

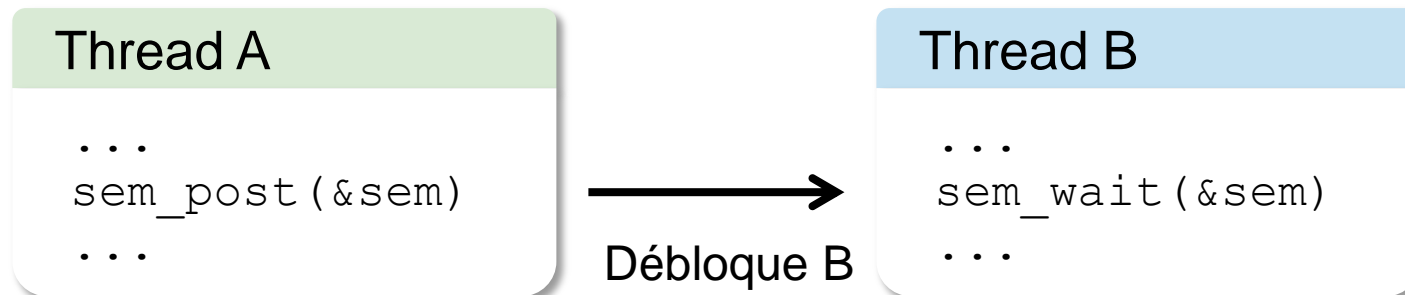
# Sémaphores

F. Gluck, *V. Pilloux*

Version 0.6

# Qu'est-ce qu'un sémaphore ?

- Un sémaphore est un mécanisme de synchronisation par **attente passive** inventé par E.W. Dijkstra en 1965.
- Un sémaphore sert à signaler un événement (souvent d'un thread à un autre, mais pas forcément).
- Sous Posix, un événement est signalé avec `sem_post()` et le code qui attend l'événement utilise `sem_wait()`
- Exemple d'utilisation:



# Définition d'un sémaphore

- Un sémaphore est une primitive de synchronisation par attente passive constituée :
  - d'une variable d'état entière (toujours  $\geq 0$ )
  - d'une file d'attente
  - de deux opérations **atomiques sur la variable entière**:
    - **increment** (incrément et réveil possible d'un thread, provoqué par `sem_post()`)
    - **decrement** (décrément et blocage potentiel, provoqué par `sem_wait()`)

# Principe de fonctionnement

- Un sémaphore est en fait représenté par un entier aux propriétés suivantes :
  - **Toujours** initialisé à une **valeur entière  $\geq 0$**
  - Les **seules** opérations possibles sont l'incrémentation ou la décrémentation.
  - Il n'est, en général, **pas possible de lire la valeur d'un sémaphore**.
  - Lorsqu'un thread **décrémente** un sémaphore, si le résultat est négatif, le thread est alors **suspendu** jusqu'à ce qu'un autre thread incremente le sémaphore.
  - Lorsqu'un thread **incrémente** un sémaphore, un des threads suspendus est réveillé (si au moins un thread en attente).

# Pseudo-code d'un sémaphore

- Pseudo-code des opérations **atomiques** d'incrémentation et de décrémentation :

```
void increment(semaphore s) {  
    s = s + 1;  
    if (!s.list.is_empty())  
        wakeup(s.list.get_one_thread());  
}  
  
void decrement(semaphore s) {  
    s = s - 1;  
    if (s < 0) {  
        block(calling_thread);  
        s.list.insert(calling_thread);  
    }  
}
```

# Valeur d'un sémaphore

- Signification de la valeur d'un sémaphore :
  - **Valeur > 0** : représente le nombre de threads pouvant décrémenter le sémaphore sans bloquer.
  - **Valeur < 0** : représente le nombre de threads bloqués en attente.
  - **Valeur == 0** : signifie qu'aucun thread n'est bloqué, mais que si un thread décrémente le sémaphore, alors le thread sera bloqué.

# Propriétés d'un sémaphore

- Propriétés découlant de la définition d'un sémaphore :
  - Avant de décrémenter un sémaphore, **pas moyen de savoir** (généralement) **si le thread sera suspendu ou pas**.
  - Après incrémentation d'un sémaphore par un thread, un autre thread est réveillé et les deux threads s'exécutent de manière concurrente ; **pas moyen de savoir quel thread va immédiatement continuer** (potentiellement aucun).
  - Lorsqu'un sémaphore est incrémenté, on ne sait pas forcément si un autre thread est en attente.

# Terminologie

- La terminologie originale utilisée par Dijkstra était **V** pour l'incrémentation (*Verhoog* signifiant augmenter) et **P** pour la décrémentation (*Probeer te verlagen* signifiant « essayer de réduire »).
- Au fil du temps, plusieurs terminologies ont vu le jour afin de rendre les opérations effectuées plus explicites.
- Terminologies communément utilisées pour l'incrémentation :
  - **V, increment, signal, post**
- Terminologies pour la décrémentation :
  - **P, decrement, wait**
- Nous utiliserons ici la terminologie POSIX : **post** et **wait**.



# Comparaison mutex / sémaphore

- Mutex et sémaphore sont des mécanismes par attente passive possédant chacun une file d'attente de threads bloqués.
- Le rôle d'un mutex est uniquement de fournir un mécanisme d'**exclusion mutuelle**.
- Un sémaphore est un mécanisme de **signalisation** permettant de résoudre une plus grande classe de problèmes que les mutex.
- Contrairement à un mutex, **un sémaphore ne possède pas de propriétaire !**
- Un sémaphore binaire (i.e. initialisé à 1) peut-être utilisé pour verrouiller une section de code similairement à un mutex.
- **ATTENTION** : un sémaphore binaire **n'est pas** un mutex ! Les mutex gèrent l'héritage de priorité, ce que les sémaphores ne font pas (hors du cadre de ce cours).

# Utilisation des sémaphores

- La spécification POSIX.1-2001 met à disposition :
  - Le fichier header `semaphore.h`
  - Un type sémaphore `sem_t`
  - Des fonctions d'initialisation et de destruction :
    - `sem_init`
    - `sem_destroy`
  - Des fonctions de manipulation :
    - `sem_wait`
    - `sem_post`

# Création et destruction

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- Créé un sémaphore et l'initialise à la valeur spécifiée ;
- `pshared` indique s'il s'agit d'un sémaphore partagé entre threads (valeur 0) ou partagé entre processus (valeur 1) ;
- Renvoie 0 en cas de succès.

```
int sem_destroy(sem_t *sem);
```

- Détruit le sémaphore spécifié ;
- **Attention** : **comportement indéterminé** si un sémaphore possédant des threads bloqués est détruit !
- Renvoie 0 en cas de succès.

# Signalisation

```
int sem_post(sem_t *sem);
```

- Incrémente le sémaphore spécifié et potentiellement réveille un des threads bloqué sur le sémaphore (si au moins un thread bloqué) ;
- Renvoie 0 en cas de succès.

# Attente

```
int sem_wait(sem_t *sem);
```

- Décrémente le sémaphore spécifié. Si la valeur du sémaphore est  $> 0$ , sa valeur est décrémentée et la fonction retourne immédiatement ;
- Si sa valeur est égal à 0, alors l'appel bloque jusqu'à ce que sa valeur devienne  $> 0$  ;
- Renvoie 0 en cas de succès et -1 en cas d'erreur.

# Sémaphore pour l'exclusion mutuelle

- On désire réaliser une exclusion mutuelle à l'aide d'un sémaphore.
- Soit les threads A et B effectuant une opération d'incrément sur une variable globale  $n$  (pseudo-code) :

Thread A

```
n = n + 1
```

Thread B

```
n = n + 1
```

- Comment garantir que l'opération d'incrément se fera en exclusion en mutuelle, sans utiliser de mutex ?

# Sémaphore pour l'exclusion mutuelle

- On désire réaliser une exclusion mutuelle à l'aide d'un sémaphore.
- Soit les threads A et B effectuant une opération d'incrément sur une variable globale  $n$  (pseudo-code) :

```
sem = semaphore(1)
```

## Thread A

```
sem.wait()  
n = n + 1  
sem.post()
```

## Thread B

```
sem.wait()  
n = n + 1  
sem.post()
```

- Comment garantir que l'opération d'incrément se fera en exclusion mutuelle ?

# Exemple d'implémentation

```
sem_t lock;
int n = 0;

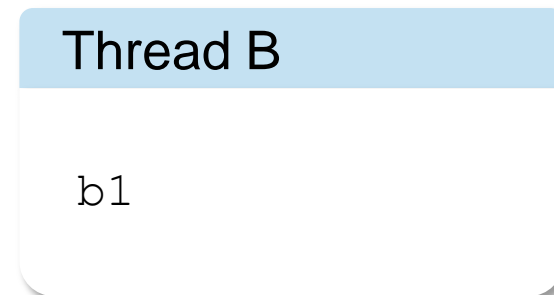
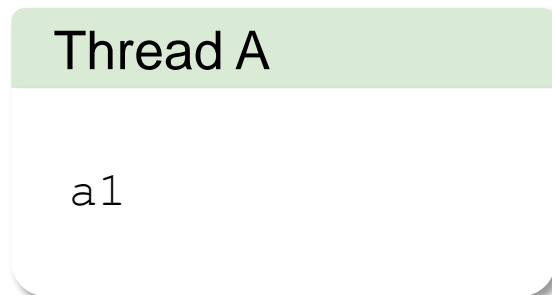
void *func(void *arg) {
    for(int i = 0; i < 1000000; i++) {
        sem_wait(&lock);
        n++;
        sem_post(&lock);
    }
}

int main() {
    // 0 indicates a non-shared semaphore (intra-process)
    // 1 is the initial value of the semaphore
    sem_init(&lock, 0, 1);
    pthread_t t1, t2;
    pthread_create(&t1, NULL, func, NULL);
    pthread_create(&t2, NULL, func, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("n = %d\n", n);
    sem_destroy(&lock);
    return EXIT_SUCCESS;
}
```



# Mécanisme de signalisation

- Soit le thread A effectuant l'opération `a1` et le thread B effectuant l'opération `b1`.
- A l'aide du mécanisme de sémaphore, donnez le pseudo code permettant de garantir que `a1` sera toujours effectué avant `b1`.



# Mécanisme de signalisation

- Soit le thread A effectuant l'opération `a1` et le thread B effectuant l'opération `b1`.
- A l'aide du mécanisme de sémaphore, donnez le pseudo code permettant de garantir que `a1` sera toujours effectué avant `b1`.

```
sem = semaphore(0)
```

Thread A

`a1`

Thread B

`b1`

# Mécanisme de signalisation

- Soit le thread A effectuant l'opération `a1` et le thread B effectuant l'opération `b1`.
- A l'aide du mécanisme de sémaphore, donnez le pseudo code (initialisation comprise) permettant de garantir que `a1` sera toujours effectué avant `b1`.

```
sem = semaphore(0)
```

## Thread A

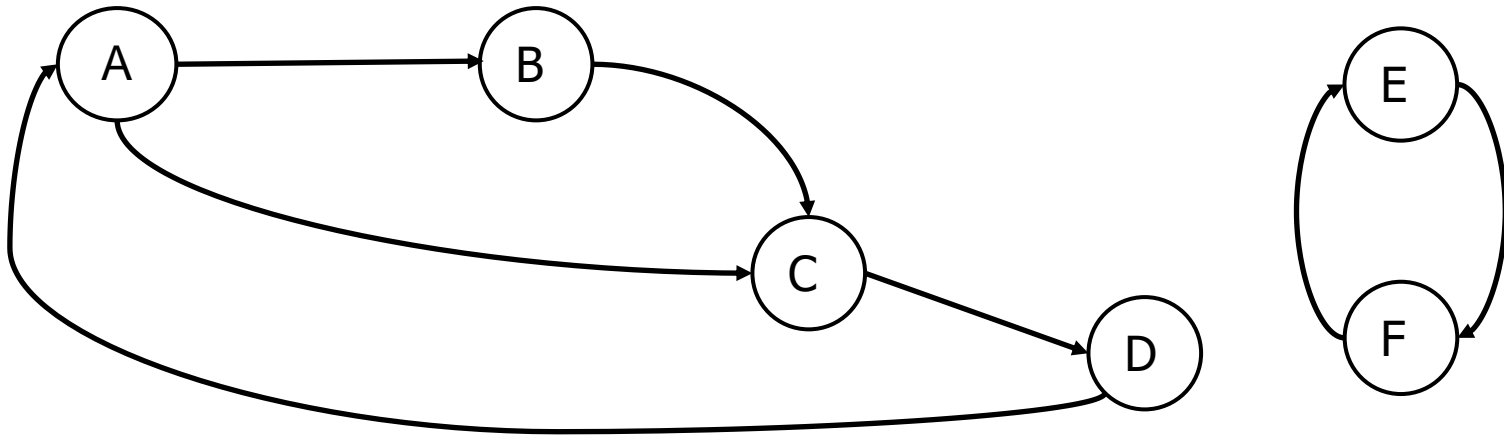
```
a1  
sem.post()
```

## Thread B

```
sem.wait()  
b1
```

# Graphe de dépendance

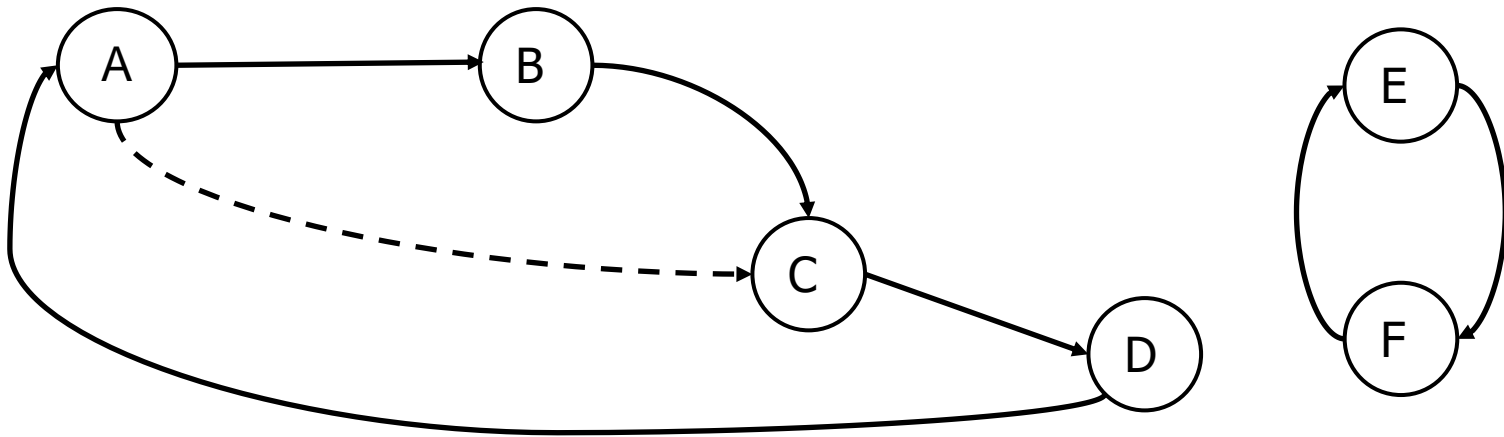
- Les sémaphores sont bien adaptées pour synchroniser des tâches dont on connaît les interdépendances
- Les interdépendances peuvent être modélisée ainsi où les cercles représentent les tâches d'un programme concurrent:



- **Combien de sémaphores faut-il utiliser pour pouvoir assurer ces dépendances?**

# Graphe de dépendance

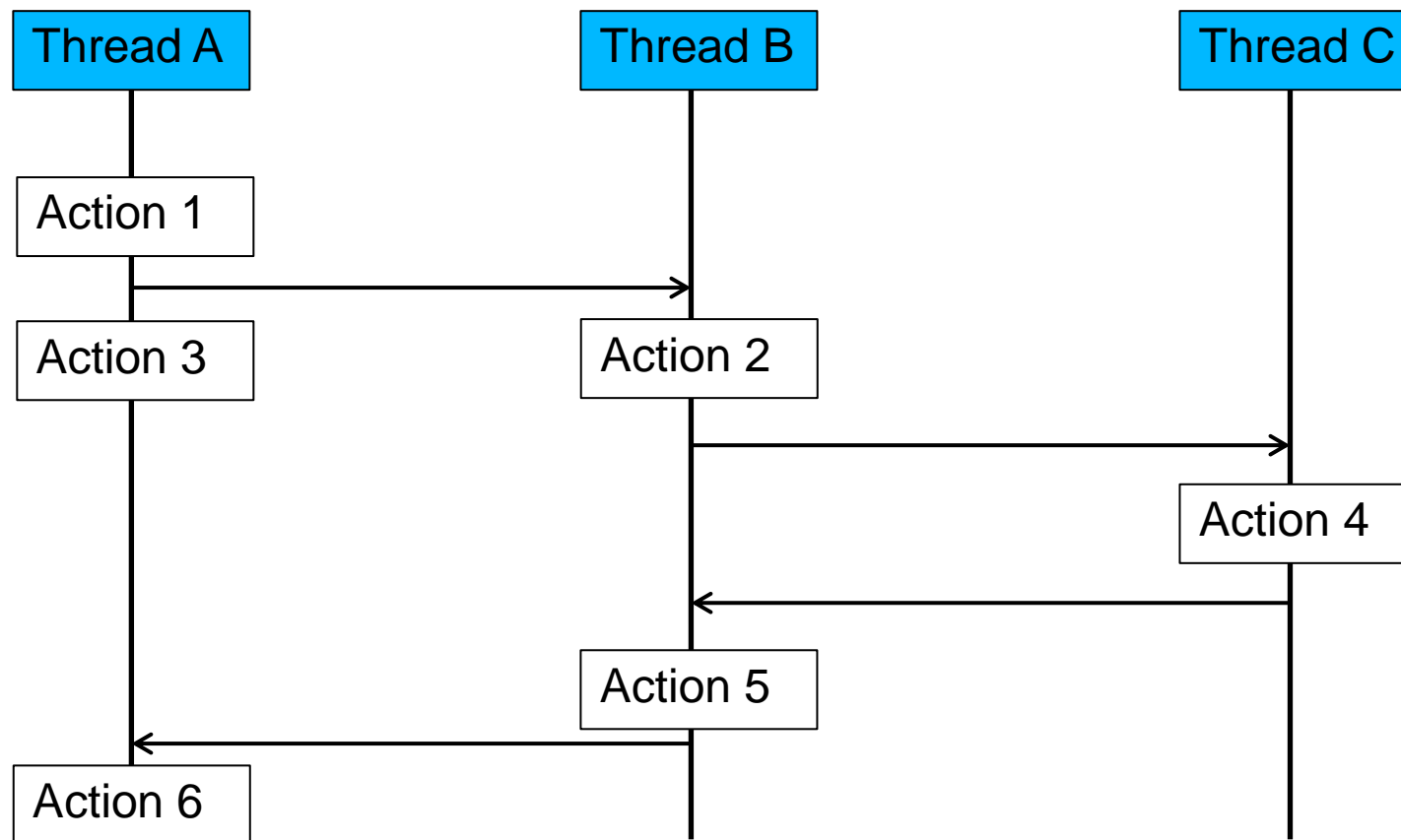
- Les sémaphores sont bien adaptés pour synchroniser des tâches dont on connaît les interdépendances
- Les interdépendances peuvent être modélisée ainsi où les cercles représentent les tâches d'un programme concurrent:



- **Combien de sémaphores faut-il utiliser pour pouvoir assurer ces dépendances?**
  - **6 !  $A \rightarrow C$  étant implicite (garanti par  $A \rightarrow B \rightarrow C$ )**

# MSC (*Message sequence chart*)

- Les MSC permettent de décrire des scénarios décrivant les interactions que les tâches peuvent avoir. Cela est pratique pour décrire la signalisation entre les tâches.
- Note: ces scénarios ne représentent pas le fonctionnement du système de façon exhaustive: il faudrait utiliser une machine d'état pour cela.



## Synchronization internals - the semaphore

- <http://www.embedded.com/design/operating-systems/4440752/Synchronization-internals---the-semaphore>

## Mutex vs. semaphores

- Partie 1: <http://blog.feabhas.com/2009/09/mutex-vs-semaphores-%E2%80%93-part-1-semaphores/>
- Partie 2: <http://blog.feabhas.com/2009/09/mutex-vs-semaphores-%E2%80%93-part-2-the-mutex/>

## man pages:

- « `man sem_overview` » dans un terminal UNIX
- « `man sem_init` » dans un terminal UNIX