Parallel Programming
Programowanie równoległe

Lecture 2: Classical problems in parallel computing.
Semaphores.

Paweł Rzążewski

Selling plane tickets

Recall the example from the last lecture. A webpage offers plane tickets. Only one ticket for a particular flight is left.

1. two users simultaneously notice that the ticket is available,
2. both users try to buy the ticket,
3. wrongly designed system sells the same ticket twice.

Mutual exclusion

Mutual exclusion is a mechanism, which guarantees that only one process may access some piece of memory at the same time. It may be obtained in different ways, e.g.:

- disabling interrupts (hardware solution),
- atomic operations (software solution)

Mutual exclusion

Mutual exclusion is a mechanism, which guarantees that only one process may access some piece of memory at the same time. It may be obtained in different ways, e.g.:

- disabling interrupts (hardware solution),
- atomic operations (software solution)

But mutual exclusion of single operations is not enough!

Non-determinism

Consider the following example.

x is a global variable, with initial value 0

Process 1:	**Process 2:**
$x \leftarrow 1$	$x \leftarrow 2$

Non-determinism

Consider the following example.
x is a global variable, with initial value 0

Process 1:
$x \leftarrow 1$

Process 2:
$x \leftarrow 2$

Scenario 1:

Process 1	Process 2	x
		0
$x \leftarrow 1$		1
	$x \leftarrow 2$	2

Final value: $x = 2$

Non-determinism

Consider the following example.
x is a global variable, with initial value 0

Process 1:
$x \leftarrow 1$

Process 2:
$x \leftarrow 2$

Scenario 1:

Process 1	Process 2	x
		0
$x \leftarrow 1$		1
	$x \leftarrow 2$	2

Final value: $x = 2$

Scenario 2:

Process 1	Process 2	x
		0
	$x \leftarrow 2$	2
$x \leftarrow 1$		1

Final value: $x = 1$

Non-determinism

Consider the following example.
x is a global variable, with initial value 0

Process 1:
$x \leftarrow 1$

Process 2:
$x \leftarrow 2$

Scenario 1:

Process 1	Process 2	x
		0
$x \leftarrow 1$		1
	$x \leftarrow 2$	2

Final value: $x = 2$

Scenario 2:

Process 1	Process 2	x
		0
	$x \leftarrow 2$	2
$x \leftarrow 1$		1

Final value: $x = 1$

The same piece of code may produce different results!

Non-determinism – continued

x, y, z are global variables, with initial values 0

Process 1:	**Process 2:**
$y \leftarrow x + 1$	$y \leftarrow x + 1$
$x \leftarrow y$	$x \leftarrow y$
$z \leftarrow z + y$	$z \leftarrow z + y$

Non-determinism – continued

x, y, z are global variables, with initial values 0

Process 1:	**Process 2:**
$y \leftarrow x + 1$	$y \leftarrow x + 1$
$x \leftarrow y$	$x \leftarrow y$
$z \leftarrow z + y$	$z \leftarrow z + y$

Scenario 1:

Proc. 1	Proc. 2	x	y	z
		0	0	0
$y \leftarrow x + 1$		0	1	0
$x \leftarrow y$		1	1	0
$z \leftarrow z + y$		1	1	1
	$y \leftarrow x + 1$	1	2	1
	$x \leftarrow y$	2	2	1
	$z \leftarrow z + y$	2	2	3

Scenario 2:

Proc. 1	Proc. 2	x	y	z
		0	0	0
$y \leftarrow x + 1$		0	1	0
	$y \leftarrow x + 1$	0	1	0
$x \leftarrow y$		1	1	0
	$x \leftarrow y$	1	1	0
$z \leftarrow z + y$		1	1	1
	$z \leftarrow z + y$	1	1	2

Scenario 3:

Proc. 1	Proc. 2	x	y	z
		0	0	0
$y \leftarrow x + 1$		0	1	0
$x \leftarrow y$		1	1	0
	$y \leftarrow x + 1$	1	2	0
$z \leftarrow z + y$		1	2	2
	$x \leftarrow y$	2	2	2
	$z \leftarrow z + y$	2	2	4

Critical section

As we have seen, mutual exclusion of single operations is not enough to guarantee proper and effective synchronization.

Critical section

The critical section is a part of the process, whose execution should not be interrupted by other processes. Whenever a process enters its critical section, it will be able to finish it without preemption.

Critical section – continued

x, y, z are global variables, with initial values 0

Process 1:
start of critical section
$y \leftarrow x + 1$
$x \leftarrow y$
$z \leftarrow z + y$
end of critical section

Process 2:
start of critical section
$y \leftarrow x + 1$
$x \leftarrow y$
$z \leftarrow z + y$
end of critical section

Critical section – continued

x, y, z are global variables, with initial values 0

Process 1:
start of critical section
$y \leftarrow x + 1$
$x \leftarrow y$
$z \leftarrow z + y$
end of critical section

Process 2:
start of critical section
$y \leftarrow x + 1$
$x \leftarrow y$
$z \leftarrow z + y$
end of critical section

Scenario 1:

Proc. 1	Proc. 2	x	y	z
		0	0	0
$y \leftarrow x + 1$		0	1	0
$x \leftarrow y$		1	1	0
$z \leftarrow z + y$		1	1	1
	$y \leftarrow x + 1$	1	2	1
	$x \leftarrow y$	2	2	1
	$z \leftarrow z + y$	2	2	3

Scenario 2:

Proc. 1	Proc. 2	x	y	z
		0	0	0
	$y \leftarrow x + 1$	0	1	0
	$x \leftarrow y$	1	1	0
	$z \leftarrow z + y$	1	1	1
$y \leftarrow x + 1$		1	2	1
$x \leftarrow y$		2	2	1
$z \leftarrow z + y$		2	2	3

Producer-consumer

Producer-consumer is a very common pattern found in parallel computing. We have two types of processes, sharing a common buffer:

Producer:

```
repeat:
  produce x
  write x to the buffer
```

Consumer:

```
repeat:
  read x from the buffer
  consume x
```

If the buffer is full (x has not been consumed), the producer has to wait. Analogously, if the buffer is empty (nothing has been produced), the consumer has to wait.

Producer-consumer – first attempt

Producer:

```
for i ← 1 to N
   produce;
   while(buf = 1)
      do nothing;
   buf ← 1;
```

Consumer:

```
for i ← 1 to N
   while(buf = 0)
      do nothing;
   consume from buf;
   buf ← 0;
```

Producer-consumer – first attempt

Producer:

```
for i ← 1 to N
   produce;
   while(buf = 1)
      do nothing;
   buf ← 1;
```

Consumer:

```
for i ← 1 to N
   while(buf = 0)
      do nothing;
   consume from buf;
   buf ← 0;
```

Does it ensure mutual exclusion?

Producer-consumer – first attempt

Producer:

```
for i ← 1 to N
   produce;
   while(buf = 1)
      do nothing;
   buf ← 1;
```

Consumer:

```
for i ← 1 to N
   while(buf = 0)
      do nothing;
   consume from buf;
   buf ← 0;
```

Does it ensure mutual exclusion? Yes.

Producer-consumer – first attempt

Producer:

```
for i ← 1 to N
   produce;
   while(buf = 1)
      do nothing;
   buf ← 1;
```

Consumer:

```
for i ← 1 to N
   while(buf = 0)
      do nothing;
   consume from buf;
   buf ← 0;
```

Does it ensure mutual exclusion? Yes.

Is it a good solution?

Producer-consumer – first attempt

Producer:

```
for i ← 1 to N
   produce;
   while(buf = 1)
      do nothing;
   buf ← 1;
```

Consumer:

```
for i ← 1 to N
   while(buf = 0)
      do nothing;
   consume from buf;
   buf ← 0;
```

Does it ensure mutual exclusion? Yes.

Is it a good solution? No.

Producer-consumer – first attempt

Producer:

for $i \leftarrow 1$ to N
  produce;
  while(buf $= 1$)
    do nothing;
  buf $\leftarrow 1$;

Consumer:

for $i \leftarrow 1$ to N
  while(buf $= 0$)
    do nothing;
  consume from buf;
  buf $\leftarrow 0$;

Does it ensure mutual exclusion? Yes.

Is it a good solution? No.

Busy waiting is always an error – it is extremely inefficient.

Producer-consumer – more processes

What happens if we have more than one producer and more than one consumer?

Producer:

```
produce;
while(buf = 1)
  do nothing;
buf ← 1;
```

Consumer:

```
while(buf = 0)
  do nothing;
consume from buf;
buf ← 0;
```

Producer-consumer – more processes

What happens if we have more than one producer and more than one consumer?

Producer:

```
produce;
while(buf = 1)
  do nothing;
buf ← 1;
```

Consumer:

```
while(buf = 0)
  do nothing;
consume from buf;
buf ← 0;
```

Does it work?

Producer-consumer – more processes

What happens if we have more than one producer and more than one consumer?

Producer:

```
produce;
while(buf = 1)
  do nothing;
buf ← 1;
```

Consumer:

```
while(buf = 0)
  do nothing;
consume from buf;
buf ← 0;
```

Does it work? No.

Producer-consumer – more processes – ctd.

Let us have 2 producers and 2 consumers.

Producer 1	Producer 2	Consumer 1	Consumer 2	buf
				0
...				0
buf $\leftarrow$ 1				1
(terminate)				1
		while(buf = 0)		1
			while(buf = 0)	1
		consume		1
			consume	1
		buf $\leftarrow$ 0		0
		(terminate)		0
			buf $\leftarrow$ 0	0
			(terminate)	0
	...			0
	buf $\leftarrow$ 1			1
	(terminate)			1

Producer-consumer – more processes – ctd.

Let us have 2 producers and 2 consumers.

Producer 1	Producer 2	Consumer 1	Consumer 2	buf
				0
...				0
buf $\leftarrow$ 1				1
(terminate)				1
		while(buf $=$ 0)		1
			while(buf $=$ 0)	1
		consume		1
			consume	1
		buf $\leftarrow$ 0		0
		(terminate)		0
			buf $\leftarrow$ 0	0
			(terminate)	0
	...			0
	buf $\leftarrow$ 1			1
	(terminate)			1

The same item is consumed twice!

Producer-consumer – even more processes

What if we had 4 Producers and 4 Consumers?

Producer 1	Producer 2	Consumer 1	Consumer 2	buf
				0
...				0
buf $\leftarrow$ 1				1
(terminate)				1
		while(buf $=$ 0)		1
			while(buf $=$ 0)	1
		consume		1
			consume	1
		buf $\leftarrow$ 0		0
		(terminate)		0
			buf $\leftarrow$ 0	0
			(terminate)	0
	...			0
	buf $\leftarrow$ 1			1
	(terminate)			1

Producer-consumer – even more processes – ctd.

Producer 3	Producer 4	Consumer 3	Consumer 4	buf
				1
		while(buf = 0)		1
			while(buf = 0)	1
		consume		1
			consume	1
		buf ← 0		0
		(terminate)		0
			buf ← 0	0
			(terminate)	0
...				0
buf ← 1				1
(terminate)				1
	...			0
	while(buf = 1)			1
	do nothing			1

Producer-consumer – even more processes – ctd.

Producer 3	Producer 4	Consumer 3	Consumer 4	buf
				1
		while(buf $=$ 0)		1
			while(buf $=$ 0)	1
		consume		1
			consume	1
		buf $\leftarrow$ 0		0
		(terminate)		0
			buf $\leftarrow$ 0	0
			(terminate)	0
...				0
buf $\leftarrow$ 1				1
(terminate)				1
	...			0
	while(buf $=$ 1)			1
	do nothing			1

Producer 4 will never terminate, as *buf* will always remain 1.

How to manage a critical section?

The solution should have the following properties:

- each process should be treated equally,
- it should be independent on the speed of processes,
- the conflicts should be solved in finite time,
- the parts of processes, which are outside the critical section, should be independent.

Semaphores

A solution satisfying these criteria was proposed by Edsger Dijkstra (1930–2002).

Picture source: Wikipedia

It was inspired by the semaphores used in the rail transport.

Counting semaphores

A semaphore is an integer variable v, accompanied with a collection q of processes (usually a queue or a priority queue). It has two operations: `wait` (acquire, decrement, down, pend, procure, P) and `signal` (release, increment, up, port, vacate, V).

Wait:

```
if v = 0
  add yourself to q
  sleep
v ← v − 1
```

Signal:

```
v ← v + 1
if |q| ≥ 1
  wake the first process in q
```

Counting semaphores

A semaphore is an integer variable v, accompanied with a collection q of processes (usually a queue or a priority queue). It has two operations: `wait` (acquire, decrement, down, pend, procure, P) and `signal` (release, increment, up, port, vacate, V).

Wait:

if $v = 0$
  add yourself to q
  sleep
$v \leftarrow v - 1$

Signal:

$v \leftarrow v + 1$
if $|q| \geq 1$
  wake the first process in q

Both these operations are atomic!

Counting semaphores

A semaphore is an integer variable v, accompanied with a collection q of processes (usually a queue or a priority queue). It has two operations: `wait` (acquire, decrement, down, pend, procure, P) and `signal` (release, increment, up, port, vacate, V).

Wait:

```
if v = 0
  add yourself to q
  sleep
v ← v − 1
```

Signal:

```
v ← v + 1
if |q| ≥ 1
  wake the first process in q
```

Both these operations are atomic!

A binary semaphore is a semaphore that may only be 0 or 1 (assume that we do not Signal a semaphore with value 1).

Producer-consumer using semaphores

Global variables: $buf = 0$, and two binary semaphores $s_empty = 1$ and $s_full = 0$.

Producer:

```
produce;
Wait(s_empty);
buf ← 1;
Signal(s_full);
```

Consumer:

```
Wait(s_full);
consume from buffer;
buf ← 0;
Signal(s_empty);
```

Note that $s_full = 1$ if $buf = 1$ and $s_empty = 1$ if $buf = 0$. The semaphores indicate if the particular process may enter its critical section.

Producer-consumer using semaphores

Global variables: $buf = 0$, and two binary semaphores $s_empty = 1$ and $s_full = 0$.

Producer:

```
produce;
Wait(s_empty);
buf ← 1;
Signal(s_full);
```

Consumer:

```
Wait(s_full);
consume from buffer;
buf ← 0;
Signal(s_empty);
```

Note that $s_full = 1$ if $buf = 1$ and $s_empty = 1$ if $buf = 0$. The semaphores indicate if the particular process may enter its critical section.

Is it always true that $s_full + s_empty = 1$?

Producer-consumer using semaphores

Global variables: $buf = 0$, and two binary semaphores $s_empty = 1$ and $s_full = 0$.

Producer:

```
produce;
Wait(s_empty);
buf ← 1;
Signal(s_full);
```

Consumer:

```
Wait(s_full);
consume from buffer;
buf ← 0;
Signal(s_empty);
```

Note that $s_full = 1$ if $buf = 1$ and $s_empty = 1$ if $buf = 0$. The semaphores indicate if the particular process may enter its critical section.

Is it always true that $s_full + s_empty = 1$?

It may happen that $s_full = 0$ and $s_empty = 0$ (during the consumption)!

Producer-consumer

We can also imagine a situation, when each process is a consumer and a producer at the same time. We have two buffers $buf_1 = 0$ and $buf_2 = 0$. We also have a pair of binary semaphores for each buffer: $s_empty_1 = 1$, $s_full_1 = 0$, and $s_empty_2 = 1$ and $s_full_2 = 0$.

Cons-Prod1:

```
Wait(s1_full);
consume from buf1;
buf1 ← 0;
Signal(s_empty1);
produce;
Wait(s_empty2);
buf2 ← 1;
Signal(s_full2);
```

Cons-Prod2:

```
Wait(s2_full);
consume from buf2;
buf2 ← 0;
Signal(s_empty2);
produce;
Wait(s_empty1);
buf1 ← 1;
Signal(s_full1);
```

Producer-consumer

We can also imagine a situation, when each process is a consumer and a producer at the same time. We have two buffers $buf_1 = 0$ and $buf_2 = 0$. We also have a pair of binary semaphores for each buffer: $s_empty_1 = 1$, $s_full_1 = 0$, and $s_empty_2 = 1$ and $s_full_2 = 0$.

Cons-Prod1:

```
Wait(s1_full);
consume from buf1;
buf1 ← 0;
Signal(s_empty1);
produce;
Wait(s_empty2);
buf2 ← 1;
Signal(s_full2);
```

Cons-Prod2:

```
Wait(s2_full);
consume from buf2;
buf2 ← 0;
Signal(s_empty2);
produce;
Wait(s_empty1);
buf1 ← 1;
Signal(s_full1);
```

What is wrong?

Deadlock

Cons-Prod1:

```
Wait(s1_full);
consume from buf1;
buf1 ← 0;
Signal(s_empty1);
produce;
Wait(s_empty2);
buf2 ← 1;
Signal(s_full2);
```

Cons-Prod2:

```
Wait(s2_full);
consume from buf2;
buf2 ← 0;
Signal(s_empty2);
produce;
Wait(s_empty1);
buf1 ← 1;
Signal(s_full1);
```

Deadlock

The deadlock is a situation, when two actions wait for each other to finish. This way none of them can terminate.

Deadlock

Cons-Prod1:

```
Wait(s_1_full);
consume from buf_1;
buf_1 ← 0;
Signal(s_empty_1);
produce;
Wait(s_empty_2);
buf_2 ← 1;
Signal(s_full_2);
```

Cons-Prod2:

```
Wait(s_2_full);
consume from buf_2;
buf_2 ← 0;
Signal(s_empty_2);
produce;
Wait(s_empty_1);
buf_1 ← 1;
Signal(s_full_1);
```

Deadlock

The deadlock is a situation, when two actions wait for each other to finish. This way none of them can terminate.

The deadlock is always a critical error of the programmer!

Producer-consumer with an infinite buffer

Now suppose that instead of a one-element buffer we have an infinite buffer. We use the following variables:
buf [], initially empty
first_free = 1 – the index of the first empty cell in *buf*
last_occ = 1 – the index of the last not-consumed cell in *buf*
counting semaphore *s_full* = 0 (why not *s_empty* as well?)
binary semaphore *s_p* = 1 for mutual exclusion of producers
binary semaphore *s_c* = 1 for mutual exclusion of producers

Producer-consumer with an infinite buffer

Now suppose that instead of a one-element buffer we have an infinite buffer. We use the following variables:

buf[], initially empty

first_free = 1 – the index of the first empty cell in *buf*

last_occ = 1 – the index of the last not-consumed cell in *buf*

counting semaphore *s_full* = 0 (why not *s_empty* as well?)

binary semaphore *s_p* = 1 for mutual exclusion of producers

binary semaphore *s_c* = 1 for mutual exclusion of producers

Producer:

```
produce x;
Wait(s_p);
buf[first_free] ← x;
first_free ← first_free + 1;
Signal(s_full);
Signal(s_p);
```

Consumer:

```
Wait(s_full);
Wait(s_c);
x ← buf[last_occ]
last_occ ← last_occ + 1
Signal(s_c);
consume x;
```

Producer-consumer with finite buffer

The assumption that *buf* is infinite may not be very realistic.

buf [] – a k-element buffer

$first_free = 1$

$last_occ = 1$

counting semaphore $s_full = 0$, $s_empty = k$

binary semaphore $s_p = 1$

binary semaphore $s_c = 1$

Producer-consumer with finite buffer

The assumption that *buf* is infinite may not be very realistic.

buf [] – a k-element buffer

$first_free = 1$

$last_occ = 1$

counting semaphore $s_full = 0$, $s_empty = k$

binary semaphore $s_p = 1$

binary semaphore $s_c = 1$

Producer:

```
produce x;
Wait(s_empty);
Wait(s_p);
buf[first_free] ← x;
first_free ← (first_free + 1) mod k;
Signal(s_full);
Signal(s_p);
```

Consumer:

```
Wait(s_full);
Wait(s_c);
x ← buf[last_occ];
last_occ ← (last_occ + 1) mod k;
Signal(s_c);
Signal(s_empty);
consume x;
```

Producer-consumer, when copying takes much time

Focus on these two innocent-looking lines:

Producer:

```
produce x;
Wait(s_empty);
Wait(s_p);
buf[first_free] ← x;
first_free ← (first_free + 1) mod k;
Signal(s_full);
Signal(s_p);
```

Consumer:

```
Wait(s_full);
Wait(s_c);
x ← buf[last_occ];
last_occ ← (last_occ + 1) mod k;
Signal(s_c);
Signal(s_empty);
consume x;
```

What if the buffer contains real data, not just integers? What if copying this data takes much time?
We are blocking whole buffer (for producers or consumers), but we need just one cell!

Producer-consumer, when copying takes much time

How to improve it?

Producer-consumer, when copying takes much time

How to improve it?

Hint

How is it done in theaters?

Producer-consumer, when copying takes much time

How to improve it?

Hint

How is it done in theaters?

Hint

Taking your place takes much time. How do they guarantee that no two people will seat on the same place, without blocking the whole room?

Producer-consumer, when copying takes much time

How to improve it?

Hint

How is it done in theaters?

Hint

Taking your place takes much time. How do they guarantee that no two people will seat on the same place, without blocking the whole room?

Use tickets, which are easy (and quick) to obtain!

Producer-consumer, when copying takes much time

Introduce two queues: *q_full* containing the indices of cells, which are full, and *q_empty* with the indices of empty cells.

Producer-consumer, when copying takes much time

Introduce two queues: *q_full* containing the indices of cells, which are full, and *q_empty* with the indices of empty cells.

$buf[]$ – a k-element buffer
$q_empty = (1, 2, \ldots, n)$ – indices of empty cells
$q_full = ()$ – indices of full cells
counting semaphore $s_empty = k$ – number of empty cells
counting semaphore $s_full = 0$ – number of full cells
binary semaphore $s_qe = 1$ – blocking access to *q_empty*
binary semaphore $s_qf = 1$ – blocking access to *q_full*

Producer-consumer, when copying takes much time

Producer:

produce x;
Wait(s_empty);
Wait(s_qe);
$i \leftarrow Dequeue(q_empty)$;
Signal(s_qe);
$buf[i] \leftarrow x$;
Wait(s_qf);
$Enqueue(q_full, i)$;
Signal(s_qf);
Signal(s_full);

Consumer:

Wait(s_full);
Wait(s_qf);
$i \leftarrow Dequeue(q_full)$;
Signal(s_qf);
$x \leftarrow buf[i]$;
Wait(s_qe);
$Enqueue(q_empty, i)$;
Signal(s_qe);
Signal(s_empty);
consume x;

When executing the highlighted lines, only $buf[i]$ is blocked – other processed may access the rest of the buffer.

Readers-writers

Readers-writers is another classical synchronization scheme. Again we have two types of processes – readers and writers. They use a common piece of memory, called the *page*. Writer writes on the page. No other process may access the page at this time. The readers only read – many readers may read at the same time. Of course we can have many writers and readers.

Readers-writers – solution

$r_count = 0$ – the number of readers currently reading the page
binary semaphore $s_rc = 1$ – blocking access to r_count
binary semaphore $s_p = 1$ – blocking access to the page

Writer:

```
Wait(s_p);
write on the page;
Signal(s_p);
```

Reader:

```
Wait(s_rc);
r_count ← r_count + 1
if r_count = 1
   Wait(s_p);
Signal(s_rc);
read the page;
Wait(s_rc);
r_count ← r_count − 1
if r_count = 0
   Signal(s_p);
Signal(s_rc);
```

Starvation

What is the problem with this solution?

Writer:

```
Wait(s_p);
write on the page;
Signal(s_p);
```

Reader:

```
Wait(s_rc);
r_count ← r_count + 1
if r_count = 1
   Wait(s_p);
Signal(s_rc);
read the page;
Wait(s_rc);
r_count ← r_count − 1
if r_count = 0
   Signal(s_p);
Signal(s_rc);
```

Starvation

What is the problem with this solution?

Writer:

```
Wait(s_p);
write on the page;
Signal(s_p);
```

Reader:

```
Wait(s_rc);
r_count ← r_count + 1
if  r_count = 1
   Wait(s_p);
Signal(s_rc);
read the page;
Wait(s_rc);
r_count ← r_count − 1
if  r_count = 0
   Signal(s_p);
Signal(s_rc);
```

Writer may not be able to write anything if the readers come!

Starvation

Starvation

Starvation is a situation when a process is perpetually denied the access to some resource.

It may be caused by different results:

- wrong synchronization scheme,
- lack of resources,
- priorities of processes.

Readers-writers using semaphores, priority to writers

We enforce a priority of writers – no reader can start reading if a writer is waiting.

$r_count = 0$ – the number of readers currently reading the page

$w_count = 0$ – the number of writers waiting

binary semaphore $s_rc = 1$ – blocking access to r_count

binary semaphore $s_wc = 1$ – blocking access to w_count

binary semaphore $s_p = 1$ – blocking access to the page

binary semaphore $s_pri = 1$ – enforcing the priority of writers

Readers-writers using semaphores, priority to writers

Writer:

```
Wait(s_wc);
w_count ← w_count + 1;
if w_count = 1
   Wait(s_pri);
Signal(s_wc);
Wait(s_p);
write on the page;
Signal(s_p);
Wait(s_wc);
w_count ← w_count − 1;
if w_count = 0
   Signal(s_pri);
Signal(s_wc);
```

Reader:

```
Wait(s_pri);
Wait(s_rc);
r_count ← r_count + 1;
if r_count = 1
   Wait(s_p);
Signal(s_rc);
Signal(s_pri);
read from the page;
Wait(s_rc);
r_count ← r_count − 1;
if r_count = 0
   Signal(s_p);
Signal(s_rc);
```

Readers-writers using semaphores, priority to writers

Writer:

```
Wait(s_wc);
w_count ← w_count + 1;
if w_count = 1
  Wait(s_pri);
Signal(s_wc);
Wait(s_p);
write on the page;
Signal(s_p);
Wait(s_wc);
w_count ← w_count − 1;
if w_count = 0
  Signal(s_pri);
Signal(s_wc);
```

Reader:

```
Wait(s_pri);
Wait(s_rc);
r_count ← r_count + 1;
if r_count = 1
  Wait(s_p);
Signal(s_rc);
Signal(s_pri);
read from the page;
Wait(s_rc);
r_count ← r_count − 1;
if r_count = 0
  Signal(s_p);
Signal(s_rc);
```

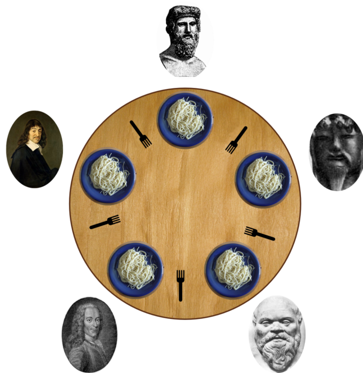
There is still something wrong – try to find out what.

Dining philosophers

N philosophers sit at a round table. For most of the time they are thinking. When they get hungry, they pick up two forks (note that each fork is shared by two philosophers!) and eat. Then they get back to thinking.

Philosopher:

```
repeat:
   think;
   pick up forks;
   eat;
   put down forks;
```



Picture source: Wikipedia

Dining philosophers – solution 1

Philosopher:

```
repeat:
   think;
   wait for the left fork;
   pick up the left fork;
   wait for the right fork;
   pick up the right fork;
   eat;
   put down the left fork;
   put down the right fork;
```

Dining philosophers – solution 1

Philosopher:

```
repeat:
  think;
  wait for the left fork;
  pick up the left fork;
  wait for the right fork;
  pick up the right fork;
  eat;
  put down the left fork;
  put down the right fork;
```

What is wrong?

Dining philosophers – solution 1

Philosopher:

```
repeat:
  think;
  wait for the left fork;
  pick up the left fork;
  wait for the right fork;
  pick up the right fork;
  eat;
  put down the left fork;
  put down the right fork;
```

What is wrong?
Deadlock.

Dining philosophers – solution 2

Philosopher:

```
repeat:
  think;
  wait for the left fork;
  pick up the left fork;
  if the right fork is occupied
    put down the left fork;
    go to beginning;
  pick up the right fork;
  eat;
  put down the left fork;
  put down the right fork;
```

Dining philosophers – solution 2

Philosopher:

```
repeat:
   think;
   wait for the left fork;
   pick up the left fork;
   if the right fork is occupied
      put down the left fork;
      go to beginning;
   pick up the right fork;
   eat;
   put down the left fork;
   put down the right fork;
```

What is wrong?

Dining philosophers – solution 2

Philosopher:

```
repeat:
  think;
  wait for the left fork;
  pick up the left fork;
  if the right fork is occupied
    put down the left fork;
    go to beginning;
  pick up the right fork;
  eat;
  put down the left fork;
  put down the right fork;
```

What is wrong?

Busy waiting.

Dining philosophers – correct solution

We have to make sure that both forks are picked up at the same time!
We introduce a table of states of philosophers *state*[]. Each philosopher can be in one of the states: T(hinking), H(ungry), E(ating). Additionally, we have a table *sem*[] of semaphores. Each philosopher has his own semaphore. Finally, we have a binary semaphore *s* to block the access to *state*[].

Dining philosophers – correct solution

We have to make sure that both forks are picked up at the same time!
We introduce a table of states of philosophers *state*[]. Each philosopher can be in one of the states: T(hinking), H(ungry), E(ating). Additionally, we have a table *sem*[] of semaphores. Each philosopher has his own semaphore. Finally, we have a binary semaphore *s* to block the access to *state*[].
The crucial building block of our solution is the following procedure, testing if the *i*-th philosopher can start eating (indices are computed modulo N).

test(i)

if $state[i-1] \neq E$ and $state[i] = H$ and $state[i+1] \neq E$:
  $state[i] \leftarrow E$
  Signal($sem[i]$)

Dining philosophers – correct solution – ctd.

Now we are ready to present the function of the philosopher.

Philosopher(i)

```
repeat:
  think;
  Wait(s);
  state[i] = H;
  test(i);
  Signal(s);
  Wait(sem[i]);
  take forks;
  eat;
  release forks;
  Wait(s);
  state[i] = T;
  test(i+1); test(i-1);
  Signal(s);
```

If the resources are not available, the philosopher waits on $sem[i]$. It can only be released when calling the function $test[i]$ (either by philosopher i or by one of his neighbors).
Whenever $sem[i]$ is released, $state[i]$ is set to E, so no neighbor of i will start eating. So we can pick up both forks in any way we like.

Unisex bathroom problem

Design a synchronization protocol for the following problem:

1. there is one toilet that is used by both women and men,
2. assume that the capacity of the toilet is unbounded (it's a really HUGE toilet),
3. however, we do not allow men and women to use the toilet at the same time; if the women are inside, the men have to wait and vice versa,

Be careful to avoid starvation (in the technical sense!).

Binary semaphores vs. counting semaphores

Counting semaphores seem to be more powerful than binary semaphores. How to emulate a counting semaphore with binary semaphores?

Binary semaphores vs. counting semaphores

Counting semaphores seem to be more powerful than binary semaphores. How to emulate a counting semaphore with binary semaphores?

Initialize(K):

```
int val ← K
BinSem gate ← init(min(1,K));
BinSem mutex ← init(1);
```

Wait:

```
Wait(gate);
Wait(mutex);
val ← val -1;
if (val > 0):
   Signal(gate)
Signal(mutex)
```

Signal:

```
Wait(mutex);
val ← val + 1;
if (val = 1):
   Signal(gate);
Signal(mutex);
```

More types of semaphores

Design the following variations of semaphores:

1. Semaphore with `Check`: a semaphore with additional operation `Check`(), which returns *false* if the value in the semaphore is 0, or returns *true* and decreases the value, if it is positive.
2. Semaphore with two kinds of processes: a semaphore, which can be accessed by two types of processes, and processes of the first type always have priority over the second type.

More types of semaphores

Design the following variations of semaphores:

1. Semaphore with `Check`: a semaphore with additional operation `Check`(), which returns *false* if the value in the semaphore is 0, or returns *true* and decreases the value, if it is positive.
2. Semaphore with two kinds of processes: a semaphore, which can be accessed by two types of processes, and processes of the first type always have priority over the second type.
3. Semaphore with priority of processors: a semaphore, in which every process has a priority (integer) and first we wake the processes with higher priority.

More types of semaphores

Design the following variations of semaphores:

1. Semaphore with `Check`: a semaphore with additional operation `Check()`, which returns *false* if the value in the semaphore is 0, or returns *true* and decreases the value, if it is positive.
2. Semaphore with two kinds of processes: a semaphore, which can be accessed by two types of processes, and processes of the first type always have priority over the second type.
3. Semaphore with priority of processors: a semaphore, in which every process has a priority (integer) and first we wake the processes with higher priority.
4. Generalized semaphores: a counting semaphore with two operations: `Wait`(n) and `Signal`(n), which decrease/increase the value of the semaphore by n.