

# Sections critiques

C'est bien beau la synchronisation sauf que la coopération entre plusieurs processus pose également des problèmes si deux processus concurrents souhaitent modifier les mêmes données au même moment.

## Définition section critique

On peut donc mettre en place une **section critique**, c'est un ensemble d'instructions qui devraient être exécutées du début à la fin sans interruption.

Une section critique est indispensable lorsque l'on traite des données partagées afin qu'elle soit protégée et que ces données partagées ne deviennent pas incohérentes.

Par exemple, si on fait par exemple une liste chaînée, elle risque de ne plus être cohérente après plusieurs modifications.

On ne peut cependant pas empêcher la concurrence entre les processus. Pour cela on va mettre en place des protections avant toute modification pour s'assurer qu'un autre processus n'est pas déjà en train de modifier la zone partagée.

## Variable partagée

Celle-ci consiste à partager une variable entre plusieurs processus, qui est initialement définie à 0. Avant d'entrer dans le processus, on boucle sur la valeur de cette variable.

Si la variable est différente de 0 on boucle (et on attend). Ensuite on place la variable à 1 avant de commencer la section critique puis on la remet à 0 une fois que cela est fini.

```
while (i != 0);  
i = 1;  
/* Section critique ici */  
i = 0;
```

## Problème

Un gros problème peut survenir si un processus revient dans l'état ready (par exemple avec la fin de son quantum de temps) entre l'instruction `while` et l'instruction de `i = 1`.

Ainsi l'autre processus peut lui aussi entrer en section critique et peut lui aussi avoir son quantum de temps qui expire durant celui-ci.

Ainsi on peut donc arriver dans une situation où plusieurs processus sont dans une section critique en même temps (ce qui est justement la chose à éviter).

Ainsi, cette méthode de protection n'est pas fiable.

En plus de cela, utiliser une boucle while comme ceci consomme inutilement du CPU.

Pour plus d'information voir la vidéo de la [séance 3 du cours d'OS 2020](#) à 2:25:50.

## Par alternance

La protection par alternance consiste de manière similaire à la méthode précédente à avoir une variable partagée mais où chaque processus attend une valeur différente.

Ainsi, par exemple un programme 1 pourrait avoir le code suivant :

```
while (tour != 0);  
/* Section critique ici */  
tour = 1;
```

Et un programme 2 pourrait avoir le code suivant :

```
while (tour != 1);  
/* Section critique ici */  
tour = 0;
```

Ainsi lorsque tour est à 0, le programme 1 peut exécuter sa section critique, une fois qu'elle a fini le programme 2 peut exécuter la sienne, et une fois que le programme 2 a fini, le programme 1 peut recommencer.

## Problèmes

Cette méthode de protection est fiable, contrairement à la précédente. Cependant elle souffre tout de même d'assez gros problèmes...

Premièrement, elle est assez difficile à gérer, surtout si il y a plus de deux processus à synchroniser.

Et deuxièmement, comme la précédente, elle est assez peu efficace car utiliser une boucle while ainsi consomme inutilement du CPU.

Pour plus d'information voir la vidéo de la [séance 3 du cours d'OS 2020](#) à 2:33:20.

## Par fichier

La protection par fichier consiste à ouvrir et créer un fichier (appelé "lock file") en mode exclusif (c'est à dire qu'un seul processus peut accéder au fichier à la fois) pour annoncer que la section critique commence.

Puis enfin à supprimer le fichier une fois que la section critique est terminée.

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define FIN_SECTION_CRITIQUE 1
#define DEBUT_SECTION_CRITIQUE -1

int quid(int op, char* nom, int essais) {
    int i;

    /*
     * Quand on débute la section critique, on crée un fichier dit "lock file" en mode exclusif,
     * si cela n'est pas possible c'est qu'une section critique est déjà en cours
     */
    if(op == DEBUT_SECTION_CRITIQUE) {
        for(i=0; i<essais; ++i) {
            /* Tenter d'écrire un fichier en mode exclusif (un seul processus a accès au fichier à la fois) et renvoyer 0 en
            cas de succès */
            if(open(nom, O_WRONLY|O_CREAT|O_EXCL) >= 0) {
                return 0;
            }

            /* Si cela n'a pas fonctionné, réessayer dans une seconde */
            else if(i<essais) {
                sleep(1);
            }
        }
    }
}
```

```

/*
 * A la fin d'une section critique on supprime le lock file
 */
if(op == FIN_SECTION_CRITIQUE) {
    /* Suppression du fichier et retourne 0 en cas de succès */
    if(unlink(nom) == 0) {
        return 0;
    }
}

/* Retourne -1 en cas d'erreur ou dans le cas où tous les essais ont échoués */
return -1;
}

int main(void) {
    printf("Attente section critique\n");
    quid(DEBUT_SECTION_CRITIQUE, "program.lock", 5);

    /* Section critique ici */
    printf("Début section critique\n");
    sleep(5);

    printf("Fin section critique\n");
    quid(FIN_SECTION_CRITIQUE, "program.lock", 5);

    return EXIT_SUCCESS;
}

```

## Problèmes

Cette solution est tout à fait fonctionnelle et fiable, cependant le fait de devoir gérer un fichier peut rendre les choses un peu compliquée, de plus cela ralentit les choses. Car pour chaque accès au fichier, le processus devra passer en état **waiting**, puis **ready**, puis de nouveau **running**.

Pour plus d'information voir la vidéo de la [séance 3 du cours d'OS 2020](#) à 2:37:50.

## Synchronisation hardware

La synchronisation hardware consiste à utiliser des instructions assembleurs pour protéger une section critique.

Voici un pseudo-code de démonstration :

```
boolean TestAndSet (boolean target) {  
    /* On copie la valeur de target */  
    boolean rv = target;  
  
    /* On met target à true */  
    target = true;  
  
    /* On retourne la copie de la valeur initiale */  
    return rv;  
}
```

Ainsi pour l'utiliser il suffirait de faire ceci :

```
/* On attends que le lock (variable partagée initialement à false) soit mis à false pour continuer */  
while (TestAndSet(lock));  
  
/* Section critique ici */  
  
/* On met le lock à false une fois terminé */  
lock = false;
```

Ainsi lorsque lock est à false, TestAndSet va la mettre à true et retourner false ce qui va donc faire passer la boucle et entrer en section critique. Une fois cette dernière terminée, le lock retourne à false.

En revanche si lock est à true, TestAndSet va retourner true et par conséquent rester dans le while, en attente jusqu'à ce que la variable soit à false.

## Problèmes

Cette méthode est fiable mais le problème avec celle-ci c'est l'utilisation du `while` qui va une fois de plus consommer du CPU pour simplement attendre.

Il est tout de même bon de noter que cette méthode est utilisée par le système d'exploitation pour gérer d'autres systèmes de protection tels que les sémaphores.



Pour plus d'information voir la vidéo de la [séance 3 du cours d'OS 2020](#) à 2:47:00.

# Sémaphore

Les [sémaphores](#) permettent de très simplement protéger une section critique, voici un exemple :

```
#include "semadd.h"
#include "sys/sem.h"
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

#define KEY_SEM1 12345
#define KEY_SEM2 12346

int main(void) {

    int sem1, sem2;

    /* On crée 2 sémaphores */
    sem1 = sem_transf(KEY_SEM1);
    sem2 = sem_transf(KEY_SEM2);

    /* On crée un nouveau processus */
    switch (fork()) {
        case -1:
            printf("Quelque chose s'est mal passé lors de la création du processus...\n");
            return EXIT_FAILURE;

        /* Pour le fils */
        case 0:
            /* Attente du père */
            printf("En attente du père\n");
            p(sem1);

            /* Section critique */
            printf("Section critique du fils commence\n");
```

```

sleep(3);

/* Annonce au père qu'il a fini */
printf("Section critique du fils se termine\n");
v(sem2);

break;

/* Pour le père */
default:
/* Section critique */
printf("Début de la section critique du père\n");
sleep(3);

/* Annonce au fils qu'il a fini */
printf("Fin de la section critique du père\n");
v(sem1);

/* Attends le fils avant de supprimer les sémaphores */
p(sem2);
semctl(sem1, IPC_RMID, 0);
semctl(sem2, IPC_RMID, 0);
}

return EXIT_SUCCESS;
}

```

Comme vu précédemment, les p et v des sémaphores sont des actions unitaires, il n'y a donc pas de risque que le processus soit arrêté au milieu. L'utilisation des sémaphores est la manière recommandée de gérer des sections critiques.

Pour plus d'information voir la vidéo de la [séance 3 du cours d'OS 2020](#) à 2:52:00.