

# Synchronisation

Thomas Lavergne  
lavergne@lisn.fr

# Threads

## Définition

Le **thread** est l'unité de base du processus

- un processus peut avoir plusieurs threads

## Définition

Le **thread** est l'unité de base du processus

- un processus peut avoir plusieurs threads
- les threads partagent le même code
- les threads partagent les mêmes données
- les threads partagent le même environnement

## Définition

Le **thread** est l'unité de base du processus

- un processus peut avoir plusieurs threads
- les threads partagent le même code
- les threads partagent les mêmes données
- les threads partagent le même environnement
- les threads ont leur propre pile

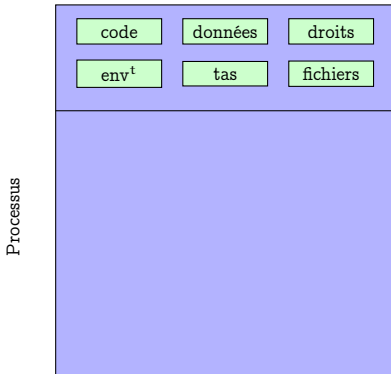
## Définition

Le **thread** est l'unité de base du processus

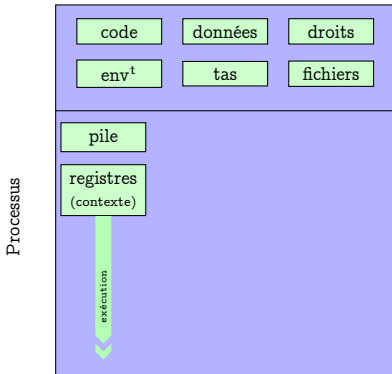
- un processus peut avoir plusieurs threads
- les threads partagent le même code
- les threads partagent les mêmes données
- les threads partagent le même environnement
- les threads ont leur propre pile

Beaucoup plus léger à créer

# Threads

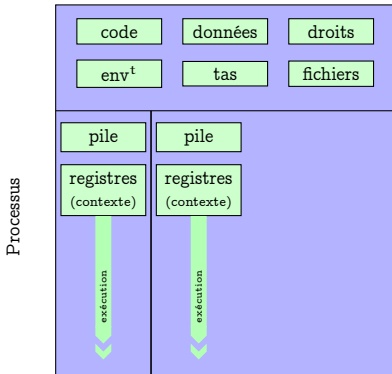


# Threads

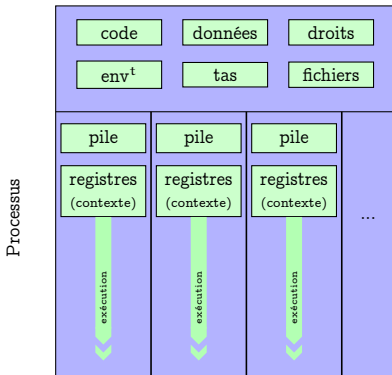




# Threads



# Threads



## Avantages

- Partage du code : gain de mémoire
- Partage des données : parallélisation simplifiée

## Avantages

- Partage du code : gain de mémoire
- Partage des données : parallélisation simplifiée

## Inconvénients

- Partage des données : parallélisation dangereuse

# Section critique

# Un exemple classique

Espace partagé



# Un exemple classique

Espace partagé

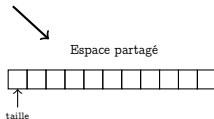


```
#define MAX 10  
int taille = 0;  
int boite[MAX];
```

# Un exemple classique

Producteur(s)

```
void produire(char c) {  
    while(taille==MAX)  
        ; // attendre que la boîte se vide  
    boîte[taille] = c;  
    taille++; // déposer et avancer  
}
```





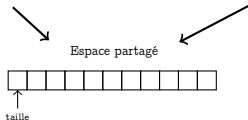
# Un exemple classique

Producteur(s)

```
void produire(char c) {  
    while(taille==MAX)  
        ; // attendre que la boîte se vide  
    boîte[taille] = c;  
    taille++; // déposer et avancer  
}
```

Consommateur(s)

```
char consommer() {  
    while (taille==0)  
        ; // attendre qu'il y ait quelque chose  
    taille--;  
    c = boîte[taille]; // lire et retirer  
    return c;  
}
```



# Un exemple classique

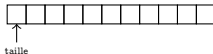
Producteur(s)

```
void produire(char c) {  
    while(taille==MAX)  
        ; // attendre que la boîte se vide  
    boîte[taille] = c;  
    taille++; // déposer et avancer  
}
```

Consommateur(s)

```
char consommer() {  
    while (taille==0)  
        ; // attendre qu'il y ait quelque chose  
    taille--;  
    c = boîte[taille]; // lire et retirer  
    return c;  
}
```

Espace partagé



Quand on ne fait que produire, tout va bien

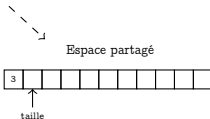
# Un exemple classique

Producteur(s)

```
void produire(char c) {  
    while(taille==MAX)  
        ; // attendre que la boîte se vide  
    boîte[taille] = c;  
    taille++; // déposer et avancer  
}
```

Consommateur(s)

```
char consommer() {  
    while (taille==0)  
        ; // attendre qu'il y ait quelque chose  
    taille--;  
    c = boîte[taille]; // lire et retirer  
    return c;  
}
```



Quand on ne fait que produire, tout va bien

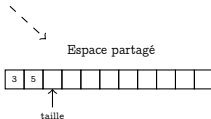
# Un exemple classique

Producteur(s)

```
void produire(char c) {  
    while(taille==MAX)  
        ; // attendre que la boîte se vide  
    boîte[taille] = c;  
    taille++; // déposer et avancer  
}
```

Consommateur(s)

```
char consommer() {  
    while (taille==0)  
        ; // attendre qu'il y ait quelque chose  
    taille--;  
    c = boîte[taille]; // lire et retirer  
    return c;  
}
```



Quand on ne fait que produire, tout va bien

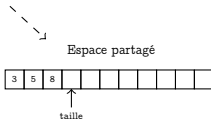
# Un exemple classique

Producteur(s)

```
void produire(char c) {  
    while(taille==MAX)  
        ; // attendre que la boîte se vide  
    boîte[taille] = c;  
    taille++; // déposer et avancer  
}
```

Consommateur(s)

```
char consommer() {  
    while (taille==0)  
        ; // attendre qu'il y ait quelque chose  
    taille--;  
    c = boîte[taille]; // lire et retirer  
    return c;  
}
```



Quand on ne fait que produire, tout va bien

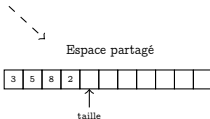
# Un exemple classique

Producteur(s)

```
void produire(char c) {  
    while(taille==MAX)  
        ; // attendre que la boîte se vide  
    boîte[taille] = c;  
    taille++; // déposer et avancer  
}
```

Consommateur(s)

```
char consommer() {  
    while (taille==0)  
        ; // attendre qu'il y ait quelque chose  
    taille--;  
    c = boîte[taille]; // lire et retirer  
    return c;  
}
```



Quand on ne fait que produire, tout va bien

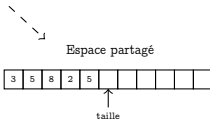
# Un exemple classique

Producteur(s)

```
void produire(char c) {  
    while(taille==MAX)  
        ; // attendre que la boîte se vide  
    boîte[taille] = c;  
    taille++; // déposer et avancer  
}
```

Consommateur(s)

```
char consommer() {  
    while (taille==0)  
        ; // attendre qu'il y ait quelque chose  
    taille--;  
    c = boîte[taille]; // lire et retirer  
    return c;  
}
```



Quand on ne fait que produire, tout va bien

# Un exemple classique

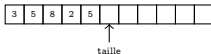
Producteur(s)

```
void produire(char c) {  
    while(taille==MAX)  
        ; // attendre que la boîte se vide  
    boîte[taille] = c;  
    taille++; // déposer et avancer  
}
```

Consommateur(s)

```
char consommer() {  
    while (taille==0)  
        ; // attendre qu'il y ait quelque chose  
    taille--;  
    c = boîte[taille]; // lire et retirer  
    return c;  
}
```

Espace partagé



Quand on ne fait que consommer, tout va bien



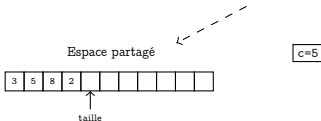
# Un exemple classique

Producteur(s)

```
void produire(char c) {  
    while(taille==MAX)  
        ; // attendre que la boîte se vide  
    boîte[taille] = c;  
    taille++; // déposer et avancer  
}
```

Consommateur(s)

```
char consommer() {  
    while (taille==0)  
        ; // attendre qu'il y ait quelque chose  
    taille--;  
    c = boîte[taille]; // lire et retirer  
    return c;  
}
```



Quand on ne fait que consommer, tout va bien

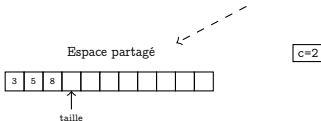
# Un exemple classique

Producteur(s)

```
void produire(char c) {  
    while(taille==MAX)  
        ; // attendre que la boîte se vide  
    boîte[taille] = c;  
    taille++; // déposer et avancer  
}
```

Consommateur(s)

```
char consommer() {  
    while (taille==0)  
        ; // attendre qu'il y ait quelque chose  
    taille--;  
    c = boîte[taille]; // lire et retirer  
    return c;  
}
```



Quand on ne fait que consommer, tout va bien

# Un exemple classique

Producteur(s)

```
void produire(char c) {  
    while(taille==MAX)  
        ; // attendre que la boîte se vide  
    boîte[taille] = c;  
    taille++; // déposer et avancer  
}
```

Consommateur(s)

```
char consommer() {  
    while (taille==0)  
        ; // attendre qu'il y ait quelque chose  
    taille--;  
    c = boîte[taille]; // lire et retirer  
    return c;  
}
```

Espace partagé



↑  
taille

Mais quand on entrelace...

# Un exemple classique

Producteur(s)

```
void produire(char c) {  
    while(taille==MAX)  
        ; // attendre que la boîte se vide  
    boîte[taille] = c;  
    taille++; // déposer et avancer  
}
```

Consommateur(s)

```
char consommer() {  
    while (taille==0)  
        ; // attendre qu'il y ait quelque chose  
    taille--;  
    c = boîte[taille]; // lire et retirer  
    return c;  
}
```

Espace partagé



↑  
taille

Mais quand on entrelace...

# Un exemple classique

Producteur(s)

```
void produire(char c) {  
    while(taille==MAX)  
        ; // attendre que la boîte se vide  
    boîte[taille] = c;  
    taille++; // déposer et avancer  
}
```

Consommateur(s)

```
char consommer() {  
    while (taille==0)  
        ; // attendre qu'il y ait quelque chose  
    taille--;  
    c = boîte[taille]; // lire et retirer  
    return c;  
}
```

Espace partagé



taille

Mais quand on entrelace...

# Un exemple classique

Producteur(s)

```
void produire(char c) {  
    while(taille==MAX)  
        ; // attendre que la boîte se vide  
    boîte[taille] = c;  
    taille++; // déposer et avancer  
}
```

Consommateur(s)

```
char consommer() {  
    while (taille==0)  
        ; // attendre qu'il y ait quelque chose  
    taille--;  
    c = boîte[taille]; // lire et retirer  
    return c;  
}
```

Espace partagé



taille

Mais quand on entrelace...

# Un exemple classique

Producteur(s)

```
void produire(char c) {  
    while(taille==MAX)  
        ; // attendre que la boîte se vide  
    boîte[taille] = c;  
    taille++; // déposer et avancer  
}
```

Consommateur(s)

```
char consommer() {  
    while (taille==0)  
        ; // attendre qu'il y ait quelque chose  
    taille--;  
    c = boîte[taille]; // lire et retirer  
    return c;  
}
```

Espace partagé



taille

Mais quand on entrelace...

# Un exemple classique

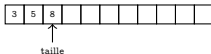
Producteur(s)

```
void produire(char c) {  
    while(taille==MAX)  
        ; // attendre que la boîte se vide  
    boîte[taille] = c;  
    taille++; // déposer et avancer  
}
```

Consommateur(s)

```
char consommer() {  
    while (taille==0)  
        ; // attendre qu'il y ait quelque chose  
    taille--;  
    c = boîte[taille]; // lire et retirer  
    return c;  
}
```

Espace partagé



Mais quand on entrelace...



# Un exemple classique

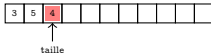
Producteur(s)

```
void produire(char c) {  
    while(taille==MAX)  
        ; // attendre que la boîte se vide  
    boîte[taille] = c;  
    taille++; // déposer et avancer  
}
```

Consommateur(s)

```
char consommer() {  
    while (taille==0)  
        ; // attendre qu'il y ait quelque chose  
    taille--;  
    c = boîte[taille]; // lire et retirer  
    return c;  
}
```

Espace partagé



Mais quand on entrelace...

## Remarque

Entrelacements au niveau du code binaire!

Coupure **entre** les instructions

Exemple: incrémentation

```
load  r0, @var
```

```
add   r0, #1
```

```
store r0, @var
```

## Remarque

Entrelacements au niveau du code binaire !

Coupure **entre** les instructions

Exemple : incrémentation

```
load  r0, @var
```

```
add   r0, #1
```

```
store r0, @var
```

## Section critique

L'accès aux **variables partagées** nécessite du code qui ne peut pas être interrompu.

## Remarque

Entrelacements au niveau du code binaire !

Coupure **entre** les instructions

Exemple : incrémentation

```
load  r0, @var
```

```
add   r0, #1
```

```
store r0, @var
```

## Section critique

L'accès aux **variables partagées** nécessite du code qui ne peut pas être interrompu.

des **sections critiques**

### Exclusion mutuelle

Garantie qu'**au plus un seul** processus ou thread est dans une section critique.

### Exclusion mutuelle

Garantie qu'**au plus un seul** processus ou thread est dans une section critique.

Qu'il soit en exécution, prêt ou en attente !

### Exclusion mutuelle

Garantie qu'**au plus un seul** processus ou thread est dans une section critique.

**Qu'il soit en exécution, prêt ou en attente !**

Les autres processus ou threads sont mis **en attente** s'ils tentent de l'exécuter.





### Exclusion Mutuelle

Un seul thread à la fois dans la section critique.

### Exclusion Mutuelle

Un seul thread à la fois dans la section critique.

### Déroulement

Un thread n'a pas d'influence sur le choix de qui peut entrer dans la section critique.

### Exclusion Mutuelle

Un seul thread à la fois dans la section critique.

### Dérroulement

Un thread n'a pas d'influence sur le choix de qui peut entrer dans la section critique.

### Vivacité

Un thread entre en section critique après un temps borné.

# Vérouillage



## Mécanisme de verrouillage

```
void verrouiller(int id);  
void déverrouiller(int id);
```

## Mécanisme de verrouillage

```
void verrouiller(int id);  
void déverrouiller(int id);
```

## Utilisation

*code non critique*

```
verrouiller(myid);
```

*code critique*

```
déverrouiller(myid);
```

*code non critique*

```
int enSC[2] = {0, 0};

void verouiller(int id) {
    while (enSC[1 - id])
        ;
    enSC[id] = 1;
}

void deverouiller(int id) {
    enSC[id] = 0;
}
```



Ce code contient une section critique...

```
void verouiller(int id) {  
    while (enSC[1 - id])  
        ;  
    enSC[id] = 1;  
}
```

enSC = { 0, 0 }

Ce code contient une section critique...

```
t0 void verouiller(int id) {  
    while (enSC[1 - id])  
        ;  
    enSC[id] = 1;  
}
```

enSC = { 0, 0 }

Ce code contient une section critique...

```
void verouiller(int id) {  
t0   while (enSC[1 - id])  
        ;  
        enSC[id] = 1;  
}
```

enSC = { 0, 0 }

Ce code contient une section critique...

```
void verouiller(int id) {  
    while (enSC[1 - id])  
        ;  
    t0    enSC[id] = 1;  
}
```

enSC = { 0, 0 }

Ce code contient une section critique...

```
t1 void verouiller(int id) {  
    while (enSC[1 - id])  
        ;  
t0    enSC[id] = 1;  
}
```

enSC = { 0, 0 }

Ce code contient une section critique...

```
void verouiller(int id) {  
t1   while (enSC[1 - id])  
      ;  
t0   enSC[id] = 1;  
}
```

enSC = { 0, 0 }

Ce code contient une section critique...

```
void verouiller(int id) {  
    while (enSC[1 - id])  
        ;  
    enSC[id] = 1;  
}
```

$t_0$   $t_1$

enSC = { 0, 0 }

Ce code contient une section critique...

```
void verouiller(int id) {  
    while (enSC[1 - id])  
        ;  
     $t_0$     enSC[id] = 1;  
     $t_1$ }  
}
```

enSC = { 0, 1 }



Ce code contient une section critique...

```
void verouiller(int id) {  
    while (enSC[1 - id])  
        ;  
    enSC[id] = 1;  
t0}
```

enSC = { 1, 1 }

Les deux threads sont en section critique !

```
int enSC[2] = {0, 0};

void verouiller(int id) {
    enSC[id] = 1;
    while (enSC[1 - id])
        ;
}

void deverouiller(int id) {
    enSC[id] = 0;
}
```

## Risque d'interblocage

```
void verouiller(int id) {  
    enSC[id] = 1;  
    while (enSC[1 - id])  
        ;  
}
```

$\text{enSC} = \{ 0, 0 \}$

## Risque d'interblocage

```
t0 void verouiller(int id) {  
    enSC[id] = 1;  
    while (enSC[1 - id])  
        ;  
}
```

enSC = { 0, 0 }

## Risque d'interblocage

```
void verouiller(int id) {  
t0   enSC[id] = 1;  
      while (enSC[1 - id])  
          ;  
}
```

enSC = { 0, 0 }

## Risque d'interblocage

```
void verouiller(int id) {  
    enSC[id] = 1;  
  
t0    while (enSC[1 - id])  
        ;  
}
```

enSC = { 1, 0 }

## Risque d'interblocage

```
t1 void verouiller(int id) {  
    enSC[id] = 1;  
t0    while (enSC[1 - id])  
        ;  
}
```

enSC = { 1, 0 }

## Risque d'interblocage

```
void verouiller(int id) {  
t1    enSC[id] = 1;  
t0    while (enSC[1 - id])  
        ;  
}
```

enSC = { 1, 0 }



## Risque d'interblocage

```
void verouiller(int id) {  
    enSC[id] = 1;  
    while (enSC[1 - id])  
        ;  
}
```

$t_0$   $t_1$

enSC = { 1, 1 }

## Risque d'interblocage

```
void verouiller(int id) {  
    enSC[id] = 1;  
    while (enSC[1 - id])  
        ;  
}
```

enSC = { 1, 1 }

## Risque d'interblocage

```
void verouiller(int id) {  
    enSC[id] = 1;  
    while (enSC[1 - id])  
        ;  
}
```

$t_0$   $t_1$

enSC = { 1, 1 }

Les deux threads sont en attente!

```
int enSC[2] = {0, 0};  
int tour;  
  
void verouiller(int id) {  
    tour = 1 - id;  
    enSC[id] = 1;  
    while (enSC[1 - id] && tour == 1 - id)  
        ;  
}  
  
void deverouiller(int id) {  
    enSC[id] = 0;  
}
```

## Risque d'interblocage

```
void verouiller(int id) {  
    tour = 1 - id;  
    enSC[id] = 1;  
    while (enSC[1 - id] && tour == 1 - id)  
        ;  
}
```

```
enSC = { 0, 0 }  
tour = ?
```

## Risque d'interblocage

```
t0 void verouiller(int id) {  
    tour = 1 - id;  
    enSC[id] = 1;  
    while (enSC[1 - id] && tour == 1 - id)  
        ;  
}
```

```
enSC = { 0, 0 }  
tour = ?
```

## Risque d'interblocage

```
void verouiller(int id) {  
t0   tour = 1 - id;  
      enSC[id] = 1;  
      while (enSC[1 - id] && tour == 1 - id)  
          ;  
}
```

```
enSC = { 0, 0 }  
tour = ?
```

## Risque d'interblocage

```
void verouiller(int id) {  
    tour = 1 - id;  
  
t0    enSC[id] = 1;  
    while (enSC[1 - id] && tour == 1 - id)  
        ;  
}
```

```
enSC = { 0, 0 }  
tour = 1
```



## Risque d'interblocage

```
void verouiller(int id) {  
    tour = 1 - id;  
    enSC[id] = 1;  
  
t0    while (enSC[1 - id] && tour == 1 - id)  
        ;  
}
```

```
enSC = { 1, 0 }  
tour = 1
```

## Risque d'interblocage

```
t1 void verouiller(int id) {  
    tour = 1 - id;  
    enSC[id] = 1;  
t0    while (enSC[1 - id] && tour == 1 - id)  
        ;  
}
```

```
enSC = { 1, 0 }  
tour = 1
```

## Risque d'interblocage

```
void verouiller(int id) {  
t1    tour = 1 - id;  
        enSC[id] = 1;  
t0    while (enSC[1 - id] && tour == 1 - id)  
        ;  
}
```

```
enSC = { 1, 0 }  
tour = 1
```

## Risque d'interblocage

```
void verouiller(int id) {  
    tour = 1 - id;  
    t1    enSC[id] = 1;  
    t0    while (enSC[1 - id] && tour == 1 - id)  
        ;  
}
```

```
enSC = { 1, 0 }  
tour = 0
```

## Risque d'interblocage

```
void verouiller(int id) {  
    tour = 1 - id;  
    enSC[id] = 1;  
t0t1    while (enSC[1 - id] && tour == 1 - id)  
        ;  
}
```

```
enSC = { 1, 1 }  
tour = 0
```

## Risque d'interblocage

```
void verouiller(int id) {  
    tour = 1 - id;  
    enSC[id] = 1;  
    t0 while (enSC[1 - id] && tour == 1 - id)  
    t1      ;  
}
```

```
enSC = { 1, 1 }  
tour = 0
```

## Risque d'interblocage

```
void verouiller(int id) {  
    tour = 1 - id;  
    enSC[id] = 1;  
    while (enSC[1 - id] && tour == 1 - id)  
        ;  
}
```

$t_1$   
 $t_0$

enSC = { 1, 1 }  
tour = 0

Un seul thread est passé!

## Problème

Les threads font de l'**attente active**.

```
while (enSC[1 - id] && tour == 1 - id) ;
```

Gaspillage de l'UC

## Solution

Des appels système :

- wait : l'OS met le thread attente
- notify : l'OS réveille les threads en attente



```
int enSC[2] = {0, 0};  
int tour;  
  
void verouiller(int id) {  
    tour = 1 - id;  
    enSC[id] = 1;  
    while (enSC[1 - id] && tour == 1 - id)  
        wait();  
}  
  
void deverouiller(int id) {  
    enSC[id] = 0;  
    notify();  
}
```

Plus de 2 threads

Généralisation : algorithme de Dekker

Plus de 2 threads

Généralisation : algorithme de Dekker

Inutilisable car trop lent

Plus de 2 threads

Généralisation : algorithme de Dekker

Inutilisable car trop lent

En pratique

Version 1 ou 2

Plus de 2 threads

Généralisation : algorithme de Dekker

Inutilisable car trop lent

En pratique

Version 1 ou 2 avec désactivation des interruptions par l'OS.