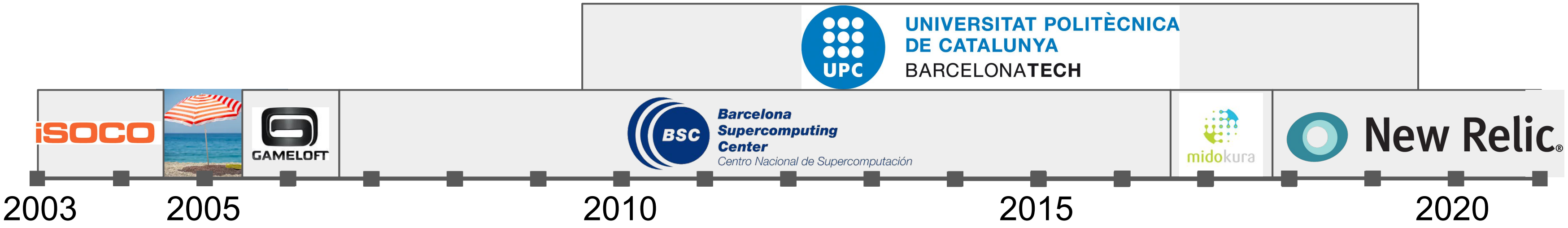
The Go concurrency model

A tour on simplicity

Mario Macías, Ph.D.
New Relic Inc
Assembler School
February 2020

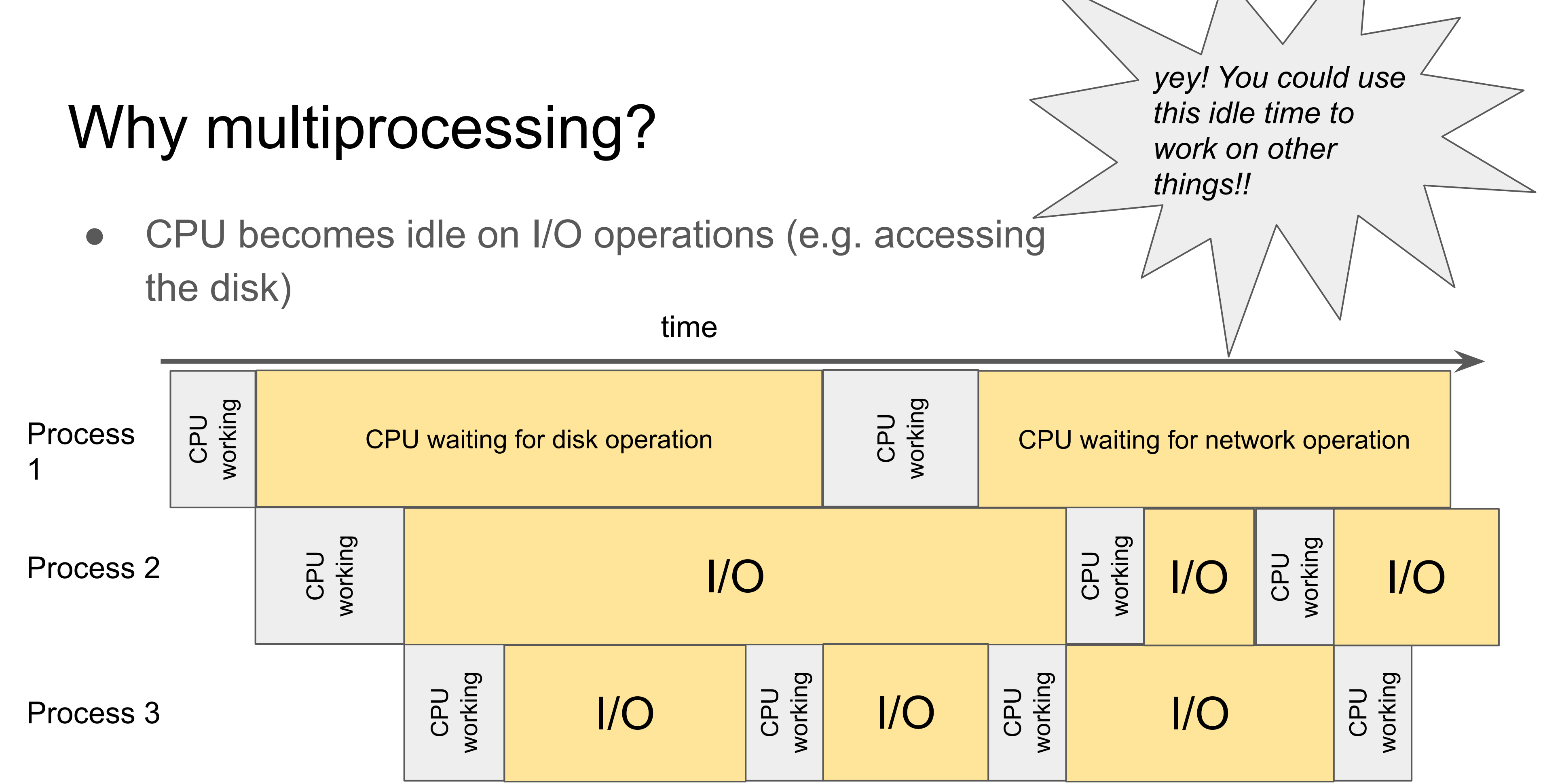
mmacias@newrelic.com
@MaciasUPC
http://macias.info

Who am I?


UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH
UPC
iSOCO
GAMELOFT
BSC
Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación
midokura
New Relic
2003
2005
2010
2015
2020

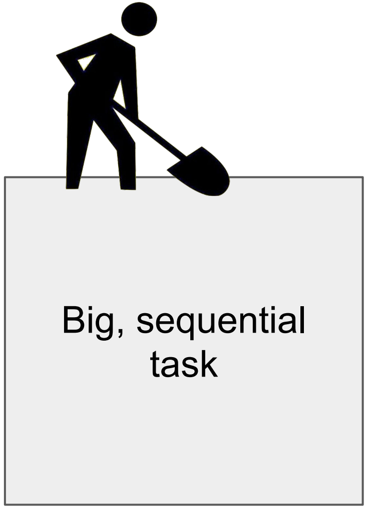
Why multiprocessing?

- CPU becomes idle on I/O operations (e.g. accessing the disk)

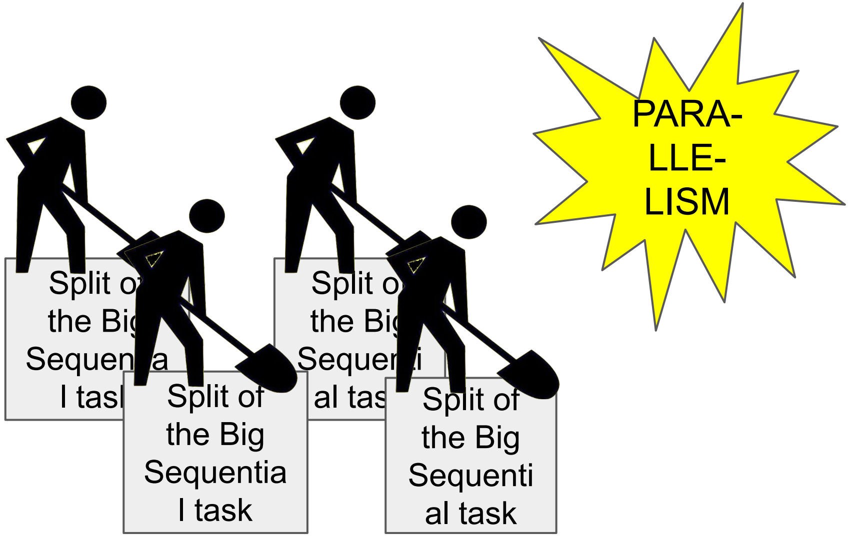


Why multiprocessing?

- Battle for GigaHertz ended
- Programs need to squeeze the power of multi-core CPUs



single-core times



Multicore approach

Terminology

- Processes
  - A program in execution: OS can spawn multiple processes in parallel
  - It has its own isolated space in the memory
  - Requires heavyweight management from the OS
- Threads
  - A process can spawn multiple threads
  - They share the process memory, but the stack
  - Lighter management than Processes
- Parallelism
  - Dividing a usually big task in multiple subtasks (e.g. threads or processes) that are executed at the same time (e.g. different cores)
- Concurrency
  - The problem of effectively sharing resources and data between parallel tasks
  - The problem of effectively coordinating parallel tasks

Threads limitations

- Despite they are lighter than processes, they are still heavyweight
  - *time for demo!*
-

Lightweight concurrency: async, callback-based

- Require a "runtime" that schedules the tasks. It can be a library.
- Requires stop coding as most languages are designed for

Call-back Hell!!

```
var async = require("async");

User.find(userId, function(err, user){
  if (err) return errorHandler(err);
  User.all({where: {id: {$in: user.friends}}}, function(err, friends) {
    if (err) return errorHandler(err);
    async.each(friends, function(friend, done){
      friend.posts = [];
      Post.all({where: {userId: {$in: friend.id}}}, function(err, posts) {
        if (err) return errorHandler(err);
        async.each(posts, function(post, donePosts){
          friend.push(post);
          Comments.all({where: post.id}, function(err, comments) {
            if (err) donePosts(err);
            post.comments = comments;
            donePosts();
          });
        }, function(err) {
          if (err) return errorHandler(err);
          done();
        });
      });
    }, function(err) {
      if (err) return errorHandler(err);
      render(user, friends);
    });
  }
});
```

Lightweight concurrency: coroutines

- The programmer needs to state when the execution can go to another part of the code
- The **runtime** needs to be part of the language implementation

```
var test = coroutine(function*() {
    console.log('Hello!');
    var x = yield;
    console.log('First I got: ' + x);
    var y = yield;
    console.log('Then I got: ' + y);
});
// prints 'Hello!'
test('a dog'); // prints 'First I got: a dog'
test('a cat'); // prints 'Then I got: a cat'
```

Goroutines

- Simplifies even more the approach of coroutines
- The programmer doesn't need to put "yields" in the code → Cleaner code
- The Go API and implementation is full of "hidden" yields that gives the control to the Go runtime where it's more efficient to do it
- Drawback: you pay the overhead in performance even if you don't use them

Threads vs Fibers

- Threads and processes: preemptive multitasking
  - The "steals" the CPU to the processes/threads when they perform given operations, or after a given time.
  - If a process/thread hogs the CPU, the OS steals it and schedules another process.
  - It has a big cost: the OS needs to save the complete CPU status for the paused process and restore the status of the new allocated process.
- Fibers (a.k.a. lightweight threads): cooperative multitasking
  - An async callback function or a coroutine are types of fibers
  - The "runtime" allocates a coroutine/function to be executed in one of the threads that run the runtime
  - The coroutines return voluntarily the execution → they only need to save the important data for the later use
  - If a coroutine hogs the CPU and "doesn't cooperate", the other coroutines aren't executed

Launching a Goroutine

```
go nameOfTheFunction(arguments)
```

or

```
go func() {
    /* your instructions here */
}()
```

Need of syncing goroutines

- The above goroutines won't usually complete (demo time)

```
for i := 0; i < 10 ; i++ {
	numTask := i
	go func() {
		fmt.Printf("Running parallel task %d\n", numTask)
	}()
}
fmt.Println("All the parallel tasks have ended. Exiting now")
```

WaitGroup

- WaitGroup allows blocking a goroutine until a group of other goroutines finish
- wg := sync.WaitGroup{}
  wg.Add(<num>)
  - Instantiates a WaitGroup
  - Sets the num of parallel goroutines we will need to wait for
- wg.Done()
  - Invoked from each goroutine
  - Decreases the waitgroup counter, meaning that this goroutine has finished
- wg.Wait()
  - Usually invoked from the instantiator of the WaitGroup
  - The execution is blocked until all the <num> goroutines invoke wg.Done()

Mutual Exclusion

- Demo time

Mutual Exclusion

- Sometimes we need to make sure that a given segment of the code is accessed only by a goroutine at the same time
- m := sync.Mutex{}
  - Instantiates a mutual exclusion
- m.Lock()
  - Gets the exclusive permission to access the code after this invocation
  - If another goroutine already has this permission, it waits
- m.Unlock()
  - Releases the permission of the Mutex so other goroutines can get it
- Alternative implementation
  - sync.RWMutex{}
    - Multiple Readers can read at the same time.
    - One single Writer locks all the accesses

Other "classic" tools

- Package "atomic"
  - Allows some atomic operations on basic types (Read and Set)
- sync.Once{}
  - Makes sure that a function is only executed once, even if invoked from multiple goroutines
- sync.Pool{}
  - Allows reusing a set of items that may be expensive to instantiate
  - E.g. Database Connections

Channels

- The most used synchronization tool from Go
- It allows sending and receiving values from different goroutines
- ch := make(chan string)
  - Creates a channel that shares strings
- ch <- "hello"
  - Sends a string to a channel
- str := <-ch
  - Receives a string via a channel
- close(ch)
  - Closes the channel

Unbuffered channel

- The receiver of a channel always block until there is something in the channel
- The sender of a channel is also blocked until the item is received
- Warn!
  - Potential deadlocks

Buffered channel

```
ch := make(chan string, 100)      // buffer length: 100 strings
```

- The receiver via a channel will block until something can be received
    - When the element is received, is removed from the buffer
- The sender:
    - If the buffer is not full, it will send the value and continue its execution
    - If the buffer is full, it will block until another element is removed

Close-based synchronization

- When a channel is closed, the receiver won't block anymore and will continue its execution, without having received anything

Multiple receivers

- Multiple goroutines can send/receive at the same time
- There is no guarantee of a fair use

Channel iteration

- A channel can be used in a "range" clause inside a loop
- It is a comfortable way to say "keep receiving values until it is closed"

```
// sweets is a channel

for sweet := range sweets {
	fmt.Println("Received a delicious", sweet)
}
```

Channel select

```
select {
case v := <- foo:
	fmt.Println("received from foo: ", v)
case v := <- bar:
	fmt.Println("from bar:", v)
}
```

- The following code will block until one of the channels has data
- I will receive data from only one channel

channel timeouts

- time.After(< duration >) returns a channel that sends a value after the duration
- With the previous "select" clause, it can implement a channel timeout

```
select {
case v := <- foo:
	fmt.Println("received from foo: ", v)
case <- time.After(5 * time.Second):
	fmt.Println("timeout while waiting for a foo")
}
```

The Go concurrency model

Thank you for your attention!

Mario Macías, Ph.D.
New Relic Inc
Assembler School
February 2020

mmacias@newrelic.com
@MaciasUPC
http://macias.info