Université Pierre et Marie Curie

ARCHITECTURE MANY-CORE

TP8 : ALMOS-MKH Procédure de boot

Auteur:
Thibault Mellier, William Fabre

Professeur:
Monsieur Franck Wajsburt, Alain greiner

Année 2019-2020

Contents

Questions sur le preloader

1.1 À quoi sert un preloader ?

Le preloader permet de charger le bootloader qui va charger l'OS. Le preloader est dépendant de l'architecture matériel.

1.2 Où se trouve stocké le code du preloader au démarrage de la plateforme TSAR? 1. Comment les coeurs accèdent_ils au code du preloader ?

Dans la plateforme TSAR LETI, le code du preloader est stocké à l'adresse 0 de l'espace d'adressage physique du cluster 0 (taille de 16 Ko) dans une mémoire non-volatile. Les autres coeurs accèdent au code du preloader en accédant à la mémoire physique du cluster 0 où est stocké celui-ci.

1.3 Comment fait-on pour qu'un code ne soit exécuté que par un seul core ?

On introduit une condition d'éxecution dans le code dépendant du numéro du core. Tout les autres cores se mettent donc en attente sauf celui concerné.

1.4 A quel moment les cores sortent-ils du preloader ?

Les cores sortent du preloader quand le core 0 leur envoie l'adresse de début du bootloader par le biais d'une IPI.

Questions sur le bootloader

2.1 À quoi sert le bootloader ? Quels périphériques doit il pouvoir accéder ? Quels formats de fichier doit-il pouvoir analyser ?

Le bootloader fait deux choses : charger l'OS dans la mémoire et charger arch_info.bin, fichier definissant l'architecture matérielle.

Le bootloader doit pouvoir accéder au périphérique où est stocké l'OS et à la mémoire physique pour transférer l'OS du stockage externe vers la mémoire principale. Il doit aussi accéder à un TTY et un IOC.

Il doit pouvoir analyser des formats ELF et BLOB (kernel.elf et arch_info.bin)

2.2 Dans quel(s) fichier(s) est rangée la description de la plateforme matérielle ?

La description de la plateforme matérielle est rangée dans le fichier arch_info.bin rangé dans le répertoire racine.

2.3 Pourquoi le code du bootloader est-il recopié dans tous les clusters ?

Le bootloader doit initialiser les segments kdata et kcode : en recopiant le code du bootloader dans tous les clusters, il initialisera une copie de kdata et kcode dans chaque cluster.

2.4 Pourquoi chaque core, a-t-il besoin d'une pile ? où sont-elles placées ?

Chaque core peut ne pas avancer en même temps que les autres et certaines données manipulées dépendent du GID c'est pourquoi il faut nécessairement des piles distinctes pour chaque core. Elles sont placées à BOOT_STACK_BASE, paramètre défini dans le boot_config.h.

2.5 Qu'est-ce qu'une IPI ? Comment sont-elles utilisées par le bootloader ?

Une IPI est une Inter Process Interrupt, il s'agit d'une interruption entre core permettant de déclencher des routines d'interruptions. Dans le bootloader, les IPI permettent de réveiller les core 0 endormis autre que celui du cluster 0 au début. Puis pour réveiller chaque core autre que 0 dans chaque cluster.

Questions sur kernel_init

3.1 Comment dans chaque cluster, le noyau a-t-il accès à la description de la plate-forme matérielle ?

kernel_init() prend pour paramètre une structure de type boot_info qui contient la description de la plate-forme matérielle. Comme kernel_init() est lançé dans chaque cluster, chaque cluster possède cette description dans le segment kdata.

3.2 Comment almos_mkh identifie-t-il les coeurs de la plate-forme matérielle ? Les threads kernel IDLE exécutent, sur chaque coeur la fonction kernel_init(). Le thread IDLE est aussi le thread choisi par le scheduler d'un coeur quand tous les autres threads placés sur ce coeur sont bloqués. Où sont rangés les descripteurs de threads IDLE ? Combien y a-t-il de threads idle ?

Il y a dans la structure boot_info l'ensemble des clusters ainsi que leur cores. Les cores sont identifiés par un unique id (GID) codé en dur qui permet de déduire CXY(numéro de cluster) et LID(numéro de core). Les descripteurs de threads IDLE sont rangés dans la section kidle dans le tableau idle_threads[]. Il y a CONFIG_THREAD_DESC_SIZE * CONFIG_MAX_LOCAL_CORES descripteurs de thread IDLE soit taille d'un descripteur x nombre max de cores dans un cluster.

3.3 Pour quelle raison la fonction kernel_init() utilise-t-elle des barrières de synchronisation ? Illustrez votre réponse avec un exemple.

kernel_init() est appelée par tous les cores, de plus, les cores partagent tous une instance du noyau ainsi que des périphériques. Les barrières de synchronisation permettent donc d'éviter qu'un des cores commençent à utiliser une structure ou un périphérique non initialisé. Par exemple, la première barrière évite que les cores utilisent le terminal TXT0 avant que le core 0 ne l'ai initialisé.

3.4 Pour quelle raison, utilise-t-on à la fois des barrières locales et des barrières globales ? Dans le cas d'une barrière globale, expliquez précisément la valeur du premier argument passé à la fonction xbarrier_wait(). L'API permettant l'accès à une barrière globale est définie dans les fichiers

Certaines initialisations concernent les clusters alors que d'autres concernent les cores. Pour cela, nous avons deux types de barrières : locales (barrière intra-cluster) et globales (barrière inter-cluster). Ces

barrières permettent donc de s'assurer que les traitements concernant les clusters et les cores soient terminés.

xbarrier_wait(XPTR(0 , global_barrier), (info-¿x_size * info-¿y_size)); Cette fonction permet d'accéder à une barrière se situant sur un autre cluster. XPTR possède deux arguments : le premier est le numéro de cluster cxy et le deuxième est le champ de la même structure située sur un cluster différent. La fonction xbarrier_wait possède elle aussi deux arguments : le premier étant l'appel à XPTR et le deuxième étant le nombre de clusters dans le 2D Mesh.

3.5 Quel est le rôle de la fonction cluster_init() définie dans le fichier almos-mkh/kernel/kern/cluster.c ?

La fonction cluster_init() permet de d'initialiser :

- PPM (Physical Page Manager)
- KHM (Kernel Heap Manager)
- KCM (Kernel Cache Manager)
- les cores du cluster
- les RPC FIFO
- le Process Manager

3.6 la structure DQDT est un bon example de structure de donnée distribuée sur tous les clusters. Quelle est l'utilité de la structure DQDT ?

DQDT (Distributed Quaternary Decision Tree) : Cette structure arborescente sert dans le cas de almosMKH a avoir une meilleur connaissance de l'utilisation des ressources décentralisées. Elle permet a definir la notion de voisinage entre noeuds cc-NUMA et donc une notion de voisinage entre les resources (memoires et cores). Chaque feuille de la DQDT est un cluster physique, chaque noeud est un niveau, cluster logique (ensemble de noeuds cc-NUMA). Cette organisation est decoree par une structure contenant : "le nombrede pages physiques libres M, le nombre de tâches actives T et le taux d'utilisation des cores U" par noeud qui permet aux noyaux de minimiser les distances d'acces aux ressources. (https://www-soc.lip6.fr/trac/almos/chrome/site/phd_thesis_ghassan_almaless_2014.pdf)

3.7 Quelle est la politique générale de almos-mkh pour gérer les accès concurrents à un même périphérique ? Qu'est-ce qu'un chdev ? Quelle est la politique de placement des chdev dans les différents clusters ?

Pour les opérations d'entrées/sorties, ALMOS-MKH implémente une politique bloquante : elle range le thread client de la ressource dans une file d'attente et l'endort jusqu'à ce qu'il soit réveillé par une ISR. Un chdev est un descripteur de périphérique. Il est représenté par une structure et permet d'interfacer différents types abstraits de périphériques comme TXT ou IOC. Le placement des chdev dans les différents clusters dépend du type de périphérique : Périphériques Internes : ces périphériques sont répliqués dans tous les clusters, ils ont donc des chdevs répliqués mais ceux ci sont partagés et accessibles par les autres clusters. Périphériques Externes : ces périphériques voient leur chdev distribués sur tous les clusters.