

Models of concurrency & synchronization algorithms

Lecture 3 of TDA384/DIT391

Principles of Concurrent Programming

Nir Piterman and Gerardo Schneider

Chalmers University of Technology | University of Gothenburg



UNIVERSITY OF
GOTHENBURG



CHALMERS
UNIVERSITY OF TECHNOLOGY

Today's menu

- Analyzing concurrency
- Mutual exclusion with only atomic reads and writes
 - Three **failed** attempts
 - Peterson's algorithm
 - Mutual exclusion with bounded waiting
- Implementing mutual exclusion algorithms in Java
- Implementing semaphores

Analyzing concurrency

State/transition diagrams

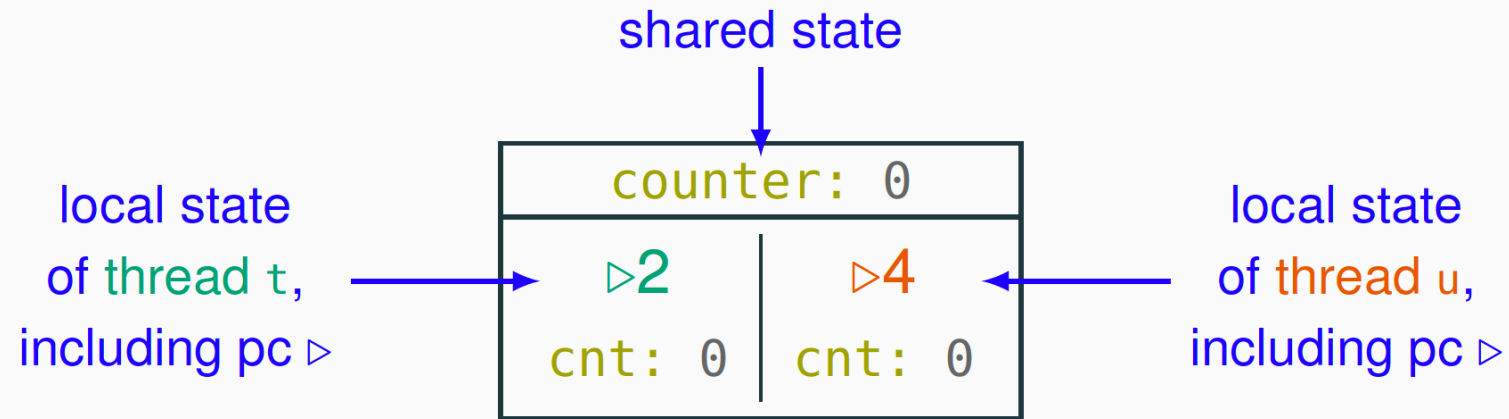
We capture essential elements of concurrent programs using **state/transition diagrams**

- Also called: *(finite) state automata*, *(finite) state machines*, or *transition systems*
- **States** in a diagram capture possible program states
- **Transitions** connect states according to execution order

Structural properties of a diagram capture semantic properties of the corresponding program

States

A **state** captures the shared and local states of a concurrent program:



`int counter = 0;`

	thread t		thread u	
	<code>int cnt;</code>		<code>int cnt;</code>	
1	<code>cnt = counter;</code>		<code>cnt = counter;</code>	3
2	<code>counter = cnt + 1;</code>		<code>counter = cnt + 1;</code>	4

States

A **state** captures the shared and local states of a concurrent program:

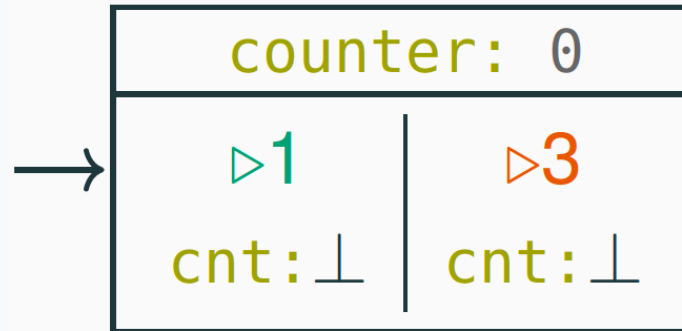
counter: 0	
▷2	▷4
cnt: 0	cnt: 0

When unambiguous, we simplify a state with only the **essential information**:

0	
▷2	▷4
0	0

Initial states

The **initial state** of a computation is marked with an incoming arrow:

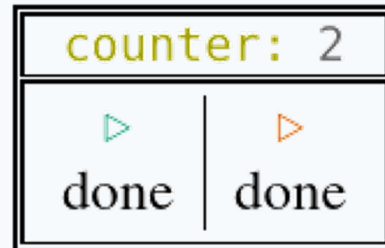


```
int counter = 0;
```

	thread t		thread u	
	int cnt;		int cnt;	
1	cnt = counter;		cnt = counter;	3
2	counter = cnt + 1;		counter = cnt + 1;	4

Final states

The **final states** of a computation – where the program terminates – are marked with double-line edges:

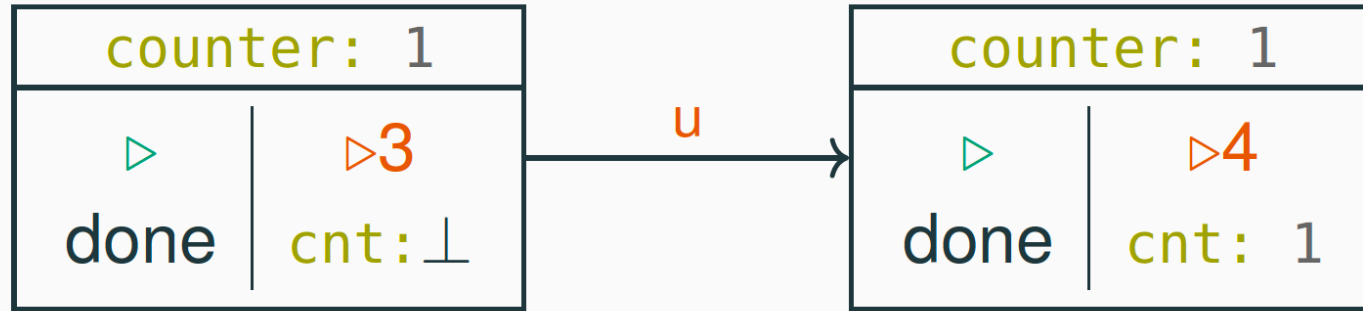


```
int counter = 0;
```

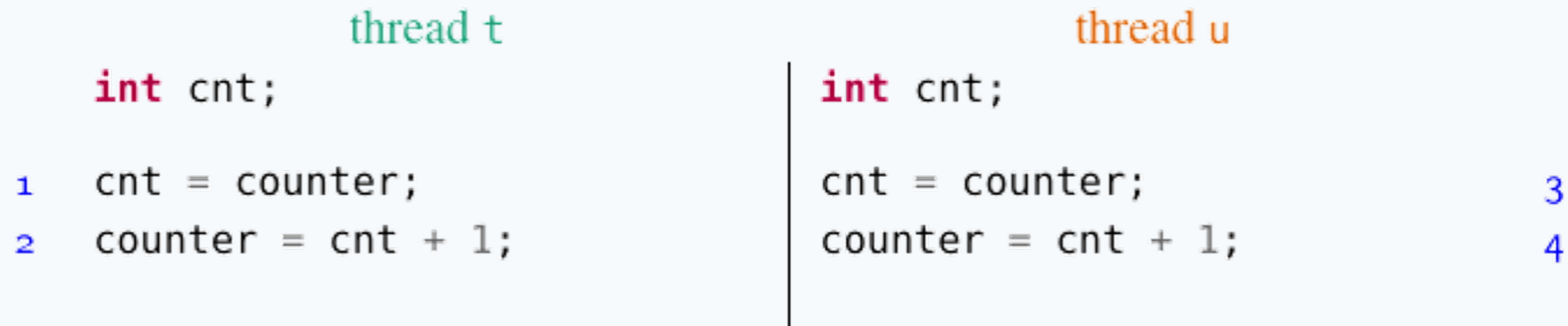
	thread t		thread u
	<pre>int cnt;</pre>		<pre>int cnt;</pre>
1	<pre>cnt = counter;</pre>		<pre>cnt = counter;</pre>
2	<pre>counter = cnt + 1;</pre>		<pre>counter = cnt + 1;</pre>
			3
			4

Transitions

A **transition** corresponds to the execution of one atomic instruction, and it is an arrow connecting two states (or a state to itself):



`int counter = 0;`



A complete state/transition

The complete state/transition diagram explicitly shows all possible interleavings

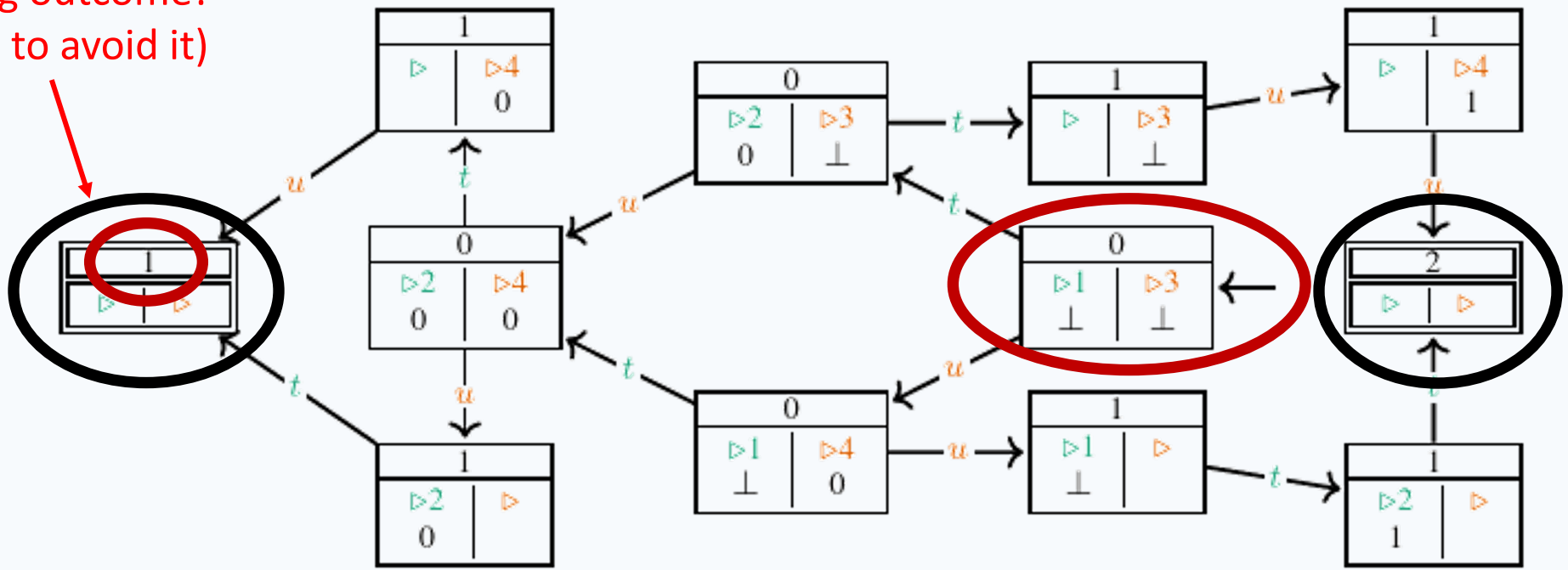
```

int counter = 0;
-----
thread t
int cnt;
1 cnt = counter;
2 counter = cnt + 1;

thread u
int cnt;
3 cnt = counter;
4 counter = cnt + 1;

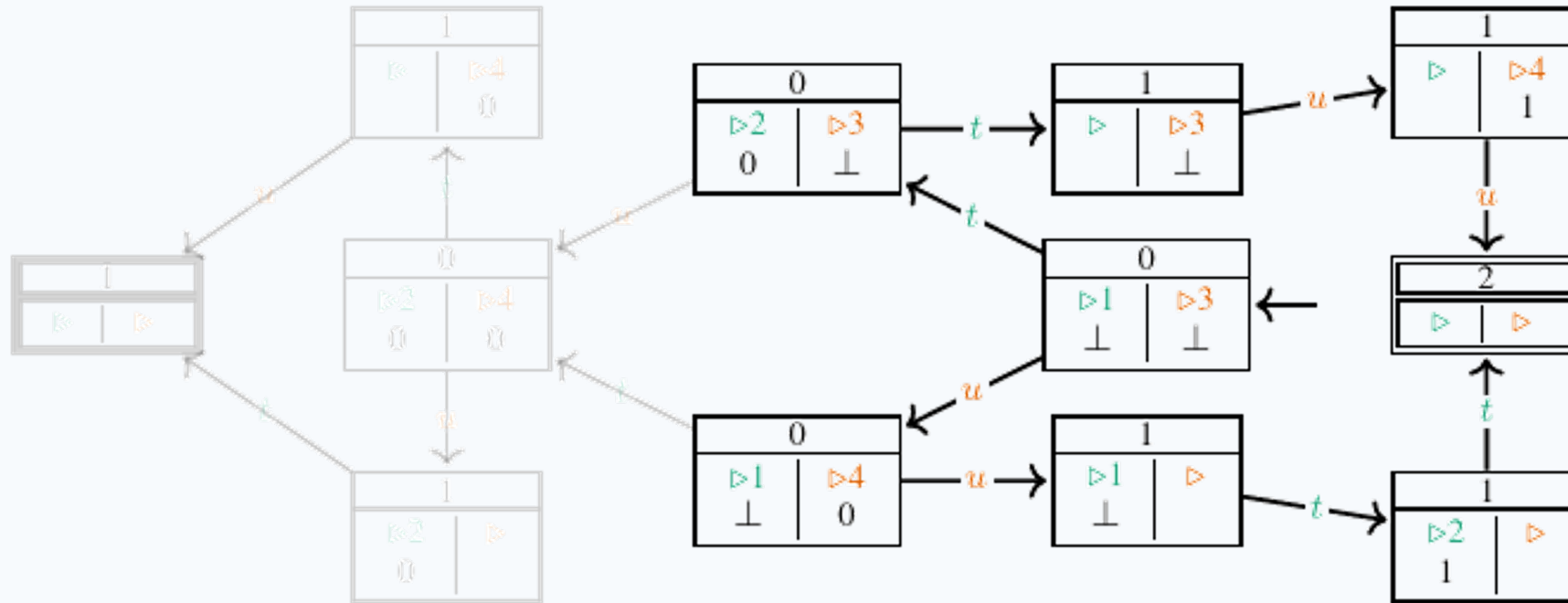
```

Wrong outcome!
(want to avoid it)



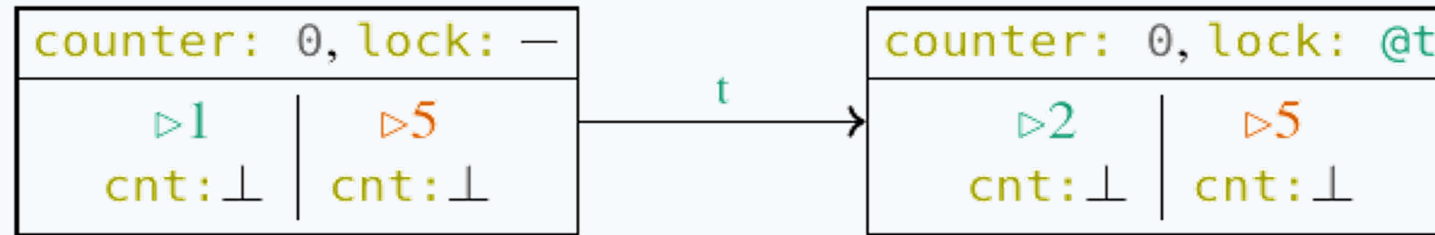
State/transition diagram with locks?

The state/transition diagram of the concurrent counter example we would like to achieve using **locks**:



Locking

Locking and unlocking are considered atomic operations



int counter = 0;	Lock lock = new Lock();
thread t	thread u
int cnt; 1 lock.lock(); 2 cnt = counter; 3 counter = cnt + 1; 4 lock.unlock();	int cnt; lock.lock(); cnt = counter; counter = cnt + 1; lock.unlock();
	5 6 7 8

This transition is only allowed if the lock is **not held by another thread**

Counter with locks: state/transition

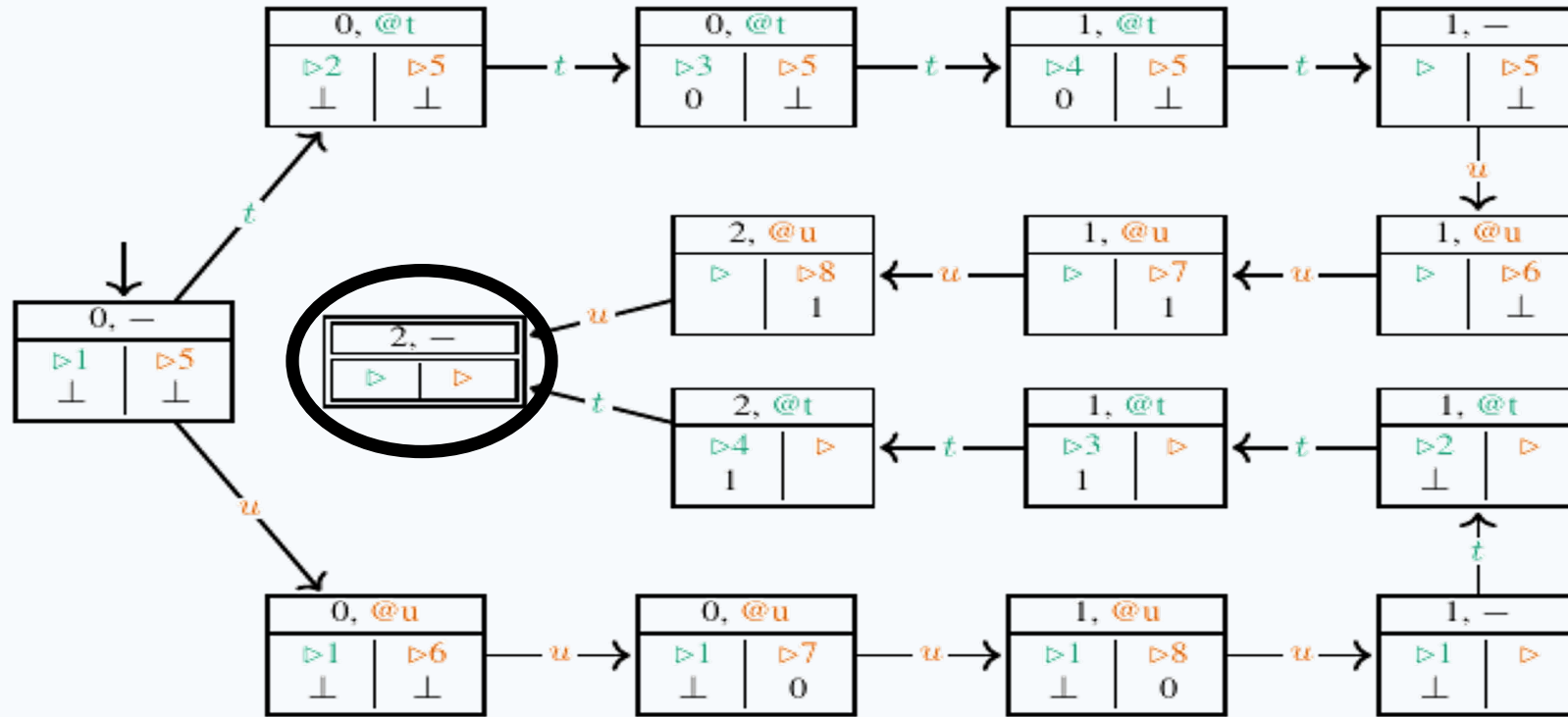
The state/transition diagram of the conc should contain **no (states representing)**

```

int counter = 0; Lock lock = new Lock();

thread t
int cnt;
1 lock.lock();
2 cnt = counter;
3 counter = cnt + 1;
4 lock.unlock();

thread u
int cnt;
5 lock.lock();
6 cnt = counter;
7 counter = cnt + 1;
8 lock.unlock();
    
```



Transition tables

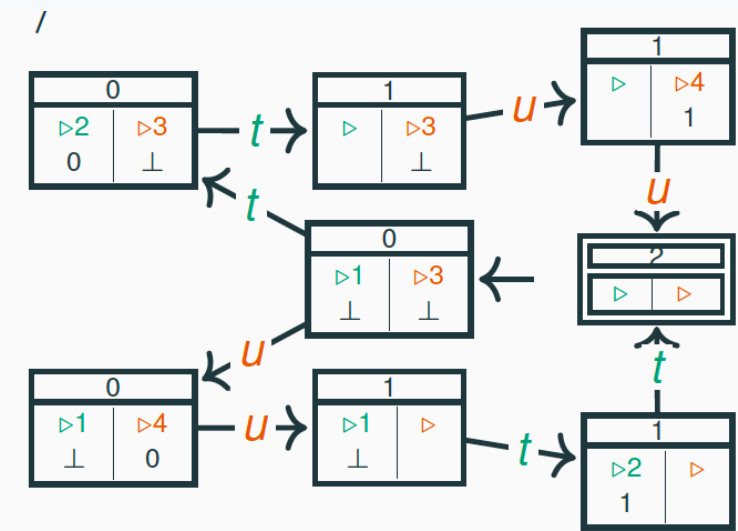
Transition tables are *equivalent representations* of state/transition diagrams

```

int counter = 0;
-----
thread t
int cnt;
1 cnt = counter;
2 counter = cnt + 1;

thread u
int cnt;
cnt = counter;
counter = cnt + 1;
  
```

CURRENT	NEXT WITH t	NEXT WITH u
$\langle 0, \triangleright 1, \perp, \triangleright 3, \perp \rangle$	$\langle 0, \triangleright 2, 0, \triangleright 3, \perp \rangle$	$\langle 0, \triangleright 1, \perp, \triangleright 4, 0 \rangle$
$\langle 0, \triangleright 2, 0, \triangleright 3, \perp \rangle$	$\langle 1, \triangleright , , \triangleright 3, \perp \rangle$	—
$\langle 0, \triangleright 1, \perp, \triangleright 4, 0 \rangle$	—	$\langle 1, \triangleright 1, \perp, \triangleright , \rangle$
$\langle 1, \triangleright , , \triangleright 3, \perp \rangle$	—	$\langle 1, \triangleright , , \triangleright 4, 1 \rangle$
$\langle 1, \triangleright 1, \perp, \triangleright , \rangle$	$\langle 1, \triangleright 2, 1, \triangleright , \rangle$	—
$\langle 1, \triangleright , , \triangleright 4, 1 \rangle$	—	$\langle 2, \triangleright , , \triangleright , \rangle$
$\langle 1, \triangleright 2, 1, \triangleright , \rangle$	$\langle 2, \triangleright , , \triangleright , \rangle$	—
$\langle 2, \triangleright , , \triangleright , \rangle$	—	—



Reasoning about program properties

The **structural properties** of a diagram capture semantic properties of the program:

Mutual exclusion: there are no states where two threads are in their critical section

Deadlock freedom: for every (non-final) state, there is an outgoing transition

Starvation freedom: there is no (looping) path such that a thread never enters its critical section while trying to do so

No race conditions: all the final states have the same (correct) result

- We will build and analyze state/transition diagrams only for simple examples, since it quickly becomes tedious
- **Model checking** is a technique that automates the construction and analysis of state/transition diagrams with billions of states
 - We'll give a short introduction to model checking in one of the last classes

Mutual exclusion with only atomic reads and writes

Locks: recap

A **lock** is a data structure (an **object** in Java) with interface:

```
interface Lock {  
    void lock();           // acquire lock  
    void unlock();        // release lock  
}
```

- Several threads share the same object **lock** of type **Lock**
- Threads calling **lock.lock()** : exactly one thread t **acquires** the lock:
 - t 's call **lock.lock()** returns: t is **holding** the lock
 - other threads **block** on the call **lock.lock()** , waiting for the lock to become available
- A thread t that is holding the lock calls **lock.unlock()** to **release** the lock:
 - t 's call **lock.unlock()** returns: the lock becomes **available**
 - another thread **waiting** for the lock may succeed in acquiring it

Mutual exclusion **without** locks

Can we implement locks using **only** atomic instructions – reading and writing shared variables?

- It is possible
- But it is also tricky!



- We present some **classical algorithms** for mutual exclusion using only **atomic reads and writes**
 - The presentation builds up to the correct algorithms in a series of attempts, which highlight the principles that underlie how the algorithms work

The mutual exclusion problem - recap

Given N threads, each executing:

```
// continuously
while (true) {
  entry protocol
  critical section {
    // access shared data
  }
  exit protocol
} /* ignore behavior
outside critical section */
```

Now protocols can use
only reads and writes
 of shared variables

Design the entry and exit protocols to **ensure**:

- mutual exclusion
- freedom from deadlock
- freedom from starvation

Initially we limit ourselves to $N = 2$ threads, t_0 and t_1

Busy waiting

In the pseudo-code, we will use the shorthand

$$\text{await}(c) \triangleq \text{while} (!c) \{ \}$$

to denote **busy waiting** (also called **spinning**):

- keep reading shared variable `c` as long as it is `false`
- proceed when it becomes `true`
- Busy waiting is generally **inefficient** (unless typical waiting times are shorter than context switching times), so you should **avoid using it**
 - We use it only because it is a good device to illustrate the nuts and bolts of mutual exclusion protocols
- Note that **await** is **not** a valid Java keyword
 - We highlight it in a different color – but we will use it as a shorthand for better readability

Mutual exclusion with only
atomic reads and writes

Three *failed* attempts

Double-threaded mutual exclusion: First naive attempt

Use Boolean flags `enter[0]` and `enter[1]`:

- each thread **waits** until the other thread is **not trying** to enter the critical section
- before thread t_k is about **to enter** the critical section, it sets `enter[k]` to true

```

      boolean[] enter = {false, false};
  
```

thread t_0		thread t_1
<pre> 1 while (true) { 2 // entry protocol 3 await (!enter[1]); 4 enter[0] = true; 5 critical section { ... } 6 // exit protocol 7 enter[0] = false; 8 } </pre>	<pre> while (true) { // entry protocol await (!enter[0]); enter[1] = true; critical section { ... } // exit protocol enter[1] = false; } </pre>	<pre> 9 10 11 12 13 14 15 16 </pre>

The first naive attempt is incorrect!

The **first attempt** does not guarantee **mutual exclusion**: t_0 and t_1 can be in the critical section at the same time

```

boolean[] enter = {false, false};

thread t0                                thread t1
1  while (true) {                          9  while (true) {
2    // entry protocol                       10 // entry protocol
3    await (!enter[1]);                       11 await (!enter[0]);
4    enter[0] = true;                          12 enter[1] = true;
5    critical section { ... }                 13 critical section { ... }
6    // exit protocol                          14 // exit protocol
7    enter[0] = false;                        15 enter[1] = false;
8  }                                           16 }
    
```

Both threads here! How?

#	t_0	t_1	SHARED
1	pc ₀ : await (!enter[1])	pc ₁ : await (!enter[0])	enter: false, false
2	pc ₀ : enter[0] = true	pc ₁ : await (!enter[0])	enter: false, false
3	pc ₀ : enter[0] = true	pc ₁ : enter[1] = true	enter: false, false
4	pc ₀ : critical section	pc ₁ : enter[1] = true	enter: true, false
5	pc ₀ : critical section	pc ₁ : critical section	enter: true, true

The **problem** seems to be that **await** is executed **before** setting enter, so one thread may proceed ignoring that the other thread is also proceeding

Double-threaded mutual exclusion: Second naive attempt

When thread t_k wants to enter the **critical section**:

- it **first** sets **enter**[k] to true
- then it **waits** until the other thread is **not trying** to enter the critical section

```

      boolean[] enter = {false, false};
  
```

thread t_0	thread t_1
<pre> 1 while (true) { 2 // entry protocol 3 enter[0] = true; 4 await (!enter[1]); 5 critical section { ... } 6 // exit protocol 7 enter[0] = false; 8 } </pre>	<pre> 9 while (true) { 10 // entry protocol 11 enter[1] = true; 12 await (!enter[0]); 13 critical section { ... } 14 // exit protocol 15 enter[1] = false; 16 } </pre>

The second naive attempt may deadlock!

The second attempt:

- **guarantees mutual exclusion**: t_0 is in the critical section iff `enter[1]` is false, iff t_1 has not set `enter[1]` to true, iff t_1 has not entered the critical section (t_1 has not executed line yet)
- does **not guarantee freedom from deadlocks**

Both threads might end up here, blocked. Why?

```

boolean[] enter = {false, false};
-----
thread t0                                thread t1
1  while (true) {                          9  while (true) {
2  // entry protocol                       10 // entry protocol
3  enter[0] = true;                         11 enter[1] = true;
4  await (!enter[1]);                       12 await (!enter[0]);
5  critical section { ... }                 13 critical section { ... }
6  // exit protocol                         14 // exit protocol
7  enter[0] = false;                       15 enter[1] = false;
8  }                                        16 }

```

#	t_0	t_1	SHARED
1	<code>pc₀: enter[0] = true</code>	<code>pc₁: enter[0] = true</code>	<code>enter: false, false</code>
1	<code>pc₀: await (!enter[1])</code>	<code>pc₁: enter[0] = true</code>	<code>enter: true, false</code>
2	<code>pc₀: await (!enter[1])</code>	<code>pc₁: await (!enter[0])</code>	<code>enter: true, true</code>

The **problem** seems to be that the **two variables** `enter[0]` and `enter[1]` are accessed independently

- each thread may be waiting for permission to proceed from the other thread

Double-threaded mutual exclusion: **Third** naive attempt

Use one single integer variable `yield`:

- thread t_k **waits** for its **turn** while `yield` is k
- when it is done with its critical section, it **yields control** to the other thread by setting `yield = k`

```
int yield = 0 || 1; // initialize to either value
```

thread t_0

```

1 while (true) {
2   // entry protocol
3   await (yield != 0);
4   critical section { ... }
5   // exit protocol
6   yield = 0;
7 }
```

thread t_1

```

8 while (true) {
9   // entry protocol
10  await (yield != 1);
11  critical section { ... }
12  // exit protocol
13  yield = 1;
14 }
```

The third naive attempt may starve some thread!

The third attempt:

- **guarantees mutual exclusion:**

t_0 is in the critical section

iff `yield` is 1

iff `yield` was initialized to 1 or t_1 has set `yield` to 1

iff t_1 is not in the critical section (t_0 has not executed line 6 yet).

- **guarantees freedom from deadlocks:** each thread enables the other thread, so that a circular wait is impossible
- **does not guarantee freedom from starvation:** if one stops executing in its **non-critical** section, the other thread will starve (after one last access to its critical section)

```
int yield = 0 || 1; // initialize to either value
```

thread t_0	thread t_1
<pre> 1 while (true) { 2 // entry protocol 3 await (yield != 0); 4 critical section { ... } 5 // exit protocol 6 yield = 0; 7 }</pre>	<pre> 8 while (true) { 9 // entry protocol 10 await (yield != 1); 11 critical section { ... } 12 // exit protocol 13 yield = 1; 14 }</pre>

Later in the course: we will discuss how model checking can help to verify whether such correctness properties hold in a concurrent program

The third naive attempt may starve some thread!

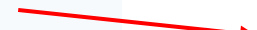
`int yield = 0 || 1; // initialize to either value`

thread t_0

```

1 while (true) {
2   // entry protocol
3   await (yield != 0);
4   critical section { ... }
5   // exit protocol
6   yield = 0;
7 }
    
```

... then thread t_0
will starve



thread t_1

```

8 while (true) {
9   // entry protocol
10  await (yield != 1);
11  critical section { ... }
12  // exit protocol
13  yield = 1;
14 }
    
```



If `yield=0` and
thread t_1 stops
executing here
(before the entry
protocol)...

Peterson's algorithm

Peterson's algorithm

Combine the ideas behind the second and third attempts:

- thread t_k first sets `enter[k]` to true
- but lets the other thread go first – by setting `yield`

```
boolean[] enter = {false, false}; int yield = 0 || 1;
```

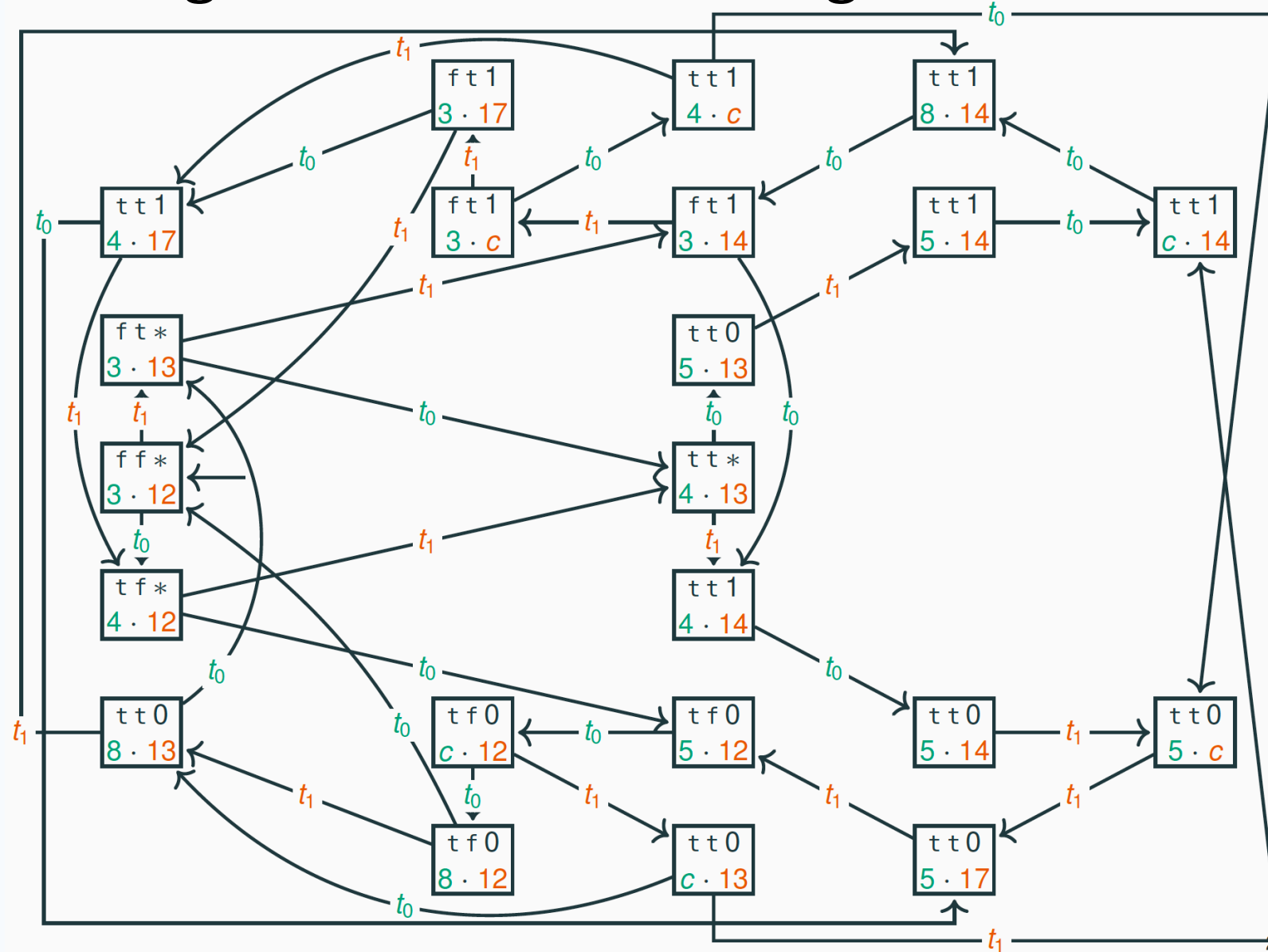
thread t_0	thread t_1
<pre> 1 while (true) { 2 // entry protocol 3 enter[0] = true; 4 yield = 0; 5 await (!enter[1] yield != 0); 6 critical section { ... } 7 // exit protocol 8 enter[0] = false; 9 } </pre>	<pre> 10 while (true) { 11 // entry protocol 12 enter[1] = true; 13 yield = 1; 14 await (!enter[0] yield != 1); 15 critical section { ... } 16 // exit protocol 17 enter[1] = false; 18 } </pre>

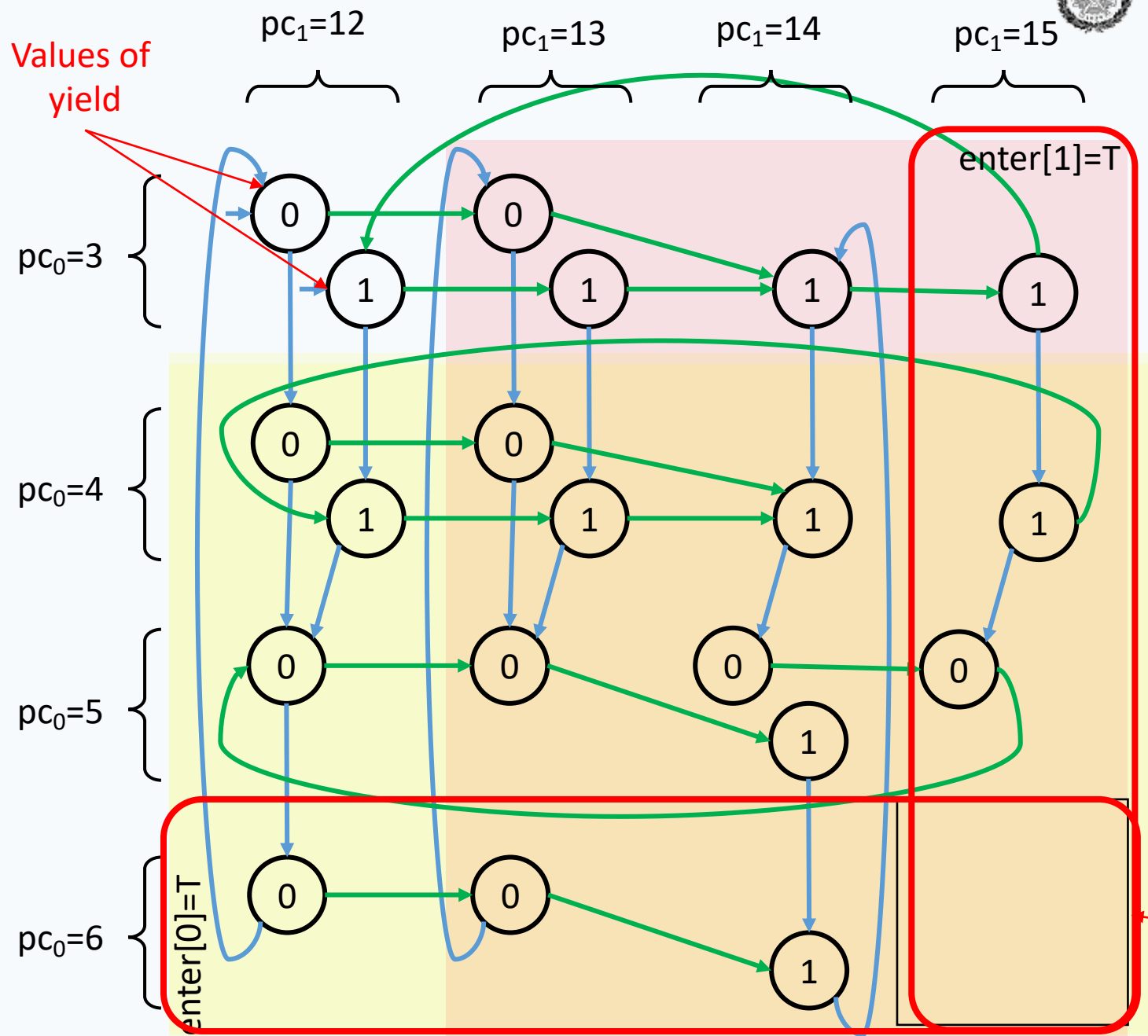
Equivalent to:
wait while
(`enter[1]=true`
&
`yield=0`)

Enter only when
(`enter[1]=false`
OR
`yield=1`)

Works even if two reads
are non-atomic

State/transition diagram of Peterson's algorithm





Another state/transition diagram of Peterson's algorithm

```

boolean[] enter = {false, false};  int yield = 0 || 1;
-----
thread t0                               thread t1
1 while (true) {                          10 while (true) {
2 // entry protocol                        11 // entry protocol
3 enter[0] = true;                         12 enter[1] = true;
4 yield = 0;                               13 yield = 1;
5 await (!enter[1] ||                      14 await (!enter[0] ||
   yield != 0);                             yield != 1);
6 critical section { ... }                 15 critical section { ... }
7 // exit protocol                         16 // exit protocol
8 enter[0] = false;                       17 enter[1] = false;
9 }                                         18 }

```

Critical Section

Omitting lines 7-9 and 16-18

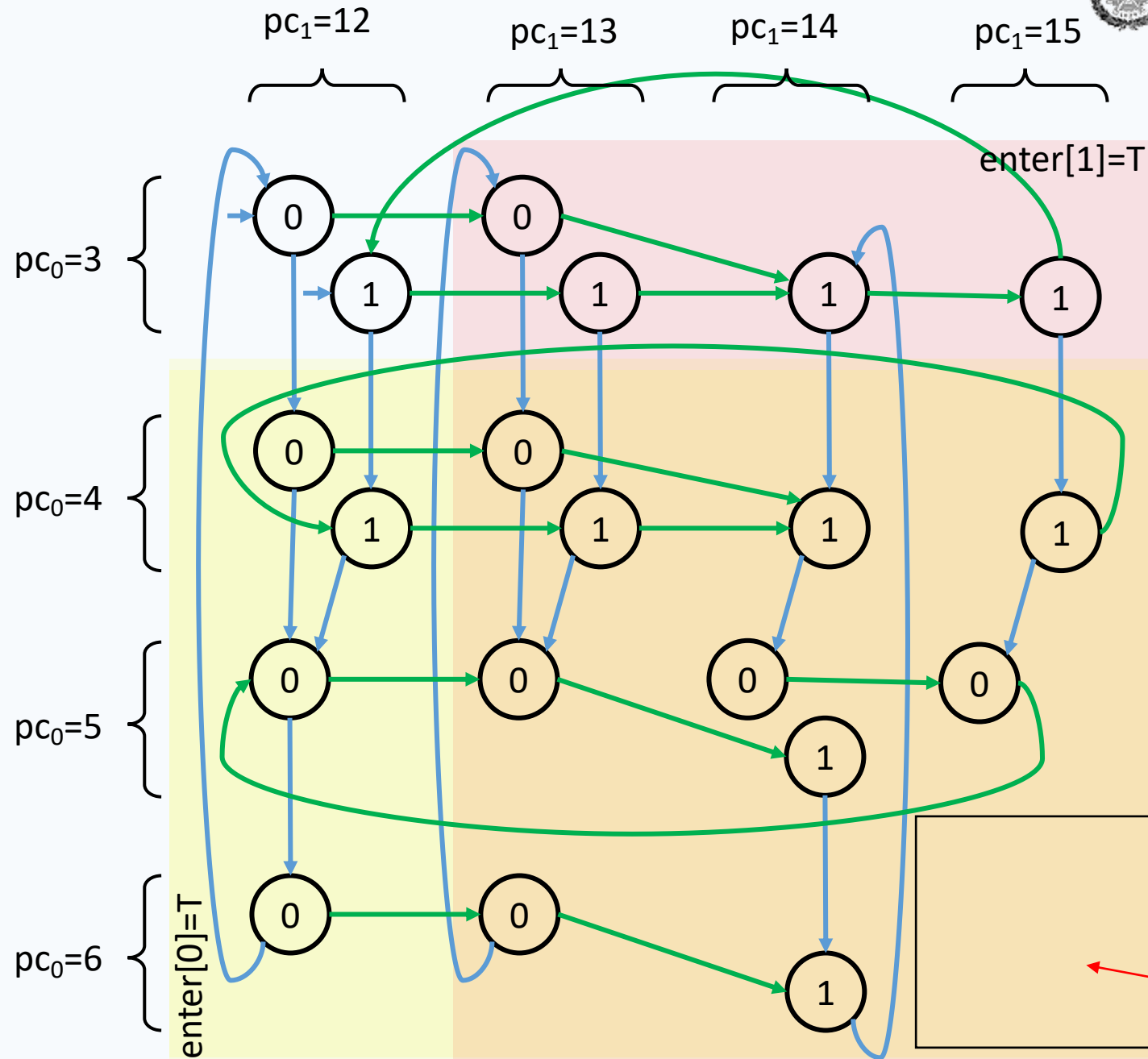
Checking the correctness of Peterson's algorithm

By inspecting the state/transition diagram, we can check that Peterson's algorithm satisfies:

mutual exclusion: there are no states where both threads are at $pc_0=6$ and $pc_1=15$ (in the critical section)

deadlock freedom: every state has at least one outgoing transition

starvation freedom: if thread t_0 is in its critical section, then thread t_1 can reach its critical section without requiring thread t_0 's collaboration after t_0 executes the exit protocol



Peterson's algorithm satisfies mutual exclusion and is deadlock free

```

boolean[] enter = {false, false};  int yield = 0 || 1;
-----
thread t0                               thread t1
1 while (true) {                          10 while (true) {
2 // entry protocol                       11 // entry protocol
3 enter[0] = true;                         12 enter[1] = true;
4 yield = 0;                               13 yield = 1;
5 await (!enter[1] || yield != 0);        14 await (!enter[0] || yield != 1);
6 critical section { ... }                15 critical section { ... }
7 // exit protocol                         16 // exit protocol
8 enter[0] = false;                       17 enter[1] = false;
9 }                                         18 }

```

Both in Critical Section

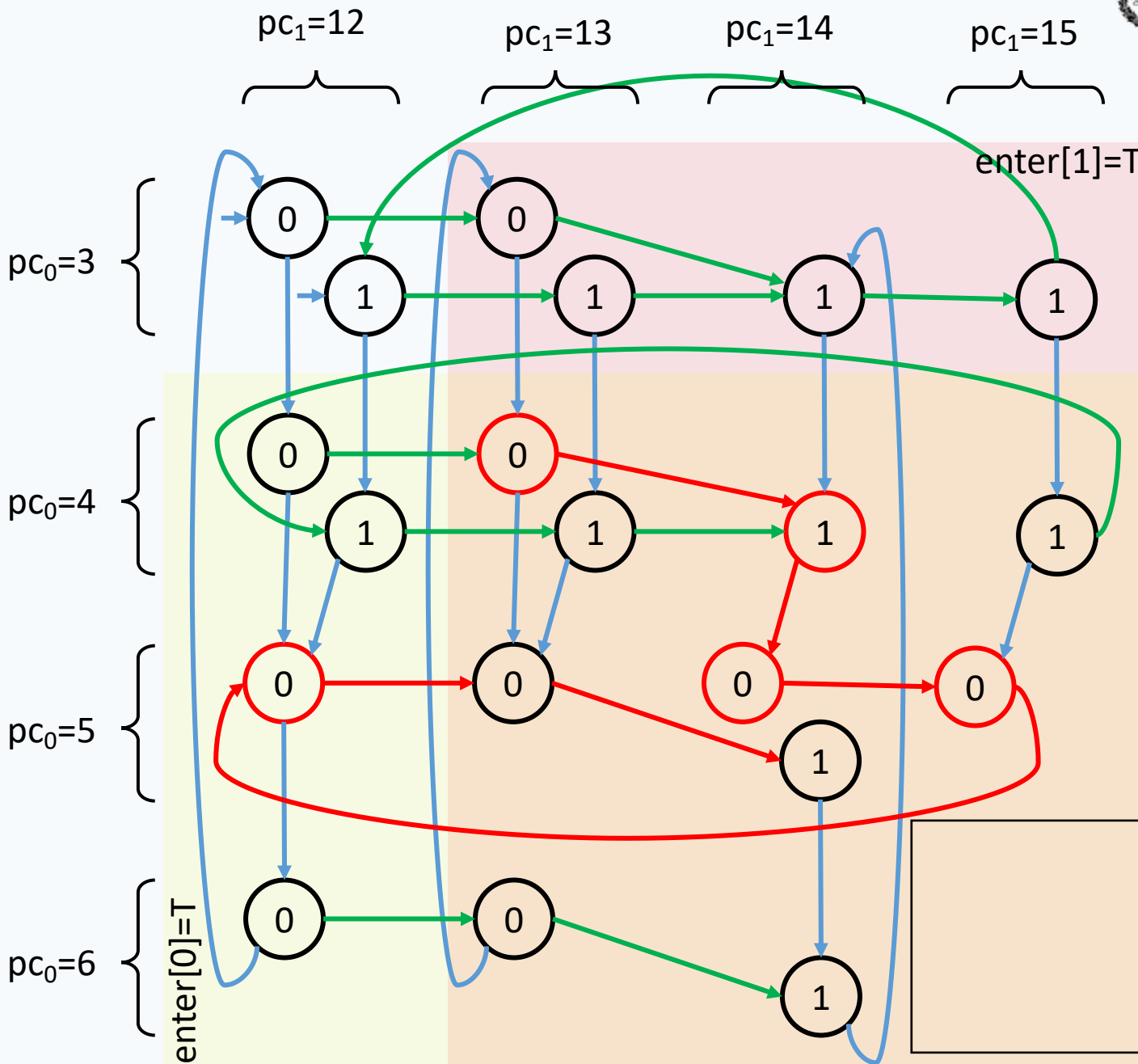
Checking the correctness of Peterson's algorithm

By inspecting the state/transition diagram, we can check that Peterson's algorithm satisfies:

mutual exclusion: there are no states where both threads are at $pc_0=6$ and $pc_1=15$ (in the critical section)

deadlock freedom: every state has at least one outgoing transition

starvation freedom: if thread t_0 is in its critical section, then thread t_1 can reach its critical section without requiring thread t_0 's collaboration after t_0 executes the exit protocol



Peterson's algorithm is starvation free

(No thread keeps waiting to enter the critical section)

```
boolean[] enter = {false, false}; int yield = 0 || 1;
```

thread t_0

thread t_1

<pre> 1 while (true) { 2 // entry protocol 3 enter[0] = true; 4 yield = 0; 5 await (!enter[1] 6 yield != 0); 7 critical section { ... } 8 // exit protocol 9 enter[0] = false; </pre>	<pre> 10 while (true) { 11 // entry protocol 12 enter[1] = true; 13 yield = 1; 14 await (!enter[0] 15 yield != 1); 16 critical section { ... } 17 // exit protocol 18 enter[1] = false; </pre>
--	---

Peterson's algorithm satisfies mutual exclusion

Instead of building the state/transition diagram, we can also **prove mutual exclusion** by **contradiction**:

- Assume t_0 and t_1 both are in their critical section
- We have $\text{enter}[0] == \text{true}$ and $\text{enter}[1] == \text{true}$ (t_0 and t_1 set them before last entering their critical sections)
- Either $\text{yield} == 0$ or $\text{yield} == 1$
 Without loss of generality, **assume** $\text{yield} == 0$
- Before last entering its critical section, t_0 must have set yield to 0; after that it cannot have changed yield again
- To enter its critical section, t_0 must have read $\text{yield} == 1$ (since $\text{enter}[1] == \text{true}$), so t_1 must have set yield to 1 **after** t_0 last changed yield to 0
- Since **neither** thread can have changed yield to 0 after that, we must have $\text{yield} == 1$

```

boolean[] enter = {false, false};  int yield = 0 || 1;

thread t0                               thread t1
1  while (true) {                       10 while (true) {
2  // entry protocol                    11 // entry protocol
3  enter[0] = true;                      12 enter[1] = true;
4  yield = 0;                             13 yield = 1;
5  await (!enter[1] ||                    14 await (!enter[0] ||
      yield != 0);                          yield != 1);
6  critical section { ... }              15 critical section { ... }
7  // exit protocol                      16 // exit protocol
8  enter[0] = false;                     17 enter[1] = false;
9  }                                      18 }

```

Contradiction!

Peterson's algorithm is starvation free

Suppose t_0 is waiting to enter its critical section. At the same time, t_1 must be doing one of four things:

1. t_1 is in its critical section: then, it will eventually leave it;
2. t_1 is in its non-critical section: then, `enter[1] == false`, so t_0 can enter its critical section;
3. t_1 is waiting to enter its critical section: then, `yield` is either 0 or 1, so one thread can enter the critical section;
4. t_1 keeps on entering and exiting its critical section: this is impossible because after t_1 sets `yield` to 1 it cannot cycle until t_0 has a chance to enter its critical section (and reset `yield`).

In all possible cases, t_0 eventually gets a chance to enter the critical section, so there is no starvation

Since starvation freedom implies deadlock freedom:

Peterson's algorithm is a correct mutual exclusion protocol

Peterson's algorithm is starvation free

```
boolean[] enter = {false, false};  int yield = 0 || 1;
```

thread t_0

thread t_1

```

1  while (true) {
2    // entry protocol
3    enter[0] = true;
4    yield = 0;
5    await (!enter[1] ||
           yield != 0);
6    critical section { ... }
7    // exit protocol
8    enter[0] = false;
9  }
```

```

10 while (true) {
11   // entry protocol
12   enter[1] = true;
13   yield = 1;
14   await (!enter[0] ||
           yield != 1);
15   critical section { ... }
16   // exit protocol
17   enter[1] = false;
18 }
```

... then thread t_0
 will **NOT starve**
 (it can go in into the
 critical section since
 enter[1]=false)

If yield=0 (or 1)
 and thread t_1 stops
 executing here
 (before the entry
 protocol)...

Peterson's algorithm for n threads

Peterson's algorithm easily generalizes to n threads

```
int[] enter = new int[n]; // n elements, initially all 0s
int[] yield = new int[n]; // use n - 1 elements 1..n-1
```

thread x

```

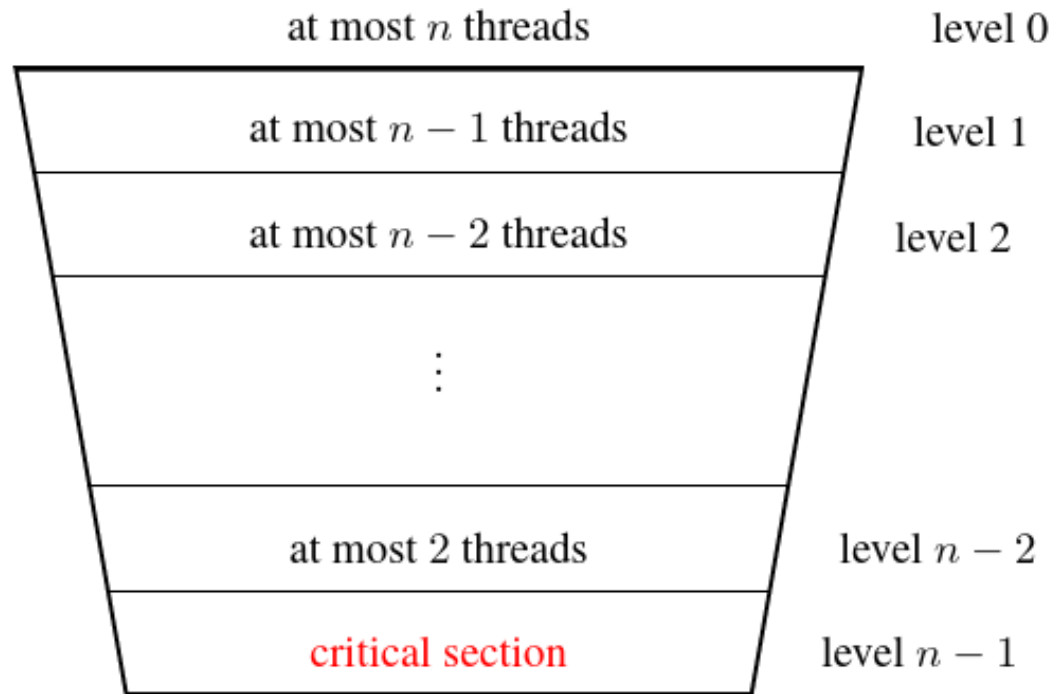
1 while (true) {
2   // entry protocol
3   for (int i = 1; i < n; i++) {
4     enter[x] = i; // want to enter level i
5     yield[i] = x; // but yield first
6     await (∀ t != x: enter[t] < i
            || yield[i] != x);
7   }
8   critical section { ... }
9   // exit protocol
10  enter[x] = 0; // go back to level 0

```

wait until all other threads are in lower levels

or another thread is yielding

Peterson's algorithm for n threads



```
int[] enter = new int[n]; // n elements, initially all 0s
int[] yield = new int[n]; // use n - 1 elements 1..n-1
```

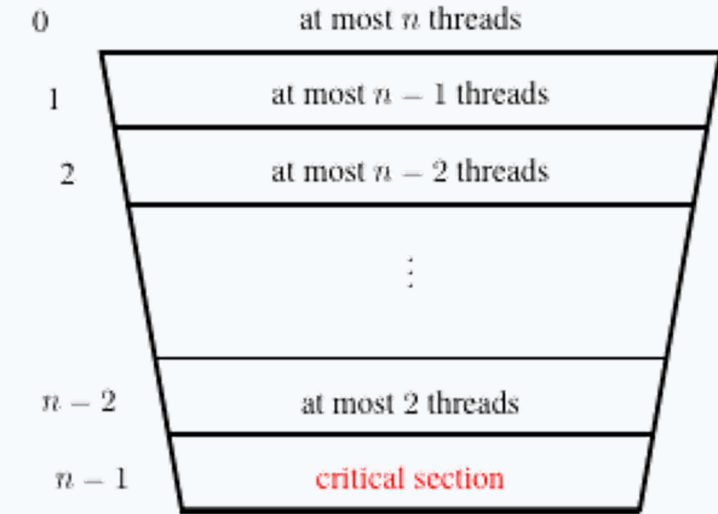
thread x

```
1 while (true) {
2   // entry protocol
3   for (int i = 1; i < n; i++) {
4     enter[x] = i; // want to enter level i
5     yield[i] = x; // but yield first
6     await (∀ t != x: enter[t] < i
7           || yield[i] != x);
8   }
9   critical section { ... }
10  // exit protocol
    enter[x] = 0; // go back to level 0
```

Peterson's algorithm for n threads

Every thread goes through $n - 1$ levels to enter the critical section:

- when a thread is at level 0 it is outside the entry region;
- when a thread is at level $n - 1$ it is in the critical section;
- Thread x is in level i when it has finished the loop at line 6 with `enter[x]=i`;
- `yield[l]` indicates the *last* thread that wants to enter level l last;
- to enter the next level, **wait until** there are no processes in higher levels, or another process (which entered the current level last) is yielding;
- **mutual exclusion**: at most $n - \ell$ processes are in level ℓ , thus at most $n - (n - 1) = 1$ processes in critical section.



```
int[] enter = new int[n]; // n elements, initially all 0s
int[] yield = new int[n]; // use n - 1 elements 1..n-1
```

thread x

```

1 while (true) {
2   // entry protocol
3   for (int i = 1; i < n; i++) {
4     enter[x] = i; // want to enter level i
5     yield[i] = x; // but yield first
6     await (∀ t != x: enter[t] < i
7           || yield[i] != x);
8   }
9   critical section { ... }
10  // exit protocol
    enter[x] = 0; // go back to level 0

```

Quiz

Models and Peterson's Algorithm

Mutual exclusion with bounded waiting

Bounded waiting (also called bounded bypass)

Peterson's algorithm guarantees freedom from starvation, but threads may get access to their critical section before other "older" threads

To describe this, we introduce more precise **properties of fairness**:

Finite waiting (starvation freedom): when a thread t is waiting to enter its critical section, it will **eventually** enter it

Bounded waiting: when a thread t is waiting to enter its critical section, the maximum number of times other arriving threads are allowed to enter their critical section before t is **bounded** by a function of the number of contending threads

r -bounded waiting: when a thread t is waiting to enter its critical section, the maximum number of times other arriving threads are allowed to enter their critical section before t is less than **$r + 1$**

First-come-first-served: **0-bounded** waiting

The Bakery algorithm

Lamport's Bakery algorithm achieves mutual exclusion, deadlock freedom, and **first-come-first-served access**

It is based on the idea of waiting threads getting **a ticket number**:

- Because of lack of atomicity, two threads may end up with the same ticket number
- In that case, their thread identifier number is used to force an order
- The tricky part is evaluating multiple variables (the ticket numbers of all other waiting processes) consistently
- Idea: a thread raises a flag when computing the number; other threads then wait to compute the numbers

Main **drawback** (compared to Peterson's algorithm): the original version of the Bakery algorithm may use arbitrarily large integers (the ticket numbers) in shared variables

Implementing mutual exclusion algorithms in Java

Now that you know how to do it...

... don't do it!

Learning how to achieve mutual exclusion using only atomic reads and writes **has educational value**, but you should not use it in realistic programs

- Use the locks and semaphores available in Java's **standard library**
- We will still give an overview of the things to know if you were to implement Peterson's algorithm, and similar ones, **from the ground up**

Peterson's lock in Java: 2 threads

```
class PetersonLock implements Lock {
    private volatile boolean enter0 = false, enter1 = false;
    private volatile int yield;
```

```
public void lock()
{
    int me = getThreadId();
    if (me == 0) enter0 = true;
    else enter1 = true;
    yield = me;
    while ((me == 0) ? (enter1 && yield == 0)
            : (enter0 && yield == 1)) {} }
}
```

```
public void unlock()
{
    int me = getThreadId();
    if (me == 0) enter0 = false;
    else enter1 = false; }
}
```

```
private volatile long id0 = 0;
```

volatile is required
for correctness

The loop will exit:
if me=0 and (enter1 is false or yield is 1)
or
if me=1 and (enter0 is false or yield is 0)

Instruction execution order

When we designed and analyzed concurrent algorithms, we implicitly assumed that threads execute instructions in textual program order

This is **not guaranteed** by the Java language – or, for that matter, by most programming languages – when threads access shared fields

(Read “The silently shifting semicolon” <http://drops.dagstuhl.de/opus/volltexte/2015/5025/> for a nice description of the problems)

- **Compilers** may reorder instructions based on static analysis, which does not know about threads.
- **Processors** may delay the effect of writes when the cache is committed to memory

This adds to the complications of writing low-level concurrent software correctly



Instruction execution order

The compiler might
Decide to move this instruction

```
class PetersonLock implements Lock {
    private volatile boolean enter0 = false, enter1 = false;
    private volatile int yield;

    public void lock()
    {
        int me = getThreadId();
        if (me == 0) enter0 = true;
        else enter1 = true;
        yield = me;
        while ((me == 0) ? (enter1 && yield == 0)
                : (enter0 && yield == 1)) {}
    }

    public void unlock()
    {
        int me = getThreadId();
        if (me == 0) enter0 = false;
        else enter1 = false;
    }

    private volatile long id0 = 0;
}
```

- **Compilers** may reorder instructions based on static analysis, which does not know about threads.
- **Processors** may delay the effect of writes when the cache is committed to memory

This adds to the complications of writing low-level concurrent software correctly



Volatile fields

Accessing a field (attribute) declared as **volatile** forces synchronization, and thus prevents any optimization from reordering instructions in a way that alters the “**happens before**” relationship defined by a program’s textual order

- By using **volatile** we ensure the value of the variable is not cached so it’s always read from the main memory

When accessing a shared variable that is accessed concurrently:

- declare the variable as **volatile**
- or guard access to the variable with **locks** (or other synchronization primitives)

Arrays and **volatile**

Java does **not support** arrays *whose elements* are **volatile**

That's why we used two scalar **boolean** var when implementating Peterson's lock

Workarounds:

- use an object of class `AtomicIntegerArray` in package `java.util.concurrent.atomic` which guarantees atomicity of accesses to its elements (the field itself need not be declared volatile)
- make sure that there is a read to a **volatile** field before every read to elements of the shared array, and that there is a write to a **volatile** field after every write to elements of the shared array; this forces synchronization indirectly (may be tricky to do correctly!)
- **explicitly** guard accesses to shared arrays with a **lock**: this is the high-level solution which we will preferably use

Peterson's lock in Java: 2 threads, with atomic arrays

```
class PetersonAtomicLock implements Lock {
    private AtomicIntegerArray enter = new AtomicIntegerArray(2);
    private volatile int yield;

    public void lock() {
        int me = getThreadId();
        int other = 1 - me;
        enter.set(me, 1);
        yield = me;
        while (enter.get(other) == 1 && yield == me) {}
    }

    public void unlock() {
        int me = getThreadId();
        enter.set(me, 0);
    }
}
```

Peterson's lock in Java: 2 threads

"Classic":

```
class PetersonLock implements Lock {
    private volatile boolean enter0 = false,
                               enter1 = false;
    private volatile int yield;

    public void lock()
    {   int me = getThreadId();
        if (me == 0) enter0 = true;
        else enter1 = true;
        yield = me;
        while ((me == 0) ? (enter1 && yield == 0)
                : (enter0 && yield == 1)) {}
    }

    public void unlock()
    {   int me = getThreadId();
        if (me == 0) enter0 = false;
        else enter1 = false;
    }
}
```

With atomic arrays:

```
class PetersonAtomicLock implements Lock {
    private AtomicIntegerArray
        enter = new AtomicIntegerArray(2);
    private volatile int yield;

    public void lock()
    {   int me = getThreadId();
        int other = 1 - me;
        enter.set(me, 1);
        yield = me;
        while (enter.get(other) == 1
                && yield == me) {}
    }

    public void unlock()
    {   int me = getThreadId();
        enter.set(me, 0);
    }
}
```

Mutual exclusion needs n memory locations

Peterson's algorithm for n threads uses $\Theta(n)$ shared memory locations (two n -element arrays)

- One can prove that this is the **minimum amount** of shared memory needed to have mutual exclusion *if only atomic reads and writes* are available
- This is one reason why synchronization using only atomic reads and writes is impractical
- We need more powerful primitive operations:
 - atomic **test-and-set** operations
 - support for **suspending** and **resuming** threads explicitly

Test-and-set

The **test-and-set** operation **boolean testAndSet()** works on a Boolean variable **b** as follows: **b.testAndSet()** **atomically** returns the current value of **b** **and** sets **b** to **true**

Java class `AtomicBoolean` implements test-and-set:

```
package java.util.concurrent.atomic;
public class AtomicBoolean {

    AtomicBoolean(boolean initialValue); // initialize to `initialValue`

    boolean get();                       // read current value
    void set(boolean newValue);          // write `newValue`

    // return current value and write `newValue`
    boolean getAndSet(boolean newValue);
        // testAndSet() is equivalent to getAndSet(true)
}
```

A lock using test-and-set

An implementation of n -process **mutual exclusion** using a **single Boolean variable** with **test-and-set** and **busy waiting**:

```
public class TASLock implements Lock {
    AtomicBoolean held =
        new AtomicBoolean(false);

    public void lock() {
        while (held.getAndSet(true)) {
            // await (!testAndSet());
        }
    }

    public void unlock() {
        held.set(false); // held = false;
    }
}
```

- Variable `held` is true iff the lock is held by some thread
- When **locking** (executing `lock`):
 - as long as `held` is true (someone else holds the lock), keep resetting it to true and wait
 - as soon as `held` is false: leave the loop and `held` is set it to true
 - You hold the lock now
- When **unlocking** (executing `unlock`): set `held` to false

A lock using **test-and-test-and-set**

A lock implementation using a **single Boolean variable with test-and-test-and-set** and busy waiting:

```
public class TTASLock extends TASLock {
    @Override
    public void lock() {
        while (true) {
            while(held.get()) {}
            if (!held.getAndSet(true))
                return;
        }
    }
}
```

When locking (executing lock):

- spin until held is false
- then check if held is still false, and if it is set it to true (you hold the lock now), then return
- otherwise it means another thread “stole” the lock from you; then repeat the locking procedure from the beginning

This variant tends to perform better, since the busy waiting is **local** to the cached copy as long as no other thread changes the lock’s state

Implementing semaphores

Semaphores: recap

A (general/counting) **semaphore** is a data structure with interface:

```
interface Semaphore {  
    int count();    // current value of counter  
    void up();      // increment counter  
    void down();   // decrement counter  
}
```

Several threads share the same object `sem` of type `Semaphore`:

- initially `count` is set to a nonnegative value `C` (the **capacity**)
- a call to `sem.up()` **atomically increments** `count` by one
- a call to `sem.down()`: **waits** until `count` is positive, and then **atomically decrements** `count` by one

Semaphores with locks

An implementation of semaphores using locks and busy waiting:

```

class SemaphoreBusy implements Semaphore {
    private int count;

    public synchronized void up() {
        count = count + 1;
    }

    public void down() {
        while (true) {
            synchronized (this) {
                if (count > 0) { // await (count > 0);
                    count = count - 1; return;
                }
            }
        }
    }

    public synchronized int count() {
        return count;
    }
}

```

Executed
exclusively

Why not lock the whole method?

To avoid blocking other threads
to enter the method
(avoid that the first thread calling
`down` is the first to get the lock!

Does this have to be **synchronized**?

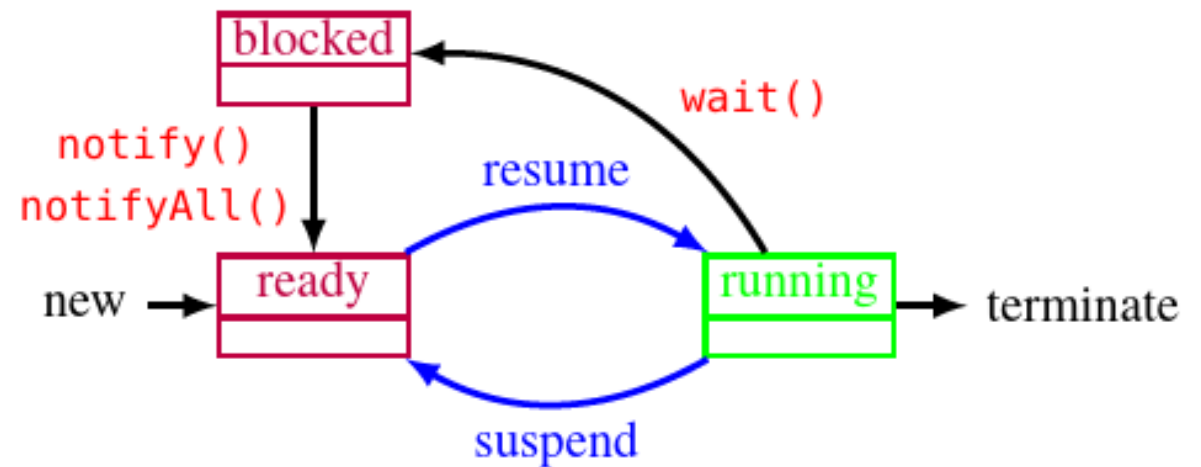
Yes, if `count` is not **volatile**

Suspending and resuming threads

To avoid **busy waiting**, we have to rely on more powerful synchronization primitives than only reading and writing variables

A standard solution uses Java's **explicit scheduling of threads**

- calling `wait()` suspends the currently running thread
- calling `notify()` moves one (nondeterministically chosen) blocked thread to the **ready** state
- calling `notifyAll()` moves all blocked threads to the **ready** state



Waiting and notifying only affects the threads that are locked on the same shared object (using **synchronized** blocks or methods)

Weak semaphores with suspend/resume

An implementation of **weak** semaphores using `wait()` and `notify()`

```

class SemaphoreWeak implements Semaphore {
    private int count;

    public synchronized void up() {
        count = count + 1;
        notify(); // wake up a waiting thread
    }

    public synchronized void down() throws InterruptedException {
        while (count == 0) wait(); // suspend running thread
        count = count - 1; // now count > 0
    }

    public synchronized int count() {
        return count;
    }
}

```

Since `notify` is nondeterministic
this is a **weak** semaphore

`wait` releases the object lock
(so other threads can enter the method even
if it is marked as “synchronized”)

In general, `wait` must be called in a loop in case of spurious wakeups;
this is not busy waiting (and it's required by Java's implementation)

Strong semaphores with suspend/resume

An implementation of **strong** semaphores using `wait()` and `notifyAll()`

```

class SemaphoreStrong implements Semaphore {
    public synchronized void up() {
        if (blocked.isEmpty()) count = count + 1;
        else notifyAll(); // wake up all waiting threads
    }

    public synchronized void down() throws InterruptedException {
        Thread me = Thread.currentThread();
        blocked.add(me); // enqueue me
        while (count == 0 || blocked.element() != me)
            wait(); // I'm enqueued when suspending
        // now count > 0 and it's my turn: dequeue me and decrement
        blocked.remove(); count = count - 1;
    }

    private final Queue<Thread> blocked = new LinkedList<>();
    private int count;
  
```

Check there are no
suspended threads

Wrong!

Keeps suspending the thread
if the count is 0 or I am not
the first in the queue


```
class SemaphoreStrong implements Semaphore {

    public synchronized void up()
    {
        if (blocked.isEmpty()) count = count + 1;
        else notifyAll();    } // wake up all waiting threads

    public synchronized void down() throws InterruptedException
    {
        Thread me = Thread.currentThread();
        blocked.add(me);    // enqueue me
        while (count == 0 || blocked.element() != me)
            wait();    // I'm enqueued when suspending
        // now count > 0 and it's my turn: dequeue me and decrement
        blocked.remove(me);    count = count - 1;    }
```

Deadlock

Remember that wait releases the object lock (that's why the yellow thread can go in)

(x)= Variables		Breakpoints	Expressions
count	0		
blocked			

```
class StrongSemUser implements Runnable {
    private SemaphoreStrong sem = new SemaphoreStrong(1);

    public void run()
    {
        while (true) {
            // Non critical
            sem.down();
            // Critical
            sem.up();
        }
    }
}
```

Strong semaphores with suspend/resume

An implementation of **strong** semaphores using `wait()` and `notifyAll()`

```
class SemaphoreStrong implements Semaphore { Removed if (blocked.isEmpty())
```

```
public synchronized void up() {  
    count = count + 1;  
    notifyAll();    // wake up all waiting threads  
}
```

Fix

```
public synchronized void down() throws InterruptedException {  
    Thread me = Thread.currentThread();  
    blocked.add(me);    // enqueue me  
    while (count == 0 || blocked.element() != me)  
        wait();    // I'm enqueued when suspending  
    // now count > 0 and it's my turn: dequeue me and decrement  
    blocked.remove();    count = count - 1;
```

```
}
```

```
private final Queue<Thread> blocked = new LinkedList<>();
```

```
private int count;
```

```
}
```



Debugging concurrent
programs is very
difficult!

General semaphores using binary semaphores

A general semaphore can be implemented using just **two binary semaphores**

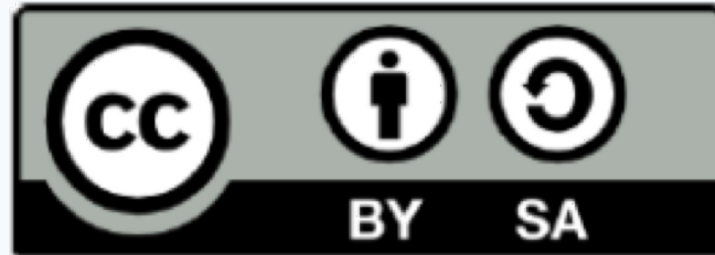
Barz's solution in pseudocode (with `capacity > 0`):

```
BinarySemaphore mutex = 1; // protects access to count
BinarySemaphore delay = 1; // blocks threads in down until count > 0
int count = capacity;      // value of general semaphore
void up()
{  mutex.down();           // get exclusive access to count
   count = count + 1;      // increment count
   if (count == 1) delay.up(); // release threads blocking on down
   mutex.up(); }          // release exclusive access to count
void down()
{  delay.down();           // block other threads starting down
   mutex.down();           // get exclusive access to count
   count = count - 1;      // decrement count
   if (count > 0) delay.up(); // release threads blocking on down
   mutex.up(); }          // release exclusive access to count
```

Quiz

Implementing Semaphores

© 2016–2019 Carlo A. Furia, Sandro Stucki



Except where otherwise noted, this work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

To view a copy of this license, visit

<http://creativecommons.org/licenses/by-sa/4.0/>.