Besoins

- Booter en utilisant un espace mémoire indépendant du nombre de processeurs.
- Booter sans connaitre à priori le nombre de processeurs.
- Booter sans faire d'assertions sur l'état initial de la mémoire.

Solutions

1. Un processeur est élu pour initialiser la mémoire, il débloque les autres processeurs pour la suite après l'initialisation de certaines cases
2. Tous les processeurs incrémentent atomiquement une variable globale comptant les processeurs de la plateforme

Déroulement

`asm_magic_val`
: Une case mémoire contenant à priori n'importe quoi, devant contenir MAGIC1 pour élire un chef, et MAGIC2 pour débloquer tout le mode

`start_bar`
: Une case mémoire contenant un booléen "doit démarrer", modifié par le chef, lu par tous

`cpu_count`
: Un compteur atomique contenant le nombre de procs sur la plateforme

`bs_cpuid`
: Globale contenant le CpuId? du chef

- Le chef:
  - Lire @`asm_magic_val`
  - Réussir à écrire atomiquement MAGIC1 -> @`asm_magic_val`
  - Init 0 -> @`start_bar`
  - Init 1 -> @`cpu_count`
  - Init cpuid() -> @`bs_cpuid`
  - Écrire MAGIC2 -> @`asm_magic_val`
  - Saut dans arch_init()
- Les autres:
  - Lire @`asm_magic_val`
  - Ne pas réussir à écrire atomiqumente MAGIC1 dans @`asm_magic_val`
  - Attendre MAGIC2 == @`asm_magic_val`
  - Incrément atomique @`cpu_count`
  - Attendre 1 @`start_bar`
  - Saut dans `stack_poll_call(arch_init_stack_pool)`

Dans `arch_init()`

- Le chef:
  - Initialise la mémoire
  - Initialise son CPU
  - Initialise les globales
  - Initialise les Pool d'allocateurs
  - Initialise le scheduler
  - Initialise la globale `arch_init_stack_pool` pour permettre aux autres CPU de démarrer
  - Initialise `running_count` = 1

        - cpu_start_other()
        - Attend `running_count` == @`cpu_count`
    - Les autres:
        - Initialise son CPU
        - Initialise son sched
        - Incrémente atomiquement `running_count`
        - Termine avec stack_poll_terminate_call(stack_poll_call, start_stack_pool)

`cpu_start_other()`
: Écrit 1 -> @`start_bar`

APIs

stack_pool

- Bas niveau:

`stack_poll_initialize(struct stack_pool_s *pool, func_t *func)`
: Initialise la stack_pool devant appeler une fonction `func`.

`stack_poll_newstack(struct stack_pool_s *pool, void *stack, size_t size)`
: Donne une nouvelle pile dans le pool

`void *stack_poll_getstack(struct stack_pool_s *pool)`
: Récupère la première stack libre dans le pool

- User:

`stack_pool_call(struct stack_pool_s *pool)`
: Utilise une pile dans l'ensemble de piles disponibles dans `pool` pour exécuter la fonction `c_function`. Le stack-pointer à l'entrée de cette fonction ne sert à rien, il peut être invalide.

`stack_pool_terminate_call(func_t *stackless_func, void *arg)`
: Relâche la pile partagée utilisée par le CPU courrant et saute dans la fonction `stackless_func(arg)`. Cette fonction ne doit pas utiliser la pile et peut recevoir un stack pointer invalide.