

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/>

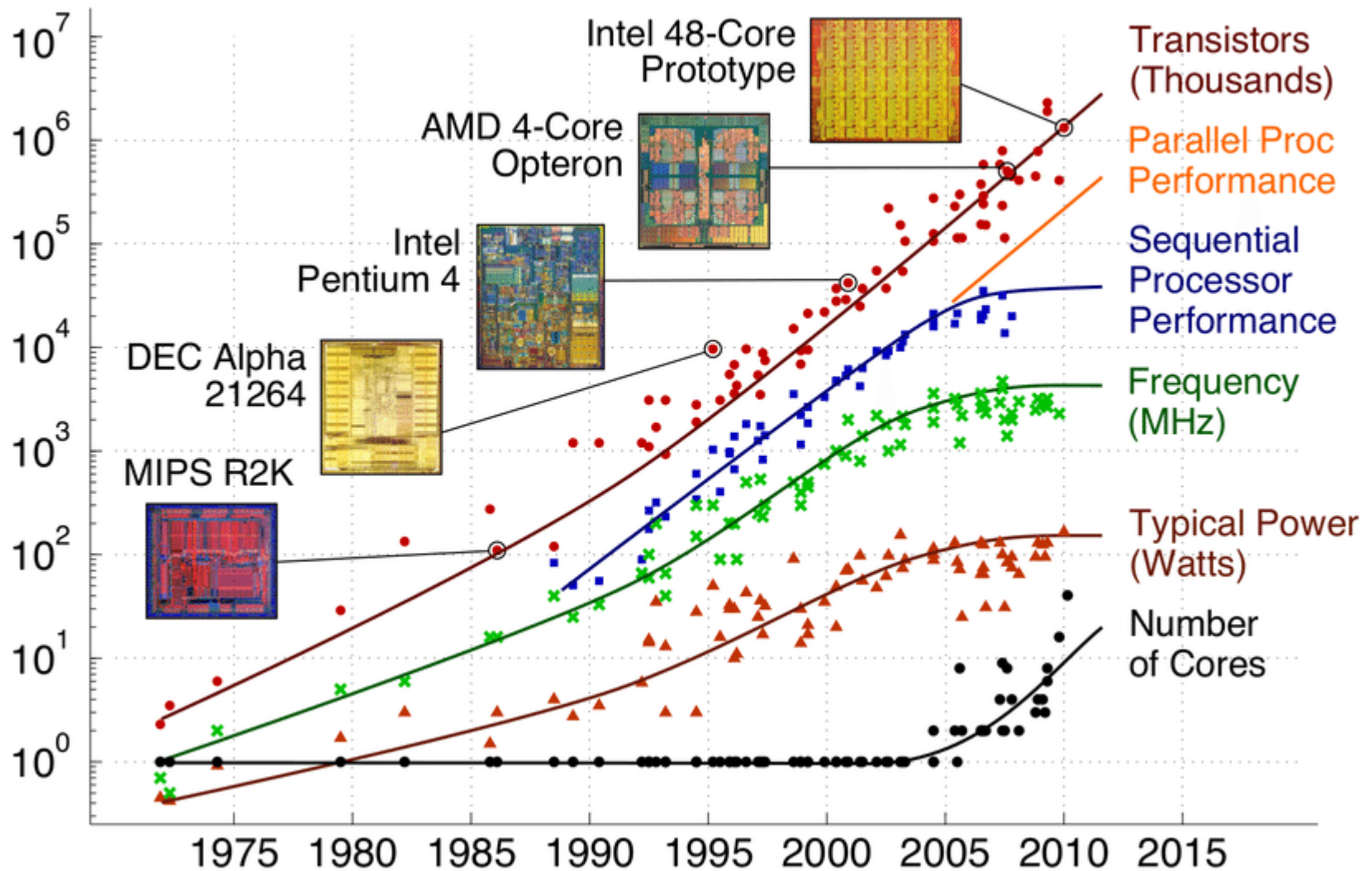
Cours de P.Sens, L.Arantes (pthread)

StackOverflow : Definitive C++ book guide

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

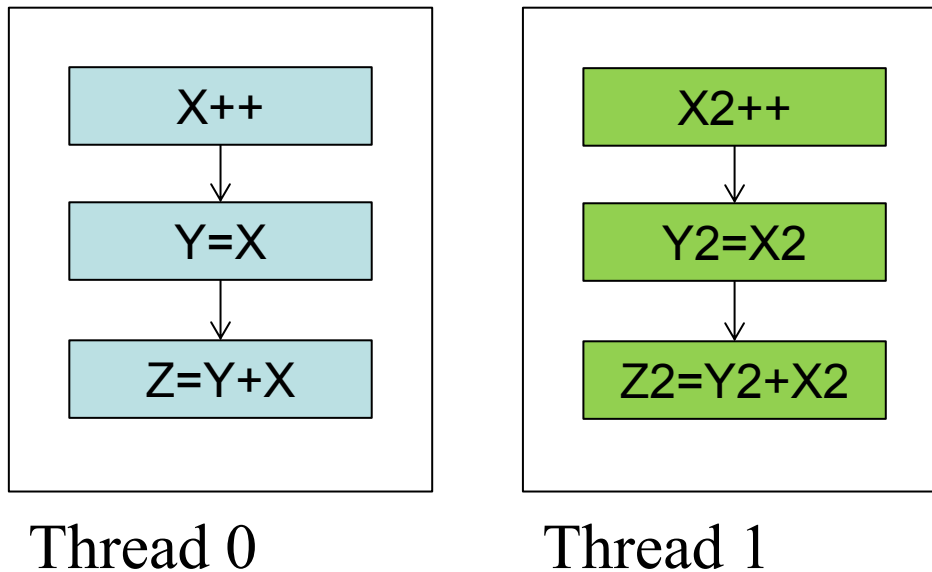
Concurrence, Principes, Problèmes

Evolutions Matérielles



Principes de la concurrence : Causalité

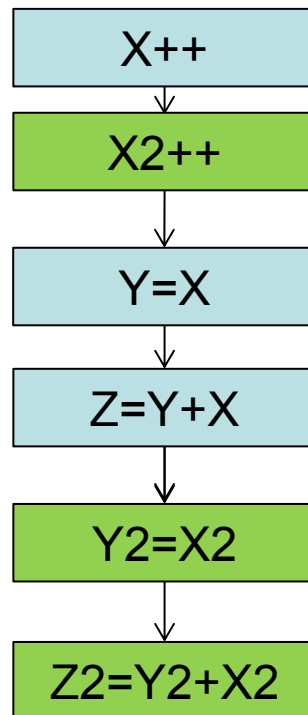
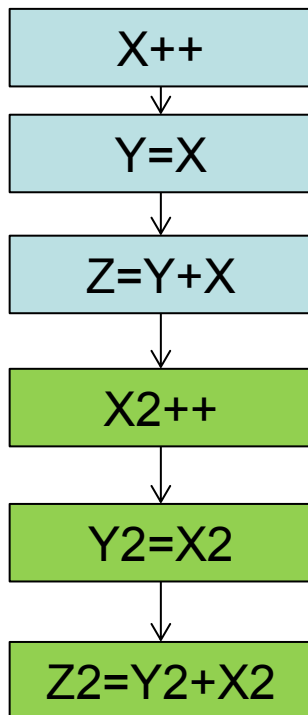
- Le traitement à réaliser est vu comme un arbre d'actions
 - Certaines actions ont des prédécesseurs
 - On parle de dépendance causale



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

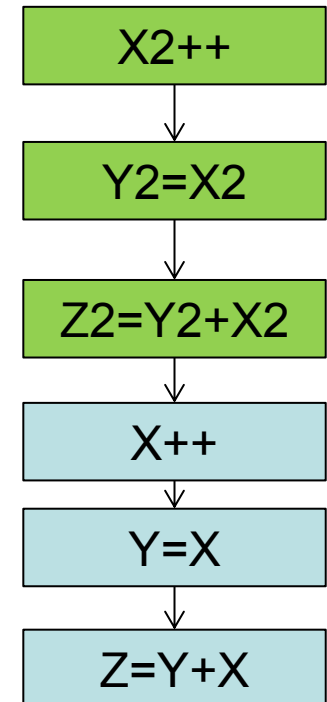
Causalité, Ordre Partiel

- 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



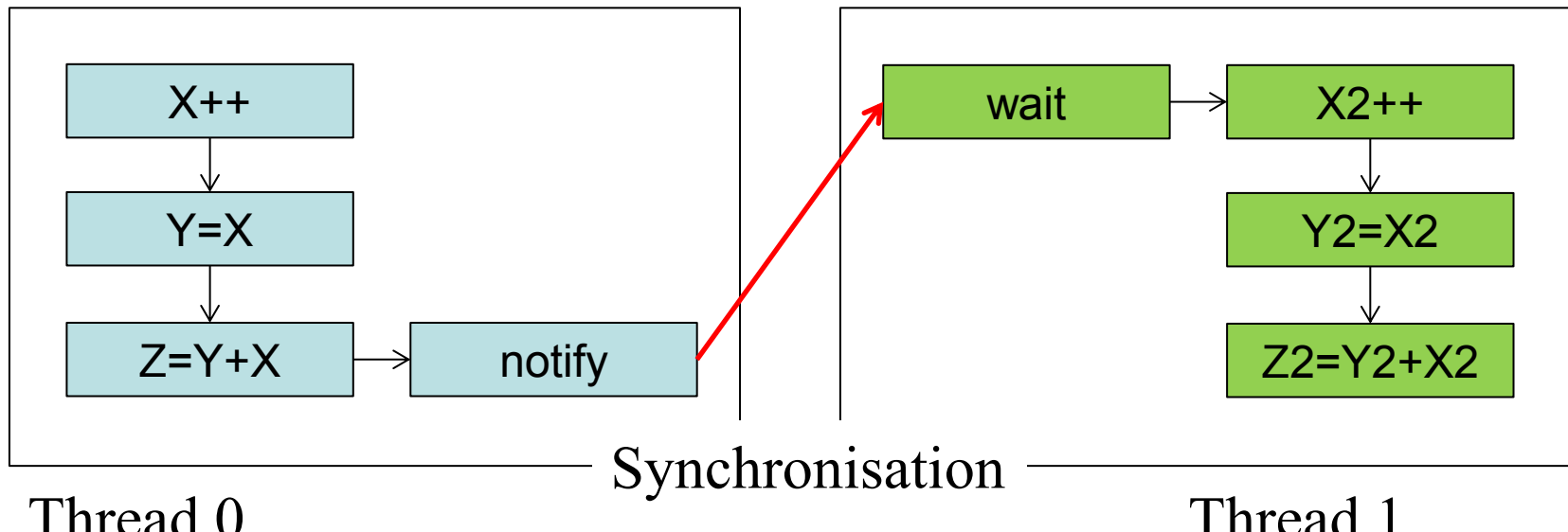
...

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

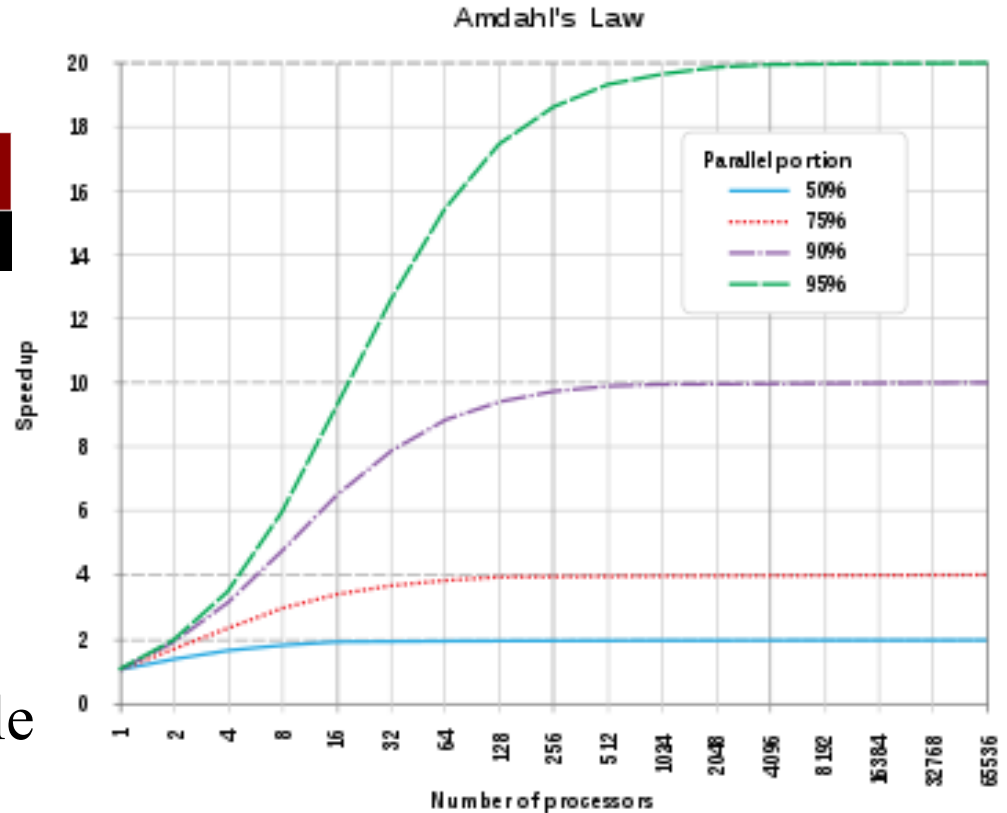


Accélération Théorique Concurrente



- Mesure de loi d'Amdahl : latence
 - Une partie séquentielle
 - Pas d'accélération
 - Une partie parallélisable
 - Proportionnel au nombre de cœurs

On ne peut pas tout paralléliser



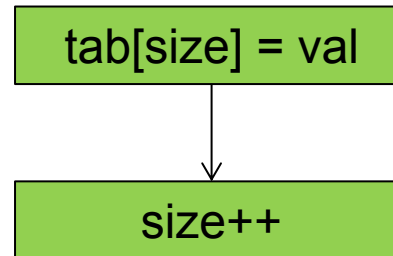
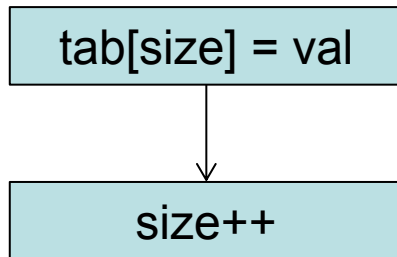
La concurrence =
Parallélisme Potentiel

Cf. aussi : Loi de Gustafson (débit théorique)

Mieux adaptée pour les architectures e.g vectorielles

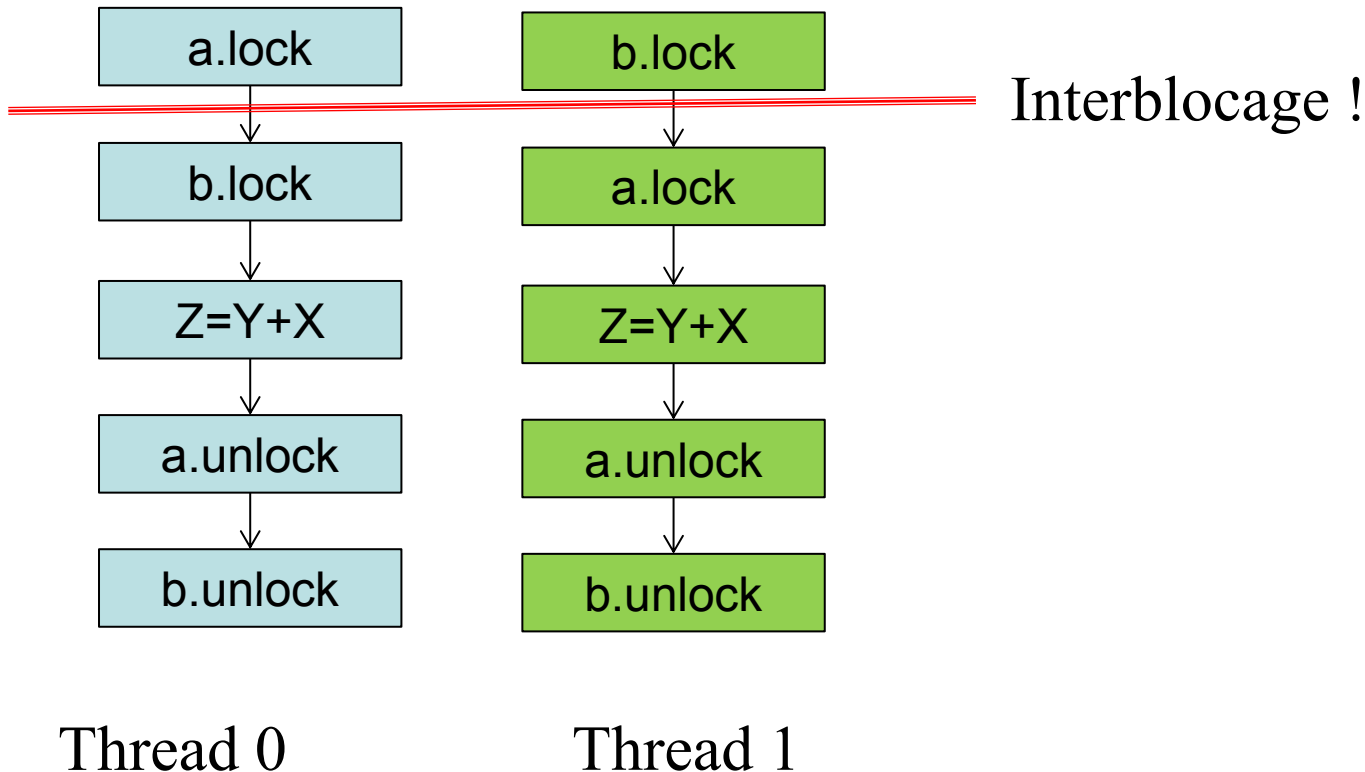
Problèmes liés à la concurrence

- 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 (!= 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"

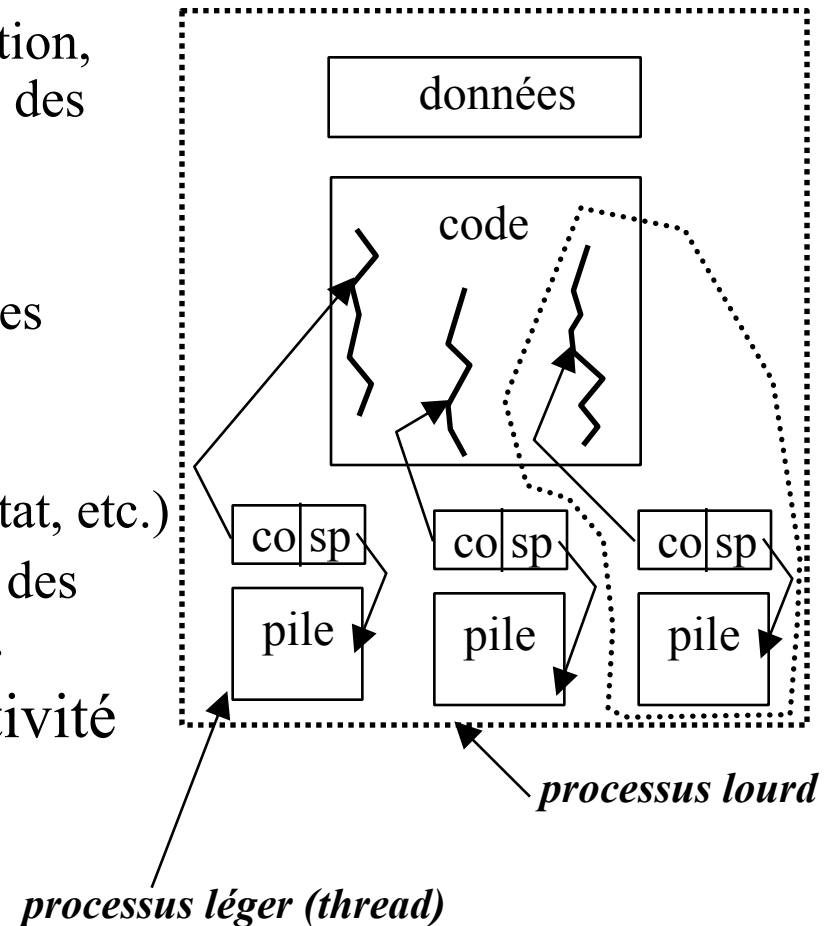
Partage les zones de code, de données, de tas + des zones du PCB (Process Control Block) :

- ✓ 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
 - 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)

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

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();
    }
}
```

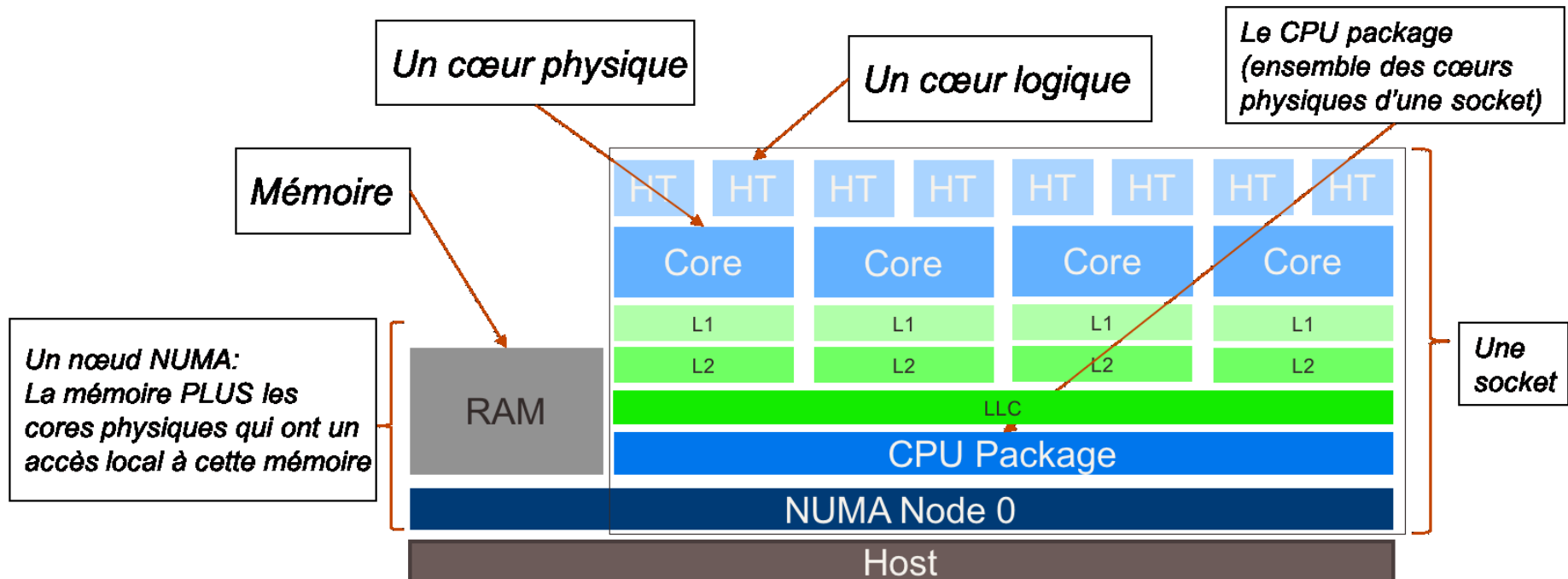
Atomic

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)
- Cependant, 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++
 - On se limite à présenter les types atomiques primitifs ici
 - Barrières mémoire abordées en cours 8

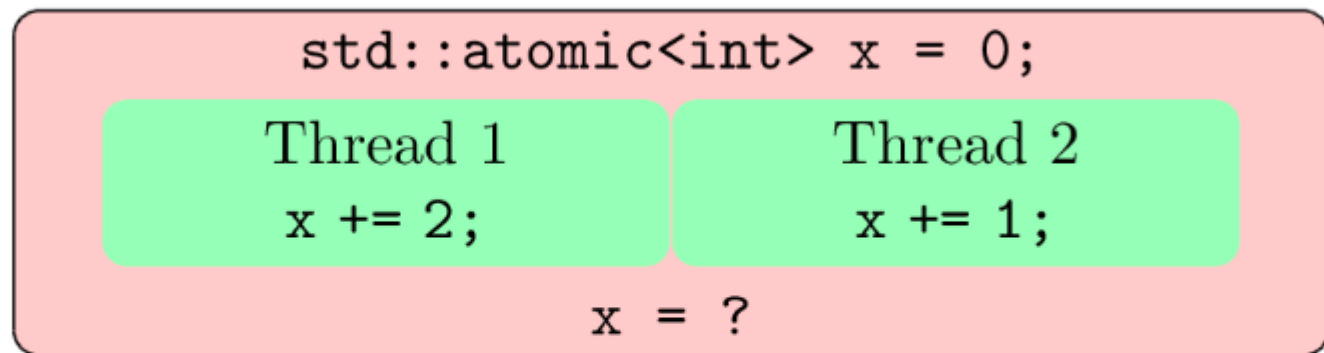
Lien sur le matériel

- Structure Matérielle NUMA
 - Cohérence de cache niveau matériel
 - Mécanisme relativement complexe
 - Opérations CAS et importance pratique
 - Structures de données « lock free »



Atomic en pratique

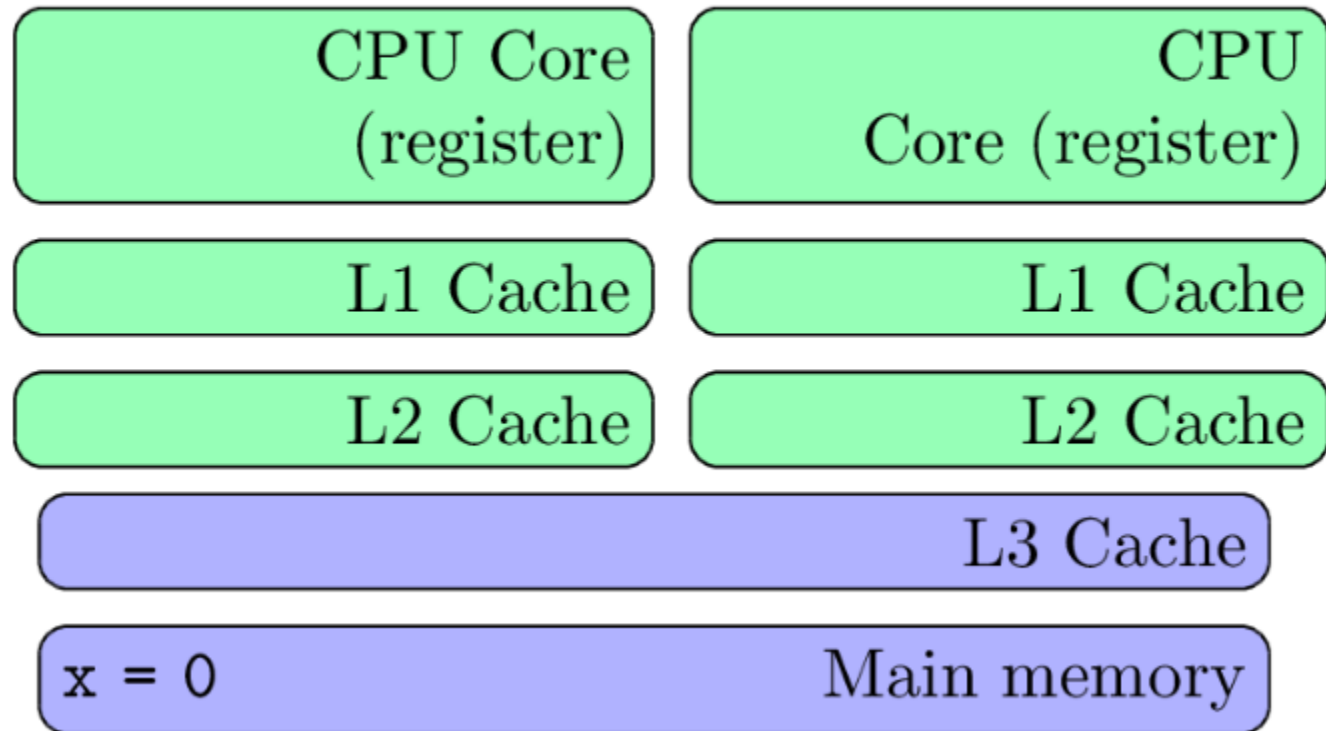
NUMA revisited

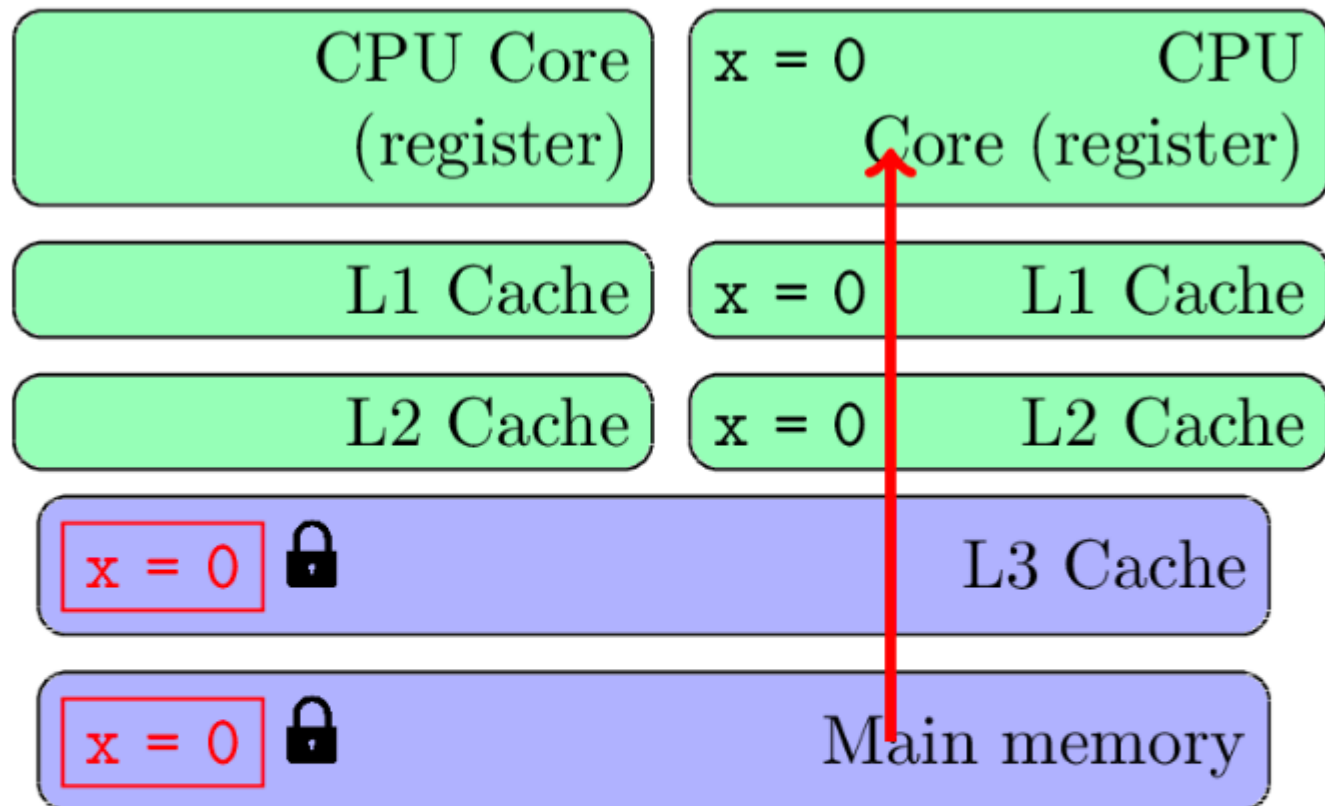


- Read modify write operation:
 - Read `x` from memory
 - Add something to `x`
 - Write `x` to memory

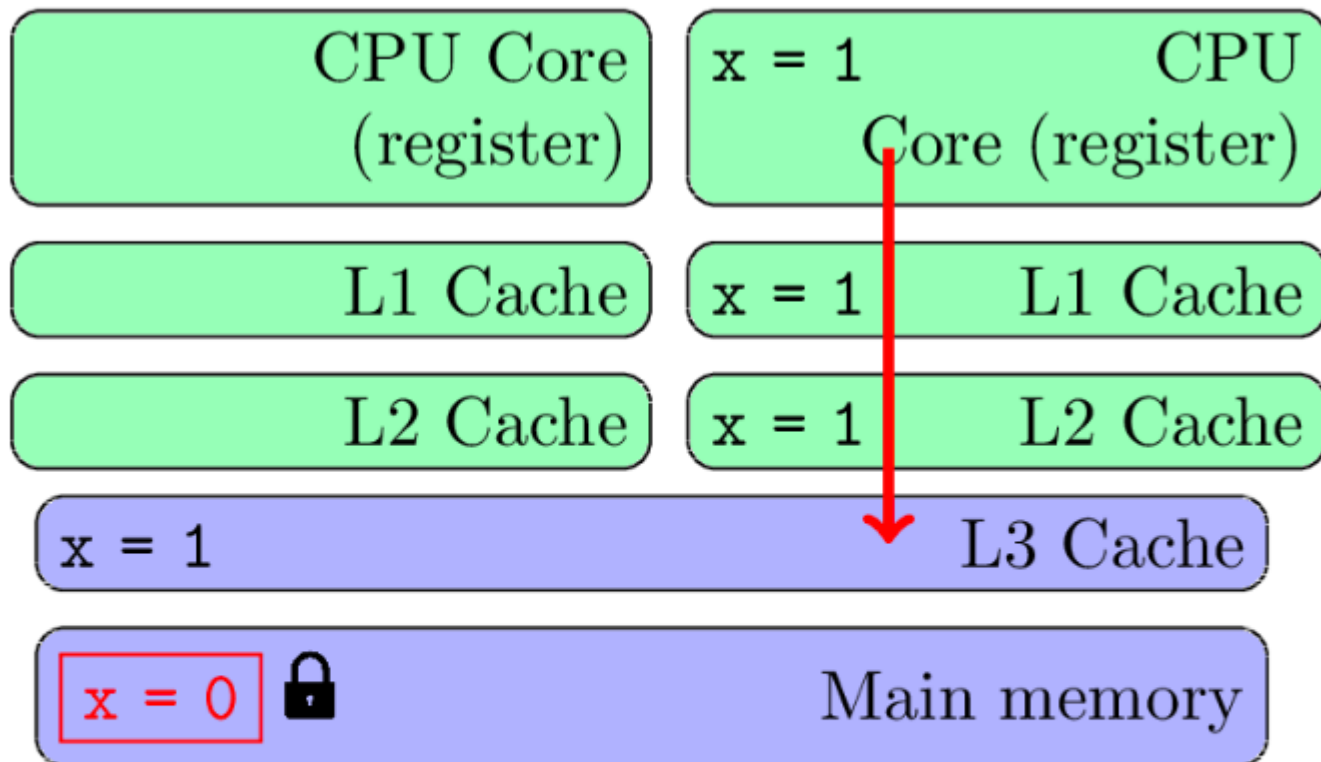
Nécessité d'une gestion matérielle

What's going on ?

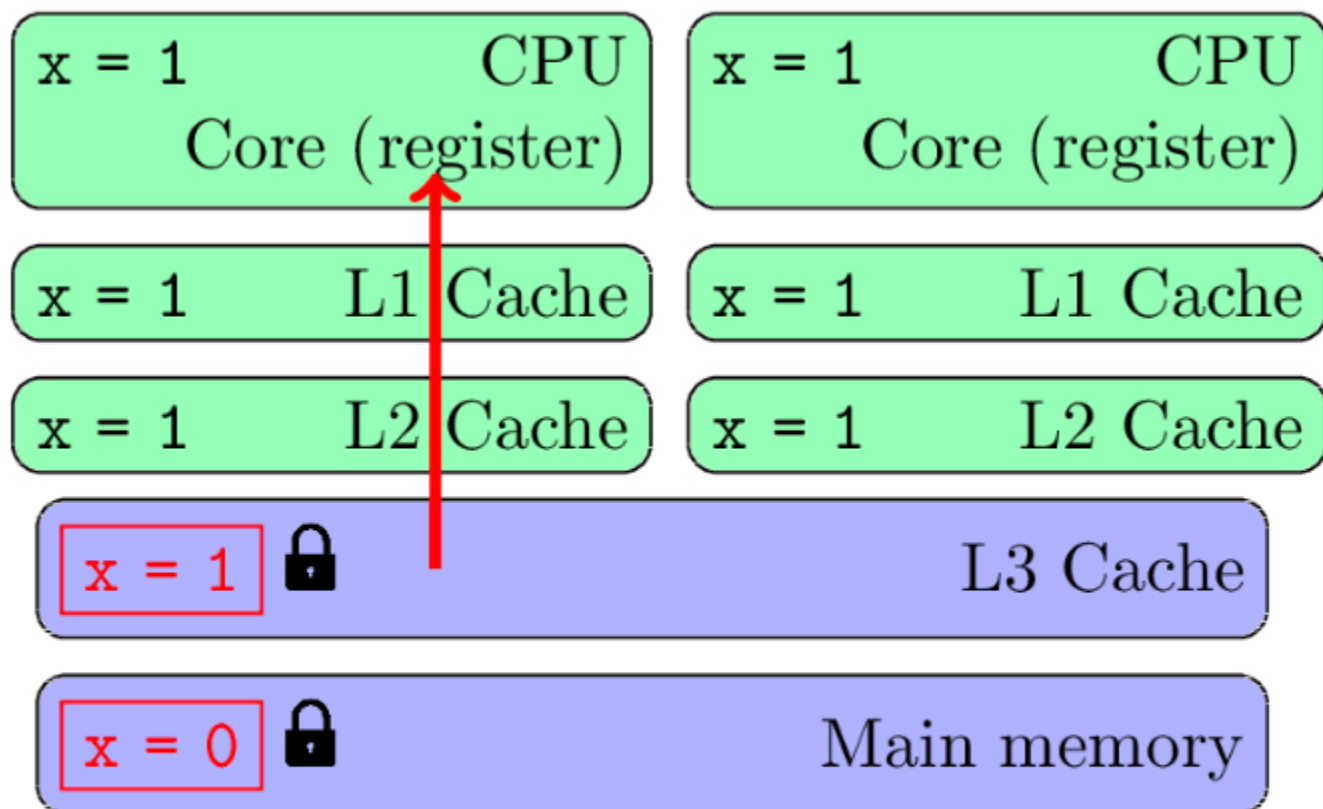




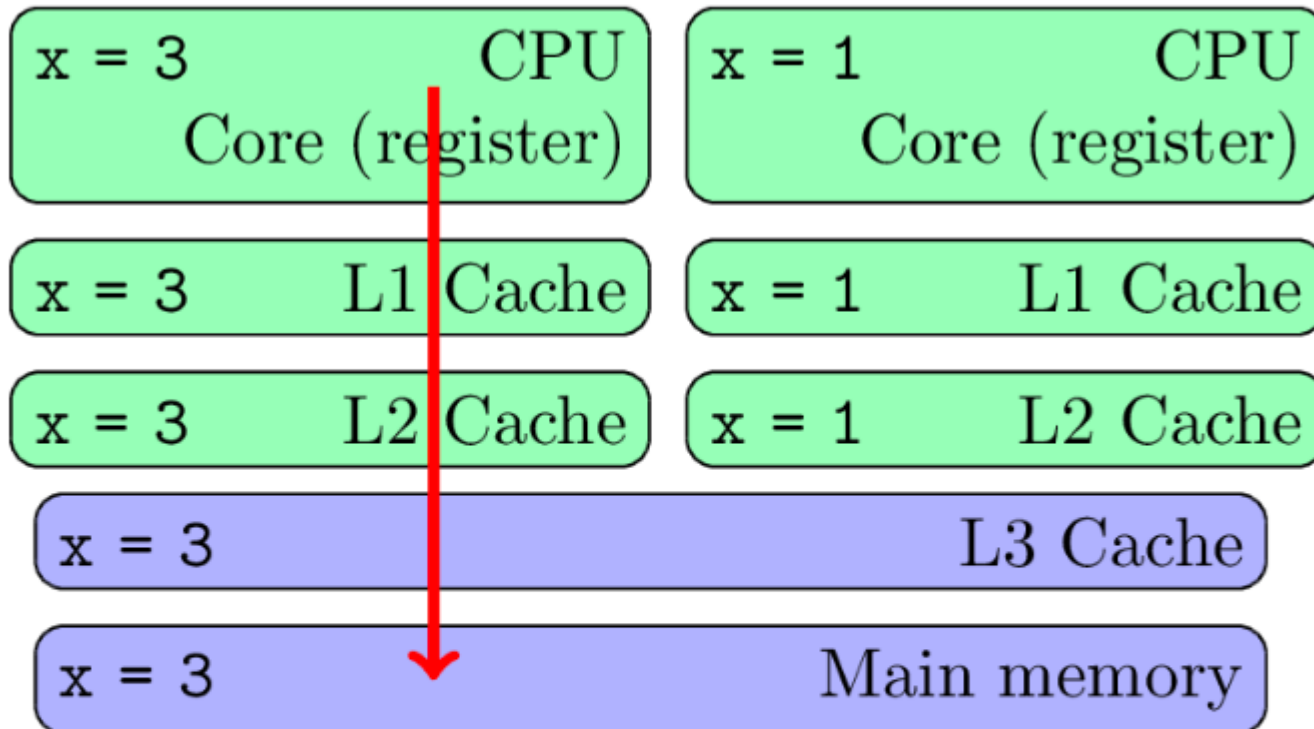
- Thread 2 executes
 - It fetches x from caches to main memory
 - It locks x address (by hardware impl.)
- Thread 1 executes
 - It try to fetch a x which is locked (goes to by hardware impl.)



- 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

Propriétés et opérateurs d'un atomic

- Offert pour les types primitifs numériques
 - Garantit l'atomicité des opérations : incrément ++ ou +=, décrement – ou -=, et opérations bit à bit &=, |=, ^=
 - Cas particulier booléen : atomic_flag
- Utilisation « naturelle »
 - Déclaration
 - `atomic<int> i =0;`
 - Opérateurs Disponibles
 - `i++`
 - `i+=5;`
 - Opérations non atomiques
 - Multiplication, manipulation classique (+)...

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;
}
```

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

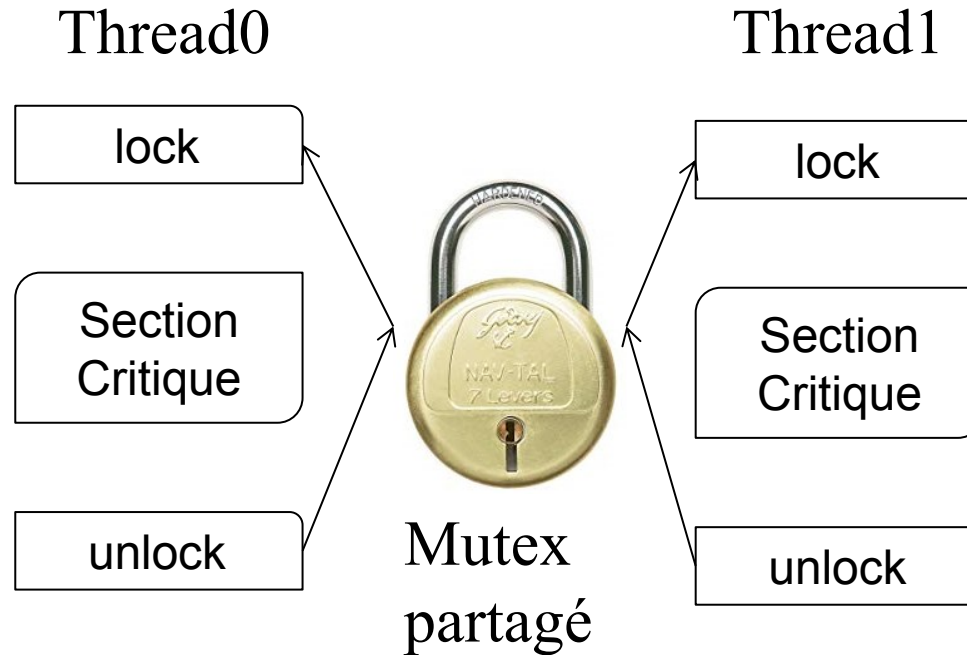
```
class SpinLock {  
    std::atomic_flag locked = ATOMIC_FLAG_INIT ;  
public:  
    void lock() {  
        while (locked.test_and_set(std::memory_order_acquire)) { ; }  
    }  
    void unlock() {  
        locked.clear(std::memory_order_release);  
    }  
};
```

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
- 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>

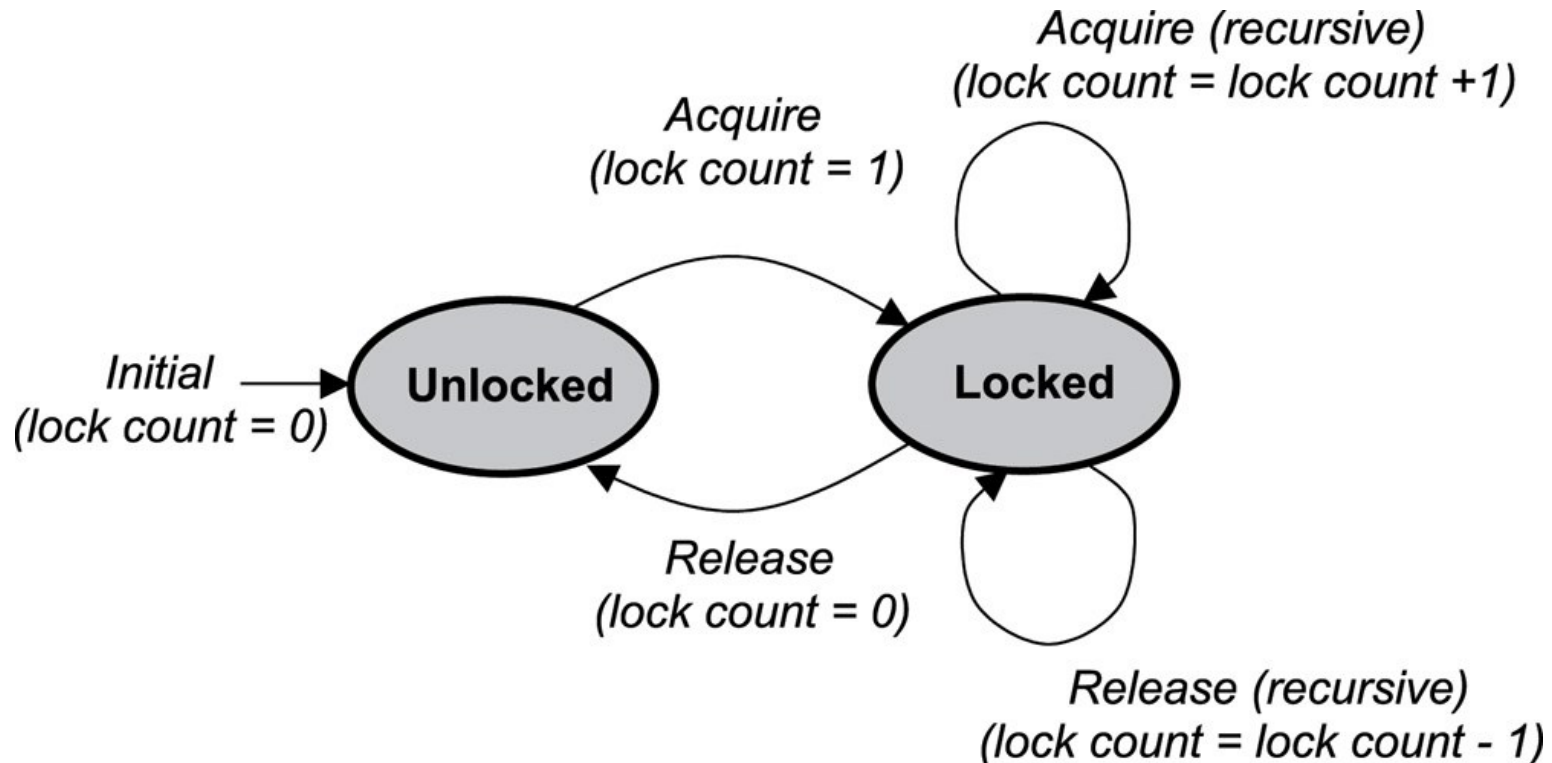
```
std::mutex mtx;           // mutex for critical section
```

```
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();
}
```

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)
}
```

MultiThread-Safe

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

MultiThread-Safe

- Les méthodes d'une classe manipulent les mêmes données
 - Comment garantir un accès cohérent aux instances ?
 - Atomic ne fournit pas toutes les garanties utiles
 - Une section critique contient plusieurs actions, le grain est arbitraire
- Solution générique :
 - Un Mutex par instance, barrière commune à l'entrée, libérée en sortie pour les opérations membres (cf. moniteurs Java)

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 « `synchronized` » de Java

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
- On peut utiliser plusieurs locks (grain fin, e.g. par bucket de la table de hash), voire simplement des atomics dans certains cas
- Des stratégies distinguant les écritures et les lectures sont utiles
 - Lecteurs concurrents = OK
 - Supporté par `shared_mutex` dans C++14/17

Lock multiples

- Problème grave possible : interblocage
 - Existence de la possibilité d'acquérir une autre ressource quand on en détient une
 - Existence de cycles possibles

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();  
}
```

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 **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
- Acquiert le lock si disponible (comme lock) et rend -1, sinon
 - Rend l'indice du lock ayant échoué
- 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 RAI
 - Acquisition du lock à la déclaration
 - Sauf si mode « diferré »
 - `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

```
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& i: threads)
```

```
        i.join();
```

```
    std::cout << a << "th and " << a+1 <<
```

```
    "th Fibonacci numbers: "
```

```
        << b << " and " << c << '\n';
```

```
}
```

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
- **notify_one(condition), notify_all(condtion)**
 - 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::lock_guard<std::mutex> lk(m);
        ready = true;
        std::cout << "main() signals data ready for
processing\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 <<
'\n';
    worker.join();
}
```

Attendre une Condition

- Toujours mettre le test sous protection du mutex
 - `if (! ready) { unique_lock<mutex> l(m) ; cv.wait(l); }`
 - Problème : `ready` est testé en dehors du mutex (pas atomique)
- En général il faut tester la condition dans une boucle
 - `unique_lock<mutex> l(m) ; if (! ready) { cv.wait(l); }`
 - `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`
- Version lambda de wait (*wait for test to be true*):
 - `cond.wait(lock, test) <=> while (!test) cond.wait(lock)`
 - Test est une lambda : `[&]()->bool { return ready; }`

<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 for
processing\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 <<
'\n';
    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 (!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 (!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 wait⁶⁴

```
// 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();
});
```

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 (!done) {
        while (queue.empty() && !done) {
            // loop to avoid spurious wakeups
            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

- 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
- Exemples :
 - Une barrière de rendez-vous :
 - les N premiers attendent, le N+1 éme débloquent tout le monde
 - Une pile ou une queue partagée
 - Bloquant sur pop si vide, Bloquant sur push si plein
 - On réveille les threads bloqués quand on pop sur plein/push sur vide

Nombreux autres exemples possibles en exercice...

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