RIVAL Gaëtan
GUYOT Sacha
SAJIDE Idriss

Compte Rendu Architecture des Systèmes
TP – Sémaphores

Lien Git : https://github.com/GaetanRival/AS.git

Tout d'abord, nous avons choisi de mettre tout notre code dans le fichier /src/kernel/sem.c car il n'était pas nécessaire d'avoir un fichier pour chaque appel système, étant donné qu'ils n'appellent pas de fonctions déclarées ailleurs que dans sem.c. Donc nous avons renommé les fonctions dans sem.c, en "sys_semget", "sys_semcmtl" et "sys_semop".

Ensuite pour la première implémentation des fonctions "create","down","up" et "destroy", voici comment nous avons procédé :

Tout d'abord, nous avons longtemps réfléchi à la structure de l'**objet sémaphore** afin de la rendre la plus flexible pour la suite de la programmation. Pour cela nous retrouvons :

- une file d'attente des processus de taille 64
- la valeur du sémaphore définie à son initialisation
- une clé
- une fille d'attente de taille 1 de processus à réveiller

Pour la file d'attente des processus, nous étions d'abord parti sur l'idée d'utiliser une liste mais nous sommes tombés face à un problème. Comme nous ne pouvions pas utiliser de malloc, nous avons décidé de changer de stratégie et d'utiliser tout simplement un tableau de 64 cases. Les sémaphores créés sont stockés dans un tableau de sémaphores, de taille fixée à 64 également.

Pour placer le sémaphore dans le tableau de sémaphore, nous parcourons notre tableau jusqu'à trouver une case dont le sémaphore a une key égale à 0 car nous avons décidé qu'une key égale à 0 correspond a un sémaphore non initialisé.

Pour la fonction **create**, elle renvoie un pointeur sur le sémaphore créé pour que la fonction semget puisse plus tard régler les champs si nous passons une key ne correspondant à aucun sémaphore existant.

La procédure **down** attribue une ressource au processus qui l'appelle.
Dans le cas où le sémaphore n'a pas de ressource à accorder (si sa valeur est nulle) alors le processus s'endort. Si la valeur est non nulle, on la décrémente et le processus continue son exécution.

La procédure **up** est symétrique à la procédure down. D'abord on vérifie si la valeur du sémaphore est nulle ou non.

Si la valeur est non nulle, on incrémente.

Si elle est nulle, alors il faut réveiller un processus qui dort.

On vérifie alors qu'il y a un processus qui dort. Si oui, on place le premier processus endormi dans une file d'attente temporaire. Nous faisons cela afin de pouvoir effectuer un wakeup sur cette file temporaire. De cette manière, on ne réveille que le processus choisi. Ensuite, on reconstruit la file d'attente de processus, en plaçant les processus restant dans le bon ordre, c'est à dire, en supprimant le premier processus car nous venons de le réveiller. Dans le cas où aucun processus ne dort, on incrémente simplement la valeur du sémaphore.

Le procédure **destroy** rend NULL la file d'attente du sémaphore et définit sa key à 0, car nous avons décidé que les sémaphores avec un key égale à 0 seraient des sémaphores non utilisés.

La fonction **semget** nous a demandé de modifier la fonction create pour la rendre compatible en retournant un pointeur sur un sémaphore. Cette dernière parcourt tout simplement le tableau de sémaphore à la recherche du sémaphore en question à l'aide d'une clé passé en paramètre. Dans le cas où la clé n'est présente, nous créons un nouveau couple sémaphore/clé, puis nous retournons le nouveau indice dans le tableau de sémaphore.

Pour l'implémentation de **semctl,** nous sommes parti sur l'idée d'utiliser un switch afin de traiter les différents cas selon la valeur de cmd et donc de faire appel à l'une des fonctions en question parmis GETVAL, SETVAL et IPC_RMID.

Enfin la fonction **semop,** reste très basique car selon le signe de la valeur de op, soit nous effectuons un down ou bien un up.

Pour la partie des tests, nous avons tout d'abord exécuter le test par défaut pour vérifier si notre implémentation à propos des sémaphores était correct. Nous avons rencontré plusieurs problèmes liés au pointeur de sémaphore que nous sommes parvenu à résoudre.

Dans un second temps, nous avons décortiqué le code du test pour mieux comprendre le principe de producteur et consommateur. Ensuite nous avons développé un second test représentant une répétition de producteur et consommateur afin de s'assurer que la gestion des sémaphores sur plusieurs producteurs et consommateurs restait opérationnelle.