

Systemes d'exploitation

William SCHMITT & Othmane AJDOR

2018-2019

1 TD1 - Dekker

Avec deux processeurs :

```
debut_SCC();  
    load $i, %r0  
    inc %r0  
    store %r0, $i  
fin_SCC();
```

Après l'exécution des instructions, $i=1|2$.
Ce qu'on veut avoir c'est obtenir $i=2$ et pas $i=1$

1.1 Zero

En utilisant cette fonction :

```
bool occ=false; //occupe  
debut_SCC(){  
    alpha:  
        if (occ)  
            goto alpha;  
    occ = true;  
}  
  
fin_SCC(){  
    occ = false;;  
}
```

On n'obtient pas d'exclusion mutuelle.

1.2 One

```
bool occ = false;
\\ecrire avant le test

debut_SCC(){
    int old = occ; // local var
    occ = true;
    alpha : if(old)
                old = occ;
            goto alpha;
}

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

Pas d'exclusion mutuelle.

1.3 Two

```
int tour = 0;
p; // num de processeur {0;1}
q = 1 - p; // num de l'autre processeur

debut_SCC(){
    alpha:
        if (tour != p){
            goto alpha;
        }
}

finSCC(){
    tour = 1 - p;
}
```

Exclusion mutuelle OK, par contre, le bout de code cause un problème de famine.
L'autre processeur ne pourra jamais executer la section critique.

1.4 Three

```
bool access[2] = {false};  
//Accede a acces[] avec p/q <-annoncer ma demande avant la boucle  
  
debut_SCC(){  
    acces[p] = true;  
    alpha:  
        if (acces[q])  
            goto alpha;  
}  
  
finSCC(){  
    acces[p] = false;  
}
```

Exclusion mutuelle OK.

Si les deux processeurs executent le code en même temps, ils rentrent dans une boucle infinie.

Probleme de **dead_lock**, les deux processeurs ne font plus rien.

1.5 Four

```
bool access[2] = {false, false};  
// Three et renoncer a sa demande  
// On recommence tout  
  
debut_SCC(){  
    alpha:  
        acces[p] = true;  
        if (acces[q])  
            acces[p] = false;  
            // en reseau recursive doubling  
            sleep(random);  
            goto alpha;  
}  
  
finSCC(){  
    acces[p] = false;  
}
```

Probleme de **dead_lock** si synchrone Il faut qu'ils arrivent en même temps et qu'ils restent synchrones pour que ça marche.

1.6 Five

```
bool access[2] = {false, false};
int tour;
// Comme la Four
// Celui dont ce n'est pas le tour renonce

debut_SCC(){
    alpha:
        acces[p] = true;
        if (p != tour && acces [q])
            acces[p] = false;
            sleep(random);
            goto alpha;
}

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

Pas d'exclusion mutuelle

1.7 Six

```
bool access[2] = {false, false};
int tour;
// Comme la Four
// Celui dont ce n'est pas le tour renonce

debut_SCC(){
    acces[p] = true;

    alpha:
        if(acces[q]){
            if(tour==p){
                beta:
                    if(access[q])
                        goto beta;
            }
            else {
                acces[p] = false
                sigma: if(tour==q):
                    goto sigma;
                goto alpha;
            }
        }
    }

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

Avant de commencer, je verifie que je suis dans le bon etat.
Meme si c'est mon tour, je vais attendre que l'autre renonce à acces.
L'autre cas c'est si je suis en conflit, je renonce à mon tour.
Une fois qu'il a fini, le tour devriendra le miens

1.8 Seven - La solution correcte de Peterson

La solution correcte :

```
bool access[2] = {false, false};
int dernier;

debut_SCC(){
    acces[p] = true;
    dernier = p;
    alpha:
        if (acces[q]){
            if(dernier ==p)
                goto alpha;
        }

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