Codage asynchrone Techniques modernes

Nicolas Noble
nicolas@grumpycoder.net

17 Février 2014

Outline

1. Problèmes
   - Software
   - Hardware
   - Abstractions Fuyantes
2. Solutions
   - Threads
   - Threadpools
   - Atomics
   - C++11
   - Réacteurs
   - Co-routines
   - MapReduce
3. Conclusion

Outline

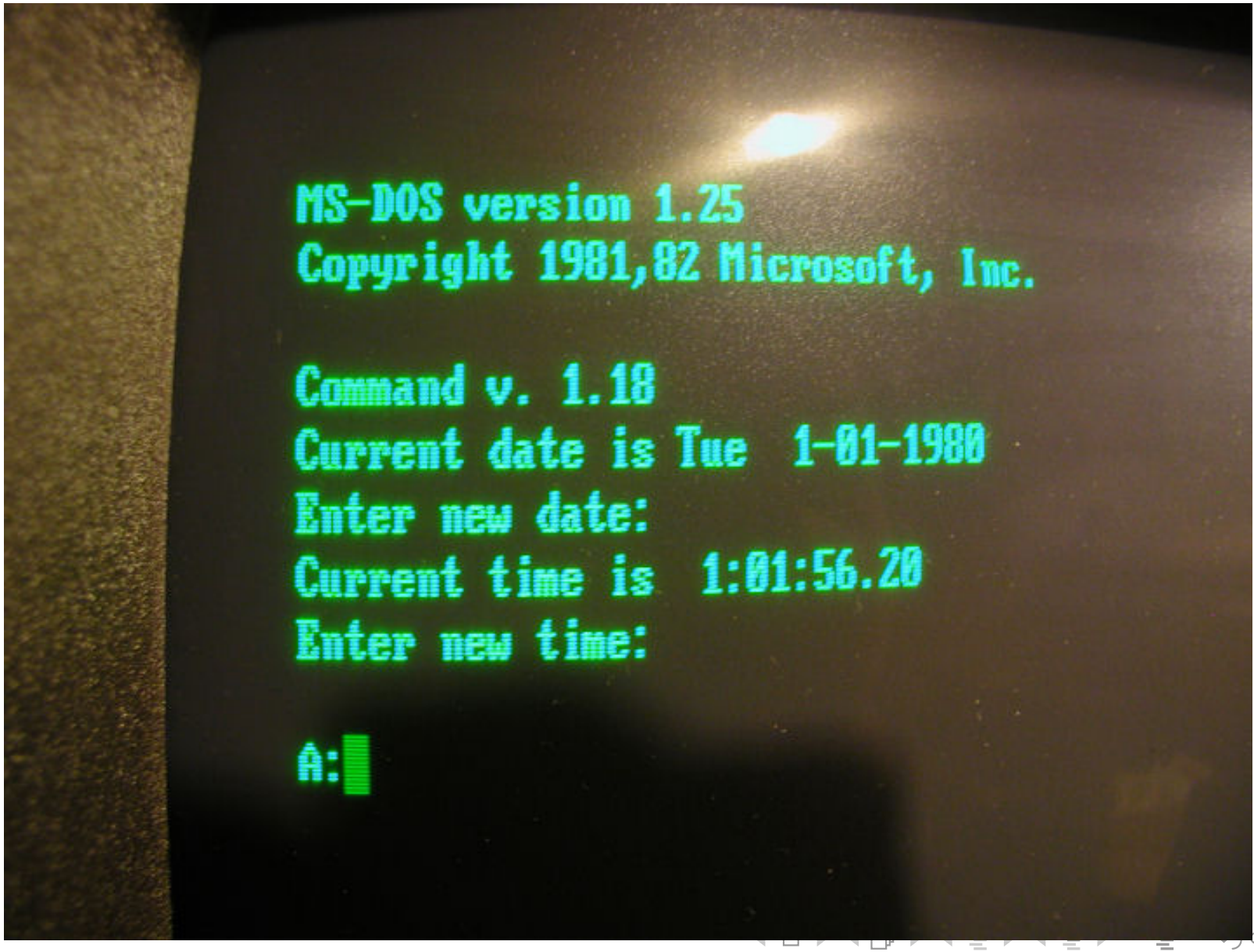
1. Problèmes
   - Software
   - Hardware
   - Abstractions Fuyantes
2. Solutions
   - Threads
   - Threadpools
   - Atomics
   - C++11
   - Réacteurs
   - Co-routines
   - MapReduce
3. Conclusion

Outline

1. Problèmes
   - Software
   - Hardware
   - Abstractions Fuyantes
2. Solutions
   - Threads
   - Threadpools
   - Atomics
   - C++11
   - Réacteurs
   - Co-routines
   - MapReduce
3. Conclusion

Outline

Evolution du software



Evolution du software

- L'OS n'était qu'une simple couche d'API.
- Votre software controllait toute la machine.

Example

```
int main() {
  init();
  while (running) {
    doInputs();
    doProcessing();
    doOutputs();
  }
  uninit();
  return errorCode;
}
```

Evolution du software

- L'OS n'était qu'une simple couche d'API.
- Votre software controllait toute la machine.

Example

```
int main() {
  init();
  while (running) {
    doInputs();
    doProcess();
    doOutputs();
  }
  uninit();
  return errorCode;
}
```

Evolution du software

- L'OS n'était qu'une simple couche d'API.
- Votre software controllait toute la machine.

Example

```
int main() {
    init();
    while (running) {
        doInputs();
        doProcess();
        doOutputs();
    }
    uninit();
    return errorCode;
}
```

Evolution du software

- L'OS n'était qu'une simple couche d'API.
- Votre software controllait toute la machine.

Example

```
int main() {
    init();
    while (running) {
        doInputs();
        doProcess();
        doOutputs();
    }
    uninit();
    return errorCode;
}
```

Evolution du software

- L'OS n'était qu'une simple couche d'API.
- Votre software controllait toute la machine.

Example

```
int main() {
    init();
    while (running) {
        doInputs();
        doProcess();
        doOutputs();
    }
    uninit();
    return errorCode;
}
```

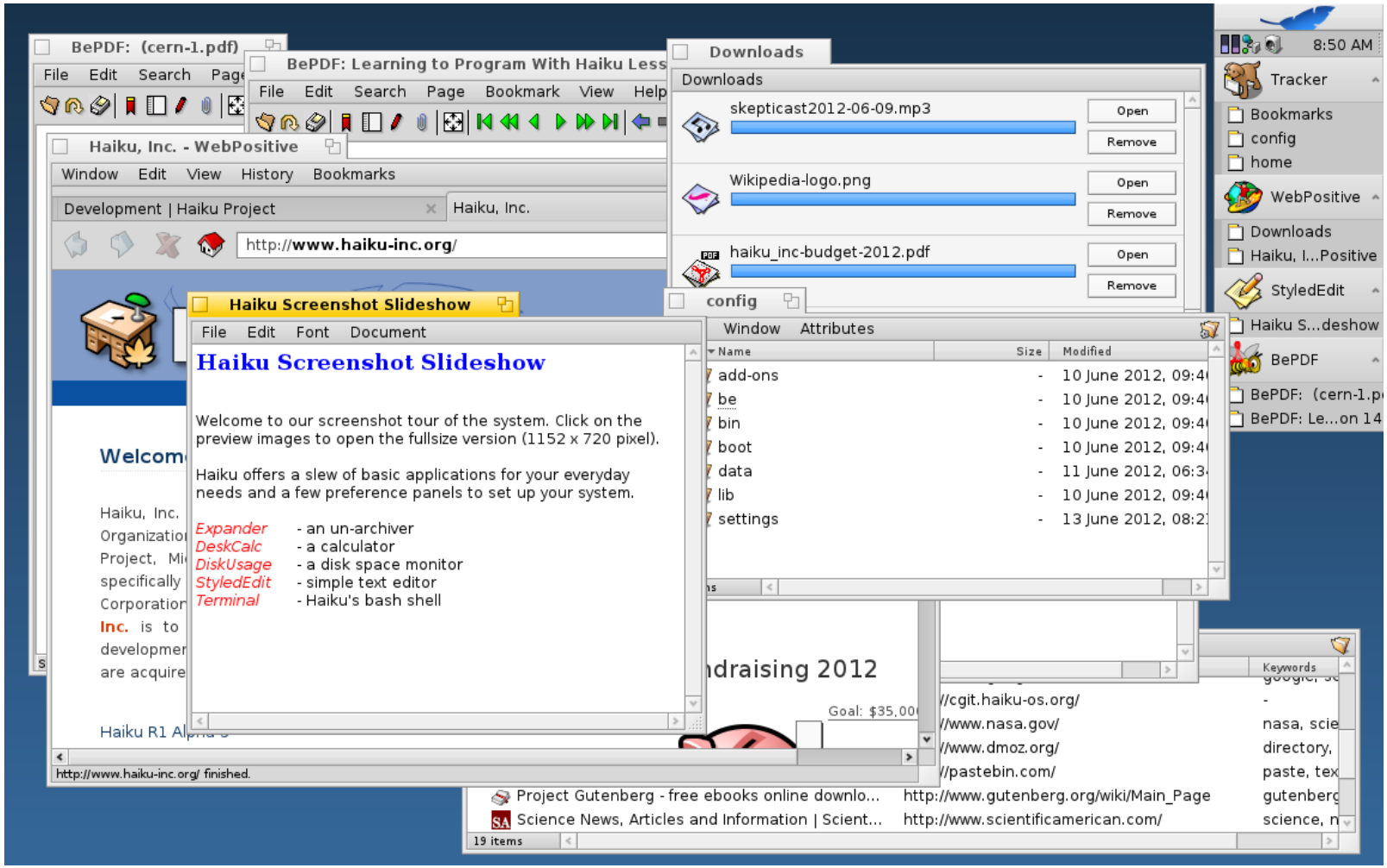
Evolution du software

- L'OS n'était qu'une simple couche d'API.
- Votre software controllait toute la machine.

Example

```
int main() {
    init();
    while (running) {
        doInputs();
        doProcess();
        doOutputs();
    }
    uninit();
    return errorCode;
}
```

Evolution du software



Evolution du software

- L'OS d'aujourd'hui est un kernel préemptif.
  → Il décide de qui s'exécute et quand.
- Votre software n'est plus tout seul, et n'est plus au contrôle de la machine.
- L'ancienne architecture typique n'est plus adaptée, mais est toujours en vogue.
  → Nouvelles architectures logicielles peu connues et mal maîtrisées.
- Même les outils classiques (grep, sed, awk, ...) ne sont pas adaptés.
  `http://www.rankfocus.com/use-cpu-cores-linux-commands`

Evolution du software

- L'OS d'aujourd'hui est un kernel préemptif.
  → Il décide de qui s'exécute et quand.
- Votre software n'est plus tout seul, et n'est plus au contrôle de la machine.
- L'ancienne architecture typique n'est plus adaptée, mais est toujours en vogue.
  → Nouvelles architectures logicielles peu connues et mal maîtrisées.
- Même les outils classiques (grep, sed, awk, ...) ne sont pas adaptés.
  `http://www.rankfocus.com/use-cpu-cores-linux-commands`

Evolution du software

- L'OS d'aujourd'hui est un kernel préemptif.
  → Il décide de qui s'exécute et quand.

- Votre software n'est plus tout seul, et n'est plus au contrôle de la machine.

- L'ancienne architecture typique n'est plus adaptée, mais est toujours en vogue.
  → Nouvelles architectures logicielles peu connues et mal maîtrisées.

- Même les outils classiques (grep, sed, awk, ...) ne sont pas adaptés.
  `http://www.rankfocus.com/use-cpu-cores-linux-commands`

Evolution du software

- L'OS d'aujourd'hui est un kernel préemptif.
  → Il décide de qui s'exécute et quand.
- Votre software n'est plus tout seul, et n'est plus au contrôle de la machine.
- L'ancienne architecture typique n'est plus adaptée, mais est toujours en vogue.
  → Nouvelles architectures logicielles peu connues et mal maîtrisées.
- Même les outils classiques (grep, sed, awk, ...) ne sont pas adaptés.
  http://www.rankfocus.com/use-cpu-cores-linux-commands

Evolution du software

- L'OS d'aujourd'hui est un kernel préemptif.
  → Il décide de qui s'exécute et quand.
- Votre software n'est plus tout seul, et n'est plus au contrôle de la machine.
- L'ancienne architecture typique n'est plus adaptée, mais est toujours en vogue.
  → Nouvelles architectures logicielles peu connues et mal maîtrisées.
- Même les outils classiques (grep, sed, awk, ...) ne sont pas adaptés.

  http://www.rankfocus.com/use-cpu-cores-linux-commands

Evolution du software

- L'OS d'aujourd'hui est un kernel préemptif.
  → Il décide de qui s'exécute et quand.
- Votre software n'est plus tout seul, et n'est plus au contrôle de la machine.
- L'ancienne architecture typique n'est plus adaptée, mais est toujours en vogue.
  → Nouvelles architectures logicielles peu connues et mal maîtrisées.
- Même les outils classiques (grep, sed, awk, ...) ne sont pas adaptés.

  http://www.rankfocus.com/use-cpu-cores-linux-commands

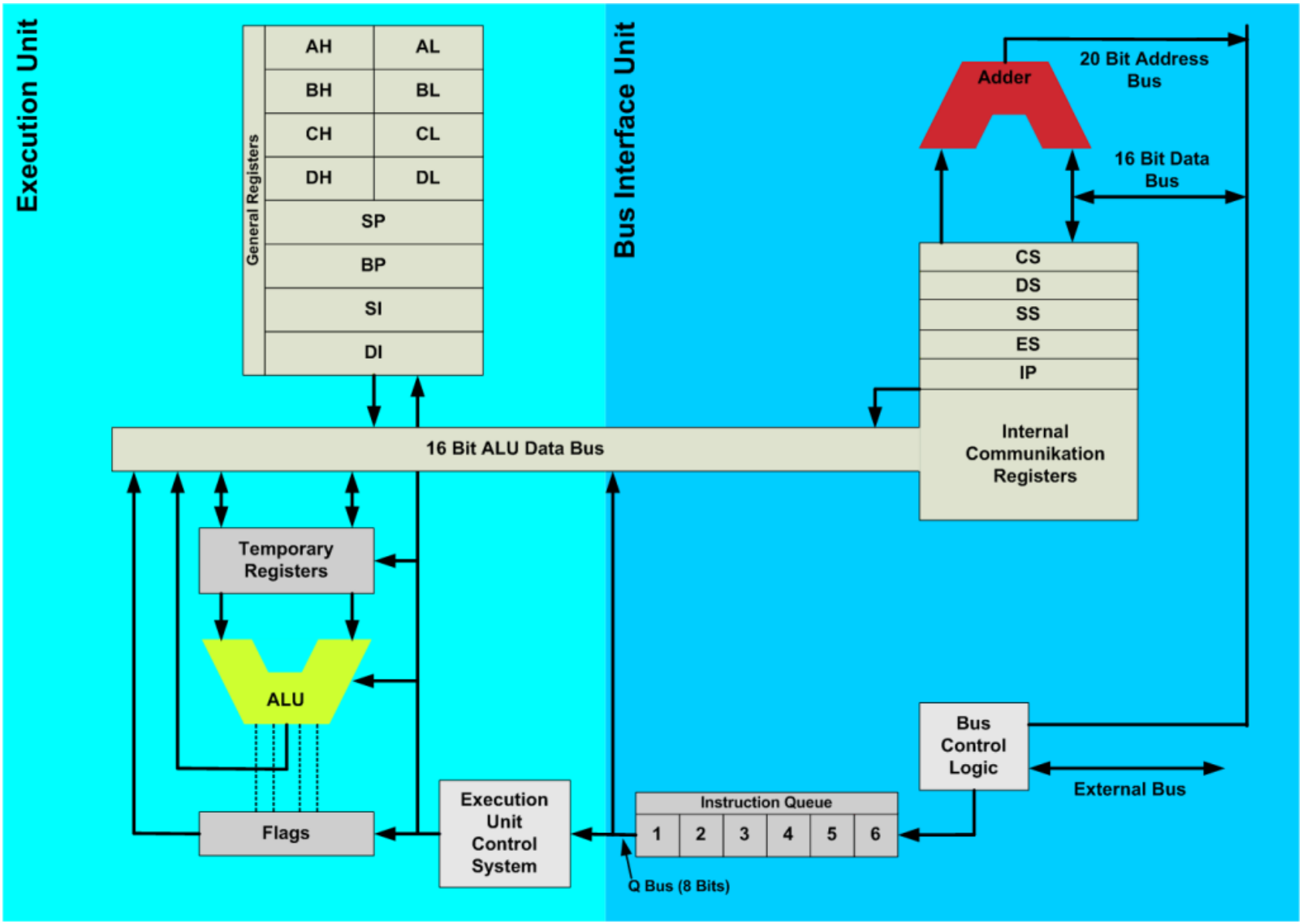
Evolution du software

- L'OS d'aujourd'hui est un kernel préemptif.
  → Il décide de qui s'exécute et quand.
- Votre software n'est plus tout seul, et n'est plus au contrôle de la machine.
- L'ancienne architecture typique n'est plus adaptée, mais est toujours en vogue.
  → Nouvelles architectures logicielles peu connues et mal maîtrisées.
- Même les outils classiques (grep, sed, awk, ...) ne sont pas adaptés.

  `http://www.rankfocus.com/use-cpu-cores-linux-commands`

Outline

Evolution du hardware



Evolution du hardware

- Les ordinateurs étaient conçus de manière linéaire.
- Un seul CPU communique via un seul bus à la mémoire et aux périphériques.
- Programmation induite linéaire.

Evolution du hardware

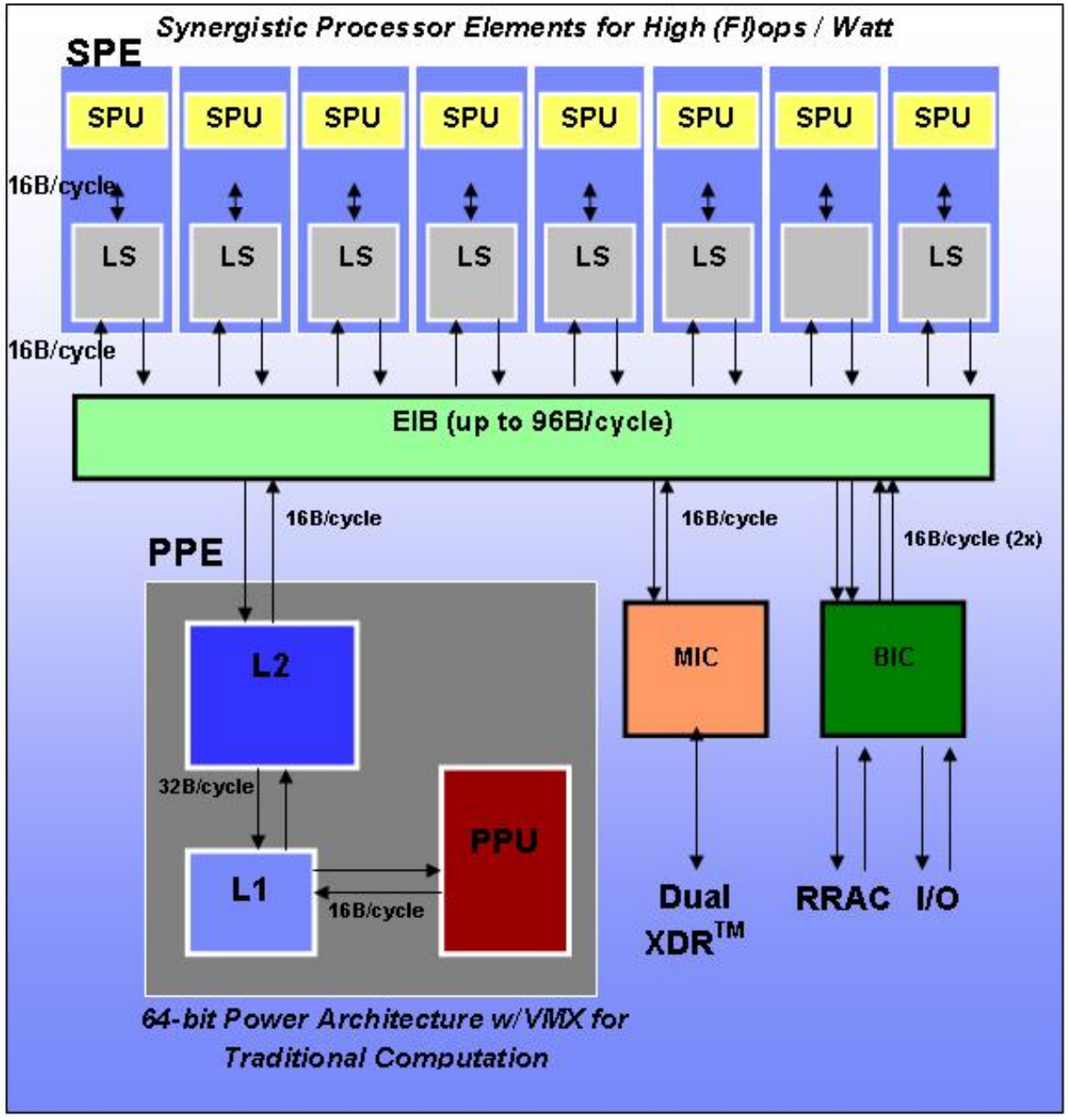
- Les ordinateurs étaient conçus de manière linéaire.
- Un seul CPU communique via un seul bus à la mémoire et aux périphériques.
- Programmation induite linéaire.

Evolution du hardware

- Les ordinateurs étaient conçus de manière linéaire.
- Un seul CPU communique via un seul bus à la mémoire et aux périphériques.
- Programmation induite linéaire.

Evolution du hardware

- Les ordinateurs étaient conçus de manière linéaire.
- Un seul CPU communique via un seul bus à la mémoire et aux périphériques.
- Programmation induite linéaire.


Synergistic Processor Elements for High (Fl)ops / Watt
SPE
SPU
SPU
SPU
SPU
SPU
SPU
SPU
SPU
16B/cycle
LS
LS
LS
LS
LS
LS
LS
16B/cycle
EIB (up to 96B/cycle)
16B/cycle
16B/cycle
16B/cycle (2x)
PPE
L2
32B/cycle
L1
16B/cycle
PPU
MIC
BIC
Dual XDR™
RRAC
I/O
64-bit Power Architecture w/VMX for Traditional Computation

Evolution du hardware

- Loi de Moore:

  La puissance des CPUs double tous les 18 mois.

- Basé sur la miniaturisation des transistors.
- *Le Mur* ralentit la loi de Moore.
  → Les effets quantiques empêchent la miniaturisation des transistors.
- La parallélisation est la solution employée pour contourner cette limitation.
- Processeurs de plus en plus complexes, avec des architectures qui ne sont plus linéaires.
- De plus en plus de machines interconnectées.

Evolution du hardware

- Loi de Moore:
  La puissance des CPUs double tous les 18 mois.
- Basé sur la miniaturisation des transistors.
- *Le Mur* ralentit la loi de Moore.
  → Les effets quantiques empêchent la miniaturisation des transistors.
- La parallélisation est la solution employée pour contourner cette limitation.
- Processeurs de plus en plus complexes, avec des architectures qui ne sont plus linéaires.
- De plus en plus de machines interconnectées.

Evolution du hardware

- Loi de Moore:

  La puissance des CPUs double tous les 18 mois.

- Basé sur la miniaturisation des transistors.

- *Le Mur* ralentit la loi de Moore.
  → Les effets quantiques empêchent la miniaturisation des transistors.
- La parallélisation est la solution employée pour contourner cette limitation.
- Processeurs de plus en plus complexes, avec des architectures qui ne sont plus linéaires.
- De plus en plus de machines interconnectées.

Evolution du hardware

- Loi de Moore:
  La puissance des CPUs double tous les 18 mois.
- Basé sur la miniaturisation des transistors.
- *Le Mur* ralentit la loi de Moore.
  → Les effets quantiques empêchent la miniaturisation des transistors.
- La parallélisation est la solution employée pour contourner cette limitation.
- Processeurs de plus en plus complexes, avec des architectures qui ne sont plus linéaires.
- De plus en plus de machines interconnectées.

Evolution du hardware

- Loi de Moore:
  La puissance des CPUs double tous les 18 mois.
- Basé sur la miniaturisation des transistors.
- *Le Mur* ralentit la loi de Moore.
  $\rightarrow$ Les effets quantiques empêchent la miniaturisation des transistors.
- La parallélisation est la solution employée pour contourner cette limitation.
- Processeurs de plus en plus complexes, avec des architectures qui ne sont plus linéaires.
- De plus en plus de machines interconnectées.

Evolution du hardware

- Loi de Moore:
  La puissance des CPUs double tous les 18 mois.
- Basé sur la miniaturisation des transistors.
- *Le Mur* ralentit la loi de Moore.
  $\rightarrow$ Les effets quantiques empêchent la miniaturisation des transistors.
- La parallélisation est la solution employée pour contourner cette limitation.
- Processeurs de plus en plus complexes, avec des architectures qui ne sont plus linéaires.
- De plus en plus de machines interconnectées.

Evolution du hardware

- Loi de Moore:
  La puissance des CPUs double tous les 18 mois.
- Basé sur la miniaturisation des transistors.
- *Le Mur* ralentit la loi de Moore.
  $\rightarrow$ Les effets quantiques empêchent la miniaturisation des transistors.
- La parallélisation est la solution employée pour contourner cette limitation.
- Processeurs de plus en plus complexes, avec des architectures qui ne sont plus linéaires.
- De plus en plus de machines interconnectées.

Evolution du hardware

- Loi de Moore:
  La puissance des CPUs double tous les 18 mois.
- Basé sur la miniaturisation des transistors.
- *Le Mur* ralentit la loi de Moore.
  $\rightarrow$ Les effets quantiques empêchent la miniaturisation des transistors.
- La parallélisation est la solution employée pour contourner cette limitation.
- Processeurs de plus en plus complexes, avec des architectures qui ne sont plus linéaires.
- De plus en plus de machines interconnectées.

Outline

Abstractions Fuyantes

- Terme créé par Joel Spolsky.
  → *The Law of Leaky Abstractions*
- Les APIs nous mentent.
- La plus grande abstraction fuyante: les I/O.
  → Abstraction software et hardware.
- Un disque logique peut être un partage réseau.
- Un disque dur ne peut lire qu'un secteur à la fois.
- Un disque dur à plateaux a un temps de positionnement non négligeable.

`http://www.joelonsoftware.com/articles/LeakyAbstractions.html`

Abstractions Fuyantes

- Terme créé par Joel Spolsky.
  $\rightarrow$ *The Law of Leaky Abstractions*
- Les APIs nous mentent.
- La plus grande abstraction fuyante: les I/O.
  $\rightarrow$ Abstraction software et hardware.
- Un disque logique peut être un partage réseau.
- Un disque dur ne peut lire qu'un secteur à la fois.
- Un disque dur à plateaux a un temps de positionnement non négligeable.

`http://www.joelonsoftware.com/articles/LeakyAbstractions.html`

Abstractions Fuyantes

- Terme créé par Joel Spolsky.
  $\rightarrow$ *The Law of Leaky Abstractions*
- Les APIs nous mentent.
- La plus grande abstraction fuyante: les I/O.
  $\rightarrow$ Abstraction software et hardware.
- Un disque logique peut être un partage réseau.
- Un disque dur ne peut lire qu'un secteur à la fois.
- Un disque dur à plateaux a un temps de positionnement non négligeable.

`http://www.joelonsoftware.com/articles/LeakyAbstractions.html`

Abstractions Fuyantes

- Terme créé par Joel Spolsky.
  → *The Law of Leaky Abstractions*
- Les APIs nous mentent.

- La plus grande abstraction fuyante: les I/O.
  → Abstraction software et hardware.
- Un disque logique peut être un partage réseau.
- Un disque dur ne peut lire qu'un secteur à la fois.
- Un disque dur à plateaux a un temps de positionnement non négligeable.

http://www.joelonsoftware.com/articles/LeakyAbstractions.html

Abstractions Fuyantes

- Terme créé par Joel Spolsky.
  → *The Law of Leaky Abstractions*
- Les APIs nous mentent.

- La plus grande abstraction fuyante: les I/O.
  → Abstraction software et hardware.
- Un disque logique peut être un partage réseau.

- Un disque dur ne peut lire qu'un secteur à la fois.
- Un disque dur à plateaux a un temps de positionnement non négligeable.

http://www.joelonsoftware.com/articles/LeakyAbstractions.html

Abstractions Fuyantes

- Terme créé par Joel Spolsky.
  → *The Law of Leaky Abstractions*
- Les APIs nous mentent.

- La plus grande abstraction fuyante: les I/O.
  → Abstraction software et hardware.

- Un disque logique peut être un partage réseau.

- Un disque dur ne peut lire qu'un secteur à la fois.
- Un disque dur à plateaux a un temps de positionnement non négligeable.

http://www.joelonsoftware.com/articles/LeakyAbstractions.html

Abstractions Fuyantes

- Terme créé par Joel Spolsky.
  → *The Law of Leaky Abstractions*
- Les APIs nous mentent.

- La plus grande abstraction fuyante: les I/O.
  → Abstraction software et hardware.

- Un disque logique peut être un partage réseau.

- Un disque dur ne peut lire qu'un secteur à la fois.
- Un disque dur à plateaux a un temps de positionnement non négligeable.

`http://www.joelonsoftware.com/articles/LeakyAbstractions.html`

Abstractions Fuyantes

- Terme créé par Joel Spolsky.
  → *The Law of Leaky Abstractions*
- Les APIs nous mentent.

- La plus grande abstraction fuyante: les I/O.
  → Abstraction software et hardware.

- Un disque logique peut être un partage réseau.

- Un disque dur ne peut lire qu'un secteur à la fois.
- Un disque dur à plateaux a un temps de positionnement non négligeable.

http://www.joelonsoftware.com/articles/LeakyAbstractions.html

Abstractions Fuyantes

- Terme créé par Joel Spolsky.
  → *The Law of Leaky Abstractions*
- Les APIs nous mentent.

- La plus grande abstraction fuyante: les I/O.
  → Abstraction software et hardware.

- Un disque logique peut être un partage réseau.

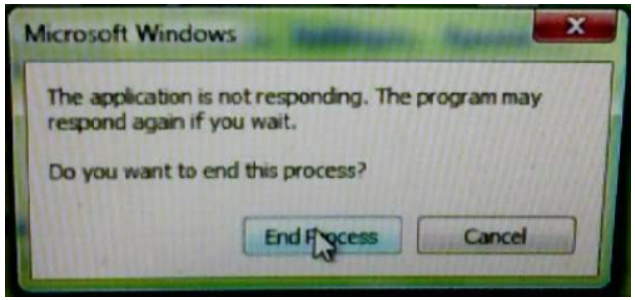
- Un disque dur ne peut lire qu'un secteur à la fois.
- Un disque dur à plateaux a un temps de positionnement non négligeable.

`http://www.joelonsoftware.com/articles/LeakyAbstractions.html`

Outline

Thread d'I/O

- Application plus répondante.



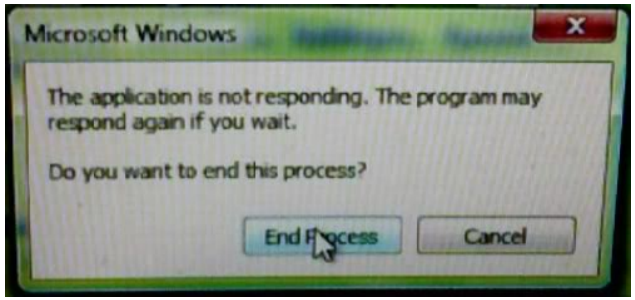
- Un seul thread réservé aux I/O.
  → Modèle N-producteurs/1-consommateur.
- Les producteurs créent des requêtes d'I/O.
- Le consommateur exécute les requêtes à la chaîne.

Erreur classique à éviter: *finish before returning*.

- Les callbacks *doivent* s'exécuter sur le thread du producteur.
  → Méchanismes de signalisation inter-threads.

Thread d'I/O

- Application plus répondante.



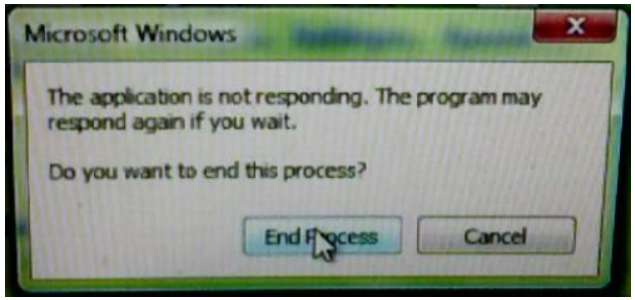
- Un seul thread réservé aux I/O.
  → Modèle N-producteurs/1-consommateur.
- Les producteurs créent des requêtes d'I/O.
- Le consommateur exécute les requêtes à la chaîne.

Erreur classique à éviter: *finish before returning*.

- Les callbacks *doivent* s'exécuter sur le thread du producteur.
  → Méchanismes de signalisation inter-threads.

Thread d'I/O

- Application plus répondante.



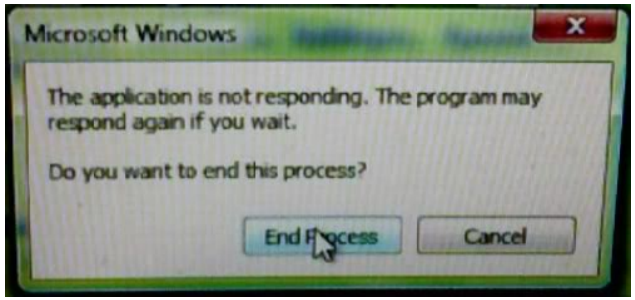
- Un seul thread réservé aux I/O.
  → Modèle N-producteurs/1-consommateur.
- Les producteurs créent des requêtes d'I/O.
- Le consommateur exécute les requêtes à la chaîne.

Erreur classique à éviter: *finish before returning*.

- Les callbacks *doivent* s'exécuter sur le thread du producteur.
  → Méchanismes de signalisation inter-threads.

Thread d'I/O

- Application plus répondante.



- Un seul thread réservé aux I/O.
  → Modèle N-producteurs/1-consommateur.

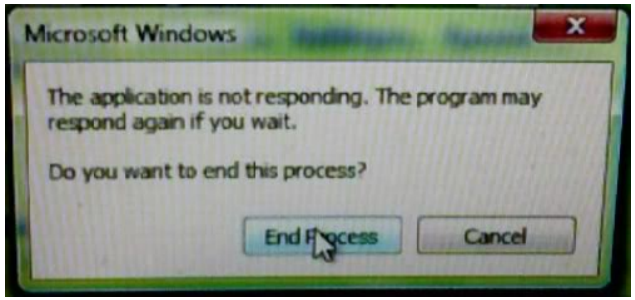
- Les producteurs créent des requêtes d'I/O.
- Le consommateur exécute les requêtes à la chaîne.

Erreur classique à éviter: *finish before returning*.

- Les callbacks *doivent* s'exécuter sur le thread du producteur.
  → Méchanismes de signalisation inter-threads.

Thread d'I/O

- Application plus répondante.



- Un seul thread réservé aux I/O.
  $\rightarrow$ Modèle N-producteurs/1-consommateur.

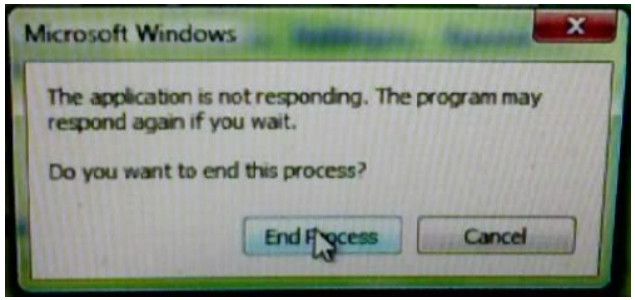
- Les producteurs créent des requêtes d'I/O.
- Le consommateur exécute les requêtes à la chaîne.

Erreur classique à éviter: *finish before returning*.

- Les callbacks *doivent* s'exécuter sur le thread du producteur.
  $\rightarrow$ Méchanismes de signalisation inter-threads.

Thread d'I/O

- Application plus répondante.



- Un seul thread réservé aux I/O.
  $\rightarrow$ Modèle N-producteurs/1-consommateur.

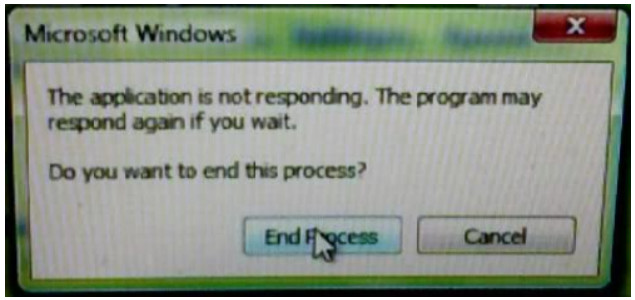
- Les producteurs créent des requêtes d'I/O.
- Le consommateur exécute les requêtes à la chaîne.

Erreur classique à éviter: *finish before returning*.

- Les callbacks *doivent* s'exécuter sur le thread du producteur.
  $\rightarrow$ Méchanismes de signalisation inter-threads.

Thread d'I/O

- Application plus répondante.



- Un seul thread réservé aux I/O.
  $\rightarrow$ Modèle N-producteurs/1-consommateur.

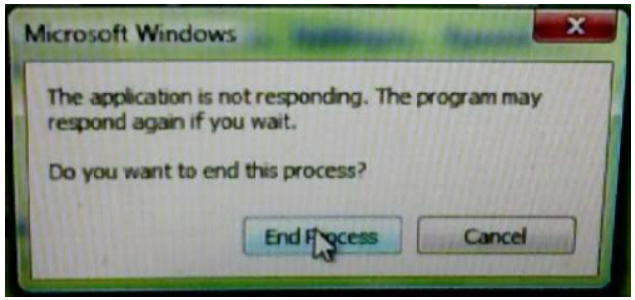
- Les producteurs créent des requêtes d'I/O.
- Le consommateur exécute les requêtes à la chaîne.

Erreur classique à éviter: *finish before returning*.

- Les callbacks *doivent* s'exécuter sur le thread du producteur.
  $\rightarrow$ Méchanismes de signalisation inter-threads.

Thread d'I/O

- Application plus répondante.



- Un seul thread réservé aux I/O.
  $\rightarrow$ Modèle N-producteurs/1-consommateur.

- Les producteurs créent des requêtes d'I/O.
- Le consommateur exécute les requêtes à la chaîne.

Erreur classique à éviter: *finish before returning*.

- Les callbacks *doivent* s'exécuter sur le thread du producteur.
  $\rightarrow$ Méchanismes de signalisation inter-threads.

Librairies de threading

- Parfois sous-exploitées.

```
create
mutex_lock
```

- Beaucoup d'autres primitives.
  - → Readers-writer locks.
  - → Barriers.
  - → Conditions.
  - → Thread local storage.
  - → Semaphores.

Les signaux Unix ne sont pas compatibles.

- Bien souvent sur-exploitées.
  - → Un socket suffit pour synchroniser deux threads.

Librairies de threading

- Parfois sous-exploitées.

  `create`
  `mutex_lock`

- Beaucoup d'autres primitives.
  → Readers-writer locks.
  → Barriers.
  → Conditions.
  → Thread local storage.
  → Semaphores.

Les signaux Unix ne sont pas compatibles.

- Bien souvent sur-exploitées.
  → Un socket suffit pour synchroniser deux threads.

Librairies de threading

- Parfois sous-exploitées.

  `create`
  `mutex_lock`

- Beaucoup d'autres primitives.
  → Readers-writer locks.
  → Barriers.
  → Conditions.
  → Thread local storage.
  → Semaphores.

  Les signaux Unix ne sont pas compatibles.

- Bien souvent sur-exploitées.
  → Un socket suffit pour synchroniser deux threads.

Librairies de threading

- Parfois sous-exploitées.

  `create`
  `mutex_lock`

- Beaucoup d'autres primitives.

  → Readers-writer locks.
  → Barriers.
  → Conditions.
  → Thread local storage.
  → Semaphores.

Les signaux Unix ne sont pas compatibles.

- Bien souvent sur-exploitées.

  → Un socket suffit pour synchroniser deux threads.

Librairies de threading

- Parfois sous-exploitées.

  `create`
  `mutex_lock`

- Beaucoup d'autres primitives.

  → Readers-writer locks.
  → Barriers.
  → Conditions.
  → Thread local storage.
  → Semaphores.

  Les signaux Unix ne sont pas compatibles.

- Bien souvent sur-exploitées.

  → Un socket suffit pour synchroniser deux threads.

Librairies de threading

- Parfois sous-exploitées.

  `create`
  `mutex_lock`

- Beaucoup d'autres primitives.

  → Readers-writer locks.
  → Barriers.
  → Conditions.
  → Thread local storage.
  → Semaphores.

Les signaux Unix ne sont pas compatibles.

- Bien souvent sur-exploitées.

  → Un socket suffit pour synchroniser deux threads.

Librairies de threading

- Parfois sous-exploitées.

  `create`
  `mutex_lock`

- Beaucoup d'autres primitives.

  → Readers-writer locks.
  → Barriers.
  → Conditions.
  → Thread local storage.
  → Semaphores.

Les signaux Unix ne sont pas compatibles.

- Bien souvent sur-exploitées.

  → Un socket suffit pour synchroniser deux threads.

Librairies de threading

- Parfois sous-exploitées.

  `create`
  `mutex_lock`

- Beaucoup d'autres primitives.

  → Readers-writer locks.
  → Barriers.
  → Conditions.
  → Thread local storage.
  → Semaphores.

Les signaux Unix ne sont pas compatibles.

- Bien souvent sur-exploitées.

  → Un socket suffit pour synchroniser deux threads.

Librairies de threading

- Parfois sous-exploitées.

  `create`
  `mutex_lock`

- Beaucoup d'autres primitives.

  → Readers-writer locks.
  → Barriers.
  → Conditions.
  → Thread local storage.
  → Semaphores.

Les signaux Unix ne sont pas compatibles.

- Bien souvent sur-exploitées.

  → Un socket suffit pour synchroniser deux threads.

Librairies de threading

- Parfois sous-exploitées.

  `create`
  `mutex_lock`

- Beaucoup d'autres primitives.

  → Readers-writer locks.
  → Barriers.
  → Conditions.
  → Thread local storage.
  → Semaphores.

  Les signaux Unix ne sont pas compatibles.

- Bien souvent sur-exploitées.

  → Un socket suffit pour synchroniser deux threads.

Librairies de threading

- Parfois sous-exploitées.

  `create`
  `mutex_lock`

- Beaucoup d'autres primitives.

  → Readers-writer locks.
  → Barriers.
  → Conditions.
  → Thread local storage.
  → Semaphores.

  Les signaux Unix ne sont pas compatibles.

- Bien souvent sur-exploitées.
  → Un socket suffit pour synchroniser deux threads.

Abus de threads

- Trop de threads tue le thread.
- Coût d'un thread élevé.

```
# cat /proc/1161/maps
7fedcd9b2000-7fedcd9b3000 ---p 00000000 00:00 0
7fedcd9b3000-7fedce1b3000 rw-p 00000000 00:00 0 [stack:1499]
7fedce1b3000-7fedce1b4000 ---p 00000000 00:00 0
7fedce1b4000-7fedce9b4000 rw-p 00000000 00:00 0 [stack:1498]
7fedce9b4000-7fedce9b5000 ---p 00000000 00:00 0
7fedce9b5000-7fedcf1b5000 rw-p 00000000 00:00 0 [stack:1497]
```

$\rightarrow$ 8Mo par thread.

- Beaucoup de bugs de synchronisation.
- Nécessite d'être plus intelligent avec ses threads.

Abus de threads

- Trop de threads tue le thread.
- Coût d'un thread élevé.

```
# cat /proc/1161/maps
7fedcd9b2000-7fedcd9b3000 ---p 00000000 00:00 0
7fedcd9b3000-7fedce1b3000 rw-p 00000000 00:00 0 [stack:1499]
7fedce1b3000-7fedce1b4000 ---p 00000000 00:00 0
7fedce1b4000-7fedce9b4000 rw-p 00000000 00:00 0 [stack:1498]
7fedce9b4000-7fedce9b5000 ---p 00000000 00:00 0
7fedce9b5000-7fedcf1b5000 rw-p 00000000 00:00 0 [stack:1497]
```

$\rightarrow$ 8Mo par thread.

- Beaucoup de bugs de synchronisation.
- Nécessite d'être plus intelligent avec ses threads.

Abus de threads

- Trop de threads tue le thread.
- Coût d'un thread élevé.

```
# cat /proc/1161/maps
7fedcd9b2000-7fedcd9b3000 ---p 00000000 00:00 0
7fedcd9b3000-7fedce1b3000 rw-p 00000000 00:00 0 [stack:1499]
7fedce1b3000-7fedce1b4000 ---p 00000000 00:00 0
7fedce1b4000-7fedce9b4000 rw-p 00000000 00:00 0 [stack:1498]
7fedce9b4000-7fedce9b5000 ---p 00000000 00:00 0
7fedce9b5000-7fedcf1b5000 rw-p 00000000 00:00 0 [stack:1497]
```

→ 8Mo par thread.

- Beaucoup de bugs de synchronisation.
- Nécessite d'être plus intelligent avec ses threads.

Abus de threads

- Trop de threads tue le thread.
- Coût d'un thread élevé.

```
# cat /proc/1161/maps
7fedcd9b2000-7fedcd9b3000 ---p 00000000 00:00 0
7fedcd9b3000-7fedce1b3000 rw-p 00000000 00:00 0 [stack:1499]
7fedce1b3000-7fedce1b4000 ---p 00000000 00:00 0
7fedce1b4000-7fedce9b4000 rw-p 00000000 00:00 0 [stack:1498]
7fedce9b4000-7fedce9b5000 ---p 00000000 00:00 0
7fedce9b5000-7fedcf1b5000 rw-p 00000000 00:00 0 [stack:1497]
```

$\rightarrow$ 8Mo par thread.

- Beaucoup de bugs de synchronisation.
- Nécessite d'être plus intelligent avec ses threads.

Abus de threads

- Trop de threads tue le thread.
- Coût d'un thread élevé.

```
# cat /proc/1161/maps
7fedcd9b2000-7fedcd9b3000 ---p 00000000 00:00 0
7fedcd9b3000-7fedce1b3000 rw-p 00000000 00:00 0 [stack:1499]
7fedce1b3000-7fedce1b4000 ---p 00000000 00:00 0
7fedce1b4000-7fedce9b4000 rw-p 00000000 00:00 0 [stack:1498]
7fedce9b4000-7fedce9b5000 ---p 00000000 00:00 0
7fedce9b5000-7fedcf1b5000 rw-p 00000000 00:00 0 [stack:1497]
```

$\rightarrow$ 8Mo par thread.

- Beaucoup de bugs de synchronisation.
- Nécessite d'être plus intelligent avec ses threads.

Abus de threads

- Trop de threads tue le thread.
- Coût d'un thread élevé.

```
# cat /proc/1161/maps
7fedcd9b2000-7fedcd9b3000 ---p 00000000 00:00 0
7fedcd9b3000-7fedce1b3000 rw-p 00000000 00:00 0 [stack:1499]
7fedce1b3000-7fedce1b4000 ---p 00000000 00:00 0
7fedce1b4000-7fedce9b4000 rw-p 00000000 00:00 0 [stack:1498]
7fedce9b4000-7fedce9b5000 ---p 00000000 00:00 0
7fedce9b5000-7fedcf1b5000 rw-p 00000000 00:00 0 [stack:1497]
```

$\rightarrow$ 8Mo par thread.

- Beaucoup de bugs de synchronisation.
- Nécessite d'être plus intelligent avec ses threads.

Outline

Threadpools

- Collection de threads-consommateurs en attente de travail.
- Chaque thread est utilisé de manière optimal, sans pour autant surcharger la machine.
- Travail découpé en tâches élémentaires à exécuter.

- Nécessite d'être capable de découper son programme.
  → Pattern de plus en plus récurrente.

- Intel®Threading Building Blocks
  → GPLv2

  https://www.threadingbuildingblocks.org/

Threadpools

- Collection de threads-consommateurs en attente de travail.
- Chaque thread est utilisé de manière optimal, sans pour autant surcharger la machine.
- Travail découpé en tâches élémentaires à exécuter.
- Nécessite d'être capable de découper son programme.
  → Pattern de plus en plus récurrente.
- Intel®Threading Building Blocks
  → GPLv2

  https://www.threadingbuildingblocks.org/

Threadpools

- Collection de threads-consommateurs en attente de travail.
- Chaque thread est utilisé de manière optimal, sans pour autant surcharger la machine.
- Travail découpé en tâches élémentaires à exécuter.

- Nécessite d'être capable de découper son programme.
  → Pattern de plus en plus récurrente.

- Intel®Threading Building Blocks
  → GPLv2

  https://www.threadingbuildingblocks.org/

Threadpools

- Collection de threads-consommateurs en attente de travail.
- Chaque thread est utilisé de manière optimal, sans pour autant surcharger la machine.
- Travail découpé en tâches élémentaires à exécuter.

- Nécessite d'être capable de découper son programme.
  → Pattern de plus en plus récurrente.

- Intel®Threading Building Blocks
  → GPLv2

  https://www.threadingbuildingblocks.org/

Threadpools

- Collection de threads-consommateurs en attente de travail.
- Chaque thread est utilisé de manière optimal, sans pour autant surcharger la machine.
- Travail découpé en tâches élémentaires à exécuter.

- Nécessite d'être capable de découper son programme.
  → Pattern de plus en plus récurrente.

- Intel®Threading Building Blocks
  → GPLv2

  https://www.threadingbuildingblocks.org/

Threadpools

- Collection de threads-consommateurs en attente de travail.
- Chaque thread est utilisé de manière optimal, sans pour autant surcharger la machine.
- Travail découpé en tâches élémentaires à exécuter.

- Nécessite d'être capable de découper son programme.
  → Pattern de plus en plus récurrente.

- Intel®Threading Building Blocks
  → GPLv2

  https://www.threadingbuildingblocks.org/

Threadpools

- Collection de threads-consommateurs en attente de travail.
- Chaque thread est utilisé de manière optimal, sans pour autant surcharger la machine.
- Travail découpé en tâches élémentaires à exécuter.

- Nécessite d'être capable de découper son programme.
  → Pattern de plus en plus récurrente.

- Intel®Threading Building Blocks
  → GPLv2

https://www.threadingbuildingblocks.org/

Threadpools

- Collection de threads-consommateurs en attente de travail.
- Chaque thread est utilisé de manière optimal, sans pour autant surcharger la machine.
- Travail découpé en tâches élémentaires à exécuter.

- Nécessite d'être capable de découper son programme.
  → Pattern de plus en plus récurrente.

- Intel®Threading Building Blocks
  → GPLv2

`https://www.threadingbuildingblocks.org/`

Outline

Atomics

- L'accès concurrent aux ressources élémentaires ne nécessite pas de mutex.
  → Taille des registres du CPU, et alignement mémoire.
- Instruction CAS, CAX, ou CMPXCHG.
- "Compare and Exchange" ou "Compare and Swap".

```
int CAS(int * dest, int comp, int val) {
   int oldVal = *dest;
   if (oldVal == comp)
      *dest = val;
   return oldVal;
}
```

→ Aussi valide pour ADD, SUB, INC, DEC, etc...

- Fonctions intrinsèques des compilateurs.

Atomics

- L'accès concurrent aux ressources élémentaires ne nécessite pas de mutex.
  → Taille des registres du CPU, et alignement mémoire.
- Instruction CAS, CAX, ou CMPXCHG.
- "Compare and Exchange" ou "Compare and Swap".

```
int CAS(int * dest, int comp, int val) {
   int oldVal = *dest;
   if (oldVal == comp)
      *dest = val;
   return oldVal;
}
```

→ Aussi valide pour ADD, SUB, INC, DEC, etc...

- Fonctions intrinsèques des compilateurs.

Atomics

- L'accès concurrent aux ressources élémentaires ne nécessite pas de mutex.
  → Taille des registres du CPU, et alignement mémoire.
- Instruction CAS, CAX, ou CMPXCHG.
- "Compare and Exchange" ou "Compare and Swap".

```
int CAS(int * dest, int comp, int val) {
   int oldVal = *dest;
   if (oldVal == comp)
      *dest = val;
   return oldVal;
}
```

→ Aussi valide pour ADD, SUB, INC, DEC, etc...

- Fonctions intrinsèques des compilateurs.

Atomics

- L'accès concurrent aux ressources élémentaires ne nécessite pas de mutex.
  → Taille des registres du CPU, et alignement mémoire.
- Instruction CAS, CAX, ou CMPXCHG.
- "Compare and Exchange" ou "Compare and Swap".

```
int CAS(int * dest, int comp, int val) {
   int oldVal = *dest;
   if (oldVal == comp)
      *dest = val;
   return oldVal;
}
```

→ Aussi valide pour ADD, SUB, INC, DEC, etc...

- Fonctions intrinsèques des compilateurs.

Atomics

- L'accès concurrent aux ressources élémentaires ne nécessite pas de mutex.
  → Taille des registres du CPU, et alignement mémoire.
- Instruction CAS, CAX, ou CMPXCHG.
- "Compare and Exchange" ou "Compare and Swap".

```
int CAS(int * dest, int comp, int val) {
   int oldVal = *dest;
   if (oldVal == comp)
      *dest = val;
   return oldVal;
}
```

→ Aussi valide pour ADD, SUB, INC, DEC, etc...

- Fonctions intrinsèques des compilateurs.

Atomics

- L'accès concurrent aux ressources élémentaires ne nécessite pas de mutex.
  → Taille des registres du CPU, et alignement mémoire.
- Instruction CAS, CAX, ou CMPXCHG.
- "Compare and Exchange" ou "Compare and Swap".

```
int CAS(int * dest, int comp, int val) {
   int oldVal = *dest;
   if (oldVal == comp)
      *dest = val;
   return oldVal;
}
```

→ Aussi valide pour ADD, SUB, INC, DEC, etc...

- Fonctions intrinsèques des compilateurs.

Atomics

- L'accès concurrent aux ressources élémentaires ne nécessite pas de mutex.
  → Taille des registres du CPU, et alignement mémoire.
- Instruction CAS, CAX, ou CMPXCHG.
- "Compare and Exchange" ou "Compare and Swap".

```
int CAS(int * dest, int comp, int val) {
   int oldVal = *dest;
   if (oldVal == comp)
      *dest = val;
   return oldVal;
}
```

  → Aussi valide pour ADD, SUB, INC, DEC, etc...

- Fonctions intrinsèques des compilateurs.

Exemple concret

Example

```
struct Semaphore {
    Semaphore() : value(1) {}
    wait() {
        while (CAS(&value, 1, 0) == 0);
    }
    signal() {
        CAS(&value, 0, 1);
    }
private:
    int value;
}
```

Exemple concret

Example

```
struct Semaphore {
    Semaphore() : value(1) {}
    wait() {
        while (CAS(&value, 1, 0) == 0);
    }
    signal() {
        CAS(&value, 0, 1);
    }
private:
    int value;
}
```

Exemple concret

Example

```
struct Semaphore {
    Semaphore() : value(1) {}
    wait() {
        while (CAS(&value, 1, 0) == 0);
    }
    signal() {
        CAS(&value, 0, 1);
    }
private:
    int value;
}
```

Exemple concret

Example

```
struct Semaphore {
    Semaphore() : value(1) {}
    wait() {
        while (CAS(&value, 1, 0) == 0);
    }
    signal() {
        CAS(&value, 0, 1);
    }
private:
    int value;
}
```

Exemple concret

Example

```
struct Semaphore {
    Semaphore() : value(1) {}
    wait() {
        while (CAS(&value, 1, 0) == 0);
    }
    signal() {
        CAS(&value, 0, 1);
    }
private:
    int value;
}
```

Exemple concret

Example

```
struct Semaphore {
    Semaphore() : value(1) {}
    wait() {
        while (CAS(&value, 1, 0) == 0);
    }
    signal() {
        CAS(&value, 0, 1);
    }
private:
    int value;
}
```

Avantages, inconvéniants et applications

- Les opérations atomiques ne nécessitent pas d'arbitrage kernel.
  → Gain de performances.
- Les opérations atomiques ne peuvent pas bénéficier d'arbitrage kernel.
  → Perte de performances.
- A n'utiliser qu'en cas de contention faible.
- Algorithmes dits "lock-free".
  → Stack.
  → Queue.
  → List.
- Problème ABA.
  → Utiliser des implémentations connues.

Avantages, inconvéniants et applications

- Les opérations atomiques ne nécessitent pas d'arbitrage kernel.
  → Gain de performances.
- Les opérations atomiques ne peuvent pas bénéficier d'arbitrage kernel.
  → Perte de performances.
- A n'utiliser qu'en cas de contention faible.
- Algorithmes dits "lock-free".
  → Stack.
  → Queue.
  → List.
- Problème ABA.
  → Utiliser des implémentations connues.

Avantages, inconvéniants et applications

- Les opérations atomiques ne nécessitent pas d'arbitrage kernel.
  → Gain de performances.
- Les opérations atomiques ne peuvent pas bénéficier d'arbitrage kernel.
  → Perte de performances.
- A n'utiliser qu'en cas de contention faible.
- Algorithmes dits "lock-free".
  → Stack.
  → Queue.
  → List.
- Problème ABA.
  → Utiliser des implémentations connues.

Avantages, inconvéniants et applications

- Les opérations atomiques ne nécessitent pas d'arbitrage kernel.
  → Gain de performances.
- Les opérations atomiques ne peuvent pas bénéficier d'arbitrage kernel.
  → Perte de performances.
- A n'utiliser qu'en cas de contention faible.
- Algorithmes dits "lock-free".
  → Stack.
  → Queue.
  → List.
- Problème ABA.
  → Utiliser des implémentations connues.

Avantages, inconvéniants et applications

- Les opérations atomiques ne nécessitent pas d'arbitrage kernel.
  → Gain de performances.
- Les opérations atomiques ne peuvent pas bénéficier d'arbitrage kernel.
  → Perte de performances.
- A n'utiliser qu'en cas de contention faible.
- Algorithmes dits "lock-free".
  → Stack.
  → Queue.
  → List.
- Problème ABA.
  → Utiliser des implémentations connues.

Avantages, inconvéniants et applications

- Les opérations atomiques ne nécessitent pas d'arbitrage kernel.
  → Gain de performances.
- Les opérations atomiques ne peuvent pas bénéficier d'arbitrage kernel.
  → Perte de performances.
- A n'utiliser qu'en cas de contention faible.
- Algorithmes dits "lock-free".
  → Stack.
  → Queue.
  → List.
- Problème ABA.
  → Utiliser des implémentations connues.

Avantages, inconvéniants et applications

- Les opérations atomiques ne nécessitent pas d'arbitrage kernel.
  → Gain de performances.
- Les opérations atomiques ne peuvent pas bénéficier d'arbitrage kernel.
  → Perte de performances.
- A n'utiliser qu'en cas de contention faible.
- Algorithmes dits "lock-free".
  → Stack.
  → Queue.
  → List.
- Problème ABA.
  → Utiliser des implémentations connues.

Avantages, inconvéniants et applications

- Les opérations atomiques ne nécessitent pas d'arbitrage kernel.
  → Gain de performances.
- Les opérations atomiques ne peuvent pas bénéficier d'arbitrage kernel.
  → Perte de performances.
- A n'utiliser qu'en cas de contention faible.
- Algorithmes dits "lock-free".
  → Stack.
  → Queue.
  → List.
- Problème ABA.
  → Utiliser des implémentations connues.

Avantages, inconvéniants et applications

- Les opérations atomiques ne nécessitent pas d'arbitrage kernel.
  → Gain de performances.
- Les opérations atomiques ne peuvent pas bénéficier d'arbitrage kernel.
  → Perte de performances.
- A n'utiliser qu'en cas de contention faible.
- Algorithmes dits "lock-free".
  → Stack.
  → Queue.
  → List.
- Problème ABA.
  → Utiliser des implémentations connues.

Avantages, inconvéniants et applications

- Les opérations atomiques ne nécessitent pas d'arbitrage kernel.
  → Gain de performances.
- Les opérations atomiques ne peuvent pas bénéficier d'arbitrage kernel.
  → Perte de performances.
- A n'utiliser qu'en cas de contention faible.
- Algorithmes dits "lock-free".
  → Stack.
  → Queue.
  → List.
- Problème ABA.
  → Utiliser des implémentations connues.

Avantages, inconvéniants et applications

- Les opérations atomiques ne nécessitent pas d'arbitrage kernel.
  → Gain de performances.
- Les opérations atomiques ne peuvent pas bénéficier d'arbitrage kernel.
  → Perte de performances.
- A n'utiliser qu'en cas de contention faible.
- Algorithmes dits "lock-free".
  → Stack.
  → Queue.
  → List.
- Problème ABA.
  → Utiliser des implémentations connues.

Outline

C++11

- Le C++11, ou comment C++ devient conscient des threads.
- Le thread local storage devient facile et rapide.
- Réservé aux type PoD, et initialisés à 0.
  → `thread_local SomeClass * l_classPtr = NULL;`
- L'astuce classique des singletons devient thread-safe.

```
struct Singleton {
    Singleton * getSingleton() {
        static Singleton singleton;
        return &singleton;
    }
}
```

- `std::atomic`, `std::thread`, `std::async`, `std::future`, `std::promise`, ...
  → Présentation "atomic Weapons" par Herb Sutter.

http://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/

C++11

- Le C++11, ou comment C++ devient conscient des threads.
- Le thread local storage devient facile et rapide.
- Réservé aux type PoD, et initialisés à 0.
  → `thread_local SomeClass * l_classPtr = NULL;`
- L'astuce classique des singletons devient thread-safe.

```
struct Singleton {
    Singleton * getSingleton() {
        static Singleton singleton;
        return &singleton;
    }
}
```

- `std::atomic`, `std::thread`, `std::async`, `std::future`, `std::promise`, ...
  → Présentation "atomic Weapons" par Herb Sutter.

http://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/

C++11

- Le C++11, ou comment C++ devient conscient des threads.
- Le thread local storage devient facile et rapide.
- Réservé aux type PoD, et initialisés à 0.

  → `thread_local SomeClass * l_classPtr = NULL;`
- L'astuce classique des singletons devient thread-safe.

```
struct Singleton {
    Singleton * getSingleton() {
        static Singleton singleton;
        return &singleton;
    }
}
```

- `std::atomic`, `std::thread`, `std::async`, `std::future`, `std::promise`, ...

  → Présentation "atomic Weapons" par Herb Sutter.

http://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/

C++11

- Le C++11, ou comment C++ devient conscient des threads.
- Le thread local storage devient facile et rapide.
- Réservé aux type PoD, et initialisés à 0.
  → `thread_local SomeClass * l_classPtr = NULL;`
- L'astuce classique des singletons devient thread-safe.

```
struct Singleton {
    Singleton * getSingleton() {
        static Singleton singleton;
        return &singleton;
    }
}
```

- `std::atomic`, `std::thread`, `std::async`, `std::future`, `std::promise`, ...
  → Présentation "atomic Weapons" par Herb Sutter.

http://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/

C++11

- Le C++11, ou comment C++ devient conscient des threads.
- Le thread local storage devient facile et rapide.
- Réservé aux type PoD, et initialisés à 0.
  → `thread_local SomeClass * l_classPtr = NULL;`
- L'astuce classique des singletons devient thread-safe.

```
struct Singleton {
   Singleton * getSingleton() {
      static Singleton singleton;
      return &singleton;
   }
}
```

- `std::atomic`, `std::thread`, `std::async`, `std::future`, `std::promise`, ...
  → Présentation "atomic Weapons" par Herb Sutter.

http://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/

C++11

- Le C++11, ou comment C++ devient conscient des threads.
- Le thread local storage devient facile et rapide.
- Réservé aux type PoD, et initialisés à 0.
  → `thread_local SomeClass * l_classPtr = NULL;`
- L'astuce classique des singletons devient thread-safe.

```
struct Singleton {
    Singleton * getSingleton() {
        static Singleton singleton;
        return &singleton;
    }
}
```

- `std::atomic`, `std::thread`, `std::async`, `std::future`, `std::promise`, ...
  → Présentation "atomic Weapons" par Herb Sutter.

http://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/

C++11

- Le C++11, ou comment C++ devient conscient des threads.
- Le thread local storage devient facile et rapide.
- Réservé aux type PoD, et initialisés à 0.
  → `thread_local SomeClass * l_classPtr = NULL;`
- L'astuce classique des singletons devient thread-safe.

```
struct Singleton {
    Singleton * getSingleton() {
        static Singleton singleton;
        return &singleton;
    }
}
```

- `std::atomic`, `std::thread`, `std::async`, `std::future`, `std::promise`, ...
  → Présentation "atomic Weapons" par Herb Sutter.

http://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/

C++11

- Le C++11, ou comment C++ devient conscient des threads.
- Le thread local storage devient facile et rapide.
- Réservé aux type PoD, et initialisés à 0.
  → `thread_local SomeClass * l_classPtr = NULL;`
- L'astuce classique des singletons devient thread-safe.

```
struct Singleton {
    Singleton * getSingleton() {
        static Singleton singleton;
        return &singleton;
    }
}
```

- `std::atomic`, `std::thread`, `std::async`, `std::future`, `std::promise`, ...
  → Présentation "atomic Weapons" par Herb Sutter.

http://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/

C++11

- Le C++11, ou comment C++ devient conscient des threads.
- Le thread local storage devient facile et rapide.
- Réservé aux type PoD, et initialisés à 0.
  → `thread_local SomeClass * l_classPtr = NULL;`
- L'astuce classique des singletons devient thread-safe.

```
struct Singleton {
    Singleton * getSingleton() {
        static Singleton singleton;
        return &singleton;
    }
}
```

- `std::atomic`, `std::thread`, `std::async`, `std::future`, `std::promise`, ...
  → Présentation "atomic Weapons" par Herb Sutter.

http://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/

Outline

Réacteurs

- Un réacteur est programme qui réagit à des évènements.
  → Reactor pattern.
- Evènements classiques:
  → Socket read / write / accept / close.
  → Thread / process terminé.
  → Signaux Unix.
  → Timeouts.
  → Métronomes.
- Ne réinventez pas la roue! Librairies spécialisées:
  → libev
  → libevent
  → glib
- Découpage du programme en tâches plus simples.
  → Problématique similaire aux threadpools.

Réacteurs

- Un réacteur est programme qui réagit à des évènements.
  → Reactor pattern.
- Evènements classiques:
  → Socket read / write / accept / close.
  → Thread / process terminé.
  → Signaux Unix.
  → Timeouts.
  → Métronomes.
- Ne réinventez pas la roue! Librairies spécialisées:
  → libev
  → libevent
  → glib
- Découpage du programme en tâches plus simples.
  → Problématique similaire aux threadpools.

Réacteurs

- Un réacteur est programme qui réagit à des évènements.
  → Reactor pattern.
- Evènements classiques:
  → Socket read / write / accept / close.
  → Thread / process terminé.
  → Signaux Unix.
  → Timeouts.
  → Métronomes.
- Ne réinventez pas la roue! Librairies spécialisées:
  → libev
  → libevent
  → glib
- Découpage du programme en tâches plus simples.
  → Problématique similaire aux threadpools.

Réacteurs

- Un réacteur est programme qui réagit à des évènements.
  → Reactor pattern.
- Evènements classiques:
  → Socket read / write / accept / close.
  → Thread / process terminé.
  → Signaux Unix.
  → Timeouts.
  → Métronomes.
- Ne réinventez pas la roue! Librairies spécialisées:
  → libev
  → libevent
  → glib
- Découpage du programme en tâches plus simples.
  → Problématique similaire aux threadpools.

Réacteurs

- Un réacteur est programme qui réagit à des évènements.
  → Reactor pattern.
- Evènements classiques:
  → Socket read / write / accept / close.
  → Thread / process terminé.
  → Signaux Unix.
  → Timeouts.
  → Métronomes.
- Ne réinventez pas la roue! Librairies spécialisées:
  → libev
  → libevent
  → glib
- Découpage du programme en tâches plus simples.
  → Problématique similaire aux threadpools.

Réacteurs

- Un réacteur est programme qui réagit à des évènements.
  → Reactor pattern.
- Evènements classiques:
  → Socket read / write / accept / close.
  → Thread / process terminé.
  → Signaux Unix.
  → Timeouts.
  → Métronomes.
- Ne réinventez pas la roue! Librairies spécialisées:
  → libev
  → libevent
  → glib
- Découpage du programme en tâches plus simples.
  → Problématique similaire aux threadpools.

Réacteurs

- Un réacteur est programme qui réagit à des évènements.
  → Reactor pattern.
- Evènements classiques:
  → Socket read / write / accept / close.
  → Thread / process terminé.
  → Signaux Unix.
  → Timeouts.
  → Métronomes.
- Ne réinventez pas la roue! Librairies spécialisées:
  → libev
  → libevent
  → glib
- Découpage du programme en tâches plus simples.
  → Problématique similaire aux threadpools.

Réacteurs

- Un réacteur est programme qui réagit à des évènements.
  → Reactor pattern.
- Evènements classiques:
  → Socket read / write / accept / close.
  → Thread / process terminé.
  → Signaux Unix.
  → Timeouts.
  → Métronomes.
- Ne réinventez pas la roue! Librairies spécialisées:
  → libev
  → libevent
  → glib
- Découpage du programme en tâches plus simples.
  → Problématique similaire aux threadpools.

Réacteurs

- Un réacteur est programme qui réagit à des évènements.
  → Reactor pattern.
- Evènements classiques:
  → Socket read / write / accept / close.
  → Thread / process terminé.
  → Signaux Unix.
  → Timeouts.
  → Métronomes.
- Ne réinventez pas la roue! Librairies spécialisées:
  → libev
  → libevent
  → glib
- Découpage du programme en tâches plus simples.
  → Problématique similaire aux threadpools.

Réacteurs

- Un réacteur est programme qui réagit à des évènements.
  → Reactor pattern.
- Evènements classiques:
  → Socket read / write / accept / close.
  → Thread / process terminé.
  → Signaux Unix.
  → Timeouts.
  → Métronomes.
- Ne réinventez pas la roue! Librairies spécialisées:
  → libev
  → libevent
  → glib
- Découpage du programme en tâches plus simples.
  → Problématique similaire aux threadpools.

Réacteurs

- Un réacteur est programme qui réagit à des évènements.
  → Reactor pattern.
- Evènements classiques:
  → Socket read / write / accept / close.
  → Thread / process terminé.
  → Signaux Unix.
  → Timeouts.
  → Métronomes.
- Ne réinventez pas la roue! Librairies spécialisées:
  → libev
  → libevent
  → glib
- Découpage du programme en tâches plus simples.
  → Problématique similaire aux threadpools.

Réacteurs

- Un réacteur est programme qui réagit à des évènements.
  → Reactor pattern.
- Evènements classiques:
  → Socket read / write / accept / close.
  → Thread / process terminé.
  → Signaux Unix.
  → Timeouts.
  → Métronomes.
- Ne réinventez pas la roue! Librairies spécialisées:
  → libev
  → libevent
  → glib
- Découpage du programme en tâches plus simples.
  → Problématique similaire aux threadpools.

Réacteurs

- Un réacteur est programme qui réagit à des évènements.
  → Reactor pattern.
- Evènements classiques:
  → Socket read / write / accept / close.
  → Thread / process terminé.
  → Signaux Unix.
  → Timeouts.
  → Métronomes.
- Ne réinventez pas la roue! Librairies spécialisées:
  → libev
  → libevent
  → glib
- Découpage du programme en tâches plus simples.
  → Problématique similaire aux threadpools.

Réacteurs

- Un réacteur est programme qui réagit à des évènements.
  → Reactor pattern.
- Evènements classiques:
  → Socket read / write / accept / close.
  → Thread / process terminé.
  → Signaux Unix.
  → Timeouts.
  → Métronomes.
- Ne réinventez pas la roue! Librairies spécialisées:
  → libev
  → libevent
  → glib
- Découpage du programme en tâches plus simples.
  → Problématique similaire aux threadpools.

Outline

Co-routines

- Terme créé par Melvin Conway, décrit par Donald Knuth.
- Une co-routine est une routine qui peut s'interrompre et reprendre à tout moment.
- Aussi appelés "threads coopératifs".
- Opération primordiale: yield.
  → Transfère l'exécution à une autre co-routine.
- Très bon support natif dans beaucoup de langages, sauf C/C++.
  → Implémentation Microsoft: Fibers.
  → Bricolage Unix via signaux.
  → Support C++ via boost.
- Utilisation optimale avec un réacteur et/ou threadpool.
  → Plus besoin de découper son algorithme.

Co-routines

- Terme créé par Melvin Conway, décrit par Donald Knuth.
- Une co-routine est une routine qui peut s'interrompre et reprendre à tout moment.
- Aussi appelés "threads coopératifs".
- Opération primordiale: yield.
  → Transfère l'exécution à une autre co-routine.
- Très bon support natif dans beaucoup de langages, sauf C/C++.
  → Implémentation Microsoft: Fibers.
  → Bricolage Unix via signaux.
  → Support C++ via boost.
- Utilisation optimale avec un réacteur et/ou threadpool.
  → Plus besoin de découper son algorithme.

Co-routines

- Terme créé par Melvin Conway, décrit par Donald Knuth.
- Une co-routine est une routine qui peut s'interrompre et reprendre à tout moment.
- Aussi appelés "threads coopératifs".

- Opération primordiale: yield.
  → Transfère l'exécution à une autre co-routine.

- Très bon support natif dans beaucoup de langages, sauf C/C++.
  → Implémentation Microsoft: Fibers.
  → Bricolage Unix via signaux.
  → Support C++ via boost.

- Utilisation optimale avec un réacteur et/ou threadpool.
  → Plus besoin de découper son algorithme.

Co-routines

- Terme créé par Melvin Conway, décrit par Donald Knuth.
- Une co-routine est une routine qui peut s'interrompre et reprendre à tout moment.
- Aussi appelés "threads coopératifs".

- Opération primordiale: yield.
  → Transfère l'exécution à une autre co-routine.

- Très bon support natif dans beaucoup de langages, sauf C/C++.
  → Implémentation Microsoft: Fibers.
  → Bricolage Unix via signaux.
  → Support C++ via boost.

- Utilisation optimale avec un réacteur et/ou threadpool.
  → Plus besoin de découper son algorithme.

Co-routines

- Terme créé par Melvin Conway, décrit par Donald Knuth.
- Une co-routine est une routine qui peut s'interrompre et reprendre à tout moment.
- Aussi appelés "threads coopératifs".

- Opération primordiale: yield.
  → Transfère l'exécution à une autre co-routine.

- Très bon support natif dans beaucoup de langages, sauf C/C++.
  → Implémentation Microsoft: Fibers.
  → Bricolage Unix via signaux.
  → Support C++ via boost.

- Utilisation optimale avec un réacteur et/ou threadpool.
  → Plus besoin de découper son algorithme.

Co-routines

- Terme créé par Melvin Conway, décrit par Donald Knuth.
- Une co-routine est une routine qui peut s'interrompre et reprendre à tout moment.
- Aussi appelés "threads coopératifs".

- Opération primordiale: yield.
  → Transfère l'exécution à une autre co-routine.

- Très bon support natif dans beaucoup de langages, sauf C/C++.
  → Implémentation Microsoft: Fibers.
  → Bricolage Unix via signaux.
  → Support C++ via boost.

- Utilisation optimale avec un réacteur et/ou threadpool.
  → Plus besoin de découper son algorithme.

Co-routines

- Terme créé par Melvin Conway, décrit par Donald Knuth.
- Une co-routine est une routine qui peut s'interrompre et reprendre à tout moment.
- Aussi appelés "threads coopératifs".

- Opération primordiale: yield.
  → Transfère l'exécution à une autre co-routine.

- Très bon support natif dans beaucoup de langages, sauf C/C++.
  → Implémentation Microsoft: Fibers.
  → Bricolage Unix via signaux.
  → Support C++ via boost.

- Utilisation optimale avec un réacteur et/ou threadpool.
  → Plus besoin de découper son algorithme.

Co-routines

- Terme créé par Melvin Conway, décrit par Donald Knuth.
- Une co-routine est une routine qui peut s'interrompre et reprendre à tout moment.
- Aussi appelés "threads coopératifs".

- Opération primordiale: yield.
  → Transfère l'exécution à une autre co-routine.

- Très bon support natif dans beaucoup de langages, sauf C/C++.
  → Implémentation Microsoft: Fibers.
  → Bricolage Unix via signaux.
  → Support C++ via boost.

- Utilisation optimale avec un réacteur et/ou threadpool.
  → Plus besoin de découper son algorithme.

Co-routines

- Terme créé par Melvin Conway, décrit par Donald Knuth.
- Une co-routine est une routine qui peut s'interrompre et reprendre à tout moment.
- Aussi appelés "threads coopératifs".

- Opération primordiale: yield.
  → Transfère l'exécution à une autre co-routine.

- Très bon support natif dans beaucoup de langages, sauf C/C++.
  → Implémentation Microsoft: Fibers.
  → Bricolage Unix via signaux.
  → Support C++ via boost.

- Utilisation optimale avec un réacteur et/ou threadpool.
  → Plus besoin de découper son algorithme.

Co-routines

- Terme créé par Melvin Conway, décrit par Donald Knuth.
- Une co-routine est une routine qui peut s'interrompre et reprendre à tout moment.
- Aussi appelés "threads coopératifs".

- Opération primordiale: yield.
  → Transfère l'exécution à une autre co-routine.

- Très bon support natif dans beaucoup de langages, sauf C/C++.
  → Implémentation Microsoft: Fibers.
  → Bricolage Unix via signaux.
  → Support C++ via boost.

- Utilisation optimale avec un réacteur et/ou threadpool.
  → Plus besoin de découper son algorithme.

Co-routines

- Terme créé par Melvin Conway, décrit par Donald Knuth.
- Une co-routine est une routine qui peut s'interrompre et reprendre à tout moment.
- Aussi appelés "threads coopératifs".

- Opération primordiale: yield.
  → Transfère l'exécution à une autre co-routine.

- Très bon support natif dans beaucoup de langages, sauf C/C++.
  → Implémentation Microsoft: Fibers.
  → Bricolage Unix via signaux.
  → Support C++ via boost.

- Utilisation optimale avec un réacteur et/ou threadpool.
  → Plus besoin de découper son algorithme.

Stackless co-routines

- Co-routines du pauvre.
- Le support du kernel et/ou langage vient de la difficulté à changer de pile d'exécution.
- On peut construire une co-routine sans pile d'exécution.
  → Très similaire à une state machine.
- Utilisation de l'astuce de Duff.

```
void send(char * from, char * to, int count) {
    int n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do { *to = *from++;
    case 7:      *to = *from++;
    case 6:      *to = *from++;
                 ...
    case 1:      *to = *from++;
        } while(--n > 0);
    }
}
```

Stackless co-routines

- Co-routines du pauvre.
- Le support du kernel et/ou langage vient de la difficulté à changer de pile d'exécution.
- On peut construire une co-routine sans pile d'exécution.
  $\rightarrow$ Très similaire à une state machine.
- Utilisation de l'astuce de Duff.

```
void send(char * from, char * to, int count) {
    int n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do { *to = *from++;
    case 7:      *to = *from++;
    case 6:      *to = *from++;
                 ...
    case 1:      *to = *from++;
        } while(--n > 0);
    }
}
```

Stackless co-routines

- Co-routines du pauvre.
- Le support du kernel et/ou langage vient de la difficulté à changer de pile d'exécution.
- On peut construire une co-routine sans pile d'exécution.
  $\rightarrow$ Très similaire à une state machine.
- Utilisation de l'astuce de Duff.

```
void send(char * from, char * to, int count) {
    int n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do { *to = *from++;
    case 7:      *to = *from++;
    case 6:      *to = *from++;
                 ...
    case 1:      *to = *from++;
        } while(--n > 0);
    }
}
```

Stackless co-routines

- Co-routines du pauvre.
- Le support du kernel et/ou langage vient de la difficulté à changer de pile d'exécution.
- On peut construire une co-routine sans pile d'exécution.
  $\rightarrow$ Très similaire à une state machine.
- Utilisation de l'astuce de Duff.

```
void send(char * from, char * to, int count) {
    int n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do { *to = *from++;
    case 7:      *to = *from++;
    case 6:      *to = *from++;
                 ...
    case 1:      *to = *from++;
        } while(--n > 0);
    }
}
```

Stackless co-routines

- Co-routines du pauvre.
- Le support du kernel et/ou langage vient de la difficulté à changer de pile d'exécution.
- On peut construire une co-routine sans pile d'exécution.
  $\rightarrow$ Très similaire à une state machine.
- Utilisation de l'astuce de Duff.

```
void send(char * from, char * to, int count) {
    int n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do { *to = *from++;
    case 7:      *to = *from++;
    case 6:      *to = *from++;
                ...
    case 1:      *to = *from++;
        } while(--n > 0);
    }
}
```

Stackless co-routines

- Co-routines du pauvre.
- Le support du kernel et/ou langage vient de la difficulté à changer de pile d'exécution.
- On peut construire une co-routine sans pile d'exécution.
  $\rightarrow$ Très similaire à une state machine.
- Utilisation de l'astuce de Duff.

```
void send(char * from, char * to, int count) {
    int n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do { *to = *from++;
    case 7:      *to = *from++;
    case 6:      *to = *from++;
            ...
    case 1:      *to = *from++;
        } while(--n > 0);
    }
}
```

Exemple de co-routine

Example

```
void rleDecomp(stream in, stream out) {
    rleCommand command;
    int length;
    char c;
    while (!in.eof()) {
        std::tie(command, length) = in.readCommand();
        switch (command) {
        case COPY:
            while (length--) {
                c = in.readChar();
                out.writeChar(c)
            }
            break;
        case REPEAT:
            c = in.readChar();
            while (length--) {
                out.writeChar(c);
            }
            break;
        }
    }
}
```

Exemple de co-routine

Example

```
void rleDecomp(stream in, stream out) {
    rleCommand command;
    int length;
    char c;
    while (!in.eof()) {
        std::tie(command, length) = in.readCommand();
        switch (command) {
        case COPY:
            while (length--) {
                c = in.readChar();
                out.writeChar(c)
            }
            break;
        case REPEAT:
            c = in.readChar();
            while (length--) {
                out.writeChar(c);
            }
            break;
        }
    }
}
```

Exemple de co-routine

Example

```
void rleDecomp(stream in, stream out) {
    rleCommand command;
    int length;
    char c;
    while (!in.eof()) {
        std::tie(command, length) = in.readCommand();
        switch (command) {
        case COPY:
            while (length--) {
                c = in.readChar();
                out.writeChar(c)
            }
            break;
        case REPEAT:
            c = in.readChar();
            while (length--) {
                out.writeChar(c);
            }
            break;
        }
    }
}
```

Exemple de co-routine

Example

```
void rleDecomp(stream in, stream out) {
    rleCommand command;
    int length;
    char c;
    while (!in.eof()) {
        std::tie(command, length) = in.readCommand();
        switch (command) {
        case COPY:
            while (length--) {
                c = in.readChar();
                out.writeChar(c)
            }
            break;
        case REPEAT:
            c = in.readChar();
            while (length--) {
                out.writeChar(c);
            }
            break;
        }
    }
}
```

Exemple de co-routine

Example

```
void rleDecomp(stream in, stream out) {
    rleCommand command;
    int length;
    char c;
    while (!in.eof()) {
        std::tie(command, length) = in.readCommand();
        switch (command) {
        case COPY:
            while (length--) {
                c = in.readChar();
                out.writeChar(c)
            }
            break;
        case REPEAT:
            c = in.readChar();
            while (length--) {
                out.writeChar(c);
            }
            break;
        }
    }
}
```

Exemple de co-routine

Example

```
void rleDecomp(stream in, stream out) {
    rleCommand command;
    int length;
    char c;
    while (!in.eof()) {
        std::tie(command, length) = in.readCommand();
        switch (command) {
        case COPY:
            while (length--) {
                c = in.readChar();
                out.writeChar(c)
            }
            break;
        case REPEAT:
            c = in.readChar();
            while (length--) {
                out.writeChar(c);
            }
            break;
        }
    }
}
```

```
void stackless::rleDecomp() {
    switch (m_state) {
    case STATE_START:
        while (!m_in.eof()) {
            m_state = STATE_READ_CMD;
    case STATE_READ_CMD:
            std::tie(m_command, m_length) = m_in.readCommand();
            switch (m_command) {
            case COPY:
                while (m_length--) {
                    m_state = STATE_COPY_READ_CHAR;
    case STATE_COPY_READ_CHAR:
                    m_c = m_in.readChar();
                    m_state = STATE_COPY_WRITE_CHAR;
    case STATE_COPY_WRITE_CHAR:
                    m_out.writeChar(m_c)
                }
                break;
            case REPEAT:
                m_state = STATE_REPEAT_READ_CHAR;
    case STATE_REPEAT_READ_CHAR:
                m_c = m_in.readChar();
                while (length--) {
                    m_state = STATE_REPEAT_WRITE_CHAR;
    case STATE_REPEAT_WRITE_CHAR:
                    m_out.writeChar(m_c);
                }
                break;
            }
        }
    }
    m_state = STATE_DONE;
}
```

```
void stackless::rleDecomp() {
    switch (m_state) {
    case STATE_START:
        while (!m_in.eof()) {
            m_state = STATE_READ_CMD;
    case STATE_READ_CMD:
            std::tie(m_command, m_length) = m_in.readCommand();
            switch (m_command) {
            case COPY:
                while (m_length--) {
                    m_state = STATE_COPY_READ_CHAR;
    case STATE_COPY_READ_CHAR:
                    m_c = m_in.readChar();
                    m_state = STATE_COPY_WRITE_CHAR;
    case STATE_COPY_WRITE_CHAR:
                    m_out.writeChar(m_c)
                }
                break;
            case REPEAT:
                m_state = STATE_REPEAT_READ_CHAR;
    case STATE_REPEAT_READ_CHAR:
                m_c = m_in.readChar();
                while (length--) {
                    m_state = STATE_REPEAT_WRITE_CHAR;
    case STATE_REPEAT_WRITE_CHAR:
                    m_out.writeChar(m_c);
                }
                break;
            }
        }
    }
    m_state = STATE_DONE;
}
```

```
void stackless::rleDecomp() {
    switch (m_state) {
    case STATE_START:
        while (!m_in.eof()) {
            m_state = STATE_READ_CMD;
    case STATE_READ_CMD:
            std::tie(m_command, m_length) = m_in.readCommand();
            switch (m_command) {
            case COPY:
                while (m_length--) {
                    m_state = STATE_COPY_READ_CHAR;
    case STATE_COPY_READ_CHAR:
                    m_c = m_in.readChar();
                    m_state = STATE_COPY_WRITE_CHAR;
    case STATE_COPY_WRITE_CHAR:
                    m_out.writeChar(m_c)
                }
                break;
            case REPEAT:
                m_state = STATE_REPEAT_READ_CHAR;
    case STATE_REPEAT_READ_CHAR:
                m_c = m_in.readChar();
                while (length--) {
                    m_state = STATE_REPEAT_WRITE_CHAR;
    case STATE_REPEAT_WRITE_CHAR:
                    m_out.writeChar(m_c);
                }
                break;
            }
        }
    }
    m_state = STATE_DONE;
}
```

```
void stackless::rleDecomp() {
    switch (m_state) {
    case STATE_START:
        while (!m_in.eof()) {
            m_state = STATE_READ_CMD;
    case STATE_READ_CMD:
            std::tie(m_command, m_length) = m_in.readCommand();
            switch (m_command) {
            case COPY:
                while (m_length--) {
                    m_state = STATE_COPY_READ_CHAR;
    case STATE_COPY_READ_CHAR:
                    m_c = m_in.readChar();
                    m_state = STATE_COPY_WRITE_CHAR;
    case STATE_COPY_WRITE_CHAR:
                    m_out.writeChar(m_c)
                }
                break;
            case REPEAT:
                m_state = STATE_REPEAT_READ_CHAR;
    case STATE_REPEAT_READ_CHAR:
                m_c = m_in.readChar();
                while (length--) {
                    m_state = STATE_REPEAT_WRITE_CHAR;
    case STATE_REPEAT_WRITE_CHAR:
                    m_out.writeChar(m_c);
                }
                break;
            }
        }
    }
    m_state = STATE_DONE;
}
```

```
void stackless::rleDecomp() {
    switch (m_state) {
    case STATE_START:
        while (!m_in.eof()) {
            m_state = STATE_READ_CMD;
    case STATE_READ_CMD:
            std::tie(m_command, m_length) = m_in.readCommand();
            switch (m_command) {
            case COPY:
                while (m_length--) {
                    m_state = STATE_COPY_READ_CHAR;
    case STATE_COPY_READ_CHAR:
                    m_c = m_in.readChar();
                    m_state = STATE_COPY_WRITE_CHAR;
    case STATE_COPY_WRITE_CHAR:
                    m_out.writeChar(m_c)
                }
                break;
            case REPEAT:
                m_state = STATE_REPEAT_READ_CHAR;
    case STATE_REPEAT_READ_CHAR:
                m_c = m_in.readChar();
                while (length--) {
                    m_state = STATE_REPEAT_WRITE_CHAR;
    case STATE_REPEAT_WRITE_CHAR:
                    m_out.writeChar(m_c);
                }
                break;
            }
        }
    }
    m_state = STATE_DONE;
}
```

```
void stackless::rleDecomp() {
    switch (m_state) {
    case STATE_START:
        while (!m_in.eof()) {
            m_state = STATE_READ_CMD;
    case STATE_READ_CMD:
            std::tie(m_command, m_length) = m_in.readCommand();
            switch (m_command) {
            case COPY:
                while (m_length--) {
                    m_state = STATE_COPY_READ_CHAR;
    case STATE_COPY_READ_CHAR:
                    m_c = m_in.readChar();
                    m_state = STATE_COPY_WRITE_CHAR;
    case STATE_COPY_WRITE_CHAR:
                    m_out.writeChar(m_c)
                }
                break;
            case REPEAT:
                m_state = STATE_REPEAT_READ_CHAR;
    case STATE_REPEAT_READ_CHAR:
                m_c = m_in.readChar();
                while (length--) {
                    m_state = STATE_REPEAT_WRITE_CHAR;
    case STATE_REPEAT_WRITE_CHAR:
                    m_out.writeChar(m_c);
                }
                break;
            }
        }
    }
    m_state = STATE_DONE;
}
```

```
void stackless::rleDecomp() {
    switch (m_state) {
    case STATE_START:
        while (!m_in.eof()) {
            m_state = STATE_READ_CMD;
    case STATE_READ_CMD:
            std::tie(m_command, m_length) = m_in.readCommand();
            switch (m_command) {
            case COPY:
                while (m_length--) {
                    m_state = STATE_COPY_READ_CHAR;
    case STATE_COPY_READ_CHAR:
                    m_c = m_in.readChar();
                    m_state = STATE_COPY_WRITE_CHAR;
    case STATE_COPY_WRITE_CHAR:
                    m_out.writeChar(m_c)
                }
                break;
            case REPEAT:
                m_state = STATE_REPEAT_READ_CHAR;
    case STATE_REPEAT_READ_CHAR:
                m_c = m_in.readChar();
                while (length--) {
                    m_state = STATE_REPEAT_WRITE_CHAR;
    case STATE_REPEAT_WRITE_CHAR:
                    m_out.writeChar(m_c);
                }
                break;
            }
        }
    }
    m_state = STATE_DONE;
}
```

```
void stackless::rleDecomp() {
    switch (m_state) {
    case STATE_START:
        while (!m_in.eof()) {
            m_state = STATE_READ_CMD;
    case STATE_READ_CMD:
            std::tie(m_command, m_length) = m_in.readCommand();
            switch (m_command) {
            case COPY:
                while (m_length--) {
                    m_state = STATE_COPY_READ_CHAR;
    case STATE_COPY_READ_CHAR:
                    m_c = m_in.readChar();
                    m_state = STATE_COPY_WRITE_CHAR;
    case STATE_COPY_WRITE_CHAR:
                    m_out.writeChar(m_c)
                }
                break;
            case REPEAT:
                m_state = STATE_REPEAT_READ_CHAR;
    case STATE_REPEAT_READ_CHAR:
                m_c = m_in.readChar();
                while (length--) {
                    m_state = STATE_REPEAT_WRITE_CHAR;
    case STATE_REPEAT_WRITE_CHAR:
                    m_out.writeChar(m_c);
                }
                break;
            }
        }
    }
    m_state = STATE_DONE;
}
```

```
void stackless::rleDecomp() {
    switch (m_state) {
    case STATE_START:
        while (!m_in.eof()) {
            m_state = STATE_READ_CMD;
    case STATE_READ_CMD:
            std::tie(m_command, m_length) = m_in.readCommand();
            switch (m_command) {
            case COPY:
                while (m_length--) {
                    m_state = STATE_COPY_READ_CHAR;
    case STATE_COPY_READ_CHAR:
                    m_c = m_in.readChar();
                    m_state = STATE_COPY_WRITE_CHAR;
    case STATE_COPY_WRITE_CHAR:
                    m_out.writeChar(m_c)
                }
                break;
            case REPEAT:
                m_state = STATE_REPEAT_READ_CHAR;
    case STATE_REPEAT_READ_CHAR:
                m_c = m_in.readChar();
                while (length--) {
                    m_state = STATE_REPEAT_WRITE_CHAR;
    case STATE_REPEAT_WRITE_CHAR:
                    m_out.writeChar(m_c);
                }
                break;
            }
        }
    }
    m_state = STATE_DONE;
}
```

Outline

MapReduce

- Algorithme Divide and Conquer.
- Brevet déposé par Google.
- Technique de parallèlisation sur un cluster de machines.

- MapReduce fonctionne avec deux fonctions:
  $Map(k_1, v_1) \rightarrow list(k_2, v_2)$
  $Reduce(k_2, list(v_2)) \rightarrow list(v_3)$

- Map est appelé en parallèle sur toutes les clefs du problème d'entrée.
- Une fois terminé, le framework regroupe toutes les mêmes clefs produites ensembles.
- Reduce est appelé en parallèle sur toutes les clefs intermédiaires.
- Le résultat est l'ensemble $\{k_2, v_3\}$

MapReduce

- Algorithme Divide and Conquer.
- Brevet déposé par Google.
- Technique de parallèlisation sur un cluster de machines.
- MapReduce fonctionne avec deux fonctions:
  $Map(k_1, v_1) \rightarrow list(k_2, v_2)$
  $Reduce(k_2, list(v_2)) \rightarrow list(v_3)$
- Map est appelé en parallèle sur toutes les clefs du problème d'entrée.
- Une fois terminé, le framework regroupe toutes les mêmes clefs produites ensembles.
- Reduce est appelé en parallèle sur toutes les clefs intermédiaires.
- Le résultat est l'ensemble $\{k_2, v_3\}$

MapReduce

- Algorithme Divide and Conquer.
- Brevet déposé par Google.
- Technique de parallèlisation sur un cluster de machines.

- MapReduce fonctionne avec deux fonctions:
  $Map(k_1, v_1) \rightarrow list(k_2, v_2)$
  $Reduce(k_2, list(v_2)) \rightarrow list(v_3)$
- Map est appelé en parallèle sur toutes les clefs du problème d'entrée.
- Une fois terminé, le framework regroupe toutes les mêmes clefs produites ensembles.
- Reduce est appelé en parallèle sur toutes les clefs intermédiaires.
- Le résultat est l'ensemble $\{k_2, v_3\}$

MapReduce

- Algorithme Divide and Conquer.
- Brevet déposé par Google.
- Technique de parallèlisation sur un cluster de machines.

- MapReduce fonctionne avec deux fonctions:
  $Map(k_1, v_1) \rightarrow list(k_2, v_2)$
  $Reduce(k_2, list(v_2)) \rightarrow list(v_3)$

- Map est appelé en parallèle sur toutes les clefs du problème d'entrée.
- Une fois terminé, le framework regroupe toutes les mêmes clefs produites ensembles.
- Reduce est appelé en parallèle sur toutes les clefs intermédiaires.
- Le résultat est l'ensemble $\{k_2, v_3\}$

MapReduce

- Algorithme Divide and Conquer.
- Brevet déposé par Google.
- Technique de parallèlisation sur un cluster de machines.

- MapReduce fonctionne avec deux fonctions:
  $Map(k_1, v_1) \rightarrow list(k_2, v_2)$
  $Reduce(k_2, list(v_2)) \rightarrow list(v_3)$

- Map est appelé en parallèle sur toutes les clefs du problème d'entrée.
- Une fois terminé, le framework regroupe toutes les mêmes clefs produites ensembles.
- Reduce est appelé en parallèle sur toutes les clefs intermédiaires.
- Le résultat est l'ensemble $\{k_2, v_3\}$

MapReduce

- Algorithme Divide and Conquer.
- Brevet déposé par Google.
- Technique de parallèlisation sur un cluster de machines.

- MapReduce fonctionne avec deux fonctions:
  $Map(k_1, v_1) \rightarrow list(k_2, v_2)$
  $Reduce(k_2, list(v_2)) \rightarrow list(v_3)$

- Map est appelé en parallèle sur toutes les clefs du problème d'entrée.
- Une fois terminé, le framework regroupe toutes les mêmes clefs produites ensembles.
- Reduce est appelé en parallèle sur toutes les clefs intermédiaires.
- Le résultat est l'ensemble $\{k_2, v_3\}$

MapReduce

- Algorithme Divide and Conquer.
- Brevet déposé par Google.
- Technique de parallèlisation sur un cluster de machines.

- MapReduce fonctionne avec deux fonctions:
  $Map(k_1, v_1) \rightarrow list(k_2, v_2)$
  $Reduce(k_2, list(v_2)) \rightarrow list(v_3)$

- Map est appelé en parallèle sur toutes les clefs du problème d'entrée.
- Une fois terminé, le framework regroupe toutes les mêmes clefs produites ensembles.
- Reduce est appelé en parallèle sur toutes les clefs intermédiaires.
- Le résultat est l'ensemble $\{k_2, v_3\}$

MapReduce

- Algorithme Divide and Conquer.
- Brevet déposé par Google.
- Technique de parallèlisation sur un cluster de machines.

- MapReduce fonctionne avec deux fonctions:
  $Map(k_1, v_1) \rightarrow list(k_2, v_2)$
  $Reduce(k_2, list(v_2)) \rightarrow list(v_3)$

- Map est appelé en parallèle sur toutes les clefs du problème d'entrée.
- Une fois terminé, le framework regroupe toutes les mêmes clefs produites ensembles.
- Reduce est appelé en parallèle sur toutes les clefs intermédiaires.
- Le résultat est l'ensemble $\{k_2, v_3\}$

MapReduce

- Algorithme Divide and Conquer.
- Brevet déposé par Google.
- Technique de parallèlisation sur un cluster de machines.

- MapReduce fonctionne avec deux fonctions:
  $Map(k_1, v_1) \rightarrow list(k_2, v_2)$
  $Reduce(k_2, list(v_2)) \rightarrow list(v_3)$

- Map est appelé en parallèle sur toutes les clefs du problème d'entrée.
- Une fois terminé, le framework regroupe toutes les mêmes clefs produites ensembles.
- Reduce est appelé en parallèle sur toutes les clefs intermédiaires.
- Le résultat est l'ensemble $\{k_2, v_3\}$

MapReduce

- Algorithme Divide and Conquer.
- Brevet déposé par Google.
- Technique de parallèlisation sur un cluster de machines.

- MapReduce fonctionne avec deux fonctions:
  $Map(k_1, v_1) \rightarrow list(k_2, v_2)$
  $Reduce(k_2, list(v_2)) \rightarrow list(v_3)$

- Map est appelé en parallèle sur toutes les clefs du problème d'entrée.
- Une fois terminé, le framework regroupe toutes les mêmes clefs produites ensembles.
- Reduce est appelé en parallèle sur toutes les clefs intermédiaires.
- Le résultat est l'ensemble $\{k_2, v_3\}$

Conclusion

- L'informatique a évoluée radicalement depuis sa conception.
  → Continue d'évoluer constamment.
- Le Mur de la loi de Moore oblige à penser différemment.
- La solution matérielle réside dans la multiplication des cores, CPUs et machines.
- Nécessaire de s'adapter à des techniques de programmation multithread, multiprocess, et multimachines.
- Souvent, la "bonne" solution est un hybride des techniques disponibles.
- La veille technologique sur ces sujets est impérative.
  → Pointeurs durant la présentation à lire et étudier.

Conclusion

- L'informatique a évoluée radicalement depuis sa conception.
  → Continue d'évoluer constamment.
- Le Mur de la loi de Moore oblige à penser différemment.
- La solution matérielle réside dans la multiplication des cores, CPUs et machines.
- Nécessaire de s'adapter à des techniques de programmation multithread, multiprocess, et multimachines.
- Souvent, la "bonne" solution est un hybride des techniques disponibles.
- La veille technologique sur ces sujets est impérative.
  → Pointeurs durant la présentation à lire et étudier.

Conclusion

- L'informatique a évoluée radicalement depuis sa conception.
  → Continue d'évoluer constamment.
- Le Mur de la loi de Moore oblige à penser différemment.
- La solution matérielle réside dans la multiplication des cores, CPUs et machines.
- Nécessaire de s'adapter à des techniques de programmation multithread, multiprocess, et multimachines.
- Souvent, la "bonne" solution est un hybride des techniques disponibles.
- La veille technologique sur ces sujets est impérative.
  → Pointeurs durant la présentation à lire et étudier.

Conclusion

- L'informatique a évoluée radicalement depuis sa conception.
  → Continue d'évoluer constamment.

- Le Mur de la loi de Moore oblige à penser différemment.
- La solution matérielle réside dans la multiplication des cores, CPUs et machines.
- Nécessaire de s'adapter à des techniques de programmation multithread, multiprocess, et multimachines.
- Souvent, la "bonne" solution est un hybride des techniques disponibles.

- La veille technologique sur ces sujets est impérative.
  → Pointeurs durant la présentation à lire et étudier.

Conclusion

- L'informatique a évoluée radicalement depuis sa conception.
  → Continue d'évoluer constamment.

- Le Mur de la loi de Moore oblige à penser différemment.
- La solution matérielle réside dans la multiplication des cores, CPUs et machines.
- Nécessaire de s'adapter à des techniques de programmation multithread, multiprocess, et multimachines.
- Souvent, la "bonne" solution est un hybride des techniques disponibles.

- La veille technologique sur ces sujets est impérative.
  → Pointeurs durant la présentation à lire et étudier.

Conclusion

- L'informatique a évoluée radicalement depuis sa conception.
  → Continue d'évoluer constamment.

- Le Mur de la loi de Moore oblige à penser différemment.
- La solution matérielle réside dans la multiplication des cores, CPUs et machines.
- Nécessaire de s'adapter à des techniques de programmation multithread, multiprocess, et multimachines.
- Souvent, la "bonne" solution est un hybride des techniques disponibles.

- La veille technologique sur ces sujets est impérative.
  → Pointeurs durant la présentation à lire et étudier.

Conclusion

- L'informatique a évoluée radicalement depuis sa conception.
  → Continue d'évoluer constamment.

- Le Mur de la loi de Moore oblige à penser différemment.
- La solution matérielle réside dans la multiplication des cores, CPUs et machines.
- Nécessaire de s'adapter à des techniques de programmation multithread, multiprocess, et multimachines.
- Souvent, la "bonne" solution est un hybride des techniques disponibles.

- La veille technologique sur ces sujets est impérative.
  → Pointeurs durant la présentation à lire et étudier.

Conclusion

- L'informatique a évoluée radicalement depuis sa conception.
  → Continue d'évoluer constamment.

- Le Mur de la loi de Moore oblige à penser différemment.
- La solution matérielle réside dans la multiplication des cores, CPUs et machines.
- Nécessaire de s'adapter à des techniques de programmation multithread, multiprocess, et multimachines.
- Souvent, la "bonne" solution est un hybride des techniques disponibles.

- La veille technologique sur ces sujets est impérative.
  → Pointeurs durant la présentation à lire et étudier.