Rapport projet Systèmes concurrents

Léa Houot
2SNR2

1. Implantation exclusion mutuelle

Pour ce qui est de l'exclusion mutuelle je l'ai initialiser dans `sys_tube()`. Pour `tubeOuvrir(...)`, `tubeFermer(...)`, `tubeEcrire(...)` et `tubeLire(...)` nous rentrons dans l'exclusion mutuelle afin qu'aucune valeur ne change pendant que la tâche est en cours d'exécution et qui pourrais entraîner des problèmes. Pour `tubeOuvrir(...)` et `tubeFermer(...)` c'est pour éviter qu'une autre tâche ne change `nbLecteurs` et `nbEcrivains` .Et, pour `tubeEcrire(...)` et `tubeLire(...)` c'est pour que le contenu du tube ne change.

2. Implantation condition

Pour les conditions j'avais déjà pris le code en main. J'ai choisi comme condition `tube_non_plein` pour que les écrivains puissent écrire et `tube_non_vide` pour que les lecteurs puissent lire. Il a donc fallut définir `tubeVide()` et `tubePlein()`. Pour `tubePlein()` il a été assez facile de le définir mais pour `tubeVide()` cela s'est avéré plus compliqué. En effet,j'ai eu du mal à comprendre que, comme les écrivains ont la priorité il se peut qu'il reste des lecteurs mais pas d'écrivain. Dans ces cas là même si le tube est vide il faut débloquer les lecteurs. Donc dans tube vide on vérifie qu'il reste tout de même des écrivain pour savoir si le tube peut se re-remplir et sinon le tube est « considéré comme vide » et les lecteurs n'attendent pas en vain. Et, lorsqu'on ferme un « tube écrivain » on vérifie qu'il en reste et si il n'en reste pas on diffuse `tube_non_vide` pour débloquer les lecteurs qui sont en attente. Donc, pour `tubeEcrire(...)` nous rentrons dans l'exclusion mutuelle puis nous vérifions que le tube n'est pas plein et si c'est le cas on attend `tube_non_plein` et on laisse la main à une autre tâche. Inversement, pour `tubeLire(...)` nous rentrons dans l'exclusion mutuelle puis nous vérifions que le tube n'est pas vide et si c'est le cas on attend `tube_non_vide` et on laisse la main à une autre tâche.

3. Implantation ordonnanceur

Finalement, j'ai essayé d'implanter un ordonnanceur pour plus d'équité. J'ai commencé par mettre la fonction `ordonnanceur()` dans `tubeEcrire(...)` et `tubeLire(...)` cependant cela n'était pas concluant, je me retrouvait toujours avec des lecteurs qui ne pouvais rien lire et les autres qui lisait plus (même si déjà ce n'était pas un seul lecteur qui lisait tout).

J'ai donc essayé avec la fonction `basculerTache()` dans `init-acces-concurrent.c`, je l'ai mis dans `lecteur()` et grâce à cela j'ai maintenant des lecteurs qui lisent à peu près le même nombre d'octets.

4. Est-ce-que ça marche ?

Afin de vérifier si cela fonctionnait bien j'ai implémenter un compteur pour les écrivains dans `ecrivain()` dans `init-acces-concurrent.c`, de cette manière ils écrivent tous le même nombre d'octets. Et donc avec cela j'ai pu voir que mes lecteurs lisaient tous les octets écrits par les écrivains. Également, j'ai pu observer l'état des variables condition et voir qu'il y a eu une diffusion de `tube_non_vide` lorsqu'il n'y avait plus d'écrivain.

En enlevant le compteur j'ai mes écrivains et mes lecteurs qui s'alternent, de cette manière il y a toujours de la place pour que les écrivains écrivent et toujours de quoi lire pour les lecteurs.

L'exécution ne se stoppe pas. Et, en regardant l'état de l'exclusion mutuelle je vois qu'il n'y a aucune attente.

5. Conclusion

Finalement, j'ai réussi a avoir mes lecteurs et mes écrivains qui s'alternent, tous les écrivains écrivent et tous les lecteurs lisent. J'ai pu vérifier que c'était cohérent, il y a autant d'octets écrits que d'octets lus et que ce lisent les lecteurs est cohérent avec ce qu'écrivent les écrivains.

J'ai fais la version non-bloquante, en effet lorsqu'un écrivain veux écrire il écrit autant qu'il peut. C'est-à-dire que si notre écrivain veut écrire 20 octets, si il reste une place de 15 octets dans le tube alors l'écrivain écrit 15 octets, dans la version bloquante il n'écrirait pas.

Afin de faire la version bloquante il faut modifier `tubeEcrire()` de cette façon :

```c
size_t tubeEcrire(Fichier * f, void * buffer, size_t nbOctets){
    ...
    if (MANUX_TUBE_CAPACITE - tube->taille > nbOctets && MANUX_TUBE_CAPACITE - tube->indiceProchain > nbOctets) {
        n = nbOctets;

        // On peut donc copier n octets dans le buffer à partir de la
        // position courante, sans risque de déborder
        memcpy(tube->donnees + tube->indiceProchain, buffer, n);

        tube->indiceProchain = (tube->indiceProchain + n) % MANUX_TUBE_CAPACITE;
        tube->taille += n;

        buffer += n;

        nbOctetsEcrits += n;
    }
    ...
    return nbOctetsEcrits;
}
```

De cette manière l'écrivain écrit seulement si il reste au minimum n octets, n étant la taille de ce qu'il veut écrire.