Fils et semaphores

4 décembre 2020

Exercice 1 : Spinlock v. Mutex.

Quelle est la différence entre un *spinlock* et un *mutex*? Pour mettre en évidence cette différence, téléchargez `http ://www.lsv.fr/~hondet/resources/archos/spinvmut.c` et complétez le de manière a ce que le fil `stupid` s'accapare le spinlock et que `wants` soit forcé d'attendre la libération de la ressource (exclusion mutuelle).

La commande `time cmd` mesure le temps d'occupation CPU en mode kernel (*sys*), le temps d'occupation CPU en mode utilisateur (*user*) et le temps réel (*real*), écoulé, présenté sous la forme

```
real    0m0.000s
user    0m0.000s
sys     0m0.000s
```

si vous utilisez la command *bash*. Il existe aussi un binaire `time`, qui présente les résultats de cette manière,

```
0.00user 0.00system 0 :00.00elapsed ?%CPU (0avgtext+0avgdata 1808maxresident)k
```

où le temps réel est noté *elapsed*.

Que donnera un appel à `time spinvmut` concernant le temps réel et le temps CPU ? Modifiez le code pour utiliser des mutex à la place. Comment devrait évoluer le temps réel et le temps CPU, en lançant `time spinvmut` ? Vérifiez.

Solution : Version avec spinlocks,

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

pthread_spinlock_t resource;

void* stupid() {
  pthread_spin_lock(&resource);
  sleep(4);
  pthread_spin_unlock(&resource);
  pthread_exit(NULL);
}
void* wants() {
  pthread_spin_lock(&resource);
```

```
  printf("At last!\n");
  pthread_spin_unlock(&resource);
  pthread_exit(NULL);
}

int main() {
  pthread_spin_init(&resource, PTHREAD_PROCESS_SHARED);
  pthread_t stupid_th, wants_th;
  pthread_create(&stupid_th, NULL, stupid, NULL);
  sleep(1);
  pthread_create(&wants_th, NULL, wants, NULL);
  pthread_join(wants_th, NULL);
  pthread_join(stupid_th, NULL);
  exit(0);
}
```

ce qui donne, par un appel à `time`,

```
  At last !

  real    0m4.002s
  user    0m2.998s
  sys     0m0.004s
```

Version avec mutex,

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

pthread_mutex_t resource;

void* stupid() {
  pthread_mutex_lock(&resource);
  sleep(4);
  pthread_mutex_unlock(&resource);
  pthread_exit(NULL);
}
void* wants() {
  pthread_mutex_lock(&resource);
  printf("At last!\n");
  pthread_mutex_unlock(&resource);
  pthread_exit(NULL);
}

int main() {
  pthread_mutex_init(&resource, NULL);
  pthread_t stupid_th, wants_th;
  pthread_create(&stupid_th, NULL, stupid, NULL);
```

```
  sleep(1);
  pthread_create(&wants_th, NULL, wants, NULL);
  pthread_join(wants_th, NULL);
  pthread_join(stupid_th, NULL);
  exit(0);
}
```

ce qui donne, par un appel à `time`,

```
  At last !

  real    0m4.002s
  user    0m0.003s
  sys     0m0.004s
```

où l'on voit que, bien que les temps réels soient égaux, on a un différence entre les temps CPU : le *spinlock*, car il réalise une attente active, occupe le processeur, on a donc un temps CPU élevé. Le *mutex*, réalisant une attente passive, libère les ressources, et prend donc peu de temps CPU.

Exercice 2 : La fonction d'Hénon, partie II.

On va (re) calculer l'orbite d'un système dynamique de dimension 2. La fonction d'Hénon est définie par le système

$$H_{a,b} = \begin{cases} x_{n+1} = a - by_n - x_n^2 \\ y_{n+1} = x_n. \end{cases} \tag{1}$$

On utilisera un thread pour calculer la suite $(x_n)_n$ et un autre thread pour la suite $(y_n)_n$.

Proposez un moyen de synchronisation permettant d'assurer l'entrelacement des calculs de x_n et de y_n. Mettez le en œuvre.

On peut ensuite créer un modèle dit producteur consommateur :

- un thread calcule l'orbite et stocke les données dans un buffer tournant,
- l'autre thread lit les données du buffer et les écrit dans un fichier.

Le fichier `henon.dat` sera sous la forme `0.3415 1.2451` où le premier nombre est x_n et le deuxième y_n.

On pourra tracer la fonction avec la commande `gnuplot henon.p` après avoir téléchargé le script `http://www.lsv.fr/~hondet/resources/archos/henon.p`. Cela produira un fichier `henon.png`.

Pour $a = 1\ .4$ et $b = -0\ .3$, la fonction est chaotique (au sens de Devaney), c'est-à-dire,

- la fonction est sensiblement dépendante des conditions initiales (effet papillon, imprédictabilité) ;
- la fonction est topologiquement transitive (indécomposabilité) ;
- les points périodiques sont denses dans le domaine de définition de $H_{a,b}$ (régularité).

et possède un attracteur étrange.

Pour plus d'informations sur les systèmes dynamiques, voir *An introduction to Chaotic Dynamical Systems*, Robert L. Devaney, Westview.

Solution :

Simple

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

#define _bound 100

// Create mutexes that indicate there is a resource available.
pthread_mutex_t xrsrc;
pthread_mutex_t yrsrc;

// Parameters
double _a = 1.4, _b = -0.3;
double x, y;
double xp = 0.5, yp = 0.5;

void printer() {
  printf("%+0.10lf %+0.10lf\n", x, y);
}

void c_xn(int bound) {
  for (int i = 1; i < bound; i++) {
    // Enclose computation in mutexes
    pthread_mutex_lock(&yrsrc);
    x = _a - _b * yp - xp * xp;
    pthread_mutex_unlock(&xrsrc);
  }
  exit(0);
}

void c_yn(int bound) {
  for (int i = 1; i < bound; i++) {
    // Enclose computation in mutexes
    pthread_mutex_lock(&xrsrc);
    y = xp;
    pthread_mutex_unlock(&yrsrc);
    xp = x; yp = y;
    printer();
  }
  exit(0);
}

int main() {
  // Setting mutexes
  pthread_mutex_init(&xrsrc, NULL);
  pthread_mutex_init(&yrsrc, NULL);
  // Lock one of two mutexes that will be unlocked by a thread
  pthread_mutex_lock(&xrsrc);

  // Creating threads
```

```
  pthread_t xn_thr, yn_thr;
  pthread_create(&yn_thr, NULL, (void *(*)(void*)) c_yn, (void*) _bound);
  pthread_create(&xn_thr, NULL, (void *(*)(void*)) c_xn, (void*) _bound);

  // Wait for termination
  pthread_join(xn_thr, NULL);
  pthread_join(yn_thr, NULL);

  exit(0);
}
```

Producteur-consommateur

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <time.h>
#include <pthread.h>
#include <semaphore.h>
#include <string.h>

#define _bound 200

int _npts = 10;

// Producer consumer
double *pc_values;
int pc_wcursor = 0;
int pc_rcursor = 0;
sem_t pc_freespace;
sem_t pc_resource;

double _a = 1.4, _b = -0.3;


void compute_orbit() {
  double x, y;
  double xp = 0.5, yp = 0.5;
  for (int i = 1; i < _bound; i++) {
    x = _a - _b * yp -  xp * xp;
    // Assert there is free space in the queue
    if (sem_wait(&pc_freespace)) perror("semwait");
    // Put a value in the queue
    pc_values[pc_wcursor] = x;
    // Signal that there is a resource to read
    if (sem_post(&pc_resource)) perror("sempost");
    // Compute next slot in the queue
```

```
    pc_wcursor = (pc_wcursor + 1) % (2 * _npts);

    y = xp;
    if (sem_wait(&pc_freespace)) perror("semwait");
    pc_values[pc_wcursor] = y;
    pc_wcursor = (pc_wcursor + 1) % (2 * _npts);
    if (sem_post(&pc_resource)) perror("sempost");

    xp = x; yp = y;
  }
  pthread_exit(NULL);
}

void writer() {
    int fd;
    if ((fd = open("henon.dat", O_DSYNC | O_CREAT | O_RDWR | S_IRUSR)) == -1)
        perror("henon.dat");
    char buf[100];
    double x, y;
    for (int i = 1; i < _bound; i++) {
      // Wait for some resource to be available
      if (sem_wait(&pc_resource)) perror("wait for resource");
      // Read values
      x = pc_values[pc_rcursor];
      // Update read cursor
      pc_rcursor = (pc_rcursor + 1) % (2 * _npts);
      // Signal that there is more free space
      if (sem_post(&pc_freespace)) perror("sempost free space");

      if (sem_wait(&pc_resource)) perror("semwait");
      y = pc_values[pc_rcursor];
      pc_rcursor = (pc_rcursor + 1) % (2 * _npts);
      if (sem_post(&pc_freespace)) perror("sempost");

      sprintf(buf, "%+0.10lf %+0.10lf\n", x, y);
      if (write(fd, buf, sizeof(char) * strlen(buf)) == -1) perror("write");
    }
    close(fd);
    pthread_exit(NULL);
}

int main() {
  // All slots are free initially
  sem_init(&pc_freespace, PTHREAD_PROCESS_SHARED, 2 * _npts);
  sem_init(&pc_resource, PTHREAD_PROCESS_SHARED, 0);
  pc_values = malloc(2 * _npts * sizeof(double));
  pthread_t orbit_thr, writer_thr;
  pthread_create(&orbit_thr, NULL, (void *(*)(void*)) compute_orbit, NULL);
  pthread_create(&writer_thr, NULL, (void *(*)(void*)) writer, NULL);
  if (pthread_join(orbit_thr, NULL)) perror("join orbit");
  if (pthread_join(writer_thr, NULL)) perror("join writer");
```

```
  exit(0);
}
```

Exercice 3 : Mandelbrot.

Soitcun nombre complexe. On considère la série $z_0 = 0$ et $z_{n+1} = z_n^2 + c$ pour $n \geq 0$. L'ensemble de Mandelbrot est défini comme l'ensemble des valeurs c telles que la série des z_n est bornée. On sait que cela est le cas si z_n ne sort jamais d'un cercle de rayon 2 autour de 0. Si jamais la série sort de ce cercle, soit $m(c)$ le plus petit indice n tel que c'est le cas. Une application populaire pour $m(c)$ est de créer de jolies images ; on associe l'écran avec un rectangle et chaque pixel avec la valeur c qui y correspond ; le pixel est ensuite peint avec une couleur correspondant à $m(c)$. La page web du cours contient un tel programme. Votre tâche consiste à l'accélérer en lançant plusieurs threads en parallèle. On peut utiliser `cat /proc/cpuinfo` pour voir combien de cœurs une machine a. Chaque thread va donc travailler sur une partie différente de l'image.

Exercice 4 : Sémaphores inter processus (facultatif).

Les semaphores, bien qu'introduits avec les threads, permettent également la synchronisation inter processus. Pour ce faire, l'API System V fournit des procédures *IPC* pour *Inter Process Communication*. Par exemple, on peut obtenir des sémaphores via `semget`. Cette commande utilise des clefs IPC permettant d'identifier de manière unique un ensemble de sémaphores.

Ces clefs peuvent être générées par la fonction `ftok`.

Pour expérimenter un peu, écrivez un programme créant une clef et deux sémaphores avec cette clef. Lancez le programme et exécutez la commande `ipcs -s`. Que constate-t-on ? On en déduira les précautions à prendre lorsque l'on utilise des clefs. La commande `ipcrm` existe.

On peut maintenant implémenter la fonction d'Hénon multi processus avec des sémaphores. L'utilisation de sémaphores partagés reste assez fastidieuse.