TP4 - Threads; Mémoire

1. Threads

Vous trouvez le code pour ce TP dans le fichier http://lipn.fr/~buscaldi/TP4.tar.gz

Note: pour compiler, souvenez vous d'utiliser le paramètre -lpthread, par exemple:

```
gcc -Wall -o threads1 threads1.c -lpthread
```

1. Exécuter le programme suivant (`threads1.c`):

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NTHREADS 4

void *f1 (void * arg) {
  int i;
  int n = *((int *) arg);
  for (i = 0 ; i < 10 ; i++) {
    printf ("Thread %d: %d\n", n, i);
  }
  pthread_exit (0);
}

int main () {
  pthread_t th[NTHREADS];
  int i;
  int thread_args[NTHREADS+1];

  for(i=1; i<= NTHREADS; i++){
        thread_args[i]=i;
        pthread_create(&th[i-1], NULL, f1, &thread_args[i]);
  }

  return 0;
}
```

Quel est le résultat de l'exécution? Pourquoi? Modifiez le programme parce que le fonctionnement soit correct (vu en cours).

1.1 Ecrire 3 fonctions différentes qui prennent en paramètre un tableau de deux éléments *tab*: la première fonction fait la somme de deux entiers contenus dans le tableau (*tab*[0]+*tab*[1]), la deuxième fait la différence (*tab*[0]-*tab*[1]) , la troisième le produit (*tab*[0]**tab*[1]). Modifiez le programme pour avoir 3 threads et pour que chaque thread exécute une fonction différente.
(note: casting pour le tableau `int* tab = (int *) arg;`)

1.2 Modifiez le programme de la façon suivante: le premier thread fait la somme des entiers d'un tableau *tab*, le deuxième fait la soustraction des entiers du même tableau, et finalement le troisième thread fait la multiplication des deux résultats antérieurs. (Par exemple, si *tab*[0]=10 et *tab*[1]=2, le premier thread obtient *s*=12 et le deuxième thread otbient *d*=8. Le troisième thread doit donc calculer *s*d* donc 12*8=96). Utilisez des variables globales pour *s* et *d*.

2. Modifiez le code du programme lipn.fr/~buscaldi/exoTP2_3.c pour utiliser deux threads qui cherchent dans le tableau, le premier dans la première moitié du tableau et le deuxième dans la deuxième moitié du tableau. Vous pouvez suivre les pistes dans l'enoncé ici dessous, mais des autres solutions sont possibles.
Comparez les résultats avec votre solution du TP2 qui utilisait des fork(): est-ce que c'est plus rapide la version avec les processus ou celle qui utilise les threads?

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <sys/types.h>
#define MAX 10

/* le tableau, val et la variable trouve sont maintenant des variables
globales */
int* a;
int val;
int trouve=0;

void init_tab(int n, int* a) {
    int i;
    fprintf(stderr, "tableau:\n");
    for(i=0; i< n; i++) {
       int r = rand() % MAX;
       a[i]= r;
       fprintf(stderr, "%d ", r);
    }
    fprintf(stderr, "\n");
}

/*
```

```
Il faut redéfinir la fonction cherche comme une fonction à utiliser par un
thread - donc void* cherche (void* arg)...
arg doit inclure les limites de la recherche (tableau de deux éléments)
*/

int main ( int argc, char *argv[] ) {
    int n=0;

    clock_t start, end;

    if ( argc != 3 ) {
       printf( "usage: %s taille_tableau valeur_a_chercher \n", argv[0] );
    } else {
       n=atoi(argv[1]);
       val=atoi(argv[2]);
    }
    srand(time(NULL)); //Initialisation du générateur de nombres aléatoires

    /* avant d'initialiser le tableau il faut lui allouer de la mémoire*/
    init_tab(n, a);


    start = clock();
    /* bloc à remplacer */
    ...
    /* fin bloc à remplacer */
    end= clock();

    if (trouve>0) fprintf(stdout, "%d est dans le tableau\n", val);
    else fprintf(stdout, "%d n'est pas dans le tableau\n", val);

    fprintf(stdout, "\nrecherche finalisée en %f millisecondes\n",
(end-start)*1000/(double)(CLOCKS_PER_SEC));


}
```

2.1 (facultatif-version recursive) Modifier le programme parce que chaque thread puisse utiliser deux sous-threads pour chercher dans sa part de tableau.

3. Le programme suivant (`threads4.c`) simule des interactions entre threads qui se portent comme des clients d'une banque: ils partagent un compte et leurs activités sont soit celle de

déposer de l'argent, soit de le retirer. Le programme assigne aléatoirement le comportement à chaque thread, ainsi que la somme d'argent à déposer ou retirer.

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <pthread.h>

#define MAX 100
#define NITERATIONS 10
#define NTHREADS 3

int compte=0;

void *deposer(void* arg) {
    int q = (*(int *) arg);

    compte=compte+q;
    printf("Déposés: %d\n", q);

    pthread_exit (0);

}

void *retirer(void* arg) {
    int q = (*(int *) arg);

    compte=compte-q;
    printf("Retire: %d\n", q);
    if(compte < 0) {
       fprintf(stderr, "\nOpération réfusée - crédit insuffisant\n");
       compte=compte+q;
    } else {
       printf("Retirés: %d - Solde: %d\n", q, compte);
    }

    pthread_exit (0);
}

int main ( int argc, char *argv[] ) {
    pthread_t th[NTHREADS];
    int params[NTHREADS];

    int r;
    int i, j;
    int amount;
```

```
    srand(time(NULL)); //Initialisation du générateur de nombres aléatoires

    for(j=0; j<NITERATIONS; j++) {
           printf("Iteration: %d\n------------------------\n", (j+1));
           for(i=0; i<NTHREADS; i++){
              r = rand() % 2;
              amount= rand() % MAX;
              params[i]=amount; //on copie pour donner le paramètre au thread
              if(r==0) {
                     /* il s'agit de quelqu'un qui depose */
                     rc=pthread_create(&th[i], NULL, deposer, &params[i]);
              } else {
                     /* il s'agit de quelqu'un qui retire */
                     pthread_create(&th[i], NULL, retirer, &params[i]);
              }
           }

           for(i=0; i<NTHREADS; i++){
              pthread_join(th[i], NULL);
           }

           printf("------------------------\nSolde: %d\n------------------------\n",
compte);
    }

    return 0;
}
```

Or, ce programme à un défaut, en certaines occasions le solde ne correspond pas à celui attendu lorsque un des threads effectue l'opération prevue, par exemple:

```
Solde: 70
------------------------
Iteration: 3
------------------------
Rétire: 14
Rétirés: 14 - Solde: 170
Déposés: 67
Déposés: 47
```

On souhaite modifier le programme pour corriger ce comportement de façon qu'un seul thread à la fois puisse modifier le solde.

2. Mémoire

1. Exécuter le programme suivant (`mem1.c`):

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int g=0;

int main(int argc, char *argv[]) {
  int i,val[1024];
  const char* txt="hello";
  int* adr;
  int *a;
  for (i=0;i<1024;i++)
    {
      val[i] = ((int )getpid())+i;
    }
    fprintf(stdout,"la variable g est à l'adresse %p\n\n",&g);
    fprintf(stdout,"val[0] est à l'adresse %p et vaut %d \n\n",(&(val[0])),val[0]);
    fprintf(stdout,"la chaine txt commence à l'adresse %p avec '%c'
\n\n",(&(txt[0])),txt[0]);
    while(1) {
       fprintf(stdout,"Entrer une adresse où vous voulez lire (en hexa): ");
       fscanf(stdin,"%p",&adr);
       fprintf(stdout,"la case à l'adresse %p  vaut %d (%c comme char)\n",adr,
*adr,*adr);
    }
  return(0);
}
```

Si val[0] est à l'adresse ADR, vérifiez le contenu des adresses ADR+1, ADR+2, ADR+3, ADR+4: est-ce que cela correspond à ce qui est contenu dans le tableau val[1], val[2], val[3], val[4]? Vérifiez le contenu de la chaîne de caractères *txt* à l'adresse ADR_T en examinant les adresses ADR_T+1, ADR_T+2, ADR_T+3, etc… Expliquez la différence avec le tableau *val*.

1.1 Dans un autre terminal, examinez le contenu du fichier /proc/*val[0]*/maps (val[0] contient le pid du processus): les adresses utilisés pour le tableau résident dans quelle région de la mémoire? Dans quelle région de la mémoire vous trouvez les adresses pour *g* et *txt* ? Expliquez les différences. Testez les limites de ce que vous pouvez effectivement lire.

1.2 Considerez le programme suivant (`mem2.c`):

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NTHREADS 2

void *run (void * arg) {
    int i,val[1024];
    int n = *((int *) arg);

    for (i=0;i<10;i++)
    {
      val[i] = (n)+i;
    }
    fprintf(stdout,"Thread %d: adresse de n: %p \n\n",n,&n);
    fprintf(stdout,"Thread %d: val[0] est à l'adresse %p et vaut %d \n\n",n,
(&(val[0])),val[0]);

    pthread_exit (0);
}

int main () {
  pthread_t th[NTHREADS];
  int i;
  int thread_args[NTHREADS+1];
  int* adr;

  fprintf(stdout,"PID du processus: %d\n\n", getpid());

  for(i=0; i< NTHREADS; i++){
        thread_args[i]=(i+1);
        pthread_create(&th[i], NULL, run, &thread_args[i]);
  }

  for(i=0; i< NTHREADS; i++){
        pthread_join(th[i], NULL);
  }

  while(1) {
        fprintf(stdout,"Entrer une adresse où vous voulez lire (en hexa): ");
        fscanf(stdin,"%p",&adr);
        fprintf(stdout,"la case à l'adresse %p  vaut %d\n",adr, *adr);
  }

  return 0;
}
```

Vérifiez dans proc/[PID]/maps la région de mémoire où les tableaux rempli par les threads se trouvent. Modifiez le programme pour avoir 4 threads. Vérifiez les adresses dans proc/[PID]/maps: vous en déduisez quoi? Tester votre hypothèse avec 0 threads.

1.3 Vérifiez la position de la variable *i* et du tableau *th* dans la carte des régions de mémoire.

2. Écrivez un programme qui ouvre un fichier et utilise mmap pour projecter ses premiers 4 octets dans la mémoire (vu en cours: `buf = mmap (0, 4, PROT_READ, MAP_PRIVATE, fd, NULL);`). Essayez de modifier buf[0]. Quel est le résultat? Modifiez les paramètres de mmap pour permettre la modification.

2.1 Modifiez le programme pour vérifier dans quelle région de mémoire mmap alloue la mémoire nécessaire (voir exercice 1).

3. Récuperez le code de la version du programme lipn.fr/~buscaldi/exoTP2_3.c qui utilise fork() pour effectuer la recherche. Utilisez mmap pour communiquer le résultat de la recherche du fils au père.