The relative powers of shared-variable and message-passing communication:

possibility and impossibility of speed-independent mutual exclusion.

Rob van Glabbeek

University of Edinburgh, Scotland

University of New South Wales, Sydney, Australia

Stanford University

2026

The Problem

Two concurrent computers should not try to update a common database at the same time.

The Problem

Two concurrent computers should not try to update a common database at the same time.

Let one computer A deduct £100 from your back account for a purchase, where another computer B adds £5 interest.

The Problem

Two concurrent computers should not try to update a common database at the same time.

Let one computer A deduct £100 from your back account for a purchase, where another computer B adds £5 interest.

The code of A: Let s be your old account value.
Calculate $s' = s + 100$. Overwrite s with s'.

The code of B: Let t be your old account value.
Calculate $t' = t + 5$. Overwrite t with t'.

What will be your balance after these operations?

The Mutual Exclusion Problem

How do we ensure certain parts of a program are executed atomically?

To solve this for real systems, we need solutions to the *mutual exclusion problem*.

The Mutual Exclusion Problem

How do we ensure certain parts of a program are executed atomically? To solve this for real systems, we need solutions to the *mutual exclusion problem*.

A program may have *critical sections*, and at any time at most one process may be in a critical section.

The Mutual Exclusion Problem

How do we ensure certain parts of a program are executed atomically? To solve this for real systems, we need solutions to the *mutual exclusion problem*.

A program may have *critical sections*, and at any time at most one process may be in a critical section.

A program with critical sections operates as follows:

forever do	**forever do**
non-critical section	*non-critical section*
entry protocol	*entry protocol*
critical section	**critical section**
exit protocol	*exit protocol*

The Mutual Exclusion Problem

How do we ensure certain parts of a program are executed atomically? To solve this for real systems, we need solutions to the *mutual exclusion problem*.

A program may have *critical sections*, and at any time at most one process may be in a critical section.

A program with critical sections operates as follows:

forever do	**forever do**
non-critical section	*non-critical section*
entry protocol	*entry protocol*
critical section	**critical section**
exit protocol	*exit protocol*

The non-critical section models the possibility that a process may do something else. It can take any amount of time (even infinite).

The Mutual Exclusion Problem

How do we ensure certain parts of a program are executed atomically? To solve this for real systems, we need solutions to the *mutual exclusion problem*.

A program may have *critical sections*, and at any time at most one process may be in a critical section.

A program with critical sections operates as follows:

forever do *non-critical section* *entry protocol* **critical section** *exit protocol*	**forever do** *non-critical section* *entry protocol* **critical section** *exit protocol*

The non-critical section models the possibility that a process may do something else. It can take any amount of time (even infinite). Our task is to find an entry and exit protocol such that two properties are satisfied.

Desiderata

We want to ensure two main properties:

- Mutual Exclusion No two processes are in their critical section at the same time.

Desiderata

We want to ensure two main properties:

- Mutual Exclusion No two processes are in their critical section at the same time.
- Starvation Freedom Once it enters its entry protocol, a process will eventually be able to execute its critical section.

Desiderata

We want to ensure two main properties:

- Mutual Exclusion No two processes are in their critical section at the same time.
- Starvation Freedom Once it enters its entry protocol, a process will eventually be able to execute its critical section.

In case starvation freedom fails, we at least seek a weaker property:

- Deadlock Freedom Once some process enters its entry protocol, some process will eventually be able to execute its critical section.

First Attempt

We can implement **await** using a busy-waiting loop.

var $turn := P$	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 **await** $turn = P$;	q_2 **await** $turn = Q$;
p_3 **critical section**	q_3 **critical section**
p_4 $turn := Q$	q_4 $turn := P$

Question

Mutual Exclusion?

First Attempt

We can implement **await** using a busy-waiting loop.

var $turn := P$	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 **await** $turn = P$;	q_2 **await** $turn = Q$;
p_3 **critical section**	q_3 **critical section**
p_4 $turn := Q$	q_4 $turn := P$

Question

Mutual Exclusion? Yup!

First Attempt

We can implement **await** using a busy-waiting loop.

var *turn* := P	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 **await** $turn = P$;	q_2 **await** $turn = Q$;
p_3 **critical section**	q_3 **critical section**
p_4 $turn := Q$	q_4 $turn := P$

Question

Mutual Exclusion? Yup!
Starvation Freedom?

First Attempt

We can implement **await** using a busy-waiting loop.

var $turn := P$	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 **await** $turn = P$;	q_2 **await** $turn = Q$;
p_3 **critical section**	q_3 **critical section**
p_4 $turn := Q$	q_4 $turn := P$

Question

Mutual Exclusion? Yup!
Starvation Freedom? Nope!

First Attempt

We can implement **await** using a busy-waiting loop.

var *turn* := *P*	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 **await** $turn = P$;	q_2 **await** $turn = Q$;
p_3 **critical section**	q_3 **critical section**
p_4 $turn := Q$	q_4 $turn := P$

Question

Mutual Exclusion? Yup!

Starvation Freedom? Nope! What if q_1 never finishes?

First Attempt

We can implement **await** using a busy-waiting loop.

var *turn* := *P*	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 **await** $turn = P$;	q_2 **await** $turn = Q$;
p_3 **critical section**	q_3 **critical section**
p_4 $turn := Q$	q_4 $turn := P$

Question

Mutual Exclusion? Yup!
Starvation Freedom? Nope! What if q_1 never finishes?
Deadlock Freedom?

First Attempt

We can implement **await** using a busy-waiting loop.

var $turn := P$	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 **await** $turn = P$;	q_2 **await** $turn = Q$;
p_3 **critical section**	q_3 **critical section**
p_4 $turn := Q$	q_4 $turn := P$

Question

Mutual Exclusion? Yup!
Starvation Freedom? Nope! What if q_1 never finishes?
Deadlock Freedom? Nope!

First Attempt

We can implement **await** using a busy-waiting loop.

var $turn := P$	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 **await** $turn = P$;	q_2 **await** $turn = Q$;
p_3 **critical section**	q_3 **critical section**
p_4 $turn := Q$	q_4 $turn := P$

Question

Mutual Exclusion? Yup!

Starvation Freedom? Nope! What if q_1 never finishes?

Deadlock Freedom? Nope! Same reason

Second Attempt

var *readyP*, *readyQ* := False, False	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 **await** *readyQ* = False;	q_2 **await** *readyP* = False;
p_3 *readyP* := True;	q_3 *readyQ* := True;
p_4 **critical section**	q_4 **critical section**
p_7 *readyP* := False	q_7 *readyQ* := False

Second Attempt

var *readyP*, *readyQ* := False, False	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 **await** *readyQ* = False;	q_2 **await** *readyP* = False;
p_3 *readyP* := True;	q_3 *readyQ* := True;
p_4 **critical section**	q_4 **critical section**
p_7 *readyP* := False	q_7 *readyQ* := False

Mutual exclusion is violated if they execute in lock-step (i.e. $p_1q_1p_2q_2p_3q_3$ etc.)

Third Attempt

var *readyP*, *readyQ* := False, False	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 *readyP* := True;	q_2 *readyQ* := True;
p_3 **await** *readyQ* = False;	q_3 **await** *readyP* = False;
p_4 **critical section**	q_4 **critical section**
p_7 *readyP* := False	q_7 *readyQ* := False

Third Attempt

var *readyP*, *readyQ* := False, False	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 *readyP* := True;	q_2 *readyQ* := True;
p_3 **await** *readyQ* = False;	q_3 **await** *readyP* = False;
p_4 **critical section**	q_4 **critical section**
p_7 *readyP* := False	q_7 *readyQ* := False

Now we have a deadlock (or stuck state) if they proceed in lock step

Peterson's Algorithm

var $readyP, readyQ :=$ False, False; **var** $turn := P$	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 $readyP :=$ True	q_2 $readyQ :=$ True
p_3 $turn := Q$	q_3 $turn := P$
p_4 **await** $readyQ =$ False $\vee$ $turn = P$	q_4 **await** $readyP =$ False $\vee$ $turn = Q$
p_5 **critical section**	q_5 **critical section**
p_6 $readyP :=$ False	q_6 $readyQ :=$ False

Peterson's Algorithm — Proof of Mutex Property

var *readyP*, *readyQ* := False, False; **var** *turn* := *P*	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 *readyP* := True	q_2 *readyQ* := True
p_3 *turn* := *Q*	q_3 *turn* := *P*
p_4 **await** $readyQ = \text{False} \lor turn = P$	q_4 **await** $readyP = \text{False} \lor turn = Q$
p_5 **critical section**	q_5 **critical section**
p_6 *readyP* := False	q_6 *readyQ* := False

Peterson's Algorithm — Proof of Starvation Freedom

var *readyP*, *readyQ* := False, False; **var** *turn* := *P*	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 *readyP* := True	q_2 *readyQ* := True
p_3 *turn* := *Q*	q_3 *turn* := *P*
p_4 **await** *readyQ* = False ∨ *turn* = *P*	q_4 **await** *readyP* = False ∨ *turn* = *Q*
p_5 **critical section**	q_5 **critical section**
p_6 *readyP* := False	q_6 *readyQ* := False

Speed

Can we make a good mutual exclusion protocol without any communication between the two processes?

Speed

Can we make a good mutual exclusion protocol without any communication between the two processes?

Yes!

Speed

Can we make a good mutual exclusion protocol without any communication between the two processes?

Yes!

Consider 7 computers. In advance, one of them agrees to enter the critical section only on Mondays, another only on Tuesdays, and so on.

Speed

Can we make a good mutual exclusion protocol without any communication between the two processes?

Yes!

Consider 7 computers. In advance, one of them agrees to enter the critical section only on Mondays, another only on Tuesdays, and so on.

An extra criterion: speed-independence.
nothing may be assumed about the relative speeds of the processes

Speed

Can we make a good mutual exclusion protocol without any communication between the two processes?

Yes!

Consider 7 computers. In advance, one of them agrees to enter the critical section only on Mondays, another only on Tuesdays, and so on.

An extra criterion: speed-independence.
nothing may be assumed about the relative speeds of the processes

Equivalently:
there can be arbitrary long pauses between any two instructions.

Speed

Can we make a good mutual exclusion protocol without any communication between the two processes?

Yes!

Consider 7 computers. In advance, one of them agrees to enter the critical section only on Mondays, another only on Tuesdays, and so on.

An extra criterion: speed-independence.
nothing may be assumed about the relative speeds of the processes

Equivalently:
there can be arbitrary long pauses between any two instructions.

Importantly: even if one process in a race is ready to step over the finish, the other may still win.

Shared variables: 6 register models

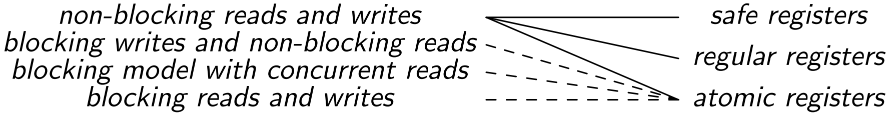
The processes in a mutex protocol communicate with each other by writing to and reading from shared registers.

Shared variables: 6 register models

The processes in a mutex protocol communicate with each other by writing to and reading from shared registers. Whether a mutex protocol is correct depends on the way these registers work.

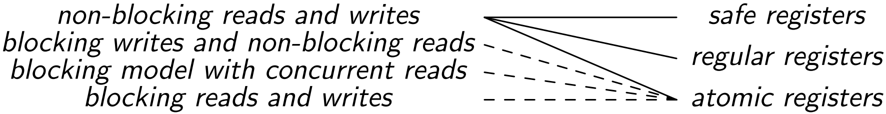
Shared variables: 6 register models

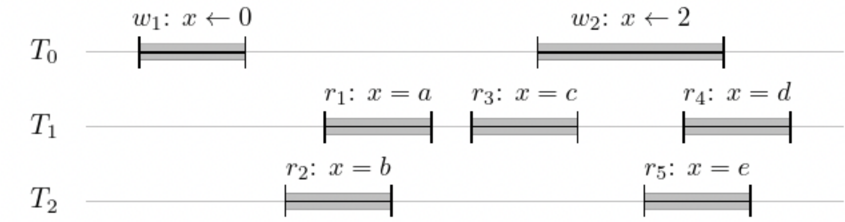
The processes in a mutex protocol communicate with each other by writing to and reading from shared registers. Whether a mutex protocol is correct depends on the way these registers work.



Shared variables: 6 register models

The processes in a mutex protocol communicate with each other by writing to and reading from shared registers. Whether a mutex protocol is correct depends on the way these registers work.





Shared variables: 4 register models

register model	*rd* block *rd*?	*rd* block *wr*?	*wr* block *rd*?	*wr* block *wr*?
blocking reads and writes	$\surd$	$\surd$	$\surd$	$\surd$
blocking model with concurrent reads	—	$\surd$	$\surd$	$\surd$
blocking writes and non-blocking reads	—	—	$\surd$	$\surd$
non-blocking reads and writes	—	—	—	—

Shared variables: 4 register models

register model	*rd* block *rd*?	*rd* block *wr*?	*wr* block *rd*?	*wr* block *wr*?
blocking reads and writes	√	√	√	√
blocking model with concurrent reads	—	√	√	√
blocking writes and non-blocking reads	—	—	√	√
non-blocking reads and writes	—	—	—	—

mutual exclusion impossible

with red registers.

Shared variables: 4 register models

register model	*rd* block *rd* ?	*rd* block *wr* ?	*wr* block *rd* ?	*wr* block *wr* ?
blocking reads and writes	√	√	√	√
blocking model with concurrent reads	–	√	√	√
blocking writes and non-blocking reads	–	–	√	√
non-blocking reads and writes	–	–	–	–

speed-independent mutual exclusion impossible

with red registers.

Shared variables: 4 register models

register model	*rd* block *rd*?	*rd* block *wr*?	*wr* block *rd*?	*wr* block *wr*?
blocking reads and writes	√	√	√	√
blocking model with concurrent reads	—	√	√	√
blocking writes and non-blocking reads	—	—	√	√
non-blocking reads and writes	—	—	—	—

Deadlock-free speed-independent mutual exclusion impossible with red registers.

Read/write overlap: the more challenging assumption

Early work on mutual exclusion did not consider read/write overlap.

Read/write overlap: the more challenging assumption

Early work on mutual exclusion did not consider read/write overlap.
Hence each read returns the value of the last write.

Read/write overlap: the more challenging assumption

Early work on mutual exclusion did not consider read/write overlap.
Hence each read returns the value of the last write.

Lamport'74 *does* allow read/write overlap.

Read/write overlap: the more challenging assumption

Early work on mutual exclusion did not consider read/write overlap.
Hence each read returns the value of the last write.

Lamport'74 *does* allow read/write overlap.
Hence also spurious values may be read.

Read/write overlap: the more challenging assumption

Early work on mutual exclusion did not consider read/write overlap.
Hence each read returns the value of the last write.

Lamport'74 *does* allow read/write overlap.
Hence also spurious values may be read.
He implies that this makes the problem harder.

Read/write overlap: the more challenging assumption

Early work on mutual exclusion did not consider read/write overlap.
Hence each read returns the value of the last write.

Lamport'74 *does* allow read/write overlap.
Hence also spurious values may be read.
He implies that this makes the problem harder.
Even so, he shows that his Bakery protocol work's perfectly well.

Read/write overlap: the more challenging assumption

Early work on mutual exclusion did not consider read/write overlap.
Hence each read returns the value of the last write.

Lamport'74 *does* allow read/write overlap.
Hence also spurious values may be read.
He implies that this makes the problem harder.
Even so, he shows that his Bakery protocol work's perfectly well.
Moreover, the Bakery protocol is speed independent.

Read/write non-overlap: the more challenging assumption

Early work on mutual exclusion did not consider read/write overlap.
Hence each read returns the value of the last write.

Lamport'74 *does* allow read/write overlap.
Hence also spurious values may be read.
He implies that this makes the problem harder.
Even so, he shows that his Bakery protocol work's perfectly well.
Moreover, the Bakery protocol is speed independent.

My claim is that atomicity is the more challenging assumption.

Read/write non-overlap: the more challenging assumption

Early work on mutual exclusion did not consider read/write overlap.
Hence each read returns the value of the last write.

Lamport'74 *does* allow read/write overlap.
Hence also spurious values may be read.
He implies that this makes the problem harder.
Even so, he shows that his Bakery protocol work's perfectly well.
Moreover, the Bakery protocol is speed independent.

My claim is that atomicity is the more challenging assumption.
For then there is no speed-independent mutual exclusion protocol.

No mutex with read/write on blocking registers

$$x := v \qquad \| \qquad \textbf{await}(x = v)$$

Peterson's Algorithm

var $readyP, readyQ :=$ False, False; **var** $turn := P$	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 $readyP :=$ True	q_2 $readyQ :=$ True
p_3 $turn := Q$	q_3 $turn := P$
p_4 **await** $readyQ =$ False $\vee$ $turn = P$	q_4 **await** $readyP =$ False $\vee$ $turn = Q$
p_5 **critical section**	q_5 **critical section**
p_6 $readyP :=$ False	q_6 $readyQ :=$ False

Peterson's algorithm without busy wait

repeat forever

$$\left\{\begin{array}{ll}
\ell_1 & \textbf{noncritical section} \\
\ell_2 & \mathit{ready}[i] := \mathit{true} \;\oplus\; \textsc{send}(\mathit{ready}[i] = \mathit{true}) \\
\ell_3 & \mathit{turn} := j \;\oplus\; \textsc{send}(\mathit{turn} = j) \\
\ell_4 & \textbf{read}(\mathit{ready}[j] = \mathit{false}) \;\parr\; \textsc{receive}(\mathit{ready}[j] = \mathit{false}) \;\parr \\
 & \textbf{read}(\mathit{turn} = i) \;\parr\; \textsc{receive}(\mathit{turn} = i) \\
\ell_5 & \textbf{critical section} \\
\ell_6 & \mathit{ready}[i] := \mathit{false} \;\oplus\; \textsc{send}(\mathit{ready}[i] = \mathit{false})
\end{array}\right.$$

Here i is the displayed process and j the other one.

Peterson's algorithm without busy wait

repeat forever

ℓ_1 **noncritical section**
ℓ_2 $\mathit{ready}[i] := \mathit{true} \;\oplus\; \textsc{send}(\mathit{ready}[i] = \mathit{true})$
ℓ_3 $\mathit{turn} := j \;\oplus\; \textsc{send}(\mathit{turn} = j)$
ℓ_4 $\mathbf{read}(\mathit{ready}[j] = \mathit{false}) \;\parr\; \textsc{receive}(\mathit{ready}[j] = \mathit{false}) \;\parr$
$\mathbf{read}(\mathit{turn} = i) \;\parr\; \textsc{receive}(\mathit{turn} = i)$
ℓ_5 **critical section**
ℓ_6 $\mathit{ready}[i] := \mathit{false} \;\oplus\; \textsc{send}(\mathit{ready}[i] = \mathit{false})$

Here i is the displayed process and j the other one.

This is a deadlock-free speed-independent mutual exclusion algorithm.

Impossibility of speed ind. mutual excl. under atomicity

For mutual exclusion to be possible, there must be a variable *readyP* that is written by Proc. P to request entry to CS.

Impossibility of speed ind. mutual excl. under atomicity

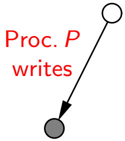
For mutual exclusion to be possible, there must be a variable *readyP* that is written by Proc. P to request entry to CS.

readyP must be read by Proc. Q, before Proc. Q can enter CS.

Impossibility of speed ind. mutual excl. under atomicity

For mutual exclusion to be possible, there must be a variable *readyP* that is written by Proc. P to request entry to CS.

readyP must be read by Proc. Q, before Proc. Q can enter CS.

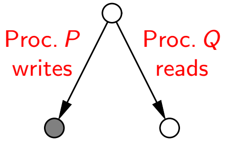


It suffices to present a scenario where Proc. P is ready to write to *readyP* yet never succeeds in doing so.

Impossibility of speed ind. mutual excl. under atomicity

For mutual exclusion to be possible, there must be a variable *readyP* that is written by Proc. P to request entry to CS.

readyP must be read by Proc. Q, before Proc. Q can enter CS.

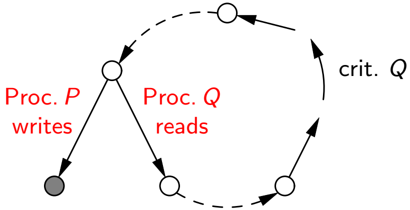


It suffices to present a scenario where Proc. P is ready to write to *readyP* yet never succeeds in doing so.

Impossibility of speed ind. mutual excl. under atomicity

For mutual exclusion to be possible, there must be a variable *readyP* that is written by Proc. *P* to request entry to CS.

readyP must be read by Proc. *Q*, before Proc. *Q* can enter CS.



It suffices to present a scenario where Proc. *P* is ready to write to *readyP* yet never succeeds in doing so.

Peterson's Algorithm — Proof of Starvation Freedom

var *readyP*, *readyQ* := False, False; **var** *turn* := *P*	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 *readyP* := True	q_2 *readyQ* := True
p_3 *turn* := *Q*	q_3 *turn* := *P*
p_4 **await** *readyQ* = False ∨ *turn* = *P*	q_4 **await** *readyP* = False ∨ *turn* = *Q*
p_5 **critical section**	q_5 **critical section**
p_6 *readyP* := False	q_6 *readyQ* := False

Peterson's algorithm — counterexample to starv. Freedom

var $readyP, readyQ :=$ False, False; var $turn := P$	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 $readyP :=$ True	q_2 $readyQ :=$ True
p_3 $turn := Q$	q_3 $turn := P$
p_4 **await** $readyQ =$ False $\vee$ $turn = P$	q_4 **await** $readyP =$ False $\vee$ $turn = Q$
p_5 **critical section**	q_5 **critical section**
p_6 $readyP :=$ False	q_6 $readyQ :=$ False

A classification of message passing mechanisms

asynchronous communication		$\rightarrow$		**synchronous communication**
one recipient		$<$		**multiple recipients**
interrupting receipt		vs.		**dedicated receipt**
non-blockable sending	$<$	**partly blockable**	$>$	blockable sending
choice-free	$<$	input-guarded choice	$<$	**mixed choice**
instantaneous choice		vs.		**gradual choice**
interruptible		vs.		**uninterruptible**

A classification of message passing mechanisms

asynchronous communication		$\rightarrow$		**synchronous communication**
one recipient		$<$		**multiple recipients**
interrupting receipt		vs.		**dedicated receipt**
non-blockable sending	$<$	**partly blockable**	$>$	blockable sending
choice-free	$<$	input-guarded choice	$<$	**mixed choice**
instantaneous choice		vs.		**gradual choice**
interruptible		vs.		**uninterruptible**

Starvation-free speed-independent mutual exclusion is possible with

- interrupting receipt
- interruptible communication
- broadcast communication with instantaneous choice

but not with any other message passing paradigm classified here.