Gestion de la mémoire

Contraintes matérielles

- La mémoire physique fait 1 TB avec un segment d'adresses de 4Go par cluster
- TSAR est une architecture NUMA avec des temps d'accès à la mémoire dépendant de la localisation du thread et du banc accédé
  - 10 cycles en local
  - Plusieurs dizaines de cycles à distance
- Les processeurs ont une architecture 32 bits
- La MMU a 2 modes
  - Activée : 32b → 40b avec des pages de 4ko ou de 2Mo avec une table de pages à 2 niveaux
  - Désactivée : 32b → 40b = Extension 8b + adresse processeur 32b pas de table de pages

---

Besoins

- LES APPLICATIONS
  - Utilisent une mémoire virtuelle mappée sur des pages physiques dans tous les clusters.
    - Elles utilisent la mémoire virtuelle pour les instructions et les données.
  - Une grande page de 2 Mo pour le code du kernel (elles y vont pour les syscall, les interruptions et les exception).
  - Tout le reste (4 Go - 2 Mo) est pour l'utilisateur découpé en segments de plusieurs types (vus plus loin)
  - Les accès mémoire fréquents doivent être dans le cluster du processeur demandeur
- LE NOYAU
  - Utilise la mémoire virtuelle pour les instructions du kernel
    - Le noyau se contente d'une page de 2 Mo, il n'accède jamais au code de l'utilisateur
    - Le noyau a besoin d'accéder à tout l'espace physique et doit donc avoir un espace d'adressage de 40b au minimun.
    - Le noyau n'accède pas directement aux données de l'utilisateur, quand il doit allez chercher ou écrire des données dans l'espace de l'utilisateur, il passe par des API comme `copy_from_user()` ou `copy_to_user()`
    - Les données du kernel ne sont pas dans l'espace d'adressage des applications.

Espace d'adressage

- POUR L'UTILISATEUR, COMPOSÉS DE SEGMENTS D'ADRESSES TYPÉS
  - Définis à la création de l'exécutable
    - Code (`.text`)
    - Données globales (`.data`.)
  - Définis à la création du processus et des threads
    - Pile d'exécution des threads `.stack`
    - Données dynamiques créées par malloc `.heap`
  - Définis à l'exécution

        - Fichier mmap nommé
        - Création de segment d'adresses virtuelles anomyme (mmap anonymous)
- POUR LE KERNEL
    - Définis à la création du kernel
        - Code (`.text`)
        - Données globales (`.data`)
    - Définis par l'architecture
        - Devices (segments d'adresses choisi pour les registres des contrôleurs de périphériques)
    - Définis à l'initialisation du kernel
        - Segment(s) recouvrant toute la mémoire disponible
        - Ce(s) segment(s) est(sont) découpés en pages de 4ko, c'est l'atome d'allocation pour le kernel.
        - Il y a principalement 2 allocateurs mémoires que l'on ne détaille pas ici,
            - Un buddy allocator pour allouer des segments de pages physiques contigües (Buddy = copain)
            - Un slab allocator pour allouer des structures de tailles fixes pour le noyau (slab = dalle)
- ESPACE D'ADRESSAGE VIRTUELLE AVEC LA MMU → 4 GO
    - Permet d'accéder à tout l'espace d'adressage physique
    - Suffisant pour une application
    - Suffisant pour le code du kernel
    - Insuffisant pour les données du kernel
- ESPACE D'ADRESSAGE PAR EXTENSION D'ADRESSE DE 8 BITS → 1 TO
    - Permet d'accéder directement (lw/sw) à la mémoire du cluster
    - Permet d'accéder indirectement en changeant le registre d'extention

---

Propriétés des données

- FAITS
    - Les données sont stockées dans des pages (segment d'adresse atomique géré par le kernel)
    - Le "structures de données" (au sens large) de l'application occupent un ensemble de pages virtuelles contigües
    - Ces structures sont mappées sur des pages de l'espace physique qui n'ont pas de raison d'être contigües
- PROPRIÉTÉ D'USAGE
    - Une structure peut être privée, c'est-à-dire utilisée par 1 seul thread
    - Une structure peut être publique, c'est-à-dire utilisée par tous les threads
- PROPRIÉTÉ DE MAPPING
    - Une structure peut être localisée, c'est-à-dire mappée intégralement dans un cluster
    - Une structure peut être distribuée, c'est-à-dire mappée sur plusieurs threads
- PROPRIÉTÉ DE COPIE
    - Une structure peut être unique, c'est-à-dire avec une seule copie.
    - Une structure peut être répliquée à l'identique avec autant de copies qu'il y a de clusters
    - Une structure peut être répliquée multi-valuée, car chaque cluster a un état différent
- REMARQUES
    - Toutes les combinaisons de propriétés n'ont pas de sens. Par exemple, il n'existe pas de segment privé et distribué, ici, cela signifierait un segment utilisé par un seul thread mais

mappé dans plusieurs clusters
- Segments privés, localisés et répliqués à l'identique
  - code kernel et code user
- Segments privés, localisés et copie unique
  - stack kernel et stack user
- Segment publique, distribué et copie unique
  - data user
- Segment publique, localisé et répliqué multivalué
  - data kernel (attention ce n'est pas un segment virtuel)
- Segment publique, localisé et copie unique
  - mmap anonyme standard, **malloc()** utilise un segment mmap
  - mmap anonlyme remote (placé explicitement)
  - file

Principe de l'allocation des pages

- OBJECTIF
  - Lorsque que l'on créé un processus, on crée un espace d'adressage virtuel pour lui
  - Le mécanisme de création d'un processus dans un noyau unix consiste à dupliquer le processus qui est à l'origine de la demande par un **fork()**, puis à le remplacer par un nouveau programme avec **exec()**
  - Nous n'allons pas entrer, ici, dans le détail de ces opérations, nous allons seulement donner le principe de placement des pages physiques dans l'espace d'adressage virtuel.
- STRUCTURES DE DONNÉES DE L'ESPACES D'ADRESSAGE
  - Pour un processus, l'espace d'adressage est décrit par deux structures : VSL et GPT.
  - La liste des segments d'adresse virtuels ouverts : VSL (Virtual Segment List)
    - On rappelle qu'un segment est un ensemble contigüe d'adresses
    - Un segment ouvert est un segment dans lequel le processus peut faire une accès légal
  - La table des pages physique : GPT (Generic Page Table)
    - Cette table est générique au sens où elle est vue comme un tableau à une dimension indexée par les numéro de pages virtuelles, et que chaque case non-vide contient un numéro de page physique et ses droits d'accès.
    - En réalité, la GPT est représenté par une espèce de radix tree, c'est à dire une table de table, dont le nombre de niveau dépend de la longueur de l'adresse et de l'architecture interne de la MMU. Pour TSAR, elle à deux niveaux, pour Intel, elle en a 4.
- ALLOCATION DES PAGES POUR LE PREMIER THREAD
  - Lorsqu'un nouveau processus est créé, après un **fork-exec**, la VSL contient 2 segment, la GPT est vide.
  - La VSL ne contient que 3 segments ouverts
    - Le segment **code** pour la section de code présente dans le fichier elf de l'exécutable
    - Le segment **data** pour la section des données globales (initialisées ou pas) de l'exécutable
    - Le segment **stack** créé par le processus père pour permettre l'exécution du code
  - La GPT ne contient rien (je n'en suis pas absolument sûr pour almos-mkh, mais sur le principe, elle est vide). Elle contient certainement la page de premier niveau, mais ça c'est du détail de l'implémentation.
  - La première chose que fait le premier thread du processus, lorsqu'il démarre, c'est demander la première instruction de son code.

    - Cet accès en lecture d'une instruction provoque l'allocation d'une page physique dans le cluster local avec la première page du code.
        - Il y a d'abord un **miss TLB** puisque la traduction n'a jamais été demandée, et la MMU parcourt la table des pages (translation table walk) pointée par le registre PTPR présent dans la MMU
        - Puis un **page fault** puisque la page virtuelle sensée contenir le début du code n'a pas encore été mappé dans l'espace physique.
        - Il y a alors une excécution du gestionnaire d'exception pour cette faute de page et l'exécution de la fonction du traitement, allouer la page dans le cluster local et redemmarrer le thread.
        - La fonction va consulter la VSL afin de savoir si l'accès demandé est légal, sinon le processus va être stoppé.
- Puis elle va allouer une page physique pour le segment de code.
    - Le segment de code est de type "privé-localisé-recopié à l'identique". Le segment de code est lié au fichier elf contenant l'exécutable, alors la page contenant le code de ce fichier est recopiée dans le cluster du thread du processus demandeur et la GPT est mise à jour.
    - Le gestionnaire d'exception redonne le processeur au thread qui va à nouveau provoquer un miss TLB mais pas de page fault.
- Ensuite, le thread accède à sa pile et là aussi, cela va provoquer le mapping d'une page physique locale
    - il y a miss-TLB → page fault → gestionnaire d'exception → consultation VSL : segment légal de type STACK (privé, localisé, copie unique) → la page est allouée dans le cluster local.
    - le thread repart et réussit son accès (après un miss-TLB)
- Ensuite, le thread accède à ses données globales et cela va aussi provoquer le mapping d'une page physique.
    - miss-TLB → page fault → gestionnaire d'exception → consultation VSL : segment légal de type STACK (publique, distribué, copie unique) → la page est allouée dans un des clusters.
    - L'idée c'est de distribuer la charge d'accès au banc mémoire.
    - Le noyau suppose que le processus va contenir beaucoup de threads répartis sur tous les clusters. il n'est donc pas jusdicieux de mettre toutes les données globales dans un seul cluster.
    - Le noyau utilise les bits de poids faible du numéro de page virtuel pour choisir le cluster.
        - Deux pages virtuelles consécutives sont donc dans des clusters différents.
        - Notez que si le processus a peu de threads (1 ou 2), ce choix n'est pas idéal, mais le noyau n'a pas d'information sur le nombre de threads que le processus va créer.
- Ensuite, l'application va faire des mmap() anonymes, des malloc() ou des remote_malloc(), le placement des pages de ces segment est toujours localisé et dépend du thread qui les demande.
    - Pour malloc() et mmap() anomyme, les pages sont créées dans le cluster du thread demandeur.
    - Pour les remote_malloc(), le numéro du cluster est donné en argument

- ALLOCATION DES PAGES POUR UN THREAD DISTANT
    - Lorsqu'un processus créé beaucoup de threads, ceux-ci vont s'exécuter ailleurs que sur le cluster d'origine.
    - nous n'allons pas parler ici de la manière dont un processus se déploie sur le SoC, nous allons parler juste de l'allocation des pages des threads distants.

- Dans un processus, il y a un seul espace d'adressage virtuelle défini par une VSL et une GPT.
- la VSL et la GPT sont consultées à chaque miss-TLB et chaque page fault
  - ce n'est pas toujours vrai, un miss-TLB consulte la GPT, mais celle-ci peut être dans le cache L1 et la page accédée peut être déjà mappée, ce qui fait que l'on ne consulte pas les structures en mémoire.
- Il n'est pas possible d'imaginer que les VSL et GPT soient dans un seul cluster car cela provoquerait des contentions sur le banc mémoire du cluster contenant les structures.
- On va donc répliquer les structures VSL et GPT dans tous les clusters contenant des threads du processus.
- Les VSL et GPT répliquées sont créées vides.
- Lorsqu'il y a un page fault, le noyau consulte la VSL locale pour savoir si le segment est legal. S'il ne contient pas le segment, il consulte la VSL de référence, il met à jour la VSL local. Si le segment n'est pas dans la VSL de référence, c'est une erreur de segmentation et le processus est stoppé.
- Si le segment est légal (nouvellement ajouté ou pas), il consulte la GPT locale. Si la page virtuelle n'est pas encore mappée. Le moyau va la mapper différement en fonction du type de segment.
  - si c'est un segment de code, la page est répliquée et donc la GPT de référence et sa copie seront différentes pour cette page virtuelle.
  - si c'est un segment de data,
    - et qu'il y a déjà une page mappée, alors elle est utilisée, et donc la GPT de référence et sa copie seront identiques pour cette page virtuelle.
    - si la page n'est pas encore mappée, alors le noyau alloue la page quelque part sur le SoC et met à jour la GPT de référence et la copie locale.
  - Si c'est un segment de pile, le noyau alloue une page localement et met la jour la copie locale de la GPT.

- RÉFÉRENCE ET COPIE
  - Le principe, c'est les deux structures VSL et GPT ont autant de répliques (copie) qu'il y a de clusters possedant au moins thread du processus. Mais les copies ne sont pas identiques à la référence.
  - Les copies sont créées vides et elles se remplissent différemment en fonction du type de segment.
  - La gestion de la cohérence des copies est faites sur le modèle de celle faites par le cache L2. A savoir que c'est du write-through mais c'est plus complexe parce qu'il y a des cas qui n'existe pas dans les caches, comme les copies répliquées multi-valuées.
  - Ce mécanisme de réplication des structures VSL et GPT et des pages pour certains segments a l'avantage de réduire les sources de contention aux bancs mémoire, mais aussi d'augmenter la quantité de mémoire utilisable pour un processus.
  - En effet chaque thread à une pile et deux threads sur deux clusters différents peuvent utiliser les mêmes adresses virtuelles car elles sont mappées sur des pages physiques différentes.
  - Le coût à payer, c'est qu'il devient impossible d'échanger de l'information entre 2 threads par les piles, et il n'est pas possible, dans le cas général de faire migrer les threads.

Accès à la mémoire distante

- L'accès à la mémoire distante, c'est quand un processeur fait un accès mémoire dans autre cluster que le sien.

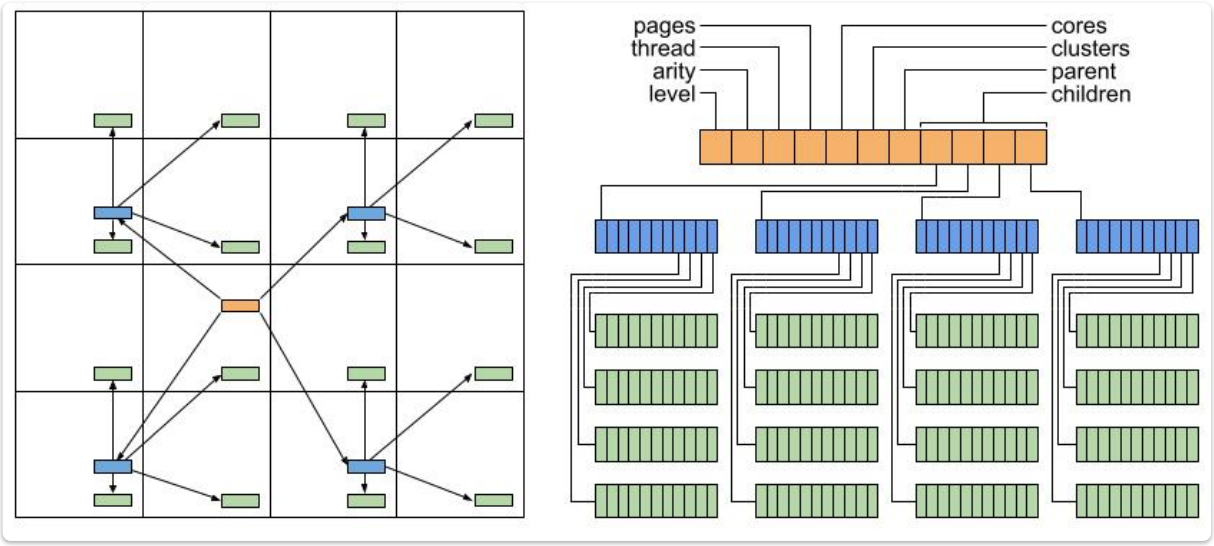
- Pour l'utilisateur, il n'y a pas de problème, c'est totalement transparent, la MMU traduit l'adresse virtuelle 32b en adresse physique 40b et le réseau se charge de l'acheminement de la requète vers le bon cluster et la bonne mémoire.
- Pour le noyau pour TSAR, les accès data se font grâce à l'extension d'adresse. Il faut donc passer par une API qui définit tous les accès remote. (load, store, atomic_add, etc.)
- Pour une machine INTEL 64 bits, cela n'a pas été implémenté, mais l'idée serait de laisser la MMU activé pour le noyau
    - Chaque processus disposerait de sa copie VSL-GPT par cluster, et les segment kernel sont mappé sur les pages physiques locale.
    - Mais, il y aurait en plus, autant de segments virtuels qui mapperaient toutes les pages physique de chaque cluster (→ faire un dessin)
    - l'accès distant passe toujours par une API, mais l'implémentation ne consiste pas à ajouter les bits d'exension, c'est seulement un calcul d'adresse virtuel pour aller vers le bon segment virtuel.

Choix du placement d'une ressource → DQDT

- Problème
    - Almos-mkh est constitué d'autant de kernels qu'il y a de clusters
    - Chaque cluster gère ses propres ressources : mémoire physique et processeur
        - La ressource mémoire est mesurée en pages physiques
        - La ressource processeur en nombre de threads
    - Almos-mkh doit répartir la charge sur l'ensemble des clusters pour augmenter le parallélisme.
    - Un kernel peut avoir besoin d'une ressource en dehors de son cluster, il doit donc avoir un état d'usage général des ressources
    - Il n'est pas souhaitable d'avoir un verrou global protégeant cet état, sinon deux threads souhaitant demander une ressource en même temps vont se séquencialiser alors qu'ils peuvent être sur des clusters distincts et que les ressources qui leur seront attribuées sont dans des clusters distincts aussi.
- idée
    - Créer une structure publique distribuée en copie unique de l'état des ressources régulièrement mise à jour, soit de manière périodique, soit au moment des allocations ou des libérations de ressources.
        - Publique parce que consultée par tous les threads de tous les processus
        - Distribuée (découpée en parties) pour éviter la contention sur un banc mémoire
        - En copie unique pour ne pas avoir à gérer la cohérence des copies
    - La modification de cette structure doit se faire sans verrou et donc chaque partie n'a qu'un propriétaire avec le droit de modification.
    - Cette structure n'a pas besoin d'être à jour. Elle donne juste une indication.
    - Lorsque qu'un thread veut allouer une ressource, il consulte la structure d'état des ressources en lecture seul et donc sans prendre de verrou et il obtient en réponse le numéro d'un cluster qui a peut-être la ressource demandée.
    - Le thread fait alors une demande ciblée au cluster désigné qui peut faire l'allocation ou la refuser.
    - Dans le cas d'un refus, le demandeur recommence la consultation de la structure d'état et demande une autre réponse, pour faire ensuite un nouvel essai. Si la ressource existe quelque-part, le demandeur finira par l'obtenir.

- On doit définir la charge des clusters ou des groupes de clusters pour les processeurs et pour la mémoire
  - *charge mémoire* = nombre de pages allouées / nombre de clusters dans le groupe
  - *charge processeur* = nombre de threads vivants / nombre de cores dans le groupe

- **Solution**



  - STRUCTURE DE DONNÉES
    - Arbre quaternaire recouvrant le mesh 2D de clusters
      - Le mesh est coupée en 4 quadrants.
      - Chaque quadrant contient l'état des ressources disponibles ou occupées
        - Nombre de cluster présents
        - Nombre de processeur (cores)
        - Nombre de threads
        - Nombre de pages physiques disponibles
      - Chaque quadrant est lui-même découpé en quadrant
    - Description
      - La racine de l'arbre contient la somme des ressources de tout le SoC
      - Le noeud de chaque quadrant est lié à son père (qui les rassemble)
      - Chaque quadrant est lui-même divisé en quadrant, de manière récursive.
      - Les feuilles de l'arbre sont sur les clusters
    - La profondeur de l'arbre dépend de la taille du mesh : cas pour 16 clusters (64 cores)
      - 1 cluster : 1 étage : 1 nœud level 0
      - 4 clusters : 2 étages : 1 nœud level 1 → 4 nœuds level0
      - 16 clusters : 3 étages : 1 nœud level 2 → 4 nœuds level 1 → 4 nœuds level 0
    - Distribution des nœuds dans les clusre
      - Chaque cluster contient 1 nœud level 0
      - Tous les 4 clusters contiennent 1 nœud level 1
      - Un cluster contient la racine
  - OPÉRATIONS
    - **initialisation**
      - Parcours récursif depuis la racine jusqu'au cluster avec initialisation lors de la remontée
    - **Consultation** pour trouver un cluster disposant d'une page ou d'un processeur - Parcours depuis la racine pour une répartition homogène - Est-ce que quadrant contient la ressource - Non → sortir - Oui et ce n'est pas le level 0 - Consulter les 4 fils à la recherche du moins chargé et boucler récursivement - Oui et c'est le level 0 → sortir avec le numéro du cluster - Parcours depuis le cluster local pour renforcer la localité -

Est-ce que le quadrant local contient la ressource - Oui → faire une recherche à partir de cette racine - Non → remonter d'un niveau et boucler récursivement

- **Mise à jour**
  - Au plus tôt, dès qu'une ressource est allouée ou libérée, on met à jour les compteurs avec un atomic add
  - Ou périodiquement, chaque noeud consulte ces 4 fils pour se mettre à jour, pas de verrou mais il y a un risque que l'information soit fausse.
  - Dans la pratique, l'allocation ou la libération d'une ressource est une opération couteuse mais rare. La mise à jour au plus tôt coûte au pire 6 atomic adds, par contre la racine est très accédée.

Démarrage d'almos-mkh

Introduction

- La procédure de démarrage, c'est ce qui se passe entre le signal de reset et le démarrage du premier processus utilisateur.
- A l'issue de cette procédure, le matériel est initialisée, les structures du noyaux sont en place, et le premier processus 0 est créé et démarré.
- Dans le cas d'almos-mkh, il faut préparer la mémoire de manière assez spécifique en raison du découpage en cluster.
- En particulier, chaque cluster dispose d'une copie publique, localisée et répliquée à l'identique du code du noyau et les variables globales sont publique, localisée et répliquée multi-valuée.
- Il y a trois étapes :
  - Exécution du **preloader**, c'est un code en ROM, indépendant de l'OS et du système de fichier, dont le rôle principal est de charger et démarrer le **bootloader**
  - Exécution du **bootloader**, ce code est spécifique à l'OS et au système de fichier, dont le rôle principal est de charger le noyau du système d'exploitation
  - Exécution de la fonction d'entrée du système d'exploitation **kernel_init()**