



# PCO - Labo5 Gestion de ressources par moniteurs

Auteurs: Alexandre Jaquier et Valentin Kaelin

## Description des fonctionnalités du logiciel

Pour ce laboratoire, nous travaillons sur un programme QT modélisant une place de jeux munie d'un toboggan. Plusieurs marches sont nécessaires afin d'accéder audit toboggan et chacune d'elle peut contenir un nombre maximum d'enfants. De plus, un nombre défini d'enfants est nécessaire sur la place de jeux avant que ceux-ci ne puissent accéder au toboggan. Lorsque ce nombre est atteint, la moitié des enfants présent sur la place peut accéder au toboggan.

Chacun enfant est modélisé par un thread. Le système de marches est géré grâce à un moniteur de Mesa et celui de la place de jeu via un moniteur de Hoare.

Une deuxième version du programme est également réalisée. Celle-ci ajoute des méthodes à appeler avant et après l'attente de chaque enfant. Ces méthodes étant gourmandes, on souhaite les appeler en dehors de l'exclusion mutuelle.

## Choix d'implémentation

### Partie A

#### Place

Nous n'avons pas fait de réels choix lors de l'implémentation de cette partie, nous nous sommes contenté d'implémenter le moniteur de Mesa comme nous l'avions vu en cours.

On pourrait commenter le fait que la condition d'accès est mise dans une boucle. En effet ne pas la mettre ne poserait normalement pas de problème dans notre cas, car lorsqu'un enfant quitte la place, il notifie un seul autre enfant que la place est libre. Malgré cela, nous avons vu en cours qu'ajouter la boucle était une bonne pratique, dans le cas où deux enfants en attente viendraient à être notifiés en même temps, un seul passera.

De plus, on vérifie dans la fonction `leave()` que le nombre d'enfants est supérieur à 0 afin d'éviter un overflow à la valeur max de l'`unsigned` (dans le cas étrange où la méthode `leave()` serait appelée sans `access()` avant).

```

// dans access()
while (nbCurrentPeople == nbMaxPeople) {
    cond.wait(&mutex);
}

// dans leave()
if (nbCurrentPeople > 0) {
    cond.notifyOne();
    nbCurrentPeople--;
}

```

## Playground

Afin de simplifier notre code, nous avons opté pour une solution à base d'une boucle. En effet, lorsque le nombre d'enfants souhaité est atteint, le dernier enfant s'occupe de libérer le bon nombre d'enfants.

```

void play()
{
    monitorIn();
    nbKidsWaiting++;
    if (nbKidsWaiting == nbKidsToLeave) {
        nbKidsWaiting -= nbKidsToFree;
        // On libère nbKidsToFree enfants
        for (unsigned i = 0; i < nbKidsToFree; ++i)
            signal(cond);
    }
    wait(cond);
    monitorOut();
}

```

La variable `nbKidsToFree` est simplement initialisée au nombre souhaité pour libérer les enfants divisé par deux (`nbKidsToLeave / 2`).

## Partie B

Dans le deuxième partie du laboratoire, il fallait utiliser deux nouvelles méthodes: `startWaiting()` ainsi que `endWaiting()` qui se trouvent dans la classe `Kid`. Ces fonctions doivent être considérées comme potentiellement longue à l'exécution.

Afin de laisser la main aux autres enfants lors de l'exécution de ces méthodes, nous devons les appeler en dehors de l'exclusion mutuelle, donc après `mutex.unlock()` ou `monitorOut()`.

### Place

```

void access(Kid &kid)
{
    bool hasAlreadyWait = false;
    mutex.lock();
    while (nbCurrentPeople == nbMaxPeople) {
        if (!hasAlreadyWait) {
            mutex.unlock();
            kid.startWaiting();
            mutex.lock();
        }
        if (nbCurrentPeople == nbMaxPeople) {
            cond.wait(&mutex);
        }
        hasAlreadyWait = true;
    }

    if (hasAlreadyWait) {
        mutex.unlock();
        kid.endWaiting();
        mutex.lock();
    }
    nbCurrentPeople++;
}

```

```

        mutex.unlock();
}

```

Dans la méthode `access` si nous utilisons la même base de code que la partie A. Un nouveau soucis apparaît à cause de la méthode `startWaiting()`. En effet, si le temps d'exécution prend beaucoup de temps, un autre thread aura pu modifier la valeur de la variable `nbCurrentPeople`, ce qui fausserait la condition testée précédemment.

Afin de résoudre ce problème nous avons choisis de simplement re-tester la condition indiquant si l'enfant a accès à la place avant d'effectuer le `wait()`.

Afin de n'appeler qu'une seule fois en cas d'attente les deux méthodes coûteuses de `Kid`, nous utilisons un boolean, `hasAlreadyWait`.

## Playground

```

void play(Kid &kid)
{
    monitorIn();
    nbKidsWaiting++;
    if (nbKidsWaiting == nbKidsToLeave) {
        nbKidsWaiting -= nbKidsToFree;
        nbKidsReleased = nbKidsToFree;
        for (unsigned i = 0; i < nbKidsToFree; ++i)
            signal(cond);
    }
    monitorOut();
    kid.startWaiting();
    monitorIn();
    if (nbKidsReleased == 0) {
        wait(cond);
    }
    nbKidsReleased--;
    monitorOut();
    kid.endWaiting();
}

```

Le problème lors de l'appel des nouvelles méthodes de `Kid` est qu'un enfant peut être signalé avant d'avoir attendu sur le `wait()`. Il se bloque donc en attendant un signal qui est déjà passé.

Afin de résoudre ce problème, il faut faire en sorte de voir, après l'exécution du `startWaiting()` et avant de faire le `wait()` si l'enfant peut directement s'en aller ou s'il doit attendre. En d'autres termes, si tous les signaux ont été émis et reçus ou s'il reste des signaux en attente. C'est à cet effet qu'est utilisée la variable `nbKidsReleased`.

Nous avons également pensé à créer un thread afin de gérer la fonction `startWaiting()`. Comme ceci, un enfant ne peut pas rater de signal et il n'y aurait donc pas besoin de libérer la section partagée afin d'exécuter la fonction.

La fonction `play` ressemblerait donc à ceci :

```

void play(Kid &kid)
{
    monitorIn();
    nbKidsWaiting++;
    if (nbKidsWaiting == nbKidsToLeave) {
        for (unsigned i = 0; i < nbKidsToFree; ++i)
            signal(cond);
    }
    PcoThread thread(&Kid::startWaiting,&kid);
    kid.startWaiting();
    wait(cond);
    thread.join();
    monitorOut();
    kid.endWaiting();
}

```

Mais la création de thread étant plus lourde qu'une simple variable à incrémenter et vérifier nous avons décidé de ne pas utiliser cette version-ci.

## Tests Effectués

Afin de tester notre programme, nous avons baissé le nombre d'enfants dans le `main.cpp` à **6** afin de pouvoir mieux visualiser l'exécution. Ceci nous permettait de voir que les enfants étaient bien libérés lorsque **3** attendaient et qu'**un** seul était libéré à la fois.

### Partie A

Description du test	Résultat attendu et observé
Nombre d'enfants max par marche	Les marches paires acceptent au maximum un enfant et les impaires au maximum deux
Attente avant accès au toboggan	Les enfants attendent le nombre d'enfants souhaité avant de pouvoir accéder à nouveau au toboggan
Libération des enfants	Une fois le nombre d'enfants atteint, seule la moitié des enfants attendant sont libérés

### Partie B

La deuxième partie du laboratoire a été plus compliquée à tester. En effet, les fonctions `startWaiting()` et `endWaiting()` ne sont pas réellement coûteuses. Nous avons donc ajouté dans ces méthodes la ligne `PcoThread::usleep(3000000);` afin de simuler un temps d'exécution de trois secondes pour avoir une meilleure idée du comportement.

Tout d'abord, les tests effectués dans la partie A ont également été testés dans cette partie B. Nous nous contenterons donc de lister les tests supplémentaires réalisés.

Description du test	Résultat attendu et observé
Les enfants sont bien libérés afin d'accéder au toboggan	Même si un enfant se trouve dans l'exécution de la fonction <code>startWaiting()</code> par exemple, elle reçoit par la suite l'information qu'elle a été libérée
Les enfants sont bien libérés afin d'accéder à la marche suivante	Après l'exécution de la fonction <code>startWaiting()</code> , l'enfant attend en prenant compte du nouvel état de la marche et non pas l'état avant l'appel de la fonction

## Conclusion

Notre code a l'air assez fonctionnel. Malgré tout, il est tout de même assez compliqué de penser et tester tous les cas limites. L'utilisation du sleep nous a bien aidé à en trouver quelques uns.

Une version avec sémaphore aurait peut-être été plus simple pour la deuxième partie car cela nous aurait permis de ne pas sortir du mutex/monitor plusieurs fois entre les appels aux méthodes coûteuses.