ETH Zürich
Chair of Software Engineering

Software Architecture

Bertrand Meyer, Carlo A. Furia, Martin Nordio

ETH Zurich, February-May 2011

Lecture 13: Designing for concurrency

(Material prepared with Sebastian Nanz)

For more

Several concurrency courses in the ETH curriculum, including our (Bertrand Meyer, Sebastian Nanz) "Concepts of Concurrent Computation" (CCC, Spring semester)

Some of the material here comes from the CCC course.

Good textbooks:

Kramer
Herlihy

Why is concurrency so important?

Traditionally, specialized area of interest to a few experts:

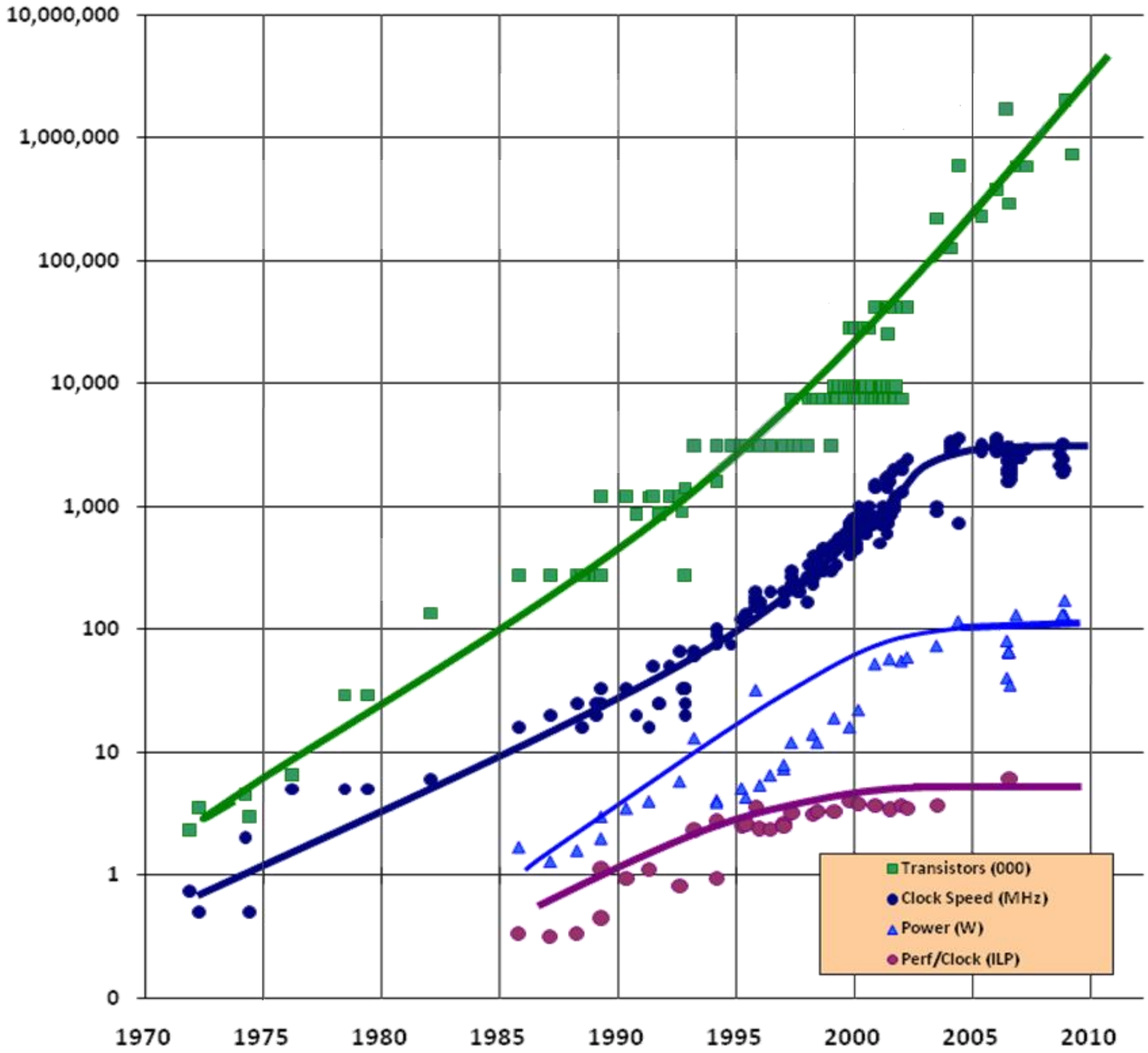
- Operating systems
- Networking
- Databases

Multicore and the Internet make it relevant to every programmer!

What they say about concurrency

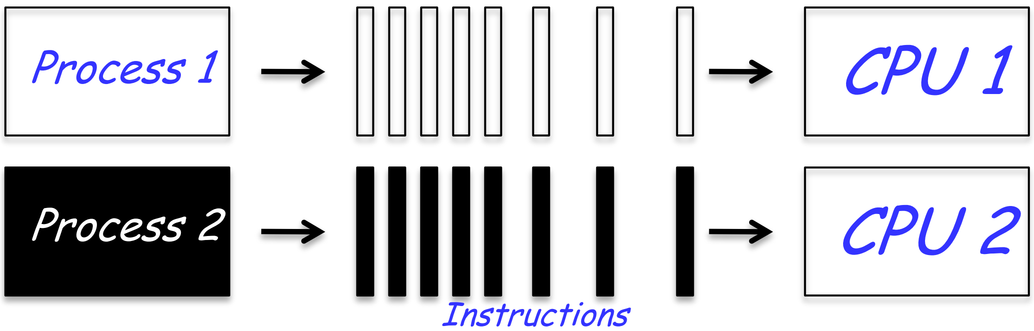
- Intel Corporation: *Multi-core processing is taking the industry on a fast-moving and exciting ride into profoundly new territory. The defining paradigm in computing performance has shifted inexorably from raw clock speed to parallel operations and energy efficiency.*
- Rick Rashid, head of Microsoft Research: *Multicore processors represent one of the largest technology transitions in the computing industry today, with deep implications for how we develop software.*
- Bill Gates: "*Multicore: This is the one which will have the biggest impact on us. We have never had a problem to solve like this. A breakthrough is needed in how applications are done on multicore devices.*

Evolution of hardware (source: Intel)



Multiprocessing

- Until a few years ago: systems with one processing unit were standard
- Today: most end-user systems have multiple processing units in the form of multi-core processors

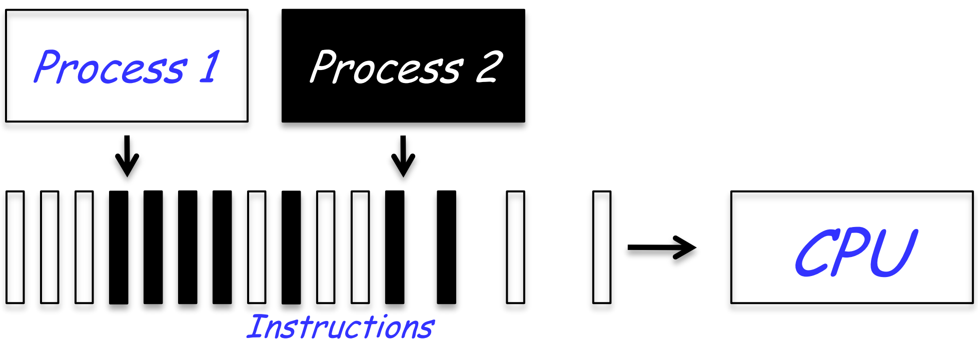


- *Multiprocessing*: the use of more than one processing unit in a system
- Execution of processes is said to be *parallel*, as they are running at the same time

Multitasking & multithreading

Even on systems with a single processing unit programs may appear to run in parallel:

- Multitasking*
- Multithreading (within a process, see in a few slides)



Multi-tasked execution of processes is said to be *interleaved*, as all are in progress, but only one is running at a time. (Closely related concept: **coroutines**.)

**This is common terminology, but "multiprocessing" was also used previously as a synonym for "multitasking"*

Processes

- A (sequential) *program* is a set of instructions
- A *process* is an instance of a program that is being executed

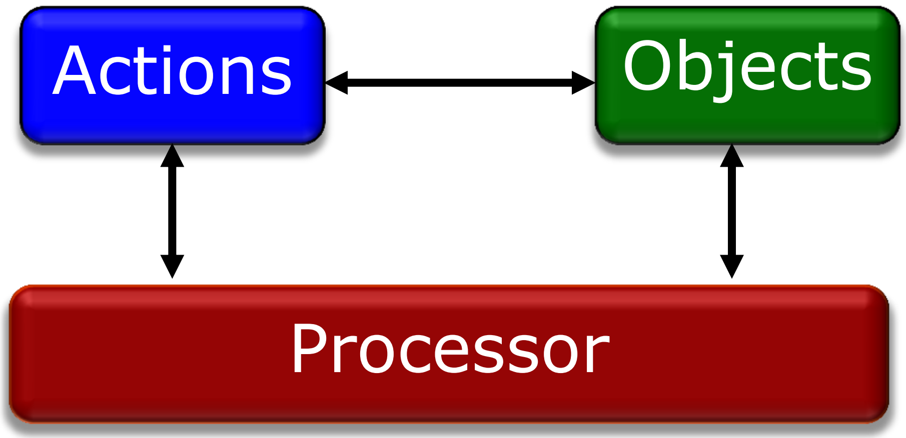
Concurrency

- Both multiprocessing and multithreading are examples of concurrent computation
- The execution of processes or threads is said to be *concurrent* if it is either parallel or interleaved

Computation

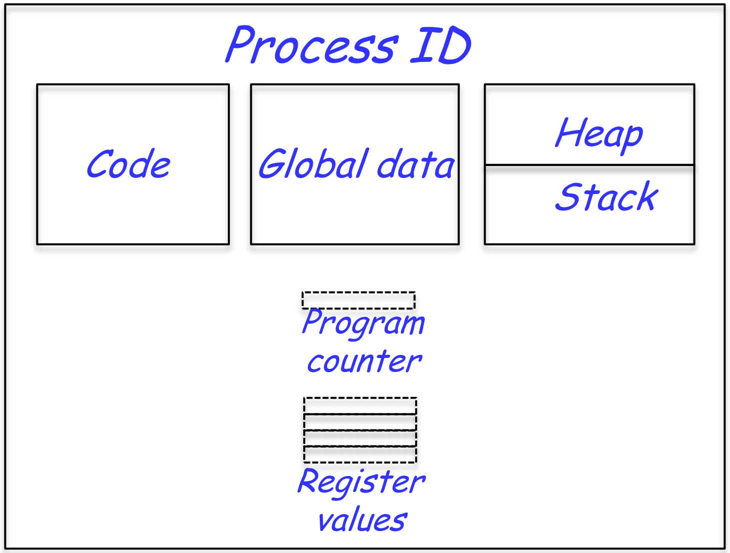
To perform a computation is

- To apply certain actions
- To certain objects
- Using certain processors



Operating system processes

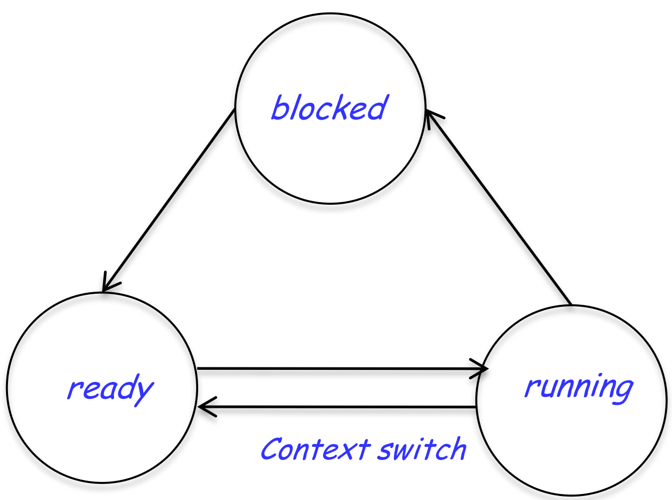
- How are processes implemented in an operating system?
- Structure of a typical process:
  - *Process identifier:* unique ID of a process.
  - *Process state:* current activity of a process.
  - *Process context:* program counter, register values
  - *Memory:* program text, global data, stack, and heap.



The scheduler

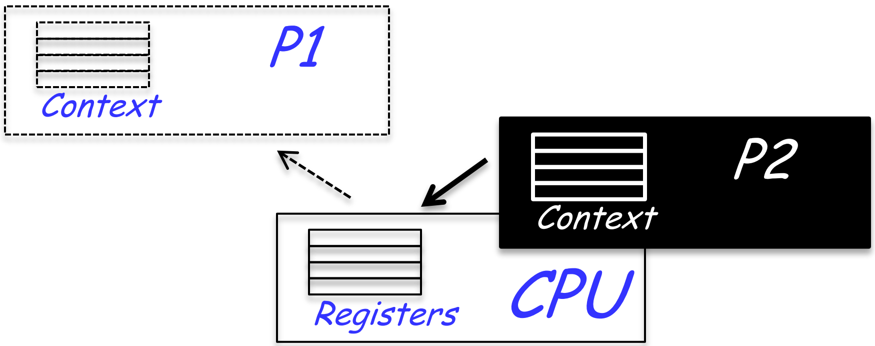
A system program called the *scheduler* controls which processes are running; it sets the process states:

- *Running*: instructions are being executed.
- *Blocked:* currently waiting for an event.
- *Ready:* ready to be executed, but has not been assigned a processor yet.



The context switch

- The swapping of processes on a processing unit by the scheduler is called a *context switch*



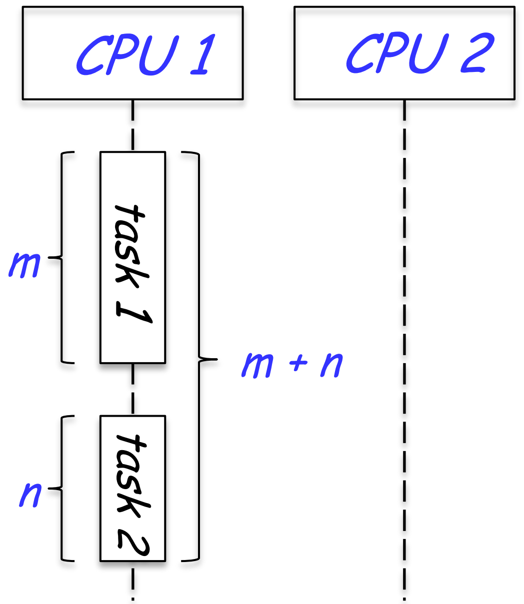
- Scheduler actions when switching processes P1 and P2:
  - P1.set_state (ready)
  - Save register values as P1's context in memory
  - Use context of P2 to set register values
  - P2.set_state (running)

Concurrency within programs

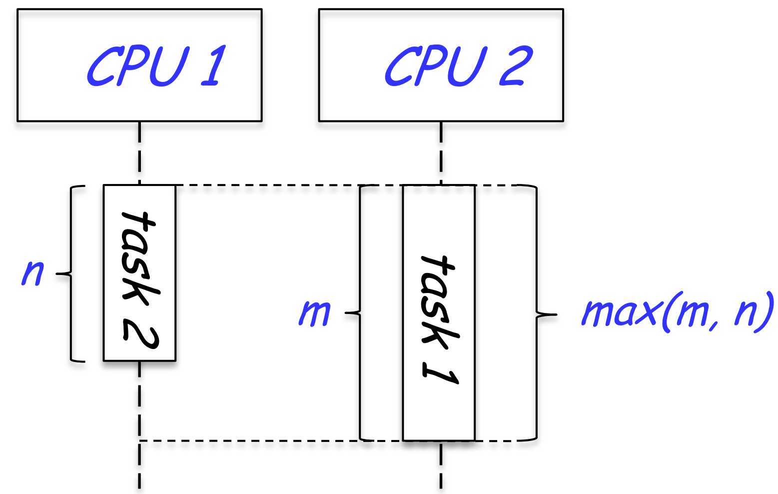
- We also want to use concurrency within programs

```
compute
    do
        t1.do_task1
        t2.do_task2
    end
```

Sequential execution:



Concurrent execution:



Threads ("lightweight processes")

Make programs concurrent by associating them with threads

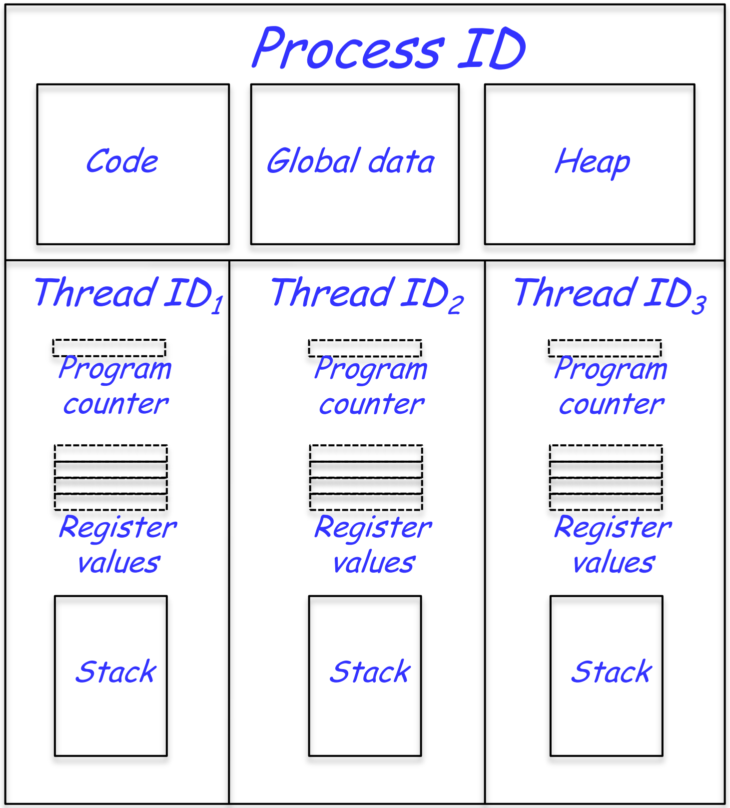
A *thread* is a part of an operating system process

Private to each thread:

- Thread identifier
- Thread state
- Thread context
- Memory: only stack

Shared with other threads:

- Program text
- Global data
- Heap



Processes vs threads

Process:

- Has its own (virtual) memory space (in O-O programming, its own objects)
- Sharing of data (objects) with another process:
  - Is explicit (good for reliability, security, readability)
  - Is heavy (bad for ease of programming)
- Switching to another process: expensive (needs to back up one full context and restore another

Thread:

- Shares memory with other threads
- Sharing of data is straightforward
  - Simple go program (good)
  - Risks of confusion and errors: data races (bad)
- Switching to another thread: cheap

Amdahl's Law

Sequential fraction

Parallel fraction

$$speedup = \frac{1}{1 - p + \frac{p}{n}}$$

Number of processors

Example

- Ten processors
- 60% concurrent, 40% sequential
- How close to 10-fold speedup?

$$speedup = \frac{1}{1 - 0.6 + \frac{0.6}{10}} = 2.17$$

Source (this slide and next three): M. Herlihy

Example

- Ten processors
- 80% concurrent, 20% sequential
- How close to 10-fold speedup?

$$speedup = \frac{1}{1-0.8+\frac{0.8}{10}} = 3.57$$

Example

- Ten processors
- 90% concurrent, 10% sequential
- How close to 10-fold speedup?

$$speedup = \frac{1}{1 - 0.9 + \frac{0.9}{10}} = 5.26$$

Example

- Ten processors
- 99% concurrent, 1% sequential
- How close to 10-fold speedup?

$$speedup = \frac{1}{1 - 0.99 + \frac{0.99}{10}} = 9.17$$

Concurrent programs in Java

Associating a computation with a thread:

```
class Thread1 extends Thread {
    public void run() {
        // implement task1 here
    }
}
class Thread2 extends Thread {
    public void run() {
        // implement task2 here
    }
}
```

```
void compute() {
    Thread1 t1 = new Thread1();
    Thread2 t2 = new Thread2();
    t1.start();
    t2.start();
}
```

- Write a class that inherits from the **class Thread** (or implements the **interface Runnable**)
- Implement the method run()

Joining threads

Often the final results of thread executions need to be combined:

```
return t1.getResult() + t2.getResult();
```

To wait for both threads to be finished, we *join* them:

```
t1.start();
t2.start();
t1.join();
t2.join();
return t1.getResult() + t2.getResult();
```

The join() method, invoked on a thread *t*, causes the caller to wait until *t* is finished

Race conditions (1)

Consider a counter class:

```
class Counter {
    private int value = 0;

    public int getValue() {
        return value;
    }

    public void setValue(int someValue) {
        value = someValue;
    }

    public void increment() {
        value++;
    }
}
```

Assume two threads:

Thread 1:

```
x.setValue(0);
x.increment();
int i = x.getValue();
```

Thread 2:

```
x.setValue(2);
```

Race conditions (2)

- Because of the interleaving of threads, various results can be obtained:

x.setValue(2) x.setValue(0) x.increment() int i = x.getValue()	x.setValue(0) x.setValue(2) x.increment() int i = x.getValue()	x.setValue(0) x.increment() x.setValue(2) int i = x.getValue()	x.setValue(0) x.increment() int i = x.getValue() x.setValue(2)
i == 1 x.value == ?	i == 3 x.value == ?	i == 2 x.value == ?	i == 1 x.value == ?

Such dependence of the result on nondeterministic interleaving is a ***race condition*** (or ***data race***)

Such errors can stay hidden for a long time and are difficult to find by testing

Race conditions (2)

- Because of the interleaving of threads, various results can be obtained:

x.setValue(2) x.setValue(0) x.increment() int i = x.getValue()	x.setValue(0) x.setValue(2) x.increment() int i = x.getValue()	x.setValue(0) x.increment() x.setValue(2) int i = x.getValue()	x.setValue(0) x.increment() int i = x.getValue() x.setValue(2)
i == 1 x.value == **1**	i == 3 x.value == **3**	i == 2 x.value == **2**	i == 1 x.value == **2**

Such dependence of the result on nondeterministic interleaving is a ***race condition*** (or ***data race***)

Such errors can stay hidden for a long time and are difficult to find by testing

Synchronization

To avoid data races, threads (or processes) must *synchronize* with each other, i.e. communicate to agree on the appropriate sequence of actions

How to communicate:

- By reading and writing to shared sections of memory (*shared memory synchronization*)
  In the example, threads should agree that at any one time at most one of them can access the resource
- By explicit exchange of information (*message passing synchronization*)

Mutual exclusion

Mutual exclusion (or "mutex") is a form of synchronization that avoids the simultaneous use of a shared resource

To identify the program parts that need attention, we introduce the notion of a *critical section* : a part of a program that accesses a shared resource, and should normally be executed by at most one thread at a time

Mutual exclusion in Java

- Each object in Java has a mutex lock (can be held only by one thread at a time!) that can be acquired and released within **synchronized** blocks:
- ```
  Object lock = new Object();

  synchronized (lock) {
      // critical section
  }
  ```
- The following are equivalent:

```
synchronized type m(args) {

    // body

}
```

```
type m(args) {
    synchronized (this) {
        // body
    }
}
```
```

Example: mutual exclusion

To avoid data races in the example, we enclose instructions to be executed atomically in synchronized blocks protected with the same lock objects

```
synchronized (lock) {
    x.setValue(0);
    x.increment();
    int i = x.getValue();
}
```

```
synchronized (lock) {
    x.setValue(2);
}
```

The producer-consumer problem

Consider two types of looping processes:

- *Producer:* At each loop iteration, produces a data item for consumption by a consumer
- *Consumer:* At each loop iteration, consumes a data item produced by a producer

Producers and consumers communicate via a shared buffer (a generalized notion of bounded queue)

Producers append data items to the back of the queue and consumers remove data items from the front

Condition synchronization

The producer-consumer problem requires that processes access the buffer properly:

- Consumers must wait if the buffer is empty
- Producers must wait if the buffer is full

Condition synchronization is a form of synchronization where processes are delayed until a condition holds

In producer-consumer we use two forms of synchronization:

- Mutual exclusion: to prevent races on the buffer
- Condition synchronization: to prevent improper access to the buffer

Condition synchronization in Java (2)

- The following methods can be called on a synchronized object (i.e. only within a synchronized block, on the lock object):
  - **wait()**: block the current thread and release the lock until some thread does a **notify()** or **notifyAll()**
  - **notify()**: resume one blocked thread (chosen nondeterministically), set its state to "ready"
  - **notifyAll()**: resume all blocked threads

- No guarantee that the notification mechanism is fair

Producer-Consumer problem: Consumer code

```
public void consume() throws InterruptedException {
    int value;
    synchronized (buffer) {
        while (buffer.size() == 0) {
            buffer.wait();
        }
        value = buffer.get();
    }
}
```

Consumer blocks if `buffer.size() == 0` is true (waiting for a `notify()` from the producer)

Producer-Consumer problem: Producer code

```
public void produce() {
    int value = random.produceValue();
    synchronized (buffer) {
        buffer.put(value);
        buffer.notify();
    }
}
```

Producer notifies consumer that the condition `buffer.size() == 0` is no longer true

The problem of deadlock

The ability to hold resources exclusively is central to providing process synchronization for resource access

Unfortunately, it brings about other problems!

A *deadlock* is the situation where a group of processes blocks forever because each of the processes is waiting for resources which are held by another process in the group

Deadlock example in Java

Consider the class

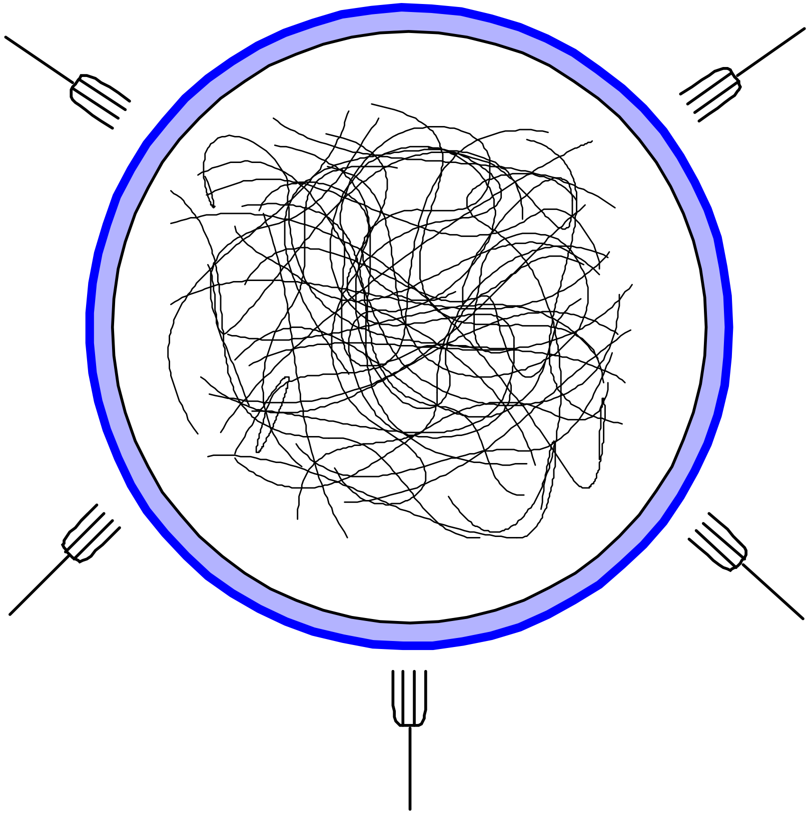
```
public class C extends Thread {
    private Object a;
    private Object b;

    public C(Object x, Object y) {
        a = x;
        b = y;
    }
    public void run() {
        synchronized (a) {
            synchronized (b) {
                ...
            }
        }
}}
```

... and this code being executed:

```
C t1 = new C(a1, b1);
C t2 = new C(b1, a1);
t1.start();
t2.start();
```

Dining philosophers



Are deadlock & data races of the same kind?

No

Two kinds of concurrency issues (Lamport):

- Safety: no bad thing will happen
- Liveness: some good thing will happen

Data from the field

Source for the next few slides:

Learning from Mistakes –
Real World Concurrency Bug Characteristics

Yuanyuan(YY) Zhou
University of Illinois, Urbana-Champaign

Microsoft Faculty Summit, 2008
See also her paper at ASPLOS 2008

Zhou study

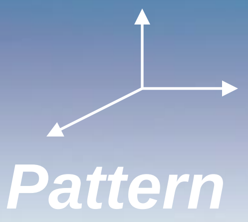
- 105 real-world concurrency bugs from 4 large open-source programs

	MySQL	***Apache***	***Mozilla***	***OpenOffice***
Software Type	*Server*	*Server*	*Client*	*GUI*
Language	*C++/C*	*Mainly C*	*C++*	*C++*
LOC (M line)	*2*	*0.3*	*4*	*6*
Bug history	*6 years*	*7 years*	*10 years*	*8 years*

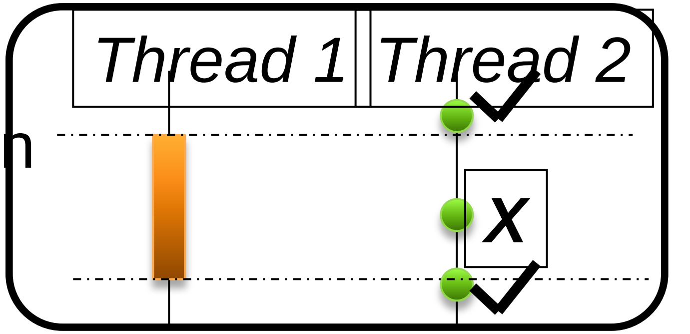
Methodology: Bug Sources

	MySQL	Apache	Mozilla	OpenOffice	Total
Non-deadlock	14	13	41	6	74
Deadlock	9	4	16	2	31

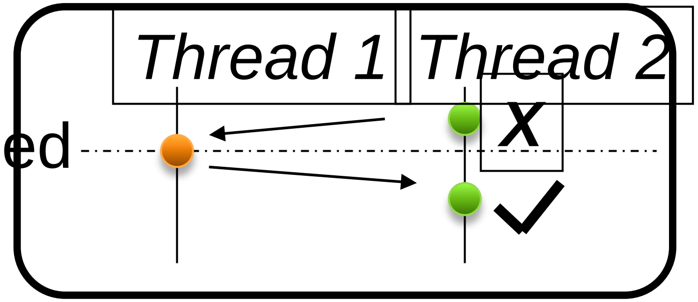
Non-Deadlock Bug Pattern



- Classified based on root causes
- Categories
  - Atomicity violation
    - The desired atomicity of certain code region is violated

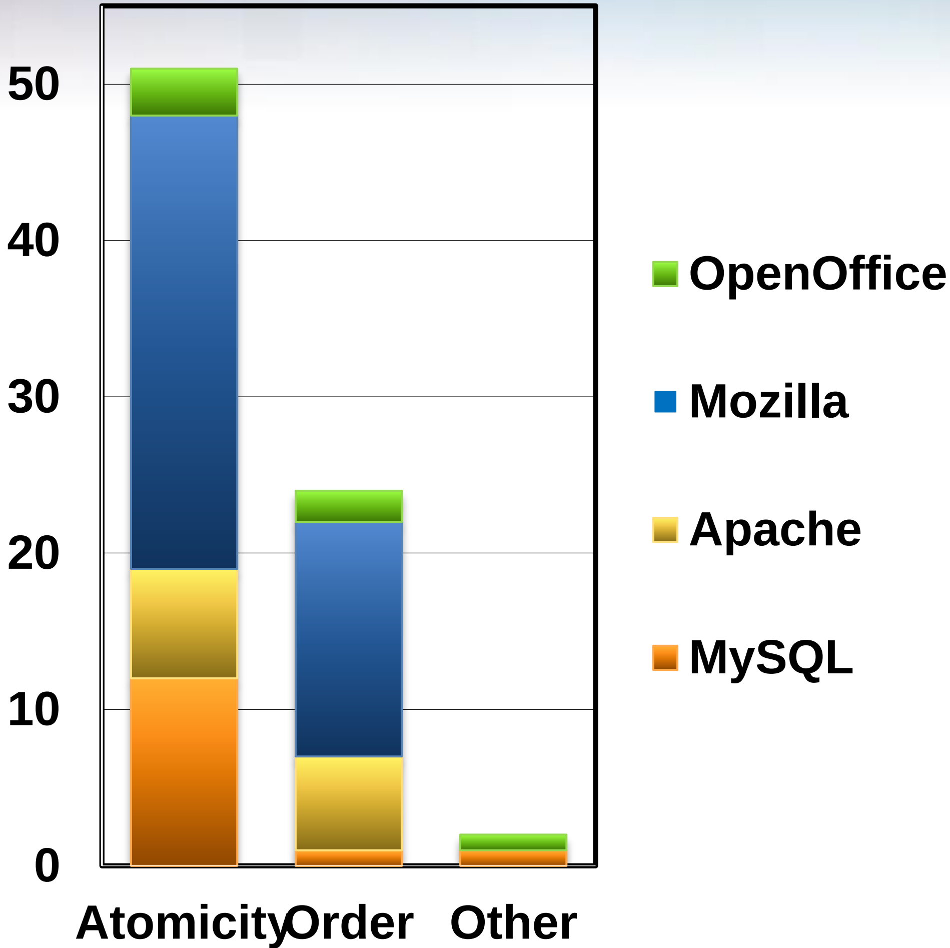


  - Order violation
    - The desired order between two (sets of) accesses is flipped



  - Others

Non-Deadlock Bug Pattern Characteristics

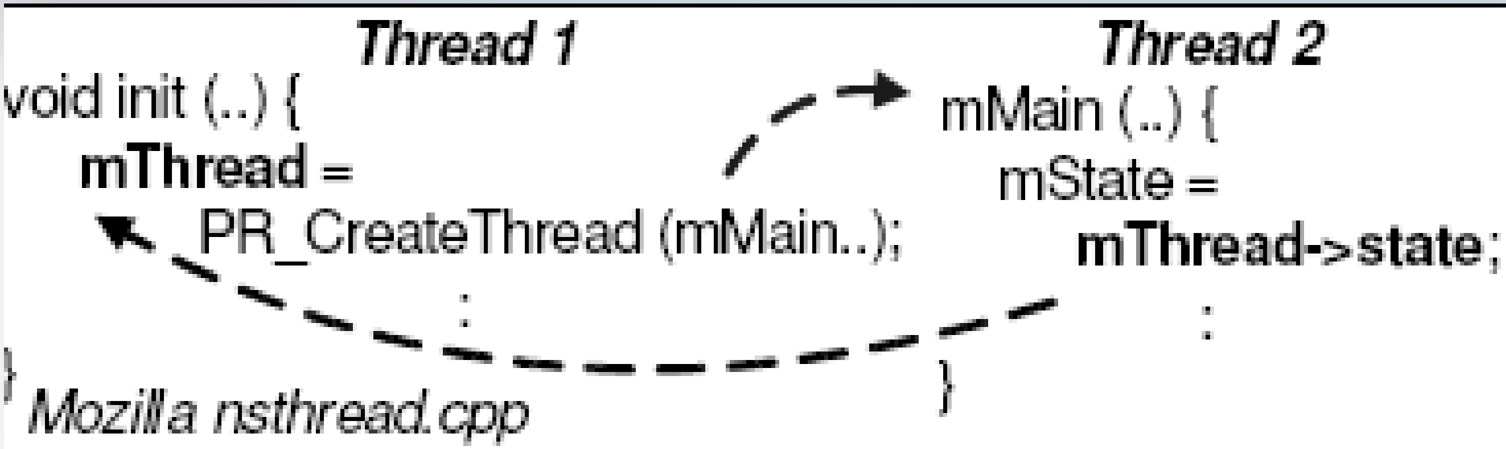


There are 3-bug overlap between Atomicity and Order

Implications

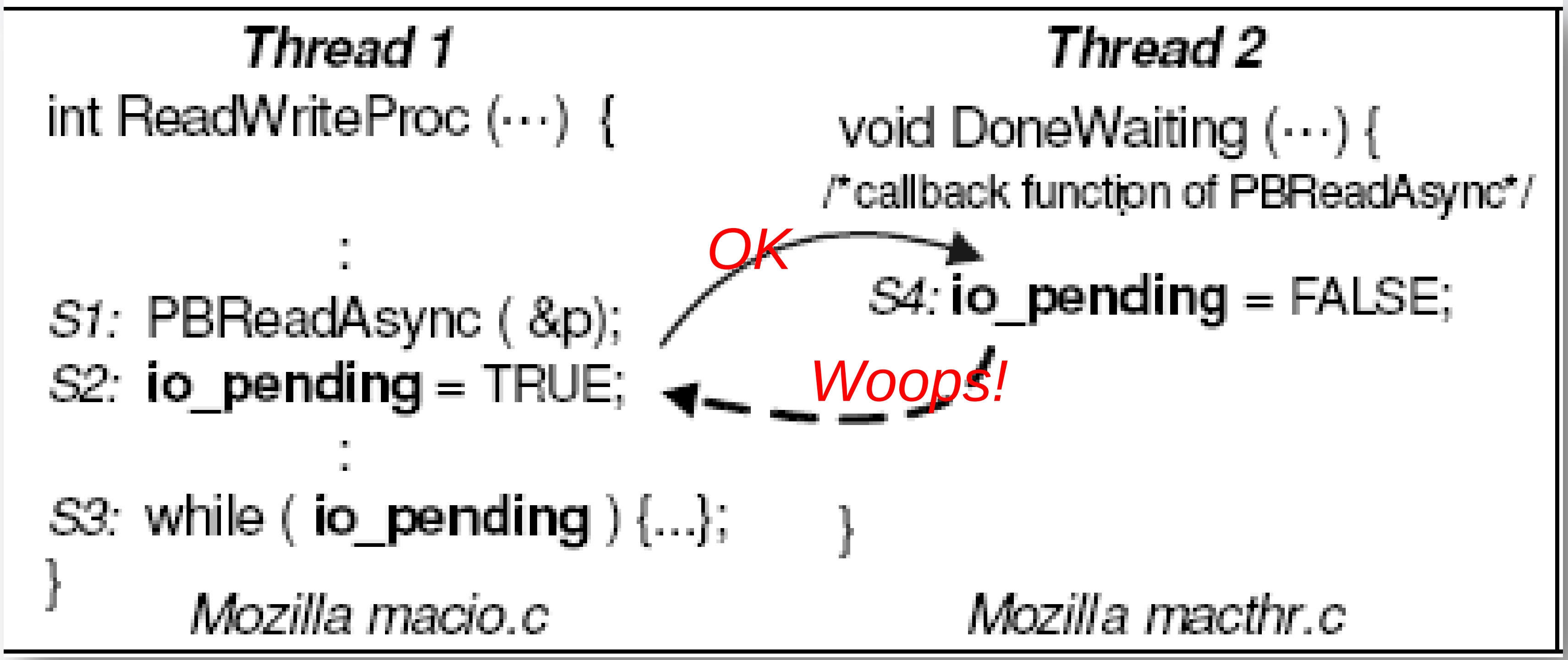
- We should focus on atomicity violation and order violation
- Bug detection tools for order violation bugs are desired

An Order-Related Bug Example



Note that order violations can be fixed by adding locks to ensure atomicity with the previous operation to ensure order. But the root cause is the incorrect assumption about execution order.

Another Example



Number Threads Involved

- 101 out of 105 (96%) bugs involve at most two threads
  - Most bugs can be reliably disclosed if we check all possible interleaving between each pair of threads
- Few bugs cannot
  - Example: Intensive resource competition among many threads causes unexpected delay

SCOOP mechanism

Simple Concurrent Object-Oriented Programming

Evolved through the last two decades

- Comm. ACM paper (1993)
- Chap. 30 of *Object-Oriented Software Construction*, 2nd edition, 1997
- Piotr Nienaltowski's ETH thesis, 2008
- Current work by Sebastian Nanz, Benjamin Morandi, Scott West and other at ETH
- Prototype implementation at ETH
- New implementation (EiffelStudio 6.8)

SCOOP preview: a sequential program

```
transfer (source, target:                    ACCOUNT;
            amount: INTEGER)
        -- If possible, transfer amount from source to target.
    do
        if source.balance >= amount then
            source.withdraw  (amount)
            target.deposit   (amount)
        end
    end
```

Typical calls:

```
transfer (acc1, acc2, 100)
transfer (acc1, acc3, 100)
```

In a concurrent setting, using SCOOP

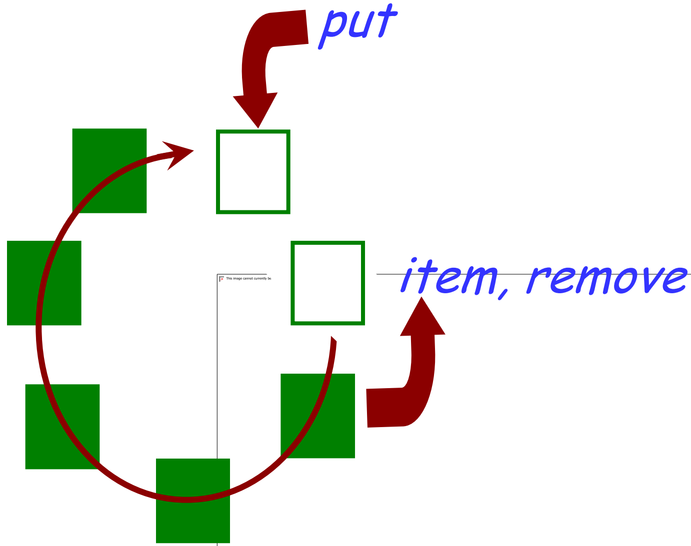
```
transfer (source, target: separate ACCOUNT;
            amount: INTEGER)
        -- If possible, transfer amount from source to target.
    do
        if source.balance >= amount then
            source.withdraw  (amount)
            target.deposit   (amount)
        end
    end
```

Typical calls:

```
transfer (acc1, acc2, 100)
transfer (acc1, acc3, 100)
```

A better SCOOP version

```
transfer (source, target: separate ACCOUNT;
            amount: INTEGER)
        -- Transfer amount from source to target.
    require
        source.balance >= amount
    do
        source.withdraw  (amount)
        target.deposit   (amount)
    ensure
        source.balance = old source.balance - amount
        target.balance = old target.balance + amount
    end
```




```
my_queue : QUEUE BUFFER [T]
...
if not my_queue.is_full then

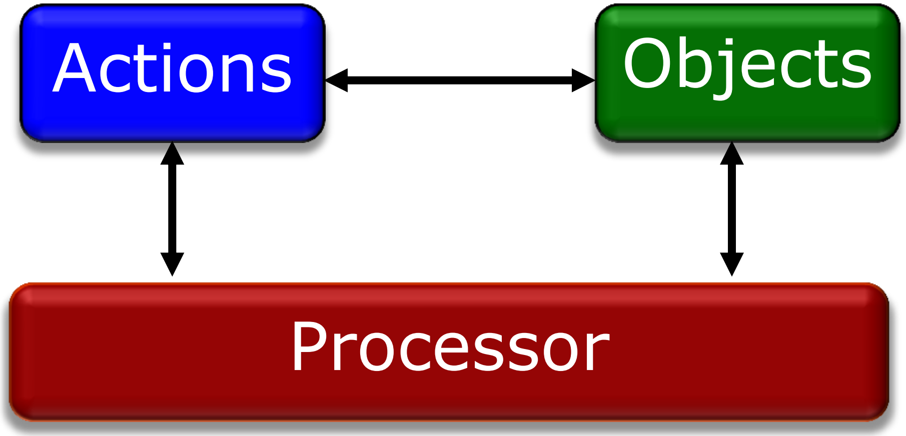
        put (my_queue, t)
end
```

```
put (b: QUEUE BUFFER [G ] ; v: G )
        -- Store v into b.
    require
        not b.is_full
    do
        ...
    ensure
        not b.is_empty
    end
```

Object-oriented computation

To perform a computation is

- To apply certain actions
- To certain objects
- Using certain processors

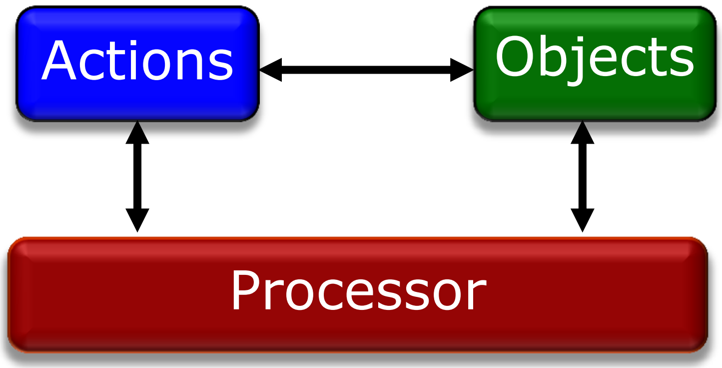


What makes an application concurrent?

Processor:
Thread of control supporting sequential execution of instructions on one or more objects

Can be implemented as:

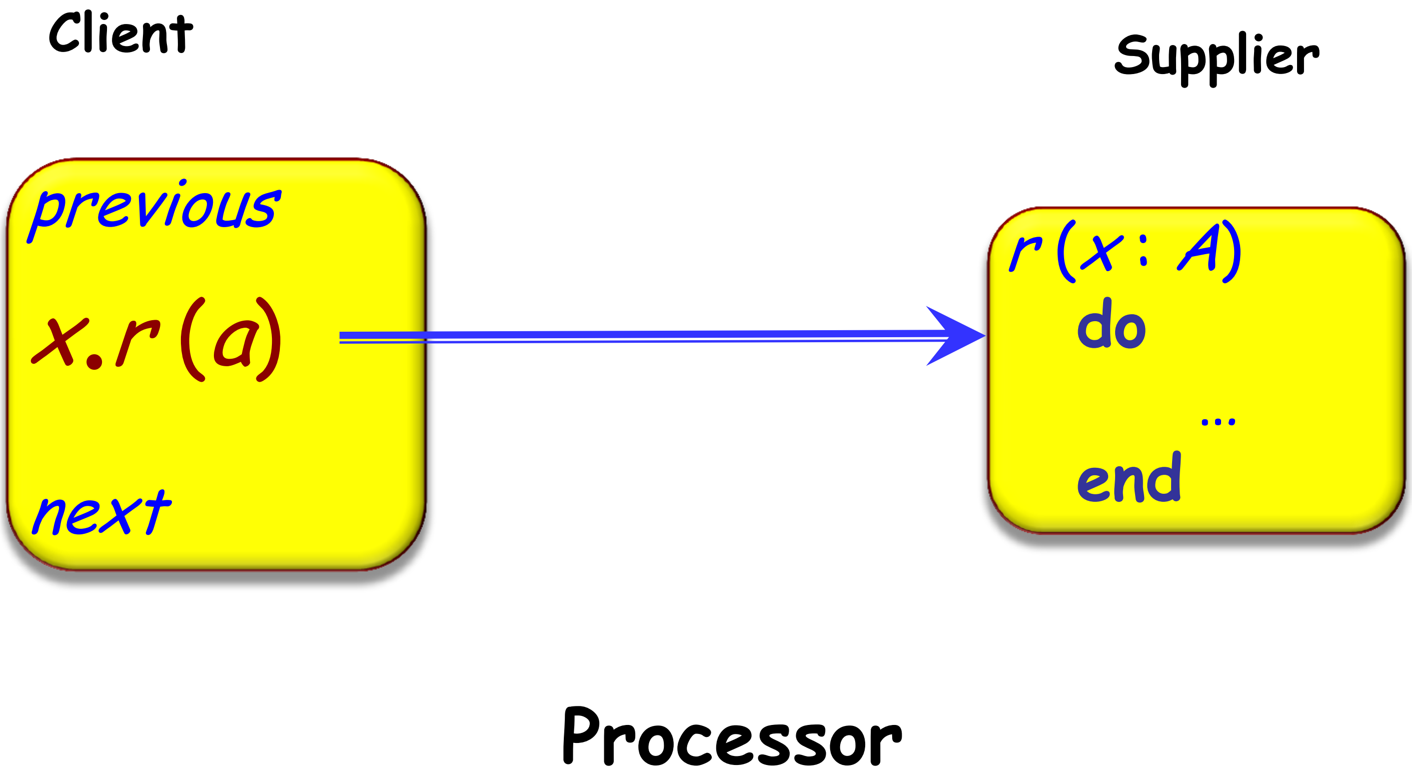
- Computer CPU
- Process
- Thread
- AppDomain (.NET) …



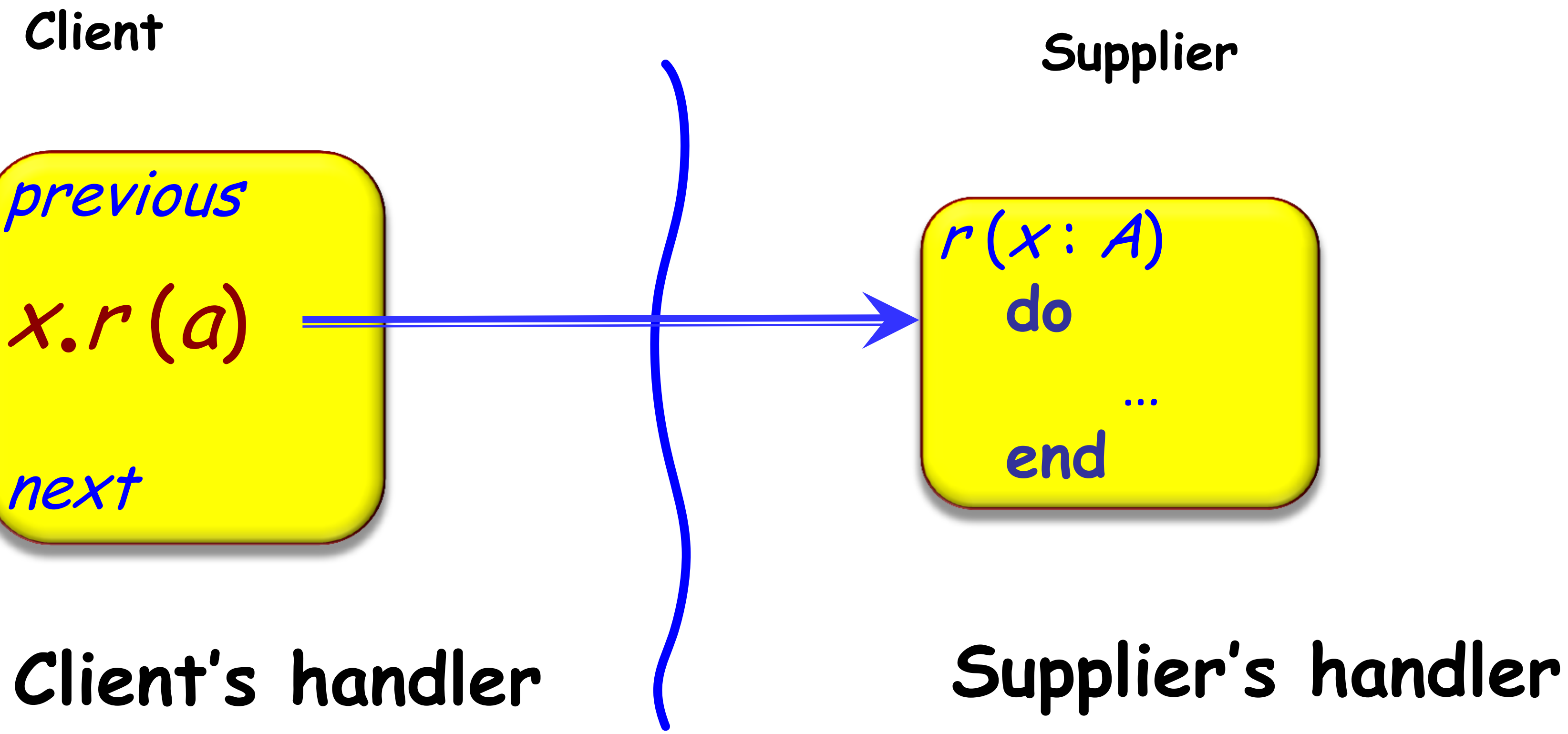
Will be mapped to computational resources

Feature call: sequential

$x.r\,(a)$



Feature call: asynchronous



The fundamental difference

To wait or not to wait:

- If same processor, synchronous
- If different processor, asynchronous

Difference must be captured by syntax:

- x: T
- x: **separate** T -- Potentially different processor

Fundamental semantic rule: *x.r (a)* waits for non-separate *x*, doesn't wait for separate *x*.

Consistency rules: avoiding traitors

nonsep: *T*

sep: **separate** *T*

nonsep := *sep*

Traitor!

nonsep.*p* (*a*)

Wait by necessity

No explicit mechanism needed for client to resynchronize with supplier after separate call.

The client will wait only when it needs to:

x.f

x.g (a)

y.f

…

value := x.some_query

Wait here!

Lazy wait (Denis Caromel, wait by necessity)

Separate argument rule (1)

Target of a separate call must be formal argument of enclosing routine:

```
put (b: separate BUFFER [T]; value : T)
        -- Store value into buffer.
    do
        b.put (value)
    end
```

To use separate object:

```
buffer : separate BUFFER [INTEGER]
create buffer
put (buffer, 10)
```

Separate argument rule (2)

The target of a separate call must be an argument of the enclosing routine

Separate call: *x.f* (...) where *x* is separate

Wait rule

A routine call with separate arguments
will execute when all corresponding processors
are available

and hold them exclusively
for the duration of the routine

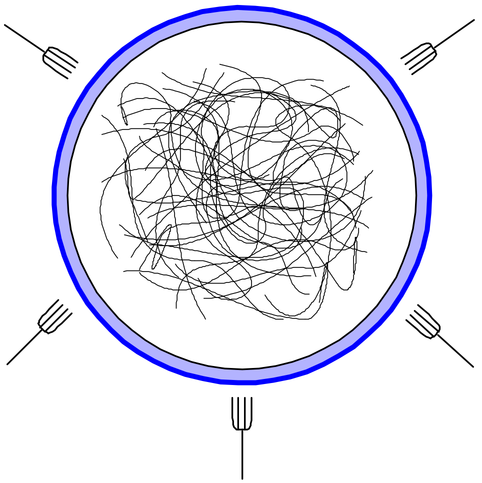
- Since all processors of separate arguments are locked and held for the duration of the routine, mutual exclusion is provided for the corresponding objects

Dining philosophers

```
class PHILOSOPHER inherit
      PROCESS
            rename
                  setup as getup
            redefine step end

feature {BUTLER}
      step
            do
                  think ;   eat (left, right)
            end

      eat (l, r: separate FORK)
                  -- Eat, having grabbed l and r.
            do … end
end
```



Typical traditional (non-SCOOP) code

Listing 4.33: Variables for Tanenbaum's solution

```
state = ['thinking'] * 5
sem = [Semaphore(0) for i in range(5)]
mutex = Semaphore(1)
```

The initial value of `state` is a list of 5 copies of `'thinking'`. `sem` is a list of 5 semaphores with the initial value 0. Here is the code:

Listing 4.34: Tanenbaum's solution

```
def get_fork(i):
    mutex.wait()
    state[i] = 'hungry'
    test(i)
    mutex.signal()
    sem[i].wait()

def put_fork(i):
    mutex.wait()
    state[i] = 'thinking'
    test(right(i))
    test(left(i))
    mutex.signal()

def test(i):
    if state[i] == 'hungry' and
    state (left (i)) != 'eating' and
    state (right (i)) != 'eating':
        state[i] = 'eating'
        sem[i].signal()
```

Condition synchronization

- SCOOP has an elegant way of expressing condition synchronization by reinterpreting preconditions as wait conditions
- Completed wait rule:

> A call with separate arguments waits until:
> - The corresponding objects are all available
> - Preconditions hold

Producer-consumer problem: consumer code

```
item (b: separate BUFFER [T]): T
    require
        not b.is_empty
    do
        Result := b.item
    end
```

Precondition becomes **wait condition**

- Consumer blocks itself if the condition buffer.size() == 0 is found to be true (waiting for a notify() from the producer)

Producer-Consumer problem: Producer code

```
put (b: separate BUFFER [T]; v: T)
   require
      not b.is_full
   local
      value: INTEGER
   do
      b.put (v)
   end
```

- Very easy to provide a solution for bounded buffers
- No need for notification, the SCOOP scheduler ensures that preconditions are automatically reevaluated at a later time

Contracts

```
put (buf : separate QUEUE [INTEGER ] ; v : INTEGER)
        -- Store v into buffer.
    require
        not buf.is_full
        v > 0
    do
        buf.put (v)
    ensure
        not buf.is_empty
    end

...
put (my_buffer, 10 )
```

Precondition becomes **wait condition**

For more

Several concurrency courses in the ETH curriculum, including our (Bertrand Meyer, Sebastian Nanz) "Concepts of Concurrent Computation" (Spring semester)

Good textbooks:

Kramer
Herlihy