TP6 Réseaux de Kahn [1]

Nous allons implanter le crible d'Ératosthène qui permet de construire la liste des nombres premiers en nous basant sur le modèle de programmation des réseaux de Kahn [1]. Ces réseaux ont été décrits par Gérard Berry de la façon suivante :

> « Les réseaux de Kahn sont le plus ancien et le plus simple des modèles pour le parallélisme coopératif. Il s'agit de processus qui communiquent par des files d'attente illimitées. Dans ce modèle, le parallélisme est un concept qui simplifie l'architecture à l'inverse des langages les plus classiques qui voient le parallélisme comme une compétition sur des ressources. Une propriété majeure des réseaux de Kahn est que le résultat d'un calcul est déterministe et ne dépend pas de l'ordonnancement interne des actions en parallèle. Bien que limité dans son pouvoir expressif, le modèle initial de G. Kahn est encore beaucoup utilisé dans les calculs basés sur les flux, par exemple en imagerie pour la télévision haute définition. Ce modèle a été étendu pour traiter des structures de données complexes par Kahn et MacQueen et pour modéliser le lambda calcul par Berry et Curien. »

Dans le langage introduit par Gilles Kahn, un programme est composé de processus qui lisent en entrée des flots de valeurs (des suites potentiellement infinies de valeurs) et produisent des flots de valeurs. Les processus sont exécutés en parallèle et connectés par des canaux. Les canaux conservent l'ordre des messages et peuvent « buffeuriser » un nombre infini de messages.

Dans ce langage, le programme qui crée la suite des entiers naturels (qui commence à 2) et qui l'affiche s'écrit de la façon suivante :

```
Process INTEGERS out Qo;
  Vars N;
  N = 1;
  repeat
    N = N + 1;
    PUT(N,Qo);
  forever
Endprocess;
```

```
Process OUTPUT in Qi;
  repeat
    PRINT(GET(Qi))
  forever
Endprocess;

Start doco channels Q;
  INTEGERS(Q); OUTPUT(Q);
closeco;
```

Ce programme est composé de deux processus : `INTEGERS` et `OUTPUT`. Le processus `INTEGERS` calcule la suite de valeurs 2, 3, 4, ... qu'il émet sur un canal de sortie `Qo`. Le processus `OUTPUT` lit les valeurs qui arrivent sur un canal `Qi` et les affiches. L'exécution de ce programme est l'exécution parallèle des deux processus où la sortie du processus `INTEGERS` est connectée par une FIFO à l'entrée de `OUTPUT` par un canal `Q`.

1. Ces exercices ont été créés par Louis Mandel.

Les codes sources donnés ci-dessous se trouvent également à l'adresse suivante :

`http://www.di.ens.fr/~pouzet/cours/systeme/tp06/tp-06.tgz`

Question 1. Définir en C l'équivalent du programme précédent qui calcule et affiche la suite des entiers naturels. Ce programme sera composé de deux threads qui communiquent par un canal manipulé par les fonctions suivantes :

- `struct channel *new_channel();` qui crée un canal ;
- `void put(int n, struct channel *chan);` qui émet un entier sur un canal ;
- `int get(struct channel *chan);` qui récupère un entier sur un canal.

Question 2. Vérifier que l'implantation suivante des canaux est incorrecte :

```
struct channel {
  int buffer;
};
struct channel *new_channel() {
  struct channel *chan = (struct channel *) malloc(sizeof(struct channel));
  return chan;
}
void put(int n, struct channel *chan) {
  chan->buffer = n;
  return ;
}
int get(struct channel *chan) {
  int res;
  res = chan->buffer;
  return res;
}
```

On souhaite maintenant garantir qu'il y a une alternance entre les lectures et les écritures dans le canal. Pour cela, on étend la définition du type `struct channel` de la façon suivante :

```
struct channel {
  sem_t vide;
  sem_t plein;
  int buffer;
};
```

Le sémaphore `vide` sert à garantir que la fonction `get` ne va jamais lire deux fois la même valeur et le sémaphore `plein` que `put` ne va pas écraser une valeur qui n'a pas encore été lue.

Question 3. Modifier l'implantation des fonctions `new_channel`, `put` et `get` pour garantir un comportement correct des canaux. Ces fonctions utiliseront les fonctions `sem_init`, `sem_wait` et `sem_post`. (Pourquoi ne faudrait-il pas utiliser les verrous, « les mutexs », dans ce cas?)

Question 4. Exécuter le réseau de Kahn suivant :

```
Process PROC MSG in Qi out Qo;
  repeat
    PRINT("%s: PUT", MSG);
    PUT(1, Qo);
    PRINT("%s: GET", MSG);
    GET(Qi);
  forever
Endprocess;

Start doco channels Q1, Q2;
  PROC("A", Q1, Q2); PROC("B", Q2, Q1);
closeco;
```

Question 5. Modifier le processus `PROC` de la façon suivante et exécuter le programme.

```
Process PROC MSG in Qi out Qo;
  repeat
    PRINT("%s: PUT", MSG);
    PUT(1, Qo);
    PRINT("%s: PUT", MSG);
    PUT(1, Qo);
    PRINT("%s: GET", MSG);
    GET(Qi);
    PRINT("%s: GET", MSG);
    GET(Qi);
  forever
Endprocess;
```

Avec l'implantation actuelle, les canaux de communication entre les processus sont munis de buffers de taille 1. Comme l'a montré l'exemple précédent, cela peut introduire des blocages lors de l'exécution de réseaux de Kahn corrects. Nous voulons donc disposer de canaux munis de buffers de taille `N`.

Question 6. En modifiant le code suivant, définir des canaux munis de buffers bornés. On rappelle que dans les réseaux de Kahn, il n'y a qu'un producteur et un consommateur associés à un canal. (N'oubliez pas de protéger les structures de données partagés avec un mutex: `pthread_mutex_init`, `pthread_mutex_lock` et `pthread_mutex_unlock`).

```
struct channel {
  int buffer[N];
  int libre;
  int prochain;
};
struct channel *new_channel() {
  struct channel *chan = (struct channel *) malloc(sizeof(struct channel));
  if ( NULL == chan ) { return NULL; }
  chan->libre = 0;
  chan->prochain = 0;
```

```
  return chan;
}
void put(int n, struct channel *chan) {
  chan->buffer[chan->libre] = n;
  chan->libre = (chan->libre + 1) % N;
  return ;
}
int get(struct channel *chan) {
  int res;
  res = chan->buffer[chan->prochain];
  chan->prochain = (chan->prochain + 1) % N;
  return res;
}
```

Dans l'article « Coroutines and Networks of Parallel Processes » [2], G. Kahn et D. MacQueen proposent le code du crible d'Ératosthène suivant :

```
Process FILTER PRIME in Qi out Qo;
  Vars N;
  repeat
    N = GET(Qi);
    if (N MOD PRIME) != 0 then PUT(N, Qo)
  forever
Endprocess;

Process SIFT in Qi out Qo;
  Vars PRIME;
  PRIME = GET(Qi);
  PUT(PRIME, Qo);
  doco channels Q;
    FILTER(PRIME, Qi, Q); SIFT(Q, Qo);
  closeco
Endprocess;

Start doco channels Q1, Q2;
  INTEGERS(Q1); SIFT(Q1, Q2); OUTPUT(Q2);
closeco;
```

Question 7. Reprogrammer cet algorithme en C.

Références

[1] Gilles Kahn. The semantics of simple language for parallel programming. In *Proceedings of IFIP 74 Conference*, pages 471–475, 1974.

[2] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In *IFIP Congress*, pages 993–998, 1977.