TD7 : Correction

A Passons à la banque

Vu en TD

B PRAM

Etant donné un tableau d'entiers de taille t et un entier p, on cherche à déterminer si p appartient au tableau.

B.1 Proposez un algorithme CREW en pseudo-code pour une machine PRAM à n éléments.

Correction :

```
init : appartient = false.
Pour tout les processus i
  l=entierSup(t/n).
   Pour tout les  j de 0 a l-1 et i*l+j < t.
    Si t[i*l+j]=p alors
      appartient = true
```

B.2 Rappelez la définition de la complexité et de l'efficacité. Quelles en sont les valeurs pour votre algorithme ?

Correction : la valeur de d'efficacité est : $e = \frac{C_{lineraire}}{n \times C_{pram}}$ Les valeur pour notre algorithme sont : $1 = \frac{t}{n \times \frac{t}{n}}$. Le problème est bien réparti sur les PRAM.

C Mutex & sémaphores

C.1 Rappelez les algorithmes associés aux deux opérations $P()$ et $V()$ sur les sémaphores. Pourquoi ces opérations doivent-elles être atomiques ?

Correction : A l'initialisation du sémaphore on lui donne une valeur (init) qui sera le nombre de jetons en sa possession.

- $P()$ ou $P(i)$ est une prise d'un jeton ou de i jetons et bloquant en cas pénurie.
- $V()$ ou $V(i)$ rend le ou les i jetons.

Ces procédures doivent être atomique pour x raison :

1. Les opération d'incrémentation et de décrémentation doivent être atomique afin de garder la consistance du nombre de jetons.
2. Il ne faut pas que deux thread puisse avoir tout deux le même jeton en appelant la fonction $P()$ en même temps.

C.2 Rappelez ce qu'est un mutex et proposez une mise en œuvre des sémaphores à l'aide de mutex.

Correction : Un mutex est un véroux interdisant deux thread d'accéder à une même ressource en même temps. Voici les algorithmes pour faire un sémaphore à l'aide d'un mutex avec un pseudo langage de programmation :

```
init : m = mutex, init =n (nombre de jetons), wl=(LinkedList)malloc
    ...
 P(i) :
  lock(m);
  add(wl,thisThread);
```

```
  while (i>n)
    unlock(m);
    wait();
    lock(m);
  remove(wl,thisThread);
  n=n-i;
  unlock(m);
  if (n>0 && size(wl)>0)
    wakeup(getHead(wl)); // Au cas ou il reste des jetons. On peut
       aussi faire i notify dans V.

V(i):
  lock(m);
  n=n+i;
  unlock(m);
  if (size(wl)>0)
    wakeup(getHead(wl));
```

C.3 Ecrivez l'algorithme en Java en utilisant synchronized.

Correction :

```
class Semaphor
  int nb;
  Semaphor(int nb){
    this.nb=nb;
  }
  synchronized P(int i){
    while (i>nb)
      wait();
    nb-=i;
    if (nb>0)
      notify();//S'il reste des jetons. On peut aussi faire i
         notify dans V.
  }
  synchronized V(int i) {
    nb+=i;
    notify();
  }
}
```

Attention ! petit rappel. Le moniteur possède une liste d'attente.

C.4 Résolvez le problème du producteur-consommateur à l'aide d'un sémaphore.

Correction :

```
init : s(0) un semaphore
producteur :
  s.v()
consomateur:
  s.p()
```

D Thread & concurrence

Sur une table de 12, il y a une seule salière.

D.1 Quelles sont les propriétés nécessaires pour réaliser l'exclusion mutuelle ?

Correction :

EM : L'exclusion mutuelle, si un thread est en SC, aucun autre thread ne peut y être.

P : Le progrès, si un groupe de thread demande à entrer en SC, l'un d'eux doit obtenir d'entrer en SC.

E : L'équité, tout thread demandant à entrer en SC doit y entrer (au bout d'un certain temps).

D.2 Ecrivez un algorithme pour passer la salière garantissant ces propriétés.

Correction : Il suffit d'utiliser un Mutex avec une liste d'attente des prétendants. (Voir exercice précédent) Ou vous pouvez utiliser une exclusion mutuelle en anneau.

E Horloge matricielle

Vu en TD