ENSEIRB-MATMECA
BORDEAUX-INP

2A TÉLÉCOMMUNICATION
PR204 - PROGAMMATION SYSTÈME

Compte rendu du projet PR204

Élèves :
Manâl RHAZZA
Oussama RAJI

Enseignant :
Guillaume MERCIER
Joachim BRUNEAU - QUEYREIX
Philippe SWARTVAGHER

24 décembre 2024

Implémentation d'une Mémoire Partagée Distribuée

24 décembre 2024

Table des matières

1 Introduction

Ce projet consiste à développer un système de **mémoire partagée distribuée (DSM)**, permettant à plusieurs processus sur différentes machines de partager une plage d'adresses commune. Le projet se divise en deux phases : le lancement et la gestion des processus distants, puis l'implémentation de la mémoire partagée. Ce travail combine des concepts de programmation système et réseau, offrant une exploration des défis liés aux systèmes distribués.

2 Lancement des processus

2.1 Introduction

La première phase de ce projet vise à développer un système permettant le lancement et la gestion de processus distants sur plusieurs machines. Cette étape repose sur la création d'un programme lanceur, `dsmexec`, qui a pour rôle de déployer les processus distants en utilisant des connexions SSH et d'assurer une communication centralisée entre ces processus. Cette phase inclut également le développement de composants complémentaires tels que `dsmwrap` pour le traitement des commandes sur les machines distantes, et des bibliothèques partagées comme `common.c` et `common_impl.h` pour les opérations communes. Chaque composant joue un rôle essentiel pour garantir une initialisation fiable des processus et une gestion robuste des ressources.

2.2 DSMEXEC

Le fichier `dsmexec.c` constitue une partie essentielle de la phase 1, responsable du lancement et de la gestion des processus distants. Les sections suivantes décrivent les différentes fonctionnalités implémentées dans ce fichier.

Configuration du gestionnaire de signaux (SIGCHLD) Un gestionnaire pour le signal `SIGCHLD` a été mis en place afin de traiter les processus fils terminés et d'éviter la création de processus zombies. La configuration utilise `struct sigaction` avec les options `SA_RESTART` et `SA_NOCLDSTOP` :

- `SA_RESTART` garantit que les appels système bloquants comme `read()` ou `accept()` reprennent automatiquement après une interruption par un signal. Cela permet d'éviter des erreurs dues à des interruptions inattendues.
- `SA_NOCLDSTOP` empêche l'appel du gestionnaire lorsque des processus fils sont suspendus ou repris, car ces événements ne sont pas pertinents dans ce contexte.

Dans le gestionnaire de signal, `waitpid` avec l'option `WNOHANG` est utilisé pour traiter tous les processus fils terminés sans bloquer l'exécution principale. À chaque terminaison d'un processus, le compteur global `num_procs_creat` est décrémenté, et un message est affiché pour indiquer la fin du processus. Cette approche assure une gestion efficace des ressources tout en minimisant les interruptions dans le programme principal.

Lecture et gestion du fichier de machines Le fichier contenant les noms des machines où les processus distants doivent être exécutés est traité en deux étapes : comptage

des machines et initialisation des structures de données.

Ouverture et validation du fichier : Le fichier est ouvert avec `fopen` en mode lecture. En cas d'erreur (fichier inexistant ou non accessible), un message est affiché avec `perror`, et le programme termine.

Comptage des machines : Une première boucle parcourt chaque ligne avec `fgets` pour compter le nombre de machines valides. Chaque ligne est nettoyée des caractères de fin de ligne (`\n`) avec `strcspn`. Les lignes vides ou incorrectes sont ignorées pour garantir que seules les entrées valides sont comptées. Ce comptage détermine la taille des tableaux dynamiques nécessaires pour stocker les informations des processus.

Allocation dynamique des tableaux : Les tableaux `proc_array`, `pipe_stdout` et `pipe_stderr` sont alloués avec `malloc` en fonction du nombre de machines compté. Une vérification des échecs d'allocation est réalisée, et le programme termine si une erreur survient.

Initialisation des structures : Une deuxième boucle lit à nouveau le fichier à l'aide de `rewind` et remplit les structures `dsm_proc_t` dans `proc_array`. Chaque machine est associée à un numéro de rang (index de la boucle), et ses champs comme le port et les descripteurs de connexion sont initialisés à des valeurs invalides (`-1`). Cela garantit une cohérence dans les données avant l'établissement des connexions.

Fermeture du fichier : Une fois toutes les informations extraites, le fichier est fermé avec `fclose` pour libérer les ressources.
Cette phase garantit que toutes les machines spécifiées dans le fichier sont correctement identifiées et préparées pour les opérations ultérieures.

Création de la socket d'écoute La socket d'écoute joue un rôle crucial en permettant aux processus distants de se connecter au lanceur. Voici les étapes détaillées :

Création de la socket : Une socket TCP est créée avec `socket(AF_INET, SOCK_STREAM, 0)`. Si l'opération échoue, une erreur est signalée avec `perror`, et le programme termine.

Configuration de réutilisation du port : La fonction `setsockopt` est utilisée avec l'option `SO_REUSEADDR` pour permettre la réutilisation rapide du port. Cela est utile en cas de redémarrage du programme, où le port pourrait être marqué comme utilisé par le système.

Initialisation de l'adresse serveur : Une structure `sockaddr_in` est configurée pour spécifier :

- `sin_family = AF_INET` pour utiliser IPv4.
- `sin_addr.s_addr = htonl(INADDR_ANY)` pour accepter les connexions sur toutes les interfaces réseau disponibles.
- `sin_port = 0` pour demander au système d'attribuer un port libre.

Association de l'adresse : L'adresse est associée à la socket avec `bind`. Une erreur entraîne l'arrêt immédiat du programme.

Récupération du port dynamique : Après l'association, `getsockname` est utilisé pour obtenir le port attribué dynamiquement. Ce port sera communiqué aux processus distants pour les connexions ultérieures.

Mise en écoute : La fonction `listen` configure la socket pour recevoir des connexions, avec une file d'attente maximale définie par `SOMAXCONN`. Cela permet au programme de gérer simultanément plusieurs connexions.

Création des processus fils et exécution SSH Chaque processus distant est lancé à l'aide d'un processus fils créé avec `fork`. Dans le processus fils, une commande `ssh` est construite et exécutée pour établir une connexion avec la machine distante.

Redirection des sorties : Les sorties standard (`stdout`) et erreur (`stderr`) sont redirigées via des tubes (`pipe`). Ces redirections permettent au lanceur de centraliser et de traiter les sorties des processus distants.

Construction de la commande SSH : Une boucle est utilisée pour gérer tous les arguments supplémentaires passés à `dsmexec`. Le tableau d'arguments pour `ssh` inclut :

— Les options SSH (`-o StrictHostKeyChecking=no`).

— Le nom de la machine distante.

— Le programme intermédiaire `dsmwrap`.

— Les arguments spécifiques, comme le port d'écoute du lanceur et le nom de l'hôte.

Exécution de la commande : Une fois construite, la commande est exécutée avec `execvp`. Si une erreur survient, elle est affichée avec `perror`, et le processus se termine. Cette méthode permet une flexibilité maximale en prenant en charge des arguments dynamiques pour chaque exécution.

Acceptation des connexions des processus distants Après la mise en écoute de la socket, `accept` est utilisé pour établir des connexions avec les processus distants. Chaque connexion est traitée comme suit :

— Le descripteur de la connexion est stocké dans `proc_array`.

— Les informations de connexion, comme le numéro de port du processus distant, sont lues et enregistrées pour une utilisation ultérieure.

Gestion des E/S avec `select` La récupération des sorties des processus distants est réalisée de manière centralisée à l'aide de `select`. Ce mécanisme surveille simultanément plusieurs descripteurs :

— Les descripteurs des tubes `pipe_stdout` et `pipe_stderr` sont ajoutés à un ensemble (`fd_set`).

— `select` détecte les descripteurs prêts pour la lecture, évitant ainsi une attente active.

— Les données disponibles sont lues avec `read` et affichées avec des informations contextuelles (rang du processus, nom de la machine).

— Si un descripteur est fermé, il est retiré de l'ensemble pour éviter des erreurs.

Libération des ressources Avant de terminer, le programme libère toutes les ressources utilisées :

— Tous les descripteurs ouverts (tubes, sockets) sont fermés proprement.

— Les tableaux alloués dynamiquement sont libérés avec `free`.

Cette gestion garantit une exécution propre et sans fuite de mémoire, même en cas d'exécutions répétées.

2.3 DSMWRAP

Le fichier `dsmwrap.c` joue un rôle essentiel en tant que processus intermédiaire pour préparer et exécuter les commandes passées par le lanceur `dsmexec`. Il permet également de transmettre les informations nécessaires à la communication entre processus dans la phase d'initialisation `dsm_init`.

Validation des arguments en entrée Le programme commence par vérifier qu'au moins trois arguments sont fournis : le numéro de port, le nom de l'hôte du lanceur, et le chemin vers le programme à exécuter.
Cette validation garantit que toutes les informations nécessaires sont disponibles avant de poursuivre. Si les arguments sont insuffisants, un message d'erreur est affiché, et le programme se termine avec `EXIT_FAILURE`. Cette vérification protège contre des comportements imprévisibles en cas d'utilisation incorrecte.

Création et configuration de la connexion avec le lanceur Une connexion TCP est établie avec le lanceur pour transmettre des informations essentielles à la phase d'initialisation. Les étapes suivantes sont réalisées :

— Une socket est créée avec `socket(AF_INET, SOCK_STREAM, 0)` pour établir une communication fiable en TCP. En cas d'échec, un message d'erreur est affiché, et le programme termine.

— L'adresse du lanceur est configurée dans une structure `sockaddr_in`. Le numéro de port est converti en format réseau (`htons`) et assigné au champ `sin_port`.

— Une tentative est faite pour convertir directement `launcher_host` en adresse IP avec la fonction `inet_pton`. Cette approche permet une gestion rapide et efficace des cas où le nom d'hôte est directement une adresse IP valide. Si la conversion échoue (retour $\leq$ `0`), cela indique que `launcher_host` n'est pas une adresse IP.
Dans ce cas, une résolution DNS est effectuée :

  — La fonction `gethostbyname` est utilisée pour convertir `launcher_host` en une structure `hostent` contenant les informations réseau associées. Si cette résolution échoue, une erreur est signalée avec `herror`, et le programme termine.

  — Les informations obtenues (adresse IP sous forme binaire) sont copiées dans le champ `sin_addr` de la structure `sockaddr_in` à l'aide de `memcpy`. Cela permet d'assurer que l'adresse est prête pour une connexion.

Cette logique hybride permet une gestion efficace des cas où `launcher_host` peut être soit une adresse IP, soit un nom d'hôte. Elle garantit une compatibilité maximale avec différentes configurations réseau.

Transmission d'informations au lanceur Une fois la connexion établie, `dsmwrap` envoie des informations essentielles au lanceur :

— Le nom de la machine hôte, obtenu avec `gethostname`, permet d'identifier la machine sur laquelle le processus s'exécute.

— Le PID du processus `dsmwrap`, envoyé via `write`, peut être utilisé pour la journalisation ou la traçabilité par le lanceur.

Création de la socket d'écoute Pour établir des connexions avec d'autres processus `dsm`, une socket d'écoute est configurée :

— Une nouvelle socket TCP est créée avec `socket(AF_INET, SOCK_STREAM, 0)`.

— L'option `SO_REUSEADDR` est activée via `setsockopt` pour permettre une réutilisation rapide du port en cas de redémarrage.

— Une structure `sockaddr_in` est configurée pour écouter sur toutes les interfaces locales (`INADDR_ANY`) avec un port dynamique (`sin_port = 0`).

— La socket est liée à cette adresse via `bind`, et le port attribué dynamiquement est récupéré avec `getsockname`.

Les informations de cette socket, notamment le numéro de port, sont ensuite transmises au lanceur pour être partagées avec les autres processus `dsm`.

Préparation des arguments et exécution du programme distant Avant d'exécuter le programme final, les arguments sont nettoyés et préparés :

— Seuls les arguments nécessaires au programme distant sont extraits des arguments passés à `dsmwrap`.

— Le chemin absolu du programme est résolu avec `realpath` pour éviter les problèmes liés à des chemins relatifs.

— Toutes les ressources inutilisées, comme les descripteurs de socket, sont fermées avant l'exécution pour prévenir les fuites.

— La commande est exécutée avec `execvp`, remplaçant le processus actuel par le programme distant.

Gestion des erreurs et nettoyage En cas d'erreur à n'importe quelle étape, un message explicite est affiché pour faciliter le débogage. Toutes les ressources allouées dynamiquement, comme le tableau des arguments, sont libérées avant la terminaison. Cela garantit une exécution robuste et propre.

2.4 COMMON

Le fichier `common.c` regroupe des fonctions utilitaires partagées entre le lanceur `dsmexec` et le processus intermédiaire `dsmwrap`. Ces fonctions permettent d'uniformiser et de simplifier les opérations courantes, comme la gestion des erreurs, la configuration des sockets, et la sécurisation des communications.

Gestion des erreurs La fonction `error` fournit un mécanisme centralisé pour gérer les erreurs critiques. Lorsqu'une erreur survient, un message descriptif est affiché via `perror`, puis le programme termine avec `EXIT_FAILURE`. Cette approche standardise la gestion des erreurs, facilitant le débogage et garantissant une terminaison propre en cas de problème.

Création et configuration des sockets Plusieurs fonctions utilitaires ont été développées pour gérer les sockets TCP de manière robuste :

— `do_socket` : Cette fonction encapsule la création d'une socket TCP avec une gestion des interruptions. Une boucle `do-while` est utilisée pour relancer la création en cas d'interruption (`errno == EINTR` ou `EAGAIN`). Cela garantit que la fonction n'échoue pas à cause d'une interruption temporaire, augmentant ainsi la fiabilité.

— `do_bind` : Associe une socket à une adresse spécifiée. Si l'opération échoue, un message explicite est affiché, et le programme termine.

— `do_listen` : Configure la socket pour accepter des connexions entrantes. La file d'attente maximale des connexions est définie par un paramètre, permettant une flexibilité selon les besoins.

— `do_accept` : Permet d'accepter des connexions entrantes avec gestion des interruptions. Une boucle `do-while` assure que l'appel est relancé en cas d'interruption, évitant des retours inutiles d'erreur.

Ces fonctions encapsulent les appels système bas-niveau, réduisant les risques d'erreurs et rendant le code principal plus lisible.

Initialisation des adresses réseau Deux fonctions sont implémentées pour configurer des adresses réseau de manière uniforme :

— `init_serv_addr` : Configure une adresse serveur en spécifiant le port et en écoutant sur toutes les interfaces disponibles (`INADDR_ANY`).

— `init_client_addr` : Configure une adresse client avec une IP et un port spécifiques. La conversion de l'IP en format réseau est effectuée via `inet_addr`, garantissant la compatibilité avec les appels système.

Ces fonctions éliminent les duplications de code pour la configuration des structures `sockaddr_in` et garantissent une initialisation cohérente.

Connexion sécurisée à un serveur La fonction `do_connect` établit une connexion TCP avec un serveur. Une boucle `do-while` permet de relancer la tentative en cas d'interruption (`errno == EINTR` ou `EAGAIN`). Si la connexion échoue après plusieurs tentatives, un message d'erreur est affiché, et le programme termine.

Création d'une socket serveur complète La fonction `creer_socket_serv` regroupe plusieurs étapes pour configurer une socket serveur de manière compacte :

— Création de la socket avec `do_socket`.

— Initialisation de l'adresse avec `init_serv_addr`.

— Association de la socket à l'adresse via `do_bind`.

— Récupération du port dynamique attribué par le système avec `getsockname`, si le port initial est défini à zéro. Le port est ensuite retourné au programme appelant.

Cette fonction encapsule les étapes complexes pour réduire la duplication dans le code principal.

Communication sécurisée Deux fonctions sont implémentées pour gérer les communications de manière fiable, même en cas d'interruptions :

— `safe_write` : Écrit des données sur un descripteur de fichier de manière sécurisée. Une boucle permet de s'assurer que la totalité des données est écrite, même si des

interruptions (`EINTR`) surviennent. En cas d'erreur persistante, la fonction retourne `-1`.

— `safe_read` : Lit des données depuis un descripteur de fichier en gérant les interruptions. La lecture continue jusqu'à ce que toutes les données soient reçues ou que le descripteur soit fermé (EOF).

Ces fonctions sont particulièrement utiles pour garantir l'intégrité des échanges de données, en particulier dans des environnements réseau où les interruptions sont fréquentes.

3 Mise en place de la DSM

3.1 Introduction

La seconde phase du projet vise à implémenter un système de mémoire partagée distribuée (DSM), permettant à plusieurs processus de partager une plage d'adresses mémoire commune. La mémoire est divisée en pages, et chaque page est associée à un processus propriétaire.
Cette section décrit les principales étapes de mise en place de la DSM, incluant l'initialisation des processus, la gestion des pages mémoire et des signaux, ainsi que les communications entre processus.

3.2 Initialisation de la DSM : DSM_INIT

L'initialisation de la DSM repose sur la fonction `dsm_init`. Cette fonction commence par récupérer les valeurs des variables d'environnement `DSMEXEC_FD` et `MASTER_FD`, qui permettent de mettre en place les communications initiales entre les processus. Les descripteurs de fichier correspondants sont utilisés pour recevoir le nombre total de processus (DSM_NODE_NUM) et l'identifiant du processus courant (DSM_NODE_ID). Cette étape garantit que chaque processus dispose des informations nécessaires sur le système distribué.

Ensuite, les pages mémoire sont allouées de manière cyclique entre les processus grâce à une stratégie de répartition en tourniquet. Pour chaque page, la fonction `num2address` calcule son adresse à partir de son numéro, et la fonction `mmap` est utilisée pour réserver cette zone mémoire. L'utilisation de `mmap` est stratégique car elle permet une gestion précise de la localisation et des permissions des pages dans l'espace mémoire des processus.

Les connexions inter-processus sont ensuite établies à l'aide de sockets TCP. Chaque processus crée une socket d'écoute grâce à socket, la configure avec `bind` pour spécifier son adresse et son port, et utilise `listen` pour attendre les connexions entrantes.
Nous avons rencontré pendant un certain temps une erreur récurrente indiquant "connection refused". Ce problème survenait car tous les processus tentaient simultanément de se connecter les uns aux autres tout en essayant également d'accepter des connexions.
C'est pour cela que nous avons décidés que les processus de rang supérieur accepteraient les connexions via `accept`, tandis que ceux de rang inférieur se connecteraient aux autres à l'aide de `connect`.

Pour gérer les erreurs de segmentation (SIGSEGV), un gestionnaire de signaux est configuré avec sigaction.
Ce gestionnaire appelle une fonction personnalisée, `segv_handler`, qui identifie la page à l'origine de l'erreur et initie les actions nécessaires pour rétablir l'accès. Parallèlement, un thread de communication est créé à l'aide de `pthread_create`. Ce thread exécute la fonction `dsm_comm_daemon`, chargée de gérer les requêtes et les messages échangés entre les processus.
L'utilisation d'un thread dédié permet de répartir les tâches et de ne pas perturber l'exécution principale.

3.3 Gestion des requêtes et des signaux

Lorsque qu'un processus tente d'accéder à une page dont il n'est pas le propriétaire, une erreur de segmentation est générée. Le gestionnaire de signaux intercepte cet événement et extrait l'adresse fautive grâce à `info->si_addr`.
La fonction `address2num` est alors appelée pour traduire cette adresse en un numéro de page. La table des pages est consultée pour identifier le processus propriétaire actuel de la page.

Une fois le propriétaire identifié, une requête est envoyée à l'aide de `dsm_send`, contenant des informations comme le numéro de la page demandée et l'identifiant du demandeur. Le processus propriétaire reçoit cette requête avec `dsm_recv`, puis envoie le contenu de la page demandée. Avant cet envoi, il utilise `mprotect` pour ajuster les permissions sur cette page et met à jour son propre état, indiquant qu'il n'en est plus le propriétaire.

Du côté du processus demandeur, la page reçue est insérée dans son espace mémoire via `mmap`. Les permissions sont ajustées avec `mprotect` pour permettre un accès en lecture ou en écriture, selon les besoins.

3.4 Gestion des Communications

La gestion des communications repose sur un thread dédié, exécutant la fonction `dsm_comm_daemon`. Ce thread utilise select pour surveiller l'ensemble des sockets des processus connectés. select est une solution efficace pour gérer les entrées/sorties de manière non bloquante, ce qui permet au système de traiter simultanément plusieurs requêtes sans interrompre les calculs principaux.
Lorsqu'une requête est détectée sur une socket, le thread identifie son type. Dans le cas d'une demande de page, le thread gère l'envoi des données appropriées via `dsm_send`.

La combinaison de select et de sockets TCP permettant aux processus de continuer leur exécution normale tout en répondant aux interactions des autres processus.

Lors de l'exécution, il semble que les trois processus distants ne se lancent pas tous correctement, car seuls deux d'entre eux apparaissent dans le terminal. Cependant, en utilisant l'option `-XG`, nous constatons que les trois processus sont bien démarrés. Nous n'avons pas réussi à identifier précisément la cause de ce comportement, mais cela pourrait

être lié à la manière dont nous avons choisi d'établir les connexions inter-processus.

4 Répartition du travail

Ce projet a été réalisé en binôme, avec une collaboration active à chaque étape, de la conception initiale à la mise en œuvre des fonctionnalités. Nous avons travaillé ensemble sur toutes les parties du projet, en discutant et en prenant des décisions communes.

Pour mieux avancer, nous avons réparti certaines tâches tout en restant impliqués dans les travaux de l'autre. Travailler en équipe nous a permis d'avancer de manière fluide et de garder une bonne cohérence dans le projet. Chacun a pu apporter sa contribution, ce qui nous a aidés à surmonter les difficultés tout en tirant le meilleur de nos efforts partagés.

5 Conclusion

Ce projet nous a permis d'approfondir les notions de programmation système et réseau en développant un système de mémoire partagée distribuée. Nous avons exploré des concepts tels que la communication inter-processus, la gestion de la mémoire et le traitement des signaux, tout en relevant les défis des environnements distribués.

La première phase a introduit un mécanisme fiable pour le lancement et la gestion des processus distants avec dsmexec et dsmwrap. La seconde phase a abouti à un système DSM fonctionnel, avec une gestion dynamique des pages et des échanges efficaces entre les processus.

Malgré quelques comportements imprévus, comme des soucis de synchronisation au démarrage, ce projet constitue une base solide pour des améliorations futures, telles que la gestion avancée des droits d'accès ou l'ajout d'outils de supervision.

En résumé, ce travail nous a permis de mettre en pratique des concepts théoriques tout en ouvrant des perspectives d'amélioration.