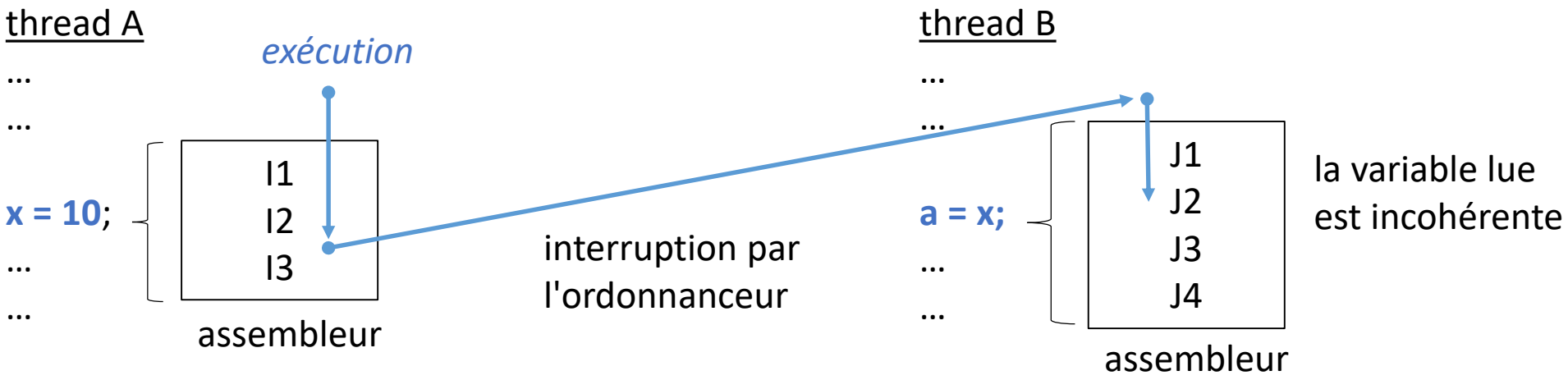
# Séance 7
# Gestion de la concurrence
# Exclusion mutuelle : Sémaphore - Mutex

# Accès concurrent à une donnée partagée

**int x** = 0;  variable globale




⇒ mettre en place un mécanisme d'exclusion mutuelle lors de l'accès à la variable (section critique)

remarque : si les deux threads accèdent en lecture, pas de problème d'accès concurrent

# Exclusion mutuelle : mutex

```
int x = 0;

pthread_mutex_t monmutex = PTHREAD_MUTEX_INITIALIZER;
```

thread A

```
…
pthread_mutex_lock(&monmutex);
x = 10;
pthread_mutex_unlock(&monmutex);
…
```

thread B

```
…
pthread_mutex_lock(&monmutex);
a = x;
pthread_mutex_unlock(&monmutex);
…
```

pthread_mutex_lock : verrouille le mutex ou attend si le mutex est déjà verrouillé
pthread_mutex_unlock : déverrouille le mutex et réveille le thread éventuellement en attente
retournent 0 (OK) ou != 0 (échec)

# Sémaphore : définition

Un mécanisme pour gérer l'accès concurrent à des ressources.

Soit un pool de N ressources identiques et interchangeables (N imprimantes, N fichiers, N variables, …).

un sémaphore S : une variable = nombre de ressources (du pool) disponibles (varie de 0 à N).

# Sémaphores : opérations

Initialisation

S = n;  n dans [0, N]

Soustraction: *-p => on veut prendre p ressources*

SUB (p)
SI S >= p  s'*il y a suffisamment de ressources*
  S = S – p
SINON
  bloquer le processus ou thread en attendant que la valeur du sémaphore permette l'opération

Addition : *+q => on libère q ressources*

ADD (q)
S = S + q
réveiller le premier processus ou thread en attente pouvant être satisfait

# Sémaphore : utilisation

Si opérations SUB et ADD atomiques => utilisation pour synchroniser l'accès aux ressources

sémaphore S

| processus ou thread A | processus ou thread B |
|---|---|
| … | … |
| … | … |
| SUB(2) sur S | SUB(3) sur S |
| *accès à 2 ressources* | *accès à 3 ressources* |
| ADD(2) sur S | ADD(3) sur S |
| … | … |

remarque : un mutex = un sémaphore valant 0 ou 1

# Sémaphore : implémentation sous Linux

| | |
|---|---|
| sem_t monsem; | un sémaphore |
| sem_init (&monsem, 0, n); | initialise à la valeur n |
| sem_wait(&monsem); | fait -1 sur le sémaphore<br>ou met le thread en attente si sémaphore nul |
| sem_post(&monsem); | fait +1 sur le sémaphore<br>réveille éventuellement le 1er thread en attente |

retournent 0 si OK ou -1 si échec