DIFFÉRENCE ENTRE PROCESSUS ET THREADS SOUS LINUX

Définition thread

- The Linux Programming Interface :

  Comme les processus, les threads sont un mécanisme qui permet à une application d'effectuer plusieurs tâches simultanément. Un seul processus peut contenir plusieurs threads. Tous ces threads exécutent indépendamment le même programme et partagent la même mémoire globale, y compris les segments de données initialisées, non initialisées et du tas.

Processus

- Instructions après instruction
  => séquentiel

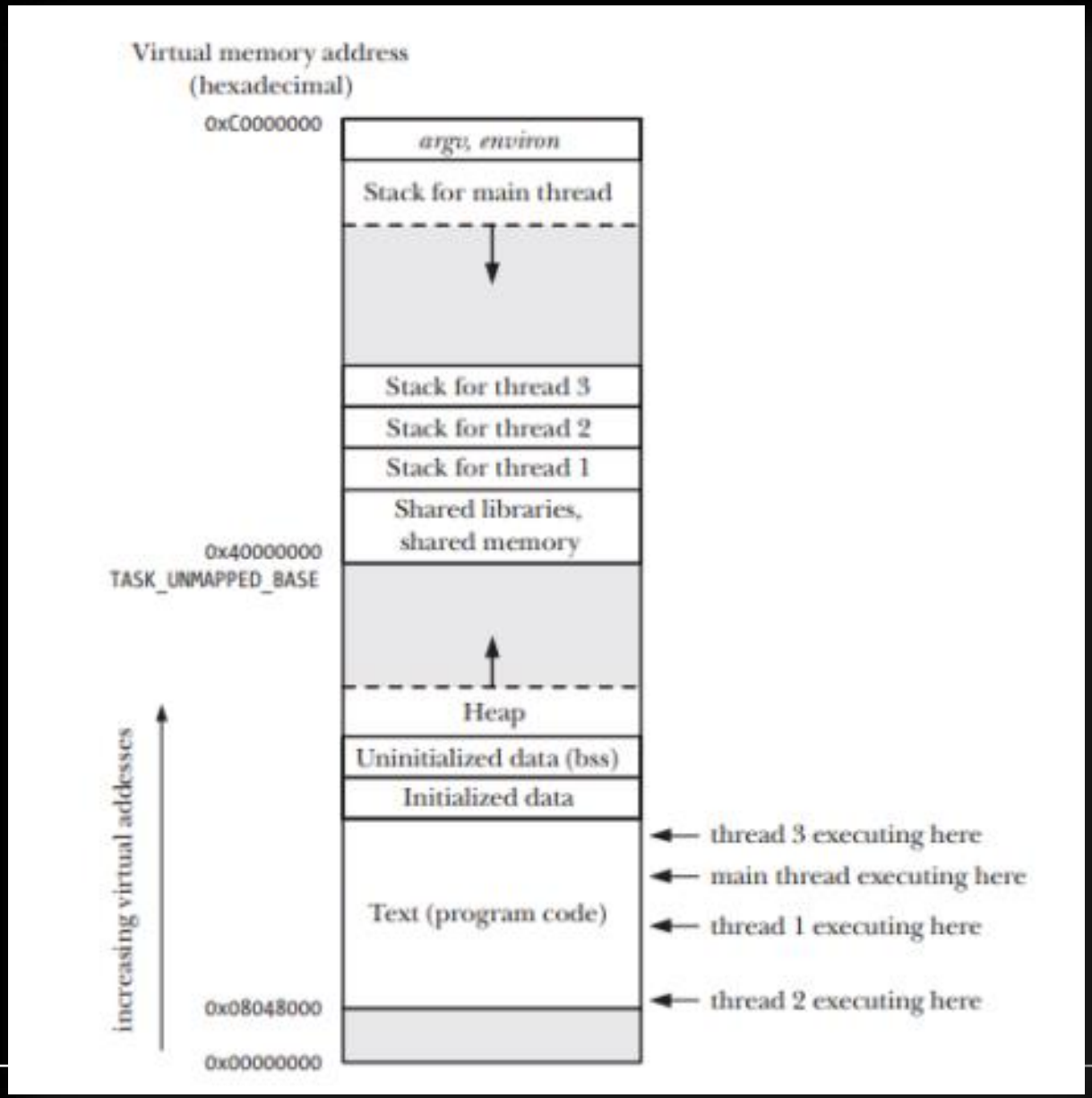
```c
#include <unistd.h>
#include <stdio.h>

void myturn(){
    for(int i = 0; i<8; i++){
        sleep(1);
        printf("my Turn\n");
    }
}
void yourturn(){
    for(int i = 0; i<8; i++){
        sleep(1);
        printf("Your Turn\n");
    }
}

int main(){
    myturn();
    yourturn();
}
```

Comment exécuter plusieurs choses en même temps? (1/2)

- *Fork* : copier le processus appelant pour donner naissance à un fils.
  - Coût en performances élevé => copie quasi-totalité de la mémoire
  - Partage de données inter-processus plus complexe

Comment exécuter plusieurs choses en même temps? (2/2)

- *Thread* :
  - Création moins coûteuse en performances
  - Copie pas toute la mémoire
  - 10 fois plus rapide qu'un *fork*
  - Il faut gérer la concurrence d'accès aux données


Virtual memory address
(hexadecimal)
0xC0000000
argv, environ
Stack for main thread
Stack for thread 3
Stack for thread 2
Stack for thread 1
Shared libraries,
shared memory
0x40000000
TASK_UNMAPPED_BASE
Heap
Uninitialized data (bss)
Initialized data
Text (program code)
0x08048000
0x00000000
increasing virtual addresses
thread 3 executing here
main thread executing here
thread 1 executing here
thread 2 executing here

Composition d’un thread

- Id thread
- Contexte d’exécution
- Pile
- Registres
- ○○○

Partagé entre threads :

- Espace d’adressage mémoire
- Code du programme
- Le tas
- Variables d’environnements
- Descripteurs de fichier
- Gestionnaire de signaux

Non partagé entre threads :

- Pile
- Gestionnaire d’exception
- Id

Comment utiliser un thread?

- Un processus possède de base un thread => thread principal (heavyweight process)
- Rajout de threads supplémentaire => multi-threading

Création de thread

- Renvoie 0 si succès, un entier positif en cas d'erreur
- pthread_t : un pointeur vers un buer de type pthread_t
- pthread_attr_t : spécie les diérents attributs du thread créé
- *(* start) (void *) : pointeur de fonction
- * arg : pointeur vers l'argument de la fonction

Création d'un thread

```
1 #include <pthread.h>
2
3 int pthread_create(pthread_t *thread, const
    pthread_attr_t *attr,
4  void *(*start)(void *), void *arg);
```

```c
void* myturn(){
    for(int i = 0; i < 8; i++){
        sleep(1);
        printf("my Turn\n");
    }
}
void yourturn(){
    for(int i = 0; i < 4; i++){
        sleep(1);
        printf("Your Turn\n");
    }
}

int main(){
    pthread_t newthread;

    pthread_create(&newthread, NULL, &myturn, NULL);

    yourturn();
}
```

Fonction gettid()

```
1  #define _GNU_SOURCE
2  #include <unistd.h>
3
4      pid_t gettid(void);
```

```c
void* routine(){
    printf("Process ID %d \n", getpid());
    pid_t tid = gettid();
    printf("Thread ID %d\n", tid);
    return NULL;
}

int main(int argc, char const *argv[])
{
    pthread_t t1,t2;

    if (pthread_create(&t1, NULL, &routine, NULL))
    {
        return 1;
    }
    if (pthread_create(&t2, NULL, &routine, NULL))
    {
        return 2;
    }

    if (pthread_join(t1, NULL))
    {
        return 3;
    }
    if (pthread_join(t2, NULL))
    {
        return 4;
    }


    return 0;
}
```

Jointure de threads

- Scope d’un thread
- Fonction Join()
- Exemple Join

Scope d’un thread

- Durée de vie du process appelant
- Le thread s’arrêtera brusquement lors de la fin du process

Fonction Join

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

```c
void* myturn(void* arg){
    for(int i = 0; i < 10; i++){
        sleep(1);
        printf("my Turn %d \n", i);
    }
return NULL;
}
void yourturn(){
    for(int i = 0; i < 4; i++){
        sleep(1);
        printf("Your Turn %d \n",i);
    }
}

int main(){
    pthread_t newthread;

    pthread_create(&newthread, NULL, &myturn, NULL);

    yourturn();

    pthread_join(newthread, NULL);
}
```

Variables

- Donnée globale :
  - Processus
  - Thread
- Donnée locale :
  - Processus
  - Petite remarque
  - Thread
  - Param thread
- Remarque

Donnée globale processus

- Lors du fork, fils copie la variable globale du parent
- Modification chez le fils ne modifie pas chez le parent
- Pour partager variable : pipe, mémoire partagée, etc.

```c
int x = 2;
int main(int argc, char const *argv[])
{

    int pid = fork();

    if(pid == -1){
        return 1;
    }
    if (pid == 0)
    {
        x++;
        printf("Valeur de x pour le process %d est : %d\n", getpid(), x);
        exit(0);

    }

    sleep(2);
    printf("Valeur de x pour le process %d est : %d\n", getpid(), x);


    if(pid != 0){
        wait(NULL);
    }
    return 0;
}
```

Donnée globale thread

- Thread partagent les données globales
- Modification chez le fils modifie aussi chez le père
- Partage de variable instantané

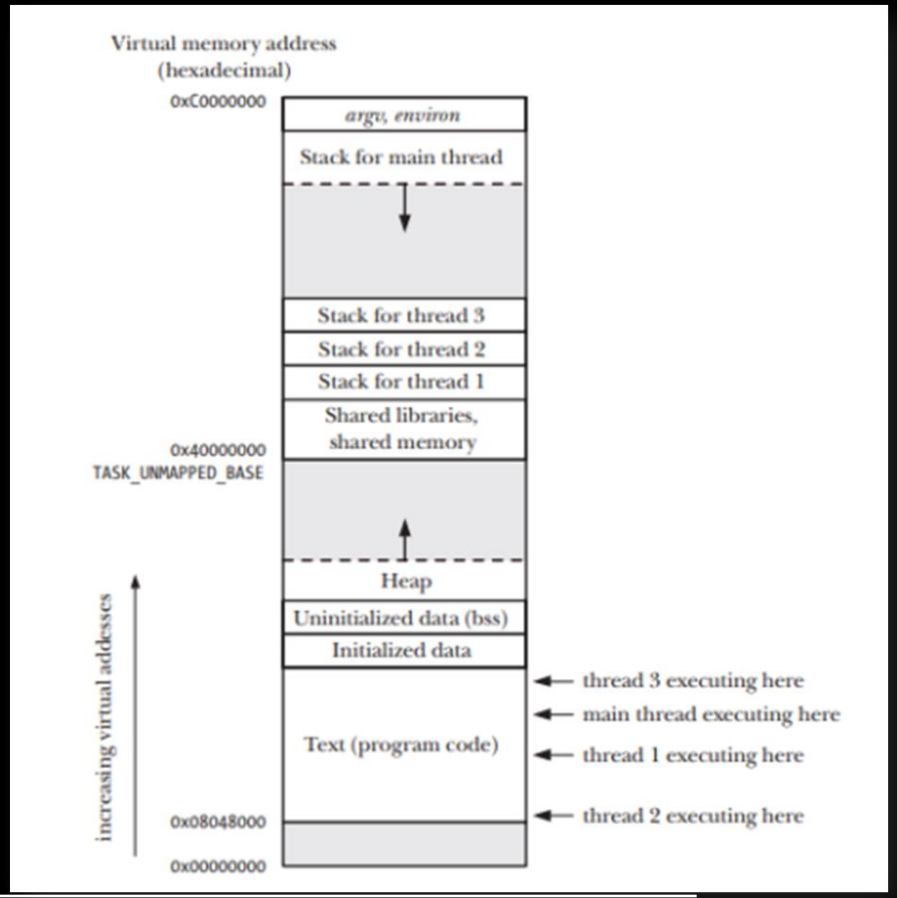
```c
int x = 2;

void* routine1(){
    x +=5;
    pid_t tid = gettid();
    printf("Le Thread %d voit pour valeur de x : %d \n", tid, x);
    return NULL;
}
void* routine2(){
    pid_t tid = gettid();
    printf("Le Thread %d voit pour valeur de x : %d \n", tid, x);
    return NULL;
}

int main(int argc, char const *argv[])
{
    pthread_t t1,t2;

    if (pthread_create(&t1, NULL, &routine1, NULL))
    {
        return 1;
    }
    sleep(2);
    if (pthread_create(&t2, NULL, &routine2, NULL))
    {
        return 2;
    }

    if (pthread_join(t1, NULL))
    {
        return 3;
    }
    if (pthread_join(t2, NULL))
    {
        return 4;
    }


    return 0;
}
```

Variable locale processus

- Identique à la variable globale
- Processus utilise le Copy-On-Write

Variable locale thread

- N'est pas partagée entre thread
- Le fils n'a pas accès aux variables du parent
- Faute à son design



Variable locale en paramètre

- Utilisation de variable locale de parent dans fils.
- On peut passer :
  - Pointeur vers type simple : Int, char, etc.
  - Pointeur vers structure

```
(pthread_create(&t1, NULL, &routine, (void *)&donnees))
```

```
struct SharedData{
    int glob;
    pthread_mutex_t mutex;
};
static void *
routine(void *arg)
{
    struct SharedData *donnees = (struct SharedData *)arg;
    pthread_mutex_lock(&(donnees->mutex));
int loc, j;
for (j = 0; j <1000000 ; j++) {
loc = donnees->glob;
loc++;
donnees->glob = loc;
}
pthread_mutex_unlock(&(donnees->mutex));
return NULL;
}
```

Remarque

- Utiliser pointeur dans processus?
- *Fork* copie tout du parent vers fils
- Le fils set dans un nouvel espace mémoire
- Pointeur sont égaux : 0X5589 et 0X5589
- Pointe vers un autre espace mémoire physique

```c
struct SharedData{
    int val;
};
int main(int argc, char const *argv[])
{
    struct SharedData* data;
    // Allocation dynamique pour la structure SharedData
        data = (struct SharedData*)malloc(sizeof(struct SharedData));

        // Vérification si l'allocation a réussi
        if (data == NULL) {
            fprintf(stderr, "Erreur d'allocation mémoire\n");
            return 1;
        }
    data->val = 0;


    int pid = fork();

    if(pid == -1){
        return 1;
    }
    if (pid == 0)
    {
        data->val = 10;
    }

    sleep(5);
    printf("Valeur de x pour le process %d est : %d\n", getpid(), data->val);


    if(pid != 0){
        wait(NULL);
    }
     free(data);
    return 0;
}
```

Synchronisation variable

- Mutex :
  - Mutex statique
  - Mutex dynamique
  - Choix de mutex
- Mutex conditionnel
  - Remarque
- Deadlock
- Remarque

Synchronisation des données

- Gérer la concurrence d'accès
- Processus pas de partage de données de base :
  - Créer un moyen de partage : pipe, mémoire partagée, etc.
  - Gérer l'accès
- Thread doit direct gérer l'accès :
  - Mutex
  - Mutex conditionnel

```c
static int glob = 0;
static void *
routine(void *arg)
{

int loc, j;
for (j = 0; j <100000000; j++) {
loc = glob;
loc++;
glob = loc;
}
return NULL;
}
int
main(int argc, char *argv[])
{
pthread_t t1, t2;

if (pthread_create(&t1, NULL, &routine, NULL))
    {
        return 1;
    }
    if (pthread_create(&t2, NULL, &routine, NULL))
    {
        return 2;
    }

    if (pthread_join(t1, NULL))
    {
        return 3;
    }
    if (pthread_join(t2, NULL))
    {
        return 4;
    }

    printf("Valeur de la variable globale : %d \n", glob);
```

```c
static int glob = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static void *
routine(void *arg)
{
    pthread_mutex_lock(&mutex);
int loc, j;
for (j = 0; j <100000000 ; j++) {
loc = glob;
loc++;
glob = loc;
}
pthread_mutex_unlock(&mutex);
return NULL;
}
int
main(int argc, char *argv[])
{
pthread_t t1, t2;

if (pthread_create(&t1, NULL, &routine, NULL))
    {
        return 1;
    }
    if (pthread_create(&t2, NULL, &routine, NULL))
    {
        return 2;
    }

    if (pthread_join(t1, NULL))
    {
        return 3;
    }
    if (pthread_join(t2, NULL))
    {
        return 4;
    }

    printf("Valeur de la variable globale : %d\n", glob);
```

- Mutex statique :

Initialisation mutex statique

```
#include <pthread.h>
    pthread_mutex_t mutex =
       PTHREAD_MUTEX_INITIALIZER;
```

- Mutex dynamique :

— La fonction d'initialisation :

Initialisation mutex dynamique

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

— La fonction de destruction :

Destruction mutex dynamique

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

```c
struct SharedData{
    int glob;
    pthread_mutex_t mutex;
};
static void *
routine(void *arg)
{
    struct SharedData *donnees = (struct SharedData *)arg;
    pthread_mutex_lock(&(donnees->mutex));
int loc, j;
for (j = 0; j <1000000 ; j++) {
loc = donnees->glob;
loc++;
donnees->glob = loc;
}
pthread_mutex_unlock(&(donnees->mutex));
return NULL;
}
```

```c
int
main(int argc, char *argv[])
{
pthread_t t1, t2;
struct SharedData donnees;
donnees.glob = 0;
if (pthread_mutex_init(&(donnees.mutex), NULL) != 0) {
        fprintf(stderr, "Erreur lors de l'initialisation du mutex\n");
        return 1;
    }



if (pthread_create(&t1, NULL, &routine, (void *)&donnees))
    {
        return 1;
    }
    if (pthread_create(&t2, NULL, &routine,(void *)&donnees))
    {
        return 2;
    }


    if (pthread_join(t1, NULL))
    {
        return 3;
    }
    if (pthread_join(t2, NULL))
    {
        return 4;
    }

    printf("Valeur de la variable globale : %d\n", donnees.glob);

if (pthread_mutex_destroy(&(donnees.mutex)) != 0) {
        fprintf(stderr, "Erreur lors de la destruction du mutex\n");
        return 6;
    }
```

Choix du mutex

- Mutex statique :
  - Petite portée
  - Durée de vie du processus
  - Peu flexible car définis en compilation
  - Peu complexe donc plus simple
- Mutex dynamique :
- Grande portée
- Durée de vie définie en allouant/détruisant la mémoire
- Grande flexibilité : alloué/libéré lors de l'exécution
- Complexe

Mutex conditionnel

- Mutex : garantis l'accès exclusif
- Variable de condition :
  - Signale ou attend une condition spécifique
  - Thread peut signaler la condition ou peut attendre la satisfaction de la condition

Initialisation mutex conditionnel

```
#include <pthread.h>
    int pthread_cond_init(pthread_cond_t *
       cond, const pthread_condattr_t *attr)
       ;
```

Wait mutex conditionel

```
#include <pthread.h>
    int pthread_cond_wait(pthread_cond_t *cond,
       pthread_mutex_t *mutex);
```

Signal mutex conditionnel

```
#include <pthread.h>
    int pthread_cond_signal(pthread_cond_t *
       cond);
```

Broadcast mutex conditionnel

```
#include <pthread.h>
    int pthread_cond_broadcast(pthread_cond_t *
       cond);
```

```c
char ressource_partagee[] = "texte";
int condition = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

void *routine1(void *arg) {
    pthread_mutex_lock(&mutex);
    char str[8] = " thread1";
    condition = 1;
    strcat(ressource_partagee, str);
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);

    return NULL;
}

void *routine2(void *arg) {
    pthread_mutex_lock(&mutex);
    while (condition != 1) {
        pthread_cond_wait(&cond, &mutex);
    }

    printf("J'ai lu %s\n", ressource_partagee);

    pthread_mutex_unlock(&mutex);

    return NULL;
}
```

```c
int main() {
    pthread_t t1, t2;

    if (pthread_create(&t2, NULL, &routine2, NULL)) {
        return 2;
    }

    sleep(5);

    if (pthread_create(&t1, NULL, &routine1, NULL)) {
        return 1;
    }

    if (pthread_join(t1, NULL)) {
        return 3;
    }
    if (pthread_join(t2, NULL)) {
        return 4;
    }

    return 0;
}
```

Remarque

- Wait entourer d’un while => faux réveil
- Ordre du signal ensuite unlock :
  - Modèle signal and wait
  - Éviter les faux réveils
  - Éviter les courses critiques

Deadlock

- Processus A détient la ressource 1 et demande la ressource 2
- Processus B détient la ressource 2 et demande la ressource 1
- Situation problématique et indésirable

```c
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;

void *fonction_thread1(void *arg) {
    printf("Thread 1 : Attente de mutex1\n");
    pthread_mutex_lock(&mutex1);

    printf("Thread 1 : Verrouillage de mutex1\n");
    sleep(1);

    printf("Thread 1 : Attente de mutex2\n");
    pthread_mutex_lock(&mutex2);


    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);

    return NULL;
}

void *fonction_thread2(void *arg) {
    printf("Thread 2 : Attente de mutex2\n");
    pthread_mutex_lock(&mutex2);

    printf("Thread 2 : Verrouillage de mutex2\n");

    sleep(1);

    printf("Thread 2 : Attente de mutex1\n");
    pthread_mutex_lock(&mutex1);


    pthread_mutex_unlock(&mutex1);
    pthread_mutex_unlock(&mutex2);

    return NULL;
}
```

```c
int main() {
    pthread_t thread1, thread2;

    // Création des threads
    pthread_create(&thread1, NULL, fonction_thread1, NULL);
    pthread_create(&thread2, NULL, fonction_thread2, NULL);

    // Attente de la fin des threads
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return 0;
}
```

Conclusion

- Ressemblant à la surface mais différent
- Pas de système supérieur
- Chacun à ses forces et faiblesses
- Choix dépends du but du programme
- Processus :
  - Pas besoins de manipuler de données
  - Peut écraser complètement le processus (execlp)
- Thread :
  - Manipule beaucoup les données en commun
  - Création rapide pour par ex petite tâche

MERCI