

Programmation Système Répartie et Concurrente

Master 1 Informatique – MU4IN400

Cours 4 et 5 : Thread, Atomic, Mutex, Condition

Yann Thierry-Mieg
Yann.Thierry-Mieg@lip6.fr

Plan

On a vu au cours précédent

- La lib standard : conteneurs, algorithmes

Aujourd'hui : Thread, Mutex et Atomic

- Programmation Concurrente : principes et problèmes
- Création et fin de thread
- Atomic
- Mutex
- La lib pthread

Références : cppreference.com , cplusplus.com

Cours de Edwin Carlinet (EPITA)

- <https://www.lrde.epita.fr/~carlinet/cours/parallel/>

Cours de P.Sens, L.Arantes (pthread), Cours de F. Kordon

StackOverflow : Definitive C++ book guide

<https://stackoverflow.com/questions/388242/the-definitive-c-book-guide-and-list?rq=1>

Parallélisme

Evolutions matérielles

Loi de Moore

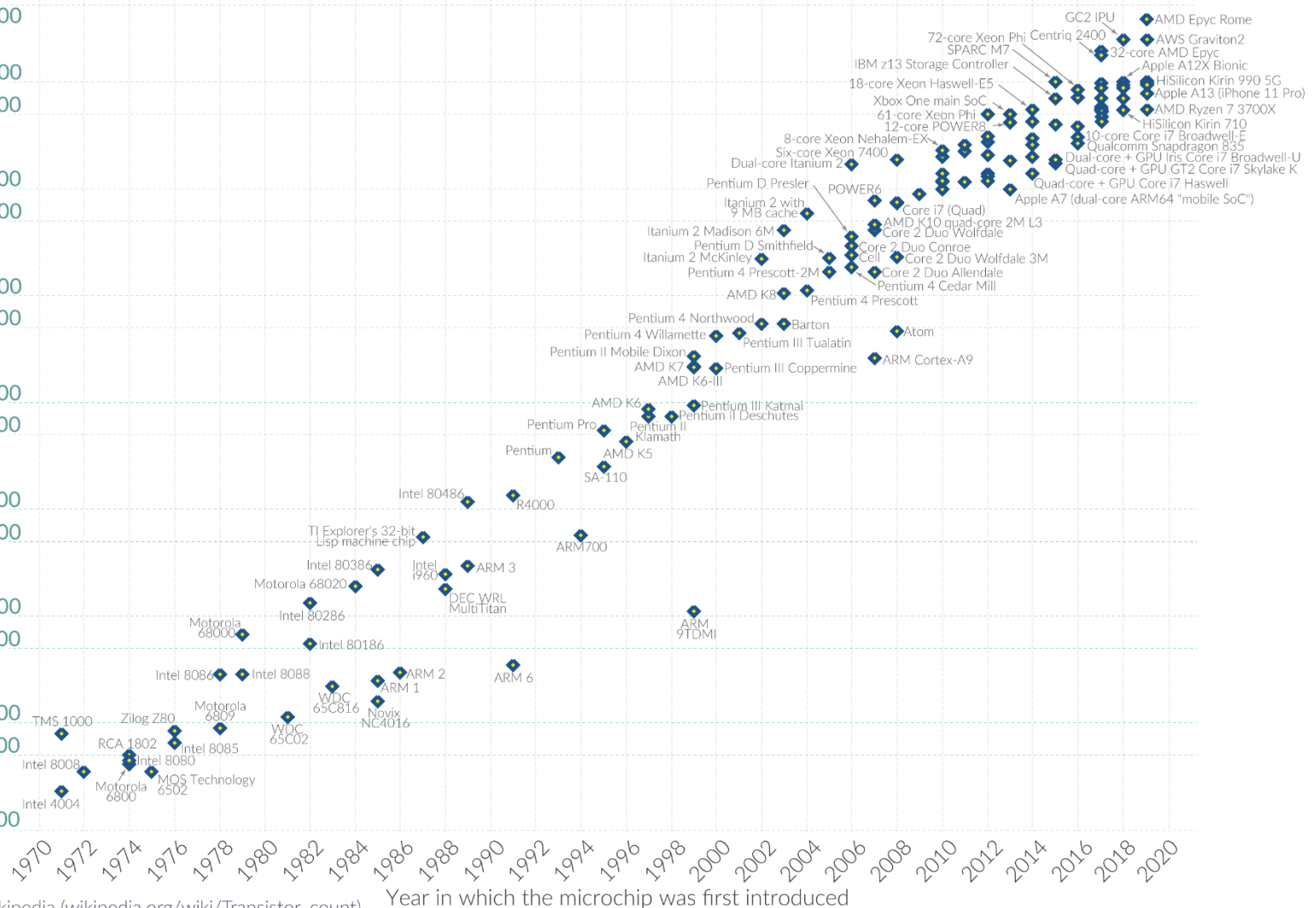
Moore's Law: The number of transistors on microchips doubles every two years

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years.

This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

Our World
in Data

Transistor count

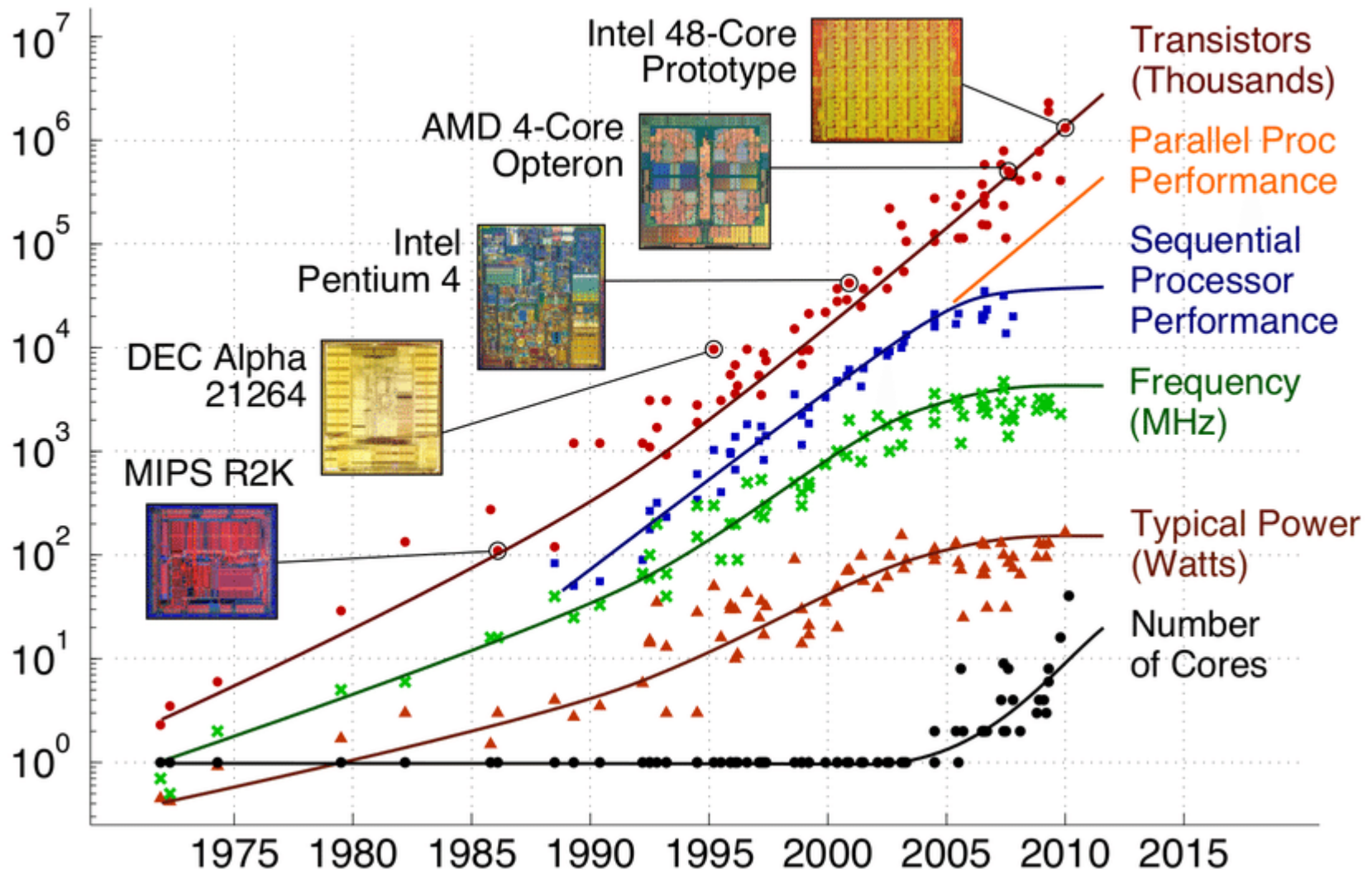


Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

OurWorldinData.org – Research and data to make progress against the world's largest problems.

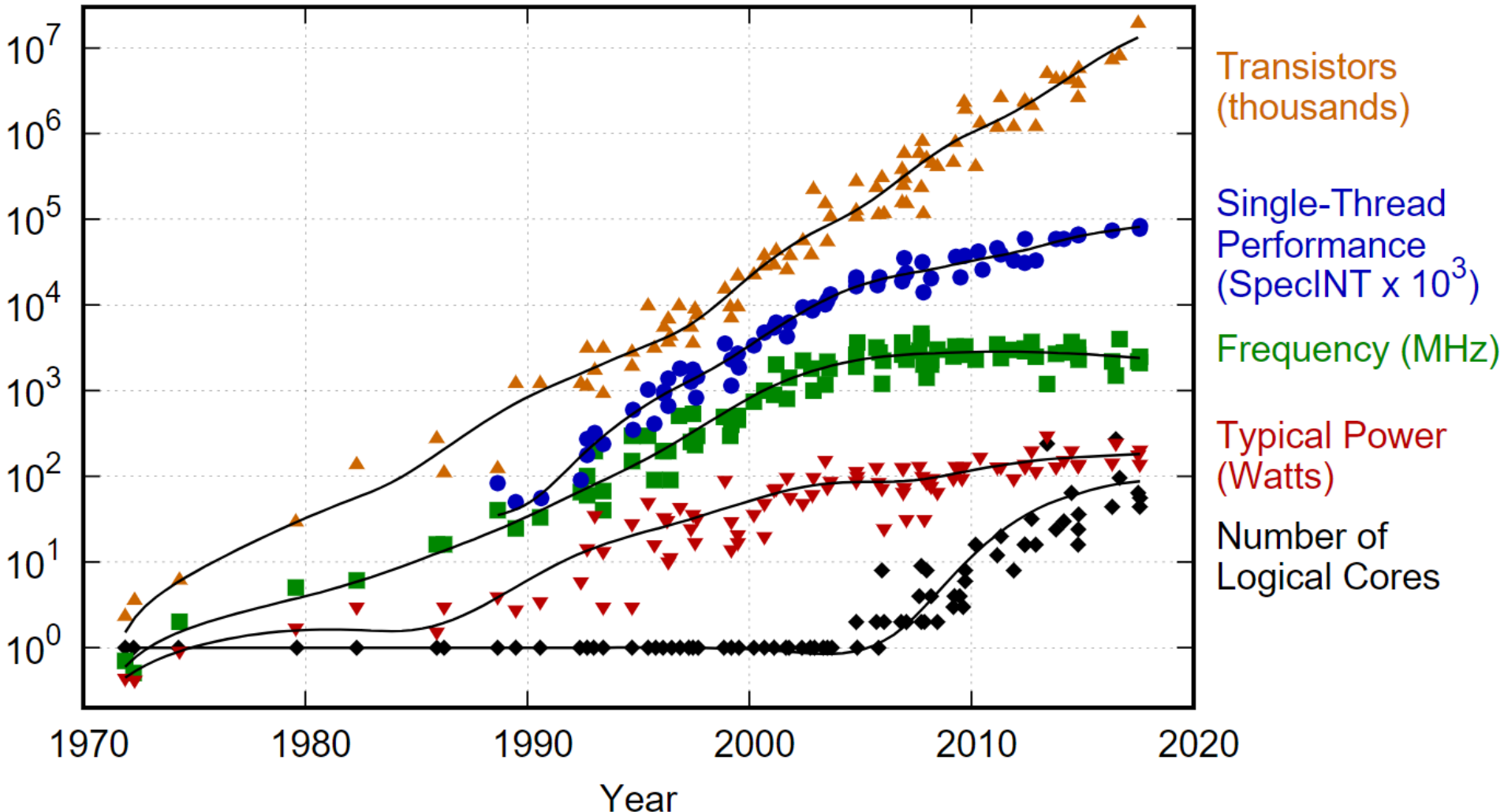
Licensed under [CC-BY](#) by the authors Hannah Ritchie and Max Roser.

Evolutions Matérielles



La tendance tient encore

42 Years of Microprocessor Trend Data



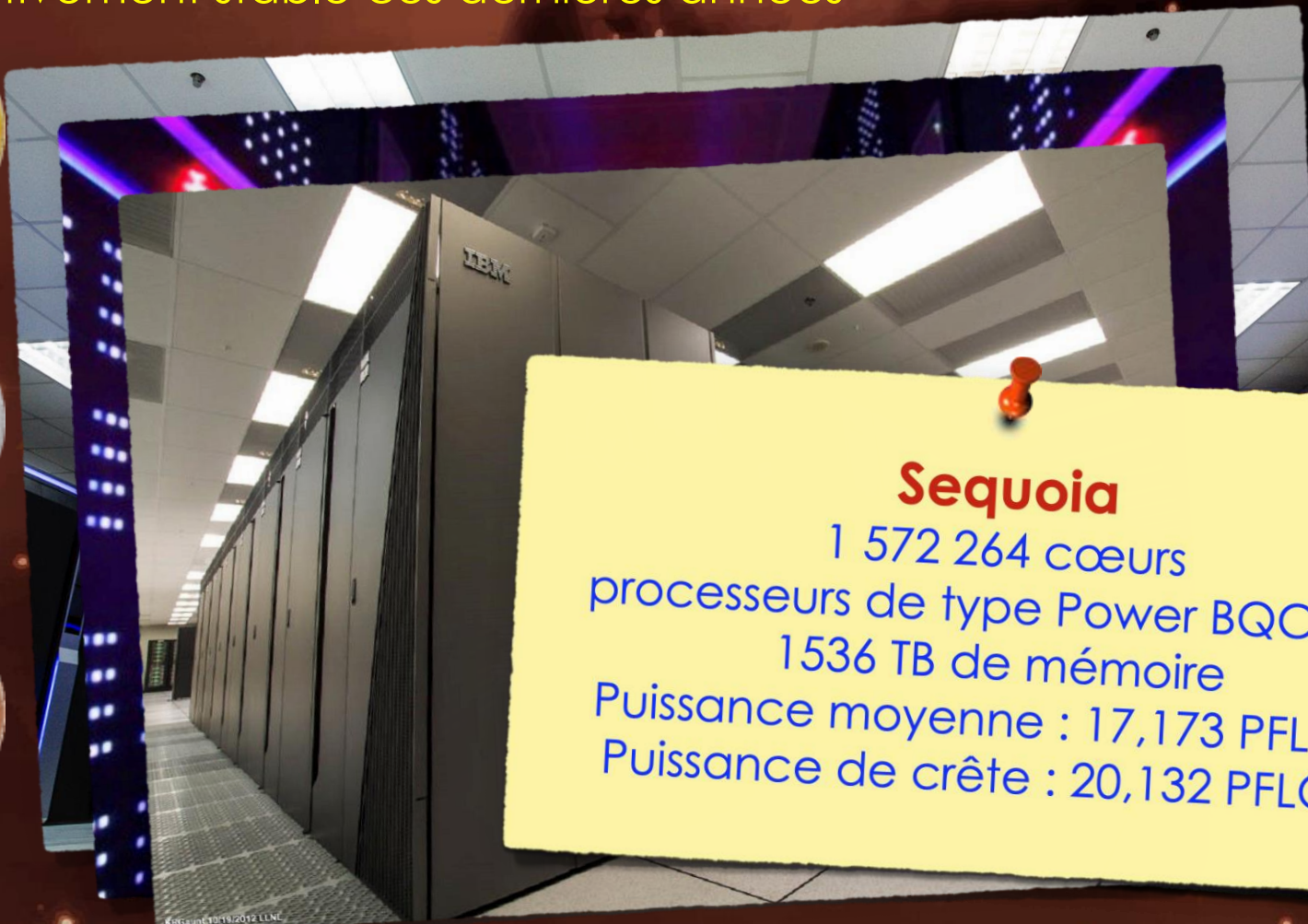
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Les plus gros ordinateurs sont parallèles (depuis belle lurette 🕒)

7

🏆 **La liste de Juin 2016 (<https://www.top500.org>)**

• Relativement stable ces dernières années



Sequoia

1 572 264 cœurs

processeurs de type Power BQC 16C

1536 TB de mémoire

Puissance moyenne : 17,173 PFLOPS

Puissance de crête : 20,132 PFLOPS

En 2024 : 2 machines en ExaFLOPS

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	Frontier - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,206.00	1,714.81	22,786
2	Aurora - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
3	Eagle - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	
4	Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899

Machines grand public

- Smartphones : 4 voire 8 core
- PC
 - intel i7 : 8 core performance + 8 core efficiency = 16 core
 - intel i9 : jusqu'à 24 core performance
- AMD Ryzen : 12 core
- Apple M2 : jusqu'à 16 core performance, + 8 core efficiency = 24 core
- ...

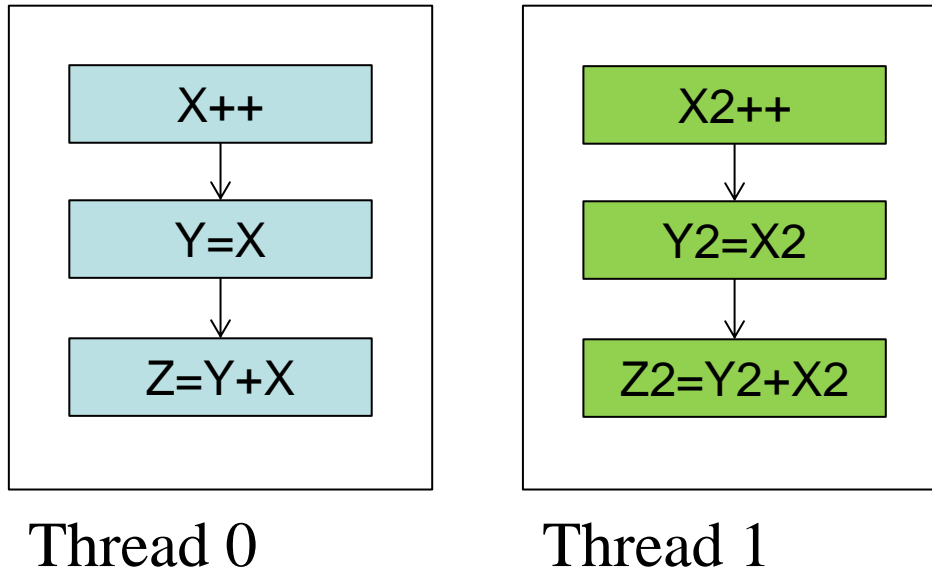
Et les cartes vidéo :

- Carte graphique RTX 4090 : 16384 core CUDA + des unités spécialisées en calcul tensoriel (IA)
- ...
- Pour exploiter cette puissance, il faut des programmes concurrents⁹

Concurrence

Principes de la concurrence : Causalité

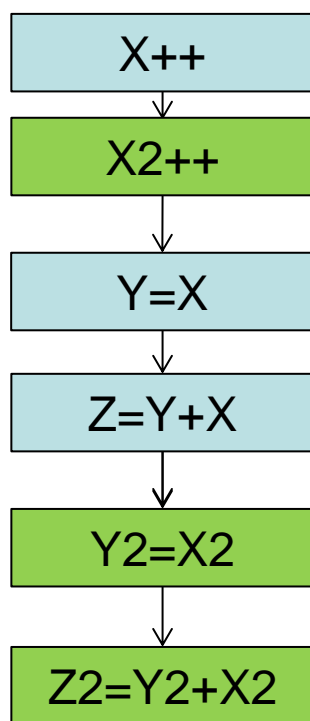
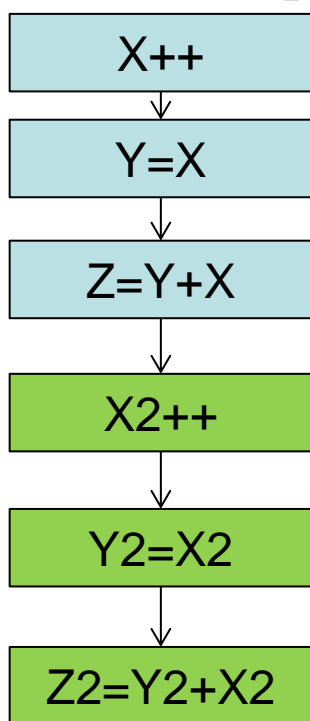
- Le traitement à réaliser est vu comme un arbre d'actions
 - Certaines actions ont des prédécesseurs
 - On parle de dépendance causale



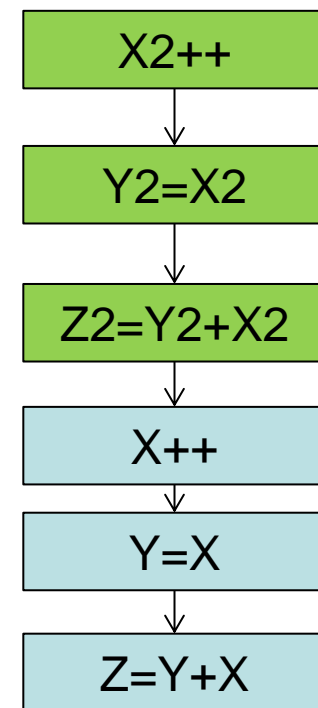
- Les actions de T0 et T1 ne sont pas ordonnées/causalement liées les unes par rapport aux autres

Causalité, Ordre Partiel

- Observation séquentielle d'un ordre partiel
 - Si l'on n'avait qu'un seul processeur + changement de contexte pour connecter la mémoire
 - Un run qui contient toutes les actions et respecte leurs précédences
 - Sémantique « consistance séquentielle » intuitive

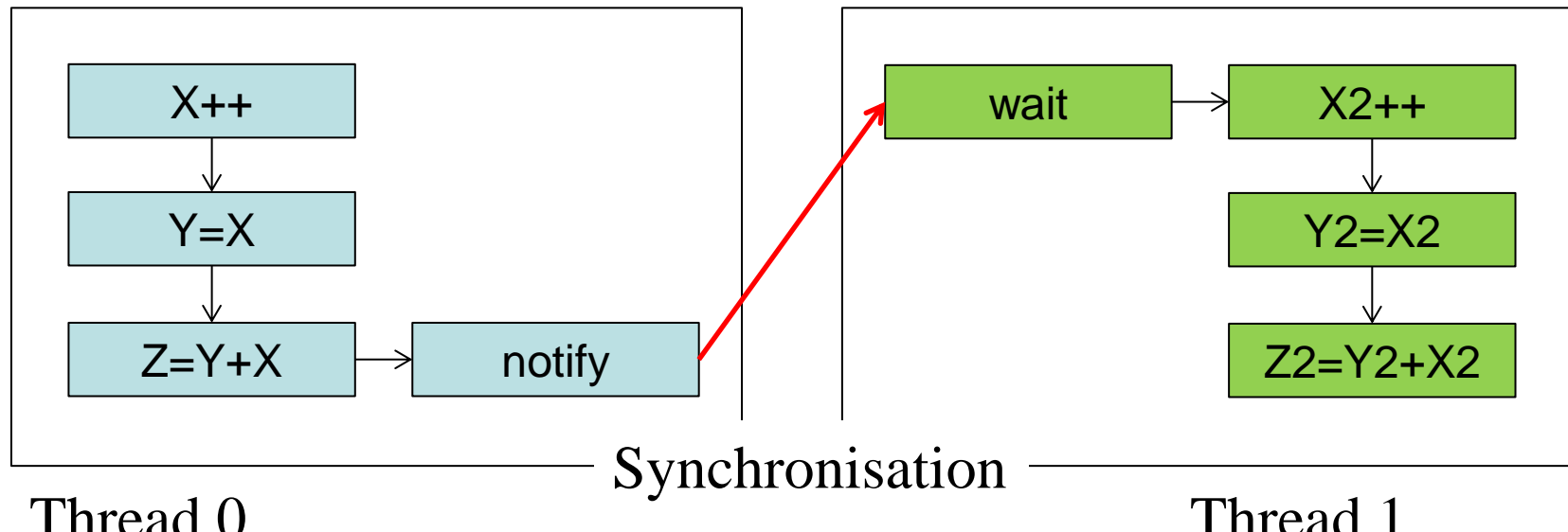


...
K étapes, N
thread =>
 K^N
entrelacements



Principes de la concurrence : synchronisation

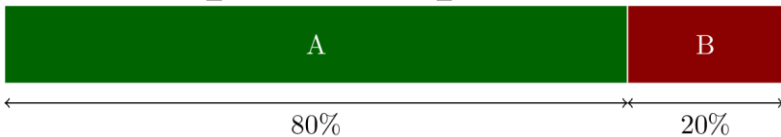
- Problème : Correctement capturer et exprimer les précédences
 - Par défaut seules les actions exécutées par un même thread sont ordonnées
 - On respecte l'ordre du programme
 - Méfiance avec Out-Of-Order execution cependant
 - C'est au programmeur de garantir les précédences essentielles **entre** threads
 - Ajout de mécanismes de **synchronisation** entre threads



Accélération Théorique Concurrente

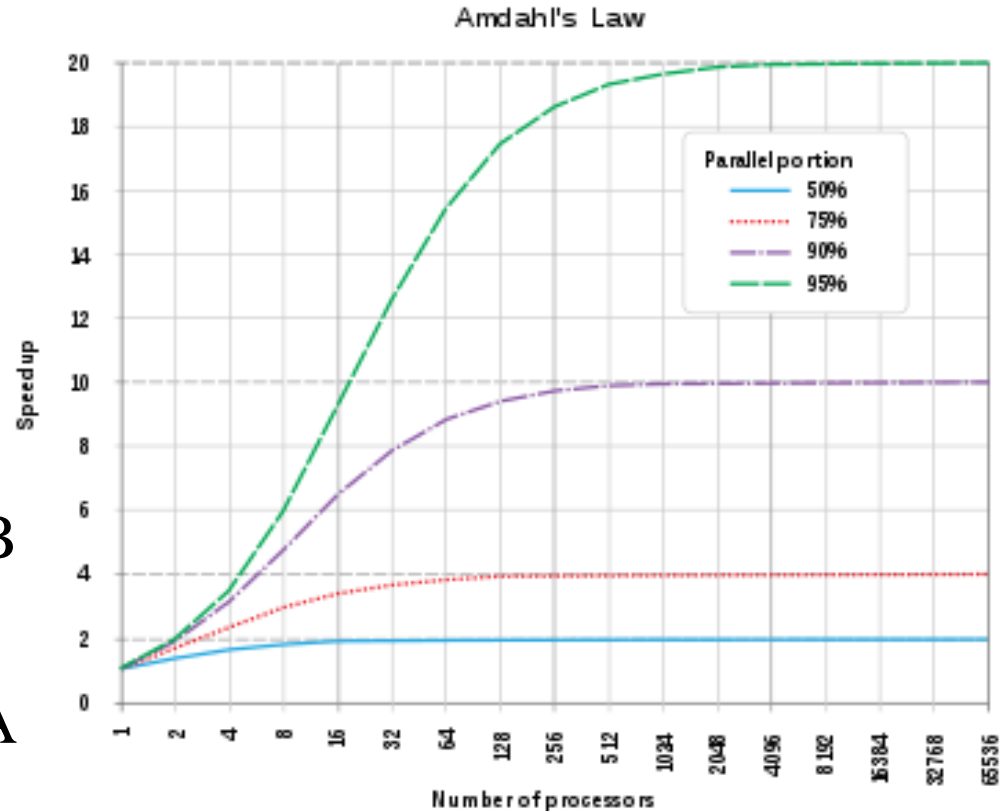
A : partie concurrente

B: partie séquentielle



- Loi d'Amdahl : latence sur +∞ CPU
 - Une partie séquentielle B
 - Pas d'accélération
 - Une partie concurrente A
 - Donc parallélisable
 - Accélération proportionnelle au nombre de cœurs => tend vers 0 ms

On ne peut pas tout paralléliser



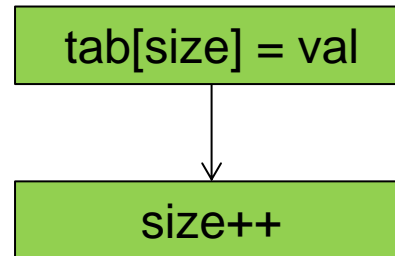
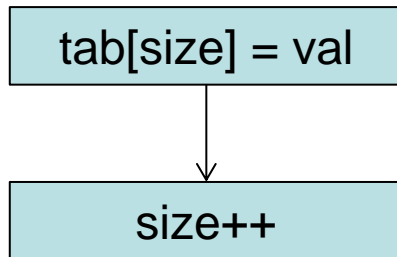
La concurrence =
Parallélisme Potentiel

Cf. aussi : Loi de Gustafson
(débit théorique)

Mieux adaptée pour les
architectures e.g vectorielles

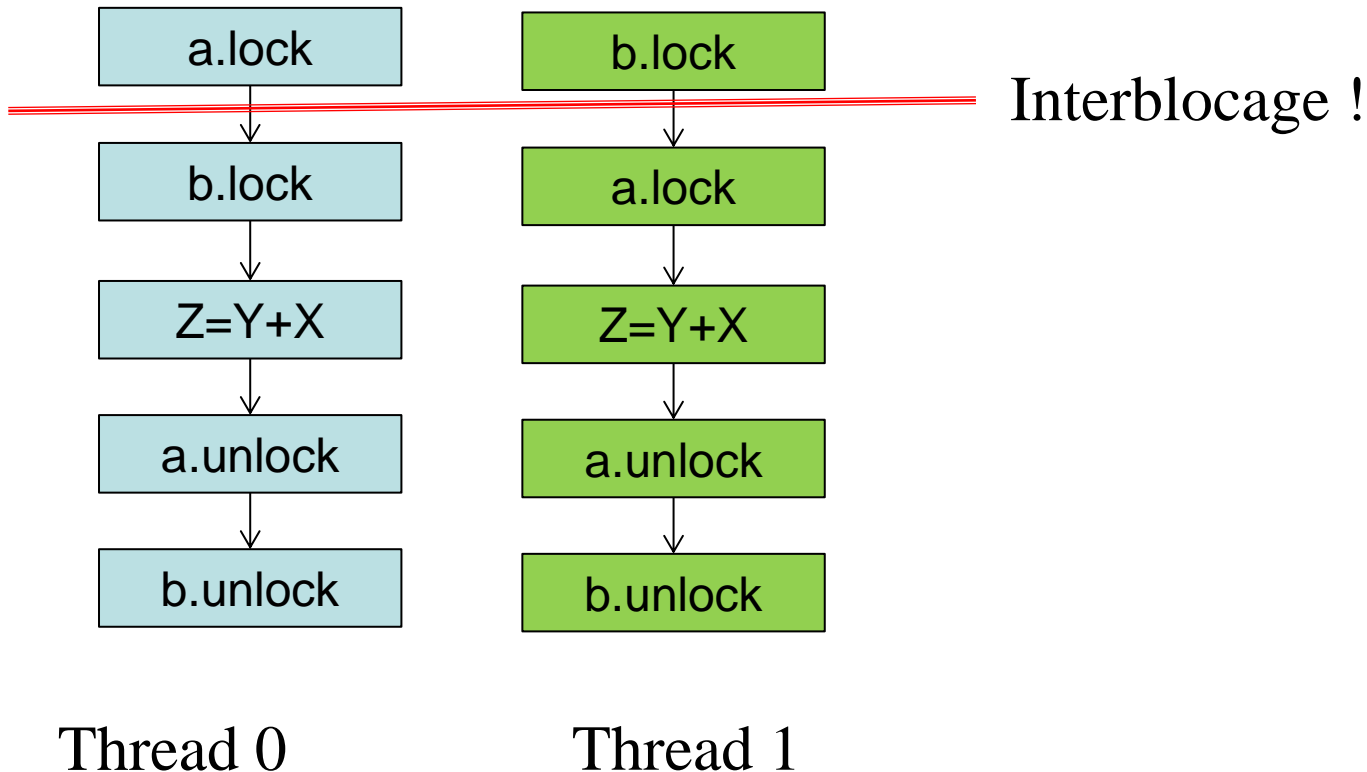
Problèmes liés à la concurrence

- Accès concurrents aux données
 - Ressources à protéger des écritures/lectures concurrentes
 - Par défaut, pas d'atomicité des instructions du C++
 - Data Race Condition
- Exemple : push_back concurrent ?



Problèmes liés à la concurrence

- Interblocages
 - Acquisition de séries de locks dans le désordre
 - Locks circulaires (dîner des philosophes)



Problèmes liés à la concurrence

- Indéterminisme (K puissance N entrelacements !)
 - Peu de contrôle sur les exécutions possibles
 - On présume que l'ordonnanceur est non déterministe
 - Pas d'a priori sur le parallélisme réel (\neq concurrence)
 - Difficile de reproduire les problèmes
 - Les programme peut fonctionner (passer un test) mais être faux
 - Un test peut échouer ou pas à cause de l'ordonnancement
- Loi de probabilité défavorable aux fautes de concurrence
 - En général, la commutation doit se passer « exactement au mauvais moment », dur à reproduire
 - Mais sur le long terme, la probabilité de faute tend vers 1

Les Thread du C++11

Processus léger ou "Thread"

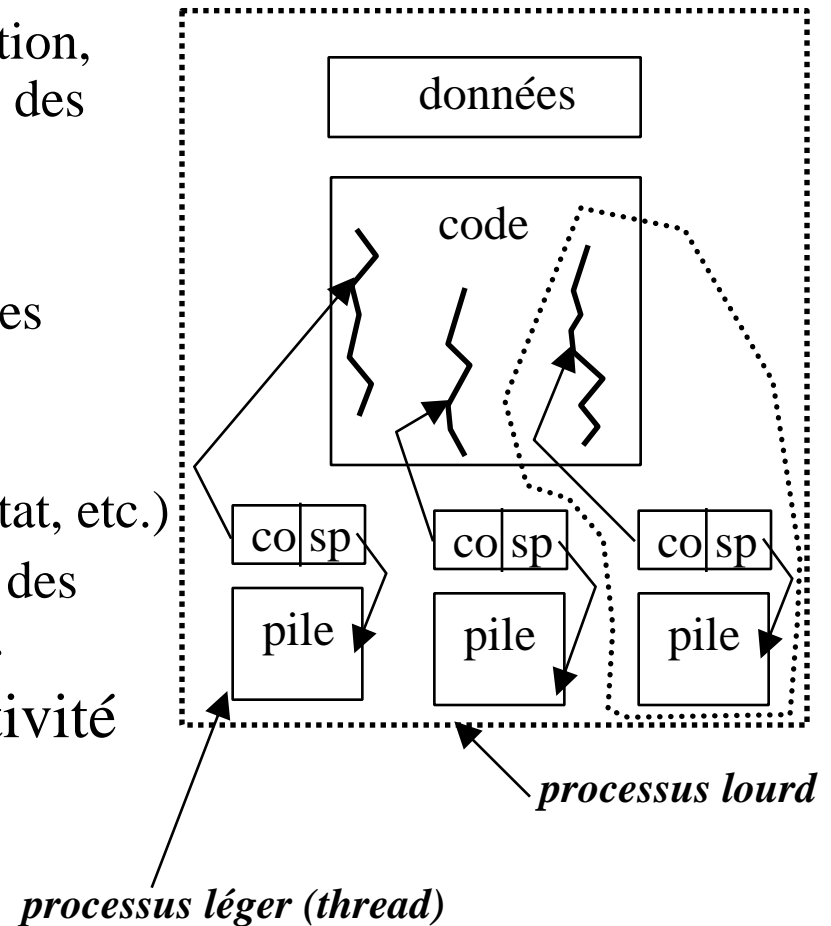
Partage les zones de code, de données, de tas + des zones du PCB (Process Control Block) :

- ✓ liste des fichiers ouverts, comptabilisation, répertoire de travail, userid et groupid, des handlers de signaux.

Chaque thread possède :

- ✓ un mini-PCB (son CO + quelques autres registres),
- ✓ sa pile,
- ✓ attributs d'ordonnancement (priorité, état, etc.)
- ✓ Quelques structures pour le traitement des signaux (masque et signaux pendants).

Un processus léger avec une seule activité
= un processus lourd.



La sémantique des threads

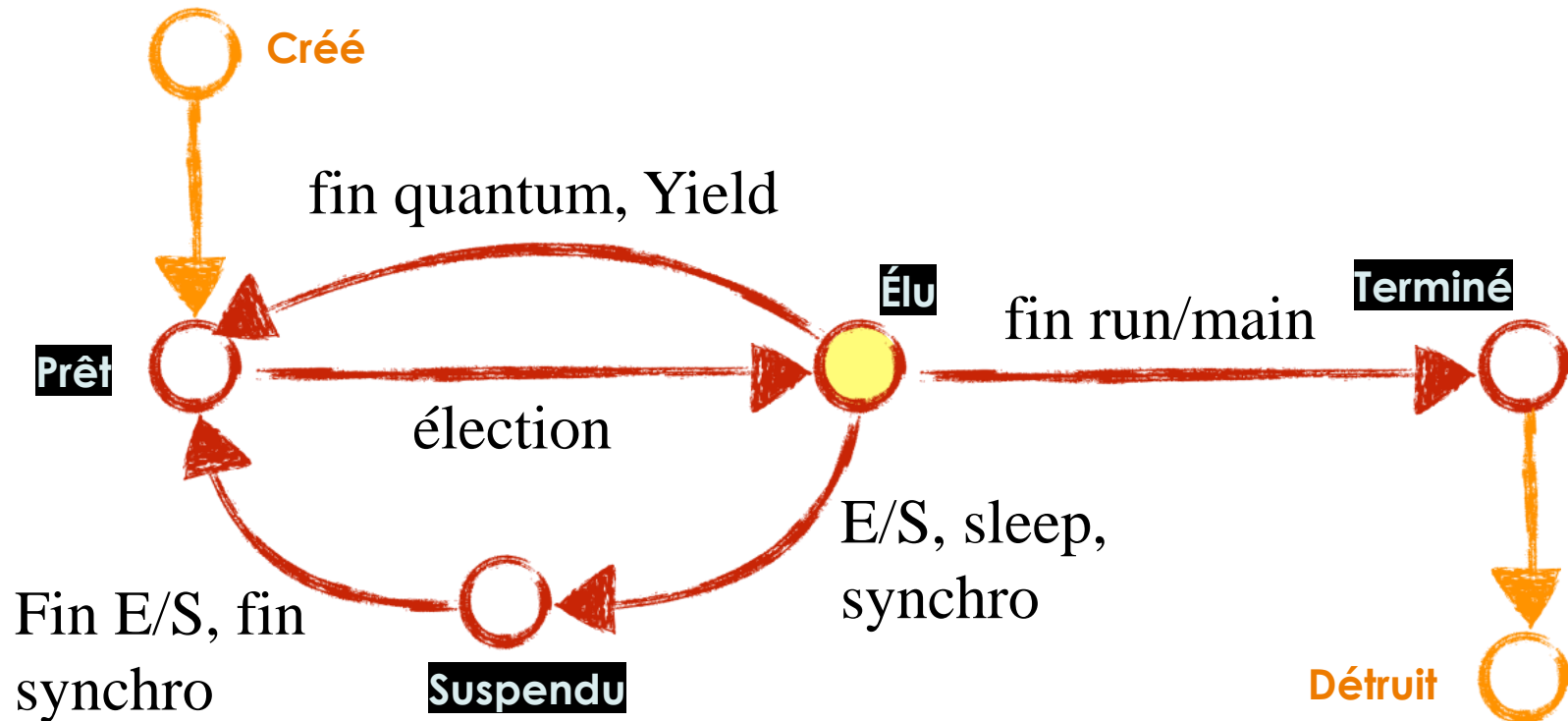
- Modèle mémoire
 - Stack (pile d'invocation) par thread
 - Chacun sa pile d'appels, fonctionnant normalement
 - Espace d'adressage commun
 - Tous les threads voient la **même** mémoire
 - En particulier, on peut voir le stack des autres Thread en C++, attention à ce qu'il reste valide pendant la durée de vie du thread ! (pour les références, `std::ref` est forcé dans constructor de thread)
 - Sémantique mémoire partagée (par opposition aux processus)
- Un seul processus système
 - Au niveau du noyau, un processus encapsule les threads
 - Les systèmes modernes (linux 2.6+, osx 10+, win 7+) ont tous des primitives et structures spécifiques aux threads

La sémantique des threads

- Ordonnancement
 - Quantum et commutation
 - Les threads sont élus et tournent sur un CPU
 - Au bout d'un **quantum**, on passe la main = commutation
 - Commutations explicites sont possibles
 - Entrées sorties, sleep, yield, ... provoquent une commutation
 - Indéterminisme en pratique de l'ordonnanceur
 - Aucune hypothèse sur le quantum etc...
 - Niveau de parallélisme indépendant du nombre de Threads
 - Application multi thread sur mono cœur, e.g. GUI
 - Un thread peut réaliser plusieurs tâches (thread pool)

Cycle de vie

- 🏆 Cycle de vie (simplifié) d'un fil d'exécution
- 🏆 On ne contrôle pas l'ordonnanceur => élection/fin quantum en particulier
- 🏆 Selon l'architecture, plusieurs threads élus simultanément possible



Concurrence et C++ moderne

- Thread
 - Briques de base
- Atomic
 - Pour les types primitif
 - Barrières mémoire fines
- Mutex
 - Exclusion mutuelle, beaucoup de variantes
- Shared
 - Locks Lecteurs/Ecrivains (C++17)
- Condition
 - Notifications et attente
- Future
 - Exécution asynchrone

C++ 20 :
coroutines, latch, barrier
sémaphore, jthread...
Le standard progresse

Thread : Création, Terminaison et Join

- Un thread est représenté par la classe `std::thread`
- Création
 - L'instanciation du thread (constructeur) prend
 - une fonction à exécuter
 - les paramètres à passer à la fonction
- Terminaison
 - Le thread se termine quand il sort de la fonction
- Join
 - L'objet thread ne sera collecté que quand il aura été **join**
 - Un seul thread peut join à la fois
 - Invocation bloquante jusqu'à la terminaison du thread ciblé

Exemple Basique : creation/join

```
#include <iostream>    // std::cout
#include <thread>        // std::thread

void foo()
{ // do stuff...
}

int main()
{
    std::thread first (foo);    // spawn new thread that calls foo()
    // do something else
    first.join();               // pauses until first finishes
    std::cout << "foo completed.\n";
    return 0;
}
```

Passage de paramètres

- Nombre et typage arbitraire
 - Syntaxe confortable et bien typée
 - Fonction retournant void
 - Liste des arguments à lui passer
- Cas particulier passage de références
 - Durée de vie de la référence passée doit être garantie par le programmeur
 - `std::ref(var)` et `std::cref(const var)`
- Pas de valeur de retour => modifier des données partagées
 - Attention aux synchronisations
 - Join est basique mais constitue un mécanisme de synchronisation fiable

Example

```
void f1(int n, bool b);  
void f2(int& n);  
int main()  
{  
    int value=0;  
    std::thread t1(f1, value + 1,true); // pass by value  
    std::thread t2(f2, std::ref(value)); // pass by reference  
    ... // do some stuff  
    t1.join(); t2.join();  
    cout << value << endl;  
}
```

Yield et Sleep

- `yield()`
 - Demande explicite (mais pas contraignante) de commutation
 - Laisse l'occasion aux autres threads de prendre la main
 - Implémentation dépendante des plateformes

`std::this_thread::yield();`

- `sleep_for(chrono::duration)`
 - Demande au système de nous réveiller au bout d'un moment
 - Implicitement force une commutation
 - Durée du sleep imprécise \geq demandé (`steady_clock` recommandé)
 - `sleep_until` variante pour attendre une date donnée

`std::this_thread::sleep_for(2s); // C++14`

`std::this_thread::sleep_for(std::chrono::milliseconds(200)); // C++11`

Threads Détachés

- Sémantique de destruction d'un Thread
 - Invocation du destructeur de l'objet thread doit avoir lieu
 - Après join ou detach
 - Sans quoi fautes mémoires (e.g. destructeurs du stack)
- Détacher un thread => plus de join
 - Il poursuit son exécution indépendamment
 - Se terminera avec le programme (après le main) i.e. avec le thread principal qui exécute « main »
- Intérêt principal
 - Threads de background simples, créations de statistiques etc...

Example

```
void printStats(const Data * d );
```

```
void independentThread(const Data * data)
{
    while (true) {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        printStats(data);
    }
}
```

```
int main()
{
    Data d;
    std::thread t(independentThread, &d);
    t.detach();
    // work with data
    int i=0;
    while (i++ < 10) {
        std::this_thread::sleep_for(
            std::chrono::milliseconds(300));
        d.update();
    }
}
```



Atomic



Accès concurrents en écriture

- Deux threads exécutent `i++` en concurrence, `i` vaut 0 initialement
 - Combien vaut `i` après ?
- Pas de garanties sur les accès concurrents
 - Séparation du « fetch » lire la valeur et du « store » la stocker
 - Problème accentué sur les matériels modernes (partage de ligne de cache, ...)

Data race

- On parle de "race" car le dernier à écrire sa valeur écrase l'autre
 - Des instructions sont perdues
- On ne contrôle pas l'ordonnanceur !
 - On **doit** supposer qu'on peut être commuté n'importe quand
 - En majorité les opérations y compris simple ($x++$, $x=\text{valeur}$, $x==\text{valeur}$) n'ont pas de garantie d'être atomique
- Les architecture modernes sont plus complexe qu'il n'y paraît
 - Le standard C++ est clair : sans utilisation de synchronisations, le compilateur a le droit d'utiliser des caches sans les synchroniser => on est dans du UB

Accès concurrents = UB

Standard C++ : section 1.10 clause 21: Data Race = UB

The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. **Any** such data race results in **undefined behavior**.

Standard C++ : section 1.3.24 : UB = ????

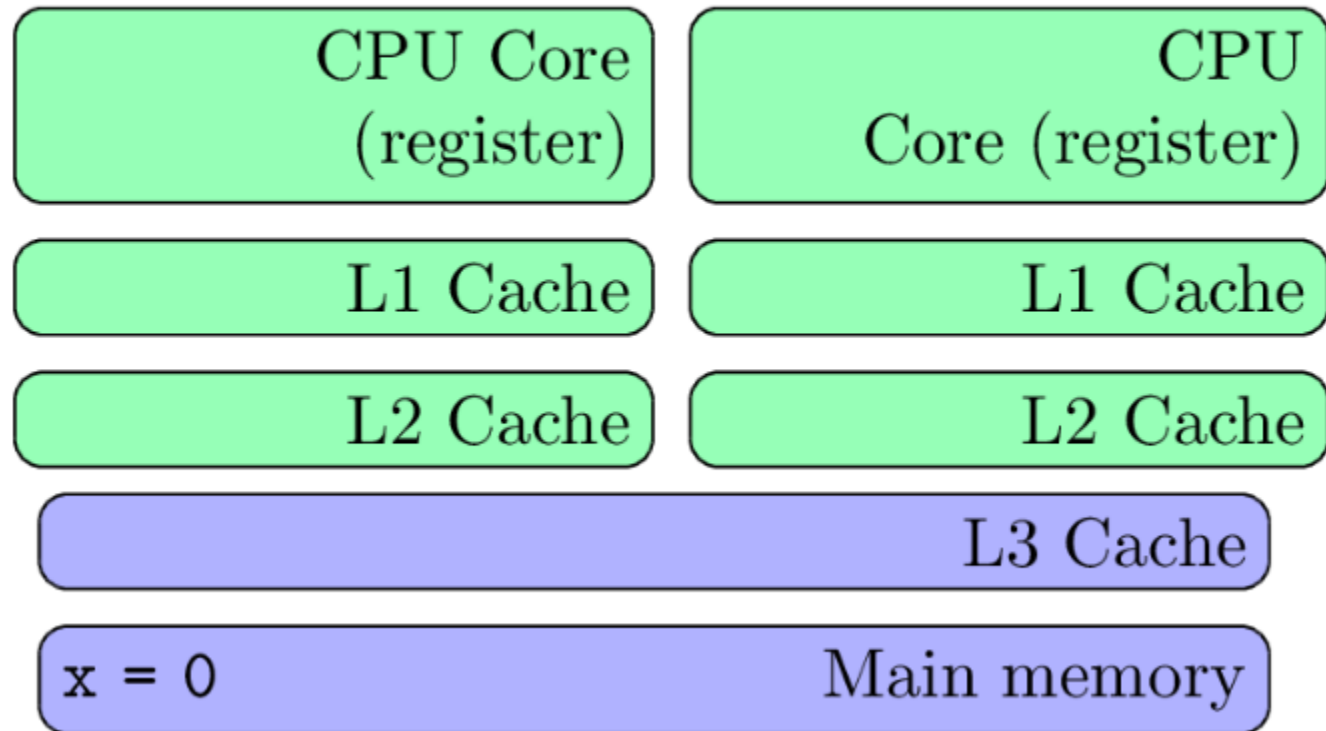
Undefined Behavior (UB) is behavior for which this International Standard imposes no requirements... Permissible undefined behavior ranges from *ignoring the situation completely* with *unpredictable results*, to behaving during translation or program execution in a documented manner characteristic of the environment...



Nécessité d'une gestion matérielle

Sans synchronisations :

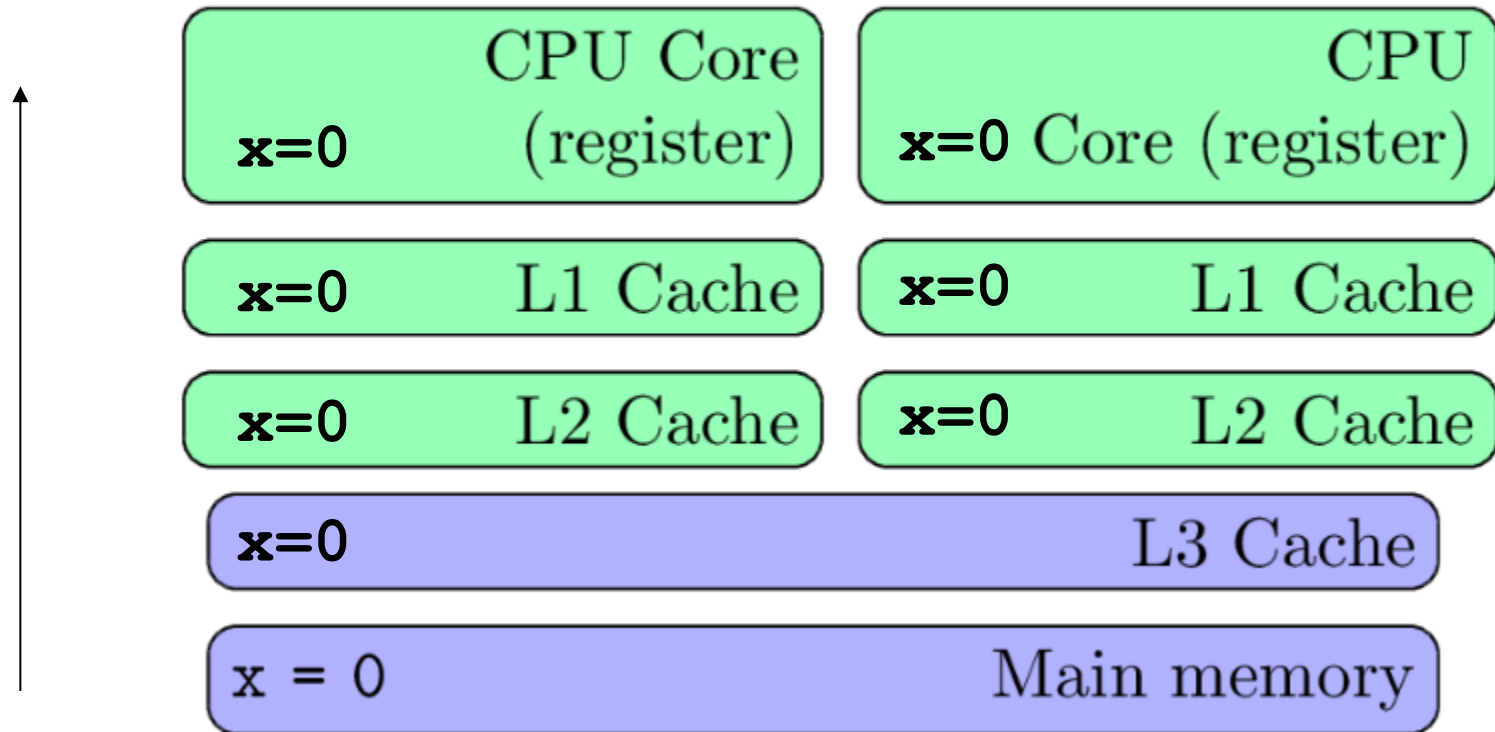
What's going on ?



Nécessité d'une gestion matérielle

Sans synchronisations :

What's going on ?

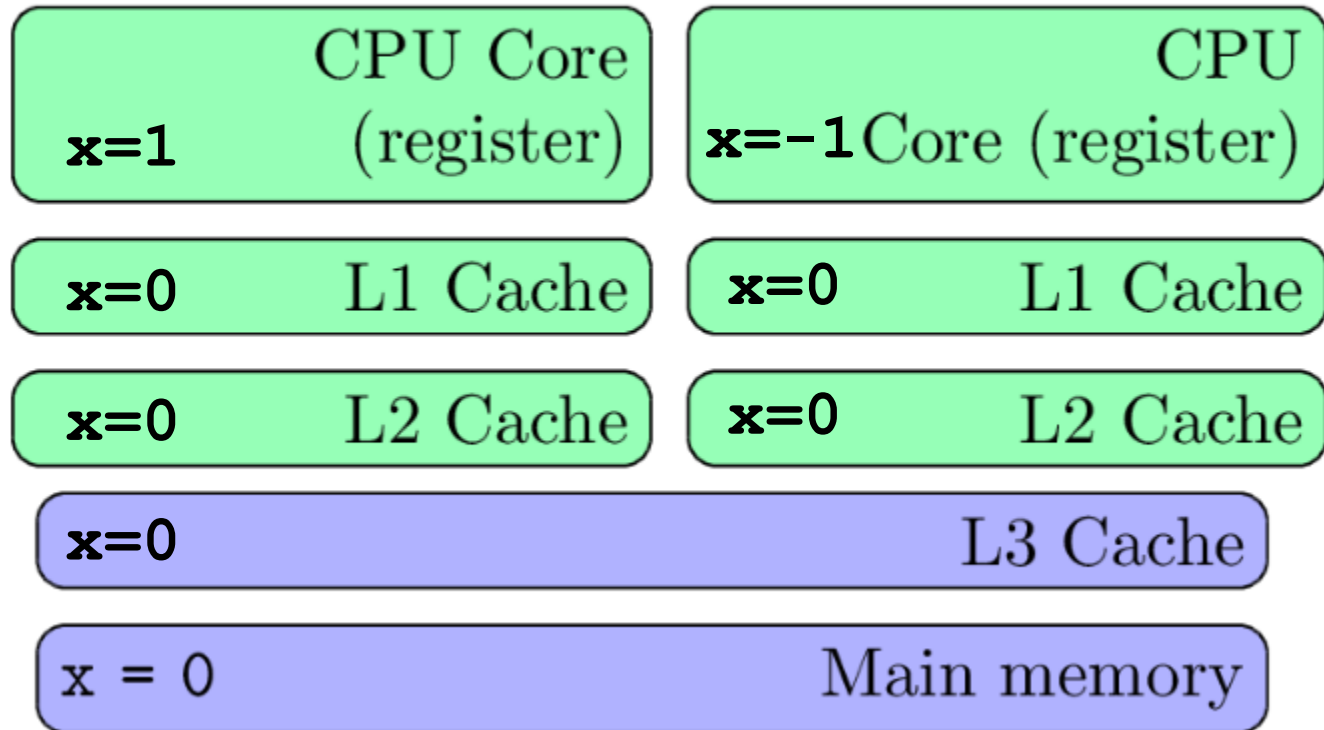


chargement
par chaque thread

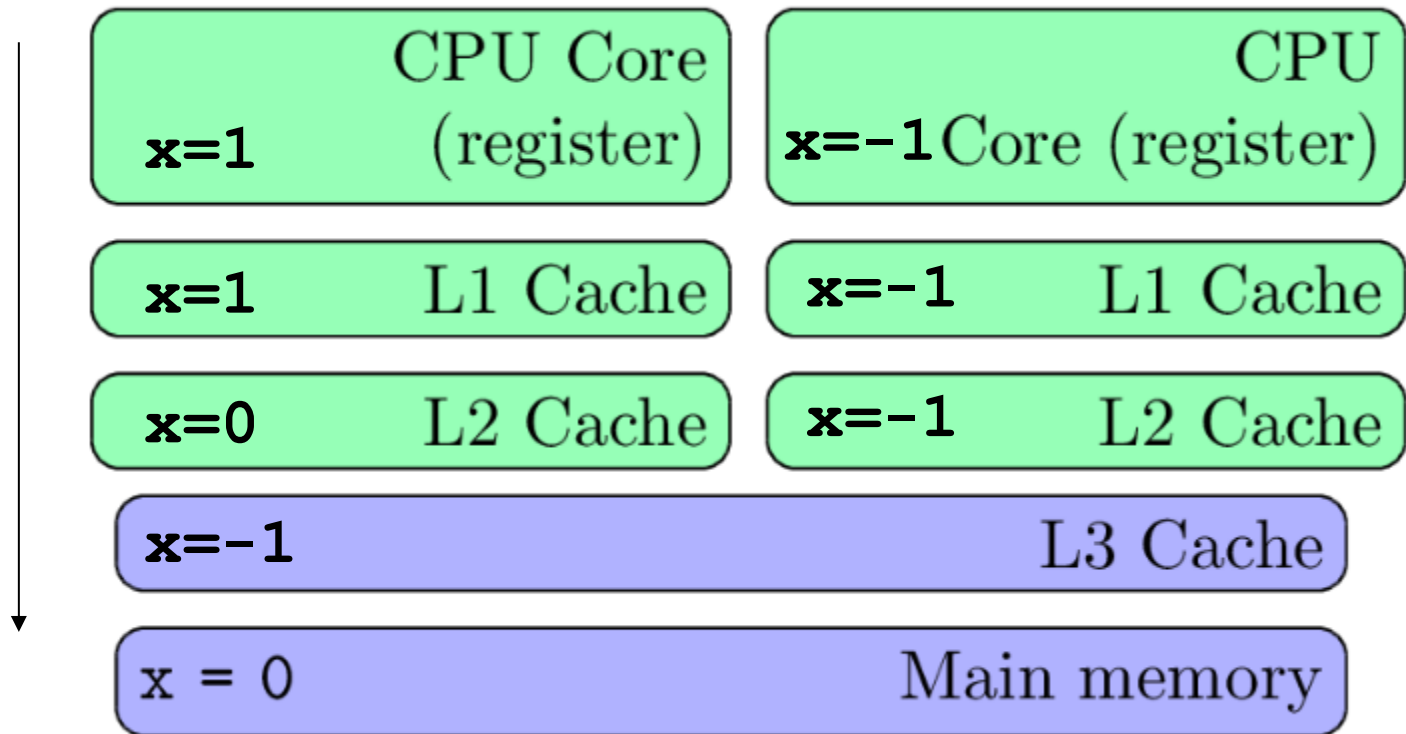
Nécessité d'une gestion matérielle

Sans synchronisations :

What's going on ?



Nécessité d'une gestion matérielle



Déchargement par chaque thread

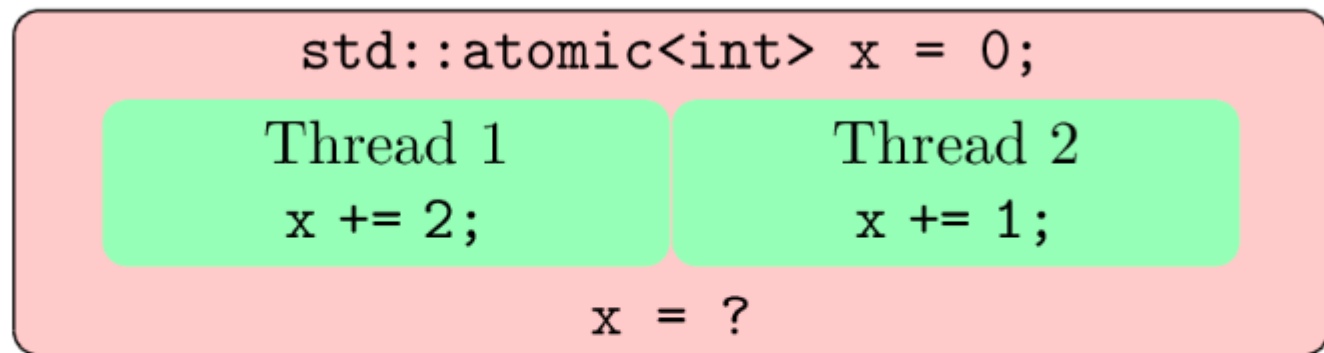
Pas de contrainte réelle sur quand le faire

L'action de thread 0 n'est pas nécessairement visible de thread 1

Data race quand on recopie dans le niveau L3

Atomic en pratique

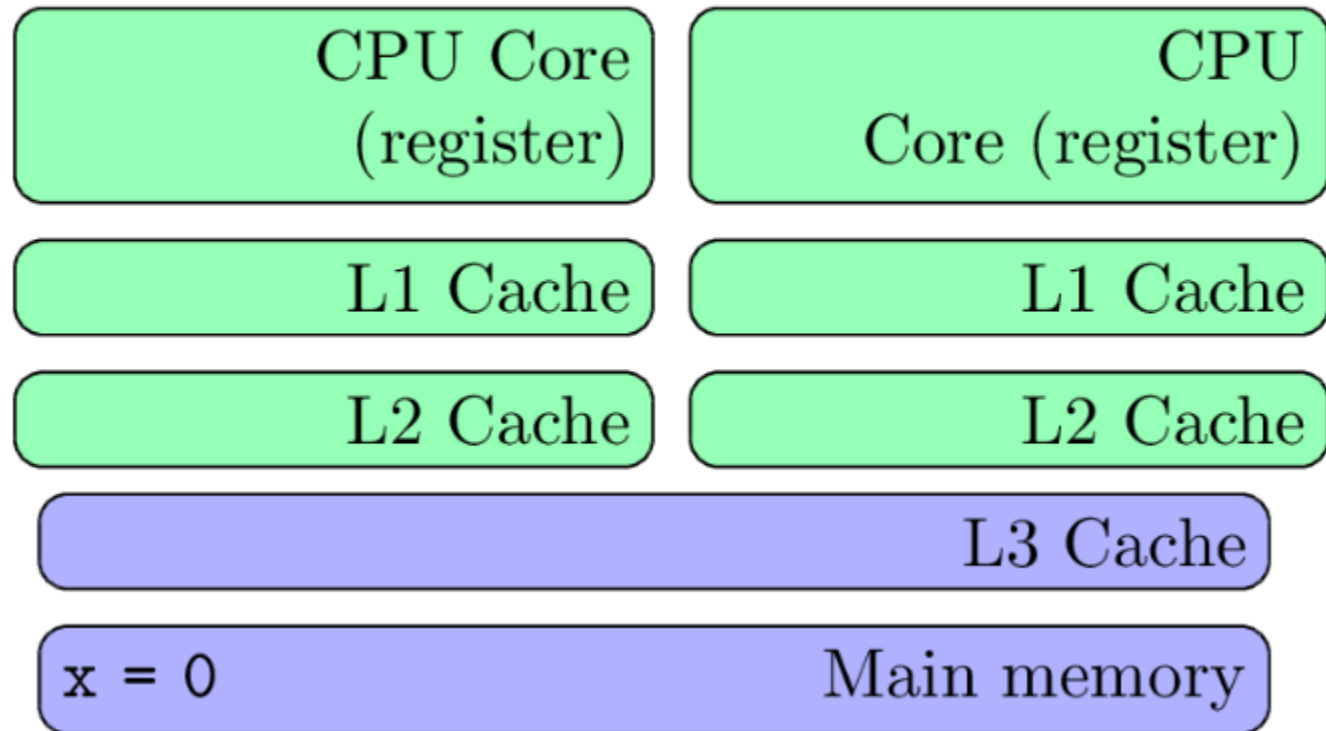
NUMA revisited

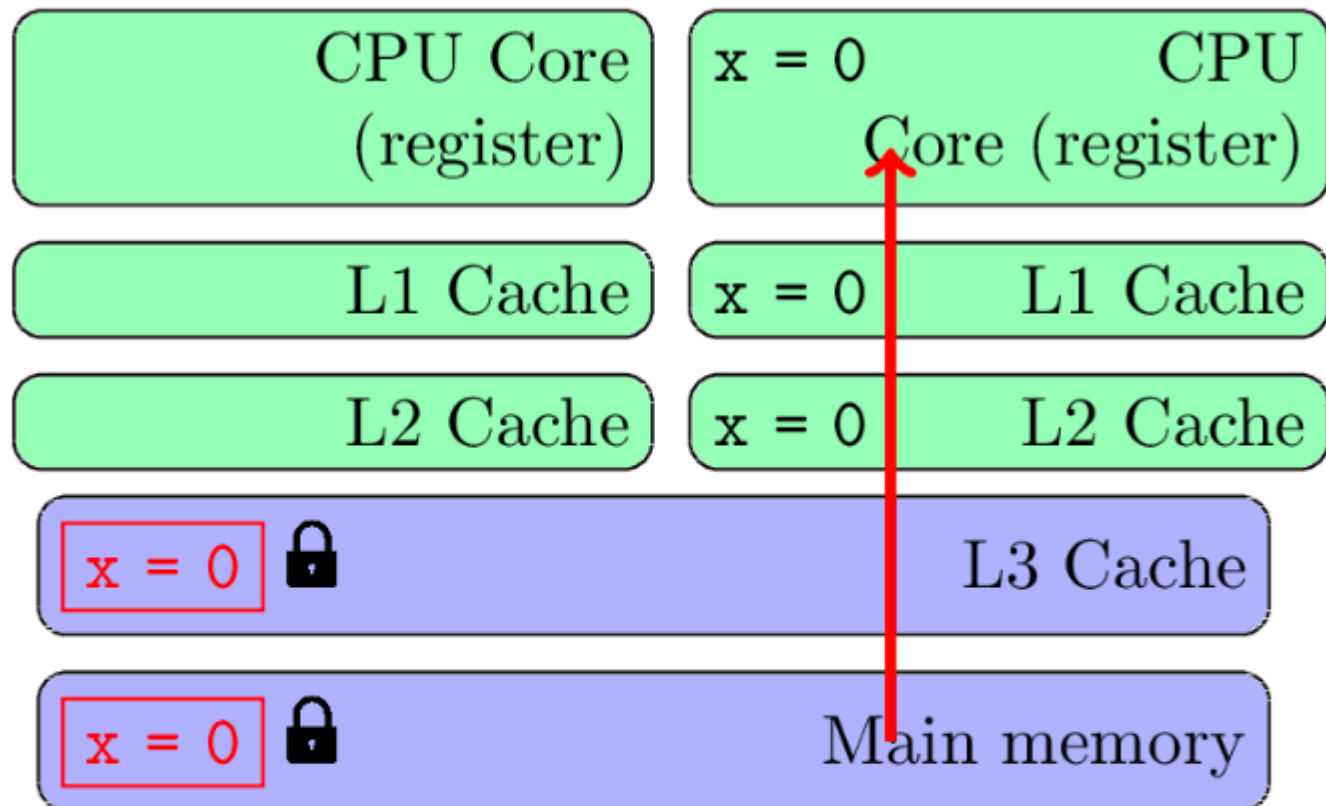


- Read modify write operation:
 - Read `x` from memory
 - Add something to `x`
 - Write `x` to memory

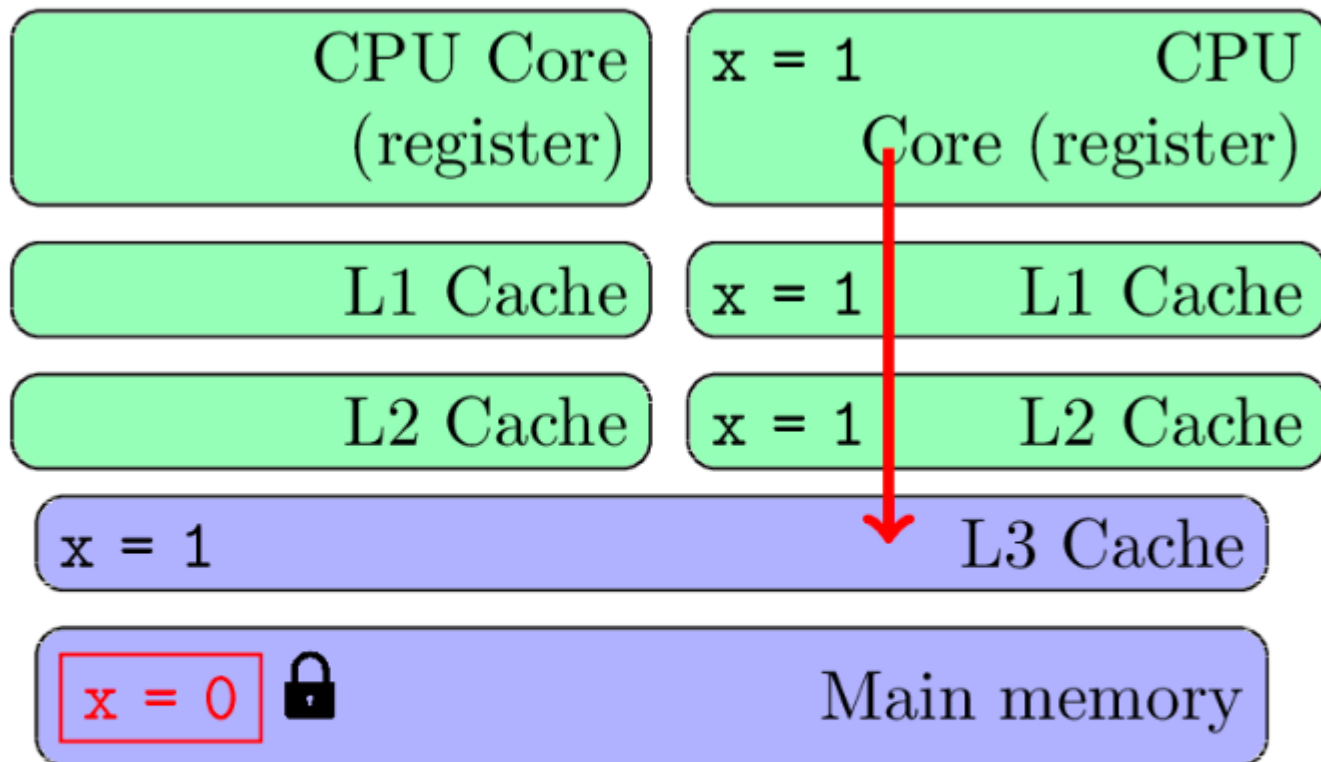
Nécessité d'une gestion matérielle

What's going on ?

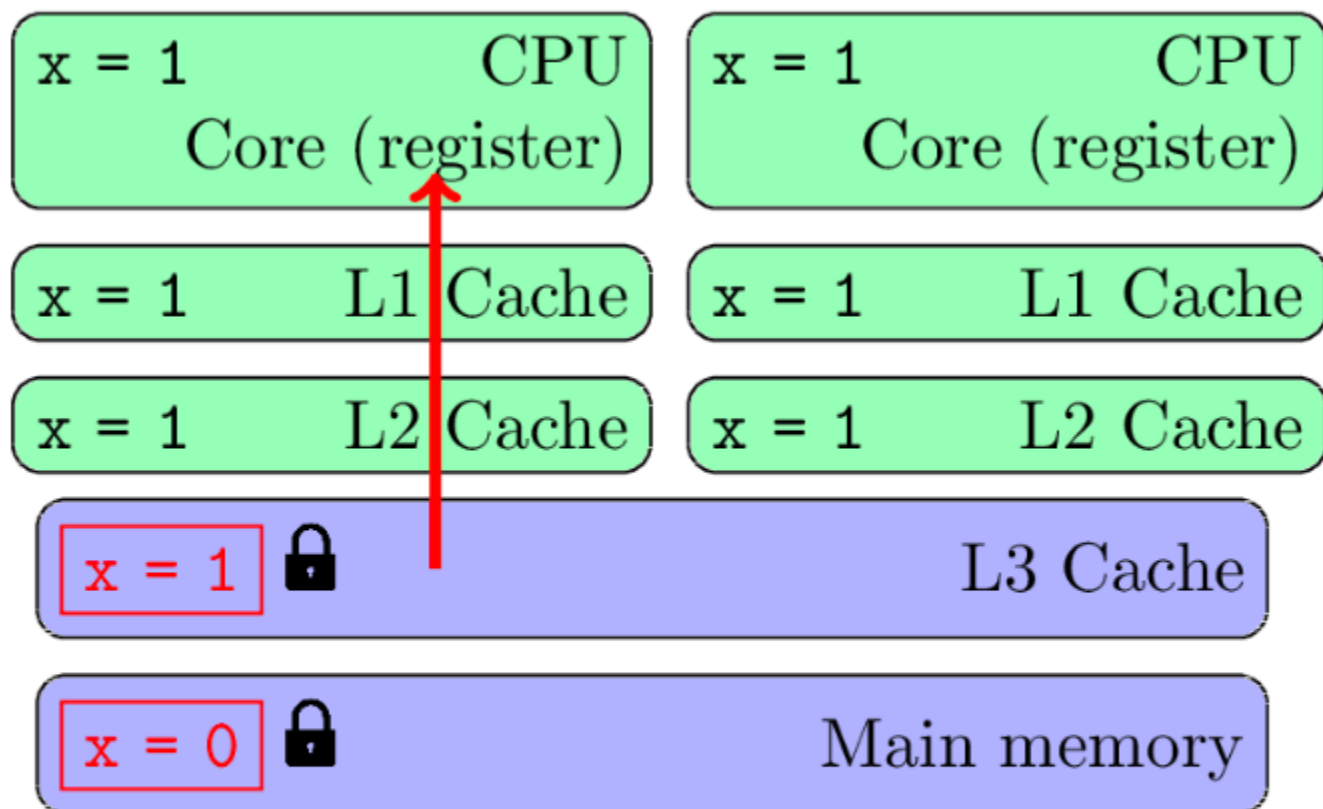




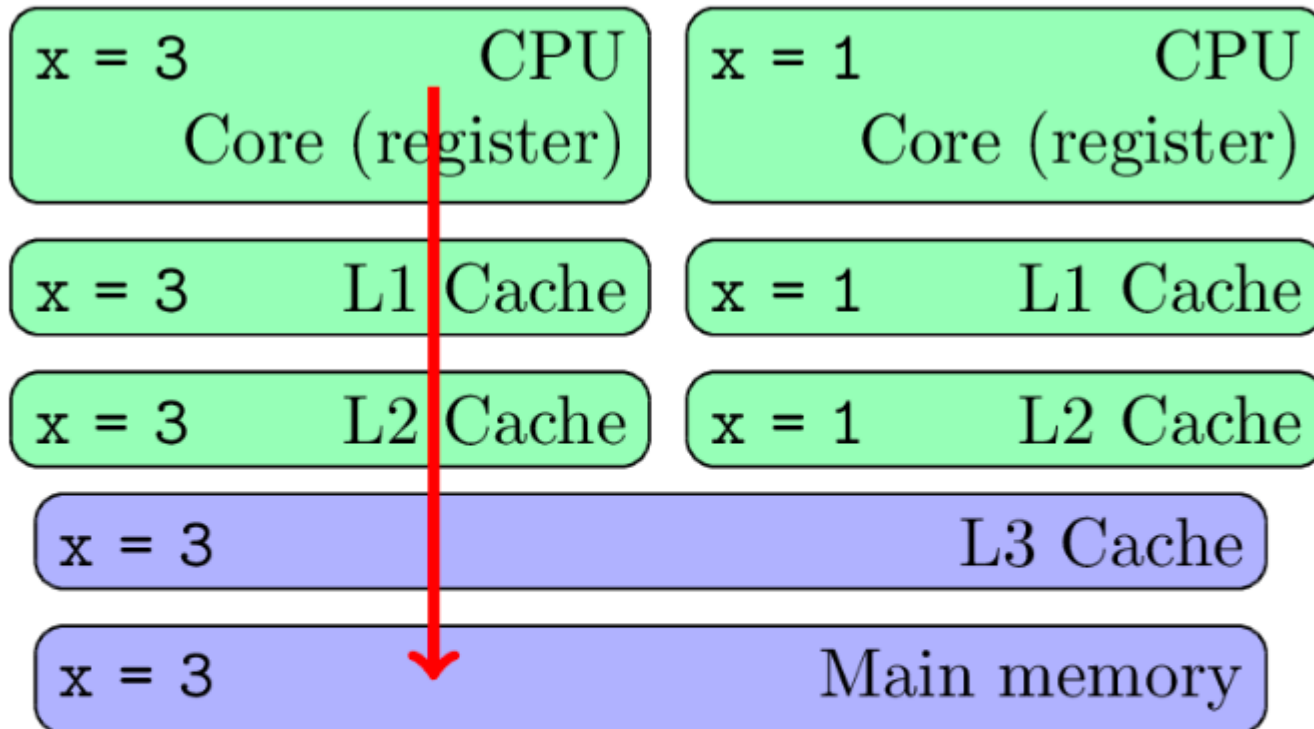
- Thread 2 executes
 - It fetches x from caches to main memory
 - It locks x address (by hardware impl.)
- Thread 1 executes
 - It try to fetch a x which is locked (goes to by hardware impl.)



- Thread 2 keeps executing
 - It updates x
 - It write back x to caches
 - It *unlocks* x address



- Thread 1 executes
 - It fetches x
 - It locks x address



- Thread 1 keeps executing
 - It updates `x`
 - It write back `x` address
 - It unlocks `x` address

La cohérence de cache assurée au niveau matériel permet
au niveau logiciel de construire des synchronisations

Accès concurrents en écriture

- Il existe des solutions matérielles
 - `fetch_and_add`, `fetch_and_sub`, `compare_exchange...`
 - Nécessaires pour que le logiciel construise ses propres briques
 - `atomic_boolean_flag` est la base de toute synchronisation
- Accessibles finement en C++
- Dans l'UE : types atomiques primitifs seulement
 - Barrières mémoire fines, conteneurs lock-free considérés hors programme
 - Cf. l'excellent livre de A. Williams : **C++ Concurrency in Action**

Propriétés et opérateurs d'un atomic

- Offert pour les types primitifs numériques
 - Garantit l'atomicité des opérations :
 - incrément ++ ou +=,
 - décrement – ou -=,
 - opérations bit à bit &=, |=, ^=
 - Cas particulier booléen : atomic_flag

Propriétés et opérateurs d'un atomic

- Utilisation « naturelle »
 - Déclaration
 - `atomic<int> i =0;`
 - Opérateurs Disponibles
 - `i++`
 - `i+=5;`
 - Opérations non atomiques
 - Multiplication, manipulation classique
 - `i = i +5` \Rightarrow pas atomique.

Example

```
std::atomic<bool> ready (false);  
std::atomic<int> counter(0);
```

```
void count1m (int id) {  
    while (!ready) { std::this_thread::yield(); }    // wait for the ready signal  
    for (int i=0; i<10000; ++i) { counter += 3;}  
};
```

```
int main ()  
{  
    std::vector<std::thread> threads;  
    for (int i=1; i<=10; ++i) threads.push_back(std::thread(count1m,i));  
    ready = true;  
    for (auto& th : threads) th.join();  
    std::cout << counter << std::endl;  
    return 0;  
} //NB: sans atomic<bool> on a data race => UB immédiatement !
```

Limites des atomic

- Mécanisme bas niveau
 - Protège des données de petite taille
 - Store et Load atomic pour les structures cependant
 - Pas d'atomicité sur les séquences d'opérations
 - Contrôle puis action ? Mise à jour d'une table ?
- Base pour la réalisation de mécanismes de synchronisation

```
class SpinLock {  
    std::atomic_flag locked = ATOMIC_FLAG_INIT ;  
public:  
    void lock() {  
        while (locked.test_and_set(std::memory_order_acquire)) { ; }  
    }  
    void unlock() {  
        locked.clear(std::memory_order_release);  
    }  
};
```

Mutex

Principe d'une Section Critique

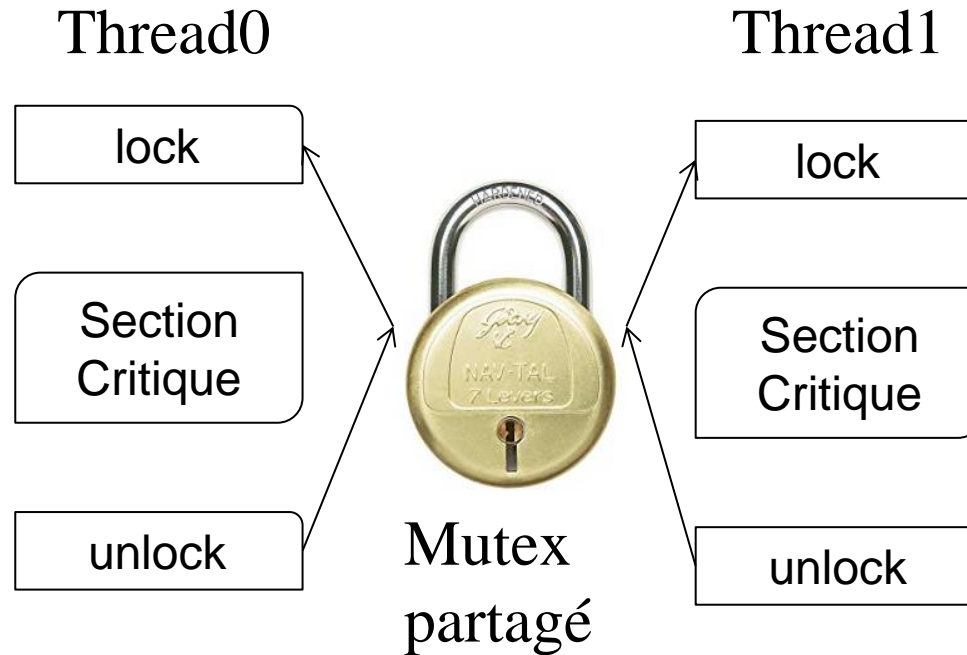
- Suites d'actions cohérentes sur des données
 - Cohérentes
 - Pas de *data race* des accès concurrents
 - Aspect Transaction
 - Tout devient visible ou rien ne l'est

Atomic ne permet pas en général de traiter ce problème.

Principe d'une Section Critique

- Un verrou (mutex) permet de garantir l'exclusion mutuelle
 - **lock** :
 - Si le mutex est disponible, l'acquiert
 - Si non disponible : bloquant, endort le thread
 - **unlock** :
 - Fait par le propriétaire/verouilleur uniquement
 - Jamais bloquant, réveille les thread bloqués en attente

Exclusion mutuelle par mutex



- Un thread en section critique peut être interrompu
 - Pas de contrôle de l'ordonnanceur
 - Mais s'il détient le lock, les autres threads ne peuvent pas lui marcher dessus
 - Un seul thread à la fois en section critique

<mutex>

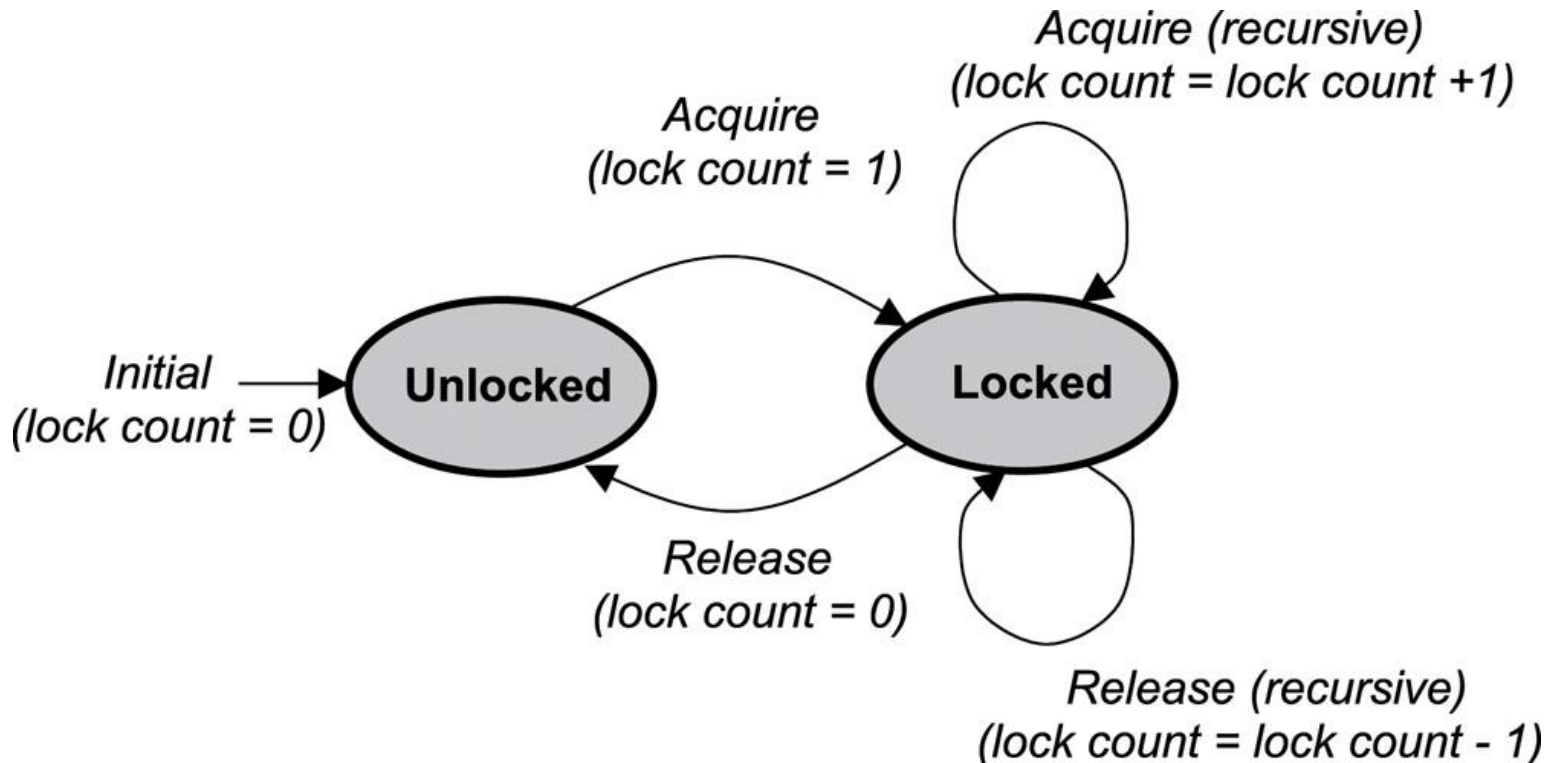
```
std::mutex mtx;           // mutex for critical section

void print_block (int n, char c) {
    // critical section (exclusive access to std::cout signaled by locking mtx):
    mtx.lock();
    for (int i=0; i<n; ++i) { std::cout << c; }
    std::cout << '\n';
    mtx.unlock();
}

int main ()
{
    std::thread th1 (print_block,50,'*');
    std::thread th2 (print_block,50,'$');
    th1.join();
    th2.join();
    return 0;
}
```

Mutex

- Le mutex possède une file d'attente des Threads qui ont fait lock alors qu'il n'était pas disponible
- Le mutex a deux états
 - recursive_mutex supporte la réacquisition par le même thread



unique_lock

- Facilité syntaxique pour un style de programmation
 - Exclusion mutuelle ayant comme portée une fonction, un bloc
 - Utilisation similaire au *synchronized* de Java
- unique_lock (RAII)
 - Mentionne un lock, acquis à la construction
 - La destruction de l'instance libère le lock (out of scope)

```
int g_i = 0;
std::mutex gi_mutex; // protects g_i

void safe_increment()
{
    unique_lock<mutex> lock(gi_mutex);
    ++g_i;
}
```

```
int main()
{
    std::thread t1(safe_increment);
    std::thread t2(safe_increment);
    t1.join();
    t2.join();
}
```

lock_guard

- Utilisation similaire à unique_lock

```
int g_i = 0;  
std::mutex gi_mutex; // protects g_i
```

```
void safe_increment()  
{  
    lock_guard<mutex> lock(gi_mutex);  
    ++g_i;  
}
```

- Légèrement plus simple
 - Force le lock à la création
 - Pas de façon de relacher en dehors du destructeur
- Pour unique_lock on a std::defer

RAII : Resource Acquisition Is Initialization

```
#include <mutex>
#include <iostream>
#include <string>
#include <fstream>
#include <stdexcept>
void write_to_file (const std::string & message) {
    // mutex to protect file access (shared across threads)
    static std::mutex mutex;
    // lock mutex before accessing file
    std::unique_lock<std::mutex> lock(mutex);
    // try to open file
    std::ofstream file("example.txt");
    if (!file.is_open())
        throw std::runtime_error("unable to open file");
    // write message to file
    file << message << std::endl;
    // file will be closed 1st when leaving scope (regardless of exception)
    // mutex will be unlocked 2nd (from lock destructor) when leaving
    // scope (regardless of exception) (exemple adapté de Wikipedia)
}
```

Classe MT-Safe

```
#include <mutex>

// Thread-safe Counter class using the big fat lock approach
class Counter {
private:
    mutable std::mutex mtx; // mutable to allow locking in const methods
    int count;

public:
    Counter() : count(0) {}

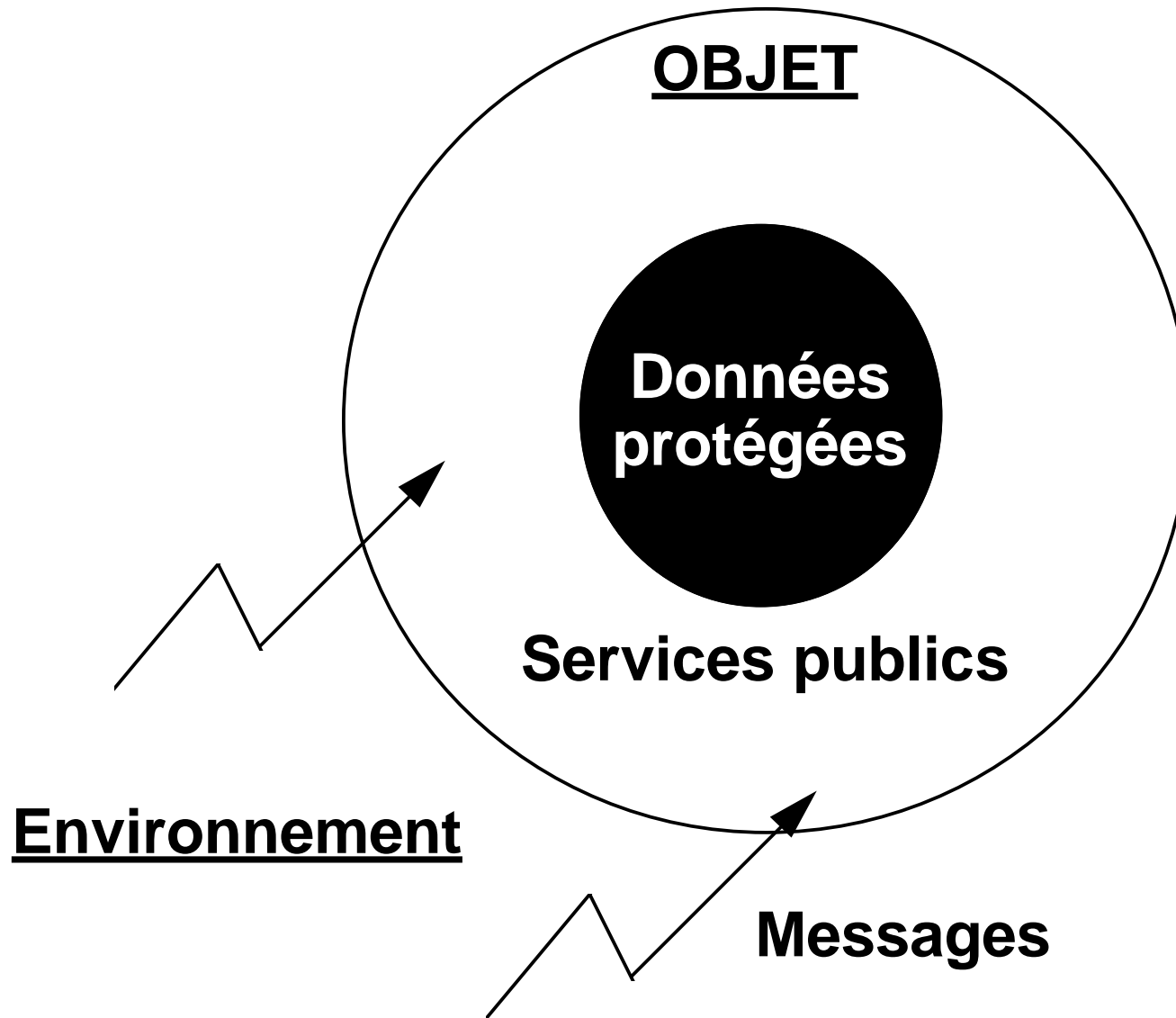
    void increment() {
        std::lock_guard<std::mutex> lock(mtx);
        ++count;
    }

    int get() const {
        std::lock_guard<std::mutex> lock(mtx);
        return count;
    }
};
```

MultiThread-Safe

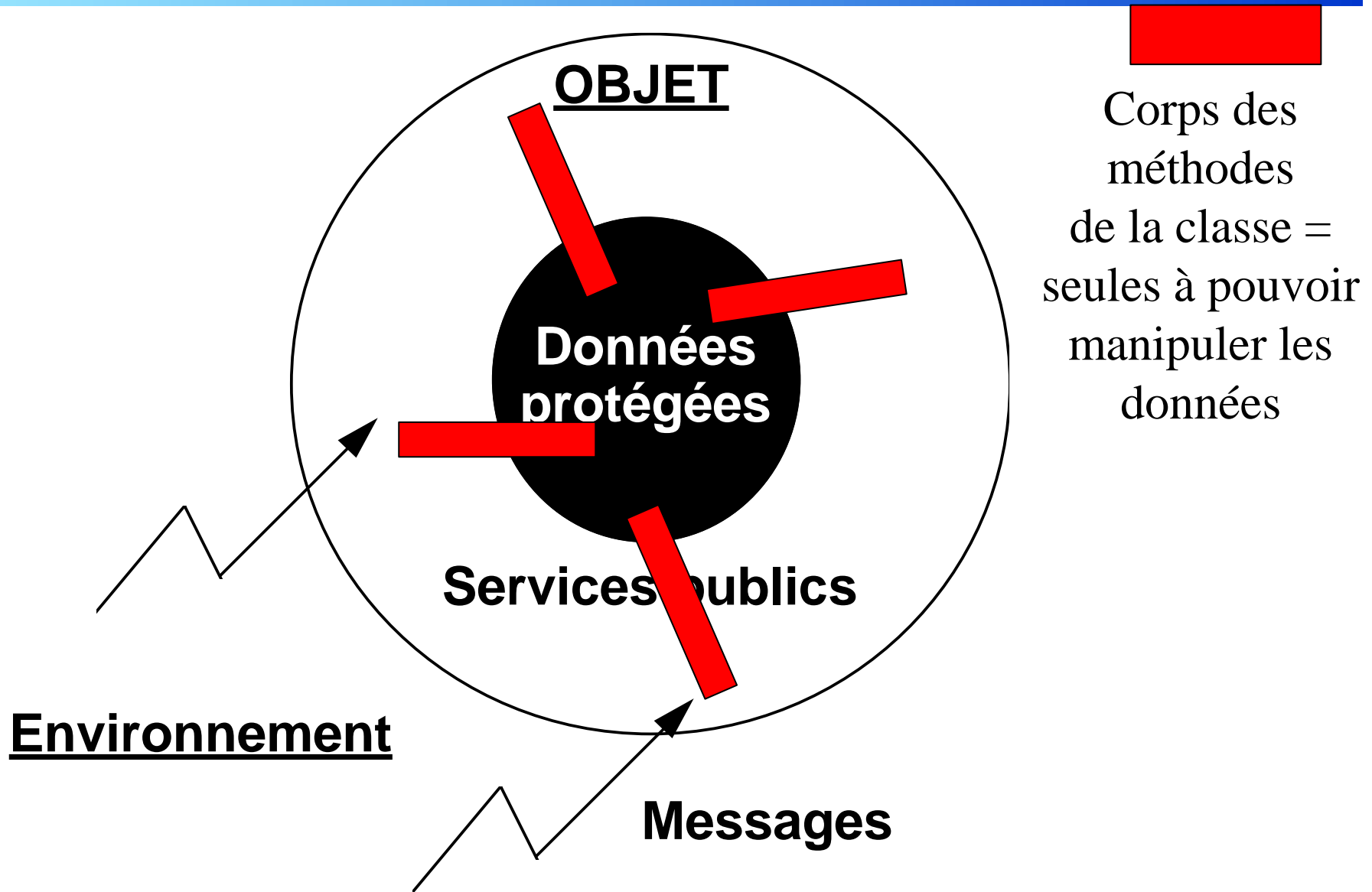
- Des fonctions manipulant des données
 - Résistantes à la réentrance (attention aux static)
 - Résistantes aux accès concurrents
- Classe MT-safe
 - Encapsule son comportement
 - Prévient les data-race
 - Utilise des mécanismes de synchronisation; certaines opérations peuvent donc être bloquantes ou retardées (contention)

L 'objet : notion d'encapsulation

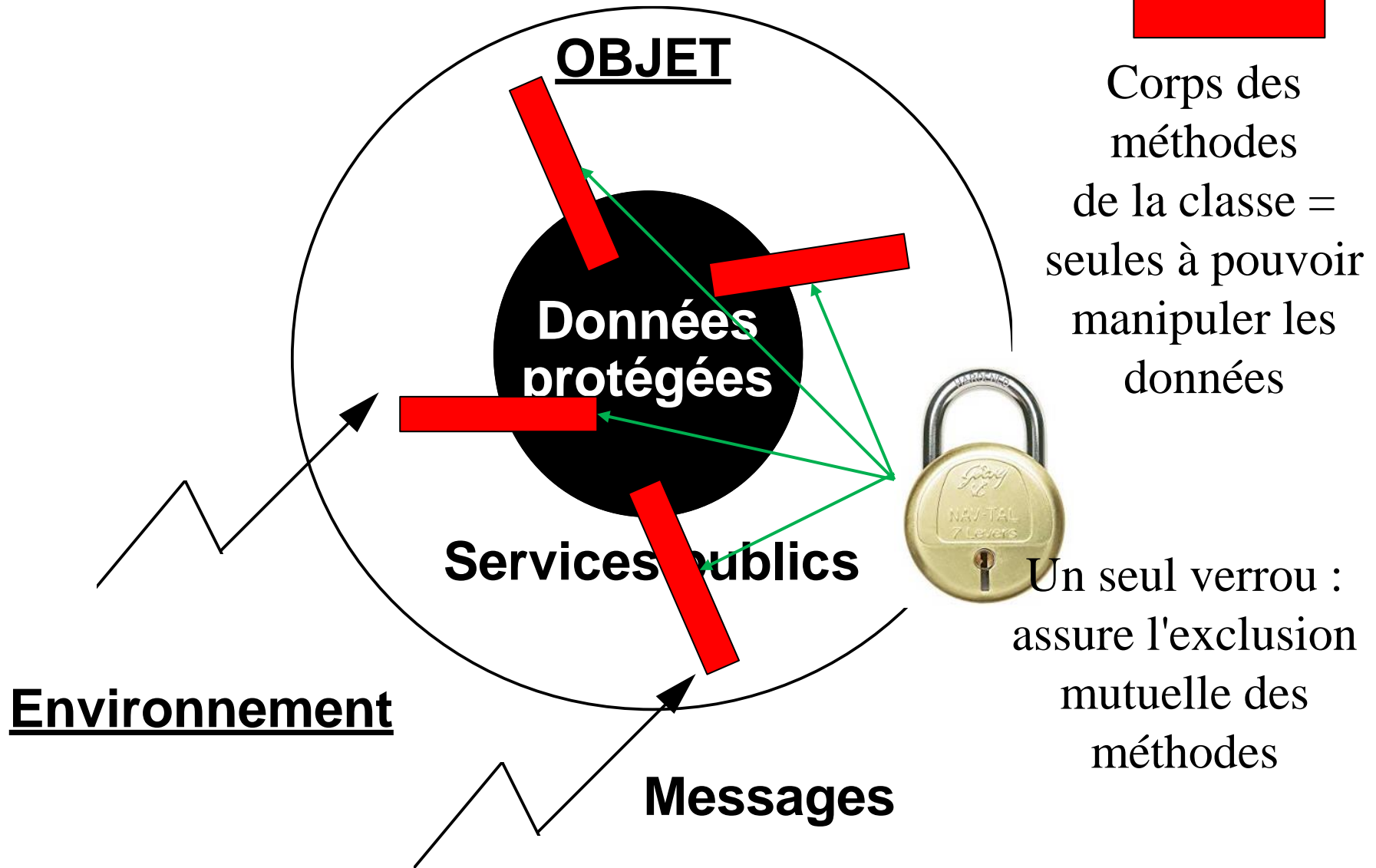


Substituabilité du noyau sans impacter les clients.

L 'objet : notion d'encapsulation



La classe "synchronized"



La classe "synchronized" Java

- Conclusion :
 - Si on a une encapsulation forte
 - Attention aux "getter" qui laisseraient accéder dans l'objet à ses données en contournant les méthodes de la classe !
 - ✓ Et que toutes les méthodes sont protégées par `unique_lock/lock_guard`
 - ✓ Alors, la classe est thread-safe par construction !
- Un pattern simple,
 - Facile à mettre en place sans trop réfléchir
- Mais correct,
 - évite donc les fautes de concurrence dues aux data race structurellement
 - rappelons que ce sont des fautes subtiles, et pas forcément révélabale via des tests

```
class Person {
private:
    std::string name;
    int age;

public:
    Person(const std::string& name, int age) : name(name), age(age) {}

    void updateName(const std::string& newName) {
        if (newName.length() > 0) {
            name = newName;
        }
    }

    void updateAge(int newAge) {
        if (newAge > age) {
            age = newAge;
        }
    }

    std::string getInfo() const {
        return "Name: " + name + ", Age: " + std::to_string(age);
    }

    bool canVote() const {
        return age >= 18;
    }
};
```

MultiThread-Safe

- Les méthodes d'une classe manipulent les mêmes données
 - Comment garantir un accès cohérent aux attributs de chaque instance d'une classe ?
 - Atomic ne fournit pas toutes les garanties utiles
 - Une section critique contient plusieurs actions, le grain est arbitraire

```
class Person {
```

```
private:
```

```
    std::string name;
```

```
    int age;
```

```
    mutable std::mutex mtx; // Mutable so it can be locked in const methods
```

```
public:
```

```
    Person(const std::string& name, int age) : name(name), age(age) {}
```

```
    void updateName(const std::string& newName) {
```

```
        std::lock_guard<std::mutex> lock(mtx);
```

```
        if (newName.length() > 0) {
```

```
            name = newName;
```

```
        }
```

```
    }
```

```
    void updateAge(int newAge) {
```

```
        std::lock_guard<std::mutex> lock(mtx);
```

```
        if (newAge > age) {
```

```
            age = newAge;
```

```
        }
```

```
    }
```

```
    std::string getInfo() const {
```

```
        std::lock_guard<std::mutex> lock(mtx);
```

```
        return "Name: " + name + ", Age: " + std::to_string(age);
```

```
    }
```

```
    bool canVote() const {
```

```
        std::lock_guard<std::mutex> lock(mtx);
```

```
        return age >= 18;
```

```
    }
```

Classe MT safe exemple

// source M. Freiholz

class Cache

```
{  
    mutable std::mutex _mtx;  
    std::map<int, std::shared_ptr<CacheData> > _map;  
public:  
    std::shared_ptr<CacheData> get(int key) const  
    {  
        std::unique_lock<std::mutex> l(_mtx);  
        std::map<int, std::shared_ptr<CacheData> >::const_iterator it;  
        if ((it = _map.find(key)) != _map.end())  
        {  
            auto val = it->second;  
            return val;  
        }  
        return std::shared_ptr<CacheData>();  
    } // auto unlock (unique lock, RAII)  
    void insert(int key, std::shared_ptr<CacheData> value)  
    {  
        std::unique_lock<std::mutex> l(_mtx);  
        _map.insert(std::make_pair(key, value));  
    } // auto unlock (unique lock, RAII)  
};
```

NB: mutable pour contrer le const

MultiThread-Safe

- Solution générique :
 - Un Mutex par instance, barrière commune à l'entrée, libérée en sortie pour les opérations membres (cf. moniteurs Java)
- Le plus souvent donc :
 - Un attribut typé **mutex**
 - Si les méthodes de la classe s'invoquent les unes les autres => utiliser un `recursive_mutex`
 - Le plus souvent il faut le déclarer **mutable** pour les accès dans les méthodes `const`
 - Instanciation d'un **unique_lock** dans chaque méthode
 - Simule + ou – le **synchronized** de Java
 - Attention en Java sémantique ré-entrante
 - `recursive_mutex` en c++ possible

Protection de Structures de Données

- La protection d'une variable structurée partagée :
 - Protéger les accès en écriture : pas d'autre écriture, ni lecture
 - Autoriser les accès partagés en lecture
 - Structures immuables, `shared_ptr`
 - Utilisation de locks spécifiques : reader/writer lock
- Attention à utiliser le même lock pour tous les accès à la variable/instance de classe
 - Mais un seul lock (big fat lock) produit une forte contention

Protection de Structures de Données

- On peut utiliser plusieurs locks de grain fin,
 - e.g. un lock par bucket de la table de hash,
 - Et même simplement des atomic dans certains cas
 - Structures « lock-free »
- Des stratégies distinguant les écritures et les lectures sont utiles
 - Lecteurs concurrents = OK
 - Supporté par `shared_mutex` dans C++14/17
 - Aidé par les annotations « `const` » en C++

Interblocages

Lock multiples et Deadlock

- Problème grave possible :
interblocage
 - Existence de la
possibilité d'acquérir une
autre ressource quand on
en détient une
 - Existence de cycles
possibles dans les
acquisitions
- La variante triviale
 - invoquer "inc()" crée un
deadlock même avec 1
seul thread !
 - =>utilisez
std::recursive_mutex

```
class NonReentrantDeadlock {  
    int a = 0;  
    int b = 0;  
    std::mutex m;  
public:  
    void inc () {  
        m.lock();  
        if (empty()) {  
            a++;  
        } else {  
            b++;  
        }  
        m.unlock();  
    }  
    bool empty() {  
        m.lock();  
        bool b = (a==0 && b==0);  
        m.unlock();  
        return b;  
    }  
}
```

Exemple (deadlock)

```
void task_a () {  
    foo.lock();  
    bar.lock();  
    std::cout << "task a\n";  
    foo.unlock();  
    bar.unlock();  
}
```

```
void task_b () {  
    bar.lock();  
    foo.lock();  
    std::cout << "task b\n";  
    bar.unlock();  
    foo.unlock();  
}
```

```
int main ()  
{    std::thread th1 (task_a);  
    std::thread th2 (task_b);  
  
    th1.join();  
    th2.join();  
}
```

Généralisation

- Pour construire un deadlock (non trivial) il faut que :
 - Les threads acquièrent plusieurs locks en séquence avant de les relâcher
 - On arrive à construire un **cycle** de dépendances où T0 attend T1 qui attend T0...
 - La commutation intervient au "mauvais" moment
 - C'est une faute de concurrence donc pas forcément immédiat à reproduire (indéterminisme)
- Comment éviter cette situation de façon robuste ?
 - Garantir qu'il y a toujours au moins un thread qui peut progresser !
 - Il va donc à terme finir d'acquérir ses locks, faire sa section critique, puis libérer les locks, permettant au suivant de progresser...

Stratégie : ordre total sur les locks

- On se donne un ordre total sur les locks
 - $10 < 11 < 12 \dots$
 - Les threads qui doivent acquérir plusieurs locks **doivent** respecter cet ordre
 - e.g. T0 veut 13, 11, 15 \Rightarrow il acquiert dans l'ordre 11,13,15
 - e.g. T1 veut 10, 15, 13 \Rightarrow il acquiert dans l'ordre 10,13,15
- ✓ Effet : le thread le plus avancé dans ses acquisition (i.e. qui détient le lock avec l'indice le plus élevé) peut forcément progresser
 - Eventuellement il se fait doubler,
 - e.g. T0 prend 11, T1 prend 10 puis 13 (et double T0)
 - mais toujours un thread qui détient le lock d'indice le plus élevé
 - T1 ne se fait plus doubler par T0 s'il possède 13, donc seul T1 peut acquérir 15

Lock multiples

- Solution simple et résistante
 - Ordonner les locks avec un ordre total : e.g. $\text{foo} < \text{bar}$
 - Les acquisitions de locks suivent toutes le même ordre
- Le système peut ordonner totalement les locks
 - Fonction `std::lock()` nombre de locks arbitraires
 - La demande lock mentionne tous les locks utilisés dans l'ordre du système
- Attention à ne pas mélanger ce mécanisme et d'autres acquisitions progressives de locks, selon un ordre « à la main »...

Example (lock multiple)

```
void task_a () {  
    std::lock (foo,bar);  
    std::cout << "task a\n";  
    foo.unlock();  
    bar.unlock();  
}
```

```
void task_b () {  
    std::lock (bar,foo);  
    std::cout << "task b\n";  
    bar.unlock();  
    foo.unlock();  
}
```

```
int main ()  
{  
    std::thread th1 (task_a);  
    std::thread th2 (task_b);  
  
    th1.join();  
    th2.join();  
  
    return 0;  
}
```


Try lock, Timed lock

- Synchronisations plus faibles
 - Possibilité d'agir si le lock n'est pas disponible
 - Possibilité de timeout
- `try_lock` :
 - Acquiert le lock si disponible (comme `lock`) et rend -1, sinon
 - Rend l'indice du lock ayant échoué dans la version multiple
- `timed_lock`
 - Se bloque en attente, mais pour une durée limitée.
 - Rend `true` ou `false` selon que le lock a été acquis

Try_lock

```
std::mutex foo,bar;
```

```
void task_a () {  
    foo.lock();  
    std::cout << "task a\n";  
    bar.lock();  
    // ...  
    foo.unlock();  
    bar.unlock();  
}
```

```
int main ()  
{  
    std::thread th1 (task_a);  
    std::thread th2 (task_b);
```

```
    th1.join();  
    th2.join();
```

```
    return 0;  
}
```

```
void task_b () {  
    int x = try_lock(bar,foo);  
    if (x==1) {  
        std::cout << "task b\n";  
        // ...  
        bar.unlock();  
        foo.unlock();  
    }  
    else {  
        std::cout << "[task b failed: mutex " <<  
        (x?"foo":"bar") << " locked]\n";  
    }  
}
```

Unique_lock

- Unique_lock : utilisation avancée
 - De base, similaire à un **lock_guard** mécanisme RAII
 - Le unique_lock acquiert le mutex (**lock**) à la déclaration, le relâche (**unlock**) à la destruction.
 - Acquisition du lock à la déclaration
 - Sauf si mode « différé »
 - `std::unique_lock<std::mutex> lock(mutex, std::defer_lock);`
 - MAIS on peut encore faire lock/unlock sur l'objet
 - Il sera unlock en fin de vie de l'objet
- Permet de s'approprier le mutex
 - Utile en combinaison avec les conditions

Unique_lock

(exemple artificiel exhibant la syntaxe)
(std::defer pour utiliser unique_lock
ET la version lock « ordre système »)

```
std::mutex m_a, m_b, m_c;
int a=1, b=1, c = 1;
void update()
{
    { // protège a
        std::unique_lock<std::mutex> lk(m_a);
        a++;
    }
    { // on ne lock pas tout de suite
        std::unique_lock<std::mutex> lk_b(m_b, std::defer_lock);
        std::unique_lock<std::mutex> lk_c(m_c, std::defer_lock);
        std::lock(lk_b, lk_c); //ordre système
        int tmp = c;
        c = b+c;
        b = tmp;
    }
}
```

```
int main()
{
    std::vector<std::thread> threads;
    for (unsigned i = 0; i < 12; ++i)
        threads.emplace_back(update);

    for (auto& t: threads)
        t.join();

    std::cout << a << "th and " << a+1 <<
    "th Fibonacci numbers: "
        << b << " and " << c << "\n";
}
```

On veut échanger b et c
Typiquement => lock de b et c

Condition, notifications, attentes

Section Critique vs Notification

- Section critique, exclusion mutuelle :
 - Éviter aux thread d'écraser le travail des autres
 - Permettre de travailler sur des données partagées
- Notifications, conditions
 - Attendre (sans CPU) qu'un autre thread ait réalisé un traitement
 - Notifier quand un travail est terminé
 - *Coopération entre les threads !*
 - Mise en place de variables partagées, indiquant la fin du traitement qu'un thread teste et l'autre met à jour
 - Pas d'attente active
- Comme il y a nécessairement des variables partagées
 - Nécessité d'utiliser aussi un mutex

Attente et notification sur une condition

- **condition.wait (mutex):**
 - Libère le mutex
 - Bloque l'appelant et l'insère dans une file d'attente associée à la condition
 - A son réveil, il réacquiert atomiquement le mutex
 - Il poursuit son exécution
- **condition.notify_one, condition. notify_all**
 - Réveille un ou tous les threads en attente sur la condition
 - Si plusieurs, ils se réveillent en séquence pour acquérir le mutex un par un
- Le mutex doit être un **unique_lock**
 - Il est lock/unlock par le processus du wait

<condition> :

example

```
// global variables
std::mutex m;
std::condition_variable cv;
std::string data;
bool ready = false;
bool processed = false;

void worker_thread()
{
    std::unique_lock<std::mutex>
        lk(m);
    while (!ready)
        cv.wait(lk);
    data += " after processing";
    // Send data back to main()
    processed = true;
    lk.unlock();
    cv.notify_one();
}
```

```
int main() {
    std::thread worker(worker_thread);
    data = "Example data";

    {
        std::unique_lock<std::mutex> lk(m);
        ready = true;
        std::cout << "main() signals data ready \n";
    }
    cv.notify_one();
    // wait for the worker
    {
        std::unique_lock<std::mutex> lk(m);
        while (!processed)
            cv.wait(lk);
    }
    std::cout << "Back in main(), data = " << data ;
    worker.join();
}
```

Si **processed**, ne pas faire wait

Attendre une Condition

- Toujours mettre le test sous protection du mutex
 - `if (! ready) { unique_lock<mutex> l(m) ; cv.wait(l); }`
 - Incorrect !
 - Problème : `ready` est testé en dehors du mutex (pas atomique)
 - `{ unique_lock<mutex> l(m) ;`
`if (! ready) { cv.wait(l); }`
`} // fin bloc déclaration du lock`
 - OK, en général c'est la bonne option :
 - On attend une condition qu'un autre thread doit activer
 \Rightarrow mutex est nécessaire et logique.

Attendre une Condition (boucle)

- En général il faut tester la condition dans une boucle
 - `unique_lock<mutex> l(m) ;`
 - `if (! ready) { cv.wait(l); } //PROBLEME`
 - `doIt() ;`
 - `ready = false;`
- Si deux threads exécutent ce code, on peut faire *doIt()* sans que **ready**
 - Si `notifyAll` (on se fait doubler) ou globalement, si *ready* est redevenu faux
 - ✓ => Refaire le test à chaque réveil sur la condition = `while`
 - `unique_lock<mutex> l(m) ;`
 - `while (! ready) { cv.wait(l); } // OK`
 - `doIt() ;`
 - `ready = false;`

Attendre une Condition : version lambda

- Vu qu'en général la boucle while est la bonne option
 - Offrir cette version directement aux clients
- Version lambda de wait (*wait for test to be true*):
 - `cond.wait(lock, test)`
 - \Leftrightarrow
 - `while (!test) cond.wait(lock)`
 - Test est souvent une lambda : `[&]()->bool{ return ready; }`
 - Attention à la négation : on attend que la condition devienne vraie
- Souvent préférer cette version qui évite certaines fautes fréquentes.

<condition> :

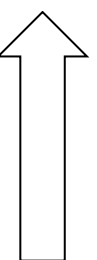
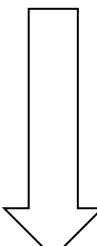
lambda

```
// global variables
std::mutex m;
std::condition_variable cv;
std::string data;
bool ready = false;
bool processed = false;

void worker_thread()
{
    std::unique_lock<std::mutex>
        lk(m);
    cv.wait(lk, [&]() { return ready; });
    data += " after processing";
    // Send data back to main()
    processed = true;
    lk.unlock();
    cv.notify_one();
}
```

```
int main() {
    std::thread worker(worker_thread);
    data = "Example data";
    {
        std::lock_guard<std::mutex> lk(m);
        ready = true;
        std::cout << "main() signals data ready \n";
    }
    cv.notify_one();
    // wait for the worker
    {
        std::unique_lock<std::mutex> lk(m);
        cv.wait(lk, [&]() { return processed; });
    }

    std::cout << "Back in main(), data = " << data;
    worker.join();
}
```



Garanties : wait dans une boucle, test protégé par mutex

```
// globales
queue<int> queue;
mutex m;
condition_variable cond_var;

int main()
{
    std::thread prod1(producer);
    std::thread prod2(producer);
    std::thread cons1(consumer);
    std::thread cons2(consumer);

    prod1.join(); prod2.join();
    done = true;
    cons1.join(); cons2.join();
});
```

Producteurs/Consommateur v1 :
Wait et la terminaison :

```
void producer() {
    for (int i = 0; i < 5; ++i) {
        this_thread::sleep_for(1s);
        unique_lock<mutex> lock(m);
        cout << "producing " << i << '\n';
        queue.push(i);
        cond_var.notify_one();
    }
}

void consumer() {
    unique_lock<mutex> lock(m);
    while (!queue.empty() || !done) {
        while (queue.empty()) {
            cond_var.wait(lock);
        }
        if (!queue.empty()) {
            std::cout << "consuming " <<
                queue.front() << '\n';
            queue.pop();
        }
    }
}
```

consommateur bloqué à la fin du programme

```
// globales
queue<int> queue;
mutex m;
condition_variable cond_var;
bool done = false;

int main()
{
    std::thread prod1(producer);
    std::thread prod2(producer);
    std::thread cons1(consumer);
    std::thread cons2(consumer);

    prod1.join(); prod2.join();
    done = true;
    cond_var.notify_all();
    cons1.join(); cons2.join();
});
```

```
void producer() {
    for (int i = 0; i < 5; ++i) {
        this_thread::sleep_for(1s);
        unique_lock<mutex> lock(m);
        cout << "producing " << i << '\n';
        queue.push(i);
        cond_var.notify_one();
    }
}

void consumer() {
    unique_lock<mutex> lock(m);
    while (!queue.empty() || !done) {
        while (queue.empty()) {
            cond_var.wait(lock);
        }
        if (!queue.empty()) {
            std::cout << "consuming " <<
                queue.front() << '\n';
            queue.pop();
        }
    }
}
```

Mais le programme ne se termine pas toujours...cons bloqué sur wait⁹⁵

```
// globales
queue<int> queue;
mutex m;
condition_variable cond_var;
bool done = false;

int main()
{
    std::thread prod1(producer);
    std::thread prod2(producer);
    std::thread cons1(consumer);
    std::thread cons2(consumer);

    prod1.join(); prod2.join();
    done = true;
    cond_var.notify_all();
    cons1.join(); cons2.join();
};
```

```
void producer() {
    for (int i = 0; i < 5; ++i) {
        this_thread::sleep_for(1s);
        unique_lock<mutex> lock(m);
        cout << "producing " << i << '\n';
        queue.push(i);
        cond_var.notify_one();
    }
}

void consumer() {
    unique_lock<mutex> lock(m);
    while (!queue.empty() || !done) {
        while (queue.empty() && !done) {
            // on commute ici, avant wait
            cond_var.wait(lock); // attente infinie
        }
        if (!queue.empty()) {
            std::cout << "consuming " <<
                queue.front() << '\n';
            queue.pop();
        }
    }
}
```

Attente relaxée, mais done mal protégé dans main

```
// globales
queue<int> queue;
mutex m;
condition_variable cond_var;
bool done = false;

int main()
{
    std::thread prod1(producer);
    std::thread prod2(producer);
    std::thread cons1(consumer);
    std::thread cons2(consumer);

    prod1.join(); prod2.join();
    {
        unique_lock<mutex> l(m);
        done = true; cond_var.notify_all();
    }
    cons1.join(); cons2.join();
};
```

Version OK : attente relaxée

```
void producer() {
    for (int i = 0; i < 5; ++i) {
        this_thread::sleep_for(1s);
        unique_lock<mutex> lock(m);
        cout << "producing " << i << '\n';
        queue.push(i);
        cond_var.notify_one();
    }
}

void consumer() {
    unique_lock<mutex> lock(m);
    while (!queue.empty() || !done) {
        while (queue.empty() && !done) {
            cond_var.wait(lock);
        }
        if (!queue.empty()) {
            std::cout << "consuming " <<
                queue.front() << '\n';
            queue.pop();
        }
    }
}
```


Condition : conclusion

- Des versions **wait** plus faibles avec timeout sont possibles
 - `wait_for(duree max), wait_until(date)`
- Des versions pour n'importe quel mutex (pas juste un `unique_lock`)
 - `condition_variable_any`
- Avec les conditions et les mutex on construit les synchronisations
 - Ce sont les briques de base
 - Atomic vient compléter à niveau plus fin, et est utilisé pour implanter les mutex et conditions
- Ne jamais faire d'attente active
 - Utiliser des conditions
 - Protéger les variables partagées avec des mutex
- Retour sur classe MT-safe
 - Opérations bloquantes implantées par des conditions

Classes de synchronisation

Notifications : le pattern while/notify_all

- C'est le pattern recommandé
 - Dans une classe déjà MT-safe
 - On souhaite avoir des comportements bloquants
 - On ajoute un "mutable std::mutex m" (ou recursive_mutex)
 - On ajoute une "condition_variable cond"
 - On ne contrôle pas combien de clients accèdent à l'objet

```
{ unique_lock<mutex> l(m);  
    cond.wait(l, [&]() { return this->ok(); }); // ou while (! ok())  
    // this->ok() is true  
    this->modify();  
}
```

```
cond.notify_all(); // à chaque modification utile
```

Toujours commencer par cette stratégie

Sémantique de wait

- On détient forcément le verrou quand on invoque wait
e.g.

```
{ unique_lock<mutex> l(m);  
    cond.wait(l);  
}
```
- Dans le wait, avant de dormir, on relâche le verrou
 - Attention on est sorti de la section critique
 - Nécessaire pour que le partenaire puisse accéder et mettre à jour l'état

```
cond.wait(l) {  
    // Ce bloc est exécuté de façon atomique  
    {  
        // On s'ajoute aux threads en attente  
        cond.waitQueue.add(currentThread());  
        // on relâche le verrou  
        l.unlock();  
        // on s'endort en attendant une notification  
        cond.sleepUntilNotify();  
    }  
    // on a été réveillé  
    // on reprend le lock avant de sortir du wait  
    l.lock();  
    return;  
}
```

Conceptuellement/pseudocode

Exemple : Sémaphore

- Sémaphore : `std::counting_semaphore<>`
 - Une classe de synchronisation
 - Proposée par Dijkstra vers '65
 - Standard POSIX entre processus
- Sémantique
 - `acquire()` peut bloquer => lève `InterruptedException`
 - Le compteur "permits"
 - Est décrémenté sur `acquire()` = P
 - Est incrémenté() sur `release()` = V
 - ne doit jamais tomber en négatif (bloquant sinon)



P = Prolaag : Probeer Verlaag
V = Verhoog

```
class Semaphore {  
public:  
    explicit Semaphore(size_t initialPermits)  
        : permits_(initialPermits) {}  
  
    void acquire() {  
        std::unique_lock<std::mutex>  
lock(mutex_);  
        condition_.wait(lock, [this] { return  
permits_ > 0; });  
        --permits_;  
    }  
  
    void release() {  
        {  
            std::lock_guard<std::mutex>  
lock(mutex_);  
            ++permits_;  
        }  
        condition_.notify_one();  
    }  
  
private:  
    std::mutex mutex_;  
    std::condition_variable condition_;  
    size_t permits_;  
};
```

Utilisation d'un sémaphore (1)

- On peut l'utiliser comme un Lock
 - Initialisé à 1
 - On remplace
 - `lock.lock()` par `sem.acquire()`
 - `lock.unlock()` par `sem.release()`
 - ✓ Fonctionne, permet d'implanter une exclusion mutuelle/section critique
- Ce n'est pas exactement comme un mutex
 - En particulier ce n'est pas nécessairement le propriétaire qui release
 - cf. `std::binary_semaphore`

Ping Pong et Semaphore ?

- On utilise deux sémaphores dont la valeur max sera 1
 - NB: avec un seul, on retrouve tous les problèmes évoqués en début de séance quand on est limités aux verrous
 - L'un est initialement passant (permits=1, pour Ping)
 - L'un est initialement bloquant (permits=0, pour Pong)

```
class PingPongSemaphore {
```

```
    std::binary_semaphore pingSem_;
```

```
    std::binary_semaphore pongSem_;
```

```
public:
```

```
    PingPongSemaphore() : pingSem_(1), pongSem_(0) {}
```

```
    void ping() {
```

```
        pingSem_.acquire();
```

```
        std::cout << "Ping\n";
```

```
        pongSem_.release();
```

```
    }
```

```
    void pong() {
```

```
        pongSem_.acquire();
```

```
        std::cout << "Pong\n";
```

```
        pingSem_.release();
```

```
    }
```

```
};
```

NB: On ne peut pas le faire avec deux mutex :
seul le propriétaire peut unlock

Une file bornée avec Semaphore ?

```
template <typename T>
class BoundedBuffer {
    size_t capacity_;
    Semaphore emptySlots_;
    Semaphore fullSlots_;
    std::mutex mutex_;
    std::deque<T> buffer_;
```

```
public:
    explicit BoundedBuffer(size_t capacity)
        : capacity_(capacity), emptySlots_(capacity),
          fullSlots_(0) {}

    void produce(const T& item) {
        emptySlots_.acquire();
        {
            std::lock_guard<std::mutex> lock(mutex_);
            buffer_.push(item);
        }
        fullSlots_.release();
    }
```

La file :

- produce est bloquant si plein
- consume est bloquant si vide

```
T consume() {
    fullSlots_.acquire();
    T item;
    {
        std::lock_guard<std::mutex>
lock(mutex_);
        item = buffer_.front();
        buffer_.pop();
    }
    emptySlots_.release();
    return item;
};
```


File bornée + sémaphore

- Un exemple assez classique d'utilisation de sémaphore
 - De nombreuses autres façons de traiter le problème existent (on en verra en TD)
 - Un sémaphore compte les emplacements libres (empêche de produire si plein)
 - Un sémaphore compte les emplacements occupés (empêche de consommer si vide)
 - Chaque opération acquiert (P) un des sémaphores et release (V) l'autre
 - Ce contrôle d'accès ne se substitue pas aux sections critiques (sous mutex donc) pour l'accès à la queue sous-jacente
- La classe de synchronisation Semaphore :
 - rend les autres codes plus simples
 - centralise la réflexion sur data race/notify etc...

Le Rendez-vous

- On a N threads qui collaborent
 - Le calcul se passe par phases
 - Il faut finir phase 1 avant de démarrer phase 2
- Les threads font un rendez-vous
 - Le N-eme débloquent les autres
 - Tout le monde peut entamer la phase suivante
- Pour une fois on wait sous if !
 - Sinon ce serait faux !
 - La barrière est réutilisable

```
class Barrier {
    std::mutex mutex_;
    std::condition_variable condition_;
    const size_t totalThreads_;
    size_t waitingThreads_;
public:
    explicit Barrier(size_t totalThreads)
        : totalThreads_(totalThreads),
        waitingThreads_(0) {}

    void await() {
        std::unique_lock<std::mutex>
lock(mutex_);
        ++waitingThreads_;
        if (waitingThreads_ < totalThreads_) {
            condition_.wait(lock);
        } else {
            waitingThreads_ = 0; // Reset for
potential reuse
            condition_.notify_all();
            lock.unlock();
        }
    }
};
```

Barrier (2)

```
class BarrierWorker {  
public:  
    BarrierWorker(Barrier& barrier, int workerId)  
        : barrier_(barrier), workerId_(workerId) {}
```

```
void operator() {  
    performTask("Phase 1");  
    barrier_.await(); // Synchronize at the barrier  
    performTask("Phase 2");  
}
```

private:

```
void performTask(const std::string& phase) {  
    std::cout << "Worker " << workerId_ << " performing " << phase << "\n";  
    // Simulate work  
    std::this_thread::sleep_for(std::chrono::milliseconds(rand() % 100));  
}
```

```
Barrier& barrier_;  
int workerId_;  
};
```

- Exemple de worker
 - On synchronise sur la barrière entre les deux phases

Barrier (3)

- Exemple de main
 - on utilise un foncteur comme argument au ctor de thread

Préférez utiliser

`std::barrier :`

`std::barrier`

`barrier(totalThreads);`

et

`barrier.arrive_and_wait();`

```
int main() {  
    const int totalThreads = 5;  
    Barrier barrier(totalThreads);  
    std::vector<std::thread> workers;  
  
    for (int i = 0; i < totalThreads; ++i) {  
        workers.emplace_back(BarrierWorker(barrier, i));  
    }  
  
    for (auto& worker : workers) {  
        worker.join();  
    }  
  
    return 0;  
}
```

Classes de synchronisation

- Classe de synchronisation
 - MT safe : protégée par mutex et ou atomic
 - Certaines opérations vont être bloquantes !
 - On attend les autres
 - En général, présence d'un mutex et d'une condition
- Dans le standard C++20 :
 - `counting_semaphore`, `binary_semaphore`
 - `latch`, `barrier`
 - `promise`, `future`, `shared_future`
- Mais toujours pas de collections concurrentes, ni de primitives de plus haut niveau
 - Il faut écrire soi même ou utiliser des libs tierces comme Boost.

La lib Pthread

Positionnement, Historique

- Librairie POSIX pour les threads
 - Standardisée
 - API en C offerte par le système pour réaliser la concurrence
 - Mode « User Space » pour les systèmes sans thread natifs
 - Implantée sur beaucoup d'OS (portabilité)
- Gros impact sur le développement des API de concurrence dans les langages
 - Si ce n'est pas réalisable sur Pthread, problème. E.g. threads dans C11
 - Cf abstractions fournies dans e.g. Java, thread C++11
- La référence : le man, partir de « man 7 pthreads » et naviguer

Les Objets Pthread

Orienté objet:

- ✓ *pthread_t* : identifiant d'une *thread*
- ✓ *pthread_attr_t* : attribut d'une *thread*
- ✓ *pthread_mutex_t* : *mutex* (exclusion mutuelle)
- ✓ *pthread_mutexattr_t* : attribut d'un *mutex*
- ✓ *pthread_cond_t* : variable de condition
- ✓ *pthread_condattr_t* : attribut d'une variable de condition
- ✓ *pthread_key_t* : clé pour accès à une donnée globale réservée
- ✓ *pthread_once_t* : initialisation unique

Fonctions Pthreads

Préfixe

- ✓ Enlever le *_t* du type de l'objet auquel la fonction s'applique.

Suffixe (exemples)

- ✓ *_init* : initialiser un objet.
- ✓ *_destroy* : détruire un objet.
- ✓ *_create* : créer un objet.
- ✓ *_getattr* : obtenir l'attribut *attr* des attributs d'un objet.
- ✓ *_setattr* : modifier l'attribut *attr* des attributs d'un objet.

Exemples :

- ✓ *pthread_create* : crée une thread (objet *pthread_t*).
- ✓ *pthread_mutex_init* : initialise un objet du type *pthread_mutex_t*.

Gestion des Threads

Une *Pthread* :

- ✓ est identifiée par un *ID* unique.
- ✓ exécute une fonction passée en paramètre lors de sa création.
- ✓ possède des attributs.
- ✓ peut se terminer (*pthread_exit*) ou être annulée par une autre thread (*pthread_cancel*).
- ✓ peut attendre la fin d'une autre thread (*pthread_join*).

Une *Pthread* possède son propre masque de signaux et signaux pendants.

La création d'un processus donne lieu à la création de la thread *main*.

- ✓ Retour de la fonction *main* entraîne la terminaison du processus et par conséquent de toutes les threads de celui-ci.

Création des Threads

Création d'une thread avec les attributs `attr` en exécutant `fonc` avec `arg` comme paramètre :

```
int pthread_create(pthread_t *tid, pthread_attr_t *attr,  
                  void * (*fonc) (void *), void *arg);
```

- ✓ **attr** : si NULL, la thread est créée avec les attributs par défaut.
- ✓ **code de renvoi** :
 - 0 en cas de succès.
 - En cas d'erreur une valeur non nulle indiquant l'erreur:
 - EAGAIN : manque de ressource.
 - EPERM : pas la permission pour le type d'ordonnancement demandé.
 - EINVAL : attributs spécifiés par *attr* ne sont pas valables.

Création : paramètres

- Un seul argument prévu, typé « void * »
 - Créer un struct qui héberge tous les paramètres nécessaires
 - struct threadargs { int x; double z; ... }
 - Passer son adresse à pthread create
 - Commencer la fonction du thread par un cast :
 - struct threadargs * argTypé = (struct threadargs *) arg;
- La valeur de retour du thread est un int
 - Comme un processus
 - En général, plutôt passer l'endroit où écrire le résultat dans les arguments

Thread principale x Threads annexes

La création d'un processus donne lieu à la création de la *thread principale (thread main)*.

- ✓ Un retour à la fonction *main* entraîne la terminaison du processus et par conséquent la terminaison de toutes ses threads.

Une thread créée par la primitive *pthread_create* dans la fonction *main* est appelée une *thread annexe*.

- ✓ Terminaison :
 - Retour de la fonction correspondante à la thread ou appel à la fonction *pthread_exit*.
 - aucun effet sur l'existence du processus ou des autres threads.
- ✓ L'appel à *exit* ou *_exit* par une thread annexe provoque la terminaison du processus et de toutes les autres threads.

Exclusion Mutuelle – Mutex (2)

Création/Initialisation (2 façons) :

- ✓ **Statique:**

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

- ✓ **Dynamique:**

```
int pthread_mutex_init(pthread_mutex_t *m, pthread_mutex_attr *attr);
```

- Attributs :
 - initialisés par un appel à :

```
int pthread_mutexattr_init(pthread_mutex_attr *attr);
```
- NULL : attributs par défaut.

- **Exemple :**

```
pthread_mutex_t sem;  
/* attributs par défaut */  
pthread_mutex_init(&sem, NULL);
```

Exclusion Mutuelle (3)

Destruction :

```
int pthread_mutex_destroy (pthread_mutex_t *m);
```

Verrouillage :

```
int pthread_mutex_lock (pthread_mutex_t *m);
```

- Bloquant si déjà verrouillé

```
int pthread_mutex_trylock (pthread_mutex_t *m);
```

- Renvoie EBUSY si déjà verrouillé

Déverrouillage:

```
int pthread_mutex_unlock (pthread_mutex_t *m);
```

Exemple 7 - exclusion mutuelle

```
#define _POSIX_SOURCE 1
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

pthread_mutex_t mutex =
PTHREAD_MUTEX_INITIALIZER;
int cont =0;

void *sum_thread (void *arg) {
    pthread_mutex_lock (&mutex);
    cont++;
    pthread_mutex_unlock (&mutex);

    pthread_exit ((void*)0);
}
```

```
int main (int argc, char ** argv) {
    pthread_t tid;

    if (pthread_create (&tid, NULL, sum_thread,
                        NULL) != 0) {
        printf("pthread_create"); exit (1);
    }

    pthread_mutex_lock (&mutex);
    cont++;
    pthread_mutex_unlock (&mutex);

    pthread_join (tid, NULL);
    printf ("cont : %d\n", cont);

    return EXIT_SUCCESS;
}
```


Les conditions : initialisation (2)

Création/Initialisation (2 façons) :

✓ **Statique:**

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

✓ **Dynamique:**

```
int pthread_cond_init(pthread_cond_t *cond,  
                      pthread_cond_attr *attr);
```

- **Exemple :**

```
pthread_cond_t cond_var;  
/* attributs par défaut */  
pthread_cond_init (&cond_var, NULL);
```

Conditions : attente (3)

```
int pthread_cond_wait(pthread_cond_t *cond,  
                      pthread_mutex_t *mutex);
```

Utilisation:

```
pthread_mutex_lock(&mut_var);  
pthread_cond_wait(&cond_var, &mut_var);  
.....  
pthread_mutex_unlock(&mut_var);
```

- ✓ Une thread ayant obtenu un *mutex* peut se mettre en attente sur une variable condition associée à ce *mutex*.
- ✓ **pthread_cond_wait:**
 - Le mutex spécifié est libéré.
 - La thread est mise en attente sur la variable de condition *cond*.
 - Lorsque la condition est signalée par une autre thread, le *mutex* est acquis de nouveau par la thread en attente qui reprend alors son exécution.

Conditions : notification (4)

Une thread peut signaler une condition par un appel aux fonctions :

```
int pthread_cond_signal(pthread_cond_t *cond);
```

✓ réveil d'une thread en attente sur *cond*.

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

✓ réveil de toutes les threads en attente sur *cond*.

Si aucune thread n'est en attente sur *cond* lors de la notification, cette notification sera perdue.

Exemple 8 - Conditions

```
#define _POSIX_SOURCE 1
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

pthread_mutex_t mutex_fin =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_fin =
    PTHREAD_COND_INITIALIZER;

void *func_thread (void *arg) {
    printf ("tid: %d\n", (int)pthread_self());

    pthread_mutex_lock (&mutex_fin);
    pthread_cond_signal (&cond_fin);
    pthread_mutex_unlock (&mutex_fin);
    pthread_exit ((void *)0);
}
```

```
int main (int argc, char ** argv) {
    pthread_t tid;

    pthread_mutex_lock (&mutex_fin);
    if (pthread_create (&tid , NULL, func_thread,
        NULL) != 0) {
        printf("pthread_create erreur\n"); exit (1);
    }
    if (pthread_detach (tid) !=0 ) {
        printf ("pthread_detach erreur"); exit (1);
    }
    pthread_cond_wait(&cond_fin,&mutex_fin);
    pthread_mutex_unlock (&mutex_fin);

    printf ("Fin thread \n");

    return EXIT_SUCCESS;
}
```

Exemple 9 - Conditions

```
#define _POSIX_SOURCE 1
#include <pthread.h>
...
int flag=0;
pthread_mutex_t m =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond=
    PTHREAD_COND_INITIALIZER;

void *func_thread (void *arg) {
    pthread_mutex_lock (&m);
    while (! flag) {
        pthread_cond_wait (&cond,&m);
    }
    pthread_mutex_unlock (&m);
    pthread_exit ((void *)0);
}
```

```
int main (int argc, char ** argv) {
    pthread_t tid;

    if ((pthread_create (&tid1 , NULL, func_thread,
        NULL) != 0) || (pthread_create (&tid2 ,
        NULL, func_thread, NULL) != 0)) {
        printf("pthread_create erreur\n"); exit (1);
    }

    sleep(1);
    pthread_mutex_lock (&m);
    flag=1;
    pthread_cond_broadcast(&cond,&m);
    pthread_mutex_unlock (&m);

    pthread_join (tid1, NULL);
    pthread_join (tid2, NULL);

    return EXIT_SUCCESS;
}
```

Pthread vs thread C++11

- L'alignement n'est pas parfait
 - Identifiant du pthread,
 - valeur de retour int (comme un processus),
 - cancel de pthread ...
 - Pthread donne accès à plus bas niveau avec les attributes
 - Pthread est l'archétype pour la concurrence
- Le thread c++11 est
 - Portable
 - Relativement facile d'emploi
 - Bibliothèques étendues pour la concurrence, atomic intégré
 - Extensions rapides au fil des standards