Races, locks and semaphores

Lecture 2 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

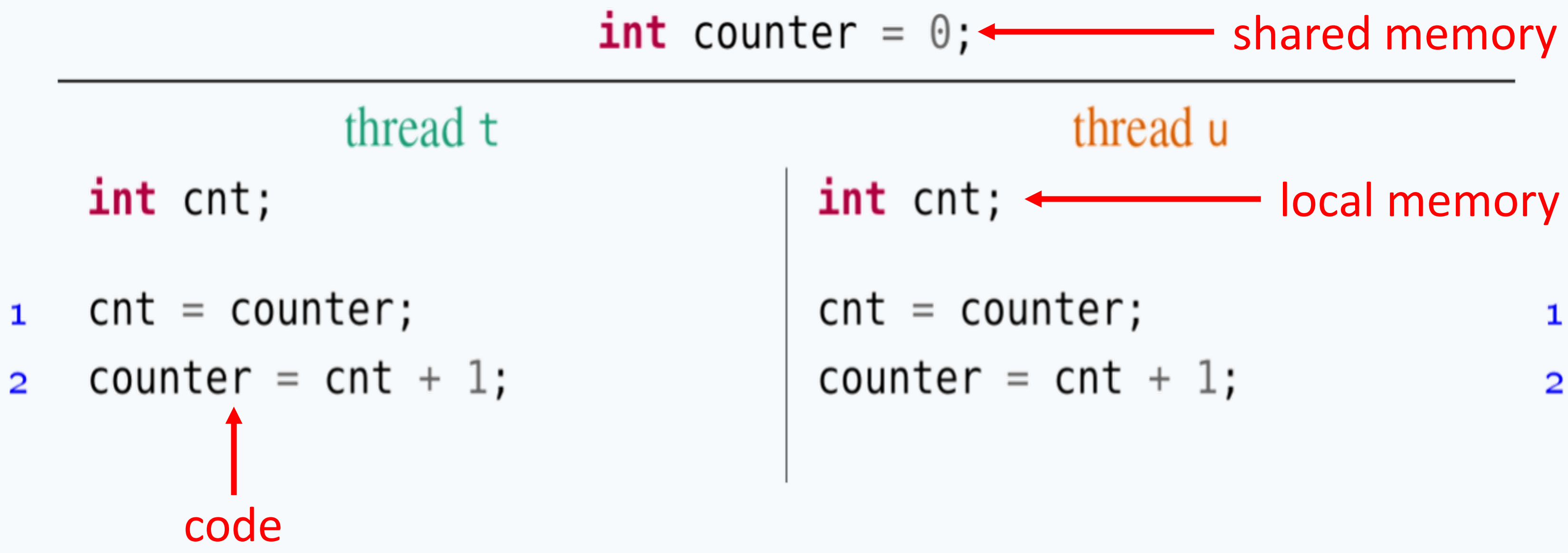
Today's menu

- Concurrent programs
- Races
- Synchronization problems
- Locks
- Semaphores
- Synchronization with semaphores

Concurrent programs

Abstraction of concurrent programs

When convenient, we will use an abstract notation for multi-threaded applications, which is similar to the pseudo-code used in Ben-Ari’s book but uses Java syntax.



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

shared memory

local memory

code

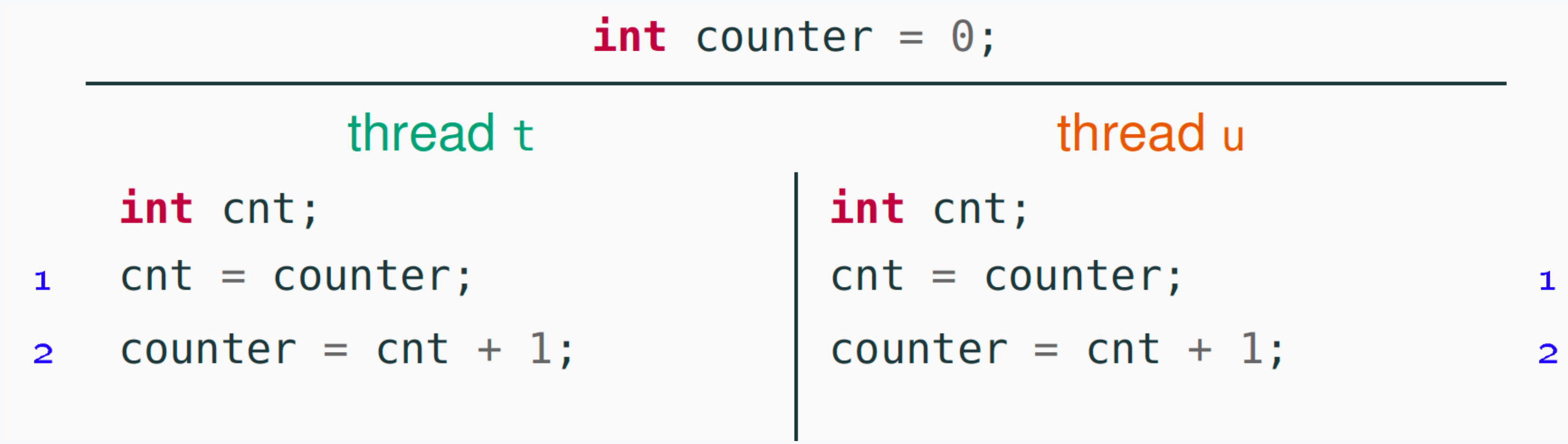
Each line of code includes exactly one instruction that can be executed atomically:

- atomic statement ≅ single read or write to global variable
- precise definition is tricky in Java, but we will learn to avoid pitfalls

Traces

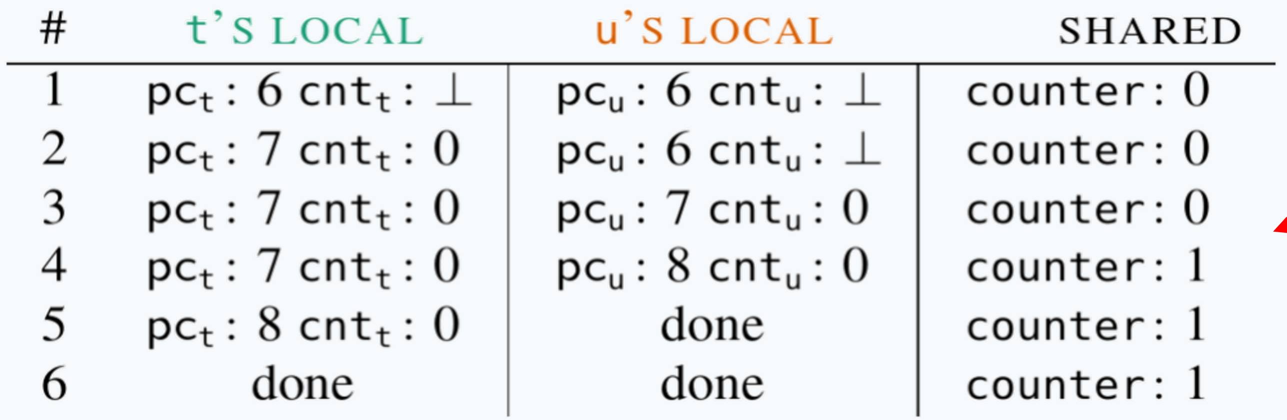
A sequence of states gives an execution trace of the concurrent program

(The program counter points to the atomic instruction that will be executed next)



```
int counter = 0;
```

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



#	t'S LOCAL	u'S LOCAL	SHARED
1	$pc_t : 6\ cnt_t : \bot$	$pc_u : 6\ cnt_u : \bot$	counter: 0
2	$pc_t : 7\ cnt_t : 0$	$pc_u : 6\ cnt_u : \bot$	counter: 0
3	$pc_t : 7\ cnt_t : 0$	$pc_u : 7\ cnt_u : 0$	counter: 0
4	$pc_t : 7\ cnt_t : 0$	$pc_u : 8\ cnt_u : 0$	counter: 1
5	$pc_t : 8\ cnt_t : 0$	done	counter: 1
6	done	done	counter: 1

One **trace**
(One possible
Interleaving)

Races

Race conditions

Concurrent programs are nondeterministic:

- Executing multiple times the same concurrent program with the same inputs may lead to different execution traces
- A result of the nondeterministic interleaving of each thread’s trace to determine the overall program trace
- In turn, the interleaving is a result of the scheduler’s decisions

A race condition is a situation where the correctness of a concurrent program depends on the specific execution

The concurrent counter example has a **race condition**:

- in some executions the final value of `counter` is **2** (correct)
- in some executions the final value of `counter` is **1** (wrong)

Race conditions can greatly complicate debugging!

Concurrency humor

A1: Knock Knock

A2: "Who’s there?"

A1: "Race condition"

A1: Knock…

A2: "Who’s there?"

A1: Knock…
"Race condition"

A1: Knock Knock

A1: "Race condition"

A2: "Who’s there?"

Data races

Race conditions are typically caused by a lack of synchronization between threads that access shared memory

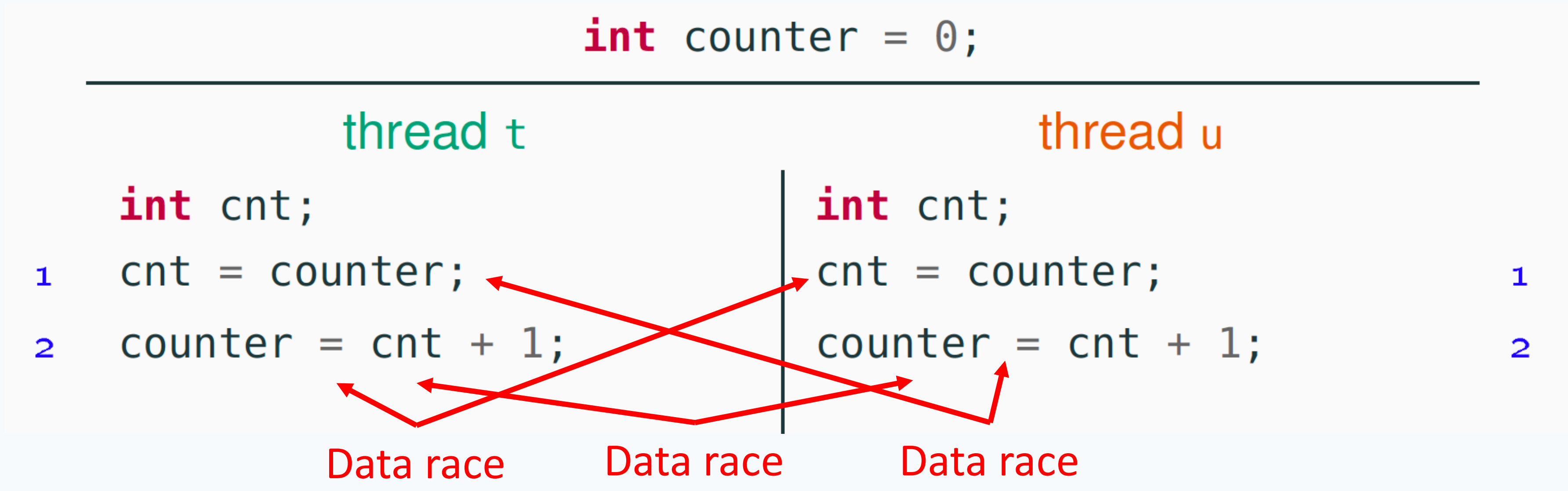
A data race occurs when two concurrent threads:

- Access a shared memory location
- At least one access is a write
- The threads use no explicit synchronization mechanism to protect the shared data

Data races

A data race occurs when two concurrent threads:

- Access a shared memory location
- At least one access is a write
- The threads use no explicit synchronization mechanism to protect the shared data



```
                         int counter = 0;
   ──────────────────────────────────────────────────────
              thread t                       thread u
     int cnt;                    │ int cnt;
1    cnt = counter;              │ cnt = counter;            1
2    counter = cnt + 1;          │ counter = cnt + 1;        2
```

Data race

Data race

Data race

Data races vs. Race conditions

A data race occurs when two concurrent threads:

- Access a shared memory location
- At least one access is a write
- The threads use no explicit synchronization mechanism to protect the shared data

Not every race condition is a data race

- Race conditions can occur even when there is no shared memory access
- Example: filesystems (open/close in wrong order) or network access

Not every data race is a race condition

- The data race may not affect the result
- Example: if two threads write the same value to shared memory

Synchronization problems

Push out the races, bring in the speed

Concurrent programming introduces:

- the **potential** for parallel execution (faster, better resource usage)
- the **risk** of race conditions (incorrect, unpredictable computations)

The main challenge of concurrent programming is thus **introducing** parallelism **without introducing** race conditions

This requires to **restrict** the amount of nondeterminism by synchronizing processes/threads that access shared resources

Synchronization

We will present several synchronization problems that often appear in concurrent programming, together with solutions

- Correctness (that is, avoiding race conditions) is **more important** than performance
  - An incorrect result that is computed faster is no good!

- However, we want to retain as much concurrency as possible
  - Otherwise we might as well stick with sequential programming

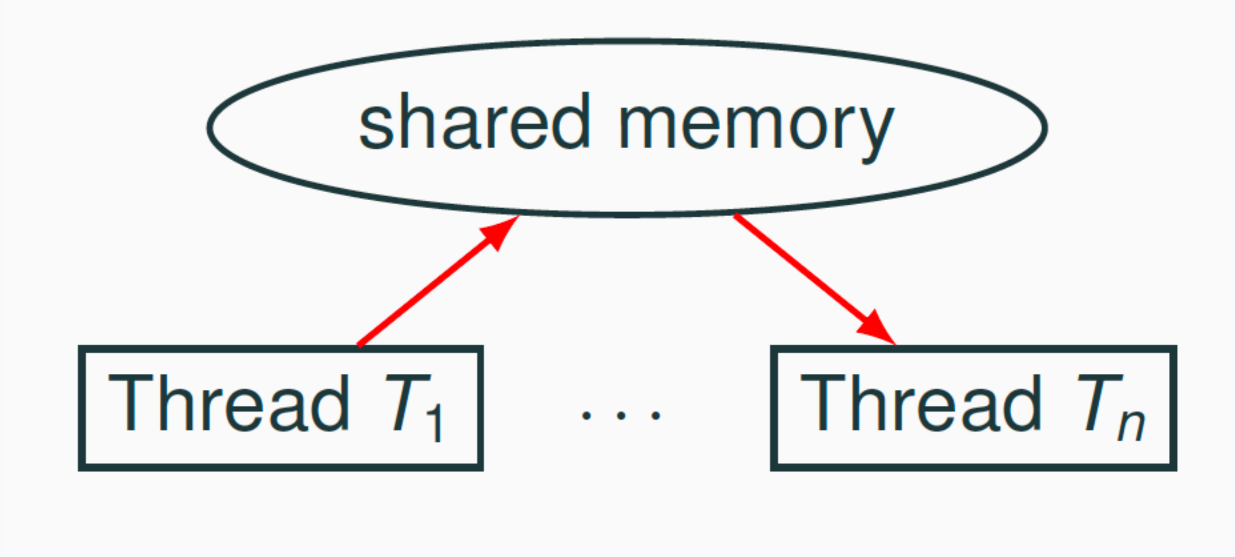
Shared memory vs. Message passing synchronization

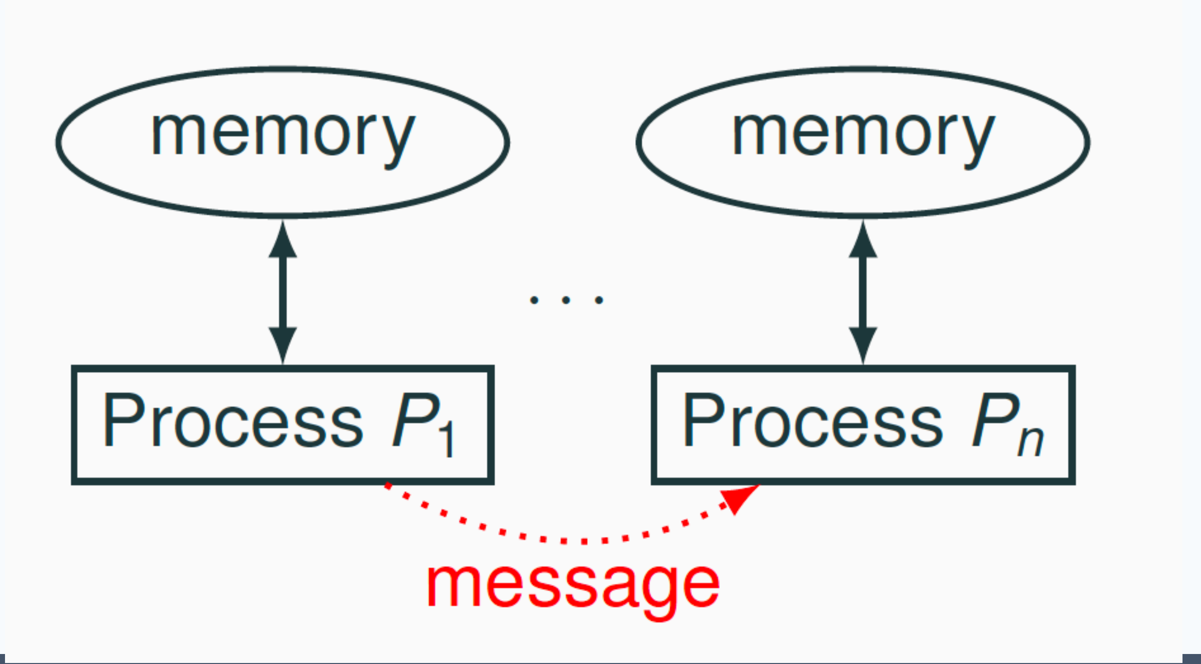
Shared memory synchronization:

- Synchronize by **writing to** and **reading from** shared memory
- Natural choice in shared memory systems such as threads

Message passing synchronization:

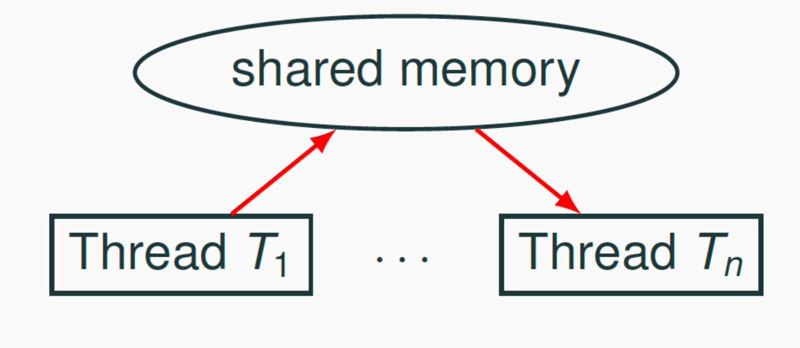
- Synchronize by **exchanging** messages
- Natural choice in distributed memory systems such as processes



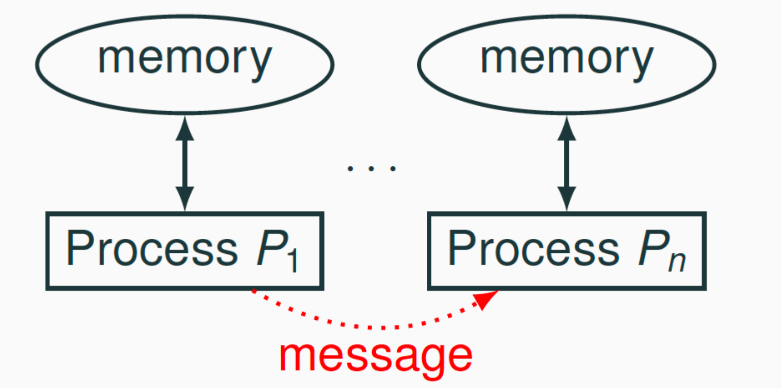


Shared memory vs. Message passing synchronization

Shared memory synchronization:



Message passing synchronization:



The two synchronization models **overlap**:

- Send a message by writing to and reading from shared memory (ex: message board)
- Share information by sending a message (ex: order a billboard)

- In the **first part** of the course we will focus on synchronization problems that arise in shared memory concurrency
- In the **second part** we will switch to message passing

The mutual exclusion problem

A fundamental synchronization problem which arises whenever multiple threads have access to a shared resource

Critical Section: Part of a program that accesses the shared resource (Ex: shared variable)

Mutual Exclusion Property: No more than 1 thread is in its critical section at any given time

Mutual Exclusion Problem: Devise a protocol for accessing a shared resource that satisfies the **mutual exclusion property**

Simplifications to present solutions in a uniform way:

- the critical section is an arbitrary block of code
- threads continuously try to enter the critical section
- threads spend a finite amount of time in the critical section
- we ignore what the threads do outside their critical sections

The mutual exclusion problem

Mutual Exclusion Problem: Devise a protocol for accessing a shared resource that satisfies the **mutual exclusion property**

```
T shared;
```

thread t_j

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

May depend on thread

thread t_k

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

Depends on computation

Mutual exclusion problem example: **Concurrent Counter**

Updating a shared variable **consistently** is an instance of the mutual exclusion problem

```
int counter = 0;
```

thread t

```
int cnt;

while (true) {
  entry protocol
  critical section {
    cnt = counter;
    counter = cnt + 1;
  }
  exit protocol
  return;
}
```

thread u

```
int cnt;

while (true) {
  entry protocol
  critical section {
    cnt = counter;
    counter = cnt + 1;
  }
  exit protocol
  return;
}
```

Take turns incrementing counter

What's a **good solution** to the mutual exclusion problem?

A fully satisfactory solution is one that achieves three properties:

1. Mutual exclusion: at most one thread is in its critical section at any given time
2. Freedom from deadlock: if one or more threads try to enter the critical section, some thread will eventually succeed
3. Freedom from starvation: every thread that tries to enter the critical section will eventually succeed

A good solution should also work for an arbitrary number of threads sharing the same memory

(NOTE: **Freedom from starvation implies freedom from deadlock**)

Deadlocks

A deadlock is the situation where a group of threads **wait forever** because each of them is waiting for resources that are held by another thread in the group (circular dependency)

- A mutual exclusion protocol provides exclusive access to shared resources to one thread at a time
- Threads that try to access the resource when it is not available will have to block and wait
- Mutually dependent waiting conditions may introduce a deadlock

Deadlock: Example

A deadlock is the situation where a group of threads **wait forever** because each of them is waiting for resources that are held by another thread in the group (circular dependency)

A protocol that achieves mutual exclusion but introduces a deadlock:

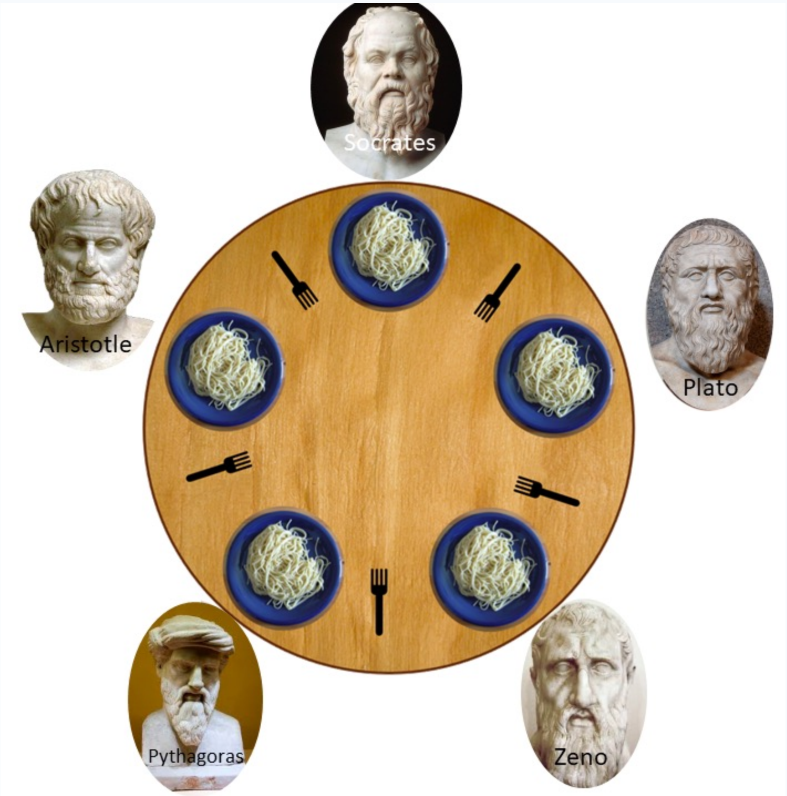
Entry protocol: Wait until all other threads have executed their critical section

Via, resti servita Madama brillante – E. Tommasi Ferroni, 2012

The Dining Philosophers

- Dining philosophers: A classic synchronization problem introduced by Dijkstra
- It illustrates the problem of deadlocks using a colorful metaphor (by Hoare)

- Five philosophers are sitting around a dinner table, with a fork in between each pair of adjacent philosophers
- Each philosopher alternates between thinking (non-critical section) and eating (critical section)
- In order to eat, a philosopher needs to pick up the two forks that lie to the philosopher's left and right
- Since the forks are shared, there is a synchronization problem between philosophers (threads)

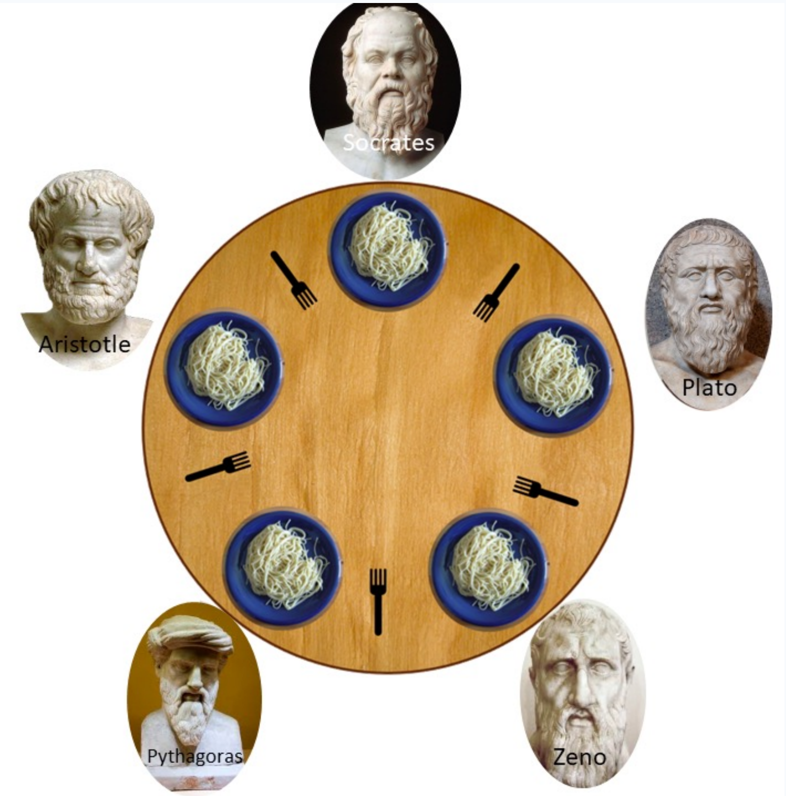


Deadlocking philosophers

An **unsuccessful attempt** at solving the dining philosophers problem:

```
entry  () {
  left_fork.acquire(); // pick up left fork
  right_fork.acquire();// pick up right fork
}
critical section { eat(); }
exit   () {
  left_fork.release(); // release left fork
  right_fork.release();// release right fork
}
```

This protocol deadlocks if all philosophers get their left forks, and wait forever for their right forks to become available



The Coffman conditions

Necessary conditions for a deadlock to occur:

1. Mutual exclusion: threads may have exclusive access to the shared resources
2. Hold and wait: a thread may request one resource while holding another one
3. No preemption: resources cannot forcibly be released from threads that hold them
4. Circular wait: two or more threads form a circular chain where each thread waits for a resource that the next thread in the chain is holding.

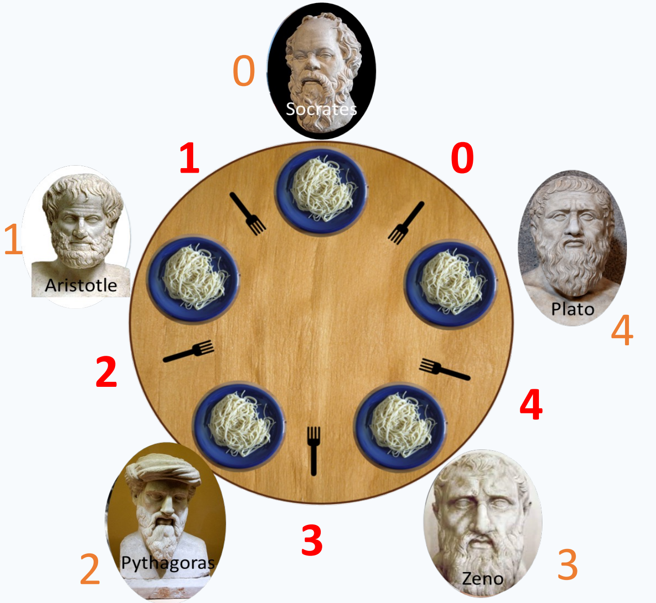
* Avoiding deadlocks requires to break **one or more** of these conditions

Breaking a circular wait

A solution to the dining philosophers problem that avoids deadlock by **breaking *circular wait***: pick up first the fork with the lowest *id* number

It avoids circular wait since not every philosopher will pick up their left fork first

```
entry   () {
  if (left_fork.id()< right_fork.id())
  { left_fork.acquire();
    right_fork.acquire();
  }
  else
  { right_fork.acquire();
    left_fork.acquire();
  }
}
critical section { eat(); }
exit    () { /* ... */ }
```

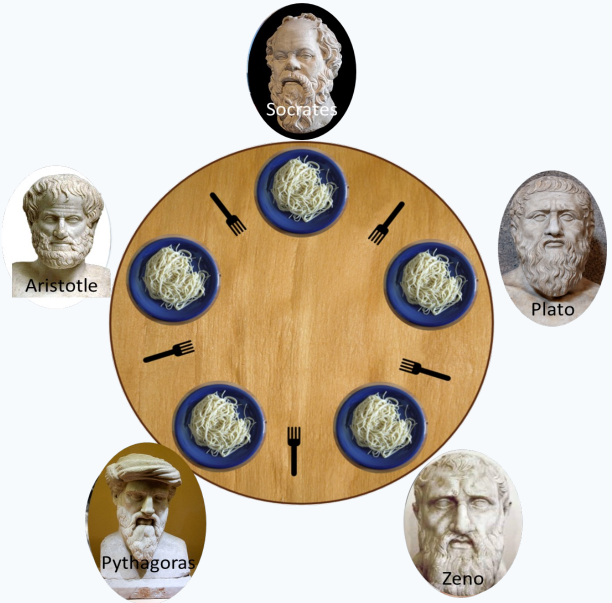


Ordering shared resources and forcing all threads to acquire the resources in order is a **common measure to avoid** deadlocks

Starving philosophers

A solution to the dining philosophers problem that avoids deadlock by **breaking *hold and wait*** (and thus ***circular wait***): pick up both forks at once (atomic op.)

```
entry  () {
  forks.acquire(); // pick up left
and right fork, atomically
}
critical section { eat(); }
exit   () {
  forks.release(); // release left
and right fork,atomically
}
```



It **avoids** deadlock, but it may **introduce** starvation: a philosopher may never get a chance to pick up the forks

Starvation

No deadlock means that the system makes progress as a whole

However, some thread may still make no progress because it is treated unfairly in terms of access to shared resources

Starvation is the situation where a thread is perpetually denied access to a resource it requests

Avoiding starvation requires an additional assumption about the scheduler

Fairness

Starvation is the situation where a thread is perpetually denied access to a resource it requests

Avoiding starvation requires the scheduler to

“give every thread a chance to execute”

Weak fairness: if a thread continuously requests (that is, without interruptions) access to a resource, then access is granted eventually (or infinitely often)

Strong fairness: if a thread requests access to a resource infinitely often, then access is granted eventually (or infinitely often)

Applied to a *scheduler*:

- request = a thread is ready (enabled)
- fairness = every thread has a chance to execute

Sequential philosophers

Other solution to dining philosophers problem to avoid deadlock and starvation:

A (fair) waiter decides which philosopher eats

The waiter gives permission to eat to one philosopher at a time

Having a centralized arbiter avoids deadlocks and starvation, but a waiter who only gives permission to one philosopher a time basically reduces the philosophers to following a sequential order without active concurrency.

```
entry  () {
  while (!waiter.can_eat(k)) {
  // wait for permission to eat
  }
  left_fork.acquire();
  right_fork.acquire();
}
critical section { eat(); }
exit   () { /* ... */ }
```

Having a centralized arbiter **avoids** deadlocks and starvation

- but a waiter who only gives permission to one philosopher a time obliges to follow a sequential order without active concurrency

Locks

Lock objects

A lock is a data structure with interface:

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

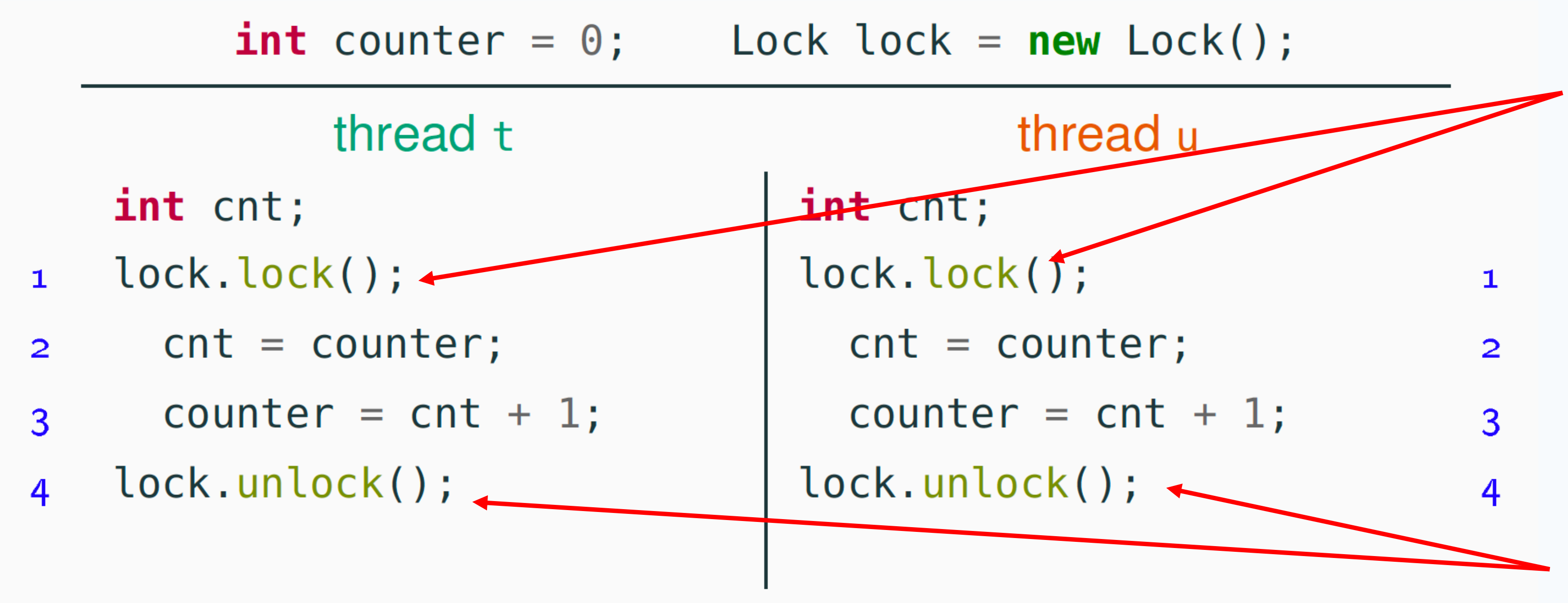
- Several threads share the same object `lock` of type `Lock`
- Many threads calling `lock.lock()`: exactly one thread *t* acquires the lock
  - ***t***'s call `lock.lock()` returns: ***t*** is holding the lock
  - other threads block on the call `lock.lock()`, waiting for the lock to become available
- 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

Locks are also called mutexes (they guarantee mutual exclusion)

Using locks

With lock objects the entry/exit protocols are trivial:

- Entry protocol: call `lock.lock()`
- Exit protocol: call `lock.unlock()`



```
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();
```

Only one thread will succeed in getting the lock

Only when releasing the lock, the other thread can get it

The implementation of the `Lock` interface should **guarantee** mutual exclusion, deadlock freedom, and starvation freedom

Using locks in Java

```
// package with lock-related classes
import java.util.concurrent.locks.*;

// shared with other synchronizing threads
Lock lock;

while (true) {
  lock.lock();      // entry protocol
  try {
    // critical section
    // mutual exclusion is guaranteed
    // by the lock protocol
  }

  finally { // lock released even if an exception
            // is thrown in the critical section
    lock.unlock(); // exit protocol
  }
}
```

Why critical section inside a try-finally?

To avoid holding the lock in case of an exception (blocking all other threads)

Counter with mutual exclusion

```java
public class LockedCounter extends CCounter
{
    @Override
    public void run() {
        lock.lock();
        try  {
            // int cnt = counter;
            // counter = counter + 1;
            super.run();
        }
        finally {
            lock.unlock();
        }
    }
    // shared by all threads working
    // on this object
    private Lock lock = new ReentrantLock();
}
```

Entry protocol

Critical section

Exit protocol

To allow threads lock a resource more than once

The `main` is as before, but instantiates an object of class `LockedCounter`

- What is printed by running:
  `java ConcurrentCount`?
- May the printed value change in different reruns?

NO: Always **2**

Built-in locks in Java

Every object in Java has an implicit lock, which can be accessed using the keyword `synchronized`

Method locking (synchronized methods):

```
synchronized T m() {
  // the critical section
  // is the whole method body
}
```

Every call to `m` implicitly:

1. acquires the lock
2. executes `m`
3. releases the lock

Block locking (synchronized block):

```
synchronized(this) {
  // the critical section
  // is the block's content
}
```

Every execution of the block implicitly:

1. acquires the lock
2. executes the block
3. releases the lock

Counter with mutual exclusion: with **synchronized**

```
public class SyncCounter
  extends CCounter
{
  @Override
  public synchronized
  void run() {
    // int cnt = counter;
    // counter = counter + 1;
    super.run();
    }
}
```

```
public class SyncBlockCounter
  extends CCounter
{
  @Override
  public void run() {
    synchronized (this) {
      // int cnt = counter;
      // counter = counter + 1;
      super.run();
    }
  }
}
```

Lock implementations in Java

The most common implementation of the `Lock` interface in Java is

```
class ReentrantLock
```

Mutual exclusion:

- `ReentrantLock` guarantees mutual exclusion

Starvation:

- `ReentrantLock` does **not** guarantee freedom from starvation by default
- however, calling the constructor with `new ReentrantLock(true)` “favors granting access to the longest-waiting thread”
- this still does not guarantee that thread scheduling is fair

Deadlocks:

- one thread will succeed in acquiring the lock
- however, deadlocks may occur in systems that use multiple locks (remember the dining philosophers)

Built-in lock implementations in Java

Built-in locks – used by `synchronized` methods and blocks – have the **same behavior** as the explicit locks of `java.util.concurrent.locks`

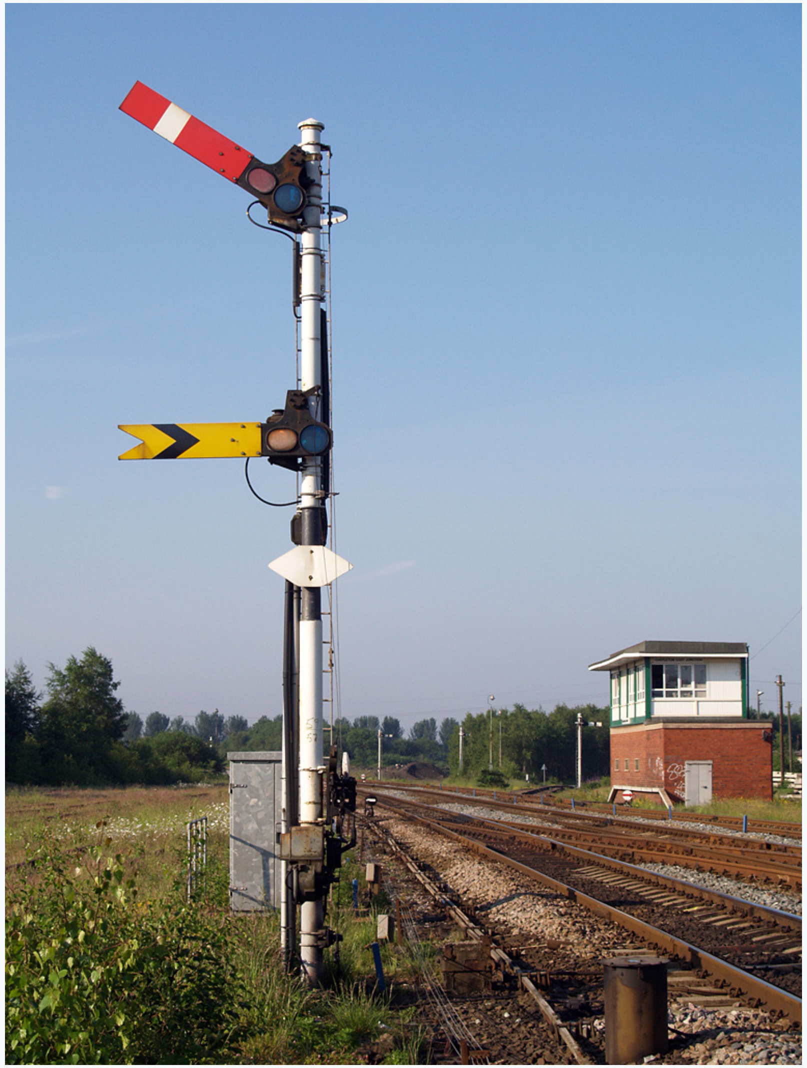
(with no guarantee about starvation)

Built-in locks, and all lock implementations in `java.util.concurrent.locks`

are ***re-entrant***: a thread holding a lock can lock it again without causing a deadlock

Semaphores

Semaphores



* Photo: British railway semaphores   David Ingham, 2008

Semaphores

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`:

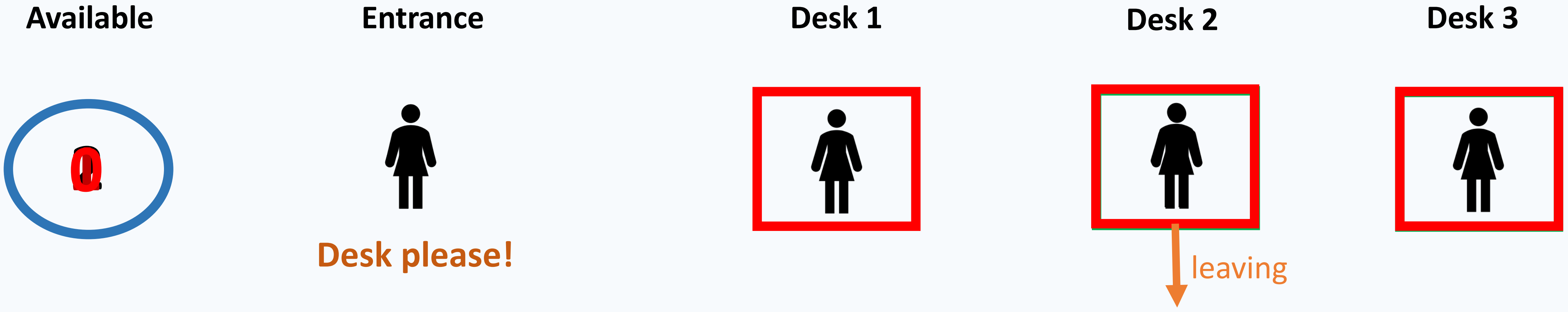
- initially `count` is set to a nonnegative value `C` (the capacity)
- a call to `sem.up()` *atomically* increments `count` by one
- a call to `sem.down()`: ***waits*** until `count` is positive, and then *atomically* decrements `count` by one

Semaphores for permissions

A semaphore is often used to regulate access permits to a **finite** number of resources:

- the capacity `C` is the number of initially available resources
- `up` (also called `signal`) releases a resource, which becomes available
- `down` (also called `wait`) acquires a resource if it is available

Example: **hot desks**



Mutual exclusion for **two** processes with semaphores

With semaphores the entry/exit protocols are trivial:

- initialize semaphore to 1
- entry protocol: call `sem.down()`
- exit protocol: call `sem.up()`

```
Semaphore sem = new Semaphore(1);
```

thread t	thread u
`int cnt;`	`int cnt;`

`sem.down();`
`  cnt = counter;`
`  counter = cnt + 1;`
`sem.up();`
	`sem.down();`
	`  cnt = counter;`
	`  counter = cnt + 1;`
	`sem.up();`

Acts as a lock

The implementation of the `Semaphore` interface **guarantees** mutual exclusion, deadlock freedom, and starvation freedom

Weak vs. strong semaphores

Every implementation of semaphores should **guarantee**:

- the atomicity of the up and down operations
- deadlock freedom (for threads only sharing one semaphore: deadlocks may still occur if there are other synchronization constraints)

Fairness is optional:

Weak semaphore**:** threads waiting to perform down are scheduled nondeterministically

Strong semaphore**:** threads waiting to perform down are scheduled fairly in FIFO (First In First Out) order

Invariants

An object’s invariant is a property that always holds between calls to the object’s methods:

- the invariant holds ***initially*** (when the object is created)
- every method call ***starts*** in a state that satisfies the invariant
- every method call ***ends*** in a state that satisfies the invariant

Ex: A **bank account** that cannot be overdrawn has an invariant `balance >= 0`

```
class BankAccount {
  private int balance = 0;
  void deposit(int amount)
    { if (amount > 0) balance += amount; }
  void withdraw(int amount)
    { if (amount > 0 && balance > amount) balance -= amount; }
}
```

Invariants in pseudo-code

- We may annotate classes with the pseudo-code keyword `invariant`
  - Note that `invariant` is **not** a valid Java keyword – we highlight it in a different color – but we will use it whenever it helps make more explicit the behavior of classes

```
class BankAccount {
  private int balance = 0;
  void deposit(int amount)
    { if (amount > 0) balance += amount; }
  void withdraw(int amount)
    { if (amount > 0 && balance > amount) balance -= amount; }
  invariant{ balance >= 0; } // not valid Java code
}
```

Invariants of semaphores

A semaphore object with *initial capacity* C satisfies the invariant:

```
interface Semaphore {
  int count();
  void up();
  void down();

  invariant{
    count() >= 0;
    count() == C + #up - #down;
  }
}
```

Number of calls to up

up can increment beyond the capacity

Number of calls to down

NOT valid Java code

Invariants **characterize** the behavior of an object, and are very useful for proofs

Binary semaphores

A semaphore with capacity 1 and such that `count()` is always at most 1 is called a binary semaphore

```
interface BinarySemaphore extends Semaphore {
  invariant
  {   0 <= count() <= 1;
      count() == C + #up - #down; }
}
```

Mutual exclusion uses a binary semaphore:

```
Semaphore sem = new Semaphore(1);
// shared by all threads
```

thread t

```
sem.down();
  // critical section
sem.up();
```

If the semaphore is ***strong*** this guarantees starvation freedom

Binary semaphores vs. locks

Binary semaphores are very similar to ***locks*** with one difference:

- In a ***lock***, only the thread that decrements the counter to 0 can increment it back to 1

- In a ***semaphore***, a thread may decrement the counter to 0 and then let another thread increment it to 1

Thus (binary) semaphores support transferring of permissions

Using semaphores in Java

```
  package java.util.concurrent;

  public class Semaphore {

    Semaphore(int permits);
                      // initialize with capacity `permits'
    Semaphore(int permits, boolean fair);
       // fair == true  iff create a strong semaphore
       // fair == false iff create a weak semaphore (default)

    void acquire();           // corresponds to down
    void release();           // corresponds to up
    int availablePermits();   // corresponds to count
  }
```

Method `acquire` may throw an `InterruptedException`: catch or propagate

Synchronization with semaphores

The *k*-exclusion problem

The k-exclusion problem: devise a protocol that allows **up to k threads** to be in their critical sections at the same time

- Mutual exclusion problem = **1**-exclusion problem
- The “hot desk” is an instance of the k-exclusion problem

A **solution** to the k-exclusion problem using a semaphore of capacity k: A straightforward generalization of mutual exclusion

```
Semaphore sem = new Semaphore(k);
// shared by all threads
```

thread t

```
sem.down();
  // critical section
sem.up();
```

Barriers

A **barrier** is a form of synchronization where there is a *point* (the **barrier**) in a program's execution that all threads in a group have to reach **before any of them** is allowed to continue

A **solution** to the barrier synchronization problem for 2 threads using binary semaphores:

```
Semaphore[]  done = {new  Semaphore(0), new Semaphore(0)};
              t0                                 t1

// code before barrier             // code before barrier
done[t0].up();   // t done         done[t1].up();    // u done
done[t1].down(); // wait u         done[t0].down(); // wait t
// code after barrier              // code after barrier
```

up done unconditionally

down waits until the other thread has reached the barrier

Capacity 0 forces up before down

QUIZ
”Races and Locks”