

# *Patterns de concurrence et performance*

Arnaud Bétrémieux  
[arb@octo.com](mailto:arb@octo.com)  
Mars 2017



*Cloud Ready Applications*

© OCTO 2015

There is a better way

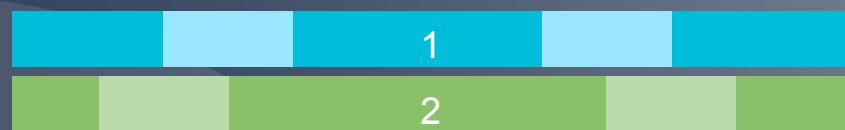
**OCTO**  
Technology

# « Concurrence » ?

Séquentiel



Concurrence parallèle



Concurrence par commutation

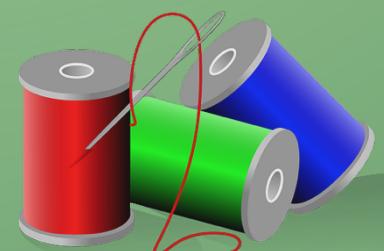


There is a better way

# Threads & processus

*Threads are the new black*

- Le **pattern de référence** pendant des années
  - Souvent **quasi-transparent** (librairies, *runtime*)
- **Abstractions** de concurrence **natives** de l'OS
  - Le **scheduler** répartit le temps CPU, gère les **priorités**
- Pourquoi rechercher autre chose ?
  - Difficulté de **synchronisation** d'accès à un **état partagé**
  - **Maîtrise fine** de la concurrence (ex : jeux vidéo)
  - **Coût du changement de contexte CPU/OS**  
→ **non bloquant** : boucles d'évennement, **callbacks**, **promesses**, **coroutines**...



There is a better way

# Limiter les changements de contexte

- **Bloquant :**

```
var contenu = fs.readFileSync('/etc/passwd',  
                           'utf8');  
console.log(contenu);
```



- **Non bloquant → callback / CPS :**

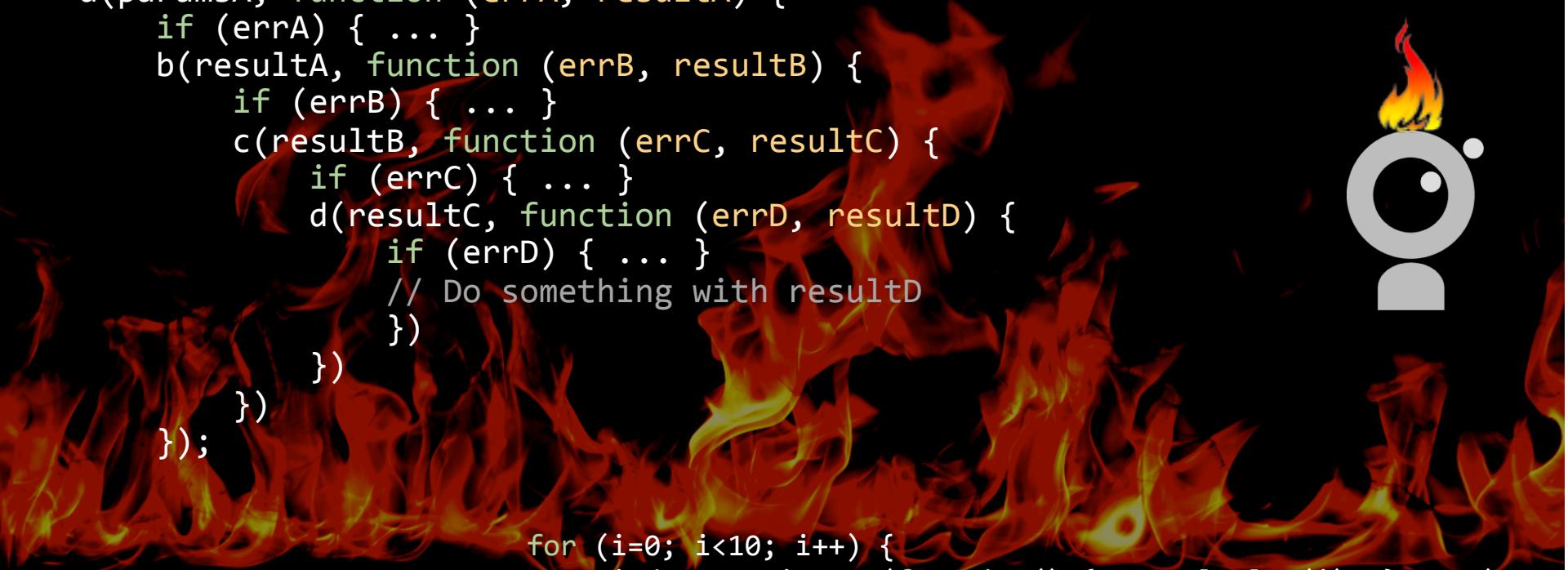
```
fs.readFile('/etc/passwd',  
          'utf8',  
          function(err, contenu) {  
            console.log(contenu);  
          });
```



# Callback hell

« Yo, Dawg, I heard you liked callbacks... »

```
a(paramsA, function (errA, resultA) {  
    if (errA) { ... }  
    b(resultA, function (errB, resultB) {  
        if (errB) { ... }  
        c(resultB, function (errC, resultC) {  
            if (errC) { ... }  
            d(resultC, function (errD, resultD) {  
                if (errD) { ... }  
                // Do something with resultD  
            })  
        })  
    })  
});  
  
for (i=0; i<10; i++) {  
    window.setTimeout(function() { console.log(i); }, 100);  
}  
→ 9, 9, 9, 9...
```



# Promesses

```
getMailFor("arb@octo.com",
    function (err, results) {
        // gérer l'erreur
        // ou les résultats
   });
```



```
promise = getMailFor("arb@octo.com");
promise.then(function (results) {
    // gérer les résultats
})
.otherwise(function (err) {
    // gérer l'erreur
})
.then( // la suite
);
```



There is a better way

# Semi-coroutines / « coroutines »

```
async function getMailFor(address) {  
    var mails;  
    try {  
        mails = await downloadMail(url);  
    } catch(e) {  
        console.error("Oops, could not get your email");  
    }  
    return mails  
}
```



There is a better way

# Coroutines, *green threads*, *fibers*

- « `await` » (`yield`) permis dans toutes les fonctions appelées
  - vs. : semi-coroutines : « `await` » uniquement dans le corps de la fonction `async`
    - **suspension implicite**  
code moins facile à lire
- Analogue à des *threads* avec ***scheduling* coopératif** :
  - *Green threads*
  - *Fibers*



# M:N threading = *threading « hybride »*

« Goroutines » de Go, processes d'Erlang...



*Scheduler  
“runtime”*



*Scheduler  
OS*

There is a better way

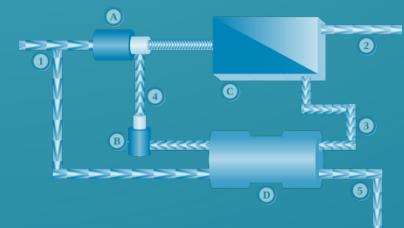


# Flux de données : push vs. pull

**Non blocage** → inversion du **contrôle** du  
**flux de données**

Pour éviter la saturation il faut limiter :

- Les **données** par chaîne de traitement  
→ **streams** avec « **backpressure** »
- Le **nombre d'unités** de concurrence  
(callbacks, acteurs...)  
→ **files d'attente, throttling**



There is a better way

# Détail d'un changement de contexte

- Suspension de la tâche (*thread* ou processus) en cours
- Sauvegarde de l'état de la tâche (registres CPU, gestion de mémoire...)
- Choix de la tâche vers laquelle basculer
  - exécution de l'algorithme de *scheduling*
- Restauration de l'état de la tâche cible
- Redémarrage de la tâche cible
- Coût total réputé important

# Mesures de Benoit Sigoure (2010)

- Linux 3.4.24, Intel E5-2620:  
coût total changement de *thread* ~1300ns [1]
- **x2** si on change de CPU
  - Augmente à mesure qu'on alloue de la mémoire
- Chiffres cohérents avec ceux de Paul Turner (Google) [2]

[1] <http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html>

[2] <https://www.youtube.com/watch?v=KXuZi9aeGTw>

# Principe : 2 processus ou threads s'échangeant le CPU en boucle

```
for (int i = 0; i < iterations; i++) {
    *futex = 0xA;
    while (!syscall(SYS_futex, futex, FUTEX_WAKE, 1, NULL, NULL, 42)) {
        // retry
        sched_yield();
    }
    sched_yield();
    while (syscall(SYS_futex, futex, FUTEX_WAIT, 0xB, NULL, NULL, 42)) {
        // retry
        sched_yield();
    }
}
```

$$\text{Temps changement de contexte moyen} = \frac{\text{temps total d'exécution}}{\text{nb changements de contexte}}$$

# Notre approche (2016)

- Ré-utilisation du code de Benoit Sigoure
- Utilisation des informations de débogage du noyau Linux
  - /proc/\$PID/sched
  - /proc/sched\_debug
  - calculs plus précis,  
plus sûrs

# Exemple de sortie de notre programme de mesure

\* Context switches par CPU :

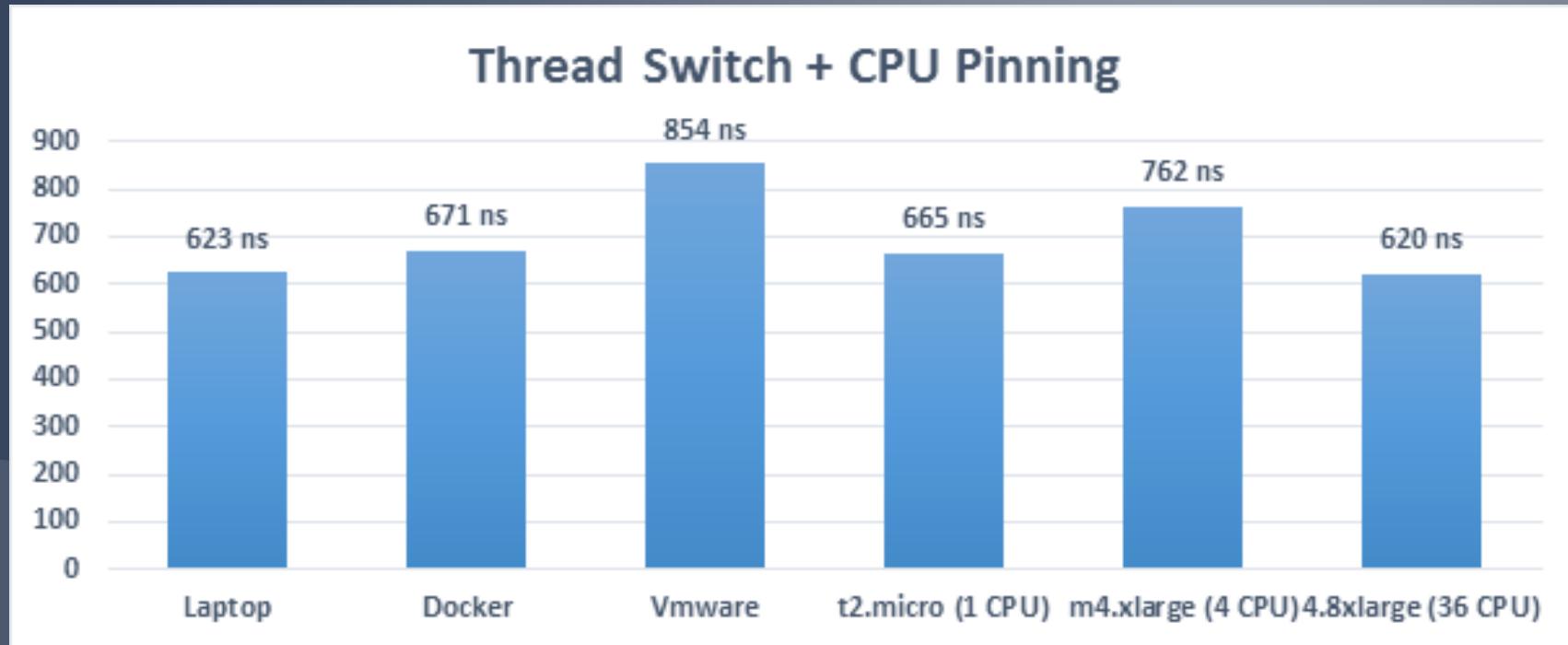
```
CPU #1 : 284909  
CPU #2 : 625395  
CPU #3 : 630746  
CPU #4 : 470013
```

\*\* Données pour le calcul. Vérifier que les pourcentages de temps d'attente et de context switch involontaires sont négligeables:

```
Le temps total du script est de : 3277260282 ns  
Le temps total d'attente est de : 30.219989 ns  
Le pourcentage de temps d'attente est de : 0.000000922%  
Le nombre total de contexts switches est de : 1000584  
Le nombre de contexts switches volontaires total est de : 1000124  
Le nombre de contexts switches involontaires total est de : 460  
Le pourcentage de contexts switches volontaires est de : 99.9540%  
Le pourcentage de contexts switches involontaires est de : 0.0459%
```

\*\* Temps moyen d'un context switch : 3275.347448869 ns

# Sur un même CPU : ~800ns



10 000 changements de contexte par seconde et par CPU

=

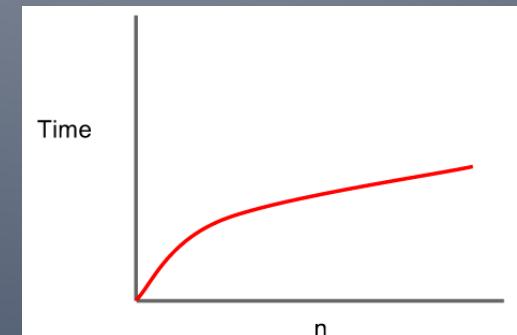
0.8% du temps CPU

# Part de l'algorithme de *scheduling* : ~ 50%

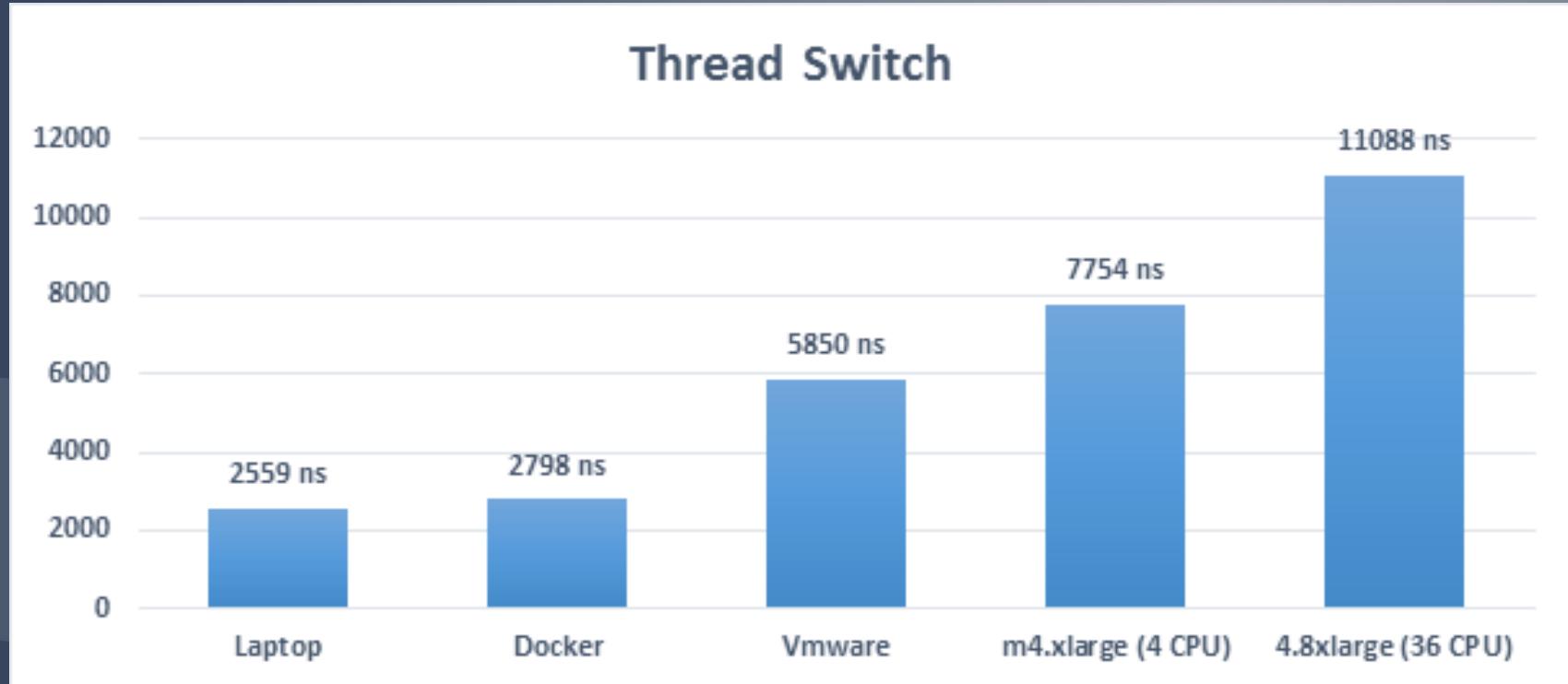
Utilisation du mode SCHED\_FIFO  
pour “contourner” le *scheduler* → ~ 400ns

Pas de lien significatif entre : nombre de threads  
temps changement de contexte

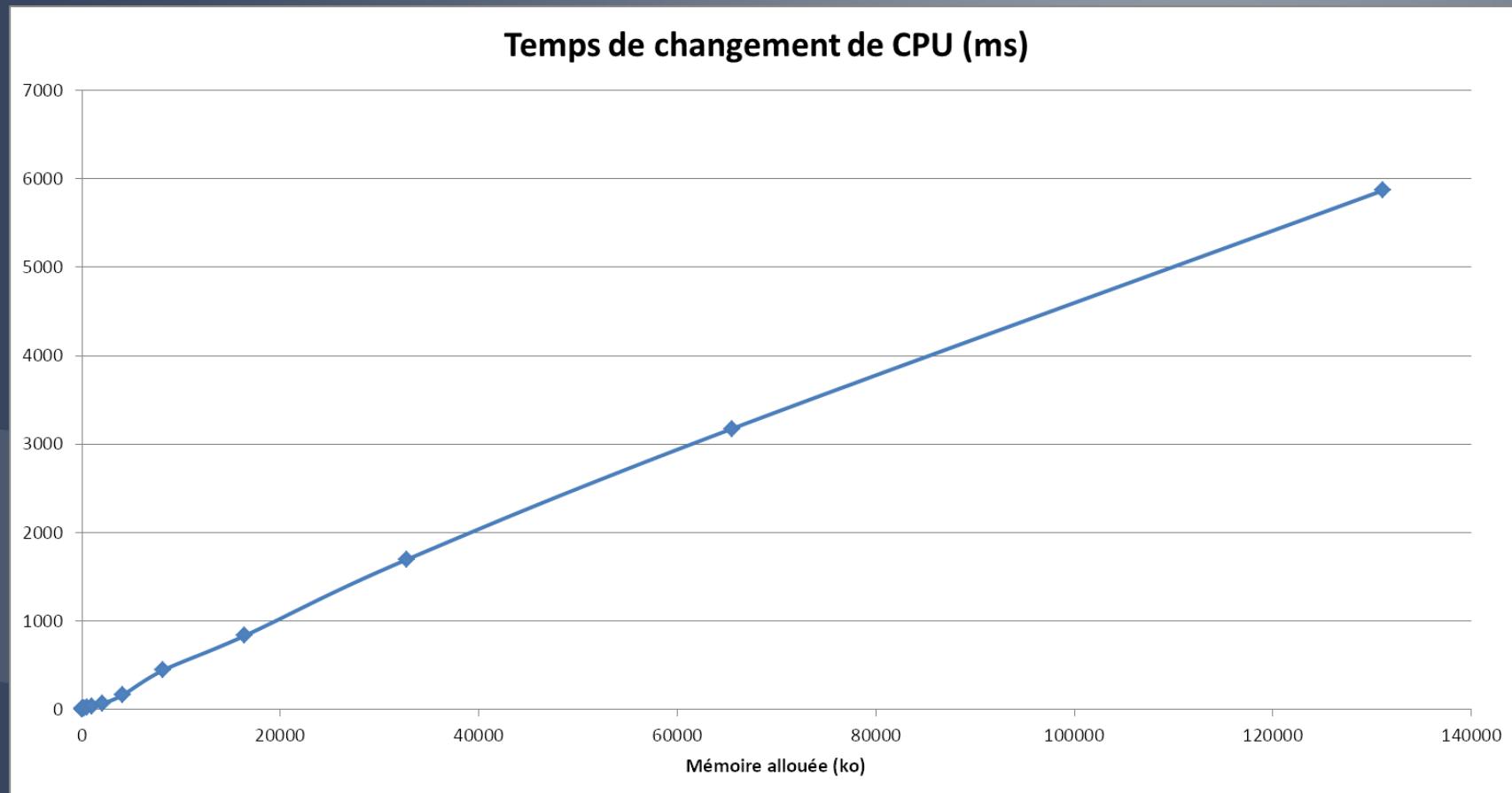
- Algorithme de *scheduling*  $O(\log(n))$
- Proche de l'asymptote très tôt



# Avec passage d'un CPU à un autre : x3 - x14 !



# Conséquences des allocations de mémoire



# Tirer le meilleur parti des *threads/processus*

Limiter les allocations de mémoire par *thread/processus*,

Linux « prédit » le coût de migration

Adapter à l'usage le paramètre `sched_migration_cost_ns` selon :

- besoins en cache
- (in)utilité de migrer

+ toutes sortes de possibilités de réglages avec les *cpuset*, *cgroups*, priorités, modes de *scheduling*...

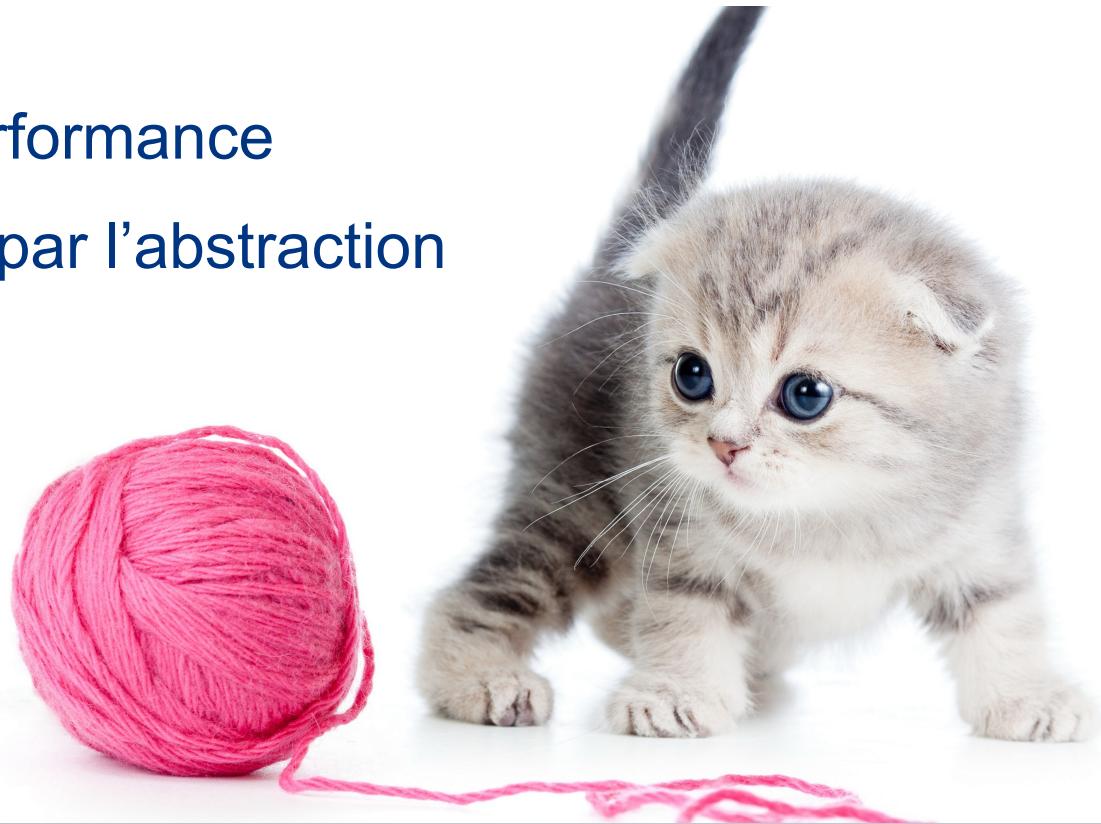
# Conclusion

« Concurrency is hard, let's go shopping »

Les *threads* ce n'est pas si mal ?

Équilibrer besoin en performance

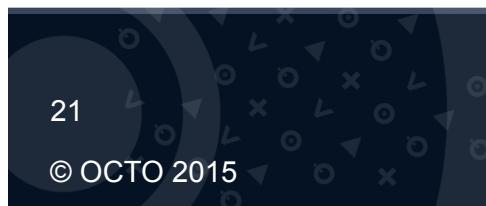
et gain apporté par l'abstraction



# Merci !

**Vous croyez que *les technologies*  
changent le monde ?**

**Nous aussi ! Rejoignez-nous !**  
[recrutement@octo.com](mailto:recrutement@octo.com)



21

© OCTO 2015

There is a better way

**OCTO**  
Technology

# Changement de contexte : nos mesures

Plateforme					Process switch + CPU Pinning	Thread switch + CPU Pinning	Thread switch SCHED_FIFO + CPU Pinning	Process switch	Thread switch
Hardware	Linux	Virtu. / Cont.	Commentaire	Taille machine cloud					
Laptop (1)	Kemel 4.4.0	N/A	Ubuntu 14.04 (LTS)	N/A	737 ns	623 ns	296 ns	2405 ns	2559 ns
Laptop (1)	Kemel 4.4.0	Docker	Ubuntu 14.04 (LTS)	N/A	778 ns	671 ns	345 ns	2231 ns	2798 ns
Laptop (1)	Kemel 4.4.0	VMWare	Ubuntu 14.04 (LTS)	N/A	988 ns	854 ns	332 ns	4600	5850 ns
AWS Instance	Kemel 4.4.0	AWS Cloud	Ubuntu 14.04 (LTS)	t2.micro (1 CPU)	877 ns	665 ns	315 ns	878 ns	677 ns
AWS Instance	Kemel 4.4.0	AWS Cloud	Ubuntu 14.04 (LTS)	m4.xlarge (4 CPU)	975 ns	762 ns	352 ns	6907 ns	7754 ns
AWS Instance	Kemel 4.4.0	AWS Cloud	Ubuntu 14.04 (LTS)	c4.8xlarge (36 CPU)	785 ns	620 ns	284 ns	10232 ns	11088 ns

Résultats conservateurs car incluant le coût de l'appel SYS\_FUTEX

# Callbacks, «boucle d'évènements»

Exemple du navigateur Web

## Programme

```
console.log("1");

window.setTimeout(function () {
    console.log("3");
}, 1000);

console.log("2");
```

## Pile d'appels

## Web APIs

## Sortie



## File d'attente callbacks

# Callbacks, «boucle d'évènements»

Exemple du navigateur Web

## Programme

```
console.log("1");  
  
window.setTimeout(function () {  
    console.log("3");  
}, 1000);  
  
console.log("2");
```

## Pile d'appels

```
log(1)  
main()
```

## Web APIs

## Sortie

```
1
```



## File d'attente callbacks

# Callbacks, «boucle d'évènements»

Exemple du navigateur Web

## Programme

```
console.log("1");

window.setTimeout(function () {
    console.log("3");
}, 1000);

console.log("2");
```

## Pile d'appels

```
setTimeout()
main()
```

## Web APIs

## Sortie

```
1
```



## File d'attente callbacks

# Callbacks, «boucle d'évènements»

Exemple du navigateur Web

## Programme

```
console.log("1");

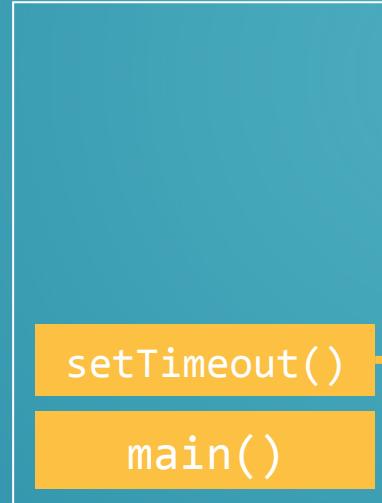
window.setTimeout(function () {
    console.log("3");
}, 1000);

console.log("2");
```

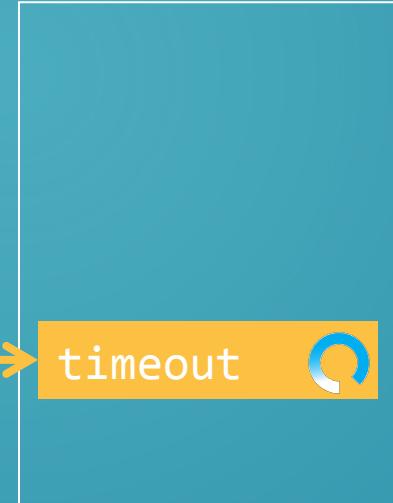
## Sortie

```
1
```

## Pile d'appels



## Web APIs



## File d'attente callbacks

# Callbacks, «boucle d'évènements»

Exemple du navigateur Web

## Programme

```
console.log("1");

window.setTimeout(function () {
    console.log("3");
}, 1000);

console.log("2");
```

## Pile d'appels

log(2)  
main()

## Web APIs

timeout 

## Sortie

1  
2



## File d'attente callbacks

# Callbacks, «boucle d'évènements»

Exemple du navigateur Web

## Programme

```
console.log("1");

window.setTimeout(function () {
    console.log("3");
}, 1000);

console.log("2");
```

## Pile d'appels

## Web APIs

## Sortie

```
1
2
```



## File d'attente callbacks

timeout A blue circular progress bar icon with a white 'Q' shape inside.

# Callbacks, «boucle d'évènements»

Exemple du navigateur Web

## Programme

```
console.log("1");

window.setTimeout(function () {
    console.log("3");
}, 1000);

console.log("2");
```

## Pile d'appels

## Web APIs

## Sortie

```
1
2
```

File d'attente callbacks

callback



# Callbacks, «boucle d'évènements»

Exemple du navigateur Web

## Programme

```
console.log("1");

window.setTimeout(function () {
    console.log("3");
}, 1000);

console.log("2");
```

## Sortie

```
1
2
```

## Pile d'appels



## Web APIs



callback()



File d'attente callbacks



# Callbacks, «boucle d'évènements»

Exemple du navigateur Web

## Programme

```
console.log("1");

window.setTimeout(function () {
    console.log("3");
}, 1000);

console.log("2");
```

## Pile d'appels

log(3)  
callback()

## Web APIs

## Sortie

```
1  
2  
3
```



## File d'attente callbacks

# Callbacks, «boucle d'évènements»

Exemple du navigateur Web

## Programme

```
console.log("1");

window.setTimeout(function () {
    console.log("3");
}, 0);

console.log("2");
```

## Pile d'appels

```
log(1)
main()
```

## Web APIs

## Sortie

```
1
```



## File d'attente callbacks

# Callbacks, «boucle d'évènements»

Exemple du navigateur Web

## Programme

```
console.log("1");

window.setTimeout(function () {
    console.log("3");
}, 0);

console.log("2");
```

## Pile d'appels

```
setTimeout()
main()
```

## Web APIs

## Sortie

```
1
```



## File d'attente callbacks

# Callbacks, «boucle d'évènements»

Exemple du navigateur Web

## Programme

```
console.log("1");

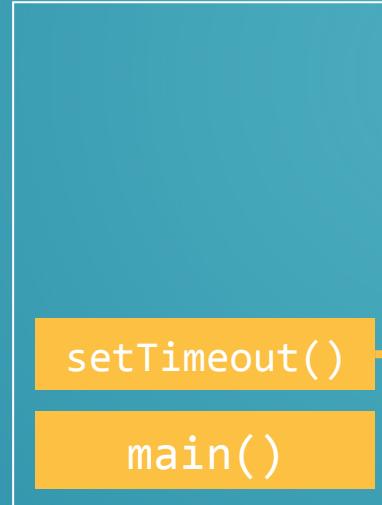
window.setTimeout(function () {
    console.log("3");
}, 0);

console.log("2");
```

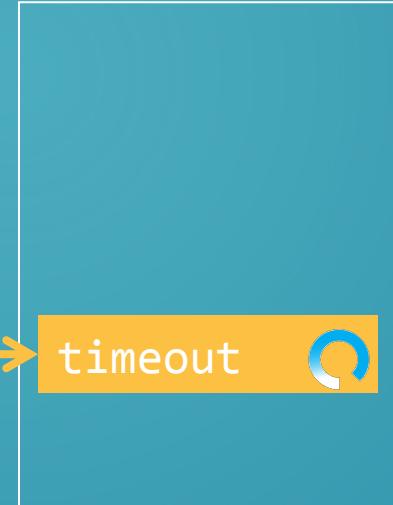
## Sortie

```
1
```

## Pile d'appels



## Web APIs



## File d'attente callbacks



# Callbacks, «boucle d'évènements»

Exemple du navigateur Web

## Programme

```
console.log("1");

window.setTimeout(function () {
    console.log("3");
}, 0);

console.log("2");
```

## Sortie

```
1
```

## Pile d'appels

```
main()
```

## Web APIs



File d'attente callbacks

```
callback
```

# Callbacks, «boucle d'évènements»

Exemple du navigateur Web

## Programme

```
console.log("1");

window.setTimeout(function () {
    console.log("3");
}, 0);

console.log("2");
```

## Pile d'appels

log(2)  
main()

## Web APIs

## Sortie

1  
2



## File d'attente callbacks

callback

# Callbacks, «boucle d'évènements»

Exemple du navigateur Web

## Programme

```
console.log("1");

window.setTimeout(function () {
    console.log("3");
}, 0);

console.log("2");
```

## Pile d'appels

## Web APIs

## Sortie

```
1
2
```



## File d'attente callbacks

callback

# Callbacks, «boucle d'évènements»

Exemple du navigateur Web

## Programme

```
console.log("1");

window.setTimeout(function () {
    console.log("3");
}, 0);

console.log("2");
```

## Sortie

```
1
2
```

## Pile d'appels



callback()

## Web APIs



File d'attente callbacks



# Callbacks, «boucle d'évènements»

Exemple du navigateur Web

## Programme

```
console.log("1");

window.setTimeout(function () {
    console.log("3");
}, 0);

console.log("2");
```

## Pile d'appels

log(3)  
callback()

## Web APIs

## Sortie

```
1  
2  
3
```



## File d'attente callbacks

# Générateurs : semi-coroutines / « coroutines »

```
async(function* () {
    var contenu1 = yield monReadFile('monFichier1');
    var contenu2 = yield monReadFile('monFichier2');
    console.log(contenu1+contenu2);
})();
```

```
function monReadFile(filename) {
    return function(callback) {
        fs.readFile(filename, 'utf8', callback);
    };
}
```



# Semi-coroutines

Exemple avec Node.js

## Programme

```
async(function* () {  
    var contenu = yield  
        monReadFile('monFichier');  
    console.log(contenu);  
})();
```

## Pile d'appels

```
monReadFile()  
async()  
main()
```

## File APIs

## Sortie



## File d'attente callbacks

# Semi-coroutines

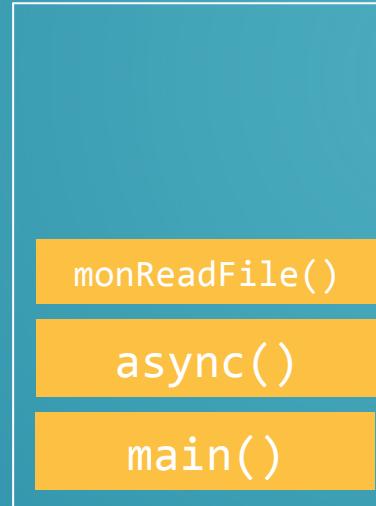
Exemple avec Node.js

## Programme

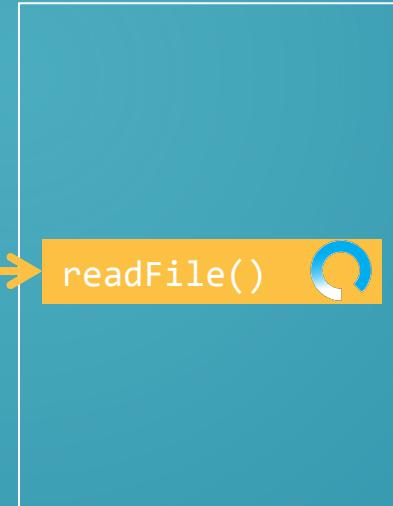
```
async(function* () {  
    var contenu = yield  
        monReadFile('monFichier');  
    console.log(contenu);  
})();
```

## Sortie

## Pile d'appels



## File APIs



## File d'attente callbacks

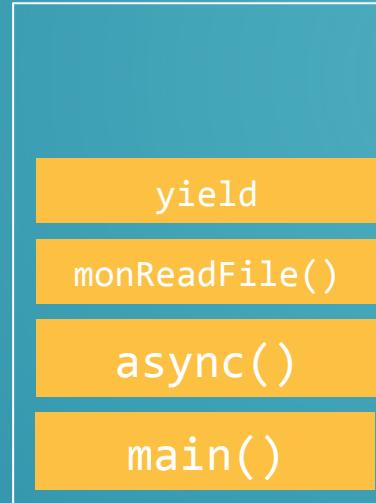
# Semi-coroutines

Exemple avec Node.js

## Programme

```
async(function* () {  
    var contenu = yield  
        monReadFile('monFichier');  
    console.log(contenu);  
})();
```

## Pile d'appels



## File APIs



## Sortie



## File d'attente callbacks

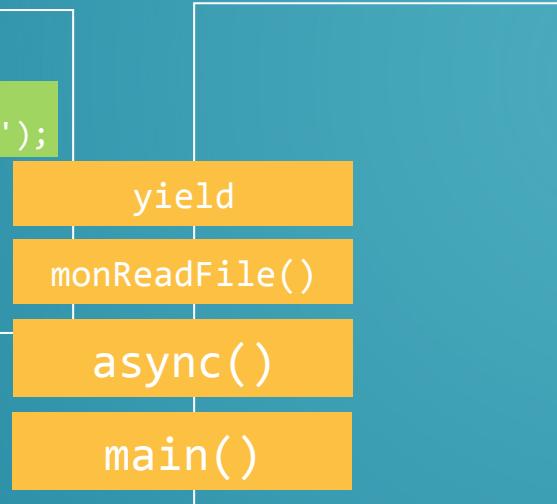
# Semi-coroutines

Exemple avec Node.js

## Programme

```
async(function* () {  
    var contenu = yield  
        monReadFile('monFichier');  
    console.log(contenu);  
})();
```

## Pile d'appels



## Filesystem APIs

readFile()

## Sortie



File d'attente callbacks

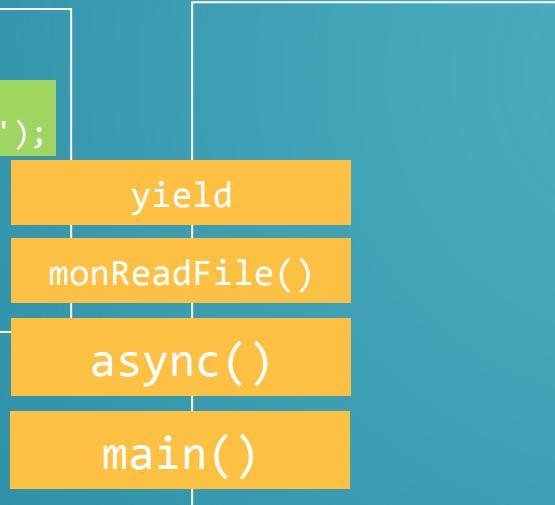
# Semi-coroutines

Exemple avec Node.js

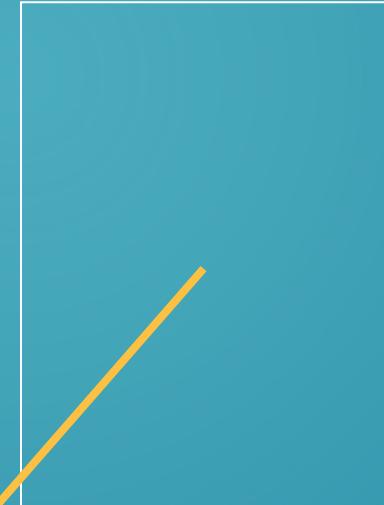
## Programme

```
async(function* () {  
    var contenu = yield  
        monReadFile('monFichier');  
    console.log(contenu);  
})();
```

## Pile d'appels



## Filesystem APIs



## Sortie



File d'attente callbacks

callback

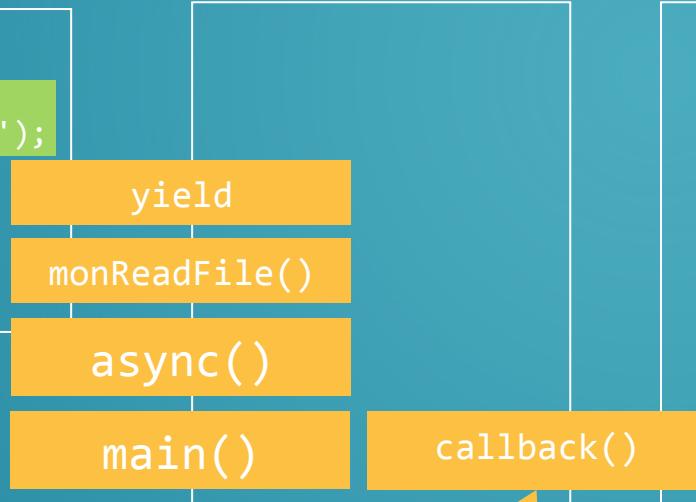
# Semi-coroutines

Exemple avec Node.js

## Programme

```
async(function* () {  
    var contenu = yield  
        monReadFile('monFichier');  
    console.log(contenu);  
})();
```

## Pile d'appels



## Filesystem APIs

## Sortie



File d'attente callbacks



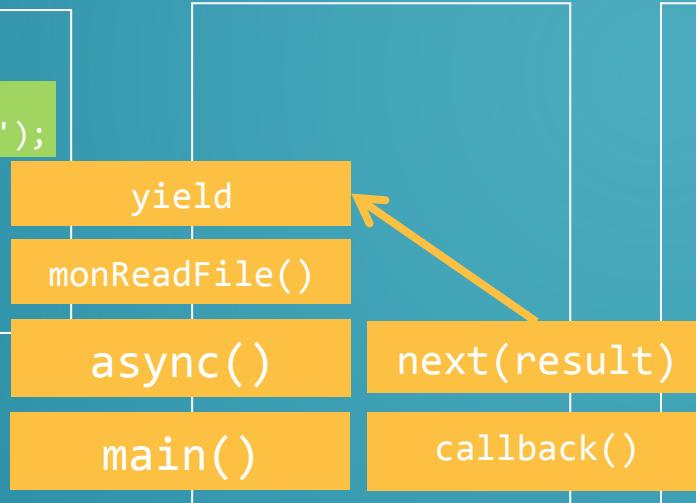
# Semi-coroutines

Exemple avec Node.js

## Programme

```
async(function* () {  
    var contenu = yield  
        monReadFile('monFichier');  
    console.log(contenu);  
})();
```

## Pile d'appels



## Filesystem APIs



## Sortie



## File d'attente callbacks



# Semi-coroutines

Exemple avec Node.js

## Programme

```
async(function* () {  
    var contenu = yield  
        monReadFile('monFichier');  
    console.log(contenu);  
})();
```

## Pile d'appels

```
var ...  
monReadFile()  
async()  
main()
```

## Filesystem APIs

## Sortie



## File d'attente callbacks

# Générateurs

Python, Javascript, Ruby ...

`yield` suspend l'exécution  
jusqu'à un appel à `next()`  
retourne l'argument passé à `next()`

`next()` retourne un objet `{value, done}`

```
function* makeIterator() {  
    yield 1;  
    yield 2;  
    yield 3;  
}  
  
var iterator = makeIterator();  
console.log(iterator.next()); // { value: 1, done: false }  
console.log(iterator.next()); // { value: 2, done: false }  
console.log(iterator.next()); // { value: 3, done: false }  
console.log(iterator.next()); // { value: undefined, done: true }
```

Valeur passée à `yield`



# Semi-coroutines / « coroutines » sur générateurs

```
async(function* () {  
    var contenu1 = yield monReadFile('monFichier1');  
    var contenu2 = yield monReadFile('monFichier2');  
    console.log(contenu1+contenu2);  
})();
```

```
function monReadFile(filename) {  
    return function(callback) {  
        fs.readFile(filename, 'utf8', callback);  
    };  
}
```

= **async/await**

