Concurrent Computing

Introduction

SE205, P1, 2017

Administrivia

- Language: (fr)anglais?
- Lectures: Fridays (15.09-03.11), 13:30-16:45, Amphi Grenat
- Web page: https://se205.wp.imt.fr/

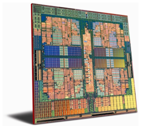
- Exam: 03.11, 15:15-16:45

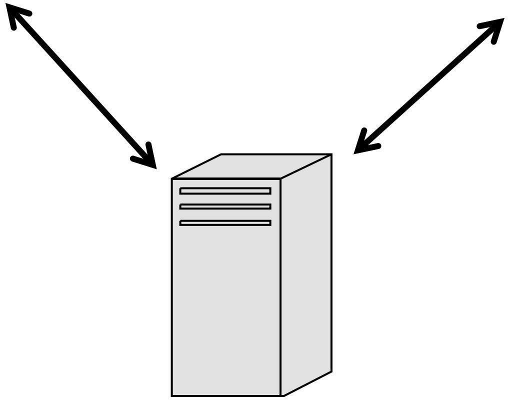
- Office hours (Petr Kuznetsov)
  - ✓ C213-2, appointments by email to petr.kuznetsov@telecom-paristech.fr

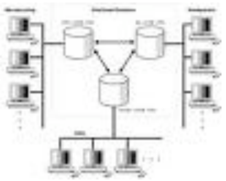
Literature (for my part)

- Lecture notes: Concurrent computing. R. Guerraoui, P. Kuznetsov (https://www.dropbox.com/s/oiu6wp7oesngh8c/book-ln.pdf?dl=0)
- M. Herlihy and N. Shavit. The art of multiprocessor programming. Morgan Kaufman, 2008 (library)

Concurrency is everywhere!







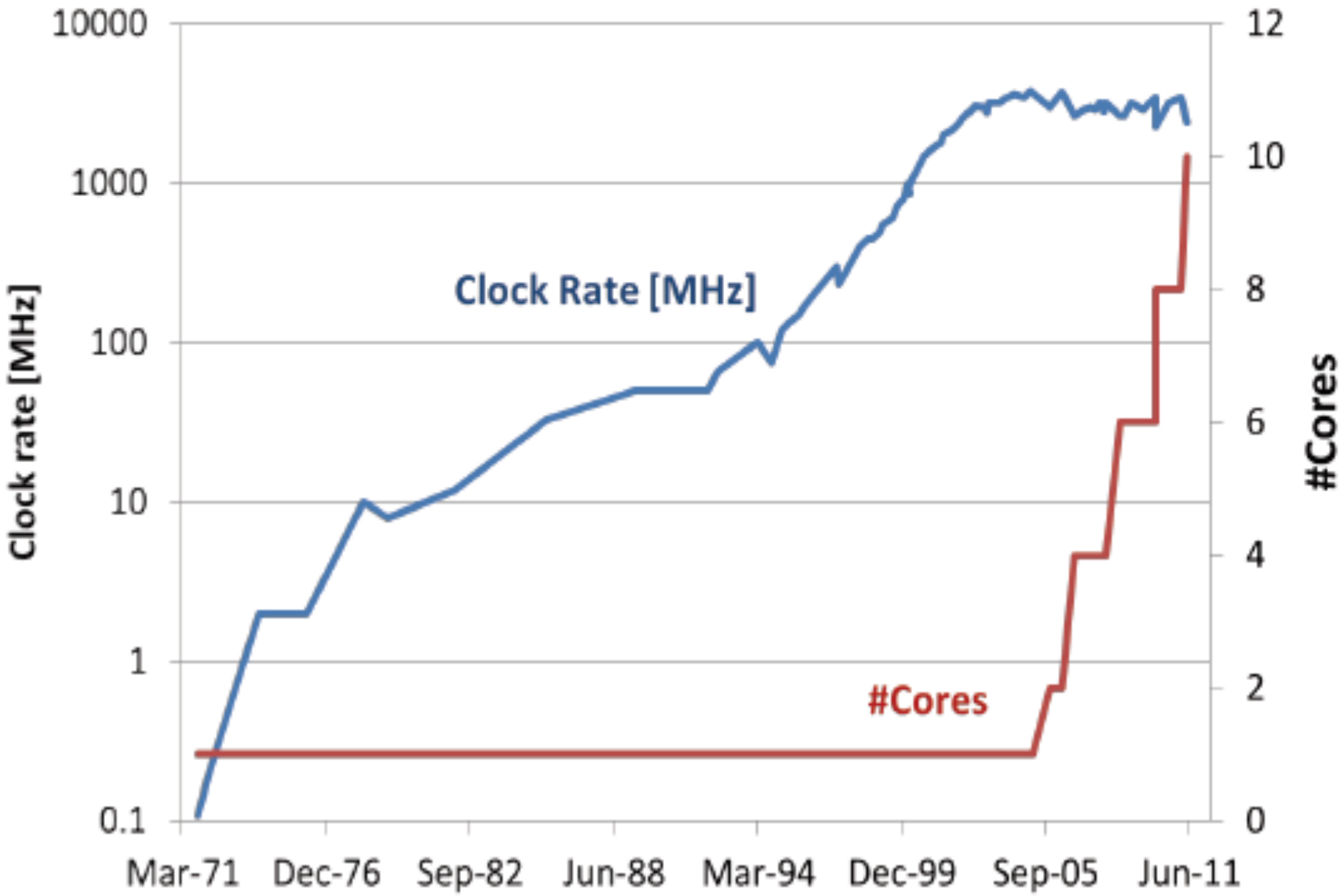
- Multi-core processors
- Sensor networks
- Internet
- Basically everything related computing

Communication models

- Shared memory
  - ✓Processes apply (read–write) operations on shared variables
  - ✓Failures and asynchrony

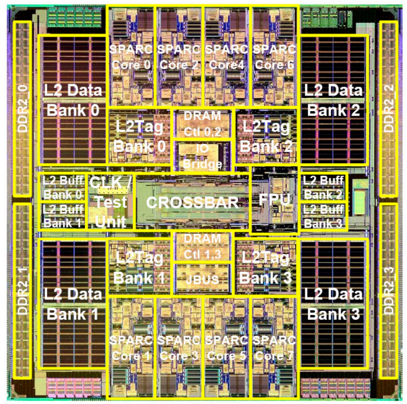
- Message passing
  - ✓Processes send and receive messages
  - ✓Communication graphs
  - ✓Message delays

The concurrency challenge



The case against the "washing machine science"

- Single-processor performance does not improve
- But we can add more cores
- Run concurrent code on multiple processors

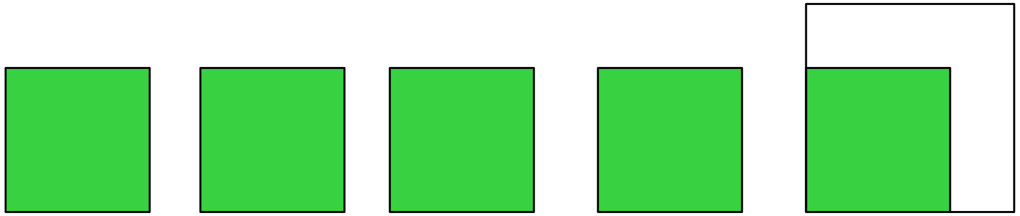


Can we expect a proportional speedup? (ratio between sequential time and parallel time for executing a job)

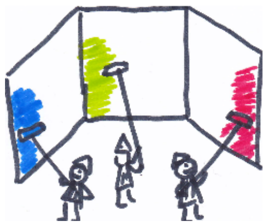
TELECOM ParisTech

Example: painting in parallel

- 5 friends want to paint 5 equal-size rooms, one friend per room
  - ✓Speedup = 5



- What if one room is twice as big?



Amdahl's Law

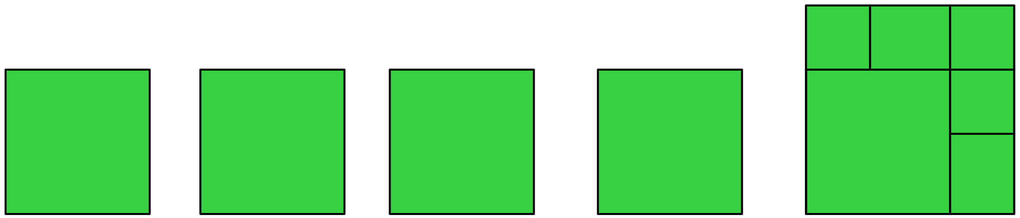
- p – fraction of the work that can be done in parallel (no synchronization)
- n - the number of processors
- Time one processor needs to complete the job = 1

$$S = \frac{1}{1 - p + p/n}$$

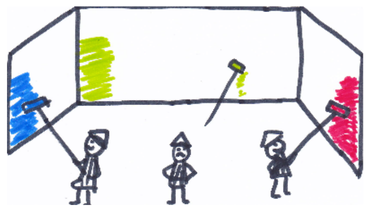
TELECOM ParisTech

A better solution

- When done, help the others
  - ✓All 5 paint the remaining half-room in parallel
- Communication and agreement is required!
- **This is a hard task**



- And this is what synchronization algorithms try to achieve!



Challenges

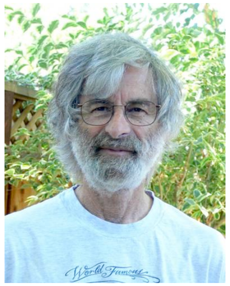
- What is a correct implementation?
  - ✓Safety and liveness
- What is the cost of synchronization?
  - ✓Time and space lower bounds
- Failures/asynchrony
  - ✓Fault-tolerant concurrency?
- How to distinguish possible from impossible?
  - ✓Impossibility results

TELECOM ParisTech

Distributed ≠ Parallel

- The main challenge is synchronization

- “you know you have a distributed system when the crash of a computer you’ve never heard of stops you from getting any work done” (Lamport)



TELECOM ParisTech

History

- Dining philosophers, mutual exclusion (Dijkstra)~60’s
- Distributed computing, logical clocks (Lamport), distributed transactions (Gray) ~70’s
- Consensus (Lynch) ~80’s
- Distributed programming models, since ~90’s
- Multicores/manycores now

TELECOM ParisTech

Why synchronize ?

- **Race condition**: results depend on the scheduling

- **Synchronization:** resolving the races

- A race-prone portion of code **critical section**
  - ✓Must be executed sequentially

- **Synchronization problems**: mutual exclusion, readers-writers, producer-consumer, …

Edsger Dijkstra
1930-2002

Dining philosophers (Dijkstra, 1965)

- To make progress (to eat) each process (philosopher) needs two resources (forks)
- Mutual exclusion: no fork can be shared
- Progress conditions:
  - ✓Some philosopher does not starve (deadlock-freedom)
  - ✓No philosopher starves (starvation-freedom)

Mutual exclusion

- No two processes are in their critical sections (CS) at the same time

+

- Deadlock-freedom: **at least one** process eventually enters its CS
- Starvation-freedom: **every** process eventually enters its CS
  - ✓ Assuming no process **blocks** in **CS** or **Entry section**

- Originally: implemented by reading and writing
  - ✓ Peterson's lock, Lamport's bakery algorithm
- Currently: in hardware (mutex, semaphores)

Peterson’s lock: 2 processes

```
bool flag[0]   = false;
bool flag[1]   = false;
int turn;
```

P0:

```
flag[0] = true;
turn = 1;
while (flag[1] and turn==1)
{
          // busy wait
}
// critical section
…
// end of critical section
flag[0] = false;
```

P1:

```
flag[1] = true;
turn = 0;
while (flag[0] and turn==0)
{
          // busy wait
}
// critical section
…
// end of critical section
flag[1] = false;
```

Peterson's lock: N ≥ 2 processes

```
// initialization
level[0..N-1] = {-1};     // current level of processes 0...N-1
waiting[0..N-2] = {-1}; // the waiting process in each level
                            // 0...N-2

// code for process i that wishes to enter CS
for (m = 0; m < N-1; m++) {
    level[i] = m;
    waiting[m] = i;
    while(waiting[m] == i &&(exists k ≠ i: level[k] ≥ m)) {
        // busy wait
    }
}
// critical section
level[i] = -1; // exit section
```

Bakery [Lamport'74,simplified]

```
// initialization
flag: array [1..N] of bool = {false};
label: array [1..N] of integer = {0}; //assume no bound

// code for process i that wishes to enter CS

flag[i] = true; //enter the “doorway”
label[i] = 1 + max(label[1], ..., lebel[N]); //pick a ticket
while (for some k ≠ i: flag[k] and (label[k],k)<<(label[i],i));
// wait until all processes “ahead” are served
…
// critical section
…
flag[i] = false; // exit section
```

Processes are served in the “ticket order”: first-come-first-serve

Readers-writers problem

- Writer updates a file
- Reader keeps itself up-to-date
- Reads and writes are non-atomic!

Why synchronization? Inconsistent values might be read

Writer	Reader
`T=0: write("sell the cat")`	
	`T=1: read("sell …")`
`T=2: write("wash the dog")`	
	`T=3: read("… the dog")`

`Sell the dog?`

Producer-consumer (bounded buffer) problem

- Producers **put** items in the buffer (of bounded size)
- Consumers **get** items from the buffer
- Every item is consumed, no item is consumed twice

  (Client-server, multi-threaded web servers, pipes, …)

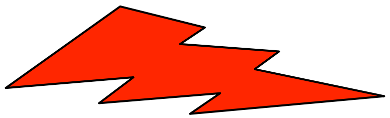
Why synchronization? Items can get lost or consumed twice:

Producer

```
/* produce item */
while (counter==MAX);
buffer[in] = item;
in = (in+1) % MAX;
counter++;
```

Consumer

```
/* to consume item */
while (counter==0);
item=buffer[out];
out=(out+1) % MAX;
counter--;
/* consume item */
```



Race!

Synchronization tools

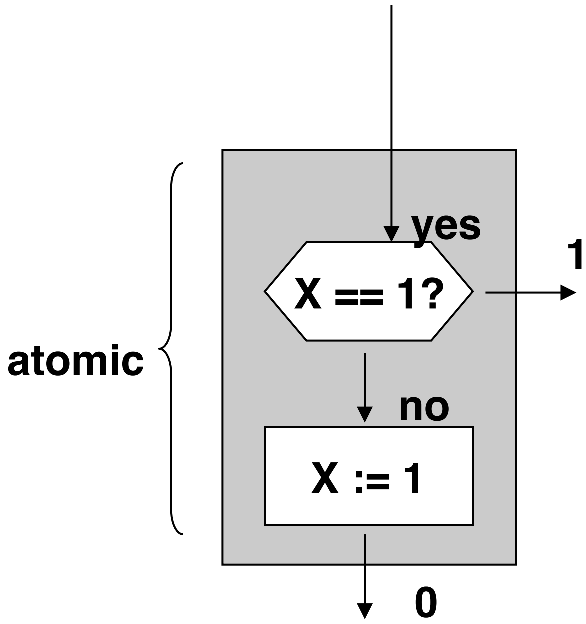
- Busy-waiting (TAS)
- Semaphores (locks), monitors
- Nonblocking synchronization
- Transactional memory

Busy-wait: Test and Set

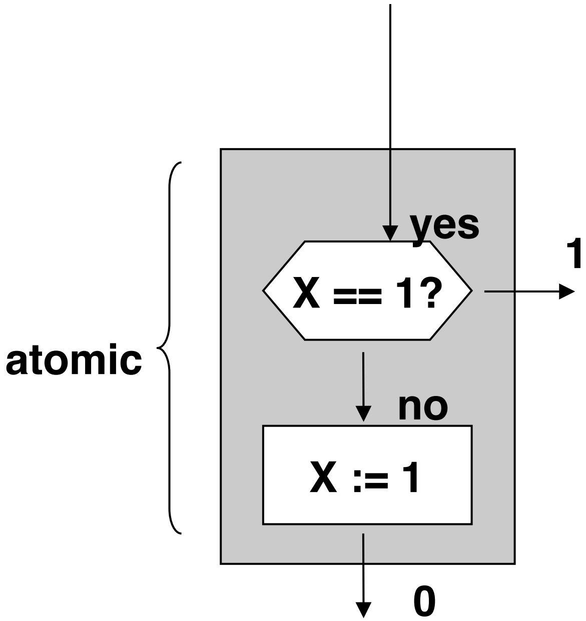
- TAS(X) **tests** if X = 1, **sets** X to 1 if not, and returns the old value of X
  - ✓ Instruction available on almost all processors

```
TAS(X):

atomic {
    if X == 1 return 1;
    X = 1;
    return 0;
}
```



Busy-wait: Test and Set



shared X:=0

Producer	Consumer
`while(counter==MAX);`	`while (counter==0);`
`. . .`	`. . .`
`buffer[in] = item;`	`item = buffer[out];`
`. . .`	`. . .`
`while TAS(X);`	`while TAS(X);`
`counter++;`	`counter--;`
`X:=0;`	`X:=0;`
`. . .`	`...`

Problems:

- busy waiting
- no record of request order (for multiple producers and consumers)

Semaphores [Dijkstra 1968]: specification

- A semaphore S is an integer variable accessed (apart from initialization) with two atomic operations P(S) and V(S)
  - ✓ Stands for “passeren” (to pass) and “vrijgeven” (to release) in Dutch

- The value of S indicates the number of resource elements available (if positive), or the number of processes waiting to acquire a resource element (if negative)

```
Init(S,v){ S := v; }

P(S){
        while S<=0;     /* wait until a resource is available */
        S--;            /* pass to a resource */
}

V(S){
        S++;            /* release a resource */
}
```

Semaphores: implementation

S is associated with a composite object:

- ✓ S.counter: the **value** of the semaphore
- ✓ S.wq: the **waiting queue,** memorizing the processes having requested a resource element

```
Init(S,R_nb) {
    S.counter=R_nb;
    S.wq=empty;
}
P(S) {
    S.counter--;
    if S.counter<0{
     put the process in S.wq and wait until
    READY;}
}
V(S) {
    S.counter++
    if S.counter>=0{
            mark 1st process in S.wq as
            READY;}
}
```

Lock

- A semaphore initialized to 1, is called a **lock** (or **mutex)**
- When a process is in a critical section, no other process can come in

shared semaphore S := 1

Producer	Consumer
`while (counter==MAX);`	`while (counter==0);`
`. . .`	`. . .`
`buffer[in] = item;`	`item = buffer[out];`
`. . .`	`. . .`
`P(S);`	`P(S);`
`counter++;`	`counter--;`
`V(S)`	`V(S);`
`. . .`	`...`

Problem: still waiting until the buffer is ready

Semaphores for producer-consumer

- 2 semaphores used :
  - ✓ **empty:** indicates empty slots in the buffer (to be used by the producer)
  - ✓ **full:** indicates full slots in the buffer (to be read by the consumer)

```
shared  semaphores empty := MAX, full := 0;
```

Producer	Consumer

`P(empty)`
`buffer[in] = item;`
`in = (in+1) % MAX;`
`V(full)`
	`P(full);`
	`item = buffer[out];`
	`out=(out+1) % MAX;`
	`V(empty);`

Potential problems with semaphores/locks

- **Blocking**: progress of a process is conditional (depends on other processes)
- **Deadlock:** no progress ever made

```
X1:=1; X2:=1
```

Process 1	Process 2
...	...
P(X1)	P(X2)
P(X2)	P(X1)
critical section	critical section
V(X2)	V(X1)
V(X1)	V(X2)
...	...

- **Starvation**: waiting in the waiting queue forever

TELECOM ParisTech

Other problems of blocking synchronization

- Priority inversion
  - ✓High-priority threads blocked
- No robustness
  - ✓Page faults, cache misses etc.
- Not composable

Can we think of anything else?

Non-blocking algorithms

A process makes progress, regardless of the other processes

shared buffer[MAX]:=empty; head:=0; tail:=0;

Producer `put(item)`	Consumer `get()`

`if (tail-head == MAX){`
`        return(full);`
`}`
`buffer[tail%MAX]=item;`
`tail++;`
`return(ok);`
	`if (tail-head == 0){`
	`        return(empty);`
	`}`
	`item=buffer[head%MAX];`
	`head++;`
	`return(item);`

Problems:

- works for 2 processes but hard to say why it works ☺
- multiple producers/consumers? Other synchronization pbs?

(stay in class to learn more)

Transactional memory

- Mark sequences of instructions as an **atomic transaction**, e.g., the resulting producer code:

```
atomic {
        if (tail-head == MAX){
        return full;
        }
        items[tail%MAX]=item;
        tail++;
}
return ok;
```

- A transaction can be either **committed** or **aborted**
  - ✓ Committed transactions are **serializable**
  - ✓ Let the transactional memory (TM) care about the conflicts
  - ✓ Easy to program, but performance may be problematic

TELECOM ParisTech

Summary

- Concurrency is indispensable in programming:
  - ✓ Every system is now concurrent
  - ✓ Every parallel program needs to synchronize
  - ✓ Synchronization cost is high (“Amdahl’s Law”)

- Tools:
  - ✓ Synchronization primitives (e.g., monitors, TAS, CAS, LL/SC)
  - ✓ Synchronization libraries (e.g., java.util.concurrent)
  - ✓ Transactional memory, also in hardware (Intel Haswell, IBM Blue Gene,…)

- Coming later:
  - ✓ Read-write transformations and snapshot memory
  - ✓ Nonblocking synchronization

Quiz

- What if we reverse the order of the first two lines the 2-process Peterson's algorithm

```
P0:                          P1:
turn = 1;                    turn = 0;
flag[0] = true;              flag[1] = true;
…                            …
```

Would it work?

- Prove that Peterson's N-process algorithm ensures:
  - ✓ mutual exclusion: no two processes are in the critical section at a time
  - ✓ starvation freedom: every process in the trying section eventually reaches the critical section (assuming no process fails in the trying, critical, or exit sections)

Bakery [Lamport'74,original]

```
// initialization
flag: array [1..N] of bool = {false};
label: array [1..N] of integer = {0}; //assume no bound

// code for process i that wishes to enter CS
flag[i] = true; //enter the doorway
label[i] = 1 + max(label[1], ..., label[N]); //pick a ticket
flag[i] = false; //exit the doorway
for j=1 to N do {
          while (flag[j]); //wait until j is not in the doorway
          while (label[j]≠0 and (label[j],j)<<(label[i],i));
          // wait until j is not "ahead"
}
…
// critical section
…
label[i] = 0; // exit section
```

Ticket withdrawal is "protected" with flags: a very useful trick