Projet de Spécialité du master informatique parcours SAR

Mémoire Partagée Répartie

Encadrant :

- Pierre Sens

Membres de l'équipe :

- Maxime Derri
- Damien Albisson
- Changrui Zhang

22/05/2023

Table des matières

1 Introduction

Au sein d'une même machine, il existe plusieurs solutions pour faire communiquer différents processus.
Ces méthodes sont appelées IPCs (Inter-process Communication). On y retrouve les fichiers, les tubes, les signaux, les mémoires partagées et bien d'autres.
Cependant, pour faire communiquer plusieurs machines nous devons utiliser les sockets qui sont des points de communications sur un réseau.
Des machines communiquant sur un tel système ne peuvent pas nativement partager leur mémoire entre elles.
De plus, un tel système pose plusieurs problèmes de conceptions au niveau de la cohérence des données et des accès concurrents.

Comment implémenter une telle mémoire partagée entre différentes machines, en assurant la cohérence des données ?

Le projet suivant a pour but d'y répondre en définissant une interface au travers d'une bibliothèque dynamique.

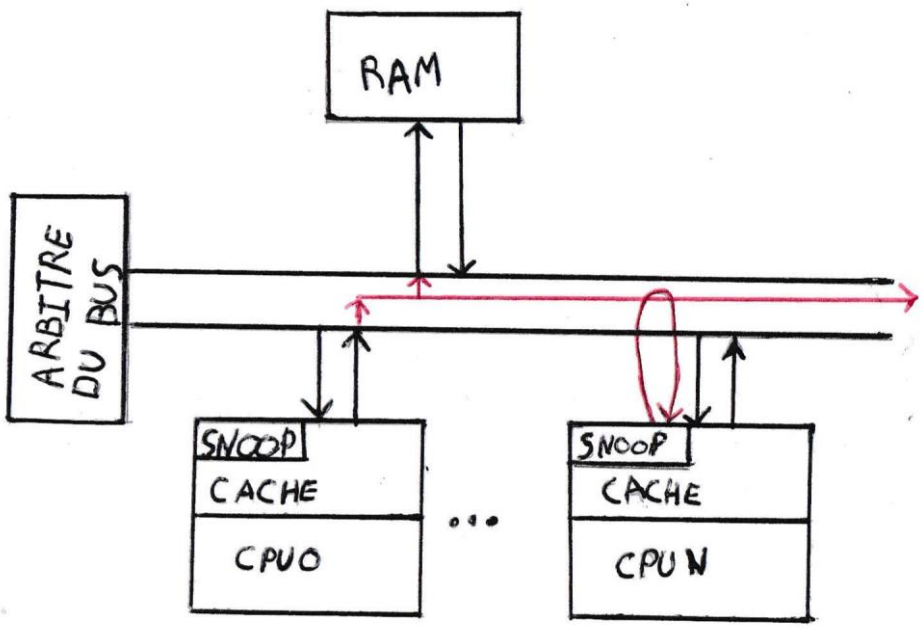
Dans un premier temps, nous expliquerons l'architecture qui sera appliquée dans ce projet, afin de définir une mémoire partagée répartie.
Dans un deuxième temps, nous détaillerons le protocole applicatif qui sera utilisée pour faire communiquer les machines sur le système répartie.
Dans un troisième temps nous aborderons la configuration ainsi que l'interface de la bibliothèque générée.
Enfin, nous discuterons de tests pour cette interface.

2 Architecture du projet

2.1 Problèmes de conception

Plusieurs problèmes de conception se posent pour définir l'architecture à utiliser :

- Pour la gestion des communications sur le réseau, il faut faire un choix sur le protocole applicatif à définir pour différencier les échanges de messages entre maitre et esclaves ainsi que sur le protocole de transport.
  Ici, nous choisirons TCP comme protocole de transport pour des communications sans perte en mode connecté.
- Le mécanisme de gestion de la mémoire sur l'esclave, pour le traitement des défauts de page et la mise à jour de ses pages.
  Cela sera effectué avec userfaultfd ainsi que les signaux et des mises à jour basé sur des invalidations. Une page mise à jour sur le maitre par un esclave sera invalidée sur tous les autres esclaves utilisant cette page. La mise à jour sera provoquée naturellement par un défaut de page lors de la tentative d'accès à cette page.
  L'invalidation est plus efficace que de forcer la mise à jour car cela permet de gagner du temps au niveau des réponses aux invalidations et une page mise à jour ne sera pas forcément réutilisée par un esclave.
  Ce mécanisme est aussi utilisé pour la cohérence des caches des processeurs sur certaines architectures multi-cœur (par exemple le « snoop »).



- La cohérence des données entre les esclaves et le maitre. Une page accédée doit être mise à jour, et le maitre doit posséder les versions les plus à jour des pages du système.
- Les accès concurrents des esclaves sur les pages du maitre ne doivent pas provoquer d'interblocage et chaque demande d'un esclave doit pouvoir être traitée dans un temps fini pour qu'il ne reste pas bloquer pour toujours.

2.2 Structures

Nous allons maintenant introduire les différentes structures qui seront utilisées dans le projet.

- struct connection {
  struct sockaddr_in adr;
  int32_t socket;
  }

La structure **connection** définie un point de connexion dans le système.

Pour les esclaves, nous définissons la structure suivante :

- Struct dsm_slave {
  void *ptr;
  struct connection co
  struct dsm_slave *next;
  pthread_t th;
  pthread_mutex_t mutex;
  <uint_CONFIG_LEN_t> size;
  int32_t uf_fd;
  uint8_t run;

  }

Où:

- **ptr** est le pointeur vers la base du segment de mémoire partagée de l'esclave,
- **size** est la taille du segment, dont le nombre d'octets dépend de la configuration (détaillé plus loin),
- **co** est utilisé pour communiquer avec le maitre,
- **mutex** est utilisé pour synchroniser le thread gérant les pages et le code de l'utilisateur au travers des lock/unlock
- **th** est le thread associé à l'esclave qui s'occupe des pages,

- **run** indique au thread de terminer ou non (while(run)),
- **uf_fd** est le descripteur userfaultfd associé à une plage mémoire
- **next** définit une liste chainée, dans le cas où un esclave pourrait posséder plusieurs maitres (on détermine quel maitre selon **ptr** et **size**).

Pour le maitre, nous définissons les structures suivantes :

- struct page {
  struct pthread_mutex_t mutex;
  }

Où les **mutex** serviront à bloquer l'accès à des pages. Une fois verrouillées, le maitre changera les droits de la page correspondant avec mprotect().
Un tableau de **struct page** représentera les différentes pages du segment afin d'assurer la cohérence au niveau des accès des esclaves. On définit un mutex par page, pour ne pas bloquer les accès d'autres esclaves qui manipuleraient des pages différentes.
En ce sens, il faudra parcourir le tableau de gauche à droite.

- struct slave {
  struct slave *next;
  struct connection co;
  uint8_t page_map[NBR_PAGE];
  }

Où :

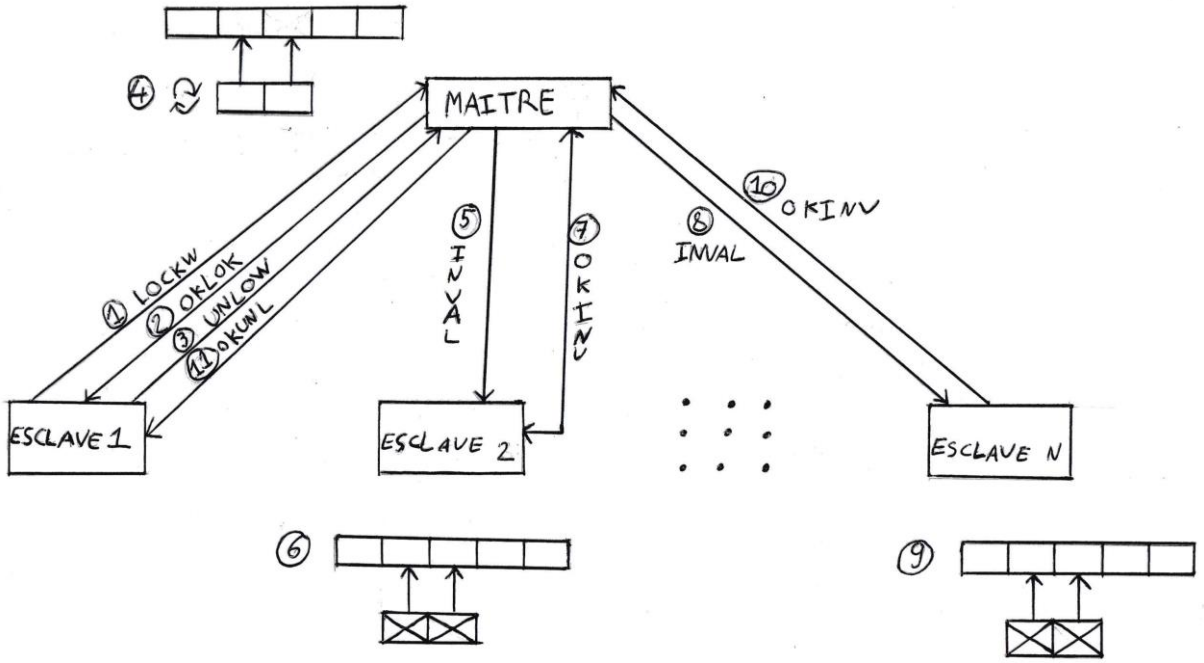
- **co** est aussi utilisé pour communiquer avec un esclave,
- **page_map** sert comme flag pour indiquer si un esclave possède ou non une page. Ce champ est modifié en fonction des mises à jour et des demandes d'invalidation de pages vers les esclaves. Chaque case correspond à une page.
- **next** est utilisé pour définir une liste chainée, afin de stocker dans chaque chainon un esclave.

- struct dsm_master {
  void *ptr;
  struct page *pages;
  struct slave *slaves;
  struct connection co;
  <uint_CONFIG_LEN_t> size;
  }
- **ptr** est le pointeur vers la base du segment de mémoire du maitre,
- **size** est la taille du segment (et configurable comme l'esclave pour la cohérence),
- **pages** référence les mutex des pages,
- **co** est le point de connexion du maitre, pour recevoir des demandes de connexion des esclaves.
- **next** est utilisé pour définir une liste chainée, afin de stocker dans chaque chainon un esclave.

2.3 Gestion de la mémoire

On appel maitre le processus sur le système répartie qui arbitre la cohérence du système.
Il est donc chargé de la gestion des pages et des droits d'accès aux pages.
Chaque esclave, est soumis à un maitre.
Le maitre exécute une fonction loop_master() afin de traiter les requêtes des esclaves. Chaque requête provoque la création d'un thread plutôt que de créer un thread par esclave directement. On utilisera donc select ou poll.
Le but est de diminuer le nombre de thread s'exécutant simultanément sur le maitre.
Une redéfinition du signal SIGINT permettra aussi de quitter cette fonction et de terminer le programme du maitre proprement.

Quand une ou plusieurs pages sont mises à jour sur le maitre, celui-ci envoie une requête aux esclaves possédant cette ou ces pages afin de les invalider et attend confirmation des esclaves avant de finaliser la libération du verrou.
Les prochaines tentatives d'accès des esclaves provoqueront un défaut de page et les pages seront mises à jour.



Si un esclave essaie d'accéder à une partie de son segment sans l'avoir bloqué auparavant, alors la modification ne sera pas prise en compte et risque d'être écrasée par les requêtes d'invalidation du maitre.
Ainsi, il faut bien faire attention à utiliser les fonctions de lock/unlock définies dans l'interface.

Du fait que les accès aux pages par les esclaves (lock) provoquent un mécanisme d'exclusion mutuelle, la mise à jour des pages par l'esclave sur le maitre est effectuée au moment de l'appel à l'une des fonctions unlock.

Chaque esclave possède un thread dont l'objectif est la gestion des pages de son segment de mémoire associé au système répartie.
Ce thread utilisera le point de communication bidirectionnel avec le maitre, ainsi que la fonctionnalité userfaultfd des distributions Linux.
Quand le maitre envoie une requête d'invalidation aux esclaves, chacun des threads analysera le contenue de la requête et changera les droits d'accès sur les pages en questions du segment de l'esclave avec mprotect et PROT_NONE. Un accès futur provoquera de nouveau un défaut de page, qui sera traité avec un signal.

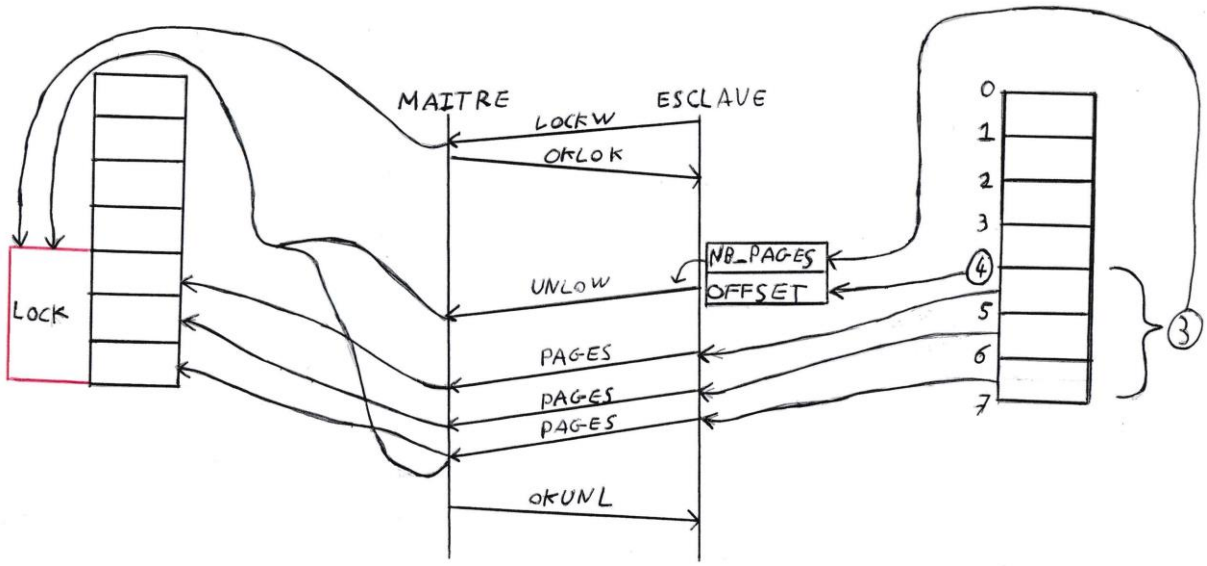
Lors d'un défaut de page, il faut distinguer deux cas. En effet, au cours du développement nous avons rencontré un problème avec userfaultfd et les services qu'il produit. Ce mécanisme fonctionne lorsqu'une page n'est pas encore mappée, mais si c'est le cas, nous pouvons uniquement placer des protections sur les écritures. Dans ce cas, si on tente une lecture alors aucun défaut de ne sera détecté…

C’est pour cette raison que les signaux sont actuellement la seule solution pour traiter les défauts après qu’une page ait été mappée dans le segment (si une modification du noyau n’est pas souhaitée).

Ainsi:

- Au premier accès, la page n’est pas mappée et c’est userfaultfd qui permettra de le détecter.
- Aux autres accès, c’est un gestionnaire du signal SIGSEGV qui s’en chargera. Un test sur la zone fautive sera effectué pour décider si le signal a été levé dans notre contexte ou non.

Ensuite, une requête est envoyé au maitre afin d’actualiser le segment de mémoire de l’esclave. Le maitre retournera la page correspondante, et le thread mettra à jour la page du segment de l’esclave. Les droits d’accès aux pages seront également changés du au traitement.



Pour ne pas mélanger les requêtes attendues par le thread, et par le code de l’utilisateur au niveau des lock et unlock, chaque accès au socket et mise à jour de la mémoire partagée sera protégé par une section critique (le mutex).
Pendant cette période, les fonctions lock/unlock de l’interface devront gérer les demandes d’invalidations des pages et des mises à jour (simple boucle tant qu’il y a encore des requêtes dans le buffer de réception et appel de la fonction correspondante).

À l’inverse, quand le thread verrouille le mutex et que l’utilisateur fait un lock ou unlock, le code utilisateur sera mis en attente.

Les fonctions de lock et unlock pour les lectures et écritures définies dans l'interface sont utilisées la synchronisation sur les accès et les modifications des pages. Quand les fonctions de lock se termine, on a la garanti d'un accès exclusif à la partie verrouillée.
Aucun autre esclave ne pourra y accéder tant que la fonction unlock correspondante ne sera pas appelée. Cela implique que toutes les mises à jour des pages et des droits soient effectuées sur le maitre et que les invalidations soient effectuées sur les esclaves avant libération.

3 Protocole applicatif

Nous allons maintenant détailler le protocole applicatif qui est implémenté.

Connexion :

- S -> M : |JOIN?|
- M -> S : |JOIN!|<CONFIG_LEN>size|
- M -> S : |ERROR|

size indique la taille du segment du côté maître. Il doit être un multiple de la taille d'une page.

Déconnexion :

- S -> M : |DECO?|
- M -> S : |DECO!|

DECO? est une demande de l'esclave d'arrêter la communication. Quand le maitre à fini de traiter la requête, il retourne DECO!.

Fin :

- M -> S : |RELEA|

Le maitre envoie |RELEA| quand il termine proprement (appel d'une fonction de l'interface).

Lock :

- S -> M : |LOCKR|<CONFIG_LEN>offset|<CONFIG_LEN>size|
- S -> M : |LOCKW|<CONFIG_LEN>offset|<CONFIG_LEN>size|
- M -> S : |OKLOC|
- M -> S : |ERROR|

LOCKR/W verrouille en lecture ou en écriture et lecture les pages à partir d'**offset**. **offset** est la position dans le segment à partir de l'adresse de base du segment.

size est le champ saisie de l'utilisateur, donc le maitre devra arrondir à la page supérieur si nécessaire.

Unlock :

- S -> M : |UNLOR|<CONFIG_LEN>offset|<CONFIG_LEN>size|
- S -> M : |UNLOW|<CONFIG_LEN>offset|<CONFIG_LEN>size|
- M -> S : |OKUNL|
- M -> S : |ERROR|

size est le champ saisie de l'utilisateur, donc le maitre devra également arrondir à la page supérieur si nécessaire.

Echanges d'informations (pages / invalider):

- M -> S : |INVAL|<CONFIG_LEN>offset|<CONFIG_LEN>nbr_page|
- S -> M : |OKINV|
- M <-> S : |PAGES|<CONFIG_LEN>offset|octets page|
- S -> M : |GETPG|<CONFIG_LEN>offset|
- M <-> S : |ERROR|

INVAL ordonne d'invalider **nbr_page** à partir de **offset** (sur une frontière de page).

OKINV est envoyé par l'esclave pour annoncer que l'invalidation a bien été effectuée.

PAGES peut être envoyé par l'esclave pour mettre à jour le maitre, ou par le maitre pour remplir un défaut de page d'un esclave.

GETPG est une requête envoyée par l'esclave pour demander une page, dans le cas d'un défaut de page.

Les entiers placés dans les requêtes seront mis sous big-endian, et devront être remis dans le sens de l'host à la réception (***endian.h***).

Quand un esclave libère en écriture, il envoie en premier UPDAT.
Ensuite, l'esclave envoie en fonction du nombre de page une série de requête PAGES (afin de maintenir la cohérence des données).
Quand c'est fini, le maitre envoie ses requêtes d'invalidation.
Enfin, l'esclave envoie UNLOW et attend la réponse OKUNL du maitre.

<CONFIG_LEN> dépend de la configuration, comme pour les champs size des structures dsm_slave et dsm_master.

4 Interface logicielle et configuration

4.1 Fonctions

Nous allons maintenant lister les fonctions qui seront disponibles dans l'interface de la bibliothèque dynamique.

- void *init_master(size_t size, uint16_t port)

  Initialise le maître. Retourne un pointeur vers le segment de mémoire partagée répartie de taille size (arrondir à la page supérieur si besoin).

- void *init_slave(char *host_master, uint16_t port)

  Initialise l'esclave ayant pour maître host_master et retourne l'adresse de la mémoire partagée répartie. La taille sera stockée dans la structure correspondante.

- void size_dsm(size_t *size, void *ptr, uint8_t who)

Retourne la taille du segment de mémoire partagée située à l'adresse **ptr**.
size stock la taille ou NULL si pas trouvé.
who indique si on cherche la taille depuis le maitre ou esclave par des macro MASTER et SLAVE.
Chaque instance d'un programme utilisant l'interface peut définir un esclave, un maitre ou les deux, on fait donc la distinction (en fonction de ces cas, les pointeurs sur ces structures seront initialisés ou NULL).

- int lock_read(void *adr, size_t size)

Verrouillage en lecture.

- int lock_write(void *adr, size_t size)

Verrouillage en lecture et écriture.

- int unlock_read(void *adr, size_t size)

Demande de libération du verrou.

- int unlock_write(void *adr, size_t size)

Demande de libération du verrou (et provoque les échanges de mise à jour / invalidation).

- int loop_master()

Faire exécuter au maitre son rôle (while avec select ou poll).

- int destroy_master()

Termine proprement le maitre.

- int destroy_slave()

Termine proprement l'esclave.

Les arguments utilisent size_t pour rester généraliste, mais il faudra faire attention aux valeurs données pour correspondre aux tailles définies par la configuration.

Les codes de retour seront détaillés dans les fichiers header.

4.2 Configuration dans le Makefile

Pour la configuration, des variables sont définis dans le Makefile générant la bibliothèque dynamique afin de paramétrer l'exécutable :

- ARCHI : 32 ou 64 bits (utilisation du préprocesseur pour choisir uint32_t ou uint64_t : CONFIG_LEN).
- BUFFER : Modifier la taille du tampon dans la communication (réception des pages principalement).

CONFIG_LEN vaudra par défaut 64 et le buffer ne sera pas modifié, si les valeurs des macros ne sont pas correctes, ou si les macros ne sont pas définies.

Pour la compilation, tout est expliqué dans README.md, situé à la racine du projet.
Il suffit de générer la bibliothèque avec la commande make à la racine du projet.
Il est également possible de modifier ARCHI et BUFFER dans la partie supérieur du Makefile.

5 Tests

Pour tester notre travail, nous avons ajouté un sous répertoire contenant des sous répertoires avec des tests :

- Des codes de test proposés
- Un fichier texte expliquera le contenue du test et comment il fonctionne
- Un Makefile sera disponible pour chaque test, afin de simplifier son utilisation. Il faudra cependant penser à générer la bibliothèque dynamique car elle sera forcément utilisée dans les tests.

Le but de cette section est donc de vérifier si les contraintes de conception ont été réglées, et de montrer des contextes d'utilisation de la mémoire partagée répartie.
Par exemple, l'un de nos test est le calcule d'une suite de Fibonacci par plusieurs esclaves.
Une structure est proposée contenant les différents éléments nécessaires au calcul de la suite ainsi qu'un compteur afin que les esclaves puissent s'arrêter après n

étapes. Ce type de test permet de montrer la cohérence des données et que l'exécution ne pose pas d'interblocage.

6 Conclusion

À travers ce document et les différentes sections, nous avons proposé une implémentation pour le sujet du projet de spécialité du parcours SAR traitant de l'implémentation des segments de mémoire partagée répartie au travers d'une bibliothèque dynamique et son interface.
L'architecture que nous avons proposée est une possible réponse aux problèmes de conceptions discutées dans les sections précédentes, mais plusieurs autres solutions existent.

Nous avons décrit la façon dont notre programme gère les segments de mémoire avec les différents mécanismes disposés au tour assurant la cohérence des données et du traitement des messages échangés sur le réseau, mais aussi au niveau des interblocages des différents esclaves lors de la tentative d'accès aux pages du maitre.

Nous avons également parlé de la façon dont nous avons dû adapter notre code au problème posé par userfaultfd, au niveau des accès aux pages déjà mappées dans le segment.

Pour pallier à ce problème, il est peut-être possible de proposer des améliorations du code en essayant de trouver une solution pour userfaultfd afin de remplacer les signaux par son utilisation complète.
Cela nécessiterait de toucher au code du noyau afin de proposer un patch mais pourrait poser des problèmes de compatibilité entre version des systèmes d'exploitations des machines esclaves.
Une interface userfaultfd complétée par cette modification permettrait une amélioration des performances.
Une autre amélioration serait le traitement de taille de pages différentes mais il faudrait adapter le code, et trouver un compromis sur la délimitation des segments et des tailles de pages différentes entre les machines.