Rapport de Projet TSE : Implémentation des Appels Système et Primitives de Synchronisation

LAKAF Massila 22411462

RAHMOUNI Mohamed Amine 22415723

Introduction

Ce projet vise à étendre le noyau my_little_kernel.c avec des appels systèmes (mlk_signal, mlk_wait, mlk_send, mlk_recv) et des primitives de synchronisation. Des processus tests (signal.c, wait.c, send.c & recv.c) ont été développés pour valider ces fonctionnalités, qui incluent également la communication inter-processus.

1. Description et Implémentation des Primitives de Synchronisation

1.1. mlk_print : Affiche un message précédé du PID du processus appelant.

Implémentation : Défini dans le noyau, utilise le PID courant et envoie le message formaté à la sortie standard.

Exemple :
mlk_print("J'attends un signal") ;
mlk_print("J'envoie un signal");

1.2. mlk_wait : suspend un processus jusqu'à réception d'un signal, en modifiant son état et son contexte d'exécution.

Objectifs : Synchroniser des processus.
L'implémentation de mlk_wait suspend un processus en l'ajoutant à une liste d'attente (waiting_processes) et en fixant son quantum à -1 pour empêcher sa planification. Une valeur d'alarme très élevée (INT_MAX) garantit qu'il reste bloqué, tandis que son contexte d'exécution (ucontext_t) est sauvegardé. Le contrôle est ensuite transféré au planificateur via swapcontext, en attendant un signal pour le réveiller.

1.2. mlk_Signal : réveille un processus spécifique en ajustant ses paramètres pour qu'il soit replanifié.

L'implémentation de mlk_signal consiste à parcourir la liste waiting_processes pour localiser le processus correspondant au PID donné. Une fois identifié, ce processus est retiré de la liste d'attente, et son état est mis à jour. Son quantum est défini à 1, permettant ainsi au planificateur de le replanifier, et son alarme est réinitialisée à l'heure actuelle (via gettimeofday), indiquant qu'il est prêt à être exécuté.

2. Comment ça fonctionne le programme :

La séquence commence avec main dans my_little_kernel, où spawn exécute init, lançant signal.so (PID 2). Ce dernier crée wait.so (PID 3) via spawn. Signal envoie ensuite des signaux à wait avec mlk_signal(3). Réveillé par ce signal, wait affiche un message, puis retourne en attente avec mlk_wait(). Si un signal est envoyé alors qu'il n'est pas pas attendu, celui-ci est perdu.
L'ordonnanceur (scheduler) fonctionne en boucle, appelant choose_next_process pour sélectionner le prochain processus prêt à s'exécuter en fonction des alarmes et quanta. Si wait est bloqué (mlk_wait met son quantum à -1 et fixe son alarme à INT_MAX), il est ignoré jusqu'à ce qu'un signal (mlk_signal) réinitialise son quantum et son alarme, permettant sa reprise. Ce mécanisme assure une synchronisation efficace entre signal et wait, évitant les blocages indéfinis tout en respectant l'ordre d'exécution.

3. Communication Inter-Processus : mlk_send et mlk_recv

3.1 Description :

mlk_send(void *msg, int l, int p) : Envoie un message de longueur l au processus p.
mlk_recv(void *buf, int l) : Reçoit un message dans un tampon buf de longueur maximale l.

3.2 Implémentation :

L'implémentation des appels systèmes mlk_send et mlk_recv repose sur la création de deux fichiers, send.c et recv.c, pour tester ces fonctionnalités, ainsi que l'ajout des définitions correspondantes dans le fichier noyau. Une structure spécifique, message_t, a été créée pour représenter les messages, comprenant un tableau pour le contenu (message), la longueur (length), et le PID de l'expéditeur (sender_pid). Cette structure est utilisée pour définir une file d'attente statique, msg_queue[MAX_PROCESS] [MAX_MSGS], qui associe chaque processus à une liste de messages. Enfin, dans le fichier syscall.h, les longueurs et les PIDs ont été déclarés dans la structure système pour garantir leur gestion et leur synchronisation dans les appels systèmes.

On a commencer par compiler et exécuter les fichiers avec: **gcc -fPIC -c send.c**, **gcc -shared -o send.so send.o scc.so**, **gcc -fPIC -c recv.c**, **gcc -shared -o recv.so recv.o scc.so**

4. Comment tester notre programme :

Pour tester **signal** et **wait**, commentez tout sauf `spawn("./signal.so")` et le `printf` associé, puis commentez la boucle `for` dans `while(1)`. Pour tester **send** et **recv**, commentez tout sauf `spawn("./send.so")` et son `printf`, puis décommentez la boucle `for`. Cela permet de mieux observer la synchronisation entre les processus.

Conclusion

L'implémentation des appels systèmes et des primitives de synchronisation dans `my_little_kernel.c` est fonctionnelle. Les tests confirment leur validité et leur conformité avec les spécifications, offrant une coordination efficace entre processus et un échange fiable de messages.