Concurrent Algorithms (Overview)

Prof R. Guerraoui
Distributed Computing Laboratory
- https://dcl.epfl.ch/site/education/ca_2019 -
Today

- Logistics
- Motivation
- Content
WARNING

- This course is different from the course: **Distributed Algorithms**
  - shared memory vs message passing
  - It does make a lot of sense to take both
This course

- Theoretical but no specific theoretical background is required

- Exercises throughout the semester

- Project (40%) + Exam (60%)

- Support: Algorithms for concurrent systems
New York Times, 8 May 2004: Major chip manufacturers announced what is perceived as a major paradigm shift in computing:

**Multiprocessors vs faster processors**
Intel ... [has] decided to focus its development efforts on «dual core» processors ... with two engines instead of one, allowing for greater efficiency because the processor workload is essentially shared.
Multicores are almost everywhere

- Dual-core commonplace in laptops
- Quad-core in desktops
- Dual quad-core in servers
- All major chip manufacturers produce multicore CPUs
  - SUN Niagara (8 cores, 32 threads)
  - Intel Xeon (4 cores)
  - AMD Opteron (4 cores)
Multicores are almost everywhere

- **Quad-core** in laptops
- **Octa-core** in desktops
- **2*12 cores** in servers
- All major chip manufacturers produce multicore CPUs
  - **Oracle Sparc** (32 cores, 256 threads)
  - **Intel Xeon** (12-16 cores)
  - **AMD Opteron** (12-16 cores)
AMD Opteron (4 cores)

- **L1 cache**
- **L2 cache**
- **L3 cache (shared)**

<table>
<thead>
<tr>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
<th>Core 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Control</td>
<td>Cache Control</td>
<td>Cache Control</td>
<td>Cache Control</td>
</tr>
<tr>
<td>128 KB</td>
<td>128 KB</td>
<td>128 KB</td>
<td>128 KB</td>
</tr>
<tr>
<td>512 KB</td>
<td>512 KB</td>
<td>512 KB</td>
<td>512 KB</td>
</tr>
<tr>
<td>2MB+</td>
<td>Integrated Memory Controller</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory | Memory
SUN’s Niagara CPU2 (8 cores)
Multiprocessors

- Multiple hardware processors: each executes a series of processes (software constructs) modeling sequential programs

- Multicore architecture: multiple processors are placed on the same chip
Principles of an architecture

- Two fundamental components that **fall apart**: processors and memory

- The Interconnect links the processors with the memory:
  - **SMP** (symmetric): bus (a tiny Ethernet)
  - **NUMA** (network): point-to-point network
The basic unit of time is the *cycle*: time to execute an instruction.

This changes with technology but the relative cost of instructions (local vs memory) does not.
Abstract view

Processor + Cache

Bus

Memory
Hardware synchronization objects

The basic unit of communication is the *read* and *write* to the memory (through the cache).

More sophisticated objects are typically provided and, as we will see, necessary: C&S, T&S, LL/SC.
The free ride is over

Cannot rely on CPUs getting faster in every generation

Utilizing more than one CPU core requires concurrency
The free ride is over

One of the biggest software challenges: exploiting concurrency

- Every programmer will have to deal with it
- Concurrent programming is hard to get right
Speed will be achieved by having several processors work on independent parts of a task

*But*

the processors would occasionally need to pause and synchronize
Concurrent processes

Shared object
public class Counter

private int c = 0;

public long getAndIncrement()
{
    return c++;
}

Locking (mutual exclusion)

Locked object
Implicit use of a lock

public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void getAndIncrement() {
        return c++;
    }
    public synchronized int value() {
        return c;
    }
}
Locking with compare&swap()

- A **Compare&Swap** object maintains a value \( x \), init to \( \bot \), and \( y \);

- It provides one operation: \( \text{c&s(old,new)} \);

  ✔ Sequential spec:
  - \( \text{c&s(old,new)} \)
    
    \{y := x; if x = old then x := new; return(y)}
Locking with compare\&swap()

lock() { 
  repeat until unlocked = this.c\&s(unlocked,locked) 
}

unlock() { 
  this.c\&s(locked,unlocked) 
}
Locking with test&set()

- A **Test&Set** object maintains binary values x, init to 0, and y;

- It provides one operation: **t&s()**

  ✓ Sequential spec:
  ✓ t&s() {y := x; x: = 1; return(y);}
Locking with test&set()

lock() {
repeat until (0 = this.t&s());
}

unlock() {
    this.setState(0);
}
Locking with test&set()

```java
lock() {
  while (true) {
    repeat until (0 = this.getState());
    if 0 = (this.t&s()) return(true);
  }
}

unlock() {
  this.setState(0);
}
```
Explicit use of a lock

Lock l = ...;
    l.lock();
    try {
        // access the resource protected by this lock
    } finally {
        l.unlock();
    }
Locking (mutual exclusion)

**Difficult:** 50% of the bugs reported in Java come from the mis-use of « synchronized »

**Slow:** a process holding a lock prevents all others from progressing
Locked object

One process at a time
Processes are asynchronous

- Page faults
- Pre-emption
- Failures
- Cache misses, ...
Processes are asynchronous

- A cache miss can delay a process by ten instructions
- A page fault by few millions
- An OS preemption by hundreds of millions...
Coarse grained locks => slow

Fine grained locks => errors
Double-ended queue

Enqueue | Dequeue
Processes are asynchronous

*Page faults, pre-emption, failures, cache misses, ...*

A process can be delayed by millions of instructions ...
Alternative to locking?
Wait-free atomic objects

*Wait-freedom:* every process that invokes an operation eventually returns from the invocation (robust ... unlike locking)

*Atomicity:* every operation appears to execute instantaneously (as if the object was locked...
In short

This course studies how to

*wait-free* implement high-level

*atomic* objects out of primitive base objects
Concurrent processes

Shared object
Roadmap

Model

Processes and objects

Atomicity and wait-freedom

Examples

Content
Processes

- We assume a *finite* set of processes: $p_1, \ldots, p_N$ or $p, q, r$

- Processes have *unique identities* and know each other

PS. Unless explicitly stated otherwise
Processes

Processes are *sequential* units of computations

Unless explicitly stated otherwise, we make no assumption on process (relative) speeds
Processes

p1

p2

p3
Processes

A process either executes the algorithm assigned to it or **crashes**

A process that crashes does **not recover** (in the context of the considered computation)

A process that does not crash in a given execution (computation or run) is called **correct** (in that execution)
Processes

p1

\[\text{crash}\]

p2

p3
On objects and processes

Processes execute local computation or access shared objects through their operations.

Every operation is expected to return a reply.
Processes

p1

operation

p2

operation

p3

operation
On objects and processes

*Sequentiality* means here that, after invoking an operation op1 on some object O1, a process does not invoke a new operation (on the same or on some other object) until it receives the reply for op1.

PS. Sometimes we talk about operations when we should be talking about operation invocations.
Processes

p1

operation

p2

operation

p3

operation
Atomicity

Every operation appears to execute at some indivisible point in time (called linearization point) between the invocation and reply time events.
Atomicity

operation

p1

operation

p2

operation

p3
Atomicity
Atomicity (the crash case)

- p1
- p2
- p3

.crash
Atomicity (the crash case)

Diagram showing the execution of operations on processes p1, p2, and p3.
Atomicity (the crash case)

- **p1**: Operation
- **p2**: Operation
- **p3**: Operation
Wait-freedom

Any correct process that invokes an operation eventually gets a reply, no matter what happens to the other processes (crash or very slow)
Wait-freedom

\[ \text{operation} \]

p1

p2

p3
Wait-freedom

Wait-freedom conveys the robustness of the implementation.

With a \( n \) wait-free implementation, a process gets replies despite the crash of the \( n-1 \) other processes.

PS. This precludes implementations based on locks (mutual exclusion).
Wait-freedom

operation

p1

p2

p3

operation

p2

p3

operation

p2

p3

crash

crash
Roadmap

- Model
  - Processes and objects
  - Atomicity and wait-freedom
- Examples
- Content
Motivation

Most synchronization primitives (problems) can be precisely expressed as atomic objects (implementations)

Studying how to ensure robust synchronization boils down to studying wait-free atomic object implementations
Example 1

The reader/writer synchronization problem corresponds to the *register* object.

Basically, the processes need to read or write a shared data structure such that the value read by a process at a time $t$, is the last value written before $t$. 

Register

A register has two operations: read() and write()

We assume that a register contains an integer for presentation simplicity, i.e., the value stored in the register is an integer, denoted by $x$ (initially 0)
Sequential specification

- **read()**
  - return($x$)

- **write($v$)**
  - $x <- v$
  - return(ok)
Atomicity?

\[ \text{write}(1) \rightarrow \text{ok} \]

\[ \text{p1} \]

\[ \text{read}() \rightarrow 2 \]

\[ \text{write}(2) \rightarrow \text{ok} \]

\[ \text{p2} \]

\[ \text{p3} \]
Atomicity?

write(1) - ok

p1

read() - 2

p2

write(2) - ok

p3
Atomicity?

write(1) - ok

read() - 1

write(2) - ok
Atomicity?

write(1) - ok

read() - 1

write(2) - ok
Atomicity?

write(1) - ok

read() - 1

read() - 1

p1

p2

p3
Atomicity?

\[ \text{write}(1) - \text{ok} \]

\[ \text{read()} - 1 \]

\[ \text{read()} - 0 \]
Atomicity?

write(1) - ok

p1

p2

read() - 0

p3

read() - 0
Atomicity?

write(1) - ok

read() - 0

read() - 0
Atomicity?

\[ \text{write}(1) - \text{ok} \]

\[ \text{read()} - 0 \]

\[ \text{read()} - 0 \]
Atomicity?

write(1) - ok

read() - 1

read() - 0
Atomicity?

write(1) - ok

read() - 1

p1

p2

p3
Example 2

- The producer/consumer synchronization problem corresponds to the *queue* object.

- *Producer* processes create items that need to be used by *consumer* processes.

- An item *cannot* be consumed by two processes and the first item produced is the first consumed.
Queue

A *queue* has two operations: `enqueue()` and `dequeue()`

We assume that a *queue internally* maintains a list $x$ which exports operation `appends()` to put an item at the end of the list and `remove()` to remove an element from the head of the list.
Sequential specification

**dequeue()**
- if(x=0) then return(nil);
- else return(x.remove())

**enqueue(v)**
- x.append(v);
- return(ok)
Atomicity?

\[ \text{enq}(x) - \text{ok} \]

\[ \text{enq}(y) - \text{ok} \]

\[ \text{deq}() - y \]

\[ \text{deq}() - x \]
Atomicity?

enq(x) - ok

deq() - y

deq() - x
Atomicity?

- enq(x) - ok
- deq() - y
- enq(y) - ok
Atomicity?

$p1$

$\text{enq}(x) - \text{ok}$

$p2$

$\text{deq}() - x$

$p3$

$\text{enq}(y) - \text{ok}$
Roadmap

Model

Processes and objects

Atomicity and wait-freedom

Examples

Content
Content

(1) Implementing *registers*
(2) The power & limitation of *registers*
(3) *Universal* objects & synchronization number
(4) The power of *time* & failure detection
(5) Tolerating *failure* prone objects
(6) *Anonymous* implementations
(7) *Non-volatile* memory
(8) *Hybrid* memory
(9) *Transactional* memory
In short

This course studies how to wait-free implement high-level atomic objects out of basic objects

Remark. Unless explicitly stated otherwise, objects mean atomic objects and implementations are wait-free