Rapport de Projet

Gestion de la synchronisation pour la mise en œuvre d'une pile de communication dans un mini os

YAO Mingliang
Dépt. Sciences du Numérique
ENSEEIHT

Décembre 2023

Table des matières

Table des matières **2**
Intruduction **3**
Analyse des codes essentiels **4**
1.1 sf\tubes.c 4
1.2 include/manux/atomique.h 5
1.3 noyau/atomique.c 7
1.4 usr\init-acces-concurrent.c 8
Travail accompli **9**
Conclusion **11**

Intruduction

Je m'engage à configurer des outils de synchronisation pour garantir le bon fonctionnement des canaux de communication au sein de "ManuX" et à fournir des versions bloquantes et non bloquantes, l'objectif est d'incorporer des outils de synchronisation dans les opérations des canaux afin de garantir leur fiabilité dans un environnement multitâche. Les modifications spécifiques pourraient concerner la protection mutuelle des opérations de lecture/écriture des canaux ainsi que l'utilisation de variables conditionnelles pour les blocages et les réveils. Il est essentiel de s'assurer que les outils de synchronisation fonctionnent correctement dans un noyau réentrant.

Analyse des codes essentiels

1.1 sf\tubes.c

Ce code semble mettre en œuvre un système de tubes de communication au sein d'un système d'exploitation, généralement utilisés pour la communication inter-processus.
Voici une explication du code selon ses sections :

Définition des Structures et Constantes :
Il débute par l'inclusion de divers en-têtes nécessaires.
Il définit une structure Tube contenant des informations sur le tube, telles que le pointeur vers la zone de données, la taille actuelle, l'indice de la prochaine insertion, le nombre d'écrivains et de lecteurs.
Il établit également des constantes telles que la capacité du tube en nombre de pages (MANUX_TUBE_NB_PAGES) et la capacité totale du tube (MANUX_TUBE_CAPACITE).

Fonctions associées aux Fichiers (Open, Close, Read, Write) :

Il met en œuvre des fonctions associées à la manipulation de fichiers (ouverture, fermeture, lecture, écriture) pour les tubes.
La fonction tubeOuvrir gère l'ouverture d'un tube en tant que fichier, en prenant en compte les modes lecture et écriture.
Les fonctions tubeFermer, tubeEcrire et tubeLire correspondent aux opérations de fermeture, écriture et lecture du tube.

Méthodes Fichier pour le Tube :

Il déclare une structure MethodesFichier qui contient des pointeurs vers les fonctions associées aux fichiers, telles qu'ouvrir, fermer, écrire et lire.
Ces méthodes sont associées aux opérations sur le tube en tant que fichier.

Implémentation de l'Appel Système tube() :

Il inclut une implémentation possible de l'appel système tube(). Cet appel système crée un tube, alloue la mémoire nécessaire, initialise la structure du tube, crée un iNoeud pour le tube, puis crée deux fichiers (un pour la lecture et un pour l'écriture) associés à ce iNoeud. Ces fichiers sont ensuite ajoutés à la tâche en cours, et les descripteurs de fichiers associés sont retournés.

En résumé, le code fournit une infrastructure de base pour la gestion des tubes de communication, y compris la création, l'écriture, la lecture et la fermeture des tubes. L'implémentation de l'appel système tube() permet la création de tubes et la récupération des descripteurs de fichiers associés.

1.2 include/manux/atomique.h

Ce fichier d'en-tête implémente des opérations atomiques, des verrous mutex et des variables conditionnelles utilisées pour la synchronisation.

Voici une explication des principales sections du fichier :

Définition du type d'opération atomique :
typedef uint32_t Atomique;
Ce type est utilisé pour mettre en œuvre des opérations atomiques.
Macros pour les opérations atomiques :
atomiqueInit(atom, val):
Initialise la variable atomique atom avec la valeur val.
atomiqueLire(atom):
Lit la valeur de la variable atomique atom.
Opérations de comparaison et d'échange :
static __inline__ uint32_t compareEtEchange(uint32_t * ptr, uint32_t cond, uint32_t val);

Cette fonction implémente une opération de comparaison et d'échange, assurant un accès sécurisé aux ressources partagées dans les opérations atomiques.
Mutex (ExclusionMutuelle) :

```
typedef struct _ExclusionMutuelle {
 Atomique verrou;
 ListeTache tachesEnAttente;
#if defined(MANUX_ATOMIQUE_AUDIT)
 int nbEntrees;
 int nbSorties;
#endif
} ExclusionMutuelle;
```

Cette structure représente un verrou mutex, comprenant une variable atomique pour le verrouillage et une file d'attente pour les tâches en attente du verrou.

exclusionMutuelleInitialiser(ExclusionMutuelle * em):
Initialise le verrou mutex.

exclusionMutuelleEntrer(ExclusionMutuelle * em):
Entre dans le verrou mutex, bloquant si le verrou n'est pas disponible.

exclusionMutuelleSortir(ExclusionMutuelle * em):
Quitte le verrou mutex.

Variable conditionnelle (Condition) :

```
typedef struct _condition {
 ListeTache tachesEnAttente;
```

```
#if defined(MANUX_ATOMIQUE_AUDIT)
 int nbSignaler;
 int nbDiffuser;
#endif
} Condition;
```

Cette structure représente une variable conditionnelle, comprenant une file d'attente pour les tâches en attente de la condition.

conditionInitialiser(Condition * cond):
Initialise la variable conditionnelle.

conditionAttendre(Condition * cond, ExclusionMutuelle * em):
Attend sur la condition.

conditionSignaler(Condition * cond):
Émet un signal à une tâche en attente dans la file d'attente.

conditionDiffuser(Condition * cond):
Émet un signal à toutes les tâches en attente dans la file d'attente.

1.3 noyau/atomique.c

Ce segment de code représente une partie essentielle du système d'exploitation ManuX dédiée à la mise en œuvre d'opérations atomiques et d'outils de synchronisation.
Voici une vue d'ensemble des principales fonctionnalités :

Définition des opérations atomiques :

- compareEtEchange : Réalise une opération de comparaison et d'échange, garantissant son achèvement en une étape atomique.
- atomiqueTestInit : Compare la valeur d'une variable atomique avec une condition donnée et la modifie si elles sont égales.

Outils de synchronisation :

- ExclusionMutuelle : Définition de structure incluant un verrou mutex et une file d'attente.
- Condition : Définition de structure représentant une variable conditionnelle, avec une file d'attente associée.
- exclusionMutuelleInitialiser : Initialisation du verrou mutex.
- exclusionMutuelleEntrer : Entrée dans la section critique du verrou mutex, avec blocage en cas de verrouillage déjà en cours.
- exclusionMutuelleSortir : Sortie de la section critique du verrou mutex.
- conditionInitialiser : Initialisation de la variable conditionnelle.
- conditionAttendre : Attente jusqu'à ce que la condition soit satisfaite.
- conditionSignaler : Éveil d'une tâche en attente sur la variable conditionnelle.
- conditionDiffuser : Éveil de toutes les tâches en attente sur la variable conditionnelle.

Surveillance d'état :
En présence de compilations conditionnelles, le code intègre une surveillance d'état pour les mutex et les variables conditionnelles, incluant des compteurs d'entrées/sorties et les tâches en attente.

- exclusionsMutuellesAfficherEtat : Affiche l'état du mutex.
- conditionsAfficherEtat : Affiche l'état de la variable conditionnelle.

Allocation de mémoire :
Sous compilation conditionnelle, l'allocation de mémoire avec kmalloc est utilisée pour suivre une liste de mutex et de variables conditionnelles.

Dans l'ensemble, ce code implémente des outils de synchronisation fondamentaux tels que les mutex et les variables conditionnelles, assurant la cohérence des données dans un environnement multitâche.

1.4 usr\init-acces-concurrent.c

Ce code représente un programme d'initialisation en mode utilisateur dont la fonction principale est de créer un canal (pipe) et de lancer plusieurs tâches de lecture et d'écriture pour la communication. Voici une explication détaillée du code :
Inclusions de fichiers d'en-tête :

- <manux/types.h> : Inclut les définitions de certains types de données de base.
- <stdio.h> : En-tête de la bibliothèque standard d'entrée/sortie.
- <manux/errno.h> : Inclut les codes d'erreur.
- <unistd.h> : Inclut les déclarations de fonctions telles que creerNouvelleTache, lire et ecrire.
- <manux/string.h> : Inclut les déclarations de fonctions pour manipuler les chaînes de caractères.

Définitions de constantes :

- TAILLE_BUFFER : Définit la taille du tampon utilisé pour la lecture à 16.
- NB_LECTEURS et NB_ECRIVAINS : Définissent respectivement le nombre de lecteurs et d'écrivains.

Variables globales :

- int fd[2] : Stocke les descripteurs de fichier du canal.

Fonction du lecteur lecteur() :

- Lit les données depuis le canal et affiche les informations de lecture.
- Utilise la fonction lire pour lire les données du canal jusqu'à la fin.

Fonction de l'écrivain ecrivain() :

- Écrit des données dans le canal et affiche les informations d'écriture.
- Utilise la fonction ecrire pour écrire des données dans le canal jusqu'à la fin.

Fonction d'initialisation init() :

- Fonction d'entrée pour l'initialisation en mode utilisateur.
- Appelle tube(fd) pour créer un canal, affiche un message d'erreur en cas d'échec.
- Crée plusieurs tâches d'écrivains et de lecteurs, leur transmet les fonctions correspondantes (ecrivain et lecteur).
- Entre dans une boucle infinie pour maintenir l'exécution du programme.

Programme principal :

- Affiche un message de bienvenue.
- Appelle init() pour l'initialisation.
- Affiche un message de fin.
- Entre dans une boucle infinie pour maintenir l'exécution du programme.

Ce programme, exécuté en mode utilisateur, crée un canal et lance simultanément plusieurs tâches de lecteurs et d'écrivains, qui communiquent via ce canal. Le programme continuera de s'exécuter en continuant les opérations de lecture et d'écriture jusqu'à l'arrêt de la boucle infinie.

Travail accompli

L'examen minutieux du code **usr\init-acces-conrcurent.c** a conduit à des modifications substantielles visant à renforcer la sécurité et la fiabilité dans un environnement multithread. Une mesure cruciale introduite est l'utilisation du verrou mutex pour assurer un accès mutuellement exclusif à la ressource partagée. Ce mécanisme de synchronisation garantit qu'à tout moment, une seule tâche peut accéder à la ressource partagée protégée.

- Initialisation du Verrou Mutex

La ligne de code suivante initialise le verrou mutex nommé "mutex" au sein de la structure de données :
ExclusionMutuelle mutex;
exclusionMutuelleInitialiser(&mutex);
Cette initialisation signifie que, au commencement de l'exécution d'une tâche, le mutex n'est pas verrouillé, autorisant ainsi l'accès à la section critique.

- Accès Mutuellement Exclusif à la Ressource Partagée

Dans les fonctions lecteur et écrivain, des procédures spécifiques garantissent un accès mutuellement exclusif à la ressource partagée.

Fonction Lecteur :
exclusionMutuelleEntrer(&mutex);
// ... opérations de lecture de la ressource partagée ...
exclusionMutuelleSortir(&mutex);
Ce code assure qu'aucune autre tâche ne peut modifier la ressource partagée simultanément à sa lecture.

Fonction Écrivain :
exclusionMutuelleEntrer(&mutex);
// ... opérations d'écriture de la ressource partagée ...
exclusionMutuelleSortir(&mutex);
De même, cela garantit qu'aucune autre tâche ne peut lire ou écrire simultanément à l'écriture de la ressource partagée.

- Utilisation de Sémaphores

Les sémaphores, notamment semaphoreAttendre et semaphoreValeur, sont employés pour des opérations synchronisées, assurant une attente appropriée des données pendant la lecture et un espace suffisant pendant l'écriture. Cette approche permet une synchronisation efficace des threads.

- Structure de Données Tube

Dans la structure de données Tube, le verrou mutex (mutex) et deux signaux (attenteLecture et attenteEcriture) ont été introduits pour garantir un accès sécurisé aux données du tuyau par les threads. Cela permet aux threads de lecture et d'écriture d'attendre ou de poursuivre leur exécution de manière appropriée.

- Fonctions de Lecture et d'Écriture

Les fonctions de lecture et d'écriture ont été modifiées pour s'exécuter sous la protection d'un verrou mutex, assurant ainsi la sûreté des opérations dans un environnement multithread.

Dans le contexte du projet ManuX, les fonctions de lecture et d'écriture des tubes doivent gérer le comportement lorsque le tube est vide (pour la lecture) ou plein (pour l'écriture).

- Version Non Bloquante (Non-Blocking):

1. Lecture (lireInterne): La fonction de lecture non bloquante doit simplement lire autant de données que possible du tube. Si le tube est vide, elle renverra immédiatement, ne bloquant pas l'exécution.
2. Écriture (ecrireInterne): La fonction d'écriture non bloquante doit écrire autant de données que possible dans le tube. Si le tube est plein, elle renverra immédiatement sans bloquer.

- Version Bloquante (Blocking):

1. Lecture (lireInterne): La fonction de lecture bloquante doit attendre jusqu'à ce qu'il y ait des données disponibles dans le tube. Elle peut utiliser une variable conditionnelle associée à un signal de lecture pour être notifiée lorsque des données sont disponibles. L'attente peut être débloquée lorsqu'une autre tâche écrit dans le tube.
2. Écriture (ecrireInterne): La fonction d'écriture bloquante doit attendre jusqu'à ce qu'il y ait de la place disponible dans le tube pour écrire. Elle peut utiliser une variable conditionnelle associée à un signal d'écriture pour être notifiée lorsque de l'espace est disponible. L'attente peut être débloquée lorsqu'une autre tâche lit du tube.

Question de l'attente (attente de quoi ? jusqu'à quand ?) et du déblocage des flots d'exécution en attente:

1. L'attente se produit lorsqu'une tâche tente de lire sur un tube vide ou d'écrire sur un tube plein.
2. Pour la lecture bloquante, l'attente est de données disponibles, et elle se poursuit jusqu'à ce que des données soient écrites dans le tube.
3. Pour l'écriture bloquante, l'attente est de la place disponible, et elle se poursuit jusqu'à ce que de l'espace soit libéré par la lecture d'autres données.
4. Le déblocage des flots d'exécution en attente se produit lorsque les conditions nécessaires sont remplies (données disponibles ou espace disponible), déclenchant les signaux associés aux variables conditionnelles.

Dans tous les cas, les fonctions renvoient le nombre d'octets lus ou écrits (éventuellement 0 pour les versions non bloquantes). Une valeur négative représente une situation d'erreur (aucun octet disponible n'est pas une situation d'erreur).

1. Le renvoi du nombre d'octets lus ou écrits permet à l'appelant de connaître la quantité d'informations traitées.
2. Une valeur nulle pour les versions non bloquantes signifie qu'aucune opération n'a pu être effectuée en raison de l'absence de données (lecture) ou de l'absence d'espace (écriture).

3. Une valeur négative indique une situation d'erreur, ce qui peut être utile pour signaler des conditions exceptionnelles telles que des échecs d'opérations de lecture ou d'écriture.

En résumé, la gestion du comportement des fonctions dans différentes situations d'attente et de déblocage contribue à assurer un fonctionnement fiable et efficace des canaux de communication dans un environnement multitâche.

Conclusion

1. Mise en œuvre des fonctions de base : en analysant le fichier sf\tubes.c, les fonctions de base des tuyaux sont mises en œuvre, y compris la création de tuyaux et les opérations de lecture et d'écriture. Il fournit un support de base pour les opérations de pipeline, en fournissant une base pour la communication.

2. Mise en œuvre d'outils de synchronisation : des outils de base pour la synchronisation multithread sont mis en œuvre à l'aide d'opérations atomiques, de verrous mutex et de variables conditionnelles dans include/manux/atomique.h. Ces outils, y compris les variables ExclusionMutuelle et Condition, fournissent le support nécessaire à la synchronisation dans un environnement multithread.

3. Améliorations de la procédure en mode utilisateur : dans usr\init-acces-concurrent.c, la procédure d'initialisation a été améliorée par l'introduction de verrous d'exclusion mutuelle et de variables conditionnelles. Cela garantit la sécurité des tâches de lecture et d'écriture multiples lors de l'accès aux ressources partagées et améliore la robustesse du programme.

4. Différenciation des versions : des versions bloquantes et non bloquantes des fonctions de lecture et d'écriture ont été introduites pour répondre aux différents besoins dans les cas où le pipeline est vide (lecture) ou plein (écriture). Cela améliore l'applicabilité et la flexibilité du programme.

5. Surveillance de l'état : la surveillance de l'état des verrous mutex et des variables de condition a été introduite, y compris le nombre d'entrées et de sorties, et les files d'attente des tâches en attente. Cela facilite le débogage et la surveillance du fonctionnement du système.

Dans l'ensemble, la fiabilité et la sécurité de la communication par pipeline dans le système d'exploitation ManuX ont été améliorées par l'introduction d'outils de synchronisation. Le rapport fournit une analyse détaillée du code clé et démontre les améliorations substantielles apportées au programme.