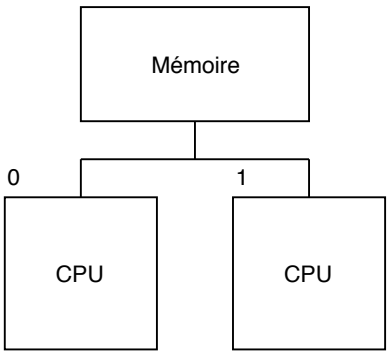
Systèmes d'exploitation : processus

William SCHMITT

2018-2019

1 Problème

A l'époque où la mémoire était plus rapide que les processus, on a eu l'idée de rajouter plusieurs unités de calcul.



Exemple de code assembleur exécuté sur deux CPU en parallèle.

```
; début section critique
load $i, %r0
inc %r0
store %r0, $i
; fin section critique
```

Quel est le résultat stocké dans i ?

Selon l'ordre des opérations, on peut obtenir 1 ou 2. Intervient alors l'algorithme de Dekker (1966), permettant de garantir l'obtention du résultat souhaité : 2.

2 Solutions

2.0 Booléens et sections critiques

Une première idée est de définir des sections critiques à l'aide de booléens.

```
bool occ = false;
debut_sc() {
    alpha:  if (occ)
                goto alpha;
            occ = true;
}

fin_sc() {
    occ = false;
}
```

Ce code est faux car on n'a pas d'exclusion mutuelle.

2.1 Ecriture de `occ` avant le test

```
bool occ = false;
debut_sc() {
    bool old = occ
    occ = true;
    alpha:  if (occ)
                old = occ;
                goto alpha;
}

fin_sc() {
    old = false
}
```

Cela ne résoud pas le problème, puisque la variable old est locale et en cas de lancement simultané, on se ramène au même cas de figure que précédemment.

2.2 Tickets

Il faut ici connaître son numéro de processeur, à cet effet on considère avoir à disposition un registre `p` permettant de récupérer son numéro de processeur. On peut donc calculer `q = (1-p)`, le numéro de l'autre processeur.

```
int tour = 0;
debut_sc() {
    alpha:  if (p != tour)
                goto alpha;
}

fin_sc() {
    tour = q
}
```

L'exclusion mutuelle fonctionne bel et bien, mais si l'un des deux processus souhaite exécuter la section critique mais que l'autre ne l'exécute pas, le premier sera sujet à de la **famine** : il attendra pour toujours.

2.3 Tableaux

On continue à utiliser le registre `p`. Cette fois-ci chaque CPU peut annoncer sa demande avant la boucle.

```
bool acces[2] = { false }
debut_sc() {
    acces[p] = true
    alpha:  if (acces[q])
                goto alpha;
}

fin_sc() {
    acces[p] = false
}
```

L'exclusion mutuelle est vérifiée, mais il y a un risque de boucle infinie en cas d'exécution simultanée : c'est un **dead-lock** (ou **interblocage**).

2.4 Tableaux 2 : electric boogaloo

On garde le principe de la solution précédente avec un renoncement à une demande et en recommençant.

```
bool acces[2] = { false, false }
debut_sc() {
    alpha:  acces[p] = true;
```

```
            if (acces[q])
                    acces[p] = false;

                goto alpha;
}

fin_sc() {
    acces[p] = false
}
```

On observe un interblocage si l'exécution est synchrone, qu'on peut donc résoudre ainsi :

```
bool acces[2] = { false, false }
debut_sc() {
    alpha:  acces[p] = true;
            if (acces[q])
                    acces[p] = false;
                    sleep(random(delta)) // avec "recursive doubling" : 1 2 4 8 ...
                goto alpha;
}

fin_sc() {
    acces[p] = false
}
```

On peut initialiser la graine de la fonction random par le numéro du CPU (par exemple).

2.5 Tour + tableaux + renoncement

Celui dont ça n'est pas le tour renonce.

```
bool acces[2] = { false }
int tour = 0;
debut_sc() {
    alpha:  acces[p] = true;
            if (access[q])
                if (tour != p)
                    acces[p] = false;
                    goto alpha;
}

fin_sc() {
```

```
    tour = q;
    acces[p] = false
}
```

Il n'y a pas d'exclusion mutuelle cas la condition `access[q]` est fausse, donc on ne boucle pas.

2.6 Algorithme de Dekker

```
bool acces[2] = { false }
int tour = 0;
debut_sc() {
    alpha:  acces[p] = true;
            if (access[q])
                if (tour == p) {
                    beta:   if (acces[q])
                                goto beta;
                }
                else
                {
                    acces[p] = false;
                    gamma:  if (tour == q)
                                goto gamma;
                            goto alpha
                }
}

fin_sc() {
    tour = q;
    acces[p] = false
}
```

On ne demande pas son accès tant que ça n'est pas son tour.

2.7 Algorithme de Petersen

15 ans plus tard, Petersen propose cette solution bien plus simple.

```
bool acces[2];
int dernier;
debut_sc() {
    acces[p] = true;
    dernier = p;
```

```
        alpha:  if (acces[q])
                    if (dernier == p)
                        goto alpha;
    }

    fin_sc() {
        acces[p] = false;
    }
```

On se sert de la variable `dernier` pour savoir lequel attend (qui est donc le dernier à avoir écrit dans la variable).