Programmation Concurrente

Principes et concepts
2023-2024

Chapitre4: Attente Active

Plan

- Définitions
- Règles fondamentales
- Tentatives
- Algorithme Dekker
- Algorithme Peterson

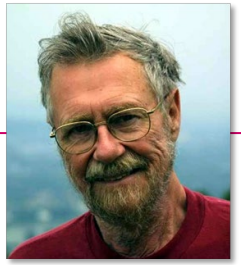
Définitions

Opération atomique	la séquence d'instructions est garantie d'être exécutée en un seul bloc malgré que les instructions apparaissent comme indivisibles;
Section critique	Section de code accédant à des ressources partagées et ne devant pas être exécutée concurremment par un autre thread.
Ressource critique	Ressource non partageable et accédée par plusieurs threads.
Interblocage (deadlock)	Situation où deux threads ou plus ne peuvent continuer leur exécution, car chacun attend sur l'autre.
Exclusion mutuelle	Garantie que si un thread entre dans une section critique accédant à des ressources partagées, alors aucun autre thread ne peut accéder à ces mêmes ressources partagées.
Situation de concurrence (race condition)	Situation où plusieurs threads lisent et écrivent une ressource partagée et le résultat final dépend du timing de l'exécution.
Famine (starvation)	Situation où un thread prêt à être exécuté n'est jamais élu par l'ordonnanceur.

Modèle typique

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

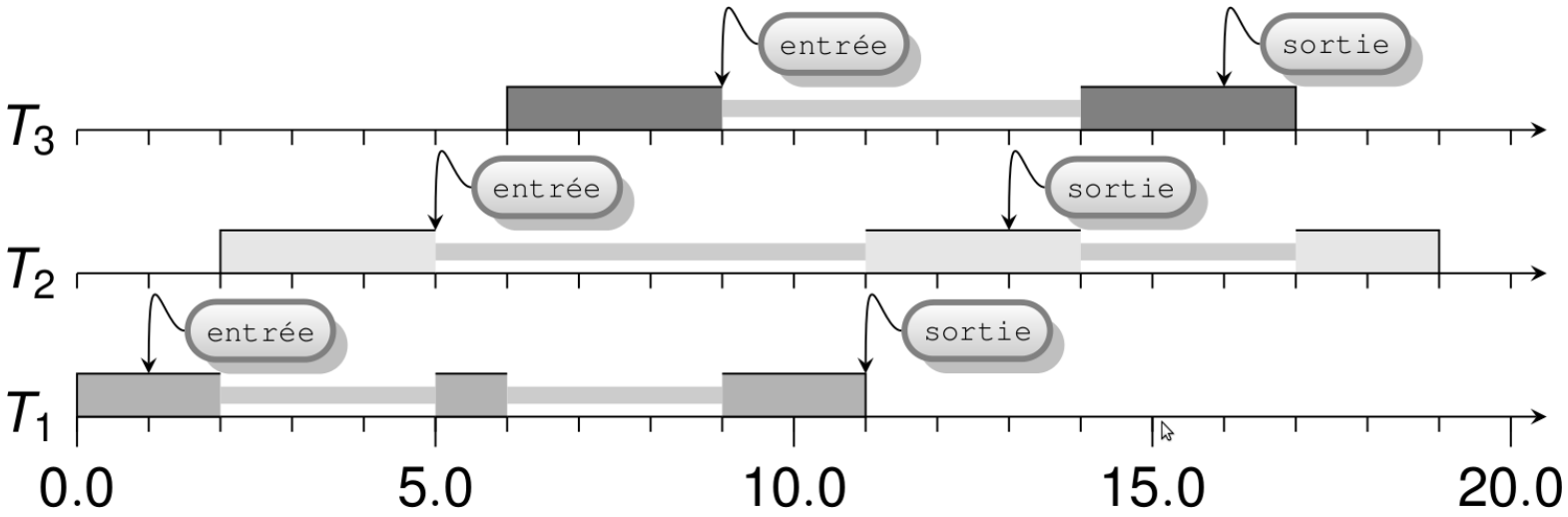
Algorithme



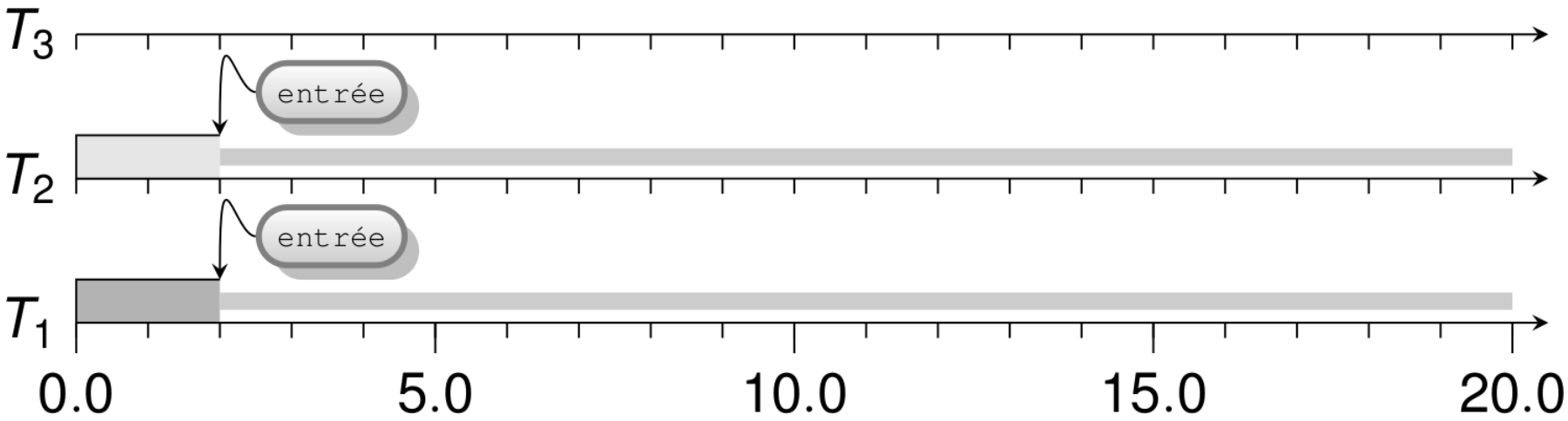
Edsger Wybe Dijkstra

- Tout algorithme d'exclusion mutuelle devrait satisfaire les propriétés suivantes :
  - Exclusion mutuelle : un seul thread à la fois est autorisé à exécuter le code de la section critique
  - Absence d'interblocage : aucun thread ne doit être bloqué indéfiniment
  - Absence de famine: tout thread tentant d'entrer en section critique doit éventuellement y entrer
  - Equité: aucun thread ne joue de rôle privilégié, la solution est la même pour tous.

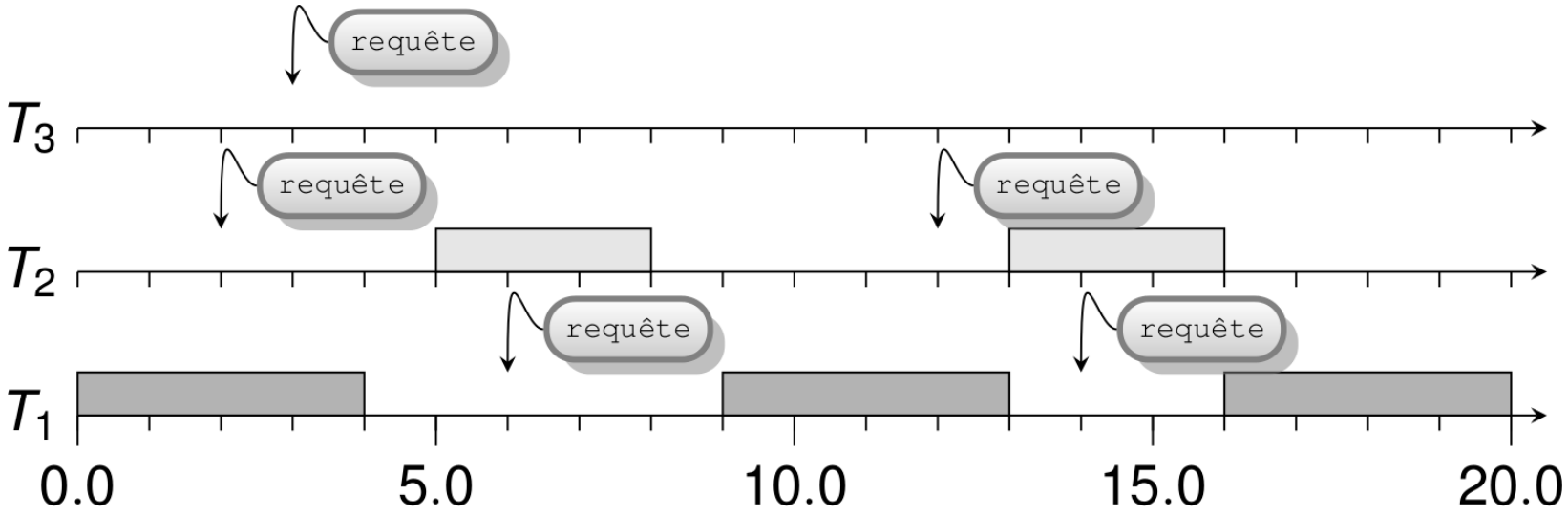
Exemple d'algorithme sans interblocage



Avec interblocage



Famine


requête
T_3
requête
requête
T_2
requête
requête
T_1
0.0
5.0
10.0
15.0
20.0

Propriétés des algorithmes

- Nos premiers algorithmes d'exclusion mutuelle dans les slides suivantes utilisent des instructions usuelles :
  - attente active par des boucles
  - variables partagées (globales)
  - ne gèrent que 2 threads concurrents (par simplicité)

- Ces algorithmes tentent d'assurer :
  - L'exclusion mutuelle
  - L'absence d'interblocage
  - L'absence de famine
  - … et d'éviter toute attente inutile

Algorithme1

```
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
  }
}
```

Est-ce que l'exclusion mutuelle est satisfaite ?

Algorithme1

```
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
  }
}
```

Est-ce que l'exclusion mutuelle est satisfaite ?

- Un seul thread en section critique ?

Algorithme1

```
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
  }
}
```

Est-ce que l'exclusion mutuelle est satisfaite ?

- Un seul thread en section critique ?

- 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 !

Algorithme1

```
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
  }
}
```

Est-ce que l'exclusion mutuelle est satisfaite ?

- Un seul thread en section critique ?

- 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 !

Les deux threads se trouvent en section critique

Algorithme2

```
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
  }
}
```

Est-ce que l'exclusion mutuelle est satisfaite ?

Algorithme2

```
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
  }
}
```

Est-ce que l'exclusion mutuelle est satisfaite ?

- Un seul thread en section critique ?

Algorithme2

```
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
  }
}
```

Est-ce que l'exclusion mutuelle est satisfaite ?

✓ Un seul thread en section critique

- Pas d'interblocage ?

Algorithme2

```
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 = 1;
    // section non-critique
  }
}
```

Est-ce que l'exclusion mutuelle est satisfaite ?

✓ Un seul thread en section critique

✓ Pas d'interblocage

Algorithme2

```
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;// turn à zero
    // section non-critique
  }
}
```

Est-ce que l'exclusion mutuelle est satisfaite ?

✓ Un seul thread en section critique

✓ Pas d'interblocage

- Le droit d'entrer dans la section critique d'un thread, dépend de l'autre thread

Algorithme2

```
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
  }
}
```

Est-ce que l'exclusion mutuelle est satisfaite ?

✓ Un seul thread en section critique

✓ Pas d'interblocage

- Le droit d'entrer dans la section critique d'un thread, dépend de l'autre thread

Couplage fort !

Algorithme3

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

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

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

Est-ce que l'exclusion mutuelle est satisfaite ?

Algorithme3

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

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

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

Est-ce que l'exclusion mutuelle est satisfaite ?

- Un seul thread en section critique ?

Algorithme3

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

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

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

Est-ce que l'exclusion mutuelle est satisfaite ?

- Un seul thread en section critique ?

- T0 lit `enter[1]` à `false`

- T1 lit `enter[0]` à `false`
- T1 met `enter[1]` à `true`
- T1 entre en section critique

- T0 met `enter[0]` à `true`
- T0 entre aussi en section critique !

Algorithme3

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

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

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

Est-ce que l'exclusion mutuelle est satisfaite ?

- Un seul thread en section critique ?

- T0 lit `enter[1]` à `false`

- T1 lit `enter[0]` à `false`
- T1 met `enter[1]` à `true`
- T1 entre en section critique

- T0 met `enter[0]` à `true`
- T0 entre aussi en section critique !

Exclusion mutuelle non satisfaite !

Tentative d'amélioration

- L'algorithme précédent ne fonctionne pas, car il peut se passer un temps arbitrairement long entre l'expression :

```
while (enter[1]) {}
```

et l'affectation :

```
enter[0] = true;
```

- L'algorithme qui suit propose d'avancer l'instruction d'affectation avant le **while** afin d'indiquer que le thread est en section critique

Algorithme4

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

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

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

Est-ce que l'exclusion mutuelle est satisfaite ?

Algorithme4

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

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

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

Est-ce que l'exclusion mutuelle est satisfaite ?

- Un seul thread en section critique ?

Algorithme4

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

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

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

Est-ce que l'exclusion mutuelle est satisfaite ?

✓ Un seul thread en section critique

Algorithme4

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

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

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

Est-ce que l'exclusion mutuelle est satisfaite ?

✓ Un seul thread en section critique

- Pas d'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

Algorithme4

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

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

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

Est-ce que l'exclusion mutuelle est satisfaite ?

✓ Un seul thread en section critique

- Pas d'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 !

Tentative d'amélioration

- Dans l'algorithme précédent, chaque thread signalait son intention d'entrer en section critique ce qui pouvait entraîner un interblocage.
- La solution suivante consiste à faire capituler un des threads.
- Si cela est nécessaire, chaque thread retire temporairement son intention d'entrer en section critique pendant un court instant.

Algorithme5

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

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

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

Est-ce que l'exclusion mutuelle est satisfaite ?

- Un seul thread en section critique ?

Algorithme5

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

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

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

Est-ce que l'exclusion mutuelle est satisfaite ?

✓ Un seul thread en section critique

Algorithme5

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

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

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

Est-ce que l'exclusion mutuelle est satisfaite ?

✓ Un seul thread en section critique

- Pas d'interblocage ?

Algorithme5

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

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

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

Est-ce que l'exclusion mutuelle est satisfaite ?

✓ Un seul thread en section critique

✓ Pas d'interblocage

- Pas de famine ?

Algorithme5

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

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

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

Est-ce que l'exclusion mutuelle est satisfaite ?

✓ Un seul thread en section critique

X Interblocage possible

- Pas de famine ?

Algorithme5

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

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

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

Est-ce que l'exclusion mutuelle est satisfaite ?

- ✓ Un seul thread en section critique
- ✓ Pas d'interblocage

- Pas de famine ?

- T0 met `enter[0]` à `true`

- T1 met `enter[1]` à `true`
- T1 entre dans la boucle `while`
- T1 met `enter[1]` à `false`
- T1 attend un peu
- T1 met `enter[1]` à `true`

- T0 entre dans la boucle `while`
- T0 met `enter[0]` à `false`
- T0 attend un peu
- T0 met `enter[0]` à `true`

Algorithme5

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

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

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

Est-ce que l'exclusion mutuelle est satisfaite ?

✓ Un seul thread en section critique

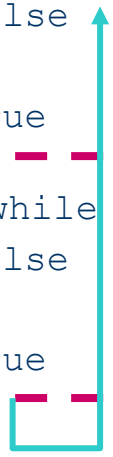
X Interblocage possible

- Pas de famine ?

- T0 met `enter[0]` à `true`

- T1 met `enter[1]` à `true`
- T1 entre dans la boucle `while`
- T1 met `enter[1]` à `false`
- T1 attend un peu
- T1 met `enter[1]` à `true`

- T0 entre dans la boucle `while`
- T0 met `enter[0]` à `false`
- T0 attend un peu
- T0 met `enter[0]` à `true`



Algorithme5

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

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

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

Est-ce que l'exclusion mutuelle est satisfaite ?

- ✓ Un seul thread en section critique
- ✓ Pas d'interblocage
- Pas de famine ?

- T0 met `enter[0]` à `true`
- T1 met `enter[1]` à `true`
- T1 entre dans la boucle `while`
- T1 met `enter[1]` à `false`
- T1 attend un peu
- T1 met `enter[1]` à `true`
- T0 entre dans la boucle `while`
- T0 met `enter[0]` à `false`
- T0 attend un peu
- T0 met `enter[0]` à `true`

Famine !

Algorithme de Dekker et Peterson

Algorithmes de Dekker(1965) et de Peterson (1981)

Principe :

- Chaque processus annonce sa candidature à l'autre processus
- En cas de candidatures simultanées, le conflit est réglé en donnant la priorité à un processus
- Pour une solution équitable la priorité doit être variable

Généralisation difficile pour Dekker, possible pour Peterson

Algorithme de Dekker

```
typedef enum {enter,leave} intention_t;
intention_t intention[2] = {leave,leave};
int turn = 0;

void *T0(void *arg){
  while (true) {
    intention[0] = enter;
    while (intention[1] == enter) {
      if (turn != 0) {
        intention[0] = leave;
        while (turn != 0) {}
        intention[0] = enter;
      }
    }
    // section critique
    turn = 1;
    intention[0] = leave;
    // section non-critique
  }

}
```

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

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
  }
}
```

Résultats

- Les algorithmes de Peterson et Dekker sont correct mais ils peuvent ne pas fonctionner sur des processeurs multi-cœur
- Le problème est lié la mémoire cache de chaque processeur
- L'ordre des accès mémoires
- **Pour remédier il faut des barrières de synchronisation**

Algorithme de masquage

- Exclusion mutuelle

```
void *thread(void *param) {
    // Protocole d'entrée
    asm("cli");  // masque les interruptions(x86)

    // Section critique
    ...

    // Protocole de sortie
    asm("sti");  // démasque les interruptions (x86)
}
```

Instructions matérielles

- Les processeurs modernes implémentent diverses instructions matérielles atomiques pour palier à ces problèmes et rendre les algorithmes d'exclusion mutuelle plus facilement implémentables et robustes:
  - Instructions TestAndSet (TAS) : teste et modifie de manière atomique une variable.
  - Instructions Compare And Swap (CAS): compare et échange d'une manière atomique deux variables
  - Instructions d'échange

Test And Set

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

```
int test_and_set (int *verrou){
  int old = *verrou;
  *verrou = 1;
   return old;
 }
void *thread(void *param) {
    // Protocole d'entrée

    // Section critique
    ...

    // Protocole de sortie
    }
```

Test And Set

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

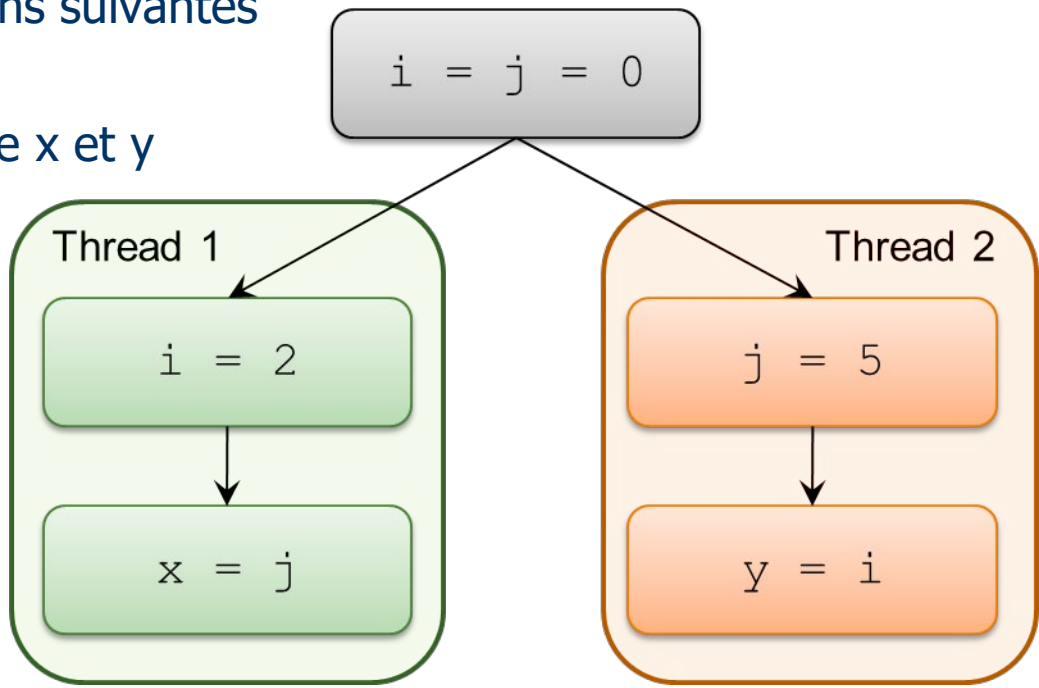
```
int test_and_set (int *verrou){
  int old = *verrou;
  *verrou = 1;
   return old;
 }
int lock = 0;

void *thread(void *param) {
    // Protocole d'entrée
    while (test_and_set(&lock)) {}
    // Section critique
    ...
    lock =0

    // Protocole de sortie
    }
```

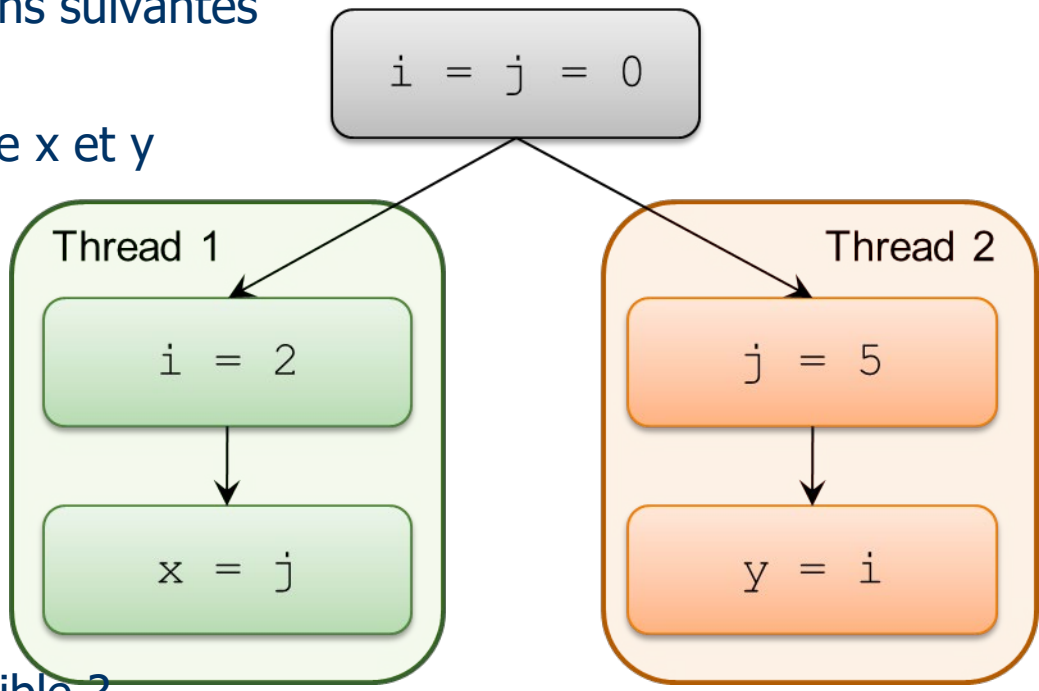
Cohérence séquentielle

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



Cohérence séquentielle

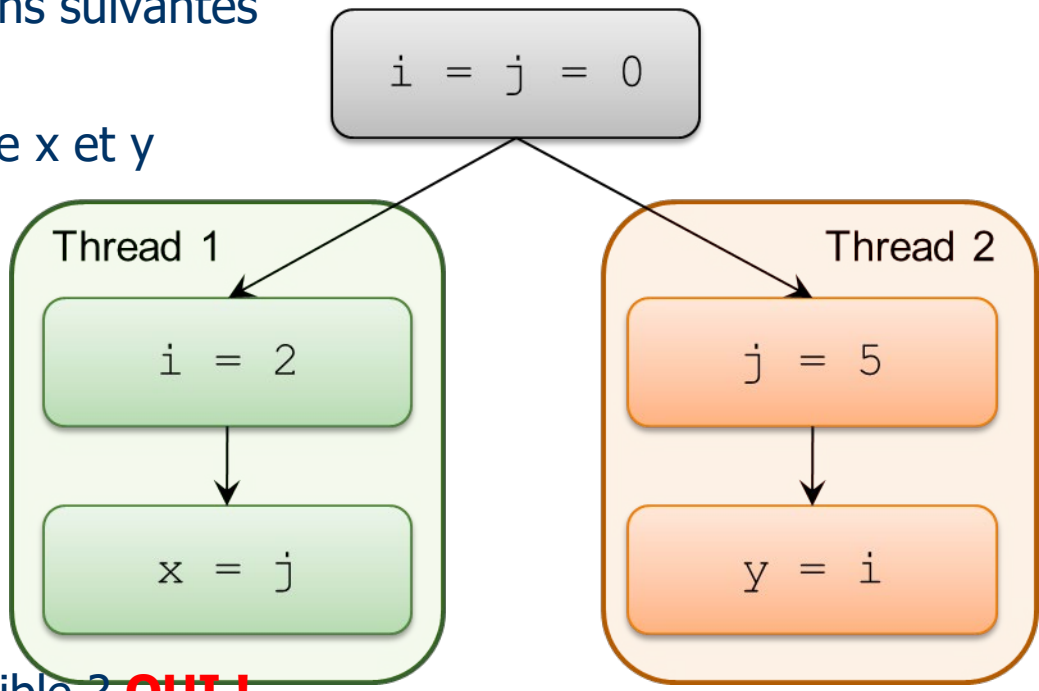
- Soit 2 threads exécutant les instructions suivantes
- Quelles sont les valeurs potentielles de x et y une fois les 2 threads terminés ?



`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 !**

Optimisation du code ⇨ le compilateur peut réarranger le code afin d'optimiser l'accès aux variables, car les instructions de chaque thread sont indépendantes !

Cohérence séquentielle

- Afin d'obtenir du code exécuté plus efficace, le compilateur effectue diverses opérations d'optimisation (variables dans des registres, réordonnancement instructions pour éviter accès mémoire successifs, suppression variables, déplacement de code, etc.).

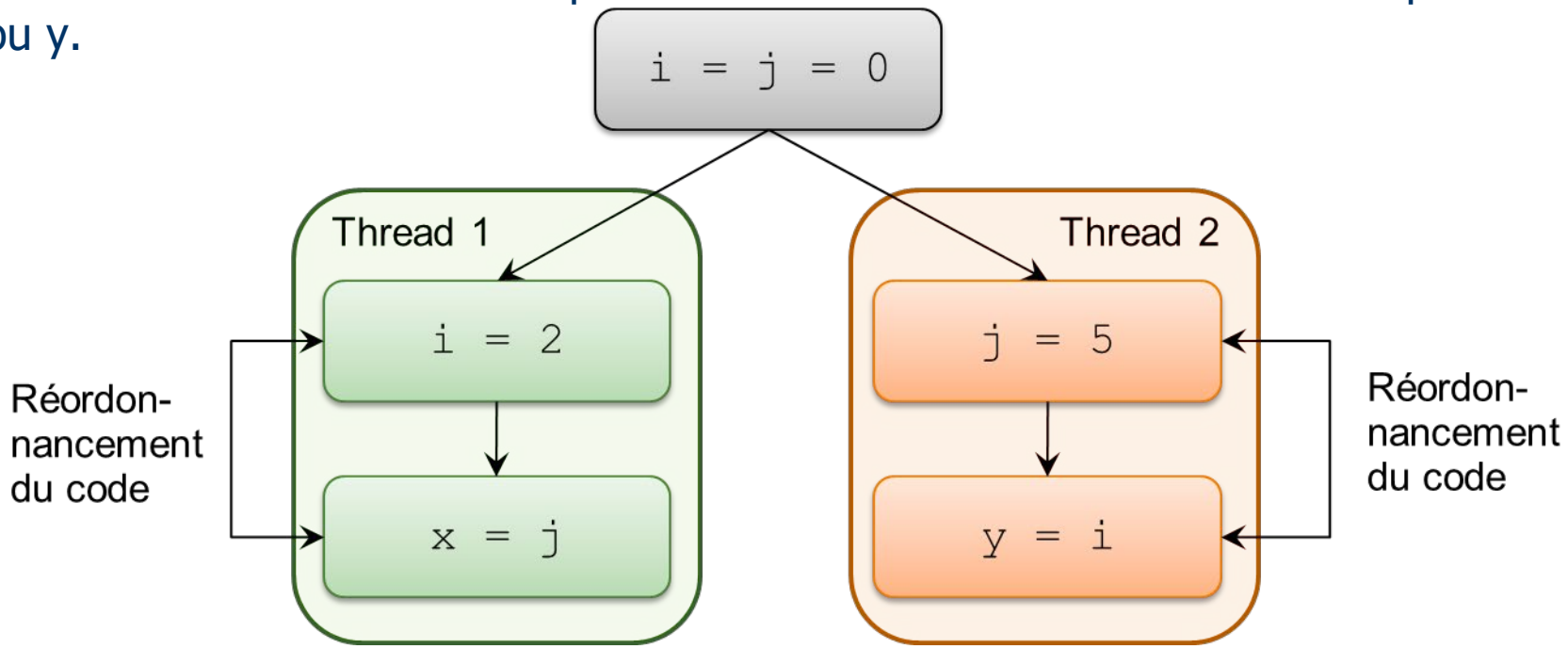
- Ces optimisations ne changent pas le comportement du code dans un contexte séquentiel.

- 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 !**

Volatile

- Dans un contexte séquentiel local à chaque thread, les instructions ci-dessous peuvent être inversées sans que cela n'affecte la valeur de i ou x et respectivement j ou y.



- Déclarer une variable volatile permet de forcer le compilateur à n'effectuer aucune optimisation sur la variable !

Attente active vs passive

- **Attente active** : le temps processeur est gaspillé au test d'une condition (boucle) pour bloquer un thread
- **Attente passive** : un thread est bloqué par le noyau et un autre thread est ordonnancé (processeur disponible)

- Les algorithmes précédents utilisent l'attente active.
- L'attente passive nécessite des constructions spéciales :
  - verrous;
  - sémaphores;
  - variables de condition.
- Ces constructions facilitent la conception d'algorithmes, mais n'éliminent pas les risques d'erreurs !
- Le noyau doit les supporter