Threads
(Chapitre 2)

Francesco Bronzino
ArchiSys

Sujets

1. Concept de thread
2. Les threads dans Linux
3. Fonctions de Pthread
4. Exemples de threads
5. Les threads dans Bash

Concept de thread

Dans Linux (et presque tous les systèmes d'exploitation), un **processus** peut avoir plusieurs flux d'exécution, appelés **Threads**.

- Les threads peuvent être considérés comme un ensemble de processus qui partagent la mémoire
- mais qui exécutent le même programme

Note : Windows permet également de créer des threads avec l'appel système `CreateThread()`.

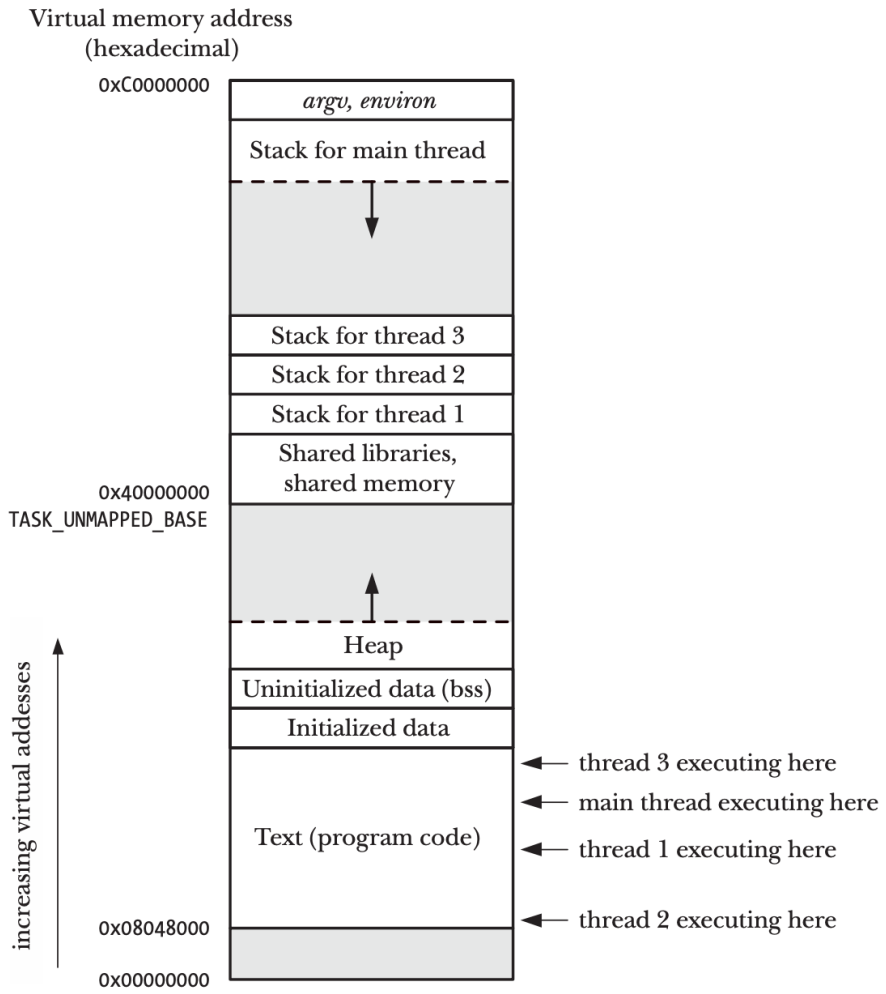
Les threads et la mémoire

Chaque Thread exécute le même programme et partage les mêmes données

- Les segments *data*, *heap* et *code* sont partagés.

Un Thread est un flux d'exécution

- Il possède son propre stack
- Il contient l'état des fonctions en cours d'exécution



Communication entre les threads

Les threads peuvent communiquer entre eux plus facilement que les processus, en utilisant :

- Des variables globales
- Constructions de synchronisation
  - Mutex
  - Variable de condition (seulement résumée ici)
  - Sémaphores

De nos jours, une architecture **multi-thread** est plus souvent utilisée qu'une architecture **multi-process**. **Pourquoi ?**

Thread POSIX ou `pthread`.

Dans les systèmes POSIX (et Linux), les **fonctions de la bibliothèque** pour gérer les threads sont appelées **pthreads**.
Les threads permettent à un processus

- D'effectuer plusieurs tâches simultanément
  - Pendant qu'un thread attend les I/O ou le réseau, un autre thread peut effectuer une autre tâche.
- Pour tirer parti d'un système **multi-cœur**.
  - Plusieurs threads s'exécutent réellement en parallèle

Historique

Les pthreads ont été initialement implémentés par la bibliothèque **LinuxThreads**.

- Les threads étaient des processus qui partageaient la mémoire, ouvraient des fichiers, etc.
- Chacun avait un PID différent
- Implémentation problématique : **concept mixte de thread et de processus**

Aujourd'hui (depuis 2002), Linux/POSIX utilise la bibliothèque **Native POSIX Threads Library (NPLT)**.

- Elle coopère avec le noyau, qui offre un support pour les threads.
- Amélioration des performances.

Ce que les threads partagent

Plusieurs threads d'un même processus partagent :

- la mémoire globale
- PID et PPID
- Fichiers ouverts
- Privilèges
- Dossier de travail

Ce que les threads ne partagent PAS

Chaque thread a :

- Un **Identifiant de thread (TID)**
  - Le noyau conserve la liste des threads et les *planifie*, les exécutant sur la CPU.
- Son *stack*
  - Afin d'exécuter des fonctions
  - Un thread mal configuré peut toujours accéder au *stack* d'un autre thread et la corrompre.
- Métadonnées

Compilation de pthreads

Le code doit inclure la directive

```
#include <pthread.h>
```

Pour compiler, vous devez inclure la bibliothèque `pthread`.

```
gcc MonProgramme.c -o MonProgramme -lpthread
```

Création d'un thread

```
#include <pthread.h>
int  pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                    void *(*start)(void *), void *arg)  ;
```

Crée un nouveau thread qui exécute la fonction `start` appelée avec l'argument `arg`

- Comme si l'on invoquait `start(arg)` sur un flux d'exécution séparé

Note : Chaque programme, lorsqu'il démarre, n'a qu'un seul thread, appelé *main thread*.

Création d'un thread

```
#include <pthread.h>
int  pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                    void *(*start)(void *), void *arg) ;
```

- L'argument `arg` est un `void*`, c'est-à-dire un pointeur vers n'importe quel type de données.
- De même, la valeur de retour de `start` est un `void*`.
- Nous ne sommes pas intéressés par l'argument `attr`, qui spécifie des attributs particuliers

Création d'un thread

```
#include <pthread.h>
int  pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                    void *(*start)(void *), void *arg)  ;
```

- L'argument `thread` est un pointeur sur une variable `pthread_t` qui contiendra l'identifiant du thread, à utiliser dans les fonctions suivantes de la bibliothèque
- En cas de succès, elle renvoie 0, sinon un code d'erreur

Fin d'un thread

Un thread se termine si :

- La fonction de démarrage `start` (pas `main` !) exécute un `return`.
- Le thread exécute un `pthread_exit()`
- Le thread exécute un `pthread_cancel()`
- Le processus se termine
  - Tout thread invoque un `exit()` ou le thread principal termine le `main`.

Fin d'un thread

```
#include <pthread.h>
void  pthread_exit(void *retval) ;
```

Termine le thread courant avec la valeur `retval`.

Cela équivaut à faire un `return` dans la fonction de démarrage du thread.

Thread ID

```
#include <pthread.h>
pthread_t  pthread_self(void) ;
```

Permet à un thread d'obtenir son propre Thread ID.

Le Thread ID doit être traité comme un `"opaque handle*"`.

- Sous Linux, c'est un `long int`.
- Mais il peut s'agir d'un pointeur sur une structure de données arbitraire
- Il n'est pas fiable de déchiffrer sa valeur.

Join un thread

```
include <pthread.h>
int  pthread_join(pthread_t thread, void **retval) ;
```

Attend que le `thread` se termine.

- S'il est déjà terminé, il revient instantanément

Enregistre la valeur de retour dans `retval`.

- `retval` est spécifié par le thread mourant via `pthread_exit()` ou `return`
- `retval` est un `void**`, c'est-à-dire un pointeur sur `void`.
  - C'est l'adresse d'une variable qui contient un pointeur

Join un thread

Les threads doivent tous être attendus via une `pthread_join()`, sinon ils deviennent des zombies.

- Comme pour les processus

En utilisant la fonction `int pthread_detach(pthread_t thread )` vous pouvez indiquer que le thread `thread` n'a pas besoin d'une `join`.

- La valeur de retour est rejetée
- Le système supprime toute information sur le thread lorsqu'il se termine.

Join d'un thread

Les threads sont pairs

- N'importe quel thread peut faire un `pthread_join` sur un autre.

Il n'y a pas de moyen d'attendre qu'un **n'importe quel** thread se termine

- Avec les processus, vous pouvez utiliser `wait` à la place.

Un `pthread_join` est toujours bloquant.

- Différent de `waitpid` avec le drapeau `WNOHANG`.

Création d'un thread

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static void * threadFunc(void *arg){
     char *s = (char *) arg ; // Conversion de void * en char *
     printf("From Thread : %s", s) ;
     return (void *) strlen(s) ; // Valeur de retour du thread
     // Équivalent à pthread_exit((void *) strlen(s))
}

int main(int argc, char *argv[]){
     pthread_t t1 ;
     void *res ; // Pour la valeur de retour
     int s ;

     s = pthread_create(&t1, NULL, threadFunc, "Hello world") ; // Création
     if (s != 0){
          printf("Cannot create thread") ;
          exit(1) ;
     }

     printf("Message de main()\n") ;
     s = pthread_join(t1, &res) ; // Join. Nécessite un void **, c'est-à-dire &res
     if (s != 0){
          printf("Cannot join thread") ;
          exit(1) ;
     }
     printf("Thread returned %ld\n", (long) res) ; // Utilisation de la valeur de retour
     exit(0) ;
}
```

Vectors de threads

Si nous créons un programme qui utilise 10 threads qui attendent une seconde avant de se terminer.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define MAXSLEEP 5
#define THREADNB 10

static void * sleepFunc(void *arg){
     char thread_number = *((char*)arg) ;
     int n=rand() % MAXSLEEP ;
     sleep(n) ;
     printf("Thread %c terminated\n", thread_number) ;
     return NULL ;
}

int main(int argc, char *argv[]){
     int i ;
     pthread_t t [THREADNB] ;
     char names [THREADNB] ;

     for (i=0;i<THREADNB;i++){
          names[i] = 'A' + i ;
          pthread_create(&t[i], NULL, sleepFunc, &names[i]) ;
     }
     for (i=0;i<THREADNB;i++)
          pthread_join(t[i], NULL) ;
     return 0 ;
}
```

Threads dans Bash

Normalement, les commandes `ps` et `top` affichent uniquement les processus.

Pour visualiser les threads :

- `ps -T opzioni`. Exemple : `ps Tax`
- `top -H`

Une démo rapide

Synchronisation

Sujets

1. Pourquoi c'est nécessaire
2. Mutex
3. Les sémaphores

Définitions

Concurrence : un programme avec des flux d'exécution multiples.

Parallélisme : un programme qui exécute plusieurs calculs simultanément.

Un programme peut être **concurrent sans être parallèle**.

- Il comporte plusieurs fils d'exécution sur un système doté d'une seule unité centrale.

Définitions

Un programme peut être **parallèle sans être concurrent**

- Les CPU modernes ont des instructions qui manipulent des données multiples.
- Paradigme **Instruction unique et données multiples (SIMD)**
- Une seule instruction pour additionner deux vecteurs
- La CPU dispose d'une UAL qui permet des opérations multiples en parallèle.
- Utilisation d'un seul thread/processus

Objectifs de la programmation parallèle

Théoriquement, en parallélisant et en utilisant N cœurs au lieu de 1, nous devrions avoir :

$$Temps\ Core = \frac{Temps\ avec\ un\ core}{N}$$

En réalité, cela ne s'applique qu'à un petit nombre de processeurs et de cœurs.

- En général, avec un nombre réduit de cœurs, il y a vraiment une augmentation
- Puis il y a un aplatissement

Loi d'Amdahl

```
L'amélioration des performances d'un système qui peut être obtenue en optimisant une
certaine partie du système est limitée par la fraction de temps pendant laquelle
cette partie est réellement utilisée
```

C'est-à-dire que la partie du code qui ne peut pas être parallélisée pénalise l'ensemble du programme.

Problème : tous les algorithmes ne sont pas parallélisables !

Parallélisation

Définition: Exécution d'un algorithme via plusieurs flux simultanés

Tous les algorithmes ne sont pas parallélisables

Parallélisable :

- Calcul de la somme d'un vecteur

Non parallélisable : Calculer les chiffres de

- Calculer les chiffres de $\sqrt{2}$.

Parallélisation

De nombreuses travailles ont été menées pour tenter de paralléliser les algorithmes

- Trouver des expédients mathématiques
- Calculer des solutions approximatives

Problème entendu dans l'**apprentissage machine**

- Entraînement d'un réseau neuronal à l'aide de nombreux cœurs (et nœuds)
  - Problème résolu
- Algorithmes de **concentration** parallèles
  - Problème partiellement ouvert

Le problème des sections critiques

Les threads partagent la mémoire

- Ils peuvent partager des informations en utilisant des **variables partagées**.

L'accès aux variables partagées doit être synchronisé

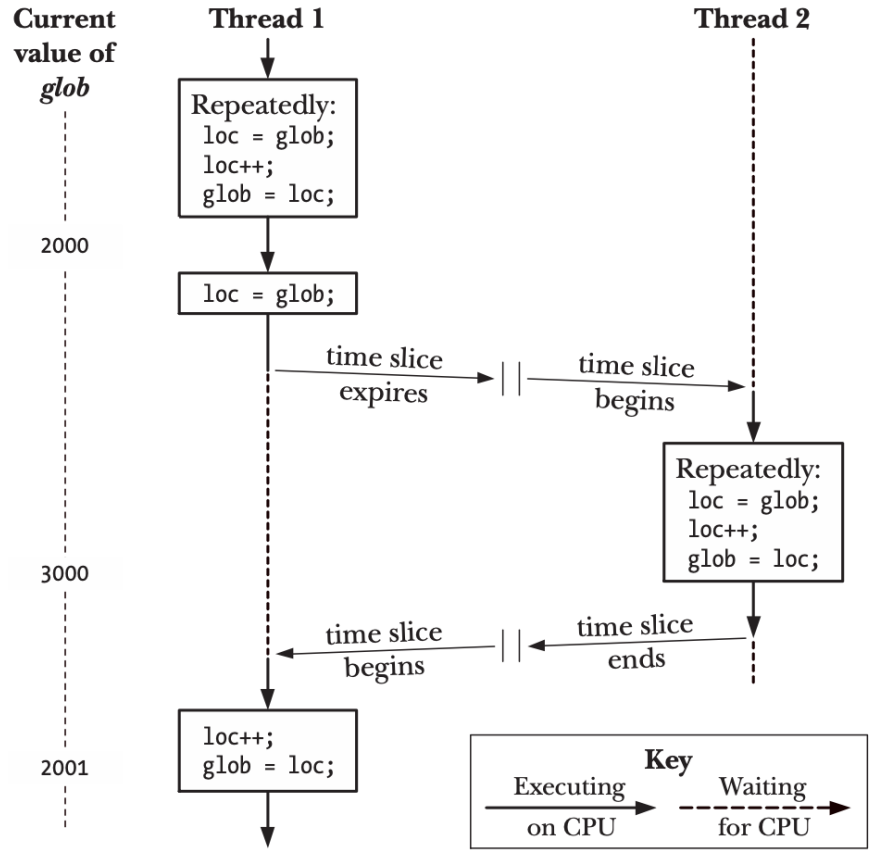
- Deux threads **ne doivent pas** y écrire en même temps
- Un thread **ne doit pas** lire une variable partagée pendant qu'un autre y écrit

Le problème des sections critiques

Imaginons deux threads exécutant le code suivant :

```
static int glob = 0 ;
static void * threadFunc(void *arg){
     int loops = *((int *) arg) ;
     int loc, j ;
     for (j = 0 ; j < loops ; j++) {
          loc = glob ;
          loc++ ;
          glob = loc ;
     }
     retour NULL ;
}
```

Quoi peut se produire ?



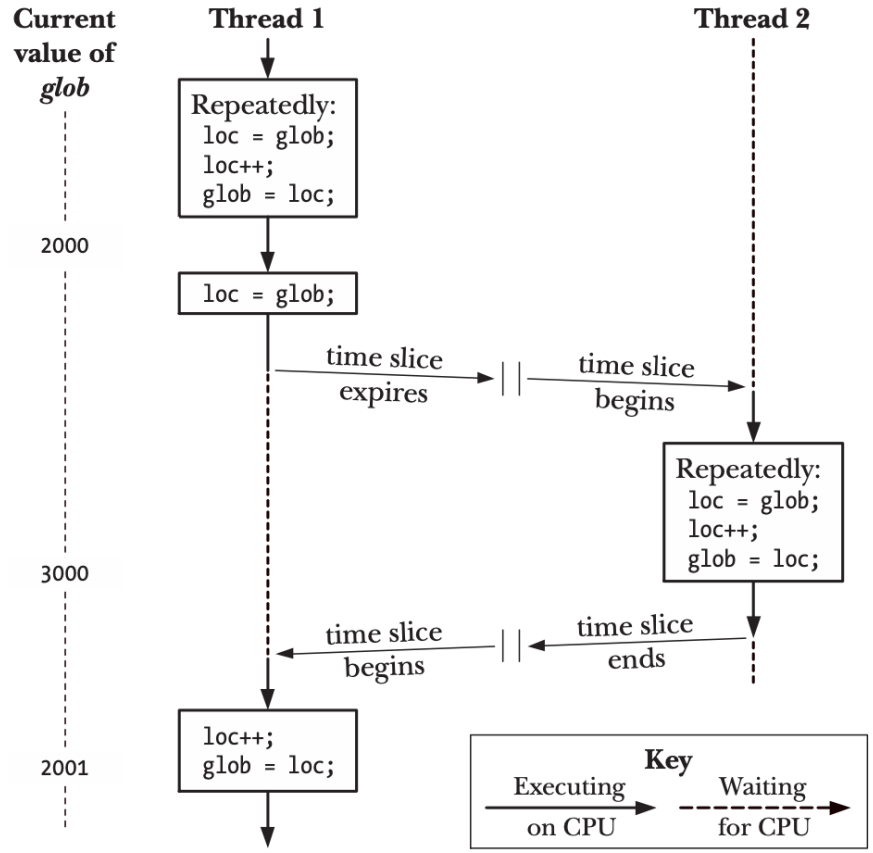
Le problème des sections critiques

Le code suivant produit des résultats imprévisibles.

Exemple :

- Le thread 1 est interrompu pendant l'incrémentation
- Le thread 2 termine l'incrémentation
- Le thread 1 termine l'incrémentation

L'incrémentation effectuée par le Thread 2 **est perdue!**.



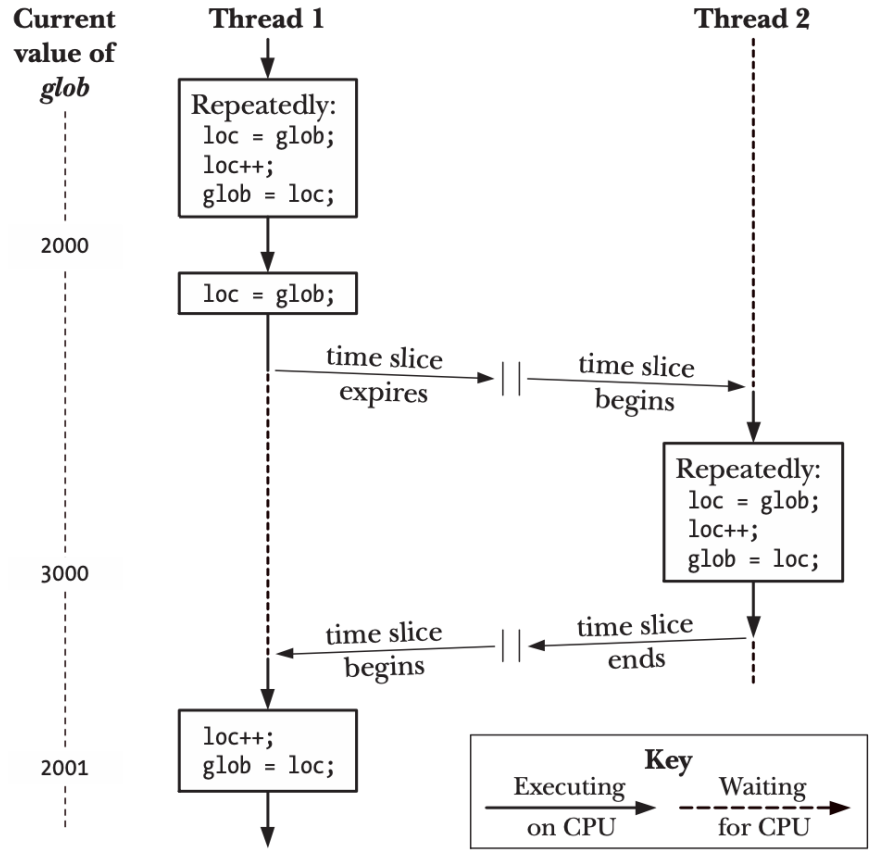
Il problema delle sezioni critiche

Observations

```
loc = glob;
loc++;
glob = loc;
```

Avec `glob++;` le problème n'est pas résolu.
Dans de nombreux processeurs (par exemple, ARM), ils n'ont pas d'instruction d'incrémentation

- Le compilateur traduit `glob++;` en instructions d'assemblage équivalentes aux 3 lignes de code ci-dessus



Définition de la section critique

Une **Critical Section** est une section de code dont l'exécution doit être atomique

- Ne peut pas être interrompu par un autre thread
- Aucun autre thread ne peut exécuter ce code en même temps

Une section critique accède aux ressources partagées

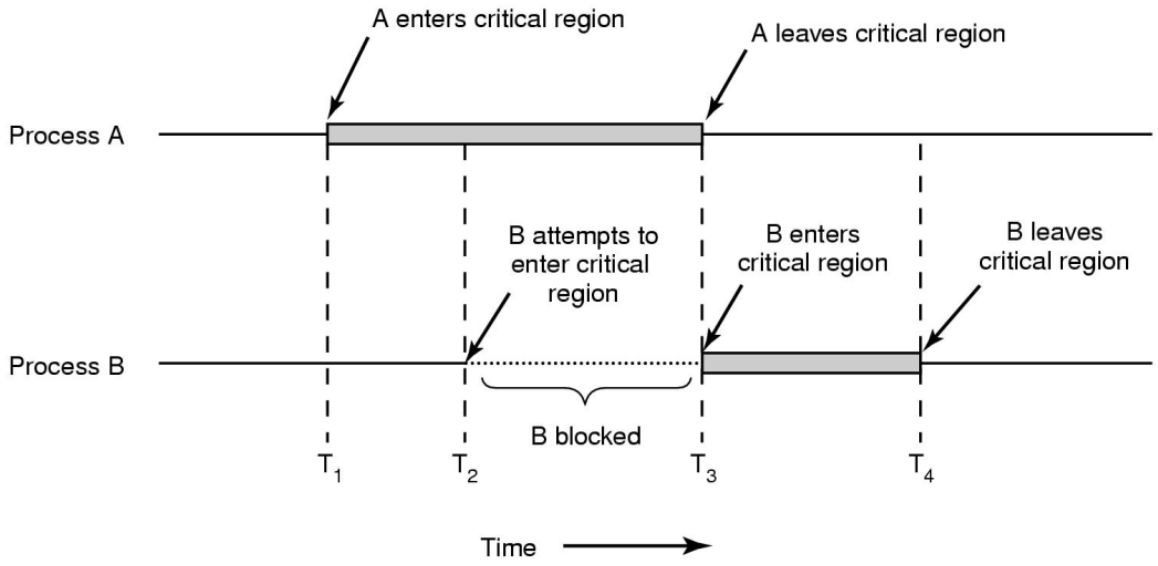
- Un seul thread peut y accéder à la fois

Les sections critiques sont également appelées **Régions critiques**

Fonctionnement de la section critique

L'accès à une section critique s'effectue en **exclusion mutuelle**

- Un thread **se réserve** pour l'accès
  - Si la section critique n'est pas utilisée, le thread y entre
  - Sinon attendez qu'il soit libre
- A la fin le thread dans la section critique **libère** la section



Mutex

Un **Mutex** est une construction de synchronisation qui gère l'accès à une section critique

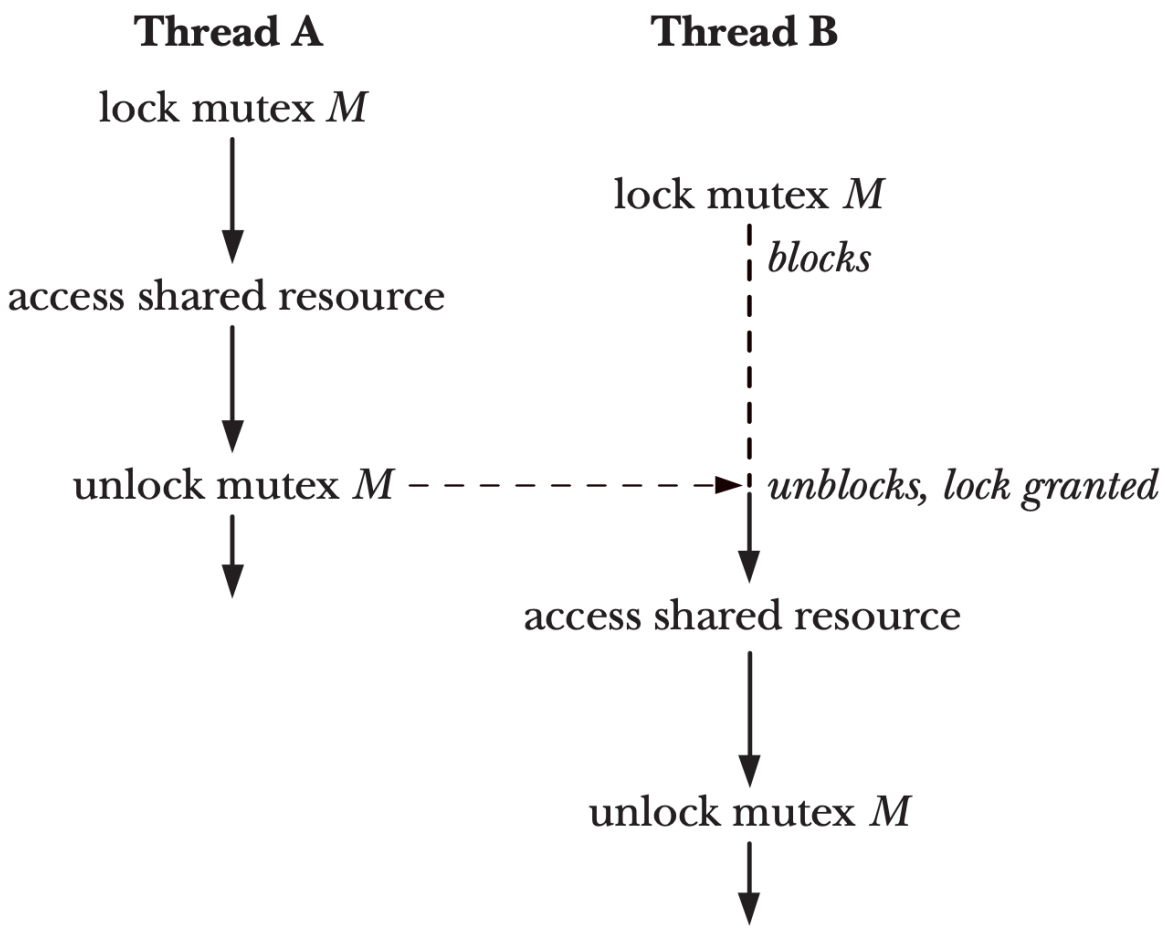
Un mutex a deux états

- **Locked** : la section est occupée
- **Free** : la rubrique est libre

Un thread peut effectuer deux actions sur un mutex :

- **Lock** : réservez l'accès pour occuper la section critique
- **Release/Unlock** : libérer la section critique

Mutex


Thread A
Thread B
lock mutex M
lock mutex M
blocks
access shared resource
unlock mutex M
unblocks, lock granted
access shared resource
unlock mutex M

Mutex

Dans Pthreads

Les mutex sont des variables de type `pthread_mutex_t`.

- Ce sont généralement des variables globales
- Initialisées par la variable `main`.
- Utilisées par n'importe quel thread

Nécessaire pour inclure :

```
#include <pthread.h>
```

Utilisé avec les fonctions de la bibliothèque `pthread_mutex_*`.

Initialisation

```
#include <pthread.h>
int  pthread_mutex_init(pthread_mutex_t * mutex , const pthread_mutexattr_t * attr ) ;
```

Initialise le mutex `mutex`, qui est passé par référence (type `pthread_mutex_t *`)
L'argument `attr` spécifie des attributs, que nous ne verrons pas

- Peut être `NULL`.

Valeur de retour, comme dans toutes les fonctions Pthread (omise dans les diapositives suivantes) :

- 0 en cas de succès
- Le code d'erreur dans le cas contraire

Lock

```
#include <pthread.h>
int  pthread_mutex_lock(pthread_mutex_t * mutex ) ;
```

Acquiert le *lock* du mutex

- Lock l'appelant jusqu'à ce que le mutex devienne libre

Release

```
#include <pthread.h>
int  pthread_mutex_unlock(pthread_mutex_t * mutex ) ;
```

Libère le mutex

Note : Le `mutex` est **toujours** passé par référence !

Variantes de *Lock*

```
#include <pthread.h>
int  pthread_mutex_trylock  ( pthread_mutex_t *mutex);
```

- Si le lock est déjà détenu par quelqu'un d'autre échoue avec l'erreur (valeur de retour) `EBUSY`

Destruction

```
#include <pthread.h>
int  pthread_mutex_destroy  ( pthread_mutex_t *mutex );
```

Libérer la mémoire occupée par le mutex

Cette mutex ne sera plus utilisable

Exemple 1/2

Implémentation du programme précédent (incrémentation d'une variable de deux threads en parallèle) en utilisant un mutex

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdlib.h>
#include <pthread.h>

static int glob = 0;
static pthread_mutex_t mtx;

static void * threadFunc(void *arg){
    int loops = *((int *) arg);
    int loc, j;
    for (j = 0; j < loops; j++) {
        pthread_mutex_lock(&mtx); /* LOCK */
        loc = glob; /* ┬ */
        loc++; /* | Critical Section */
        glob = loc; /* ┴ */
        pthread_mutex_unlock(&mtx); /* RELEASE */
    }
    return NULL;
}
int main(int argc, char *argv[]){
    pthread_t t1, t2;
    int loops = 10000000;

    pthread_mutex_init(&mtx, NULL);
    pthread_create(&t1, NULL, threadFunc, &loops);
    pthread_create(&t2, NULL, threadFunc, &loops);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_mutex_destroy(&mtx);
    printf("glob = %d\n", glob);
    exit(0);
}
```

Exemple 2/2

Le programme sans l'utilisation de mutex :

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

static int glob = 0;

static void * threadFunc(void *arg){
     int loops = *((int *) arg);
     int loc, j;
     for (j = 0; j < loops; j++) {
          loc = glob; /* ┬ */
          loc++; /* | Critical Section */
          glob = loc; /* ┴ */
     }
     return NULL;
}

int main(int argc, char *argv[]){
     pthread_t t1, t2;
     int loops = 10000000;
     pthread_create(&t1, NULL, threadFunc, &loops);
     pthread_create(&t2, NULL, threadFunc, &loops);
     pthread_join(t1, NULL);
     pthread_join(t2, NULL);
     printf("glob = %d\n", glob);
     exit(0);
}
```

La somme n'est pas 20000000, mais un nombre inférieur (par exemple, 10493368). **Pouquoi ?**

Deadlocks

Un **Deadlock** est une situation dans laquelle deux ou plusieurs threads sont bloqués

- Chacun attend une condition qui ne peut jamais se produire
- Le programme cesse de s'exécuter

Lorsque l'on utilise deux mutex ou plus, de telles situations peuvent se produire

- Il est nécessaire pour le programmeur de les prévoir et de les éviter

Exemple

Thread A:

```
pthread_mutex_lock(mutex1) ; // <--- LOCK 1
pthread_mutex_lock(mutex2) ; // <--- LOCK 2
... Section critique ...
pthread_mutex_unlock(mutex2) ;
pthread_mutex_unlock(mutex1) ;
```

Thread B:

```
pthread_mutex_lock(mutex2) ; // <--- LOCK 2
pthread_mutex_lock(mutex1) ; // <--- LOCK 1
... Section critique ...
pthread_mutex_unlock(mutex1) ;
pthread_mutex_unlock(mutex2) ;
```

Les deadlocks

Comment éviter les blocages :

- Utiliser d'autres types de synchronisation lorsque c'est possible :
  - Pipe, FIFO
- Utiliser un faible nombre de mutex
- Modélisation de l'utilisation de nombreux mutex
  - Techniques basées sur les graphes
  - Nous ne voyons pas dans ce cours

Nous allons étudier le problème des deadlocks en détail

Les sémaphores

Un **sémaphore** est un **entier positif** partagé par plusieurs threads

- Initialisé à une certaine valeur au moment de la création

Les threads simultanés peuvent faire deux choses :

- **Augmentater de** 1
- **Décrémenter de** 1

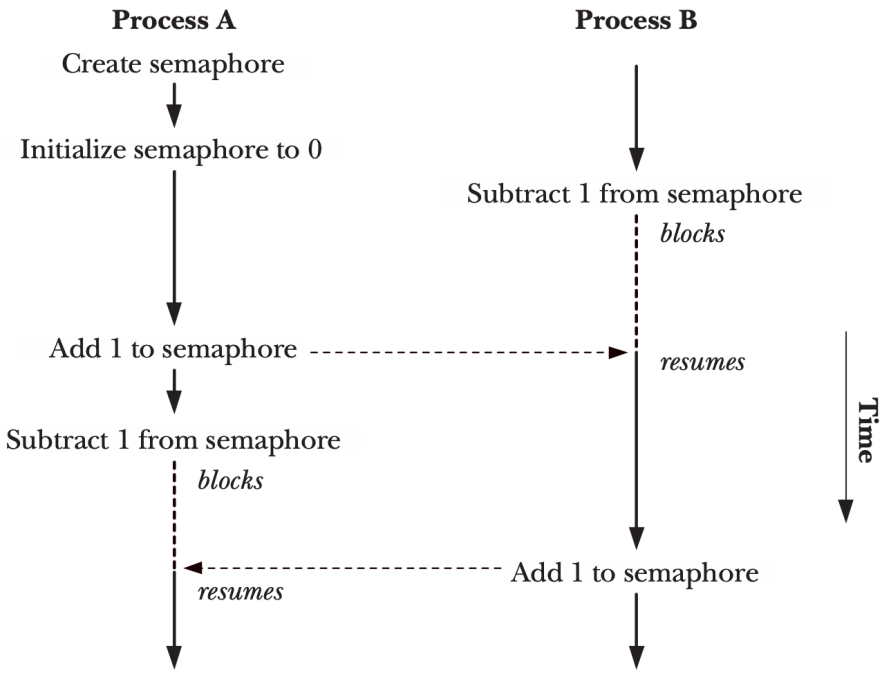
Le sémaphore ne peut **jamais** prendre des valeurs **négatives**.
Si la décrémentation fait que le sémaphore devient négatif

- Le thread **se bloque**
- **En attente** qu'un autre thread fasse un incrément

Opération

Exemple :

1. Le sémaphore est initialisé à 0
2. B decrement
   - Le sémaphore ne peut pas prendre de valeurs négatives
   - B est mis en attente
3. Incréments A
   - B se libere
   - Le sémaphore a pour valeur 0
4. Décréments de A
   - A se bloque
5. Incréments de B
   - A se libere
6. Le sémaphore a pour valeur 0



Histoire

Ils constituent une construction de synchronisation simple, puissante et flexible

- Inventé par Dijkstra en 1965
- Utilisé à diverses fins dans tous les langages de programmation et systèmes d'exploitation

Sous Linux, deux implémentations

- *Sémaphores System V* : plus anciens, plus complexes. Nous ne verrons pas
- ***Sémaphores POSIX*** : **on voie dans les diapositives**
  REMARQUE : ils peuvent également être utilisés entre différents processus (et pas seulement entre les threads du même processus)

Sémaphores nommés et non nommés

Les *sémaphores POSIX* peuvent être :

- ***Named*** : ils ont un nom unique. Ils peuvent être utilisés par plusieurs processus indépendants (même non liés).
- ***Unnamed*** : ils n'ont pas de nom. Ils peuvent être partagés entre :
  - les threads, sans dispositions particulières
  - Les processus : s'ils sont créés via `fork` et résident dans une zone de mémoire partagée (avec `shmget` ou `mmap`).

Sémaphores nommés et non nommés

Le principe de fonctionnement est le même :

1. Le sémaphore est créé/internalisé
2. Les processus/filières peuvent faire :
    - *Post* pour incrémenter le feu tricolore
    - *Wait* pour décrémenter le sémaphore (et éventuellement attendre)
3. Le sémaphore est détruit/fermé

Sémaphores nommés

Les fonctions suivantes sont utilisées :

1. `sem_open()`
2. `sem_post(sem)`, `sem_wait(sem)` et `sem_getvalue()`
3. `sem_close()` et `sem_unlink()`.

Nécessité d'inclure l'en-tête :

```
#include <semaphore.h>
```

Les sémaphores sont des handles opaques :

```
sem_t
```

Sémaphores nommés - Création

```
#include <fcntl.h> /* Définit les constantes O_* */
#include <sys/stat.h> /* Définit les constantes de mode */
#include <semaphore.h>

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

Crée un sémaphore nommé `name`.

- Doit commencer par `/`
- Peut être n'importe quel identifiant
  Exemple : `/mysem`

Sémaphores nommés - Création

```
#include <fcntl.h> /* Définit les constantes O_* */
#include <sys/stat.h> /* Définit les constantes de mode */
#include <semaphore.h>

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

L'argument `oflag` spécifie ce qu'il faut faire si le sémaphore existe ou non :

- `O_CREAT` : crée et ouvre le sémaphore s'il n'existe pas. Ouvre s'il existe
- `O_CREAT | O_EXCL` : crée et ouvre. Échec s'il existe

Sémaphores nommés - Création

```
#include <fcntl.h> /* Définit les constantes O_* */
#include <sys/stat.h> /* Définit les constantes de mode */
#include <semaphore.h>

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

Arguments optionnels :

- `value` spécifie la valeur initiale
- `mode` spécifie les permissions, comme pour les fichiers

Si vous utilisez l'option `O_CREAT`, `value` doit être spécifié !

Valeur de retour : le sémaphore en cas de succès, sinon `SEM_FAILED`.

Sémaphores nommés - Fermer et détruire

```
#include <semaphore.h>
int  sem_close(sem_t * sem ) ;
int  sem_unlink(const char * name ) ;
```

`sem_close` ferme le sémaphore pour le processus en cours

`sem_unlink` supprime le sémaphore pour tous les processus

Valeur de retour : 0 en cas de succès, sinon -1

Sémaphores nommés - Opérations

```
#include <semaphore.h>
int  sem_wait(sem_t * sem ) ;
int  sem_post(sem_t * sem ) ;
```

`sem_wait` décrémente le sémaphore de 1.

- Si le sémaphore prend une valeur négative, il arrête l'appelant

`sem_post` incrémente le sémaphore de 1.

Valeur de retour : 0 en cas de succès, sinon -1.

Sémaphores nommés - Opérations spéciales

```
#include <semaphore.h>
int  sem_trywait(sem_t *sem)  ;
int  sem_getvalue(sem_t *restrict sem, int *restrict sval)  ;
```

`sem_trywait` comme `sem_wait`.

- Mais ne bloque pas au cas où sem deviendrait négatif
- Mais échoue

`sem_getvalue` place la valeur du sémaphore dans l'entier pointé par `sval`.

- Aucune garantie que la valeur restera valide !

Sémaphores nommés - Exemple

Créons deux programmes qui communiquent par l'intermédiaire d'un sémaphore.

- Le premier crée un `post` chaque fois que l'utilisateur appuie sur *Enter*.
- Le second imprime une chaîne de caractères chaque fois que le premier fait un `post`.

Sémaphores nommés - Exemple

Programme 1

```c
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <errno.h>
#include <semaphore.h>
#include <string.h>

int  main(int argc, char *argv[]){
     sem_t * s ;

     s = sem_open("/semaphore", O_CREAT , S_IRUSR | S_IWUSR, 0) ;
     if(s == SEM_FAILED) {
          printf("Error creating/opening the semaphore %s\n", strerror(errno)) ;
          exit (1) ;
     }

     while(1){
          printf("Press enter for a post : ") ;
          getchar() ;
          sem_post(s) ;
     }
     sem_close(s) ; /* Code inaccessible*/
     return 0 ;
}
```

Sémaphores nommés - Exemple

Programme 2

```c
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <errno.h>
#include <semaphore.h>
#include <string.h>

int  main(int argc, char *argv[]){
     sem_t * s ;
     int  i = 0 ;

     s = sem_open("/semaphore", O_CREAT , S_IRUSR | S_IWUSR, 0) ;
     if(s == SEM_FAILED) {
          printf("Error creating/opening the semaphore %s\n", strerror(errno)) ;
          exit  (1) ;
     }

     while(1){
          sem_wait(s) ;
          printf("Wait %d effettuata\n", i) ;
          i++ ;
     }
     sem_close(s) ;  /* Code inaccessible*/
     return  0 ;
}
```

Sémaphores nommés - Exemple

Observations :

- La valeur du sémaphore est persistante. Si le programme 2 n'est pas exécuté, la valeur du sémaphore peut augmenter
- Plusieurs instances des deux programmes peuvent être exécutées
  - Plusieurs instances du programme 1 accumulent de la valeur dans le sémaphore.
  - S'il y a plusieurs instances du programme 2, une seule peut être déverrouillée par incrément.
  - Le système d'exploitation a tendance à être *équitable*. Il répartit la charge entre plusieurs sémaphores en attente

Sémaphores sans nom

Ils sont utilisés d'une manière similaire, mais plus simple que les *sémaphores nommés*.

Différentes procédures d'ouverture et de fermeture

```
#include <semaphore.h>
int  sem_init(sem_t * sem , int pshared , unsigned int value ) ;
```

Crée le sémaphore et le place dans `sem`, initialisé à `value`.

Sémaphores sans nom

```
#include <semaphore.h>
int  sem_init(sem_t * sem , int pshared , unsigned int value ) ;
```

Important :

`sem_open` renvoie un pointeur de sémaphore (`sem_t *`), qui est alloué par la bibliothèque

`sem_init` place le pointeur de sémaphore dans `sem` .

- Le programmeur doit décider où allouer le sémaphore, de type `sem_t` .
- Il peut s'agir d'une variable globale, locale, allouée dynamiquement ou sur une région de mémoire partagée

Sémaphores sans nom

```
#include <semaphore.h>
int  sem_init(sem_t * sem , int pshared , unsigned int value ) ;
```

Si `pshared` est 0, `sem` n'est pas partagé entre les processus, seulement entre les threads

- `sem` peut être une variable globale commune

Si `pshared` est $\neq 0$, le sémaphore est partagé entre les processus (via `fork`)

- `sem` doit être dans une zone de mémoire partagée

Conséquence : il est préférable d'utiliser les sémaphores nommés dans les applications multiprocessus.

Sémaphores sans nom

```
#include <semaphore.h>
int  sem_destroy(sem_t * sem );
```

Détruit le sémaphore `sem`.

S'il est partagé entre les processus, tous les processus doivent appeler `sem_destroy`

Remarque : `sem_close` et `sem_unlink` ne sont utilisés qu'avec les *sémaphores nommés*

Utilisation

Utilisez `sem_post()` et `sem_wait()` comme pour *Sémaphores nommés*

Sémaphores sans nom - Exemple

Créez un programme avec deux threads. Le premier envoie chaque seconde un message au second.
La seconde l'imprime.

```
#include <stdio.h>
#include <unistd.h>
#include <semaphore.h>
#include <pthread.h>

sem_t s_ecriture, s_lecture;
char buffer [50];

void * sender(void *arg){
    ...
}

void receiver(){
    ...
}

int main(int argc, char *argv[]){
    pthread_t t;
    sem_init(&s_ecriture, 0, 0);
    sem_init(&s_lecture, 0, 1);
    pthread_create(&t, NULL, sender, NULL);
    receiver();
}
```

Sémaphores sans nom - Exemple

Il faut éviter qu'un thread lise pendant qu'un autre écrit

- Une string incohérente pourrait être lue !
- Sans terminateur !

Il faut deux semaphores :

- `s_ecriture` notifie que `sender` a terminé une écriture
  - `sender` met un *token* quand il a fini d'écrire, `receiver` attend le token pour commencer à lire
- `s_lecture` notifie que `receiver` a fini de lire
  - `receiver` met un *token* à la fin de la lecture, `sender` attend le token commence une nouvelle écriture

`s_ecriture` doit être initialisé à 0 pour que `receiver` attende la première écriture
`s_lecture` doit être initialisé à 1 pour que `sender` fasse la première écriture

Sémaphores sans nom - Exemple

Sender et Receiver:

```
void * sender(void *arg){
    int i = 0;
    while (1){
        sem_wait(&s_lecture);
        sprintf(buffer, "Message %d\n", i);
        sem_post(&s_ecriture);
        i++;
        sleep(1);
    }
}

void receiver(){
    while (1){
        sem_wait(&s_ecriture);
        printf("Received: %s\n", buffer);
        sem_post(&s_lecture);
    }
}

...
sem_init(&s_ecriture, 0, 0);
sem init(&s lecture, 0, 1);
```

Classical IPC Problems

- The operating systems literature is full of interesting problems that have been widely discussed and analyzed using a variety of synchronization methods
- We look at two examples:
  - The readers and writers problem
  - The dining philosophers problem

The readers and writers problem

L'une des tâches les plus courantes dans les systèmes concurrents est illustrée par le problème producteur-consommateur. Dans ce problème, les threads ou processus sont divisés en deux types relatifs : un thread producteur est responsable de l'exécution d'une tâche initiale qui aboutit à la création d'un résultat et un thread consommateur qui utilise ce résultat initial pour une tâche ultérieure. Entre les threads/processus, il y a une mémoire partagée ou une file d'attente qui stocke les résultats transmis. L'une des principales caractéristiques de ce problème est que le consommateur retire les données de la file d'attente et les consomme en les utilisant ultérieurement. Il n'existe aucun moyen pour le ou les processus/threads consommateurs d'accéder de manière répétée aux mêmes données de la file d'attente.

Readers and writers problem

A logical solution

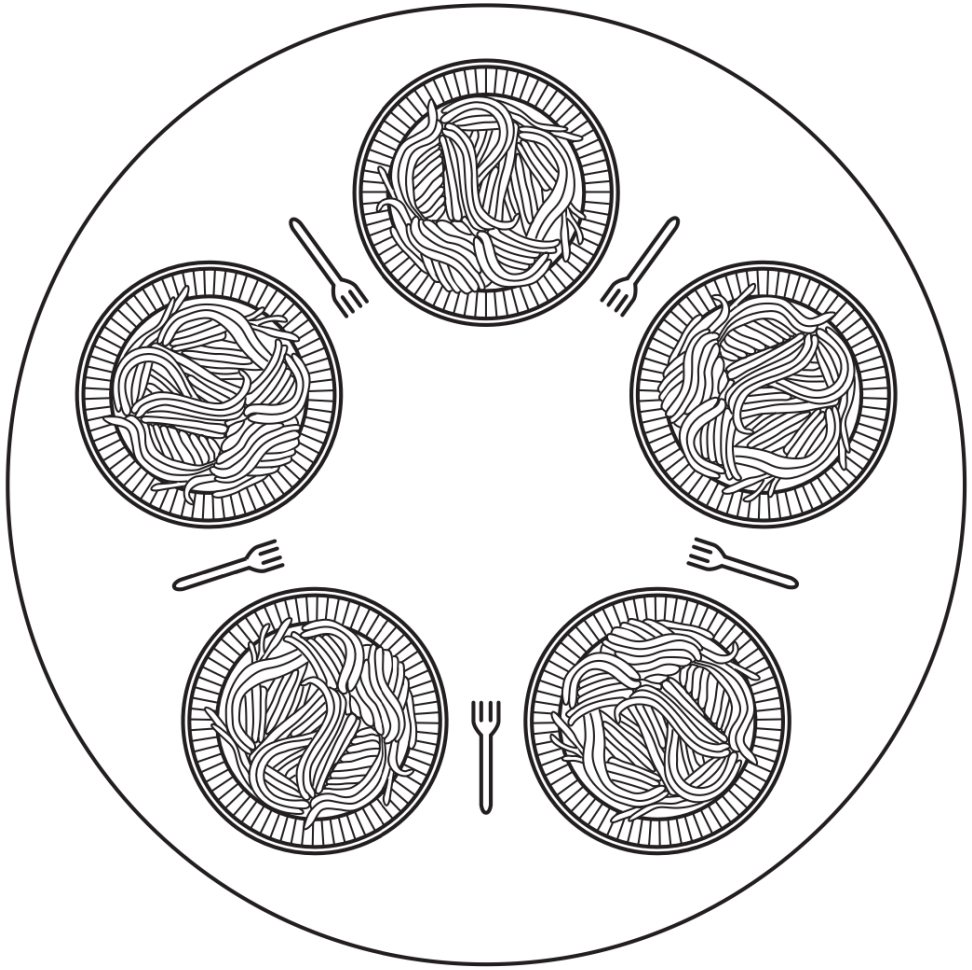
```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
        while (TRUE) {                  /* repeat forever */
                down(&mutex);           /* get exclusive access to 'rc' */
                rc = rc + 1;            /* one reader more now */
                if (rc == 1) down(&db); /* if this is the first reader ... */
                up(&mutex);             /* release exclusive access to 'rc' */
                read_data_base( );      /* access the data */
                down(&mutex);           /* get exclusive access to 'rc' */
                rc = rc – 1;            /* one reader fewer now */
                if (rc == 0) up(&db);   /* if this is the last reader ... */
                up(&mutex);             /* release exclusive access to 'rc' */
                use_data_read( );       /* noncritical region */
        }
}
```

Readers and writers problem

A logical solution (part 2)

```
void writer(void)
{
        while (TRUE) {                  /* repeat forever */
                think_up_data( );       /* noncritical region */
                down(&db);              /* get exclusive access */
                write_data_base( );     /* update the data */
                up(&db);                /* release exclusive access */
        }
}
```

Dinining philosophers problem



Dinining philosophers problem

A solution...

```
#define N 5          /*number of philosophers*/

void  philosopher(int i)  {    /*i: philosopher number, from 0 to 4*/

   while(TRUE) {
      think();                 /*philosopher is thinking*/
      take_fork(i);            /*take left fork*/
      take_fork(i+1)%N;        /*take right for;% is modulo operator*/
      eat():                   /*self-explanatory*/
      put_fork(i);             /*put left fork back on table*/
      put_fork(i+1)%N;         /*put right fork back on table*/
   }

}
```

Does not work. Why?

Dinining philosophers problem

You will see the working solution on Friday 😄