Telecom ENSEIRB-MATMECA

Projet de programmation Système et Réseau

Implémentation d'une Mémoire Partagée Distribuée (DSM)

Salma Azzouzi
Lucas Fauchereau

Module : PR204
2021/2022

Table des matières

Introduction

Ce projet a pour but de développer un système de mémoire partagée et distribuée sur des machines en réseau. Il est découpé en deux phases, dont la première a pour but le développement d'un lanceur de programmes, qui a pour tâche de lancer différents processus sur des machines distantes via SSH, tout en récupérant et en centralisant les sorties standards et d'erreur de ces machines. Ce premier programme permet aussi de mettre en place tout le nécessaire pour la DSM dans l'étape suivante, comme la mise en place de sockets d'écoute menant à une interconnexion de tous les processus entre eux.
La deuxième phase de ce projet a pour but la mise en place du protocole de DSM grâce à tout ce qui a été mis en place lors de la première phase. Dans cette phase ont donc été implémentés l'échange des pages et la gestion des accès mémoire par les différents processus en réseau.
Nous nous proposons donc dans ce rapport d'expliciter les solutions choisies et les difficultés rencontrées au cours de ce projet dans un premier temps pour la première phase, puis dans une deuxième partie nous parlerons de la deuxième phase du projet.

1 Phase 1

La première phase a consisté, comme indiqué dans l'introduction, à la réalisation de programmes permettant le lancement de processus distants via SSH, avec une centralisation des sorties standards et d'erreur de ces programmes distants. Les deux programmes en question sont appelés `dsmexec` et `dsmwrap`, et ils seront évoqués dans les sous-sections suivantes.

1.1 Le programme `dsmexec`

Le premier des deux programmes implémentés est en effet le programme `dsmexec`, qui est le lanceur de processus à proprement parler. La première étape de ce programme est de lire un fichier nommé *machine_file*, fichier contenant un nom de machine distante par ligne non vide.

1.1.1 Lecture du *machine_file*

La lecture du fichier *machine_file* se fait en deux étapes, dont la première est le décompte du nombre de lignes non vides dans ce fichier, car il correspond au nombre de machines distantes à lancer. Ce comptage est réalisé grâce à une ouverture du fichier avec la fonction `fopen`. Ceci nous permet ensuite d'utiliser la fonction `fgets`, qui va lire ligne par ligne notre fichier, jusqu'à obtenir la valeur *EOF*, et on exécute donc la fonction tant qu'on n'obtient pas cette valeur, en incrémentant le compteur *num_procs* à chaque nouvelle ligne. On va donc utiliser cette fonction une première fois juste pour compter le nombre de lignes sans lire leur contenu.
Ensuite, nous pouvons lire les lignes du fichier une par une en utilisant d'abord la fonction `rewind` pour remettre la tête de lecture au début du fichier, puis en utilisant une deuxième fois la fonction `fgets`. Ensuite nous pouvons stocker le nom des machines dans une structure `dsm_proc_conn`, via le champ `machine` qui stocke une chaîne de caractères, et en leur attribuant un numéro via le champ `rank`, qui correspond à sa position dans *machine_file*.

1.1.2 Tubes de communication

Un des usages de `dsmexec` est la centralisation des sorties standards des processus distant. Cette étape passe par la création de tubes de communication permettant au processus `dsmexec` de communiquer avec ses processus fils, qui eux communiquent en SSH avec les machines distantes. Une fois le nombre de machines distantes connu, on peut allouer la mémoire nécessaire au stockage des descripteurs de fichier pour les tubes de communication.

Pour cela, nous avons créé un tableau tri-dimensionnel, noté fds, indexé de cette manière : $fds[i][j][k]$, avec :

- i représentant le rang du processus
- j représentant `stdout` lorsqu'il vaut 0, et `stderr` lorsqu'il vaut 1
- k représentant l'extrémité en lecture du tube lorsqu'il vaut 0, et l'extrémité en écriture lorsqu'il vaut 1

Par exemple, $fds[2][0][1]$ est le descripteur de fichier de l'extrémité en écriture du tube de communication pour `stdout` du processus n°2.
On obtient donc grâce à ce tableau toutes les extrémités des tubes pour `stdout` et `stderr` pour chaque processus, ce qui nous servira pour l'étape de centralisation des sorties standards.

Après l'étape précédente, nous pouvons créer les processus fils, et la première chose à faire après la création des processus fils est la redirection des flux. Chaque processus fils doit dont rediriger sa sortie standard et sa sortie d'erreur sur l'extrémité en écriture du tube qui lui correspond. Il va donc d'abord fermer le descripteur de fichier de sa sortie standard avant de dupliquer le descripteur de fichier de l'extrémité en écriture correspondant à la sortie standard (autrement dit dupliquer $fds[i][0][1]$ où i est son numéro de rang) puis faire les deux mêmes étapes avec sa sortie d'erreur (donc avec $fds[i][1][1]$).

1.1.3 Création du nouveau tableau d'arguments

Le but principal de `dsmexec` est de lancer les processus distants, et pour cela il a besoin de créer autant de processus fils qu'il a besoin de processus distants. Chaque processus fils fera exécuter le programme `dsmwrap` (qui sera explicité dans la sous-section 1.2) sur les machines distantes, et `dsmwrap`, lui, exécutera le programme final voulu par l'utilisateur, avec tous ses arguments.
Il est donc nécessaire de créer un nouveau tableau d'arguments que recevra `dsmwrap` avant de le redonner au programme final, avec quelques arguments en moins, comme nous le verrons plus tard. Le nouveau tableau d'arguments créé doit contenir ces éléments suivants (dans cet ordre) en plus des arguments fournis par l'utilisateur au lancement du programme :

- ssh, pour indiquer que l'on veut exécuter un programme sur une machine distante
- le nom de la machine distante sur laquelle on veut exécuter le programme
- dsmwrap, qui est le nom du processus intermédiaire
- le nom de la machine sur laquelle `dsmexec` a été lancé
- le port de la socket d'écoute créée au préalable par `dsmexec`
- le nombre total de processus distants qui seront créés
- $NULL$, pour indiquer qu'il s'agit de la fin des arguments

Les arguments fournis pas l'utilisateur au lancement de `dsmexec` se situent entre le troisième nouvel argument et le quatrième : entre l'argument "dsmwrap" et le nom de la machine sur laquelle le processus a été lancé. Grâce à cela, les processus fils de `dsmexec` peuvent effectuer leur `execvp`.

1.1.4 Échange des informations de connexion des processus distants

Une phase importante du programme est l'échange des informations concernant les sockets créées par `dsmexec` et par `dsmwrap` (pour rappel, on en parle dans la sous-section 1.2). De temps en temps, l'écriture des informations dans la socket renvoyait une erreur de type EPIPE, et avec cette erreur nous recevions le signal SIGPIPE, qui arrêtait tout simplement notre programme. Le choix que nous avons alors fait a été de masquer le signal SIGPIPE, et ainsi d'ignorer les cas où on écrit dans la socket mais personne n'est là de l'autre côté pour lire. Le masquage a été effectué grâce aux fonctions `sigaddset` et `sigprocmask`.

1.1.5 Récupération des processus fils "morts"

Une fois que les processus fils de `dsmexec` ont fini leur exécution, ils "meurent" et si on ne fait rien pour les récupérer, ils restent dans l'état zombie. Au moment de leur mort, il envoient à leur père le signal SIGCHLD. Nous avons donc décidé de mettre en place un traitant de signal pour SIGCHLD, qui contient un `wait` pour récupérer le fils mort, et qui par ailleurs décrémente un compteur qui stocke le nombre de processus fils encore vivants.
Nous savons cependant que mettre une fonction bloquante telle que `wait` dans un traitant de signal n'est probablement pas la meilleure des idées, mais c'est la seule que nous ayons trouvée.

1.1.6 Centralisation des sorties standard et d'erreur

Un autre des rôles majeurs de `dsmexec` est de centraliser les affichages des machines distantes, tout en affichant de quelle machine l'information vient, et même plus précisément s'il s'agit de sa sortie standard ou de sa sortie d'erreur. Pour ce faire, nous utilisons le tableau explicité en 1.1.2, couplé avec la fonction `poll`.
Nous remplissons un tableau de STRUCT POLLFD avec tous les descripteurs de fichiers du tableau fds déclaré précédemment, et en bouclant dessus, nous arrivons à savoir de quel processus l'information provient, et même de quelle sortie il s'agit, pour écrire le "filtre" correspondant (il contient le rang du processus, le nom de la machine sur laquelle il tourne et la sortie concernée). La lecture des données dans les tubes s'effectue caractère par caractère jusqu'à obtenir la valeur EOF, car on ne connaît pas à l'avance la taille des données qui sont à récupérer. C'est donc après avoir obtenu EOF que nous affichons l'entièreté des données récupérées.

1.2 Le programme `dsmwrap`

Comme évoqué dans la sous-section précédente, le programme `dsmwrap` a joué de nombreux rôles importants dans la mise en place de cette première phase du projet.

1.2.1 Échange des données avec `dsmexec`

La première action importante de ce programme est la récupération des données envoyées par `dsmexec`, premièrement par les arguments qui lui sont passés. On récupère ainsi le nom de la machine qui l'exécute, ainsi que le port de sa socket d'écoute et le nombre total de processus lancés. Ces trois arguments vont être cruciaux car il vont permettre à `dsmwrap` de récupérer le reste des informations dont il a besoin grâce à cette socket. Il va donc s'y connecter et récupérer les données dans l'ordre bien précis spécifié dans le sujet du projet.
Enfin, il va créer sa propre socket d'écoute, dont le descripteur de fichier est appelé $master\_fd$, dont il va envoyer les informations à `dsmexec`.

1.2.2 Sockets dans les variables d'environnement

Après avoir créé sa socket d'écoute, notre programme doit pourvoir la "léguer" au programme qui va lui succéder à travers l'`execvp`. Pour cela, il va stocker les descripteurs de fichier de ses deux sockets ($dsmexec\_fd$ pour communiquer avec `dsmexec` et $master\_fd$ pour communiquer avec les autres processus plus tard) dans des variables d'environnement, respectivement nommées DSMEXEC_FD et MASTER_FD. Comme les descripteurs de fichiers "survivent" après un `exec`, alors le processus suivant pourra y accéder sans problème. Cependant, nous avons eu un petit problème lors de la réalisation de cette tâche. En effet, nous avons inversé les valeurs de ces deux descripteurs de fichiers dans les variables d'environnement au moment d'utiliser `setenv`, et ainsi le protocole n'a pas pu fonctionner lors de la phase d'évaluation de notre programme.

1.2.3 Nouveau tableau d'arguments

Enfin, la dernière tâche de `dsmwrap` avant l'`exec` final est de créer un dernier tableau d'arguments, dépourvu de toute chose dont le programme suivant n'aura pas besoin. Pour cela, on récupère simplement les arguments "utiles" au milieu de tous ceux que reçoit `dsmwrap` et on les rassemble dans un nouveau tableau qui sera passé au processus final (à savoir le processus de DSM) à travers l'`exec`.

2 Phase 2

La mise en place de la bibliothèque de DSM est assurée par le programme `dsm.c`. Tout d'abord, on récupère les informations de connexions envoyées par `dsmexec` à tous les processus distants, cela se fait à travers les variables d'environnement précédemment ajoutées dans `dsmwrap`. Ensuite, les connexions inter-processus sont établies en suivant la convention suivante :

2.1 Communications inter-processus

L'algorithme de connexion est :

- Chaque processus initialise un tableau *conn_sockets* de taille *DSM_NODE_ID* .
- Il crée par la suite des sockets de connexions avec les processus dont le rang est inférieur à son rang et il se connecte avec eux en stockant les descripteurs de fichier à la position *"rang"* du tableau et leur envoie son rang.
- Pour le reste des processus, il accepte leur demande de connexion et reçoit le rang de chacun afin de stocker le descripteur de la socket à la bonne position dans le tableau.

2.2 Mise en place d'un traitant de signal

Les pages sont réparties en tourniquet entre les processus et chaque processus possède un tableau *table_pages* où sont stockées les informations sur le propriétaire et l'état de chaque page.
Quand un processus veut accéder à une page qui ne lui appartient pas, un signal SIGSEGV se déclenche. Ce signal est capté par le traitant de signal, ce dernier utilise le champ *sa_sigaction* pour indiquer l'action affectée au signal et localiser l'emplacement mémoire ayant causé l'erreur.
A partir de cette adresse mémoire, on récupère le numéro de page, le propriétaire et son descripteur de fichier. On envoie ensuite une structure de type *dsm_req_t* qui contient le rang du demandeur, le numéro de la page, le nouveau propriétaire avec un champ type qui indique le type de la requête. Dans ce projet, on suit la convention suivante :

DSM_REQ pour demander une page

DSM_PAGE pour envoyer une page

DSM_NREQ pour demander à tous les processus de mettre à jour le tableau *table_pages*.

DSM_FINALIZE pour synchroniser la terminaison des processus une fois qu'ils arrivent à *dsm_finalize*.

2.3 Le thread de communication

Un thread de communication est créé pour traiter les requêtes des autres processus. Un tableau de *pollfds* est initialisé à la taille *DSM_NODE_NUM -1* et il est rempli avec les descripteurs des sockets précédemment créées. La fonction `poll` détecte s'il y a de l'activité sur ce tableau de descripteurs. Quand on reçoit une requête, le type de la structure *dsm_req_t* est examiné. Selon ce type, le processus doit :

- A la demande d'un autre processus, il copie le contenue de la page dont le numéro est indiqué dans le champ *numpage* dans un buffer puis il libère la page et envoie une requête de type *DSM_PAGE* au demandeur pour lui informer que la page a été envoyée. Juste après l'envoi de la structure, il envoie le buffer qui contient la page en elle même. Pour s'assurer que tous les processus ont la même table de pages, le processus envoie la structure de type *DSM_NREQ* à tous les autres processus.
- A la réception d'une requête de type *DSM_PAGE*, le processus alloue la page et change les informations du tableau, il reçoit ensuite le buffer de données qu'il copie à l'adresse de la nouvelle page.

2.4 Synchronisation inter-thread

Le problème de synchronisation se pose quand le demandeur d'une page envoie sa requête et n'attend pas la réponse, ce qui déclenche une boucle infinie de signal SIGSEGV. La solution que nous avons adoptée est la mise en place d'un sémaphore qui a été déclaré en variable globale et initialisé dans *dsm_init* avec zéro jeton. Dans le traitant et après avoir envoyé la demande, on décrémente le sémaphore en prenant un jeton et puisque ce dernier est vide, le thread se bloque dans le traitant mais le thread de communication attend toujours de l'activité.
Quand on reçoit la réponse dans le thread de communication, on alloue la page en question et on rajoute un jeton au sémaphore, ce qui entraîne le déblocage du thread qui va continuer cette fois-ci sans déclencher un autre signal SIGSEGV .

2.5 Synchronisation inter-processus

Il faut que tous les processus se terminent en même temps, comme cela on est sûr si l'un aura besoin d'une page, son propriétaire sera encore en vie pour lui répondre. Pour ce faire, on déclare un pointeur d'entier *Nbr_finalize* de taille le nombre de processus moins un en variable globale et on l'initialise dans *dsm_init* avec des zéros. Quand un processus exécute *dsm_finalize*, il envoie aux autres une structure de type *DSM_FINALIZE*. Dans le thread de communication, à la réception d'un type *DSM_FINALIZE*, on cherche dans le tableau la position correspondante à la source et on met sa valeur à un.
A chaque fois que l'on reçoit une requête d'un autre type que *DSM_FINALIZE*, on remet la valeur à la position de la source à zéro car un processus peut entrer dans le `dsm_finalize` et puis recevoir une demande sur le thread de communication et donc basculer de nouveau à `dsm_init`.
Le processus examine dans `dsm_finalize` le tableau et s'il trouve que toutes les positions sont mises à un, alors tous les processus sont prêts à terminer, il fait alors un `pthread_join` sur le thread de communication et termine en fermant les sockets.
Cependant, quand le thread reçoit l'information qu'il est le seul programme encore actif et donc qu'il est sensé s'arrêter, il est bloqué entre le début de la boucle `while` et le `poll`. Une solution est alors de créer un tube de communication dont on est à la fois lecteur et écrivain, de compter le descripteur de l'extrémité en lecture dans les descripteurs à vérifier avec `poll` (donc l'ajouter au tableau $pollfds$) puis envoyer quelque-chose dans ce tube quand on souhaite terminer le thread, pour qu'il sorte du `poll` et donc du thread.
Malheureusement, nous avons essayé de mettre cette solution en place mais cela n'a pas fonctionné, donc le thread ne se termine *presque* jamais.