Chapitre 4

Les threads

4.1 Introduction

Le modèle de processus décrit au chapitre précédent est un programme qui s'exécute selon un chemin unique avec un seul compteur ordinal. On dit qu'il a un **flot de contrôle unique** ou un seul **thread**. De nombreux systèmes d'exploitation modernes offrent la possibilité d'associer à un même processus plusieurs chemins d'exécution ou multithread (figure 4.1). Ils permettent ainsi l'exécution simultanée des parties d'un même processus. Chaque partie correspond à un chemin d'exécution du processus. Le processus est vu comme étant un ensemble de ressources (code exécutable, segments de données, de fichiers, de périphériques, etc.) que ces parties appelées flots de contrôle ou processus légers (*threads* en anglais) partagent. Chaque flot de contrôle (thread) a cependant, en plus des ressources communes, sa propre zone de données ou de variables locales, sa propre pile d'exécution, ses propres registres et son propre compteur ordinal.

4.1.1 Avantages

Comparativement aux processus à un flot de contrôle unique, un *thread* ou processus léger avec plusieurs flots de contrôle présente plusieurs avantages, notamment :

- **Réactivité**. Le processus léger continue à s'exécuter même si certaines de ses parties sont bloquées.
- **Partage de ressources**.
- **Économie d'espace mémoire et de temps**. Par exemple sous Solaris, la création d'un processus est 30 fois plus lente que celle d'un proces-

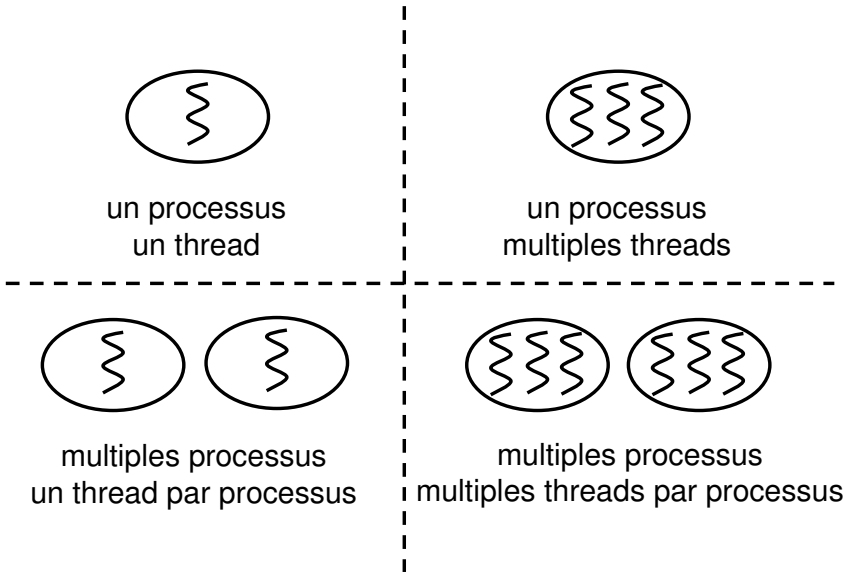

FIG. 4.1 – Threads vs. processus.

sus thread.

4.1.2 Threads utilisateur et noyau

La majorité des systèmes permettent le **multiflot** (*multithreading*). Ils sont offerts soit au niveau utilisateur, soit au niveau noyau (voir le Chapitre **??** *Ordonnancement des processus,* pour les détails sur l'ordonnancement des threads et des processus).

Les threads utilisateur sont supportés au-dessus du noyau et sont implantés par une bibliothèque de threads au niveau utilisateur (par exemple `pthread` sous Linux ou `thread` dans Solaris). Ils sont portables sur différentes plate-formes. Ils sont gérés par une application où le blocage du thread peut bloquer le processus complet. Le changement de contexte est rapide.

Les threads noyau sont directement supportés par le noyau du système d'exploitation. Le système d'exploitation se charge de leur gestion et le changement de contexte est lent.

Les threads combinés sont implantés par le système d'exploitation (utilisateur et système). Les threads utilisateur sont associés à des threads

système. La plupart des tâches gestion s'effectuent sous le mode utilisateur.

Les opérations concernant les threads sont notamment la création, la terminaison, la suspension et la relance.

4.2 Services Posix de gestion de threads

Linux ne fait pas de distinction entre les processus et les threads. Un thread est un processus qui partage un certain nombre de ressources avec le processus créateur : l'espace d'adressage, les fichiers ouverts ou autres. Pour la gestion **Posix** de threads, Linux utilise la bibliothèque `pthread`, qui doit être appelée par l'éditeur de liens.

Création de threads

```
int pthread_create (pthread_t *thread ,
                    pthread_attr_t *attr,
                    void *nomfonction,
                    void *arg );
```

Le service `pthread_create()` crée un processus léger qui exécute la fonction `nomfonction` avec l'argument `arg` et les attributs `attr`. Les attributs permettent de spécifier la taille de la pile, la priorité, la politique de planification, etc. Il y a plusieurs formes de modification des attributs.

Suspension de threads

```
int pthread_join(pthread_t *thid,  void **valeur_de_retour);
```

`pthread_join()` suspend l'exécution d'un processus léger jusqu'à ce que termine le processus léger avec l'identificateur `thid`. Il retourne l'état de terminaison du processus léger.

Terminaison de threads

```
void pthread_exit(void *valeur_de_retour);
```

`pthread_exit()` permet à un processus léger de terminer son exécution, en retournant l'état de terminaison.

```
int pthread_attr_setdetachstate(pthread_attr_t *attr,
                                int detachstate);
```

`pthread_attr_setdetachstate()` sert à établir l'état de terminaison d'un processus léger :

- Si `detachstate = PTHREAD_CREATE_DETACHED` le processus léger libérera ses ressources quand il terminera.
- Si `detachstate = PTHREAD_CREATE_JOINABLE` le processus léger ne libérera pas ses ressources. Il sera donc nécessaire d'appeler `phtread_join()`.

▶ **Exemple 1.** Le programme `thread-pid.c` montre l'implantation de threads dans GNU/Linux, ainsi que la récupération du `pid` du thread.

Listing 4.1 – thread-pid.c

```
#include <unistd.h>    // pour sleep
#include <pthread.h>   // pthread_create, pthread_join, pthread_exit
#include <stdio.h>

void *fonction(void *arg)
{
  printf("pid du thread fils = %d\n", (int) getpid());

  while(1); // forever
  return NULL;
}

int main()
{
 pthread_t thread;

 printf("pid de main = %d\n", (int) getpid());
 pthread_create(&thread, NULL, &fonction, NULL);
 while(1); // forever

 return 0;
}
```

Étant donné que aussi bien `main()` que la `fonction_thread()` tournent tout le temps, cela nous permet de regarder de près des détails de leur exécution. Le programme doit être compilé avec la librairie `-lpthread` L'exécution en tâche de fond de `pthread-pid.c` est :

```
leibnitz> gcc -o pthread-pid thread-pid.c -lpthread
leibnitz> pthread-pid &
[1] 24133
leibnitz> pid de main = 24133
pid du thread fils = 24136
```

```
leibnitz> ps x
 PID TTY      STAT   TIME COMMAND
23928 pts/2    S      0:00 -tcsh
24133 pts/2    R      0:53 pthread-pid
24135 pts/2    S      0:00 pthread-pid
24136 pts/2    R      0:53 pthread-pid
24194 pts/2    R      0:00 ps x
```

Le `pid 24133` correspond au processus de `main()`, `24136` est le PID du `fonction_thread()` elle même, mais qui est le processus `24135` qui — en plus— dort ? D'un autre coté, la seule façon de terminer l'exécution du programme précèdent, c'est l'utilisation de la commande `kill`, avec le `pid` correspondant :

```
leibnitz> kill 24133
leibnitz> [1]    Terminated    pthread-pid
```

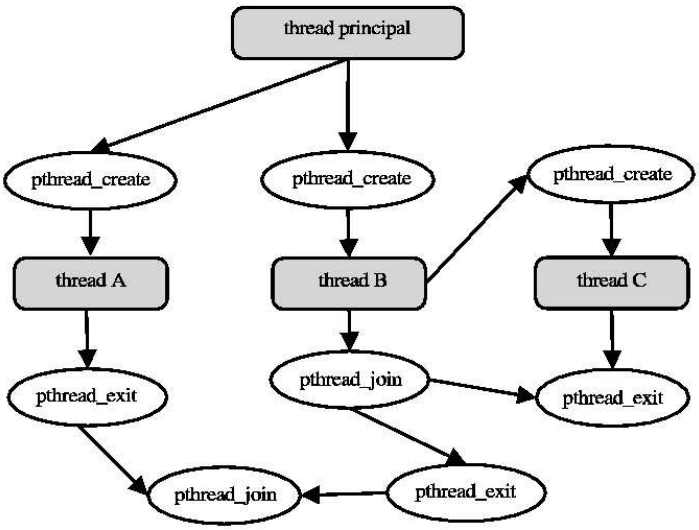

FIG. 4.2 – Exemple de synchronisation de threads.

▶ **Exemple 2.** La figure 4.2 illustre un exemple de création de threads. Le thread principal crée deux threads A et B et attend la fin de leurs exécutions. Un de ces threads crée lui-même un thread C quìl attendra avant de

terminer. Le listing `exemple-threads.c` montre une implémentation de cet exemple.

Listing 4.2 – exemple-threads.c

```
#define _REENTRANT

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


void afficher(int n, char lettre)
{
  int i,j;

  for (j=1; j<n; j++) {
    for (i=1; i < 10000000; i++);
    printf("%c",lettre);
    fflush(stdout);
  }

}


void *threadA(void *inutilise)
{

  afficher(100,'A');
  printf("\n Fin du thread A\n");
  fflush(stdout);

  pthread_exit(NULL);

}


void *threadC(void *inutilise)
{
  afficher(150,'C');
  printf("\n Fin du thread C\n");
  fflush(stdout);

  pthread_exit(NULL);
}



void *threadB(void *inutilise)
```

```
{
  pthread_t thC;
  pthread_create(&thC, NULL, threadC, NULL);


  afficher(100,'B');

  printf("\n Le thread B attend la fin du thread C\n");
  pthread_join(thC,NULL);
  printf("\n Fin du thread B\n");
  fflush(stdout);


  pthread_exit(NULL);
}

int main()
{
  int i;
  pthread_t thA, thB;

  printf("Creation du thread A");
  pthread_create(&thA, NULL, threadA, NULL);
  pthread_create(&thB, NULL, threadB, NULL);
  sleep(1);

  //attendre que les threads aient termine


  printf("Le thread principal attend que les autres se terminent\n");

  pthread_join(thA,NULL);
  pthread_join(thB,NULL);


  exit(0);
}
```

▶ **Exemple 3.** Le programme `threads.cpp` montre l'utilisation des variables globales, ainsi que le passage de paramètres lors des appels de threads.

Listing 4.3 – threads.cpp

```
#include <unistd.h>     //sleep
#include <pthread.h>    //pthread_create, pthread_join, pthread_exit
```

```
#include <stdio.h>

int glob=0;
void* increment(void *); void* decrement(void *);

int main()
{
        pthread_t tid1, tid2;
        printf("ici main[%d], glob = %d\n", getpid(),glob);

      // écration d'un thread pour increment
      if ( pthread_create(&tid1, NULL, increment, NULL) != 0)
              return -1;
      printf( "ici main : creation du thread[%d] avec èsuccs\n", tid1);

      // écration d'un thread pour decrement
      if ( pthread_create(&tid2, NULL, decrement, NULL) != 0)
              return -1;
      printf( "ici main : creation du thread [%d] avec èsuccs\n", tid2);

      // attendre la fin des threads
      pthread_join(tid1,NULL);
      pthread_join(tid2,NULL);
      printf("ici main : fin des threads, glob = %d \n",glob);
   return 0;
}

void* decrement(void *)
{
      int dec=1;
      sleep(1);
      glob= glob - dec ;
      printf("ici decrement[%d], glob = %d\n", pthread_self(), glob);
      pthread_exit(NULL);
}

void* increment (void *)
{
      int inc=2;
      sleep(10);
      glob=glob + inc;
      printf("ici increment[%d], glob = %d\n",pthread_self(), glob);
      pthread_exit(NULL);
}
```

Résultat du programme `threads.cpp` :

```
leibnitz> gcc -o pthread threads.cpp -lpthread
```

```
leibnitz> pthread
ici main[21939], glob = 0
ici main : creation du thread[1026] avec succès
ici main : creation du thread [2051] avec succès
ici increment[21941], glob = 2
ici decrement[21942], glob = 1
ici main : fin des threads, glob = 1
leibnitz>
```

Le programme `thread-param.c` suivant illustre le passage de paramètres entre threads.

▶ **Exemple 4.** Passage de paramètres.

Listing 4.4 – thread-param.c

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

void *thread_function(void *arg);
char message[] = "Hola mundo";

int main()
{
      int res;
      pthread_t a_thread;
      void *thread_result;

      res = pthread_create(&a_thread, NULL, thread_function,
            (void *)message);
      if (res != 0)
      {
         perror("Thread creation failed");
         exit(EXIT_FAILURE);
      }

      printf("En attente que le thread termine...\n");
      res = pthread_join(a_thread, &thread_result);
      if (res != 0)
      {
         perror("Echec dans l'union du thread");
         exit(EXIT_FAILURE);
      }
      printf("Thread termine %s\n", (char *)thread_result);
      printf("Le message est %s\n", message);
      exit(EXIT_SUCCESS);
```

```
}

void *thread_function(void *arg)
{
      printf("La fonction du thread demarre. L'argument est '%s'\n",
            (char *) arg);
      sleep(5);
40    strcpy(message, "Adios!");
      pthread_exit("Merci par le temps de CPU");
}
```

Résultat du programme `thread-param.c` :

```
leibnitz> gcc -o thread-param thread-param.c -lpthread
leibnitz> thread-param
leibnitz> Le thread demarre. L'argument est 'Hola mundo'
En attente que le thread termine...
Thread terminé Merci pour le temps de l'UCT
Le message est Adios!
leibnitz>
```

Algorithmes vectoriels

Il y a plusieurs exemples d'utilisation des threads pour le traitement parallèle de l'information. Un **serveur de fichiers**[1] par exemple, nécessite la gestion et la manipulation de plusieurs requêtes en même temps, qui peuvent être effectuées par l'utilisation des threads. Ainsi, la création d'une thread à chaque requête est envisageable : les threads effectuent les traitements nécessaires et envoient les résultats au client.

La manipulation des matrices et des vecteurs est un autre champ particulièrement propice pour les algorithmes parallèles. La multiplication de deux matrices **a** et **b** de m colonnes = n lignes = 3, par exemple, peut être parallélisé, puisque :

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} . \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} =$$

$$\begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} & a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} & a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} & a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33} \end{pmatrix}$$

[1] Le modèle client-serveur sera étudié au Chapitre **??** *Introduction aux systèmes distribués.*

Ainsi, chaque élément de la matrice résultat peut être un thread. On aura donc 9 thread dans cet exemple.

▶ **Exemple 5.**

Le programme `suma-mat.c` ci-dessous illustre l'utilisation des vecteurs de threads pour faire la somme de deux matrices `A` et `B` de `N` éléments. La somme de chaque ligne est calculée avec des threads indépendants en parallèle. Les composantes sont générées aléatoirement avec `srand()`.

Listing 4.5 – suma-mat.c

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 4                          // Taille des matrices
#define M 10                         // Nb Maximum

void *somme_vectorielle(void *arg);     // Somme des matrices
void imprime(int matrice[N][N]);
pthread_t tid[N];                    // Vecteurs threads
int A[N][N], B[N][N];                // Matrices
int C[N][N];                         // C=A+B
int fila;                            // La file des 2 matrices

int main(void)
{
 int j, k, semilla;
 // Generation des matrices
 semilla = time(NULL);
 srand(semilla);
 for(j=0; j<=N-1; j++)
  for(k=0; k<=N-1; k++)
    {
      A[j][k] = rand()%(M);  //aleatoire 0-M
      B[j][k] = rand()%(M);
    }
 for(j=0; j<=N-1; j++)
  {
    printf("Thread file %d\n",j);
    pthread_create(&tid[j], NULL, somme_vectorielle, (void *) j);
  }
 for(j=0; j<=N-1; j++)
 {
   printf("Attendre thread %d\n",j);
   pthread_join(tid[j],NULL);
 }
 printf("\n\nMatrice A");
```

```
  imprime(A);
  printf("\n\nMatrice B");
  imprime(B);
  printf("\n\n Matriz C=A+B");
  imprime(C);
 return 0;
}

void imprime(int matrice[N][N])
{
 int j, k;
 for(j=0; j<=N-1; j++)
 {
  printf("\n");
  for(k=0; k<=N-1; k++)
   printf("%d \t",matrice[j][k]);
 }
}

void *somme_vectorielle(void *arg)
{
 int f, k;
 f = (int) arg;
 printf("Somme de file %d\n",f);
 for(k=0; k<=N-1; k++)
    C[f][k] = A[f][k]+B[f][k];

 pthread_exit(NULL);
}
```

Exécution du programme `suma-mat.c` :

```
leibnitz> suma-mat
Thread file 0
Thread file 1
Thread file 2
Somme de file 0
Somme de file 2
Thread file 3
Somme de file 3
Attendre thread 0
Attendre thread 1
Somme de file 1
Attendre thread 2
Attendre thread 3

Matrice A
```

```
6       6       0       9
5       5       7       7
2       7       0       6
4       1       6       5

Matrice B
0       3       9       7
9       3       5       6
6       8       6       4
4       8       2       3

Matriz C=A+B
6       9       9       16
14      8       12      13
8       15      6       10
8       9       8       8       leibnitz>
```

▶ **Exemple 6.** Le programme `threads.cpp` montre l'utilisation des variables globales et le passage de paramètres.

Listing 4.6 – threads.cpp

```
#include <unistd.h>     //sleep
#include <pthread.h>    //pthread_create, pthread_join, pthread_exit
#include <stdio.h>

int glob=0;
void* increment(void *); void* decrement(void *);

int main()
{
        pthread_t tid1, tid2;
        printf("ici main[%d], glob = %d\n", getpid(),glob);

      // écration d'un thread pour increment
      if ( pthread_create(&tid1, NULL, increment, NULL) != 0)
              return -1;
      printf( "ici main : creation du thread[%d] avec èsuccs\n", tid1);

      // écration d'un thread pour decrement
      if ( pthread_create(&tid2, NULL, decrement, NULL) != 0)
              return -1;
      printf( "ici main : creation du thread [%d] avec èsuccs\n", tid2);

      // attendre la fin des threads
      pthread_join(tid1,NULL);
```

```
      pthread_join(tid2,NULL);
      printf("ici main : fin des threads, glob = %d \n",glob);
   return 0;
}

void* decrement(void *)
{
      int dec=1;
      sleep(1);
      glob= glob - dec ;
      printf("ici decrement[%d], glob = %d\n", pthread_self(), glob);
      pthread_exit(NULL);
}

void* increment (void *)
{
      int inc=2;
      sleep(10);
      glob=glob + inc;
      printf("ici increment[%d], glob = %d\n",pthread_self(), glob);
      pthread_exit(NULL);
}
```

Trois exécutions du programme `threads.cpp`, sur la machine Unix `jupiter`:

```
jupiter% gcc -o threads threads.cpp -lthread
jupiter% threads
ici main[18745], glob = 0
ici increment[18745], glob = 2
ici decrement[18745], glob = 1
ici main, fin des threads, glob = 1
jupiter% threads
ici main[18751], glob = 0
ici decrement[18751], glob = -1
ici increment[18751], glob = 1
ici main, fin des threads, glob = 1
jupiter% threads
ici main[18760], glob = 0
ici increment[18760], glob = 1
ici decrement[18760], glob = 1
ici main, fin des threads, glob = 1
```

► **Exemple 7.**

Le programme `threads2.cpp` effectue le passage de paramètres à un thread :

Listing 4.7 – threads2.cpp

```
#include <unistd.h> //pour sleep
#include <thread.h> // thr_create et thr_join
#include <stdio.h>

int glob=0;
void* increment(void *inc);
void* decrement(void *dec);

int main()
{
   int inc = 2 , dec = 1;
   printf("ici main[%d], glob = %d\n", getpid(),glob);
   thr_create(NULL,0,increment, &inc,0,NULL);
   thr_create(NULL,0,decrement, &dec,0,NULL);
   while((thr_join(NULL,NULL,NULL)==0));
   printf("ici main, fin des threads, glob = %d \n",glob);
   return 0;
}

void* decrement(void *dec)
{
   int locd=*(int *) dec;
   sleep(1);
   glob -= locd;
   printf("ici decrement[%d], glob = %d\n",getpid( ),glob);
   return (NULL);
}

void* increment (void *inc)
{
   int loci=*(int *) inc;
   sleep(1);
   glob += loci;
   printf("ici increment[%d], glob = %d\n",getpid(), glob);
   return (NULL);
}
```

Exécution programme `threads2.cpp` :

```
jupiter% gcc threads2.cpp -lthread
jupiter% a.out
ici main[19451], glob = 0
ici decrement[19451], glob = 1
```

```
ici increment[19451], glob = 2
ici main, fin des threads, glob = 1
```

4.3 Exercices

1. On sait qu'un thread est créé par un processus parent, mais quelle différence existe-il entre un thread et un processus fils ?
2. Quelle serait un avantage principal à utiliser des threads plutôt que des processus fils ?