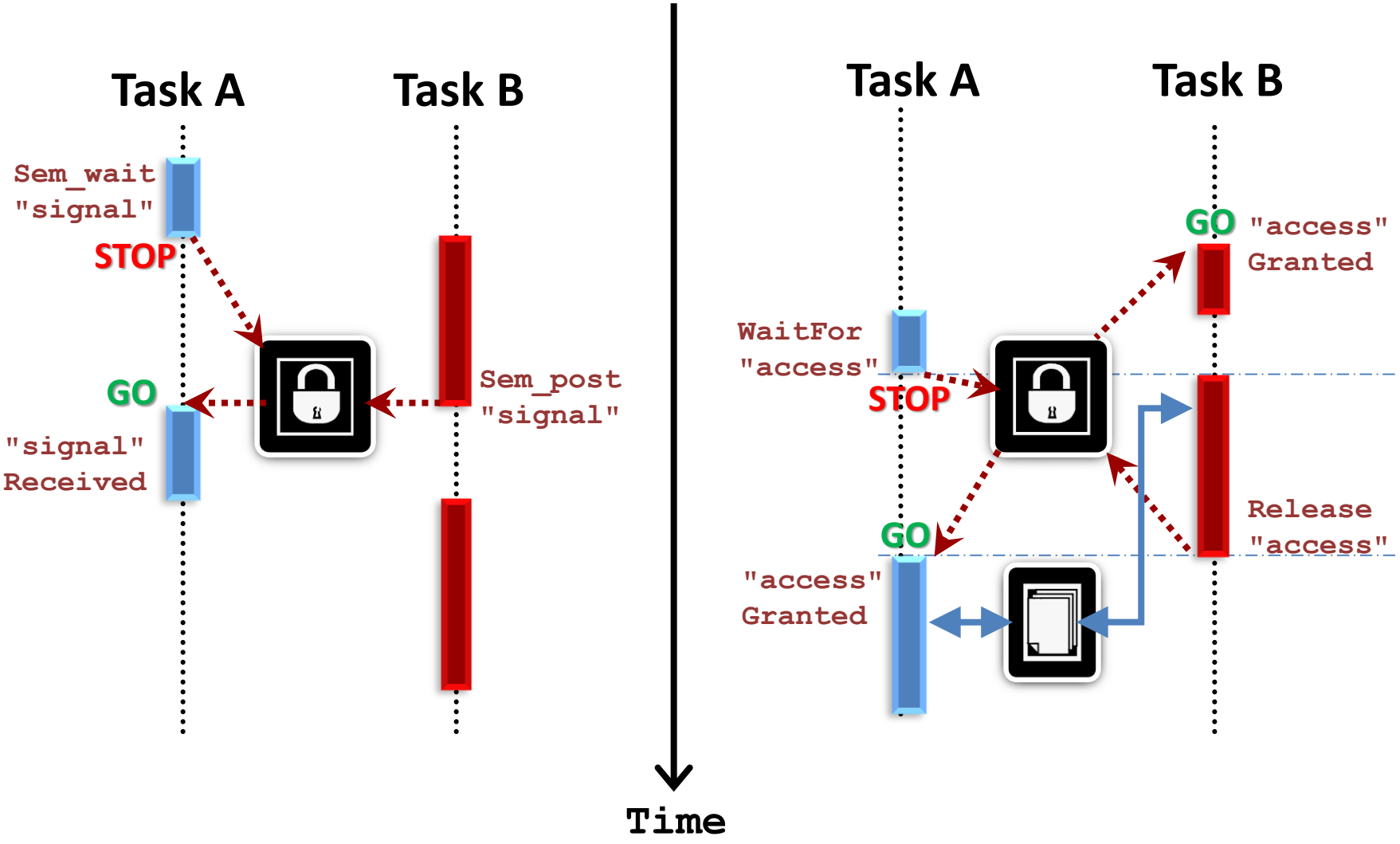
Développement d'applications natives

Synchronisation

Problème ?

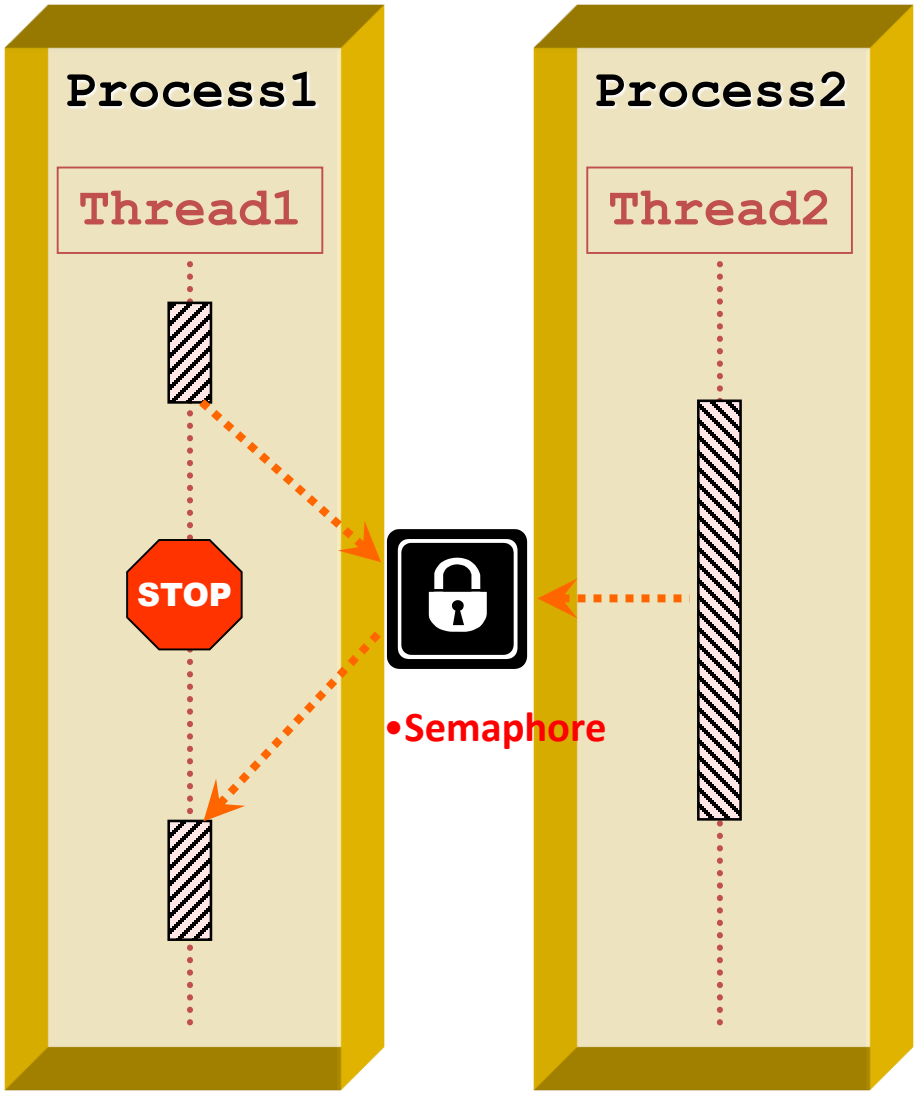
- Quoi : échange d'information et garantie de l'intégrité des données
  - Accès simultanés (concurrence)
  - Lecteur/Ecrivain asynchrones
- Comment: utiliser les mécanismes de synchronisation
  - Signalisation
  - Protection

Signalisation vs. Protection



Inter process

- Contexte :
  - Différents espaces d'adressage
- Mécanismes :
  - Semaphore
  - wait



Generalités

- Attente d'un **événement système**:
  - Un **`semaphore`** est relâché (incrémenté)
  - Un **`thread`** se termine
  - Un **`process`** se termine

- Utilisation du type **`sem_t`**
- Inclusion des en-têtes: **`fcntl.h`** (pour les constantes 0_*), **`sys/stat.h`** (pour les constantes "modes"), **`semaphore.h`**.
- Lors de l'édition de liens: **`-lrt`** et/ou **`-lpthread`**.

- Pour obtenir un sémaphore sur le même objet noyau dans différents processus:
  - Nommer les objets noyau par un nom commençant par "/"

Création… – API

```
sem_t *sem_open(const chr *name, int oflag);
sem_t *sem_open(const chr *name,
                int oflag,
                mode_t mode,
                unsigned int value);
```

- name: chaine de caractère représentant le nom du sémaphore
- oflag: O_CREAT (pour la création) et O_EXCL (pour l'execution)
- mode: droit du fichier (Obligatoire si O_CREAT)
- value: valeur initiale du sémaphore (≥ 0, obligatoire si O_CREAT).

->valeur de retour: adresse du sémaphore en cas de succès. Null en cas d'échec, Errno sera à SEM_FAILED.

```
MySem= sem_open("/sync1", O_CREAT  , 0644, 0); // creation
MySem= sem_open("/sync1", 0   , 0644, 0);      // ouverture
```

Destruction… – API

Libération du sémaphore:

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

- sem: sémaphore que l'on souhaite libérer.

->valeur de retour: 0 succès. -1 en cas d'échec, Errno sera à EINVAL.

- Tous les process doivent fermer le sémaphore avant de pouvoir le libérer.

Suppression du nom du sémaphore:

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

- sem: sémaphore que l'on souhaite libérer.

->valeur de retour: 0 succés. -1 en cas d'échec, Errno sera à: ENONENT

Synchronisation… – API

Attente du sémaphore:

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

- sem: sémaphore que l'on souhaite attendre.

->valeur de retour: 0 succès. -1 en cas d'échec, Errno sera à EAGAIN.

- La valeur du sémaphore sera décrémentée. Si elle devient égale à 0, l'appel bloquant jusqu'à incrémentation par un autre process.

Relâchement du sémaphore:

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

- sem: sémaphore que l'on souhaite relâcher.

->valeur de retour: 0 succés. -1 en cas d'échec, Errno sera à: EINVAL

- La valeur du sémaphore sera incrémentée. Si elle devient supérieur à 0,un process de la file d'attente (sem_wait) sera réveillé et procédera au verrouillage du sémaphore.

Synchronisation… – API

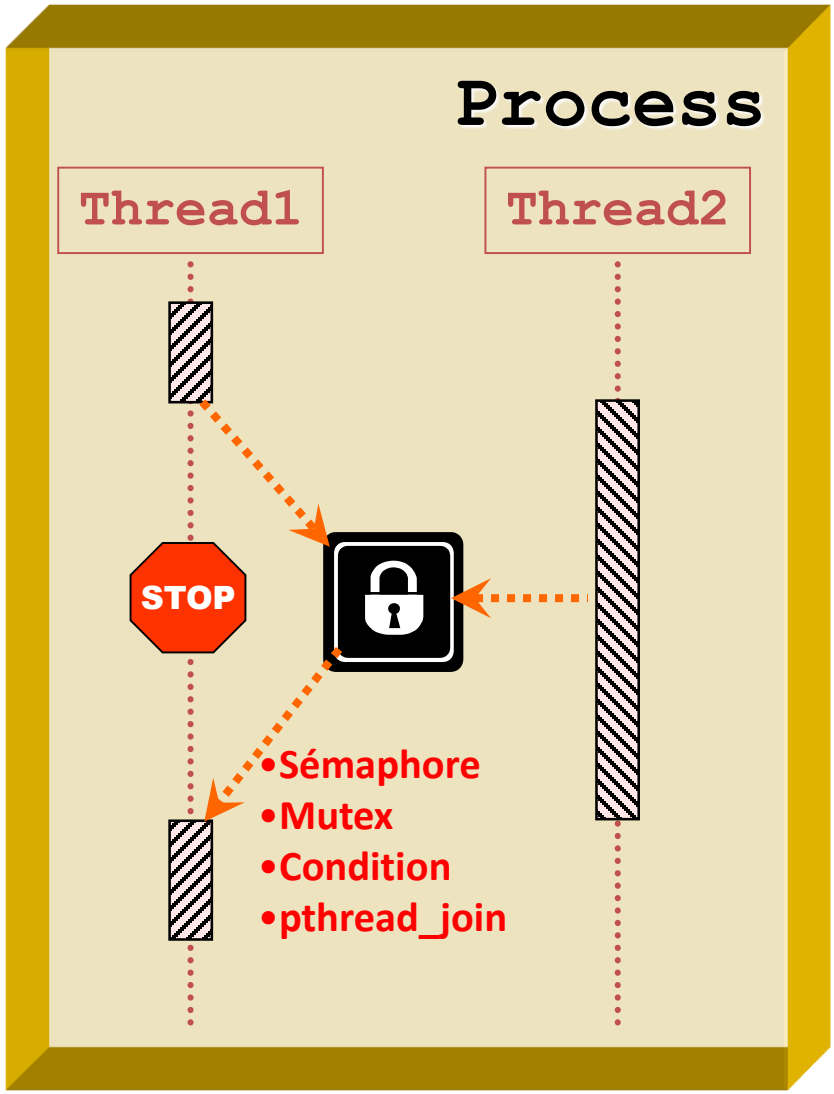
```
wait();
```

Bloque le processus appelant jusqu'à ce qu'un de ses processus fils se termine.

```
pid_t pid;
pid = fork();
switch(pid){
    case -1: // gestion de l'erreur
        …
    case 0: //code fils
        …
    default: //code père
        wait(NULL);
         …
```

Intra process

- Contexte :
  - Même espace d'adressage
- Mécanismes :
  - Sémpahore
  - Mutex
  - Condition
  - pthread_join



Sémaphore

- 2 types de sémaphores: POSIX et systeme V
  - **POSIX** :
    - Même type que pour l'Inter process: `sem_t.`
  - **Système V:**
    - Type: `struct semaphore;`
    - Initialisation:

```
void sema_init(struct semaphore *sem, int val);
```

      - Val: valeur initiale assignée au sémaphore
    - Attente:

```
void down(struct semaphore *sem);
```

    - Relachement:

```
void up(struct semaphore *sem);
```

Mutex

- Objet d'exclusion mutuelle (MUTual Exclusion device)
  - 2 état: déverrouillé ou verrouillé.
  - Ne peut être pris que par un seul thread à la fois.
  - Un thread qui tente de verrouiller un mutex déjà verrouillé sera suspendu jusqu'à ce que le mutex soit déverrouillé.
- API :
  - type: `pthread_mutex_t`
  - Création: `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);` `(mutexattr = NULL)`
  - Verrouillage: `int pthread_mutex_lock(pthread_mutex_t *mutex));`
  - Déverrouillage: `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
  - Destruction:`int pthread_mutex_destroy(pthread_mutex_t *mutex);`

Condition

- Une condition permet de signaler qu'un thread a terminé une activité, et ainsi réveiller un thread en attente.
  - Elle doit toujours être associée à un mutex
- API :
  - type: `pthread_cond_t`
  - Création: `int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);`
  - Levée de la condition: `int pthread_cond_signal(pthread_cond_t *cond);`
  - Attente de la condition: `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
  - Destruction: `int pthread_cond_destroy(pthread_cond_t *cond);`

pthread_join

```
int pthread_join(pthread_t thread, void **value_ptr);
```

thread: thread attendu.

value_ptr: valeur retournée par *pthread_exit*.

-> retourne 0 si le thread se termine correctement, sinon le code d'erreur correspondant.