Process et Thread

Objectifs

- Comprendre ce qu'est un process et un thread pour Almos-mkh et l'implémentation inter-cluster
- Présenter la structure **cluster manager**
- Présenter les structures de représentation de l'espace virtuel
- Présenter comment les threads puis les process naissent et meurt

Plan

- Process et Thread
- Remote Procedure Call
- Création et desctruction d'un thread
- Création et destruction d'un process

Process et Thread

Process

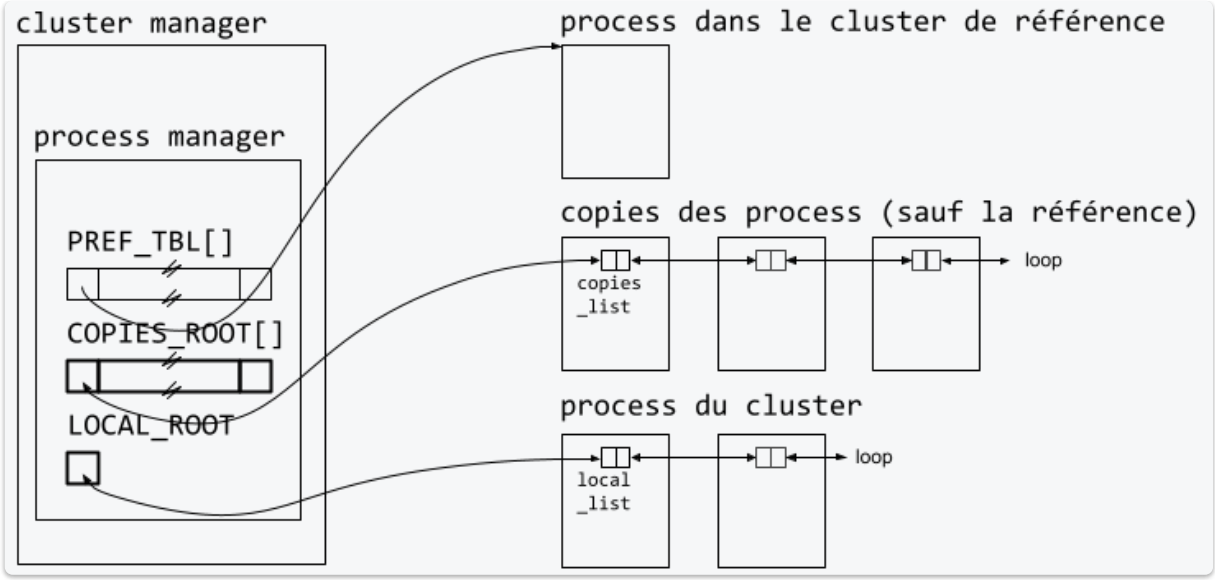
- Un process est le conteneur des ressources nécessaires à l'exécution d'un programme
- Un process est défini par
  - un espace d'adressage virtuel
  - un ensemble de fichiers et de sockets ouverts
  - une console
  - une liste de threads
- Almos-mkh est multi-kernel
  - un process peut avoir plus threads dans plusieurs clusters
  - la structure **process** est distribuée et parttiellement répliquée dans tous les clusters qui contiennent au moins un thread du process
- Un process est identifié par un PID sur 32 bits (Process IDentifier)
  - 16 bits MSB = Coordonnées X, Y du cluster owner (X et Y sont chacun sur 1 octets)
  - 16 bits LSB = LPID (Local PID) numéro dans le cluster owner
  - Cette manière d'identifier les ressources est utilisée pour d'autres stuctures par elle permet de savoir directement quel cluster interroger pour accéder aux ressources
- Pour un process on distingue 3 types de cluster
  - le **cluster owner** qui est le cluster de naissance du process
  - le **cluster de référence** qui contient la structure de référence du process
  - les **clusters de copies** qui contiennent au moins 1 thread et qui ne sont pas le cluster de référence

Les différents clusters

- Le cluster owner est le cluster de naissance du process.
- Quand on connait le PID, on connait le cluster owner
- Chaque cluster possède une structure regroupant les threads locaux
- Le cluster de référence contient toutes les structures du process (les plus complète), c'est là que s'exécute le thread main()
- Les clusters de copies contiennent les stuctures du process, mais incomplètes.
- Au départ, le cluster de référence est le cluster owner, mais en cas de migration, le cluster owner ne change pas, seul le cluster de référence migre.
- Le cluste owner sait toujours où se trouve le cluster de référence.

Cluster manager

- dans chaque cluster, il existe une variable gloabale qui référence toutes les ressources du cluster
- Paramètres globaux sur la plateforme
  - taille
- Paramètres locaux
  - nombre de core
  - taille de la mémoire
- structures pour la gestion des cores
  - scheduler
- liste des devices du clusters
- structures de la mémoire
  - descripteur de pages physiqyes
  - allocateurs
- structure pour les RPD
  - fifos
  - threads RPC
- DQDT pour connaitre l'usages de certaines ressources de la plateforme
- Process manager
  - **PRED_TBL[LPID]** : tableau des pointeurs sur les process appartenants au cluster, indéxé par LPID
  - **COPIES_ROOT[LPID]** : tableau des copies pour chaque process (liste interclusters)
  - **local_root** : liste des process du cluster



Structure Process

- PID
- PPID : parent
- PREF : xptr sur process de référence
- VSL : Virtual Segment List
- GPT : Generic Page Table
- FDT : File descriptor
- TH_TBL : thread du cluster
- local_list : chainage des copies du process dans le cluster
- copies_list : chainages de toutes copies du process
- chidren_root : racine de la liste des process enfants
- children_list : chainage des enfants du process

Structure thread

- Présentation
  - 4 types de threads : USR, DEV, RPC, IDL
  - Identifié par un TRDID
    - 16 bits MSB = Coordonnées X, Y du cluster owner (X et Y sont chacun sur 1 octets)
    - 16 bits LSB = LTID (Local PID) numéro dans le cluster
  - les threads sont fixes (ils ne migrent pas)
  - Un process s'exécutant sur plusieurs clusters possède une copie de la structure process qui liste les threads locaux
- TRDID
- TYPE : USR, DEV, RPC, IDL
- FLAGS : attributs (detached, join done, kill done)
- BLOCKED : bits de blocage
- PROCESS : process local
- CORE
- CPU & FPU contextes
- XLIST : chainage des threads en attente sur une ressource
- chidren_root : racine de la liste des threads enfants
- children_list : chainage des enfants du threads
- locks count

Création d'un process

- Pour créer un process on utilise **fork()** et **exec()**
- **fork()** duplique le process, le nouveau process hérite de VSL, GPT, FDT et de la console
- **exec()** remplace l'application courante par une nouvelle mais hérite de FDT et de la console

fork()

- Le nouveau process est créé dans un autre cluster - désigné par la DQDT - ou choisi explicitement - Ce choix induit le nom PID
- Le nouveau process hérite d'une copie du thread qui réalise le **fork()** ainsi que son contexte (CPU & FPU) pour qu'il puisse se réveiller sur la bonne ligne du code avec le même état
- Le nouveau processs avec son thread a un nouvel espace d'adressage et le threads conservé utilise l'une des piles.
- le nouveau process n'a qu'un seul thread
- le nouveau process n'a pas une copie fidèle des VSEG et GPT, cela dépend de leur type
  - DATA, MMAP, REMOTE (partagé et non répliqué) sont copiés à l'identique mais les pages référencées sont **no-writable**, la page est tagée **COW** (copy on write) pour le père et le fils, et il y a un compteur de référence dans le descripteur de la page.
  - STACK : seule la pile du thread qui a fait le fork est conservée mais les pages sont **no-writable** et **COW**
  - CODE : le segment est copié mais pas les pages pour les mappée localement à la demande
  - FILE : copié à l'identique

exec()

- **exec()** permet d'exécuter un nouveau programme qui hérite du PID de son créateur, de ses fichiers ouverts et de son environnement
- On commence par
  - détruire tous les threads de toutes les copies du process
  - libérer (délivrer :) tous les segments de la VSL et les GPT pour aller chercher le nouveau code pour un nouveau thread

Création d'un thread

- La difficulté est lorsqu'on crée un thread sur un nouveau cluster.

1. envoyer un RPC_THREAD_USER_CREATE vers un cluster
   - arg pthread_attr_t
   - res TRDID
2. le cluster solliciter exécute **cluster_get_reference_from_pid()**
3. Création d'une copie du process avec **remote_mem_copy()**
   - allocatoin VSL, GPT, FDT
   - GPT et VSLI sont vides remplies à la demande lors des pages faults
4. Création du nouveau thread sur un des cores pour une exécution à un prochain ordonnancement

Remote Procedure Call RPC

Idée

- La structure multi-kernel du noyau permet de rendre explicite les accès aux structures locales au cluster et les accès aux structures distantes.

- Accéder aux ressources distantes signifie
  - prendre la propriété par un verrou pour la modifier
  - la parcourir pour la lire
- Ces deux services posent problème
  - Si c'est une ressource très partagée, les spinlock coutent cher, même si l'attente est caché, il y a des avalanche de lecture à chaque changement d'état du verrou
  - Quand on parcourt une ressource distante
    - on cree un trafic sur le NOC
    - on polue le cache locale avec des lignes lues une seule fois
- Une solution consiste à proposer un mécanisme d'exécution de service à distance en mode client-serveur.
- Pour accéder à une ressource distante, le noyau peut :
  - Accéder directement avec les accès remote (en prenant des verrous)
  - envoyer une demande de service
    - Chaque cluster est propriétaire de ses structures (il est le serveur)
    - le cache des clients n'est pas polué
    - le trafic sur le NoC est réduit donc la consommation aussi
    - mail le round trip est long...
- L'OS doit choisir pour chaque structure le mode d'accès et c'est le H de MKH pour hybride

Fonctionnement

- Chaque noyau est propriétaire de ses ressources
- Quand un thread veut allouer ou modifier une ressource dans un autre cluster il fait une RPC
- 2 type de RPC
  - simple : un client un serveur
  - parallel : un client demande à plusieurs serveurs
  - Le client décrit une requete chez-lui et poste l'adresse dans la fifo de requete du serveur
  - Le serveur
  - Thread_client → requete RPC → thread_serveur
  - Thread serveur → reponse