

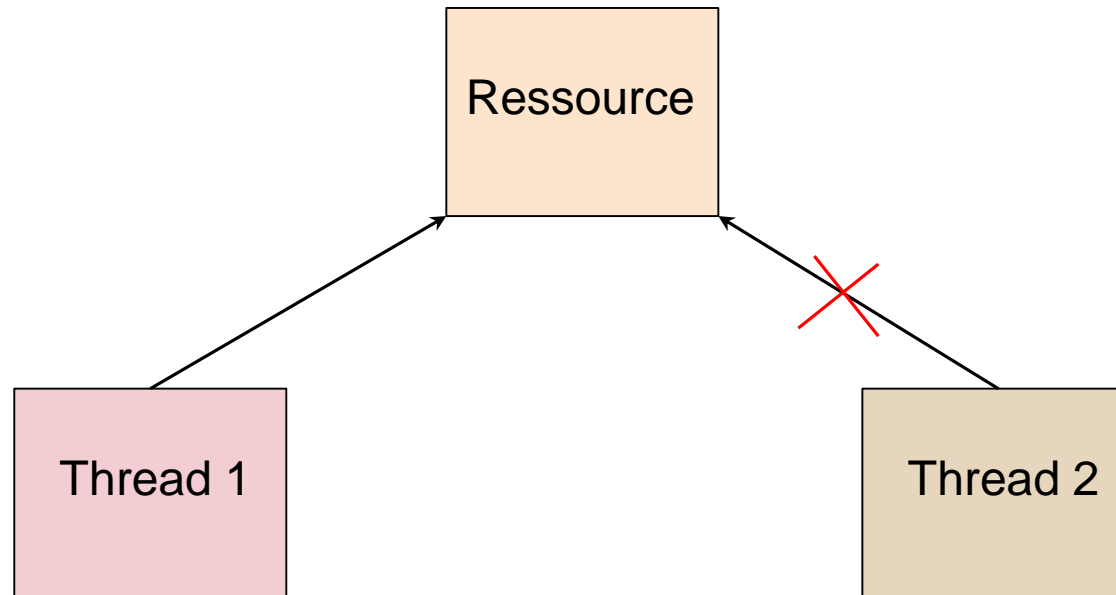
Mécanisme d'exclusion mutuelle

F. Gluck, V. Pilloux

Version 0.9

Exclusion mutuelle

- L'exclusion mutuelle est un mécanisme utile lorsque plusieurs threads doivent pouvoir accéder à une ressource unique
- La ressource unique peut être une simple variable globale, une fonction ou un pilote entier
- Il s'agit alors de s'assurer **qu'un seul thread** puisse avoir accès à cette ressource en même temps



Modèle d'exclusion mutuelle

- L'accès à une section critique se fait par un algorithme d'exclusion mutuelle en deux parties :
 - **Protocole d'entrée**
 - **Protocole de sortie**

Modèle typique d'exclusion mutuelle pour un thread :

```
void *thread(void *arg) {  
    ...  
    // Instructions  
    ...  
    PROTOCOLE D'ENTREE (prélude)  
    SECTION CRITIQUE // Accès à la ressource critique  
    PROTOCOLE DE SORTIE (postlude)  
    ...  
    // Instructions  
    ...  
}
```

Exemple d'exclusion mutuelle

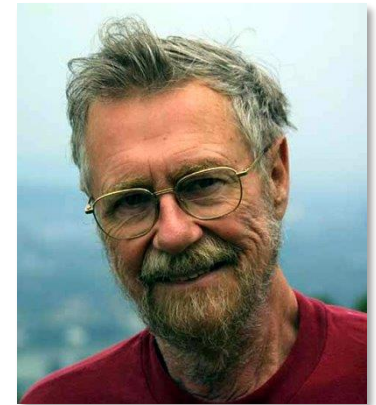
Exemple :

```
int n = 0;

void *func1(void *arg) {
    for (int i = 0; i < COUNT; i++) {
        Entrée en section critique
        n++;          // SECTION CRITIQUE
        Sortie de la section critique
    }
    printf("Thread 1 terminated.\n");
    return NULL;
}

void *func2(void *arg) {
    for (int i = 0; i < COUNT; i++) {
        Entrée en section critique
        n++;          // SECTION CRITIQUE
        Sortie de la section critique
    }
    printf("Thread 2 terminated.\n");
    return NULL;
}
```

Propriétés d'un algorithme d'excl. mutuelle



Un algorithme d'exclusion mutuelle doit au moins satisfaire les propriétés suivantes, énoncées initialement par Edsger W. Dijkstra* :

1. **Exclusion mutuelle** : un seul thread à la fois est autorisé à exécuter le code de la section critique.
2. **Absence d'interblocage (progression)** : un thread doit progresser et ne doit donc jamais être bloqué indéfiniment.
3. **Absence de famine (attente bornée)** : tout thread désirant entrer en section critique doit pouvoir y parvenir à un moment donné.

* Dijkstra, E. W. "Solution of a problem in concurrent programming control". Communications of the ACM. 8 (9): 569, 1965.

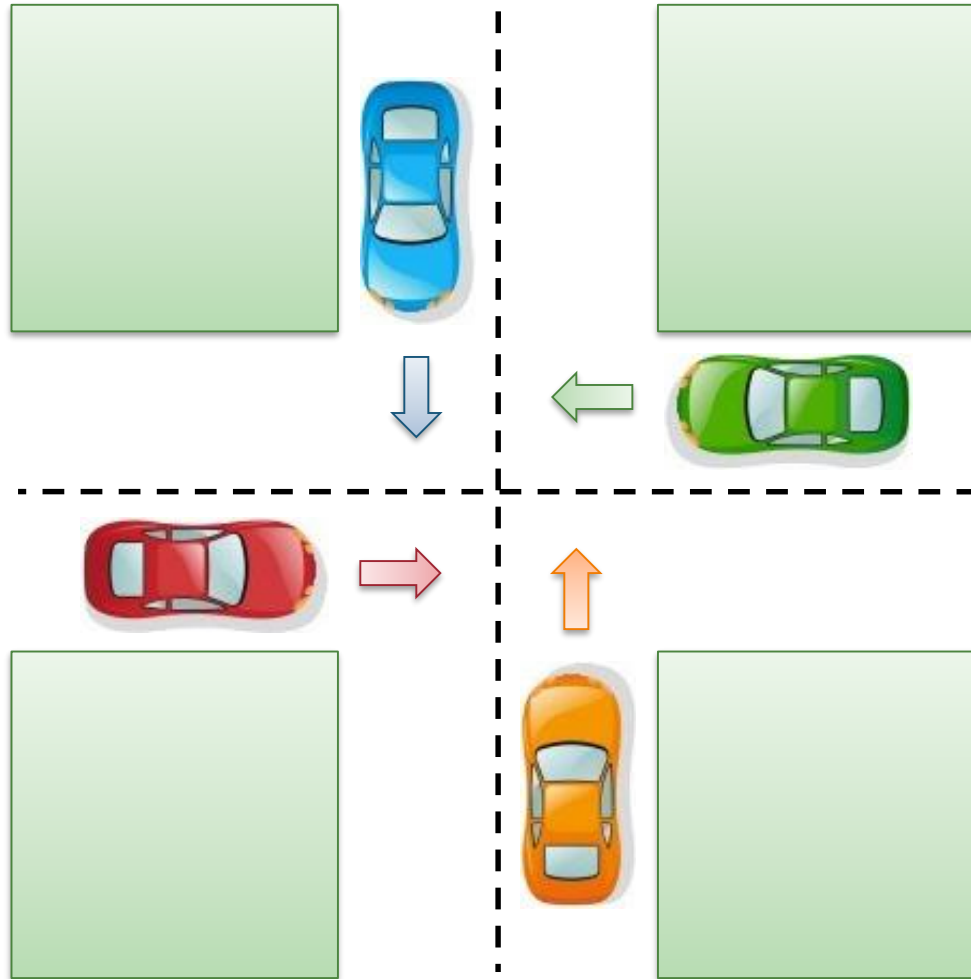
Interblocage (1)

- Un **interblocage** (*deadlock*) est une situation où un ou plusieurs threads sont bloqués de manière permanente ou si il n'y a plus de progression.
- Un thread est bloqué s'il ne s'exécute plus et attend qu'un événement se réalise.
- Exemple :
 - T0 attend sur la réception d'un message en provenance de T1. Si T1 se termine ou n'envoie pas de message → interblocage !
- Un **interblocage global** apparaît lorsque tous les threads non terminés sont interbloqués.



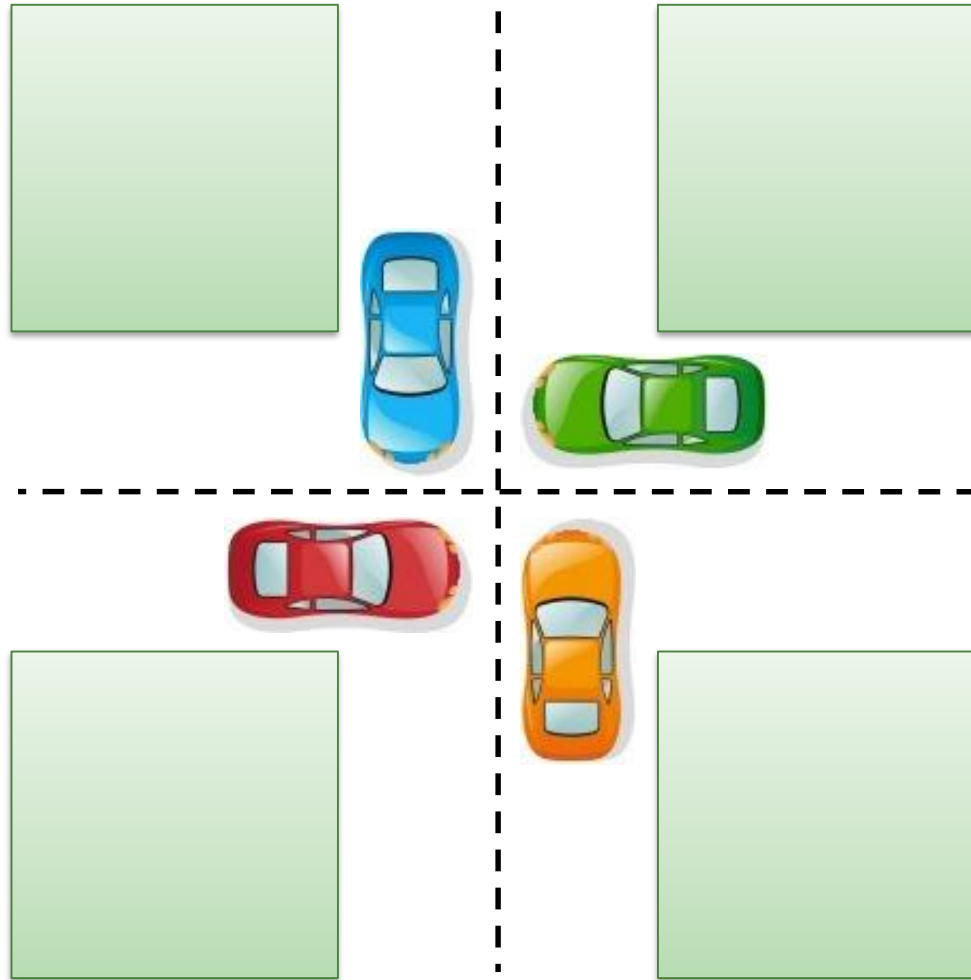
Interblocage (2)

- Risque potentiel d'interblocage :



Interblocage (3)

- Interblocage:



Interblocage (4)

- Deux threads concurrents (T0 et T1) bloquent mutuellement :
 - T0 s'est emparé de la ressource critique R0 et veut s'emparer de la ressource critique R1.
 - T1 s'est emparé de R1 et veut s'emparer de R0.

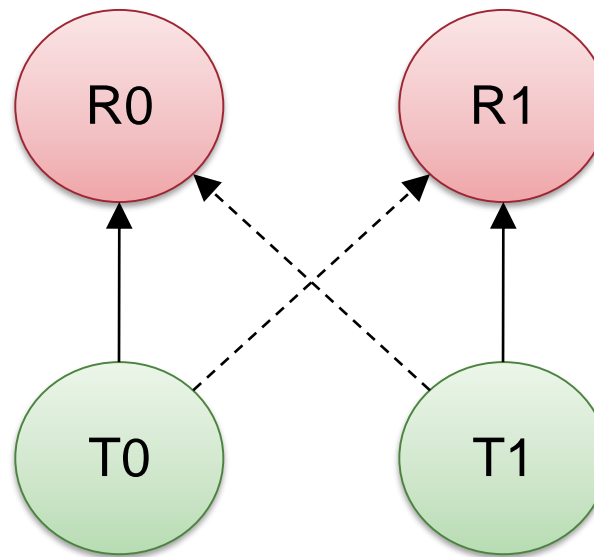


Figure 1: Interblocage des threads T0 et T1

Famine (1)

- Une **famine** (*starvation*) est une situation où un ou plusieurs threads n'ont pas la garantie d'accéder à leur section critique en un temps fini.
- Une famine peut, entre-autre, être causée par :
 - algorithme d'ordonnancement de tâches perfectible ;
 - algorithme d'exclusion mutuelle perfectible ;
 - contention envers une ou plusieurs ressources critiques.
- Si la contention envers une section critique est faible, alors une famine est peu probable.



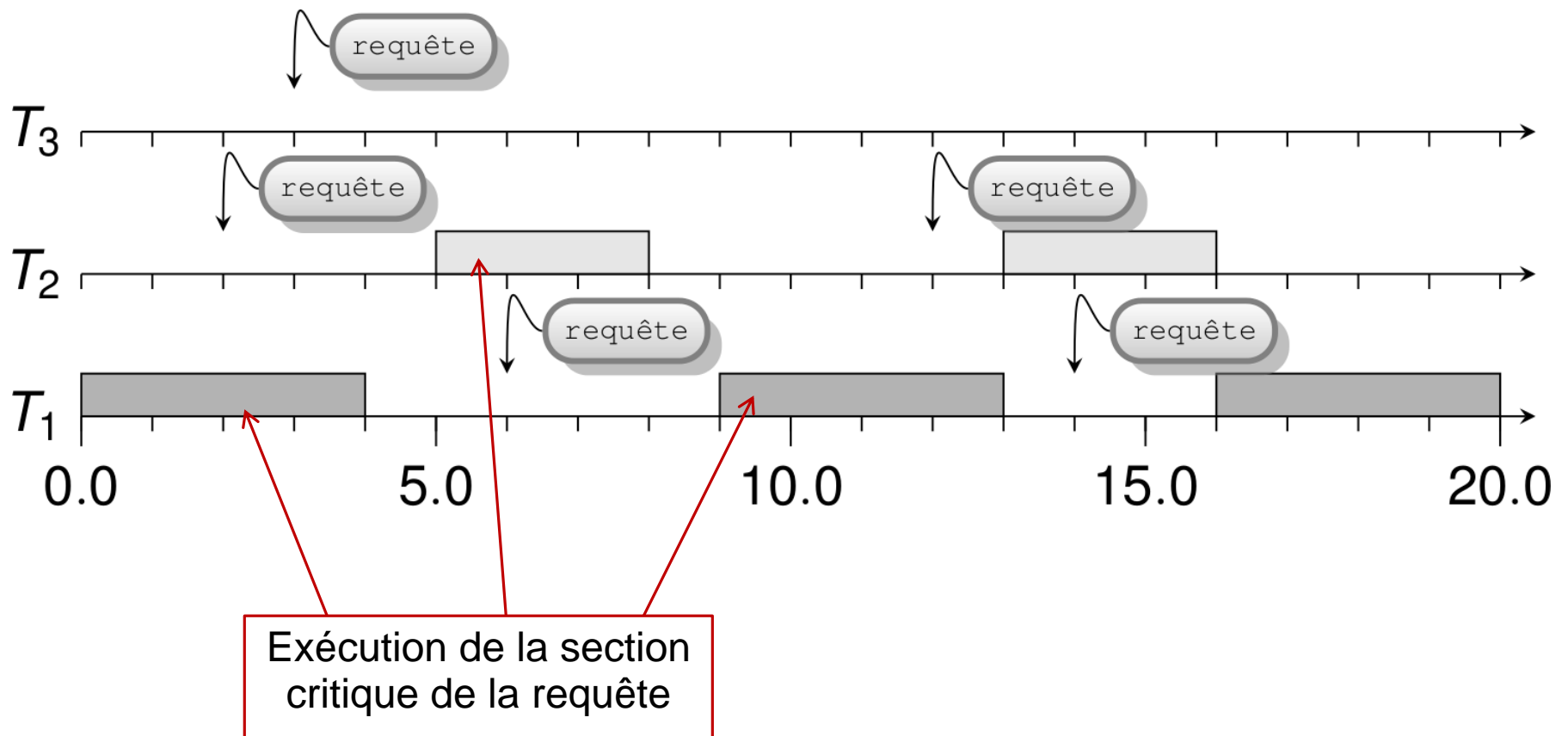
Famine (2)

- Une famine est similaire à un interblocage : un ou plusieurs threads sont bloqués.
- Le blocage dû à une famine n'est toutefois **pas permanent**.
- Une famine peut se produire lorsqu'un thread attend l'accès à une ressource alors que celle-ci continue à être utilisée par un autre thread.
- L'absence de famine est une garantie **plus forte** que l'absence d'interblocage : un algorithme d'exclusion mutuelle choisissant de laisser entrer un thread en section critique arbitrairement est absent d'interblocage, mais pas de famine*.

* Concurrent Programming: Algorithms, Principles, and Foundations. Springer.

Famine (3)

Exemple d'algorithme d'exclusion mutuelle souffrant d'une famine :



Propriétés des algorithmes

- Les algorithmes d'exclusion mutuelle présentés ici utilisent des instructions usuelles :
 - **attente active** par des boucles
 - variables partagées (globales)
 - ne gèrent que deux threads concurrents (plus simple)
 - Ces algorithmes tentent d'assurer :
 - L'exclusion mutuelle
 - L'absence d'interblocage
 - L'absence de famine
- ... et d'éviter toute attente inutile !

Le problème de la section critique

Le problème de la section critique implique plusieurs threads exécutant le code suivant :

```
while (true) {  
    ...  
    section non-critique    // un thread peut se terminer ici  
    ...  
    protocole d'entrée  
    section critique        // accès à des ressources partagées  
    protocole de sortie  
    ...  
    section non-critique    // un thread peut se terminer ici  
    ...  
}
```

Algorithme 1

```
bool busy = false;

void *T0(void *arg){
    while (true) {
        while (busy){}
        busy = true;
        // section critique
        busy = false;
        // section non-critique
    }
}

void *T1(void *arg){
    while (true) {
        while (busy){}
        busy = true;
        // section critique
        busy = false;
        // section non-critique
    }
}
```

Satisfait-il les critères d'exclusion mutuelle ?

- Garantit l'exclusion mutuelle ?

Algorithme 1

```
bool busy = false;

void *T0(void *arg){
    while (true) {
        while (busy){}
        busy = true;
        // section critique
        busy = false;
        // section non-critique
    }
}

void *T1(void *arg){
    while (true) {
        while (busy){}
        busy = true;
        // section critique
        busy = false;
        // section non-critique
    }
}
```

Satisfait les critères d'exclusion mutuelle ?

- Garantit l'exclusion mutuelle ?
- T0 lit busy à false
-
- T1 lit busy à false
- T1 met busy à true
- T1 entre en section critique
-
- T0 met busy à true
- T0 entre aussi en section critique !



**Exclusion mutuelle
non satisfaite !**

Algorithme 2

```
int turn = 0; // ou 1

void *T0(void *arg){
    while (true) {
        while (turn == 1) {}
        // section critique
        turn = 1;
        // section non-critique
    }
}

void *T1(void *arg){
    while (true) {
        while (turn == 0) {}
        // section critique
        turn = 0;
        // section non-critique
    }
}
```

Satisfait les critères d'exclusion mutuelle ?

- Garantit l'exclusion mutuelle ?

Algorithme 2

```
int turn = 0; // ou 1

void *T0(void *arg){
    while (true) {
        while (turn == 1) {}
        // section critique
        turn = 1;
        // section non-critique
    }
}

void *T1(void *arg){
    while (true) {
        while (turn == 0) {}
        // section critique
        turn = 0;
        // section non-critique
    }
}
```

Satisfait les critères d'exclusion mutuelle ?

✓ Garantit l'exclusion mutuelle

Algorithme 2

```
int turn = 0; // ou 1

void *T0(void *arg){
    while (true) {
        while (turn == 1) {}
        // section critique
        turn = 1;
        // section non-critique
    }
}

void *T1(void *arg){
    while (true) {
        while (turn == 0) {}
        // section critique
        turn = 0;
        // section non-critique
    }
}
```

Satisfait les critères d'exclusion mutuelle ?

✓ Garantit l'exclusion mutuelle

- Interblocage ?

Algorithme 2

```
int turn = 0; // ou 1

void *T0(void *arg){
    while (true) {
        while (turn == 1) {}
        // section critique
        turn = 1;
        // section non-critique
    }
}

void *T1(void *arg){
    while (true) {
        while (turn == 0) {}
        // section critique
        turn = 0;
        // section non-critique
    }
}
```

Satisfait les critères d'exclusion mutuelle ?

✓ Garantit l'exclusion mutuelle

- Interblocage ?
- T0 passe le `while`
- T0 entre et sort de sa section critique
- T0 met `turn` à 1
- T0 se termine ou meurt dans sa section non-critique

-
- T1 fait un tour de boucle complet
 - T1 reste bloqué dans le `while` !



Interblocage !

Algorithme 3

```
bool enter[2] = {false, false};
```

```
void *T0(void *arg){  
    while (true) {  
        enter[0] = true;  
        while (enter[1]){}  
        // section critique  
        enter[0] = false;  
        // section non-critique  
    }  
}
```

```
void *T1(void *arg){  
    while (true) {  
        enter[1] = true;  
        while (enter[0]){}  
        // section critique  
        enter[1] = false;  
        // section non-critique  
    }  
}
```

Satisfait les critères d'exclusion mutuelle ?

- Garantit l'exclusion mutuelle ?

Algorithme 3

```
bool enter[2] = {false, false};
```

```
void *T0(void *arg){  
    while (true) {  
        enter[0] = true;  
        while (enter[1]){}  
        // section critique  
        enter[0] = false;  
        // section non-critique  
    }  
}
```

```
void *T1(void *arg){  
    while (true) {  
        enter[1] = true;  
        while (enter[0]){}  
        // section critique  
        enter[1] = false;  
        // section non-critique  
    }  
}
```

Satisfait les critères d'exclusion mutuelle ?

✓ Garantit l'exclusion mutuelle ?

Algorithme 3

```
bool enter[2] = {false, false};

void *T0(void *arg){
    while (true) {
        enter[0] = true;
        while (enter[1]){}
        // section critique
        enter[0] = false;
        // section non-critique
    }
}

void *T1(void *arg){
    while (true) {
        enter[1] = true;
        while (enter[0]){}
        // section critique
        enter[1] = false;
        // section non-critique
    }
}
```

Satisfait les critères d'exclusion mutuelle ?

✓ Garantit l'exclusion mutuelle ?

- Interblocage ?

Algorithme 3

```
bool enter[2] = {false, false};

void *T0(void *arg){
    while (true) {
        enter[0] = true;
        while (enter[1]){}
        // section critique
        enter[0] = false;
        // section non-critique
    }
}

void *T1(void *arg){
    while (true) {
        enter[1] = true;
        while (enter[0]){}
        // section critique
        enter[1] = false;
        // section non-critique
    }
}
```

Satisfait les critères d'exclusion mutuelle ?

✓ Garantit l'exclusion mutuelle ?

- Interblocage ?

- T0 met enter[0] à true

- T1 met enter[1] à true

- T1 bloque car enter[0] est vrai

- T0 bloque car enter[1] est vrai



Interblocage !

Algorithme de Peterson

- Formulé en 1981 par le mathématicien Gary L. Peterson
- Algorithme satisfaisant les trois propriétés :
 - exclusion mutuelle ;
 - progression ;
 - attente bornée.

Algorithme de Peterson

```
bool enter[2] = { false, false };  
int turn = 0;
```

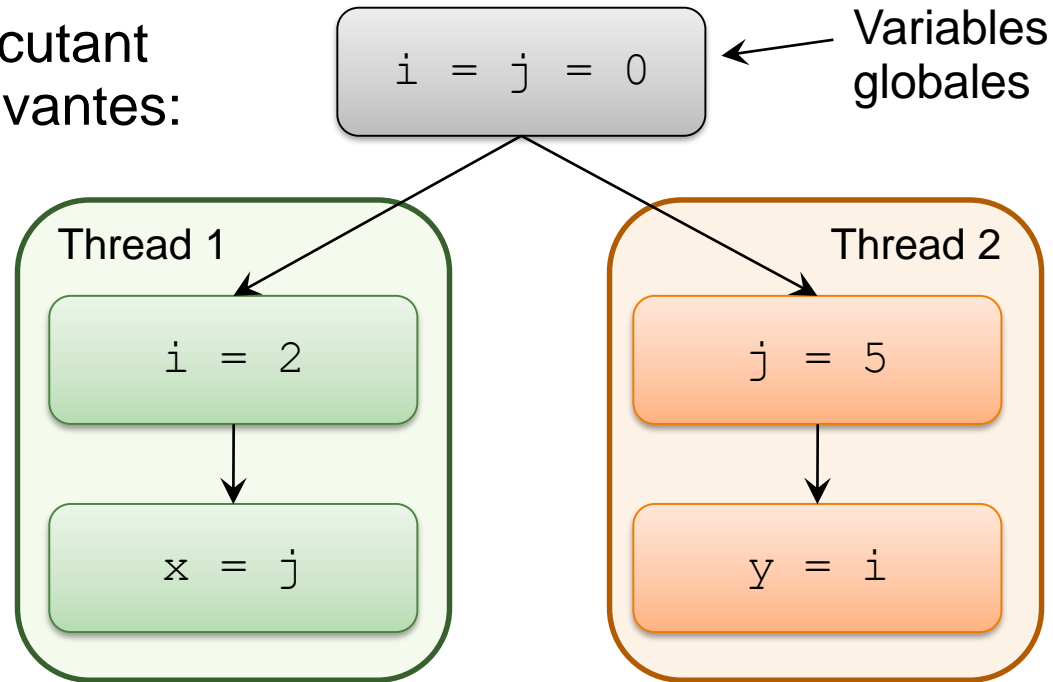
```
void *T0(void *arg){  
    while (true) {  
        enter[0] = true;  
        turn = 1;  
        while (enter[1]  
                && turn == 1) {  
        }  
        // section critique  
        enter[0] = false;  
        // section non-critique  
    }  
}
```

```
void *T1(void *arg){  
    while (true) {  
        enter[1] = true;  
        turn = 0;  
        while (enter[0]  
                && turn == 0) {  
        }  
        // section critique  
        enter[1] = false;  
        // section non-critique  
    }  
}
```

- Cet algorithme fonctionne-t-il à tous les coups?
 - Oui, dans le cas d'un monoprocesseur
 - Non, dans le cas général!

Problème de cohérence séquentielle

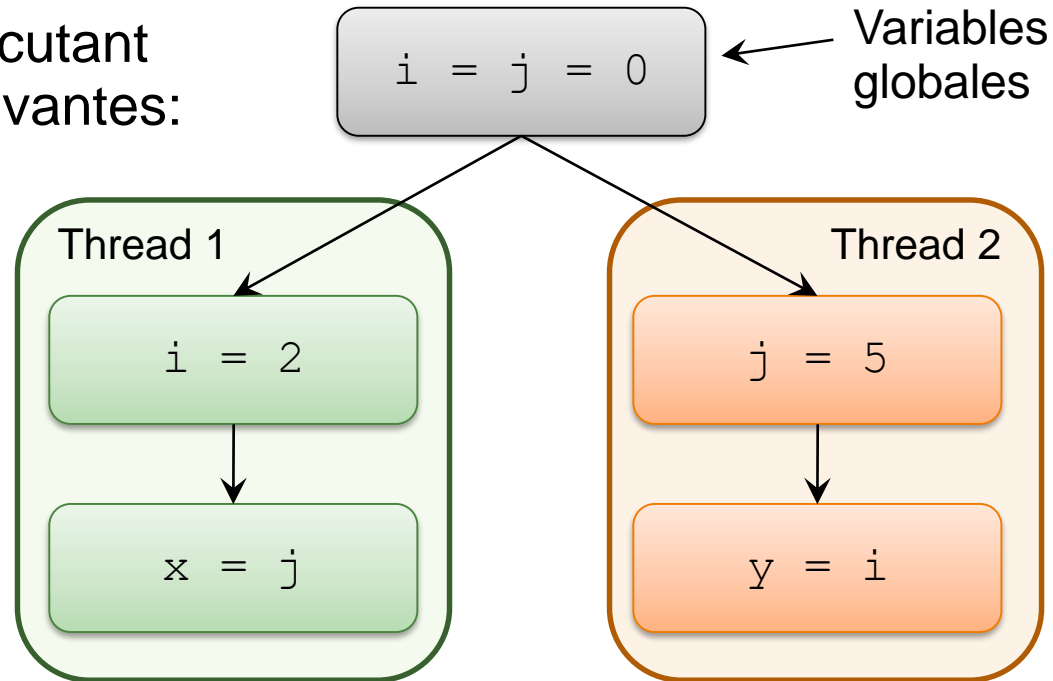
- Soit 2 threads exécutant les instructions suivantes:



- Quelles sont les valeurs potentielles de x et y une fois les 2 threads terminés ?

Cohérence séquentielle

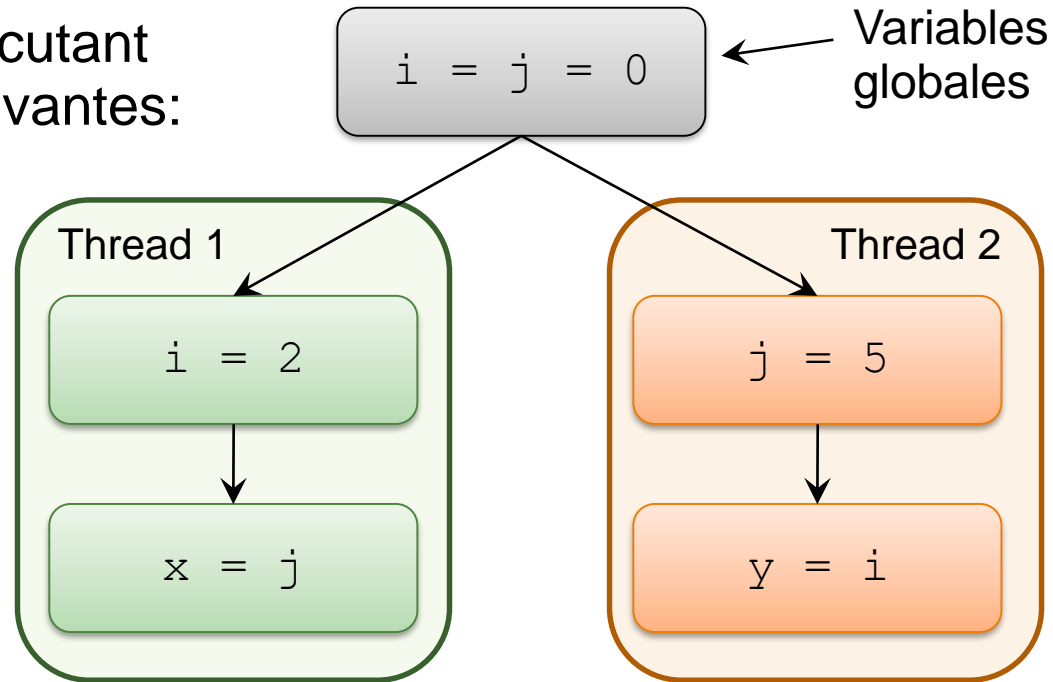
- Soit 2 threads exécutant les instructions suivantes:



- Quelles sont les valeurs potentielles de x et y une fois les 2 threads terminés ?
- $x = 0$ et $y = 0$ est une solution possible ?

Cohérence séquentielle

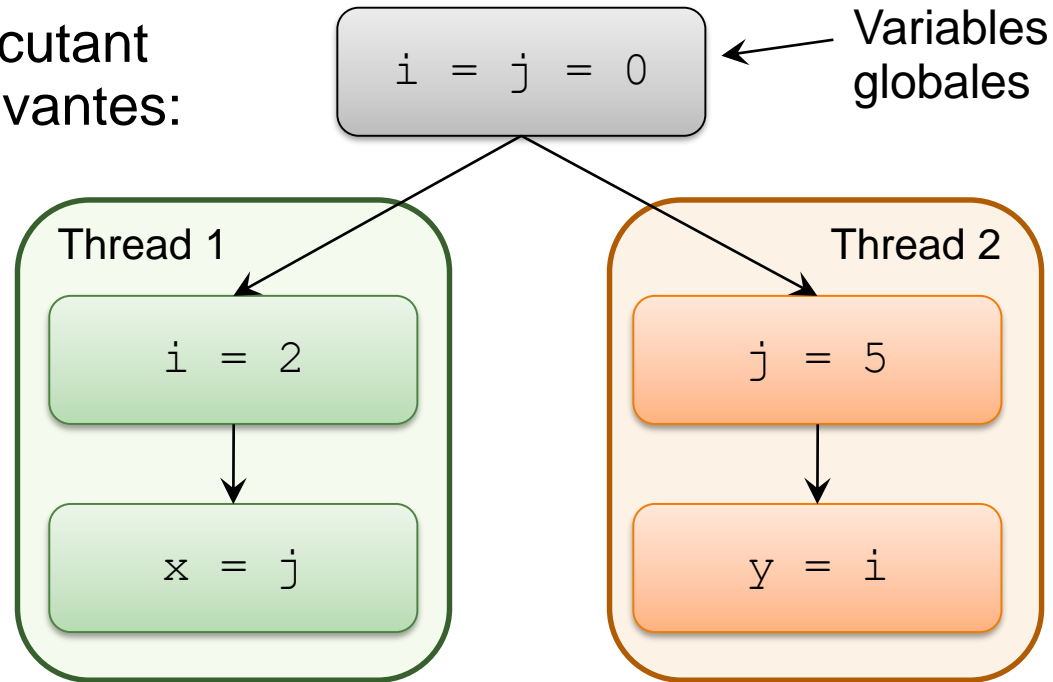
- Soit 2 threads exécutant les instructions suivantes:



- Quelles sont les valeurs potentielles de x et y une fois les 2 threads terminés ?
- $x = 0$ et $y = 0$ est une solution possible ? **OUI !**

Cohérence séquentielle

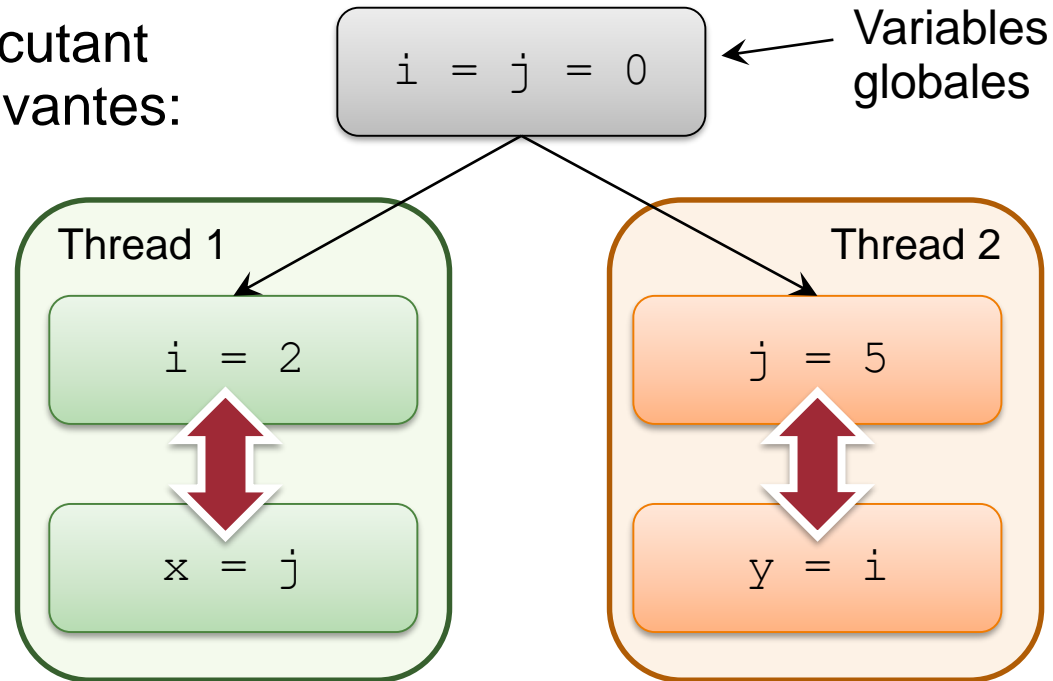
- Soit 2 threads exécutant les instructions suivantes:



- Quelles sont les valeurs potentielles de `x` et `y` une fois les 2 threads terminés ?
- `x = 0` et `y = 0` est une solution possible ? **OUI !**
- Optimisation du code \Rightarrow le compilateur (considérant qu'il s'agit de 2 morceaux de code séquentiels) peut réarranger le code afin d'optimiser l'accès aux variables car les instructions de chaque thread sont indépendantes !

Cohérence séquentielle

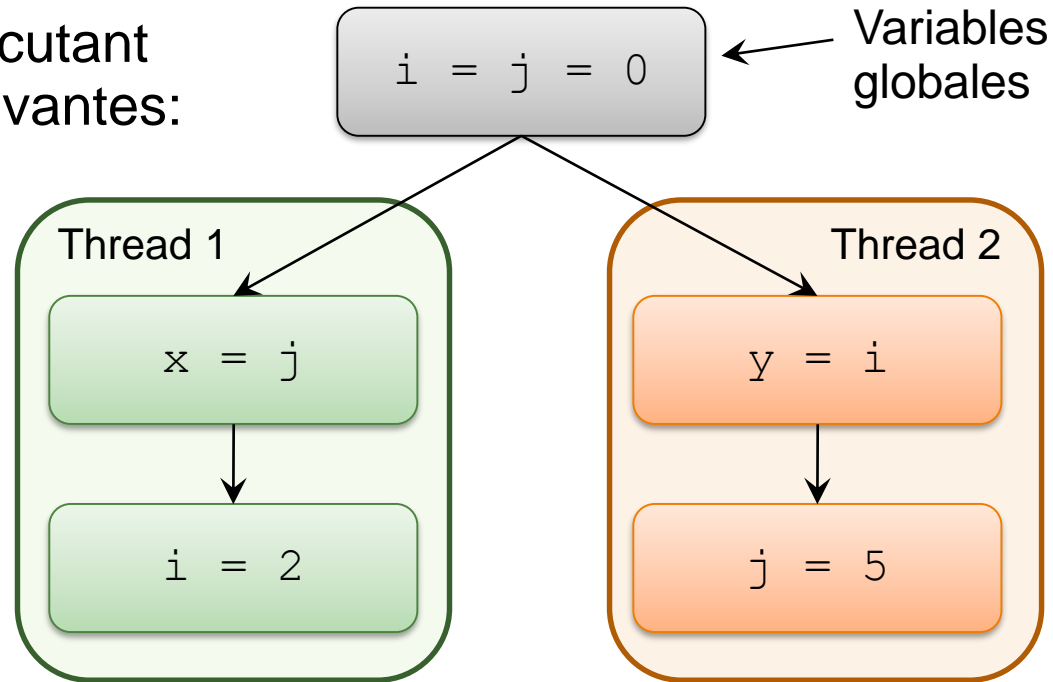
- Soit 2 threads exécutant les instructions suivantes:



- Quelles sont les valeurs potentielles de x et y une fois les 2 threads terminés ?
- $x = 0$ et $y = 0$ est une solution possible ? **OUI !**
- Optimisation du code \Rightarrow le compilateur peut réarranger le code afin d'optimiser l'accès aux variables car les instructions de chaque thread sont indépendantes !

Cohérence séquentielle

- Soit 2 threads exécutant les instructions suivantes:



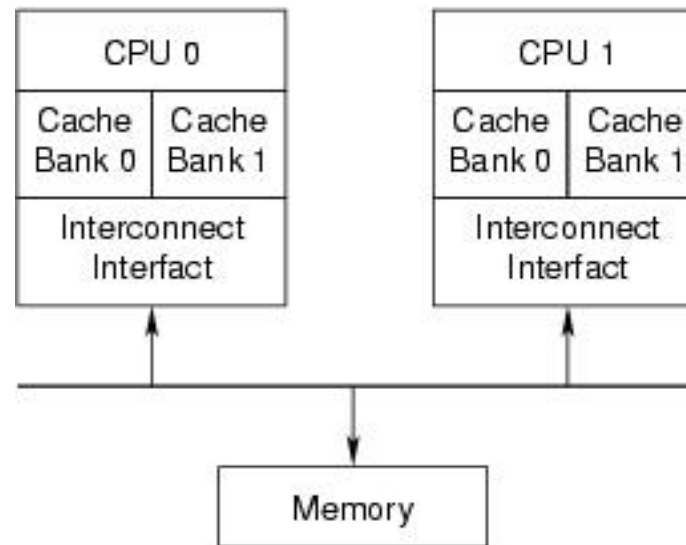
- Quelles sont les valeurs potentielles de `x` et `y` une fois les 2 threads terminés ?
- `x = 0` et `y = 0` est une solution possible ? **OUI !**
- Optimisation du code \Rightarrow le compilateur peut réarranger le code afin d'optimiser l'accès aux variables car les instructions de chaque thread sont indépendantes !

Concurrence et compilateur

- Afin d'améliorer la vitesse du code exécuté, le compilateur effectue diverses optimisations (variables dans des registres, réordonnancement instructions pour éviter accès mémoire successifs, suppression variables, déplacement de code, etc.).
 - Ces optimisations ne **changent pas** le comportement du code **dans un contexte séquentiel**.
 - Cependant, dans un contexte concurrent ou parallèle, ce n'est plus nécessairement le cas !
- ⇒ **le compilateur C analyse le code uniquement dans le cadre d'un contexte séquentiel !**
- Le mot-clé **volatile** permet de forcer le compilateur à n'effectuer aucune optimisation sur la variable, mais il n'est pas suffisant, comme nous allons le voir.

Problème de cohérence mémoire

- Accès à la mémoire « très lent » \Rightarrow mémoire cache permet de fortement diminuer la latence des accès.
- **Problème** sur architectures multi-processeurs : chaque CPU possède une cache locale \Rightarrow cohérence de la mémoire partagée ?



- La cohérence mémoire à un instant t n'est **pas garantie** et dépend de l'architecture matérielle!

Atomicité sur un processeur récent

- Alors comment assurer l'atomicité de la séquence:
 1. Lecture de la valeur d'une variable pour déterminer si la section critique est accessible
 2. Réservation de l'accès à la section critique en modifiant la valeur d'une variable
- Sur un système multiprocesseur avec plusieurs niveau de mémoire cache?
- La solution est **matérielle**: il faut fournir au programmeur une ou plusieurs instructions qui assurent d'être atomiques et qui permettent de constituer un verrou

Barrières mémoires

- Sur une architecture moderne, 2 CPU peuvent potentiellement utiliser la même variable globale, mais comme chaque CPU a sa propre mémoire cache, l'ajournement d'une variable globale par l'un des CPU ne met pas forcément à jour la mémoire principale tout de suite.
- La synchronisation entre deux tâches parallèles qui utilisent cette variable est alors aléatoire.
- **Une barrière mémoire** est une instruction processeur (`mfence` sur x86) forçant le CPU à invalider le contenu de sa mémoire cache et de ne faire aucune optimisation sur les accès mémoire.
- L'accès à la mémoire devient alors **sérialisé** et la consistance mémoire est ainsi **garantie**.
 - ⇒ **Désavantage**: le code accédant à la mémoire est alors beaucoup plus lent.
- Les primitives de synchronisation (mutex, sémaphore, etc.) utilisent des barrières mémoires lors de l'accès en section critique.

Instructions matérielles

- Les processeurs modernes implémentent diverses instructions matérielles **atomiques** pour palier aux problèmes vus précédemment et rendre les algorithmes d'exclusion mutuelle plus facilement implémentables et robustes :
 - instructions *Test And Set* (TAS) ;
 - instructions *Compare And Swap* (CAS) ;
 - instructions d'échange.

Algorithme avec instruction matérielle

- Exclusion mutuelle à l'aide d'une instruction *Test And Set*:

```
// Émulation de l'instruction atomique TAS
int test_and_set(int *val) {
    int prev = *val;
    *val = 1;
    return prev;
}

void *thread(void *param) {
    // Protocole d'entrée

    // Section critique

    // Protocole de sortie
}
```

Algorithme avec instruction matérielle

- Exclusion mutuelle à l'aide d'une instruction *Test And Set*:

```
// Émulation de l'instruction atomique TAS
int test_and_set(int *val) {
    int prev = *val;
    *val = 1;
    return prev;    // le 1er appel retourne 0
}
```

```
int lock = 0;
```

```
void *thread1(void *param) {
    // Protocole d'entrée
    while (test_and_set(&lock)) {}

    // Section critique

    // Protocole de sortie
    lock = 0;
}
```

```
void *thread2(void *param) {
    // Protocole d'entrée
    while (test_and_set(&lock)) {}

    // Section critique

    // Protocole de sortie
    lock = 0;
}
```

Exclusion mutuelle par masquage des interruptions

- La plupart des systèmes d'exploitation implémentent la commutation de tâche avec un timer déclenchant une interruption toutes les X millisecondes (typiquement 10 ms). Cette période est appelée 'tick' de l'OS.
- Cette période spécifie le quantum de temps disponible pour chaque thread. A chaque interruption le thread courant est bloqué et un nouveau thread est élu afin d'être exécuté par le processeur.
- Chaque interruption, ce timer peut provoquer un changement de contexte.
- **Sur une architecture **monoprocasseur**, masquer les interruptions garanti qu'une section de code ne soit pas interrompue (→ atomicité).**
- **Inconvénients :**
 - Fonctionne uniquement sur des architectures **monoprocesseurs**.
 - Empêche l'exécution d'I/O simultanées ⇒ dégradation des performances.

Algorithme par masquage d'interruptions

- Exclusion mutuelle en masquant, puis démasquant les interruptions à l'aide d'instructions assembleur :

```
void *thread(void *param) {  
    // Protocole d'entrée  
    cli // masque les interruptions (x86)  
  
    // Section critique  
    ...  
  
    // Protocole de sortie  
    sti // démasque les interruptions (x86)  
}
```

- La section critique doit **d'être courte**, sinon **risque** de pertes d'interruptions provenant des périphériques.

Attente active et passive

- **Attente active** : le temps processeur est gaspillé au test d'une condition (boucle) pour bloquer un thread.
 - **Attente passive** : le CPU s'endort (=faible consommation) jusqu'à ce qu'il reçoive une interruption. Lorsque celle-ci arrive, elle peut débloquent un thread.
-
- Les algorithmes vus précédemment utilisaient l'attente active.
 - L'attente passive étant préférable, les mécanismes suivants l'utilisent:
 - verrous ;
 - sémaphores ;
 - variables de condition.
 - Ces mécanismes facilitent la conception d'algorithmes, mais n'éliminent pas les risques d'erreurs !
 - L'OS doit les supporter

Récapitulatif des définitions

Opération atomique	Opération indivisible , c'est-à-dire exécutée en un seul bloc, du point de vue d'un thread.
Section critique	Section de code accédant à des ressources partagées accédée par un seul thread.
Ressource critique	Ressource non partageable accédée par plusieurs threads.
Interblocage (<i>deadlock</i>)	Situation où un ou plusieurs threads sont bloqués de manière permanente.
Exclusion mutuelle	Garantie qu'un seul thread au maximum peut s'exécuter en section critique.
Situation de concurrence (<i>race condition</i>)	Situation où plusieurs threads accèdent à une ressource partagée en même temps. Le résultat dépend du timing de l'exécution.
Famine (<i>starvation</i>)	Situation où un ou plusieurs threads n'ont pas la garantie d'accéder à leur section critique en un temps fini.

Ressources

- « **Modern multithreading** », Richard H. Carver, Kuo-Chung Tai, Wiley.
- « **Operating System Concepts (9th Edition)** », Avi Silberschatz, Peter Baer Galvin, Greg Gagne John Wiley & Sons, Inc.
- « **The Art of Multiprocessor Programming** » Maurice Herlihy, Nir Shavit – Editions Morgan Kaufmann Publishers
- gcc built-in functions for atomic memory access
<http://gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/Atomic-Builtins.html>