Manuel de développement = [[PageOutline]]

1. Architecture du MUTEKP

La figure suivante illustre la modélisation du système :

Les bibliothèques constituant le système sont :

- pthread : contient l'interface système des Threads POSIX.
- libc : contient l?interface des services système tel que malloc, printf, read, memcpy ..etc.
- mwmr : contient l?interface système des FIFO MWMR.
- sys : contient le code système qui ne dépend pas de l'architecture de la plate-forme ou de type des processeurs utilisés.
- cpu : contient le code système en assembleur qui dépend de type des processeurs de la plate-forme.
- arch : contient le code C qui dépend de la plate-forme tel que les configurations du système vis-à-vis des composants de la plate-forme, les ISR d?interruption des différant types de cibles... etc.

En cas de modification au niveau de la configuration matérielle, il suffit d?adapter le code système des deux bibliothèques cpu et arch pour pouvoir déployer Mutek*P* sur la nouvelle plate-forme.

2. Introduction au noyau MUTEKP

2.1 Le concept d?un Thread dans le système

Un Thread est un fil d?exécution d?un programme.
Tous les Threads de l?application partagent le même espace d?adressage, où chaque Thread possède :

- Son propre contexte d?exécution (le PC, un pointeur de pile et d?autres registres de travail du processeur).
- Deux piles.
- Plie utilisateur
- Pile système

Quelques avantages :

- Création et gestion plus rapide (vs processus).
- Partage des ressources par défaut.
- Communication entre les Threads plus simple via la mémoire (les variables globales).
- Déploiement plus efficace de l?application sur des architectures multiprocesseurs.

2.2 États d?un Thread

La durée de vie d?un Thread peut être divisée en un ensemble d?états.
Les différents états d?un Thread sont :

- Run : quand le Thread s?exécute sur le processeur en faisant son calcul.

- Kernel : quand le Thread « tombe » dans le noyau suite à un appel aux services noyau ou une interruption matérielle.
- Wait : lors que le Thread demande une ressource qui n?est pas disponible.
- Ready : quand le Thread est en attente de gagner le processeur pour poursuivre son exécution.
- Zombie : quand le Thread se termine en attendant qu?un autre thread prenne acte de sa terminaison.
- Create : état spécial dans lequel le Thread vient d?être créé. Il n?a pas encore été chargé sur un processeur et attend de le gainer pour la première fois.
- Dead : état spécial dans lequel le Thread est déclaré définitivement mort. Toute tentative de joindre un Thread dans cet état échouera.

2.3 Ordonnancement des Threads

Le système partitionne l?ensemble des Threads de l?application en sous-ensembles dans le but de les ordonnancer. Le nombre de ces sous-ensembles est en bijection avec le nombre des processeurs disponibles dans la plate-forme matérielle. Il existe une structure d?ordonnancement, pour chaque sous-ensemble, responsable de l?ordonnancement de ses Threads selon sa propre politique d?ordonnancement. Une fois qu?un Thread est créé, il est affecté à un seul processeur tout au long de sa vie, le noyau Mutek*P* n?implémente pas la migration de tâches, ou la répartition dynamique de la charge du système.

Supposant que le nombre des processeurs dans la plate-forme est égal à N alors :

- Le système partitionne l?ensemble des Threads de l?application en N sous-ensembles.
- Le système dispose de N structures d?ordonnancement pour ordonnancer ces N sous-ensembles.
- En régime permanent (lors que chaque processeur est en train d?exécuter un Thread), il existe N Threads s?exécutant en parallèle, tandis que les autres Threads de chaque sous-ensemble s?exécutent en pseudo parallèle grâce au temps partagé (changement de contexte à chaque fin de quantum suite à une interruption horloge).

Ainsi le noyau Mutek*P* dispose d?un mécanisme d?ordonnancement distribué à tâches affectées. Dans le cas où l?Ordonnanceur d?un processeur ne trouve aucun Thread à l?état Ready, Il charge un Thread particulier nommé Thread Idle.
Cette situation peut se produire notamment au démarrage du système et avant la création des Threads de l?application, aucun Thread n?est disponible pour être chargé sur un processeur (exception du Thread main). Ou encore lors que tous les Threads d?un processeur sont en attente sur des ressources non disponibles.

L?utilité de ce Thread Idle est double :

- Pour ne pas bloquer le processeur vis-à-vis des interruptions et de pouvoir ainsi exécuter leurs ISR.
- Peut être programmé pour exécuter un code spécial de débogage ou d?observation de l?état du système.

2.4 Organisation et gestion de la mémoire

2.4.1 L'organisation de la mémoire

L?espace d?adressage est découpé en segments d'adresses contigües :

- Les registres de contrôle des périphériques
- Les segments de RAM (ROM) réservés au système
- Les segments de RAM (ROM) de l'utilisateur

Les informations sur les segments de mémoire dépendent de la plate-forme matérielle. Le système les connait par l'intermédiaire du fichier segmentation.h (dans le répertoire arch_soclib). Les propriétés (caché / non caché), (système / utilisateur) sont codées dans les adresses. La lecture d'une donnée dans un segment caché est d?abord

recherchée dans le cache du processeur.

- En cas de hit, le contrôleur du cache fournit la donnée au processeur évitant ainsi de réaliser un accès à la mémoire.
- En cas de miss, le contrôleur du cache réalise la mise à jour d'une ligne de cache avant de fournir la donnée au processeur.

Si un cache contient la copie du contenu d'une adresse mémoire et que cette adresse est modifiée par un autre processeur, alors le cache n'est pas à jour.

Conséquence de ce comportement

- Les variables globales de l'application et du système doivent être mappées dans un segment de type non caché.
- La structure de la pile d'un Threads est mappée dans un segment de type caché (puisqu?aucun Thread ne modifie pas la pile d'un autre Thread !).
- L'échange de données entre Threads passe par de la mémoire non caché ou une invalidation partielle du cache.

2.4.2 La gestion de la mémoire =

Quatre segments de mémoire data :

- mémoire cachée et non cachée système
- mémoire cachée et non cachée utilisateur

Le noyau gère ces zones mémoire d?une manière minimaliste, qui consiste à garder quatre pointeurs, dans une structure de donnée dédiée, définissant l?occupation de ces zones. À chaque allocation, l?espace disponible est vérifié, s?il n'y en a plus, le système retourne un pointeur nul, sinon il mis à jours le pointeur de la zone allouée. Le système ne propose pas de libérer une zone mémoire dynamiquement allouée.

2.5 Le buffer système

Le noyau dispose d?un tableau de tampons système. Chaque entrée de ce tableau est une structure de type FIFO MWMR réalisent un tampon. L?index de chaque tampon est utilisé en tant que descripteur du buffer Ces buffers système permettent au noyau de pouvoir stocker le flux de caractères qui arrivent depuis un terminal TTY en attendant qu?un Thread puisse les consommer (c.f: read())

3. Détails du noyau et des structures de données système

3.1 Gestion des Threads

3.1.1 La structure de donnée du Thread

3.1.2 La structure de donnée Ordonnanceur et la table d?ordonnancement

3.2 Gestion de la mémoire

3.2.1 La structure gestionnaire mémoire

3.3 Gestion des périphériques

3.3.1 La structure gestionnaire des verrous

3.3.2 Représentation des cibles (TTY, Timer et ICU)

3.4 Gestion des interruptions