Introduction à la programmation concurrente

Exclusion mutuelle par attente active

Yann Thoma, Fiorenzo Gamba

Février 2021

Reconfigurable and Embedded Digital Systems Institute
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

Ressource critique

Ressource critique

- Une ressource critique est une ressource non partageable et accédée par plusieurs tâches.

Ressource critique

- Une ressource critique est une ressource non partageable et accédée par plusieurs tâches.
- Exemples:
  - une variable globale (ressource logique) accédée en lecture et en écriture par des tâches;
  - une ressource physique (périphérique) accédée par des tâches.

Ressource critique: exemple

```
static int counter = 0;
const int NB_ITERATIONS = 1000000;

void run() {
    for(int i = 0; i < NB_ITERATIONS; i++) {
        counter = counter + 1;
    }
}

int main(int argc, char *argv[])
{
    std::vector<PcoThread*> threads;
    for(int i = 0; i < 2; i++)
        threads.push_back(new PcoThread(run, i));
    for(int i = 0; i < 2; i++)
        threads[i]->join();

    std::cout << "Fin des taches : counter = " << counter
        << " (" << 2 * NB_ITERATIONS << ")" << std::endl;

    return 0;
}
```

Exemple d'exécution

```
Fin de Tache 2
Fin de Tache 1
Fin des taches: counter = 2000000 (2000000)
```

```
Fin de Tache 1
Fin de Tache 2
Fin des taches: counter = 1405922 (2000000)
```

```
Fin de Tache 1
Fin de Tache 2
Fin des taches: counter = 2000000 (2000000)
```

Explication

- Comment est traduit `counter = counter + 1` ?
- En langage machine:

```
mov counter,r1    % copie la valeur de la mémoire
                  % commune désignée par counter
                  % dans un registre local r1;

addi 1,r1         % ajoute 1 au registre;

mov r1,counter    % stocke le contenu du registre
                  % local r1 à l'adresse de
                  % counter
```

Explication

Exemple d'une exécution erronée

Tâche 1	$counter = 10$	Tâche 2
$r1 \leftarrow 10$		
	changement de contexte	
		$r1 \leftarrow 10$
		$r1 \leftarrow r1 + 1$
		$counter \leftarrow r1$ (vaut 1
	changement de contexte	
$r1 \leftarrow r1 + 1$		
$counter \leftarrow r1$		
	counter vaut ainsi 11 et non 12!	

Question

Question

Quelle est la valeur théorique minimale possible de la variable *counter*?

- 1'000'000
- 500'000
- 2
- 1

Réponse

Valeur minimale de *counter*

Tâche 1	Tâche 2
$r1 \leftarrow counter = 0$	

Réponse

Valeur minimale de *counter*

Tâche 1	Tâche 2
$r1 \leftarrow counter = 0$	
	...
	$counter = 999'999$
$r1 \leftarrow r1 + 1 = 1$	
$counter \leftarrow r1 = 1$	
	$r1 \leftarrow counter = 1$ (dernière itération, counter écrasé)
...	
$counter = 1'000'000$	
	$r1 \leftarrow r1 + 1 = 2$
	$counter \leftarrow r1 = 2$

Section critique

- Une ressource critique doit être exécutée en exclusion mutuelle (c.-à-d. les accès à la ressource s'excluent mutuellement)
- La portion de code décrivant un accès à une ressource critique (RC) est appelée *section critique*
- L'exclusion mutuelle doit être assurée dans une section critique
- L'accès à une section critique est géré par un algorithme d'exclusion mutuelle en deux parties:
  - protocole d'entrée
  - protocole de sortie

Section critique

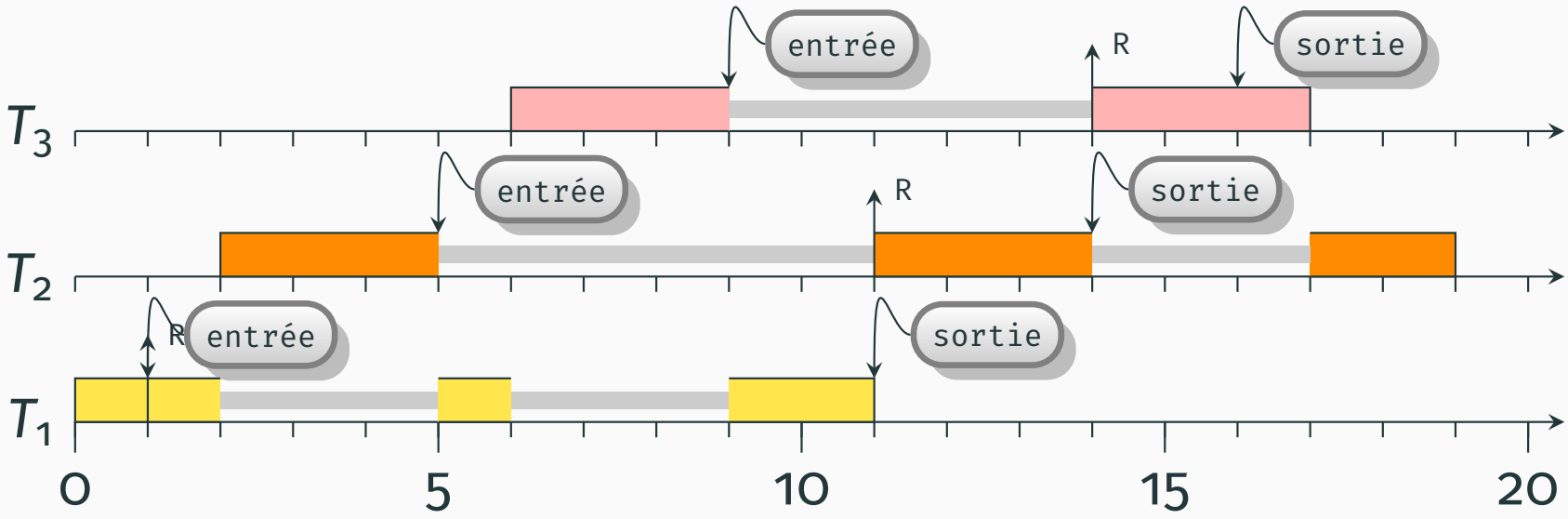
Modèle typique pour une tâche

```
 void T(void* arg) {
  déclarations locales;
  instructions;
  while (true) {
    instructions;
    <prélude>           // Protocole d'entrée
    <section critique> // Accès à la RC
    <postlude>          // Protocole de sortie
    instructions;
  }
}
```

Propriétés des algorithmes

Eviter les interblocages

Les tâches doivent pouvoir avancer (*liveness*): éviter l'interblocage (*deadlock*). Exemple correct:

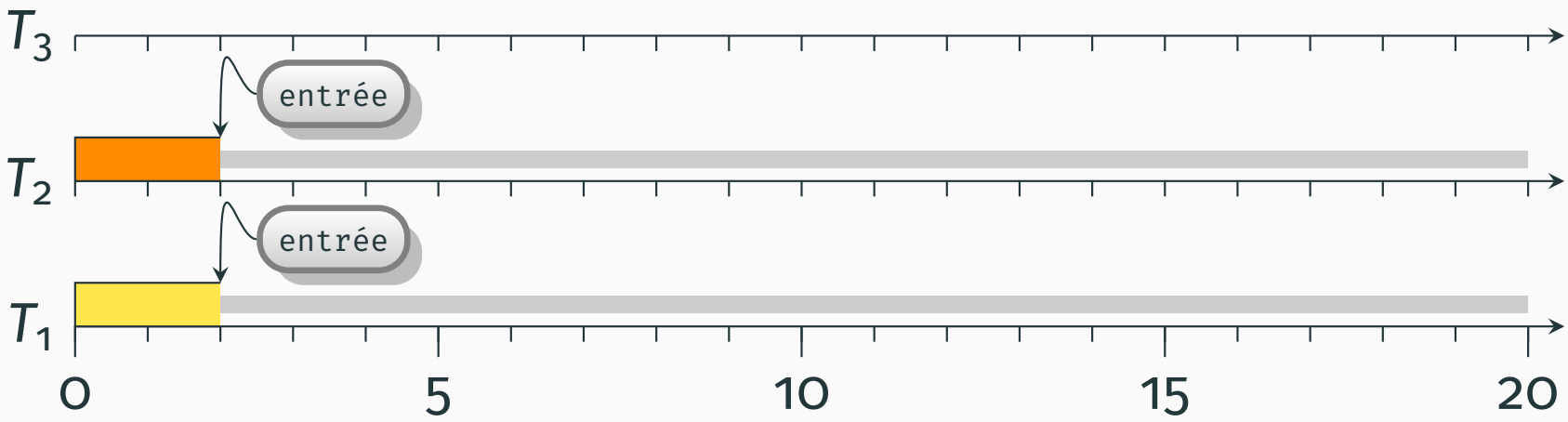


Légende

- Gris: tâche suspendue
- Couleur: tâche s'exécute (1 seul coeur)
- R: obtention de la ressource en accès exclusif

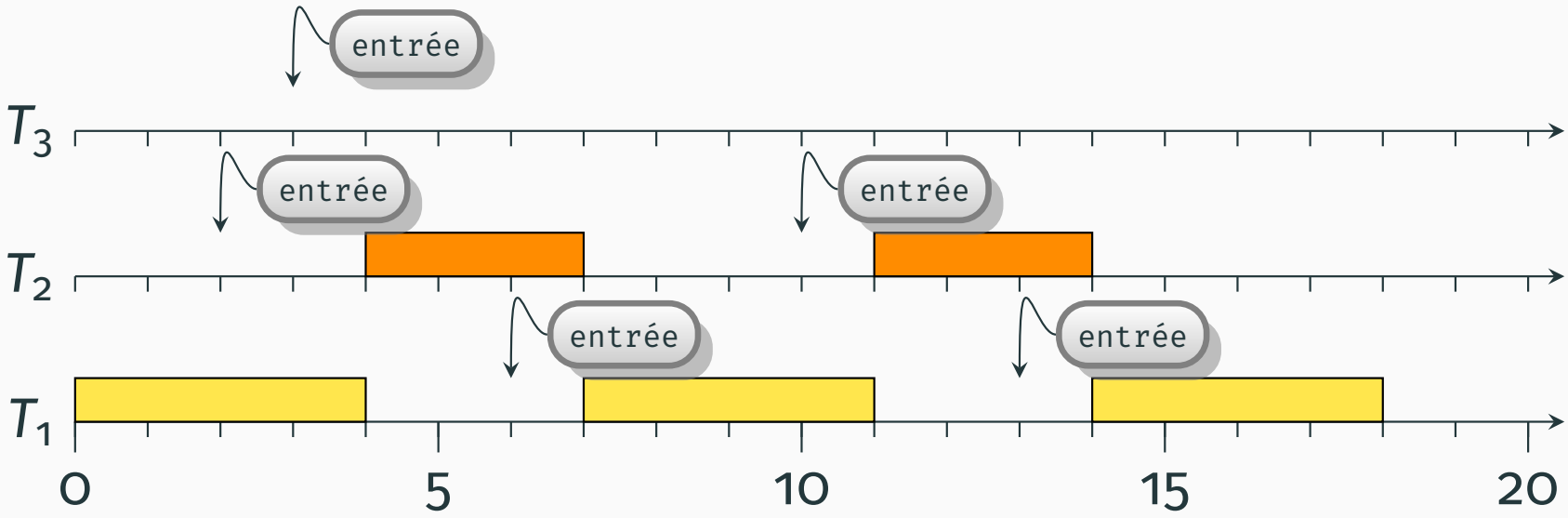
Exemple d'interblocage

Exemple incorrect (avec interblocage):



Eviter la famine

Il faut éviter la famine (*starvation*): les autres tâches se "liguent" pour empêcher une tâche d'accéder à la ressource



Légende

- Couleur: obtention de la ressource en accès exclusif

Protocole d’entrée et attente

- Plusieurs tâches peuvent réclamer une ressource

Protocole d'entrée et attente

- Plusieurs tâches peuvent réclamer une ressource
- L'attente peut être:
  - *FIFO*
  - *Linéaire*: une tâche ne peut accéder deux fois la ressource si une autre est en attente
  - *bornée par une fonction* $f(n)$: pour n tâches, une tâche en attente ne peut se faire dépasser par $f(n)$ tâches
  - *finie*: l'attente n'est pas bornée mais pas infinie

Propriété des algorithmes

Règles pour la mise au point des algorithmes:

1. A tout instant, une seule tâche peut se trouver en section critique

Propriété des algorithmes

Règles pour la mise au point des algorithmes:

1. A tout instant, une seule tâche peut se trouver en section critique
2. Si plusieurs tâches sont bloquées en attente d'entrer en section critique alors qu'aucune tâche ne s'y trouve, l'une d'entre elles doit pouvoir y accéder au bout d'un temps fini (pas d'interblocage)

Propriété des algorithmes

Règles pour la mise au point des algorithmes:

1. A tout instant, une seule tâche peut se trouver en section critique
2. Si plusieurs tâches sont bloquées en attente d'entrer en section critique alors qu'aucune tâche ne s'y trouve, l'une d'entre elles doit pouvoir y accéder au bout d'un temps fini (pas d'interblocage)
3. Le comportement d'une tâche en dehors de la section critique et des protocoles qui en gèrent l'accès n'a aucune influence sur l'algorithme d'exclusion mutuelle

Propriété des algorithmes

Règles pour la mise au point des algorithmes:

1. A tout instant, une seule tâche peut se trouver en section critique
2. Si plusieurs tâches sont bloquées en attente d'entrer en section critique alors qu'aucune tâche ne s'y trouve, l'une d'entre elles doit pouvoir y accéder au bout d'un temps fini (pas d'interblocage)
3. Le comportement d'une tâche en dehors de la section critique et des protocoles qui en gèrent l'accès n'a aucune influence sur l'algorithme d'exclusion mutuelle
4. Aucune tâche ne joue de rôle privilégié, la solution est la même pour toutes

Tentatives d’algorithme

Tentatives d’algorithme

- Nos premiers algorithmes d’exclusion mutuelle utilisent des instructions usuelles :
  - l’attente active par des boucles
  - des variables partagées (globales)

Tentatives d'algorithme

- Nos premiers algorithmes d'exclusion mutuelle utilisent des instructions usuelles :
  - l'attente active par des boucles
  - des variables partagées (globales)
- Ils possèdent des risques d'erreurs comme
  - exclusion mutuelle non satisfaite
  - interblocage
  - famine

Tentatives d'algorithme

- Nos premiers algorithmes d'exclusion mutuelle utilisent des instructions usuelles :
  - l'attente active par des boucles
  - des variables partagées (globales)
- Ils possèdent des risques d'erreurs comme
  - exclusion mutuelle non satisfaite
  - interblocage
  - famine
- Ils tentent
  - d'éviter toute attente inutile
  - d'assurer l'équité si possible

Tentative 1

```
bool occupe = false;

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

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

Pas d'exclusion	Interblocage	Famine	Couplage

Tentative 1

```
bool occupe = false;

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

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

Pas d'exclusion	Interblocage	Famine	Couplage
✗			

Tentative 2

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

void T0()
{
  while (true) {
    while (tour != 0)
      ;
    /* section critique */
    tour = 1;
    /* section non-critique */
  }
}
```

```
void T1()
{
  while (true) {
    while (tour != 1)
      ;
    /* section critique */
    tour = 0;
    /* section non-critique */
  }
}
```

Pas d'exclusion	Interblocage	Famine	Couplage

Tentative 2

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

void T0()
{
  while (true) {
    while (tour != 0)
      ;
    /* section critique */
    tour = 1;
    /* section non-critique */
  }
}
```

```
void T1()
{
  while (true) {
    while (tour != 1)
      ;
    /* section critique */
    tour = 0;
    /* section non-critique */
  }
}
```

Pas d'exclusion	Interblocage	Famine	Couplage
		✗	✗

Tentative 3

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

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

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

Pas d'exclusion	Interblocage	Famine	Couplage

Tentative 3

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

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

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

Pas d'exclusion	Interblocage	Famine	Couplage
✗			

Tentative 4

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

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

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

Pas d'exclusion	Interblocage	Famine	Couplage

Tentative 4

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

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

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

Pas d'exclusion	Interblocage	Famine	Couplage
	✗		

Tentative 5

```
bool etat[2] = {false,false};
void T0()
{
  while (true) {
    etat[0] = true;
    while (etat[1]) {
      etat[0] = false;
      while (etat[1])
        ;
      etat[0] = true;
    }
    /* section critique */
    etat[0] = false;
    /* section non-critique */
  }
}
```

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

Pas d'exclusion	Interblocage	Famine	Couplage

Tentative 5

```
bool etat[2] = {false,false};
void T0()
{
  while (true) {
    etat[0] = true;
    while (etat[1]) {
      etat[0] = false;
      while (etat[1])
        ;
      etat[0] = true;
    }
    /* section critique */
    etat[0] = false;
    /* section non-critique */
  }
}
```

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

Pas d'exclusion	Interblocage	Famine	Couplage
	✗		

Algorithme de Dekker

Algorithme de Dekker

Concept

```
void To()
{
  while (true)
  {
    setEnSectionCritique();
    while(autreEnSectionCritique()) {
      if (pasMonTour()) {
        clearEnSectionCritique();
        while(pasMonTour())
          ;
        setEnSectionCritique();
      }
    }
    /* section critique */
    passeMonTour();
    clearEnSectionCritique();
    /* section non-critique */
  }
}
```

Algorithme de Dekker

Concept

```
void T0()
{
  while (true)
  {
    setEnSectionCritique();
    while(autreEnSectionCritique()) {
      if (pasMonTour()) {
        clearEnSectionCritique();
        while(pasMonTour())
          ;
        setEnSectionCritique();
      }
    }
    /* section critique */
    passeMonTour();
    clearEnSectionCritique();
    /* section non-critique */
  }
}
```

Implémentation

```
bool etat[2] = {false,false};
int tour = 0; // ou 1

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

Algorithme de Dekker: implémentation complète

```
bool etat[2] = {false,false};
int tour = 0; // ou 1

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

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

Algorithme de Peterson

Algorithme de Peterson

Concept

```
void To()
{
  while (true) {
    setIntention();
    setTourAutre();
    while (intentionAutre() && pasMonTour())
      ;
    /* section critique */
    clearIntention();
    /* section non-critique */
  }
}
```

Algorithme de Peterson

Concept

```
void T0()
{
  while (true) {
    setIntention();
    setTourAutre();
    while (intentionAutre() && pasMonTour())
      ;
    /* section critique */
    clearIntention();
    /* section non-critique */
  }
}
```

Implémentation

```
bool intention[2] = {false,false};
int tour = 0; // ou 1

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

Algorithme de Peterson: implémentation complète

```
bool intention[2] = {false,false};
int tour = 0; // ou 1

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

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

Analyse

- Mathématiquement parlant, les algorithmes de Peterson et Dekker sont corrects
- Mais les processeurs multi-coeur les mettent à mal:
  - Cohérence des mémoires caches
  - Ordre des accès mémoires

$\Rightarrow$ Ils ne sont plus sûrs. Il faut placer des barrières de synchronisation mémoire pour qu'ils fonctionnent

- En C++:

  ```
  std::atomic_thread_fence(std::memory_order_acq_rel)
  ```

  Garantie que tous les load et store présents avant la barrière sont effectués avant les load et store présents après

Algorithme de Dekker: implémentation multicore

```
#define BARRIER std::atomic_thread_fence(std::memory_order_acq_rel)
```

```
bool etat[2] = {false,false};
int tour = 0; // ou 1

void T0()
{
  while (true) {
    etat[0] = true;
    BARRIER;
    while (etat[1])
      if (tour == 1) {
        etat[0] = false;
        while (tour == 1)
          ;
        etat[0] = true;
        BARRIER;
      }
    /* section critique */
    tour = 1;
    etat[0] = false;
    /* section non-critique */
  }
}
```

```
void T1()
{
  while (true) {
    etat[1] = true;
    BARRIER;
    while (etat[0])
      if (tour == 0) {
        etat[1] = false;
        while (tour == 0)
          ;
        etat[1] = true;
        BARRIER;
      }
    /* section critique */
    tour = 0;
    etat[1] = false;
    /* section non-critique */
  }
}
```

Algorithme de Peterson: implémentation multicore

```
bool intention[2] = {false,false};
int tour = 0; // ou 1

void T0()
{
  while (true) {
    intention[0] = true;
    tour = 1;
    BARRIER;
    while (intention[1] && tour == 1)
      ;
    /* section critique */
    intention[0] = false;
    /* section non-critique */
  }
}
```

```
void T1()
{
  while (true) {
    intention[1] = true;
    tour = 0;
    BARRIER;
    while (intention[0] && tour == 0)
      ;
    /* section critique */
    intention[1] = false;
    /* section non-critique */
  }
}
```

Propriété des algorithmes

Définitions

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

- Les algorithmes proposés utilisent l'attente active
- L'attente passive nécessite des constructions spéciales
  - Verrous
  - Sémaphores
  - Moniteurs
- Ces constructions facilitent la conception d'algorithmes, mais n'éliminent pas le risque d'erreurs!
- Le noyau doit les supporter (états de processus ou de thread)

Exercice

```
bool intention[2] = {false,false};
int tour = 0; // ou 1

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

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

- L'exclusion mutuelle est-elle garantie par les parties prélude et postlude des tâches? Justifiez votre réponse.

Code source

```
run:../code/3-exclusion/increment
run:../code/3-exclusion/exclusion1
run:../code/3-exclusion/exclusion1
run:../code/3-exclusion/exclusion2
run:../code/3-exclusion/exclusion2
run:../code/3-exclusion/exclusion3
run:../code/3-exclusion/exclusion3
run:../code/3-exclusion/exclusion4
run:../code/3-exclusion/exclusion4
run:../code/3-exclusion/exclusion5
run:../code/3-exclusion/exclusion5
run:../code/3-exclusion/dekker
run:../code/3-exclusion/peterson
run:../code/3-exclusion/dekker_multicore
run:../code/3-exclusion/peterson_multicore
```