

# Mutex, barrière de synchronisation, spinlock

F. Gluck, V. Pilloux

version 0.6

# Mutex

- **Mutex (*mutual exclusion*)** : mécanisme de verrou permettant l'exclusion mutuelle par attente passive.
- Un mutex est une structure de donnée constituée :
  - d'une variable d'état booléenne (verrouillé/libre),
  - d'une file d'attente,
  - d'un propriétaire,
  - de deux opérations **atomiques** :
    - `lock` (verrouillage)
    - `unlock` (déverrouillage).

# Pseudo-code d'un mutex

Pseudo-code des opérations de verrouillage et déverrouillage :

```
void lock(mutex m) {  
    if (m.locked) {  
        block(calling_thread);  
        list_associated_to_m.insert(calling_thread);  
    }  
    else  
        m.locked = true;  
}  
  
void unlock(mutex m) {  
    if (m.owner != thread_courant) error(); // *  
    if (list_associated_to_m.empty())  
        m.locked = false;  
    else  
        wakeup_one_of_the_waiting_thread();  
}
```

\* idéalement: en réalité le comportement n'est pas toujours déterminé

# Mutex – points importants

- Un mutex possède un « propriétaire » : le thread ayant obtenu le verrou est le **propriétaire** du mutex jusqu'à ce qu'il le déverrouille :
  - ⇒ le thread ayant verrouillé le mutex est responsable de le déverrouiller (relâcher); **un thread ne doit pas déverrouiller un mutex déjà verrouillé par un autre thread.**

# Mutex – points importants

- Un mutex possède un « propriétaire » : le thread ayant obtenu le verrou est le **propriétaire** du mutex jusqu'à ce qu'il le déverrouille :
  - ⇒ le thread ayant verrouillé le mutex est responsable de le déverrouiller (relâcher); **un thread ne doit pas déverrouiller un mutex déjà verrouillé par un autre thread.**
- Les fonctions `lock` et `unlock` sont **atomiques** !

# Mutex – points importants

- Un mutex possède un « propriétaire » : le thread ayant obtenu le verrou est le **propriétaire** du mutex jusqu'à ce qu'il le déverrouille :
  - ⇒ le thread ayant verrouillé le mutex est responsable de le déverrouiller (relâcher); **un thread ne doit pas déverrouiller un mutex déjà verrouillé par un autre thread.**
- Les fonctions `lock` et `unlock` sont **atomiques** !
- Verrouiller un mutex déjà verrouillé bloque le thread appelant ⇒ **deadlock** assuré si un même thread verrouille un mutex deux fois de façon consécutive (sauf pour mutex récursifs).

# Mutex – points importants

- Un mutex possède un « propriétaire » : le thread ayant obtenu le verrou est le **propriétaire** du mutex jusqu'à ce qu'il le déverrouille :
  - ⇒ le thread ayant verrouillé le mutex est responsable de le déverrouiller (relâcher); **un thread ne doit pas déverrouiller un mutex déjà verrouillé par un autre thread.**
- Les fonctions `lock` et `unlock` sont **atomiques** !
- Verrouiller un mutex déjà verrouillé bloque le thread appelant ⇒ **deadlock** assuré si un même thread verrouille un mutex de façon consécutive (sauf pour mutex rékursifs).
- Déverrouiller un mutex plusieurs fois n'a pas d'effet ⇒ pas de « mémoire » du nombre de fois qu'un mutex a été déverrouillé (sauf pour mutex rékursifs).

# Utilisation de mutex

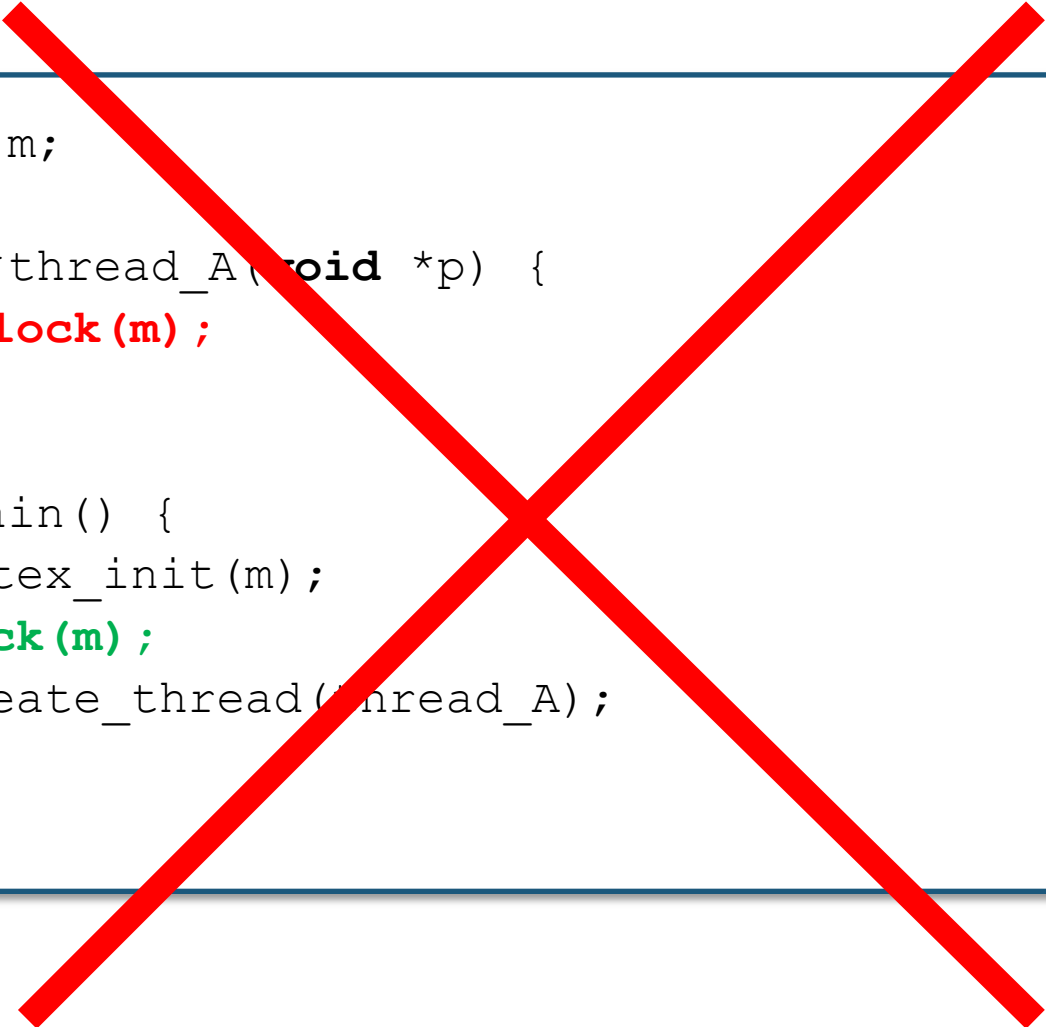
- Typiquement les sections critiques sont protégées par des mutex
- L'utilisation de mutex fonctionne pour un nombre quelconque de threads

```
mutex m;  
mutex_init(m);  
.  
.  
.  
lock(m);  
// section critique  
unlock(m);  
.  
.  
.
```



# Mauvaise utilisation de mutex

Un thread ne **doit pas** déverrouiller un mutex déjà verrouillé par un autre thread !



```
mutex m;  
  
void *thread_A(void *p) {  
    unlock(m);  
}  
  
int main() {  
    mutex_init(&m);  
    lock(m);  
    create_thread(thread_A);  
    ...  
}
```

# Utilisation de mutex

- La librairie pthread met à disposition :
  - le type `pthread_mutex_t`
  - des fonctions de manipulation :
    - `pthread_mutex_lock(pthread_mutex_t *mutex)`
    - `pthread_mutex_unlock(pthread_mutex_t *mutex)`
  - des fonctions et macros d'initialisation et destruction :
    - `pthread_mutex_init()`
    - `pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;`
    - `pthread_mutexattr_init()`
    - `pthread_mutexattr_...()`
    - `pthread_mutex_destroy()`

# Types de mutex

- La librairie POSIX Threads définit trois types de mutex:
  - **mutex normal** (ou rapide);
  - **mutex sécurisé** (*error checking*);
  - **mutex récursif** (*recursive*).
- Le type d'un mutex définit le comportement à adopter en cas de verrouillage et déverrouillage.

# Types de mutex et comportement

- **Verrouillages successifs par le même thread :**
  - **mutex normal** : le thread est bloqué (deadlock)
  - **mutex sécurisé** : erreur de verrouillage; renvoie la valeur EDEADLK.
  - **mutex récursif** : verrouillage réussi; le thread devra déverrouiller le mutex autant de fois qu'il aura été verrouillé.
- **Déverrouillage d'un mutex verrouillé par un autre thread :**
  - **mutex normal**  $\Rightarrow$  **comportement indéterminé !**
  - **mutex sécurisé** et **mutex récursif** : vérifie que le propriétaire est bien le thread appelant et dans le cas contraire, renvoie une erreur; le mutex reste verrouillé.

# Création de mutex

- Déclaration et initialisation statique avec les macros :

```
pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;  
pthread_mutex_t recmutex = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;  
pthread_mutex_t errchkmutex = PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
```

- Initialisation dynamique avec le type par défaut (normal) :

```
pthread_mutex_t mutex;  
pthread_mutex_init(&mutex, NULL);
```

- Initialisation dynamique avec spécification du type de mutex :

```
pthread_mutexattr_t attr;  
pthread_mutexattr_init(&attr);  
// PTHREAD_MUTEX_ERRORCHECK ou PTHREAD_MUTEX_RECURSIVE  
pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_NORMAL);  
pthread_mutex_init(&mutex, &attr);
```

- Après initialisation, un mutex n'est jamais verrouillé.

# Verrouillage

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

- Si le mutex est déverrouillé, il devient verrouillé et son propriétaire est alors le thread appelant.
- Mutex récursif : peut être verrouillé plusieurs fois; un compteur mémorise le nombre de verrouillages effectués.
- Si le mutex est déjà verrouillé par un autre thread, le thread appelant est suspendu jusqu'à ce que le mutex soit déverrouillé.

# Déverrouillage

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

- Déverrouille (libère) le mutex; le mutex est supposé être verrouillé par le thread appelant avant l'appel à `pthread_mutex_unlock`.
- Si c'est un mutex normal, il est toujours déverrouillé.
- Si c'est un mutex récursif, son compteur est décrémenté; lorsqu'il atteint 0, le mutex est déverrouillé.
- Un mutex normal ne vérifie pas que le thread appelant est propriétaire du mutex; le mutex est déverrouillé même s'il n'en est pas le propriétaire  
⇒ **A EVITER** car pas portable !
- Les autres types de mutex retournent une erreur s'ils sont déverrouillés par un thread autre que son propriétaire et leur état reste inchangé.
- `pthread_mutex_unlock` retourne 0 en cas de succès.

# Destruction

- Libère les ressources utilisées par un mutex :

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Si des attributs ont été créés, ne pas oublier de les libérer avec :

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```



# Exemple

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int n = 0;

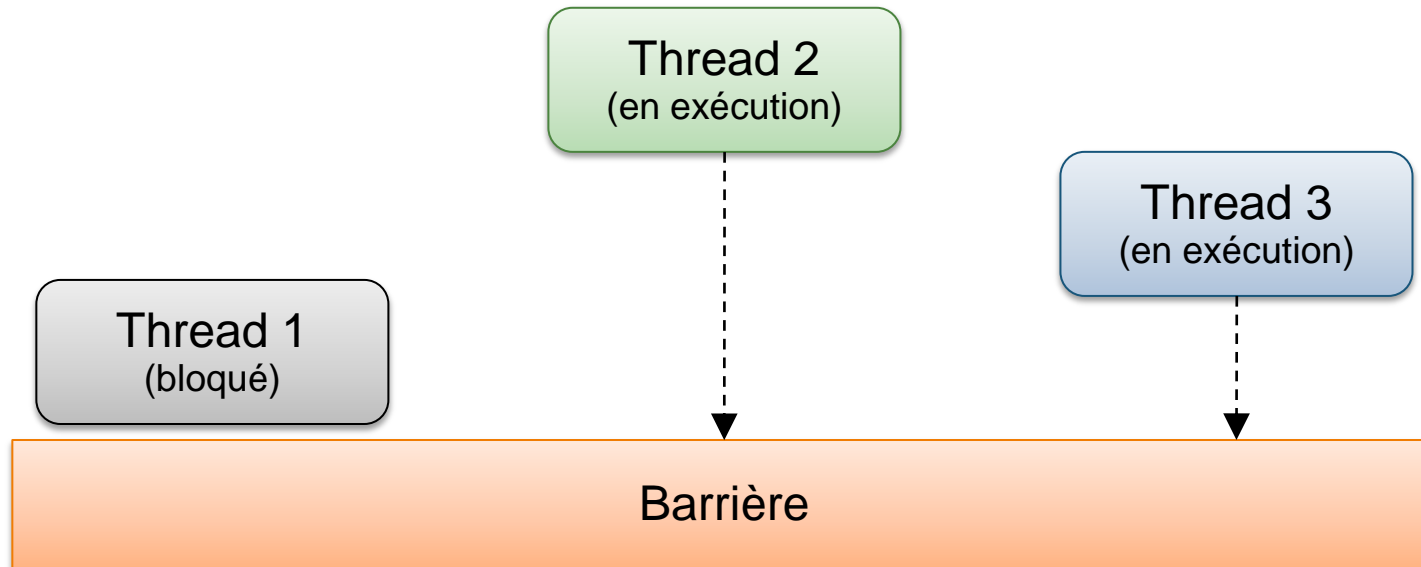
void *func(void *arg) {
    for(int i = 0; i < 1000000; i++) {
        pthread_mutex_lock(&mutex);
        n++;
        pthread_mutex_unlock(&mutex);
    }
}

int main() {
    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);
    return EXIT_SUCCESS;
}
```

# Verrouillage et performances

- Mutex implémentés à l'aide d'instructions machines atomiques comme « Test And Test » ou « Compare And Swap ».
- Implémentations pour architectures modernes nécessitent l'utilisation de barrières mémoires  $\Rightarrow$  garanti sérialisation et cohérence mémoire durant les accès aux variables à l'intérieur de la section critique.
- Une section critique implique une sérialisation  $\Rightarrow$  dégradation des performances !
- Toute section critique devrait être **la plus courte possible !**

# Barrière de synchronisation



- Une barrière est un point de synchronisation pour  $n$  threads (dans cet exemple ci-dessus, 3).
- Aucun thread ne peut passer la barrière tant que tous n'y sont pas encore arrivés  $\Rightarrow$  attente passive.

# Fonctionnement d'une barrière

Fonctionnement d'une barrière de synchronisation POSIX:

1. Le nombre de threads à synchroniser est spécifié lors de la création de la barrière.
2. Chaque thread notifie son arrivée à la barrière.
3. Tant que tous les threads n'ont pas notifiés la barrière, ceux déjà arrivés sont **bloqués**.
4. Une fois que tous les threads ont notifiés la barrière, celle-ci débloque tous les threads en attente (un par un, dans un ordre **indéterminé**).
5. La barrière est ensuite **réinitialisée** à la valeur spécifiée lors de sa création  $\Rightarrow$  barrière réutilisable.

# Utilisation des barrières

```
int pthread_barrier_init(pthread_barrier_t *barrier,  
                        const pthread_barrierattr_t *attr, unsigned count);  
  
int pthread_barrier_wait(pthread_barrier_t *barrier);  
  
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

- `pthread_barrier_init` crée une barrière pour `count` threads
- Chaque thread notifie son arrivée via `pthread_barrier_wait`
- Les ressources liées à une barrière sont libérées avec `pthread_barrier_destroy`
- Toutes ces fonctions renvoient 0 en cas de succès.

# Destruction d'une barrière

## Attention !

- La destruction d'une barrière de synchronisation est **bloquante** si un ou plusieurs threads sont encore en attente sur celle-ci lors de l'appel à `pthread_barrier_destroy`.
- **Risque de deadlock potentiel !**

# Exemple d'utilisation

```
pthread_barrier_t b;

void* func() {
    printf("2nd thread before barrier\n");
    pthread_barrier_wait(&b);
    printf("2nd thread after barrier \n");
}

int main() {
    printf("main thread started\n");
    pthread_barrier_init(&b, NULL, 2);

    pthread_t thread;
    pthread_create(&thread, NULL, func, NULL);

    pthread_barrier_wait(&b);
    printf("main thread finished\n");
    pthread_join(thread, NULL);

    pthread_barrier_destroy(&b);
    return EXIT_SUCCESS;
}
```

# Important : compilation

- Selon le type de système/compilateur utilisé, il se peut que les fonctions ci-dessous ne soient pas reconnues par le compilateur :
  - mutex sécurisés et récursifs
  - barrières de synchronisation
- Deux possibilités pour régler ce problème :
  1. Préciser le standard GNU11 à la compilation avec l'option :  
`-std=gnu11`
  2. Définir le symbole `_GNU_SOURCE` **avant** l'inclusion des headers :

```
#define _GNU_SOURCE
#include <stdio.h>
#include <pthread.h>
...
```



# Documentation développement

- Sur un système de type Debian/Ubuntu, les pages de manuel (`man`) pour les barrières ne sont pas installées par défaut, de même qu'une partie de la documentation pour les mutex.
- Pour obtenir le manuel complet des fonctions de la librairie `pthread`, il est nécessaire d'installer le package `manpages-posix-dev`

```
~$ sudo apt-get install manpages-posix-dev
```

# Attente active vs passive (1)

- Attente active sur une ressource :
  - CPU utilisé à 100% dans sa boucle d'attente.
  - **Gaspillage** de cycles processeur pendant l'attente !
- Solution pour éviter l'attente active ?
  - Attente passive.
  - Thread **inactif** pendant l'attente à la ressource critique  
⇒ 0% processeur utilisé !

# Attente active vs passive (2)

- Soit un accès à une section critique déjà verrouillée :
  - Attente passive : threads bloqués mis en attente passive par l'OS (exemple: mutex).
  - Attente active : threads essaient continuellement d'acquérir le verrou !
- Cas où attente active peut être préférable ?

# Attente active vs passive (2)

- Soit un accès à une section critique déjà verrouillée :
  - Attente passive : threads bloqués mis en attente passive par l'OS.
  - Attente active : threads essaient continuellement d'acquérir le verrou !
- Cas où attente active peut être préférable ? Rarement:
  - Lorsque latence très faible est requise : temps de réponse plus rapide car aucun thread à réveiller lorsque verrou devient libre.
  - Quand le changement de contexte est plus long que le temps d'attente *moyen* ou *maximum* (ex : systèmes temps réels).
  - Code kernel : lorsque le code ne peut pas être bloqué (ex : gestionnaire d'interruption)

# Spinlock (informel)

- Un spinlock, ou « verrou tournant » est un mécanisme d'exclusion mutuelle par **attente active**.
- Spinlock implémenté avec instructions matérielles spéciales, ex : Test And Test (TAS), Compare And Swap (CAS), etc.
- La librairie pthread met à disposition :
  - le type `pthread_spinlock_t`
  - des fonctions de manipulation :
    - `int pthread_spin_init(pthread_spinlock_t *lock, int pshared)`
    - `int pthread_spin_destroy(pthread_spinlock_t *lock)`
    - `int pthread_spin_lock(pthread_spinlock_t *lock)`
    - `int pthread_spin_unlock(pthread_spinlock_t *lock)`

# Approche hybride (informel)

- L'attente passive a de nombreux avantages par rapport à l'attente active mais elle souffre d'une latence plus importante.
- Est-il possible de faire mieux ?
- Approche hybride :
  - Combiner les avantages des deux types d'attentes :
    - Effectuer une attente active pendant un certain temps (court).
    - Puis passer à une attente passive.
  - Le noyau Linux implémente un mécanisme basé sur ce principe\*.

\*Fuss, *Futexes and Furwocks: Fast Userlevel Locking in Linux*, H. Franke, R. Russell, M. Kirkwood, Ottawa Linux Symposium 2002