Migrating Peterson Algorithm from SC to RA Memory Order

Computer Systems

Johann Blieberger

1. Peterson Algorithm
2. Migration of Peterson Algorithm to Memory Order Release-Acquire

Peterson Algorithm

Peterson Algorithm

- Peterson is a simple algorithm for synchronizing two threads in a non-blocking way

Peterson Algorithm

- Peterson is a simple algorithm for synchronizing two threads in a non-blocking way
- first software-only solution to synchronization from 1981

Peterson Algorithm

- Peterson is a simple algorithm for synchronizing two threads in a non-blocking way
- first software-only solution to synchronization from 1981
- known to be correct for memory order SC

Peterson Algorithm

- Peterson is a simple algorithm for synchronizing two threads in a non-blocking way
- first software-only solution to synchronization from 1981
- known to be correct for memory order SC
- named after its inventor Gary L. Peterson

Peterson Algorithm

Variable `v` is initially set to `0`, the Booleans `i0` and `i1` are initially set to `false`.

	abbr.	R/A	Thread P_0	abbr.	R/A	Thread P_1
1	i0t	R	`i0 := true`	i1t	R	`i1 := true`
2	v0	R	`v := 0`	v1	R	`v := 1`
3	i1r	A	`0: i1'READ`	i0r	A	`1: i0'READ`
4	vr	A	`v'READ`	vr	A	`v'READ`
5	if0		`if i1 and v=0 then goto 0`	if1		`if i0 and v=1 then goto 1`
6	s0	?	`...`	s1	?	`...`
7	i0f	R	`i0 := false`	i1f	R	`i1 := false`

Possible execution:

Peterson Algorithm

Variable `v` is initially set to `0`, the Booleans `i0` and `i1` are initially set to `false`.

	abbr.	R/A	Thread P_0	abbr.	R/A	Thread P_1
1	i0t	R	`i0 := true`	i1t	R	`i1 := true`
2	v0	R	`v := 0`	v1	R	`v := 1`
3	i1r	A	`0: i1'READ`	i0r	A	`1: i0'READ`
4	vr	A	`v'READ`	vr	A	`v'READ`
5	if0		`if i1 and v=0 then goto 0`	if1		`if i0 and v=1 then goto 1`
6	s0	?	`...`	s1	?	`...`
7	i0f	R	`i0 := false`	i1f	R	`i1 := false`

Possible execution:

$(\text{i0}, \text{i1}, v) = (f, f, 0) \xrightarrow{\text{i0t}} (\underline{t}, f, 0)$

Peterson Algorithm

Variable `v` is initially set to `0`, the Booleans `i0` and `i1` are initially set to `false`.

	abbr.	R/A	Thread P_0	abbr.	R/A	Thread P_1
1	i0t	R	`i0 := true`	i1t	R	`i1 := true`
2	v0	R	`v := 0`	v1	R	`v := 1`
3	i1r	A	`0: i1'READ`	i0r	A	`1: i0'READ`
4	vr	A	`v'READ`	vr	A	`v'READ`
5	if0		`if i1 and v=0 then goto 0`	if1		`if i0 and v=1 then goto 1`
6	s0	?	`...`	s1	?	`...`
7	i0f	R	`i0 := false`	i1f	R	`i1 := false`

Possible execution:

$(\mathrm{i0}, \mathrm{i1}, v) = (f, f, 0) \xrightarrow{\mathrm{i0t}} (\underline{t}, f, 0) \xrightarrow{\mathrm{v0}} (t, f, \underline{0})$

Peterson Algorithm

Variable `v` is initially set to `0`, the Booleans `i0` and `i1` are initially set to `false`.

	abbr.	R/A	Thread P_0	abbr.	R/A	Thread P_1
1	i0t	R	`i0 := true`	i1t	R	`i1 := true`
2	v0	R	`v := 0`	v1	R	`v := 1`
3	i1r	A	`0: i1'READ`	i0r	A	`1: i0'READ`
4	vr	A	`v'READ`	vr	A	`v'READ`
5	if0		`if i1 and v=0 then goto 0`	if1		`if i0 and v=1 then goto 1`
6	s0	?	`...`	s1	?	`...`
7	i0f	R	`i0 := false`	i1f	R	`i1 := false`

Possible execution:

$(\mathrm{i0}, \mathrm{i1}, v) = (f, f, 0) \xrightarrow{\mathrm{i0t}} (\underline{t}, f, 0) \xrightarrow{\mathrm{v0}} (t, f, \underline{0}) \xrightarrow{\mathrm{i1t}} (t, \underline{t}, 0)$

AUTOMATION SYSTEMS GROUP

Peterson Algorithm

Variable `v` is initially set to `0`, the Booleans `i0` and `i1` are initially set to `false`.

	abbr.	R/A	Thread P_0	abbr.	R/A	Thread P_1
1	i0t	R	`i0 := true`	i1t	R	`i1 := true`
2	v0	R	`v := 0`	v1	R	`v := 1`
3	i1r	A	`0: i1'READ`	i0r	A	`1: i0'READ`
4	vr	A	`v'READ`	vr	A	`v'READ`
5	if0		`if i1 and v=0 then goto 0`	if1		`if i0 and v=1 then goto 1`
6	s0	?	`...`	s1	?	`...`
7	i0f	R	`i0 := false`	i1f	R	`i1 := false`

Possible execution:

$(\mathrm{i0}, \mathrm{i1}, v) = (f, f, 0) \xrightarrow{\mathrm{i0t}} (\underline{t}, f, 0) \xrightarrow{\mathrm{v0}} (t, f, \underline{0}) \xrightarrow{\mathrm{i1t}} (t, \underline{t}, 0) \xrightarrow{\mathrm{v1}} (t, t, \underline{1})$

Peterson Algorithm

Variable `v` is initially set to `0`, the Booleans `i0` and `i1` are initially set to `false`.

	abbr.	R/A	Thread P_0	abbr.	R/A	Thread P_1
1	i0t	R	`i0 := true`	i1t	R	`i1 := true`
2	v0	R	`v := 0`	v1	R	`v := 1`
3	i1r	A	`0: i1'READ`	i0r	A	`1: i0'READ`
4	vr	A	`v'READ`	vr	A	`v'READ`
5	if0		`if i1 and v=0 then goto 0`	if1		`if i0 and v=1 then goto 1`
6	s0	?	`...`	s1	?	`...`
7	i0f	R	`i0 := false`	i1f	R	`i1 := false`

Possible execution:

$(\mathrm{i0}, \mathrm{i1}, v) = (f, f, 0) \xrightarrow{\mathrm{i0t}} (\underline{t}, f, 0) \xrightarrow{\mathrm{v0}} (t, f, \underline{0}) \xrightarrow{\mathrm{i1t}} (t, \underline{t}, 0) \xrightarrow{\mathrm{v1}} (t, t, \underline{1}) \xrightarrow{\mathrm{i1r}} (t, t, 1)$

Peterson Algorithm

Variable `v` is initially set to `0`, the Booleans `i0` and `i1` are initially set to `false`.

	abbr.	R/A	Thread P_0	abbr.	R/A	Thread P_1
1	i0t	R	`i0 := true`	i1t	R	`i1 := true`
2	v0	R	`v := 0`	v1	R	`v := 1`
3	i1r	A	`0: i1'READ`	i0r	A	`1: i0'READ`
4	vr	A	`v'READ`	vr	A	`v'READ`
5	if0		`if i1 and v=0 then goto 0`	if1		`if i0 and v=1 then goto 1`
6	s0	?	`...`	s1	?	`...`
7	i0f	R	`i0 := false`	i1f	R	`i1 := false`

Possible execution:

$$(\text{i0}, \text{i1}, v) = (f, f, 0) \xrightarrow{\text{i0t}} (\underline{t}, f, 0) \xrightarrow{\text{v0}} (t, f, \underline{0}) \xrightarrow{\text{i1t}} (t, \underline{t}, 0) \xrightarrow{\text{v1}} (t, t, \underline{1}) \xrightarrow{\text{i1r}} (t, t, 1) \xrightarrow{\text{vr}} (t, t, 1)$$

Peterson Algorithm

Variable `v` is initially set to `0`, the Booleans `i0` and `i1` are initially set to `false`.

	abbr.	R/A	Thread P_0	abbr.	R/A	Thread P_1
1	i0t	R	`i0 := true`	i1t	R	`i1 := true`
2	v0	R	`v := 0`	v1	R	`v := 1`
3	i1r	A	`0: i1'READ`	i0r	A	`1: i0'READ`
4	vr	A	`v'READ`	vr	A	`v'READ`
5	if0		`if i1 and v=0 then goto 0`	if1		`if i0 and v=1 then goto 1`
6	s0	?	`...`	s1	?	`...`
7	i0f	R	`i0 := false`	i1f	R	`i1 := false`

Possible execution:

$(\text{i0}, \text{i1}, v) = (f, f, 0) \xrightarrow{\text{i0t}} (\underline{t}, f, 0) \xrightarrow{\text{v0}} (t, f, \underline{0}) \xrightarrow{\text{i1t}} (t, \underline{t}, 0) \xrightarrow{\text{v1}} (t, t, \underline{1}) \xrightarrow{\text{i1r}} (t, t, 1) \xrightarrow{\text{vr}} (t, t, 1) \xrightarrow{\text{i0r}} (t, t, 1)$

Peterson Algorithm

Variable `v` is initially set to `0`, the Booleans `i0` and `i1` are initially set to `false`.

	abbr.	R/A	Thread P_0	abbr.	R/A	Thread P_1
1	i0t	R	`i0 := true`	i1t	R	`i1 := true`
2	v0	R	`v := 0`	v1	R	`v := 1`
3	i1r	A	`0: i1'READ`	i0r	A	`1: i0'READ`
4	vr	A	`v'READ`	vr	A	`v'READ`
5	if0		`if i1 and v=0 then goto 0`	if1		`if i0 and v=1 then goto 1`
6	s0	?	`...`	s1	?	`...`
7	i0f	R	`i0 := false`	i1f	R	`i1 := false`

Possible execution:

$(\text{i0}, \text{i1}, v) = (f, f, 0) \xrightarrow{\text{i0t}} (\underline{t}, f, 0) \xrightarrow{\text{v0}} (t, f, \underline{0}) \xrightarrow{\text{i1t}} (t, \underline{t}, 0) \xrightarrow{\text{v1}} (t, t, \underline{1}) \xrightarrow{\text{i1r}} (t, t, 1) \xrightarrow{\text{vr}} (t, t, 1) \xrightarrow{\text{i0r}}$
$(t, t, 1) \xrightarrow{\text{vr}} (t, t, 1)$

Peterson Algorithm

Variable `v` is initially set to `0`, the Booleans `i0` and `i1` are initially set to `false`.

	abbr.	R/A	Thread P_0	abbr.	R/A	Thread P_1
1	i0t	R	`i0 := true`	i1t	R	`i1 := true`
2	v0	R	`v := 0`	v1	R	`v := 1`
3	i1r	A	`0: i1'READ`	i0r	A	`1: i0'READ`
4	vr	A	`v'READ`	vr	A	`v'READ`
5	if0		`if i1 and v=0 then goto 0`	if1		`if i0 and v=1 then goto 1`
6	s0	?	`...`	s1	?	`...`
7	i0f	R	`i0 := false`	i1f	R	`i1 := false`

Possible execution:

$(\mathrm{i0}, \mathrm{i1}, v) = (f, f, 0) \xrightarrow{\text{i0t}} (\underline{t}, f, 0) \xrightarrow{\text{v0}} (t, f, \underline{0}) \xrightarrow{\text{i1t}} (t, \underline{t}, 0) \xrightarrow{\text{v1}} (t, t, \underline{1}) \xrightarrow{\text{i1r}} (t, t, 1) \xrightarrow{\text{vr}} (t, t, 1) \xrightarrow{\text{i0r}}$
$(t, t, 1) \xrightarrow{\text{vr}} (t, t, 1) \xrightarrow{\text{if1}} (t, t, 1)$

Peterson Algorithm

Variable `v` is initially set to `0`, the Booleans `i0` and `i1` are initially set to `false`.

	abbr.	R/A	Thread P_0	abbr.	R/A	Thread P_1
1	i0t	R	`i0 := true`	i1t	R	`i1 := true`
2	v0	R	`v := 0`	v1	R	`v := 1`
3	i1r	A	`0: i1'READ`	i0r	A	`1: i0'READ`
4	vr	A	`v'READ`	vr	A	`v'READ`
5	if0		`if i1 and v=0 then goto 0`	if1		`if i0 and v=1 then goto 1`
6	s0	?	`...`	s1	?	`...`
7	i0f	R	`i0 := false`	i1f	R	`i1 := false`

Possible execution:

$$(\mathrm{i0}, \mathrm{i1}, v) = (f, f, 0) \xrightarrow{\text{i0t}} (\underline{t}, f, 0) \xrightarrow{\text{v0}} (t, f, \underline{0}) \xrightarrow{\text{i1t}} (t, \underline{t}, 0) \xrightarrow{\text{v1}} (t, t, \underline{1}) \xrightarrow{\text{i1r}} (t, t, 1) \xrightarrow{\text{vr}} (t, t, 1) \xrightarrow{\text{i0r}} (t, t, 1) \xrightarrow{\text{vr}} (t, t, 1) \xrightarrow{\text{if1}} (t, t, 1) \xrightarrow{\text{if0}} (t, t, 1)$$

Peterson Algorithm

Variable `v` is initially set to `0`, the Booleans `i0` and `i1` are initially set to `false`.

	abbr.	R/A	Thread P_0	abbr.	R/A	Thread P_1
1	i0t	R	`i0 := true`	i1t	R	`i1 := true`
2	v0	R	`v := 0`	v1	R	`v := 1`
3	i1r	A	`0: i1'READ`	i0r	A	`1: i0'READ`
4	vr	A	`v'READ`	vr	A	`v'READ`
5	if0		`if i1 and v=0 then goto 0`	if1		`if i0 and v=1 then goto 1`
6	s0	?	`...`	s1	?	`...`
7	i0f	R	`i0 := false`	i1f	R	`i1 := false`

Possible execution:

$(\text{i0}, \text{i1}, v) = (f, f, 0) \xrightarrow{\text{i0t}} (\underline{t}, f, 0) \xrightarrow{\text{v0}} (t, f, \underline{0}) \xrightarrow{\text{i1t}} (t, \underline{t}, 0) \xrightarrow{\text{v1}} (t, t, \underline{1}) \xrightarrow{\text{i1r}} (t, t, 1) \xrightarrow{\text{vr}} (t, t, 1) \xrightarrow{\text{i0r}}$
$(t, t, 1) \xrightarrow{\text{vr}} (t, t, 1) \xrightarrow{\text{if1}} (t, t, 1) \xrightarrow{\text{if0}} (t, t, 1) \xrightarrow{\text{s0}} (t, t, 1)$

Peterson Algorithm

Variable `v` is initially set to `0`, the Booleans `i0` and `i1` are initially set to `false`.

	abbr.	R/A	Thread P_0	abbr.	R/A	Thread P_1
1	i0t	R	`i0 := true`	i1t	R	`i1 := true`
2	v0	R	`v := 0`	v1	R	`v := 1`
3	i1r	A	`0: i1'READ`	i0r	A	`1: i0'READ`
4	vr	A	`v'READ`	vr	A	`v'READ`
5	if0		`if i1 and v=0 then goto 0`	if1		`if i0 and v=1 then goto 1`
6	s0	?	`...`	s1	?	`...`
7	i0f	R	`i0 := false`	i1f	R	`i1 := false`

Possible execution:

$$(\mathrm{i0}, \mathrm{i1}, v) = (f, f, 0) \xrightarrow{\mathrm{i0t}} (\underline{t}, f, 0) \xrightarrow{\mathrm{v0}} (t, f, \underline{0}) \xrightarrow{\mathrm{i1t}} (t, \underline{t}, 0) \xrightarrow{\mathrm{v1}} (t, t, \underline{1}) \xrightarrow{\mathrm{i1r}} (t, t, 1) \xrightarrow{\mathrm{vr}} (t, t, 1) \xrightarrow{\mathrm{i0r}}$$
$$(t, t, 1) \xrightarrow{\mathrm{vr}} (t, t, 1) \xrightarrow{\mathrm{if1}} (t, t, 1) \xrightarrow{\mathrm{if0}} (t, t, 1) \xrightarrow{\mathrm{s0}} (t, t, 1) \xrightarrow{\mathrm{i0f}} (\underline{f}, t, 1)$$

Peterson Algorithm

Variable `v` is initially set to `0`, the Booleans `i0` and `i1` are initially set to `false`.

	abbr.	R/A	Thread P_0	abbr.	R/A	Thread P_1
1	i0t	R	`i0 := true`	i1t	R	`i1 := true`
2	v0	R	`v := 0`	v1	R	`v := 1`
3	i1r	A	`0: i1'READ`	i0r	A	`1: i0'READ`
4	vr	A	`v'READ`	vr	A	`v'READ`
5	if0		`if i1 and v=0 then goto 0`	if1		`if i0 and v=1 then goto 1`
6	s0	?	`...`	s1	?	`...`
7	i0f	R	`i0 := false`	i1f	R	`i1 := false`

Possible execution:

$$(\text{i0}, \text{i1}, v) = (f, f, 0) \xrightarrow{\text{i0t}} (\underline{t}, f, 0) \xrightarrow{\text{v0}} (t, f, \underline{0}) \xrightarrow{\text{i1t}} (t, \underline{t}, 0) \xrightarrow{\text{v1}} (t, t, \underline{1}) \xrightarrow{\text{i1r}} (t, t, 1) \xrightarrow{\text{vr}} (t, t, 1) \xrightarrow{\text{i0r}}$$

$$(t, t, 1) \xrightarrow{\text{vr}} (t, t, 1) \xrightarrow{\text{if1}} (t, t, 1) \xrightarrow{\text{if0}} (t, t, 1) \xrightarrow{\text{s0}} (t, t, 1) \xrightarrow{\text{i0f}} (\underline{f}, t, 1) \xrightarrow{\text{i0r}} (f, t, 1)$$

Peterson Algorithm

Variable `v` is initially set to `0`, the Booleans `i0` and `i1` are initially set to `false`.

	abbr.	R/A	Thread P_0	abbr.	R/A	Thread P_1
1	i0t	R	`i0 := true`	i1t	R	`i1 := true`
2	v0	R	`v := 0`	v1	R	`v := 1`
3	i1r	A	`0: i1'READ`	i0r	A	`1: i0'READ`
4	vr	A	`v'READ`	vr	A	`v'READ`
5	if0		`if i1 and v=0 then goto 0`	if1		`if i0 and v=1 then goto 1`
6	s0	?	`...`	s1	?	`...`
7	i0f	R	`i0 := false`	i1f	R	`i1 := false`

Possible execution:

$$(\mathrm{i0}, \mathrm{i1}, v) = (f, f, 0) \xrightarrow{\mathrm{i0t}} (\underline{t}, f, 0) \xrightarrow{\mathrm{v0}} (t, f, \underline{0}) \xrightarrow{\mathrm{i1t}} (t, \underline{t}, 0) \xrightarrow{\mathrm{v1}} (t, t, \underline{1}) \xrightarrow{\mathrm{i1r}} (t, t, 1) \xrightarrow{\mathrm{vr}} (t, t, 1) \xrightarrow{\mathrm{i0r}}$$

$$(t, t, 1) \xrightarrow{\mathrm{vr}} (t, t, 1) \xrightarrow{\mathrm{if1}} (t, t, 1) \xrightarrow{\mathrm{if0}} (t, t, 1) \xrightarrow{\mathrm{s0}} (t, t, 1) \xrightarrow{\mathrm{i0f}} (\underline{f}, t, 1) \xrightarrow{\mathrm{i0r}} (f, t, 1) \xrightarrow{\mathrm{vr}} (f, t, 1)$$

Peterson Algorithm

Variable `v` is initially set to `0`, the Booleans `i0` and `i1` are initially set to `false`.

	abbr.	R/A	Thread P_0	abbr.	R/A	Thread P_1
1	i0t	R	`i0 := true`	i1t	R	`i1 := true`
2	v0	R	`v := 0`	v1	R	`v := 1`
3	i1r	A	`0: i1'READ`	i0r	A	`1: i0'READ`
4	vr	A	`v'READ`	vr	A	`v'READ`
5	if0		`if i1 and v=0 then goto 0`	if1		`if i0 and v=1 then goto 1`
6	s0	?	`...`	s1	?	`...`
7	i0f	R	`i0 := false`	i1f	R	`i1 := false`

Possible execution:

$$(\mathrm{i0}, \mathrm{i1}, v) = (f, f, 0) \xrightarrow{\text{i0t}} (\underline{t}, f, 0) \xrightarrow{\text{v0}} (t, f, \underline{0}) \xrightarrow{\text{i1t}} (t, \underline{t}, 0) \xrightarrow{\text{v1}} (t, t, \underline{1}) \xrightarrow{\text{i1r}} (t, t, 1) \xrightarrow{\text{vr}} (t, t, 1) \xrightarrow{\text{i0r}}$$
$$(t, t, 1) \xrightarrow{\text{vr}} (t, t, 1) \xrightarrow{\text{if1}} (t, t, 1) \xrightarrow{\text{if0}} (t, t, 1) \xrightarrow{\text{s0}} (t, t, 1) \xrightarrow{\text{i0f}} (\underline{f}, t, 1) \xrightarrow{\text{i0r}} (f, t, 1) \xrightarrow{\text{vr}} (f, t, 1) \xrightarrow{\text{if1}}$$
$$(f, t, 1)$$

Peterson Algorithm

Variable `v` is initially set to `0`, the Booleans `i0` and `i1` are initially set to `false`.

	abbr.	R/A	Thread P_0	abbr.	R/A	Thread P_1
1	i0t	R	`i0 := true`	i1t	R	`i1 := true`
2	v0	R	`v := 0`	v1	R	`v := 1`
3	i1r	A	`0: i1'READ`	i0r	A	`1: i0'READ`
4	vr	A	`v'READ`	vr	A	`v'READ`
5	if0		`if i1 and v=0 then goto 0`	if1		`if i0 and v=1 then goto 1`
6	s0	?	`...`	s1	?	`...`
7	i0f	R	`i0 := false`	i1f	R	`i1 := false`

Possible execution:

$$(\mathrm{i0}, \mathrm{i1}, v) = (f, f, 0) \xrightarrow{\mathrm{i0t}} (\underline{t}, f, 0) \xrightarrow{\mathrm{v0}} (t, f, \underline{0}) \xrightarrow{\mathrm{i1t}} (t, \underline{t}, 0) \xrightarrow{\mathrm{v1}} (t, t, \underline{1}) \xrightarrow{\mathrm{i1r}} (t, t, 1) \xrightarrow{\mathrm{vr}} (t, t, 1) \xrightarrow{\mathrm{i0r}}$$

$$(t, t, 1) \xrightarrow{\mathrm{vr}} (t, t, 1) \xrightarrow{\mathrm{if1}} (t, t, 1) \xrightarrow{\mathrm{if0}} (t, t, 1) \xrightarrow{\mathrm{s0}} (t, t, 1) \xrightarrow{\mathrm{i0f}} (\underline{f}, t, 1) \xrightarrow{\mathrm{i0r}} (f, t, 1) \xrightarrow{\mathrm{vr}} (f, t, 1) \xrightarrow{\mathrm{if1}}$$

$$(f, t, 1) \xrightarrow{\mathrm{s1}} (f, t, 1)$$

Peterson Algorithm

Variable `v` is initially set to `0`, the Booleans `i0` and `i1` are initially set to `false`.

	abbr.	R/A	Thread P_0	abbr.	R/A	Thread P_1
1	i0t	R	`i0 := true`	i1t	R	`i1 := true`
2	v0	R	`v := 0`	v1	R	`v := 1`
3	i1r	A	`0: i1'READ`	i0r	A	`1: i0'READ`
4	vr	A	`v'READ`	vr	A	`v'READ`
5	if0		`if i1 and v=0 then goto 0`	if1		`if i0 and v=1 then goto 1`
6	s0	?	`...`	s1	?	`...`
7	i0f	R	`i0 := false`	i1f	R	`i1 := false`

Possible execution:

$$(\mathrm{i0}, \mathrm{i1}, v) = (f, f, 0) \xrightarrow{\mathrm{i0t}} (\underline{t}, f, 0) \xrightarrow{\mathrm{v0}} (t, f, \underline{0}) \xrightarrow{\mathrm{i1t}} (t, \underline{t}, 0) \xrightarrow{\mathrm{v1}} (t, t, \underline{1}) \xrightarrow{\mathrm{i1r}} (t, t, 1) \xrightarrow{\mathrm{vr}} (t, t, 1) \xrightarrow{\mathrm{i0r}}$$

$$(t, t, 1) \xrightarrow{\mathrm{vr}} (t, t, 1) \xrightarrow{\mathrm{if1}} (t, t, 1) \xrightarrow{\mathrm{if0}} (t, t, 1) \xrightarrow{\mathrm{s0}} (t, t, 1) \xrightarrow{\mathrm{i0f}} (\underline{f}, t, 1) \xrightarrow{\mathrm{i0r}} (f, t, 1) \xrightarrow{\mathrm{vr}} (f, t, 1) \xrightarrow{\mathrm{if1}}$$

$$(f, t, 1) \xrightarrow{\mathrm{s1}} (f, t, 1) \xrightarrow{\mathrm{i1f}} (f, \underline{f}, 1)$$

AUTOMATION SYSTEMS GROUP

Interleavings Graph with SC Memory Order

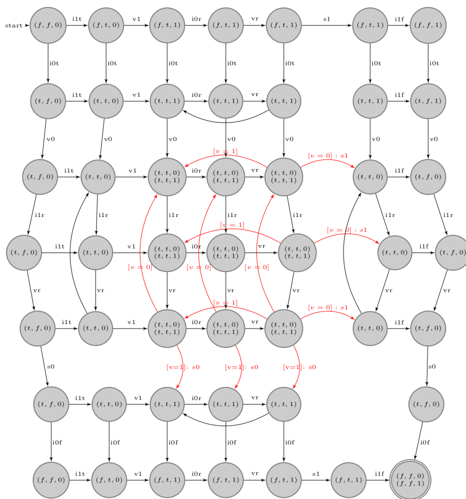

Abbildung: Interleavings Graph – Peterson with SC memory order

Peterson Algorithm

- Nodes and edges which cannot be reached in the interleavings graph have been deleted.

Peterson Algorithm

- Nodes and edges which cannot be reached in the interleavings graph have been deleted.
- Note that no reordering of statements can occur in a sequentially consistent program.

Peterson Algorithm

- Nodes and edges which cannot be reached in the interleavings graph have been deleted.
- Note that no reordering of statements can occur in a sequentially consistent program.
- Because of the structure of the graph, the critical sections s0 and s1 cannot execute at the same time.

Peterson Algorithm

- Nodes and edges which cannot be reached in the interleavings graph have been deleted.
- Note that no reordering of statements can occur in a sequentially consistent program.
- Because of the structure of the graph, the critical sections s0 and s1 cannot execute at the same time.
- For this reason, *mutual exclusion* is provided.

Peterson Algorithm

- Nodes and edges which cannot be reached in the interleavings graph have been deleted.
- Note that no reordering of statements can occur in a sequentially consistent program.
- Because of the structure of the graph, the critical sections s0 and s1 cannot execute at the same time.
- For this reason, *mutual exclusion* is provided.
- The critical point is the “hole” in the graph near the lower right corner. It ensures correct synchronization.

Peterson Algorithm

- To facilitate understanding, we notice that the nine nodes in the middle can carry both value triples $(t, t, 0)$ and $(t, t, 1)$.

Peterson Algorithm

- To facilitate understanding, we notice that the nine nodes in the middle can carry both value triples $(t, t, 0)$ and $(t, t, 1)$.
- This means, that there definitely is a *data race* at those nine nodes.

Peterson Algorithm

- To facilitate understanding, we notice that the nine nodes in the middle can carry both value triples $(t, t, 0)$ and $(t, t, 1)$.
- This means, that there definitely is a *data race* at those nine nodes.
- However, the race does no harm.

Peterson Algorithm

- To facilitate understanding, we notice that the nine nodes in the middle can carry both value triples $(t, t, 0)$ and $(t, t, 1)$.
- This means, that there definitely is a *data race* at those nine nodes.
- However, the race does no harm.
- Red edges are conditional edges; the corresponding condition [cond] is given as an edge label. If a statement is executed along a conditional edge, the label reads [cond] : stmt.

Migration of Peterson Algorithm to Memory Order Release-Acquire

Migration of Peterson Algorithm to Memory Order Release-Acquire

$\overline{x}$ denotes that the effect of statement x ist visible to the "home" core, $\underline{x}$ denotes that the effect of x is visible to all the other cores.

Migration of Peterson Algorithm to Memory Order Release-Acquire

$\overline{x}$ denotes that the effect of statement x ist visible to the "home" core, $\underline{x}$ denotes that the effect of x is visible to all the other cores.

Dependence analysis for thread P_0 results in the constraints

$\underline{i1r} < \overline{if0}$ and
$\underline{vr} < \overline{if0}$.

Migration of Peterson Algorithm to Memory Order Release-Acquire

$\overline{x}$ denotes that the effect of statement x ist visible to the “home” core, $\underline{x}$ denotes that the effect of x is visible to all the other cores.

Dependence analysis for thread P_0 results in the constraints

$\underline{i1r} < \overline{if0}$ and
$\underline{vr} < \overline{if0}$.

In addition, memory fences are derived from release and acquire operations. These are

$\underline{i0t} < \overline{v0}$,
$\underline{if0} < \overline{s0}$, and
$\underline{i0t} < \overline{i0f}$.

AUTOMATION SYSTEMS GROUP

Migration of Peterson Algorithm to Memory Order Release-Acquire

$\overline{x}$ denotes that the effect of statement x ist visible to the "home" core, $\underline{x}$ denotes that the effect of x is visible to all the other cores.

Dependence analysis for thread P_0 results in the constraints

$\underline{i1r} < \overline{if0}$ and
$\underline{vr} < \overline{if0}$.

In addition, memory fences are derived from release and acquire operations. These are

$\underline{i0t} < \overline{v0}$,
$\underline{if0} < \overline{s0}$, and
$\underline{i0t} < \overline{i0f}$.

Similar dependencies hold for thread P_1. In the following we concentrate on thread P_0. The arguments are virtually the same for thread P_1.

Migration of Peterson Algorithm to Memory Order Release-Acquire

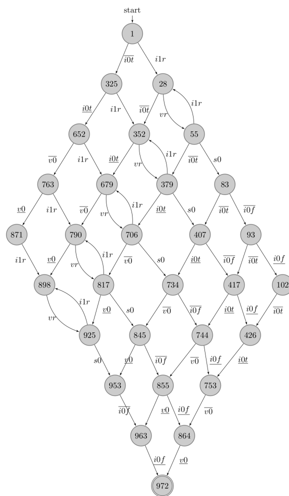

Abbildung: Peterson with RA Memory order, 1st Try (plainly wrong)

Migration of Peterson Algorithm to Memory Order Release-Acquire

- In this graph we immediately see that $ir1$ can precede $\underline{i0t}$.

Migration of Peterson Algorithm to Memory Order Release-Acquire

- In this graph we immediately see that $ir1$ can precede $\underline{i0t}$.
- A more thorough analysis shows that this may lead to both threads entering the critical section at line 6.

Migration of Peterson Algorithm to Memory Order Release-Acquire

- In this graph we immediately see that $ir1$ can precede $\underline{i0t}$.
- A more thorough analysis shows that this may lead to both threads entering the critical section at line 6.
- Thus we need additional memory fences in order to get the algorithm right with RA memory order.

Migration of Peterson Algorithm to Memory Order Release-Acquire

- To overcome this problem we introduce an additional constraint, namely $\underline{i0t} < ir1$.

Migration of Peterson Algorithm to Memory Order Release-Acquire

- To overcome this problem we introduce an additional constraint, namely $\underline{i0t} < ir1$.
- The matrix for this constraint reads

$$\begin{pmatrix} \underline{i0t} & i1r \\ . & i1r \end{pmatrix}$$

because of the loop containing an arbitrary number of $i1r$ statements.

Migration of Peterson Algorithm to Memory Order Release-Acquire

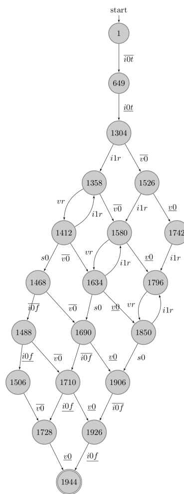

Abbildung: Peterson with RA Memory order, 2nd Try (still wrong)

Migration of Peterson Algorithm to Memory Order Release-Acquire

However, the path
$\overline{i0t} \rightarrow \underline{i0t} \rightarrow \overline{i1t} \rightarrow \underline{i1t} \rightarrow \overline{v1} \rightarrow \underline{v1} \rightarrow i1r \rightarrow vr \rightarrow \overline{v0} \rightarrow \underline{v0} \rightarrow i0r \rightarrow vr \rightarrow s0 \rightarrow s1$
in the Kronecker sum of the graph above and its P_1 variant shows that P_0 and P_1 can enter the critical section at the same time.

Thus, constraint $\underline{i0t} < ir1$ is too weak to ensure correct synchronization.

Migration of Peterson Algorithm to Memory Order Release-Acquire

Our next try is to prohibit vr from preceding $\underline{i0t}$,

The additional constraint is $\underline{v0} < i1r$.

Its matrix reads

$$\begin{pmatrix} \underline{v0} & i1r \\ . & i1r \end{pmatrix}$$

because of the loop containing an arbitrary number of $i1r$ statements.

Migration of Peterson Algorithm to Memory Order Release-Acquire

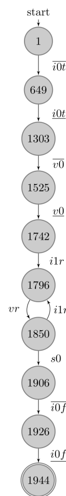

Abbildung: Peterson with RA Memory order, 3rd Try (correct, but maybe inefficient)

Migration of Peterson Algorithm to Memory Order Release-Acquire

- This version of the algorithm is correct.

Migration of Peterson Algorithm to Memory Order Release-Acquire

- This version of the algorithm is correct.
- This can be proven by applying Kronecker sum to the graph above and its P_1 variant and, further on, removing nodes and edges which cannot be reached in the resulting graph.

Migration of Peterson Algorithm to Memory Order Release-Acquire

- This version of the algorithm is correct.
- This can be proven by applying Kronecker sum to the graph above and its P_1 variant and, further on, removing nodes and edges which cannot be reached in the resulting graph.

- On the other hand, the linear graph does not allow for any instruction reordering.

Migration of Peterson Algorithm to Memory Order Release-Acquire

- This version of the algorithm is correct.
- This can be proven by applying Kronecker sum to the graph above and its P_1 variant and, further on, removing nodes and edges which cannot be reached in the resulting graph.

- On the other hand, the linear graph does not allow for any instruction reordering.
- A short reflection shows that this is too restrictive.

Migration of Peterson Algorithm to Memory Order Release-Acquire

- This version of the algorithm is correct.
- This can be proven by applying Kronecker sum to the graph above and its P_1 variant and, further on, removing nodes and edges which cannot be reached in the resulting graph.

- On the other hand, the linear graph does not allow for any instruction reordering.
- A short reflection shows that this is too restrictive.
- In detail, the constraint $\underline{v0} < i1r$ makes the constraint $\underline{i0t} < \overline{v0}$ dispensable.

Migration of Peterson Algorithm to Memory Order Release-Acquire

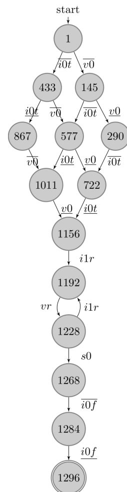

Abbildung: Peterson with RA Memory order with Relaxed instead of Release Memory Fence (still correct and maybe more efficient)

Lessons Learned

- Migrating from sequentially consistent memory order to release-acquire memory order ist not at all *straight-forward*.

Lessons Learned

- Migrating from sequentially consistent memory order to release-acquire memory order ist not at all *straight-forward*.
- It requires deep insights into the code of the algorithm and into the memory models.

Lessons Learned

- Migrating from sequentially consistent memory order to release-acquire memory order ist not at all *straight-forward*.
- It requires deep insights into the code of the algorithm and into the memory models.
- *Relaxing* is useful to gain performance.

Content inspired by

Bartosz Milewski, *Who ordered memory fences on an x86?*, https://bartoszmilewski.com/2008/11/05/who-ordered-memory-fences-on-an-x86/

Bartosz Milewski, *Who ordered sequential consistency?*, https://bartoszmilewski.com/2008/11/11/who-ordered-sequential-consistency/

Bartosz Milewski, *C++ atomics and memory ordering*, https://bartoszmilewski.com/2008/12/01/c-atomics-and-memory-ordering/