Models of concurrency & synchronization algorithms

Lecture 3 of TDA384/DIT391

Principles of Concurrent Programming

Nir Piterman

Chalmers University of Technology | University of Gothenburg

SP1 2021/2022

UNIVERSITY OF GOTHENBURG

CHALMERS UNIVERSITY OF TECHNOLOGY

Models of concurrency & synchronization algorithms

Today's menu

- Modeling concurrency
- Mutual exclusion with only atomic reads and writes
  - Three failed attempts
  - Peterson's algorithm
  - Mutual exclusion with strong fairness
- Implementing mutual exclusion algorithms in Java
- Implementing semaphores

Modeling concurrency

State/transition diagrams

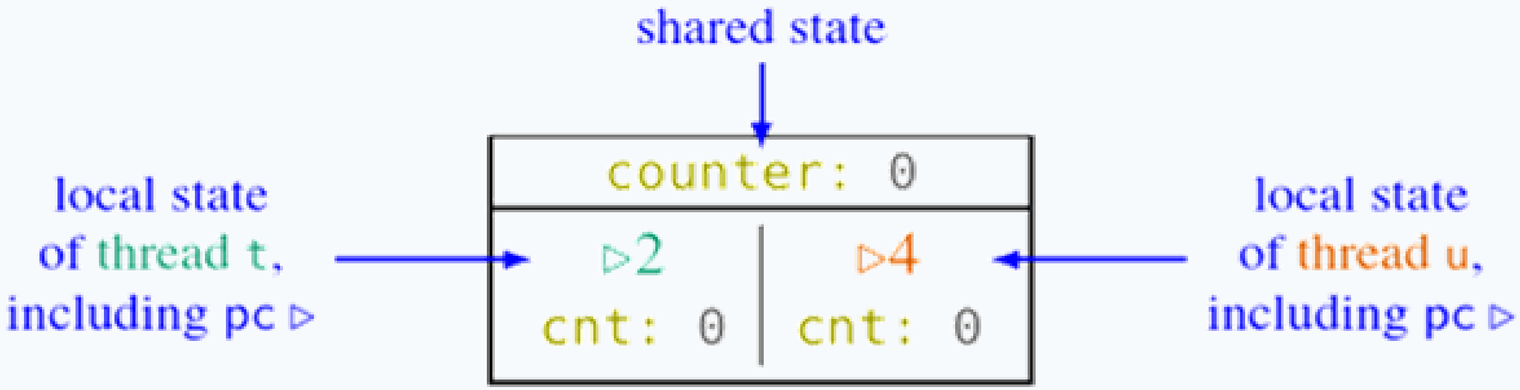
We capture the essential elements of concurrent programs using state/transition diagrams (also called: *(finite) state automata*, *(finite) state machines*, or *transition systems*).

- 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:

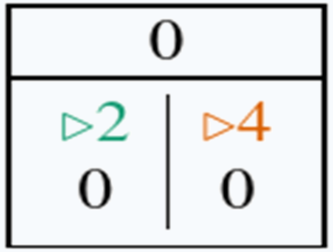



```
int counter = 0;
-----------------------------------------------------------
        thread t              |          thread u
   int cnt;                   |  int cnt;
                              |
1  cnt = counter;             |  cnt = counter;          3
2  counter = cnt + 1;         |  counter = cnt + 1;      4
```

States

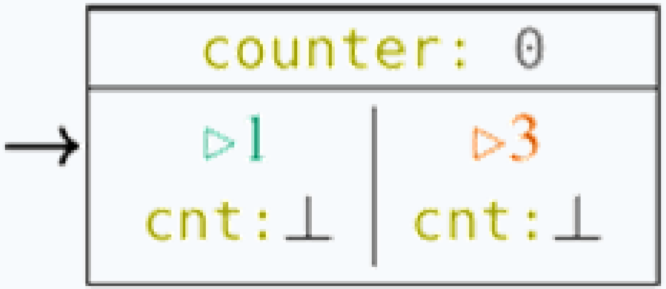
A state captures the shared and local states of a concurrent program:

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



Initial states

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

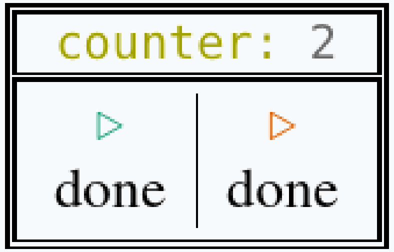



```
int counter = 0;
```

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:

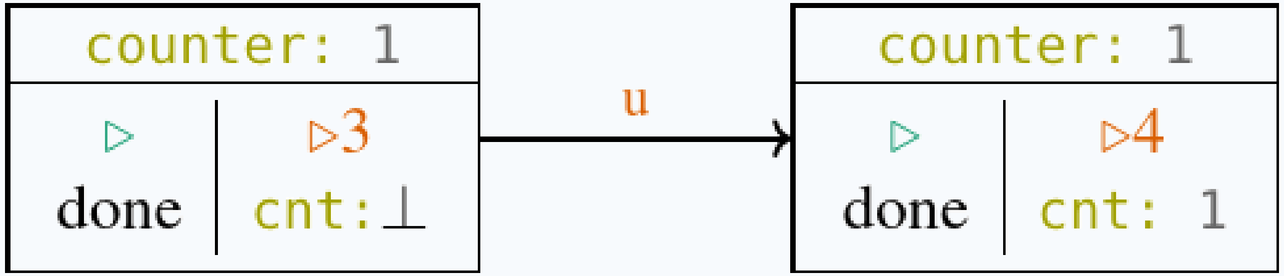



```
int counter = 0;
```

thread t	thread u
`int cnt;`	`int cnt;`
1 `cnt = counter;`	`cnt = counter;` 3
2 `counter = cnt + 1;`	`counter = cnt + 1;` 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):

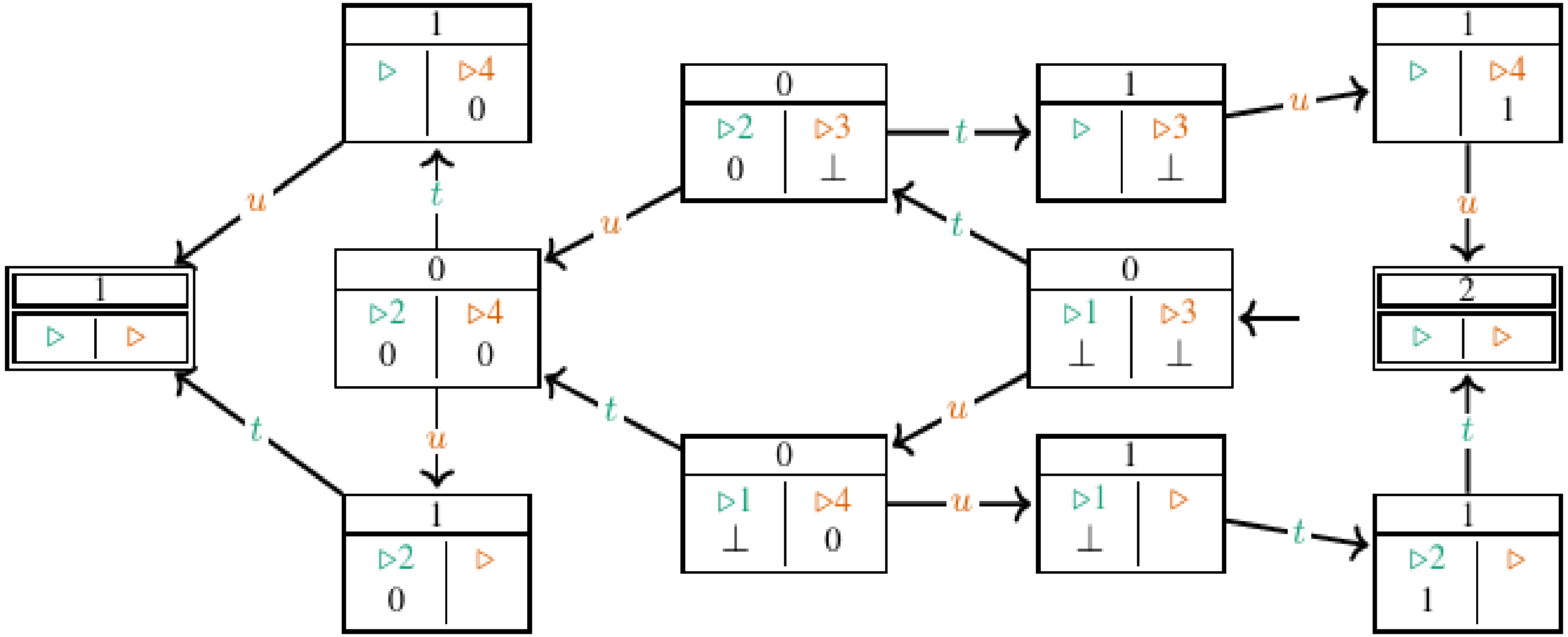



```
int counter = 0;
```

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

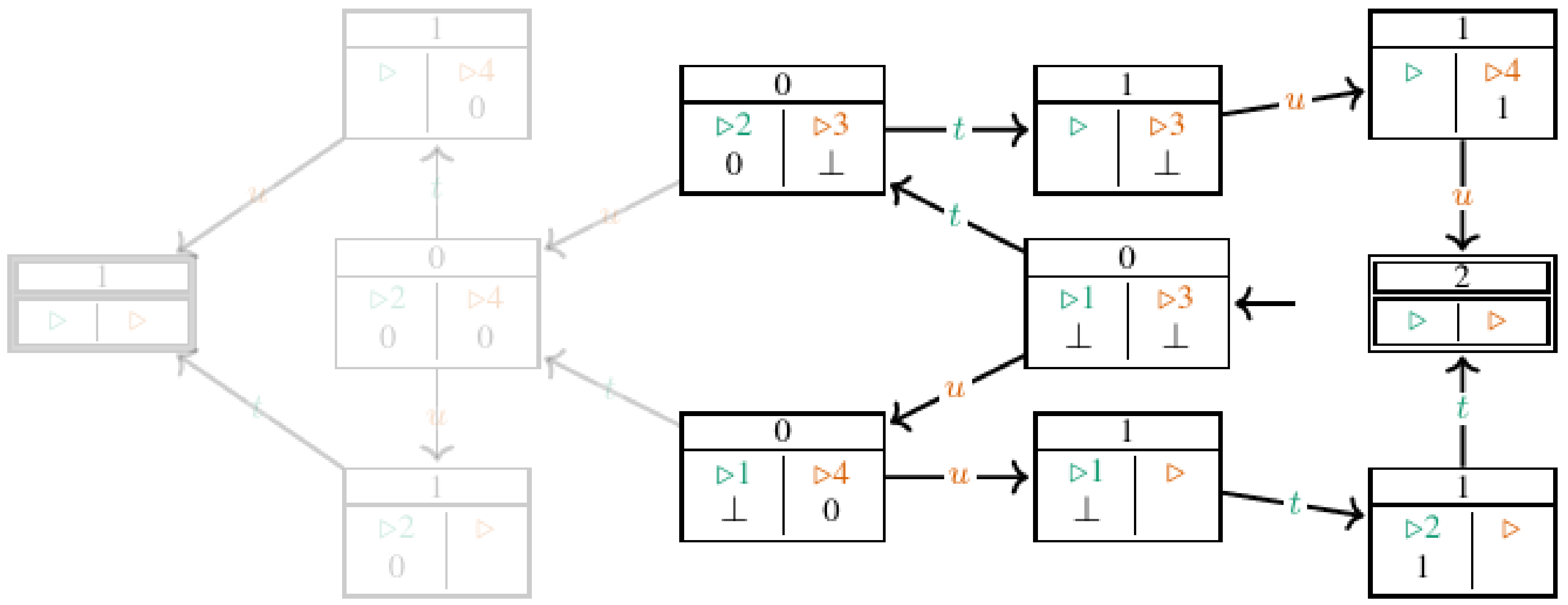
A complete state/transition diagram

The complete state/transition diagram for the concurrent counter example explicitly shows all possible interleavings:



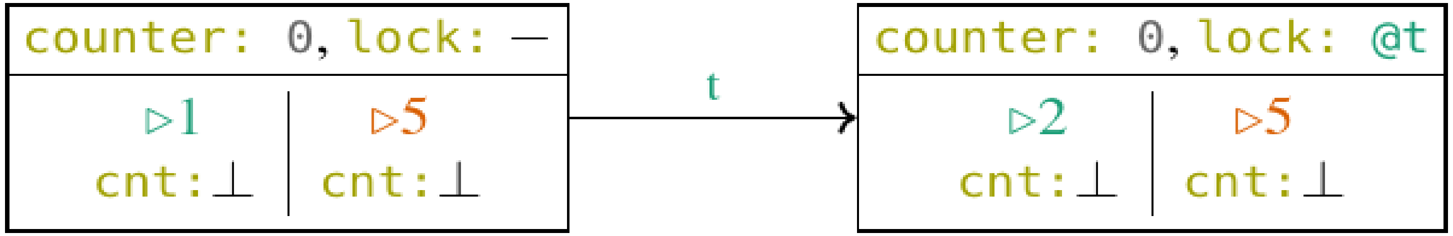
State/transition diagram with locks?

The state/transition diagram of the concurrent counter example using locks should contain no (states representing) race conditions:



Locking

Locking and unlocking are considered atomic operations.



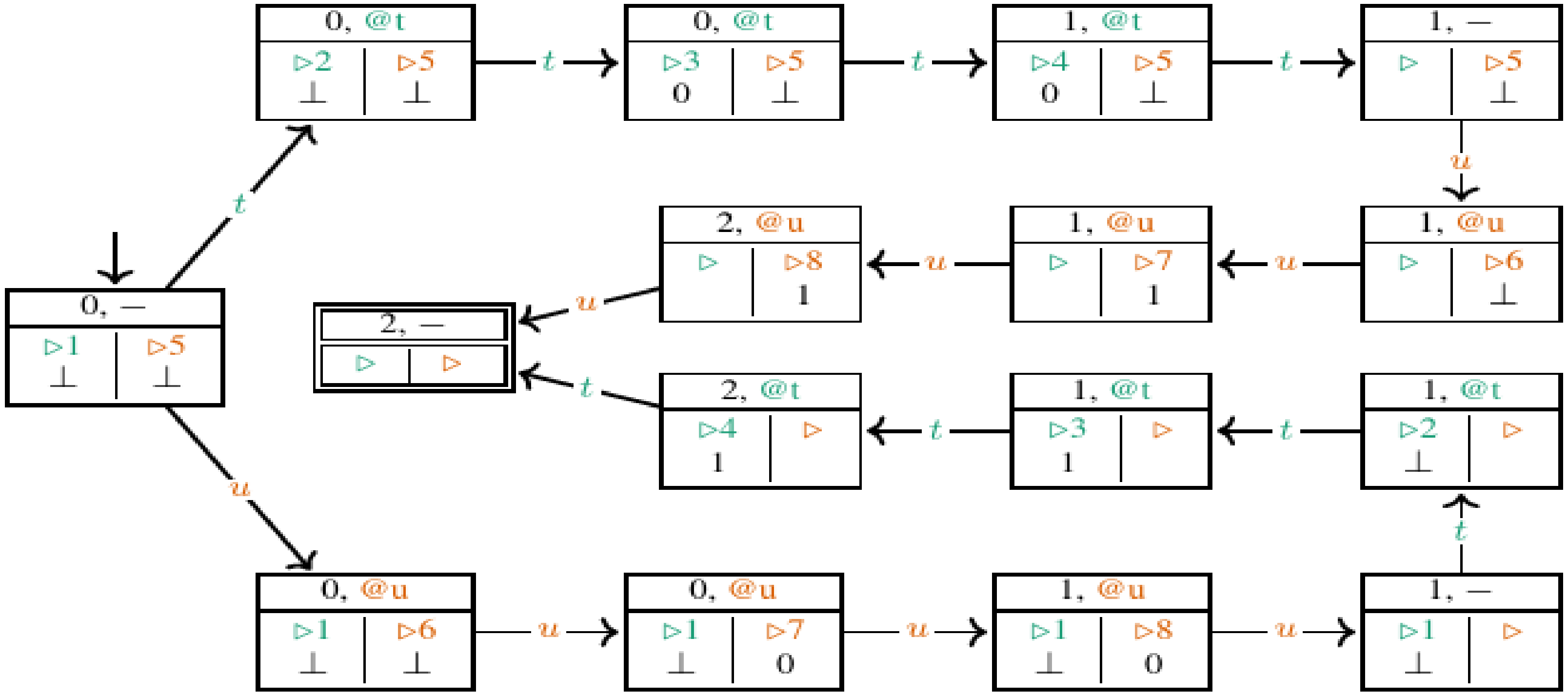

```
int counter = 0;     Lock lock = new Lock();
```

thread t	thread u
`int cnt;`	`int cnt;`
`lock.lock();`	`lock.lock();`
`  cnt = counter;`	`  cnt = counter;`
`  counter = cnt + 1;`	`  counter = cnt + 1;`
`lock.unlock();`	`lock.unlock();`

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

Counter with locks: state/transition diagram

The state/transition diagram of the concurrent counter example using locks contains no (states representing) race conditions:



Reasoning about program properties

The structural properties of a diagram capture semantic properties of the corresponding 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.

Transition tables

Transition tables are *equivalent representations* of the information of state/transition diagrams.

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

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()` results in exactly one thread *t* acquiring 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 achieve the behavior of 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

```
await(c)  ≜   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`**

Note that 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 – that is why 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

```
while (true) {
  // entry protocol
  await (!enter[1]);
  enter[0] = true;
  critical section { ... }
  // exit protocol
  enter[0] = false;
}
```

thread t_1

```
while (true) {
  // entry protocol
  await (!enter[0]);
  enter[1] = true;
  critical section { ... }
  // exit protocol
  enter[1] = false;
}
```

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.

#	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

```
while (true) {
  // entry protocol
  enter[0] = true;
  await (!enter[1]);
  critical section { ... }
  // exit protocol
  enter[0] = false;
}
```

thread t_1

```
while (true) {
  // entry protocol
  enter[1] = true;
  await (!enter[0]);
  critical section { ... }
  // exit protocol
  enter[1] = false;
}
```

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

#	t_0	t_1	SHARED
1	pc_0: `enter[0] = true`	pc_1: `enter[0] = true`	enter: false, false
1	pc_0: `await (!enter[1])`	pc_1: `enter[0] = true`	enter: true, false
2	pc_0: `await (!enter[1])`	pc_1: `await (!enter[0])`	enter: true, true

The problem seems to be that there are two variables `enter[0]` and `enter[1]` that are accessed independently, so 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

```
while (true) {
  // entry protocol
  await (yield != 0);
  critical section { ... }
  // exit protocol
  yield = 0;
}
```

thread t_1

```
while (true) {
  // entry protocol
  await (yield != 1);
  critical section { ... }
  // exit protocol
  yield = 1;
}
```

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)

In future classes, we discuss how model checking can more rigorously and systematically verify whether such correctness properties hold in a concurrent program.

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

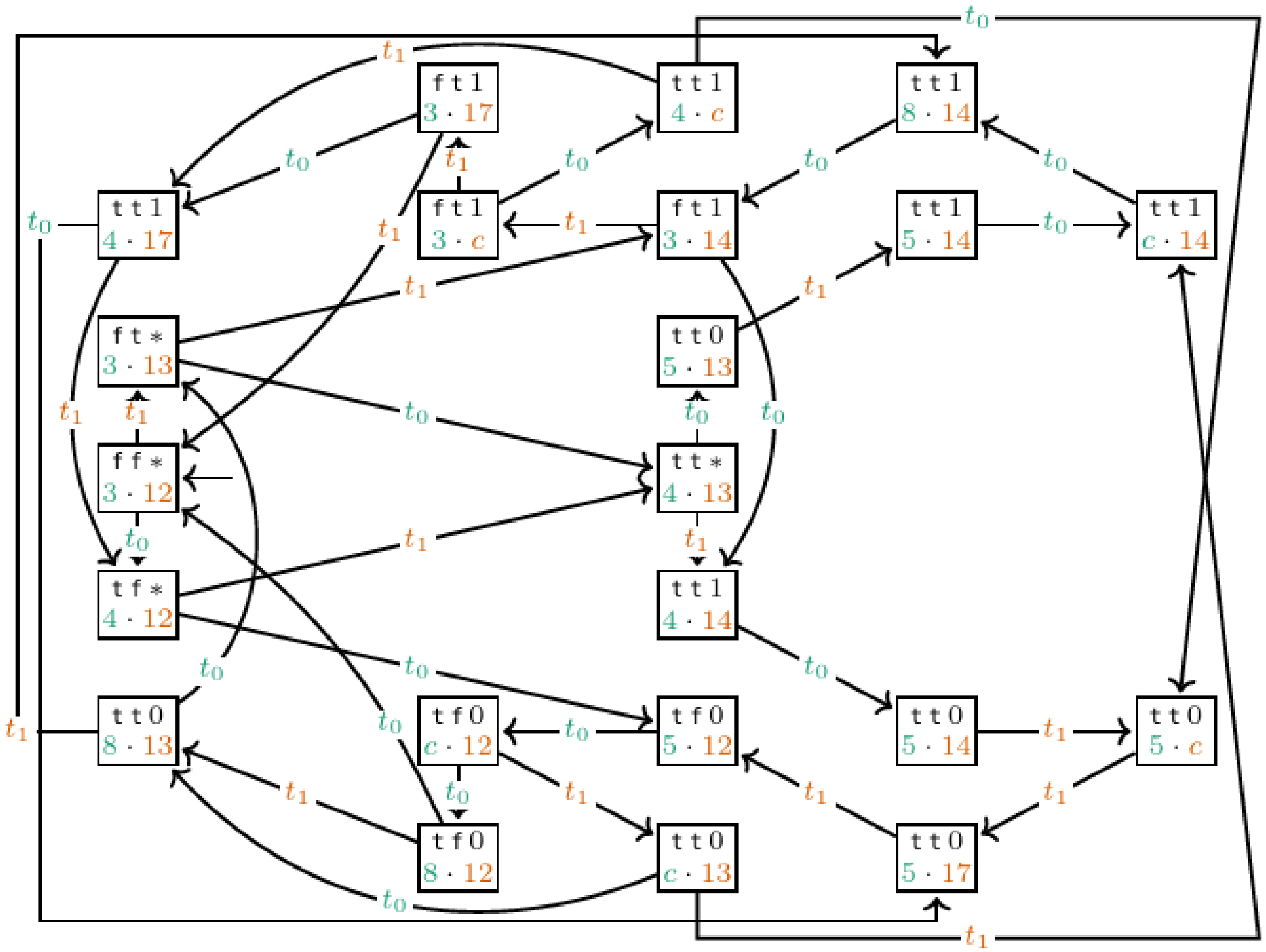
```
while (true) {
 // entry protocol
 enter[0] = true;
 yield = 0;
 await (!enter[1]        ||
          yield != 0);
 critical section { ... }
 // exit protocol
 enter[0] = false;
}
```

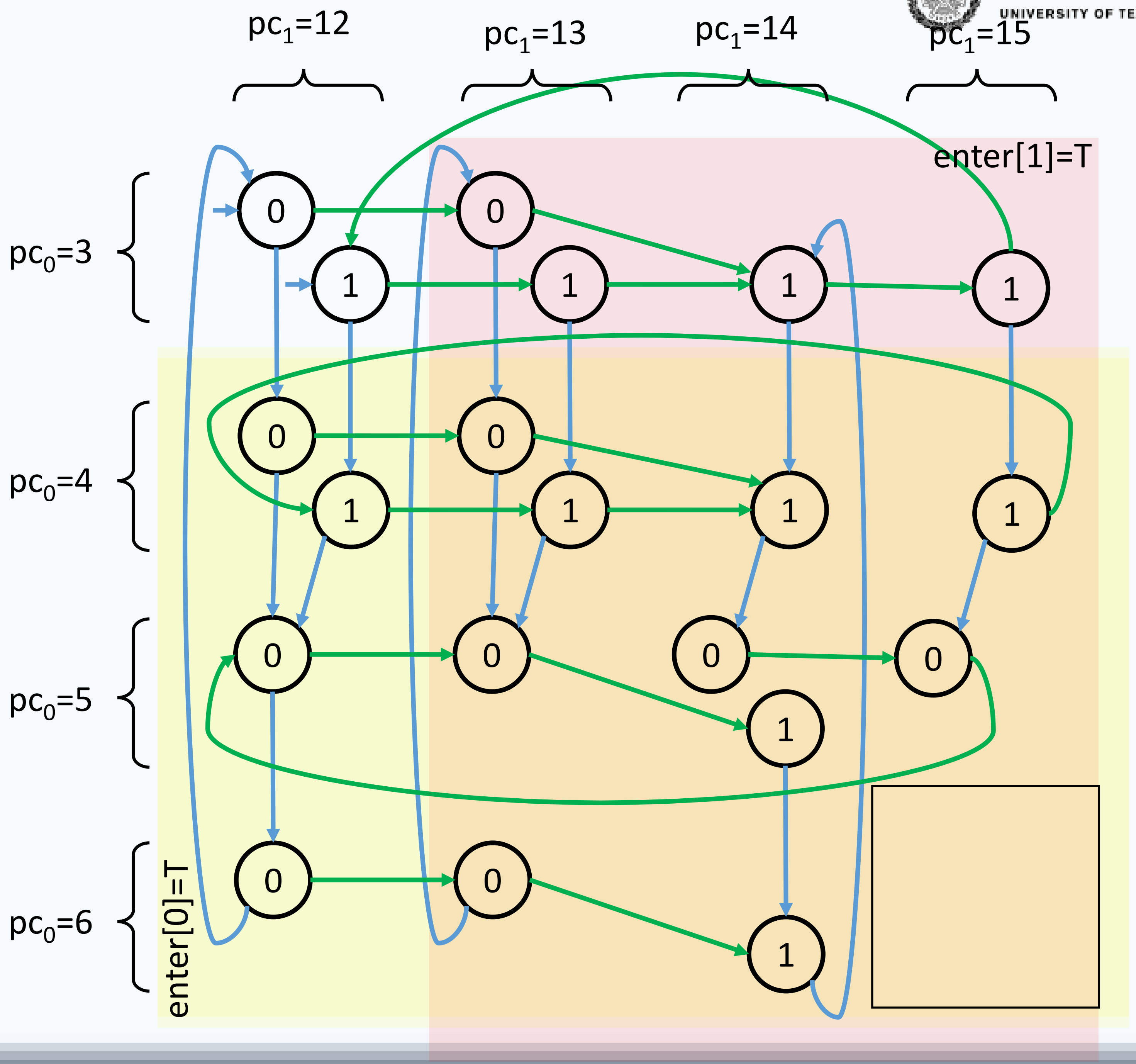
thread t_1

```
while (true) {
 // entry protocol
 enter[1] = true;
 yield = 1;
 await (!enter[0]        ||
          yield != 1);
 critical section { ... }
 // exit protocol
 enter[1] = false;
}
```

Works even if
two reads are non-atomic

State/transition diagram of Peterson's algorithm




pc$_1$=12
pc$_1$=13
pc$_1$=14
pc$_1$=15
pc$_0$=3
pc$_0$=4
pc$_0$=5
pc$_0$=6
enter[1]=T
enter[0]=T
0
1

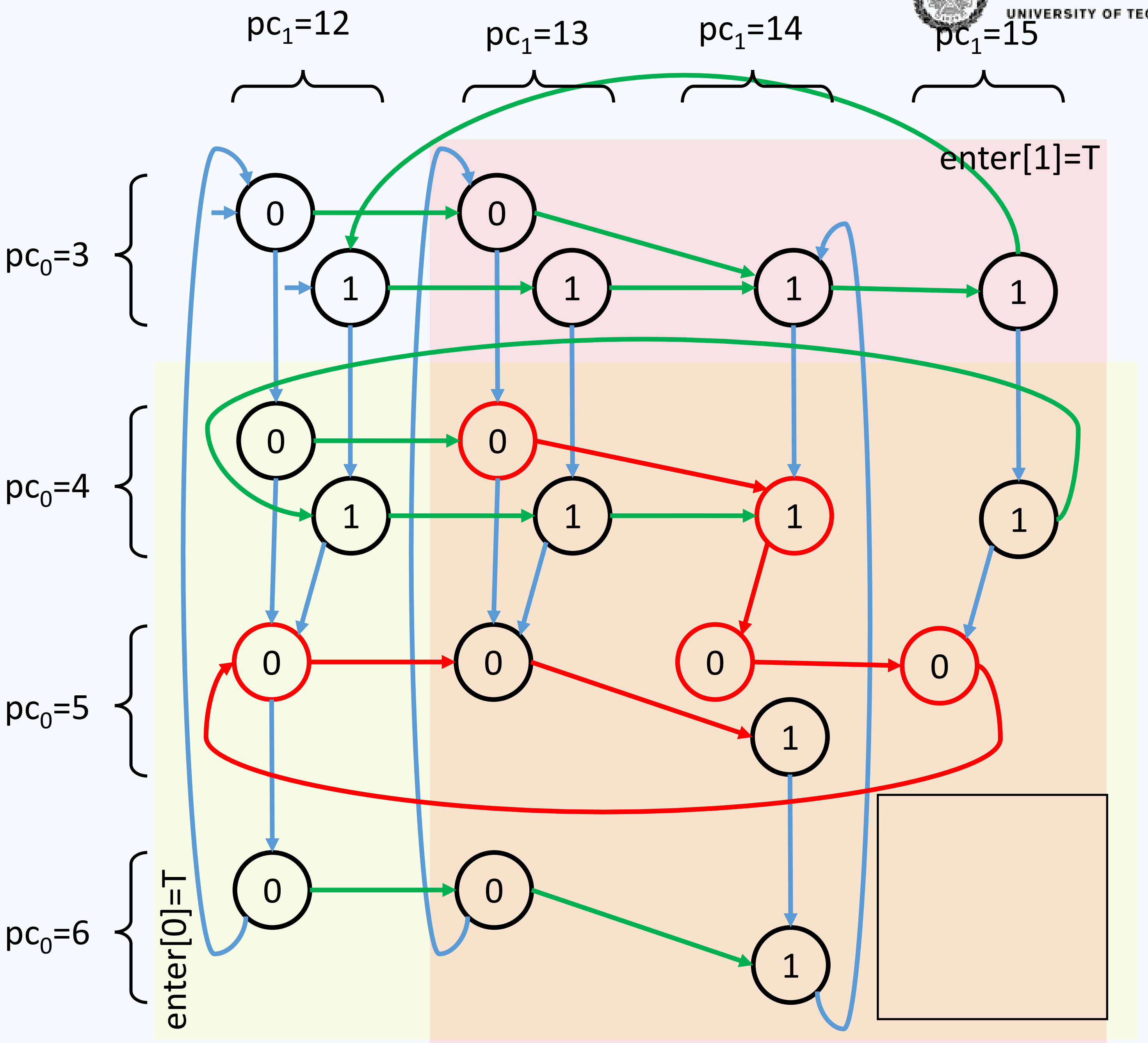
Checking the correctness of Peterson's algorithm

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

mutual exclusion: there are no states where both threads are C – that is, 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 it executes the exit protocol.


pc1=12
pc1=13
pc1=14
pc1=15
pc0=3
pc0=4
pc0=5
pc0=6
enter[1]=T
enter[0]=T
0
1
0
1
1
1
0
1
0
1
1
1
0
0
0
0
1
0
0
1

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 `enter[0] == true` and `enter[1] == true` (t_0 and t_1 set them before last entering their critical sections).
- Either `yield == 0` or `yield == 1`.
  Without loss of generality, assume `yield == 0`.
- Before last entering its critical section, t_0 must have set `yield` to `0`; after that it cannot have changed `yield` again.
- After that, to enter its critical section, t_0 must have read `yield == 1` (since `enter[1] == 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 `yield == 1`. 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 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
```

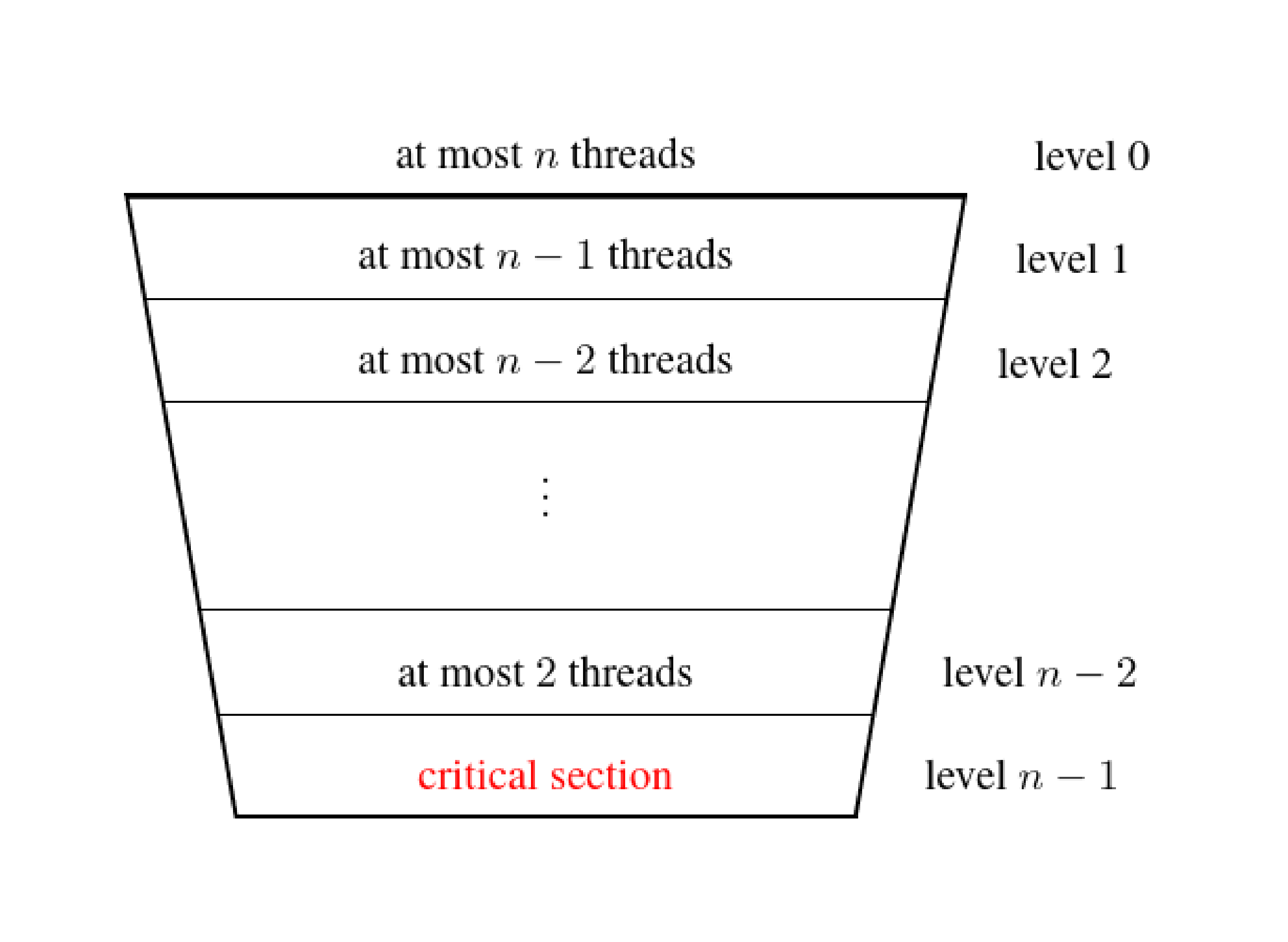
thread x

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

wait until all other threads are in lower levels

or another thread is yielding

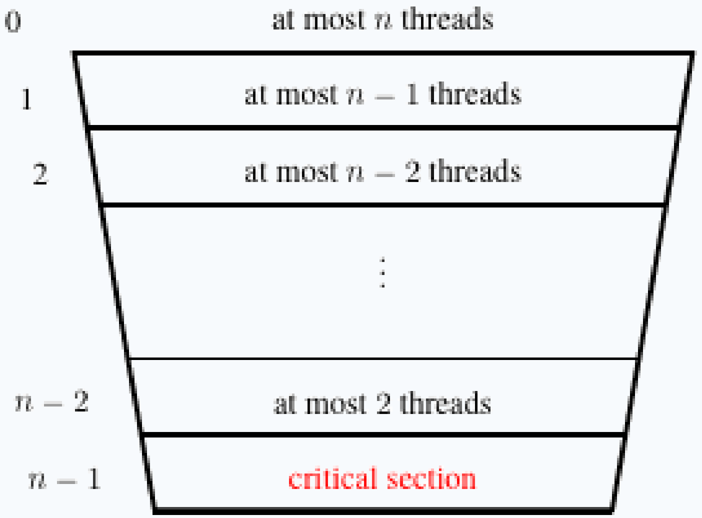
Peterson's algorithm for n threads



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 `t` is in level `i` when it has finished the loop at line 6 with `enter[t]=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.



Mutual exclusion with strong fairness

Bounded waiting (also called bounded bypass)

Peterson's algorithm guarantees freedom from starvation, but threads may get access to their critical section before other threads that have been waiting longer. 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 fairness. It is based on the idea of waiting threads getting a ticket number (like in a bakery, or everywhere in Sweden):

- 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

The main drawback, compared to Peterson's algorithm, is that 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

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 to 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.

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`. This is why we used two scalar `boolean` variables in the implementation of 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);
    }
```

Mutual exclusion needs $\mathbf{n}$ 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, 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` still is false, and if it is set it to true – you hold the lock now; 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;
    }
}
```

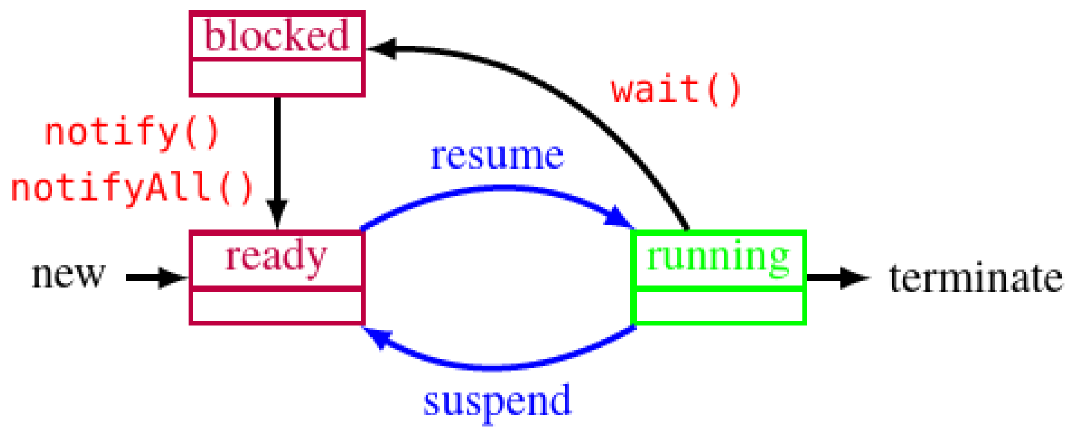
Executed exclusively

why not lock the whole method?

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

in general, wait must be called in a loop in case of spurious wakeups; this is not busy waiting (and is 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;
```

Wrong!

```
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;    }
```

Deadlock

(x)= Variables	Breakpoints	Expressions
count	0	
blocked		

```
class StrongSemUser implements Runnable {
    private SemaphoreStrong sem = new SemaphoreString(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()`.

Fix

```
class SemaphoreStrong implements Semaphore {

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

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
```