

# Programmation Concurrente

Département Informatique

IUT2 de Grenoble

Semestre 6

# Apple A16 (Iphone 14) vs. A15 (Iphone 13) SOC



# Intel Core i9 Raptor Lake S



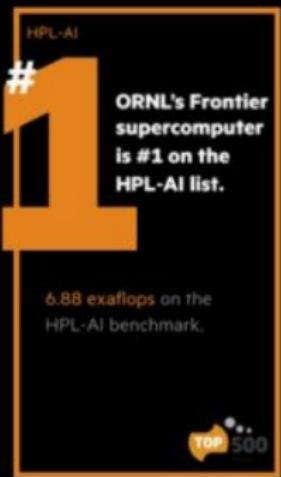
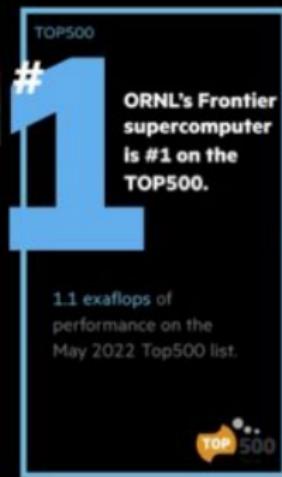
# Frontier (ORNL)



## OAK RIDGE NATIONAL LABORATORY'S FRONTIER SUPERCOMPUTER

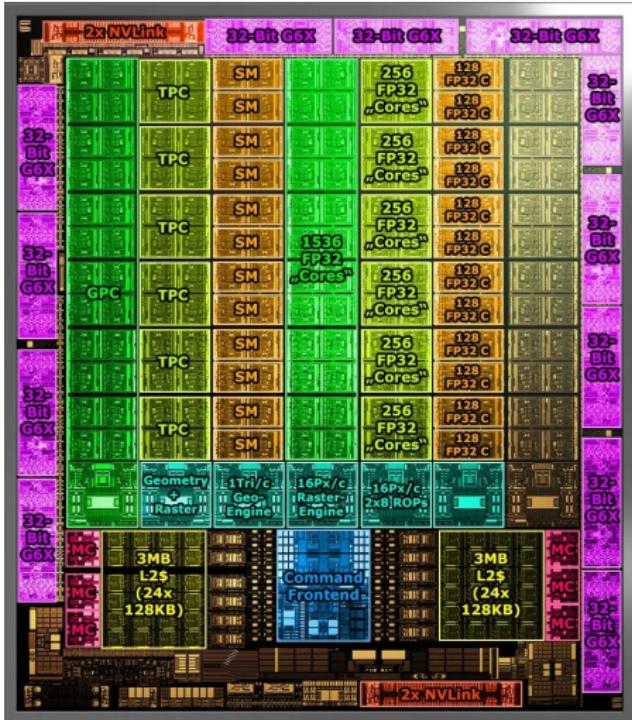
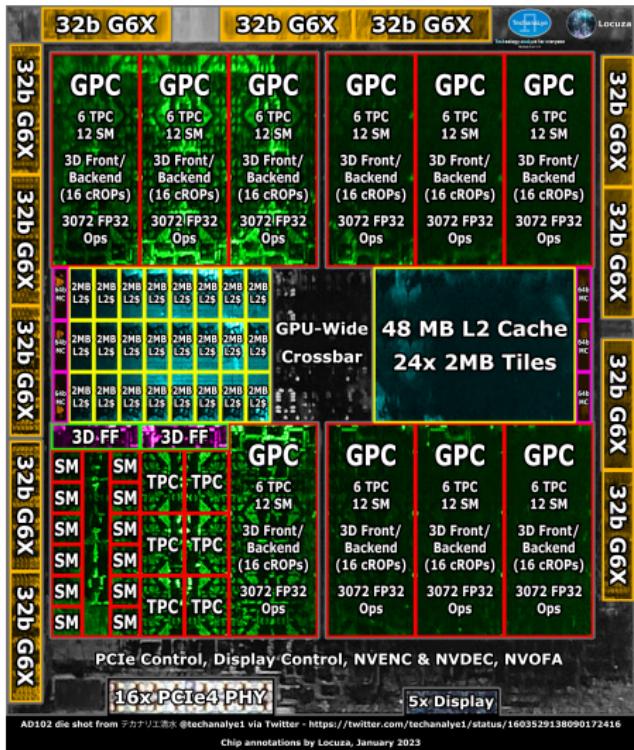


- 74 HPE Cray EX cabinets
- 9,472 AMD EPYC CPUs, 37,888 AMD GPUs
- 700 petabytes of storage capacity, peak write speeds of 5 terabytes per second using Cray Clusterstor Storage System
- 90 miles of HPE Slingshot networking cables



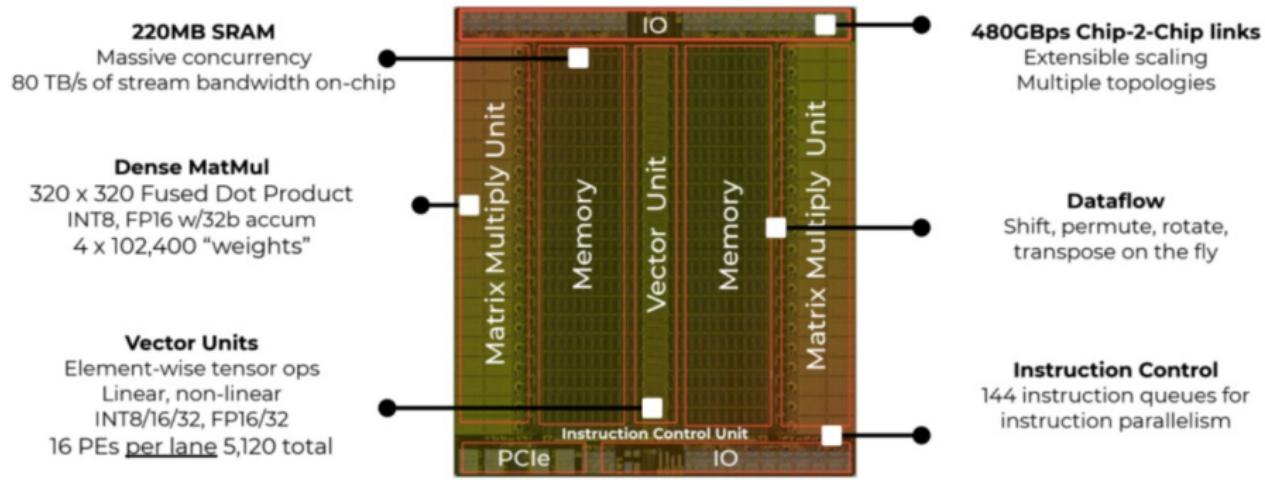
Sources: May 30, 2022 Top500 release

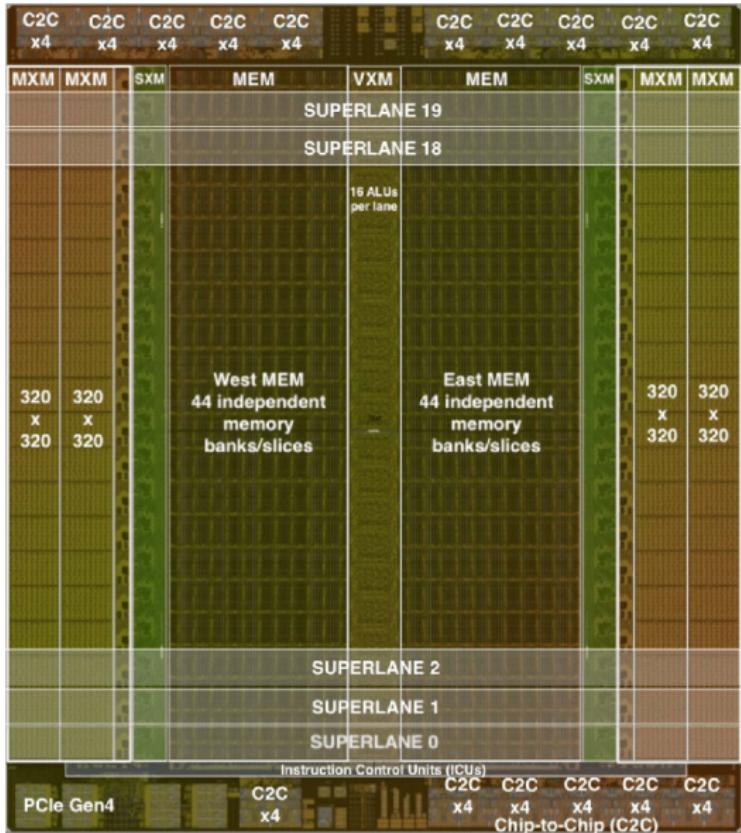
# GPU Nvidia



## Tensor Streaming Processor at a Glance

### Groq TSP™, Scalable Architecture

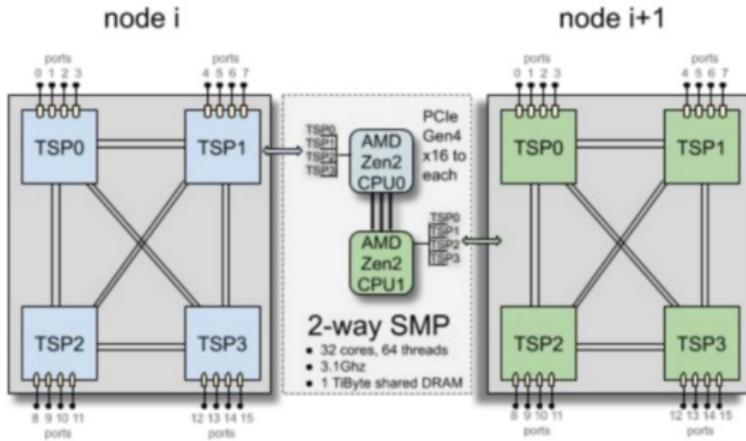


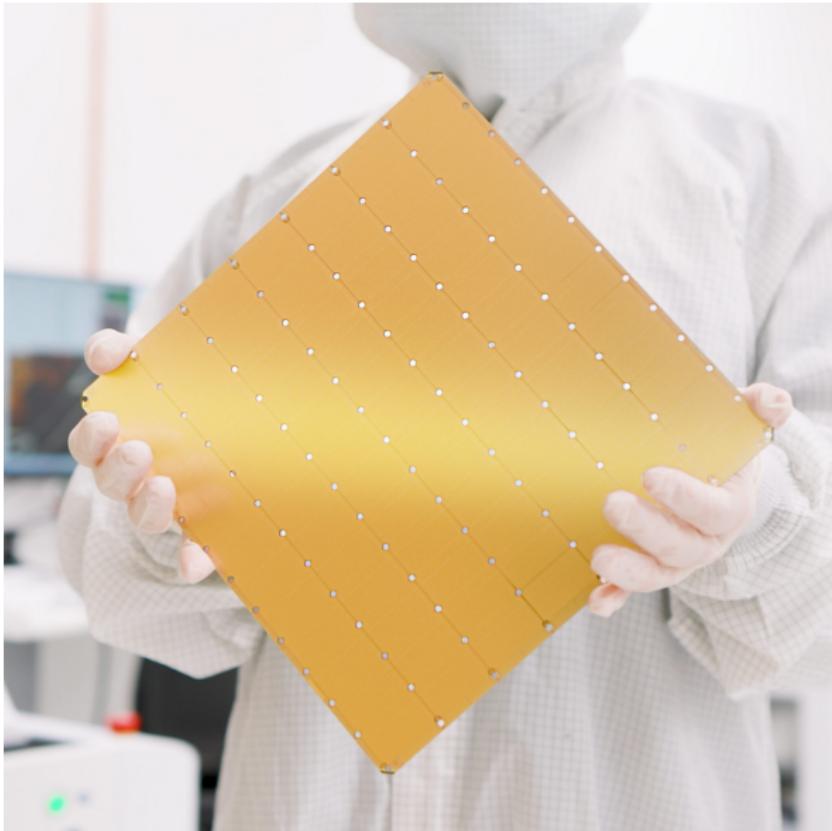


## The Groq Node Organization

### Dense Compute Platform

- 1.5 PetaFlops FP16 (FP32 ACC) and 6 PetaOps INT8
- Contains 8 Groq TSP100 cards in a 4U chassis
- Configured as 8 cards: 2x4 cards or 8 independent cards
- Dual-socket AMD 'Rome' server with 160 PCIe lanes
- 3.3 KWs of power





# Organisation de l'enseignement

sujet	semaine	Cours	TP
Intro, Threads et Mutex	01	2h	2 × 2h
Variable Condition, Moniteurs	02	2h	2 × 2h
Synchronisation par les données	03		2 × 2h

## Evaluation

QCM (sem. 05)

Encadrants [[Prenom.Nom@univ-grenoble-alpes.fr](mailto:Prenom.Nom@univ-grenoble-alpes.fr)]

- Cours : **Pierre-Francois.Dutot**
- TP : **Laurent.Bonnaud P-F.Dutot Cedric.Gerot**

## Langage utilisé

C++

## Programmation Concurrente

1. Thread
2. Section critique et mutex
3. Moniteur
4. Moniteur (avancé)

## Programmation Concurrente

1. Thread
2. Section critique et mutex
3. Moniteur
4. Moniteur (avancé)

# 1 . Thread

## 1.1. Plusieurs exécutions pour un même espace mémoire

- thread : exécution non-exclusive d'un programme
- espace mémoire partagé... si mémoire synchronisée

## 1.2. C++ **std::thread**

- création **std::thread::thread()**
- destruction après détachement **std::thread::join()**  
**std::thread::detach()**
- création de thread vs. processus

# 1 . Thread

## 1.1. Plusieurs exécutions pour un même espace mémoire

- thread : exécution non-exclusive d'un programme
- espace mémoire partagé... si mémoire synchronisée

## 1.2. C++ **std::thread**

- création **std::thread::thread()**
- destruction après détachement **std::thread::join()**  
**std::thread::detach()**
- création de thread vs. processus

# thread : exécution non exclusive d'un programme

## Des threads pour quoi faire ?

- plusieurs parties d'un programme exécutées simultanément
- un même espace mémoire partagé entre les exécutions

## Les threads **d'un** processus partagent

en espace mémoire :

- text (code)
- data (variables globales)

dans le descripteur :

- PID, PPID, ...
- la gestion des signaux
- les fichiers ouverts

## Les threads **d'un** processus ne partagent pas

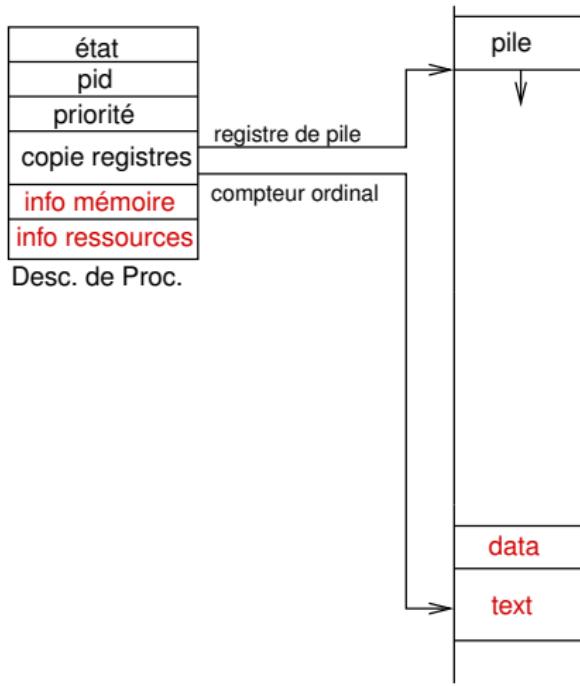
en espace mémoire :

- leur pile (variables locales)

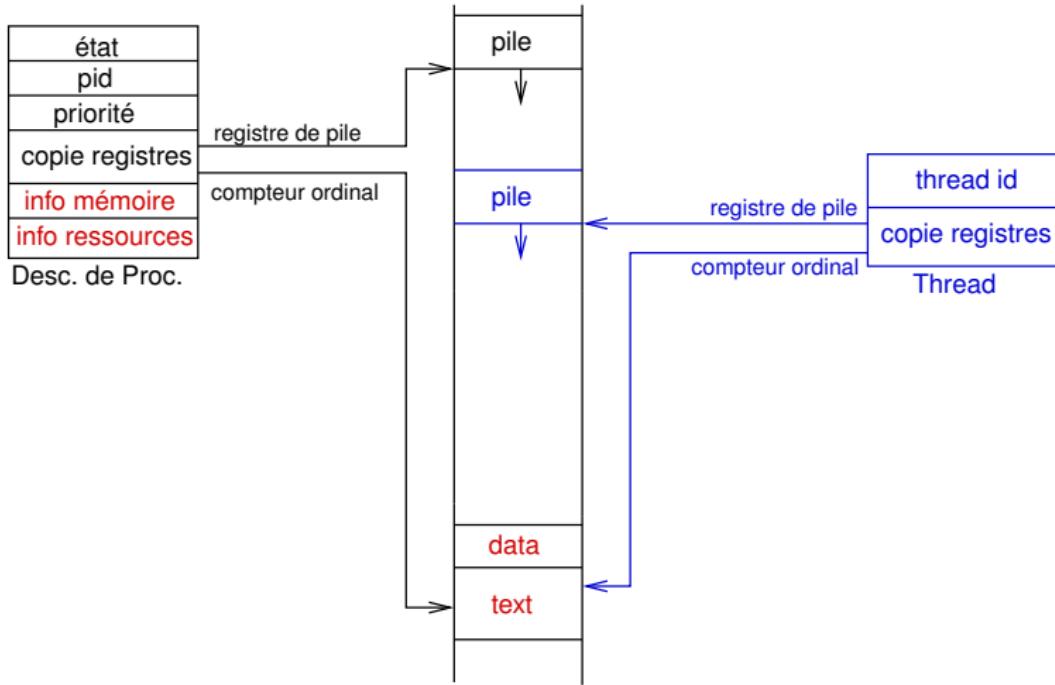
dans le descripteur :

- TID, ...
- la copie des registres  
(PC+SP)

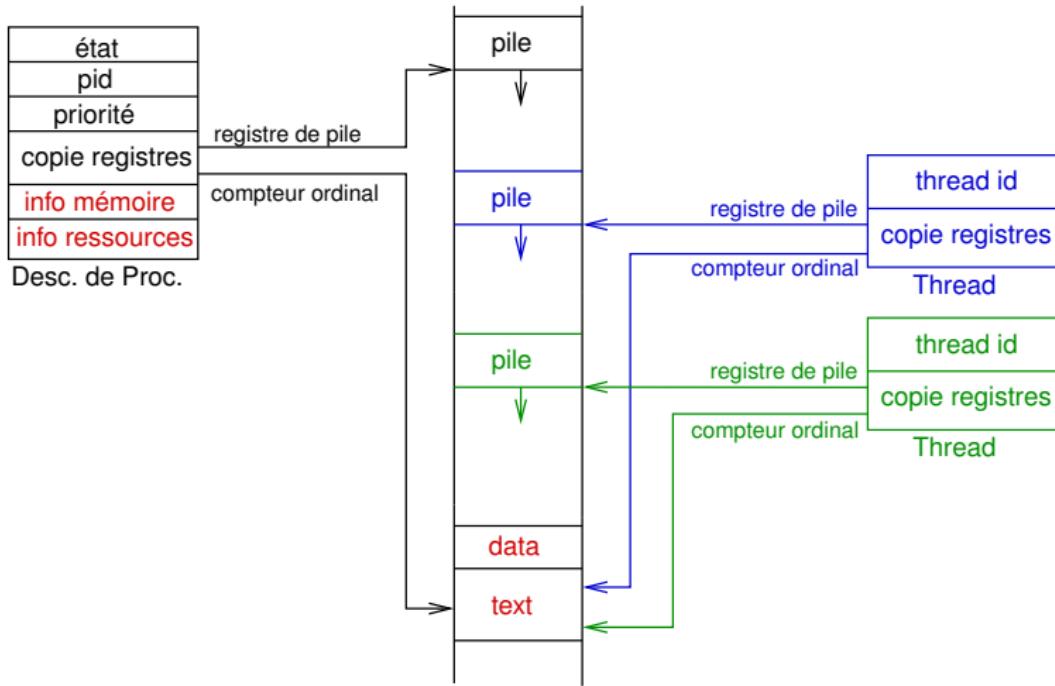
# Exemple : un processus et deux threads



# Exemple : un processus et deux threads



# Exemple : un processus et deux threads



# 1 . Thread

## 1.1. Plusieurs exécutions pour un même espace mémoire

- thread : exécution non-exclusive d'un programme
- espace mémoire partagé... si mémoire synchronisée

## 1.2. C++ **std::thread**

- création **std::thread::thread()**
- destruction après détachement **std::thread::join()**  
**std::thread::detach()**
- création de thread vs. processus

# Espace mémoire partagé entre threads

## Avantage

partager des données entre exécutions

## Besoin de synchronisation de la mémoire entre threads

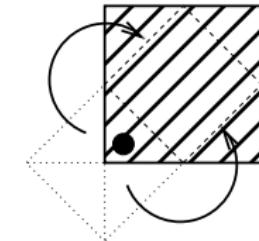
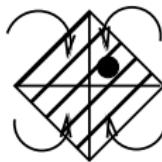
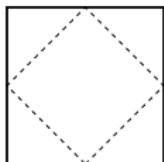
- optimisations : ordre des opérations mémoire, registres...  
⇒ l'effet d'une instruction pas visible par les autres threads
- volatile trop violent : supprime optimisations utiles
- **barrière de mémoire** : instruction qui assure que les opérations mémoires précédentes dans le code sont effectuées

## Synchroniser la mémoire insuffisant pour synchroniser les threads

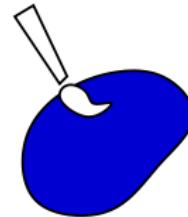
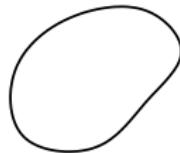
- certains accès simultanés indésirables
- section critique

# Synchronisation de threads / de la mémoire

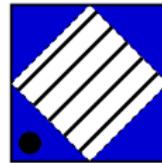
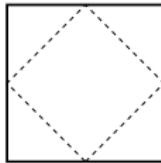
thread 1 :  
plieur  
retourneur



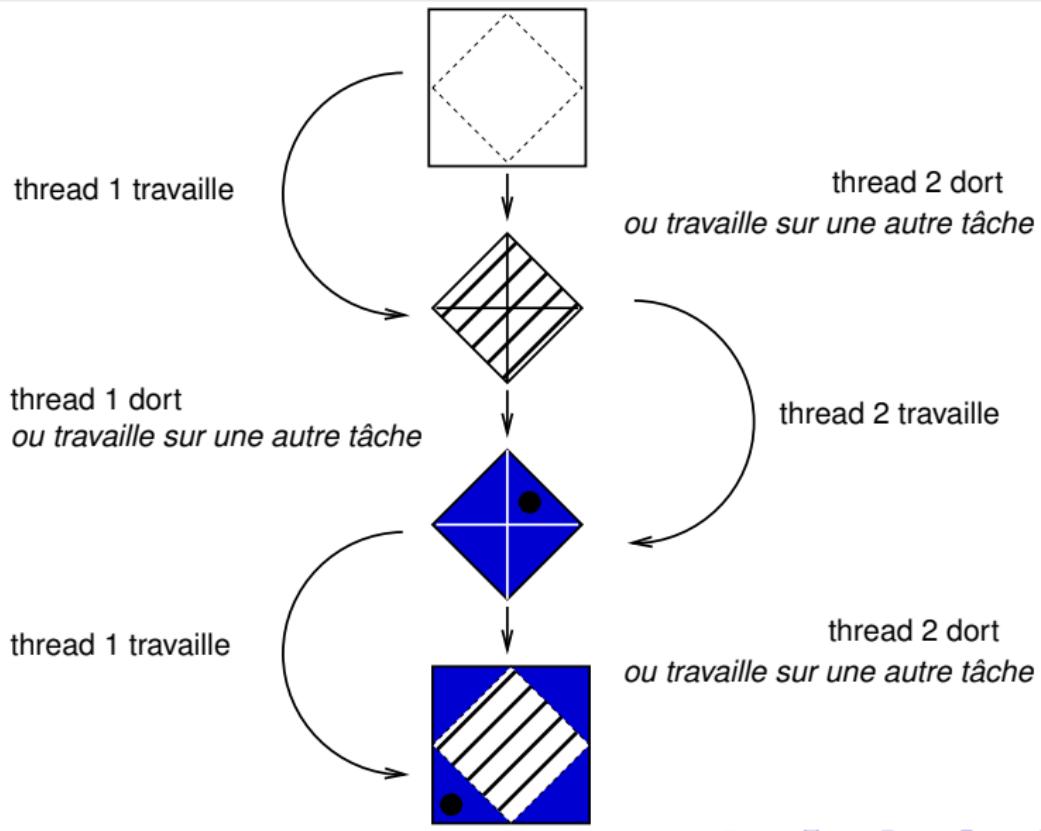
thread 2 :  
peintre



objectif :

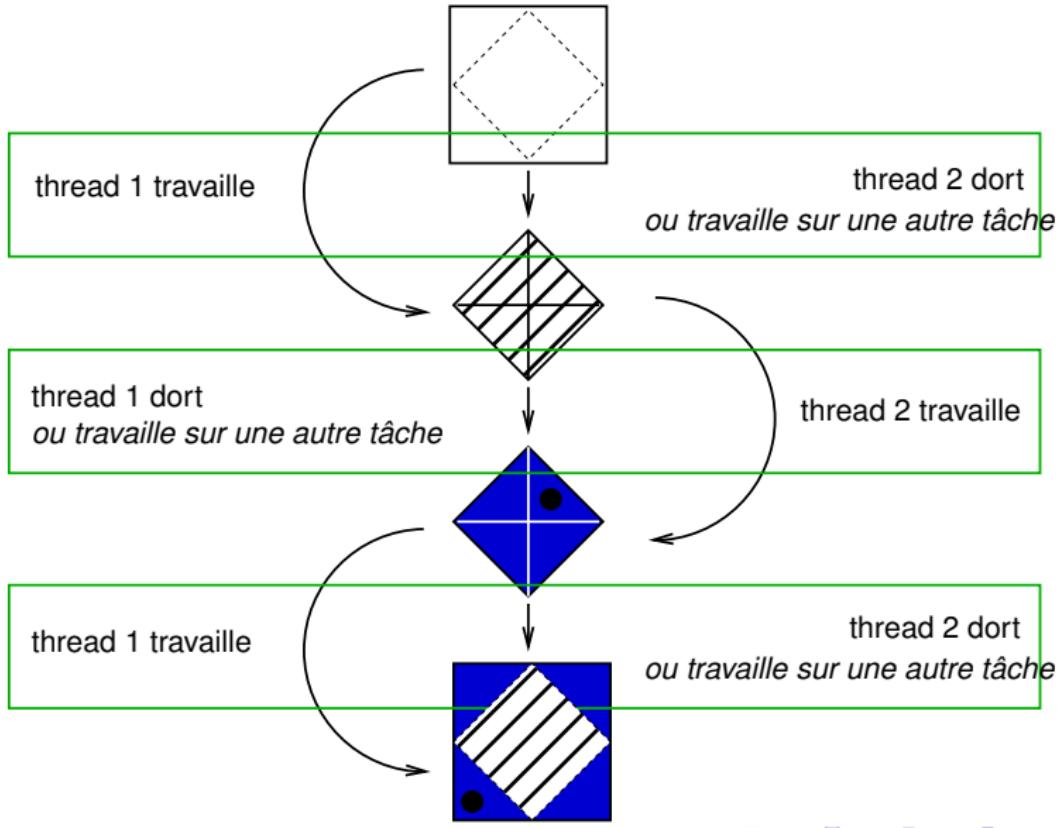


# Synchronisation de threads / de la mémoire

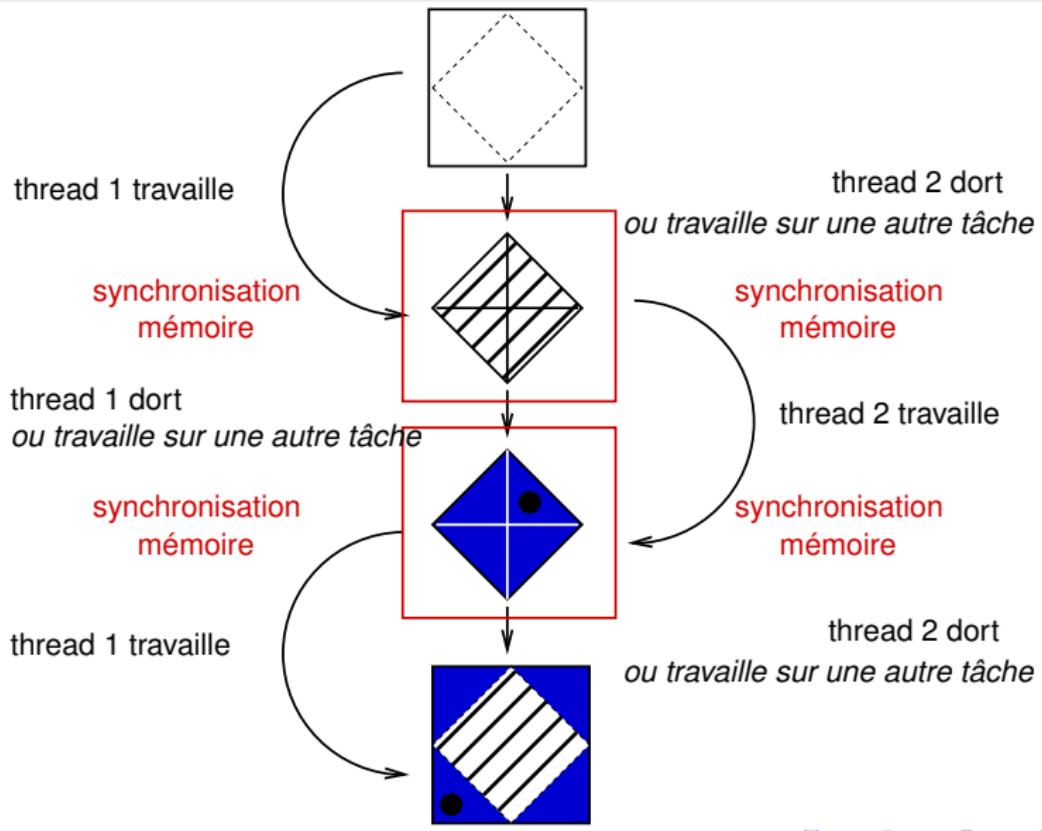


# Synchronisation de threads / de la mémoire

synchronisation threads



# Synchronisation de threads / de la mémoire



# 1 . Thread

## 1.1. Plusieurs exécutions pour un même espace mémoire

- `thread` : exécution non-exclusive d'un programme
- espace mémoire partagé... si mémoire synchronisée

## 1.2. C++ `std::thread`

- création `std::thread::thread()`
- destruction après détachement `std::thread::join()`  
`std::thread::detach()`
- création de thread vs. processus

# Création de threads `std::thread::thread()`

construction par initialisation

```
#include <thread>
...
std::thread t(fn, arg, ...);
```

description

- création de `t` de type `std::thread`, exécutant `fn(arg, ...)`
- `fn` est un objet **appelable** :
  - fonction
  - objet d'une classe avec opérateur `()`
- `fn` et `arg, ...` sont copiés avant d'être invoqués

problème si passage par référence

`void fn(int& arg1) ⇒ std::thread t(fn,arg)` Pb!!  
car fonction s'exécuterait avec une référence vers la copie interne

# Création de threads `std::thread::thread()`

fonction

```
void fn(int& arg1, int arg2)
{ arg1 = arg1 + arg2; }
```

classe avec opérateur ()

```
class Fn {
public :
    Fn(int& arg1_, int arg2_) : arg1(arg1_), arg2(arg2_) {}
    void operator()() { arg1 = arg1 + arg2; }
private :
    int& arg1;
    int arg2; };
```

création de thread avec objet anonyme de la classe Fn

```
int a=0, b=1;
std::thread t( Fn(a,b) );
```

# 1 . Thread

## 1.1. Plusieurs exécutions pour un même espace mémoire

- `thread` : exécution non-exclusive d'un programme
- espace mémoire partagé... si mémoire synchronisée

## 1.2. C++ `std::thread`

- création `std::thread::thread()`
- destruction après détachement `std::thread::join()`  
`std::thread::detach()`
- création de thread vs. processus

# Dissocier thread et exécution pour destr. sans erreur méthodes **join()** et **detach()**

## Destruction d'un objet de type **std::thread**

- si associé à une exécution ⇒ **terminate()**
- avant sa destruction : **dissocier le thread de toute exécution**

### **std::thread::join()**

- bloque le thread exécutant **t.join()** jusqu'à la fin de l'exécution associée à t
- rend t dissocié de toute exécution ⇒ peut être détruit

### **std::thread::detach()**

- rend t dissocié de toute exécution ⇒ peut être détruit
- attention : exécution interrompue si fin du thread principal ⇔ fin du processus

## Exemple complet std::thread (1/2)

```
#include <iostream>
#include <thread>
#include "Fn.hpp" //contient déf. de Fn avec opérateur ()  
using namespace std;  
  
static void fn_sans_ref(int arg1, int arg2) {  
    cout << "somme = " << arg1 + arg2 ;  
}  
  
int main(void){  
    int a=0, b=1;  
    thread t1(fn_sans_ref, a, b);  
    thread t2( Fn(a, b) );  
    t1.join(); t2.join();  
    cout << " et a = " << a << endl;  
    return 0;  
}
```

```
$ ./mon_programme  
somme = 1 et a = 1
```

## Exemple complet std::thread (2/2)

```
#include <iostream>
#include <vector>
#include <thread>
#include "Fn.hpp" //contient déf. de Fn avec opérateur ()
```

```
using namespace std;
```

```
static void fn_sans_ref(int arg1, int arg2) {
    cout << "somme = " << arg1 + arg2 ; }
```

```
int main(void){
    int a=0, b=1;
    vector<thread> vt;
    vt.push_back(thread(fn_sans_ref, a, b));
    vt.push_back(thread( Fn(a, b) ));
    vt.at(0).join(); vt.at(1).join();
    cout << " et a = " << a << endl;
    return 0; }
```

```
$ ./mon_programme
somme = 1 et a = 1
```



# 1 . Thread

## 1.1. Plusieurs exécutions pour un même espace mémoire

- `thread` : exécution non-exclusive d'un programme
- espace mémoire partagé... si mémoire synchronisée

## 1.2. C++ `std::thread`

- création `std::thread::thread()`
- destruction après détachement `std::thread::join()`  
`std::thread::detach()`
- **création de thread vs. processus**

# Création de thread versus processus

Procédure exécutée par le thread / processus fils

```
void fn (int arg) { cout << a << endl; }
```

## Thread

```
int main(void) {
    int a = 0;
    thread t(fn, a);
    ... le thread principal
    t.join();
    return 0;
}
```

## Processus

```
int main(void) {
    int a = 0;
    pid_t res=fork();
    if (res==0)
        fn(a);
    else
        { ... le père
        waitpid(res,NULL,0);
        }
    return 0;
}
```

# Organisation du cours

## Programmation Concurrente

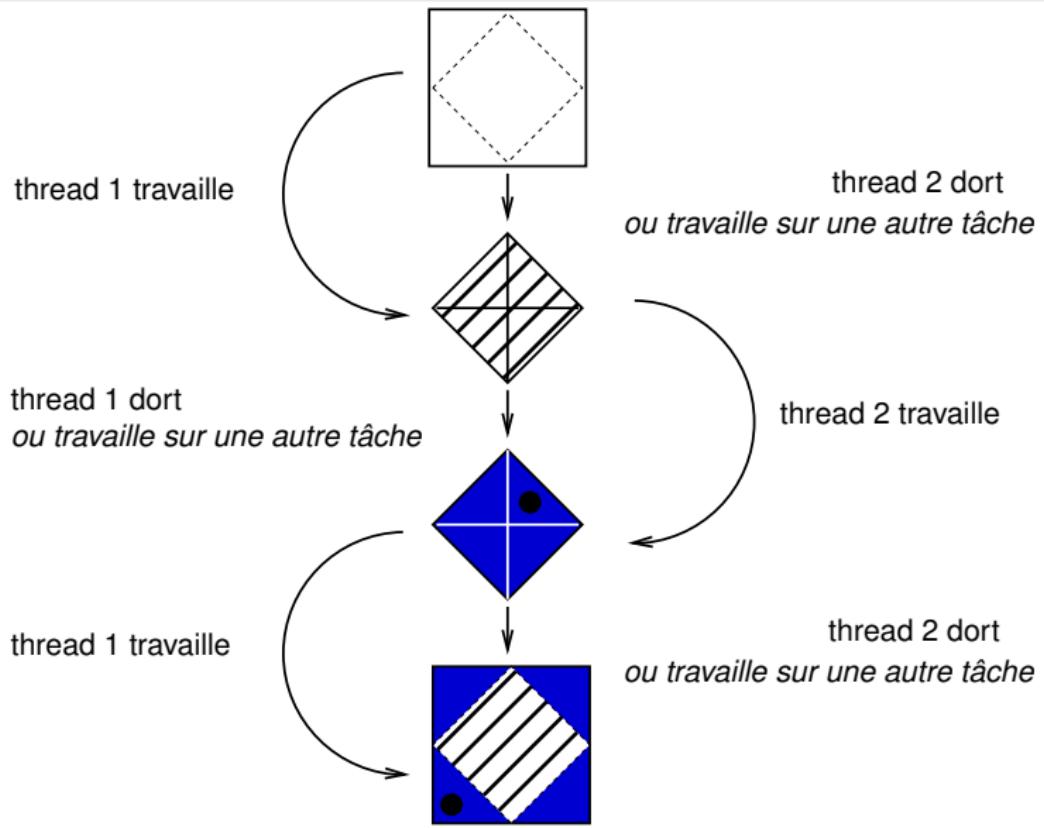
1. Thread
2. Section critique et mutex
3. Moniteur
4. Moniteur (avancé)

# Organisation du cours

## Programmation Concurrente

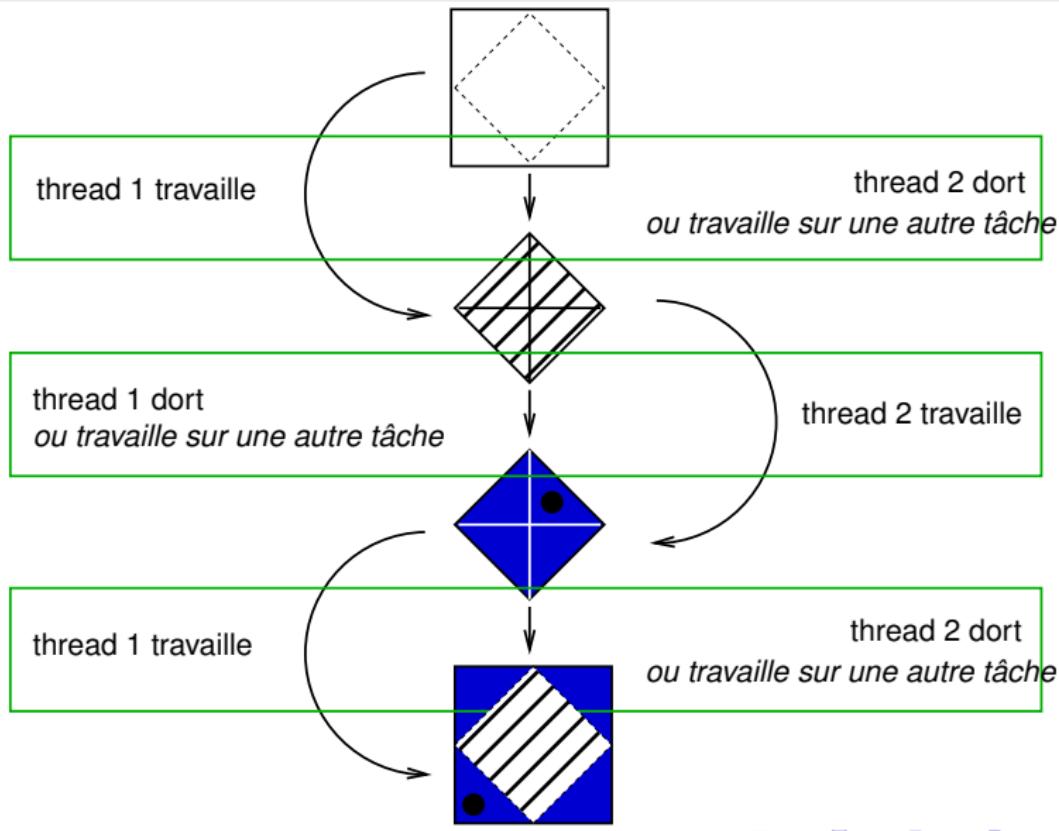
1. Thread
2. Section critique et mutex
3. Moniteur
4. Moniteur (avancé)

# Synchronisation de threads / de la mémoire



# Synchronisation de threads / de la mémoire

synchronisation threads



# 2 . Section critique et mutex

## 2.1. Section critique

- Exclusion Mutuelle
- Pas d'interblocage
- Pas de famine

## 2.2. Verrou

- Une des trois réalisations possibles de SC
- Principe
- Risque d'interblocage

## 2.3. Verrous C++ <**mutex**>

- classe de base **std::mutex**
- autres classes **std::lock\_guard** et **std::unique\_lock**

# 2 . Section critique et mutex

## 2.1. Section critique

- Exclusion Mutuelle
- Pas d'interblocage
- Pas de famine

## 2.2. Verrou

- Une des trois réalisations possibles de SC
- Principe
- Risque d'interblocage

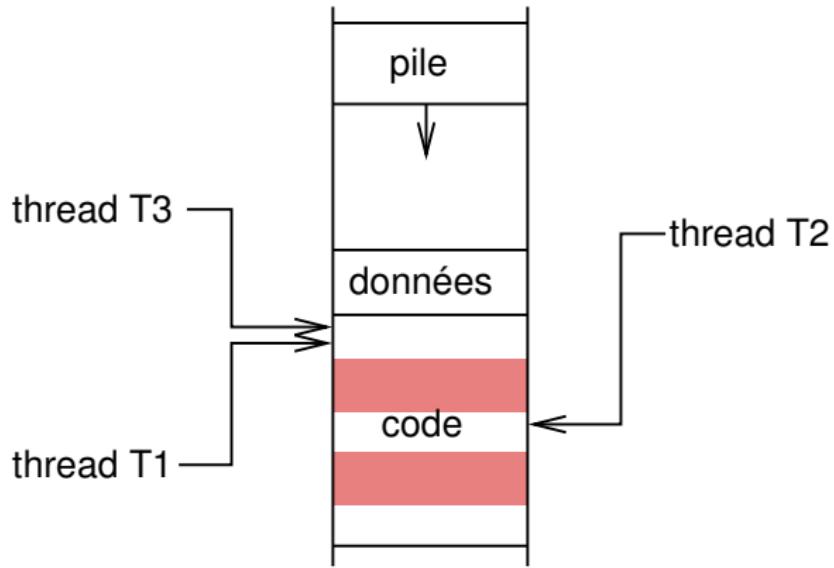
## 2.3. Verrous C++ <mutex>

- classe de base **std::mutex**
- autres classes **std::lock\_guard** et **std::unique\_lock**

# Section critique : définitions

## Section critique

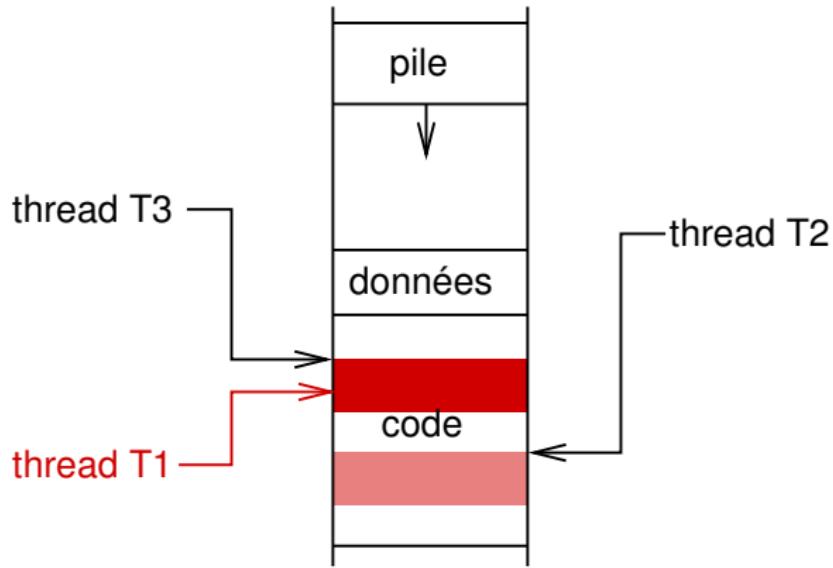
morceau de code manipulant une ressource critique (non partageable)



# Section critique : définitions

## Section critique

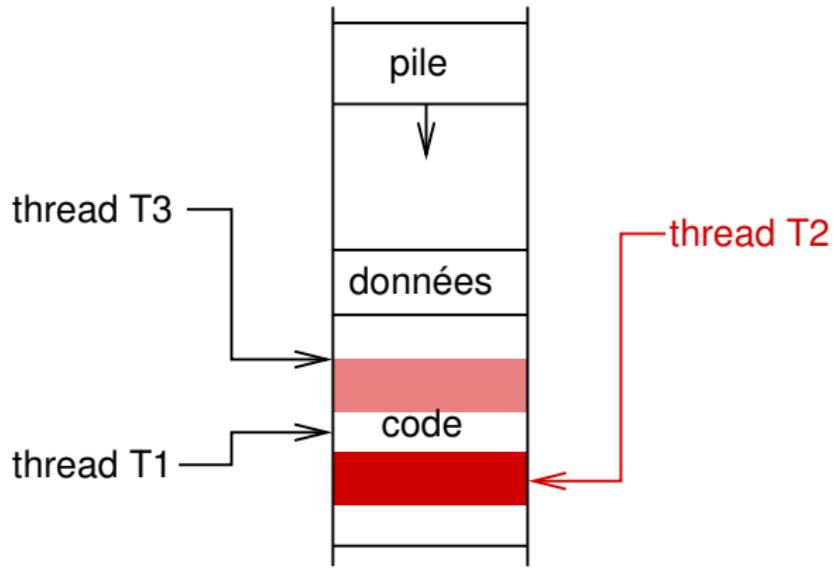
morceau de code manipulant une ressource critique (non partageable)



# Section critique : définitions

## Section critique

morceau de code manipulant une ressource critique (non partageable)



# Section critique : propriétés

## Exclusion mutuelle

**un seul** thread accède à une ressource non partageable.

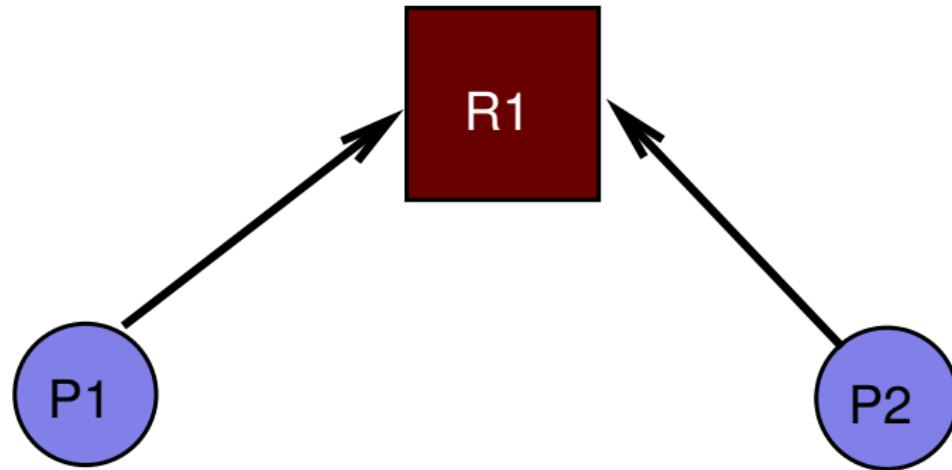
## Empêcher les Interblocages

**un groupe de threads se bloquent mutuellement.**

## Empêcher la Famine

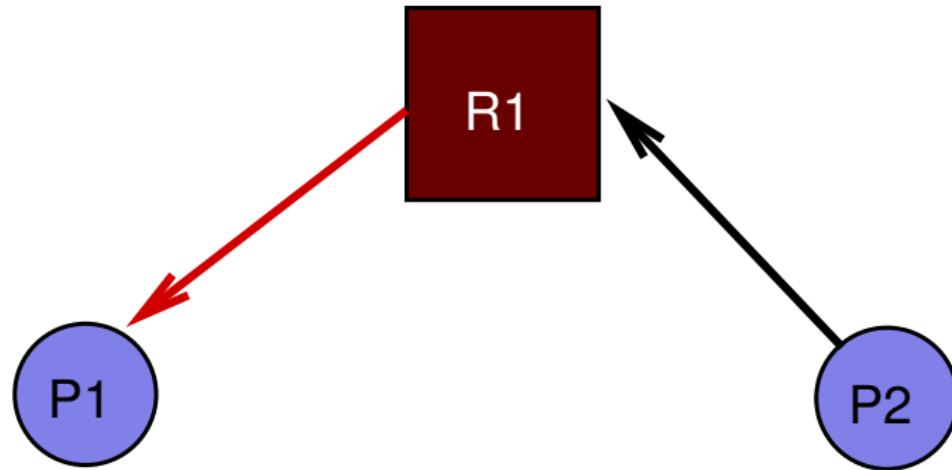
**un thread ne parvient pas** à accéder à une ressource.

# Exclusion mutuelle



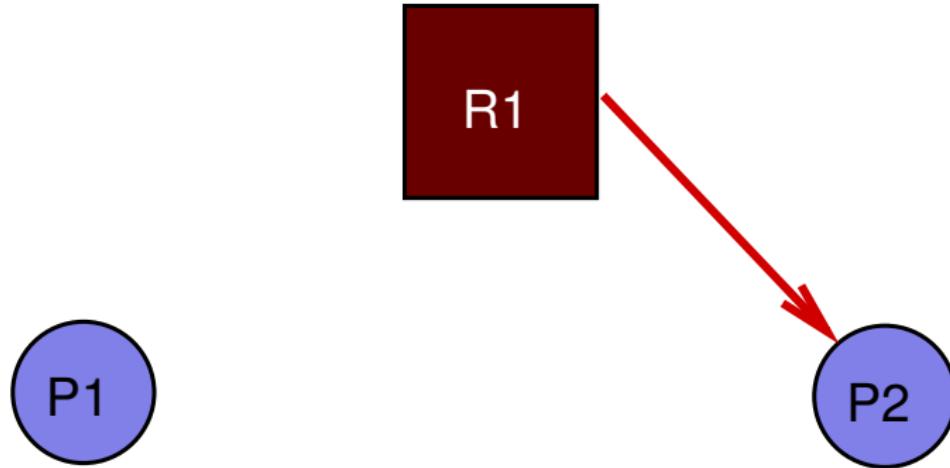
Graphe d'allocation des ressources

# Exclusion mutuelle



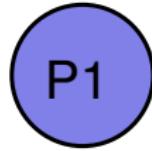
Graphe d'allocation des ressources

# Exclusion mutuelle



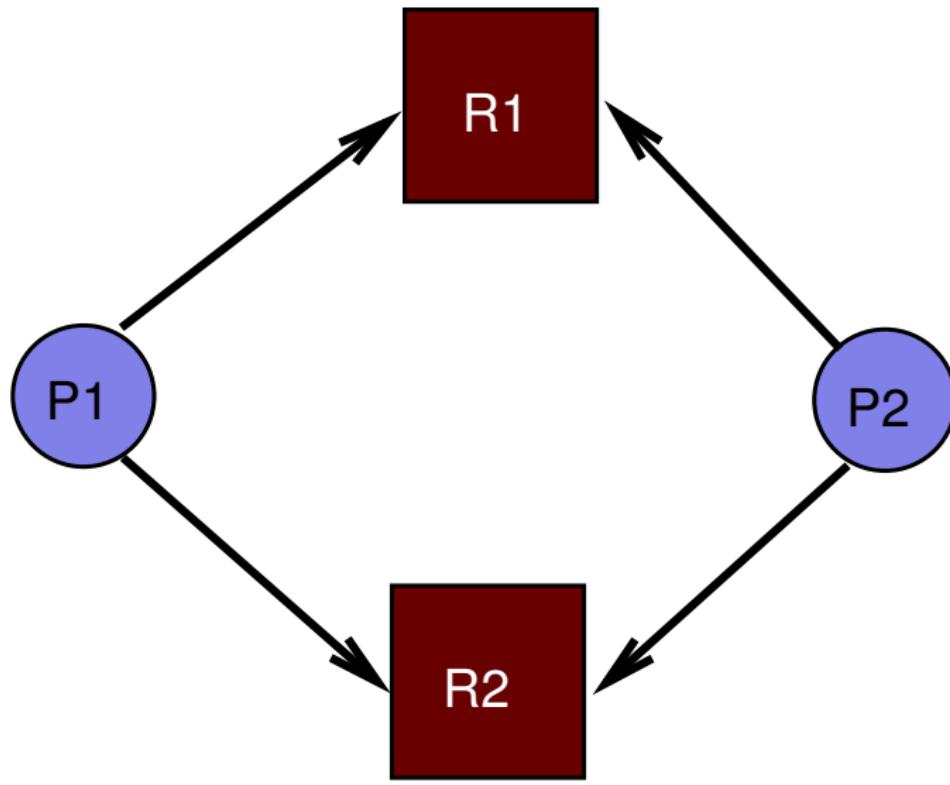
Graphe d'allocation des ressources

# Exclusion mutuelle

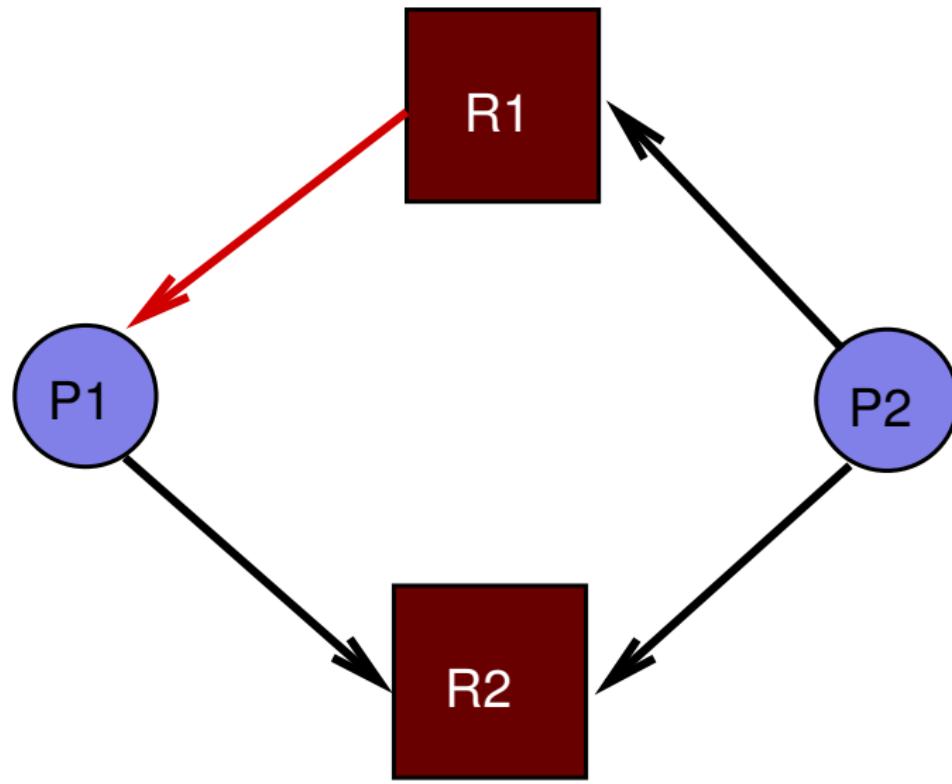


Graphe d'allocation des ressources

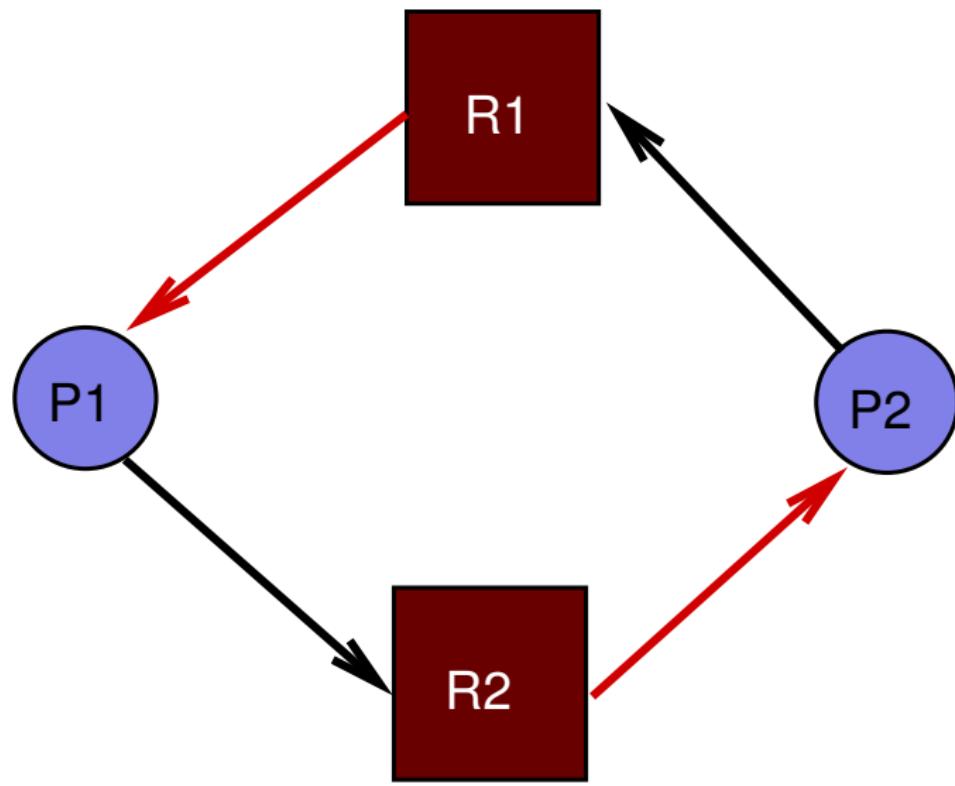
# Interblocage



# Interblocage



# Interblocage



# Interblocage

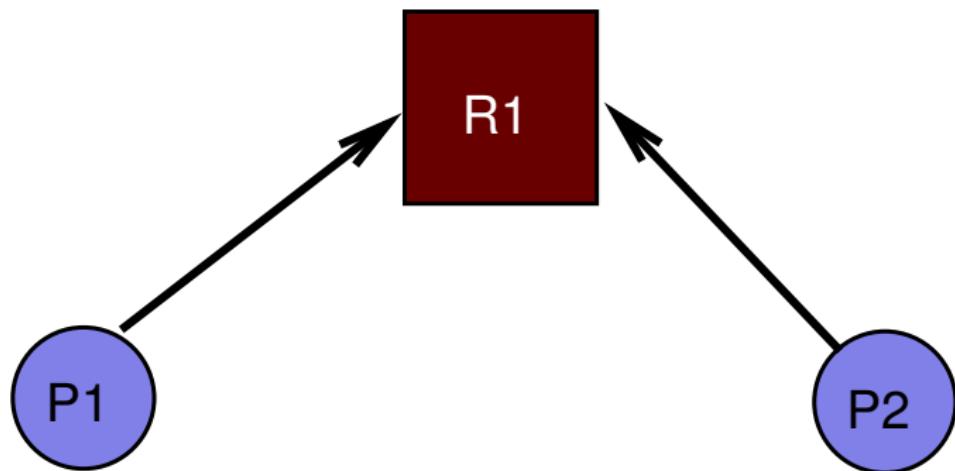
## Quatre conditions nécessaires

- ① **Exclusion mutuelle** : ressources privées
- ② **Toujours plus** : demandes successives de ressources différentes
- ③ **Pas de réquisition** : ressources allouées jusqu'à libération
- ④ **Attente cyclique** : ressources non-ordonnées

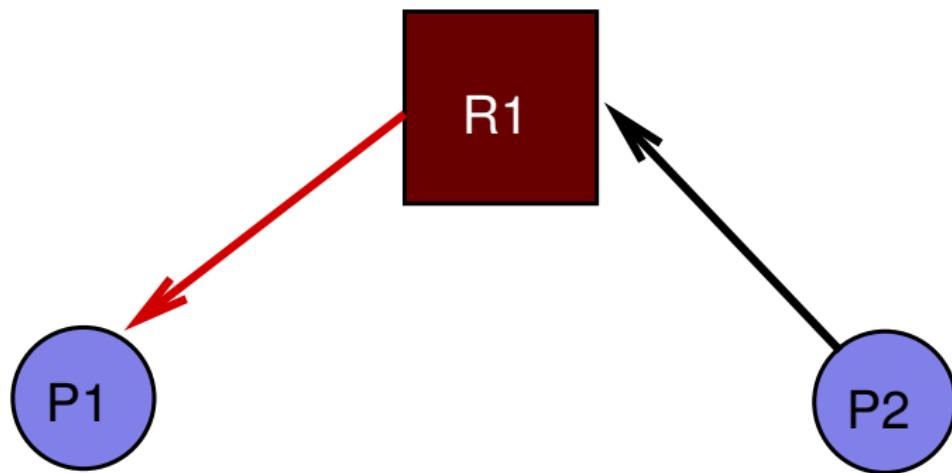
## Section critique en pratique

invalide 2ème ou 4ème condition

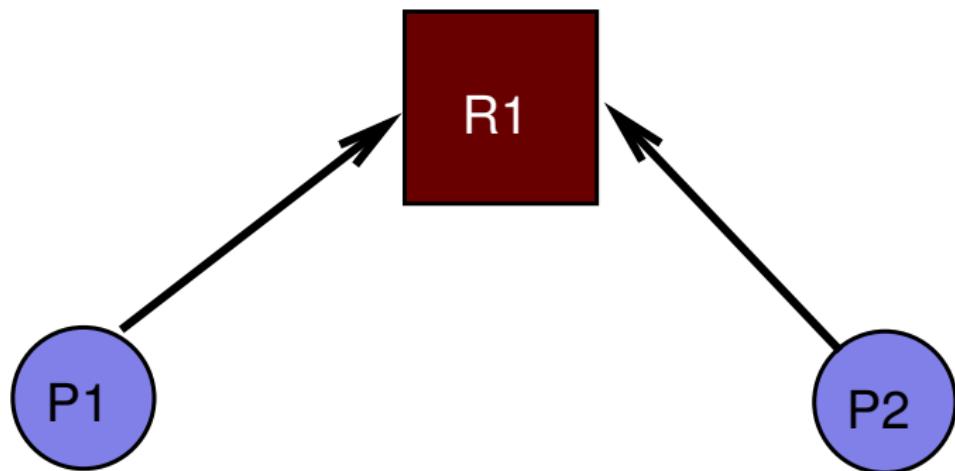
# Famine



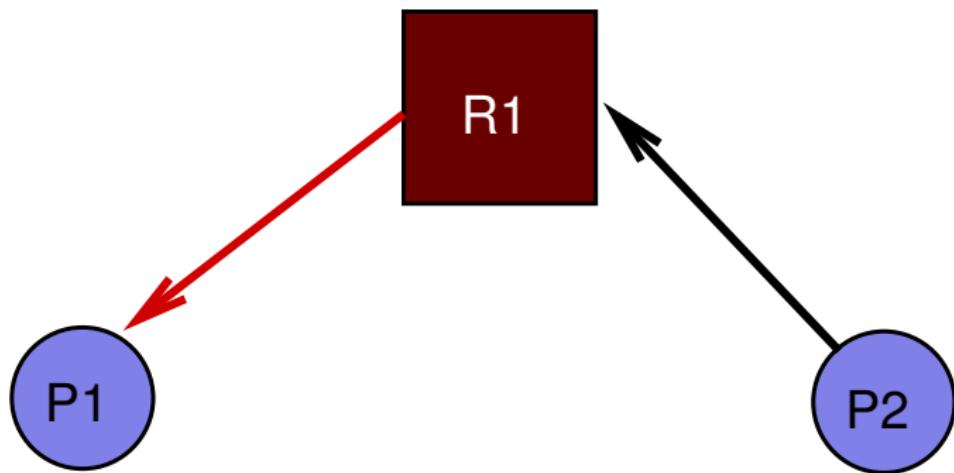
# Famine



# Famine



# Famine



# Section Critique

Exclusion mutuelle

**un seul** thread accède à une ressource non partageable.

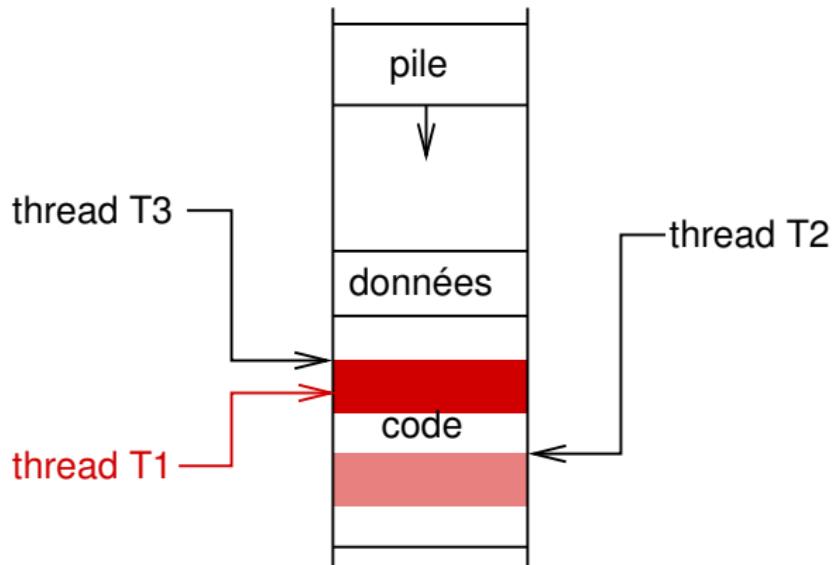
Pas d'interblocage

**un groupe de threads ne se bloquent jamais mutuellement.**

Pas de famine

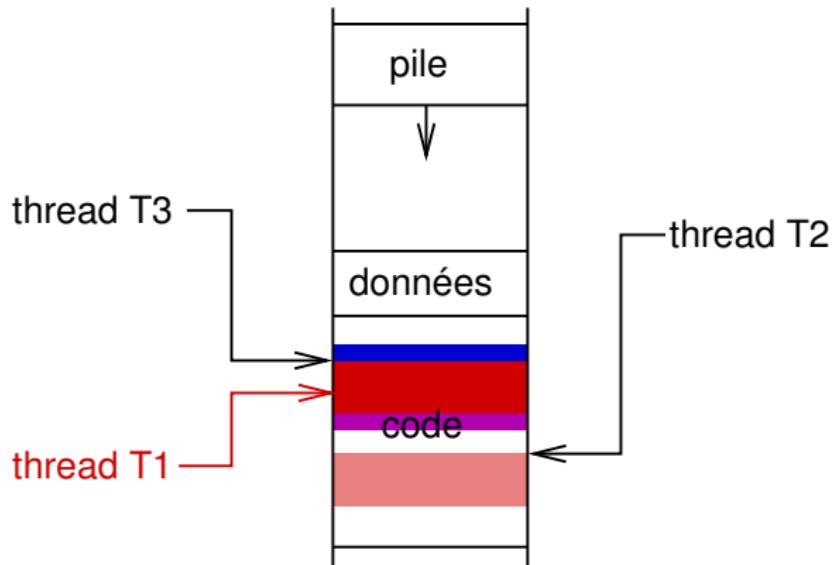
**tout thread parvient** à accéder à une ressource.

# Réalisation d'une section critique



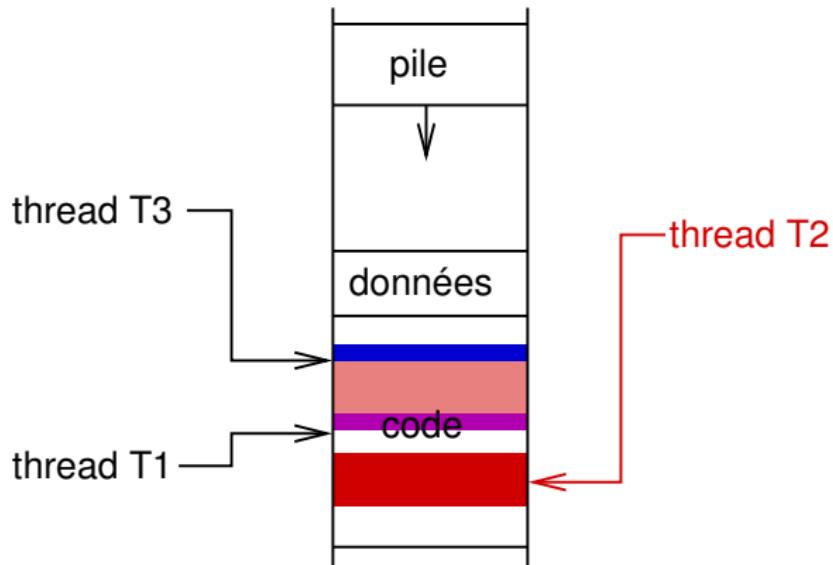
- section d'entrée
- section de sortie

# Réalisation d'une section critique



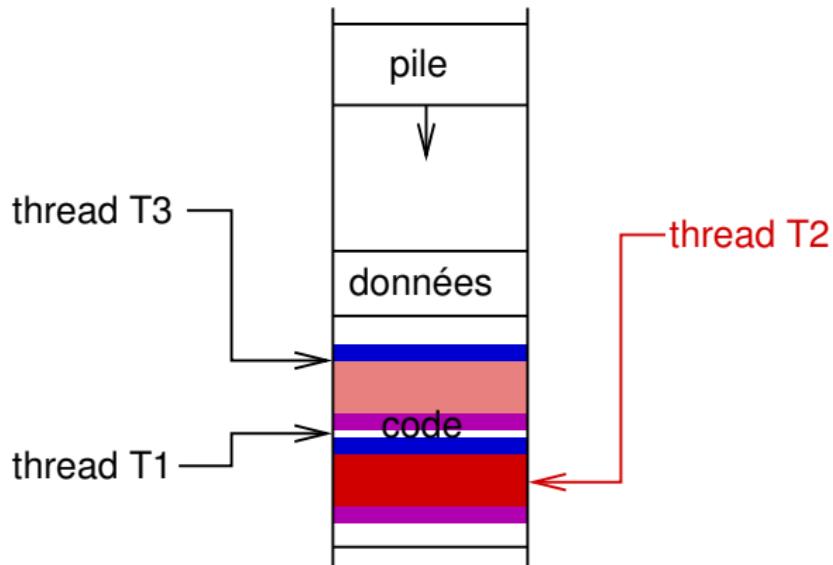
- section d'entrée
- section de sortie

# Réalisation d'une section critique



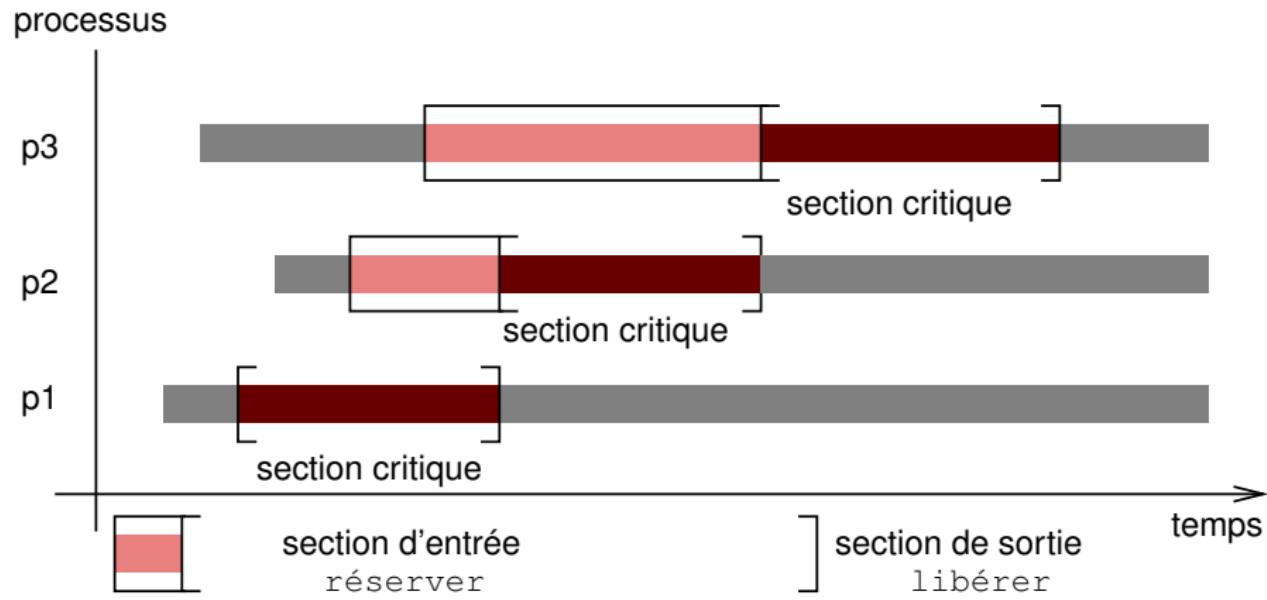
- section d'entrée
- section de sortie

# Réalisation d'une section critique



- section d'entrée
- section de sortie

# Réalisation d'une section critique



Chronogramme d'exécution concurrente en exclusion mutuelle

## 2 . Section critique et mutex

### 2.1. Section critique

- Exclusion Mutuelle
- Pas d'interblocage
- Pas de famine

### 2.2. Verrou

- Une des trois réalisations possibles de SC
- Principe
- Risque d'interblocage

### 2.3. Verrous C++ <**mutex**>

- classe de base **std::mutex**
- autres classes **std::lock\_guard** et **std::unique\_lock**

# Deux autres réalisations d'une section critique

## Masquage des interruptions

### **section d'entrée / de sortie :**

interdire/autoriser commutation  
de processus

**inconvénient** : exclut **tous** les processus  
**d'un** processeur

Pour :

- SC courtes (sans E/S)
- en mode noyau
- avec mono-processeur

## Attente active (*spinlock*)

### **section d'entrée :**

boucler tq cond. d'attente est vraie  
rendre la condition vraie

### **section de sortie :**

rendre la condition fausse

**inconvénient** : tps CPU perdu

Pour :

- SC courtes (sans E/S)
- plutôt en mode noyau
- plutôt multi-processeur

# 2 . Section critique et mutex

## 2.1. Section critique

- Exclusion Mutuelle
- Pas d'interblocage
- Pas de famine

## 2.2. Verrou

- Une des trois réalisations possibles de SC
- **Principe**
- Risque d'interblocage

## 2.3. Verrous C++ <**mutex**>

- classe de base **std::mutex**
- autres classes **std::lock\_guard** et **std::unique\_lock**

# Verrou - Principe

Définition : Dijkstra 1965 - sémaphore binaire

objet partageable entre threads comportant :

- un booléen : loquet ouvert/fermé
- une liste FIFO de threads demandant à entrer en SC

Procédures associées : verrouiller() déverrouiller()

- seuls accès aux composants du verrou
- exécution indivisible (via masquage d'interruption ou spinlock)
- **verrouiller()** : **si** loquet==fermé  
                 **alors** bloquer le thread en queue de FIFO  
                 **finsi**  
                 loquet=fermé
- **déverrouiller()** : loquet=ouvert  
                 **si** liste FIFO non vide  
                 **alors** libérer le thread en tête de FIFO  
                 **finsi**

# Verrou - Principe

## Application à la réalisation d'une SC

- **section d'entrée** : verrouiller(verrou)
- **section de sortie** : déverrouiller(verrou)

## Absence de Famine

- garantie par la liste FIFO

## Absence d'interblocage

- non garantie !
- **responsabilité du programmeur**

# Verrou

## Inconvénients

- file de processus/threads + exécution procédures indivisible : coûteux

## Quand l'utiliser ?

- pour des SC longues
- en mode noyau ou utilisateur
- sur mono ou multi-processeur

# 2 . Section critique et mutex

## 2.1. Section critique

- Exclusion Mutuelle
- Pas d'interblocage
- Pas de famine

## 2.2. Verrou

- Une des trois réalisations possibles de SC
- Principe
- **Risque d'interblocage**

## 2.3. Verrous C++ <**mutex**>

- classe de base **std::mutex**
- autres classes **std::lock\_guard** et **std::unique\_lock**

# Attente active ou verrou

Interblocage si

- ① **Exclusion mutuelle** : garantie par les verrous
- ② **Toujours plus** : demandes successives de ressources différentes
- ③ **Pas de réquisition** : garantie par les verrous
- ④ **Attente cyclique** : ressources non-ordonnées

L'interblocage est évité si

- pas de *toujours plus* (**SC non imbriquées**)
- pas attente cyclique (**réservations dans même ordre**)

# Risque d'interblocage : SC imbriquées

Thread 1

```

...
réserver(r1);
manip. ress. 1
réserver(r2);
manip. ress. 2
libérer(r2);
manip. ress. 1
libérer(r1);
...

```

processus

p2



p1



Thread 2

```

...
réserver(r2);
manip. ress. 2
réserver(r1);
manip. ress. 1
libérer(r1);
manip. ress. 2
libérer(r2);
...

```

temps

# Risque d'interblocage : SC imbriquées

Thread 1

```
...
réserver(r1);
manip. ress. 1
réserver(r2);
manip. ress. 2
libérer(r2);
manip. ress. 1
libérer(r1);
...
```

processus

p2



p1



Thread 2

```
...
réserver(r2);
manip. ress. 2
réserver(r1);
manip. ress. 1
libérer(r1);
manip. ress. 2
libérer(r2);
...
```

...

temps

# Risque d'interblocage : SC imbriquées

Thread 1

```

    ...
réserver(r1);
    manip. ress. 1
réserver(r2);
    manip. ress. 2
libérer(r2);
    manip. ress. 1
libérer(r1);
    ...

```

processus



Thread 2

```

    ...
réserver(r2);
    manip. ress. 2
réserver(r1);
    manip. ress. 1
libérer(r1);
    manip. ress. 2
libérer(r2);
    ...

```

...

# Risque d'interblocage : SC imbriquées

Thread 1

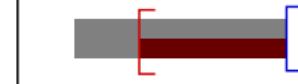
```

    ...
    résérvier(r1);
    manip. ress. 1
    résérvier(r2);
    manip. ress. 2
    libérer(r2);
    manip. ress. 1
    libérer(r1);
    ...
  
```

processus



p1



Thread 2

```

    ...
    résérvier(r2);
    manip. ress. 2
    résérvier(r1);
    manip. ress. 1
    libérer(r1);
    manip. ress. 2
    libérer(r2);
    ...
  
```

...

temps

# Risque d'interblocage : SC imbriquées

Thread 1

```

    ...
    résérvier(r1);
    manip. ress. 1
    résérvier(r2);
    manip. ress. 2
    libérer(r2);
    manip. ress. 1
    libérer(r1);
    ...

```

processus



p1

Thread 2

```

    ...
    résérvier(r2);
    manip. ress. 2
    résérvier(r1);
    manip. ress. 1
    libérer(r1);
    manip. ress. 2
    libérer(r2);
    ...

```

...

temps

# Risque d'interblocage : SC imbriquées

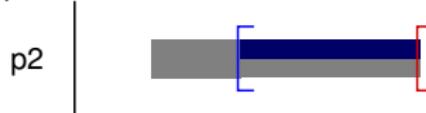
Thread 1

```

    ...
réserver(r1);
manip. ress. 1
réserver(r2);
manip. ress. 2
libérer(r2);
manip. ress. 1
libérer(r1);
    ...

```

processus



Thread 2

```

    ...
réserver(r2);
manip. ress. 2
réserver(r1);
manip. ress. 1
libérer(r1);
manip. ress. 2
libérer(r2);
    ...

```

interblocage !



# Eviter interblocage : SC non imbriquées

Thread 1

```

réserver(r1);
manip. ress. 1
libérer(r1);
réserver(r2);
manip. ress. 2
libérer(r2);
réserver(r1);
manip. ress. 1
libérer(r1);

```

processus

p2



p1



Thread 2

```

réserver(r2);
manip. ress. 2
libérer(r2);
réserver(r1);
manip. ress. 1
libérer(r1);
réserver(r2);
manip. ress. 2
libérer(r2);

```

→  
temps

# Eviter interblocage : SC non imbriquées

Thread 1

```

réserver(r1);
manip. ress. 1
libérer(r1);
réserver(r2);
manip. ress. 2
libérer(r2);
réserver(r1);
manip. ress. 1
libérer(r1);

```

processus

p2



p1

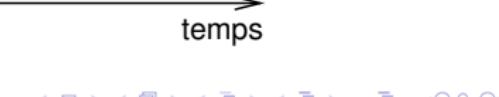


Thread 2

```

réserver(r2);
manip. ress. 2
libérer(r2);
réserver(r1);
manip. ress. 1
libérer(r1);
réserver(r2);
manip. ress. 2
libérer(r2);

```

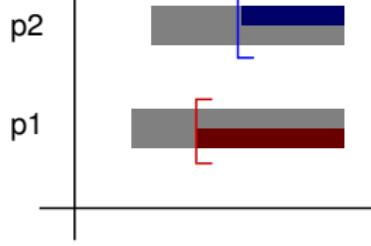


# Eviter interblocage : SC non imbriquées

Thread 1

```
réserver(r1);
manip. ress. 1
libérer(r1);
réserver(r2);
manip. ress. 2
libérer(r2);
réserver(r1);
manip. ress. 1
libérer(r1);
```

processus



Thread 2

```
réserver(r2);
manip. ress. 2
libérer(r2);
réserver(r1);
manip. ress. 1
libérer(r1);
réserver(r2);
manip. ress. 2
libérer(r2);
```



# Eviter interblocage : SC non imbriquées

Thread 1

```

réserver(r1);
  manip. ress. 1
libérer(r1);
réserver(r2);
  manip. ress. 2
libérer(r2);
réserver(r1);
  manip. ress. 1
libérer(r1);
processus

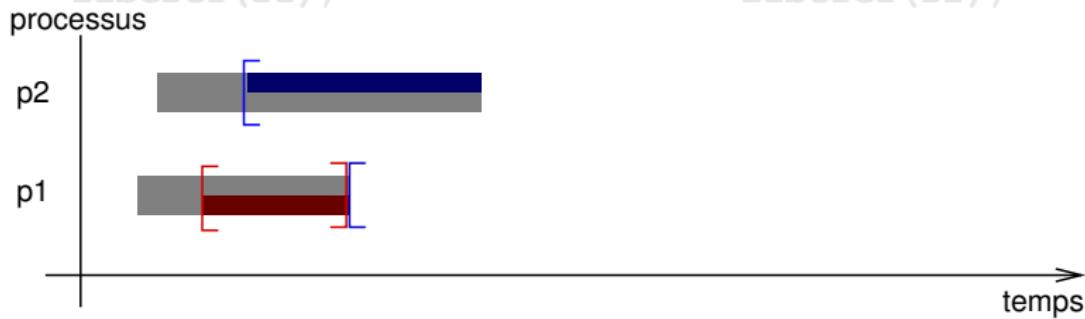
```

Thread 2

```

réserver(r2);
  manip. ress. 2
libérer(r2);
réserver(r1);
  manip. ress. 1
libérer(r1);
réserver(r2);
  manip. ress. 2
libérer(r2);

```



# Eviter interblocage : SC non imbriquées

Thread 1

```

réserver(r1);
  manip. ress. 1
libérer(r1);
réserver(r2);
  manip. ress. 2
libérer(r2);
réserver(r1);
  manip. ress. 1
libérer(r1);
processus

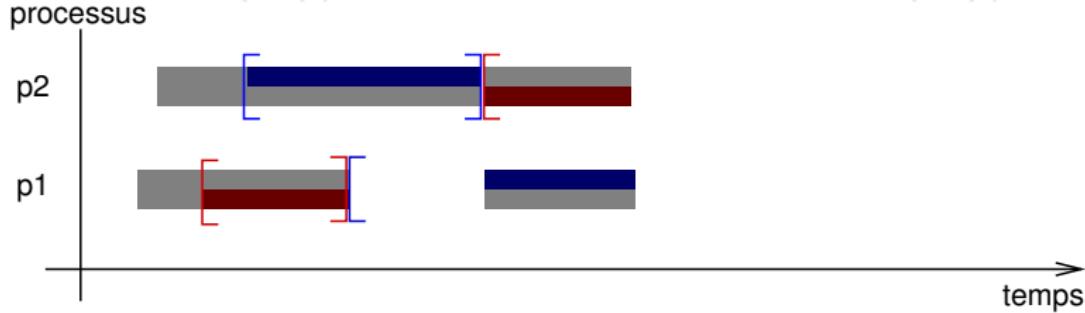
```

Thread 2

```

réserver(r2);
  manip. ress. 2
libérer(r2);
réserver(r1);
  manip. ress. 1
libérer(r1);
réserver(r2);
  manip. ress. 2
libérer(r2);

```



# Eviter interblocage : SC non imbriquées

Thread 1

```

réserver(r1);
  manip. ress. 1
libérer(r1);
réserver(r2);
  manip. ress. 2
libérer(r2);
réserver(r1);
  manip. ress. 1
libérer(r1);
processus

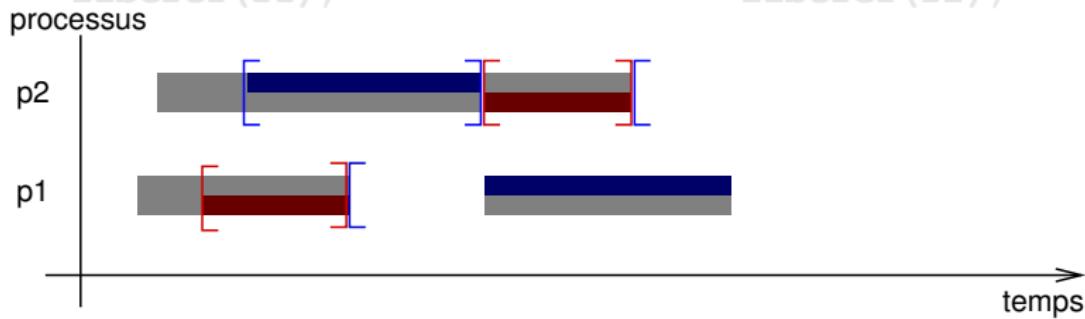
```

Thread 2

```

réserver(r2);
  manip. ress. 2
libérer(r2);
réserver(r1);
  manip. ress. 1
libérer(r1);
réserver(r2);
  manip. ress. 2
libérer(r2);

```



# Eviter interblocage : SC non imbriquées

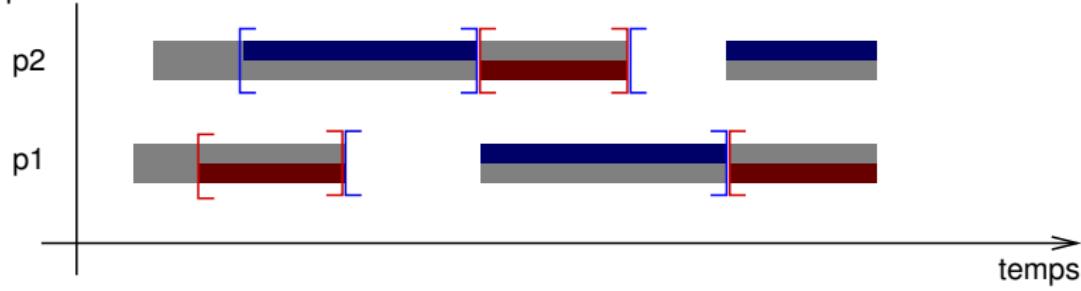
Thread 1

```

réserver(r1);
  manip. ress. 1
libérer(r1);
réserver(r2);
  manip. ress. 2
libérer(r2);
réserver(r1);
  manip. ress. 1
libérer(r1);

```

processus



Thread 2

```

réserver(r2);
  manip. ress. 2
libérer(r2);
réserver(r1);
  manip. ress. 1
libérer(r1);
réserver(r2);
  manip. ress. 2
libérer(r2);

```

# Eviter interblocage : SC non imbriquées

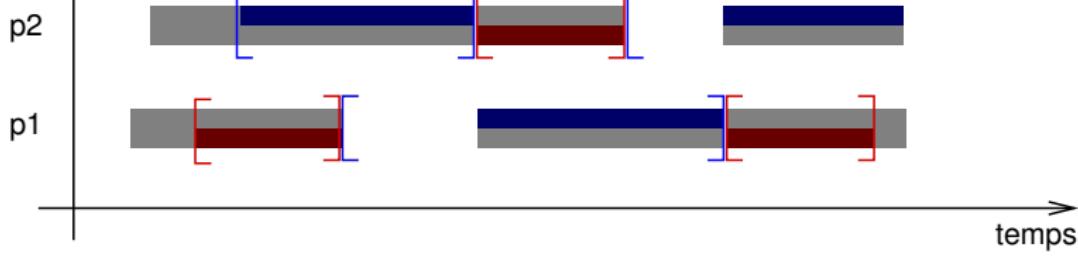
Thread 1

```

réserver(r1);
  manip. ress. 1
libérer(r1);
réserver(r2);
  manip. ress. 2
libérer(r2);
réserver(r1);
  manip. ress. 1
libérer(r1);

```

processus



Thread 2

```

réserver(r2);
  manip. ress. 2
libérer(r2);
réserver(r1);
  manip. ress. 1
libérer(r1);
réserver(r2);
  manip. ress. 2
libérer(r2);

```

# Eviter interblocage : SC non imbriquées

Thread 1

```

réserver(r1);
  manip. ress. 1
libérer(r1);
réserver(r2);
  manip. ress. 2
libérer(r2);
réserver(r1);
  manip. ress. 1
libérer(r1);
processus

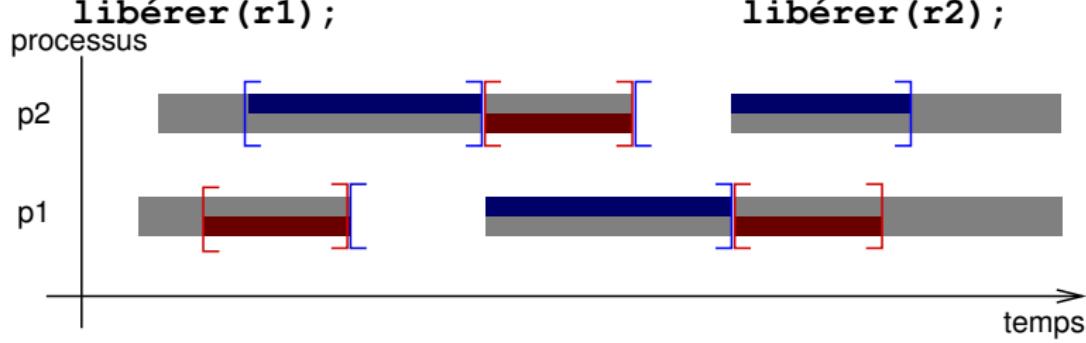
```

Thread 2

```

réserver(r2);
  manip. ress. 2
libérer(r2);
réserver(r1);
  manip. ress. 1
libérer(r1);
réserver(r2);
  manip. ress. 2
libérer(r2);

```



# Eviter interblocage : réservations dans même ordre

Thread 1

```
...
réserver(r1);
manip. ress. 1
réserver(r2);
manip. ress. 2 et 1
libérer(r2);
manip. ress. 1
libérer(r1);
...
```

thread



Thread 2

```
...
réserver(r1);
réserver(r2);
manip. ress. 2
manip. ress. 1 et 2
libérer(r1);
manip. ress. 2
libérer(r2);
...
```

...



# Eviter interblocage : réservations dans même ordre

Thread 1

```

    ...
réserver(r1);
manip. ress. 1
réserver(r2);
manip. ress. 2 et 1
libérer(r2);
manip. ress. 1
libérer(r1);
    ...

```

thread

t2



t1



Thread 2

```

    ...
réserver(r1);
réserver(r2);
manip. ress. 2
manip. ress. 1 et 2
libérer(r1);
manip. ress. 2
libérer(r2);
    ...

```

→  
temps

# Eviter interblocage : réservations dans même ordre

Thread 1

```

    ...
réserver(r1);
manip. ress. 1
réserver(r2);
manip. ress. 2 et 1
libérer(r2);
manip. ress. 1
libérer(r1);
    ...

```

thread



Thread 2

```

    ...
réserver(r1);
réserver(r2);
manip. ress. 2
manip. ress. 1 et 2
libérer(r1);
manip. ress. 2
libérer(r2);
    ...

```

...



# Eviter interblocage : réservations dans même ordre

Thread 1

```

    ...
réserver(r1);
manip. ress. 1
réserver(r2);
manip. ress. 2 et 1
libérer(r2);
manip. ress. 1
libérer(r1);
    ...

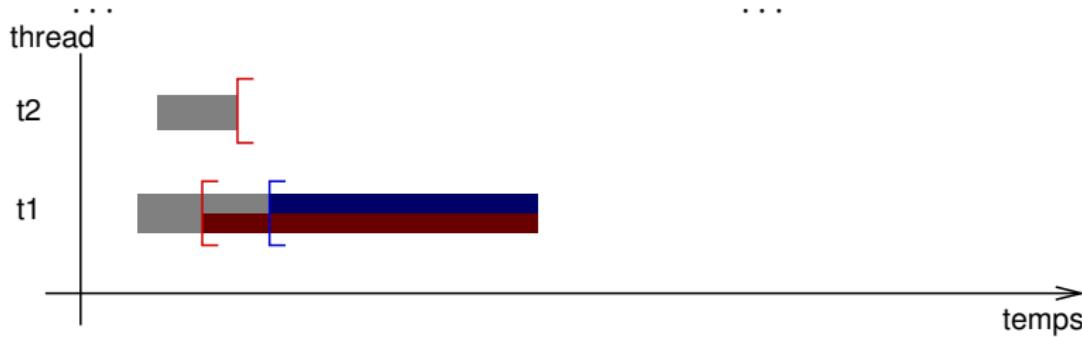
```

Thread 2

```

    ...
réserver(r1);
réserver(r2);
manip. ress. 2
manip. ress. 1 et 2
libérer(r1);
manip. ress. 2
libérer(r2);
    ...

```



# Eviter interblocage : réservations dans même ordre

Thread 1

```

    ...
réserver(r1);
manip. ress. 1
réserver(r2);
manip. ress. 2 et 1
libérer(r2);
manip. ress. 1
libérer(r1);
    ...

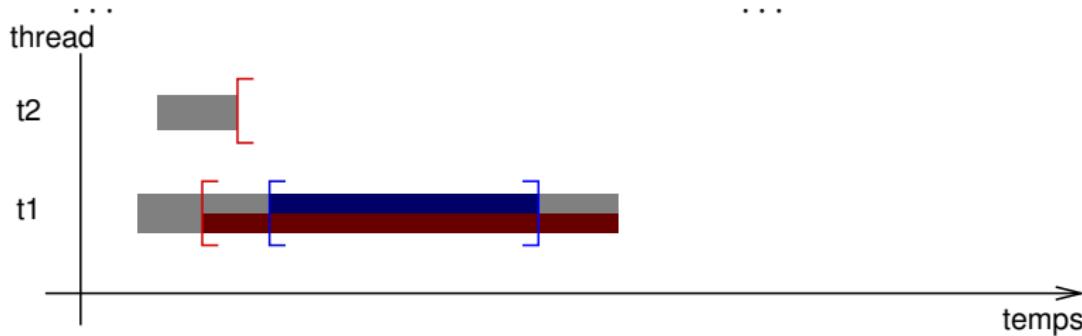
```

Thread 2

```

    ...
réserver(r1);
réserver(r2);
manip. ress. 2
manip. ress. 1 et 2
libérer(r1);
manip. ress. 2
libérer(r2);
    ...

```



# Eviter interblocage : réservations dans même ordre

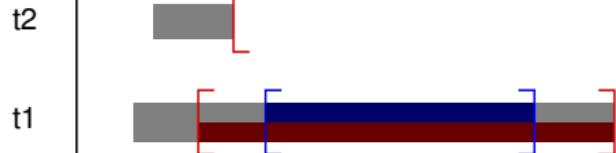
Thread 1

```

    ...
réserver(r1);
manip. ress. 1
réserver(r2);
manip. ress. 2 et 1
libérer(r2);
manip. ress. 1
libérer(r1);
    ...

```

thread



Thread 2

```

    ...
réserver(r1);
réserver(r2);
manip. ress. 2
manip. ress. 1 et 2
libérer(r1);
manip. ress. 2
libérer(r2);
    ...

```

...

...

...

→  
temps

# Eviter interblocage : réservations dans même ordre

Thread 1

```

    ...
réserver(r1);
manip. ress. 1
réserver(r2);
manip. ress. 2 et 1
libérer(r2);
manip. ress. 1
libérer(r1);
    ...

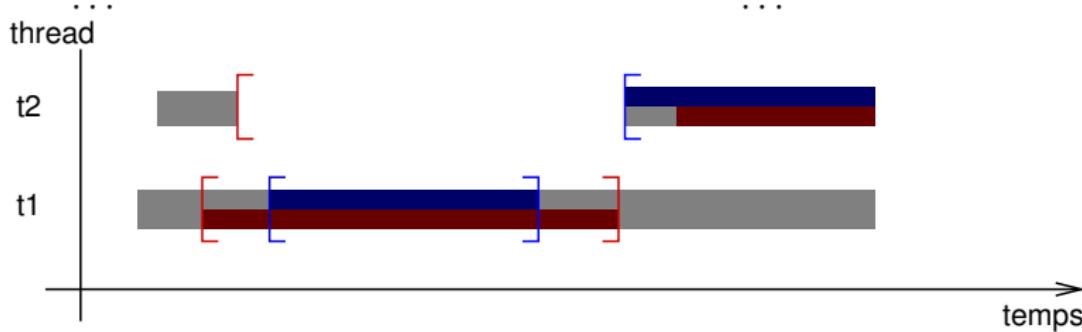
```

Thread 2

```

    ...
réserver(r1);
réserver(r2);
manip. ress. 2
manip. ress. 1 et 2
libérer(r1);
manip. ress. 2
libérer(r2);
    ...

```



# Eviter interblocage : réservations dans même ordre

Thread 1

```

    ...
réserver(r1);
manip. ress. 1
réserver(r2);
manip. ress. 2 et 1
libérer(r2);
manip. ress. 1
libérer(r1);
    ...

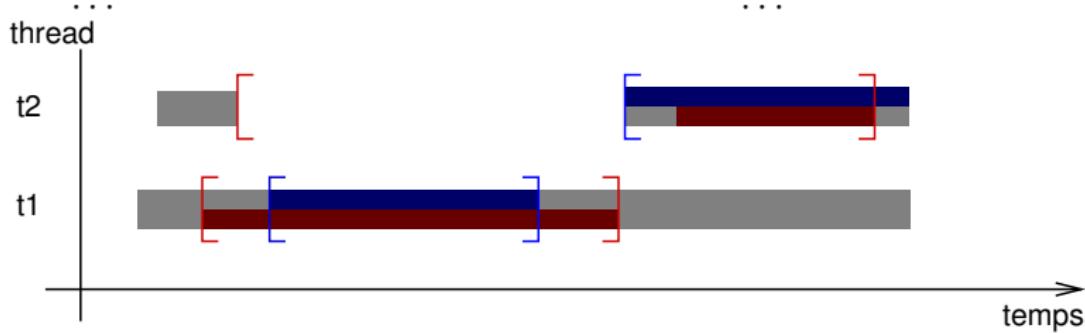
```

Thread 2

```

    ...
réserver(r1);
réserver(r2);
manip. ress. 2
manip. ress. 1 et 2
libérer(r1);
manip. ress. 2
libérer(r2);
    ...

```



# Eviter interblocage : réservations dans même ordre

Thread 1

```

    ...
réserver(r1);
manip. ress. 1
réserver(r2);
manip. ress. 2 et 1
libérer(r2);
manip. ress. 1
libérer(r1);
    ...

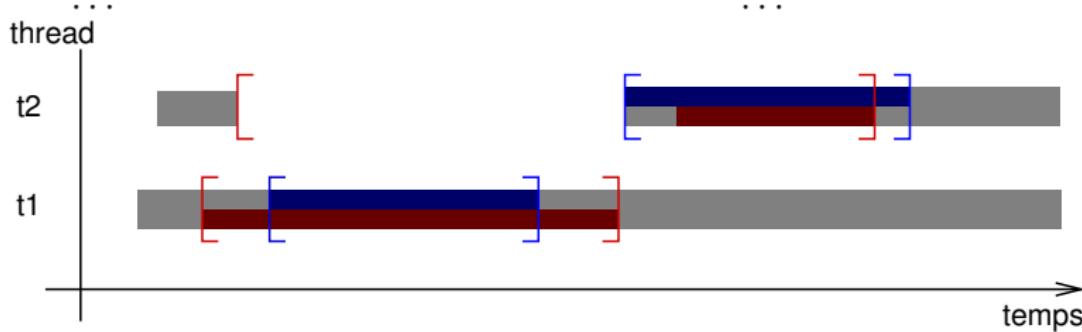
```

Thread 2

```

    ...
réserver(r1);
réserver(r2);
manip. ress. 2
manip. ress. 1 et 2
libérer(r1);
manip. ress. 2
libérer(r2);
    ...

```



## 2 . Section critique et mutex

### 2.1. Section critique

- Exclusion Mutuelle
- Pas d'interblocage
- Pas de famine

### 2.2. Verrou

- Une des trois réalisations possibles de SC
- Principe
- Risque d'interblocage

### 2.3. Verrous C++ <mutex>

- classe de base `std::mutex`
- autres classes `std::lock_guard` et `std::unique_lock`

# Verrou - std::mutex

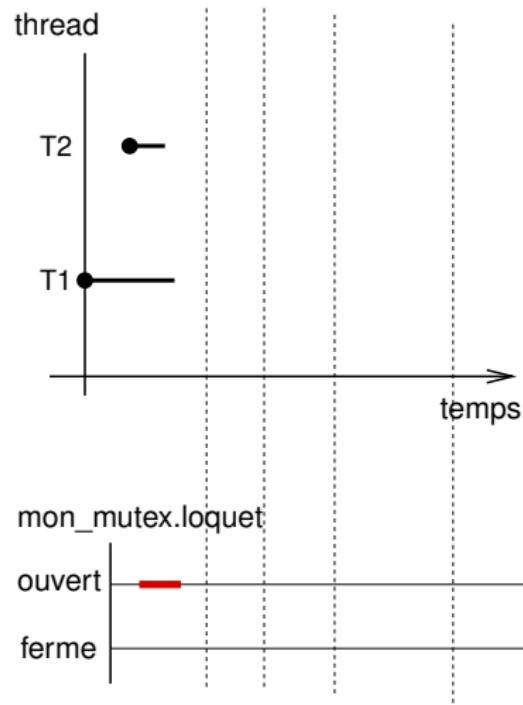
```
#include <mutex>
```

```
std::mutex mon_mutex;
```

```
mon_mutex.lock();
```

*...section critique ...*

```
mon_mutex.unlock();
```



# Verrou - std::mutex

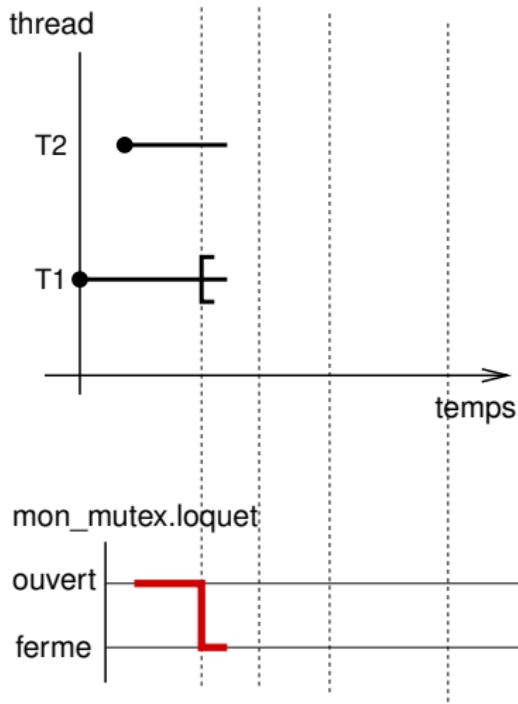
```
#include <mutex>
```

```
std::mutex mon_mutex;
```

```
mon_mutex.lock();
```

*...section critique ...*

```
mon_mutex.unlock();
```



# Verrou - std::mutex

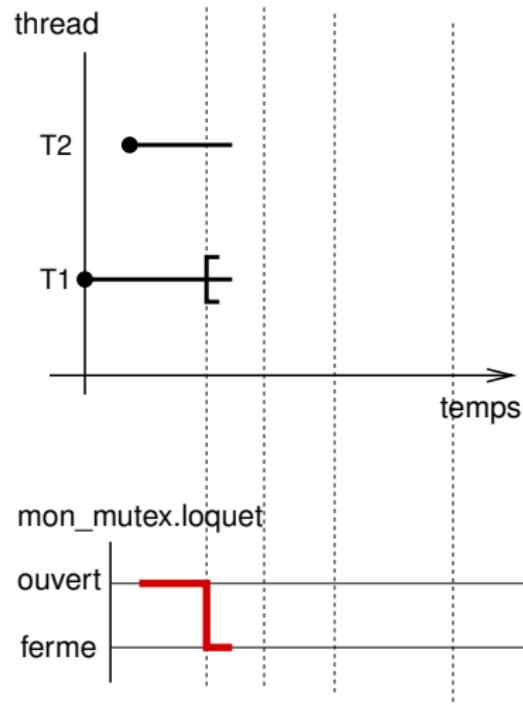
```
#include <mutex>
```

```
std::mutex mon_mutex;
```

```
mon_mutex.lock();
```

*...section critique ...*

```
mon_mutex.unlock();
```



# Verrou - std::mutex

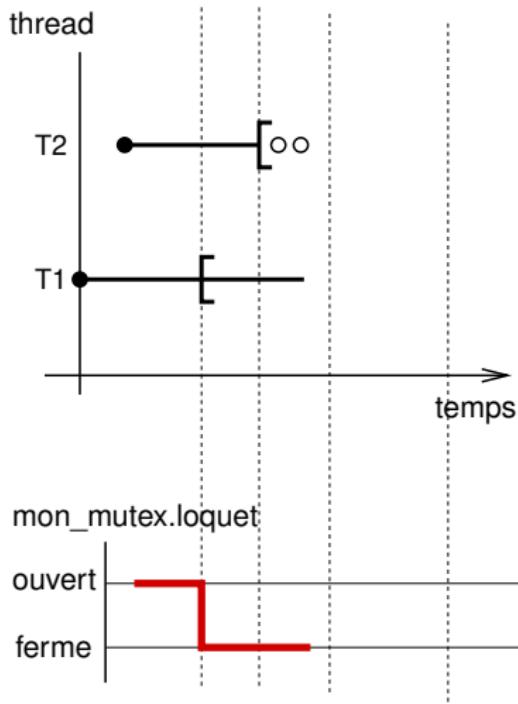
```
#include <mutex>
```

```
std::mutex mon_mutex;
```

```
mon_mutex.lock();
```

*...section critique ...*

```
mon_mutex.unlock();
```



# Verrou - std::mutex

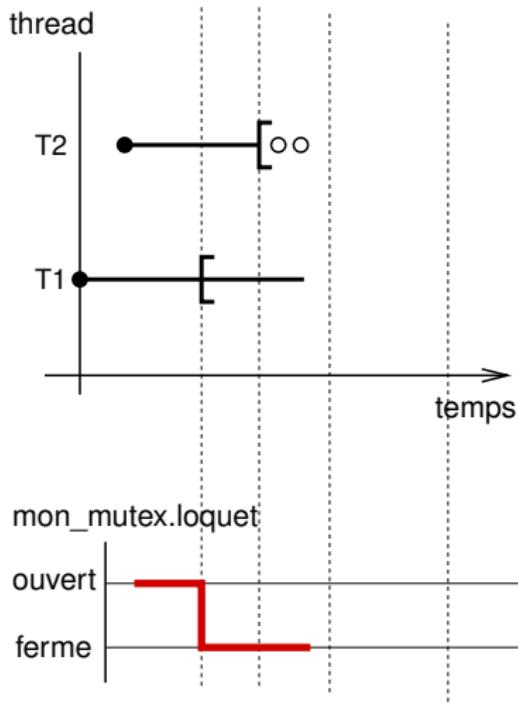
```
#include <mutex>
```

```
std::mutex mon_mutex;
```

```
mon_mutex.lock();
```

*...section critique ...*

```
mon_mutex.unlock();
```



# Verrou - std::mutex

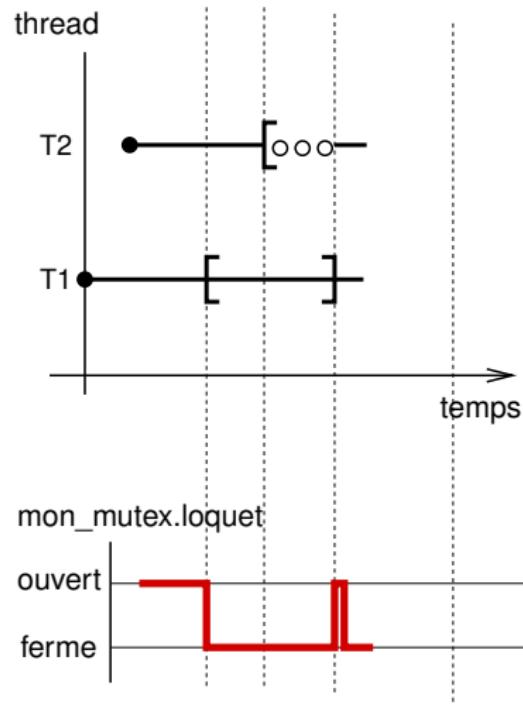
```
#include <mutex>
```

```
std::mutex mon_mutex;
```

```
mon_mutex.lock();
```

*...section critique ...*

```
mon_mutex.unlock();
```



# Verrou - std::mutex

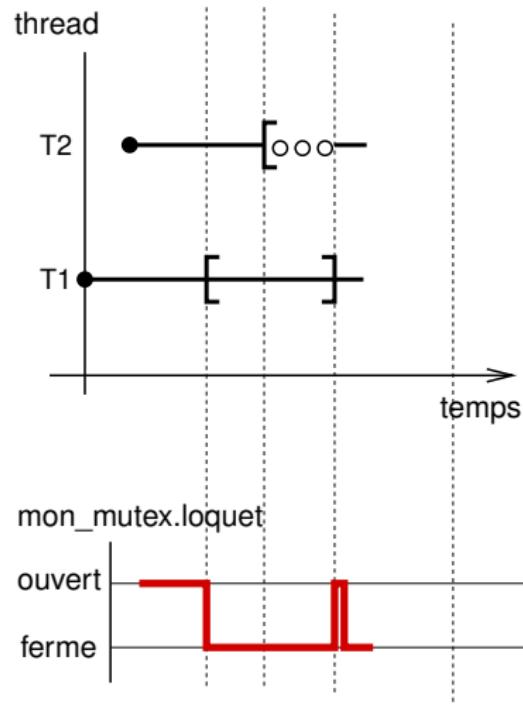
```
#include <mutex>
```

```
std::mutex mon_mutex;
```

```
mon_mutex.lock();
```

*...section critique ...*

```
mon_mutex.unlock();
```



# Verrou - std::mutex

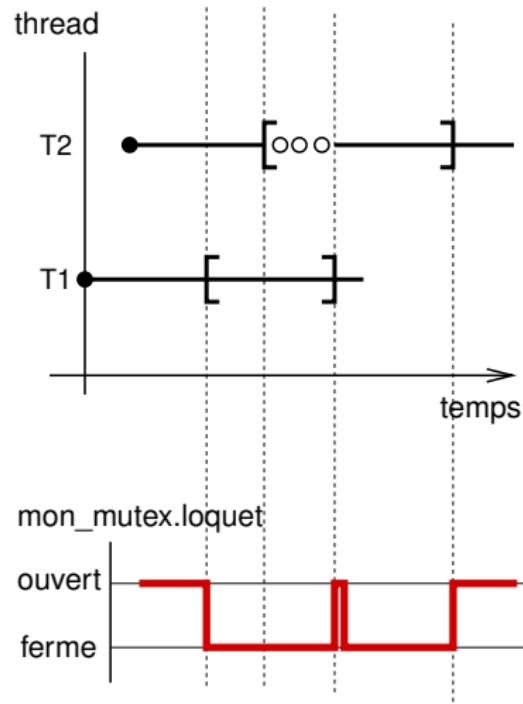
```
#include <mutex>
```

```
std::mutex mon_mutex;
```

```
mon_mutex.lock();
```

*...section critique ...*

```
mon_mutex.unlock();
```



# Verrou - `std::mutex`

## Remarque

- non copiable (comme `std::thread`)
- non *movable* (un `std::thread` l'est)

## Précautions

- FIFO non garanti : famine temporaire possible
- Si un thread déverrouille un mutex qu'il n'a pas verrouillé : comportement indéfini
- Si un thread verrouille à nouveau un mutex qu'il a déjà verrouillé : comportement indéfini (*deadlock* ou exception)

## Conseils

- pour chaque ens. de ressources protégées par un mutex :
  - une classe
  - attributs : les ressources + le mutex
  - méthodes : seuls accès aux ressources (en SC)
- équilibre entre créer des SC courtes / peu nombreuses

# 2 . Section critique et mutex

## 2.1. Section critique

- Exclusion Mutuelle
- Pas d'interblocage
- Pas de famine

## 2.2. Verrou

- Une des trois réalisations possibles de SC
- Principe
- Risque d'interblocage

## 2.3. Verrous C++ <mutex>

- classe de base `std::mutex`
- autres classes `std::lock_guard` et `std::unique_lock`

# Verrou - autres classes et fonctions utiles de <mutex>

## `std::lock_guard`

- emballage de mutex pour mécanisme RAII
- le mutex est verrouillé à la construction de l'emballage
- le mutex est déverrouillé à la destruction de l'emballage

```
#include <mutex>
using namespace std;

mutex mon_mutex;

{ //début de portée
    lock_guard<mutex> verrou(mon_mutex);
    ...section critique...
} //fin de portée
```

# Verrou - autres classes et fonctions utiles de <mutex>

## std::unique\_lock

- emballage de mutex pour RAII + utilisation classique
- le mutex est **par défaut** verrouillé à la construction de l'emballage
- le mutex est déverrouillé **si nécessaire** à la destruction

```
#include <mutex>
using namespace std;

mutex mon_mutex;

{ // début de portée
    unique_lock<mutex> verrou(mon_mutex);
    ...section critique...
}

} // fin de portée
```

# Verrou - autres classes et fonctions utiles de <mutex>

## std::unique\_lock

- emballage de mutex pour RAII + utilisation classique
- le mutex est **par défaut** verrouillé à la construction de l'emballage
- le mutex est déverrouillé **si nécessaire** à la destruction

```
#include <mutex>
using namespace std;

mutex mon_mutex;

{ // début de portée
    unique_lock<mutex> verrou(mon_mutex);
    ...section critique...
    verrou.unlock()
    ...hors section critique...
} // fin de portée
```

# Verrou - autres classes et fonctions utiles de <mutex>

## std::lock()

- fonction permettant de verrouiller plusieurs mutex sans interblocage
- utilisable avec `std::mutex` ou `std::unique_lock`

```
#include <mutex>
using namespace std;

mutex mon_mutex1, mon_mutex2;

lock(mon_mutex1, mon_mutex2);
...section critique pour les deux ressources ...

mon_mutex1.unlock(); mon_mutex2.unlock();
```

# Bilan

## En résumé

# Section critique

## Définition

morceau de code manipulant une ressource critique (non partageable)

## Propriétés

- Exclusion mutuelle
- Absence de Famine (souhaitable)
- Absence d'Interblocage

## Réalisation d'une section critique

- section d'entrée
- section de sortie

# Verrou

Définition : Dijkstra 1965 - sémaphore binaire

objet partageable entre processus/threads comportant :

- un booléen : loquet ouvert/fermé
- une liste FIFO de processus/threads demandant à entrer en SC

Procédures associées : verrouiller() déverrouiller()

- seuls accès aux composants du verrou
- exécution indivisible (via masquage d'interruption ou spinlock)
- **verrouiller()** : **si** loquet==fermé  
    **alors** bloquer le thread en queue de FIFO  
    **finsi**  
    loquet=fermé
- **déverrouiller()** : loquet=ouvert  
    **si** liste FIFO non vide  
    **alors** libérer le thread en tête de FIFO  
    **finsi**

# Verrou

## Application à la réalisation d'une SC

- **section d'entrée** : verrouiller(verrou)
- **section de sortie** : déverrouiller(verrou)

## Inconvénients

- manipulation de processus/threads : coûteux
- attention aux SC imbriquées => interblocage

## Quand l'utiliser ?

- pour des SC longues
- en mode noyau ou utilisateur
- sur mono ou multi-processeur

# Verrou - classes dans <mutex>

```
#include <mutex>

using namespace std;

mutex mon_mutex;

mon_mutex.lock();
...section critique...
mon_mutex.unlock();

{ //début de portée
    lock_guard<mutex> verrou(mon_mutex);
    ...section critique...
} //fin de portée
```