Linux 2 : Appels Systèmes - BINV2181 (tp06- IPCs)

Avant de résoudre ces exercices, nous vous conseillons de lire le document suivant qui présente une vue d'ensemble sur les sémaphores et la mémoire partagée, et qui résume également les principaux appels systèmes liés à ces matières : "Les ressources IPC : sémaphore, mémoire partagée et files de messages"[1]

Après avoir exécuté vos programmes, il est possible que certains "ipcs" non-utilisés soient réservés sur la machine sur laquelle tournait vos programmes. Pensez à utiliser les commandes `ipcs` et `ipcrm` pour supprimer ces "ipcs".

1. Le radar de recul (mémoire partagée)

Un radar de recul est un système permettant au conducteur d'estimer la distance séparant son véhicule d'un obstacle[2]. Nous vous demandons d'écrire :

- un programme "**radar**" dont le but est de simuler un radar de recul; et
- un programme "**dashboard**" dont le but est de simuler un tableau de bord qui affiche les informations transmises par le radar de recul.

Pratiquement, votre programme radar écrira toutes les 3 secondes, pendant 1 minute, un nombre entier aléatoire dans une mémoire partagée (utilisez la fonction *randomIntBetween()* du module *utils*). Votre programme dashboard affichera toutes les 5 secondes, pendant 1 minute, l'entier se trouvant dans la mémoire partagée.

Nul besoin de signaux. Utilisez simplement des *sleep* pour écrire ces deux programmes.

2. Mon identité (section critique)[3]

Dans un premier temps, nous vous demandons d'écrire un programme "**famille1**" qui crée 2 processus fils. Chaque fils affiche 3 fois le message « je suis le fils <pid> », chaque affichage étant séparé par 1 seconde d'attente. Les fils pourraient, par exemple, afficher le texte ci-dessous.

```
je suis le fils 123
je suis le fils 124
je suis le fils 123
je suis le fils 124
je suis le fils 123
je suis le fils 124
```

[1] M. Bagein, S. Frémal, S. Mahmoudi et P. Manneback, *Les ressources IPC : sémaphore, mémoire partagée et files de messages*, Travaux pratique d'Informatique Temps Réel, UMONS/Polytech, TP 3, Année Académique 201415

[2] https://fr.wikipedia.org/wiki/Radar_de_recul, 15/03/2018

[3] M. Bagein, S. Frémal, S. Mahmoudi et P. Manneback, *Ibid*

Dans un deuxième temps, nous souhaitons empêcher l'affichage simultané des deux fils, en considérant l'accès à *stdout* comme une ressource critique. Pour ce faire, nous vous demandons d'écrire un programme "**famille2**" qui est une variante du programme "famille1". Lorsqu'un fils démarre son exécution, il réserve l'accès à *stdout* et réalise l'affichage de ses 3 messages. Ensuite, il "libère" l'accès *stdout*. Le deuxième fils peut alors réaliser son affichage. Par exemple, les deux fils pourraient afficher le texte ci-dessous.

```
je suis le fils 123
je suis le fils 123
je suis le fils 123
je suis le fils 124
je suis le fils 124
je suis le fils 124
```

3. Le distributeur de tickets (mémoire partagée et section critique)

Lorsqu'une personne habitant Louvain-la-Neuve se rend à la maison communale, elle reçoit un ticket portant un numéro et va s'asseoir dans la salle d'attente. Quelques instants plus tard, le numéro de son ticket s'affiche sur un grand écran. La personne se rend alors au guichet où se trouve un employé communal.

Nous vous demandons d'écrire un programme "distributeur" et un programme "ecran".

- Votre programme "**distributeur**" ne fait qu'une chose : il imprime sur *stdout* le numéro du prochain ticket (qui est simplement incrémenté de 1) et se termine. La numérotation des tickets commence à 1.
- Votre programme "**ecran**" ne fait également qu'une chose : simulant le fait qu'une place s'est libérée au guichet, il affiche le numéro de la personne suivante qui peut se présenter au guichet sur le grand écran. Ensuite, le programme se termine. S'il existe un prochain numéro (i.e. un ticket avec ce numéro a été distribué), "ecran" l'imprime sur *stdout*, sinon il affiche le message "`Il n'y a plus personne!`".

Les deux programmes modifient la mémoire partagée de manière adéquate. Le programme "distributeur" s'exécute entièrement chaque fois qu'un nouvel arrivant déclenche une demande de tickets. De même, votre programme "ecran" s'exécute entièrement chaque fois qu'une place se libère au guichet.

Notez que plusieurs programmes "distributeur" et plusieurs programmes "ecran" peuvent s'exécuter simultanément. Vous devez donc protéger vos accès à la mémoire partagée à l'aide de sémaphores.

En plus des deux programmes, nous vous conseillons d'écrire un troisième programme "**admin1**" dont le but

- est de créer et d'initialiser (si l'option -c est activée) la mémoire partagée et les sémaphores de manière indépendante (pour se prémunir contre le risque de *race condition*) ; et
- est de détruite (si l'option -d est activée) la mémoire partagée et les sémaphores de manière indépendante.

Par exemple, ce programme sera exécuté une fois avant les programmes "distributeur" et "ecran".

4. Synchronisation de processus[4]

Nous avons vu lors des exercices précédents l'une des deux utilisations spécifiques des sémaphores : la *gestion d'accès concurrents à une ressource critique*. Dans cet exercice, nous abordons la seconde manière d'utiliser les sémaphores : la *synchronisation dans le temps des actions effectuées par deux processus*.

Nous vous demandons d'écrire :

- Un programme "**client**" qui prend en paramètre une chaîne de 10 caractères minuscules. Il écrit ces 10 caractères dans une mémoire partagée.
- Un programme "**serveur**" qui traduit le texte de la mémoire partagée en majuscules. Lorsque cette traduction est terminée, le programme "client" affiche le texte modifié sur *stdin*.

Les programmes client et serveur doivent se synchroniser à l'aide de sémaphores.

Nous vous conseillons de créer et initialiser la mémoire partagée et les sémaphores dans un programme "**admin2**" similaire au programme "**admin1**" de l'exercice 4.

[4] M. Bagein, S. Frémal, S. Mahmoudi et P. Manneback, *Ibid.*

5. Appels systèmes relatifs aux signaux, à la mémoire partagée et aux sémaphores [examen de sept. 2019]

Cette question se focalise sur un simulateur d'un aiguillage de train (cf Figure 1 Aiguillage).

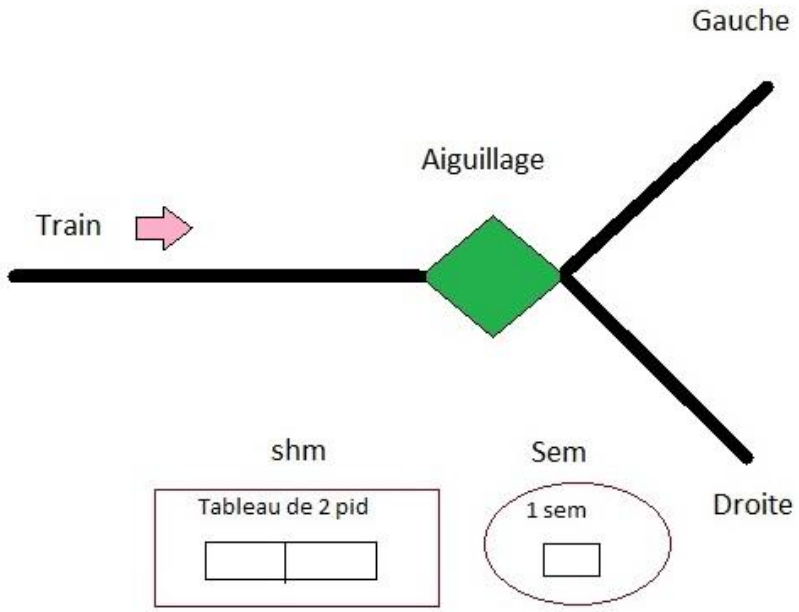

Figure 1 Aiguillage

Ce simulateur est un système distribué qui exécute une unique instance du programme **switch** simulant un aiguillage et des instances du programme **train** souhaitant utiliser l'aiguillage. Ce simulateur manipule un sémaphore et de la mémoire partagée de la manière suivante :

- Le sémaphore est utilisé pour la **réservation** de l'aiguillage par un train.
- La mémoire partagée est constituée d'un tableau de deux **pid** :
  - le pid de l'aiguillage (indice 0 du tableau) ;
  - le pid du train qui a réservé l'aiguillage (indice 1 du tableau).

Lorsqu'un train souhaite utiliser l'aiguillage, **le train** applique le protocole suivant :

1) Le train réserve l'aiguillage.
2) Le train écrit son pid dans la mémoire partagée.
3) Si le train souhaite aller à gauche, il envoie un signal SIGUSR1 à l'aiguillage. Sinon, s'il souhaite aller à droite, il envoie un signal SIGUSR2 à l'aiguillage.
4) Le train attend le signal SIGUSR1 de l'aiguillage qui lui indiquera que l'aiguillage est placé dans la bonne direction et qu'il peut être traversé.
5) Le train traverse l'aiguillage pendant 10 secondes (utilisez *sleep*).
6) Le train libère l'aiguillage.

L'aiguillage exécute une boucle qui traite les demandes des trains tant qu'il est actif. Lorsqu'un train souhaite utiliser **l'aiguillage**, il applique le protocole suivant :

1) Lorsque l'aiguillage reçoit la demande de se positionner à gauche ou à droite et qu'il est déjà en bonne position, l'aiguillage envoie directement un signal SIGUSR1 au train.
2) Lorsque l'aiguillage reçoit la demande de se positionner à gauche ou à droite et qu'il n'est pas en bonne position, l'aiguillage se positionne dans la bonne direction. Pour ce faire, il attend 5 secondes (utilisez *sleep*). Ensuite, il envoie un signal SIGUSR1 au train.

Lorsque l'aiguillage reçoit un signal SIGINT, il n'est plus actif et se termine. Notez que l'on considère qu'à tout moment il n'y a qu'un seul programme *switch* qui s'exécute. Vous ne devez pas le vérifier. En revanche, un train ne peut pas passer s'il n'y a aucun programme *switch* qui s'exécute. Notez également que lorsque l'aiguillage est initialisé, il se positionne à gauche.

Nous vous demandons de compléter les 5 fichiers suivants :

1. Un **makefile**

2. **ipc_conf.h** : fichier de configuration contenant toutes définitions de constantes liées aux IPC, afin d'éviter que ces définitions soient copiées dans chacun des programmes qui les utilise.

3. **admin3.c** : permet de gérer les IPC (mémoire partagée et sémaphore)

   - `./admin3 -c` : crée les IPC qui sont utilisés par les programmes "distributeur" et "écran". Si les IPC existent déjà, le programme affiche le message « IPCs already created ».

- `./admin3 -d` : détruit les IPC créés avec l'option -c. Si les IPC n'existent pas, le programme affiche le message « IPCs not existing ».

Si *admin3* est exécuté sans argument ou avec des arguments ne correspondant pas à -c ou -d, le programme se termine en affichant comment utiliser la commande.

4. **switch.c :** permet de générer un programme qui simule l'aiguillage. Ci-dessous, vous trouverez le résultat de l'exécution du programme *switch* lorsqu'un premier train souhaite aller à gauche, ensuite un deuxième train souhaite aller à droite et enfin le programme est arrêté par un SIGINT. Utilisez la fonction *getTime()* pour afficher la date et l'heure (cf. module *utils*).

```
$$$ ./create_ipc
$$$ ./switch
[Fri Mar 19 11:53:05 2021] SWITCH UP ON THE LEFT SIDE
[Fri Mar 19 11:53:11 2021] A TRAIN WANTS TO GO LEFT
[Fri Mar 19 11:53:11 2021] SWITCH UP ON THE LEFT SIDE
[Fri Mar 19 11:53:11 2021] SWITCH OK
[Fri Mar 19 11:53:26 2021] A TRAIN WANTS TO GO RIGHT
[Fri Mar 19 11:53:26 2021] SWITCH UP ON THE LEFT SIDE
[Fri Mar 19 11:53:26 2021] MODIFYING SWITCH SIDE
[Fri Mar 19 11:53:31 2021] SWITCH OK
^C[Fri Mar 19 11:54:19 2021] SWITCH DOWN
$$$ ./delete_ipc
$$$
```

5. **train.c** : permet de générer un programme qui simule un train. Ce programme prend un paramètre L si le train souhaite aller à gauche et R si le train souhaite aller à droite. Vérifiez qu'il y a bien un argument sur la ligne de commande. Ci-dessous, vous trouverez le résultat de la simulation d'un premier train souhaitant aller à gauche, ensuite d'un deuxième train souhaitant aller à droite. Utilisez la fonction *getTime()* pour afficher la date et l'heure (cf. module *utils*).

```
$$$ ./train L
[Fri Mar 19 11:53:11 2021] WAIT UNTIL THE SWITCH IS FREE
[Fri Mar 19 11:53:11 2021] WAIT UNTIL THE SWITCH IS ON THE CORRECT SIDE
[Fri Mar 19 11:53:11 2021] START CROSSING THE SWITCH
[Fri Mar 19 11:53:21 2021] END CROSSING THE SWITCH
$$$ ./train R
[Fri Mar 19 11:53:26 2021] WAIT UNTIL THE SWITCH IS FREE
[Fri Mar 19 11:53:26 2021] WAIT UNTIL THE SWITCH IS ON THE CORRECT SIDE
[Fri Mar 19 11:53:31 2021] START CROSSING THE SWITCH
[Fri Mar 19 11:53:41 2021] END CROSSING THE SWITCH
```