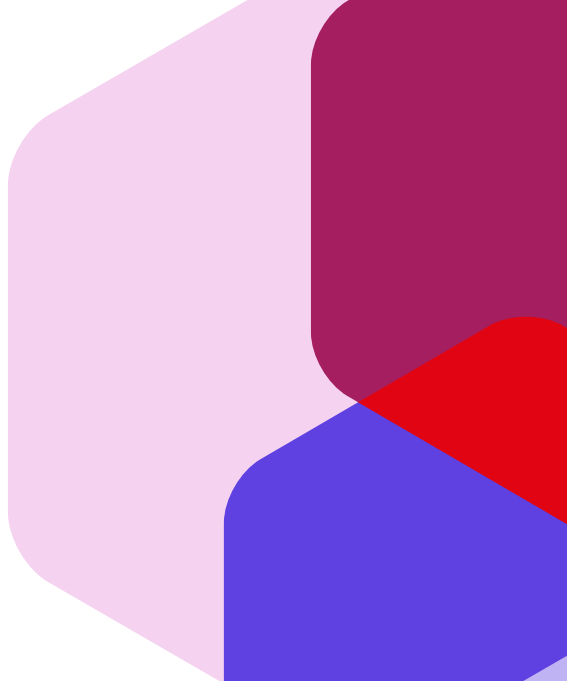


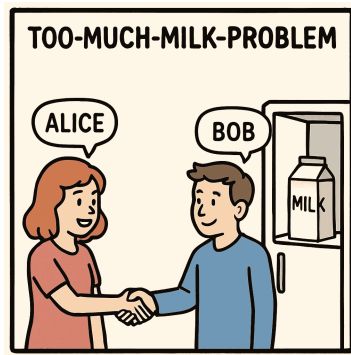
Principles, Algorithms and Models of Concurrency and Synchronization

02 Mutual Exclusion



Introductory Example: Too Much Milk

- Alice and Bob share a fridge.
- They want to make sure that there is always enough milk in the fridge.
- They don't want too much or too little milk.



Naive Solution

When someone sees that there is no milk in the fridge, they go and buy milk.

Incorrectness of Naive Solution

5:00 p.m. Alice
comes home

Bob

Incorrectness of Naive Solution

Alice

Bob

5:00 p.m. comes home

5:05 p.m. notices: no milk in the fridge

Incorrectness of Naive Solution

Alice

Bob

5:00 p.m. comes home
5:05 p.m. notices: no milk in the fridge
5:10 p.m. goes to the supermarket

Incorrectness of Naive Solution

	Alice	Bob
5:00 p.m.	comes home	
5:05 p.m.	notices: no milk in the fridge	
5:10 p.m.	goes to the supermarket	
5:15 p.m.		comes home

Incorrectness of Naive Solution

	Alice
5:00 p.m.	comes home
5:05 p.m.	notices: no milk in the fridge
5:10 p.m.	goes to the supermarket
5:15 p.m.	
5:20 p.m.	

Bob
comes home
notices: no milk in the fridge

Incorrectness of Naive Solution

	Alice
5:00 p.m.	comes home
5:05 p.m.	notices: no milk in the fridge
5:10 p.m.	goes to the supermarket
5:15 p.m.	
5:20 p.m.	
5:25 p.m.	

Bob
comes home
notices: no milk in the fridge
goes to the supermarket

Incorrectness of Naive Solution

	Alice
5:00 p.m.	comes home
5:05 p.m.	notices: no milk in the fridge
5:10 p.m.	goes to the supermarket
5:15 p.m.	
5:20 p.m.	
5:25 p.m.	
5:30 p.m.	comes back from the supermarket, puts milk in the fridge

Bob
comes home
notices: no milk in the fridge
goes to the supermarket

Incorrectness of Naive Solution

	Alice
5:00 p.m.	comes home
5:05 p.m.	notices: no milk in the fridge
5:10 p.m.	goes to the supermarket
5:15 p.m.	
5:20 p.m.	
5:25 p.m.	
5:30 p.m.	comes back from the supermarket, puts milk in the fridge
5:40 p.m.	

Bob
comes home
notices: no milk in the fridge
goes to the supermarket
comes back from the supermarket, puts milk in the fridge

Incorrectness of Naive Solution

	Alice	Bob
5:00 p.m.	comes home	
5:05 p.m.	notices: no milk in the fridge	
5:10 p.m.	goes to the supermarket	
5:15 p.m.		comes home
5:20 p.m.		notices: no milk in the fridge
5:25 p.m.		goes to the supermarket
5:30 p.m.	comes back from the supermarket, puts milk in the fridge	
5:40 p.m.		comes back from the supermarket, puts milk in the fridge

⇒ **Too much milk in the fridge**

Too Much Milk: Specification

Alice and Bob agree to leave **notes on the fridge** that both of them can read.

This is to ensure that:

- At most one of them goes to buy milk when there is no milk in the fridge.
- One of them goes to buy milk when there is no milk in the fridge.



Not allowed: Only Bob buys milk. Because: Then Alice may wait an infinite amount of time for milk (violates condition 2)

Assumption: Alice and Bob cannot see each other, they only communicate via notes.

1st Attempt

If there is no note on the fridge and no milk in the fridge, then write a note on the fridge, go buy milk, put the milk in the fridge, and then remove the note from the fridge.

As processes:

Program for Alice:

```
(A1) if no note then
(A2)   if no milk then
(A3)     write note;
(A4)     buy milk;
(A5)     remove note;
```

Program for Bob:

```
(B1) if no note then
(B2)   if no milk then
(B3)     write note;
(B4)     buy milk;
(B5)     remove note;
```

1st Attempt

If there is no note on the fridge and no milk in the fridge, then write a note on the fridge, go buy milk, put the milk in the fridge, and then remove the note from the fridge.

As processes:

Program for Alice:

```
(A1) if no note then  
(A2)   if no milk then  
(A3)     write note;  
(A4)     buy milk;  
(A5)     remove note;
```

Program for Bob:

```
(B1) if no note then  
(B2)   if no milk then  
(B3)     write note;  
(B4)     buy milk;  
(B5)     remove note;
```

1st Attempt

If there is no note on the fridge and no milk in the fridge, then write a note on the fridge, go buy milk, put the milk in the fridge, and then remove the note from the fridge.

As processes:

Program for Alice:

(A1) if no note then
(A2) if no milk then
(A3) write note;
(A4) buy milk;
(A5) remove note;

Program for Bob:

(B1) if no note then
(B2) if no milk then
(B3) write note;
(B4) buy milk;
(B5) remove note;

1st Attempt

If there is no note on the fridge and no milk in the fridge, then write a note on the fridge, go buy milk, put the milk in the fridge, and then remove the note from the fridge.

As processes:

Program for Alice:

```
(A1) if no note then  
(A2)   if no milk then  
(A3)     write note;  
(A4)     buy milk;  
(A5)     remove note;
```

Program for Bob:

```
(B1) if no note then  
(B2)   if no milk then  
(B3)     write note;  
(B4)     buy milk;  
(B5)     remove note;
```



1st Attempt

If there is no note on the fridge and no milk in the fridge, then write a note on the fridge, go buy milk, put the milk in the fridge, and then remove the note from the fridge.

As processes:

Program for Alice:

```
(A1) if no note then  
(A2)   if no milk then  
(A3)     write note;  
(A4)     buy milk;  
(A5)     remove note;
```

Program for Bob:

```
(B1) if no note then  
(B2)   if no milk then  
(B3)     write note;  
(B4)     buy milk;  
(B5)     remove note;
```

1st Attempt

If there is no note on the fridge and no milk in the fridge, then write a note on the fridge, go buy milk, put the milk in the fridge, and then remove the note from the fridge.

As processes:

Program for Alice:

```
(A1) if no note then  
(A2)   if no milk then  
(A3)     write note;  
(A4)     buy milk;  
(A5)     remove note;
```

Program for Bob:

```
(B1) if no note then  
(B2)   if no milk then  
(B3)     write note;  
(B4)     buy milk;  
(B5)     remove note;
```

Note

1st Attempt

If there is no note on the fridge and no milk in the fridge, then write a note on the fridge, go buy milk, put the milk in the fridge, and then remove the note from the fridge.

As processes:

Program for Alice:

```
(A1) if no note then  
(A2)   if no milk then  
(A3)     write note;  
(A4)     buy milk;  
(A5)     remove note;
```

Program for Bob:

```
(B1) if no note then  
(B2)   if no milk then  
(B3)     write note;  
(B4)     buy milk;  
(B5)     remove note;
```

Note



1st Attempt

If there is no note on the fridge and no milk in the fridge, then write a note on the fridge, go buy milk, put the milk in the fridge, and then remove the note from the fridge.

As processes:

Program for Alice:

```
(A1) if no note then  
(A2)   if no milk then  
(A3)     write note;  
(A4)     buy milk;  
(A5)     remove note;
```

Program for Bob:

```
(B1) if no note then  
(B2)   if no milk then  
(B3)     write note;  
(B4)     buy milk;  
(B5)     remove note;
```



1st Attempt

If there is no note on the fridge and no milk in the fridge, then write a note on the fridge, go buy milk, put the milk in the fridge, and then remove the note from the fridge.

As processes:

Program for Alice:

```
(A1) if no note then  
(A2)   if no milk then  
(A3)     write note;  
(A4)     buy milk;  
(A5)     remove note;
```

Program for Bob:

```
(B1) if no note then  
(B2)   if no milk then  
(B3)     write note;  
(B4)     buy milk;  
(B5)     remove note;
```

Note



1st Attempt

If there is no note on the fridge and no milk in the fridge, then write a note on the fridge, go buy milk, put the milk in the fridge, and then remove the note from the fridge.

As processes:

Program for Alice:

```
(A1) if no note then  
(A2)   if no milk then  
(A3)     write note;  
(A4)     buy milk;  
(A5)     remove note;
```

Program for Bob:

```
(B1) if no note then  
(B2)   if no milk then  
(B3)     write note;  
(B4)     buy milk;  
(B5)     remove note;
```

Note



1st Attempt

If there is no note on the fridge and no milk in the fridge, then write a note on the fridge, go buy milk, put the milk in the fridge, and then remove the note from the fridge.

As processes:

Program for Alice:

```
(A1) if no note then
(A2)   if no milk then
(A3)     write note;
(A4)     buy milk;
(A5)     remove note;
```

Program for Bob:

```
(B1) if no note then
(B2)   if no milk then
(B3)     write note;
(B4)     buy milk;
(B5)     remove note;
```



First, leave a note on the fridge, then check if there is a note from the other person. Only if there is no other note, check if there is milk and go buy milk if there is no milk. Then remove the note.

As processes:

Program for Alice:

```
(A1) write note "Alice";  
(A2) if no note from Bob then  
(A3)   if no milk then  
(A4)     buy milk;  
(A5) remove note "Alice";
```

Program for Bob:

```
(B1) write note "Bob";  
(B2) if no note from Alice then  
(B3)   if no milk then  
(B4)     buy milk;  
(B5) remove note "Bob";
```

2nd Attempt

First, leave a note on the fridge, then check if there is a note from the other person. Only if there is no other note, check if there is milk and go buy milk if there is no milk. Then remove the note.

As processes:

Program for Alice:

```
(A1) write note "Alice";  
(A2) if no note from Bob then  
(A3)   if no milk then  
(A4)     buy milk;  
(A5) remove note "Alice";
```

Program for Bob:

```
(B1) write note "Bob";  
(B2) if no note from Alice then  
(B3)   if no milk then  
(B4)     buy milk;  
(B5) remove note "Bob";
```

Alice

First, leave a note on the fridge, then check if there is a note from the other person. Only if there is no other note, check if there is milk and go buy milk if there is no milk. Then remove the note.

As processes:

Program for Alice:

```
(A1) write note "Alice";  
(A2) if no note from Bob then  
(A3)   if no milk then  
(A4)     buy milk;  
(A5) remove note "Alice";
```

Program for Bob:

```
(B1) write note "Bob";  
(B2) if no note from Alice then  
(B3)   if no milk then  
(B4)     buy milk;  
(B5) remove note "Bob";
```

Alice Bob

First, leave a note on the fridge, then check if there is a note from the other person. Only if there is no other note, check if there is milk and go buy milk if there is no milk. Then remove the note.

As processes:

Program for Alice:

```
(A1) write note "Alice";  
(A2) if no note from Bob then  
(A3)   if no milk then  
(A4)     buy milk;  
(A5) remove note "Alice";
```

Program for Bob:

```
(B1) write note "Bob";  
(B2) if no note from Alice then  
(B3)   if no milk then  
(B4)     buy milk;  
(B5) remove note "Bob";
```

Alice Bob

First, leave a note on the fridge, then check if there is a note from the other person. Only if there is no other note, check if there is milk and go buy milk if there is no milk. Then remove the note.

As processes:

Program for Alice:

```
(A1) write note "Alice";  
(A2) if no note from Bob then  
(A3)   if no milk then  
(A4)     buy milk;  
(A5) remove note "Alice";
```

Program for Bob:

```
(B1) write note "Bob";  
(B2) if no note from Alice then  
(B3)   if no milk then  
(B4)     buy milk;  
(B5) remove note "Bob";
```

Alice Bob

First, leave a note on the fridge, then check if there is a note from the other person. Only if there is no other note, check if there is milk and go buy milk if there is no milk. Then remove the note.

As processes:

Program for Alice:

```
(A1) write note "Alice";  
(A2) if no note from Bob then  
(A3)   if no milk then  
(A4)     buy milk;  
(A5) remove note "Alice";
```

Program for Bob:

```
(B1) write note "Bob";  
(B2) if no note from Alice then  
(B3)   if no milk then  
(B4)     buy milk;  
(B5) remove note "Bob";
```

Bob

First, leave a note on the fridge, then check if there is a note from the other person. Only if there is no other note, check if there is milk and go buy milk if there is no milk. Then remove the note.

As processes:

Program for Alice:

```
(A1) write note "Alice";  
(A2) if no note from Bob then  
(A3)   if no milk then  
(A4)     buy milk;  
(A5) remove note "Alice";
```

Program for Bob:

```
(B1) write note "Bob";  
(B2) if no note from Alice then  
(B3)   if no milk then  
(B4)     buy milk;  
(B5) remove note "Bob";
```

No milk!

3rd Attempt

Alice

As before: Write a note, if there is no note from Bob, check if there is milk and go buy milk. Remove note.

Bob

Write note, wait until there is no note from Alice on the fridge, then check if milk is empty, go buy milk, remove own note.

As processes:

Program for Alice:

```
(A1) write note "Alice";  
(A2) if no note from Bob then  
(A3)   if no milk then  
(A4)     buy milk;  
(A5) remove note "Alice";
```

Program for Bob:

```
(B1) write note "Bob";  
(B2) while note from Alice do skip;  
(B3) if no milk then  
(B4)   buy milk;  
(B5) remove note "Bob";
```

3rd Attempt

Alice

As before: Write a note, if there is no note from Bob, check if there is milk and go buy milk. Remove note.

Bob

Write note, wait until there is no note from Alice on the fridge, then check if milk is empty, go buy milk, remove own note.

As processes:

Program for Alice:

```
(A1) write note "Alice";  
(A2) if no note from Bob then  
(A3)   if no milk then  
(A4)     buy milk;  
(A5) remove note "Alice";
```

Program for Bob:

```
(B1) write note "Bob";  
(B2) while note from Alice do skip;  
(B3) if no milk then  
(B4)   buy milk;  
(B5) remove note "Bob";
```

Alice

3rd Attempt

Alice

As before: Write a note, if there is no note from Bob, check if there is milk and go buy milk. Remove note.

Bob

Write note, wait until there is no note from Alice on the fridge, then check if milk is empty, go buy milk, remove own note.

As processes:

Program for Alice:

```
(A1) write note "Alice";  
(A2) if no note from Bob then  
(A3)   if no milk then  
(A4)     buy milk;  
(A5) remove note "Alice";
```

Program for Bob:

```
(B1) write note "Bob";  
(B2) while note from Alice do skip;  
(B3) if no milk then  
(B4)   buy milk;  
(B5) remove note "Bob";
```

Alice Bob

3rd Attempt

Alice

As before: Write a note, if there is no note from Bob, check if there is milk and go buy milk. Remove note.

Bob

Write note, wait until there is no note from Alice on the fridge, then check if milk is empty, go buy milk, remove own note.

As processes:

Program for Alice:

```
(A1) write note "Alice";  
(A2) if no note from Bob then  
(A3)   if no milk then  
(A4)     buy milk;  
(A5) remove note "Alice";
```

Program for Bob:

```
(B1) write note "Bob";  
(B2) while note from Alice do skip;  
(B3) if no milk then  
(B4)   buy milk;  
(B5) remove note "Bob";
```

Alice Bob

3rd Attempt

Alice

As before: Write a note, if there is no note from Bob, check if there is milk and go buy milk. Remove note.

Bob

Write note, wait until there is no note from Alice on the fridge, then check if milk is empty, go buy milk, remove own note.

As processes:

Program for Alice:

```
(A1) write note "Alice";  
(A2) if no note from Bob then  
(A3)   if no milk then  
(A4)     buy milk;  
(A5) remove note "Alice";
```

Program for Bob:

```
(B1) write note "Bob";  
(B2) while note from Alice do skip;  
(B3) if no milk then  
(B4)   buy milk;  
(B5) remove note "Bob";
```

Alice Bob

3rd Attempt

Alice

As before: Write a note, if there is no note from Bob, check if there is milk and go buy milk. Remove note.

Bob

Write note, wait until there is no note from Alice on the fridge, then check if milk is empty, go buy milk, remove own note.

As processes:

Program for Alice:

```
(A1) write note "Alice";  
(A2) if no note from Bob then  
(A3)   if no milk then  
(A4)     buy milk;  
(A5) remove note "Alice";
```

Program for Bob:

```
(B1) write note "Bob";  
(B2) while note from Alice do skip;  
(B3) if no milk then  
(B4)   buy milk;  
(B5) remove note "Bob";
```

Bob

3rd Attempt

Alice

As before: Write a note, if there is no note from Bob, check if there is milk and go buy milk. Remove note.

Bob

Write note, wait until there is no note from Alice on the fridge, then check if milk is empty, go buy milk, remove own note.

As processes:

Program for Alice:

```
(A1) write note "Alice";  
(A2) if no note from Bob then  
(A3)   if no milk then  
(A4)     buy milk;  
(A5) remove note "Alice";
```

Program for Bob:

```
(B1) write note "Bob";  
(B2) while note from Alice do skip;  
(B3) if no milk then  
(B4)   buy milk;  
(B5) remove note "Bob";
```

Alice Bob

3rd Attempt

Alice

As before: Write a note, if there is no note from Bob, check if there is milk and go buy milk. Remove note.

Bob

Write note, wait until there is no note from Alice on the fridge, then check if milk is empty, go buy milk, remove own note.

As processes:

Program for Alice:

```
(A1) write note "Alice";  
(A2) if no note from Bob then  
(A3)   if no milk then  
(A4)     buy milk;  
(A5) remove note "Alice";
```

Program for Bob:

```
(B1) write note "Bob";  
(B2) while note from Alice do skip;  
(B3) if no milk then  
(B4)   buy milk;  
(B5) remove note "Bob";
```

Alice Bob

3rd Attempt

Alice

As before: Write a note, if there is no note from Bob, check if there is milk and go buy milk. Remove note.

Bob

Write note, wait until there is no note from Alice on the fridge, then check if milk is empty, go buy milk, remove own note.

As processes:

Program for Alice:

```
(A1) write note "Alice";  
(A2) if no note from Bob then  
(A3)   if no milk then  
(A4)     buy milk;  
(A5) remove note "Alice";
```

Program for Bob:

```
(B1) write note "Bob";  
(B2) while note from Alice do skip;  
(B3) if no milk then  
(B4)   buy milk;  
(B5) remove note "Bob";
```

Alice Bob

3rd Attempt

Alice

As before: Write a note, if there is no note from Bob, check if there is milk and go buy milk. Remove note.

Bob

Write note, wait until there is no note from Alice on the fridge, then check if milk is empty, go buy milk, remove own note.

As processes:

Program for Alice:

```
(A1) write note "Alice";  
(A2) if no note from Bob then  
(A3)   if no milk then  
(A4)     buy milk;  
(A5) remove note "Alice";
```

Program for Bob:

```
(B1) write note "Bob";  
(B2) while note from Alice do skip;  
(B3) if no milk then  
(B4)   buy milk;  
(B5) remove note "Bob";
```

Bob

3rd Attempt

Alice

As before: Write a note, if there is no note from Bob, check if there is milk and go buy milk. Remove note.

Bob

Write note, wait until there is no note from Alice on the fridge, then check if milk is empty, go buy milk, remove own note.

As processes:

Program for Alice:

```
(A1) write note "Alice";  
(A2) if no note from Bob then  
(A3)   if no milk then  
(A4)     buy milk;  
(A5) remove note "Alice";
```

Program for Bob:

```
(B1) write note "Bob";  
(B2) while note from Alice do skip;  
(B3) if no milk then  
(B4)   buy milk;  
(B5) remove note "Bob";
```

Alice Bob

3rd Attempt

Alice

As before: Write a note, if there is no note from Bob, check if there is milk and go buy milk. Remove note.

Bob

Write note, wait until there is no note from Alice on the fridge, then check if milk is empty, go buy milk, remove own note.

As processes:

Program for Alice:

```
(A1) write note "Alice";  
(A2) if no note from Bob then  
(A3)   if no milk then  
(A4)     buy milk;  
(A5) remove note "Alice";
```

Program for Bob:

```
(B1) write note "Bob";  
(B2) while note from Alice do skip;  
(B3) if no milk then  
(B4)   buy milk;  
(B5) remove note "Bob";
```

Alice Bob

3rd Attempt

Alice

As before: Write a note, if there is no note from Bob, check if there is milk and go buy milk. Remove note.

Bob

Write note, wait until there is no note from Alice on the fridge, then check if milk is empty, go buy milk, remove own note.

As processes:

Program for Alice:

```
(A1) write note "Alice";  
(A2) if no note from Bob then  
(A3)   if no milk then  
(A4)     buy milk;  
(A5) remove note "Alice";
```

Program for Bob:

```
(B1) write note "Bob";  
(B2) while note from Alice do skip;  
(B3) if no milk then  
(B4)   buy milk;  
(B5) remove note "Bob";
```

Alice Bob

3rd Attempt

Alice

As before: Write a note, if there is no note from Bob, check if there is milk and go buy milk. Remove note.

Bob

Write note, wait until there is no note from Alice on the fridge, then check if milk is empty, go buy milk, remove own note.

As processes:

Program for Alice:

```
(A1) write note "Alice";  
(A2) if no note from Bob then  
(A3)   if no milk then  
(A4)     buy milk;  
(A5) remove note "Alice";
```

Program for Bob:

```
(B1) write note "Bob";  
(B2) while note from Alice do skip;  
(B3) if no milk then  
(B4)   buy milk;  
(B5) remove note "Bob";
```

Bob

3rd Attempt

Alice

As before: Write a note, if there is no note from Bob, check if there is milk and go buy milk. Remove note.

Bob

Write note, wait until there is no note from Alice on the fridge, then check if milk is empty, go buy milk, remove own note.

As processes:

Program for Alice:

```
(A1) write note "Alice";  
(A2) if no note from Bob then  
(A3)   if no milk then  
(A4)     buy milk;  
(A5) remove note "Alice";
```

Program for Bob:

```
(B1) write note "Bob";  
(B2) while note from Alice do skip;  
(B3) if no milk then  
(B4)   buy milk;  
(B5) remove note "Bob";
```

The solution only works if the relative speed is right. Otherwise:

$(A1), (B1), (B2), \underbrace{(A2), (A5), (A1), (B2)}_{\text{infinitely often}}$ without buying milk.

infinitely often

A Solution

Program for Alice

```
(A1) write a1;  
(A2) if b2  
(A3)   then write a2;  
(A4)   else remove a2;  
(A5) while  $b1 \wedge ((a2 \wedge b2) \vee (\neg a2 \wedge \neg b2))$   
(A6)   do skip;  
(A7) if no milk  
(A8)   then buy milk;  
(A9) remove a1;
```

Program for Bob

```
(B1) write b1;  
(B2) if  $\neg a2$   
(B3)   then write b2;  
(B4)   else remove b2  
(B5) while  $a1 \wedge ((a2 \wedge \neg b2) \vee (\neg a2 \wedge b2))$   
(B6)   do skip;  
(B7) if no milk  
(B8)   then buy milk;  
(B9) remove b1;
```

A Solution

Program for Alice

```
(A1) write a1;  
(A2) if b2  
(A3)   then write a2;  
(A4)   else remove a2;  
(A5) while  $b1 \wedge ((a2 \wedge b2) \vee (\neg a2 \wedge \neg b2))$   
(A6)   do skip;  
(A7) if no milk  
(A8)   then buy milk;  
(A9) remove a1;
```

Program for Bob

```
(B1) write b1;  
(B2) if  $\neg a2$   
(B3)   then write b2;  
(B4)   else remove b2  
(B5) while  $a1 \wedge ((a2 \wedge \neg b2) \vee (\neg a2 \wedge b2))$   
(B6)   do skip;  
(B7) if no milk  
(B8)   then buy milk;  
(B9) remove b1;
```

a1	a2	b2	b1	checks milk
✓	dc	dc	×	Alice
×	dc	dc	✓	Bob

dc = don't care

a1	a2	b2	b1	checks milk
✓	×	✓	✓	Alice
✓	✓	×	✓	Alice
✓	✓	✓	✓	Bob
✓	×	×	✓	Bob

A Solution

Program for Alice

```
(A1) write a1;  
(A2) if b2  
(A3)   then write a2;  
(A4)   else remove a2;  
(A5) while b1  $\wedge ((a2 \wedge b2) \vee (\neg a2 \wedge \neg b2))$   
(A6)   do skip;  
(A7) if no milk  
(A8)   then buy milk;  
(A9) remove a1;
```

Program for Bob

```
(B1) write b1;  
(B2) if  $\neg a2$   
(B3)   then write b2;  
(B4)   else remove b2  
(B5) while a1  $\wedge ((a2 \wedge \neg b2) \vee (\neg a2 \wedge b2))$   
(B6)   do skip;  
(B7) if no milk  
(B8)   then buy milk;  
(B9) remove b1;
```

a1 a2

✓ dc

× dc

dc = don't care

means "... wants to access the fridge"

b1 checks milk

✓ Alice

✓ Alice

✓ Bob

✓ Bob

✓ ✓ ✓

✓ × ×

A Solution

Program for Alice

```
(A1) write a1;
(A2) if b2
(A3)   then write a2;
(A4)   else remove a2;
(A5) while b1  $\wedge$  ((a2  $\wedge$  b2)  $\vee$  ( $\neg$ a2  $\wedge$   $\neg$ b2))
(A6)   do skip;
(A7) if no milk
(A8)   then buy milk;
(A9) remove a1;
```

Program for Bob

```
(B1) write b1;
(B2) if  $\neg$ a2
(B3)   then write b2;
(B4)   else remove b2
(B5) while a1  $\wedge$  ((a2  $\wedge$   $\neg$ b2)  $\vee$  ( $\neg$ a2  $\wedge$  b2))
(B6)   do skip;
(B7) if no milk
(B8)   then buy milk;
(B9) remove b1;
```

a1	a2	b2	b1	cl	has milk
✓	dc	dc	×	A	
×	dc	dc	✓	B	
dc = don't care					

attempts to give priority to Bob

✓	✓	✓	✓	Bob
✓	×	×	✓	Bob

A Solution

Program for Alice

```
(A1) write a1;
(A2) if b2
(A3)   then write a2;
(A4)   else remove a2;
(A5) while b1  $\wedge$  ((a2  $\wedge$  b2)  $\vee$  ( $\neg$ a2  $\wedge$   $\neg$ b2))
(A6)   do skip;
(A7) if no milk
(A8)   then buy milk;
(A9) remove a1;
```

Program for Bob

```
(B1) write b1;
(B2) if  $\neg$ a2
(B3)   then write b2;
(B4)   else remove b2
(B5) while a1  $\wedge$  ((a2  $\wedge$   $\neg$ b2)  $\vee$  ( $\neg$ a2  $\wedge$  b2))
(B6)   do skip;
(B7) if no milk
(B8)   then buy milk;
(B9) remove b1;
```

a1	a2	b2	b1	checks milk
✓	attempts to give priority to Alice		✓	Alice
×			✓	Alice
dc = don't care		✓	✓	Bob
		✓	×	Bob

Formalization of the Mutual Exclusion Problem

- Definition of the problem
- Requirements on a solution
- Assumptions about the model
- Deadlock-Freedom and Starvation-Freedom

The Mutual Exclusion Problem

- Mutual exclusion problem informally:
Ensure **exclusive access** to a shared resource
- the part of the program that accesses the shared resource is called the **critical section**
- **Race condition**: Situations in which multiple processes access a shared resource and the value of the resource depends on the order of execution
- Goal of the mutual exclusion problem: **avoid race conditions**

The Mutual Exclusion Problem, Formally

Code structure of each process

```
loop forever
  remaining code
  initialization code
  critical section
  exit code
end loop
```

The Mutual Exclusion Problem, Formally (2)

Assumptions

- In the remaining code, a process cannot influence other processes. Otherwise, everything is permitted there (infinite loops, etc.).
- Resources (program variables) used in the initialization code or exit code are not changed by the code in the critical section and the remaining code.

```
loop forever
  remaining code
  initialization code
  critical section
  exit code
end loop
```

The Mutual Exclusion Problem, Formally (3)

Assumptions (continued)

- During execution of the initialization code, the code in the critical section, and the exit code **errors do not occur**
- Code in the critical section and in the exit code:
Only a finite number of execution steps (no infinite loops!).
Implied (due to fairness): After entering the critical section, the process leaves it and executes the exit code.

```
loop forever
  remaining code
  initialization code
  critical section
  exit code
end loop
```

Solution to the mutual exclusion problem

Write the initialization and exit code so that the following requirements are met:

- **Mutual exclusion**: Never are two or more processes in their critical section at the same time.
- **Deadlock freedom**: If a process wants to enter its critical section, then some process will eventually enter the critical section.

Stronger requirement than deadlock freedom:

Starvation-freedom

- **Starvation-freedom**: If a process wants to enter its critical section, then it must enter it after a finite number of computation steps.
- A starvation-free algorithm is also deadlock-free.

- Mutual exclusion is a so-called **safety property**, a property that must **always** be satisfied.
- Deadlock-freedom and starvation-freedom are so-called **liveness properties**, properties that must be satisfied at some point.
- Temporal logic can be used to provide a formal proof that these properties are satisfied.
- For example, model checking can be used for the proof.
- We provide the proofs directly (rather informally).

Algorithms for 2 Processes with Atomic Read and Write

Algorithms for 2 Processes with Atomic Read & Write

In the following:

- Some algorithms for the mutual exclusion problem
- Concurrent program with exactly **two** processes

Assumptions:

- Only **read** and **write** as atomic memory operations of the shared memory
- E.g., `swap(X,Y)`, `X := Y`, etc. are prohibited for shared variables

Abbreviation:

- **`await condition;`**
instead of
`while  $\neg$ condition do skip;`

Dekker's Algorithm

- Historically the first correct solution
- Dutch mathematician Theodorus Jozef Dekker (1927-2021)
- Mention of Dekker's algorithm: Dijkstra 1965
- Not a very simple solution

Dekker's Algorithm

Initial: $wantp = \text{False}$, $wantq = \text{False}$, $turn = 1$

Process P :

```
loop forever
(P1)  remaining code
(P2)   $wantp := \text{True}$ ;
(P3)  while  $wantq$  do
(P4)    if  $turn=2$  then
(P5)       $wantp := \text{False}$ ;
(P6)      await  $turn=1$ ;
(P7)       $wantp := \text{True}$ ;
(P8)  critical section
(P9)   $turn := 2$ ;
(P10)  $wantp := \text{False}$ ;
end loop
```

Process Q :

```
loop forever
(Q1)  remaining code
(Q2)   $wantq := \text{True}$ ;
(Q3)  while  $wantp$  do
(Q4)    if  $turn=1$  then
(Q5)       $wantq := \text{False}$ ;
(Q6)      await  $turn=2$ ;
(Q7)       $wantq := \text{True}$ ;
(Q8)  critical section
(Q9)   $turn := 1$ ;
(Q10)  $wantq := \text{False}$ ;
end loop
```

Run: Only P wants to enter the critical section
 $wantp=\text{False}$, $wantq =\text{False}$, $turn =1$

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  while wantq do
(P4)    if turn=2 then
(P5)      wantp := False;
(P6)      await turn=1;
(P7)      wantp := True;
(P8)  critical section
(P9)  turn := 2;
(P10) wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  while wantp do
(Q4)    if turn=1 then
(Q5)      wantq := False;
(Q6)      await turn=2;
(Q7)      wantq := True;
(Q8)  critical section
(Q9)  turn := 1;
(Q10) wantq := False;
end loop
```

Run: Only *P* wants to enter the critical section
wantp=False, wantq =False , turn =1

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  while wantq do
(P4)    if turn=2 then
(P5)      wantp := False;
(P6)      await turn=1;
(P7)      wantp := True;
(P8)  critical section
(P9)  turn := 2;
(P10) wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  while wantp do
(Q4)    if turn=1 then
(Q5)      wantq := False;
(Q6)      await turn=2;
(Q7)      wantq := True;
(Q8)  critical section
(Q9)  turn := 1;
(Q10) wantq := False;
end loop
```

Run: Only *P* wants to enter the critical section
wantp=True, wantq =False , turn =1

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  while wantq do
(P4)    if turn=2 then
(P5)      wantp := False;
(P6)      await turn=1;
(P7)      wantp := True;
(P8)  critical section
(P9)  turn := 2;
(P10) wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  while wantp do
(Q4)    if turn=1 then
(Q5)      wantq := False;
(Q6)      await turn=2;
(Q7)      wantq := True;
(Q8)  critical section
(Q9)  turn := 1;
(Q10) wantq := False;
end loop
```

Run: Only *P* wants to enter the critical section
wantp=True, wantq =False , turn =1

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  while wantq do
(P4)    if turn=2 then
(P5)      wantp := False;
(P6)      await turn=1;
(P7)      wantp := True;
(P8)  critical section
(P9)  turn := 2;
(P10) wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  while wantp do
(Q4)    if turn=1 then
(Q5)      wantq := False;
(Q6)      await turn=2;
(Q7)      wantq := True;
(Q8)  critical section
(Q9)  turn := 1;
(Q10) wantq := False;
end loop
```

Run: Only *P* wants to enter the critical section
wantp=True, wantq =False , turn =1

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  while wantq do
(P4)    if turn=2 then
(P5)      wantp := False;
(P6)      await turn=1;
(P7)      wantp := True;
(P8)  critical section
(P9)  turn := 2;
(P10) wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  while wantp do
(Q4)    if turn=1 then
(Q5)      wantq := False;
(Q6)      await turn=2;
(Q7)      wantq := True;
(Q8)  critical section
(Q9)  turn := 1;
(Q10) wantq := False;
end loop
```

Run: Only *P* wants to enter the critical section
wantp=True, wantq =False , turn =2

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  while wantq do
(P4)    if turn=2 then
(P5)      wantp := False;
(P6)      await turn=1;
(P7)      wantp := True;
(P8)  critical section
(P9)  turn := 2;
(P10) wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  while wantp do
(Q4)    if turn=1 then
(Q5)      wantq := False;
(Q6)      await turn=2;
(Q7)      wantq := True;
(Q8)  critical section
(Q9)  turn := 1;
(Q10) wantq := False;
end loop
```

Run: Only *P* wants to enter the critical section
wantp=False, wantq =False , turn =1

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  while wantq do
(P4)    if turn=2 then
(P5)      wantp := False;
(P6)      await turn=1;
(P7)      wantp := True;
(P8)  critical section
(P9)  turn := 2;
(P10) wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  while wantp do
(Q4)    if turn=1 then
(Q5)      wantq := False;
(Q6)      await turn=2;
(Q7)      wantq := True;
(Q8)  critical section
(Q9)  turn := 1;
(Q10) wantq := False;
end loop
```

Run: *P* first in critical section
wantp=False, wantq =False , turn =1

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  while wantq do
(P4)    if turn=2 then
(P5)      wantp := False;
(P6)      await turn=1;
(P7)      wantp := True;
(P8)  critical section
(P9)  turn := 2;
(P10) wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  while wantp do
(Q4)    if turn=1 then
(Q5)      wantq := False;
(Q6)      await turn=2;
(Q7)      wantq := True;
(Q8)  critical section
(Q9)  turn := 1;
(Q10) wantq := False;
end loop
```

Run: *P* first in critical section
wantp=True, wantq =False , turn =1

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)   remaining code
(P2)   wantp := True;
(P3)   while wantq do
(P4)       if turn=2 then
(P5)           wantp := False;
(P6)           await turn=1;
(P7)           wantp := True;
(P8)   critical section
(P9)   turn := 2;
(P10)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)   remaining code
(Q2)   wantq := True;
(Q3)   while wantp do
(Q4)       if turn=1 then
(Q5)           wantq := False;
(Q6)           await turn=2;
(Q7)           wantq := True;
(Q8)   critical section
(Q9)   turn := 1;
(Q10)  wantq := False;
end loop
```

Run: *P* first in critical section
wantp=True, wantq =False , turn =1

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  while wantq do
(P4)    if turn=2 then
(P5)      wantp := False;
(P6)      await turn=1;
(P7)      wantp := True;
(P8)  critical section
(P9)  turn := 2;
(P10) wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  while wantp do
(Q4)    if turn=1 then
(Q5)      wantq := False;
(Q6)      await turn=2;
(Q7)      wantq := True;
(Q8)  critical section
(Q9)  turn := 1;
(Q10) wantq := False;
end loop
```

Run: *P* first in critical section
wantp=True, wantq =False , turn =1

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  while wantq do
(P4)    if turn=2 then
(P5)      wantp := False;
(P6)      await turn=1;
(P7)      wantp := True;
(P8)  critical section
(P9)  turn := 2;
(P10) wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  while wantp do
(Q4)    if turn=1 then
(Q5)      wantq := False;
(Q6)      await turn=2;
(Q7)      wantq := True;
(Q8)  critical section
(Q9)  turn := 1;
(Q10) wantq := False;
end loop
```

Run: *P* first in critical section
wantp=True, wantq =False , turn =1

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)   remaining code
(P2)   wantp := True;
(P3)   while wantq do
(P4)     if turn=2 then
(P5)       wantp := False;
(P6)       await turn=1;
(P7)       wantp := True;
(P8)   critical section
(P9)   turn := 2;
(P10)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)   remaining code
(Q2)   wantq := True;
(Q3)   while wantp do
(Q4)     if turn=1 then
(Q5)       wantq := False;
(Q6)       await turn=2;
(Q7)       wantq := True;
(Q8)   critical section
(Q9)   turn := 1;
(Q10)  wantq := False;
end loop
```

Run: *P* first in critical section
wantp=True, wantq =False , turn =1

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  while wantq do
(P4)    if turn=2 then
(P5)      wantp := False;
(P6)      await turn=1;
(P7)      wantp := True;
(P8)  critical section
(P9)  turn := 2;
(P10) wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  while wantp do
(Q4)    if turn=1 then
(Q5)      wantq := False;
(Q6)      await turn=2;
(Q7)      wantq := True;
(Q8)  critical section
(Q9)  turn := 1;
(Q10) wantq := False;
end loop
```

Run: *P* first in critical section
wantp=True, wantq =True , turn =1

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)   remaining code
(P2)   wantp := True;
(P3)   while wantq do
(P4)     if turn=2 then
(P5)       wantp := False;
(P6)       await turn=1;
(P7)       wantp := True;
(P8)   critical section
(P9)   turn := 2;
(P10)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)   remaining code
(Q2)   wantq := True;
(Q3)   while wantp do
(Q4)     if turn=1 then
(Q5)       wantq := False;
(Q6)       await turn=2;
(Q7)       wantq := True;
(Q8)   critical section
(Q9)   turn := 1;
(Q10)  wantq := False;
end loop
```

Run: *P* first in critical section
wantp=True, wantq =True , turn =1

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  while wantq do
(P4)    if turn=2 then
(P5)      wantp := False;
(P6)      await turn=1;
(P7)      wantp := True;
(P8)  critical section
(P9)  turn := 2;
(P10) wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  while wantp do
(Q4)    if turn=1 then
(Q5)      wantq := False;
(Q6)      await turn=2;
(Q7)      wantq := True;
(Q8)  critical section
(Q9)  turn := 1;
(Q10) wantq := False;
end loop
```

Run: *P* first in critical section
wantp=True, wantq =True , turn =1

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  while wantq do
(P4)    if turn=2 then
(P5)      wantp := False;
(P6)      await turn=1;
(P7)      wantp := True;
(P8)  critical section
(P9)  turn := 2;
(P10) wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  while wantp do
(Q4)    if turn=1 then
(Q5)      wantq := False;
(Q6)      await turn=2;
(Q7)      wantq := True;
(Q8)  critical section
(Q9)  turn := 1;
(Q10) wantq := False;
end loop
```

Run: *P* first in critical section
wantp=True, wantq =False , turn =1

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)   remaining code
(P2)   wantp := True;
(P3)   while wantq do
(P4)     if turn=2 then
(P5)       wantp := False;
(P6)       await turn=1;
(P7)       wantp := True;
(P8)   critical section
(P9)   turn := 2;
(P10)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)   remaining code
(Q2)   wantq := True;
(Q3)   while wantp do
(Q4)     if turn=1 then
(Q5)       wantq := False;
(Q6)       await turn=2;
(Q7)       wantq := True;
(Q8)   critical section
(Q9)   turn := 1;
(Q10)  wantq := False;
end loop
```

Run: *P* first in critical section
wantp=True, wantq =False , turn =1

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)   remaining code
(P2)   wantp := True;
(P3)   while wantq do
(P4)     if turn=2 then
(P5)       wantp := False;
(P6)       await turn=1;
(P7)       wantp := True;
(P8)   critical section
(P9)   turn := 2;
(P10)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)   remaining code
(Q2)   wantq := True;
(Q3)   while wantp do
(Q4)     if turn=1 then
(Q5)       wantq := False;
(Q6)       await turn=2;
(Q7)       wantq := True;
(Q8)   critical section
(Q9)   turn := 1;
(Q10)  wantq := False;
end loop
```

Run: *P* first in critical section
wantp=True, wantq =False , turn =1

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  while wantq do
(P4)    if turn=2 then
(P5)      wantp := False;
(P6)      await turn=1;
(P7)      wantp := True;
(P8)  critical section
(P9)  turn := 2;
(P10) wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  while wantp do
(Q4)    if turn=1 then
(Q5)      wantq := False;
(Q6)      await turn=2;
(Q7)      wantq := True;
(Q8)  critical section
(Q9)  turn := 1;
(Q10) wantq := False;
end loop
```

Run: *P* first in critical section
wantp=True, wantq =False , turn =2

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)   remaining code
(P2)   wantp := True;
(P3)   while wantq do
(P4)       if turn=2 then
(P5)           wantp := False;
(P6)           await turn=1;
(P7)           wantp := True;
(P8)   critical section
(P9)   turn := 2;
(P10)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)   remaining code
(Q2)   wantq := True;
(Q3)   while wantp do
(Q4)       if turn=1 then
(Q5)           wantq := False;
(Q6)           await turn=2;
(Q7)           wantq := True;
(Q8)   critical section
(Q9)   turn := 1;
(Q10)  wantq := False;
end loop
```

Run: *P* first in critical section
wantp=True, wantq =False , turn =2

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)   remaining code
(P2)   wantp := True;
(P3)   while wantq do
(P4)       if turn=2 then
(P5)           wantp := False;
(P6)           await turn=1;
(P7)           wantp := True;
(P8)   critical section
(P9)   turn := 2;
(P10)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)   remaining code
(Q2)   wantq := True;
(Q3)   while wantp do
(Q4)       if turn=1 then
(Q5)           wantq := False;
(Q6)           await turn=2;
(Q7)           wantq := True;
(Q8)   critical section
(Q9)   turn := 1;
(Q10)  wantq := False;
end loop
```

Run: *P* first in critical section
wantp=True, wantq =False , turn =2

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  while wantq do
(P4)    if turn=2 then
(P5)      wantp := False;
(P6)      await turn=1;
(P7)      wantp := True;
(P8)  critical section
(P9)  turn := 2;
(P10) wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  while wantp do
(Q4)    if turn=1 then
(Q5)      wantq := False;
(Q6)      await turn=2;
(Q7)      wantq := True;
(Q8)  critical section
(Q9)  turn := 1;
(Q10) wantq := False;
end loop
```

Run: *P* first in critical section
wantp=False, wantq =False , turn =2

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  while wantq do
(P4)    if turn=2 then
(P5)      wantp := False;
(P6)      await turn=1;
(P7)      wantp := True;
(P8)  critical section
(P9)  turn := 2;
(P10) wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  while wantp do
(Q4)    if turn=1 then
(Q5)      wantq := False;
(Q6)      await turn=2;
(Q7)      wantq := True;
(Q8)  critical section
(Q9)  turn := 1;
(Q10) wantq := False;
end loop
```

Run: *P* first in critical section
wantp=False, wantq =True , turn =2

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  while wantq do
(P4)    if turn=2 then
(P5)      wantp := False;
(P6)      await turn=1;
(P7)      wantp := True;
(P8)  critical section
(P9)  turn := 2;
(P10) wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  while wantp do
(Q4)    if turn=1 then
(Q5)      wantq := False;
(Q6)      await turn=2;
(Q7)      wantq := True;
(Q8)  critical section
(Q9)  turn := 1;
(Q10) wantq := False;
end loop
```

Run: *P* first in critical section
wantp=False, wantq =True , turn =2

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  while wantq do
(P4)    if turn=2 then
(P5)      wantp := False;
(P6)      await turn=1;
(P7)      wantp := True;
(P8)  critical section
(P9)  turn := 2;
(P10) wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  while wantp do
(Q4)    if turn=1 then
(Q5)      wantq := False;
(Q6)      await turn=2;
(Q7)      wantq := True;
(Q8)  critical section
(Q9)  turn := 1;
(Q10) wantq := False;
end loop
```

Run: *P* first in critical section
wantp=False, wantq =True , turn =2

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  while wantq do
(P4)    if turn=2 then
(P5)      wantp := False;
(P6)      await turn=1;
(P7)      wantp := True;
(P8)  critical section
(P9)  turn := 2;
(P10) wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  while wantp do
(Q4)    if turn=1 then
(Q5)      wantq := False;
(Q6)      await turn=2;
(Q7)      wantq := True;
(Q8)  critical section
(Q9)  turn := 1;
(Q10) wantq := False;
end loop
```

Run: *P* first in critical section
wantp=False, wantq =True , turn =1

Dekker's Algorithm

Initial: wantp = False, wantq = False, turn = 1

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  while wantq do
(P4)    if turn=2 then
(P5)      wantp := False;
(P6)      await turn=1;
(P7)      wantp := True;
(P8)  critical section
(P9)  turn := 2;
(P10) wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  while wantp do
(Q4)    if turn=1 then
(Q5)      wantq := False;
(Q6)      await turn=2;
(Q7)      wantq := True;
(Q8)  critical section
(Q9)  turn := 1;
(Q10) wantq := False;
end loop
```

Run: *P* first in critical section
wantp=False, wantq =False , turn =1

Dekker's Algorithm is Correct (1)

Lemma

Dekker's algorithm satisfies mutual exclusion.

Proof by contradiction:

Assume the claim is false.

$\implies \exists$ state where P and Q are in the critical section

There are cases

1. P and Q never run into the loop body, i.e.
 - sequence without Q -commands ... (P3), (P8) ..., and
 - sequence without P -commands ... (Q3), (Q8)

impossible, since wantp and wantq have to be false

Initial: wantp = False,
wantq = False,
turn = 1

Process P :

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  while wantq do
(P4)    if turn=2 then
(P5)      wantp := False;
(P6)      await turn=1;
(P7)      wantp := True;
(P8)  critical section
(P9)  turn := 2;
(P10) wantp := False
end loop
```

Process Q :

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  while wantp do
(Q4)    if turn=1 then
(Q5)      wantq := False;
(Q6)      await turn=2;
(Q7)      wantq := True;
(Q8)  critical section
(Q9)  turn := 1;
(Q10) wantq := False
end loop
```

Dekker's Algorithm is Correct (2)

Lemma

Dekker's algorithm satisfies mutual exclusion.

Proof:

2. Process P runs through the loop body while the other process Q is in the critical section.

P will never evaluate the while condition to False as long as Q is in the CS. Therefore: **Impossible**.

3. Both processes run through the loop body.
 $\implies$ One process gets stuck at await (turn is only changed after the critical section). **Impossible**.

I.e., assumption false, mutual exclusion satisfied.

Initial: wantp = False,
 wantq = False,
 turn = 1

Process P :

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  while wantq do
(P4)      if turn=2 then
(P5)          wantp := False;
(P6)          await turn=1;
(P7)          wantp := True;
(P8)  critical section
(P9)  turn := 2;
(P10) wantp := False;
end loop
```

Process Q :

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  while wantp do
(Q4)      if turn=1 then
(Q5)          wantq := False;
(Q6)          await turn=2;
(Q7)          wantq := True;
(Q8)  critical section
(Q9)  turn := 1;
(Q10) wantq := False;
end loop
```

Dekker's Algorithm is Correct (3)

Lemma

Dekker's algorithm is starvation-free.

Proof: Assumption: Claim is false.

W.l.o.g. P starves in the initialization code. Possible reasons:

(1) stuck at await. Where is Q located?

- Q is in the remaining code. Impossible: P entered the while loop, so wantq was true at some point, meaning Q passed through the CS and exit code, where turn would have been set to 1.
- Q is in the critical section or exit code
 $\Rightarrow$ turn is set to 1 and turn can no longer be set to 2
- Q is in the initialization code. wantp must be false, since P is in (P6).
 $\Rightarrow Q$ enters critical section, then sets turn = 1.

Initial: wantp = False,
wantq = False,
turn = 1

Process P :

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  while wantq do
(P4)    if turn=2 then
(P5)      wantp := False
(P6)      await turn=1
(P7)      wantp := True;
(P8)  critical section
(P9)  turn := 2;
(P10) wantp := False;
end loop
```

Process Q :

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  while wantp do
(Q4)    if turn=1 then
(Q5)      wantq := False
(Q6)      await turn=2
(Q7)      wantq := True;
(Q8)  critical section
(Q9)  turn := 1;
(Q10) wantq := False;
end loop
```

Dekker's Algorithm is Correct (4)

Lemma

Dekker's algorithm is starvation-free.

Proof (cont.):

(2) while loop is executed infinitely often:

- No hanging at await, therefore $\text{turn} = 1$
- Value of turn can only be changed by P in the exit code
- Q is in the remaining code. Then $\text{wantq} = \text{False}$, while is not executed infinitely often
- Q is in the critical section or in the exit code. Then at some point $\text{wantq} = \text{False}$ is set and therefore while is not run infinitely.
- Q in the initialization code: Then Q must hang on await (since $\text{turn} = 1$). Then $\text{wantq} = \text{False} \Rightarrow$ while is not run infinitely.

Initial: $\text{wantp} = \text{False}$,
 $\text{wantq} = \text{False}$,
 $\text{turn} = 1$

Process P :

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  while wantq do
(P4)      if turn=2 then
(P5)          wantp := False
(P6)          await turn=1
(P7)          wantp := True;
(P8)  critical section
(P9)  turn := 2;
(P10) wantp := False;
end loop
```

Process Q :

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  while wantp do
(Q4)      if turn=1 then
(Q5)          wantq := False
(Q6)          await turn=2
(Q7)          wantq := True;
(Q8)  critical section
(Q9)  turn := 1;
(Q10) wantq := False;
end loop
```

Peterson's Algorithm

- Simpler alternative to Dekker's algorithm
- Article by Gary L. Peterson (American computer scientist) 1981
- Algorithm and proof of correctness in 2 pages

Peterson's Algorithm

Initial: $wantp = \text{False}$, $wantq = \text{False}$, $turn = \text{d.c.}$

Process P:

```
loop forever
(P1)  remaining code
(P2)   $wantp := \text{True};$ 
(P3)   $turn := 1;$ 
(P4)  await  $wantq = \text{False}$  or  $turn = 2$ 
(P5)  critical section
(P6)   $wantp := \text{False};$ 
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)   $wantq := \text{True};$ 
(Q3)   $turn := 2;$ 
(Q4)  await  $wantp = \text{False}$  or  $turn = 1$ 
(Q5)  critical section
(Q6)   $wantq := \text{False};$ 
end loop
```

Example run 1

wantp	wantq	turn
False	False	d.c.

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 1

wantp	wantq	turn
False	False	d.c.

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 1

wantp	wantq	turn
False	False	d.c.

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 1

wantp	wantq	turn
True	False	d.c.

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 1

wantp	wantq	turn
True	True	d.c.

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 1

wantp	wantq	turn
True	True	1

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 1

wantp	wantq	turn
True	True	2

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 1

wantp	wantq	turn
True	True	2

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 1

wantp	wantq	turn
True	True	2

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 1

wantp	wantq	turn
True	True	2

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 1

wantp	wantq	turn
False	True	2

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 1

wantp	wantq	turn
False	True	2

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 1

wantp	wantq	turn
False	True	2

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 1

wantp	wantq	turn
False	False	2

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 2

wantp	wantq	turn
False	False	d.c.

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 2

wantp	wantq	turn
False	False	d.c.

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 2

wantp	wantq	turn
True	False	d.c.

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 2

wantp	wantq	turn
True	False	d.c.

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 2

wantp	wantq	turn
True	True	d.c.

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 2

wantp	wantq	turn
True	True	2

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 2

wantp	wantq	turn
True	True	1

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 2

wantp	wantq	turn
True	True	1

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 2

wantp	wantq	turn
True	True	1

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 2

wantp	wantq	turn
True	True	1

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 2

wantp	wantq	turn
True	False	1

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 2

wantp	wantq	turn
True	False	1

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 2

wantp	wantq	turn
True	False	1

Peterson's Algorithm

Initial: wantp = False, wantq = False, turn d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

Example run 2

wantp	wantq	turn
False	False	1

The Peterson Algorithm is Correct (1)

Lemma

Peterson's algorithm guarantees mutual exclusion.

Proof:

Assumption: Statement is false.

- There is a state such that P and Q are in the critical section.
- The tests in (P4) and (Q4) cannot have become true directly after each other, i.e.
neither (P4),(Q4),(P5),(Q5) nor (P4),(P5),(Q4),(Q5) (analogous with P, Q reversed)
Because: turn would then have only allowed one of the two processes to pass (wantp and wantq must both be true) executed at least 1 command

Initial: wantp = False, wantq = False, turn = d.c.

Process P :

```

loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False
      or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop

```

Process Q :

```

loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False
      or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop

```

The Peterson Algorithm is Correct (2)

Lemma

Peterson's algorithm guarantees mutual exclusion.

Proof:

- Therefore: The second process to enter the critical section must have executed at least 1 command **after** the first process entered the critical section.
(P5), ..., (Q3), (Q4), (Q5) or analogously if Q first in the CS
- (Q3) (or (P3), resp) sets the value of turn so that the setting process Q (or P) cannot pass the await condition.

Initial: wantp = False, wantq = False, turn = d.c.

Process P :

```

loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False
      or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop

```

Process Q :

```

loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False
      or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop

```

The Peterson Algorithm is Correct (3)

Lemma

Peterson's algorithm is starvation-free.

Proof:

Assume this is false. $\implies$ A process gets stuck at `await`. Other process:

- ❶ It remains in its remaining code forever. $\implies$ Its `want` variable is false. Impossible.
- ❷ It also remains in its initialization code forever. Impossible, since the `await` condition is always true.
- ❸ It repeatedly runs through its critical section. Impossible, since after the first run, it sets the `turn` variable in the waiting process's favour when executing the initialization code for the second time.

Initial: `wantp = False, wantq = False, turn = d.c.`

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False
      or turn = 2
(P5)  critical section
(P6)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False
      or turn = 1
(Q5)  critical section
(Q6)  wantq := False;
end loop
```

- Variation of Peterson's algorithm
- Published in 1982 by Joep L. W. Kessels (Dutch computer scientist)
- Advantage over Dekker's/Peterson's algorithm: Single-writer, multiple-reader algorithm:
 - shared memory locations are only written to by one process (but read by several)
 - this makes it easy to implement it in message-passing models

Idea for Modifying Peterson's Algorithm

- Peterson uses a shared variable `turn`, which both processes read and write
- Idea: Use two variables `turnp` and `turnq`, so that
 - Only process P writes to `turnp`
 - Only process Q writes to `turnq`
 - The actual value of `turn` is calculated as:

$\text{turn} = 1, \quad \text{if } \text{turnp} = \text{turnq}$

$\text{turn} = 2, \quad \text{if } \text{turnp} \neq \text{turnq}$

Kessels' Algorithm

Initial: wantp = False, wantq = False, turnp, turnq d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  localp := turnq;
(P4)  turnp := localp;
(P5)  await
      wantq = False
      or localp ≠ turnq
(P6)  critical section
(P7)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  localq := if turnp = 1
                then 2
                else 1;
(Q4)  turnq := localq;
(Q5)  await
      wantp = False
      or localq = turnp
(Q6)  critical section
(Q7)  wantq := False;
end loop
```

Kessels' Algorithm

Initial: wantp = False, wantq = False, turnp, turnq d.c.

Process P :

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  localp := turnq;
(P4)  turnp := localp;
(P5)  await
      wantq = False
      or localp  $\neq$  turnq
(P6)  critical section
(P7)  wantp := False;
end loop
```

Process Q :

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  localq := if turnp = 1
               then 2
               else 1;
(Q4)  turnq := localq;
(Q5)  await
      wantp = False
      or localq = turnp
(Q6)  critical section
(Q7)  wantq := False;
end loop
```

Note: Only P writes to turnp and only Q writes to turnq

Kessels' Algorithm

Initial: wantp = False, wantq = False, turnp, turnq d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  localp := turnq;
(P4)  turnp := localp;
(P5)  await
      wantq = False
      or localp ≠ turnq
(P6)  critical section
(P7)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  localq := if turnp = 1
               then 2
               else 1;
(Q4)  turnq := localq;
(Q5)  await
      wantp = False
      or localq = turnp
(Q6)  critical section
(Q7)  wantq := False;
end loop
```

P attempts to set turnp such that turnp = turnq and thus gives *Q* priority

Kessels' Algorithm

Initial: wantp = False, wantq = False, turnp, turnq d.c.

Process P:

```
loop forever
(P1)  remaining code
(P2)  wantp := True;
(P3)  localp := turnq;
(P4)  turnp := localp;
(P5)  await
      wantq = False
      or localp ≠ turnq
(P6)  critical section
(P7)  wantp := False;
end loop
```

Process Q:

```
loop forever
(Q1)  remaining code
(Q2)  wantq := True;
(Q3)  localq := if turnp = 1
               then 2
               else 1;
(Q4)  turnq := localq;
(Q5)  await
      wantp = False
      or localq = turnp
(Q6)  critical section
(Q7)  wantq := False;
end loop
```

Q attempts to set turnq such that turnp ≠ turnq and thus gives *P* priority

Theorem

Kessels' algorithm guarantees mutual exclusion and is starvation-free.

Proof: Analogous to Peterson's algorithm, although more case distinctions are necessary (since there are more steps).

Summary: Mutual Exclusion for 2 Processes

- Mutual exclusion problem: fill the initialization code + exit code so that mutual exclusion and deadlock freedom apply.
- Distinction between deadlock freedom and starvation freedom.
- Three algorithms for two processes: Dekker, Peterson, Kessels
- Model: Only reading and writing to shared memory is atomic

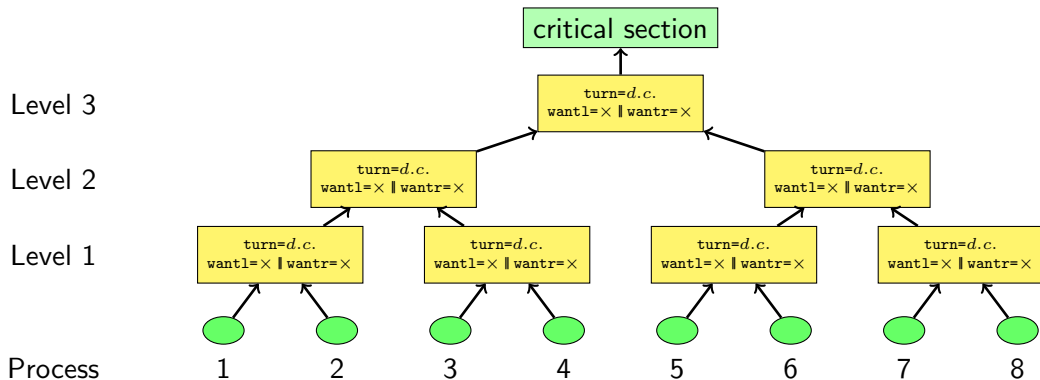
Tournament Algorithms

A simple solution for mutual exclusion of n processes



- Wanted: Algorithms that work for n processes.
- Assumption: n is known and remains constant.
- Simple idea:
Use the algorithms for 2 processes “tree-like”

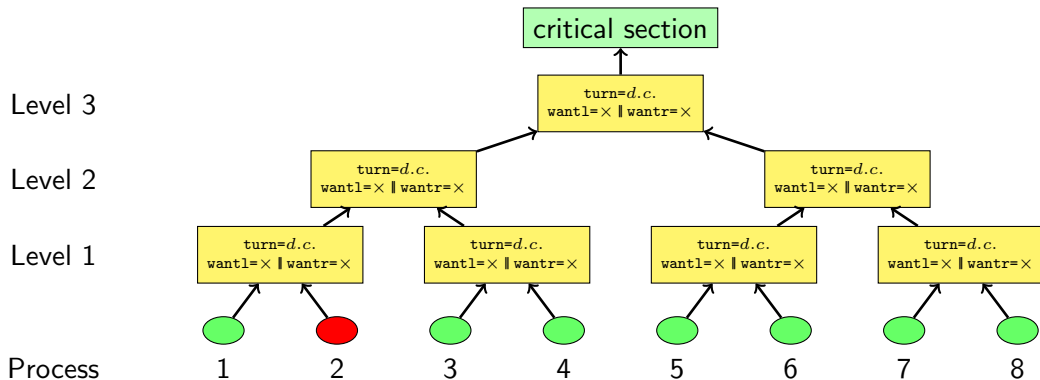
Tournament Algorithm (based on Peterson's Alg.)



turn: 2 = left child may pass, 1 = right child may pass

wantl = want of the left child, wantr = want of the right child

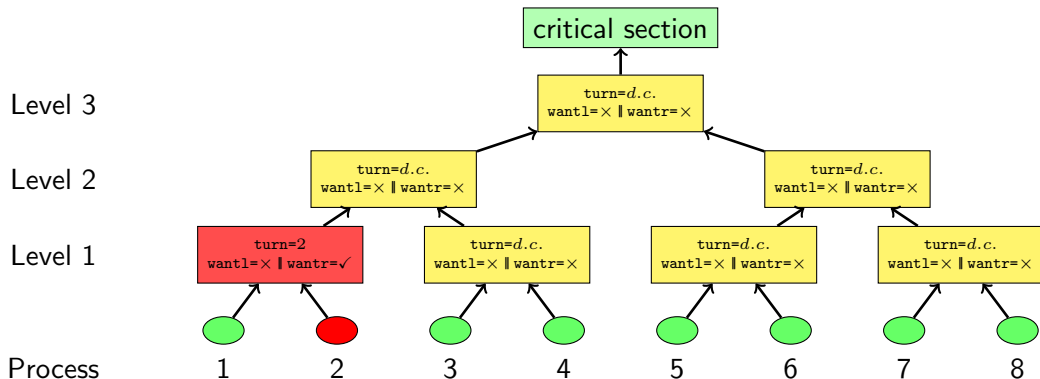
Tournament Algorithm (based on Peterson's Alg.)



turn: 2 = left child may pass, 1 = right child may pass

wantl = want of the left child, wantr = want of the right child

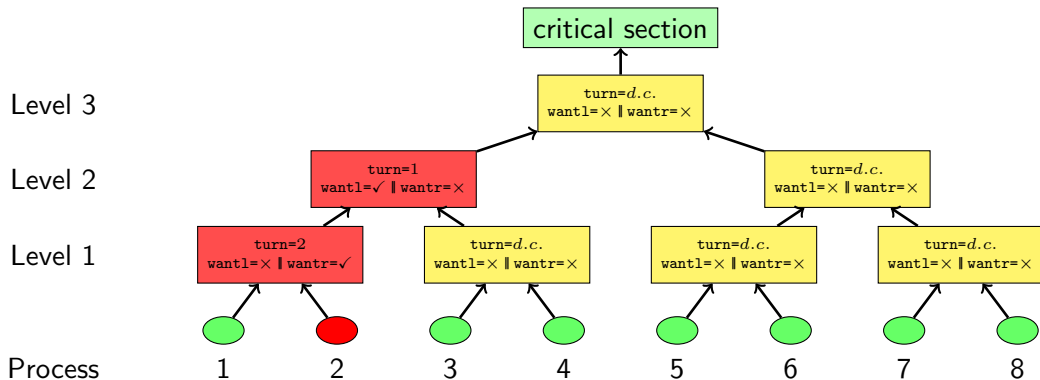
Tournament Algorithm (based on Peterson's Alg.)



turn: 2 = left child may pass, 1 = right child may pass

wantl = want of the left child, wantr = want of the right child

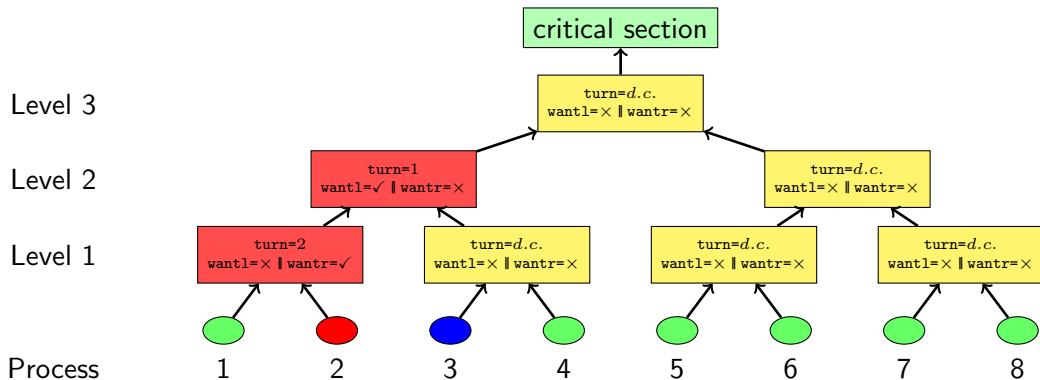
Tournament Algorithm (based on Peterson's Alg.)



turn: 2 = left child may pass, 1 = right child may pass

wantl = want of the left child, wantr = want of the right child

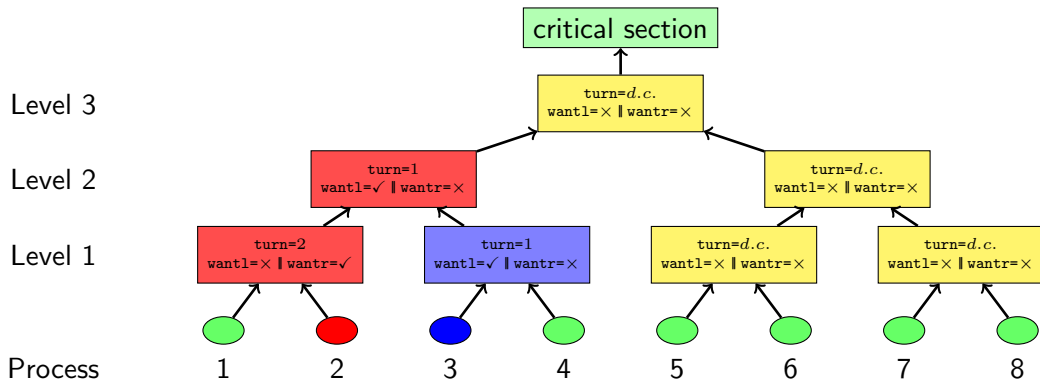
Tournament Algorithm (based on Peterson's Alg.)



turn: 2 = left child may pass, 1 = right child may pass

wantl = want of the left child, wantr = want of the right child

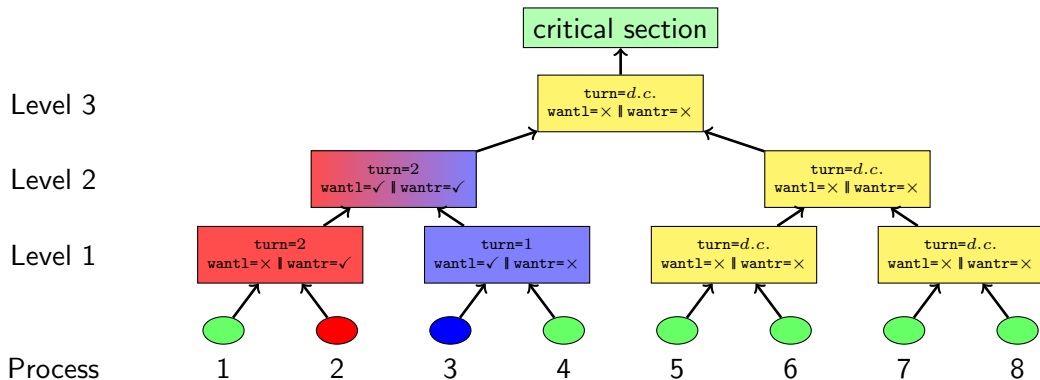
Tournament Algorithm (based on Peterson's Alg.)



turn: 2 = left child may pass, 1 = right child may pass

wantl = want of the left child, wantr = want of the right child

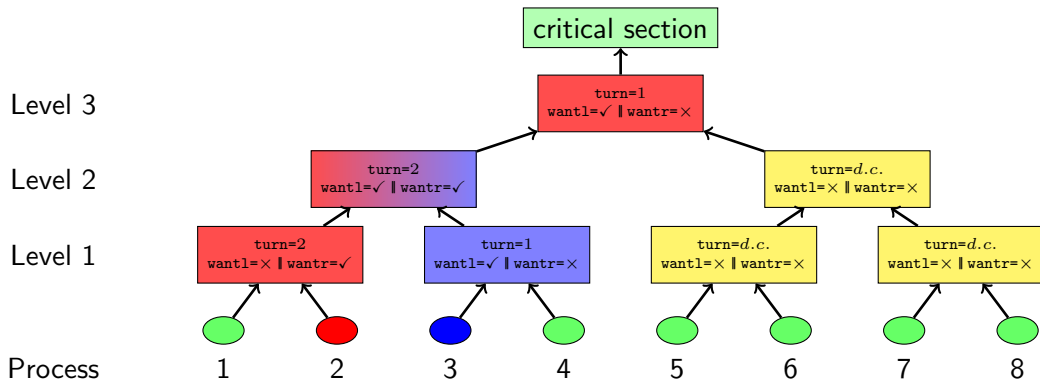
Tournament Algorithm (based on Peterson's Alg.)



turn: 2 = left child may pass, 1 = right child may pass

wantl = want of the left child, wantr = want of the right child

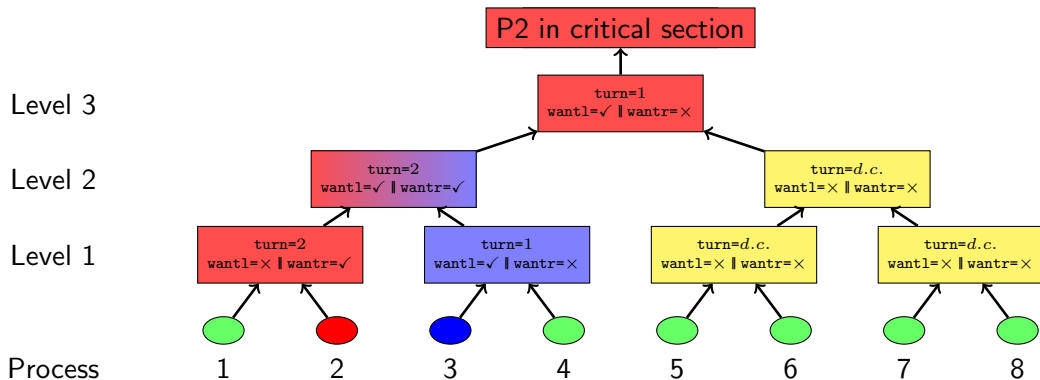
Tournament Algorithm (based on Peterson's Alg.)



turn: 2 = left child may pass, 1 = right child may pass

wantl = want of the left child, wantr = want of the right child

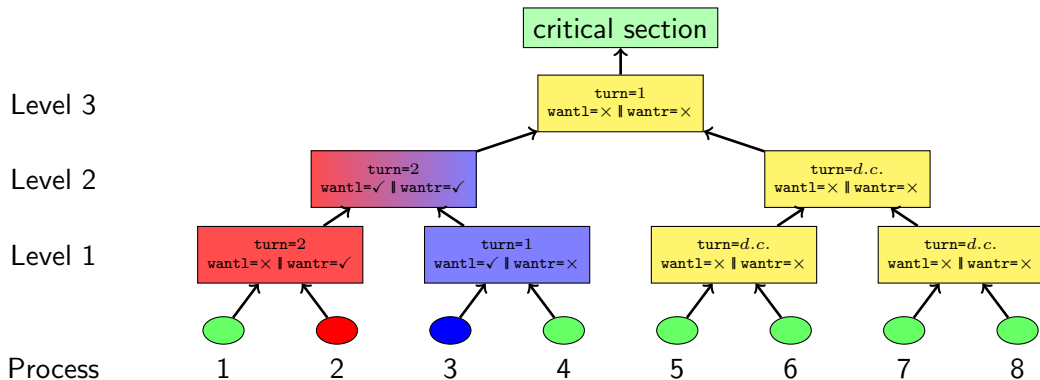
Tournament Algorithm (based on Peterson's Alg.)



turn: 2 = left child may pass, 1 = right child may pass

wantl = want of the left child, wantr = want of the right child

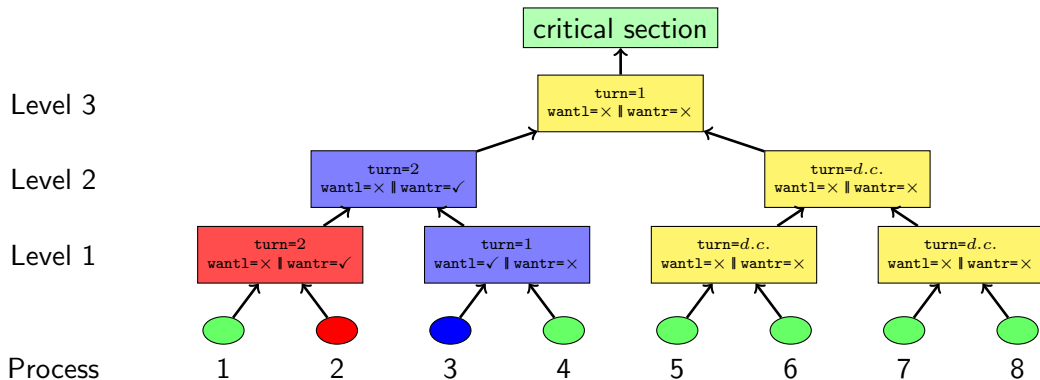
Tournament Algorithm (based on Peterson's Alg.)



turn: 2 = left child may pass, 1 = right child may pass

wantl = want of the left child, wantr = want of the right child

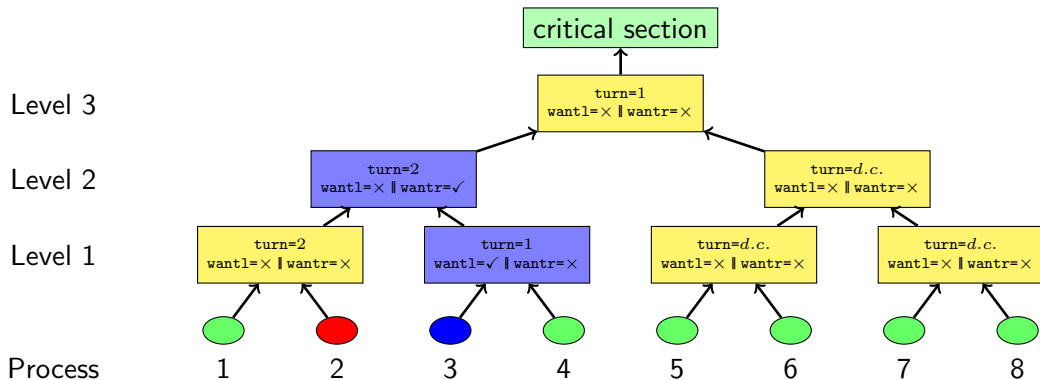
Tournament Algorithm (based on Peterson's Alg.)



turn: 2 = left child may pass, 1 = right child may pass

wantl = want of the left child, wantr = want of the right child

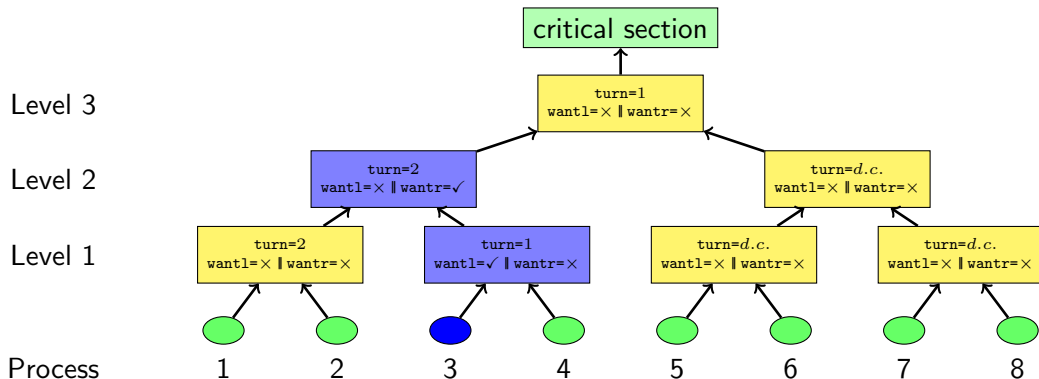
Tournament Algorithm (based on Peterson's Alg.)



turn: 2 = left child may pass, 1 = right child may pass

wantl = want of the left child, wantr = want of the right child

Tournament Algorithm (based on Peterson's Alg.)



turn: 2 = left child may pass, 1 = right child may pass

wantl = want of the left child, wantr = want of the right child

- Easy to implement
- Mutual exclusion, deadlock-free, and starvation-free, if the algorithm for 2 processes has these properties.
- Disadvantage: $\lceil \log_2 n \rceil$ -time execution of the initialization and exit code
Even if **only a single process** wants to enter the critical section!

Lamport's Algorithm



- Leslie Lamport (American computer scientist, *1941, Turing Award 2013)
- Algorithm published in 1987
- Fast algorithm for N processes
- fast = If a single process wants to enter the critical section, then the runtime is constant

Lamport's Algorithm: Program of Process i

Initial: $y=0$, for $i = 1, \dots, n$: $\text{want}[i]=\text{False}$, x any val

loop forever

(1) remaining code

(2) $\text{want}[i] := \text{True}$;

(3) $x := i$;

(4) if $y \neq 0$ then

(5) $\text{want}[i] := \text{False}$;

(6) await $y = 0$;

(7) goto (2);

(8) $y := i$;

(9) if $x \neq i$ then

(10) $\text{want}[i] := \text{False}$;

(11) for $j := 1$ to n do

 await $\neg \text{want}[j]$;

(12) if $y \neq i$ then

(13) await $y = 0$;

(14) goto (2);

(15) critical section

(16) $y := 0$;

(17) $\text{want}[i] := \text{False}$;

end loop

Lamport's Algorithm: Program of Process i

hsrm*

Initial: $y=0$, for $i = 1, \dots, n$: $\text{want}[i]=\text{False}$, x any val

loop forever

(1) remaining code

(2) $\text{want}[i] := \text{True}$;

(3) $x := i$;

(4) if $y \neq 0$ then

(5) $\text{want}[i] := \text{False}$;

(6) await $y = 0$;

(7) goto (2);

(8) $y := i$;

(9) if $x \neq i$ then

(10) $\text{want}[i] := \text{False}$;

(11) for $j := 1$ to n do
 await $\neg \text{want}[j]$;

(12) if $y \neq i$ then

(13) await $y = 0$;

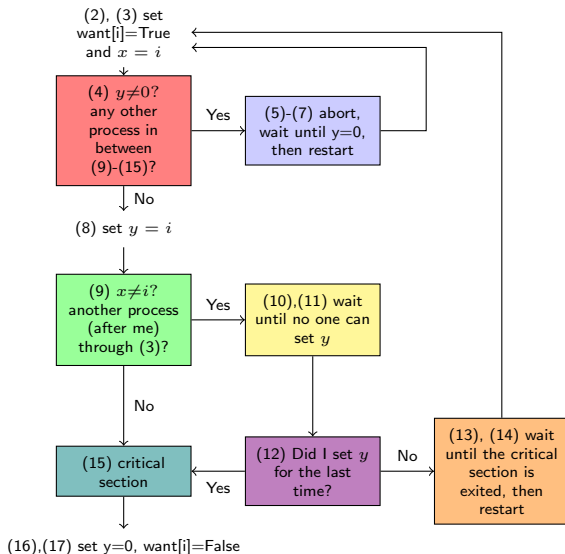
(14) goto (2);

(15) critical section

(16) $y := 0$;

(17) $\text{want}[i] := \text{False}$;

end loop



Lamport's Algorithm: Program of Process i

hsrm*

Initial: $y=0$, for $i = 1, \dots, n$: $\text{want}[i]=\text{False}$, x any val

loop forever

(1) remaining code

(2) $\text{want}[i] := \text{True}$;

Two ways to reach the critical section!

(6) await $y = 0$;

(7) goto (2);

(8) $y := i$;

(9) if $x \neq i$ then

(10) $\text{want}[i] := \text{False}$;

(11) for $j := 1$ to n do
 await $\neg \text{want}[j]$;

(12) if $y \neq i$ then

(13) await $y = 0$;

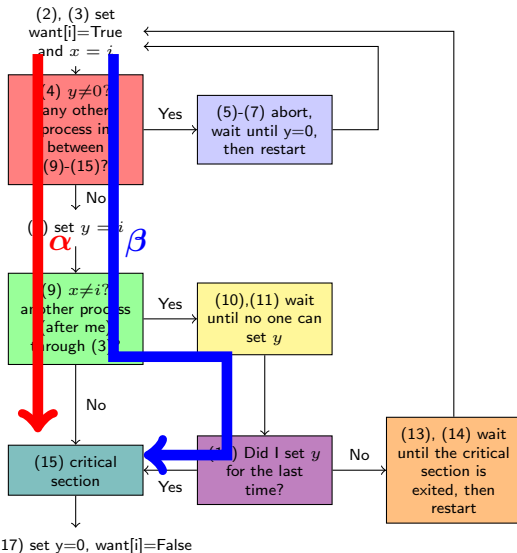
(14) goto (2);

(15) critical section

(16) $y := 0$;

(17) $\text{want}[i] := \text{False}$;

end loop



Theorem

The following applies to Lamport's algorithm:

- If a single process wants to enter the critical section, it only performs a constant number of operations.
- Lamport's algorithm is not starvation-free.

Proof: Follows directly from the algorithm/flowchart.

Theorem

Lamport's algorithm guarantees mutual exclusion.

We omit the full proof:

The ideas are

- Process i can enter the CS, if $y = i$ and $\text{want}[j] = \text{False}$, for all other processes j
- Thus y decides which process is the “first” one.
- The commands $x := i$ and the check $x \neq i$ ensure that not all processes think they are the first one

Lamport's Algorithm is Deadlock-Free

Theorem

Lamport's algorithm is deadlock-free.

Proof ideas:

- If $y = 0$, then one process can proceed.
- If $y \neq 0$ then there is a process before me.
- The first process can proceed into the CS.

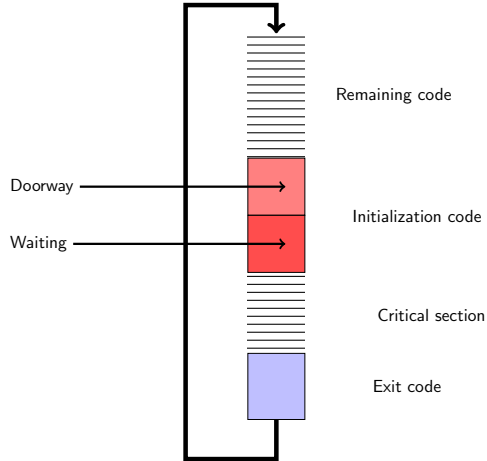
In summary:

- Fulfills mutual exclusion
- Fulfills deadlock freedom
- Constant number of operations if only one process wants to enter the critical section
- Does not fulfill starvation freedom

Bakery Algorithm

- Remember: **Starvation-free**: If a process wants to enter its critical section, then it must enter it after a finite number of computation steps.
- Does not say anything about **how long** a process has to wait.
- **Waiting** process: Process actively waits until another process stops waiting
- Example: Waiting at a `await` until another process changes the value so that the condition becomes true.
- Split the initialization code into: **Doorway** and **Waiting**

Bounded Waiting (2)



Definition

A mutual exclusion algorithm satisfies

r-bounded waiting, if the following applies to every process *i*: If process *i* leaves the doorway before process *j* (with $j \neq i$) enters the doorway, then process *j* enters the critical section at most *r* times before process *i* enters the critical section.

bounded waiting, if there is an $r \in \mathbb{N}$ such that the algorithm satisfies *r*-bounded waiting.

the FIFO property, if it satisfies 0-bounded waiting. (FIFO = first-in-first-out)

- Bounded waiting does **not** imply deadlock freedom!
- Lamport's algorithm does not satisfy bounded waiting!
- FIFO property informally: The first process to cross the doorway is the first to enter the critical section.

Bakery Algorithm

- Fulfills the FIFO property (0-bounded waiting)
- Idea of the algorithm:
- At the doorway, the process "draws" a number that is greater than any other number assigned.
- The process with the smallest number is allowed into the critical section.
- The method was probably used in bakeries (hence the name).



Bakery algorithm: Simple variant

Initial: for $i = 1, \dots, n$ $\text{number}[i] = 0$

Program of the i -th process

loop forever

(1) remaining code

(2) $\text{number}[i] := 1 + \max(\text{number})$;

(3) for $j := 1$ to n do

(4) await $\text{number}[j] = 0$ or $(\text{number}[j], j) \geq_{lex} (\text{number}[i], i)$

(5) critical section

(6) $\text{number}[i] := 0$

end loop

Bakery algorithm: Simple variant

Initial: for $i = 1, \dots, n$ $\text{number}[i] = 0$

Program of the i -th process

loop forever

(1) remaining code

(2) $\text{number}[i] := 1 + \max(\text{number})$;

(3) for $j := 1$ to n do

(4) await $\text{number}[j] = 0$ or $(\text{number}[j], j) \geq_{lex} (\text{number}[i], i)$

(5) critical section

(6) $\text{number}[i] := 0$

end loop

- Doorway, Waiting

Bakery algorithm: Simple variant

Initial: for $i = 1, \dots, n$ $\text{number}[i] = 0$

Program of the i -th process

loop forever

(1) remaining code

(2) $\text{number}[i] := 1 + \max(\text{number})$;

(3) for $j := 1$ to n do

(4) await $\text{number}[j] = 0$ or $(\text{number}[j], j) \geq_{lex} (\text{number}[i], i)$

(5) critical section

(6) $\text{number}[i] := 0$

end loop

- Doorway, Waiting
- $\geq_{lex}$ = lexicographic order

Bakery algorithm: Simple variant

Initial: for $i = 1, \dots, n$ $\text{number}[i] = 0$

Program of the i -th process

loop forever

(1) remaining code

(2) $\text{number}[i] := 1 + \text{max}(\text{number})$;

(3) for $j := 1$ to n do

(4) await $\text{number}[j] = 0$ or $(\text{number}[j], j) \geq_{lex} (\text{number}[i], i)$

(5) critical section

(6) $\text{number}[i] := 0$

end loop

- Doorway, Waiting
- $\geq_{lex}$ = lexicographic order
- **Assumption**: Maximum calculation and assignment are **atomic**

Example

Program of the 1st process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[j]=0 or
      (number[j],j)  $\geq_{lex}$  (number[1],1)
(5) critical section
(6) number[i] := 0
end loop
```

Program of the 2nd process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[j]=0 or
      (number[j],j)  $\geq_{lex}$  (number[2],2)
(5) critical section
(6) number[i] := 0
end loop
```

number[1]	number[2]
0	0

Example

Program of the 1st process

```
loop forever
(1) remaining code
(2)  $\text{number}[i] := 1 + \max(\text{number});$ 
(3) for  $j:=1$  to 2 do
(4)   await
       $\text{number}[j]=0$  or
       $(\text{number}[j],j) \geq_{lex} (\text{number}[1],1)$ 
(5) critical section
(6)  $\text{number}[i] := 0$ 
end loop
```

Program of the 2nd process

```
loop forever
(1) remaining code
(2)  $\text{number}[i] := 1 + \max(\text{number});$ 
(3) for  $j:=1$  to 2 do
(4)   await
       $\text{number}[j]=0$  or
       $(\text{number}[j],j) \geq_{lex} (\text{number}[2],2)$ 
(5) critical section
(6)  $\text{number}[i] := 0$ 
end loop
```

number[1]	number[2]
1	0

Example

Program of the 1st process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[j]=0 or
      (number[j],j)  $\geq_{lex}$  (number[1],1)
(5) critical section
(6) number[i] := 0
end loop
```

Program of the 2nd process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[j]=0 or
      (number[j],j)  $\geq_{lex}$  (number[2],2)
(5) critical section
(6) number[i] := 0
end loop
```

number[1]	number[2]
1	2

Example

Program of the 1st process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[j]=0 or
      (number[j],j)  $\geq_{lex}$  (number[1],1)
(5) critical section
(6) number[i] := 0
end loop
```

Program of the 2nd process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[j]=0 or
      (number[j],j)  $\geq_{lex}$  (number[2],2)
(5) critical section
(6) number[i] := 0
end loop
```

number[1]	number[2]
1	2

Example

Program of the 1st process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[j]=0 or
      (number[j],j)  $\geq_{lex}$  (number[1],1)
(5) critical section
(6) number[i] := 0
end loop
```

Program of the 2nd process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[j]=0 or
      (number[j],j)  $\geq_{lex}$  (number[2],2)
(5) critical section
(6) number[i] := 0
end loop
```

number[1]	number[2]
1	2

Example

Program of the 1st process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[j]=0 or
      (number[j],j)  $\geq_{lex}$  (number[1],1)
(5) critical section
(6) number[i] := 0
end loop
```

Program of the 2nd process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[j]=0 or
      (number[j],j)  $\geq_{lex}$  (number[2],2)
(5) critical section
(6) number[i] := 0
end loop
```

number[1]	number[2]
1	2

Example

Program of the 1st process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[1]=0 or
      (number[1],1)  $\geq_{lex}$  (number[1],1)
(5) critical section
(6) number[i] := 0
end loop
```

Program of the 2nd process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[j]=0 or
      (number[j],j)  $\geq_{lex}$  (number[2],2)
(5) critical section
(6) number[i] := 0
end loop
```

number[1]	number[2]
1	2

Example

Program of the 1st process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[2]=0 or
      (number[2],1)  $\geq_{lex}$  (number[1],1)
(5) critical section
(6) number[i] := 0
end loop
```

Program of the 2nd process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[j]=0 or
      (number[j],j)  $\geq_{lex}$  (number[2],2)
(5) critical section
(6) number[i] := 0
end loop
```

number[1]	number[2]
1	2

Example

Program of the 1st process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[j]=0 or
      (number[j],j)  $\geq_{lex}$  (number[1],1)
(5) critical section
(6) number[i] := 0
end loop
```

Program of the 2nd process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[j]=0 or
      (number[j],j)  $\geq_{lex}$  (number[2],2)
(5) critical section
(6) number[i] := 0
end loop
```

number[1]	number[2]
1	2

Example

Program of the 1st process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[j]=0 or
      (number[j],j)  $\geq_{lex}$  (number[1],1)
(5) critical section
(6) number[i] := 0
end loop
```

Program of the 2nd process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[1]=0 or
      (number[1],1)  $\geq_{lex}$  (number[2],2)
(5) critical section
(6) number[i] := 0
end loop
```

number[1]	number[2]
1	2

Example

Program of the 1st process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[j]=0 or
      (number[j],j)  $\geq_{lex}$  (number[1],1)
(5) critical section
(6) number[i] := 0
end loop
```

Program of the 2nd process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[1]=0 or
      (number[1],1)  $\geq_{lex}$  (number[2],2)
(5) critical section
(6) number[i] := 0
end loop
```

number[1]	number[2]
0	2

Example

Program of the 1st process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[j]=0 or
      (number[j],j)  $\geq_{lex}$  (number[1],1)
(5) critical section
(6) number[i] := 0
end loop
```

Program of the 2nd process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[j]=0 or
      (number[j],j)  $\geq_{lex}$  (number[2],2)
(5) critical section
(6) number[i] := 0
end loop
```

number[1]	number[2]
0	2

Example

Program of the 1st process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[j]=0 or
      (number[j],j)  $\geq_{lex}$  (number[1],1)
(5) critical section
(6) number[i] := 0
end loop
```

Program of the 2nd process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[j]=0 or
      (number[j],j)  $\geq_{lex}$  (number[2],2)
(5) critical section
(6) number[i] := 0
end loop
```

number[1]	number[2]
0	2

Example

Program of the 1st process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[j]=0 or
      (number[j],j) ≥lex (number[1],1)
(5) critical section
(6) number[i] := 0
end loop
```

Program of the 2nd process

```
loop forever
(1) remaining code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)   await
      number[j]=0 or
      (number[j],j) ≥lex (number[2],2)
(5) critical section
(6) number[i] := 0
end loop
```

number[1]	number[2]
0	0

Bakery algorithm: Simple variant, problem with (2)

Initial: for $i = 1, \dots, n$ $\text{number}[i] = 0$

Program of the i -th process

```
loop forever
(1)  remaining code
(2)   $\text{number}[i] := 1 + \max(\text{number})$ ;
(3)  for  $j := 1$  to  $n$  do
(4)      await  $\text{number}[j] = 0$  or  $(\text{number}[j], j) \geq_{lex} (\text{number}[i], i)$ 
(5)  critical section
(6)   $\text{number}[i] := 0$ 
end loop
```

If $\max(\text{number})$ is calculated atomically, but the assignment $\text{number}[i] := 1 + \max(\text{number})$ is only performed in the next step:

- Processes i and j with $i < j$ can set $\text{number}[i]$ and $\text{number}[j]$ to the same value.
- Mutual exclusion does not apply: j executes (4) while $\text{number}[i] = 0$, and i executes (4) afterwards.

- Simple Bakery Algorithm satisfies mutual exclusion, starvation freedom, and FIFO property.
- But: Assumption about atomic maximum is too strong!
- Therefore: Extended Bakery Algorithm

Extended Bakery Algorithm

Initial: for $i = 1, \dots, n$ $\text{number}[i]=0$, $\text{choosing}[i]=\text{False}$

Program of the i -th process

```
loop forever
(1)  remaining code
(2)  choosing[i] := True;
(3)   $\text{number}[i] := 1 + \text{maximum}(\text{number});$ 
(4)  choosing[i] := False;
(5)  for  $j:=1$  to  $n$  do
(6)      await choosing[j]=False;
(7)      await  $\text{number}[j]=0$  or  $(\text{number}[j], j) \geq_{lex} (\text{number}[i], i);$ 
(8)  critical section
(9)   $\text{number}[i] := 0$ 
end loop
```

- choosing marks entry into and exit from the doorway
- await query ensures that numbers are only compared when number is set "correctly."

Computation of the maximum:

- (1) $\text{max} := 0$
- (2) for $j := 1$ to n do
- (3) $\text{current} := \text{number}[j]$;
- (4) if $\text{max} < \text{current}$ then $\text{max} := \text{current}$
- (5) $\text{number}[i] := 1 + \text{max}$;

Lemma 1

If the value of $\text{number}[k]$ is not changed while process i calculates the maximum, then the value $\text{number}[i]$ is subsequently greater than the value of $\text{number}[k]$.

Correctness of the Bakery Algorithm

Notation:

loop forever

(1) remaining code

(2) choosing[i] := True;

(3) number[i] := 1+maximum(number);

(4) choosing[i] := False;

(5) for j:=1 to n do

(6) await choosing[j]=False;

(7) await number[j]=0 or
 (number[j],j) $\geq_{lex}$ (number[i],i);

(8) critical section

(9) number[i] := 0

end loop

Doorway



Bakery



Lemma II

If process i and process k are both in the bakery and i enters the bakery before k enters the doorway, then $\text{number}[i] < \text{number}[k]$.

Proof: Obvious from Lemma I. Because process i calculates its value, while process k cannot change its number value (except to set it to 0).

```
loop forever
(1) remaining code
(2) choosing[i] := True;
(3) number[i] := 1+maximum(number);
(4) choosing[i] := False;
(5) for j:=1 to n do
(6)     await choosing[j]=False;
(7)     await number[j]=0 or
        (number[j].j)  $\geq_{lex}$  (number[i].i);
(8) critical section
(9) number[i] := 0
end loop
```

Correctness of the Bakery Algorithm (2)

Lemma III

If process i is in the critical section and process k is in the bakery, then $(\text{number}[i], i) <_{lex} (\text{number}[k], k)$

```
loop forever
(1) remaining code
(2) choosing[i] := True;
(3) number[i] := 1+maximum(number);
(4) choosing[i] := False;
(5) for j:=1 to n do
(6)     await choosing[j]=False;
(7)     await number[j]=0 or
        (number[j].j)  $\geq_{lex}$  (number[i].i);
(8) critical section
(9) number[i] := 0
end loop
```

Proof:

- Let T_6^i be the step in which process i passes through the await statement for $j = k$ in line (6) (i.e., process i reads $\text{choosing}[k]$ for the last time).
- Similarly, let T_7^i be the step in which process i reads $\text{number}[k]$ for the last time.
- It must hold that T_6^i precedes T_7^i (noted as $T_6^i < T_7^i$).
- For process k , let T_2^k, T_3^k, T_4^k be the steps in which it last executed program lines (2), (3), (4), respectively. It must hold that $T_2^k < T_3^k < T_4^k$.

Correctness of the Bakery Algorithm (3)

Lemma III

If process i is in the critical section and process k is in the bakery, then $(\text{number}[i], i) <_{lex} (\text{number}[k], k)$

```
loop forever
(1)  remaining code
(2)  choosing[i] := True;
(3)  number[i] := 1+maximum(number);
(4)  choosing[i] := False;
(5)  for j:=1 to n do
(6)    await choosing[j]=False;
(7)    await number[j]=0 or
        (number[j],j)  $\geq_{lex}$  (number[i],i);
(8)  critical section
(9)  number[i] := 0
end loop
```

- In step T_6^i , choosing[k] had the value False, and in steps T_2^k , T_3^k , and T_4^k , choosing[k] had the value True.
- This results in two cases
 - 1 $T_6^i < T_2^k$: From Lemma II, it follows that $\text{number}[i] < \text{number}[k]$
 - 2 $T_4^k < T_6^i$: Then $T_3^k < T_4^k < T_6^i < T_7^i$ applies, i.e., at the point in time when process i executes line (7) and reads $\text{number}[k]$, $\text{number}[k] \neq 0$ applies (it was set at point in time T_3^k !). Since the await condition in line (7) is true, $(\text{number}[i], i) <_{lex} (\text{number}[k], k)$ must have been true.

Correctness of the Bakery Algorithm (4)

Theorem

The extended Bakery algorithm guarantees mutual exclusion, is starvation-free, and satisfies the FIFO property.

Proof:

- Mutual exclusion follows directly from Lemma III, since otherwise a contradiction can be derived.
- The FIFO property follows from Lemmas II and III.

```
loop forever
(1)  remaining code
(2)  choosing[i] := True;
(3)  number[i] := 1+maximum(number);
(4)  choosing[i] := False;
(5)  for j:=1 to n do
(6)      await choosing[j]=False;
(7)      await number[j]=0 or
           (number[j],j)  $\geq_{lex}$  (number[i],i);
(8)  critical section
(9)  number[i] := 0
end loop
```

Correctness of the Bakery Algorithm (5)

Theorem

The extended Bakery algorithm guarantees mutual exclusion, is starvation-free, and satisfies the FIFO property.

```
loop forever
(1)  remaining code
(2)  choosing[i] := True;
(3)  number[i] := 1+maximum(number);
(4)  choosing[i] := False;
(5)  for j:=1 to n do
(6)      await choosing[j]=False;
(7)      await number[j]=0 or
          (number[j].j)  $\geq_{lex}$  (number[i].i);
(8)  critical section
(9)  number[i] := 0
end loop
```

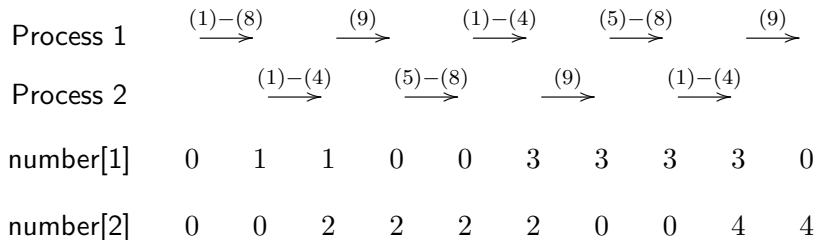
Proof:

- Deadlock freedom: proof by contradiction.
 - infinitely long sequence of computations such that no process enters the critical section
 - but at least one process is left in the bakery
 - then: there is a point in time at which no process is left in the Doorway.
 - From this point on, a process must be able to run through the for loop.
- Starvation-freedom follows from deadlock-freedom and the FIFO property.

Disadvantages of the Bakery Algorithm

- Algorithm **not fast**: If only one process wants to enter the critical section, it must execute $O(n)$ steps in the initialization code.

Disadvantages of the Bakery Algorithm (2)



- Maximum is always calculated before $\text{number}[i]$ is reset to 0.
- This means that the algorithm requires arbitrarily large numbers in the number array.
- There are variants (Black-White-Bakery algorithm) that can manage with a limited number of numbers.

So far...

- Mutual exclusion problem with n processes
- Tournament algorithms (simple but not fast)
- Lamport's algorithm (fast, but no bounded waiting)
- Bakery algorithm (FIFO property = 0-bounded waiting, not fast)
- Outlook: Is there a better way?
How many shared variables are needed?
How long must a process wait before it is allowed to enter the critical section?

Complexity Results

- Exact bound on space complexity
- A result on the runtime complexity
- The model is still: only atomic read and write commands for shared memory.

A Lower Bound for Space Complexity

Theorem

Every deadlock-free mutual exclusion algorithm for n processes requires at least n shared memory locations.

Theorem

Every deadlock-free mutual exclusion algorithm for n processes requires at least n shared memory locations.

Comments on the proof (1980 by J. Burns and N. A. Lynch):

- Simple for single-writer, multiple-reader algorithms, since every process must modify the memory before it is in the critical section, otherwise mutual exclusion would be impossible (processes must recognize whether another process is in the critical section).

Theorem

Every deadlock-free mutual exclusion algorithm for n processes requires at least n shared memory locations.

Comments on the proof (1980 by J. Burns and N. A. Lynch):

- Simple for single-writer, multiple-reader algorithms, since every process must modify the memory before it is in the critical section, otherwise mutual exclusion would be impossible (processes must recognize whether another process is in the critical section).
- The general proof is more complex. Parts of the argument are that with fewer memory locations, it is not possible to distinguish whether a process is in the remaining code or in the critical section, which refutes mutual exclusion.
- We omit the proof.

Theorem

There is a deadlock-free mutual exclusion algorithm for n processes that uses n shared bits.

Proof:

- One-bit algorithm
- both J.E.Burns in 1981 and L.Lamport in 1986

Idea of the One-Bit Algorithm

Initial: for $i = 1, \dots, n$ want[i] = False,

Program of the i -th process

```
loop forever
(1)  remaining code
(2)  want[ $i$ ] := True;
(3)  for local := 1 to  $n$  do
(4)    if local  $\neq i$  then await want[local] = False;
(5)  critical section
(6)  want[ $i$ ] := False
end loop
```

Idea of the One-Bit Algorithm

Initial: for $i = 1, \dots, n$ want[i] = False,

Program of the i -th process

```
loop forever
(1)  remaining code
(2)  want[i] := True;
(3)  for local := 1 to  $n$  do
(4)    if local  $\neq i$  then await want[local] = False;
(5)  critical section
(6)  want[i] := False
end loop
```

- Fulfills mutual exclusion:
The first process in the critical section has want set to True, every other process will read this and get stuck.
- Does **not** fulfill deadlock freedom.
- **Algorithm is wrong!**

The One-Bit Algorithm

Initial: for $i = 1, \dots, n$ $\text{want}[i] = \text{False}$,

Program of the i -th process

```
loop forever
(1) remaining code
(2) repeat
(3)    $\text{want}[i] := \text{True}$ ;
(4)    $\text{local} := 1$ ;
(5)   while ( $\text{want}[i] = \text{True}$ ) and ( $\text{local} < i$ ) do
(6)     if  $\text{want}[\text{local}] = \text{True}$  then
(7)        $\text{want}[i] := \text{False}$ ;
(8)       await  $\text{want}[\text{local}] = \text{False}$ ;
(9)        $\text{local} := \text{local} + 1$ 
(10) until  $\text{want}[i] = \text{True}$ ;
(11) for  $\text{local} := i+1$  to  $n$  do
(12)   await  $\text{want}[\text{local}] = \text{False}$ ;
(13) critical section
(14)  $\text{want}[i] := \text{False}$ 
end loop
```

The One-Bit Algorithm

Initial: for $i = 1, \dots, n$ want[i] = False,

Program of the i-th process

loop forever

(1) remaining code

(2) repeat

(3) want[i] := True;

(4) local := 1;

(5) while (want[i] = True) and (local < i) do

(6) if want[local] = True then

(7) want[i] := False;

(8) await want[local] = False;

(9) local := local + 1

(10) until want[i] = True;

(11) for local := i+1 to n do

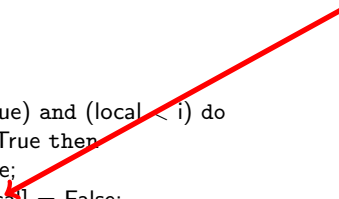
(12) await want[local] = False;

(13) critical section

(14) want[i] := False

end loop

Tests for $j = 1 \dots i - 1$



- Rough idea as before:
Test all other want values for False before entering the critical section.
- Therefore: mutual exclusion is fulfilled.

Tests for $j = i + 1 \dots n$



The One-Bit Algorithm (2)

Initial: for $i = 1, \dots, n$ want[i] = False,

Program of the i-th process

```
loop forever
(1) remaining code
(2) repeat
(3)   want[i] := True;
(4)   local := 1;
(5)   while (want[i] = True) and (local < i) do
(6)     if want[local] = True then
(7)       want[i] := False;
(8)       await want[local] = False;
(9)       local := local + 1
(10) until want[i] = True;
(11) for local := i+1 to n do
(12)   await want[local] = False;
(13) critical section
(14) want[i] := False
end loop
```

Deadlock freedom:

- Proof idea: In a deadlock computation sequence, one can conclude:
- At some point, all processes:
 - await in line (8)
 - in the for loop in lines (11)-(12)
 - or forever in the remaining code
- and: at least one process in the for loop
- process with the largest index will run through the for loop

Properties of the One-Bit Algorithm

- Guarantees mutual exclusion and deadlock freedom
- Starvation is possible
- Not symmetric: e.g., process with number 1 runs through the repeat loop immediately
- Not fast: If only one process wants to enter the CS, it must test all n bits
- But: Space-optimal, since only n bits of shared memory

Theorem (R. Alur and G. Taubenfeld, 1992)

There is no (deadlock-free) mutual exclusion algorithm for 2 (or even n) processes that has an upper bound for the number of memory accesses (of the shared memory) that a process must perform before it is allowed to enter the critical section.

- This means that processes **must** "wait" for any length of time until they are allowed to enter the critical section.
- There is no algorithm that can prevent this.
- Note: For **this** model (atomic read and write operations)!
- Result refers to all cases; there are subcases in which a bound can be specified.
- e.g.: Case in which only one process wants to enter the critical section.

Let M be a deadlock-free mutual exclusion algorithm for 2 processes P_1 and P_2

Computation tree T_M for M :

- Binary tree
- Each node corresponds to the state of execution (all assignments)
- Process-internal steps (steps that do not operate on memory) are represented as a node.
- Root: First interesting state: P_1 and P_2 immediately before entering initialization code
- Left child of a node: Successor state after a step of P_1
- Right child of a node: Successor state after a step of P_2
- Leaf: P_1 or P_2 has entered critical section (then stop)

Marking of the nodes of T_M

- Leaf is marked with exactly 1 or exactly 2, depending on which P_i is in the CS
- Inner node is marked with 1, 2, or (1 and 2), depending on how its children are marked.

Similarity

- Two nodes v, w are **similar with respect to P_i** (written $v \langle P_i \rangle w$), if
 - Steps taken by P_i from the root to v = Steps taken by P_i from the root to w
 - Shared variables and local variables of P_i are identical for v and w

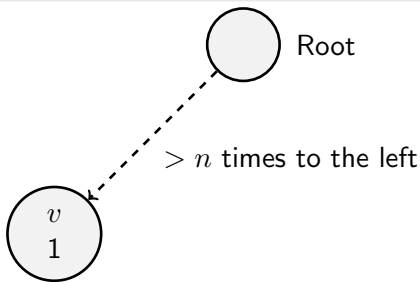
Proof (3)

The theorem is proven if:

For every $n > 0$ and $i \in \{1, 2\}$:

There is a leaf v with label i such that
on the path from the root to v

more than n steps are executed for process P_i

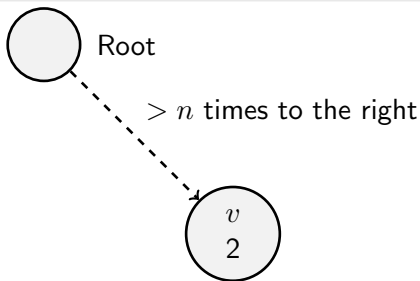


Proof (3)

The theorem is proven if:

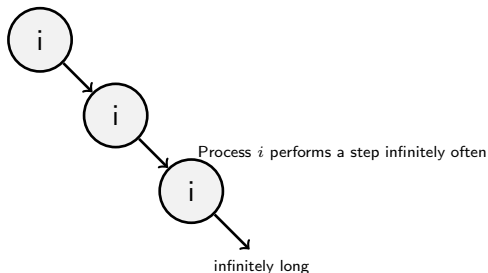
For every $n > 0$ and $i \in \{1, 2\}$:

There is a leaf v with label i such that
on the path from the root to v
more than n steps are executed for process P_i



Proof (4)

Case: There is an infinitely long path in T_M that contains infinitely many nodes, all marked with i , and process P_i performs infinitely many steps on this path.



Then: For every n , the desired path can be constructed.

Therefore: **Assumption A:** T_M has no such infinite path.

Proof (5)

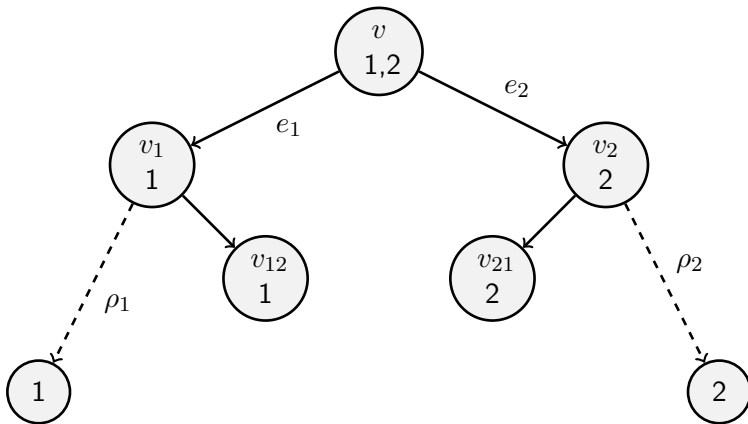
We now show: Assumption A leads to a contradiction.

Since the algorithm is deadlock-free (and assumption A holds):

There are nodes v, v_1, v_2 with

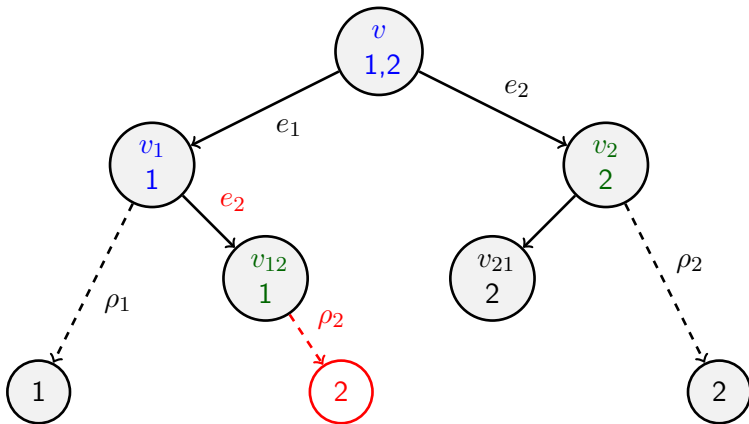
- v is marked with 1, 2
- The two nodes v_1 and v_2 are each marked with exactly one number.

Proof (6): Case 1: v_1 marked with 1, v_2 marked with 2



Assumption A $\implies$ leftmost path finite length, rightmost path finite length
(leaf marked with 1 or 2)

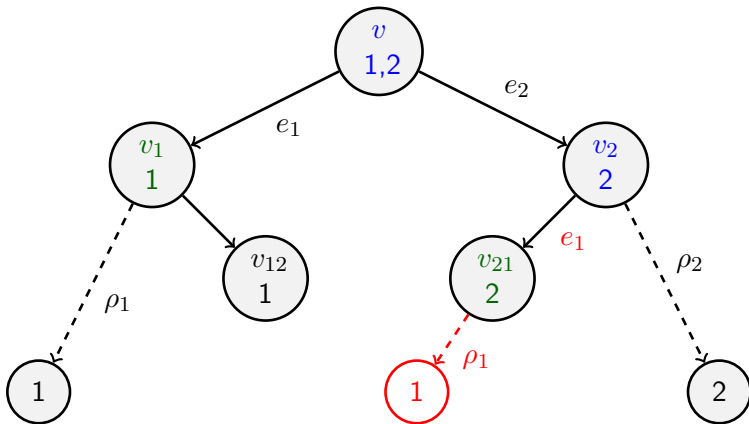
Proof (6): Case 1: v_1 marked with 1, v_2 marked with 2



e_1 read operation: Then $v \langle P_2 \rangle v_1$ and also $v_2 \langle P_2 \rangle v_{12}$ apply.

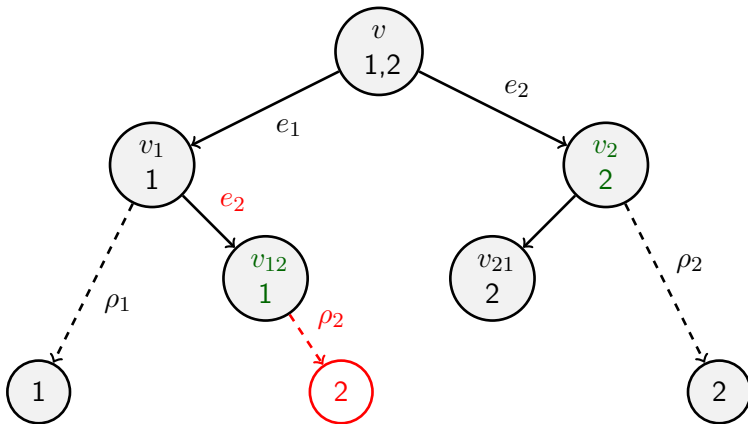
Contradiction, since ρ_2 must also lead to leaf marked with 2 for v_{12}

Proof (6): Case 1: v_1 marked with 1, v_2 marked with 2



e_2 read operation: Then $v \langle P_1 \rangle v_2$ and also $v_1 \langle P_1 \rangle v_{21}$ apply.
 Contradiction, since ρ_1 must also lead to leaf with 1 for v_{21}

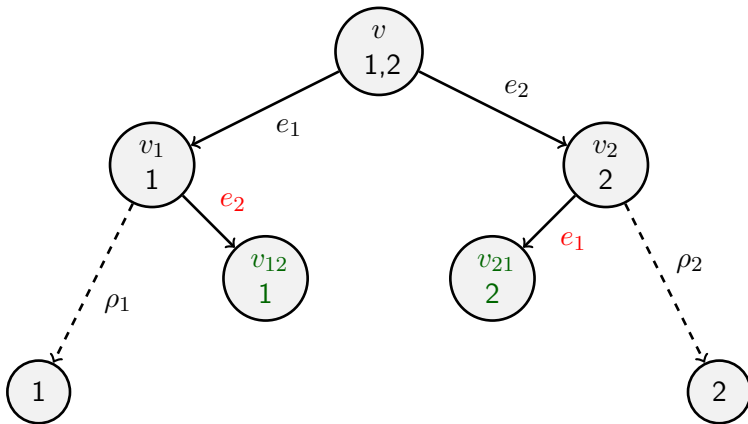
Proof (6): Case 1: v_1 marked with 1, v_2 marked with 2



e_1 and e_2 write operation on **same** variables:

Then $v_2 \langle P_2 \rangle v_{12}$ applies. Contradiction (as before)

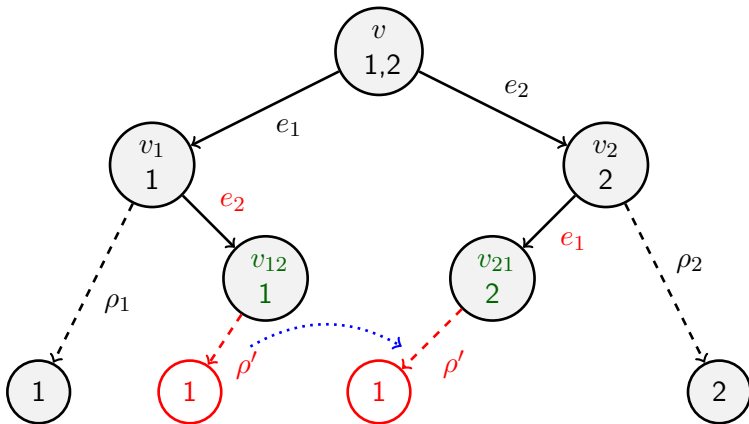
Proof (6): Case 1: v_1 marked with 1, v_2 marked with 2



e_1 and e_2 write operation on **different** variables:

Then $v_{12}\langle P_i \rangle v_{21}$ applies for $i = 1, 2$.

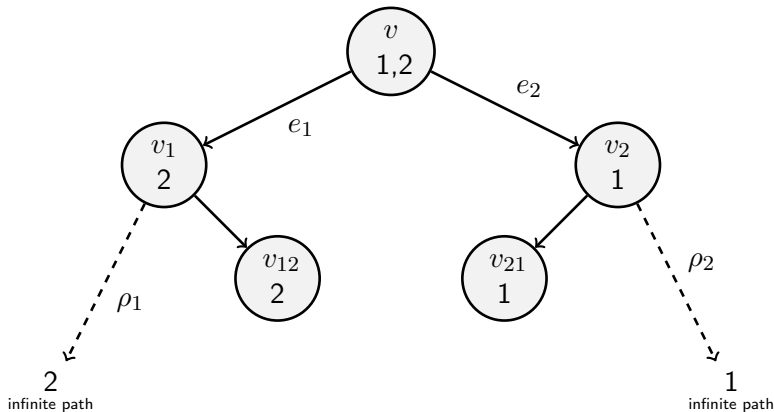
Proof (6): Case 1: v_1 marked with 1, v_2 marked with 2



e_1 and e_2 write operation on **different** variables:

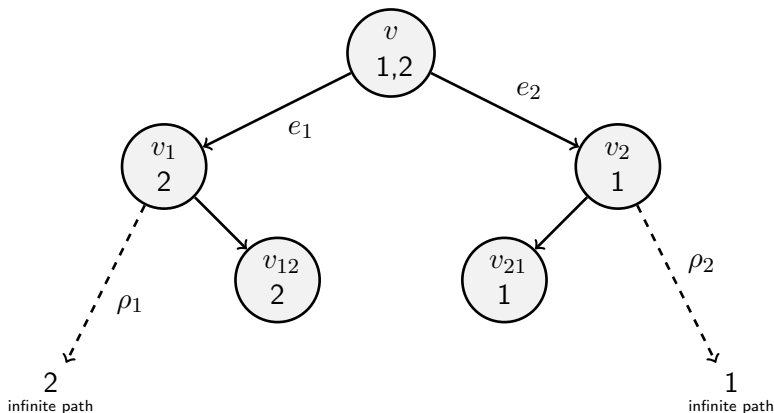
Then $v_{12}\langle P_i \rangle v_{21}$ applies for $i = 1, 2$. Contradiction: Since markings are

Proof (7): Case 2: v_1 marked with 2, v_2 marked with 1



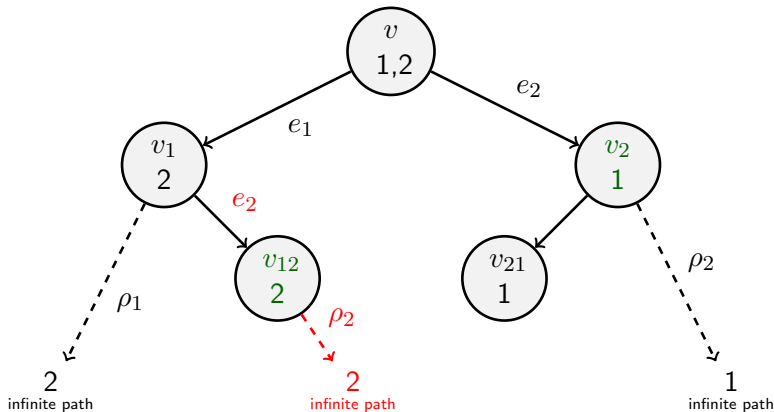
P_2 not in CS in $v \implies P_2$ not in CS in v_1
 $\implies$ infinite path far left from v_1 (since only P_1 takes steps)

Proof (7): Case 2: v_1 marked with 2, v_2 marked with 1



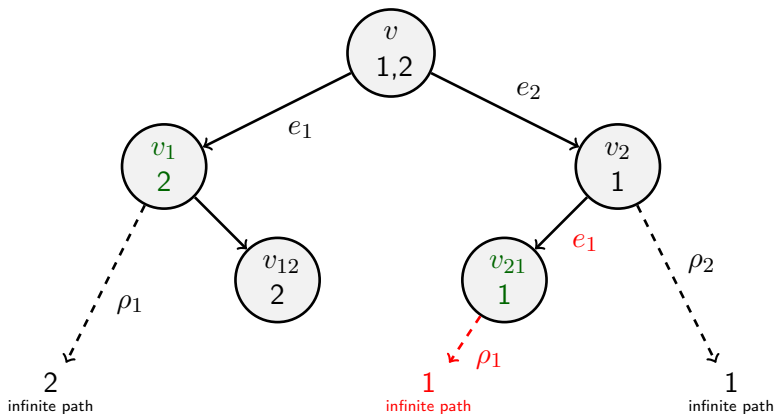
analogous: infinite path far right from v_2

Proof (7): Case 2: v_1 marked with 2, v_2 marked with 1



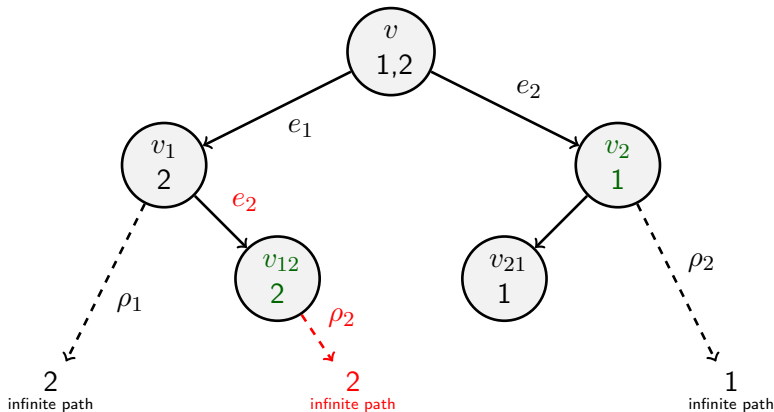
e_1 is read operation $\Rightarrow v_2 \langle P_2 \rangle v_{12}$.
 $\Rightarrow \rho_2$ can also be executed from v_{12} .
 $\Rightarrow$ infinitely long path from P_2 steps,

Proof (7): Case 2: v_1 marked with 2, v_2 marked with 1



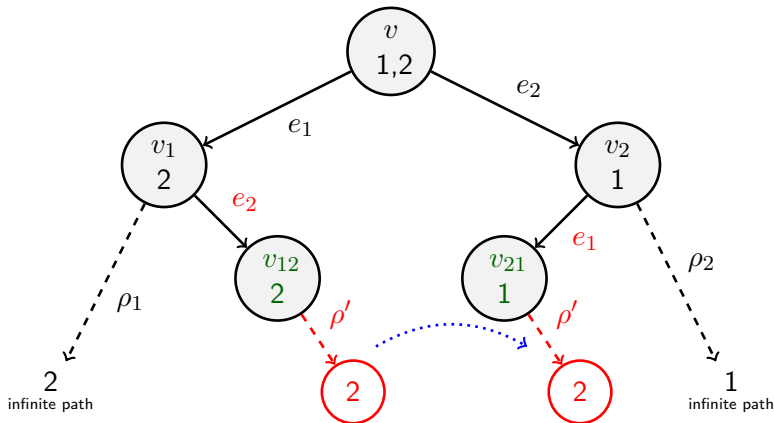
e_2 is a read operation: analogous

Proof (7): Case 2: v_1 marked with 2, v_2 marked with 1



e_1 and e_2 are write operations to the same variable. Then $v_{12}\langle P_2\rangle v_2$ applies.
Contradiction.

Proof (7): Case 2: v_1 marked with 2, v_2 marked with 1



e_1 and e_2 are write operations to different variables. Then, for $i = 1, 2$, the following applies: $v_{12}\langle P_i \rangle v_{21}$. Impossible.

Summary:

- Solving the mutual exclusion problem with only atomic read and write is not easy
- Proving properties of the algorithms and the problems is usually difficult

Outlook:

- What happens in the presence of stronger memory primitives?
- Can we classify memory primitives?