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 même données
- les threads partagent le même environnement

Définition

Le thread est l'unité de base du processus

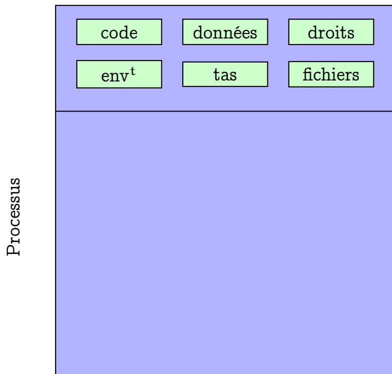
- un processus peut avoir plusieurs threads
- les threads partagent le même code
- les threads partagent les même données
- les threads partagent le même environnement
- les threads ont leur propre pile

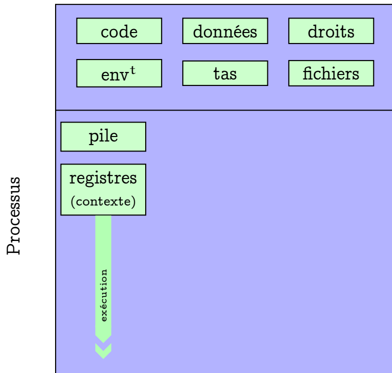
Définition

Le thread est l'unité de base du processus

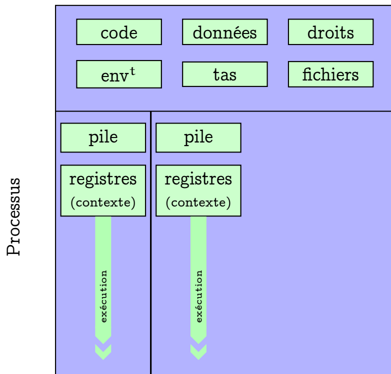
- un processus peut avoir plusieurs threads
- les threads partagent le même code
- les threads partagent les même données
- les threads partagent le même environnement
- les threads ont leur propre pile

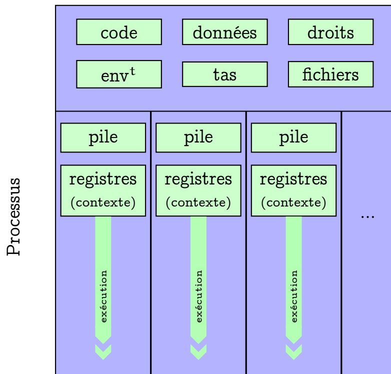
Beaucoup plus léger à créer


code
données
droits
env^t
tas
fichiers
Processus



code
données
droits
env^t
tas
fichiers
pile
registres
(contexte)
exécution
Processus


code
données
droits
env^t
tas
fichiers
pile
pile
registres
(contexte)
registres
(contexte)
exécution
exécution
Processus


code
données
droits
env^t
tas
fichiers
pile
registres
(contexte)
exécution
pile
registres
(contexte)
exécution
pile
registres
(contexte)
exécution
...
Processus

Avantages

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

Avantages

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

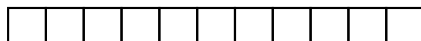
Inconvénients

- Partage des données : parallélisation dangereuse

Section critique

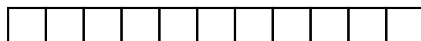
Un exemple classique

Espace partagé



Un exemple classique

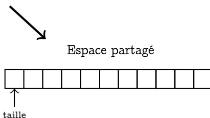
Espace partagé



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

Producteur(s)

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



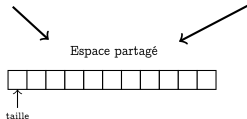
Un exemple classique

Producteur(s)

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

Consommateur(s)

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

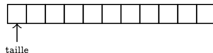


Producteur(s)

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

Consommateur(s)

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



Quand on ne fait que produire, tout va bien

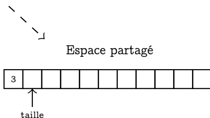
Un exemple classique

Producteur(s)

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

Consommateur(s)

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



Quand on ne fait que produire, tout va bien

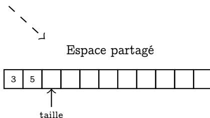
Un exemple classique

Producteur(s)

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

Consommateur(s)

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



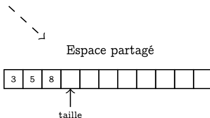
Quand on ne fait que produire, tout va bien

Producteur(s)

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

Consommateur(s)

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



Quand on ne fait que produire, tout va bien

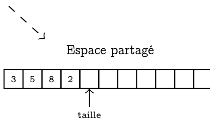
Un exemple classique

Producteur(s)

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

Consommateur(s)

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



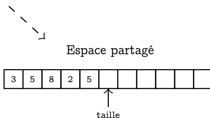
Quand on ne fait que produire, tout va bien

Producteur(s)

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

Consommateur(s)

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



Quand on ne fait que produire, tout va bien

Un exemple classique

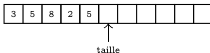
Producteur(s)

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

Consommateur(s)

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

Espace partagé



Quand on ne fait que consommer, tout va bien

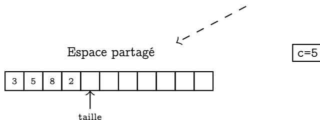
Un exemple classique

Producteur(s)

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

Consommateur(s)

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



Quand on ne fait que consommer, tout va bien

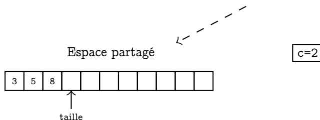
Un exemple classique

Producteur(s)

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

Consommateur(s)

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



Quand on ne fait que consommer, tout va bien

Un exemple classique

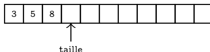
Producteur(s)

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

Consommateur(s)

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

Espace partagé



Mais quand on entrelace...

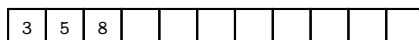
Producteur(s)

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

Consommateur(s)

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

Espace partagé



taille

Mais quand on entrelace...

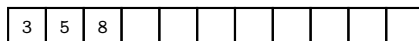
Producteur(s)

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

Consommateur(s)

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

Espace partagé



taille

Mais quand on entrelace...

Un exemple classique

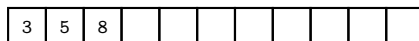
Producteur(s)

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

Consommateur(s)

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

Espace partagé



taille

Mais quand on entrelace...

Un exemple classique

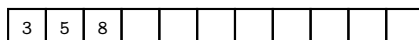
Producteur(s)

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

Consommateur(s)

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

Espace partagé



Mais quand on entrelace...

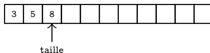
Producteur(s)

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

Consommateur(s)

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

Espace partagé



Mais quand on entrelace...

Un exemple classique

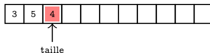
Producteur(s)

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

Consommateur(s)

```
char consommer() {
   while (taille==0)
      ; // attendre qu'il y ait quelque chose
   taille--;
   c = boite[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.

Déroulement

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

Les mutex

Méchanisme de vérouillage

```
void vérouiller(int id);
void dévérouiller(int id);
```

Méchanisme de vérouillage

```
void vérouiller(int id);
void dévérouiller(int id);
```

Utilisation

code non critique

```
vérouiller(myid);
```

code critique

```
dévérouiller(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...

t_0
```
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])
            ;
t0     enSC[id] = 1;
t1}
```

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

Ce code contient une section critique...

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

t_0

```
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'interbloquage

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

```
enSC = { 0, 0 }
```

Risque d'interbloquage

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

```
enSC = { 0, 0 }
```

Risque d'interbloquage

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

```
enSC = { 0, 0 }
```

Risque d'interbloquage

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

enSC = { 1, 0 }

Risque d'interbloquage

t_1 / t_0

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

enSC = { 1, 0 }

Risque d'interbloquage

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

enSC = { 1, 0 }

Risque d'interbloquage

$t_0 t_1$

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

enSC = { 1, 1 }

Risque d'interbloquage

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

enSC = { 1, 1 }

Risque d'interbloquage

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

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;
}
```

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

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

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

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

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

enSC = { 0, 0 }

tour = ?

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

enSC = { 0, 0 }

tour = 1

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

enSC = { 1, 0 }
tour = 1

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

enSC = { 1, 0 }

tour = 1

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

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

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

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

$t_0 t_1$

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

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

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

enSC = { 1, 1 }

tour = 0

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

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

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 interuptions par l'OS.

Sémaphores

Principe (simplifié)

On définit un objet partagé. (le sémaphore)

- que l'on peut acquerir;
- qui met en attente ceux qui le demandent ;
- qui donne la main dans l'ordre des demandes.

Principe (simplifié)

On définit un objet partagé. (le sémaphore)

- que l'on peut acquerir;
- qui met en attente ceux qui le demandent ;
- qui donne la main dans l'ordre des demandes.

Utilisation

- Acquisition avant d'entrer en SC
- Libération après être sorti de SC

Les sémaphore sont des primitives de haut niveau

Les sémaphore sont des primitives de haut niveau

Implémentation

Les fonctions `acquire(sem)` et `release(sem)` contiennent elle même des sections critiques

Les sémaphore sont des primitives de haut niveau

Implémentation

Les fonctions `acquire(sem)` et `release(sem)` contiennent elle même des sections critiques

Elles sont fournies par l'OS

Les sémaphore sont des primitives de haut niveau

Implémentation

Les fonctions `acquire(sem)` et `release(sem)` contiennent elle même des sections critiques

Elles sont fournies par l'OS

Avantage

Les sémaphores permettent de définir le nombre de ressources disponibles :

- Le nombre de threads pouvant entrer dans la SC en même temps

Conclusion

Threads

- plus léger que les processus
- mais plus difficiles à utiliser

Synchronisation

- Partager des ressources
- Notion de section critique
- Exclusion mutuelle, déroulement et vivacité
- Problème d'attente active: `wait` et `notify`
- Mutex et sémaphores