Systèmes d'exploitations

Cours 3: Multi threading

Matthieu Lemerre
CEA LIST

Année 2023-2024

Sommaire

Introduction

On souhaite développer un atelier d'émission télégraphique 2.0, dont les spécifications sont les suivantes

- Le télégraphiste entrera son texte au clavier. Il ne faut pas oublier de touche (mémoire du clavier limitée).
- Le texte entré sera traduit en morse par la machine.
- La traduction sera affichée sur la LED de la machine (temps d'une unité: 1 seconde, avec une jigue de +/- 10ms)
- On utilisera le temps processeur libre pour accéder à internet et miner du bitcoin

Une tentative en mono-tâche (programme *séquentiel*)

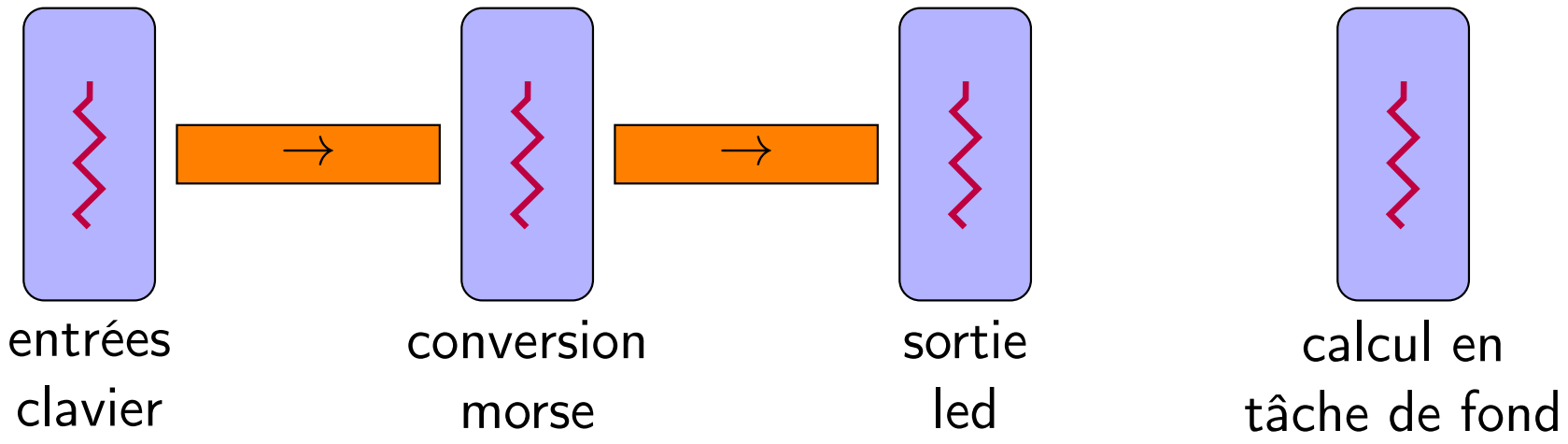
```
char input_buffer[100];
int in_idx = 0;
int out_idx = 0;
_Bool output_buffer[1000];

int last_time;
while(1) {
 if(char_available())
    input_buffer[in_idx++] = getchar();
 int new_time = current_time();
 if(new_time > last_time)
   set_led_to(output_buffer[out]);
 translate_input_to_output();
 // et du code réseau et cryptographique
 //découpé en paquets qui durent < 10 ms
}
```

Une tentative en mono-tâche (programme *séquentiel*)

```
char input_buffer[100];
int in_idx = 0;
int out_idx = 0;
_Bool output_buffer[1000];

int last_time;
while(1) {
 if(char_available())
    input_buffer[in_idx++] = getchar();
 int new_time = current_time();
 if(new_time > last_time)
   set_led_to(output_buffer[out]);
 translate_input_to_output();
 // et du code réseau et cryptographique
 //découpé en paquets qui durent < 10 ms
}
```

- Est-ce que la période de 10ms est la bonne pour l'entrée clavier?
- La jigue sur la led est importante en moyenne (mais jigue maximale respectée)
- Découper le code bitcoin en paquets inférieurs <10ms est pénible et fragile

Une tentative en multi-tâche (programme *concurrent*)



4 tâches, deux tuyaux de communication:

- Une tâche réveillée automatiquement à chaque appui sur le clavier (priorité élevée)
- Une tâche qui s'exécute toutes les secondes pour piloter la LED (priorité élevée aussi)
- Une tâche de calcul entre ces deux là (priorité moins importante que ces dernières)
- La fonction de minage de bitcoin (priorité la plus basse).

Intérêt d'une conception multi-tâche

- Optimiser l'utilisation de la machine
  - Plusieurs fonctionnalités fournies par le même ordinateur
  - Parallélisation des entrées-sorties avec le CPU
- Traiter des évènements asynchrones
  - Interruptions à des instants pas entièrement déterminés
- Simplifier la conception
  - On peut découper en tâche comme on découpe en fonctions,modules,classes. . .
  - Tâche = unité de partage des ressources entre les objectifs fonctionnels
  - Tâches ayant des échelles de temps différentes
  - Des degrés de criticité différents

Intérêt d'une conception multi-tâche

- Optimiser l'utilisation de la machine
  - Plusieurs fonctionnalités fournies par le même ordinateur
  - Parallélisation des entrées-sorties avec le CPU
- Traiter des évènements asynchrones
  - Interruptions à des instants pas entièrement déterminés
- Simplifier la conception
  - On peut découper en tâche comme on découpe en fonctions,modules,classes...
  - Tâche = unité de partage des ressources entre les objectifs fonctionnels
  - Tâches ayant des échelles de temps différentes
  - Des degrés de criticité différents

Et aussi: (mais pas seulement!)

- Profiter des multiples resources de calcul
  - Multiple processeurs
  - Multiple coeurs
  - Multiples machines physiques ou virtuelles

Concurrence $\neq$ parallélisme

Concurrence = tâches accèdent simultanément à une même ressource
Parallélisme = utilisation de plusieurs resources

Les grandes questions du multi-tâche

1. Qu'est-ce qu'une tâche? Comment en créer?
2. Comment les tâches *communiquent*, se *synchronisent*, *partagent* les ressources?
3. *Ordonnancement*: Quelle tâche exécuter?
4. Comment *protéger* les tâches les unes des autres?

Les grandes questions du multi-tâche

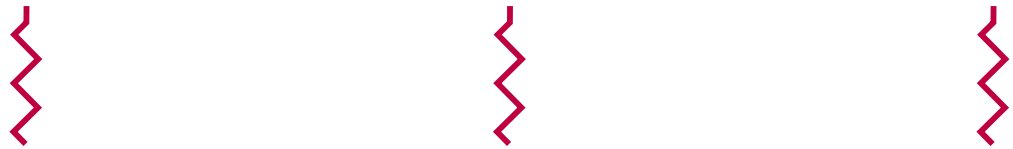
1. Qu'est-ce qu'une tâche? Comment en créer?
2. Comment les tâches *communiquent*, se *synchronisent*, *partagent* les ressources?
3. *Ordonnancement*: Quelle tâche exécuter?
4. Comment *protéger* les tâches les unes des autres?

La gestion du multi-tâche est la principale responsabilité de l'OS

- Quels mécanismes sont fournis par l'OS pour répondre à ces questions?
- Comment les utiliser? ($\rightarrow$ cours de programmation concurrente)
- Comment fonctionnent-ils? Comment sont-ils implantés?

Sommaire

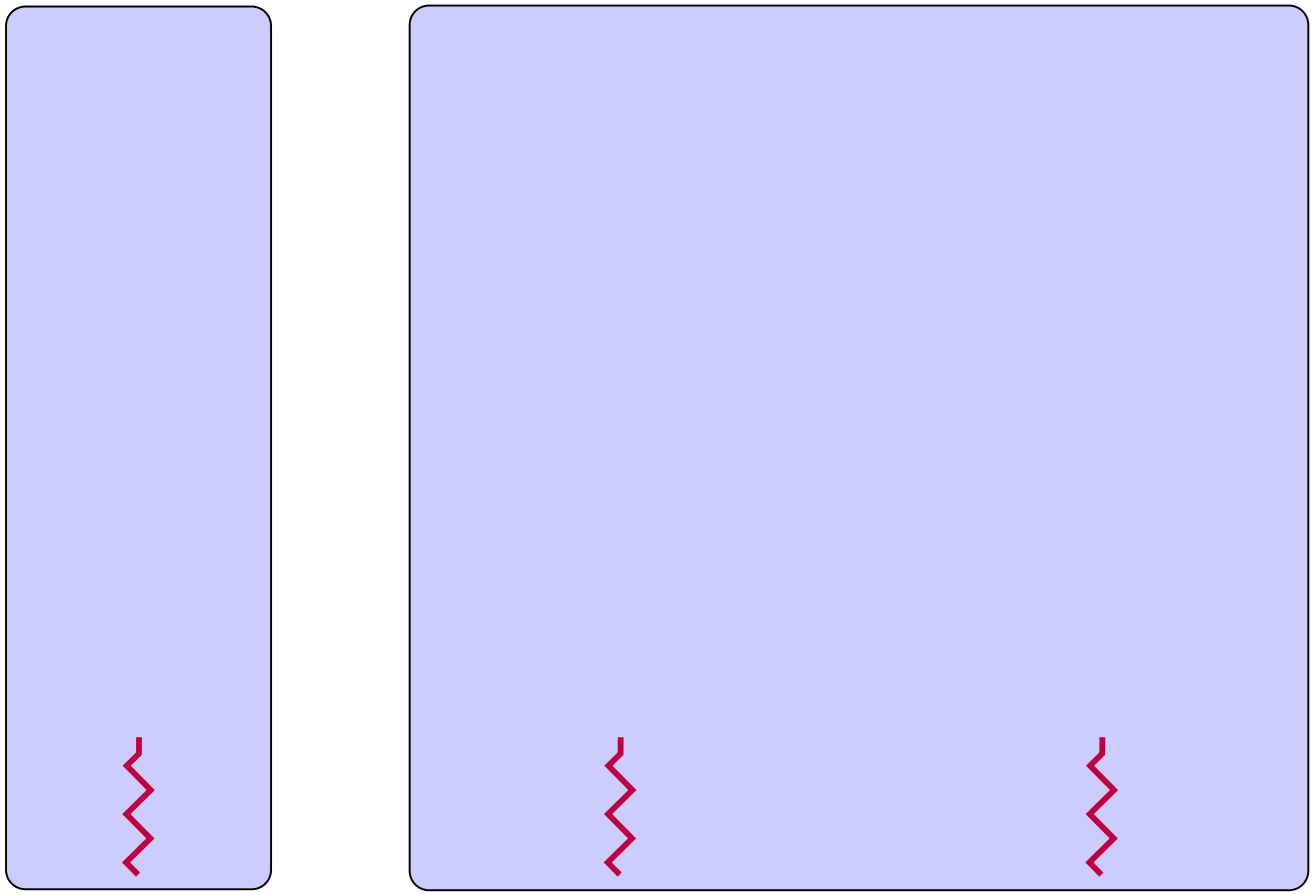
Thread, processus, tâche



Définition (Thread)

- Thread = séquence d'exécution indépendante
- Unité d'ordonnancement (on ordonnance des threads)

Thread, processus, tâche

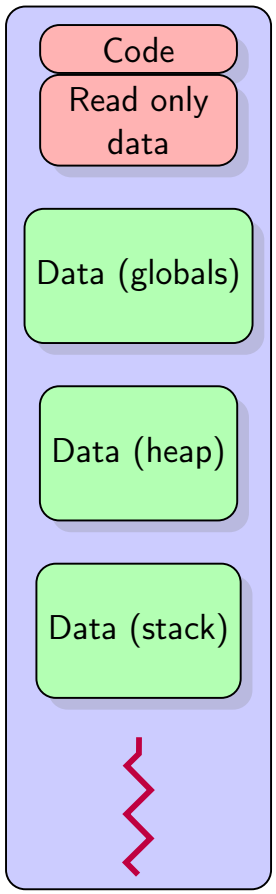


Processus mono-threadé Processus multi-threadé

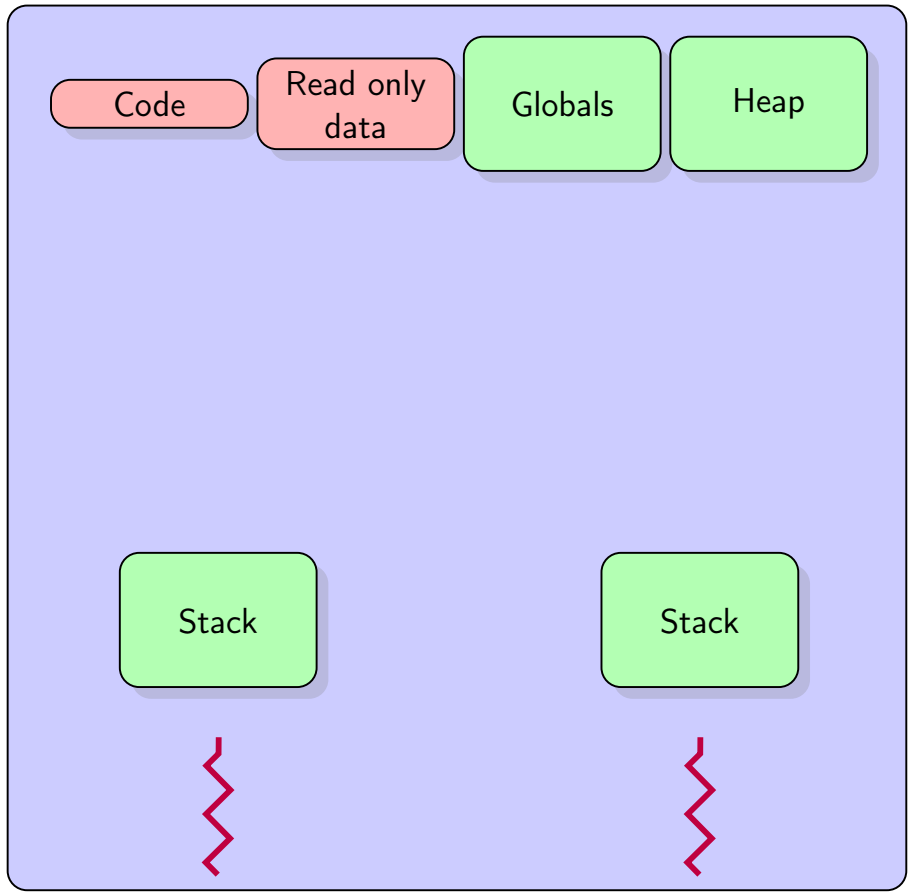
Définition (Processus)

- Processus = threads + permissions (dont: espace d'adressage séparé)
- En général: instance de l'exécution d'un programme (fichier .exe)

Thread, processus, tâche



Processus mono-threadé

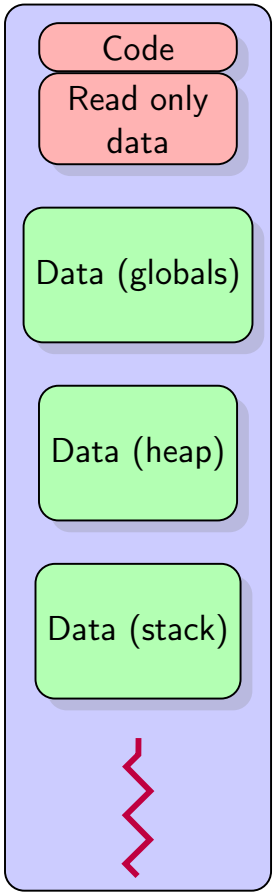


Processus multi-threadé

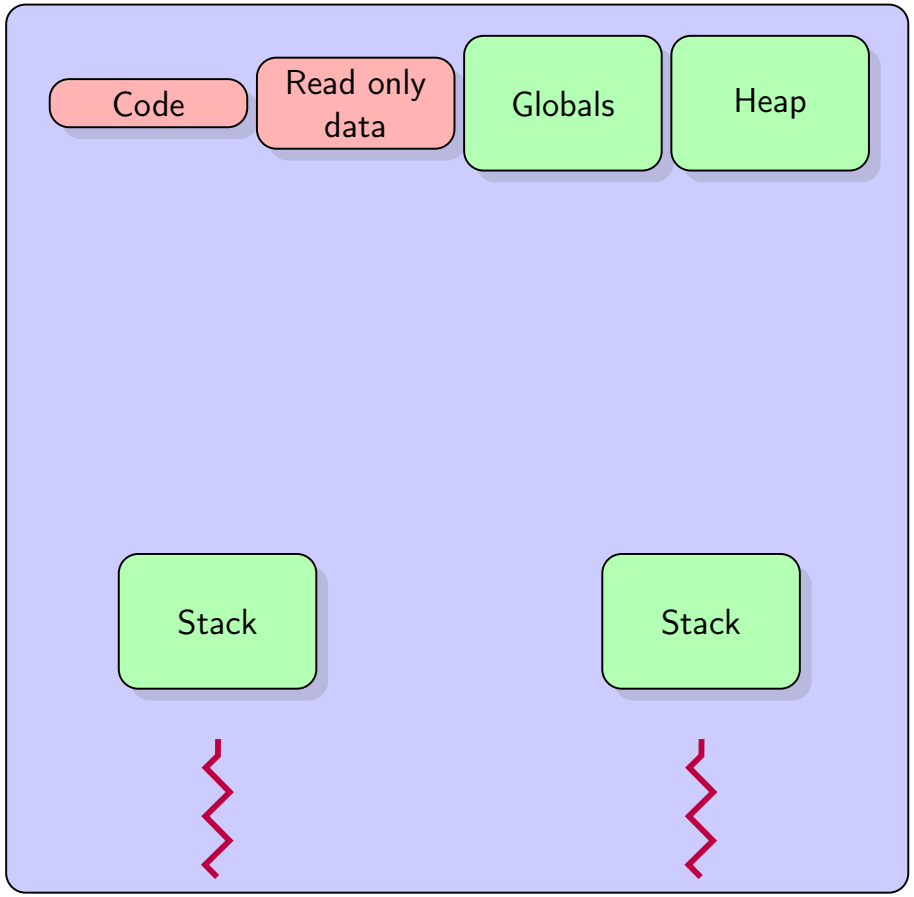
Définition (Processus)

- Processus = threads + permissions (dont: espace d'adressage séparé)
- En général: instance de l'exécution d'un programme (fichier .exe)

Thread, processus, tâche



Processus mono-threadé



Processus multi-threadé

Définition (Tâche)

- Terme ambigu
- Souvent synonyme de processus, mais parfois de thread

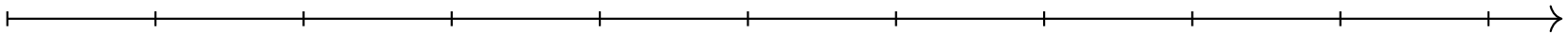
Concrètement: le multi-thread sur un seul processeur

Exemple (Thread A)

```
int a = 0; int b = 0;

while(1){
  a++; b++;

}
```

Exemple (Thread B)

```
doing();
other();

things();
```



- Exécution entrelacée
- Sauvegarde et restauration du *contexte d'exécution*

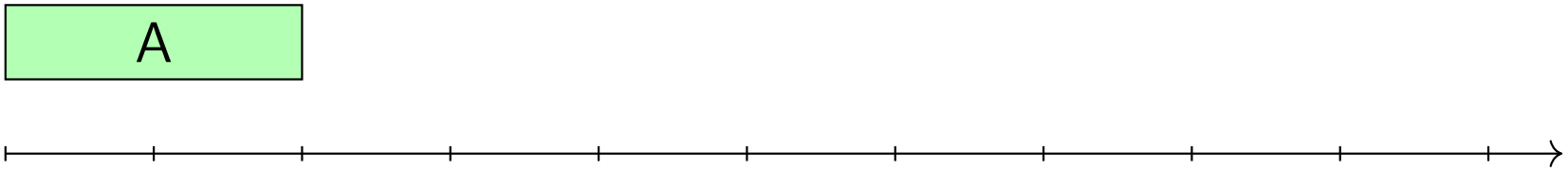
Concrètement: le multi-thread sur un seul processeur

Exemple (Thread A)

```
int a = 0; int b = 0;

while(1){
  a++; b++;

}
```

Exemple (Thread B)

```
doing();
other();

things();
```



- Exécution entrelacée
- Sauvegarde et restauration du *contexte d'exécution*

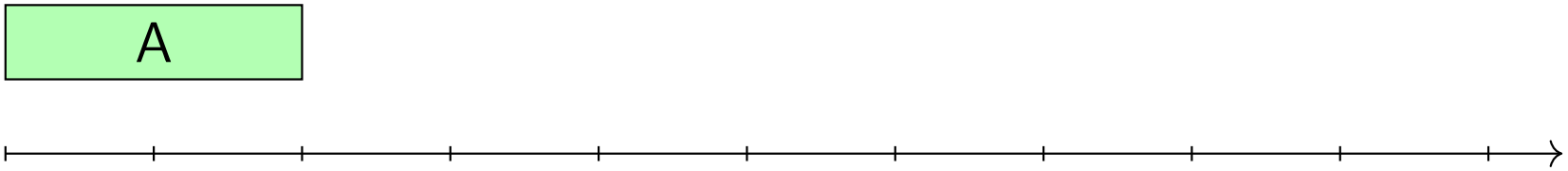
Concrètement: le multi-thread sur un seul processeur

Exemple (Thread A)

```
int a = 0; int b = 0;

while(1){
  a++; b++;

}
```

Exemple (Thread B)

```
doing();
other();

things();
```



- Exécution entrelacée
- Sauvegarde et restauration du *contexte d'exécution*

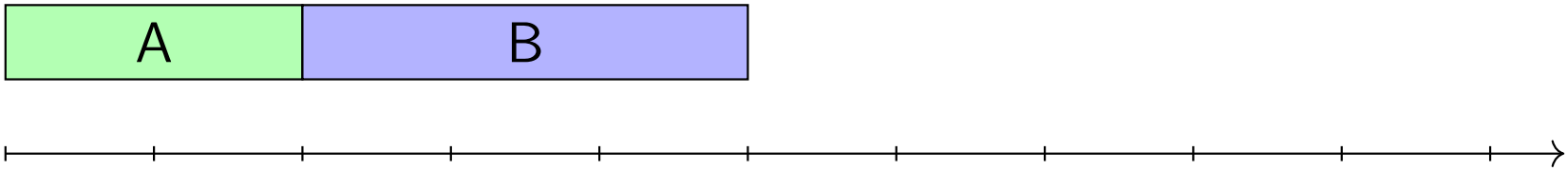
Concrètement: le multi-thread sur un seul processeur

Exemple (Thread A)

```
int a = 0; int b = 0;

while(1){
  a++; b++;

}
```

Exemple (Thread B)

```
doing();
other();

things();
```



- Exécution entrelacée
- Sauvegarde et restauration du *contexte d'exécution*

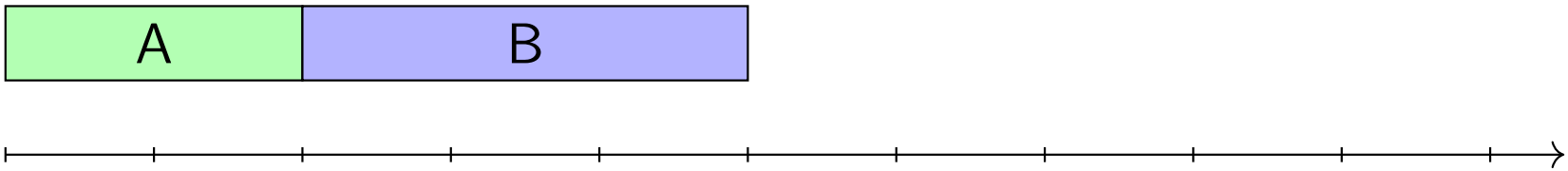
Concrètement: le multi-thread sur un seul processeur

Exemple (Thread A)

```
int a = 0; int b = 0;

while(1){
  a++; b++;

}
```

Exemple (Thread B)

```
doing();
other();

things();
```



- Exécution entrelacée
- Sauvegarde et restauration du *contexte d'exécution*

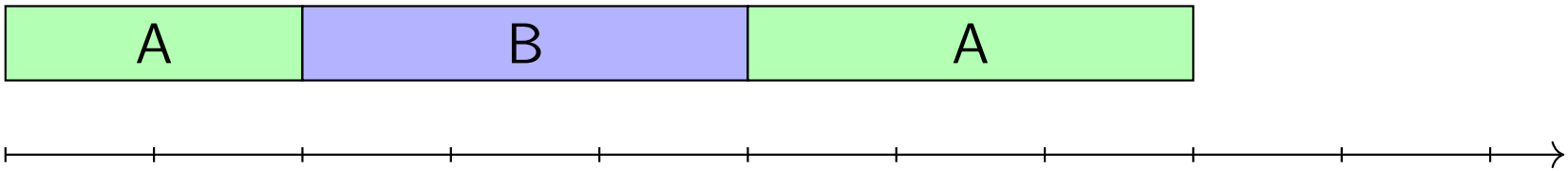
Concrètement: le multi-thread sur un seul processeur

Exemple (Thread A)

```
int a = 0; int b = 0;

while(1){
  a++; b++;

}
```

Exemple (Thread B)

```
doing();
other();

things();
```



- Exécution entrelacée
- Sauvegarde et restauration du *contexte d'exécution*

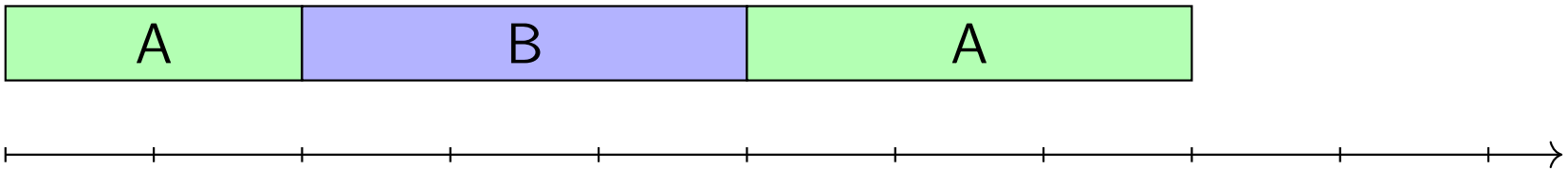
Concrètement: le multi-thread sur un seul processeur

Exemple (Thread A)

```
int a = 0; int b = 0;

while(1){
   a++ ; b++;

}
```

Exemple (Thread B)

```
doing();
other();

things();
```



- Exécution entrelacée
- Sauvegarde et restauration du *contexte d'exécution*

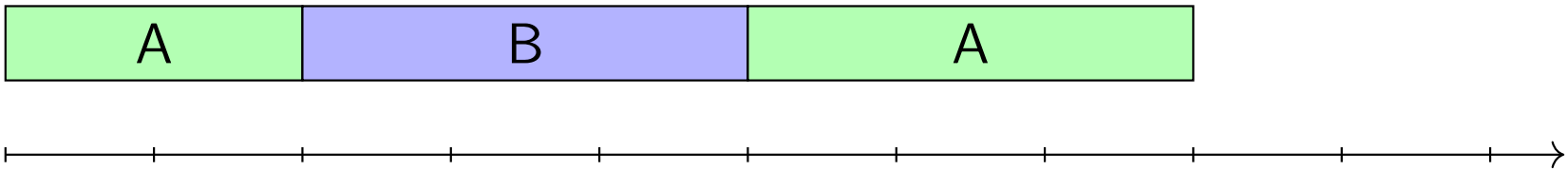
Concrètement: le multi-thread sur un seul processeur

Exemple (Thread A)

```
int a = 0; int b = 0;

while(1){
  a++; b++;

}
```

Exemple (Thread B)

```
doing();
other();

things();
```



- Exécution entrelacée
- Sauvegarde et restauration du *contexte d'exécution*

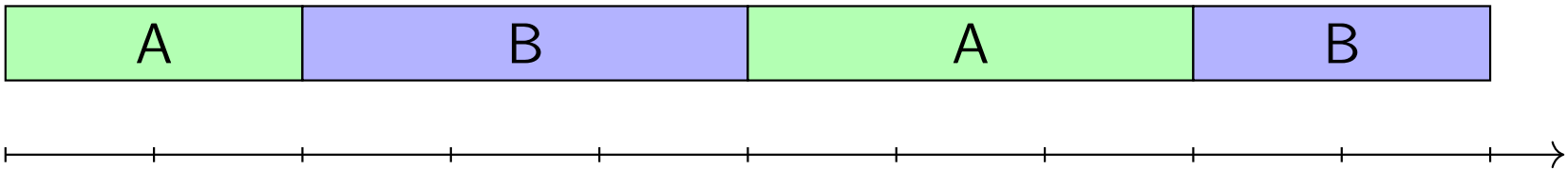
Concrètement: le multi-thread sur un seul processeur

Exemple (Thread A)

```
int a = 0; int b = 0;

while(1){
  a++; b++;

}
```

Exemple (Thread B)

```
doing();
other();

things();
```



- Exécution entrelacée
- Sauvegarde et restauration du *contexte d'exécution*
- Multiprocesseur: similaire, sauf que plusieurs tâches sont simultanément en exécution.

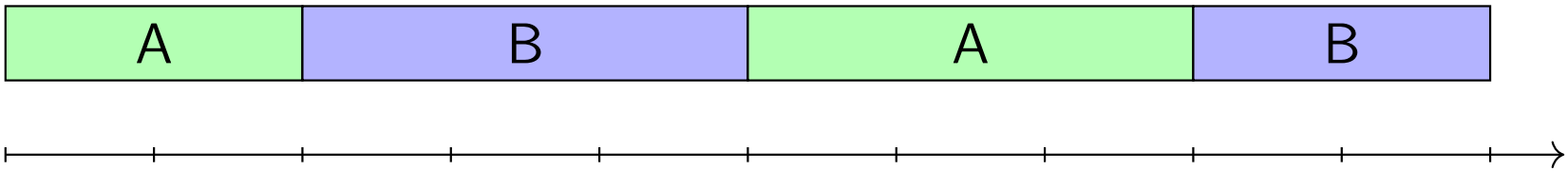
Concrètement: le multi-thread sur un seul processeur

Exemple (Thread A)

```
int a = 0; int b = 0;

while(1){
  a++; b++;

}
```

Exemple (Thread B)

```
doing();
other();

things();
```



- Quand changer de thread?

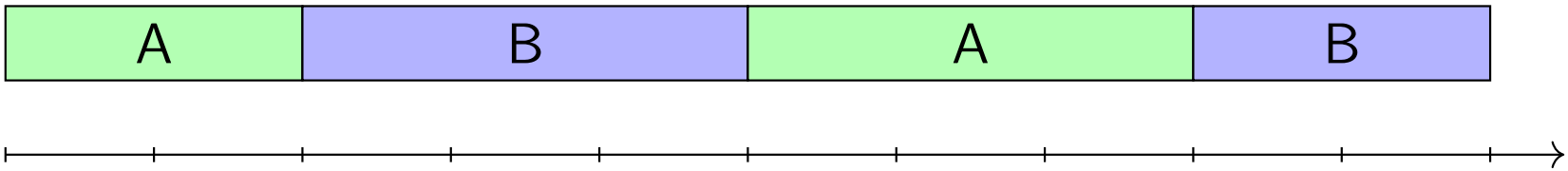
Concrètement: le multi-thread sur un seul processeur

Exemple (Thread A)

```
int a = 0; int b = 0;
yield();
while(1){
  a++; b++;
  yield();
}
```

Exemple (Thread B)

```
doing();
other();
yield();
things();
```



- Quand changer de thread?
  - **multi-threading coopératif**: points de préemptions explicites

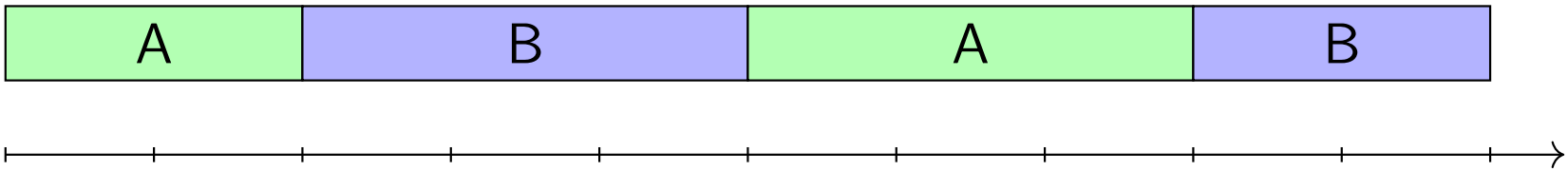
Concrètement: le multi-thread sur un seul processeur

Exemple (Thread A)

```
int a = 0; int b = 0;

while(1){
  a++; b++;

}
```

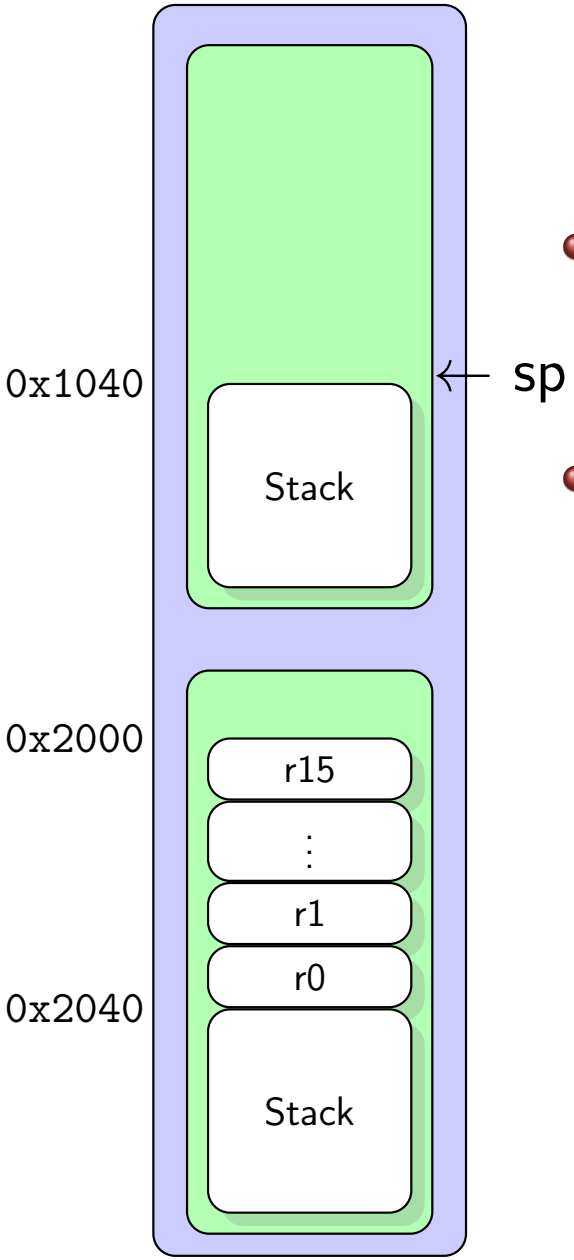
Exemple (Thread B)

```
doing();
other();

things();
```



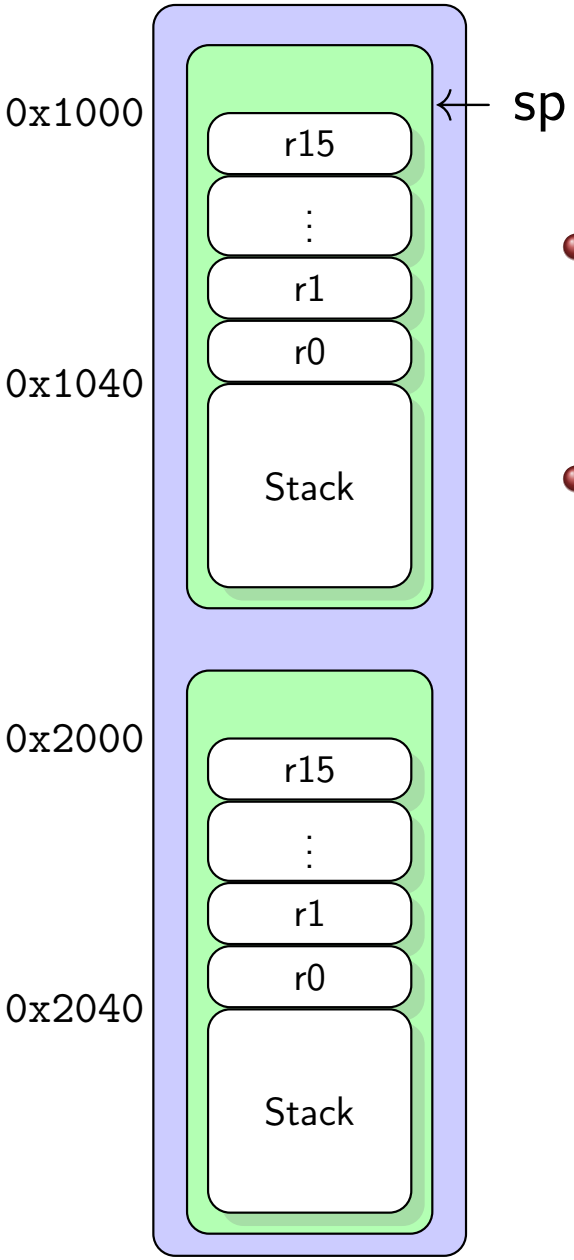
- Quand changer de thread?
  - **multi-threading coopératif**: points de préemptions explicites
  - **multi-threading preemptif**: le hardware et l'OS coupent automatiquement entre deux instructions

Implantation du changement de contexte



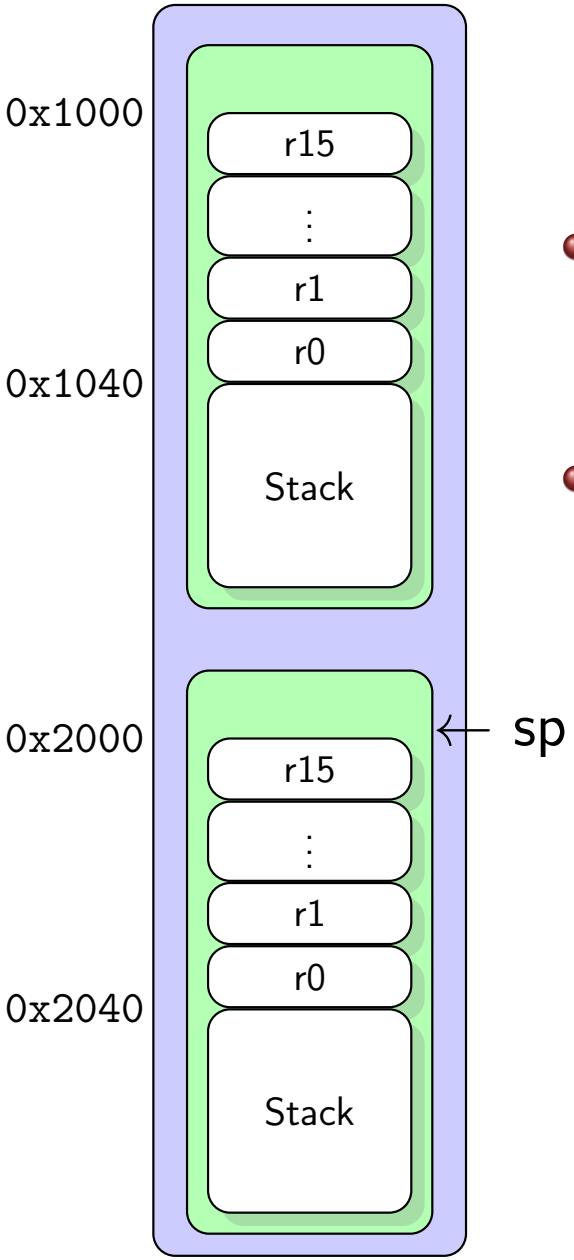
- L'état d'un calcul séquentiel (d'un thread), aussi appellé *contexte*, est représenté par l'état de la pile et la valeur des registres.
- Pour changer de thread:

Implantation du changement de contexte



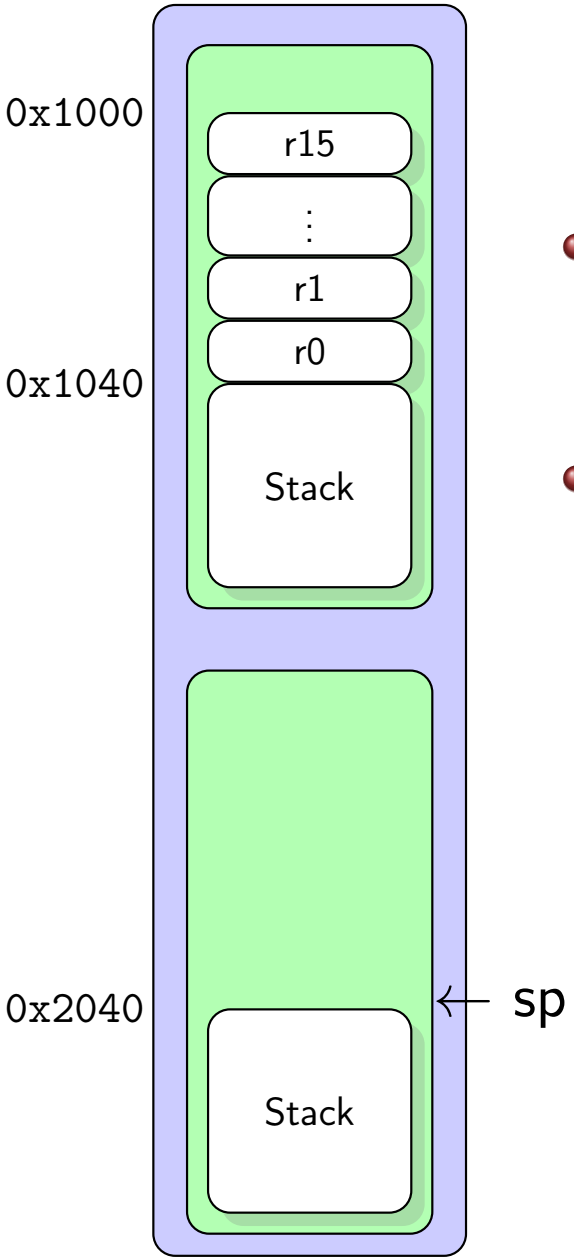
- L'état d'un calcul séquentiel (d'un thread), aussi appellé *contexte*, est représenté par l'état de la pile et la valeur des registres.
- Pour changer de thread:
  1. On sauvegarde la valeur des registres (e.g. au sommet de la pile)

Implantation du changement de contexte



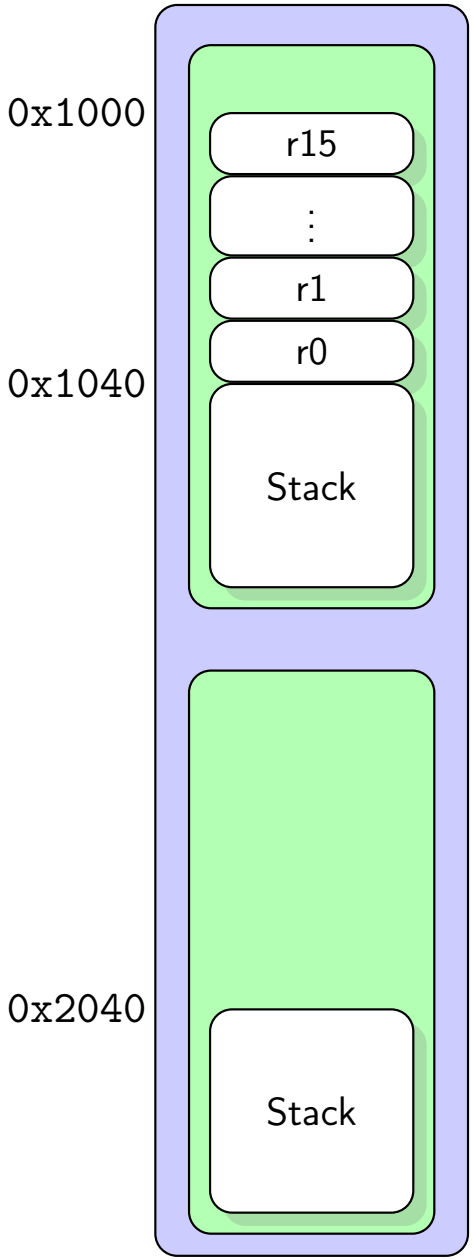
- L'état d'un calcul séquentiel (d'un thread), aussi appellé *contexte*, est représenté par l'état de la pile et la valeur des registres.
- Pour changer de thread:
  1. On sauvegarde la valeur des registres (e.g. au sommet de la pile)
  2. On change de pile et on recharge la valeur des registres

Implantation du changement de contexte



- L'état d'un calcul séquentiel (d'un thread), aussi appellé *contexte*, est représenté par l'état de la pile et la valeur des registres.
- Pour changer de thread:
  1. On sauvegarde la valeur des registres (e.g. au sommet de la pile)
  2. On change de pile et on recharge la valeur des registres

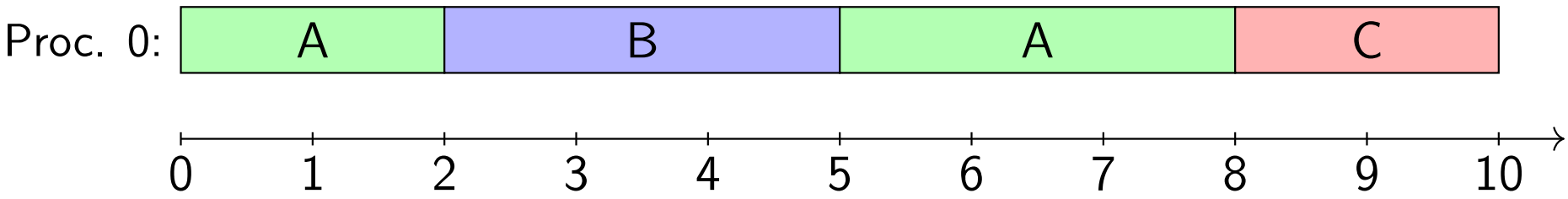
Implantation du changement de contexte



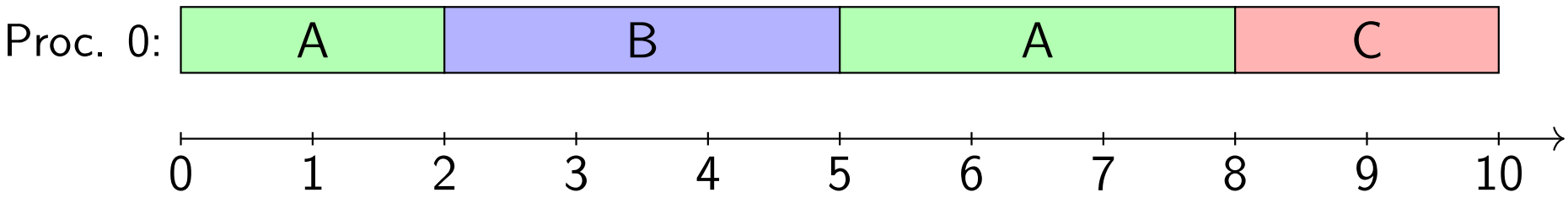
- L'état d'un calcul séquentiel (d'un thread), aussi appellé *contexte*, est représenté par l'état de la pile et la valeur des registres.
- Pour changer de thread:
  1. On sauvegarde la valeur des registres (e.g. au sommet de la pile)
  2. On change de pile et on recharge la valeur des registres
- Les autres sections mémoires (code, variables globales, tas) sont inchangés
  - Elles peuvent être utilisés pour communiquer (communication en mémoire partagée).

Sommaire

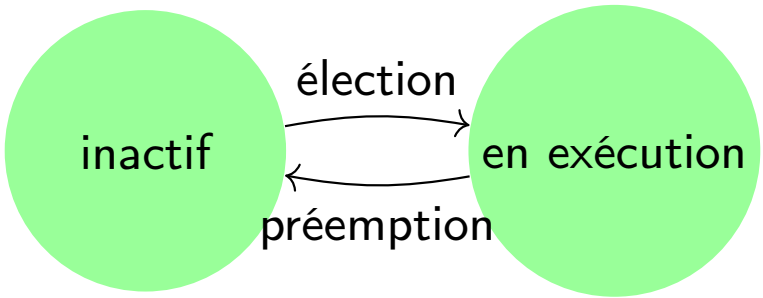
Ordonnancement et statuts des threads (1)



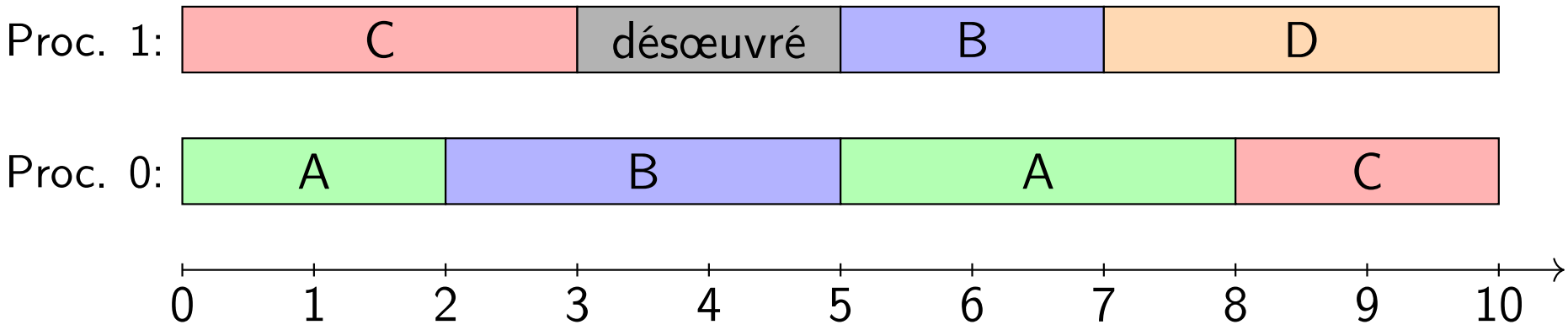
Ordonnancement et statuts des threads (1)



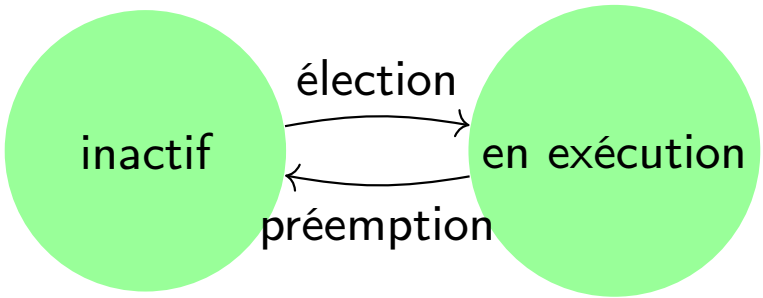
Au moins deux statuts d'un thread: en exécution (actif) ou non en exécution (inactif).



Ordonnancement et statuts des threads (1)



Au moins deux statuts d'un thread: en exécution (actif) ou non en exécution (inactif).



Multi-processeur: plusieurs threads différents simultanément en exécution!

Ordonnancement et statuts des threads (2)

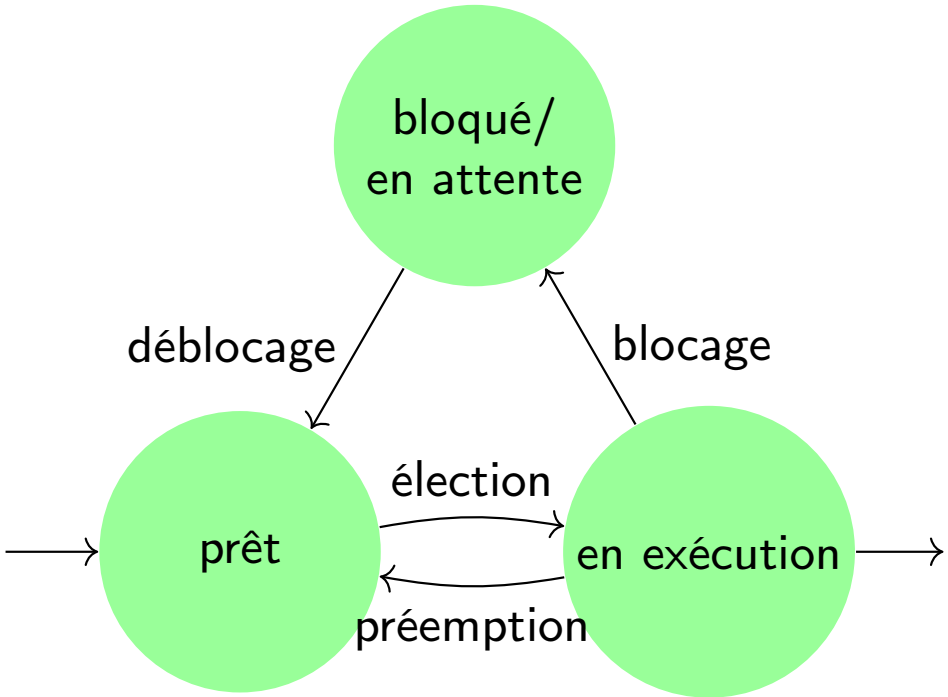
- Que faire quand un thread a besoin d'une entrée du clavier?
- Que faire quand un thread a besoin d'un résultat pas encore fournit par un autre thread?

Ordonnancement et statuts des threads (2)

- Que faire quand un thread a besoin d'une entrée du clavier?
- Que faire quand un thread a besoin d'un résultat pas encore fournit par un autre thread?
- Le thread peut rendre la main (yield) quand il n'a rien à faire

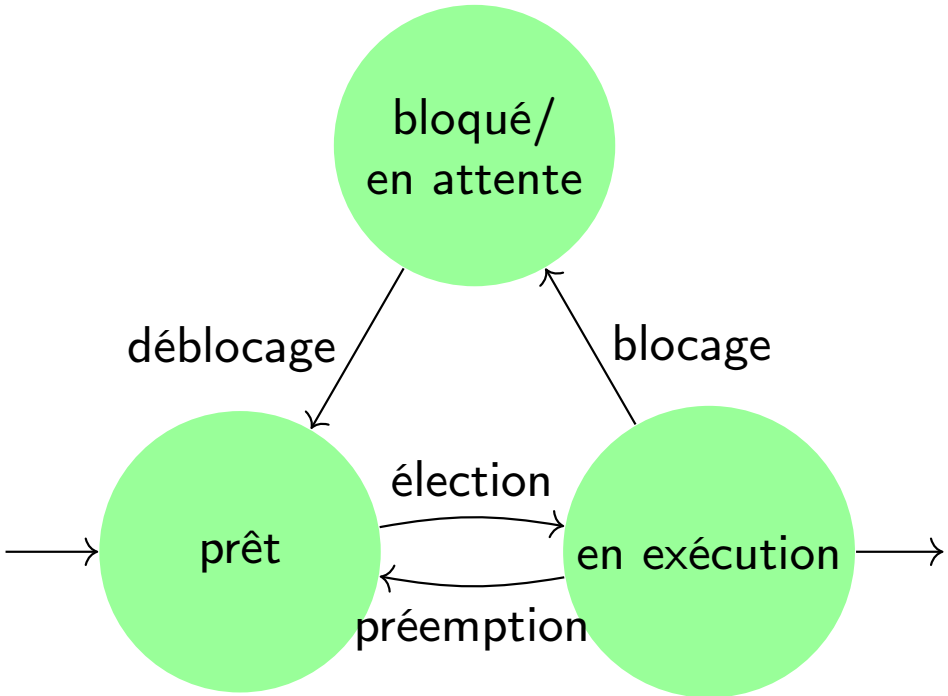
Ordonnancement et statuts des threads (2)

- Que faire quand un thread a besoin d'une entrée du clavier?
- Que faire quand un thread a besoin d'un résultat pas encore fournit par un autre thread?
- Le thread peut rendre la main (yield) quand il n'a rien à faire
- $\Rightarrow$ Encore mieux: le thread peut ne pas s'exécuter du tout
- $\Rightarrow$ Séparation du statut "inactif" en "prêt" et "en attente"



Ordonnancement et statuts des threads (2)

- Que faire quand un thread a besoin d'une entrée du clavier?
- Que faire quand un thread a besoin d'un résultat pas encore fournit par un autre thread?
- Le thread peut rendre la main (yield) quand il n'a rien à faire
- $\Rightarrow$ Encore mieux: le thread peut ne pas s'exécuter du tout
- $\Rightarrow$ Séparation du statut "inactif" en "prêt" et "en attente"



- Autres statuts possibles: init, zombie, ...

Sommaire

Le problème de la cohérence des données

Exemple (Une horloge partagée)

```
char       hour;   // 12
char    minutes;   // 59
```

```
void write_clock(char h, //13
                 char m){//00
  hour = h;
  minutes = m;
}
```

```
void read_clock(char *ph,
                char *pm){
  *ph = hour;
  *pm = minutes;
}
```

Le problème de la cohérence des données

Exemple (Une horloge partagée)

```
char       hour;   // 12
char    minutes;   // 59
```

```
void write_clock(char h, //13
                 char m){//00
  hour = h;
  minutes = m;
}
```

```
void read_clock(char *ph,
                char *pm){
  *ph = hour; // 12
  *pm = minutes;
}
```

Le problème de la cohérence des données

Exemple (Une horloge partagée)

```
char       hour;   // 13
char    minutes;   // 59
```

```
void write_clock(char h, //13
                 char m){//00
  hour = h;
  minutes = m;
}
```

```
void read_clock(char *ph,
                char *pm){
  *ph = hour; // 12
  *pm = minutes;
}
```

Le problème de la cohérence des données

Exemple (Une horloge partagée)

```
char       hour;   // 13
char    minutes;   // 00
```

```
void write_clock(char h, //13
                 char m){//00
  hour = h;
  minutes = m;
}
```

```
void read_clock(char *ph,
                char *pm){
  *ph = hour; // 12
  *pm = minutes;
}
```

Le problème de la cohérence des données

Exemple (Une horloge partagée)

```
char       hour;   // 13
char    minutes;   // 00
```

```
void write_clock(char h, //13
                 char m){//00
  hour = h;
  minutes = m;
}
```

```
void read_clock(char *ph,
                char *pm){
  *ph = hour; // 12
  *pm = minutes; // 00
}
```

Le problème de la cohérence des données

Exemple (Une horloge partagée)

```
char       hour;   // 13
char    minutes;   // 00
```

```
void write_clock(char h, //13
                 char m){//00
  hour = h;
  minutes = m;
}
```

```
void read_clock(char *ph,
                char *pm){
  *ph = hour; // 12
  *pm = minutes; // 00
}
```

Atomicité (linéarizabilité)

- Informellement:
  f est atomique par rapport à g si f apparait s'exécuter d'un seul coup sans être interrompue par l'exécution de g.

Le problème de la cohérence des données

Exemple (Une horloge partagée)

```
char       hour;   // 13
char    minutes;   // 00

void write_clock(char h, //13
                 char m){//00
  hour = h;
  minutes = m;
}

void read_clock(char *ph,
                char *pm){
  *ph = hour; // 12
  *pm = minutes; // 00
}
```

Atomicité (linéarizabilité)

- Plus formellement:
  Deux fonctions f et g sont atomiques si l'exécution entrelacée de f et g est équivalente à une exécution séquentielle de f puis g ou de g puis f

Implémentations de l'atomicité : primitives matérielles

Exemple (Une horloge partagée)

char hour; char minutes;

```
void write_clock(char h,
                 char m){
  hour = h;
  minutes = m;
}
```

```
void read_clock(char *ph,
                char *pm){
  *ph = hour;
  *pm = minutes;
}
```

Implémentations de l'atomicité : primitives matérielles

Exemple (Une horloge partagée)

char hour; char minutes;

```
void write_clock(char h,
                 char m){
  // pas de yield: la
  // fonction est atomique
  hour = h;
  minutes = m;
}
```

```
void read_clock(char *ph,
                char *pm){
  // pas de yield: la
  // fonction est atomique
  *ph = hour;
  *pm = minutes;
}
```

Avec des threads coopératifs

- On contrôle les points de préemption avec les appels à `yield()` (ou aux fonctions qui l'emploient)
- L'exécution entre deux appels à `yield()` est atomique.

Implémentations de l'atomicité : primitives matérielles

Exemple (Une horloge partagée)

volatile short clock;

```
void write_clock(char h,
                 char m){
  short c = (h << 8) | m;
  clock = c;
}
```

```
void read_clock(char *ph,
                char *pm){
  short c = clock;
  *ph = c >> 8;
  *pm = c & 255;
}
```

Avec des threads préemptifs (monoprocesseur)

- Le processeur garantit que les préemptions se font entre deux instructions.
- Attention:
  - Un statement C = plusieurs instructions assembleurs (e.g. `i++;`)
  - Le compilateur peut réordonner les instructions.

Implémentations de l'atomicité : primitives matérielles

Exemple (Une horloge partagée)

volatile short clock;

```
void write_clock(char h,
                 char m){
  short c = (h << 8) | m;
  clock = c;
}
```

```
void read_clock(char *ph,
                char *pm){
  short c = clock;
  *ph = c >> 8;
  *pm = c & 255;
}
```

Avec des threads préemptifs (multiprocesseur)

- Le système garantit l'atomicité des accès du bus
- Attention:
  - Le processeur peut réordonner les écritures et les lectures

Implémentations de l'atomicité : primitives logicielles

Exemple (Une horloge partagée)

char hour; char minutes; mutex_t clock_mutex;

```
void write_clock(char h,
                 char m){
  mutex_lock(&clock_mutex);
  hour = h;
  minutes = m;
  mutex_unlock(&clock_mutex);
}
```

```
void read_clock(char *ph,
                char *pm){
  mutex_lock(&clock_mutex);
  *ph = hour;
  *pm = minutes;
  mutex_unlock(&clock_mutex);
}
```

Avec le support de l'OS

- Utilisation de primitives de synchronisation fournies par l'OS
- Comment fonctionnent-elles?

Sommaire

Synchronisation: définition

Définition (Synchronisation)

Mécanismes permettant de *coordonner* l'exécution de plusieurs threads en *bloquant* leur exécution à des points de programmes précis.

Synchronisation: définition

Définition (Synchronisation)

Mécanismes permettant de *coordonner* l'exécution de plusieurs threads en *bloquant* leur exécution à des points de programmes précis.

Bloquer = changer le *statut des tâches* à bloqué
Libérer/Débloquer = changer le *statut des tâches* à prêt

Pourquoi synchroniser?

- Résoudre les problèmes de cohérence mémoire pendant la communication de données (en mémoire partagée)
- Spécifier des dépendences entre traitements:
  - Contrôler l'ordre d'execution des threads
    - Ex: producteur/consommateur: cas des listes pleines et vides, semaphores
  - Exécution de section critique
    - Ex: commandes de peripheriques/du materiel (e.g. s'assurer qu'on n'envoie pas simultanément deux commandes disques contraires)

En général

Régler les problèmes de concurrence sur l'accès à une *ressource (logicielle ou matérielle) partagée*

pthread_mutex_lock(3) - Linux man page

Name

pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock - lock and unlock a mutex

Synopsis

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Comment synchroniser: implantation de primitive (mutex)

```c
typedef struct {
  volatile int tiquet; volatile int panneau;
} mutex_t;

void mutex_lock(mutex_t *m){
  int mon_tiquet = m->tiquet;
  m->tiquet++;
  while(mon_tiquet != panneau){}
}

void mutex_unlock(mutex_t *m){
  m->panneau = m->panneau + 1;
}
```

Comment synchroniser: implantation de primitive (mutex)

```
typedef struct {
  volatile int tiquet; volatile int panneau;
} mutex_t;

void mutex_lock(mutex_t *m){
  int mon_tiquet = m->tiquet;
  m->tiquet++;
  while(mon_tiquet != panneau){}
}

void mutex_unlock(mutex_t *m){
  m->panneau = m->panneau + 1;
}
```

- Que se passe-t-il si le lock est déjà pris?
  - monoprocesseur (coopératif vs préemptif);
  - multiprocesseur

Comment synchroniser: implantation de primitive (mutex)

```
typedef struct {
  volatile int tiquet; volatile int panneau;
} mutex_t;

void mutex_lock(mutex_t *m){
  int mon_tiquet = m->tiquet;
  m->tiquet++;
  while(mon_tiquet != panneau){ yield(); }
}

void mutex_unlock(mutex_t *m){
  m->panneau = m->panneau + 1;
}
```

- Que se passe-t-il si le lock est déjà pris?
  - monoprocesseur (coopératif vs préemptif);
  - multiprocesseur

Comment synchroniser: implantation de primitive (mutex)

```
typedef struct {
  volatile int tiquet; volatile int panneau;
} mutex_t;

void mutex_lock(mutex_t *m){
  int mon_tiquet = m->tiquet;
  m->tiquet++;
  while(mon_tiquet != panneau){ yield(); }
}

void mutex_unlock(mutex_t *m){
  m->panneau = m->panneau + 1;
}
```

- Que se passe-t-il si le lock est déjà pris?
  - monoprocesseur (coopératif vs préemptif);
  - multiprocesseur
- En préemptif, ++ nécessite plusieurs accès mémoires

Comment synchroniser: implantation de primitive (mutex)

```
typedef struct {
  volatile int tiquet; volatile int panneau;
} mutex_t;

void mutex_lock(mutex_t *m){
  int mon_tiquet = fetch_and_add(&m->tiquet);
  while(mon_tiquet != panneau){}
}

void mutex_unlock(mutex_t *m){
  m->panneau = m->panneau + 1;
}
```

- Que se passe-t-il si le lock est déjà pris?
  - monoprocesseur (coopératif vs préemptif);
  - multiprocesseur
- En préemptif, ++ nécessite plusieurs accès mémoires

Comment synchroniser: implantation de primitive (mutex)

```
typedef struct {
  volatile int tiquet; volatile int panneau;
} mutex_t;

void mutex_lock(mutex_t *m){
  int mon_tiquet = fetch_and_add(&m->tiquet);
  while(mon_tiquet != panneau){}
}

void mutex_unlock(mutex_t *m){
  m->panneau = m->panneau + 1;
}
```

- Que se passe-t-il si le lock est déjà pris?
  - monoprocesseur (coopératif vs préemptif);
  - multiprocesseur
- En préemptif, ++ nécessite plusieurs accès mémoires
- Comment éviter de se faire réveiller pour constater qu'il n'y a rien à faire?

Comment synchroniser: implantation de primitive (mutex)

```
typedef struct {
  volatile int tiquet; volatile int panneau;
  thread_queue_t waiting;
} mutex_t;
void mutex_lock(mutex_t *m){
  int mon_tiquet = fetch_and_add(&m->tiquet);
  while(mon_tiquet != panneau){
    add_to_queue(m->waiting, thread_self());
    thread_self()->status = WAITING;
    yield();
  }
}

void mutex_unlock(mutex_t *m){
  thread_t th = remove_head(m->waiting);
  th->status = READY;
  m->panneau = m->panneau + 1;
}
```

- Que se passe-t-il si le lock est déjà pris?
  - monoprocesseur (coopératif vs préemptif);
  - multiprocesseur
- En préemptif, ++ nécessite plusieurs accès mémoires
- Comment éviter de se faire réveiller pour constater qu'il n'y a rien à faire?

Synchronisation résumé

- La synchronisation sert à coordonner l'exécution de threads
- Implantation des mécanismes de synchronisation à l'intérieur d'un OS (mémoire partagée).
- Problèmes non vus:
  - Famine et équité, deadlocks, inversion de priorité...

Sommaire

Communication inter-thread et processus: motivation

- Le découpage d'un travail en threads et/ou processus implique une communication entre les threads et processus.
- Pour les threads: on peut communiquer par mémoire partagée
  - Efficace, mais sujet à erreur
  - L'utilisation de mécanisme de communication de plus haut niveau est souhaitable
- Pour les processus: on interdit aux processus de communiquer par mémoire partagée
  - $\Rightarrow$ C'est à l'OS d'établir la communication
- Dans tous les cas:
  - Le mécanisme de passage de message est implanté en utilisant de la mémoire partagée et de la synchronisation.

Linux Programmer's Manual for `pipe`, `read`, `write`

Name

`pipe`, `pipe2` - create pipe

Synopsis

```
int pipe(int pipefd[2]);
```

Linux Programmer's Manual for `pipe`, `read`, `write`

Name

`pipe`, `pipe2` - create pipe

Synopsis

```
int pipe(int pipefd[2]);
```

Name

`write` - write to a file descriptor

Synopsis

```
ssize_t write(int fd, const void *buf, size_t count);
```

Linux Programmer's Manual for `pipe`, `read`, `write`

Name

`pipe`, `pipe2` - create pipe

Synopsis

```
int pipe(int pipefd[2]);
```

Name

`write` - write to a file descriptor

Synopsis

```
ssize_t write(int fd, const void *buf, size_t count);
```

Name

`read` - read from a file descriptor

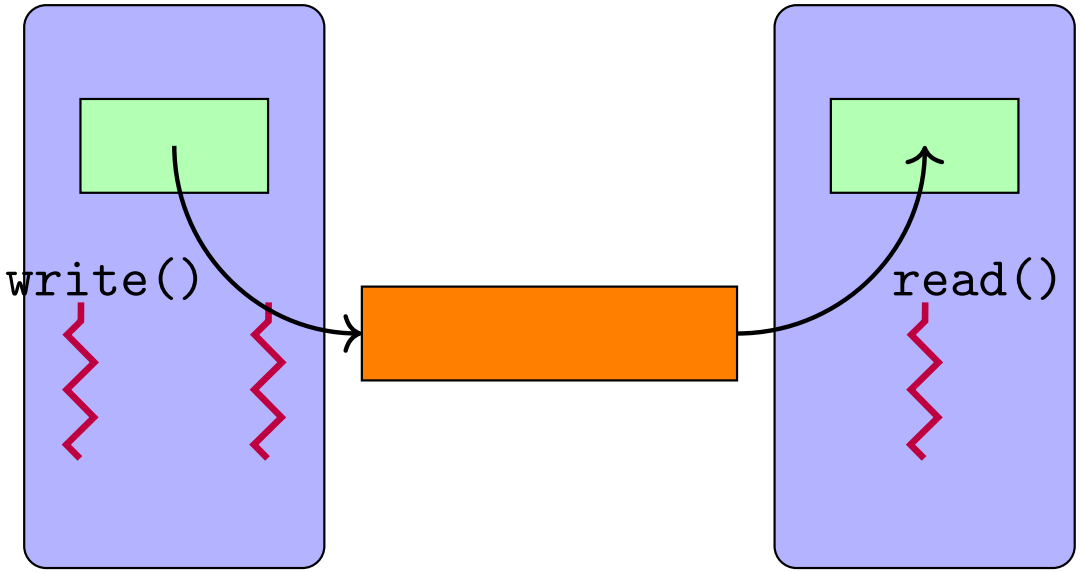
Synopsis

```
ssize_t read(int fd, void *buf, size_t count);
```

Communication inter-processus (ou inter-thread) : le pipe

Exemple (Writer)

```
write("Hel");
write("lo ");
write("wo");
write("rl");
write("d!");
```



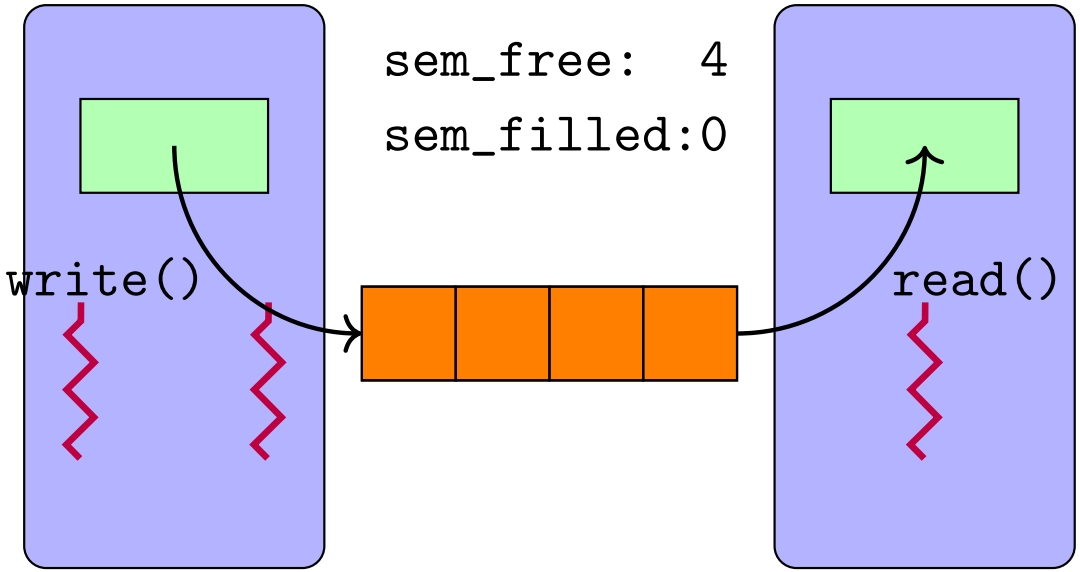
Exemple (Reader)

```
read(&buf);
read(&buf+3);
read(&buf+6);
read(&buf+8);
read(&buf+10);
```

Communication inter-processus (ou inter-thread) : le pipe

Exemple (Writer)

```
write("Hel");
write("lo ");
write("wo");
write("rl");
write("d!");
```



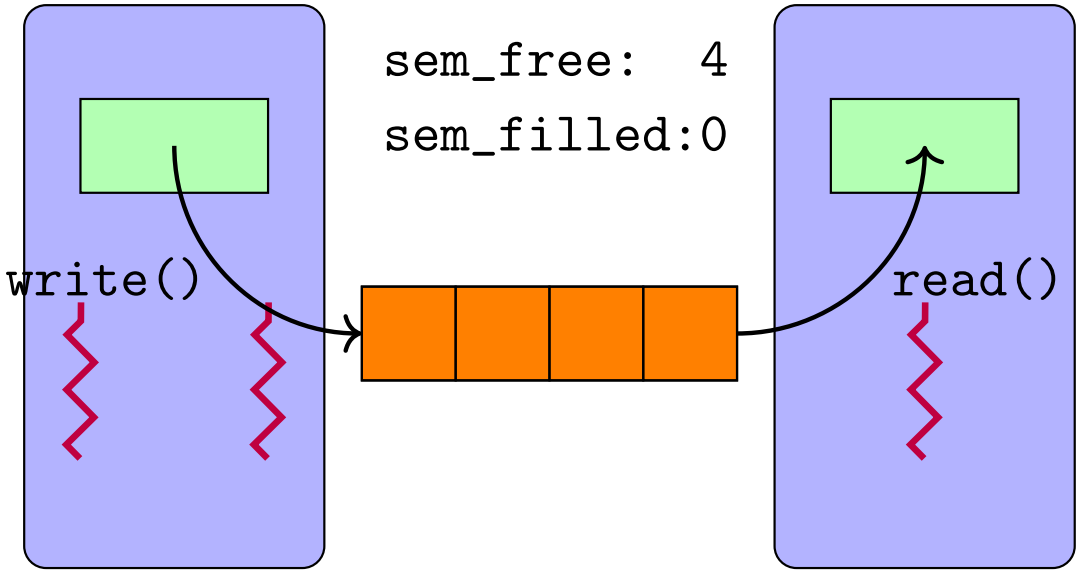
Exemple (Reader)

```
read(&buf);
read(&buf+3);
read(&buf+6);
read(&buf+8);
read(&buf+10);
```

Communication inter-processus (ou inter-thread) : le pipe

Exemple (Writer)

```
write("Hel");
write("lo ");
write("wo");
write("rl");
write("d!");
```



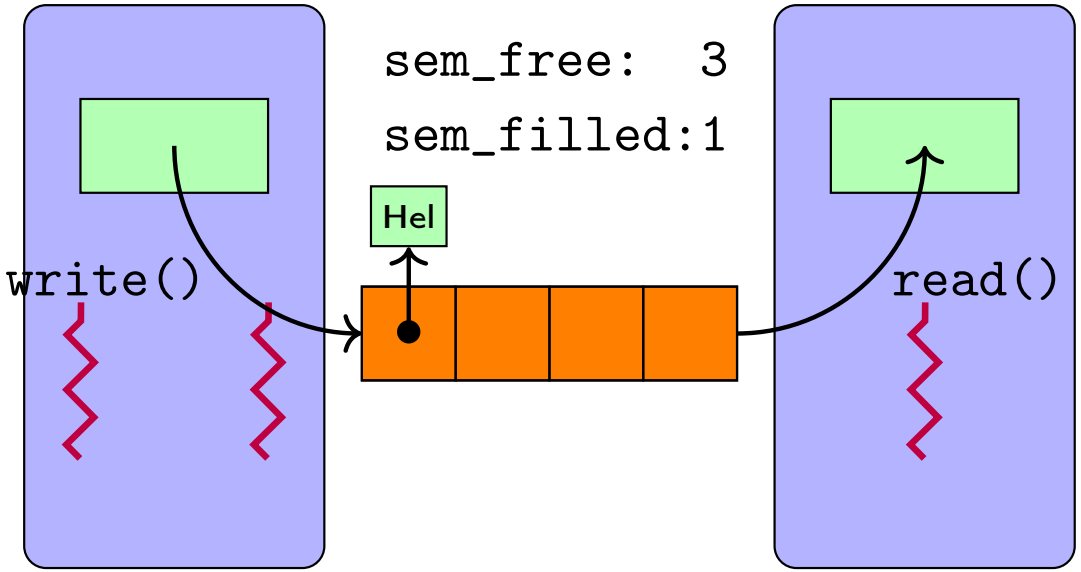
Exemple (Reader)

```
read(&buf);
read(&buf+3);
read(&buf+6);
read(&buf+8);
read(&buf+10);
```

Communication inter-processus (ou inter-thread) : le pipe

Exemple (Writer)

```
write("Hel");
write("lo ");
write("wo");
write("rl");
write("d!");
```



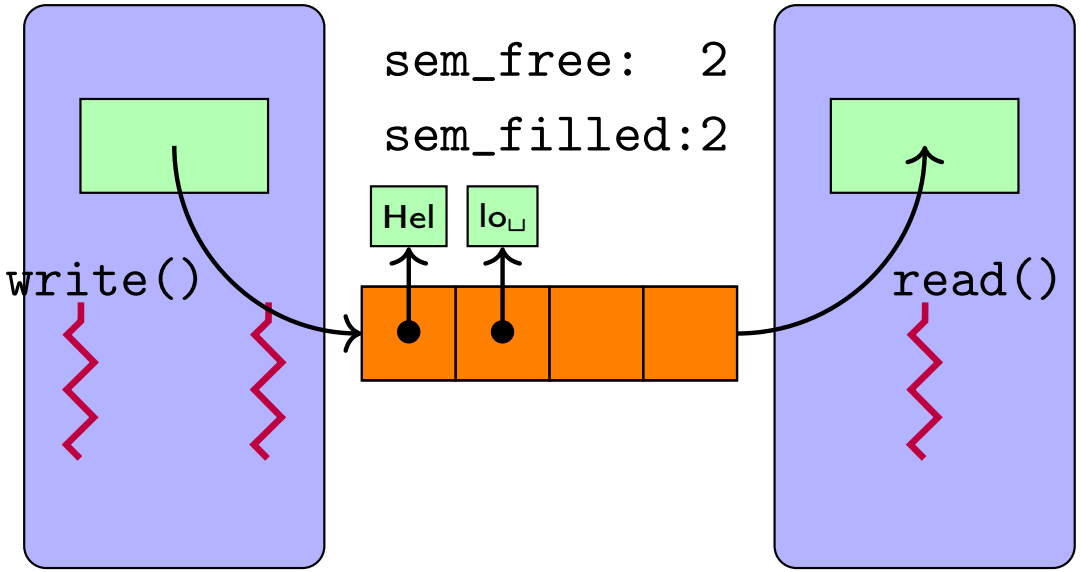
Exemple (Reader)

```
read(&buf);
read(&buf+3);
read(&buf+6);
read(&buf+8);
read(&buf+10);
```

Communication inter-processus (ou inter-thread) : le pipe

Exemple (Writer)

```
write("Hel");
write("lo ");
write("wo");
write("rl");
write("d!");
```



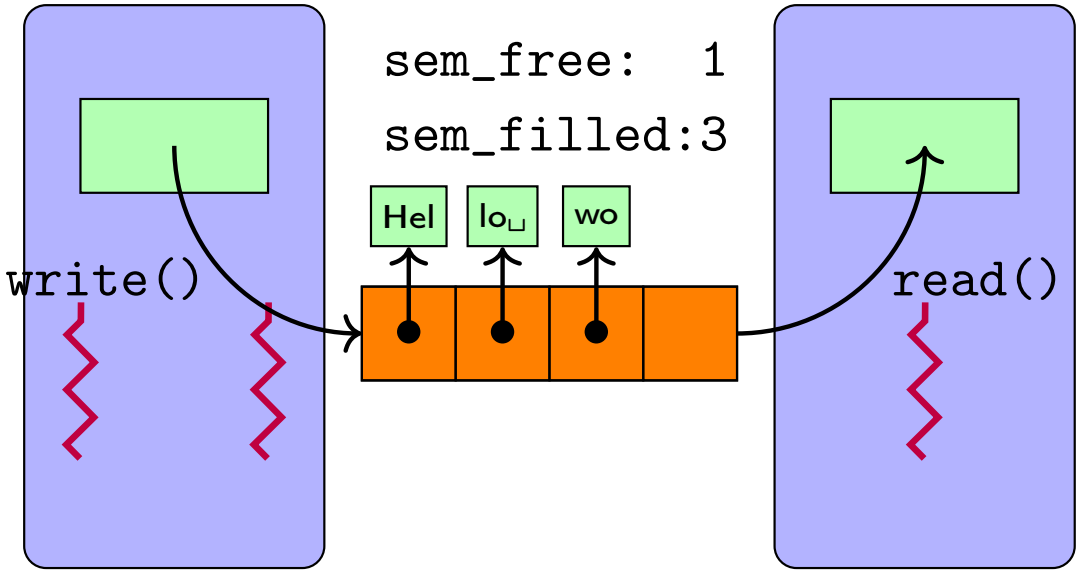
Exemple (Reader)

```
read(&buf);
read(&buf+3);
read(&buf+6);
read(&buf+8);
read(&buf+10);
```

Communication inter-processus (ou inter-thread) : le pipe

Exemple (Writer)

```
write("Hel");
write("lo ");
write("wo");
write("rl");
write("d!");
```



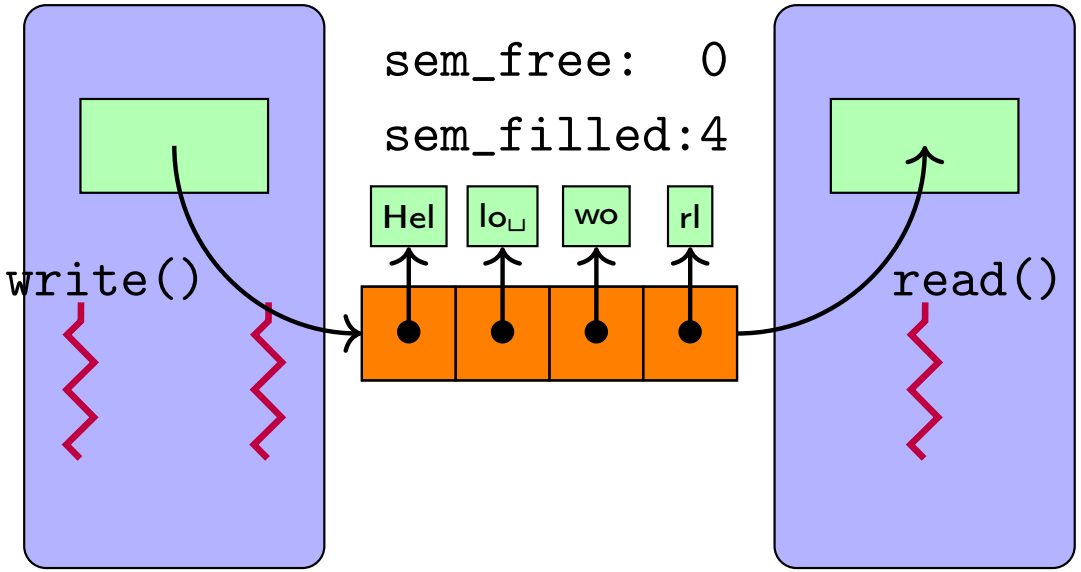
Exemple (Reader)

```
read(&buf);
read(&buf+3);
read(&buf+6);
read(&buf+8);
read(&buf+10);
```

Communication inter-processus (ou inter-thread) : le pipe

Exemple (Writer)

```
write("Hel");
write("lo ");
write("wo");
write("rl");
write("d!");
```



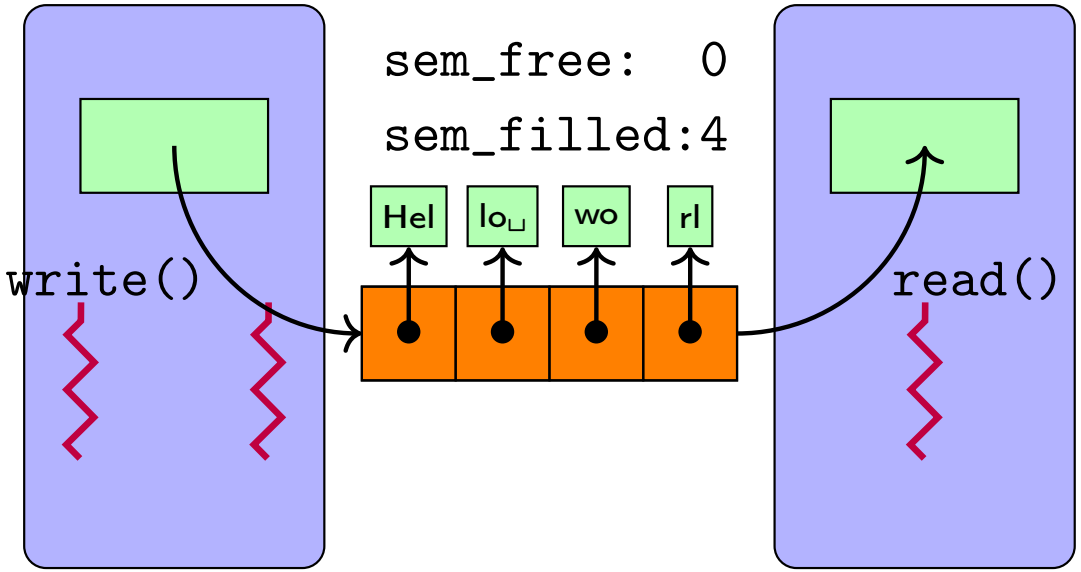
Exemple (Reader)

```
read(&buf);
read(&buf+3);
read(&buf+6);
read(&buf+8);
read(&buf+10);
```

Communication inter-processus (ou inter-thread) : le pipe

Exemple (Writer)

```
write("Hel");
write("lo ");
write("wo");
write("rl");
write("d!");
```



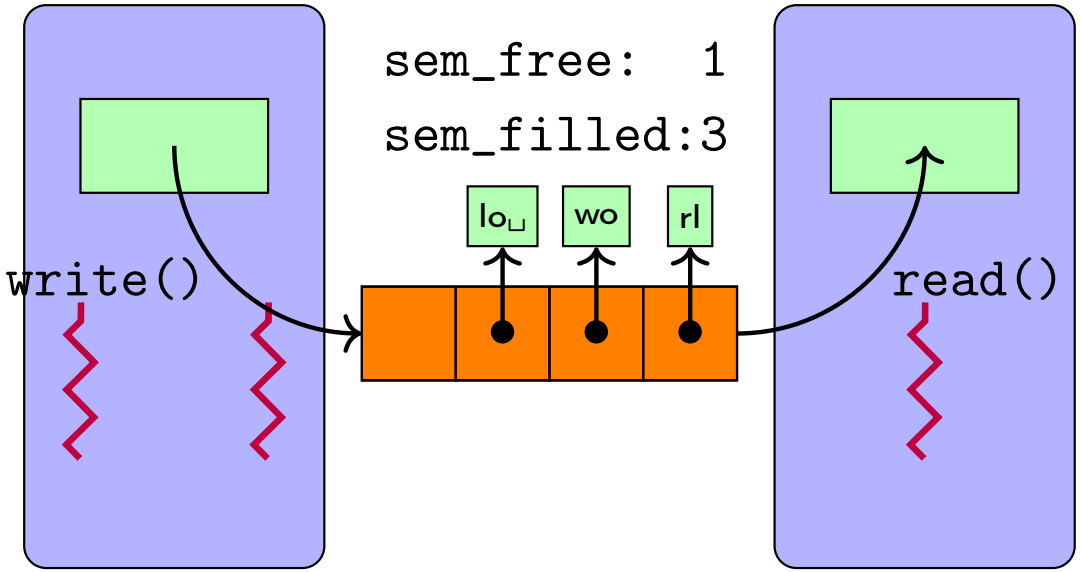
Exemple (Reader)

```
read(&buf);
read(&buf+3);
read(&buf+6);
read(&buf+8);
read(&buf+10);
```

Communication inter-processus (ou inter-thread) : le pipe

Exemple (Writer)

```
write("Hel");
write("lo ");
write("wo");
write("rl");
write("d!");
```



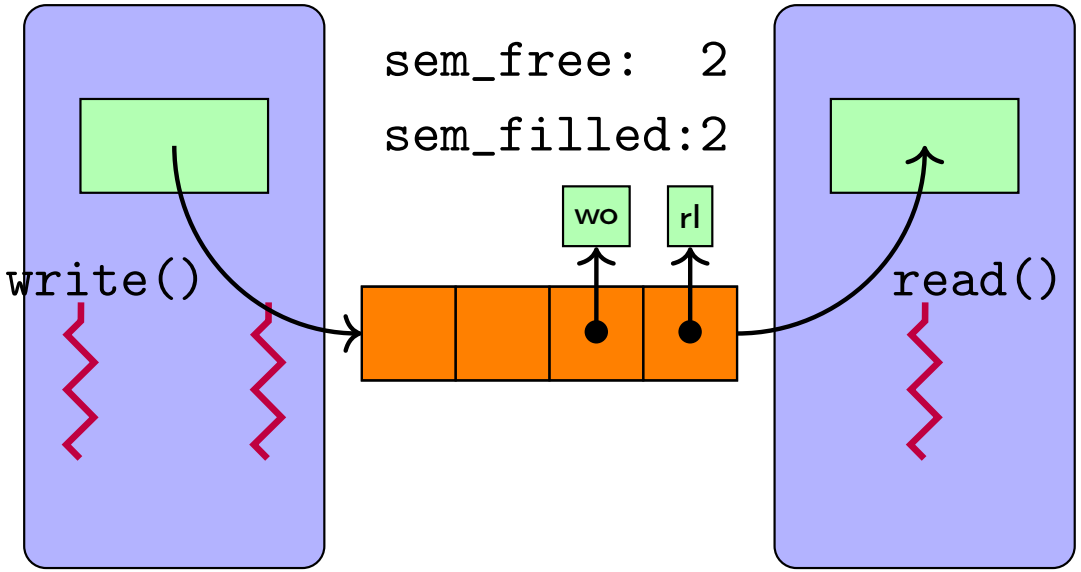
Exemple (Reader)

```
read(&buf);
read(&buf+3);
read(&buf+6);
read(&buf+8);
read(&buf+10);
```

Communication inter-processus (ou inter-thread) : le pipe

Exemple (Writer)

```
write("Hel");
write("lo ");
write("wo");
write("rl");
write("d!");
```



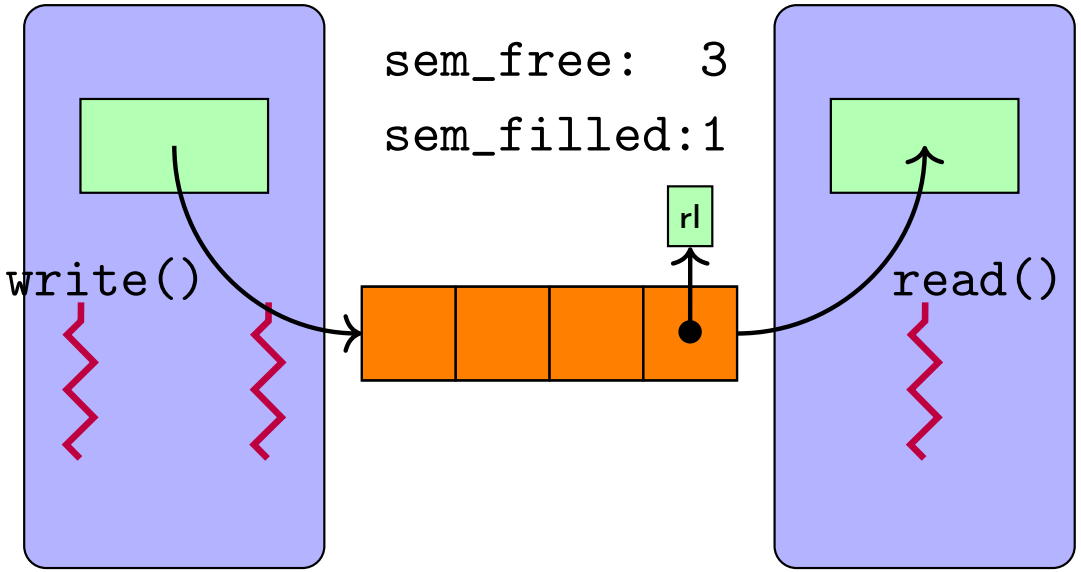
Exemple (Reader)

```
read(&buf);
read(&buf+3);
read(&buf+6);
read(&buf+8);
read(&buf+10);
```

Communication inter-processus (ou inter-thread) : le pipe

Exemple (Writer)

```
write("Hel");
write("lo ");
write("wo");
write("rl");
write("d!");
```



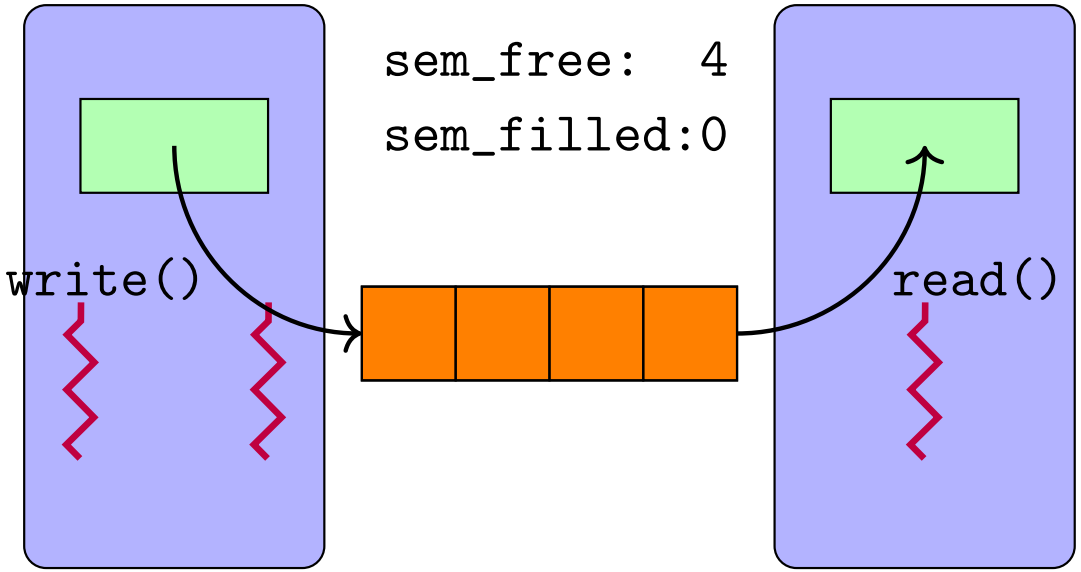
Exemple (Reader)

```
read(&buf);
read(&buf+3);
read(&buf+6);
read(&buf+8);
read(&buf+10);
```

Communication inter-processus (ou inter-thread) : le pipe

Exemple (Writer)

```
write("Hel");
write("lo ");
write("wo");
write("rl");
write("d!");
```



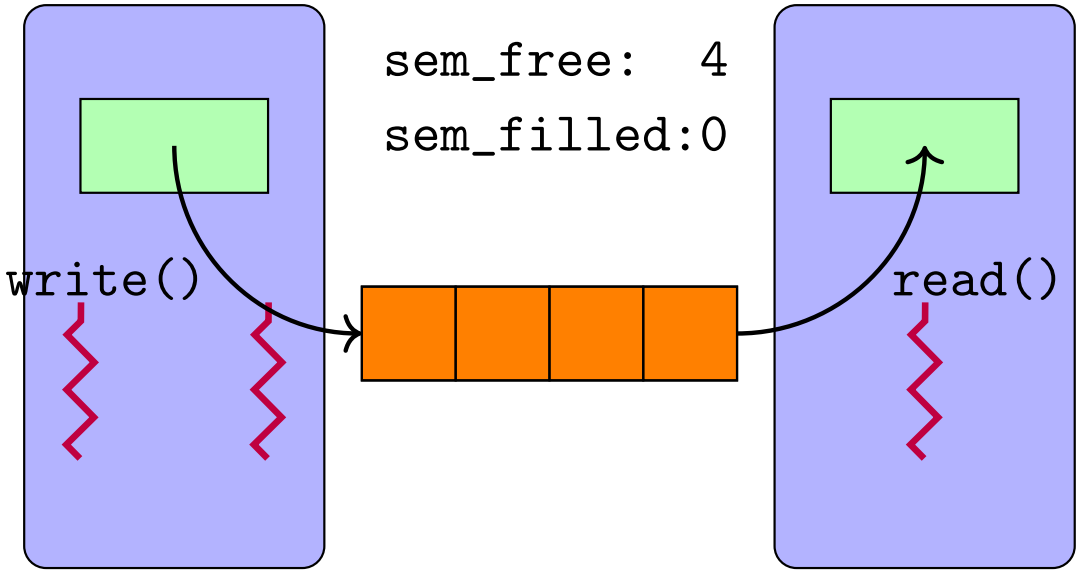
Exemple (Reader)

```
read(&buf);
read(&buf+3);
read(&buf+6);
read(&buf+8);
read(&buf+10);
```

Communication inter-processus (ou inter-thread) : le pipe

Exemple (Writer)

```
write("Hel");
write("lo ");
write("wo");
write("rl");
write("d!");
```



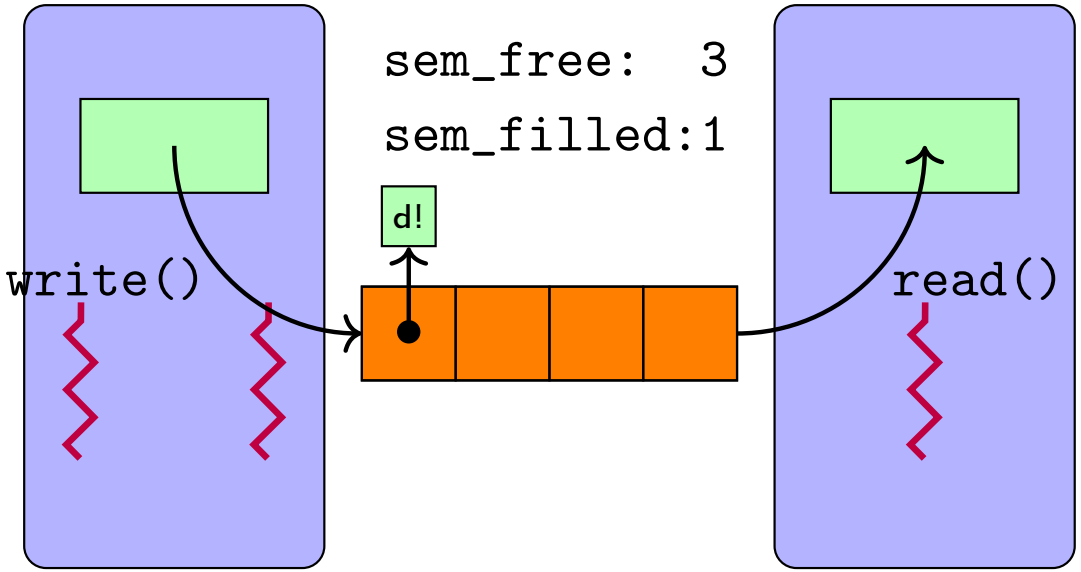
Exemple (Reader)

```
read(&buf);
read(&buf+3);
read(&buf+6);
read(&buf+8);
read(&buf+10);
```

Communication inter-processus (ou inter-thread) : le pipe

Exemple (Writer)

```
write("Hel");
write("lo ");
write("wo");
write("rl");
write("d!");
```



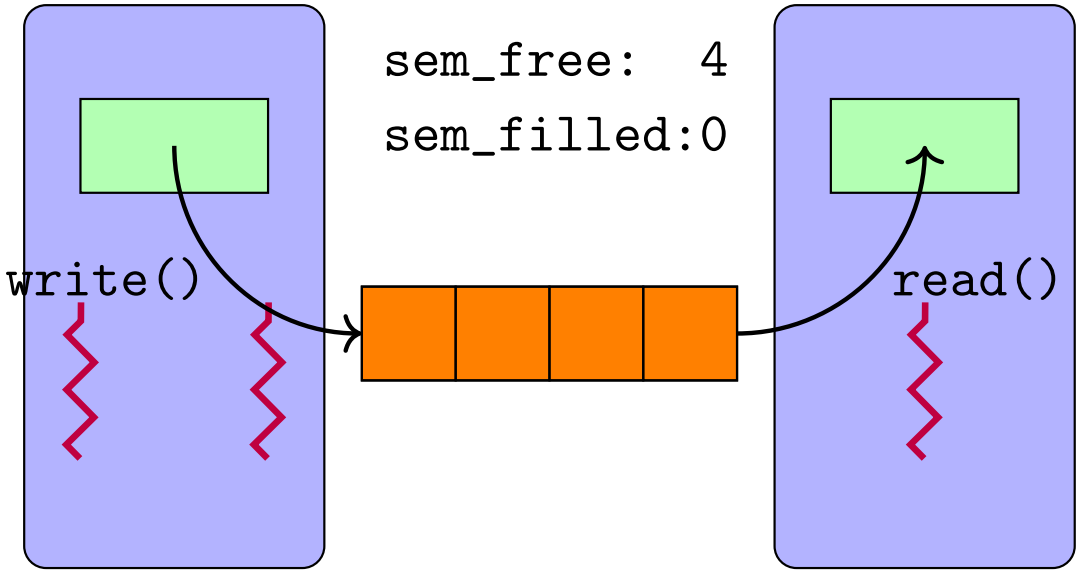
Exemple (Reader)

```
read(&buf);
read(&buf+3);
read(&buf+6);
read(&buf+8);
read(&buf+10);
```

Communication inter-processus (ou inter-thread) : le pipe

Exemple (Writer)

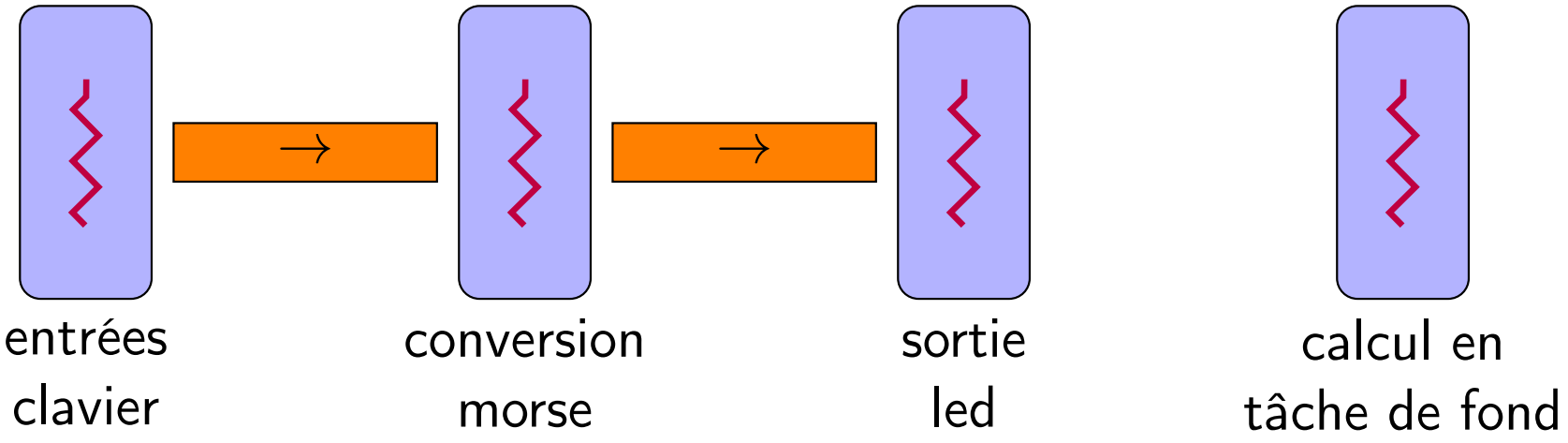
```
write("Hel");
write("lo ");
write("wo");
write("rl");
write("d!");
```



Exemple (Reader)

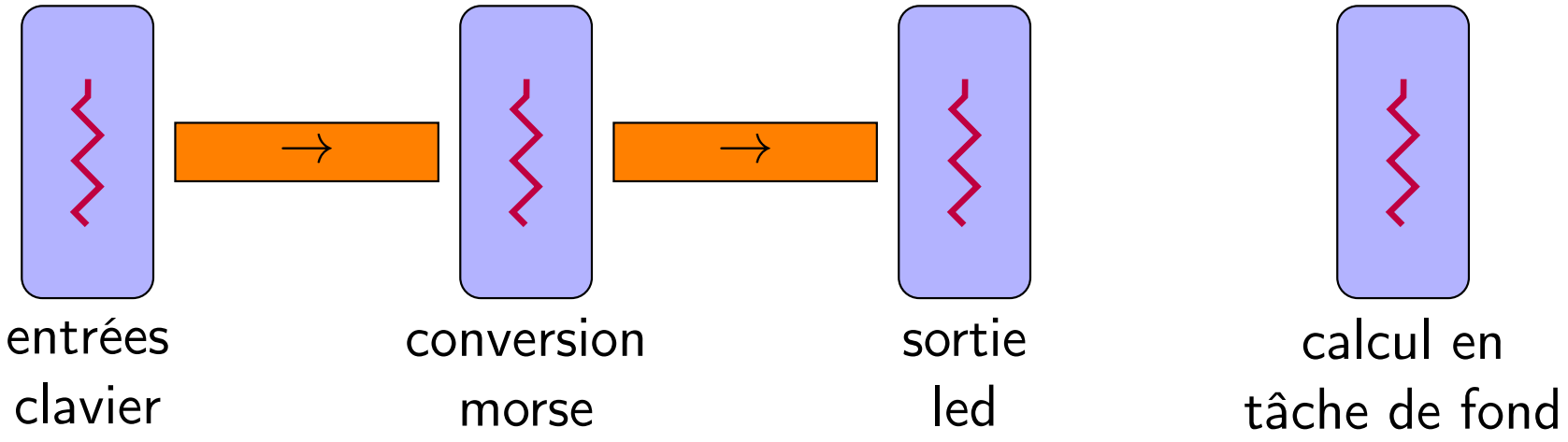
```
read(&buf);
read(&buf+3);
read(&buf+6);
read(&buf+8);
read(&buf+10);
```

Retour sur le problème



- Les threads et les mécanismes de communication simplifient la conception de l'application:
  - Attribution d'une ressource par thread
  - Toutes les synchronisations sont implicites dans l'application
    - Faites par les mécanismes de communication

Retour sur le problème



- Nous avons vu:
  - Qu'est-ce qu'un thread et comment on les réalise
  - Comment on synchronise les threads
  - Comment on les fait communiquer (en utilisant les synchronisations)
- Nous n'avons pas encore vu:
  - Comment on choisi quel thread exécuter (politique d'ordonnancement)
  - À quoi correspondent les boîtes bleues (espaces d'adressage différent)

Sommaire

Résumé du cours

- Un thread consiste en une pile d'exécution et une sauvegarde des registres du processeur.
- Le thread est l'unité d'ordonnancement d'un système (on ordonnance des threads).
- Les 3 états principaux du thread sont *prêt*, *en attente*, ou *en exécution*.
- La synchronisation permet de coordonner des threads en bloquant leur exécution.
- Le processeur et l'OS fournissent différents mécanismes d'accès atomique aux données.
- Communiquer entre thread peut se faire par mémoire partagée ou par passage de message
  - Le passage de message est implanté en utilisant la mémoire partagée
  - La communication repose sur la *synchronisation*