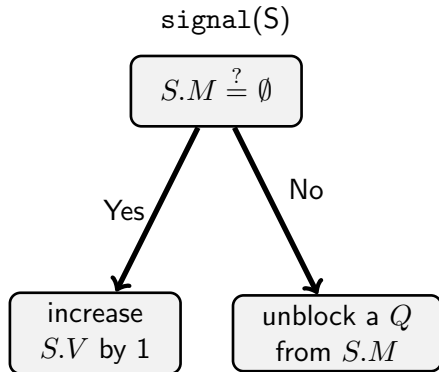
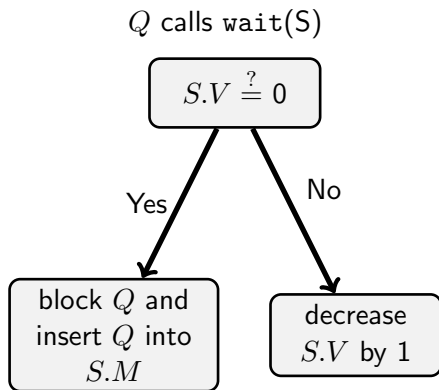
- we had **general** semaphores, for **binary semaphores**:
 - $0 \leq S.V \leq 1$
 - wait unchanged
 - signal must not increase arbitrarily:

```
procedure signal( $S$ )  
  if  $S.V = 1$  then  
    undefined  
  else if  $S.M = \emptyset$  then  
     $S.V := 1$   
  else  
    select an element  $Q$  from  $S.M$ ;  
     $S.M := S.M \setminus \{Q\}$ ;  
     $Q.state := ready$ ;
```

Binary Semaphores (2)

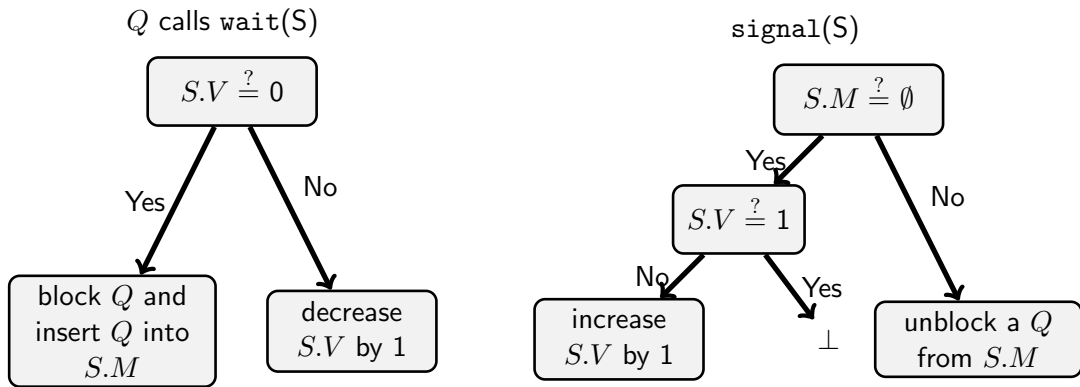
- are often referred to as **mutex** in programming languages
- invariants still hold **if** a corresponding wait is executed before each signal

Summary: Semaphore



Summary: Binary Semaphore

V only 0 or 1



Mutual Exclusion with Binary Semaphores

Initial: Let S be a binary semaphore, initialized with 1

Program of the i -th process

```
loop forever
(P1)  remaining code
(P2)  wait(S)
(P3)  critical section
(P4)  signal(S)
end loop
```

Theorem

The algorithm guarantees mutual exclusion and is deadlock-free.

Proof: Mutual exclusion

- $\#_{CS}(\mathcal{P})$ = Number of processes in the critical section after execution of $\mathcal{P}$
- $\#_{CS}(\mathcal{P}) = \#_{wait}(S, \mathcal{P}) - \#_{signal}(S, \mathcal{P}) - |S.M|$
- With invariant $S.V = k + |S.M| + \#_{signal}(S, \mathcal{P}) - \#_{wait}(S, \mathcal{P})$ this yields

$$\begin{aligned}\#_{CS}(\mathcal{P}) + S.V &= \#_{wait}(S, \mathcal{P}) - \#_{signal}(S, \mathcal{P}) - |S.M| \\ &\quad + k + |S.M| + \#_{signal}(S, \mathcal{P}) - \#_{wait}(S, \mathcal{P}) \\ &= k\end{aligned}$$

- Since $k = 1$, the following must hold: $\#_{CS}(\mathcal{P}) \leq 1$ (i.e., mutual exclusion)

Theorem

The algorithm guarantees mutual exclusion and is deadlock-free.

Proof: Deadlock-freedom

- $\mathcal{P}$ = infinitely long computation sequence such that deadlock occurs
- $\mathcal{P}_1$ = prefix of $\mathcal{P}$ such that from $\mathcal{P}_1$:
 - No process in the critical section, i.e., $\#_{CS}(\mathcal{P}_1) = 0$
 - At least one process P is waiting (is blocked), i.e., $S.V = 0$ and $P \in S.M$
- Impossible, since $\#_{CS}(\mathcal{P}_1) + S.V = 1$

Additional Properties

- With a maximum of 2 processes: Algorithm is starvation-free.
- With more processes: **not starvation-free**:

$S.V$	$S.M$	Process 1	Process 2	Process 3
1	$\emptyset$	rem. code	rem. code	rem. code

Additional Properties

- With a maximum of 2 processes: Algorithm is starvation-free.
- With more processes: **not starvation-free**:

$S.V$	$S.M$	Process 1	Process 2	Process 3
1	$\emptyset$	rem. code	rem. code	rem. code
1	$\emptyset$	wait(S)	rem. code	rem. code

Additional Properties

- With a maximum of 2 processes: Algorithm is starvation-free.
- With more processes: **not starvation-free**:

$S.V$	$S.M$	Process 1	Process 2	Process 3
1	$\emptyset$	rem. code	rem. code	rem. code
1	$\emptyset$	wait(S)	rem. code	rem. code
0	$\emptyset$	critical section	wait(S)	rem. code

Additional Properties

- With a maximum of 2 processes: Algorithm is starvation-free.
- With more processes: **not starvation-free**:

$S.V$	$S.M$	Process 1	Process 2	Process 3
1	$\emptyset$	rem. code	rem. code	rem. code
1	$\emptyset$	<code>wait(S)</code>	rem. code	rem. code
0	$\emptyset$	critical section	<code>wait(S)</code>	rem. code
0	$\{2\}$	critical section	(blocked)	<code>wait(S)</code>

Additional Properties

- With a maximum of 2 processes: Algorithm is starvation-free.
- With more processes: **not starvation-free**:

$S.V$	$S.M$	Process 1	Process 2	Process 3
1	$\emptyset$	rem. code	rem. code	rem. code
1	$\emptyset$	<code>wait(S)</code>	rem. code	rem. code
0	$\emptyset$	critical section	<code>wait(S)</code>	rem. code
0	$\{2\}$	critical section	(blocked)	<code>wait(S)</code>
0	$\{2, 3\}$	critical section	(blocked)	(blocked)

Additional Properties

- With a maximum of 2 processes: Algorithm is starvation-free.
- With more processes: **not starvation-free**:

$S.V$	$S.M$	Process 1	Process 2	Process 3
1	$\emptyset$	rem. code	rem. code	rem. code
1	$\emptyset$	<code>wait(S)</code>	rem. code	rem. code
0	$\emptyset$	critical section	<code>wait(S)</code>	rem. code
0	$\{2\}$	critical section	(blocked)	<code>wait(S)</code>
0	$\{2, 3\}$	critical section	(blocked)	(blocked)
0	$\{2, 3\}$	<code>signal(S)</code>	(blocked)	(blocked)

Additional Properties

- With a maximum of 2 processes: Algorithm is starvation-free.
- With more processes: **not starvation-free**:

$S.V$	$S.M$	Process 1	Process 2	Process 3
1	$\emptyset$	rem. code	rem. code	rem. code
1	$\emptyset$	<code>wait(S)</code>	rem. code	rem. code
0	$\emptyset$	critical section	<code>wait(S)</code>	rem. code
0	$\{2\}$	critical section	(blocked)	<code>wait(S)</code>
0	$\{2, 3\}$	critical section	(blocked)	(blocked)
0	$\{2, 3\}$	<code>signal(S)</code>	(blocked)	(blocked)
0	$\{3\}$	rem. code	critical section	(blocked)

Additional Properties

- With a maximum of 2 processes: Algorithm is starvation-free.
- With more processes: **not starvation-free**:

$S.V$	$S.M$	Process 1	Process 2	Process 3
1	$\emptyset$	rem. code	rem. code	rem. code
1	$\emptyset$	wait(S)	rem. code	rem. code
0	$\emptyset$	critical section	wait(S)	rem. code
0	$\{2\}$	critical section	(blocked)	wait(S)
0	$\{2, 3\}$	critical section	(blocked)	(blocked)
0	$\{2, 3\}$	signal(S)	(blocked)	(blocked)
0	$\{3\}$	rem. code	critical section	(blocked)
0	$\{1, 3\}$	wait(S)	critical section	(blocked)

Additional Properties

- With a maximum of 2 processes: Algorithm is starvation-free.
- With more processes: **not starvation-free**:

$S.V$	$S.M$	Process 1	Process 2	Process 3
1	$\emptyset$	rem. code	rem. code	rem. code
1	$\emptyset$	<code>wait(S)</code>	rem. code	rem. code
0	$\emptyset$	critical section	<code>wait(S)</code>	rem. code
0	$\{2\}$	critical section	(blocked)	<code>wait(S)</code>
0	$\{2, 3\}$	critical section	(blocked)	(blocked)
0	$\{2, 3\}$	<code>signal(S)</code>	(blocked)	(blocked)
0	$\{3\}$	rem. code	critical section	(blocked)
0	$\{1, 3\}$	<code>wait(S)</code>	critical section	(blocked)
0	$\{1, 3\}$	(blocked)	<code>signal(S)</code>	(blocked)

Additional Properties

- With a maximum of 2 processes: Algorithm is starvation-free.
- With more processes: **not starvation-free**:

$S.V$	$S.M$	Process 1	Process 2	Process 3
1	$\emptyset$	rem. code	rem. code	rem. code
1	$\emptyset$	$\text{wait}(S)$	rem. code	rem. code
0	$\emptyset$	critical section	$\text{wait}(S)$	rem. code
0	$\{2\}$	critical section	(blocked)	$\text{wait}(S)$
0	$\{2, 3\}$	critical section	(blocked)	(blocked)
0	$\{2, 3\}$	$\text{signal}(S)$	(blocked)	(blocked)
0	$\{3\}$	rem. code	critical section	(blocked)
0	$\{1, 3\}$	$\text{wait}(S)$	critical section	(blocked)
0	$\{1, 3\}$	(blocked)	$\text{signal}(S)$	(blocked)
0	$\{3\}$	critical section	rem. code	(blocked)

Additional Properties

- With a maximum of 2 processes: Algorithm is starvation-free.
- With more processes: **not starvation-free**:

$S.V$	$S.M$	Process 1	Process 2	Process 3
1	$\emptyset$	rem. code	rem. code	rem. code
1	$\emptyset$	wait(S)	rem. code	rem. code
0	$\emptyset$	critical section	wait(S)	rem. code
0	$\{2\}$	critical section	(blocked)	wait(S)
0	$\{2, 3\}$	critical section	(blocked)	(blocked)
0	$\{2, 3\}$	signal(S)	(blocked)	(blocked)
0	$\{3\}$	rem. code	critical section	(blocked)
0	$\{1, 3\}$	wait(S)	critical section	(blocked)
0	$\{1, 3\}$	(blocked)	signal(S)	(blocked)
0	$\{3\}$	critical section	rem. code	(blocked)
0	$\{3\}$	critical section	wait(S)	(blocked)

Additional Properties

- With a maximum of 2 processes: Algorithm is starvation-free.
- With more processes: **not starvation-free**:

$S.V$	$S.M$	Process 1	Process 2	Process 3
1	$\emptyset$	rem. code	rem. code	rem. code
1	$\emptyset$	wait(S)	rem. code	rem. code
0	$\emptyset$	critical section	wait(S)	rem. code
0	$\{2\}$	critical section	(blocked)	wait(S)
0	$\{2, 3\}$	critical section	(blocked)	(blocked)
0	$\{2, 3\}$	signal(S)	(blocked)	(blocked)
0	$\{3\}$	rem. code	critical section	(blocked)
0	$\{1, 3\}$	wait(S)	critical section	(blocked)
0	$\{1, 3\}$	(blocked)	signal(S)	(blocked)
0	$\{3\}$	critical section	rem. code	(blocked)
0	$\{3\}$	critical section	wait(S)	(blocked)
0	$\{2, 3\}$	critical section	(blocked)	(blocked)

Other Types of Semaphores

- Previous semaphores are **weak semaphores**, as the selection of the process to be unblocked is arbitrary.
- **Strong semaphores**: FIFO order

Queue / list $S.L$ instead of set $S.M$

```
procedure wait( $S$ )  
  if  $S.V > 0$  then  
     $S.V := S.V - 1$ ;  
  else  
     $S.L := \text{append}(S.L, P)$ ;  
     $P.\text{state} := \text{blocked}$ ;
```

```
procedure signal( $S$ )  
  if  $\text{isEmpty}(S.L)$  then  
     $S.V := S.V + 1$ ;  
  else  
     $Q := \text{head}(S.L)$ ;  
     $S.L := \text{tail}(S.L)$ ;  
     $Q.\text{state} := \text{ready}$ ;
```

- With strong semaphore: Algorithm is starvation-free & satisfies FIFO property

Other Types of Semaphores (2)

- **Unfair** or **Busy-Wait** semaphores
- No guarantee when a blocked process becomes unblocked
- Blocked = busy waiting

No $S.M$ component, only $S.V$

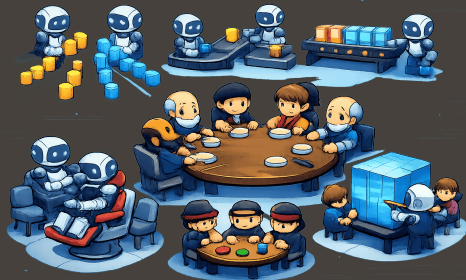
```
procedure wait( $S$ )  
  await  $S.V > 0$ ;  
   $S.V := S.V - 1$ ;
```

```
procedure signal( $S$ )  
   $S.V := S.V + 1$ ;
```

- Algorithm not starvation-free even with 2 processes.

Application Examples and Problem Solutions

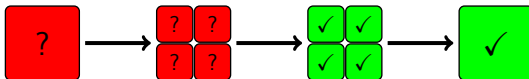
- Coordination - Example: Merge sort
- Producer / consumer: Infinite / bounded buffer
- Dining philosophers
- The sleeping barber
- Cigarette smoker's problem
- Readers & writers



Fork-Join Pattern for Parallelization

Idea analogous to Divide-and-Conquer:

- Solve subproblems with threads
- Wait for threads to finish
- Assemble overall solution



Mergesort: Coordinating the Order

Parallelization of Mergesort with Fork-Join Pattern:

- Divide input into two halves
- Sort both halves (recursively) in parallel (fork)
- Then merge the results (join)

Problem: Merge only **after** the two halves have been sorted.

Mergesort with Binary Semaphore

Initial: left, right: Binary semaphore initialized with 0

merge process:

- (1) wait(left);
- (2) wait(right);
- (3) merge

Process for left half

- (1) sort left half;
- (2) signal(left);

Process for right half

- (1) sort right half;
- (2) signal(right);

This requires 2 semaphores per recursion step!

Problems:

- Naive parallelization leads to **slowdown**
- Too many threads for too few processor cores

Solutions:

- Create few threads
- Estimate or test the number
- Even better: Use a thread pool
(limit the number of threads, but threads can process several tasks in succession)

Producer / Consumer

Problem:

- Producers produce data
- Consumers consume data
- Example: Keyboard / operating system, etc.

Exchange via buffer:

- Queue / list
- Producer writes to the end of the list
- Consumer consumes from the front of the list
- Two variants:
 - List of any length (infinite buffer)
 - Limited space (bounded buffer)

Producer / Consumer with Infinite Buffer

Requirements:

- Reading / writing to the buffer **correctly** (atomic)
- Consumer needs protection in case the buffer is empty

Producer/Consumer with Infinite Buffer (2)

Initial: notEmpty: General semaphore, initialized with 0
 mutex: Binary semaphore, initialized with 1
 l: List

Producer (produces e)

- (1) produce e;
- (2) wait(mutex);
- (3) l := append(l,e);
- (4) signal(notEmpty);
- (5) signal(mutex);

Consumer (consumes e)

- (1) wait(notEmpty);
- (2) wait(mutex);
- (3) e := head(l);
- (4) l := tail(l);
- (5) signal(mutex);
- (6) consume e;

List empty at the beginning $\implies$ Invariant: $\text{notEmpty}.V = \text{length}(l)$

Producer/Consumer with Bounded Buffer

Requirements:

- Reading/writing to the buffer **safely** (atomic)
- Consumer needs protection in case the buffer is empty
- Producer needs protection in case the buffer is full

Producer/Consumer with Bounded Buffer (2)

Initial: notEmpty: General semaphore, initialized with 0
 notFull: General semaphore, initialized with N
 mutex: Binary semaphore, initialized with 1
 l: List

Producer (produces e)

- (1) produce e;
- (2) wait(notFull);
- (3) wait(mutex);
- (4) l := append(l,e);
- (5) signal(notEmpty);
- (6) signal(mutex);

Consumer (consumes e)

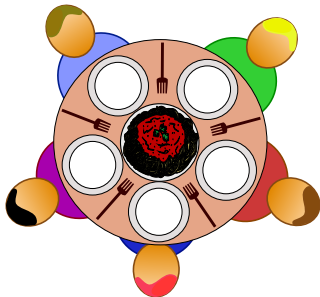
- (1) wait(notEmpty);
- (2) wait(mutex);
- (3) e := head(l);
- (4) l := tail(l);
- (5) signal(notFull);
- (6) signal(mutex);
- (7) consume e;

Invariant: $\text{notEmpty}.V + \text{notFull}.V = N$
“(notEmpty,notFull) = Split Semaphore”

Dining Philosophers

Situation

- Philosophers think or eat spaghetti, alternately.
- Philosophers need both forks to eat.
- Philosophers take forks **one after the other**.



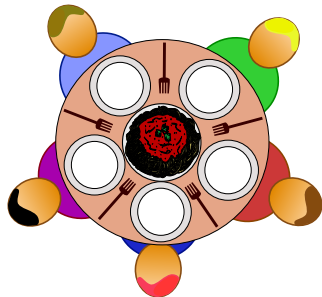
Dining Philosophers

Situation

- Philosophers think or eat spaghetti, alternately.
- Philosophers need both forks to eat.
- Philosophers take forks **one after the other**.

Requirements:

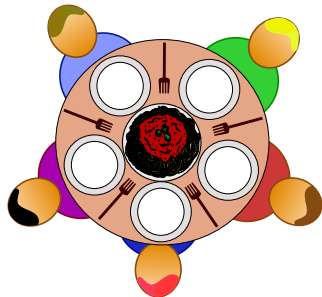
- No deadlock: **some** philosopher can always eat after a finite amount of time
- No starvation: **every** philosopher eats after a finite amount of time



Dining Philosophers

Situation

- Philosophers think or eat spaghetti, alternately.
- Philosophers need both forks to eat.
- Philosophers take forks **one after the other**.



Requirements:

- No deadlock: **some** philosopher can always eat after a finite amount of time
- No starvation: **every** philosopher eats after a finite amount of time

Modeling:

- Philosophers numbered $i \in \{1, \dots, N\}$
- Fork = binary semaphore
- Left fork: $\text{fork}[i]$, right fork: $\text{fork}[i+1]$ (modulo N)

Initially, all forks are initialized with 1

Philosopher i

loop forever

(1) Philosopher thinks;

(2) wait(fork[i]); // left fork

(3) wait(fork[$i+1$]); // right fork

(4) Philosopher eats

(5) signal(fork[$i+1$]);

(6) signal(fork[i]);

end loop

Example

Philosopher 1	Philosopher 2	Philosopher 3
---------------	---------------	---------------

Example

Philosopher 1	Philosopher 2	Philosopher 3
<code>wait(forks[1])</code>		

Example

Philosopher 1	Philosopher 2	Philosopher 3
<code>wait(forks[1])</code> "has left fork"	<code>wait(forks[2])</code>	

Example

Philosopher 1	Philosopher 2	Philosopher 3
<code>wait(forks[1])</code> "has left fork"	<code>wait(forks[2])</code> "has left fork"	<code>wait(forks[3])</code>

Example

Philosopher 1	Philosopher 2	Philosopher 3
<code>wait(forks[1])</code> "has left fork"	<code>wait(forks[2])</code> "has left fork"	<code>wait(forks[3])</code> "has left fork"

Example

Philosopher 1	Philosopher 2	Philosopher 3
<code>wait(forks[1])</code> "has left fork"	<code>wait(forks[2])</code> "has left fork"	<code>wait(forks[3])</code> "has left fork"
<code>wait(forks[2])</code>		

Example

Philosopher 1	Philosopher 2	Philosopher 3
<code>wait(forks[1])</code> "has left fork"	<code>wait(forks[2])</code> "has left fork"	<code>wait(forks[3])</code> "has left fork"
<code>wait(forks[2])</code> blocked	<code>wait(forks[3])</code>	

Example

Philosopher 1	Philosopher 2	Philosopher 3
<code>wait(forks[1])</code> "has left fork"	<code>wait(forks[2])</code> "has left fork"	<code>wait(forks[3])</code> "has left fork"
<code>wait(forks[2])</code> blocked	<code>wait(forks[3])</code> blocked	<code>wait(forks[1])</code>

Example

Philosopher 1	Philosopher 2	Philosopher 3
<code>wait(forks[1])</code> "has left fork"	<code>wait(forks[2])</code> "has left fork"	<code>wait(forks[3])</code> "has left fork"
<code>wait(forks[2])</code> blocked	<code>wait(forks[3])</code> blocked	<code>wait(forks[1])</code> blocked

Deadlock possible: Everyone has the left fork, no one has the right one!

Philosophers: Attempt 2

Initially, all forks are initialized with 1, **mutex**: Binary semaphore, initialized with 1

Philosopher i

loop forever

(1) Philosopher thinks;

(2) **wait(mutex);**

(3) **wait(fork[i]);** // left fork

(4) **wait(fork[i+1]);** // right fork

(5) Philosopher eats

(6) **signal(fork[i+1]);**

(7) **signal(fork[i]);**

(8) **signal(mutex);**

end loop

Philosophers: Attempt 2

Initially, all forks are initialized with 1, **mutex**: Binary semaphore, initialized with 1

```
Philosopher i
  loop forever
    (1) Philosopher thinks;
    (2) wait(mutex);
    (3) wait(fork[i]);    // left fork
    (4) wait(fork[i+1]); // right fork
    (5) Philosopher eats
    (6) signal(fork[i+1]);
    (7) signal(fork[i]);
    (8) signal(mutex);
  end loop
```

Deadlock-free, but starvation possible: One philosopher is repeatedly "overtaken." Also bad: Only one philosopher eats at a time

Philosophers: Attempt 3

Initially, all forks are initialized with 1, **room**: general semaphore, initialized with $N - 1$

Philosopher i

loop forever

(1) Philosopher thinks;

(2) **wait(room);**

(3) **wait(fork[i]);** // left fork

(4) **wait(fork[i+1]);** // right fork

(5) Philosopher eats

(6) **signal(fork[i+1]);**

(7) **signal(fork[i]);**

(8) **signal(room);**

end loop

room always allows a maximum of $N - 1$ philosophers to use the forks at the same time

Philosophers: Attempt 3

Initially, all forks are initialized with 1, **room**: general semaphore, initialized with $N - 1$

Philosopher i

loop forever

(1) Philosopher thinks;

(2) **wait(room);**

(3) **wait(fork[i]);** // left fork

(4) **wait(fork[i+1]);** // right fork

(5) Philosopher eats

(6) **signal(fork[i+1]);**

(7) **signal(fork[i]);**

(8) **signal(room);**

end loop

room always allows a maximum of $N - 1$ philosophers to use the forks at the same time

Deadlock and starvation-free

Philosophers: Attempt 4

Initially, all forks are initialized with 1

Philosopher i , $i < N$

loop forever

- (1) Philosopher thinks;
 - (2) wait(fork[i]); // left fork
 - (3) wait(fork[$i+1$]); // right fork
 - (4) Philosopher eats
 - (5) signal(fork[$i+1$]);
 - (6) signal(fork[i]);
- end loop

Philosopher N

loop forever

- (1) Philosopher thinks;
 - (2) wait(fork[$i+1$]); // right fork
 - (3) wait(fork[i]); // left fork
 - (4) Philosopher eats
 - (5) signal(fork[i]);
 - (6) signal(fork[$i+1$]);
- end loop

Philosophers: Attempt 4

Initially, all forks are initialized with 1

Philosopher i , $i < N$

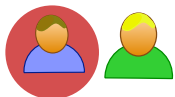
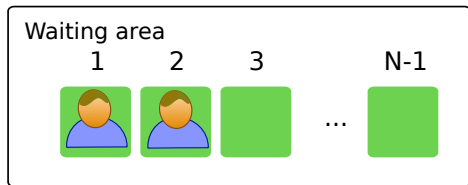
```
loop forever
(1) Philosopher thinks;
(2) wait(fork[i]);    // left fork
(3) wait(fork[i+1]); // right fork
(4) Philosopher eats
(5) signal(fork[i+1]);
(6) signal(fork[i]);
end loop
```

Philosopher N

```
loop forever
(1) Philosopher thinks;
(2) wait(fork[i+1]); // right fork
(3) wait(fork[i]);   // left fork
(4) Philosopher eats
(5) signal(fork[i]);
(6) signal(fork[i+1]);
end loop
```

Deadlock and starvation-free

The Sleeping Barber



Barber Barber
Chair

Situation:

- Barber with $N - 1$ waiting places and one barber chair
- If a customer arrives, he gets a haircut
- If there are no customers, the barber sleeps; the next customer wakes him up
- If the barber chair is occupied, the customer sits in a waiting place
- If all seats are occupied, the customer leaves immediately.

Solution to the Sleeping Barber Problem

Initial: waiting: atomic register, initially 0
 customers: general semaphore, initially 0
 mutex: binary semaphore, initially 1
 synch,barber: binary semaphore, initially 0

Barber

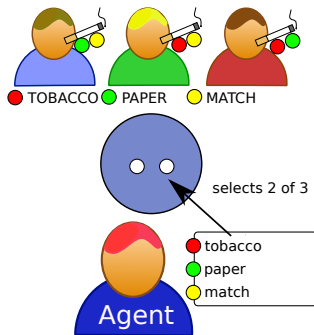
```
loop forever
// sleep as long as there are no customers:
(1)  wait(customers);
(2)  wait(mutex);
(3)  waiting := waiting -1;
// take next customer:
(4)  signal(barber);
(5)  signal(mutex);
(6)  cut hair;
// wait until customer leaves shop:
(7)  wait(synch);
end loop
```

Customer

```
(1)  wait(mutex);
(2)  if waiting <  $N$  then
(3)    waiting := waiting + 1;
// Wake up barber (or increase customers):
(4)    signal(customers);
(5)    signal(mutex);
// Wait until barber is ready:
(6)    wait(barber);
(7)    get a haircut;
// Leave shop:
(8)    signal(synch);
// leave immediately
(9)  else signal(mutex);
```

Cigarette Smoker's Problem

Problem description:



- 4 people: 3 smokers, 1 agent
- To smoke a cigarette, you need: tobacco, paper, matches
- The smokers each have only one of the ingredients (an infinite amount of it)
- The agent has all three
- The agent places 2 of the 3 ingredients on the table
- The smoker who has the third ingredient takes the other two and smokes
- Problem: Synchronize smokers and agents

Cigarette Smoker's Problem (2)

Modeling using 4 binary semaphores for the agent

- $S[1]$ (tobacco), $S[2]$ (paper), $S[3]$ (match):
indicates whether ingredient is on the table (initial 0)
- agent indicates whether the agent places ingredients or must wait (initial 1)

- Agent program already given:

```
loop forever
  (1)  select i and j randomly from {1, 2, 3};
  (2)  wait(agent);
  (3)  signal( $S[i]$ );
  (4)  signal( $S[j]$ );
end loop
```

Cigarette Smoker's Problem: Attempt 1

Smoker with tobacco

```
loop forever
(1)  wait(S[2]);
(2)  wait(S[3]);
(3)  "smoke";
(4)  signal(agent);
end loop
```

Smoker with paper

```
loop forever
(1)  wait(S[1]);
(2)  wait(S[3]);
(3)  "smoke";
(4)  signal(agent);
end loop
```

Smoker with match

```
loop forever
(1)  wait(S[1]);
(2)  wait(S[2]);
(3)  "smoke";
(4)  signal(agent);
end loop
```

S[1].V S[2].V S[3].V
0 0 0

Cigarette Smoker's Problem: Attempt 1

Smoker with tobacco

```
loop forever
(1)  wait(S[2]);
(2)  wait(S[3]);
(3)  "smoke";
(4)  signal(agent);
end loop
```

Smoker with paper

```
loop forever
(1)  wait(S[1]);
(2)  wait(S[3]);
(3)  "smoke";
(4)  signal(agent);
end loop
```

Smoker with match

```
loop forever
(1)  wait(S[1]);
(2)  wait(S[2]);
(3)  "smoke";
(4)  signal(agent);
end loop
```

Agent: Tobacco and Paper (signal(S[1]) + signal(S[2]))

S[1].V	S[2].V	S[3].V
1	1	0

Cigarette Smoker's Problem: Attempt 1

Smoker with tobacco

```
loop forever
(1) wait(S[2]);
(2) wait(S[3]);
(3) "smoke";
(4) signal(agent);
end loop
```

Smoker with paper

```
loop forever
(1) wait(S[1]);
(2) wait(S[3]);
(3) "smoke";
(4) signal(agent);
end loop
```

Smoker with match

```
loop forever
(1) wait(S[1]);
(2) wait(S[2]);
(3) "smoke";
(4) signal(agent);
end loop
```

Agent: Tobacco and Paper (signal(S[1]) + signal(S[2]))

S[1].V	S[2].V	S[3].V
0	1	0

Cigarette Smoker's Problem: Attempt 1

Smoker with tobacco

```
loop forever
(1)  wait(S[2]);
(2)  wait(S[3]);
(3)  "smoke";
(4)  signal(agent);
end loop
```

Smoker with paper

```
loop forever
(1)  wait(S[1]);
(2)  wait(S[3]);
(3)  "smoke";
(4)  signal(agent);
end loop
```

Smoker with match

```
loop forever
(1)  wait(S[1]);
(2)  wait(S[2]);
(3)  "smoke";
(4)  signal(agent);
end loop
```

Agent: Tobacco and Paper (signal(S[1]) + signal(S[2]))

S[1].V S[2].V S[3].V
0 0 0

Cigarette Smoker's Problem: Attempt 1

Smoker with tobacco

```
loop forever
(1)  wait(S[2]);
(2)  wait(S[3]);
(3)  "smoke";
(4)  signal(agent);
end loop
```

Smoker with paper

```
loop forever
(1)  wait(S[1]);
(2)  wait(S[3]);
(3)  "smoke";
(4)  signal(agent);
end loop
```

Smoker with match

```
loop forever
(1)  wait(S[1]);
(2)  wait(S[2]);
(3)  "smoke";
(4)  signal(agent);
end loop
```

Agent: Tobacco and Paper (signal(S[1]) + signal(S[2]))

S[1].V S[2].V S[3].V
0 0 0

DEADLOCK!

Cigarette Smoker's Problem: Attempt 2

NEW: $R[i]$, $i = 1, \dots, 6$, bin. Sem. (initial 0),
 mutex: bin. Sem. (initial 1),
 t: atom. Register (initial 0)

Helper (Tobacco)

```
loop forever
(1)  wait(S[1]);
(2)  wait(mutex);
(3)  t := t+1;
(4)  if t ≠ 1 then
(5)    signal(R[t]);
(6)  signal(mutex);
end loop
```

Smoker with tobacco

```
loop forever
(1)  wait(R[6]);
(2)  t := 0;
(3)  "smoke";
(4)  signal(agent);
end loop
```

Helper (Paper)

```
loop forever
(1)  wait(S[2]);
(2)  wait(mutex);
(3)  t := t+2;
(4)  if t ≠ 2 then
(5)    signal(R[t]);
(6)  signal(mutex);
end loop
```

Smoker with paper

```
loop forever
(1)  wait(R[5]);
(2)  t := 0;
(3)  "smoke";
(4)  signal(agent);
end loop
```

Helper (Match)

```
loop forever
(1)  wait(S[3]);
(2)  wait(mutex);
(3)  t := t+4;
(4)  if t ≠ 4 then
(5)    signal(R[t]);
(6)  signal(mutex);
end loop
```

Smoker with match

```
loop forever
(1)  wait(R[3]);
(2)  t := 0;
(3)  "smoke";
(4)  signal(agent);
end loop
```

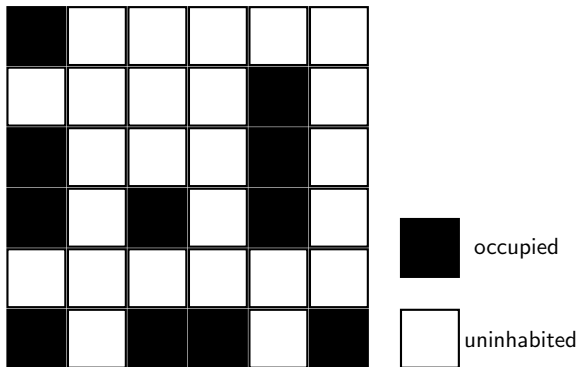
The Right Smoker is Awakened ...

Ingredients on the table	first Helper	second (=awakening) Helper	Value by t	awakened Smoker
Tobacco & Paper	(Tobacco)	(Paper)	$1+2 = 3$	R[3] (=Matches)
Tobacco & Paper	(Paper)	(Tobacco)	$2+1 = 3$	R[3] (=Matches)
Tobacco & Matches	(Tobacco)	(Matches)	$1+4 = 5$	R[5] (=Paper)
Tobacco & Matches	(Matches)	(Tobacco)	$4+1 = 5$	R[5] (=Paper)
Paper & Matches	(Paper)	(Matches)	$2+4 = 6$	R[6] (=Tobacco)
Paper & Matches	(Matches)	(Paper)	$4+2 = 6$	R[6] (=Tobacco)

- some algorithms run in “phases”
- processes perform computations, but at one step, all processes wait for each other.
- only if all have arrived at this point, the processes continue computing.
- The situation was similar in the merge sort example, but there, only one process waits for two others.

Application Example: Game of Life

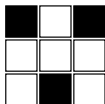
Playing field of Conway's Game of Life: $N \times N$ matrix



Goal: Always calculate the next generation from the current one.

Rules of the Game

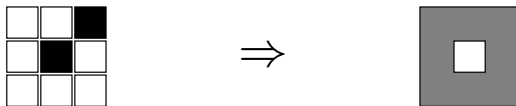
- Field uninhabited: Reinhabited exactly if $\#(\text{neighbors}) = 3$, e.g.



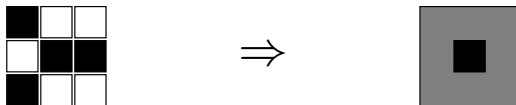
Rules of the Game

– Field occupied:

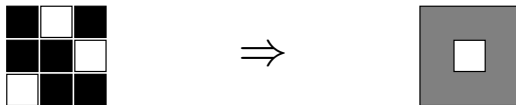
– If $\#(\text{neighbors}) < 2$, then unoccupied (underpopulation), e.g.



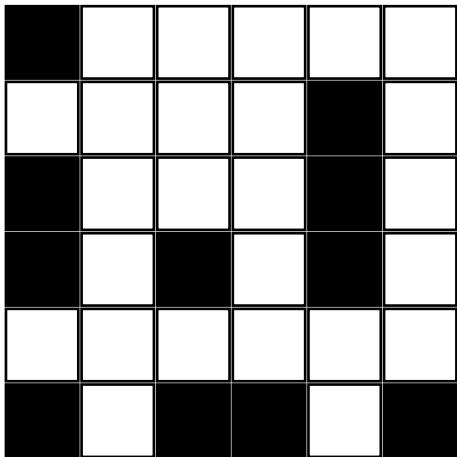
– If $\#(\text{neighbors}) \in \{2, 3\}$, then still inhabited. E.g.



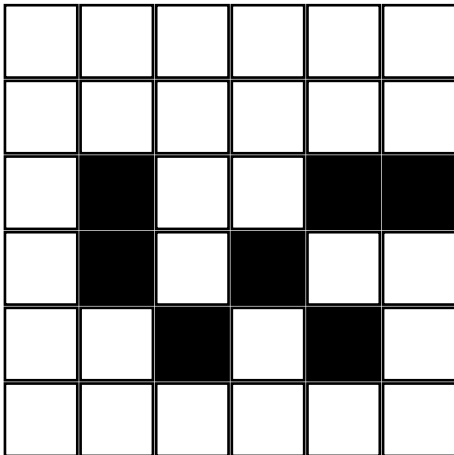
– If $\#(\text{neighbors}) > 3$, then uninhabited afterwards (overpopulation), e.g.



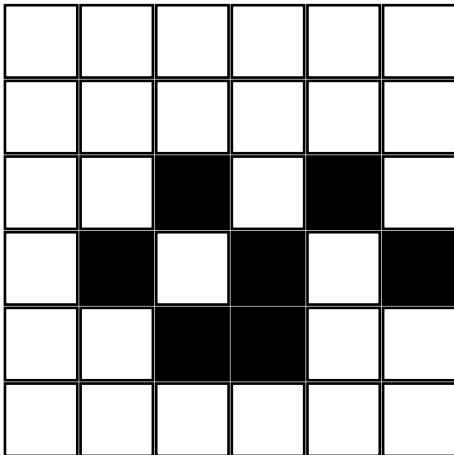
Example



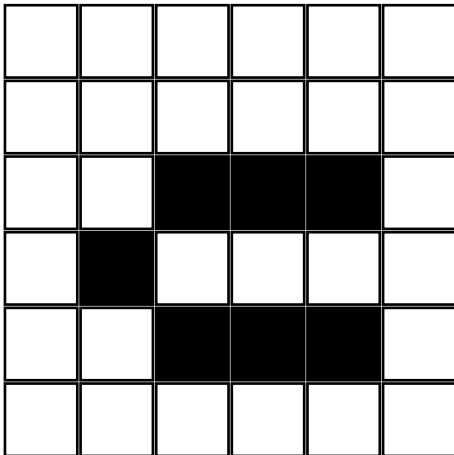
Example



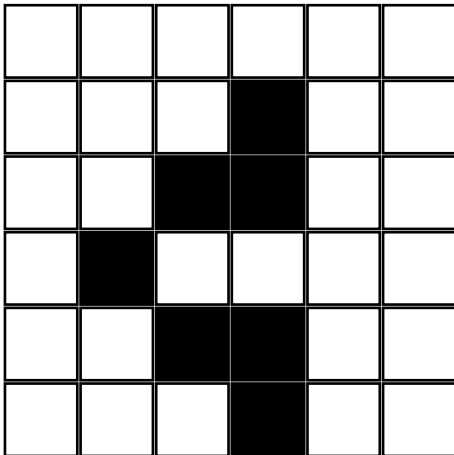
Example



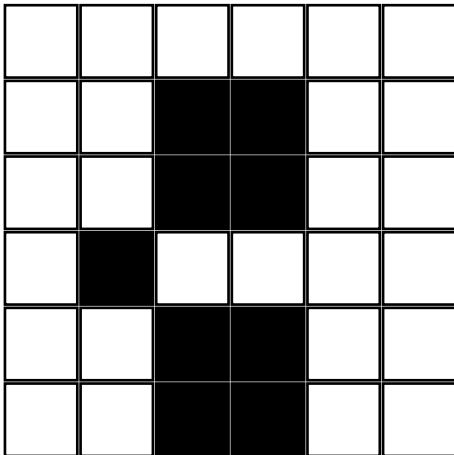
Example



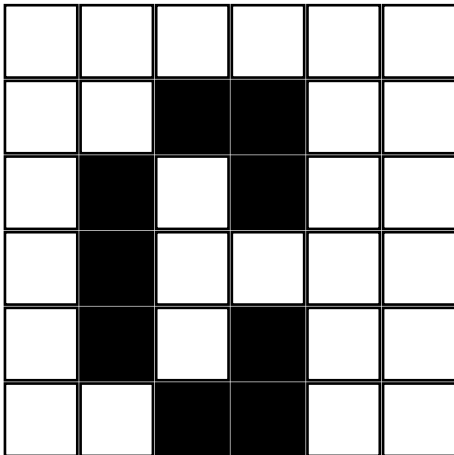
Example



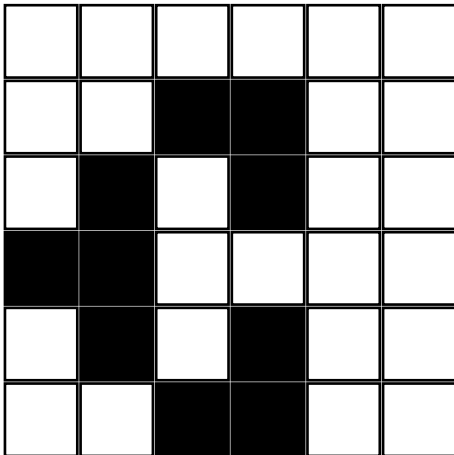
Example



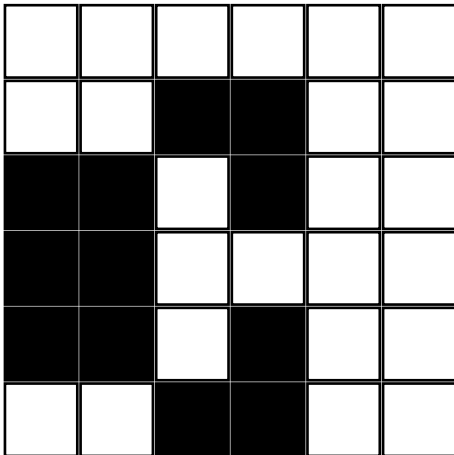
Example



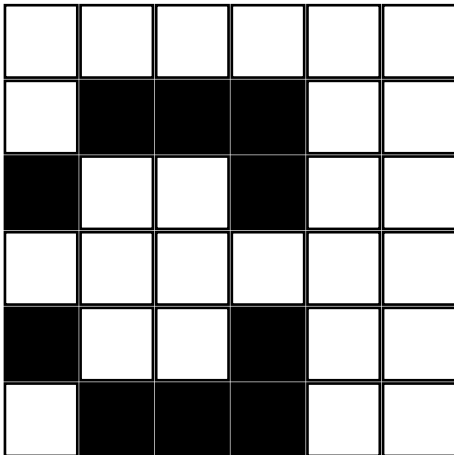
Example



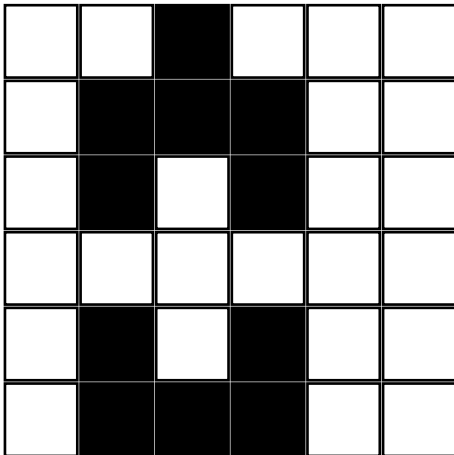
Example



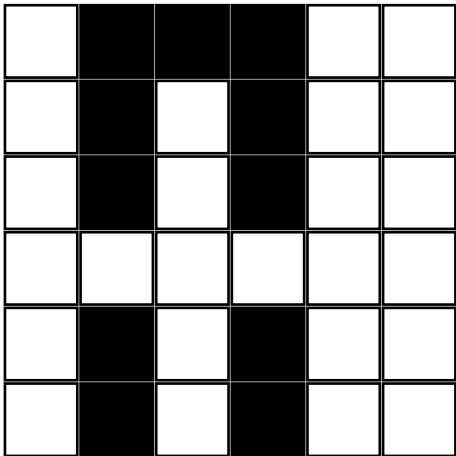
Example



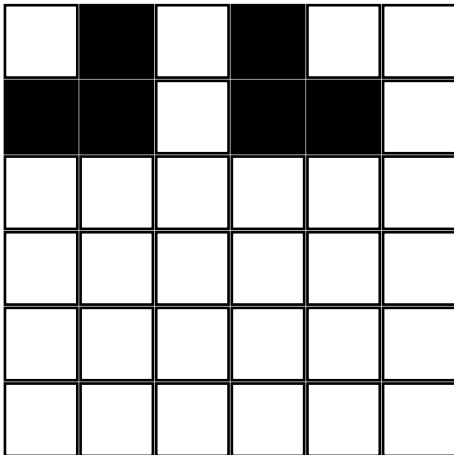
Example



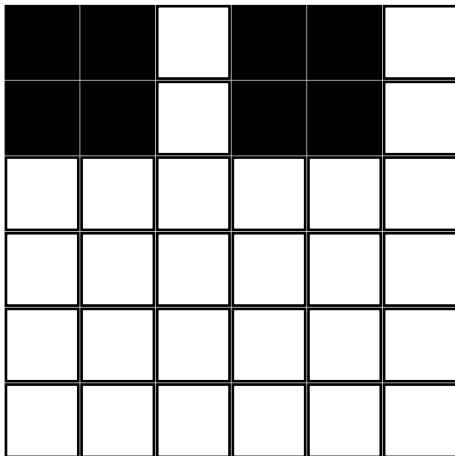
Example



Example



Example



Implementation: Sequential

Assumptions:

- Playing field in $N \times N$ array with Boolean entries
- `nextValue( $i, j, field$ )`: calculate the next value for entry (i, j) : Read all neighbors and (i, j) -entry, then decide True or False

Calculate k th generation:

Calculate k th generation

field: Initialized $N \times N$ array representing the playing field

field2: $N \times N$ array for temporary storage

Algorithm:

```
for g:=1 to k do
  for i=1 to N do
    for j=1 to N do field2[i,j] := nextValue(i,j,field);
  for i=1 to N do
    for j=1 to N do field[i,j] := field2[i,j];
```

- One process per field (i,j) , calculates the entry for the field

Parallel Algorithm

field: Initialized $N \times N$ array representing the playing field

$(N \times N)$ processes: one per playing field

Program for process (i,j) :

```
for g:=1 to k  
  v := nextValue(i,j,field);  
  field[i,j]:=v;
```

Does not work!

Parallel Algorithm

field: Initialized $N \times N$ array representing the playing field

$(N \times N)$ processes: one per playing field

Program for process (i,j) :

for $g:=1$ to k

$v := \text{nextValue}(i,j,\text{field});$

 Wait until all processes have calculated v

$\text{field}[i,j] := v;$

 Wait until all processes have written their update

Parallel Algorithm

field: Initialized $N \times N$ array representing the playing field

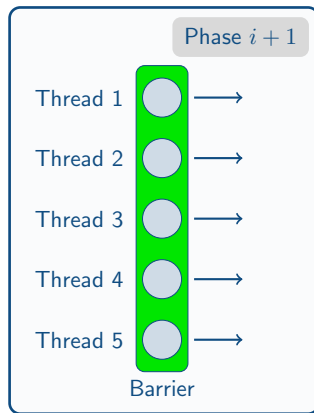
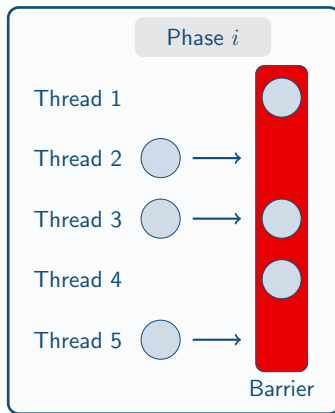
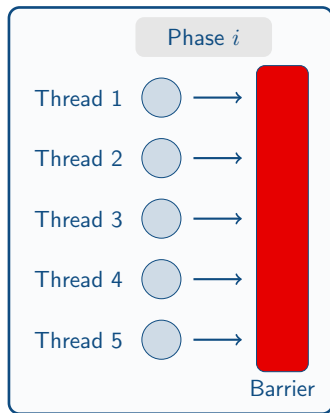
barrier: Barrier for $N \times N$ processes

$(N \times N)$ processes: one per playing field

Program for process (i,j) :

```
for g:=1 to k
  v := nextValue(i,j,field);
  synchBarrier(barrier);
  field[i,j] := v;
  synchBarrier(barrier);
```

General Schema



Barrier for 2 Processes

Initial: p1ready, p2ready: binary semaphores at the beginning 0

Program for process 1:

- (1) Calculation before the barrier
- (2) `signal(p1ready);`
- (3) `wait(p2ready);`
- (4) Calculation after the barrier

Program for process 2:

- (1) Calculation before the barrier
- (2) `signal(p2ready);`
- (3) `wait(p1ready);`
- (4) Calculation after the barrier

p1ready.V	p1ready.M		p2ready.V	p2ready.M
0	∅		0	∅

Barrier for 2 Processes

Initial: p1ready, p2ready: binary semaphores at the beginning 0

Program for process 1:

- (1) Calculation before the barrier
- (2) signal(p1ready);
- (3) wait(p2ready);
- (4) Calculation after the barrier

Program for process 2:

- (1) Calculation before the barrier
- (2) signal(p2ready);
- (3) wait(p1ready);
- (4) Calculation after the barrier

p1ready.V	p1ready.M		p2ready.V	p2ready.M
0	∅		0	∅

Barrier for 2 Processes

Initial: p1ready, p2ready: binary semaphores at the beginning 0

Program for process 1:

- (1) Calculation before the barrier
- (2) signal(p1ready);
- (3) wait(p2ready);
- (4) Calculation after the barrier

Program for process 2:

- (1) Calculation before the barrier
- (2) signal(p2ready);
- (3) wait(p1ready);
- (4) Calculation after the barrier

p1ready.V	p1ready.M		p2ready.V	p2ready.M
0	∅		0	∅

Barrier for 2 Processes

Initial: p1ready, p2ready: binary semaphores at the beginning 0

Program for process 1:

- (1) Calculation before the barrier
- (2) `signal(p1ready);`
- (3) `wait(p2ready);`
- (4) Calculation after the barrier

Program for process 2:

- (1) Calculation before the barrier
- (2) `signal(p2ready);`
- (3) `wait(p1ready);`
- (4) Calculation after the barrier

p1ready.V	p1ready.M		p2ready.V	p2ready.M
1	∅		0	∅

Barrier for 2 Processes

Initial: p1ready, p2ready: binary semaphores at the beginning 0

Program for process 1:

- (1) Calculation before the barrier
- (2) `signal(p1ready);`
- (3) `wait(p2ready);`
- (4) Calculation after the barrier

Program for process 2:

- (1) Calculation before the barrier
- (2) `signal(p2ready);`
- (3) `wait(p1ready);`
- (4) Calculation after the barrier

p1ready.V	p1ready.M		p2ready.V	p2ready.M
1	∅		0	{P1}

Barrier for 2 Processes

Initial: p1ready, p2ready: binary semaphores at the beginning 0

Program for process 1:

- (1) Calculation before the barrier
- (2) `signal(p1ready);`
- (3) `wait(p2ready);`
- (4) Calculation after the barrier

Program for process 2:

- (1) Calculation before the barrier
- (2) `signal(p2ready);`
- (3) `wait(p1ready);`
- (4) Calculation after the barrier

p1ready.V	p1ready.M		p2ready.V	p2ready.M
1	∅		0	∅

Barrier for 2 Processes

Initial: p1ready, p2ready: binary semaphores at the beginning 0

Program for process 1:

- (1) Calculation before the barrier
- (2) `signal(p1ready);`
- (3) `wait(p2ready);`
- (4) Calculation after the barrier

Program for process 2:

- (1) Calculation before the barrier
- (2) `signal(p2ready);`
- (3) `wait(p1ready);`
- (4) Calculation after the barrier

p1ready.V	p1ready.M		p2ready.V	p2ready.M
0	∅		0	∅

Barrier for 2 Processes

Initial: p1ready, p2ready: binary semaphores at the beginning 0

Program for process 1:

- (1) Calculation before the barrier
- (2) signal(p1ready);
- (3) wait(p2ready);
- (4) Calculation after the barrier

Program for process 2:

- (1) Calculation before the barrier
- (2) signal(p2ready);
- (3) wait(p1ready);
- (4) Calculation after the barrier

p1ready.V	p1ready.M		p2ready.V	p2ready.M
0	∅		0	∅

Barrier for 2 Processes

Initial: p1ready, p2ready: binary semaphores at the beginning 0

Program for process 1:

- (1) Calculation before the barrier
- (2) `signal(p1ready);`
- (3) `wait(p2ready);`
- (4) Calculation after the barrier

Program for process 2:

- (1) Calculation before the barrier
- (2) `signal(p2ready);`
- (3) `wait(p1ready);`
- (4) Calculation after the barrier

p1ready.V	p1ready.M		p2ready.V	p2ready.M
0	∅		0	∅

Barrier for n Processes

Initial: arrive, bin. Semaphore initialized with 1, leave, bin. Semaphore initialized with 0
counter: atomic register initialized with 0

Program for process i :

- (1) Calculation before the barrier;
- (2) wait(arrive);
- (3) counter := counter + 1;
- (4) if counter < n // Special task for last process
- (5) then signal(arrive);
- (6) else signal(leave);
- (7) wait(leave);
- (8) counter := counter - 1;
- (9) if counter > 0 // Special task for last process
- (10) then signal(leave);
- (11) else signal(arrive);
- (12) Calculation after the barrier;

Barriers as an Abstract Data Type

- We also use barriers as ADTs.
- Internal representation, e.g., as a 4-tuple $(n, arrive, leave, counter)$

Operations:

- `newBarrier( $k$ )`: Creates a barrier for k processes.
(internally: creates two semaphores for arrive and leave and an atomic register counter,
return value is 4-tuple $(n, arrive, leave, counter)$)
- `synchBarrier( $B$ )`: Synchronize at the barrier, i.e., lines (2) to (12) are executed for the 4-tuple B .

Grouping of processes into

- **Readers:** Processes that access a shared resource by **reading**
- **Writers:** Processes that access the shared resource by **writing**

Example: Flight booking system

- Some accesses are read-only: which flights are available, when, etc.
- Other accesses book flights, thereby changing the available flights

Allowed / Not allowed

- Multiple read processes simultaneously, but
- Only one process writes at a time

Problem:

- Resolve access so that many read simultaneously, but never write simultaneously.

Various solutions:

- Priority for readers
- Priority for writers

Priority for Readers

Initial: countR: atomic register, initially 0
mutex, mutexR, w: bin. Sem. initially 1

Program for Reader

```
(1)  wait(mutexR);  
(2)  countR := countR + 1;  
(3)  if countR = 1 then  
(4)    wait(w);  
(5)  signal(mutexR);  
(6)  critical section  
(7)  wait(mutexR);  
(8)  countR := countR - 1;  
(9)  if countR = 0 then  
(10)    signal(w);  
(11) signal(mutexR);
```

Program for Writer

```
(1)  wait(mutex);  
(2)  wait(w);  
(3)  signal(mutex);  
(4)  critical section;  
(5)  signal(w)
```

Priority for Readers

Initial: countR: atomic register, initially 0
mutex, mutexR, w: bin. Sem. initially 1

Program for Reader

```
(1)  wait(mutexR);
(2)  countR := countR + 1;
(3)  if countR = 1 then
(4)    wait(w);
(5)  signal(mutexR);
(6)  critical section
(7)  wait(mutexR);
(8)  countR := countR - 1;
(9)  if countR = 0 then
(10)    signal(w);
(11) signal(mutexR);
```

Program for Writer

```
(1)  wait(mutex);
(2)  wait(w);
(3)  signal(mutex);
(4)  critical section;
(5)  signal(w)
```

- mutexR protects access to countR

Priority for Readers

Initial: countR: atomic register, initially 0
mutex, mutexR, w: bin. Sem. initially 1

Program for Reader

```
(1) wait(mutexR);  
(2) countR := countR + 1;  
(3) if countR = 1 then  
(4)     wait(w);  
(5) signal(mutexR);  
(6) critical section  
(7) wait(mutexR);  
(8) countR := countR - 1;  
(9) if countR = 0 then  
(10)     signal(w);  
(11) signal(mutexR);
```

Program for Writer

```
(1) wait(mutex);  
(2) wait(w);  
(3) signal(mutex);  
(4) critical section;  
(5) signal(w)
```

- mutexR protects access to countR
- first reader blocks writer, or waits for writer to finish

Priority for Readers

Initial: countR: atomic register, initially 0
mutex, mutexR, w: bin. Sem. initially 1

Program for Reader

```
(1)  wait(mutexR);
(2)  countR := countR + 1;
(3)  if countR = 1 then
(4)    wait(w);
(5)  signal(mutexR);
(6)  critical section
(7)  wait(mutexR);
(8)  countR := countR - 1;
(9)  if countR = 0 then
(10)    signal(w);
(11) signal(mutexR);
```

Program for Writer

```
(1)  wait(mutex);
(2)  wait(w);
(3)  signal(mutex);
(4)  critical section;
(5)  signal(w)
```

- mutexR protects access to countR
- first reader blocks writer, or waits for writer to finish
- last reader unblocks writer

Priority for Readers

Initial: countR: atomic register, initially 0
mutex, mutexR, w: bin. Sem. initially 1

Program for Reader

```
(1) wait(mutexR);  
(2) countR := countR + 1;  
(3) if countR = 1 then  
(4)     wait(w);  
(5) signal(mutexR);  
(6) critical section  
(7) wait(mutexR);  
(8) countR := countR - 1;  
(9) if countR = 0 then  
(10)     signal(w);  
(11) signal(mutexR);
```

Program for Writer

```
(1) wait(mutex);  
(2) wait(w);  
(3) signal(mutex);  
(4) critical section;  
(5) signal(w)
```

- mutexR protects access to countR
- first reader blocks writer, or waits for writer to finish
- last reader unblocks writer
- **mutex ensures that a maximum of 1 writer waits at w**

Priority for Writers

Initial: countR, countW: atomic registers, initially 0
mutexR, mutexW, mutex, w, r: bin. sem. at the beginning 1

Program for Readers

```
(1) wait(mutex);  
(2) wait(r);  
(3) wait(mutexR);  
(4) countR := countR + 1;  
(5) if countR = 1 then wait(w);  
(6) signal(mutexR);  
(7) signal(r);  
(8) signal(mutex);  
(9) critical section;  
(10) wait(mutexR);  
(11) countR := countR - 1;  
(12) if countR = 0 then signal(w);  
(13) signal(mutexR);
```

Program for Writer

```
(1) wait(mutexW);  
(2) countW := countW + 1;  
(3) if countW = 1 then wait(r);  
(4) signal(mutexW);  
(5) wait(w);  
(6) critical section;  
(7) signal(w);  
(8) wait(mutexW);  
(9) countW := countW - 1;  
(10) if countW = 0 then signal(r);  
(11) signal(mutexW);
```

Priority for Writers

Initial: countR, countW: atomic registers, initially 0
mutexR, mutexW, mutex, w, r: bin. sem. at the beginning 1

Program for Readers

```
(1) wait(mutex);
(2) wait(r);
(3) wait(mutexR);
(4) countR := countR + 1;
(5) if countR = 1 then wait(w);
(6) signal(mutexR);
(7) signal(r);
(8) signal(mutex);
(9) critical section;
(10) wait(mutexR);
(11) countR := countR - 1;
(12) if countR = 0 then signal(w);
(13) signal(mutexR);
```

Program for Writer

```
(1) wait(mutexW);
(2) countW := countW + 1;
(3) if countW = 1 then wait(r);
(4) signal(mutexW);
(5) wait(w);
(6) critical section;
(7) signal(w);
(8) wait(mutexW);
(9) countW := countW - 1;
(10) if countW = 0 then signal(r);
(11) signal(mutexW);
```

- mutexR, mutexW protect countR, countW

Priority for Writers

Initial: countR, countW: atomic registers, initially 0
mutexR, mutexW, mutex, w, r: bin. sem. at the beginning 1

Program for Readers

```
(1) wait(mutex);  
(2) wait(r);  
(3) wait(mutexR);  
(4) countR := countR + 1;  
(5) if countR = 1 then wait(w);  
(6) signal(mutexR);  
(7) signal(r);  
(8) signal(mutex);  
(9) critical section;  
(10) wait(mutexR);  
(11) countR := countR - 1;  
(12) if countR = 0 then signal(w);  
(13) signal(mutexR);
```

Program for Writer

```
(1) wait(mutexW);  
(2) countW := countW + 1;  
(3) if countW = 1 then wait(r);  
(4) signal(mutexW);  
(5) wait(w);  
(6) critical section;  
(7) signal(w);  
(8) wait(mutexW);  
(9) countW := countW - 1;  
(10) if countW = 0 then signal(r);  
(11) signal(mutexW);
```

- mutexR, mutexW protect countR, countW
- first reader blocks writer

Priority for Writers

Initial: countR, countW: atomic registers, initially 0
mutexR, mutexW, mutex, w, r: bin. sem. at the beginning 1

Program for Readers

```
(1) wait(mutex);  
(2) wait(r);  
(3) wait(mutexR);  
(4) countR := countR + 1;  
(5) if countR = 1 then wait(w);  
(6) signal(mutexR);  
(7) signal(r);  
(8) signal(mutex);  
(9) critical section;  
(10) wait(mutexR);  
(11) countR := countR - 1;  
(12) if countR = 0 then signal(w);  
(13) signal(mutexR);
```

Program for Writer

```
(1) wait(mutexW);  
(2) countW := countW + 1;  
(3) if countW = 1 then wait(r);  
(4) signal(mutexW);  
(5) wait(w);  
(6) critical section;  
(7) signal(w);  
(8) wait(mutexW);  
(9) countW := countW - 1;  
(10) if countW = 0 then signal(r);  
(11) signal(mutexW);
```

- mutexR, mutexW protect countR, countW
- first reader blocks writer
- first writer blocks reader

Priority for Writers

Initial: countR, countW: atomic registers, initially 0
 mutexR, mutexW, mutex, w, r: bin. sem. at the beginning 1

Program for Readers

```
(1) wait(mutex);
(2) wait(r);
(3) wait(mutexR);
(4) countR := countR + 1;
(5) if countR = 1 then wait(w);
(6) signal(mutexR);
(7) signal(r);
(8) signal(mutex);
(9) critical section;
(10) wait(mutexR);
(11) countR := countR - 1;
(12) if countR = 0 then signal(w);
(13) signal(mutexR);
```

Program for Writer

```
(1) wait(mutexW);
(2) countW := countW + 1;
(3) if countW = 1 then wait(r);
(4) signal(mutexW);
(5) wait(w);
(6) critical section;
(7) signal(w);
(8) wait(mutexW);
(9) countW := countW - 1;
(10) if countW = 0 then signal(r);
(11) signal(mutexW);
```

- mutexR, mutexW protect countR, countW
- first reader blocks writer
- first writer blocks reader
- **mutex** ensures that only one reader can wait at **r**

Monitors



Advantages

- Many problems of concurrent programming can be solved with semaphores
- Semaphores are implemented in many programming languages

Disadvantages

- Explicit setting and removal of locks
- Prone to errors:
 - If the programmer forgets a `signal` call, a deadlock may occur
 - If the programmer forgets a `wait` call, exclusive access may be violated

- Monitors attempt to remedy the weaknesses of semaphores.
- Enable **structured** programming.
- Data and access to data are encapsulated in a monitor.
- Fit the object-oriented model.

Monitors (2)

```
class Account {  
    int Balance;  
    int AccountNumber;  
    int CustomerId  
  
    withdraw(int x) {  
        Balance := Balance - x;  
    }  
  
    deposit(int x) {  
        Balance := Balance + x;  
    }  
}
```

Monitors (2)

```
monitor Account {  
    int Balance;  
    int AccountNumber;  
    int CustomerId  
  
    withdraw(int x) {  
        Balance := Balance - x;  
    }  
  
    deposit(int x) {  
        Balance := Balance + x;  
    }  
}
```

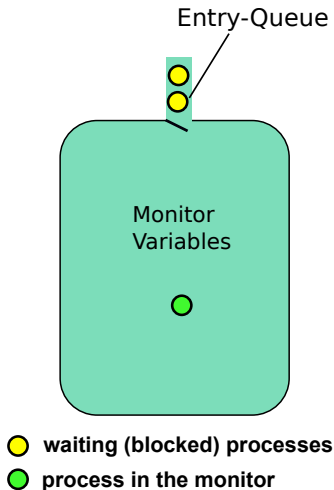
- Encapsulation of data and methods
- **No** direct access to attributes
- Access only via methods

Monitors (2)

```
monitor Account {  
    int Balance;  
    int AccountNumber;  
    int CustomerId  
  
    withdraw(int x) {  
        Balance := Balance - x;  
    }  
  
    deposit(int x) {  
        Balance := Balance + x;  
    }  
}
```

- Encapsulation of data and methods
- **No** direct access to attributes
- Access only via methods
- Only one process can be in the monitor at a time
- This means that only one method from one process can be executed at a time
- Other processes are blocked

Monitors (3)



- Waiting processes either in FIFO queue or
- Set, i.e., no starvation-freedom
- Depends on implementation

Mutual Exclusion with Monitor

```
monitor Mutex {  
    ...  
    Variables of the critical section  
    ...  
    inCriticalSection(...) {  
        Critical section;  
    }  
}
```

Process i:

```
loop forever  
(1) Remaining code  
(2) Mutex.inCriticalSection()  
end loop
```

Advantage over semaphores

Setting/removing the lock happens implicitly

Advantages and Disadvantages

Advantage over semaphores

Setting/removing the lock happens implicitly

Disadvantage over semaphores

Many problems require explicit blocking of processes

Advantage over semaphores

Setting/removing the lock happens implicitly

Disadvantage over semaphores

Many problems require explicit blocking of processes

Examples:

- Bounded buffer: The producer must be blocked if the buffer is full, and the consumer must be blocked if the buffer is empty
- Readers and writers: Multiple reader processes should be allowed, and writer processes must be blocked while reading, etc.

Advantage over semaphores

Setting/removing the lock happens implicitly

Disadvantage over semaphores

Many problems require explicit blocking of processes

Examples:

- Bounded buffer: The producer must be blocked if the buffer is full, and the consumer must be blocked if the buffer is empty
- Readers and writers: Multiple reader processes should be allowed, and writer processes must be blocked while reading, etc.

For readers and writers: Reading processes must not remain in the monitor while reading, otherwise only one process can read at a time!

- Previous monitor definition too weak (compared to semaphores)
- Therefore: extension with **condition variables** or **condition expressions**
- Two different approaches, both of which occur in programming languages
- In the following: first condition variables, then (briefly) condition expressions

- FIFO queue (usually) with operations
- The name of the condition variable is usually chosen to explain the condition that will become true, but
- The condition is **not** checked by the runtime system. It is only **a name**
- Operations for condition variable `cond`:
 - `waitC(cond)`
 - `signalC(cond)`
 - sometimes: `empty(cond)`
- Attention: Semantics differ from `signal` and `wait` in semaphores!

Condition Variables (2)

Simulation of a semaphore with a monitor
monitor Semaphore {

```
    int s := k;
```

```
    condition notZero;
```

```
    wait() {
```

```
        if s = 0 then
```

```
            waitC(notZero)
```

```
            s := s - 1
```

```
    }
```

```
    signal() {
```

```
        s := s + 1
```

```
        signalC(notZero)
```

```
    }
```

```
}
```

- Let P be the calling process, $cond$ a condition variable in the monitor.
- Reminder: $cond$ is a FIFO queue (list).
- Let $lock$ be the implicit lock of the monitor (lock to the entry queue).

- Let P be the calling process, $cond$ a condition variable in the monitor monitor.
- Reminder: $cond$ is a FIFO queue (list).
- Let $lock$ be the implicit lock of the monitor ($lock$ to the entry queue).

Semantics of `waitC`

```
waitC(cond) {  
    cond := append(cond,P); // Process is added to the queue  
    P.state := blocked;    // Process blocked  
    monitor.lock := release; // Monitor lock is released  
}
```

Semantics of Condition Variables (2)

Semantics of signalC

```
signalC(cond) {  
  if cond  $\neq$  empty then  
    Q := head(cond);    // First process in the queue  
    cond := tail(cond); // is removed from the queue  
    Q.state := ready;   // and unblocked  
}
```

Semantics of signalC

```
signalC(cond) {  
  if cond  $\neq$  empty then  
    Q := head(cond);    // First process in the queue  
    cond := tail(cond); // is removed from the queue  
    Q.state := ready;   // and unblocked  
}
```

Attention:

- The semantics are still underspecified:
- Presumably: Calling process and unblocked process **simultaneously** in the monitor
- Clarification later, assumption for now: Calling process leaves monitor

Semantics of Condition Variables (3)

Semantics of empty

```
empty(cond) {  
    return(cond == empty); }
```

Semaphore Simulation – Again

```
monitor Semaphore {  
    int s := k;  
    condition notZero;  
  
    wait() {  
        if s = 0 then  
            waitC(notZero)  
        s := s - 1  
    }  
  
    signal() {  
        s := s + 1  
        signalC(notZero)  
    }  
}
```

Comparison of wait, signal, waitC, signalC

Semaphores Sem	Monitors (Condition Variable cond)
<code>wait(Sem)</code> can lead to blocking, but does not have to	

Comparison of wait, signal, waitC, signalC

Semaphores Sem

`wait(Sem)` can lead to blocking, but does not have to

Monitors (Condition Variable cond)

`waitC(cond)` blocks the process **always**

Comparison of wait, signal, waitC, signalC

Semaphores Sem	Monitors (Condition Variable cond)
<code>wait(Sem)</code> can lead to blocking, but does not have to	<code>waitC(cond)</code> blocks the process always
<code>signal(Sem)</code> always has an effect: Unblocking a process or Increasing <code>Sem.V</code>	

Comparison of wait, signal, waitC, signalC

Semaphores Sem	Monitors (Condition Variable cond)
<code>wait(Sem)</code> can lead to blocking, but does not have to	<code>waitC(cond)</code> blocks the process always
<code>signal(Sem)</code> always has an effect: Unblocking a process or Increasing <code>Sem.V</code>	<code>signalC(cond)</code> can be ineffective: Either the process in <code>cond</code> is unblocked, or ineffective if <code>cond</code> is empty

Producer / Consumer with Monitor

```
monitor BoundedBuffer {  
  bufferType buffer := empty;  
  condition notEmpty;  
  condition notFull;  
  produce(v) {  
    if length(buffer) = N then  
      waitC(notFull);  
    buffer := append(v,buffer);  
    signalC(notEmpty);  
  }  
  consume() {  
    if length(buffer) = 0 then  
      waitC(notEmpty)  
    w := head(buffer);  
    buffer := tail(buffer);  
    signalC(notFull)  
    return(w);  
  }  
}
```

Producer code:

```
loop forever  
(1) produce e  
(2) BoundedBuffer.produce(e)  
end loop
```

Consumer code:

```
loop forever  
(1) e := BoundedBuffer.consume();  
(2) consume e;  
end loop
```

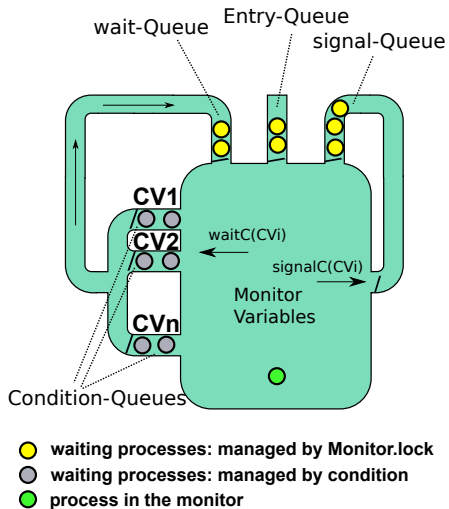
More Precise Modeling of Monitors

- Reminder:
When executing `signalC`, it is not clear who should have the monitor lock afterwards.
- There are different variants
- If the awakened process is allowed to continue computing:
Where does the signaling process wait?

More Precise Modeling of Monitors

- Reminder:
When executing `signalC`, it is not clear who should have the monitor lock afterwards.
- There are different variants
- If the awakened process is allowed to continue computing:
Where does the signaling process wait?
- Therefore, more accurate modeling:
- There are several queues:
 - Condition queue (one per condition variable)
 - wait queue (one per monitor)
 - signal queue (one per monitor)
 - entry queue (one per monitor)

Monitor Modeling with 3 Queues for the Condition Variable



Condition queues: This is where the processes blocked by `waitC` wait.

Wait queue: If a `signalC` is executed, the first waiting process in the condition queue is added to the wait queue.

Signal queue: The process that executes `signalC` is placed in the signal queue **if** it has unblocked a process in the condition queue.

Entry queue: Waiting processes that want to enter the monitor

Condition queues: This is where the processes blocked by `waitC` wait.

Wait queue: If a `signalC` is executed, the first waiting process in the condition queue is added to the wait queue.

Signal queue: The process that executes `signalC` is placed in the signal queue **if** it has unblocked a process in the condition queue.

Entry queue: Waiting processes that want to enter the monitor

Problem

Which of the three queues at the monitor lock has priority over the others?

Problem

Which of the three queues at the monitor lock has priority over the others?

- Depending on the implementation, there are different strategies.
- We write E, W, S for the priorities of the entry, wait, and signal queues.

Priorities for the Queues (2)

13 possibilities

- 1 $E = W = S$
- 2 $E = W < S$
- 3 $E = S < W$
- 4 $E < W = S$
- 5 $E < W < S$
- 6 $E < S < W$
- 7 $E > W = S$
- 8 $E = S > W$
- 9 $S > E > W$
- 10 $E = W > S$
- 11 $W > E > S$
- 12 $E > S > W$
- 13 $E > W > S$

Priorities for the Queues (2)

13 possibilities

- 1 $E = W = S$
- 2 $E = W < S$
- 3 $E = S < W$
- 4 $E < W = S$
- 5 $E < W < S$
- 6 $E < S < W$
- 7 $E > W = S$ not meaningful
- 8 $E = S > W$
- 9 $S > E > W$
- 10 $E = W > S$
- 11 $W > E > S$
- 12 $E > S > W$ not meaningful
- 13 $E > W > S$ not meaningful

- New processes were given priority over old ones.
- W and S processes may have to wait arbitrarily long.
- The W and S queues may become increasingly full.

Priorities for the Queues (2)

13 possibilities

- 1 $E = W = S$
- 2 $E = W < S$
- 3 $E = S < W$
- 4 $E < W = S$
- 5 $E < W < S$
- 6 $E < S < W$
- 7 $E > W = S$ not meaningful
- 8 $E = S > W$ not meaningful
- 9 $S > E > W$ not meaningful
- 10 $E = W > S$ not meaningful
- 11 $W > E > S$ not meaningful
- 12 $E > S > W$ not meaningful
- 13 $E > W > S$ not meaningful

- Similar problem
- New processes would be given priority over old ones
- W or S processes may have to wait for any length of time
- The W or S queues may become increasingly full
- $\Rightarrow E$ should not be larger than W or S

Priorities for the Queues (2)

13 possibilities

- | | | |
|----|-------------|--------------------|
| 1 | $E = W = S$ | |
| 2 | $E = W < S$ | |
| 3 | $E = S < W$ | |
| 4 | $E < W = S$ | |
| 5 | $E < W < S$ | |
| 6 | $E < S < W$ | Classic Definition |
| 7 | $E > W = S$ | not meaningful |
| 8 | $E = S > W$ | not meaningful |
| 9 | $S > E > W$ | not meaningful |
| 10 | $E = W > S$ | not meaningful |
| 11 | $W > E > S$ | not meaningful |
| 12 | $E > S > W$ | not meaningful |
| 13 | $E > W > S$ | not meaningful |

- First, the waiting process may continue computing
- Then the signaling process
- Finally, new processes
- Designations:

Signal and Urgent Wait
(Hoare 1974)

or also

Immediate Resumption
Requirement

Priorities for the Queues (2)

13 possibilities

- 1 $E = W = S$
- 2 $E = W < S$
- 3 $E = S < W$
- 4 $E < W = S$
- 5 $E < W < S$ Signal and Continue
- 6 $E < S < W$ Classic Definition
- 7 $E > W = S$ not meaningful
- 8 $E = S > W$ not meaningful
- 9 $S > E > W$ not meaningful
- 10 $E = W > S$ not meaningful
- 11 $W > E > S$ not meaningful
- 12 $E > S > W$ not meaningful
- 13 $E > W > S$ not meaningful

Priorities for the Queues (2)

13 possibilities

- | | | |
|----|-------------|---------------------|
| 1 | $E = W = S$ | |
| 2 | $E = W < S$ | |
| 3 | $E = S < W$ | Signal and Wait |
| 4 | $E < W = S$ | |
| 5 | $E < W < S$ | Signal and Continue |
| 6 | $E < S < W$ | Classic Definition |
| 7 | $E > W = S$ | not meaningful |
| 8 | $E = S > W$ | not meaningful |
| 9 | $S > E > W$ | not meaningful |
| 10 | $E = W > S$ | not meaningful |
| 11 | $W > E > S$ | not meaningful |
| 12 | $E > S > W$ | not meaningful |
| 13 | $E > W > S$ | not meaningful |

Priorities for the Queues (2)

13 possibilities

- | | | |
|----|-------------|---------------------|
| 1 | $E = W = S$ | |
| 2 | $E = W < S$ | Wait and Notify |
| 3 | $E = S < W$ | Signal and Wait |
| 4 | $E < W = S$ | |
| 5 | $E < W < S$ | Signal and Continue |
| 6 | $E < S < W$ | Classic Definition |
| 7 | $E > W = S$ | not meaningful |
| 8 | $E = S > W$ | not meaningful |
| 9 | $S > E > W$ | not meaningful |
| 10 | $E = W > S$ | not meaningful |
| 11 | $W > E > S$ | not meaningful |
| 12 | $E > S > W$ | not meaningful |
| 13 | $E > W > S$ | not meaningful |

- Used in Java
- Problematic: Condition variable condition may be false again until the process is actually active in the monitor again
- Solution: Circular checking and waiting:

Example: Only decrease s if $s \neq 0$

```
while s = 0 do
    waitC(sIsNotZero);
s := s - 1;
```

Monitors with Condition Expressions

- Instead of condition variables:
- Real Boolean expressions may be used with `waitCE`
(we write `waitCE` instead of `waitC`)
- The operation `signalC` does **not** exist
- The runtime system checks whether the expressions become true and then automatically unblocks.

Producers / Consumers with Condition Expressions

```
monitor BoundedBuffer {  
  bufferType buffer := empty;  
  constant length := N;  
  
  produce(v) {  
    waitCE(length(buffer) < N); // Buffer full?  
    buffer := append(v,buffer);  
  }  
  
  consume() {  
    waitCE(length(buffer) > 0); // Buffer empty?  
    w := head(buffer);  
    buffer := tail(buffer);  
    return(w);  }  
}
```

Producer code:

```
loop forever  
  (1) produce e  
  (2) BoundedBuffer.produce(e)  
end loop
```

Consumer code:

```
loop forever  
  (1) e := BoundedBuffer.consume();  
  (2) consume e;  
end loop
```

Application Examples

- Some of the well-known application examples coded with monitors
- We use monitors with condition variables
- With the classic definition $E < S < W$ (Signal and Urgent Wait)



Readers & Writers with Monitors

```
monitor RW {  
  int countR, countW := 0;  
  condition okToWrite, okToRead;  
  
  startRead() {  
    if countW  $\neq$  0 or not(empty(okToWrite))  
    then waitC(okToRead);  
    countR := countR + 1;  
    signalC(okToRead);}   
  
  endRead() {  
    countR := countR - 1;  
    if countR = 0 then signalC(okToWrite);}   
  
  ... }
```

Readers & Writers with Monitors (2)

...

```
startWrite() {  
    if countR  $\neq$  0 or countW  $\neq$  0  
    then waitC(okToWrite);  
    countW := 1;}  
  
endWrite() {  
    countW := 0;  
    if empty(okToRead)  
    then signalC(okToWrite);  
    else signalC(okToRead);}
```

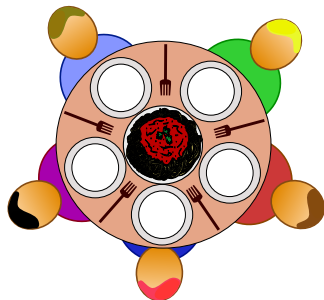
Reader code:

- (1) RW.startRead();
- (2) Read;
- (3) RW.endRead();

Writer code:

- (1) RW.startWrite();
- (2) Write;
- (3) RW.endWrite();

Dining Philosophers with Monitors



- Modeling without semaphores
- Forks are only used implicitly
- For each philosopher, it is noted how many forks are in front of him (0, 1, or 2)
- Philosopher only eats if he has two forks
- Modification of the number of forks:
 - Forks in front of philosopher i are modified by:
left and right neighbors of philosopher i

Dining Philosophers with Monitors (2)

```
monitor Forks {  
    constant numberPhil :=  $N$ ;  
    int array [1,...,numberPhil] forks := [2,...,2];  
    condition array [1,..., numberPhil] okToEat;  
    takeForks(i) {  
        if forks[i]  $\neq$  2 then waitC(okToEat[i]);  
        forks[i+1] := forks[i+1]-1;  
        forks[i-1] := forks[i-1]-1;}  
    releaseForks(i) {  
        forks[i+1] := forks[i+1]+1;  
        forks[i-1] := forks[i-1]+1;  
        if forks[i+1] = 2 then signalC(okToEat[i+1]);  
        if forks[i-1] = 2 then signalC(okToEat[i-1]);}  
}
```

Note: $i+1$, $i-1$ are meant to be taken modulo N

Dining Philosophers with Monitors (3)

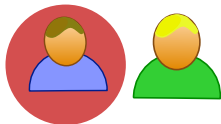
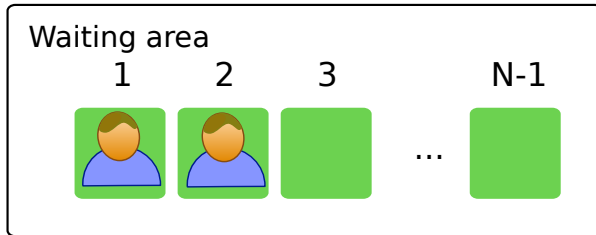
Philosopher i:

```
loop forever
(1)  think;
(2)  Forks.takeForks(i);
(3)  eat;
(4)  Forks.releaseForks(i);
end loop
```

Properties

- Solution is deadlock-free
- but not starvation-free!

Sleeping Barber with Monitors



Barber Barber
Chair

Sleeping Barber with Monitors (2)

```
monitor Barbershop {  
  int waiting:= 0;  
  condition customerPresent;  
  condition barberReady;  
  condition leaveShop;  
  
  // Methods for the hairdresser  
  
  takeCustomer() {  
    if waiting = 0 then  
      waitC(customerPresent);  
    waiting := waiting - 1;  
    signalC(barberReady)}  
  
  endCustomer() {  
    waitC(leaveShop)}
```

Sleeping Barber with Monitors (3)

```
...  
// Methods for the customer  
getHaircut() {  
  if waiting < N then  
    waiting := waiting + 1;  
    signalC(customerPresent)  
    waitC(barberReady)  
    return(True);  
  else return(False);}  
leaveShop(){  
  signalC(leaveShop)}  
}
```

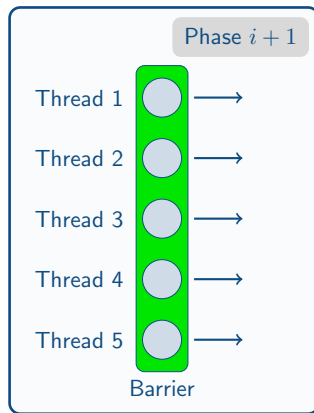
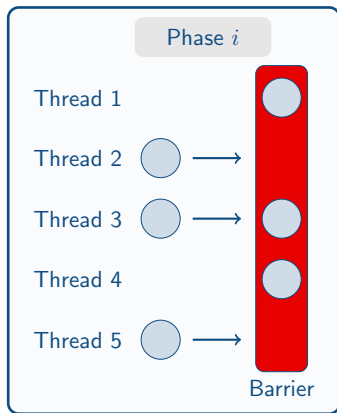
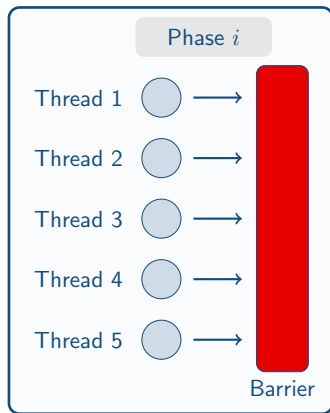
Barber:

```
loop forever  
  Barbershop.takeCustomer();  
  CutHair;  
  Barbershop.endCustomer()
```

Customer

```
if Barbershop.getHaircut()  
then  
  get a haircut;  
  leaveShop();
```

Barrier with Monitor



Barrier with Monitor (2)

```

monitor Barrier {
  int arrived := 0;
  int maxProcesses := N;
  condition allArrived;

  barrier () {
    arrived := arrived + 1;
    if arrived < maxProcesses then
      waitC(allArrived);
    else
      arrived := 0;
      signalAllC(allArrived);
    }
  }
}

```

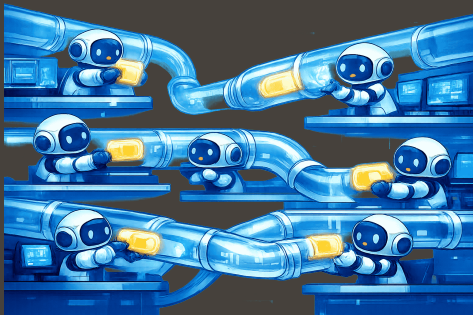
```

Process i:
  loop forever
    Code before the barrier
    Barrier.barrier()
    Code after the barrier
  end loop

```

signalAllC: new operation: All waiting processes are unblocked

Channels



- The previous programming constructs semaphores and monitors are used in shared memory to enable process communication and synchronization.
- In this section: **Channels**
- These do **not** require shared memory
- Synchronization and communication only via **receiving and sending messages**
- Implementation in shared memory **possible**
- But: Can also be used in distributed systems

- Synchronous channels:
Receiving and sending happens in one step, i.e.
 - Sender is blocked until receiver is available
 - Receiver is blocked until sender is available
- Asynchronous channels: Receiving and sending can occur in different steps
- We now consider: **synchronous** channels
- Such channels were introduced by C.A.R Hoare in the so-called **Communicating sequential processes** formalism (CSP)
- Channels are often implemented as libraries for programming languages.
- Google Go: Channels are built in natively

- A channel connects a sending process to a receiving process
- Often allowed: a channel connects **multiple** sending and receiving processes
- Channels are **typed**: Only elements (messages) of the same type can be sent via the channel

Channels: Operations

- Let ch be a channel
- $ch \Leftarrow w$
 - corresponds to: “send w via the channel ch ”
 - where w is a value of the appropriate type
 - we also write $ch \Leftarrow x$, for a program variable x

Semantics: Send the value of variable x via channel ch

- $ch \Rightarrow x$
 - corresponds to “receive via channel ch and set variable x to the received value”
 - Here: Only variables allowed!

Example: Producer / Consumer without Buffering

ch: Channel of type τ

y: Program variable of type τ

Producer:

```
loop forever
(1)  produce e (of type  $\tau$ );
(2)  ch  $\leftarrow$  e;
end loop
```

Consumer:

```
loop forever
(1)  ch  $\Rightarrow$  y;
(2)  consume y;
end loop
```

Example: Producer / Consumer without Buffering

ch: Channel of type τ

y: Program variable of type τ

Producer:

```
loop forever
(1)  produce e (of type  $\tau$ );
(2)  ch  $\leftarrow$  e;
end loop
```

Consumer:

```
loop forever
(1)  ch  $\Rightarrow$  y;
(2)  consume y;
end loop
```

Example: Producer / Consumer without Buffering

ch: Channel of type τ

y: Program variable of type τ

Producer:

```
loop forever
(1)  produce e (of type  $\tau$ );
(2)   $ch \leftarrow e$ ;
end loop
```

Consumer:

```
loop forever
(1)   $ch \Rightarrow y$ ;
(2)  consume y;
end loop
```

Evaluation: Communication not possible, producer is blocked

Example: Producer / Consumer without Buffering

ch: Channel of type τ

y: Program variable of type τ

Producer:

```
loop forever
(1)  produce e (of type  $\tau$ );
(2)   $ch \leftarrow e$ ;
end loop
```

Consumer:

```
loop forever
(1)   $ch \Rightarrow y$ ;
(2)  consume y;
end loop
```

Evaluation: Communication possible,
Communication takes place in one step

Example: Producer / Consumer without Buffering

ch: Channel of type τ

y: Program variable of type τ

Producer:

loop forever

(1) produce e (of type τ);

(2) ch $\leftarrow$ e;

end loop

Consumer:

loop forever

(1) ch $\Rightarrow$ y;

(2) consume e;

end loop

Evaluation: Both program pointers jump directly to the next line!

Example: Producer / Consumer without Buffering

ch: Channel of type τ

y: Program variable of type τ

Producer:

```
loop forever
(1)  produce e (of type  $\tau$ );
(2)  ch  $\Leftarrow$  e;
end loop
```

Consumer:

```
loop forever
(1)  ch  $\Rightarrow$  y;
(2)  consume ;
end loop
```

Sender and consumer block,
as long as no "counterpart" is available

One channel – multiple senders/receivers

Process 1:

$ch \leftarrow \text{True}$

Process 2:

$ch \leftarrow \text{False}$

Process 3:

$ch \Rightarrow x$
print x;

What does process 3 print?

Process 1:

$ch \Rightarrow x$
print x;

Process 2:

$ch \Rightarrow x$
print x;

Process 3:

$ch \Leftarrow \text{True}$

Which process prints True?

Process 1:

$ch \Rightarrow x$
`print x;`

Process 2:

$ch \Rightarrow x$
`print x;`

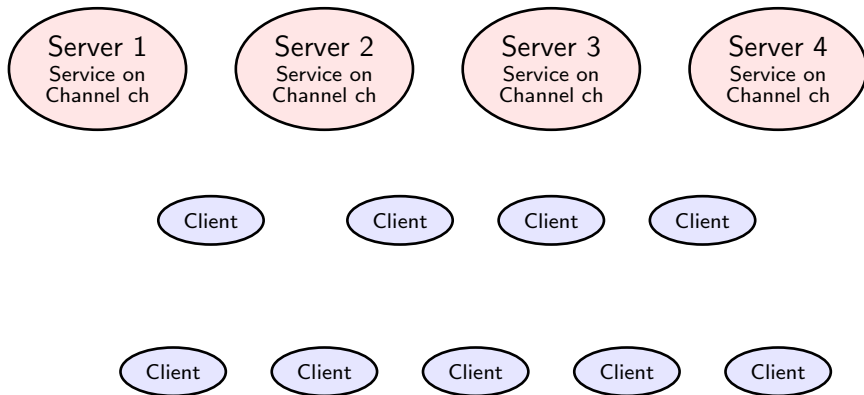
Process 3:

$ch \Leftarrow \text{True}$

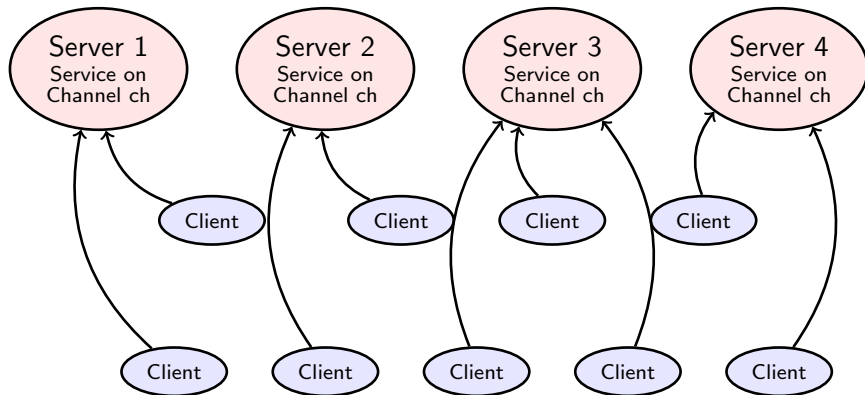
Which process prints True?

Depending on the sequence (quasi random)

Can be Useful ...

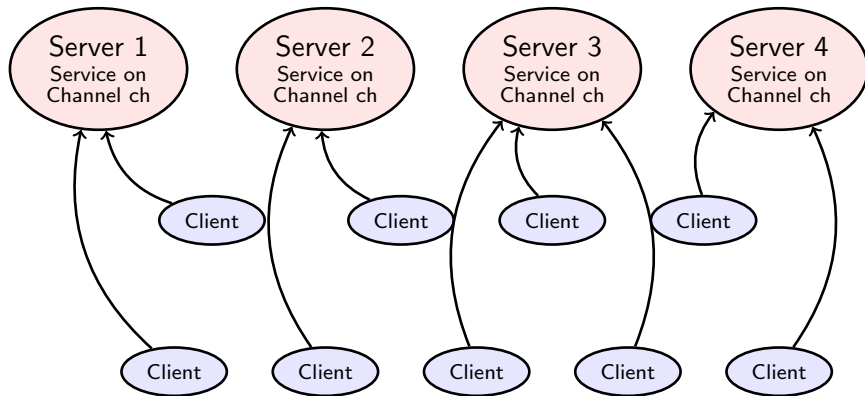


Can be Useful ...



“Automatic load distribution”

Can be Useful ...



“Automatic load distribution”

Disadvantage: everything runs through one channel!

- Idea: A process as a **guard** that guards the critical section
- Only those who receive a message from the guard are allowed to enter the critical section

Mutual Exclusion with Channels (2)

mutex: Channel of type Bool
local: Local variable, initial value irrelevant

Process i

```
loop forever  
(1)  Remaining code;  
(2)  mutex  $\Rightarrow$  local;  
(3)  critical section;  
(4)  mutex  $\Leftarrow$  True;  
end loop
```

Guard

```
loop forever  
(1)  mutex  $\Leftarrow$  True;  
(2)  mutex  $\Rightarrow$  local;  
end loop
```

Mutual Exclusion with Channels

- fulfills mutual exclusion
- is deadlock-free (a waiting process always receives the message)
- **not** starvation-free

Modeling Shared Memory

- Even if no shared memory is available, it can be modeled using channels.
- Idea: A process represents the memory and "administrates" it.
- Access to memory by sending and receiving messages

A Memory Cell

- We consider a memory cell
- Operations: **read** and **write**
- Implementation uses two channels:
 - **requestChannel**:
Read or write requests are sent to the administrator via this channel
 - **replyChannel**:
The administrator responds via this channel.
- Type of cell content: `CellType`
- Type of requestChannel: `(Bool, CellType)`
- Type of replyChannel: `CellType`

A Memory Cell (2)

Variables:

x: Local variable of the server process of type CellType

dummy: Any variable of type CellType

Server process for the cell:

```
loop forever
  requestChannel  $\Rightarrow$  r;
  if fst(r) then // write operation
    x := snd(r);
  else // read operation
    replyChannel  $\Leftarrow$  x;
  end loop
```

Methods for accessing the cell:

```
read(requestChannel, replyChannel) {
  requestChannel  $\Leftarrow$  (False, dummy);
  replyChannel  $\Rightarrow$  x;
  return(x);
}

write(reqCh, replyCh, x) {
  requestChannel  $\Leftarrow$  (True, x)
}
```

A Memory Cell (3)

Variant: Server as a recursive function

Server process for the cell:

```
cell(x) {  
  requestChannel  $\Rightarrow$  r;  
  if fst(r) then // write operation  
    cell(snd(r));  
  else // read operation  
    replyChannel  $\Leftarrow$  x;  
    cell(x); }
```

Methods for accessing the cell:

```
read(requestChannel, replyChannel) {  
  requestChannel  $\Leftarrow$  (False,dummy);  
  replyChannel  $\Rightarrow$  x;  
  return(x);  
}  
  
write(reqCh, replyCh, x) {  
  requestChannel  $\Leftarrow$  (True,x)  
}
```

- Extension by another construct
- Enables waiting to receive on multiple channels
- As soon as a message is received on **one** channel, other channels are no longer considered
- Non-deterministic selection if there are multiple possibilities

```
either  
  ch1 ⇒ var1  
or  
  ch2 ⇒ var2  
or  
  ch3 ⇒ var3
```

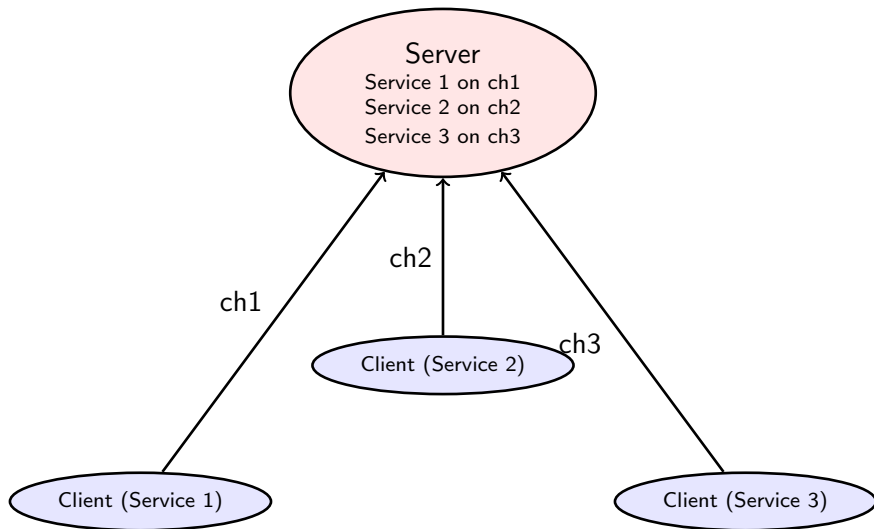
Selective Input (2)

Process 1	Process 2	Process 3	Process 4
either	$ch1 \Leftarrow e1$	$ch2 \Leftarrow e1$	$ch3 \Leftarrow e1$
$ch1 \Rightarrow var1$			
or			
$ch2 \Rightarrow var2$			
or			
$ch3 \Rightarrow var3$			

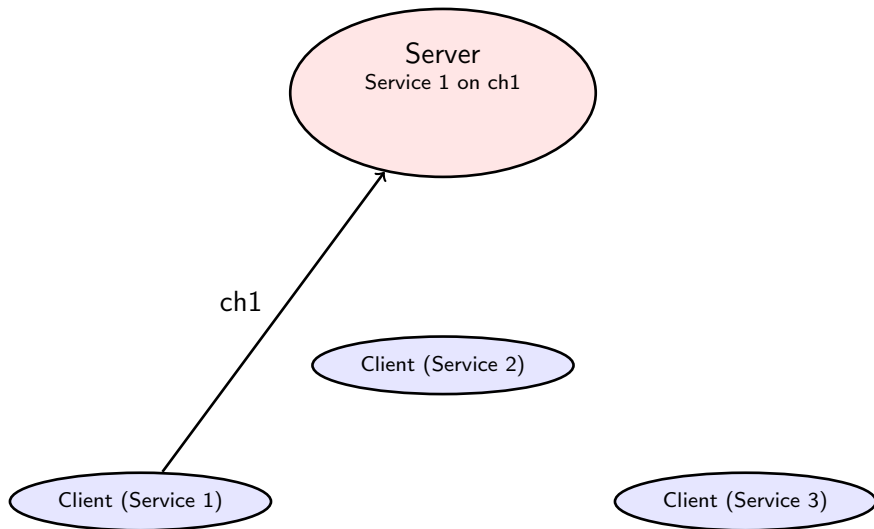
3 Possibilities thereafter

- | Process 1 | Process 2 | Process 3 | Process 4 |
|----------------|-----------|---------------------|---------------------|
| $(var1 := e1)$ | | $ch2 \Leftarrow e2$ | $ch3 \Leftarrow e3$ |
- | Process 1 | Process 2 | Process 3 | Process 4 |
|----------------|---------------------|-----------|---------------------|
| $(var2 := e2)$ | $ch1 \Leftarrow e1$ | | $ch3 \Leftarrow e3$ |
- | Process 1 | Process 2 | Process 3 | Process 4 |
|----------------|---------------------|---------------------|-----------|
| $(var3 := e3)$ | $ch1 \Leftarrow e1$ | $ch2 \Leftarrow e2$ | |

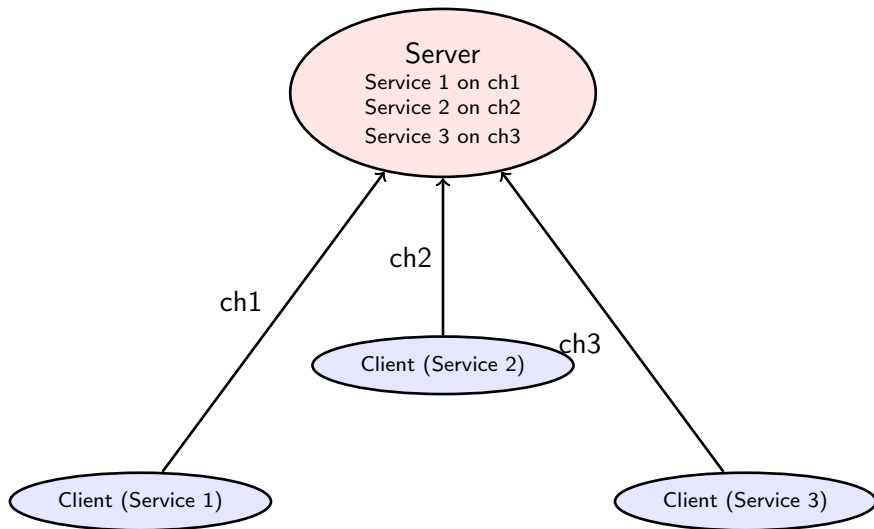
E.g. Useful for ...



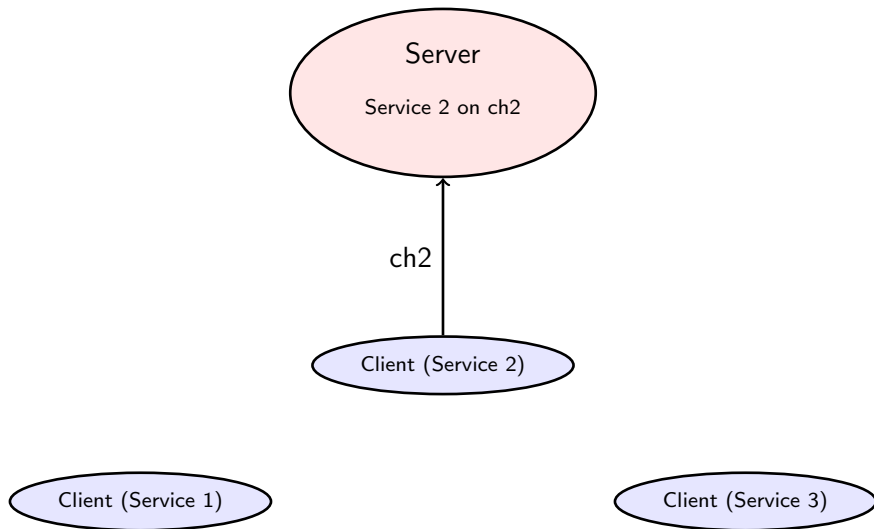
E.g. Useful for ...



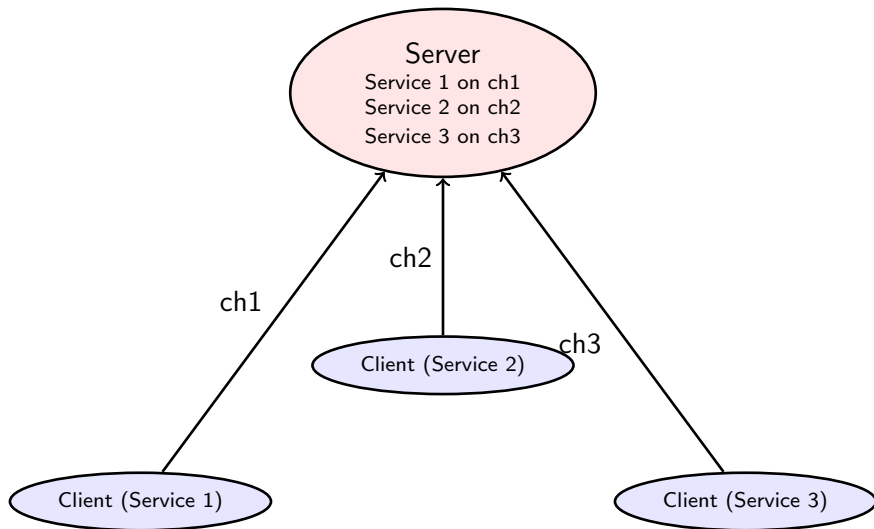
E.g. Useful for ...



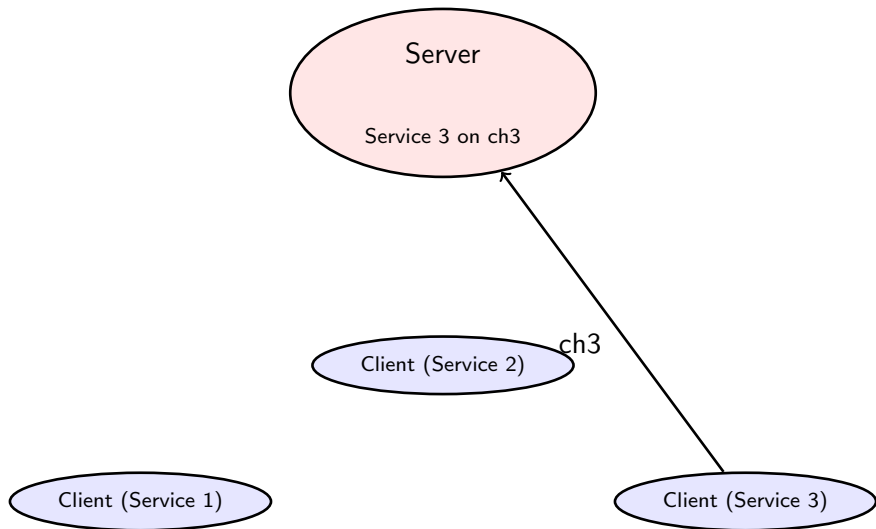
E.g. Useful for ...



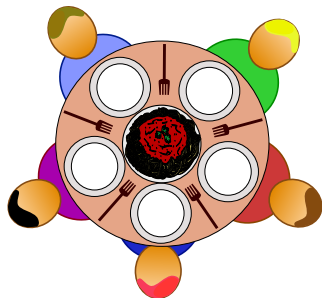
E.g. Useful for ...



E.g. Useful for ...



Dining Philosophers with Channels



- per fork: one process connected via a channel to left and right philosophers
- Philosopher process: Attempts to obtain left and right forks via receipt of messages

Dining Philosophers with Channels (2)

forks : field of channels of type Bool
x: local variables

<u>Philosopher i</u>	<u>fork i</u>
loop forever	loop forever
(1) think;	(1) forks[i] $\Leftarrow$ True
(2) forks[i] $\Rightarrow$ x	(2) forks[i] $\Rightarrow$ x
(3) forks[i+1] $\Rightarrow$ x	end loop
(4) eat;	
(5) forks[i] $\Leftarrow$ True	
(6) forks[i+1] $\Leftarrow$ True	
end loop	

Dining Philosophers with Channels (2)

forks : field of channels of type Bool
x: local variables

<u>Philosopher i</u>	<u>fork i</u>
loop forever	loop forever
(1) think;	(1) forks[i] $\Leftarrow$ True
(2) forks[i] $\Rightarrow$ x	(2) forks[i] $\Rightarrow$ x
(3) forks[i+1] $\Rightarrow$ x	end loop
(4) eat;	
(5) forks[i] $\Leftarrow$ True	
(6) forks[i+1] $\Leftarrow$ True	
end loop	

Not deadlock-free

Dining Philosophers with Channels (2)

forks : field of channels of type Bool
x: local variables

Philosopher

```
loop forever
(1) think;
(2) forks[i+1]  $\Rightarrow$  x
(3) forks[i]  $\Rightarrow$  x
(4) eat;
(5) forks[i+1]  $\Leftarrow$  True
(6) forks[i]  $\Leftarrow$  True
end loop
```

Philosopher i < n

```
loop forever
(1) think;
(2) forks[i]  $\Rightarrow$  x
(3) forks[i+1]  $\Rightarrow$  x
(4) eat;
(5) forks[i]  $\Leftarrow$  True
(6) forks[i+1]  $\Leftarrow$  True
end loop
```

fork i

```
loop forever
(1) forks[i]  $\Leftarrow$  True
(2) forks[i]  $\Rightarrow$  x
end loop
```

Deadlock-free, not starvation-free