Programmation Système et Réseau

5 – Multithreading : Thread POSIX

Mohamed Maachaoui
Juan Ángel Lorenzo del Castillo
Seytkamal Medetov
Son Vu

CY Tech
ING2 – GSI
2020 – 2021

Introduction

Il est possible d'implémenter des algorithmes parallèles avec ce qu'on a vu jusque là (IPC), mais c'est lourd :

- **fork** – appel système lourd :
  - les processus fils et le processus père sont des processus indépendants
  - chaque processus a son espace d'adressage
  - chaque processus a sa pile d'exécution
  - changement de contexte (context switch) lent/coûteux
- communication inter-processus généralement lente
- partage de données délicat

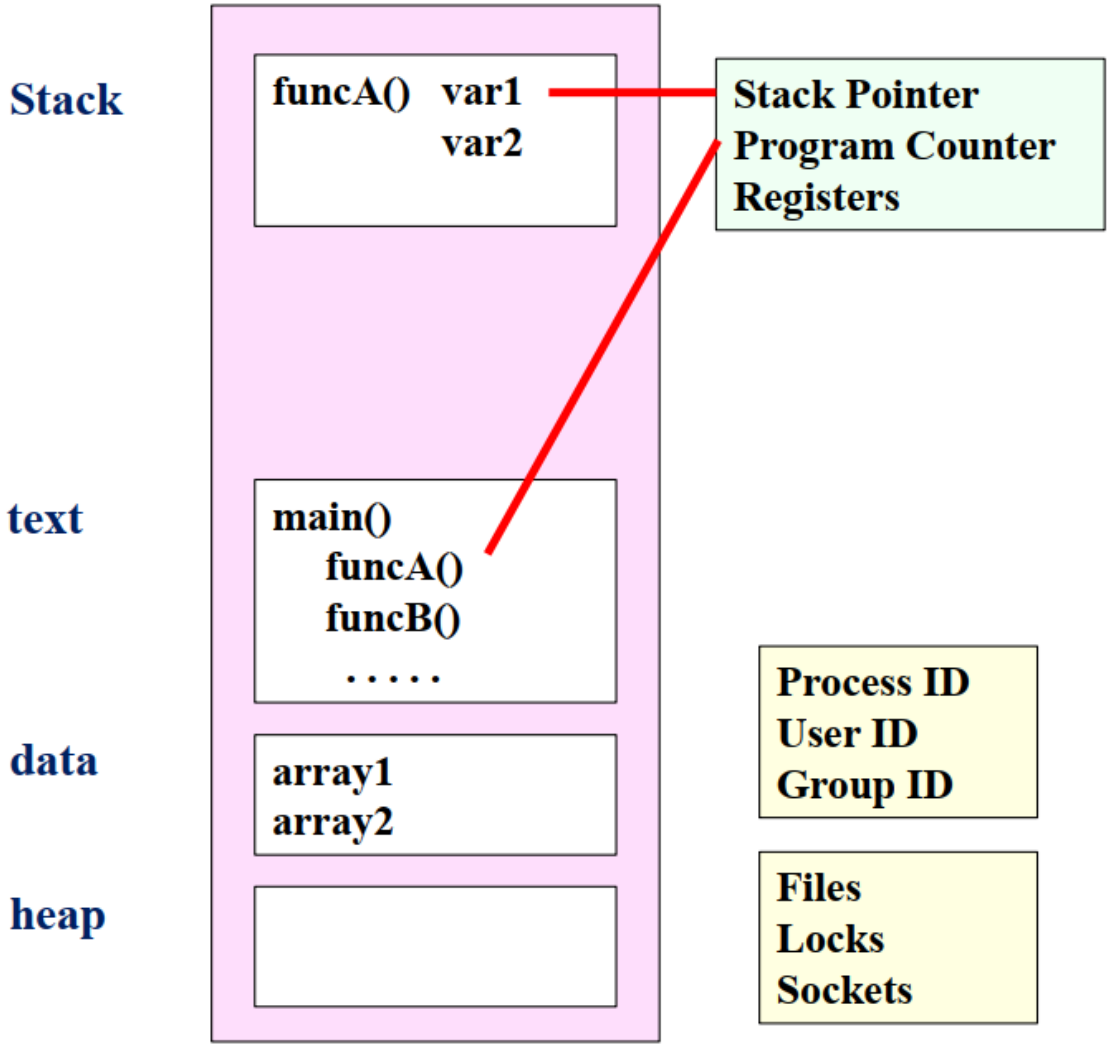
Solutions : les processus légers : « threads »

Processus légers : *threads*

- Un processus qui crée des *threads* ne crée pas d'autres processus, tout se passe dans le même espace d'adressage
  - un thread est une « partie » d'un processus
  - un processus est l'exécution d'un ensemble (>=1) de threads
- Chaque *thread* est associée à une pile d'exécution indépendante
- Différence entre un ensemble de *threads* et un ensemble de processus :
  - les *threads* partagent pratiquement tout :
    - ✓ les variables globales, les variables statiques locales, les descripteurs de fichiers ouverts, le PID, le PPID, les utilisateurs propriétaires, les handlers de signaux, le répertoire courant, le masque de fichiers, …)
  - ne partagent pas :
    - ✓ les identifiants des threads, la pile, le masque des signaux, errno...
  - ce partage implique la gestion par le programmeur de la concurrence d'accès à ces variables (c-a-d., la synchronisation n'est plus gérée au niveau du système, mais est laissée à l'utilisateur)

CY TECH

Processus légers : *threads*

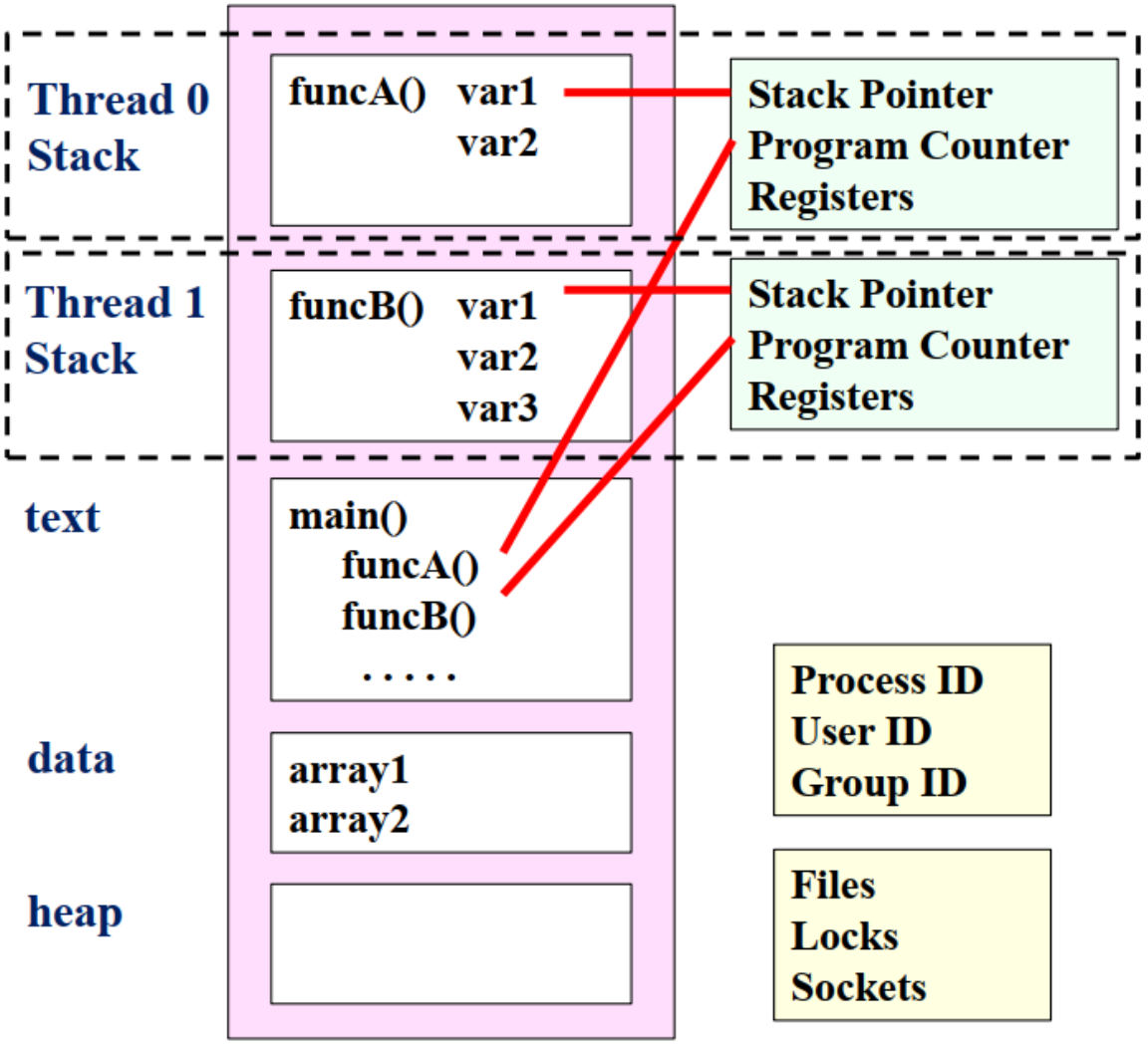


Processus

- Une instance d'exécution d'un programme.

- Le contexte d'exécution d'un programme en cours d'exécution... c'est-à-dire les ressources associées à l'exécution d'un programme.

Processus légers : *threads*



Threads :

- Les fils sont des "processus légers".

- Les threads partagent l'état du processus entre plusieurs threads... ce qui réduit considérablement le coût du changement de contexte.

Threads: avantages et inconvénients

- Avantages et inconvénients par rapport à des processus communiquants avec les « anciens » mécanismes
  - partage de la mémoire : mécanisme rapide de communication inter-thread
  - plus léger : moins de données système à recopier
  - plus rapide : le context-switch (le changement de contexte) est plus facile
- Les inconvénients sont (uniquement) des « difficultés » de programmation:
  - les threads utilisent les mêmes copies des librairies : les librairies doivent être « MT-safe »
  - il faut gérer la synchronisation (mutex, sémaphores...)
- En cas de mauvaise synchronisation :
  - comportement « aléatoire » : bugs, segfaults, exploitations…
  - interblocage...

CY TECH

Threads: Modèles d'implantation

- **Modèle M:1**
  - Tous les threads d'un processus mappés sur un seul thread noyau
  - Gestion des threads au niveau utilisateur
- **Modèle 1:1**
  - Chaque thread utilisateur mappé sur un thread noyau
  - Modification du noyau pour un support aux threads
  - Gestion des threads au niveau système (LightWeight Process – LWP)
- **Modèle M:N**
  - M threads utilisateur mappés sur N threads noyau ($M \geq N \geq 1$)
  - Services utilisateur basés sur des services noyau
  - Coopération entre l'ordonnanceur noyau et un ordonnanceur utilisateur

CY TECH

Créer un thread

Au départ, le processus est constitué d'un unique *thread* (**thread principal**) : celui qui exécute la fonction main :

- Création d'un processus : création d'un ***thread natif***
- Exécution du **main** dans ce *thread principal*
- Si terminaison du thread principal ⇒ terminaison des autres threads du processus

Créer un thread

Un *thread (POSIX)* concurrent est créé dynamiquement dans le processus auquel le thread appelant appartient grâce à la fonction suivante :

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

- En cas de réussite, pthread_create() renvoie 0 ; en cas d'erreur, elle renvoie un numéro d'erreur, et le contenu de *thread est indéfini
- Erreurs :
  - **EAGAIN** - ressources insuffisantes pour créer un nouveau thread, ou une limite sur le nombre de threads imposée par le système a été atteinte. Ce dernier cas peut arriver de deux façons :
    - la limite souple RLIMIT_NPROC (changée par setrlimit(2)), qui limite le nombre de processus pour un identifiant d'utilisateur réel, a été atteinte ;
    - ou alors la limite imposée par le noyau sur le nombre total de threads, /proc/sys/kernel/threads-max, a été atteinte.
  - **EINVAL** - Paramètres invalides dans attr.
  - **EPERM** - Permissions insuffisantes pour définir la politique d'ordonnancement et les paramètres spécifiés dans attr.
  - page man (https://man.developpez.com/man3/pthread_create/)

Créer un thread

Un *thread (POSIX)* concurrent est créé dynamiquement dans le processus auquel le thread appelant appartient grâce à la fonction suivante :

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

- Ses arguments sont suivantes :
  - **thread** – pointeur sur le thread créé – l'adresse mémoire où sera copié l'identifiant du nouveau thread
  - **attr** – les attributs du nouveau thread créé (ressources système, priorité, …)
    - ✓ Si attr est égal NULL, les attributs par défaut sont utilisé : le thread créé est joignable (non détaché) et utilise la politique d'ordonnancement normale (pas temps-réel)
  - **start_routine** – la fonction exécuté par le thread (la fonction d'entrée du thread)
  - **arg** – le première argument passé à la fonction **start_routine**

- Un **pthread** n'a pas de PID propre (ils partagent tous le même PID, celui du processus contenant les threads)
- L'identifiant d'un **pthread** est un objet du **pthread_t** : la norme ne dit pas ce qu'il y a dans **pthread_t** (*objet opaque*).

Terminaison et retour d'un pthread

- Similairement à un processus, un thread renvoie un code retour : un pointeur
- Similairement à un fils qui devient zombi, le code retour du thread est gardée en mémoire jusqu'à ce qu'un autre thread le « rejoigne »
- Il n'y a pas de notion de *père/fils* :
  - n'importe quel thread peut joindre n'importe quel thread
  - si plusieurs attentes du même thread : comportement indéfini
- Achèvement d'un thread :
  - Quand **start_routine** termine, le thread se termine
  - Un thread peut également terminer avec **pthread_exit()**
- Terminaison d'un processus (mono ou multi-thread) :
  - Le thread qui exécute la routine **main** est spécial, sa terminaison termine le processus, même si d'autres threads sont encore en exécution
  - l'appel **exit()** dans n'importe quel thread termine le processus (i.e. tous les threads)
  - lorsqu'un signal provoquant la terminaison du processus est reçu

Primitives liées aux threads

- void **pthread_exit(**int ***status)** : termine le thread appelant avec une valeur de retour égale à **status** ; qui peut être consulté par un autre thread en utilisant **pthread_join()**
- int **pthread_kill(**pthread_t **thread,** int **sig)** : permet d'envoyer un signal à un thread (concept Unix, pas les signaux des moniteurs)
- int **pthread_join(**pthread_t **thread,** void ****retval)** : suspend le thread appelant jusqu'à terminaison du **thread** désigné (soit après avoir été annulé, soit terminé en appelant **pthread_exit()**); retval – un pointeur sur une variable (contenant un pointeur), où le code de retour sera copié
- pthread_t **pthread_self(**void**)** : retourne l'identificateur du thread
- int **pthread_detach(**pthread_t **thread)** : place un thread en cours d'exécution dans l'état détaché sauf s'il un autre thread a déjà joint ce thread :
  - Cela garantie que les ressources mémoire consommées par thread seront immédiatement libérées lorsque l'exécution de thread s'achèvera
  - Cela empêche les autres threads de se synchroniser sur la mort de thread en utilisant pthread_joint()
  - Si on ne veut pas avoir à gérer la fin d'un thread, on peut le « détacher »

Un exemple d'utilisation des threads

```
...
void *my_thread(void * arg){
  int i;
  for (i = 0 ; i < 5 ; i++) {
                printf ("Thread %s: %d\n", (char*)arg, i);
                sleep (1); //essayer de commenter cette ligne
   }
  pthread_exit (0);
}

int main (int ac, char **av){
  pthread_t th1, th2;
  void *ret;
  if (pthread_create (&th1, NULL, my_thread, "1") < 0) {
                fprintf (stderr, "pthread_create error for thread 1\n");
                exit (1);
  }
  if (pthread_create (&th2, NULL, my_thread, "2") < 0) {
                fprintf (stderr, "pthread_create error for thread 2\n");
                exit (1);
  }
  (void)pthread_join (th1, &ret);
  (void)pthread_join (th2, &ret);
}
```

La fonction **pthread_create** permet de créer le thread et de l'associer à la fonction **my_thread**. On notera que le paramètre **void *arg** est passé au thread lors de sa création. Après création des deux threads, le programme principal attend la fin des threads en utilisant la fonction **pthread_join**.

Après compilation de ce programme avec option **-lpthread**, il donne à l'exécution:

./thread_ex1
Thread 1: 0
Thread 2: 0
Thread 1: 1
Thread 2: 1
Thread 1: 2
Thread 2: 2
Thread 1: 3
Thread 2: 3
Thread 1: 4
Thread 2: 4

Commenter la fonction **sleep()**, et essayer plusieurs fois, ...

Primitives de manipulation de l'argument *attr*

- **pthread_attr_t** – objet (opaque) spécifiant les attributs d'un thread.
- int **pthread_attr_init(**pthread_attr_t ***attr)** : initialise l'objet d'attributs de thread pointé par **attr** avec des valeurs d'attributs par défaut.
  - Après cet appel, les attributs individuels de cet objet peuvent être modifiés en utilisant diverses fonctions, et l'objet peut alors être utilisé dans un ou plusieurs appels de pthread_create() pour créer des threads.
- int **pthread_attr_destroy**(pthread_attr_t ***attr**) : quand un objet d'attributs de thread n'est plus nécessaire, il devrait être détruit en appelant cette fonction.
- int **pthread_attr_setdetachstate**(pthread_attr_t ***attr**, int **detachstate**) : définit l'attribut d'état de détachement de l'objet d'attributs de thread auquel **attr** fait référence à la valeur indiquée par detachstate.
  - Cet attribut d'état de détachement détermine si un thread créé en utilisant l'objet d'attributs de thread attr sera dans un état joignable ou détaché. Les valeurs suivantes peuvent être spécifiées dans detachstate :
  - **PTHREAD_CREATE_DETACHED** : Les threads créés avec attr seront dans un état détaché.
  - **PTHREAD_CREATE_JOINABLE** : Les threads créés avec attr seront dans un état joignable.
- int **pthread_attr_getdetachstate**(pthread_attr_t ***attr**, int ***detachstate**) : envoie, dans le tampon pointé par detachstate, l'attribut contenant l'état de détachement de l'objet d'attributs de thread attr. [man : https://man.developpez.com/man3/pthread_create/]

Un exemple de manipulation de l'argument *attr*

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 3

void *travail(void *null) {
 int i;
 double result=0.0;
 for (i=0; i<1000000; i++)
   result = result + (double)random();
 printf("Resultat = %e\n",result);
 pthread_exit(NULL);
}

int main(int argc, char *argv[ ]) {
 pthread_t thread[NUM_THREADS];
 pthread_attr_t attr;
 int rc, t;
 /* Initialise et modifie l'attribut  */
 pthread_attr_init(&attr);
 pthread_attr_setdetachstate(&attr,
                PTHREAD_CREATE_JOINABLE);
```

```c
 for(t=0;t<NUM_THREADS;t++) {
  printf("Creating thread %d\n", t);
  if ((rc = pthread_create(&thread[t], &attr,
                           travail, NULL))) {
   printf("ERREUR de _create() : %d\n", rc);
   exit(-1);
 }}
 /* Libère l'attribut */
 pthread_attr_destroy(&attr);
 /* Attend les autres */
 for(t=0;t<NUM_THREADS;t++) {
  if ((rc = pthread_join(thread[t],
     /*pas de retour attendu*/NULL))) {
   printf("ERREUR de _join() : %d\n", rc);
   exit(-1);
  }
  printf("Rejoint thread %d. (ret=%d)\n",
         t, status);
 }
 pthread_exit(NULL);
}
```

Création des threads: Régions parallèles

```
#pragma omp parallel num_threads(4)
{
    foobar ();
}
```

```
void thunk ()
{
    foobar ();
}

pthread_t tid[4];
for (int i = 1; i < 4; ++i)
   pthread_create (
         &tid[i],0,thunk, 0);
thunk();

for (int i = 1; i < 4; ++i)
    pthread_join (tid[i]);
```

- Seulement trois threads sont créés parce que la dernière section parallèle sera invoquée à partir du thread principal.

Accès concurrent

Les threads partagent les données des processus.

- Le partage de mémoire est généralement voulu et avantageux :
  - cela évite de gaspiller de la mémoire

- L'important est de bien savoir gérer l'accès concurrent à la mémoire. Trois mécanismes de synchronisation :
  - mutex : exclusion mutuelle (verrous)
  - sémaphores : introduit lors de la 5-ème révision du standard
  - les « signaux » liés aux moniteurs du cours de Programmation concurrente

MUTEX : création/initialisation

Un mutex est un objet d'exclusion mutuelle (**MUT**ual **EX**clusion), et est très pratique pour protéger des données partagées de modifications concurrentes et pour implémenter des sections critiques.

- Un mutex peut être dans deux états :
  - déverrouillé (pris par aucun thread) ou
  - verrouillé (appartenant à un thread).
- Un mutex ne peut être pris que par un seul thread à la fois. Un thread qui tente de verrouiller un mutex déjà verrouillé est suspendu jusqu'à ce que le mutex soit déverrouillé.
- int **pthread_mutex_init**(pthread_mutex_t ***mutex**, const pthread_mutexattr_t ***mutexattr**) - allocation mémoire et initialisation dynamique. Cette primitive initialise le mutex pointé par **mutex** selon les attributs de mutex spécifié par **mutexattr**.
  - Si **mutexattr** vaut **NULL**, les paramètres par défaut sont utilisés.
  - Les variables de type **pthread_mutex_t** peuvent aussi être initialisées de manière statique, en utilisant les constantes **PTHREAD_MUTEX_INITIALIZER** (pour les mutex « rapides »), **PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP** (pour les mutex « récursifs »), et **PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP** (pour les mutex à « vérification d'erreur »)
- int **pthread_mutexattr_init**(pthread_mutexattr_t ***attr**) : initialise l'objet attributs de mutex **attr** et le remplit avec les valeurs par défaut
- int **pthread_mutex_destroy**(pthread_mutex_t ***mutex**) : détruit un mutex, libérant les ressources qu'il détient. Le mutex doit être déverrouillé.

Verrouillage/Déverrouillage d'un mutex

- Verrouillage :
- **int pthread_mutex_lock(pthread_mutex_t *mutex)** : verrouille le mutex.
  - Si le **mutex** est déverrouillé, il devient verrouillé et est possédé par le thread appelant et **pthread_mutex_lock()** rend la main immédiatement.
  - Si le **mutex** est déjà verrouillé par un autre thread, **pthread_mutex_lock()** suspend le thread appelant jusqu'à ce que le mutex soit déverrouillé.

- Déverrouillage :
- **int pthread_mutex_unlock(pthread_mutex_t *mutex)** : déverrouille le mutex.

- Remarque : Seul le thread qui a effectué l'opération de verrouillage, peut effectuer l'opération de déverrouillage.

Portée du mutex

- Portée locale :
- portée limitée au processus (défaut)

- Portée globale :
- permet de synchroniser des threads appartenant à plusieurs processus. Requiert que le mutex soit alloué dans une zone de mémoire partagée.

Un exemple d'utilisation mutex

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

static pthread_mutex_t my_mutex;
static int tab[5];

int main (int argc, char **argv){
  pthread_t th1, th2;
  void *ret;
  pthread_mutex_init(&my_mutex, NULL);//initialise le mutex par défaut
  if (pthread_create(&th1, NULL, write_tab_process, NULL) < 0) {
                fprintf (stderr, "pthread_create error for thread 1\n");
                exit (1);
  }
  if (pthread_create(&th2, NULL, read_tab_process, NULL) < 0) {
                fprintf (stderr, "pthread_create error for thread 2\n");
                exit (1);
  }
  (void)pthread_join (th1, &ret);
  (void)pthread_join (th2, &ret);
}
...
```

Un exemple d'utilisation mutex (suite)

```
...
void *read_tab_process(void *arg)
{
  int i;
  pthread_mutex_lock(&my_mutex);
  for (i=0; i!=5; i++)
                        printf("read_process, tab[%d] vaut  %d\n", i, tab[i]);
  pthread_mutex_unlock(&my_mutex);
            pthread_exit (0);
}

void *write_tab_process(void *arg)
{
            int i;
  pthread_mutex_lock(&my_mutex);
  for (i=0; i!=5; i++){
                        tab[i] = 2*i;
                        printf("write_process, tab[%d] vaut  %d\n", i, tab[i]);
                        sleep(1);//ralentit le thread d'écriture
  }
  pthread_mutex_unlock(&my_mutex);
            pthread_exit (0);
}
```

```
gcc -o thread2 thread2.c -lpthread
./thread2

write_process, tab[0] vaut 0
write_process, tab[1] vaut 2
write_process, tab[2] vaut 4
write_process, tab[3] vaut 6
write_process, tab[4] vaut 8

read_process, tab[0] vaut 0
read_process, tab[1] vaut 2
read_process, tab[2] vaut 4
read_process, tab[3] vaut 6
read_process, tab[4] vaut 8
```

Programmation Système et Réseau

6 – Synchronisation entre processus : Sémaphores POSIX

Mohamed Maachaoui
Juan Ángel Lorenzo del Castillo
Seytkamal Medetov
Son Vu

CY Tech
ING2 – GSI
2020 – 2021

Problème de l'exclusion mutuelle

- Exemple : deux banques modifient un compte en même temps

Agence Nancy

```
1. courant = get_account(1867A)
2. nouveau = courant + 10
3. update_account (1867A, nouveau)
```

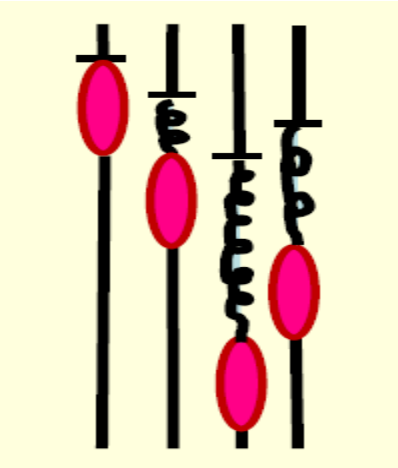
Agence Karlsruhe

```
1. aktuelles = get_account(1867A)
2. neue = aktuelles - 10
3. update_account(1867A, neue)
```

- variables partagées + exécutions parallèles entremêlées

  ==> différents résultats

- C'est une **condition de compétition** (*race condition*)



- **Solution** : opérations **atomiques** ; pas d'exécutions entremêlées
- Cette opération est une **section critique** à exécuter en **exclusion mutuelle**

Réalisation d'une section critique

- Schéma général

Processus 1

```
...
entrée en section critique
    section critique
sortie de section critique
...
```

Processus 2

```
...
entrée en section critique
    section critique
sortie de section critique
...
```

- Exclusion mutuelle garantie par les opérations

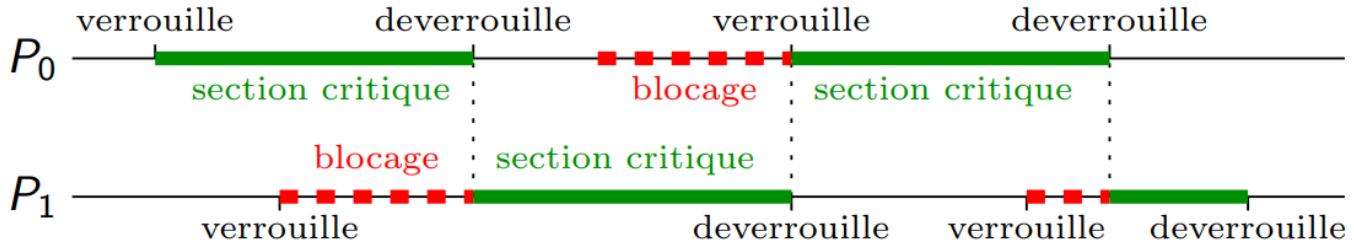
`(entrée en section critique) et (sortie de section critique)`

Exclusion mutuelle par verrouillage

- Verrouiller une ressource garanti un accès exclusif à la ressource
- **Appels systèmes** de verrouillage/déverrouillage (*noms symboliques*) :
  - verrouille(f) : verrouille f avec accès exclusif
  - déverrouille(f) : déverrouille f
- **Propriétés :**
  - Opérations atomiques (assuré par l'OS)
  - Verrouillage par au plus un processus
  - Tentative de verrouillage d'une ressource verrouillée ➔ blocage du processus
  - Déverrouillage ➔ réveille d'un processus en attente du verrou (et un seul)

```
...
verrouille(fich);
accès à fich (section critique);
deverrouille(fich);
...
```

```
...
verrouille(fich);
accès à fich (section critique);
deverrouille(fich);
...
```



Notion d'interblocage

- **Utilisation simultanée de plusieurs verrous ➔ problème potentiel**
- Situation : deux processus verrouillent deux fichiers

Déroulement

- Première possibilité : 1a ; 1b ; 2a ; 2b
- Seconde possibilité : 2a ; 2b ; 1a ; 1b
- Troisième possibilité : 1a ; 2a ; 1b ; 2b

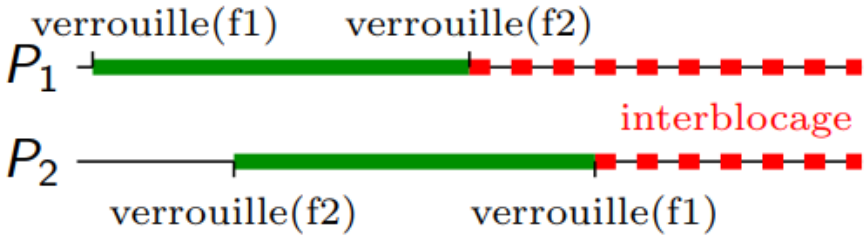
Processus 1

```
...
verrouille (f1) /* 1A */
accès à f1
...
verrouille (f2) /* 1B */
accès à f1 et f2
deverrouille (f2)
deverrouille (f1)
```

Processus 2

```
...
verrouille (f2) /* 2A */
accès à f2
...
verrouille (f1) /* 2B */
accès à f1 et f2
deverrouille (f2)
deverrouille (f1)
```

Exécution de 1a ;2a ;1b ;2b



P1 et P2 sont bloqués :

- P1 attend le deverrouille(f2) de P2
- P2 attend le deverrouille(f1) de P1
- C'est un **interblocage** (deadlock)

Situation d'interblocage

- **Définition**
  - Plusieurs processus bloqués dans l'attente d'une action de l'un des autres
  - Impossible de sortir d'un interblocage sans intervention extérieure
- **Conditions d'apparitions**
  - Plusieurs processus en compétition pour les mêmes ressources
  - Cycle dans la chaîne des attentes

Situation réelle d'interblocage

Comment prévenir l'interblocage ?

- **Solution 1 : réservation globale**
  - Demandes en bloc de toutes les ressources nécessaires
  - Inconvénient : réduit les possibilités de parallélisme
  - Analogie du carrefour : damier jaune
- **Solution 2 : requêtes ordonnées**
  - Tous les processus demandent les ressources dans ***le même ordre***
  - Interblocage alors impossible
  - Analogie du carrefour : construire un rond-point

```
verrouille (f1)
verrouille (f2)
accès à f1 et f2
deverrouille (f2)
deverrouille (f1)
```

```
verrouille (f1)
verrouille (f2)
accès à f1 et f2
deverrouille (f2)
deverrouille (f1)
```

- **Solution 3 : modification de l'algorithme**
  - Modifier code utilisateur pour rendre impossible l'interblocage
  - Analogie du carrefour : construire un pont au dessus du carrefour

Problèmes de synchronisation (résumé)

- **Condition de compétition (race condition)**
- **Définition** : le résultat change avec l'ordre des instructions
- Difficile à corriger car difficile à reproduire (ordre aléatoire)
- Egalement type de problème de sécurité :
  - Un programme crée un fichier temporaire, le remplit puis utilise le contenu
  - L'attaquant crée le fichier avant le programme pour contrôler le contenu
- **Interblocage (deadlock)**
- **Définition** : un groupe de processus bloqués en attente mutuelle
- Evitement parfois difficile (correction de l'algorithme)
- Détection assez simple, mais pas de guérison sans perte
- **Famine (starvation)**
- **Définition** : un processus attend indéfiniment une ressource (problème d'équité)
- Servir équitablement les processus demandeurs

Sémaphore

- **Généralisation de l'exclusion mutuelle**
- Objet composé :
  - D'une variable : valeur du sémaphore (nombre de places restantes)
  - D'une file d'attente : liste des processus bloqués sur le sémaphore
- Primitives associées :
  - Initialisation (avec une valeur positive ou nulle)
  - Prise (P, *Probeer*, down, wait) = demande d'autorisation (« puis-je ? »)
  - Si valeur = 0, blocage du processus ; Si non, valeur = valeur − 1
  - Validation (V, *Verhoog*, up, signal) = fin d'utilisation (« vas-y »)
    - Si valeur = 0 et processus bloqué, déblocage d'un processus ;
    - Si non, valeur = valeur + 1

Schémas de synchronisation

Situations usuelles se retrouvant lors de coopérations inter-processus

- **Exclusion mutuelle** : ressource accessible par une seule entitée à la fois
- **Problème de cohorte** : ressource partagée par au plus N utilisateurs
- **Rendez-vous** : des processus collaborant doivent s'attendre mutuellement
- **Producteurs/Consommateurs** : un processus doit attendre la fin d'un autre
- **Lecteurs/Rédacteurs** : notion d'accès exclusif entre catégories d'utilisateurs

Comment résoudre ces problèmes avec les sémaphores ?

Exclusion mutuelle par sémaphore

Très simple

Initialisation

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

Agence Nancy

```
P(sem)
courant = get_account(1867A)
nouveau = courant + 1000
update_account (1867A, nouveau)
V(sem)
```

Agence Karlsruhe

```
P(sem)
aktuelles = get_account(1867A)
neue = aktuelles - 1000
update_account(1867A, neue)
V(sem)
```

Cohortes et sémaphores

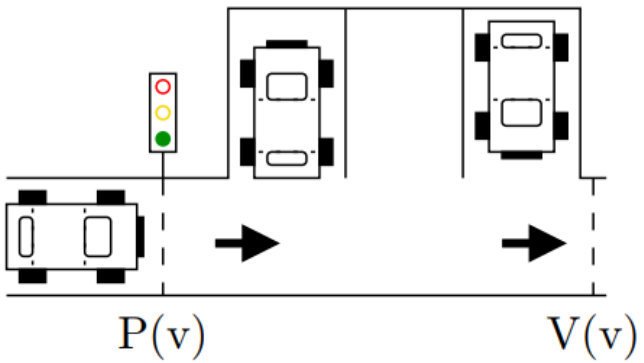
Sémaphore inventée pour cela

Initialisation

```
sem=semaphore(3)    /* nombre de places */
```

Garer sa voiture

```
P(sem)
poser sa voiture au parking
aller faire les courses
Reprendre la voiture
V(sem)
partir
```



Rendez-vous et sémaphores

- **Envoi de signal**
  - Un processus indique quelque chose à un autre (disponibilité donnée)

Initialisation
```
top=semaphore(0)
```

Processus 1
```
...
calcul(info)
V(top)
...
```

Processus 2
```
...
P(top)   /*Bloque en attente*/
utilisation(info)
...
```

  - top = **sémaphore privée** (initialisée à 0)
- utilisée pour synchro avec quelqu'un, pas sur une ressource
- **Rendez-vous entre deux processus**
  - Les processus s'attendent mutuellement

Initialisation
```
roméo=semaphore(0)
juliette=semaphore(0)
```

Processus romeo
```
P(romeo)       /*se bloque*/
V(juliette)    /*libère J*/
```

Processus juliette
```
V(romeo)       /*libère R*/
P(juliette)     /*bloque*/
```

- **Rendez-vous entre trois processus et plus**
  - On parle de barrière de synchronisation
  - La solution précédente est généralisable, mais un peu lourde
  - Souvent une primitive du système

Rendez-vous et sémaphores

- On parle ici de l'interface POSIX, pas de celle IPC Système V
- `#include <semaphore.h>`
- Type : **sem_t**

Interface de gestion

- int **sem_init**(sem_t *sem, int pshared, unsigned int valeur)
  pshared !=0 ⇒ sémaphore partagée entre plusieurs processus (pas sous linux)
- int **sem_destroy**(sem_t * sem) (personne ne doit être bloqué dessus)

Interface d'usage

- int **sem_wait**(sem_t * sem) réalise un P()
- int **sem_post**(sem_t * sem) réalise un V()
- int **sem_trywait**(sem_t * sem) P() ou renvoie EAGAIN pour éviter blocage
- int **sem_getvalue**(sem_t * sem, int * sval)

Sémaphores : exemple

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

sem_t semaphore;

static void * routine_thread (void * numero_thread);
static int aleatoire (int maximum);

int main (void)
{
    int i;
    pthread_t thread;

    sem_init(& semaphore, 0, 3);
    for (i = 0; i < 10; i ++)
        pthread_create(& thread, NULL, routine_thread, (void *) i);
    pthread_exit(NULL);
}
```

Sémaphores : exemple

```c
void * routine_thread (void * numero_thread)
{
    int i;
    for (i = 0; i < 2; i ++)
    {
        sem_wait(& semaphore);
        fprintf(stdout, "Thread %d dans portion critique \n",
                                    (int) numero_thread);
        sleep(aleatoire(4));
        fprintf(stdout, "Thread %d sort de la portion critique \n",
                                    (int) numero_thread);
        sem_post(& semaphore);
        sleep(aleatoire(4));
    }
    return NULL;
}
```

Références

- Les liens sur internet :
  - Initiation à la programmation multitâche en C avec Pthreads : https://franckh.developpez.com/tutoriels/posix/pthreads/
  - Utiliser Thread et MUTEX (API pthread) : https://ressourcesinformatiques.com/article.php?article=823
  - Multi-threading sous LINUX : http://pficheux.free.fr/articles/lmf/threads/
  - pthreads in C – a minimal working example
  - Manuel du programmeur Linux :
    - http://manpagesfr.free.fr/man/man3/pthread_mutex_init.3.html
    - http://man7.org/linux/man-pages/man3/pthread_join.3.html
    - https://man.developpez.com/man3/pthread_create/
  - Cours Réseaux et Systèmes de Lucas Nussbaum: https://members.loria.fr/lnussbaum/RS/CM-ch5.pdf