Multithreading et synchronisation

Guillaume Salagnac

Insa de Lyon – Informatique – 3IF Systèmes d'Exploitation

2018–2019

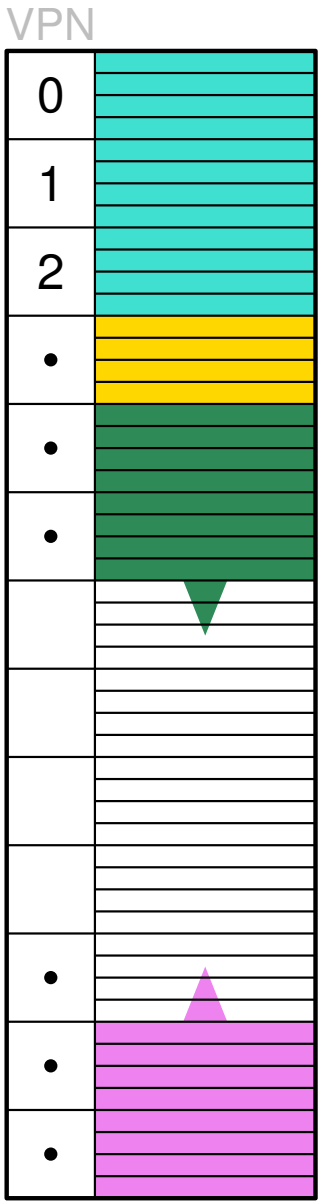
Plan

Rappel : la notion de processus

Définition : processus

«un programme en cours d'exécution»

- isolés les uns des autres
  - en temps : CPU virtuel
  - en espace : mémoire virtuelle
- Process Control Block
  - numéro = PID
  - environnement, répertoire courant, fichiers ouverts...
  - copie des registres CPU
  - vue mémoire = Page Table
- Page Table
  - instructions = .text
  - variables globales = .data
  - tas d'allocation = .heap
  - pile d'exécution = .stack



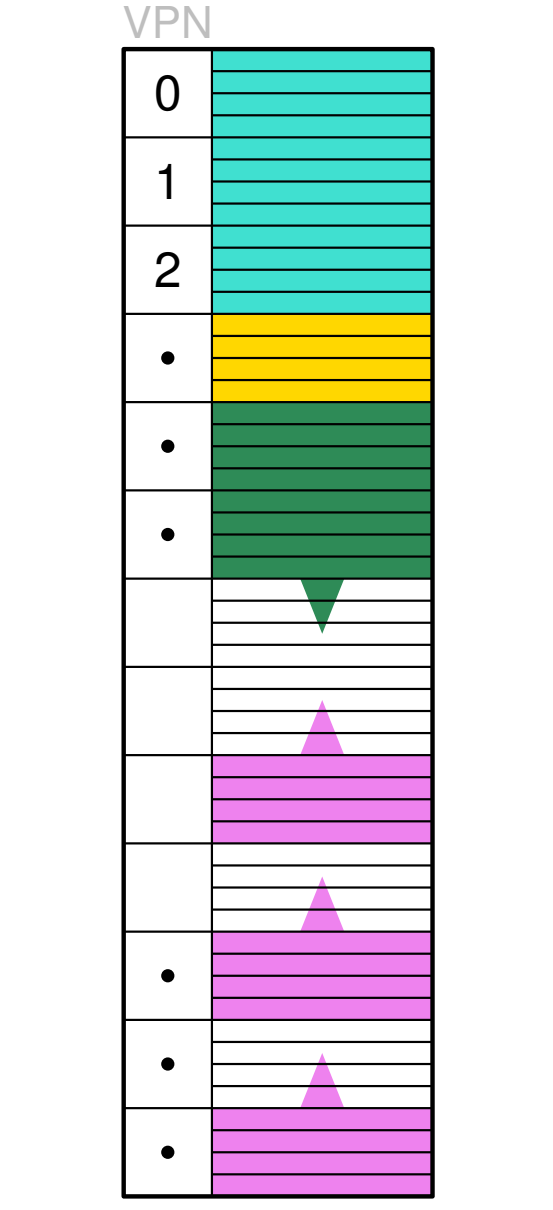
Virtual Address Space

Notion de thread (VF fil d'exécution)

Définition : thread

«une tâche indépendante à l'intérieur d'un processus»

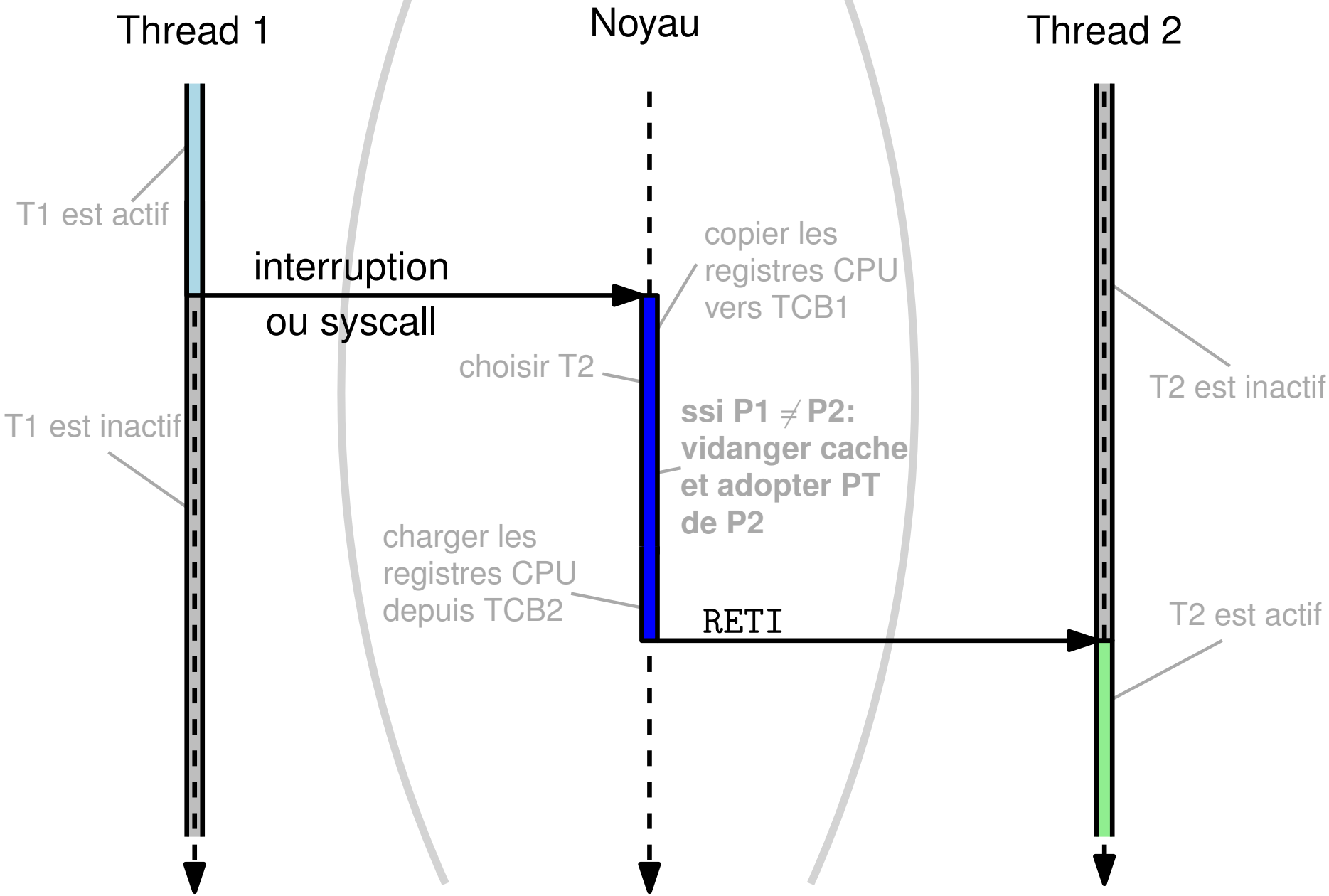
- pourquoi les threads ?
  - profiter de plusieurs CPU
  - faciliter la programmation
- vue mémoire commune
  - pas d'isolation matérielle
  - variables globales partagées
  - tas d'allocation commun
- ordonnancement indépendant
  - un VCPU privé
    TCB = *Thread Control Block*
  - une pile d'exécution privée
  - variables locales privées



Notion de thread : remarques

- parfois appelé «processus léger» mais vision archaïque
  - en vrai : un PCB = une PT et un/plusieurs TCB
  - par ex : `task_struct` et `mm_struct` dans Linux
- un thread ne peut pas vivre en dehors d'un processus
  - besoin d'une vue mémoire
- un processus vivant a toujours au moins un thread
  - *«main thread»* = thread qui exécute `main()`
  - lorsque zéro thread ▶ processus terminé

Rappel : changement de contexte



API POSIX : Threads

```
#include <pthread.h>

/* opaque typedefs */ pthread_t, pthread_attr_t;

// create and start a new thread
int pthread_create(pthread_t *thread,
                   pthread_attr_t *attr,
                   void * (*function) (void *),
                   void *arg);

// terminate the current thread
void pthread_exit(void *retval);
// terminate another thread
int pthread_cancel(pthread_t thread);

// wait for another thread to terminate
int pthread_join(pthread_t thread, void **retvalp);
```

Plan

Accès concurrents à une variable partagée

Variable partagée

```
int var = 5;
```

Thread A

```
{
   ...
   ...
   var = var+1;
   ...
   ...
}
```

Thread B

```
{
   ...
   ...
   var = var-1;
   ...
   ...
}
```

Question : que vaut `var` à la fin de l'exécution ?

- intuition : var==5
- réalité : var==5 ou var==4 ou var==6

Explication : code source $\neq$ instructions du processeur

Variable partagée

```
var: 00000005
```

Thread A

```
...
...
LOAD    REGa←[var]
INCR    REGa
STORE   REGa→[var]
...
...
```

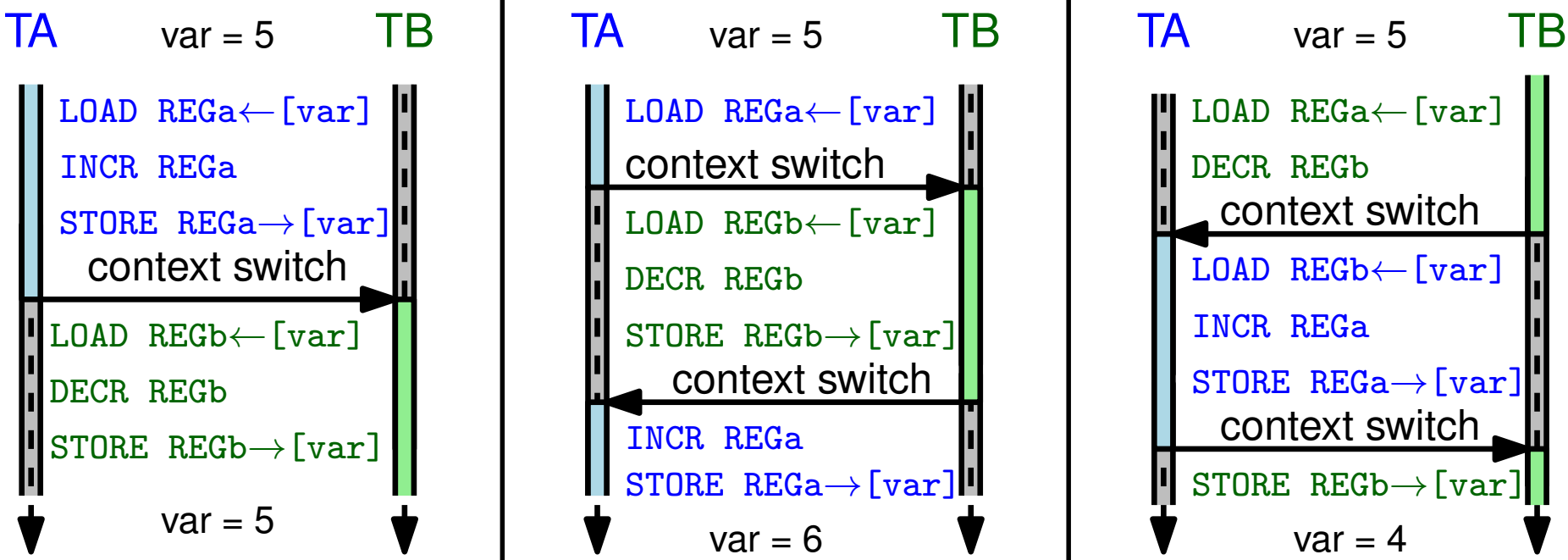
Thread B

```
...
...
LOAD    REGb←[var]
DECR    REGb
STORE   REGb→[var]
...
...
```

Remarque : A et B exécutés sur des (V)CPU distincts

- REGa et REGb (physiquement ou logiquement) distincts

Quelques exécutions possibles



Remarque : 1 CPU ou 2 CPU ▶ problème semblable

Notion de *«race condition»*

VF «situation de concurrence», course critique, accès concurrents

Définition : *race condition*

Situation où le résultat du programme dépend de l'ordre dans lequel sont exécutées les instructions des threads

Remarques

- plusieurs accès concurrents à une ressource partagée
  - variable globale, fichier, réseau, base de données...
  - écriture+écriture = problème
  - écriture+lecture = problème
- concurrence : parallélisme et/ou entrelacement
  - i.e. quand on ne maîtrise pas l'ordre temporel des actions
- risques : corruption de données et/ou crash

- mauvaise nouvelle : très difficile à débugger en pratique
- bonne nouvelle : des protections efficaces existent

Situation de concurrence : exemples

- deux écritures concurrentes = conflit

```
Thread A:   x=10
Thread B:   x=20
```

Question : valeur finale de x ?

- une lecture et une écriture concurrentes = conflit

```
Init:       x=5
Thread A:   x=10
Thread B:   print(x)
```

Question : valeur affichée ?

Précepte : *race condition* = bug

Un programme dans lequel plusieurs tâches peuvent se retrouver en situation de concurrence est un programme incorrect.

Objectif : garantir l'exclusion mutuelle

Définitions

- Action atomique : action au cours de laquelle aucun état intermédiaire n'est visible depuis l'extérieur
- Ressource critique : objet partagé par plusieurs threads et susceptible de subir une race condition
- Section critique : morceau de programme qui accède a une ressource critique

Idée : on veut que chaque section critique soit atomique

Définition : exclusion mutuelle

Interdiction pour plusieurs threads de se trouver simultanément à l'intérieur d'une section critique

Idée : «verrouiller» l'accès à une section critique déjà occupée

Exclusion mutuelle par verrouillage

Variables partagées

```
int var = 5;
lock_t L;
```

Thread A

```
{
  ...
  lock(L);
  var = var+1;
  unlock(L);
  ...
}
```

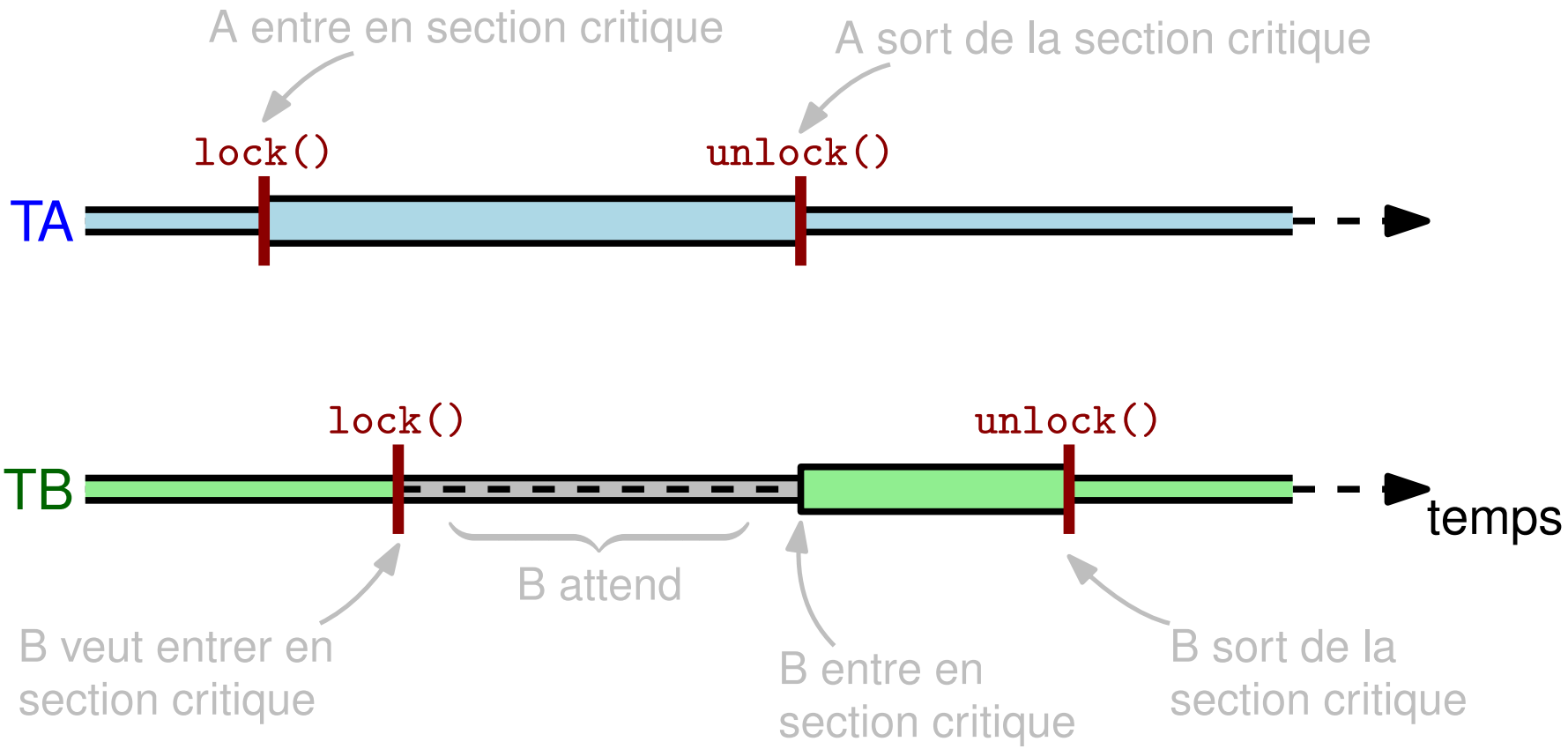
Thread B

```
{
  ...
  lock(L);
  var = var-1;
  unlock(L);
  ...
}
```

On voudrait ces deux méthodes atomiques :

- `lock(L)` pour prendre le verrou L en exclusivité
  - un seule thread peut entrer en section critique
- `unlock(L)` pour relâcher le verrou L
  - permet aux autres threads de le prendre à leur tour

Exclusion mutuelle : illustration



Problème : comment garantir l'exclusion mutuelle ?

Autrement dit : comment implémenter `lock()` et `unlock()` ?

Propriétés souhaitables

- Exclusion mutuelle : à chaque instant, au maximum une seule tâche est en section critique
  - sinon risque de *race condition*
- Progression : si aucune tâche n'est en section critique, alors une tâche exécutant `lock()` ne doit pas se faire bloquer
  - sinon risque de *deadlock*, en VF interblocage
- Équité : aucune tâche ne doit être obligée d'attendre indéfiniment avant de pouvoir entrer en section critique
  - sinon risque de *starvation*, en VF famine, privation
- Généralité : pas d'hypothèses sur les vitesses relatives ou sur le nombre des tâches en présence
  - on veut une solution universelle
- Bonus : implem simple, algo prouvable, exécution efficace...

Solution naïve (et incorrecte)

partagé

```
int turn = 1;
```

Thread A

```
while(1)
{ ...
  while(turn==2)
  {/* attendre */ }
// section critique
  turn = 2;
  ...
}
```

Thread B

```
while(1)
{ ...
  while(turn==1)
  {/* attendre */ }
// section critique
  turn = 1;
  ...
}
```

- Exclusion mutuelle : OK
- Attente active : exécution pas très efficace
- Problème : alternance stricte ▶ progression non garantie

Solution naïve n° 2 (incorrecte aussi)

partagé

```
bool occup = 0;
```

Thread A

```
while(1)
{ ...
  while(occup != 0)
  {/* rien */ }
  occup = 1;
// section critique
  occup = 0;
  ...
}
```

Thread B

```
while(1)
{ ...
  while(occup != 0)
  {/* rien */ }
  occup = 1;
// section critique
  occup = 0;
  ...
}
```

- Progression : OK
- Exclusion mutuelle : non garantie
- Problème : consultation-modification non atomique

Solutions correctes

Masquer les interruptions

- idée : empêcher tout changement de contexte

▶ dangereux, et inapplicable sur machine multiprocesseur

Approche purement logicielle

- idée : programmer avec des instructions atomiques
- autrefois seulement `LOAD` et `STORE` ▶ par ex. algo de Peterson
- de nos jours : `TEST-AND-SET`, `COMPARE-AND-SWAP` ▶ spin-lock

▶ attente active = souvent inefficace à l'exécution

Approche noyau : intégrer synchronisation et ordonnancement

- idée : programmer avec des instructions atomiques
  - mais les cacher dans le noyau (derrière des appels système)

▶ permet de bloquer / réveiller les threads au bon moment

Mutex : définition

Verrou exclusif, ou en VO mutex lock

- objet abstrait = opaque au programmeur
- deux états possibles : libre=unlocked ou pris=locked
- offre deux méthodes atomiques
  - `lock(L)` : si le verrou est libre, le prendre
    sinon, attendre qu'il se libère
  - `unlock(L)` : relâcher le verrou,
    c.à.d. le rendre libre à nouveau

Remarques

- `lock()` et `unlock()` implémentés comme appels système
- threads en attente = état `BLOCKED` dans l'ordonnanceur
  - une file de threads suspendus pour chaque mutex
- invoquer `unlock()` réveille *un* thread suspendu (s'il y en a)
  - attention : ordre de réveil non spécifié

API POSIX : Mutex locks

```
#include <pthread.h>

/* opaque typedefs */ pthread_mutex_t, pthread_mutexattr_t;

// create a new mutex lock
int pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t *mutexattr);

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

// attempt to lock a mutex without blocking
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Exclusion mutuelle : en résumé

Notion de «*race condition*»

- plusieurs accès concurrents à une même variable
- accès non atomique ▶ données incohérentes

Section critique

- morceau de code qu'on veut rendre atomique
- ▶ exécution nécessairement en exclusion mutuelle

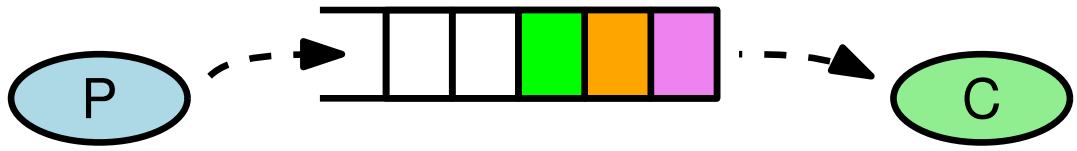
Solution : utiliser un mutex lock

- ```
  lock(L);
  /* section critique */
  unlock(L);
  ```
- nécessite que tous nos threads jouent le jeu
  ```

Plan

Scénario producteur-consommateur : introduction

Deux threads communiquent via une file FIFO partagée



Producteur

```
while(1)
{
  item=produce();
  fifo_put(item);
}
```

Consommateur

```
while(1)
{
  item = fifo_get();
  consume(item);
}
```

Remarques :

- file = tampon circulaire de taille constante
- producteur doit attendre tant que la file est pleine
- consommateur doit attendre tant que la file est vide

Prod.-consomm. : solution naïve (et incorrecte)

partagé

```
item_t buffer[N];
int count=0;
```

Producteur

```
int in = 0;
while(1)
{
  item=produce()
  while(count == N) {}
  buffer[in] = item;
  in = (in+1) % N;
  count = count + 1;
}
```

Consommateur

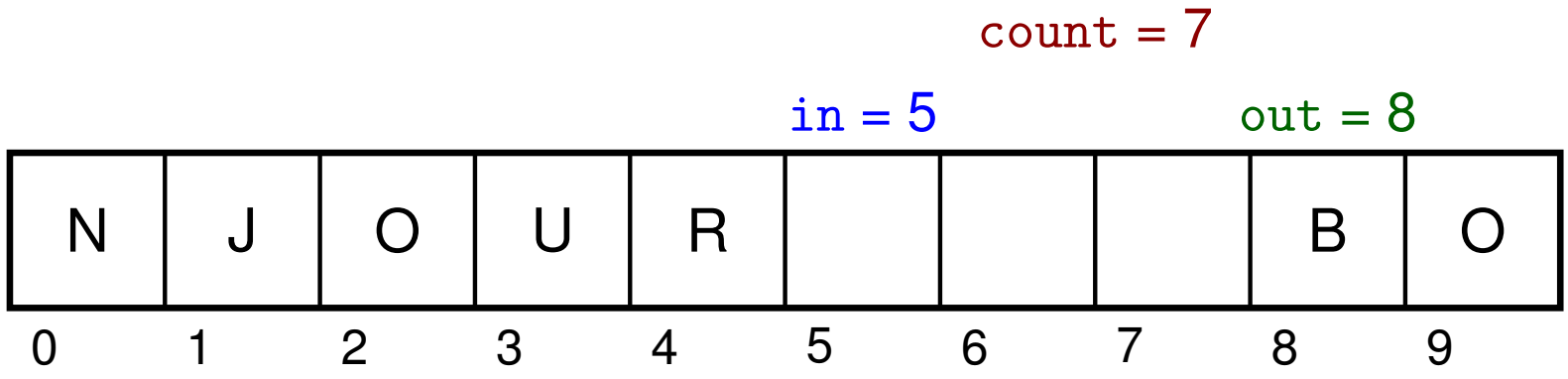
```
int out = 0;
while(1)
{
  while(count == 0) {}
  item = buffer[out];
  out = (out+1) % N;
  count = count - 1;
  consume(item);
}
```

Observation : ce programme a des bugs de synchronisation

- Question : comment corriger le problème ?

Producteur-consommateur : remarques

- `buffer` partagé de taille N (constante) initialement vide
  - buffer circulaire : `x%N` = x modulo N
- fonctions `produce()` et `consume()` non pertinentes
  - supposées «purement séquentielles» AKA inoffensives
- variable partagée `count` pour la synchronisation
  - indique le nombre d'éléments actuellement dans le buffer
- variables `in` et `out` : non partagées
- exemple avec `N`=10 :

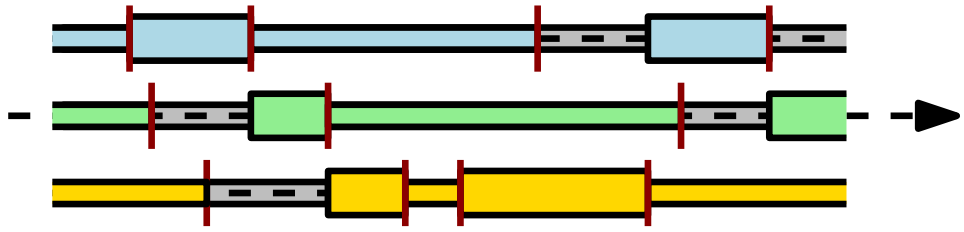


Mauvaise nouvelle

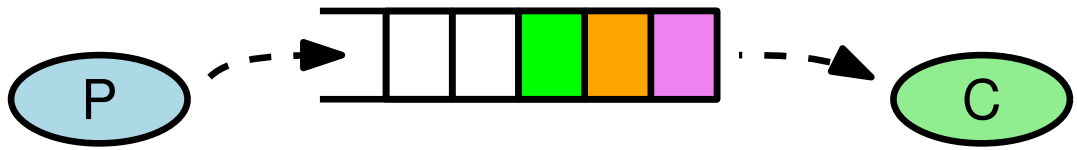
Cette synchro est irréalisable avec seulement `lock()`/`unlock()`

Quelques scénarios classiques de synchronisation

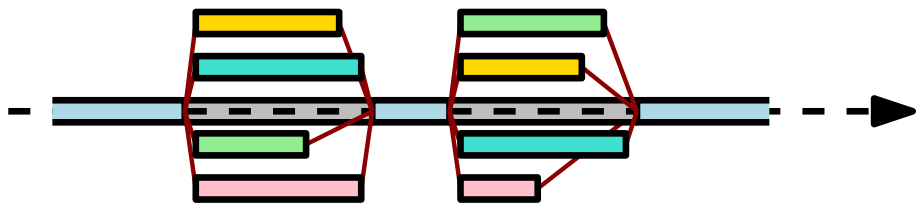
Exclusion mutuelle



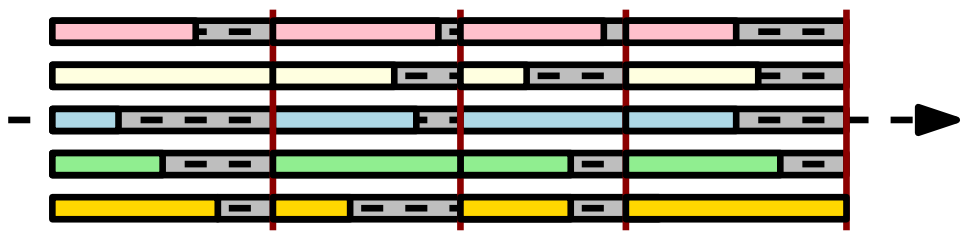
Producteur consommateur, en VO bounded buffer problem



Boucle parallèle, en VO fork-join



Rendez-vous, en VO barrier



Notion de sémaphore (Edsger W. Dijkstra 1930–2002)

- objet abstrait = opaque au programmeur
- contient une variable entière *k*
  - initialisée avec k$\geqslant$0 lors de la création du sémaphore
- contient une file d'attente de threads bloqués
- offre deux méthodes atomiques `P()` et `V()`

P(S)

```
S.k = S.k - 1;
if( S.k < 0 )
{
  /* suspendre le thread
     appelant, et le
     mettre dans la file
     d'attente de S */
}
```

V(S)

```
S.k = S.k + 1;
if( S.k <= 0 )
{
/* réveiller l'un des
   threads de la file
   d'attente de S */
}
```

Sémaphore : remarques

- mécanisme de synchronisation très polyvalent
  - permet de résoudre tous les scénarios ci-dessus

- implem : instructions atomiques dans appel système
- file d'attente = état `BLOCKED` dans l'ordonnanceur
  - attention : ordre de réveil non spécifié

- cas général parfois appelé «sémaphore à compteur»
  - $k \geqslant 0$ ▶ nombre de «jetons disponibles» = k
  - $k \leqslant 0$ ▶ nombre de tâches en attente = $-k$
- «sémaphore binaire» si k initialisée à 1
  - équivalent à un mutex : `P()=lock()` et `V()=unlock()`
- attention : aucun moyen de consulter la valeur de k

- propriétés souhaitables : sûreté, progression, équité, généralité, simplicité, prouvabilité, performance...

Producteur-consommateur avec sémaphores

partagé

```
item_t buffer[N];
sem_t emptyslots=N;
sem_t fullslots=0;
```

Producteur

```
in = 0;
while(1)
{
  item=produce()
  P(emptyslots);
  buffer[in] = item;
  in = (in+1) % N;
  V(fullslots);
}
```

Consommateur

```
out = 0;
while(1)
{
  P(fullslots);
  item = buffer[out];
  out = (out+1) % N;
  V(emptyslots);
  consume(item);
}
```

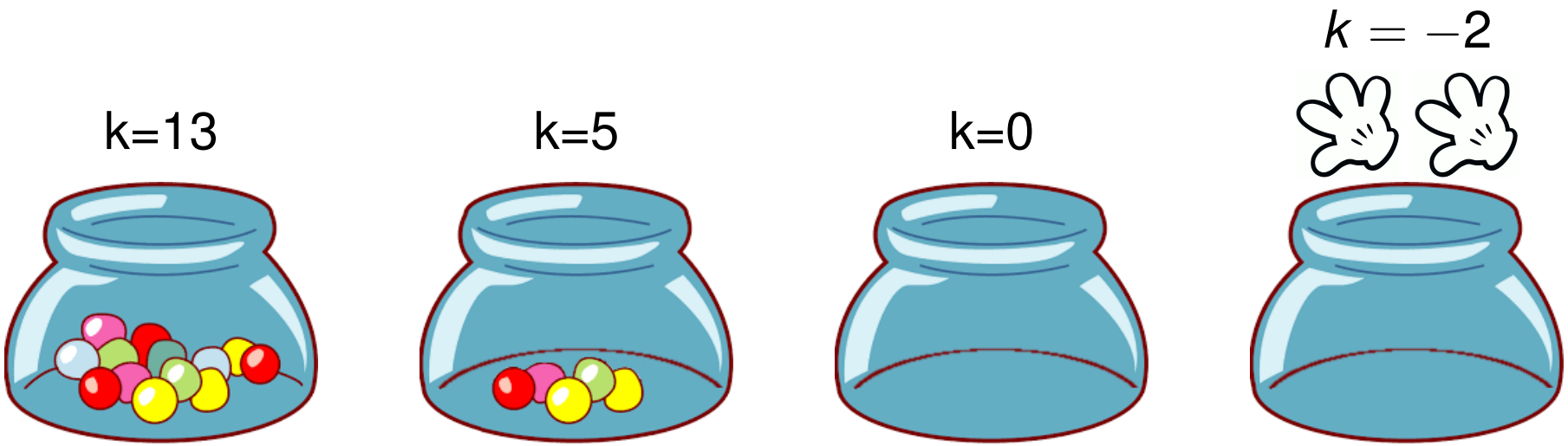
Sémpahores POSIX

```
#include <semaphore.h>
/* opaque typedef */ sem_t;

// semaphore initialization and destruction
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);

// synchronization methods
int sem_wait(sem_t *sem); // wait = P = down
int sem_post(sem_t *sem); // post = V = up = signal
```

Semaphores : intuitions



- `P()` = «essayer de prendre un jeton, me suspendre si aucun dispo»
  - AKA wait, acquire, pend, down
  - un example d'appel bloquant
- `V()` = «ajouter un jeton et peut-être réveiller un autre thread»
  - AKA post, signal, release, post, up
  - un exemple d'appel non-bloquant

Notion de «*deadlock*», en VF interblocage

Définition : interblocage, en VO deadlock

Situation dans laquelle deux (ou plusieurs) tâches concurrentes se retrouvent suspendues car elles s'attendent mutuellement

- ...pour toujours

Exemple :

```
Init: Semaphore X=1, Y=1
Thread A: P(X); P(Y); print("A");
Thread B: P(Y); P(X); print("B");
```

Question : résultat ?

Exemple de trace d'exécution menant à l'interblocage

1. Thread A : P(X)
2. Thread B : P(Y)
3. Thread A : P(Y) ▶ bloque A
4. Thread B : P(X) ▶ bloque B

Plan

Threads et synchronisation : en résumé

Processus VS thread VS mémoire virtuelle

- unité d'ordonnancement = thread
- unité d'isolation mémoire = espace d'adressage virtuel
- un processus = un espace d'adressage + un/plusieurs threads

Exclusion mutuelle AKA mutex

- stratégie permettant d'éviter les *«race conditions»*
- mécanisme : méthodes `lock()` et `unlock()` atomiques

Sémaphore

- mécanisme de synchronisation universel
- `P()` = «essayer de prendre un jeton, me suspendre si aucun dispo»
- `V()` = «ajouter un jeton et peut-être réveiller un autre thread»