Processus - Signaux - Synchronisations

https://sleek-think.ovh/enseignement

Jehan-Antoine Vayssade

Programme

- Niveau 0 : Processus et signaux
- Niveau 1 : Communication et mémoire partager
- Niveau 2 : Les threads, mutex et sémaphores
- Niveau 3 : Les instructions SIMD (skip)
- Niveau 4 : Les instructions atomique et structures lock-free (skip)
- Niveau 5 : OpenMP, OpenCL et Cuda (skip)

Niveau 0

Processus et signaux

Processus

Programme

- C’est un code compilé contenant des instructions.
- Pour une architecture cible (arm, 64bit, 32bit, fpga, …)
- Pour un système cible (Linux, BSD, Solaris, … windows, android, ipomme)
- EXE pour windows, ELF pour les système UNIX, DOS MZ, …

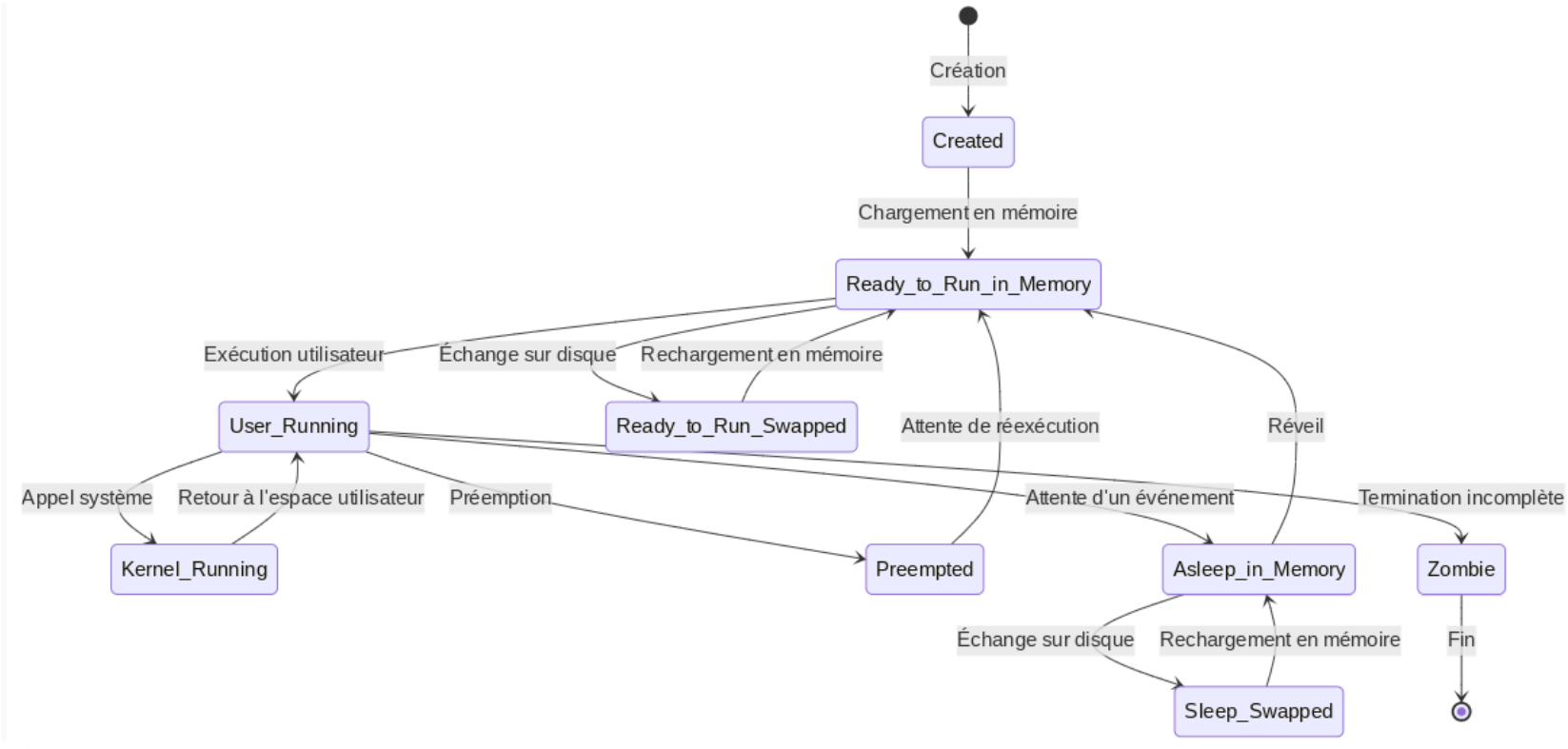
Processus

- Un processus est l’instance d’un programme en cours d’exécution par une unité de calcule.
- Il utilise des ressources système comme la mémoire et le processeur.
- Il a un identifiant `<pid>` pour `process id`.
- Il est dans un état.

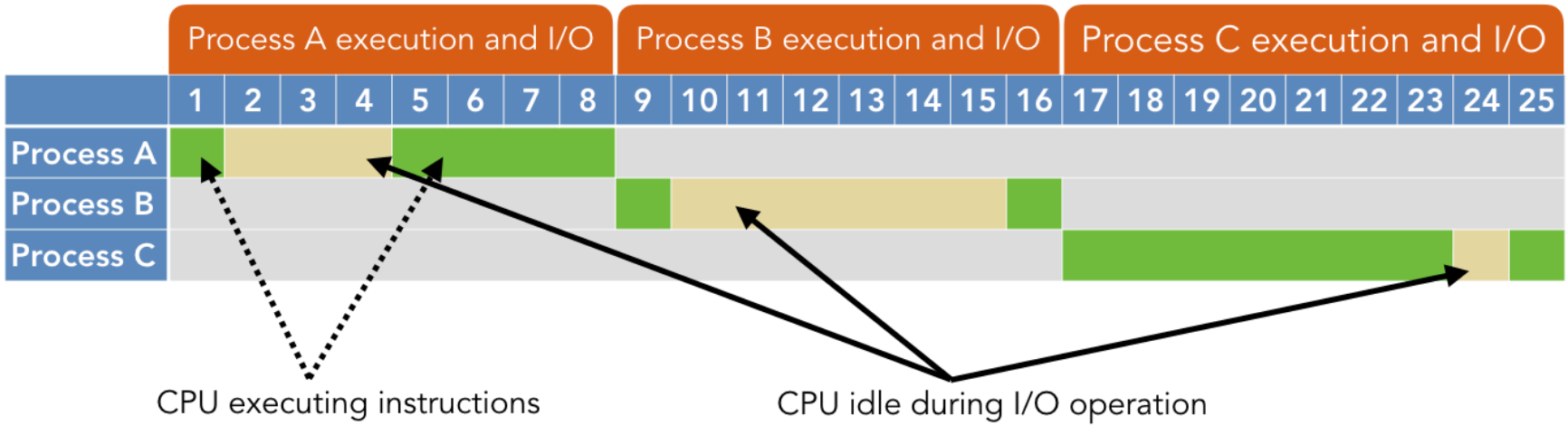
État d’un processus (version simple)

- Création : Un processus est créé via la fonction fork() ou exec() sur les systèmes Unix.
- Exécution : Le processus s’exécute en utilisant des ressources système.
- Suspension : Un processus peut être suspendu par un signal ou par le système (Ctrl+S).
- Termination : Un processus se termine normalement ou par un signal (Ctrl+C).
- Zombie : Le processus a terminé mais son père n’a pas encore récupéré son statut.

Cycle de vie d'un processus



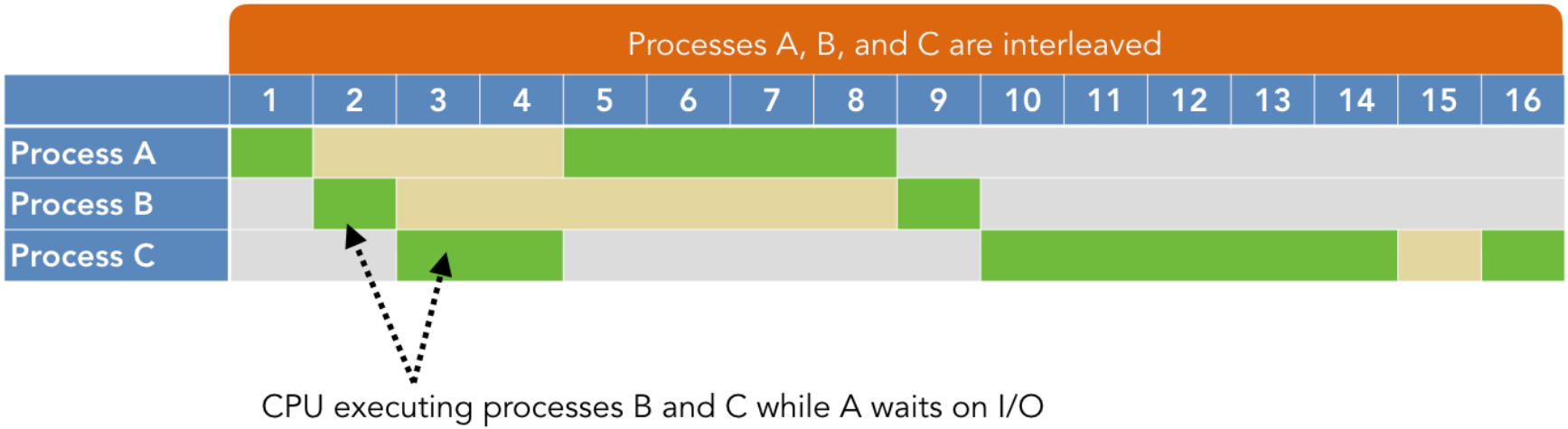
Systeme uniprogramming



Chaque processus a un temps d'éxécution géré par le kernel.
Lorsqu'un processus accede a un I/O (ex lecture fichier) il attend simplement.
On perd beaucoup de temps a attendre.

> Pour passé d'un programme a un autre, on parle de commutation de context

Systeme multiprogramming



Chaque processus a un temps d'éxécution géré par le kernel.
En cas d'attente I/O (ex lecture fichier) on donne la main a un autre programme.

> Attention, le scheduler (ordonanceur system) peu préempter une tache a n'import quel moment. Pour changer de process.

Fork et wait

La fonction `fork()` crée un nouveau processus (appelé processus enfant) en dupliquant le processus actuel (appelé processus parent). La copie contient tout les descripteurs déjà ouvert (fichier, etc).

- Dans le processus parent, fork() retourne l'ID du processus enfant.
- Dans le processus enfant, fork() retourne 0.
- En cas d'erreur, fork() retourne -1.
- Attention, fork() fait une copie de la stack et de la heap !

La fonction `wait(int*status)` fait suspendre le processus parent jusqu'à ce qu'un de ses enfants se termine et enregistre le code de retour dans le pointeur status. L'ID du processus enfant qui s'est terminé.

- La fonction `waitpid(pid_t, int*status)` permet d'attendre en processus spécifique.
- La fonction `pid_t getpid(void);` Donne le pid du processus actuel
- La fonction `pid_t getppid(void);` Donne le pid du processus parent

Exemple

```c
int main() {
  pid_t pid;
  int status;
  pid = fork();

  if (pid == 0) { // Processus enfant (connait le pid parent via getppid())
    printf("Je suis l'enfant avec PID %d\n", getpid());
    sleep(2); // Attendre 2 secondes
    printf("Enfant terminé\n");
    exit(12); // Terminer avec un statut de sortie
  }
  else if (pid > 0) { // Processus parent (connait le pid de l'enfant)
    printf("Je suis le parent avec PID %d, enfant %d\n", getpid(), pid);
    pid = wait(&status); // Attendre la fin de l'enfant (status => 12)
    printf("Enfant terminé avec statut %d\n", status);
  }
  else {
    perror("fork"); // Erreur lors de fork
    exit(1);
  }

  printf("Fin du processus\n");
  return 0;
}
```

Commande kill et signaux

La commande `kill` est utilisée pour envoyer des signaux à un processus.

Les signaux courants :

- SIGKILL (9) : Force la fin immédiate d'un processus.
- SIGTERM (15) : Demande poliment à un processus de se terminer.
- SIGSTOP (19) : Suspend un processus.
- SIGCONT (18) : Réactive un processus suspendu.

Exemple:

```
kill -9 <pid>
```

Système et signaux

Ils peuvent aussi être générés par le kernel en réponse à des événements système.

- SIGSEGV : Signal généré lors d'un accès invalide à la mémoire (segfault).
  - souvent une erreur de programmation, double free, etc
- SIGFPE : Signale une erreur arithmétique fatale.
  - probablement une div par zero
- SIGILL : Tente d'exécuter une instruction illégale ou malformée.
  - éxécutable corrompu ? hack ?
- SIGBUS : Tentative d'accès à une adresse non alignée ou à une zone mémoire non valide.

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void handler(int sig) {
    printf("Signal %d reçu.\n", sig);
}

int main() {
    // Installation du gestionnaire de signal
    signal(SIGILL, handler); // Pour SIGILL
    signal(SIGBUS, handler); // Pour SIGBUS
    signal(SIGTRAP, handler); // Pour SIGTRAP
    signal(SIGABRT, handler); // Pour SIGABRT

    while(1) {
        printf("Processus en cours...\n");
        sleep(1);
    }

    return 0;
}
```

Niveau 1

Mémoire partagées

Mémoire partagées

> Lorsque fork est appelé, le processus enfant reçoit une copie des pages mémoire du processus parent.
> Cependant, ces pages sont marquées comme étant en lecture seule et sont partagées entre les deux processus.

> Sun processus enfant tente de modifier une page partagée, le système génère une exception de page (page fault).
> En réponse, le système crée une copie privée de la page pour le processus qui a tenté la modification.

> Chaque processus va écrire dans ça propre version / mémoire privé !

Il exist plusieurs façon de partagé des données.

Communication entre processus

- Pipes : Canaux unidirectionnels pour l’échange de données.
- Named Pipes (FIFO) : Canaux nommés pour l’échange de données.
- Shared Memory : Mémoire partagée entre processus.
- Message Queues : Files d’attente de messages pour l’échange de données.
  - Peu usité
- Sockets : Communication réseau entre processus.

Utilisation des pipes

Les pipes sont des canaux unidirectionnels pour l'échange de données entre processus.

```
int pipefd[2];
pipe(pipefd);
pipefd[0] // descripteur de fichier pour la lecture (sortie du tube / read).
pipefd[1] // descripteur de fichier pour l'écriture (entrée du tube / write).
```

En cas de succès, pipe renvoie 0.

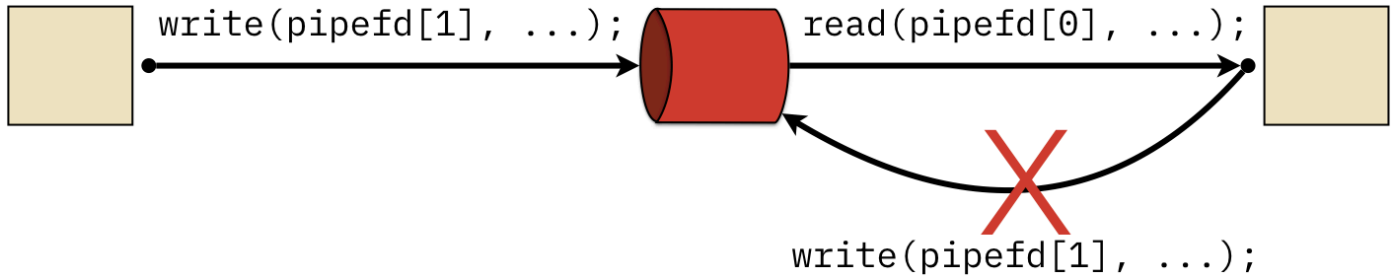
```
int pipefd[2];
if (pipe(pipefd) == -1) {
    perror("pipe");
    exit(EXIT_FAILURE);
}
```

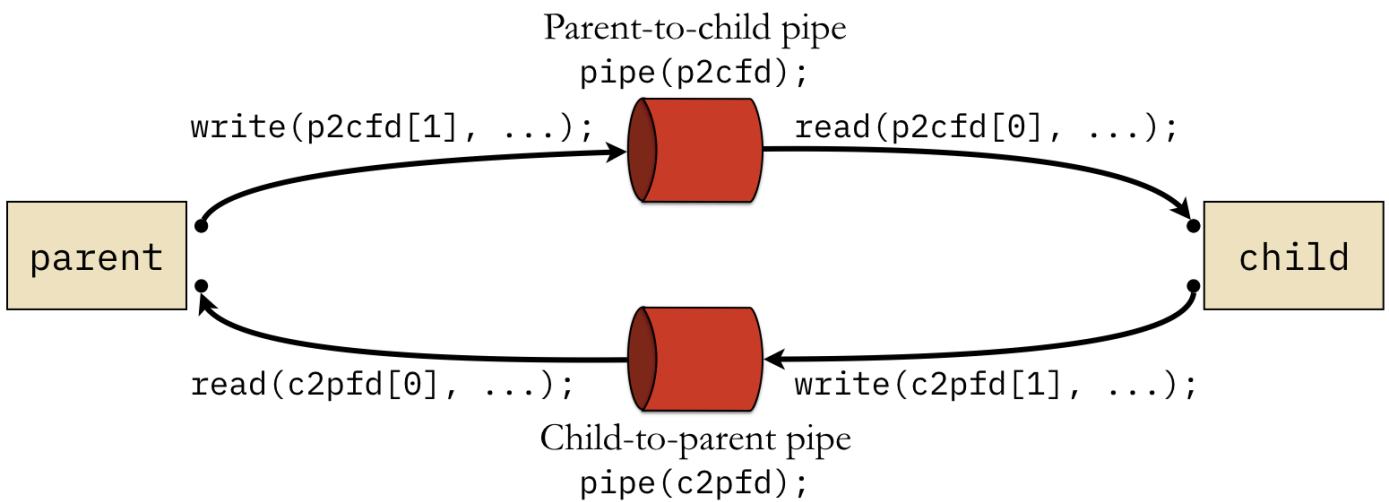
Lorsque vous créez un processus fils avec fork, celui-ci hérite des descripteurs de fichiers ouverts par le processus père, y compris ceux du pipe. Pour éviter les problèmes de synchronisation, chaque processus doit fermer l'extrémité du pipe qu'il n'utilise pas.

Particularité des pipes

- **Tamponnage :** Les données écrites dans un pipe sont stockées en mémoire tampon par le noyau jusqu'à ce qu'elles soient lues. Cela signifie que si le lecteur ne lit pas les données immédiatement, elles restent dans le tampon jusqu'à ce qu'elles soient traitées.
- **Synchronisation :** Il est important de synchroniser les lectures et écritures sur un pipe pour éviter les problèmes de concurrence. Cela peut être fait en fermant les extrémités inutilisées et en utilisant des mécanismes de synchronisation comme les sémaphores ou les verrous.
- **Limitations de taille :** Les pipes ont une limite de taille pour les données tamponnées. Si cette limite est atteinte, les écritures bloquent jusqu'à ce que des données soient lues.
- **Blocage en cas de pipe vide :** La fonction read met le processus en attente jusqu'à ce que des données soient disponibles ou que l'extrémité d'écriture soit fermée.
- `ls -l | sort -n -k 5 | tail -n 1 | awk '{print $NF}'`
- `./main < test.txt`

Les pipes sont unidirectionel !





Exemple

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    int pipefd[2];
    pipe(pipefd); // 1 process en écriture, 1 process en lecture

    pid_t pid = fork();
    if (pid == 0) { // Processus enfant
        close(pipefd[1]); // Fermer l'extrémité d'écriture
        char buffer[100];
        read(pipefd[0], buffer, 100);
        printf("Enfant : %s\n", buffer);
        close(pipefd[0]);
    }
    else { // Processus parent
        close(pipefd[0]); // Fermer l'extrémité de lecture
        char *msg = "Hello, Child!";
        write(pipefd[1], msg, strlen(msg) + 1);
        close(pipefd[1]);
    }

    return 0;
}
```

Utilisation des sockets

Les sockets permettent la communication entre processus en passant par les couches du réseau. Il est nécessaire de suivre les différentes étapes de connexion réseau du modèle OSI. Par conséquent, il faut avoir un processus serveur qui accepte les connexions, tandis que le processus fils se connecte à ce serveur.

```
int server_fd = socket(AF_INET, SOCK_STREAM, 0); // création de la socket serveur
bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr); // Liaison du socket à une interface reseau
// communication
recvfrom(server_fd, buffer, BUFFER_SIZE, 0, (struct sockaddr *)&sender_addr, &sender_len);
sendto(server_fd, msg, strlen(msg) + 1, 0, (struct sockaddr *)&sender_addr, sender_len);
// idem sur le client (processus fils)
```

> Cette solution est parfois utilisée, notamment pour permettre une communication pour des processus externes a votre solution. Dans les autres cas, elle est peu recommandable. Des processus comme DBUS, X11 et MySQL utilisent cette méthode.

Utilisation de mmap

mmap permet de mapper un fichier ou une zone mémoire anonyme dans l'espace d'adressage d'un processus. Avec le flag `MAP_SHARED`, plusieurs processus peuvent accéder à la même zone mémoire.

```
void *ptr = mmap(NULL, taille, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
```

Les mémoires partagées peuvent être nommé, c'est à dire associé a un descripteur de fichier :

```
int fd = shm_open("/example", O_RDWR | O_CREAT, 0666);
void *ptr = mmap(NULL, 1024, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

> Attention, mmap permet a plusieurs processus d'écrire simultanément dans la mémoire. Mais `mmap`, ne gère pas ces accès concurrents (race conditions). Dans le cas ou deux processus écrive simultanément dans la mémoire, aucune garantie sur l'ordre d'éxécution n'est possible ! Dans le meilleurs des cas une des deux données est stocké, dans le pire des cas une supperposition des données est possible, donnant un résultat aléatoire.

Exemple mmap

```c
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    int fd = shm_open("/example", O_RDWR | O_CREAT, 0666);
    // verification erreur d'ouverture @fd
    ftruncate(fd, 1024); // Taille de la mémoire partagée

    void *ptr = mmap(NULL, 1024, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    // verification erreur d'ouverture de @ptr

    // Écriture dans la mémoire partagée
    char *msg = "Hello, World!";
    memcpy(ptr, msg, strlen(msg) + 1);
    // Lecture dans la mémoire partagée
    printf("Données lues : %s\n", (char *)ptr);

    munmap(ptr, 1024);
    close(fd);
    shm_unlink("/example");

    return 0;
}
```

Niveau 2

Thread, mutex et sémaphore

Thread, mutex et sémaphore

Limitation de `fork(), exec(), wait() ...`

- Création de Processus : fork crée un nouveau processus en dupliquant le processus parent.
  Cela est coûteux en termes de ressources (cpu et mémoire).
- Mémoire Non Partagée : Les processus créés par fork n'ont pas de mémoire partagée par défaut.
  Nécessite des mécanismes IPC supplémentaires.
- Zombies : Si un parent ne wait pas correctement, les processus enfants peuvent devenir des zombies.
  Occupation des entrées dans la table des processus.

Les threads POSIX (Pthreads) sont une bibliothèque pour écrire des programmes multi-threads.
Ces derniers viennent répondre a ces limitations.

- Pthreads permet de créer des threads dans un processus, partageant la même mémoire et ressources.
- Moins coûteux que fork, communication interne plus facile via la même mémoire (partagée).
- Chaque thread a sa propre stack, le reste est partagé (même le pid)

Utilisation

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

void* threadFunction(void* arg) {
    printf("Hello from thread!\n");
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, threadFunction, NULL);
    pthread_join(thread, NULL); // Attendre la fin du thread
    return 0;
}
```

Passage de variables

```
struct args {
  int a, b;
};

struct results {
  int sum, difference;
};

void * calculator (void *_args) {
  struct args *args = (struct args *) _args;

  struct results *results = malloc (sizeof (struct results));
  results->sum = args->a + args->b;
  results->difference = args->a - args->b;

  free (args);
  pthread_exit (results);
}

pthread_t child;
struct args *args = (struct args*)malloc(sizeof (struct args));
pthread_create (&child, NULL, calculator, &args)
struct results *results;
pthread_join (child[i], (void **)&results);
```

Quel est l'ordre d'éxécution de ce code ?

```c
int number = 3;

void * thread_A (void* args) {
  int x = 5;
  printf ("A: %d\n", x + number);
}

void * thread_B (void* args) {
  int y = 2;
  printf ("B: %d\n", y + number);
}

int main (int argc, char** argv) {
  /* Create thread A */
  /* Create thread B */
  /* Wait for threads to finish */
  return 0;
}
```

- Indice Load & Store

Quel est l'ordre d'éxécution de ce code ?

```c
int number = 3;

void * thread_A (void* args) {
  number += 5;
  printf ("A: %d\n", number);
}

void * thread_B (void* args) {
  number -= 2;
  printf ("B: %d\n", number);
}

int main (int argc, char** argv) {
  /* Create thread A */
  /* Create thread B */
  /* Wait for threads to finish */
  return 0;
}
```

- Indice Load & Store

Système des jetons ferroviaires : Un jeton physique garantit qu'un seul train entre dans une section à voie unique. En informatique nous avons un jeton numérique qui garantie l'access par un seul processus/thread à la section.

Section critique

Une `section critique` est une partie d'un programme où des ressources partagées sont accédées sans interférence pour éviter les problèmes de synchronisation (`race condition`) et donc les incohérences de comportement ou de données.

- **Autorisation :** Un processus a besoin d'un "jeton" pour accéder à une section critique, comme un train pour une voie unique.
- **Exclusion Mutuelle :** Un seul processus peut accéder à la ressource partagée, comme un seul train sur la voie.
- **Synchronisation :** Des mécanismes comme les sémaphores garantissent que les processus entrent un par un.
- **Prévention des Collisions :** Les ressources sont accédées séquentiellement pour éviter les problèmes de données.

Géré des exlusions mutuel (mutex)

Les mutex (mutual exclusion locks) sont des primitives de synchronisation utilisées pour protéger les sections critiques du code, où plusieurs threads ne doivent pas accéder simultanément aux mêmes données.

- **Initialisation :** Un mutex doit être initialisé avant utilisation

  par exemple avec `PTHREAD_MUTEX_INITIALIZER`.
- **Verrouillage (bloquant) :** Un thread peut verrouiller un mutex avec `pthread_mutex_lock()`.
- **Verrouillage (non-bloquant) :** Essaye de verrouiller un mutex avec `pthread_mutex_trylock()`.
- **Déverrouillage :** Un mutex est déverrouillé avec `pthread_mutex_unlock()` après la section critique.

Exemple

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

int shared_data = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* threadFunction(void* arg) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&mutex);
        shared_data++;
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, threadFunction, NULL);
    pthread_create(&thread2, NULL, threadFunction, NULL);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    printf("Final shared data: %d\n", shared_data);
    return 0;
}
```

Qu’est-ce qui ne va pas avec ce code ?

```c
pthread_mutex_t mutex1;
pthread_mutex_t mutex2;

void *thread1_function(void *arg) {
    pthread_mutex_lock(&mutex1);
    sleep(1); // Simulate some work
    pthread_mutex_lock(&mutex2);

    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);
    return NULL;
}

void *thread2_function(void *arg) {
    pthread_mutex_lock(&mutex2);
    sleep(1); // Simulate some work;
    pthread_mutex_lock(&mutex1);

    pthread_mutex_unlock(&mutex1);
    pthread_mutex_unlock(&mutex2);
    return NULL;
}
```

Introduction aux Sémaphores

Les sémaphores sont des primitives de synchronisation utilisées pour contrôler l'accès à des ressources partagées entre plusieurs threads ou processus. Ils sont représentés par un entier qui peut être incrémenté ou décrémenté atomiquement. Par exemple on peu compter le nombre de case vide dans un buffer.

```c
#include <semaphore.h>

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

Exemple producteur - consomateur

```c
int buffer[BUFFER_SIZE]; // Déclaration du tampon
int in = 0, out = 0; // Index pour le producteur et le consommateur
sem_t empty; // Nombre de places vides dans le tampon
sem_t full; // Nombre d'éléments dans le tampon
sem_t mutex;  // Mutex pour l'accès exclusif au tampon

sem_init(&empty, 0, BUFFER_SIZE); // Tampon vide au début
sem_init(&full, 0, 0); // Aucun élément dans le tampon
sem_init(&mutex, 0, 1); // Mutex pour accès exclusif

void* producer(void* arg) {
    sem_wait(&empty); // Attendre qu'il y ait de la place dans le tampon
    sem_wait(&mutex); // Section critique

    int item = rand() % 100;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;

    sem_post(&mutex);
    sem_post(&full); // Signaler qu'un item est disponible
    return NULL;
}
```

Multiple producteurs - multiple consommateurs

- A voir en TP (MPMC)