Processus multitâches ou multi-threads et synchronisation

October 8, 2018

1 Introduction

1.1 La base : Parallélisme de tâches

1. Solution permettant de montrer (voir à l'écran) qu'un processus peut lancer plusieurs tâches (threads) en parallèle :

```c
#include<stdio.h>
#include<unistd.h> //fork(), getpid(),
#include<sys/types.h> //toutes
#include<sys/wait.h>
#include<stdlib.h>
#include<pthread.h>
#include<stdbool.h>



int  partage=0;

struct args_matrice{
  int** matrice;
  int rows;
  int res;
  bool (*method)(int);
} typedef args_matrice;

bool isPair(int i) {
  return ((i & 1) == 0);
}

bool isImpair(int i) {
  return ((i & 1) == 1);
}

void* threading(void* arg) {
  args_matrice* o = (args_matrice*)arg;
  int res = 0;
  int n = o->rows;

  for(int i=0;i<n;i++)
```

```
34      if(o->method(i)) res++;
35
36    o->res = res;
37    printf("je suis le thread %lu \n",pthread_self());
38    partage+=50;
39    return NULL;
40  }
41
42  int main(int argc,char** argv){
43
44    int n = 5;
45    int** matrice = malloc(n * sizeof(int));
46
47    for(int i=0;i<n;++i){
48      matrice[i]=malloc(n*sizeof(int));
49      for (int j = 0; j < n; ++j) {
50        matrice[i][j] = 1;
51      }
52    }
53
54    pthread_t idpth1,idpth2;
55    struct args_matrice* args1 = malloc(sizeof(args_matrice));
56    args1->matrice = matrice;
57    args1->rows=5;
58    args1->method = isPair;
59
60    struct args_matrice* args2 = malloc(sizeof(args_matrice));
61    args2->matrice = matrice;
62    args2->rows=5;
63    args2->method = isImpair;
64
65    pthread_create(&idpth1,NULL,threading,(void *)args1);
66    pthread_create(&idpth2,NULL,threading,(void *)args2);
67    pthread_join(idpth1,NULL);
68    pthread_join(idpth2,NULL);
69
70    printf("nombre de lignes pair : %d \t nombre de lignes impair %d
        \t partage %d \n",args1->res,args2->res,partage);
71
72    return 0;
73  }
```

threads.c

2 & 3 . Lorsqu'un thread, principal ou secondaire, fait exit(), il termine le processus ! Donc tous les threads seront arrêtés.

4. Problèmes de synchronisation :

```
1  #include<stdio.h>
2  #include<unistd.h> //fork(), getpid(),
3  #include<sys/types.h> //toutes
4  #include<sys/wait.h>
5  #include<stdlib.h>
6  #include<pthread.h>
7  #include<stdbool.h>
```

```c

void *T1 (void * par){
  int * cp = (int*)(par);
  for(int i=0; i < 1500; i++) ++(*cp);
  pthread_exit(NULL);
}

void *T2 (void * par){
  int * cp = (int*)(par);
  for(int i=0; i < 3000; i++) ++(*cp);
  pthread_exit(NULL);
}

int main(){
  pthread_t idT1, idT2;
  int counter = 0;          //donn e en m moir partag e
  if (pthread_create(&idT1, NULL, T1, &counter) != 0)
      printf("erreur creation \n");
  if (pthread_create(&idT2, NULL, T2, &counter) != 0)
      printf("erreur creation \n");

  int res = pthread_join(idT1, NULL);
  res = pthread_join(idT2, NULL);
  printf("Total : %d \n",counter);

  return 0;
}
```

ressourcesPartages.c

2 Exercice : produit scalaire multi-threads

1. Premier schema algorithmique :

```c
#include<stdio.h>
#include<unistd.h> //fork(), getpid(),
#include<sys/types.h> //toutes
#include<sys/wait.h>
#include<stdlib.h>
#include<pthread.h>
#include<stdbool.h>
#include<time.h>




struct args_vecteur{
  int* vecteur1;
  int* vecteur2;
  int index;
} typedef args_vecteur;


void fillVector(int* T,int n){
```

```
22    for(int i=0;i<n;i++)
23      T[i]=rand()%10;
24  }
25
26  void displayVector(int* T,int n){
27    printf("Vector : ");
28    for(int i=0;i<n;i++)
29      printf("%d \t",T[i]);
30    printf("\n");
31  }
32
33  int finalVecteur[10];
34
35  void* threadMult(void* arg) {
36
37    args_vecteur* o = (args_vecteur*)arg;
38    int* v1 = o->vecteur1;
39    int* v2 = o->vecteur2;
40    int ind= o->index;
41    finalVecteur[ind]=v1[ind]*v2[ind];
42    printf("threadNum : %lu    index : %d    vecteur1 : %d  * vecteur2
        : %d  = %d  \n",pthread_self(),ind,v1[ind],v2[ind],v1[ind]*v2[
      ind]);
43
44  }
45
46  int main(int argc,char** argv){
47
48    int sum=0;
49    int taille=atoi(argv[1]);
50    int vect1[taille],vect2[taille];
51    srand(time(NULL));
52
53    fillVector(vect1,taille);
54    fillVector(vect2,taille);
55    displayVector(vect1,taille);
56    displayVector(vect2,taille);
57
58
59
60    pthread_t idpth[taille];
61
62    //thread Multiplication
63    for(int i=0;i<taille;i++){
64
65      struct args_vecteur* args = malloc(sizeof(args_vecteur));
66    args->vecteur1 = vect1;
67      args->vecteur2 = vect2;
68      args->index = i;
69
70      pthread_create(&idpth[i],NULL,threadMult,(void *)args);
71
72    }
73
74     for(int i=0;i<taille;i++)
75        pthread_join(idpth[i],NULL);
76
```

```
77
78  //thread Addition
79
80      for(int i=0;i<taille;i++){
81        sum += finalVecteur[i];
82      }
83
84      printf("resultat : %d \n",sum);
85
86   return 0;
87 }
```

produitScalaire1.c

2. Deuxième schéma algorithmique :